

CODING Code Repositories

Best Practices

Product Documentation



Copyright Notice

©2013-2024 Tencent Cloud. All rights reserved.

Copyright in this document is exclusively owned by Tencent Cloud. You must not reproduce, modify, copy or distribute in any way, in whole or in part, the contents of this document without Tencent Cloud's the prior written consent.

Trademark Notice



All trademarks associated with Tencent Cloud and its services are owned by Tencent Cloud Computing (Beijing) Company Limited and its affiliated companies. Trademarks of third parties referred to in this document are owned by their respective proprietors.

Service Statement

This document is intended to provide users with general information about Tencent Cloud's products and services only and does not form part of Tencent Cloud's terms and conditions. Tencent Cloud's products or services are subject to change. Specific products and services and the standards applicable to them are exclusively provided for in Tencent Cloud's applicable terms and conditions.

Contents

Best Practices

Git Code Rollback and Retrieval

Best Practices

Git Code Rollback and Retrieval

Last updated : 2023-12-25 17:08:18

This document describes code rollback and retrieval in Git repositories.

Open Project

1. Log in to the CODING Console and click the team domain name to go to CODING.
2. Click



in the upper-right corner to open the project list page and click a project icon to open the corresponding project.

3. Select **Code Repositories** in the menu on the left.

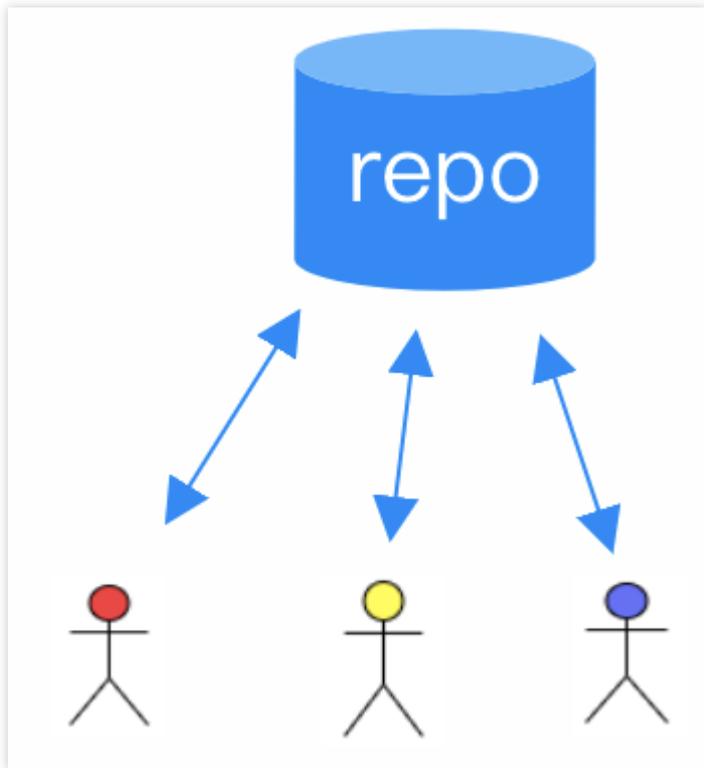
Git is a flexible version control tool that helps teams enhance team collaboration and track different versions of codes. In real-world scenarios, some developers may intentionally or unintentionally delete some Git functions, causing problems to the team or even leading to the loss of important codes. Improper code rollback is one of the main problems.

This document explains how to perform code rollback in various scenarios to recover content mistakenly deleted.

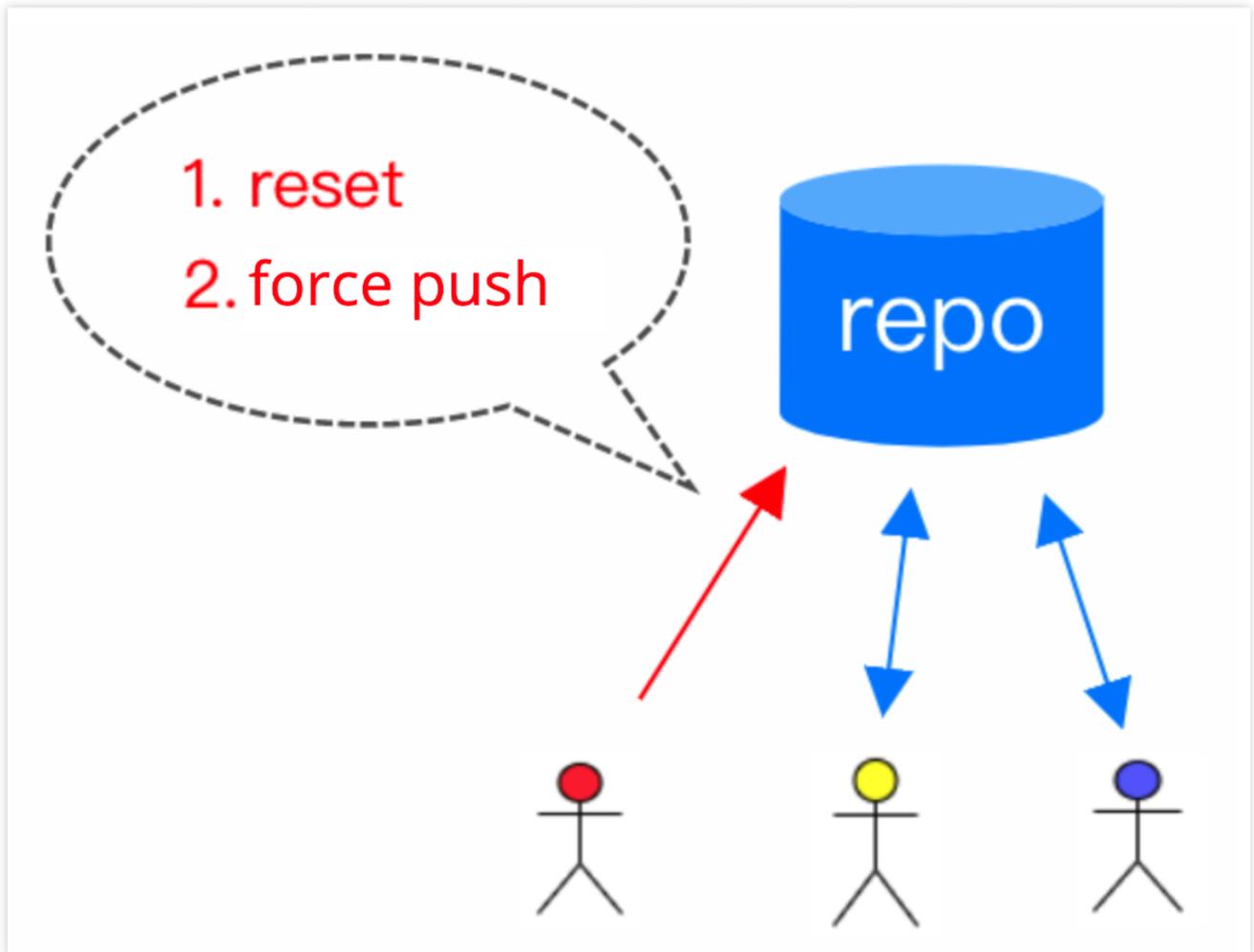
Case Study

Let's start with a typical real-world case that demonstrates the problems caused by improper code rollback.

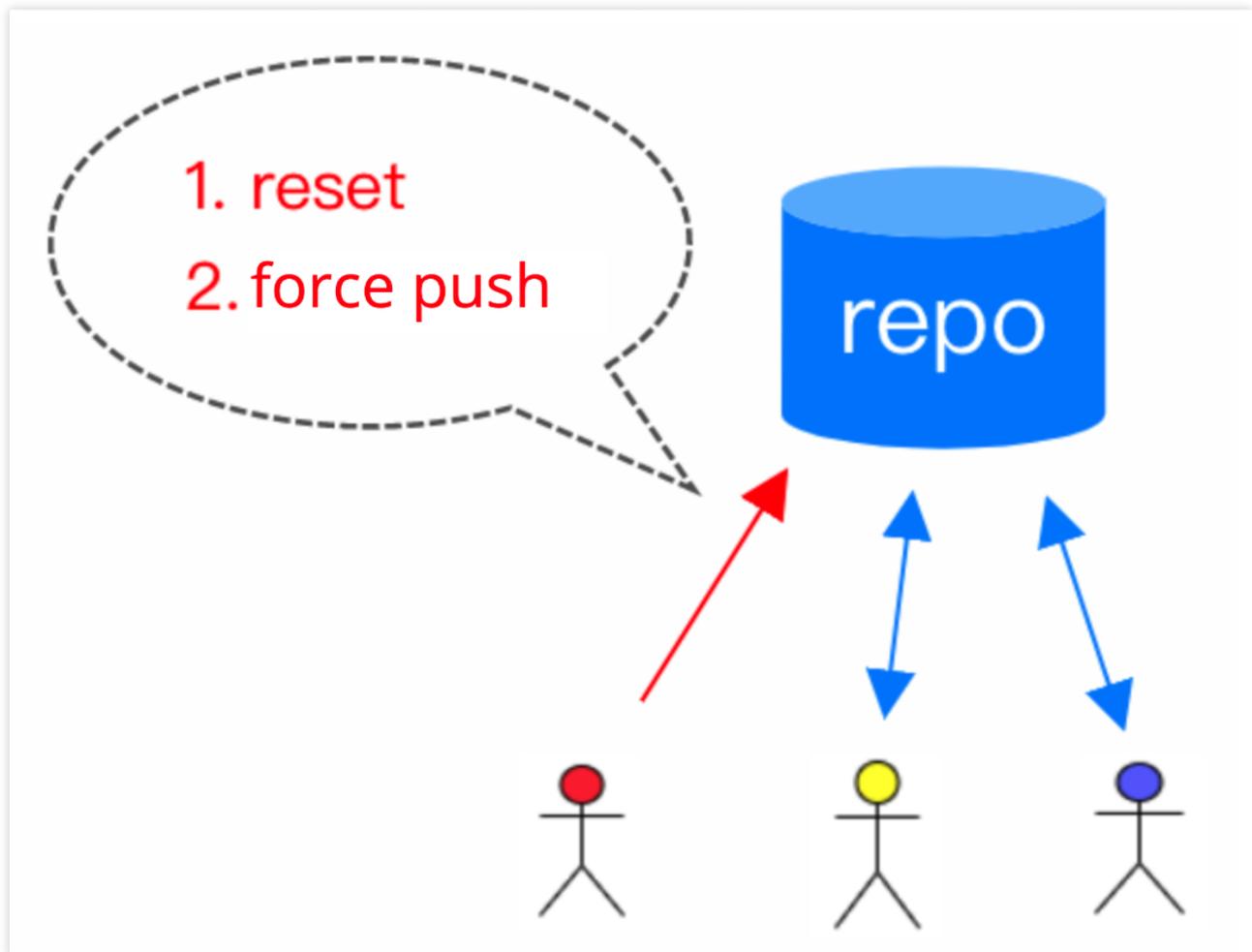
- (1) John, Jane, and Sam worked on the same branch.



(2) John used `reset` to roll back some content and found that the push failed. Finally, he completed the operation using `push -f`. However, `push -f` prompted that the target is a protected branch (such as `master`), so the push failed. Therefore, John unlocked the branch and then run `push -f`.



(3) Jane and Sam performed a normal git pull, but encountered many conflicts and the commit history was messed up.



(4) After a while, they needed to check the source code of a release, but couldn't find the exact code. That's because the code was deleted by the `reset` performed by John.

Four Working Areas of Git

Before analyzing the common code rollback scenarios, let's learn about the four Git working areas.

After you clone a repository, you will see a local directory containing all the project files. We can divide the content into four working areas:

WorkspaceA workspace is also known as a working directory or working replica. It is the directory containing the project files after you clone a repository. This is the working area where you perform routine development operations.

Local repository (`.git`)

The workspace has a hidden `.git` directory. This is the database of the local Git repository. The directory files in the workspace are checked out from here. After you modify files and commit your changes, they are recorded in the local repository. **Tips: Do not manually modify content in the `.git` directory.**

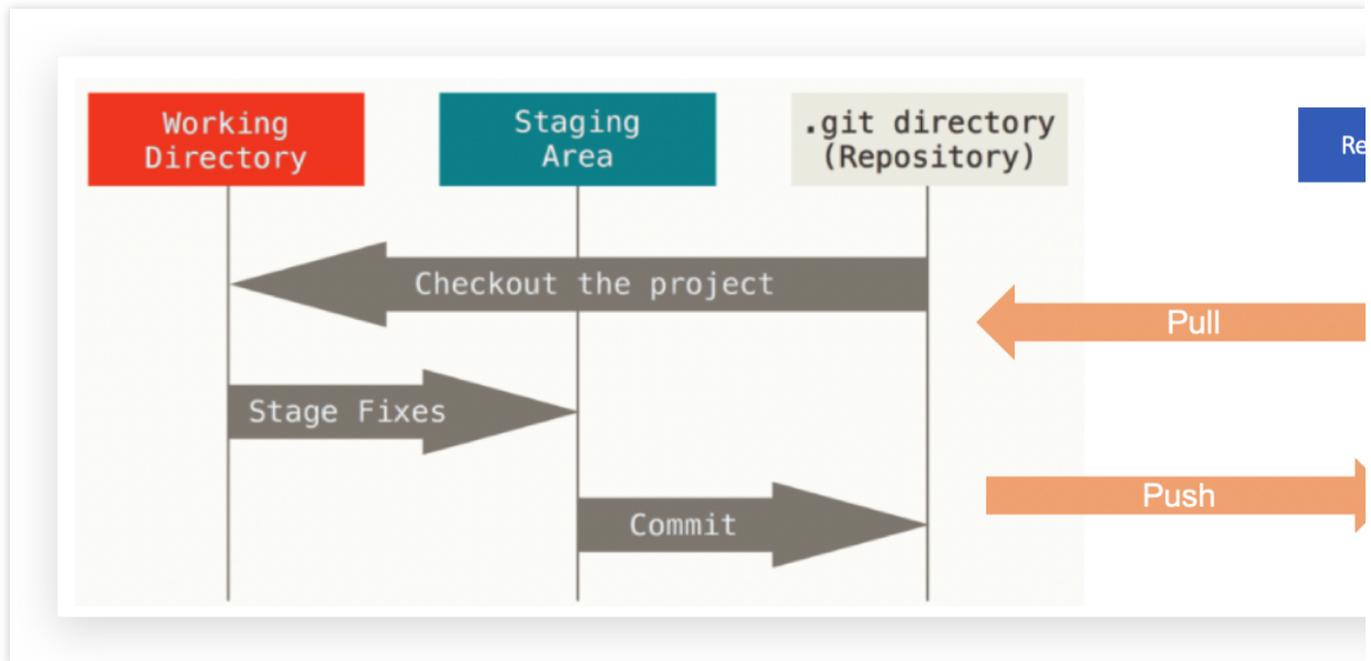
Staging area

A staging area is also called cache, an area in between the workspace and local repository. It is mainly used to mark

modified content. By default, the content in the staging area is recorded to the local repository in the next commit.

Remote repository

For team collaboration, you need to specify a remote repository (or multiple repositories in certain cases). Team members interact with the remote repository during collaboration.



A basic Git workflow is as follows:

1. Modify files in the `workspace` .
2. Temporarily store files into the `staging area` .
3. Commit content from the `staging area` to the `local repository` .
4. Push content from the `local repository` to the `remote repository` .

Common Code Rollback Scenarios

Rollback scenario: Changes in workspace only

When you have modified files in the workspace but the changes have not been committed to the staging area or local repository, you can use `git checkout -- file name` to roll back the changes.

Note: These changes will not be committed to the Git repository and will not appear in the Git history.

After the rollback, these changes are discarded permanently.

Example: When you use `git status` , changes not committed are displayed in "Changes not staged for commit".

```
$ git status
```

```
.....
```

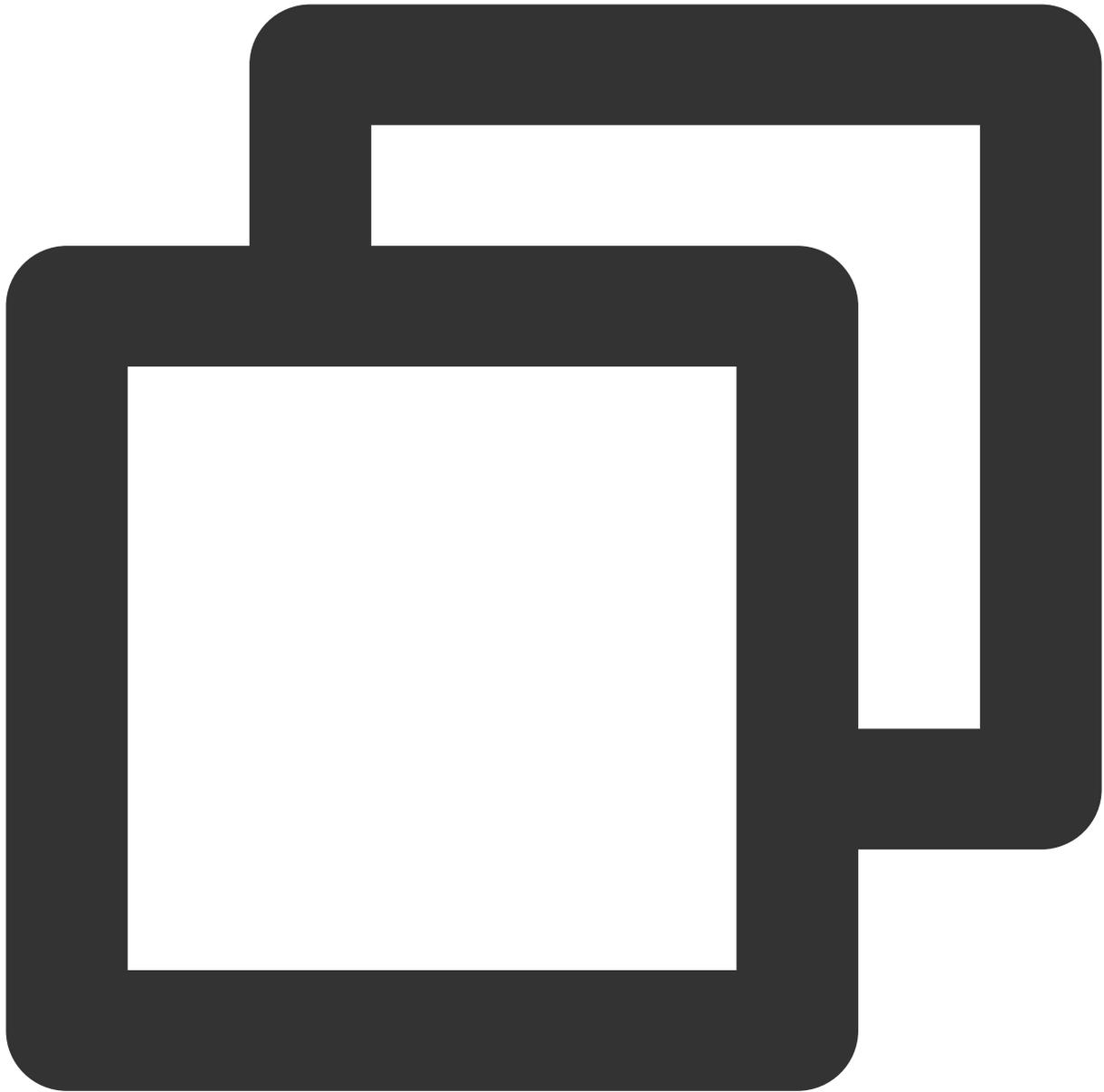
```
Changes not staged for commit:
```

```
(use "git add <file>..." to update what will be committed)
```

```
(use "git checkout -- <file>..." to discard changes in work
```

```
modified: build.sh
```

Run the following command to roll back changes in the workspace:



```
git checkout -- build.sh
```

Rollback scenario: Changes in staging area

If you run `git add` to add changes to the staging area, but have not commit them, you can use `git reset HEAD fill name` to roll back the changes. When you use `git status`, the following prompt is displayed:

```
$ git status
```

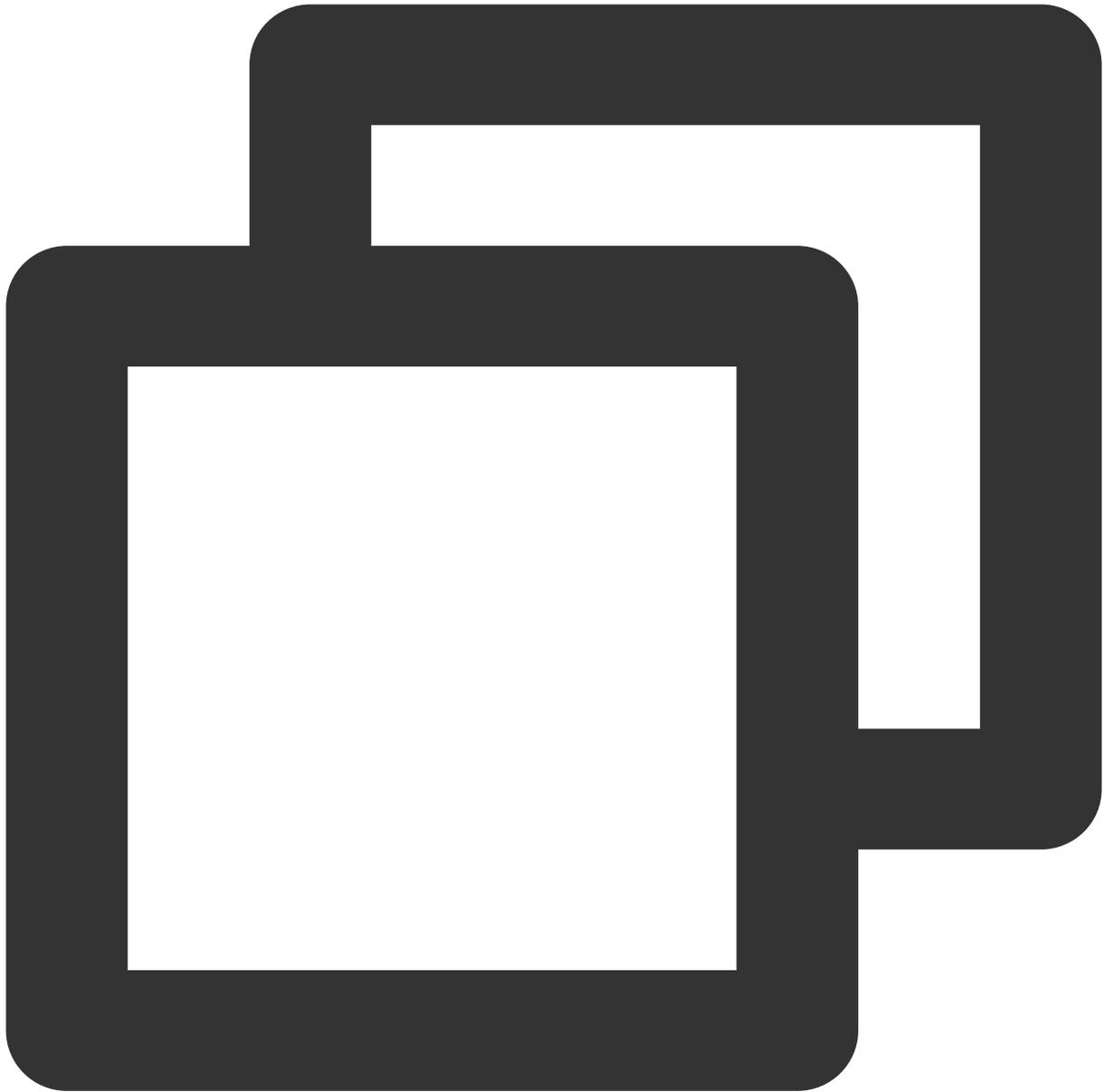
```
.....
```

```
Changes to be committed:
```

```
(use "git reset HEAD <file>..." to unstage)
```

```
modified: build.sh
```

Run the following command to rollback changes in the staging area:



```
git reset HEAD build.sh
```

After rollback, the changes will be retained in the workspace, so you can edit and commit again, or use `git checkout -- file name` to permanently discard the changes.

Rollback scenario: Committed, but not pushed

When you have committed changes to the local repository but have not pushed them to the remote repository, you can use the `git reset` command in the format:

```
git reset <commit to roll back to> or git reset --hard <commit to roll back to>
```

You must note that any commits after the **commit to roll back to** will be discarded.

Example:

```
$ git log --oneline -3
a568b63 (HEAD -> master) just for testing
30c1d8c ignore some dirs and files
fa02a21 renamed config script
$ git reset fa02a21 # Roll back to a commit (the following records are discarded, but the ch
$ git log --oneline -3 # The record after the specified commit no longer exists
fa02a21 (HEAD -> master) renamed config script
563b269 delete test script
4304054 update build type
```

By default, the changes discarded by `git reset` are retained in the workspace so you can edit and commit again. If you add `--hard` to the command, the changes are not retained. Proceed with caution.

Rollback scenario: Modify the last local commit

If additional changes are required after committing, you can use "git reset" to add them in the commit,. However, Git also provides a simple method for updating the last commit.

The command format is as follows:

```
git commit --amend [ -m <commit description> ]
```

If `-m <commit description>` is not included in the command, Git pulls up the editor to enter the log description. Example:

```

$ git commit -m 'add build type' # Remember that there are other changes to be recorded in this
$ vim build.sh } # Modifying the file is optional. If there is no file modification, the commit --
$ git add build.sh } # changing the commit log.commit
$ git commit --amend -m 'add build type and package type' # Modify the content of the
$ git log --oneline -3 # we can see that the latest commit has been updated
02dd82a (HEAD -> master) add build type and package type
3fa6d52 format introduction
c0d8450 init build.sh

```

The "git commit --amend" can only be used to modify local commits that have not been pushed.

Rollback scenario: Pushed to remote repository

**Note: Use "git revert" instead of "git reset".

We want to emphasize this point because "git reset" will delete your history. This can cause various problems if you have already pushed records. But "git revert" only rolls back a commit and makes a new one without erasing history.

Command	Whether to Erase History
git reset	Yes, the history of the rollback will disappear
git revert	No, the history will be preserved and the commit will be regenerated after rollback

Example:

```
$ git revert -n 140ce0c # Roll back a commit (such as 140ce0c), the -n option indicates  
(If there is a conflict in the process, you need to deal with the conflict first, and then execute "git revert  
$ git revert --continue # Pull up Vim by default to fill in the log, keep the default log  
$ git push # After processing, remember to push to the remote
```

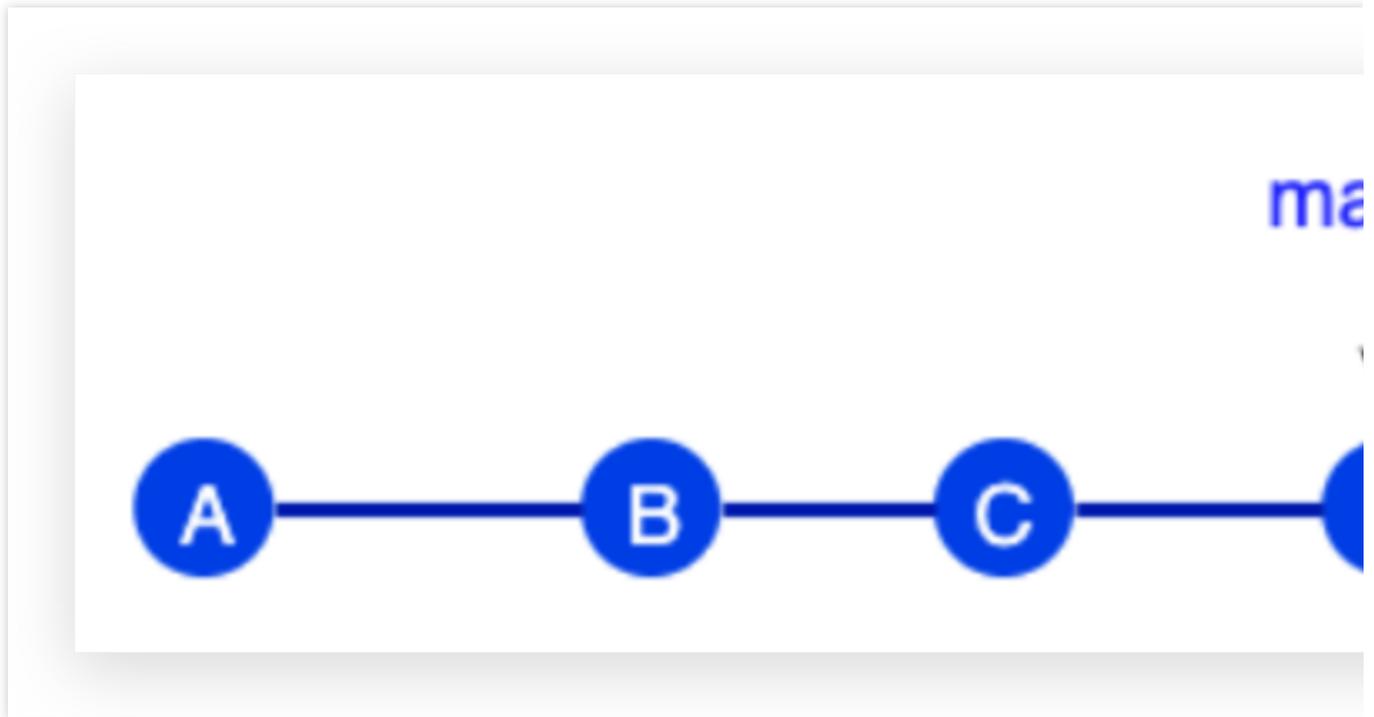
If you encounter a problem in this process (such as commit history was messed up when you handle a conflict), use "git revert --abort" to cancel this rollback operation.

If you want to roll back a merge commit, add "-m " when performing revert to specify the parent node record to use as the main thread after rollback. A merge commit generally has two parent nodes numbered 1 and 2 in order. To roll back a commit that merges a branch into the master, you can use "-m 1" to take the master record as the main thread. Rolling back a merge commit is a complicated operation. In general, we recommend you avoid this operation. For more information, see <https://github.com/git/git/blob/master/Documentation/howto/revert-a-faulty-merge.txt>

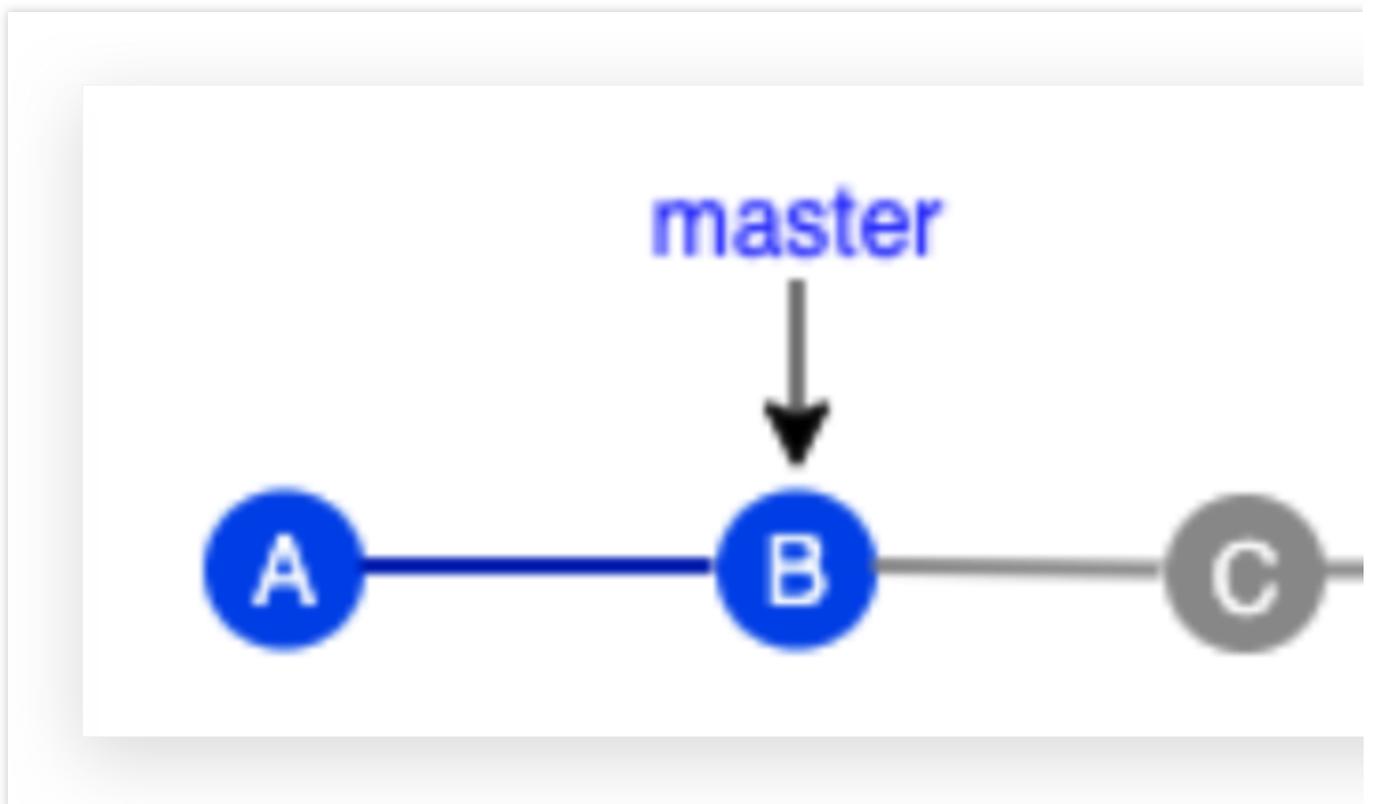
Reset vs. Revert

This section will give an example to show the difference between `git reset` and `git revert` .

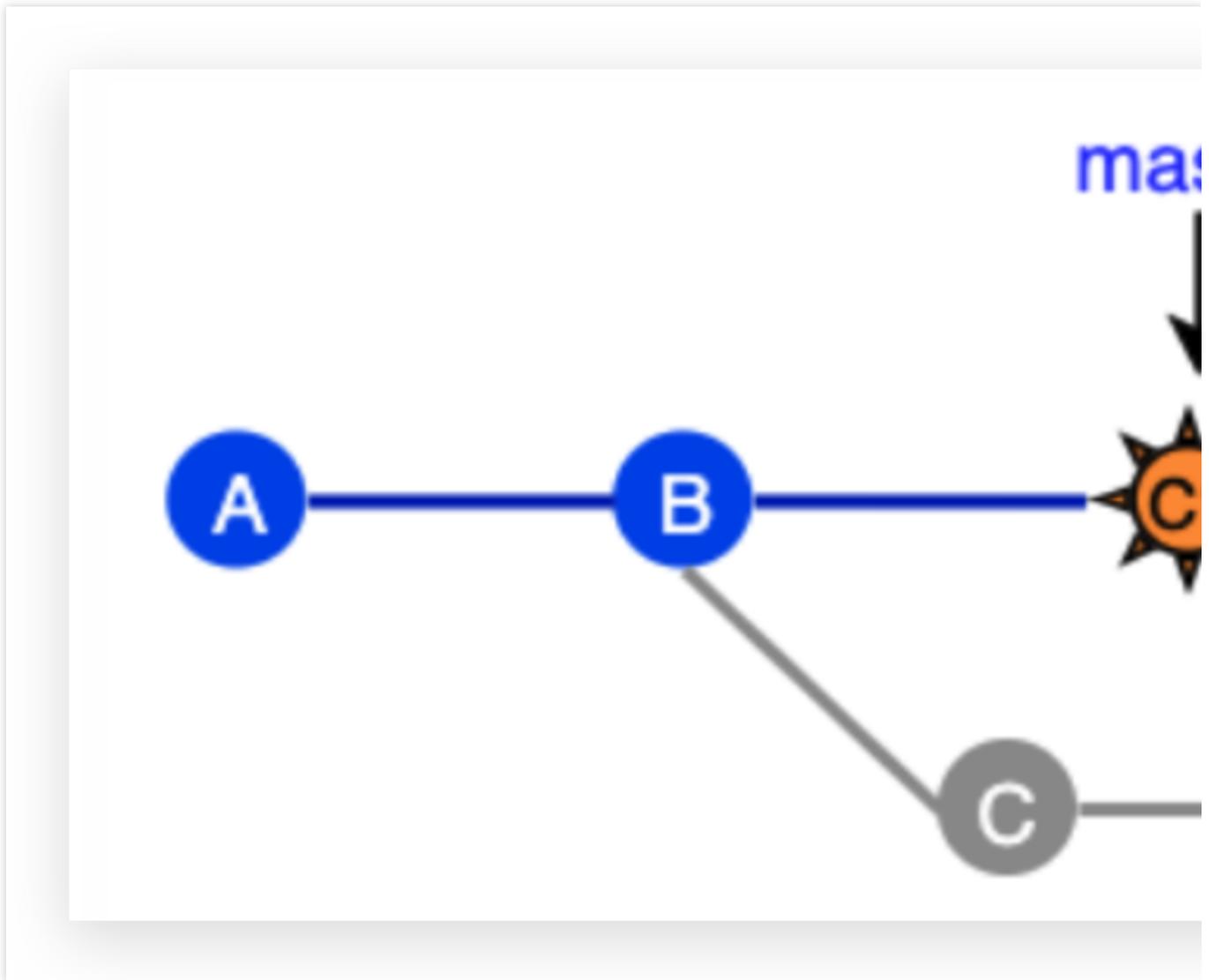
The initial status of the branch is as follows:



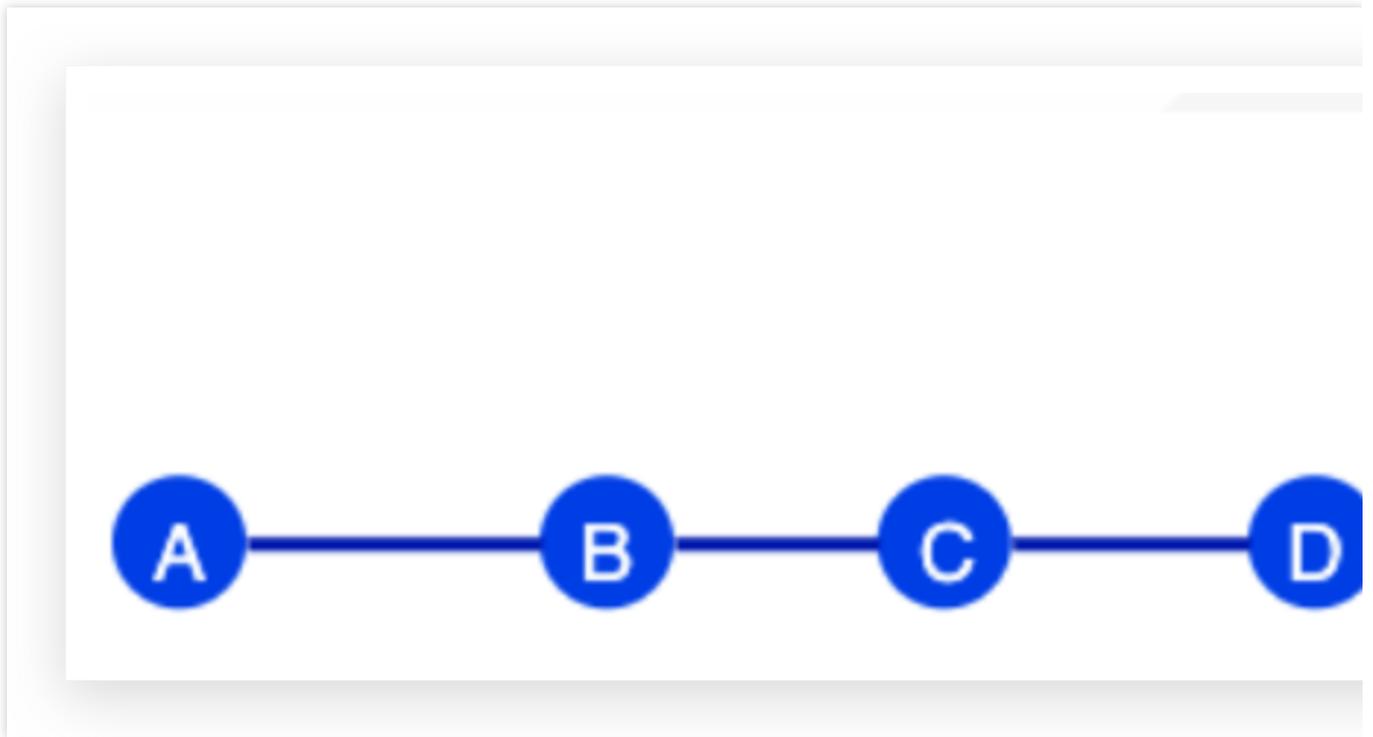
If you run `git reset B`, the workspace will roll back to `B`, discarding all subsequent commits (C and D).



If you generate a new commit (`C1`), `C1` will have no relationship with C and D.



If you run `git revert B`,
a new commit (`E`) is generated to roll back to `B`. This does not modify the existing commit history.



Retrieve Deleted Content

Although Git is a powerful version control tool and you generally do not have to worry about code committed to repositories, you still may have to recover content in certain situations, such as after an improper reset operation or accidental branch deletion. In such case, you can use `git reflog`.

The "git reflog" is a powerful tool used to recover local history. It can restore almost any local record, such as a commit discarded by reset or a branch deleted.

However, "**git reflog**" **cannot recover all records**. It does not work for the following content:

1. Non-local operation records

The "git reflog" manages local workspace records, not non-local records (such as those generated by other users or on other machines).

2. Uncommitted content

This operation cannot recover content rolled back only in the workspace or staging area (`git checkout -- file` or `git reset HEAD files`).

3. Content from too long ago

The "git reflog" only retains records for a limited time (90 days by default). Records that exceed the time limit are automatically cleared. In addition, you can manually run the `clear` command to clear records before their expiration.

Reflog - Revert to a specific commit

A typical use case for this command is when you use `reset` to roll back and then discover a rollback error. In this case, you can revert to another commit status.

```
$ git log --oneline
d57f339 (HEAD -> master) add docs
7bed786 update readme and build.sh
7ebfc00 show basic info
468213d init build script
d9b967a init readme
$ git reset --hard 468213d # Rollback to a specific commit
HEAD is now at 468213d init build script
$ git log --oneline
468213d (HEAD -> master) init build script
d9b967a init readme

(After a while, I found that what I really wanted to reset was "show basic info")
```

Use `git reflog` to view commit operation history. Find the target commit and then perform `reset` to revert to this commit.

```
$ git reflog # Local operation record
468213d (HEAD -> master) HEAD@{0}: reset: moving to
d57f339 HEAD@{1}: commit: add docs
7bed786 HEAD@{2}: commit (amend): update readme an
6fcd68b HEAD@{3}: commit: show password
7ebfc00 HEAD@{4}: commit: show basic info
468213d (HEAD -> master) HEAD@{5}: commit: init build
d9b967a HEAD@{6}: commit (initial): init readme

(The commit of "show basic info" was found to be 7ebfc00)

$ git reset --hard 7ebfc00 # Revert to this commit
$ git log --oneline
7ebfc00 (HEAD -> master) show basic info
468213d init build script
d9b967a init readme
```

This example shows that a **clear and meaningful commit log is very helpful**. For example, if the commit log only has vague descriptions, such as "update" or "fix", it is difficult to find the target commit even with the "git reflog" tool.

Reflog - Restore a file in a specific commit

Scenario: After reset rollback, you find you have discarded some necessary files. Solution: Use reflog to find the target commit and then run the following command to restore a specific file from the commit.

```
git checkout <target commit> -- <file>
```

Example: After you run reset to roll back to commit 468213d, you find the **build.sh** file in the latest status (commit d57f339) is still needed. Therefore, you want to restore this file to the workspace.

```
$ git reflog # Local operation record
468213d (HEAD -> master) HEAD@{0}: reset: moving to 468213d
d57f339 HEAD@{1}: commit: add docs
7bed786 HEAD@{2}: commit (amend): update readme and build.sh
6fcd68b HEAD@{3}: commit: show password
7ebfc00 HEAD@{4}: commit: show basic info
468213d (HEAD -> master) HEAD@{5}: commit: init build script
d9b967a HEAD@{6}: commit (initial): init readme

$ git checkout d57f339 -- build.sh # Restore build.sh in target commit
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
        modified:   build.sh
```

Reflog - Retrieve a branch deleted locally

Scenario: After you use "git branch -D" to delete a local branch, you discover that the branch was mistakenly deleted because this branch was not merged. Solution: Use reflog to find the branch deleted by the current commit and rebuild the branch based on a target commit.

```
git branch <branch name> <target commit>
```

In the Reflog record, the commit record between "to " (for example: moving from master to dev/pilot-001) and switching to another branch (for example: moving from dev/pilot-001 to master) is a change on the branch, from which you can select the desired commit to rebuild the branch.

Example:

```
$ git branch -D dev/pilot-001 # Force delete branch
(.....After a while, found that I deleted it by mistake.....)
$ git reflog # Local operation record
f241ce8 (HEAD -> master) HEAD@{0}: commit: add greetings
d57f339 HEAD@{1}: checkout: moving from dev/pilot-001 to master
2183516 HEAD@{2}: commit: try new build method
32e92cf HEAD@{3}: commit: renew iOS certificate
d57f339 HEAD@{4}: checkout: moving from master to dev/pilot-001
d57f339 HEAD@{5}: reset: moving to d57f339
$ git branch dev/pilot-001 2183516 # Create branch based on target
```

From "to
branches, it is t

Retrieve a deleted branch after merging

A Git's best practice is to delete branches after merging to keep the code repository clean and only maintain active branches. Some developers may wish to retain branches after merging in case they may still be useful. But content merged into the master will not be deleted and can be recovered at any time by rebuilding the branch from a specific commit. Moreover, it is rarely necessary to use old development branches. Generally, even in the case of a function bug, a new branch can be created to fix the problem and verify the solution.

However, if you want to rebuild a merged branch, you can find the branch merge record through the master history, find the branch node, and create a new branch based on this commit, for example:

```
git branch dev/feature-abc 1f85427
```

Graph	Description
	<p>🔗 master enhance mail feature</p> <p>Merge branch 'dev/feature-abc'</p> <p>define platform</p> <p>basic config</p> <p>init introduction</p> <p>show greetings</p> <p>init project</p>

Some Suggestions for Code Rollback

The following are some suggestions for specific commands:

Command	Particularity	Recommendation
git checkout -- file	Roll back unstaged changes in the local workspace, discarded content cannot be recovered	Be sure to confirm that the operation is proceeding
git reset HEAD file	Rollback changes to files in the staging area	Generally do not add "--hard"
git reset <commit>	Roll back to the target commit, discard the commit record after the commit, and keep the changes made by the discarded record in the workspace	1. Only operate the local pushed. 2. Use the "--hard" option
git commit -- amend	Modify the content and commit log of the last commit	Only operate the local re pushed.
git revert <commit>	Roll back the changes made by the relevant commit, and submit again will generate a new commit, and the historical commit record will not be affected	If you want to roll back the

In addition, you should be careful when performing rollback. Do not rely too much on rollback and avoid using "git push -f". In the words of a wise man, **If you use "git push -f", you must be doing something wrong!**

