NAME

gitfaq - Frequently asked questions about using Git

SYNOPSIS

gitfaq

DESCRIPTION

The examples in this FAQ assume a standard POSIX shell, like **bash** or **dash**, and a user, A U Thor, who has the account **author** on the hosting provider **git.example.org**.

CONFIGURATION

What should I put in **user.name**?

You should put your personal name, generally a form using a given name and family name. For example, the current maintainer of Git uses "Junio C Hamano". This will be the name portion that is stored in every commit you make.

This configuration doesn't have any effect on authenticating to remote services; for that, see **credential.username** in **git-config**(1).

What does **http.postBuffer** really do?

This option changes the size of the buffer that Git uses when pushing data to a remote over HTTP or HTTPS. If the data is larger than this size, libcurl, which handles the HTTP support for Git, will use chunked transfer encoding since it isn't known ahead of time what the size of the pushed data will be.

Leaving this value at the default size is fine unless you know that either the remote server or a proxy in the middle doesn't support HTTP/1.1 (which introduced the chunked transfer encoding) or is known to be broken with chunked data. This is often (erroneously) suggested as a solution for generic push problems, but since almost every server and proxy supports at least HTTP/1.1, raising this value usually doesn't solve most push problems. A server or proxy that didn't correctly support HTTP/1.1 and chunked transfer encoding wouldn't be that useful on the Internet today, since it would break lots of traffic.

Note that increasing this value will increase the memory used on every relevant push that Git does over HTTP or HTTPS, since the entire buffer is allocated regardless of whether or not it is all used. Thus, it's best to leave it at the default unless you are sure you need a different value.

How do I configure a different editor?

If you haven't specified an editor specifically for Git, it will by default use the editor you've configured using the **VISUAL** or **EDITOR** environment variables, or if neither is specified, the

system default (which is usually **vi**). Since some people find **vi** difficult to use or prefer a different editor, it may be desirable to change the editor used.

If you want to configure a general editor for most programs which need one, you can edit your shell configuration (e.g., **~/.bashrc** or **~/.zshenv**) to contain a line setting the **EDITOR** or **VISUAL** environment variable to an appropriate value. For example, if you prefer the editor **nano**, then you could write the following:

```
export VISUAL=nano
```

If you want to configure an editor specifically for Git, you can either set the **core.editor** configuration value or the **GIT_EDITOR** environment variable. You can see **git-var**(1) for details on the order in which these options are consulted.

Note that in all cases, the editor value will be passed to the shell, so any arguments containing spaces should be appropriately quoted. Additionally, if your editor normally detaches from the terminal when invoked, you should specify it with an argument that makes it not do that, or else Git will not see any changes. An example of a configuration addressing both of these issues on Windows would be the configuration **"C:\Program Files\Vim\gvim.exe" --nofork**, which quotes the filename with spaces and specifies the **--nofork** option to avoid backgrounding the process.

CREDENTIALS

How do I specify my credentials when pushing over HTTP?

The easiest way to do this is to use a credential helper via the **credential.helper** configuration. Most systems provide a standard choice to integrate with the system credential manager. For example, Git for Windows provides the **wincred** credential manager, macOS has the **osxkeychain** credential manager, and Unix systems with a standard desktop environment can use the **libsecret** credential manager. All of these store credentials in an encrypted store to keep your passwords or tokens secure.

In addition, you can use the **store** credential manager which stores in a file in your home directory, or the **cache** credential manager, which does not permanently store your credentials, but does prevent you from being prompted for them for a certain period of time.

You can also just enter your password when prompted. While it is possible to place the password (which must be percent-encoded) in the URL, this is not particularly secure and can lead to accidental exposure of credentials, so it is not recommended.

How do I read a password or token from an environment variable?

The **credential.helper** configuration option can also take an arbitrary shell command that produces

the credential protocol on standard output. This is useful when passing credentials into a container, for example.

Such a shell command can be specified by starting the option value with an exclamation point. If your password or token were stored in the **GIT_TOKEN**, you could run the following command to set your credential helper:

```
$ git config credential.helper \
    '!f() { echo username=author; echo "password=$GIT_TOKEN"; };f'
```

How do I change the password or token I've saved in my credential manager?

Usually, if the password or token is invalid, Git will erase it and prompt for a new one. However, there are times when this doesn't always happen. To change the password or token, you can erase the existing credentials and then Git will prompt for new ones. To erase credentials, use a syntax like the following (substituting your username and the hostname):

```
$ echo url=https://author@git.example.org | git credential reject
```

How do I use multiple accounts with the same hosting provider using HTTP?

Usually the easiest way to distinguish between these accounts is to use the username in the URL. For example, if you have the accounts **author** and **committer** on **git.example.org**, you can use the URLs **https://author@git.example.org/org1/project1.git** and **https://committer@git.example.org/org2/project2.git**. This way, when you use a credential helper, it will automatically try to look up the correct credentials for your account. If you already have a remote set up, you can change the URL with something like **git remote set-url origin https://author@git.example.org/org1/project1.git** (see **git-remote**(1) for details).

How do I use multiple accounts with the same hosting provider using SSH?

With most hosting providers that support SSH, a single key pair uniquely identifies a user. Therefore, to use multiple accounts, it's necessary to create a key pair for each account. If you're using a reasonably modern OpenSSH version, you can create a new key pair with something like **ssh-keygen -t ed25519 -f ~/.ssh/id_committer**. You can then register the public key (in this case, **~/.ssh/id_committer.pub**; note the **.pub**) with the hosting provider.

Most hosting providers use a single SSH account for pushing; that is, all users push to the **git** account (e.g., **git@git.example.org**). If that's the case for your provider, you can set up multiple aliases in SSH to make it clear which key pair to use. For example, you could write something like the following in **~/.ssh/config**, substituting the proper private key file:

```
# This is the account for author on git.example.org.
Host example_author
    HostName git.example.org
    User git
    # This is the key pair registered for author with git.example.org.
    IdentityFile ~/.ssh/id_author
    IdentitiesOnly yes
# This is the account for committer on git.example.org.
Host example_committer
    HostName git.example.org
    User git
    # This is the key pair registered for committer with git.example.org.
    IdentityFile ~/.ssh/id_committer
    IdentitiesOnly yes
```

Then, you can adjust your push URL to use **git@example_author** or **git@example_committer** instead of **git@example.org** (e.g., **git remote set-url git@example_author:org1/project1.git**).

COMMON ISSUES

I've made a mistake in the last commit. How do I change it?

You can make the appropriate change to your working tree, run **git add <file>** or **git rm <file>**, as appropriate, to stage it, and then **git commit --amend**. Your change will be included in the commit, and you'll be prompted to edit the commit message again; if you wish to use the original message verbatim, you can use the **--no-edit** option to **git commit** in addition, or just save and quit when your editor opens.

I've made a change with a bug and it's been included in the main branch. How should I undo it?

The usual way to deal with this is to use **git revert**. This preserves the history that the original change was made and was a valuable contribution, but also introduces a new commit that undoes those changes because the original had a problem. The commit message of the revert indicates the commit which was reverted and is usually edited to include an explanation as to why the revert was made.

How do I ignore changes to a tracked file?

Git doesn't provide a way to do this. The reason is that if Git needs to overwrite this file, such as during a checkout, it doesn't know whether the changes to the file are precious and should be kept, or whether they are irrelevant and can safely be destroyed. Therefore, it has to take the safe route and always preserve them.

It's tempting to try to use certain features of **git update-index**, namely the assume-unchanged and

skip-worktree bits, but these don’t work properly for this purpose and shouldn’t be used this way.

If your goal is to modify a configuration file, it can often be helpful to have a file checked into the repository which is a template or set of defaults which can then be copied alongside and modified as appropriate. This second, modified file is usually ignored to prevent accidentally committing it.

I asked Git to ignore various files, yet they are still tracked

A **gitignore** file ensures that certain file(s) which are not tracked by Git remain untracked. However, sometimes particular file(s) may have been tracked before adding them into the **.gitignore**, hence they still remain tracked. To untrack and ignore files/patterns, use **git rm --cached <file/pattern>** and add a pattern to **.gitignore** that matches the <file>. See **gitignore**(5) for details.

How do I know if I want to do a fetch or a pull?

A fetch stores a copy of the latest changes from the remote repository, without modifying the working tree or current branch. You can then at your leisure inspect, merge, rebase on top of, or ignore the upstream changes. A pull consists of a fetch followed immediately by either a merge or rebase. See **git-pull**(1).

MERGING AND REBASING

What kinds of problems can occur when merging long-lived branches with squash merges?

In general, there are a variety of problems that can occur when using squash merges to merge two branches multiple times. These can include seeing extra commits in **git log** output, with a GUI, or when using the **...** notation to express a range, as well as the possibility of needing to re-resolve conflicts again and again.

When Git does a normal merge between two branches, it considers exactly three points: the two branches and a third commit, called the *merge base*, which is usually the common ancestor of the commits. The result of the merge is the sum of the changes between the merge base and each head. When you merge two branches with a regular merge commit, this results in a new commit which will end up as a merge base when they’re merged again, because there is now a new common ancestor. Git doesn’t have to consider changes that occurred before the merge base, so you don’t have to re-resolve any conflicts you resolved before.

When you perform a squash merge, a merge commit isn’t created; instead, the changes from one side are applied as a regular commit to the other side. This means that the merge base for these branches won’t have changed, and so when Git goes to perform its next merge, it considers all of the changes that it considered the last time plus the new changes. That means any conflicts may need to be re-resolved. Similarly, anything using the **...** notation in **git diff**, **git log**, or a GUI will result in showing all of the changes since the original merge base.

As a consequence, if you want to merge two long-lived branches repeatedly, it's best to always use a regular merge commit.

If I make a change on two branches but revert it on one, why does the merge of those branches include the change?

By default, when Git does a merge, it uses a strategy called the **ort** strategy, which does a fancy three-way merge. In such a case, when Git performs the merge, it considers exactly three points: the two heads and a third point, called the *merge base*, which is usually the common ancestor of those commits. Git does not consider the history or the individual commits that have happened on those branches at all.

As a result, if both sides have a change and one side has reverted that change, the result is to include the change. This is because the code has changed on one side and there is no net change on the other, and in this scenario, Git adopts the change.

If this is a problem for you, you can do a rebase instead, rebasing the branch with the revert onto the other branch. A rebase in this scenario will revert the change, because a rebase applies each individual commit, including the revert. Note that rebases rewrite history, so you should avoid rebasing published branches unless you're sure you're comfortable with that. See the NOTES section in **git-rebase**(1) for more details.

HOOKS

How do I use hooks to prevent users from making certain changes?

The only safe place to make these changes is on the remote repository (i.e., the Git server), usually in the **pre-receive** hook or in a continuous integration (CI) system. These are the locations in which policy can be enforced effectively.

It's common to try to use **pre-commit** hooks (or, for commit messages, **commit-msg** hooks) to check these things, which is great if you're working as a solo developer and want the tooling to help you. However, using hooks on a developer machine is not effective as a policy control because a user can bypass these hooks with **--no-verify** without being noticed (among various other ways). Git assumes that the user is in control of their local repositories and doesn't try to prevent this or tattle on the user.

In addition, some advanced users find **pre-commit** hooks to be an impediment to workflows that use temporary commits to stage work in progress or that create fixup commits, so it's better to push these kinds of checks to the server anyway.

CROSS-PLATFORM ISSUES

I'm on Windows and my text files are detected as binary.

Git works best when you store text files as UTF-8. Many programs on Windows support UTF-8, but some do not and only use the little-endian UTF-16 format, which Git detects as binary. If you can't use UTF-8 with your programs, you can specify a working tree encoding that indicates which encoding your files should be checked out with, while still storing these files as UTF-8 in the repository. This allows tools like **git-diff**(1) to work as expected, while still allowing your tools to work.

To do so, you can specify a **gitattributes**(5) pattern with the **working-tree-encoding** attribute. For example, the following pattern sets all C files to use UTF-16LE-BOM, which is a common encoding on Windows:

```
*.c     working-tree-encoding=UTF-16LE-BOM
```

You will need to run **git add --renormalize** to have this take effect. Note that if you are making these changes on a project that is used across platforms, you'll probably want to make it in a per-user configuration file or in the one in **$GIT_DIR/info/attributes**, since making it in a **.gitattributes** file in the repository will apply to all users of the repository.

See the following entry for information about normalizing line endings as well, and see **gitattributes**(5) for more information about attribute files.

I'm on Windows and git diff shows my files as having a **^M** at the end.

By default, Git expects files to be stored with Unix line endings. As such, the carriage return (**^M**) that is part of a Windows line ending is shown because it is considered to be trailing whitespace. Git defaults to showing trailing whitespace only on new lines, not existing ones.

You can store the files in the repository with Unix line endings and convert them automatically to your platform's line endings. To do that, set the configuration option **core.eol** to **native** and see the following entry for information about how to configure files as text or binary.

You can also control this behavior with the **core.whitespace** setting if you don't wish to remove the carriage returns from your line endings.

Why do I have a file that's always modified?

Internally, Git always stores file names as sequences of bytes and doesn't perform any encoding or case folding. However, Windows and macOS by default both perform case folding on file names. As a result, it's possible to end up with multiple files or directories whose names differ only in case. Git can handle this just fine, but the file system can store only one of these files, so when Git reads the other file to see its contents, it looks modified.

It’s best to remove one of the files such that you only have one file. You can do this with commands like the following (assuming two files **AFile.txt** and **afile.txt**) on an otherwise clean working tree:

```
$ git rm --cached AFile.txt
$ git commit -m ’Remove files conflicting in case’
$ git checkout .
```

This avoids touching the disk, but removes the additional file. Your project may prefer to adopt a naming convention, such as all-lowercase names, to avoid this problem from occurring again; such a convention can be checked using a **pre-receive** hook or as part of a continuous integration (CI) system.

It is also possible for perpetually modified files to occur on any platform if a smudge or clean filter is in use on your system but a file was previously committed without running the smudge or clean filter. To fix this, run the following on an otherwise clean working tree:

```
$ git add --renormalize .
```

What’s the recommended way to store files in Git?

While Git can store and handle any file of any type, there are some settings that work better than others. In general, we recommend that text files be stored in UTF-8 without a byte-order mark (BOM) with LF (Unix-style) endings. We also recommend the use of UTF-8 (again, without BOM) in commit messages. These are the settings that work best across platforms and with tools such as **git diff** and **git merge**.

Additionally, if you have a choice between storage formats that are text based or non-text based, we recommend storing files in the text format and, if necessary, transforming them into the other format. For example, a text-based SQL dump with one record per line will work much better for diffing and merging than an actual database file. Similarly, text-based formats such as Markdown and AsciiDoc will work better than binary formats such as Microsoft Word and PDF.

Similarly, storing binary dependencies (e.g., shared libraries or JAR files) or build products in the repository is generally not recommended. Dependencies and build products are best stored on an artifact or package server with only references, URLs, and hashes stored in the repository.

We also recommend setting a **gitattributes**(5) file to explicitly mark which files are text and which are binary. If you want Git to guess, you can set the attribute **text=auto**. For example, the following might be appropriate in some projects:

```
# By default, guess.
*       text=auto
# Mark all C files as text.
*.c     text
# Mark all JPEG files as binary.
*.jpg   binary
```

These settings help tools pick the right format for output such as patches and result in files being checked out in the appropriate line ending for the platform.

GIT

Part of the **git**(1) suite