計算機科学実験及演習3
ハードウェア「Gitの使い方」

京都大学情報学科計算機科学コース

実験３ハードウェアでは、バージョン管理と設計データ・課題提出にGitHubを用います。

① **Gitの概要**
バージョン管理システムとGitの基本概念を紹介

② **Gitチュートリアル**
Gitの基本的使い方を簡単な例を用いて説明

③ **リモートリポジトリ・GitHubの概要**
本実験で使用するリモートリポジトリGitHubの紹介

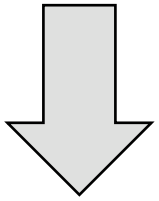
④ **GitHubチュートリアル**
GitHubの使い方を説明

⑤ **GitHub Classroom**
GitHub Classroomの説明

①Gitの概要

PCでの作業で起こりうること

- エラーを含んだ状態でプログラムを保存してしまった。
- レポートのファイルを間違って削除してしまった
- １週間ぶりに開くファイル、以前どんな編集をしたのか忘れてしまった。
- 二人で１つのファイルを編集してしまい、一方の人がした編集が反映されなかった。



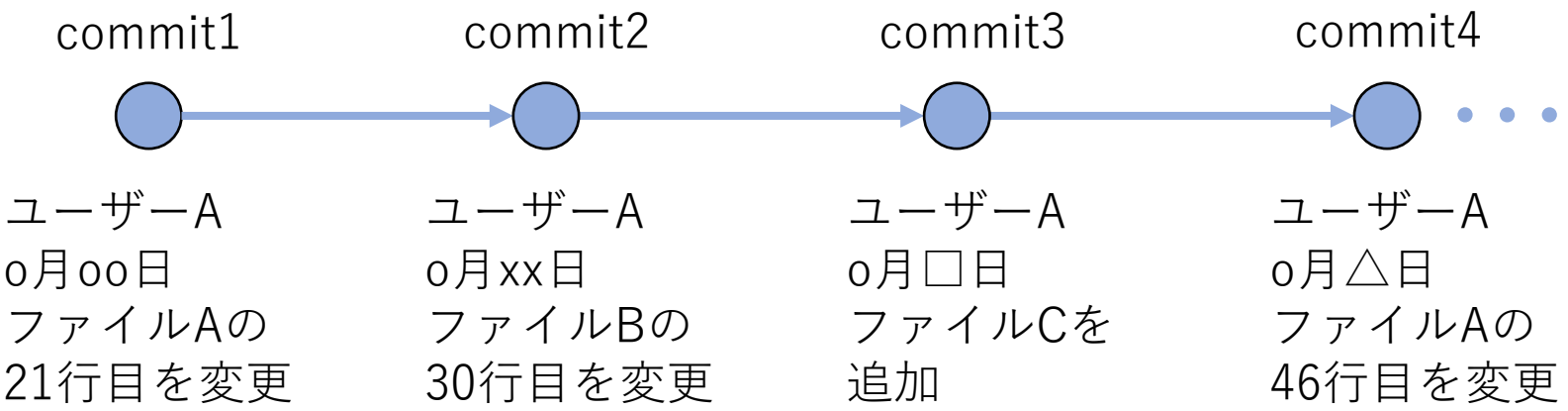
適切なバージョン管理が必要

Gitによるバージョン管理

- Gitは分散型のバージョン管理システム。
- Linuxのソースコードを管理するためにLinus Torvalds自身が開発。
- Gitの機能
  - ファイルの変更履歴を保存する。
  - 履歴には、いつ、誰が、どんな変更を行ったかを記録できる。
  - いつでもファイルを保存した履歴の状態に戻せる。
  - 他人が編集したファイルを上書きしようとすると警告を出す。

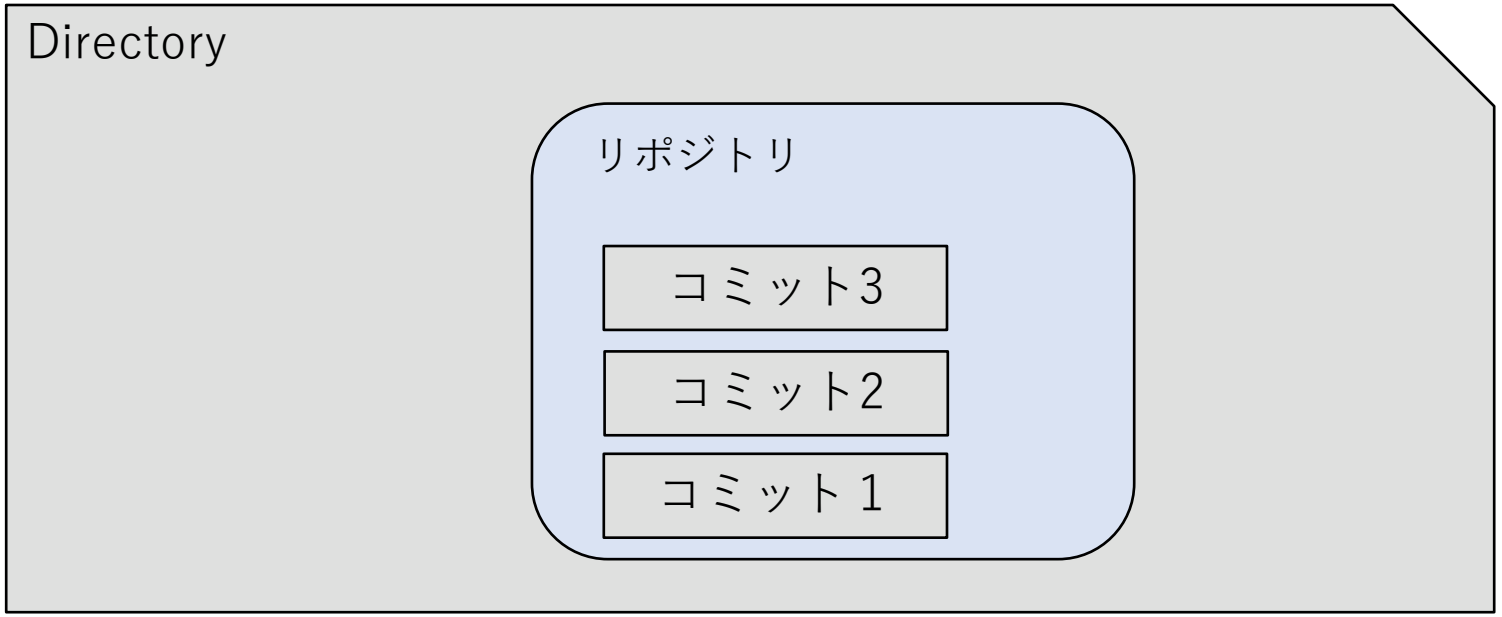
commit

- Gitでファイルの変更履歴をつけていくことをcommitという。
- commitには、いつ、誰が、どんな変更を行ったかを記録しておける。
- いつでも、任意のcommitに戻ることができる。



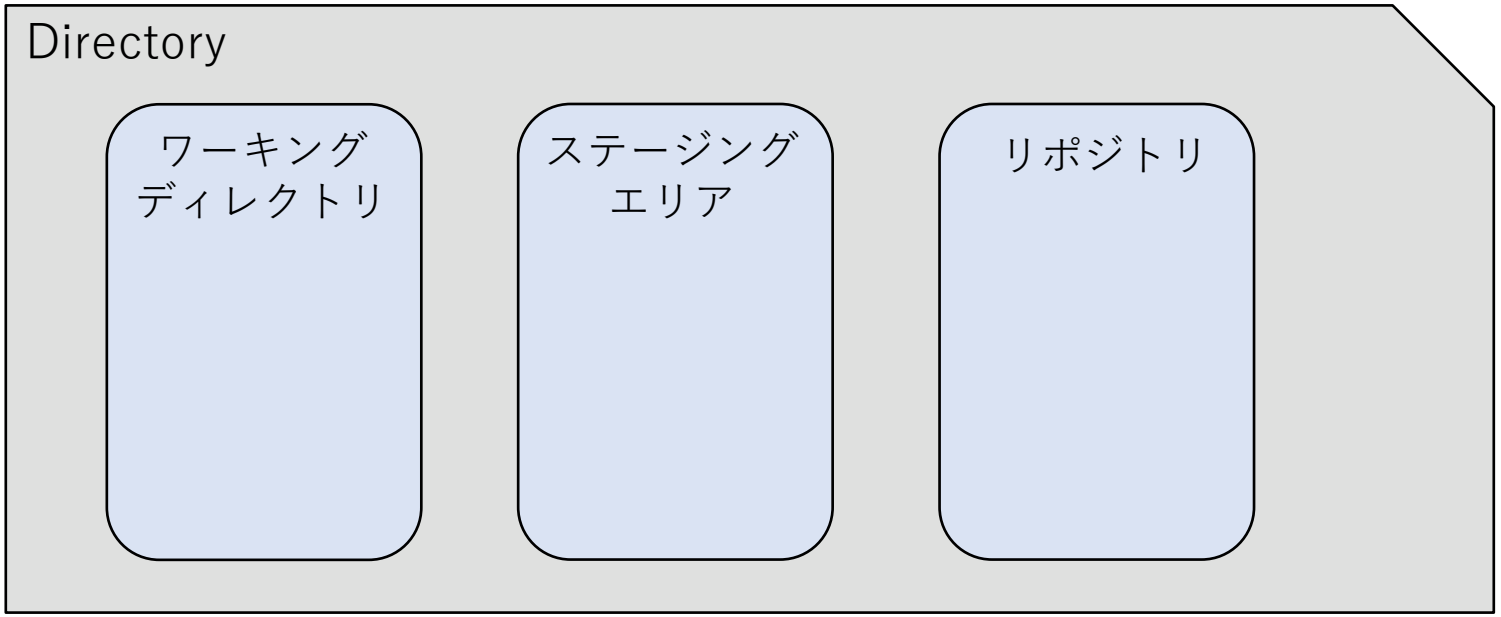
リポジトリ

- commitはリポジトリと呼ばれる場所に保存されていく。
- commitするたびに、その時のファイルの状態が保存されていく。



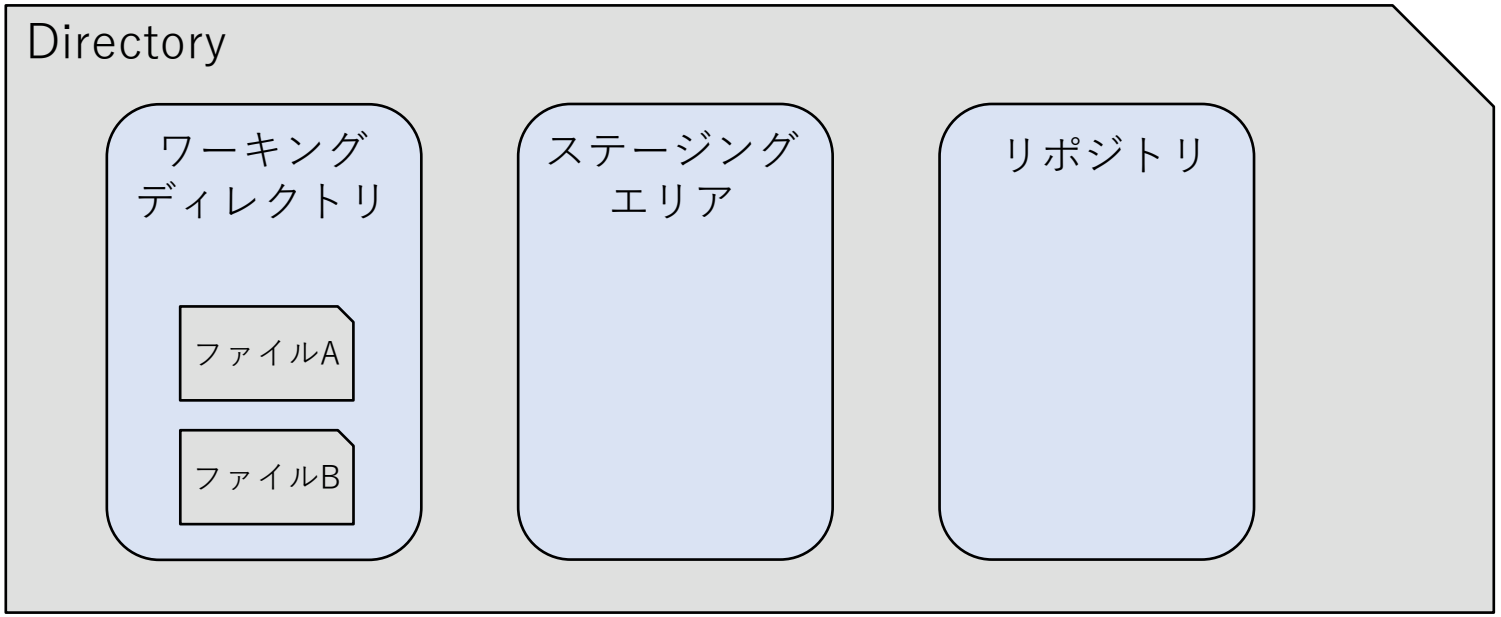
commitの手順

- ディレクトリをGitの管理下に置く（初期化する）とワーキングディレクトリ、ステージングエリア、リポジトリの３つの場所が作られる。



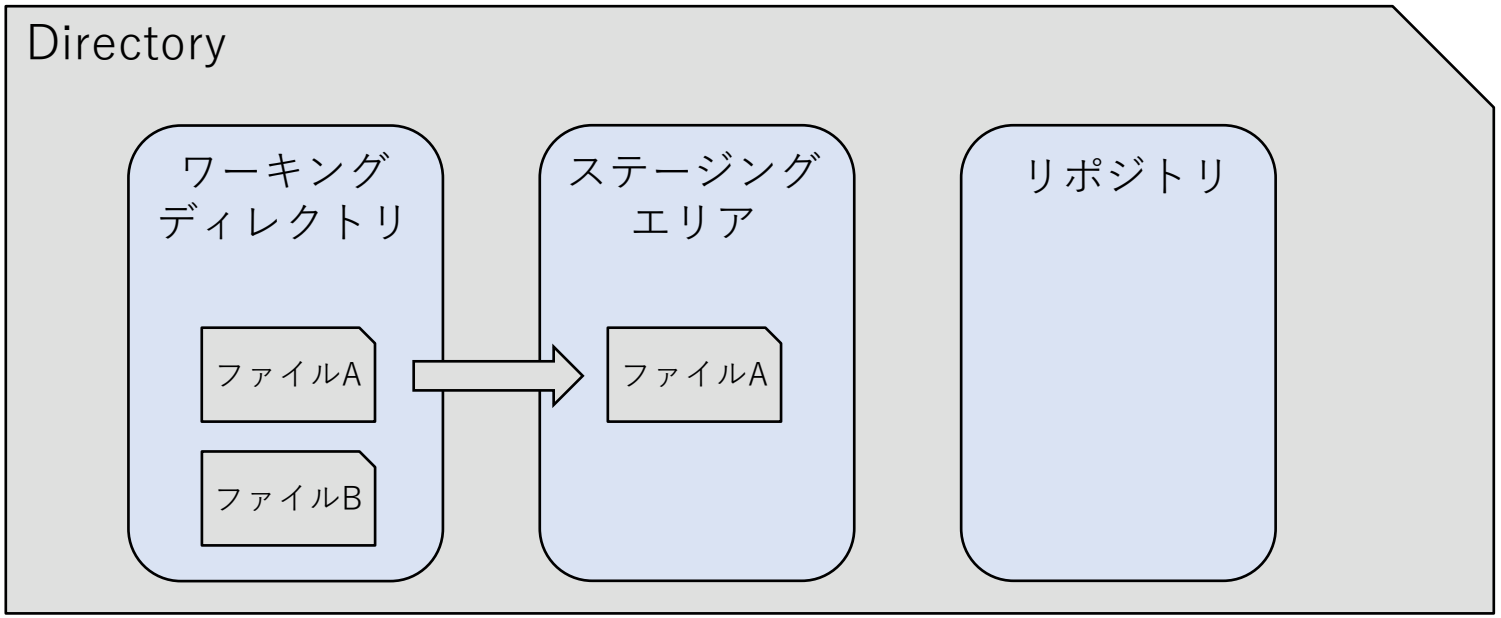
commitの手順

- Gitで管理している、実際の作業を行うディレクトリのことをワーキングディレクトリという。



commitの手順

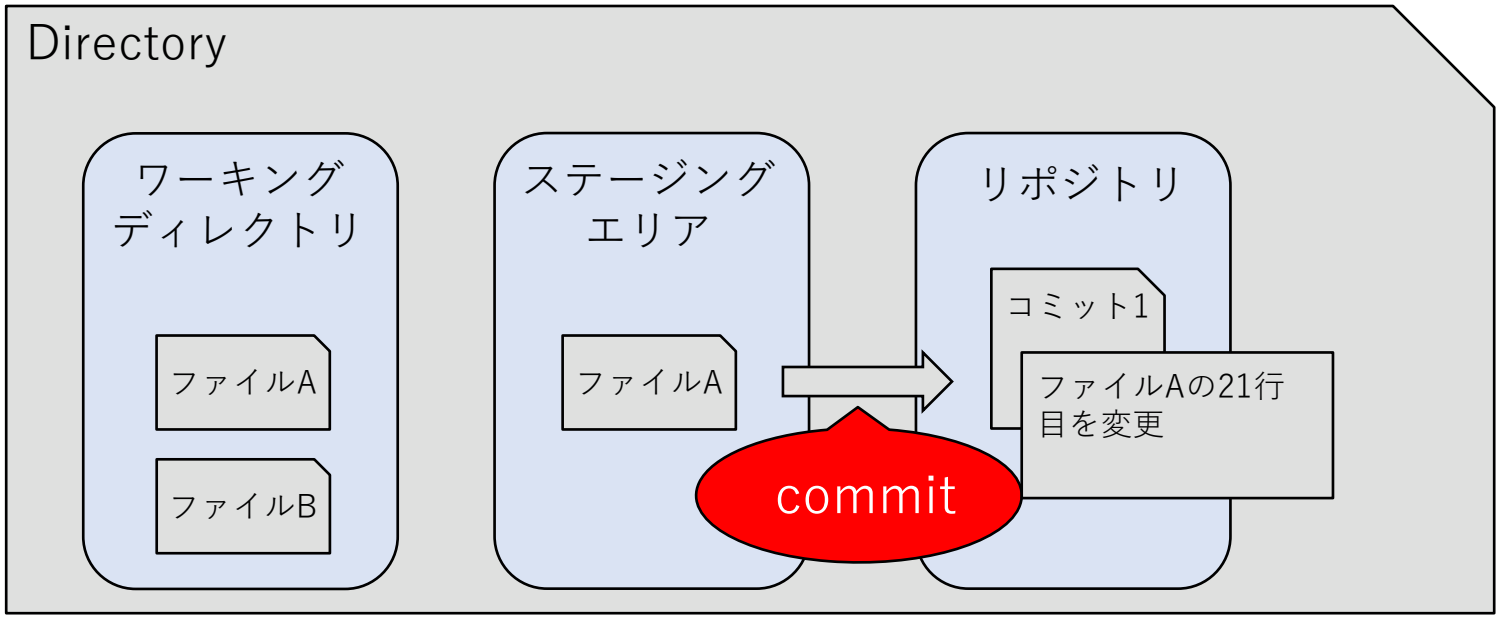
- 変更履歴として保存するファイルを選択し、置いておく場所をステージングエリアという。
- コミットの前に、ワーキングディレクトリのファイルの中から履歴として保存したいファイルを選びステージングする。



commitの手順

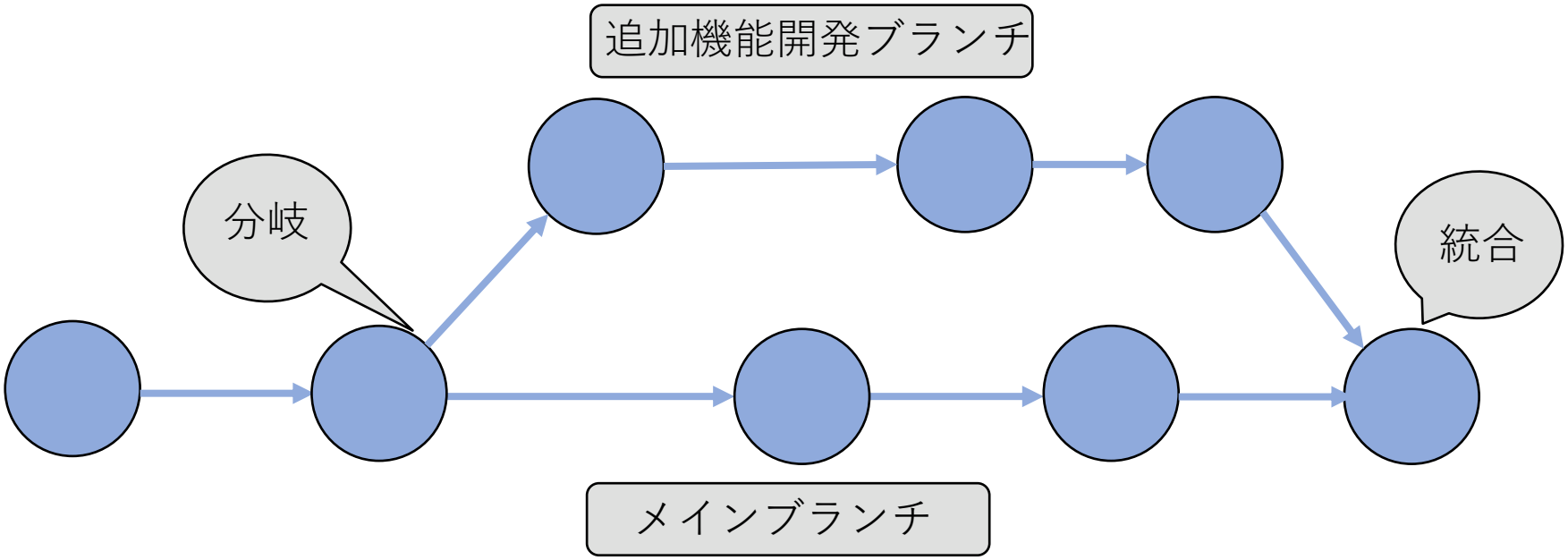
- commitをすると、ステージングエリアに置かれている
  ファイルの変更履歴が、リポジトリに保存される。

- ステージングエリアがあることで、ワーキングディレクト
  リの中の保存しておきたいファイルだけを選んで、
  commitすることができる。



branch

- 変更履歴の流れを分岐することで並行での編集を支援する機能をbranchという。
- あるブランチへの変更は、他のbranchに影響しない。
- ブランチ同士の統合（merge）も可能。



②Gitチュートリアル

Gitの利用

- **Windowsの場合**
  - Git for Windows をインストールする
  - WSL（Windows Subsystem for Linux）を利用する
    - CygwinなどのLinuxライクな環境を用意する互換レイヤーをインストールする
- **Mac の場合**
  - Xcode Command Line Tools をインストールする
    - 「git –version」などとした際に未インストールならインストール画面へ
      （近年では、install 済みであることが多い）
- **Linux の場合**
  - 「sudo apt install git-all」など

Gitの初期設定

```
$ git config --global user.name “JOHO KYODAI"
$ git config --global user.email "foo@bar.com"
```

- Gitの操作はターミナル上で行う。
- git config --globalコマンドによりユーザー名とメールアドレスを登録する。
- 登録したユーザーがcommitの際に、記録される

- 私の例：

```
$ git config list
user.email=m_nagano@i.kyoto-u.ac.jp
user.name=nagano28
```

Gitリポジトリを初期化する

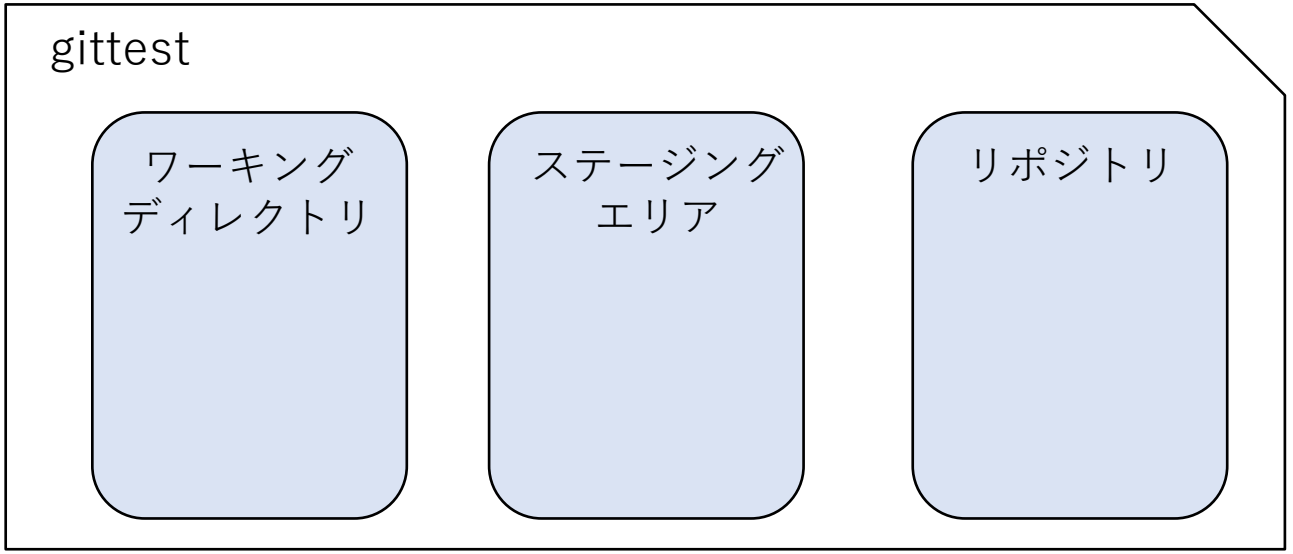
git init

現在のディレクトリをGitリポジトリとして初期化する。

Gitリポジトリを初期化する

```
$ mkdir gittest  ←フォルダを作成
$ cd gittest     ←フォルダへ移動
$ git init       ←リポジトリの初期化
Initialized empty Git repository in <ディレクトリ>
/gittest/.git
```

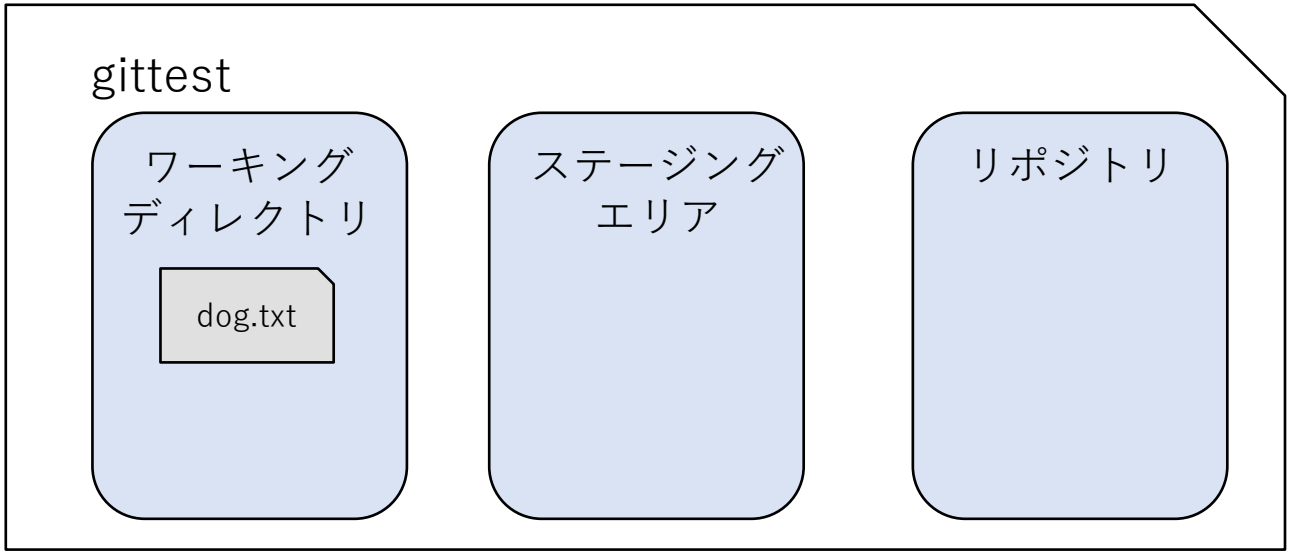
- gittestディレクトリがGitリポジトリとして認識される。
- ディレクトリには.gitディレクトリが作成されている。



ディレクトリ内にテキストファイルを作成する。

```
$ echo “bow wow” >> dog.txt
```

（Linuxにおいて、テキスト作成・編集の一般的なコマンド）
※ファイルの中身の確認がしたければ、$ cat dog.txt



ワーキングディレクトリにdog.txtが追加される。

現在のディレクトリ内の状態を確認する

git status

ディレクトリ内の状態を確認する。

現在のディレクトリ内の状態を確認する

```
$ git status
On branch main

Initial commit

Untracked files:
(Use “git add<file>…” to include what will commited)

        dog.txt

nothing added to commit but untracked files present(use
“git add” to track
```

- 新しく作成したファイル（dog.txt）がUntracked files:の一覧に表示される。
- Untracked filesには一度もステージングしたことがないファイルが表示される

指定したファイルをステージングエリアへ追加する

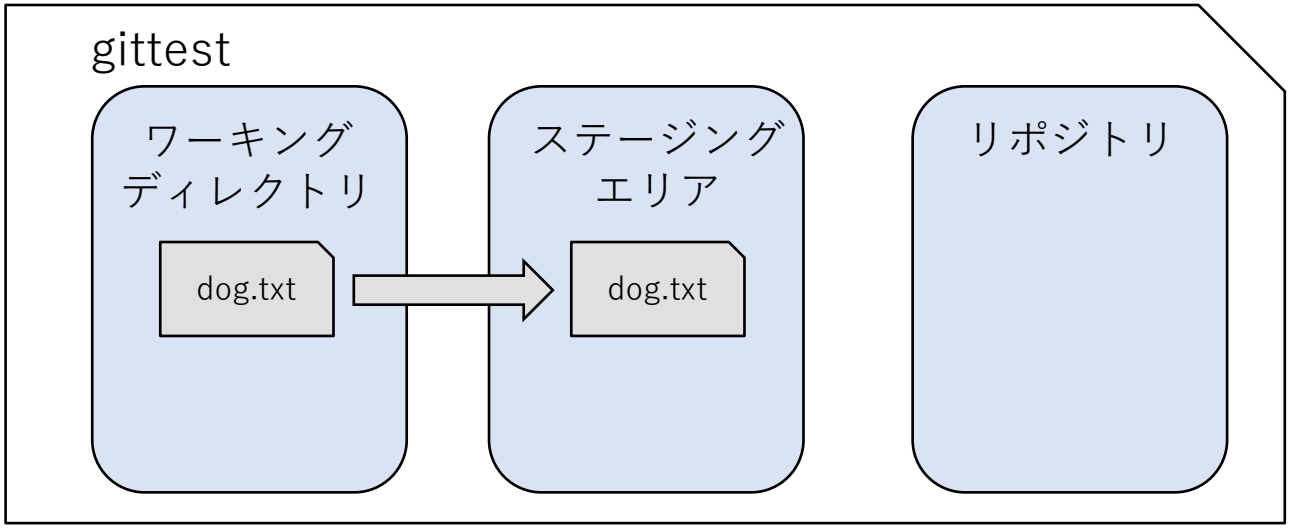
git add <ファイル名>

<ファイル名>をステージングエリアに追加する。

指定したファイルをステージングエリアへ追加する

```
$ git add dog.txt
```

- dog.txtがステージングエリアに追加される。



ステージングエリア追加後の状態を確認する

```
$ git status
…
Changes to be commited:
(Use “git rm –cached <file>…” to unstage)

        new file: dog.txt
```

- ステージング後に、再びgit statusコマンドにより状態を確認する。
- Changes to be commited:の一覧にdog.txtがnew fileとして表示されている。ステージングエリアに新たに加えられたことを示している。

ステージングされたファイルをcommitする。

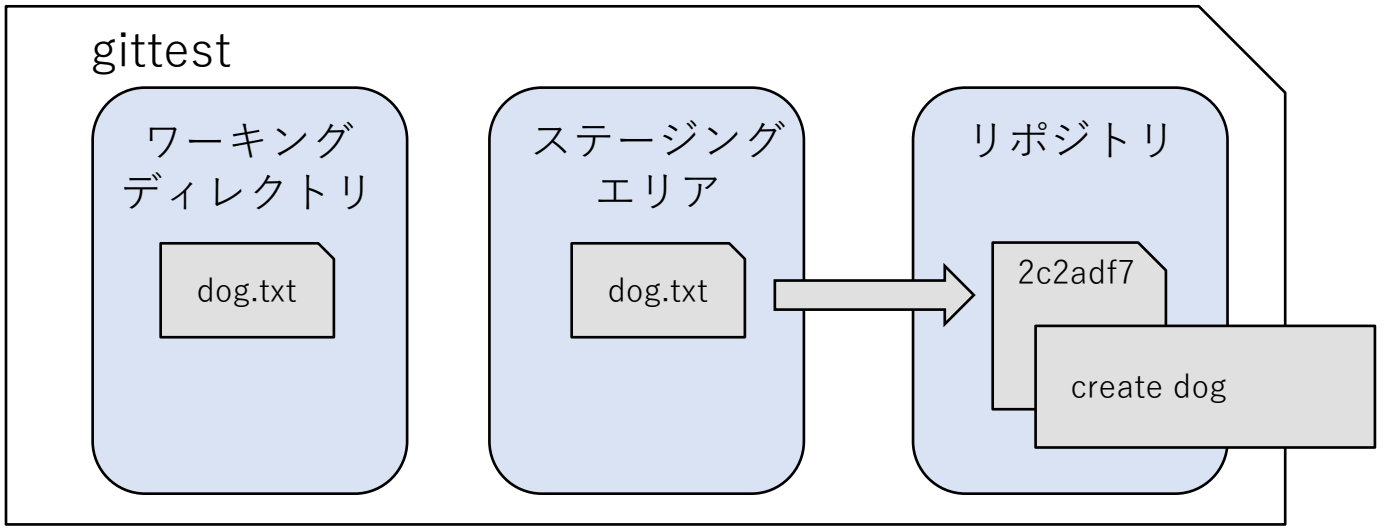
git commit -m ＜コメント＞

- ステージングされているファイルをcommitする。
- -mオプションをつけると１行のコメントをつけることができる。
  - どのような処理をしたのか、
    分かり易い見出しがオススメ。

ステージングされたファイルをcommitする。

```
$ git commit –m “create dog”
[master (root-commit) 2c2adf7] create dog
1 file changed, 1 insertion(+)
```

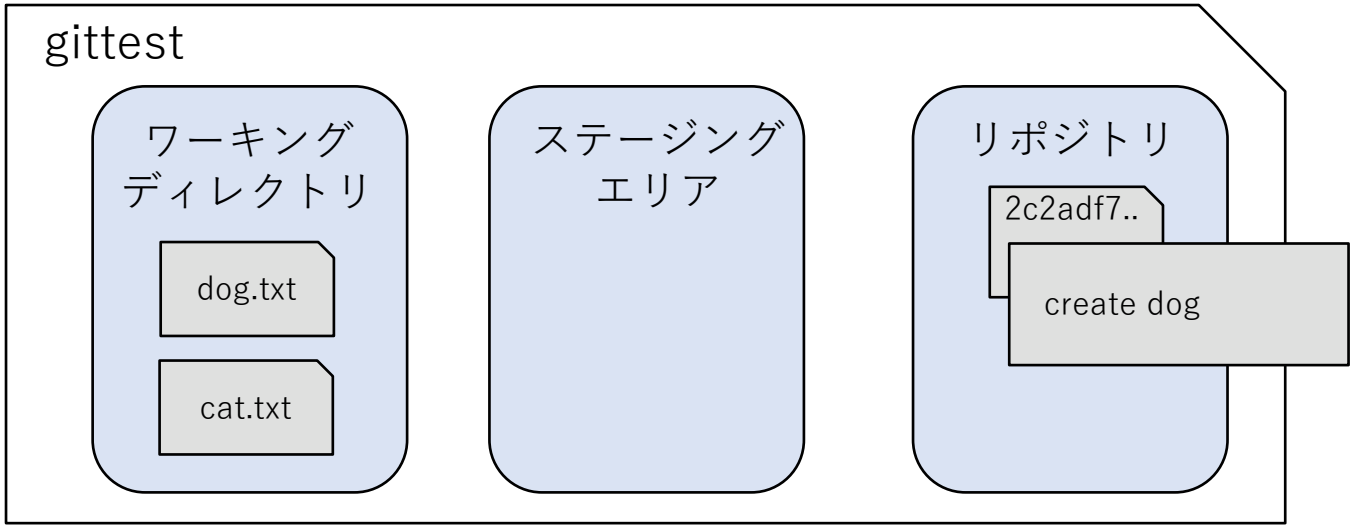
- create dogというコメントをつけてcommitする。
- ステージングされていたdog.txtがcommitされる。



dog.txtの変更と、新規ファイルの追加。

```
$ echo “mew” >> cat.txt
$ echo “wan wan” >> dog.txt
```

- dog.txtを編集し、新たにcat.txtを追加する。

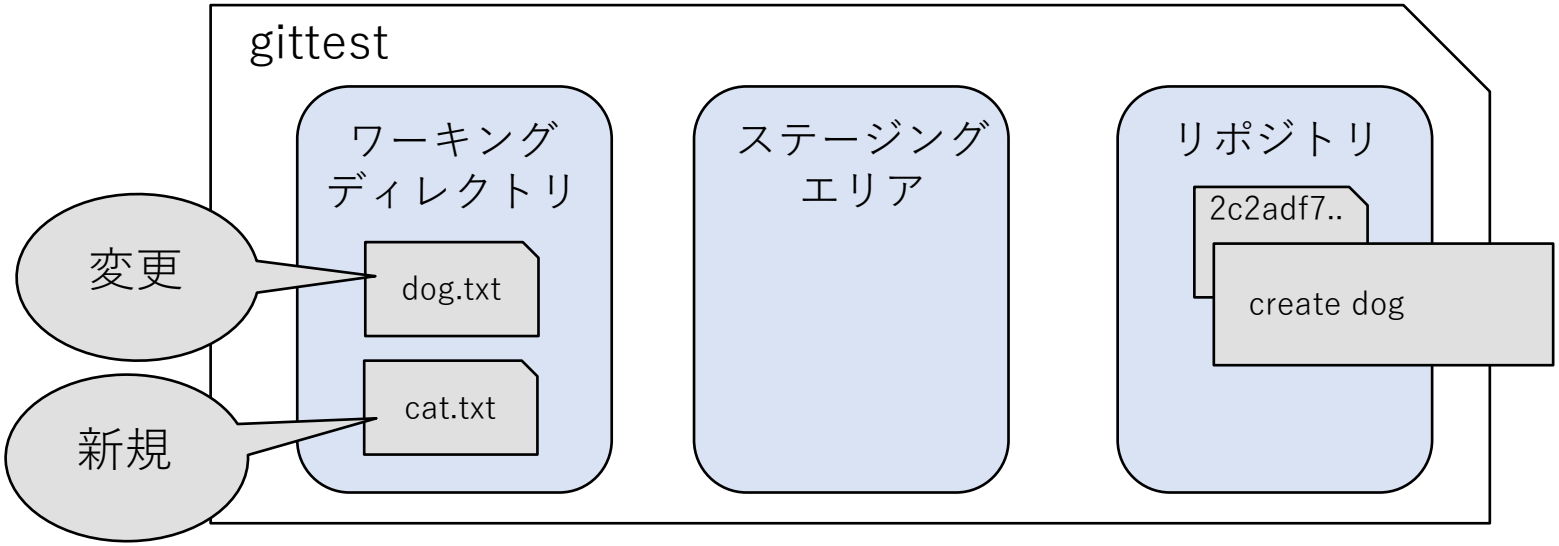


dog.txtの変更と、新規ファイルの追加。

```
$ git status
On branch main
Changes not staged for commit:
…
    modified: dog.txt
Untracked files:
…
     cat.txt
```

- Changes not staged for commitには過去にステージングしたことがあり、かつ最新のcommitから変更されているファイルが表示される。
- Untracked filesにはcat.txtが表示される。

dog.txtの変更と、新規ファイルの追加。



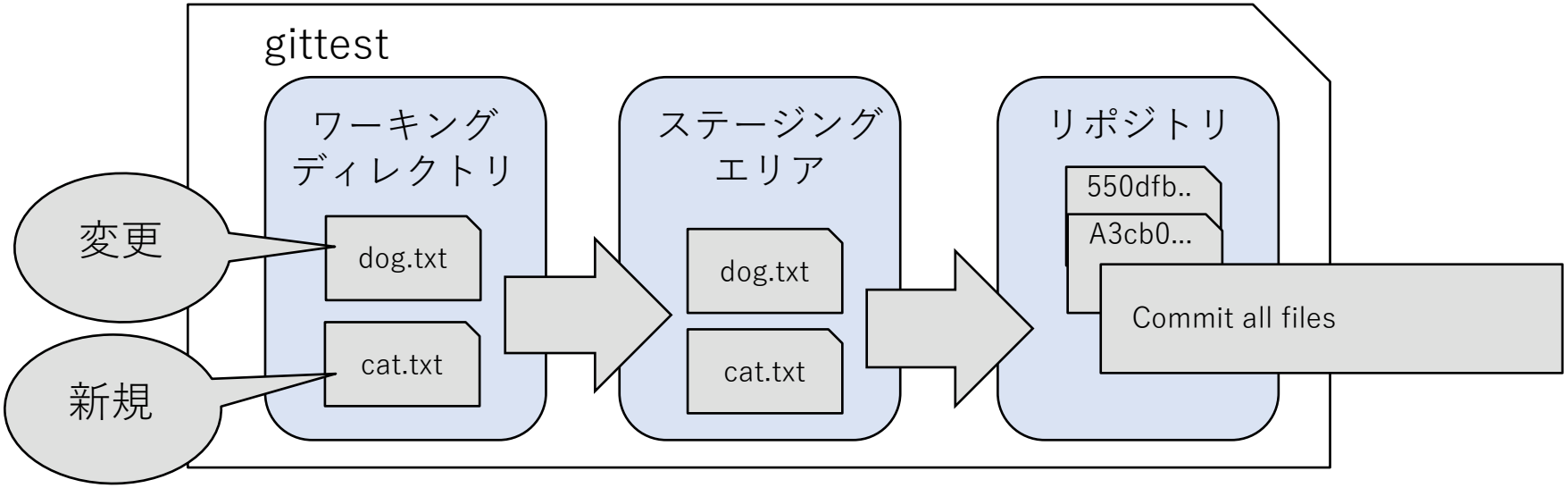
全ての変更をコミット。

```
$ git add *.txt
$ git commit –m “commit all files”
```

- dog.txtの変更とcat.txtの追加が一つの変更としてコミットされる。
- git addのファイル指定ではワイルドカードが使える。
- “.”を指定すると変更があった全てのファイルをステージングする。

[＋α]

- git add --allなどで全てのファイルをステージング
- git reset xxxで、commitやaddを取り消し可能（xxxは用途によって変更）



commitの履歴を確認する

git log

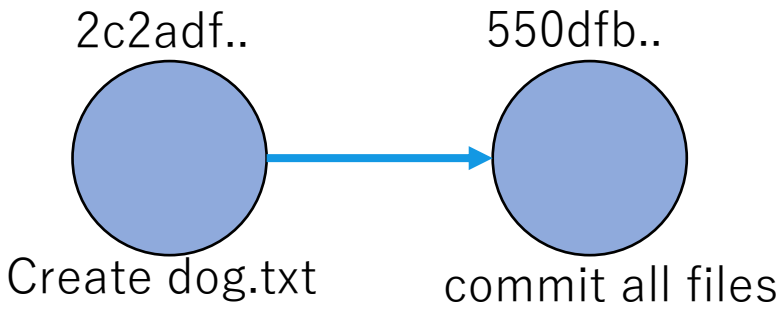
リポジトリの保存されているcommitの履歴を確認する。

commitの履歴を確認する

```
$ git log
commit 550dfb1f4de340c3abe04549f52c4a6598ad4cbf
Author: Kazunari Kato<kato@example.com>
Date: Fri Apr 27 14:01:23 2018 +0900

        commit all files

commit 2c2adf7291e16fd4301a625c8ac5589edc703f83
Author: Kazunari Kato<kato@example.com>
Date: Fri Apr 27 13:58:26 2018 +0900

         create dog.txt
```

- commit履歴が新しいものから順番に表示される。
- １行目の文字列はcommit IDといって、commitを識別するためのもの。

commitの履歴を確認する



以前のcommitの状態に戻す

```
git reset --hard <commit id>
```

ファイルを<commit id>で指定したcommitの状態に戻す。

以前のcommitの状態に戻す

```
$ echo “tweet” >> bird.txt
$ echo “meow meow” >> cat.txt
$ git status
// bird.txtの追加とcat.txtが変更されているメッセージが出る。
$ git reset --hard //コミットIDを省略すると最新のコミットに戻る
$ git status
// 変更が何もないというメッセージがでる。（bird.txtに関して以外）
```

- 新たにbird.txtを追加し、cat.txtを変更する。
- git reset --hardコマンドで、ファイルを最新のcommitの状態に戻す。
  - どちらかで、変更が何もないというメッセージがでる。
    - .gitignoreでbird.txtを追跡対象から外す。
    - git clean -fdコマンドで、未追跡ファイルを取り除く（bird.txt）
- lessコマンドなどで、ファイルを確認すると、元に戻っている。

Gitでトラッキング対象を管理する。

.gitignore

リポジトリ内の変更履歴を追跡するファイルを管理する。

一つのファイルの追跡を外す

```
$ touch .gitignore
$ echo "bird.txt" >> .gitignore
$ git status
// 変更が何もないというメッセージがでる。
```

- .gitignoreにbird.txtを管理しないように記載
- 変更履歴が不要なファイルや作業対象でないファイルなどを管理する。

branch一覧を表示する。

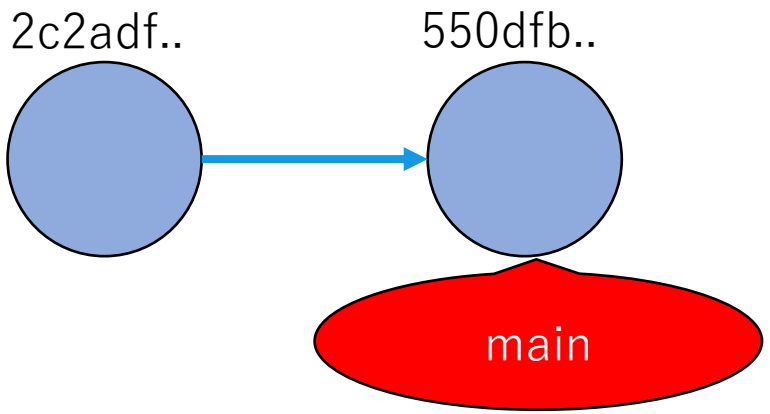
```
git branch
```

作成されているbranchの一覧を表示する。

branchを利用する-branch一覧を表示する。

```
$ git branch
* main
```

- 作成済みのbranchが表示される。
  - main branchは最初から存在している。
    - （デフォルトが master branch の場合も）
- ＊は現在選択（checkout）されているbranchを示している。



新しくbranchを作成する

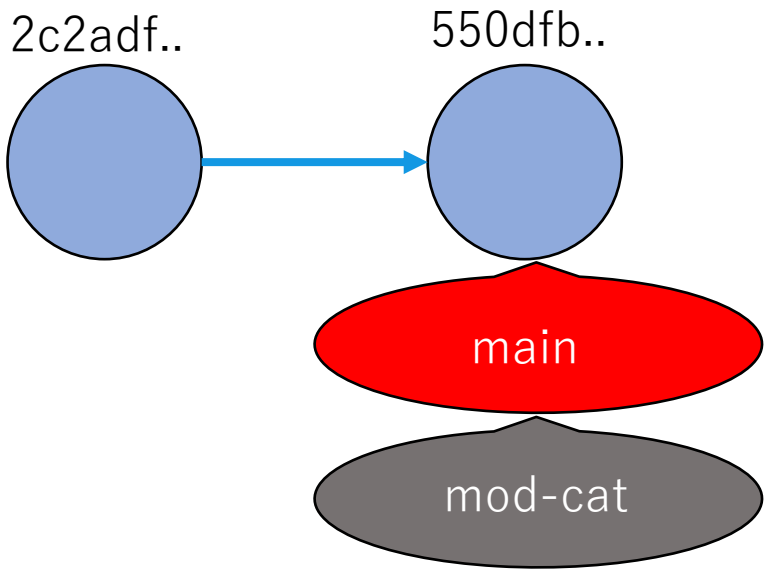
```
git branch <branch名>
```

新しくbranchを作成する。

新しくbranchを作成する

```
$ git branch mod-cat
$ git branch
* main
  mod-cat
```

- mod-catというbranchを作成する。
- git branch で確認するとmod-cat branchが作成されているのがわかる。



branchを選択する

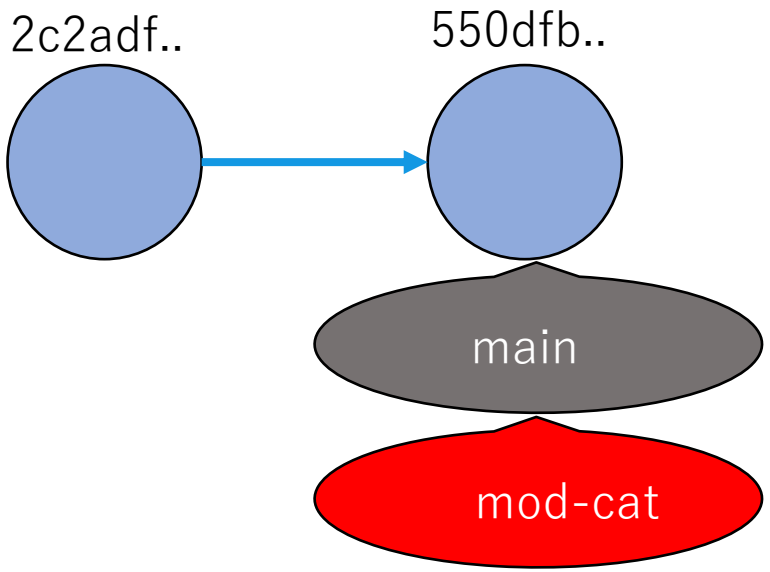
```
git checkout <branch名>
```

<branch名>で指定したブランチを選択する。

branchを選択する

```
$ git checkout mod-cat
Switched to branch ‘mod-cat’
$ git branch
  main
* mod-cat
```

mod-catブランチを選択して、git branchコマンドでmod-catが選択されていることを確認する。



（参考）branchを作成して選択する

git checkout –b <branch名>

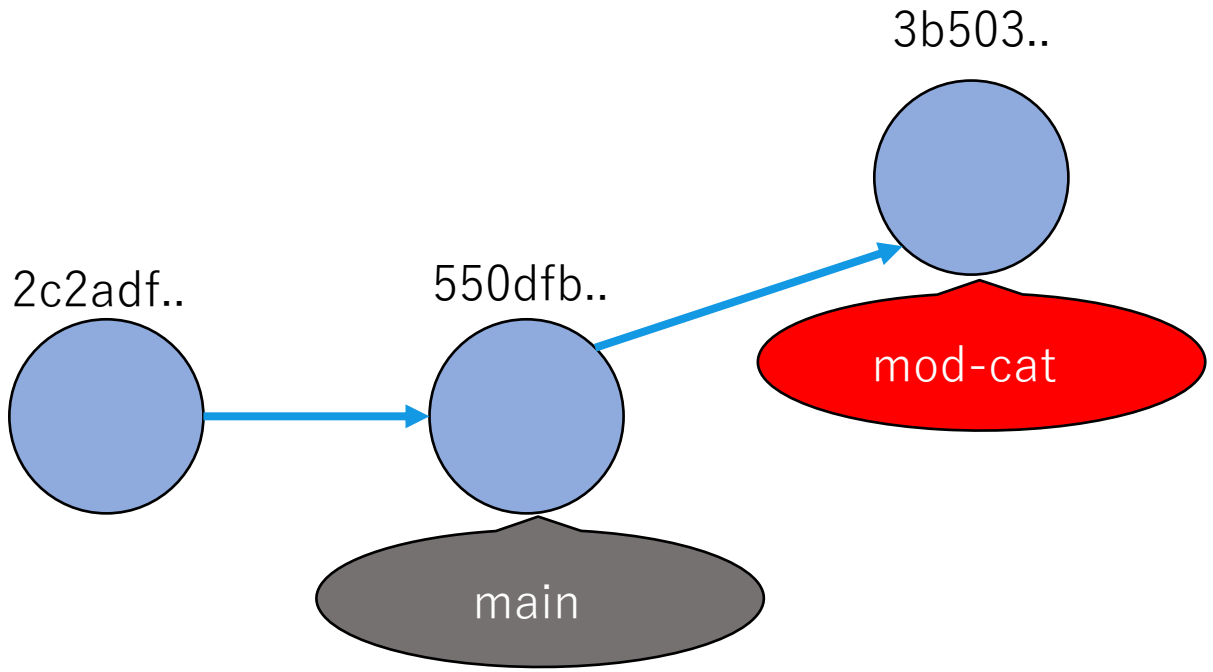
-b コマンドでを使うことで新規ブランチの作成と選択を同時に行うことも出来る。

mod-catブランチでcommitする

```
$ echo “nya-” >> cat.txt
$ git add cat.txt
$ git commit –m “mod cat”
$ git log --oneline
* 3b503ba mod cat
* 55d0fb1f commit all files
* 2c2adf72 create dog
```

- cat.txtを変更して、commitする。
- git log コマンドに--onelineオプションをつけると履歴を省略形で表示する。
- ３つのcommitがあることを確認できる。

mod-catブランチでcommitする



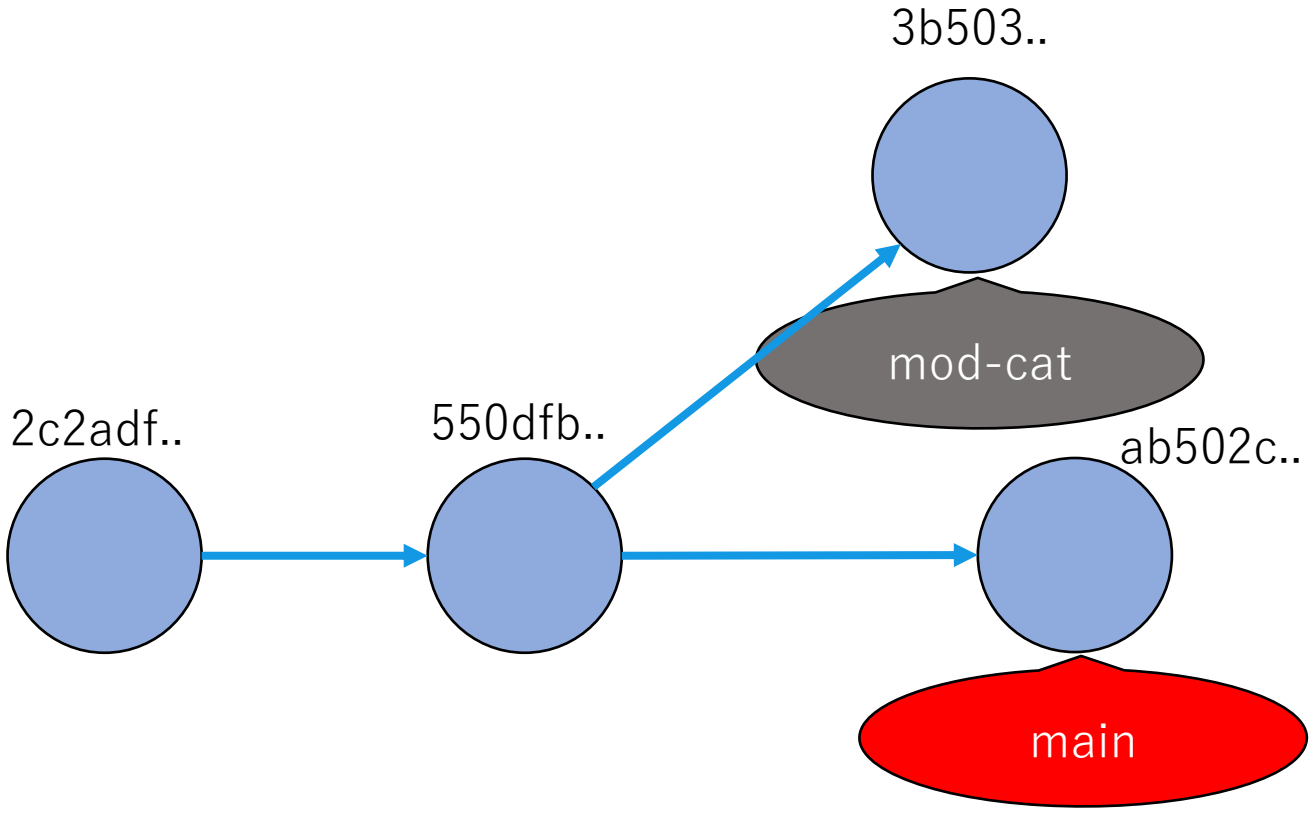
- 作成されたcommitはmod-catブランチで更新される。
- mainブランチは変わらずに550dfb..のcommitを保持している。

mainブランチを更新する

```
$ git checkout main
$ echo “waon” >> dog.txt
$ git add dog.txt
$ git commit –m “mod dog”
$ git log
* e6fa391 mod dog
* 55d0fb1 commit all files
* 2c2adf7 create dog
```

- mainブランチを選択してdog.txtを変更しcommitする。
- commit履歴を確認するとmod-catブランチでのcommitは含まれておらず、mainブランチで行なったcommitのみが表示される。

mainブランチでcommitを行う



- 新しく作成されたcommitはmainブランチで更新される。
- mod-catブランチで行われた変更はmainブランチには影響されない。

branchをmergeする

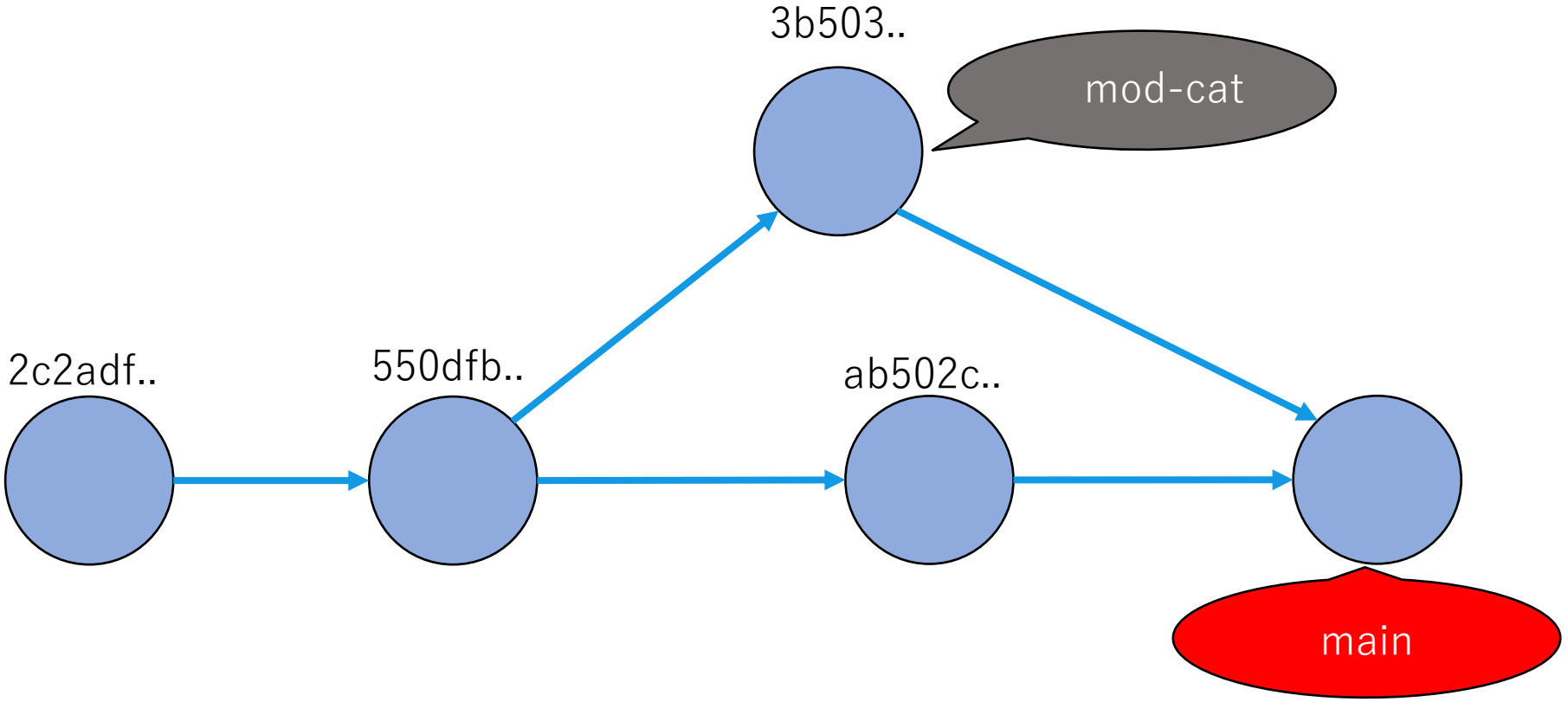
git merge <branch 名>

現在選択されているbranchに<branch 名>で指定したbranchをmergeする。

mainブランチにmod-catブランチをmergeする

```
$ git checkout main
$ git merge mod-cat
```

- mainブランチにcheckoutして、mod-catブランチをmergeする。



mainブランチにmod-catブランチをmergeする

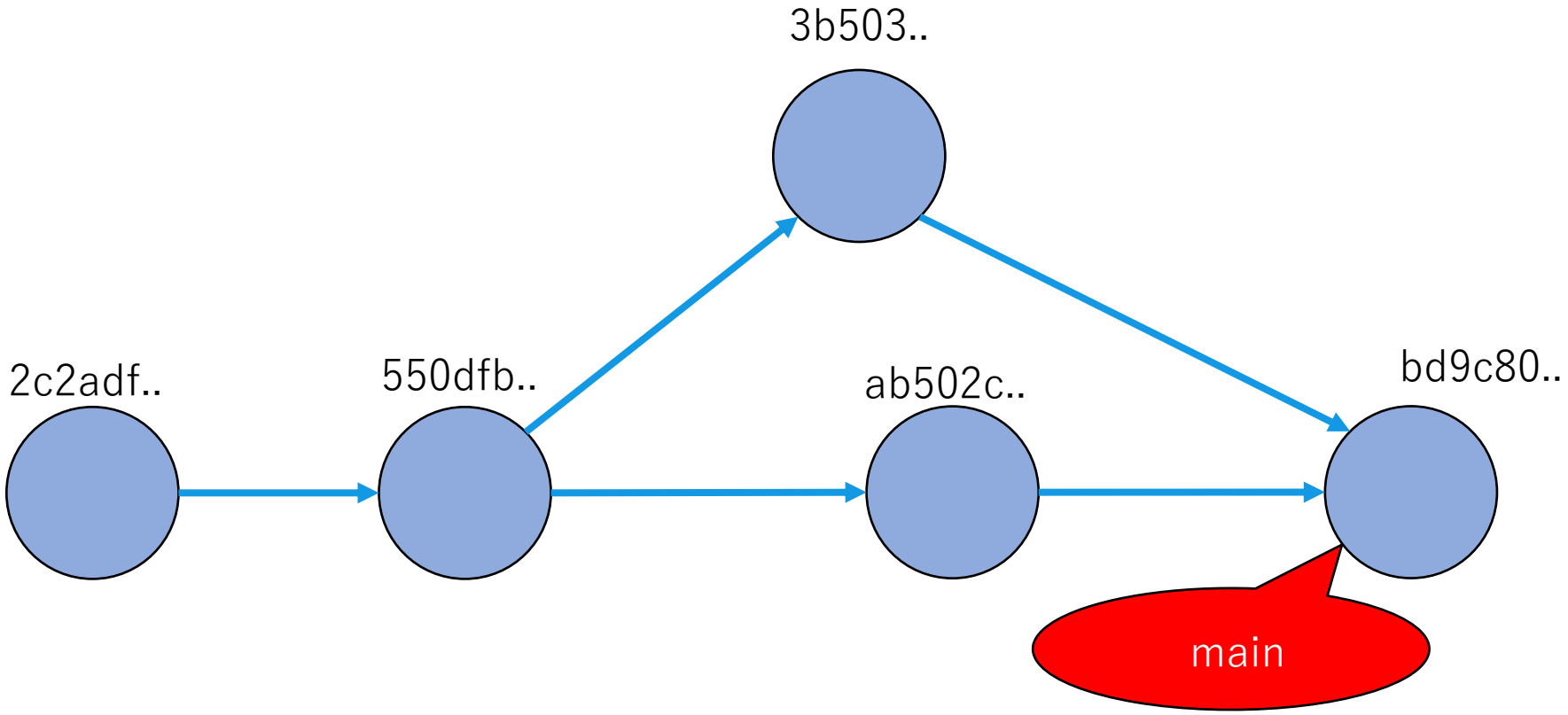
```
Merge branch ‘mod-cat’

#........
//省略
```

- merge commitを行うためのエディタが立ち上がるので、必要に応じてコメントを修正し、保存・終了する。
  - エディタはvimなので、vimのコマンドを参照

mainブランチにmod-catブランチをmergeする

```
$ git log --oneline --graph
//省略
```

- git logコマンドに--graphオプションをつけると、履歴の分岐をグラフィカルに表示する。



branchを破棄する

```
git branch -d <branch 名>
```

git branchに-dオプションをつけると<branch 名>で指定したbranchを破棄する。

branchを破棄する

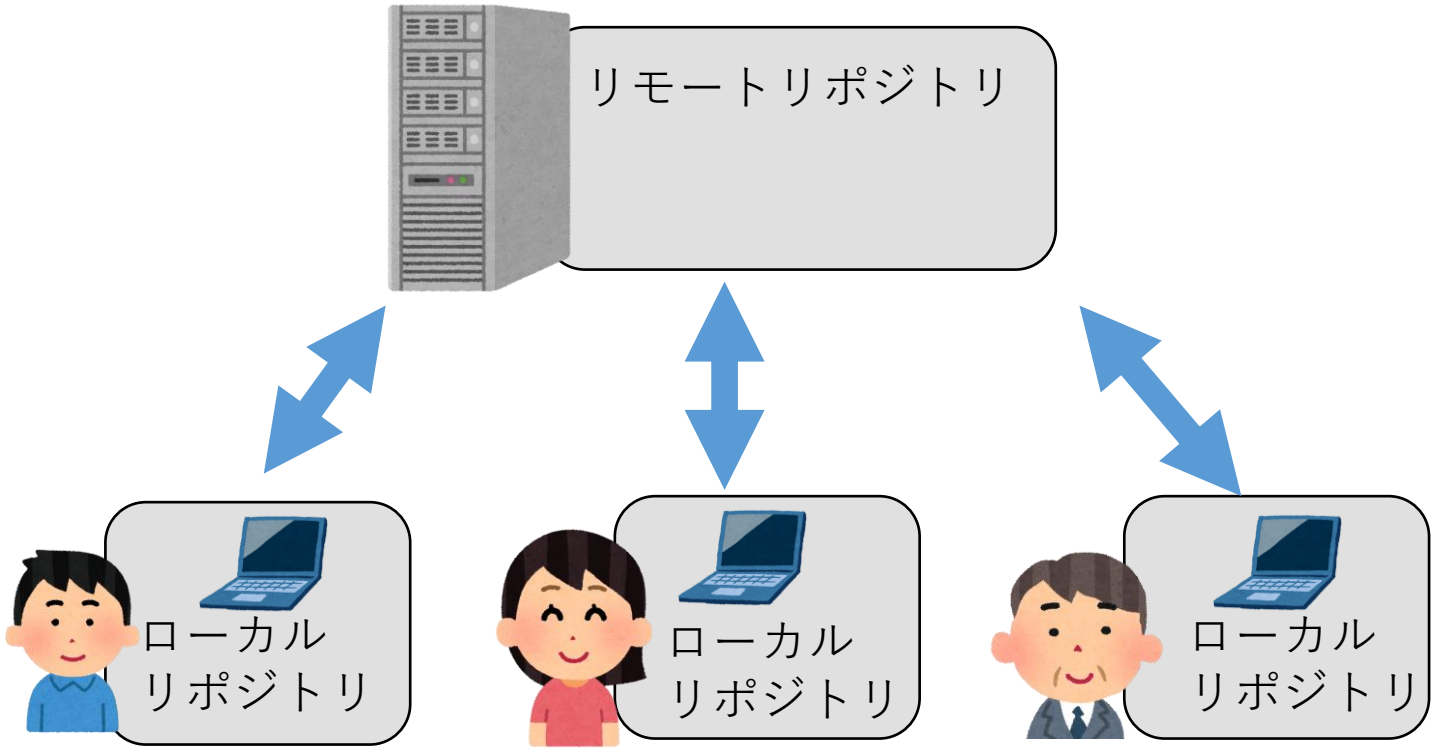
```
$ git branch -d mod-cat
Delete branch mod-cat (was 23e1793)
```

- 機能追加が完了し、不必要となったbranchは消去する。

③リモートリポジトリ・GitHubの概要

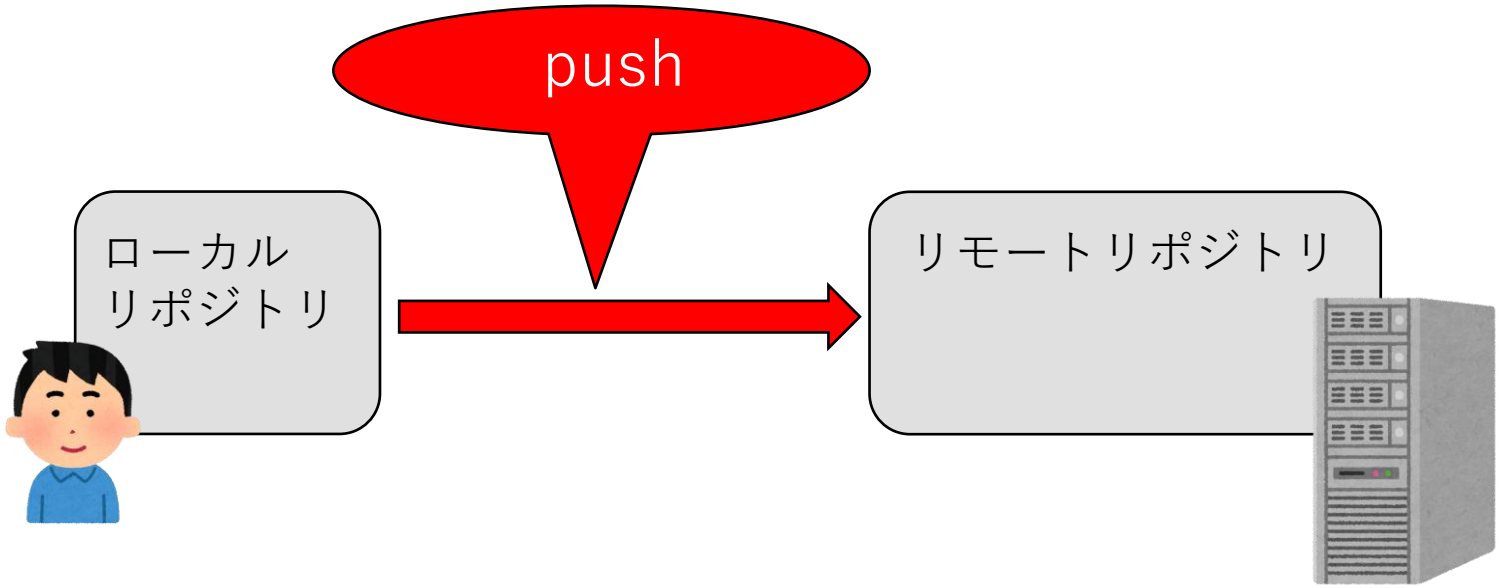
ローカルリポジトリとリモートリポジトリ

- 自分の端末に存在し、個人で使用するリポジトリをローカルリポジトリ、サーバー上に存在し、複数の人が参照することのできるリポジトリをリモートリポジトリという。



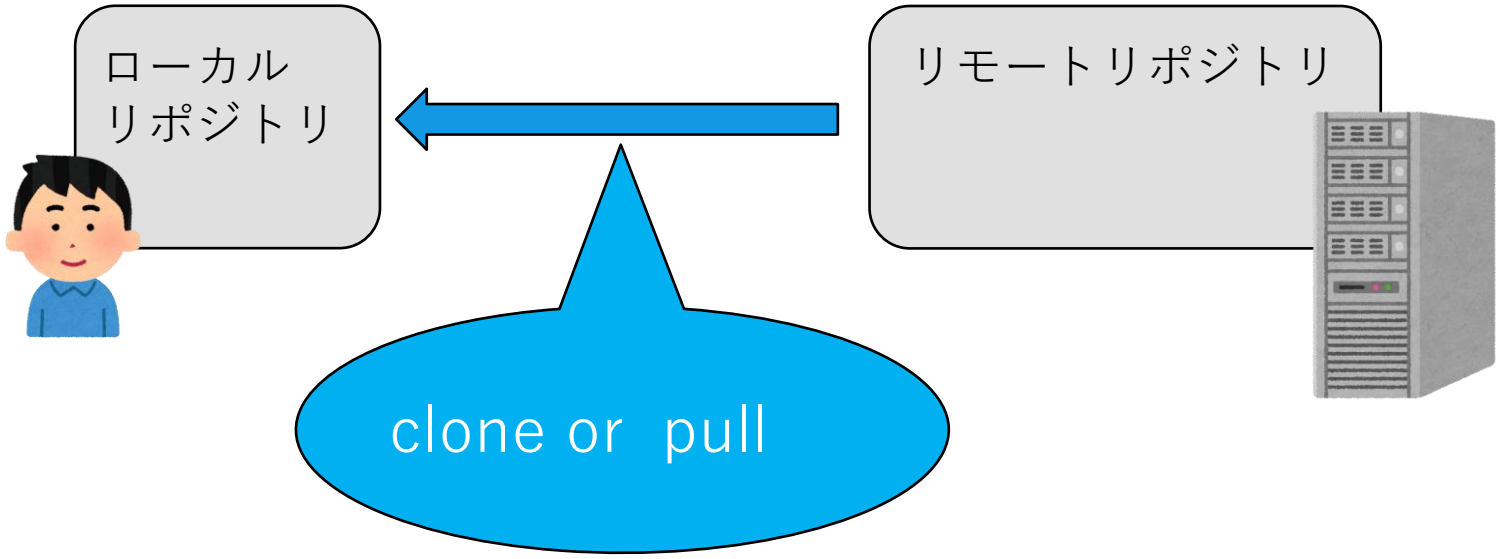
Push

- リモートリポジトリに自分のローカルリポジトリにある変更履歴をアップロードすることをpushという。



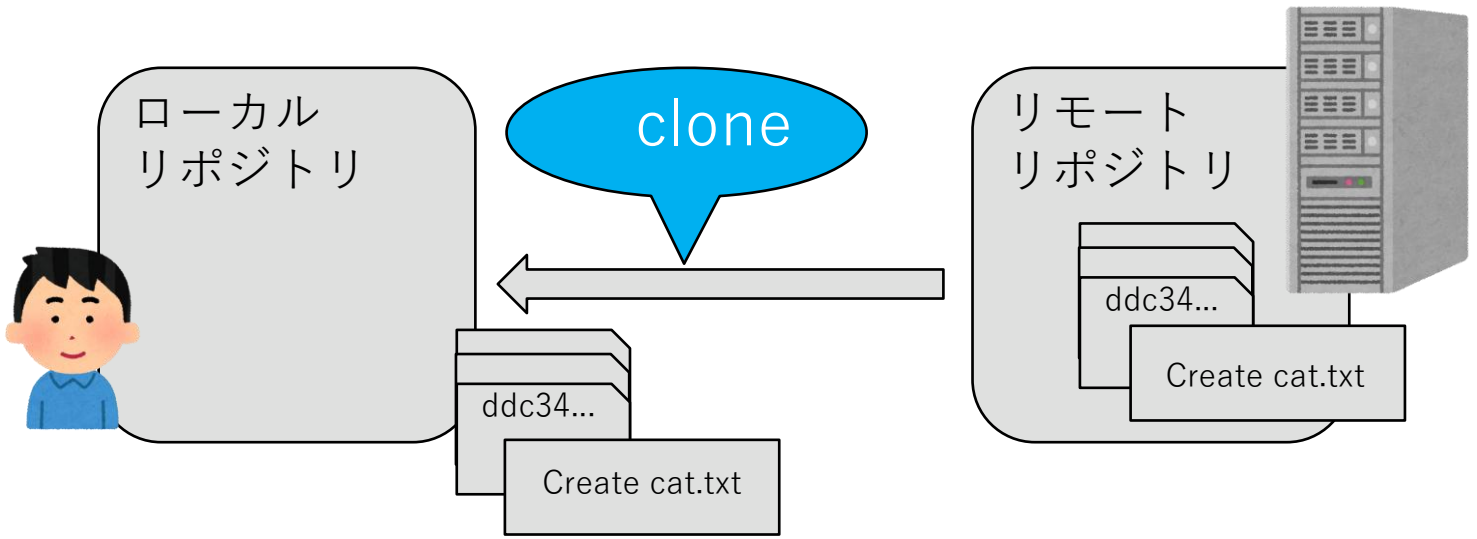
cloneとpull

- リモートリポジトリにある変更履歴を自分のローカルリポジトリにダウンロードする方法に、cloneとpullがある。



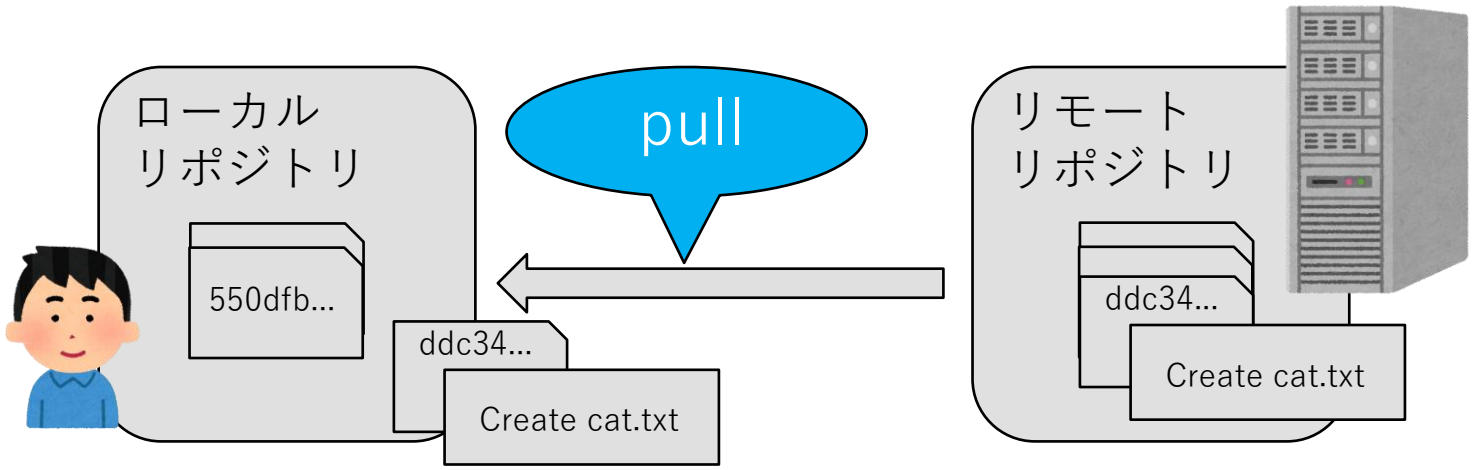
clone

- cloneは空のローカルリポジトリにリモートリポジトリのコピーを作成する時に使用。



pull

- pullはローカルリポジトリをリモートリポジトリの履歴で更新する時に使用。



GitHub

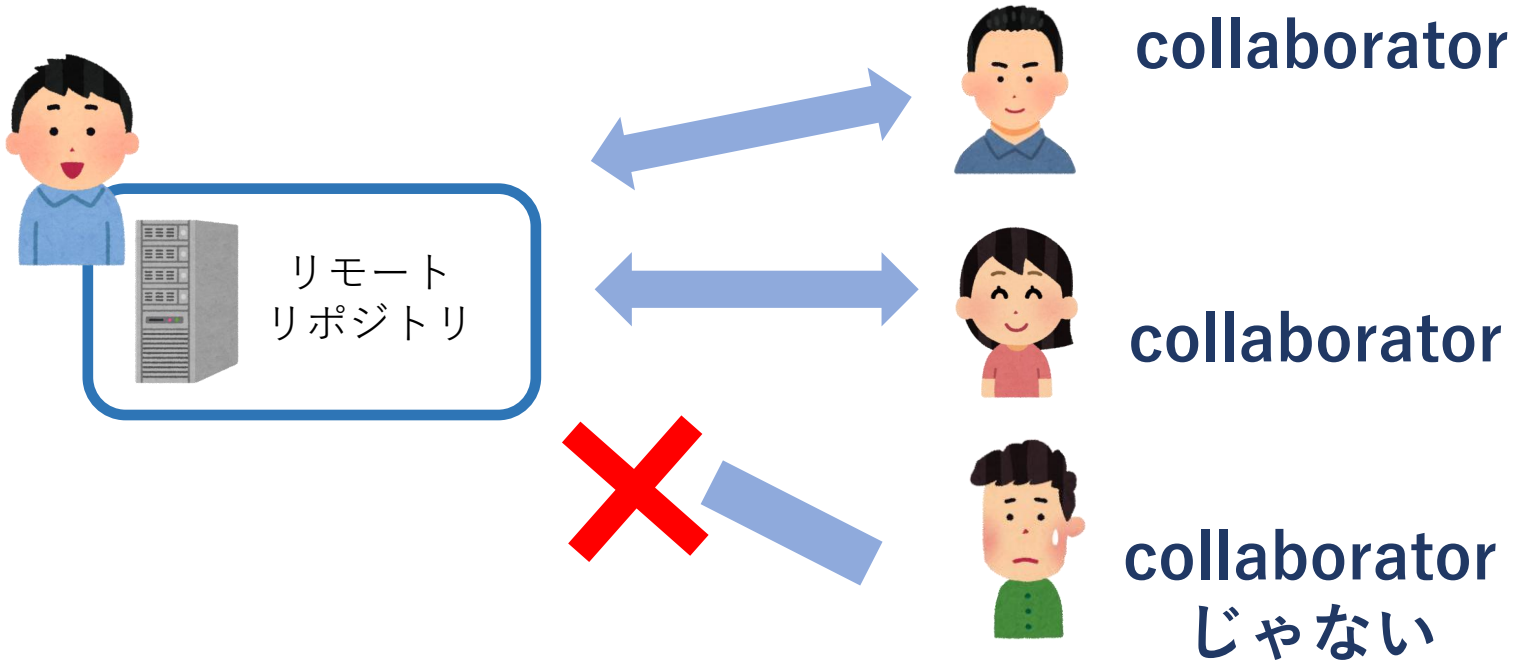
- GitHubはGitの仕組みを利用して、プログラムのソースコードなどを共有・ホスティングできるサービス。
  リモートリポジトリとして使用する。

リポジトリの公開設定

- GitHubでは自分のリポジトリをpublicとprivateいずれかに設定できる。
- Publicでは不特定多数の人がアクセス可能となる。
- Privateでは自分とcollaboratorに設定した人しかアクセスができない。
- 本実験で扱うリポジトリは必ずprivateにすること。

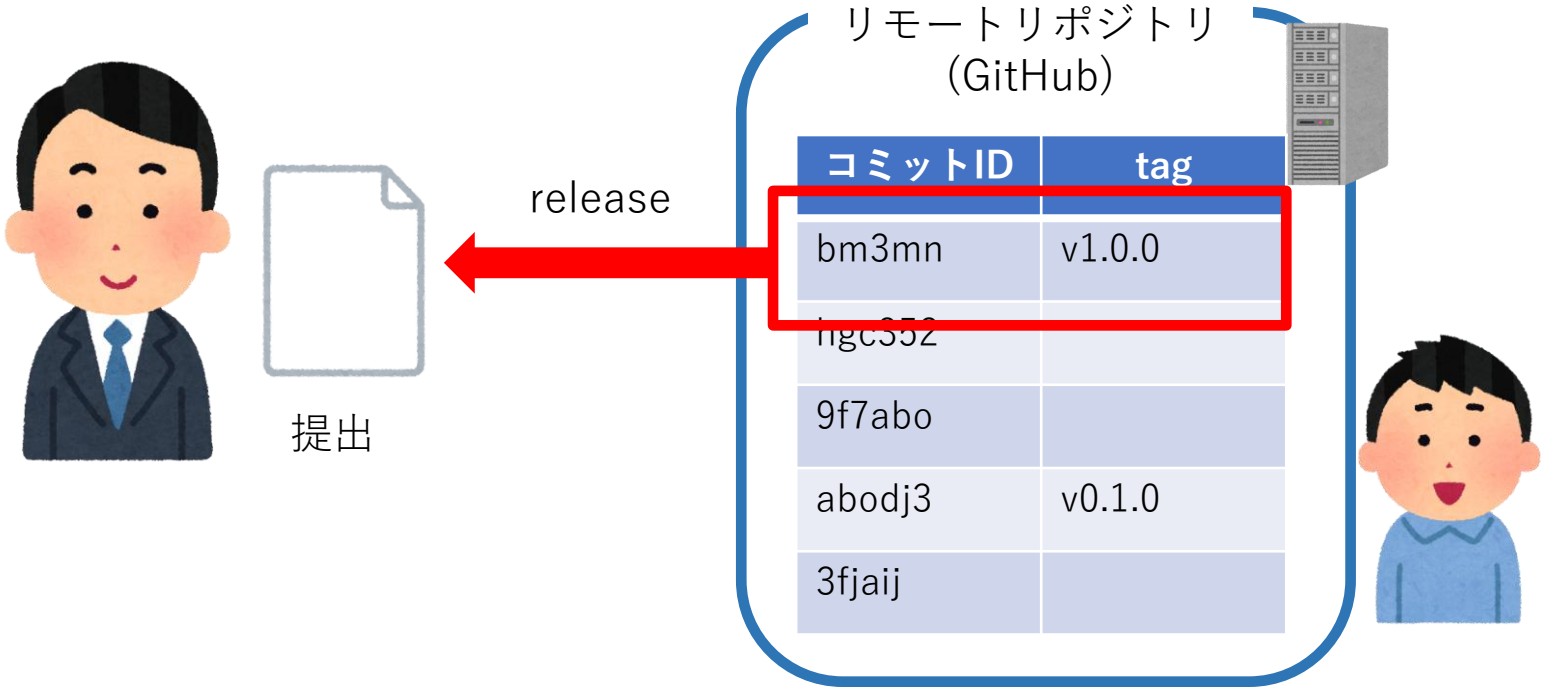
collaborator

- 自分のprivateリポジトリに他のユーザを**collaborator**として登録すると、リポジトリを共有することができる。



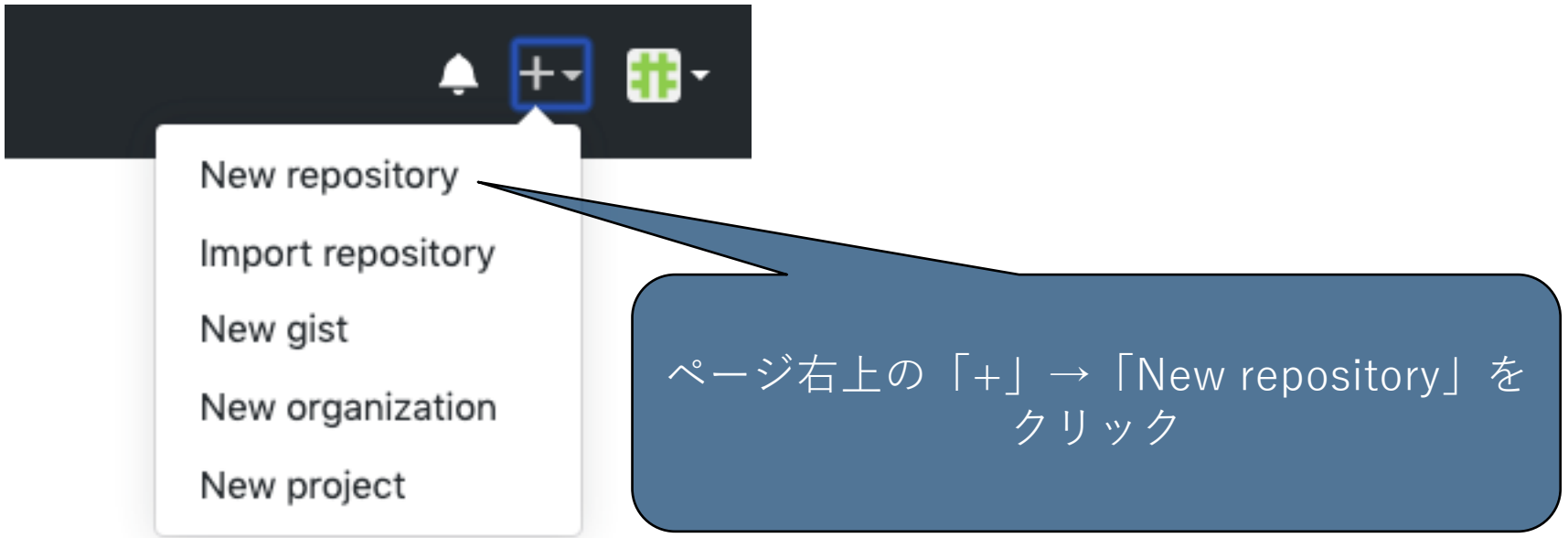
tagとrelease

- 重要なコミットに分かりやすい名前をつけるための機能をtagという。
- tagがつけられたcommitを、他の人が使用できるように提供する機能をreleaseという。本実験ではreleaseにより課題の提出を行う。



④GitHubチュートリアル

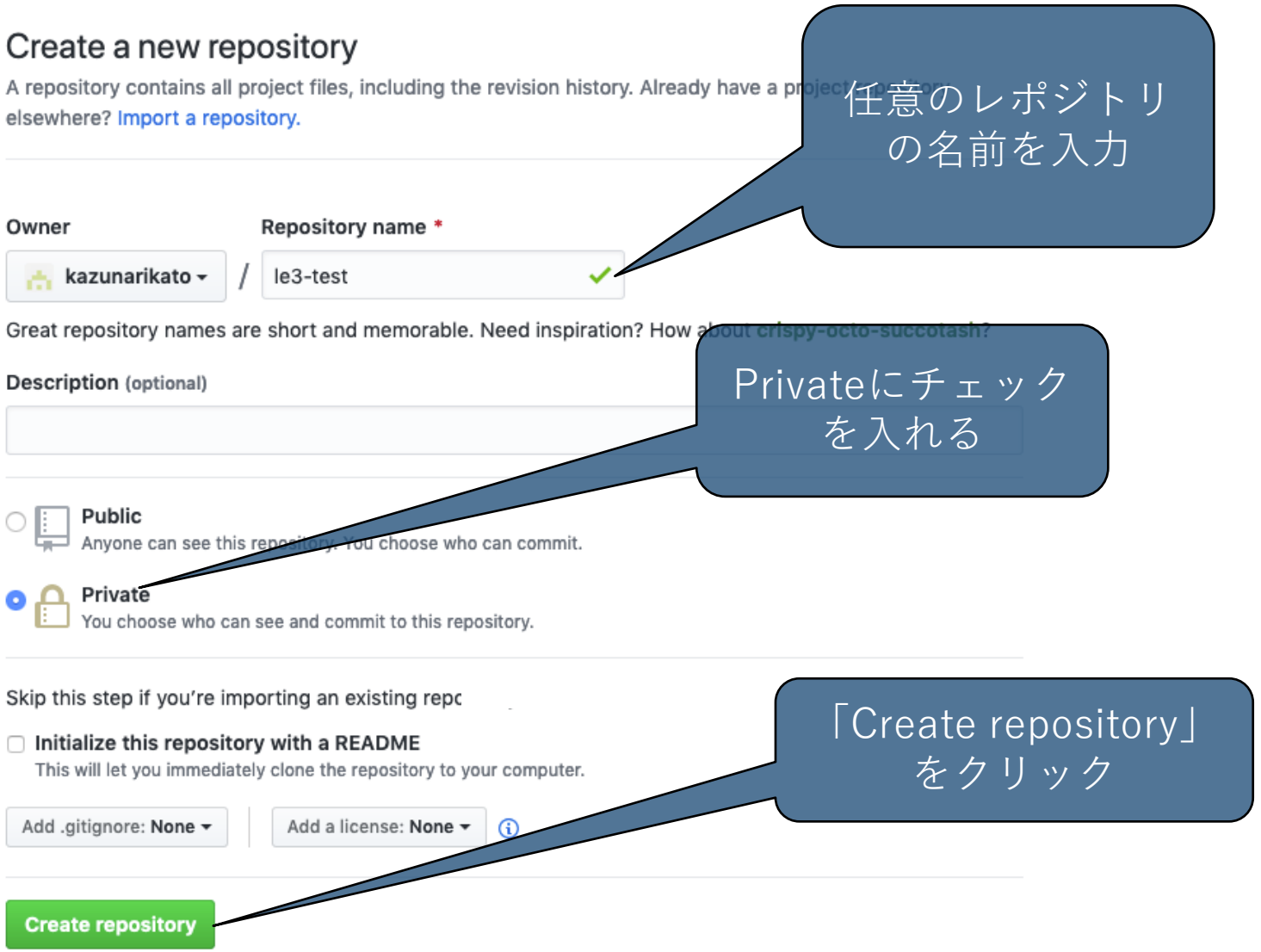
リモートリポジトリの作成



【リポジトリ名に関しての注意】

- 導入課題では、2026-intro-<アカウント名>
  - 例) **2026-intro-nagano28**
- プロセッサ設計演習では、 2026-simple-team<チーム番号>
  - 例) **2026-simple-team1**

リモートリポジトリの作成



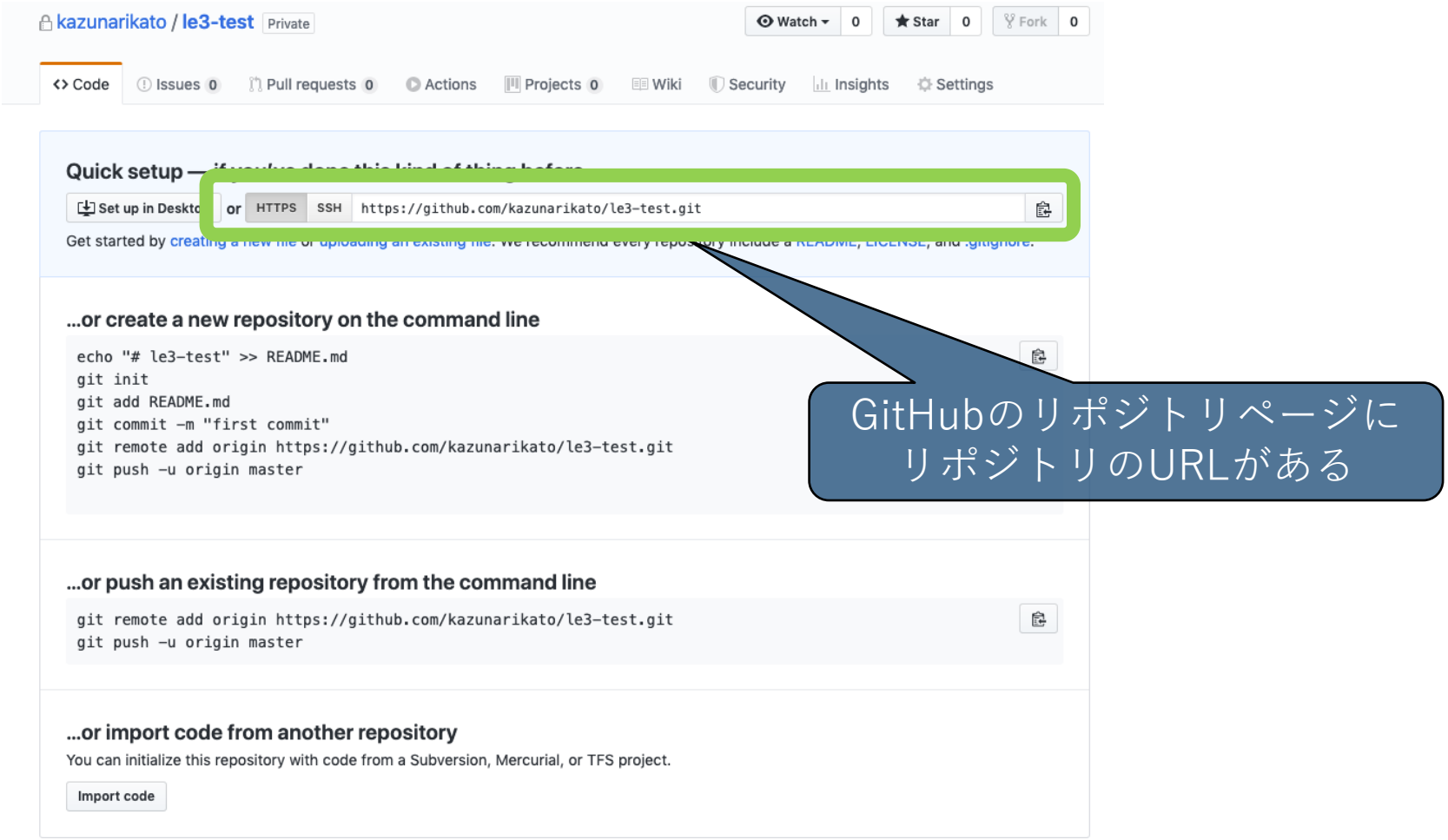
リモートリポジトリの作成

リモートリポジトリのアドレス登録

git remote add <エイリアス> <URL>

<URL>を<エイリアス>という名前でリモートリポジトリとして登録する。

リモートリポジトリのアドレス登録



リモートリポジトリのアドレス登録

```
$ git remote add origin <リモートリポジトリのURL>
```

- 一般的にエイリアス名には “origin”という名前をつける。
- リモートリポジトリとしてURLをoriginというエイリアス名で設定する。

リモートリポジトリへのプッシュ

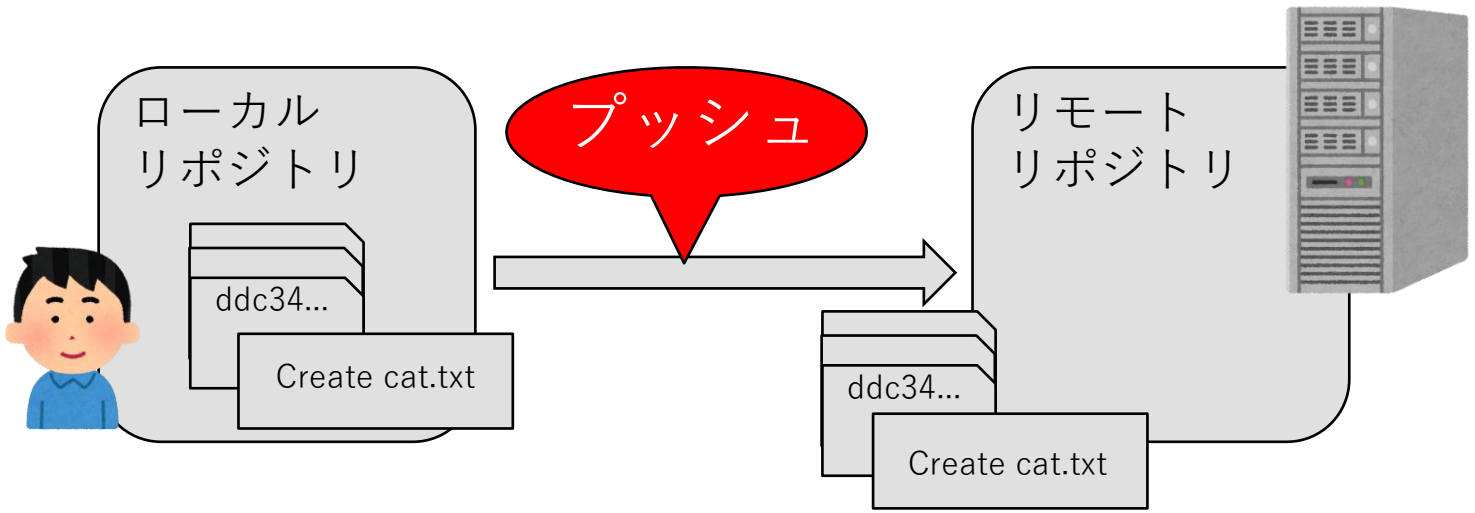
git push <エイリアス> <ブランチ>

<エイリアス>のリモートリポジトリにローカルの<ブランチ>をpushする。

リモートリポジトリへのプッシュ

```
$ git push origin main
```

- originのリモートリポジトリにmainブランチをプッシュする。
  - エイリアスをしていなければ、URLを打ち込むことになる。



リモートリポジトリからのpull

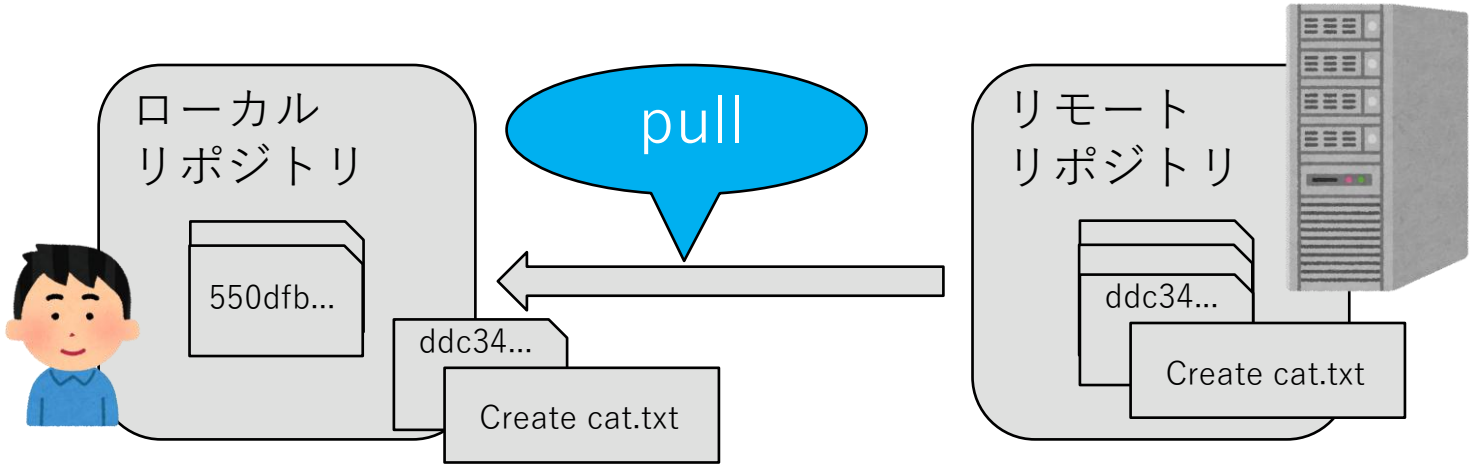
git pull <エイリアス> <ブランチ>

<エイリアス>のリモートリポジトリの内容でローカルの<ブランチ>を更新する。

ローカルリポジトリを更新

```
$ git pull origin main
```

- リモートリポジトリの開発がローカルリポジトリよりも進んでいるとき、リモートリポジトリの内容でローカルリポジトリを更新する。



リモートリポジトリからのclone

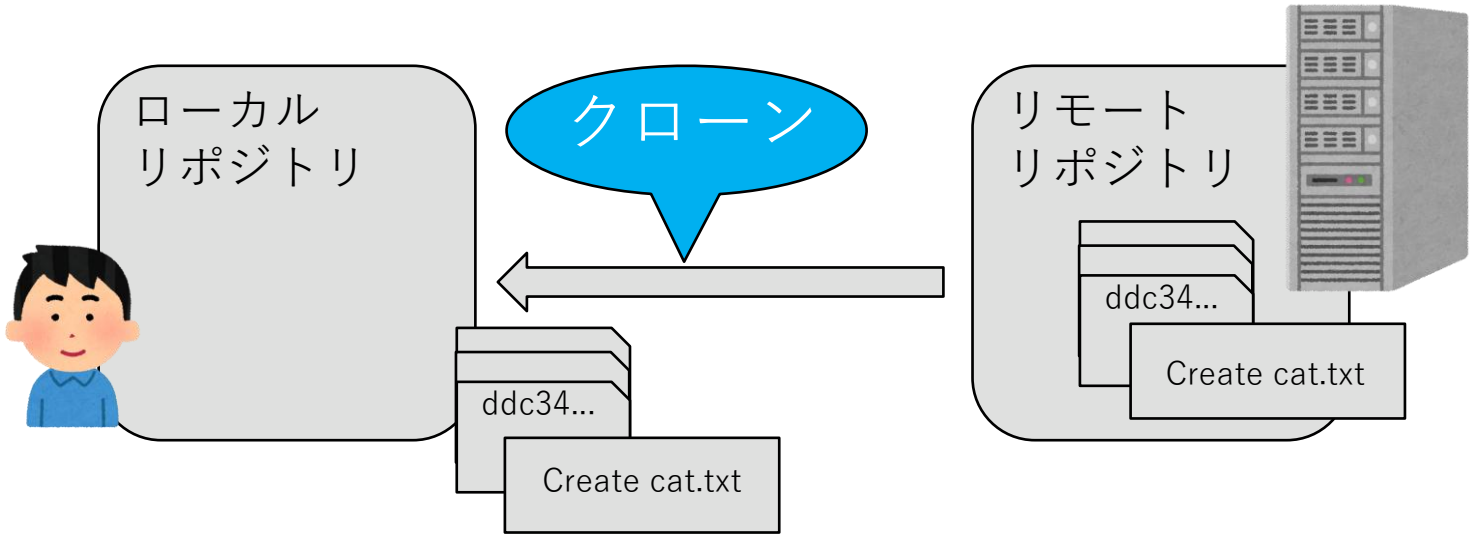
```
git clone <URL>
```

<URL>のリモートリポジトリの内容をローカルにコピーする。

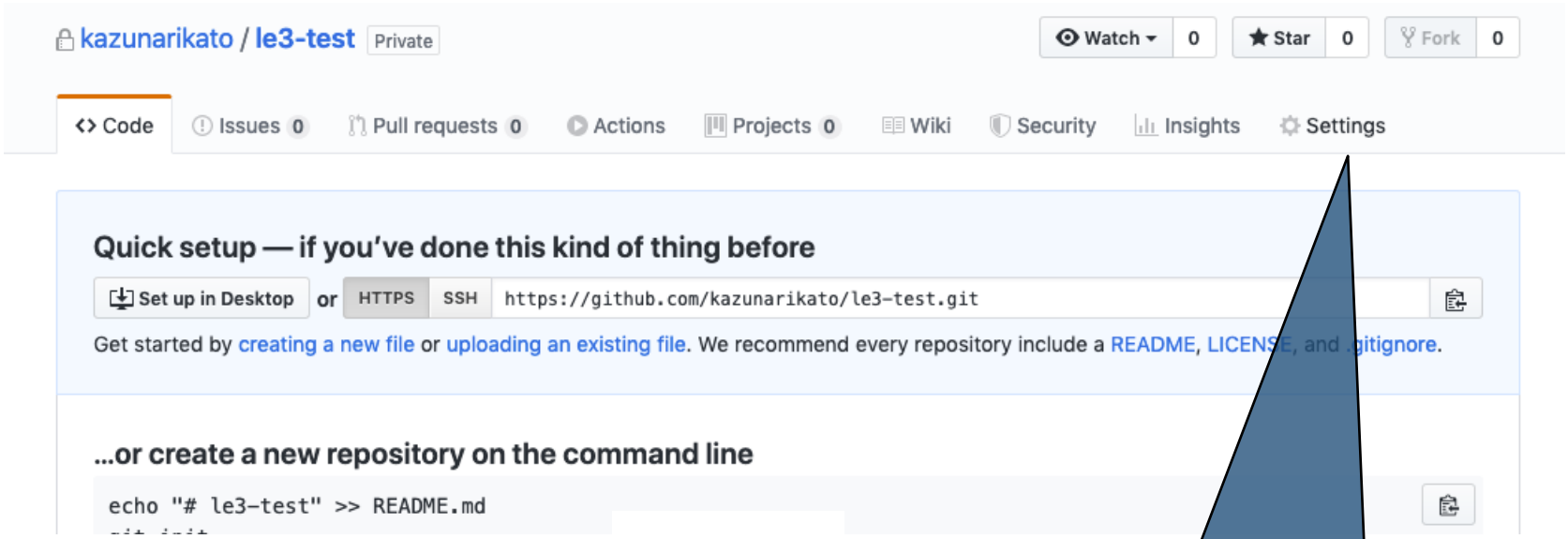
リモートリポジトリからのclone

```
$ git clone <URL>
```

- <URL>で指定したリモートリポジトリをcloneする。

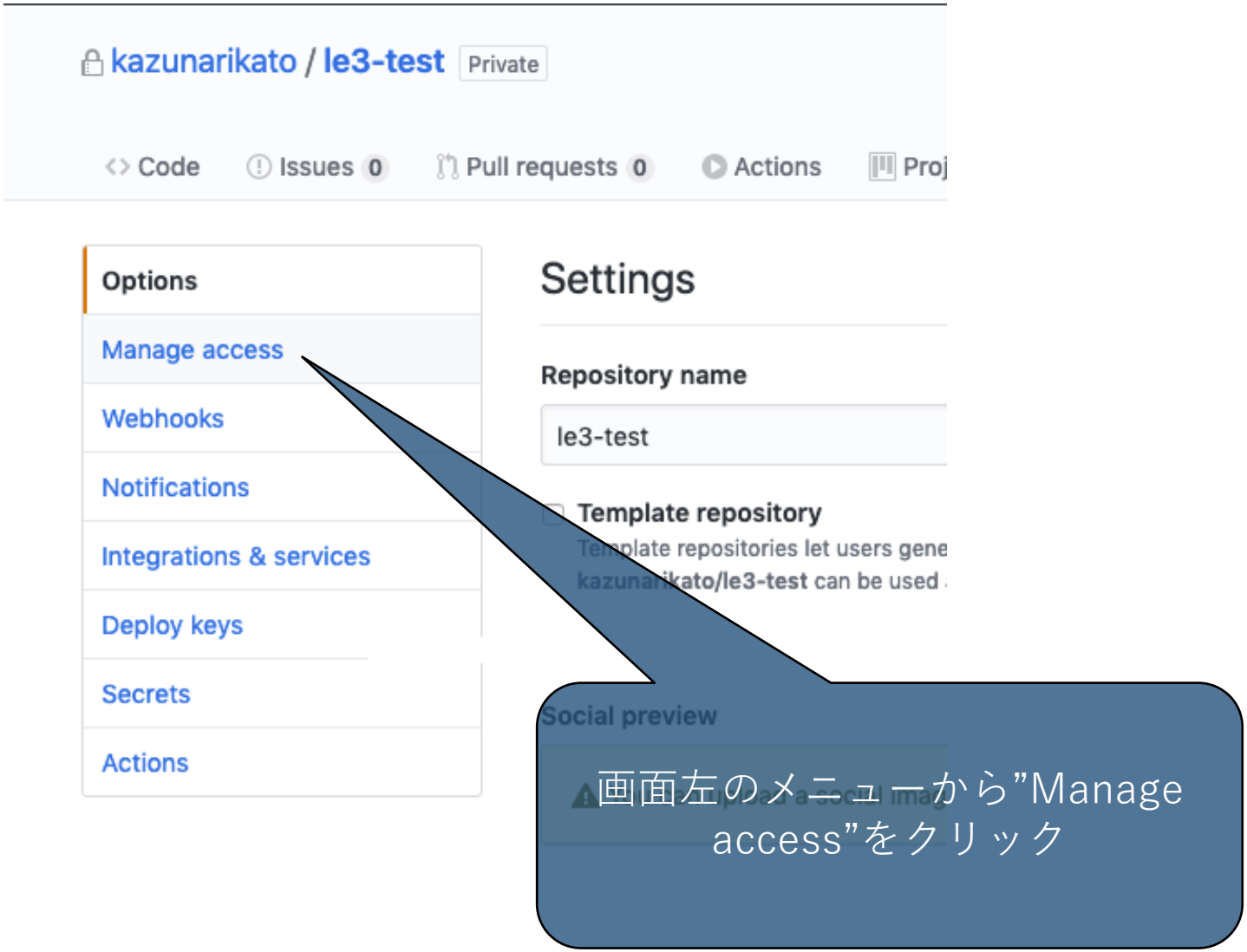


collaboratorの登録

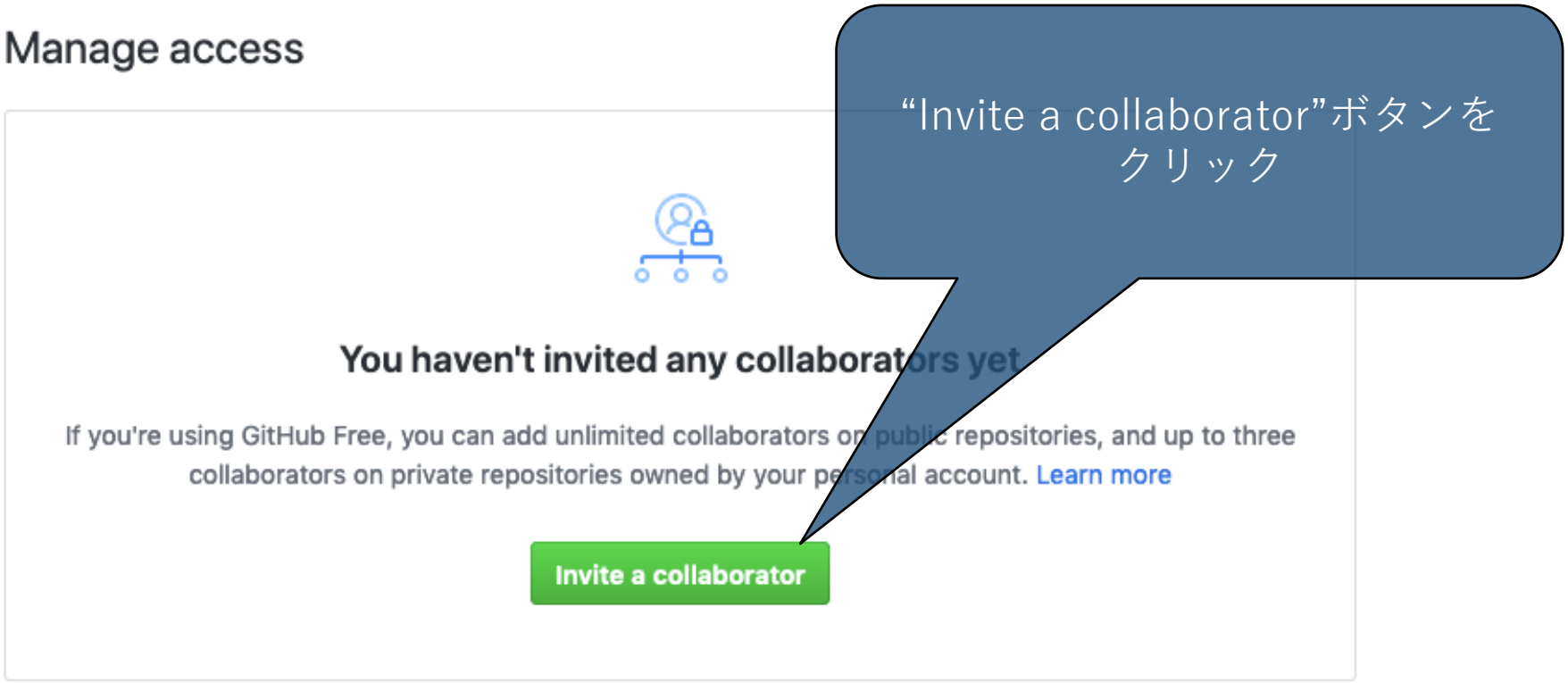


リポジトリのページで
“Setting”をクリック

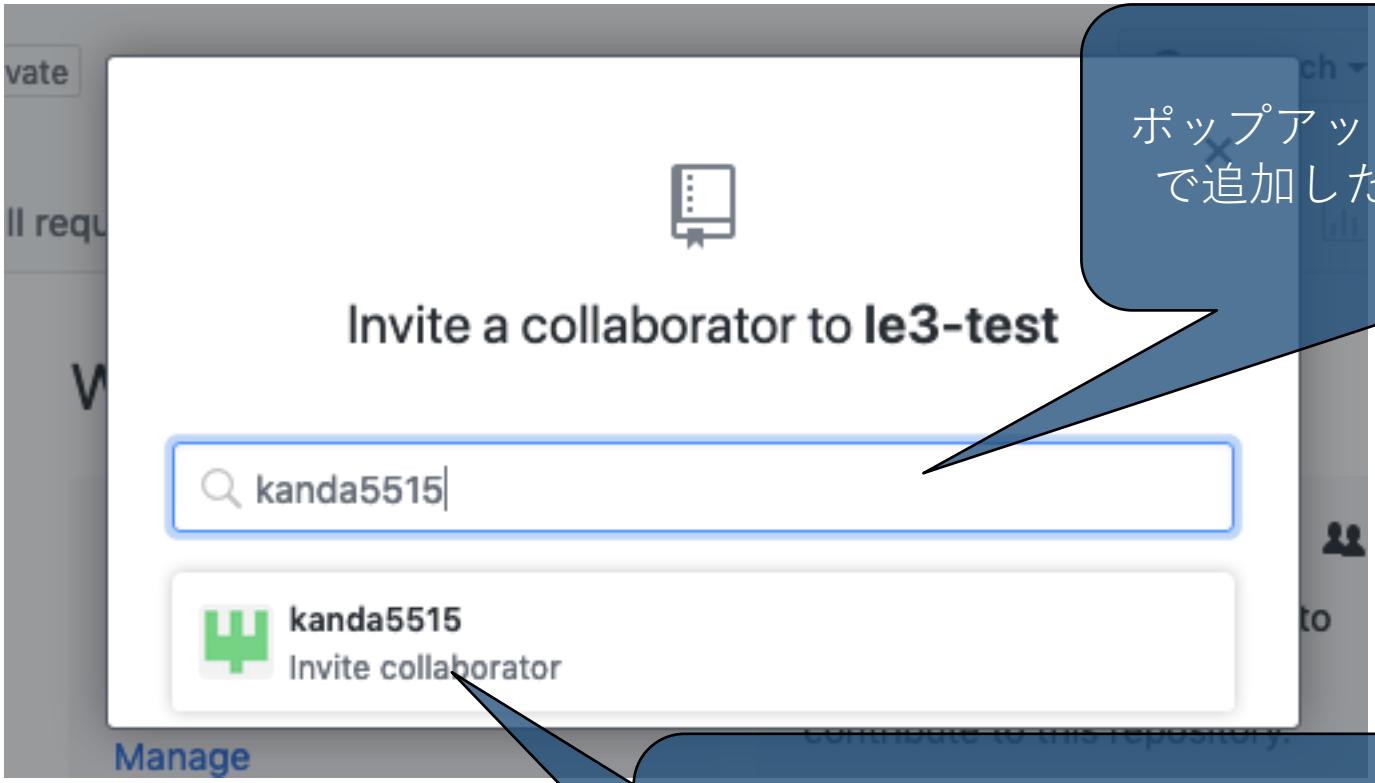
collaboratorの登録



collaboratorの登録



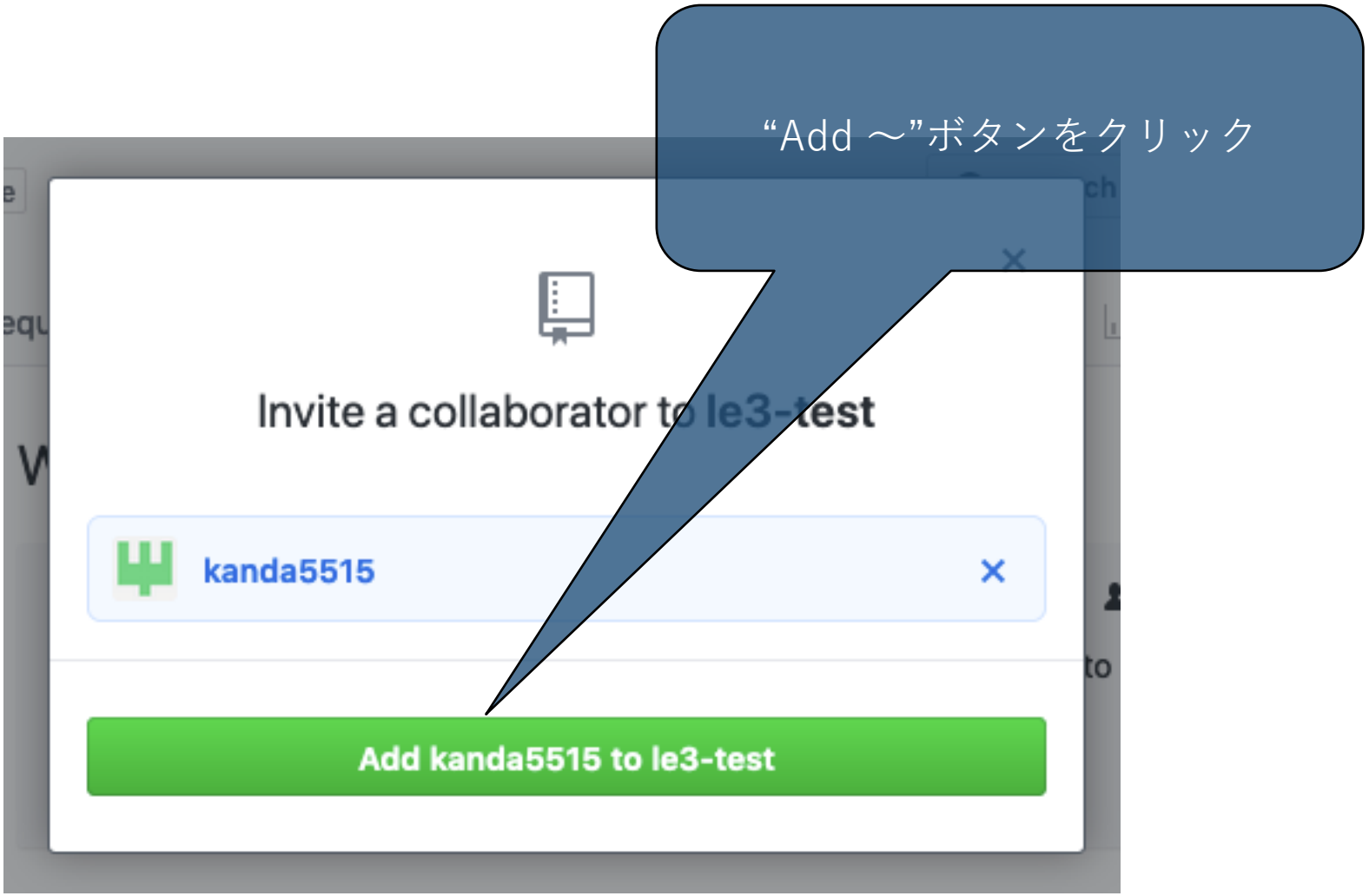
collaboratorの登録



ポップアップウィンドウの検索窓で追加したいユーザー名を検索

検索されたユーザーをクリック

collaboratorの登録



collaboratorの登録

コミットにtagをつける

git tag -a <タグ名> -m <コメント>

最新のコミットに対して、tagをつける。

一般的にタグ名はセマンティックバージョニングに準じてつけるが、本実験では課題ごとに指定されたタグ名をつける。

-a の後にタグ名、-m の後にコメントを記載

過去のコミットにタグをつけたい場合は、<コメント>の後に、コミットIDを書く。

tagの確認

git tag

つけたタグの一覧を確認できる。

git show <タグ名>

指定したタグ名のコミットを確認できる。

tagのpush

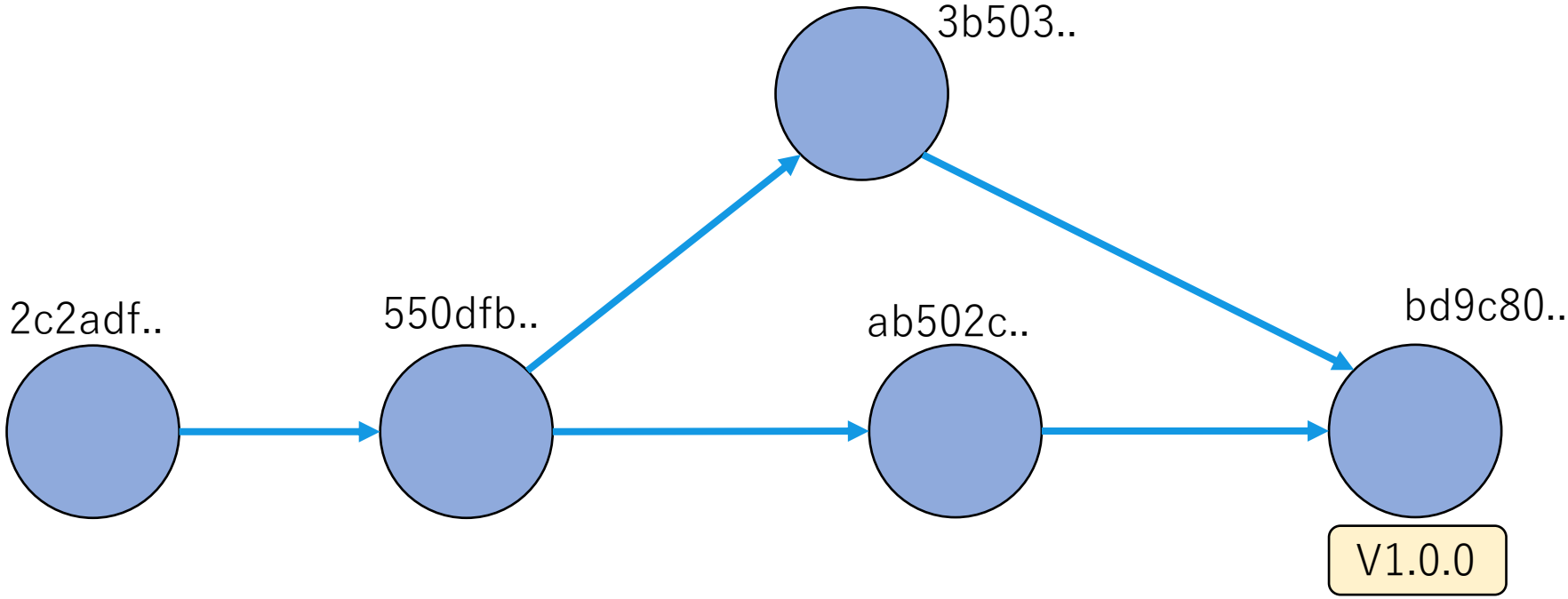
git push <エイリアス> --tag

git pushに--tagオプションをつけると、タグの情報がリモートリポジトリに反映される。
普通にpushしただけでは、リモートリポジトリにtagは反映されない。（逆にこのコマンドだけではブランチへのpushができない）。
必ず “git push <エイリアス> <ブランチ>”でブランチへのpushも完了しておくこと。

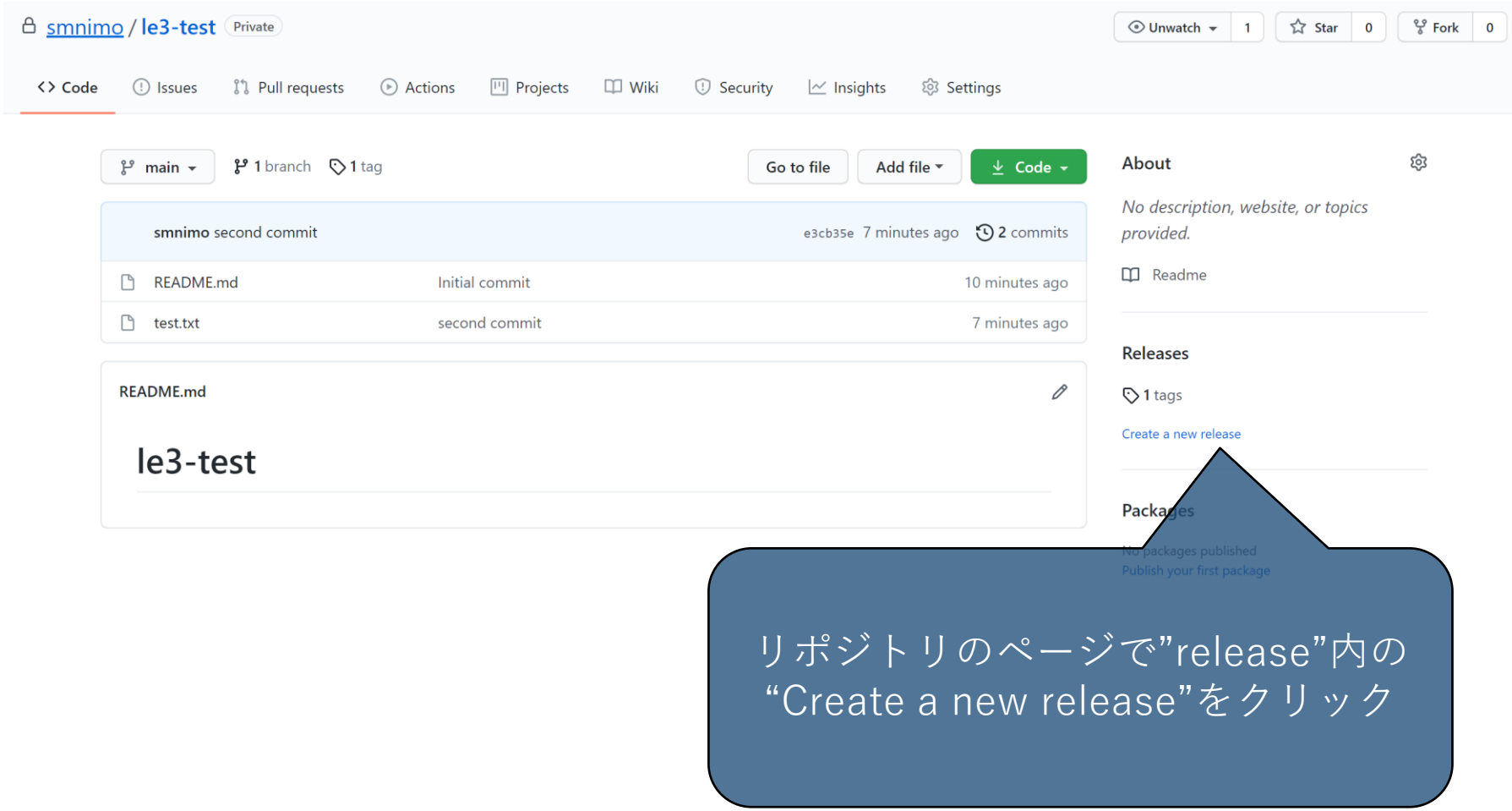
コミットにtagをつける

```
git tag -a v1.0.0 -m ‘add v1.0.0’
git push origin --tag
```

*必ず “git push <エイリアス> <ブランチ>”でブランチへのpushも完了しておくこと。



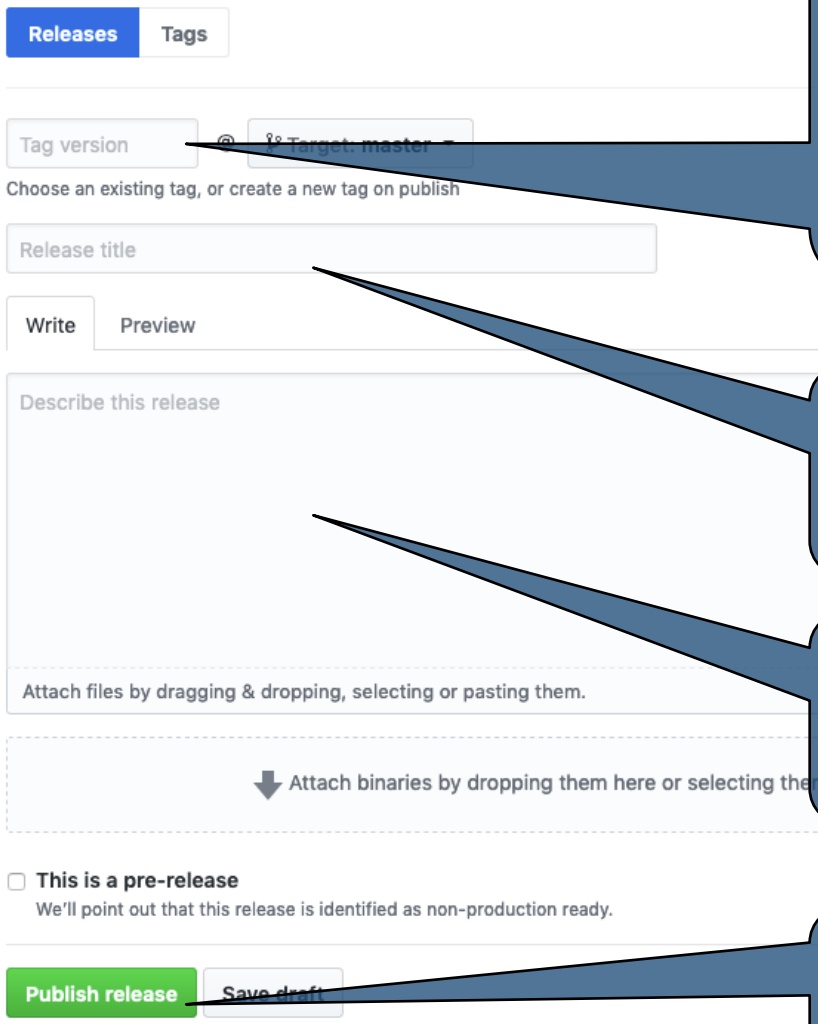
releaseの作成


smnimo / le3-test
Private
Unwatch 1
Star 0
Fork 0
Code
Issues
Pull requests
Actions
Projects
Wiki
Security
Insights
Settings
main
1 branch
1 tag
Go to file
Add file
Code
smnimo second commit
e3cb35e 7 minutes ago
2 commits
README.md
Initial commit
10 minutes ago
test.txt
second commit
7 minutes ago
README.md
le3-test
About
No description, website, or topics provided.
Readme
Releases
1 tags
Create a new release
Packages
No packages published
Publish your first package
リポジトリのページで”release”内の
“Create a new release”をクリック

Releaseの作成


Releases
Tags
Draft a new release
7 minutes ago
v1.0.0
edcbeef
zip
tar.gz
作成したタグが表示されている。
“Draft a new release”
ボタンをクリック

Releaseの作成



Releaseのバージョン番号を入力。バージョンはGit tagでつけたタグ名から選択。タグ名は課題ごとに指定されている。

リリースのタイトルを入力

リリースの説明を入力

“Publish release”をクリック

Releaseの作成


Releases
Tags
Edit release
Delete
Latest release
v1.0.0
edcbeef
Compare
v1.0.0
kazunarikato released this now
the first version
Assets 2
Source code (zip)
Source code (tar.gz)
Releaseが作成される
ファイルがダウンロードできる形で提供される

⑤GitHubに公開鍵を登録

SSH用の公開鍵の登録

SSHで使うRSAキーの公開鍵をGitHub登録することでターミナルで「git clone git@github.com:kuis-isle3hw/.....」

詳しくは
https://docs.github.com/en/authentication/connecting-to-github-with-ssh　を参考

ここではざっくり過程を見せる。

SSH用の公開鍵の登録

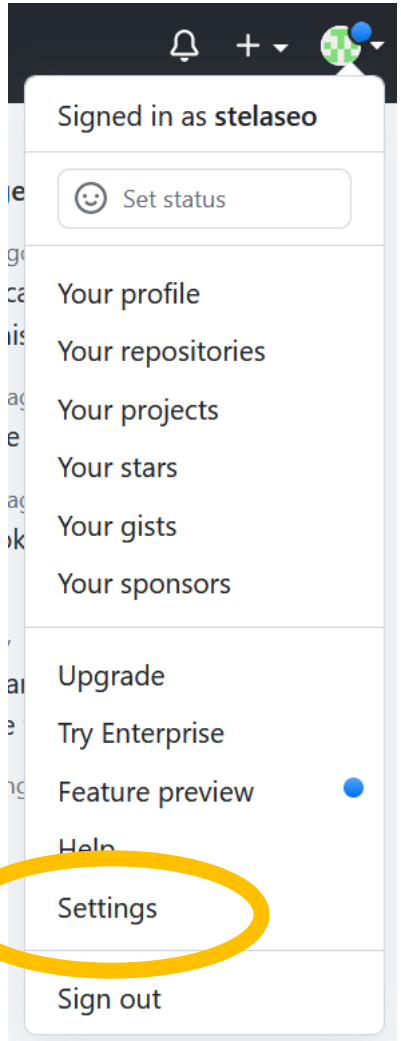
ターミナルで「ssh-keygen」ツールを使ってキーを作成

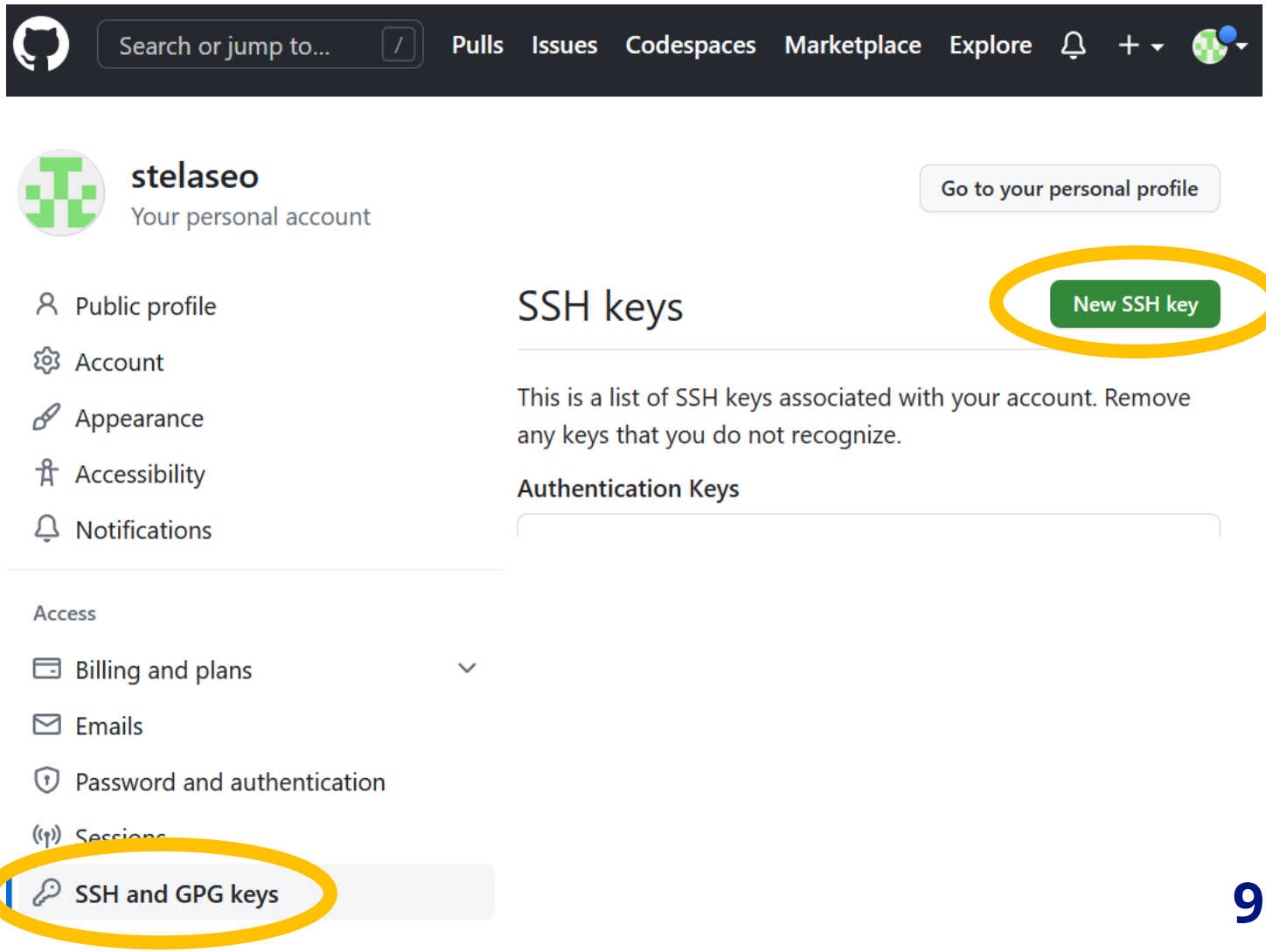
```
stela@jupiter:~$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/home/stela/.ssh/id_rsa):
Created directory '/home/stela/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/stela/.ssh/id_rsa
Your public key has been saved in /home/stela/.ssh/id_rsa.pub
The key fingerprint is:
SHA256:EtwevEOyzifh82x7F4+/A6MIwX4Fr0NTgDXtdl5Ztnc stela@jupiter
The key's randomart image is:
+---[RSA 3072]----+
|       o+o       |
|     ..o..o     o|
|     .+ ==     +.|
|      o*oo= . o.E|
|     .+oS= o .  o|
|     +oo+.  =    |
|      *o.o . *   |
|       *o o o o  |
|        .+o . .oo |
+----[SHA256]-----+
stela@jupiter:~$
```

SSH用の公開鍵の登録

公開鍵をGitHubに登録する

Settings → SSH and GPG keys → New SSH Key





SSH用の公開鍵の登録

公開鍵をGitHubに登録する

```
stela@jupiter:~$ cat ~/.ssh/id_rsa.pub
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABgQCoA9WqejOcpRrdWWI58cJoD6EXy
dOm70k5nQPx36FGougS1xj2h2dRMvZw24D7hgXJ8enwWZ5FJSpJ1Kf2jE5VrVX4J2
OzLNPEfVgVtaq3kM5UNguFzVKyYhithMyuKZiWDuAVinMSm4xk9Am1RglsHVVHNGG
P/BO3L5vSpuE/AzwLApJ3s[illegible]OGE16wb/pxtEya7/vU2R1fKugsiT2KqXsi2qB4zUr
Cf/+EJSIKUh3Z+/[illegible]
n4CAF54ok9ZpBkDg
eLtroouPUsEGZXMs
9dYxUXqIGAqA7c4i
IM/YAs8RGZAlmSPz
```

id_rsa.pubの中身を
コピー＆ペースト

SSH key / Add new

Title

Key type

Authentication Key

Key

Begins with 'ssh-rsa', 'ecdsa-sha2-nistp[illegible]', 'ecdsa-sha2-nistp384', 'ecdsa-sha2-nistp521', 'ssh-ed25519', 'sk-ecdsa-sha2-nistp256@openssh.com', or 'sk-ssh-ed25519@openssh.com'

Add SSH key

SSH用の公開鍵の登録

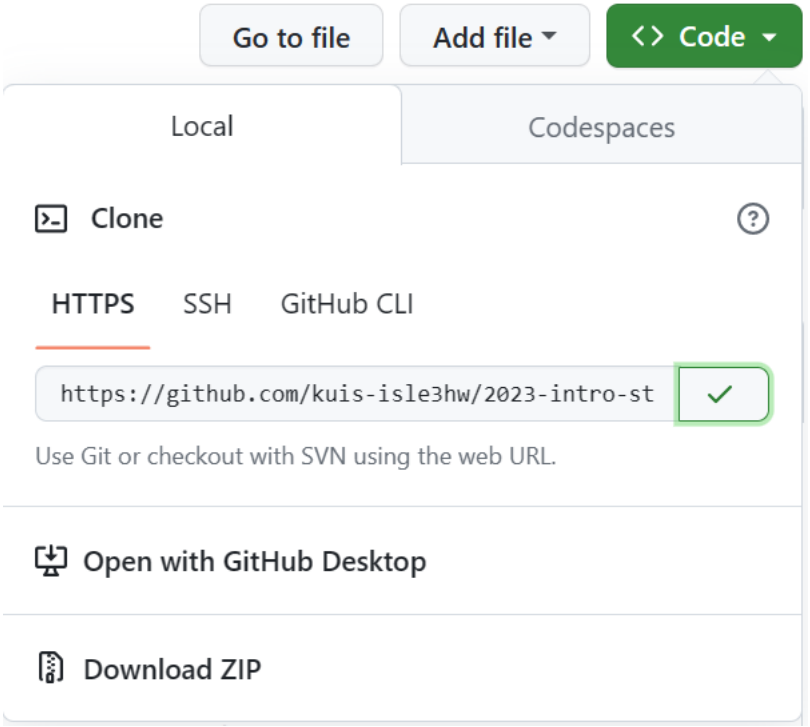
ターミナルでgit repositoryをclone

```
stela@jupiter:~$ git clone git@github.com:kuis-isle3hw/2023-intro-stelaseo.git
Cloning into '2023-intro-stelaseo'...
Enter passphrase for key '/home/stela/.ssh/id_rsa':
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 1 (delta 0), pack-reused 0
Receiving objects: 100% (3/3), done.
stela@jupiter:~$
```

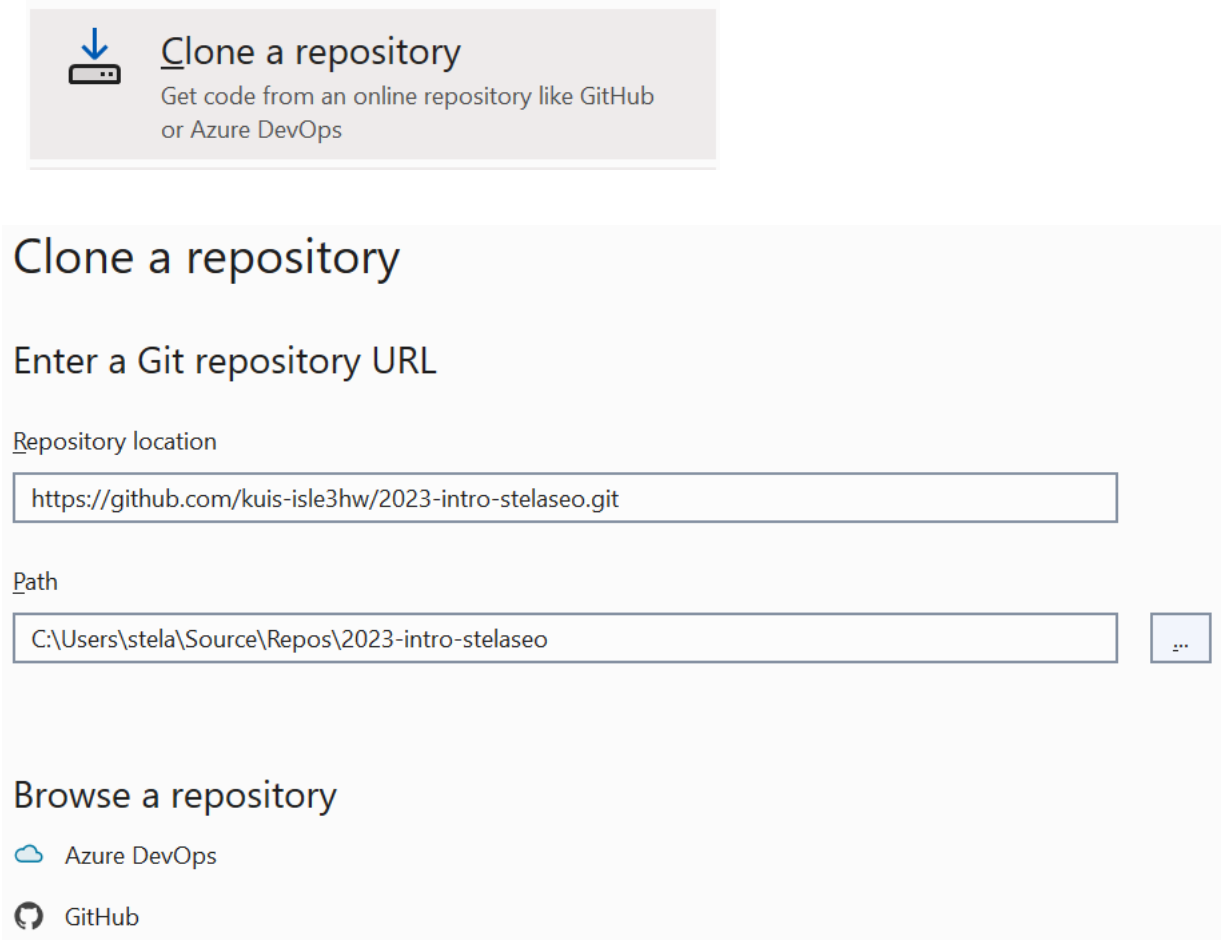
⑤GitHubをVS2022内でアクセス

公開鍵をGitHub登録せずIDE内で使いたい場合

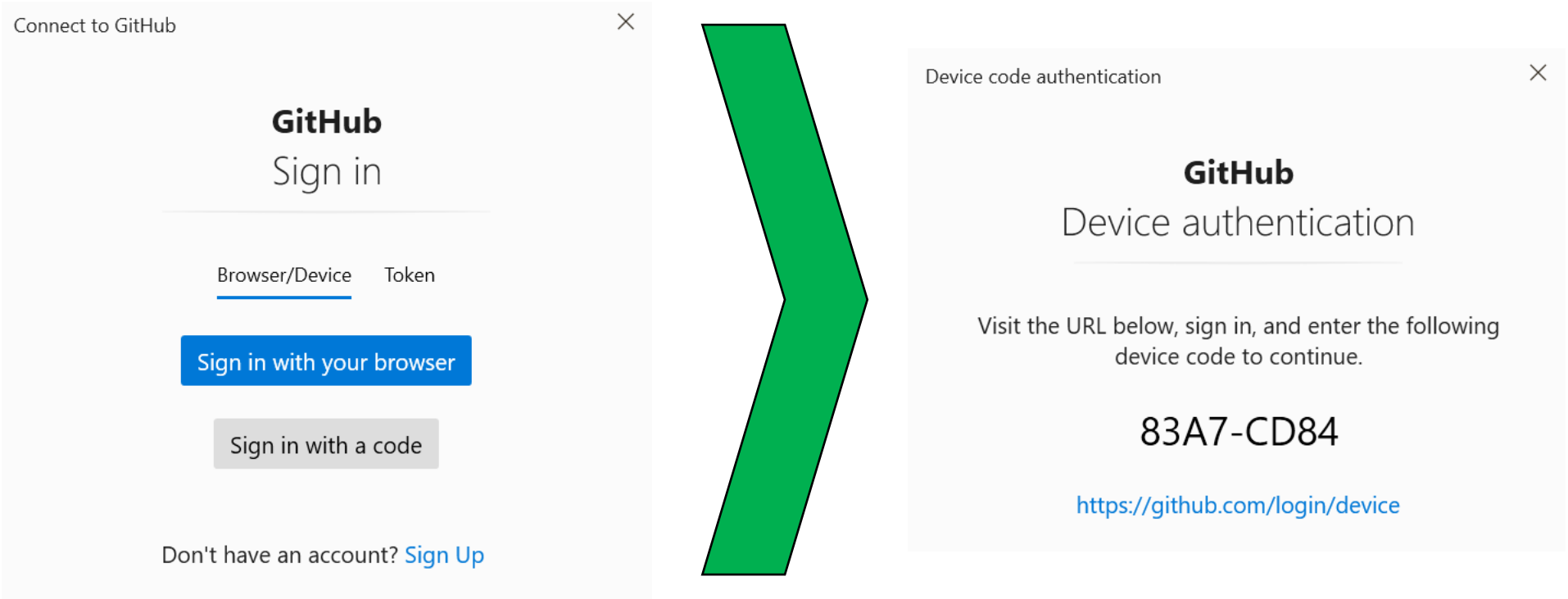
GitHubからrepositoryのHTTPSをコピー



VSでcloneを始める



Popupから「Sign in with a code」を選択

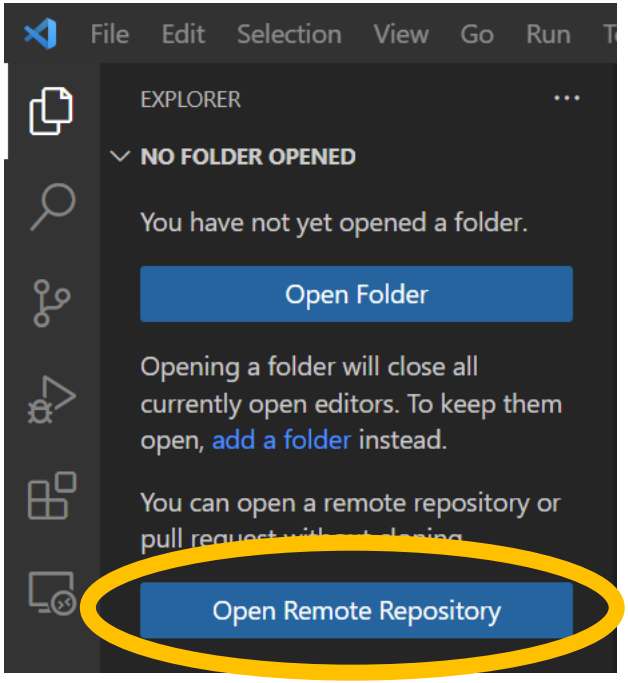


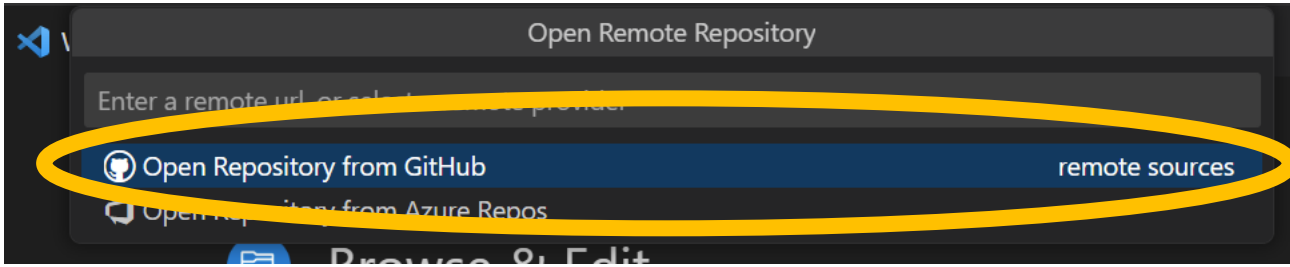
https://github.com/login/device　のページにコードを入力する

⑤GitHubをVS Code内でアクセス

Open Remote Repositoryを選択

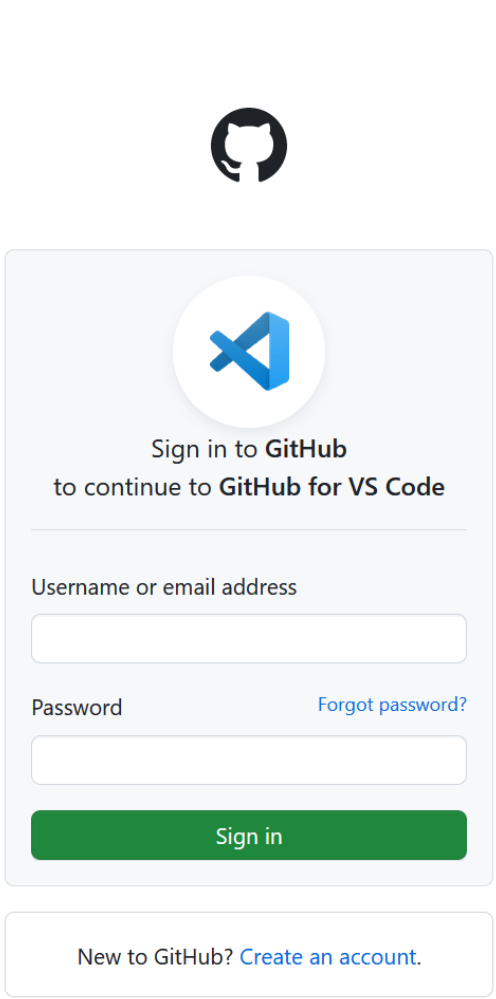
Open Repository from GitHubを選択

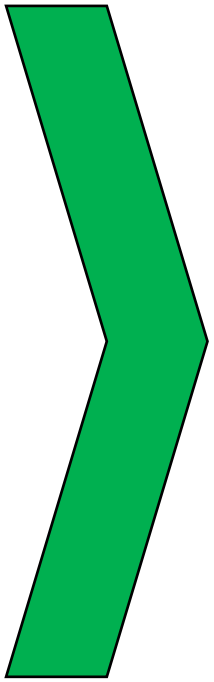


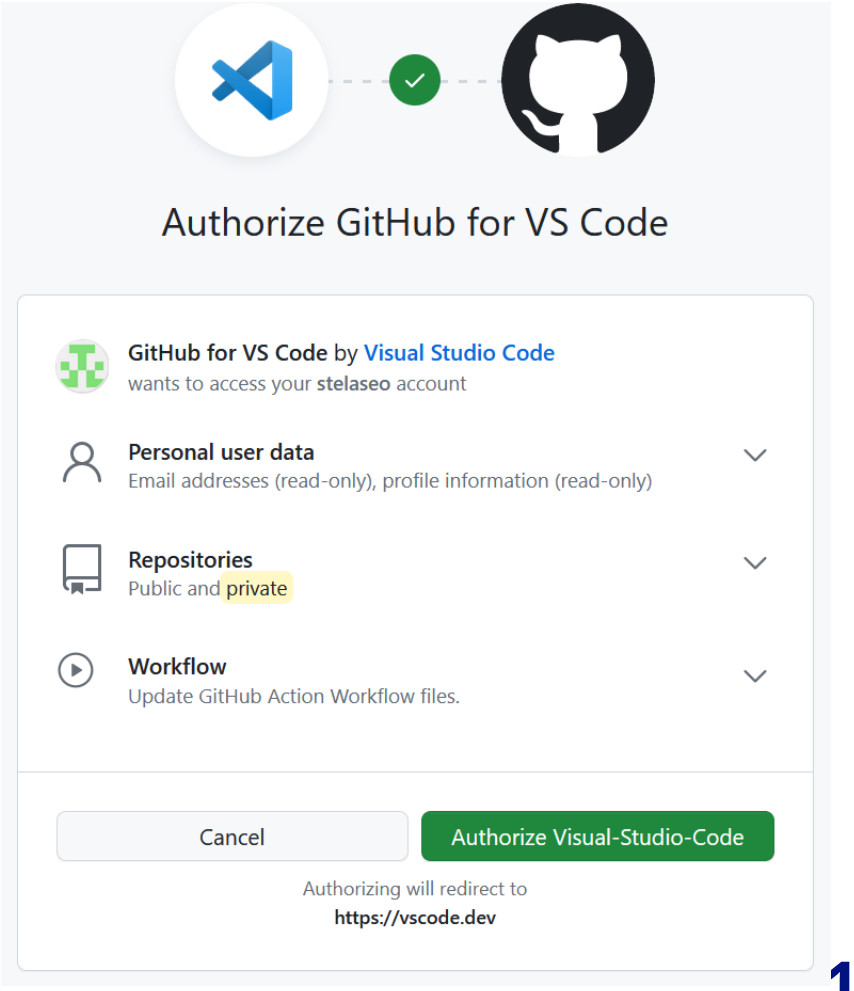


ログイン後

Authorize Visual-Studio-Codeを押す







⑥GitHub Classroomの利用

GitHub Classroom

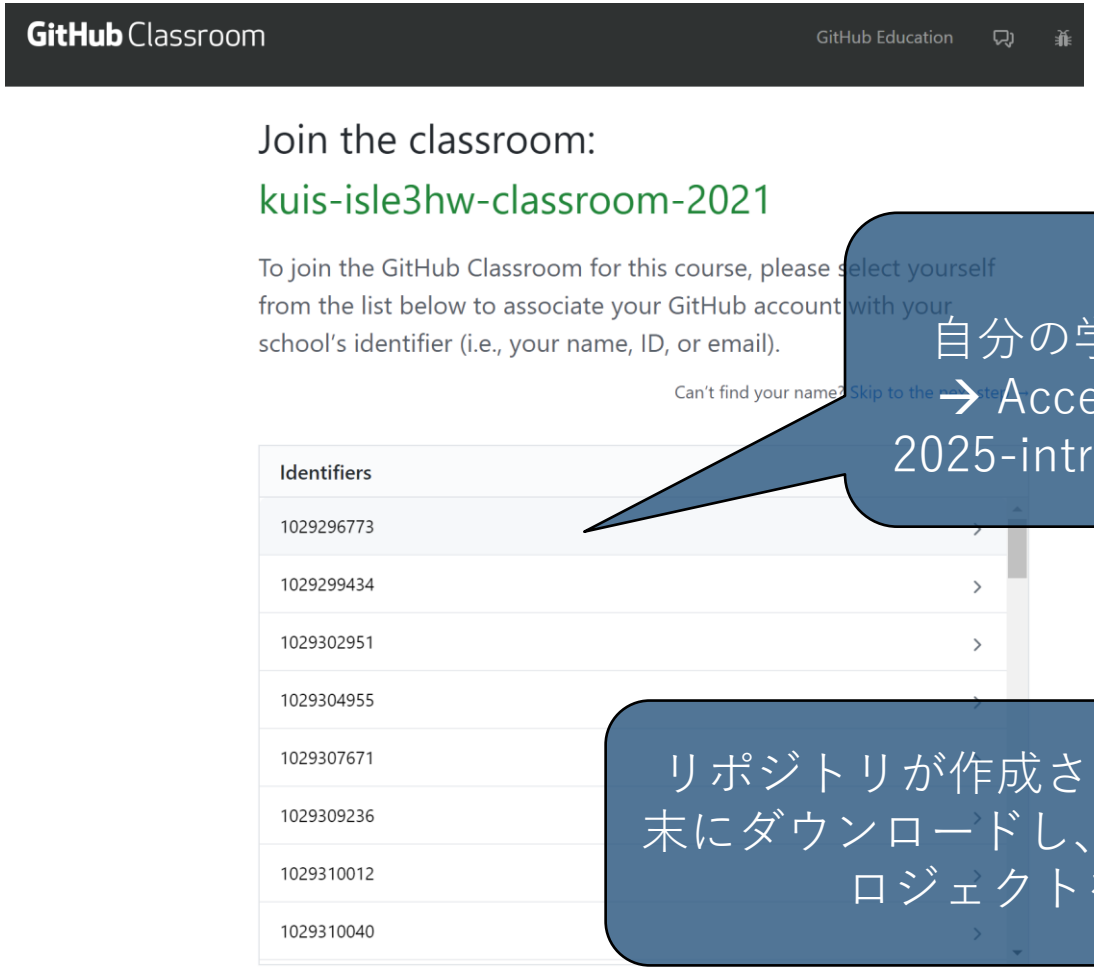
- **GitHub Classroom は GitHub を講義で利用するためのプラットフォーム**
  - 個人用，グループ用の課題の作成が可能
  - 各学生の個人GitHub アカウントで作成したリポジトリとリンクさせることで課題用のリポジトリを一括管理
  - 本実験ではこのGitHub Classroom を利用し，課題は各リポジトリでRelease を作成することで提出

GitHub Classroomの導入

- 招待リンクはPandA内のお知らせを参照
  - リポジトリは２種
    - 導入課題（個人課題）：2025-Intro
    - プロセッサ課題（グループ課題）：2025-Simple
- まずGitHub Classroom と GitHub の連携について authorizationを求められるので承認
  - GitHub アカウントを持っていない人はここで作成を

導入課題

- 導入課題は各学生１人に対して１つのリポジトリ



Identifiers の中から
自分の学生番号を探し出してクリック
→Accept でリポジトリが作成される
2025-intro-XXX（Github アカウント名）

※自分の学生番号が見つからない場合はスタッフまで連絡を

リポジトリが作成されたら、”git clone”で自分端末にダウンロードし、そのフォルダでquartusのプロジェクトを作り課題を進める

プロセッサ課題

- プロセッサ課題は基本的に学生２人に１つのリポジトリ
  - ３人グループもある
- グループはこちらで割り振って連絡しますので後ほどPandAを確認してください.
- １人の学生がチームの作成（連動してリポジトリの作成）
  → もう１人の学生は作成されたチームを探して参加
  - チーム名はグループ分けの番号と対応して「teamXX」（例：team01）としてください.
    - リポジトリは 2025-simple-teamXX が作成されます.

プロセッサ課題

- リポジトリが作成されたら、グループの一人がcloneで自分の端末にダウンロードし、そのフォルダでquartusのプロジェクトを作成し、push。

- 別のグループ員は上記作業後にcloneして、以後それぞれの端末で課題を進める。

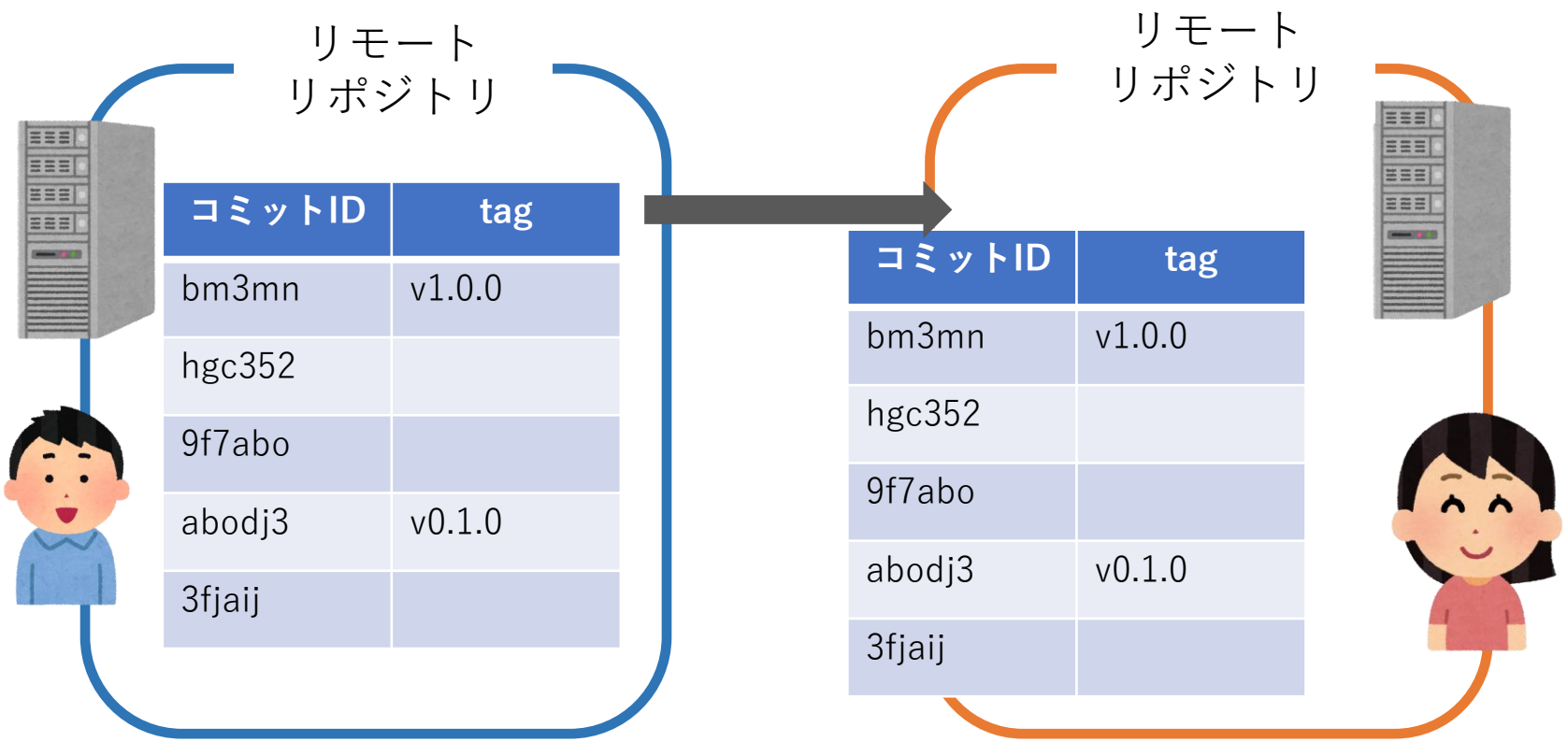
課題の提出

- 導入課題、プロセッサ課題の提出はGithubの**release**を使って行う。
- **release**については本資料のp61とp81～88を参照。

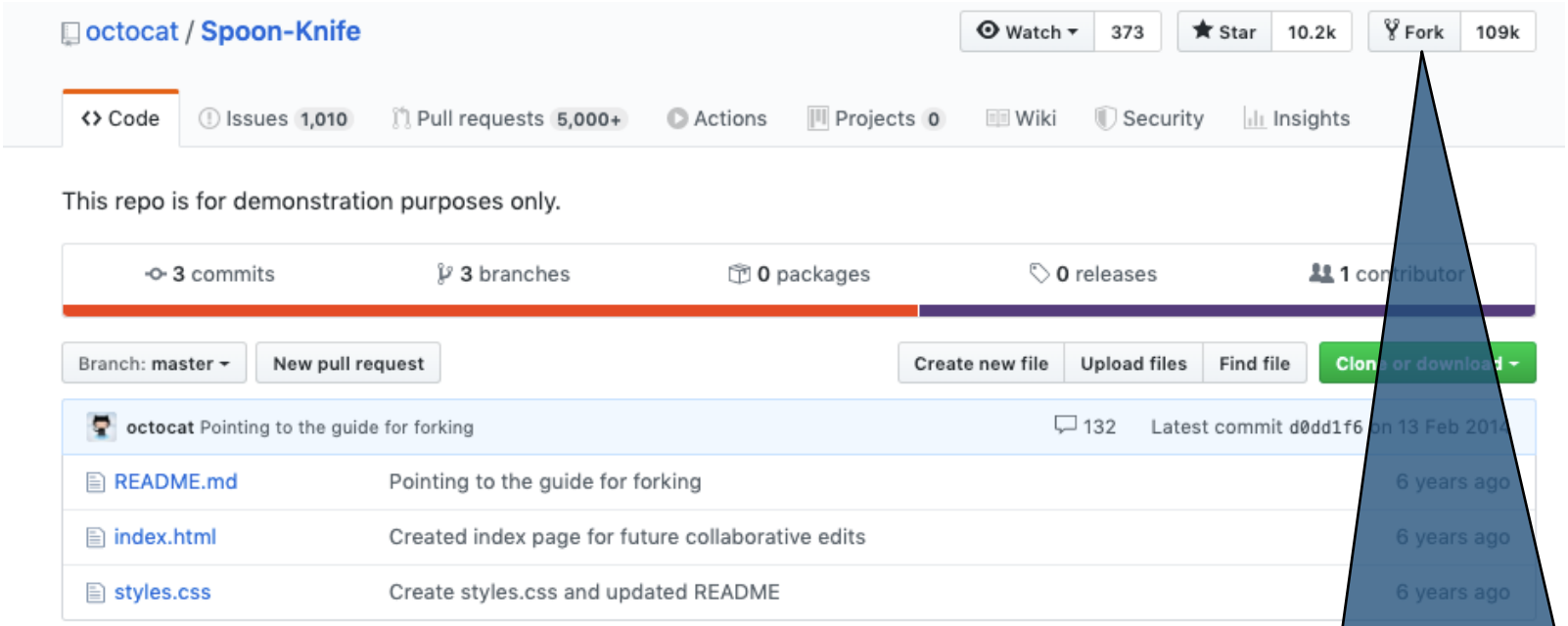
⑥その他のGitHub機能

fork

- 他の人のリモートリポジトリを自分のリモートリポジトリにコピーする。

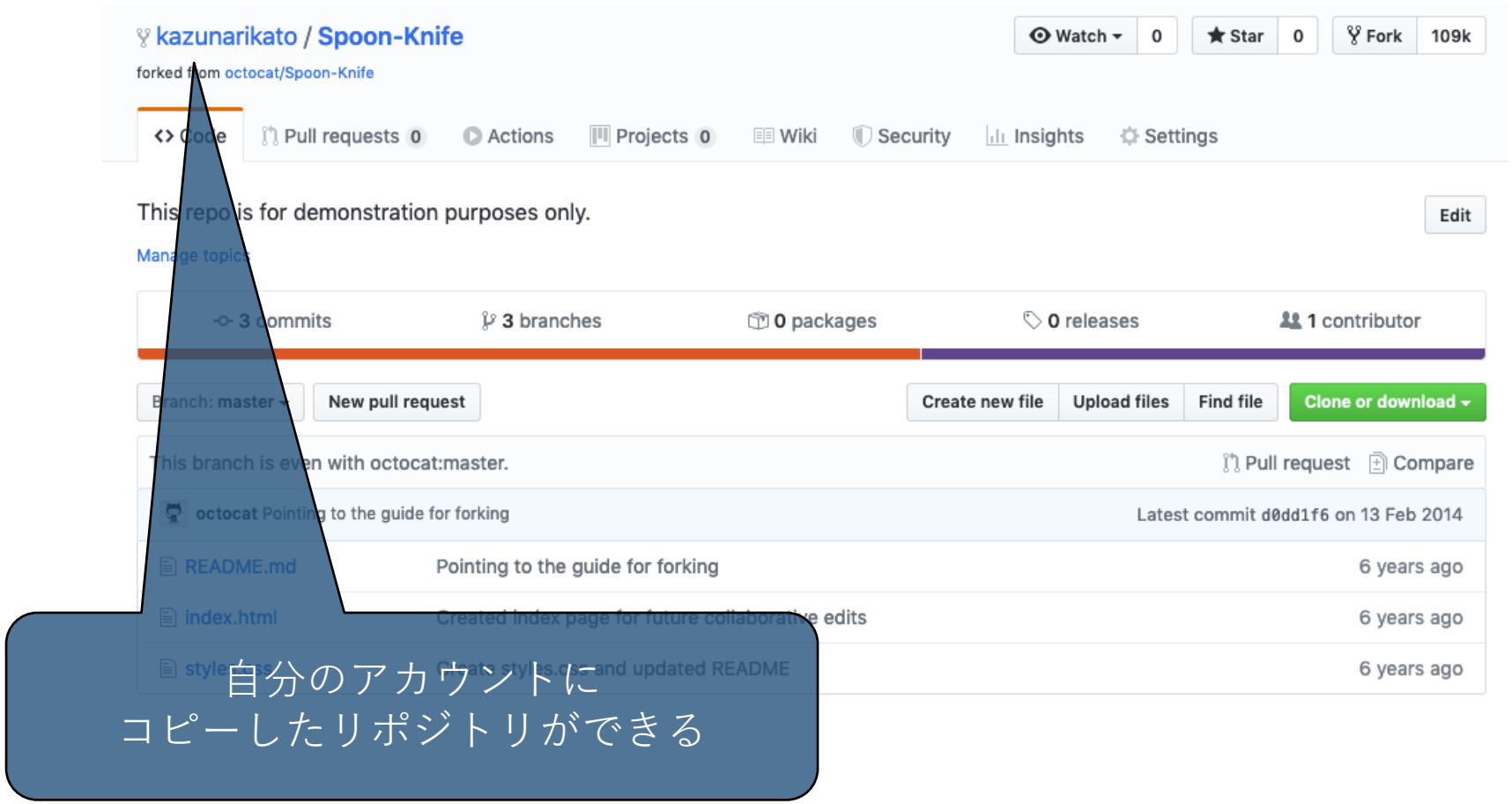


forkの方法



コピーしたいリポジトリページの
リポジトリ名の右側にある
”Fork”をクリック

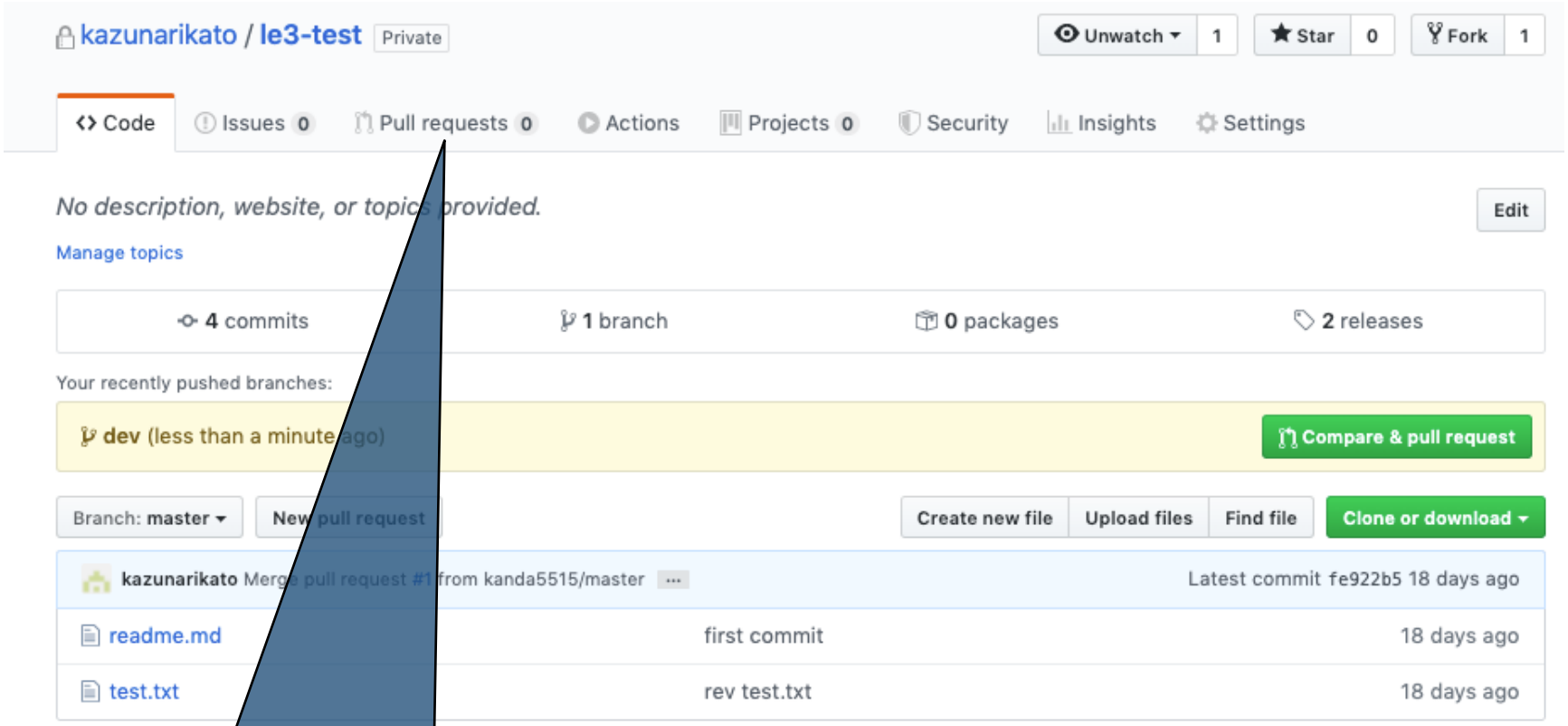
forkの方法


kazunarikato / Spoon-Knife
forked from octocat/Spoon-Knife
Watch 0
Star 0
Fork 109k
Code
Pull requests 0
Actions
Projects 0
Wiki
Security
Insights
Settings
This repo is for demonstration purposes only.
Edit
Manage topics
3 commits
3 branches
0 packages
0 releases
1 contributor
Branch: master
New pull request
Create new file
Upload files
Find file
Clone or download
This branch is even with octocat:master.
Pull request
Compare
octocat Pointing to the guide for forking
Latest commit d0dd1f6 on 13 Feb 2014
README.md
Pointing to the guide for forking
6 years ago
index.html
Created index page for future collaborative edits
6 years ago
6 years ago
自分のアカウントに
コピーしたリポジトリができる

Pull request

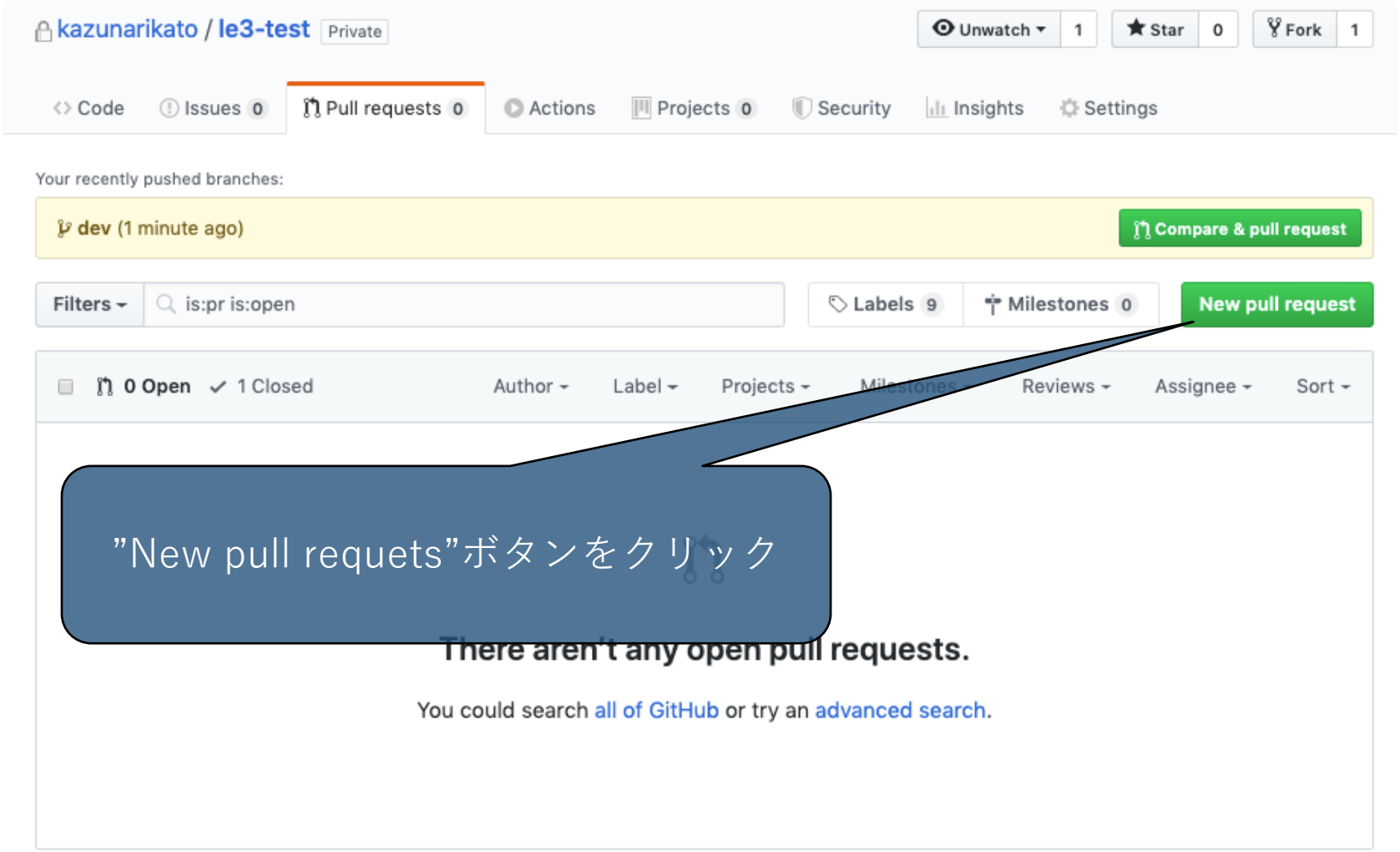
- 新機能を追加し、ファイルをcommitした時、いきなりリモートリポジトリのmainにpushすると、そこにバグが含まれていた時に問題となる。
- Pull requestを使うと、pushする前にローカルリポジトリでの変更内容をリモートリポジトリを共有している他の人に通知することができる。
- 変更内容を確認後、リモートリポジトリ上でmergeすることで、リモートリポジトリが更新される。
- コード・レビューがしやすくなる。

Pull requestの作成



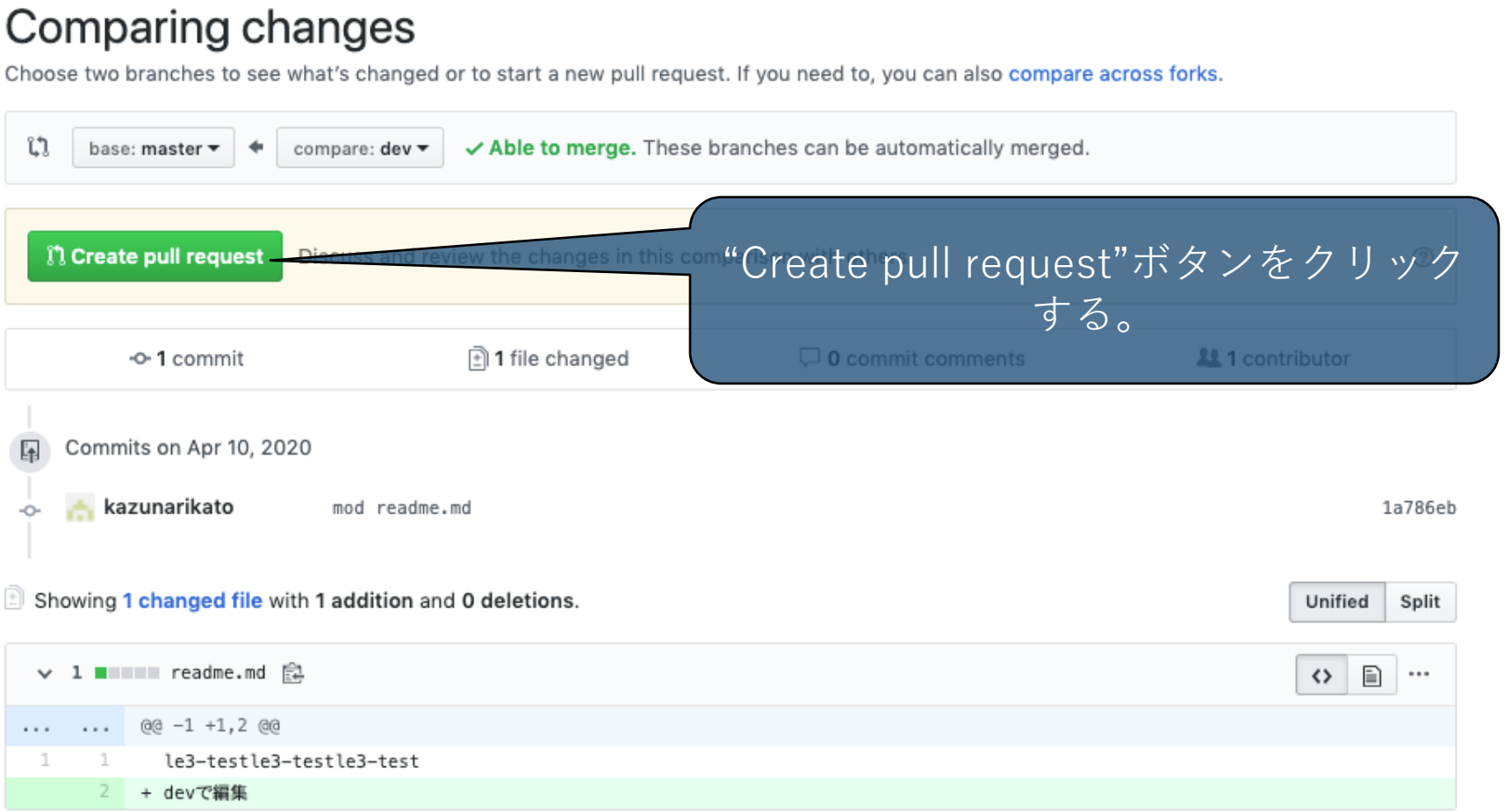
リポジトリページの
”Pull requests”をクリック

Pull requestの作成

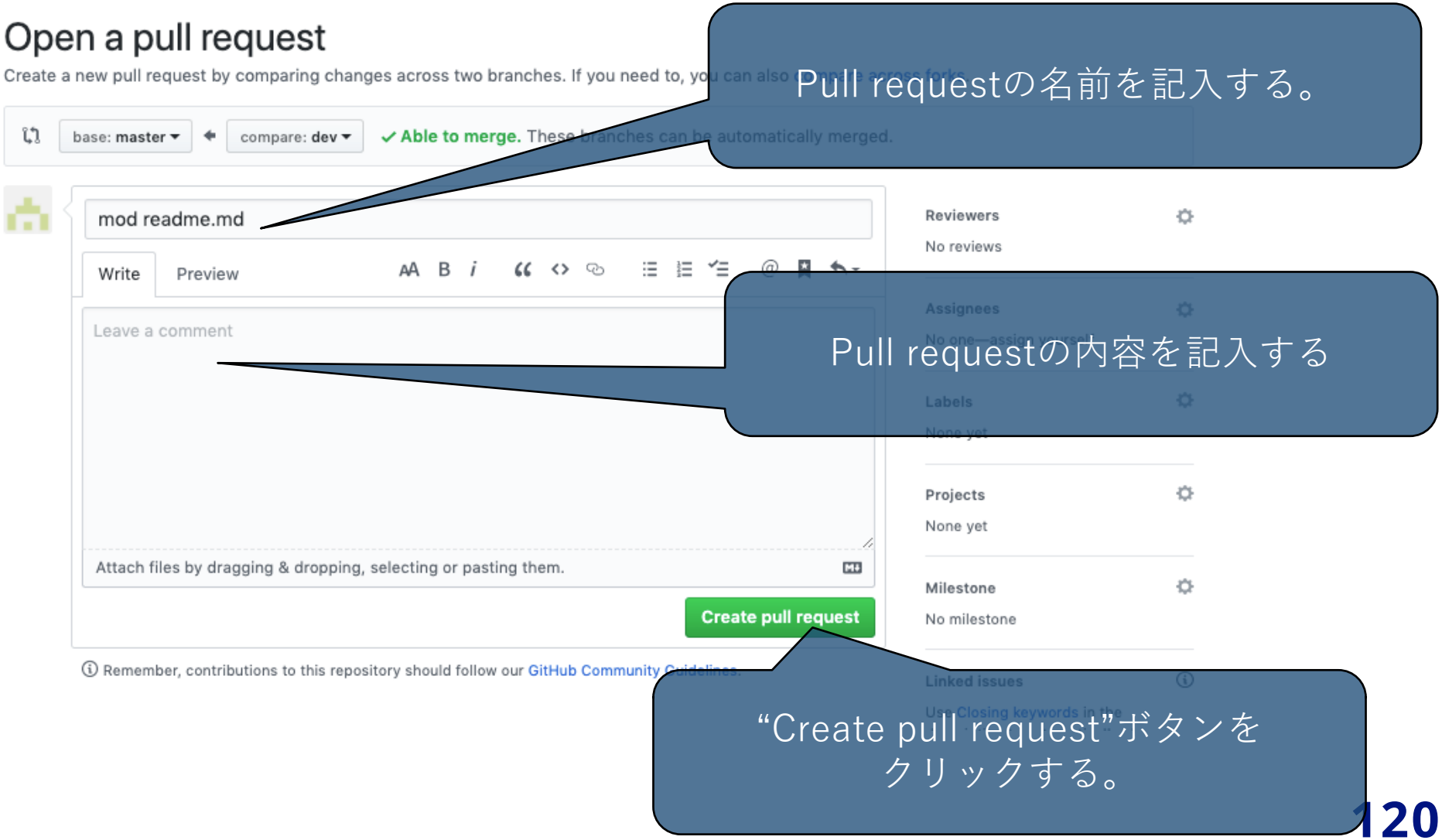


Pull requestの作成

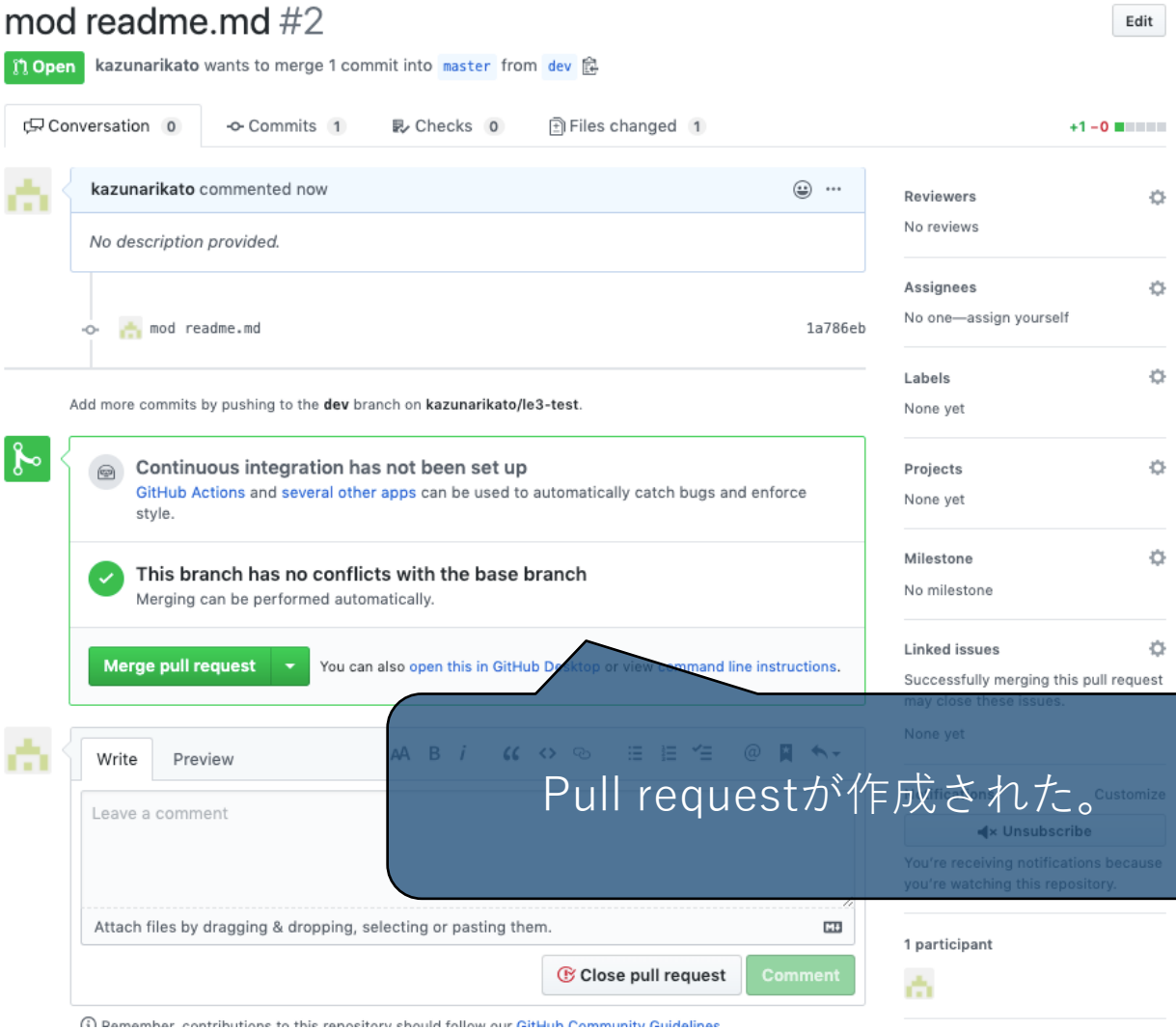
Pull requestの作成



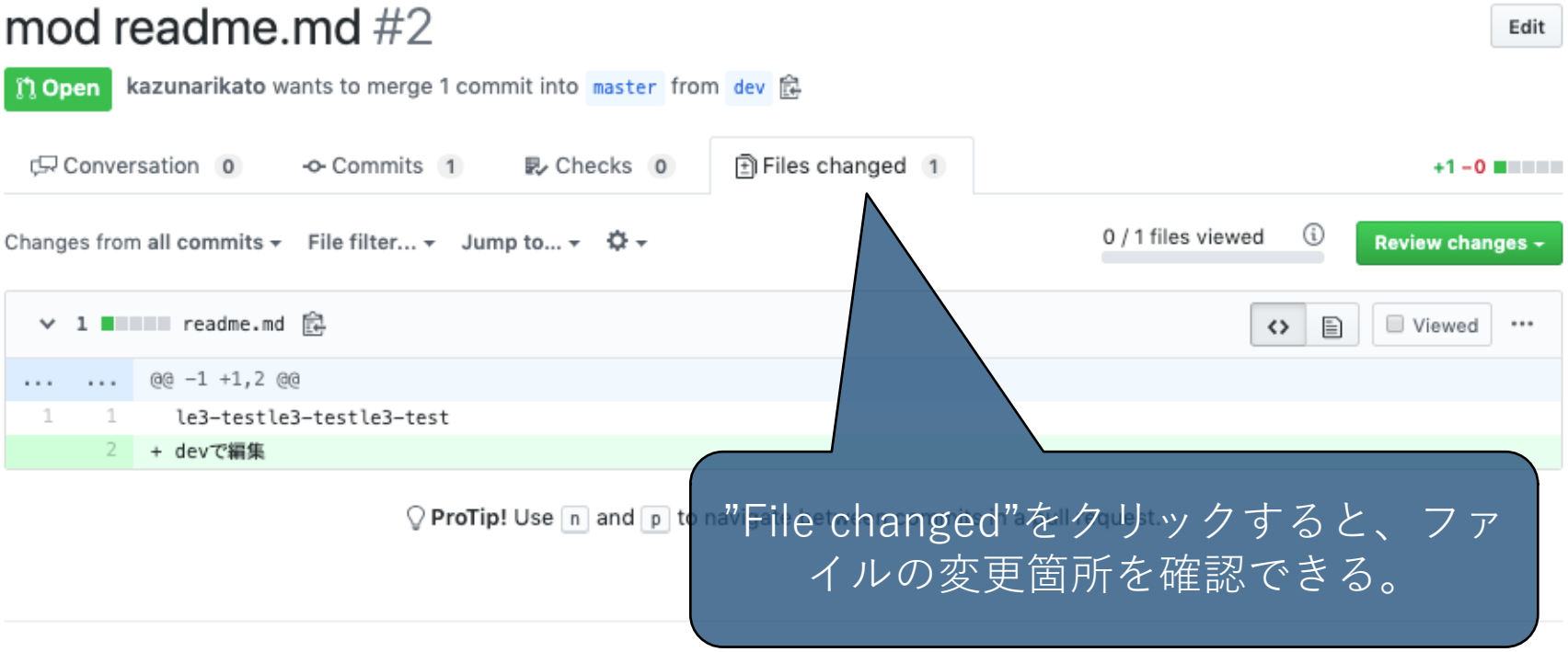
Pull requestの作成



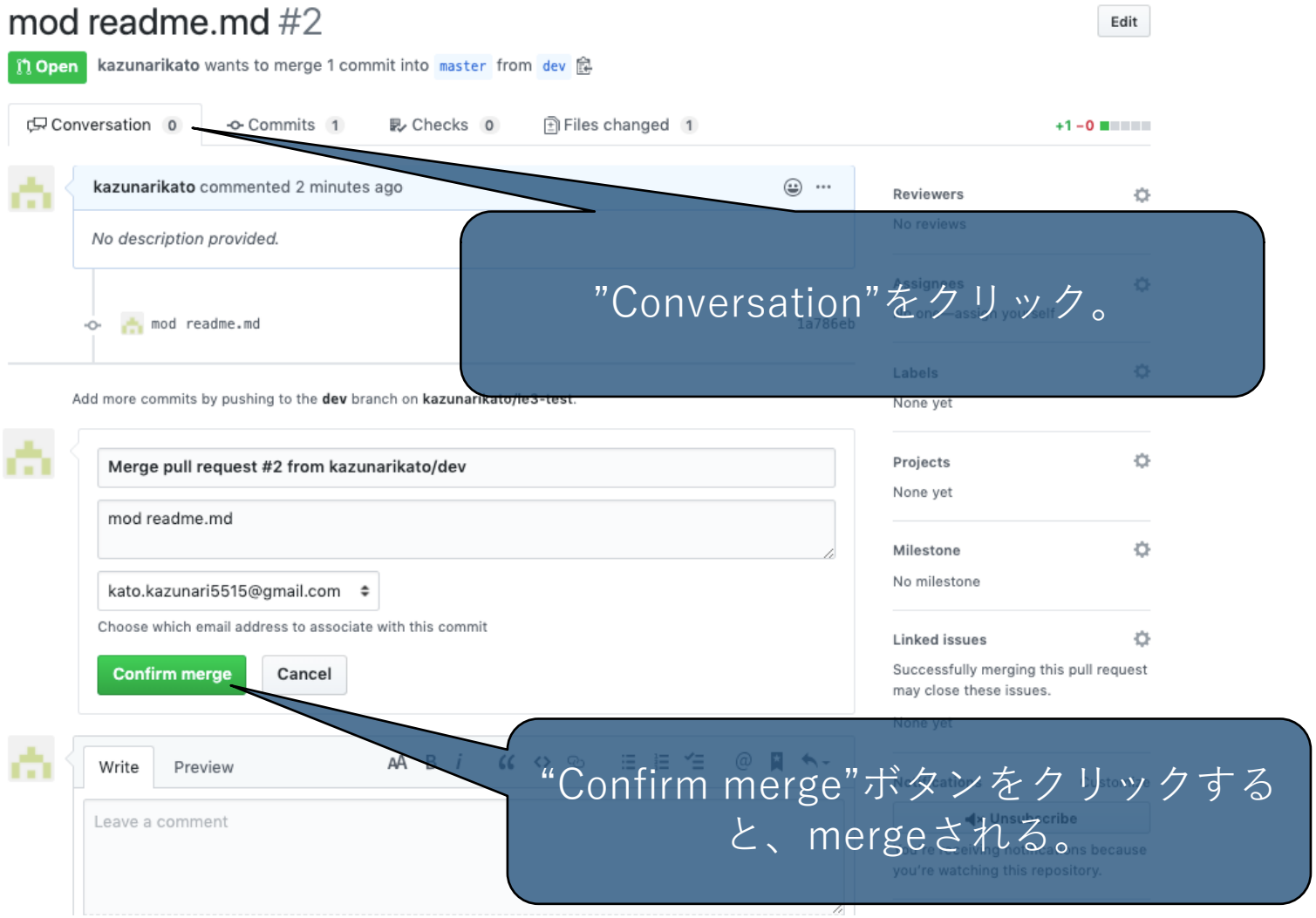
Pull requestの作成



Pull requestの作成

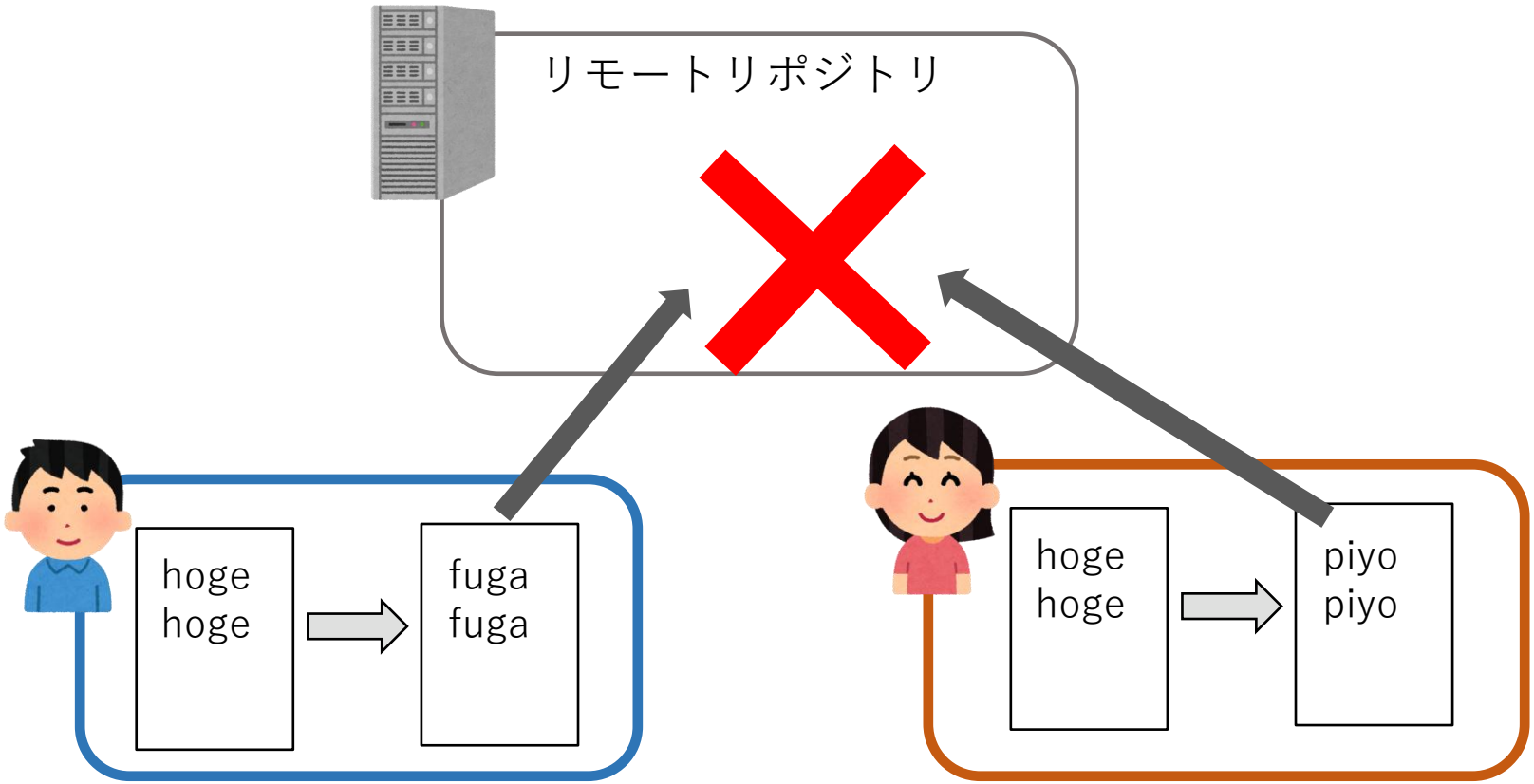


Pull requestの作成



コンフリクト

- 他の人が編集したファイルに重複した編集をした状態でpushやpullをするとコンフリクトが起きる。（1人で作業する場合でもmergeの時に起きる可能性あり）



コンフリクト対策

- コンフリクトを未然に防ぐ
  - .gitignoreを設定して、ログファイルなど不要なファイルはGitの管理対象外にする。(https://docs.github.com/ja/get-started/git-basics/ignoring-files)
  - プルリクエストを使う。
    - まず、作業前（ファイル編集前）にpullする。
  - 二人で１つのファイルを同時に編集しないようにする。
    - ブランチを活用する。
  - こまめに、コミットする。

コンフリクト対策

- コンフリクトしてしまったら
  - git statusコマンドでコンフリクトしているファイルを確認。
  - 直接ファイルを開く。編集が重複している部分が“<<<<HEAD”、“=======”、“>>>>”でハイライトされているので、不要部分を削除する。
  - 編集したファイルをステージング、コミットする。

参考情報

- サルでもわかるGit入門
  http://www.backlog.jp/git-guide/
- Pro Git book（日本語版）
  https://git-scm.com/book/ja/v2
- Git Cheat Sheet
  https://services.github.com/on-demand/downloads/github-git-cheat-sheet.pdf
- GitHubヘルプドキュメント
  https://help.github.com/ja/github