G^2 のための Git 概説

制作者: 原凪利

制作日: 2024/6/9

目次

1　はじめに

今回は細かいことを省略して概説する。

しかし、Git を含むバージョン管理システムは社会に広く普及しているため、各自で理解を深めることを推奨する。

また、本サークルにおいて Git、及び GitHub の、少なくとも今回の内容は必修に値するものとする。

内容として、2 章から 5 章は知識編、6 章以降が実践編となっている。

なお、この解説は Git、GitHub、GitHub Desktop、Unity を用いた開発を想定している。

2　Git とは

Git とは、分散型のバージョン管理システムである。

2.1　バージョン管理システムとは

バージョン管理システムとは端的に言えば、ファイルへの変更履歴を記録しておくシステムのことである。

2.2　　分散型と集中型の違い

バージョン管理システムは分散型と集中型に大別される。

集中型は、プロジェクトメンバーで共有しているオンライン上のファイル群にアクセスして、直接変更を加える。

分散型は、プロジェクトメンバーで共有しているオンライン上のファイル群をそれぞれの開発者のコンピュータ内にコピーして、コピーしたファイル群に変更を加える。そして、自身のコンピュータ内で行った変更が正常に動くことを確認してから、それらをオンライン上にアップロードする。

3　構造・基本的な使い方

3.1　共同開発における全体の構造

バージョン管理システムは、オンライン上にあるファイル群の保管庫であるリモートリポジトリに変更点のみをアップロードすることで、共同開発を効率的に行うことを実現している。

共同開発を行う際の構造を全体的に広く見たときの概要を以下の図 1 に示す。

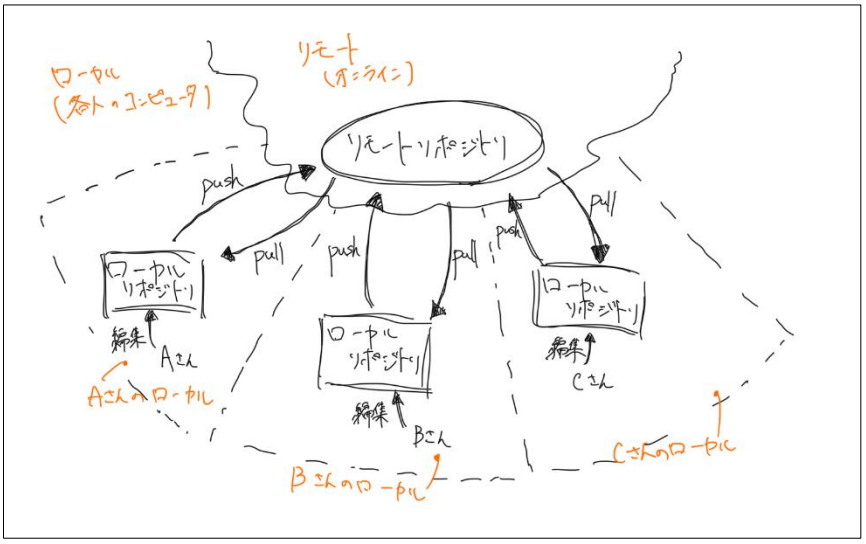

図 1: 共同開発における構造概要

Git では、オンライン環境のことをリモート(remote)、各人のコンピュータ内の環境のことをローカル(local)と呼んでいる。また、ファイル群、ソースコード群を管理する保管庫のことをリポジトリ(repository)と呼んでいる。

Git における共同開発では、リモートリポジトリからコピーしてきたローカルリポジトリを各開発者が編集する。ここで、自分のコンピュータ内にローカルリポジトリが無い状態でリモートリポジトリから最初にコピーすることを、クローン(clone)と呼んでいる。clone の概要は同章 2 節で説明する。

各開発者は clone してきたローカルリポジトリ内で内容の変更を行い、問題がなければリモートリポジトリにアップロードする。このアップロードのことを、プッシュ(push)という。

また、他者が行った変更を自身のローカルリポジトリに反映させることを、プル(pull)という。ただし pull に関して、厳密に言えば少し違う。その詳細、及び push、pull の概要は同章 3 節で説明する。

3.2　プロジェクトへの参加(clone)

この節では、プロジェクト(リモートリポジトリ)が既に存在していて、それに参加することを想定する。リモートリポジトリの作成方法に関しては、実践編の方で説明する。

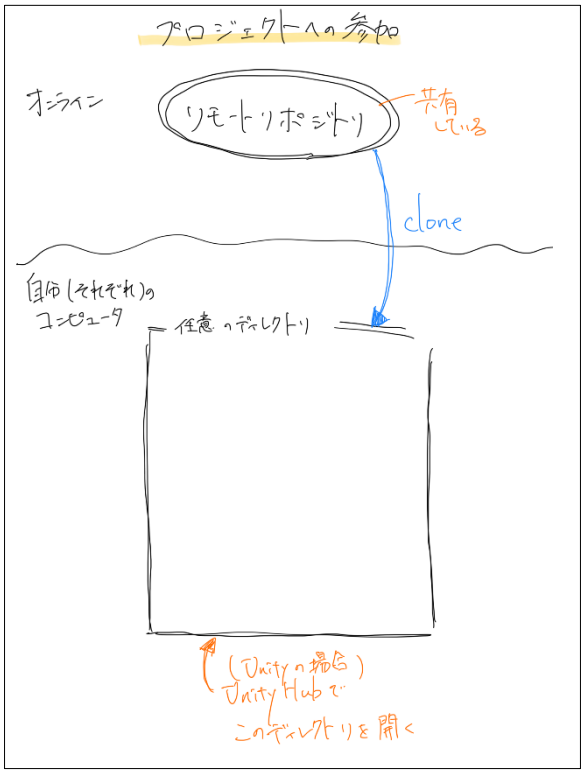

図 2: プロジェクトへの参加概要

プロジェクトへの参加方法の概要を、図 2 に示した。

まずはローカルリポジトリを準備する必要がある。

そのため、前の節で述べた clone を行う。また、Unity で開発を行う場合は、Unity Hub を用いて clone してきたプロジェクトを開く必要がある。

3.3　編集の仕方(add、commit、push、pull (fetch、merge))

この節では、前の節で clone してきたローカルリポジトリを編集し、その変更をリモートリポジトリに反映させる方法、また、他人が変更したリモートリポジトリを自身のローカル

環境に反映させる方法を紹介する。

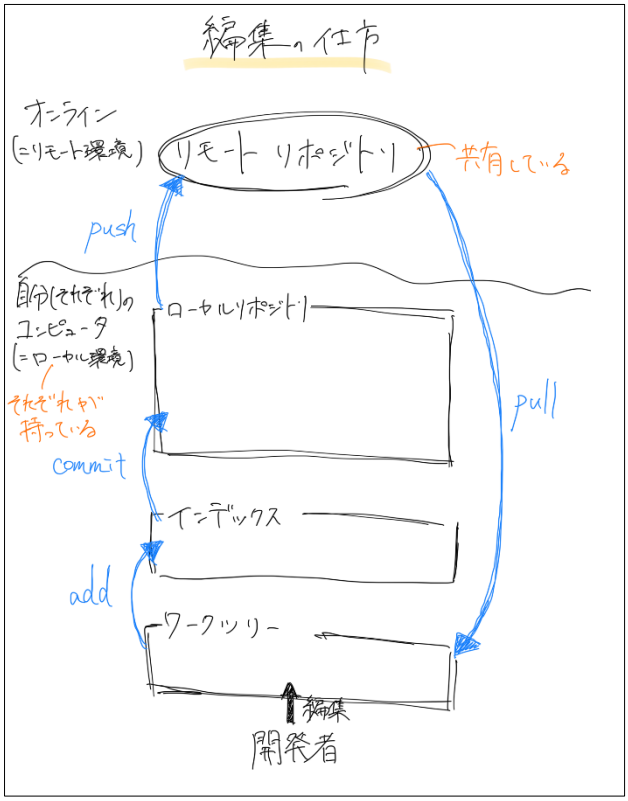

図 3: 編集の仕方概要

編集の仕方概要を図 3 に示した。

リモートリポジトリを clone すると、ワークツリー、インデックス、ローカルリポジトリという領域がローカル内に自動的に作られる。

ワークツリーは自身の編集する対象となるディレクトリのことであり、インデックスは行った編集の内どのファイルの変更を反映するかを控えておく場所のことである。

また、ローカルリポジトリはローカル内におけるリポジトリのことであり、ワークツリーとは別のものである。ワークツリーは編集する対象のファイルそのものであるが、ローカルリポジトリは変更履歴などを保管しているデータベースである。

開発者はワークツリーで編集を行い、行った変更の内反映させたいものをインデックスに add する。ひとつの変更が完了しそれらの変更をインデックスに add したら、変更の内容を記してからローカルリポジトリにそれらをコミット(commit)する。この commit が、バージョンと呼ばれる変更履歴となる。

ローカルリポジトリに異常がないことを確認したら、リモートリポジトリにそれらを push する。

また、他人がリモートリポジトリを変更しているか確認するためにフェッチ(fetch)し、更

新があったらそれをワークツリーにマージ(merge)する。ただし基本的には、fetch と merge をあわせたショートカットである pull が使われる。

全くもって穿っていない極めてひねくれたまとめ方をすると、ワークツリーで変更をし、そのうち反映したいものをインデックスに add し、そのひと纏まりの変更を名前付きでローカルリポジトリに commit する。このとき、ローカルリポジトリには変更履歴が保管されている。そしてその変更履歴を全体で共有するために、リモートリポジトリに push する。するとリモートリポジトリに変更履歴が反映され、その変更が実際に行われる、といった感じである。

4　ブランチ

4.1　ブランチとは

ブランチとは、バージョンを分岐させることが出来る機能のことである。

機能の追加、バグ修正などを独立、並行して行えて、例えばもし機能 A ブランチの制作中により優先度の高い他の機能 B を追加する必要がある場合、機能 A のない状態のマスターブランチ(メインとなるブランチ)から機能 B のブランチを作成することが出来る。

また機能 A が不要になった場合でも、マスターブランチに結合していなければ影響が出ることがない。

ここで、図 4 にブランチのイメージを示した。ここで、変更の履歴とは commit の履歴のことである。

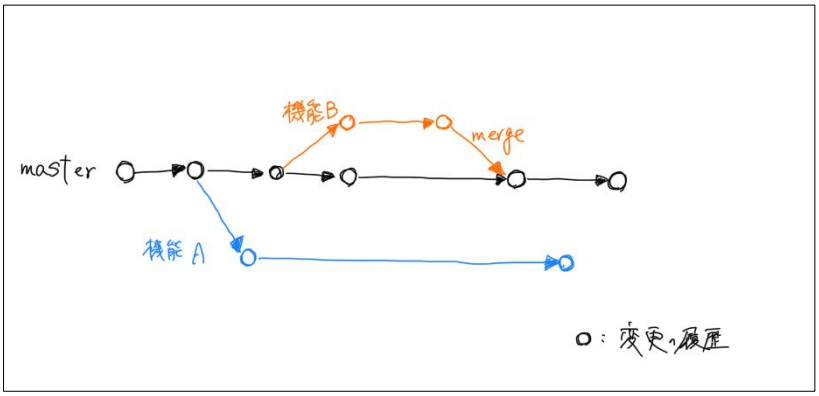

図 4: ブランチのイメージ

ブランチを他のブランチに結合するには、merge を行う必要があり、これには要求(pull request)が必要である。この大きな目的は、競合の発見である。競合に関しては、次の節で解説する。

また別の理由として、merge を許可する側のブランチの管理者が、merge しても問題ないか人的に確認をするためということが考えられる。例えば既存のプロジェクト A に新入社員 C が参加し、マスターブランチを管理しているベテラン B に研修としてミニゲームを作るよう依頼された場合を想定する。C はミニゲーム用のブランチを作成し、その中で目的のミニゲームを完成させ、pull request をマスターブランチに対して送信した。このとき、ミニゲームに潜在的で致命的なバグがあった場合、確認があることで B さんはそれを発見することが出来る。

また、GitHub では pull request に対してコメントを残すことが出来るため、修正して欲しい部分をコメントすることが出来る。

用語として、別のバージョン(別の commit 直後)に移動することをチェックアウト(check out)という。

ブランチを使うことで、複数の差分バージョンの比較なども簡単に行うことが出来るため、使えると便利である。

4.2 競合(コンフリクト)

ブランチの merge においては、競合ということが起こることが往々にしてある。

競合は、複数のブランチで同じ部分を変更し、それらを merge しようとした際に発生する。

競合が発生したら、merge を許可する側のブランチがどの変更を採用するのか決定する必要がある。

5 用語の整理

Remote(リモート)	：オンラインのこと
Local(ローカル)	：開発者のコンピュータのことであり、開発者の数だけある
Repository(リポジトリ)	：ファイルを保管する保管庫
Work Tree(ワークツリー)	：開発者が編集する直接の対象
Index(インデックス)	：反映する変更の置き場所であり、ここにあるファイルは staging(ステージング)されていると言う
Clone(クローン)	：リモートリポジトリからローカルリポジトリを作成すること
Add	：反映する変更を index に追加すること
Commit(コミット)	：コメントを付けて index にある変更を保存することであり、これがバージョン(変更履歴)になる
Push(プッシュ)	：ローカルリポジトリからリモートリポジトリに commit 履歴をアップロードすることで、commit 履歴を反映することでファイ

ル群の中身も更新される

Pull(プル)　　：ワークツリーをリモートリポジトリの最新の状態に更新すること(fetch + Merge)

Fetch(フェッチ)　　：リモートリポジトリの最新の状態を読み込むこと

Merge(マージ)　　：Push の一部としては、fetch した最新の状態をワークツリーに反映させること

：ブランチにおいては、他のブランチと結合すること

Branch(ブランチ)　　：バージョンを分岐させる機能

6　リポジトリの作成・初期化（実践）

これ以降の章では、実際の手順を紹介していく。

6.1　GitHub Desktop を用いた方法

6.1.1 リモートリポジトリの用意

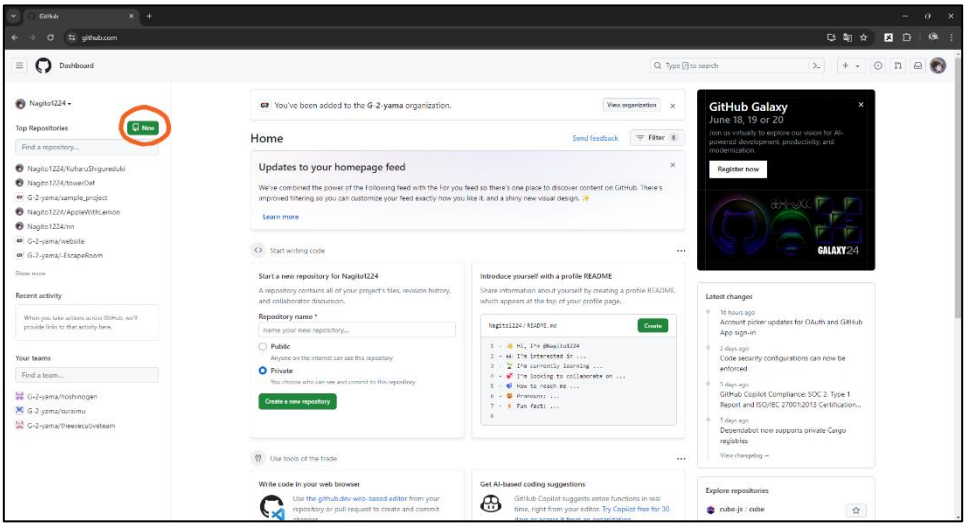

図 5: GitHub　ホーム画面

まずはプロジェクトを始めるために、リモートリポジトリの作成と初期化を行う。

リモートリポジトリを作成するには、まず GitHub のウェブサイトにアクセスし、ログインをする必要がある。また、アカウントを持っていない場合は作成をする。

ログインが完了すると、図 5 のような画面になる。

ここで新たなリモートリポジトリを作成するには､橙色の丸で示した左上の New を押す。

すると、図 6 のような画面になる。

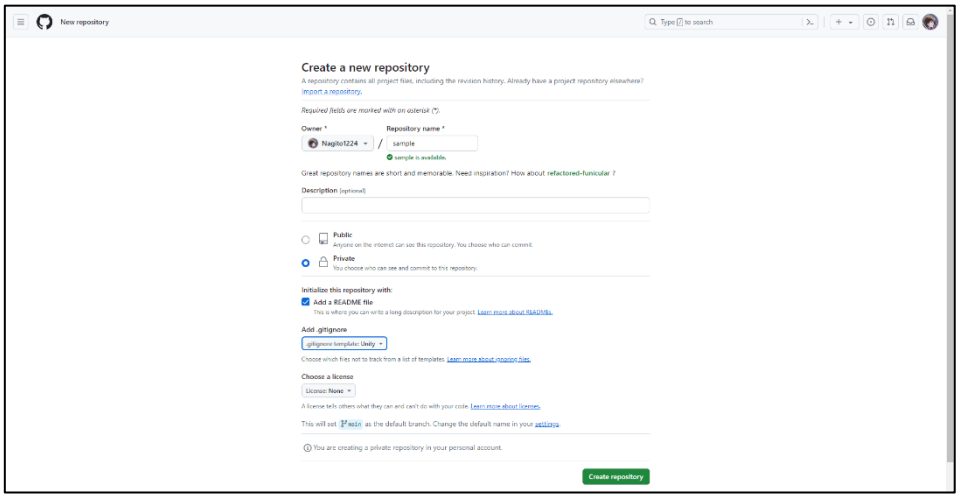

図 6: GitHub リモートリポジトリ作成画面

Owner にはリモートリポジトリの所有者、Repository name にはリモートリポジトリの名前、Description にはリポジトリの説明(任意)を入力する。

また、Public はリポジトリの中身を全ての人が閲覧可能な設定で、Private は許可した人のみ閲覧可能という設定である。

Add a README file は説明書のようなものを含めるかどうか、Add .gitignore は除外したいファイル群のテンプレートの選択、Choose a license は他人がリポジトリのファイルを利用する際のライセンスの選択をする。gitignore ファイルは、どのツールに最適化するか選択する場所だと考えてもらって構わない。例えば Unity を用いて制作を行う場合、ここで Unity を選択しておけば、Unity が用意したファイルの内、リポジトリで管理する必要のないファイルを除外してくれる。

入力を終えたら、Create repository を押す。

これで、リモートリポジトリの作成は完了した。

6.1.2 プロジェクトの初期化

Unity を用いる場合、Unity が用意するファイルをリモートリポジトリに含める必要がある。

まず、リモートリポジトリの clone を行う。クローンの手順については冗長性を減らすために、7 章を参照して行うものとする。

次に、Unity のプロジェクトを作成する。この手順に関しては Unity の基本操作について説明した資料を見て欲しい。

この時点でローカルには Unity のプロジェクトディレクトリと、クローンしたリポジトリ(ワークツリー)が存在する。

ここで、Unity のプロジェクトディレクトリの中身を全てワークツリーの中に移動する。

この段階で GitHub Desktop を見ると、28 個の変更がインデックスの中に自動で追加されていることが分かる(インデックスに入っている変更の数に多少の変動はあるかもしれないけれど、10000 を超えるような場合は流石に何かミスっていると思ったほうが良い)。このときの GitHub Desktop の様子を図 7 に示す。

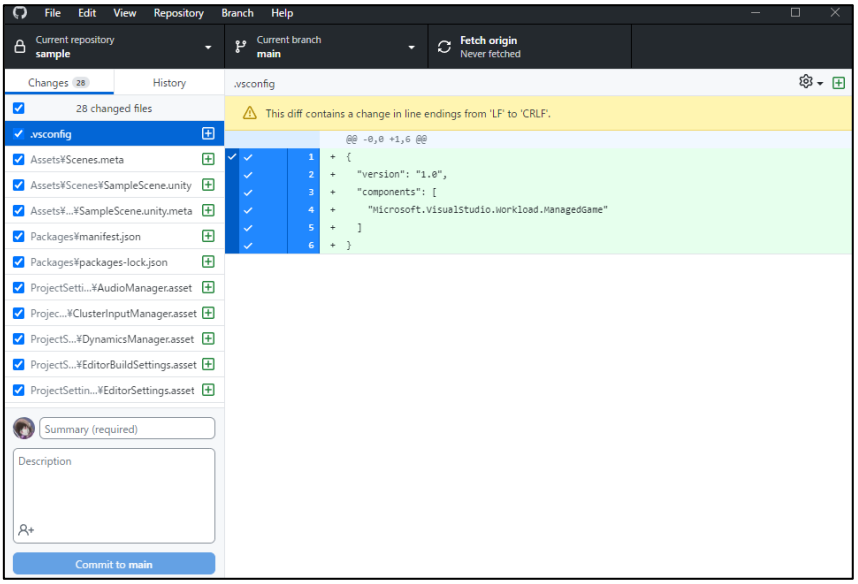

図 7: 最初の commit 時の GitHub Desktop 画面

ここで init commit や first commit など適当なコメントとともに commit を行ってから、GitHub Desktop 中央よりやや右の上の方にある Push origin を押すと、プロジェクトの初期化が完了する。

commit、push の詳細な方法に関しては 8 章を参照することとする。

最後に、Unity Hub のプロジェクトの中から先程作成したプロジェクト(ファイルを移動したためにもう開かないプロジェクト)を削除して、”追加”から先ほど Unity プロジェクトを移行した先のディレクトリ(ワークツリー)を開けば編集を開始できる。

6.2 Visual Studio を用いた方法（推奨）

Visual Studio を用いてローカルリポジトリ、及びリモートリポジトリを同時に作成する。ここでは、既存のディレクトリ(Unity プロジェクト)をワークツリーとしてローカルリポジトリを作成する。同時にそれをもとにリモートリポジトリを作成し、push を行うということをする。

特定のディレクトリを Git として初期化し、リモートリポジトリを作成してそこに push するというのは、おおよそ CLI での操作と同じことをしているが、Visual Studio はこれらの初期化に関わる操作を一つの GUI 操作として実現している。

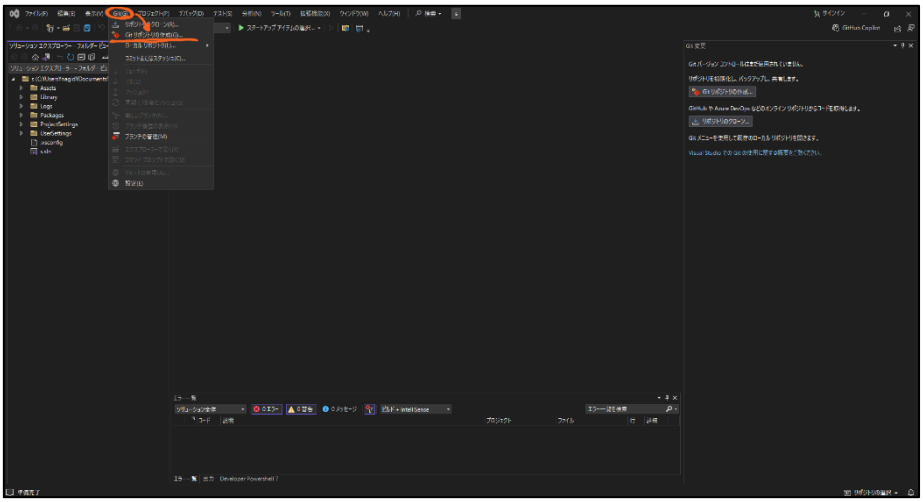

図 8: Visual Studio の Git ツール選択画面

まずは、Unity Hub からプロジェクトを作成する。この手順に関しては Unity の基本操作について説明した資料を見て欲しい。

そして、Visual Studio を起動して、右側にある”ローカルフォルダを開く(F)”を選択する。ここはバージョンに依って違うかもしれないけれど、その場合は同様の操作を探して実行する。ここで先程開いた Unity のプロジェクトを選択して、プロジェクトを開く。

次に、画面上部の”Git(G)”を選択して、その中から”Git リポジトリの作成(G)”を選択する。この様子を図 8 に示した。

(“Git(G)”の項目がない場合は、”ツール(T)”から”ツールと機能を取得(T)”を選択して、”個別のコンポーネント”から”Git for Windows”にチェックを入れて有効にすると表示される)

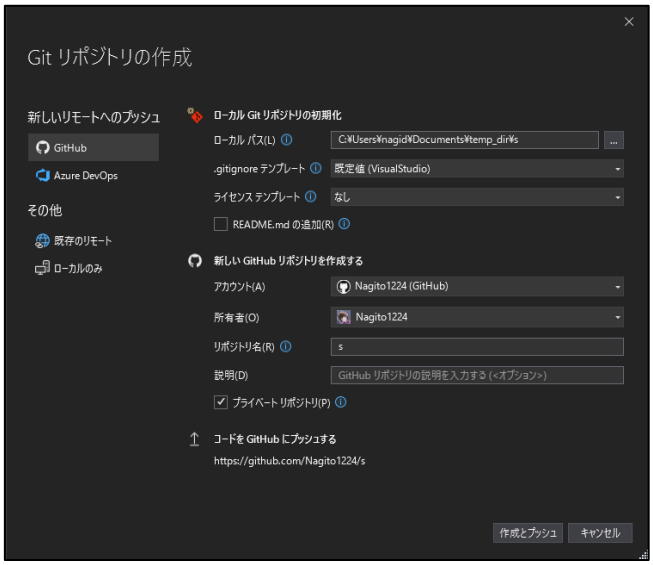

図 9: Visual Studio のリポジトリ作成画面

図 9 のようなウィンドウが表示されたら、ローカルパスが適切なディレクトリ(ここでは先ほど開いた Unity プロジェクト)であることを確認して.gitignore テンプレートに適切なものを選択する。ここで.gitignore というのがあるが、これは 6 章 1 節にある説明を見て欲しい。端的に言えば、Git で管理する必要のないファイルを除外して、最適化してくれるファイルのことである。今回は Unity を選択する。

ライセンステンプレートについても、6 章 1 節の説明を見て欲しい。今回はなしで進める。README.md も同様である。

“新しい GitHub リポジトリを作成する”は、リモートリポジトリの作成についてである。

ここでは、GitHub アカウントと連携して適当な入力をして進める。このあたりの説明も 6 章 1 節と同様である。

また、ローカルのみを作成する場合や、既存のリモートに追加する場合は、左側”その他”から適宜選択すること。

最後に作成とプッシュを押せば完了である。

なお、以降の編集についても GitHub Desktop を用いずに Visual Studio や CLI のみで可能であるが、ここではあえて説明しないものとする。ただし、考え方や用語、構造については同じであるため、移行は簡単である。

7　プロジェクトへの参加(clone の仕方) (実践)

この章では、リモートリポジトリが用意されていてローカルリポジトリがまだない状態から、ローカルリポジトリを作成する方法について紹介する。

まずは GitHub の web サイトの中で参加したいリモートリポジトリのページに移動する。

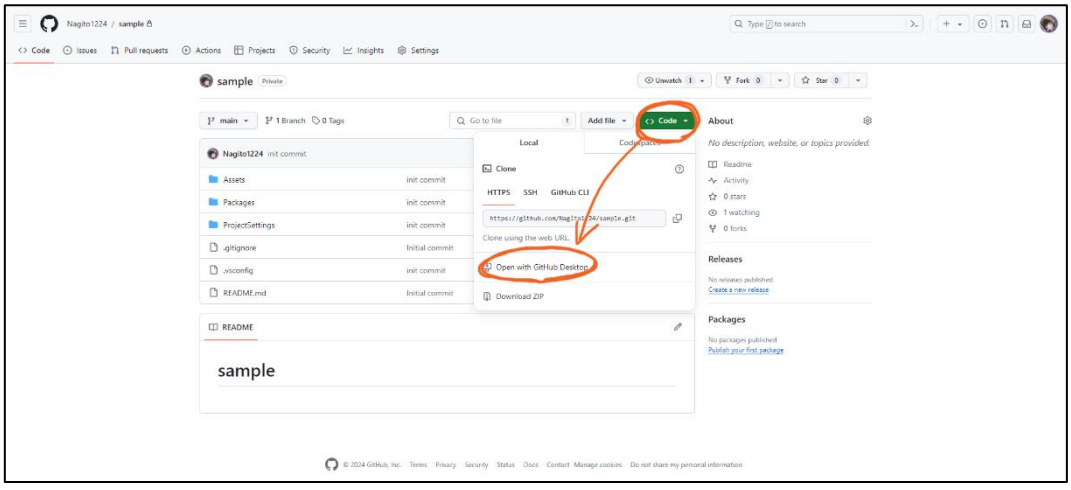

図 10: GitHub のリポジトリ画面(clone)

その中で、図 10 に示したように、”Code”を押してから、”Open with GitHub Desktop”を押すと、図 11 のような画面に自動的に移動する。自動的に移動しない場合は、GitHub Desktop 左上の”Current repository”から、”Add”、”Clone a repository”を押すか、あるいは”Clone a repository from the Internet…”を押して、上の入力欄に web サイトのリポジトリページ、”Code”の HTTPS の下にある URL をペーストすれば同様の画面になる。

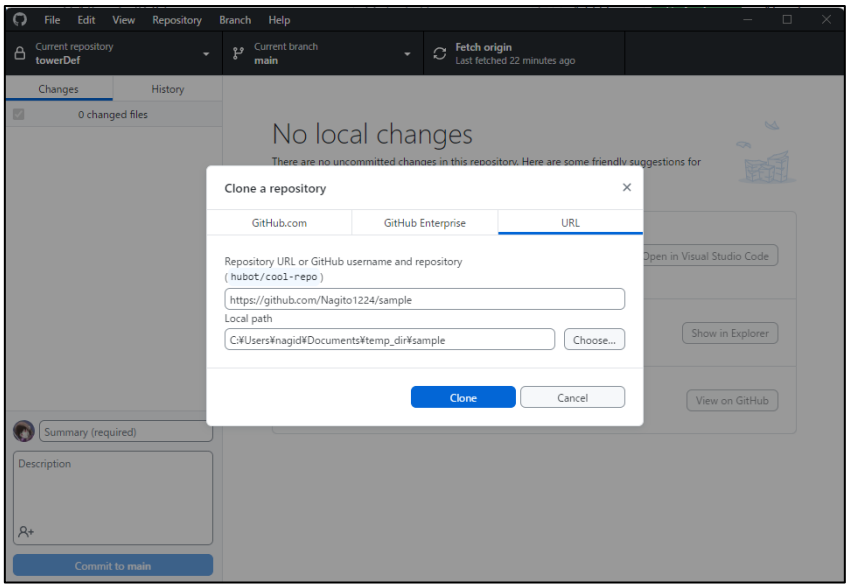

図 11: GitHub Desktop の clone 画面

ここで上にはリモートリポジトリの URL、下にはローカルリポジトリを作りたい任意のディレクトリを入力する。ただし、自動でこの画面に遷移した場合は上の欄はデフォルトで入力されている。

次に、”Clone”を押すとローカルリポジトリが作成され、ワークツリーの中にリモートリポジトリの中身と同じものが作成される。

6 章のプロジェクトの初期化を行っている場合は、ここで 6 章に戻る。

最後に、Unity Hub の”追加”からクローンしたディレクトリ(ワークツリー)を開けば編集を開始できる。

8 編集方法（実践）

8.1 add・commit（ローカル内での変更の保存）

ワークツリー内で変更を加えたら、GitHub Desktop を使っている場合であれば自動的に add が行われる。ステージングされた(インデックス内の)ファイルは GitHub Desktop 左側に表示されている。図 12 の橙色で囲まれた部分がそれである。

また、右側にはファイルの中身が書かれている。

変更を反映したくない場合は、これらのチャックを外すことでインデックスから除外することが出来る。

（ただし、CLI や Visual Studio、その他 EGit、Tortoise Git、Sourcetree などを用いる場合、add は手動で行う必要があることに注意する。というか、基本は手動であり、GitHub Desktop が特殊であると考えたほうが良い。

また、ここで行った変更が表示されない場合は、ワークツリーで変更を保存したか確認して欲しい）

ある程度まとまった変更を行ったら、適当なコメントを書いて commit を行う。

行った変更は、commit 単位でローカルリポジトリに保存される。変更履歴は左側の History から確認でき、任意の変更箇所で右クリックをすることで様々な操作を行うことが出来る。

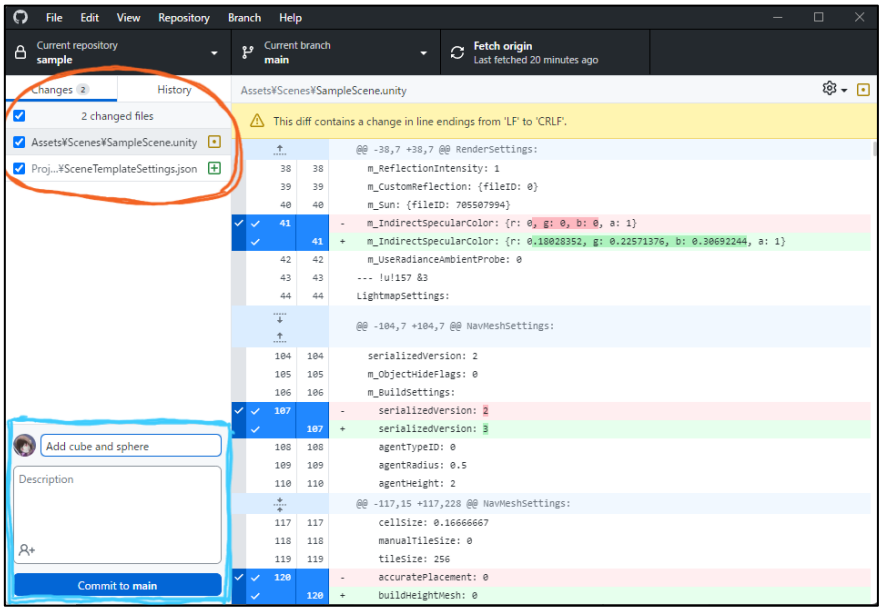

図 12: GitHub Desktop の commit 説明用画像

ここでは、図 13 に示した 2 つと、push 前の変更を右クリックしたときのメニューである Amend、及び Undo について説明する。

まず、Undo は commit を取り消して、直前の状態に戻す操作である。間違えて commit してしまった場合などに有効に使える。

Amend は、直前の commit の内容を修正する操作である。ただし既に commit してしまったファイルの消去は行えず、追加のみであるため、追加し忘れた変更がある場合は有効に使える。

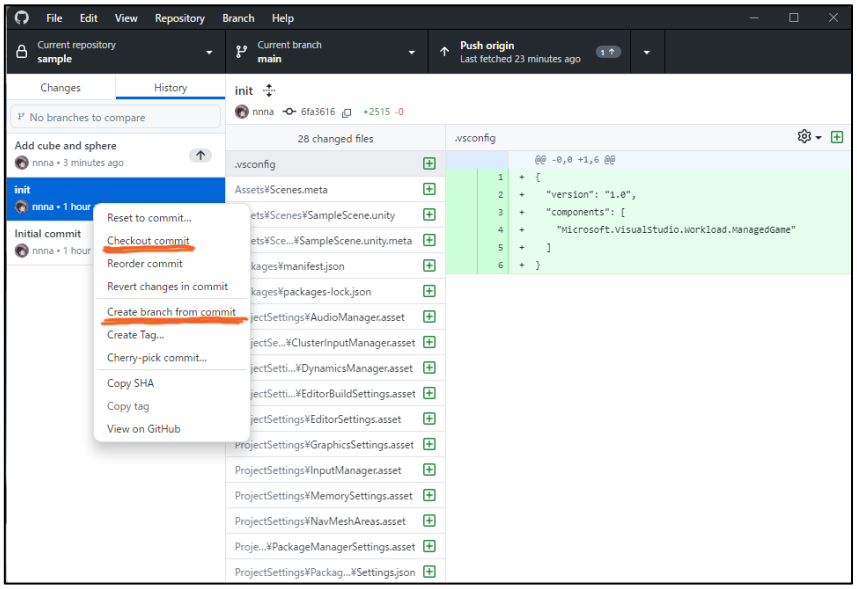

図 13: GitHub Desktop の History 表示画面

Checkout commit は知識編で紹介したように、そのバージョンに移動することである。以前の commit 時点に戻って操作をやり直したい場合は有効であるが、以降示すブランチという機能を用いたほうが良い。別ブランチのバージョンに移動する際も、同様に checkout という。

Create branch form commit は、そのバージョン(commit 直後)からブランチを切ることが出来る。ブランチを切るとは、ブランチを作って分岐させることを意味している。

ブランチについては、4 章と 9 章で説明している。

8.2 push (リモートへの反映)

ローカルである程度編集をして、リモートに反映しても問題ないことを確認したら push を行う。

図 14 に示したように、GitHub Desktop 中央よりやや右上の方にある”Push origin”を押すと、push が行われる。これで、ローカルリポジトリの変更がリモートリポジトリに変更が反映される。

なお、”History”の各項目の内右側に上矢印が書かれているものが、push 前の変更たちである。

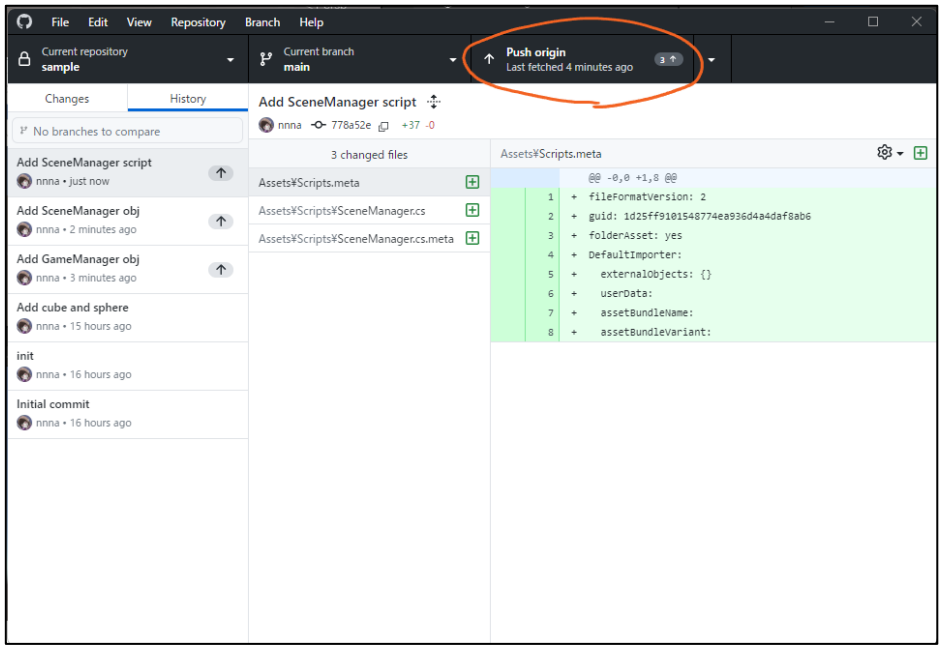

図 14: GitHub Desktop の push 前画面

8.3 pull (最新状態のコピー)

他の開発者が変更を行った場合、pull(fetch + merge)を行って自身のワークツリーを最新の状態にする必要がある。

pull は図 14 の push と同じ場所で行うことが出来る。

”Fetch origin”と表示されている場所を押せば、fetch を行うことが出来、変更があれば表示が pull に変わる。そのまま”pull”を押せば、ワークツリーはリモートリポジトリの最新の状態が反映された状態になる。

9 ブランチ（実践）

ブランチの作成は GitHub Desktop の”History”内の、ブランチを切るもとになる commit を右クリックして表示される”Create branch from commit”を選択することで行われる。

ここで、図 14 の push と同じ場所にある”Publish branch”を選択するとリモートリポジトリにブランチが反映される点に注意する。

GitHub Desktop 内で merge を行いたい場合(ローカル内のみで使っているブランチを merge する方法)は、残したい大筋となる方のブランチにチェックアウト(移動)して、図 15 のような手順で選択していき、”Merge into ~”というメニューで結合したいブランチを選択して行えば良い。結合時に Rebase や Squash という項目を選択することができるが、ここでは紹介を省くものとする。これらの違いは、commit の履歴をどのように結合するかであ

る。気になる場合はぜひ各自で調べて欲しい。

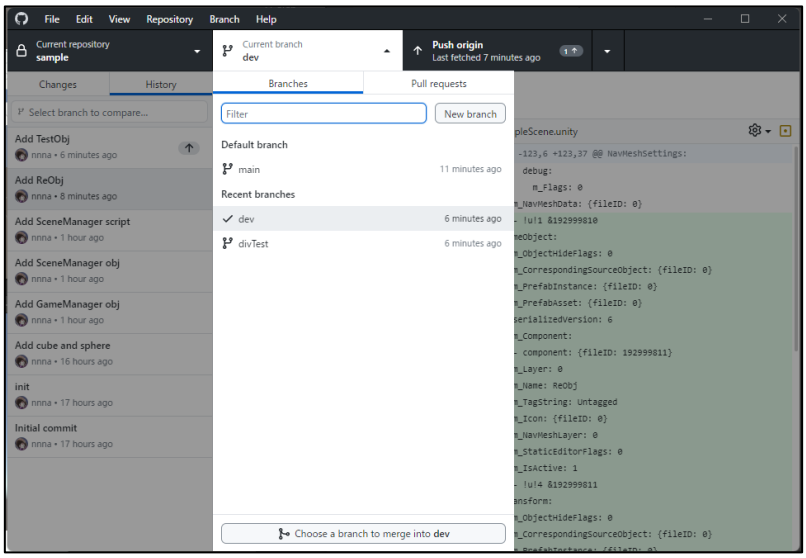

図 15: GitHub Desktop の branch 選択画面

また、リモートリポジトリで直接ブランチを切るには、GitHub の web サイトのファイルが表示されている上にある”Branches”をクリックして遷移した画面で図 16 のように”New branch”を選択して、New branch name に適当な名前を入力してから分岐元となるブランチを選び、”Create new branch”を選択すれば出来る。

ただし、分岐元となるブランチの最新の状態からブランチが切られるという点に注意する。

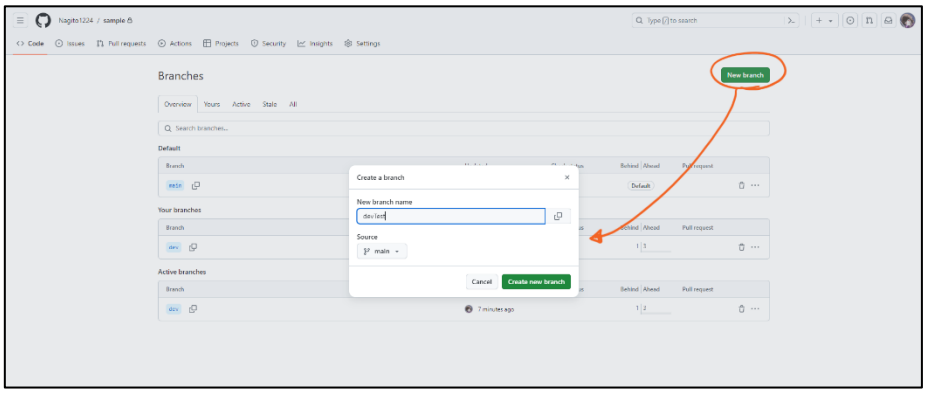

図 16: GitHub の branch 作成画面

GitHub で merge 要求(pull request)を送るには、自身が管理しているブランチに変更があった場合に GitHub に表示される”Compare & pull request”を選択して、適当なコメントを書いてリクエストを送ることで出来る。ここで注意して欲しいのは、pull request を送る対象となるブランチが適当であるかということである。pull request 作成画面を図 17 に示す。

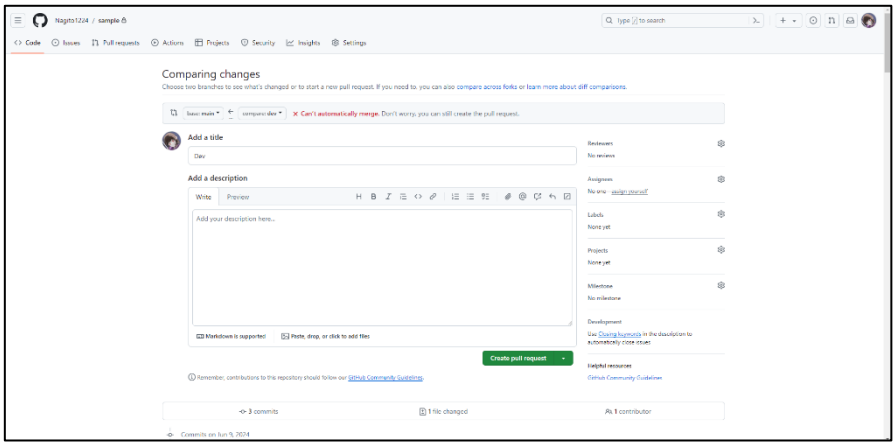

図 17: GitHub の pull request 作成画面

pull request が送られたブランチは、それを承認する必要がある。

もし変更が必要な場合などは、図 18 の青色で示した Add a comment からコメントを残すことが出来る。

ここで競合が発生した場合、図 18 の橙色で示した”Resolve conflicts”などから解消する必要がある。

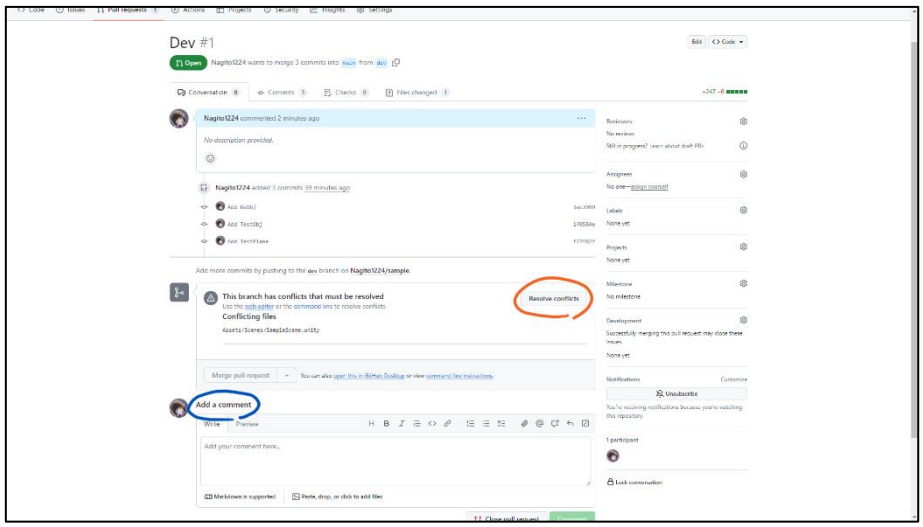

図 18: GitHub の pull request 応答画面(競合あり)

競合が発生した箇所は図 19 のように黄色で表示される。この内どちらを採用するか決めて、採用されなかった方を適切に消去する必要がある。

このとき、”<<<<<<< A”、”=======”、”>>>>>>> main”なども消去することに注意する。

解消を確認したら、画面上部の”mark as resolve”を選択した後、”Commit merge”を選択する。

```
147       m_TagString: Untagged
148       m_Icon: {fileID: 0}
149       m_NavMeshLayer: 0
150       m_StaticEditorFlags: 0
151       m_IsActive: 1
152     <<<<<<< dev
153     --- !u!4 &192999811
154     =======
155     --- !u!4 &318322152
156     >>>>>>> main
157   Transform:
158     m_ObjectHideFlags: 0
159     m_CorrespondingSourceObject: {fileID: 0}
160     m_PrefabInstance: {fileID: 0}
161     m_PrefabAsset: {fileID: 0}
```

図 19: GitHub の競合解消画面

すると、pull request 応答画面が図 20 のようになるため、”Merge pull request”を選択して、適切なコメントを残して merge する。

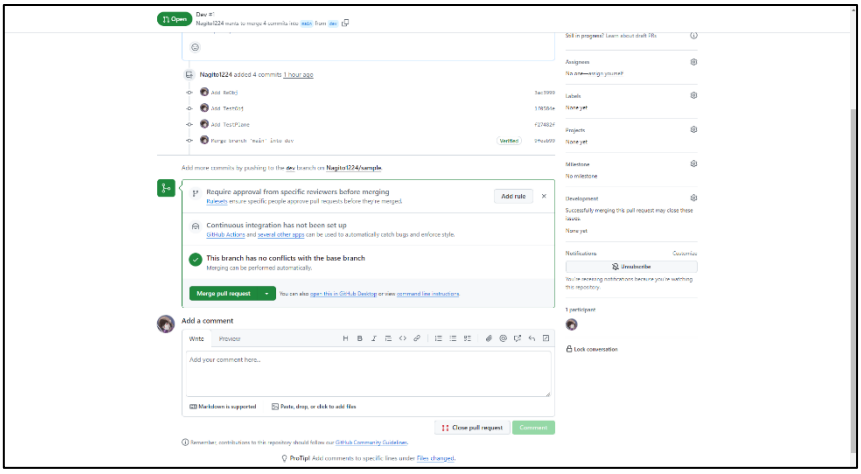

図 20: GitHub の pull request 応答画面(競合なし)

10 最後に

Git に関係するあとがきとしては、自分がどのブランチ、バージョンで作業しているのかは意識するように、ということだと思う。

さて、ではここからはこの文書の作成者として、また 2024 年度の代表としての話をここでしようと思う。

この文書を作成したのは、本サークルに Git を流行らせるというか、使うのが当然のシステムといった具合で、今後使っていって欲しいからなのだけれど、まあそこまで強制するつもりも、矯正するつもりもない。

他に使いやすいシステムがあればそれを使えば良いと思っている。ただ、Git は使えると

便利だということは、個人的な感想として述べておく。

また、Git、GitHub などは他にも機能はある。例えば、web ページを公開する機能。これは”Setting”から”Pages”に移動すれば簡単に行えるし、2024 年に公開する G^2 公式の web サイトではこれを使う予定である。

そもそも、この文書ではかなりの内容を省いている。最初に概説すると宣言したので、まあその通りに書いている。

ただし、ここで学んだ内容は、他の機能を調べるときやさらに学習を進める際に役に立つだろう。

まあともかく、G^2 というサークル内で使えるくらいの基本操作は覚えてくれると良いと思う。