Shelving v1/v2/v3 Development Review

This is a review of the development of experimental support for shelving in Subversion. Shelving v1 was released in Subversion 1.10, shelving v2 in Subversion 1.11, and shelving v3 in Subversion 1.12. These versions are not a linear sequence of improvements but rather three different designs, each developing a different aspect, with different pros and cons.

The Start

Shelving has been on the Subversion developers' wish list for a long time. This item was initially picked by my employer to be a bite-sized task to show off creating a new feature. The idea was that we could quickly make a wrapper around "svn diff" and "svn patch" that would save the user's changes into a file, revert those changes from the working copy, and later apply the changes back to the working copy. A built-in light-weight patch management feature.

Other popular version control systems have their own kinds of shelving. For example, git has a simple and powerful "git stash", and perforce neatly treats "shelved" as one possible storage state of a changeset, like a variant of "committed".

I started by reviewing the kinds of shelving that have been proposed for Subversion in the past, and the facilities offered by other version control systems, and sketching some ideas of what a shelving design for Subversion could look like, ranging from light-weight management of patch files through to DVCS-like local branching. One notable idea, yet to be developed, was that we should integrate the existing "changelist" feature into shelving: "shelve" would mean moving a changelist from the WC to a shelf, and "unshelve" applying a changelist from a shelf to the WC, all the while keeping track of it by its name. But first we would need something as simple as possible, a place to store shelved changes, and mechanisms to shelve and unshelve changes.

Shelving and Checkpointing

In the wish list along with shelving was also the idea of checkpointing one's work in the working copy. The idea was along the lines of taking a snapshot of the working copy before making changes, so that we could roll back to an earlier state if the changes didn't work out, particularly if the changes were something like an "update" that changes the WC base state, or a merge that might result in conflicts.

I had been involved in previous discussions around this idea, when the focus was on how we could implement snapshots within the WC DB. My own view of the WC DB is that its design and implementation is far too hairy and fragile to attempt something like this at the low level. I could imagine a design at a higher level where snapshots are saved more or less like commits in a repository. The architecture of Subversion's client side and working copy is too far removed from the repository architecture to make that a feasible integration in the current code base. I did explore embedding an actual Subversion repository, or some layers of a repository, to use as a shelf storage unit. There are some possibilities there but not in a short term development.

Instead I re-imagined checkpointing as referring to taking snapshots of work in progress in the form of a series of numbered patches, capturing only the committable changes but not the WC base state. This would not provide the ability to roll back after a bad update, which is a pity. However, it would be useful. The evidence of this is that developers including myself already tend to save a series of numbered patch files to capture iterations of our work in progress, in certain cases where we feel committing is not appropriate or not convenient.

Therefore I ended up lumping the terms "shelving" and "checkpointing" together, "shelf" referring to a series of snapshots sharing a common name, description, and WC base; and "checkpoint" referring to one numbered snapshot within a series. Thus these would become light-weight UI mechanisms on top of the same underlying implementation which I refer to generically as shelving.

Shelving v1: Patches

We recognized from the outset that "svn diff" and "svn patch" do not support the full range of changes that Subversion can represent in a commit. They have no way to represent "binary" file content, nor copies and moves, nor creation and deletion of a directory. Defining representations for all types of change has been on the wish list for ever, and was something I thought we could come back to later, after implementing shelving of the currently supported change types. It is almost an orthogonal issue, with little bearing on the patch management layer other than defining a suitable failure mode when unsupported change types are present.

Shelving v1 therefore is basically a user interface for writing the output of "svn diff" into a named and numbered patch file, optionally running "svn revert" on the affected files, and later replaying a selected patch file into "svn patch". Minor additions include saving a descriptive message, like a commit log message, for the user's reference.

Handling of unsupported change types is rudimentary. A shelving attempt fails if unsupported changes are present in the requested scope. An unshelving attempt fails if any local modifications are present in the working copy among the paths to be unshelved. At least that is what the command-line interface implements. It was necessary to provide some checking or "dry run" APIs so that the client code could fail gracefully and let the user know that unsupported changes were going to prevent shelving or unshelving. Especially the latter, as we do not want the client to start applying changes to the WC and then hit a failure part way through, as the WC interface provides no roll-back mechanism to recover cleanly in such a case.

A more sophisticated client might want to offer the option of skipping unsupported changes instead of completely refusing to shelve anything. However, skipping some changes can lead to further difficulties. For example, if we skip shelving a "rename directory", that makes it awkward to shelve and unshelve any changes that occurred inside that directory. Therefore keeping the options simple and crude was the most expedient decision to get something useful working.

This implementation was released as an experimental feature in Subversion 1.10.

Shelving v2: Binary Snapshots

Patch files are all very well for text changes, but a large class of customers who pay for Subversion hosting are games developers who use Subversion because of its support for large binary files such as images and videos. What can we do to enable shelving those?

We could upgrade the patch file format to include binary data, perhaps encoded into some text-compatible form.

Or we could store file blobs as separate files.

There are trade-offs. Speed is a concern, as “binary” files tend to be large. Using whole files brings the possibility of moving files on the filesystem when the user does not need to keep a copy, and perhaps using linking or copy-on-write semantics when they do. Using a patch, we would be encoding and decoding the binary blobs into a patch stream, which means linear processing on every use. The patch format is not natively indexed for quick seeking, so that limits all accesses to the patch, even accesses to small files, unless we invent and add an index.

A second concern was patch application versus 3-way merging. A patch is a context diff. It stores the before and after version of each changed hunk, and a little context around each hunk, but not the whole original file. Applying a patch works reliably when the file we’re applying it to is the same original file as when the change was shelved. If not, perhaps because we updated and fetched changes that somebody else committed in the mean time, then applying a patch works only so far. It works when the changes being unshelved neither overlap nor functionally interact with changes that were brought in by the update. There is no way to convert the patch application to a 3-way merge and review the two sets of changes independently, unless we were to read the original version numbers from the patch file and fetch those versions from the repository.

A better position to be in, when resolving conflicts, is to have immediate access to all the versions we need for a three-way merge. The original base version of the change, and the whole changed file (which we can either store explicitly, or reconstruct from base + patch), and the new base version onto which we want to apply the patch.

The three-way merge enables (1) more robust merging of text changes, because the whole context is available to inspect, not just bits of context; and (2) the possibility of merging non-text files such as images, or arbitrary document file types, by using an external three-way merge tool that understands those file types.

We changed the shelf storage implementation to store the ‘before’ and ‘after’ version of each changed file, for all files. (Also for property changes.) No more patch file.

To apply changes, it uses a 3-way merge per file. It does not yet use a 3-way merge for tree changes. It is not yet integrated with the merge conflict handling mechanism that ‘update’ and ‘merge’ commands use.

This version was released in Subversion 1.11.

Shelving v3: Piping Changes between WC and Shelf

For the next version of shelving I determined to address the architectural problem. What problem? The storage for a shelf needs the same core semantics as the WC: a base layer plus some committable modifications. The WC had no generic API for “read the committable changes”, nor for “write these changes to the WC”. The core operations of shelving should be as simple as two pipes:

The core of “shelve”:

- wc.read_changes() | shelf.write_changes()

The core of “unshelve”:

- shelf.read_changes() | wc.write_changes()

What was it like before?

Shelving v2 walked the tree of WC paths “manually”, calling separate API functions to read and write each little bit of user data (file text, properties, directories). The API for the shelf storage was completely different from the WC API. For one conceptually simple job, copying changes this way or that way, there were two large chunks of manually written code, with no re-use and lots of room for bugs.

Shelving v1 used the API of “svn diff” and of “svn patch”:

- wc.diff() > shelf_patch_file
- wc.patch() < shelf_patch_file

That had the desired properties of re-using generic APIs and using a single API entry point to process the whole tree, but it did not have the right semantics for the stored changes. As a context diff, it lacked the base version, and the patch format lacked support for binary data, mkdir/rmdir, copies and moves.

Shelving v3 adds APIs to libsvn_wc that are analogous to “diff” and “patch” but instead of streaming the changes to or from patch file format they connect to the existing “delta editor” API which is what Subversion uses to transfer changes to and from the repository, in commit and update operations. The delta editor API is the “pipe” in the core operations of shelve and unshelve:

- wc.read_changes() | shelf.write_changes()
- shelf.read_changes() | wc.write_changes()

Providing the new wc.read_changes() involved a relatively simple extraction of part of the existing WC “commit” logic, removing all the repository-specific details to leave just the core function of sending changes to a delta editor. (This new API is actually named “svn_client__wc_replay”.)

The new wc.write_changes() API was written from scratch. (This new API is actually named “svn_client__wc_editor”.) Ideally this might have been extracted from some existing code such as “merge”, “patch”, or “update”; but “merge” and “patch” both work with context diffs rather than the delta editor interface, and the implementation within “update” was not readily re-usable. One place was found where this new API could be re-used as a drop-in replacement: in the implementation of repository-to-WC copy, which previously incorporated a dedicated delta editor that only knew how to apply “add” operations.

Next we needed to re-implement shelf storage in such a way that we could store the base of the changed files/paths, and write and read changes on top of that base via a delta editor API.

Shelving v3: Storing the Base

The quickest ways to create a WC base layer are the same ways we currently get a WC: either “checkout”, or copy an existing WC and revert the local modifications. Neither is ideal, but both work.

Ideally, what we want for the shelf base storage is some sort of skeleton tree that includes only the paths that have shelved changes, and storing for each such path just a reference to the existing WC base storage. This would take very little space and time. The APIs to accomplish this have not yet been developed. (They are needed not only for shelving but also for “view specs”, a requested feature for saving and restoring the WC base shape.)

In order to prove the concept of using the delta editor API for shelving, “checkout” is initially used to create a WC base for each shelf. The down-sides to this are of course that unless there is a very fast connection to the server this can take a long time, and it is heavy on disk space because it is not constrained to just the shelved paths.

Shelving: Work Needed

Currently Subversion the project has insufficient developer resources to continue these developments. If and when a volunteer or paid developer wants to take up the development, here are some suggestions.

Improving Shelving v3

Making a copy of the WC base state more efficiently is the first priority. The WC design does not seem amenable to cleanly creating a WC whose base layer is a reference to the base layer in another WC, nor to cleanly creating a “shelf” working layer inside the main WC that refers to the existing base layer. A quick-and-dirty approach could be to copy the whole WC and revert all the changes. To do the same but copying only the metadata “.svn” directory without copying the working files, might require modifying the revert code to make it work in fully “metadata-only” mode, which would be a useful upgrade anyway.

Shelving v3 currently fails some of its tests. Some are related to “mkdir”, which may require a relatively minor fix. Some are related to merging when applying changes into a WC that no longer matches the original base state. This is because the “apply” code does not attempt merging. Somehow the “apply” code needs to be hooked up to a merge. Subversion’s main merge code is unfortunately not in a good state to be re-used like this. It may be desirable to implement a “merge” alternative to the simple “apply” API (svn_wc__editor).

Shelving: Current State

[UPDATED 2020-03-24] Subversion 1.14 LTS is due to be released soon. Subversion 1.14 looks set to include shelving v2 and v3, disabled by default and opt-in by setting an environment variable. The reason for making them disabled by default is that 1.14 is intended to be a long-term support release focusing on stability rather than experimental features.

Originally published at: https://blog.foad.me.uk/2020/03/02/svn-shelving-development-review/