GitHub Flow

ゴール

- GitHub Flowにそった開発手順を体感する
- git コマンドを使う際に、状態がイメージできる

内容

- どんなにときに、なぜ？
- GitHubとGitHub Flow
- 実践
  - Git 操作で気をつけること
  - GitHub Flow体験
  - より開発しやすくする環境づくり

sfzでの具体的な案件・場面

- sfzで扱う案件はすべてプロジェクトベースで進行 → https://github.com/sforzando

なぜGitHubを使って開発をすすめるのか

- プロジェクトメンバーでタスク／コード／進捗などの管理および共有が可能なため
- 外部協力者とも共同開発が行ないやすいため
- 納品物（ドキュメント／コード）とすることもできるため

Git（GitHub）って？

- Gitはバージョン管理システム
- GitHubは、Gitをクラウド上で管理／共有できるサービス

sfzではどのように使用しているか

- 案件（プロジェクト）毎にリポジトリを用意
- **GitHub Flow**を基本にして、開発を進める

GitHub Flow

なぜGitHub Flowを使うのか

- 着実に開発を進めていくため
- 開発を進める上で起こりがちな不都合を回避できるため

どんな不都合が起こるのか

- 共同開発者同士で同じIssueに着手してしまう
- 共同開発者が互いにどんな変更を加えられたか分からなくなってしまう
- 開発途中で動かない状態ができてしまう

GitHub Flowとは

- 開発を着実に進めていくために行なう手順
  - 課題設定（Issue） →
  - → 開発ブランチ作成 → コーディング（Coding+Test）→
  - → 変更点まとめ（PR）→ 変更点のレビュー（Review）→
  - → 変更点をmasterへ統合（Merge）→ デプロイ（Deploy）

- 主な流れとなるのは2種類
    - masterブランチ（本流: 1つ）と開発用ブランチ（支流: 複数）
- masterブランチ（本流）
    - バグなどがなく、正常に稼働できる状態のもの
    - 開発用ブランチが取り込まれることによって、成長していく
- 開発用ブランチ（支流）
    - masterブランチから別れ、何かしらの変更を加えるためのもの

重要! sfzにおける3つのGitHub Flowのルール

- masterは常に稼働状態
- commitはエラーがでない変更に対して行なう
- Issueファースト（必ず最初にIssueを作る）

Git（GitHub）実践

Gitの初期設定

- Git でのユーザ名を設定する - GitHub Docs
  - `$ git config --global user.name "Mona Lisa"`
  - `$ git config --global user.email "email@example.com"`

sfzのリポジトリをクローンする

- sforzando/WorkingRules: The Working Rules
- sforzando/EmployeeTraining: Employee training documents for sforzando LLC. and Inc.
- `ghq get ~` を使うと良い
- pecoとの組み合わせでリポジトリを移動すると良い
  - `cd $(ghq root)/$(ghq list | peco)` → alias 登録 → `repo`

Gitの操作で迷子にならないために

- gitの登場人物を意識する
  - リモートリポジトリ、ローカルリポジトリ、ステージングエリア、ワークツリー
- 状態を意識する
  - どのブランチにいて、どういう状態になっていて、どうしようとしているのか
  - gitコマンドを実行した後に、どういう状態になるのかイメージできているか
  - エディタ(Visual Studio Codeなど)でのGit機能を使うと、状態がつかみやすくなる
    - source control機能（VScode標準機能）
    - eamodio/vscode-gitlens（VScode拡張機能）
    - DonJayamanne/gitHistoryVSCode（VScode拡張機能）

Git の操作で気をつけること

- `git <command> -f` は使わない
  - `-f`, `--force` オプションは強制的にコマンドを実行
  - 使いたくなる場面（pushできないってアラートでるけど、、）の場合、それは使うと不都合なことになる場面だから注意してくれている
  - → 使ってしまうと不都合になる
- `commit`, `push` の前には、よくよく状態を確認する
  - `git status`
  - `git log`, `git show`
  - `git diff`

よく使う git コマンド

- `git log`, `git show`, `git status`
- `git branch`, `git checkout`
- `git add`, `git commit`
- `git reset`
- `git push`
- `git pull`, `git fetch`, `git merge`
- `git stash`

Zsh + prezto での alias を見ておこう

- `$ alias | rg git`

GitHub Flowやってみよう

1. Issue作成
2. 開発
3. PR&レビュー&修正
4. 開発進行と同期をとりつつ、新しい開発へ

cf. sfz 流 GitHub Flow の流れ - sfz

ステップ1: Issue作成

- GitHubでリモートリポジトリを作成
- GitHubでIssueを作成

ステップ2: 開発

- リモートリポジトリをローカルリポジトリへcloneする
- ローカルリポジトリでIssueに対応するbranchを作成
  - ブランチ名は、`<Issue番号>_<タイトル>` としている（Issue番号は0埋め3桁）
- 作成したbranchへcheckoutする
- コーディングし、変更をcommitする
  - 変更点の確認
  - 変更点をステージに乗せる（=addする）
  - ステージに乗った変更点をcommitする
- commitをリモートリポジトリへpushする

ステップ3: PR&レビュー&修正

- Issueに対応できたら、GitHubでPR（Pull Request）を作成する
- レビューおよび修正
- PRにapproveがもらえたら、masterブランチへマージする

ステップ4: 開発進行と同期をとりつつ、新しい開発へ

- ローカルリポジトリでのmasterブランチも最新の状態にしておく
  - masterブランチへcheckout
  - リモートリポジトリからfetchしmergeする（=pullする）
  - 新しいIssueに取りかかる

リポジトリの環境を整えよう

- GitHubの設定で、masterブランチへのマージ条件を制限する
- Issue, PRなどのテンプレートを用意する
- `.gitignore` を用意する
  - GitHubで共有する必要のない、共有してはいけないファイルを定義する
  - `$ git status` に表示されなくなるため、`$ git add` → `$ git commit` によって対象のファイルやディレクトリがcommitに含まれるのを防ぐができる
  - → こちらで生成できるので利用すると良い
    - gitignore.io - Create Useful .gitignore Files For Your Project
- Slackと連携
- Codecovと連携
  - The Leading Code Coverage Solution | Codecov
    - コードのカバレッジを可視化、通知などしてくれるサービス

まとめ

- GitHub Flowで着実に開発を進めていきます
- Git操作では状態を意識しましょう