Subversion Overview

(Based on Chapter 2 of Version Control With Subversion [http://svnbook.red-bean.com/])

(Compiled from r4615M)

Original authors:
Ben Collins-Sussman
Brian W. Fitzpatrick
C. Michael Pilato
Modified by:
Nathan Sprague

Subversion Overview: (Based on Chapter 2 of Version Control With Subversion [http://svnbook.red-bean.com/]): (Compiled from r4615M)

by Ben Collins-Sussman, Brian W. Fitzpatrick, C. Michael Pilato, and Nathan Sprague

Table of Contents

Chapter 1. Basic Usage

About this Document

This is a hastily trimmed-down version of Chapter 2 of the the book Version Control With Subversion. The original book can be downloaded from: http://svnbook.red-bean.com/. Wherever you see ???, the original chapter contained a reference to material from another chapter of the book.

Introduction

This chapter will not provide exhaustive coverage of all of Subversion's commands—rather, it's a conversational introduction to the most common Subversion tasks that you'll encounter. This chapter assumes that you've read and understood ??? and are familiar with the general model of Subversion. For a complete reference of all commands, see ???.

Help!

It goes without saying that this book exists to be a source of information and assistance for Subversion users new and old. Conveniently, though, the Subversion command-line is self-documenting, alleviating the need to grab a book off the shelf (wooden, virtual, or otherwise). The **svn help** command is your gateway to that built-in documentation:

```
$ svn help
Subversion command-line client, version 1.7.0.
Type 'svn help <subcommand>' for help on a specific subcommand.
Type 'svn --version' to see the program version and RA modules
  or 'svn --version --quiet' to see just the version number.

Most subcommands take file and/or directory arguments, recursing
on the directories.  If no arguments are supplied to such a
command, it recurses on the current directory (inclusive) by default.

Available subcommands:
   add
   blame (praise, annotate, ann)
   cat
…
```

As described in the previous output, you can ask for help on a particular subcommand by running **`svn help SUBCOMMAND`**. Subversion will respond with the full usage message for that subcommand, including its syntax, options, and behavior:

```
$ svn help help
help (?, h): Describe the usage of this program or its subcommands.
usage: help [SUBCOMMAND...]

Global options:
  --username ARG           : specify a username ARG
  --password ARG           : specify a password ARG
…
```

Options and Switches and Flags, Oh My!

The Subversion command-line client has numerous command modifiers. Some folks refer to such things as “switches” or “flags”—in this book, we'll call them “options”. You'll find the options supported by a given **svn** subcommand, plus a set of options which are globally supported by all subcommands, listed near the bottom of the built-in usage message for that subcommand.

Subversion's options have two distinct forms: short options are a single hyphen followed by a single letter, and long options consist of two hyphens followed by several letters and hyphens (e.g., `-s` and `--this-is-a-long-option`, respectively). Every option has at least one long format. Some, such as the `--changelist` option, feature an abbreviated long-format alias (`--cl`, in this case). Only certain options—generally the most-used ones—have an additional short format. To maintain clarity in this book, we usually use the long form in code examples, but when describing options, if there's a short form, we'll provide the long form (to improve clarity) and the short form (to make it easier to remember). Use the form you're more comfortable with when executing your own Subversion commands.

Getting Data into Your Repository

You can get new files into your Subversion repository in two ways: **svn import** and **svn add**. We'll discuss **svn import** now and will discuss **svn add** later in this chapter when we review a typical day with Subversion.

Importing Files and Directories

The **svn import** command is a quick way to copy an unversioned tree of files into a repository, creating intermediate directories as necessary. **svn import** doesn't require a working copy, and your files are immediately committed to the repository. You typically use this when you have an existing tree of files that you want to begin tracking in your Subversion repository. For example:

```
$ svn import /path/to/mytree \
             http://svn.example.com/svn/repo/some/project \
             -m "Initial import"
Adding         mytree/foo.c
Adding         mytree/bar.c
Adding         mytree/subdir
Adding         mytree/subdir/quux.h

Committed revision 1.
$
```

The previous example copied the contents of the local directory `mytree` into the directory `some/project` in the repository. Note that you didn't have to create that new directory first—**svn import** does that for you. Immediately after the commit, you can see your data in the repository:

```
$ svn list http://svn.example.com/svn/repo/some/project
bar.c
foo.c
subdir/
$
```

Note that after the import is finished, the original local directory is *not* converted into a working copy. To begin working on that data in a versioned fashion, you still need to create a fresh working copy of that tree.

Creating a Working Copy

Most of the time, you will start using a Subversion repository by performing a *checkout* of your project. Checking out a directory from a repository creates a working copy of that directory on your local machine. Unless otherwise specified, this copy contains the youngest (that is, most recently created or modified) versions of the directory and its children found in the Subversion repository:

```
$ svn checkout http://svn.example.com/svn/repo/trunk
A    trunk/README
A    trunk/INSTALL
A    trunk/src/main.c
A    trunk/src/header.h
…
Checked out revision 8810.
$
```

Although the preceding example checks out the trunk directory, you can just as easily check out a deeper subdirectory of a repository by specifying that subdirectory's URL as the checkout URL:

```
$ svn checkout http://svn.example.com/svn/repo/trunk/src
A    src/main.c
A    src/header.h
A    src/lib/helpers.c
…
Checked out revision 8810.
$
```

Since Subversion uses a copy-modify-merge model instead of lock-modify-unlock (see ???), you can immediately make changes to the files and directories in your working copy. Your working copy is just like any other collection of files and directories on your system. You can edit the files inside it, rename it, even delete the entire working copy and forget about it.

While your working copy is “just like any other collection of files and directories on your system,” you can edit files at will, but you must tell Subversion about *everything else* that you do. For example, if you want to copy or move an item in a working copy, you should use **svn copy** or **svn move** instead of the copy and move commands provided by your operating system. We'll talk more about them later in this chapter.

Unless you're ready to commit the addition of a new file or directory or changes to existing ones, there's no need to further notify the Subversion server that you've done anything.

What Is This .svn Directory?

The topmost directory of a working copy—and prior to version 1.7, every versioned subdirectory thereof—contains a special administrative subdirectory named `.svn`. Usually, your operating system's directory listing commands won't show this subdirectory, but it is nevertheless an important directory. Whatever you do, don't delete or change anything in the administrative area! Subversion uses that directory and its contents to manage your working copy.

Notice that in the previous pair of examples, Subversion chose to create a working copy in a directory named for the final component of the checkout URL. This occurs only as a convenience to the user when the checkout URL is the only bit of information provided to the **svn checkout** command. Subversion's command-line client gives you additional flexibility, though, allowing you to optionally specify the local directory name that Subversion should use for the working copy it creates. For example:

```
$ svn checkout http://svn.example.com/svn/repo/trunk my-working-copy
A    my-working-copy/README
A    my-working-copy/INSTALL
A    my-working-copy/src/main.c
```

```
A    my-working-copy/src/header.h
…
Checked out revision 8810.
$
```

If the local directory you specify doesn't yet exist, that's okay—**svn checkout** will create it for you.

Basic Work Cycle

Subversion has numerous features, options, bells, and whistles, but on a day-to-day basis, odds are that you will use only a few of them. In this section, we'll run through the most common things that you might find yourself doing with Subversion in the course of a day's work.

The typical work cycle looks like this:

1. *Update your working copy.* This involves the use of the **svn update** command.
2. *Make your changes.* The most common changes that you'll make are edits to the contents of your existing files. But sometimes you need to add, remove, copy and move files and directories—the **svn add**, **svn delete**, **svn copy**, and **svn move** commands handle those sorts of structural changes within the working copy.
3. *Review your changes.* The **svn status** and **svn diff** commands are critical to reviewing the changes you've made in your working copy.
4. *Publish (commit) your changes.* The **svn commit** command transmits your changes to the repository where, if they are accepted, they create the newest versions of all the things you modified. Now others can see your work, too!

Update Your Working Copy

When working on a project that is being modified via multiple working copies, you'll want to update your working copy to receive any changes committed from other working copies since your last update. These might be changes that other members of your project team have made, or they might simply be changes you've made yourself from a different computer. To protect your data, Subversion won't allow you commit new changes to out-of-date files and directories, so it's best to have the latest versions of all your project's files and directories before making new changes of your own.

Use **svn update** to bring your working copy into sync with the latest revision in the repository:

```
$ svn update
Updating '.':
U    foo.c
U    bar.c
Updated to revision 2.
$
```

In this case, it appears that someone checked in modifications to both `foo.c` and `bar.c` since the last time you updated, and Subversion has updated your working copy to include those changes.

When the server sends changes to your working copy via **svn update**, a letter code is displayed next to each item to let you know what actions Subversion performed to bring your working copy up to date. To find out what these letters mean, run **`svn help update`** or see ??? in ???.

Make Your Changes

Now you can get to work and make changes in your working copy. You can make two kinds of changes to your working copy: *file changes* and *tree changes*. You don't need to tell Subversion that you intend to change a file; just make your changes using your text

editor, word processor, graphics program, or whatever tool you would normally use. Subversion automatically detects which files have been changed, and in addition, it handles binary files just as easily as it handles text files—and just as efficiently, too. Tree changes are different, and involve changes to a directory's structure. Such changes include adding and removing files, renaming files or directories, and copying files or directories to new locations. For tree changes, you use Subversion operations to "schedule" files and directories for removal, addition, copying, or moving. These changes may take place immediately in your working copy, but no additions or removals will happen in the repository until you commit them.

Versioning Symbolic Links

On non-Windows platforms, Subversion is able to version files of the special type *symbolic link* (or "symlink"). A symlink is a file that acts as a sort of transparent reference to some other object in the filesystem, allowing programs to read and write to those objects indirectly by performing operations on the symlink itself.

When a symlink is committed into a Subversion repository, Subversion remembers that the file was in fact a symlink, as well as the object to which the symlink "points." When that symlink is checked out to another working copy on a non-Windows system, Subversion reconstructs a real filesystem-level symbolic link from the versioned symlink. But that doesn't in any way limit the usability of working copies on systems such as Windows that do not support symlinks. On such systems, Subversion simply creates a regular text file whose contents are the path to which the original symlink pointed. While that file can't be used as a symlink on a Windows system, it also won't prevent Windows users from performing their other Subversion-related activities.

Here is an overview of the five Subversion subcommands that you'll use most often to make tree changes:

`svn add FOO`

Use this to schedule the file, directory, or symbolic link `FOO` to be added to the repository. When you next commit, `FOO` will become a child of its parent directory. Note that if `FOO` is a directory, everything underneath `FOO` will be scheduled for addition. If you want only to add `FOO` itself, pass the `--depth=empty` option.

`svn delete FOO`

Use this to schedule the file, directory, or symbolic link `FOO` to be deleted from the repository. If `FOO` is a file or link, it is immediately deleted from your working copy. If `FOO` is a directory, it is not deleted, but Subversion schedules it for deletion. When you commit your changes, `FOO` will be entirely removed from your working copy and the repository.[1]

`svn copy FOO BAR`

Create a new item `BAR` as a duplicate of `FOO` and automatically schedule `BAR` for addition. When `BAR` is added to the repository on the next commit, its copy history is recorded (as having originally come from `FOO`). **svn copy** does not create intermediate directories unless you pass the `--parents` option.

`svn move FOO BAR`

This command is exactly the same as running **`svn copy FOO BAR; svn delete FOO`**. That is, `BAR` is scheduled for addition as a copy of `FOO`, and `FOO` is scheduled for removal. **svn move** does not create intermediate directories unless you pass the `--parents` option.

`svn mkdir FOO`

[1] Of course, nothing is ever totally deleted from the repository—just from its `HEAD` revision. You may continue to access the deleted item in previous revisions. Should you desire to resurrect the item so that it is again present in `HEAD`, see ???.

This command is exactly the same as running **`mkdir FOO; svn add FOO`**. That is, a new directory named `FOO` is created and scheduled for addition.

Review Your Changes

Once you've finished making changes, you need to commit them to the repository, but before you do so, it's usually a good idea to take a look at exactly what you've changed. By examining your changes before you commit, you can compose a more accurate *log message* (a human-readable description of the committed changes stored alongside those changes in the repository). You may also discover that you've inadvertently changed a file, and that you need to undo that change before committing. Additionally, this is a good opportunity to review and scrutinize changes before publishing them. You can see an overview of the changes you've made by using the **svn status** command, and you can dig into the details of those changes by using the **svn diff** command.

Look Ma! No Network!

You can use the commands **svn status**, **svn diff**, and **svn revert** without any network access even if your repository *is* across the network. This makes it easy to manage and review your changes-in-progress when you are working offline or are otherwise unable to contact your repository over the network.

Subversion does this by keeping private caches of pristine, unmodified versions of each versioned file inside its working copy administrative area (or prior to version 1.7, potentially multiple administrative areas). This allows Subversion to report—and revert—local modifications to those files *without network access*. This cache (called the *text-base*) also allows Subversion to send the user's local modifications during a commit to the server as a compressed *delta* (or “difference”) against the pristine version. Having this cache is a tremendous benefit—even if you have a fast Internet connection, it's generally much faster to send only a file's changes rather than the whole file to the server.

See an overview of your changes

To get an overview of your changes, use the **svn status** command. You'll probably use **svn status** more than any other Subversion command.

Because the **cvs status** command's output was so noisy, and because **cvs update** not only performs an update, but also reports the status of your local changes, most CVS users have grown accustomed to using **cvs update** to report their changes. In Subversion, the update and status reporting facilities are completely separate. See ??? for more details.

If you run **`svn status`** at the top of your working copy with no additional arguments, it will detect and report all file and tree changes you've made.

```
$ svn status
?       scratch.c
A       stuff/loot
A       stuff/loot/new.c
D       stuff/old.c
M       bar.c
$
```

In its default output mode, **svn status** prints seven columns of characters, followed by several whitespace characters, followed by a file or directory name. The first column tells the status of a file or directory and/or its contents. Some of the most common codes that **svn status** displays are:

`? item`
: The file, directory, or symbolic link `item` is not under version control.

`A item`
: The file, directory, or symbolic link `item` has been scheduled for addition into the repository.

`C item`
: The file `item` is in a state of conflict. That is, changes received from the server during an update overlap with local changes that you have in your working copy (and weren't resolved during the update). You must resolve this conflict before committing your changes to the repository.

`D item`
: The file, directory, or symbolic link `item` has been scheduled for deletion from the repository.

`M item`
: The contents of the file `item` have been modified.

If you pass a specific path to **svn status**, you get information about that item alone:

```
$ svn status stuff/fish.c
D       stuff/fish.c
```

svn status also has a `--verbose` (`-v`) option, which will show you the status of *every* item in your working copy, even if it has not been changed:

```
$ svn status -v
M               44        23    sally     README
                44        30    sally     INSTALL
M               44        20    harry     bar.c
                44        18    ira       stuff
                44        35    harry     stuff/trout.c
D               44        19    ira       stuff/fish.c
                44        21    sally     stuff/things
A                0         ?     ?        stuff/things/bloo.h
                44        36    harry     stuff/things/gloo.c
```

This is the “long form” output of **svn status**. The letters in the first column mean the same as before, but the second column shows the working revision of the item. The third and fourth columns show the revision in which the item last changed, and who changed it.

None of the prior invocations to **svn status** contact the repository—they merely report what is known about the working copy items based on the records stored in the working copy administrative area and on the timestamps and contents of modified files. But sometimes it is useful to see which of the items in your working copy have been modified in the repository since the last time you updated your working copy. For this, **svn status** offers the `--show-updates` (`-u`) option, which contacts the repository and adds information about items that are out of date:

```
$ svn status -u -v
M      *        44        23    sally     README
M               44        20    harry     bar.c
       *        44        35    harry     stuff/trout.c
D               44        19    ira       stuff/fish.c
A                0         ?     ?        stuff/things/bloo.h
Status against revision:   46
```

Notice in the previous example the two asterisks: if you were to run **`svn update`** at this point, you would receive changes to `README` and `trout.c`. This tells you some very useful information—because one of those items is also one that you have locally modified (the file `README`), you'll need to update and get the server's changes for that file before you commit, or the repository will reject your commit for being out of date. We discuss this in more detail later.

svn status can display much more information about the files and directories in your working copy than we've shown here—for an exhaustive description of **svn status** and its output, run **`svn help status`** or see ??? in ???.

Examine the details of your local modifications

Another way to examine your changes is with the **svn diff** command, which displays differences in file content. When you run **`svn diff`** at the top of your working copy with no arguments, Subversion will print the changes you've made to human-readable files in your working copy. It displays those changes in *unified diff* format, a format which describes changes as “hunks” (or “snippets”) of a file's content where each line of text is prefixed with a single-character code: a space, which means the line was unchanged; a minus sign (`-`), which means the line was removed from the file; or a plus sign (`+`), which means the line was added to the file. In the context of **svn diff**, those minus-sign- and plus-sign-prefixed lines show how the lines looked before and after your modifications, respectively.

Here's an example:

```
$ svn diff
Index: bar.c
===================================================================
--- bar.c (revision 3)
+++ bar.c (working copy)
@@ -1,7 +1,12 @@
+#include <sys/types.h>
+#include <sys/stat.h>
+#include <unistd.h>
+
+#include <stdio.h>

 int main(void) {
-  printf("Sixty-four slices of American Cheese...\n");
+  printf("Sixty-five slices of American Cheese...\n");
 return 0;
 }

Index: README
===================================================================
--- README (revision 3)
+++ README (working copy)
@@ -193,3 +193,4 @@
+Note to self:  pick up laundry.

Index: stuff/fish.c
===================================================================
--- stuff/fish.c (revision 1)
+++ stuff/fish.c (working copy)
-Welcome to the file known as 'fish'.
-Information on fish will be here soon.

Index: stuff/things/bloo.h
===================================================================
--- stuff/things/bloo.h (revision 8)
+++ stuff/things/bloo.h (working copy)
+Here is a new file to describe
+things about bloo.
```

The **svn diff** command produces this output by comparing your working files against its pristine text-base. Files scheduled for addition are displayed as files in which every line was added; files scheduled for deletion are displayed as if every line was removed from those files. The output from **svn diff** is somewhat compatible with the **patch** program—more so with the **svn patch** subcommand

introduced in Subversion 1.7. Patch processing commands such as these read and apply *patch files* (or "patches"), which are files that describe differences made to one or more files. Because of this, you can share the changes you've made in your working copy with someone else without first committing those changes by creating a patch file from the redirected output of **svn diff**:

```
$ svn diff > patchfile
$
```

Subversion uses its internal diff engine, which produces unified diff format, by default. If you want diff output in a different format, specify an external diff program using `--diff-cmd` and pass any additional flags that it needs via the `--extensions` (`-x`) option. For example, you might want Subversion to defer its difference calculation and display to the GNU **diff** program, asking that program to print local modifications made to the file `foo.c` in context diff format (another flavor of difference format) while ignoring changes made only to the case of the letters used in the file's contents:

```
$ svn diff --diff-cmd /usr/bin/diff -x "-i" foo.c
…
$
```

Commit Your Changes

Finally! Your edits are finished, you've merged all changes from the server, and you're ready to commit your changes to the repository.

The **svn commit** command sends all of your changes to the repository. When you commit a change, you need to supply a log message describing your change. Your log message will be attached to the new revision you create. If your log message is brief, you may wish to supply it on the command line using the `--message` (`-m`) option:

```
$ svn commit -m "Corrected number of cheese slices."
Sending        sandwich.txt
Transmitting file data .
Committed revision 3.
```

However, if you've been composing your log message in some other text file as you work, you may want to tell Subversion to get the message from that file by passing its filename as the value of the `--file` (`-F`) option:

```
$ svn commit -F logmsg
Sending        sandwich.txt
Transmitting file data .
Committed revision 4.
```

If you fail to specify either the `--message` (`-m`) or `--file` (`-F`) option, Subversion will automatically launch your favorite editor (see the information on `editor-cmd` in ???) for composing a log message.

If you're in your editor writing a commit message and decide that you want to cancel your commit, you can just quit your editor without saving changes. If you've already saved your commit message, simply delete all the text, save again, and then abort:

```
$ svn commit
Waiting for Emacs...Done

Log message unchanged or not specified
(a)bort, (c)ontinue, (e)dit
a
$
```

The repository doesn't know or care whether your changes make any sense as a whole; it checks only to make sure nobody else has changed any of the same files that you did when you weren't looking. If somebody *has* done that, the entire commit will fail with a message informing you that one or more of your files are out of date:

```
$ svn commit -m "Add another rule"
Sending        rules.txt
svn: E155011: Commit failed (details follow):
svn: E155011: File '/home/sally/svn-work/sandwich.txt' is out of date
…
```

(The exact wording of this error message depends on the network protocol and server you're using, but the idea is the same in all cases.)

At this point, you need to run **`svn update`**, deal with any merges or conflicts that result, and attempt your commit again.

That covers the basic work cycle for using Subversion. Subversion offers many other features that you can use to manage your repository and working copy, but most of your day-to-day use of Subversion will involve only the commands that we've discussed so far in this chapter. We will, however, cover a few more commands that you'll use fairly often.

Summary

Now we've covered most of the Subversion client commands. Notable exceptions are those dealing with branching and merging (see ???) and properties (see ???). However, you may want to take a moment to skim through ??? to get an idea of all the different commands that Subversion has—and how you can use them to make your work easier.