Git - SVN Crash Course

Welcome to the Git version control system! Here we will briefly introduce you to Git usage based on your current Subversion knowledge. You will need the latest Git installed; There is also a potentially usefultutorial in the Git documentation.

This page is not maintained anymore! The up-to-date version of this tutorial is the GitSvnCrashCourse page at the Git wiki. The copy below might be better edited and nicer to read, but is likely to contain some advices and commands that may not match the current best practices anymore.

- How to Read Me
- Things You Should Know
- Commiting
- Browsing
- Tagging and Branching
- Merging
- Going Remote
- Sharing the Work

If you are just after tracking someone else's project, this get you started quickly:

git clone *url*	svn checkout *url*
git pull	svn update

How to Read Me

In those small tables, at the left we always list the Git commands for the task, while at the right the corresponding Subversion commands you

would use for the job are listed. If you are in hurry, just skimming over them should give you a good idea about the Git usage basics.

Before running any command the first time, it's recommended that you at least quickly skim through its manual page. Many of the commands have very useful and interesting features (that we won't list here) and sometimes there are some extra notes you might want to know. There's a quick usage help available for the Git commands if you pass them the `-h`switch.

Things You Should Know

There are couple important concepts it is good to know when starting with Git. If you are in hurry though, you can skip this section and only get back to it when you get seriously confused; it should be possible to pick up with just using your intuition.

- **Repositories.** With Subversion, for each project there is a single repository at some detached central place where all the history is and which you checkout and commit into. Git works differently, each copy of the project tree (we call that the *working copy*) carries its own repository around (in the `.git` subdirectory in the project tree root). So you can have local and remote branches. You can also have a so-called *bare repository* which is not attached to a working copy; that is useful especially when you want to publish your repository. We will get to that.
- **URL.** In Subversion the *URL* identifies the location of the repository and the path inside the repository, so you organize the layout of the repository and its meaning. Normally you would have `trunk/`, `branches/` and `tags/` directories. In Git the *URL* is just the location of the repository, and it always contains branches and tags. One of the branches is the default (normally named master).
- **Revisions.** Subversion identifies revisions with ids of decimal numbers growing monotonically which are typically small (although they can get quickly to hundreds of thousands for large projects). That is impractical in distributed systems like Git. Git identifies revisions with SHA1 ids, which are long 160-bit numbers written in hexadecimal. It may look scary at first, but in

practice it is not a big hurdle - you can refer to the latest revision by `HEAD`, its parent as `HEAD^` and its parent as `HEAD^^ = HEAD~2` (you can go on adding carrets), cut'n'paste helps a lot and you can write only the few leading digits of a revision - as long as it is unique, Git will guess the rest. (You can do even more advanced stuff with revision specifiers, see the git-rev-parse manpage for details.)

- **Commits.** Each commit has an *author* and a *committer* field, which record who and when *created* the change and who *committed* it (Git is designed to work well with patches coming by mail - in that case, the author and the committer will be different). Git will try to guess your realname and email, but especially with email it is likely to get it wrong. You can check it using `git config -l` and set them with:

  ```
  git config --global user.name "Your Name Comes Here"
  git config --global user.email you@yourdomain.example.com
  ```

- **Colors.** Git can produce colorful output with some commands; since some people hate colors way more than the rest likes them, by default the colors are turned off. If you would like to have colors in your output:

  ```
  git config --global color.diff auto
  git config --global color.status auto
  git config --global color.branch auto
  ```

- **Visualize.** You may find it convenient to watch your repository using the `gitk` repository as you go.

Commiting

For the first introduction, let's make your project tracked by Git and see how we get around to do daily development in it. Let's `cd` to the directory with your project and initialize a brand new Git repository with it:

git init git add . git commit	svnadmin create *repo* svn import *file://repo*

`git init` will initialize the repository, `git add .` will add all the files under the current directory and `git commit` will create the initial import, given that repositories are coupled with working copies.

Now your tree is officially tracked by Git. You can explore the `.git`subdirectory a bit if you want, or don't if you don't care. Do some random changes to your tree now - poke into few files or such. Let's check what we've done:

git diff svn diff | less

That's it. This is one of the more powerful commands. To get a diff with an specific revision and path do:

git diff *rev path* svn diff -r*rev path*

Git embeds special information in the diffs about adds, removals and mode changes:

git apply patch -p0

That will apply the patch while telling Git about and performing those "meta-changes".

There is a more concise representation of changes available:

git status svn status

This will show the concise changes summary as well as list any files that you haven't either ignored or told Git about. In addition, it will also show at the top which branch you are in.

While we are at the status command, over time plenty of the "Untracked files" will get in there, denoting files not tracked by Git. Wait a moment if you want to add them, run `git clean` if you want to get rid of all of them, or add them to the `.gitignore` file if you want to keep them around untracked (works the same as the `svn:ignore` property in SVN).

To restore a file from the last revision:

git checkout *path*	svn revert *path*

You can restore everything or just specified files.

So, just like in SVN, you need to tell Git when you add, move or remove any files:

git add *file*	svn add *file*
git rm *file*	svn rm *file*
git mv *file*	svn mv *file*

You can also recursively add/remove whole directories and so on; Git's cool!

So, it's about time we commit our changes. Big surprise about the command:

git commit -a	svn commit

to commit all the changes or, as with Subversion, you can limit the commit only to specified files and so on. A few words on the commit message: it is *customary* to have a short commit summary as the first line of the message, because various tools listing commits frequently show only the first line of the message. You can specify the commit message using the `-m` parameter as you are used, but you can pass several `-m` arguments and they will create separate paragraphs in the commit message:

If you don't pass any `-m` parameter or pass the `-e` parameter, your favorite `$EDITOR` will get run and you can compose your commit message there, just as with Subversion. In addition, the list of files to be committed is shown.

And as a bonus, if you pass it the `-v` parameter it will show the whole patch being committed in the editor so that you can do a quick last-time review.

By the way, if you screwed up committing, there's not much you can do with Subversion, except using some enigmatic `svnadmin` subcommands. Git does it better - you can amend your latest commit (re-edit the

metadata as well as update the tree) using `git commit --amend`, or toss your latest commit away completely using `git reset HEAD^`, this will not change the working tree.

Browsing

Now that we have committed some stuff, you might want to review your history:

git log	svn log | less
git blame *file*	svn blame *file*

The log command works quite similar in SVN and Git; again, `git log` is quite powerful, please look through its options to see some of the stuff it can do.

The blame command is more powerful as it can detect the movement of lines, even with file copies and renames. But there is a big chance that you probably want to do something different! Usually, when using annotate you are looking for the origin of some piece of code, and the so-called *pickaxe* of Git is much more comfortable tool for that job (`git log -S`*`string`* shows the commits which add or remove any file data matching *string*).

You can see the contents of a file, the listing of a directory or a commit with:

git show *rev*:*path/to/file*	svn cat *url*
git show *rev*:*path/to/directory*	svn list *url*
git show *rev*	svn log -r*rev url*
	svn diff -c*rev url*

Tagging and branching

Subversion marks certain checkpoints in history through copies, the copy is usually placed in a directory named tags. Git tags are much more powerful. The Git tag can have an arbitrary description attached (the first line is special as in the commit case), some people actually store the

whole release announcements in the tag descriptions. The identity of the person who tagged is stored (again following the same rules as identity of the committer). You can tag other objects than commits (but that is conceptually rather low-level operation). And the tag can be cryptographically PGP signed to verify the identity (by Git's nature of working, that signature also confirms the validity of the associated revision, its history and tree). So, let's do it:

git tag -a *name*	svn copy http://example.com/svn/trunk http://example.com/svn/tags/*name*

To list tags and to show the tag message:

git tag -l	svn list http://example.com/svn/*tags/*
git show *tag*	svn log --limit 1 http://example.com/svn/tags/*tag*

Like Subversion, Git can do branches (surprise surprise!). In Subversion, you basically copy your project to a subdirectory. In Git, you tell it, well, to create a branch.

git branch *branch*	svn copy http://example.com/svn/trunk http://example.com/svn/branches/*branch*
git checkout *branch*	svn switch http://example.com/svn/branches/*branch*

The first command creates a branch, the second command switches your tree to a certain branch. You can pass an extra argument to `git branch` to base your new branch on a different revision than the latest one.

You can list your branches conveniently using the aforementioned `git-branch` command without arguments the listing of branches. The current one is denoted by an "*".

git branch	svn list http://example.com/svn/branches/

To move your tree to some older revision, use:

git checkout *rev*	svn update -r *rev*
git checkout *prevbranch*	svn update

or you could create a temporary branch. In Git you can make commits on top of the older revision and use it as another branch.

Merging

Git supports merging between branches much better than Subversion - history of both branches is preserved over the merges and repeated merges of the same branches are supported out-of-the-box. Make sure you are on one of the to-be-merged branches and merge the other one now:

	(assuming the branch was created in revision 20 and you
git	*are inside a working copy of trunk)*
merge *branch*	svn merge -r 20:HEAD
	http://example.com/svn/branches/*branch*

If changes were made on only one of the branches since the last merge, they are simply replayed on your other branch (so-called *fast-forward merge*). If changes were made on both branches, they are merged intelligently (so-called *three-way merge*): if any changes conflicted, `git merge` will report them and let you resolve them, updating the rest of the tree already to the result state; you can `git commit` when you resolve the conflicts. If no changes conflicted, a commit is made automatically with a convenient log message (or you can do `git merge --no-commit` *`branch`*to review the merge result and then do the commit yourself).

Aside from merging, sometimes you want to just pick one commit from a different branch. To apply the changes in revision *rev* and commit them to the current branch use:

git cherry-pick *rev*	svn merge -c *rev url*

Going Remote

So far, we have neglected that Git is a *distributed* version control system. It is time for us to set the record straight - let's grab some stuff from remote sites.

If you are working on someone else's project, you usually want to *clone*its repository instead of starting your own. We've already mentioned that at the top of this document:

git clone *url* svn checkout *url*

Now you have the default branch (normally `master`), but in addition you got all the remote branches and tags. In clone's default setup, the default local branch tracks the *origin* remote, which represents the default branch in the remote repository.

Remote branch, you ask? Well, so far we have worked only with local branches. Remote branches are a mirror image of branches in remote repositories and you don't ever switch to them directly or write to them. Let me repeat - you never mess with remote branches. If you want to switch to a remote branch, you need to create a corresponding local branch which will "track" the remote branch:

git checkout --track -b *branch* origin/*branch* svn switch *url*

You can add more remote branches to a cloned repository, as well as just an initialized one, using `git remote add` *`remote url`*. The command `git remote` lists all the remotes repositories and `git remote show` *`remote`*shows the branches in a remote repository.

Now, how do you get any new changes from a remote repository? You fetch them: `git fetch`. At this point they are in your repository and you can examine them using `git log` *`origin`* (`git log HEAD..`*`origin`* to see just the changes you don't have in your branch), diff them, and obviously, merge them - just do `git merge` *`origin`*. Note that if you don't specify a branch to fetch, it will conveniently default to the tracking remote.

Since you frequently just fetch + merge the tracking remote branch, there is a command to automate that:

git pull svn update

Sharing the Work

Your local repository can be used by others to *pull* changes, but normally you would have a private repository and a public repository. The public repository is where everybody pulls and you... do the opposite? *Push* your changes? Yes! We do `git push remote` which will push all the local branches with a corresponding remote branch - note that this works generally only over SSH (or HTTP but with special webserver setup). It is highly recommended to setup a SSH key and an SSH agent mechanism so that you don't have to type in a password all the time.

One important thing is that you should push only to remote branches that are not currently checked out on the other side (for the same reasons you never switch to a remote branch locally)! Otherwise the working copy at the remote branch will get out of date and confusion will ensue. The best way to avoid that is to push only to remote repositories with no working copy at all - so called *bare* repositories which are commonly used for public access or developers' meeting point - just for exchange of history where a checked out copy would be a waste of space anyway. You can create such a repository. See Setting up a public repository for details.

Git can work with the same workflow as Subversion, with a group of developers using a single repository for exchange of their work. The only change is that their changes aren't submitted automatically but they have to push (however, you can setup a post-commit hook that will push for you every time you commit; that loses the flexibility to fix up a screwed commit, though). The developers must have either an entry in htaccess (for HTTP DAV) or a UNIX account (for SSH). You can restrict their shell account only to Git pushing/fetching by using the `git-shell` login shell.

You can also exchange patches by mail. Git has very good support for patches incoming by mail. You can apply them by feeding mailboxes with patch mails to `git am`. If you want to *send* patches use `git format-patch` and possibly `git send-email`. To maintain a set of patches it is best to use the **StGIT** tool (see the StGIT Crash Course).

If you have any questions or problems which are not obvious from the documentation, please contact us at the **Git mailing list** at git@vger.kernel.org. We hope you enjoy using Git!