研究者のためのGit入門 実践編

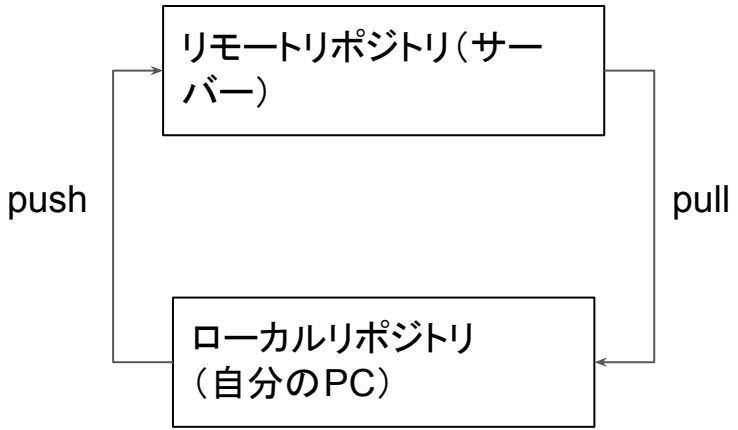
DeepFlow株式会社
神志那 純

はじめに

- Gitをどう使うのか？
- 特にDeepFlowではどうしているのか
  - Git導入時にどう動くのかを時系列順に寄り道しながら
  - 研究室やチームで使うことを想定した話
- 課題
  - チーム開発の効率化
    - 過去の自分自身ともうまく連携する
  - ソースコードの品質管理
- キーワード
  - Pull Request(PR)
  - CI：継続的インテグレーション

Gitのホスティング

- Gitのリモートポジトリを管理するサーバーをどうするのか？
- SaaS
  - GitHub, GitLab, Bitbucket
  - Webサービス
  - 機能やコストを比較して
  - アカデミックプランも活用する
- セルフホスト
  - GitLab
  - 自前で運用
  - 一切データを外に出したくないときに
  - インフラが得意な人がいないならオススメしない
- DeepFlowは
  - GitHub → GitLab → GitHub
  - Gitの情報は移行出来る

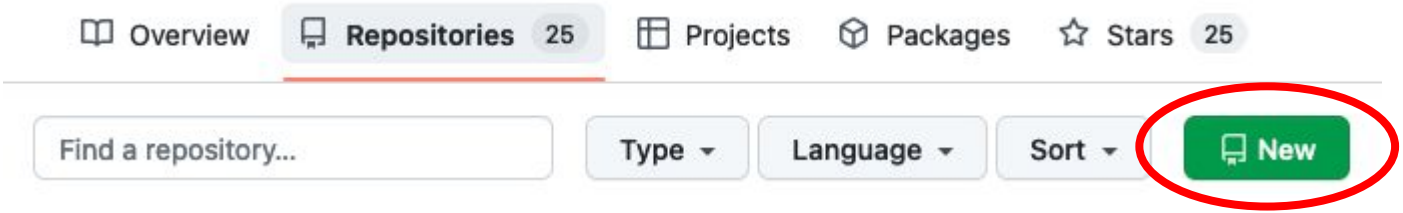


ユーザー管理

- OrganizationやGroupという機能でまとめる
  - 権限の管理が楽になる
- 最小権限の原則
  - ロールの設定では必要最低限の権限を与えるようにする
  - TeamやSubgroup等の機能を駆使する
- 2要素認証（2FA）の強制
  - IDとパスワードの情報だけではログイン出来ないようにする
  - 強制するような設定ができる
  - スマホアプリの Google Authenticatorや物理キーの Yubikeyとか
  - 後から強制するのは大変なので，最初に設定しておく
- パスワード
  - パスワードマネージャを活用する
  - パスワードの使い回しは絶対にしない

リモートリポジトリの作成（Web上）

- リポジトリ
  - Gitでバージョン管理するプロジェクトの単位
- 原則プライベートで作成する
- チームで使う場合は誰か1人が作れば良い
- リポジトリの設定
  - メンバーのアクセス権の設定（最小権限の原則）
  - ブランチの保護設定（後述）
  - 迷ったら厳しい方に
  - 運用しながら調整していく



ローカルリポジトリの準備（自分のPC）

- Gitのインストール・アップデート(2.36.1が最新)
- Gitの初期化
  - $ git init
- リモートとローカルの対応を登録
  - $ git remote add origin https://github.com/{user}/{new-repository}
- 設定
  - $ git config --local user.name “Jun Kohjina”
  - $ git config --local user.email “jun.kohjina@example.com”
- コマンドラインではなくGUIで操作してもよい
  - 自分はVSCodeを使ってGUIでGit操作しています
  - GUIの裏側でどういうコマンドが叩かれているかは知っておいた方がいい

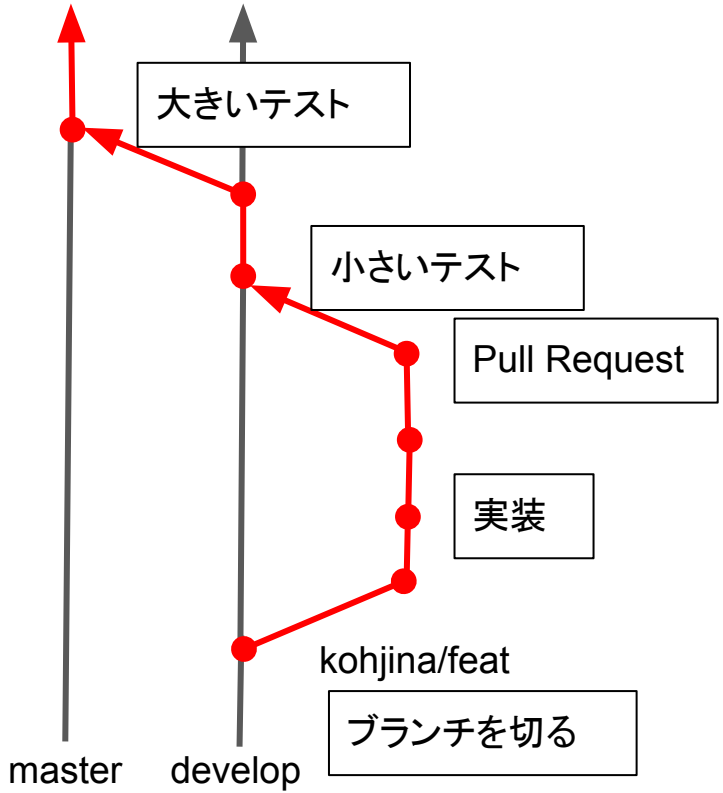
リポジトリに含めるファイル

- README.md
  - 最低でも動かし方だけは書いておく（半年後とかでも動かせますか？）
  - 最近はマークダウン形式が多い
  - オープンソース化するときはライセンスも書く
- .gitignore
  - gitで管理しないファイルを指定する
  - *.o とか *.exeとかはgitで管理しない
- .gitattributes
  - 改行コード(\n vs \r\n)の設定等
- .vscode/settings.shared.json
  - DeepFlowではエディタはみんなVSCodeを使っている（これはたまたま）
  - フォーマッタやリンタの設定を共有 (Workspace Config+)

リポジトリに含めないファイル

- .gitignoreに書くもの
- バイナリファイル
  - リポジトリの容量をくいやすい
  - ビルド済みの実行ファイルや中間生成物とか
    - *.o *.exe *.a *.dylib .DS_Store .venv
  - ディレクトリの指定も出来る
  - 実験用データ
  - https://github.com/github/gitignore を参考にする
- 実験用・テスト用のデータはDVCというツールで管理している
  - 他にはgit-lfsを使う方法がある
- workspaceディレクトリ
  - 作業用ディレクトリ。各メンバーが自由に使う
  - 出力やログはすべてworkspace以下にはくようにする

Gitワークフロー（概要）



既存コードに変更を加える手順

1. developからブランチkohjina/featをきる
2. kohjina/featブランチで実装を進める
3. Pull Request(PR)を作成（Draftで作る）
4. テストに通るまで頑張る
5. 必要であればコードレビューを依頼
6. PRをdevelopブランチにマージ
7. 大きいデータを使ってテスト
8. developをmasterにマージ

ここからの話のながれ

- 開発ルールの設定
  - 何を保証して何を保証しないかをはっきりさせる。保証するならテストをする。
  - ブランチ運用
  - テスト
- 開発ルールのもとでどう目的（コードの変更）を達成するか
  - コードの変更手順
  - コミット
  - Pull Request / Merge Request
- 開発ルールを楽に守るにはどうするのか
  - 保護ブランチ機能 (Protected Branch)
  - CI：継続的インテグレーション
  - 開発を加速させる
    - 保証されていることがあるから加速させることができる

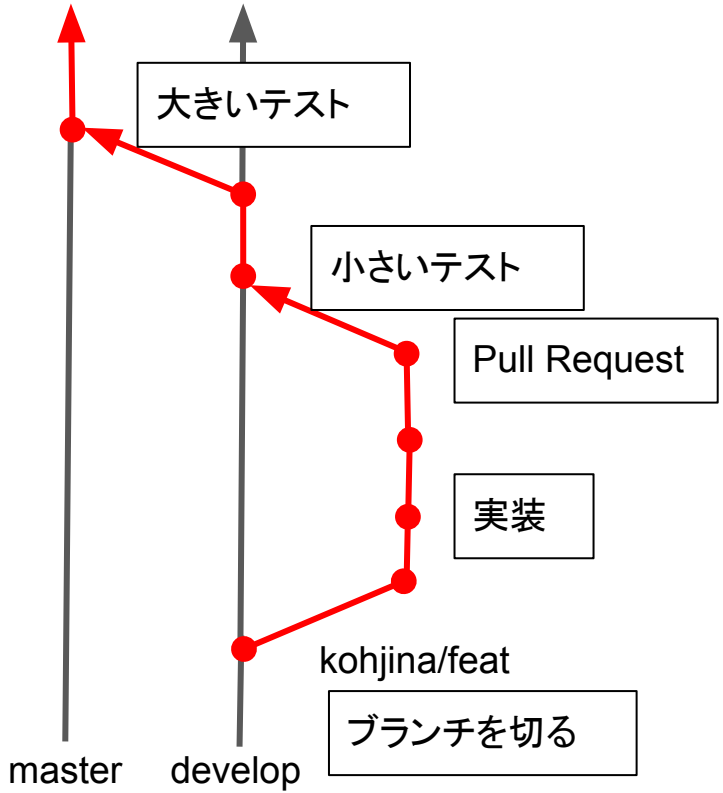
ブランチ運用

- 品質保証レベルで分類
- master
  - 十分に検証されたもののみマージする（大きいデータでテスト）
  - 時間のかかる検証を行うので更新頻度は低い
  - 直接push禁止
- develop
  - ビルド・テストにすべてパスしたもののみマージする（小さいデータでテスト）
  - 自動で検証できるので更新頻度は高い
  - 直接push禁止
- {username}/{branch-name}
  - リポジトリのフォークはしない
  - ブランチ名にユーザー名をいれて責任を明確にする
  - 検証中のものや実装途中のものも気軽に pushする
  - pushするたびに自動でビルドやテストが走る

テスト

- ビルドできるか
  - -Wall をつけてビルドする。警告も無視しない
  - 自分のPCでしかビルドできないとかはよくある
- スモークテスト
  - エラーをはかずに最後まで実行できるか
  - 出力結果は気にしない
- リグレッションテスト
  - 同一の入力データに対して同じ出力を返しているか？
  - 入出力データのペアを管理する
  - 浮動小数点では誤差が閾値以下かをみる（基本的には完全に一致するか）
- 単体テスト，コードフォーマットのテスト
- 上から順に導入して，品質を上げていく
  - コードに変更をいれなくていいものから対応する
  - 既存コードが小さいなら，フォーマットテストは早めに入れたほうがよい

開発ルールのまとめ



左に行くほど

- 高品質（難しいテストもパスしている）
- 変更頻度が低い

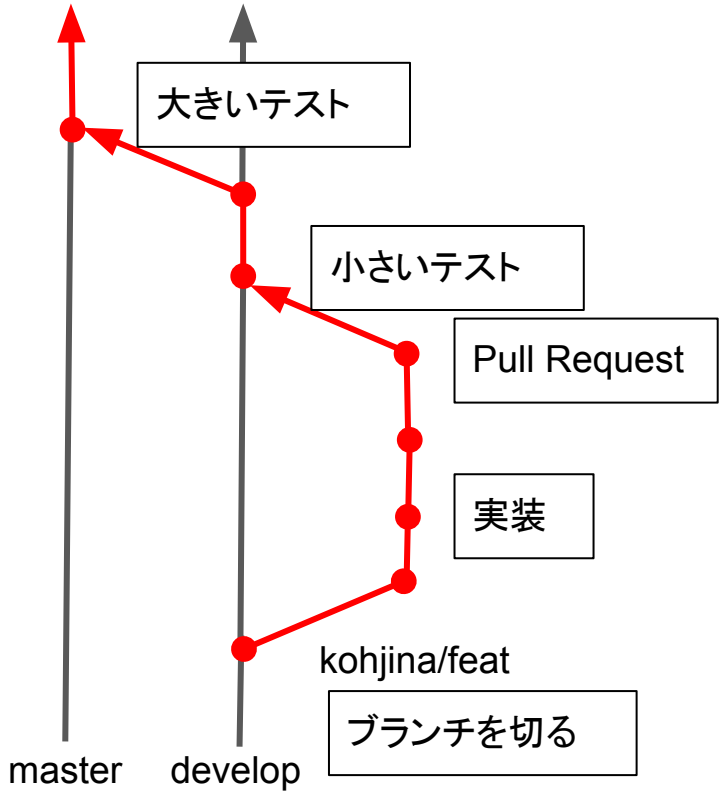
テストの実装

- masterやdevelopを左に動かすということ
- テストを追加して保証できることを増やす

ここからの話のながれ

- 開発ルールの設定（ソフトウェアの品質を保証する）
  - 何を保証して何を保証しないかをはっきりさせる。保証するならテストをする。
  - ブランチ運用
  - テスト
- 開発ルールのもとでどう目的（コードの変更）を達成するか
  - コミット
  - Pull Request / Merge Request
- 開発ルールを楽に守るにはどうするのか
  - 保護ブランチ機能 (Protected Branch)
  - CI：継続的インテグレーション
  - 開発を加速させる

Gitワークフローの具体的手順



既存コードに変更を加える手順

1. developからブランチkohjina/featをきる
2. kohjina/featブランチで実装を進める
3. Pull Request(PR)を作成（Draftで作る）
4. テストに通るまで頑張る
5. 必要であればコードレビューを依頼
6. PRをdevelopブランチにマージ
7. 大きいデータを使ってテスト
8. developをmasterにマージ

コミットの作成（自分のPC）

- 最新のバージョンを取得
  - $ git fetch
- developからブランチをきってチェックアウト
  - $ git checkout -b kohjina/feat origin/develop
  - developはテストに通ることを保証している
    - これ以降テストに通らなくなったら自分がバグを埋め込んだということ
- コードを変更
- ステージ
  - $ git add {コミットに含めたいファイル }
  - 慣れるまでは $ git add . でいいと思っています
- ステージしたファイルをコミット
  - $ git commit -m ‘コミットメッセージ’

コミット

- どのタイミングでコミットするのか
  - 基本的に細かくコミットする
  - ビルドに通った，動くようになった，テストしたい，デバッグが大変になってきた
  - 基本的に試行錯誤というかデバッグしてるのでそのログを残す感じで
  - 細かくコミットして，レビュー前にまとめたりする
- コミットメッセージ
  - Conventional Commits(https://www.conventionalcommits.org/ja/v1.0.0/)
  - コミットメッセージにタグを付ける
  - コミットメッセージよりかは PRのコメントを重視
  - 日本語
  - 例
    - fix: 境界条件の修正
    - feat: 出力形式にXXXを追加

Pull Request (PR) / Merge Request (MR)の作成

- kohjina/featのコードをdevelopに反映させるときに使う機能
  - Gitそのものの機能ではなくGitHubやGitLabの機能
  - Web上で行う
- ローカルの変更をリモートに反映させる（アップロード）
  - $ git push origin kohjina/feat
  - ローカルと同じ名前のリモートリポジトリに pushする
- GitHubにアクセスしてPull Requestを作成
- 実装が完全に終わってなくてもPRを作る
  - DraftとしてPRを作成する機能がある
  - 早い段階で何をしているかを共有する
  - こまめにpushしてバックアップする

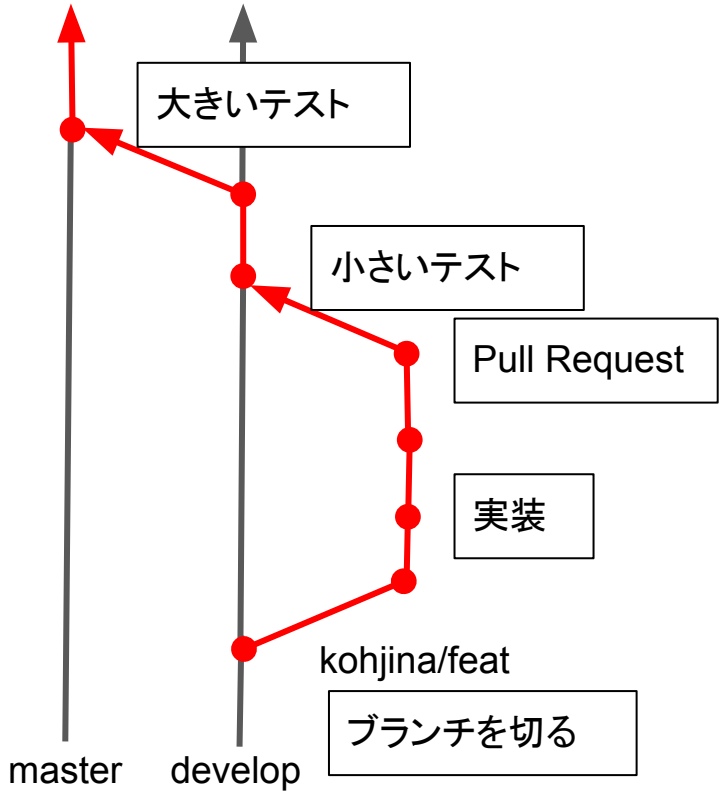
Pull Request (PR) / Merge Request (MR)

- コードレビュー
  - developにマージして大丈夫かレビュアーがチェックする
  - DeepFlowでは必須ではなく，必要に応じてお願いする
  - レビューを頑張るよりかはテストを充実させる
  - 機械的にチェックできるものは人にチェックさせない
  - 普段からコードについてはよく議論している
- マージ
  - テストに通っていれば，レビューなしでマージして OK
  - Gitに慣れてない人がチームに居るときは Squash Merge
    - 複数のコミットをすべて1つのコミットに自動でまとめられてからマージされる
    - きれいなコミットログを作るよりも，Git操作が単純になるように
  - みんな慣れているなら，コミットを整えてからマージ
    - 単純にsquashするとコミットが大きくなりすぎることがある
    - Conventional Commits

ここからの話のながれ

- 開発ルールの設定
  - 何を保証して何を保証しないかをはっきりさせる。保証するならテストをする。
  - ブランチ運用
  - テスト
- 開発ルールのもとでどう目的（コードの変更）を達成するか
  - コードの変更手順
  - コミット
  - Pull Request / Merge Request
- 開発ルールを楽に守るにはどうするのか
  - CI：継続的インテグレーション
  - 保護ブランチ機能 (Protected Branch)
  - 開発を加速させる
    - 保証されていることがあるから加速させることができる

Gitワークフローを楽に運用する



既存コードに変更を加える手順

1. developからブランチkohjina/featをきる
2. kohjina/featブランチで実装を進める
3. Pull Request(PR)を作成（Draftで作る）
4. テストに通るまで頑張る
5. 必要であればコードレビューを依頼
6. PRをdevelopブランチにマージ
7. 大きいデータを使ってテスト
8. developをmasterにマージ

CI（継続的インテグレーション）

- 自動テストを活用した開発手法
- GitHub Actions, GitLabCI, CircleCI
- マージ時に必要なテストを自動化する
  - ビルド，リグレッションテスト，フォーマットチェック
  - gitでpushするたびに自動でテストを走らせる
    - GitHubやGitLabの機能として自動で走らせることができる
    - 設定ファイルをリポジトリに追加すると動く
- テストはCI環境で動作するように調整する
  - 自分のPCでしか動かないということにならないように
  - Dockerのような仮想環境も活用する
  - どうやって動かすかのドキュメント代わりにもなる

保護ブランチ機能(Protected Branch)

- 経験的にはこれが1番大事な設定
- CIまで構築出来るとよいが，その前にこの設定を入れたほうがいい
- master/developブランチへの直接プッシュを禁止する
- プルリクエストを経由しないとdevelopブランチにマージできないように
- CIに通ってないとdevelopブランチにマージできないように
- 安全に失敗できるように
- 慣れていてもミスすることがある
- いちいちPR作るのは面倒かもしれないが，CIと組み合わせると便利になる
- 特に，慣れてない人がmasterに間違えてプッシュしてGitが怖くなるようなことを避けたい

開発を加速させる

- コミットとテスト結果がひもづく
  - GitHubでこのコミットはテストに通っているかはすぐに分かる
  - どのタイミングでバグが出たかがわかる
  - 進捗の共有になる（テストに失敗したときにログも共有できる）
  - 将来なぜこういう実装になっているか調査する際のヒントになる
- カジュアルにpushしてCIを走らせる
  - 正しく動くか頭で考える前にまず pushしてテストを走らせる
  - 細かくpushしていると早い段階で失敗できる
  - テストしている間に実装を進める
  - 結果としてデバッグがはやくなる
  - CIでテストに落ちることは恥ずかしいことではない
    - elkurageは成功率62.89%

まとめ

- 最小権限の原則
- 検証レベルに応じてブランチをわける
  - master：十分に検証されたもののみ
  - develop：ビルドができて数分程度で終わるテストにパスする
  - {username}/{branch-name}：テストに通ってなくてもOK
- Pull Requestを通した開発
- CIを利用した高速な開発
  - 自動テスト・自動ビルド
- 保護ブランチ機能やCIによるブランチの保護
  - ビルドやテストに通らないものは developやmasterにマージできないように
  - 大事なブランチには直接変更を加えない
  - 安全にGit操作で失敗できるように
- ブランチでの品質を保証しつついかに楽をするか