

0 から学ぶ

# Git と共同開発



Author/Speaker 先進計算力学研究室

鈴木 祐太

Contributor 数理システム設計学研究室

高橋 和真



## ことわり

- コマンド由来のものや用語はあえて英語で書いていることがあります  
表記ゆれが気になる場合は教えてください
- ターミナルは見やすさのために変わった表示をすることがあります

ターミナルの表現

```
~/repository  
> git --version  
git version 2.34.1
```

表しているもの

```
カレントディレクトリ  
> コマンド  
出力
```

ただ、必要がないときには簡略化しています

- 何度も同じ話をします。わざとです
- ハンズオン形式で同時に作業をすることがあります  
ターミナル（シェル）はそれなりに使える想定です
- Git がインストールされ、GitHub のアカウントにログインできることを確認してください

## はじめに

- Git は仕組みが複雑で挫折するひとも多い
  - わかりやすさのために嘘が書かれていることがある
  - このゼミは可能な限り正しい説明を心がけていて、少し難しい
  - 全てを 2 回のゼミで覚える必要はない
- ゼミ中は好きに発言して OK
  - わかった、わからない、自分の解釈が正しいか確認したいなど
- 発言しにくい場合は Slack や Slido で投げる（できるだけ見ます）
- とにかく積極的に使ってみる！
- 最終課題は Git 管理してみよう

## TABLE OF CONTENTS

---

第Ⅰ部	個人開発を管理しよう
第Ⅱ部	共同開発を管理しよう

## 第Ⅰ部

# 個人開発を管理しよう

### TABLE OF CONTENTS

---

- 01. Git 導入のモチベーション
- 02. Git/GitHub/GitLab 管理の流れ
- 03. Git コマンド 〈超基本編〉
- 04. Git コマンド 〈基本編〉
- 05. Git を試してみよう


## 第Ⅱ部

# 共同開発を管理しよう

# 始める前にまずは結論（絶対に覚える）

-  **git** でバージョン（履歴）管理、 **GitHub** ・  **GitLab** でバックアップ/共有することができる

  
自分の PC（ローカル）

 どっかのサーバー/インターネット（リモート）

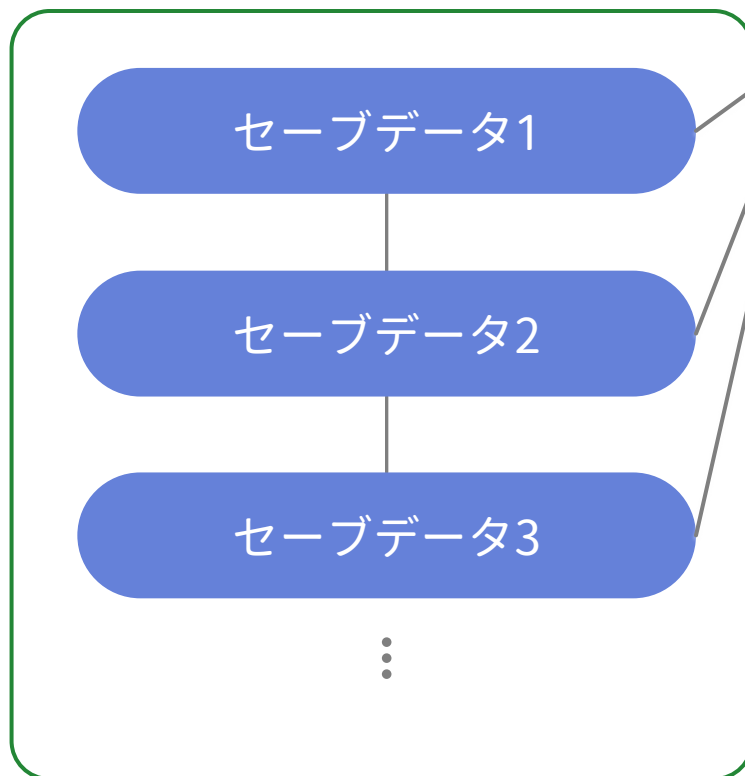
- 履歴はすべて **.git ディレクトリ**に入っている
- リモートをメインに管理してローカルで変更、リモートに反映
- 必須コマンド
  - add : 履歴を記録する準備
  - commit : 履歴を記録する
  - push : GitHub ・ GitLab に履歴を送信する
  - pull : GitHub ・ GitLab から履歴を持ってくる
- 明示的にコマンドを実行することでのみ、**履歴を保存しバックアップ**が取られる
  - ▶ 自動では保存されない

# Git とは？（わかりやすさ重視）



コードの履歴を管理するツール

これを記録してくれるのが Git



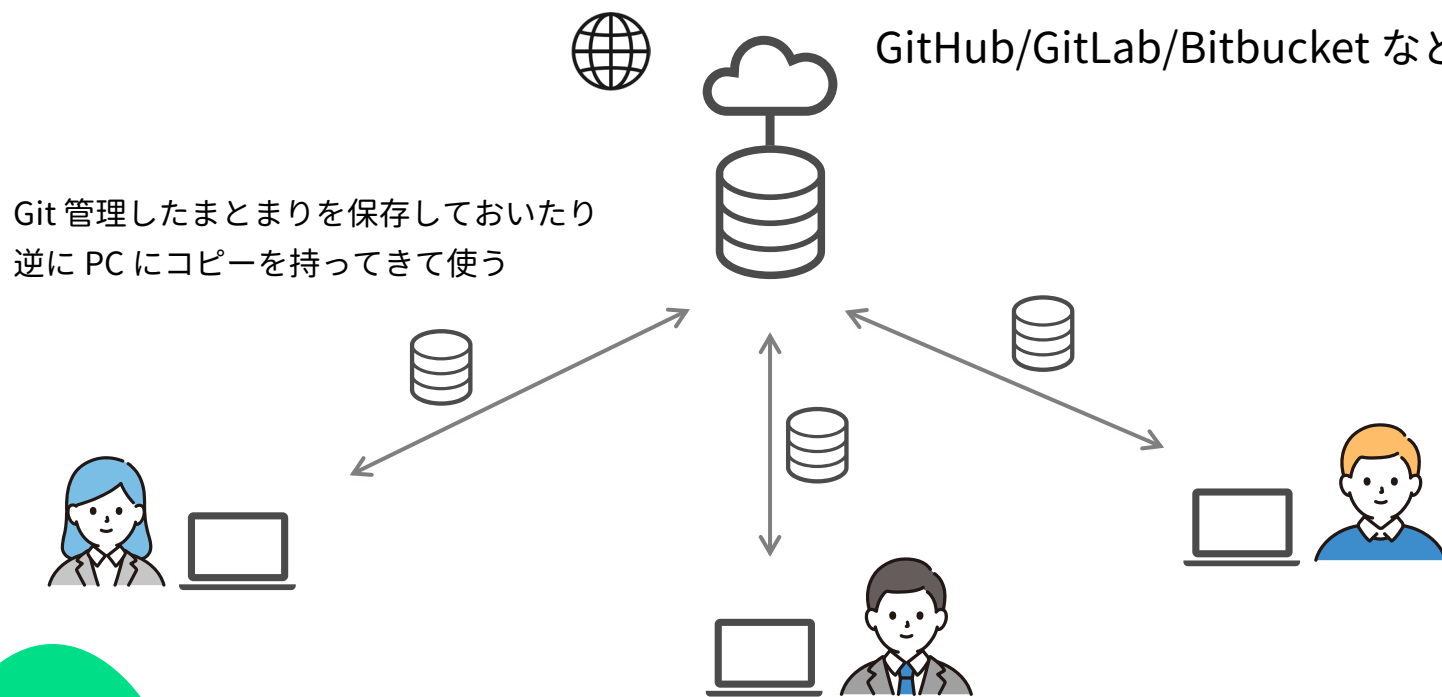
Git では **commit** という



# GitHub とは？（わかりやすさ重視）



Git 管理のバックアップを取るためのツール  
他の人と一緒に編集できる



GitHub/GitLab/Bitbucket などはこっちを指す（リモート環境）

Git 管理したまとまりを保存しておいたり  
逆に PC にコピーを持ってきて使う

同じ場所にバックアップを取るので  
変更が競合することがある（Conflict/コンフリクト）

→ 適切な手順で解消することで  
いい感じに統合（Merge/マージ）できる

自分だけ独立して編集（Branch/ブランチ）してから  
完成後にバックアップを取る

→ 競合のタイミングを  
コントロールできる

自分の PC 側は Git の管理領域（ローカル環境）



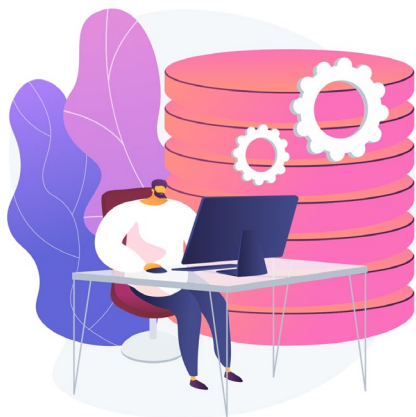
# Git とは？

分散型バージョン管理システム (DVCS: Distributed Version Control System)

## バージョン管理

変更を commit（記録）することによって、過去の状態を保持することができる。

また、branch を用いて作業の種類ごとに分離して開発が可能。



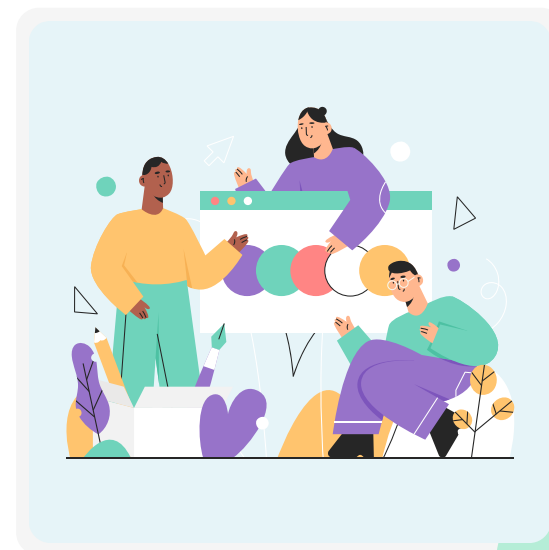
## 状態の復元

変更を誤って削除した場合や、なんらかの理由でコードが動かなくなった場合など、過去の状態を確認したいときに、過去の commit 時に戻ることができる。



## 共同開発

branch によって複数の人が自分の作業が他の人の変更に影響されず、最終的には変更を merge（統合）することができる。



# バージョン管理とは？

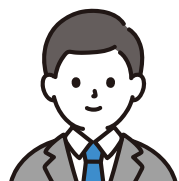
変更の履歴を記録すること

もっとも原始的なバージョン管理は変更したファイルを変更のたびに保存しておくこと



作成する

おはようございます  
こんにちは



編集する

おはようございます  
こんにちは  
こんばんは



挨拶.txt



挨拶\_修正版.txt

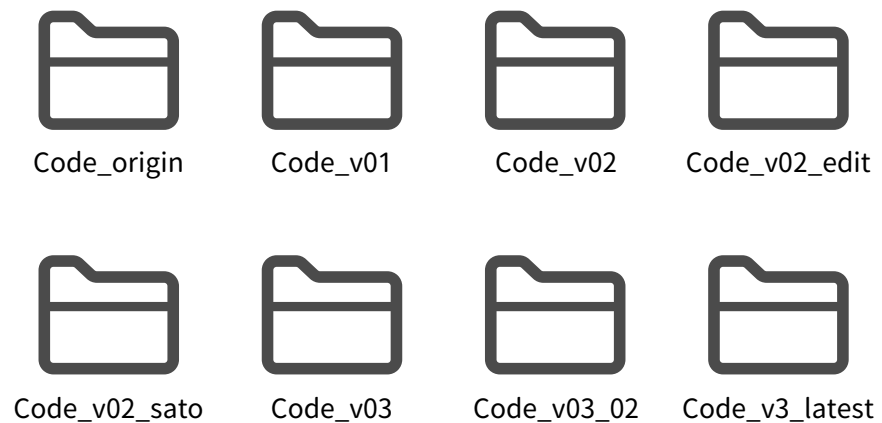


# Git を使うと...

repository（リポジトリ）単位で管理する



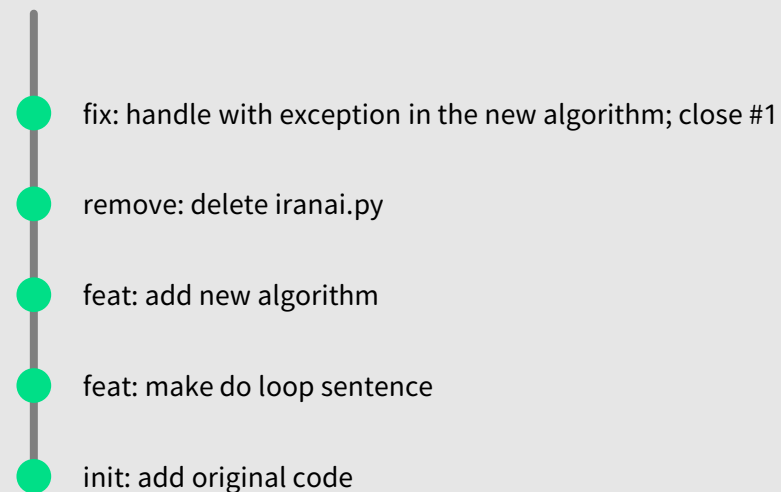
## Git 導入前



repository



## Git 導入後



- ✓ 履歴がひとつのディレクトリにまとまる
- ✓ いつ何の変更をしたかを残せる（commit message）



Code

# ● なんのための Git ?

ファイルのバックアップを取っておく

ファイルを間違えて消してしまった

コードを改良しようとしたが動かなくなった

▶ 復元できる！！

書いたコードを共有する

コードを共同で編集することができる ▶ 第II部 共同開発を管理しよう

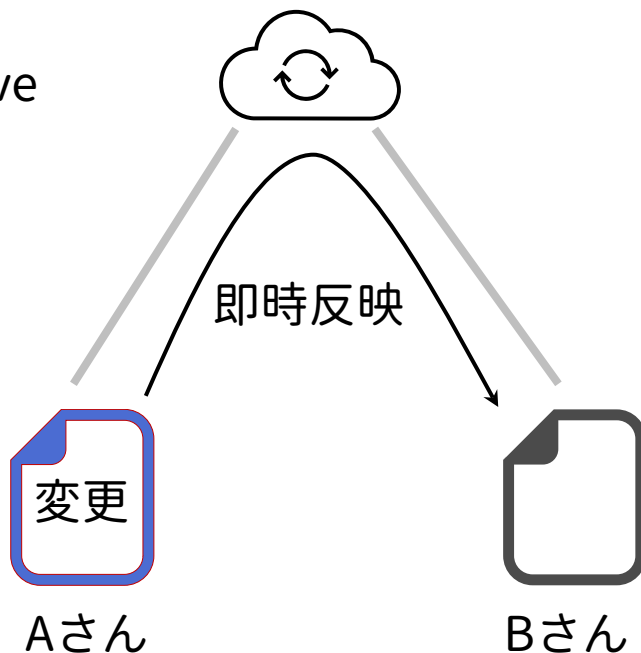
コードの編集の記録が残るので、引き継ぎでも意図を汲み取りやすい

読みにくいのは他人のコードだけでない

▶ 昨日の自分は赤の他人

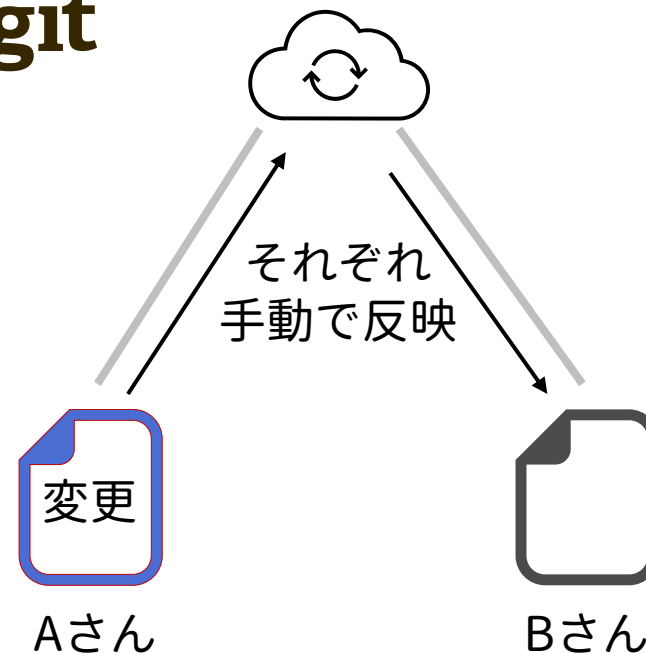
# なぜ Git ?

OneDrive



変更がされ次第随時反映される (Auto)

git



手順を踏まない限り変更は反映されない (Manual)

## Git の利点

- 他人の変更が開発中は反映されない
- コンフリクトの精査が可能である
- 変更の説明を書ける

# ● Git 勉強について

勉強に良いサイトなどをGitHubとかにまとめて公開してリンクを貼ります

とりあえずリポジトリを作りました

<https://github.com/suzuyuyuyu/git-seminar>

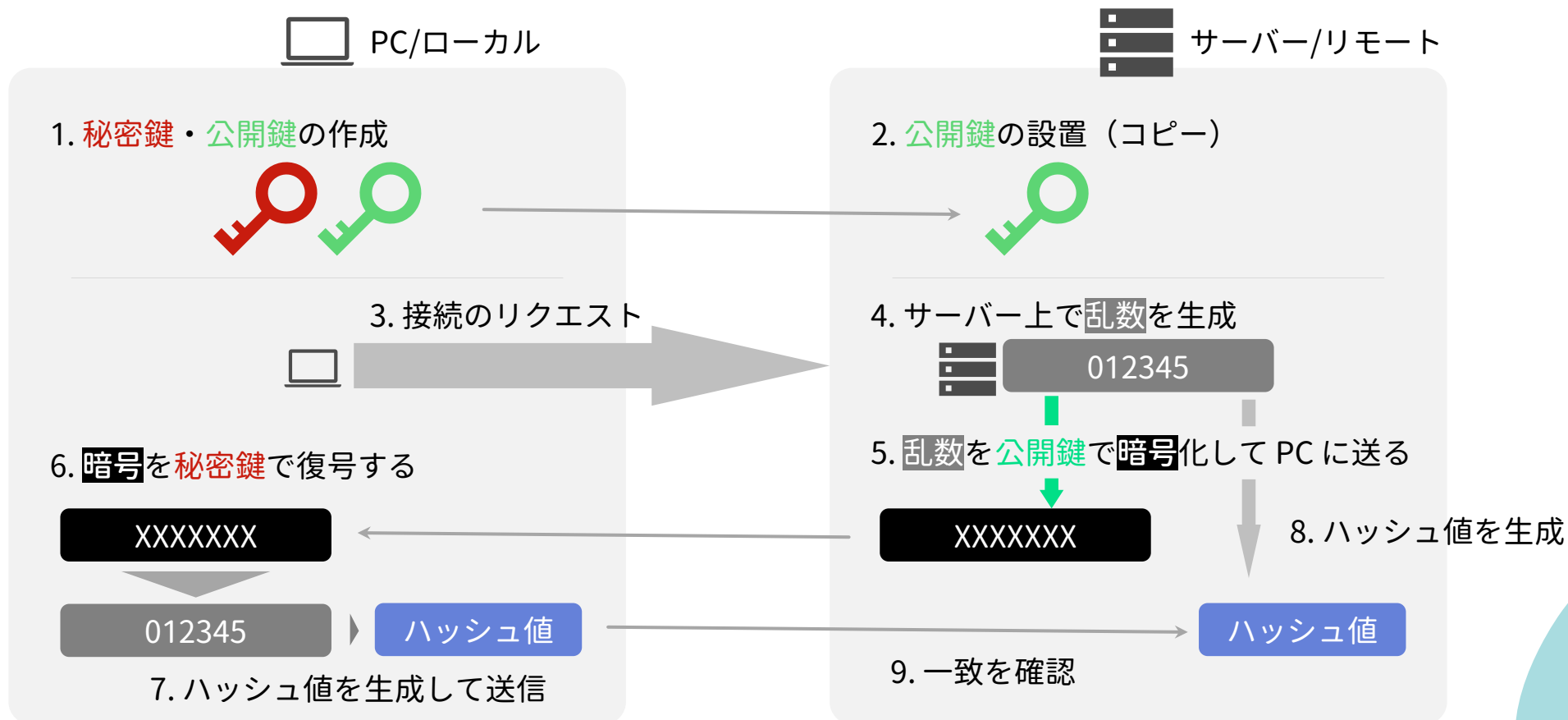
# SSH の設定をしよう

GitHub を使うためには SSH (Secure SHell) 接続の設定をするのが好ましい

SSH

パスワード認証方式と公開鍵認証方式がある

GitHub や GitLab は公開鍵認証方式によっても push や pull が可能



# SSH の設定をしよう

## 秘密鍵と公開鍵の生成

```
mkdir -p ~/.ssh && cd $_  
ssh-keygen -t ed25519 -C "name@dc.tohoku.ac.jp" -f id_ed25519_ubuntu2github
```

Enter passphrase (empty for no passphrase):

と聞かれるのは無視して Enter で良い ▶ 秘密鍵が流出した場合に役に立つほか、リモートによっては設定を要求するものもある

### Commands

ssh	SSH でリモートホストに接続・ログイン・コマンド実行する
ssh-keygen	公開鍵認証方式のキーペア（公開鍵・秘密鍵）を生成する
scp	SSH を使用してリモートホストとファイルの授受を行う



# SSH の設定をしよう

## キーペア生成の確認

```
ls
```

```
config      id_ed25519_ubuntu2github  id_ed25519_ubuntu2github.pub
```

秘密鍵と公開鍵のキーペアが生成されていれば OK

## 公開鍵のコピー

```
cat ./id_ed25519_ubuntu2github.pub
```

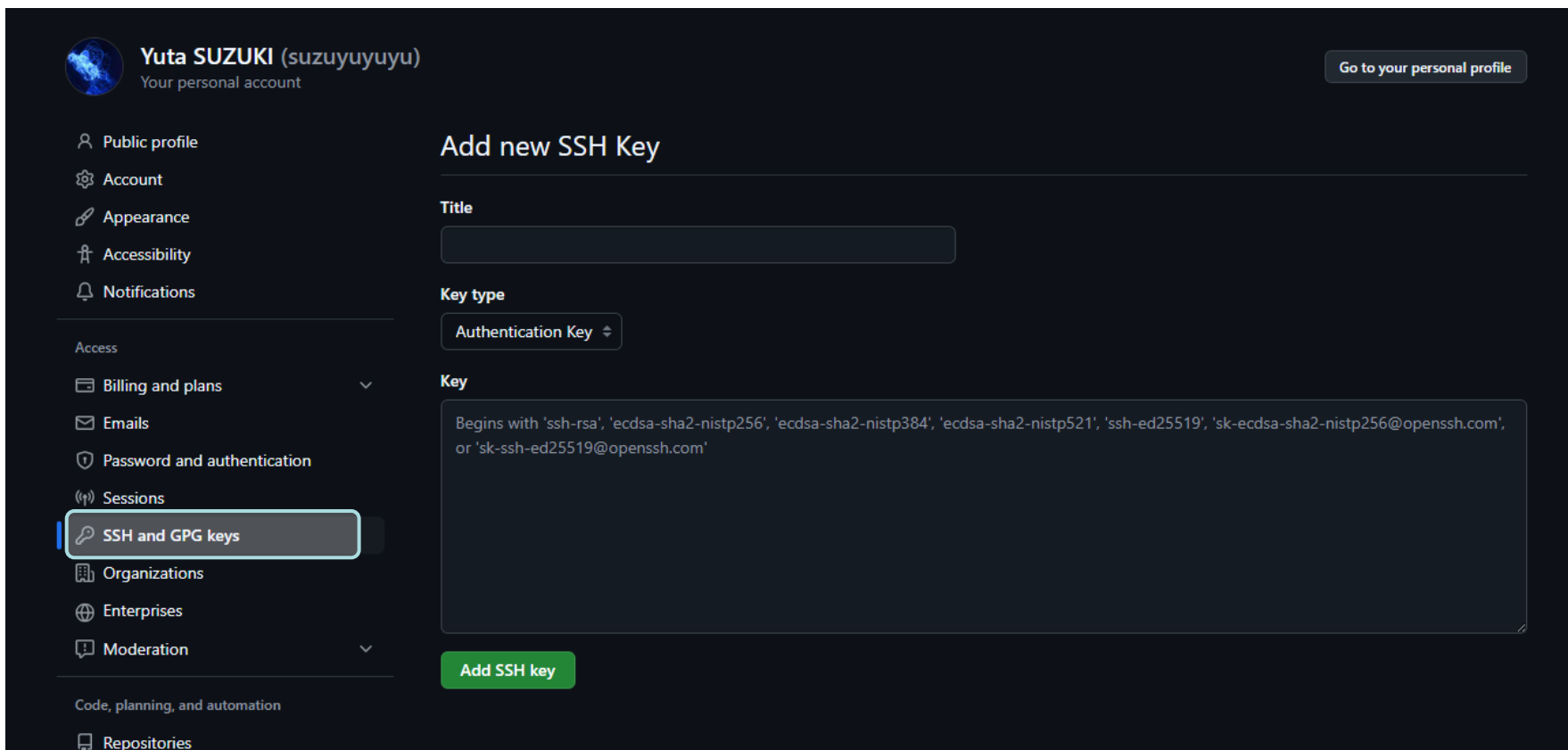
絶対に **.pub** を付けること。./id\_ed25519\_ubuntu2github は開かない  
その後クリップボードにコピーし、GitHub でペースト

# SSH の設定をしよう

## 公開鍵のコピー

その後クリップボードにコピーし、GitHub でペースト

 Settings → SSH and GPG keys → New SSH key → ペースト → Add SSH key



The screenshot shows the GitHub 'Add new SSH Key' page. On the left is a sidebar with the user's profile 'Yuta SUZUKI (suzuyuyuyu)' and a list of settings: Public profile, Account, Appearance, Accessibility, Notifications, Access, Billing and plans, Emails, Password and authentication, Sessions, SSH and GPG keys (highlighted with a red box), Organizations, Enterprises, and Moderation. The main content area is titled 'Add new SSH Key' and contains three fields: 'Title' (an empty text box), 'Key type' (a dropdown menu set to 'Authentication Key'), and 'Key' (a large text area containing a list of valid key prefixes: 'ssh-rsa', 'ecdsa-sha2-nistp256', 'ssh-ed25519', and 'sk-ecdsa-sha2-nistp256@openssh.com' or 'sk-ssh-ed25519@openssh.com'). At the bottom right of the main area is a green 'Add SSH key' button.

# SSH の設定をしよう

## config の作成（名前解決）

```
ls
```

```
Host github.com
  HostName github.com
  User git
  IdentityFile ~/.ssh/id_25519_github
```

名前解決

通常は毎回 User や HostName を入力するところを Host 名の定義によって省略する  
SSH 鍵へのパスを設定すれば公開鍵認証方式が簡単に可能になる

優先順位は 「コマンドライン > ~/.ssh/config > /etc/ssh/ssh\_config」

# SSH の設定をしよう

## 接続の確認

```
ssh -T git@github.com
```

```
Hi username! You've successfully authenticated, but...
```

こんなふうに表示されれば接続完了。

名前解決の観点からこのようにしても良い

```
ssh -T github.com
```

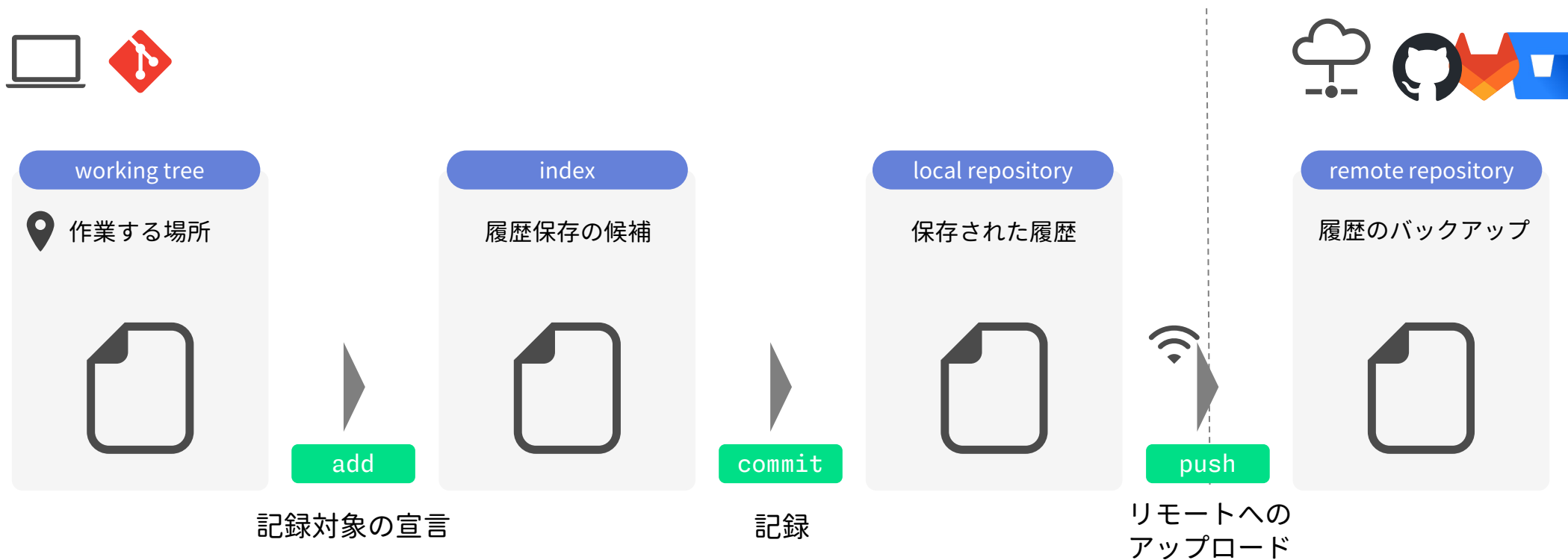
ついでに...

## Git の設定

```
git config --global user.name suzuki  
git config --global user.email "123456789+username@users.noreply.github.com"
```

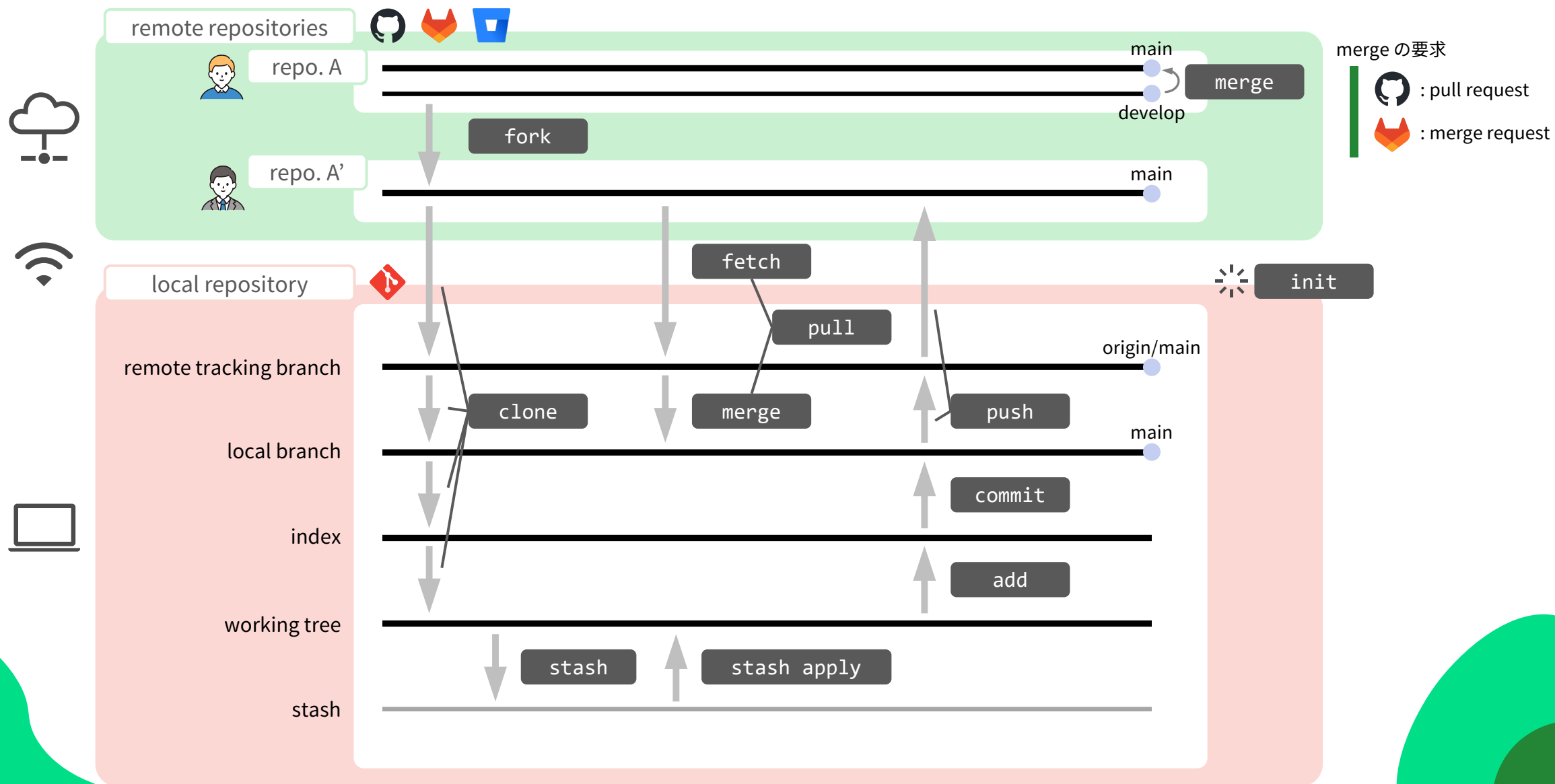
- GitHub はダミーのメールアドレスが割り当てられている
- Settings → Emails から確認可能
- GitHub に登録しているメールアドレスでもOK

# 簡単な流れ

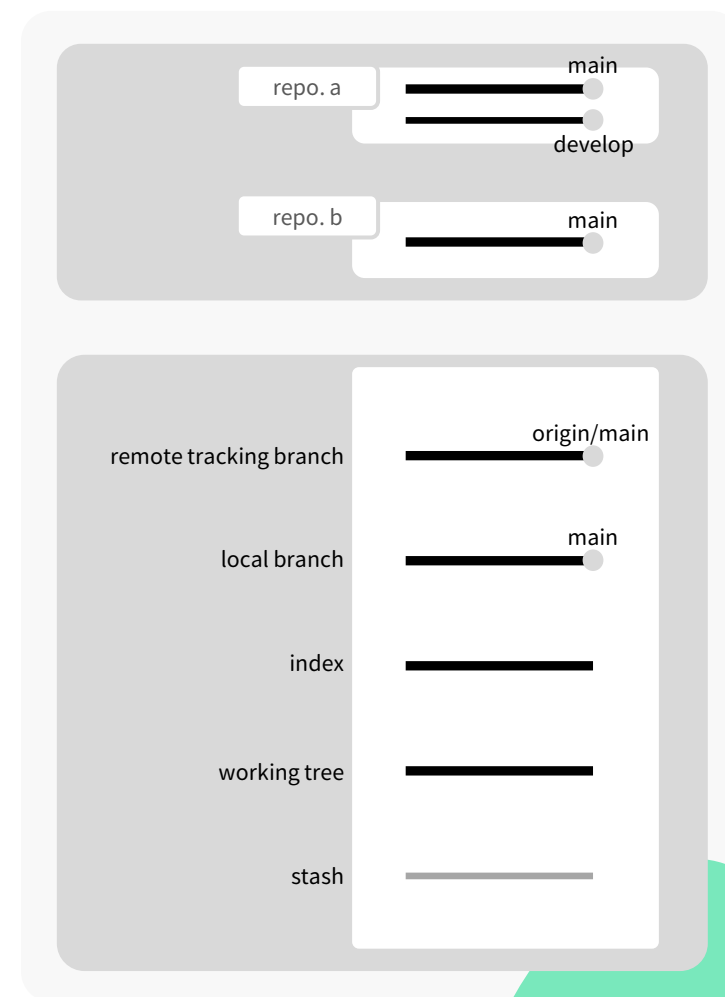
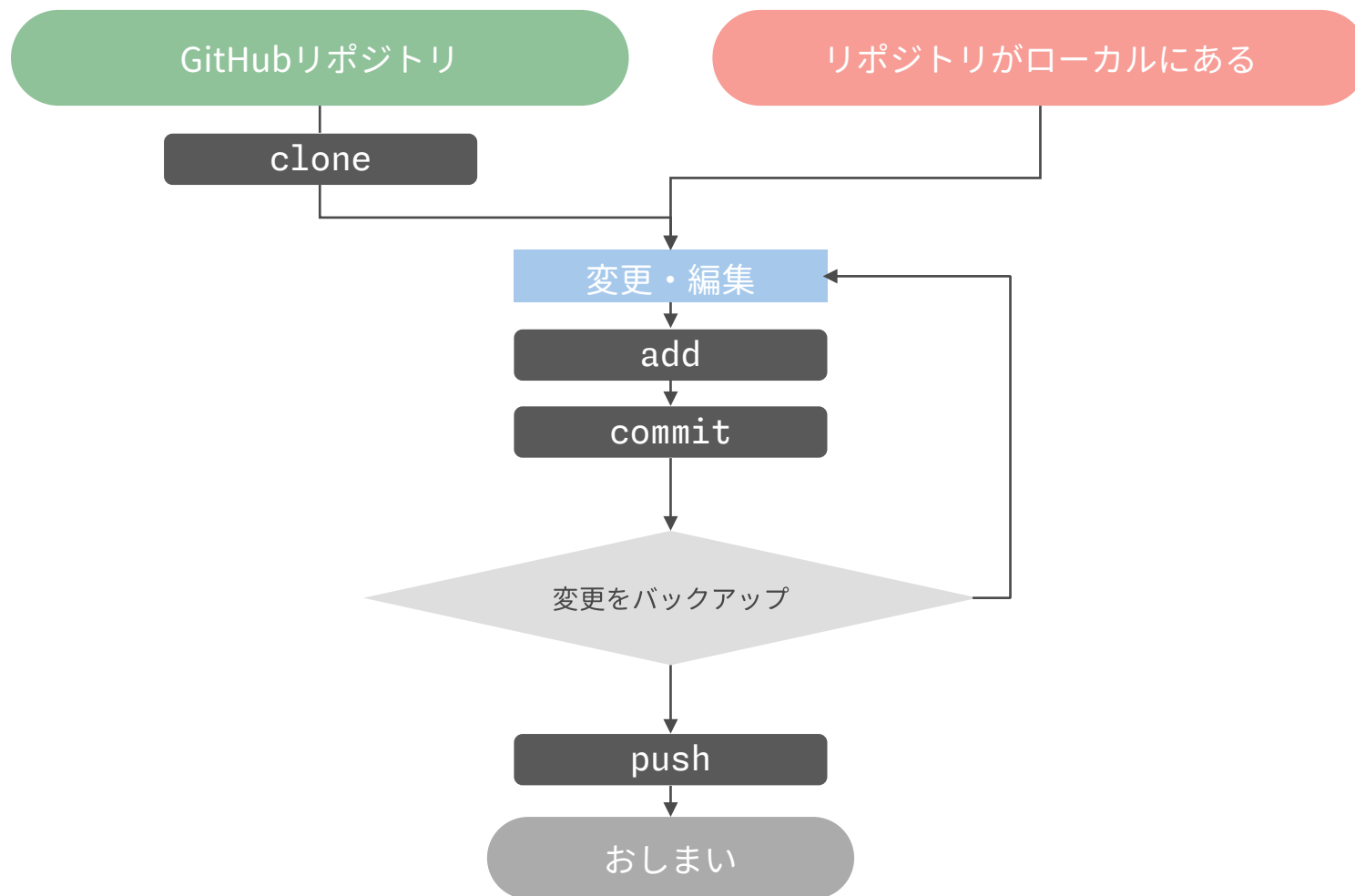


毎回 commit することで変更されたファイルのスナップショットを記録する  
Git の repository は **commit の集積** (ちょっと言い過ぎ)

# 全体の流れ



# 普段やることの復習（絶対に覚える）



# 絶対覚える

clone

add(/add -p)

commit(/commit -amend)

push

pull (fetch/merge)

status



# clone

```
git clone <repository link>
```

<repository link> のリポジトリをローカルに持ってくる

SSH なら `git@github.com:username/repository_name.git` のようになっているはず

## Options

```
--recursive
```

サブモジュールを含むリポジトリのデフォルトブランチを clone する

```
--depth <n>
```

最新から指定したコミット数だけ clone する

```
-b <branch name>
```

ブランチを指定して clone する

# add

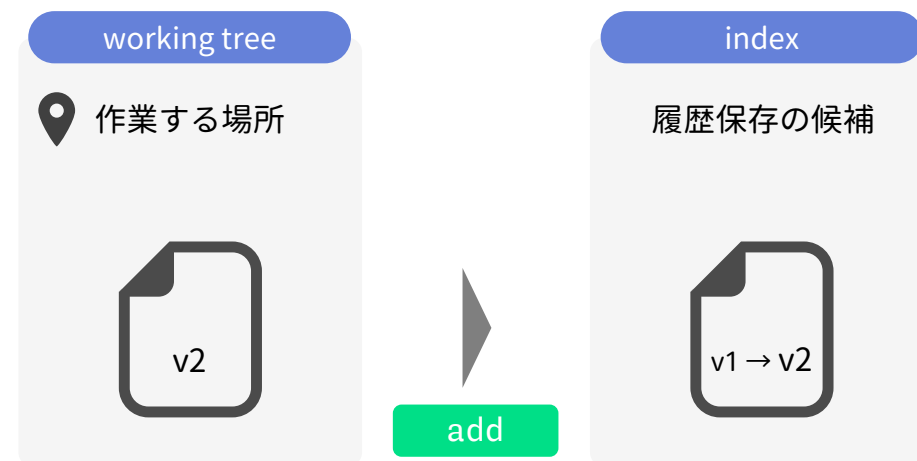
```
git add <file path>
```

<file path> を index に反映する

```
git add .
```

 でカレントディレクトリすべてを追加

```
git add ./*.py
```

 ですべての Python ファイルを追加

## Options

```
-A (or --all)
```

すべて (Untracked、Modified、Deleted) を add する

```
-u (or --update)
```

Modified、Deleted を add する

```
-p (or --patch)
```

部分的に選択しながら add する

# commit

```
git commit -m <commit message>
```

indexにあるファイルのスナップショットを記録する

-m <commit message> でどのような変更をしたかを記述しなければならない

-m オプションなしなら Vim などのエディターが開いて入力を要求される

commit message には “feat:”、“fix:”、“update:” などの prefix をつけることが推奨されていたりする

## Option

--amend

新しい commit を作成して、現在のブランチを置き換える  
要するに commit message を書き換えることができる

--verbose

-m オプションなし、--verbose オプションありのとき、細かい変更情報が表示される

# push

```
git push
```

ローカルの変更をリモートに反映する

```
git push origin main:main
```

```
git push <repository link> <local ref>:<remote ref>
```

が正式な書き方

<repository link> push 先を指定する。デフォルトは origin

<local ref> push する手元のブランチを指定する。デフォルトは HEAD (@と書く)

<remote ref> push 先を指定する。デフォルトは <local ref> と同じ名前のブランチ

## Options

`-u (or --set-upstream)` ローカルの branch とリモートの branch を関連づける (追跡関係を設定する)

`-f (or --force)` 強制的に push

`--force-with-lease` ローカルがリモートと比較して最新である場合のみ push (安全な force push)

## エイリアスの設定方法

git push --force-with-lease のようなコマンドは長くて打つのが面倒、かつたまにしか使わないので忘れがち

```
git config --global alias.fpush "push --force-with-lease"
```

で設定するか

~/.gitconfig に

```
[alias]
  fpush = push --force-with-lease
  wdiff = diff --ignore-all-space --word-diff
```

としておくと `git fpush` が `git push --force-with-lease` エイリアスとして登録される

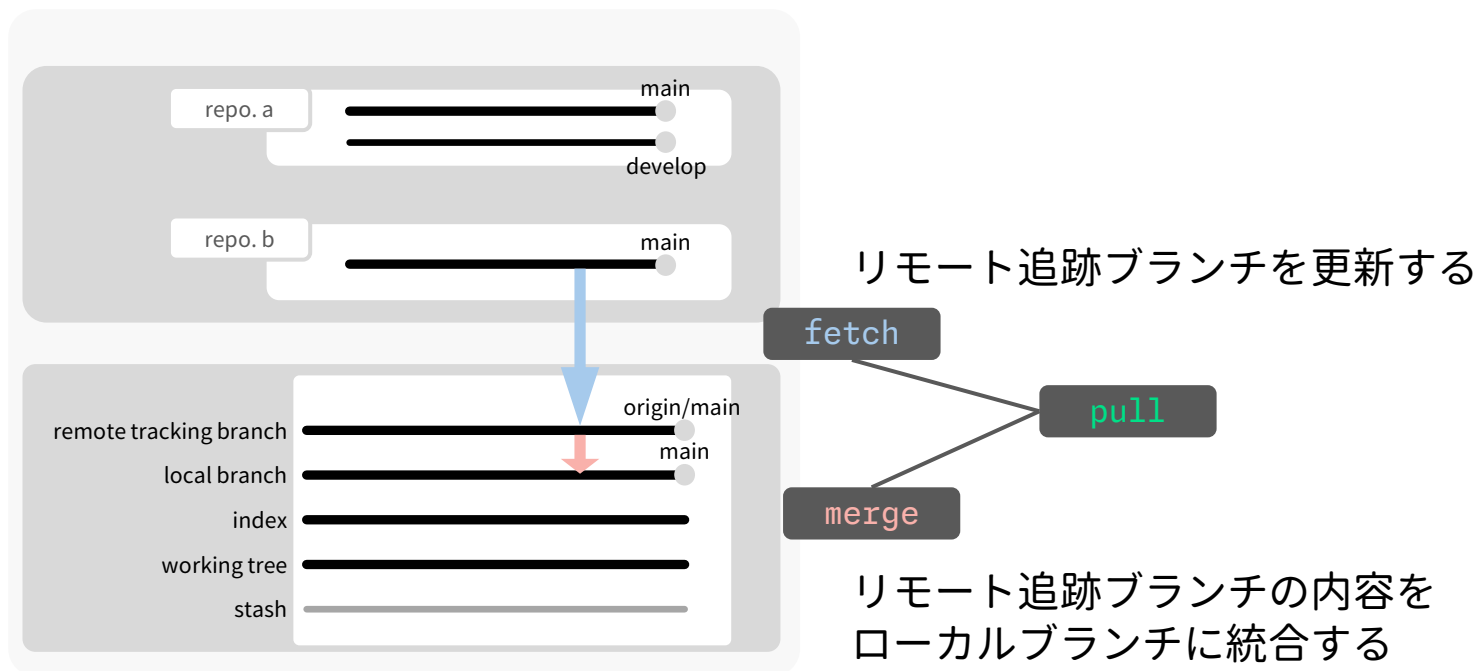
# pull

```
git pull
```

リモートの状態を手元に反映するコマンド

git fetch + git merge FETCH\_HEAD をやっている

詳しく理解するのは難しいので「手元を最新状態にしておくコマンド」として覚えておく



# status

```
git status
```

working tree/index/local branch の状態を表示するコマンド

branch の情報

例: 

```
On branch main
Your branch is up to date with 'origin/main'.
```

up to date	最新
behind	リモートの方が進んでいる
ahead	ローカルの方が進んでいる

ファイルの状態

エリア情報

Changes not staged for commit

working tree のみで変更されている

Changes to be committed

index に変更が反映され、commit を待機している

ファイル情報

Untracked 新しく作成され、add されていない

new file 新しく作成され、add された

modified 変更された

deleted 削除された

## ● 調べながら使えるようにしておく

init

reset

clean -f

stash (-/pop/apply/-u/save)

merge

cherry-pick



# init

```
git init
```

Git 管理をローカル側から開始するコマンド

.git ディレクトリが作成される

```
~/git-seminar
> ls -A
README.md  graph.py

~/git-seminar
> git init
Initialized empty Git repository in /home/yuta/git-seminar/.git/

~/git-seminar
> ls -A
.git README.md  graph.py
```

# reset

```
git reset <option> <commit ID> / git reset <commit ID> <file path>
```

- オプションあり

- ▶ HEAD の移動

<commit ID>

HEADからの相対位置やコミットIDで指定する

▶ [第II部 共同開発を管理しよう](#)

## Options

--hard

reset前のファイルが消える (<commit>のワーキングツリーを再現)

--mixed

デフォルトの値、ワーキングツリーにreset前のファイルが残る

--soft

インデックスにreset前のファイルが残る

- オプションなし

- ▶ ファイルの復元

<file path> を <commit ID> 時点の状態に変更

# clean

```
git clean <options>
```

untracked ファイルを削除するために使う

```
git reset --hard HEAD
```

 で消せないファイルを削除する

色々オプションがあるが、--force だけで十分

## Options

```
-f (or --force)
```

 デフォルトの設定ではこれがないとエラーになる

# stash

```
git stash <sub command> <option>
```

working tree での変更を一度退避する

pull などで conflict が発生するとき、手元の変更をいったん置いておきたいときに使う

## Sub commands

**save "comment"** stash に対して"comment"を残す

**list** stash のリストを表示

**apply** 退避を HEAD に戻す

**pop** 退避を HEAD に戻し、退避は削除する

**show** 退避の差分の要約を見る

**drop/clear** 最新・指定/すべての退避を削除する

## Options

**-u** Untracked なファイルも退避する

**-a** Ignore の対象であるファイルも退避する

# merge

```
git merge <branch> <option>
```

ある <branch> を HEAD（現在いる branch）に統合する

Conflictが発生しうるタイミングのひとつ（解消は [第II部 共同開発を管理しよう](#) で説明）

通常の commit と同じように commit（merge commit）が作成される

pull と同じくしっかり理解するには複雑なので、いったん無視

## Options/Args

`-m "message"` 作成される merge commit のメッセージを入力する

`--squash` ブランチの全変更を1つの commit にまとめる

`--ff` 可能な場合は fast-forward で merge する。--no-ff や --ff-only など制御する

# cherry-pick

```
git cherry-pick <commit ID>
```

特定の commit のみを部分的に取り込む

branch のすべての変更（すべての commit）を merge する必要がないときに使う

複数の commit を同時に適用することができる

## Options/Args

```
--continue
```

や

```
--abort
```

などがあるが、省略

とりあえず一緒にやってみよう

# ● やってほしいこと

SSH の設定

GitHub でのリポジトリの作成

comput-seminar-final-assignment

SSH を使った clone

ファイルの作成

add


commit


push



# 第 I 部の復習・まとめ

-  **git** でバージョン（履歴）管理、 **GitHub** ・  **GitLab** でバックアップ/共有することができる

  
自分の PC（ローカル）

 どっかのサーバー/インターネット（リモート）

- 履歴はすべて **.git ディレクトリ**に入っている
- リモートをメインに管理してローカルで変更、リモートに反映
- 必須コマンド

add : 履歴を記録する準備

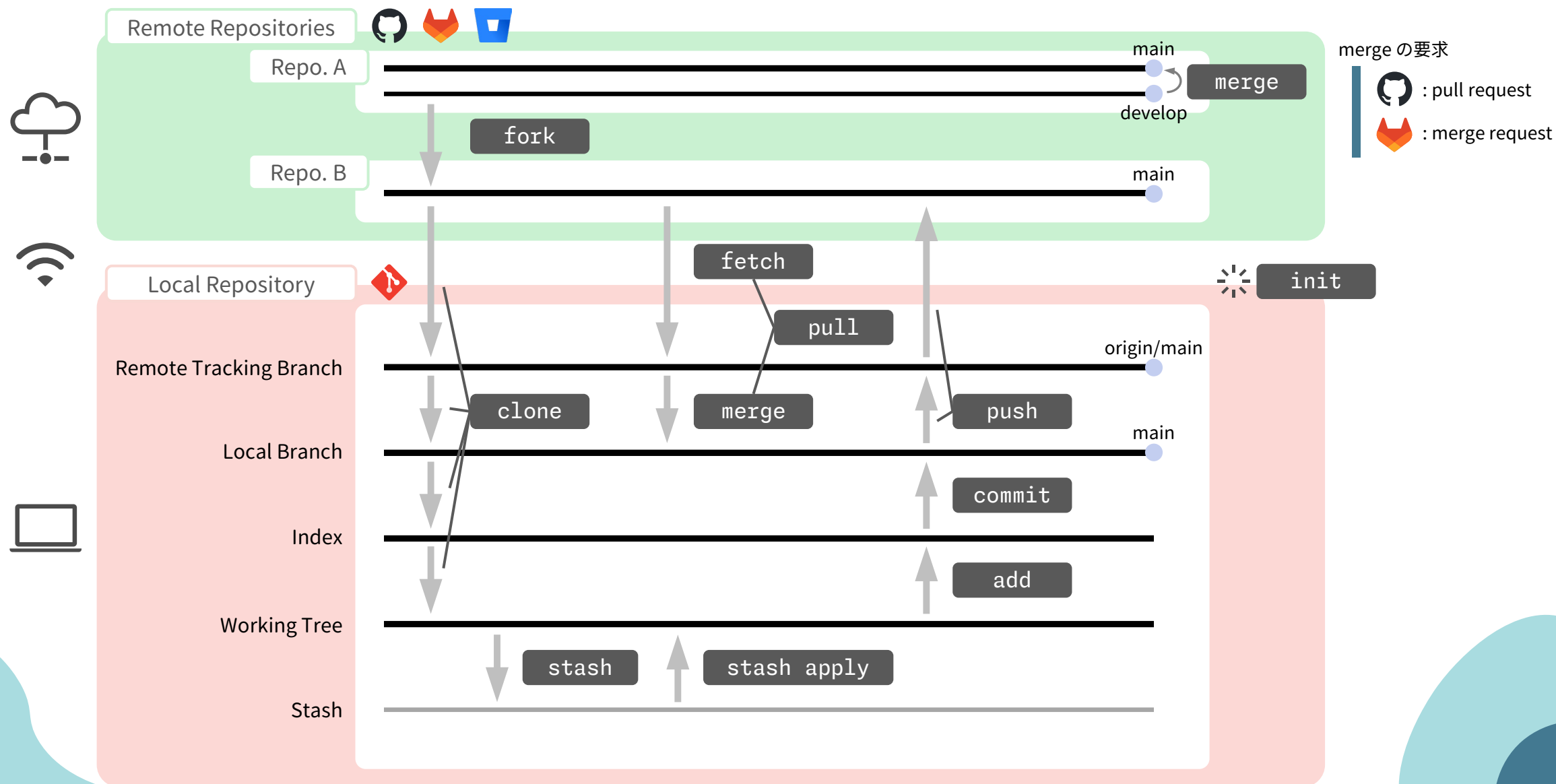
commit : 履歴を記録する

push : GitHub ・ GitLab に履歴を送信する

pull : GitHub ・ GitLab から履歴を持ってくる

- 明示的にコマンドを実行することでのみ、**履歴を保存しバックアップ**が取られる
  - ▶ 自動では保存されない

# 第 I 部の復習・まとめ



## 宿題（やってみてほしいこと）

- ここまでのコマンドを駆使して自分のコードを Git 管理するイメージを掴む

GitHub でリポジトリの作成

clone して .git ディレクトリを確認

ファイルを作成・編集してみる

add, commit, push して GitHub が更新されていることを確認

```
git status
```

で何が起きているかを確認しながら進める

- 何度かこれを繰り返して、バージョンの履歴が作成される様子を見る
- 過去のコミット時点に移動してみる
- ローカルリポジトリを削除して、もう一度 clone すれば問題なく復旧できる
- なにかしらプログラムを作成しても OK



## 次回

- commit とはなにか、人とどうやって共同で開発するかを話します

## 第Ⅰ部

# 個人開発を管理しよう

## 第Ⅱ部

# 共同開発を管理しよう

## TABLE OF CONTENTS

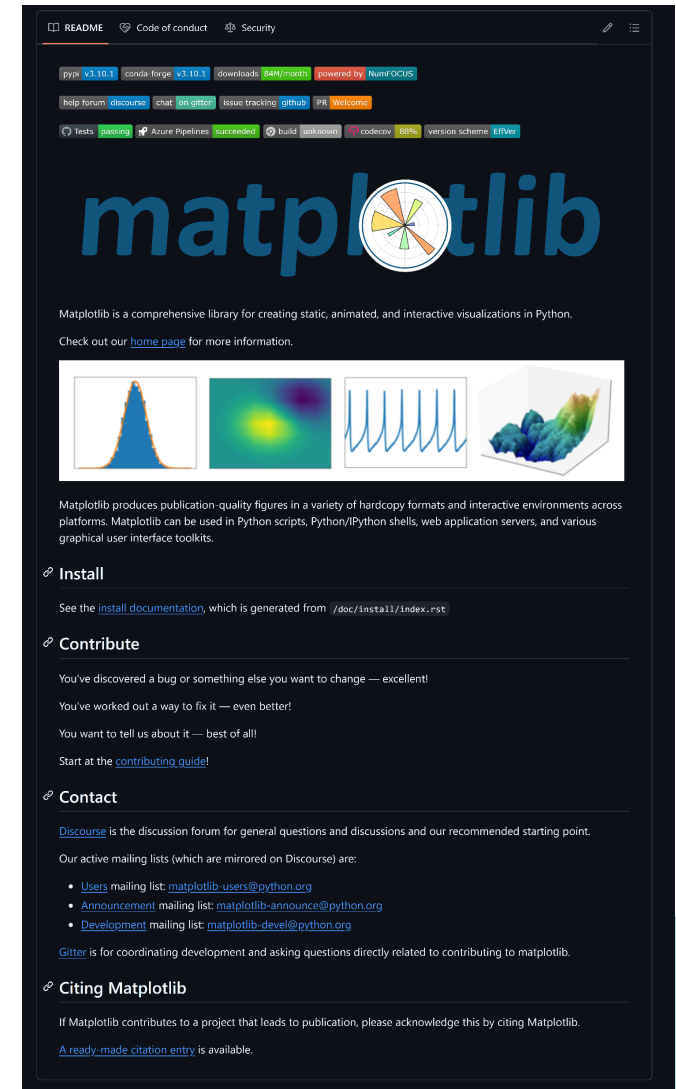
---

- 01. Issue を立てる
- 02. Git の難関 “branch” を理解する
- 03. Pull request を作成する
- 04. Conflict を解消する
- 05. Git コマンド 〈共同開発編〉

# まずは、“良い”リポジトリを目指そう

- リポジトリにコードだけを保存していても煩雑でわかりにくい
  - リポジトリに特別なファイルを設置して解消できる
  - Git では
    - README.md      リポジトリの説明
    - .gitignore      Git で管理対象外にするファイルのリスト
    - LICENCE      リポジトリの使用規則
- など、事前に決められたファイルを用いて  
特殊な役割をもたせることができる
- いろいろなリポジトリを覗いて書き方を学ぶとよい

## matplotlib の README.md



# ● 先ず Issue より始めよ

なによりも先に「いまからこれをやります」を説明する必要がある

Issue は GitHub (GitLab) で立てる

Issue を立てて

問題提起

問題点の議論

タスクの割り当て      を行う

Issue 作成者が1人で解決する必要はない。問題提起だけでもOK

Issue に対応した branch を作成する（切る）      ▶ 02. 作成した branch で作業する

Git flow, GitHub flow...

# 先ず Issue より始めよ (GitHub)

The screenshot shows a GitHub repository page for 'suzuyuyuyu / ext-download-manager'. The 'Issues' tab is active, showing an issue titled 'Load settings using popup' with ID #2. The issue is in the 'Open' state. The issue body contains a title 'add popup feature by using 'action' in manifest.json' and a list of bullet points in Japanese. A green box highlights the issue title and ID. Another green box highlights the issue body content. A third green box highlights the 'Add a comment' section at the bottom. The right sidebar shows the 'Assignees' section with 'suzuyuyuyu' assigned. The 'Labels' section shows 'No labels'. The 'Projects' section shows 'No projects'. The 'Milestone' section shows 'No milestone'. The 'Relationships' section shows 'None yet'. The 'Development' section shows 'Open in Workspace'. The 'Notifications' section shows 'Unsubscribe'. The bottom of the page has a 'Close issue' button and a 'Comment' button.

Issue タイトル Load settings using popup #2 Issue 番号

Open

Issue 担当者

Assignees

suzuyuyuyu

Labels

No labels

Projects

No projects

Milestone

No milestone

Relationships

None yet

Development

Open in Workspace

Create a branch for this issue or link a pull request.

Notifications

Customize

Unsubscribe

You're receiving notifications because you're subscribed to this thread.

add popup feature by using 'action' in manifest.json

- popup.htmlによって拡張機能アイコンクリック時に設定入力のポップアップを表示する
- プレースホルダーを用いてファイル名を設定できるようにする
- 設定はchrome.storage.syncによって保存する

Create sub-issue

Markdown で書く

コメントで議論できる

Add a comment

Write Preview

Use Markdown to format your comment

Paste, drop, or click to add files

Close issue

Comment



## Issue



## Markdown で書く

# Issue を立ててみよう

練習用のリポジトリを使って共同開発（プロジェクト）の練習をしよう

## プロジェクトの概要

- README にプロフィールを追加していく
- 変更するときには Issue を立てる
- 今回は自分のプロフィールを branch を使って追加する

まずは、自分のプロフィールを追加する Issue を作成しよう

# Markdown (.md) を書こう

文章を書くときに記号を用いて、見出しや箇条書きなどを表現する記述方法



Markdown は

GitHub、GitLab の README

Qiita などの技術系の情報共有サイト

Notion、Slack

などに用いられる記法

ソフトなどのインストールをせずに書くことができる

Notion、VSCode、Obsidian などのソフトや GitHub を使うと綺麗に表示できる

README.md

# Git の仕組み・構造



repository

- feat: 新しいアルゴリズム
- feat: ちょっと修正
- init: ソースコードを作成

- .git ディレクトリがリポジトリのすべてを管理している
- .git ディレクトリを消すと履歴が消え、Git 管理から外れる



.git ディレクトリ



...



⋮

- commit の履歴
  - commit 時点のスナップショット
- 現在地 (HEAD) の情報
- branch の情報
- remote repository (origin) の情報
- ⋮

ここになんかすごい  
データがたくさん集  
まっている

# Git は意外と原始的

Git がやっていることは

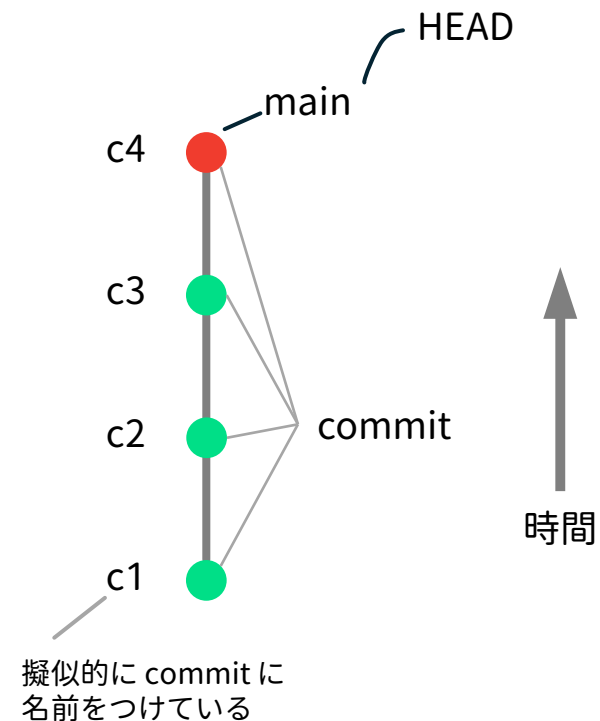
- 変更があったファイルを圧縮して記録・保存する
- 記録 (commit) に番号 (ハッシュ値) をつける
- 記録の繋がり (依存関係) をハッシュ値を用いて記録する

自分が派生した元の commit を親コミット (parent) という

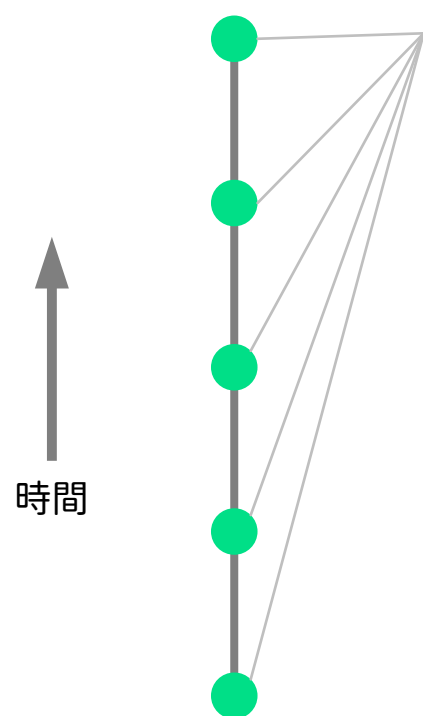
▶ e.g., c3 の親コミットは c2

- インターネットを使ってバックアップする

すべて .git ディレクトリに保存されている



# Commit/HEAD とは何か？



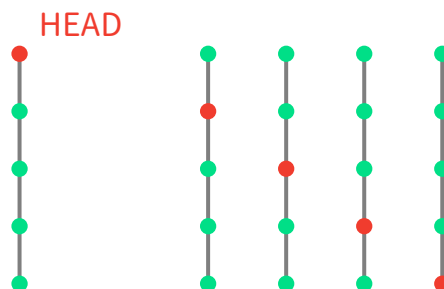
## commit

ある時点に記憶させた履歴、セーブポイント  
どこから派生したか、親となる commit を覚えている

## HEAD

いま、編集しようとしている commit。「現在地」と説明される  
commit のいずれかを指す ▶ commit のポインター  
どの commit にも移動 (checkout/switch) できる

▶ 第 II 部 共同開発を管理しよう

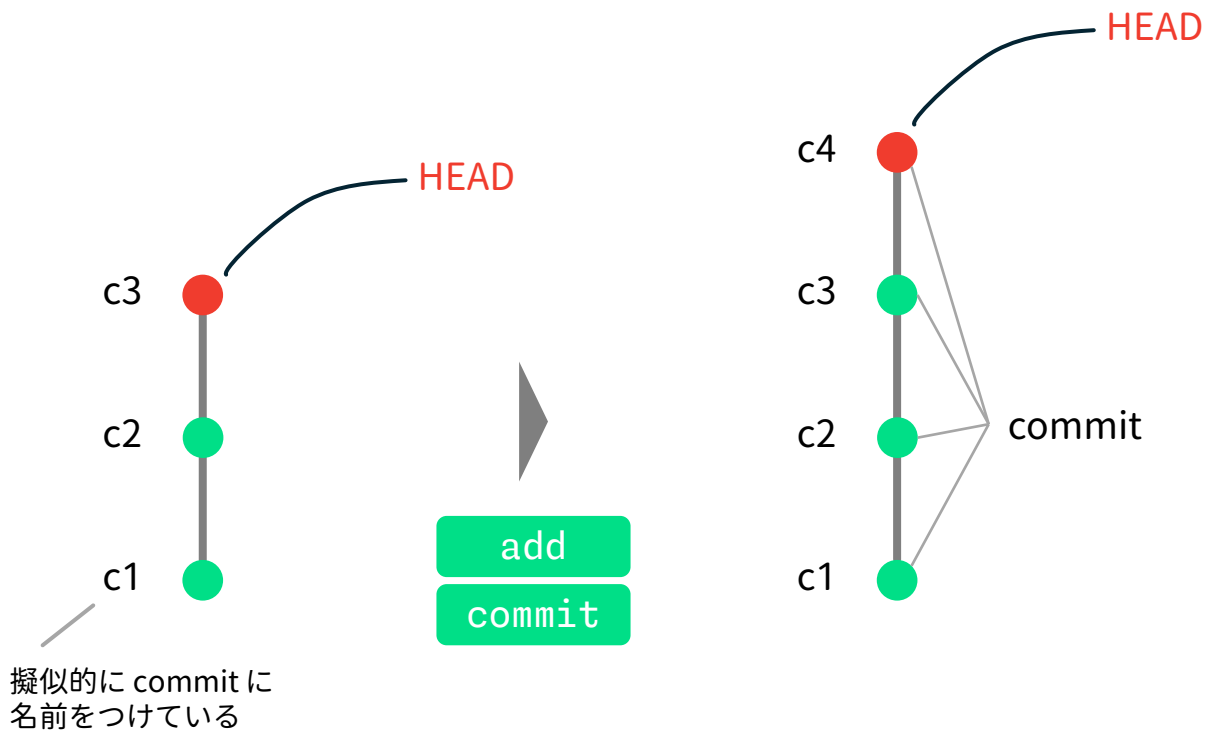


いずれもありうる

ただし、1つ以外は避けるべき状況

▶ 「detached HEAD」 (あとですこし触れる)

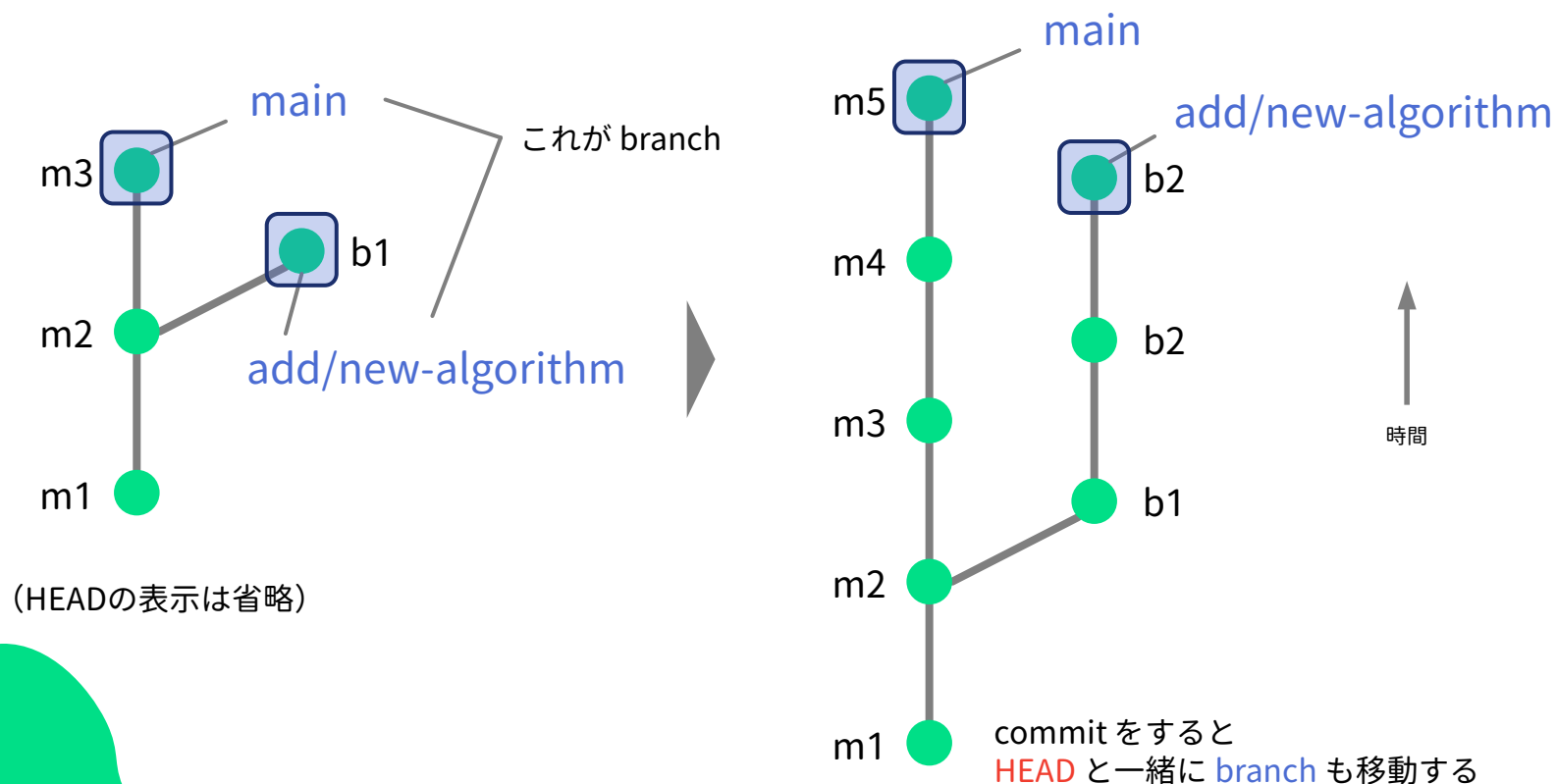
# commit で何が起こる？



commit の作成  
HEAD（現在地）の移動  
が実行される

# Branch とは何か？

作業環境を分岐することができる機能のこと  
実際は commit に目印・名前をつけているだけ



(HEADの表示は省略)

branch を作成して移動

```
git branch <branch name>
```

```
git checkout <branch name>
```

開発方針によって branch の切り方が異なる

GitHub Flow/Git-flow/GitLab Flow...

prefix と内容を “/” で区切る命名が多い

例:

feature/improve-ui

fix/typo-in-output

hotfix/#1-release-version

▶ Issue 番号を書く



# Branch とは何か？

他のブランチに内容を統合（merge）できる

右図は add/new-algorithm を main に取り込んだ例

merge すると branch を取り込んだ commit が作成される

▶ merge commit は親コミットが2つある

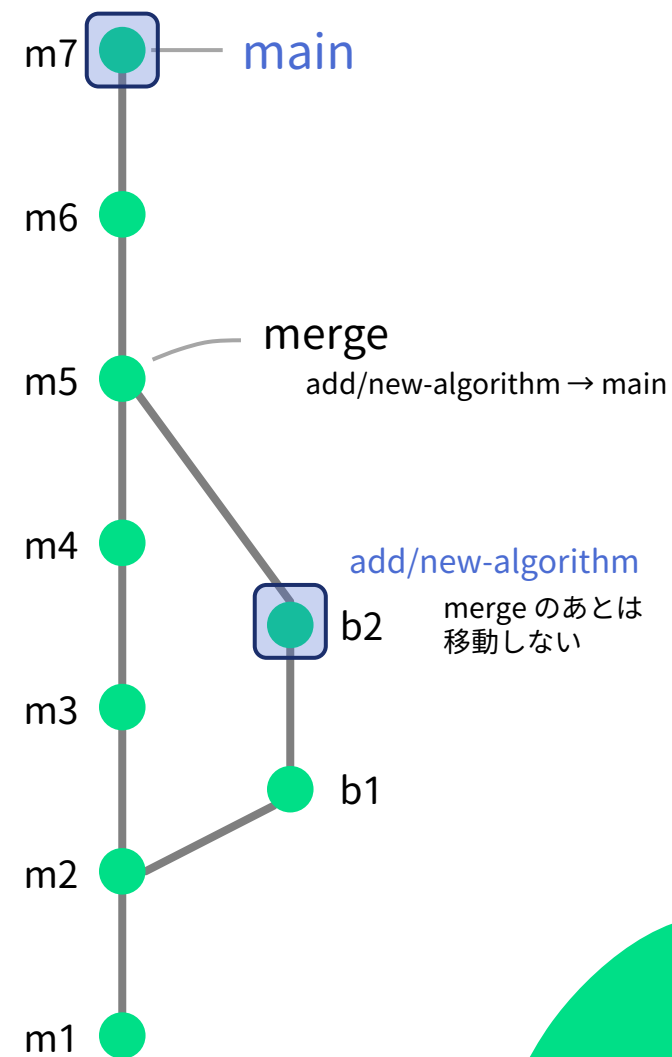
## merge の手順

取り込む側の branch に移動

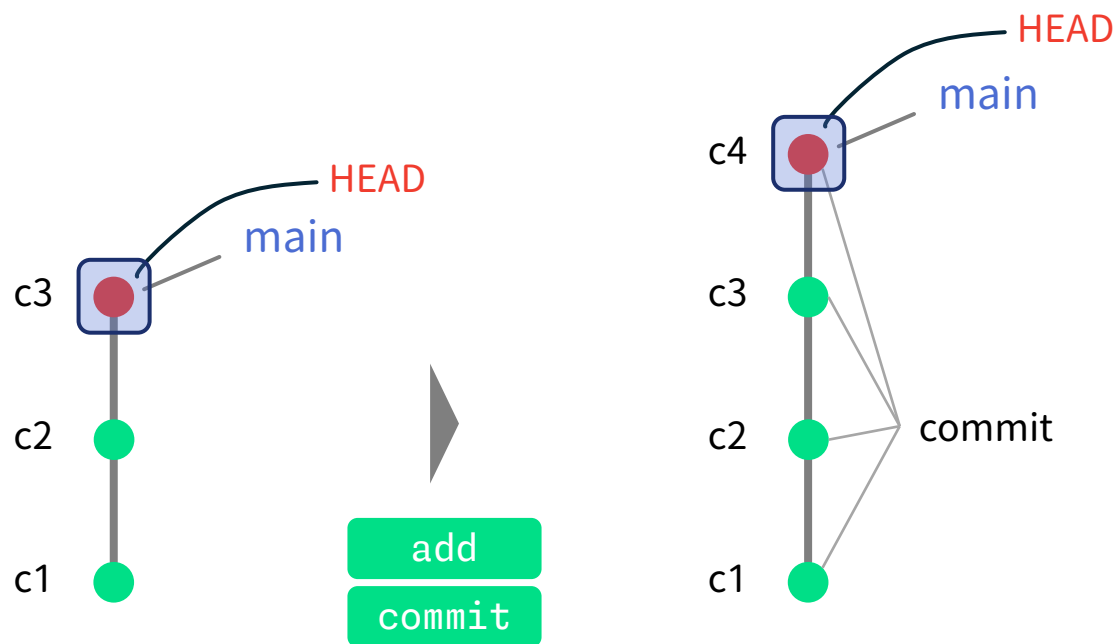
```
git checkout main
```

add/new-algorithm ブランチを merge する

```
git merge add/new-algorithm
```



## branch 上での commit で何が起こる？



commit の作成

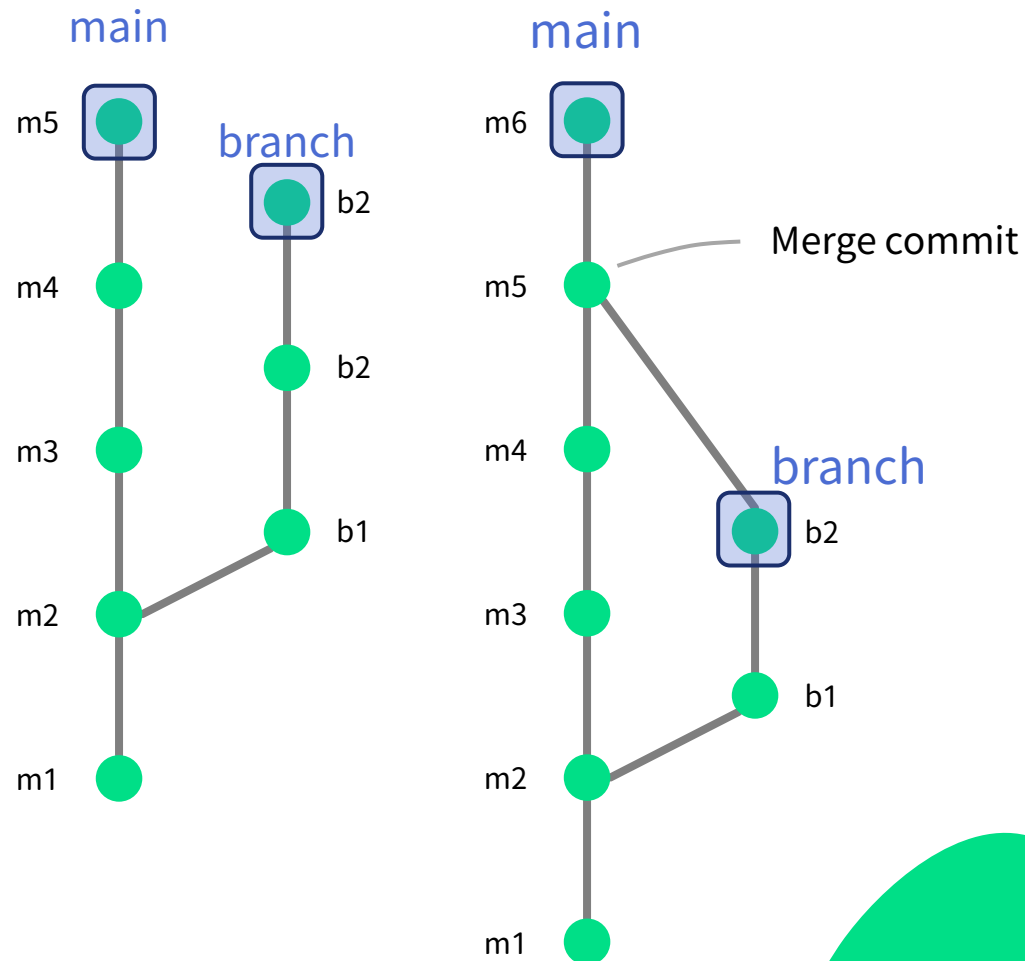
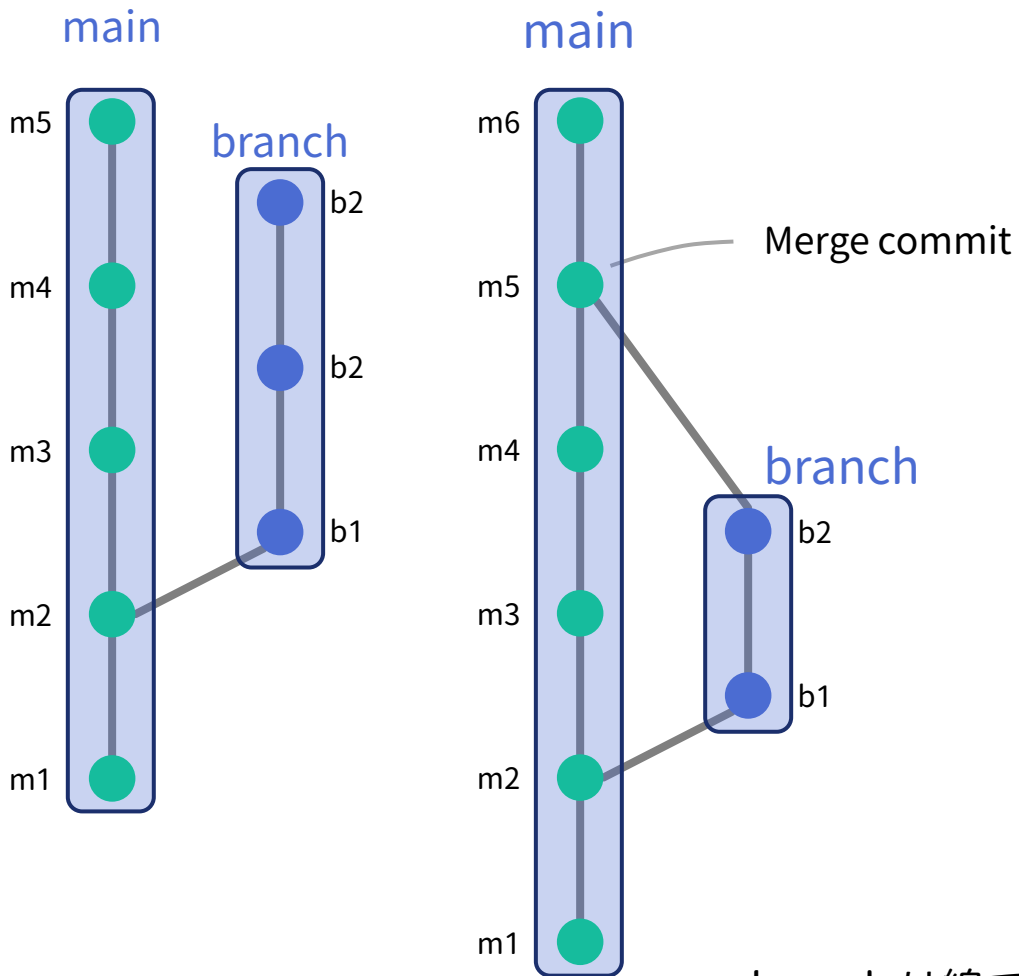
HEAD（現在地）の移動

HEAD が branch を指していた場合は

branch の移動

が実行される

# Branch のよくある誤解



branch は線ではない  
あくまでも 1 つの commit を指す

# ^ と ~ による commit の相対位置

# commit ID の入力がめんどくさい

commit は ID だけでなく **相対位置** によって表現することもできる

● を基準として

- 1つ前の commit (親 commit) を指す

● ~1    ● ~    ● ^    ● ^1

The first parent of ●

```
git checkout HEAD~1
```

右図なら m5 に checkout する

- 2つ前の commit を指す

● ~2    ● ~~    ● ^^    ~~● ^2~~

The first parent of    The first parent of ●

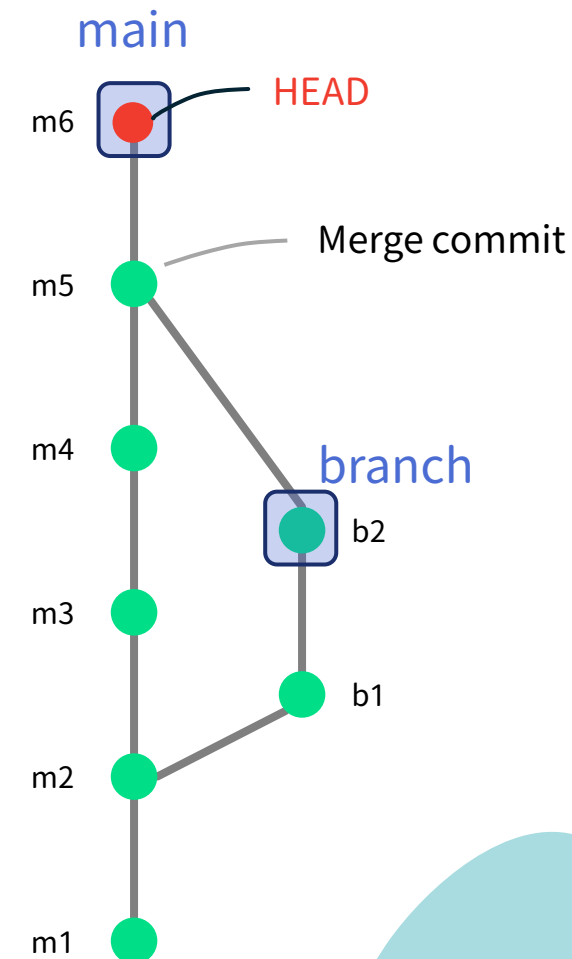
```
git checkout HEAD~2
```

右図なら m4 に checkout する

~    チルダ (tilde)

^    キャレット (caret)

● ^^ = (● ^)^



# commit ID の入力がめんどくさい

commit は ID だけでなく 相対位置 によって表現することもできる

● を基準として

- 1つ前の 2つめの親 commit を指す

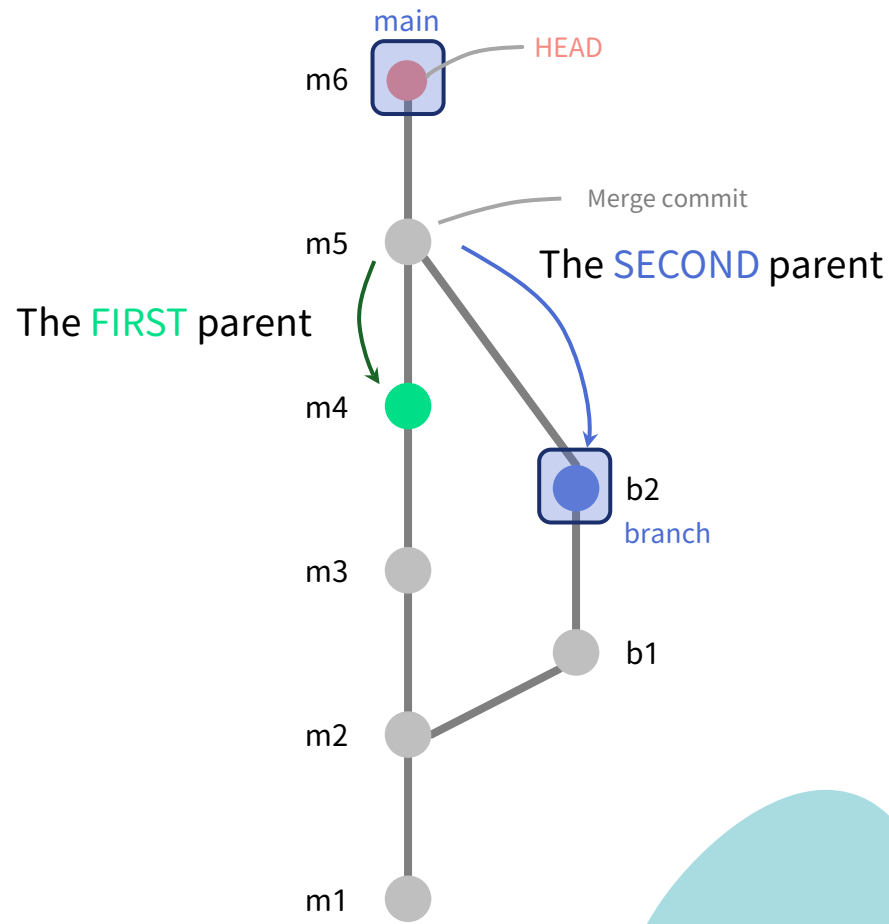
● ^2

The SECOND parent of ●

Merge commit のように親 commit が 2 つある場合に特定する

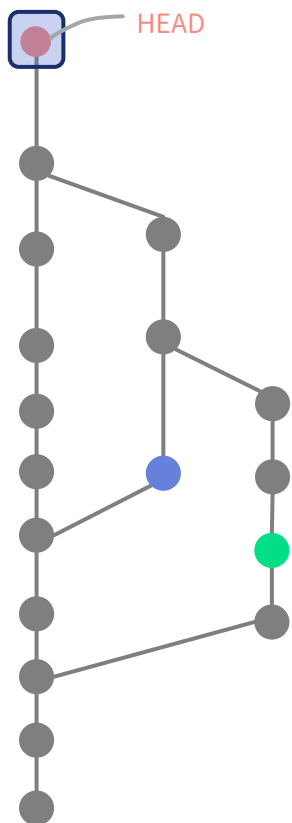
```
git checkout m5^2
```

右図なら b2 に checkout する



# Quiz

HEAD からの相対位置を使って ● と ● はどう表すでしょうか？



HEAD~^2~2

HEAD^^2^^

などでも OK



HEAD~^2~^2~2

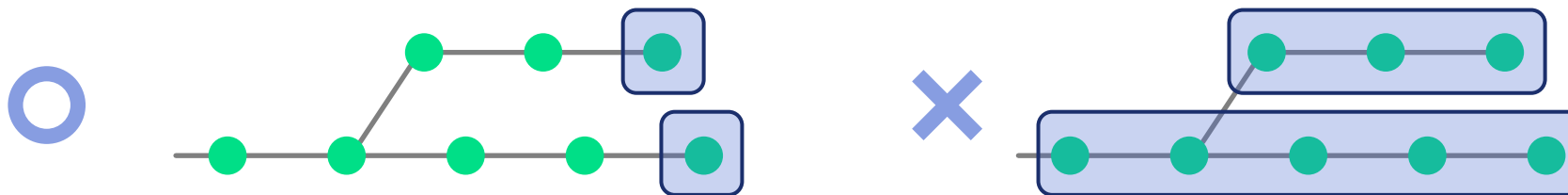
## 結論

### commit ▶ ある瞬間のスナップショット

差分を記録しているわけではなく、変更があったファイルを圧縮して保存している  
自動で発行される commit ID（ハッシュ値）によって特定することができる

### branch ▶ commit に対する目印（commit のポインター）

main、feature/add-alg、hotfix/error-handling のような名前をつけて移動しやすくする



### HEAD ▶ いまユーザーが作業している commit（commit のポインター）

checkout や switch で移動する

branch でないところに HEAD がある状態を detached HEAD という

tag ▶ commit に対する目印。移動しない。（commit のポインター）



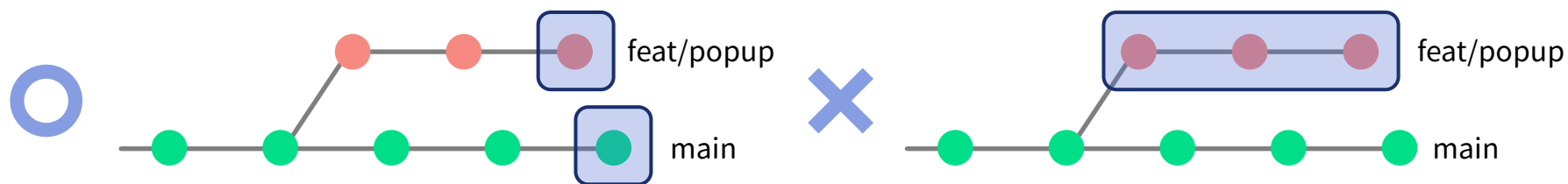
# branch を作成しよう

共同開発では main ブランチにバグを作らないために branch で分岐して作業する

## 〈復習〉

branch ▶ 目印のついた commit (commit のポインター)

main、feature/add-alg、hotfix/error-handling のような名前をつけて移動しやすくする



明示的に統合 (merge) しない限り他のブランチには影響しない

## まずはリポジトリを clone

```
git clone git@github.com:username/repository-name.git
```

リポジトリをまるっとローカルに複製する

.git ディレクトリを認識して、ローカルリポジトリになる

```
~  
> git clone git@github.com:suzuyuyuyu/git-seminar.git  
Cloning into 'git-seminar'...  
remote: Enumerating objects: 41, done.  
remote: Counting objects: 100% (41/41), done.  
remote: Compressing objects: 100% (26/26), done.  
remote: Total 41 (delta 12), reused 35 (delta 6), pack-reused 0 (from 0)  
Receiving objects: 100% (41/41), 9.06 KiB | 1.81 MiB/s, done.  
Resolving deltas: 100% (12/12), done.
```

```
~  
> ls  
git-seminar
```

# branch (feature/name)を作成しよう

```
git branch feature/name
```

branch (feature/name) を作成する

branch 名は作業内容がわかるように簡潔な名前をつける

一般には prefix/issue\_name のようにする

prefix は feature/、fix/ などがつけられる

issue\_name には対応する Issue の番号をつけることもある

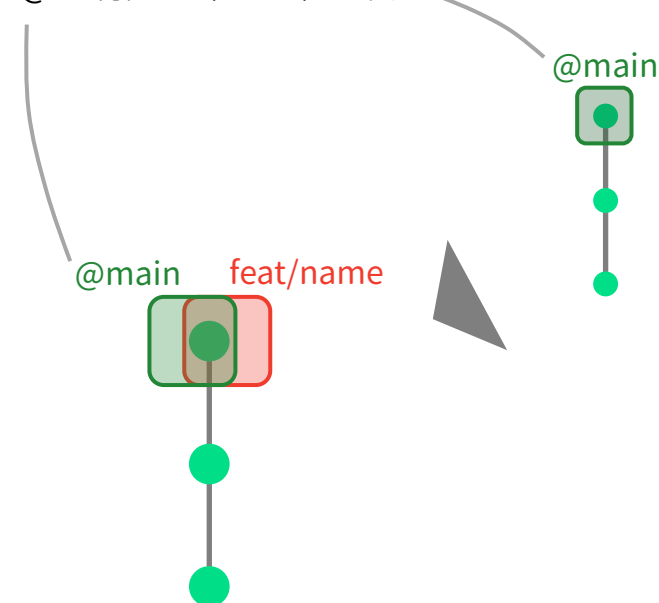
branch 名は “/” で区切られることが多い

▶ (.git の内部でディレクトリ名として解釈されるため)

```
~/repository
> git branch
branch/a
branch/b
* main
```

```
~/repository
> tree -a .git/refs/heads
.git/refs/heads
├── branch
│   ├── a
│   └── b
└── main
```

@ は現在地 (HEAD) を表す



新しい branch “feat/name” が最新のコミットを指した

“\*” がついているのが現在地 (HEAD)

## feature/name に入ろう

```
git checkout feature/name
```

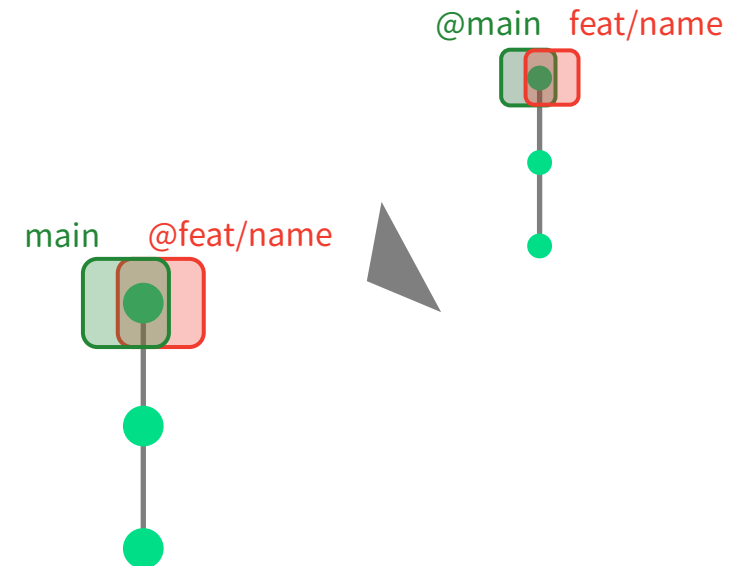
/

```
git switch feature/name
```

指定した branch に移動する

HEAD が移動することを確認しよう

```
~/repository  
> git status  
On branch feature/name  
nothing to commit, working tree clean
```

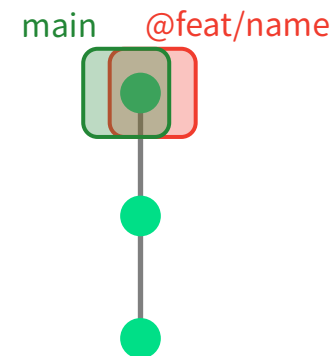


# ファイルを編集しよう

ファイルを編集して、Git が変更を認識するか確認する

```
~/repository
> git status
On branch feature/name
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working
directory)
        modified:   README.md

no changes added to commit (use "git add" and/or "git commit -a")
```



編集しただけ。変わらない  
Working Tree だけ変わっている

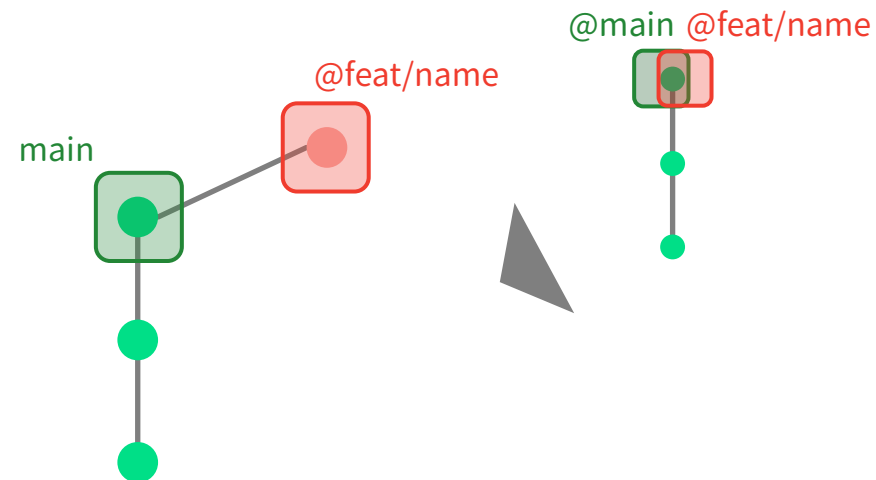
# commit しよう

```
~/repository
> git add README.md

~/repository
> git status
On branch feature/name
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
       modified:   README.md
```

```
~/repository
> git commit -m 'docs: add my profile'
[branch/c df7d54e] docs: add my profile
 1 file changed, 1 insertion(+)
```

```
~/repository
> git status
On branch feature/name
nothing to commit, working tree clean
```



commit が作られて feat/name ブランチが移動した  
HEAD がブランチを指していたため、ブランチの移動に追従した

# pushしよう

```
~/repository
> git branch -vv
* branch/c df7d54e docs: add my profile
main      860d7fc [origin/main] betsuno commit message
```

つまり origin/feature/name

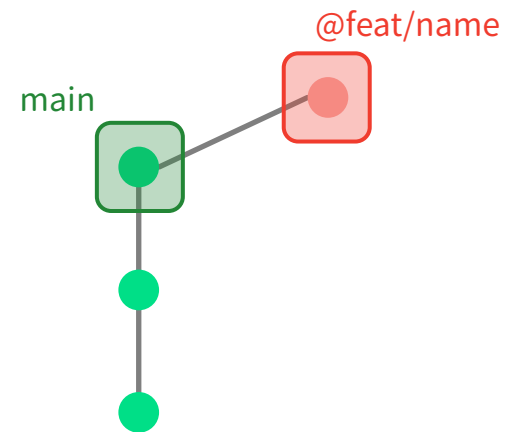
リモートの feature/name ブランチがまだないので

```
git push -u origin feature/name
```

最初は明示的に push する

これであとは統合 (merge) するだけの状態になった

自分のブランチなら何回 commit しても OK



push ではローカルのブランチは変わらない

# 自分の変更を反映するために

- main（デフォルトブランチ）に変更を反映するには作業が必要
  - merge: feat/name → main

- この merge を求める作業を



Pull Request



Merge Request

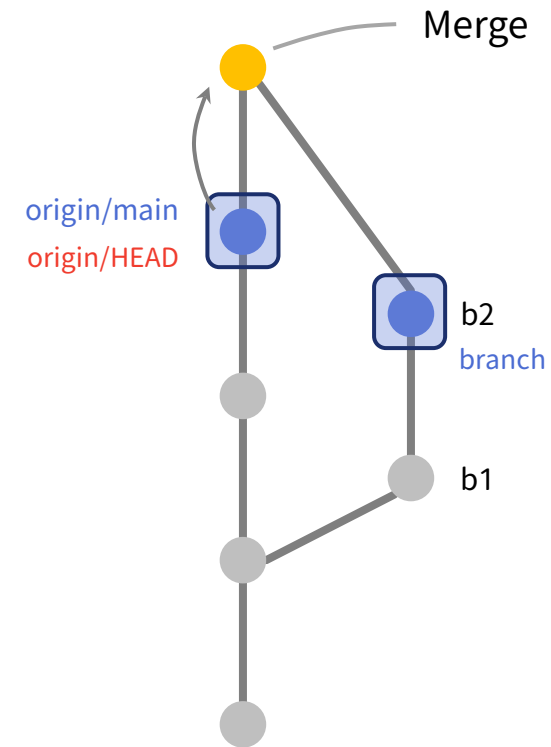
という

- うまく merge できないこともある

- コンフリクト・競合

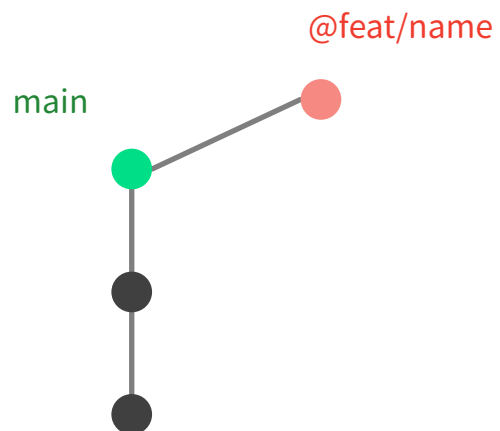
origin/HEAD

リモートのデフォルトブランチ  
GitHubを開いたときに最初に見える  
ブランチのこと

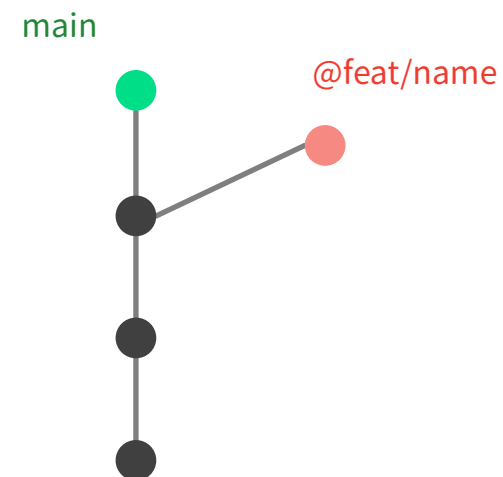




# Conflict (競合) とは？

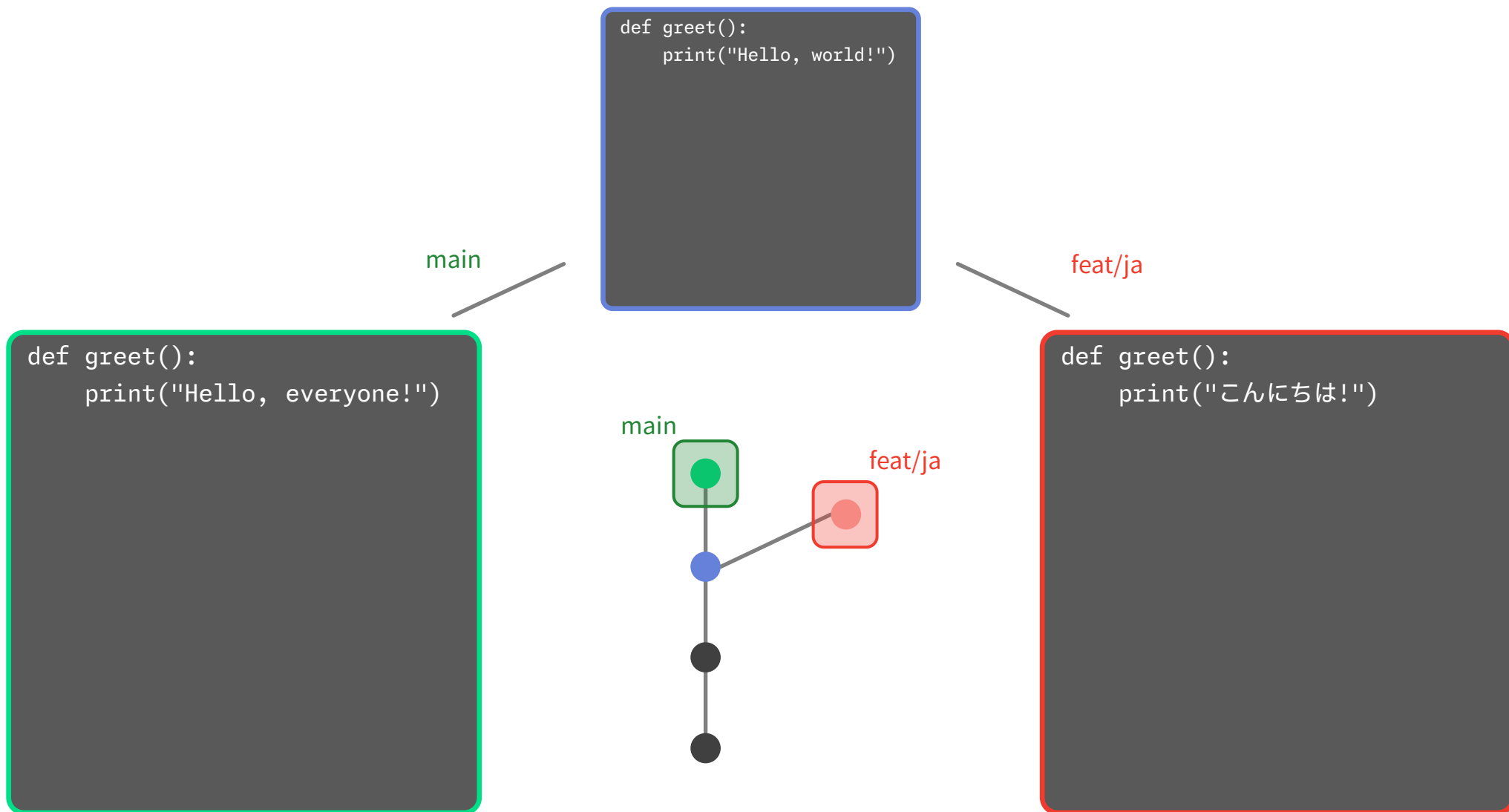


最終的にこんなブランチだったが



その間に main も更新されていたとする

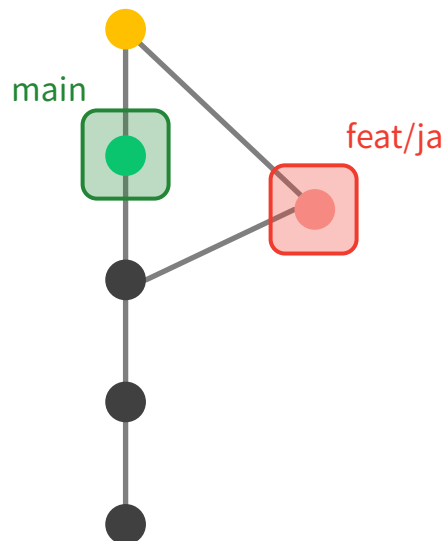
# Conflict (競合) とは？



# Conflict (競合) とは？

```
def greet():  
    print("Hello, everyone!")
```

merge したいとき...



```
def greet():  
    print("こんにちは!")
```

`git merge feat/ja`

どちらを採用すべきか  
容易に判断できない

conflict (競合)

# Conflict（競合）とは？

どちらを採用すべきか  
容易に判断できない

conflict

```
~  
(main) > git merge feat/ja  
Auto-merging hello.py  
CONFLICT (content): Merge conflict in hello.py  
Automatic merge failed; fix conflicts and then commit the result.
```

```
~  
(main) > cat hello.py  
def greet():  
<<<<<<< HEAD  
    print("Hello, everyone!")  
=====  
    print("こんにちは!")  
>>>>>> feat/ja
```

2つの変更が  
並んで表示される

好きに編集するだけ

main のこと

# ● Conflict（競合）を解消しよう

解消の方法は2つある

- ▶ GitHub 上で解消する ← ここではやりません
- ▶ ローカルで解消する

# ローカルで conflict（競合）を解消しよう

どう直してもよい。というか、それぞれの変更をどう統合するかは人間にしかできない

```
def greet():  
<<<<<<< HEAD  
    print("Hello, everyone!")  
=====  
    print("こんにちは!")  
>>>>>> feat/ja
```

いままでのはどうでもいい  
そのまま取り込みたい

```
def greet():  
    print("こんにちは!")
```

どっちも...

```
def greet():  
    print("Hello, everyone!")  
    print("こんにちは!")
```

いいところりをしたい  
新しくしたい

```
def greet():  
    print("みなさんこんにちは!")
```

# ローカルで conflict（競合）を解消しよう

Merge したという記録を残す → Merge commit

```
~  
> git status  
On branch main  
You have unmerged paths.  
  (fix conflicts and run "git commit")  
  (use "git merge --abort" to abort the merge)  
  
Unmerged paths:  
  (use "git add <file>..." to mark resolution)  
      both modified:   hello.py  
  
no changes added to commit (use "git add" and/or ...)
```

```
~  
> git add hello.py
```

```
~  
> git commit
```

—— オプションなしで実行してみる

hello.py

```
def greet():  
    print("みなさんこんにちは!")
```

# ローカルで conflict（競合）を解消しよう

Merge したという記録を残す → Merge commit

```
~  
> git commit
```

Vim などの  
エディタが開く

```
Merge branch 'feat/ja'
```

デフォルトで入力される  
merge commit message

```
# Conflicts:  
#     hello.py  
#  
# It looks like you may be committing a merge.  
# If this is not correct, please run  
#     git update-ref -d MERGE_HEAD  
# and try again.  
  
# Please enter the commit message for your changes. Lines starting  
# with '#' will be ignored, and an empty message aborts the commit.  
#  
# On branch main  
# All conflicts fixed but you are still merging.  
#  
# Changes to be committed:  
#     modified:   hello.py
```

hello.py

```
def greet():  
    print("みなさんこんにちは!")
```



# ローカルで conflict（競合）を解消しよう

Merge のときに編集した通りになっている

```
~  
> cat hello.py  
def greet():  
    print("みなさんこんにちは!")
```

hello.py

```
def greet():  
    print("みなさんこんにちは!")
```

最新の commit (main) も merge commit になっている

```
~  
> git graph  
* 1a4d4a4 [2025-06-12] (HEAD -> refs/heads/main) Merge branch 'feat/ja' to support Japanese @suzuyuyuyu  
|\n| * 0578fd1 [2025-06-12] (refs/heads/feat/ja) translate into Japanese @suzuyuyuyu  
* | bdc6f6a [2025-06-12] change greeting word @suzuyuyuyu  
|/  
* ce09afd [2025-06-12] initial commit @suzuyuyuyu
```

alias

```
graph = "!f() { git log --graph --decorate=full -20 --date=short --format='%C(yellow)%hC(reset) %C(magenta)[%ad]C(reset)%C(auto)%dC(reset) %s %C(cyan)@%an%C(reset)' ¥"$@¥"; }; f"
```

## ● ブランチ周辺のコマンドまとめ

branch

checkout

rebase(/merge/-autostash)

restore

# branch

```
git branch <branch name> <option>
```

ブランチを管理するコマンド

引数、オプションなしならローカルブランチの一覧を表示

オプションなし、引数 <branch name> のみなら <branch name> branch を作成する（ブランチを切る）

オプションの与え方で挙動が変わる

コミットに新たに目印を付けて分岐する

## Options

`-a (--all)`

リモート追跡ブランチも含わせてすべてのブランチを一覧表示

`-r (--remotes)`

リモート追跡ブランチを表示

`-d (--delete)`

ローカルブランチの削除

`-u`

リモート追跡ブランチ（ローカルブランチの追跡先）を決める。upstream の u

# checkout/switch

```
git checkout <commit ID>
```

/

```
git switch <branch name>
```

ブランチの移動（checkout は branch でない commit にも移動可能）

checkout の方が多機能だが、誤用によるファイル変更のリスクがある

switch はブランチ操作に特化しており、ファイルの復元などはできない

```
git checkout
```

<branch> <branch>ブランチに HEAD を移動

-b <branch> <branch>ブランチを作成してそこに移動

- 直前にいたブランチに戻る

<commit ID> <commit ID>コミットに HEAD を移動

<commit ID> -- <file> <file>を<commit ID>コミット時点の状態にする

. カレントディレクトリ“.”以下のステージされていない変更をすべて取り消す

```
git switch
```

<branch> <branch>ブランチに HEAD を移動

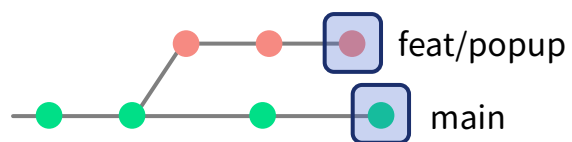
-c <branch> <branch>ブランチを作成してそこに移動

- 直前にいたブランチに戻る

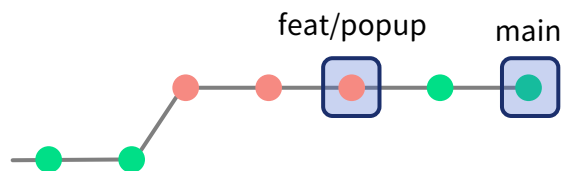
# rebase

```
git rebase
```

「ブランチの履歴を最新の履歴に“乗せ直す”操作」（ブランチの分岐元を付け替える操作）  
分岐元を最新にする→分岐元を取り込むことと同じ。つまり merge に近い作業  
commit 履歴を merge と比較して綺麗に管理できる（ログが 1 本にまとまる）



## rebase

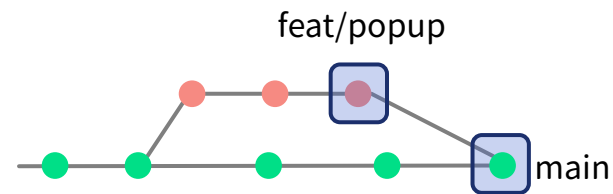


main で

```
git rebase feat/popup
```

元となる commit (base) を変える

## merge



main で

```
git merge feat/popup
```

feat/popup を HEAD に統合する

# restore

```
git restore
```

Git 管理されているファイルを削除してしまったときに復旧するコマンド

```
git status
```

で消したファイルのパスやファイル名を確認してから使うのがよい

# 宿題

- 研究室の GitLab を眺めてみよう

履歴を見ると、誰がどんな変更をしているのかがわかる

- ここまでの流れを復習する


最初は死ぬほど難しいので少しずつで OK

とにかく積極的に使ってみる

Git 管理に慣れないうちは、Git と並行してファイルを取っておいても OK

わからないことは小さいことでも質問し合う

- 計算機ゼミ最終課題は Git・GitHub/GitLab 管理で進めよう



## さいごに

- まだまだたくさんコマンドや仕様があります  
| もっと便利なもの、面白い挙動を見つけたらこっそり教えてください。