GitとGitHub

一人で使うgit

差分を記録するツール

gitはファイルの変更履歴を管理するCLIツール

補足：GUIとCLI

- **CLIツール**
  - コマンドによって操作できるもの
- **GUIツール**
  - 画面上に表示されるUIによって操作できるもの

gitコマンドを使ってファイルの変更履歴を保存しておくことで、

- 過去のある時点のデータを後から見返すことができる
- どこでバグが紛れ込んだのか調べたり、必要に応じて過去の状態に復元したりもできる

RPGのセーブ機能のようなもの

リポジトリ

gitで変更履歴を管理するには、まず**リポジトリ**を作る必要がある

- リポジトリ ＝ 履歴データの保存場所
- プロジェクト単位で作ることが多い

git init

リポジトリを新規作成するコマンド

.gitディレクトリ

- `git init`コマンドを実行すると、実行フォルダ配下に`.git`という隠しディレクトリが作成される
- この`.git`ディレクトリの作成によって、そのフォルダ内の変更履歴の保存や操作が可能になる

注意

- .gitディレクトリの中身を直接操作することはない
- .gitディレクトリを削除すると、そのPC上からは履歴データがすべて消えてしまう

コミット

変更履歴を登録する操作

コミットの流れ

1. `git add`でコミット対象のファイルを登録
2. `git commit`で対象ファイルの差分を記録

git add .

カレントディレクトリの全ファイルを対象とする

git add [path]

指定したファイルのみを対象とする

なぜgit addが必要？

addは一部の変更だけ確定するための仕組み

- 複数箇所を同時に編集している場合がある
- 完成した部分（ファイルや行）だけaddする

コミットの粒度

一つのコミットとしてまとめる変更の大きさ

- プロジェクトによって方針は異なる
- 基本は、一つの変更を一つのコミットとする

git commit -m "hoge"

- 作業内容のタイトルをつけておくのが作法
- `-m`でコミットメッセージを追加する

複数人で使うgit

変更内容を共有するツール

差分を他のコンピュータと受送信できる

git push

他のコンピュータへ変更内容を送信する

git pull

他のコンピュータから変更内容を受信し、

自分のマシン上のコードにも同じ変更を加える

git clone

他のコンピュータにあるリポジトリを、

自分のマシン上にコピーする

コマンドでの管理は大変

- 確定してからコミット→送信する流れだと、
  作業中はお互いの作業内容がわからない...
- 単純に差分を見るのがつらい...
- etc.

GitHub

gitで管理されているものをWebで共有する

GitHubでできること

- コードを公開する
- コミット履歴をWebサイト上で可視化
- 作業内容が他の人にも見える状態で開発
- etc.

gitとGitHubの関係

gitでは、**他のコンピュータ**と差分を受送信できた

1. gitコマンドで**GitHubサーバー**と差分を受送信
2. すると、GitHubのサイトに反映される

…みたいなイメージ

ローカル／リモートリポジトリ

- **ローカルリポジトリ**
  - 自分のコンピュータ上にあるリポジトリ
  - `.git`ディレクトリにデータがある
- **リモートリポジトリ**
  - GitHubサーバー上にあるリポジトリ

ローカル→リモート

ローカルで作成したリポジトリをGitHubにも作る

1. GitHubのサイト上でリポジトリを作成
2. `git remote add origin [url]`
   - 送信先のリモートリポジトリを登録
3. `git push origin [default-branch-name]`
   - リモートリポジトリに送信

補足：originについて

- 送信先のリモートリポジトリは複数登録できる
- 各リモートリポジトリに名前を設定する

メインのGitHubリポジトリは`origin`という名前にするのが慣習

リモート→ローカル

GitHub上のリポジトリをローカルにもインストール

- **git cloneを使う**
  - ローカルに.gitディレクトリが作られる
  - gitコマンドでGitHubとやり取りできる
  - 実際に開発に参加する場合はこの方法で
- **zipとしてダウンロード**
  - ローカルに.gitファイルは作られない
  - コード全文をダウンロードするだけ
  - ただコードを試したい場合とか？

GitHubでの共同開発

ブランチ

履歴を分割することで、

複数作業を同時に進行させる仕組み

例：投稿へのコメント機能といいね機能を追加

投稿機能が実装された状態（developブランチ）から、それぞれ新たなブランチを作る

1. Aさんはfeature-commentブランチを作って、そこで作業
2. Bさんはfeature-likeブランチを作って、そこで作業

ブランチを分けることで、

他人の作業に影響を与えずに自分の作業ができる

ブランチを作る流れ

Aさんを例にすると、

1. `git checkout develop`
   - 開始状態となるブランチに移動
2. `git branch feature-comment`
   - 作業用ブランチを新規作成
3. `git checkout feature-comment`
   - 作業ブランチに移動

この状態で作業して、`git commit`で履歴を積む

ブランチの関係を表す慣用表現

- devalopからfeature-commentブランチを**切る**
- feature-commentはdevalopから切ったブランチ
- feature-commentの**向き先**はdevelop

ブランチの新規作成・切り替えのショートカット

`git branch -b feature-comment`は、以下と同じ

1. `git branch feature-comment`
2. `git checkout feature-comment`

ブランチを公開する流れ

1. なにか作業して`git commit`で履歴を積む
2. `git push origin HEAD`でGitHubに公開

HEADとは？

今いるブランチの先頭を指す

- ブランチの先頭＝ブランチの最新のコミット

`git checkout feature-comment`した状態なら、

どっちも同じ意味

- `git push origin feature-comment`
- `git push origin HEAD`

マージ

あるブランチでの変更内容を、

今いるブランチに反映する（取り込む）こと

develop → feature-comment

feature-commentブランチで作業している間に、

developブランチが更新されたら...

1. git checkout devalop
2. git pull origin developで最新状態を取得
3. git checkout feature-comment
4. git merge --no-ff develop

feature-comment → develop

`feature-comment`ブランチでの実装が終わったら...

`git merge`でもできるが、業務では通常、

1. プルリクエストを作る
2. プルリクエストをレビューしてもらう
3. プルリクエストをマージする

プルリクエスト（PR）

マージ前にレビューやチェックを行うための
GitHubの機能

PRを作る目的

- 機能単位で差分を残しておく（機能の文書化）
- 他の人によるコードレビューを行う
- CIによるチェックやデプロイを設定する

コンフリクト

同じファイルの同じ行が編集されているけど、

どっちが正しい変更なの？

コンフリクトの解消

コンフリクト箇所の差分を見比べて、

正しい方になるよう手動で編集すること