GKVコードのGit/GitHubリポジトリの利用

前山 伸也

名大理

GKV定例会合, 2020年11月13日

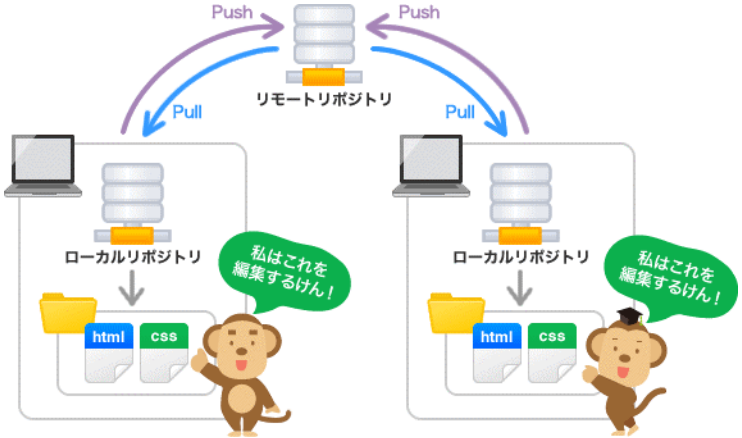
Contents

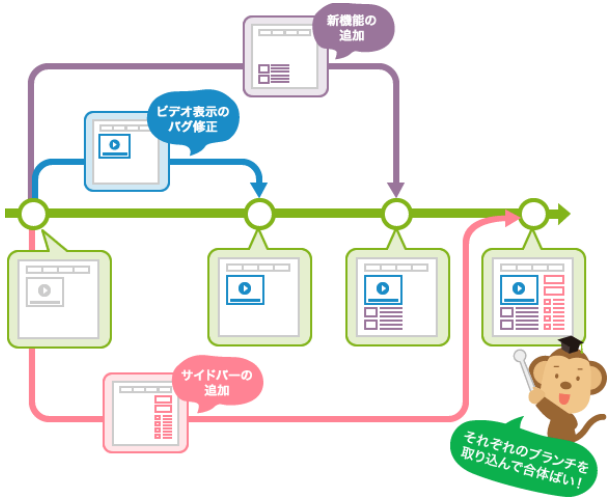
Git

Git･･･分散型バージョン管理システム

- ✓ 共有のリモートリポジトリとは別に、自分のPC内だけで作業できる個人のローカルリポジトリがある（上図）
- ✓ ブランチによる平行作業、コミットによる変更履歴の記録により、コード開発をサポート（下図）

※もちろん複数人で共同コード開発する際に強力な機能だが、自分のPC上で個人的にコード開発する際のバージョン管理にも使える。
- 昔のコミットの時点に戻りたい
- 実装方法AとBどっちが良いか両方試してから決めよう、など



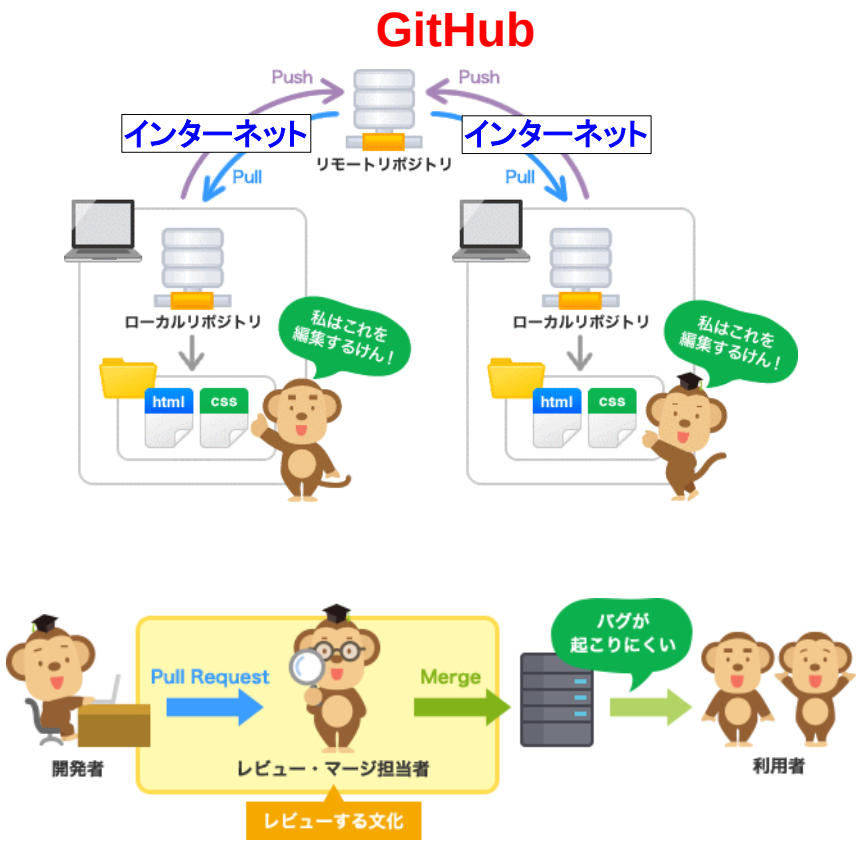


https://backlog.com/ja/git-tutorial/より転載。

GitHub

GitHub･･･Gitを利用したコード開発のためのホスティングサービス。国内･国外ともシェア最大
（要はWeb上でリモートリポジトリを提供してくれる。上図）

- ✓ 元々のGitにはない機能も提供
  - fork （他ユーザのリモートリポジトリの内容を個人のリモートリポジトリにコピーすること。外部団体のオープンソースソフトウェア開発に貢献する場合など。P研では使いません）
  - pull request （開発ブランチをmainブランチにマージするように依頼すること。下図）
  - Wikiページやブランチのグラフ表示など
- ✓ GKVは無料版オープンリポジトリとして公開しています。



https://backlog.com/ja/git-tutorial/より転載。

※他にも同じようなホスティングサービスはいろいろある。例：GitLab (無料でプライベートグループも使える。メニューの日本語表示もデフォルト対応。ホームページからの利用に加えて、自分でサーバを立てて運用することも可能。)

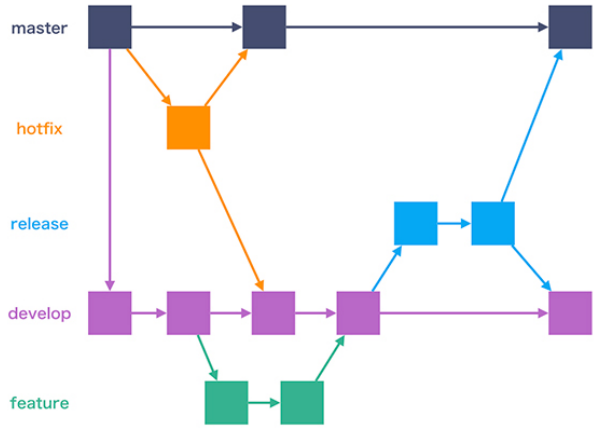
Git/GitHub の運用を開始する前に

普通、チーム内の共通ルールを決める。
※ネット上の記事やコマンド例を鵜呑みにしない。分からないなら実行せずに、チームの管理者に聞く。

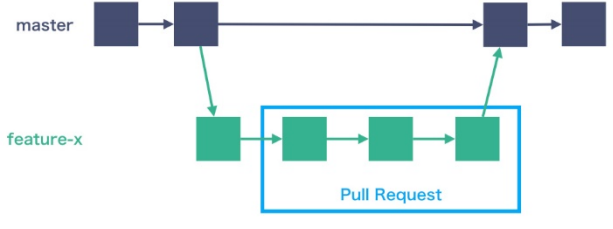
1. ブランチ管理について
    - Git-flow [Driessen (2010)]
      階層的で複雑になりがち。
    - GitHub-flow [Chacon (2011)]
      マスターと開発ブランチしかなく単純。
    - GitLab-flow [Sijbrandij (2014)]
      マスター・開発に加え、リリースブランチ。

2. その他（プルリクエストの粒度、命名規則など）
    - Webページ運営（例:cookpad）などを想定すると、なるべく細かい方がよい（もともと更新頻度高い。即座にサービスに反映。）
    - コード開発であれば、なるべくモジュール単位、サブルーチン単位が良い。（独立性が高く、機能検証しやすい。ブランチの命名も実際行う機能を指すように。）

Git-flow



GitHub-flow



https://www.atmarkit.co.jp/ait/articles/1708/01/news015.htmlより転載。

Contents

- Git, GitHubについて
- **GKVのGit/GitHubの利用**
  - GitHub運用体制
  - 実践：具体例から考える

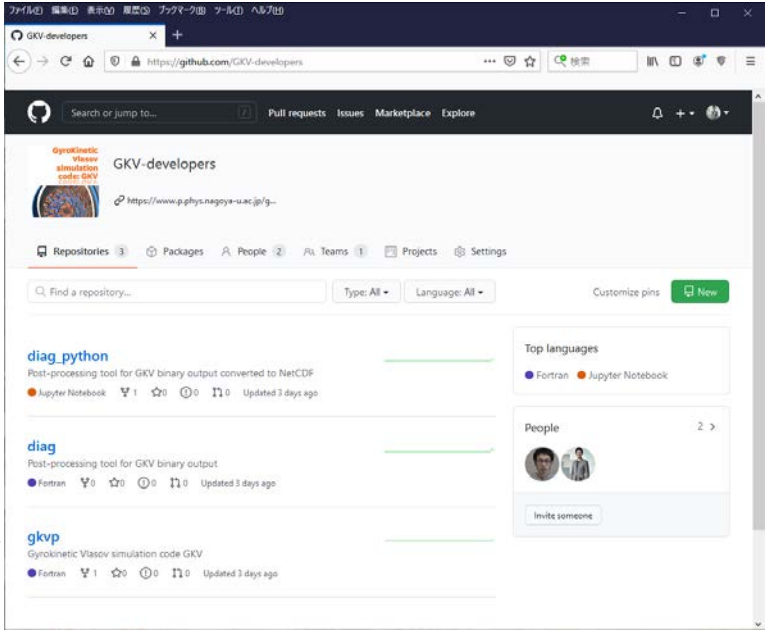
GKVのGit/GitHubの利用

GKVのGitHub運用体制

Organization: GKV-developers

メンバーは以下のいずれかに属する

- Owner・・・渡邉、前山
  管理者権限。リポジトリの作成
  プルリクエストのマージ処理
  適切なタイミングでリリース
- Team/developers・・・石澤、沼波、仲田、朝比
  既存リポジトリに対しブランチの作成
  プルリクエストの発行
- それ以外・・・現状0名
  メンバー外ユーザと同等。閲覧。Issueコメント

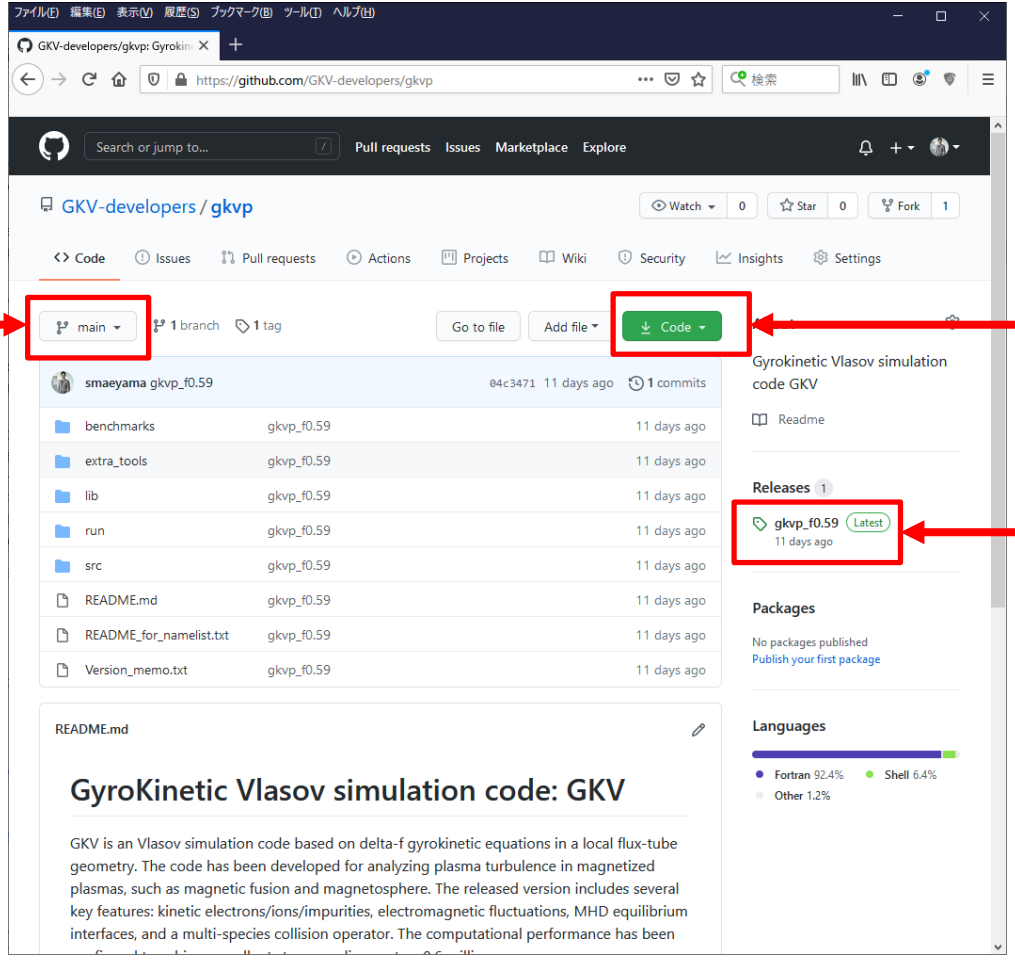


公開リポジトリ

- **gkvp**・・・GKVコード本体
- **diag**・・・ポスト処理プログラム(Fortran版)
- **diag_python**・・・ポスト処理プログラム（Python版）（開発途中）

GKVのGit/GitHubの利用

gkvpリポジトリ画面



ブランチの指定

git cloneコマンドで開発版ブランチをダウンロードする場合はここから

過去のリリースなどはここからダウンロード

（管理者が適当なタイミングでリリースする）

GKVのGit/GitHubの利用：具体例から考える

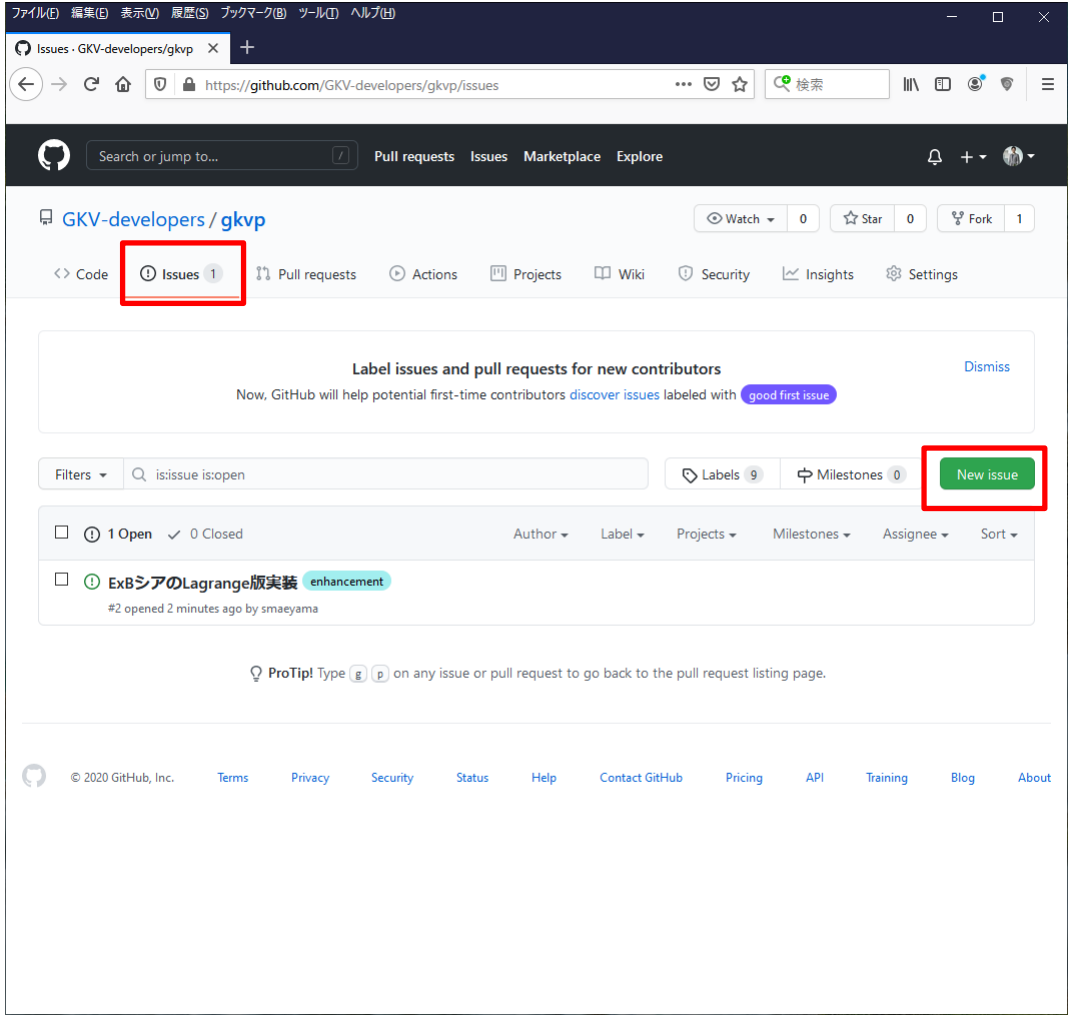
具体例1・・・進行中の開発項目についての情報共有
→ Issueを利用。単なるWikiサーバ扱い。

具体例2・・・最新版コードを利用したい, or, 過去のコードの結果を再現したい
→ GitHubで誰でも最新版にアクセスできる、履歴をたどれるのが利点。
単なるダウンローダ扱い。なんならまだGit不要。

具体例3・・・開発した新機能を共用のリモートリポジトリに反映させたい。
→ 開発方針が決まった段階で、共用のリモートリポジトリに新しいブランチを作って、開発の途中経過もコミット&プッシュ。新機能が動くようになった段階でプルリクエスト作成。mainにマージ。粒度大。

具体例4・・・ちょっとしたバグを見つけた。ちょっとした改良を一部加えた。
→ バグ修正ブランチを作ってプルリクエスト作成。mainにマージ。粒度小。

具体例1・・・Issuesを利用した開発状況の情報共有

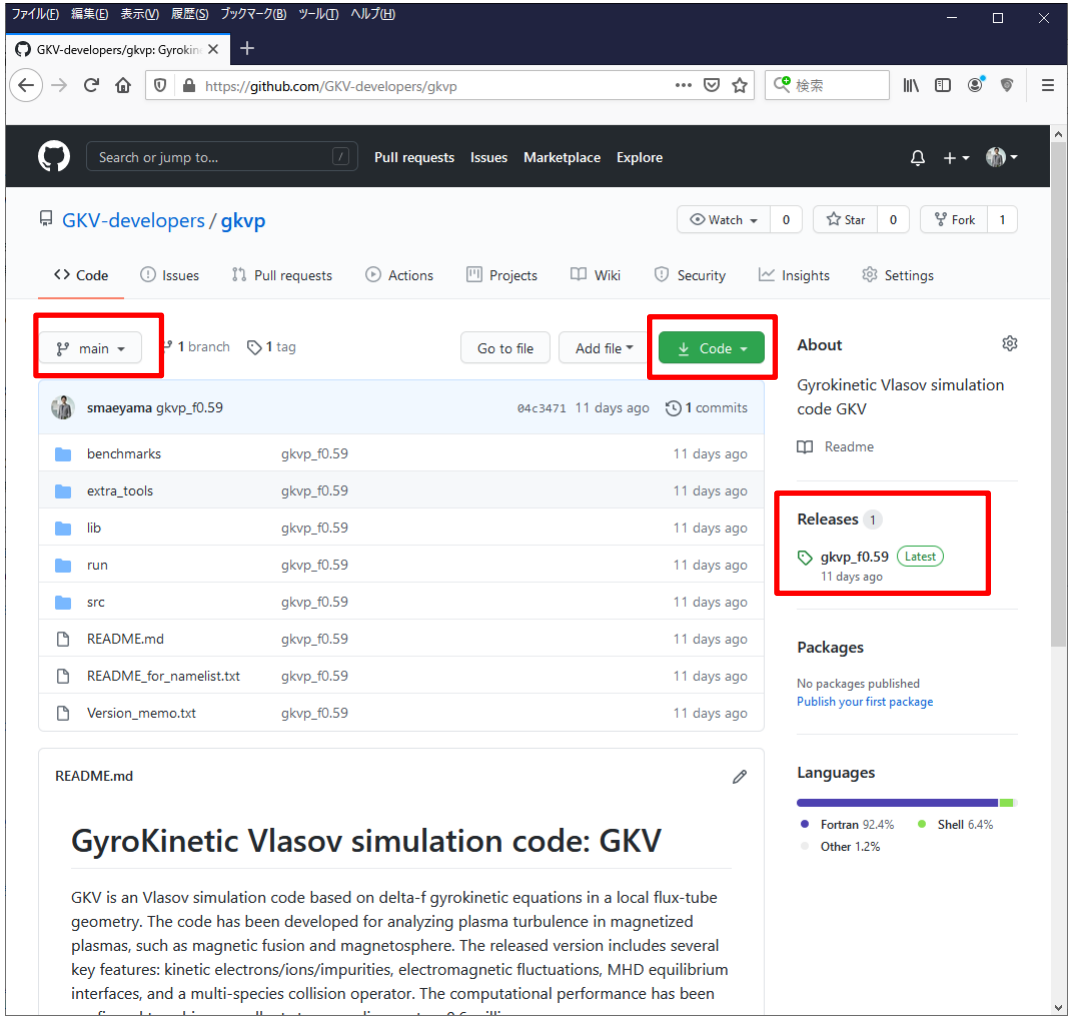


1. Issues タブへ進む。

2. New issue で新規Issue発行。
- **類似のissueが既にないか確認**
- **なるべく内容の分かるタイトル**
- **ラベル付けする**

3. 個別メールのやり取りよりも、履歴がだれでも見られる形で残るので、開発進行状況を整理しやすくなる。

具体例2・・・最新版ブランチまたは過去のリリースのダウンロード



コードをもらってくるだけなら、GitHubからダウンロードするだけでももらってこれます。

1. ダウンロードしたいブランチをプルダウンから指定し、Codeボタンから指定ブランチをダウンロード。

2. あるいは、Releasesボタンから過去のリリースのソースコードをダウンロード。

具体例2・・・最新版ブランチまたは過去のリリースのダウンロード

ターミナルからコマンドラインでGitを操作する方法を説明します。

1. Gitの利用を開始する（このページは初回のみ設定）

あくまでGit/GitHubは共用のソースコード管理のためだけのものと割り切って、
~/git/　　・・・Gitを利用して共用ソースコードのダウンロード
　　あるいはアップロードするためだけのディレクトリ
~/workspace/　　・・・ダウンロードしてきたソースコードをコピーして、
　　実際に計算実行するディレクトリ
というようにディレクトリを分けましょう。

- ローカルリポジトリの作成。
  ```
  $ mkdir ~/git
  $ cd ~/git/
  $ git init
  ```

- 名前とメールアドレスの登録(~/.gitconfigを編集してもよい)
  ```
  $ git config --global user.name “UserA”
  $ git config --global user.email “user_A@p.phys.nagoya-u.ac.jp”
  ```

具体例2・・・最新版ブランチまたは過去のリリースのダウンロード

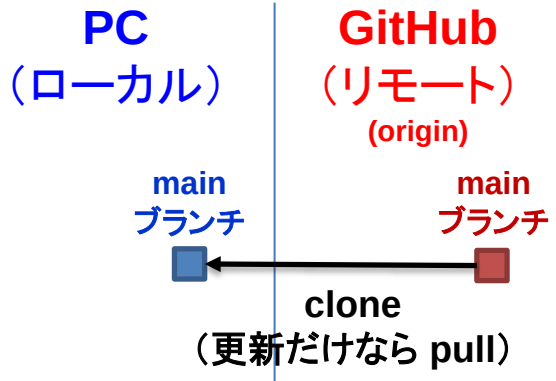
2. リモートリポジトリからソースコードを入手。

- リモートhttps://~にあるmainブランチをローカルにコピー
  $ git clone https://github.com/GKV-developers/gkvp.git
  （なお、リモートリポジトリの確認は $ git remote –v ）

- 計算実行環境にコピー
  $ cp –r * ~/workspace/
  パラメータ変更。計算実行。結果をグラフに。コードを編集。
  → ~/git/以外の場所で何をしても、Gitとは関係ない各自のローカルPC内でのことなので、ご自由にどうぞ。

- リモートリポジトリにアップデートがあったらしいので、自分のソースも更新したい。
  $ git pull origin main
  (origin というのが https://~を指す。リモートoriginにあるmainブランチをローカルのmainにコピーする)

  ※このとき、ローカルでmainを書き換えてると、mergeする前にcommitしろとか、とかいろいろメッセージが出てpullできない。



具体例3・・・開発した新機能を共有リポジトリに反映

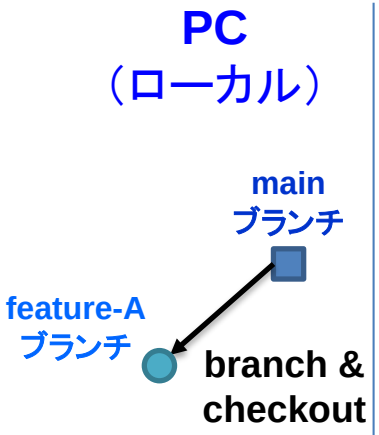
3. 機能を実装するための作業環境を作る

- ブランチを切る。

  $ git branch feature-A

  $ git checkout feature-A

  （なお、$ git branch -a　存在するブランチと作業中ブランチ(*印, HEAD)の確認）



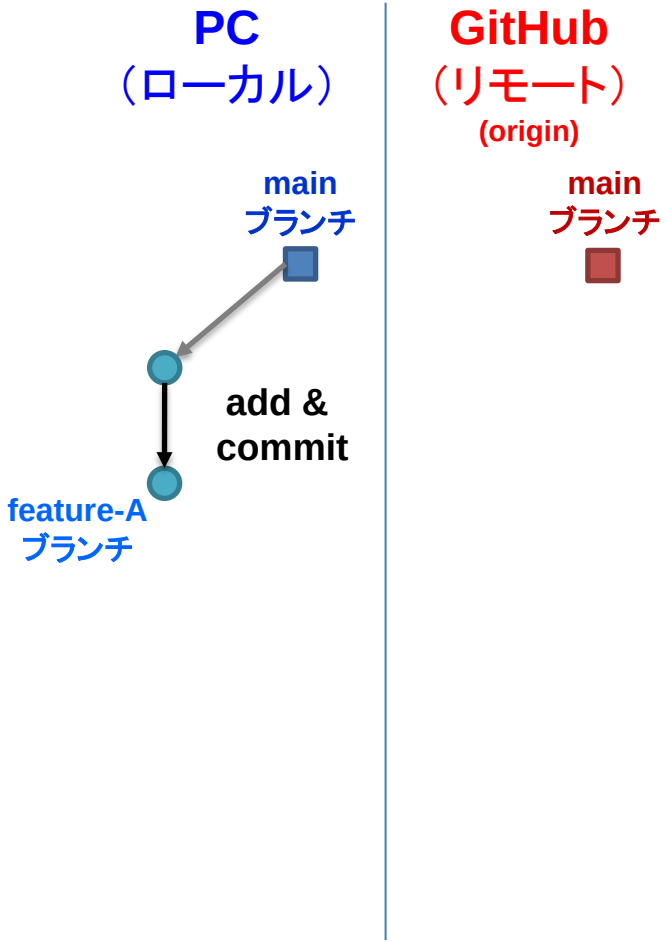
具体例3・・・開発した新機能を共有リポジトリに反映

3. 機能を実装するための作業環境を作る

- ブランチを切る。
  $ git branch feature-A
  $ git checkout feature-A
  （なお、$ git branch -a　存在するブランチと作業中ブランチ(*印, HEAD)の確認）

4. 編集した結果をブランチに記録しておく

- ファイルを編集して機能を実装。
- ブランチに取り込みたいファイルをステージング
  $ git add 編集済ファイル
- ブランチに記録する（まだローカルのみ）
  $ git commit –m “メッセージ”
  （なお、$ git status でステージング・コミット状況確認）



具体例3・・・開発した新機能を共有リポジトリに反映

3. 機能を実装するための作業環境を作る

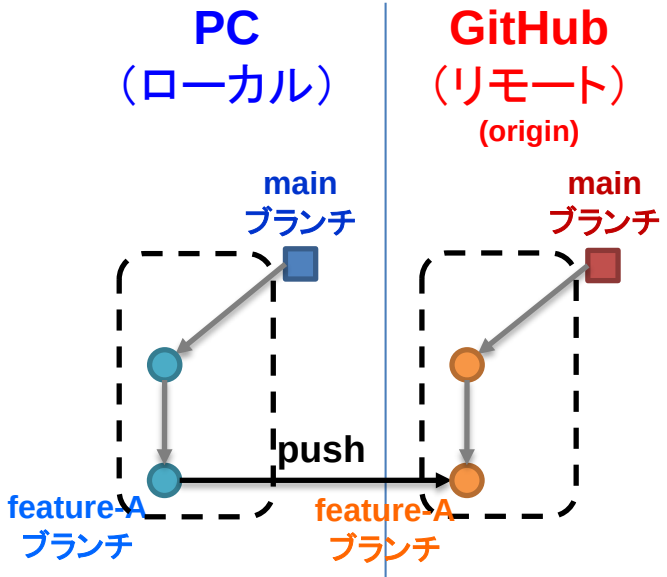
- ブランチを切る。
  $ git branch feature-A
  $ git checkout feature-A
  （なお、$ git branch -a　存在するブランチと作業中ブランチ(*印, HEAD)の確認）

4. 編集した結果をブランチに記録しておく

- ファイルを編集して機能を実装。
- ブランチに取り込みたいファイルをステージング
  $ git add 編集済ファイル
- ブランチに記録する（まだローカルのみ）
  $ git commit –m “メッセージ”
  （なお、$ git status でステージング・コミット状況確認）

5. リモートに同じブランチを作成

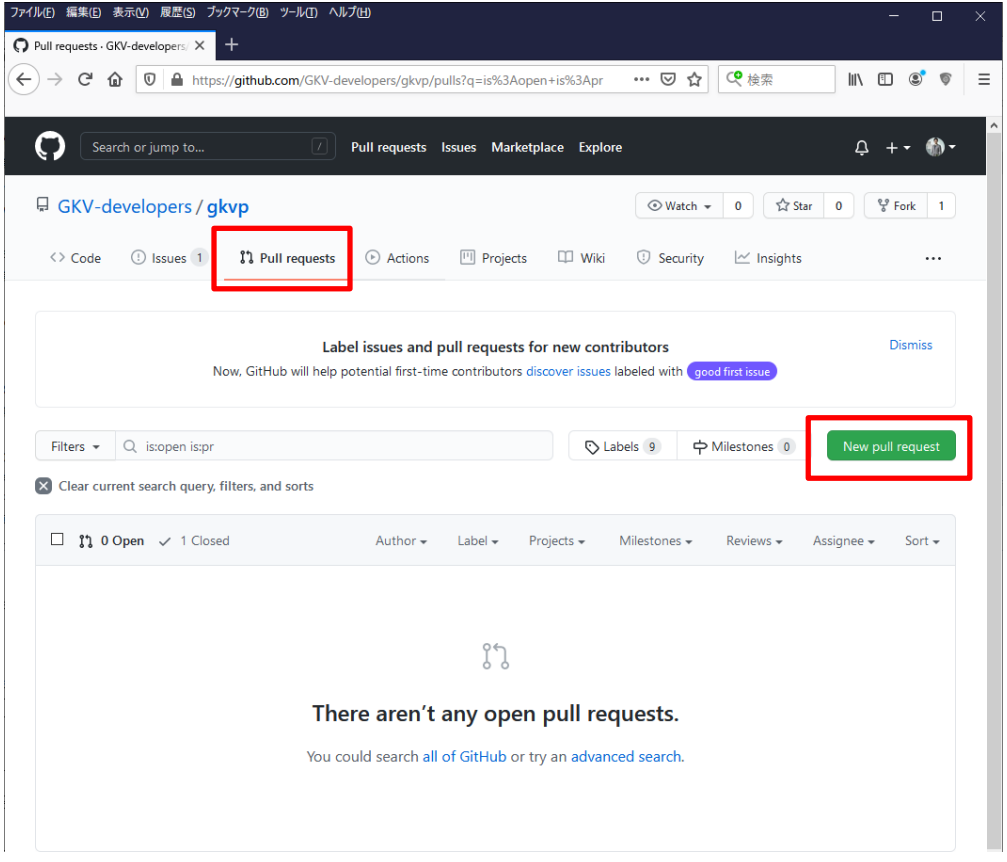
$ git push --set-upstream origin feature-A
(--set-upstreamはリモート側に対応するブランチがない場合のみ)

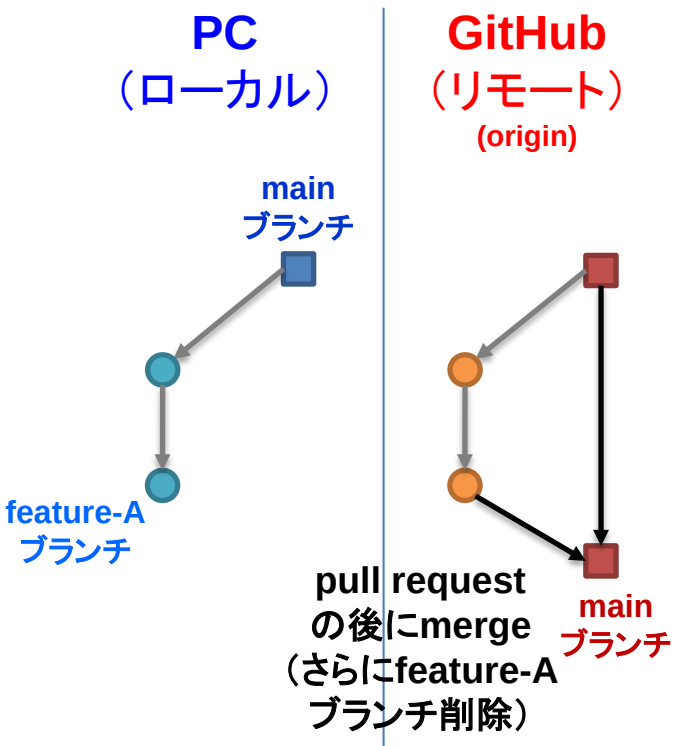


具体例3・・・開発した新機能を共有リポジトリに反映

6. プルリクエスト

- GitHub上でプルリクエストを作成する。
- 管理者側でコードレビュー後にマージします。





具体例3・・・開発した新機能を共有リポジトリに反映

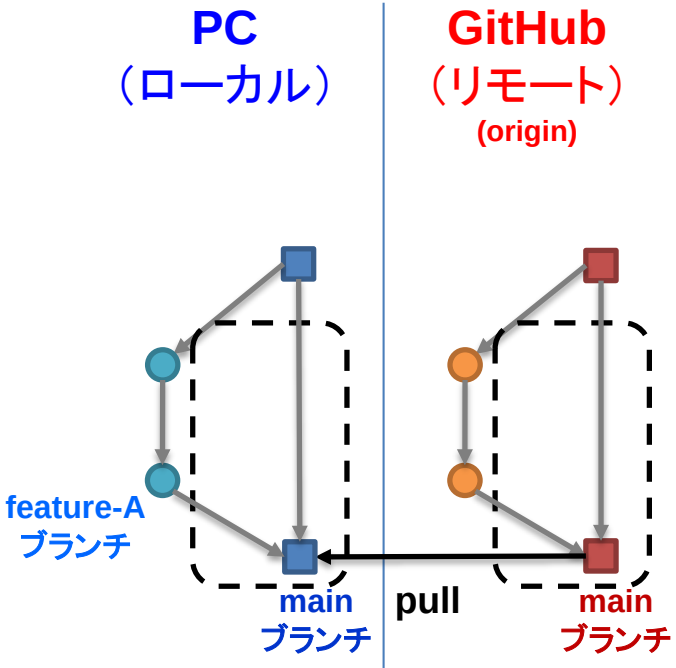
6. プルリクエスト

- GitHub上でプルリクエストを作成する。
- 管理者側でコードレビュー後にマージします。

7. ローカルに最新のmainを反映する

- ローカル更新

  $ git pull origin main



具体例3・・・開発した新機能を共有リポジトリに反映

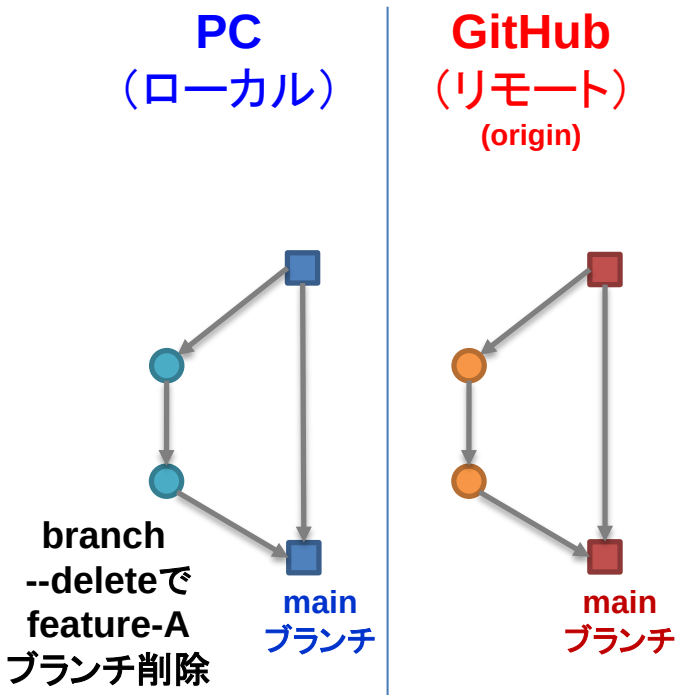
6. プルリクエスト

- GitHub上でプルリクエストを作成する。
- 管理者側でコードレビュー後にマージします。

7. ローカルに最新のmainを反映する

- ローカル更新
  $ git pull origin main
- さらに不要となったブランチ削除
  $ git branch --delete feature-A

※具体例4・・・バグを修正する場合も流れは同じです。
$ git branch bugfix-B
$ git checkout bugfix-B
$ git add バグ修正したファイル
$ git commit –m “メッセージ”
$ git push origin
GitHub上でプルリクエスト作成。



発展・・・競合への対処

8. コードの競合

- 先の例と同様にfeature-Bブランチを作成。

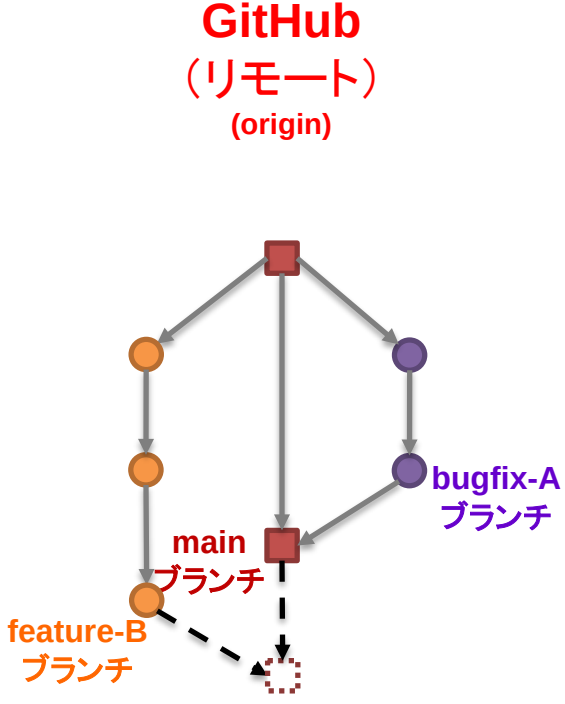
一方その頃、mainのソースコードにバグが見つかった。他の人がbugfix-Aブランチを作ってpull request、すぐにmainに反映された。

- feature-Bブランチをpull requestしたところ、競合が発生。どう対処すべきか。

→いろいろな対処が考えられますが、運用方針はチームごとに違って当然ですので、**管理者・ブランチA開発者・ブランチB開発者で方針を相談しましょう。**

- 大したことないのでその場で修正してmasterにマージしてよさそうか
- いったん別のマージテスト用ブランチパターンA、パターンBを作って、それぞれ動作確認して、合格した方をmasterに改めてマージするか、など

※ネット上の記事やコマンド例を鵜呑みにしない。不明な場合は管理者に聞く。



Tips: チーム内での取り決め

マージ頻度
- 新機能開発は、検証が済んでから。粒度大
- ちょっとしたデバッグやちょっとした拡張はこまめに。粒度小

リリース頻度
- 基本的には、ソースコードへの大きめの変更の段階でリリース。
- ちょっとしたデバッグや拡張程度ではリリースしない。(例:富岳用Makefile加えた)
- とはいえ、デバッグ等が積み重なったらある段階でリリースする。
- GKVホームページからダウンロードできるソースコードも稀に更新。

ブランチ命名規則
- 誰が作ったブランチか一目でわかるようにする。
- どういう目的のブランチか分かるようにする。

良い例: nunami_sxaurora_optimize, nakata_exb_shear_lagrange,
maeyama_bugfix_trans
悪い例: 2020_11_13, f0.59_rev1