Class 7 -6

Git

バージョン管理ツール

- **バージョン管理**[Version Control]ツールは、ディレクトリ（フォルダ）やファイルの変更履歴を完全に保存・追跡するバージョン管理機能を提供し、ソフトウェア開発者にとって必須のツールであり、ソフトウェア会社にとってはインフラとなるものとでも言えます。

バージョン管理ツールの機能

- バージョン管理の最も重要な機能は、**ファイルの変更を追跡**することです。
- バージョン管理ツールは、**いつ**、**誰が**、**どのファイルのどこ**を変更したかを忠実に記録します。ファイルが変更されるたびに、そのファイルのバージョン番号が増加します。
- もう一つの重要な機能は、**並行開発**であります。
- ソフトウェア開発は、多くの場合、複数人の共同作業で行われます。バージョン管理は、異なる開発者間のバージョンの同期と開発コミュニケーションの問題を効果的に解決し、共同開発の効率を高めることができます。 並行開発でよくあるバグ修正の問題も、バージョン管理ツールで**ブランチマージ**を行うことで解決することができます。

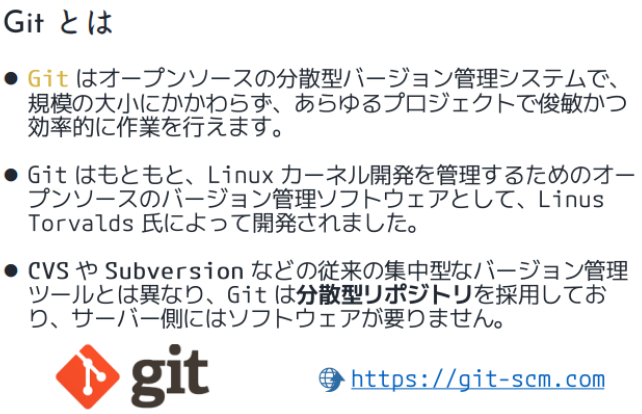
スライド読み上げるで OK

バージョン管理ツールの役割

- 共同開発：チームで同じプロジェクトに取り組む。
- バージョニング：プロジェクトのバージョンを継続的に増やしていく形で、段階的にプロジェクトを完成させる。
- データのバックアップ：開発されたすべてのバージョンの履歴を保存している。
- 権限管理：チームの開発者に異なる権限を割り当てる。
- ブランチマネジメント：開発チームが複数のラインで同時にタスクを進めることで、さらなる効率化を図ることができる。

役割として、何が一番良いかというと、誰が開発したかや更新履歴も残るし、

同時に開発できて、それが一目瞭然でわかるから、プロジェクトの管理そのものがしやすいよね



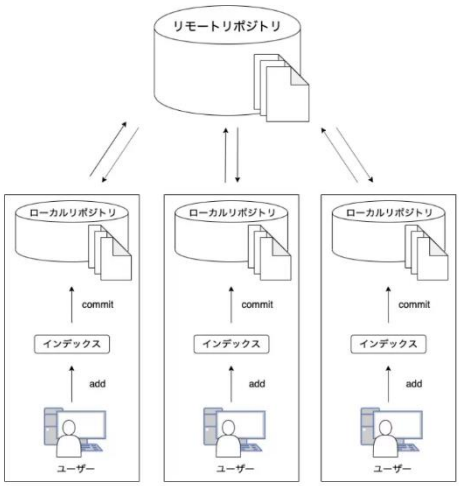
Git とは、分散型バージョン管理システムです。といっても分かりにくいのですが、ざっくりいうとファイルのバージョン管理が簡単にできるツールといえます。バージョンとはそのファイルをアップデートしたり更新した時に変化するアレですね。iPhone なんかでも最新バージョンにアップデートするように通知が来たりしますよね。ただ、アップデートしてしまうと基本的には、古いバージョンに戻すことはできませんし、するとしても手間がかかります。

しかし、Git で管理しているファイルであれば、コンピューター上でファイルの編集履歴を管理できるので、編集前のファイルを残したまま、新しく編集したファイルを保存することができます。なので古いバージョンから新しいバージョンまでの管理が簡単です。

プログラマーにとっては、多くのコードを編集した上で何か不具合が起きたときに、元のバージョンへ戻すことは日常茶飯事です。かといって、ひとつひとつコードの編集の度に古いバージョンの日付や時刻ををつけて保存して、また新しい作業をするようなことをしていては、時間はかかりますし、人的ミスも増えることは、いうまでもありません。そういった作業を無駄なく、効率的に行うためのツールが Git です。

補足資料を挟んで説明をする

Git を使い始める前に、図を参考にしながら、最低限押さえておきたい用語を解説します。



1：リポジトリ

リポジトリとは、ファイルやディレクトリを入れて保存しておく貯蔵庫のことです。Git における

リポジトリは以下の 2 種類に分かれています。

- リモートリポジトリ:特定のサーバー上に設置して複数人で共有するためのリポジトリです。
- ローカルリポジトリ:ユーザーごとに配置される手元のマシンで編集できるリポジトリです。

2 種類のリポジトリに分けることで、普段の作業はそれぞれのユーザーが手元のローカルリポジト

リで行い、作業内容を共有するときにリモートリポジトリで公開するという使い方になります。リ

モートリポジトリを介して他のユーザーの作業内容を把握することも可能です。上の図の一番上の

部分ですね。

２：コミット

コミットは、ファイルやディレクトリの編集作業をローカルリポジトリに記録するために必要な操作のことです。コミットを実行するとファイルを編集した日時を記録したファイルが生成されます。コミットを実行するごとにファイルが生成され、時系列順にならんで格納されるので、ファイルを編集した履歴やその内容を確認することができるわけです。

3：ワークツリーとインデックス

ユーザーが編集している作業中のディレクトリのことをワークツリーといいます。また作業場所であるワークツリーと、保存場所であるローカルリポジトリの間には、インデックスという中間領域が設けられています。

Git の仕様上、ワークツリーで編集したファイルをコミットしたい場合は、一度インデックスに登録しなければなりません。編集したファイルをリポジトリへコミットする前にインデックスへ登録して仮置き（add）しておくようなイメージです。

ワークツリーからリポジトリに直接保存するとミスが増えますし、効率が良くありません。というのも編集作業を行うファイルは、一つだけとは限りませんし、編集した複数のファイルを一つ一つ確認してコミットするのは、作業的にも非効率ですからね。

コミット予定のファイルをインデックスに仮置きしておけば、後からまとめて確認した上でコミットできるので、編集したファイルのコミットし忘れなどを防ぐことができますし、余分なファイルを含めずにコミットできるというわけです。

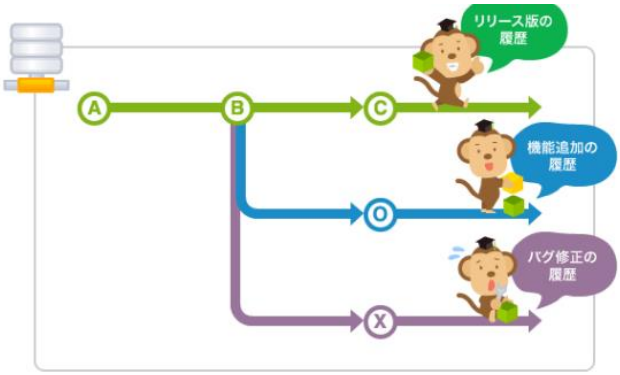
・クローン

一言でいうとダウンロードに近いものと思っていただければ結構です。複数人で共有しているファイル（リモートリポジトリ）をまるごと自分のローカル環境（ローカルリポジトリ）に保存する機能ですので、ほとんどの場合 Git で最初に行う作業になります

・プッシュ

プッシュとは、ローカルリポジトリにあるファイルをリモートリポジトリに送信して保存する機能です。いわゆるアップロードに近い感覚ですね。

誰かが共有しているファイルをクローンして、ワークツリーで作業したファイルをインデックスに一度仮置きし、まとめてローカルリポジトリに登録（コミット）する。ローカルリポジトリにコミットしたファイルを共有するためにリモートリポジトリにプッシュするのが基本的な流れになります。



参考: https://backlog.com/ja/git-tutorial/

いったん、Lighthouse のスライドに戻る

・ブランチ

ファイルの編集履歴を分岐させて記録していく機能のことです。WEB サービスやソフトウェアの開発において、バグの修正や、機能の追加などのファイル編集作業は複数のユーザーが同時に行うことも少なくありません。並行して同時に行われる作業を正確に管理するために Git にはブランチという機能が用意されています。これが Git のバージョン管理を効率的にし、間違いを減らすためにもっとも活かされている機能ともいえるでしょう。

マスターブランチと呼ばれるメインのブランチと、そこから分岐してバグの修正や、機能の追加を行っているブランチを例に挙げると下記のようになります。

・マージ

さっき説明した複数のブランチを一つにまとめて、完成形に近づけることをマージと呼びます。バグの修正や、機能の追加を行ったブランチがマスターブランチに統合されている部分のことですね。

・プル

共有されているリモートリポジトリに保存されているファイルのうち、ローカルリポジトリ（あなたのローカル環境）に無いファイルや他のユーザーが更新したファイルのみをダウンロードする機能です。**リモートリポジトリのファイルをすべて丸ごとダウンロードするクローンに対して、ローカルリポジトリとの差分のみをダウンロードして更新するのがプルになります。**

・フェッチ

リモートリポジトリからファイルの最新情報を取得してくる操作のことです。共有されているファイル（リモートリポジトリ）の更新を確認したり、複数人の作業の擦り合わせのために使う機能といえます。**プルとは違い、ローカルのファイルを更新することはありません。**

リモートリポジトリのファイルをダウンロードし更新（プル）するのか、ファイルの更新があったかどうかを確認する（フェッチ）という点が異なります。また、**前述のプルは、フェッチとマージを同時に行う機能といえるでしょう。**

何も確認せずにプル（フェッチ＋マージ）してしまうと、あなたがローカルで編集したファイルと同じファイルが更新されていた場合、エラーが出たりします。フェッチは、それらを未然に防ぎ、**複数人で同じファイルを編集しているときでもお互い干渉しないようにするための機能というわけですね。**

Git の作業フロー

- Git のファイルを作業ディレクトリに**クローン**する。
- 他の人がファイルを変更した場合、どのファイルがどう変更するかは**確認**できる。
- クローンしたファイルに**追加・変更**する。
- コミットする前に**変更点を確認**する。
- 変更を**コミット**する。
- 変更完了後、もしエラーが見つかったら、コミットを撤回し、変更し直してコミットすることができる。

まずは、gi t のファイルもしくは、フォルダをダウンロードします。

次にダウンロードしたものを変更なり追加したりして、開発をします。

そして、コミットしてエラーが発生したらもう一回コミットしなおすことができます。

Git と Github の違い

初心者の方は、Git と Github を混同されることが多いかと思いますが、実際は別物を指しています。Git を活用されている多くの方は、Github を活用されている印象ですが、以下のような違いがあります。

- Git：誰がいつどのように編集したかを正確に把握できるバージョン管理システムのこと。
- Github:Git の仕組みと連携して、他のユーザーとやりとりしやすくしている WEB サービスの名称。

端的に説明するとこのような感じですね。Github は、文字通り Git の hub（拠点）であり、世界中のユーザーが編集したコードやデザインデータを保存・共有しやすくするための WEB サービスになります。

Git は先述の説明の通り、CUI 仕様ですので、不慣れな方にとっては使いにくいです。一方で Github は GUI 仕様なので、画面上でマウスを使って操作できたり、複数のユーザーでコミュニケーションをとりやすいように機能が整備されています。このような点が Github が活用されている要因の一つでしょう。

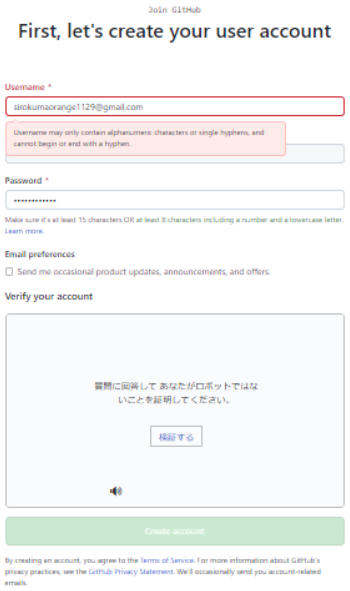
GitHub アカウント作成方法 1. https://github.co.jp/ にアクセスします。

2. 画面中央付近の「Github に登録する」をクリックしてください。

もし見当たらない場合は画面右上の「サインアップ」をクリック してください。

3. 登録画面へ進むと英語になります。画面上部の Username （ユーザ名)、Email address（自身のメールアドレス)、 Password（(設定するパスワード）を入力し、

その下の Verify your account から指示に従ってアカウント認証を行ってく ださい。入力（ 認証が完了したら画面下部の Create account ボタンをクリックしてください。



4. プランの選択画面になりますので、Free（(無料プラン）を 選択しましょう。画面下部の Finish sign up をクリックする と、仮登録が完了します。 5. 登録したメールアドレス宛に GitHub から確認メールが送 られてきます。メールに記載されている「Verify email address」をクリックしましょう。 6. 登録完了メールが届いたら本登録が完了します。

Git のインストール

- Windows: https://gitforwindows.org
- Mac（Terminal で実行）：
  - インストールパッケージの管理ツールをインストール:

```
/bin/bash -c "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

  - Git をインストール:

```
brew install git
```

- インストールを確認します。cmd または Terminal で:

```
git --version
```

ここからは、git を皆さんにインストールしていただいて、実際に作業をしていただきます。

リポジトリのクローン

- リポジトリを**クローン**するコマンドは以下の通り:

```
git clone [url]
```

- 例えば、「grit」というリポジトリをクローンする場合は、以下のコマンドを使用します:

```
git clone git://github.com/schacon/grit.git
```

- このコマンドを実行すると、カレントディレクトリに「grit」というディレクトリが作成されます。その中には、ダウンロードしたすべてのバージョンレコードを保持する「.git」ファイルを含むディレクトリが作成されます。
- クローンしたプロジェクトディレクトリに独自の名前を定義したい場合は、上記のコマンドの最後に新しい名前を指定します。例えば、クローンしたプロジェクトを「mygrit」と名付けて保存したい場合は、以下のコマンドを実行:

```
git clone git://github.com/schacon/grit.git mygrit
```

- いまのプロジェクトの**最新バージョンを**、リモートリポジトリからローカルリポジトリに**取り込む**には、以下のコマンドを実行:

```
git pull
```

https://github.com/Izwa-Akemi/Fee-Simulation

（WEB 幹事料金シミュレーションをダウンロードしたもらう）

ブランチ管理

- **ブランチの一覧**を表示する基本的なコマンドは:

```
git branch
```

- 手動で**ブランチを作成**するには、「git branch [ブランチ名] 」を実行:

```
git branch testing
```

- checkout コマンドを使って、変更**したいブランチに切り替え**ます:

```
git checkout testing
```

次へ

- 「git checkout -b [ブランチ名]」というコマンドで新しいブランチを作成し、すぐにそのブランチに切り替えてその中で操作できるようにすることも可能:

```
git checkout -b newtest
```

- **ブランチを削除**するには、「git branch -d [ブランチ名]」を実行:

```
git branch -d testing
```

リポジトリの表示と変更

- status コマンドで今の**リポジトリの状態を確認**できます:

```
git status
```

- add コマンドで、ステージングエリアに**ファイルを追加**できます:

```
git add a.txt
```

- 「add .」ですべてのファイルを追加することもできます:

```
git add .
```

- ファイルを追加できたら、status コマンドでステージングエリアにファイルの変更を確認できます。

変更のコミットとプッシュ

- 変更を**コミット**するには、commit コマンドを:

```
git commit -m "Commit Message"
```

- 「-m」オプションを省略すると、Git はテキストエディターを開き、そこに複数行のコミットメッセージを書けます。
- コミットする前に、必ず add コマンドを実行してください。
- 現在のブランチをリモートリポジトリの対応するブランチに**プッシュ**（更新）するには:

```
git push
```

ブランチマージ

- push コマンドを実行すると、リモートリポジトリには変更がうまく反映されますが、master ブランチには反映されないので、**マージ**を行う必要があります。
- 自分がプロジェクトのオーナーである場合、直接に merge コマンドを使ってローカルでマージし、マージした master ブランチをリモートリポジトリにプッシュすることができます。
- 自分がただのプロジェクトの貢献者で、マスター権限がない場合は、Git の「pull request」という機能を使ってプロジェクト管理者にマージを依頼することができます。

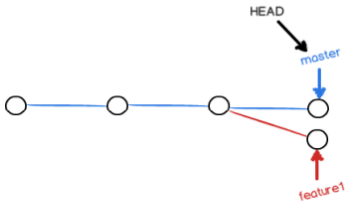
マージのコマンド

- まずブランチをターゲットのブランチに切り替え、次にマージしたいソースのブランチをマージします:

```
git checkout master
git merge test
```

- この二つのコマンドで、test ブランチを master にマージしました。

Git 衝突



- 上図のように、feature1 を master にマージしたいときに、feature1 の**変更している間に**他の誰かが master に変更を加えた可能性があります。
- その後で master ブランチにマージしようとすると、ファイルの変更によって**衝突**[Conflict]が発生する可能性があります（たとえば、両方が同じ行のコードを変更した）。

衝突の発生

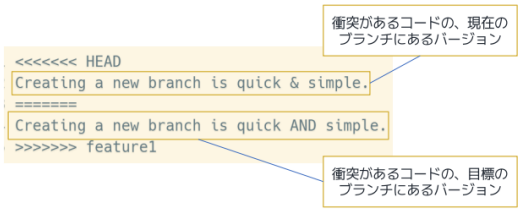
- 衝突の例（Mac）：

```
$ git merge feature1
Auto-merging readme.txt
CONFLICT (content): Merge conflict in readme.txt
Automatic merge failed; fix conflicts and then commit the result.
```

- Git は「readme.txt に衝突があるので、コミットする前に衝突を手動で修正しろ」と指示しました。
- status コマンドで、Git は衝突があるファイルについての詳細を教えてくれます。

衝突の表示

- Git は「<<<<<<<」「=======」「>>>>>>>」のような特殊符号を使って、競合するファイル中の異なるブランチの内容をマークする。これにより、コンフリクトを含むファイルを開いて手動で解決することができます：



衝突の解消

- <<<<<<<、=======、>>>>>>> がマークした内容を手動で以下のように修正して、保存します：

```
Creating a new branch is quick AND simple.
```

- 上記のように**ブランチの 1 つのみ**への変更を保持して、<<<<<<<、=======、>>>>>>> の行も完全に削除しなければなりません。
- すべてのファイルの衝突を解決したら、各ファイルで add コマンドを使用してステージングエリアに追加します。衝突するファイルがステージされると、Git はそれらを解決済みとしてマークします。

Markdown

- Markdown 言語は、2004 年に John Gruber 氏によって作られた、読みやすく、書きやすいプレーンテキスト形式で文書を書くことができる軽量のマークアップ言語であります。
- Markdown で書かれた文書は、HTML、Word 文書、画像、PDF、Epub などのさまざまな形式に変換できます。
- Markdown ファイルの拡張子は「.markdown」 または「.md」になります。
- 書き方については、こちらのリンクを参考してください:

https://github.com/zhongtaoaki/lighthouse-july-java/blob/main/markdown.md

https://qiita.com/tbpgr/items/989c6badefff69377da7