

# Git によるバージョン管理入門

田中 健策（株式会社 RAKUDO）

## 第二回

# git を使ってみよう

前回の講義の課題をクリアしていれば、  
<https://github.com/tannakaken/nugitlecture2020> のコラボ  
レーターになっているはず。  
(講義の資料のソースは  
<https://github.com/tannakaken/gitlecture> においてあり  
ます)。  
それでは、このリポジトリをローカルにクローンしてみましょう。  
CLIを使っている人は  
`git clone https://github.com/tannakaken/nugitlecture2020.git`  
でいいです。  
GUIを使っている人はそれぞれの環境で試してみましょう。

# ファイルを push してみましょう

clone したフォルダになんでもいいからファイルを作って commit しましょう。(commit の仕方は web で調べれば出てきます)。それを github に push してみてください (git にメールアドレスと名前を設定する必要があるかもしれません)。もし、push できなければ一度 pull してください。conflict が起きた、と言われたら次のページを見ましょう。

# conflict とは何か？

リポジトリに違う場所から矛盾する変更を加えると **conflict**（衝突）が起こります。

例えば Web ブラウザに表示した github のページ上とローカルで矛盾する変更を加えれば人工的に conflict が起こせます。

git はリモートで conflict が起こらないようにします。なので、pull すべき内容があるときに push することはできません（リモートで conflict が起こる可能性があるのです）。

pull して、conflict が起こった場合、それを解消するまで push することはできません。

# conflict の例

例えば git pull をして conflict が起こると、ファイルが次のようになります。

```
<<<<<<< HEAD
goodbye world
=====
hello world
>>>>>>> 943145c978f2c15e0ee82ec7baae9671dfdec54e
```

HEAD は現在のレポジトリの先頭  
下は HEAD と衝突している commit の番号です。

# conflict を解消する

これを github に push することはできません。  
どちらかいない方を消すか、またはより正しい記述に修正するかして、**conflict** を解消しないといけないのです。  
もし大人数で一つのリポジトリを同時に編集していくと、誰かが編集するたびに conflict が発生する可能性があります。  
これを俗に「**conflict 地獄**」と呼ぶ

# branch を分ける

なので、編集する目的に分けてリポジトリのコピーを作り、別々に編集します。それを **branch** (枝) と言います。  
大規模なチームでは、細かく branch を分けます (**branch を切ると**という謎の方言を使う人も多い)。  
この branch の分け方の手順を **flow** といい、いくつか考案されています。

# 分けた branch を merge する

branch を分けることで、一つ一つの branch ではそれほど衝突の危険を冒さずに編集できます。

そして branch の編集が終わったらそれを、main などの名前のついた中央 branch に **merge** (融合) させます。

その時に conflict が発生すれば、解消します。

これによって、conflict 地獄に陥らずに、編集ができるわけです。また編集途中でも中央 branch は汚れていないので、他の人がそこから別の編集を始めるときにもやりやすくなります。

しかし、branch を分けて長く merge しないままにすると、merge した際に conflict やその他の問題が起きやすくなります。

ですので適切な flow が必要になるわけです。



# commit の粒度

一つの commit はあまり大きくないことが推奨されます。  
何か困ったことがあったあった時に、どの commit が原因かがわかりやすくしたいです。  
また diff で差分を調べたときにも、見るべき場所が少なくてわかりやすい方が良いです。  
一つの commit は一つの変更、commit のコメントが一言で言えるようにすると良いと言われています。  
でも、今回はプログラミングではなく文書なので、正しいやり方は世間でもまだ見つかっていません。

# その他 git は奥深い

git には過去を改変したり、過去のグラフを繋げ変えたり、消えてしまった過去を修復したり、変更を一時的に記録したり、と様々な奥深い機能があります。

今回はそこまで行くつもりはませんが（もし必要になったら、この授業がかなり失敗してる証拠かもしれません）、もしかしたら将来必要になるかもしれません。

git は変更を木状に分岐し、それを融合させてグラフ状になるという構造をしています。

それ自体数学的になかなか面白い構造をしているので、興味があったら調べてみるといいかもしれません。

# 今週の課題

- なんでもいいから github に push してみよう。
- もし conflict が起こったら、解消して push しましょう。

github に push ができていれば合格です。

もしまだコラボレーターになっていない方は、前回の課題からやり直してください。