GIT応用

Ver.202010
作成者：伊賀　将之

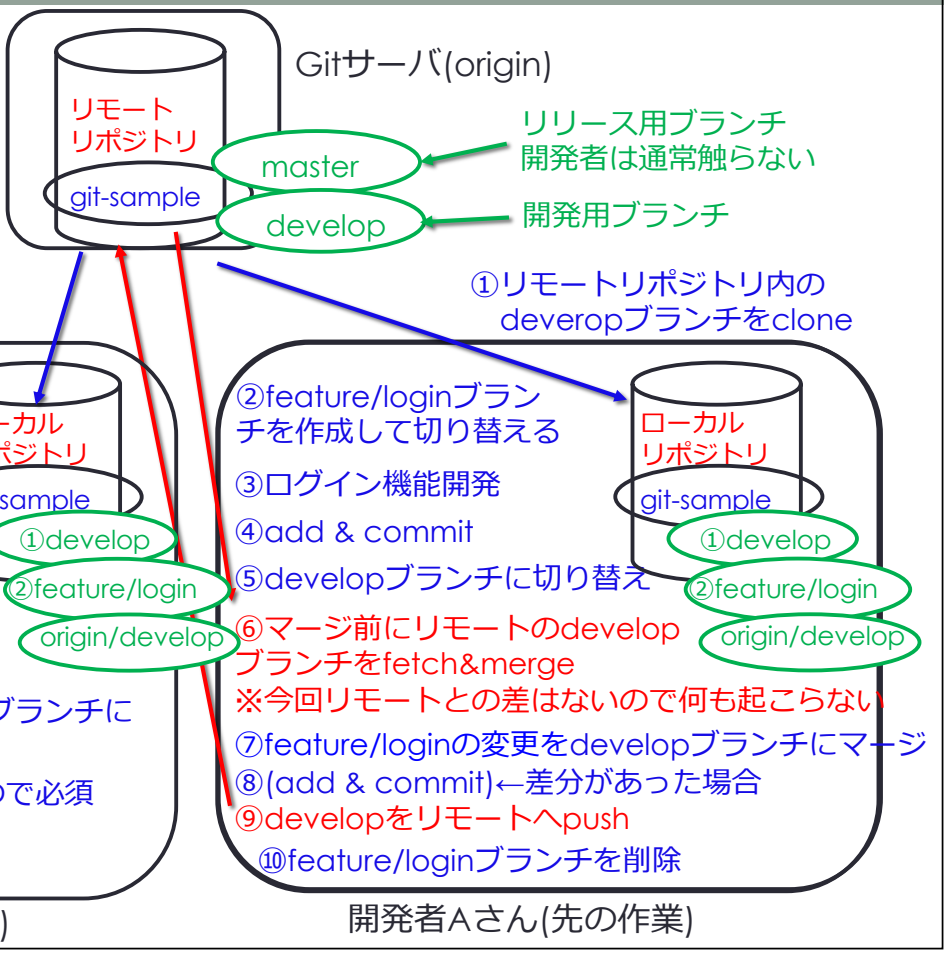
第4章
GITでチーム開発

本章の目的

・チーム開発を行う際のGitの使い方を理解する

ブランチとは、本流の開発を邪魔することなく作業を続ける機能(領域)のこと

マージとは、自分のブランチのPGを他のブランチのPGと統合すること



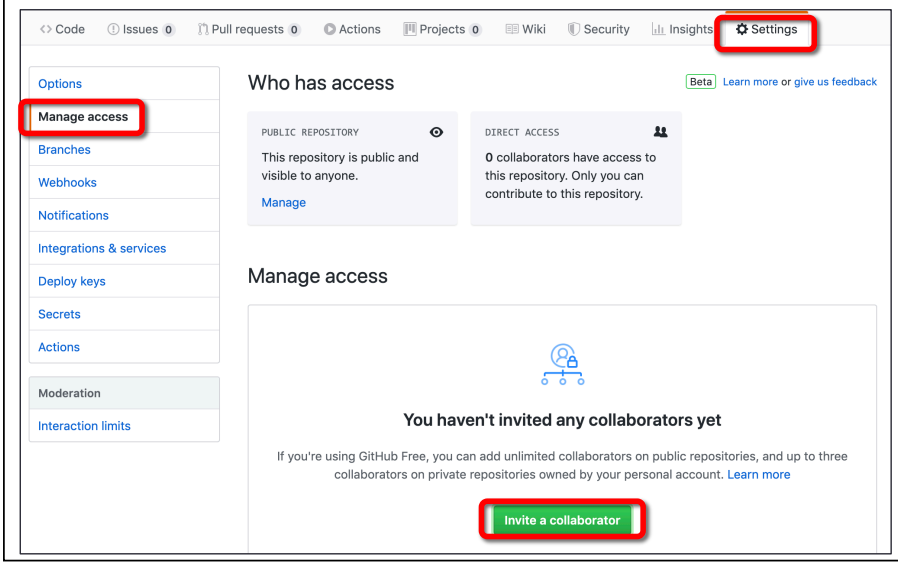
最初の手順(2人1ペアになり、AさんBさんを決めます)
※サポートされたい人同士のペアは避けます

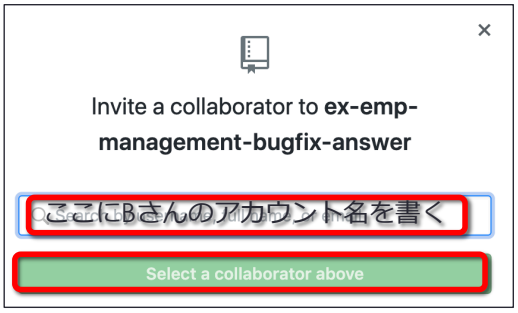
- Aさんが、新規にリモートリポジトリを作成する
- Aさんが、Eclipse(またはSTS)にてプロジェクトを作成する
- Aさんが、今作成したプロジェクト内でgit bushを起動し、git initを実行する
- Aさんが、プロジェクトに１つのクラスを作成する(Carクラスなど)
  ※同時にsrc/main/resources/staticに中身が空のindex.css
  src/main/resources/templatesに中身が空のindex.html　を作って入れておく
  ディレクトリ内が空の場合はpushされないため
- Aさんが、今作成したプロジェクトを(add & commit後)、リモートリポジトリにpushする
- Aさんが、リモートリポジトリでdevelopブランチを作成する

最初の手順

- Aさんは、「cd ..」をした後、今作成したローカルのプロジェクトをワークスペース上から削除する(あとでもう一度cloneしてくるため)

- Aさんは、GitHub上でCollaboratorsにBさんを追加する





最初の手順

- Bさんは、承認依頼メールがくるので承認する
(これでBさんがAさんのリポジトリにpushできるようになります)

- ２人ともc:¥env¥springworkspaceをエクスプローラで開き、Git Bashを起動する

AさんBさんが並行してやる作業

- ①リモートリポジトリのdevelopブランチをclone
  - git clone **-b develop** git@github.com:igamasayuki/git-sample-iga.git (URLはGitHubからコピーしてきます)
  - cd git-sample-iga　←cloneしてきたディレクトリの中へ移動する
  - git branch　←branchの確認(developであることを確認する)
    Eclipse(STS)で、cloneしてきたプロジェクトをインポートする
- ②branchの作成と切り替え
  - git branch feature/login　←branchの作成
  - git branch　←branchの確認 (feature/loginが作成されたことを確認する)
  - git switch feature/login　←branchの切り替え
  - git branch　←branchの確認し切り替わっていることを確認する
  - ※ブランチの作成と切替を一気に行う方法
    - git switch –c feature/login

AさんBさんが並行してやる作業

- ③ログイン機能開発
  - 実際に割り当てられた担当分の機能開発を行う
    - ※この際、AさんとBさんは同じ行を修正すること。
      例えばCarクラスのprivate int speed 部分など)
  - git status　←ファイルのステータスを確認
- ④add & commit
  - git add –A
  - git commit
- ⑤developブランチに切り替え
  - git switch develop
  - git branch　←切り替わったことを確認

Aさんの作業(先にやる)

- Bさんは作業をストップしてください
- ⑥リモート(origin)のdevelopから最新を取得(fetch)し、ローカルのdevelopに統合(merge)
  - git fetch origin develop ←リモートのdevelopから**ローカルのorigin/developブランチ**に最新を取得(マージするローカルのdevelopブランチで行う)
  - git merge origin/develop ←リモートから取得した最新を自分のdevelopに統合(merge)
  - ※上記２行は右の１行と同じ　git pull origin develop
  - **※今回はリモートの差分が無いため競合が起こらない**
- ⑦feature/loginの変更をdevelopにマージ
  - git merge feature/login ←マージするdevelopブランチで行う
  - git log ←ログを確認

Aさんの作業(先にやる)

- ⑧(add & commit)←今回は競合が発生しないため必要ない
- ⑨developをリモートへpush
  - git push origin develop
  - ※他の人が先にpushしていたらもう一度fetch&mergeしてからpush
  - Github上にきちんと追加(or修正)したものが上がっているかも確認すること！
- ⑩feature/loginブランチを削除
  - git branch -d feature/login
  - git branch←削除されたか確認
  - ※複雑になるため、慣れないうちは開発を終えたブランチは必ず削除し、使いまわさないでください

Bさんの作業(後にやる)

- Aさんの後にBさんが行う手順

- ⑥リモート(origin)のdevelopから最新を取得(fetch)し、ローカルのdevelopに統合(merge)
  - git fetch origin develop ←リモートのdevelopから**ローカルのorigin/developブランチ**に最新を取得(マージするローカルのdevelopブランチで行う)
  - git merge origin/develop ←リモートから取得した最新を自分のdevelopに統合(merge)
  - ※上記２行は右の１行と同じ　git pull origin develop
  - **※Aさんの開発した差分が取り込まれる**

Bさんの作業(後にやる)

- ⑦feature/loginの変更をdevelopにマージ
  - git merge feature/login ←マージするdevelopブランチで行う
  - **※ここで競合が発生するのでBさんが競合を修正する**

```
伊 賀 将 之 @DESKTOP-RBFQOJQ MINGW64 /c/env/springworkspace2/0.test (develop)
$ git merge feature/login
Auto-merging src/main/java
CONFLICT (content): Merge conflict in src/main/java/com/example/Exam01Controller.java
Automatic merge failed; fix conflicts and then commit the result.
```

CONFLICTと表示されたら競合(コンフリクト)発生したという意味

Exam01Controller.java

```
@Controller
@RequestMapping("/exam01")
public class Exam01Controller {

    @RequestMapping("")
    public String init() {
<<<<<<< HEAD
        System.out.println("Macで修正");
=======
        System.out.println("Windowsで修正");
>>>>>>> feature/login
        return "exam01";
    }

    @RequestMapping("/input")
    public String input(String name, Model model) {
<<<<<<< HEAD
        System.out.println("Macで修正");
=======
        System.out.println("Windowsで修正");
>>>>>>> feature/login
        model.addAttribute("name", name);
        return "exam01-result";
    }

}
```

Aさんの修正

Bさんの修正

CONFLICT(競合)を発生させたBさんがAさんと何が正しいかを話し合い修正する

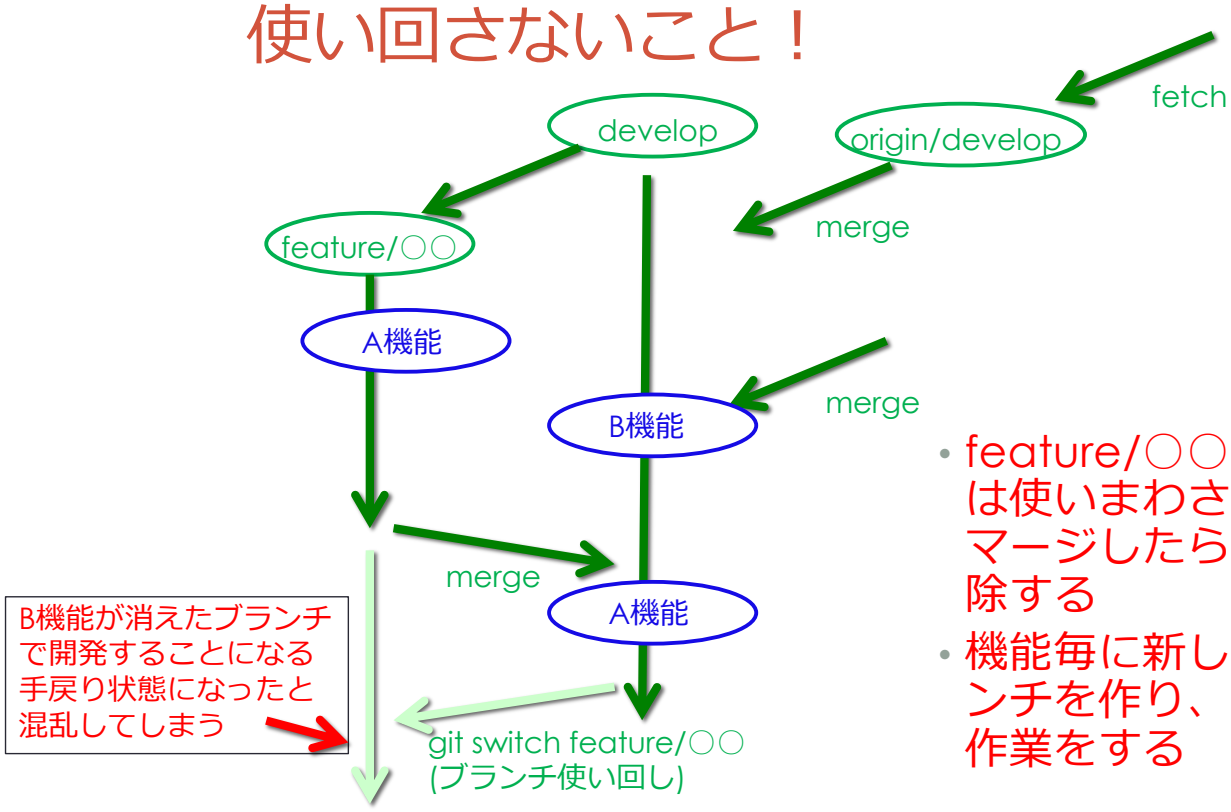
Exam01Controller.java

```
package com.example;

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.Reques

@Controller
@RequestMapping("/exam01")
public class Exam01Controller {

    @RequestMapping("")
    public String init() {
        System.out.println("MacとWindowsで修正");
        return "exam01";
    }


    @RequestMapping("/input")
    public String input(String name, Model model) {
        System.out.println("MacとWindowsで修正");
        model.addAttribute("name", name);
        return "exam01-result";
    }

}
```

Bさんの作業(後にやる)

- ⑧add & commit(競合を修正したため必要)
  - git add –A
  - git commit
- ⑨developをリモートへpush
  - git push origin develop
  - ※他の人が先にpushしていたらもう一度fetch&mergeしてからpush
  - git log
  - Github上にきちんと追加(or修正)したものが上がっているかも確認すること！
- ⑩feature/loginブランチを削除
  - git branch -d feature/login
  - git branch -a←削除されたか確認
  - ※複雑になるため、慣れないうちは開発を終えたブランチは必ず削除し、使いまわさないでください
- **後は本日やった内容を2人で一緒に何回も行いGitでのチーム開発に慣れる**

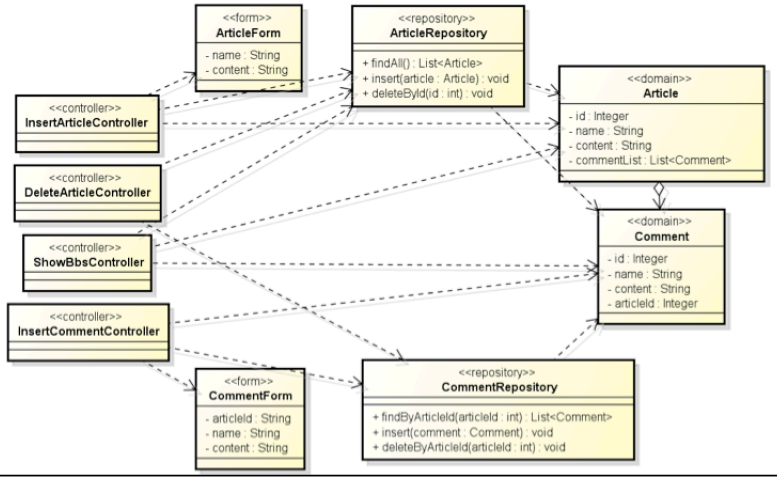
(参考)feature/○○ブランチは使い回さないこと！



- feature/○○ブランチは使いまわさずに、マージしたら必ず削除する
- 機能毎に新しいブランチを作り、そこで作業をする

(参考)競合を起こす確率を減らすために

- チームのメンバーが近くにいる時は、pushしたい人は「今からfetch(またはpull)/pushします！」と声をかけて、自分がpushするまで他の人にpushするのを待ってもらう
- コントローラを**ユースケース毎に作成**し、一つのクラスの中に書くプログラムを１つの役割のみにする
  - ※ドメインやリポジトリは今まで通り1テーブルにつき1クラス
  - ※(参考)この方針に基づいた掲示板のクラス図は以下の通り



総合演習

- ２人ペア(グループよりも2人ペアがお勧め)で前回作成した掲示板を作成してください
- この際、前のスライドにある「クラス図(ユースケースごとに１つのコントローラ)」が競合が起こりにくい設計になっているのでそれを参考にします
- コーディング技術の向上ではなく、gitでのチーム開発に慣れることが目的のため、一番複雑なRepository内のコードやapplication.ymlは以前作成したものからコピーしてもってきてOKです

(補足) rebase コミットを繋げ直す

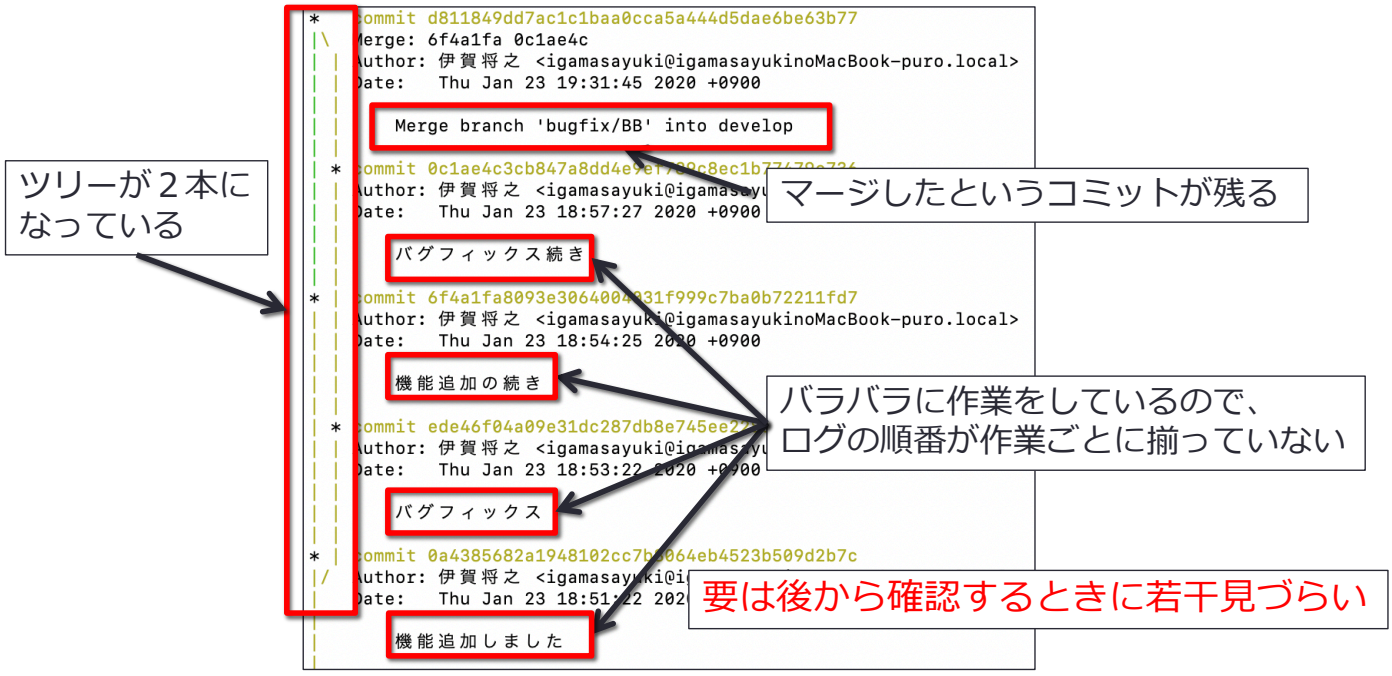
- 「○○機能追加」中に「バグの修正依頼」がきた場合、一般的には以下の２つのブランチに分けて対応します。
  - 機能追加用のブランチ　(feature/XX)
  - バグ修正用のブランチ　(bugfix/XX)
- この２つのブランチをdevelopブランチに統合する場合、今まではgit mergeコマンドを使用していました
- しかしgit rebaseを使用してもgit merge同様に統合することができます
- ここではgit mergeではなく、git rebaseを使用した場合の動きについてご紹介します

(補足) rebase コミットを繋げ直す

- 前提：以下の手順で開発を進めたとします
  - feature/XXで機能追加を途中まで実施してcommit
  - bugfix/XXでバグフィックスを途中まで実施してcommit
  - feature/XXで機能追加を最後まで実施してcommit
  - bugfix/XXでバグフィックスを最後まで実施してcommit
  - 最後にdevelopブランチでfeature/XX→bugfix/XXの順で git merge または git rebase
  - git log --graph --date-orderでグラフ付きのログを表示してログを確認
- 次のスライドからgit mergeとgit rebaseで統合した場合の違いを解説します

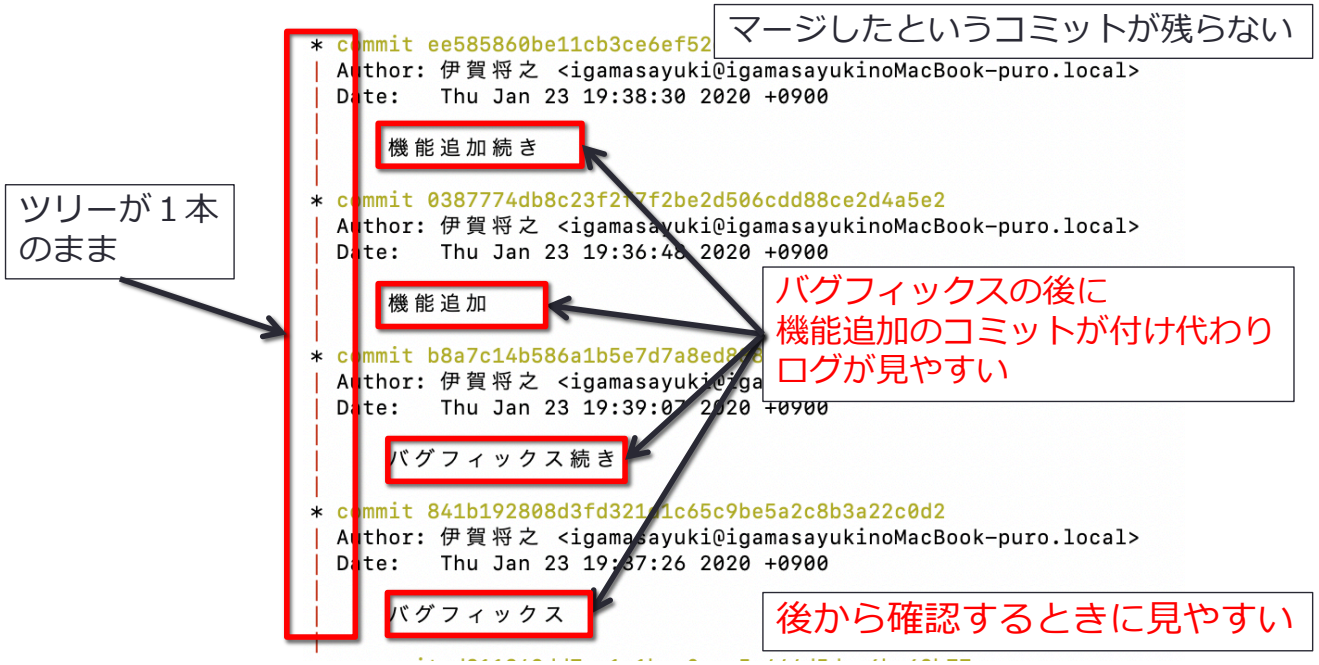
(補足) rebase コミットを繋げ直す

- git mergeで統合した場合のログ



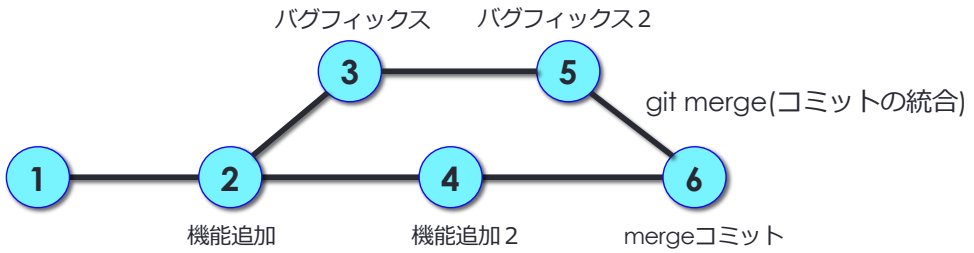
(補足) rebase コミットを繋げ直す

- git rebaseで統合した場合

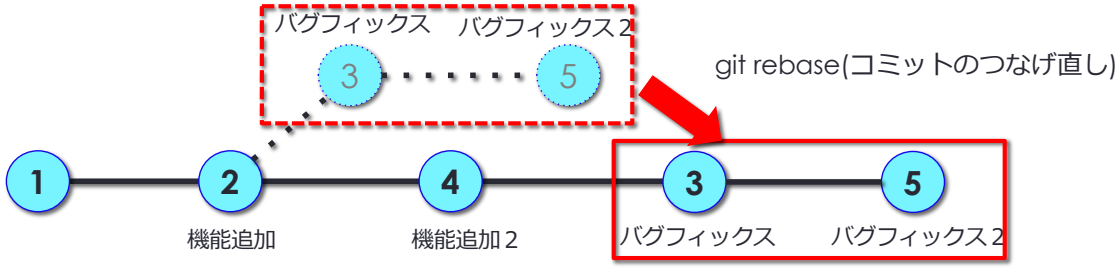


(補足) rebase コミットを繋げ直す

- mergeのイメージ



- rebaseのイメージ



(補足) rebase コミットを繋げ直す

- rebaseはいつどのような時に使うのか？
  - タイミング：任意のbranch上で、リモートブランチにpushする以前の開発中
  - 用途：コミットグラフの整理
- mergeの代わりではダメなのか？
  - rebaseは必須ではなく、mergeでもよい。
  - rebaseを利用することで、mergeと比較してコミットグラフの可読性が上がる。
- 注意点！
  - ただしリモートリポジトリに一度pushしたbranchの場合、rebaseは利用不可。
  - →rebaseを利用すると既存のコミットを破棄して新しいコミットを作成します。もし既存のコミットを誰かが取得してそれを利用して開発を続けた場合、自分がgit rebaseでコミットを書き換えて再pushするととてもややこしくかつおかしなことになってしまうため