INTRODUCTION TO THE GIT REFERENCE

This is the Git reference site. It is meant to be a quick reference for learning and remembering the most important and commonly used Git commands. The commands are organized into sections of the type of operation you may be trying to do, and will present the common options and commands needed to accomplish these common tasks.

Each section will link to the next section, so it can be used as a tutorial. Every page will also link to more in-depth Git documentation such as the official manual pages and relevant sections in the [**Pro Git book**](http://git-scm.com/book), so you can learn more about any of the commands. First, we'll start with thinking about source code management like Git does.

HOW TO THINK LIKE GIT

The first important thing to understand about Git is that it thinks about version control very differently than Subversion or Perforce or whatever SCM you may be used to. It is often easier to learn Git by trying to forget your assumptions about how version control works and try to think about it in the Git way.

Let's start from scratch. Assume you are designing a new source code management system. How did you do basic version control before you used a tool for it? Chances are that you simply copied your project directory to save what it looked like at that point.

$ cp -R project project.bak

That way, you can easily revert files that get messed up later, or see what you have changed by comparing what the project looks like now to what it looked like when you copied it.

If you are really paranoid, you may do this often, maybe putting the date in the name of the backup:

$ cp -R project project.2010-06-01.bak

In that case, you may have a bunch of snapshots of your project that you can compare and inspect from. You can even use this model to fairly effectively share changes with someone. If you zip up your project at a known state and put it on your website, other developers can download that, change it and send you a patch pretty easily.

$ wget http://example.com/project.2010-06-01.zip

$ unzip project.2010-06-01.zip

$ cp -R project.2010-06-01 project-my-copy

$ cd project-my-copy

$ (change something)

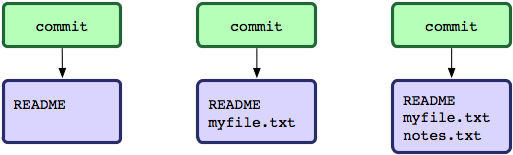
$ diff project-my-copy project.2010-06-01 > change.patch

$ (email change.patch)

Now the original developer can apply that patch to their copy of the project and they have your changes. This is how many open source projects have been collaborated on for several years.

This actually works fairly well, so let's say we want to write a tool to make this basic process faster and easier. Instead of writing a tool that versions each file individually, like Subversion, we would probably write one that makes it easier to store snapshots of our project without having to copy the whole directory each time.

This is essentially what Git is. You tell Git you want to save a snapshot of your project with the git commitcommand and it basically records a manifest of what all of the files in your project look like at that point. Then most of the commands work with those manifests to see how they differ or pull content out of them, etc.



If you think about Git as a tool for storing and comparing and merging snapshots of your project, it may be easier to understand what is going on and how to do things properly.

## GETTING AND CREATING PROJECTS

In order to do anything in Git, you have to have a Git repository. This is where Git stores the data for the snapshots you are saving.

There are two main ways to get a Git repository. One way is to simply initialize a new one from an existing directory, such as a new project or a project new to source control. The second way is to clone one from a public Git repository, as you would do if you wanted a copy or wanted to work with someone on a project. We will cover both of these here.

## [docs](http://git-scm.com/docs/git-init)   [book](http://git-scm.com/book/en/Git-Basics-Getting-a-Git-Repository" \l "Initializing-a-Repository-in-an-Existing-Directory" \t "new)git init initializes a directory as a Git repository

To create a repository from an existing directory of files, you can simply run git init in that directory. For example, let's say we have a directory with a few files in it, like this:

**$ cd konnichiwa**

**$ ls**

README hello.rb

This is a project where we are writing examples of the "Hello World" program in every language. So far, we just have Ruby, but hey, it's a start. To start version controlling this with Git, we can simply run git init.

**$ git init**

Initialized empty Git repository in /opt/konnichiwa/.git/

Now you can see that there is a .git subdirectory in your project. This is your Git repository where all the data of your project snapshots are stored.

**$ ls -a**

. .. .git README hello.rb

Congratulations, you now have a skeleton Git repository and can start snapshotting your project.

**In a nutshell**, you use git init to make an existing directory of content into a new Git repository. You can do this in any directory at any time, completely locally.

## [docs](http://git-scm.com/docs/git-clone)   [book](http://git-scm.com/book/en/Git-Basics-Getting-a-Git-Repository" \l "Cloning-an-Existing-Repository" \t "new)git clone copy a git repository so you can add to it

If you need to collaborate with someone on a project, or if you want to get a copy of a project so you can look at or use the code, you will clone it. You simply run the git clone [url] command with the URL of the project you want to copy.

**$ git clone git://github.com/schacon/simplegit.git**

Initialized empty Git repository in /private/tmp/simplegit/.git/

remote: Counting objects: 100, done.

remote: Compressing objects: 100% (86/86), done.

remote: Total 100 (delta 35), reused 0 (delta 0)

Receiving objects: 100% (100/100), 9.51 KiB, done.

Resolving deltas: 100% (35/35), done.

**$ cd simplegit/**

**$ ls**

README Rakefile lib

This will copy the entire history of that project so you have it locally and it will give you a working directory of the main branch of that project so you can look at the code or start editing it. If you change into the new directory, you can see the .git subdirectory - that is where all the project data is.

**$ ls -a**

. .. .git README Rakefile lib

**$ cd .git**

**$ ls**

HEAD description info packed-refs

branches hooks logs refs

config index objects

By default, Git will create a directory that is the same name as the project in the URL you give it - basically whatever is after the last slash of the URL. If you want something different, you can just put it at the end of the command, after the URL.

**In a nutshell**, you use git clone to get a local copy of a Git repository so you can look at it or start modifying it.

## BASIC SNAPSHOTTING

Git is all about composing and saving snapshots of your project and then working with and comparing those snapshots. This section will explain the commands needed to compose and commit snapshots of your project.

An important concept here is that Git has an 'index', which acts as sort of a staging area for your snapshot. This allows you to build up a series of well composed snapshots from changed files in your working directory, rather than having to commit all of the file changes at once.

**In a nutshell**, you will use git add to start tracking new files and also to stage changes to already tracked files, then git status and git diff to see what has been modified and staged and finally git commit to record your snapshot into your history. This will be the basic workflow that you use most of the time.

## [docs](http://git-scm.com/docs/git-add)   [book](http://git-scm.com/book/en/Git-Basics-Recording-Changes-to-the-Repository" \l "Tracking-New-Files" \t "new)git add adds file contents to the staging area

In Git, you have to add file contents to your staging area before you can commit them. If the file is new, you can rungit add to initially add the file to your staging area, but even if the file is already "tracked" - ie, it was in your last commit - you still need to call git add to add new modifications to your staging area. Let's see a few examples of this.

Going back to our Hello World example, once we've initiated the project, we would now start adding our files to it and we would do that with git add. We can use git status to see what the state of our project is.

**$ git status -s**

?? README

?? hello.rb

So right now we have two untracked files. We can now add them.

**$ git add README hello.rb**

Now if we run git status again, we'll see that they've been added.

**$ git status -s**

A README

A hello.rb

It is also common to recursively add all files in a new project by specifying the current working directory like this: git add .. Since Git will recursively add all files under a directory you give it, if you give it the current working directory, it will simply start tracking every file there. In this case, a git add . would have done the same thing as a git add README hello.rb, or for that matter so would git add \*, but that's only because we don't have subdirectories which the \* would not recurse into.

OK, so now if we edit one of these files and run git status again, we will see something odd.

**$ vim README**

**$ git status -s**

AM README

A hello.rb

The 'AM' status means that the file has been modified on disk since we last added it. This means that if we commit our snapshot right now, we will be recording the version of the file when we last ran git add, not the version that is on our disk. Git does not assume that what the file looks like on disk is necessarily what you want to snapshot - you have to tell Git with the git add command.

**In a nutshell**, you run git add on a file when you want to include whatever changes you've made to it in your next commit snapshot. Anything you've changed that is not added will not be included - this means you can craft your snapshots with a bit more precision than most other SCM systems.

For a very interesting example of using this flexibility to stage only parts of modified files at a time, see the '-p' option to git add in the Pro Git book.

## [docs](http://git-scm.com/docs/git-status)   [book](http://git-scm.com/book/en/Git-Basics-Recording-Changes-to-the-Repository" \l "Checking-the-Status-of-Your-Files" \t "new)git status view the status of your files in the working directory and staging area

As you saw in the git add section, in order to see what the status of your staging area is compared to the code in your working directory, you can run the git status command. Using the -s option will give you short output. Without that flag, the git status command will give you more context and hints. Here is the same status output with and without the -s. The short output looks like this:

**$ git status -s**

AM README

A hello.rb

Where the same status with the long output looks like this:

**$ git status**

# On branch master

#

# Initial commit

#

# Changes to be committed:

# (use "git rm --cached <file>..." to unstage)

#

# new file: README

# new file: hello.rb

#

# Changed but not updated:

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

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

#

# modified: README

#

You can easily see how much more compact the short output is, but the long output has useful tips and hints as to what commands you may want to use next.

Git will also tell you about files that were deleted since your last commit or files that were modified or staged since your last commit.

**$ git status -s**

M README

D hello.rb

You can see there are two columns in the short status output. The first column is for the staging area, the second is for the working directory. So for example, if you have the README file staged and then you modify it again without running git add a second time, you'll see this:

**$ git status -s**

MM README

D hello.rb

**In a nutshell**, you run git status to see if anything has been modified and/or staged since your last commit so you can decide if you want to commit a new snapshot and what will be recorded in it.

## [docs](http://git-scm.com/docs/git-diff)   [book](http://git-scm.com/book/en/Git-Basics-Recording-Changes-to-the-Repository" \l "Viewing-Your-Staged-and-Unstaged-Changes" \t "new)git diff shows diff of what is staged and what is modified but unstaged

There are two main uses of the git diff command. One use we will describe here, the other we will describe later in the [**"Inspection and Comparison"**](http://gitref.org/inspect) section. The way we're going to use it here is to describe the changes that are staged or modified on disk but unstaged.

#### git diff show diff of unstaged changes

Without any extra arguments, a simple git diff will display in unified diff format (a patch) what code or content you've changed in your project since the last commit that are not yet staged for the next commit snapshot.

**$ vim hello.rb**

**$ git status -s**

M hello.rb

**$ git diff**

diff --git a/hello.rb b/hello.rb

index d62ac43..8d15d50 100644

--- a/hello.rb

+++ b/hello.rb

@@ -1,7 +1,7 @@

class HelloWorld

def self.hello

- puts "hello world"

+ puts "hola mundo"

end

end

So where git status will show you what files have changed and/or been staged since your last commit, git diff will show you what those changes actually are, line by line. It's generally a good follow-up command to git status

#### git diff --cached show diff of staged changes

The git diff --cached command will show you what contents have been staged. That is, this will show you the changes that will currently go into the next commit snapshot. So, if you were to stage the change to hello.rb in the example above, git diff by itself won't show you any output because it will only show you what is *not yet*staged.

**$ git status -s**

M hello.rb

**$ git add hello.rb**

**$ git status -s**

M hello.rb

**$ git diff**

**$**

If you want to see the staged changes, you can run git diff --cached instead.

**$ git status -s**

M hello.rb

**$ git diff**

**$**

**$ git diff --cached**

diff --git a/hello.rb b/hello.rb

index d62ac43..8d15d50 100644

--- a/hello.rb

+++ b/hello.rb

@@ -1,7 +1,7 @@

class HelloWorld

def self.hello

- puts "hello world"

+ puts "hola mundo"

end

end

#### git diff HEAD show diff of all staged or unstaged changes

If you want to see both staged and unstaged changes together, you can run git diff HEAD - this basically means you want to see the difference between your working directory and the last commit, ignoring the staging area. If we make another change to our hello.rb file then we'll have some changes staged and some changes unstaged. Here are what all three diff commands will show you:

**$ vim hello.rb**

**$ git diff**

diff --git a/hello.rb b/hello.rb

index 4f40006..2ae9ba4 100644

--- a/hello.rb

+++ b/hello.rb

@@ -1,7 +1,7 @@

class HelloWorld

+ # says hello

def self.hello

puts "hola mundo"

end

end

**$ git diff --cached**

diff --git a/hello.rb b/hello.rb

index 2aabb6e..4f40006 100644

--- a/hello.rb

+++ b/hello.rb

@@ -1,7 +1,7 @@

class HelloWorld

def self.hello

- puts "hello world"

+ puts "hola mundo"

end

end

**$ git diff HEAD**

diff --git a/hello.rb b/hello.rb

index 2aabb6e..2ae9ba4 100644

--- a/hello.rb

+++ b/hello.rb

@@ -1,7 +1,8 @@

class HelloWorld

+ # says hello

def self.hello

- puts "hello world"

+ puts "hola mundo"

end

end

#### git diff --stat show summary of changes instead of a full diff

If we don't want the full diff output, but we want more than the git status output, we can use the --stat option, which will give us a summary of changes instead. Here is the same example as above, but using the --stat option instead.

**$ git status -s**

MM hello.rb

**$ git diff --stat**

hello.rb | 1 +

1 files changed, 1 insertions(+), 0 deletions(-)

**$ git diff --cached --stat**

hello.rb | 2 +-

1 files changed, 1 insertions(+), 1 deletions(-)

**$ git diff HEAD --stat**

hello.rb | 3 ++-

1 files changed, 2 insertions(+), 1 deletions(-)

You can also provide a file path at the end of any of these options to limit the diff output to a specific file or subdirectory.

**In a nutshell**, you run git diff to see details of the git status command - *how* files have been modified or staged on a line by line basis.

## [docs](http://git-scm.com/docs/git-commit)   [book](http://git-scm.com/book/en/Git-Basics-Recording-Changes-to-the-Repository" \l "Committing-Your-Changes" \t "new)git commit records a snapshot of the staging area

Now that you have staged the content you want to snapshot with the git add command, you run git commit to actually record the snapshot. Git records your name and email address with every commit you make, so the first step is to tell Git what these are.

**$ git config --global user.name 'Your Name'**

**$ git config --global user.email you@somedomain.com**

Let's stage and commit all the changes to our hello.rb file. In this first example, we'll use the -m option to provide the commit message on the command line.

**$ git add hello.rb**

**$ git status -s**

M hello.rb

**$ git commit -m 'my hola mundo changes'**

[master 68aa034] my hola mundo changes

1 files changed, 2 insertions(+), 1 deletions(-)

Now we have recorded the snapshot. If we run git status again, we will see that we have a "clean working directory", which means that we have not made any changes since our last commit - there is no un-snapshotted work in our checkout.

**$ git status**

# On branch master

nothing to commit (working directory clean)

If you leave off the -m option, Git will try to open a text editor for you to write your commit message. In vim, which it will default to if it can find nothing else in your settings, the screen might look something like this:

# Please enter the commit message for your changes. Lines starting

# with '#' will be ignored, and an empty message aborts the commit.

# On branch master

# Changes to be committed:

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

#

# modified: hello.rb

#

~

~

".git/COMMIT\_EDITMSG" 9L, 257C

At this point you add your actual commit message at the top of the document. Any lines starting with '#' will be ignored - Git will put the output of the git status command in there for you as a reminder of what you have modified and staged.

In general, it's very important to write a good commit message. For open source projects, it's generally a rule to write your message more or less in this format:

Short (50 chars or less) summary of changes

More detailed explanatory text, if necessary. Wrap it to about 72

characters or so. In some contexts, the first line is treated as the

subject of an email and the rest of the text as the body. The blank

line separating the summary from the body is critical (unless you omit

the body entirely); some git tools can get confused if you run the

two together.

Further paragraphs come after blank lines.

- Bullet points are okay, too

- Typically a hyphen or asterisk is used for the bullet, preceded by a

single space, with blank lines in between, but conventions vary

here

# Please enter the commit message for your changes. Lines starting

# with '#' will be ignored, and an empty message aborts the commit.

# On branch master

# Changes to be committed:

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

#

# modified: hello.rb

#

~

~

~

".git/COMMIT\_EDITMSG" 25L, 884C written

The commit message is very important. Since much of the power of Git is this flexibility in carefully crafting commits locally and then sharing them later, it is very powerful to be able to write three or four commits of logically separate changes so that your work may be more easily peer reviewed. Since there is a separation between committing and pushing those changes, do take the time to make it easier for the people you are working with to see what you've done by putting each logically separate change in a separate commit with a nice commit message so it is easier for them to see what you are doing and why.

#### git commit -a automatically stage all tracked, modified files before the commit

If you think the git add stage of the workflow is too cumbersome, Git allows you to skip that part with the -aoption. This basically tells Git to run git add on any file that is "tracked" - that is, any file that was in your last commit and has been modified. This allows you to do a more Subversion style workflow if you want, simply editing files and then running git commit -a when you want to snapshot everything that has been changed. You still need to run git add to start tracking new files, though, just like Subversion.

**$ vim hello.rb**

**$ git status -s**

M hello.rb

**$ git commit -m 'changes to hello file'**

# On branch master

# Changed but not updated:

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

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

#

# modified: hello.rb

#

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

**$ git commit -am 'changes to hello file'**

[master 78b2670] changes to hello file

1 files changed, 2 insertions(+), 1 deletions(-)

Notice how if you don't stage any changes and then run git commit, Git will simply give you the output of the git status command, reminding you that nothing is staged. The important part of that message has been highlighted, saying that nothing is added to be committed. If you use -a, it will add and commit everything at once.

This now lets you complete the entire snapshotting workflow - you make changes to your files, then use git addto stage files you want to change, git status and git diff to see what you've changed, and then finally git commit to actually record the snapshot forever.

**In a nutshell**, you run git commit to record the snapshot of your staged content. This snapshot can then be compared, shared and reverted to if you need to.

## [docs](http://git-scm.com/docs/git-reset)   [book](http://git-scm.com/book/en/Git-Basics-Undoing-Things" \l "Unstaging-a-Staged-File" \t "new)git reset undo changes and commits

git reset is probably the most confusing command written by humans, but it can be very useful once you get the hang of it. There are three specific invocations of it that are generally helpful.

#### git reset HEAD undo the last commit and unstage the files

First, you can use it to unstage something that has been accidentally staged. Let's say that you have modified two files and want to record them into two different commits. You should stage and commit one, then stage and commit the other. If you accidentally stage both of them, how do you *un-*stage one? You do it with git reset HEAD -- file. Technically you don't have to add the -- - it is used to tell Git when you have stopped listing options and are now listing file paths, but it's probably good to get into the habit of using it to separate options from paths even if you don't need to.

Let's see what it looks like to unstage something. Here we have two files that have been modified since our last commit. We will stage both, then unstage one of them.

**$ git status -s**

M README

M hello.rb

**$ git add .**

**$ git status -s**

M README

M hello.rb

**$ git reset HEAD -- hello.rb**

Unstaged changes after reset:

M hello.rb

**$ git status -s**

M README

M hello.rb

Now you can run a git commit which will just record the changes to the README file, not the now unstagedhello.rb.

In case you're curious, what it's actually doing here is it is resetting the checksum of the entry for that file in the "index" to be what it was in the last commit. Since git add checksums a file and adds it to the "index", git reset HEAD overwrites that with what it was before, thereby effectively unstaging it.

If you want to be able to just run git unstage, you can easily setup an alias in Git. Just run git config --global alias.unstage "reset HEAD". Once you have run that, you can then just run git unstage [file]instead.

If you forget the command to unstage something, Git is helpful in reminding you in the output of the normal git status command. For example, if you run git status without the -s when you have staged files, it will tell you how to unstage them:

**$ git status**

# On branch master

# Changes to be committed:

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

#

# modified: README

# modified: hello.rb

#

When you run git reset without specifying a flag it defaults to --mixed. The other options are --soft and --hard.

#### git reset --soft undo the last commit

The first thing git reset does is undo the last commit and put the files back onto the stage. If you include the --soft flag this is where it stops. For example, if you run git reset --soft HEAD~ (the parent of the HEAD) the last commit will be undone and the files touched will be back on the stage again.

**$ git status -s**

M hello.rb

**$ git commit -am 'hello with a flower'**

[master 5857ac1] hello with a flower

1 files changed, 3 insertions(+), 1 deletions(-)

**$ git status**

# On branch master

nothing to commit (working directory clean)

**$ git reset --soft HEAD~**

**$ git status -s**

M hello.rb

This is basically doing the same thing as git commit --amend, allowing you to do more work before you roll in the file changes into the same commit.

#### git reset --hard undo the last commit, unstage files AND undo any changes in the working directory

The third option is to go --hard and make your working directory look like the index, unstage files and undo the last commit. This is the most dangerous option and not working directory safe. Any changes not in the index or have not been commited will be lost.

**$ git status**

# On branch master

# Changes to be committed:

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

#

# modified: README

#

# 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: README

#

**$ git reset --hard HEAD**

HEAD is now at 5857ac1 hello with a flower

**$ git status**

# On branch master

nothing to commit (working directory clean)

In the above example, while we had both changes ready to commit and ready to stage, a git reset --hardwiped them out. On top of that, the last commit has been undone.

You can replace HEAD with a commit SHA-1 or another parent reference to reset to that specific point.

**In a nutshell**, you run git reset HEAD to undo the last commit, unstage files that you previously ran git add on and wish to not include in the next commit snapshot

## [docs](http://git-scm.com/docs/git-rm)   [book](http://git-scm.com/book/en/Git-Basics-Recording-Changes-to-the-Repository" \l "Removing-Files)git rm remove files from the staging area

git rm will remove entries from the staging area. This is a bit different from git reset HEAD which "unstages" files. To "unstage" means it reverts the staging area to what was there before we started modifying things. git rmon the other hand just kicks the file off the stage entirely, so that it's not included in the next commit snapshot, thereby effectively deleting it.

By default, a git rm file will remove the file from the staging area entirely and also off your disk (the working directory). To leave the file in the working directory, you can use git rm --cached .

#### git mv git rm --cached orig; mv orig new; git add new

Unlike most other version control systems, Git does not track file renames. Instead, it just tracks the snapshots and then figures out what files were likely renamed by comparing snapshots. If a file was removed from one snapshot and another file was added to the next one and the contents are similar, Git figures it was most likely a rename. So, although the git mv command exists, it is superfluous - all it does is a git rm --cached, moves the file on disk, then runs a git add on the new file. You don't really need to use it, but if it's easier, feel free.

In its normal form the command is used to delete files. But it's often easier to just remove the files off your disk and then run git commit -a, which will also automatically remove them from your index.

**In a nutshell**, you run git rm to remove files from being tracked in Git. It will also remove them from your working directory.

## [docs](http://git-scm.com/docs/git-stash)   [book](http://git-scm.com/book/en/Git-Tools-Stashing)git stash save changes made in the current index and working directory for later

You're in the middle of some changes but something comes up that you need to jump over to, like a so-urgent-right-now bugfix, but don't want to commit or lose your current edits. git stash is there for you.

#### git stash add current changes to the stack

Stashing takes the current state of the working directory and index, puts it on a stack for later, and gives you back a clean working directory. It will then leave you at the state of the last commit.

**$ git status -s**

M hello.rb

**$ git stash**

Saved working directory and index state WIP on master: 5857ac1 hello with a flower

HEAD is now at 5857ac1 hello with a flower

**$ git status**

# On branch master

nothing to commit (working directory clean)

#### git stash list view stashes currently on the stack

It's helpful to know what you've got stowed on the stash and this is where git stash list comes in. Running this command will display a queue of current stash items.

**$ git stash list**

stash@{0}: WIP on master: 5857ac1 hello with a flower

The last item added onto the stash will be referenced by stash@{0} and increment those already there by one.

**$ vim hello.rb**

**$ git commit -am 'it stops raining'**

[master ee2d2c6] it stops raining

1 files changed, 1 insertions(+), 1 deletions(-)

**$ vim hello.rb**

**$ git stash**

Saved working directory and index state WIP on master: ee2d2c6 it stops raining

HEAD is now at ee2d2c6 it stops raining

**$ git stash list**

stash@{0}: WIP on master: ee2d2c6 it stops raining

stash@{1}: WIP on master: 5857ac1 hello with a flower

#### git stash apply grab the item from the stash list and apply to current working directory

When you're ready to continue from where you left off, run the git stash apply command to bring back the saved changes onto the working directory.

**$ 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: hello.rb

#

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

By default it will reapply the last added stash item to the working directory. This will be the item referenced bystash@{0}. You can grab another stash item instead if you reference it in the arguments list. For example, git stash apply stash@{1} will apply the item referenced by stash@{1}.

If you also want to remove the item from the stack at the same time, use git stash pop instead.

#### git stash drop remove an item from the stash list

When you're done with the stashed item and/or want to remove it from the list, run the git stash drop command. By default this will remove the last added stash item. You can also remove a specific item if you include it as an argument.

In this example, our stash list has at least two items, but we want to get rid of the item added before last, which is referenced by stash@{1}.

**$ git stash drop stash@{1}**

Dropped stash@{1} (0b1478540189f30fef9804684673907c65865d8f)

If you want to remove of all the stored items, just run the git stash clear command. But only do this if you're sure you're done with the stash.

**In a nutshell**, run git stash to quickly save some changes that you're not ready to commit or save, but want to come back to while you work on something else.