Git Concept Handbook

Basic

Working Directory

The area where the files and folders that you actually see and operate in the local file system are located. All operations and initial modifications are reflected in the working directory.

Stage Area

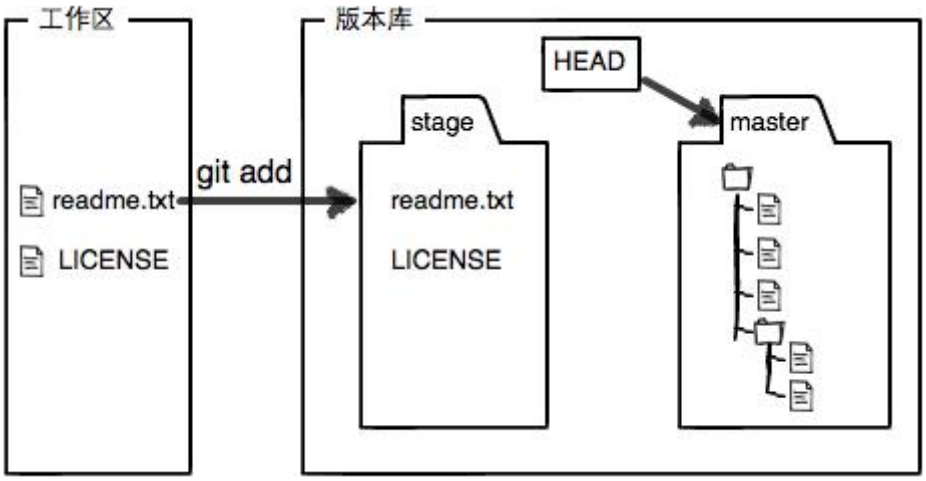
After a file is modified (added), the modified file can be added to the staging area. At this time, the modification in the working directory will be conceptually copied to the staging area, waiting to be committed to the local repository. It is equivalent to putting the modifications you want to commit into a shopping cart first and then checking out and paying when you want to commit.

Local Repository

Each commit permanently saves the contents of the staging area to the local repository. The local repository stores the complete version history of the project. Each commit is a snapshot of the project at a specific point in time. Once the commit is completed, the staging area will be emptied (preparing for the next round of additions and commits), while the files in the working directory will remain in the modified state. You can continue to make new modifications in the working directory and repeat the process of adding to the staging area and committing to the local repository.

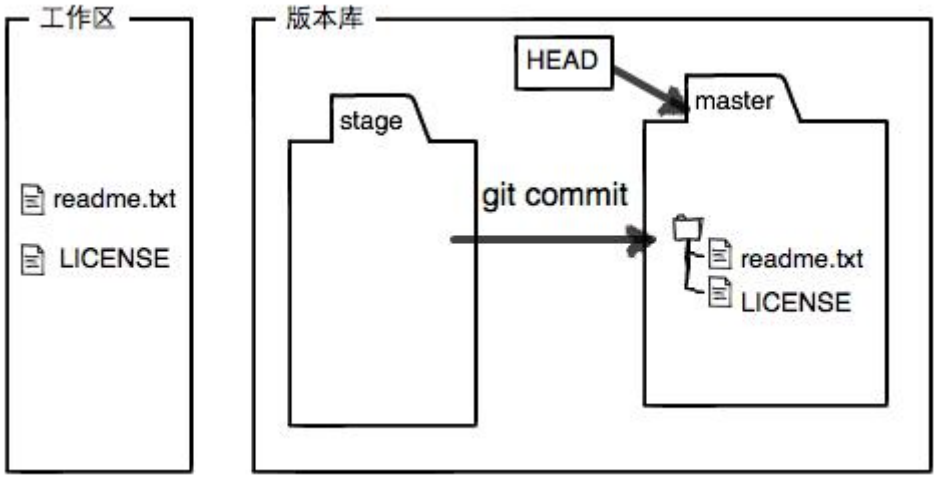
Add

`git add`: Adds file modifications to the staging area.



Commit

`git commit`: When the file modification reaches a certain level, you can save a snapshot, and this behavior is called `commit`. `git commit` is only responsible for committing the modifications in the staging area. The code that has not been added to the staging area is not controlled by `commit`.



Log

`git log`: Displays the commit logs from the most recent to the farthest.
`git reflog`: Displays each command from the most recent to the farthest record. If the version has been rolled back and the ID after the rollback ID cannot be found through the terminal history command line and `log`, you can use `reflog` to find the historical command corresponding to each `commit` and find the ID in the historical command. The following demonstrates rolling back from the latest version to `HEAD^` and then using `reflog` to find the version after `HEAD^` (which cannot be found in `log`).

```
PS D:\project\learn-git> git log
commit c7fd9214d7175af0e25c2c3dcc4cb4a522272d93 (HEAD -> master)
Author: UserName <UserEmail>
Date:   Wed Dec 4 22:42:08 2024 +0800

    add distributed

commit ede7da9dd80cc3a6a92062529e480d7e87b3dfa7
Author: UserName <UserEmail>
Date:   Wed Dec 4 22:37:13 2024 +0800

    commit 3 files

commit 17dc1170224332e1380366eda8dd856ee59f76be
Author: UserName <UserEmail>
Date:   Wed Dec 4 22:35:59 2024 +0800

    write a readme file: readme.txt
PS D:\project\learn-git> git reflog
c7fd921 (HEAD -> master) HEAD@{0}: reset: moving to HEAD^
176a961 HEAD@{1}: commit: add GPL
c7fd921 (HEAD -> master) HEAD@{2}: commit: add distributed
ede7da9 HEAD@{3}: commit: commit 3 files
17dc117 HEAD@{4}: commit (initial): write a readme file: readme.txt
PS D:\project\learn-git> git reset --hard 176a
HEAD is now at 176a961 add GPL
PS D:\project\learn-git> git log
commit 176a961d3fc426ce9084aa8be5152bf088379ed8 (HEAD -> master)
Author: UserName <UserEmail>
Date:   Wed Dec 4 22:44:47 2024 +0800

    add GPL

commit c7fd9214d7175af0e25c2c3dcc4cb4a522272d93
Author: UserName <UserEmail>
Date:   Wed Dec 4 22:42:08 2024 +0800

    add distributed

commit ede7da9dd80cc3a6a92062529e480d7e87b3dfa7
```

```
Author: UserName <UserEmail>
Date:   Wed Dec 4 22:37:13 2024 +0800

    commit 3 files

commit 17dc1170224332e1380366eda8dd856ee59f76be
Author: UserName <UserEmail>
Date:   Wed Dec 4 22:35:59 2024 +0800

    write a readme file: readme.txt
```

HEAD

Each `commit` will have a corresponding ID: `commit e475afc93c209a690c39c13a46716e8fa000c366`. However, this ID is difficult to remember or type. Therefore, Git uses `HEAD` to point to the current version of the current branch, that is, the latest commit ID of the current branch. Similarly, the previous version is `HEAD^`, the version before that is `HEAD^^`, and of course, writing N `^` for N versions up is easy to count incorrectly. So it can be written as `HEAD~N`.

Revocate

The command `git checkout -- readme.txt` means to completely revoke all modifications of the `filename` file in the working directory. There are two cases here: One is that the file has been modified but has not been added to the staging area. In this case, the revocation operation will make the working directory return to the exact same state as the `HEAD` version (return to the state of the most recent `commit`); the other is that after the file has been added to the staging area and then modified again. In this case, the revocation operation will make the working directory return to the state after being added to the staging area (return to the state of the most recent `add`).

To revoke the `add` operation: The command `git reset HEAD <filename>` can revoke the modification in the staging area (unstage). The `git reset` command can both roll back the version and roll back the modification in the staging area to the working directory. When we use `HEAD`, it represents the latest version. At this time, the file stored in the staging area is removed from the staging area and returned to the working directory. At this time, the modification before the `add` operation can still be seen in the working directory. At this time, using the `checkout` operation again will revoke the modification in the file area to the state of the most recent `commit`.

Remove

In Git, deletion is also a modification operation. Generally, you can directly delete the file in the file manager or use the `rm` command. After deletion, Git will be aware of this deletion. Therefore, the working directory and the version library will be inconsistent. The `git status` command will display which files have been deleted.
`git rm <filename>` is equivalent to `rm <filename>` and then `git add`.

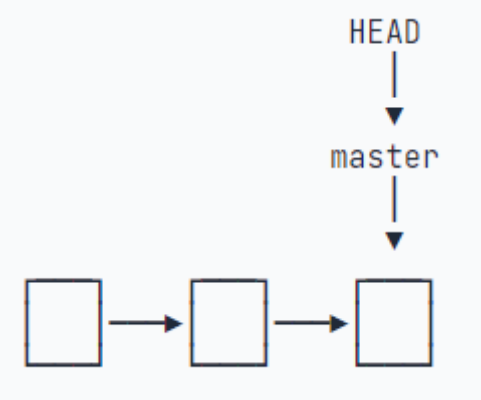
The command `git rm -r --cached <filename>` can remove the `filename` from the working directory without affecting the state of the file in the system.

Branch

Basic Operation

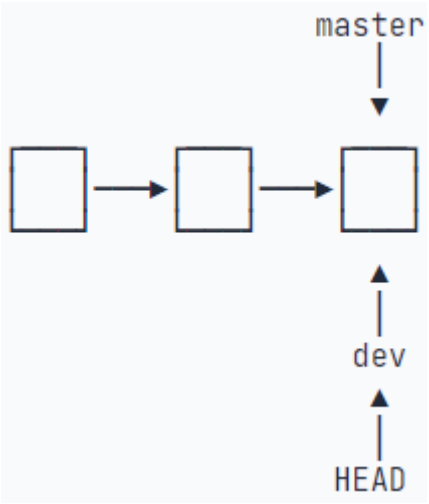
Each commit is strung together by Git into a timeline, and this timeline is a branch. Up to now, there is only one timeline. In Git, this branch is called the main branch, that is, the `master` branch. Strictly speaking, `HEAD` does not point to a commit, but points to `master`, and `master` points to the commit. Therefore, `HEAD` points to the current branch.

At the beginning, the `master` branch is a line. Git uses `master` to point to the latest commit and then uses `HEAD` to point to `master` to determine the current branch and the commit point of the current branch.



Each commit makes the `master` branch move forward one step.

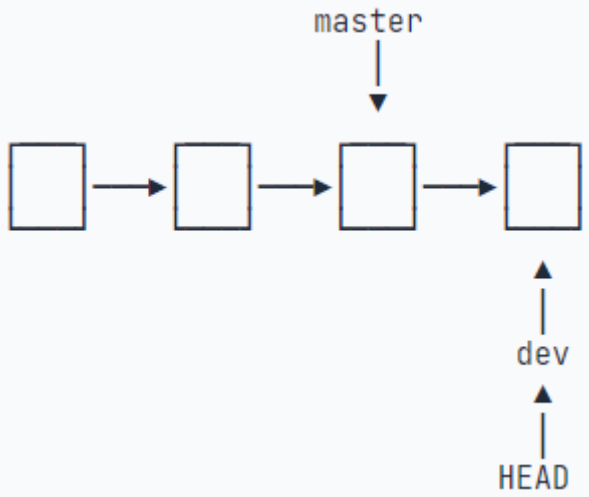
When creating a new branch, for example, `dev`, Git creates a new pointer called `dev`, which points to the same commit as `master`, and then points `HEAD` to `dev`, indicating that the current branch is on `dev`. This operation is equivalent to adding a `dev` pointer and modifying the pointing of `HEAD` to `dev`. The files in the working directory do not change.



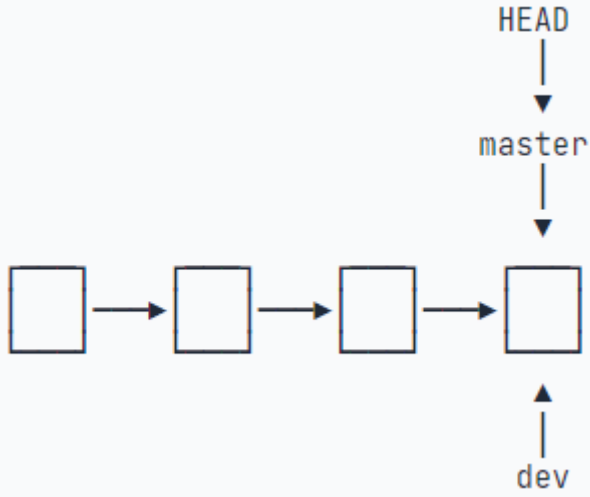
The `git branch dev` command creates a `dev` branch.
The `git branch` command lists all branches, and the current branch is marked with a `*` in front.
The `git switch dev` command can switch between branches.

The `git switch -c dev` command can create a new branch and switch to this branch.

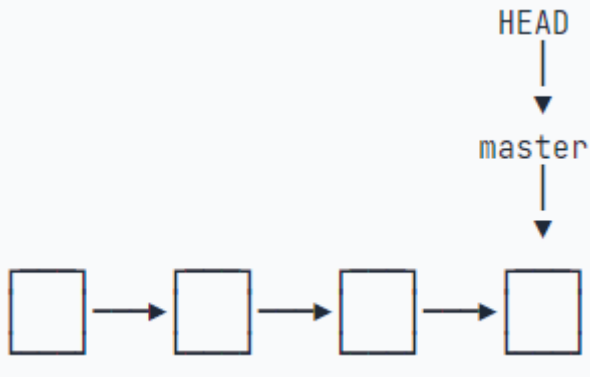
From now on, the modifications and commits in the working directory are for the `dev` branch. After a new commit, the `dev` pointer moves forward one step, while the `master` pointer remains unchanged.



Merging is to directly point `master` to the current commit of `dev`.
The `git merge dev` command is used to merge the specified branch into the current branch.



The `git branch -d dev` is used to delete the branch.



If the branch has not been merged, deletion will result in the loss of modifications. If you want to force deletion, you need to use the uppercase `-D` parameter.
`git branch -D dev`

Solve Conflict

Git conflicts usually occur in the following situations:

- The same part of the same file is modified in two branches.
- When merging branches, Git cannot automatically merge these changes because it is not sure which modification should be retained.

For example, suppose a certain line of content is modified on the `master` branch, and the same line of content is also modified on the `feature1` branch. When trying to merge these two branches, Git will not be able to automatically choose which modification to retain and will mark the conflict.

The solution steps are as follows:

1. Understand the conflict markers: The conflict part will be marked as follows:
    - `<<<<<<< HEAD`: The content of the current branch.
    - `=======`: The separator, indicating the start and end of the conflict.
    - `>>>>>>> feature1`: The content of the target branch.
2. **Manual merge**: According to the requirements, choose to retain the content of the current branch, the target branch, or a combination of both.
3. **Delete the conflict markers**: After resolving the conflict, delete the conflict markers (`<<<<<<<`, `=======`, `>>>>>>>`) to keep the file clear.
4. Add to the staging area and commit:
    - Use `git add <file>` to add the resolved file.
    - Use `git commit` to commit.d

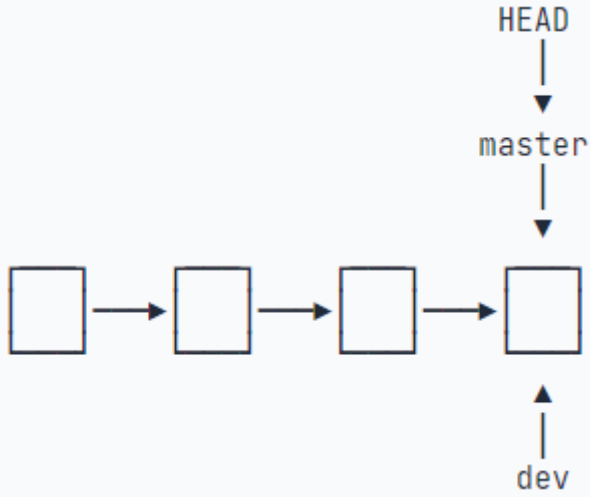
```
PS D:\project\learn-git> git merge feature1
Auto-merging readme.txt
CONFLICT (content): Merge conflict in readme.txt
Automatic merge failed; fix conflicts and then commit the result.
PS D:\project\learn-git> git add .\readme.txt
PS D:\project\learn-git> git commit -m "slove the conflict"
[master 14d521f] slove the conflict
PS D:\project\learn-git> git log --graph --pretty=oneline --abbrev-commit
*   14d521f (HEAD -> master) slove the conflict
|\
| * 4edd20b (feature1) commit 2 in feature1 branch
| * 562f9b1 commit 1 in feature1 branch
* | 235a472 commit 1 in master branch
|/
* 58a61cd add a line in dev branch
* 654d204 remove file:removefile.txt
* 83bb013 add a file to remove
* bab68eb commit second add of stage
* f85400c git track changes
* 55be549 understand how stage works
* 176a961 add GPL
```

```
* c7fd921 add distributed
* ede7da9 commit 3 files
* 17dc117 write a readme file: readme.txt
```

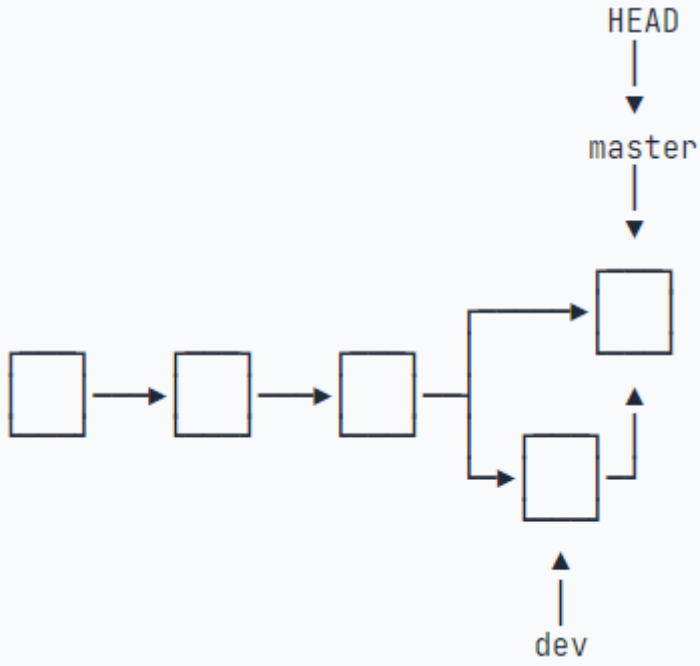
Fast Forward

Usage scenario: If the current branch (such as `master`) has no new commits since the target branch (such as `feature`) was created, Git can perform a fast-forward merge.

Characteristics: In a fast-forward merge, Git directly points the current branch to the latest commit of the target branch without generating an additional merge commit. For example, if `master` has no other commits, Git will directly point the `master` branch pointer to the latest commit of the `feature` branch. Therefore, the merge speed is very fast.



Sometimes you may want to avoid fast-forward merges and keep the merge records. This can be achieved by using the `--no-ff` parameter to force Git to create a merge commit. At the same time of merging, a new `commit` is required. In this way, the branch information can be seen from the branch history. Even if the branch is eventually deleted, the merge commit will record the branch information that once existed.




```
PS D:\project\learn-git> git switch -c dev
```

```
PS D:\project\learn-git> git add .\readme.txt
PS D:\project\learn-git> git commit -m "commit 1 in dev branch"
[dev 8a87f56] commit 1 in dev branch
 1 file changed, 1 insertion(+)
PS D:\project\learn-git> git log --graph --pretty=oneline --abbrev-commit
* 8a87f56 (HEAD -> dev) commit 1 in dev branch
*   14d521f (master) slove the conflict
|\
| * 4edd20b commit 2 in feature1 branch
| * 562f9b1 commit 1 in feature1 branch
* | 235a472 commit 1 in master branch
|/
* 58a61cd add a line in dev branch
* 654d204 remove file:removefile.txt
* 83bb013 add a file to remove
* bab68eb commit second add of stage
* f85400c git track changes
* 55be549 understand how stage works
* 176a961 add GPL
* c7fd921 add distributed
* ede7da9 commit 3 files
* 17dc117 write a readme file: readme.txt
PS D:\project\learn-git> git switch master
Switched to branch 'master'
PS D:\project\learn-git> git merge --no-ff dev
Merge made by the 'ort' strategy.
 readme.txt | 1 +
 1 file changed, 1 insertion(+)
PS D:\project\learn-git> git log --graph --pretty=oneline --abbrev-commit
*   10b20fa (HEAD -> master) Merge branch 'dev'
|\
| * 8a87f56 (dev) commit 1 in dev branch
|/
*   14d521f slove the conflict
|\
| * 4edd20b commit 2 in feature1 branch
| * 562f9b1 commit 1 in feature1 branch
* | 235a472 commit 1 in master branch
|/
* 58a61cd add a line in dev branch
* 654d204 remove file:removefile.txt
* 83bb013 add a file to remove
* bab68eb commit second add of stage
* f85400c git track changes
* 55be549 understand how stage works
* 176a961 add GPL
* c7fd921 add distributed
* ede7da9 commit 3 files
* 17dc117 write a readme file: readme.txt
```

From the perspective of pointers:

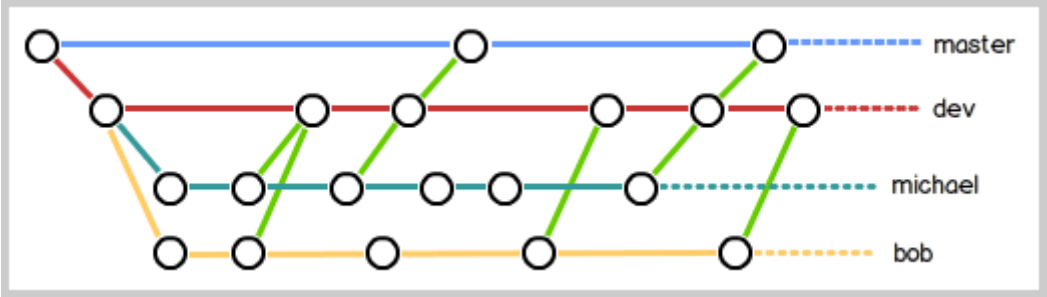
- When performing a fast-forward merge:
  - Git will directly move the `master` branch pointer from `C` to `E` (the latest commit of the `feature` branch).
  - The commit history of `master` will no longer have a merge record but will directly inherit all commits of the `feature` branch.

```
A---B---C  (master)
           \
            D---E  (feature)

            merge

A---B---C---D---E  (master, feature)
```

- When performing a non-fast-forward merge:
  - Git will create a new merge commit `M`, merging the changes of the `master` and `feature` branches.
  - The `master` branch pointer will point to the merge commit `M`.

```
A---B---C---F---M  (master)
           \     /
            D---E  (feature)
```

Branch Strategy



Stash

Sometimes when the work on a branch is not completed and you need to switch to another branch to work, but you don't want the uncommitted work on this branch to disappear.
The `stash` function can "store" the current working branch scene. After restoring the scene later, you can continue working. It is equivalent to temporarily storing the content of the current working directory and clearing the uncommitted modifications on this branch. After stashing, you can switch the current branch, complete the task in another branch, and then switch back to the original branch and restore the work in the `stash`.
Use the `git stash list` command to view the content stored in the `stash`.
To restore the original work content, there are two methods:

1. One is to use `git stash apply` to restore, but after restoration, the `stash` content is not deleted, and you also need to use `git stash drop` to delete it.
2. Another way is to use `git stash pop`, which deletes the `stash` content while restoring.
   You can also stash multiple times. When restoring, first use `git stash list` to view, and then restore the specified `stash` using the command:

```
git stash apply stash@{0}
```

Remote Repository

The local repository and the remote repository need to be associated first. You can use `git remote add origin <ssh location>`. After adding, the name of the remote repository is `origin`. This is the default name in Git and can also be changed to other names, but the name `origin` clearly indicates that it is a remote repository.
The next step is to push all the contents of the local repository to the remote repository:

```
$ git push -u origin master
Counting objects: 20, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (15/15), done.
Writing objects: 100% (20/20), 1.64 KiB | 560.00 KiB/s, done.
Total 20 (delta 5), reused 0 (delta 0)
remote: Resolving deltas: 100% (5/5), done.
To github.com:michaelliao/learngit.git
 * [new branch]      master -> master
Branch 'master' set up to track remote branch 'master' from 'origin'.
```

The `git push` command actually pushes the selected local repository branch `master` to the remote repository.
Since the remote repository is empty, when we push the `master` branch for the first time, we add the `-u` parameter. Git will not only push the content of the local `master` branch to the new `master` branch of the remote repository but also associate the local `master` branch with the remote `master` branch, which can simplify the command in future pushes or pulls.

Cooperation

View

To view the information of the remote repository, use `git remote -v`.

Push

If a newly created local branch is not pushed to the remote repository, it is invisible to others.
Pushing a branch means pushing all local commits on that branch to the remote repository. When pushing, you need to specify the local branch, so that Git will push the

branch to the corresponding remote branch of the remote repository.

```
$ git push origin <branchname>
```

Branch pushing principle: `master` and `dev` are always synchronized with the remote repository, and other branches are determined according to work requirements.

Pull

When cloning from a remote repository, by default, only the local `master` branch can be seen.
If you want to develop on other branches, you must create other branches of the remote `origin` locally. Use this command to create a local branch of the remote `origin`, provided that this branch has been pushed to the remote repository.

```
$ git checkout -b dev origin/dev
```

If the same branch is pulled to the local of different users, and different users make different modifications to the same branch and then push them to the remote repository at the same time, the push will fail.
This is because the latest commits of different users conflict, and the remote branch is newer than the local branch. Therefore, there is a push order. If user 1 pushes the modified branch to the remote repository first, user 2 needs to use `git pull` to pull the latest commit from the corresponding `origin/branch` first, then merge locally, resolve the conflict, and then push.
If you directly use `git pull`, an error will be prompted:

```
$ git pull
There is no tracking information for the current branch.
Please specify which branch you want to merge with.
See git-pull(1) for details.

    git pull <remote> <branch>

If you wish to set tracking information for this branch you can do so
with:

    git branch --set-upstream-to=origin/<branch> dev
```

According to the prompt, before `pull`, you also need to specify the link between the local `dev` branch and the remote `origin/dev` branch:

```
$ git branch --set-upstream-to=origin/dev dev
Branch 'dev' set up to track remote branch 'dev' from 'origin'.
```

Then `pull`:

```
$ git pull
Auto-merging env.txt
CONFLICT (add/add): Merge conflict in env.txt
Automatic merge failed; fix conflicts and then commit the result.
```

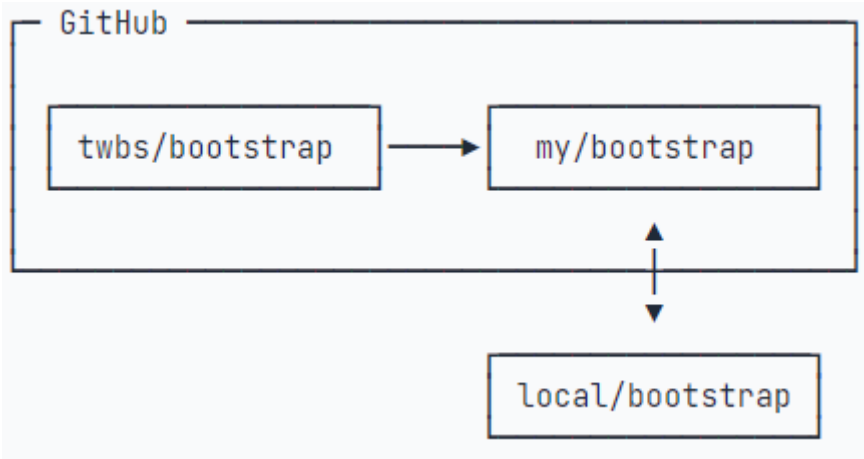
This time `git pull` is successful, but there is a merge conflict. You need to resolve it manually, commit, and then push:

```
$ git commit -m "fix env conflict"
[dev 57c53ab] fix env conflict

$ git push origin dev
Counting objects: 6, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (6/6), 621 bytes | 621.00 KiB/s, done.
Total 6 (delta 0), reused 0 (delta 0)
To github.com:michaelliao/learngit.git
   7a5e5dd..57c53ab  dev -> dev
```

Appendix

Fork

Using `Fork`, you can participate in an open-source project of others. `Fork` can clone the remote repository of others under your own account, and then clone from your own account.
Be sure to clone the repository from your own account so that you can push modifications. If you directly clone the repository of others and modify it locally, you will not be able to push the local modifications because you do not have permission.
Taking the open-source project `bootstrap` as an example, the relationship diagram of the `bootstrap` repository `twbs/bootstrap`, the cloned repository `myname/bootstrap` on GitHub, and the cloned repository on the local computer is as follows:



If you hope that the official can accept the local modifications, you can initiate a pull request on GitHub and wait for the other party's permission.

Config

The `--global` parameter in `git config` is a global parameter. When configuring Git, adding `--global` affects the current user. If not added, it only affects the current repository.
The Git configuration file of each repository is placed in the `.git/config` file, and the Git configuration file of the current user is placed in a hidden file `.gitconfig` in the user's home directory.

```
d:\project\learn-git>cat ~/.gitconfig
[alias]
        br = branch
        st = status
        ci = commit
        re = reset
        sw = switch
        sk = stash
        glog = log --graph --pretty=oneline --abbrev-commit
[user]
        email = [UserEmail]
        name = [UserName]
```