gitの動作を理解する。

コミット、ブランチを詳しく知る
繰り返しの部分もありますが少し座学にお付き合いください。

コミットとは？

変更の記録を行う。

Gitの基本: コミットとは？

前にも説明しましたがコミットは、変更を記録するスナップショットのようなものです。

- ファイルの変更履歴を保存
- 変更内容にメッセージを付けられる
- `git commit -m "メッセージ"`

コミットが持つ情報

コミットは以下の情報を持っています。

- **コミットハッシュ**: 一意の識別子
- **Author (作成者)**: 誰がコミットしたか
- **Date (日付)**: コミットの作成日時
- **Commit Message**: 変更の説明
- **Parent (親コミット)**: 直前のコミット

```
git log --pretty=fuller
```

コミット履歴の連続性

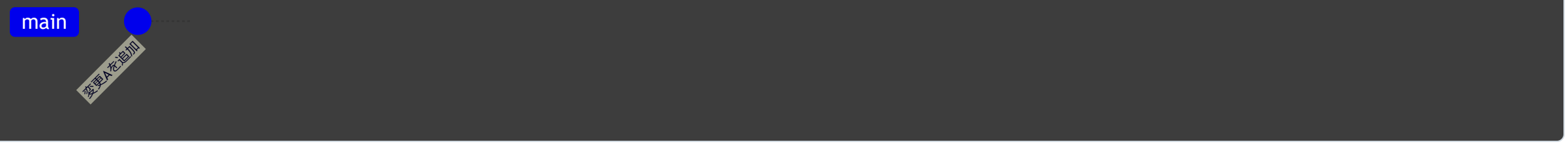
- すべてのコミットは **親コミット (Parent)** を持つ
- Parent をたどることで **必ず最古のコミットまで遡れる**

どんなにブランチで分岐してても最古は一つ！

コミットの作成手順

1. 変更を加える（`git add`）
2. コミットを作成（`git commit -m "メッセージ"`）

```
git add ファイル名
git commit -m "変更内容を説明するメッセージ"
```



これは知っての通りですね♪

ブランチとは？

特定のコミットからの分岐

Gitの基本: ブランチとは？

ブランチはコミットの流れを分岐させる仕組みです。

```
git branch feature-x
```

ChatGPTさんが謎のfeature-xブランチを作る手順を教えてくれているところｗ

ブランチの情報

ブランチは以下の情報を持っています。

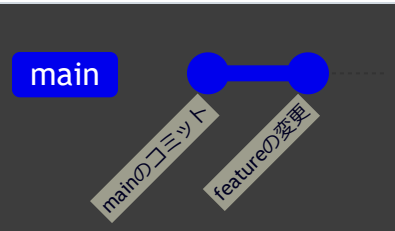
- **ブランチ名**: `main`, `feature` など
- **現在のHEAD**: どのコミットを指しているか
- **リモートとの関係**: `origin/main` など

```
git branch -v
```

ブランチの削除

- マージ後、不要になったブランチは削除可能

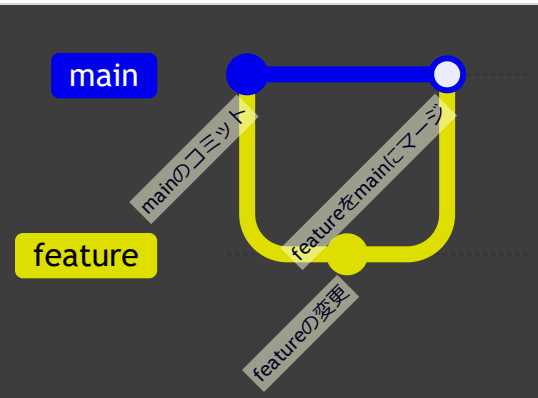
```
git branch -d feature
```



ブランチのマージ

既に一般的なビジネス用語化している気がしますが、ここでの意味は、複数のブランチを一つに統合する操作です。通常、開発作業を別々のブランチで行い、作業が完了した後にその変更を一つのブランチに統合するために使用します。マージを行うことで、異なるブランチでの作業内容を一つにまとめ、最終的にmainブランチに反映させます。

```
git checkout main
git merge feature-x
```



リモートリポジトリとのやり取り

ここでエラーとかあるある

リモートへのプッシュ

プッシュは、ローカルリポジトリで行った変更をリモートリポジトリに反映させる操作です。これにより、他の開発者と変更内容を共有できるようになります。プッシュを行うことで、リモートリポジトリに自分の変更が加わり、チーム全体でその変更が利用できるようになります。

```
git push origin feature-x
```

プルとフェッチ

複数人で作業を行っていくと当然リモートリポジトリとローカルで進み具合に差が出ます。
他の人のプッシュが作業中のリポジトリで先行した場合とかですね。

この場合ローカルリポジトリ側がリモートと合っていないため、プッシュを行おうとしてもエラーになります。まずはリモートの最新の変更をローカルに取り込んでからプッシュを行う必要があります。

このリモートリポジトリの変更を取り込むために`git pull`または`git fetch`を利用します。

git pull / git fetch

`git pull` と `git fetch` の違い

コマンド	動作
git pull	git fetch + git merge（自動的に変更を統合）
git fetch	リモートの変更を取得するが、ローカルには適用しない

ということで `git fetch` から説明します！

git fetch

どのような動作をしているのか？

リモートリポジトリの変更点`C`を`origin/main`として取得しますがローカルのmainブランチへの適用までは行わない。

git pull

どのような動作をしているのか？

`git fetch`してリモートリポジトリの変更を取得した後にその変更点を`git merge`でmainブランチに取り込みまで行います。

注意点としてはローカル側で想定外の変更点が取り込まれる可能性があります。

ここまでで大体OK!!

そこそこちゃんと使えます！！！

他のユーザーと協力する。

GitHubでコードで高度な会話をしよう。

フォークからプルリクまで

GitHubでのフォークからプルリクエストの概要

GitHubで他の人のリポジトリをForkし、修正を加えてPull Requestを送る流れを説明します。
プルリクってもうエンジニアの一般用語ですよね？ね？？

- **Fork**: 他人のリポジトリを自分のアカウントにコピー
- **Clone**: Forkしたリポジトリをローカルにダウンロード
- **修正 & Push**: 変更を加え、自分のリポジトリに反映
- **Pull Request**: 親リポジトリに変更の提案を送る

GitHubでのフォークとは？

フォークとは、他人のリポジトリをコピーして自分のGitHubアカウントに作成する機能です。

- 他の人のプロジェクトを編集・改良できる
- オリジナルのリポジトリに影響を与えない
- Pull Requestを通じて変更を提案できる

ローカルで修正、変更をプッシュ

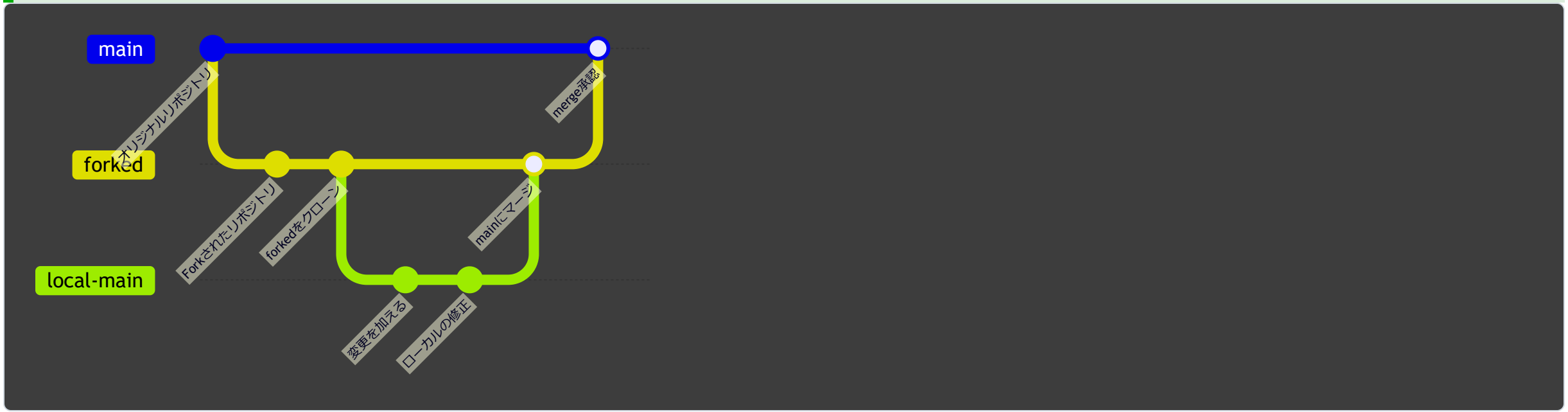
フォークしてきたリポジトリも修正変更を行う場合は基本ローカルにクローンして行いましょう。

Pull Request を作成

プルリクってよく聞くけど何？

- 修正内容を親リポジトリの管理者に送るための仕組み
- 承認されるとオリジナルのリポジトリにマージされる

ちなみにGitHubではプルリクですが、GitLabではマージリクエストという名前になります。
公式ドキュメントはこちらから



まとめ

改めてGitHubの開発まとめ

1. Fork でリポジトリをコピー
2. Clone でローカルに取得
3. 修正 & Push で変更を反映
4. Pull Request で変更を提案

Appendix

Git Command Graphical Cheat Sheet (ver 1.2) Edit by @kozzy0919

git って？

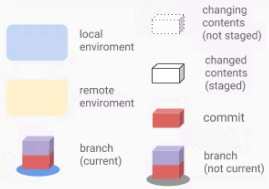
Git とはバージョン管理システムの一つ

以下のような特徴がある [1]

- ノンリニア開発のための分岐システム
- プロジェクト変更履歴を管理できる
- CLI / GUI で扱うことができる
- コミット済、修正済、ステージ済の3つの状態を持つ

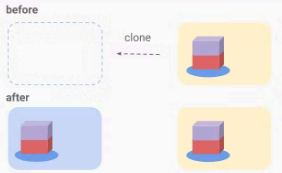
ここではgit でよく使われるコマンドの概念をイラスト化しながら簡単に解説します

イラストの注釈



※branch における commit は上にあるものが新しい commit とする

git clone（プロジェクトを手元にコピーする）

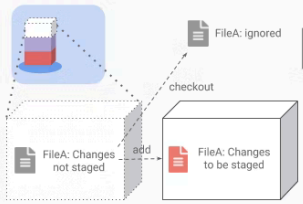



```
$ git clone {repository_url}
```

git clone はリモートからリポジトリをクローンするために行います。

言い換えると、自分の環境外からプロジェクトを自分の手元にコピーします。

git add / git checkout（ファイルの変更を確定する、変更前に戻す）




```
$ git add {file_name} # 確定
$ git checkout {file_name} # 無視
```

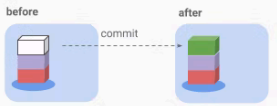
git add は git clone したプロジェクト上のファイルに変更を加えた際にその変更を確定するため行います。git checkout は変更を加えたファイルに対して、その変更を元に戻すために行います。

git status（変更中、変更済ファイルを確認する）


```
$ git status
```

git status はリポジトリ内にて git add したファイル群とそうでないファイル群の状況ををリスト表示するために行います。

git commit（ファイルの変更群をひとまとめにする）




```
$ git commit -m {commit_message}
```

git commit は git add したファイル群を、commit message と呼ばれる コメントをつけた上でひとまとまりの変更群として反映させるために行います。

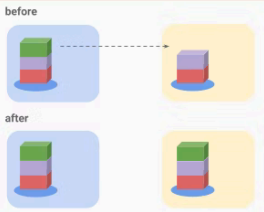
※ 表現上、説明を簡略化しております。このメモの内容の誤りなどによる損害が発生してもいかなる責任を負いません

git mv / rm（ファイルを変更、削除する）

git mv は ファイル名を変更するために行います。
git rm は ファイルを削除するために行います。また、--cached オプションを付けることで、ファイルを消さずに git の管理対象から外すことができます。

```
$ git mv {file_name} # 移動
$ git rm {file_name} # 削除
$ git rm --cached {file_name} # git の管理対象から外す
```

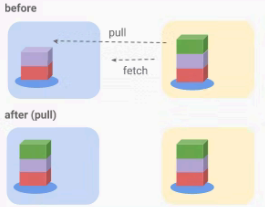
git push（ブランチの状態をリモートに反映させる）




```
$ git push
```

git push は ローカルブランチのコミット状態をリモートに反映させるために行います。

git fetch / git pull（リモートの状態をローカルに取得、反映させる）

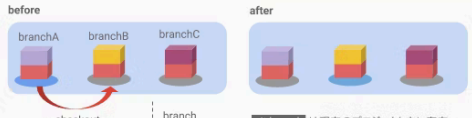



```
$ git fetch # 変更情報取得
$ git pull # 変更反映
```

git fetch は clone した元に変更があった場合に、その情報のみを取得するために行うコマンドです。その変更は反映されません。

git pull は clone した元に変更があった場合に、その情報を取得してその変更を作業しているローカルブロジェクトに取り込むために行うコマンドです。

git branch / git checkout（ブランチを確認する・切り替える）




```
$ git branch # 確認
* branchA
  branchB
  branchC

$ git checkout branchB # 切り替え
```

git branch は現在のプロジェクト内に存在する全てのブランチをリスト表示させる、現在いるブランチを確認するために行うコマンドです。

git checkout は変更を加えたファイルに対して、その変更を無視する、という使い方以外にも、作業対象とするブランチを変更するためにも行われます。

git remote（変更反映先の確認、追加を行う）

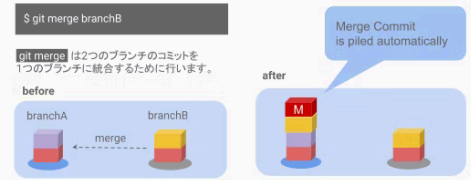
```
$ git remote -v # 確認
$ git remote add {repository_url} # 追加
```

git remote はリモートリポジトリの確認や追加を行うコマンドです。

git merge（2つのブランチを統合する）

```
$ git merge branchB
```

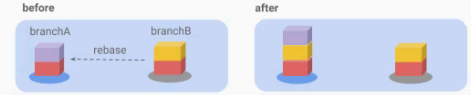
git merge は2つのブランチのコミットを1つのブランチに統合するために行います。



git rebase（2つのブランチのコミットを並び替えて1つにする）

```
$ git rebase branchB
```

git rebase は2つのブランチのコミットを並び替えて1つのブランチにするために行います。

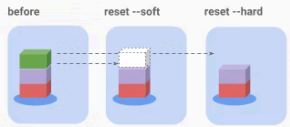


git cherry-pick（コミットを指定して、ブランチに追加する）

```
$ git cherry-pick XXX
```

git cherry-pick はコミットを指定して、現在いるブランチに追加するために行います。git log / reflog などによって、指定するための コミットID を確認できます。

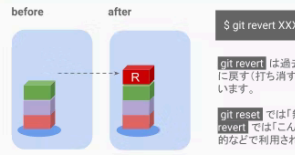
git reset（過去のコミットを巻き戻す、削除する）




```
$ git reset --soft HEAD^
$ git reset --hard HEAD^
```

git reset は過去のコミットに巻き戻す、もしくは削除するために行います。また、HEAD^ で直前のコミットを指します。

git revert（過去のコミットを打ち消すコミットを追加する）

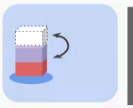



```
$ git revert XXX
```

git revert は過去のコミットで行なった変更内容を元に戻す（打ち消す）ためのコミットを追加するために行います。

git reset では「無かったこと」にするのに対して git revert では「こんな誤りもあった」というログを残す目的などで利用されます

git diff（変更差分を確認する）



```
$ git diff # 差分確認

$ git diff HEAD^ # 直前のコミットにおける差分確認
```

git diff は変更差分を確認するために行います。デフォルトでは変更中のファイル群の差分を表示します。また、HEAD^ での直前のコミットを指定することでそれとの変更点を確認することができます。

git tag（ブランチにタグを付与する）

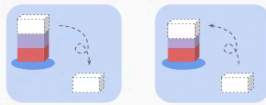
```
$ git tag
v0.1
v0.2
v0.3
v1.0

$git tag -a v1.1 -m {tag_message}
```

git tag はコミットにタグを付与するために行います。

リリースのバージョンなどをタグにすることなどが多いようです。

git stash / git stash pop（変更中のファイルを一時的に退避させる）




```
$ git stash # 退避
$ git stash pop # 復旧
```

git stash は変更中のファイルを一時的に退避させるために行い、git stash pop で退避させたファイルを元に戻すために行います。

git reflog（過去の参照ログを確認する）

```
$ git reflog

XXX HEAD@{0}: checkout: moving
XXX HEAD@{1}: checkout: moving
XXX HEAD@{2}: commit: Fix bug
YYY HEAD@{3}: commit: Fix bug
```

git reflog は自分のリポジトリで何をしたのかを示す情報をリスト表示させるために行います。

git reset --hard HEAD@{1} とすれば、その時点の状態に戻すことができます。

git log（過去のコミットを確認する）

```
$ git log

commit XXX
Author: kozzy <xxx@gmail.com>
Date:  Tue Feb 26 10:01:09 2019 +0900

    Commit Message #1
```

git log は過去のコミットの情報を詳細にリスト表示させるために行います。

git cherry-pick に用いる commit ID などはこのコマンドから調べられます。

git config（git の設定を行う）

git config は git を利用する上での設定及び、それら確認をするために行います。
オプションに --system を付けると全ユーザーの全リポジトリ、--global がログイン中ユーザの全リポジトリ、--local を付けると現在いるリポジトリのみに設定が反映されるようになります。

```
$ git config user.email # 確認

$ git config --global user.email
hoge@hoge.com # 確認
```

参考元

[1] Git - Documentation, https://git-scm.com/doc (参照 2019-02-27)

[2] 逆引きGit _ サルでもわかるGit入門【プロジェクト管理ツール-Backlog】, https://backlog.com/ja/git-tutorial/reference/ (参照 2019-02-27)

edited by kozzy (@kozzy0919)

Appendix

おすすめサイト！

https://backlog.com/ja/git-tutorial/