git

Git とは

Git(ギット) とは、プログラムソースなどの変更履歴を管理する、分散型のバージョン管理システムのことです。

バージョン管理システムとは

バージョン管理システムでは何ができるのか、代表的な機能を記載します。

ファイルに対して変更履歴が残せる

特にソフトウェア開発においてソースコードの管理に用いられることが多いですが、EXCEL や画像などファイル全般を管理できます。
変更はすべてバージョン管理されており、バージョンごとの情報を取り出すことができます。

変更履歴を共有できる

サーバーにて変更履歴が管理されており、すべての開発者は同じコードを共有することができます。

以前のバージョンに簡単に戻せる

変更履歴が残されているので、指定したバージョンやファイルに戻すことができます。

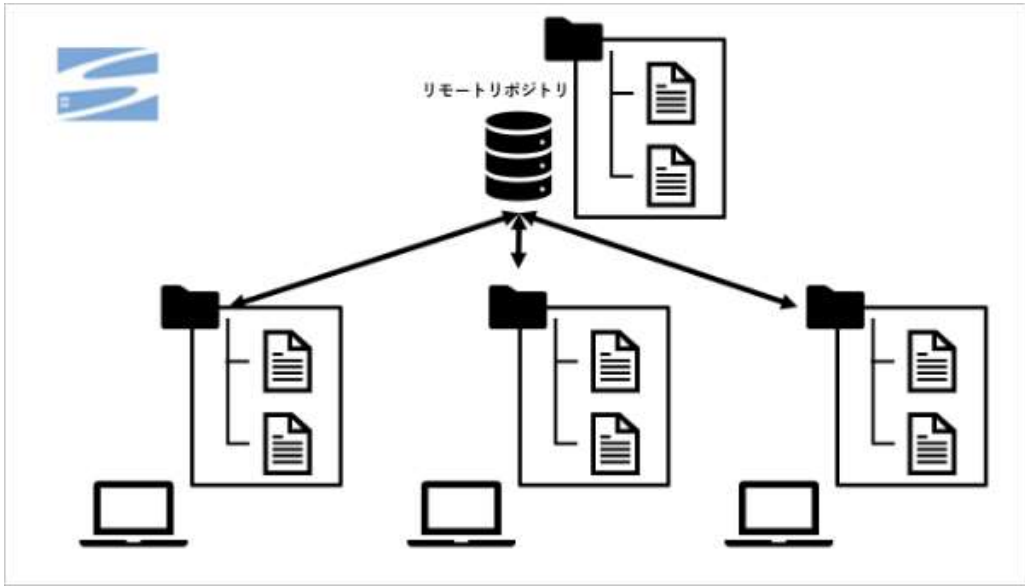
分散型と集中型

今までよく使われてきた Subvertion などは、**集中型** バージョン管理システムと呼ばれます。
それに対して git は、**分散型** バージョン管理システムと呼ばれます。

集中型

リモートリポジトリだけを持ちます。

開発者はリモート環境のリポジトリにアクセスし、ローカル環境にあるファイルをコミット(アップロード)したり、ローカル環境へチェックアウト(ダウンロード)したりします。



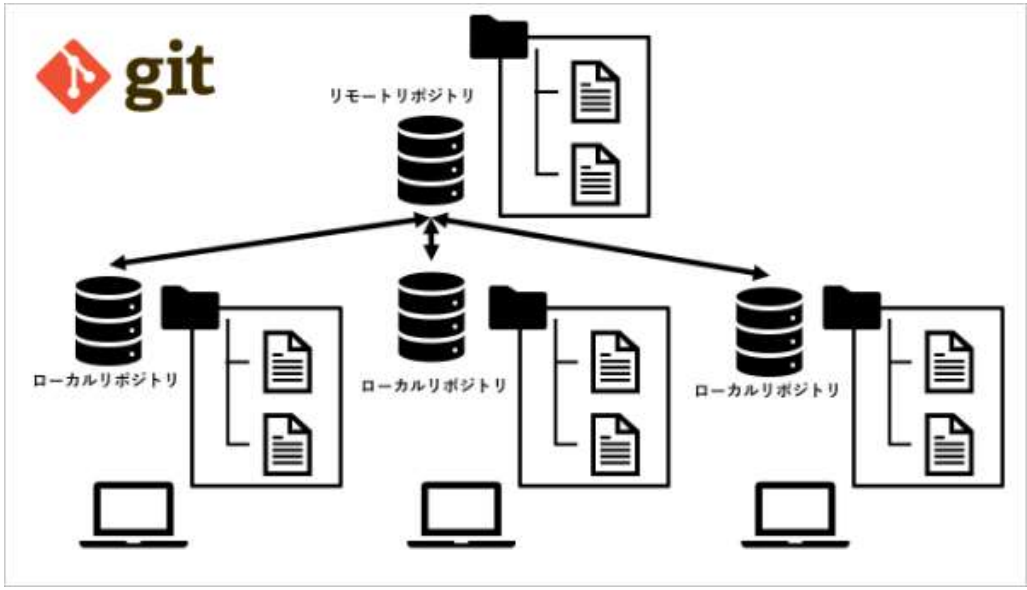
有名な集中型バージョン管理システムを下記に記載します。

- Subverstion (SVN)
- CVS

分散型

リモートリポジトリと、そのクローン(コピー)として作成するローカルリポジトリを持ちます。

開発者はローカル環境のリポジトリに対し、コミットやチェックアウトを行います。
ローカル環境でコミットしたファイルについては、適切なタイミングでプッシュ(アップロード)を行い、リモート環境のリポジトリへ反映させます。
また、ローカル環境のリポジトリを作る際は、リモート環境のリポジトリから特定のバージョンの情報をプル(ダウンロード)で取得します。



クラウドサービス・セルフホスト

git を使用する場合サーバアプリケーションが必要となり、クラウドサービスを利用するか、自前のサーバにセルフホストする必要があります。
選択肢となる有名なサービスを記載します。

サービス	クラウド	オンプレミス
GitHub	●	×
GitLab	●	●
BitBucket	●	●
GitBucket	×	●

選び方

クラウドサービスを契約するのか、それともセルフホストするのか、それぞれのメリット・デメリットを記載します。
基本的には、**ランニングコスト** と **ポリシー** で決まるかと思われます。
特にポリシーでクラウド環境の使用が禁止されているような場合は、セルフホストしか選択肢は無くなるでしょう。

クラウドサービス

メリット

- 自前で準備する物が無い
- サポートが受けられる

デメリット

- 月額料金がかかる
  - サービスによっては無料プランが有るが、プライベートプロジェクトとしてするなら現実的ではない
    とはいえ、例えば GitHub では 2019/11/22 現在 1 ユーザあたり月額 1000 円くらいです
- サービスでの障害発生時は、ひたすら解決を待つしかない
  - 以前の障害発生時は 1 日で解決しました
  - ローカルリポジトリが有るので、何もできない訳では無い

オンプレミス

メリット

- OSS なら、無料でセルフホストできる物がある

デメリット

- サーバを自前で準備しなければならない
- 運用コストがかかる
  - セキュリティパッチの適用など
  - バックアップを取るなど
  - 当然ながら保守費用がかかるので、オンプレミスなら無料という訳ではない
- トラブル時には自力での解消が必要

git の機能

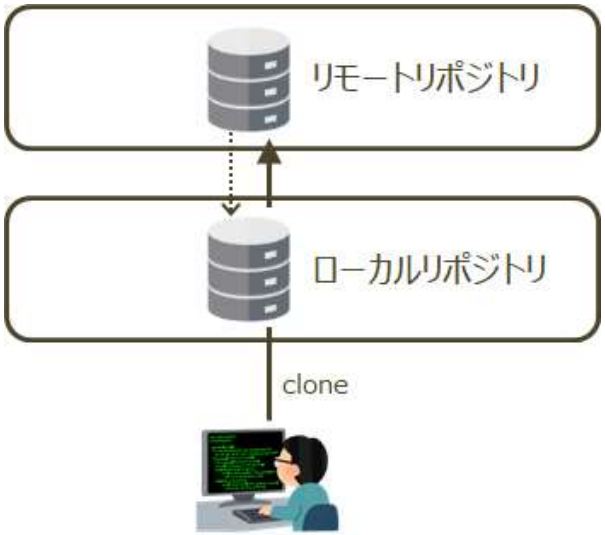
git の基本的な仕組みと機能・用語を紹介します。

repository

ファイルの変更履歴の格納場所です。
サーバに置いて共有するための **リモートリポジトリ** と、自分の端末に置いて作業するための **ローカルリポジトリ** があります。

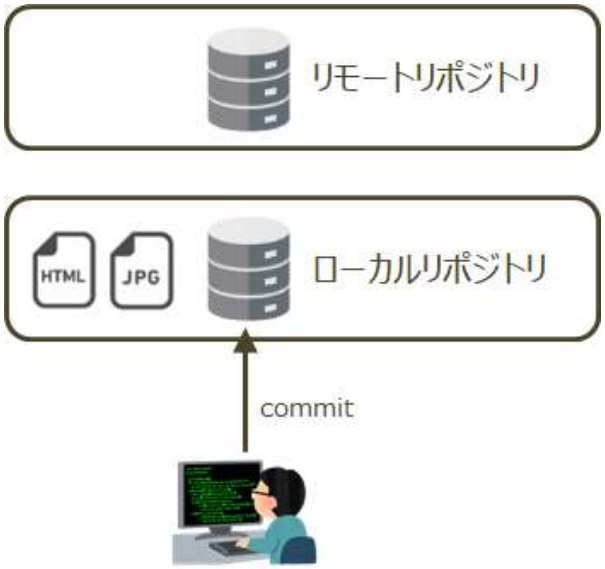
clone

リモートリポジトリ をコピーして、**ローカルリポジトリ** を作成します。



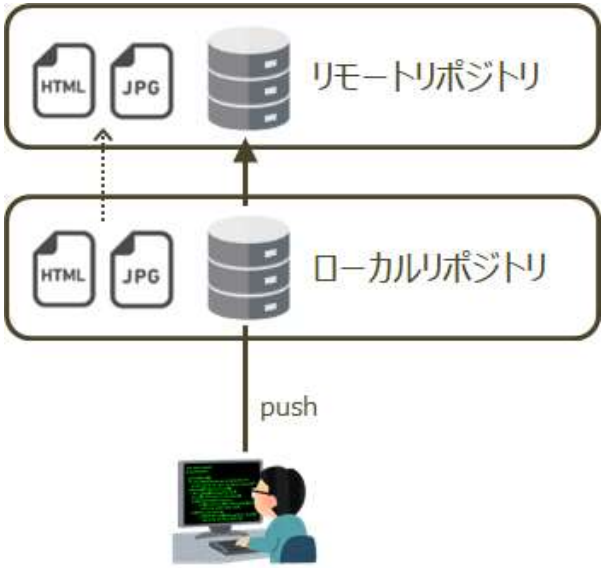
commit

自分の **ローカルリポジトリ** に登録します。



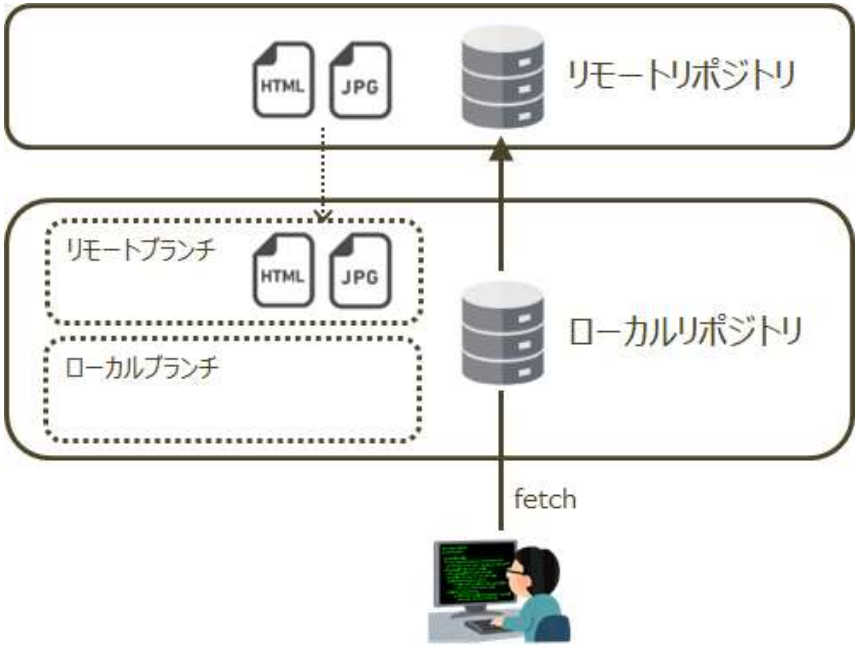
push

自分の **ローカルリポジトリ** の変更履歴を、**リモートリポジトリ** にアップロードします。



fetch

リモートリポジトリ から最新の変更履歴をダウンロードしてきて、自分の **ローカルリポジトリ** にその内容を取り込みます。
ただし、コマンドは他のリポジトリのデータを取得するだけで、ローカルで作業しているファイルを書き換えたりマージしたりするようなことはありません。

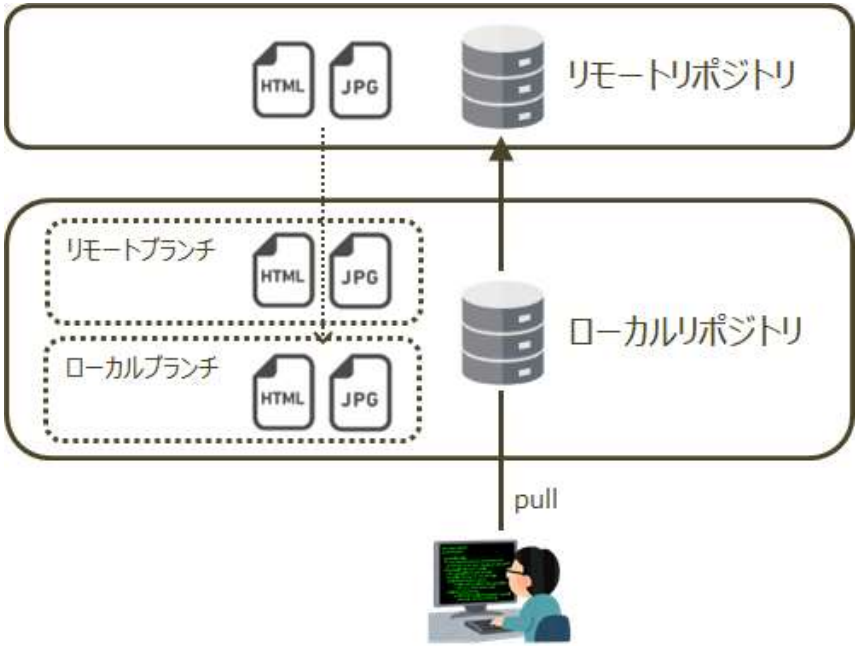


merge

特定のブランチやリビジョンを、現在のブランチに取り込みます。

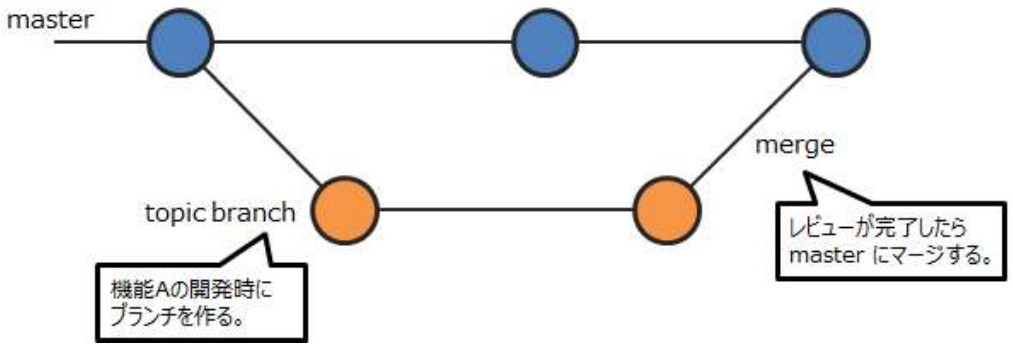
pull

fetch と merge の両方を組み合わせたコマンドです。
つまり、fetch で最新の変更履歴をダウンロードして、merge で現在のブランチに取り込みます。



ブランチとは

ブランチ(branch)は、1 つのプロジェクトから分岐させることにより、プロジェクト本体に影響を与えずに開発を行える機能のことを言います。
ブランチは直訳すると「木の枝、支流、支系」の意味となります。
現行のバージョンのプロジェクトから枝分かれさせて他の作業を行うときに使われます。



ブランチ戦略

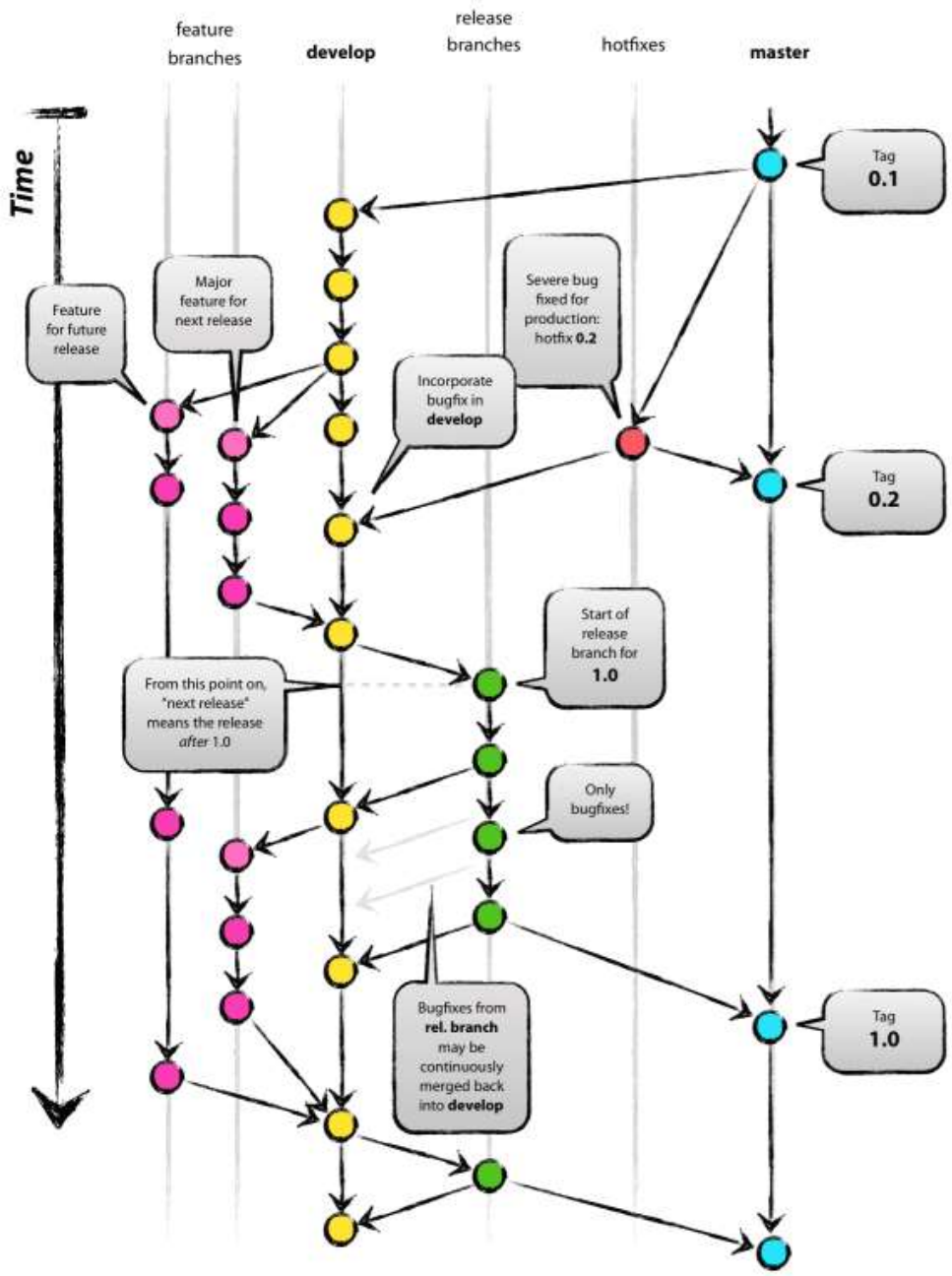
開発を開始するにあたり、まずはブランチ戦略を決める必要があります。

ブランチ戦略とは、「機能Aと機能Bの開発を同時に進めたいが、各々影響を与えあわない状況下で同時に開発を進めたい」、といった場合に活躍する仕組みの事です。

有名なブランチ戦略を、下記に記載します。

ブランチ戦略	説明
git-flow	リリースを中心に考えられており、5 つのブランチを使用します。
GitHub Flow	master ブランチと feature ブランチだけで進めるモデル。
GitLab flow	Git-flow の複雑さや GitHub flow のリリース周りの問題点を指摘して、それらを解決しようとしたもの。

例として、git-flow について説明します。



ブランチ	削除	説明

ブランチ	削除	説明
master	-	常にリリース可能な状態。
hotfix	●	緊急対応用。master から分岐して、develop と master にマージ。
release	●	リリース作業用。develop から分岐して、develop と master にマージ。
develop	-	次のリリースのための最新の開発作業の変更を常に反映する。
feature	●	新機能開発用。develop から分岐して、develop にマージ。

プロジェクトに適用するには

- 最初に決めたブランチ戦略にこだわる必要は有りません
- プロジェクトの状況に応じて、都度見直しましょう
- 開発体制や開発サイクルなどによって、最適なものを選びましょう。
  小規模なプロジェクトで複雑なフローを選択すると非効率になったり、逆に大規模なのにキチンとしなければ、修正内容などが不明確になってしまいます。
- 少なくとも、ブランチは使用しましょう。
- まずはシンプルなフローから初めてみましょう。
- 極端な話し、「master ブランチは常にデプロイ可能」といったルールされ守れば大丈夫です。

クライアントツール

クライアントから git を操作できるツールは多々ありますが、メジャーなツールを 3 つ選んでみました。

ツール	UI	料金	日本語
Git Bash	CUI	無料	△
TortoiseGit	GUI	無料	●
SourceTree	GUI	無料	●

出来るだけ、GitBash の使用を推奨します。
その他の GUI ツールを使った場合、実際の流れが分かりづらく、中々覚えられません。

必ずしもプロジェクト内で使用するツールを統一する必要はありませんが、ツールや日本語化によってコマンド名が異なる場合、会話が難しくなってしまいます。

git をより使いこなすために

git そのものの機能では有りませんが、GitHub や GitLab などの有名サービスは便利な機能を提供しています。
有名なものをいくつか紹介します。

pull request

コードレビュー機能です。
修正を元のリポジトリやブランチに取り込む際に、この機能を用いて他の開発者に修正の内容を確認してもらいます。
バグの混入を防いだり、より良いソースコードに修正したりできます。
設定により、pull request 経由でなければ master など特定のブランチへの merge を抑止できたりもします。
※GitLab では merge request と呼びます。

Issues

バグトラッキングシステムです。
redmine のように issue(チケット) を作成できます。
プロジェクトやソースコードの課題を管理することができ、ソースコードやプルリクエスト、修正履歴といったサービス上の情報と、課題を関連付けて管理することができます。

CI/CD

CI(Continuous Integration) は日本語で **継続的インテグレーション** と呼ばれており、ソフトウェア開発において短期間で品質管理を行う手法のことです。
コンパイル・テスト・デプロイといったソフトウェア開発のサイクル (ビルド) を頻繁に繰り返し実行することで、問題の早期発見や開発の効率化などが可能となっています。

CIツールを使うと何ができるのか？

- テストの自動化
  自動テストを決まったタイミングで実行するようスケジュールを組むことにより、手動でのテスト工数を削減することが可能です。またコードに変更があるごとにテストが実行されるため、早期に不具合を把握でき迅速に対応することができます。
- 開発状況の把握
  ビルド結果の履歴を時系列に並べて視認可能な状態にすることにより、ビルドエラーの発生状況や自動テストのテスト結果などの状況を把握することができます。
- ソースコードの静的解析
  コーディングルールやバグのチェックツールを組み込むことにより、開発中のソースコードの問題部分をあぶり出すことができます。また、将来発生する恐れがある不具合の発見や保守性の向上など品質を高めていくことも可能です。
- 継続的デリバリー
  継続的デリバリー (Continuous Delivery) とは、継続的インテグレーションを拡張したもので、コード変更が発生すると、自動的にビルド、テスト、および本番環境へのリリース準備が実行されるというものです。

分散型と集中型のどちらを選ぶべきか

ここまでの説明で、git の特徴は伝わったかと思います。
実際のプロジェクトで **集中型** と **分散型** のどちらを選択するべきなのか、双方のメリットとデメリットを記載します。

メリット

開発中のコードを共有できる

開発ブランチを作成すると、開発途中のコードもチームで共有できます。
Subvertion などでは、個別にファイルを共有するか、一旦リポジトリにコミットしなければなりません。

コードを綺麗に保てる

ブランチを利用しない開発フローと比べて、コードを綺麗に保つことができます。

- 修正途中の中途半端なコードが master ブランチに含まれない
- レビューしていないコードが master ブランチに含まれない

修正ごとに別のブランチを作成することで、「修正・レビューが完了したソースコードのみを master ブランチに取り込む」という流れを守れば、正しい変更のみを master ブランチに取り込むことになります。

レビューがしやすい

ある修正に対して、その修正の範囲の差分なのか関係のない差分なのか判断する必要がありません。
例えば開発ブランチから master へ取り込もうとした場合、ソースコードの差分が表示されます。

他の開発者のミスで影響を受けにくい

master ブランチに取り込まれているのは、レビューを受けて承認されたコードのみです。
Subvertion などでは修正中だったり未レビューのコードがコミットされがちですが、そういったコードが master に入らないので、他の開発者のミスで影響を受けにくいといえます。

デメリット

学習・教育コストが高い

Subverstion などに比べて、覚えなければいけないコマンドがかなり多く、Subverstion に毛が生えた程度でしょ、と思っていると痛い目に会います。
とはいえ、そこまで難しい訳では無いので、大きなデメリットとは言えません。

運用設計コストが高い

ブランチ戦略やコミット規約・マージタイミングなど、ルールを決めないで進めてしまうと、痛い目に会います。
プロジェクトにあった運用ルールを決めるには、知識と時間が必要です。

WORD/EXCEL など、バイナリで管理されるファイルに弱い

バイナリの場合は自動マージができないので、コンフリクトが発生した場合の対応が困難です。
設計書などのドキュメントは、SVN を使った方が良いです。
※ドキュメントを Markdown や PlantUML 使ってテキストベースにするなら、git での管理も可能だと思います。

個人的なまとめ

- 新規プロジェクトなら、規模にかかわらず **git 一択**。
- ポリシーでクラウド禁止なら、**オンプレミス**一択。許されるなら**クラウドサービス**。

この資料について

この資料は、VisualStudioCode を使用して Markdown で記載しました。
参照用として VisualStudioCode から HTML 化を行い、chorome の機能で PDF 化しています。
見出しの青い帯などのデザインは、CSS を適用しています。