GitHub 入門

産業技術大学院大学
中鉢欣秀

2016-08-14

概要

これは Git の初心者が，基礎的な Git コマンドの利用方法から，GitHub フローに基づく協同開発の方法までを学ぶ演習である．

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．また 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．

第 1 章は Git 初心者（初めてさわる者）を対象に基礎を学ぶ．第 2 章は個人による GitHub の初歩的な使い方を取り扱う．第 3 章ではチームによる GitHub の使い方を知ろう．

1　Git 入門

1.1　Git の操作方法と初期設定

1.1.1　はじめに：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

1.1.2　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
```

1.1.3　名前とメールアドレスの登録

- 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
```

1.1.4　その他の設定

- 次のとおり，設定を行っておく
  - 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）

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

1.1.5　設定の確認方法

- ここまでの設定を確認する

```
git config -l
```

1.2　Git のリポジトリ

1.2.1　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
- Git のリポジトリバージョン管理ができるようになる
- GitHub と連携させることで共同作業ができる

1.2.2　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
- 本章では 1. について解説する（次章からは 2. で行う）

1.2.3 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

```
1 mkdir ~/my_project
2 cd ~/my_project
3 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

1.2.4 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

```
1 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

1.2.5 「.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
  - むろん，削除もしてはならない

1.3 コミットの作成方法

1.3.1 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，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

1.3.2 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

```
1 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

1.3.3 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

```
1 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

1.3.4 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
  - 変更をコミットするためには，ステージングしなくてはならない
  - 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

1.3.5 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

1.3.6 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

1.3.7 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'
```

1.3.8 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

1.4 変更履歴の作成

1.4.1 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

1.4.2 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

1.4.3 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

1.4.4 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
- ステージさせると git diff の結果が空になる
  - この場合，「git diff –staged」で確認可能

1.4.5 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'
```

1.5 履歴の確認

1.5.1 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

1.5.2 一つのファイルの履歴

- 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
```

1.5.3 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（概ね先頭 4 文字でよい）
  - ブランチごとの比較もできる（後述）

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

1.5.4 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
```

1.6 ブランチの使い方

1.6.1 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

1.6.2 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

1.6.3 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」が分かるような名前を付ける

1.6.4 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

1.6.5 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

1.6.6 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

1.6.7 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

1.6.8 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

1.7 TODO 操作を取り消すコマンド

1.7.1 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
```

2 GitHub 入門

2.1 GitHub とは

2.1.1 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

2.1.2 GitHub アカウントの作成

- まず, 次の URL の指示に従いアカウントを作成
  - 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

2.1.3 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
- 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
- もし SSH の登録がうまく行かなかったら，HTTPS を用いて接続し，GitHub のユーザ名とパスワードでアクセスできる

2.2 リモートリポジトリ

2.2.1 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

2.2.2 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

2.3 GitHub flow

2.3.1 GitHub floww による開発の流れ

- GitHub flow
  - Understanding the GitHub Flow ·GitHub Guides
- 言葉による説明
  1. リモートリポジトリをローカルに複製
  2. master から作業用ブランチを作成
  3. ブランチで編集作業
  4. ブランチでコミットの作成
  5. ブランチをリモートに送る
  6. GitHub でプルリクエストを作成
  7. GitHub でレビュー（＋自動テスト）
  8. GitHub でプルリクエストをマージ
  9. ローカルの master を最新にする（2. に戻る）

2.3.2 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 にある our_project リポジトリの URL にして実行
  - リモートの URL はブラウザで確認する
  - ssh 接続の場合，URL は「git@...」（HTTPS の場合「https:...」

```
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
  - 別なマシンで作業したいときなどは話は別

2.3.3 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
```

2.3.4 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
```

2.3.5 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

2.3.6 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（push）
  - 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
```

2.3.7 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

2.3.8 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

2.3.9 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

2.3.10 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
```

- 練習のため，ここで手順 2: に戻り，一連の作業を複数回繰り返すこと
  - 体に叩き込む！

2.4 コンフリクトについて

2.4.1 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
- 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）
  - Pull requests でコンフリクトが発生し、自動的にマージできない状態になったら、その PR を送った人がコンフリクトを自分で解消する（あるいは解消方法をメンバーに聞く）

2.4.2 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: 一度 master ブランチに移動．2: 手元の master を最新版に．3: 作業中のブランチへ．4: ここで master を手動でマージ．コンフリクトが発生するので解消する．5: 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 origin master
git checkout new_feature
git merge master
git push origin new_feature
```

2.4.3 コンフリクト解消の練習

TODO

3 TODO GitHub によるチーム開発

3.1 TODO チーム開発

3.1.1 チーム編成

- ここまでの演習内容が終わったものは教員か TA に教えること
- 終わったものから順番にチームを編成する
- チームができたら代表者 1 名が GitHub でリポジトリを作成する
  - 名前は「team_project」とする

3.1.2 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
  - GitHub のリポジトリをブラウザで開く．
  - Settings -> Collaborators を選ぶ
  - メンバーを招待する
  - 招待されたメンバーには確認のメールが届くので，リンクをクリックする

4 TODO 演習課題

4.1 ペアで行う GitHub

4.1.1 課題 1：ペアで GitHub を使ってみよう

1. 隣同士でペアを組む
2. レポジトリを作成する（どちらか一方）
   - `bundle gem` でひな形を作る（初心者は Gem でなくても良い）
3. レポジトリの Collaborators に登録する
4. レポジトリに対して、次のことを行う
   - Pull requests を利用してみる
   - Issue を利用してみる
   - Wiki を利用してみる

4.1.2 課題 1 の続き

1. Pull request & merge の作業を各自 5 回以上行う
   - ディスカッションやコードレビューもやってみる
2. Issue を 5 個以上登録する
   - Pull request による Issue の close なども試す
3. Wiki でページを作成する
   - ページを 5 つ程度作成して、リンクも貼る

4.2 グループで行う GitHub

4.2.1 課題：グループで GitHub (1)

1. ペアを 2 つ組み合わせて 4 人グループを作成する
   - 課題 1 が終わったペアから順番にグループ編成
2. 作りたい Gem について相談して仕様を決める
   - テーマはなんでも良い
     - Web API を利用したコマンドラインツールなど
   - ある程度の役割分担も決めておく
3. レポジトリを作成する（代表者 1 名）
   - コラボレーターを追加する
4. 今まで学んだ知識を活用して Gem を開発する

4.2.2 課題：グループで GitHub (2)

1. グループメンバーで Gem を共同で作成する

2. GitHub Flow の実践
3. Travis CI によるテストの自動化
4. RubyGems.org への自動ディプロイ
5. その他、GitHub の各種機能の活用

5 TODO 演習の成果物の提出

5.1 TODO アカウントの作成

5.1.1 課題

GitHub にアカウントを作成せよ

5.1.2 提出

TODO: Google form