使い始める

Git のインストール

Git の最新版をインストールします。事前にインストール済みの場合はバージョンを確認し、古いバージョンであれば最新にアップデートしましょう。

Windows にインストール

Windows では、Windows で Git を使用する場合と WSL2 で Git を使用する場合があります。

Windows に Git をインストールする

Windows で Git を使用する場合は、Git for Windows (https://gitforwindows.org/) をダウンロードしてインストールします。

WSL に Git をインストールする

WSL2 の Ubuntu で Git を使用する場合は、ターミナルを開いて次のコマンドを実行します。

```
sudo add-apt-repository ppa:git-core/ppa
sudo apt-get update
sudo apt-get install git-all
```

Mac にインストール

Mac では Xcode Command Line Tools をインストールすると、Git もインストールされます。より新しいバージョンをインストールしたい場合は、バイナリからインストールするか Homebrew でインストールします。

Apple Git (Xcode) でインストールする

ターミナルを開き、次のコマンドを入力してください。

```
git version
```

Git がインストール済みであればバージョンが出力されます。エラーになった場合は、XCode Command Line Tools をインストールします。

```
xcode-select --install
```

バイナリでインストールする

ダウンロードサイト（https://git-scm.com/download/mac）からバイナリをダウンロードしてインストールします。

Homebrew でインストールする

ターミナルから次のコマンドでインストールします。

```
brew install git
```

バージョンの確認

次のコマンドでバージョンを確認してください。正しくインストールできていればバージョンが表示されます。

```
git version
```

ヘルプの確認

次のコマンドを実行します。

```
git help
```

特定のコマンドのヘルプを参照したい場合は、次を実行します。

```
git help <command>
```

コマンド補完とプロンプト

Git のコマンドやオプションは数も多く、すべて手打ちで作業を進めるのは非効率です。後述の設定を実施することでタブ補完が効くようになり、また現在のリポジトリの状態を確認できるようになります。なお、Git for Windows はインストール時点でいずれも使えるようになっているため、ここでの作業は必要ありません。

ターミナルから次のコマンドでシェルの種類を確認し、以下の指示に従って設定してください。

```
echo $SHELL
```

共通

Bash/Zsh 環境でプロンプトにリポジトリの状態を表示するようにするスクリプトをダウンロードします。

```
curl -s -o ~/.git-prompt.sh
https://raw.githubusercontent.com/git/git/master/contrib/completion/git-prompt.sh
```

Bash の場合

Bash 環境でタブ補完をできるようにするスクリプトをダウンロードします。

```
curl -s -o ~/.git-completion.bash
https://raw.githubusercontent.com/git/git/master/contrib/completion/git-
completion.bash
```

~/.bash_profile に次の行を追記します。

```
if [ -r ~/.bashrc ]; then
```

```
  . ~/.bashrc
fi
```

次に ~/.bashrc に次の行を追記します。

```
source ~/.git-completion.bash
source ~/.git-prompt.sh
PS1='[¥u@¥h ¥W$(__git_ps1 " (%s)")]¥$ '
```

Zsh の場合

~/.zshrc に次の行を追記します。

```
# タブ補完
autoload -Uz compinit && compinit

# プロンプト変更
source ~/.git-prompt.sh
setopt PROMPT_SUBST ; PS1='[%n@%m %c$(__git_ps1 " (%s)")]¥$ '
```

Git の構成

これから Git を使用するにあたり、先にいくつかの設定を行います。

Git には、git config を呼ばれるツールが付属しています。このツールを使うと、現在の設定情報を参照したり、設定を変更したりすることができます。

Git の設定には三つのレベルがあります。

- システム全体に対する設定
- ユーザに対する設定
- 単一リポジトリに対する設定

同じ設定が複数で変更されている場合は、範囲が狭い方の設定が優先されます。

```
システム設定 ＜ ユーザ設定 ＜ リポジトリ設定
```

ユーザ情報の設定

ユーザ情報はコミットごとに記録され、誰の変更か後で確認できるようになります。この情報はどのリポジトリでも共通になるため、ユーザ用設定に保存します。

```
git config --global user.name "名前"

git config --global user.email mail@example.com
```

--global オプションを指定すると、現在のユーザに対する設定になります。この設定を行うのは一度だけです。

設定の確認

次のコマンドで、現在の設定値を確認できます。

```
git config --list
```

特定の設定を確認したい場合は、git config〈key〉で確認できます。

```
git config user.name
```

デフォルトブランチ

(この手順は必要な場合のみ実施します)

ブランチ名が master になっている場合は、ブランチの名前を main に変更してください。

```
git branch -m main
```

これ以降、作成するリポジトリでデフォルトのブランチ名が main になるように設定します。

```
git config init.defaultBranch main
```

Git 基礎

Git はチームでアプリケーションを開発する上で必須となる、バージョン管理システムです。
ここでは、まずバージョン管理システムについて知り、次に Git の使い方を身に付けます。

バージョン管理システムについて

まずは、バージョン管理システムがどのようなものか見ていきます。どのような課題があり、どういった経緯で Git が誕生したのかを確認します。

次の順に見ていきます。

1. コードのバックアップ
2. ローカルバージョン管理
3. 集中バージョン管理システム
4. 分散バージョン管理システム

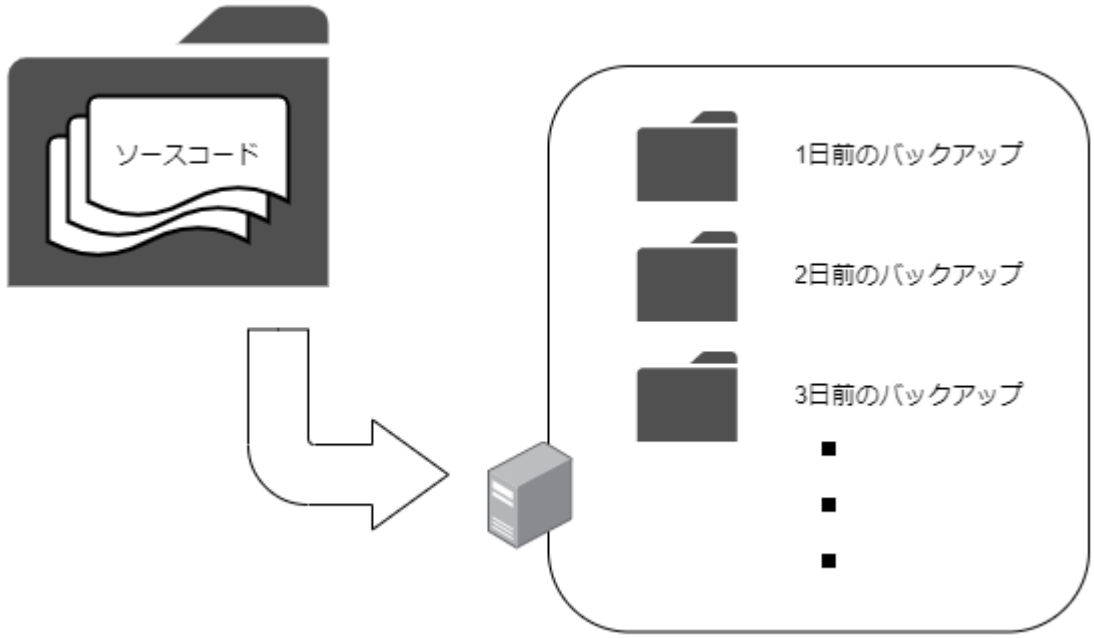
コードのバックアップ

これまで演習等でファイルにプログラムを書いてきましたが、その中で次のようなことを経験したことはありませんか？

- 動いていたプログラムを色々編集していたら動かなくなり、元に戻すこともできなくなった
- 操作を誤ってファイルを削除してしまい、元に戻せなくなった
- 演習の問題に直接答えを書いていたので、問題をもう一度やるには改めてファイルをダウンロードする必要があった

大規模なアプリケーションとなるとファイル数もコード量も多く、少しの変更が広範囲に影響を与えることがあります。動いていたアプリケーションがエラーで動かなくなったりしたときに、すぐに動く状態に戻せないと長時間サービスを停止しなくてはなりません。

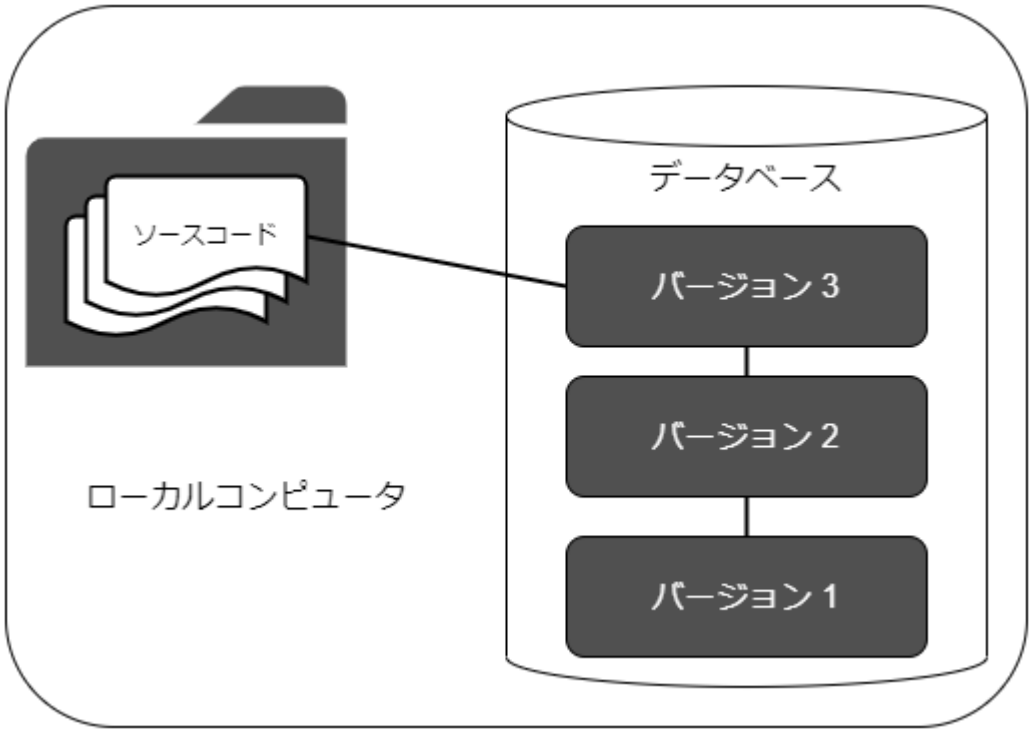
そこで、動かなくなった場合に備えて、コードのバックアップを取り保管するようになりました。アプリケーションのソースコードが含まれるディレクトリを定期的にコピー・圧縮してサーバに保管するなどの方法が取られました。



ローカルバージョン管理

バックアップを取ることで動作する状態に戻すことは可能となりましたが、例えば日次でバックアップを作成しているとすると　1 日分の作業が失われることになります。また、いつどのような変更をしたかはわからず、過去に遡って問題の調査をすることは困難でした。

そこで、ローカルのバージョン管理システムが開発されました。このシステムはファイルのバージョンごとの変更をデータベースに保存します。このシステムによって、いつどのような変更が行われたか追跡できるようになりました。

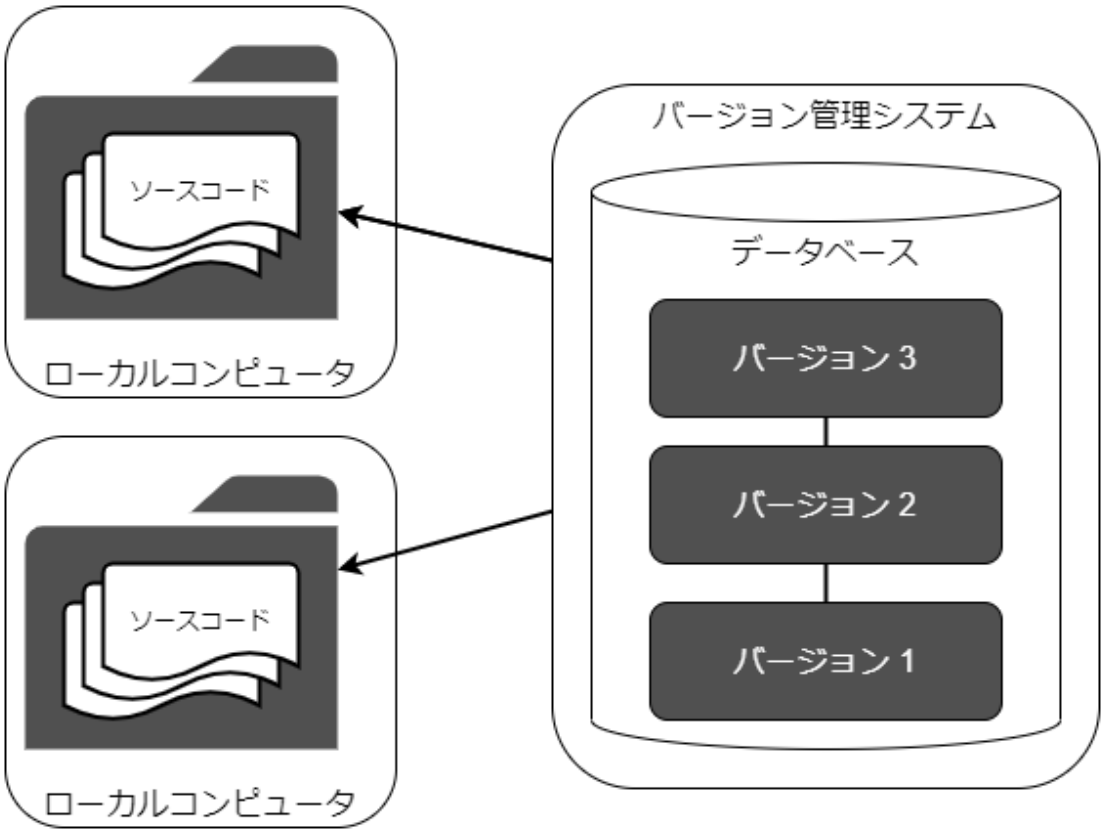


集中バージョン管理システム

ローカルバージョン管理システムによって、履歴を遡れるようになりました。ですが、このシステムはローカル PC 上にあるため、他の開発者と共同作業するには向いていません。

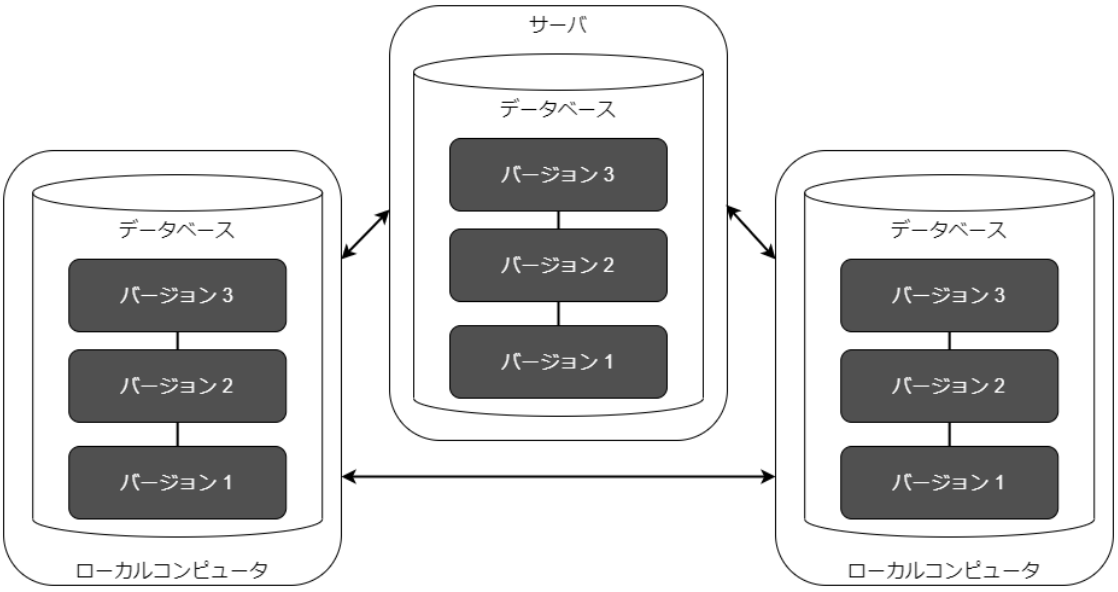
この問題に対処するために、集中バージョン管理システムが開発されました。すべてのファイルをバージョン管理するシステムをサーバ上に構築し、各開発者はそこからファイルをコピー（チェックアウトと呼びます）することで、いつ誰がどのように変更を行ったか辿れるようになりました。

この構成は長く標準として使われていましたが、中央のサーバに障害が発生した場合に開発者全員の作業が止まってしまうと問題を抱えていました。



分散バージョン管理システム

サーバの障害によって生じる問題を回避するために、分散バージョン管理システムが開発されました。開発者は変更するファイルをチェックアウトするのではなく、リポジトリ（アプリケーションのすべてのファイルを格納したディレクトリとその履歴）全体のコピーをローカルに持ちます。すべてのコピーを複数の開発者が持っているため、サーバに障害が発生したとしても、いずれかの開発者のリポジトリをコピーしてすぐにサーバを修復することができます。



この分散バージョン管理システムの一つとして、Git　が誕生しました。

使い始める

インストール

指示に従って、Git をインストールしてください。

Git 構成

コマンドラインで Git を使いやすくするために、最初に設定を行います。

Git の基本

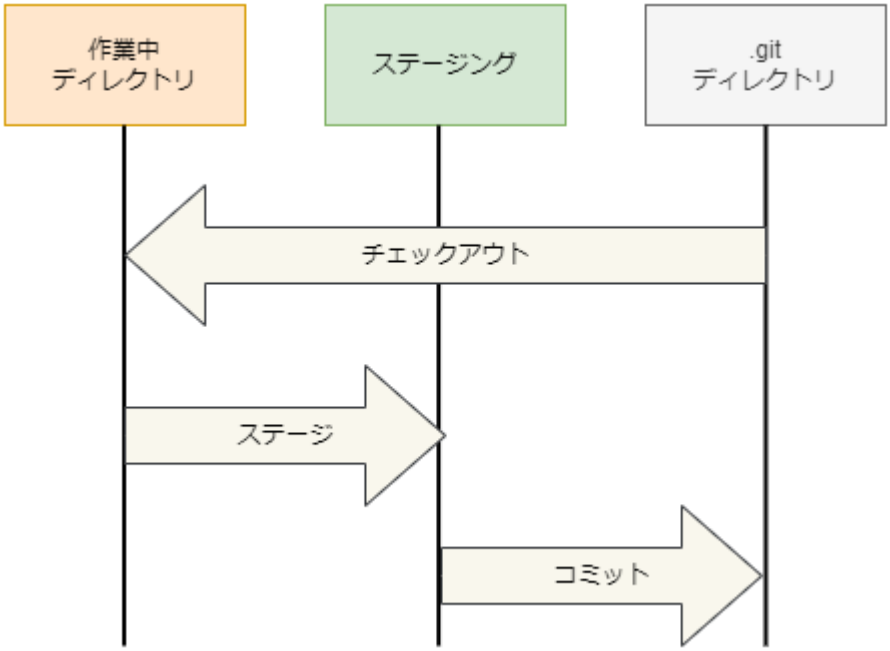
履歴の持ち方

Git は、ある時点におけるリポジトリの状態をスナップショットとして保存します。開発者は、ある時点のリポジトリ全体をいつでも復元可能です。また、履歴をローカルPC 上に保管しているため、ある時点の履歴を取得したり新しい変更を履歴として保存したりする際に、サーバと通信ができなくなっても大丈夫です。

三つの状態

Git で管理されるファイルは、主に三つの状態を持ちます。コミット済、修正済、ステージ済の三つです。

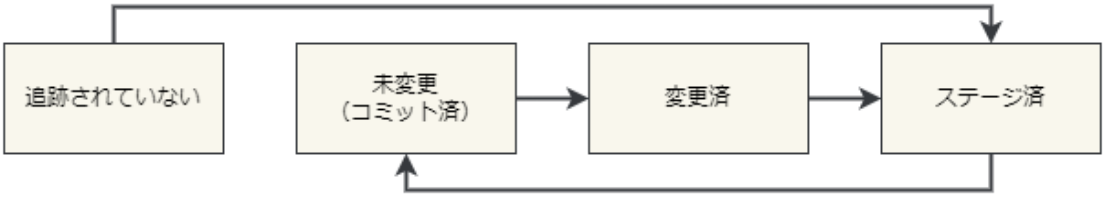
ファイルを変更し保存した状態が修正済です。このとき、データベースに変更は保存されていません。修正済のファイルに、これから履歴として保存するための印がついた状態がステージ済です。ステージ済のファイルの変更を記録すると、コミット済になります。



1. .git ディレクトリはリポジトリルートにあり、プロジェクトの情報と履歴を保存するデータベースがあるところです。
2. .git ディレクトリから一つのバージョンを取得することをチェックアウトと呼びます。チェックアウトされたファイルが作業ディレクトリに配置されます。
3. 作業ディレクトリでの作業後、次の履歴に含まれると印がつけられたものがステージングに保存されます。ステージングにあるものを記録に残すことをコミットと呼びます。

基本的なワークフロー

アプリケーションに必要なすべてのファイルを格納するルートディレクトリをリポジトリと呼びます。リポジトリは開発開始時に初期化された後、修正を加え、変更内容をステージし、コミットするというフローを繰り返し実施します。まず、ローカル PC 上でこのワークフローを確認してみましょう。



ターミナルを開き、リポジトリ用のディレクトリを作成してそのディレクトリに移動してください。

```
cd ~/src
mkdir git-primer
cd git-primer
```

リポジトリの初期化

プロジェクトを Git で管理し始めるときは、そのプロジェクトのディレクトリで次のように打ち込みます。

```
git init
```

実行後、.git という名前のディレクトリが作成され、リポジトリに必要なすべてのファイルがその中に格納されます。

```
ls -la
ls -la .git/
```

変更内容のリポジトリへの記録

新しいファイル first.txt を作成し、何か書き込んで保存してください。

作業ディレクトリ内には、追跡されているファイルと追跡されていないファイルが存在します。「追跡されていない」とは Git によって管理されていないということで、現在 first.txt は追跡されていないファイルです。

この時点でファイルがどの状態にあるかを確認するために使用するのが次のコマンドです。

```
git status
```

次のような結果になるはずです。

```
(省略)
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    first.txt
```

Untracked files には、追跡されていないファイルが出力されます。first.txt が追跡されていないため、これを Git の管理対象に追加してみます。新しいファイルの追跡を開始するには、次のコマンドを使用します。

```
git add first.txt
```

もう一度、状態を確認してみましょう。

```
git status
```

次のような結果になるはずです。

```
(省略)
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:    first.txt
```

"Changes to be committed" とあり、ステージされています。ステージの状態にあるファイルはコミット（記録）できます。

```
git commit
```

前述のコマンドを実行すると、Vim　が起動します。最初の行にコミットメッセージを記述して、保存してください。2　行目以降はコメントが表示されます。

```
最初のコミット
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
(省略)
```

これで、最初のコミットが作られました。

もう一度、first.txt　を編集します。何か追記して保存してください。作業を行ったら状態を確認しましょう。

```
git status
```

今回はこのような出力になるはずです。

```
On branch main
Changes not staged for commit:
  (use "git add <file>..."  to update what will be committed)
  (use "git restore <file>..."  to discard changes in working directory)
    modified:     first.txt
```

変更はコミットのためにステージされていません。先程と同様、次のコマンドでステージします。

```
git add first.txt
```

続けて、コミットしましょう。コミットメッセージを記述して保存してください。

```
git commit
```

これで、二番目のコミットが作られました。

コミット履歴の閲覧

作成されたコミットの履歴を確認するには、次のコマンドを使用します。

```
git log
```

次のような結果が出力されます。（実際の出力は手元で確認してください）

```
commit xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx（HEAD -> main）
Author: <ユーザ名> <メールアドレス>
Date:    <日時>

    二番目のコミット
commit xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Author: <ユーザ名> <メールアドレス>
Date:    <日時>

    最初のコミット
```

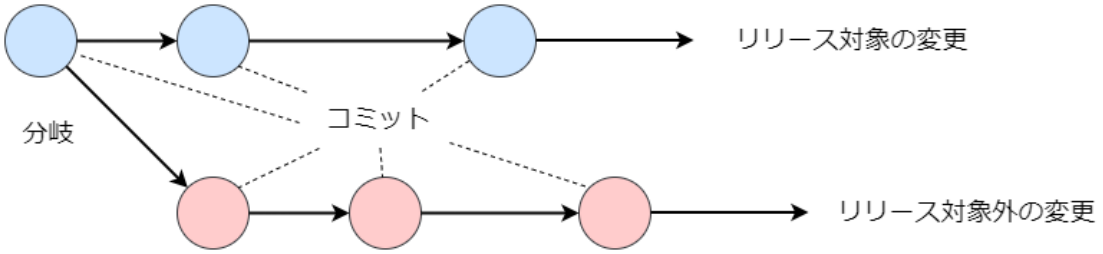
直近のコミットが上に表示されます。

ここまでの内容を、演習で振り返りましょう。演習の　1 から 6 を実施して下さい。

ブランチ機能

ブランチとは

チーム開発では、複数の機能が並行して実装されることが少なくありません。ですが、ある機能をリリースするときに、他の開発中のコードは含めたくない場合があります。このような場合、リリース対象の開発から分岐して、互いに干渉しないように作業を続ける機能をブランチ機能と呼びます。



ブランチの作成

新しいブランチを作成するには、次のコマンドを使用します。

```
git branch testing
```

これで新しいブランチが作成されました。次のコマンドで確認できます。

```
git branch
```

現在のブランチが ＊ で示されているはずです。

ブランチの切り替え

ブランチを切り替えるには、次のコマンドを使用します。

```
git checkout testing
```

現在のブランチが変わっているか確認してみましょう。

```
git branch
```

testing の前に * が表示されたはずです。次のコマンドでログを確認しましょう。

```
git log --all --decorate --graph
```

main と testing は同じコミットを指しています。

ブランチの分岐

ここで、first.txt を編集して、コミットまで実施します。その後、次のコマンドでログを確認しましょう。

```
git log --all --decorate --graph
```

main と testing は別のコミットになりました。

ブランチを切り替えます。

```
git checkout main
```

さらに first.txt を編集し、コミットしてみます。ログを確認するとどうなるでしょうか？

```
git log --all --decorate --graph
```

ブランチが分岐している様子が見えます。

このように、ブランチを行ったり来たりしながら並行して作業することが可能です。

分岐したブランチは、最終的にマージされます。今、各ブランチで同じファイルに別々の変更を行いました。これを main ブランチに統合します。main ブランチにいる状態で次のコマンドを実行します。

```
git merge testing
```

次のような結果が出力されます。

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

これは、互いのブランチでの変更が競合しているため、自動でマージできないというメッセージです。競合が解決されないと処理を進めることができません。first.txt を開くと、このような部分が含まれています。

```
<<<<<<< HEAD
mainでの変更
=======
testingでの変更
>>>>>>> testing
```

HEAD が今いるブランチです。======= より上が今いるブランチの変更、下が testing ブランチでの変更です。どちらを残すか、または両方残すのかといった判断は開発者が行います。最終的に、<<<<<、 =====、 >>>>> の行はすべて削除します。

編集が完了したら、次を実行してマージを解決済みにします。

```
git add first.txt
```

```
git commit
```

これで、testing ブランチの変更は main ブランチにマージされました。この時点で testing ブランチは不要になったので、次のコマンドでブランチを削除します。

```
git branch -d testing
```

次のコマンドでブランチを確認しましょう。

```
git branch
```

main のみになっているはずです。

ここまでの内容を演習で振り返りましょう。演習の 7 から 13 を実施してください。

※一日目はここまでです。お疲れ様でした。

GitHub

今後の研修では、Git のホスティングサービスである GitHub を利用します。指示に従ってアカウントの作成とセットアップを行ってください。

リモートリポジトリを使った作業

リモートリポジトリのクローン

リモートしかリポジトリが存在しない場合は、まず、ローカルにクローンします。

```
git clone git@github.com:<user>/<repository>.git
```

リモートブランチの確認

次のコマンドで、リモートブランチを確認できます。

```
git branch -r
```

このような表示になります。origin は、clone したときに Git が自動で設定したリモートリポジトリの名前です。

```
  origin/HEAD -> origin/main
  origin/main
```

ローカルでの作業

復習を兼ねて、以下の操作をローカルで実施してください。

1. main ブランチから新しいブランチを作成する

   ブランチ名は user/ユーザ名 とする

2. README.md に、自分のユーザ名を追記する

3. コミットする

リモートブランチへのプッシュ

ブランチの内容を他の開発者と共有するため、各自のローカルでの変更をリモートに反映します。この作業をリモートへのプッシュと呼びます。

プッシュは自動的に行われることはありません。共有したいブランチは明示的にプッシュする必要があります。

コマンドは git push リモートリポジトリ名 ブランチ名 を使います。

リモートリポジトリ名は origin ですので、次のコマンドでプッシュします。

```
git push -u origin user/ユーザ名
```

-u は、プッシュしたときに Git がローカルのブランチとリモートのブランチを紐付ける情報を作成するためのオプションです。初めてプッシュするブランチの場合だけ指定します。

リモートブランチの取得

他の開発者がプッシュしたブランチを、ローカルリポジトリに取り込みます。

まず git fetch リモートリポジトリ名 で、まだローカルリポジトリに存在しない情報を取り込みます。このとき、作業ディレクトリのファイルの変更は行われず、更新されるのは .git/ のファイルだけです。

```
git fetch origin
```

.git/ のデータが更新されたあとは、ローカルでブランチを切り替えたり、自分のブランチに新しいブランチの変更内容をマージしたりすることができます。

```
git merge --ff-only
```

ブランチを確認します。-a はローカルとリモート両方のブランチを表示します。

```
git branch -a
```

リモートブランチ名から リモートリポジトリ名/ を取り除いたブランチ名を使ってブランチを切り替えることができます。

```
git checkout ブランチ名
```

リモートのブランチとローカルのブランチが対応している場合、git fetch + git merge を実行する代わりに、git pull を実行するだけでリモートからローカルに作業ディレクトリの変更を取り込む事ができます。

リモートの詳細情報確認

ここで、リモートリポジトリの情報を確認してみます。次のコマンドを実行してください。

```
git remote show origin
```

こちらは、main ブランチしかない状態のリモートの情報ですが、これより多くの情報が出力されているはずです。

```
* remote origin
  Fetch URL: git@github.com:rpentry202207f/git-getting-started.git
  Push  URL: git@github.com:rpentry202207f/git-getting-started.git
  HEAD branch: main
```

```
Remote branch:
   main tracked
Local branch configured for 'git pull':
   main merges with remote main
Local ref configured for 'git push':
   main pushes to main (up to date)
```

下記の内容を見ると、git pull したときに自動的にマージされるブランチと git push したときにどのリモートブランチにプッシュされるかを確認することができます。

```
Local branch configured for 'git pull':
Local ref configured for 'git push':
```

リモートブランチの削除

マージされたことで不要になったリモートブランチは、git push に --delete オプションを指定して削除します。

```
git push origin --delete ブランチ名
```

※二日目はここまでです。お疲れ様でした。

付録

作業のやり直し

開発時は、様々な場面で Git の操作をやり直したいと考えることがあります。以下にいくつか例をあげます。

- コミットに含める変更を add するのを忘れていたため、コミットをやり直したい
- 誤って作成したファイルを add してしまったが取り消したい
- いくつか前のコミットはリリースしないことになったのでなかったことにしたい

いずれの場合も、操作をやり直すことは可能です。ただし、リモートに push 済みのコミットを書き換えると他のメンバーに迷惑がかかることがあるため、push 済みの内容は変更しないように気をつけましょう。

stage された状態

git add された状態から編集中に戻します。

```
git restore --staged <file>...
```

commit された状態

コミットメッセージを変更したり、ファイルのコミット漏れを追加したりするなどの目的で、直近のコミットをやり直します。

```
git commit --amend
```

（コミット漏れがあれば先に add する）

コミットを取り消す

コミットを取り消すコミットを作ることで、そのコミット前の状態に戻すことができます。

```
git revert <取り消したいコミット>
```

コミットをなかったことにする

歴史の改変と呼ばれる方法です。履歴から対象のコミットをなかったことにします。

直近のコミットを消す場合

```
git reset HEAD^
```

いくつか前のコミットを消す場合

```
git rebase -i <対象コミットの一つ前のコミット>
```

作業中の状態をなかったことにする

stage されていない状態から working tree clean な状態にします。

```
git checkout <file>
```

（実行すると作業中の内容は失われるため、注意が必要です）

その他補足

エディタ

git のデフォルトのエディタは Vim や nano になっていますが、これは変更できます。デフォルトエディタを変更すると、コミットの記述やマージで使用するエディタが変わります。設定 core.editor で変更できます。

```
git config --global core.editor 'code --wait'
```

エイリアス

コマンドにエイリアス（別名）をつけて、短縮することができます。

例えば、次のログコマンドは長いので、

```
git log --all --decorate --graph
```

lg と短い名前をつけます。

```
git config --global alias.lg 'log --all --decorate --graph'
```

次のように呼び出せるようになります。

```
git lg
```

ブランチの切り替えと作成

次のコマンドで、新しいブランチを作成すると同時にそのブランチに切り替えることができます。

```
git checkout -b new-branch
git switch -c new-branch
```

switch コマンドはブランチを切り替えるコマンドですが、バージョン 2.38.0 の時点では「実験的なコマンドで挙動が変更される可能性がある」とされています。

競合の解決を指定したエディタで実行する

マージツールを指定すると、そのツールでコンフリクトの解決を行えます。マージの際、次のコマンドを実行します。

```
git mergetool
```

最初にどのツールを使用するか確認されるので、そこで使用したいツールを指定してください。

無視リスト

リポジトリで管理したくないファイルがあります。そういったファイルは予め Git でバージョン管理されないように除外することができます。.gitignore ファイルを作成し、除外するパターンを指定します。

除外リストは、OS や言語などによってよく利用されるものがまとまっているので、参考にするとよいでしょう。

github/gitignore（https://github.com/github/gitignore）

一時的に作業中のファイルを退避する

git stash を使うと、一時的に作業ディレクトリをクリーンな状態にできます。このようにすることで、リモートの変更を取り込んだあとで作業に戻るといったことが可能になります。

```
git stash

# 最後に statsh した作業を戻すとき
git stash pop

# stash されているリストの確認
git stash list

# 特定の stash を適用するとき
git stash apply[1]
```

Git 演習

ローカルリポジトリの操作

1

新しいリポジトリ git-practice を作成してください。作成したら、リポジトリを初期化してください。

2

新規ファイル README.md を作成してください。次の内容を README.md に記載して保存してください。

```
# Git 演習
```

保存後に、リポジトリの状態を確認してください。

3

2 で作成したファイルを Git の管理に追加し、その後リポジトリの状態を確認してください。

4

3 でステージに追加されたファイルをコミットしてください。コミット後にリポジトリの状態を確認してください。

5

README.md に、ここまで使用したコマンドをすべて記載してください。その後、コミットまで実施してください。

6

コミット履歴を確認してください。

7

新しいブランチ feature/branching を作成してください。作成されたか確認してください。

8

新たに作成したブランチに切り替えてください。切り替わったか確認してください。

9

README.md を編集して、コミットしてください。コミット後にリポジトリの状態とログを確認してください。

10

ブランチを main に切り替えてください。

11

README.md を編集し、コミットしてください。

12

feature/branching を main にマージしてください。

13

feature/branching を削除してください