Stashing and Cleaning

Often, when you've been working on part of your project, things are in a messy state and you want to switch branches for a bit to work on something else. The problem is, you don't want to do a commit of half-done work just so you can get back to this point later. The answer to this issue is the `git stash`command.

Stashing takes the dirty state of your working directory — that is, your modified tracked files and staged changes — and saves it on a stack of unfinished changes that you can reapply at any time (even on a different branch).

Note

Migrating to `git stash push`

As of late October 2017, there has been extensive discussion on the Git mailing list, wherein the command `git stash save` is being deprecated in favour of the existing alternative `git stash push`. The main reason for this is that `git stash push`introduces the option of stashing selected *pathspecs*, something `git stash save` does not support.

`git stash save` is not going away any time soon, so don't worry about it suddenly disappearing. But you might want to start migrating over to the `push` alternative for the new functionality.

Stashing Your Work

To demonstrate stashing, you'll go into your project and start working on a couple of files and possibly stage one of the changes. If you run `git status`, you can see your dirty state:

```
$ git status

Changes to be committed:

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



      modified:   index.html
```

```
Changes not staged for commit:

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

  (use "git checkout -- <file>..." to discard changes
in working directory)



      modified:   lib/simplegit.rb
```

Now you want to switch branches, but you don't want to commit what you've been working on yet; so you'll stash the changes. To push a new stash onto your stack, run `git stash` or `git stash save`:

```
$ git stash

Saved working directory and index state \

  "WIP on master: 049d078 added the index file"

HEAD is now at 049d078 added the index file

(To restore them type "git stash apply")
```

You can now see that your working directory is clean:

```
$ git status

# On branch master

nothing to commit, working directory clean
```

At this point, you can switch branches and do work elsewhere; your changes are stored on your stack. To see which stashes you've stored, you can use `git stash list`:

```
$ git stash list

stash@{0}: WIP on master: 049d078 added the index file

stash@{1}: WIP on master: c264051 Revert "added
file_size"

stash@{2}: WIP on master: 21d80a5 added number to log
```

In this case, two stashes were done previously, so you have access to three different stashed works. You can reapply the one you just stashed by using the command shown in the help output of the original stash command: `git stash apply`. If you want to apply one of the older stashes, you can specify it by naming it, like this: `git stash apply stash@{2}`. If you don't specify a stash, Git assumes the most recent stash and tries to apply it:

```
$ git stash apply

On branch master

Changes not staged for commit:

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

  (use "git checkout -- <file>..." to discard changes
in working directory)



      modified:   index.html

      modified:   lib/simplegit.rb



no changes added to commit (use "git add" and/or "git
commit -a")
```

You can see that Git re-modifies the files you reverted when you saved the stash. In this case, you had a clean working directory when you tried to apply the stash, and you tried to apply it on the same branch you saved it from. Having a clean working directory and applying it on the same branch aren't necessary to successfully apply a stash. You can save a stash on one branch, switch to another branch later, and try to reapply the changes. You can also have modified and uncommitted files in your working directory when you apply a stash — Git gives you merge conflicts if anything no longer applies cleanly.

The changes to your files were reapplied, but the file you staged before wasn't restaged. To do that, you must run the `git stash apply` command with a `--index` option to tell the command to try to

reapply the staged changes. If you had run that instead, you'd have gotten back to your original position:

```
$ git stash apply --index

On branch master

Changes to be committed:

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



      modified:   index.html



Changes not staged for commit:

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

  (use "git checkout -- <file>..." to discard changes
in working directory)



      modified:   lib/simplegit.rb
```

The apply option only tries to apply the stashed work — you continue to have it on your stack. To remove it, you can run `git stash drop` with the name of the stash to remove:

```
$ git stash list

stash@{0}: WIP on master: 049d078 added the index file

stash@{1}: WIP on master: c264051 Revert "added
file_size"

stash@{2}: WIP on master: 21d80a5 added number to log

$ git stash drop stash@{0}

Dropped stash@{0}
(364e91f3f268f0900bc3ee613f9f733e82aaed43)
```

You can also run `git stash pop` to apply the stash and then immediately drop it from your stack.

Creative Stashing

There are a few stash variants that may also be helpful. The first option that is quite popular is the `--keep-index` option to the `stash save` command. This tells Git to not only include all staged content in the stash being created, but simultaneously leave it in the index.

```
$ git status -s

M  index.html

 M lib/simplegit.rb



$ git stash --keep-index

Saved working directory and index state WIP on master:
1b65b17 added the index file

HEAD is now at 1b65b17 added the index file



$ git status -s

M  index.html
```

Another common thing you may want to do with stash is to stash the untracked files as well as the tracked ones. By default, `git stash` will stash only modified and staged *tracked* files. If you specify `--include-untracked` or `-u`, Git will include untracked files in the stash being created.

```
$ git status -s

M  index.html

 M lib/simplegit.rb

?? new-file.txt
```

```
$ git stash -u

Saved working directory and index state WIP on master:
1b65b17 added the index file

HEAD is now at 1b65b17 added the index file



$ git status -s

$
```

Finally, if you specify the `--patch` flag, Git will not stash everything that is modified but will instead prompt you interactively which of the changes you would like to stash and which you would like to keep in your working directory.

```
$ git stash --patch

diff --git a/lib/simplegit.rb b/lib/simplegit.rb

index 66d332e..8bb5674 100644

--- a/lib/simplegit.rb

+++ b/lib/simplegit.rb

@@ -16,6 +16,10 @@ class SimpleGit

          return `#{git_cmd} 2>&1`.chomp

        end

      end

+

+    def show(treeish = 'master')

+      command("git show #{treeish}")

+    end
```

```
 end

 test

Stash this hunk [y,n,q,a,d,/,e,?]? y



Saved working directory and index state WIP on master:
1b65b17 added the index file
```

Creating a Branch from a Stash

If you stash some work, leave it there for a while, and continue on the branch from which you stashed the work, you may have a problem reapplying the work. If the apply tries to modify a file that you've since modified, you'll get a merge conflict and will have to try to resolve it. If you want an easier way to test the stashed changes again, you can run `git stash branch <branch>`, which creates a new branch for you with your selected branch name, checks out the commit you were on when you stashed your work, reapplies your work there, and then drops the stash if it applies successfully:

```
$ git stash branch testchanges

M     index.html

M     lib/simplegit.rb

Switched to a new branch 'testchanges'

On branch testchanges

Changes to be committed:

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



      modified:   index.html



Changes not staged for commit:
```

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

  (use "git checkout -- <file>..." to discard changes
in working directory)



      modified:   lib/simplegit.rb



Dropped refs/stash@{0}
(29d385a81d163dfd45a452a2ce816487a6b8b014)
```

This is a nice shortcut to recover stashed work easily and work on it in a new branch.

Cleaning your Working Directory

Finally, you may not want to stash some work or files in your working directory, but simply get rid of them. The `git clean` command will do this for you.

Some common reasons for this might be to remove cruft that has been generated by merges or external tools or to remove build artifacts in order to run a clean build.

You'll want to be pretty careful with this command, since it's designed to remove files from your working directory that are not tracked. If you change your mind, there is often no retrieving the content of those files. A safer option is to run `git stash --all` to remove everything but save it in a stash.

Assuming you do want to remove cruft files or clean your working directory, you can do so with `git clean`. To remove all the untracked files in your working directory, you can run `git clean -f -d`, which removes any files and also any subdirectories that become empty as a result. The `-f` means *force* or "really do this".

If you ever want to see what it would do, you can run the command with the `-n` option, which means "do a dry run and tell me what you *would* have removed".

```
$ git clean -d -n

Would remove test.o
```

```
Would remove tmp/
```

By default, the `git clean` command will only remove untracked files that are not ignored. Any file that matches a pattern in your `.gitignore` or other ignore files will not be removed. If you want to remove those files too, such as to remove all `.o` files generated from a build so you can do a fully clean build, you can add a `-x` to the clean command.

```
$ git status -s
 M lib/simplegit.rb
?? build.TMP
?? tmp/

$ git clean -n -d
Would remove build.TMP
Would remove tmp/

$ git clean -n -d -x
Would remove build.TMP
Would remove test.o
Would remove tmp/
```

If you don't know what the `git clean` command is going to do, always run it with a `-n` first to double check before changing the `-n` to a `-f` and doing it for real. The other way you can be careful about the process is to run it with the `-i` or "interactive" flag.
This will run the clean command in an interactive mode.

```
$ git clean -x -i
Would remove the following items:
  build.TMP  test.o
```

```
*** Commands ***

    1: clean                2: filter by pattern    3:
select by numbers    4: ask each             5: quit

    6: help

What now>
```

This way you can step through each file individually or specify patterns for deletion interactively.

Note	There is a quirky situation where you might need to be extra forceful in asking Git to clean your working directory. If you happen to be in a working directory under which you've copied or cloned other Git repositories (perhaps as submodules), even `git clean -fd`will refuse to delete those directories. In cases like that, you need to add a second `-f`option for emphasis.