Git Worktree

高度な GIT



Amanda Crawford-Adamo
Software and Data Engineer

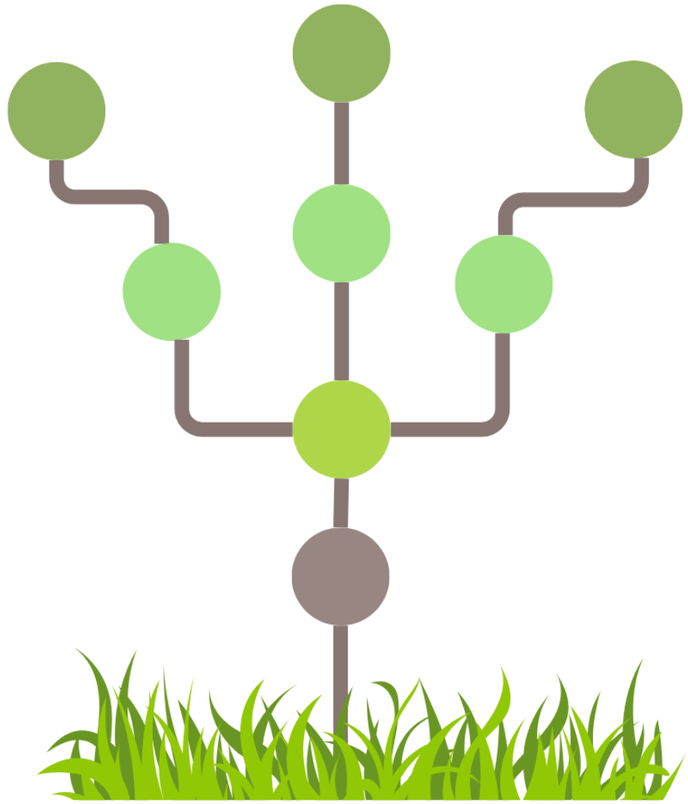
datacamp

Git Worktree とは

Git Worktree コマンド

```
git worktree
```

- ワークスペースで複数ブランチを同時に checkout 可能
- 通常のリポジトリ checkout に近いが効率的
- stash は不要
- 開発中にブランチを切り替える必要なし



Git Worktree と Git Switch の比較

この表は、開発ワークフローにおける git worktree と git switch の比較です。

Git Worktree	Git Switch
複数のアクティブブランチ	同時に1つのアクティブブランチ
個別のディレクトリ	単一の作業ディレクトリ
stash 不要	stash が必要になる場合あり

Git Worktree の作成

<branch> から <path> に新しいワークツリーを作成

```
git worktree add <path> <branch>
```

例

`bugfix/data-validation` ブランチから `../etl-bugfix` ディレクトリに新しいワークツリーを作成

```
git worktree add ../etl-bugfix bugfix/data-validation
```

ワークツリーの一覧と削除

- すべてのアクティブなワークツリーを一覧表示: `git worktree list`

出力例

```
$ git worktree list
flight-pipeline             a1b2c3d [main]
flight-pipeline-feature     e4f5g6h [feature]
flight-pipeline-hotfix      i7j8k9l [hotfix]
```

- <path> のワークツリーを削除: `git worktree remove <path>`

出力例

```
$ git worktree remove flight-pipeline-hotfix
flight-pipeline-hotfix: deleted
```

Git Worktree を使うべきとき

使用に適する場面:

- 複数機能を同時開発する
- 進行中の作業を乱さず緊急バグ修正
- 異なるブランチで並行テスト
- 開発を続けながらコードレビュー

再検討すべき場合:

- ディスク容量が限られる
- 更新頻度が高くマージが複雑なプロジェクト

Git Worktree のベストプラクティス

Git worktree 利用時のポイント:

1. ワークツリーのディレクトリ名規約を明確にする
2. 使わないワークツリーは定期的に prune して整理
3. 大規模プロジェクトでは特にディスク容量に注意
4. 混乱防止のため短期の並行作業に使う

練習しましょう！

高度な GIT

Gitサブモジュール

高度な GIT



Amanda Crawford-Adamo
Software and Data Engineer

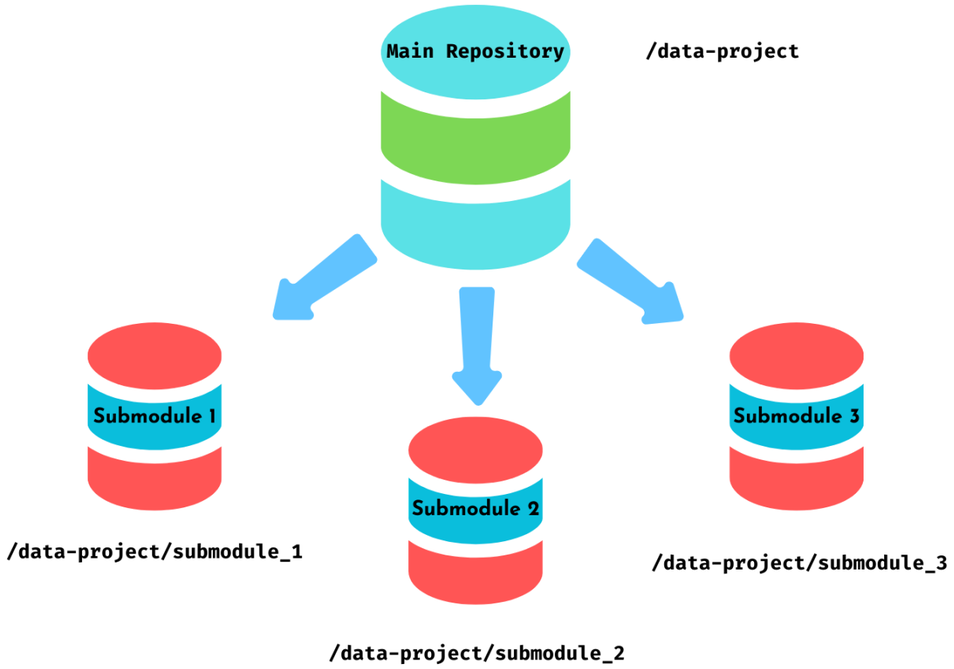
datacamp

Gitサブモジュールとは

Gitサブモジュール

```
git submodule
```

- 別リポジトリをリポジトリ内にネスト
- それぞれ独立した履歴とバージョン管理
- サブモジュールの変更はメインに影響しない
- メインはサブモジュールの特定バージョンを参照可能



サブモジュールの追加

パス配下にあるリンクまたはディレクトリを指定してサブモジュールを追加します。

```
git submodule add <repository link|dir> <path>
```

例

ETLプロジェクトの `libs/validator` 配下に data validator ライブラリを追加します。

```
git submodule add https://github.com/example/data-validator.git libs/validator
```

サブモジュールの一覧表示

プロジェクト内のサブモジュール一覧

```
git submodule status
```

例

```
$ git submodule status
e1f2...7w8x9 data_cleaning_lib
a1b2...q7r8 api_connector
d9e8...t3u2 visualization_toolkit
```

サブモジュールの更新

最新の変更でサブモジュールを更新

方法はいくつかあります:

1. ソースがローカルにある全サブモジュールを更新。

```
git submodule update --init
```

2. ソースがリモートにある全サブモジュールを更新。

```
git submodule update --init --remote
```

3. 特定のサブモジュールのみ更新。

```
git submodule update --init <path_to_submodule>
```

サブモジュールの削除

サブモジュール削除の手順

1. サブモジュールを初期化解除。

```
git submodule deinit <submodule_name>
```

2. Gitのインデックスから削除。

```
git rm <path>
```

大規模リポジトリからのサブモジュール抽出

1. 新しいサブモジュール用に必要なファイルを、リポジトリ外の別フォルダへコピーします。
2. そのフォルダ内でサブモジュール用の新規リポジトリを作成:

```
git init <new-submodule>
```

3. git **filter-repo** でメインから対象ファイルと履歴を抽出:

```
git filter-repo --path <extract_path> --invert-paths
```

4. 抽出したリポジトリをメインに**サブモジュール**として追加:

```
git submodule add <new-submodule_path> <path_to_store_submodule>
```

サブモジュールの利用場面とベストプラクティス

ユースケース:

1. 外部ライブラリの管理
2. 複数プロジェクトでのコード共有
3. 依存関係の特定バージョン維持

ベストプラクティス:

1. サブモジュールを最新に保つ
2. 相対パスを使用
3. 変更をチームと共有

Let's practice!

高度な GIT

Git Large File Storage

高度な GIT



Amanda Crawford-Adamo
Software and Data Engineer

datacamp

Git Large File System とは？

Git LFS コマンド：

```
git lfs
```

- Git LFS: Git Large File Storage
- リポジトリ内の大きなファイルを置換
- 小さなポインターファイル
- 大きなファイルはリポジトリ外で管理

利点：

1. リポジトリサイズを削減
2. クローンと取得が高速化
3. バイナリの扱いが効率的
4. 大容量ファイルでの共同作業が向上

Git LFS の初期化手順

- Git LFS を初期化

```
git lfs install
```

- 追跡対象を設定し `.gitattributes` を生成

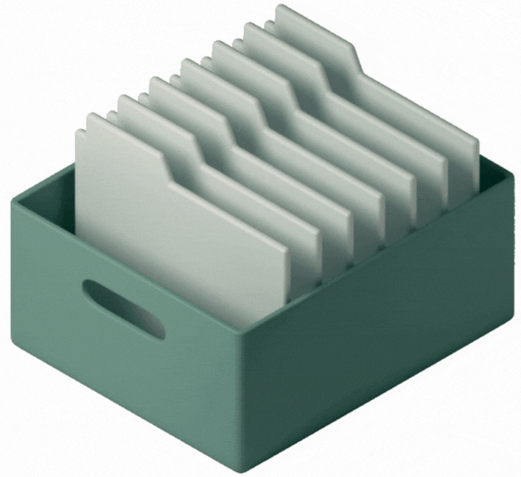
```
git lfs track "*.csv"
```

- 追跡設定の `.gitattributes` をインデックスに追加

```
git add .gitattributes
```

- 変更をコミット

```
git commit -m "Track CSV files"
```



Git LFS の更新手順

1. `git add` で新規ファイルを追加

```
git add large_file.csv
```

2. コミットしてプッシュ

```
git commit -m "Update large CSV file"
git push origin main
```

3. 変更を取得

```
git pull
git lfs pull  # 必要なら LFS 内容を明示的に取得
```

Git LFS を使うべきとき

利用する場面：

1. 大規模データセット（CSV、JSON など）の変更管理
2. 機械学習モデル
3. バイナリアセット（画像、動画）
4. 圧縮ファイルやインストーラーのバージョン管理

利用しない場面：

1. ほとんど更新しない大容量ファイル
2. コードなどの小さなテキストファイル
3. ストレージ制限が厳しい場合

ベストプラクティス

1. 大きなファイルを効率管理
2. データ量の多いプロジェクトで協業を改善
3. Git ワークフローとシームレス統合

ヒント：

1. 対象ファイルを選択的に追跡
2. LFS 利用をチームに周知
3. LFS キャッシュを定期的に削除

Passons à la pratique !

高度な GIT

トランクベース開発

高度な GIT

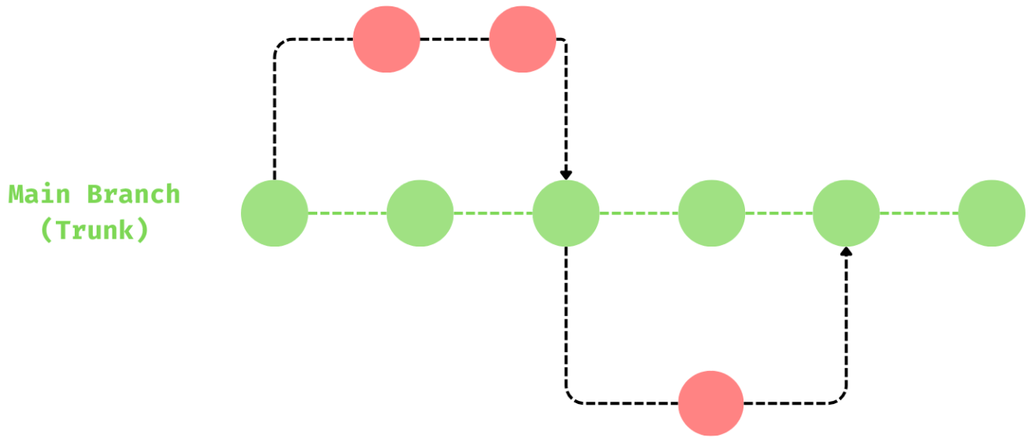


Amanda Crawford-Adamo
Software and Data Engineer

datacamp

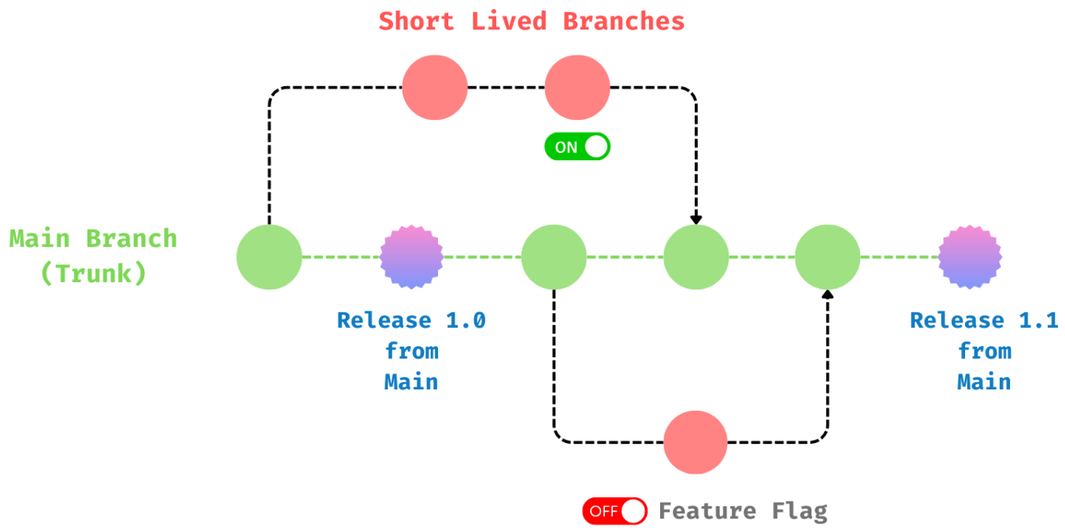
トランクベース開発とは？

- ソース管理のブランチと CI/CD モデル
- 開発者は別のリリースブランチへは押し込まない
- 変更は短命ブランチから main へ
- 小さく頻繁な更新



トランクベース開発の中核原則

1. main への高頻度コミット
2. 短命の機能ブランチ（< 1 日）
3. 継続的インテグレーション
4. 未完成作業にフィーチャーフラグ



TBD におけるフィーチャーフラグ

- 未完成機能を管理
- ユーザーへの影響を防止
- 段階的にリリース

フィーチャーフラグの例

```
if feature_flag_enabled('new_feature'):
    # New feature code
else:
    # Old feature code
```

TBD における継続的インテグレーション

- main へのコミットで自動ビルドとテストを起動
- 保守負荷を削減し、リリースを高速化
- 常に信頼性と安定性を確保
- セキュアコードと業界標準への準拠を担保
- コード品質を維持し、バグリスクを低減

TBD の利点と課題

利点:

- マージ競合の減少
- リリースの高速化
- コード品質の向上
- コラボレーションの強化

課題:

- チームの規律が必要
- 強固なテストが必要
- 初期の学習コスト
- 未完成機能の管理

ベストプラクティス

1. 小さな変更を高頻度でコミット
2. テストとデプロイを自動化
3. 未完成作業にフィーチャーフラグを使用
4. 定期的なコードレビュー
5. デプロイ後を監視

練習してみましょう！

高度な GIT

まとめ

高度な GIT



Amanda Crawford-Adamo
Software and Data Engineer

datacamp

章の学び

Chapter 1 **Advanced Merging Strategies**	Chapter 2 **Git History and Exploration**	Chapter 3 **Advanced Repository Management**
Fast Forward Recursive Squash Octopus Rebase	Cherry Picking Bisect Filter-Repo Reflog	Worktrees Submodules Git Large File Storage Trunk Based Development

重要ポイント

Manage complex merge and commit history

Handle large files and organize code efficiently



Investigate and modify repository history

Implement modern dev workflows like Trunk Based Development

次のステップ

- git hooks を学ぶ
- 高度な CI/CD 連携手法を探る
- オープンソースに貢献して実践する
- 新しい Git 機能と更新情報を追う

おめでとうございます！

高度な GIT