Gitブランチフロー規約

フューチャー株式会社

Table of Contents

本ガイドラインは、世の中のシステム開発プロジェクトのために無償で提供する。  
ただし、掲載内容および利用に際して発生した問題、それに伴う損害については、フューチャー株式会社（以下、フューチャー）は一切の責務を負わないものとする。  
また、掲載している情報は予告なく変更する場合があるため、あらかじめご了承いただきたい。

::: warning 免責事項: 有志で作成したドキュメントである

* フューチャーには多様なプロジェクトが存在し、それぞれの状況に合わせて工夫された開発プロセスや高度な開発支援環境が存在する。本ガイドラインはフューチャーの全ての部署／プロジェクトで適用されているわけではなく、有志が観点を持ち寄って新たに整理したものである
* 相容れない部分があればその領域を書き換えて利用することを想定している
  + プロジェクト固有の背景や要件への配慮は、ガイドライン利用者が最終的に判断すること
* 本ガイドラインに必ず従うことは求めておらず、設計案の提示と、それらの評価観点を利用者に提供することを主目的としている

:::

# はじめに

本規約はGitブランチ管理の標準的な運用ルールをまとめている。以下の想定で作成されているため留意すること。

* GitHub ／ GitLab の利用
* トランクベース開発（フィーチャーフラグ）を **採用しない**
* ライブラリではなく、アプリケーション（CLIツール、Webアプリケーションなどの）開発で利用する

# 基本方針

一般的なGitブランチ運用のプラクティスに従い、本規約も以下の方針に則る。

* すべての機能開発や不具合修正に、featureブランチを使用する
* プルリクエストを経由してfeatureブランチの修正内容をマージする
* 永続ブランチは各環境にデプロイ可能となるよう整合性を保つ

# ブランチの種類

本規約で想定する、ブランチの種類とその役割を説明する。

| ブランチ名称 | 役割 | ライフサイクル | 派生元ブランチ | 命名規則 | 直プッシュ |
| --- | --- | --- | --- | --- | --- |
| main | プロダクション環境との同期 | 永続的 | - | main 固定 | ❌️ |
| feature | 特定機能の追加/変更 | 短命 | main／develop | feature/${チケット番号}: 詳細は[featureブランチ](#featureブランチ) を参照 | ✅️※1 |
| develop | 開発の大元 | 永続的 | main | develop 固定。複数必要な場合は develop2 と連番にする | ❌️ |
| release | リリース作業用途 | 短命 | develop | release/${yyyymmdd} や release/${リリースバージョン} など | ❌️ |
| hotfix | mainブランチに対する即時修正 | 短命 | main | hotfix/${チケット番号}: featureブランチに準じる | ✅️ |
| topic | 複数人での機能開発用途 | 短命 | feature | topic/${チケット番号}: featureブランチに準じる | ✅️ |

※1: topicブランチを利用する場合は、派生させたfeatureブランチへの直プッシュはNGとなる

## mainブランチ

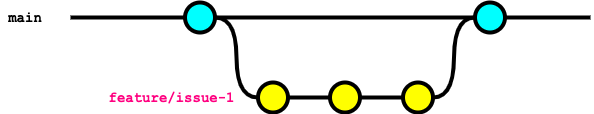
Gitリポジトリを新規作成するとデフォルトで作成されるブランチ。masterからmainに改名された経緯を持つ[[1]](#footnote-22)。

マージ毎にプロダクション環境へデプロイし同期を取る。

## featureブランチ

機能追加や変更を行うブランチで、主な特徴は以下である。

* ひとつの変更に対してひとつのfeatureブランチを作成し、作業完了後に削除するため、開発中で最も使われる短命なブランチである
* 基本的に1人の開発者のみが利用する



feature branch

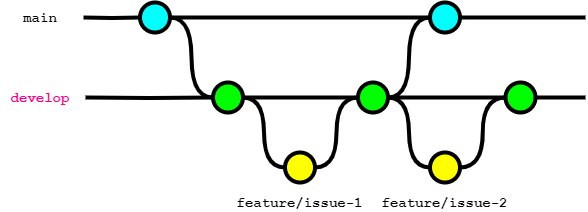
以下の命名に従う。

* feature/ のプレフィックスを付ける
* 課題管理システムと紐付けられるようなブランチ名にする

# OK（課題管理システムの課題番号をブランチ名に利用）  
feature/#12345  
  
# OK（GitHub Issue や JIRA や Backlog のプロジェクトIDをブランチ名に利用）  
feature/<PROJECTID>-9403  
  
# NG（プレフィックスが無い）  
fixtypo

## developブランチ

開発の中心となるブランチである。

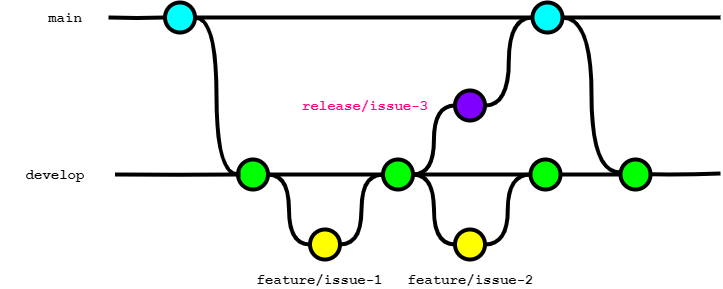


develop branch

## releaseブランチ

リリースするために使用するブランチで、主な特徴は以下である。

* リリース前の検証を開発と並行して実施する場合に利用する
* releaseブランチではバグ修正、ドキュメント生成、その他のリリースに伴うタスクのみを実施する
* mainブランチのマージコミットにリリースタグを打ち、mainブランチをdevelopブランチへマージ後、releaseブランチを削除する

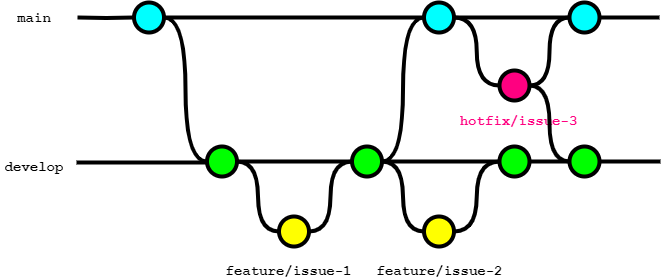


release branch

## hotfixブランチ

本番リリースに対して迅速にパッチを当てて修正する場合に使用するブランチで、主な特徴は以下である。

* 修正が完了するとmainとdevelopの両方(あるいは進行中のreleaseブランチ)にマージされる
* main／developブランチがあると必要になる可能性がある。main／featureブランチのみの運用では必須ではない（管理上の目的でfeatureとhotfixを分けることはあり得る）

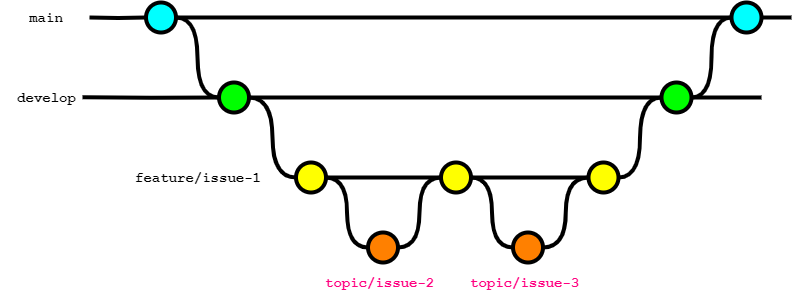


hotfix branch

## topicブランチ

featureブランチで実現する機能を複数人で開発する場合に使用するブランチである。

* topicブランチが必要なケースでは、featureブランチへの直接プッシュを行ってはならない
* GitHub Flowではfeatureブランチのことをtopicブランチと呼称する場合があるが、本規約ではfeatureブランチから派生するブランチをtopicブランチと定義する



topic branch

# ブランチ戦略の選定

ブランチ戦略は以下の方針で選定する。

* できるかぎりシンプルなモデルを選択し、運用コストを下げる
* プロジェクトのフェーズや体制に応じて、変更を許容する

有名なブランチ戦略として以下がある。

* [git-flow](https://nvie.com/posts/a-successful-git-branching-model/)
* [GitHub flow](https://docs.github.com/ja/get-started/using-github/github-flow)
* [GitLab Flow](https://docs.gitlab.co.jp/ee/topics/gitlab_flow.html)

本規約で推奨するブランチ戦略は次の2パターンであり、これをベースとして選択する。

| 名称 | 利用ブランチ | デフォルトブランチ | リリース作業ブランチ | 備考 |
| --- | --- | --- | --- | --- |
| Lite GitLab Flow※1 | maindevelopfeaturetopic hotfix | develop | develop | ・GitLab Flowからreleaseブランチを除いたパターン・リリース作業時にdevelopマージを止められる場合に利用する |
| GitLab Flow | maindeveloprelease featuretopic hotfix※2 | develop | release | ・リリース作業と開発作業が並行して行う必要があるか、断面を指定して複数テスト環境にデプロイしたい場合に利用する |

* ※1: 特定の呼称はないためLite GitLab Flowと命名する
* ※2: 本規約では、本来のGitLab Flowの呼称である productionをmain、pre productionをreleaseに言い換えている

# ブランチ戦略とデプロイメント環境

各ブランチ戦略ごとに、デプロイメント環境に対応するブランチを整理する。プロダクション環境リリース前には、mainブランチでタグを打つこととする。

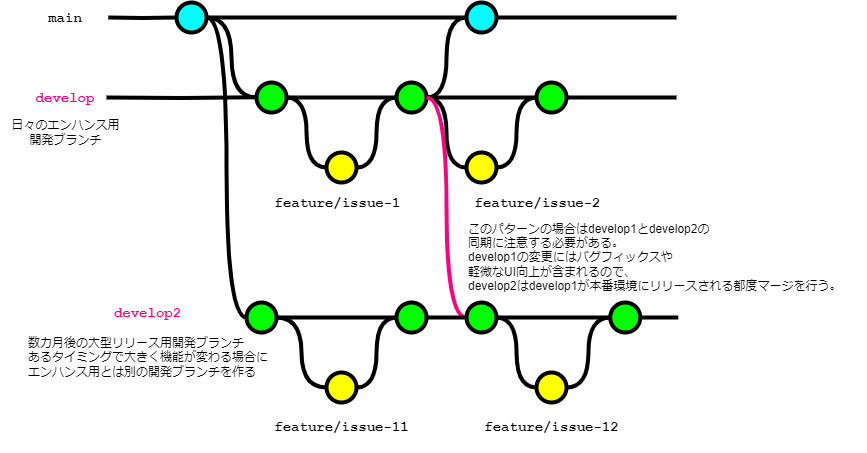
| 名称 | 開発環境 | ステージング環境 | プロダクション環境 | 備考 |
| --- | --- | --- | --- | --- |
| Lite GitLab Flow | develop | develop | main | ・開発環境へはdevelopマージをトリガーにCI/CDでデプロイを推奨する・開発環境へのデプロイ漏れを防ぐため定期的にCI/CDでdevelop断面をリリースすることを推奨する・動作確認など理由がある場合はfeatureブランチから直接開発環境へのデプロイも許容する・ステージング環境は日次など定期的なCI/CDによるデプロイを推奨する |
| GitLab Flow | develop | release | main | ・開発環境へはdevelopマージをトリガーにCI/CDでデプロイを推奨する・検証期間が長引きそうな場合は、PRレビュー承認後にfeatureブランチから開発環境へのデプロイを許容する |

# ブランチ戦略の拡張

次のような要件があった場合には、ベースとなるブランチ戦略を拡張する必要がある。

1. developブランチを複数作成する場合
2. 過去バージョンをサポートする場合

## developブランチを複数作成する場合



multi develop branch

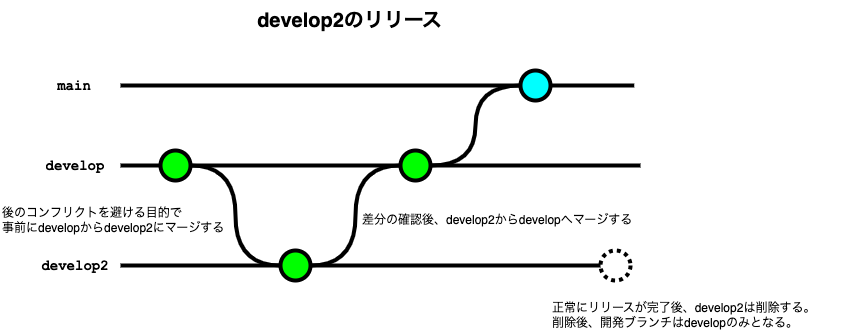
日々のエンハンス開発と並行して、数カ月後に大型リリースを行いたい場合がある。このときは複数リリースバージョンを並行して開発するため、 develop、develop2 といった複数のdevelopブランチを作る必要がある。

概要:

* develop の変更にはバグフィックスや軽微なUI向上が含まれ、日次／週次などの頻度でプロダクション環境へリリースされる
* develop2 はdevelop ブランチの変更をすべて取り込んだ上で、大型機能の準備を行う必要がある

develop2 同期の注意点:

* リベースすると develop2 を元にfeatureブランチを作成して開発している開発者が混乱することになるため、マージコミットを用いる
* 誤操作を避ける目的でcherry-pickは行わない
* develop2 への同期は、 develop -> main ブランチに反映されるタイミングで同期を行う（これにより、品質保証済みの変更のみ取り入れることができる）



release multi develop branch

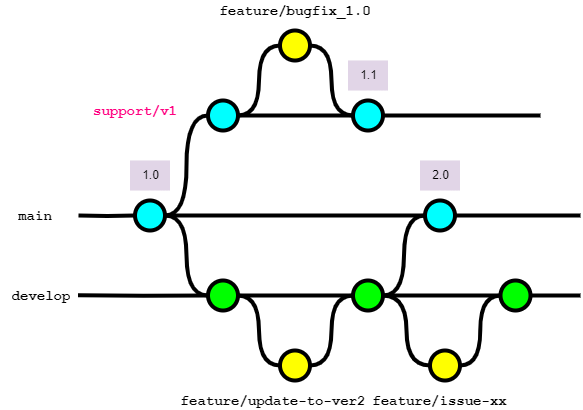
### develop2のリリース手順

1. developからdevelop2へマージコミットする（2でコンフリクトが起こらないよう、前準備の意味合いで実施する。）
2. develop2からdevelopにマージを行い、その後は通常のリリースフローに従う
3. 問題なくリリースが完了し次第、develop2を削除する

developからdevelop2へマージ後、develop2をmainブランチに反映させる手順も考えられるが、develop2からdevelopへのマージとすると以下のメリットがある。

* プロダクション環境（=develop）との差分を把握することができる
* より一般的な名称である develop ブランチのみ残るため、新規参画者フレンドリーである

## 過去バージョンをサポートする場合



multi version branch

（社内外の）ライブラリでインターフェースの大型改善や仕様変更を受けて、メジャーバージョンを1→2に上げることがある。この時に過去バージョンもサポートする必要があると、バージョン別にsupportブランチを作成する。

概要:

* メインの更新はversion2（mainブランチ）に対して行っていくが、version1の利用ユーザーが存在する場合、バグfixやセキュリティアップデートを並行して行う
  + version1を示すブランチ（support/v1）を別途作成、そのブランチからfeatureブランチを作成する
* featureブランチのマージ後、マイナーバージョン（あるいはパッチバージョン）を上げたタグをコミットし、リリースする
  + ※この例ではversion1とversion2が別リソースとして動いていることを前提としている。同一リソースで複数バージョンが稼働する場合はversion2のブランチで対応を行う必要がある。

# マージ戦略の選定

マージ戦略とは、複数のブランチ間で生じた変更の取り込み方針を指す。

具体的には次の3ケースそれぞれで、「マージコミット」 「リベース」 「スカッシュマージ」のどれを採用するか判断する。

1. developブランチからfeatureブランチへ変更を取り込む
2. featureブランチからdevelopブランチへ変更を取り込む
3. 永続ブランチ間で変更を取り込む

以下に影響を与えるため、Gitの利用開始前に決めチームで統制を図ることが重要である。

* プロジェクトのコミット履歴の管理
* 開発プロセスの円滑な進行
* 最終的なソフトウェア品質

## developブランチからfeatureブランチへ変更を取り込む

featureブランチでの作業中に、developブランチが更新された場合、品質保証の観点でdevelopブランチの変更をfeatureブランチに取り込んだ上で、テストなどの検証作業を行う必要がある。

developブランチの変更をfeatureブランチに取り込む方法には、下表の2つの方法が存在する。

機能ブランチに対して開発ブランチの変更を取り込む方法は「マージ」 「リベース」2つの方法が考えられる。スカッシュマージはこのケースでは選択できない。

| 1. マージコミット | 2. リベース |
| --- | --- |
|  |  |
| git fetch & git merge（≒ git pull）。マージコミットが作成される | git fetch & git rebase（≒ git pull --rebase）。最新の開発ブランチの先頭から新たにコミットを作り直され、マージコミットは作成されない |

本規約の推奨は「1. マージコミット」である。

理由は次の通り。

* リベース方式で設定すべき rerere.enabled オプションを有効にしても、1度解消したコンフリクトの再対応をゼロにすることはできないため
* マージコミットが作成され履歴が複雑になるが、1度解消したコンフリクトの再対応がゼロにできる点を優先する

::: tip リベース方式を採用する場合

もし、リベース方式を採用する場合は、以下の設定を行う。

1. git pull 時の挙動がリベースになるよう git config pull.rebase true を実行する
2. developブランチの変更を取り込む場合、同じコンフリクトの解消を何度も求められることを解消するため、git config rerere.enabled true を実行する

マージによる変更の取り込みが既存のブランチを変更しないのに対し、リベースは全く新しい（元のコミットIDとは別のコミットIDで）コミットを作成するため、次の1点に注意すること。

1. リモートにプッシュ済のブランチがあり、developブランチからさらに変更をリベースで取り込んだ場合、強制プッシュ（Force Push）が必要になる
   * git push origin HEAD --force-with-lease --force-if-includes とすることで、意図せずリモートブランチの変更を上書きしないようにする
     + --force-with-lease: ローカルのリモート追跡ブランチの ref とリモートの ref を比較し、ローカルの状態が最新でない場合（プッシュ先のリモートブランチに変更が入ったが、ローカルで git fetch していない場合）は、プッシュに失敗する。逆にいうと、プッシュ前に git fetch を実行済みの場合は、リモートの変更を上書きする形で強制プッシュができてしまうため、これを防ぐには --force-if-includes フラグを併用する
     + --force-if-includes: リモート追跡ブランチの変更がローカルに全て取り込まれていない場合は、プッシュに失敗する。これにより意図せず他の人のコミットを上書きすることを防ぎつつ、必要な変更を強制的にプッシュすることができる

:::

::: tip 強制プッシュでレビューコメントは消えるのか？

強制プッシュすることにより、レビューコメントが消えてしまわないかという懸念を聞くことがある。2024年7月に実施した調査結果では問題なかった。特にリベース方式では重要である。

* a.履歴保持: 強制プッシュを行い、GitHub投稿したレビューコメントが履歴として何かしらのページで取得できるかどうか。GitHubではConversationタブで確認
* b.行単位の紐づけ（該当行の変更なし）: レビューコメントが付けられた行とは別の変更を行い、強制プッシュしたときにレビューコメントの紐づけが残るかどうか。GitHubではFile changedタブで確認
* c.行単位の紐づけ（該当行の変更あり）: レビューコメントで付けられた行を修正し、強制プッシュ時の挙動。レビュー対応をしたとみなしレビューコメントのひも付きは解除されているべきである。GitHubではFile changedタブで確認

| サービス | a.履歴保持 | b.行単位の紐づけ（該当行の変更なし） | c.行単位の紐づけ（該当行の変更あり） |
| --- | --- | --- | --- |
| GitHub | 残る | 残る | 消える |
| GitLab | 残る | 残る | 消える |

:::

### プルリクエスト作成前にアップストリームをプルする

featureブランチの開発が終わり、プルリクエストを作成する際には、改めてアップストリーム（developブランチ）の変更をfeatureブランチに取り込み、差分が無いことを確認すべきである。

理由は次の通り。

* レビュアーの負荷軽減のため
  + レビュアーがプルリクエストの差分以外の部分を参照した際に、それが古いバージョンであると、誤指摘、混乱してしまうなどの懸念がある
* マージ後のdevelopブランチでテスト失敗するリスクを減らすため
  + コンフリクトせずにマージ可能だったとしても、何かしらの依存関係や整合性が狂い、マージ後のテストが失敗する可能性がある

### プルリクエストのレビュー依頼までにどこまでテストしておくべきか

本規約で推奨する Lite GitLab Flow GitLab Flow ともに、開発環境へはdevelopマージをトリガーにCI/CDでデプロイを推奨している。

そのため、プルリクエスト作成時点では開発環境（≒AWSなどクラウド環境の想定）へのデプロイ＋動作検証は不要である。

ローカルでの開発のみで品質担保が難しく手戻りが多い場合は、サンドボックス環境や開発環境にfeatureブランチからデプロイして動作検証する作業が必要になる。開発環境を共有する場合は、デプロイタイミングの制御がチーム内で必要になるため、運用ルールを検討する必要がある。

### Terraformはレビュー依頼時点でどこまで確認しておくべきか

Terraformはplanが成功しても、applyが失敗することは多々あり（サブネットが足りなかった、force\_destroy=trueを明示的な設定が必要だったなど）、レビューでの見極めは難しいことが多い。そのため、applyをどのタイミングで実施するかがチームの生産性の鍵となる。

大別すると以下の3方式が存在する

1. マージ後にapply
   * PR -> CI（planを含む） -> レビュー -> developマージ -> apply(CI)
2. Approve後にapply
   * PR -> CI（planを含む） -> レビュー -> apply -> developマージ -> apply(CI)
3. レビュー依頼前にapply
   * apply -> PR -> CI(plan含む) -> レビュー -> developマージ -> apply(CI)

それぞれの特徴を下表にまとめる。

| 観点 | ①マージ後にapply | ②Approve後にapply | ③レビュー依頼前にapply |
| --- | --- | --- | --- |
| 説明 | developブランチにマージ後にapply。アプリコードと同じメンタルモデルを共有可能 | レビュアー承認後にapply。featureブランチからapplyするため、あるべき姿からは外れる | レビュー依頼前にapplyで成功したことを確認する方式 |
| developブランチ品質 | ❌️一時的にapplyが失敗するコードが混入するリスク | ✅️apply可能なコードのみに保つことができる | ✅️apply可能なコードのみに保つことができる |
| レビュー負荷 | ❌️applyの成否は不明なので心理的負荷あり | ❌️applyの成否は不明なので心理的負荷あり | ✅️applyが成功している前提で対応可能。apply結果をコンソールからも確認可能 |
| apply失敗時のコスト | ❌️再度PRを作る必要があり手間 | ✅️同一PRを流用できる | ✅️apply成功後にPR作成が可能 |
| PRのトレーサビリティ | ❌️PRが割れると面倒 | ✅️同一PRである | ✅️同一PRである |
| 環境のバッティング | ✅️ない | ⚠️Approveからdevelopマージまでの間に、他メンバーの作業と重複するとややこしい | ❌️作業調整が必要 |
| ガバナンス | ✅️applyをCIのみに絞るなど自動化と相性が良い | ⚠️レビュアー承認後のコードのみapply対象とできる | ❌️ノーレビューのインフラ変更を適用するため、初学者が多いチームには適用が難しい |
| 結論 | applyの成功率が高く維持できる場合に有効 | applyの成功率が低い場合に有効 | 少数精鋭の場合に採用可能な、上級者向けの方式 |

本規約の推奨は以下。

* 新規参画者が多く統制を取りたい場合や、applyの成功率が高く維持できる場合は①を選択
* ある程度インフラメンバーが絞れ、かつapplyの失敗率が高くレビュー負荷が高くなってしまう懸念がある場合は②を選択
* インフラメンバーが少数精鋭（通常、同時の作業はほぼ発生しない）の場合は必要に応じて、②をベースにしながら③を取り入れて生産性を上げる

## featureブランチからdevelopブランチへ変更を取り込む

プルリクエスト（以下、PR）を経由して、開発が完了したfeatureブランチをメインのdevelopブランチに取り込むためには、GitHub（GitLab）上でPRを経由する運用を行う。

developブランチにfeatureブランチの変更を取り込む方法は下表のように3パターン存在する。

|  | 1.マージコミット | 2.リベース | 3.スカッシュマージ |
| --- | --- | --- | --- |
| 名称 | Create a merge commit | Rebase and merge | Squash and merge |
| 流れ |  |  |  |
| 説明 | git merge --no-ff で変更を取り込む | featureブランチを最新のdevelopブランチにリベースし、git merge --ff で変更を取り込む | git merge --squash で変更を取り込む |
| 特徴 | developブランチにマージコミットが作成される | マージコミットは作成されず、履歴が一直線になる | featureブランチで行った変更YとZを1つにまとめたコミットがdevelopブランチに作成される |

::: tip GitLabを利用する場合

GitLabでも開発ブランチに機能ブランチの変更を取り込む方法は3種類ある。

ただし、マージリクエスト上のオプションによってコミット履歴が変わる点が注意である。

|  | 1. Merge commit | 2. Merge commit with semi-linear history | 3. Fast-forward merge |
| --- | --- | --- | --- |
| 流れ |  | 省略 | 省略 |
| 説明 | GitHubにおける Create a merge commit と同様のマージ方法 | Merge commit と同じコマンドを使用して、機能ブランチの変更を取り込む方法 | GitHubにおける Rebase and merge と同様のマージ方法 |
| 注意 | Squash commits を選択してマージした場合、squash commit と merge commit の2つのコミットが作成される | ソースブランチがターゲットブランチより古い場合はリベースしないとマージできない。 | マージリクエスト上で Squash commits を選択してマージした場合、GitHubにおける Squash and merge と同様のマージ方法になる（※補足1） |

（※補足1）マージ方法で Merge commit を選択して、マージリクエスト上で Squash commits オプションを選択してマージした場合は以下と同義である

git checkout `git merge-base feature/A develop`  
git merge --squash feature/A  
SOURCE\_SHA=`git rev-parse HEAD`  
git checkout develop  
git merge --no-ff $SOURCE\_SHA

:::

本規約の推奨は、「スカッシュマージ」による方法である。

理由は次の通り。

* featureブランチのコミットログが、汚れることは許容したいため
* developブランチの履歴をクリーンに保てるため
* PRをよりシンプルに保つインセンティブとしたいため（単一のコミットメッセージで表現できる程度の方がレビューコストも小さいため）

「スカッシュマージ」を行うと、変更元のfeatureブランチのコミットをまとめたコミットが新たに作成されるため、元のfeatureブランチを再利用しPRを作成するとコンフリクトが発生する。そのためマージ後はリモート/ローカルの双方で速やかにfeatureブランチを削除させるため、以下の設定を加える。

* マージ後にfeatureブランチを自動削除する設定
  + リモート側: GitHubでは「Automatically delete head branches」を選択することで、マージ後に自動でブランチの削除が行われる（GitLabではプロジェクト設定で「Enable “Delete source branch” option by default」を選択する）
  + ローカル側: git config --global fetch.prune true: リモート側で削除されたブランチをローカル側でも削除する

「スカッシュマージ」による変更の取り込みを行う場合、次の2点に注意すること。

1. 部分的なコミットの取り消しができない
   * 履歴上は1つのコミットになるため、マージ後に一部の変更だけの取り消しが不可能。そのためPRをなるべく小さなまとまりにする
2. Authorが失われる
   * featureブランチにコミットを行った人がAuthorになるのではなく、「スカッシュマージ」を行った人がAuthorになる。OSS開発を行う場合など、厳密にコントリビューションを管理する必要がある場合は注意する
   * GitHubでは「スカッシュマージ」を行う場合、デフォルトでコミットメッセージに co-authored-by トレーラーが追加され、1つのコミットが複数の作成者に帰属するようにするようになっている[[2]](#footnote-68)。この記述は削除しないようにする

### マージはだれが行うべきか

プルリクエストの承認（Approve）をもらった後、マージはレビュアー／レビュイーのどちらが行うべきか議論になる場合がある。

| 観点 | レビュアー派 | レビュイー派 |
| --- | --- | --- |
| 説明 | 開発者の責務が、developブランチにマージするまでという役割分担の場合に有効 | 各開発者がその機能のリリースについて責任を負うモデルの場合に有効 |
| 生産性 | ⚠️レビュアーがブロッキングになりがち | ✅️高い。コメントはあるがApproveしたので、適時対応してマージして、といった運用が可能 |
| 統制 | ✅️レビュアーが管理しやすい | ✅️メンバーの自主性に依存 |
| 要求スキル | ✅️低い。中央で統制を行いやすい | ⚠️開発メンバーの練度が求められる |

上記にあるように、そのプルリクエストで実装した機能を、本番環境にデリバリーする責務をどちらに持たせるかという観点で、意思決定することが多い。

本規約の推奨は以下。

* プロダクトオーナー（業務側）などでリリースタイミングを完全にコントロールしたいといった分業制を取る場合は、レビュアーがマージする
* 各開発者により自律性を持たせ、アジャイル的に生産性を重視するのであれば、レビュイーがマージする

## 永続ブランチ間で変更を取り込む

永続ブランチ同士の変更を取り込むケースとして、develop ブランチを main ブランチや releaseブランチにマージするといった場合がある。

ブランチ間の同期が取れないため「リベース」「スカッシュマージ」は選択できないため、「マージコミット」を採用する。

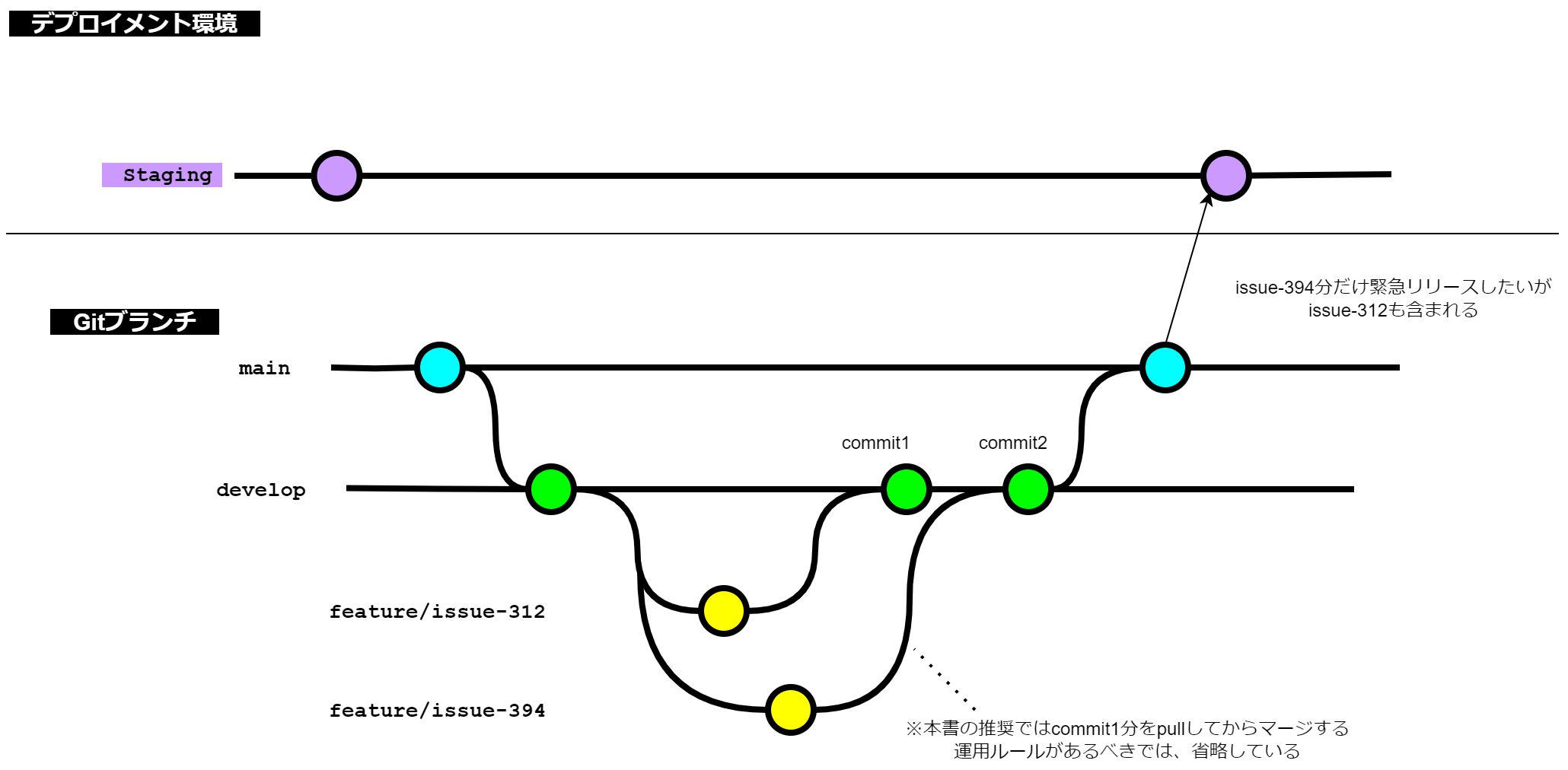
# ブランチ運用アンチパターン

ブランチ運用でよく課題に上がるパターンとその対応を紹介する。

## 追い抜きリリース

以下のような状況とする。

* 2つのチケット（issue-312、issue-394とする）があり、どちらも同じファイルの修正を含む
* 先にissue-312がdevelopにマージされ、その後に着手されたissue-394がマージされた
* 以下のような条件があるため、issue-394分を先にリリースしたい
  + issue-312のリリースは業務上の合意が得られていない（エンドユーザ操作に影響があるため、事前告知した日時でリリースしたいなど）
  + issue-394は不具合修正であり業務上の優先度が高いため、なるべく早くリリースしたい



同一ファイルを複数

よく陥りがちな対策としては次の2点が考えられる。

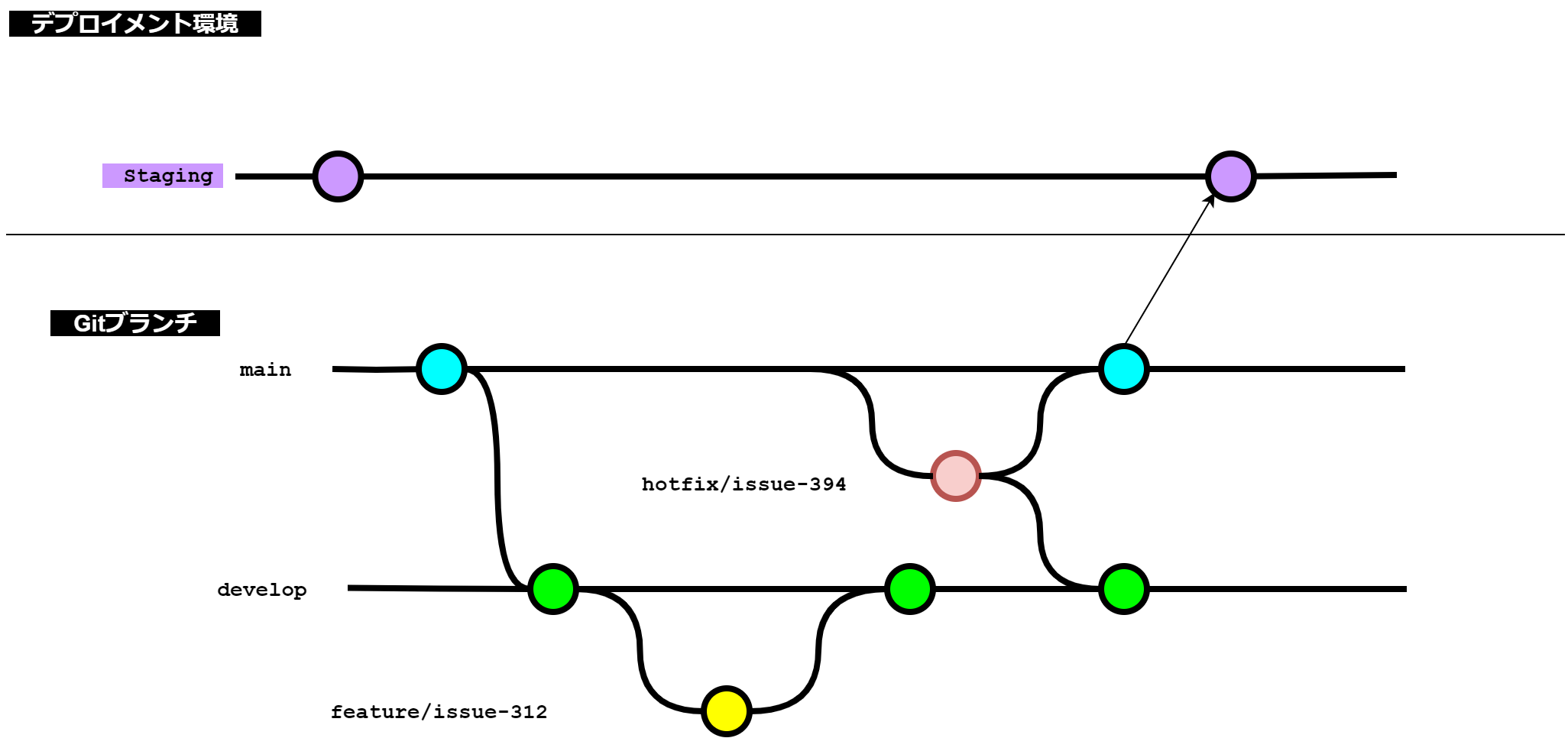
1. issue-312をリバートする
2. issue-394のコミットのみをcherry pick してmainブランチにマージする

1のリバートはGitHubの機能で提供されていることもあり簡単に行えるが、手戻りであることは間違いないし、コミットの履歴が汚れるため、保守運用の視点ではマイナスである。2のcherry pickは操作、管理ともに煩雑でミスが出やすいという課題がある。

処方箋だが、前提条件によって別の対応策が考えられる。

1. issue-312のマージがおかしいとするケース
   * 本来想定していたリリーススケジュールから見て、issue-312がdevelopにマージされている状態が正しくないのであれば、issue-312はdevelopにマージせず待機しておくべきだった
   * 誤ってissue-312をマージしてしまったことが原因であれば、リバートを行うことが正しい
2. issue-394のマージがおかしいとするケース
   * 本来想定していたリリーススケジュールを破って、issue-394を優先してリリースしたいというのであれば、feature ではなく hotfix ブランチで対応すべきであった

2の例を以下に図示する



hotfixで追い抜き

# ブランチ命名規則

ブランチ名の命名規則は、[ブランチの種類](#ブランチの種類) 章に従うこと。

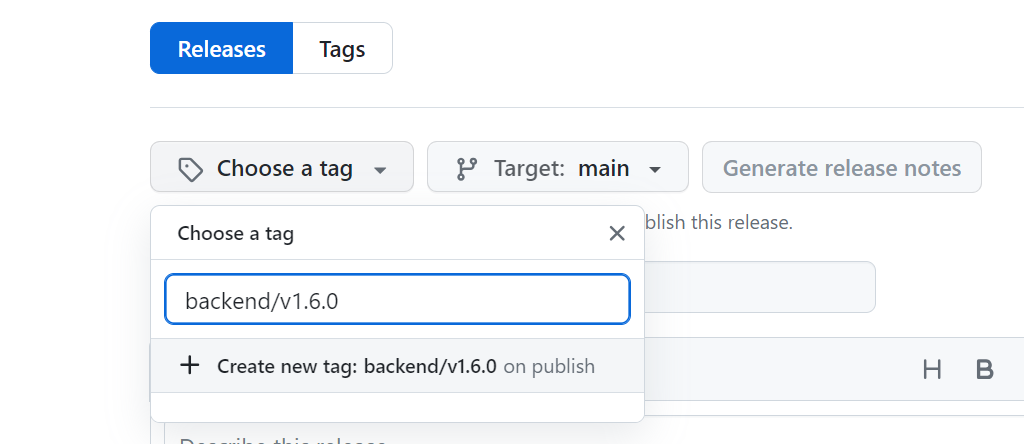
# タグ規則

Gitには[タグ](https://git-scm.com/book/ja/v2/Git-%e3%81%ae%e5%9f%ba%e6%9c%ac-%e3%82%bf%e3%82%b0)機能があり、リリースポイントとしてタグを作成する運用とする。

これにより、リリースしたアプリケーションやライブラリに何か不具合があれば、切り戻しや原因追求が容易になる利点がある。

## タグの運用ルール

* リリースごとに新しいバージョンを示したタグを発行する
* (推奨) GitHubなどの画面経由でタグを作成する
* mainブランチにてタグを作成する
* 入力間違えなどのケースを除き、一度タグをつけた後は削除しない
* 後述する「タグの命名規則」に従う



GitHub画面でbackend/v1.6.0のタグを作成する

何かしらの理由で、コマンドラインからタグを作成する必要がある場合は、以下に注意する。画面経由・コマンドライン経由でのタグ作成は混ぜないようにし、運用手順は統一する。

* 軽量 (lightweight) 版ではなく、注釈付き (annotated) 版のタグを利用する

# OK（注釈付きタグを利用する）  
$ git tag "v1.0.4" -m "v1.0.4 🐛Fix item api log"  
  
# NG（軽量タグは利用しない）  
$ git tag "v1.0.4"

::: warning タグが迷子になる？

タグはmainなど永続ブランチで作成する必要がある。例えばfeatureブランチでタグを作成し、git push origin {tag\_name} でリモートにプッシュしたとしても、rebase でその紐付けが消えてしまう。なぜなら、タグはコミットのハッシュに付与されるラベルであり、rebase でコミットのハッシュが変わると、どのコミットに対するタグなのか分からなくなってしまうからである。 :::

## タグの命名規則

* v1.2.4 などの [セマンティックバージョニング](https://semver.org/lang/ja/) を基本とする
* モノリポの場合は frontend/v1.0.0、backend/v2.0.1 など領域ごとにプレフィックスを付与する形式を取る
  + プレフィックスにすることで、タグをリスト表示した場合に視認性を上げることができる

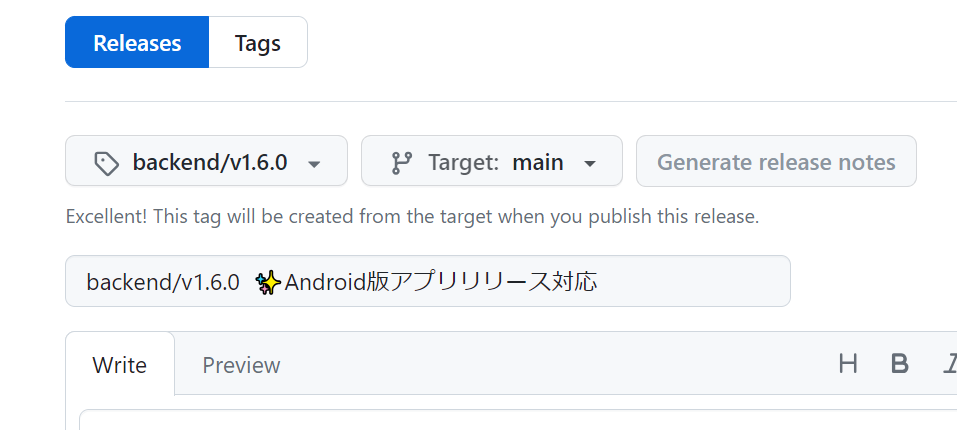
命名に従うと、次のようなコマンドで絞り込みで表示できる。

$ git tag -l --sort=-version:refname "frontend/v\*"  
frontend/v2.0.0  
frontend/v1.3.0  
frontend/v1.2.0  
frontend/v1.1.0  
...

また、Gitクライアントによっては / を使うことでフォルダのように階層表示ができるため、プレフィックスの区切り文字は - ハイフンではなく、スラッシュとする。

## タグメッセージの規則

* (推奨) GitHubを利用中の場合、「[Generate release notes](https://docs.github.com/en/repositories/releasing-projects-on-github/automatically-generated-release-notes)」を用いて、タイトルや本文を自動生成する
* フロントエンド・バックエンドで整合性を保っているのであれば、メモ目的でバージョンを記載する運用を推奨とする
* 実用的な利用用途が思いつかない場合は、開発者視点での楽しみリリースの大きなマイルストーンの名称など、チームの関心事を記入することを推奨とする



create new tag

何かしらの理由で、コマンドラインからタグを作成する必要がある場合は、GitHub利用時の規則に合わせて次のように作成する。

入力例:

# OK  
$ git tag -a backend/v1.8.0 -m "backend/v1.8.0"  
$ git tag -a backend/v1.9.0 -m "backend/v1.9.0 🚀Release with frontend-v3.0.1"  
$ git tag -a backend/v2.0.0 -m "backend/v2.0.0 ✨Android版アプリリリース対応"  
  
# NG  
$ git tag -a backend/v3.0.0 -m "🚀Release version v2.0.0"

## バージョンアップ規則

* 開発しているプロダクトがライブラリの場合、セマンティックバージョニングに厳密に従う
* 開発しているプロダクトがシステム（アプリケーション）の場合、その成熟度や初回リリースの区切りでバージョンアップを行うことを推奨する。適切なバージョンアップを行うことで視認性が上がり、運用負荷を下げることができる
  + 例1: 初回リリース、カットオーバーで v1.0.0 に上げる
  + 例2: 稼働後1年以上経過し、中規模以上の大きな機能アップデートがあったので、 v2.0.0 に上げる

# ラベル規則

IssueやPRを分類することができるラベルについての利用は自由とする。

PRに適切なラベルを設定し、 [自動生成リリースノート - GitHub Docs](https://docs.github.com/ja/repositories/releasing-projects-on-github/automatically-generated-release-notes) に記載があるように .github/release.yml への設定を行うことで、リリースノートの生成をラベル単位にグルーピングできる。

PRを後で探しやすくするための検索キーとしての位置づけと、リリースノート自動生成という観点でラベルを準備すること。

# コミットメッセージ規則

Gitのコミットメッセージは原則自由とする。理由は以下である。

* 通常、作業はチケット管理システムを駆動に開発するため、情報が重複する
* リリースノートの自動生成での扱いは、どちらかといえばラベルとPRのタイトルが重要
* メンバーによっては粒度の小さいコミットを好む場合も多く、運用の徹底化を図る負荷が高い

チーム規模や特性によっては、Gitのコミットメッセージをルール化することによるメリットが見込まれる場合は、 Conventional Commits をベースとした以下の規約を推奨する。

::: tip Conventional Commitsの勧め Gitのコミットメッセージの書式についてルール化することで、コミットの目的がわかりやすくなる、履歴からのトラッキングの容易になる利点がある。

本規約のコミットメッセージの書式としては、Conventional Commitsをベースとした規約としている。

以下の形式でコミットメッセージを記載することとする。

<type>: <subject> <gitmoji>

コミットメッセージは type、subject、gitmojiの最大3つの要素から構成され、それぞれは後述する書式に従うものとする。 この中でも、type、subjectについては必須とし、ほかの要素についてはプロジェクトの運用にしたがい任意とする。

## type

typeについては必須の要素となり、以下のいずれかを選択するものとする。

| type | 説明 |
| --- | --- |
| feat | 新機能の追加 |
| fix | バグの修正 |
| docs | ドキュメンテーションの更新 |
| refactor | リファクタリング |

## subject

subjectについては必須の要素となり、変更内容を簡潔に記載するものとする。 issue idについては、PRから参照する運用を想定し、コミットメッセージの必須要素とはしないこととする。

## gitmoji

gitmojiについては任意の要素となり、変更内容を視認しやすい絵文字の使用を可能とする。

変更内容と選択される絵文字の対応については厳密とせず、開発者が任意に選択するものとする。

type(feat, fix, docs, refactorなど)に基づく、選択例を以下に示す。

==== Emojis ====  
 :ambulance: 🚑致命的なバグ修正(fix)  
 :bug: 🐛バグ修正(fix)  
 :+1: 👍機能改善・機能修正(fix)  
 :cop: 👮セキュリティ関連の修正(fix)  
 :art: 🎨レイアウト関連の修正(fix)  
 :green\_heart: 💚テストやCIの修正・改善(fix)  
 :wrench: 🔧設定ファイルの修正(fix)  
 :building\_construction: 🏗️アーキテクチャの変更(fix)  
 :tada: 🎉大きな機能追加(feat)  
 :sparkles: ✨部分的な機能追加(feat)  
 :up: 🆙依存パッケージ等のアップデート(feat)  
 :memo: 📝ドキュメント修正(docs)  
 :bulb: 💡ソースコードへのコメント追加や修正(docs)  
 :lipstick: 💄Lintエラーの修正やコードスタイルの修正(refactor)  
 :recycle: ♻️リファクタリング(refactor)  
 :fire: 🔥コードやファイルの削除(refactor)  
 :rocket: 🚀パフォーマンス改善(refactor)

## コミットメッセージ例

上記のルールに従った、コミットメッセージのサンプルは以下のようなものとなる。 以下のようなコミットをルールとすることで、変更内容を視覚的に把握しやすくなる利点がある。

feat: カレンダー機能の追加 🎉

fix: メモリリークの修正 🚑

docs: デプロイフローをドキュメント化 📝

refactor: Lintエラーの修正 💄

:::

# git config推奨設定

git config の推奨設定を紹介する。特にGitワークフローの設定が重要である。

# 基礎  
git config --global user.name "Your Name"  
git config --global user.email "your\_email@example.com"  
  
# プロキシ設定（存在する場合）  
git config --global http.proxy http://id:password@proxy.example.co.jp:8000/  
git config --global https.proxy http://id:password@proxy.example.co.jp:8000/  
  
# プロキシが独自の証明書を持っている場合は、git config http.sslVerify false ではなく、証明書を設定する  
git config --global http.sslCAInfo ~/custom\_ca\_sha2.cer  
  
# Gitワークフロー  
git config --global pull.rebase true  
git config --global rerere.enabled true  
git config --global fetch.prune true  
  
# エイリアス（メンバーそれぞれで別のエイリアスを登録されると、チャットなどのトラブルシュート時に混乱をきすため、ベーシックなものはチームで統一して、認識齟齬を減らす目的で設定を推奨する）  
git config --global alias.st status  
git config --global alias.co checkout  
git config --global alias.ci commit  
git config --global alias.br branch

::: tip git workflowの補足説明

* pull.rebase: pull時にリベースする
* rerere.enabled: コンフリクトの解決を記録しておき、再び同様のコンフリクトが発生した場合に自動適用する
* fetch.prune: リモートリポジトリで削除されたブランチを削除する :::

# クレデンシャル情報の混入防止

## git-secrets

[git-secrets](https://github.com/awslabs/git-secrets)を用いることで、ユーザーパスワードや AWS アクセスキーなどの機密情報が含まれる可能性のあるコードなどをGit リポジトリに追加されないようにできる。

本規約の推奨と理由は以下。

* チームメンバー全員にgit-secretsを導入する
  + Push Protection によってクレデンシャルのリモートへの Push は防げるが、ローカルにクレデンシャルを含むコミット履歴が残っていると復元される可能性がある。そこで、git-secrets を用いてコミット段階でクレデンシャルを拒否する設定を行っておく

## Push Protection

Push Protectionを有効化することで、コードをプッシュする際にクレデンシャルが含まれていないかチェックする。 もしクレデンシャルが検知されると、プッシュが拒否されるようになる。 メンバー全員がgit-secretsを設定していれば不要であるが、設定漏れなどでプッシュされてしまうことを防ぐために、本規約では有効化しておくことを推奨する。

::: info 参考

* [GitLab Docs | Secret push protection](https://docs.gitlab.com/user/application_security/secret_detection/secret_push_protection/)
* [GitHub | プッシュ保護について](https://docs.github.com/ja/code-security/secret-scanning/introduction/about-push-protection)

:::

## Secret Scanning

Secret Scanningを利用することで、Git リポジトリにクレデンシャルが存在するとメールなどでアラートを送ってくれるようにできる。 Secret Scanningではコードだけでなく、IssueやPull Requestなどもスキャンできるため、本規約では利用することを推奨する。

::: info 参考

* [GitHub | シークレットスキャンについて](https://docs.github.com/ja/code-security/secret-scanning/introduction/about-secret-scanning)
* [GitLab | Secret Detection](https://docs.gitlab.com/user/application_security/secret_detection/)

:::

# コミットフックでテスト実行は行わない

[git hooks](https://git-scm.com/book/ja/v2/Git-%E3%81%AE%E3%82%AB%E3%82%B9%E3%82%BF%E3%83%9E%E3%82%A4%E3%82%BA-Git-%E3%83%95%E3%83%83%E3%82%AF) を用いて、コミットやプッシュ時に単体テスト実行などのカスタム処理を追加することができる。これを用いると、ローカルでの動作検証などを未実施な状態でレビュー依頼をしてしまうといった状況を未然に防ぎ、開発フローを強制的に適用することができる。

本規約の推奨と理由は以下。

* git hooksを用いたテスト実行は行わない
  + gitのコマンドを実行するライフサイクルと、動作検証を行いたいライフサイクルは同じでないため、軽微な修正の度にテストが実行されると、作業効率が下がるため
  + CI側でテストが実行されるため、最悪CIで検知が可能
  + 開発者にとって作業効率を考えると、CIで検知ではなくローカルでテスト実行を通してからプッシュするため、CIが整備されている前提ではgit hooksで強制する必然性がないため

::: tip git hooksで何を行うべきか テスト、コード生成、リンターなど、実行時間が長いものは行うべきではない。

実行時間が短いフォーマットであれば、git hooksで実行させると便利なことが多く（CIで違反に気づいて対応する手戻りが減るためである）、必要に応じて導入しても良い。 :::

# .gitattributes

## eol

チーム開発において開発環境がWindows/Macなど複数存在することは少なくなく、また、Gitリポジトリ上の改行コードは統一した方が余計な差分が生じず扱いやすくなる。このときよく用いるのが、 core.autocrlf という設定である。

| 名称 | 設定値 | チェックアウト時の挙動 | コミット時の挙動 |
| --- | --- | --- | --- |
| core.autocrlf | true | 改行コードをCRLFに変換 | 改行コードをLFに変換 |
|  | input | 何もしない | 改行コードをLFに変換 |
|  | false | 何もしない | 何もしない |

特にWindowsでの開発者の作業ミスを防ぐため、 git config --global core.autocrlf input のような設定を行うチームも多い。

しかし、上記の設定漏れや手順が増えてしまうため、本規約では .gitattributes での対応を推奨する。

.gitattributes というファイルをGitリポジトリのルートにコミットしておけば、そのGitリポジトリを使う全員で改行コードの扱いをLFに統一できる。

sh .gitattributes \* text=auto eol=lf

通常、改行コードやインデントの設定は[EditorConfig](https://editorconfig.org/)で行うことが多く、 .gitattributes の設定とは重複する。しかし、環境構築ミスなど何らかのトラブルで動作しなかった場合に改行コードミスで特にジュニアクラスのメンバーが困る状況もゼロとは言えないため、本規約では .gitattributes も作成しておくことを推奨する。

::: warning 特定のファイルのみCRLFでコミットしたい テスト目的であるファイルだけCRLFで読み込ませたいとする。さきほどの .gitattributes の設定ではチェックアウト時に強制的にLFに変換されてしまうため、CRLFのファイルのみ個別で改行コードを指定する必要がある。例えば、testdata/eol配下のCSVをCRLFで扱いたい場合は、以下となる。

```sh .gitattributes \* text=auto eol=lf

# 個別で指定

testdata/eol/\*.csv text eol=crlf

前の行に書いた設定は、後ろの行に書いた設定によって上書きされるため、記載順は「全体に適用する原則」→「個別設定」となるように注意する。  
  
この指定がちゃんと効いているか確認する場合は、 `git check-attr` コマンドを用いると良い。以下のように eolがcrlfで設定されたことが分かる。  
  
```sh  
$ git check-attr -a testdata/eol/input1.csv  
testdata/eol/input1.csv: text: set  
testdata/eol/input1.csv: eol: crlf

::: info 参考

* [行終端を処理するようGitを設定する - GitHub Docs](https://docs.github.com/ja/get-started/getting-started-with-git/configuring-git-to-handle-line-endings)
* [.gitattributesのeol=crlfは改行コードをCRLFに変換してチェックインするものではない - エンジニア的考察ブログ](https://chryfopp.hatenablog.com/entry/2013/04/13/113754)

:::

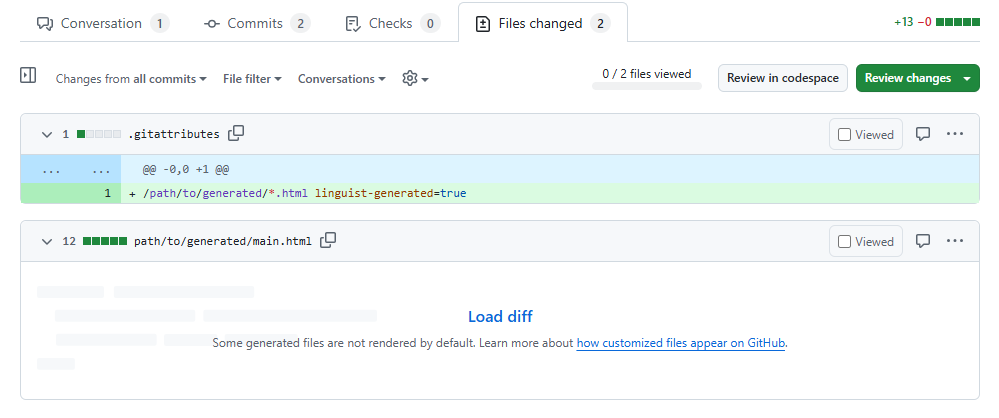
## linguist-generated

自動生成で変更が発生し、かつ大量の変更が頻繁に発生する場合には、レビュワーが毎回レビューをすることは効率的でない。

.gitattributes で linguist-generated=true の設定を行うことで、差分をデフォルトで表示させず、プルリクエストの可視性を向上させることができる。

sh .gitattributes # 自動生成されたHTMLファイルの差分を無視する /path/to/generated/\*.html linguist-generated=true

上記の設定で /path/to/generated/main.html をコミットすると、差分が以下のように非表示となる（Load diffをクリックすることで差分表示は可能）。



プルリクエストのFile changedでmain.htmlの差分がLoad diffと表示

本規約の推奨は以下の通り。

* ツールなどによる生成ファイルをレビュー対象外とする場合は、linguist-generated=true を設定し、レビュアーの負荷を下げる
* レビュアーは差分が省略された場合は、レビュー対象外としてファイルの中身の確認は任意とする

::: tip 生成コードをレビュー対象としたい場合

GitHubでは、[言語毎に生成ファイルと判定する処理](https://github.com/github-linguist/linguist/blob/v9.0.0/lib/linguist/generated.rb)があり、例えツールで作成されたファイルであっても、レビュー確認を必須としたい場合には、クリックする手間が増える分、逆に非効率になる。

例えば、Javaなど複数の言語では3行目までに Generated by the protocol buffer compiler. DO NOT EDIT! が含まれていると[Protocol Bufferの生成コードとみなされる](https://github.com/github-linguist/linguist/blob/63cfd70d54ee8f76c41a73fe56689ed8229c9622/lib/linguist/generated.rb#L348-L359)。

もし、明示的に差分を表示させたい場合、linguist-generated=false を設定する必要がある。

sh .gitattributes # 以下はコード生成されたファイルだが、レビュー対象としたいためlinguist-generated=falseを設定し、差分を表示させる /path/to/generated/\*.java linguist-generated=false

:::

::: info 参考

* [変更したファイルの GitHub での表示方法をカスタマイズする - GitHub Docs](https://docs.github.com/ja/repositories/working-with-files/managing-files/customizing-how-changed-files-appear-on-github)
* [GitHubでファイル差分が表示されない！？レビューを快適にするための差分の非表示ロジックを解説](https://zenn.dev/hacobell_dev/articles/show-diff-in-github)

:::

# .gitignore

Gitで管理したくないファイル名のルールを定義する.gitignoreファイルも入れる。ウェブフロントエンドであれば新規プロジェクトを作成すると大抵作成されるのでそれを登録すれば良いが、もしない場合、あるいは複数の言語を使っている場合などは[GitHubが提供するテンプレート](https://github.com/github/gitignore)を元に作成すると良い。GlobalフォルダにはWindows/macOSのOS固有設定や、エディタ設定などもある。

環境設定を.envで行うのが一般的になってきているが、.env.local、.env.dev.localといった.localがついたファイルはクレデンシャルなどの機微な情報を扱うファイルとして定着しているため、 \*.localも追加すると良い。

# 個人用のファイルをGit管理対象外とする

.gitignore を用いると、チームでGit対象外とするファイルを一律で設定できる。

一方で、動作確認用のちょっとしたスクリプトなどで以下の要件が出てくることがある。

* 個人的にGitリポジトリ配下のフォルダに格納したいが、コミットしたくない（≒自分のローカルリポジトリのみ必要である）
* あくまで個人用途であるため .gitignore に追記したくない

上記の場合は、.git/info/exclude を利用することを推奨する。

::: info 参考 [個人的Gitおすすめtips 7選 #GitHub - Qiita](https://qiita.com/hichika/items/f3c980dd069df0f3a56e) :::

# Pull Request / Merge Request テンプレート

GitHubやGitLabでは、プルリクエスト作成時のテンプレートを作ることができる。チームでプルリクエストで書いてほしいことを明示的にすることで、レビュー効率の向上や障害調査に役立てることができる。

GitHubでは .github/PULL\_REQUEST\_TEMPLATE.md に記載する。（GitLabでは .gitlab/merge\_request\_templates/{your\_template}.md を配置する。）

テンプレートの例を以下にあげる。

## チケットURL  
  
## 特に見てほしいレビューポイント  
  
## 残課題（別チケットで対応予定の内容、別プルリクエストで対応予定の内容）  
  
## 動作確認内容（画面キャプチャなど）  
  
## セルフチェックリスト  
  
- [ ] 開発規約(DEVELOPMENT.md) を確認した  
- [ ] Files changed を開き、変更内容を確認した  
- [ ] コードの変更に伴い、同期必要な設計ドキュメントを更新した  
- [ ] 今回のPRでは未対応の残課題があればIssueに起票した

# GitHub推奨設定

業務利用でのチーム開発を想定しており、リポジトリは以下の条件を満たす前提とする。

* プライベートリポジトリ
* Organization配下に作成
* Teamsプラン以上の有料契約（※プロテクトブランチの機能などを利用するために必要）

## General

| Category | Item | Value | Memo |
| --- | --- | --- | --- |
| General | Require contributors to sign off on web-based commits | チェックなし | 著作権・ライセンス承諾の場合に用いるが、業務アプリ開発では不要 |
|  | Default branch | develop |  |
| Pull Requests | Allow merge commits | ✅️ | main <- developなどのマージ時に必要 |
|  | Allow squash merging | ✅️ | develop <- feature はSquash mergeを推奨 |
|  | Allow rebase merging | - | 利用しないため、チェックを外す |
|  | Allow suggest updating pull request branches | ✅️ | Pull Request作成後、ベースブランチが更新された場合、ソースブランチの更新を提案してくれる |
|  | Automatically delete head branches | ✅️ | マージ後にfeature branchを削除するため有効にする |
| Pushes | Limit how many branches and tags can be updated in a single push | 5 | git push origin –mirrorで誤ってリモートブランチを破壊しないようにする。推奨値の5を設定する |
| Security | Secret scanning | ✅️ | コードやIssue、コメント等のクレデンシャル情報を検知し、通知を行う |
|  | Push Protection | ✅️ | プッシュ時にクレデンシャル情報が検知された場合、プッシュをブロックする |

## Access

| Category | Item | Value | Memo |
| --- | --- | --- | --- |
| Collaborators and teams | Choose a role | 任意の権限 | ※後述 |

* 各ロールの権限については、公式ドキュメントを参照
* 通常、開発者には「Write」ロールを付与する
* 開発を行わない、例えばスキーマファイルの参照のみ必要であれば、「Read」権限を、Issueの起票などのみ実施するマネージャーであれば「Triage」ロールを付与する
* 「Maintain」権限は、付与しない
* 「Admin」権限は、マネージャークラスに対して合計2~3名を付与し、属人化しないようにする
  + 1名でも、4名以上でもNGとする

## Code and automation

### Branches

Branch protection rules にdevelop, mainなど永続的なブランチに保護設定を追加する。

| Category | Item | Value | Memo |
| --- | --- | --- | --- |
| Protect matching branches | Require a pull request before merging | ✅️ | プルリクエストを必須とする |
|  | Require approvals | ✅️ | レビューを必須とする |
|  | Required number of approvals before merging | 1 | 最低1名以上の承認を必須とする |
|  | Dismiss stale pull request approvals when new commits are pushed | - | レビュー承認後のPushで再承認を必要とするかだが、レビュー運用上に支障となることも多く、チェックを外す |
|  | Require status checks to pass before merging | ✅️ | CIの成功を条件とする |
|  | Require branches to be up to date before merging | 任意 | CIパイプラインのワークフロー名を指定 |
|  | Require conversation resolution before merging | - | レビューコメントがすべて解決していることを条件とする。チェックを外す |
|  | Require signed commits | ✅️ | 署名付きコミットを必須化し、セキュアな設定にする |
|  | Require linear history | ✅️/- | mainブランチの場合はOFFとするが、developの場合はSquash mergeを求めるため有効にする |
|  | Do not allow bypassing the above settings | ✅️ | パイパスを許容しない |

developブランチに対し「require linear history」を選択することを推奨することで、「Create a merge commit」が選択できないようにする。

また、意図しない方法でのマージを避けるためにブランチごとにマージ戦略を設定しておき、想定外のマージ戦略が選択された時に警告色を表示するというサードパーティ製のChrome拡張[[3]](#footnote-137)も存在する。必要に応じて導入を検討する。

### Tags

| Category | Item | Value | Memo |
| --- | --- | --- | --- |
|  | Protect tags | v[0-9]+.[0-9]+.[0-9] | セマンティックバージョニングに則ったタグのみ、削除を防ぐ |

### GitHub Actions

| Category | Item | Value | Memo |
| --- | --- | --- | --- |
| Actions permissions | Allow asset-taskforce, and select non-asset-taskforce, actions and reusable workflows > Allow actions created by GitHub | ✅️ |  |
|  | Allow asset-taskforce, and select non-asset-taskforce, actions and reusable workflows > Allow actions Marketplace verified creators | ✅️ |  |

### Code security and analysis

| Category | Item | Value | Memo |
| --- | --- | --- | --- |
| Dependabot | Dependabot alerts | ✅️ | 依存パッケージのアップデートを検知するため |
|  | Dependabot security updates | ✅️ |  |
|  | Dependabot version updates | ✅️ |  |

# GitLab推奨設定

* GitHubの「Automatically delete head branches」
  + マージリクエストから「Delete source branch」オプションを有効にすることが該当
  + プロジェクトの設定で「Enable “Delete source branch” option by default」を選択しておくとデフォルトで有効になる

# ローカルでのGit操作

## gitコマンド

# 変更作業  
git checkout -b <branchname>  
git add  
git commit -a  
  
# リモートブランチの変更を同期  
git pull origin develop  
  
# コンフリクト対応  
git add <file1> <file2> ...  
git commit -a  
  
# リモートブランチへプッシュ（pullした際にリベースしているため、オプションは必須である）  
git push origin HEAD --force-with-lease --force-if-includes

## VS Code

利用頻度が高いとされるGitクライアントである、VS Code上でのGit操作を紹介する。

::: tip VSCode上でのGit操作の紹介

VSCode上でのGit操作は、サイドバーの “Source Control” から行うことができる。ほとんど全ての操作はコマンドパレットからも実行可能だが、説明は割愛する。

### 推奨する拡張機能

GUIでのGit操作にあたり、次の2つの拡張機能をインストールしておくと利便性が高い。業務上はほぼ必須と見て良い。

* [GitLens](https://marketplace.visualstudio.com/items?itemName=eamodio.gitlens)
  + Gitに関する様々な機能を提供する拡張機能
  + 詳細： [VSCodeでGitLensを使う - フューチャー技術ブログ](https://future-architect.github.io/articles/）20240415a/)
* [Git Graph](https://marketplace.visualstudio.com/items?itemName=mhutchie.git-graph)
  + コミットグラフを表示する拡張機能
  + GitLensにもコミットグラフはありますが、Pro（有料版）限定の提供のため、ここではこちらの拡張機能を使用する

以降では、これらの拡張機能がインストールされていることを前提に説明を行う。

### リポジトリのクローン (git clone)

1. サイドバー > Explorer か Source Control > Clone Repository ボタンをクリック
2. URLを入力すると、リポジトリをクローンできる

| 1 | 2 |
| --- | --- |
| Clone1 | Clone2 |

### コミットグラフの表示

1. SOURCE CONTROL パネル > 黒丸のグラフアイコン (View Git Graph (git log)) をクリック
2. コミットグラフが表示される

| 1 | 2 |
| --- | --- |
| Graph1 | Graph2 |

白丸のグラフアイコン (Show Commit Graph) はGitLensのコミットグラフだが、冒頭の記述通り、Pro版でのみの提供となる。

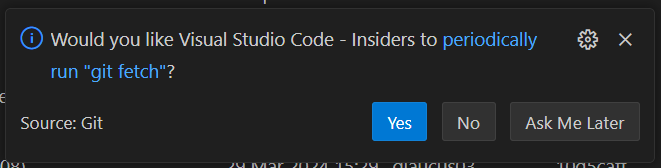
### リモートのフェッチ／プル (git fetch / git pull)

以下のいずれかの操作を実行すると、リモートリポジトリをフェッチできる。

* 1. SOURCE CONTROL パネル > 三点リーダーアイコン (More Actions…) をクリックし、 Fetch を選択
  2. コミットグラフ > 雲アイコン (Fetch from Remote(s)) をクリック

| a, bの手順両方を記載 |
| --- |
| Fetch1 |

なお、フェッチ後に以下のようなダイアログが表示される場合があるが、 “Yes” を選択すると、自動で定期的にフェッチを行う。



Fetch2

### ブランチの作成／チェックアウト (git branch / git checkout)

以下のいずれかの操作を実行すると、ブランチを作成できる。

* 1. SOURCE CONTROL パネル > 三点リーダーアイコン (More Actions…) をクリックし、Branch > Create Branch… を選択
  + 現在チェックアウトしているブランチから新規ブランチが作成されますが、Create Branch From… を選択すると、作成元のブランチを選択することができる
  + 作成したブランチに自動的にチェックアウトする
  1. コミットグラフ > 作成元コミットの行上で右クリックし、Create Branch… を選択
  + “Check out” にチェックを入れると、作成したブランチにチェックアウトする

| a | b |
| --- | --- |
| Branch1 | Branch2 |

### ステージ／コミット／プッシュ (git add / git commit / git push)

1. SOURCE CONTROL パネル > 変更ファイルの行 > +アイコン (Stage Changes) をクリックすると、対象ファイルをステージできる。（Changes > +アイコン (Stage All Changes) をクリックすると、すべての変更をステージする）
2. 必要な変更をステージ後、 SOURCE CONTROL パネル内でコミットメッセージを入力し、 Commit ボタンをクリックすると、コミットを作成できる

| 1 | 2 |
| --- | --- |
| Stage | Commit |

以下のa～cいずれかの操作を実行すると、作成したコミットをリモートリポジトリにプッシュできる。

* 1. SOURCE CONTROL パネル > 三点リーダーアイコン (More Actions…) をクリックし、Push を選択
  2. BRANCHES パネル > 対象ブランチの行 > 雲アイコン (Publish Branch) をクリック
  3. コミットグラフ > 対象ブランチの上で右クリックし、Push Branch… を選択

| a | b | c |
| --- | --- | --- |
| push1 | push2 | push3 |

:::

# 謝辞

このアーキテクチャガイドラインの作成にあたり、多大なるご協力をいただきました皆様に心より感謝申し上げます。

* **作成者**（宮崎将太、真野隼記、武田大輝、澁川喜規、清水 雄一郎）
* **レビュアー**（藤田春佳、中村眞之介）

皆様のご尽力なしには、本ガイドラインの完成はなしえませんでした。改めて深く感謝いたします。

1. [github/renaming: Guidance for changing the default branch name for GitHub repositories](https://github.com/github/renaming) [↑](#footnote-ref-22)
2. [複数の作者を持つコミットを作成する - GitHub Docs](https://docs.github.com/ja/pull-requests/committing-changes-to-your-project/creating-and-editing-commits/creating-a-commit-with-multiple-authors) [↑](#footnote-ref-68)
3. [GitHubで誤ったマージ戦略のマージを防ぐChrome拡張機能の開発をした](https://zenn.dev/daku10/articles/github-merge-guardian) [↑](#footnote-ref-137)