Git と GitHub 初心者向け図解ガイド：最初のコマンドから共同作業まで

Taro Meidai

2025 年 8 月 27 日

目次

はじめに：あなたの旅がここから始まる

プログラミングやウェブ制作の世界へようこそ。これから学ぶ多くのツールの中でも、Git（ギット）とGitHub（ギットハブ）は、現代の開発者にとって欠かせない存在です。しかし、多くの専門用語や黒い画面でのコマンド操作に、最初は戸惑いを感じるかもしれません。ご安心ください。この記事は、そんなあなたのためのガイドです。この旅では、複雑に見える概念を身近な比喩で解きほぐし、一つひとつのコマンド操作を丁寧に図解しながら、あなたのペースで進めていきます。

まずは、この先で何度も登場する重要な言葉たちを、簡単な「翻訳機」を通して見てみましょう。これらの比喩が、あなたの理解を助ける羅針盤となるはずです。

Git 専門用語 翻訳機

専門用語	簡単な比喩	簡単な説明
バージョン管理 (Version Control)	プロジェクトのタイムマシン	ファイルの変更履歴を記録し、いつでも過去の状態に戻せる仕組み。
リポジトリ (Repository)	プロジェクト専用の保管庫	プロジェクトのファイルと、その全履歴が保存されている場所。
コミット (Commit)	セーブポイント	ファイルの特定の状態を「写真に撮って」履歴に保存する操作。
ブランチ (Branch)	パラレルワールド	プロジェクトの歴史を分岐させ、安全に新しい作業を進めるための仕組み。
マージ (Merge)	パラレルワールドの統合	分岐したブランチでの作業を、メインの歴史に合流させること。
リモート (Remote)	インターネット上の保管庫	GitHub など、ネットワーク上にあるリポジトリのこと。
プッシュ (Push)	ローカルからリモートへアップロード	自分の PC で行った変更を、インターネット上のリポジトリに反映させること。
プル (Pull)	リモートからローカルへダウンロード	インターネット上のリポジトリの最新の変更を、自分の PC に持ってくること。

1　あなたが知らなかったタイムマシン（なぜバージョン管理が必要なのか？）

レポートや企画書を作成している時、こんな経験はありませんか？ report_final.doc、report_final_v2.doc、report_final_REALLY_final.doc...。ファイル名がどんどん増えていき、どれが本当に最新版なのか分からなくなる。これは、多くの人が経験する「バージョン管理」の悩みです。

1.1 システムがないと直面する問題

システムを使わずに手作業でファイルのバージョンを管理しようとすると、いくつかの問題が必ず発生します。

- 何かを壊すことへの恐怖:「この一行を変えたら、全部動かなくなるかも…」という不安から、変更を加えるたびにファイルのコピーを延々と作り続けることになります。これは非効率的で、ディスク容量も圧迫します。
- 過去の作業の喪失: 誤ってファイルを上書きしてしまい、以前の良かったバージョンを永遠に失ってしまうことがあります。「元に戻す」機能では救えない、致命的なミスにつながる可能性があります。
- 共同作業のカオス: 複数人で作業する場合、メールでファイルを送り合うのは悪夢の始まりです。誰かが加えた変更を自分のファイルに手作業で取り込み、気づかぬうちに他の人の修正を上書きしてしまうことも珍しくありません。「この変更は誰が、いつ、なぜ行ったのか？」という問いに答える術がなくなります。

1.2 解決策：セーフティネットとしてのバージョン管理

Git のようなバージョン管理システム（Version Control System, VCS）は、これらの問題を解決するための強力なセーフティネットです。

- 完全な履歴の記録:VCS は、ファイルに加えられた全ての変更を記録します。誰が、いつ、どのような目的で変更したのか、そのすべてが記録として残ります。これは、プロジェクトの信頼できる航海日誌のようなものです。
- 自在なタイムトラベル: プロジェクトの歴史の中の、どの時点にでも簡単かつ正確に戻ることができます。これは単なる「元に戻す」操作ではありません。歴史上の任意の「セーブポイント」へ瞬時にジャンプするタイムトラベルです。この機能があるからこそ、開発者は失敗を恐れずに新しいアイデアを試すことができるのです。
- 構造化された共同作業: VCS は、複数人が同じプロジェクトで効率的に作業するための明確なルールと仕組みを提供します。他の人の作業を上書きしてしまう心配なく、それぞれの変更を安全に統合できます。

バージョン管理の最も重要な価値は、技術的な機能そのものよりも、それがもたらす「心理的安全性」にあります。それは「どれだけ大胆な変更を加えても、いつでも元に戻せる」という安心感です。この安心感が、学習と創造性を加速させます。

2 Git と GitHub の関係（言語と国のようなもの）

Git と GitHub はよく混同されますが、この 2 つは全く別のものです。

- Git: プロジェクトの歴史を記録・管理するための「ソフトウェア」（コマンドラインツール）。あなたの PC 上で動作します。
- GitHub: Git で管理されているプロジェクトを、インターネット上で保管・共有するための「ウェブサービス」。

なぜ両方が必要なのか？ Git だけでも、個人のプロジェクトを管理することは可能です。しかし、GitHub（や GitLab、Bitbucket といった類似のサービス）を組み合わせることで、Git の真価が発揮されます。GitHub

は、あなたのプロジェクトのバックアップ場所として機能し、他の開発者とコードを共有し、チームでの作業を円滑に進めるためのプラットフォームを提供します。Git というスキル（言語）を身につければ、GitHub だけでなく、GitLab など他の「国」でも活躍できるのです。

3　開発者のツールキットを揃えよう

ここからは、実際に手を動かして、あなたの開発環境を整えていきましょう。職人が仕事道具を揃えるように、この一度きりのセットアップが、これからのすべての作業の基礎となります。

3.1　パート 1：Git のインストール（道具を手に入れる）

3.1.1　Windows の場合

1. インストーラーのダウンロード: まずは Git の公式サイト（https://git-scm.com）にアクセスし、Windows 用のインストーラーをダウンロードします。
2. インストールの実行: ダウンロードした .exe ファイルを実行します。インストールウィザードが起動しますが、たくさんの選択肢が表示されても、基本的にすべての画面でデフォルト設定のまま「Next」をクリックし続けて問題ありません。
3. インストールの確認: インストールが完了したら、スタートメニューから「Git Bash」を起動するか、PowerShell またはコマンドプロンプトを開きます。そこで以下のコマンドを入力してください。

```
git --version
```

‘git version 2.XX.X‘ のようにバージョン情報が表示されれば、インストールは成功です。

3.1.2　macOS の場合

1. インストールの確認: まず、ターミナルアプリを開き、‘git –version‘ を試し、Git がすでにインストールされていないか確認します。
2. Homebrew のインストール（推奨）: Homebrew は、macOS 用のパッケージ管理システムです。まだインストールしていない場合は、公式サイト（https://brew.sh）に記載されているコマンドを実行してインストールしてください。
3. Git のインストール: Homebrew がインストールできたら、ターミナルで以下のコマンドを実行するだけで Git がインストールされます。

```
brew install git
```

この方法の利点は、将来的に ‘brew upgrade git‘ という簡単なコマンドで Git を最新の状態に保てることです。

3.2　パート 2：GitHub アカウントの作成（あなたのオンライン上の身分証明）

1. GitHub 公式サイトへアクセス: ブラウザで GitHub の公式サイト（https://github.com）を開き、画面右上の「Sign up」ボタンをクリックします。
2. 情報の入力: 画面の指示に従い、メールアドレス、パスワード、ユーザー名を入力します。ユーザー名は、あなたの公開される ID となるため、慎重に選びましょう。
3. メール認証: 登録したメールアドレスに GitHub から認証コードが記載されたメールが届きます。そのコードを画面に入力して、認証を完了させます。

3.3 パート 3：Git への自己紹介（初回設定）

インストールした Git にあなたの情報を教える必要があります。これは、これからあなたが行うすべての「セーブ」（コミット）に、作成者としてあなたの名前とメールアドレスが記録されるため、非常に重要です。ターミナル（Windows の場合は Git Bash）を開き、以下の 2 つのコマンドを一行ずつ実行してください。‘”Your Name”‘と ‘”your.email@example.com”‘の部分は、ご自身の名前に置き換えてください。メールアドレスは GitHub に登録したものと同じものを使うのが一般的です。

```
git config --global user.name "Your Name"
git config --global user.email "your.email@example.com"
```

‘–global‘ というオプションは、この設定をこのコンピュータ上のすべての Git プロジェクトで共通して使用するという意味です。設定が正しく行われたかを確認するには、以下のコマンドを実行します。

```
git config --list
```

一覧の中に ‘user.name‘ と ‘user.email‘ が正しく表示されていれば、準備は完了です。

4 最初のソロプロジェクト - 進捗をローカルに保存する

ツールの準備が整いました。いよいよ、Git を使って最初のプロジェクトを管理してみましょう。ここでは、あなたのコンピュータの中だけで完結する、基本的な流れを学びます。

4.1 Git の 3 つのエリア（比喩で理解する）

Git の操作を理解するためには、まず 3 つの主要な「場所」の概念を掴むことが重要です。これを、机仕事に例えてみましょう。

- ワーキングディレクトリ (Working Directory) - あなたの「仕事机」: あなたが実際にファイルを作成したり、編集したりする場所です。
- ステージングエリア (Staging Area / Index) -「提出用の下書きフォルダ」: 次の「セーブポイント」（コミット）に含めたい変更だけを一時的に置いておくための場所です。
- ローカルリポジトリ (Local Repository) -「鍵付きの書類保管庫」: ステージングエリアにある変更内容を、メッセージ付きで永久に記録・保管する場所です。プロジェクトフォルダ内に作られる ‘.git‘ という隠しフォルダがその実体です。

4.2 コアワークフローの実践

それでは、簡単なウェブサイトのプロジェクトを例に、実際のコマンド操作を見ていきましょう。

4.2.1 git init (保管庫の準備)

まず、プロジェクト用のフォルダを作成し、そのフォルダに移動します。そして、‘git init‘ コマンドを実行します。これにより、‘.git‘ フォルダ（書類保管庫）が作成され、バージョン管理が開始されます。

```
mkdir my-first-website
cd my-first-website
git init
```

4.2.2 git status (状況確認)

このコマンドは、現在の「仕事机」と「下書きフォルダ」の状態を教えてくれます。どのファイルが変更され、何がセーブの準備ができているのかを確認できます。

```
# （index.html というファイルを作成した後で）
git status
```

4.2.3 git add (下書きフォルダへ移動)

ファイルに変更を加えたら、それを次のセーブポイントに含めるためにステージングエリアに追加します。

```
# 特定のファイルを追加
git add index.html

# フォルダ内の全ての変更を追加（よく使う）
git add .
```

4.2.4 git commit (セーブポイントの作成)

ステージングエリアにセーブしたい変更をすべて追加したら、コミットです。‘git commit‘ は、分かりやすいラベル（コミットメッセージ）を付けて「書類保管庫」に永久保存する操作です。

```
git commit -m "最初のホームページ HTML を追加"
```

この ‘add‘ -> ‘commit‘ というサイクルが、Git を使った開発の基本的なリズムです。

5 傑作を共有する - GitHub との接続

これまでの作業は、すべてあなたのコンピュータの中だけで完結していました。次に、このプロジェクトを GitHub に預けて、バックアップや共有ができるようにしましょう。

5.1 シナリオ A：ローカルで始めたプロジェクトを GitHub に公開する

1. GitHub 上でリモートリポジトリを作成する: GitHub にログインし、「New repository」を作成します。リポジトリ名を入力し、「Public」または「Private」を選択して作成します。
2. ローカルとリモートを紐付ける: ローカルリポジトリに、今作ったリモートリポジトリの場所を教えます。GitHub のリポジトリページに表示されている URL をコピーし、ターミナルで以下のコマンドを実行します。

   ```
   git remote add origin https://github.com/あなたのユーザー名/リポジトリ名.git
   ```

   ‘origin‘ はリモートリポジトリに付ける標準的な「あだ名」です。
3. 最初のプッシュ: ローカルリポジトリのコミット履歴を、GitHub にアップロード（プッシュ）します。

   ```
   git push -u origin main
   ```

   ‘-u‘ は初回のみ必要な設定で、ローカルの ‘main‘ ブランチとリモートの ‘main‘ ブランチを関連付けます。次回からは ‘git push‘ だけで済みます。

5.2 シナリオ B：GitHub に既にあるプロジェクトに参加する

5.2.1 git clone (クローン)

リモートリポジトリの完全なコピー（歴史もすべて含む）を、あなたの PC に一度だけダウンロードしてくるコマンドです。

```
git clone https://github.com/他の人のユーザー名/プロジェクト名.git
```

5.2.2 git pull (プル)

クローンした後、他の人がリモートリポジトリにプッシュした最新の変更を、あなたのローカルリポジトリに取り込むために ‘git pull‘ を使います。

```
git pull origin main
```

6 パラレルワールドの力 - ブランチ入門

現在の安定したバージョンに影響を与えることなく、安全に新しい機能を試すための場所が「ブランチ」です。

6.1 ブランチの比喩：パラレルワールド

ブランチを作成するとは、プロジェクトの歴史から分岐した「パラレルワールド」を作り出すようなものです。

- main ブランチ: プロジェクトの公式な歴史が記録されている「メインの世界線」。
- フィーチャーブランチ: 新機能開発やバグ修正のために作る「パラレルワールド」。何を試してもメインの世界線には影響がありません。

6.2 ブランチ操作コマンド

6.2.1 ブランチの作成と切り替え

ブランチを新しく作り、そのブランチに移動（チェックアウト）するには、以下のコマンドが最も一般的です。‘-c‘ は”create”（作成）を意味します。

```
git switch -c login-feature
```

6.2.2 ブランチの一覧表示

現在存在するブランチの一覧を表示し、今自分がどのブランチにいるか（‘*‘ が付いている）を確認します。

```
git branch
```

6.2.3 既存ブランチへの切り替え

‘main‘ ブランチに戻るなど、すでに存在するブランチに移動します。

```
git switch main
```

6.3 git switch vs git checkout

古いチュートリアルでは ‘git checkout‘ が使われていますが、これは多機能で分かりにくいことがあるため、ブランチ操作専用の ‘git switch‘ を使うことをお勧めします。

操作	モダンなコマンド (git switch
既存ブランチへ切り替え	‘git switch feature‘
新規ブランチを作成して切り替え	‘git switch -c new-feature‘
ファイルの復元	(‘git restore <file>‘ を使用)

7 チームワークが夢を叶える - プルリクエストの流れ

チームでは、フィーチャーブランチの変更を ‘main‘ に統合する前に、他のメンバーにレビューしてもらうのが一般的です。この仕組みが GitHub の「プルリクエスト（Pull Request, PR）」です。

7.1 プルリクエストの物語

開発者「Alex」が、新しい問い合わせフォームを追加するタスクを担当する、という物語を通して流れを見ていきましょう。

1. 最新の状態から始める:Alex は作業前に、リモートの ‘main‘ ブランチの最新の状態をローカルに取り込みます。

   ```
   git switch main
   git pull origin main
   ```

2. フィーチャーブランチを作成する: 自分のタスク用のブランチを作成します。

```
git switch -c alex-add-contact-form
```

3. 作業を行う: このブランチで実装を行い、‘git add‘ と ‘git commit‘ を繰り返します。
4. ブランチをプッシュする: 作業が完了したら、フィーチャーブランチを GitHub にプッシュします。

```
git push origin alex-add-contact-form
```

5. プルリクエストを開く: GitHub のプロジェクトページで「Compare & pull request」ボタンを押し、変更のタイトルと説明を記述してプルリクエストを作成します。
6. レビュープロセス: 他の開発者「Ben」がコードの変更点を確認し、コメントを残します。
7. フィードバックを反映し、更新する: Alex はフィードバックを元にコードを修正し、再度 ‘git push‘ します。新しいコミットは自動的にプルリクエストに追加されます。
8. マージ: Ben が変更を「Approve」（承認）すると、管理者が「Merge pull request」ボタンをクリックし、変更が ‘main‘ ブランチに正式に取り込まれます。
9. 後片付け: マージが完了したら、役目を終えたフィーチャーブランチは削除するのが一般的です。

8 時間軸が交わる時 - マージコンフリクトの解決

チームで開発していると、いつか必ず「マージコンフリクト（衝突）」に遭遇します。これは、Git が「同じファイルの同じ行を別々の方法で変更したため、どちらを正解とすれば良いか自動で判断できません。人間が解決してください」と助けを求めているサインです。

8.1 コンフリクトの具体例

‘style.css‘ というファイルで、Alex が文字色を ‘blue‘ に、Ben が ‘red‘ に変更した場合、マージ時にコンフリクトが発生します。

8.2 解決プロセス

1. コンフリクトの特定: ‘git status‘ を実行すると、どのファイルがコンフリクトしているか（‘both modified‘ と表示される）を確認できます。
2. ファイルを開く: コンフリクトしたファイルをエディタで開くと、Git が以下のような特別なマーカーを自動で挿入しています。

```
  color: blue;
```

‘«««< HEAD‘ から ‘=======‘ までが現在のブランチの変更、‘=======‘ から ‘»»»>‘ までが相手のブランチの変更です。
3. 意思決定を行う: 開発者であるあなたが、このブロックを編集して最終的なコードを決定します。例えば ‘color: red;‘ を採用する場合、マーカーをすべて削除し、その行だけを残します。
4. 解決を確定する: ファイルの編集が終わったら、Git に解決したことを伝えます。

```
# 解決したファイルをステージングする
git add style.css


# 解決を記録するためのコミットを行う
git commit
```

9　プロジェクトを整理整頓する - 必須のツールとテクニック

9.1　パート 1：不要なファイルを無視する .gitignore

プログラムのログや個人設定など、バージョン管理に含めるべきでないファイルは ‘.gitignore‘ ファイルで無視するよう設定します。これはプロジェクトのルートディレクトリに配置するテキストファイルです。

```
# OS が自動生成するファイルを無視
.DS_Store
Thumbs.db

# ログファイルを無視
*.log

# 依存パッケージのフォルダを無視（npm の場合）
node_modules/

# 環境変数ファイル（秘密情報を含むことが多い）を無視
.env
```

‘*‘ はワイルドカード、行末の ‘/‘ はディレクトリ全体を意味します。‘.gitignore‘ は、まだ追跡されていないファイルにのみ有効です。

9.2　パート 2：一歩進んだ歴史の整理 - merge vs. rebase

ブランチの変更を統合する方法には ‘merge‘ と ‘rebase‘ があります。

- git merge (ありのままの歴史): 2 つのブランチの歴史をそのまま残し、統合のための新しい「マージコミット」を作成します。歴史は分岐と合流を繰り返す形になります。
- git rebase (歴史の書き換え): あなたのブランチの変更を、統合先ブランチの最新コミットの先端に付け直します。コミット履歴が一本の直線になり、きれいに見えます。

リベースの黄金律: 他のメンバーと共有しているブランチでは、決して ‘rebase‘ を使わないこと。歴史を書き換えるため、チームのリポジトリに深刻な混乱を引き起こす可能性があります。初心者の方は、常に安全な ‘merge‘ を使うようにしてください。

付録 A　コマンド早見表

コマンド	何をするか
‘git config –global user.name ”Your Name”‘	すべてのリポジトリで使うあなたのユーザー情報を設定する。
‘git init‘	現在のフォルダに新しいローカルリポジトリを作成する。
‘git clone [url]‘	リポジトリとその全歴史をダウンロードする。
‘git status‘	ワーキングディレクトリとステージングエリアの状態を表示する。
‘git add [file]‘	ファイルの変更をステージングエリアに追加する (‘.‘ ですべて)。
‘git commit -m ”[msg]”‘	ステージされた変更をリポジトリの歴史に保存する。
‘git push‘	ローカルのコミットをリモートリポジトリにアップロードする。
‘git pull‘	リモートの最新の変更をダウンロードしてマージする。
‘git switch -c [branch]‘	新しいブランチを作成し、そのブランチに切り替える。
‘git switch [branch]‘	既存のブランチに切り替える。
‘git branch‘	ローカルのブランチを一覧表示する。
‘git merge [branch]‘	指定したブランチを現在のブランチにマージする。