Fork and Pull Request Workflow

Susam Pal

Version 0.6.0 (2018-11-19)

This document describes how developers may contribute pull requests to an upstream repository and how upstream owners may merge pull requests from contributors according to the very popular fork and pull request workflow followed in many projects on GitHub.

The most recent version of this document is available at git.io/gitpr.

Contents

1 Introduction

Every project has a main development branch where the developers push commits on a day-to-day basis. Usually, the main development branch is `master` but some projects choose to have `develop` or `trunk` or another branch for day-to-day development activities. We refer to this main development branch as the *main development branch* throughout this document to keep the text general. However in the command examples and ASCII-diagrams, we use `master` as an example of the main development branch.

We use the following placeholders in the command examples and ASCII-diagrams in this document:

- `GITHUB`: `github.com` or domain name/hostname of your private GitHub Enterprise system.
- `USER` or `CONTRIBUTOR`: The user that forks an upstream repository, creates pull requests, and sends them to the upstream repository.
- `UPSTREAM-OWNER`: Owner of the upstream repository. This is the name of the user or organization that merges pull requests into the upstream repository.
- `REPO`: Repository name.
- `FILES`: One or more filenames to be staged for a commit.
- `TOPIC-BRANCH`: Feature-specific or bug-specific branch where a contributor develops her or his contribution. This is referred to as the *topic branch* in the text.

These placeholders should be substituted with appropriate values while executing the commands presented in this document.

Beginners to this workflow should always remember that a Git branch is not a container of commits, but rather a lightweight moving pointer that points to a commit in the commit history.

```
A---B---C
        ↑
     (master)
```

When a new commit is made in a branch, its branch pointer simply moves to point to the last commit in the branch.

```
A---B---C---D
            ↑
         (master)
```

A branch is merely a pointer to the tip of a series of commits. With this little thing in mind, seemingly complex operations like rebase and fast-forward merges become easy to understand and use.

The next section, *Quick Reference*, provides a brief summary of all the frequently used commands involved in creating and merging pull requests.

2 Quick Reference

Here is a brief summary of all the commands used to create and merge pull requests in this document. This section serves as a quick reference.

```
# CREATE PULL REQUEST
# ===================
# Fork upstream and clone your fork.
git clone https://GITHUB/USER/REPO.git
cd REPO
git remote add upstream https://GITHUB/UPSTREAM-OWNER/REPO.git
git remote -v

# Work on pull request in a new topic branch.
git checkout -b TOPIC-BRANCH
git add FILES
git commit
git push origin TOPIC-BRANCH

# Go to your fork on GitHub, switch to the topic branch, and
# click *Compare & pull request*.

# Keep your fork's main development branch updated with upstream's.
git checkout master
git pull upstream master
git push origin master

# Amend last commit (optional).
git add FILES
git commit --amend

# Rebase topic branch on the main development branch (optional).
git checkout TOPIC-BRANCH
git rebase master

# Edit commits, e.g., last 3 commits in topic branch (optional).
git checkout TOPIC-BRANCH
git rebase -i HEAD~3

# Force push rebased/edited commits to the pull request (optional).
git push -f origin TOPIC-BRANCH

# Delete topic branch branch after pull request is merged.
git checkout master
git branch -D TOPIC-BRANCH
git push -d origin TOPIC-BRANCH
```

```
# MERGE PULL REQUEST (WITHOUT MERGE COMMIT)
# =========================================
# Clone upstream repo.
git clone https://GITHUB/UPSTREAM-OWNER/REPO.git
cd REPO

# Keep the local main development branch up-to-date.
git checkout master
git pull

# Pull changes in pull request into a temporary branch.
git checkout -b pr
git pull https://GITHUB/CONTRIBUTOR/REPO.git TOPIC-BRANCH

# If the above command does not create a new merge commit, ignore
# this comment and follow the remaining commands after this comment.
#
# If the above command creates a new merge commit, first consider
# that it is okay to have a merge commit. However, if you want to
# get rid of it, choose exactly one of the following three options:
#
#   - Request the contributor to rebase the topic branch on the main
#     development branch. After the contributor has done so, delete
#     the temporary branch, start over, and pull the changes again.
#     (Commands: git checkout master; git branch -D pr)
#
#   - Go to the pull request page on GitHub, click on the dropdown
#     menu next to 'Merge pull request', select 'Rebase and merge',
#     and click 'Rebase and merge'.
#
#   - Rebase the topic branch on the main development branch. Then
#     follow the commands below. Finally close the pull request on
#     GitHub manually. (Command: git rebase master)

# Merge pull request and delete temporary branch.
git checkout master
git merge pr
git branch -d pr

# Push the updated main development branch to upstream repo.
git push origin master
```

```
# MERGE PULL REQUEST (WITH MERGE COMMIT)
# ======================================
# Clone upstream repo.
git clone https://GITHUB/UPSTREAM-OWNER/REPO.git
cd REPO

# Keep the local main development branch up-to-date.
git checkout master
git pull

# Pull changes in pull request into a temporary branch.
git checkout -b pr
git pull https://GITHUB/CONTRIBUTOR/REPO.git TOPIC-BRANCH

# Merge pull request and delete temporary branch.
git checkout master
git merge --no-ff pr
git branch -d pr

# Push the updated main development branch to upstream repo.
git push origin master
```

The next two sections, *Create Pull Request* and *Merge Pull Request*, elaborate these commands in detail.

3 Create Pull Request

This section is meant for developers who contribute new commits to the upstream repository from their personal fork.

3.1 Fork and Clone

On GitHub, fork the upstream repository to your personal user account.

Then clone your fork from your personal GitHub user account to your local system and set the upstream repository URL as a remote named `upstream`.

```
git clone https://GITHUB/USER/REPO.git
cd REPO
git remote add upstream https://GITHUB/UPSTREAM-OWNER/REPO.git
git remote -v
```

Now the remote named `upstream` points to the upstream repository and the remote named `origin` points to your fork.

3.2 Work on Pull Request

Work on a new pull request in a new topic branch and commit it to your fork. Remember to use a meaningful name instead of `TOPIC-BRANCH` in the commands below.

```
git checkout -b TOPIC-BRANCH
git add FILES
git commit
git push origin TOPIC-BRANCH
```

Create pull request via GitHub web interface as per the following steps:

- Go to your fork on GitHub.
- Switch to the topic branch.
- Click *Compare & pull request.*
- Click *Create pull request.*

Wait for an upstream developer to review and merge your pull request.

If there are review comments to be addressed, continue working on your branch, commiting, optionally rebasing, amending, squashing, and dropping them, and pushing them to the topic branch of `origin` (your fork). Any changes to the topic branch automatically become available in the pull request.

In the fork and pull request workflow, a contributor should never commit anything to the main development branch of personal fork. This makes it very easy to keep the main development branch of your fork in sync with that of the upstream repository. This is explained in the next subsection.

3.3 Keep Your Fork Updated

As new pull requests get merged into the upstream's main development branch, the main development branch of your fork begins falling behind it.

The commands below show how to update your fork's main development branch with the new commits in the upstream's main development branch.

```
git checkout master
git pull upstream master
git push origin master
```

The `git pull` command above simply fast-forwards the main development branch from an earlier commit to the last commit in the upstream's main development branch.

When your main development branch falls behind the upstream's main development branch, the upstream's main development branch extends linearly from the last commit in your main development branch, provided that there are no additional commits to your main development branch that has caused it to diverge.

```
              E---F---G  (TOPIC-BRANCH)
             /
A---B---C---D  (master)
             \
              H---I---J  (upstream/master)
```

With such a commit history, when the upstream's main development branch is merged into your main development branch, the merge is done by simply fast-forwarding the pointer of your main development branch to the last commit in the upstream's main development branch.

```
              E---F---G  (TOPIC-BRANCH)
             /
A---B---C---D
             \
              H---I---J  (upstream/master, master)
```

After the merge is complete, the upstream's main development branch and your main development branch point to the same commit.

3.4 Amend Last Commit

This is an optional step to rework on the last commit. After commiting some work, one may realize that some files need to be modified or the commit message needs to updated.

```
     E---F---G  (TOPIC-BRANCH)
    /
A---B---C---D   (master)
```

Git allows us to pick the last commit, modify the changes made in that commit including the commit message, and reapply the changes along with the modifications as a new commit that replaces the last commit.

```
     E---F---G' (TOPIC-BRANCH)
    /
A---B---C---D   (master)
```

To do so, first rework on the files and make the necessary changes. Then add, remove, move, or rename files to stage them for commiting. Finally, amend the last commit. The commit message may be modified when the editor comes up to show the commit message.

```
git add FILES
git commit --amend
```

Although the above example shows only the `git add` command to add new or modified files for commiting, one may amend the last commit by removing, moving, and renaming files with the `git rm` and `git mv` commands before amending the last commit.

To update only the commit message without changing files, run only `git commit --amend`.

3.5 Rebase Commits

This is an optional step to keep the commit history as linear as possible.

The main development branch may have diverged since the topic branch was created.

```
     E---F---G (TOPIC-BRANCH)
    /
A---B---C---D  (master)
```

It may be a good idea to move the commits in the topic branch and place them on top of the main development branch, so that the topic branch extends linearly from the last commit in the main development branch.

```
              E'--F'--G' (TOPIC-BRANCH)
             /
A---B---C---D  (master)
```

The following commands show how to rebase the topic branch on the main development branch.

```
git checkout TOPIC-BRANCH
git rebase master
```

3.6 Edit Commits

This is an optional step to keep the commit history clean and concise.

A pull request may contain multiple commits. Sometimes one may need to amend a commit that is not the last commit in the pull request. Commits prior to the last commit can be amended with the interactive rebase command.

After developing the required feature or bug-fix in a topic branch, the developer or a reviewer may notice issues in the work that need to be addressed before the pull request can be merged into the upstream repository. This may lead to multiple new commits in the topic branch that should ideally have been part of the first commit that implemented that feature or bug-fix.

```
     E---F---G (TOPIC-BRANCH)
    /
A---B---C---D  (master)
```

In such cases, it may be a good idea to squash multiple commits in the topic branch into a single coherent commit with all changes for the feature or bug-fix being developed.

```
     E'        (TOPIC-BRANCH)
    /
A---B---C---D  (master)
```

The following example shows how to amend the last 3 commits.

```
git checkout TOPIC-BRANCH
git rebase -i HEAD~3
```

This brings up an editor with three lines for the last 3 commits ordered from earliest to last followed by instructions on how to edit these lines to amend the commits.

For example, to squash all three commits into one, leave the first commit untouched, replace `pick` with `squash` in the next two lines, save, and quit the editor. This brings up the editor again. Clean up the commit message, save, and quit the editor.

Apart from `pick` and `squash`, there are other operations such as `reword`, `edit`, `drop`, etc. to amend or drop commits. Follow the instructions that appear in the editor to use them. When a rebase operation involves multiple steps, Git walks you through each step by offering suggestions and commands that you need to use in each step.

3.7 Force Push

The steps in the last three sections overwrite the history of the branch. If these steps are performed after the pull request branch has already been pushed to GitHub, then it is necessary to use the `-f` or `--force` option to push the overwritten history to GitHub.

```
git push -f origin TOPIC-BRANCH
```

The `-f` (force) option in the `git push` command is necessary only if you are pushing to an already existing pull request branch because doing so overwrites the history of the branch. Normally, overwriting history is strictly discouraged but this is one of the rare scenarios where it is safe to overwrite the commit history because the commits are being pushed to a personal branch in a personal fork without affecting the upstream repository.

3.8 Delete Branch

Once the upstream developer merges your pull request, you may delete the topic branch from your local system as well as from your fork.

```
git checkout master
git branch -D TOPIC-BRANCH
git push -d origin TOPIC-BRANCH
```

The next section, *Merge Pull Request*, explains how an upstream owner can merge pull request from contributors into the upstream repository.

4 Merge Pull Request

This section is meant for lead developers who own the upstream repository and merge pull requests from contributors to it.

There are two popular methods to merge commits: one that does not introduce an additional merge commit, and another that does. Both are perfectly acceptable. Both are discussed below.

Which method you choose depends on whether you want to maintain a concise commit history consisting only of development commits or if you want to introduce additional merge commits for every merge into your commit history.

4.1 Without Merge Commit

Clone the upstream repository to your local system.

```
git clone https://GITHUB/UPSTREAM-OWNER/REPO.git
cd REPO
```

If the repository was already cloned to the local system earlier, then ensure that the local main development branch is up-to-date with that in the upstream repository.

```
git checkout master
git pull
```

Create a temporary branch (`pr` for example) to pull the contribution (pull request) from the CONTRIBUTOR's branch in it.

```
git checkout -b pr
git pull https://GITHUB/CONTRIBUTOR/REPO.git TOPIC-BRANCH
```

If the main development branch has diverged from the branch in the pull request, the above command creates a new merge commit to merge the pull request into the temporary branch.

Note: It is okay to have a merge commit while merging pull requests. There is nothing wrong about it. If you are comfortable having a merge commit for the pull request, skip the list of three points below and continue with merging the pull request.

If you really want to get rid of the additional merge commit, follow exactly one of these options:

- Request the contributor to rebase the topic branch on the main development branch. See the *Rebase Commits* section for more details. After the contributor has rebased the topic branch on the main development branch, delete the temporary branch.

  ```
  git checkout master
  git branch -D pr
  ```

  Then start over and pull the pull request again. In fact, if at any time the contributor rebases the pull request, or edits, squashes, amends, or drops commits in the pull request, delete the temporary branch and start over, and pull the pull request again.

- Go to the pull request page on GitHub, click on the dropdown menu next to the *Merge pull request* button, select *Rebase and merge*, and click *Rebase and merge*.

  Caveat: The rebase operation rewrites the commit history of the pull request submitted by the contributor. As a result, it modifies the commit hash, committer, and commit date of each change in the pull request. The committer and commit date fields are separate from the author and author date fields. Use `git log --format=fuller` to see all four fields.

  This caveat is usually not a major problem because the author and author date still remain intact. Further since the rebase and merge is done via GitHub, it closes the pull request successfully and sets its status to *Merged*.

  However if you are not comfortable with the committer details changing, then do not choose this option and choose the previous option instead.

- Rebase the pull request on the main development branch as explained in the *Rebase Commits* section.

  Caveat: In addition to the caveat mentioned in the previous point, in this case, GitHub remains oblivious of the changed commit hashes because the rebase operation is done locally, not via GitHub. Therefore after the merged pull request is pushed to the upstream repository, GitHub does not close the pull request.

  The pull request on GitHub can then be closed manually but its status would not appear as *Merged*. It would appear as *Closed* instead. This can be confusing and misleading. Therefore this option is not recommended. Choose one of the previous two options instead.

After sufficient testing, merge the commits in the pull request into the main development branch and remove the pull request branch.

```
git checkout master
git merge pr
git branch -d pr
```

Finally, push the current state of the main development branch to the upstream repository.

```
git push origin master
```

The `git merge` command above simply fast-forwards the main development branch from an earlier commit to the last commit in the pull request thereby making both the main development branch and the pull request branch point to the same commit. This achieves the merge from the pull request branch into the main development branch without creating a new merge commit.

After a pull request branch is rebased on the main development branch, the pull request branch becomes a linear extension of the main development branch.

```
             E'---F'---G' (pr)
            /
A---B---C---D (master)
```

With such a commit history, when the pull request branch is now merged into the main development branch, the merge is done by simply fast-forwarding the pointer of the main development branch to the last commit in the pull request branch.

```
             E'---F'---G' (pr, master)
            /
A---B---C---D
```

4.2 With Merge Commit

Clone the upstream repository to your local system.

```
git clone https://GITHUB/UPSTREAM-OWNER/REPO.git
cd REPO
```

If the repository was already cloned to the local system earlier, then ensure that the local main development branch is up-to-date with that in the upstream repository.

```
git checkout
git pull
```

Create a temporary branch (`pr` for example) to pull the contribution (pull request) from the CONTRIBUTOR's branch in it.

```
git checkout -b pr
git pull https://GITHUB/CONTRIBUTOR/REPO.git TOPIC-BRANCH
```

If the contributor adds new commits to the pull request later, then run these commands again to pull the new commits.

After sufficient testing, merge the commits in the pull request into the main development branch.

```
git checkout master
git merge --no-ff pr
git branch -d pr
```

Finally, push the current state of the main development branch (with the commits from the pull request in it) to the upstream repository.

```
git push origin master
```

The `--no-ff` (no fast-forward) option in the `git merge` command ensures that a merge commit is always created even when a fast-forward merge is possible.

```
              E'---F'---G' (pr)
             /           \
A---B---C---D-------------H (master)
```

5 Nifty Commands

This is a bonus section that describes a few aliases and commands that may be useful during day-to-day development activities.

5.1 Pretty Log

The following commands create aliases to run `git log` with various subsets of {`--pretty`, `--graph`, `--all`} options to display commit logs in a compact form, i.e., one line per log.

```
# Define aliases.
git config --global pretty.fmt '%C(auto)%h %C(magenta)%ad %C(cyan)%an%C(auto)%d %s'
git config --global alias.lga 'log --pretty=fmt --date=short --graph --all'
git config --global alias.lg  'log --pretty=fmt --date=short --graph'
git config --global alias.la  'log --pretty=fmt --date=short --all'
git config --global alias.ll  'log --pretty=fmt --date=short'
git config --global alias.lf  'log --pretty=fuller --stat'

# Use aliases.
git lga
git lg
git la
git ll
git lf
```

5.2 Staged Diff

While `git diff` shows the unstaged changes in the working directory, it is necessary to use the `--cached` option with `git diff` to see the changes staged for the next commit. The following commands create convenient aliases for this option.

```
# Define aliases.
git config --global alias.diffc "diff --cached"
git config --global alias.dc "diff --cached"

# Use aliases.
git diffc
git dc
```

5.3 Branch Listing

The following commands provides aliases to list branches with verbose information. The second alias includes remote branches too in the output.

```
# Define aliases.
git config --global alias.br "branch -vv"
git config --global alias.brr "branch -vva"

# Use aliases.
git br
git brr
```

5.4 More Aliases

Here are a few more commands to define aliases for very frequently used commands.

```
# Define aliases.
git config --global alias.co "checkout"
git config --global alias.cob "checkout -b"
git config --global alias.ca "commit --amend"

# Use aliases.
git co
git cob
git ca
```

5.5 Merge Base

Find a common ancestor of two branches with this command. It helps to find the commit after which two branches began diverging.

```
git merge-base upstream/master TOPIC-BRANCH
```

5.6 Commit Partial Changes

Sometimes when you are in the zone, you may make large sweeping changes to a file. However it is a good practice to create separate and small commits for separate concerns. To select some changes in a file while ignoring other changes in the same file for the next commit, stage the changes interactively with this command.

```
git add -p
```

This command shows every change hunk one by one. Enter `y` to stage a hunk for the next commit, `n` to ignore it, and `s` to split the hunk into multiple smaller hunks. Enter `?` to print help message about each interactive option that can be entered.

5.7 Some Git Wisdom

Enter one of the commands below in your command shell depending on your operating system, or just open the URL with your web browser.

```
open https://xkcd.com/1597/
xdg-open https://xkcd.com/1597/
start https://xkcd.com/1597/
```

6 License

7 Support

To report bugs, suggest improvements, or ask questions, create issues.

To contribute improvements to this document, fork this project and *create pull request!* ;-)