

Using CVS

Concurrent Versions System - <http://www.cvshome.org/>

Jeremy Mates <jmates@sial.org>

Notes:

This presentation should be available somewhere on the sial.org website.

The Road Map

1. What is CVS?
2. Anatomy of a Repository
3. Getting the Goodies
4. Working with Stuff
5. Issues & Caveats
6. Resources

Notes:

Blah, blah blah.

What is CVS?

- CVS tracks document evolution in a hierarchical archive.
- Evolved from RCS.
- Leading Open Source Version Control (VC) system.
- Relatively Operating System agnostic.

Notes:

Many VC systems available today, e.g. RCS, SCCS, MS Visual SourceSafe, VOODOO, etc. Some cost money, some don't. Some are platform specific, some not.

CVS is popular, free, and platform agnostic (heavy unix leanings, though).

Why do I need Version Control?

- Who broke `foo.pl`?
- When was `foo.pl` broken?
- Can we revert `foo.pl` to a working version?
- I want to develop my own `foo.pl` ...
- ... and merge my changes back in ...
- There's a bug in a old `foo.pl`, and the new `foo.pl` is still experimental... can we fix the old one for them?
- And more!

Notes:

Anatomy of a Repository

- All CVS documents stored in a repository.
- Just a bunch of files and directories.
- Preferences in the repository, too:

```
$ ls -R /tmp/archive  
/tmp/archive:  
CVSROOT/
```

```
/tmp/archive/CVSROOT:  
Emptydir/      config          editinfo,v     modules,v     taginfo  
checkoutlist   config,v        history         notify         taginfo,v  
checkoutlist,v cvswrappers     loginfo         notify,v      val-tags  
commitinfo     cvswrappers,v  loginfo,v      rcsinfo       verifymsg  
commitinfo,v   editinfo        modules         rcsinfo,v    verifymsg,v
```

Notes:

The funny ,v files are RCS version control format files. They contain the history of the document in a series of patches, as well as various other things to make your life easier (e.g. tags).

Creating a Repository

- A directory and a `cv`s command latter...

```
$ setenv CVSROOT /tmp/archive
$ mkdir -p $CVSROOT
$ cvs init
```

- CVS commands all begin with `cv`s, followed by a sub command to do something, e.g. `init` or `checkout`
- The `init` creates an empty, default repository in the specified path, given by `$CVSROOT`, or the `-d` option:

```
$ mkdir -p /tmp/archive
$ cvs -d /tmp/archive init
```

Notes:

For users of the Bourne-compatible shells (bash, ksh, zsh):

```
$ export CVSROOT=/tmp/archive
```

While I'm on the topic of shell differences, I might as well get a cheap shot in at the csh-based shells:

<http://www.landfield.com/faq/unix-faq/shell/csh-whynot/>

CVS commands have the general format of:

```
cvs [global options] sub command [sub command options] [subcommand-args]
```

Consult the documentation for all the various options and arguments...

Connecting to a CVS Repository

- Use **checkout** (**co**) to obtain a working copy (“sandbox”) of a “module” in the repository:

```
$ cd /tmp
$ cvs -Q checkout CVSROOT
$ ls CVSROOT
CVS/          logininfo
checkoutlist modules
commitinfo   notify
config        rcsinfo
cvswrappers  taginfo
editinfo      verifymsg
```

- Modules are directories in the repository, or more...

Notes:

CVS commands generally have both long and short forms. O’Reilly’s [CVS Pocket Reference](#) has a nice table of the various commands and their short names. For example, the following all create a sandbox for a module:

```
cvs checkout MODULE
cvs co MODULE
cvs get MODULE
```

More ways to get stuff:

- Can also obtain a module across a network:

```
client$ setenv CVS_RSH /usr/bin/ssh
client$ setenv CVSROOT \
    :ext:user@server:/tmp/archive
client$ cd /tmp
client$ cvs checkout CVSROOT
```

- Or via the CVS pserver, run from inetd(8):

```
client$ cvs -d \
    :pserver:user@server:/tmp/archive \
    checkout CVSROOT
```

Notes:

Creating Initial Modules

- Use the CVS `import` command (annoying) to import existing sources.
- Or, checkout the entire repository, and use `cv`s `add` to create new modules as needed.
- Modules are just directories in the repository.
- Modules can also be groups of modules/files if one hacks up the `CVSROOT/modules` file.

Notes:

To manually add a “perl-scripts” module to a repository, cleaning up after ourselves:

```
$ mkdir ~/checkout && cd ~/checkout
$ cvs checkout .
cvs checkout: Updating .
cvs checkout: Updating CVSROOT
U CVSROOT/checkoutlist
U CVSROOT/commitinfo
U CVSROOT/config
U CVSROOT/cvswrappers
U CVSROOT/editinfo
U CVSROOT/logininfo
U CVSROOT/modules
U CVSROOT/notify
U CVSROOT/rcsinfo
U CVSROOT/taginfo
U CVSROOT/verifymsg
$ mkdir perl-scripts
$ cvs add perl-scripts
Directory /tmp/archive/perl-scripts added to the repository
$ cvs release -d .
```

Adding files

- The `cv`s `add` command also does files:

```
$ cd /tmp
$ cvs -Q checkout perl-scripts
$ cd perl-scripts
$ ls
CVS/
$ touch foo.pl
$ cvs add foo.pl
cvs add: scheduling file `foo.pl' for addition
cvs add: use 'cvs commit' to add this file permanently
$ cvs commit -m "Added empty foo perl script." foo.pl
RCS file: /tmp/archive/perl-scripts/foo.pl,v
done
Checking in foo.pl;
/tmp/archive/perl-scripts/foo.pl,v <-- foo.pl
initial revision: 1.1
done
```

Notes:

`cv`s `commit` is actually quite clever; if no filenames are supplied, it will search out and commit whatever files under your current path need committing.

`cv`s `add` is not so clever, and requires explicit file and/or directory names to be added to the repository.

Common CVS commands

- Use `cv`s `commit` to submit changed local files to the central repository.
- Use `cv`s `update` to synchronize local copy (“sandbox”) to repository.
- Use `cv`s `diff` to view differences between file versions.
- Certain utilities support the common `cv`s commands internally, e.g. the emacs VC Mode.

Notes:

There’s a lot more CVS commands out there, but the above three will be used all the time, as part of the update/hack-file/diff/commit routine:

```
$ cvs up
cvs update: Updating .
$ vi foo.pl
$ cvs up
cvs update: Updating .
M foo.pl
$ cvs diff -u foo.pl
(lots of output)
$ cvs commit -m "Added blah print."
cvs commit: Examining .
Checking in foo.pl;
/tmp/archive/perl-scripts/foo.pl,v <-- foo.pl
new revision: 1.2; previous revision: 1.1
done
$ cvs up
cvs update: Updating .
```

The `-u` for `cv`s `diff` produces the best output, and can be defaulted to with a good `~/cvs` file. However, certain operating systems have a default `diff` that does not support the unified output; these should be taken out and shot (e.g. Solaris, at time of writing).

Issues & Caveats

- File & directory structure hard to change (plan well before adding new modules/files).
- Text orientation. Binary files are supported via '`cvsexec add -kb logo.gif`' but there is no "diff" support.
- Line orientation. Moved code is a delete from the source and an add elsewhere.
- Syntax oblivious. White space changes for formatting will be treated as a sweeping change.

Notes:

Multiple Developers

- CVS uses an optimistic *merging model* to allow concurrent development.
- Can use `edit` and `watch` for more restrictive use.
- Communication is the key.

Notes:

As opposed to RCS/SCCS, which use a *locking model*, where a checkout file cannot be edited until the modified file is committed back in.

I have limited experience at present with multiple users in a CVS repository; however, various projects like OpenBSD or those on SourceForge manage huge trees with multiple developers, so it is workable...

Advanced Stuff

- Blank templates can be created to base new development off of:

```
$ touch blank.pl; cvs add -kk blank.pl
$ cvs commit -m "Default perl script template added."
```

- CVS can keep track of “tags” on files, to associate symbolic names (like “release-2001-02-27”) with a group of files.
- Branches off the main line of development can be done with tags, e.g. to apply a bugfix to a past release, or to develop off in an experimental direction.

Notes:

The blank perl script template I use is available at:

<http://www.sial.org/code/perl/scripts/blank.pl>

The `-kk` option prevents CVS keyword expansion from taking place. Keywords are special tags like `Id` or `$Revision$` that generally get expanded out:

```
#!/usr/bin/perl
# $Id: foo.pl,v 1.3 2001/02/27 22:05:43 jmates Exp $

print "blah\n";
```

You can also do spiffy things, like get revision numbers directly into simple scripts. For instance, I use the following in my perl scripts to set a `$VERSION` variable to be the current CVS revision:

```
my $VERSION; ($VERSION = '$Revision$ ') =~ s/[^0-9.]//g;
```

Branches are tricky, and I’ve only done them once or twice to bugfix prior tagged versions. There is a lot of documentation out there on them, between Cederqvist and the CVS Black Book.

Scripting Stuff

- CVS has good support for scripting, through various administrative files found under the CVSROOT module.
- CVS comes with some sample contrib scripts.
- Makefiles can also be inserted into the directory structure to automate various testing, building, and CVS commands:

```
TAGNAME = release  
tag:  
        @cvs tag -cF $(TAGNAME)
```

Notes:

For perl code, I have two shell scripts that will prevent a commit from happening if the code is invalid, or the POD isn't present or invalid:

http://www.sial.org/code/shell/scripts/perl_podchecker.sh

http://www.sial.org/code/shell/scripts/perl_syntax.sh

I also have a (kinda beta) script that regenerates a tarball from my skel directory that I keep under CVS when the appropriate tag is applied to a file:

http://www.sial.org/code/shell/scripts/build_profile.sh

Resources

- CVS Homepage: <http://www.cvshome.org/>
- Documentation central: <http://www.cvshome.org/docs/>
- Open Source Development with CVS:
<http://cvsbook.red-bean.com/>
- CVS Pocket Reference:
<http://www.oreilly.com/catalog/cvspr/>

Notes:

Another notable is the `cvs2cl.pl` script, for easy generation of GNU-style changelogs (or XML output) from the cvs log output:

<http://www.red-bean.com/cvs2cl/>

Be sure to spend some time under the docs area on the cvshome site; it has links to a dearth of information, including the manual and FAQ;

<http://www.cvshome.org/docs/manual/>

<http://www.cvshome.org/docs/infopages.html>

Something broke!

- To revert a file to a previous revision, one must run `log` on the file to figure out which version was the last working one. This may involve committing a currently broken file first:

```
$ cvs log foo.pl | less
```

```
$ cvs diff -r 1.1 -r 1.2 foo.pl
```

```
$ cvs update -j 1.2 -j 1.1 foo.pl
```

```
$ cvs commit -m "Reverted bad 1.2 to 1.1."
```

Notes:

This probably should be organized better, but hey...

A few random commands...

- The `cvstag` command can be used to mark a project that has just shipped:

```
$ cvs tag -fFc foo-project-2001-02-27
```

- To “tag” a modified file with a new revision, clearing the sticky bit that gets set afterwards:

```
$ cvs commit -r 2.0 foo.pl  
$ cvs update -Ad
```

Notes:

Some more random commands...