PyGit

PyGit is an attempt to recreate the core functionalities of Git from scratch using Python. After reading through GitInternals.docx, it's clear that git's operations can be broken down into some modules: Take a file and calculates its SHA-1 value, store it as a blob, index a number of blobs to a tree, write the tree, add an author, time, and message to a tree, and write a commit. Keeping this in mind, let's jump into it.

Tools required:

SHA1 Algorithm: Secure Hash Algorithm -1 (SHA-1) is a pretty popular cryptographic algorithm. It takes in an input and outputs a 160 bit long hash-value, usually represented as a 40-character long hex value. Also it is designed to be a one-way algorithm.

SHA characteristics: SHA-1 has following 3 important characters:

1. Pre-image resistance: Very hard and time consuming to find the original value m, given the has value h.
2. Second pre-image resistance: Given a message M1, it is very hard to find a new message M2 that hashes to the same value.
3. Collision resistance: Two messages having same hash value, such messages are extremely difficult to find.

Process:

- Works by inputting a string of length < 2^{64} bits, and outputting a 160-bit hash value.
- Simple method explained here, two methods.
- Take 'abc' for example.
- Represent it in binary.
- Generate 5 random strings of hex characters. These are H0, H1…H4.N
- Now, to the original binary message (24 bits), append a 1, and then append 0's until it reaches 448 bits.
- The length of the message (24) is represented in 64 bits and added to the binary string, bringing its length to 512.
- Let this string be M1.
- The M1 is divided into chunks of 512 bits each (only 1 chunk here).
- Each chunk is divided into 16 32-bit words. W0, W1…W15.
- For each chunk, begin 80 iterations.
- For iterations 16 to 79, where 16 <= I <= 79, perform W(i) = S^1(W(i-3) ^ W(i-8) ^ W(i-14) ^ W(i-16)), where ^ represents XOR operation.
- S^n is the left circular shift of a binary string by n-bits.
- Now define the following variables: A = H0, B = H1, C = H2, D = H3, and E = H4.

- Now, for 80 iterations, 0 <= i <= 79, do the following.
- TEMP = S^5 * (A) + f (i; B, C, D) + E + W(i) + K(i)
- Now reassign the variables:
  - E = D
  - D = C
  - C = S^{30}(B)
  - B = A
  - A = TEMP
- Store the value of chunk's hash to the overall value of all chunks as shown, and proceed to next chunk.
  - H0 = H0 + A
  - H1 = H1 + B
  - H2 = H2 + C
  - H3 = H3 + D
  - H4 = H4 + E
- The f function is a function on B,C,D and is dependent on value of i.

$$f(i;B,C,D) = (B \wedge C) \vee ((\neg B) \wedge D) \qquad \text{for } 0 \geq i \geq 19$$

$$f(i;B,C,D) = B \oplus C \oplus D \qquad \text{for } 20 \geq i \geq 39$$

$$f(i;B,C,D) = (B \wedge C) \vee (B \wedge D) \vee (C \wedge D) \qquad \text{for } 40 \geq i \geq 59$$

$$f(i;B,C,D) = B \oplus C \oplus D \qquad \text{for } 60 \geq i \geq 79.$$

f produces a 32-bit output by taking in 3 32-bit inputs.

- K(i) used is a hex string and is also dependent upon i.

$$K(i) = 5A827999, \qquad \text{where } 0 \leq i \leq 19$$

$$K(i) = 6ED9EBA1, \qquad \text{where } 20 \leq i \leq 39$$

$$K(i) = 8F1BBCDC, \qquad \text{where } 40 \leq i \leq 59$$

$$K(i) = CA62C1D6, \qquad \text{where } 60 \leq i \leq 79.$$

K also returns a 32-bit output.

- The final hash value H would be, H = $S^{128}(H0)$ OR $S^{96}(H1)$ OR $S^{64}(H2)$ OR $S^{32}(H3)$ OR (H4).

For our purpose, we'd use python's hashlib library. The sha1() of hashlib would compute the SHA1 hash-value for a string, and you can access the digest using the hexdigest(), it converts the 160-bit string to 40 character hex value.

Bash-compatible shell: Bash is a command process that runs commands that causes actions. We need a python compatible bash shell, so either you can run it on a UNIX-like environment, or use WSL if on windows.

Library to parse command-line arguments: To make the application work on command line interface, we need a library to parse command-line arguments. We will use 'argparse'.

Argparse: This library provides functionality to parse command line arguments. It has a couple of different components. It has argument, options, and parameters.

- First create a ArguementParser object. It holds all the necessary information to parse the command line into python data types.
- Adding arguments tells the parser how to take string and turn them into objects.
- parse_args() method will parse the command line, convert each argument to appropriate data type, and then invoke the appropriate method.
- Subparsers are added to module the commands into different subcommands (git commit, git init etc).

Configuration file

Git uses a configuration file format in Microsoft INI format. It is a configuration file for computer software. It consists of key-value pairs for different properties of the software. Different keys are partitioned into different sections denoted by [].

We'll implement this using Python library ConfigParser.

Compression

Git compresses files using zlib, so we'll use the python library zlib.

Starting off

- Create two files, a code file named libpygit.py, and an executable file called pygit.
- Inside the executable file, import the libpygit, and then call the main function.
- Make the file executable by going to your shell and using chmod +x pygit. (Use sudo if permission issues arise).

- All the functionalities would be implemented inside src folder, and all the functions would be called to libpygit.py as a wrapper.

File Utilities

- Built inside src/fil_utilities.py
- read_file function takes in a path parameter, opens the data present in the file, and returns the data in binary form.
- write_file function takes in a path and data parameter, opens the file present at the path in binary form, and writes the data.

Hash Object

- hash_object takes in data, obj_type, and a bool write as inputs.
- The header is a binary string. The first word is the file type, followed by a space, and then includes the length of the data.
- The full data is the header appended with a null character and the data string.
- Find the sha1 value of the full data using hashlib.sha1. Convert it to hexdigest.
- If write bool is true, open a file in the path. Path is inside the .pygit/objects/sha1[:2]/sha1[2:].
- Inside the file, dump the full data after compressing it with zlib.

Read and Find Object

- Built inside the object_utilities.py.
- find_obj takes in a sha1 prefix and finds an object using it.
- Check if the length of sha1 prefix > 2.
- Go inside the .pygit/objects/sha1[:2] directory. If it does not exist, return an exception.
- Maintain rest = sha1[2:].
- Cycle through the files inside the path. If any of them starts with the rest, appends it to an objects list.
- If the length of the objects list is 0, return an error saying 'no objects found'.
- If the length of the objects list is > 1, return an error saying multiple files found.
- Else, return the path to the object file found.

- read_obj also takes in an sha1-prefix and returns the object type as well as the data present inside.
- First call the find_obj using the sha1-prefix, store its return value in a path
- Extract full data using read_file(path) and decompressing it.
- Find the null character index in the full data.
- Extract header as full_data[:null_index].
- Extract obj_type and obj_size by splitting the header by the space.

- Extract data as full data [null_index + 1:].
- Check if the obj_size == len(data).
- Return obj_type and the data.

Index

Now let's start with the index part of the pygit. Indexing is one of the most important components of the pygit functionality.

Structure of the index object.

- We build a schema for IndexEntry using collections.namedtuple.
- The keys include:
  - ctime_s: The last time a file's metadata has changed (seconds).
  - ctime_n: The last time a file's metadata has changed (nanoseconds).
  - mtime_s: The last time a file's data has changed (seconds).
  - mtime_n: The last time a file's data has changed (nanoseconds).
  - dev: ID of device containing this file.
  - ino: The file's inode number.
  - mode: The object type. Eithe b100 (regular), b1010 (symlink), b1110 (gitlink).
  - uid: User ID of owner.
  - gid: Group ID of owner.
  - size: Size of this object, in bytes.
  - sha1: SHA1 hash-value of the object.
  - flags: Flags for validation.
  - path: Path to the object.
- The index is read in the read_index() function.
- Read the contents of the file .pygit/index
- The last 20 characters are a checksum, so create a digest using the rest of the file contents. Hash them and store them in form of bytes.
- Check if the checksum is equal to the digest.
- If yes, unpack signature, version, and number of entries from the first 12 charcters.
- Assert that the signature is equal to b'DIRC', else raise an exception.
- Let the entry data be data[12:-20]
- Loop over the entire data, the start index of every index be i.
- Extract the fields from the data[i:i+62]
- Extract the path from fields_end(i+62) to the next null character.
- Convert all this data into an IndexEntry object and append it to the entries list.
- Assert that len(entries) == num_entries
- Return the list of entries.

- The index is written in the write_index function.
- It takes in entries list as an argument.
- Loop over the entries list.

- Build an entry_head by packing up the entry object.
- Encode the path.
- Find the length of the data.
- Build the packed entry by appending entry_head and path and padding it with null characters.
- Append the packed_entry to the list of packed entried.
- Build an header for the index file by packing a signature, version, and number of entries.
- Build the all_data by appending the header with all the entries.
- Find the hash of the all_data and append it at the end.
- Write the whole thing to the index file.

Reading and Writing trees

- Reading the tree is done in read_tree.py.
- It needs either the hash of the tree or the actual data to be passed into it.
- Initialize a loop to run 1000 times. Also initialize a variable i = 0.
- Find the next null character from i.
- If there is none, break the loop.
- Find the mode and path by splitting the string by space.
- Convert the mode to an integer.
- Find the digest by taking string from end + 1 to end + 21.
- Make a tuple of mode, path, and digest.
- Append the tuple to the list of entries.
- Set i = end + 20
- Return the list of entries.

- Write tree is implemented in write_tree.py.
- Loop over the entries returned by read_index().
- Assert that each one's path is a file and not a folder.
- Make a header mode_path by appending mode and path with a space.
- Create a tree entry by joining mode_path and hash value of the entry and joining with a null character.
- Append it to a list of tree entries.
- Write it to an object using hash_object function. Pass the appended b-string of all the entries, and pass the obj_type 'tree'.

Cat_file

- cat_file function takes in the mode and the hash value as the parameters.
- Find the obj_type and the data using read_object.

- If mode is either ‘blob’, ‘tree’, or ‘commit’, check if mode == obj_type. If yes, flush the contents in stdout.
- Else if mode is size, print the len of the data.
- Else if mode is type, print the obj_type.
- Else if mode is pretty, flush the content of the data in stdout if the obj_type is either blob or commit.
- If the obj_type is tree, use the read_tree function. It’d return a list of tree_entries.
- Check the mode and print either ‘blob’ or ‘tree’ for each entry.
- Then print a formatted string of mode, sha1, and path for each entry.

Add

- The add function is implemented in the add.py.
- It takes in a list of paths of files to be added.
- Read the entries from the index using read_index.
- Create a list of entries which are not in paths.
- Loop over the paths.
- For each path, create the hash of the file using hash_object.
- Find the stats for each path.
- Create flags = len(path.encode()).
- Create an IndexEntry object using appropriate values.
- Append the entry to the list entries.
- Sort them by path.
- Write the entries to index using write_index.

Get Master hash value

- Read the file .pygit/refs/heads/master.
- Decode the data present in the file.
- Return the sha1 value of the master commit.

Commit

- Commit takes in a message and an author.
- First, return a tree by write_tree.
- Then, get a parent from using get_local_master_hash.
- Extract timestamp from the system.
- Also extract utc offset.
- Then define the author time.
- Initialize a list of lines as starting with ‘tree’+tree.
- If the commit has a parent, add a line ‘parent’+parent.
- Add a line ‘author {} {}’.format(author, author_time).
- Add a similar line for commiter.

- Leave a line, then add a line for the message.
- Leave another line.
- Extract data by joining all the lines with '\n'.
- Find its sha1 value by passing it to hash_object. It also saves the object.
- In the master_file, .pygit/refs/heads/master, write the sha value of this commit.
- Print a commit message to screen, then return the sha1 value of the commit object ast the return value of the function.

LS files

- Used to list the files currently in the index.
- Takes a bool 'details' as input.
- Return the list of entries in index using read_index.
- Loop over them.
- If details is true, print a bunch of stuff on the screen (mode, sha1.hexdigest(), stage, path)
- Else, just print the path on the screen.

Status

- Used to find if any of the files present in the index have changed.
- status is the main function.
- It imports 3 sets from the get_status function, changed, new, and deleted.
- Build a set of paths using os.walk.
- Build a set of index_paths by read_index and converting it to a set.
- A file is changed if its path is in both paths and entry_paths, but have different sha1 values.
- The new files are the set difference between paths and entry_paths.
- The deleted files are the set difference between entry_paths and paths.
- Print all of them inside the status function.

Diff

- Diff is used to highlight the changes in the file and in the index.
- Get a set of changed files from get_status.
- Get a list of index entries from read_index.
- Loop over the changed files by index and path.
- Using the path, get the hash value of the previous file from the index by using the path.
- Read the object using the hash value. Find the data of the precious version of the file.
- Check if the object is of type 'blob'.
- Make a list of lines from the index copy of the file by splitting data by newline.

- Make a list of lines from the working copy of the file by splitting the return value of read_file(path) by newline.
- Create a list of different_lines using difflib.unified_diff.
- Print all the diff_lines.

Checkout

- Used to return the working area to a previous commit.
- Checkout takes in input a message which is associated to a previous commit.
- First, delete everything in the working area.
- Now, get master hash using get_local_master_hash.
- Get the hash of the commit you want by passing the hash and the message to checkout_recur.
- Checkout_recur recursively checks each commit if its message is same as the command given by the user. If found, it returns the hash value of the commit.
- If checkout_recur returns -1, raise an error saying no commit found with said message.
- Else, read the commit_object using read_object.
- Find the hash of the tree associated to the commit.
- Feed it to the tree_checkout function.
- Inside the tree_checkout, find the entries of the tree using read_tree.
- Loop over the entries.
- Write the data of the entry at the path of the entry.

Logs

- Inside the log function, first find the head sha using get_local_master_hash.
- Feed it to the get_log function.
- The get log function inputs a hash of a commit function, and prints the commit.
- If the commit has a parent, then it recursively calls the parent of the commit.