GitHubとコンフリクト

J23072　蝦名武尊

# 初めに

このドキュメントの目的はGitHubを用いて開発を行う際の初期設定や機能についてまとめることである。また、コンフリクトとは何か、起こる理由についてもまとめることを目的としている。

# GitHub

GitHubとはGitで管理されたファイルをインターネット上にアップロードし、取り扱うことができるプラットフォームである。GitHubには様々な機能が搭載されているが、今ディレクトリではチームでのコード管理機能をピックアップして解説を行う。

## 2.1 メリット

　GitHubを使う主なメリットは、Gitで管理しているコードを複数人で管理・運用することができることである。Gitのみを用いてコードの管理を行う場合、自分のPC内に存在するコードに対しては編集履歴の作成やデータの復元などをコードの管理を容易に行うことができる。しかし、他人のPC内にあるコードに対して編集履歴の確認や自分のコードとのマージの実行などをGitのみで行うにはとても面倒な作業が必要になる。

　そこでGitHubを使用することで作業の面倒さが格段に解消される。インターネット上にあるGitHubにコードをアップロードすることでWeb上で他人のコードの編集履歴が閲覧できるだけでなく、GitHubからダウンロードすることで自分のコードにマージすることもできる。

　また、GitHub上にコードと履歴をアップロードすることでバックアップの保管場所として使用できることも大きなメリットである。

## 初期設定

GitHubを使用するにはGitHubアカウントが必要である。アカウントを作成するにはGitHubの公式サイト（<https://github.com>　）にアクセスし、「Sign up」と記載されたボタンをクリックし、メールアドレスやパスワード、ユーザーネームを入力する必要がある。その後、入力したメールアドレスに送信された確認コードを入力し、いくつかの質問に答えればアカウントの作成は完了である。アカウント作成後にセキュリティの向上のため、2段階認証を有効にしておくべきである。

　次にパーソナルアクセストークンの設定を行う必要がある。パーソナルアクセストークンとはGitHubアカウントのユーザー名とパスワードの代わりに使用することができる認証トークンのことである。設定には画面右上のアイコンをクリックし、『Settings → Developer Settings → Personal access tokens → Tokens(classic)』と進んでいく。そして、画面中央に表示されている『Generate new token』をクリックし、『Generate new token(classic)』を選択する。その結果、トークンの設定に移るので「Note」にはトークンの名称、「Expiration」には使用期限を決定し、「Select scope」にてアクセスの種類を指定することができる。全ての設定を終えるとトークンが表示されるので、それをコピーし安全な場所に保管することで初期設定は完了である。

## リモートリポジトリ

　リモートリポジトリとはGitHub上で管理しているリポジトリであり、Web上で作成することができる。反対に自分のPC内で管理しているリポジトリをローカルリポジトリといい、管理したいコードが格納されているファイル直下で『git init』コマンドを実行することで作成することができる。

　リモートリポジトリとローカルリポジトリは連携することが可能である。連携を行うことによりローカルリポジトリのあるブランチの特定のソースコードをリモートリポジトリのブランチにアップロードすることができるようになる。このアップロードを「push」と呼ぶ。

また、反対に他人が作成したリモートリポジトリのmainブランチをローカルにダウンロードすることもできる。このダウンロードを「clone」と呼ぶ。一度ダウンロードされると自動でローカルリポジトリが作成され、GitHubとの連携も行われる。

さらにリモートリポジトリ上で特定のコードに変更があった場合にその変更をローカルリポジトリに反映することができる。このことを「pull」という。

## 2.4 基本操作

この章ではGitHubの基本的な操作を実際の画面のキャプチャを用いて説明していく。

### 2.4.1ソースコードをGitHub上で管理する

　ここではすでにローカルリポジトリ上に存在するソースコードをGitHubにアップロードして管理する場合の操作を説明する。今回は具体例として「test」というフォルダの中にある「test1」というフォルダを使用する。このフォルダには「code\_1.py」、「code\_2.py」というファイルが格納されており、この二つのコードをGitHub上で管理する。

グラフィカル ユーザー インターフェイス, テキスト, アプリケーション

自動的に生成された説明

　まず初めに、自分のGitHubアカウントのダッシュボードに移動し、画面右上の「＋」マークから『New repository』を選択する。

グラフィカル ユーザー インターフェイス, テキスト, アプリケーション, メール

自動的に生成された説明

移動が完了すると上の画面が表示される。「Repository name」の欄にリポジトリの名前を入力し、各種設定を決定していく。それぞれの設定が完了したら、ページ最下部の「Create repository」を選択することでリモートリポジトリの作成が完了する。

　次はローカルリポジトリの作成を行う。

テキスト

自動的に生成された説明

　cdコマンドでtest1ファイル直下に移動した後にinitコマンドでローカルリポジトリの作成を行う。この時、test1ファイルに隠しファイルとして.gitファイルが作成される。

テキスト

自動的に生成された説明

その後、addコマンドでソースコードをステージングエリアに追加し、commitコマンドでコミットを行う。コメントの入力時にシングルクォーテーションを誤って使用しないように注意しておくべきである。

　次はローカルリポジトリとリモートリポジトリを結びつける。

テキスト

自動的に生成された説明

始めに、statusコマンドを実行して現在のブランチがmainである事を確認する。その後、作成したローカルリポジトリとリモートリポジトリを結びつけるために以下のコマンドを実行する。

git remote add リモートリポジトリ名 リモートリポジトリURL

この時、リモートリポジトリ名は通常「origin」とする。「origin」とは起源という意味であり、このリポジトリはソースコードの原点であるという意味を示している。また、リモートリポジトリのURLは先述したGitHubのリモートリポジトリの作成完了画面に表示されているHTTPSを使用する。しかし、そのまま使用するのではなく、以下のように改変する。

https://ユーザー名:アクセストークン@github.com/XX/XX.git

　次にリモートリポジトリにファイルのプッシュを行うために以下のコマンドを実行する。

git push -u リポジトリ名　ローカルブランチ名:リモートブランチ名

実行時、ローカルブランチ名とリモートブランチ名が同じ場合、”:リモートブランチ名”を省略することができる。また、”-u”というオプションは上流ブランチを指定しており、このプッシュ以降のpushコマンドやpullコマンドの実行時にブランチ名の指定を省略することができる。

　以上の操作ですでにローカルリポジトリ上に存在するソースコードをGitHubにアップロードして管理することができる。

## GitHubからソースコードをダウンロードする

　次にすでにGitHubで管理されているソースコードをダウンロードする操作を説明する。今回は先ほどGitHubにアップロードしたtest1フォルダを具体例として扱う。

テキスト

自動的に生成された説明

　始めにローカルでダウンロードを行いたい場所まで移動し、リモートリポジトリをダウンロードを行うため以下のコマンドを実行する。

git clone リモートリポジトリURL

このリモートリポジトリURLは前述したリモートリポジトリURLと同じように以下のように記述する。

https://ユーザー名:アクセストークン@github.com/XX/XX.git

以上の動作でリモートリポジトリのデフォルトブランチをダウンロードすることができる。また、cloneコマンドの実行時にローカルリポジトリとリモートリポジトリの結び付けは完了している。

## 2.5ブランチの運用

GitHubを利用してチーム開発を行う際のブランチの運用について解説する。

　実際の開発現場において、ほとんどの場合「git-flow」というブランチモデルを簡略化した運用を行っている。今回は初心者用として１つのmainブランチと機能ごとに存在するfeatureブランチを例に解説する。

　前章にてローカルリポジトリとリモートリポジトリの結び付けは完了しており、それぞれにmainブランチが存在する状況を作ることができた。この時に重要な点はリモートリポジトリのmainブランチが実際にリリースする正式版であるということである。ローカルリポジトリのmainブランチのソースコードがリモートリポジトリのソースコードを自動で上書きすることはないので注意が必要である。

　機能の開発はmainブランチから分けて開発していく。新たなブランチを作成する際にはローカルリポジトリとリモートリポジトリのmainブランチの内容が一致しているか確認を行い、一致していなければpullコマンドにて更新をかける必要がある。最新であることを確認したならば、mainからブランチを切り、新しい機能を開発するためのfeatureブランチを作成する。featureブランチは1ブランチにつき、1機能や１つのバグ修正など区切り良く開発することができる単位で作成される。

　開発を進め、コミットを行い、一段落した段階でpushコマンドでリモートリポジトリにソースコードをアップロードする。この時にローカルリポジトリとリモートリポジトリのブランチ名を統一しておくことで操作のミスを減らすことができる。

### 2.5.1 操作例

　上記のブランチの運用の実際の操作例を記述する。例として使用するファイルを「test1」、新たに追加するソースコードを「code\_3.py」とする。

テキスト

自動的に生成された説明

　現在のブランチがmainであることを確認し、ソースコードが最新でなかった場合には次のコマンドを入力する。

git pull リモートリポジトリ名 ブランチ名

上のキャプチャではpull以降は省略されているが、これは前章の操作にてpushコマンドを実行した際に”-u”オプションを付与し、上流ブランチをあらかじめ指定しておいたからである。

　最新であることを確認した後、機能開発を行うためのブランチを作成するために次のコマンドを入力する。

git checkout -b 新しいブランチ名

-bオプションを使用することにより自動で新たに作成したブランチに移動することができる。

　移動したブランチ下で新たにtest1フォルダに「code\_3.py」ファイルを作成した後は基本操作通りにaddコマンドとコミットを行い、ローカルリポジトリのfeatureブランチをリモートリポジトリのfeatureブランチにプッシュを行う。

テキスト

自動的に生成された説明

プッシュを行う際にリモートリポジトリに指定したブランチが存在しない場合、自動でリモートリポジトリにブランチの作成を行う。また、プッシュはコミットを行い次第実行するのではなく、ある程度区切りが良い場面で実行すべきである。

## 2.6コンフリクト

　コンフリクトとはマージする際に、複数の開発者が編集した同じファイルの内容が異なるためにどの変更を採用すべきか自動的に判断できない状態のことを指す。コンフリクトが発生した場合、Gitが自動的にどちらの変更を採用するか判断することは不可能であるので手動で解決する必要がある。

### 2.6.1 発生要因

　コンフリクトが発生する主な要因はマージやプルなど複数のブランチを一つにまとめるときに「同じ場所を異なる方法で変更した」ことである。以降は具体例である。

1. 同じ行の変更

条件：複数のブランチで同じファイルの同じ行を変更した場合

例：featureブランチでa.pyの２行目を変更し、mainでも同じ行を変更した場合

a.py a.py

# main #festure

a=3 a=3

print(a) print(a)

1. 同じファイルの変更

条件：同じファイルが異なるブランチで変更された場合はそれぞれ変更箇所が異なる限りGitが自動でマージする。ただし、変更内容に依存関係があったり、同じ部分で変更が異なる場合に発生

例：featureブランチでab.pyの関数calcを変更し、mainでも関数calcに修正があった場合

ab.py ab.py

#main #feature

def calc(x,y) def calc(x,y)

result = x + y result = x - y

return result return result

1. ファイルの削除と変更

条件：片方のブランチで削除したファイルを別のブランチで変更を加えた場合

例：featureブランチでabc.pyを削除し、mainでabc.pyに変更を加えた場合

1. ファイル追加と削除

条件：片方のブランチでファイルを追加し、別のブランチでそのファイルを削除した場合

例：featureブランチでabcd.pyを追加し、mainでabcd.pyを削除した場合

1. 同じファイルを移動または名前の変更

条件：同じファイルが異なるブランチで移動または名前を変更され、両方の変更がマージされる場合

例：featureブランチでabcde.pyをxyz.pyにリネームし、mainでabcde.pyを別のフォルダに移動した場合

1. ブランチ間の履歴の違い

条件：複数のブランチで独立した開発が進行した場合、同時に別の変更を行うことで履歴にギャップが生じることで発生する

例：mainからfeature-Aブランチとfeature-Bブランチを作成し、それぞれで別の変更を行いmainにマージした場合

1. マージの順序

条件：片方のブランチを先にもう一方のブランチにマージし、その後に逆方向にマージを行い。両者の変更が衝突した場合

例：featureブランチを先にmainにマージし、その後、mainをfeatureブランチにマージする場合

## 2.7 マージ

　ここではマージの動作について解説する。マージの操作について、コンフリクトが発生している場合としていない場合で異なるのでそれぞれ解説を行う

### 2.7.1 コンフリクトが発生しない場合

　ローカル上で開発が完了したブランチをGitHub上にプッシュし、GitHub用のブランチをmainにマージする。この時、マージの操作はGitHubのWebページに表示されている「pull request」から行う。「pull request」を行うと、自分以外の開発メンバーに通知が送信されレビュアーとして確認を要求する。ソースコードに問題がなく、マージが容認されたらレビュアーは「Approve」を行い、それがレビューを公開した側に通知され、「Merge pull request」を行うことでマージが実行される。

### 2.7.2 コンフリクトが発生した場合

　コンフリクトが発生した場合、マージを実行する直前にコンフリクトが発生した旨のメッセージが送信される。コンフリクトの解消はWeb上でも可能であるが安全性を考慮し、ローカル上で行うべきである。

　コンフリクトの解消にはいくつかの段階が存在する。

1. ローカル上で作業するブランチをmainに切り替え、pullコマンドで状態を最新にする
2. ローカル上でコンフリクトのが発生したブランチに切り替え、そのブランチにmainをマージする
3. コンフリクトが発生するのでこれを解消する

以上の操作を行い、GitHubにてマージを実行すればよい。

# 3. 終わりに

　以上のようにGitHubを利用することでチーム開発の効率が向上し、別々の開発者が作成したソースコードの管理を行うことができる。また、コンフリクトの発生要因やその解消法を理解することにより発生する確率を減らし、もし発生してしまっても的確に対処を

行うことができる。

　しかし、GitHubを利用する上でプッシュするソースコードに機密性が高い内容が含まれていないか、ソースコードの公開範囲を間違えていないかといったセキュリティ上の面で注意が必要である。

# 参考文献

・Pythonプログラミング VTuber サプー.” **【GitHub入門】初心者向け！GitHubでチーム開発するための基本操作を解説！.**”Youtube,2022/10/30, <https://youtu.be/Dz95iUNt-fg>

・株式会社Nuco.” **【徹底解説】これを読めばもう怖くないGitコンフリクト**

.”Qiita,2024/12/10, <https://qiita.com/tgc_nuco/items/c65f36e15243535c095b>