Day 1 Git/GitHub

赤松 祐希

cookpad

講師

クックパッド株式会社 新規サービス開発部

あかまつ ゆうき

赤松 祐希(@ukstudio)

・サーバーサイドエンジニア
・今回のインターンの実行委員長

TA

クックパッド株式会社 買物事業部
エンジニア

さとう　あつや
佐藤 敦也 (@n_atmark)

・2019.04 クックパッド新卒入社
・2019.05~ 買物事業部

この講義の目的

- GitとGitHubの基本的な使い方と開発ワークフローを理解する
- 特に10 Dayの後半のOJTやPBLで重要になってきます

バージョン管理

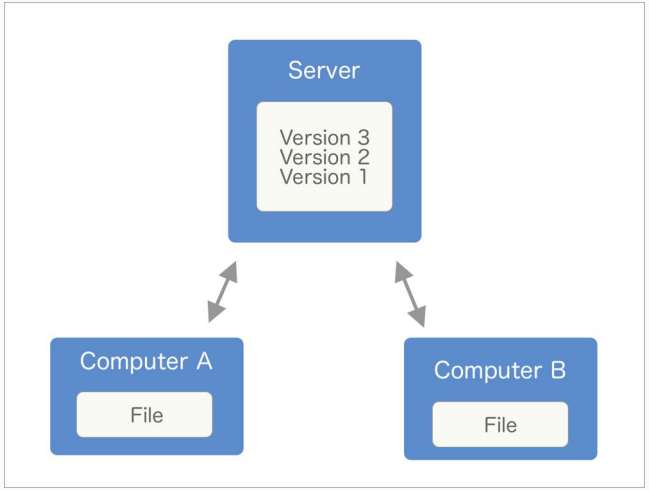
- ファイルやファイルの集合に対して、日々の変更を記録するシステム・ツールのこと
- 変更を取り消したり、過去のと特定のバージョンを呼び出すということも可能

Git

- 分散型のバージョン管理システム
  - 他の分散型ではMercurialなど
- 対して中央集権型のバージョン管理システムもある
  - Subversionなど
- Linuxの作者であるLinus TorvaldsがLinuxの開発のために作った
- GitHubの存在もあり、ほぼデファクトと言ってもいい状態

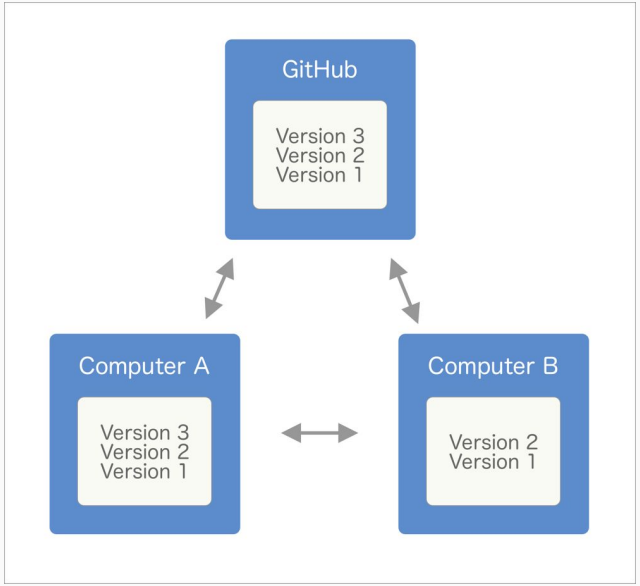
中央集権型

- 分散型が主流になる前
- ネットワーク上にあるサーバーに変更をコミットしていく
- 開発マシンがオフラインだったり、サーバーがダウンしているとコミットすることができず開発が継続できない



分散型

- 最近の主流
- ローカルにリポジトリをコピーするため、ローカルでも履歴を持つことができる
- とはいえ、オリジンとなるリポジトリは必要
  - 例えばGitHubにあるリポジトリ
- ローカルで履歴を持つため、ローカルでコミットすることが可能



Gitの操作

初期設定

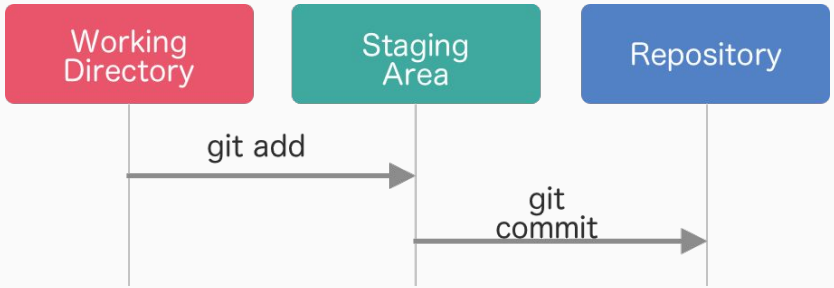
- コミットに記録される名前とメールアドレスを設定します
- git config --global user.name “Your Name”
- git config --global user.email “Your email”

リポジトリの作成とファイルの追加

- リポジトリの作成
  - mkdir -p ~/works/git-handson
  - cd ~/works/git-handson
  - git init
- ファイルを追加
  - echo “# README” > README.md
  - git add README.md
  - git commit -m “Add README.md”

3つのエリア

- Working Directory
  - ディスク上に配置されているファイル郡がある場所
- Staging Area(.git/index)
  - 次のコミットに含まれる情報が蓄えられている場所
- Repository(.git/objects)
  - リポジトリのデータベース
  - メタデータやオブジェクトなどが格納されている



ブランチ

- 本流(master)から分岐して履歴を記録することができる機能
- 複数のブランチを作ることができるので、並行して複数機能の開発も可能
- git branch new-feature
- git checkout new-feature
- echo “## New Heading” >> README.md
- git add README.md
- git commit -m “Update README.md”

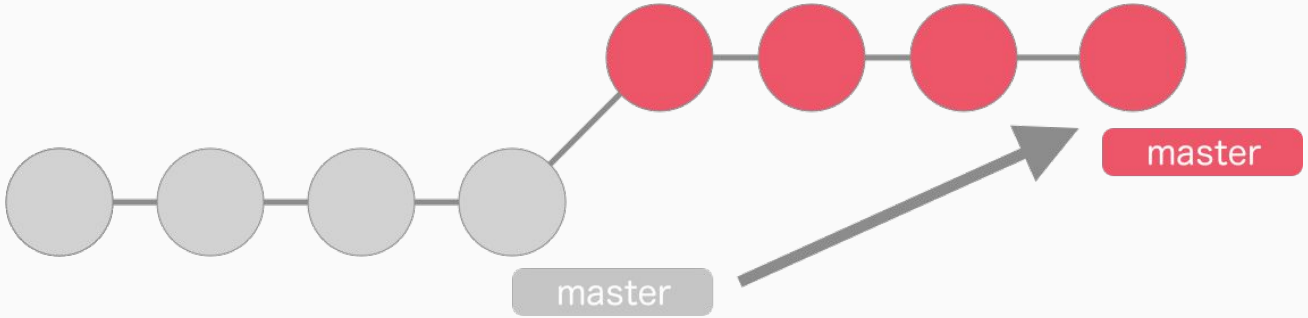
merge

- ブランチでの開発が完了したら、変更をmasterに取り込む
- git checkout master
- git merge --no-ff new-feature

fast forward

- 雑に表現するとmerge時にmergeコミットを作るか、作らないか
- --no-ff の場合、「必ず」mergeコミットを作る
  - git log を叩いて、mergeコミットができることを確かめよう
- --no-ffをつけない場合、Gitがfast forwardできるか判断する
  - git reset --hard HEAD^
  - git merge new-feature
  - git logを叩いて、mergeコミットができていないことを確かめよう

fast forwardの仕組み



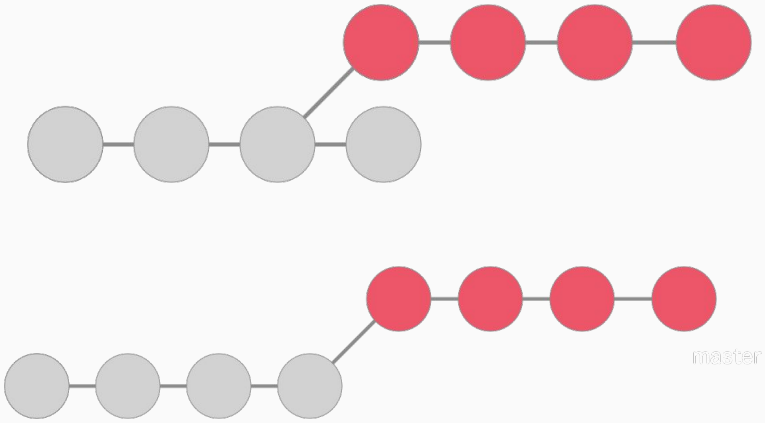
- fast forwardの場合、実はマージ処理をしていない
  - masterが示すリビジョンが変わっているだけ
  - リビジョン=特定のコミットを認識するための一意な文字列

ff or no-ff

- 個人的にはno-ffの方がよく使う
  - GitHubのPull Requestのmergeもno-ff
- mergeコミットが存在していると、その機能の差し戻し(revert)がラク
- ローカルでの開発でfeatureブランチから更に派生させたブランチをfeatureブランチに手元でmergeする時に使う

rebase

- git checkout feature-branch
- git rebase master
- ブランチの派生元を変更するイメージ
  - masterの変更をブランチに反映させるときなどに使う
- 親コミットが変わるため、コミットのリビジョンが変わることに注意



rebase vs merge

- masterの変更を反映させるならmergeする方法もある
- rebase
  - ツリーがまっすぐキレイになる
  - fast forwardでmergeできるようになる
  - リビジョンが変わるため、リモートにforce pushするしかなくなる
- merge
  - コンフリクト解消がrebaseよりラク
  - mergeコミットが多いと、履歴が順に読みづらくなる
- コンフリクトが少ない & 人とブランチを共有していない状態ならrebase
  - とは言え、結構どちらでもいいと思っています(個人の意見)

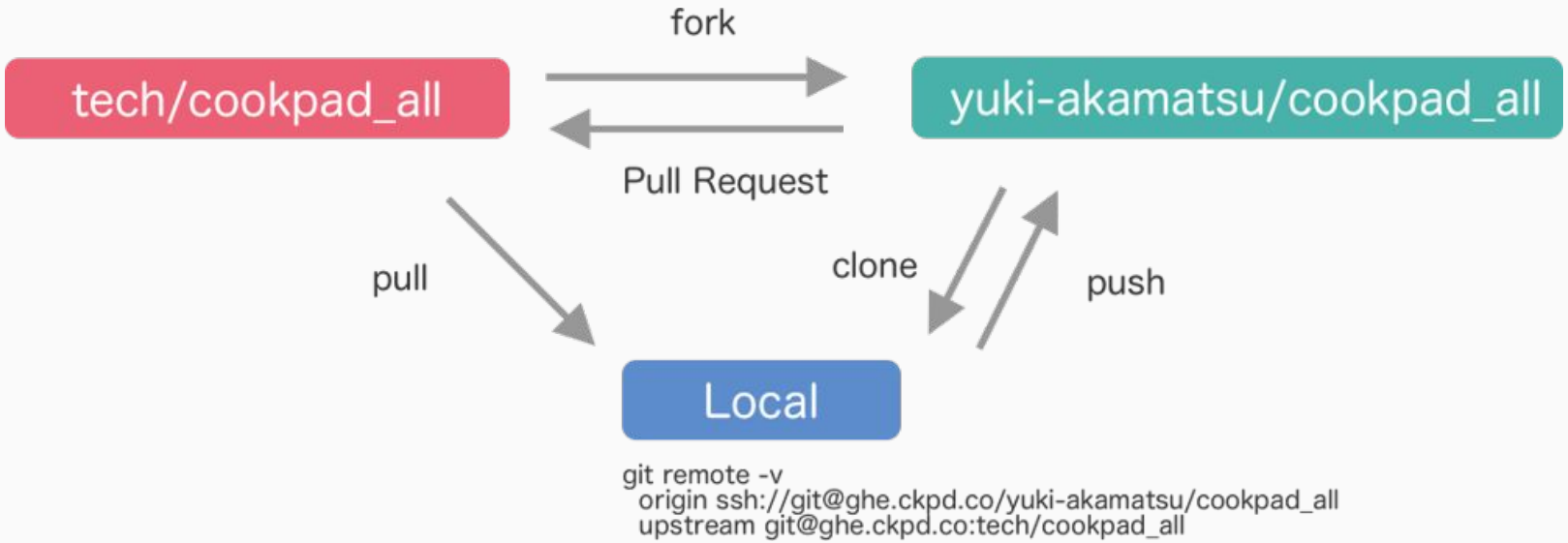
チーム開発

cookpad

GitHub

- 開発に多数の人間が関わる場合、リポジトリの管理や変更の取り込みの管理などが大変
- GitHubを使うことで管理の手間を減らす
- GitHub = ソースコードホスティングと表現されることもあるが、どちらかというと開発のやり取りをする場
- オマケ: クックパッドではGitHub Enterpriseを使用
    - GitHubとは若干の違いがあったりします(新しい機能がまだ使えないとか)

GitHubでのチーム開発の全体象



GitHubでのチーム開発の全体象

1. 対象プロジェクトをGitHub上でforkする
    a. gitにforkという機能は実はないが、慣習的な理由もあり人のリポジトリのコピーを作ることをforkと言う
2. forkして出来たリモートリポジトリをローカルにclone
3. ローカルでbranchを切り、変更作業を行いコミット
4. ブランチをリモートリポジトリにpush
5. GitHub上でPull Requestを作成する
6. CIの確認、コードレビューなどをし、問題がなければマージする(or してもらう)

ローカルリポジトリとリモートリポジトリ

- ネットワーク上にあるリポジトリをリモートリポジトリと言う
- GitHubにあるリポジトリとローカルにあるリポジトリは別のものなので区別するために使う
- 1つのローカルリポジトリに複数のリモートリポジトリを設定することが可能
- 慣習的に以下の名前をつけることが多い
  - upstream : forkの元となったオリジナルのリモートリポジトリ
  - origin : cloneの元となったリモートリポジトリ

Pull Requestを送ってみよう

- https://ghe.ckpd.co/yuki-akamatsu/2019-summer-intern-git をfork
- cd ~/works
- git clone https://ghe.ckpd.co/[your-name]/2019-summer-intern-git
- cd 2019-summer-intern-git
- git branch add-my-readme
- git checkout add-my-readme
- vim [your-name].md
    - 今回のインターンの意気込みを書こう
- git add [your-name].md
- git commit -m “Add [your-name].md”
- git push origin add-my-readme
- GHEからPull Requestを送ろう

Upstreamの更新をローカルに反映させよう

- git remote add upstream git@ghe.ckpd.co:yuki-akamatsu/2019-summer-intern-git.git
- git remote -v
- git fetch upstream
- git checkout master
- git merge upstream/master

重要だけど話していないこと

cookpad

わかりやすいコミットを作る

- コードを読む時間は書く時間より長いという話もある
- コードを読む際にコミットはとても貴重な情報源となる
  - 例えば「ここはなぜこうなっているのだろう?」という疑問があったときにgit blameでコミットを見にいくとか
- コードレビューの時にもまずザッとPRに含まれているコミットを眺める人も多い

コミットメッセージ

- 「fix」というメッセージのコミットが一杯並んでいてもなにもわからない
- コミットメッセージのタイトルには概要を簡潔に書く
  - 「Use foo method instead of bar method that deprecated」
- コミットメッセージの本文には「なぜ」を書くことを意識する
  - 実際にどうやっているかはコードを読めばわかる
  - 「なぜ」はコードを読んでもわからない
  - タイトルで説明が済んでいれば本文はなくても良い

コミットの粒度

- 適切な粒度を説明するのは難しいが、少なくとも「大きすぎる」より「小さすぎる」コミットの方がよい
  - 迷ったら小さくする
- 意図が伝わるコミットを目指す
  - やりたいことに対して本質的な変更のみにする
  - たまたま見つけたからと言って、関係のないtypoなどを一緒にまとめたりしない

履歴を整える

- とは言え、開発しながらコミットの粒度を保つのは大変
  - コミット後に修正漏れに気づくとか
- 開発中はある程度適当にコミットしつつ、後で整える
  - git rebase -i
  - git fixup

Pull Request

- Pull Requestもコミット同様に大切
- Pull Requestも大きいより小さい方が良い
  - レビュアーの負担が減る
  - フォーマットの変更、リファクタリングなどは別に先に出す
- Pull Requestのdescriptionを丁寧に
  - Pull Requestの目的をちゃんと伝える
  - Pull Requestを見る人が背景の事情を知っているとは限らない
  - 画面の変更が入る場合、before / after のスクリーンショットを貼るなどする