Git/GitHub 演習

産業技術大学院大学
中鉢 欣秀

2016 年度

目次

1 Git/GitHub 演習

1.1 Git/GitHub 演習について

1.1.1 授業の概要

■この資料の入手先

- https://github.com/ychubachi/github_practice

■この演習について

- この授業では Git の初心者が，基礎的な Git コマンドの利用方法から，GitHub Flow に基づく協同開発の方法までを学ぶ

■事前準備

- 事前に git コマンドが利用できる環境を用意しておくこと
- CUI 端末での shell による基本的な操作を知っているとスムーズに演習ができる

■授業の構成

- 個人演習では，テキストの指示に従い，Git/GitHub を利用するにあたり必要となる知識を学ぶ
- チーム演習では，GitHub を活用した協同開発の方法を深く学ぼう

■授業の進め方

1. 演習の解説
   - 講師が授業の進め方を説明する
2. Git/GitHub を学ぶ個人演習
   - 個人演習を通して Git/GitHub の使い方を学ぶ
3. チーム演習
   - チームでの開発演習を実施する

1.1.2 個人演習からチーム演習へ

■個人演習からチーム演習への流れ

- この授業では最初に個人演習を行い，その後，チームによる演習に進む
- その際，チーム編成が既に済んでいるか，または，そうでないかで演習の進め方が異なる

■チーム編成が済んでいる場合

- 個人演習としてテキストの課題に取り組む
- テキストを終えたメンバーは他のメンバーを積極的に助ける
- 全員がテキストを終えることを目指す
- 全員が完了，もしくは，時間になったらチーム演習に進む

■チームがまだできていない場合

- 後ほど席の移動をするので荷物をまとめておく
- 個人演習としてテキストの課題に取り組む
- テキストを完了したら講師・TA に伝えること
- その後，チーム編成を経てチーム演習に進む

■チームがまだできていない場合の編成方法

- 個人演習が完了した者から順番に 2 名づつのペアを組んでいく
- できたペアは空いている席に移動して，チーム演習を開始する
- 受講者の半数がペアになったら，それ以降にテキストを終えた者は既存のペアに追加していく
- 最終的に 3〜4 人のグループにする

1.1.3 成績評価の方法

■チーム演習の評価

- チーム演習での GitHub のリポジトリを対象に，主に以下の項目について評価する
  - コミットの数（一人 5 個以上）
  - コミットの粒度（意味のある単位でできるだけ細かく）
  - コミットメッセージの分かりやすさ
  - ブランチの名前が作業内容を表しているか
  - プルリクエストの活用
  - Wiki や Issue の活用
  - コンフリクトの解消ができたか
  - README/LICENCE
- ようするに「GitHub Flow」がうまく回せたか？

■提出物

- 提出物は次のとおり
  - 名前
  - 学籍番号
  - GitHub のアカウント名
  - GitHub のリポジトリの Web URL
    - 個人演習「our_enpit」
    - チーム演習「team_enpit」
  - 各自が行った作業の内容
  - 自分が作成したコミット数
  - 自己評価
    - 5 段階：5 はとても優れている，4 は優れている，3 は普通，2 は劣っている
  - 自己評価の理由
  - 演習全体の感想

■成果物の提出方法と補足資料

- 成果物の提出方法や，その他の補足資料は Wiki を参照
  - Home · ychubachi/github_practice Wiki

2 Git/GitHub 演習（個人演習編）

2.1 Git 入門

2.1.1 Git の操作方法と初期設定

■Git チートシート（カンニング表）

- 主な Git コマンドの一覧表
  - Git チートシート（日本語版）
- 必要に応じて印刷しておくとよい

■Git コマンドの実行確認

- 端末を操作して Git コマンドを起動してみよう．
- 次のとおり操作することで Git のバージョン番号が確認できる．

```
git --version
```

■名前とメールアドレスの登録

- （まだなら）git に名前とメールアドレスを登録しておく
- 次のコマンドの$NAME と$EMAIL を各自の名前とメールアドレスに置き換えて実行せよ
  - 名前はローマ字で設定すること

```
git config --global user.name $NAME
git config --global user.email $EMAIL
```

■その他の設定

- 次のとおり，設定を行っておく

```
git config --global color.ui auto
git config --global push.default simple
git config --global core.editor emacs
```

- 1 行目: 色付きで表示を見やすく
- 2 行目: push する方法（詳細省略）
- 3 行目: vim を使う場合は emacs ではなく vim を

■設定の確認方法

- ここまでの設定を確認する

```
git config -l
```

2.1.2 Git のリポジトリ

■プロジェクト用のディレクトリ

- リポジトリとはプロジェクトでソースコードなどを配置するディレクトリ
  - このディレクトリを「ワーキングディレクトリ（ないしはワーキングツリー）」とも言う
- Git によりバージョン管理ができる
  - ファイルに対する編集作業の内容が追跡され，記録される
- 将来的に GitHub と連携させることで共同作業ができるようになる

■Git リポジトリを利用するには

- リポジトリを利用する方法には主に 2 種類ある
  1. git init コマンドで初期化する方法
  2. git clone コマンドで GitHub から入手する方法
- ここでは，まず 1. について解説する

■Git リポジトリの初期化方法

- my_project ディレクトリを作成し，Git リポジトリとして初期化するコマンドは次のとおり
  - 1〜2 行目：ディレクトリを作成して移動
  - 3 行目：ディレクトリをリポジトリとして初期化

```
mkdir ~/my_project
cd ~/my_project
git init
```

- 以降の作業は作成した my_project ディレクトリで行うこと
  - 現在のディレクトリは「pwd」コマンドで確認できる

■リポジトリの状態を確認する方法

- 現在のリポジトリの状態を確認するコマンドは次のとおり

```
git status
```

- このコマンドは頻繁に使用する
- 何かうまく行かないことがあったら，このコマンドで状態を確認する癖をつけるとよい
  - 表示される内容の意味は徐々に覚えていけば良い

■「.git」ディレクトリを壊すべからず

- ティレクトリにリポジトリを作成すると「.git」という隠しディレクトリができる
  - ls -a で確認できるが…
- このディレクトリは 絶対に手動で変更してはならない
  - もし削除したら Git とは無関係の単なるディレクトリになる

2.1.3 コミットの作成方法

■コミットについて

- Git の用語における「コミット」とは，「ひとかたまりの作業」をいう
  - 新しい機能を追加した，バグを直した，ドキュメントの内容を更新した，など
- Git は作業の履歴を，コミットを単位として管理する
  - コミットは次々にリポジトリに追加されていき，これらを記録することでバーションの管理ができる（古いバージョンに戻る，過去の変更内容を確認する，など）
- コミットには，作業の内容を説明するメッセージをつける
  - 更に，コミットには自動的に ID が振られることも覚えておくと良い

■README ファイルの作成

- my_project リポジトリに README ファイルを作成してみよう

```
echo "My README file." > README
```

- プロジェクトには 必ず README ファイルを用意 しておくこと

■リポジトリの状態の確認

- git status で現在のリポジトリの状態を確認する

```
git status
```

- 未追跡のファイル（Untracked files:）の欄に作成した README ファイルが（赤色で）表示される

■変更内容のステージング

- コミットの一つ手前にステージングという段階がある
  - コミットしたい変更はステージングしておく
    - 逆に言えば，変更をコミットするためには，ステージングしておかなくてはならない
  - 新しいファイルをステージングすると，これ以降，git がそのファイルの変更を追跡するようになる
    - これをトラッキングという

■ステージングの実行

- 作成した README ファイルをステージングするには，次のコマンドを打つ

```
git add .
```

- 「git add」の「.（ピリオド）」を忘れないように
  - ピリオドは，リポジトリにおけるすべての変更を意味する
  - 複数のファイルを変更した場合には，ファイル名を指定して部分的にステージングすることもできる…
    - が，このやりかたは好ましくない
    - 一度に複数の変更を行うのではなく，一つの変更を終えたらこまめにコミットする

■ステージング後のリポジトリへの状態

- 再度，git status コマンドで状態を確認しよう

```
git status
```

- コミットされる変更（Changes to be committed:）の欄に，README ファイルが（緑色で）表示されれば正しい結果である

■ステージングされた内容をコミットする

- ステージング段階にある変更内容をコミットする
- コミットにはその内容を示すメッセージ文をつける
- 「First commit」というメッセージをつけて新しいコミットを作成する
  - 「-m」オプションはそれに続く文字列をメッセージとして付与することを指示するもの

```
git commit -m 'First commit'
```

■コミット後の状態の確認

- コミットが正常に行われたことを確認する
  - ここでも git status コマンドか活躍する

```
git status
```

- 「nothing to commit, . . . 」との表示からコミットすべきものがない（＝過去の変更はコミットされた）ことがわかる
- この表示がでたら（無事コミットできたので）一安心してよい

2.1.4 変更履歴の作成

■更なるコミットを作成する

- リポジトリで変更作業を行い，新しいコミットを追加する
  - README ファイルに新しい行を追加する
- 次の$NAME をあなたの名前に変更して実行しなさい

```
echo $NAME >> README
```

- 既存のファイルへの追加なので「»」を用いていることに注意

■変更後の状態の確認

- リポジトリの状態をここでも確認する

```
git status
```

- コミットのためにステージされていない変更（Changes not staged for commit:）の欄に，変更された（modified）ファイルとして README が表示される

■差分の確認

- トラックされているファイルの変更箇所を確認する

```
git diff
```

- 頭に「+」のある（緑色で表示された）行が新たに追加された内容を示す
  - 削除した場合は「-」がつく

■新たな差分をステージングする

- 作成した差分をコミットできるようにするために，ステージング段階に上げる

```
git add .
```

- git status を行い，README ファイルが「Changed to be commited:」の欄に（緑色で）表示されていることを確認する

■ステージングされた新しい差分のコミット

- 変更内容を示すメッセージとともにコミットする

```
git commit -m 'Add my name'
```

2.1.5 履歴の確認

■バージョン履歴の確認

- これまでの変更作業の履歴を確認
  - 2 つのコミットが存在する

```
git log
```

- 各コミットごとに表示される内容
  - コミットの ID（commit に続く英文字と数字の列）
  - Author と Date
  - コミットメッセージ

■2 つのコミットの比較

- 異なる 2 つのコミットの変更差分は次のコマンドで確認できる
  - コミットの ID は log で確認できる（コマンドで ID を指定する場合は，概ね先頭 4 文字を入力し後は省略してよい）

```
git diff $COMMIT_ID_1 $COMMIT_ID_2
```

- （参考）後で説明するブランチはコミットのエイリアスなので，ブランチごとの比較もできる

■コミットの情報確認

- 次のコマンドでコミットで行った変更内容が確認できる

```
git show $COMMIT_ID
```

2.1.6 ブランチの使い方

■ブランチとは

- 「ひとまとまりの作業」を行う場所

- ソースコードなどの編集作業を始める際には必ず新しいブランチを作成する
- Git の内部的にはあるコミットに対するエイリアス（alias）である

■master は大事なブランチ

- Git リポジトリの初期化後，最初のコミットを行うと master ブランチができる
- 非常に重要なブランチであり，ここで 直接編集作業を行ってはならない
  - ただし，本演習や，個人で Git を利用する場合はこの限りではない

■ブランチの作成と移動

- 新しいブランチ「new_branch」を作成して，なおかつ，そのブランチに移動する
  -「-b」オプションで新規作成
  - オプションがなければ単なる移動（後述）

```
git checkout -b new_branch
```

- 本来，ブランチには 「これから行う作業の内容」 が分かる名前を付ける

■ブランチの確認

- ブランチの一覧と現在のブランチを確認する
  - もともとある master と，新しく作成した new_branch が表示される

```
git branch -vv
```

- ブランチに紐づくコミットの ID が同じことも確認
- git status の一行目にも現在のブランチが表示される

■ブランチでのコミット作成

- README に現在の日時を追加

```
date >> README
git add .
git commit -m 'Add date'
```

- 新しいコミットが追加できたことを git log で確認
- git branch -vv でコミットの ID が変化したことも確認

■ブランチの移動

- new_branch ブランチでコミットした内容を master に反映させる
  - まずは master に移動する

```
git checkout master
```

- git status，git branch -vv で現在のブランチを確認すること
- この段階では，README ファイルに行った変更が 反映されてない ことを確認すること

■変更を master にマージ

- new_branch で行ったコミットを master に反映させる

```
git merge new_branch
```

- README に更新が反映されたことを確認
- git branch -vv により両ブランチのコミット ID が同じになったことも確認
- git log も確認しておきたい

■マージ済みブランチの削除

- マージしたブランチはもはや不要なので削除して良い

```
git branch -d new_branch
```

- git branch -vv コマンドで削除を確認

2.1.7 コンフリクト

■コンフリクトとは

- ファイルの同じ箇所を，異なる内容に編集すると発生する
- Git はどちらの内容が正しいのかわからない
- 次のシナリオに従い，コンフリクトを発生させてみよう

■コンフリクトのシナリオ

-「のび太」の作業
  - nobita ブランチを作成する
  - README ファイルの 一行目 を「Nobita's README.」に変更する
  - 変更を add して commit する
- ここで一度， master ブランチにもどる
  - README がもとのままだあることを確認
-「しずか」の作業
  - shizuka ブランチを作成する
  - README ファイルの 一行目 を「Shizuka's file.」に変更する
  - 変更を add して commit する

■マージとコンフリクト発生

- master ブランチに移動する
- まず，nobita ブランチをマージ
  - 問題なくマージできる
- 次に，shizuka ブランチをマージ
  - ここでコンフリクトが発生する

■コンフリクト時のメッセージ

- merge に失敗するとようなメッセージが出る（長いので改行を加えた）

```
Auto-merging README
CONFLICT (content):
  Merge conflict in README
Automatic merge failed; fix conflicts
  and then commit the result.
```

- また，git status すると Unmerged pathes:の欄に，「both modified: README」と表示される

■README ファイルの内容

- README を開くとコンフリクトが起きた箇所がわかる

```
<<<<<<< HEAD
Nobita's README.
=======
Shizuka's file.
>>>>>>> shizuka
（以下略）
```

- =======の上がマージ前の master ブランチ，下がマージしようとした shizuka ブランチの内容

■コンフリクトの解消

- テキストエディタで修正し，手動でコンフリクトを解消する

```
Nobita & Shizuka's READMEfile.
（以下略）
```

■解消した結果をコミットする

- その後はいつもどおり，add して commit すれば作業が継続できる
  - マージ済みの master から新しくブランチを作成すること
- なお，テキストエディタを用い，手動で正しくコンフリクトを解消する前でも commit できてしまう ので，この点には注意する

2.2 GitHub 入門

2.2.1 GitHub とは

■GitHub でソーシャルコーディング

- ソーシャルコーディングのためのクラウド環境
  - GitHub
  - GitHub Japan
- GitHub が提供する主な機能
  - GitHub flow による協同開発
  - Pull requests
  - Issue / Wiki

■GitHub アカウントの作成

- （まだなら）次の URL の指示に従い GitHub アカウントを作成
  - Signing up for a new GitHub account - User Documentation
- アカウントの種類
  - 無料版で作成する場合「Join GitHub for Free」を選択する
  - 学生の場合「Student Developer Pack」にアップグレードすることもできる
- その後，確認メールが届くので，必要に応じて残りの手順を実施せよ
  - GitHub Help

■SSH による GitHub アクセス

- GitHub へのアクセスは SSH を用いた公開鍵暗号方式の認証を用いる
  - SSH 公開鍵の設定を行えば以降のパスワード認証が不要になる
- （まだなら）SSH を生成して GitHub に登録しなさい
  - 鍵を生成するとき「passphrases」が聞かれるが，この演習では何も入力しなくてよい
  - Generating an SSH key - User Documentation

2.2.2 リモートリポジトリ

■リモート VS ローカルリポジトリ

- ローカルリポジトリ
  - git init コマンドを用いて作成したリポジトリを「ローカルリポジトリ」という
- リモートリポジトリ
  - 「リモートリポジトリ」とは，サーバ上にあるリポジトリであり，ローカルのリポジトリと連携させることができる
- リモートリポジトリの利点
  - ネットワークを経由してどこからでも利用することができる
  - 複数人のチームで協同作業をするときに活用できる

■リモートリポジトリの作成

- リモートリポジトリを GitHub で作成する
  - 名前は「our_project」とする
- 次の手順で作成する
  - Creating a new repository - User Documentation
- README とライセンスを追加すること
  - 「Initialize this repository with a README」にチェックを入れる
  - 「Add a license:」から「MIT License」を選ぶ

2.2.3 GitHub flow

■GitHub floww による開発の流れ

- GitHub flow
  - Understanding the GitHub Flow · GitHub Guides

■1: リモートリポジトリをローカルに複製

- リモートにあるリポジトリをローカルに複製することを clone という
  - Cloning a repository - User Documentation
- 下記の「$GITHUB_URL」の部分を GitHub の our_project リポジトリ URL にして実行
  - URL は「git@...」で始まる SSH 接続用のものを用いる
    - リポジトリの URL はブラウザ用の URL とは異なるので注意！

```
cd ~
git clone $GITHUB_URL
cd our_project
```

- この作業は基本的にはプロジェクトに対して一度だけ行うこと

■2: master から作業用ブランチを作成

- 作業用のブランチを作成して移動する
  - ブランチの名前は「greeting」とする

```
git checkout -b greeting
```

■3: ブランチで編集作業を行う

- ここでは，hello.txt という名前のファイルを作成する

```
echo 'Hello GitHub' > hello.txt
```

■4: ブランチでコミットを作成

- 変更した内容をステージングしてからコミットする

```
git add .
git commit -m 'Create hello.txt'
```

- この編集，add，commit の作業は作業が一区切りつくまで何回も繰り返してよい…
  - が，こまめに push するのが良いとされる

■5: ブランチをリモートに送る

- ブランチで作成したコミットをリモートに送る
  - 下記の origin はリポジトリの URL の別名として自動で設定されているもの
  - greeting は作業しているブランチ名

```
git push -u origin greeting
```

■6. GitHub でプルリクエストを送る

- ブランチが GitHub に登録されたことを確認し，Pull request を作成する
- 手順は次のとおり
  - Using pull requests - User Documentation の前半
  - Creating a pull request - User Documentation

■7. GitHub でレビュー（＋自動テスト）

- プルリクエストを用いたレビューの方法は下記参照
  - Using pull requests - User Documentation の後半
- 人手によるレビューの他，自動的なテストも行うのが望ましい（説明は省略）

■8. GitHub でプルリクエストをマージ

- Pull request のレビューが済んだらマージする
  - Merging a pull request - User Documentation
- マージが完了したら，ローカル・リモート共に，マージ済みのブランチは削除してよい

■9. ローカルの master を最新版にする

- GitHub で行ったマージをローカルに反映させる
  - master ブランチに移動して git pull
  - 不要になった作業用ブランチは削除

```
git checkout master
git pull
git branch -d greeting
```

■GitHub Flow に習熟するには？

- ここで手順 2:（2.2.3 節）に戻り，一連の作業を複数回（5 回以上!）繰り返すこと
  - 体に叩き込む！

2.2.4 コンフリクトについて

■GitHub flow におけるコンフリクトについて

- コンフリクトとは？
  - コンフリクトは、コードの同じ箇所を複数の人が別々に編集すると発生
- コンフリクトが起きると？
  - GitHub に提出した Pull requests が自動的にマージできない

■コンフリクトへの基本的な対処法

- 初心者は、演習の最初の方では「他人と同じファイルを編集しない」ことにして、操作になれる
  - 上達したら積極的にコンフリクトを起こしてみて、その解決方法を学ぶ
- コミットはできるだけ細かく作成すると良い
  - その分，他の人とかち合う可能性が減る

■GitHub でのコンフリクトの解消方法

- new_feature ブランチで作業中であり、最新の更新は commit 済とする
- 解消するための操作は次のとおり
  - 1 行目〜2 行目:master を最新版にする．2 行目:リモートの master を shizuka にマージ．3 行目:コンフリクトを解消する．4〜6 行目:このブランチを再度 push
- これにより，プルリクエストがマージ可能になれば成功

```
git checkout master
git pull
# コンフリクトの起きたファイルを編集
git add .
git commit -m 'Merge'
git push
```

2.3 （参考) より進んだ使い方

2.3.1 ファイルの削除と名前の変更

■Git に無視させたいファイル

- ツールが生成する中間ファイルなど，Git で管理させたくないファイルは予め「.gitignore」ファイルに記述しておく
- なお，「.gitignore」ファイル自体は Git がトラッキングするファイルに含める
- .gitignore の書き方については各自で調べよ

■Git が追跡するファイルの削除と名前の変更

- Git が追跡しているファイルであっても，シェルの rm コマンドや mv コマンドで削除や名前の変更をしてよい
- 「git add .」コマンドを実行すると，Git は削除や名前の変更も自動的に検知する
  - 「git rm」や「git mv」は使わなくてよい

2.3.2 操作を取り消すコマンド

■Git で行った操作の取り消し

- まちがって
  - ファイルをステージングさせた！
  - ステージングをコミットした！
- などの場合，操作を取り消すことができる
  - 特定のファイルの変更の取り消し
  - 特定のコミットの取り消し

■HEAD によるコミットの指定

- 特定のコミットの ID を指定する方法に「HEAD」を使った相対指定がある
  - show コマンドで確認しながら用いると良い（下記はサンプル）

```
git show HEAD~1
git show HEAD~1^2
```

■ステージング/コミットの修正　ファイルのステージングを取り消す

```
git reset $FILE
```

$COMMIT_ID より後のコミットの取り消し（ローカルは保存）

```
git reset $COMMIT_ID
```

$COMMIT_ID より後のコミットの取り消し（ローカルの変更も破棄）

```
git reset --hard $COMMIT_ID
```

■誤って編集や削除や修正したファイルの回復

- file.txt を誤って編集や削除した場合
- add する前
  - ステージング領域からの取り出し

```
git checkout file.txt
```

- add した後
  - 直近のコミットからの取り出し

```
git checkout HEAD file.txt
```

■push 済みのコミットの取り消し

- 最後に行ったコミットが理由でコンフリクトが発生したような場合，次の操作により，「取り消しコミット」を作成することができる

```
git revert HEAD
```

- 最後の作業が取り消されていることを確認
- その後は，この取り消しコミットを push すると，リモートでの変更内容も取り消される

2.3.3　作業の一度中断と再開

■やりかけの作業の stash

- あるブランチで作業中に他のブランチに一時的に移動したいことがある
  - 作業の途中で master ブランチを最新にする，など
- このような場合，git stash コマンドが活用できるので調べてみよ

2.3.4　その他知っておくと良いコマンド

■チートシートにある残りのコマンド

- 以下，Git チートシートにあるコマンドで，ここまで取り上げなかったものを取り上げる

■ステージングしたファイルの差分表示

- git add でステージングすると git diff で差分が表示されない
- この場合，次のコマンドで確認できる

```
git diff --staged
```

■特定のファイルにのみ関連する履歴確認

- 将来，複数のファイルを履歴管理するようになったら特定のファイルの履歴のみ確認したい
- その場合，次のとおりにする

```
git log --follow README
```

■リモートブランチの最新情報を取得

- git fetch はリモートにあるブランチの最新情報をローカルに取ってくるコマンド
- 例えば，git pull は master ブランチで次の 2 つを実行することと同じ意味

```
git fetch origin
git merge origin/master
```

■git ls-files

- 省略

3　GitHub 演習（チーム演習編）

3.1　チーム編集の準備

3.1.1　演習のための準備

■概要

- この演習では，最終的に全員で HTML による Web サイトを作ることを目指す
- Web サーバは使わず，スタティックなサイトで構わない
- 可能ならば CSS や JavaScript を使っても良いが必須ではない

■リモートリポジトリの作成

- チームの代表者 1 名が GitHub でリポジトリを作成する
  - 名前は「team_project」とする
- 次の手順で作成する
  - Creating a new repository - User Documentation
- README とライセンスを追加すること
  -「Initialize this repository with a README」にチェックを入れる
  -「Add a license:」から「MIT License」を選ぶ

■コラボレーターの追加

- 代表者は残りのメンバーを協同作業者（コラボレータ）として追加する
  - Inviting collaborators to a personal repository - User Documentation
- 招待されたメンバーには確認のメールが届く
  - これにより，全員が GitHub のリポジトリに push できるようになる

■リポジトリの clone

- 全員，リポジトリをローカルに clone する
  - Cloning a repository - User Documentation

```
cd ~
git clone $GITHUB_URL
cd team_project
```

3.2　チーム演習

3.2.1　チーム演習について

■課題 1: GitHub の Issue/Wiki を学ぶ

- リポジトリの Issue 機能を使ってみよう
  - 一人 1 つ Issue を登録する
  - メンバーの Issue に挨拶する（投稿する）
  - 終わったら Issue を閉じてみる
- リポジトリの Wiki を使ってみよう
  - Wiki を使ってチームメンバーの自己紹介をしてみよう
- なお，この演習にあまり時間をかけてはならない

■課題 2: まずは全員 1 回コミットしよう　[この課題では練習のため master ブランチをそのまま使う]

- 全員，1 つファイルをコミットしてプッシュする
- まず，コミットするファイルを作る
- ファイル名は各自の GitHub のアカウント名＋「.html」とする
- 下記の$MY_FILE をファイル名に置き換えて実行
- コミットメッセージも書く

```
git add $MY_FILE
git commit -m '＜メッセージ＞'
git push
```

- 全員完了したらローカルの master ブランチを最新にする

```
git pull
```

- ローカルにファイルができているか確認

■課題 3: ブランチを push する

- 全員 1 回，最初の GitHub Flow を成功させよう
- まず，master が最新版であることを確認
-「作業の内容がわかりやすい名前」でブランチを作る
- $MY_FILE に，中身を追加してみよう（内容は何でも良い）
- git add/commit/push を正確に実行しよう
- ブランチが無事 push できたら，GitHub をブラウザで確認する

■課題 4: いよいよプルリク

- プルリクエストを出してみよう

- 他のメンバーのプルリクエストにコメントしてみよう
- コメントには顔文字なども利用できるので活用してみよう
  - やり方は各自で調べること

■課題 5: そしてマージ

- マージしてみよう
- この段階でコンフリクトが出ることはないはず（同じファイルを編集していない）だが，もしマージできない場合は，プルリクエストを削除し，課題 3 からやり直す

■課題 6: 何回も繰り返す

- 同じファイルに更なる変更を加え，GitHub Flow を回してみよう
- これを最低 3 回は繰り返したい

■課題 7: ぼちぼちコンフリクト

- 誰かが空の「index.html」ファイルを作成する
- 全員で index.html を編集してみよう
  - $MY_FILE へのリンクを貼る
- push してプルリクエストを出してみる
- 何人かはコンフリクトになるはずだ

■課題 8: コンフリクトの解消

- コンフリクトが出たメンバーは，それを解消してみよう
- コンフリクトが出なかったメンバーは，コンフリクトが出ているメンバーの作業を見る
  - 困っていたら助けてあげよう

■課題 9: Web サイトを作ってみよう

- チームで内容を相談し，Web サイトを作ってみよう
- index.html や$MY_FILE 以外にもファイルを追加して素敵な Web サイトを作ろう

■注意事項

- 実は，GitHub では，git コマンドを使わなくても，ブラウザベースでファイルのアップロードや編集，コミットの作成などができるが，このことに気がついてはならない
  - 万が一，気がついてしまったものはしょうがないものとする