--everything-is-local

Search entire site...

This book is available in English.

Full translation available in

български език,
Español,
Français,
Ελληνικά,
日本語,
한국어,
Nederlands,
Русский,
Slovenščina,
Tagalog,
Українська
简体中文,

Partial translations available in

Čeština,
Македонски,
Polski,
Српски,
Ўзбекча,
繁體中文,

Translations started for

Беларуская,
Deutsch,
فارسی,
Indonesian,
Italiano,
Bahasa Melayu,
Português (Brasil),
Português (Portugal),
Türkçe.

The source of this book is hosted on GitHub.
Patches, suggestions and comments are welcome.

Chapters ▾

1st Edition

.3 Git Tools - Stashing

Stashing

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.

Stashing Your Work

To demonstrate, 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
# 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)
#
#      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`:

```
$ 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")
```

Your working directory is clean:

```
$ git status
# On branch master
nothing to commit, working directory clean
```

At this point, you can easily 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)
#
#      modified:   index.html
#      modified:   lib/simplegit.rb
#
```

You can see that Git re-modifies the files you uncommitted 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; but 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)
#
#      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.

Un-applying a Stash

In some use case scenarios you might want to apply stashed changes, do some work, but then un-apply those changes that originally came from the stash. Git does not provide such a `stash unapply` command, but it is possible to achieve the effect by simply retrieving the patch associated with a stash and applying it in reverse:

```
$ git stash show -p stash@{0} | git apply -R
```

Again, if you don't specify a stash, Git assumes the most recent stash:

```
$ git stash show -p | git apply -R
```

You may want to create an alias and effectively add a `stash-unapply` command to your Git. For example:

```
$ git config --global alias.stash-unapply '!git stash show -p | git apply -R'
$ git stash apply
$ #... work work work
$ git stash-unapply
```

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`, which creates a new branch for you, 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
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)
#
#      modified:   lib/simplegit.rb
#
Dropped refs/stash@{0} (f0dfc4d5dc332d1cee34a634182e168c4efc3359)
```

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

prev | next