Gitことはじめ

バージョン管理の基本とGitHubの使い方

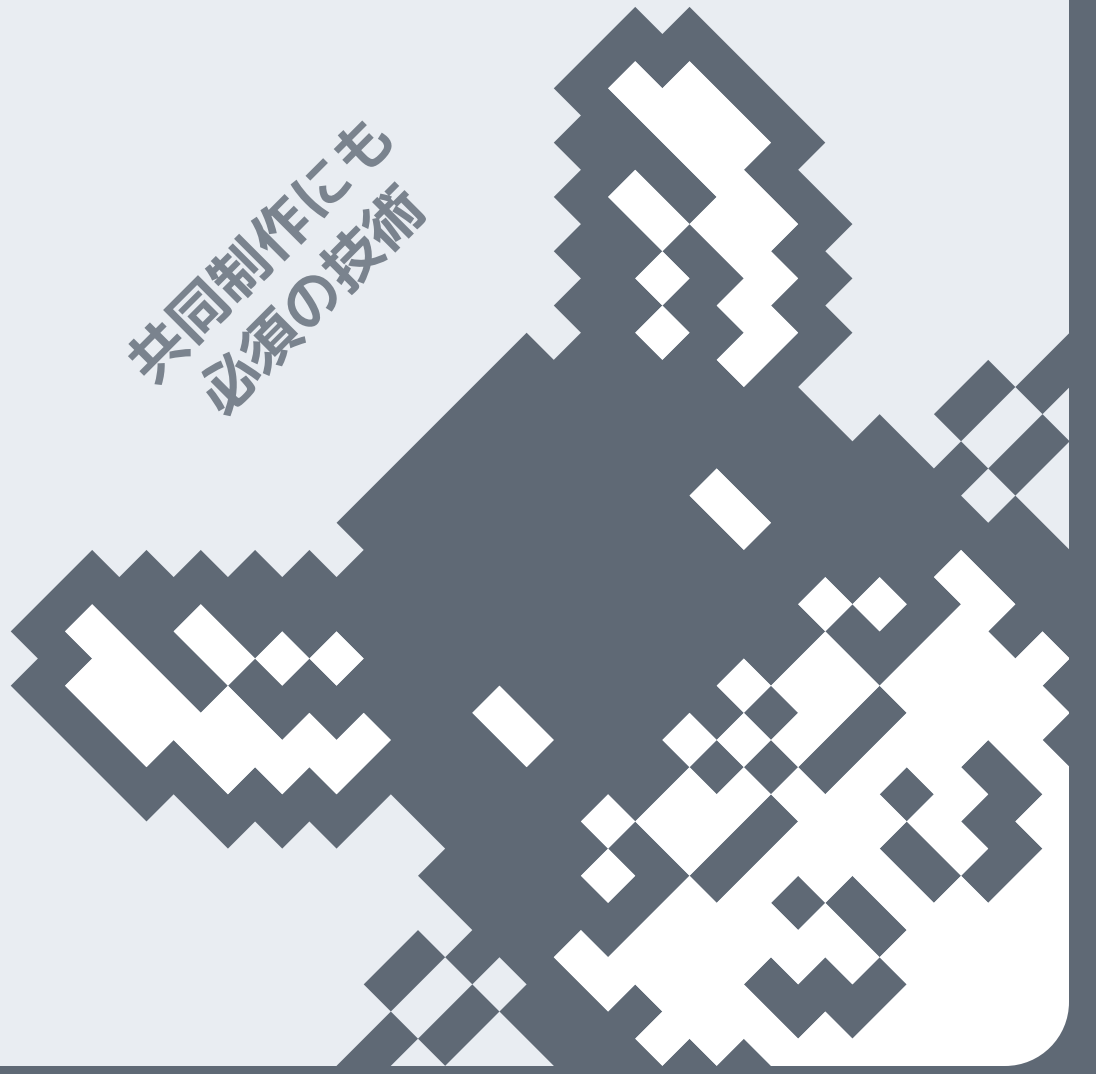
2026年度版

STUDIO-UNSLOW

2026/05/05 更新

はじめに

バージョン管理の仕組みを知ろう



この資料について

Gitの基本とゲーム開発での運用上の注意点を説明します

- Git と GitHub の基本的な考え方を説明します
- ゲーム開発で Git を運用するときの注意点を扱います
- 特に Git でのUnity プロジェクト管理にはいくつか
  注意すべき点がありますので、それらについても説明します

バージョンを管理するとは

なぜバージョン管理が必要か

- ゲーム制作では、ファイルを保存できるだけでは足りません
  誰が・いつ・何を変えたかを共有できること が重要です
- 以下のような問題は、ファイルをコピーして残すだけでは解決できません
  - 意図せず大事なファイルを壊したり削除してしまった
  - 複数人が同じファイルを触って変更がぶつかった
  - 不具合が起きたときに、どの変更が原因かを特定できない
- バージョン管理は、単なるバックアップではなく
  共同作業を破綻させないために必要な仕組み です

バージョン管理システムとは

ファイルの変更履歴を記録する仕組み

- バージョン管理システム（Version Control System, VCS）とは
  ファイルの変更履歴をたどれるようにする仕組みです
- 開発作業中にトラブルが起きても過去の状態に戻れるだけでなく
  各メンバーの変更を個別に記録し、あとから統合できる ため
  各人が自由かつ安全に作業できるようになります
- ファイルをコピーして残すだけでは、変更の履歴や理由を記録できません
  VCSは履歴管理と共同作業の両方を支える仕組みです

チーム制作とバージョン管理システム

共同作業の強力なインフラ

- 全員が同じファイルを直接編集するような運用では
  変更の衝突や上書きによるトラブルを避けられません
- バージョン管理システムを使えば、各自が独立した環境で安全に作業し
  変更の正当性を確認してから本流に取り込む流れを作れます
- 履歴があることで、問題が起きても
  どの変更が原因か、どこで戻すべきか、誰に確認すべきか を判断できます
- つまりバージョン管理システムは、コードやアセットの管理ツールであると同時に
  チームでの共同作業を成立させるためのインフラ でもあります

バージョン管理ツール Git

現在もっとも使われているバージョン管理システム

- Gitは分散型バージョン管理システム（DVCS）です
- 各作業者のPCに履歴を含むリポジトリを持てるため
  ネットワークがない状態でも履歴管理とコミットができます
- ブランチとマージが高速で、試行錯誤しやすい設計になっています
- オープンソースで利用でき、関連ツールが非常に豊富です
- ゲーム制作に限らず、ソフトウェア開発全般で広く使われている
  デファクトスタンダードなバージョン管理ツールです

Git以外のバージョン管理ツール

Git以外のVCSの一例

- **Subversion (SVN)**
  Gitが普及する前に広く使われていた中央集権型のツールです
  ゲーム開発においては今でも使用されることがあります
- **Mercurial**
  Gitと同じくオープンソースで分散型のバージョン管理ツールです
  Gitほど広く使われてはいませんが、熱心なユーザーがいます
- **Perforce**
  大規模プロジェクト向けの商用ツールです
  AAAゲーム開発などで使用されることがあります

バージョン管理ツールの選択

Gitが唯一の選択肢ではない

- Gitは現在もっとも広く使われているバージョン管理ツールですが
  プロジェクトやチームの方針によっては他のツールを使うこともあります
- 例えばゲーム開発ではソースコード以外の大容量アセットを管理するために
  商用の管理ツールを使用したりGitと併用したりする場合があります
- Git以外にも多数のバージョン管理ツールがありますが
  バージョン管理の概念は共通しています
  バージョン管理システムの基本を理解すれば、ツールが変わっても応用できます

GitHubの紹介

Gitを使うならまず検討したい定番サービス

- Gitは単体でも使えますが、バックアップや共有、共同作業まで考えるなら
  まず候補に入るのが **GitHub** です
- **GitHubはGitをベースにしたリモートリポジトリのホスティングサービスです**
  個人制作でもチーム制作でも広く使われています
- Gitを学ぶなら、まずは **https://github.com** でアカウントを作成して
  無料プランから始めてみましょう

GitHubが提供するサービス

Gitホスティングに加えて、開発全体を支える

- GitHubには、リモートリポジトリを置くだけでなく
  変更を共有したり、履歴を確認したりしやすくする仕組みがあります
- たとえば **プルリクエスト（Pull Request）** で変更内容をレビューしたり
  Issue で課題や相談事項を管理したりできます
- さらに、公開プロフィール、README、Star、Follow などを通じて
  他の開発者の活動を見たり、自分の制作物を発信することもできます
- **つまりGitHubは、単にGitをホストするだけでなく**
  開発・共有・レビュー・発信などをまとめて扱える複合的なサービスです

GitとGitHubの関係

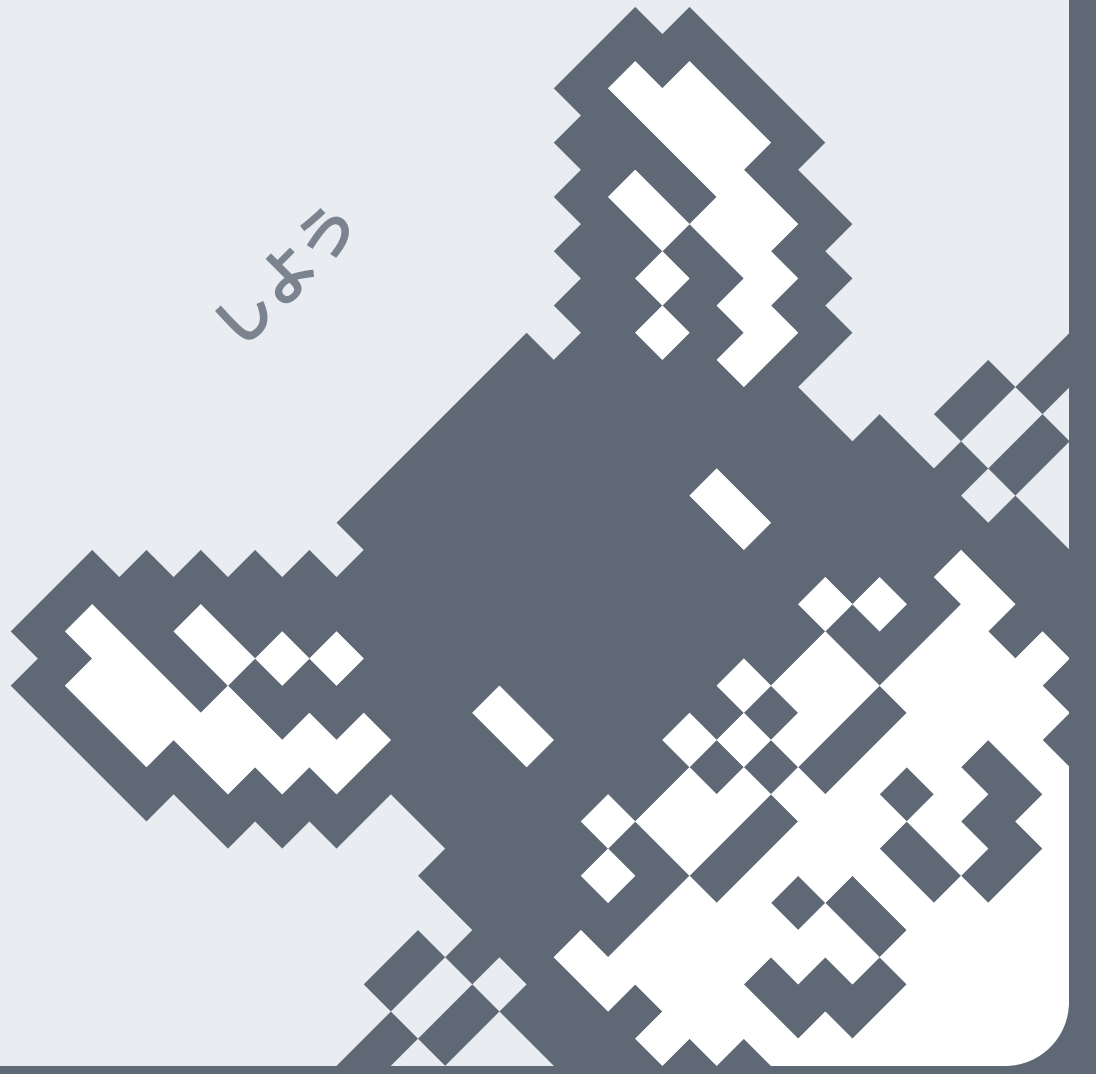
うっかり混同しないように注意

	Git	GitHub
役割	バージョン管理システム	Gitリポジトリのホスティング＋共同開発プラットフォーム
場所	自分のPC上で動く	インターネット上のサービス
組織	オープンソースコミュニティ	現在はMicrosoft社
関係	VCSの本体	Gitを使うウェブサービス

- **Git**は変更履歴を管理するVCSツールです
- **GitHub**はGitを土台に開発の支援やコミュニケーション機能を提供するサービスです
- 混同されやすいですが**GitとGitHubは別物ですので注意しましょう**

Gitの基本概念

基本の仕組みを理解しよう



リポジトリ

変更履歴を保存する保管場所

- **リポジトリ（Repository）** はプロジェクトの変更履歴を保存する場所です
  作業中の変更をコミットすることで、変更内容がリポジトリに記録されます
- 自分のPC上にあるリポジトリを**ローカルリポジトリ**
  GitHubなどのサーバー上にあるリポジトリを**リモートリポジトリ**と呼びます
- **ローカルリポジトリで変更を記録し**
  定期的にリモートリポジトリに反映することで
  変更を安全に保存し、他の人と共有できるようになります

Gitの基本的なワークフロー

手順	操作	コマンド
1	作業する	
2	変更内容を確認する	`git status`
3	変更をステージする	`git add <ファイル名>`
4	コミットする	`git commit -m "変更内容"`
5	リモートへ送る	`git push`

- この繰り返しが基本です

  「作業がまとまったらコミット」を忘れず、こまめに履歴を残しましょう

ステージングエリア

コミットする変更を選ぶ場所

- `git add` は **「今回のコミットに含めるファイルを選ぶ操作」** です
  変更したファイルを add する操作を**ステージングする**と言います
- Gitではコミットの前に、今回記録したい変更をステージングして
  ステージングエリアに入れます
- **`git commit` で記録されるのは、ステージングエリアに入っている内容だけです**
  コミットしたい変更だけをステージングしておくことで
  無関係な変更をコミットしてしまうのを防げます
- add の後でさらに編集した分はコミットに入りません
  必要なら、もう一度 `git add` しましょう

コミット

変更を履歴として記録する、Gitの基本的な操作

- **「ファイルを変更する → ステージングする → コミットする」**のが
  Gitを使った作業の基本的な流れです
- **コミット（commit）**はステージングした変更内容を履歴に記録する操作です
  コミットには変更の理由と内容を簡潔に説明した
  「コミットメッセージ」を添えます
- **コミットを積み重ねることでプロジェクトを更新し育てていくのが
  Gitの基本的な使い方です**

コミットのポイント

- **コミットメッセージは後から見てわかるように書きましょう**

  例: "プレイヤーがジャンプできるようにした"
  例: "敵のHP初期値を100から50に変更"

  駄目な例: "直した" ← 何を直したのかがわからない
  駄目な例: "2月3日の更新" ← タイムスタンプはGitが記録してくれるので不要です

- **変更は小さく、意味のある単位でコミットしましょう**
  複数の作業が混ざった大きすぎるコミットは影響範囲が広くなり
  問題が起きたときに原因を特定しづらくなります

- **コミットの積み重ねがプロジェクトの歴史になります**
  後から見て何をしたかがわかるように、丁寧なコミットを心掛けましょう

プッシュ・プル・フェッチ

リモートリポジトリと連携する3つの操作

- **プッシュ（push）**：ローカルリポジトリの変更をリモートリポジトリに送る
  自分の変更をGitHub上などのリモートリポジトリに反映させる操作です
- **プル（pull）**：リモートリポジトリの変更をローカルリポジトリに取り込む
  他の人の変更や、別のPCからの変更を受け取る操作です
  マージやコンフリクトの解決が必要になることがあります
- **フェッチ（fetch）**：リモートリポジトリの情報を取得する
  プルと違い、手元のファイルは変更しません

クローン

リモートリポジトリをローカルにコピーする

- **クローン（clone）** は既存のリモートリポジトリを
  自分のPC上にまるごとコピーする操作です
- 既存プロジェクトに参加する際や
  自分のリモートリポジトリを新しいPCに持ってくるときに使います

```
git clone https://github.com/ユーザー名/リポジトリ名.git
```

リポジトリを手元にコピーするのがGitの特徴

Gitは分散型バージョン管理システム（DVCS）です

- 各作業者のPCに履歴を含むリポジトリを持つため
  ネットワークにつながっていない状態でも履歴管理やコミットができます
  また、他の開発者の手元に履歴が残っていれば復旧しやすいのも特徴です
- 逆に中央集権型のVCS（Subversion等）では
  中央サーバーが履歴管理の中心になります
  履歴を更新したり取得したりする操作にはネットワーク接続が必要ですが
  ゲーム開発では同時編集を防ぐためのファイルロックの都合や
  大容量アセットの管理の都合から中央集権型のVCSが選ばれることもあります

Gitクライアントを使用する

コマンド操作が苦手でも大丈夫

- **GitHub Desktop**
  GitHub公式のシンプルなGUIクライアント
  まずはこれから始めるのがおすすめです
- **VS CodeのGit統合**
  エディタから直接 add・commit・push ができます
  コードと同じ画面で操作できるので便利です
- **SourceTree**
  ブランチやコミット履歴を視覚的に確認しやすい
  人気のGUIクライアント、こちらもおすすめです

おすすめのGitクライアント

まずはGitHub Desktopから始めてみましょう

- **GitHub Desktop** はGitHub公式のシンプルなGUIクライアントです
  単独でGitの主要な操作を行えます
  直感的なインターフェースで、初心者でも使いやすい設計になっています
- VS CodeのGit統合も便利です
  慣れたらこちらだけでも十分に作業できます
- **まずはGitHub Desktopから始めてみて**
  慣れてきたら他のクライアントも試してみるのがおすすめです

なお
より高度な操作では
コマンドライン（CLI）を
使うこともあります

順を追って覚えましょう

ゆっくりで構いません

GitHubをポートフォリオとして活用する

作ったものと活動の履歴を公開できる

- GitHubは、就職活動で制作実績を示す
  ポートフォリオとしても活用できます
- リポジトリに**README.md**を作成することで
  プロジェクトの概要・操作方法・スクリーンショットなどを
  まとめて公開できます
- **コミット履歴を積み上げて、プロジェクトを充実させることで**
  自分の活動の軌跡を示すことができます

GitHub Student Developer Pack

認証済みの学生はGitHub Proを無料で利用できる

- **GitHub Education**で学生認証を受けると
  Student Developer Packを利用できます
- 学生である間はGitHub Proを無料で利用できます
  是非申請してGitHub Proの機能を活用してください
- なお、認証には13歳以上で、学位または卒業証書を授与する
  教育課程に在籍していることなどの条件があります
  詳しくはGitHub Educationのサイトを確認してください
  https://education.github.com

Git運用の全体像

一人開発とチーム開発
それぞれのイメージ

ブランチ

作業を枝分かれさせる

- **ブランチ（branch）** は、変更の流れを分岐させる仕組みです
- 本流（main）から自分の作業用のブランチを枝分かれさせることで
  新機能の追加や実験的な変更を安全に試せます
- ブランチ上での変更はブランチ元に影響しません
  作業が完了したら本流に **マージ（merge）** して作業成果を統合します
- ブランチをマージして本流を育てていくのがGitの基本的な運用スタイルです

一人開発の運用イメージ

シンプルに管理する

```
main ブランチ（本流）
  ├─ commit: プロジェクト作成
  ├─ commit: プレイヤー移動を実装
  ├─ commit: 敵のスポーンを実装
  ├─ commit: UIを仮実装
  └─ commit: バグ修正

feature/jump ブランチ（実験的な実装）
  └─ main にマージして完成
```

- 最初はmainブランチでこまめにコミットするだけでも十分です
- 慣れたら実験的な機能を試すときにブランチを切る運用をしてみましょう

チーム開発の運用イメージ

プルリクエストでレビューを挟む

```
main（本流）
  ↑
  └ merge ← プルリクエスト（レビュー済み）
                ↑
                feature/attack-system（機能ブランチ）
                ├ commit: 攻撃の当たり判定を追加
                ├ commit: 攻撃アニメーションを繋ぐ
                └ commit: ダメージ計算を修正
```

- 各自がブランチで作業し、プルリクエストでレビューしてから main に取り込みます
- **コンフリクト（競合）** が発生した場合は
  どちらの変更を活かすかを手動で解決します

マージとは

別々に進めた変更をひとつにまとめる操作

- **マージ（merge）** は合併するという意味で、ここでは
  別ブランチで行った変更をひとつのブランチに取り込んで統合する操作です
- たとえば `feature/jump` で作ったジャンプ機能を
  `main` に取り込むときにマージします
- Gitの作業は「ブランチを分けて終わり」ではなく
  変更を取り込み先のブランチへ反映し続ける 流れで進みます

なぜマージが必要か

別のブランチで行った作業を共有する

- ブランチを使うと、他の作業へ影響を与えずに変更を進められます
- 作業が完了したら、変更を `main` などの共有するブランチに
  マージしてチーム全体で利用できるようにします
- たとえばチーム開発では `featureブランチ → mainブランチ` に
  変更を取り込むなどの流れで作業を進めます
- **マージは別々に進めた作業をひとつにまとめ**
  プロジェクトへ反映するための操作です

コンフリクトとは

同じ場所を別々に変更したときに起きる競合

- マージは通常、Gitが変更箇所を比較して自動的に統合を行います
- ですが、同じファイルの同じ行付近を別々のブランチで変更した場合などは
  変更が重なって自動的に統合できず **コンフリクト（conflict）** になります
- 自動では解決できない問題なため
  作業者が内容を確認して手動で統合する必要があります
- コンフリクトは厄介ですが、並行作業では起こりうるものです
  落ち着いて内容を確認し、解決しましょう

コンフリクト解決の手順

競合した変更を確認して統合する

1. 最新の `main` を取り込んで競合箇所を確認する
2. **競合マーカー**を見て、採用する内容を手で編集する
3. ビルドや実行で動作確認する
4. 修正内容を `add` して `commit` する
5. プッシュしてリモートに反映する

競合マーカーとは

Gitが「どこでぶつかったか」を示す印

- コンフリクトが起きると、Gitは競合箇所に**競合マーカー** を入れます

```
// mainを取り込んでコンフリクトが起きた例
<<<<<<< HEAD
    playerSpeed = 5f; // こっちは今いるブランチの内容
=======
    playerSpeed = 8f; // こっちは取り込もうとしている相手側の内容
>>>>>>> origin/main
```

- この例では `<<<<<<< HEAD` から `=======` までが
  現在の作業ブランチ側の内容です
- `=======` から `>>>>>>> origin/main` までが
  取り込もうとしている `main` 側の内容です

コンフリクト解決の例

競合マーカーを取り除く

- 例えば、自分のブランチではプレイヤーの移動速度を5にしていて
  mainブランチでは8にしているとします

```
<<<<<<< HEAD
    playerSpeed = 5f; // こっちは今いるブランチの内容
=======
    playerSpeed = 8f; // こっちは取り込もうとしている相手側の内容
>>>>>>> origin/main
```

- この場合、自分のブランチの内容を残すなら、`playerSpeed = 5f;` を残して
  `<<<<<<< HEAD` と `=======` と `>>>>>>> origin/main` を削除します

```
    playerSpeed = 5f; // 自分の変更を優先した
```

コンフリクト解決のポイント

競合を正しく直すために

- コンフリクトを解決するときは
  必要な行だけ残して競合マーカーは削除 します
- どちらか一方を採用するだけが正解ではありません
  状況によっては両方の更新意図を踏まえて
  コードを書き直す必要があるかもしれません
- 競合解決は難しく、ミスも混入しやすい作業です
  テキスト上で解決できたように見えても安心せずに
  ビルドや動作確認をして意図どおりに動くかを必ず確認しましょう

Gitをゲーム制作に活用する

Unityを例にしたGitの使い方と注意点

新しいリポジトリを作る手順

GitHubから始める方法

1. GitHubでリポジトリを新規作成する
   `.gitignore` のテンプレートには「Unity」を選択する
2. ローカルにクローンする
   `git clone <URL>`
3. Unityのプロジェクトファイルをフォルダに入れる
4. 変更内容と除外対象を確認する
   `git status`
5. `git add` → `git commit` → `git push`

管理しないファイルを除外する

.gitignore とは

- Gitで追跡したくないファイルやフォルダを指定するファイルです
- 例えばUnityプロジェクトでは
  `Library/` や `Temp/` などの環境依存のファイルが自動生成されます
  こうした**リポジトリに含めるべきではないファイルを**
  .gitignoreに指定することでGitの管理から除外できます
- GitHubがUnity用の.gitignoreテンプレートを公開しています
  参考にしてください
  https://github.com/github/gitignore/blob/main/Unity.gitignore

コミット時のよくあるミスと対処

避けるべき運用の落とし穴

- **コミットメッセージを雑に書く**
  → 「update」「fix」のような書き方では後から何の変更か分からなくなります
  具体的に何を変えたかを書く習慣をつけましょう

- **長期間コミットしないで変更を大量に溜め込む**
  → 問題が起きたときに原因の特定が困難になり、マージも難しくなります
  機能単位でこまめにコミットするよう心掛けてください

- **大容量のバイナリファイルを通常のGitで管理する**
  → リポジトリが肥大化して遅くなるほか、
  GitHubなどのファイルサイズ・容量制限に抵触することがあります
  Git LFS（後述）の使用を検討してください

Unity Prefabのコンフリクト

誰もがつまずくUnityでの鬼門

- Prefabは内部的にYAMLテキストで記述されていますが
  Unityが自動で編集するため意図しない競合が発生しやすいファイルです
- 同じPrefabを別ブランチで同時編集すると
  わずかな数値の変更でも大きなコンフリクトになりがちです
- 競合解消を誤ると参照切れやMissing Scriptが発生し
  シーン実行するまでわからない不具合の原因になります
- **Prefabのコンフリクトの解決は人力では難しい場合があります**
  このため最初からコンフリクトさせないことが基本の解決策になります

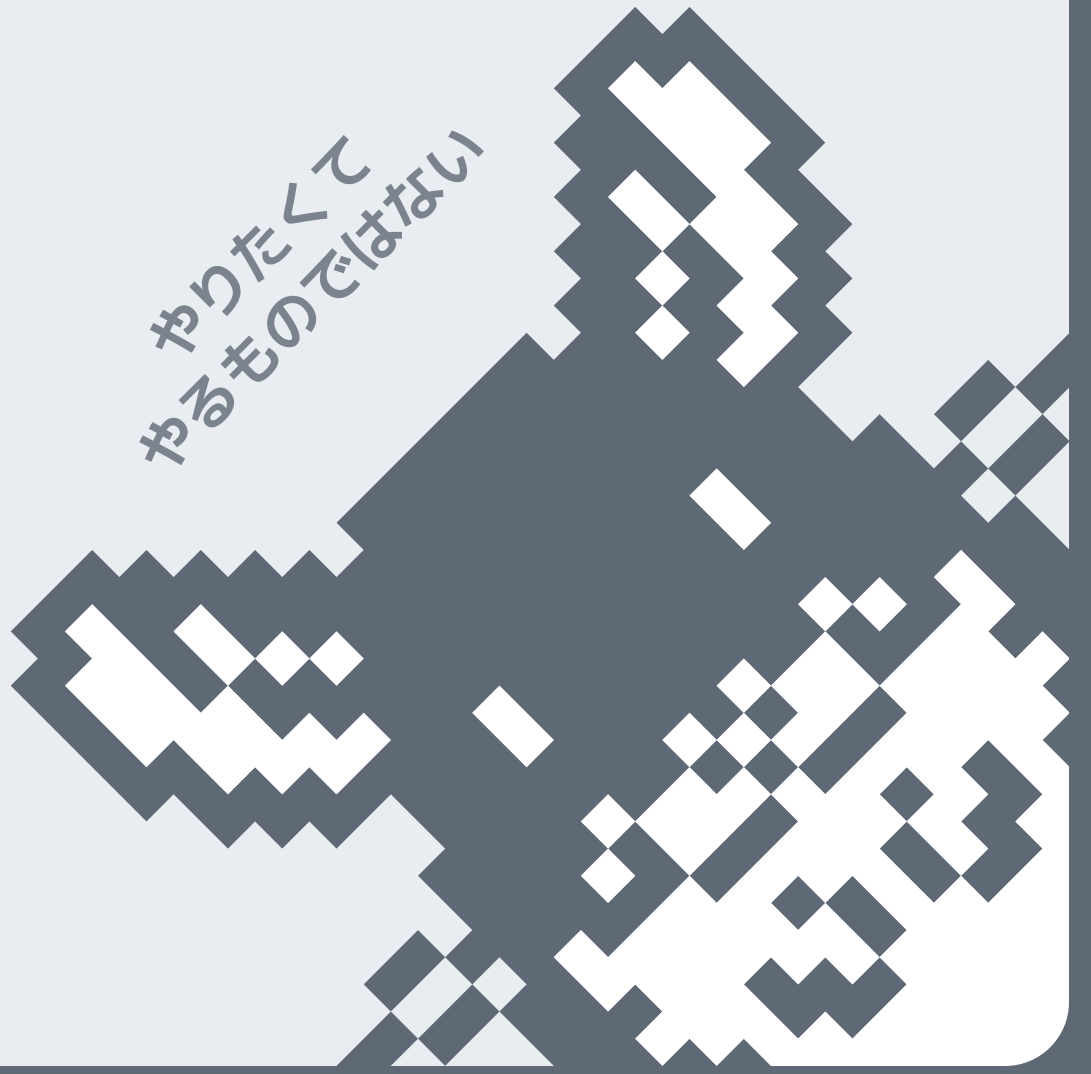
Prefabのコンフリクトの解決方法

Unity初心者向けの安全手順

1. まず競合ファイルを特定し、どちらの変更が必要か整理する
2. 迷ったらテキストで無理に直さず、片側を採用してUnityで再編集する
3. Unityを開き、PrefabをInspectorで開いて意図した設定に手で戻す
4. シーンで参照切れ（Missing）がないか確認し、Playして動作確認する
5. 問題なければ保存して `git add` → `git commit` する

- **テキストの競合を無理に直そうとせず、**
  片側の変更を採用してからUnityで再編集する方法が安全です

なお
PrefabのYAMLを理解して
テキストで解決することも
不可能ではありません



Prefabのコンフリクトの予防策

事前の運用ルールで回避する

- 担当分けを徹底し、同じPrefabを同時に触らないようにする
- 機能ごとにPrefabを分割し、変更範囲を小さくする
- Prefab Variant を活用して、共通部分と個別部分を分ける
- YAMLを理解すればテキスト上での解決も不可能ではありませんが
  時に非現実的なほどに難しい作業となります
- **コンフリクトを発生させないことが最も効果的な解決策です**
  事前に役割分担を相談し、運用を徹底しましょう

Git LFSとは

ゲーム制作では重いデータの管理も必要

- Gitはテキスト差分の管理が得意ですが
  画像・音声・3Dモデルなどのバイナリは差分効率が悪く
  コミットの都度リポジトリサイズが大きくなってしまいます
- 通常のGitリポジトリに大容量ファイルを入れ続けると
  clone や pull が非常に遅くなり、作業効率が著しく低下します
- また、サービスによってはリポジトリサイズに上限があり
  超えると新しい変更ができなくなります
- こうした問題を解決するために大きなファイルを扱う仕組みが
  Git LFS（Large File Storage） です

Git LFSの仕組み

大きなバイナリファイルを管理する

- **Git LFS（Large File Storage）**はGitの拡張機能で
  大容量ファイルをGitの履歴とは別のストレージで管理する仕組みです
- Gitの履歴にはファイルの実体ではなく「ポインター（参照情報）」だけを保存し
  実体ファイルはLFSサーバー側に置くことでリポジトリの肥大化を防ぎます
- `.psd` `.fbx` `.blend` `.wav` `.mp4` など
  大容量のバイナリファイルを扱う際は、Git LFSの導入を検討しましょう

Git LFSの基本的な使い方

インストールと設定

- Git LFSはGitの拡張機能で、環境によっては別途導入が必要です
  必要に応じて公式サイトから導入してください
  https://git-lfs.com/
- 導入後、次のコマンドでGit LFSの初期設定を行います
  （通常は環境ごとに一度実行すれば十分です）

```
# Git LFSの初期設定
git lfs install
```

Git LFSの設定例

.gitattributesについて

- Git LFSを使うには、リポジトリのルートに `.gitattributes` というファイルを作成し管理したい拡張子を指定します
- git lfs trackコマンドを使うと設定が `.gitattributes` に追記されます

```
# 例: よく使うLFS設定
git lfs track "*.psd"
git lfs track "*.fbx"
git lfs track "*.wav"
```

- `.gitattributes` ファイルはリポジトリで共有する必要があります
  チーム全員が同じLFS設定を使うために、必ずコミットしておきましょう

Git LFSの注意点

使う前に知っておくべきこと

- `.gitattributes` はリポジトリで共有しないと
  メンバー間でLFS設定がずれてトラブルの原因になります
  忘れずにコミットしましょう
- 対象にすべきでないファイルまでLFSに含めると逆に運用が複雑になります
  チーム運用の場合はLFS管理するファイルを事前に相談しましょう
- GitHubなどのホスティングサービスでは
  Git LFSの保存容量とデータ転送量に利用枠を設定しています
  導入前に最新の利用枠や料金を確認しましょう

Git LFSは無敵ではない

その他の大容量ファイル管理の選択肢

- Git LFSは大容量ファイルを管理するための便利なツールですが
  すべてのケースで最適とは限りません
- 頻繁に更新される大容量アセットや巨大な動画ファイルなどは
  Git LFSへの負担が大きすぎたり非効率な場合もあります
- 動画のような独立性の高い大容量ファイルのやりとりには
  クラウドストレージ（Google Drive、Dropboxなど）の方が向いています
  必要に応じてGit LFS以外の管理方法の活用も検討してください

UnityとGit LFS

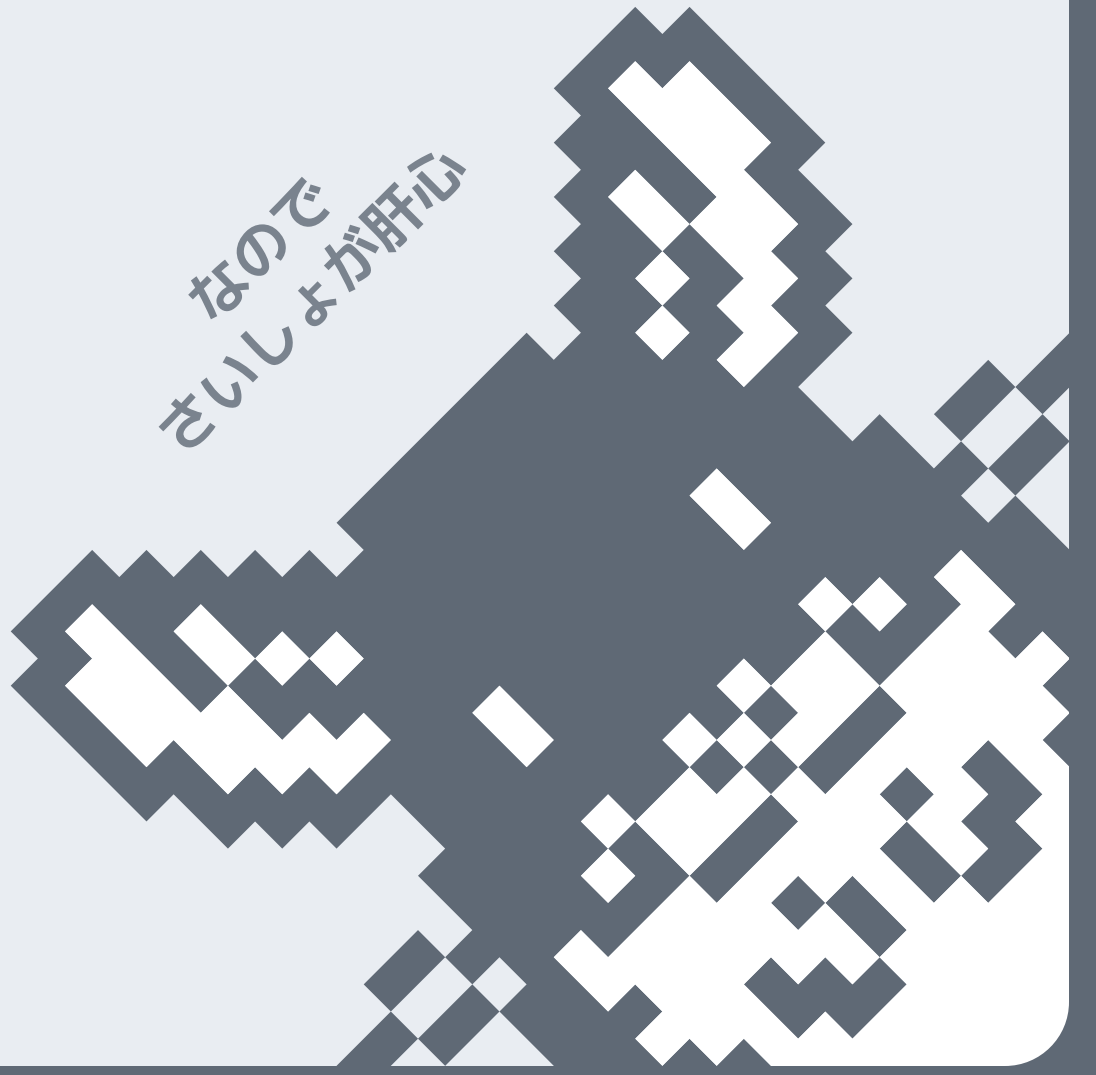
おすすめのLFS管理対象

- **画像・音声・3Dモデルなど**
  サイズが大きくなりやすいバイナリファイルはLFS管理の候補になります
- Unity向けの.gitattributesの設定例が公開されています
  これをベースに、プロジェクトで必要な拡張子を都度追加していくのがおすすめです
  https://github.com/gitattributes/gitattributes/blob/master/Unity.gitattributes
- テキスト形式の `.asset` や `.meta` は通常のGitで管理します
  ただし、ライトベイクデータなどサイズの大きなデータは
  個別にLFS管理を検討してください

注意

後からLFSを設定しただけでは過去にコミットしたファイルはLFSに移行されません

移行のためには履歴の書き換えが必要です



まとめ

これだけ覚えて、まずは触ってみましょう

用語	役割
Git	変更履歴を管理するツール
GitHub	Gitリポジトリを共有・活用するサービス
リポジトリ	プロジェクトの変更履歴を保存する場所
ステージング	次のコミットに含める変更を選ぶこと
コミット	選んだ変更を履歴に記録すること
クローン	リモートリポジトリを手元にコピーすること
プッシュ・プル・フェッチ	変更を送る・取り込む・情報を取得する操作
ブランチ・マージ	作業を分け、変更を統合する仕組み

「わたしはエンジニアじゃない！」という方へ

Gitはエンジニア以外も使って得するツールです

- Gitはドキュメント管理やデザインファイルのバージョン管理など
  エンジニア以外の職種でも使われはじめています
- 現在のソフトウェア開発の世界では
  Gitが標準的なバージョン管理ツールになっています
  エンジニア以外の人でもGitを使えると仕事の幅が広がります
- **操作に迷ったときはAIに相談するのも手です**
  手順やコマンドの意味を確認し、対話しながら活用してみましょう
- **Gitはエンジニア以外の人にも役立つツールです**
  恐れずに挑戦してみてください！

最後に

Gitを使うにあたって

- Gitは複雑なツールで、使いこなすには経験が必要です
- ですが完璧に理解してから始める必要はありません
  基本的な操作から少しずつ身につけていきましょう
- はじめは練習用のリポジトリで
  コミットやブランチの操作に慣れるのがおすすめです
- **Gitは使いながら覚えるものです**
  まずは手を動かしてみましょう！

参考

- **GitHub Docs: GitHub と Git について**
  https://docs.github.com/ja/get-started/start-your-journey/about-github-and-git
- **GitHub Docs: Hello World**
  https://docs.github.com/ja/get-started/start-your-journey/hello-world
- **Pro Git 日本語版**
  https://git-scm.com/book/ja/v2
- **Unity Learn: Set Up Git LFS to Manage Large Files in Unity**
  https://learn.unity.com/course/collaborate-with-github-desktop/tutorial/set-up-git-lfs-to-manage-large-files-in-unity

おわり