git, subversion and branches

Wolfgang Häfelinger
häfelinger IT

February 5, 2010
version 1.1

Introduction

Why am I using git?

git [1], a *distributed* version control system, got a lot of attention recently. The term distributed is a bit oddly chosen. For me, the key concept of git is that it is *decentralized*. A decentralized version control system enables you to work offline, that is you can create create branches and tags, add, change,merge or delete files without contacting a central server. Once you are on-line again, you just synchronize your changes. Thats really great.

However thats not the reason why I started using git. I am using git because I can make local experiments without bothering others. git makes it easy to create branches, switch to another branch and to create, share and merge changes. On top of all that, git is *extremely fast* and reliable.

Git yes, but only local

The problem with a decentralized version control system is its decentralization. If there is a central place for version, then you get well defined and authoritative answers for questions about the latest version, branch and tag names, revisions and so on. Furthermore, if you back up a central repository, you are save. For all those reasons I tried to setup a central repository based on git technologies [2]. This route did not work out for me and so I decided to continue to use subversion as central repository while working locally with git. This works much better than expected thanks to `git-svn`, the git-subversion bridge [3]

git-svn and branches

So my normal work-flow is to checkout the trunk of my subversion project using `git-svn` and to work locally on that trunk perhaps by creating a lot of git branches. When happy I commit back to my upstream trunk. When working with upstream branches however, `git-svn` behaves in unexpected ways.

The main idea of this article is to conduct a couple of scenarios related to upstream branches and to see how `git-svn` behaves.

[1] further information on git is available at http://en.wikipedia.org/wiki/Git_(software)

[2] I tried gitosis.

[3] The git-subversion bridge is named *git-svn*. When being used from the command line, you type `git svn ..` to execute the program `git-svn`. In fact, the program `git` is nothing more than a dispatcher providing a uniform interface for a rather huge bunch of programs.

Conventions

Before diving into `git-svn` its necessary to inform you about some conventions used in this document.

The code examples given assume that there is a subversion repository accessible via some URL `URL`. In its initial state, that repository shall have the following structure and content:


```
.
|
‘-trunk
| ‘-t1.txt
|
‘-branches
  ‘-A
    ‘-a1.txt
```

The hopefully easy to remember convention is, that subversion repositories have upper case roman letters. Files have a combination of lower case letters and digits. Usually the files start with a letter which makes it easy to remember where they belong to.

Often I use the word `git-svn` in explanations while in code examples Im writing something like

```
$ git svn info
```

The thing is that the program `git` is just a kind of dispatcher. It takes the first non-option argument to locate the program to execute. On my macport installation for example, `git svn` runs the program `git-svn` in folder `/opt/-local/git-exec/core`.

Furthermore, in a session snippet like shown next

```
$ git svn --version
git-svn version 1.6.5.3 (svn 1.6.5)
```

is the line starting with character `$` the command you typeexcept `$ ` being the 'prompt. All other lines contain output from the previous command.

Getting started with git-svn

Now, the standard way of interacting with an upstream subversion repository is rather straight forward when using `git-svn`. Clone a subversion project, hack away locally using git for all version management tasks and when happy, run a commit against subversion to commit your changes:


```
$ git svn clone URL/trunk
$ # hack and use git
$ # eventually ..
$ git svn rebase       # optional
$ git svn dcommit
```

Before you commit your changes back to your subversion repository, it is recommended to make sure that you are in sync with your upstream repository. To do so, you run `git-svn rebase` which downloads and applies changes from the subversion repository to your local working branch. Having resolved all changes, run `git-svn dcommit` to push your changes. This may fail of course because your subversion repository may have new changes meanwhile cause it may have taken you some while to resolve all conflicts. It is therefore good practice to stay in sync with your upstream repository as close as possible.

And this is already an example where git shows its strengths. You can easily check how much you are out of sync. The idea is really simple. Just create a branch of your current working branch and try to rebase. This allows you to test what problems you may run into without actually disturbing your current work. Then, when done with testing just throw away your test branch and switch back to your working branch:

```
$ git stash                        # optional (in case
    of uncommitted changes)
$ git checkout -b rebasetest       # create and switch
    to a test branch
$ git svn rebase                   # test re-basing
$ # eventually ..
$ git checkout -f master           # switch to previous
    branch (assuming master)
$ git branch -d rebasetest         # throw away test
    branch
```

How about branches?

This all works and its rather painless. But how about subversion branches? A simple albeit straight forward way is to clone a subversion branch using `git-svn clone` just as shown above. In that case you would use an URL that points to your branch instead of the `trunk`. Notice that I was using `trunk` because that is the usual name for the main development tree. It does not need to be named `trunk`. `git-svn` allows you to clone *any* sub-tree from your subversion repository.

There is nothing particular wrong with this way of working except that it is not *gitish*. When working with git, you have one well defined folder for your project and switching to a branch does not change that location. It just changes the contents of that location.

The rest of this article is about how to work with an upstream subversion repository in a *gitish* way. I will show this based on code examples. Also shown are things which are *not* working. And that is a bit of a problem when using `git-svn` - the behaviour is often rather unexpected.

Initial Cloning

This is the standard way of cloning such a repository

```
$ git svn clone -T trunk -b branches URL
```

This creates a local project (here named `helloworld`) and it contains the most up-to-date version of subversions `trunk`.

```
$ cd helloworld
$ ls
t1.txt
```

Git is aware of the overall structure of my subversion project:

```
$ git branch -a
* master
  remotes/A
  remotes/trunk
```

Experiment 1

What will happen if I add or change a file locally? Answer: It will be added to my upstream `trunk`.

```
$ touch t2.txt
$ git add . && git commit && git svn dcommit
$ svn ls URL/trunk
t1.txt
t2.txt
```


In other words, my git branch `master` has been automatically setup to be linked with `remotes/trunk`. It would be good, if that would be indicated when doing a `git branch` command. However it is not. In order to see the linkage of your current git branch, run a `git-svn info` command.

```
# on branch [master]
$ git svn info
URL: URL/trunk
```

Experiment 2

Add a file directly via subversions import facility to simulate concurrent work on my upstream project:

```
$ mkdir tmp
$ (cd tmp  && touch t3.txt && svn import URL/trunk)
$ rm -rf tmp
```

You can check that `URL/trunk` contains three files by now. How do we sync our local git branch? As already show above, just run a `git-svn rebase`:

```
$ git svn rebase
$ ls
t1.txt  t2.txt  t3.txt
```

Experiment 3

My git branch `master` is now in a state which I would like to branch out in my upstream repository. Lets create a new git branch named `B` then. The

following could have hap-pend then:

a. `B` has been prepared that when running a `dcommit` on that branch, a equivalent named branch gets create in subversion

b. the new branch is a *raw* git branch and has no linkage to subversion at all, i.e. running `dcommit` has no effect on my upstream repository

c. the new branch B is still linked with `URL/trunk`

I guess that everyone has here different expectations what will happen. From a user friendliness perspective I expected that my local branch `B` is automatically connected with an upstream repository branch `B`. I was a bit shocked to learn that Im still connected with the main trunk. In fact the designer of `git-svn` implemented the third alternative. I assume this all happens because command `git branch` does not know about `git-svn`. They are both independently implemented and just operate on *public data structures* in the meta folder `.git`. We need to keep this in mind when working with git:

```
$ git checkout -b B && git branch -a
* B
  master
  remotes/A
  remotes/trunk
$ git svn info
  URL: URL/trunk
```

This does not work, so we remove branch `B` to avoid any confusion.

```
$ git checkout master && git branch -d B && git branch
    -a
* master
  remotes/A
  remotes/trunk
```

Instead we ask `git-svn` to create and populate a upstream subversion branch:

```
$ git svn branch B && svn ls URL/branches/B
t1.txt
t2.txt
t3.txt
```

Notice that the upstream branch got populated with my masters content. In other words, `git-svn branch` creates the upstream branch and then applies all changes in my git branch.

Again, it would seem natural that creating a upstream branch has an impact on my local git branches. This is still not the case. Yes, git knows after this operation about an upstream repository branch named `B` but Im still working on my master branch connected to `trunk`.


```
$ git branch -a
* master
  remotes/A
  remotes/B
  remotes/trunk
$ git svn info
URL: URL/trunk
```

So `git-svn branch B` does the very same as `svn copy branches/B` does. Like in subversion, the current working directory tree is not affected by this operation. If you continue to make changes, you still work on the trunk.

Experiment 4

In the previous experiment I created an upstream branch `B` using `git-svn branch`. Assume now that I want to commit local changes into that branch instead of continuing to work in `trunk`. The previous experiment showed, that my local git branch `master` is still connected with upstream `trunk`. So any `dcommit` I would run would commit into the wrong branch.

Lets try therefore command `git-svn branch` again:

```
$ touch t4.txt
$ git add . && git commit      # t4.txt added to master
$ git svn branch B
branch B already exists
```

The overall hope was that `git-svn` would just *merge* the new change into upstream branch `B`. However this does not work.

The problem is, that my current git branch `master` is not related to upstream branch `B`. Git does not know which changes have already been applied to

upstream B. The way I setup this experiment it is logical that the only thing git needs to do is to apply the very last commit on top of B. However branch master does not keep track of the history of upstream B.

What else can we do then? Create a local git branch and link that branch with your upstream branch.


```
$ git branch --track  b remotes/B
```

What we have by now is a local branch b being aware of B`s history. If we now switch to `b, we look at the contents of upstream B:

```
$ git checkout b && ls
t1.txt
t2.txt
t3.txt
```

We can further check that we are indeed connected with upstream B as promised.

```
$ git branch -a
* b
  master
  remotes/A
  remotes/B
  remotes/trunk
$ git svn info
URL: URL/branches/B
```

Be aware however, that your upstream repository didnt got contacted when populating branch b. What you get are all those changes that are stored in remotes/B. To be in sync with upstream B , run

```
$ git svn rebase
```

This will fetch the latest changes from your upstream branch into remotes/B and in a second step apply those changes to your local upstream-linked branch b.

At this stage we have a local branch b which is in sync with upstream B. There

are also changes made in `master` we wish to apply on upstream `B`. That is the overall subject of this experiment. We are almost there. All that needs to be done is to merge changes done in `master` onto `b` and eventually commit this changes to upstream `B`.

```
$ git merge master
$ git svn dcommit
```


To summarize: When working with an upstream branch, create a local git branch which is linked against the upstream branch. Do so by using `git branch --track`. Then switch to this branch and continue to work as usual.

Experiment 5

Assume that you have created an upstream branch `master`. You might have done it via `git-svn branch` or directly via `svn mkdir`. Then, when you cloned your repository with `git-svn clone`, you get:

```
$ git branch -a
* master
  remotes/master
  remotes/trunk
```

The name of the branch does not really matter. I used `master` just because you get this rather annoying warning messages from git, that there is a name conflict between a remote and a local branch name. Assume now further, that I noticed my mistake and want to remove my faulty upstream branch. Since I *can* create a branch in my upstream repository using `git-svn branch` you might be tempted to run

```
$ git svn branch -d master
branch name required
```

because `git branch -d` is the git way of getting rid of a branch. However, you learn by this experiment that a 1:1 translations of options between `git--branch` and `git-svn branch` is a rather naive way of thinking [4]. So this does not work. Try this instead

[4] I personally regard this discrepancy as a bad design example

```
$ git branch -rd master && git branch -a
* master
  remotes/trunk
```

to get rid of the git branch `remotes/master`. Notice that this operation has no impact on your upstream repository. For unknown reasons, does `git branch -d remotes/master` not work.

Experiment 6

In the previous experiment a upstream branch named `master` has been created by accident. The experiment showed then how to get rid of the local git branch reflecting this upstream branch in situations where this might be appropriate.

Assume now further, that I removed that faulty subversion branch named `master` using plain subversion commands:

```
$ svn delete URL/branches/master
```

If you now compare your previously cloned git repository with that your upstream repository, then you will see some kind of discrepancy:

```
$ svn ls URL/branches
# => no branches
$ git branch -a
* master
  remotes/master
  remotes/trunk
```

You may wonder about a `git-svn` command that would somehow update the local git repository to reflect the changed upstream situation. After having applied that command, I would see any new upstream branches when running `git branch -a` and similar, all *removed* upstream branches would be gone.

What git (and git-svn) does, is to provide me with the projects history. Removing an upstream branch does not mean that that branch is wiped out in the projects history. That branch is still there. All it means is, that starting with a certain project revision, a folder with name `branches/master` does not longer exist. Nevertheless, that branch is still part of subversions history.

Still it makes perfectly sense to have that branch history locally when cloning the repository. git allows me to continue working with that branch if I wish. I could also reanimate that branch in my upstream repository by just using `git-svn branch` if I dare to.

Summary

Working with `git-svn` without upstream branches is a no-brainer. It starts to be a different beast when working with upstream branches. The you need to have a good understanding of gits philosophy, otherwise you will get stuck sooner or later.

I believe that the problems I faced with `git-svn` stems from using CVS for a very long time. When using CVS, I have a local snapshot of my upstream project taken at time T. The history of my project was far away. When working with git, I am working very close with the project history. This is much more than translating a CVS (or subversion) command into the equivalent git command. It requires a different way of thinking.

Colophon

This document got written in Asciidoc markup and translated into DocBook by using the `asciidoc` command. From DocBook it got translated into LaTeXusing `dblatex` and from LaTeXeventually into PDF by using XeTeX.