研究者のためのGit入門

v1.1.0

DeepFlow株式会社
神志那 純

DeepFlow株式会社

- 九大発のDeepTechスタートアップ（2018年7月設立）
- elkurageの開発・運用
  - 超高速大規模マルチ物理シミュレータ
    - 離散微分幾何を利用した汎用シミュレータ
  - フルスクラッチでゼロから開発（ Haskellで10万行くらい）
  - 6人チームでメッシュ作成からソルバー開発，可視化まですべて対応している
  - 物理や数学の博士も活躍
- 各種解析支援・ITシステム構築
  - elkurage開発で得たノウハウの展開
  - エンジニアリングの知見で研究開発のイテレーションを加速させたい
  - 研究で求められるコードと運用で求められるコードは違う
  - アカデミックな事情をくんだエンジニアリング

YouTubeチャンネル

自己紹介

- 神志那 純（こうじな じゅん）
- DeepFlow株式会社 研究開発部
- 大学は制御系でその後ITベンチャーへ
  - 物理は詳しくない
- 取り組んでいること
  - elkurageの高速化
  - 開発環境の整備
    - Gitリポジトリの整備
    - 自動テスト（CI）の導入
    - リファクタリング

はじめに

- Git
  - Linuxカーネルのソースコード管理のために開発されたバージョン管理ツール
  - デファクト・スタンダード
- なぜGitを使うのか？
  - チーム開発の生産性を飛躍的に高めることができるから
  - GitHubやGitLabという便利なホスティングサービスの存在
    - コードレビュー，自動テスト，issue管理 etc.
  - Gitの導入がモダンな開発体制への最初の 1歩
  - 1人開発でも役に立つ（昔の自分は実質他人）
- 今日のスローガン
  - 「Gitの気持ちを考えてGitを使う」

目次

- バージョン管理
  - どういう課題があるのか
  - 単純な解決策はないのか
  - Gitを作る気持ちで
- Gitのアプローチ
  - バージョン管理の課題をGitはどう解決してるのか？
- Gitを用いた開発の進め方

1.バージョン管理

バージョン管理したくなるシチュエーション

- そもそもどういうことをしたいのか？
  - 「新しいアイデアを実装したが結果は良くなかった．元にも戻せなくなった」
  - 「あのときは動いていたのに，今は動かない」
  - 「バグが見つかった．このバグはいつから？」
  - 「なぜこの処理が必要なんだっけ？誰に聞けばいいの？」
  - 「本当にあのとき動いていたコード？」
  - 「手元にあるコードは本当に最新版？どっちが最新？」
  - 「ちょっとパラメータを変更したが，計算を待ってる間に元の値も忘れてしまった」
  - 「卒論を保存していたUSBメモリが壊れた，PC壊れた」
  - 「この前いれたはずの修正が消えてるんだが，一体何が？」
  - 「今日はすごい頑張った．何行くらい書いたんだろう」
  - 「昨日入った変更で結果が改善，一体誰がどんな変更を？」
  - 「新しくチームに入った彼，ちゃんと開発できてるかな？」

バージョン管理に求められる機能

- ファイルの状態を過去に戻すことができる
- ファイルの状態を表すメタデータ（5W1H）
- メタデータからの状態の検索
- バージョン間の比較ができる
- 他人との連携・共有
- 現実的な時間・データ容量で実現できる

モックの検討

- そもそもバージョン管理は特別なツールはいらないのでは？
  - 一般にツールの導入にはコストがかかるので検討する必要がある
- ツール無しで運用するシンプルなアイデア
  - バージョンが変化するたびに新しくディレクトリを作成してそこにコピーする
  - ディレクトリの中にCHANGELOG.txtを作成し，変更履歴を追記していく
- 例
  - 20220626
    - prog.c
    - CHANGELOG.txt
  - 20220627
    - prog.c
    - CHANGELOG.txt
- このアイデアうまくいくのか？

ファイルデータの持ち方

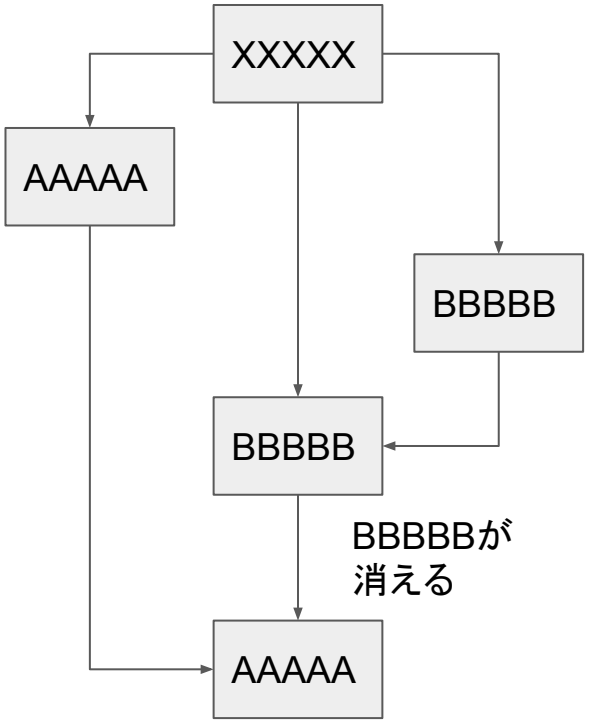
- シンプルでわかりやすい
  - バージョンの付け方がわかりやすい
  - ディレクトリ名を見るとどれが最新かがわかる
- 全部コピーするのはデータ効率が悪い
  - ほとんどのファイルは変更されない
  - zipで圧縮？
- ファイル権限
  - 操作ミスでファイルが変更されないように書き込み禁止にする
- 日付管理のディレクトリが歯抜けにならないか？操作ミスと区別がつくか
- 研究不正としての過去のコード・データの改竄

バージョン番号(yyyyMMdd)

- バージョン間の関連性がわかりやすい
  - 2つのバージョン新しいのはどっち？
- 1日に1回しか変更できない
  - 日時を含める？タイムゾーンは？フォーマットは？日付 +連番？
- 複数名で開発するときにバージョン番号が衝突する
  - A氏の20220627とB氏の20220627はバージョンが同じでも中身が違う
- 独自にバージョン番号の拡張を運用し始める可能性
  - 20220627-kohjina-0001
  - バージョン間の関連が管理できなくなる
    - 何をベースに変更したのかわからなくなる
    - 古いコードをベースにしてしまい，それ以降の変更が消える
- 昔のコードを微修正して動かす時そのバージョンはどうなる？
- 単純なことだが意外に難しい

複数名開発

- 共有サーバーでバージョンデータを共有
- データの競合
  - XXXXXをベースにA氏が実装を開始
  - XXXXXをベースにB氏が細かい変更を加える
  - バージョンがBBBBBになる
  - A氏が実装を終えて変更を加える
  - バージョンがAAAAAになる
  - A氏はXXXXXをベースに実装したのでBBBBBの変更が消失
- 他の人の変更を待たないといけない
  - チームで開発してるのに生産性がまったく上がらない



変更履歴

変更履歴を記録したファイルをプロジェクトに設置

- 人間は楽をするので，誰も何も書かなくなる可能性
  - いつ誰が何をしたのかという情報が残らなくなる
  - 変更がこれだけなのかの保証ができない
  - タイポを直しただけで記録を残すのか？
  - タイポを修正するだけで動かなくなることもあります
- 履歴が積み重なると情報の検索が困難
  - コメントのフォーマットを統一してないと機械処理できない
- みんながCHANGELOG.mdを変更すると，データの衝突が多発する

課題のまとめ

- ファイルデータの持ち方
  - 工夫しないとデータ効率が悪い
- バージョン番号
  - 意外に難しい
  - 他人のバージョンとの衝突，バージョン間の関連
- 複数名開発
  - データやバージョンの競合
- 変更履歴
  - 良心にだけ任せているとログが残らない
- 更新頻度が低ければ運用できそうだが，長期間のプロジェクトやチームで開発するには厳しそう
- このスライドの管理みたいなものには使えそう

2.Gitのアプローチ

キーアイデア

- ファイルやディレクトリの内容を**ハッシュ**(SHA-1)で管理する
  - 2283e0e9af55689215afa39c03beb2315ce18e83
  - 履歴データの効率化
  - 各種操作の高速化
  - 分散環境での整合性
- ローカルにもDBをもつ
  - 履歴データは全て手元におくことができる
  - 多くの操作をネットワークを介さずにできる
- 履歴をDAG(Directed Acyclic Graph)でもつ
  - 枝分かれする
  - 合流する

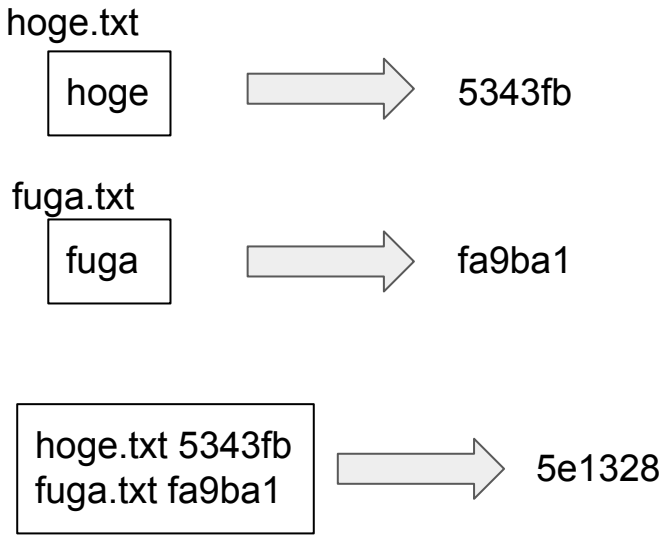
SHA-1ハッシュ

- SHA-1
  - どんなサイズのデータも160bit(20byte)のデータに変換する性質のよい関数
  - sha1(“”)=da39a3ee5e6b4b0d3255bfef95601890afd80709 (16進表記)
  - データをぐちゃぐちゃにして160bitにまとめるイメージ
- 性質
  - 同じデータに対しては誰が生成しても必ず同じハッシュ値になる
  - データが異なれば極めて高い確率でハッシュ値は異なる
    - 現実的にはハッシュ値の衝突は起こらないと考えていいレベル
    - 衝突させた例はある(2017, SHAttered)
    - Gitはハッシュの衝突も考慮に入れた実装になっている
  - データの違いがわずかでもハッシュ値は大きく異なる
    - sha1(“”)=da39a3ee5e6b4b0d3255bfef95601890afd80709
    - sha1(“ ”)=b858cb282617fb0956d960215c8e84d1ccf909c6
  - ハッシュ値のみからは元のデータを復元できない（暗号化ではない）

ハッシュがどう役に立つのか？

- バージョンごとにディレクトリ全体をコピーしてるとデータ効率が悪い
- ディレクトリの情報を2つに分ける
  - ディレクトリの内容を表すハッシュ値
    - 各ファイルのハッシュ値を求める
    - 各ファイルパスとそのハッシュ値のデータを集めたもののハッシュ値を求める
    - これがディレクトリ全体のデータを表すハッシュ値
  - ハッシュ値からファイルデータを復元するためのデータストア（全バージョンで共有）
- どう役に立っているのか？
  - ほとんどのファイルは変化しないので，無駄なコピーを減らしている
    - ファイルが変化していなければハッシュ値も変化しない
  - **ディレクトリ全体のデータを160bitのハッシュ値で表現できる**
  - ハッシュ値をもとにファイルデータを完全に復元できる
  - 他人の生成するハッシュ値との調整機構が必要ない（分散環境に強い）
    - ネットワークに接続しなくても矛盾なくハッシュ値を生成できる

スナップショットのハッシュ化



DB

key	value
5343fb	hoge
fa9ba1	fuga
5e1328	hoge.txt 5343fb fuga.txt fa9ba1

5e1328というハッシュ値からファイルを復元することができる

バージョン管理をどう実現するか

ハッシュを駆使して実現する

- バージョン情報のDB (.git)
- 履歴データ（commit）
- バージョン番号(commit hash/tag)
- 履歴の分岐(branch)
- 履歴の合流(merge/rebase)
- 履歴の共有(remote/push/pull/fetch)

.git

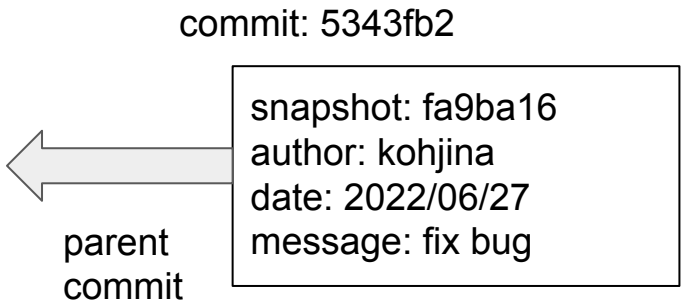
プロジェクト内に作られるバージョン管理のDBのディレクトリ

- バージョン履歴の全データが入っている
- Git用に特別にデータベースを用意する必要はない
- サーバーの履歴データが消えてもローカルのデータで復元できる
- .gitの中身
  - HEAD
  - config
  - description
  - hooks
  - info
  - objects (ここにハッシュ値とファイルのデータが入る)
  - refs

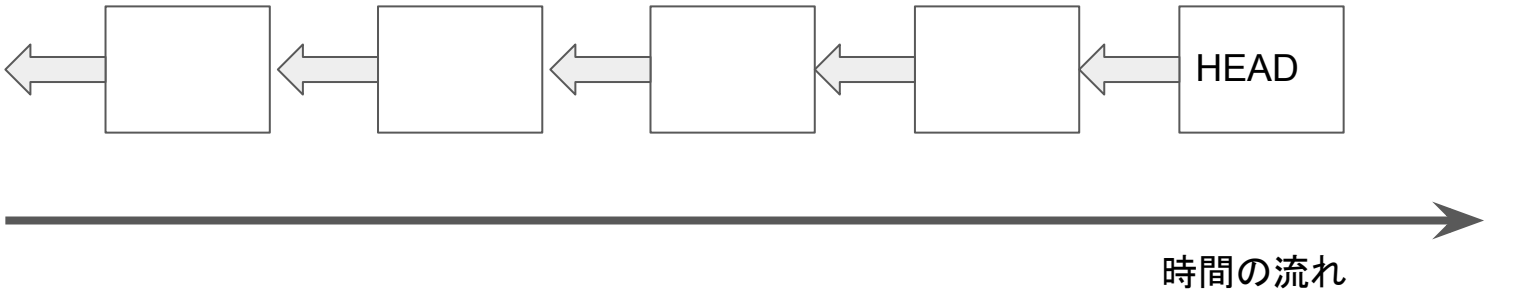
コミットオブジェクト（履歴データ）

- Gitでバージョン管理する上で1番基本になるデータ
- 全ての操作をコミットで表現する
- コードを変更するたびにこのデータを生成することになる
- データの内容
  - ディレクトリのデータ全体（スナップショット）を表すハッシュ値
  - 親コミットを表すハッシュ値（どのバージョンをベースに変更したのか）
  - 日時
  - コミットデータの作成者
  - ファイルを変更した人
  - コメント
- 何をベースにして最終的にどうなったのかというデータ
  - 差分のデータでもっているわけではない

コミットオブジェクト



親をたどっていくと歴史がわかる



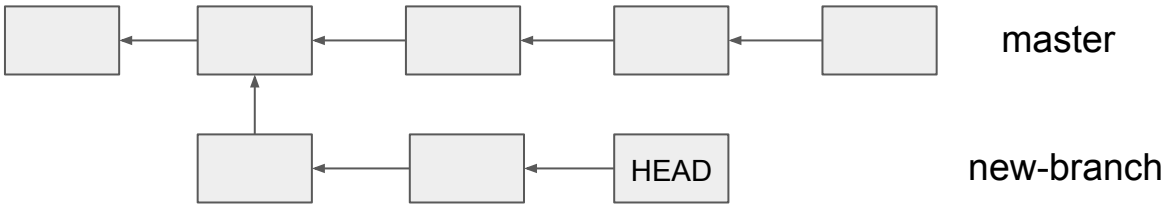
バージョン番号をどうつけるか？

コミットハッシュとタグ

- コミットハッシュ
  - 1番細かい変更の単位
  - コミットオブジェクト（履歴データ）の SHA-1ハッシュ値
    - 他人のハッシュ値と衝突しない
    - バージョン間の関連はコミットオブジェクトの親コミットから復元する
  - gitの最新のコミットは 39c15e485575089eb77c769f6da02f98a55905e0
- タグ
  - ある程度のまとまりのある変更の単位
  - コミットに対してタグをつける
  - gitの最新のタグは v2.36.1
  - セマンティック バージョニングを使うところが多い？
    - X.Y.Zみたいな形式．仕様もある

履歴の分岐（branch）

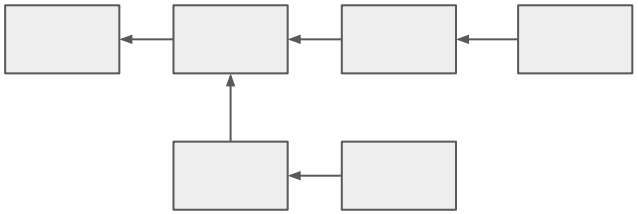
- 履歴を分岐させることができる
- 何か変更するときはブランチを切るところから始める
- 実体
  - どのコミットに対して変更を加えようとしているかを表すもの
  - コミットするときの親コミットに人間がわかりやすい名前をつけたもの
- ブランチの作成・切り替えは高速に実行できる
  - 作成：HEADを書き換えるだけ
  - 切り替え：対象コミットのスナップショットを読み出すだけ



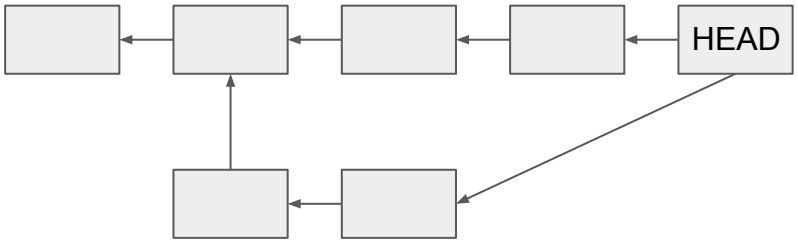
履歴の合流（merge/rebase）

- 分岐したブランチを1つにまとめることができる
  - ブランチでの作業内容を本体に取り込む
  - 本体の最新の変更を作業ブランチに反映させる
  - 変更の競合が発生することがある（コンフリクト）
  - ある程度の競合は自動で解消してくれる
- マージ
  - 親コミットが複数になるコミットオブジェクトを生成する
  - Gitに不慣れなうちはこちらがおすすめ
- リベース
  - 親コミットを付け替えて，履歴が分岐しないように履歴が 1本道になるようにする
  - 親コミットを変更するのでコミットハッシュが変更されるので注意

マージとリベース

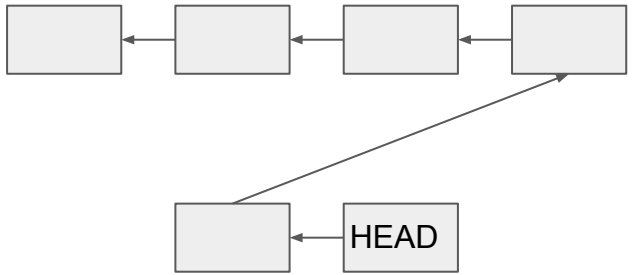


マージ



親が複数のコミットを作成

リベース



ブランチの親をつけかえる

履歴の共有（fetch/pull/push）

- チームメンバーとどう履歴を共有するのか
  - リモートのGitレポジトリを経由する
  - マシン間の共有にも使える
- リモートとローカルの同期をどう取るのか
  - ローカルにリモートのブランチを管理する用のブランチを持つ
  - ローカルのmasterとリモートのorigin/master の2つで管理
  - リモートに変更があってもローカルのブランチに影響はない
- fetch
  - ローカルにあるリモートブランチの情報を更新（変更のダウンロード）
- pull（= fetch + merge）
  - ローカルのリモート情報を更新し，リモートの変更を現在のローカルブランチに反映
  - origin/masterをmasterにマージ（変更の反映）
- push
  - ローカルのブランチをリモートに作成する（変更のアップロード）

Gitのアプローチのまとめ

- ファイルデータの持ち方
  - ファイルのSHA-1ハッシュとファイルデータをローカル DBで管理する
- バージョン番号
  - コミットオブジェクトの SHA-1ハッシュ値
  - バージョン間の関連は親コミットの情報から復元
- 複数名開発
  - ハッシュの性質によりバージョン番号は衝突しない
  - 履歴の分岐・合流が簡単にできる
  - 履歴のデータをみんながローカルにもっている
- 変更履歴
  - 変更を加えるたびにコミットオブジェクトを生成する

3. Gitを用いた開発の進め方

準備

- Gitのインストール
  - git --version が実行できれば問題ないです
  - 深刻な脆弱性が見つかったこともあるので，最新のものにアップデートしましょう！
- Gitのホスティングサービス
  - GitHub，GitLab, Bitbucketあたりが有名．
  - GitとGitHubは違う
  - 外部サービスを使わずにオンプレで運用することも可能
  - ちなみにDeepFlow社はGitLab→GitHub
  - バージョン管理以上の豊富な機能が用意されている

Gitの設定

- $ git --version
  - バージョンの確認
- $ git config --global user.name “Jun Kohjina”
  - コミット情報に追加される
  - 恥ずかしい名前にはしない方が良いです
- $ git config --global user.email “jun.kohjina@example.com”
- $ git config --global core.editor 好きなエディタ
  - お好きなエディタで
  - 突然見慣れないエディタが起動して困惑することがある
- ~/.gitconfig, project/.git/config
  - プロジェクトごとに設定は変更できる

ローカルリポジトリの準備

- 新しいリポジトリを作成する
  - チーム開発の場合は誰か 1人がやればいい
- リモートからダウンロードする
  - GitHubやGitLabに上がっているプロジェクトをダウンロードする
- 他のバージョン管理ツールから移行
  - SubversionやMercurialからGitへの移行もできる
    - git-svn
  - 今日は話さない

ローカルリポジトリの準備

- 新しくリポジトリを作成
  - git init
    - ls -aすると.gitというディレクトリが作られていることがわかる
    - .gitに全履歴データがつまっている
  - git remote add 名前 リポジトリのURL
    - ローカルのリポジトリにGitHubのリポジトリを結びつける
    - GitHubで先にリポジトリを作成しておく
    - リモートは複数登録できる
    - ローカルでのみ使うのであればやる必要はない
- リモートからダウンロード
  - git clone リポジトリのURL
  - GitHubにあるオープンソースのソフトウェアを使うときにも使う

開発の基本的な流れ

- ブランチを切る
- コードを書く
- 変更したコードをステージする
- コミット
- リモートにプッシュ
- プルリクエスト/マージリクエストを作成
- 元のブランチにマージ

ブランチを切る

- ブランチの作成
  - $ git branch 新しいブランチ どこからつくるか
  - $ git branch new-feature origin/master
  - どこから分岐するかは省略可能でデフォルトは HEAD
- ブランチの切り替え
  - $ git checkout ブランチ名
- ブランチの確認
  - $ git branch
  - 今いるブランチに*がつく
- ブランチの作成と切り替えを同時に
  - $ git checkout -b new-feature origin/master
  - 最近はgit switchというのもある

ステージ

- 次のコミットに含めたい変更をインデックスに登録する
  - ファイルは以下の4つの状態をもつ
    - Untracked：新規
    - Unmodified：変更されてない
    - Modified：変更されている（次のコミットに含まれない）
    - Staged：ステージされている（次のコミットに含まれる）
  - $ git statusで確認できる
- ファイルをステージ
  - $ git add 対象ファイル
- ファイルをアンステージ
  - $ git restore --staged 対象ファイル

コミット

ステージされたファイルの変更を反映させたコミットオブジェクト（履歴データ）を作成する

- $ git commit -m ‘メッセージ’
- コミットしただけではリモートには反映されない
  - ネットワークに繋がっていなくてもコミットできる
- 最後のコミットの取り消し
  - git reset --soft HEAD^
  - 取り消すならばリモートに反映する前に

リモートにプッシュ

- コミットしただけではリモートに反映されない
- git push プッシュ先 ブランチ名
  - git push origin new-feature
  - リモートにあるブランチに変更を反映させる
- バックアップになるので定期的にpushする
- リモートのブランチはみんなで共有されるもの
  - マナーを守る（コミットハッシュを書き換えるような危険な操作は特に気をつける）
  - リモートのブランチ名が他の人と衝突しうる（ pushに失敗する）
  - アクセストークン等の他人に知られてはいけない認証情報をプッシュしない

プルリクエストの作成

- GitHubやGitLabなどのホスティングサービスにある機能
  - Gitそのものの機能ではない
  - GitLabではマージリクエストという
- メインのブランチにこの変更を取り込んで欲しいというリクエスト
- メインに取り込んでいいのか検証する
  - どういう変更が行われているか
  - コードレビューしてもらう
  - テストをパスするか
  - メインのブランチと競合していないか
- web上で簡単にマージできる（権限があれば）

変更の取り込み

リモートにある変更をローカルのブランチに反映させる

- ローカルのバージョン管理DBにリモートの情報を反映させる
  - git fetch
  - 現在作業中のファイルを書き換えたりはしない
- リモートブランチの変更をローカルブランチに取り込む
  - git merge origin/master
  - ネットワークにつながっていなくてもできる
  - コンフリクトが発生する可能性がある
  - リベースで取り込むこともできるが，Gitに不慣れなうちはおすすめしない

開発の基本的な流れ（再掲）

- ブランチを切る
- コードを書く
- 変更したコードをステージする
- コミット
- リモートにプッシュ
- プルリクエスト/マージリクエストを作成
- 元のブランチにマージ

まとめ

- バージョン管理
  - 手動でのバージョン管理は運用が大変（特にチーム開発での）
- Gitのアプローチ
  - コミットハッシュ（差分ではなくスナップショットで管理）
  - コミットの親子関係をメタ情報にもつ
- Gitを用いた開発の進め方
  - ブランチを切る(git branch,git checkout/git switch)
  - 変更したファイルをステージ (git add)
  - ステージしたものをコミット (git commit)
  - リモートにプッシュ (git push)
  - プルリクエスト経由でメインのブランチにマージ
- Gitの導入はモダンな開発体制への第一歩
  - DeepFlowで導入支援しています