

8.2 Customizing Git - Git Attributes

Git Attributes

Some of these settings can also be specified for a path, so that Git applies those settings only for a subdirectory or subset of files. These path-specific settings are called Git attributes and are set either in a `.gitattributes` file in one of your directories (normally the root of your project) or in the `.git/info/attributes` file if you don't want the attributes file committed with your project.

Using attributes, you can do things like specify separate merge strategies for individual files or directories in your project, tell Git how to diff non-text files, or have Git filter content before you check it into or out of Git. In this section, you'll learn about some of the attributes you can set on your paths in your Git project and see a few examples of using this feature in practice.

Binary Files

One cool trick for which you can use Git attributes is telling Git which files are binary (in cases it otherwise may not be able to figure out) and giving Git special instructions about how to handle those files. For instance, some text files may be machine generated and not diffable, whereas some binary files can be diffed. You'll see how to tell Git which is which.

Identifying Binary Files

Some files look like text files but for all intents and purposes are to be treated as binary data. For instance, Xcode projects on macOS contain a file that ends in `.pbxproj`, which is basically a JSON (plain-text JavaScript data format) dataset written out to disk by the IDE, which records your build settings and so on. Although it's technically a text file (because it's all UTF-8), you don't want to treat it as such because it's really a lightweight database – you can't merge the contents if two people change it, and diffs generally aren't helpful. The file is meant to be consumed by a machine. In essence, you want to treat it like a binary file.

To tell Git to treat all `pbxproj` files as binary data, add the following line to your `.gitattributes` file:

```
*.pbxproj binary
```

Now, Git won't try to convert or fix CRLF issues; nor will it try to compute or print a diff for changes in this file when you run `git show` or `git diff` on your project.

Diffing Binary Files

You can also use the Git attributes functionality to effectively diff binary files. You do this by telling Git how to convert your binary data to a text format that can be compared via the normal diff.

First, you'll use this technique to solve one of the most annoying problems known to humanity: version-controlling Microsoft Word documents. Everyone knows that Word is the most horrific editor around, but oddly, everyone still uses it. If you want to version-control Word documents, you can stick them in a Git repository and commit every once in a while; but what good does that do? If you run `git diff` normally, you only see something like this:

```
$ git diff
diff --git a/chapter1.docx b/chapter1.docx
index 88839c4..4afc7c 100644
Binary files a/chapter1.docx and b/chapter1.docx differ
```

You can't directly compare two versions unless you check them out and scan them manually, right? It turns out you can do this fairly well using Git attributes. Put the following line in your `.gitattributes` file:

```
*.docx diff=word
```

This tells Git that any file that matches this pattern (`.docx`) should use the “word” filter when you try to view a diff that contains changes. What is the “word” filter? You have to set it up. Here you'll configure Git to use the `docx2txt` program to convert Word documents into readable text files, which it will then diff properly.

First, you'll need to install `docx2txt`; you can download it from <https://sourceforge.net/projects/docx2txt>. Follow the instructions in the `INSTALL` file to put it somewhere your shell can find it. Next, you'll write a wrapper script to convert output to the format Git expects. Create a file that's somewhere in your path called `docx2txt`, and add these contents:

```
#!/bin/bash
docx2txt.pl "$1" -
```

Don't forget to `chmod a+x` that file. Finally, you can configure Git to use this script:

```
$ git config diff.word.textconv docx2txt
```

Now Git knows that if it tries to do a diff between two snapshots, and any of the files end in `.docx`, it should run those files through the “word” filter, which is defined as the `docx2txt` program. This effectively makes nice text-based versions of your Word files before attempting to diff them.

Here's an example: Chapter 1 of this book was converted to Word format and committed in a Git repository. Then a new paragraph was added. Here's what `git diff` shows:

```
$ git diff
diff --git a/chapter1.docx b/chapter1.docx
index 0b013ca..ba25db5 100644
--- a/chapter1.docx
+++ b/chapter1.docx
@@ -2,6 +2,7 @@
 This chapter will be about getting started with Git. We will begin at the beginning by explaining some background on version control tools, then move
 1.1. About Version Control
 What is "version control", and why should you care? Version control is a system that records changes to a file or set of files over time so that you
+Testing: 1, 2, 3.
 If you are a graphic or web designer and want to keep every version of an image or layout (which you would most certainly want to), a Version Control
 1.1.1. Local Version Control Systems
 Many people's version-control method of choice is to copy files into another directory (perhaps a time-stamped directory, if they're clever). This ap
```

Git successfully and succinctly tells us that we added the string “Testing: 1, 2, 3.”, which is correct. It's not perfect – formatting changes wouldn't show up here – but it certainly works.

Another interesting problem you can solve this way involves diffing image files. One way to do this is to run image files through a filter that extracts their EXIF information – metadata that is recorded with most image formats. If you download and install the `exiftool` program, you can use it to convert your images into text

about the metadata, so at least the diff will show you a textual representation of any changes that happened. Put the following line in your `.gitattributes` file:

```
*.png diff=exif
```

Configure Git to use this tool:

```
$ git config diff.exif.textconv exiftool
```

If you replace an image in your project and run `git diff`, you see something like this:

```
diff --git a/image.png b/image.png
index 88839c4..4afcb7c 100644
--- a/image.png
+++ b/image.png
@@ -1,12 +1,12 @@
ExifTool Version Number      : 7.74
-File Size                   : 70 kB
-File Modification Date/Time : 2009:04:21 07:02:45-07:00
+File Size                   : 94 kB
+File Modification Date/Time : 2009:04:21 07:02:43-07:00
File Type                    : PNG
MIME Type                    : image/png
-Image Width                 : 1058
-Image Height                : 889
+Image Width                 : 1056
+Image Height                : 827
Bit Depth                    : 8
Color Type                   : RGB with Alpha
```

You can easily see that the file size and image dimensions have both changed.

Keyword Expansion

SVN- or CVS-style keyword expansion is often requested by developers used to those systems. The main problem with this in Git is that you can't modify a file with information about the commit after you've committed, because Git checksums the file first. However, you can inject text into a file when it's checked out and remove it again before it's added to a commit. Git attributes offers you two ways to do this.

First, you can inject the SHA-1 checksum of a blob into an `Id` field in the file automatically. If you set this attribute on a file or set of files, then the next time you check out that branch, Git will replace that field with the SHA-1 of the blob. It's important to notice that it isn't the SHA-1 of the commit, but of the blob itself. Put the following line in your `.gitattributes` file:

```
*.txt ident
```

Add an `Id` reference to a test file:

```
$ echo '$Id$' > test.txt
```

The next time you check out this file, Git injects the SHA-1 of the blob:

```
$ rm test.txt
$ git checkout -- test.txt
$ cat test.txt
$Id: 42812b7653c7b88933f8a9d6cad0ca16714b9bb3 $
```

However, that result is of limited use. If you've used keyword substitution in CVS or Subversion, you can include a timestamp – the SHA-1 isn't all that helpful, because it's fairly random and you can't tell if one SHA-1 is older or newer than another just by looking at them.

It turns out that you can write your own filters for doing substitutions in files on commit/checkout. These are called “clean” and “smudge” filters. In the `.gitattributes` file, you can set a filter for particular paths and then set up scripts that will process files just before they're checked out (“smudge”, see [The “smudge” filter is run on checkout](#)) and just before they're staged (“clean”, see [The “clean” filter is run when files are staged](#)). These filters can be set to do all sorts of fun things.

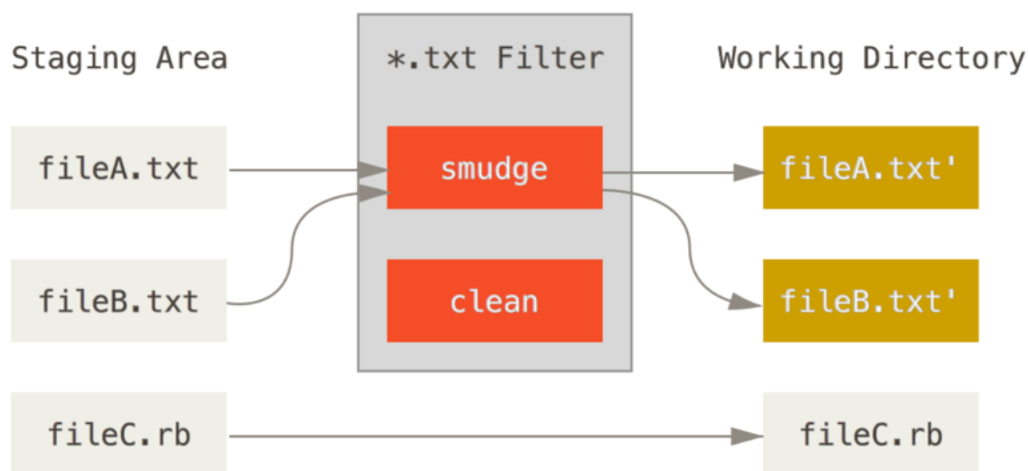


Figure 143. The “smudge” filter is run on checkout

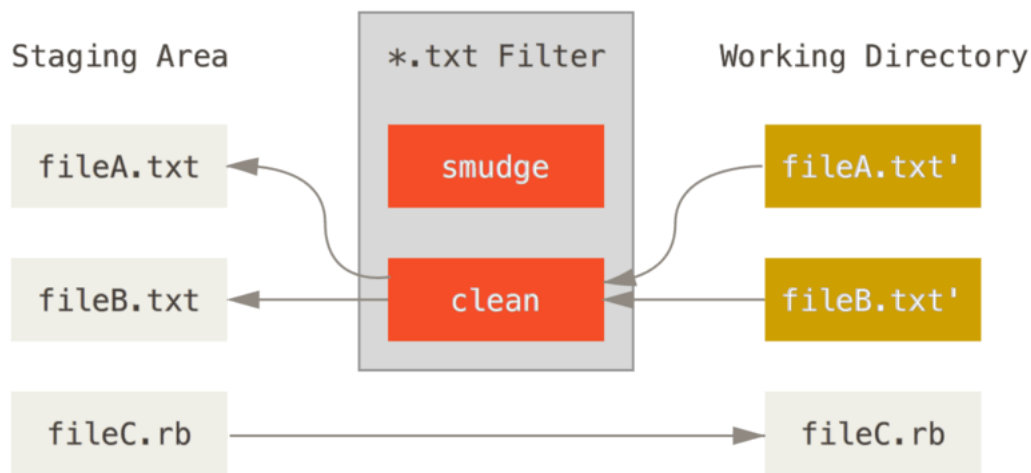


Figure 144. The “clean” filter is run when files are staged

The original commit message for this feature gives a simple example of running all your C source code through the `indent` program before committing. You can set it up by setting the filter attribute in your `.gitattributes` file to filter `*.c` files with the “indent” filter:

```
*.c filter=indent
```

Then, tell Git what the “indent” filter does on smudge and clean:

```
$ git config --global filter.indent.clean indent
$ git config --global filter.indent.smudge cat
```

In this case, when you commit files that match `*.c`, Git will run them through the `indent` program before it stages them and then run them through the `cat` program before it checks them back out onto disk. The `cat` program does essentially nothing: it spits out the same data that it comes in. This combination effectively filters all C source code files through `indent` before committing.

Another interesting example gets `$Date$` keyword expansion, RCS style. To do this properly, you need a small script that takes a filename, figures out the last commit date for this project, and inserts the date into the file. Here is a small Ruby script that does that:

```
#!/usr/bin/env ruby
data = STDIN.read
last_date = `git log --pretty=format:"%ad" -1`
puts data.gsub('$Date$', 'Date: ' + last_date.to_s + '$')
```

All the script does is get the latest commit date from the `git log` command, stick that into any `$Date$` strings it sees in stdin, and print the results – it should be simple to do in whatever language you’re most comfortable in. You can name this file `expand_date` and put it in your path. Now, you need to set up a filter in Git (call it `dater`) and tell it to use your `expand_date` filter to smudge the files on checkout. You’ll use a Perl expression to clean that up on commit:

```
$ git config filter.dater.smudge expand_date
$ git config filter.dater.clean 'perl -pe "s/\\$Date[^\$]*\\$/\\$Date\\$/'"
```

This Perl snippet strips out anything it sees in a `$Date$` string, to get back to where you started. Now that your filter is ready, you can test it by setting up a Git attribute for that file that engages the new filter and creating a file with your `$Date$` keyword:

```
date*.txt filter=dater
$ echo '# $Date$' > date_test.txt
```

If you commit those changes and check out the file again, you see the keyword properly substituted:

```
$ git add date_test.txt .gitattributes
$ git commit -m "Test date expansion in Git"
$ rm date_test.txt
$ git checkout date_test.txt
$ cat date_test.txt
# $Date: Tue Apr 21 07:26:52 2009 -0700$
```

You can see how powerful this technique can be for customized applications. You have to be careful, though, because the `.gitattributes` file is committed and passed around with the project, but the driver (in this case, `dater`) isn’t, so it won’t work everywhere. When you design these filters, they should be able to fail gracefully and have the project still work properly.

Exporting Your Repository

Git attribute data also allows you to do some interesting things when exporting an archive of your project.

export-ignore

You can tell Git not to export certain files or directories when generating an archive. If there is a subdirectory or file that you don’t want to include in your archive file but that you do want checked into your project, you can determine those files via the `export-ignore` attribute.

For example, say you have some test files in a `test/` subdirectory, and it doesn’t make sense to include them in the tarball export of your project. You can add the following line to your Git attributes file:

```
test/ export-ignore
```

Now, when you run `git archive` to create a tarball of your project, that directory won’t be included in the archive.

export-subst

When exporting files for deployment you can apply `git log`'s formatting and keyword-expansion processing to selected portions of files marked with the ``export-subst`` attribute.

For instance, if you want to include a file named `LAST_COMMIT` in your project, and have metadata about the last commit automatically injected into it when `git archive` runs, you can for example set up your `.gitattributes` and `LAST_COMMIT` files like this:

```
LAST_COMMIT export-subst

$ echo 'Last commit date: $Format:%cd by %aN$' > LAST_COMMIT
$ git add LAST_COMMIT .gitattributes
$ git commit -am 'adding LAST_COMMIT file for archives'
```

When you run `git archive`, the contents of the archived file will look like this:

```
$ git archive HEAD | tar xCf ../deployment-testing -
$ cat ../deployment-testing/LAST_COMMIT
Last commit date: Tue Apr 21 08:38:48 2009 -0700 by Scott Chacon
```

The substitutions can include for example the commit message and any `git notes`, and `git log` can do simple word wrapping:

```
$ echo '$Format:Last commit: %h by %aN at %cd%n%+w(76,6,9)%B$' > LAST_COMMIT
$ git commit -am 'export-subst uses git log\'\'\'s custom formatter'
```

```
git archive uses git log\'\'\'s `pretty=format:` processor
directly, and strips the surrounding `Format:` and `$`
markup from the output.
```

```
$ git archive @ | tar xF0 - LAST_COMMIT
Last commit: 312ccc8 by Jim Hill at Fri May 8 09:14:04 2015 -0700
export-subst uses git log's custom formatter
```

```
git archive uses git log's `pretty=format:` processor directly, and
strips the surrounding `Format:` and `$` markup from the output.
```

The resulting archive is suitable for deployment work, but like any exported archive it isn't suitable for further development work.

Merge Strategies

You can also use Git attributes to tell Git to use different merge strategies for specific files in your project. One very useful option is to tell Git to not try to merge specific files when they have conflicts, but rather to use your side of the merge over someone else's.

This is helpful if a branch in your project has diverged or is specialized, but you want to be able to merge changes back in from it, and you want to ignore certain files. Say you have a database settings file called `database.xml` that is different in two branches, and you want to merge in your other branch without messing up the database file. You can set up an attribute like this:

```
database.xml merge=ours
```

And then define a dummy `ours` merge strategy with:

```
$ git config --global merge.ours.driver true
```

If you merge in the other branch, instead of having merge conflicts with the `database.xml` file, you see something like this:

```
$ git merge topic
Auto-merging database.xml
Merge made by recursive.
```

In this case, `database.xml` stays at whatever version you originally had.

[prev](#) | [next](#)