SE 2142

Git ignore

Git sees every file in your working copy as one of three things:

1. tracked - a file which has been previously staged or committed;

2. untracked - a file which has not been staged or committed; or

3. ignored - a file which Git has been explicitly told to ignore.

Ignored files are usually build artifacts and machine generated files that can be derived from your repository source or should otherwise not be committed. Some common examples are:

- dependency caches, such as the contents of `/node_modules` or `/packages`
- compiled code, such as `.o`, `.pyc`, and `.class` files
- build output directories, such as `/bin`, `/out`, or `/target`
- files generated at runtime, such as `.log`, `.lock`, or `.tmp`
- hidden system files, such as `.DS_Store` or `Thumbs.db`
- personal IDE config files, such as `.idea/workspace.xml`

Ignored files are tracked in a special file named `.gitignore` that is checked in at the root of your repository.

Git ignore patterns

`.gitignore` uses globbing patterns to match against file names. You can construct your patterns using various symbols:

Pattern	Example matches	Explanation*

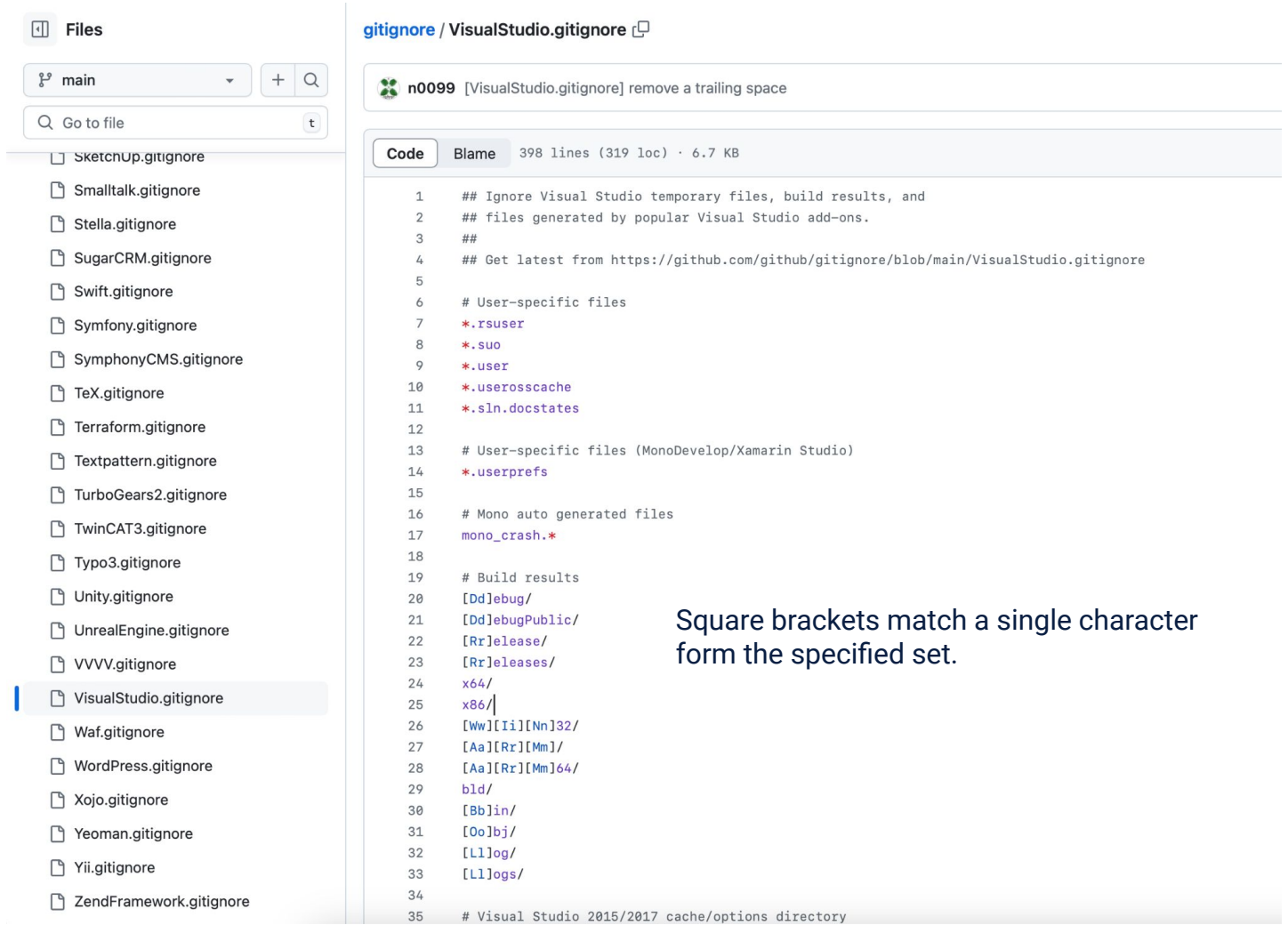
`**/logs`	`logs/debug.log`
	`logs/monday/foo.bar`
	`build/logs/debug.log`
		You can prepend a pattern with a double asterisk to match directories anywhere in the repository.

`**/logs/debug.log`	`logs/debug.log`
	`build/logs/debug.log`
	but not
	`logs/build/debug.log`
		You can also use a double asterisk to match files based on their name and the name of their parent directory.

`*.log`	`debug.log`
	`foo.log`
	`.log`
	`logs/debug.log`
		An asterisk is a wildcard that matches zero or more characters.

`*.log`
`!important.log`
	`debug.log`
	but not
	`logs/debug.log`
		Prepending an exclamation mark to a pattern negates it. If a file matches a pattern, but *also* matches a negating pattern defined later in the file, it will not be ignored.

`/debug.log`	`debug.log` *but not* `logs/debug.log`	Patterns defined after a negating pattern will re-ignore any previously negated files.
`debug.log`	`debug.log` `logs/debug.log`	Prepending a slash matches files only in the repository root.
`debug?.log`	`debug0.log` `debugg.log` *but not* `debug10.log`	A question mark matches exactly one character.
`debug[0-9].log`	`debug0.log` `debug1.log` *but not* `debug10.log`	Square brackets can also be used to match a single character from a specified range.
`debug[01].log`	`debug0.log` `debug1.log` *but not* `debug2.log` `debug01.log`	Square brackets match a single character form the specified set.


Files
main
Go to file
t
SketchUp.gitignore
Smalltalk.gitignore
Stella.gitignore
SugarCRM.gitignore
Swift.gitignore
Symfony.gitignore
SymphonyCMS.gitignore
TeX.gitignore
Terraform.gitignore
Textpattern.gitignore
TurboGears2.gitignore
TwinCAT3.gitignore
Typo3.gitignore
Unity.gitignore
UnrealEngine.gitignore
VVVV.gitignore
VisualStudio.gitignore
Waf.gitignore
WordPress.gitignore
Xojo.gitignore
Yeoman.gitignore
Yii.gitignore
ZendFramework.gitignore
gitignore / VisualStudio.gitignore
n0099 [VisualStudio.gitignore] remove a trailing space
Code
Blame
398 lines (319 loc) · 6.7 KB
1 ## Ignore Visual Studio temporary files, build results, and
2 ## files generated by popular Visual Studio add-ons.
3 ##
4 ## Get latest from https://github.com/github/gitignore/blob/main/VisualStudio.gitignore
5
6 # User-specific files
7 *.rsuser
8 *.suo
9 *.user
10 *.userosscache
11 *.sln.docstates
12
13 # User-specific files (MonoDevelop/Xamarin Studio)
14 *.userprefs
15
16 # Mono auto generated files
17 mono_crash.*
18
19 # Build results
20 [Dd]ebug/
21 [Dd]ebugPublic/
22 [Rr]elease/
23 [Rr]eleases/
24 x64/
25 x86/
26 [Ww][Ii][Nn]32/
27 [Aa][Rr][Mm]/
28 [Aa][Rr][Mm]64/
29 bld/
30 [Bb]in/
31 [Oo]bj/
32 [Ll]og/
33 [Ll]ogs/
34
35 # Visual Studio 2015/2017 cache/options directory
Square brackets match a single character form the specified set.

Shared .gitignore files in your repository

Git ignore rules are usually defined in a .gitignore file at the root of your repository.

- simplest approach, is to define a single .gitignore file in the root.
- as your .gitignore file is checked in, it is versioned like any other file in your repository and shared with your teammates when you push.
- only include patterns in .gitignore that will benefit other users of the repository.

Personal Git ignore rules

.git/info/exclude

These are not versioned, and not distributed with your repository, so it's an appropriate place to include patterns that will likely only benefit you.

Global Git ignore rules

In addition, you can define global Git ignore patterns for all repositories on your local system by setting the Git **core.excludesFile** property.

Once you've created the file, you'll need to configure its location with git config:

```
$ touch ~/.gitignore
$ git config --global core.excludesFile ~/.gitignore
```

Note:

You should be careful what patterns you choose to globally ignore, as different file types are relevant for different projects.

Special operating system files or temporary files created by some developer tools are typical candidates for ignoring globally.

Ignoring a previously committed file

If you want to ignore a file that you've committed in the past, you'll need to delete the file from your repository and then add a .gitignore rule for it.

```
$ echo debug.log >> .gitignore

$ git rm --cached debug.log
rm 'debug.log'

$ git commit -m "Start ignoring debug.log"
```

You can omit the `--cached` option if you want to delete the file from both the repository and your local file system.

Committing an ignored file

Force an ignored file to be committed to the repository using the -f (or --force) option with git add:x

```
$ cat .gitignore
*.log

$ git add -f debug.log

$ git commit -m "Force adding debug.log"
```

Commit a specific file. A better solution is to define an exception to the general rule:

```
$ echo !debug.log >> .gitignore

$ cat .gitignore
*.log
!debug.log

$ git add debug.log

$ git commit -m "Adding debug.log"
```

Prepending an exclamation mark to a pattern negates it. If a file matches a pattern, but *also* matches a negating pattern defined later in the file, it will not be ignored.

Stashing an ignored file

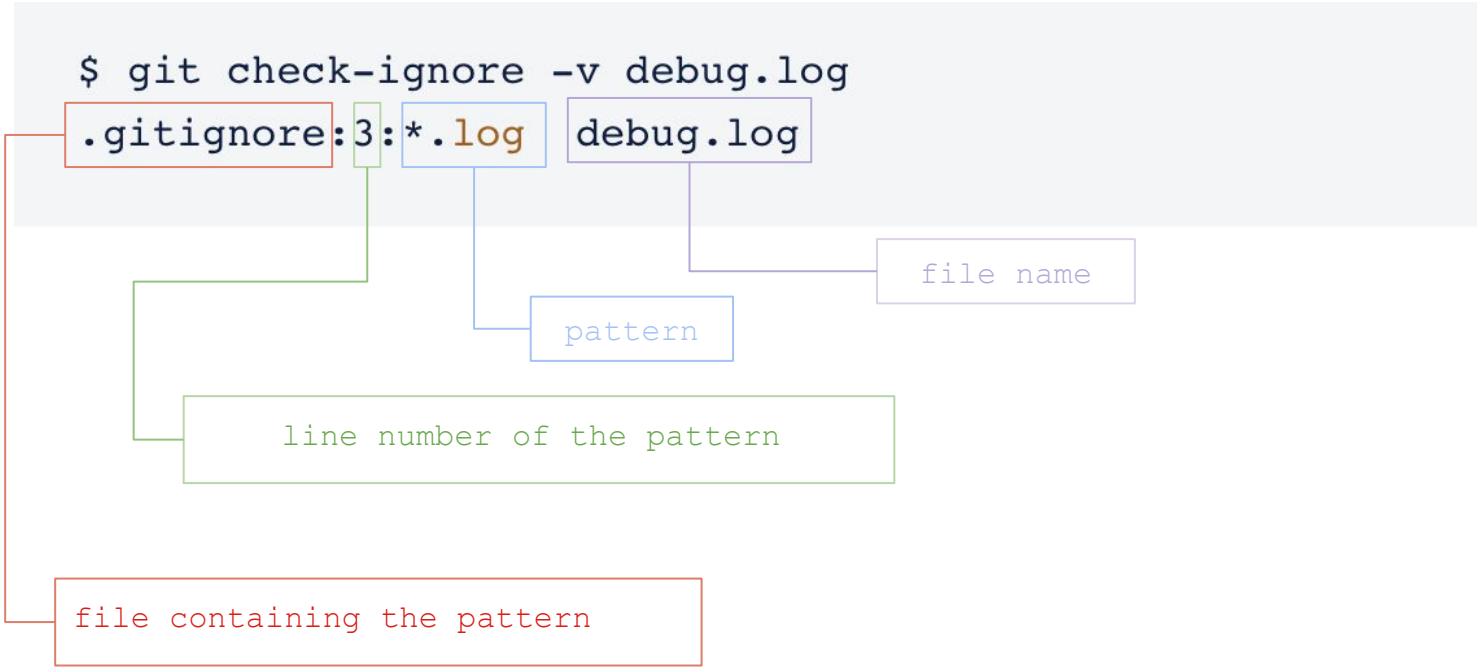
`git stash` is a powerful Git feature for temporarily shelving and reverting local changes, allowing you to re-apply them later on.

By default `git stash` ignores ignored files and only stashes changes to files that are tracked by Git. However, you can invoke `git stash with the --all option` to stash changes to ignored and untracked files as well.

`git stash` command takes your uncommitted changes (both staged and unstaged), saves them away for later use, and then reverts them from your working copy.

```
$ git status
On branch main
Changes to be committed:

        new file:   style.css

Changes not staged for commit:

    modified:   index.html

$ git stash
Saved working directory and index state WIP on main: 5002d47 our new homepage
HEAD is now at 5002d47 our new homepage

$ git status
On branch main
nothing to commit, working tree clean
```

Debugging .gitignore files

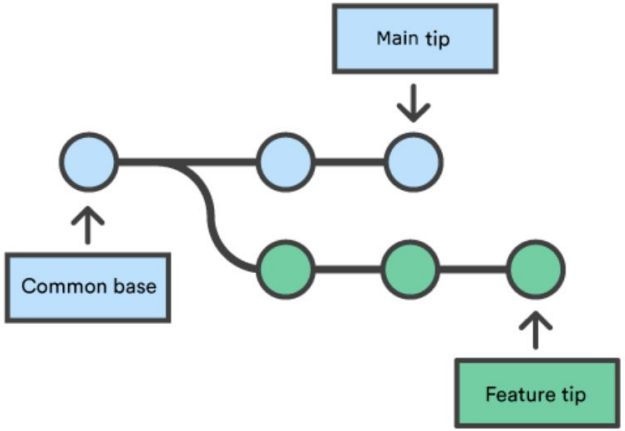


Git merge

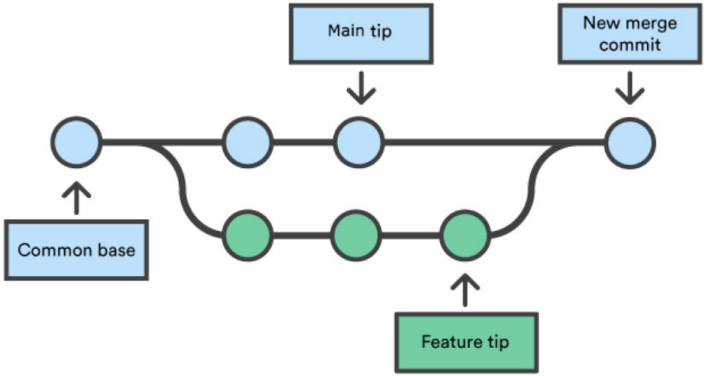
The git merge command lets you take the independent lines of development created by git branch and integrate them into a single branch.

Git merge will combine multiple sequences of commits into one unified history.

Before



After



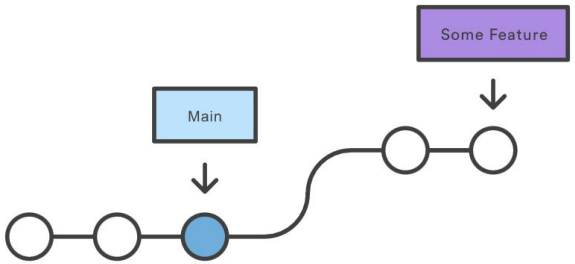
There are two main ways Git will merge:

- Fast Forward
- Three way

Fast forward merge

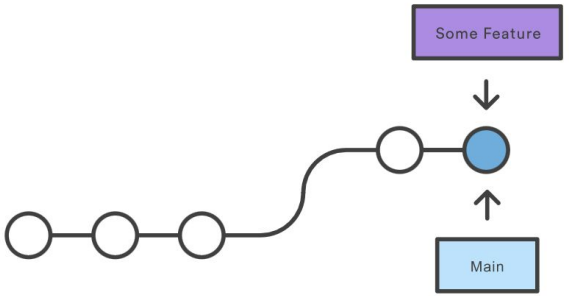
A fast-forward merge can occur when there is a **linear path** from the current branch tip to the target branch.

Before Merging




```
main:     A---B
             \
feature:     C---D---E
```

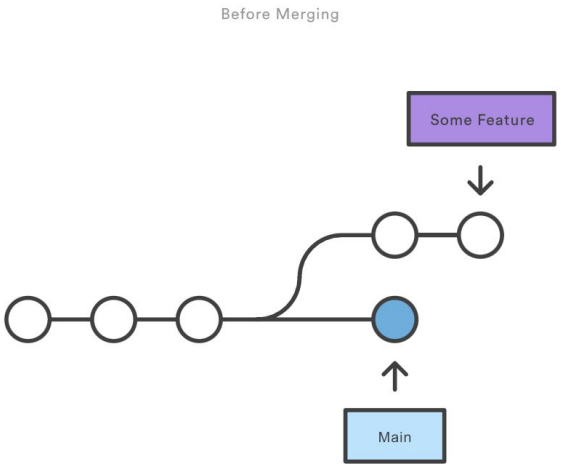
After a Fast-Forward Merge

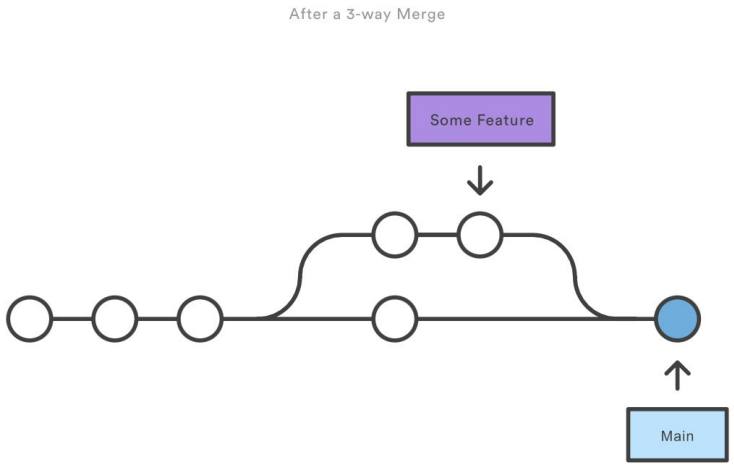



```
main:     A---B---C---D---E
```

3-way merge

When there is not a linear path to the target branch, Git has no choice but to combine them via a 3-way merge. 3-way merges use a dedicated commit to tie together the two histories.





How conflicts are presented

When Git encounters a conflict during a merge, It will edit the content of the affected files with visual indicators that mark both sides of the conflicted content. These visual markers are: <<<<<<<, =======, and >>>>>>>.

```
here is some content not affected by the conflict
<<<<<<< main
this is conflicted text from main
=======
this is conflicted text from feature branch
>>>>>>> feature branch;
```

Git merge conflicts

Version control systems are all about managing contributions between multiple distributed authors (usually developers).

The `git merge` command's primary responsibility is to combine separate branches and resolve any conflicting edits.

Understanding merge conflicts

Conflicts generally arise when two people have changed the same lines in a file, or if one developer deleted a file while another developer was modifying it.

In these cases, Git cannot automatically determine what is correct.

Types of merge conflicts

Git fails to start the merge

A merge will fail to start when Git sees there are changes in either the working directory or staging area of the current project.

Git fails during the merge

A failure DURING a merge indicates a conflict between the current local branch and the branch being merged.

Git commands that can help resolve merge conflicts

General tools

```
git status
```

The status command is in frequent use when a working with Git and during a merge it will help identify conflicted files.

```
git log --merge
```

Passing the --merge argument to the git log command will produce a log with a list of commits that conflict between the merging branches.

```
git diff
```

diff helps find differences between states of a repository/files. This is useful in predicting and preventing merge conflicts.

Tools for when git fails to start a merge

`git checkout`

checkout can be used for *undoing* changes to files, or for changing branches

`git reset --mixed`

reset can be used to undo changes to the working directory and staging area.

Tools for when git conflicts arise during a merge

```
git merge --abort
```

Executing git merge with the --abort option will exit from the merge process and return the branch to the state before the merge began.

```
git reset
```

Git reset can be used during a merge conflict to reset conflicted files to a know good state

A conflict arises when two separate branches have made edits to the same line in a file, or when a file has been deleted in one branch but edited in the other. Conflicts will most likely happen when working in a team environment.

Review/Break