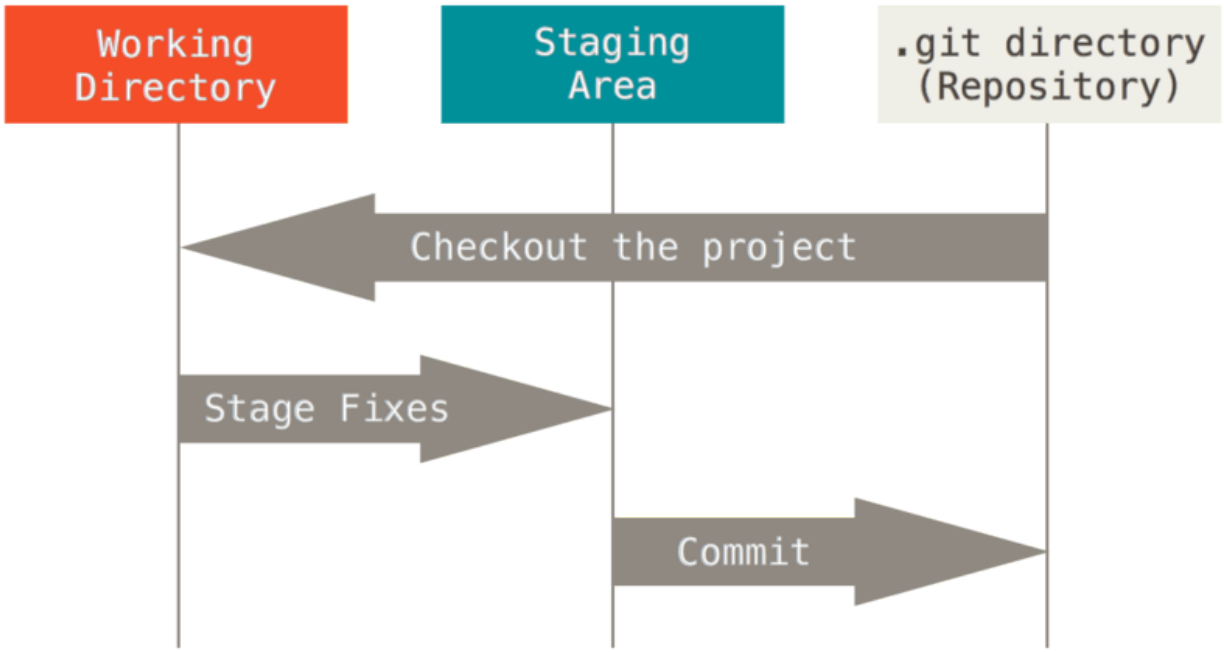
Git

一、Git概述

1.1 Git三种状态

- 已提交（committed）：Modified means that you have changed the file but have not committed it to your database yet.
- 已修改（modified）：Staged means that you have marked a modified file in its current version to go into your next commit snapshot.
- 已暂存（staged）：Committed means that the data is safely stored in your local database.

1.2 Git三个阶段



- 工作区（Working Tree）：The working tree is a single checkout of one version of the project. These files are pulled out of the compressed database in the Git directory and placed on disk for you to use or modify.
- 暂存区（Staging Area / Index）：The staging area is a file, generally contained in your Git directory, that stores information about what will go into your next commit. Its technical name in Git parlance is the “index”, but the phrase “staging area” works just as well.
- Git目录（Git Directory）：The Git directory is where Git stores the metadata and object database for your project. This is the most important part of Git, and it is what is copied when you *clone* a repository from another computer.

1.3 Workflow Of Git

The basic Git workflow goes something like this:

- You modify files in your working tree.
- You selectively stage just those changes you want to be part of your next commit, which adds *only* those changes to the staging area.
- You do a commit, which takes the files as they are in the staging area and stores that snapshot permanently to your Git directory.

1.4 用户信息

The first thing you should do when you install Git is to set your user name and email address. This is important because every Git commit uses this information, and it’s immutably baked into the commits you start creating:

```
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com
```

NOTICE:**You need to do this only once if you pass the `--global` option, because then Git will always use that information for anything you do on that system. If you want to override this with a different name or email address for specific projects, you can run the command without the `--global` option when you’re in that project.**

1.5 Checking Your Settings

If you want to check your configuration settings, you can use the `git config --list` command to list all the settings Git can find at that point:

```
$ git config --list
user.name=John Doe
user.email=johndoe@example.com
color.status=auto
color.branch=auto
color.interactive=auto
color.diff=auto
...
```

二、 Git Bacis

2.1 Initializing a Repository in an Existing Directory

If you have a project directory that is currently not under version control and you want to start controlling it with Git, you first need to go to that project’ s directory. Then type:

```
$ git init
```

If you want to start version-controlling existing files (as opposed to an empty directory), you should probably begin tracking those files and do an initial commit. You can accomplish that with a few `git add` commands that specify the files you want to track, followed by a `git commit` :

```
$ git add *.c
$ git add LICENSE
$ git commit -m 'Initial project version'
```

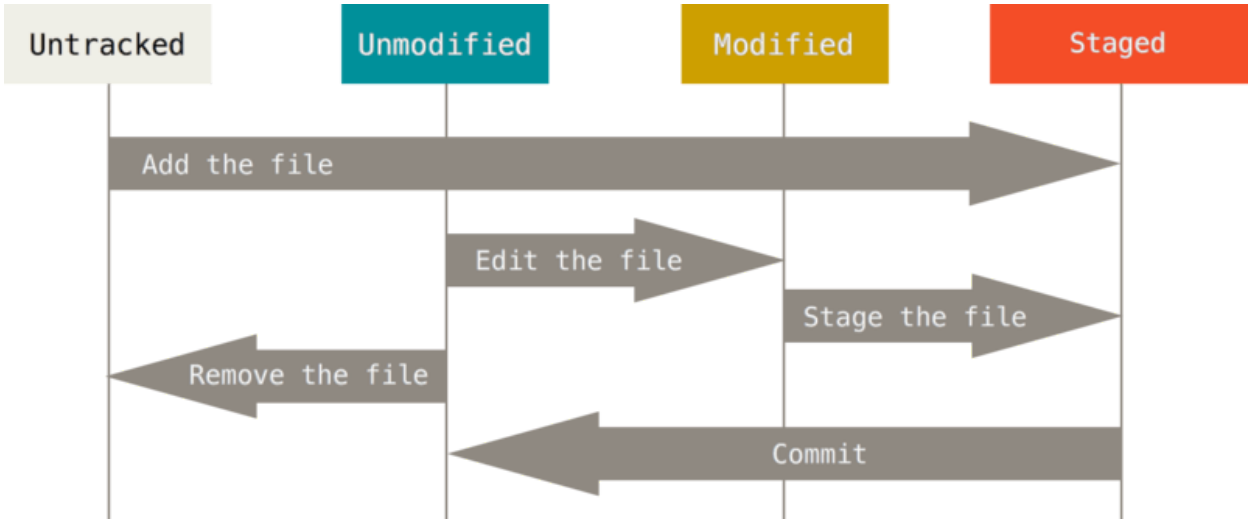
2.2 Recording Changes to the Repository

Checking the Status of Your Files

The main tool you use to determine which files are in which state is the `git status` command.

```
$ git status
```

The lifecycle of the status of your files



2.3 Ignoring Files

The rules for the patterns you can put in the `.gitignore` file are as follows:

- Blank lines or lines starting with `#` are ignored.
- Standard glob patterns work, and will be applied recursively throughout the entire working tree.
- You can start patterns with a forward slash (`/`) to avoid recursivity.
- You can end patterns with a forward slash (`/`) to specify a directory.
- You can negate a pattern by starting it with an exclamation point (`!`).

Glob patterns are like simplified regular expressions that shells use. An asterisk (`*`) matches zero or more characters; `[abc]` matches any character inside the brackets (in this case a, b, or c); a question mark (`?`) matches a single character; and brackets enclosing characters separated by a hyphen (`[0-9]`) matches any character between them (in this case 0 through 9). You can also use two asterisks to match nested directories; `a/**/z` would match `a/z` , `a/b/z` , `a/b/c/z` , and so on.

Here is another example `.gitignore` file:

```
# ignore all .a files
*.a

# but do track lib.a, even though you're ignoring .a files above
```

```
!lib.a

# only ignore the TODO file in the current directory, not
subdir/TODO
/TODO

# ignore all files in any directory named build
build/

# ignore doc/notes.txt, but not doc/server/arch.txt
doc/*.txt

# ignore all .pdf files in the doc/ directory and any of its
subdirectories
doc/**/*.pdf
```

2.4 Removing Files

To remove a file from Git, you have to remove it from your tracked files (more accurately, remove it from your staging area) and then commit. The `git rm` command does that, and also removes the file from your working directory so you don't see it as an untracked file the next time around.

The primary function of `git rm` is to remove tracked files from the Git index. Additionally, `git rm` can be used to remove files from both the staging index and the working directory.

Another useful thing you may want to do is to keep the file in your working tree but remove it from your staging area. In other words, you may want to keep the file on your hard drive but not have Git track it anymore. This is particularly useful if you forgot to add something to your `.gitignore` file and accidentally staged it, like a large log file or a bunch of `.a` compiled files. To do this, use the `--cached` option:

```
git rm --cached README
```

2.5 Viewing Your Staged and Unstaged Changes

```
git diff
```

Show changes between the working tree and the index or a tree, changes between the index and a tree, changes between two trees, changes resulting from a merge, changes between two blob objects, or changes between two files on disk.

The following examples will be executed in a simple repo. The repo is created with the commands below:

```
$:> mkdir diff_test_repo
$:> cd diff_test_repo
$:> touch diff_test.txt
$:> echo "this is a git diff test example" > diff_test.txt
$:> git init .
Initialized empty Git repository in /Users/kev/code/test/.git/
$:> git add diff_test.txt
$:> git commit -am"add diff test file"
[main (root-commit) 6f77fc3] add diff test file
1 file changed, 1 insertion(+)
create mode 100644 diff_test.txt
```

we execute `git diff` at this point, there will be no output. This is expected behavior as there are no changes in the repo to diff. Once the repo is created and we've added the `diff_test.txt` file, we can change the contents of the file to start experimenting with diff output.

```
$:> echo "this is a diff example" > diff_test.txt
```

Executing this command will change the content of the `diff_test.txt` file. Once modified, we can view a diff and analyze the output. Now executing `git diff` will produce the following output:

```
diff --git a/diff_test.txt b/diff_test.txt
index 6b0c6cf..b37e70a 100644
--- a/diff_test.txt
+++ b/diff_test.txt
@@ -1 +1 @@
-this is a git diff test example
+this is a diff example
```

```
@@ -1 +1 @@
-this is a git diff test example
+this is a diff example
```

The first line is the chunk header. Each chunk is prepended by a header inclosed within `@@` symbols. The content of the header is a summary of changes made to the file. In our simplified example, we have -1 +1 meaning line one had changes. In a more realistic diff, you would see a header like:

```
@@ -34,6 +34,8 @@
```

In this header example, 6 lines have been extracted starting from line number 34. Additionally, 8 lines have been added starting at line number 34.

2.6 Viewing the Commit History

After you have created several commits, or if you have cloned a repository with an existing commit history, you'll probably want to look back to see what has happened. The most basic and powerful tool to do this is the `git log` command.

When you run `git log` in this project, you should get output that looks something like this:

```
$ git log
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    Change version number

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
```

```
Date:   Sat Mar 15 16:40:33 2008 -0700

    Remove unnecessary test

commit a11bef06a3f659402fe7563abf99ad00de2209e6
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 10:31:28 2008 -0700

    Initial commit
```

By default, with no arguments, `git log` lists the commits made in that repository in reverse chronological order; that is, the most recent commits show up first. As you can see, this command lists each commit with its SHA-1 checksum, the author's name and email, the date written, and the commit message.

Common options to `git log`

Option	Description
`-p`	Show the patch introduced with each commit.
`--stat`	Show statistics for files modified in each commit.
`--shortstat`	Display only the changed/insertions/deletions line from the --stat command.
`--name-only`	Show the list of files modified after the commit information.
`--name-status`	Show the list of files affected with added/modified/deleted information as well.
`--abbrev-commit`	Show only the first few characters of the SHA-1 checksum instead of all 40.
`--relative-date`	Display the date in a relative format (for example, "2 weeks ago") instead of using the full date format.

Option	Description
`--graph`	Display an ASCII graph of the branch and merge history beside the log output.
`--pretty`	Show commits in an alternate format. Option values include oneline, short, full, fuller, and format (where you specify your own format).
`--oneline`	Shorthand for `--pretty=oneline --abbrev-commit` used together.

Option `--pretty`

This option changes the log output to formats other than the default. A few prebuilt option values are available for you to use.

The most interesting option value is `format`, which allows you to specify your own log output format. This is especially useful when you're generating output for machine parsing — because you specify the format explicitly, you know it won't change with updates to Git:

```
$ git log --pretty=format:"%h - %an, %ar : %s"
ca82a6d - Scott Chacon, 6 years ago : Change version number
085bb3b - Scott Chacon, 6 years ago : Remove unnecessary test
a11bef0 - Scott Chacon, 6 years ago : Initial commit
```

Useful specifiers for `git log --pretty=format` lists some of the more useful specifiers that `format` takes.

Specifier	Description of Output
`%H`	Commit hash
`%h`	Abbreviated commit hash

Specifier	Description of Output
`%T`	Tree hash
`%t`	Abbreviated tree hash
`%P`	Parent hashes
`%p`	Abbreviated parent hashes
`%an`	Author name
`%ae`	Author email
`%ad`	Author date (format respects the --date=option)
`%ar`	Author date, relative
`%cn`	Committer name
`%ce`	Committer email
`%cd`	Committer date
`%cr`	Committer date, relative
`%s`	Subject

The `format` option values are particularly useful with another `log` option called `--graph` . This option adds a nice little ASCII graph showing your branch and merge history:

```
$ git log --pretty=format:"%h %s" --graph
* 2d3acf9 Ignore errors from SIGCHLD on trap
*  5e3ee11 Merge branch 'master' of git://github.com/dustin/grit
|\
| * 420eac9 Add method for getting the current branch
* | 30e367c Timeout code and tests
* | 5a09431 Add timeout protection to grit
* | e1193f8 Support for heads with slashes in them
|/
* d6016bc Require time for xmlschema
*  11d191e Merge branch 'defunkt' into local
```

Limiting Log Output

In addition to output-formatting options, `git log` takes a number of useful limiting options; that is, options that let you show only a subset of commits.

In fact, you can do `-<n>`, where `n` is any integer to show the last `n` commits, such as

```
$ git log -2
commit f71a4a8c84482e41c64c7992f629f0937234cdf9 (HEAD -> master)
Author: Ecifics <ecifics@gmail.com>
Date:   Thu Dec 1 16:22:02 2022 +0800

    add diff_test.txt

commit a1ab6fd41350d9356a90d6b910ef9026b395e57e
Author: Ecifics <ecifics@gmail.com>
Date:   Thu Dec 1 16:20:30 2022 +0800

    delete RM.md
```

`git log -2` shows the last 2 commits

2.7 Undoing Things

```
$ git commit --amend
```

This command takes your staging area and uses it for the commit. If you've made no changes since your last commit (for instance, you run this command immediately after your previous commit), then your snapshot will look exactly the same, and all you'll change is your commit message.

As an example, if you commit and then realize you forgot to stage the changes in a file you wanted to add to this commit, you can do something like this:

```
$ git commit -m 'Initial commit'
$ git add forgotten_file
$ git commit --amend
```

You end up with a single commit — the second commit replaces the results of the first.

Unstaging a Staged File

For example, let's say you've changed two files and want to commit them as two separate changes, but you accidentally type `git add *` and stage them both. How can you unstage one of the two? The `git status` command reminds you:

```
$ git add *
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    renamed:    README.md -> README
    modified:   CONTRIBUTING.md
```

Right below the "Changes to be committed" text, it says use `git reset HEAD <file>...` to unstage. So, let's use that advice to unstage the `CONTRIBUTING.md` file:

```
$ git reset HEAD CONTRIBUTING.md
Unstaged changes after reset:
M   CONTRIBUTING.md
$ git status
On branch master
```

```
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    renamed:    README.md -> 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:   CONTRIBUTING.md
```

The command is a bit strange, but it works. The `CONTRIBUTING.md` file is modified but once again unstaged.

Unmodifying a Modified File

What if you realize that you don't want to keep your changes to the `CONTRIBUTING.md` file? How can you easily unmodify it — revert it back to what it looked like when you last committed (or initially cloned, or however you got it into your working directory)? Luckily, `git status` tells you how to do that, too. In the last example output, the unstaged area looks like this:

```
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:   CONTRIBUTING.md
```

It tells you pretty explicitly how to discard the changes you've made. Let's do what it says:

```
$ git checkout -- CONTRIBUTING.md
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    renamed:    README.md -> README
```

You can see that the changes have been reverted.

2.8 Working with Remotes

Showing Your Remotes

To see which remote servers you have configured, you can run the `git remote` command. It lists the shortnames of each remote handle you've specified.

```
$ git remote
origin
```

You can also specify `-v`, which shows you the URLs that Git has stored for the shortname to be used when reading and writing to that remote:

```
$ git remote -v
origin  https://github.com/schacon/ticgit (fetch)
origin  https://github.com/schacon/ticgit (push)
```

Adding Remote Repositories

To add a new remote Git repository as a shortname you can reference easily, run `git remote add <shortname> <url>`:

```
$ git remote
origin
$ git remote add pb https://github.com/paulboone/ticgit
$ git remote -v
origin  https://github.com/schacon/ticgit (fetch)
origin  https://github.com/schacon/ticgit (push)
pb  https://github.com/paulboone/ticgit (fetch)
pb  https://github.com/paulboone/ticgit (push)
```

Fetching and Pulling from Your Remotes

```
$ git fetch <remote>
```

The command goes out to that remote project and pulls down all the data from that remote project that you don’t have yet. After you do this, you should have references to all the branches from that remote, which you can merge in or inspect at any time.

It’s important to note that the `git fetch` command only downloads the data to your local repository — it doesn’t automatically merge it with any of your work or modify what you’re currently working on. You have to merge it manually into your work when you’re ready.

Running `git pull` generally fetches data from the server you originally cloned from and automatically tries to merge it into the code you’re currently working on.

Pushing to Your Remotes

The command for this is simple: `git push <remote> <branch>`. If you want to push your `master` branch to your `origin` server, then you can run this to push any commits you’ve done back up to the server:

```
$ git push origin master
```

This command works only if you cloned from a server to which you have write access and if nobody has pushed in the meantime. If you and someone else clone at the same time and they push upstream and then you push upstream, your push will rightly be rejected. You'll have to fetch their work first and incorporate it into yours before you'll be allowed to push.

Inspecting a Remote

If you want to see more information about a particular remote, you can use the `git remote show <remote>` command. If you run this command with a particular shortname, such as `origin`, you get something like this:

```
$ git remote show origin
* remote origin
  Fetch URL: https://github.com/schacon/ticgit
  Push  URL: https://github.com/schacon/ticgit
  HEAD branch: master
  Remote branches:
    master                               tracked
    dev-branch                           tracked
  Local branch configured for 'git pull':
    master merges with remote master
  Local ref configured for 'git push':
    master pushes to master (up to date)
```

It lists the URL for the remote repository as well as the tracking branch information. The command helpfully tells you that if you're on the `master` branch and you run `git pull`, it will automatically merge the remote's `master` branch into the local one after it has been fetched. It also lists all the remote references it has pulled down.

Renaming and Removing Remotes

You can run `git remote rename` to change a remote's shortname. For instance, if you want to rename `pb` to `paul`, you can do so with `git remote rename`:

```
$ git remote rename pb paul
$ git remote
origin
paul
```

It's worth mentioning that this changes all your remote-tracking branch names, too. What used to be referenced at `pb/master` is now at `paul/master`.

If you want to remove a remote for some reason — you've moved the server or are no longer using a particular mirror, or perhaps a contributor isn't contributing anymore — you can either use `git remote remove` or `git remote rm`:

```
$ git remote remove paul
$ git remote
origin
```

Once you delete the reference to a remote this way, all remote-tracking branches and configuration settings associated with that remote are also deleted.

2.9 Tagging

Typically, people use this functionality to mark release points (`v1.0`, `v2.0` and so on).

Listing Your Tags

Listing the existing tags in Git is straightforward. Just type `git tag` (with optional `-l` or `--list`):

```
$ git tag
v1.0
v2.0
```

This command lists the tags in alphabetical order; the order in which they are displayed has no real importance.