

# Gitを用いたバージョンコントロール

池田騰, 木村有祐, 宮崎清人, 吉井英人

# 本日の流れ

(1) Gitの基本概念編(担当: 木村)

Gitの基本概念を学ぶ

(2) オブジェクトストア編(担当: 宮崎)

オブジェクトストアについて学ぶ

(3) リポジトリ編(担当: 池田)

リポジトリについて学ぶ

(4) Git実践編(担当: 吉井)

実際のコマンドを用いて実践する

(5) Git練習編(担当: B4)

実際にGitを用いて学ぶ

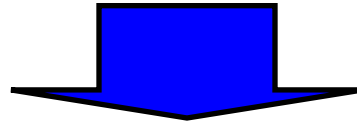
# Git勉強会用スライド (Gitの基本概念編)

木村有祐

# バージョン管理システムとは

計算機上のデータは簡単に上書きでき、元の状態がわからなくなる

➡ ある時点ごとの状態をバックアップしながら開発する



ソフトウェアの異なるバージョンを管理し追跡するツールを利用

バージョン管理システムを使う利点

- (1) 変更履歴がすぐに確認可能なため、異常の原因解明が容易
- (2) リリース版や開発版といった目的に応じた状態管理が可能
- (3) 集団で開発するときに、個人が行った作業が明確

# バージョン管理

ファイル  
A-0

ファイル  
B-0

ファイル  
C-0



編集中

# バージョン管理

コピー

バージョン1

ファイル  
A-0

ファイル  
B-0

ファイル  
C-0

.zip

ファイル  
A-0

ファイル  
B-0

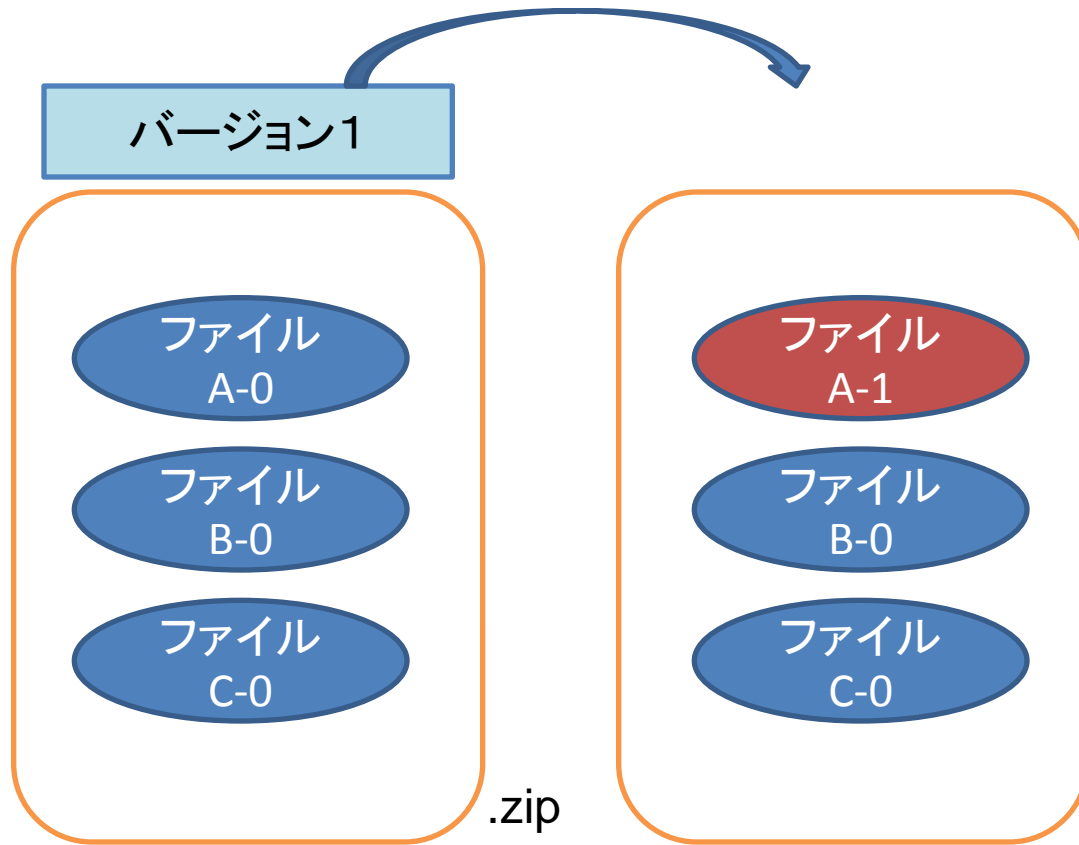
ファイル  
C-0



編集中

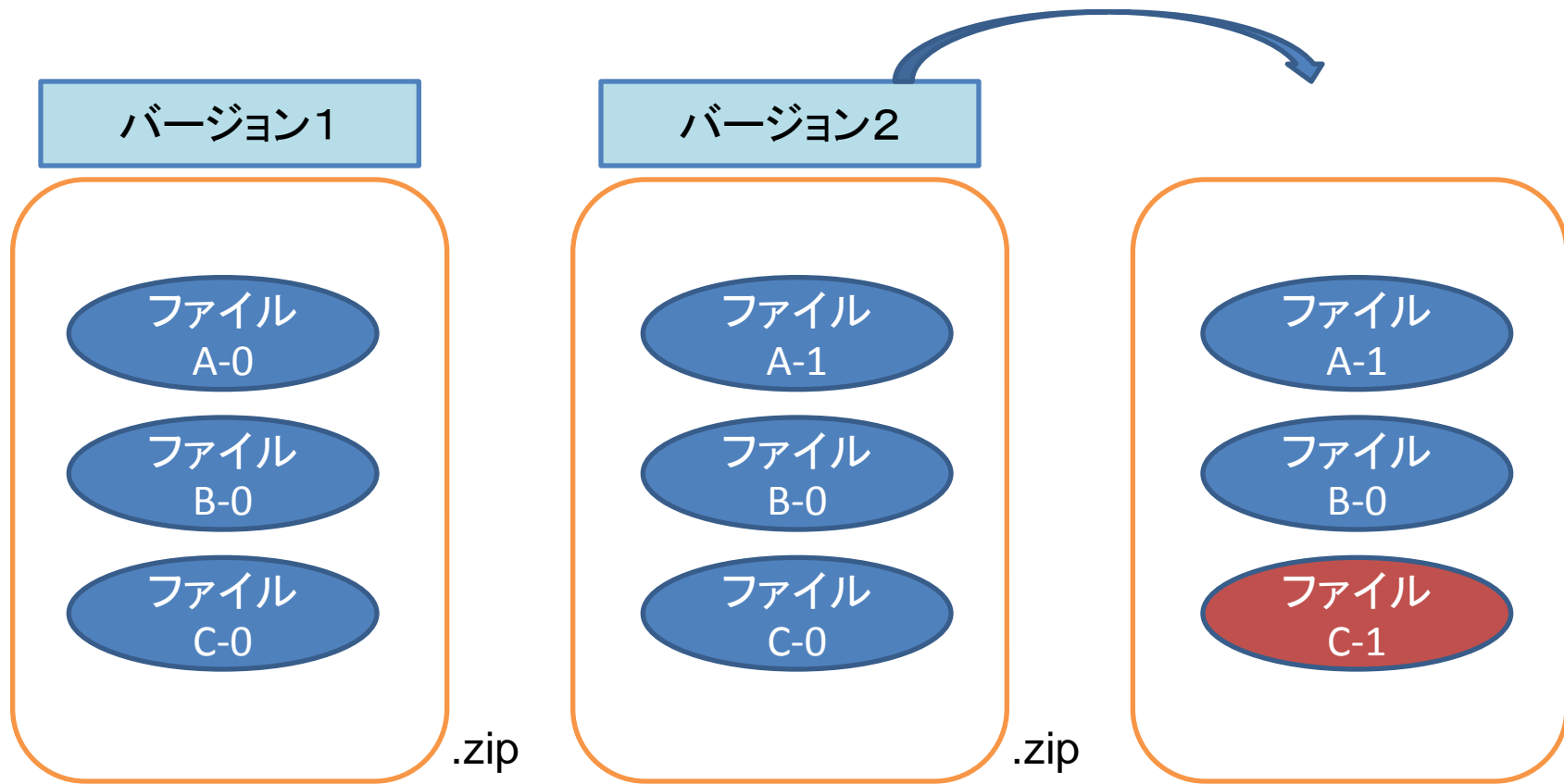
# バージョン管理

コピー



# バージョン管理

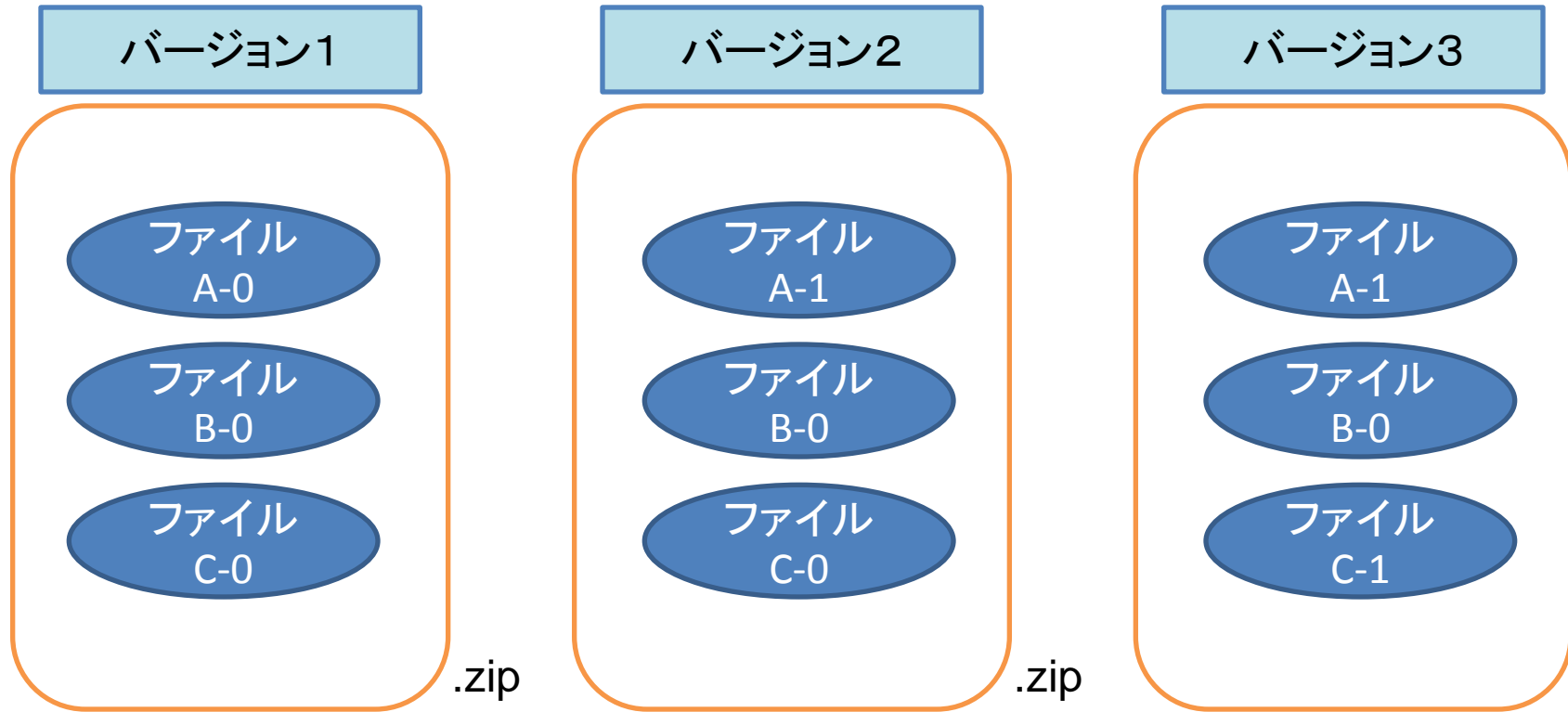
コピー



編集中



# バージョン管理



# バージョン管理システムの用語

## (1) リポジトリ[収納場所]

ファイルやディレクトリのバージョンに関する変更内容などを保持するデータ格納庫

## (2) ブランチ[木の枝]

メインの開発ラインと分けて、独自の開発を行うときに作るメインから分岐した開発ライン

## (3) マージ[合流]

2つ以上の開発ラインを統合する作業

## (4) コンフリクト[競合]

マージのときに、同じ箇所に違う変更を加えていて、マージ後の適切な状態が決まらないこと

# バージョン管理システムGit

<Gitとは>

Linuxの生みの親リーナス・トーバルズによって開発された  
オープンソースのバージョン管理システム

Gitによって開発が管理されている有名なソフトウェア

- (1) Linuxカーネル
- (2) Ruby on Rails
- (3) Git

乃村研究室内での利用

- (1) 乃村研ホームページ(ノムニチ)
- (2) TwinOS(NewOS)
- (3) LastNote

# Gitリポジトリ

GitはGitリポジトリを用いてバージョン管理を行う

## <Gitリポジトリ>

バージョン管理を行う上で必要な全情報を持つ

### (1) オブジェクト格納領域(object store)

(a) 施されたすべての変更を追跡する場所

(b) 元のデータファイル, 全ログメッセージ, 作成者情報, 日付 等の様々な情報を持つ

### (2) インデックス(index)

(a) リポジトリ全体のディレクトリ構造が記述された, 一時的かつ動的なバイナリファイル

# オブジェクト格納領域(object store)

Gitのオブジェクトストアは4種類のオブジェクトから構成

## <blob>

(1) ファイルデータを格納

## <tree>

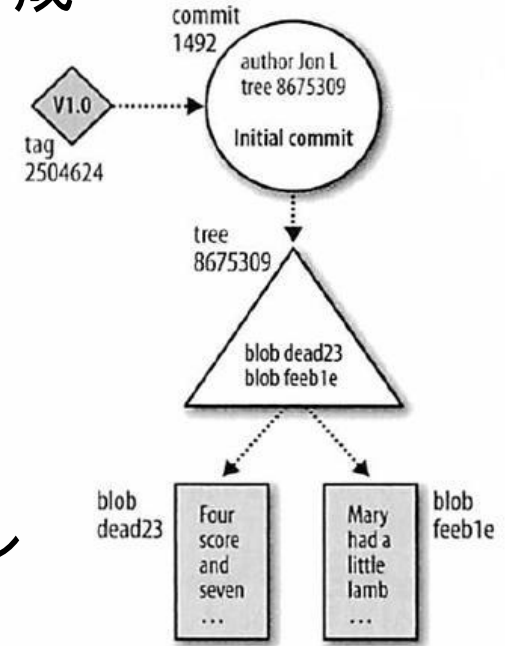
- (1) 1つ以上の“blob”オブジェクトを参照し、ディレクトリ構成を作成
- (2) treeは他のtreeを参照し、ディレクトリ階層を作成

## <commit>

- (1) 特定バージョンの情報を含んだオブジェクト
- (2) コミットした人物, 日付などの情報を保有
- (3) “tree”オブジェクトを参照

## <tag>

- (1) オブジェクトに署名をつける目的で利用
- (2) 主に“commit”オブジェクトを特定するシンボル
- (3) 任意に作成可能



リポジトリの変更とともにオブジェクトの情報は変化

# オブジェクトの特定

Gitは各オブジェクトに対してSHA1のハッシュ値を付与

## <SHA1>

40桁の16進数であらわされるユニークなハッシュ値

例) lfb58b4l53e90eda08c2b022ee32d9072958

➡ 正確にオブジェクトの1つを指定可能

## <簡略化>

人間はハッシュ値を覚えにくく、間違いが起こりやすい

➡ 2つの方法で簡略化が可能

(1) そのIDだと判別できる範囲で省略可能

例 : lfb58

(2) タグ名を使った指定

例 : v1.0

# コミット(Commit)

リポジトリへの変更を記録するためのもの

## <commitオブジェクトが持っている要素>

- (1) リポジトリ内のルートにあたる“tree”オブジェクトのハッシュ値
- (2) commitした人物
- (3) commitした時刻
- (4) commitした理由 (commitメッセージ)
- (5) parent commit (ひとつ前のcommit object) のハッシュ値

 コミットの履歴を参照可能

## <commit ID>

commitオブジェクトのSHA1のハッシュ値

# コミットグラフ(Aをコミット)

masterブランチ



コミットA

<ブランチ>

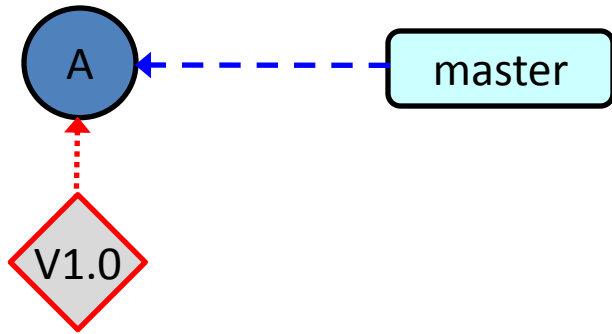
- (1) 開発ラインの名前
- (2) 常に最新コミットを参照

<masterブランチ>

- (1) Gitは最初のブランチの名前を“**master**”とする
- (2) masterブランチに特別な機能はない
- (3) 開発者は、リポジトリ内で、最も堅牢で、信頼できる開発ラインとすることに努める



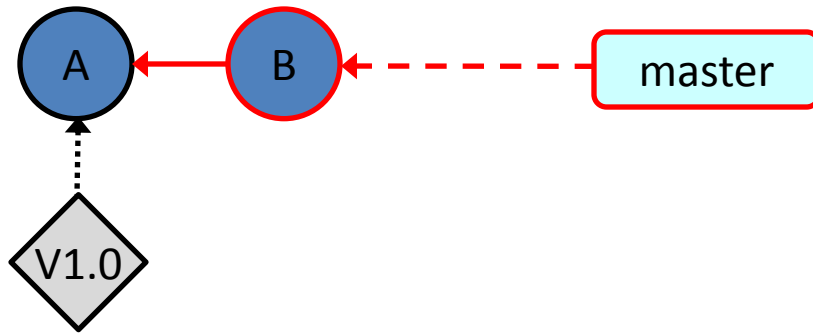
# コミットグラフ(タグの指定)



コミットAに“**V1.0**”タグを指定

タグV1.0がコミットAを参照

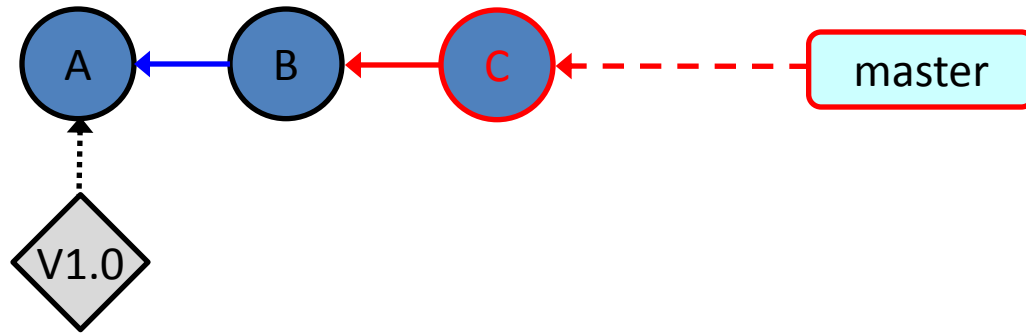
# コミットグラフ(Bをコミット)



コミットBを追加

“V1.0”タグはコミットAを参照したまま, masterブランチは最新コミットのコミットBを参照

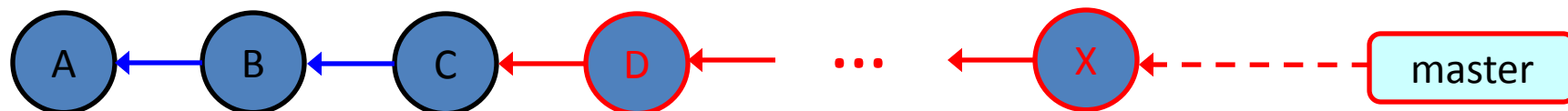
# コミットグラフ(Cをコミット)



コミットCを追加

“V1.0”タグはコミットAを参照したまま、masterブランチは最新コミットのコミットCを参照

# 開発ラインの分離要求



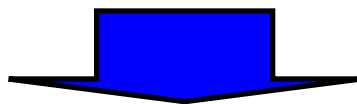
## <疑問>

1つのリポジトリで1つの開発ラインしか作れないのか？

## <要求>

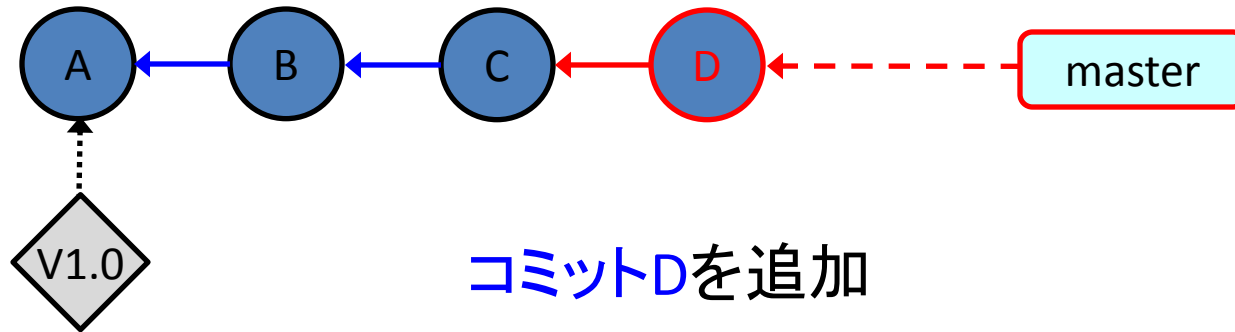
開発時、現在の開発ラインと分離したいという要求が出る

例：運用用の開発ラインと機能開発用の開発ラインを分離

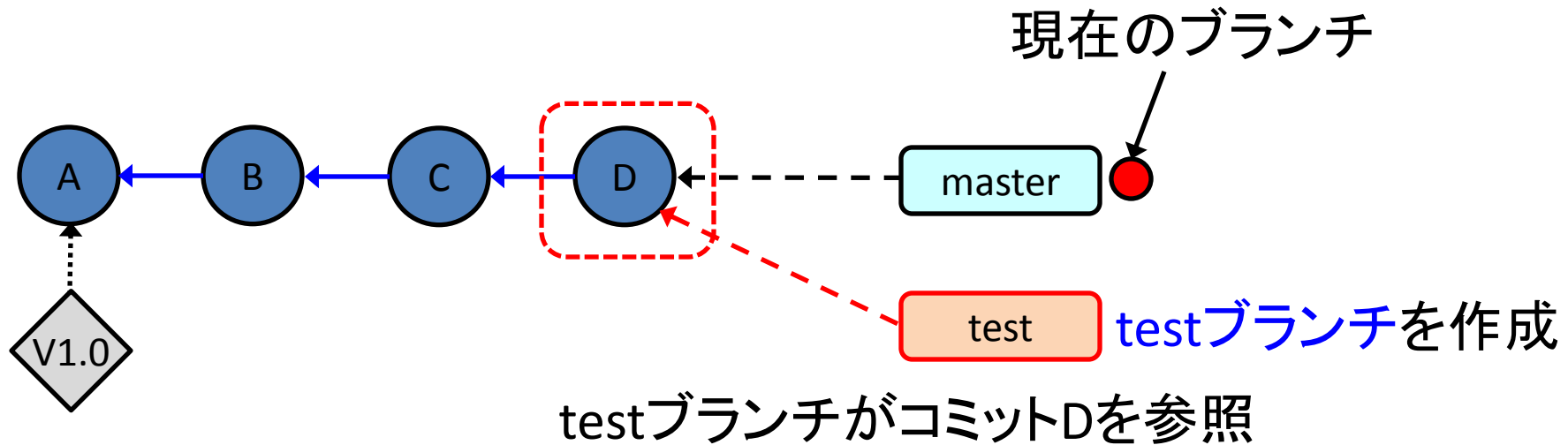


ブランチを利用

# コミットグラフ(Dをコミット)



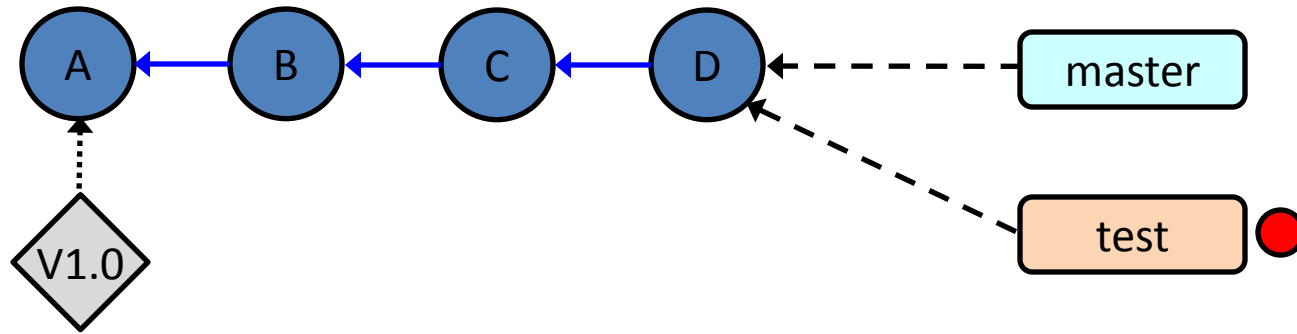
# ブランチの作成



## <ブランチの作成(ブランチをきる)>

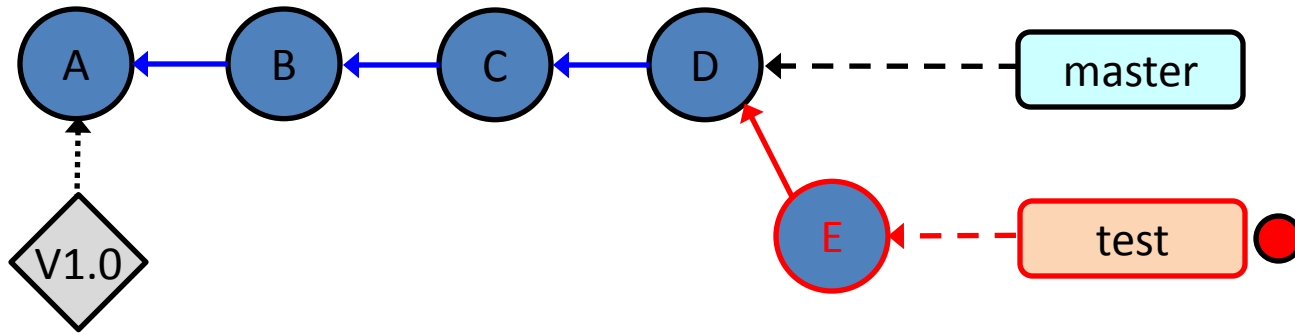
- (1) 現在のブランチの最新コミットからブランチを作成可能
- (2) 1つのリポジトリ内では, 1つのブランチでのみ作業可能
- (3) ブランチは切り替え可能(チェックアウト)

# testブランチへチェックアウト



testブランチで作業するため, testブランチへチェックアウト

# testブランチでのコミット



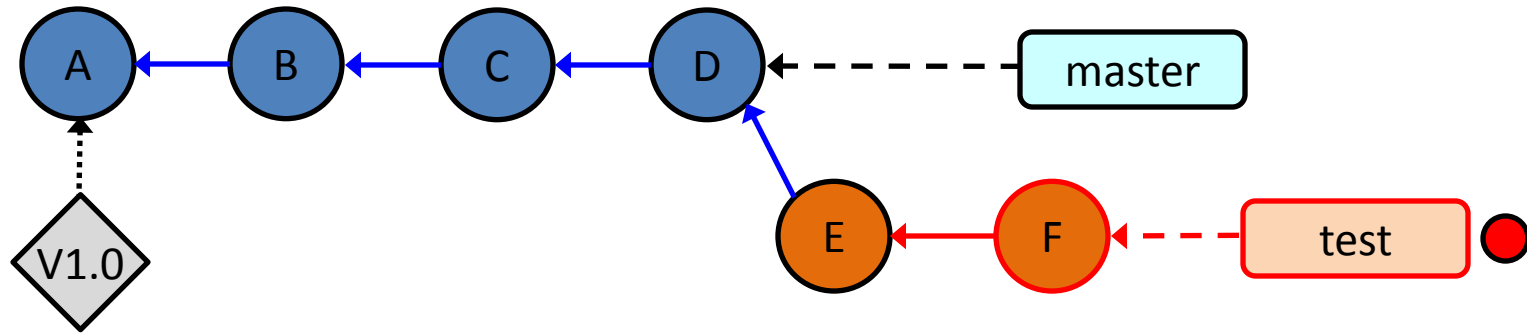
testブランチでコミットEを追加

testブランチはコミットEを参照

testブランチのコミットは, masterブランチに影響しない



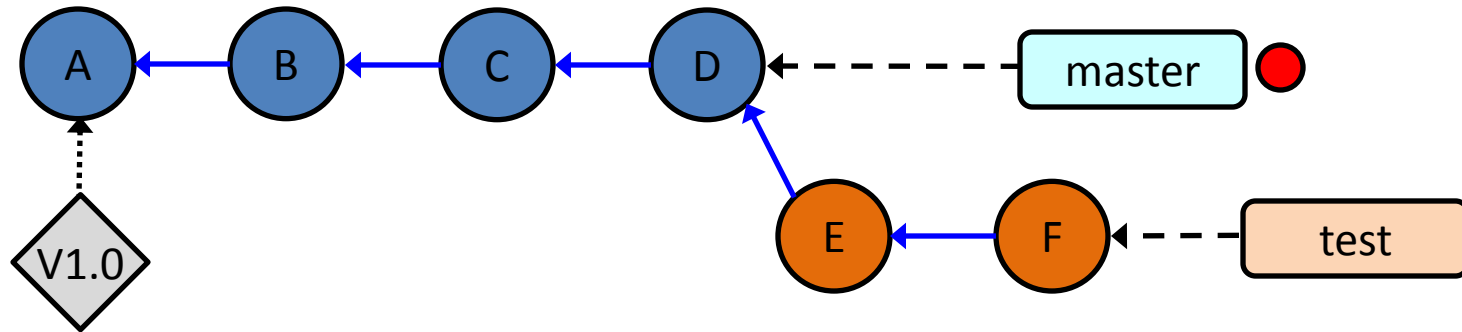
# ブランチでの作業



testブランチでコミットFを追加

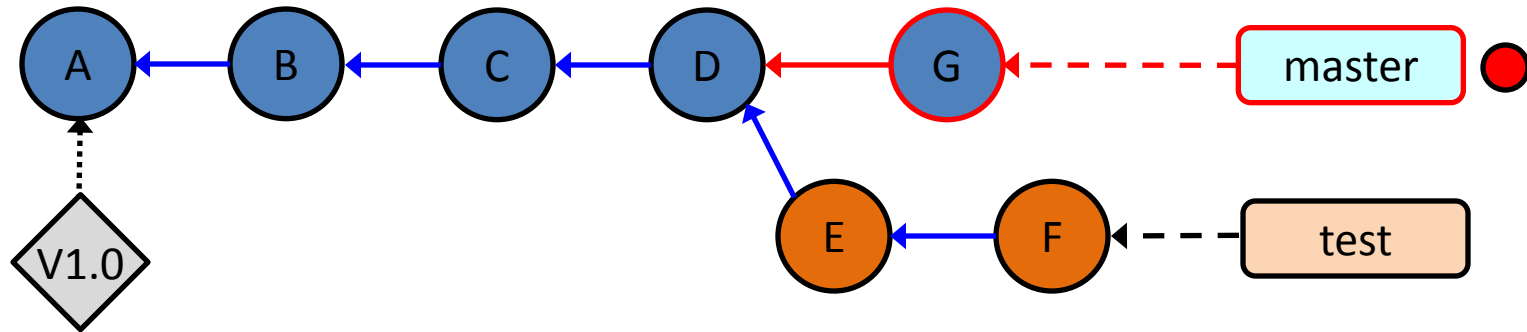
testブランチはコミットFを参照

# masterブランチへチェックアウト



masterブランチで作業するため, masterブランチへチェックアウト

# 再びmasterブランチの作業

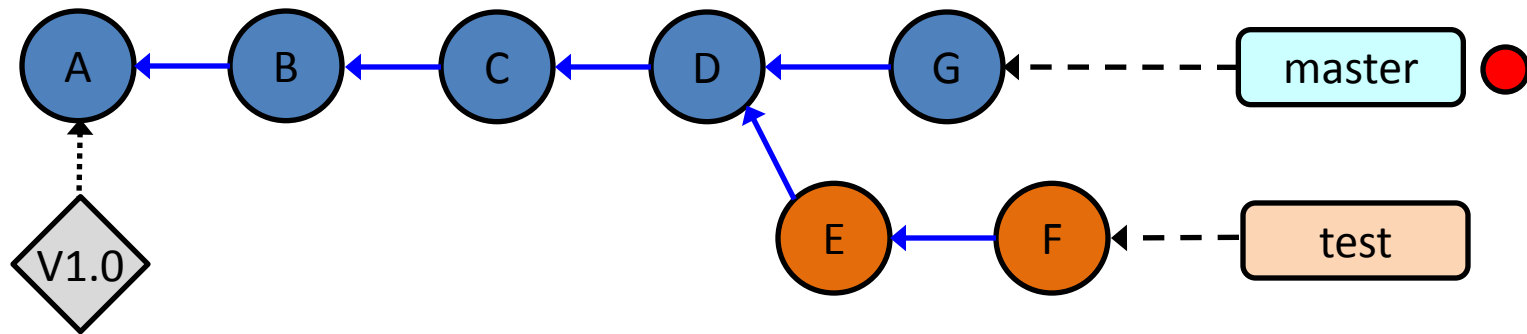


masterブランチでコミットGを追加

masterブランチはコミットGを参照

masterブランチのコミットは、testブランチに影響しない

# 開発ラインの統合要求



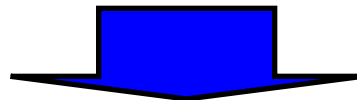
## <疑問>

testブランチのコミットE, Fは, masterブランチで使えないのか？

## <要求>

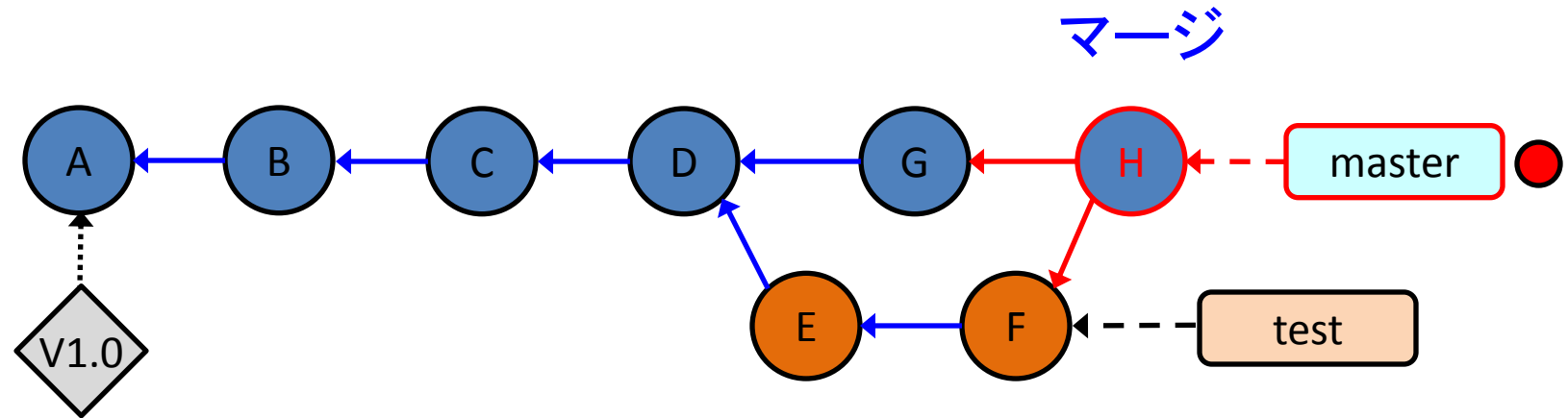
現在のブランチと違うブランチを統合したいという要求が出る

例: 新しい機能を作成したので, 開発ラインに統合



**ブランチのマージを利用**

# マージ



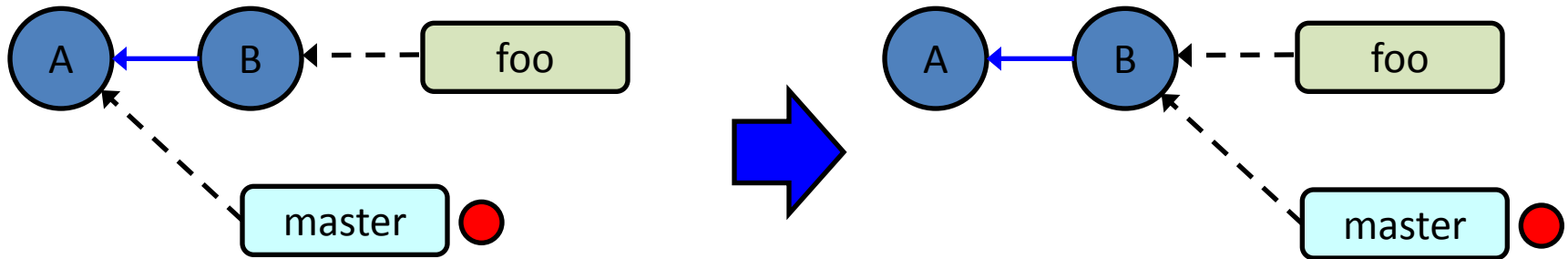
masterブランチにtestブランチをマージ

## <マージ>

コミット履歴を持つ, 2つ以上のブランチの統合

コミットHには「コミットEとFをマージした」というコミットができる

# 特殊なマージ



masterブランチにfooブランチをマージ

(1) masterブランチが参照する場所がBになる

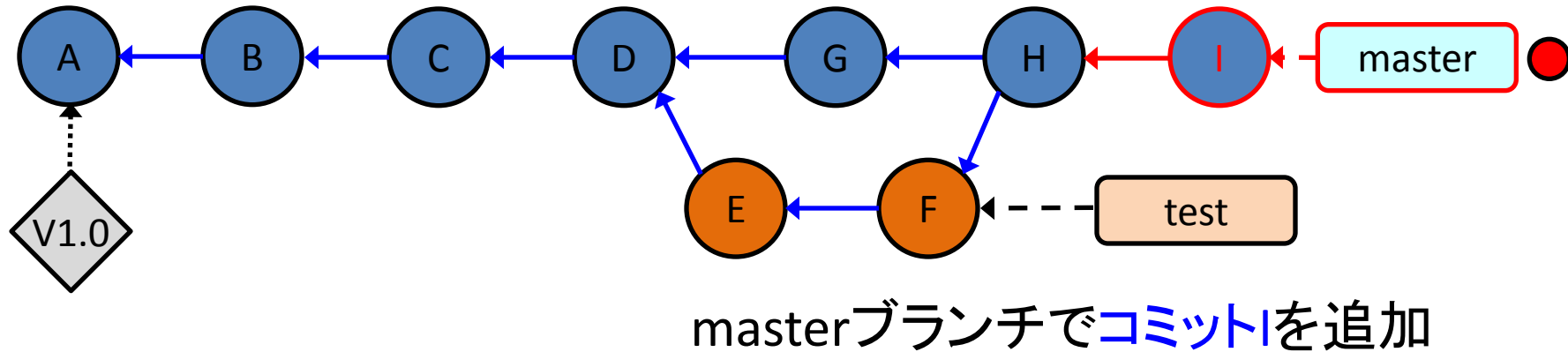
(2) マージしたというコミットはできない

## <注意点>

マージ先ブランチ(foo)の全てのコミット(A, B)がマージ元ブランチ(master)の全てのコミット(A)を含む場合[ファストフォワード]

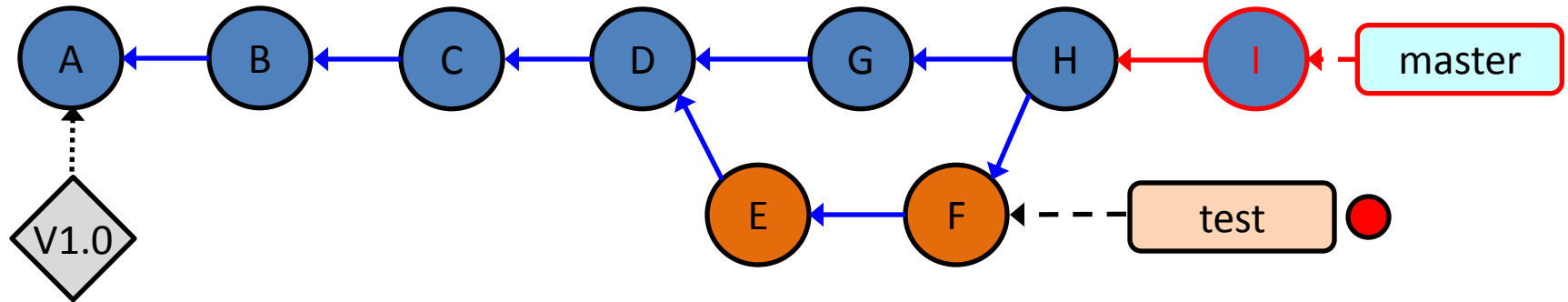
➡ コミットはできず, ブランチの参照する場所が変わる

# マージ後のmasterブランチ



masterブランチはコミットIを参照  
testブランチはコミットFを参照

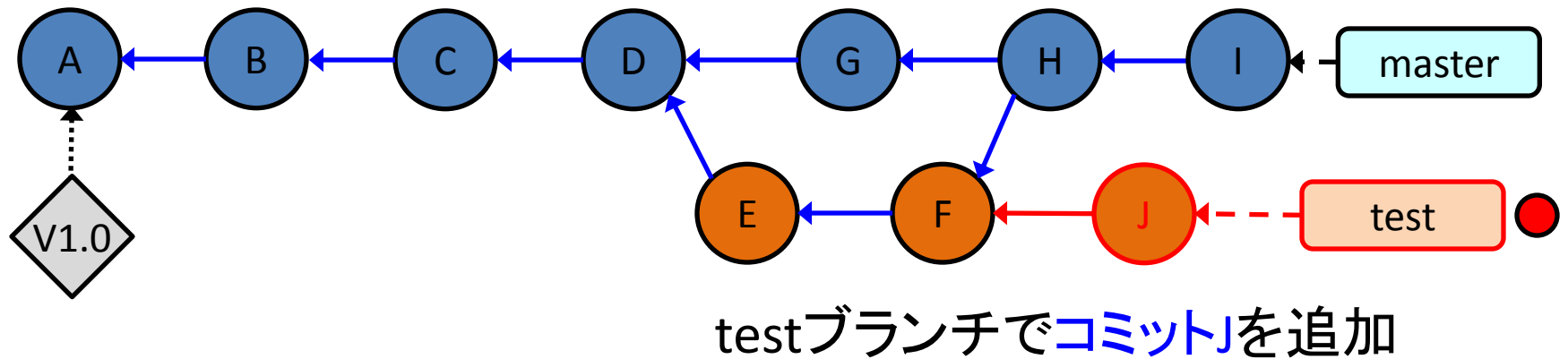
# testブランチへチェックアウト



testブランチで作業するため, testブランチへチェックアウト



# マージ後のtestブランチ



testブランチはコミットJを参照  
masterブランチはコミットIを参照

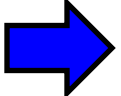
# シンボリック参照

## <シンボリック参照(symrefs)>

Gitは、4つの特別な参照用のシンボル(symrefs)を用意している

### (1) HEAD

(A) 現在のブランチを参照

 現在のブランチの最新コミットを参照

(B) ブランチを切り替えると、切り替えたブランチを参照

 切り替えたブランチの最新コミットを参照

(2) ORIG\_HEAD

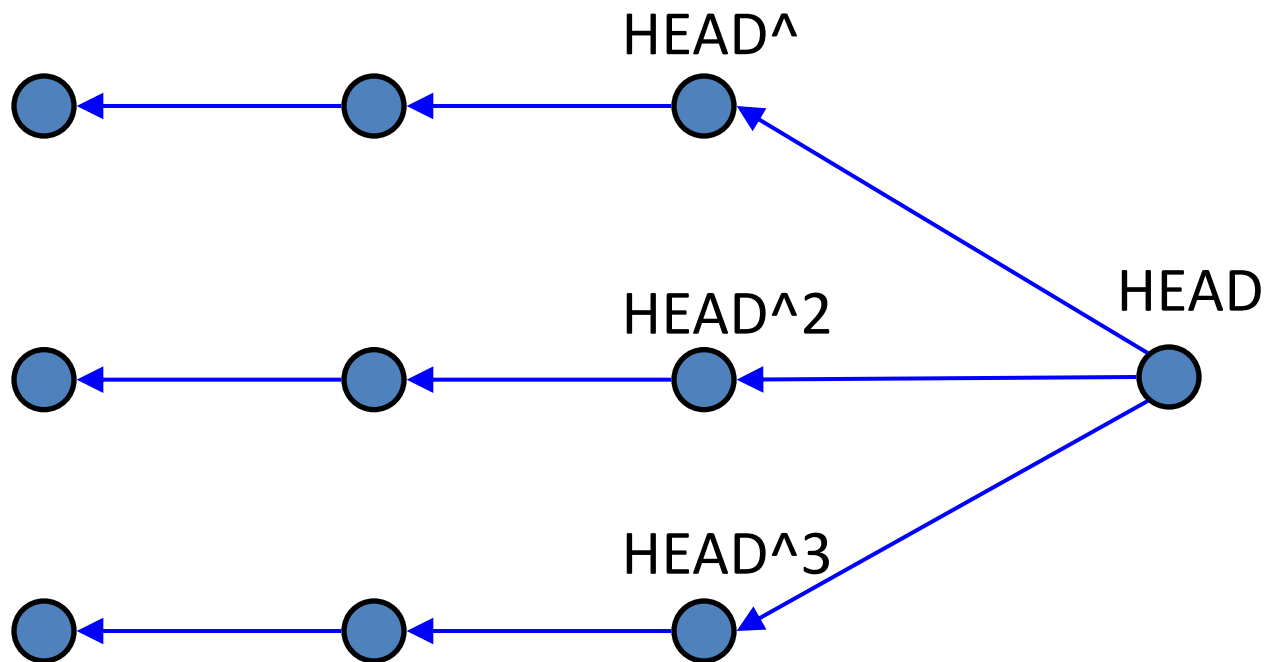
(3) FETCH\_HEAD

(4) MERGE\_HEAD

# 相対的なコミット名(^)

commit objectの関係を使って, commit objectを指定可能

例 : commit objectに関連がある場合



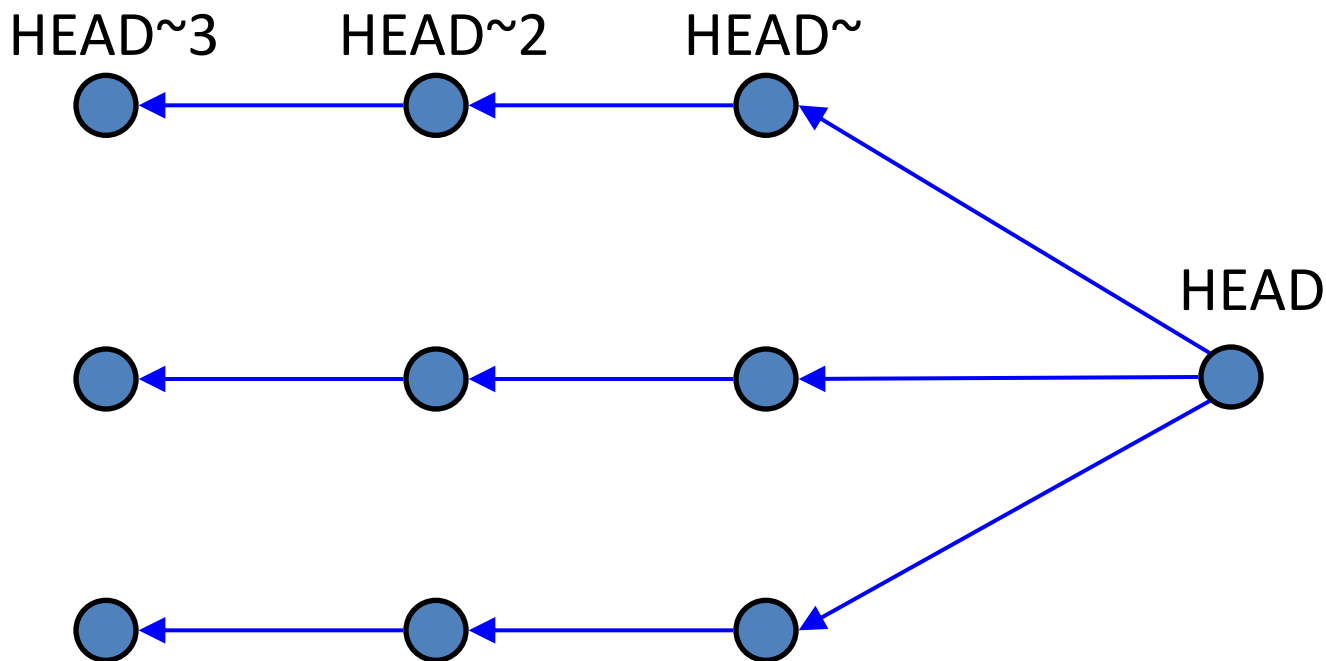
^ :parent commit objectを指定する記号

^を使うと, それぞれのコミットを上記のように指定可能

# 相対的なコミット名(~)

commit objectの関係を使って, commit objectを指定可能

例 : commit objectに関連がある場合



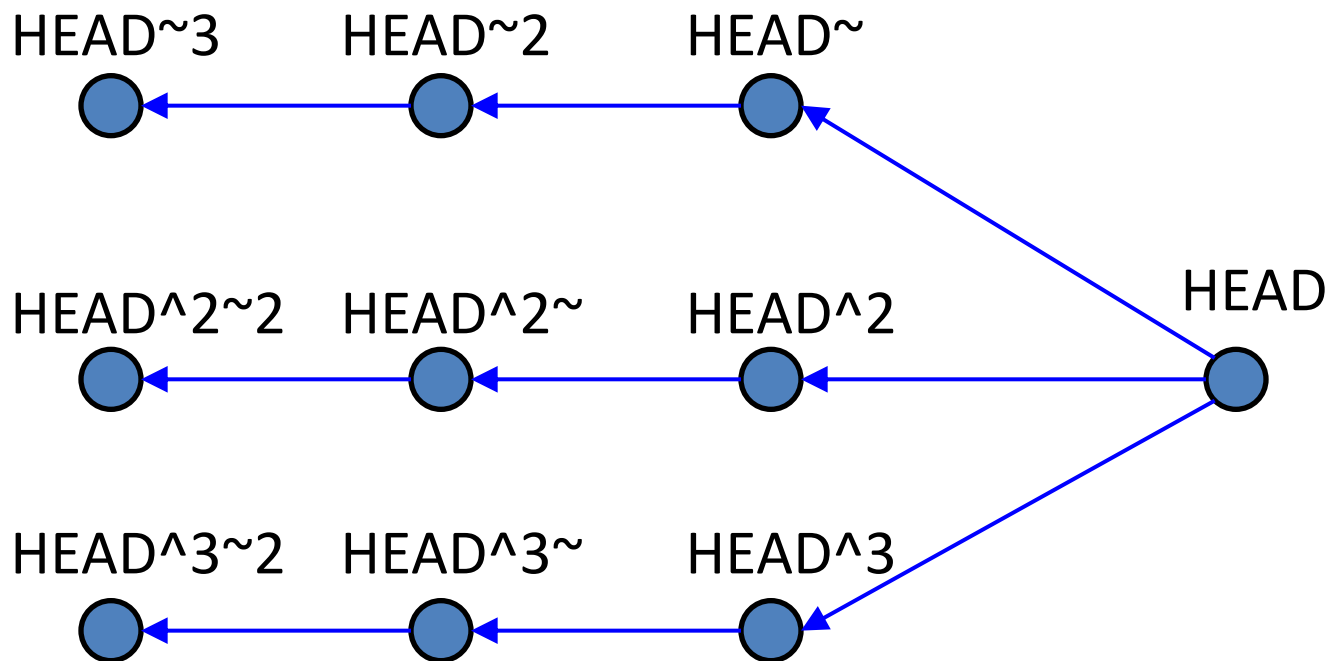
~ :first parentの世代を指定する記号

~を使うと, それぞれのコミットを上記のように指定可能

# 相対的なコミット名(^と~)

commit objectの関係を使って, commit objectを指定可能

例 : commit objectに関連がある場合



組み合わせると, それぞれのコミットを指定可能

# 再度Gitリポジトリ

## <Gitリポジトリ>

### (1) オブジェクト格納領域(object store)

(a) blob

ファイルデータを格納

(b) tree

ディレクトリ構成を作成

(c) commit

特定バージョンの情報を含んだオブジェクト

(d) tag

オブジェクトに署名をつける目的で利用

### (2) インデックス(index)

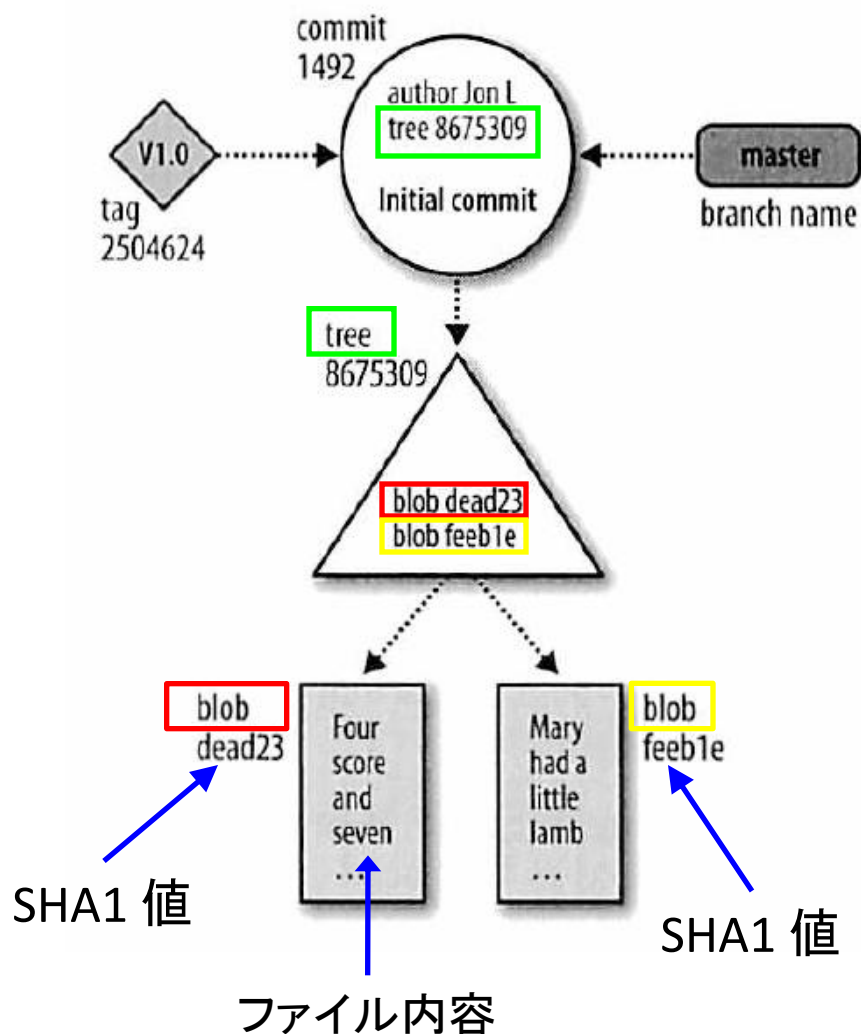
(a) リポジトリ全体のディレクトリ構造が記述された、一時的かつ動的なバイナリファイル

# Git勉強会用スライド (オブジェクトストア編)

宮崎清人

# オブジェクトストアの図(1/2)

2つのファイルをコミット時のオブジェクトストア



tag : commitを指す

commit : 著者情報, コミット  
メッセージを保持  
トップディレクトリの  
treeを指す

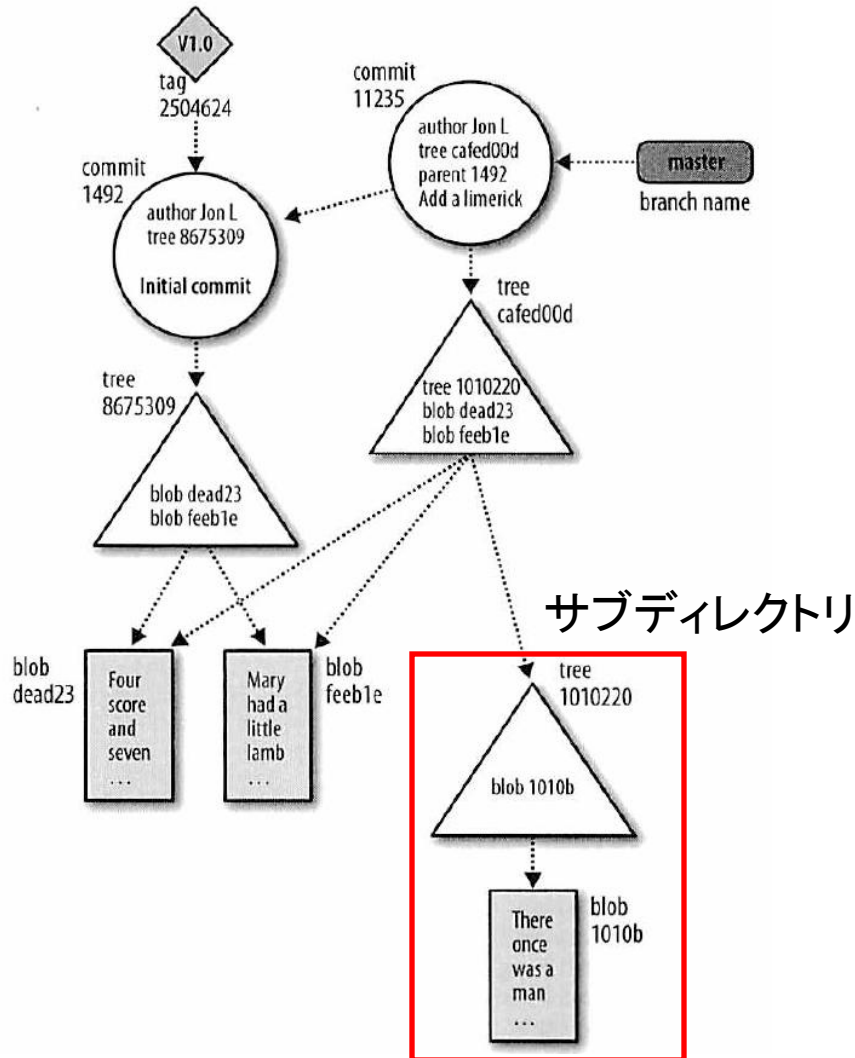
tree : 各blobを指す

blob : ファイル情報を保持



# オブジェクトストアの図(2/2)

2回目のコミットを行ったときのオブジェクトストアの例



tag : commitを指す

commit : 著者情報, コミット  
メッセージを保持  
**親のcommit**,  
トップディレクトリの  
treeを指す

tree : **サブディレクトリのtree**,  
各blobを指す

blob : ファイル情報を保持

# インデックス(Index)

## <Index>

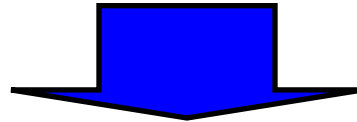
ソートされたパス名とパーミッション, blob の SHA1値の一覧を含むバイナリファイル

## <目的>

(1) ワーキングディレクトリにおける変更の蓄積

## <働き>

- (1) ファイルの全体の内容は含まない
- (2) コミット対象の内容だけを追跡

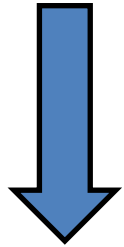


Gitは何をコミットするべきかについて**インデックスを参照**

# Gitのファイル管理の構成

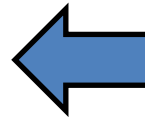
ワーキングディレクトリ

管理対象のファイルを格納し、  
作業するディレクトリ



ワーキングディレクトリの変更をインデックスに蓄積  
(蓄積を「ステージする」と呼ぶ)

インデックス



修正, 収集のために挿入



差分としてコミット

オブジェクト格納領域

変更をコミットするのに2ステップ必要

# Git's Object Model(1/8)

Working  
directory



<初期状態>

ディレクトリを作成

<コマンド>

\$ mkdir foo

# Git's Object Model(2/8)

Working  
directory



Index



Object  
store



<リポジトリ作成>

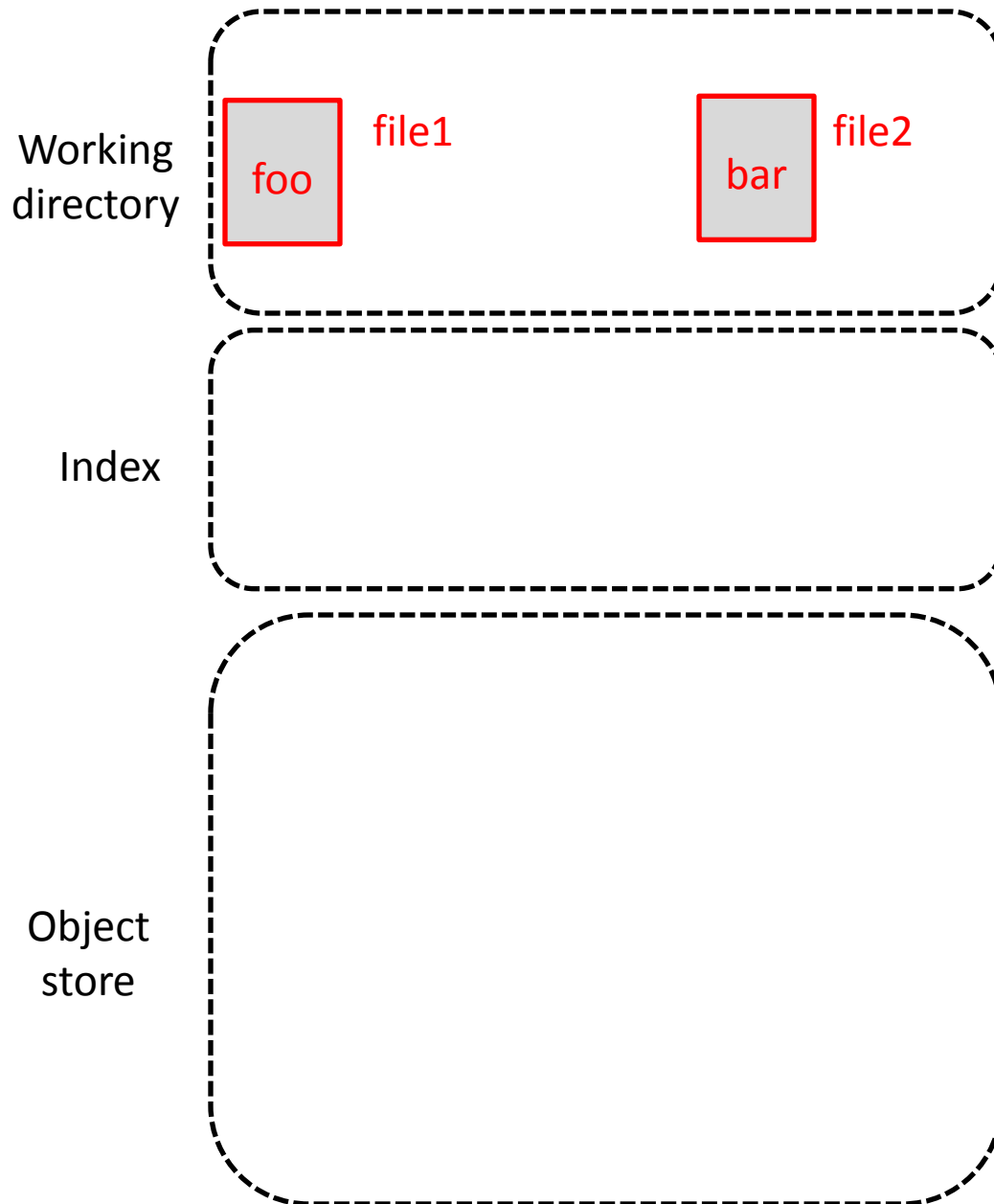
Gitリポジトリを作成

IndexとObject storeが作成される

<コマンド>

\$ git init

# Git's Object Model(3/8)



## <ファイル作成>

Working directoryに以下の2つのファイルを作成する.

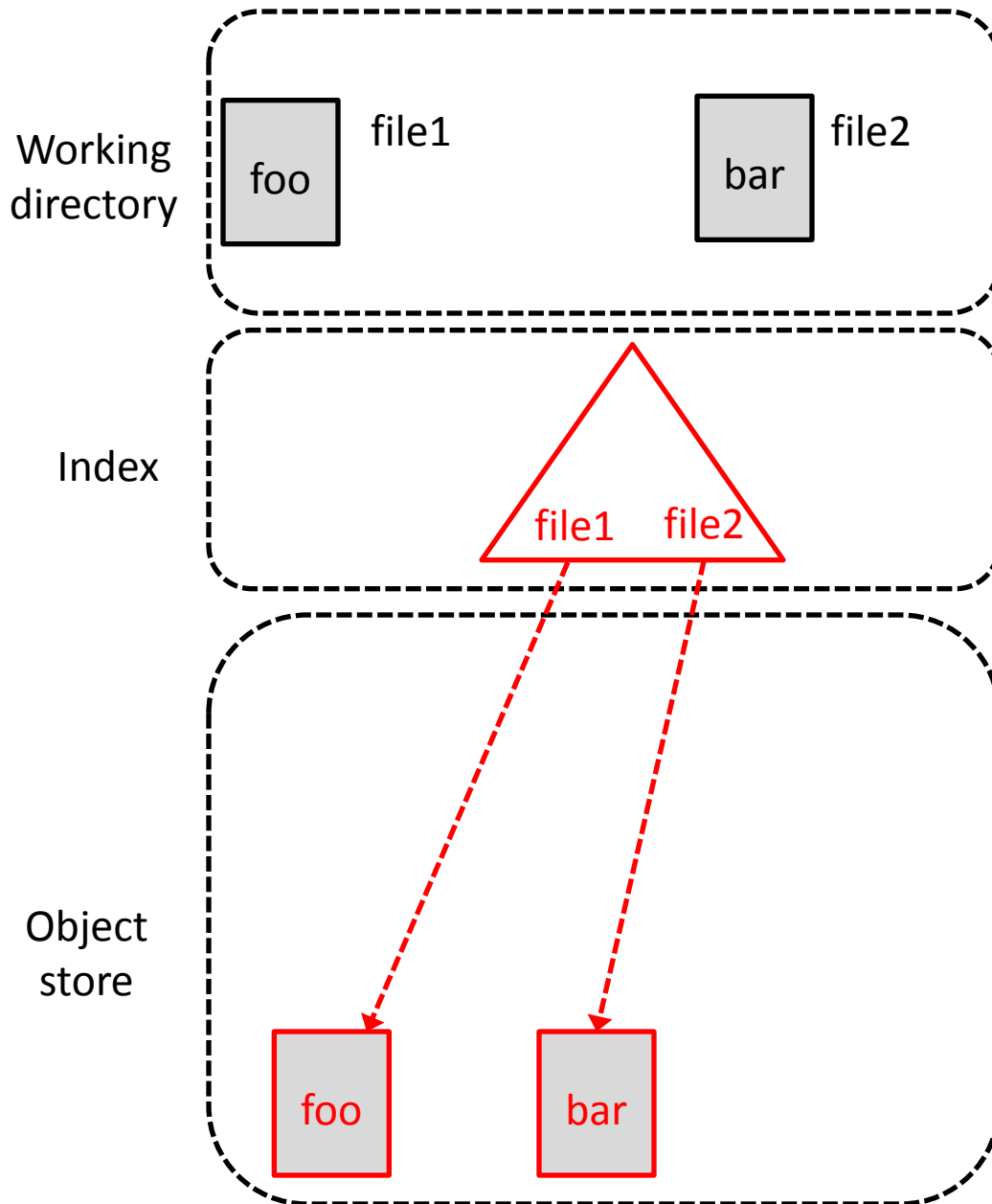
- (1) file1  
fooという内容
- (2) file2  
barという内容

## <コマンド>

```
$ echo "foo" > file1
```

```
$ echo "bar" > file2
```

# Git's Object Model(4/8)



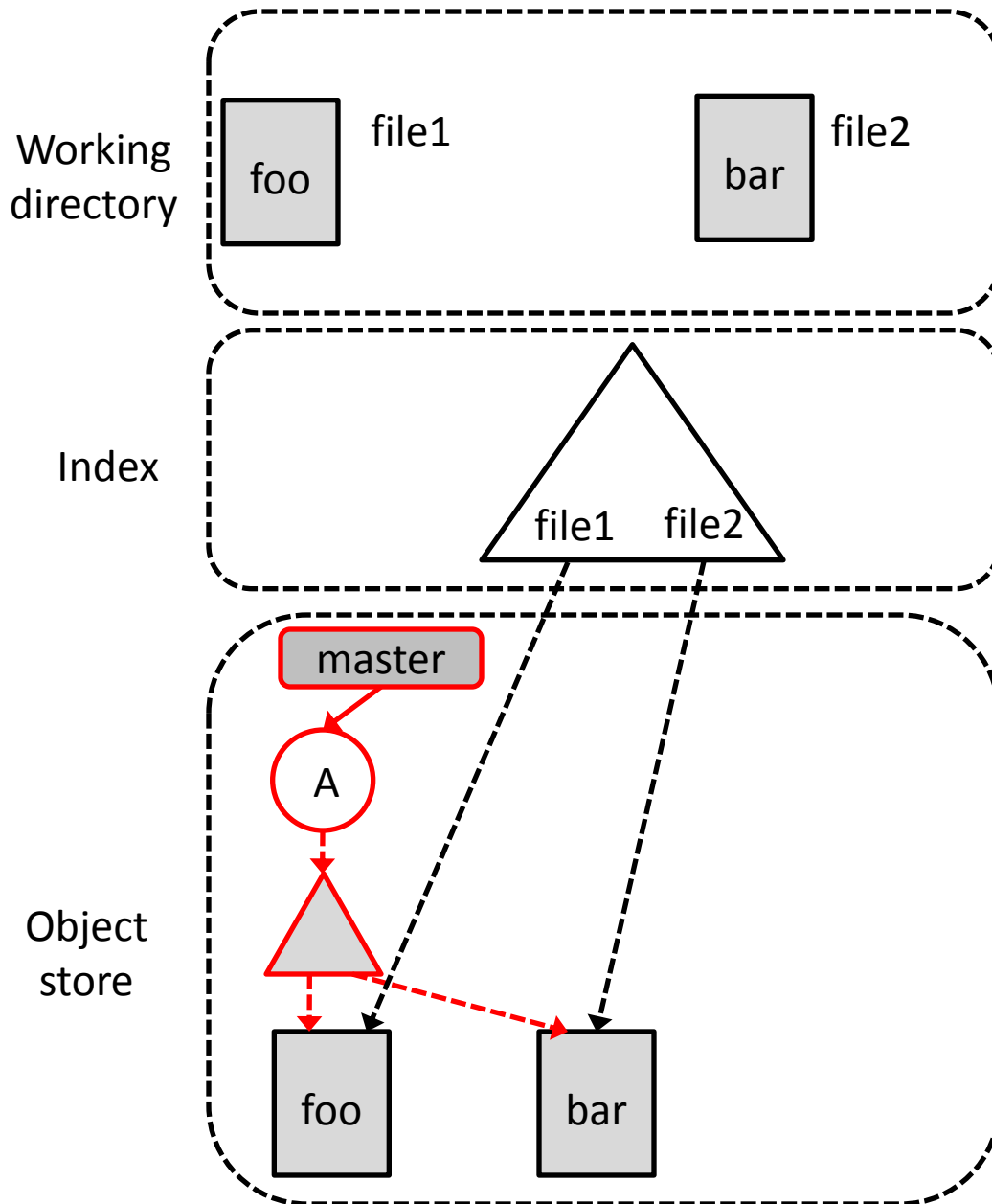
<ステージ>

Indexが各ファイルを指す

<コマンド>

\$ git add file1 file2

# Git's Object Model(5/8)



<コミット>

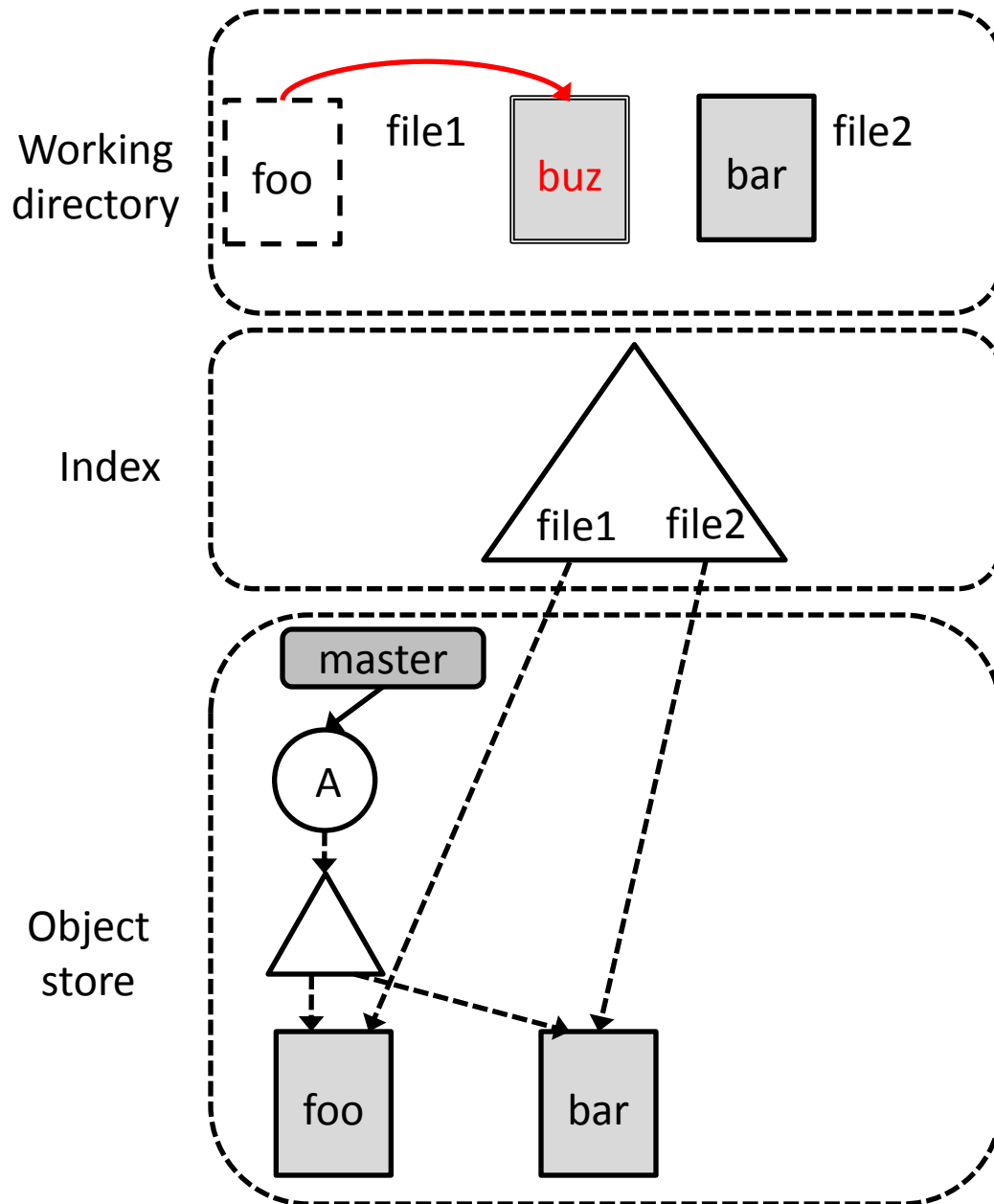
- (1) Indexの内容をコミット
- (2) Aコミットが作られる

<コマンド>

\$ git commit



# Git's Object Model(6/8)



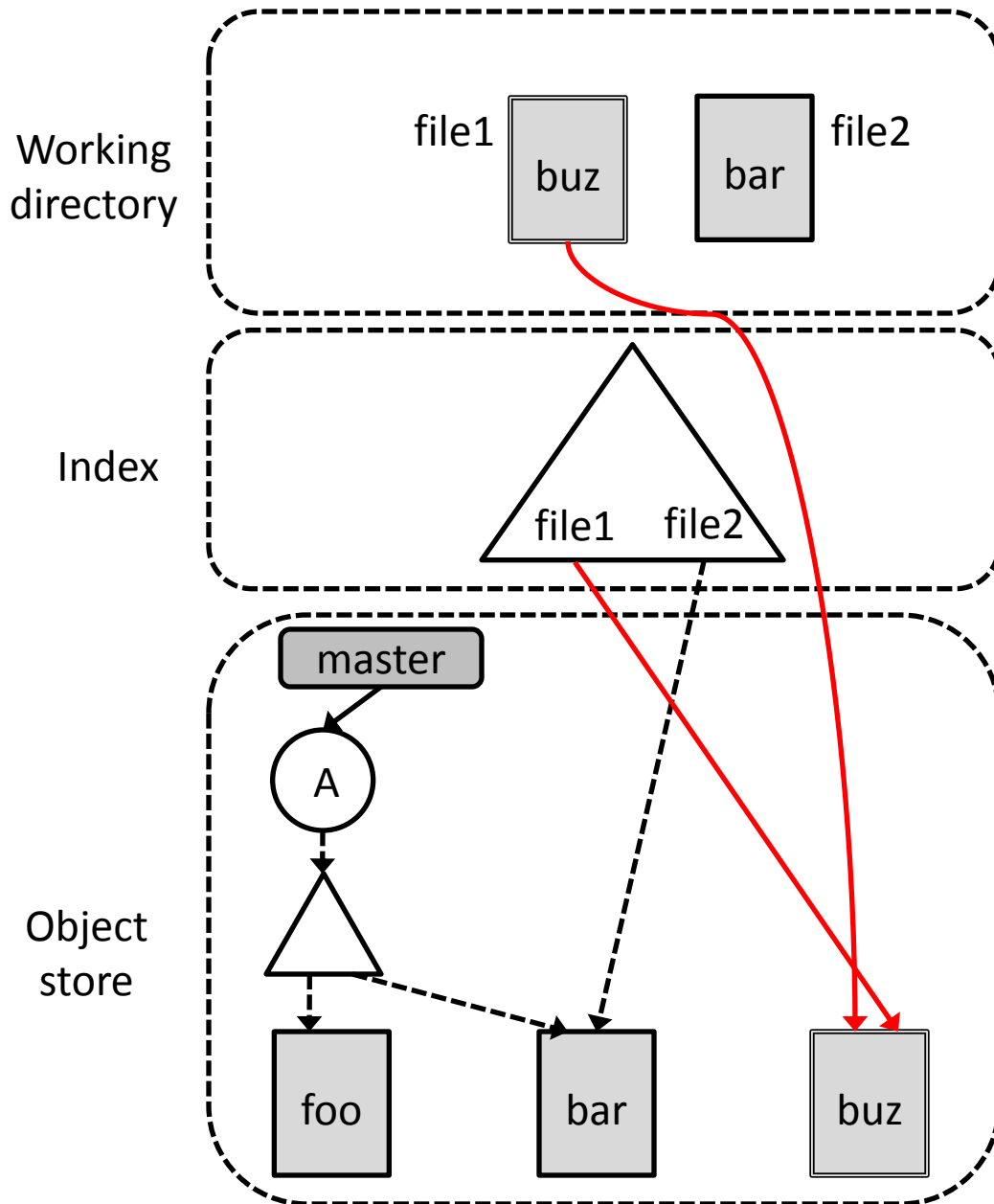
<ファイル編集>

file1の内容をfooから  
buzに編集

<コマンド>

\$ vi file1

# Git's Object Model(7/8)



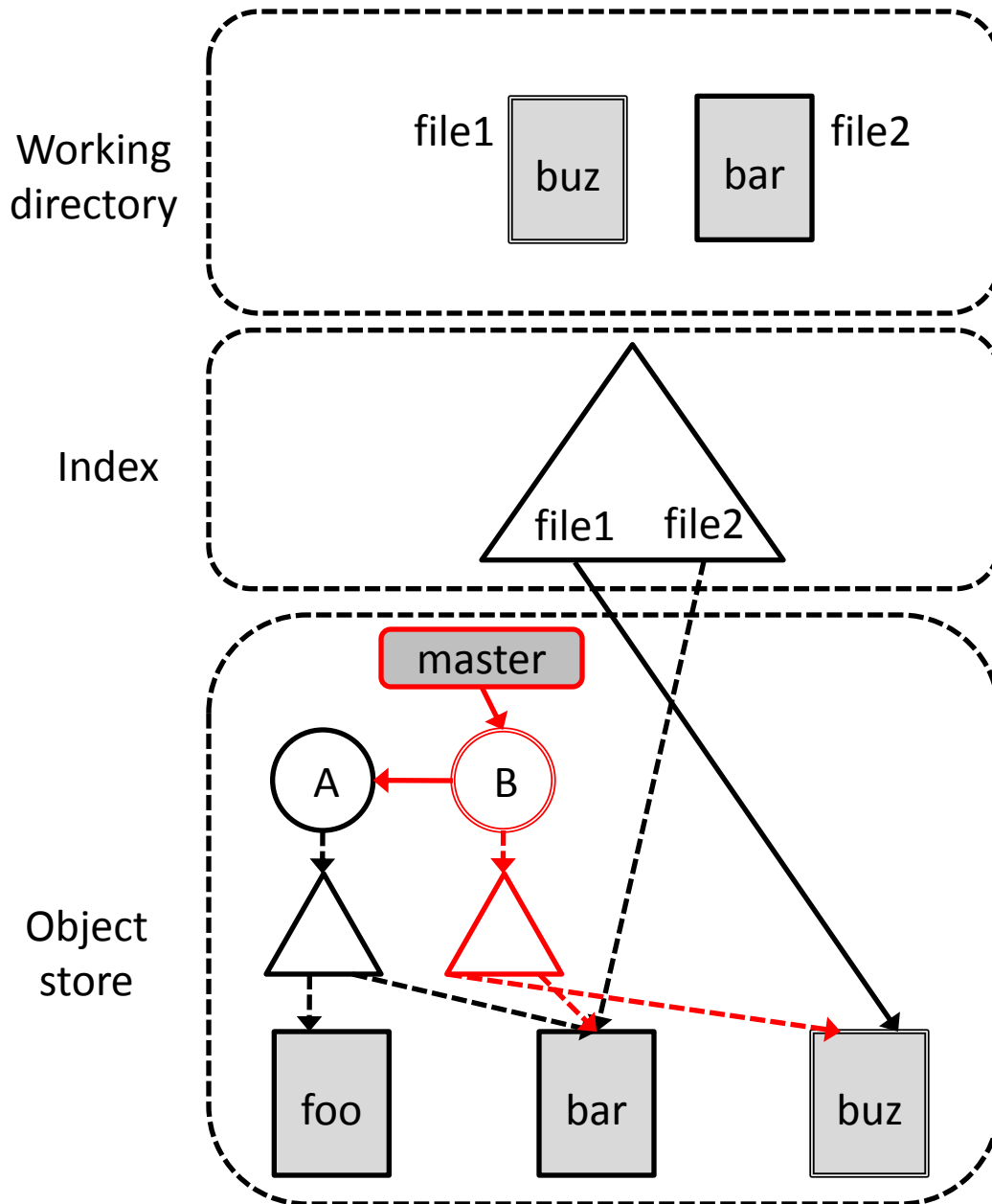
<ステージ>

Indexが書き換えられたfile1を指す

<コマンド>

\$ git add file1

# Git's Object Model(8/8)



<コミット>

- (1) Indexの内容をコミット
- (2) Bコミットが作られる

<コマンド>

\$ git commit

# Git勉強会用スライド (リポジトリ編)

池田 騰

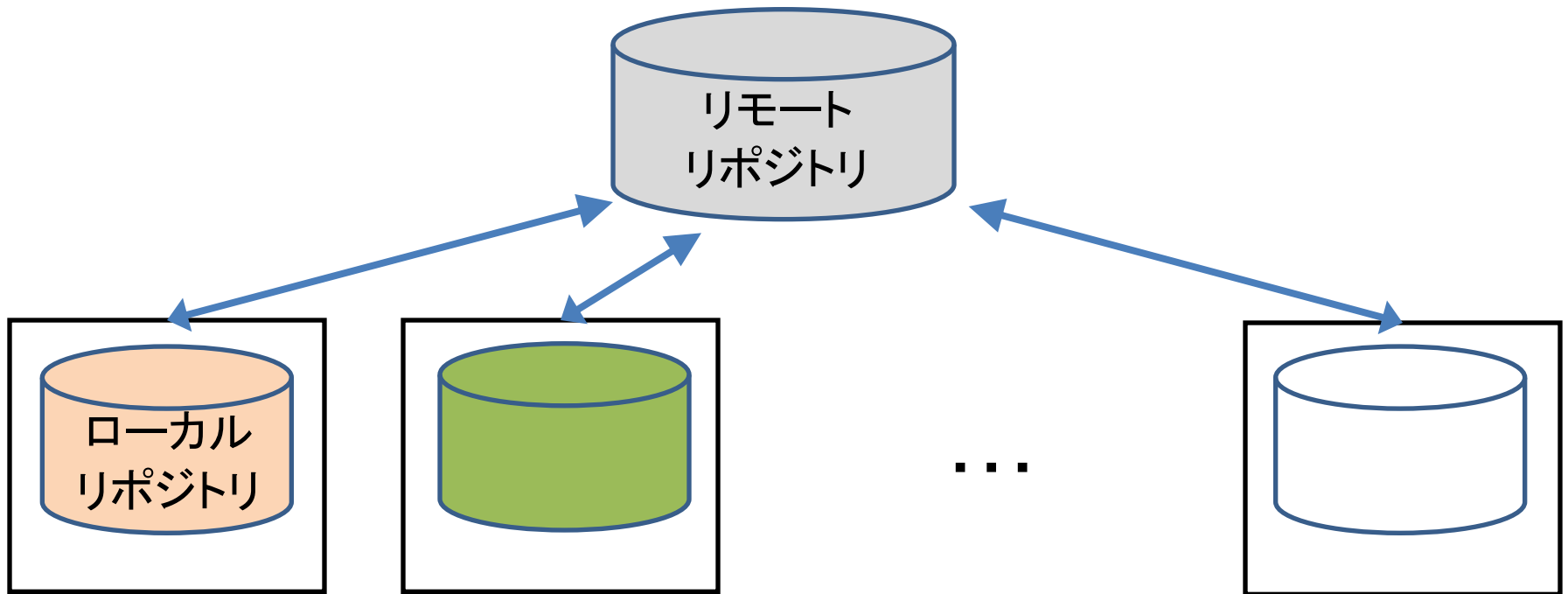
# リポジトリの種類

## <ローカルリポジトリ>

- (1) 自身の作業するリポジトリ

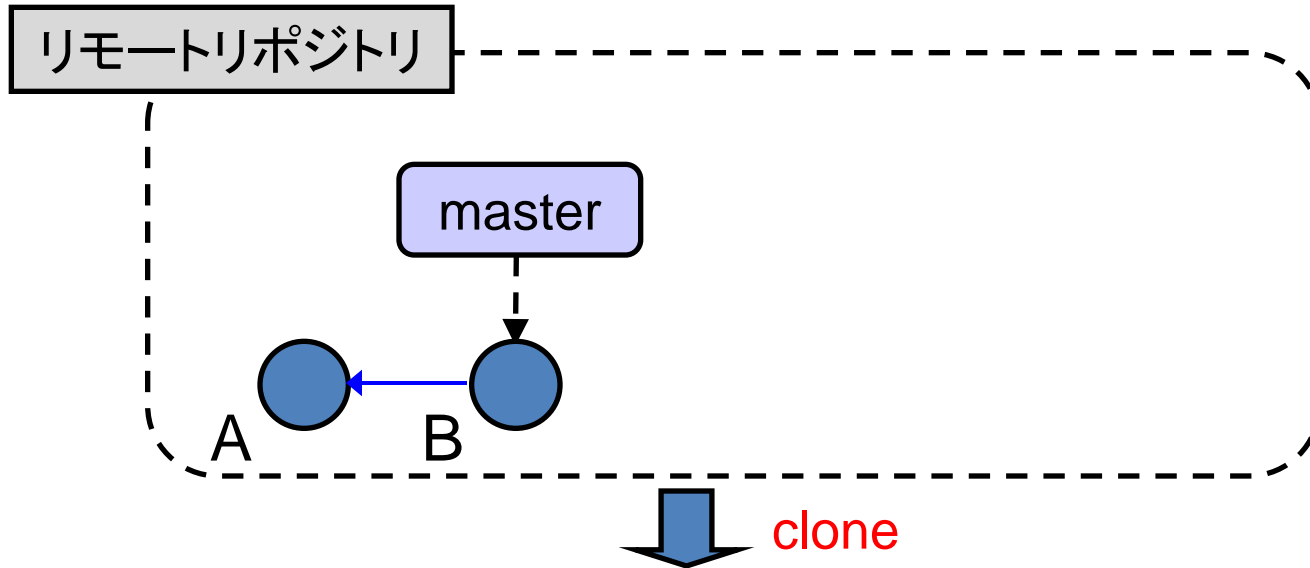
## <リモートリポジトリ>

- (1) ローカルリポジトリとデータを交換するリポジトリ
- (2) 物理的にリモート(遠隔)であるとは限らない



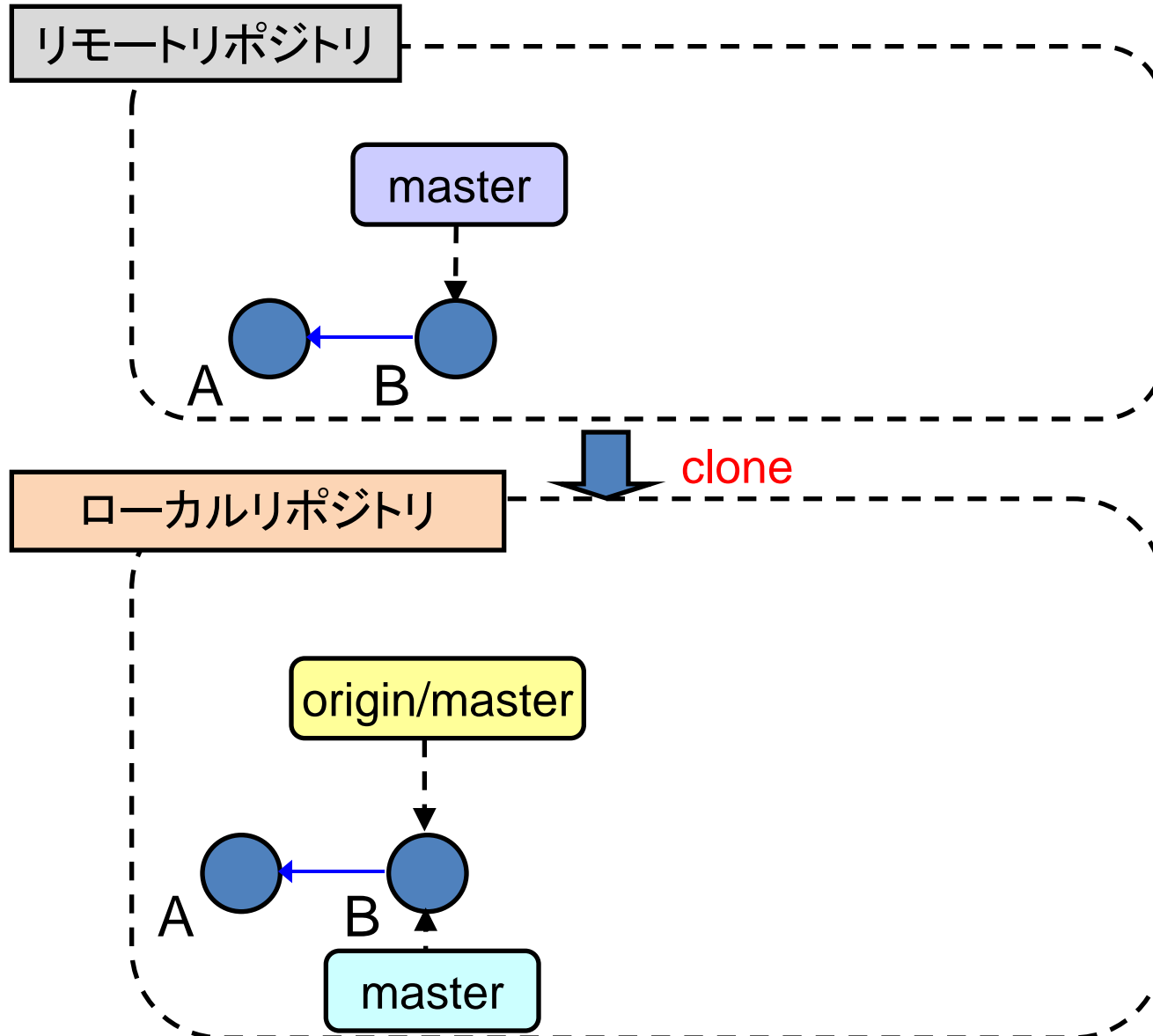
# リモートリポジトリのclone(1/2)

<リポジトリのクローン作成>



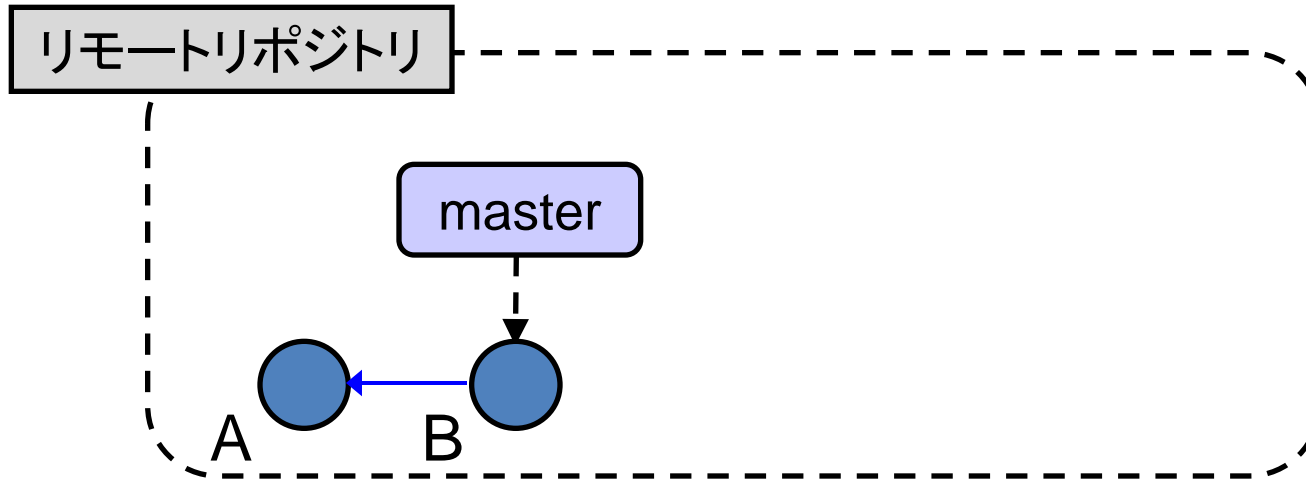
# リモートリポジトリのclone(2/2)

## <リポジトリのクローン作成>



# リモート追跡ブランチ

## <リポジトリのクローン作成>



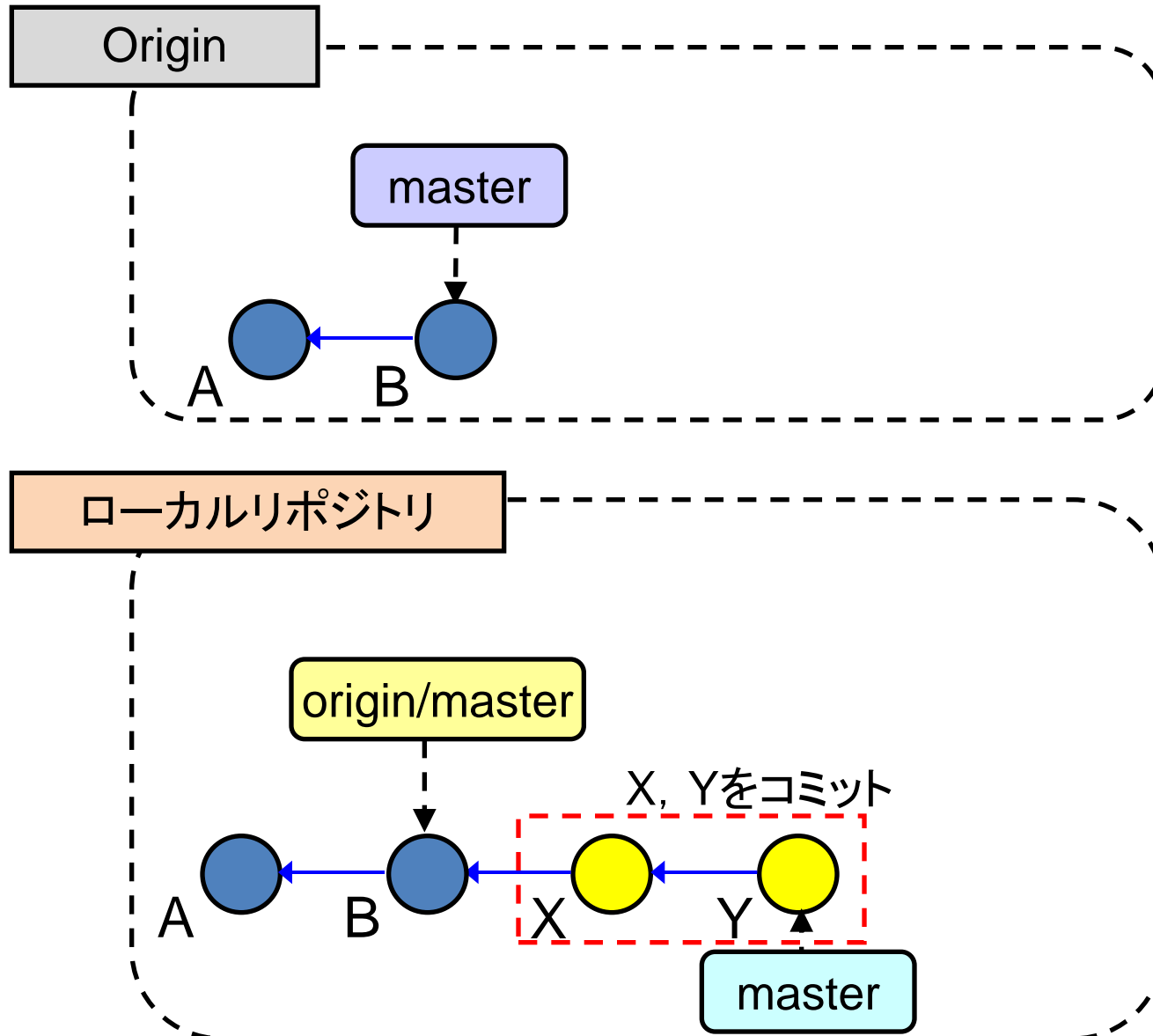
### リモート追跡ブランチ

- (1) 元のリポジトリの変更を追跡する  
ブランチ
- (2) 追跡ブランチに対して変更は加えない



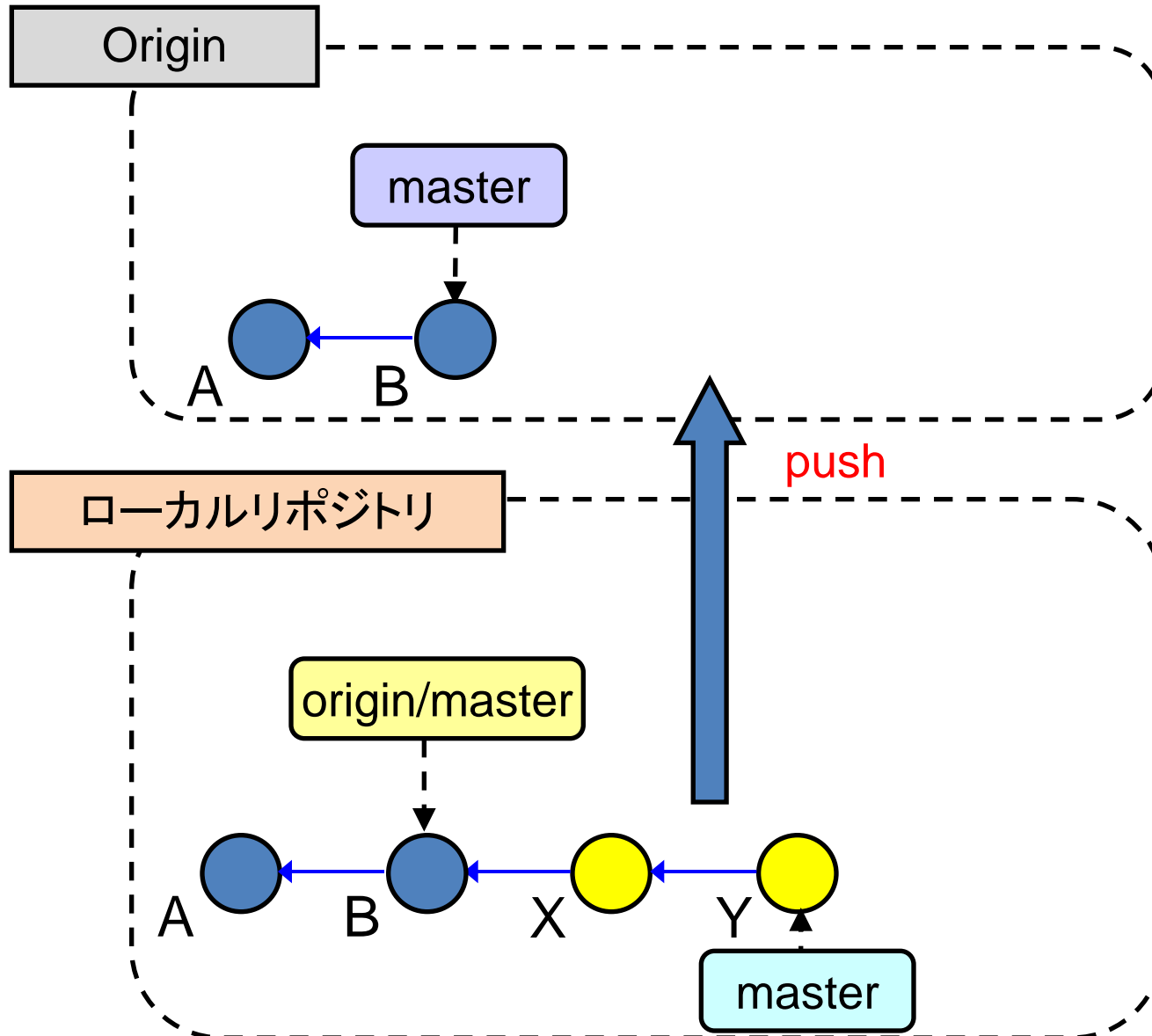
# ローカルリポジトリでのコミット

## <変更の送信>



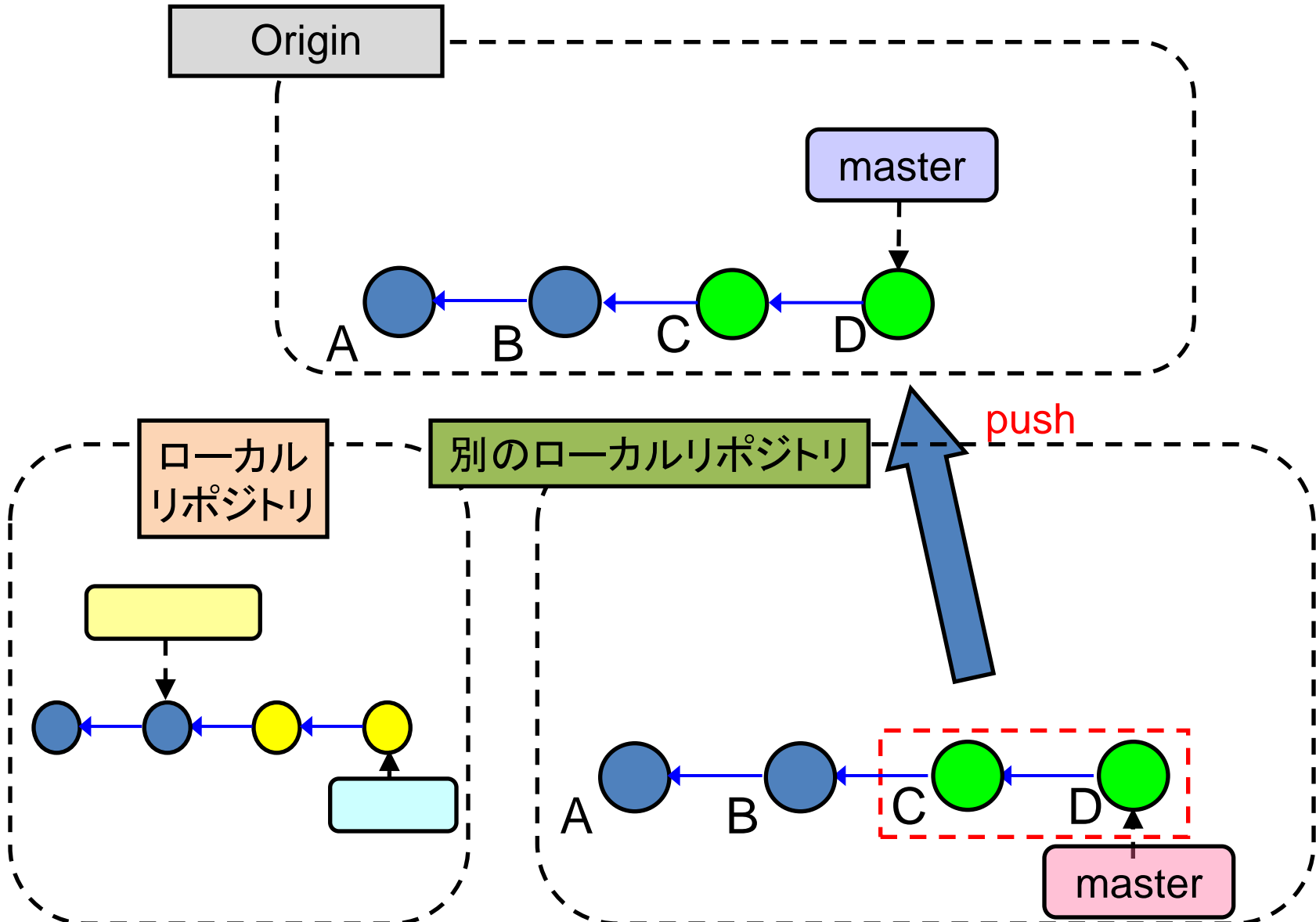
# Originへのpush(1/4)

<変更の送信>



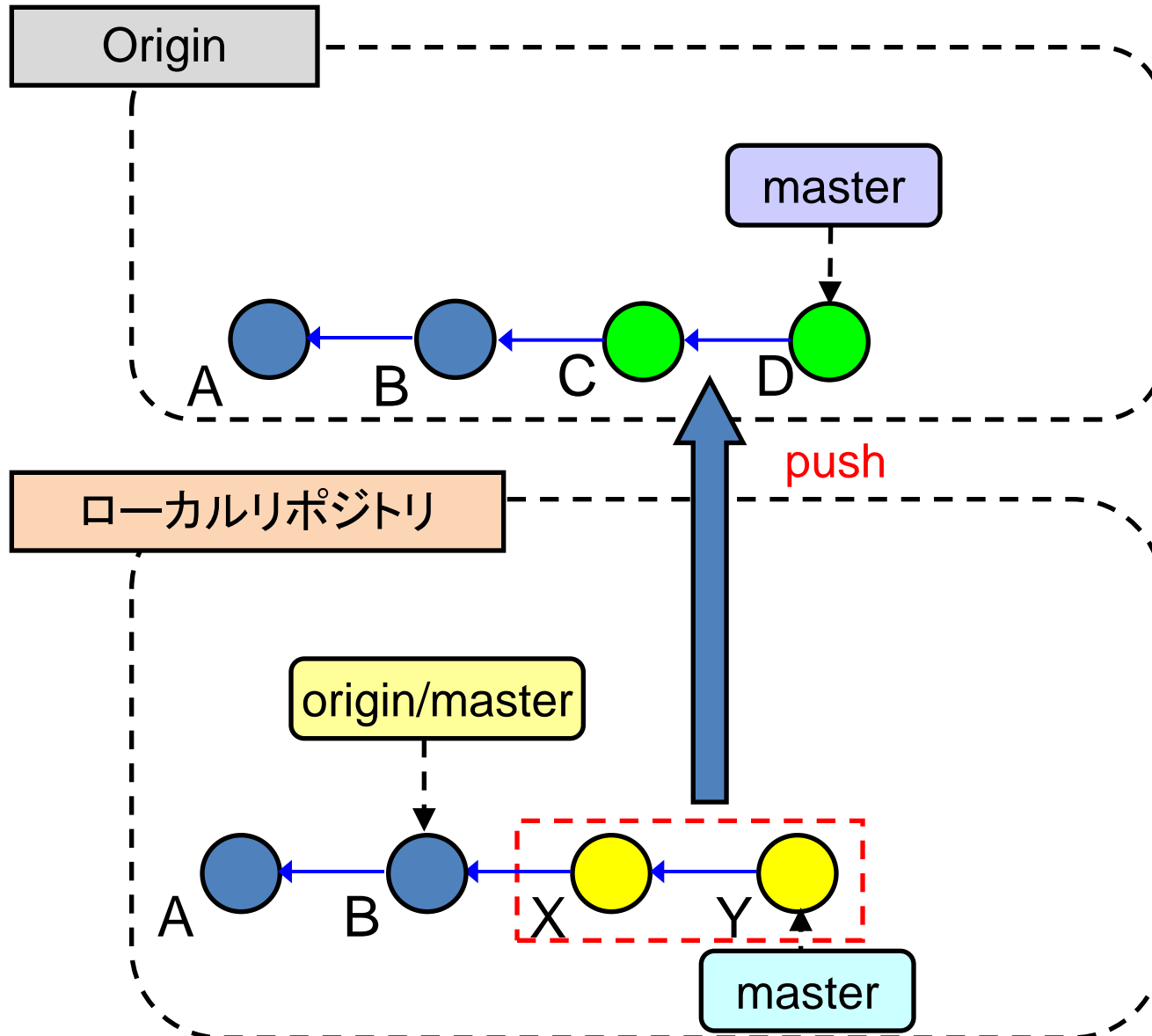
# Originへのpush(2/4)

<変更の送信>



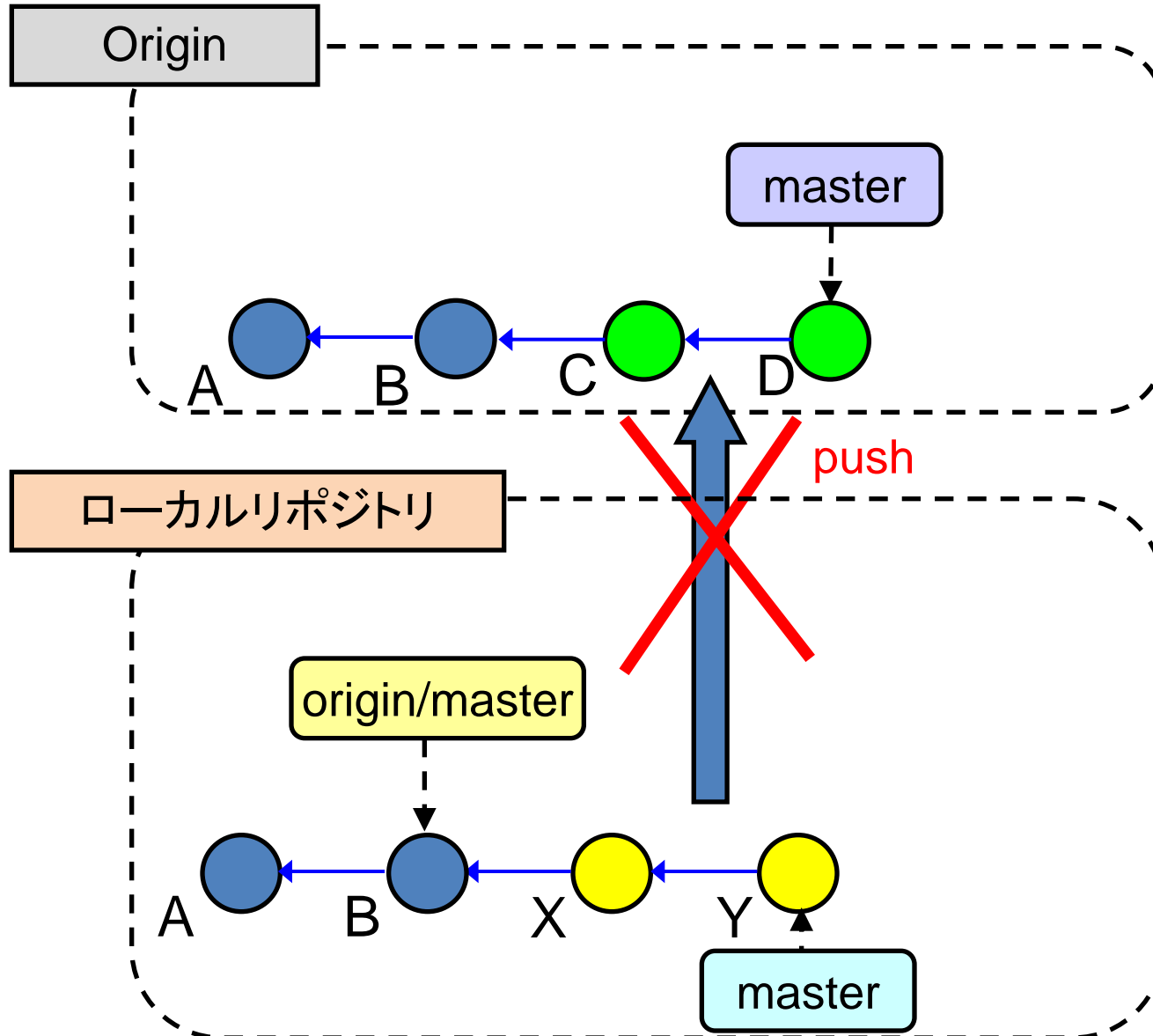
# Originへのpush(3/4)

<変更の送信>



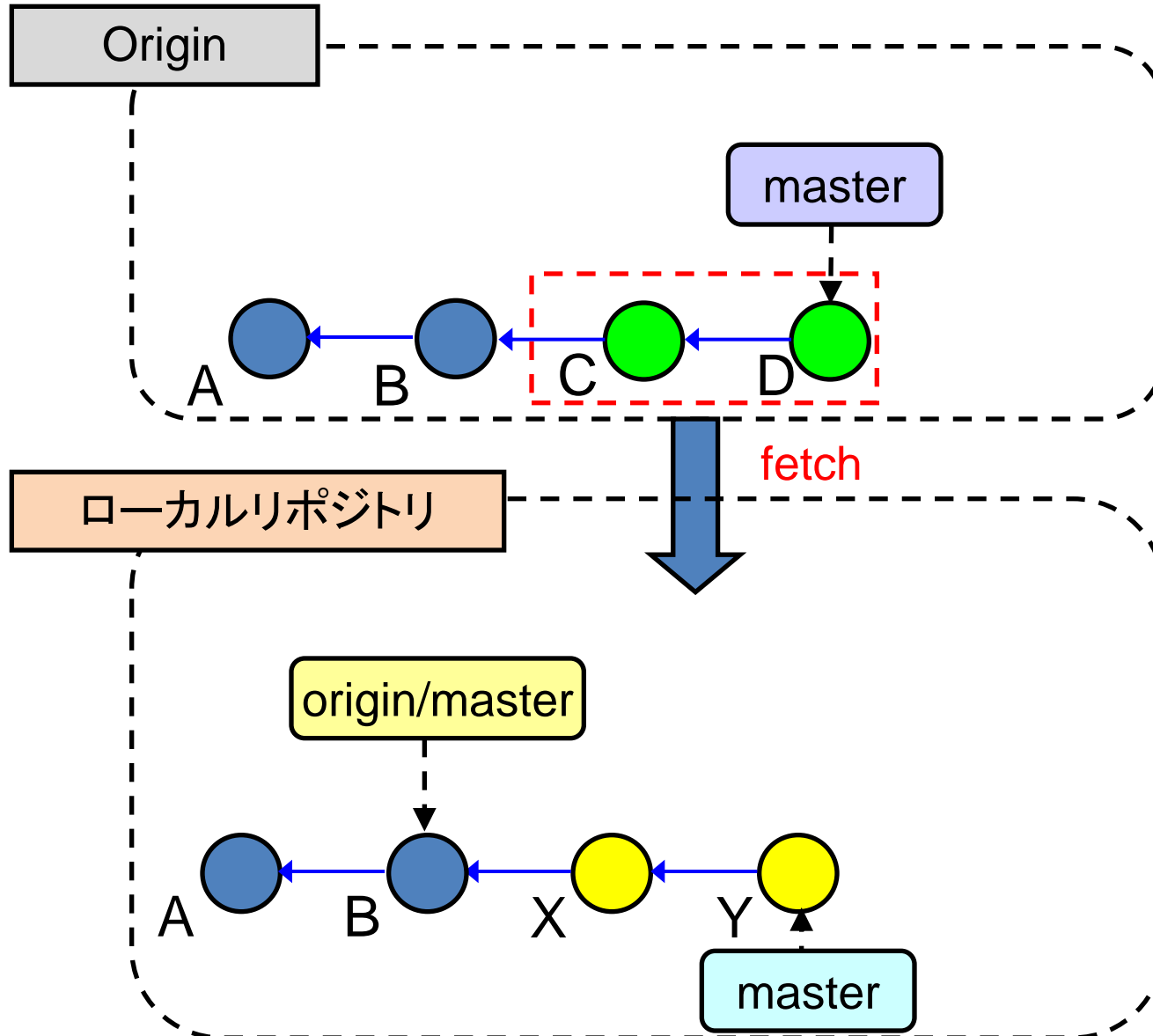
# Originへのpush(4/4)

<fast-forwardではないプッシュ>



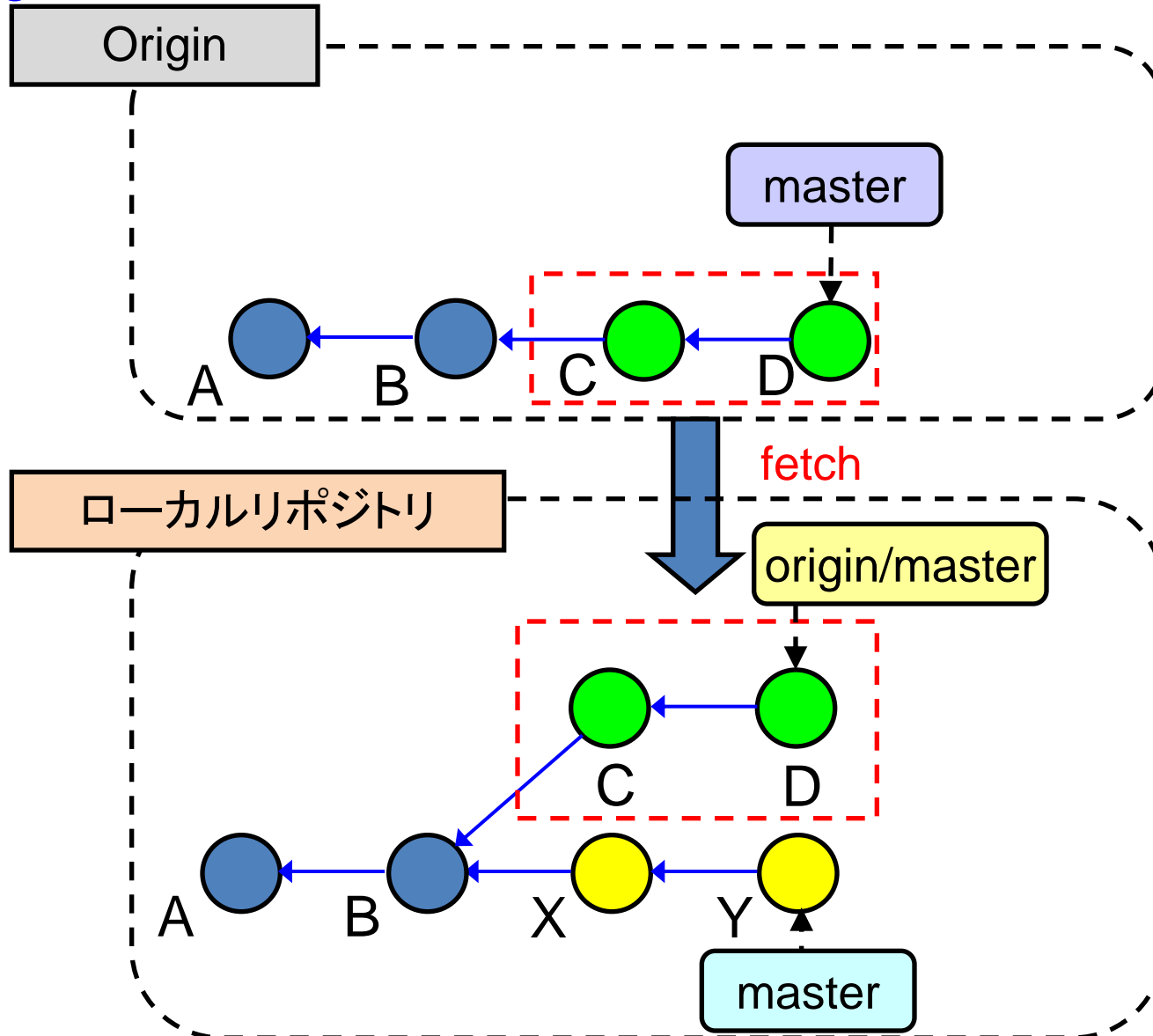
# フェッチ(fetch)(1/2)

<Originの変更履歴を取得>



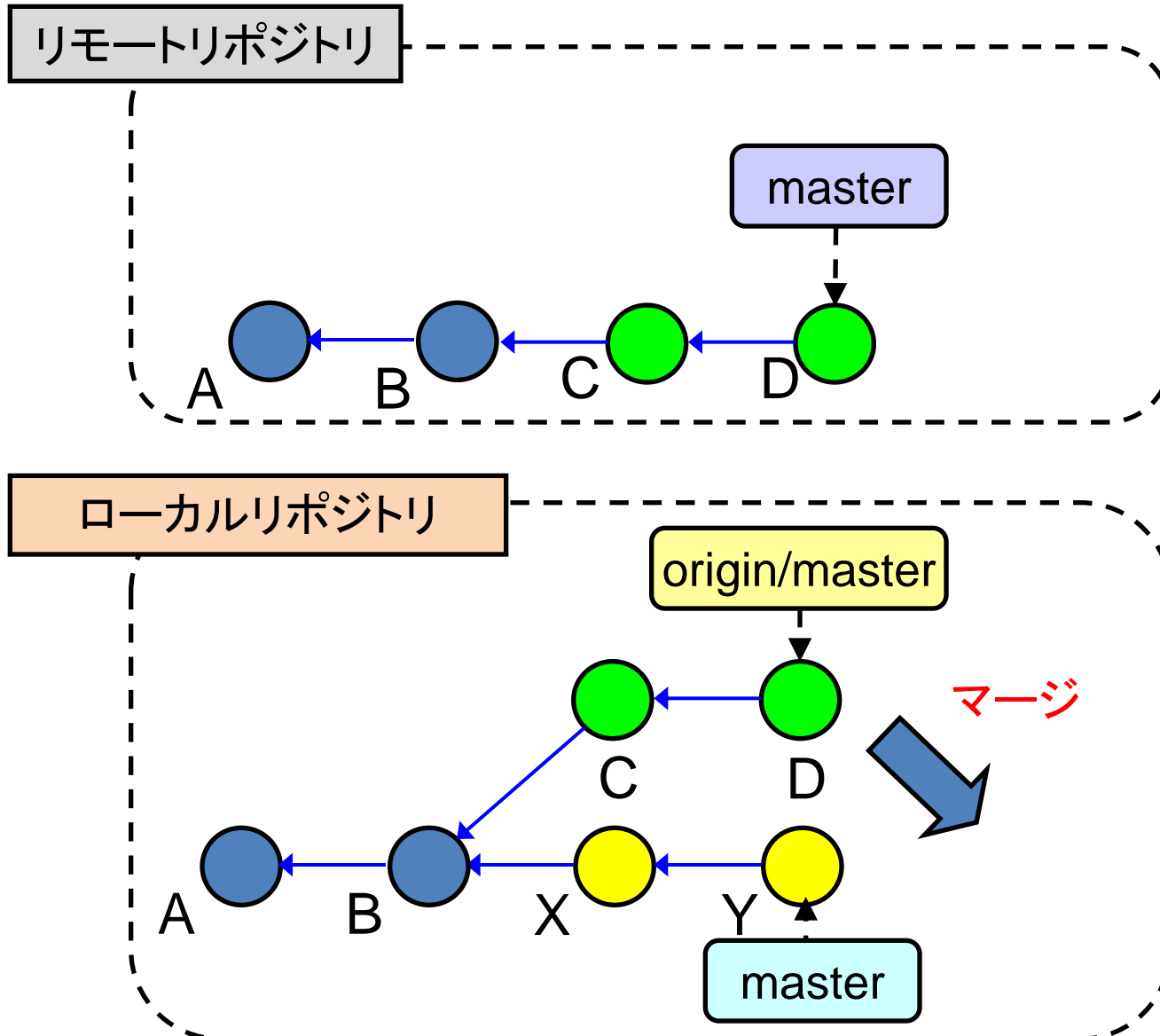
# フェッチ(fetch)(2/2)

< Originの変更履歴を取得 >



# 変更点をマージ(1/2)

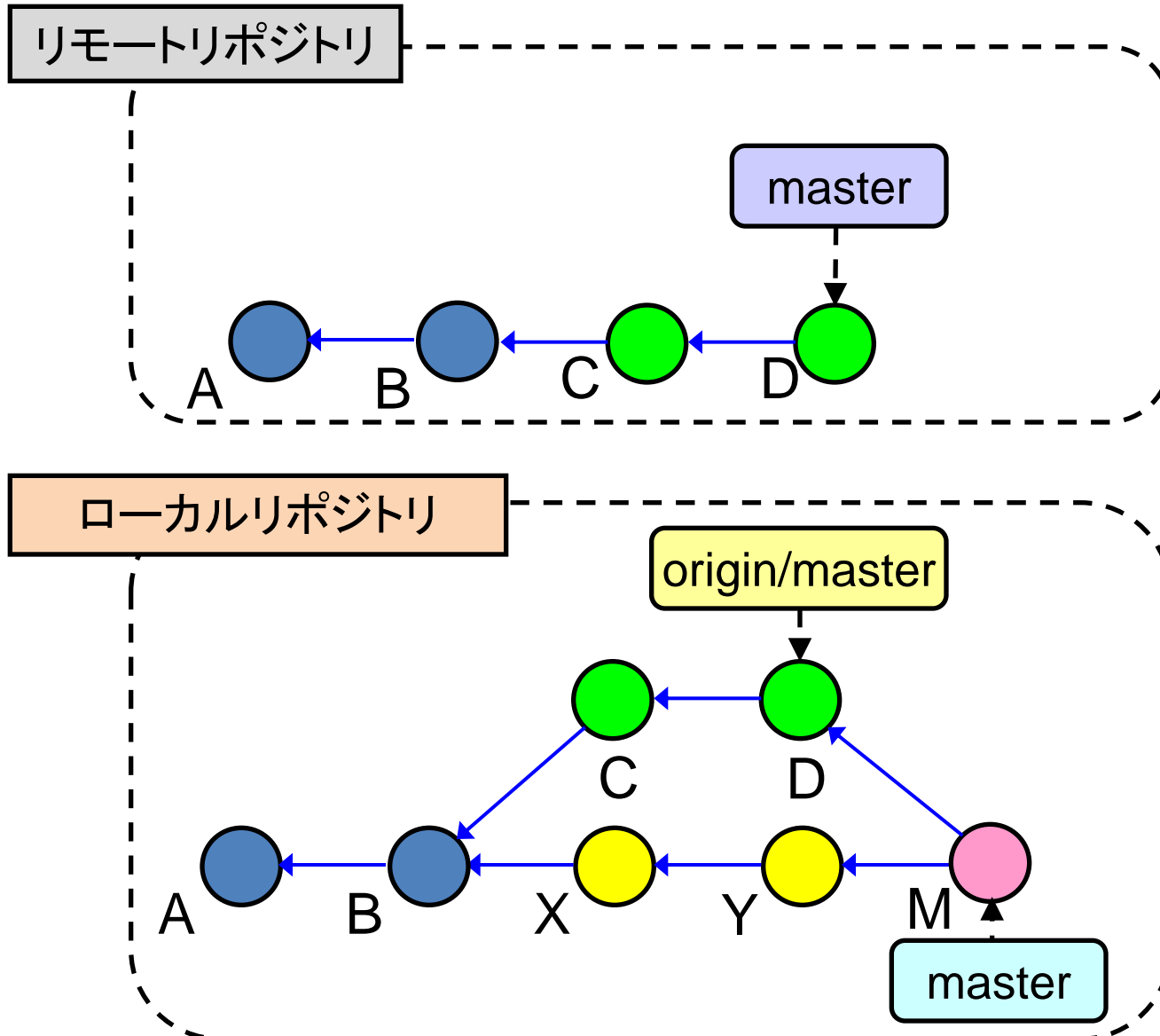
## <履歴のマージ>





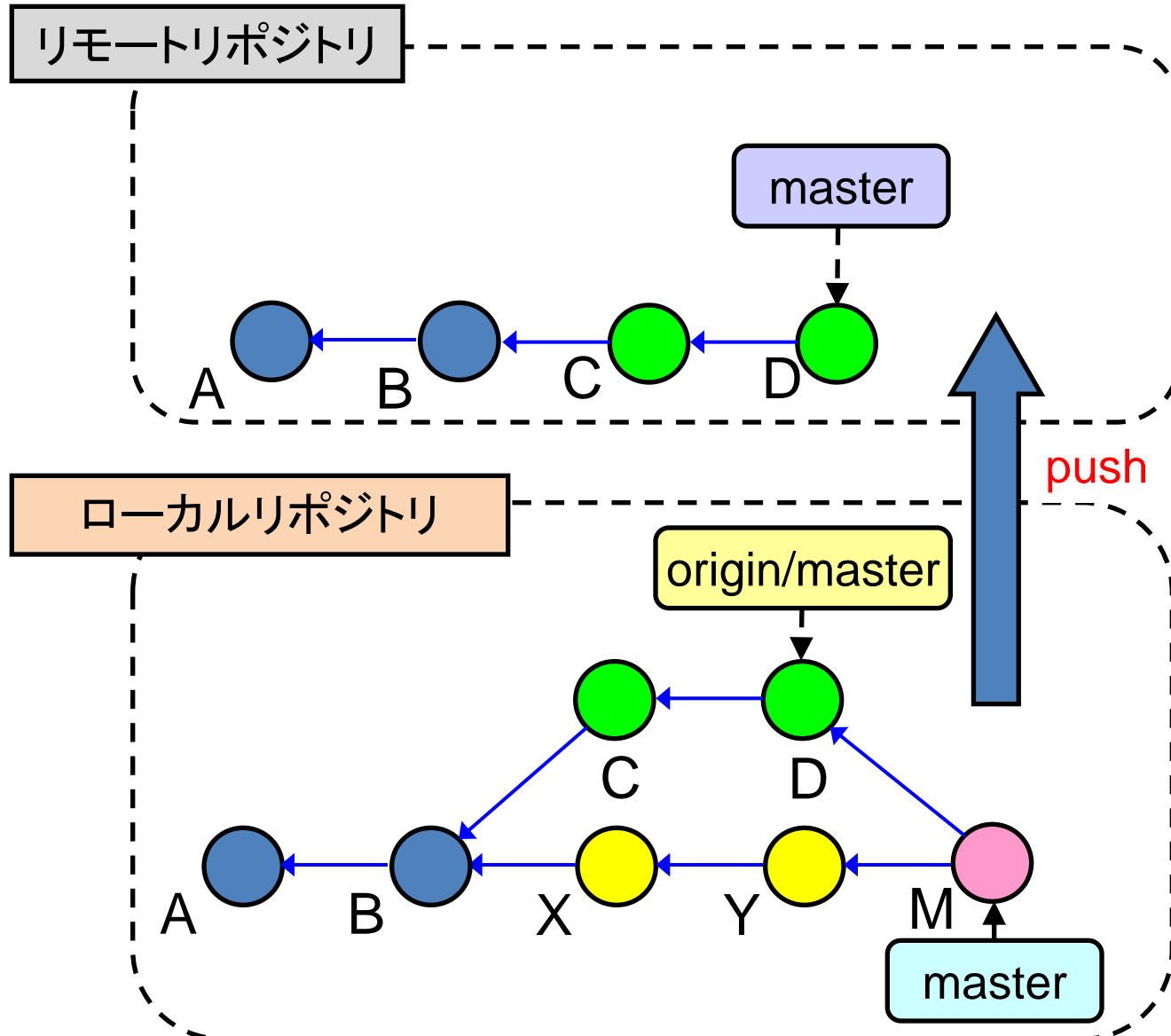
# 変更点をマージ(2/2)

## <履歴のマージ>



# 変更点をpush

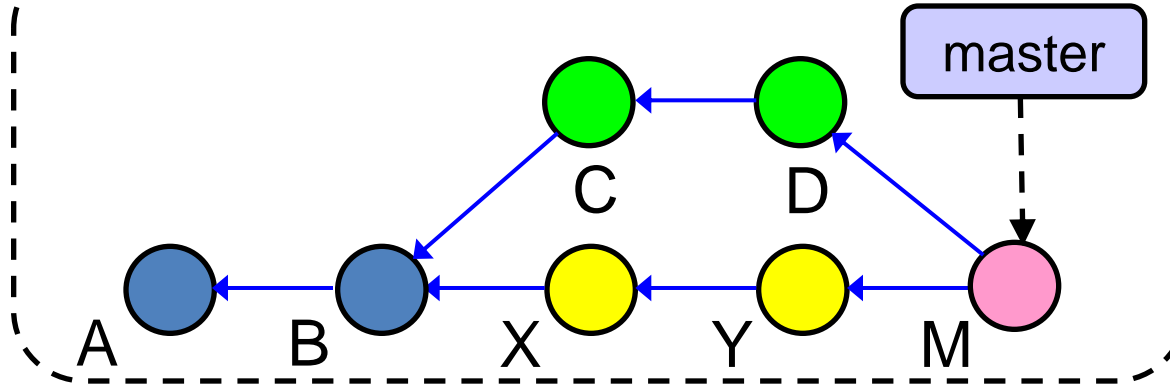
<マージされた履歴のプッシュ>



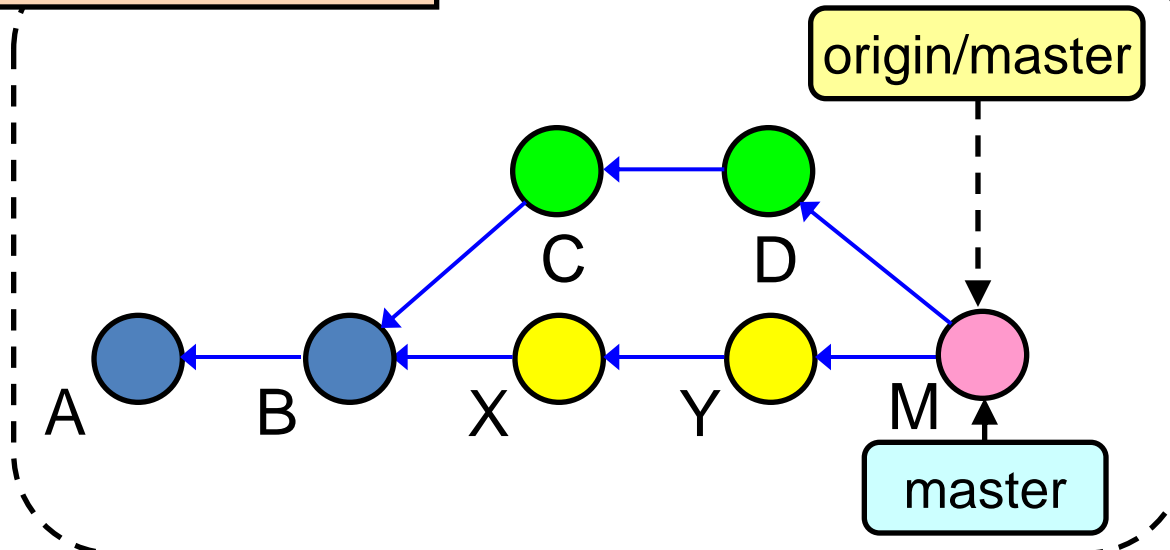
# 最終状態

<プッシュ後のマージ済み履歴>

リモートリポジトリ



ローカルリポジトリ



# Git勉強会用スライド (Git実践編)

吉井英人

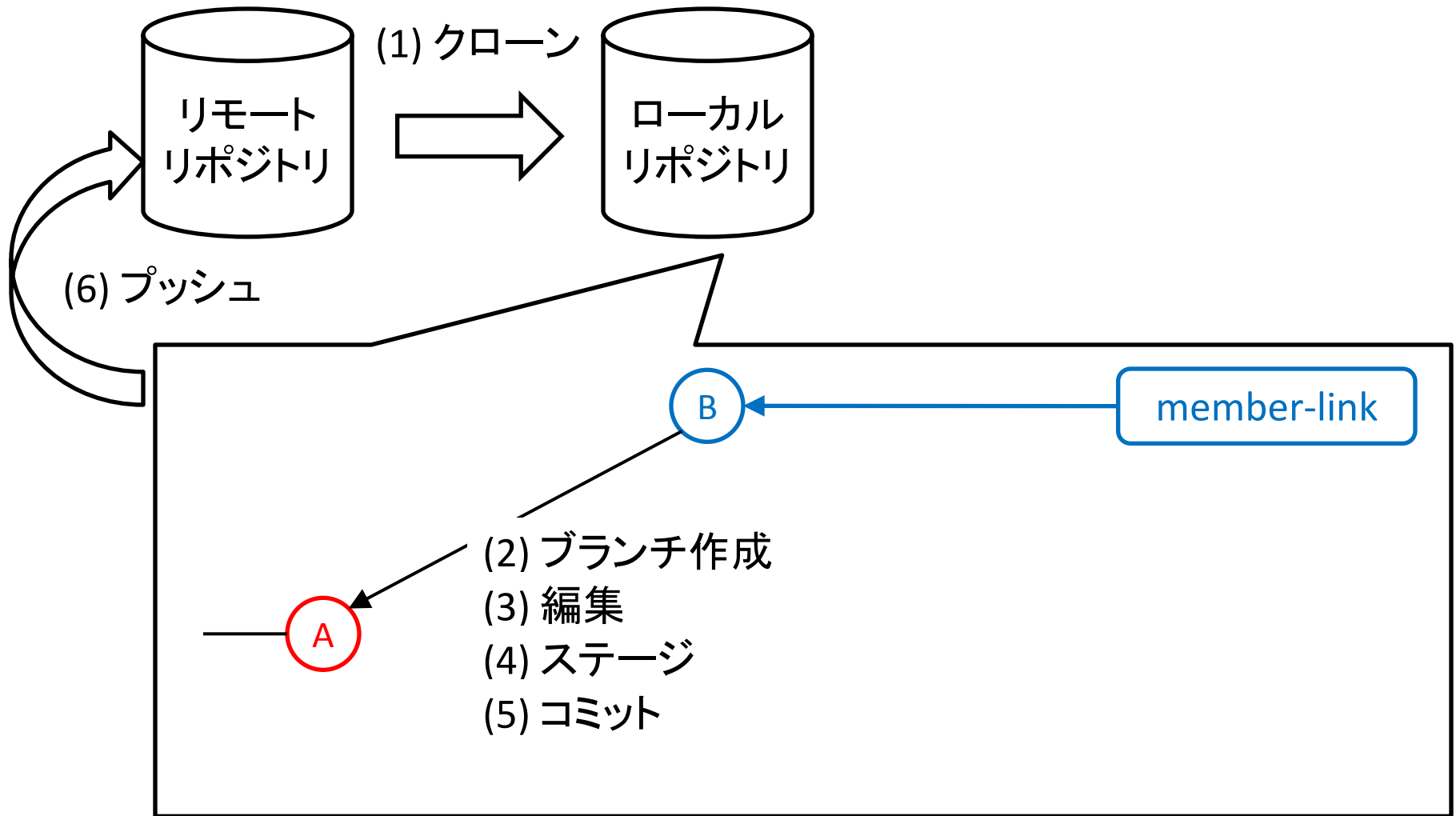
# はじめに

実際にノムニチを例にしてGitを扱う様子を示す

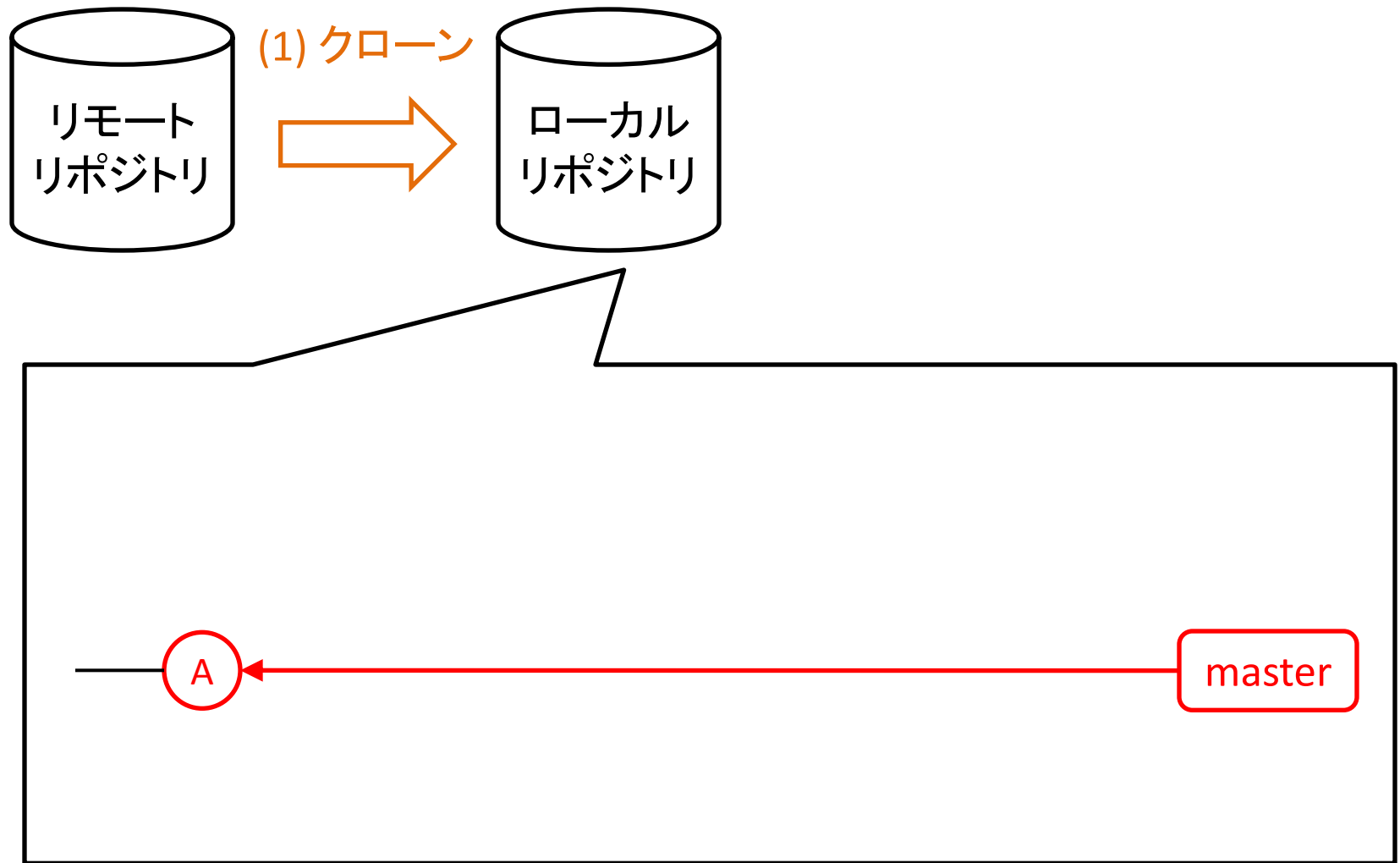
乃村研情報ページにリンクを追加する  
構成員ページに吉井のリンクを追加する

本日はこの要望の解決を例として, Gitの使い方を示す

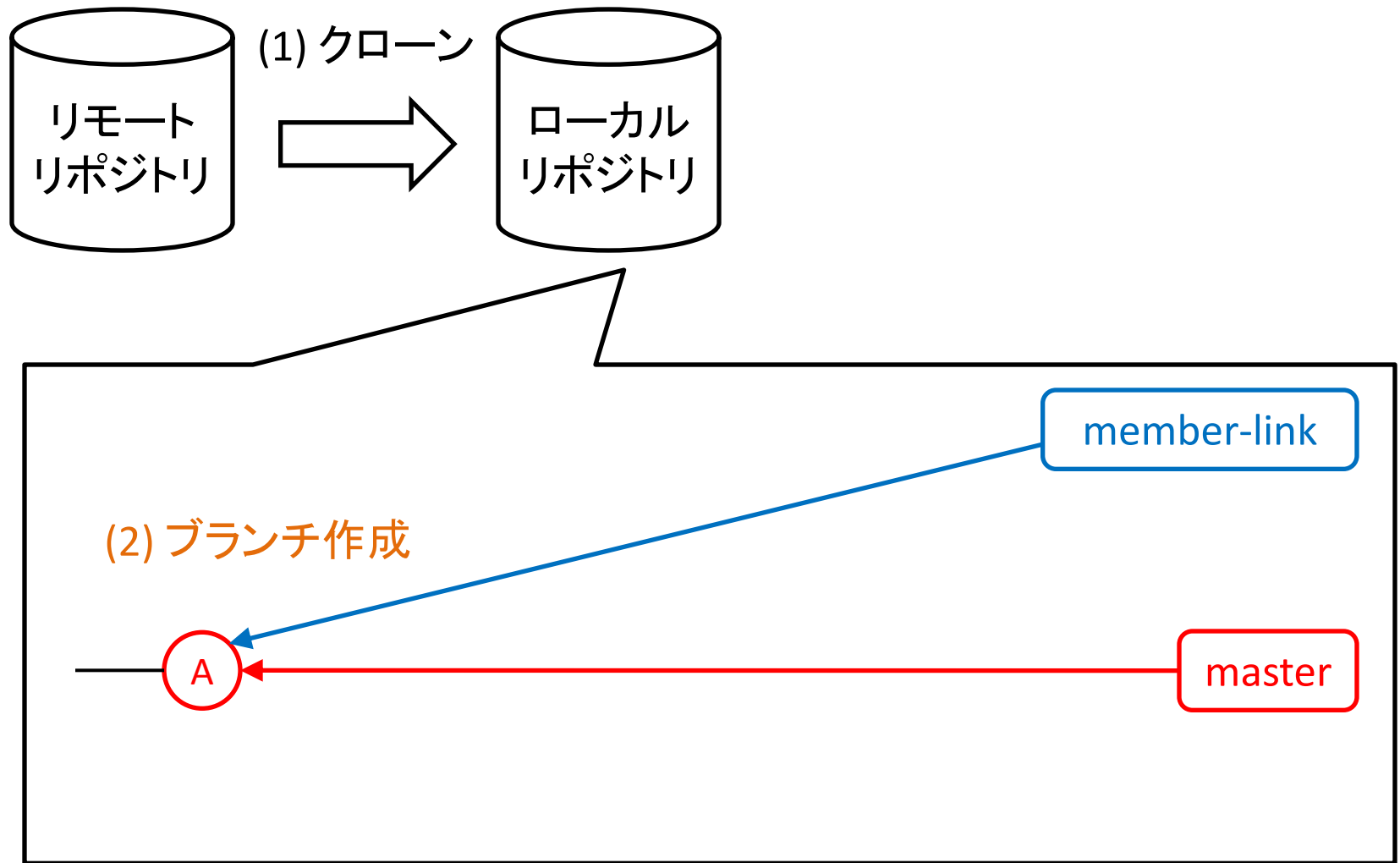
# 本日の処理流れ



# 本日の処理流れ

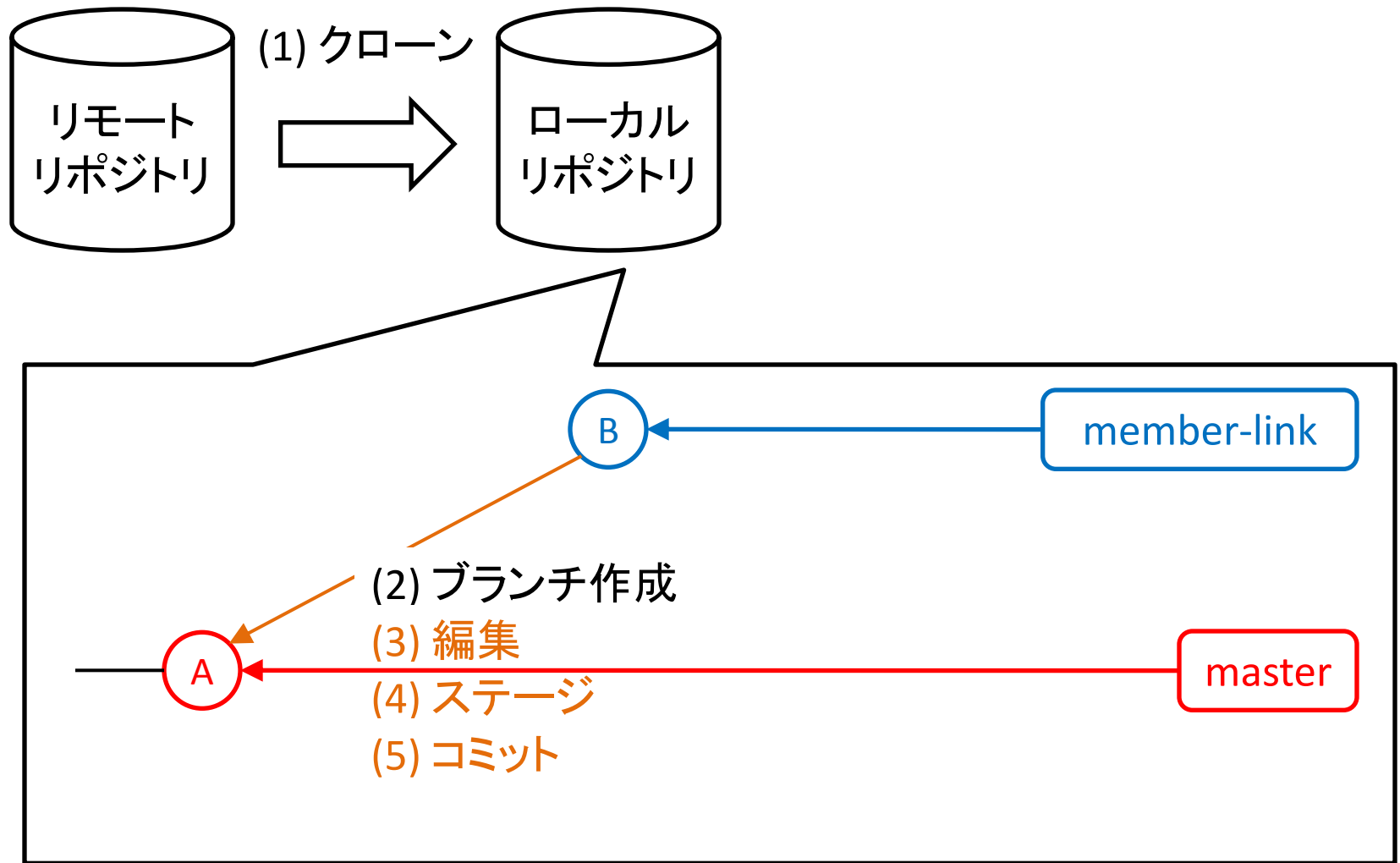


# 本日の処理流れ

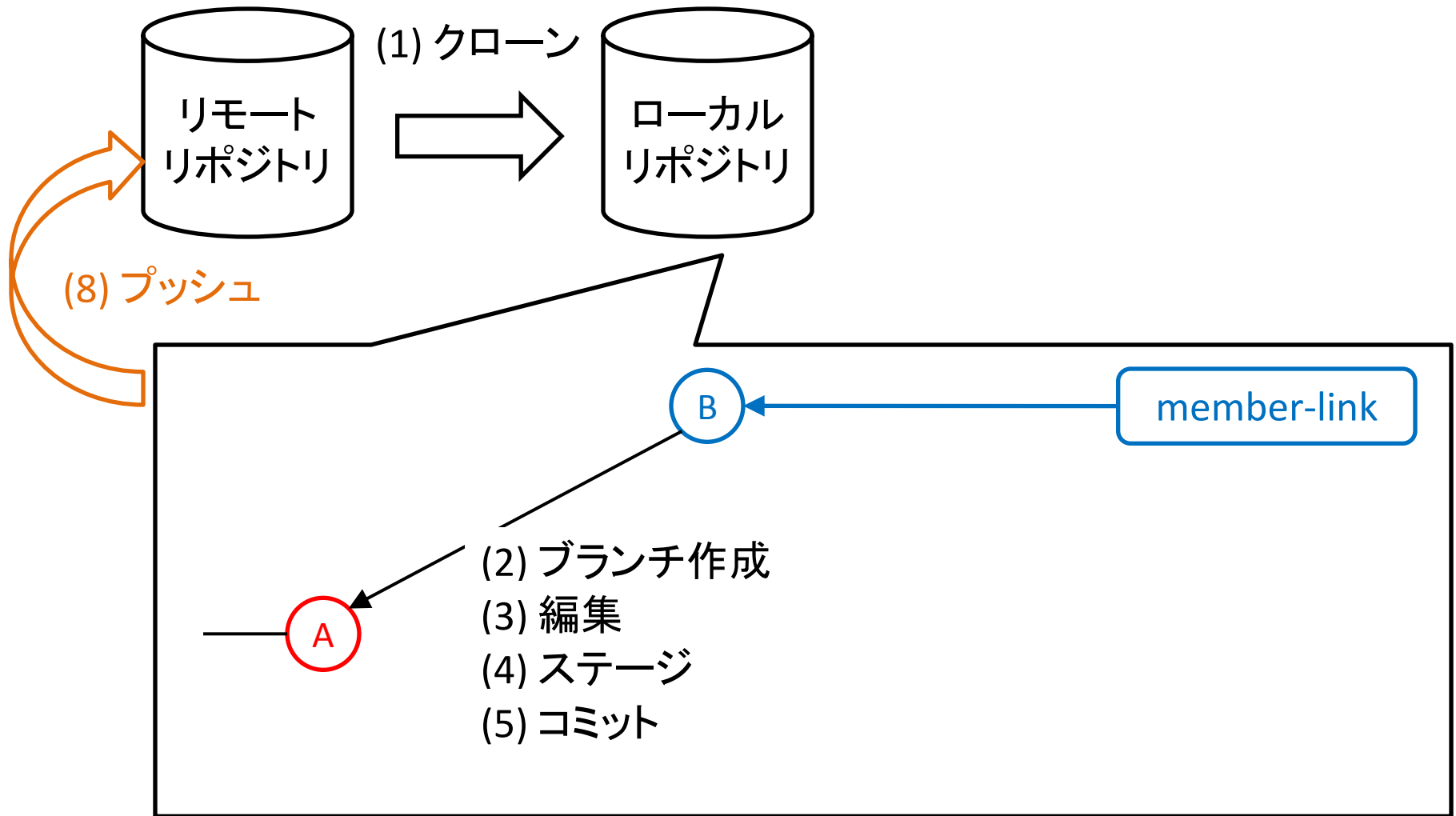




# 本日の処理流れ



# 本日の処理流れ



# git clone

## リポジトリの複製

### 例

\$ git clone gitosis@redmine.swlab.cs.okayama-u.ac.jp:nomnichi  
(複製するリポジトリのURL) nomnichi(作成するディレクトリ名)

※ ディレクトリ名は省略可能  
(省略した場合は自動的にディレクトリを作成)

### <オプション>

--bare オプションをつけるとベアリポジトリを作成

※ ベアリポジトリ: ワーキングディレクトリを持たないリポジトリ  
マスタリポジトリにはベアリポジトリを使用

# git log

## コミットログの表示

例

```
$ git log
```

```
commit 32cf85f18fce5c9b4d03d01b28e5f2d289e67618
Author: ikeda-n <ikeda-n@swlab.cs.okayama-u.ac.jp>
Date: Thu Apr 5 11:29:16 2012 +0900
```

平成24年度Newグループ新人研修課題の追加

```
commit ec61639c9558cd8ee53a097da3c6dfe44ec93375
Author: Yoshii Hideto <yoshii@swlab.cs.okayama-u.ac.jp>
Date: Tue Apr 3 18:29:19 2012 +0900
```

個人ページの更新(吉井)

⋮

コミットID

コミット者

タイムスタンプ

コミットメッセージ

## <オプション>

--graph オプションをつけるとマージの様子をグラフィカルに確認

# git branch

## ブランチの作成もしくは一覧表示

### <ブランチの作成>

例 `$ git branch member-link`(作成するブランチ名)

member-linkという名前のブランチを作成

### <ブランチの一覧表示>

例 `$ git branch`  
カレントブランチにはアスタリスクが表示される

git branchの後に何も入力しなければブランチの一覧を表示

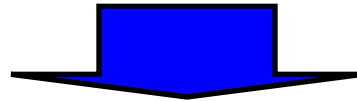
### <オプション>

-rオプションをつけるとリモートブランチを表示

# git checkout

## ブランチの切り替え

ワーキングディレクトリの更新内容を反映できるのは、単一のブランチのみ



他のブランチで作業を始める場合、ブランチを切り替える必要

例 `$ git checkout member-link`(切り替え先ブランチ名)  
Switched to branch 'member-link'

member-linkブランチに切り替え

### <オプション>

-bオプションをつけるとブランチを作成して、作成したブランチに切り替え

例 `$ git checkout -b member-link`(作成し、切り替えるブランチ名)

# git status

## 変更されたファイル一覧の表示

例

```
$ git status
# On branch member-link
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   public_html/members.html
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       config/database.yml
#       db/development.sqlite3
#       db/large_objects/
#       log/
```

カレントブランチ名

ステージされているファイル  
git commit実行時にコミットする

ステージされていないファイル  
git commit実行時にコミットしない

# git diff

ファイルに加えられた変更点の表示

## <基本的な5つの比較>

(1) `git diff`

ワーキングディレクトリとインデックスの比較

(2) `git diff --cached`

インデックスとHEADの比較

(3) `git diff "commit"`

ワーキングディレクトリと指定したcommitの比較

(4) `git diff --cached "commit"`

インデックスと指定したcommitの比較

(5) `git diff "commit1" "commit2"`

指定した2つのcommitの比較



# git add

ファイルをステージ(インデックスに追加)

例 `$ git add public_html/member.html`

public\_html/member.htmlをステージ

ファイルをコミットするためには, ファイルを編集する度にgit addを実行する必要がある

※ ディレクトリに対して行くと, ディレクトリの中の全てのファイルとサブディレクトリを再帰的にステージ

例 `$ git add .`

カレントディレクトリ以下のファイルを全てをステージ

# git commit

インデックスに蓄積された変化をリポジトリにコミット

例 `$ git commit`

自動的にエディタが起動

# がついていない行にコミットメッセージを入力



```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch member-link
# Changes to be committed:
#   (use "git reset HEAD^1 <file>..." to unstage)
#
#       modified:   public_html/members.html
#
```

<オプション>

例 `$ git commit -m "message"`

message の部分にコミットメッセージを入力

# git push

オブジェクトとこれに関連したメタデータをリモートブランチに転送

例 `$ git push origin(プッシュ先リポジトリ) master(プッシュ元ローカルブランチ)`

- プッシュ先リポジトリとプッシュ元ローカルブランチを指定
- git pushのみの場合は、ローカルのmasterブランチをoriginのmasterブランチにプッシュ

※ プッシュ元ローカルブランチがファストフォワードでないとプッシュできない

ファストフォワード: ローカルブランチにリモートブランチのコミットが全て含まれている状態

## <オプション>

--force オプションをつけると強制的にプッシュ可能

※ 基本的にこのオプションは使用しないこと

# git merge

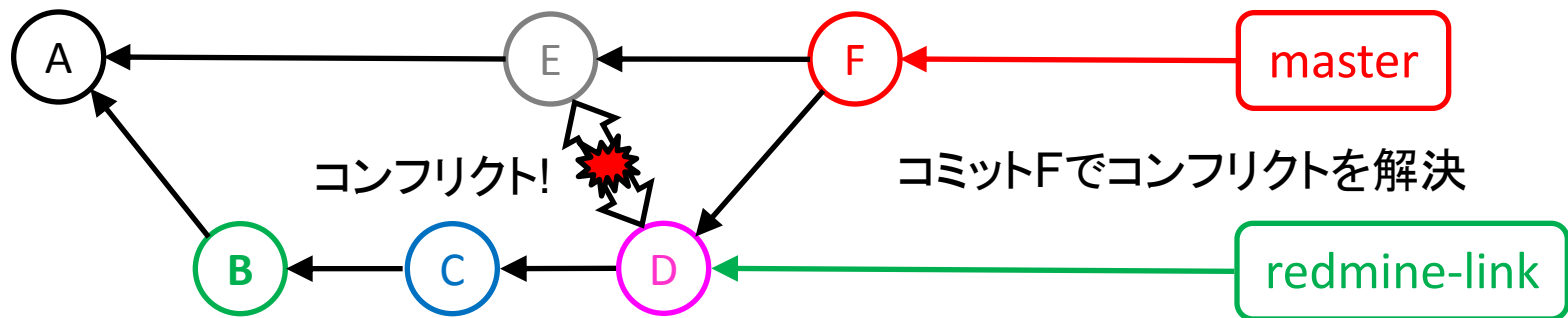
2つ以上のブランチを1つにマージ(統合)

例 `$ git merge redmine-link(マージしたいブランチ名)...`

現在のブランチに指定したブランチのコミットをマージ

ファストフォワードの場合は, ブランチが指し示すコミットが変更されるだけで新しいコミットは作成されない

※ コンフリクトが発生する可能性あり



# git fetch

リモートリポジトリの変更を取得

例 `$ git fetch`

リモートトラッキングブランチを更新

リモートブランチの変更点をローカルブランチにマージするには,  
`git merge`を使用

※ `git pull`を使用すれば`git fetch`と`git merge`を同時に行うことが可能

＜redmine-linkブランチをプルしたい場合＞

例 `$ git pull origin(プルしたいリモートリポジトリ) redmine-link(プルしたいブランチ)`

クローン作成元のリポジトリはデフォルトで`origin`という名前が割り当てられる

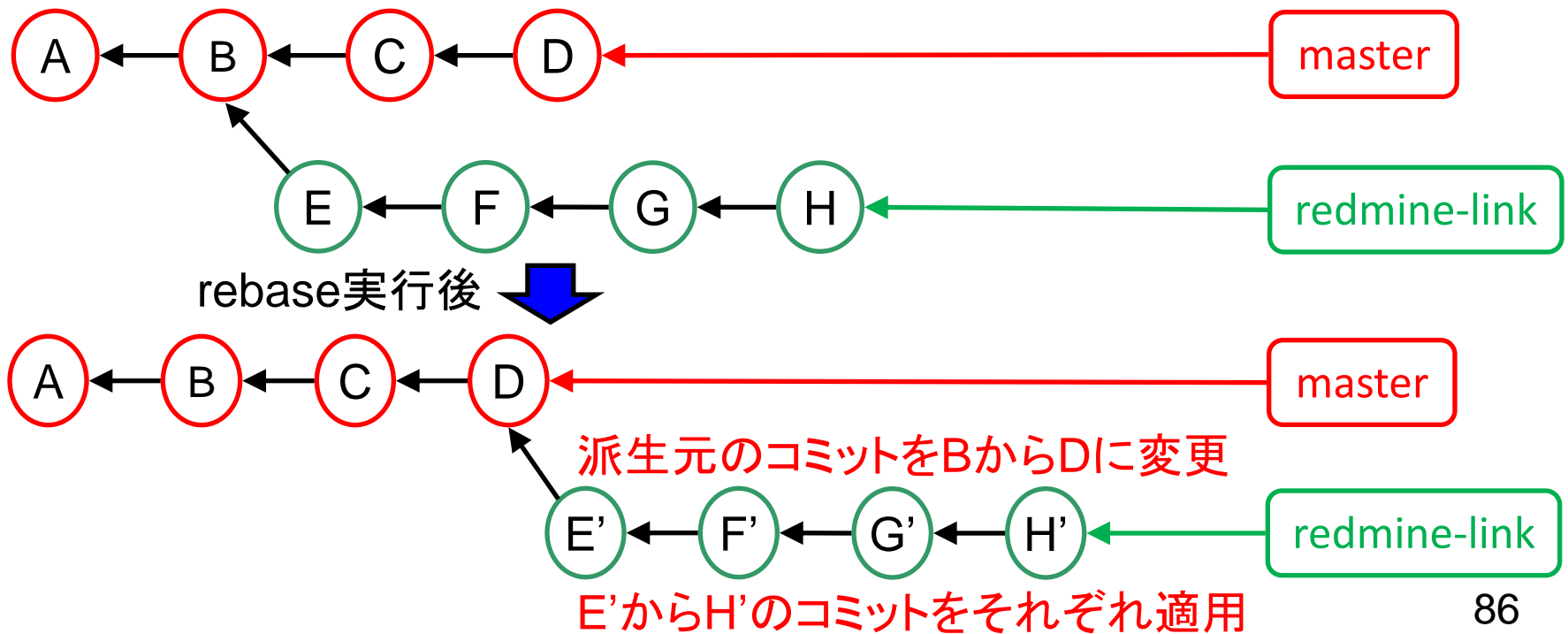
# git rebase

## ブランチの派生元(上流)を変更

例 `$ git rebase master(redmine-link(派生先ブランチ名))`

派生先ブランチをファストフォワードにする

< `$git rebase master redmine-link` の例 >



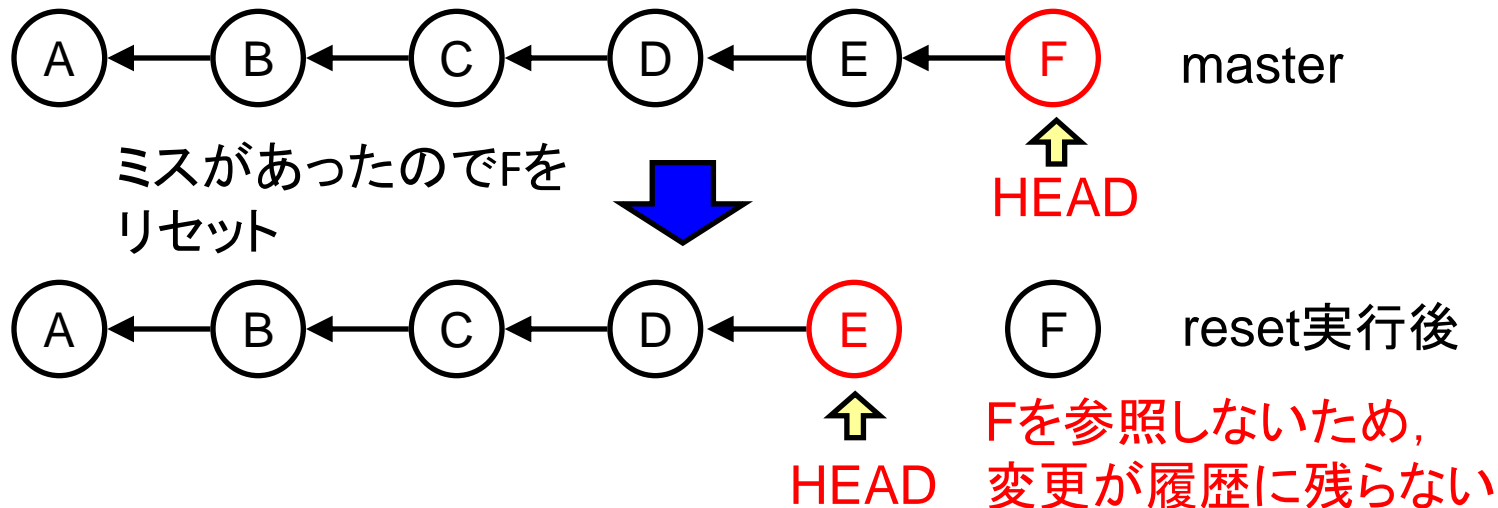
# git reset(1/2)

リポジトリと作業ディレクトリを特定の状態(コミット)に変更  
(HEADの指すコミットを変更)

例 `$ git reset e203f6`(コミットID等)

- コミットIDを指定しない場合はHEADを指定
- コミットIDの修飾子として^と~を利用可能

<\$ git reset master~ とした時の例>



# git reset(2/2)

リポジトリと作業ディレクトリを特定の状態(コミット)に変更  
(HEADの指すコミットを変更)

例 `$ git reset e203f6`(コミットID等)

- コミットIDを指定しない場合はHEADを指定
- コミットIDの修飾子として^と~を利用可能

## <オプション>

3種類のオプションが存在(指定しない場合は--mixed)

Option	HEAD	Index	Working directory
--soft	○	×	×
--mixed	○	○	×
--hard	○	○	○

○: 変更を適用, ×: 変更を適用しない



# Git勉強会用スライド (Git練習編)

全員

# Gitの練習

## (1) 乃村研

ノムニチの個人ページの作成を通して, Gitの勉強を行う  
リポジトリのURL

`gitosis@redmine.swlab.cs.okayama-u.ac.jp:nomnichi`

## (2) 谷口研

システムコールの追加を通して, Gitの勉強を行う  
リポジトリのURL

`ssh://newgroup.swlab.cs.okayama-u.ac.jp/var/git/TwinOS26.git`