学内の範囲で自由に使用してください

はじめての Git と GitHub

Git+GitHub解説スライド

最終更新日　18/11/29

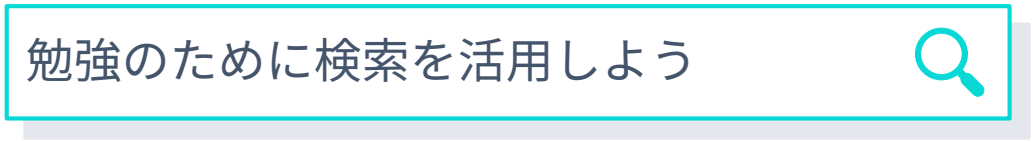
このスライドは、

開発をはじめとする様々な場面で利用される

Git と **GitHub** について、

はじめての方 向けに、

情報を減らしつつ説明したスライドです。

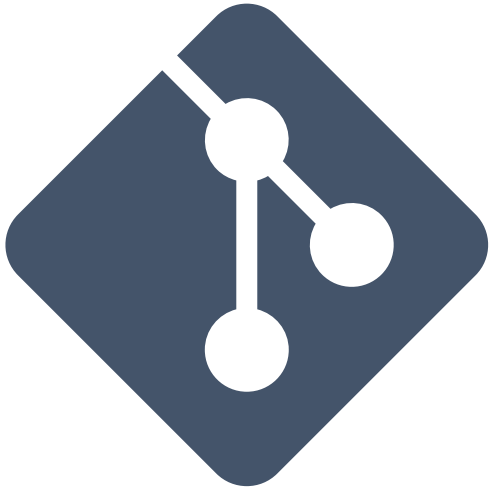


目次

イントロダクション

Git って何？
GitHub って何？

Git とは？



プログラムのソースコードなどの

変更履歴を記録・追跡できる、

分散型**バージョン管理システム**

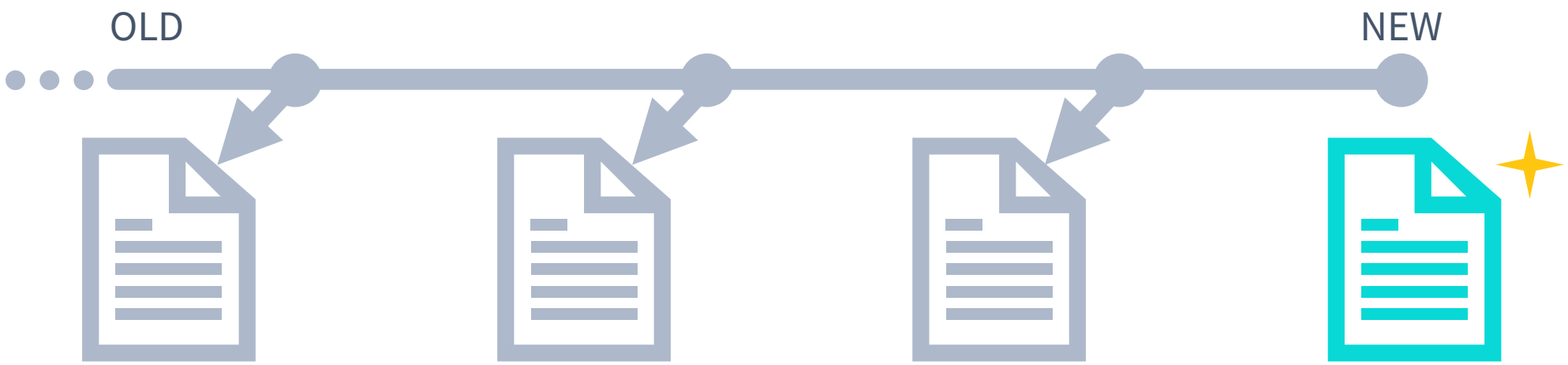
過去の変更内容すべてを履歴として保存！

何ができる？

変更内容のメモも残せる！

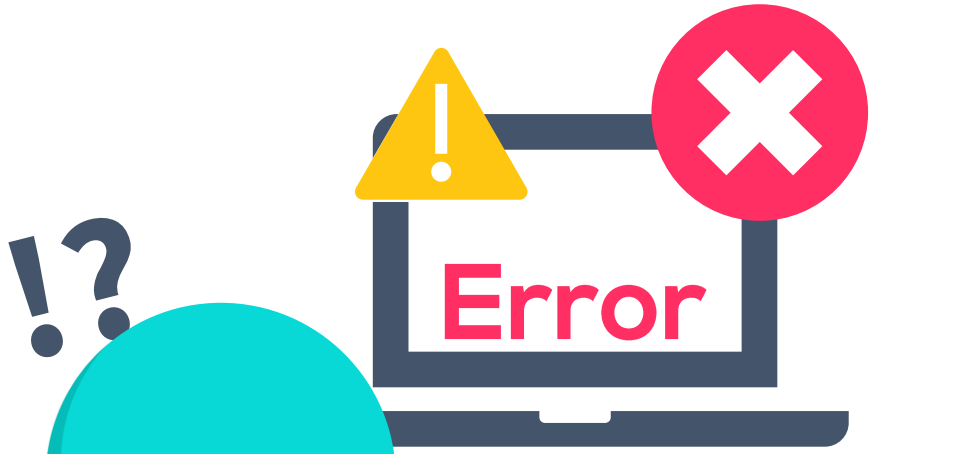
今までの編集内容を記録して

ファイルをいくらでも**過去の状態に戻す**ことができる



つまり？

「困ったことが起きても、**平和なあの頃に帰れる**――――。」



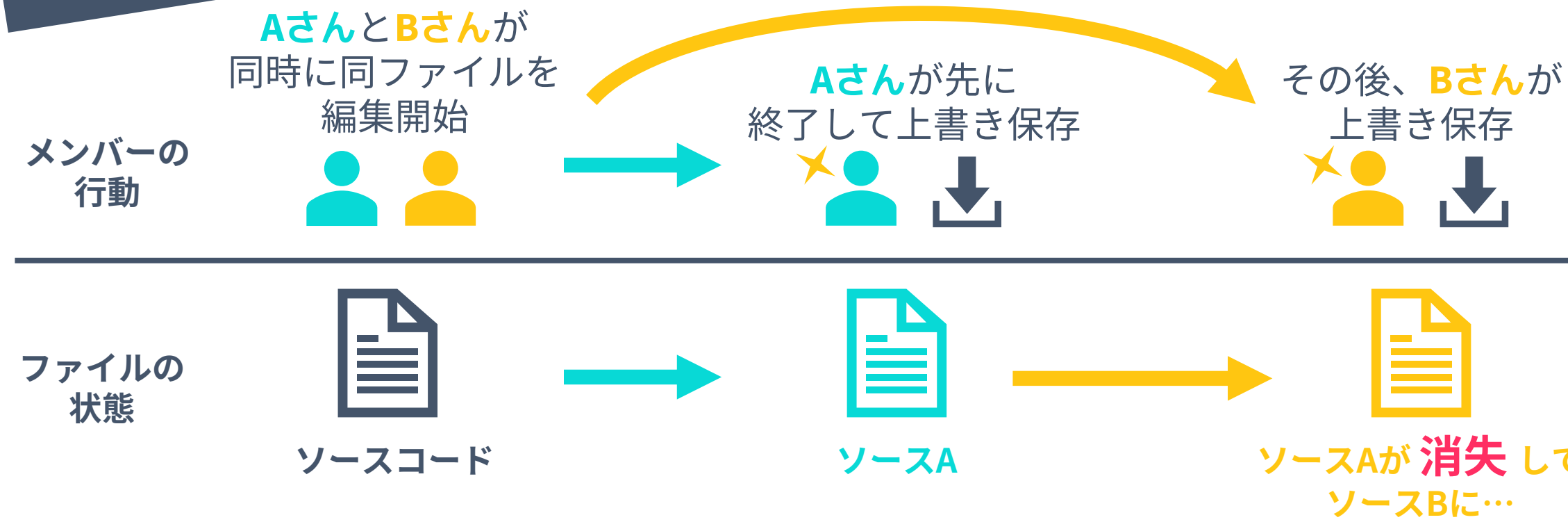
データ保存後に
意味不明なエラー

唐突すぎる
仕様変更

さらには？

複数人で使うとき、あるメンバーの作業が
他メンバーの作業によって**消されてしまうことを防いでくれる**

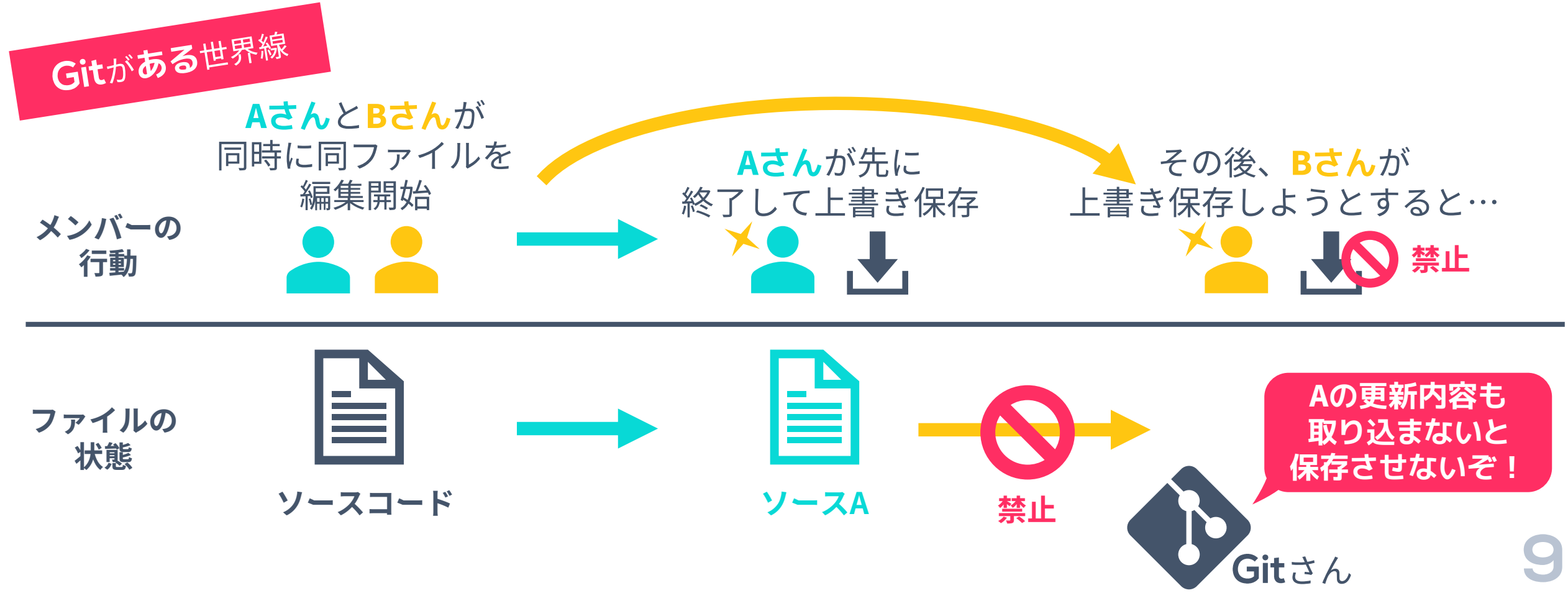
Gitがない世界線



さらには？

複数人で使うとき、あるメンバーの作業が
他メンバーの作業によって**消されてしまうことを防いでくれる**

Gitがある世界線



おまけに？

余計なファイルを残さずに綺麗に管理できる

もっと言うと？

いろいろなファイル形式を扱える

でも…

Gitはローカル（自分のPC内）で動かすツール。

「Gitだけ」では他の人と一緒にオンラインで作業ができない。

そこで  GitHub の出番。

GitHub とは？

Gitを利用したプログラムの管理や

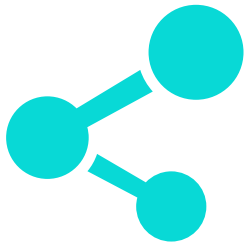
ソースコード等の共有を行うことのできる

Webサービス

Tips

Git	GitHub
= バージョン管理システム	= Gitを利用したWebサービス
オフライン	オンライン

GitHub でできること（一部抜粋）



ソースコードなどを相互共有

Gitでの作業内容をサイトにアップロードし、
他の開発メンバーや世界の人々にその作業内容を円滑に共有できる。

プロジェクトの管理

開発における課題(Issues) をサイト上に登録・管理したり、プロジェクトの
Wikiを作成して、目標や制約など様々な情報を整理したりできる。

情報の可視化

過去の更新でどのファイルが誰によってどう変化したかの具体的な差分を
確認したり、今までの変更履歴すべての軌跡をビジュアルで確認できる。

Git と GitHub を使うことで

プロジェクト全体の進捗・課題を明確にしつつ

円滑・円満な開発 を進めやすくなる。

実践編

GitとGitHubを使ってみよう

準備

Git に触れるその前に

CUIを恐れない

gitは基本的にCUI（コマンドの記入・実行）で操作します。
不慣れなツールで不安もあると思いますが、基本的にPCが壊れたりしません。

実行結果の英文を読もう

エラー文など全て英語で表示されますが、難しいことは書かれていなく、
コマンドが成功したか失敗したかなど、様々な情報を得られます。
Gitの場合、次にユーザがすべき行動の提案まで書いてくれています。

Git 操作のツール「ターミナル」を起動

Macの人

1

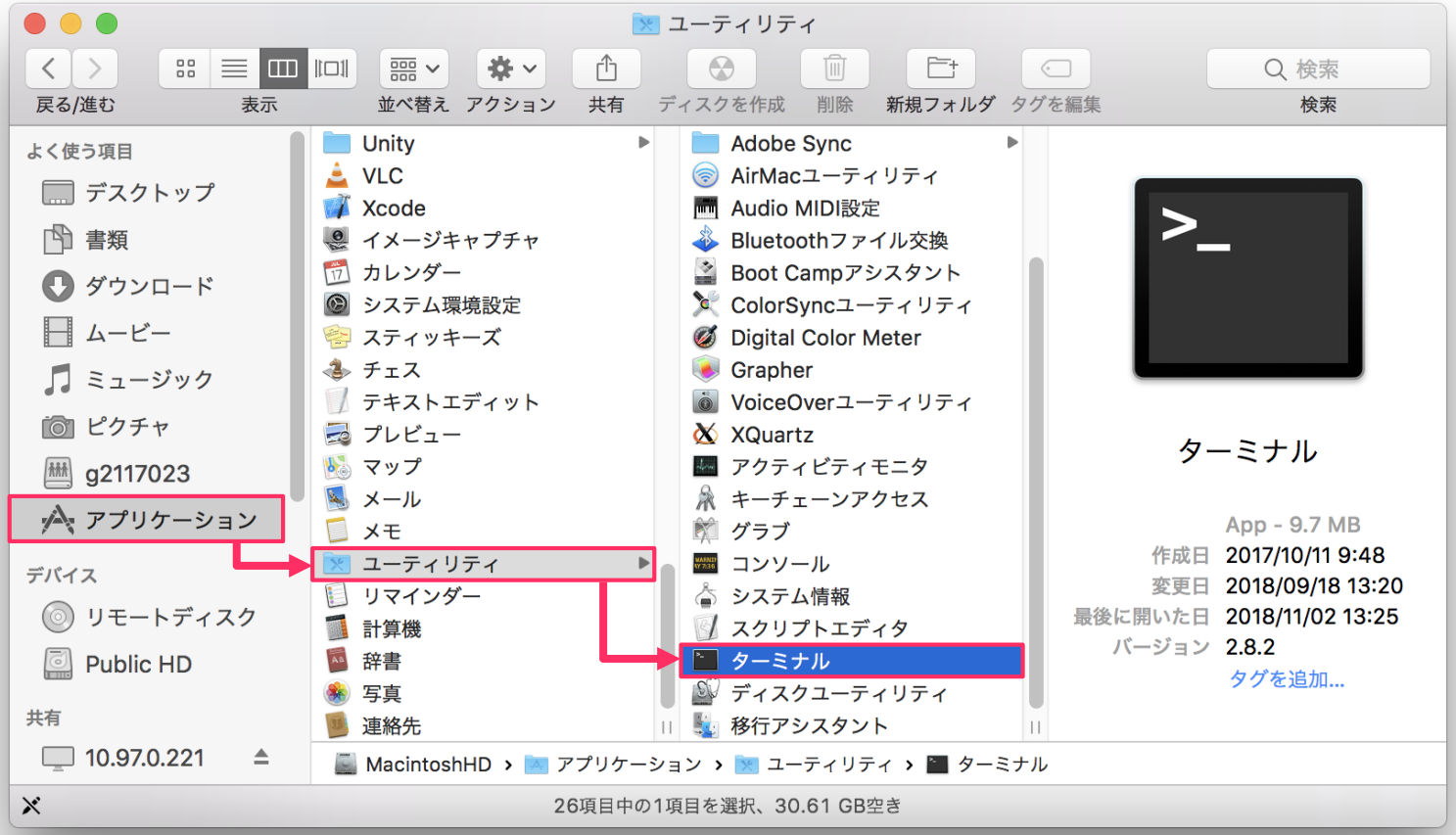
デスクトップ画面左下
ファインダーを起動

2

アプリケーション

ユーティリティ

ターミナルを選択



Winの人

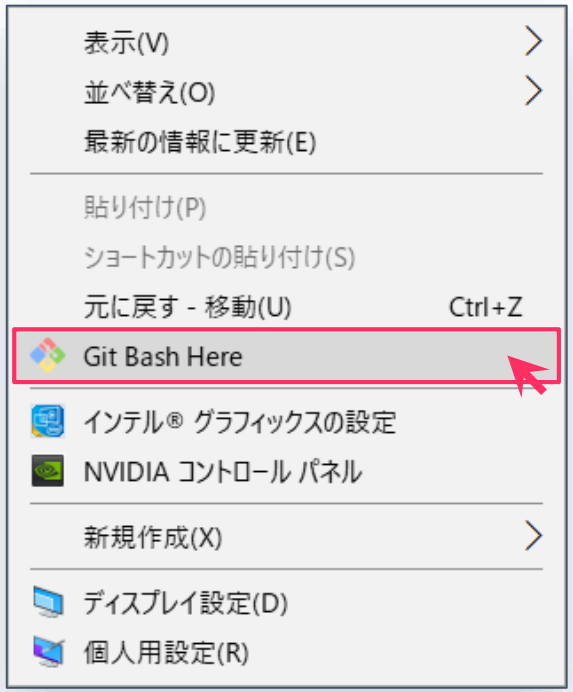
Git 操作のツール「Git Bash」を起動

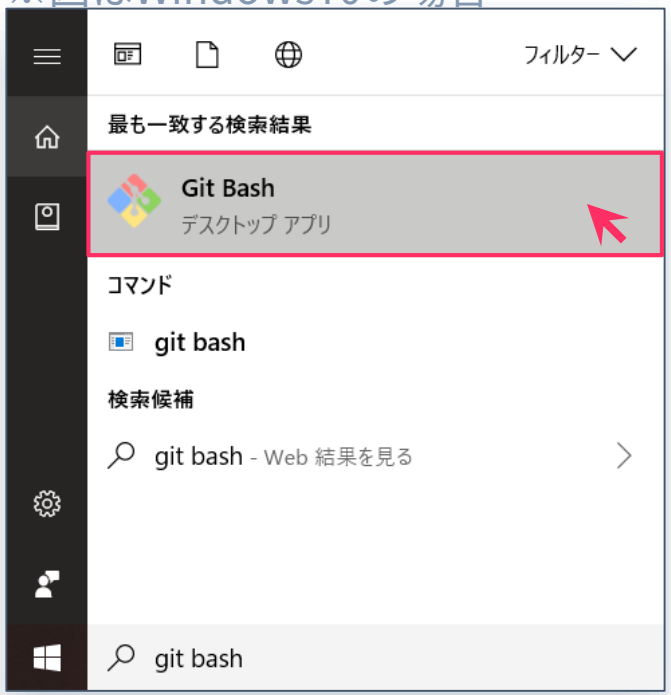
右クリックメニューから
Git Bashを選択

or

Cortanaなどで
Git Bashを検索

※図はWindows10の場合





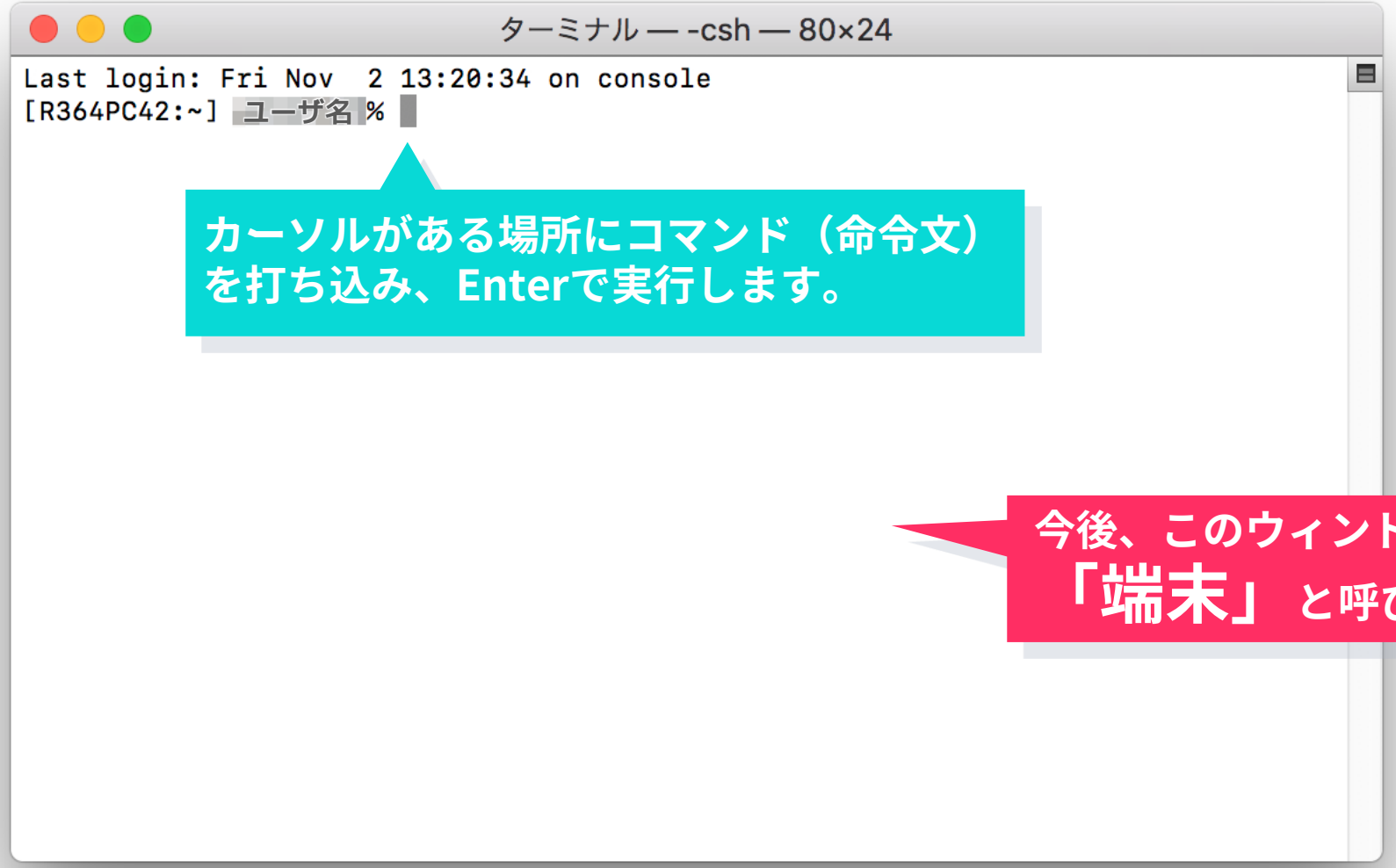
Tips

Q 「コマンドプロンプト」でGitは使えない？

A コマンドプロンプトでもGitコマンドは利用できます。しかし、Git以外の一般的なコマンド（ls, cd など）がデフォルトでは利用できず、学習のネックになりやすいです。

このような画面が表示されたらOKです。

※Macでは白背景（下図）、Winでは黒背景のウィンドウが表示されます。



実践編

GitとGitHubを使ってみよう

Let’s Try

Git + GitHub を利用した開発の流れ（一例）

Git Git操作あり　GitHub GitHub操作あり

1 Git GitHub

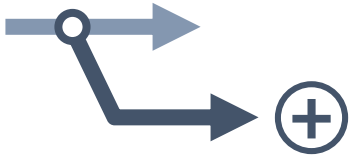
GitHub上に
プロジェクト作成

2 Git GitHub

プロジェクトを
ローカルに複製

3 Git

ブランチの作成



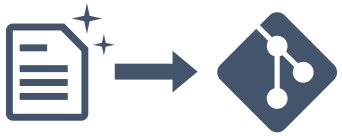
4

実装作業

左下へ

5 Git

作業内容をGitに
履歴として記録

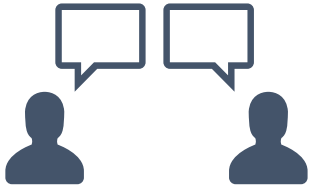


6 Git

Gitの記録を
GitHubにアップ

7 GitHub

他メンバに
作業内容を報告



8 Git GitHub

他メンバの
作業を確認

戻る 3

※ 勉強会では2~7工程のみを行います

コマンドの見方

コマンドやオプションの単語の間には
必ず「半角スペース」を入力します。

例

```
$ ls -a
testDocument.txt
```

コマンド文

実行結果例

1 “$”以降の文章が端末に入力するコマンド文になります。

先頭に“$”が無い場合は、コマンド実行結果の一例を示します。参考にしてください。

2 **青文字** は **コマンド本体** を示します。

3 **黄文字** はコマンドに付随する **オプション** 等の入力を示します。

“[ ]”がついている場合、ファイル名など、状況に応じて変化する文字列を示します。

この括弧は取り外し、適宜書き換えて入力してください。(下記参照)

例

スライドでの表記

```
$ mkdir [任意のフォルダ名]
```

▶

実際の入力例

```
$ mkdir TEST
```

作業するための場所を作ろう。

※ディレクトリ ≒ フォルダ

まずは端末が参照している場所を確認（今見ているディレクトリの確認）

```
$ pwd
```

参照している場所をデスクトップに変更（ディレクトリ移動）

```
$ cd ~/Desktop
```

作業用ディレクトリを作成して移動（ディレクトリの新規作成）

```
$ mkdir [任意のディレクトリ名（例：WorkSpace）]
$ cd [作成したディレクトリ名]
```

gitの初期設定を行う（以下の2つのコマンドを1度ずつ行えば大丈夫です）

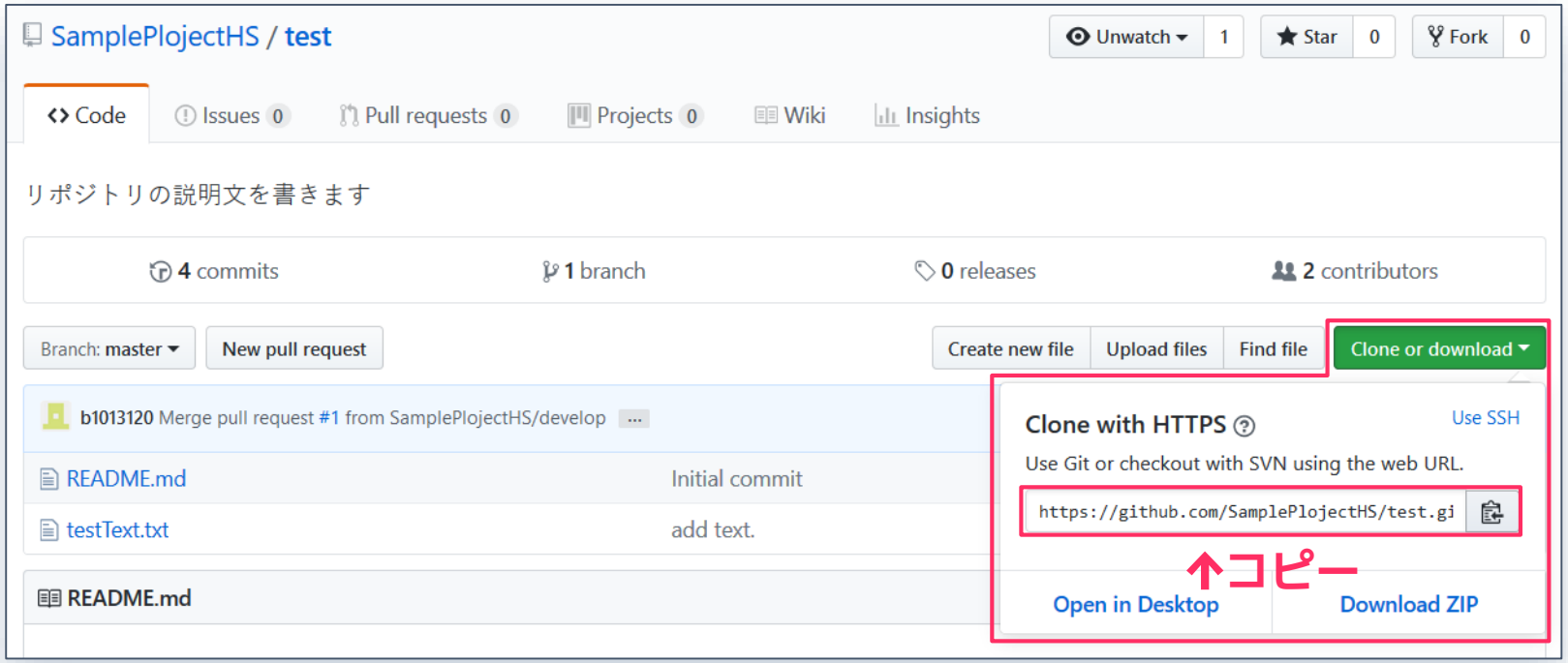
```
$ git config --global user.name “[自分の名前]"
$ git config --global user.email "[メールアドレス]"
```

GitHub上のプロジェクトをローカルに複製しよう。

演習用GitHubプロジェクトにアクセス

https://github.com/p2hacks/github-practice

プロジェクトの複製用URLをクリップボードにコピー



※図はサンプルで、実際のページと差があります。

GitHub上のプロジェクトをローカルに複製しよう。

プロジェクトを複製

```
$ git clone [コピーしたURL]
Cloning into ‘プロジェクト'...
…略…
Unpacking objects: 100% (10/10), done.
```

プロジェクトのディレクトリが作成されます

成功

作成されたプロジェクトのディレクトリに移動

```
$ cd [ディレクトリ名]/
```

Tips 覚えよう！

GitHub上のプロジェクト
＝
名 リモートリポジトリ

（複製した）
ローカルのプロジェクト
＝
名 ローカルリポジトリ

Gitが管理している場所=「リポジトリ」

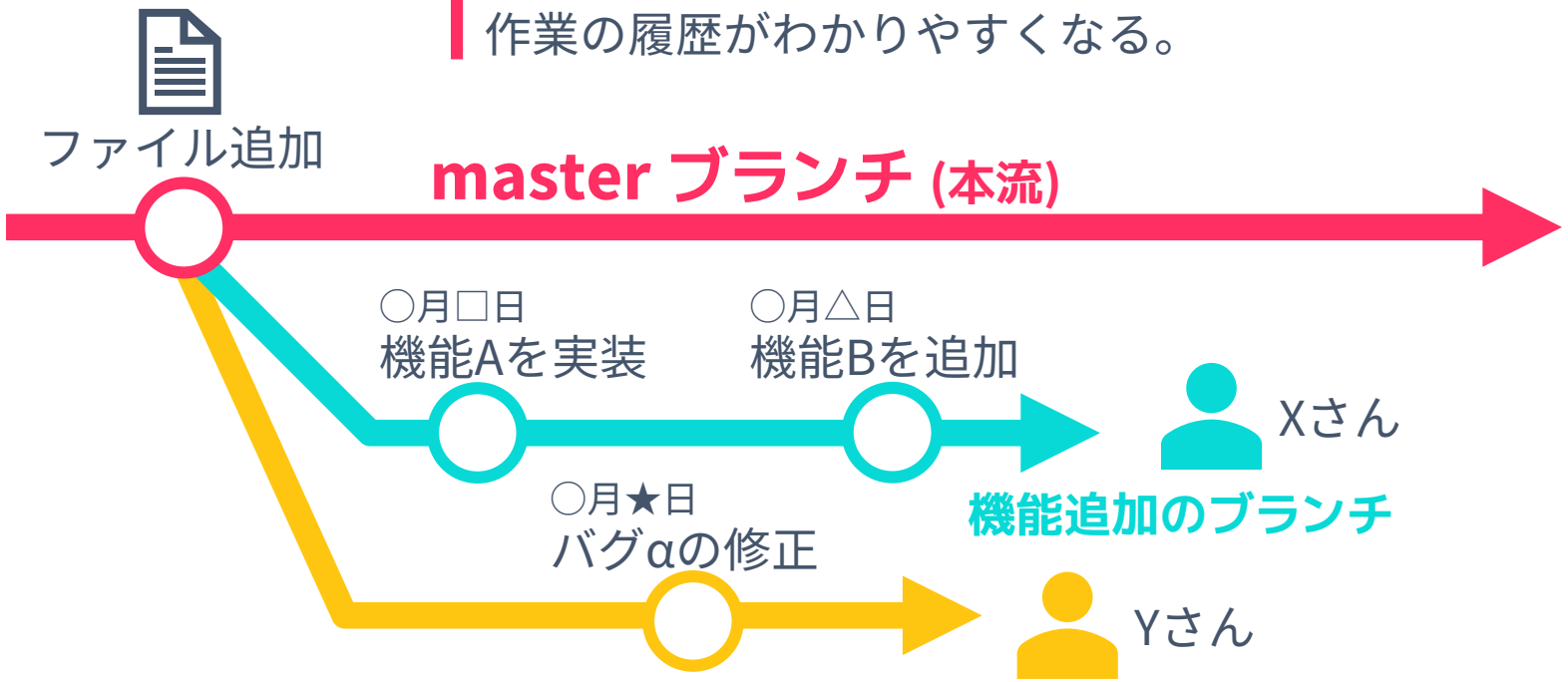
ブランチ（枝）を作ろう。

名 ブランチ

開発の本流から分岐し、本流の開発を邪魔することなく作業を続ける機能のこと。

最初から存在する「masterブランチ」から新たなブランチを作成し、**作成したブランチ上で実装を行う。**

ブランチを使えば、複数人での作業を同時に行えたり、作業の履歴がわかりやすくなる。



ブランチでの作業は他のブランチに一切影響が出ない

ブランチ（枝）を作ろう。

ローカルリポジトリにある全てのブランチ名を確認してみる

```
$ git branch
* master
```

現在Gitが見ているブランチに*アスタリスクが付きます。

ブランチの新規作成（一般的に「**ブランチを切る**」と表現されます）

```
$ git branch [任意のブランチ名(例：NewBranch)]
```

**コマンド実行時にGitが見ているブランチから
新しいブランチが作成されます。（例：左図）**

（作成した）ブランチに移動する

```
$ git checkout [(作成した)ブランチ名]
```

Let’s 実装。

ローカルリポジトリ

ソースコードなど

ローカルリポジトリ内でソースコード等を作成したり、
編集することが実装工程です。

練習として、以下のようにテキストファイルを作成し、
ローカルリポジトリ内に配置してください。

[ファイル名]　学籍番号.txt（b0000000の形式）
[内容]　氏名

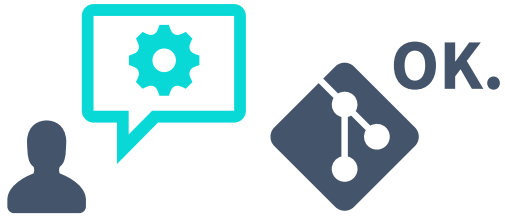
実装の後で必要なこと。

リポジトリ内でファイルの新規作成・更新・削除が行われると、対象のファイルを**Git**が感知します。

ですが、**Git**はユーザからファイルをどう扱ってほしいか教えられていないため、まだ何もしません。

そこで、**実装作業を行うたびに、管理してほしいファイルを**

Gitに教えてあげる（追跡させる, addする）必要があります。



Gitに変更内容を教えよう。

まずはgitの状態を確認しよう（git statusコマンド）

```
$ git status
On branch NewBranch
Untracked files:
 (use "git add <file>..." to include in what will be committed)

        “b0000000.txt"

nothing added to commit but untracked files present (use "git add"
to track)
```

**リポジトリ内で変化が起きた
ファイル一覧が表示される。**

**下部に現在のGitの状態と、
ユーザが次に行うべき行動の提案が書かれている。**

追跡していないファイルがあります
追跡させたい場合は”**git add**”を使って下さい

Gitに変更内容を教えよう。

Gitにファイルを追跡させよう（git addコマンド）

```
$ git add [対象のファイル名(例：b0000000.txt)]
```

“git add A B C …”の様にすると
複数ファイルまとめてaddできます。

追跡されたか確認

```
$ git status
On branch NewBranch
Changes to be committed:
 (use "git reset HEAD <file>..." to unstage)

       new file:  “b0000000.txt"
```

Gitが追跡を開始したファイルが
一覧表示される。追跡完了！

「もし間違って追跡させたなら、”git reset HEAD [ファイル名]”で追跡を解除できるよ」

ファイルを追跡させた後は…

Gitがリポジトリ内のファイルを追跡してくれるようになりましたが
Gitに更新内容を履歴として記録できていません。
このままでは、後からこの更新内容を参照したりできません。

そこで、自分の行った作業を
履歴としてGitに記録する必要があります。(commit)

変更内容を履歴として記録しよう。

Gitに履歴を記録（git commitコマンド）

```
$ git commit -m “[変更内容を一言で書きます(コミットメッセージ)]"
```

Tips

“git commit”のみを実行した場合は端末上でテキストエディタが起動し、
エディタ(vim)でコミットメッセージを書きます。
このエディタ操作が初心者には難しいため、-mを付ける方法を推奨します。

commitされているか確認

```
$ git status
On branch NewBranch
nothing to commit, working tree clean
```

**add と commit が完了すると
status がクリーンな状態になります。**

「コミットするものは何もないよ。」

変更内容をアップロード。

Gitの履歴をGitHubにアップロード（git pushコマンド）

```
$ git push origin [作業していたブランチ名]
```

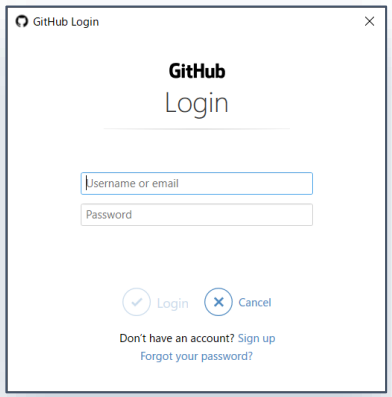
**origin は リモートリポジトリを示していると
解釈してください。**

push後、あなたのGitHubのアカウント名と
パスワードが聞かれます(毎回)

**煩わしいと思ったら、
「github ssh」で検索！**

右図のようなウィンドウが表示された場合は
このウィンドウを利用してください。

(Gitのバージョンによって変わります)



端末で入力する場合、文字を入力してもカーソルの位置が変化しませんが、
問題なく入力できているのでそのままEnterを押してください。

**「100%」や「done.」と
表示されれば ▶▶▶▶▶**

ローカルでの作業完了

「プルリクエスト」を活用しよう。

名 プルリクエスト
Pull Request

プルリクエストの良い使い方
等はp44のフローを参考に！

ローカルリポジトリでの変更を他の開発者に通知することができる。

GitHubを代表する機能の一つ。

更新内容のレビュー用Webページが作られ、レビュー完了後には

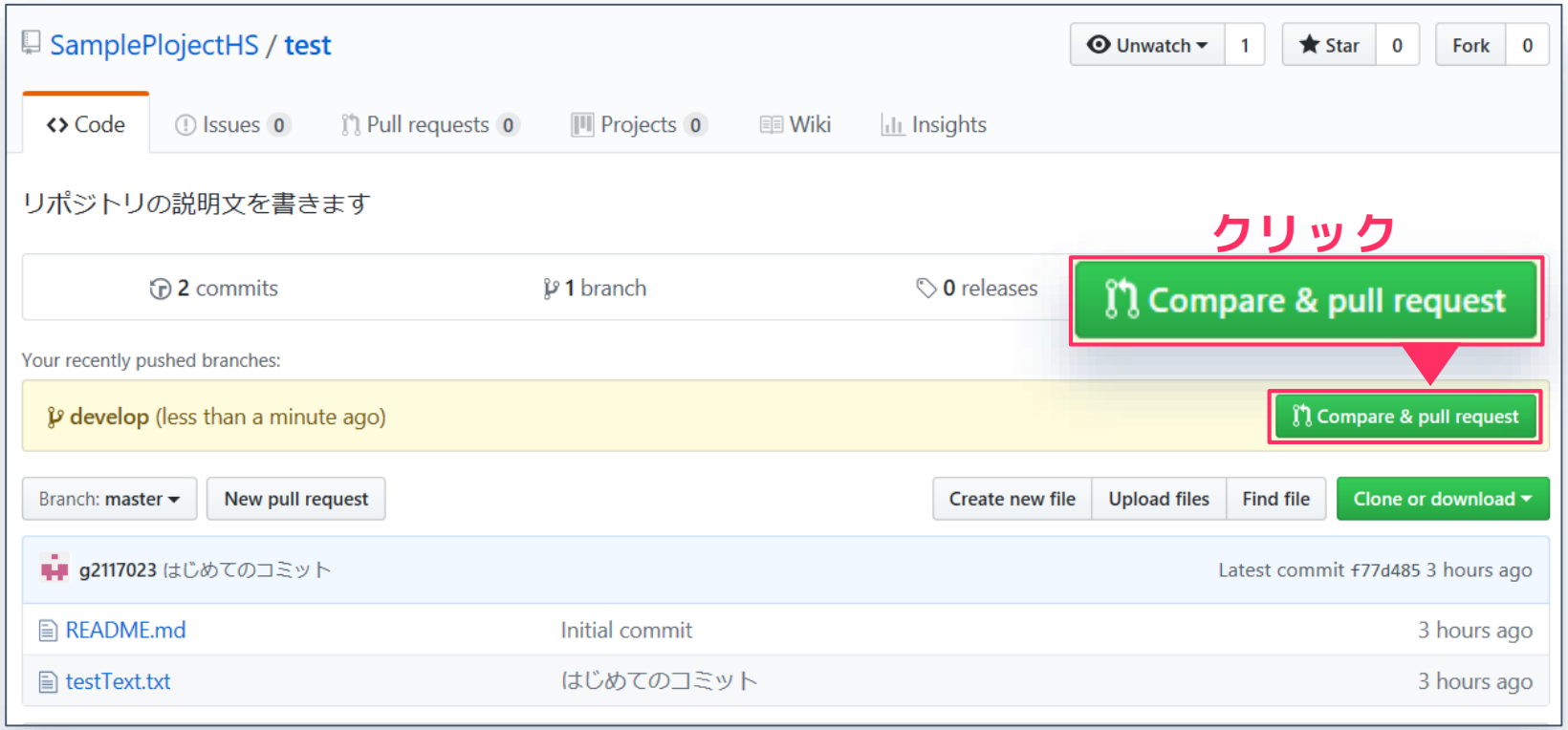
masterなどの**本流のブランチに自分のブランチを統合（マージ merge）**

してくれる。

実装作業の殆どは支流のブランチで行い、本流のブランチには
バグなどが無い、きれいな状態をマージしていきます。開発のキホン。

初めてのプルリクエスト。

演習用GitHubプロジェクトにアクセスし、「Pull Request」をクリック



※図はサンプルで、実際のページと差があります。

初めてのプルリクエスト。

各フォームで情報を入力し、「Create pull request」で作成。

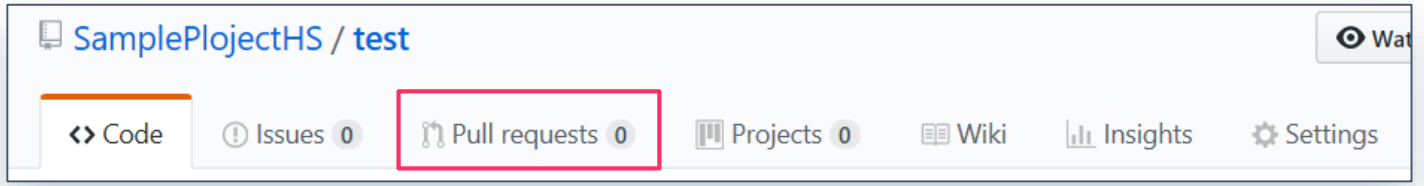
※図はサンプルで、
実際のページと
差があります。

1. プロジェクトを
ローカルに複製
   I. 作業スペースの用意
   II. プロジェクトの複製
2. ブランチの作成
3. 実装作業
4. 作業内容をGitに
履歴として記録
   I. ファイルを追跡させる
   II. 変更履歴の記録
5. Gitの記録を
GitHubにアップ
6. **他メンバに
作業内容を報告**

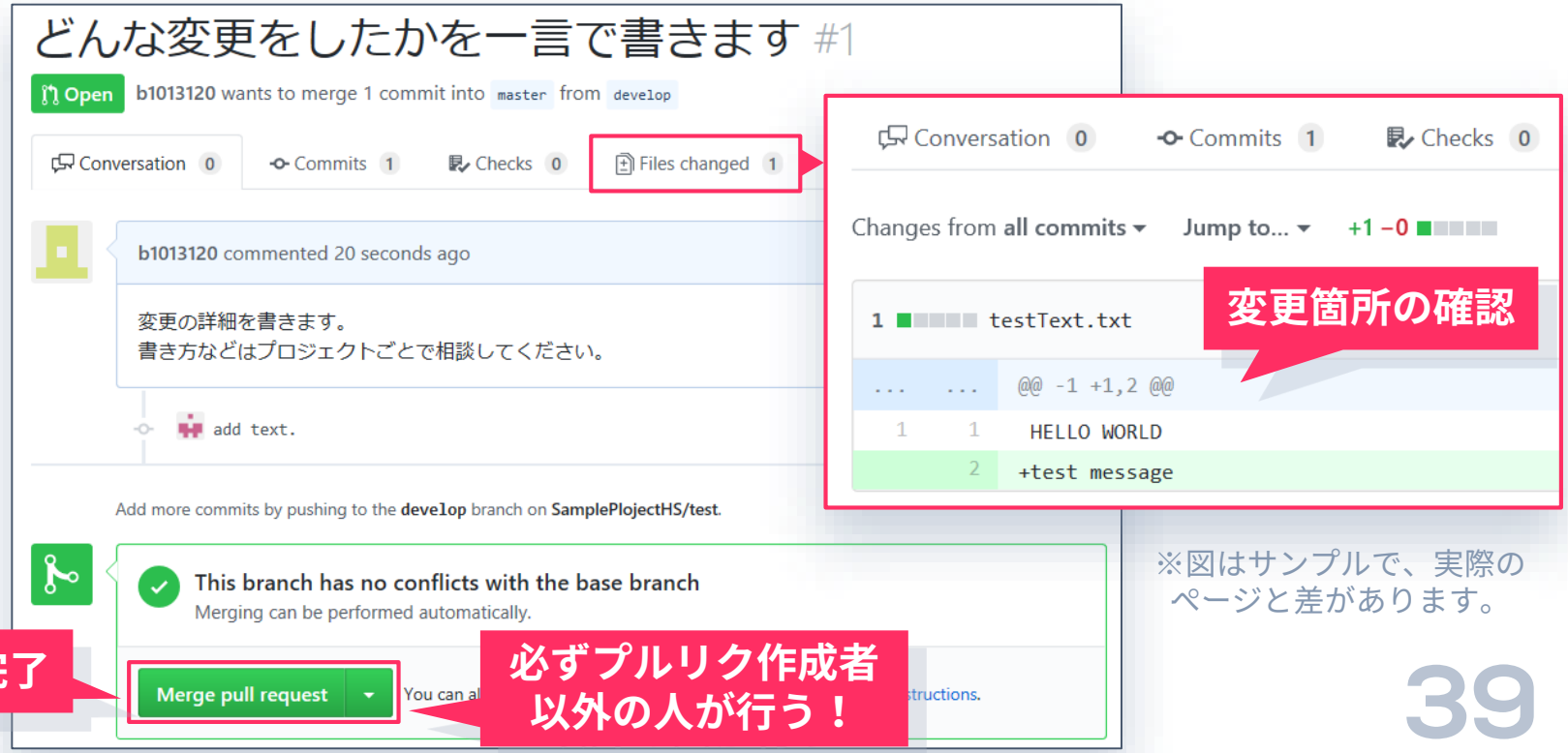
End.

おまけ：レビューの仕方

リポジトリページの
上部タブから「Pull requests」を選択 ➔ 他メンバのプルリクエストを選択



変更内容を確認したり、コメント書いたり、マージを実行したり…



※図はサンプルで、実際のページと差があります。

おさらい

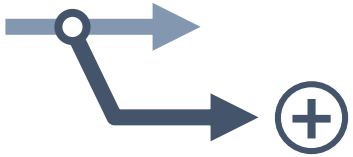
Git + GitHub を利用した開発の流れ（一例）

Git Git操作あり　GitHub GitHub操作あり

1 Git GitHub
リモートリポジトリ作成

2 Git GitHub
リポジトリ複製（git clone）

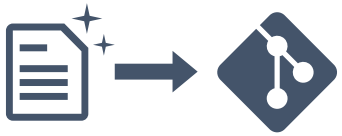
3 Git
ブランチの作成（git branch）



4
実装作業

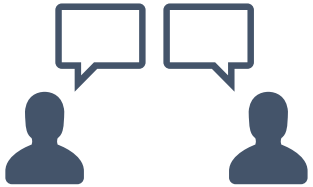
左下へ

5 Git
作業内容を記録（add, commit）



6 Git
GitHubにアップ（git push）

7 GitHub
Pull Request作成



8 Git GitHub
他メンバのPull Request確認

戻る 3

※ 1と8の工程はおまけスライドで説明しています。

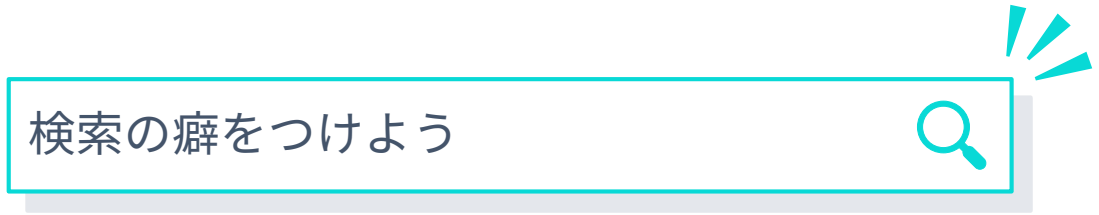
はじめての方向けに Git の使い方を紹介しましたが、

紹介していない便利な機能、知ってた方がいいことが

実はまだまだあります。（git stash, conflict など）

今回の説明でもわからない点があるかもしれませんが、

使っているうちに、自然とわかってきます。



はじめての Git と GitHub

終

おまけ　～知ってた方がいいコト～

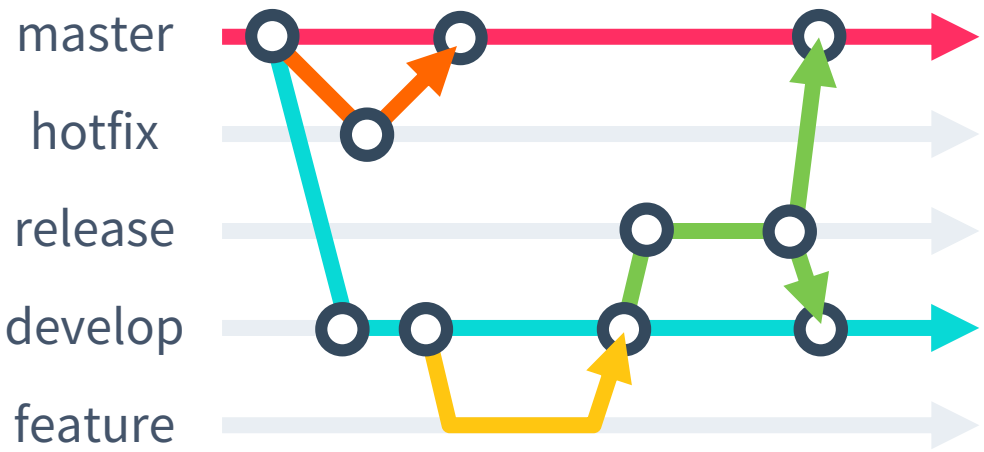
1 「GitHub Flow」を知ろう

名 GitHub Flow

Git・GitHubを利用した開発におけるワークフロー（作業の一連の流れ）。
主な６つのルールを守ることで、GitHubの円滑な運用に繋がるというもの。
Git Flowという似たワークフローも広く知られている。

Git Flow

5種類のブランチを使い分ける



GitHub Flow

2種類のブランチを使い分ける
少ない代わりにプルリクエストを活用

機能追加、バグ修正など、目的に応じて
様々なtopicブランチを切る

1 「GitHub Flow」を知ろう

GitHub Flowで守る6つのルール

1 masterブランチは常にリリース(公開)可能な状態を維持する
masterブランチで作業を行わない。このブランチにバグなどを含ませないようにする。

2 masterブランチから説明的な名前のtopicブランチを切る
例えば、「add_機能名」といった名前にする。何のブランチかわかるようにする。

3 topicブランチを定期的にpushする
こまめにcommitとpushを行い、作業内容を他メンバーに共有する。

4 プルリクエストを使ってコードレビューをする
アドバイスが欲しい時にも、ブランチをマージしたい時にもプルリクエストを作成する。

5 mergeはプルリクエストのレビューが完了してから行う
全ての課題が解決されていないブランチはmasterにマージしてはいけない。

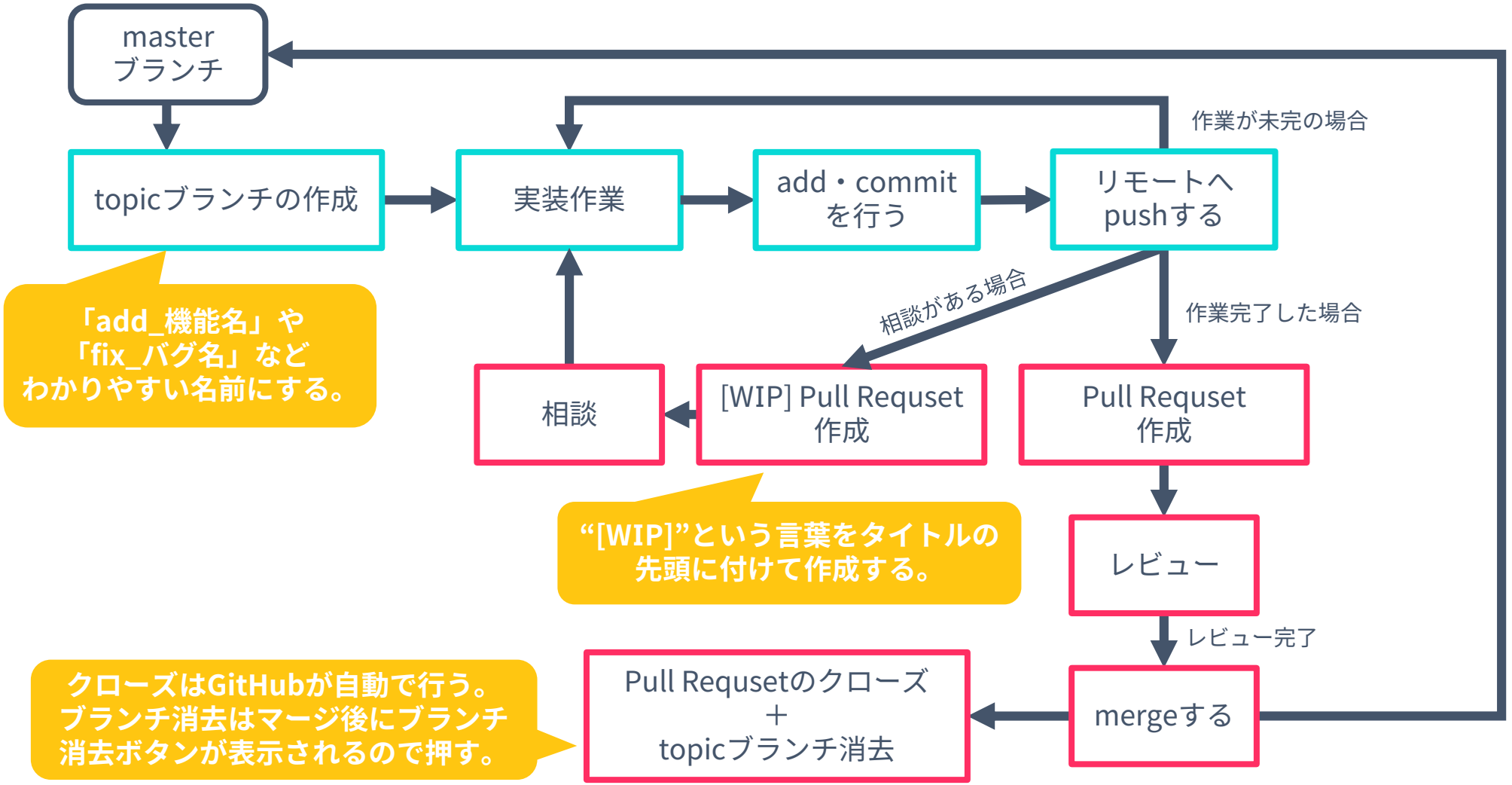
6 レビューが完了次第、速やかにマージを行う
マージを行わず放置すると、他メンバーの作業に影響が出てしまう。

1 「GitHub Flow」を知ろう

作業内容 Gitでの作業

作業内容 GitHubでの作業

GitHub Flowでの開発の流れ（簡易的）

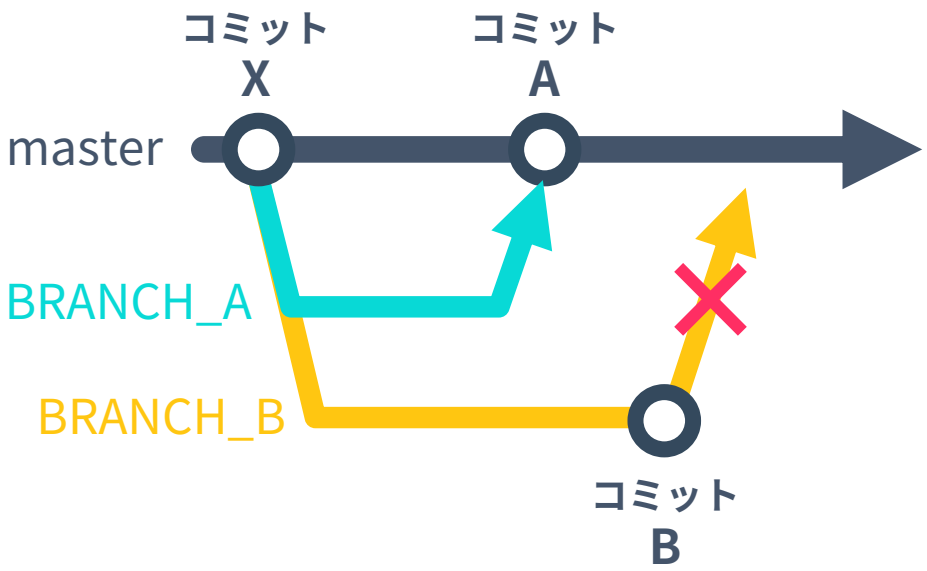


※リモートの該当ブランチのみが消去され、ローカルのは残る

2 開発の天敵「コンフリクト」

名 Conflict コンフリクト(競合)

2つのブランチを統合する際、それぞれのブランチにある同一ファイル間で整合性が取れていないと発生する状態のこと。解消しないとプルリクエストでブランチを統合できない。(勿論ローカルでもできない)



コミットXから２つの
ブランチが切られている

例 コミットBからmasterブランチにプルリクエストを使って
mergeを行おうとしたとき…

コミットAのtxtファイル

1	Hello
2	World.

≠

コミットBのtxtファイル

1	Hello
2	Japan!

お互い２行目で異なる記述がされていたため、どちらの状態が
正しいのかGitが判断できず、コンフリクトする。

プルリクエストを作ろうとすると、以下の様に表示される

✕ Can't automatically merge. Don't worry, you can still create the pull request.

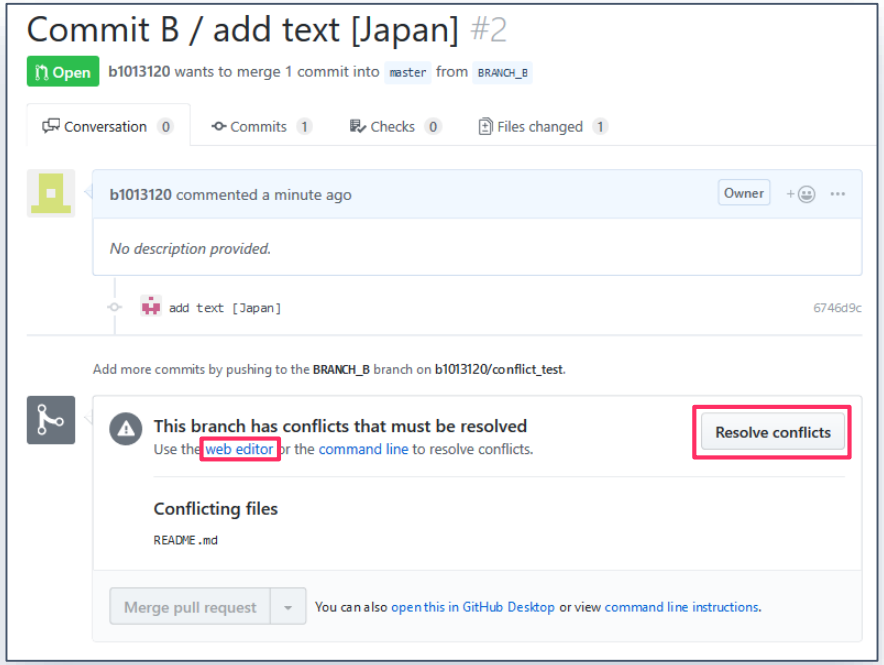
2 開発の天敵「コンフリクト」

コンフリクト修正方法１
「web editor」を使う ①

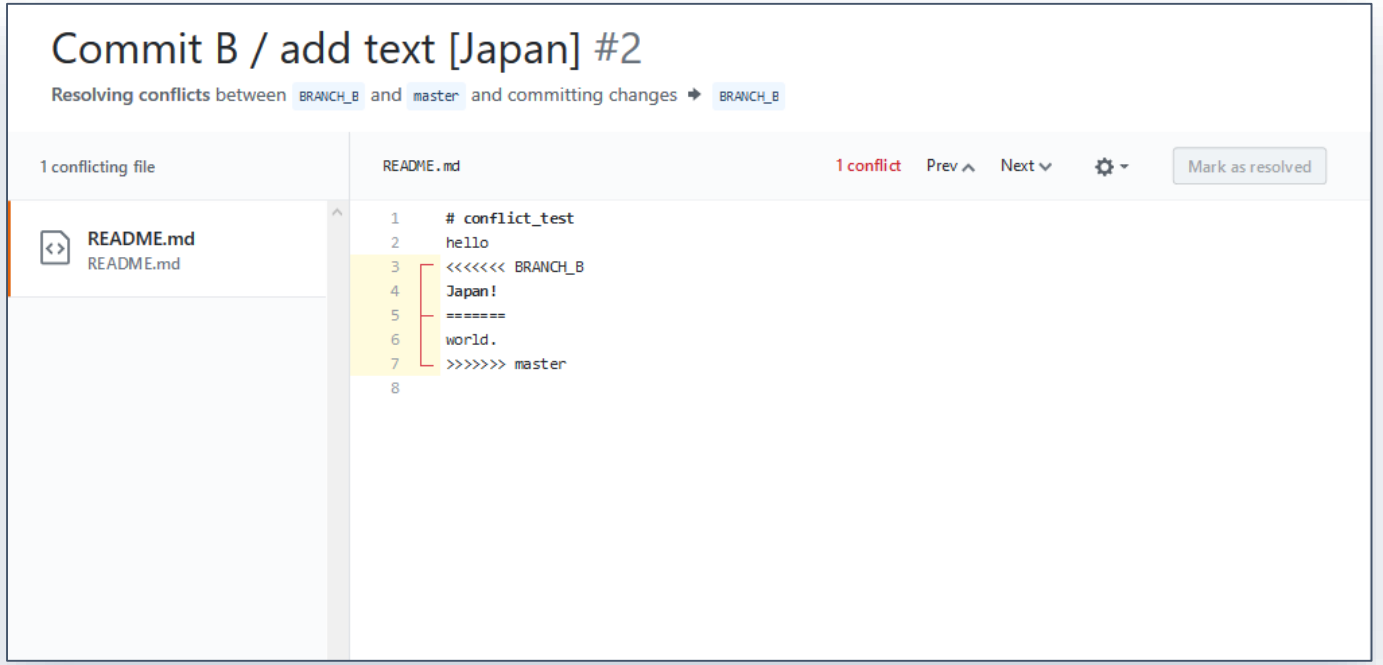
メリット　簡単

デメリット　プログラムの動作を検証しながら直せない

コンフリクトが起きたプルリクエストのページを開き
「web editor」又は「Resolve conflicts」をクリック



GitHub上でテキストエディタが表示される



※図はサンプルです。

2 開発の天敵「コンフリクト」

コンフリクト修正方法１
「web editor」を使う ②

コンフリクトが発生したファイルの該当箇所は
自動的に以下のスタイルに書き換えられる。

```
<<<<<<< 統合元のブランチ名
統合元の内容
=======
統合先の内容
>>>>>>> 統合先のブランチ名
```

<<<や===など、余計なテキストを削除し、
どの内容を残すのか、または消すのかを考え、修正する。

Before

```
# conflict_test
hello
<<<<<<< BRANCH_B
Japan!
=======
world.
>>>>>>> master
```

After

```
# conflict_test
hello
world.
```

修正後は、エディタ右上の Mark as resolved ボタンを押し、

続けて Commit merge ボタンを押せば修正完了です。

2 開発の天敵「コンフリクト」

コンフリクト修正方法2

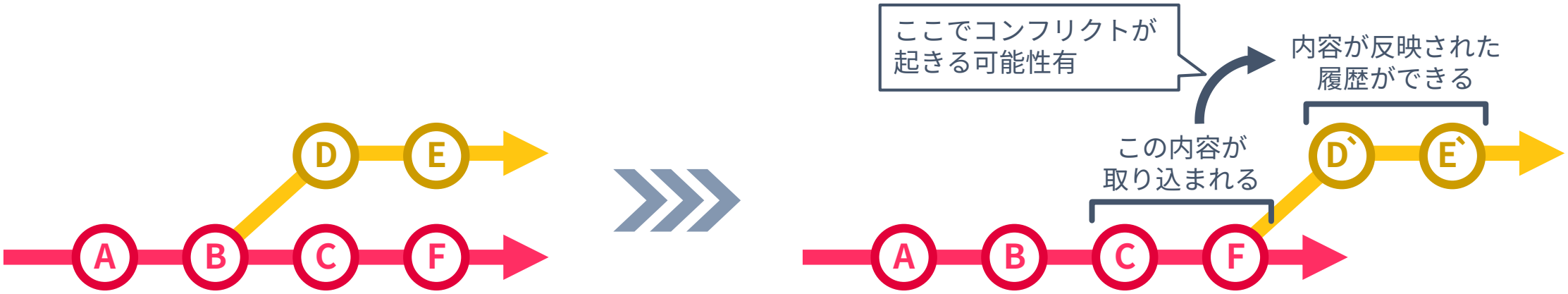
git rebaseコマンドを使う ①

メリット 履歴が綺麗になる / 他の作業を取り込みやすい

デメリット 1人で作業してるブランチのみ使える / 一部操作が危険

名 Rebase リベース

2つのブランチを統合する方法の1つ。**分岐元の最新のコミット内容を現在のブランチに取り込み**、
履歴を書き換えるコマンド。他の人の作業内容を自分の作業に取り込みやすく、便利なコマンドだが、
履歴を変に改変しないよう、扱いには注意が必要。



2 開発の天敵「コンフリクト」

コンフリクト修正方法2
git rebaseコマンドを使う ②

方針 コンフリクトをローカルで発生させながら、コンフリクト箇所を修正する

前提 masterブランチからtopicブランチを作成して実装
pushしてプルリクエストを作成したら、”README.md”というファイルでコンフリクト発生

1

```
$ git checkout master
$ git pull
```

masterブランチに移動し、
リモートの更新をローカルに持ってくる。

2

```
$ git checkout topic
$ git rebase master
…省略…
CONFLICT (content): Merge conflict in README.md
error: Failed to merge in the changes.
…省略…
```

topicブランチに移動し、rebaseを実行する。
README.mdにてコンフリクトが起きていることが確認できる。

備考 rebaseの作業そのものを取り消したい場合は、
“**git rebase --abort**”コマンドを実行すれば作業前に巻き戻る。

2 開発の天敵「コンフリクト」

コンフリクト修正方法2
git rebaseコマンドを使う ③

方針 コンフリクトをローカルで発生させながら、コンフリクト箇所を修正する

前提 masterブランチからtopicブランチを作成して実装
pushしてプルリクエストを作成したら、”README.md”というファイルでコンフリクト発生

3

```
<<<<<<< HEAD
統合元の内容
=======
統合先の内容
>>>>>>> 統合先のコミットメッセージ
```

コンフリクトが起きたファイルの該当箇所は左の様に自動で書き換えられる。余計なテキストを消し、修正して保存する。

4

```
$ git add README.md
$ git rebase --continue
Applying: 統合先のコミットメッセージ
```

修正したファイルをaddして、”**git rebase --continue**”コマンドを実行する。コンフリクトが解消されていれば「Applying: …」と表示され、rebaseは完了する。

再度エラーが出た場合は、**3** に戻り、作業を繰り返す。

2 開発の天敵「コンフリクト」

コンフリクト修正方法2
git rebaseコマンドを使う ④

方針 コンフリクトをローカルで発生させながら、コンフリクト箇所を修正する

前提 masterブランチからtopicブランチを作成して実装
pushしてプルリクエストを作成したら、”README.md”というファイルでコンフリクト発生

5 `$ git push -f origin topic` End.

▲ rebase後は-fを付けないと実行できない

topicブランチをpushする。push後にプルリクエストを確認すると、マージ可能な状態に変わっている。

注 pushの-f オプション

リモートとローカルに不整合がある場合（rebase後の様にローカルでは履歴が変化している場合など）はpushを行うことができない。だが、-fを付けることで強制的にpushを実行でき、**リモートの履歴がローカルのものに強制的に書き換えられる。**その結果、書き換えられる前の履歴で作業していた他メンバーがpushできなくなる（他メンバーには不整合が生じるため）など、**開発に大きなダメージを与える場合がある**。

1つのブランチは1人だけが担当するなどの対策をしていれば、自分の作業ブランチの履歴が変わっても、自分にしか影響が出ないため、rebase＋push -fの利用は可能。だが、push時の入力ミスなどを危惧して、rebaseを使わないという選択肢も考えたほうが良いだろう。

2 開発の天敵「コンフリクト」

コンフリクト修正方法3
git mergeコマンドを使う

メリット rebaseもpush –fも使わず安心

デメリット 履歴が少し汚くなりやすい

方針 コンフリクト箇所を修正して、ローカルでmergeしてしまう

前提 masterブランチからtopicブランチを作成して実装
pushしてプルリクエストを作成したら、”README.md”というファイルでコンフリクト発生

…topicブランチに移動するまで

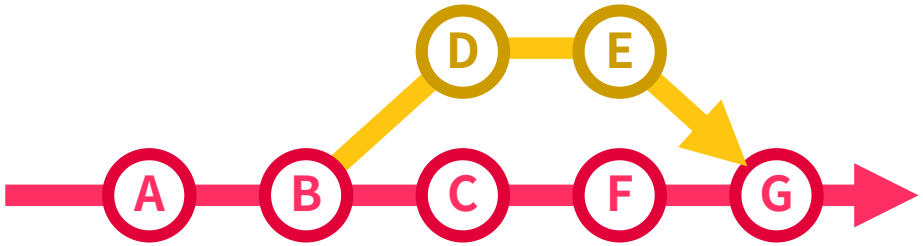
git rebaseと同じ手順なため省略…

```
$ git merge master
<コンフリクト発生・修正>
$ git push origin topic
```

End.

git rebaseコマンドをgit mergeコマンドに書き換える
以外は殆ど同じ工程。

備考 rebaseとは違い、merge後の履歴は以下の様になる。



3 邪魔なファイルを自動で除外「.gitignore」

名 .gitignore

Git の管理に含めたくないファイルを指定するためのファイル。
一部の環境設定ファイルなど、リポジトリに紛れ込むと不具合を起こすものがある。
リポジトリ内に「.gitignore」という名前で作成し、除外したいファイル等を記述しておけば、
Gitはそれらのファイルを管理しなくなる。

3 邪魔なファイルを自動で除外「.gitignore」

.gitignoreの書き方（一部抜粋）

```
# “#” で始まる行はコメント。
# どのディレクトリかを問わず、file.txtを除外
file.txt
# .gitignoreと同じディレクトリのfile.txtを除外
/file.txt
# exeという拡張子のファイルを除外
*.exe
# binというフォルダの中身ごと除外
bin/
```

言語・環境などによって、最適な.gitignoreも変わる。
.gitignoreの例を検索してみると良い。

例 swift gitignore 

ファイルの指定には**ワイルドカード**や**正規表現**が利用できる

ワイルドカード

全てのパターンにマッチする文字列のこと

* ➔ “/”以外の文字列とマッチ
? ➔ “/”以外の一文字にマッチ

正規表現

いくつかの文字列の表現方法

[0-9] ➔ 0から9の中のどれか1文字
[abc] ➔ aまたはbまたはc

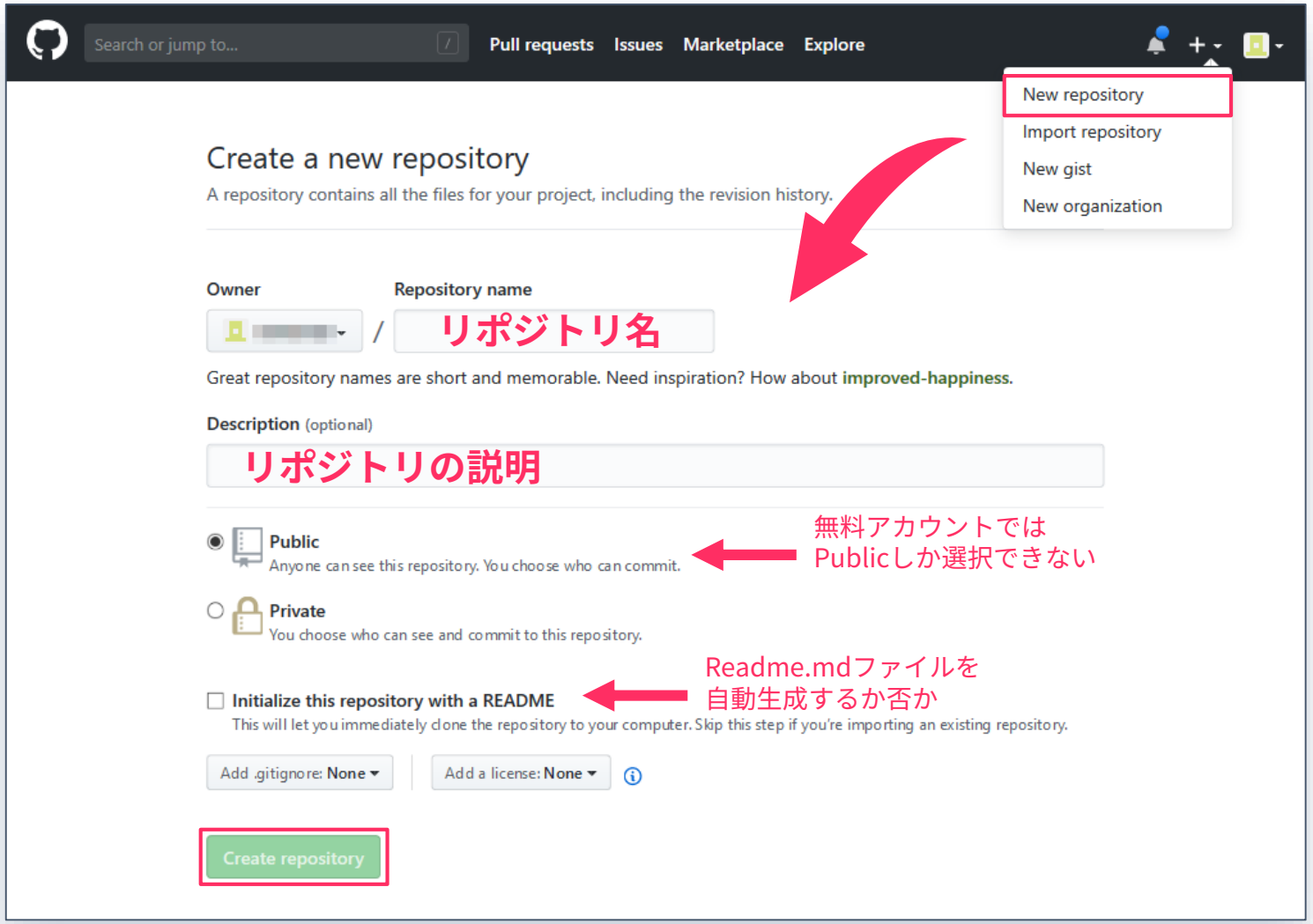
4 リモートリポジトリ・ローカルリポジトリの作り方

リモートリポジトリの作り方

GitHubにログイン。
ヘッダから「New repository」をクリック。（右上）

必要に応じて、何のリポジトリかの説明を書いたり、READMEファイル作成のオプションを選択する。

画面下、
Create repositoryで作成。



※図はサンプルです。

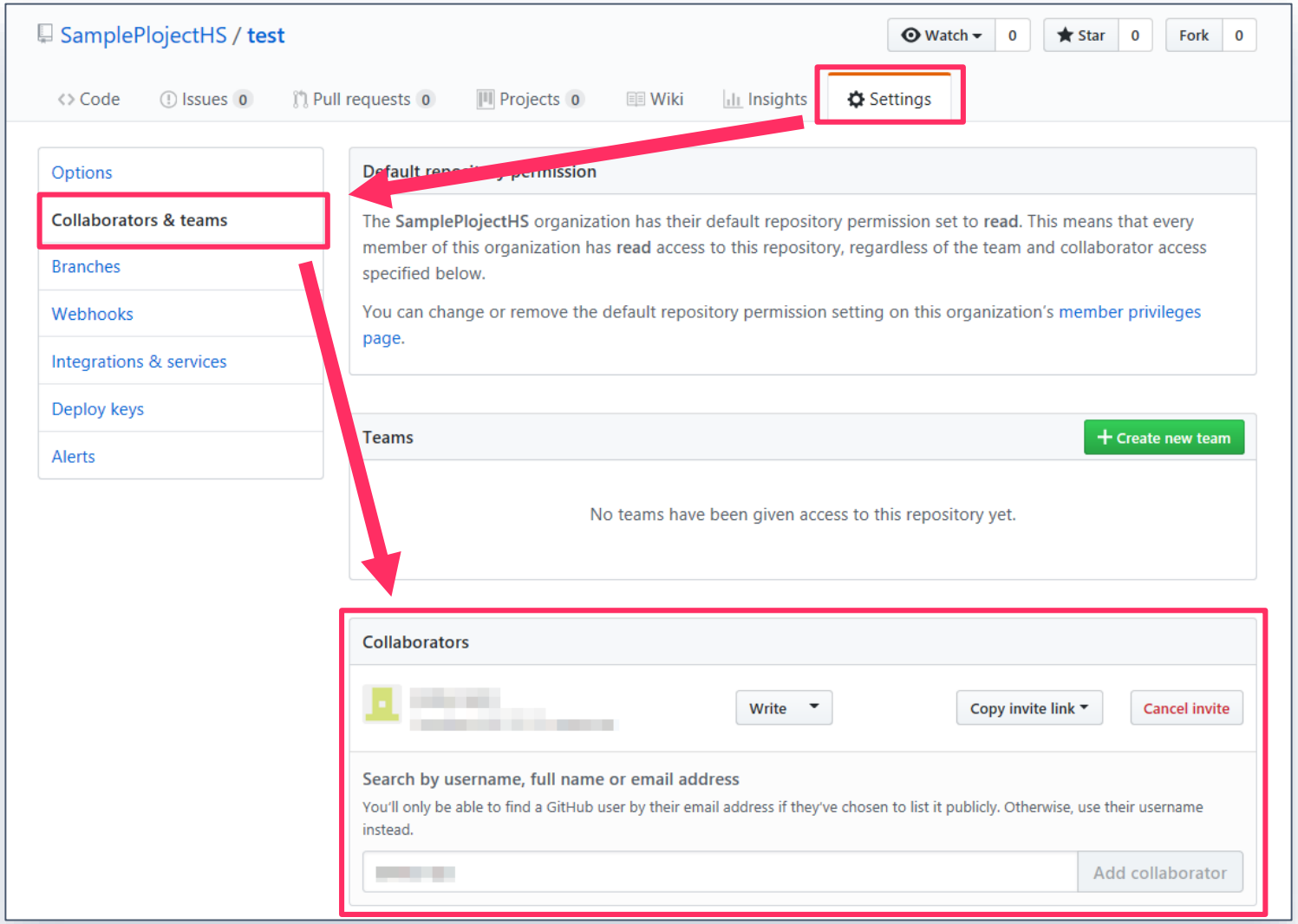
4 リモートリポジトリ・ローカルリポジトリの作り方

リモートリポジトリの初期設定　~Collaborators設定~

リモートリポジトリには
Collaborators（共同編集者）
を登録でき、この登録が無い
アカウントからpushはできない。

リモートリポジトリのページを
開き、
settingsタブ
➔ Collaborators & teams
を選択。

画面下部のCollaboratorsで
参加させたいアカウントのIDを
入力してADD。
対象アカウント宛てにメールが
届き、認証を行えば完了。



※図はサンプルです。

4 リモートリポジトリ・ローカルリポジトリの作り方

ローカルリポジトリの作り方（git cloneを使わない）

GitHubを利用する開発ではローカルリポジトリの生成にgit cloneを利用するため、あまり使わない。
ローカルだけでGitを利用したい時にどうぞ。

1つもコミットが無い状態ではブランチを作ることができない。
そこで、中身が空の「空コミット」をまず最初に行っておく。コミットメッセージは何でもよい。

現在のディレクトリにリポジトリを生成する

```
$ git init
```

リポジトリが作られたかを確認（ディレクトリ内のファイル一覧を表示）

```
$ ls -a
./  ../  .git/
```

「-a」は隠し属性になっているファイルも表示するオプション。

「.git」があればリポジトリは作成済み。

＜開発準備＞空コミットをする

```
$ git commit --allow-empty -m "first commit"
```

5 リモートの更新内容をローカルに取り込む方法

pushを行うことでリモートリポジトリは更新されるが、ローカルリポジトリは自動で更新されない。
そのため、他の開発者の更新内容を手動でローカルに取り込む必要がある。

リモートの内容をローカルに取り込む（git pullコマンド）

git pullは、対象のブランチがクリーンな状態(commitするものが無い)でなければ実行できない。

git pullはこまめに行い、他の開発メンバーの更新を取り込み忘れないよう気を付けよう。

例：最新のmasterブランチをローカルに取り込む

```
$ git pull origin master
```

originはリモートリポジトリを示す

master以外のブランチを取り込むときはmasterの部分を他のブランチ名に書き換える

5 リモートの更新内容をローカルに取り込む方法

pushを行うことでリモートリポジトリは更新されるが、ローカルリポジトリは自動で更新されない。
そのため、他の開発者の更新内容を手動でローカルに取り込む必要がある。

リモートにあるブランチをローカルで確認する（ブランチの追跡）

他の人がpushしたブランチをローカルで確認するには、そのリモートのブランチを追跡するローカルブランチを作る必要がある。

プルリクエストをレビューする際はこの追跡するブランチを作り、手元でバグなど無いか確認する。

作成するブランチ名とリモートのブランチ名は同じで問題ない。

リモートにある全てのブランチ名を確認する（ブランチ名確認に便利）

```
$ git branch –a
```

リモートのブランチを追跡するブランチを作る

```
$ git branch [ローカルに作るブランチ名] origin/[リモートのブランチ名]
```

6 便利なGitコマンド達

git log

過去のコミットログを確認できるコマンド。
コミットメッセージやそのコミットに付けられるIDも確認できる。

ログを一覧表示する

```
$ git log [オプション(無くても良い)]
```

※ログを表示している状態から抜ける際は「q」を押す。

様々なオプション（一部抜粋）

--graph	履歴を可視化する	--since=“3 days ago” --until="2018/11/01"	日付で条件を付ける
--no-merges	マージコミットを除く	-[数字]	出力件数を指定する
--oneline	1行にまとめる	--name-only	変化があったファイル名も表示

6 便利なGitコマンド達

git tag

コミットにタグやコメントを付加するコマンド。
大きな更新があったコミットに付けておくと、後から振り返りやすい。

最新のcommitにtagと注釈をつける

```
$ git tag -a [タグ名] -m “[注釈]”
```

特定のcommitにtagと注釈をつける

```
$ git tag -a [タグ名] -m “[注釈]” [コミットID]
```

※「 -m “[注釈]” 」は省略可能。

tagをリモートリポジトリに共有する

```
$ git push origin [タグ名]
```

※タグは自動でリモートに
反映されたいためpushする。

tagの一覧を確認する

```
$ git tag
```

※「-l ‘[文字列]’」といったオプションを付けるとマッチング
したタグだけ確認できる

特定のタグが付いたコミットを確認する

```
$ git show [タグ名]
```

6 便利なGitコマンド達

git reset

レポジトリの様々な状態を以前の状態に戻すコマンド。
addやcommit実行後にミスを発見した場合に重宝する。

[commit前]　addしたことを取り消す

```
$ git reset [ファイル名]
```

最新のcommitを取り消す

```
$ git reset --soft HEAD^
```

最新のcommitとそこで行ったaddを取り消す

```
$ git reset --mixed HEAD^
```

※--mixedは省略可

最新のcommitとaddとファイルの更新を全て取り消す

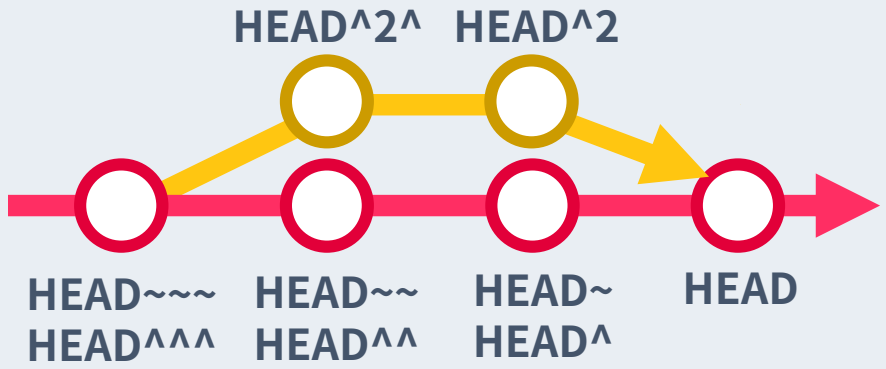
```
$ git reset --hard HEAD^
```

Tips

HEAD ＝ 最新のコミットのことを示す。

「~」と「^」を使って、最新を基準に昔のコミットも指定可能

例



チルダ(~)：先祖を指定する

キャレット(^)：親を指定する
^ :1番目の親 / **^2** :2番目の親 / **^3** :3番目 …

6 便利なGitコマンド達

git reflog

HEAD やブランチ先端の動きの履歴を確認できるコマンド。
git resetで取り消し過ぎてしまったときなど、Gitで失敗した時に便利。

履歴の確認と操作の取り消し例

```
$ git reset --hard HEAD^^ ⇦ 最新コミットだけを取り消すつもりが2個前まで取り消してしまった！
$ git reflog
[コミットID] HEAD@{0}: head^^: updating HEAD　⇦ HEAD@{0}が最新の履歴。
[コミットID] HEAD@{1}: …省略(どんな操作があったか書かれている)…　⇦ この状態に戻りたい。
[コミットID] HEAD@{2}: …省略(どんな操作があったか書かれている)…
…省略…
$ git reset --hard HEAD@{1} ⇦ reset --hard HEAD^^する前の状態に戻すことができる
```

おまけコマンド

ファイルの差分を確認する

```
$ git diff
```

ファイルを最新のcommitの状態に戻す

```
$ git checkout [対象のファイル名(パス)]
```

不要になったブランチを削除する

```
$ git branch –d [ブランチ名]
```

ブランチの作成と移動を一度に行う

```
$ git checkout –b [任意のブランチ名]
```

過去の特定のコミットに一時的に戻る

```
$ git checkout [コミット名]
```

過去の特定のコミットからブランチを切る

```
$ git checkout -b [ブランチ名] [commit_hash]
```

現在の作業を一旦退避させる

```
$ git stash
```

最新のコミットメッセージを書き直す

```
$ git commit --amend
```