A Data Structure Oriented Introduction to Git and GitHub Part 1

Daniel Allen Langdon
http://danielsadventure.info

Why?

Up until recently, I was the guy in this XKCD. (http://www.xkcd.com/1597/) There are many books and videos about Git. It has been my experience that Git is usually explained by pointing out commands that are used for common tasks.

I want to take you beyond the commands to how Git actually works.

I will teach you about Git's data structures at the expense of teaching the commands because the commands are easy to understand when you learn the data structures first.

If you have ever deleted your copy of a Git repo and recloned because you didn’t know what was going on…

If you have ever made a copy of your data outside of Git because you feared destroying your data with the wrong command...

This is for you!

Keep This in Mind...

“Bad programmers worry about the code. Good programmers worry about data structures and their relationships.” -Linus Torvalds, creator of Git

Let’s worry about the data structures and their relationships rather than the code!

That being said, there is a data structure fundamental to Git which may look familiar.

The simplest Git repo resembles a stack.

When a new value is pushed, a node is created to hold it, and when a value is removed, its node is removed.

```
function Stack() {
    function Node(value, next) {
        this.value = value;
        this.next = next;
    }

    this.top = null;

    this.push = function push(value) {
        this.top = new Node(value, this.top);
    }

    this.pop = function pop() {
        var value;
        if (this.top === null) {
            throw new Error("No data to pop from stack.");
        } else {
            value = this.top.value;
            this.top = this.top.next;
            return value;
        }
    }
}
```

Each incremental change in a Git repo is called a **commit.** A commit contains the contents of every file that is changed by that commit, as well as who created the commit and when it was created.

By placing these structures in a stack structure as in the previous slide, we can save incremental changes and keep track of who made the changes and when.

Simply traverse the stack from top to bottom and you have all the files in the project, as well as the history!

```
function Commit(author, filesChanged, dateTime) {
    this.author = author;
    this.filesChanged = filesChanged;
    this.dateTime = dateTime;
}
```

A Simple Worked Example

Suppose that we have a Git repo that will be used to track documentation for a project. The first file added is “english.txt”, the documentation for a product in English, written by Ernie Englishman. Sammy Spaniard and Frank Frenchman are asked to translate the document into Spanish and French, respectively and add the translation to the repo.

Let’s work thru this example in the Javascript I provided.

```
var stack = new Stack();
stack.push(new Commit("Ernie", [{ name: "english.txt", contents: "Hello World!" }], new Date()));
stack.push(new Commit("Sammy", [{ name: "spanish.txt", contents: "Hola Mundo!" }], new Date()));
stack.push(new Commit("Frank", [{ name: "french.txt", contents: "Bonjour Monde!" }], new Date()));
```

How does it really work?

The previous slides were for illustrative purposes, but the data structures demonstrated are similar to what Git actually uses. A Git repo with a single branch works just like a stack data structure.

When we think of a stack, we usually think of each node residing at a specific place in memory. The stack structure uses pointers to allow us to traverse the structure.

Git commits are stored on disk, and rather than use a specific location on the disk, each commit is identified by a hash of its contents. A frequently asked question is: “Why doesn’t Git it simply number the commits sequentially?” The answer is that when multiple users work collaboratively, they would most certainly try to use the same number for completely different commits.

Git uses these hashes to identify what commit is associated to what other commit, and eventually, there will be commits that become disconnected from the graph. These can be garbage collected, just like in the Javascript stack example.

When one commit is added to a repo, we say that the commit that preceeded it is the **parent commit**.

Introducing a few basic commands

Git has over 100 commands with many parameters, but fear not! You can do 99% of your work with a few basic commands, and failing that, Professor Google is there to help you find what you need. Remember, if you focus on understanding the data structures, the commands are easy to learn and use, but if you don't understand the data structures, the commands may as well be Chinese.

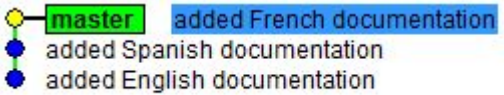
git init: initialize a new blank Git repo

git add: add files to the index, which refers to files that will be contained in a newly created commit

git commit: create a new commit from the index, and point it back to the previous commit

Here is how the three commands are used to create a new repo and add three files.
Each addition is a two-step process: add the file to the index and then commit the index to the repo. This process is just like pushing onto a stack, as previously described.

To drive the point home, please see this graph of the resulting repo produced with a handy utility called gitk.




```
C:\Users\Daniel\Documents\GitHub\test> git init
Initialized empty Git repository in C:/Users/Daniel/Documents/GitHub/test/.git/
C:\Users\Daniel\Documents\GitHub\test [master +3 ~0 -0 !]> git add english.txt
C:\Users\Daniel\Documents\GitHub\test [master +1 ~0 -0 | +2 ~0 -0 !]> git commit -m "added English documentation"
[master (root-commit) 668a120] added English documentation
 1 file changed, 1 insertion(+)
 create mode 100644 english.txt
C:\Users\Daniel\Documents\GitHub\test [master +2 ~0 -0 !]> git add spanish.txt
C:\Users\Daniel\Documents\GitHub\test [master +1 ~0 -0 | +1 ~0 -0 !]> git commit -m "added Spanish documentation"
[master 5d55d61] added Spanish documentation
 1 file changed, 1 insertion(+)
 create mode 100644 spanish.txt
C:\Users\Daniel\Documents\GitHub\test [master +1 ~0 -0 !]> git add french.txt
C:\Users\Daniel\Documents\GitHub\test [master +1 ~0 -0]> git commit -m "added French documentation"
[master 6f06483] added French documentation
 1 file changed, 1 insertion(+)
 create mode 100644 french.txt
C:\Users\Daniel\Documents\GitHub\test [master]>
```

What about when I need to go back?

Suppose that Sammy Spaniard turns out to be an incompetent writer, and we want to fire him and replace him with someone new, and we want to make our repo as if he had never worked on it.

Git provides three very handy commands for reverting the repo to a previous state.

git checkout: move the head pointer back to a previous commit in order to temporarily go back without changing the contents of the repo.

git revert: create a new commit that does the reverse of a given commit, for example, if we revert Sammy' s insertion of “spanish.txt”, the result will be a commit that deletes “spanish.txt”

git reset: move the branch pointer and the head pointer back to a previous commit, which may orphan one or more commits so that they are disconnected from the graph.

What’s a branch pointer? What’s a head pointer? These questions will be answered shortly.

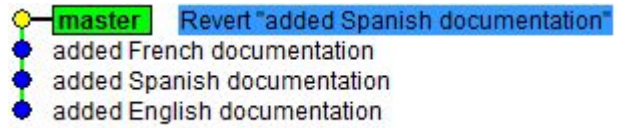
git log: list the commits that are in the repo. This is needed to use the previous two commits because you must specify the hash of the commit you want to revert or reset to.

A worked example with git revert

```
C:\Users\Daniel\Documents\GitHub\test [master]> git log
commit 6f06483dc9e64f0a408eddddb632a400ce7a16ad
Author: Daniel Langdon <me@danielsadventure.info>
Date:   Sat Dec 12 10:33:28 2015 -0700

    added French documentation

commit 5d55d61701f6e86c5095e08b7c9cc6547ef3c547
Author: Daniel Langdon <me@danielsadventure.info>
Date:   Sat Dec 12 10:33:14 2015 -0700

    added Spanish documentation

commit 668a12093795e328f35b94121f924718fda03420
Author: Daniel Langdon <me@danielsadventure.info>
Date:   Sat Dec 12 10:32:52 2015 -0700

    added English documentation
C:\Users\Daniel\Documents\GitHub\test [master]> git revert 5d55d61
[master 5bbb98a] Revert "added Spanish documentation"
 1 file changed, 1 deletion(-)
 delete mode 100644 spanish.txt
C:\Users\Daniel\Documents\GitHub\test [master]>
```

Note that when specifying the commit to revert, it is okay to specify only the first seven characters of the hash so long as they are unique.

Below is another gitk diagram. Note that the reversion is a new commit, so undoing a commit results in one new commit, not the removal of any commit.



What is a head pointer, anyways?

I previously mentioned that the checkout command moves the head pointer and the reset command moves both the branch pointer and the head pointer, but I didn't tell you what that means.

When you are working in a Git repo, the **head pointer** is the hash of the last commit in your working directory. The contents of your working directory are computed by going back from the head pointer to the beginning of the repo. Using the stack analogy, the head pointer is like the pointer to the top of the stack. Once again, you can use the checkout command to move the head pointer to any commit in the graph. This will alter the contents of your working directory based on where you move the head pointer. For example, if you use checkout to move the head pointer to where Ernie added the English documentation, the Spanish and French documentation will disappear from your working directory because they weren't yet added at that point. However, these commits still exist; you just don't see them anymore.

To undo your checkout, use the checkout command again with the branch you were working on. What's a branch? Coming up!

What is a branch pointer, anyways?

Now, it is time to break that nice stack analogy I introduced Git with. We will move on to a tree analogy, but don't worry, we'll break that nice analogy, too! A **branch pointer**, like a head pointer, is a reference to a commit by its hash value. There is only one head pointer at a time, but there can be unlimited branch pointers. By default, there is one branch pointer, and the one branch it points to is called **master**.

A Git repo with only one branch is like a stack. By adding branches, we make it more like a tree. Going back to our repo with documentation, let's suppose that our team members need to make changes. Ernie is asked to create separate documentation for British and American English, and Frank has been asked to correct punctuation errors in the French documentation.

Let's create some branches, and I promise to return to the question of how to use the reset command.

Branches can be created, deleted, and listed using the **git branch** command.

git branch: list all branches
git branch XXX: create the XXX branch
git branch -D XXX: delete the XXX branch

Ernie's Work

Ernie creates a new branch to separate the documentation into American and British English. On this branch, he performs his task and then creates a new commit. The boss will then add Ernie's work to the master branch.

First, he creates a "dialects" branch and moves the head pointer to dialects with the checkout command. He adds his changes to the index using the add command, and then creates a new commit using the commit command.

Let's look at the resulting graph with gitk, but first, let's check in on Frank.

```
C:\Users\Daniel\Documents\GitHub\test [master +2 ~0 -1 !]> git branch dialects
C:\Users\Daniel\Documents\GitHub\test [master +2 ~0 -1 !]> git checkout dialects
D       english.txt
Switched to branch 'dialects'
C:\Users\Daniel\Documents\GitHub\test [dialects +2 ~0 -1 !]> git add --all
C:\Users\Daniel\Documents\GitHub\test [dialects +2 ~0 -1]> git commit -m "divide English into two dialects"
[dialects 45e4a70] divide English into two dialects
 3 files changed, 4 insertions(+), 1 deletion(-)
 create mode 100644 american english.txt
 create mode 100644 british english.txt
 delete mode 100644 english.txt
C:\Users\Daniel\Documents\GitHub\test [dialects]>
```

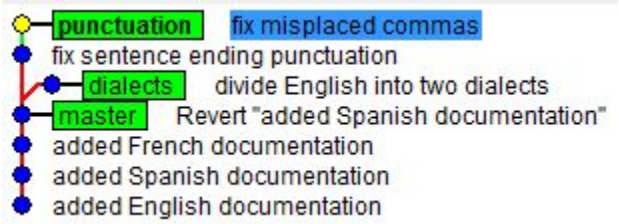
Frank's Work

Like Ernie, Frank creates a new branch for his work. He needs to fix the punctuation. He does this in two steps. He makes changes to fix all the sentence-ending punctuation and commits the change. He then makes changes to fix all misplaced commas. He then commits the change.

Let's look at the graph, once again using gitk.

```
C:\Users\Daniel\Documents\GitHub\test [master]> git branch punctuation
C:\Users\Daniel\Documents\GitHub\test [master]> git checkout punctuation
Switched to branch 'punctuation'
C:\Users\Daniel\Documents\GitHub\test [punctuation]> git add .\french.txt
C:\Users\Daniel\Documents\GitHub\test [punctuation +0 ~1 -0]> git commit -m "fix sentence ending punctuation"
[punctuation 9cabe5c] fix sentence ending punctuation
 1 file changed, 1 insertion(+), 1 deletion(-)
C:\Users\Daniel\Documents\GitHub\test [punctuation]> git add .\french.txt
C:\Users\Daniel\Documents\GitHub\test [punctuation +0 ~1 -0]> git commit -m "fix misplaced commas"
[punctuation 7a49497] fix misplaced commas
 1 file changed, 1 insertion(+), 1 deletion(-)
C:\Users\Daniel\Documents\GitHub\test [punctuation]> _
```

An overview of what we've done so far



On the master branch, we created three files for English, Spanish, and French documentation. We then deleted the Spanish documentation. Notice that the master branch points to the removal of the Spanish documentation, the last commit we made on the master branch, because Ernie and Frank's new tasks have not yet been added to the master branch.

Then, our beautiful stack splits into a tree with two branches. Ernie's "dialects" branch contains a commit that provides separate files for British and American English. Frank's "punctuation" branch contains two commits which together correct all punctuation errors in the French documentation.

The boss says that this all looks good and he wants to move all these changes onto the master branch, but there's a problem. The company's guidelines dictate that each task shall have one and exactly one commit to represent changes for that task. Frank is in violation of the policy with his two commits for the punctuation task. Let's see how Frank fixes this.

Getting Frank into Compliance

I will now give an example of how to use the reset command. Recall that the commands revert, checkout, and reset all allow Frank to go back. I warned that reset can result in permanent loss of data. In this case, we want to replace two commits with one commit and discard the old commit.

Pay careful attention here because the reset command is dangerous!

git reset XXX: move the branch pointer and the head pointer to the commit with hash XXX. **This may leave one or more commits inaccessible from any branch.** If the **--hard** parameter is applied, it will also reset the contents of the working directory based on the new commit. **This may result in permanent data loss**. Otherwise, we call it a “soft reset”.

Another shortcut: Thus far, I have always used the first seven characters of a hash to refer to a commit. There is a shortcut. You can use head~n to refer to the commit n commits behind the head pointer.

Also note, there is another command called **reflog** which will allow us to see commits that are no longer accessible from any branch. This command may be used to prevent data loss, but beware: such commits are eventually garbage collected, so get them before the clock runs out!

In short, reset is a dangerous command, but the damage can usually be mitigated.

Be careful, Frank! git reset is dangerous!

In this case, Frank's task is relatively simple. Frank must discard two commits and replace them with a single new commit.

In order to guard against data loss, Frank creates a new branch called punctuation2 and uses git checkout to move the head pointer to it.

He uses "git reset head~2" to go back two commits. The commits are no longer on the punctuation2 branch and the changes from those two commits remain in the working directory. He then creates a new commit and sends it to the boss for review.

```
C:\Users\Daniel\Documents\GitHub\test [punctuation]> git branch punctuation2
C:\Users\Daniel\Documents\GitHub\test [punctuation]> git checkout punctuation2
Switched to branch 'punctuation2'
C:\Users\Daniel\Documents\GitHub\test [punctuation2]> git reset head~2
Unstaged changes after reset:
M       french.txt
C:\Users\Daniel\Documents\GitHub\test [punctuation2 +0 ~1 -0]> git add french.txt
C:\Users\Daniel\Documents\GitHub\test [punctuation2 +0 ~1 -0]> git commit -m "Fix all punctuation in French"
[punctuation2 2d6fc2a] Fix all punctuation in French
 1 file changed, 1 insertion(+), 1 deletion(-)
C:\Users\Daniel\Documents\GitHub\test [punctuation2]> _
```

The Boss' Task

The boss now has a job to do. He has his QA people review the work done by Ernie and Frank, and everything looks good. He wants to copy the work done into the master branch.

There are two commands used to do this, each having advantages and disadvantages. I will demonstrate the task with both.

git merge: Merge commits from two branches together. This has the advantage of often being the easier operation when many commits are involved, but at the same time, it has the disadvantage of leaving forks in our master branch.

git rebase: Apply commits from one branch on top of commits from another branch. This has the advantage that it avoids forks in the master branch, but we may have merge conflicts to resolve for each commit, compared to just one potential merge conflict using merge.

In the example I am using, Ernie and Frank are working on completely separate files, so there are no conflicts to resolve. (Conflict resolution is a broad topic for another day.) Still, I will demonstrate both commands.

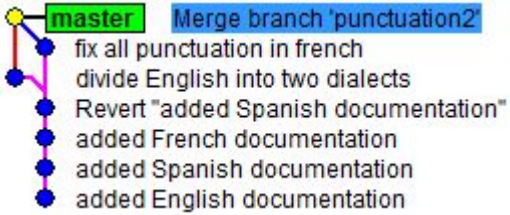
Remember what I said about keeping in mind the data structures? Without understanding the data structures, these common commands are extremely confusing!

git merge

This is where I keep my promise to break not only the stack structure for the master branch, but the tree structure, as well. git merge creates a new commit from combining two branches and the resulting commit has two parent commits.

In this example, the boss merges Ernie and Frank's work into the master branch and then deletes the old branches because they aren't needed anymore.

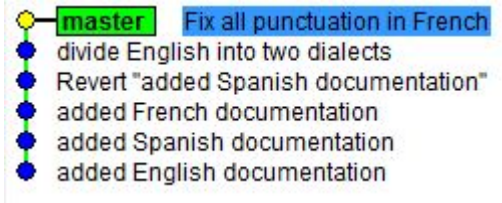
```
C:\Users\Daniel\Documents\GitHub\test [master]> git merge dialects
Updating 5bbb98a..1f4f4eb
Fast-forward
 american english.txt | 2 ++
 british english.txt  | 2 ++
 2 files changed, 4 insertions(+)
 create mode 100644 american english.txt
 create mode 100644 british english.txt
C:\Users\Daniel\Documents\GitHub\test [master]> git merge punctuation2
Merge made by the 'recursive' strategy.
 french.txt | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
C:\Users\Daniel\Documents\GitHub\test [master]> git branch -D punctuation
Deleted branch punctuation (was d11e0ee).
C:\Users\Daniel\Documents\GitHub\test [master]> git branch -D punctuation2
Deleted branch punctuation2 (was 4a0dbde).
C:\Users\Daniel\Documents\GitHub\test [master]> git branch -D dialects
Deleted branch dialects (was 1f4f4eb).
C:\Users\Daniel\Documents\GitHub\test [master]> gitk --all
C:\Users\Daniel\Documents\GitHub\test [master]>
```



git rebase

With git rebase, the commits from punctuation2 and dialects are applied to the master branch. The old branches may then be deleted. Notice that this preserves the master branch in a stack structure.

```
C:\Users\Daniel\Documents\GitHub\test [master]> git rebase punctuation2
First, rewinding head to replay your work on top of it...
Fast-forwarded master to punctuation2.
C:\Users\Daniel\Documents\GitHub\test [master]> git rebase dialects
First, rewinding head to replay your work on top of it...
Applying: Fix all punctuation in French
C:\Users\Daniel\Documents\GitHub\test [master]> git branch -D punctuation
Deleted branch punctuation (was 7a49497).
C:\Users\Daniel\Documents\GitHub\test [master]> git branch -D punctuation2
Deleted branch punctuation2 (was 2d6fc2a).
C:\Users\Daniel\Documents\GitHub\test [master]> git branch -D dialects
Deleted branch dialects (was 45e4a70).
C:\Users\Daniel\Documents\GitHub\test [master]> ls


    Directory: C:\Users\Daniel\Documents\GitHub\test


Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
-a---         12/12/2015   1:22 PM         83 american english.txt
-a---         12/12/2015   1:22 PM         32 british english.txt
-a---         12/12/2015   1:23 PM         15 french.txt


C:\Users\Daniel\Documents\GitHub\test [master]>
```



Review of Part 1

Thus far, we have discussed how Git stores your data and its history.

We have discussed a number of commands for common tasks including:
git init
git add
git commit
git checkout
git revert
git reset
git log
git branch
git merge
git rebase

We worked an example of several people working together, but this is not how it typically works. Typically, each person has their own copy of the repo, and it is necessary to use additional commands to synchronize work across each person's copy. We will cover that in Part 2.