コードレビューガイドライン

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

[プログラマが知るべき97のこと](https://xn--97-273ae6a4irb6e2hsoiozc2g4b8082p.com/) の14個目に「コードレビュー」について記述がある。コードレビューの目的は「コードの質を上げ、欠陥を減らすため」だけではなく、「チーム全員に同じ知識を共有させること、またコーディングにおいて全員が守るべきガイドラインを確立すること」が大切だとある。そして「 レビューを楽しいものにすること」がおそらく最も有効だとある。

コードレビューを楽しい場にするためにも、レビュアー／レビュイー が同じ方向性を向くことが重要である。同時に、人によってコメントなどの表現が揺れ、それにより認識齟齬があると楽しむ以前の問題であるため、守るべきルールも存在するはずである。このガイドラインは推奨する行動と、守るべきルールの両方を定義し、コードレビューをより有意義で学びが多く、生産性と品質を高める場とすることを目指す。

::: info 参考 [コードレビューとは | GitLab](https://about.gitlab.com/ja-jp/topics/version-control/what-is-code-review/) にも似た旨の記載がある :::

# 定義

* **レビュアー**: レビューを行う側の担当者
* **レビュイー**: レビューを依頼する側の担当者。プルリクエストを起票する開発者でもある
* **プルリクエスト**: GitLabではマージリクエストともいうが、本ガイドラインではプルリクエストに統一して呼称する

# 適用範囲

GitHubやGitLabなどのサービスを利用した、コーディングについてのレビューのみを対象とする。

プロジェクト計画によっては、開発プロセスを複数の「フェーズ」に分割し、各フェーズの終了時点で「フェーズレビュー」を挟むようなケースも考えられるが、それらは対象外とする。

# 前提条件

## 利用するGitブランチ戦略

[Gitブランチフロー規約 > ブランチ戦略の選定](https://future-architect.github.io/coding-standards/documents/forGitBranch/git_branch_standards.html#%E3%83%95%E3%82%99%E3%83%A9%E3%83%B3%E3%83%81%E6%88%A6%E7%95%A5%E3%81%AE%E9%81%B8%E5%AE%9A) にある、以下の2パターンのいずれかを利用しているとする。

* Lite GitLab Flow
* GitLab Flow

そのため当然であるが、全ての機能開発や不具合修正に、feature ブランチを使用したプルリクエストを経由してマージする方針を取る。その他の feature ブランチから develop ブランチへのマージ方法などもGitブランチフローに記載した推奨事項に則る。

## 事前設定（GitHub）

[Gitブランチフロー規約](https://future-architect.github.io/coding-standards/documents/forGitBranch/git_branch_standards.html#git-config%E6%8E%A8%E5%A5%A8%E8%A8%AD%E5%AE%9A) に則った設定を実施する。例えば、ブランチ保護やレビューの設定には以下の内容がある。できるかぎり自動化やテンプレートを用意することで、個々人が考慮すべきことを減らすことで運用負荷を下げ、規則を形骸化にせず実効性を上げることを基本姿勢とする。

| 設定項目 | 設定例 | 説明 |
| --- | --- | --- |
| Require a pull request before merging | ✔ | プルリクエストを必須とする |
| Require approvals | 1 | 1名以上の承認を必須とする |
| Require status checks to pass before merging | ✔ | CIの成功をマージ条件とする |
| [CODEOWNERS](https://docs.github.com/ja/repositories/managing-your-repositorys-settings-and-features/customizing-your-repository/about-code-owners) | (省略) | 重要ファイル（例えば、ddl.sqlやopenapi.yamlなど）は承認が必須なレビュアーを指定する |
| [linguist-generated=true](https://docs.github.com/ja/repositories/working-with-files/managing-files/customizing-how-changed-files-appear-on-github) | (省略) | ツールなどによる生成ファイルの差分を非表示にする .gitattributesの設定 |

## 機械的なチェックが可能な場合はCIに寄せる方針としている

コードレビューのやり取りで、「コードフォーマットされていない」 「リンターを通していない」といった指摘は不毛である。そのため、CIでそれらを実行し、パスしているという前提条件を作ることが重要である。それにより、レビュアーは人間が本来見るべき点に集中できるためである。

例えば、下表のような内容はフォーマッタやリンターに寄せることができる可能性が高い。

| 自動化可能だと考えられる指摘例 | ツール例 |
| --- | --- |
| 使われていないメソッドや関数 | Pythonの場合、Flake8 など |
| デッドコード | Pythonの場合、Vultureなど |
| デバック用のログの残骸 | JSの場合は、ESLint（console.logの検知など）。 Pythonの場合は、Pylintなど |
| 不適切なレイヤーに対するimport | Pythonの場合、import-linterなど |
| //TODO コメントが残っている | JSでは、eslint-plugin-no-inline-commentsなど |
| フォーマッタが適用されているかどうか | JSでは prettier --check "yourfile.js"、Pythonでは black --check yourfile.py など |

## CIによるレビューコメント連携を行う

Linterなどの検知結果をreviewdogなどと連携することで、形式的な指摘はツール側に寄せる。それにより開発者個人個人のやり取りを減らし、本当に集中すべき事項に時間を割くこととする。

## プルリクエストテンプレートでチェックリストを設ける

プルリクエストにチェックリストを追加し、レビュー依頼前にセルフチェックする運用を採用するチームも多い。チェックリストが多すぎると形骸化するため、項目は具体的にする（例: 設計ドキュメントを同時に直したか？など）とともに、必要最低限の項目に留め、チームの成熟度に応じて適時見直す運用とする。

::: info 参考 [リポジトリ用のプルリクエストテンプレートの作成 - GitHub Docs](https://docs.github.com/ja/communities/using-templates-to-encourage-useful-issues-and-pull-requests/creating-a-pull-request-template-for-your-repository) :::

## チームで合意した開発規約が存在し遵守しつつ育てる

以下のような、チームで守るべきと合意形成されたコーディング規約や命名ルールが存在し、これらの内容をチームで改善し続ける運用であることを前提とする。

* [Javaコーディング規約](https://future-architect.github.io/coding-standards/documents/forJava/Java%E3%82%B3%E3%83%BC%E3%83%87%E3%82%A3%E3%83%B3%E3%82%B0%E8%A6%8F%E7%B4%84.html)
* [SQLコーディング規約](https://future-architect.github.io/coding-standards/documents/forSQL/SQL%E3%82%B3%E3%83%BC%E3%83%87%E3%82%A3%E3%83%B3%E3%82%B0%E8%A6%8F%E7%B4%84%EF%BC%88PostgreSQL%EF%BC%89.html)
* [AWSインフラ命名規約](https://future-architect.github.io/coding-standards/documents/forAWSResource/AWS%E3%82%A4%E3%83%B3%E3%83%95%E3%83%A9%E3%83%AA%E3%82%BD%E3%83%BC%E3%82%B9%E5%91%BD%E5%90%8D%E8%A6%8F%E7%B4%84.html)
* [OpenAPI Specification 3.0.3規約](https://future-architect.github.io/coding-standards/documents/forOpenAPISpecification/OpenAPI_Specification_3.0.3.html)

# レビュー観点

「フォーマッタやリンター」などのツールをCIと組み合わせることで、レビュアーが見るべき領域を減らすことができる。また、「チームで合意された開発規約」をレビュイーが遵守できれば、ツールがカバーできない領域に対してもレビュアーの負担減に繋がる。

| 大分類 | 中分類 | 小分類 | レビュイーが気にすべきか | レビュアーが見るべきか |
| --- | --- | --- | --- | --- |
| 開発規約に記載済 | ツールでカバー可能 |  | ⚠️ツール補助あり | 不要 |
|  | ツールでカバー不可 |  | ✅️規約を準拠しているか確認する必要がある | ✅️規約を準拠しているか確認する必要がある |
| 開発規約に未記載 | 規約に記載可能 |  | ⚠️Yesだが、難しい場合が多い | ✅️言語化して開発規約に追加すべき |
|  | 規約に記載不可 | 機能要件の充足 | ✅️ | ✅️ |
|  |  | 文脈依存が強く汎化不可 | ⚠️Yesだが、難しい場合が多い | ✅️個別の命名や、他に類似の実装パターンがない場合は個別に見るしかない |

レビュー観点は、開発しているプロダクトの性質、開発体制、採用技術で大きく変動するためここで詳細は述べない。

基本的には、チームで合意形成された開発規約（参考: [開発アーキテクチャドキュメント](https://jisou-programmer.beproud.jp/%E5%AE%9F%E8%A3%85%E3%81%AE%E9%80%B2%E3%82%81%E6%96%B9/39-%E9%96%8B%E7%99%BA%E3%82%A2%E3%83%BC%E3%82%AD%E3%83%86%E3%82%AF%E3%83%81%E3%83%A3%E3%83%89%E3%82%AD%E3%83%A5%E3%83%A1%E3%83%B3%E3%83%88.html)）やツール（リンターやフォーマッタ）を充実させる方向に寄せるべきである。以下に、開発規約にはどのような内容を記載すべきかを以下に参考情報としてまとめる。

# 開発ガイドライン（例）  
  
## 仕様確認  
  
- 開発の入力情報となる、設計ドキュメント、チケットなどの前提が正しいかどうかの確認は開発者が行う  
- もし、入力情報が疑わしい場合は開発者自身が有識者に確認、必要に応じてチケットを起票して、ドキュメント修正を行うか、修正依頼を行う  
  
## 機能要件  
  
フロントエンド:  
  
- 設計書通りの見た目となっているか確認する  
- 必須/任意、Readonlyなど表示仕様は適切か確認する  
- 表示項目の編集仕様（時刻など）は他機能と整合性が取れているか確認する  
  
バックエンド:  
  
- 環境によって変わる値がハードコードになっていないか  
  
## 性能観点  
  
SQL:  
  
- N+1クエリはなるべく避けるようにする  
- アプリケーション側でジョインしてはならない  
- SQLの検索条件で、パーティションキーの指定が抜けていないように注意する  
  
## 可用性  
  
- 外部API呼び出し時などでタイムアウトは設定されているか  
- 外部API呼び出し時に、リトライ設定は行っているか  
  
## 可読性  
  
- [命名規則](https://java-standard.example.com) に従う  
- コードコメントは、JavaDoc形式（Markdown）をなるべく使用する  
  
## ログ  
  
- loggerはxxxを利用すること  
- ループ内で大量に呼び出される場合は、debugログであっても性能劣化懸念があるため不可とする  
  
## テスト  
  
- カバレッジはC0 90%程度が目安  
- バックエンドについては、デシジョンテーブル（決定表）からテストケースを作成する

::: tip 関連 [コードレビューの観点 | google-eng-practices-ja](https://fujiharuka.github.io/google-eng-practices-ja/ja/review/reviewer/looking-for.html) :::

# レビュイーの推奨行動

## 重要な変更の場合、方向性を事前にチームと合意形成しておく

実装する内容が、設計書に基づく新規機能の改修や、不具合改修でチケットに原因分析や対応方針が明記されている場合は、チーム内で改めて対応方針のすり合わせを行う必要はない。

考慮の残がありかつ、すでに稼働済みの機能の挙動変更を行う場合や、原因が特定されていない不具合修正の場合は、以下の内容をチームで合意形成してから実装に入った方が、認識齟齬や手戻りを減らせ、レビュイー／レビュアー双方の負荷を下げることができる

* （不具合対応の場合）
  + 事象
  + 業務影響
  + システム影響
  + 原因分析
  + 対応方針
  + この変更に伴う影響範囲（分割リリースを行うかどうか）
* （既存機能の挙動変更）
  + 変更理由（顧客要望）が何か
  + 対応方針
  + この変更に伴う影響範囲（分割リリースを行うかどうか）

::: tip スキーマファイルは事前にレビューしておくと手戻りを防ぎやすい  
「WebAPIのスキーマ定義」や「DBスキーマ定義」など、システム構成要素の境界となるようなスキーマファイルの変更は、システムに対する影響度が大きいためレビュアーとしても設計を妥協しにくい。また、これらのファイルに指摘が入ると、ほぼ必然的にアプリケーションコードも修正することになるため、手戻りも大きくなる。これら境界となるようなファイルは先行して、Draftプルリクエストでレビュアーに確認してもらってから作業する工夫は有効である。また、そのような開発ルールにするチームも多い。  
:::

参考: [Design Doc でチームを跨いだ開発を円滑に行う - 一休.com Developers Blog](https://user-first.ikyu.co.jp/entry/2024/12/09/143648)

## プルリクエストの単位を小さくする

複数の改修内容が単一のプルリクエストに含まれており、変更内容が多い（20ファイル以上などの）場合、レビュアーの負荷が高くなってしまう。コミットを意味のある粒度に保ちつつ細かくするか、プルリクエストの粒度を細かくするかの2通りの対応が考えられる。

推奨は以下の通り。

* プルリクエストの単位を小さくする
* 分割したという旨を、プルリクエストの説明に追記する
* 何が残りの作業であるかは、チケットやプルリクエストに記載し、レビュアーが把握できるようにする（作業漏れとして指摘すべきか判断可能にするため）

::: tip 大きな変更の場合、目的別にプルリクエストを分ける  
プルリクエストの粒度を細かくすることで、レビュアーの負荷を下げることができる。例えば、ある機能改修を行う際に、リファクタリングを行ってから作業をおこないたい場合は、リファクタリングのみを行うプルリクエストと、機能追加を行うプルリクエストを分けることがあげられる。なお、ローカル変数の名称を変更するといった小さな変更は、わざわざプルリクエストを分ける必要はない。  
:::

::: tip プルリクエストを小さくすることは学習効果もある？

[Findyの爆速成長を支えるエンジニア教育メソッド ~ 育成ノウハウの一部を初公開 ~ Findy Tech Blog](https://tech.findy.co.jp/entry/2025/02/18/070000)に以下の記述がある。

手数に関しては、とにかく小さい修正でもいいので適切な粒度でPull requestを作り続け、大量のフィードバックを得るようにします。

量は質に転化すると言われているように、手数が増えれば増えるほど結果的に質と理解が深まっていき、最終的にはスピードが身に付きます。

また、指摘事項が多い場合はPull requestの粒度が大きすぎるケースが多く、そういったケースでは粒度を細かくしてPull requestを作り直してもらいます。一度に100個指摘されて100個覚えるよりも、1個指摘されて1個覚えることを100回繰り返すほうが覚えやすいからです。

このように、学習観点でのプルリクエストの単位を小さくするメリットを説いている。一度に全てのことを覚えるのは誰にとっても難しいため、段階を追って1個ずつ学ぶことができるような進め方をすると良い。

:::

## コミットメッセージはこだわらなくても良い

[commitlint](https://commitlint.js.org/) を適用し、レビュー負荷軽減のためコミットメッセージの統制を図るという考え方も存在する。

推奨は以下の通り。

* コミットメッセージは開発者の裁量に任せる
  + 「wip」や「リファクタリング」など荒くても良い
* コミットメッセージより、プルリクエストのタイトルに気を配る

理由は以下。

* プルリクエストの単位を小さくする前提であるため、コミットの粒度やメッセージを気にすることによるメリットが小さい
* Gitブランチ規約から、最終的にスカッシュマージするため重要度が下がる

## プルリクエストのタイトルにプレフィックスを入れる

プルリクエストのタイトルに命名規則をもたせることで、レビュアーの負荷軽減に繋げることができる。また、履歴からのトラッキングが容易になる。

推奨は以下の通り。

* タイトルは {type}: {subject} や {type}: {feature\_id}: {subject} という構成に従う
* type は下表のいずれかを選択し、複数該当する場合はより上位にあるものを選ぶ

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

::: tip ラベルとの使い分け  
プルリクエストの[ラベル](https://docs.github.com/ja/issues/using-labels-and-milestones-to-track-work/managing-labels)でも同じ意図を持たせることができるが、レビュアーにとっての視認性を重視するため、重複した内容であってもタイトルにプレフィックスを含ませる。

関連: [Gitブランチフロー規約 > ラベル](https://future-architect.github.io/coding-standards/documents/forGitBranch/git_branch_standards.html#%E3%83%A9%E3%83%98%E3%82%99%E3%83%AB%E8%A6%8F%E5%89%87)  
:::

## 差分（diff）を極小化することにこだわりすぎない

レビュー負荷軽減や、影響度の極小化の観点から、なるべく差分を小さくすることを志向した改修を行うべきか迷う場面がある。

推奨は以下の通り。

* あるべき変更を最優先とし、その後でレビュー負荷軽減のため差分を小さくできないか考える
  + リファクタリングなど、可読性／保守性を上げる改善は継続する必要がある
  + 差分を小さくするために、コードベースの改善活動を止めるのは本末転倒である（良いモノを作るためにコードレビューをしている面がある）
* 必要に応じて、プルリクエストの分割を検討する

::: tip GitHubやGitLabでは空白のみの差分を無視できる

GitHubやGitLabではURLの最後に、 ?w=1 を付与することで、空白のみの差分を無視することができる（画面上からの設定も可能）。

例えば、Markdownテーブルに項目を編集するとフォーマッタにより大きな差分が出ることが多く、変更に対する心理的ハードルが上がってしまう。しかし、このオプションを利用することでレビュアーの負荷は下がり本質的なレビューに集中できる。もし、レビュアーがこのオプションを知らないようであれば、 ?w=1 つきのURLをプルリクエストのDescriptionやレビューコメントで追記すると良い。

* [Ignore white space in code review - The GitHub Blog](https://github.blog/news-insights/product-news/ignore-white-space-in-code-review/)

:::

## ファイルの移動やリネームと同時に大きく編集しない

リファクタリングや構成の整理で、ファイルの移動やリネームを行う場面は多々ある。

推奨は以下の通り。

* ファイルの移動やリネームと同時に、ファイル自体を大きく変更しない
  + gitの同一ファイル判定が狂い、ファイルの削除＋ファイルの新規作成となり、該当ファイルの変更履歴が消える可能性があるため
  + レビュアーにとって、新規ファイル扱いになると、本当の差分がどれか画面上は分からず、確認コストが上がってしまうため
* どうしても同一プルリクエストでファイルのリネームと編集を行う必要がある場合は、以下の対応をする
  + git mv で操作してからファイルを変更する
  + ファイルの移動やリネームで、git commit し、その後にファイルの変更を行う

## 作業途中のプルリクエストはDraftにする

何かしらの理由で、マージされたくない作業途中の状態で、プルリクエストを作成したい場合がある。例えば、作業途中の内容を部分的にメンバーに確認してもらいたい場面や、セルフチェックが完了前に別タスクを行う必要が出た場合などがある。誤ってレビューやマージされたくない

推奨は以下の通り。

* GitHubにおいて、プルリクエストの[ドラフトステータス](https://docs.github.com/ja/pull-requests/collaborating-with-pull-requests/proposing-changes-to-your-work-with-pull-requests/changing-the-stage-of-a-pull-request)機能を利用して、誤操作を防ぐ
  + 誤認識、誤操作による作業を防ぐ
  + プルリクエストが滞留している状況で、レビュー待ちなのか作業中なのかが区別して可視化できるようにし、開発状況をマネージャーが把握しやすくするため

::: tip Draftプルリクエストの早期作成のススメ  
作業途中であっても、早期にDraftプルリクエストを作成することで、作業の可視化／透明性に繋げることができる。他にも開発者が作業を再開しやすくなるなどメリットが大きいため、ローカルでの開発作業が完了する前の段階でDraftプルリクエストを利用することを推奨する。  
:::

## プルリクエスト本文にチケット番号や関連情報を貼る

トレーサビリティのため、チケット番号（のURL）や、設計検討したドキュメントのリンクや、議論スレッドのURLなどを、チケット本文に記載することで、後からトレース可能にする。チリツモであるがチームの開発生産性を上げることに繋がるため。

## チケット番号があるのであれば、プルリクエスト本文に重複記載は不要

作業目的（どういった要望／不具合に対する対応か）などは、作業チケットに記載されているのであれば、プルリクエスト本文での記述は不要である。  
なお、サマリ事項だけ転記するなど、詳細な運用ルールはチームごとに決めて良い。

## 長くなる場合はMarkdownの折りたたみを利用する

GitHubでは <details> タグで[折りたたんだセクション](https://docs.github.com/ja/get-started/writing-on-github/working-with-advanced-formatting/organizing-information-with-collapsed-sections)が作成可能である。例えば、単体テストケース化できないような動作確認を行い、かつ動作検証ログをエビデンスとして貼り付けるような場面では、折りたたんだセクションを活用することで、本文を簡潔に保つことができる。

::: tip スラッシュコマンドを利用する  
[スラッシュコマンド](https://docs.github.com/ja/issues/tracking-your-work-with-issues/using-issues/about-slash-commands)を用いることで、簡単に折りたたみできる詳細領域を挿入できる。入力時に / を押すとメニューが表示されるため、「Details」を選択する。  
:::

## 画像をエビデンスとして貼るべきかどうか

[2023年5月9日のアップデート](https://github.blog/changelog/2023-05-09-more-secure-private-attachments/)で、プライベートリポジトリのプルリクエストなどにアップロードされたファイルの閲覧に、認証が必須となった（それまではURLを知っていれば無認証でアクセスができてしまっていた）。

推奨は以下の通り。

* フロントエンドなど画面開発の場合は、作成／修正後の画像（必要に応じてGIF動画）を貼ることを推奨する
* スタックトレースなどは、検索性を考えてテキストで貼ることを推奨する

::: info 参考 GIF動画を作成するには、以下の記事で紹介されているツールの利用者も多い。

* [Future社員が使っているWindows便利ツール（新人さん向け） | フューチャー技術ブログ](https://future-architect.github.io/articles/20220107a/#ScreenToGif)

:::

## Files changedで余計なファイルがコミットされていないか確認する

ジュニアメンバー／シニアメンバー問わず、特に新規参画した当初は新しい環境に慣れるのに手一杯で、誤った操作をしがちである。よくあるミスには、余分なファイルをコミットに含めてしまうことや、誤ったブランチからブランチを作成してしまったため、余計な差分が発生してしまうことなどがある。

推奨は以下の通り。

* レビュー依頼前に、プルリクエスト画面の Files changed タブを開き、対象のファイルが想定通りか確認する

このように、 Files changed を見ると、ローカルのエディタで見るのとは違ったセルフレビューの効果があるため、レビュアーになった気持ちで変更内容を確認することも推奨する。

## Files changedで特に見て欲しいポイントを補足する

プルリクエストで、特に念入りに確認して欲しいポイントや、リファクタリングなど複数のコミットが混ざり、メインの改修ポイントがわかりにくくなってしまう場合がある。後者はコミットごとにレビューすれば解決するケースもあるが、featureブランチはスカッシュマージするチームでは、必ずしもきれいなコミットを求めないケースも多い。

推奨は以下の通り。

* 特に注力してレビューして欲しいポイントがあれば、Files changed でセルフコメントを付ける

::: warning セルフコメントが非推奨の場合

* 意図がわかりにくい実装の補足や、背景情報などは、プルリクエスト上のコメントではなく、設計ドキュメントや、コードコメントに記載する。それにより保守運用性を高めることができる
* 「PJの実装ガイドライン・ルールから敢えて外れる実装を選択している」場合は、コードコメントに記載すべきである

意図や背景情報をレビュアーに強調したい場合は、コードコメントに対して「実装意図はコードコメントに記載」などセルフコメントすると良い。

:::

## Assigneesは自分を設定する

プルリクエストのAssigneesは基本的にはレビュイー本人にする。もし、作成したプルリクエストを別の担当者に引き継ぐときは、Assigneesも変更する。これにより、最終的にどの開発者がどのような作業をしたか可視化しやすくなる。

Assigneesで指定された人が、そのプルリクエストの責任を持つ。つまり、マージさせるかクローズさせるか最終的に意思決定されるよう、推進する必要がある。

## Reviewers設定はレビュー依頼時にブランクで良い

Reviewersに設定することで、レビュアーが自身のアサインに気が付きやすくするメリットや、後からどれくらいレビューしたか把握しやすくできる。

推奨は以下の通り。

* レビュアーが確定している場合は、レビュー依頼時に予め必要な数のReviewersに設定しておく
* チームのだれがレビューするか不明な場合は、ブランクにする（候補者を片っ端からReviewersに設定しない）
* CODEOWNERSで指定したファイルを編集したなどで、自明であればオーナーをReviewersに追加しておく

## CIの成功を確認する

ローカルでLinterやテストが通っていると、プルリクエスト作成後、すぐにレビュー依頼を行いたくなる。しかし、何かしらの理由でレビュアーが確認した際にCIによるチェックが失敗している場面もよく見る。

推奨は以下の通り。

* レビュー依頼を出す前にCIが通っていることを確認する

理由は以下。

* レビュアーとして、最初のフィードバックは「CIが落ちているので修正して欲しい」となり、余計なやり取りのホップが発生するだけであるため

## レビュー依頼時にURLを貼る

プルリクエストのタイトルや、#511 などの番号だけで依頼を出すのではなく、URLをSlackなどチャットに貼ることを推奨する。

理由は以下。

* 齟齬を生じさせない依頼が可能となる
* 後々、該当のプルリクエストのURLでSlackなどチャット検索が可能とするため
  + 前提条件など、Slackのスレッド上で確認しているやり取りの有無を確認したい場面がある

## レビュー依頼はメンションを飛ばす

コードレビュー待ちは他作業が捗らないことが多いし、別作業に移ったとしても集中したタイミングでレビューフィードバックがあり、集中が途切れることもしばしばある。

推奨は以下の通り。

* チャット上でのレビュー依頼はメンションを飛ばす
  + なるべく早くレビューしてもらうことで、待機時間を最小化する
  + レビュー可能なタイミングの回答をもらい、別タスクに着手するかの計画に活かす
* メンションは、自チームが必要最小限所属する、Slackでいうユーザーグループを利用することが好ましい

::: tip レビュー反映後の連絡 レビュー対応が完了した場合も、再依頼が必要な場合はメンションを飛ばす方が良い。レビュアーとしても再度見て欲しいのかどうかは関心事であるためである。 :::

## Markdown記法を活用する

Markdown記法を活用することで、Descriptionやレビューコメントを視覚的に整理し、意図を明確に伝えることを推奨する。

特に、見出しやコードブロックやリストを使用することで、よりわかりやすいコメントを提供する事ができる。

::: info 参考 [基本的な書き方とフォーマットの構文 - GitHub Docs](https://docs.github.com/ja/get-started/writing-on-github/getting-started-with-writing-and-formatting-on-github/basic-writing-and-formatting-syntax) :::

## マージはレビュイーが実施する

[Gitブランチフロー規約](/documents/forGitBranch/git_branch_standards.html#%E3%83%9E%E3%83%BC%E3%82%B7%E3%82%99%E3%81%AF%E3%81%9F%E3%82%99%E3%82%8C%E3%81%8B%E3%82%99%E8%A1%8C%E3%81%86%E3%81%B8%E3%82%99%E3%81%8D%E3%81%8B)で記載された推奨ルールに従う。チームの自律性と生産性を重視するのであれば、レビュイーがマージすべきである。

信頼関係が構築できていれば、以下のようなやり取りが可能となり、スピード感が上がるためである。

* 「コメントはしたけど承認（Approve）したので、随時対応してマージしてOKです」
* 「Conflictだけ対応したら、マージしちゃってください」

何かしらの理由でレビュアーの関与度を増やす必要がある場合は、レビュアーがマージする。

::: tip やり取りを減らすために、レビュアーが即マージする typo修正など大したことがない修正であれば、やり取りを減らすためレビュアーが直接マージまで行った方が良い場面もある。 :::

# レビュアーの推奨行動

## グラウンドルール

**レビューはコミュニケーションである。コメント記載時には相手への敬意を忘れないこと。**

特にレビュアー側はその責務上、指摘コメントを多く行う役割であり、より一層の配慮（上下関係を作らないやりとり）が求められる。

レビューは作成者を詰問する場ではなく、より良いものをチーム一丸となって作るという考えが重要である。

::: info 参考 [プルリクエストを見る時、出す時に重要なマインドセット - NRIネットコムBlog](https://tech.nri-net.com/entry/important_mindset_for_pull_requests) :::

## レビュー依頼へのリアクションはできるだけ即レス

推奨は以下の通り。

* コードレビューの対応優先度を上げる
* もし、利害関係者との会議中など、即座に対応することが難しい場合は、いつごろ確認するかのコメントを記載したり、代理でレビュー可能な人に依頼する

理由は以下。

* レビュアーが気がついていないのか、気がついているが忙しくて対応できないかで、レビュイーがリマインドを送るかどうかの行動に影響があるため
* レビュイーの作業見積もりに影響するため（レビュイーとして待機すべきか、他の作業を着手すべきか迷ったり、どれくらいで開発作業がクローズできそうかといった作業計画にも影響するため）

::: tip コードレビューの優先度を上げる

チーム開発において、レビュー待ちでの待機時間を最小化することでチームでの生産性を最大化できるとされる。

::: info 参考 [どうして手を動かすよりもチーム内のコードレビューを優先するべきなのか - Speaker Deck](https://speakerdeck.com/okashoi/dousiteshou-wodong-kasuyorimotimunei-nokodorebiyuwoyou-xian-surubekinanoka) :::

## Reviewersを自分に設定する

プルリクエストに対して、レビューを実際に行う場合はReviewersに自分を選択する。以下の理由がある。

* 後からどのプルリクエストに対してレビューを実施したか、トレース／可視化しやすくする
* 他のレビュアーが、◯◯さんが見るなら終わってからレビューを行う／スキップするという判断材料を提供する

## hotfixの場合はマージ先のブランチを確認する

通常のプルリクエストであれば、featureブランチはデフォルトブランチ（多くはdevelopブランチ）がデフォルトで選択されるため、間違えは少ない。一方でhotfixブランチの場合は、main、developの2ブランチにマージ先がありえるため間違えやすい。

特に慣れていないメンバーの場合は作業ミスも多発しやすいため、最初にマージ先のブランチが正しいか確認するクセを付けると良い。

::: tip 複数のdevelopブランチがある場合も間違えやすい  
複数のリリースバージョンを並行して開発する場合、一時的に develop、develop2 と複数のアップストリームを保つ場合がある。このケースは特にマージ先を間違えやすいため注意する。  
:::

## 「Start a review」と「Add single comment」の使い分け

GitHubでのコードレビューは、Start a review を行うと Submit review しない限り、レビューコメントが自分以外のメンバーが参照できない。

Add single comment の場合は投稿が即参照可能となる。

推奨は以下の通り。

* なるべく早くレビュイーにフィードバックを返すことを重視し、 Add single comment の利用を推奨する

理由は以下。

* レビュイーとして、今レビューをしてもらっていると気付くことができ、心理的に楽になる
* コメントを貰った部分の疑問点を即返信しレビュアーにフィードバックしたり、随時修正できる

## レビュイーの思考を予想する

レビュアーは、レビュイーが気がついていないかもしれない観点や影響範囲を意識して見るべきである。

レビュイーの経験やスキルセットから、不足していると考えられる面を重点的に確認することで、レビューの質を上げることができる。

## 暗黙知を形式知にする

レビュー対象の変更内容が、機能上は問題なくても、他の実装方針と揺れている場合が存在する。

推奨は以下の通り。

* レビュイーに「機能上は問題ないが、保守性／可読性の観点から、他の設計方針に合わせてほしい」といった旨のフィードバックを行う
* 開発規約にその設計方針が未記載の場合は、レビュアーが追記する（Linterに追加できれば理想である）。追記内容のレビュアーに、先ほどのレビュイーを含めることが望ましい

暗黙知を形式知にすることで、チームの方針が明文化され、透明性のある運営に繋げることができる。新規参画者にとってのオンボーディング負荷が下がり、戦力化が早まるメリットがある。

## レビューコメントにラベルを追加する

レビューコメントにはラベル（プレフィックス）をつけることを推奨する。レビューコメントの意図を明確にすることで、開発者間の非同期的なコミュニケーションを円滑にし、作業効率を向上させる。

以下にプレフィックスを利用した例を示す。

| プレフィクス | 説明 | 例 |
| --- | --- | --- |
| LGTM | Look Good To Me（良さげです） | [LGTM] 責務分解が明瞭で、良い実装です！💯 |
| ASK | 質問 | [ASK] この実装意図が理解できず、どうして必要なのか教えてもらえますか❓️🤔 |
| Q | ASKと同じ意味 | [Q] 同上 |
| MUST | 必須 | [MUST] ソート処理が漏れているため、順序が不正になってしまう可能性があります！ |
| SHOULD | 提案 | [SHOULD] 愚直なループではなく、□□□の書き方の方が可読性および性能が良いです。書き換えを検討してください。 |
| IMO | In My Opinion（自分ならこうしますが、どうでしょうか？） | [IMO] スコープの広さの割に変数名が短いため、 name ではなく pre\_ordered\_product\_name などが良いかと思いました |
| NITS | Nitpick（枝葉ですが直して欲しい） | [NITS] この変数名は、単数形よりも複数形 users または usersList が適しているかと思います。 |

それぞれの定義は以下。特に「ASK（Q）」と「MUST」はレビューコメントが解決しないと、プルリクエストの承認を行うべきではない。

| 分類 | プレフィクス | 承認のために解決が必須 | レビュイーの対応 |
| --- | --- | --- | --- |
| 礼賛 | LGTM |  | リアクション不要 |
| 質問 | ASK | ✅️ | 回答が必須 |
|  | Q | ✅️ | 同上 |
| 修正依頼 | MUST | ✅️ | 対応しない場合はコメントバックまたは、コード変更の対応が必須 |
|  | SHOULD |  | 対応しない場合はコメントバックまたは、コード変更をできる限り実施する。対応見送りの場合は、チケット起票および、次リリースまでの対応が目安 |
| **提案** | IMO |  | 絵文字リアクションやコメントバックが推奨だが、対応の是非はレビュイーが判断する |
|  | NITS |  | 同上 |

参考: [コード品質向上のテクニック：第51回 確信的な質問 | LINEヤフー](https://techblog.lycorp.co.jp/ja/20241121icq) にも、レビューアの「要求」であるのにもかかわらず、単なる「質問」にも読み取れてしまうミスコミュニケーションの例があり、 コードレビューにおいては要求と質問は明確に区別できるように書かれるべき とある。

## 適度なユーモアを取り入れる

心理的安全性を担保しつつレビューを楽しい場とするためには、ポジティブな絵文字（🚀✨️💯）を使うとともに、レビューコメントに適度にユーモアを入れると良い。

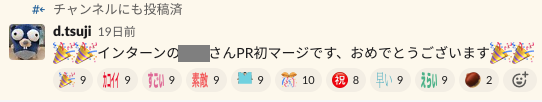
例えば、慣れない文化・チーム・技術スタックの中で成果を出していきたい新規参画者にとっては、どんなに論理的・建設的な内容であるレビュー指摘であっても、指摘が多いと精神的に辛くなってしまう傾向にある。そのため、何か良い点があればオーバーに褒めることや、ネガティブな内容であればちょっとしたジョークを入れることで、心理的安全性を上げていく必要がある。辛い場面であっても、辛い場面こそジョークで吹き飛ばしていけると良い。

以下に例を示す。

* 「このチューニングが入ると操作性がチート並に爆速になって、お客様も喜んでくれそうですね🐆💨」
* 「さすがに年齢設定の上限が1000になっています。鶴は1000年生きると言いますが、ここは150までにしておきましょう🧝」

::: tip 初マージはチーム全員で祝う

苦労すればするほど、初めてマージされる瞬間の喜びは大きいものである。基本的に新規参画者の初めてのマージはチーム全員で祝うと良い。



:::

## 指摘内容は具体的であればあるだけ良い

レビューコメントで「可読性が低いです」 といったコメントは、レビュイーが受け取るとどのように修正して良いか判断ができず、次のアクションに繋がりにくいため不毛である。そのため、面倒であっても、どのように直して欲しいか、できる限り具体的に指示する方が建設的である。

推奨は以下の通り。

* 「可読性が低いです」 「{任意の非機能要件}を満たしていません」といったコメントで終始せず、具体的にどの部分を、どのように修正して欲しいかをフィードバックする
* もし、修正する部分を（教育目的などで）レビュイーに考えてほしい場合は、その旨をコメントに追記する。レビュアーのお気持ちあてゲームにしないこと

## 指摘内容を論理的に説明できないのであれば、コメントすべきではない

「なんとなく汚いです」は指摘ではない。悪いコードだと感じたのであれば、相手が納得できるように論理的に説明する。論理的に説明できないのであれば、指摘をすべきではない。

::: info 引用元 [デキるプログラマだけが知っているコードレビュー7つの秘訣 | PPT](https://www.slideshare.net/rootmoon/7-37892729) p26 :::

## 出典があれば追記できるとベター

指摘内容の補足情報があれば追加するとベターである。例えば、「◯◯の観点で、このコードは△△の懸念があるため、□□□になるように修正をお願いします。公式ブログの{URL}にも記載があり、参考にできると思います。」といった形式である。

レビュイーが自分で調べれば事足りる場合もあるが、レビュアー側が出典を示すことで納得感が増す。また、誤った記事（過去のバージョンの記事や、見当違いの記事）を読んでしまい空回ってしまうことも、探す手間も減る。

## なるべく1度のレビューで指摘し切る

レビュイーからすると、レビューの「依頼1→指摘1→対応1→確認依頼1→別の指摘2→対応2→確認依頼2→別の指摘3→…」と、依頼のたびに指摘をもらうとタスクの予実管理も作業見積もりも建てられずストレスである（ムービング・ゴールポストである）。

推奨は以下の通り。

* できるかぎり最初のレビューで、指摘事項を出し切る
* もし、2回目以降のレビューで追加要素に気がついた場合は、「[imo] 1度目の指摘で見逃してしまってすいません。この部分ですが、～の観点のため修正してもらいたく。」などと非を認めつつ、コメントする

## 本筋から離れるコードコメントやREADMEのtypoなどはsuggestにする

プルリクエスト上のコメントで、レビュイーに対して直接変更内容を提案することができる[Suggested changes](https://docs.github.com/ja/pull-requests/collaborating-with-pull-requests/reviewing-changes-in-pull-requests/incorporating-feedback-in-your-pull-request)という機能がある。

推奨は以下の通り。

* 可能であればできる限り、Suggested changes形式を利用する。特にtypo、ドキュメントの言い回しや表現の訂正は、直接的に変更を指示しないと、やり取りのホップ数が増える傾向にあり、非生産的である
* 新規参画者やジュニアメンバー向けに、教育目的で編集させたいと考える場合もあるが、何度かSuggested changesでフィードバックして直らなかったなど、限定的な場面で行う

Suggested changesを使わないほうが良い場合もある。

* 変数名のリネーム（一箇所だけ変更してもビルドエラーになり、その後のリファクタリングが余計に面倒であるため）
* 上記のリネームの例のように、リンターやテストに影響を与えるものは、レビューコメントという形でコメントする方が良い

::: info 参考 [君は GitHub の Suggested change を知っているか？ - BASEプロダクトチームブログ](https://devblog.thebase.in/entry/2018/12/02/130657) :::

## nits（WANT以下）のコメントだけであれば、先にapproveする

対応が必須ではない、いわゆるnitsな指摘事項だけの場合がある。

この場合の推奨は以下。

* さきにapproveしてから、「コメント部分を適時対応してマージして下さい」と伝える
  + レビュイーの心理的負荷の軽減（レビュイーの裁量が増し信頼感が作れる）
  + 余分なコミュニケーションが減り、スピード感が上がる
* 先にapproveすることにより、nitsであるがその対応可否をまったく検討せず、セルフマージするようなメンバーがいた場合は、対応状況を見極めてからapproveしても良い
  + つまり、初回、2回目は信頼してapproveすべきである

## レビュアーはローカル環境で動作確認をすべきか

レビュー対象のリモートブランチを、ローカル環境にてテストなど動作確認すべきか迷う場合がある。レビュアーがローカルで動作検証を行うと、おそらく品質は高まるはずだが、レビュイーとの作業重複でもあるため工数は非常に高くなり、費用対効果は必ずしも良いと言えない。

推奨は以下の通り。

* レビュアーはローカル環境での動作確認はしなくても良い
  + もちろん、しても良いがMUSTではない
  + CIで単体テストが成功／失敗しているかは確認できずはずである
  + どのように品質を担保するかはテスト戦略に依存させるべきである（その後のデプロイメント環境での結合テスト、システムテスト、性能テスト、障害テストなどに含まれるはずである）

::: tip 動作確認はレビュイーにやってもらう プルリクエスト作成ルールとして、CIで担保できる部分以外（例えばUIのスクリーンショットなど）は、 [画像をエビデンスとして貼るべきかどうか](#画像をエビデンスとして貼るべきかどうか) なども参考にしつつ、レビュイーにエビデンスの添付を行ってもらうルールを考えるとよい。 :::

## 疑問点は臆せず質問する

レビュアーは意図が掴みきれない部分があった場合は、そのままにせず理解できるように務める必要がある。分かったふりをすると、後でその部分が火を噴くことが多いからである。また、「チーム全員が同じ知識を共有する」というコードレビューの意図からすると、レビュイーだけ分かっていればよいという状態は不健全である。レビュイーがリーダー（テックリード）であっても臆する必要はないし、レビュイーが自分より若手であったとしても、自分が調べても分からないことを聞くことは恥ではないので、堂々と教えてもらう。

推奨は以下の通り。

* 不明点が少ない場合は、レビューコメントで質問する
* 不明点が多い場合は、同期的なコミュニケーション（リモート会議。SlackのハドルやGoogle Meet）で確認する。やり取りのホップ数をなるべく減らすことを意識する
* リモート会議を行う場合は、レビュアー側から時間や枠を調整する

## 自信がない場合は、有識者にディスパッチする

レビュアーという立場だからといって、必ずしも開発チームが利用する全技術スタックや、開発プロダクトの過去の経緯に精通している訳ではない。それでも、自信が無い領域についてコードレビューを求められるケースも多々ある。

推奨は以下の通り。

* 自分に自信がない領域については、レビュアーが無理せず有識者にディスパッチする体制にする
  + 分からないことは恥ではない。それより、知ったかぶりで手違いが発生する方がチームにとってのリスクである
* 業務要件／機能要件が不明確であれば、プロダクトリーダー（業務担当者）に移譲する。非機能面で自信が無ければ、アーキテクトに移譲する。チーム規模によっては恒常的にこれら2つの役割のレビュアーで分担してレビュー方式も考えられる

## ジュニアメンバーに対して大量の指摘がある場合にどうするか

対象領域のスキル不足などが原因で、レビュー事項が大量にある場合に対応をどうすべきか迷う場合がある。

推奨は以下の通り。

* 50でも100でも指摘事項があるのであれば、全てコメントを付ける
  + 最初に全量を提示することで、レビュイーに対して作業見積もりを可能とさせる。逆に最初にコメント領域を絞ると、ムービング・ゴールポストである
  + もし、作業負荷的に厳しい場合は同期コミュニケーションを入れることをためらわない
* 指摘事項が多い場合、レビュイーが落胆してしまう懸念があるため、特に新規参画者で信頼構築ができていない場合は、コードレビューとは別にケアする
  + 指摘事項が多くても問題ない旨を、チャット上でフィードバックする
  + 指摘事項の背景情報（開発規約や、採用技術）について説明する会議をセッテイングする（適度に同期的なコミュニケーションを取る）
  + ペアプロで指摘内容を修正する
* 可能であれば、Suggested changesを用いて、レビュイーの負荷を下げる工夫を取る

なお、類似の指摘がある場合に、省略するかどうか判断に迷う場面もある。これについては以下を推奨する。

* 「ある1点だけ取り上げて、横展開をお願いします」は機能しないため、該当箇所を全てコメントする
  + そのプルリクエスト内でレビュイーがキャッチアップしきれない可能性があるため、レビュアー側で補助した方がベター
  + 横展開が漏れることが多く、余計なやり取りが発生し、レビュアー・レビュイーともにストレスになることが多いため

## レビュー終了の連絡

レビュー終了した場合、レビューが終わった旨をプルリクエスト上でメンションをつけてコメントしたり、Slackなどのチャットで通知することで、レビュアーに対応を促すことができる。「Add single comment」の場合は、レビュー終了の切れ目が分からないためである。

推奨は以下の通り。

* Slackでのメンション付きでレビュー終了の旨のコメントする

理由は以下。

* 普段利用しているチャットツールの方がレビュイーの気づく確率が高いため

# 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**（真野隼記、村田靖拓、山本竜玄、清水雄一郎、水田祐介）
* **レビュアー**（谷村元気、栗田真、後藤玲雄、八木雅斗、辻大志郎）

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。