ソース管理・ブランチ運用マニュアル

Project： ○○

Rev 1.00　2018/12/1

×× Inc.

変更履歴

Rev	日付	作成者	変更内容	備考
1.00	2018/12/1	△△	新規作成	

目次

1. 基本方針

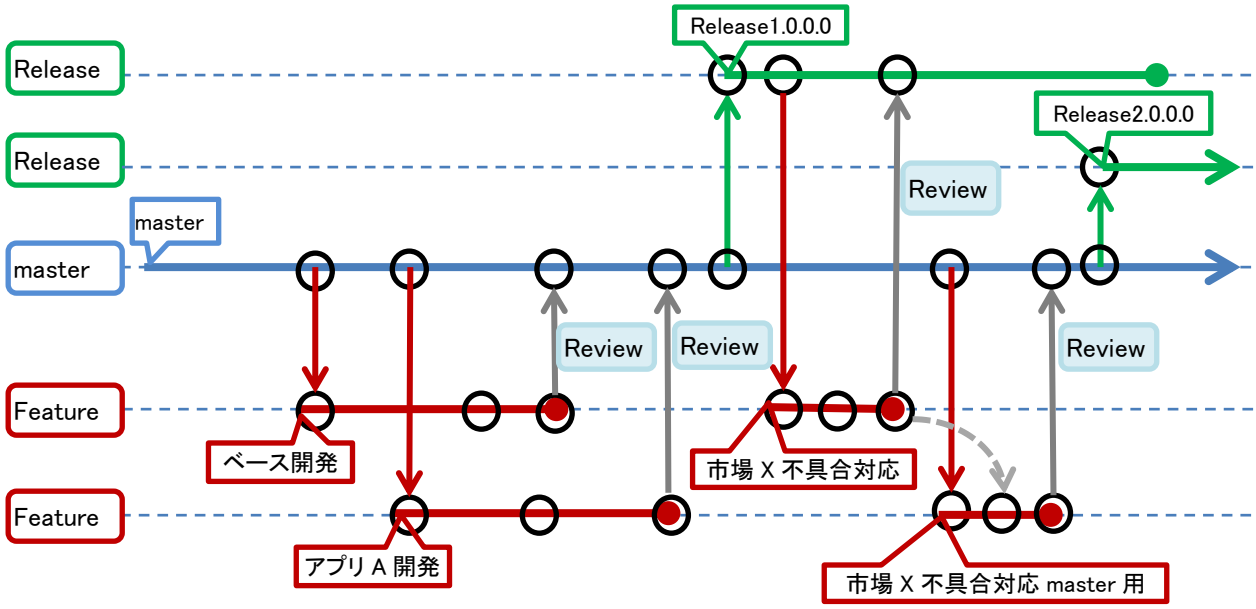
1.1. はじめに

本書は、製品開発プロジェクトにおける構成管理ツール「git」および「GitLab」を利用したブランチ運用について、基本的な知識を得ること、および運用フローを明確にすることを目的とします

1.2. ブランチ構成・目的

運用が軽量で必要十分であるとされている GitHub-Flow のブランチ戦略に準じつつ、組込み製品開発の特性(*1)に合わせカスタマイズしてブランチの種類と生存期間、利用方法を規定します

(*1)Web 系開発では master がそのまま製品としてデプロイされるため、ユーザの目に触れるブランチは master です。それに対して、組込み開発では製品コードを長期保守する release ブランチを必要とするため、ユーザの目に触れるブランチは release となります。このことにより組込み製品開発では master の安定度を Web 系と比べて低いものとしてブランチ設計します



補足 【ブランチ図のルール】: 中央に master ブランチを配置し、より安定しているブランチを上、より不安定なブランチを下に描くことでブランチの安定度を視覚的に表現します

種別: Release （ブランチ名は「Release1.0.0.0」「Release1.0.1.4」「Release2.0.0.0」など）
・市場に出ているブランチ。総合デバッグへの投入タイミングでブランチ生成し、市場から消えるタイミングで削除する
・緊急性のある不具合のみ commit する

種別: master （ブランチ名は「master」）
・開発者全体で共有する永続的なブランチ。最新であり、かつ、どの時点の commit からでもビルドが通る
・コードレビュー済み、かつ、ビルドが通るコードのみコミットできる
・最新の commit からいつでも Release や Feature ブランチを生成できる

種別: Feature （ブランチ名は「ベース開発」「アプリ A 開発」「市場 X 不具合対応」など）
・各担当者が各案件の実装作業を行うブランチ。必要なだけ複数存在できる。作業開始のタイミングでブランチ生成し、上位ブランチにマージ commit した時点で削除する
・上位ブランチへのマージ commit はコードレビューを行い許可が出た場合のみ行うことができる
・Feature ブランチへは他の Feature ブランチのマージ commit も行うことができる。例えば案件が大きな場合に、複数の開発者に分割していた作業を、統合してから master へマージ commit することができる

2. 運用ルール

ブランチの構成は基本方針の通りです。重複しますが、各ブランチ上での開発の運用ルールについて以下に補足します

2.1. 運用ルール詳細

【ルール 1】 master ブランチは常に共有・リリース可能な状態を保つ

全ての作業を master ブランチへマージするようにすることで、ブランチモデルを簡単にしています。しかしながら、ブランチに不具合が混入していた場合、他の開発者の生産性に影響を与えます。master ブランチへマージするブランチは、テスト、レビューされ、master ブランチは常に安定して動作するように保つ必要があります。

【ルール 2】 新機能の実装は master ブランチから作成する

通常の開発用ブランチは master ブランチから作成し、成果を master ブランチにマージします。

【ルール 3】 緊急のバグフィックスは Release ブランチから作成する

緊急のバグフィックス用のブランチは Release ブランチから作成し、成果を Release ブランチにマージします。その後、必要に応じて master ブランチへもマージします。

【ルール 4】 ローカルの Feature ブランチを定期的にプッシュする

作業用の Feature ブランチは、なるべく早い時期に開発者全員で共有するために、頻繁に中央リポジトリにプッシュします。担当者の端末のハードディスクの万が一の故障からコードの喪失を防ぐ効果もあります。

【ルール 5】 マージリクエストを利用してレビューを行う

master ブランチの安定度を高めるために重要なのは、コードレビューです。GitLab のマージリクエストの機能を利用して、ソースコードのレビューを行い、master ブランチにマージします。(GitLab を利用しない場合でも、上位リポジトリにマージするときにはコードレビューを行わなければなりません)

【ルール 6】 master にマージされたマージリクエストは、直ちに全体ビルド・実機テストする

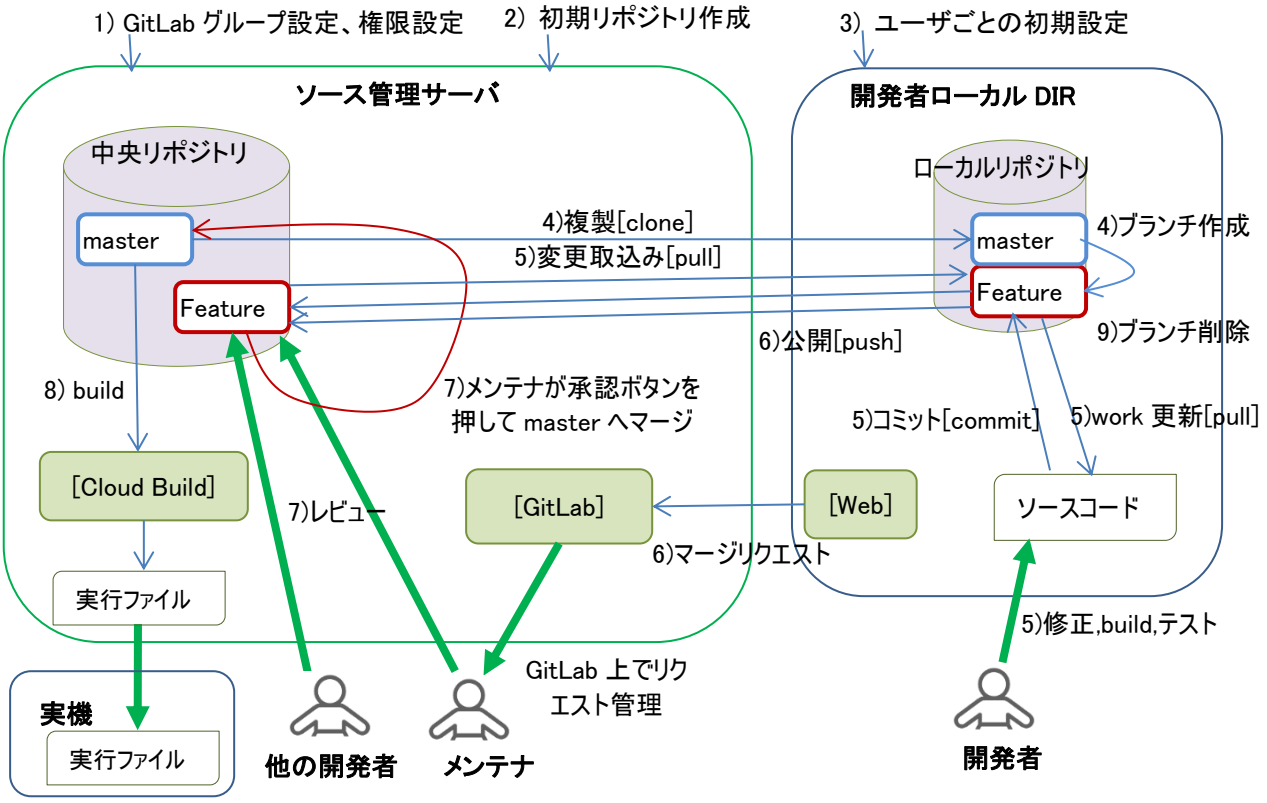
Feature ブランチがマージされた master ブランチのコードは、すぐに全体ビルドし実機にてテストします。バグの修正は、バグが混入した時点より発見が遅れれば遅れるほど、時間がかかり、また、発見も困難になります。早期にビルドし動作確認を行うことにより、バグを早期に発見し、修正コストも抑えることができます。

3. 手順

運用の手順とフローを説明します

3.1. 概要フロー

下記の概要フローに従い運用します



準備	1	GitLab にグループ設定
		GitLab にユーザ登録、各ユーザの権限設定
	2	GitLab に project・初期リポジトリ作成
	3	ユーザごとの初期設定
開発サイクル中	4	開発者は master ブランチを複製(clone)した後、作業用ブランチをローカルリポジトリ上で作成
	5	開発者は作業用ブランチ上に変更をコミットし、テスト。 中央リポジトリが更新された時には変更取り込み
	6	開発者は作業用ブランチを master ブランチに反映してもらうために、中央リポジトリに公開(push)してマージリクエストを発行
	7	メンテナ、他の開発者はマージリクエストに対して差分のコードレビュー。良好ならばメンテナの権限で master へマージ
	8	master ブランチへのマージで、Cloud Build 等で自動的に master ブランチの実行ファイルを作成。 →これを必要に応じてダウンロードして動作確認
	9	開発者は作業用ブランチを削除し 1 サイクルを終える

3.2. コマンド操作・GUI 操作

ここでは、概要フローに従って作業する上で必要最低限な操作に絞って説明します。
Git 自体の使い方については別途 Web 等で調査し学習してください。

3.2.1. 使用ツール

操作は以下のツールを使用した場合について記述しています。他のツールを利用する場合は、【コマンドの場合】の処理内容が充足できるよう適宜読み替えてください。

- **GitLab = GitLab.com [Bronze Plan]**
- **GUI ツール = TortoiseGit version 2.7.0 (OS=Windows)**
- **コマンド = git version 2.15 (OS=Linux)**

3.2.2. 準備

	1	GitLab にグループ設定
		GitLab にユーザ登録、各ユーザの権限設定
		GitLab に project・初期リポジトリ作成
	2	ユーザごとの初期設定

1. グループ・ユーザ設定、project・初期リポジトリの作成

プロジェクト所有者(Owner)が行います。GitLab におけるアクセス管理は、グループ、プロジェクト、ブランチ、ユーザの位置づけについての理解が必要です。これら定義と操作手順について必要であれば補足を参照してください。

2. ユーザごとの初期設定

メンテナ(Maintainer)、開発者(Developer)等、リポジトリを clone する必要があるユーザが各自で行います。
補足「ユーザごとの初期設定」を参照してください。

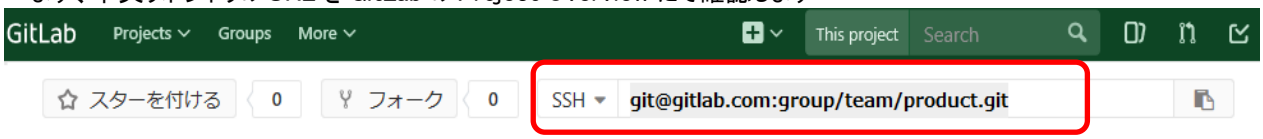
3.2.3. 開発サイクル中

	[1, 2]	開発者は master ブランチを複製(clone)した後、作業用ブランチをローカルリポジトリ上で作成

1. ローカルリポジトリ作成・設定

中央リポジトリをクローンしてローカルリポジトリを作成し、リポジトリ毎の設定をしてから、checkout します。
最初に1回のみ実行します

まず、中央リポジトリの URL を GitLab の Project Overview にて確認します



【コマンドの場合】

開発者ローカルマシン上でコマンド実行し、ローカルリポジトリを作成します(-n でチェックアウトは抑止)

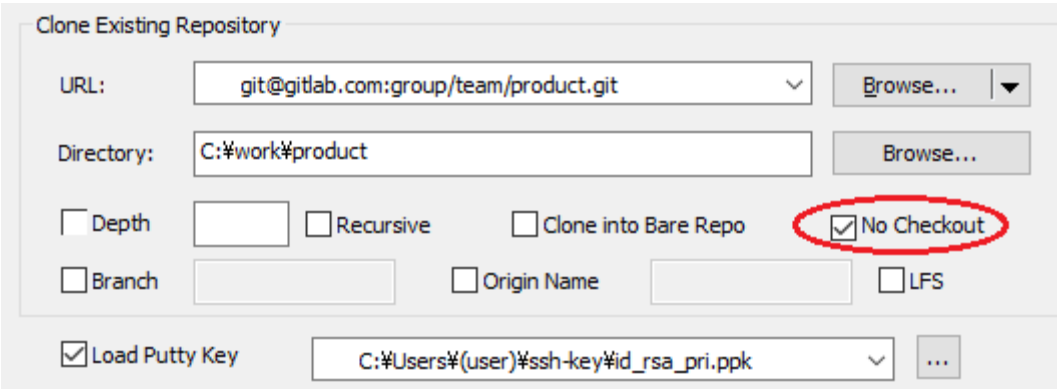
```
$ git clone -n git@gitlab.com:group/team/product.git
```

ローカルリポジトリ内でコマンド実行し、user 名とメールアドレスを登録します

```
$ cd product                    ← clone にて .git を除いた名前でディレクトリが生成されるので、移動します
$ git config --local user.name ユーザ名
$ git config --local user.email メールアドレス
$ git config --local core.autocrlf false      ← CRLF 自動変換は抑止します
```

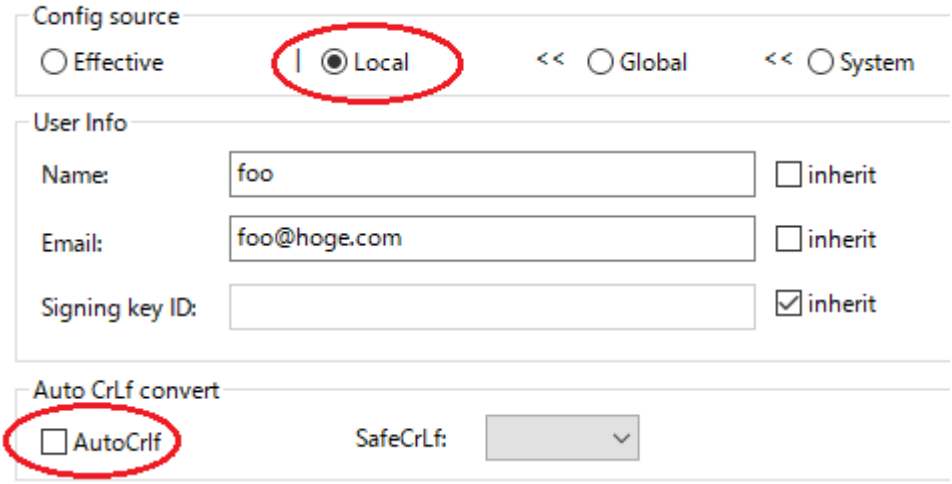
【GUI の場合】

開発者ローカルマシン上の任意のディレクトリで、右クリック → Git Clone… で TortoiseGit を起動します



URL = 中央リポジトリの URL を指定します
No Checkout = チェックアウトは抑止します
Load Putty Key = あらかじめ作成しておいた ppk ファイルを指定します
(ファイルの例: C:¥Users¥(user_name)¥ssh-key¥id_rsa_pri.ppk)
OK で、clone され、Directory:に指定した名前で作業ディレクトリが生成されます。

生成された作業ディレクトリに移動し、右クリック → TortoiseGit → settings を選択
左メニューの”Git” → Config source に local を選択します



Name, Email に user 名とメールアドレスを登録します
AutoCrlf のチェックは外します

<<ここまでで、ローカルリポジトリ作成は完了です。まだ作業ファイルは見えませんが、次項で行います>>

<<これ以降、clone で作成された作業ディレクトリへ移動し、ディレクトリ内での操作となります>>

2. 作業用ブランチ作成

まず、ローカルリポジトリを master にして中央リポジトリの master と同期します。
次に、ローカルリポジトリの master ブランチから作業用のブランチを作成します。ブランチ名は作業単位として識別しやすいものを付けてください。

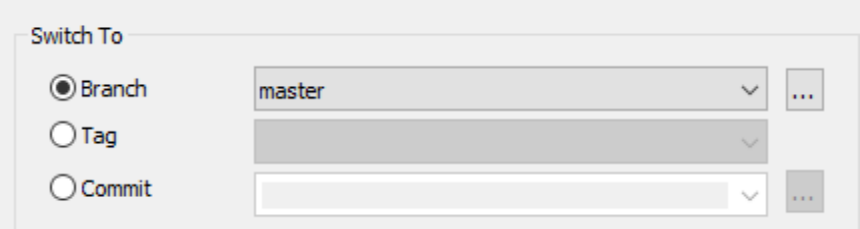
【コマンドの場合】

```
$ git branch                    ←今いるブランチを確認。ブランチのリストが表示される。*が今いるブランチ。
$ git checkout master           ←今いるブランチを master に切り替える（すでに master にいるならば不要）
$ git fetch --all -p
$ git pull --ff                 ←master に(中央リポジトリの)master の変更を取り込む

$ git checkout -b DEV-APP-AA    ←master ブランチから作業用のブランチを作成
```

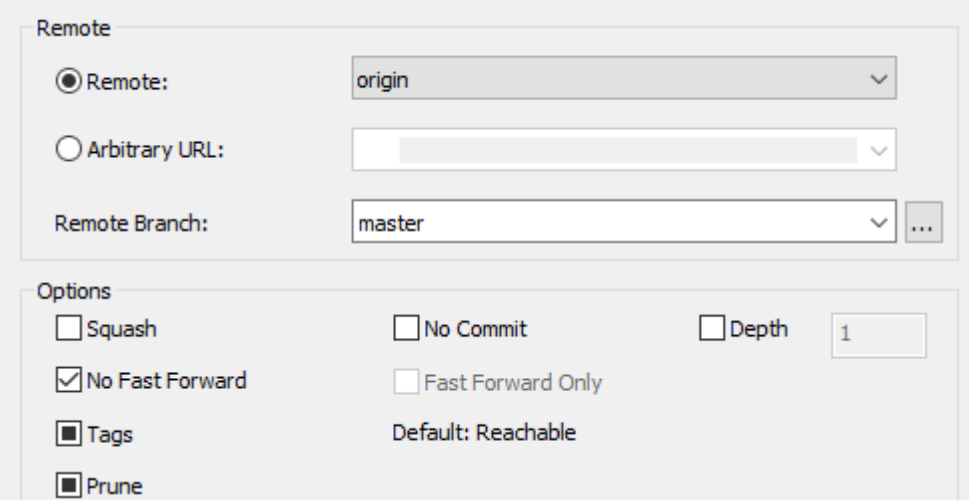
【GUI の場合】

2.1. 作業ディレクトリ内で右クリック→TortoiseGit→Switch/Checkout



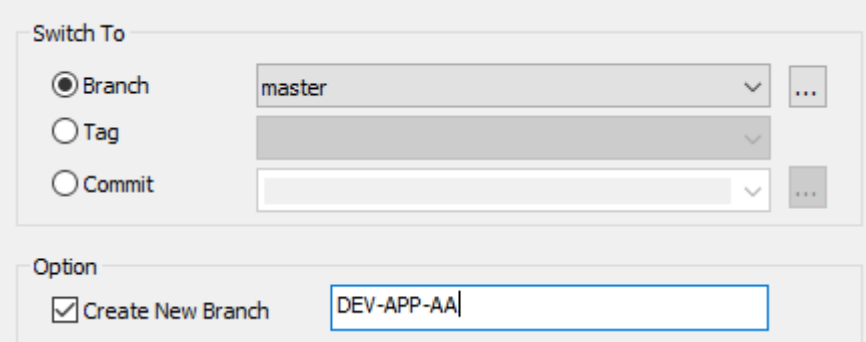
・branch に、master を指定して、OK で master に切替える

2.2. 作業ディレクトリ内で右クリック→TortoiseGit→Pull



・remote に、origin、Remote Branch に master を指定、Depth のチェックを外し、
・No Fast Forward のチェックを入れ、
OK でローカルの master に(中央リポジトリの)master の変更を取り込む

2.3. 作業ディレクトリ内で右クリック→TortoiseGit→Switch/Checkout



・branch に、master を指定
・Create New Branch に、これから作りたい作業用のブランチを指定
OK で master ブランチから作業用のブランチを作成する

	[3, 4]	開発者は作業用ブランチ上に変更をコミットし、テスト。 中央リポジトリが更新された時には変更取り込み

3. 作業ブランチの更新

中央リポジトリの master に入った変更を作業ブランチに取り込みます。 中央リポジトリの master に変更があったタイミングで速やかに行ってください。pull でコンフリクトしたときは適宜解消してください

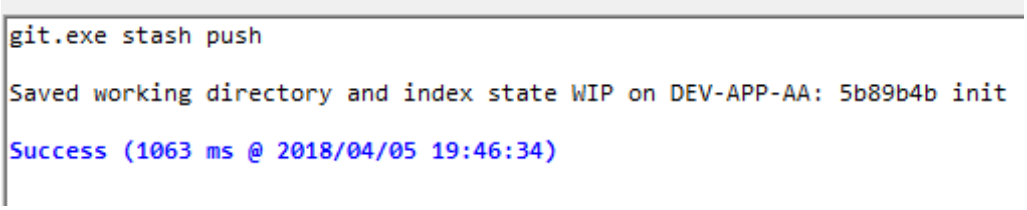
【コマンドの場合】

```
$ git fetch --all -p
$ git stash          ←作業中のファイルを一時退避
$ git pull origin master    ←作業ブランチに master の変更を取り込む
$ git stash pop      ←先ほど一時退避したファイルをマージする
```

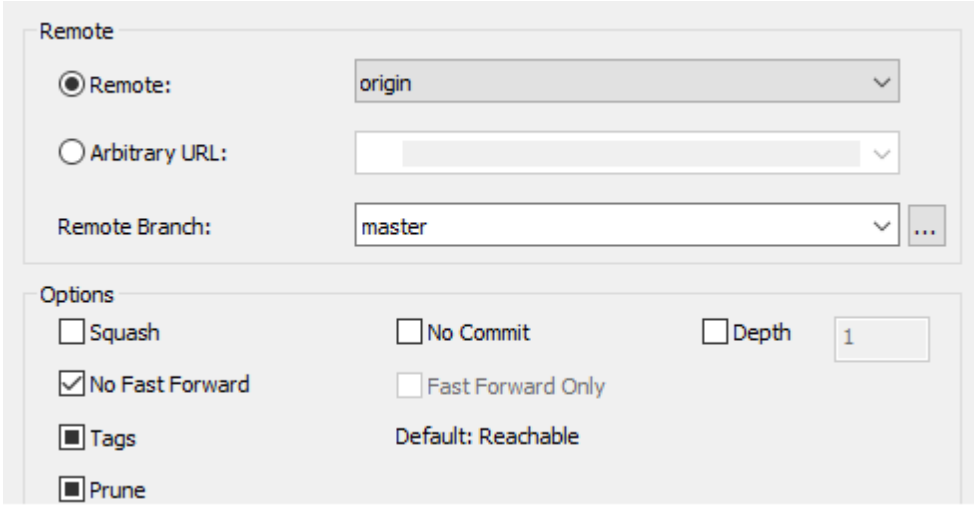
【GUI の場合】

3.1. 作業ディレクトリ内で右クリック→TortoiseGit→Stash changes

・include untracked、 --all のチェックを外し
OK で master ブランチから作業用のブランチを作成する



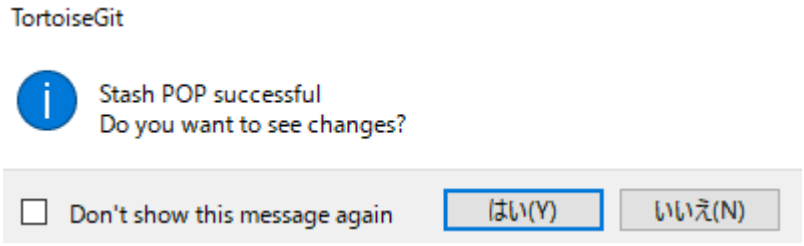
3.2. 作業ディレクトリ内で右クリック→TortoiseGit→Pull



・remote に、origin、**Remote Branch を master に変更**して、Depth のチェックを外し、
・No Fast Forward のチェックを入れ、
OK でローカルの作業用のブランチ”DEV-APP-AA”に(中央リポジトリの)master の変更を取り込む

3.3. 作業ディレクトリ内で右クリック→TortoiseGit→Stash Pop

※先ほど一時退避したファイルがマージされる。 以下の画面となれば完了



4. 作業ブランチでファイル作成、修正、ローカルリポジトリへのコミット

作業ブランチでの作業をローカルリポジトリへコミットします。(パーミッションも変更管理されますので適切なパーミッションになっているか確認してから add してください。)

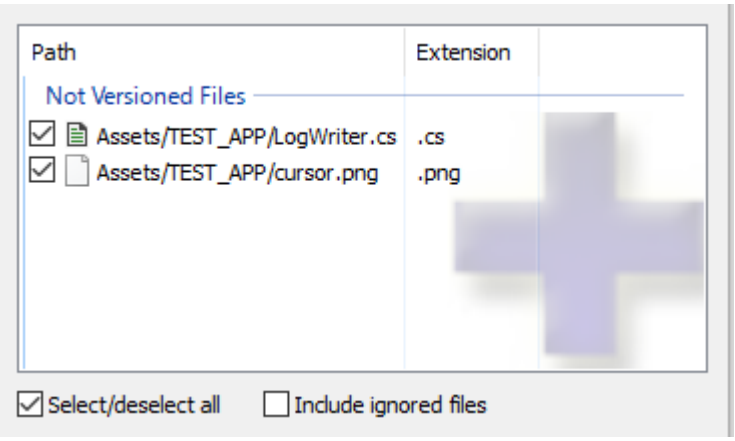
【注意】 コミットログは修正履歴等の意味のある内容にしてください
ソースの中に修正履歴を書くのは避け、コミットログに修正履歴を書くようにしてください

【コマンドの場合】

```
$ git add ファイル名      ←作成したファイルを追加
$ git add -u              ←一度 commit したファイルはこのコマンドでも OK
$ git commit -v
```

【GUI の場合】

4.1. 作業ディレクトリ内で右クリック→TortoiseGit→Add



候補のファイル一覧が表示されるので、コミットしたいファイルにチェックを付けて OK

4.2. 作業ディレクトリ内で右クリック→TortoiseGit→”Git Commit->”ブランチ名”を選択
Message: にコミットログを書いて Commit ボタン押下

	[5, 6]	開発者は作業用ブランチを master ブランチに反映してもらうために、中央リポジトリに公開(push)してマージリクエストを発行

5. 作業ブランチのプッシュ

作業ブランチを中央リポジトリ(の作業ブランチ)へプッシュします。これにより、他の開発者と作業内容を共有するとともに、担当者の端末のハードディスクの万が一の故障からコードの喪失を防ぎます。

この作業はマージリクエストを発行するまでは何度行っても構いません。むしろ、この作業ブランチはビルドが通らないレベルでも良いので頻繁にプッシュして個人ブランチとして有効に活用してください。

【注意】pull でコンフリクトすることもあるかもしれませんが、中央への push で
コンフリクトすると最悪のマナー、生産性となります。pull した時点でコンフリクトさせましょう

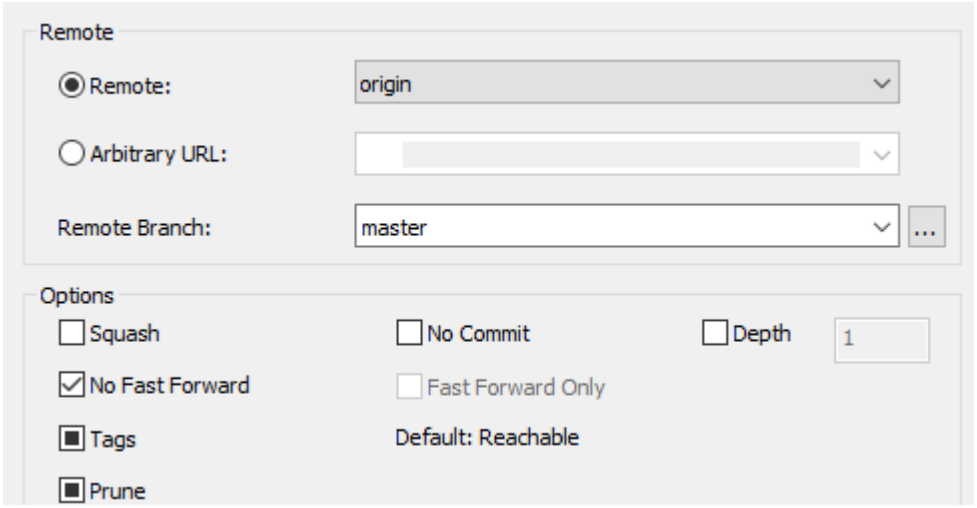
【コマンドの場合】

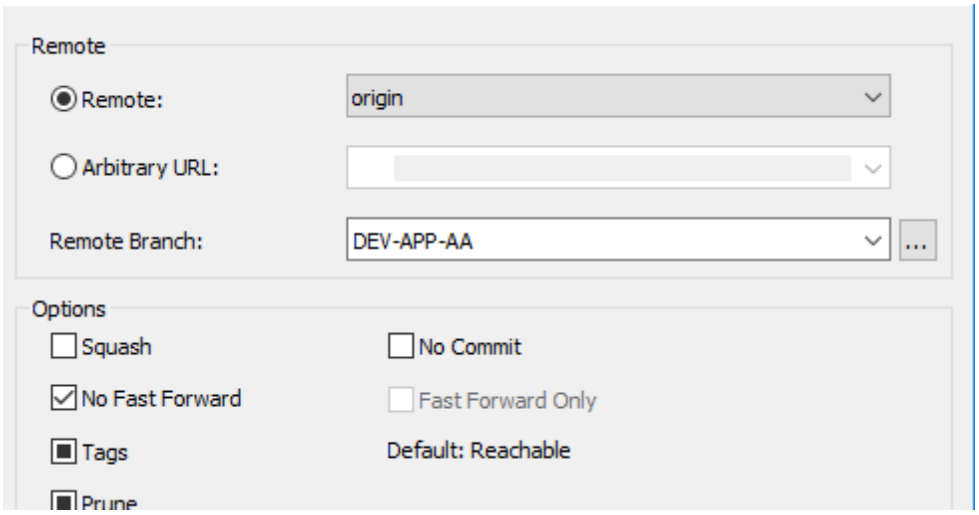
```
$ git fetch --all -p
$ git pull origin master        ←作業ブランチに master の変更を取り込む
$ git pull
$ git push origin DEV-APP-AA
```

【GUI の場合】

5.1. 作業ディレクトリ内で右クリック→TortoiseGit→Pull

初めに、ローカルの作業用のブランチ”DEV-APP-AA”に(中央リポジトリの)master の変更を取り込み、
続いて、ローカルの作業用のブランチ”DEV-APP-AA”に(中央リポジトリの) DEV-APP-AA の変更を取り込む

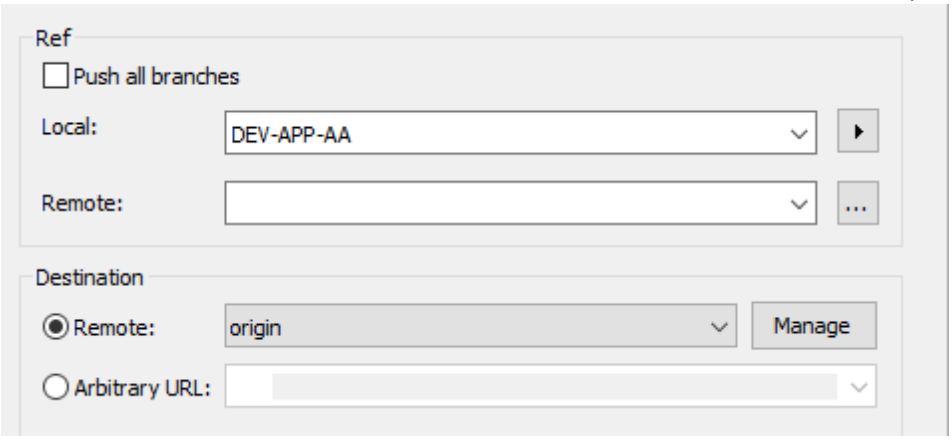




・No Fast Forward のチェックを入れ、OK

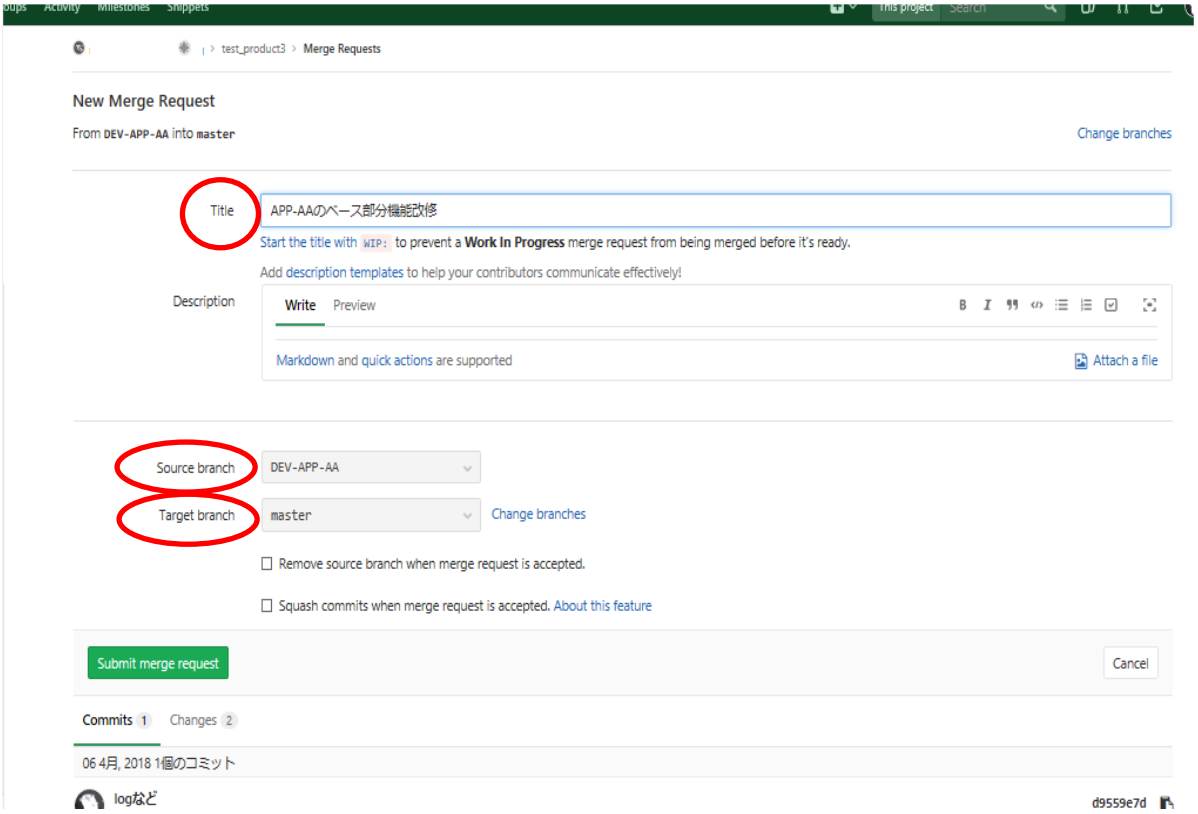
5.2. 作業ディレクトリ内で右クリック→TortoiseGit→Push

ローカルの作業用のブランチ”DEV-APP-AA”を(中央リポジトリの) DEV-APP-AA へ push



6. **マージリクエスト**
作業ブランチでの作業を完了したら、今まで作業ブランチにコミットして来た複数のコミットを中央リポジトリの mster ブランチへマージしてもらうため、マージリクエストを作成します。

中央リポジトリの作業ブランチから中央リポジトリの master ブランチへの指定でマージリクエストを作成します。
「タイトル」＝タイトルを入力
「Source branch」＝作業ブランチが自動設定されます
「Target branch」＝master が自動設定されます



「Submit merge request」を押下

	[7, 8, 9]	メンテナ、他の開発者はマージリクエストに対して差分のコードレビュー。良好ならばメンテナの権限で master へマージ

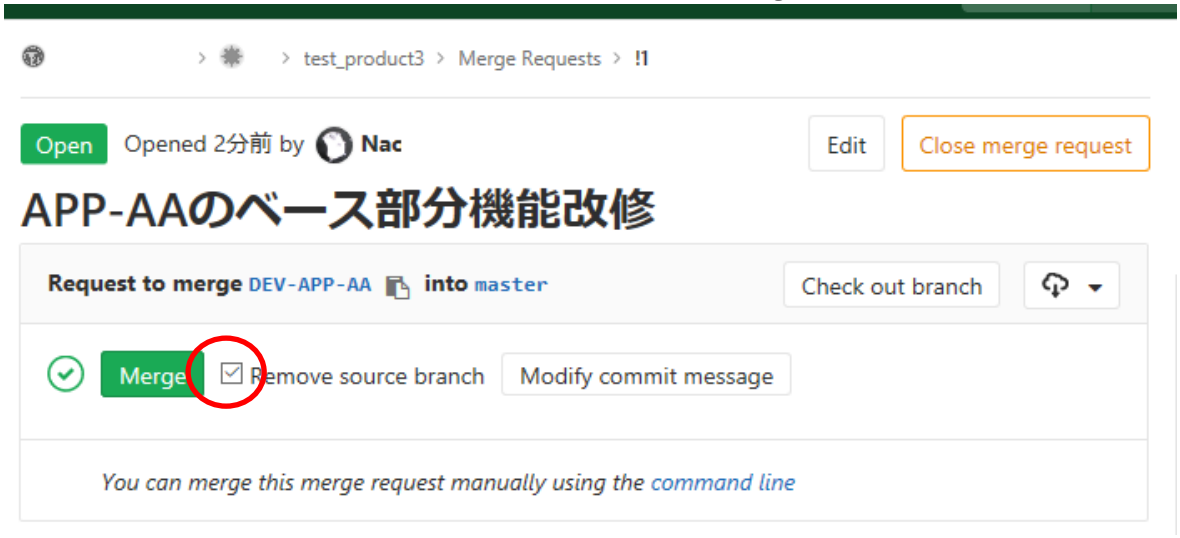
7. **レビュー**
レビューアーはマージリクエストをレビューします。GitLab を利用した場合、マージリクエストによる master ブランチとの差分を Web 画面から確認することもできます
また、Web 画面のソースの任意の場所にレビューコメントを追加してわかりやすく開発者に伝えることができます

8. **レビュー指摘事項修正**
マージリクエストを作成した担当者は、レビューアーのコメントを受け、修正した変更を中央リポジトリへプッシュします。
マージリクエストを送った作業ブランチを修正し、そのままプッシュするだけです。特別な作業は必要ありません。

修正したプッシュは、自動的にマージリクエストに反映されます。
追加のコミットもマージリクエスト上で管理されるので、マージリクエストの画面を見れば、誰がどのようなコメントを行い、どのような修正がなされたか確認できます。

9. マージ

メンテナはレビュー承認したら、作業用ブランチを master ブランチへマージし、マージリクエストをクローズします。
作業としては、
Web 画面で、[Remove source branch]にチェックを入れ、マージ承認[Merge]ボタンを押します



これで、中央リポジトリにおいて、master に作業用ブランチの内容が反映され、作業用ブランチが削除されます。
(※同じ作業単位で作業する場合でも、一貫して「作業用のブランチを削除→同名だけど作成する」の流れを推奨します。)

	[10]	master ブランチへのマージで、Cloud Build 等で自動的に master ブランチの実行ファイルを作成。 →これを必要に応じてダウンロードして動作確認

10. master ブランチの動作確認

Cloud Build サーバで exe が自動生成されます。必要に応じてサーバからダウンロードしてテストします。

	[11, 12]	開発者は作業用ブランチを削除し 1 サイクルを終える

11. ローカルリポジトリのメンテナンス

ローカルリポジトリを master に戻して中央リポジトリの master と同期します。(ここでは、Fast Forward が良いでしょう)
また、中央リポジトリ上で作業用ブランチが削除されましたので、ローカルリポジトリも同様に削除します。

【コマンドの場合】

```
$ git checkout master
```
←今いるブランチを master に切り替える

切り替えのタイミングで以下のようなメッセージが出ることがありますが、**(以下の*2)**で同期します
「このブランチは 'origin/master' に比べて n コミット遅れています。fast-forward することができます」

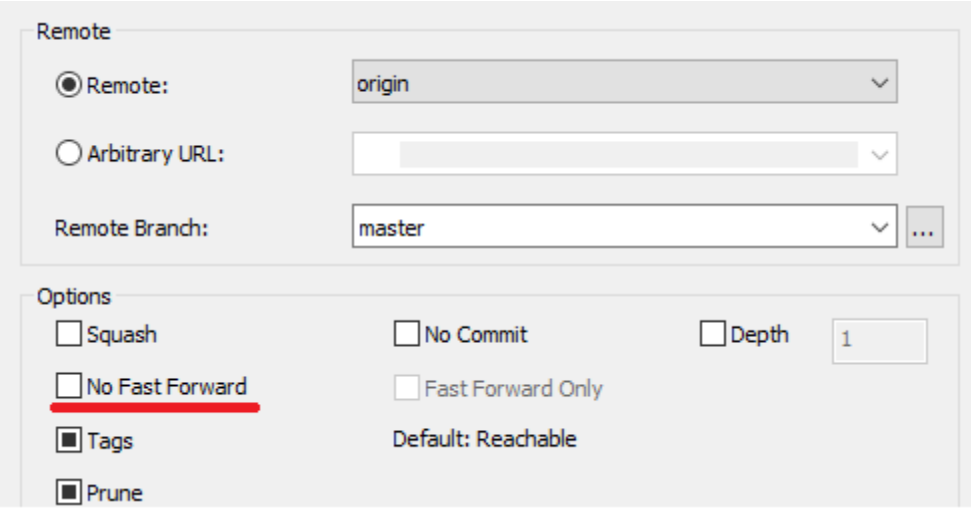
```
$ git fetch --all -p
$ git pull --ff
$ git branch -d DEV-APP-AA
```
←master に(中央リポジトリの)master の変更を取り込む**(*2)**
←作業ブランチはこのタイミングで削除

【GUI の場合】

11.1. 作業ディレクトリ内で右クリック→TortoiseGit→Switch/Checkout

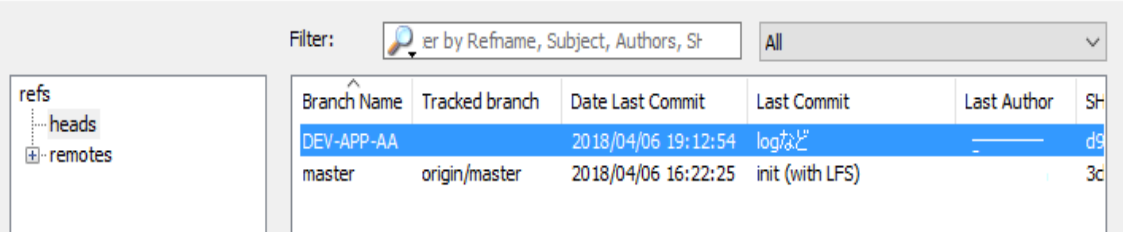
・branch に、master を指定して、OK で master に切替える

11.2. 作業ディレクトリ内で右クリック→TortoiseGit→Pull



・remote に、origin、Remote Branch に master を指定、Depth のチェックを外す
・No Fast Forward のチェックは入れない
OK でローカルの master に(中央リポジトリの)master の変更を取り込む

11.3. 作業ディレクトリ内で右クリック→TortoiseGit→Browse references



・ローカルの作業用のブランチ”DEV-APP-AA”を右クリック→Delete branch
で削除する

12. 次のサイクルへ

2 へ戻り、新たに master ブランチから作業用のブランチを作成し、次の仕事を始めます。
(※同じ作業単位で作業する場合でも、一貫して「作業用のブランチを削除→同名だけど作成する」の流れを推奨します。)

3.3. トラブル回避のためのコツ

Git で共同作業をして行く上で、最初の内は他の開発者に迷惑をかけないかと不安になることもあると思います。
以下のわずかな追加事項に留意して作業することで無用なトラブルを回避してください。

※ Push する直前には必ず Pull する
※ 履歴の削除・改変(Rebase)を行わない

3.3.1. Push する直前には必ず Pull する

運用手順でも Push する前に Pull する手順になっています。
コンフリクトが発生するような修正を行うことは日常的なものですが、このコンフリクトを自分のローカルリポジトリ内で発生させれば他の開発者に影響を与えません。
自分のローカルリポジトリ内で落ち着いてゆっくりとコンフリクトの解消作業を行えば良いのです。
そして、Pull することでコンフリクト発生の可能性を低めたコードを、中央リポジトリへ Push するようにしてください。

3.3.2. 履歴の削除・改変(Rebase)を行わない

Rebase の是非については意見が分かれるところですが、無用なトラブルを避けるため基本的には rebase は行わないようにしてください。

rebase を行うことには運用上のデメリット(Web 等で調査してください)が多く、対するメリットは、「ソースコードの変更履歴の内、個人的に不要だと感じたものを消し、見易い履歴になる」ことだけです。
rebase はこのメリットを実現する手段としてリポジトリへの入力時の操作で行っていますが、現在では別の手段としてリポジトリからの出力時に行うことが、GitHub や GitLab 等のサービスのネットワークグラフ機能で可能になっています。（master へのマージのタイミングの履歴だけを見る等）
このように、ネットワークグラフ機能が利用できる場合は、rebase を行う運用には合理性が無いと判断されます。

【ネットワークグラフの利用例】
左メニューの”Repository” → ネットワークグラフで表示する。
個人的な途中経過の履歴等も記録されているが、rebase で無理に削除する必要は無く、
確認したい内容が master へマージされた差分だけの場合は、A と B とを指定して出力時に読み飛ばすことができる。

4. 補足

4.1. Git コマンド

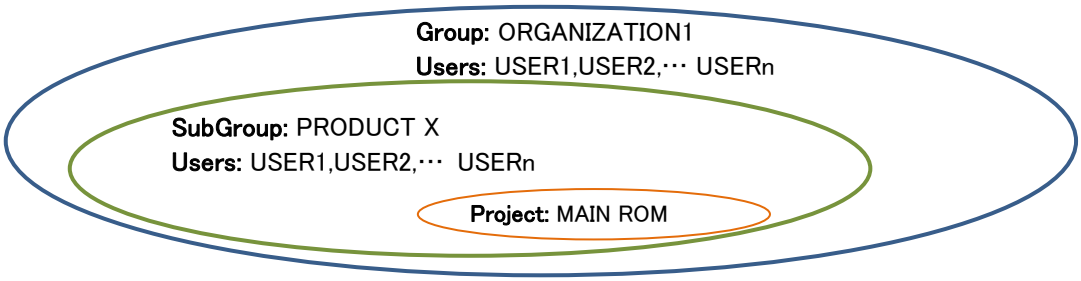
ローカルリポジトリで Git の便利なコマンドを使いたい場合や、マージ時にコンフリクトしたときなど、Git 自体の使い方については web を参照してください。

4.2. GitLab でのアクセス管理について

グループ、サブグループ、プロジェクト、ブランチへの push 制限、ユーザ種別 でアクセス管理します。

4.2.1. グループ、サブグループ、プロジェクト、ユーザ

・グループ、サブグループ、プロジェクト、ユーザの位置づけ



ソースコードはプロジェクト内に置かれ、プロジェクトはグループに属するものとして登録されます。
グループはサブグループを作成することで階層構造とすることができます。
※ ユーザはグループやサブグループ毎に登録し、「メンテナ(Maintainer)」「開発者(Developer)」等の権限を付与します。
この時、サブグループにて権限の上書きを行うことができます。
基本的には、サブグループでユーザがアクセスできる範囲を絞ることになります。
※ プロジェクトはグループ、又は、サブグループ内に登録することでアクセスできるユーザを管理しています。プロジェクトに直接ユーザを登録することは行わないでおきます。

4.2.2. 権限

プロジェクトは中央リポジトリに 3 種(Release/master/Feature)のブランチを持ちます。
ブランチの種類に応じて、「マージ承認ボタン」の表示制限と、直接 push の制限をかけます。

・下記の通りユーザ権限によって重要なブランチへの「マージ承認ボタン」の表示を制限し、開発者単独での不用意なマージを防止します

ユーザ種別	Release ブランチ	master ブランチ	Feature ブランチ
開発者（Developer）	マージ制限 (protected)	マージ制限 (protected)	マージ許可
メンテナ（Maintainer）	マージ許可	マージ許可	マージ許可

・下記の通り重要なブランチへの直接 push を制限します

ユーザ種別	Release ブランチ	master ブランチ	Feature ブランチ
開発者（Developer）	push 制限	push 制限	push 許可
メンテナ（Maintainer）	push 制限	push 制限	push 許可

＊プロジェクトを作成後、プロジェクトに存在するブランチへ上記方針に従い制限設定をします

4.3. グループごとの初期設定

まずは、グループ全体に関わる作業を行う担当者を決めて、それに従ってユーザを登録します。

1) **オーナ(Owner)を決める**
2) **メンテナ(Maintainer)権限を持つ人を決める**
3) **GitLab に、グループ、プロジェクトを登録する**
4) **GitLab に、ユーザを登録する**
5) **プロジェクトにリポジトリを作成する**

1) オーナ(Owner)を決める

オーナ(Owner)は何でもできてしまう権限を持った人です。
ユーザ登録、プロジェクト登録、ブランチへの push 制限付与など全体に関わる作業は Owner が行います。
【仕事内容】
*GitLab にプロジェクト(yyy, zzz など)を登録／削除すること
*GitLab にアクセスできるユーザを登録／削除すること

2) メンテナ(Maintainer)権限を持つ人を決める

・メンテナ(Maintainer)権限を持つ人をチームで話し合って決めてください。
→ソースコードレビューは全開発者が誰でも行うことができますが、master, Release* 等の特別なブランチへソースコードをマージできる人はメンテナ(Maintainer)のみになります。

※メンテナとしての適任者を複数人用意しておき、メンテナがボトルネックとなることがないようにしましょう。
→ソースコードの読み書きができ、かつ、例えば以下のような特性を持つ人が適任でしょう。
・チーム古参
・プロダクト全体の理解度が高い
・自分の担当部分だけではなく全体のソースコードの品質を気にかけている

3) GitLab に、グループ、プロジェクトを登録する

グループを開発組織名で作成し、サブグループを製品名で作成、プロジェクトを適宜作成することにします。
（補足：グループは、GitLab.com のユーザ数に応じたライセンス費支払いの単位にもなっています）

1. グループを作る

グループ名（開発組織名）を決めてグループを登録し、以下の設定を行います

開発製品のソースコードを GitLab.com 上で外部に漏らさないために、グループの可視設定をプライベートにします
（※グループ全体に適用されますので、グループ配下のプロジェクトは全てプライベートになります）
[MENU] グループ名→Settings→General
→Visibility Level ←プライベートに設定
→Share with group lock ←□Prevent sharing … チェックする

また、ユーザ数に応じてグループ単位で支払いを行う都合上、グループ内のユーザを固定しておきます
（※ユーザを新規に追加したいときには、一時的に固定を解除して作業します）
[MENU] グループ名→Settings→General→Member lock ←□Prevent adding… チェックする

2. サブグループを作る

サブグループ名（製品名）を決めてサブグループを登録します
（後ほど、サブグループにユーザを登録していきます。このサブグループでユーザがアクセスできるプロジェクトを絞ることになります）

3. プロジェクトを作る

プロジェクト名を決めてプロジェクトを登録します

4) GitLab にユーザを登録する

Owner は、GitLab に、参加者のユーザ登録をしてください。

1. ログインするためのユーザ名を適切に決定
   （GitLab.com 使用時は、ユーザ登録用にメールアドレスも決める必要があります）
2. 特定の製品の開発にアクセスできる人を、製品のサブグループに「開発者:Developer」として登録
3. どの製品にもアクセスできる人は、グループ(ORG1)に「開発者:Developer」として登録
4. 製品ごとに決めたメンテナとなる人をサブグループ(yyy, zzz など)に「メンテナ:Maintainer」として登録

【例】（下表のように登録することにより、網掛け部分の権限が付きます）

名前	グループ（ORG1）	サブグループ(yyy)		サブグループ(zzz)	
		Maintainer	Developer	Maintainer	Developer
A さん （ORG1 全てに参加、yyy のメンテナ）	A	A	–	–	–
B さん （ORG1 全てに参加、zzz のメンテナ）	B	–	–	B	–
C さん （ORG1 全てに参加、開発者）	C	–	–	–	–
D さん （yyy のみに参加、メンテナ）	–	D	–	–	–
E さん （yyy のみに参加、開発者）	–	–	E	–	–
F さん （zzz のみに参加、メンテナ）	–	–	–	F	–
G さん （zzz のみに参加、開発者）	–	–	–	–	G

5) プロジェクトにリポジトリを作成する

プロジェクト作成直後は、リポジトリが存在していません。

1. 最初のソースコードを master ブランチとして push することでリポジトリを作成します。

   リポジトリには、以下の設定ファイルを含めます。
   - .gitignore → 管理対象外ファイルを指定します
   - .gitattributes → Git-LFS の対象ファイルを指定します

2. master ブランチ作成後、以下のように master, Release* 等の特別なブランチへの操作権限を設定します

   [MENU] プロジェクト名→Settings→Repository→Protected Branches
   ここに以下の通り設定する。（master は自動で Protected になっています）

Protected branch	Last commit	Allowed to merge	Allowd to push
master	–	Maintainers	許可なし
Release*	–	Maintainers	許可なし

（↑ブランチ名が Release から始まるものの意味。”1.2.ブランチ構成・目的”の項を参照）

4.4. ユーザごとの初期設定

メンテナ、開発者等、リポジトリを clone する必要のあるユーザは、各自で下記の初期設定を行ってください。

1)	**GitLab ログイン確認**
2)	**git クライアントでユーザ設定**
3)	**ssh-key 生成**
4)	**GitLab に ssh-key 登録**
5)	**global 設定**
6)	**Git-LFS の path 設定**

1) GitLab ログイン確認

自分のユーザ名でログインしてください（Email でのログインではないです）。
初回ログイン時にはパスワード変更が要求されますので、ここでいつものパスワードに変えてください。

URL: **https://gitlab.com:group/team/product** （※GitLab-URL は別途連絡）

2) git クライアントでユーザ設定

【コマンドの場合】

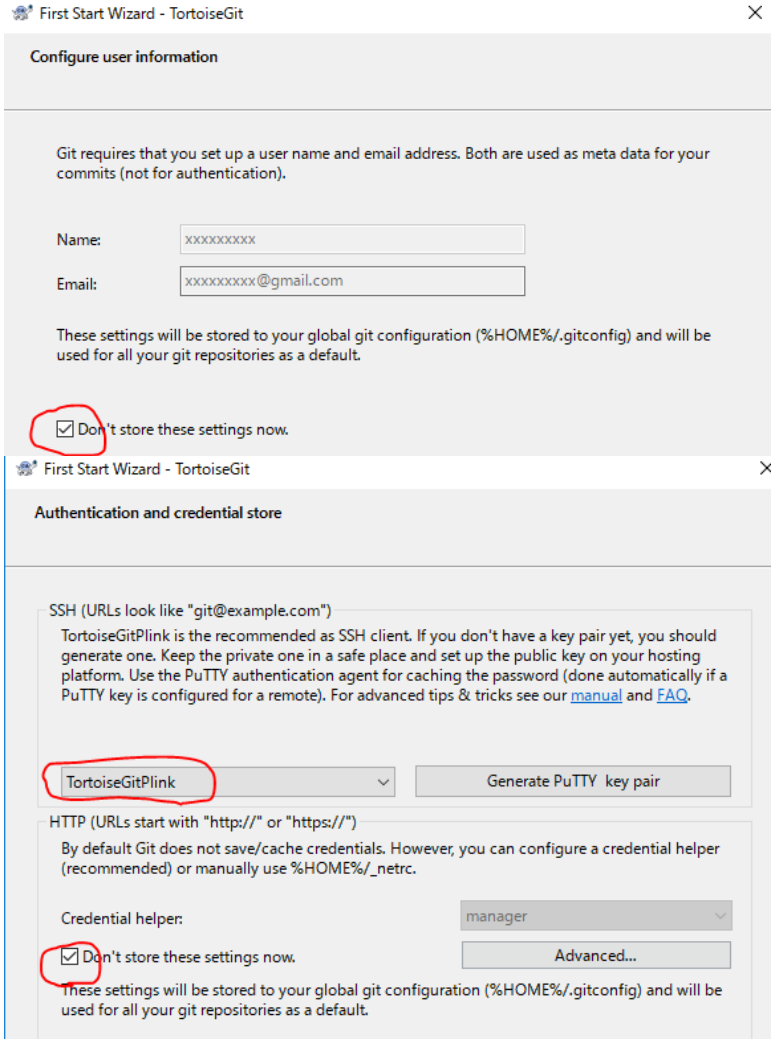
user.name、user.email はローカルリポジトリ毎に設定するため global では設定しません

【GUI の場合】

TortoiseGit を初期設定に注意しながらインストールします。

Point 1: “Name”, “Email” はローカルリポジトリ毎に設定するため「Don’t store …」をチェックします

Point 2: SSH 関連の credential 関連 はローカルリポジトリ毎に設定するため「Don’t store …」をチェックします



＊ SSH クライアントは TortoiseGitPlink を使用。　Credential helper は manager を使用

3) ssh-key 生成

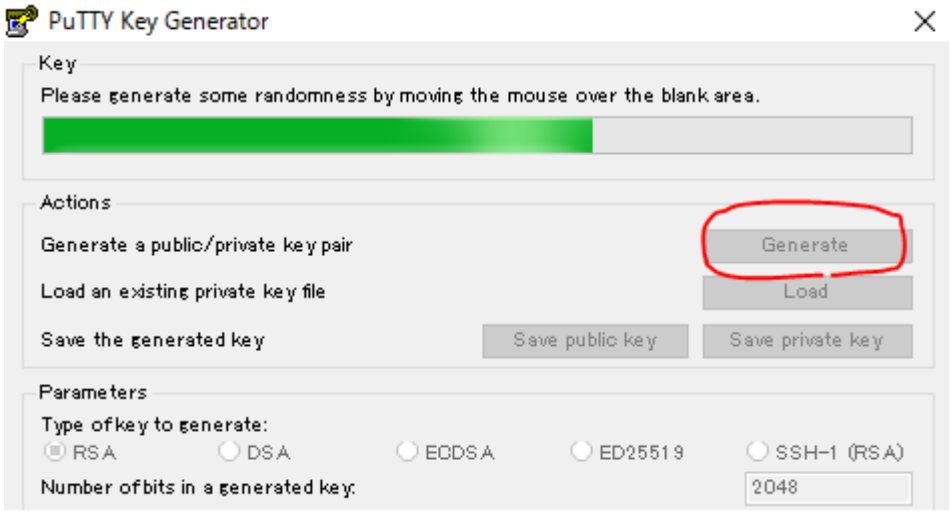
【コマンドの場合】

URL: https://gitlab.com/help/ssh/README#generating-a-new-ssh-key-pair
を参照して、ssh-key を生成します。(手元の PC や開発用サーバ上でコマンド実行)

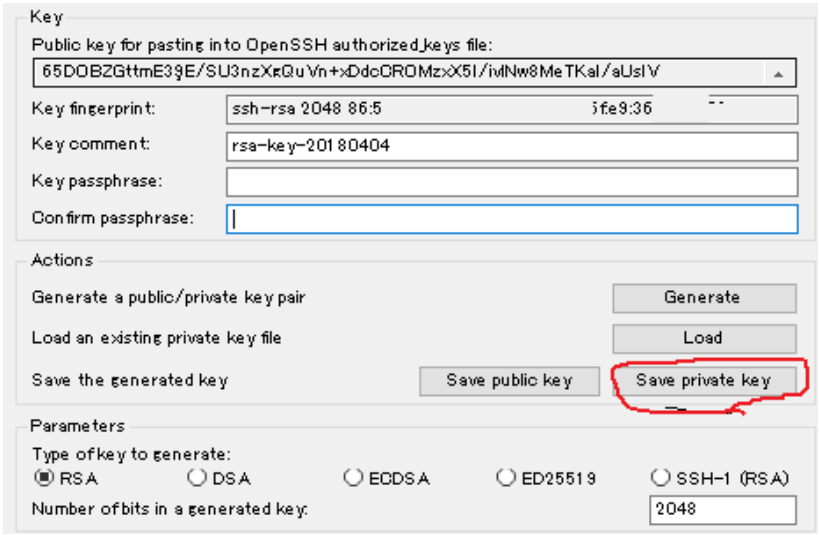
【GUI の場合】

TortoiseGit に付属の PuTTYgen を使用し ssh-key を生成します。

1. Windows のスタートメニュー → TortoiseGit → PuTTYgen
2. 「Generate」を押してからマウスを左右に動かし key を生成



3. 「Save private key」で秘密鍵を保存
   ファイル種類は、ppk を選択し保存します
   (ファイル名の例: C:¥Users¥(user_name)¥ssh-key¥id_rsa_pri.ppk)



4) GitLab に ssh-key 登録

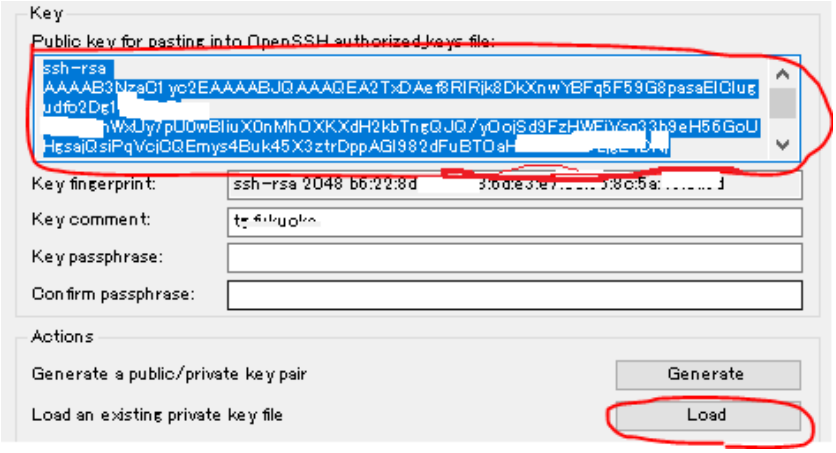
1. 保存済みの公開鍵を取得します

【コマンドの場合】

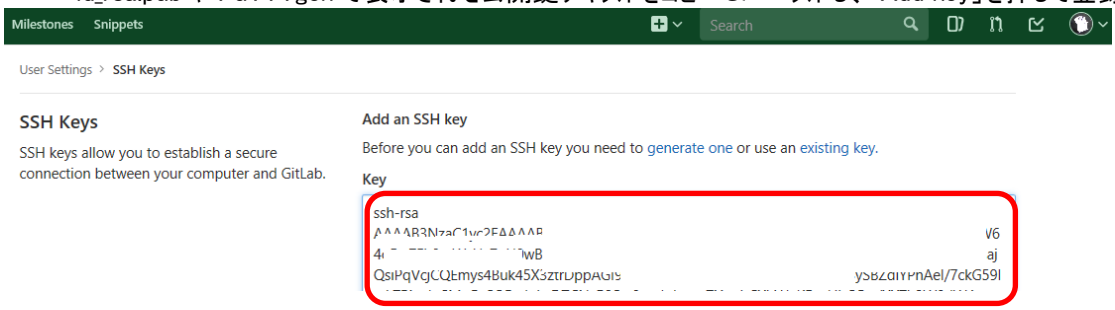
URL: https://gitlab.com/help/ssh/README#generating-a-new-ssh-key-pair
を参照して、(おそらく.ssh/id_rsa.pub に)保存済みの公開鍵テキストを取得します

【GUI の場合】

Windows のスタートメニュー → TortoiseGit → PuTTYgen
Load を押して、先ほど秘密鍵を保存したファイルを読み込みすることで
公開鍵をテキストで表示します



2. GitLab にログインして、右上のユーザアイコンから settings を選択して User settings 画面へ入り、
左メニューで[SSH Keys]を選択。
id_rsa.pub や PuTTYgen で表示された公開鍵テキストをコピー＆ペーストし、「Add key」を押して登録



5) global 設定

このマニュアルは、git クライアントの global 設定が下記のとおりであることを前提として記述しています。
必要に応じて設定してください。

【コマンドの場合】

・user.name, user.email の設定を削除します。

```
$ git  config  --global  --unset-all  user.name
$ git  config  --global  --unset-all  user.email
```

・merge では no-ff をデフォルトにし、pull では ff をデフォルトにします。

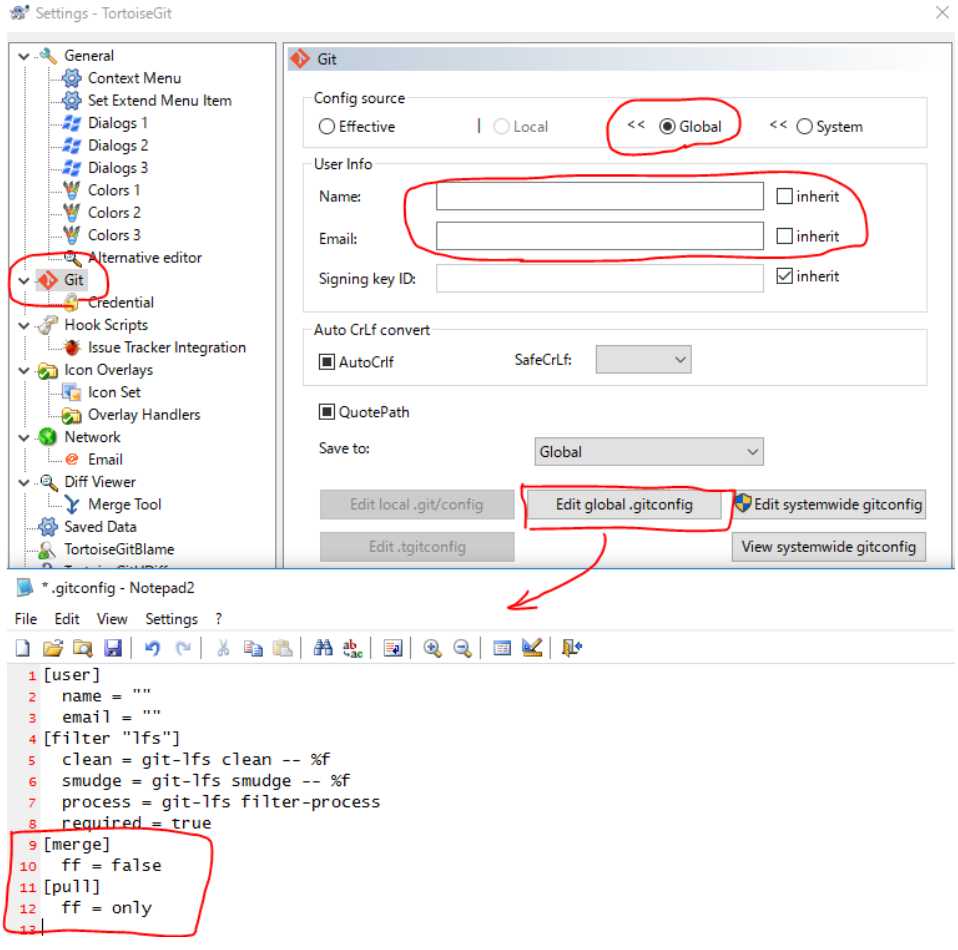
```
$ git  config  --global  merge.ff  false
$ git  config  --global  pull.ff  only
```

【GUI の場合】

TortoiseGit インストール時の設定を確認し、必要に応じて修正・設定します。

1. デスクトップ上等で右クリック → TortoiseGit → settings を選択
2. 左メニューの”Git” → Config source に Global を選択します
3. User Info を空に設定します
4. “Edit global .gitconfig” を押すとエディタが起動しますので下記設定をします
   （注：ここで行った ff の設定は、マージの GUI のチェックボックスに反映されない可能性があります。）

```
[merge]
   ff = false
[pull]
   ff = only
```

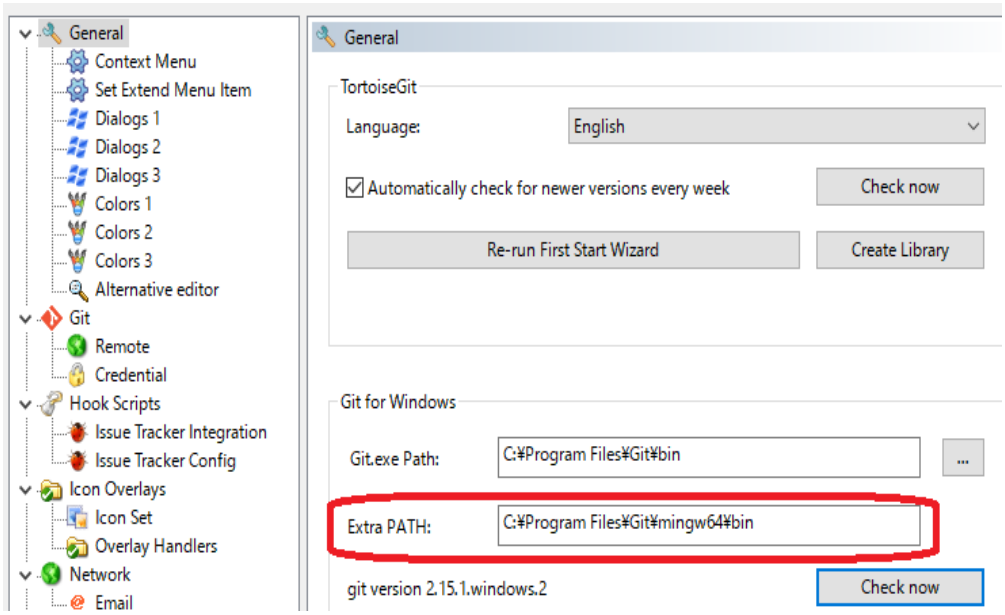


6） Git-LFS の path 設定

【GUI の場合】

TortoiseGit に Git-LFS コマンドの場所を追加登録します

1. デスクトップ上等で右クリック → TortoiseGit → settings を選択
2. 左メニューの”General” → Extra PATH に git-lfs.exe が存在するディレクトリを設定します



【例】Extra PATH:　C:¥Program Files¥Git¥mingw64¥bin

4.5. 設定ファイルの例

4.5.1. gitattributes （for Unity ）

```
## Unity Git LFS settings ##

## git-lfs ##

#Image
*.jpg filter=lfs diff=lfs merge=lfs -text
*.jpeg filter=lfs diff=lfs merge=lfs -text
*.png filter=lfs diff=lfs merge=lfs -text
*.gif filter=lfs diff=lfs merge=lfs -text
*.psd filter=lfs diff=lfs merge=lfs -text
*.ai filter=lfs diff=lfs merge=lfs -text
*.tif filter=lfs diff=lfs merge=lfs -text

#Audio
*.mp3 filter=lfs diff=lfs merge=lfs -text
*.wav filter=lfs diff=lfs merge=lfs -text
*.ogg filter=lfs diff=lfs merge=lfs -text

#Video
*.mp4 filter=lfs diff=lfs merge=lfs -text
*.mov filter=lfs diff=lfs merge=lfs -text

#3D Object
*.FBX filter=lfs diff=lfs merge=lfs -text
*.fbx filter=lfs diff=lfs merge=lfs -text
*.blend filter=lfs diff=lfs merge=lfs -text
*.obj filter=lfs diff=lfs merge=lfs -text

#ETC
*.exe filter=lfs diff=lfs merge=lfs -text
*.a filter=lfs diff=lfs merge=lfs -text
*.exr filter=lfs diff=lfs merge=lfs -text
*.tga filter=lfs diff=lfs merge=lfs -text
*.pdf filter=lfs diff=lfs merge=lfs -text
*.zip filter=lfs diff=lfs merge=lfs -text
*.dll filter=lfs diff=lfs merge=lfs -text
*.unitypackage filter=lfs diff=lfs merge=lfs -text
*.aif filter=lfs diff=lfs merge=lfs -text
*.ttf filter=lfs diff=lfs merge=lfs -text
*.rns filter=lfs diff=lfs merge=lfs -text
*.reason filter=lfs diff=lfs merge=lfs -text
*.lxo filter=lfs diff=lfs merge=lfs -text
```

4.5.2. .gitignore （for Unity ）

```
### gitignore file for Unity project.

# Unity generated
[Ll]ibrary/
[Tt]emp/
[Oo]bj/
[Bb]uild/
[Bb]uilds/
Assets/AssetStoreTools*
Assets/public/
Assets/public.meta
Assets/XmlTest/
Assets/XmlTest.meta

# Unity3D generated meta files
*.pidb.meta
*.pdb.meta

# Visual Studio cache directory
/.vs/

# Visual Studio / MonoDevelop generated
ExportedObj/
.consulo/
*.csproj
*.unityproj
*.sln
*.suo
*.tmp
*.user
*.userprefs
*.pidb
*.booproj
*.svd
*.pdb

# OS generated
*.bak

*~
.DS_Store
.DS_Store?
._*
.Spotlight-V100
.Trashes
Icon?
ehthumbs.db
Thumbs.db

# for others
*.log              #log files, for some plugins
*.pyc               #python bytecode cache, for some plugins.
sysinfo.txt     #Unity3D Generated File On Crash Reports
```