Working doc. Git For Team.

Writted by Hong Le Nguyen, last update 10.2024

1)
Branching Model

Git's branching model allows developers to work in isolated branches e.g. features or bug fixes, reducing the risk of conflicts and ensuring code stability.

There are three types of branches when using Git as a team collaboration tool:

- Master branch (often called main or stable branch)
- Staging branch (sometimes called development branch)
- Feature branches

while the master branch and the staging branch are created once and stay in place throughout the whole process of the project development, the feature branches are created for the period during which a certain feature of the project is developed and later get merged into the staging branch.
Each team member should be working on their dedicated feature branch.

2)
Working with branch

Branch terminology ---

A local branch --

exists only on your local machine *that you can work in and add commits to*.
You can list local branches with

```
git branch
```

A remote branch --

is a branch on a remote location (in most cases origin).

A remote tracking branch --

is *a **local** copy of a remote branch* at a given time.

This branch should never be edited. But You can get or update remote tracking branches to be in sync with the corresponding remote branches using git fetch or git pull. An Example,
fetching all remote tracking branches from the remote repository

```
git fetch origin
```

or fetching the remote tracking branch "origin/myNewBranch" from remote branch "myNewBranch"

```
git fetch origin myNewBranch
```

Remote tracking branches can be listed and typically look like "origin/master", "origin/myNewBranch", …

```
git branch -r
```

A local tracking branch --

*is a **local** branch* that tracks a remote tracking branch. This is so that you can push/pull commits to/from the remote branch.

If a remote tracking branch is available (after operations as e.g. git push origin <branch>, or git fetch origin <branch>), we can retrieve *the local tracking branch*

```
git checkout <branch-name>
```

Example, get the local tracking branch of the remote tracking branch "origin/myNewBranch"

```
git checkout myNewBranch
```

How to create a feature branch ? --

Create *local branch* named "myNewBranch" *from current-branch* e.g. staging

```
// git checkout staging
git branch myNewBranch
git checkout myNewBranch    // switch to the new branch
```

You can create a new branch and immediately switch to it.

git checkout -b myNewBranch

Add some changes and commits to the local branch

```
git add .
// git add src/Dao/BaseDao.php
git commit -m "add class BaseDao by dev-1"
```

You can push the newly created local branch myNewBranch to the remote "origin".

git push -u origin myNewBranch

So now a new *remote branch* named "myNewBranch" is created on the remote machine named "origin".
The -u option is needed to use *git push* and *git pull* without specifying an <remote> <branch>.

When myNewBranch is pushed to origin using the command above, *a remote tracking branch* named "origin/myNewBranch" is created on your machine. It tracks the state of the remote branch "myNewBranch" on origin.

You can get *the local tracking branch* "myNewBranch" to track the remote tracking branch "origin/myNewBranch" and add changes/commits to that before push to server.

```
git checkout myNewBranch
// git add .
// git commit -m "some messages"
```

How to synchrony local and remote branch "myNewBranch"? --

The Command synchronizes local and remote branch "myNewBranch"

git pull

updates remote tracking branch "origin/myNewBranch" to be in sync with the remote branch "myNewBranch" on "origin" and pulls these new commits from "origin/myNewBranch" to local branch "myNewBranch" which you just switched to.

git pull

The git pull command **fetches** remote changes and **merges** them with your local changes. It create a new commit into their local repository.

git pull --rebase

This command first fetches the remote changes, then rebases your local commits on top of the remote branch. Rebasing creates a linear commit history that is easier to understand and maintain.

Git Merge vs. Rebase

Git provides two main ways to integrate changes from one branch into another: git merge and git rebase. While both commands achieve the same goal, they have different implications for the commit history and workflow.

The choice between git merge and git rebase depends on your team's preferences and the specific situation. In general, it's recommended to use git rebase for local, private branches
and git merge for integrating public branches or when collaborating with others.

To delete a local branch, use

```
git branch -d myNewBranch
```

Deleting remote branch in Git

```
git push --delete <remote> <branch> // e.g. git push --delete origin myNewBranch
```

3)
Keep a branch up-to-date

Regardless of the branch (staging/feature) you are working on, sometimes you need *to update your local version of this branch with the changes from the remote branch*, because someone else pushed updates to it (the remote branch).

git pull --rebase origin your-branch-name

If you have changed a file that has been changes remotely too, it can happen that you run into a merge conflict during the rebase. If this happens, resolve the file's conflicts and continue on the command line the following way:

```
git add .
git rebase --continue
```

If your pull rebase goes wrong, you can always abort it:

```
git rebase --abort.
```

Note: *Before* you go through with updates, we suggest you follow these steps:

```
git fetch origin   // If one of your team members has created a branch that is not available to you locally
git checkout your-branch-name  // Navigate to the branch with
git commit   // If you made any changes to the files, do // or
git stash      // (this option stores your changes so you can apply them again in later stages)
```

4)
keep a feature branch up-to-date with staging

Once you started to work on a feature branch, you may want to keep the branch up to date with the staging branch in case anyone else merged their feature branches into staging.

So when do you want to keep your feature branch up to date with staging:
You wish to make /a pull request (PR) of your feature branch/ to merge it into staging, with all the recent changes from staging included, while avoiding merge conflicts.
You wish to include an update from the staging (e.g. hotfix, library upgrade, dependent feature from someone else) and continue working on the feature branch with no blocking issues.

```
git checkout your-branch-name
then follow 3): keep a branch up-to-date

git checkout staging
then follow 3): keep a branch up-to-date

git checkout your-branch-name
git rebase staging    // merges all your changes from your feature branch on top of the staging branch
```

```
// git checkout your-branch-name
// then follow 3): keep a branch up-to-date ?

git push origin your-branch-name  // pushes the updated feature branch to your remote repository
```

In case you have previously pushed your branch to the remote repository, you'll have to force push your changes because the history of your branch changed during the rebase with the staging branch:

```
git push -f origin your-branch-name
```

But be careful with a force push: If someone else made changes in between on this branch, a force push will forcefully overwrite all these changes.

5)

Merge your feature branch into the staging branch

If everything is fine with your feature branch, continue on the command line:

```
git checkout staging
git merge --no-ff your-branch-name   // merges your feature branch into staging
git push origin staging
```

Note:
A pull request (PR) will merge all your feature branch's changes into the staging branch.
Even though a merge would be possible without a PR, a PR enables other people to review your feature on platforms like GitHub or Gitlab.
That's why a best practice would be to open a PR for your feature branch once you finished the implementation of the actual feature on this branch and pushed everything to your remote repository. The git workflow looks as follows:

```
follow: How to keep a feature branch up-to-date with staging?
do: Open Pull Request on GitHub/Gitlab/... for your feature branch.
wait: CI/CD, discussion/review, approval of team members.
optional: Push more commits to your remote branch if changes are needed due to discussion.
```

Working doc. Git Basic.

Writted by Hong Le Nguyen, last update 10.2024

1)

Creating a local repository ---

git init

Cloning an existing remote repository as a new local repository

git clone <url>

Check the status of a repository

git status

Repository specific name and e-mail:

git config user.name “My Name”

git config user.email myemail@myserver.com

2)

Staging and Commit ---

Staging is a preparation for the next commit

Add a file for staging (also for tracking if it is not yet tracked):

git add <filename>

git reset <filename> // undo

After some changes

git add .

Deleting local branch

git branch -d <branch> // e.g. git branch -d feature-2

Commit

create a local committed *snapshot* from *the current stage*

git commit -m "<commit_message>"

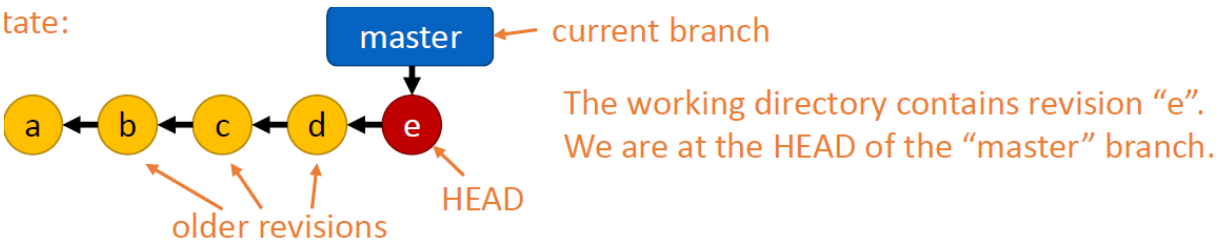
3)

Branch ---

A branch is an independent line of development, *a series of committed snapshots*

snapshots/revisions are identified by their *SHA-1 checksums*

the currently checked-out snapshot is called *HEAD*



We use a branch for e.g. bugfix or feature without affecting other branches

list commits in a branch

git checkout <branch-name>

git log --oneline

List all branches

git branch

Create a new branch (does not switch to the branch!):

git branch <branch-name>

Switch to a branch:

git checkout <branch-name>

Create a new branch and switch directly to it

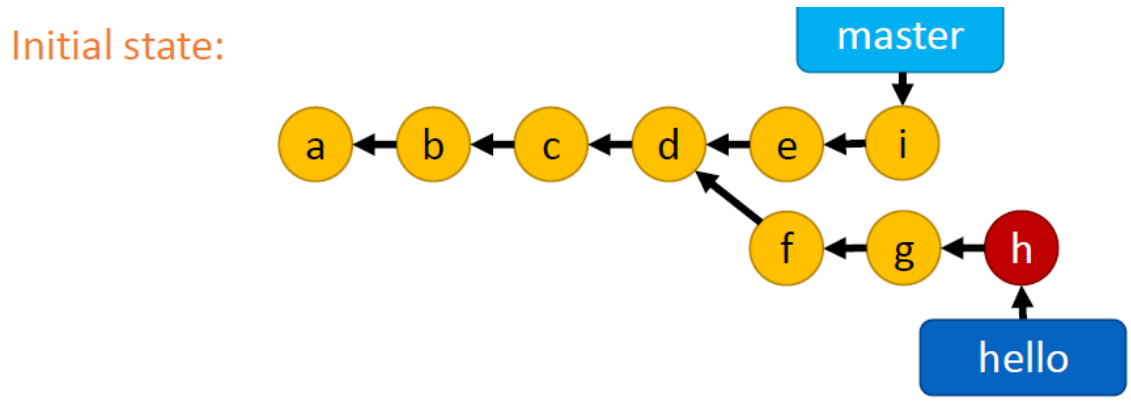
git checkout -b <branch-name>

Example :

```
git branch hello        // create a new branch from current-branch e.g. master
git checkout hello      // go to new branch
git checkout -b hello   // shortcut
```

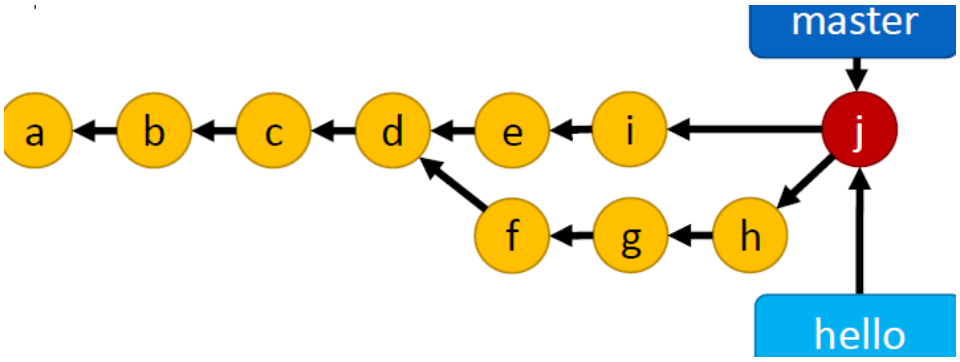
Merging vs. rebasing ---

The first thing to understand about git rebase is that it solves the same problem as git merge.
Both of these commands are designed *to integrate changes from one branch into another branch*—they just do it in very different ways.



Merge : add changes from another branch to the current branch
After we are finished with the hello feature, we can merge it (hello branch) to the master branch:

git checkout master
*git **merge** hello*



We can *delete* the hello branch if we don't need it anymore. Its commits are available from the master branch:

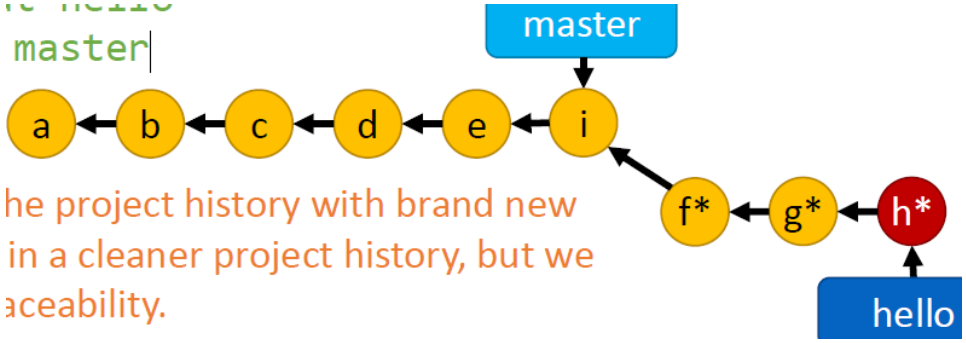
git branch -d hello

or git branch -d <branch-name>

Rebase: add changes from our current branch to the other branch

git checkout hello
*git **rebase** master*

Be careful: *never rebase public branches (e.g. master) on top of your private branch!*
It changes the history of the public branch, and other developers will be confused.



4)
Fetch a branch from the remote repository (downloads all commits and files)

remote branch vs local branch (tracking branch) – as an example

```
--A---B-----  origin/master (remote branch)
       \
        o---  master (local tracking branch)
```

Fetch a branch from the remote repository
git fetch <remote> <branch-name>
=> the ***remote branch*** in local, referred to as <remote>/<branch>
e.g. git fetch origin feature-1 => the remote branch in local, referred to as "origin/feature-1"

Get the ***local branch*** of the remote branch
git checkout <branch-name> // e.g. git checkout feature-1

fetch all branches
git fetch <remote> // e.g. git fetch origin

The downloaded remote branches can be listed with:
git branch -r // result e.g. "origin/master", "origin/feature-1", …

Note: *Fetch will **not overwrite** our local branches*, it only creates remote branches (branches prefixed by their remote names) e.g. origin/master, origin/some-nice-feature, origin/v1 etc.

5)
Add some changes and commits to the (local) branch
Now you commit some changes X, Y to your local tracking branch:

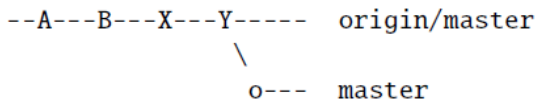
```
--A---B----------  origin/master
       \
        X---Y----  master
```

and want to push them to the server.

6)
Push the branch to the origin repository

git push <remote> <branch> // e.g. *git push origin master*

If the server is still at commit B, this will result in …, and you are done.

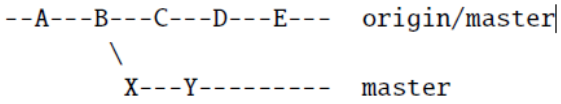


6.1)
However, *if somebody has committed changes to the server **before** you push*, you will get an error message

! [rejected] master -> master (fetch first)
error: failed to push some refs to [...]

To fix the problem, you need to `git fetch' to update the remote branch in local:

git fetch origin

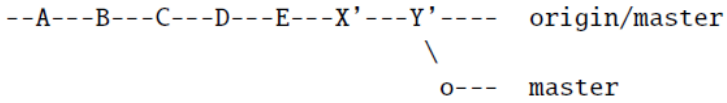


Now your task is to bring the two lines of development together, and you can either do this using **rebase**, or using **merge**.

Rebasing

git rebase origin/master

and end up with



While it is completely feasible to first *fetch*, then *rebase*, you can have both in one command:

git pull --rebase origin master

generally

git pull --rebase <remote> <branch>

This is equivalent to

git fetch <remote> <branch>
git rebase <remote>/<branch>

To push your committed changes, run

git push <remote> <branch>

Merge

Can also be executed in one command

git pull <remote> <branch>

is equivalent to

git fetch <remote> <branch>
git merge <remote>/<branch>

e.g. *git pull origin master*
or git fetch origin
git merge origin/master

Deleting *remote branch* in Git

git push --delete <remote> <branch> // e.g. git push --delete origin feature-1