
Introduction

Checking large binary files into a source repository (Git or otherwise) is a bad idea because repository size quickly becomes unreasonable. Even if the instantaneous working tree stays manageable, preserving repository integrity requires all binary files in the entire project history, which given the typically poor compression of binary diffs, implies that the repository size will become impractically large. Some people recommend checking binaries into different repositories or even not versioning them at all, but these are not satisfying solutions for most workflows.

Features of `git-fat`

- clones of the source repository are small and fast because no binaries are transferred, yet fully functional with complete metadata and incremental retrieval (`git clone --depth` has limited granularity and couples metadata to content)
- `git-fat` supports the same workflow for large binaries and traditionally versioned files, but internally manages the “fat” files separately
- `git-bisect` works properly even when versions of the binary files change over time
- selective control of which large files to pull into the local store
- local fat object stores can be shared between multiple clones, even by different users
- can easily support fat object stores distributed across multiple hosts
- depends only on stock Python and rsync

Related projects

- `git-annex` is a far more comprehensive solution, but with less transparent workflow and with more dependencies.
- `git-media` adopts a similar approach to `git-fat`, but with a different synchronization philosophy and with many Ruby dependencies.

Installation and configuration

Place `git-fat` in your `PATH`.

Edit (or create) `.gitattributes` to regard any desired extensions as fat files.

```
1 $ cd path-to-your-repository
2 $ cat >> .gitattributes
3 *.png filter=fat -crlf
```

```
4 *.jpg filter=fat -crlf
5 *.gz  filter=fat -crlf
6 ^D
```

Run `git fat init` to activate the extension. Now add and commit as usual. Matched files will be transparently stored externally, but will appear complete in the working tree.

Set a remote store for the fat objects by editing `.gitfat`.

```
1 [rsync]
2 remote = your.remote-host.org:/share/fat-store
```

This file should typically be committed to the repository so that others will automatically have their remote set. This remote address can use any protocol supported by rsync.

Most users will configure it to use remote ssh in a directory with shared access. To do this, set the `sshuser` and `sshport` variables in `.gitfat` configuration file. For example, to use rsync with ssh, with the default port (22) and authenticate with the user “*fat*”, your configuration would look like this:

```
1 [rsync]
2 remote = your.remote-host.org:/share/fat-store
3 sshuser = fat
```

A worked example

Before we start, let’s turn on verbose reporting so we can see what’s happening. Without this environment variable, all the output lines starting with `git-fat` will not be shown.

```
1 $ export GIT_FAT_VERBOSE=1
```

First, we create a repository and configure it for use with `git-fat`.

```
1 $ git init repo
2 Initialized empty Git repository in /tmp/repo/.git/
3 $ cd repo
4 $ git fat init
5 $ cat > .gitfat
6 [rsync]
7 remote = localhost:/tmp/fat-store
8 $ mkdir -p /tmp/fat-store          # make sure the remote
   directory exists
9 $ echo '*.gz filter=fat -crlf' > .gitattributes
10 $ git add .gitfat .gitattributes
11 $ git commit -m'Initial repository'
12 [master (root-commit) eb7facb] Initial repository
```

```
13 2 files changed, 3 insertions(+)
14 create mode 100644 .gitattributes
15 create mode 100644 .gitfat
```

Now we add a binary file whose name matches the pattern we set in `.gitattributes`.

```
1 $ curl https://nodeload.github.com/jedbrown/git-fat/tar.gz/master -o
  master.tar.gz
2   % Total    % Received % Xferd  Average Speed   Time    Time       Time
3   Current                      Dload  Upload  Total  Spent  Left
4 100  6449  100  6449    0     0   7741      0 --:--:-- --:--:--
  --:--:--  9786
5 $ git add master.tar.gz
6 git-fat filter-clean: caching to /tmp/repo/.git/fat/objects/
  b3489819f81603b4c04e8ed134b80bace0810324
7 $ git commit -m'Added master.tar.gz'
8 [master b85a96f] Added master.tar.gz
9 git-fat filter-clean: caching to /tmp/repo/.git/fat/objects/
  b3489819f81603b4c04e8ed134b80bace0810324
10 1 file changed, 1 insertion(+)
11 create mode 100644 master.tar.gz
```

The patch itself is very simple and does not include the binary.

```
1 $ git show --pretty=oneline HEAD
2 918063043a6156172c2ad66478c6edd5c7df0217 Add master.tar.gz
3 diff --git a/master.tar.gz b/master.tar.gz
4 new file mode 100644
5 index 00000000..12f7d52
6 --- /dev/null
7 +++ b/master.tar.gz
8 @@ -0,0 +1 @@
9 +##$ git-fat 1f218834a137f7b185b498924e7a030008aee2ae
```

Pushing fat files

Now let's push our fat files using the rsync configuration that we set up earlier.

```
1 $ git fat push
2 Pushing to localhost:/tmp/fat-store
3 building file list ...
4 1 file to consider
5
6 sent 61 bytes  received 12 bytes  48.67 bytes/sec
7 total size is 6449  speedup is 88.34
```

We might normally set a remote now and push the git repository.

Cloning and pulling

Now let's look at what happens when we clone.

```
1 $ cd ..
2 $ git clone repo repo2
3 Cloning into 'repo2'...
4 done.
5 $ cd repo2
6 $ git fat init                                # don't forget
7 $ ls -l                                       # file is just a placeholder
8 total 4
9 -rw-r--r-- 1 jed users 53 Nov 25 22:42 master.tar.gz
10 $ cat master.tar.gz                          # holds the SHA1 of the file
11 ## git-fat 1f218834a137f7b185b498924e7a030008aee2ae
```

We can always get a summary of what fat objects are missing in our local cache.

```
1 Orphan objects:
2 1f218834a137f7b185b498924e7a030008aee2ae
```

Now get any objects referenced by our current `HEAD`. This command also accepts the `--all` option to pull full history, or a revision to pull selected history.

```
1 $ git fat pull
2 receiving file list ...
3 1 file to consider
4 1f218834a137f7b185b498924e7a030008aee2ae
5      6449 100%      6.15MB/s      0:00:00 (xfer#1, to-check=0/1)
6
7 sent 30 bytes received 6558 bytes 4392.00 bytes/sec
8 total size is 6449 speedup is 0.98
9 Restoring 1f218834a137f7b185b498924e7a030008aee2ae -> master.tar.gz
10 git-fat filter-smudge: restoring from /tmp/repo2/.git/fat/objects/1
    f218834a137f7b185b498924e7a030008aee2ae
```

Everything is in place

```
1 $ git status
2 git-fat filter-clean: caching to /tmp/repo2/.git/fat/objects/1
    f218834a137f7b185b498924e7a030008aee2ae
3 # On branch master
4 nothing to commit, working directory clean
5 $ ls -l                                # recovered the full file
6 total 8
7 -rw-r--r-- 1 jed users 6449 Nov 25 17:10 master.tar.gz
```

Summary

- Set the “fat” file types in `.gitattributes`.
- Use normal git commands to interact with the repository without thinking about what files are fat and non-fat. The fat files will be treated specially.
- Synchronize fat files with `git fat push` and `git fat pull`.

Retroactive import using `git filter-branch` [Experimental]

Sometimes large objects were added to a repository by accident or for lack of a better place to put them. *If* you are willing to rewrite history, forcing everyone to reclone, you can retroactively manage those files with `git fat`. Be sure that you understand the consequences of `git filter-branch` before attempting this. This feature is experimental and irreversible, so be doubly careful with backups.

Step 1: Locate the fat files

Run `git fat find THRESH_BYTES > fat-files` and inspect `fat-files` in an editor. Lines will be sorted by the maximum object size that has been at each path, and look like

```
1 something.big          filter=fat -text #      8154677 1
```

where the first number after the # is the number of bytes and the second number is the number of modifications that path has seen. You will normally filter out some of these paths using `grep` and/or an editor. When satisfied, remove the ends of the lines (including the #) and append to `.gitattributes`. It's best to `git add .gitattributes` and commit at this time (likely enrolling some extant files into `git fat`).

Step 2: `filter-branch`

Copy `.gitattributes` to `/tmp/fat-filter-files` and edit to remove everything after the file name (e.g., `sed s/ \+filter=fat.*$/`). Currently, this may only contain exact paths relative to the root of the repository. Finally, run

```
1 git filter-branch --index-filter \
2   'git fat index-filter /tmp/fat-filter-files --manage-gitattributes' \
3   --tag-name-filter cat -- --all
```

(You can remove the `--manage-gitattributes` option if you don't want to append all the files being enrolled in `git fat` to `.gitattributes`, however, future users would need to use `.git/info/attributes` to have the `git fat` filters run.) When this finishes, inspect to see if everything is in order and follow the Checklist for Shrinking a Repository in the `git filter-branch` man page, typically `git clone file:///path/to/repo`. Be sure to `git fat push` from the original repository.

See the script `test-retroactive.sh` for an example of cleaning.

Implementation notes

The actual binary files are stored in `.git/fat/objects`, leaving `.git/objects` nice and small.

```
1 $ du -bs .git/objects
2 2212      .git/objects/
3 $ ls -l .git/fat/objects          # This is where the file
    actually goes, but that's not important
4 total 8
5 -rw----- 1 jed users 6449 Nov 25 17:01 1
    f218834a137f7b185b498924e7a030008aee2ae
```

If you have multiple clones that access the same filesystem, you can make `.git/fat/objects` a symlink to a common location, in which case all content will be available in all repositories without extra copies. You still need to `git fat push` to make it available to others.

Some refinements

- Allow pulling and pushing only select files
- Relate orphan objects to file system
- Put some more useful message in smudged (working tree) version of missing files.
- More friendly configuration for multiple fat remotes
- Make commands safer in presence of a dirty tree.
- Private setting of a different remote.
- Gracefully handle unmanaged files when the filter is called (either legacy files or files matching the pattern that should some reason not be treated as fat).