Case Study: Git

Feature Location: Worktrees

1.1.

Question

Find the data structure representing a worktree

Answer

worktree@worktree.h:8

1.2.

Question

Find the function that gets the current worktree

Answer

get_main_worktree@worktree.c:47

1.3.

Question

How does git decide if a repository is bare?

Answer

It checks that the worktree's common dir path ends in ".git" (see worktree.c:55).

Feature Location: Submodules

2.1.

Question

Find the data structure that represents a git submodule

Answer

”submodule” in submodule-config.h

2.3.

Question

A function that creates a submodule from a name or path

Answer

`config_from`

Alternate answers:

`submodule_from_name` , `submodule_from_path`

2.4.

Question

The function that actually initializes the submodule structure

Answer

lookup_or_create_by_name@submodule-config.c:241 git_parse_source (or one of its callers), config.c:695

2.5.

Question

One main (and many helper) lower-level functions used for parsing git config files

Answer

git_parse_source (or one of its callers), config.c:695

2.6.

Question

A callback (function passed to another function) that processes the keys in the submodules config file

Answer

parse_config@submodule-config.c:394 or gitmodules_cb@submodule-config.c:612

2.7.

Question

Find two things that change in the parent repository upon adding a submodule

Answer

Possible answers:

- The git SUBMODULES file
- A new revision is made in the parent repository with a commit adding the submodule
- A new tree is added to the parent repository pointing at the submodule

Freestyle

These answers come from Azhar Desai, and were deemed especially high quality (though note he spent >30 minutes on average each). Notice the use of looking for relevant data structures first.

Is it possible to efficiently find the commits that is a direct child of/that points to a given commit?

- `struct commit` includes a list of parents it's points to, but no children
- what's a commit-graph?
- looks like it just builds for a given commit, parents, dates, tree, so not of direct interest (commit-graph.h)
- there is `is_descendant_of` (commit.{h,c}) which checks if a given struct commit is a descendant of a commit list, but requires specifying a known commit list ahead of time
- used during a git pull to decided whether a fast-forward merge is possible
- `struct revinfo` (revisions.{c,h}) looks interesting because there are functions like `set_childrен` and `add_child`

- these ultimately these add to the parent commit the children as a decoration
- but judging by `struct revinfo` this starts from a list of `struct commit_list *commits` on it
- My guess is no: unless you're providing a set of starting commits/revisions, it's not efficiently possible

How does a git repositry discover which "objects" it has, that a remote repositry doesn't have to push?

- there needs to be some data structure that represents the remote repository
- `struct remote` in remote.h looks like good starting point, especially it's `receivepack` member
- this leads me to `struct git_transport_options` in `transport.h`
- there's a promising constant `TRANSPORT_PUSH_ALL` in `transport.h`
- it's used in `transport.c` shortly before `match_push_refs(local_refs, &remote_refs, rs` !
- That's where the magic is! it uses `struct refs` from back in the `remote.h` to represent both local and remote ends
- it uses a `struct refspec` to decide what which refs to push to remote end
- following the match logic few functions call down, in .git file terms, it's considering entries in `.git/refs/`
- it goes through all the refs to be pushed, and figures which commits needs to be pushed, building a `struct oid_array` of objects to push

What happens to commits that are not parents of any other commits and are not pointed to by branch/tag? Are they pruned, or do they live forever?

- let's start with `struct commit` again
- let's find where it's persisted to or read from disk, if there's some interface/abstract data structure that represents the store of commits/object it might tell us something
- there's `get_commit_buffer` which calls if the cached read fails, calls `read_object_file(&commit->object.oid..);` both in `commit.c`]
- i think we're interested in how objects indentified by oid are read, and persisted, and if we see places where oid's are deleted we might be onto something
- `struct object_id` just contains a fixed size-string represent a hash, reading it from disk returns a type, an array of bytes, and a length
- let's search for all functions that operate those structs, and see if there's deletion options
- `object-store.h` looks interesting, but nothing there seems to delete/clear anything, but `struct raw_object_store` looks like the representation of the objects on disk we're looking for

- the actually writing to disk happens in `sha1-file.c` where there's file descriptors, searching there for references to delete
- Trying another tack: searched repo for "prune"
- there is `prune_packed_objects()` in `built-in.h` - it deals with a datastructure I'd been ignoreing the packfile
- this corresponds to directly to a git command `git prune-pack`, checking the docs that seems to not be quite what we're looking for
- that refers to deleting 'loose' objects that already exist in 'packed files' (`struct packed_git` in `object-store.h`) which clearly pack together objects
- I was expecting we'd see a delete object in `object-store.h`, which the packfile interface would call, but not, it simply calls `unlink_or_warn(path)` to delete the file corresponding to the object
- I think we can assume a function, or one like that would have be to called to delete any object, so let's search for all instances of it, or functions like it
- `git-compat-util.h` has 3 functions then that remove files:

```
int unlink_or_warn(const char *path);
int unlink_or_msg(const char *file, struct strbuf *err);
int remove_or_warn(unsigned int mode, const char *path);
```

- ah! there's a `gc.h` which cleans corresponds to the `git gc` command and uses `unlink_or_warn` nothing else looks interesting
- but it removes unreachable objects! and not commits
- my conclusion then is: commits aren't deleted, but unreachable objects created will at some point be in the normal course of activities when `git gc` is called another command