SVN Integration

Overview

Git allows you to interact not only with other Git repositories, but also with SVN repositories. This means that you can use SmartGit like an SVN client:

- Cloning from an SVN repository is similar to checking out an SVN working copy.
- Pulling from an SVN repository is similar to updating an SVN working copy.
- Pushing to an SVN repository is similar to committing from an SVN working copy to the SVN server.

In addition to the SVN functionality, you can use all (local) Git features like local commits and branching. SmartGit performs all SVN operations transparently, so you will notice only a few points in the program where you need to understand which server VCS you are using.

Compatibility and Incompatibility Modes

SmartGit's SVN integration is available in two modes:

- **Normal Mode** This is the recommended mode of operation. It is used by default when a repository is freshly cloned with SmartGit (not with *git-svn*). All features are supported in this mode. The created repositories are not compatible with *git-svn*.
- **git-svn Compatibility Mode** In the git-svn compatibility mode (or just 'compatibility mode ') SmartGit can work with repositories that were created using the *git-svn* command. In this mode advanced features like *EOLs-*, *ignores-* and *externals-translation* are turned off. The SVN history is processed in a similar way as by *git-svn*.

Ignores (Normal Mode Only)

SmartGit maps `svn:ignore` properties to `.gitignore` files. Unlike the `git svn create-ignore` command SmartGit puts `.gitignore` files under version control. If you modify a `.gitignore` file and pushes the change, the corresponding `svn:ignore` property is changed.

The `.gitignore` syntax is significantly more powerful than the `svn:ignore` syntax. Hence, `svn:ignore` can be mapped losslessly to `.gitignore`, however a `.gitignore` file may contain a pattern that can't be mapped backed to `svn:ignore`. In that case the pattern is not translated.

Adding or removing a recursive pattern in `.gitignore` corresponds to setting or unsetting that pattern on every existing directory in the SVN repository. Conversely, when an SVN revision is fetched (back) into the Git repository, a recursive pattern will be translated to a set of non-recursive patterns, one pattern for each directory.

Example

Let's assume we have the following directories in the SVN repository:

```
A {
  B {}
  C {}
}
```

And we add `.gitignore` with only one line:

```
somefile
```

and push. This will set the `svn:ignore`-property to `somefile` for all directories: *A*, *B*, *C*. After fetching such a revision we have the following `.gitignore` contents (ordering of lines is unimportant):

```
A/somefile
A/B/somefile
A/C/somefile
```

Git doesn't support patterns that contain spaces. Hence, SmartGit replaces all spaces in the `svn:ignore` value with `[!!-~]` during the creation of `.gitignore`. Conversely, all newly added patterns containing `[!!-~]` are converted to `svn:ignore` with spaces at the corresponding places.

EOLs (Normal Mode Only)

When using *git-svn* on Windows, different EOLs on different systems may cause trouble. For instance, if the SVN repository contains a file with `svn:eol-style` set to `CRLF`, its content is stored with CRLF line endings. Moreover, *git-svn* puts the file contents directly into Git blobs without modification. Now, if you have the `core.autocrlf` Git option set to `true`, it may be impossible to get a *clean* working tree, and hence `git svn dcommit` won't work. This happens because while checking whether the working tree is *clean*, Git converts working tree file EOLs to *LF* and compares with the blob contents (which has *CRLF*). On the other hand, setting `core.autocrlf` to `false` causes problems with files that contain *LF* EOLs.

Instead of setting a global option, SmartGit carefully sets the EOL for every file in the SVN repository using its `svn:eol-style` and `svn:mime-type` values. It uses the versioned `.gitattributes` file for this purpose. Its settings have higher priority than the *core.autocrlf*-option, so with SmartGit it doesn't matter what the *core.autocrlf* value is.

Warning

The `.git/info/attributes` file has higher priority than the versioned `.gitattributes` files, so it is strongly recommended to delete the former or leave it empty. Otherwise, this may confuse Git or SmartGit.

By default, a newly added text file (or more precisely, a file that Git thinks is a text file) which is pushed has `svn:eol-style` set to `native` and no `svn:mime-type` property set. A newly added binary file has no properties at all.

One can control individual file properties using the `svneol` Git attribute. The syntax is `svneol=<svn:eol-style value>#<svn:mime-type value>`, so for example

```
*.c svneol=LF#unset
```

means all `*.c` files will have `svn:eol-style=LF` and no `svn:mime-type` set after pushing. Recursive attributes are translated like recursive ignores: Their changes result in changes of properties of all files in the SVN repository.

Externals (Normal Mode Only)

SmartGit maps `svn:externals` properties to its own kind of submodules, that have the same interface as Git submodules.

> **Note**
>
> Only externals pointing to directories are supported, not externals pointing to individual files ('file externals').

SVN submodules are defined in the file `.gitsvnextmodules`, which has the following format:

```
[submodule "path/to/submodule"]
  path = path/to/submodule
  owner = /
  url = https://server/path
  revision = 1234
  branch = trunk
  fetch = trunk:refs/remotes/svn/trunk
  branches = branches/*:refs/remotes/svn/*
  tags = tags/*:refs/remote-tags/svn/*
  remote = svn
  type = dir
```

- **path** specifies the submodule location relative to the working tree root.
- **owner** specifies the SVN directory that has a corresponding svn:externals property. The owner directory should be a parent of the submodule location. If the owner is the root of the parent repository itself, the option should be set to "/".
- **url** specifies the SVN URL to be cloned there (svn:externals syntax can be used here) without a certain branch.
- **revision** specifies the revision to be cloned. Absence of this option or using *HEAD* means the latest available revision.
- **fetch, branches, tags** they all specify the SVN repository layout and have the same meaning as the corresponding *git-svn* options of `.git/config`.
- **branch** specifies the branch to checkout. It should be a path relative to the URL of the *url* option, and it must be consistent with the SVN repository layout. The *empty* branch (if `fetch=:refs/remotes/git-svn`) should be specified using slash `/`.
- **remote** specifies the name of the *svn-git-remote* section of the submodule.
- **type** specifies the type of the submodule (default: `dir`). In practice, this is usually a directory. If svn:externals points to a file, this option should have the value `file`.

Changes in `.gitsvnextmodules` are translated to the SVN repository as changes in `svn:externals` and vice versa.

There are two types of SVN submodules between which you can choose during *submodule initialization*:

- **snapshot submodules** contain exactly one revision of the SVN repository. They are useful in those cases where the external points to a third party library that is not changed as part of the project (parent repository).
- **normal submodules** are completely cloned repositories of the corresponding externals. It's recommended to use them when working in both the parent repository and the submodule repository.

SmartGit shows the repository status in the *Directories* view. If the submodule's current state does not exactly correspond to the state defined by `.gitsvnextmodules` (same URL, revisions, ...), it will show ups as *modified*. In this case, can use **Local|Stage** to update the `.gitsvnextmodules` configuration to the current SVN submodule state or you can use **Remote|Submodule|Reset** to put the submodule back into the state as it is registered in `.gitsvnextmodules`.

Symlinks and Executable Files

Symlink processing and *executable*-bit processing work in the same way as in *git-svn*. SVN uses the `svn:special` property to mark a file as being a *symlink*. Then its content should look like this:

```
link path/to/target
```

Such files are converted to *Git symlinks*. In a similar way, files with `svn:executable` are converted to *Git executable* files and vice versa.

Other SVN properties (Normal Mode Only)

SmartGit maps all other properties which do not have a special meaning in the Git world to *Git attributes*.

Depending on the size of the property value, SmartGit may store the entire property definition (name and value) just as an attribute in `.gitattributes` or it may decide to store the property value in a separate file. In the latter case, `.gitattributes` will contain an attribute `attr` which is just *set*, i.e. has no value, denoting the presence of the property for the corresponding file. The value will be stored in `.gitsvnattributes/path/to/entry/attr` instead, where `path/to/entry` is the same path as in `.gitattributes`, i.e. the working tree path of the file or directory to which the property belongs. The mapping between SVN property name and Git attribute name depends on the property name:

- A property which belong to the *svn:*-namespace and does not yet have a special mapping (as explained in the previous sections) will be mapped to a Git attribute starting with the `svn_`-prefix. This means, that property `svn:keywords` will be mapped to attribute `svn_keywords` and `svn:needs-lock` will be mapped to `svn_needs-lock`.
- For all other properties (custom properties), an SVN property with name `foo` will be mapped to a Git attribute with name `svnc_foo`.

Adding a property

If the property value is a small, one-line text property, you may add it directly to `.gitattributes`.

Example

To add custom property *foo* with value *bar* to file `file.txt`, insert following line into `.gitattributes` (or add the attribute to an already existing line for `/file.txt`):

```
/file.txt svnc_foo=bar
```

If the property value is large, consists of multiple lines or is even of binary content, you have to add the *control* entry to `.gitattributes` and create the file `.gitsvnattributes/path/to/entry/attr` containing the property's value.

Example

To add a binary property *foo* with some binary value to file `file.txt`, insert following line into `.gitattributes` (or add the attribute to an already existing line for `/file.txt`):

```
/file.txt svnc_foo
```

and add file `.gitattributes/file.txt/svnc_foo` with the desired binary property content.

Modifying a property

Depending on the size of the property value, the value will either be stored directly in `.gitattributes`, or in `.gitsvnattributes`, as explained before. To modify the value, locate either of the two places where it is stored and modify the value there. After having committed this change and pushed to SVN, the modified property value will show up in the SVN repository.

Example

To change a small, one-line property value *foo* for `file.txt` which is stored in `.gitattributes` to a larger, multi-line or binary property value, replace the Git attribute value by the Git attribute name itself (i.e. mark it as *set*):

```
/file.txt svnc_foo
```

and add file `.gitattributes/file.txt/svnc_foo` with the desired property content.

Removing a property

Depending on the size of the property value, the value will either be stored directly in `.gitattributes`, or in `.gitsvnattributes`, as explained before. To remove the property, remove the corresponding attribute from `.gitattributes` and remove the corresponding file from `.gitsvnattributes`, if it's present there. After having committed this change and pushed to SVN, the property deletion will show up in the SVN repository.

Tags

Unlike *git-svn*, SmartGit creates Git tags for SVN tags. If an SVN tag was created by a simple directory copying, SmartGit creates a tag that points to the copy-source; otherwise SmartGit creates a tag that points to the corresponding commit of `refs/remote-tags/svn/<tagname>`. Git tags can be sent back to the SVN repository (as SVN tags) by right-clicking the tag in the **Branches** view and invoking **Push**.

> **Note**
>
> Git tags that are actually *objects* on their own (not just simple *refs*) are not supported.

History Processing

Branch Replacements

In *compatibility mode*, SmartGit processes the SVN history like *git-svn* does, with the difference that SmartGit doesn't support the `svk:merge` property. In the case where one SVN branch was replaced, SmartGit and *git-svn* create a *merge-commit*.

In *normal mode*, SmartGit uses its own way of history processing: In case of branch replacements no *merge commit* is created; instead a Git reference `refs/svn-attic/svn/<branch name>/<the latest revision where the branch existed>` is created.

> **Tip**
>
> Though that functionality should be used with care, it is easy to create a branch replacement commit from SmartGit:
>
> - Use **Local|Reset** to reset to some other commit
> - Invoke **Remote|Push**. SmartGit will ask whether the current branch should be replaced.

Merges

Translating Merges from SVN to Git

Completely merged SVN branches correspond to merged Git branches. In particular, for SVN revisions that change `svn:mergeinfo` in such a way that some branch becomes completely merged, SmartGit creates a Git *merge-commit*. For branches which have not been completely merged, no *merge-commit* is created.

Translating Merges from Git to SVN

Pushing Git *merge-commits* results in a corresponding `svn:mergeinfo` modification, denoting that the branch has been completely merged.

Cherry-picks

SmartGit supports translation of two kinds of cherry-pick merges between SVN and Git:

- either done using SmartGit,
- or done using another Git client, without the `--no-commit` option, to make sure that the commit message meta info (`git-svn: ...`) is preserved.

Only cherry-picks of Git commits that correspond to (already pushed) SVN revisions (but not local commits) are supported. Pushing of a cherry-pick commit results in a corresponding `svn:mergeinfo` change.

Branch Creation

SmartGit allows to create SVN branches simply by pushing locally created Git branches. In this case, SmartGit will ask you to configure the branch for pushing.

> **Note**
>
> SmartGit always creates a separate SVN revision when creating a branch, which contains purely the branch creation. This helps to avoid troubles when merging from that branch later.

Anonymous Branches

Anonymous branches show up very often in Git repositories where the default Pull behavior is *merge* instead of *rebase*. Such branches are not mapped back to SVN, as *anonymous SVN branches* are not supported. For instance, the following history:

```
  E-F
 /   \
A-B-C-D-G-H (branch)
```

will be pushed as a linear list of commits: *A,B,C,D,G,H*. *E* and *F* won't be pushed at all.

The Push Process

Pushing a commit consists of 3 phases:

- sending the commit to SVN
- fetching it back
- replacing the existing local commit with the commit being fetched back

Note that not only the local commit is replaced but also all commits and tags that depend on it. For example, if there is a local commit with a Git tag attached to it, after pushing the Git tag will be moved to the commit that has been fetched back from the SVN repository.

The pushing process requires the working tree to be clean to start, and it uses the working tree very actively during the whole process. Hence, it is *STRONGLY RECOMMENDED* not to make any changes in the working tree during the pushing process, otherwise these changes may become discarded.

Sometimes it is impossible to replace the existing local commit with the commit being fetched back, because other commits (from other users) might have been fetched back as well, containing changes that conflict with the remaining local commits. In this case, SmartGit leaves the working tree clean and asks you whether to resolve the problem. The easiest way to do so is to press Pull with the **Rebase** option turned on, which will start the rebase process.

Example
The last repository revision is *r10*. There are 2 local commits *A* and *B* that will be pushed. First, *A* is sent, resulting in revision *r12*, because in the meantime someone else had committed *r11*. Now, *r11* and *r12* (corresponding to local commit *A*) are fetched back. Let's assume that *r11* and local commit *B* contain changes for the same file in the same line. Hence, replacing *A* by the fetched-back commits *r11* and *r12* won't work, because the changes of *B* are conflicting now and can't be applied on top of *r12*.

SVN Support Configuration

SVN URL and SVN Layout Specification

In *compatibility mode*, `.git/config` is used for the specification of the SVN URL and the SVN repository layout. In *normal mode*, SmartGit uses the file `.git/svn/.svngit/svngitkit.config` for this purpose.

In *compatibility mode*, the SVN URL and SVN layout are specified in the `svn-remote` section. In *normal mode*, the corresponding section is called `svn-git-remote`.

The section generally looks like this:

```
[svn-git-remote "svn"]
                url = https://server/path
                rewriteRoot = https://anotherserver/path
                fetch = trunk:refs/remotes/svn/trunk
                branches = branches/*:refs/remotes/svn/*
                additional-branches = path/*:refs/remotes/*;another/path:
refs/remotes/another/branch
                tags = tags/*:refs/remote-tags/svn/*
```

- **url** specifies the physical SVN URL with which to connect to the SVN repository.
- **rewriteRoot** specifies the URL to be used in the Git commit messages of fetched commits. If this option is omitted, it is assumed to be the same as the value of the *url* option. The *rewriteRoot* option is useful for continuing working with the repository after the original SVN URL has been changed (in this case *rewriteRoot* should be changed to the old SVN URL value).
- **fetch, branches, additional-branches, tags** specify pairs consisting of an SVN path and a Git reference for various interesting paths in the SVN repository. All paths beyond these won't be considered by SmartGit. There's practically no difference between these options. Options *fetch*, *branches*, *tags* are supported by `git-svn` and allow only 1 pair. Option *additional-branches* is only supported by SmartGit and allows an arbitrary number of `;`-separated pairs. Option *fetch* for *compatibility mode* defines the branch to be checked out and configured as tracked after fetch. SmartGit doesn't support `git-svn` patterns in the config and only allows the usage of asterisks (`*`). The number of asterisks in the SVN path and Git reference pattern must be equal. No patterns except the *fetch* pattern must intersect.

Translation Options

SmartGit keeps all translation options in the `core` section of the file `.git/svn/.svngit/svngitkit.config`.

The section looks like this:

```
[core]
                        processExternals = true
                        processIgnores = true
                        processEols = true
                        processTags = true
                        processOtherProperties = true
                        gitSvnAttributesThreshold = 32
```

These boolean options specify whether SmartGit should enable special handling of `svn:externals`, `svn:eol-style`/`svn:mime-type`, `svn:ignore`, SVN tags and all other SVN properties, as explained above. The `gitSvnAttributesThreshold` specifies the maximum length of a Git attribute value, which may be stored in `.gitattributes`. If the length of the attribute value is larger, it will be stored in the `.gitsvnattributes` directory structure, as explained in Other SVN properties (Normal Mode Only).

> **Warning**
>
> These options are set once before the first fetch and must not be changed afterwards.

In *nomal mode*, all these options are set to `true` by default except when SmartGit detects that the `.gitattributes` file has become too large (in that case *processEols* and *processOtherProperties* is set to `false`). In *compatibility mode*, all these options are set to `false`.

Tracking Configuration

SmartGit's SVN support has a tracking configuration similar to Git's. If some local branch (say, `refs/heads/branch`) tracks some remote branch (`refs/remotes/svn/branch`), then:

- it is possible to push the local branch, and doing so will result in the corresponding SVN branch modification according to the repository layout. If the local branch is not configured as tracking branch of some remote branch, it won't be pushed;
- while fetching, SmartGit proposes to rebase the local tracking branch onto the tracked branch after a Pull if the corresponding option is selected.

Warning

If some branch contains a merge-commit that has a merge-parent that doesn't belong to any tracking branch, `svn:mergeinfo` won't be modified when pushing such a branch.

SmartGit uses the `branch` sections of the `.git/svn/.svngit/svngitkit.config` file for tracking configuration.

The section looks like this:

```
[branch "master"]
                        tracks = refs/remotes/svn/trunk
                        remote = svn
```

The name of the section is the local branch name.

- **tracks** specifies the remote tracked branch
- **remote** specifies the remote section name with the SVN URL and SVN repository layout

Tweaking the configuration

When cloning a repository with **Just initialize clone (expert mode)** option, SmartGit will only create a repository stub which will be reported as *in complete*. This allows you to tweak the *Translation options* and the *Tracking Configuration* by changing the `.git/svn/.svngit/svngitkit.config` file accordingly. After you have done so, you will have to restart the clone by pulling your repository.

After the clone has been initialized, only `[svn-git-remote]` can be tweaked. `[core]` options can't be changed anymore at this stage, instead they have to be customized using system properties before starting SmartGit. Changing the `[core]` configuration by system properties, will not only affect SmartGit's processing but also the Git repository configuration:

- If you set `processEols=false`, then `core.autocrlf` is set to `true`: the idea behind this logic is that if EOLs are controlled by `.gitattributes` settings (`processEols=true`), then `core.autocrlf` should be `false` in order not to interfere with `.gitattributes`. If `processEols=false` then there will be no `.gitattributes` which (usually) performs CRLF-to-LF canonicalization in the repository (what usually is desirable). `core.autocrlf=true` has proven to be the lesser evil in this case.

NTLM authentication

If your server supports NTLM authentication and SmartGit fails to connect to your server with authentication-related errors like `401 Authorization Required`, you should try to force SmartGit to use *Basic* authentication first by setting following system property:

```
svnkit.http.methods=Basic,NTLM
```

SVN authors mapping

SmartGit does not support Git config `svn.authorsfile`, but provides a similar mechanism via system property `smartgit.svn.authorsfile` : the specified file is respected when translating SVN revisions to Git commits. For the opposite direction it's the SVN server which chooses the revision author (usually the same as the credentials username). The file format is the same as git-svn's authors file (for `--authors-file` option). File encoding is UTF-8 (since version 17.1).

Example

On Windows, if your authors-file is located at `c:\temp\svn.authors`, set the system property to:

```
smartgit.svn.authorsfile=c:/temp/svn.authors
```

System properties affecting the SVN integration

Following system properties can used to customize the SVN integration:

smartgit.clone.svnAllowed

Set to `false` to disable the possibility to clone SVN repositories.

smartgit.defaultConnectionLogging

Set to `true` to have SVN connection logging enabled. This will create a connection.log in SmartGit's settings directory which will be helpful for error diagnosis.

smartgit.svn.gitAttributesSizeThreshold

For certain SVN repositories, the .gitattributes file may become very large, which would slow down various (Smart)Git operations. For that reason, the mapping of svn:eol-style and svn:mime-type will be disabled if the size of the .gitattributes file exceeds the threshold specified by the specified value (in bytes). For more information, see http://www.syntevo.com/smartgit/documentation.html?page=concepts-svn To have the .gitattributes mapping always enabled, you may set the threshold to a large value.

smartgit.submoduleRecurseInUnchangedSvn

Set to `true` to enable recursion into unchanged SVN submodules when a submodule update is performed. This may be useful if you always want to fetch the latest revisions from an SVN repository even if the svn:external which is mapped to .gitsvnextmodules does require fetching these revisions.

smartgit.svn.glueFeature

Set to `false` to prevent glueing of multiple revisions together when performing an SVN clone. Glueing revisions together improves performance and usually has no negative side-effects. Sometimes, for large repositories, it may result in out-of-memory errors, though.

smartgit.svn.scanSubmodulesForNonSvnParents

Set to `true` to scan (refresh) SVN sub-modules for non-SVN parent repositories.

> This option not officially supported and may be removed in future SmartGit versions.

smartgit.svn.defaultCommitMessage

Use this option to change the default SVN "commit message" when e.g. pushing a tag. There are a couple of variables, which will be replaced by proper strings: `{Action}, {Actiond}, {action}, {actiond}, {target}, {source}, {sourceRevision}`.

Example

Some example definitions:

```
{Actiond} {target}{ from {source}{:{sourceRevision}}}
{target} {action} { from {source}{:{sourceRevision}}}
```

Known Limitations

There are following notable limitations of SmartGit's SVN integration,

File externals not supported

File externals are currently not supported. There are following possible workarounds:

- create an unversioned symlink locally without adding it to the repository
- replace the file with a symlink in the repository (won't work on Windows)
- use directory externals (they are supported by SmartGit)

Further Limitations

- Empty directories can't be managed by Git and won't be available
- File locks are not supported
- Sparse check-outs are not supported
- Explicit copy and move operations are not possible; Git recognizes them automatically
- The svk:merge property is not supported