



# Git and Other Systems

**T**he world isn't perfect. Usually, you can't immediately switch every project you come in contact with to Git. Sometimes you're stuck on a project using another VCS, and many times that system is Subversion. You'll spend the first part of this chapter learning about `git svn`, the bidirectional Subversion gateway tool in Git.

At some point, you may want to convert your existing project to Git. The second part of this chapter covers how to migrate your project into Git: first from Subversion, then from Perforce, and finally via a custom import script for a nonstandard importing case.

## Git and Subversion

Currently, the majority of open source development projects and a large number of corporate projects use Subversion to manage their source code. It's the most popular open source VCS and has been around for nearly a decade. It's also very similar in many ways to CVS, which was the big boy of the source-control world before that.

One of Git's great features is a bidirectional bridge to Subversion called `git svn`. This tool allows you to use Git as a valid client to a Subversion server, so you can use all the local features of Git and then push to a Subversion server as if you were using Subversion locally. This means you can do local branching and merging, use the staging area, use rebasing and cherry-picking, and so on, while your collaborators continue to work in their dark and ancient ways. It's a good way to sneak Git into the corporate environment and help your fellow developers become more efficient while you lobby to get the infrastructure changed to support Git fully. The Subversion bridge is the gateway drug to the DVCS world.

### `git svn`

The base command in Git for all the Subversion bridging commands is `git svn`. You preface everything with that. It takes quite a few commands, so you'll learn about the common ones while going through a few small workflows.

It's important to note that when you're using `git svn`, you're interacting with Subversion, which is a system that is far less sophisticated than Git. Although you can easily do local branching and merging, it's generally best to keep your history as linear as possible by rebasing your work and avoiding doing things like simultaneously interacting with a Git remote repository.

Don't rewrite your history and try to push again, and don't push to a parallel Git repository to collaborate with fellow Git developers at the same time. Subversion can have only

a single linear history, and confusing it is very easy. If you're working with a team, and some are using SVN and others are using Git, make sure everyone is using the SVN server to collaborate—doing so will make your life easier.

## Setting Up

To demonstrate this functionality, you need a typical SVN repository that you have write access to. If you want to copy these examples, you'll have to make a writable copy of my test repository. In order to do that easily, you can use a tool called `svnsync` that comes with more recent versions of Subversion—it should be distributed with at least 1.4. For these tests, I created a new Subversion repository on Google code that was a partial copy of the `protobuf` project, which is a tool that encodes structured data for network transmission.

To follow along, you first need to create a new local Subversion repository:

```
$ mkdir /tmp/test-svn
$ svnadmin create /tmp/test-svn
```

Then, enable all users to change `revprops`—the easy way is to add a `pre-revprop-change` script that always exits 0:

```
$ cat /tmp/test-svn/hooks/pre-revprop-change
#!/bin/sh
exit 0;
$ chmod +x /tmp/test-svn/hooks/pre-revprop-change
```

You can now sync this project to your local machine by calling `svnsync init` with the `to` and `from` repositories.

```
$ svnsync init file:///tmp/test-svn http://progit-example.googlecode.com/svn/
```

This sets up the properties to run the sync. You can then clone the code by running

```
$ svnsync sync file:///tmp/test-svn
Committed revision 1.
Copied properties for revision 1.
Committed revision 2.
Copied properties for revision 2.
Committed revision 3.
...
```

Although this operation may take only a few minutes, if you try to copy the original repository to another remote repository instead of a local one, the process will take nearly an hour, even though there are fewer than 100 commits. Subversion has to clone one revision at a time and then push it back into another repository—it's ridiculously inefficient, but it's the only easy way to do this.

## Getting Started

Now that you have a Subversion repository to which you have write access, you can go through a typical workflow. You'll start with the `git svn clone` command, which imports an entire Subversion repository into a local Git repository. Remember that if you're importing from

a real hosted Subversion repository, you should replace the `file:///tmp/test-svn` here with the URL of your Subversion repository:

```
$ git svn clone file:///tmp/test-svn -T trunk -b branches -t tags
Initialized empty Git repository in /Users/schacon/projects/testsvnsync/svn/.git/
r1 = b4e387bc68740b5af56c2a5faf4003ae42bd135c (trunk)
  A   m4/acx_pthread.m4
  A   m4/stl_hash.m4
...
r75 = d1957f3b307922124eec6314e15bcda59e3d9610 (trunk)
Found possible branch point: file:///tmp/test-svn/trunk => \
  file:///tmp/test-svn /branches/my-calc-branch, 75
Found branch parent: (my-calc-branch) d1957f3b307922124eec6314e15bcda59e3d9610
Following parent with do_switch
Successfully followed parent
r76 = 8624824ecc0badd73f40ea2f01fce51894189b01 (my-calc-branch)
Checked out HEAD:
  file:///tmp/test-svn/branches/my-calc-branch r76
```

This runs the equivalent of two commands—`git svn init` followed by `git svn fetch`—on the URL you provide. This can take a while. The test project has only about 75 commits and the codebase isn't that big, so it takes just a few minutes. However, Git has to check out each version, one at a time, and commit it individually. For a project with hundreds or thousands of commits, this can literally take hours or even days to finish.

The `-T trunk -b branches -t tags` part tells Git that this Subversion repository follows the basic branching and tagging conventions. If you name your trunk, branches, or tags differently, you can change these options. Because this is so common, you can replace this entire part with `-s`, which means *standard layout* and implies all those options. The following command is equivalent:

```
$ git svn clone file:///tmp/test-svn -s
```

At this point, you should have a valid Git repository that has imported your branches and tags:

```
$ git branch -a
* master
  my-calc-branch
  tags/2.0.2
  tags/release-2.0.1
  tags/release-2.0.2
  tags/release-2.0.2rc1
  trunk
```

It's important to note how this tool namespaces your remote references differently. When you're cloning a normal Git repository, you get all the branches on that remote server available locally as something like `origin/[branch]`, namespaced by the name of the remote. However, `git svn` assumes that you won't have multiple remotes and saves all its references to points on the remote server with no namespacing. You can use the Git plumbing command `show-ref` to look at all your full reference names:

```
$ git show-ref
1cbd4904d9982f386d87f88fce1c24ad7c0f0471 refs/heads/master
aee1ecc26318164f355a883f5d99cff0c852d3c4 refs/remotes/my-calc-branch
03d09b0e2aad427e34a6d50ff147128e76c0e0f5 refs/remotes/tags/2.0.2
50d02cc0adc9da4319eeba0900430ba219b9c376 refs/remotes/tags/release-2.0.1
4caaa711a50c77879a91b8b90380060f672745cb refs/remotes/tags/release-2.0.2
1c4cb508144c513fff1214c3488abe66dcb92916f refs/remotes/tags/release-2.0.2rc1
1cbd4904d9982f386d87f88fce1c24ad7c0f0471 refs/remotes/trunk
```

A normal Git repository looks more like this:

```
$ git show-ref
83e38c7a0af325a9722f2fdc56b10188806d83a1 refs/heads/master
3e15e38c198baac84223acfc6224bb8b99ff2281 refs/remotes/gitserver/master
0a30dd3b0c795b80212ae723640d4e5d48cabdff refs/remotes/origin/master
25812380387fdd55f916652be4881c6f11600d6f refs/remotes/origin/testing
```

You have two remote servers: one named `gitserver` with a `master` branch; and another named `origin` with two branches, `master` and `testing`.

Notice how in the example of remote references imported from `git svn`, tags are added as remote branches, not as real Git tags. Your Subversion import looks like it has a remote named tags with branches under it.

## Committing Back to Subversion

Now that you have a working repository, you can do some work on the project and push your commits back upstream, using Git effectively as a SVN client. If you edit one of the files and commit it, you have a commit that exists in Git locally that doesn't exist on the Subversion server:

```
$ git commit -am 'Adding git-svn instructions to the README'
[master 97031e5] Adding git-svn instructions to the README
 1 files changed, 1 insertions(+), 1 deletions(-)
```

Next, you need to push your change upstream. Notice how this changes the way you work with Subversion—you can do several commits offline and then push them all at once to the Subversion server. To push to a Subversion server, you run the `git svn dcommit` command:

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
    M       README.txt
Committed r79
    M       README.txt
r79 = 938b1a547c2cc92033b74d32030e86468294a5c8 (trunk)
No changes between current HEAD and refs/remotes/trunk
Resetting to the latest refs/remotes/trunk
```

This takes all the commits you've made on top of the Subversion server code, does a Subversion commit for each, and then rewrites your local Git commit to include a unique identifier. This is important because it means that all the SHA-1 checksums for your commits change. Partly for this reason, working with Git-based remote versions of your projects

concurrently with a Subversion server isn't a good idea. If you look at the last commit, you can see the new `git-svn-id` that was added:

```
$ git log -1
commit 938b1a547c2cc92033b74d32030e86468294a5c8
Author: schacon <schacon@4c93b258-373f-11de-be05-5f7a86268029>
Date: Sat May 2 22:06:44 2009 +0000
```

Adding `git-svn` instructions to the README

```
git-svn-id: file:///tmp/test-svn/trunk@79 4c93b258-373f-11de-be05-5f7a86268029
```

Notice that the SHA checksum that originally started with `97031e5` when you committed now begins with `938b1a5`. If you want to push to both a Git server and a Subversion server, you have to push (`dcommit`) to the Subversion server first, because that action changes your commit data.

## Pulling in New Changes

If you're working with other developers, then at some point one of you will push, and then the other one will try to push a change that conflicts. That change will be rejected until you merge in their work. In `git svn`, it looks like this:

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
Merge conflict during commit: Your file or directory 'README.txt' is probably \
out-of-date: resource out of date; try updating at /Users/schacon/libexec/git-\
core/git-svn line 482
```

To resolve this situation, you can run `git svn rebase`, which pulls down any changes on the server that you don't have yet and rebases any work you have on top of what is on the server:

```
$ git svn rebase
M README.txt
r80 = ff829ab914e8775c7c025d741beb3d523ee30bc4 (trunk)
First, rewinding head to replay your work on top of it...
Applying: first user change
```

Now, all your work is on top of what is on the Subversion server, so you can successfully `dcommit`:

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
M README.txt
Committed r81
M README.txt
r81 = 456cbe6337abe49154db70106d1836bc1332deed (trunk)
No changes between current HEAD and refs/remotes/trunk
Resetting to the latest refs/remotes/trunk
```

It's important to remember that unlike Git, which requires you to merge upstream work you don't yet have locally before you can push, `git svn` makes you do that only if the changes conflict. If someone else pushes a change to one file and then you push a change to another file, your `dcommit` will work fine:

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
    M    configure.ac
Committed r84
    M    autogen.sh
r83 = 8aa54a74d452f82eee10076ab2584c1fc424853b (trunk)
    M    configure.ac
r84 = cdbac939211ccb18aa744e581e46563af5d962d0 (trunk)
W: d2f23b80f67aaaa1f6f5aaef48fce3263ac71a92 and refs/remotes/trunk differ, \
  using rebase:
:100755 100755 efa5a59965fbbb5b2b0a12890f1b351bb5493c18 \
  015e4c98c482f0fa71e4d5434338014530b37fa6 M    autogen.sh
First, rewinding head to replay your work on top of it...
Nothing to do.
```

This is important to remember, because the outcome is a project state that didn't exist on either of your computers when you pushed. If the changes are incompatible but don't conflict, you may get issues that are difficult to diagnose. This is different than using a Git server—in Git, you can fully test the state on your client system before publishing it, whereas in SVN, you can't ever be certain that the states immediately before commit and after commit are identical.

You should also run this command to pull in changes from the Subversion server, even if you're not ready to commit yourself. You can run `git svn fetch` to grab the new data, but `git svn rebase` does the fetch and then updates your local commits:

```
$ git svn rebase
    M    generate_descriptor_proto.sh
r82 = bd16df9173e424c6f52c337ab6efa7f7643282f1 (trunk)
First, rewinding head to replay your work on top of it...
Fast-forwarded master to refs/remotes/trunk.
```

Running `git svn rebase` every once in a while makes sure your code is always up to date. You need to be sure your working directory is clean when you run this, though. If you have local changes, you must either stash your work or temporarily commit it before running `git svn rebase`—otherwise, the command will stop if it sees that the rebase will result in a merge conflict.

## Git Branching Issues

When you've become comfortable with a Git workflow, you'll likely create topic branches, do work on them, and then merge them in. If you're pushing to a Subversion server via `git svn`, you may want to rebase your work onto a single branch each time instead of merging branches together. The reason to prefer rebasing is that Subversion has a linear history and doesn't deal with merges like Git does, so `git svn` follows only the first parent when converting the snapshots into Subversion commits.

Suppose your history looks like the following: you created an experiment branch, did two commits, and then merged them back into master. When you dcommit, you see output like this:

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
    M      CHANGES.txt
Committed r85
    M      CHANGES.txt
r85 = 4bfebeec434d156c36f2bcd18f4e3d97dc3269a2 (trunk)
No changes between current HEAD and refs/remotes/trunk
Resetting to the latest refs/remotes/trunk
COPYING.txt: locally modified
INSTALL.txt: locally modified
    M      COPYING.txt
    M      INSTALL.txt
Committed r86
    M      INSTALL.txt
    M      COPYING.txt
r86 = 2647f6b86ccfcaad4ec58c520e369ec81f7c283c (trunk)
No changes between current HEAD and refs/remotes/trunk
Resetting to the latest refs/remotes/trunk
```

Running dcommit on a branch with merged history works fine, except that when you look at your Git project history, it hasn't rewritten either of the commits you made on the experiment branch—instead, all those changes appear in the SVN version of the single merge commit.

When someone else clones that work, all they see is the merge commit with all the work squashed into it; they don't see the commit data about where it came from or when it was committed.

## Subversion Branching

Branching in Subversion isn't the same as branching in Git; if you can avoid using it much, that's probably best. However, you can create and commit to branches in Subversion using git svn.

### Creating a New SVN Branch

To create a new branch in Subversion, you run `git svn branch [branchname]`:

```
$ git svn branch opera
Copying file:///tmp/test-svn/trunk at r87 to file:///tmp/test-svn/branches/opera...
Found possible branch point: file:///tmp/test-svn/trunk => \
    file:///tmp/test-svn/branches/opera, 87
Found branch parent: (opera) 1f6bfe471083cbca06ac8d4176f7ad4de0d62e5f
Following parent with do_switch
Successfully followed parent
r89 = 9b6fe0b90c5c9adf9165f700897518dbc54a7cbf (opera)
```

This does the equivalent of the `svn copy trunk branches/opera` command in Subversion and operates on the Subversion server. It's important to note that it doesn't check you out into that branch; if you commit at this point, that commit will go to trunk on the server, not opera.

## Switching Active Branches

Git figures out what branch your `dcommit`s go to by looking for the tip of any of your Subversion branches in your history—you should have only one, and it should be the last one with a `git-svn-id` in your current branch history.

If you want to work on more than one branch simultaneously, you can set up local branches to `dcommit` to specific Subversion branches by starting them at the imported Subversion commit for that branch. If you want an opera branch that you can work on separately, you can run

```
$ git branch opera remotes/opera
```

Now, if you want to merge your opera branch into trunk (your master branch), you can do so with a normal `git merge`. But you need to provide a descriptive commit message (via `-m`), or the merge will say `Merge branch opera` instead of something useful.

Remember that although you're using `git merge` to do this operation, and the merge likely will be much easier than it would be in Subversion (because Git will automatically detect the appropriate merge base for you), this isn't a normal Git merge commit. You have to push this data back to a Subversion server that can't handle a commit that tracks more than one parent; so, after you push it up, it will look like a single commit that squashed in all the work of another branch under a single commit. After you merge one branch into another, you can't easily go back and continue working on that branch, as you normally can in Git. The `dcommit` command that you run erases any information that says what branch was merged in, so subsequent merge-base calculations will be wrong—the `dcommit` makes your `git merge` result look like you ran `git merge --squash`. Unfortunately, there's no good way to avoid this situation—Subversion can't store this information, so you'll always be crippled by its limitations while you're using it as your server. To avoid issues, you should delete the local branch (in this case, opera) after you merge it into trunk.

## Subversion Commands

The `git svn` toolset provides a number of commands to help ease the transition to Git by providing some functionality that's similar to what you had in Subversion. Here are a few commands that give you what Subversion used to.

### SVN Style History

If you're used to Subversion and want to see your history in SVN output style, you can run `git svn log` to view your commit history in SVN formatting:

```
$ git svn log
-----
r87 | schacon | 2009-05-02 16:07:37 -0700 (Sat, 02 May 2009) | 2 lines
autogen change
```



```
-----
r86 | schacon | 2009-05-02 16:00:21 -0700 (Sat, 02 May 2009) | 2 lines
```

```
Merge branch 'experiment'
```

```
-----
r85 | schacon | 2009-05-02 16:00:09 -0700 (Sat, 02 May 2009) | 2 lines
```

```
updated the changelog
```

You should know two important things about `git svn log`. First, it works offline, unlike the real `svn log` command, which asks the Subversion server for the data. Second, it only shows you commits that have been committed up to the Subversion server. Local Git commits that you haven't dcommitted don't show up; neither do commits that people have made to the Subversion server in the meantime. It's more like the last known state of the commits on the Subversion server.

## SVN Annotation

Much as the `git svn log` command simulates the `svn log` command offline, you can get the equivalent of `svn annotate` by running `git svn blame [FILE]`. The output looks like this:

```
$ git svn blame README.txt
2   temporal Protocol Buffers - Google's data interchange format
2   temporal Copyright 2008 Google Inc.
2   temporal http://code.google.com/apis/protocolbuffers/
2   temporal
22  temporal C++ Installation - Unix
22  temporal =====
2   temporal
79  schacon Committing in git-svn.
78  schacon
2   temporal To build and install the C++ Protocol Buffer runtime and the Protocol
2   temporal Buffer compiler (protoc) execute the following:
2   temporal
```

Again, it doesn't show commits that you did locally in Git or that have been pushed to Subversion in the meantime.

## SVN Server Information

You can also get the same sort of information that `svn info` gives you by running `git svn info`:

```
$ git svn info
Path: .
URL: https://schacon-test.googlecode.com/svn/trunk
Repository Root: https://schacon-test.googlecode.com/svn
Repository UUID: 4c93b258-373f-11de-be05-5f7a86268029
Revision: 87
```

```

Node Kind: directory
Schedule: normal
Last Changed Author: schacon
Last Changed Rev: 87
Last Changed Date: 2009-05-02 16:07:37 -0700 (Sat, 02 May 2009)

```

This is like `blame` and `log` in that it runs offline and is up to date only as of the last time you communicated with the Subversion server.

## Ignoring What Subversion Ignores

If you clone a Subversion repository that has `svn:ignore` properties set anywhere, you'll likely want to set corresponding `.gitignore` files so you don't accidentally commit files that you shouldn't. `git svn` has two commands to help with this issue. The first is `git svn create-ignore`, which automatically creates corresponding `.gitignore` files for you so your next commit can include them.

The second command is `git svn show-ignore`, which prints to `stdout` the lines you need to put in a `.gitignore` file so you can redirect the output into your project exclude file:

```
$ git svn show-ignore > .git/info/exclude
```

That way, you don't litter the project with `.gitignore` files. This is a good option if you're the only Git user on a Subversion team, and your teammates don't want `.gitignore` files in the project.

## Git-Svn Summary

The `git svn` tools are useful if you're stuck with a Subversion server for now or are otherwise in a development environment that necessitates running a Subversion server. You should consider it crippled Git, however, or you'll hit issues in translation that may confuse you and your collaborators. To stay out of trouble, try to follow these guidelines:

- Keep a linear Git history that doesn't contain merge commits made by `git merge`. Rebase any work you do outside of your mainline branch back onto it; don't merge it in.
- Don't set up and collaborate on a separate Git server. Possibly have one to speed up clones for new developers, but don't push anything to it that doesn't have a `git-svn-id` entry. You may even want to add a pre-receive hook that checks each commit message for a `git-svn-id` and rejects pushes that contain commits without it.

If you follow those guidelines, working with a Subversion server can be more bearable. However, if it's possible to move to a real Git server, doing so can gain your team a lot more.

## Migrating to Git

If you have an existing codebase in another VCS but you've decided to start using Git, you must migrate your project one way or another. This section goes over some importers that are included with Git for common systems and then demonstrates how to develop your own custom importer.

## Importing

You'll learn how to import data from two of the bigger professionally used SCM systems—Subversion and Perforce—both because they make up the majority of users I hear of who are currently switching, and because high-quality tools for both systems are distributed with Git.

### Subversion

If you read the previous section about using `git svn`, you can easily use those instructions to `git svn clone` a repository; then, stop using the Subversion server, push to a new Git server, and start using that. If you want the history, you can accomplish that as quickly as you can pull the data out of the Subversion server (which may take a while).

However, the import isn't perfect; and because it will take so long, you may as well do it right. The first problem is the author information. In Subversion, each person committing has a user on the system who is recorded in the commit information. The examples in the previous section show `schacon` in some places, such as the `blame` output and the `git svn log`. If you want to map this to better Git author data, you need a mapping from the Subversion users to the Git authors. Create a file called `users.txt` that has this mapping in a format like this:

```
schacon = Scott Chacon <schacon@geemail.com>
selse = Someo Nelse <selse@geemail.com>
```

To get a list of the author names that SVN uses, you can run this:

```
$ svn log --xml | grep author | sort -u | perl -pe 's/.\>(.*?)<./$1 = /'
```

That gives you the log output in XML format—then it looks for the authors, creates a unique list, and then strips out the XML. (Obviously this only works on a machine with `grep`, `sort`, and `perl` installed.) Then, redirect that output into your `users.txt` file so you can add the equivalent Git user data next to each entry.

You can provide this file to `git svn` to help it map the author data more accurately. You can also tell `git svn` not to include the metadata that Subversion normally imports, by passing `--no-metadata` to the `clone` or `init` command. This makes your import command look like this:

```
$ git-svn clone http://my-project.googlecode.com/svn/ \
  --authors-file=users.txt --no-metadata -s my_project
```

Now you should have a nicer Subversion import in your `my_project` directory. Instead of commits that look like this:

```
commit 37efa680e8473b615de980fa935944215428a35a
Author: schacon <schacon@4c93b258-373f-11de-be05-5f7a86268029>
Date: Sun May 3 00:12:22 2009 +0000
```

```
fixed install - go to trunk
```

```
git-svn-id: https://my-project.googlecode.com/svn/trunk@94 4c93b258-373f-11de-
be05-5f7a86268029
```

they look like this:

```
commit 03a8785f44c8ea5cdb0e8834b7c8e6c469be2ff2
Author: Scott Chacon <schacon@geemail.com>
Date: Sun May 3 00:12:22 2009 +0000
```

```
fixed install - go to trunk
```

Not only does the Author field look a lot better, but the `git-svn-id` is no longer there, either.

You then need to do a bit of post-import cleanup. For one thing, you should clean up the weird references that `git svn` set up. First you'll move the tags so they're actual tags rather than strange remote branches, and then you'll move the rest of the branches so they're local.

To move the tags to be proper Git tags, run

```
$ cp -Rf .git/refs/remotes/tags/* .git/refs/tags/
$ rm -Rf .git/refs/remotes/tags
```

This takes the references that were remote branches that started with `tag/` and makes them real (lightweight) tags.

Next, move the rest of the references under `refs/remotes` to be local branches:

```
$ cp -Rf .git/refs/remotes/* .git/refs/heads/
$ rm -Rf .git/refs/remotes
```

Now all the old branches are real Git branches and all the old tags are real Git tags. The last thing to do is add your new Git server as a remote and push to it. Because you want all your branches and tags to go up, you can run this:

```
$ git push origin --all
```

All your branches and tags should be on your new Git server in a nice, clean import.

## Perforce

The next system you'll look at importing from is Perforce. A Perforce importer is also distributed with Git, but only in the `contrib` section of the source code—it isn't available by default like `git svn`. To run it, you must get the Git source code, which you can download from `git.kernel.org`:

```
$ git clone git://git.kernel.org/pub/scm/git/git.git
$ cd git/contrib/fast-import
```

In this `fast-import` directory, you should find an executable Python script named `git-p4`. You must have Python and the `p4` tool installed on your machine for this import to work. For example, you'll import the Jam project from the Perforce Public Depot. To set up your client, you must export the `P4PORT` environment variable to point to the Perforce depot:

```
$ export P4PORT=public.perforce.com:1666
```

Run the `git-p4 clone` command to import the Jam project from the Perforce server, supplying the depot and project path and the path into which you want to import the project:

```
$ git-p4 clone //public/jam/src@all /opt/p4import
Importing from //public/jam/src@all into /opt/p4import
Reinitialized existing Git repository in /opt/p4import/.git/
Import destination: refs/remotes/p4/master
Importing revision 4409 (100%)
```

If you go to the `/opt/p4import` directory and run `git log`, you can see your imported work:

```
$ git log -2
commit 1fd4ec126171790efd2db83548b85b1bbbc07dc2
Author: Perforce staff <support@perforce.com>
Date: Thu Aug 19 10:18:45 2004 -0800
```

Drop 'rc3' moniker of jam-2.5. Folded rc2 and rc3 RELNOTES into the main part of the document. Built new tar/zip balls.

Only 16 months later.

```
[git-p4: depot-paths = "//public/jam/src/": change = 4409]
```

```
commit ca8870db541a23ed867f38847eda65bf4363371d
Author: Richard Geiger <rmg@perforce.com>
Date: Tue Apr 22 20:51:34 2003 -0800
```

Update derived jamgram.c

```
[git-p4: depot-paths = "//public/jam/src/": change = 3108]
```

You can see the `git-p4` identifier in each commit. It's fine to keep that identifier there, in case you need to reference the Perforce change number later. However, if you'd like to remove the identifier, now is the time to do so—before you start doing work on the new repository. You can use `git filter-branch` to remove the identifier strings en masse:

```
$ git filter-branch --msg-filter '
    sed -e "/^\[git-p4:/d"
,
Rewrite 1fd4ec126171790efd2db83548b85b1bbbc07dc2 (123/123)
Ref 'refs/heads/master' was rewritten
```

If you run `git log`, you can see that all the SHA-1 checksums for the commits have changed, but the `git-p4` strings are no longer in the commit messages:

```
$ git log -2
commit 10a16d60cffca14d454a15c6164378f4082bc5b0
Author: Perforce staff <support@perforce.com>
Date: Thu Aug 19 10:18:45 2004 -0800
```

Drop 'rc3' moniker of jam-2.5. Folded rc2 and rc3 RELNOTES into the main part of the document. Built new tar/zip balls.

Only 16 months later.

```
commit 2b6c6db311dd76c34c66ec1c40a49405e6b527b2
Author: Richard Geiger <rimg@perforce.com>
Date: Tue Apr 22 20:51:34 2003 -0800
```

Update derived jamgram.c

Your import is ready to push up to your new Git server.

## A Custom Importer

If your system isn't Subversion or Perforce, you should look for an importer online—quality importers are available for CVS, Clear Case, Visual Source Safe, even a directory of archives. If none of these tools works for you, you have a rarer tool, or you otherwise need a more custom importing process, you should use `git fast-import`. This command reads simple instructions from stdin to write specific Git data. It's much easier to create Git objects this way than to run the raw Git commands or try to write the raw objects (see Chapter 9 for more information). This way, you can write an import script that reads the necessary information out of the system you're importing from and prints straightforward instructions to stdout. You can then run this program and pipe its output through `git fast-import`.

To quickly demonstrate, you'll write a simple importer. Suppose you work in `current`, you back up your project by occasionally copying the directory into a time-stamped `back_YYYY_MM_DD` backup directory, and you want to import this into Git. Your directory structure looks like this:

```
$ ls /opt/import_from
back_2009_01_02
back_2009_01_04
back_2009_01_14
back_2009_02_03
current
```

In order to import a Git directory, you need to review how Git stores its data. As you may remember, Git is fundamentally a linked list of commit objects that point to a snapshot of content. All you have to do is tell `fast-import` what the content snapshots are, what commit data points to them, and the order they go in. Your strategy will be to go through the snapshots one at a time and create commits with the contents of each directory, linking each commit back to the previous one.

As you did in the “An Example Git-Enforced Policy” section of Chapter 7, you'll write this in Ruby, because it's what I generally work with and it tends to be easy to read. You can write this example pretty easily in anything you're familiar with—it just needs to print the appropriate information to stdout.

To begin, you'll change into the target directory and identify every subdirectory, each of which is a snapshot that you want to import as a commit. You'll change into each subdirectory and print the commands necessary to export it. Your basic main loop looks like this:

```

last_mark = nil

# loop through the directories
Dir.chdir(ARGV[0]) do
  Dir.glob("**").each do |dir|
    next if File.file?(dir)

    # move into the target directory
    Dir.chdir(dir) do
      last_mark = print_export(dir, last_mark)
    end
  end
end

```

You run `print_export` inside each directory, which takes the manifest and mark of the previous snapshot and returns the manifest and mark of this one; that way, you can link them properly. *Mark* is the fast-import term for an identifier you give to a commit; as you create commits, you give each one a mark that you can use to link to it from other commits. So, the first thing to do in your `print_export` method is generate a mark from the directory name:

```
mark = convert_dir_to_mark(dir)
```

You'll do this by creating an array of directories and using the index value as the mark, because a mark must be an integer. Your method looks like this:

```

$marks = []
def convert_dir_to_mark(dir)
  if !$marks.include?(dir)
    $marks << dir
  end
  ($marks.index(dir) + 1).to_s
end

```

Now that you have an integer representation of your commit, you need a date for the commit metadata. Because the date is expressed in the name of the directory, you'll parse it out. The next line in your `print_export` file is

```
date = convert_dir_to_date(dir)
```

where `convert_dir_to_date` is defined as

```

def convert_dir_to_date(dir)
  if dir == 'current'
    return Time.now().to_i
  else
    dir = dir.gsub('back_', '')
    (year, month, day) = dir.split('_')
    return Time.local(year, month, day).to_i
  end
end

```

That returns an integer value for the date of each directory. The last piece of meta-information you need for each commit is the committer data, which you hardcode in a global variable:

```
$author = 'Scott Chacon <schacon@example.com>'
```

Now you're ready to begin printing out the commit data for your importer. The initial information states that you're defining a commit object and what branch it's on, followed by the mark you've generated, the committer information and commit message, and then the previous commit, if any. The code looks like this:

```
# print the import information
puts 'commit refs/heads/master'
puts 'mark :' + mark
puts "committer #{author} #{date} -0700"
export_data('imported from ' + dir)
puts 'from :' + last_mark if last_mark
```

You hardcode the time zone (-0700) because doing so is easy. If you're importing from another system, you must specify the time zone as an offset.

The commit message must be expressed in a special format:

```
data (size)\n(contents)
```

The format consists of the word `data`, the size of the data to be read, a newline, and finally the data. Because you need to use the same format to specify the file contents later, you create a helper method, `export_data`:

```
def export_data(string)
  print "data #{string.size}\n#{string}"
end
```

All that's left is to specify the file contents for each snapshot. This is easy, because you have each one in a directory—you can print out the `deleteall` command followed by the contents of each file in the directory. Git will then record each snapshot appropriately:

```
puts 'deleteall'
Dir.glob("**/*").each do |file|
  next if !File.file?(file)
  inline_data(file)
end
```

---

**Note** Because many systems think of their revisions as changes from one commit to another, `fast-import` can also take commands with each commit to specify which files have been added, removed, or modified and what the new contents are. You could calculate the differences between snapshots and provide only this data, but doing so is more complex—you may as well give Git all the data and let it figure it out. If this is better suited to your data, check the `fast-import` man page for details about how to provide your data in this manner.

---



The format for listing the new file contents or specifying a modified file with the new contents is as follows:

```
M 644 inline path/to/file
data (size)
(file contents)
```

Here, 644 is the mode (if you have executable files, you need to detect and specify 755 instead), and inline says you'll list the contents immediately after this line. Your `inline_data` method looks like this:

```
def inline_data(file, code = 'M', mode = '644')
  content = File.read(file)
  puts "#{code} #{mode} inline #{file}"
  export_data(content)
end
```

You reuse the `export_data` method you defined earlier, because it's the same as the way you specified your commit message data.

The last thing you need to do is to return the current mark so it can be passed to the next iteration:

```
return mark
```

That's it. If you run this script, you'll get content that looks something like this (you can download the full script from <http://github.com/progit/book-examples>):

```
$ ruby import.rb /opt/import_from
commit refs/heads/master
mark :1
committer Scott Chacon <schacon@geemail.com> 1230883200 -0700
data 29
imported from back_2009_01_02deleteall
M 644 inline file.rb
data 12
version two
commit refs/heads/master
mark :2
committer Scott Chacon <schacon@geemail.com> 1231056000 -0700
data 29
imported from back_2009_01_04from :1
deleteall
M 644 inline file.rb
data 14
version three
M 644 inline new.rb
data 16
new version one
(...)
```

To run the importer, pipe this output through `git fast-import` while in the Git directory you want to import into. You can create a new directory and then run `git init` in it for a starting point, and then run your script:

```
$ git init
Initialized empty Git repository in /opt/import_to/.git/
$ ruby import.rb /opt/import_from | git fast-import
git-fast-import statistics:
-----
Alloc'd objects:      5000
Total objects:       18 (      1 duplicates      )
  blobs  :           7 (      1 duplicates      0 deltas)
  trees  :           6 (      0 duplicates      1 deltas)
  commits:           5 (      0 duplicates      0 deltas)
  tags   :           0 (      0 duplicates      0 deltas)
Total branches:      1 (      1 loads      )
  marks:          1024 (      5 unique      )
  atoms:           3
Memory total:        2255 KiB
  pools:          2098 KiB
  objects:         156 KiB
-----
pack_report: getpagesize() =      4096
pack_report: core.packedGitWindowSize = 33554432
pack_report: core.packedGitLimit = 268435456
pack_report: pack_used_ctr =          9
pack_report: pack_mmap_calls =         5
pack_report: pack_open_windows =        1 /          1
pack_report: pack_mapped =      1356 /      1356
-----
```

As you can see, when it completes successfully, it gives you a bunch of statistics about what it accomplished. In this case, you imported 18 objects total for 5 commits into 1 branch. Now, you can run `git log` to see your new history:

```
$ git log -2
commit 10bfe7d22ce15ee25b60a824c8982157ca593d41
Author: Scott Chacon <schacon@example.com>
Date:   Sun May 3 12:57:39 2009 -0700

    imported from current

commit 7e519590de754d079dd73b44d695a42c9d2df452
Author: Scott Chacon <schacon@example.com>
Date:   Tue Feb 3 01:00:00 2009 -0700

    imported from back_2009_02_03
```

There you go—a nice, clean Git repository. It's important to note that nothing is checked out—you don't have any files in your working directory at first. To get them, you must reset your branch to where master is now:

```
$ ls
$ git reset --hard master
HEAD is now at 10bfe7d imported from current
$ ls
file.rb lib
```

You can do a lot more with the `fast-import` tool—handle different modes, binary data, multiple branches and merging, tags, progress indicators, and more. A number of examples of more complex scenarios are available in the `contrib/fast-import` directory of the Git source code; one of the better ones is the `git-p4` script I just covered.

## Summary

You should feel comfortable using Git with Subversion or importing nearly any existing repository into a new Git one without losing data. The next chapter will cover the raw internals of Git so you can craft every single byte, if need be.