COMPUTER:Subversion:LongTutorial

From VirtualRDCWiki
Main Page > Category:Subversion > COMPUTER:Subversion:QuickStartLinux > COMPUTER:Subversion:LongTutorial

This is a general overview of how Subversion is structured, how it supports different workflows, and how both beginners and intermediate users can interface with it.

Contents

1 Basic structure

Each repository is located at a specific URL[1]. These can be local or remote - for the workflow and functionality, this does not matter (note that some of the GUI tools, however, do not work with all of the possible access mechanisms).

1.1 Generic setup

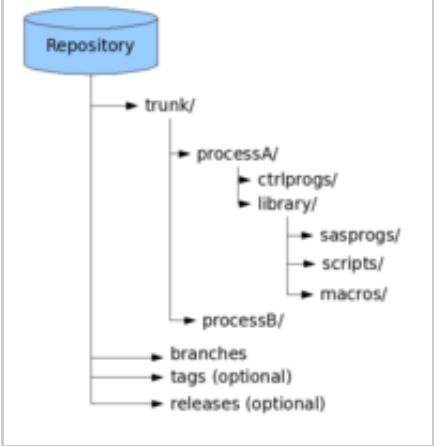
Generic repository setup

The generic setup suggests/recognizes the following root directories:

- trunk - this will be the "clean" code - i.e., code that is committed to the trunk has been previously tested for functionality, and works. (it is possible to relax this constraint, see the LEHD model below).
- branches - in all models, initial work on any code change should occur in branches. Literally, code is branched from the trunk (or another branch, or a release), and from there on, takes on a life of its own, until (select) code gets merged back into the trunk.
- tags and releases - in the generic Subversion book, tags are locked-in versions of the code. In a more sophisticated model, one might distinguish from the frozen code versions more formal 'releases' - code that was released to production, or to a client, etc. Strictly speaking, neither of these are necessary, since one could as easily identify a particular revision (more on that later) along the trunk, with no loss of functionality.

1.2 LEHD-style setup

Each site will have a specific setup - the above setup is only the most general. One setup frequently used at Cornell or LEHD-related sites is the "LEHD structure":



LEHD structure

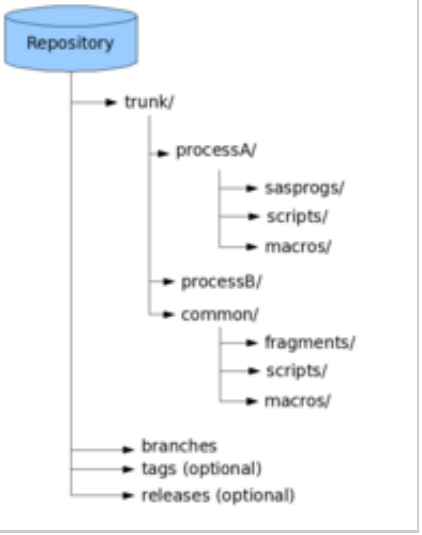
In this structure,

- processes (or products, or filetypes) each have their own sub-tree
- ctrlprogs/ contains setup code and standard parameters that may be subsequently modified by the setup program
- library/ contains all the process-specific code that defines the processing sequence, and may contain
- sasprogs/ the SAS programs that are actually run
- scripts/ any non-SAS scripts that are executed
- macros/ any SAS macros that are called through the SASAUTOS facility
- others - which may include doc/, layouts/, formats/, or whatever suits the code maintainer

Branches then accomodate the entire copy of a process, or even of the trunk, as needed (see Section on branching later).

1.3 Simplified style

A somewhat simplified structure (with a less formal distinction as to setup and runtime programs) may be structured by the following:



Simplified structure

In this structure,

- processes (or products, or filetypes) are again in their own sub-tree
- sasprogs/ the SAS programs that are actually run
- scripts/ any non-SAS scripts that are executed
- macros/ any SAS macros that are called through the SASAUTOS facility
- common/ is used to store procedures common to all {processes/products/filetypes}
- fragments/ are SAS programs (or other code) meant to be included as '%includes' in SAS
- macros/ are SAS macros that are called through the SASAUTOS facility
- scripts/ are scripts that are executed directly (typically, you would want the trunk/common/scripts check-out directory to be included in your PATH on Linux)

2 Local structure on your workstation/workarea

The previous section discussed the layout of the (remote or local, does not matter) Subversion repository. You do not work directly with a repository, rather, you generically walk through a workflow that has the following components:

- check out code into a local copy
- modify code
- update the repository with modified code (check-in)
- (possibly) merge code into common code base (trunk)

We will define several common workflows in the next sections, but we start by defining the local layout. Again, depending on your specific setup, several scenarios might exist.

2.1 Structured environment

In a structured environment, where you run even your alternate code may be "well-defined", because you might be running within a defined system. For instance, you might have the layout in "Structured environment filesystem layout": Under a common directory tree (/programs/production), you have

- *prod/current/* where the actual production(-quality) code resides (it will be run/executed elsewhere),

with

- *processA/* separate directories for each process, replicating the structure previously described in the "LEHD repository"

- *prod/run* is where the programs are actually run (using a template runtime program that references the code in *prod/current*)
  - *processA/* again organized by process (product/filetype/)
- *devX/* development/testing environments, which again contain the same subdirectory structure as the *prod/* area, including independent *current* and *run* structures. There are multiple, independent DEV areas, used by different (groups of) developers.

Structured environment filesystem layout

2.2 Semi-structured environment

Semi-structured environment filesystem layout

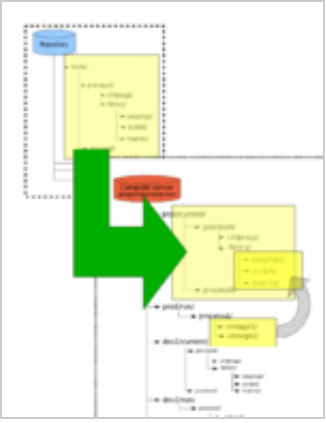
In a slightly less well-defined production environment, only the "production-quality" area might have a well-defined structure mirroring the trunk of the repository ("Semi-structured environment filesystem layout"). Both the repository branches and the local-workstation development areas might have less structure. In this setup, structure is imposed by a central code coordinator copying code from a branch into its proper location in the trunk (see workflow later).

- */programs/PROJECT* (or */rdcprojects/PROJECT/EXTRAPATH*) will house the per-project code
- *current/* houses the most current code extracted from the repository (this might be from trunk/ or from the latest releases/ - see workflow)
- underneath you find again a replica of the structure of the repository trunk.
- *vintages/* (or *runs*) again houses the production-level ("good") runs of the programs
- *branches/* houses separate areas for each developer, who develop and run code in whatever fashion suits them (leaving it up to the poor code coordinator to integrate their code back into the trunk...).

3 Workflow

Several different workflows may exist. We will describe some, including scenarios that may arise in real-life code development and data production. The Subversion book (http://svnbook.red-bean.com/en/1.4/index.html) is always an excellent reference.

3.1 Standard workflow in structured environment



Standard workflow in structured environment

To understand the development workflow, it is useful to first consider the regular "production" workflow. For this, think of the way that you install software on your PC. Typically (unless you are a developer), you will download (or copy from CD) software to your computer, where it is installed and run. Even if you encounter bugs in the software, you don't fix them in the software that is installed on your computer, then send a copy of that software back to the software provider. The same flow of code is implemented in the standard workflow in a structured environment (see "Standard workflow in structured environment"):

- from *repository:trunk/*, code is copied ("checked out") into the *server:prod/current* environment (where we now use the notation "location:path" to distinguish a particular path in the repository from a path on the server). This is typically accomplished with the Subversion command

```
server> svn co repository:trunk /programs/production/prod/current
```

(where the notation `server>` means that the command is executed while logged on to the server (you are never logged on to the repository).

- where it is accessed ("run") or referenced ("linked" or "included") from other locations on the compute server (in the *prod/run* areas), denoted by the hashed arrow (the code "points" to the *prod/current* area, but only reads from there). This linking/including/etc. can be achieved with SAS code similar to this fragment, a copy of which would be located in *prod/run/processA/vintage1* (vintage1=YYYYMMDD_HHMM) and appropriately modified in any other vintage directory:

```
/*=== run/vintage specific parameters ===*/
%let process_id=processA;
%let vintage=YYYYMMDD_HHMM;
%let module_start=1;
%let module_end=5;

/*=== generic config ===*/
%let module_location=/programs/production/prod/current/&process_id./library/sasprogs;
%let macro_location=/programs/production/prod/current/&process_id./library/macros;
%let vintage_location=/programs/production/prod/run/&process_id./&vintage.;

/*=== define where to look for macros ===*/
options sasautos=
(
"!SASROOT/sasautos",\
"/programs/production/prod/current/common/library/macros",\
"&macro_location"
);

/*=== define where to look for formats ===*/
libname f_global "/data/production/prod/current/formats/common";
libname f_proces "/data/production/prod/current/formats/&process_id.";
```

```
options fmtsearch=(f_proces f_global work);

/*=== define where to look for programs to run ===*/
filename sasprogs "&module_location";

/*=== Loop over each of the modules to be run ===*/
%macro moduleloop(start=&start_module,end=&end_module);
 %put Modules being run: &start to &end;
  %do mods=&start %to &end;
   %if &mods < 10 %then %let module=0&mods;
     %else %let module=&mods;
   %put EXECUTING MODULE &module AT %sysfunc(putn(%sysfunc(datetime()),datetime.));
   %include sasprogs("&module._&process_id..sas") / source2;
   %put MODULE &module ENDED AT %sysfunc(putn(%sysfunc(datetime()),datetime.));
  %end;
%mend moduleloop;
%moduleloop
/*=== done ===*/
```

Note that there is no arrow pointing back into the repository: just as in the PC software case, no code is ever copied from the *prod/current* area back into the trunk of the repository.

3.2 Development workflow in structured environment

Development workflow in structured environment, step 1

Now consider that some development needs to be done in *processA* whether it relates to a bug, or a new feature is irrelevant. Let's say that the particular project is tracked by *issue123* (this could be ticket number in Trac).[2]

The process starts by branching a copy from trunk into the branch that we will create for this issue:

```
svn mkdir repository:/branches/issue123 -m 'Creating a branch for issue 1
svn cp repository:/trunk/processA repository:/branches/issue123/ -m 'Bran
```

Explanation of the Subversion syntax:

- svn mkdir (http://svnbook.red-bean.com/en/1.4/svn.ref.svn.c.mkdir.html) creates the directory (with an inline comment)
- svn cp (http://svnbook.red-bean.com/en/1.4/svn.ref.svn.c.copy.html) copies the directory on the repository - a so-called server-side copy.

This step is outlined in "Development workflow in structured environment, step 1".

Now that the branch has been created, we need to check the branch out. We are going to assume that *DEV1* is available (and empty - although that is not critical).

```
server> mkdir -p /programs/production/dev1/current              (this step may not be necessary)
server> svn co repository:/branches/issue123/processA  /programs/production/dev1/current/processA
```

Development goes on, and the developer arrives at a point where she wants to checkpoint her work, possibly to share it with a (remote) colleague. She checks the state of her work back into her branch.

```
server> cd /programs/production/dev1/current/processA
server> svn status (http://svnbook.red-bean.com/en/1.4/svn.ref.svn.c.status.html)       will list c
server> svn add (http://svnbook.red-bean.com/en/1.4/svn.ref.svn.c.add.html)  (list of files)
server> svn delete (http://svnbook.red-bean.com/en/1.4/svn.ref.svn.c.delete.html)  (list of files)
server> svn commit (http://svnbook.red-bean.com/en/1.4/svn.ref.svn.c.commit.html)  -m 'This is my i
```

where the first couple of commands are designed to properly synchronize the repository of changes to the programs.

Development workflow in structured environment, step 2

- By itself, Subversion is not aware of files deleted from the filesystem - you should always use the Subversion command, which will delete the file AND register that change to the repository.
- Also, it is very important to rename files using Subversion, then start modifying them - this will preserve the history of a file despite name changes. The appropriate command is `svn move old-name new-name (http://svnbook.red-bean.com/en/1.4/svn.ref.svn.c.move.html)`
- Finally, she commits all of her changes (this will become a revision (http://svnbook.red-bean.com/en/1.4/svn.basic.in-action.html#svn.basic.in-action.revs)) using a single command

The above is just one example. Consult another example.

3.3 Common Tasks

We have already covered how to check out code from the repository. Here are some additional common tasks.

3.3.1 Accessing a previous version of a file

The whole point of Subversion is to have access to the entire history of changes. So what if you realize that you need the copy of the file from a version one year ago that is no longer there, or at least not in the form that you now realize worked better? It was there in Revision 1234... If you want to copy it into your current working copy, do a simple repository copy (there may not be an exact equivalent in the GUI of your choice):

```
svn cp -r1234 workingfile.sas mycopy/where/I/want/it/workingfile.sas
```

This will be marked both immediately for addition to your branch (when you commit), and more importantly, will retain the fact that it came from the older version. Alternatively, you might just want to look at the file. You can either use a web-based Subversion browser (Trac, ViewCV) or export it - without adding it - to your working directory:

```
svn export -r1234 workingfile.sas
```

3.3.2 Identifying changes to your working copy

Before you commit, you can verify what the changes are. You may be used to the Unix command 'diff', using it as

```
diff oldfile newfile
```

Subversion allows you to do that on your working copy:

```
svn diff newfile
```

and will present you with the differences to the previous version - no need to manually keep around a copy of the old file (in fact, Subversion keeps a 'pristine' version around, behind the scenes or rather, in those hidden '.svn' directories).

3.4 Advanced Tasks

3.4.1 Merging a branch back into the trunk

Once you have completed all your work, you want to merge it back into the trunk (or your Change Control Coordinator may do this, or you may be the CCC). Code versioning systems differ primarily in the way they handle this. Subversion provides you with the tools to do it, but is actually not the strongest of the code versioning systems in this regard - it makes you do a lot of things manually. So let's get started. [3]

3.4.1.1 Safety

Make sure all changes are committed:

```
svn status
```

3.4.1.2 Remember which changes you want to implement

Identify the revision numbers, for instance by running

```
svn log --stop-on-copy
```

or by going to a log page for a (partial branch) in your web-based Subversion browser. Let's say that the branch was created in Revision 927, and is currently at Revision 1428.

3.4.1.3 Change to the trunk

You can either change to a location where the trunk has been checked out, or switch your current working directory to the trunk. Also, let's define a useful variable

```
SVNURL=https://repository/path/
```

which references the Subversion server.

3.4.1.3.1 Separate directory

Use or create a complete checkout of the trunk elsewhere:

```
cd ../elsewhere
svn
```


If pre-existing, make sure the working copy is up-to-date:

```
cd ../elsewhere/productioncode
svn up
```

3.4.1.3.2 Switch the branch to the trunk

```
svn switch $SVNURL/trunk/productioncode
```

This will back out all the changes of your branch, and bring you back to the original state. Don't worry (unless you didn't commit your changes!) - we'll get it all back. Sometimes, this fails. Then use the first approach.

3.4.1.4 Check which changes will occur

Always good to do a dry-run:

```
svn merge --dry-run -r927:1428 $SVNURL/branches/mycode
```

3.4.1.5 Now implement the changes

Just drop the --dry-run part.

3.4.1.6 Then commit the changes

You should

- note where the changes come from (the r927:1428 part)
- a verbose description of the branch where they came from
- a reference to the tickets that are solved (a reference to the milestone might be sufficient)

3.4.1.7 Script all in one place for cut-and-paste

Modify the first line. Sometimes modify the second line

```
process=productioncode; BRANCH=mycode; TICKET=35
svn status
SVNURL=https://repository/path/
svn log --stop-on-copy
```

Now get the numbers, edit the next two lines, then run the next part.

```
RSRC=927; RDEST=1428
svn switch $SVNURL/trunk/$process
svn merge --dry-run -r${RSRC}:${RDEST} $SVNURL/branches/$BRANCH
```

Review, then run the final commit.

```
svn merge -r${RSRC}:${RDEST} $SVNURL/branches/$BRANCH
svn ci -m "Re #${TICKET}. Based on $BRANCH (r${RSRC}:${RDEST}). EDIT ME"
```

If you are paranoid, you can check that the two branches are now identical (if these were the only changes to merge in) by *exporting* both branches:

```
svn export $SVNURL/branches/$BRANCH ${process}-branch
svn export $SVNURL/trunk/$process ${process}-trunk
diff -r ${process}-branch ${process}-trunk| grep -vE '\$Id|\$URL|diff -r|---|[0-9]c[0-9]'
```

The output from the diff command should be empty.

3.4.2 Complex multi-directory branch-develop-merge project

This is taken from a real-life project. LEHD has "processes" that distinguish different functional areas of the code base. A project to seamlessly integrate data-provider-provided "patches" lead to a major re-design of a process that touched many other processes.

3.4.2.1 Setting up the branch

The following was noted in the LEHD-standard review report, and allows for clean tracking of where the code came from (even if in the meantime it had moved on), and what changes were made.

```
1003 SVNURL=https://repository/path/
1004  svn mkdir $SVNURL/branches/readin-fix-methodology -m 'Re #328 and #342: creating a branch'
1005  svn co $SVNURL/branches/readin-fix-methodology
1006  cd readin-fix-methodology/
1007  svn cp $SVNURL/releases/control/1.2.24 control
1008  svn cp $SVNURL/releases/es202/1.1.44 es202
1009  svn cp $SVNURL/releases/admin/1.1.06 admin
1010  svn cp $SVNURL/releases/register/1.1.16 register
1011  svn cp $SVNURL/releases/globals/1.1.40 globals
```

```
1012  svn ci -m 'Re #328 and #342: establishing the baseline processes'

Committed revision 2513.
```

3.4.2.2 Modifying the branches

Now that all modifications have been collected in one place, changes can be made. Note that from a developer perspective, all changes are in a single location

3.5 Test your knowledge

Can you accomplish the following tasks

- using CLI client tools (command line on Linux/Unix)
- using GUI tools (SmartSVN)
- using server-based tools (Trac browser)

where appropriate? In order to do these tests, you can use the test repository at http://repository.vrdc.cornell.edu/public/test (When prompted for a login, use ‘testuser’/'testuser’).

3.5.1 Simple tests

- Create a branch
- Modify and commit an existing file
- Add a new file to the repository
- Write a commit message that links changes back to a SCM/Ticket system

3.5.2 Advanced tests

- Identify all changes to a live checked-out code tree
  - using CLI client tools (command line on Linux/Unix)
  - using GUI tools (SmartSVN)
- Identify all changes between arbitrary locations in the code repository
  - using client tools (command line, GUI)
  - using server-side tools (Trac browser)
- Merge a branch back into the trunk
- Merge a branch back into the trunk, after other, independent merges (scenario: Branch A branches from Rev 1000, Branch B branches from Rev 1500, Branch B merges back into trunk at Rev 1600, now merge Branch A back in.)
- Merge TWO branches into the trunk
- Merge in vendor/external code

4 Footnotes

1. ↑ See Subversion book, Chapter 1 (http://svnbook.red-bean.com/en/1.4/svn.basic.in-action.html)
2. ↑ Most of this is discussed in SVN book (http://svnbook.red-bean.com/en/1.4/svn.branchmerge.using.html) .
3. ↑ This is based on suggestions in the SVN book (http://svnbook.red-bean.com/nightly/en/svn-book.html#svn.branchmerge.copychanges.bestprac) .Reference text

---