9.2 Git and Other Systems - Migrating to Git

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 for common systems, and then demonstrates how to develop your own custom importer. You'll learn how to import data from several of the bigger professionally used SCM systems, because they make up the majority of users who are switching, and because high-quality tools for them are easy to come by.

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

That generates the log output in XML format, then keeps only the lines with author information, discards duplicates, strips out the XML tags. 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.

Note If you're trying this on a Windows machine, this is the point where you'll run into trouble. Microsoft have provided some good advice and samples at https://docs.microsoft.com/en-us/azure/devops/repos/git/perform-migration-from-svn-to-git.

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. The metadata includes a `git-svn-id` inside each commit message that Git will generate during import. This can bloat your Git log and might make it a bit unclear.

Note You need to keep the metadata when you want to mirror commits made in the Git repository back into the original SVN repository. If you don't want the synchronization in your commit log, feel free to omit the `--no-metadata` parameter.

This makes your `import` command look like this:

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

```
$ for t in $(git for-each-ref --format='%(refname:short)' refs/remotes/tags); do git tag ${t/tags\//} $t && git branch -D -r $t; done
```

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

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

```
$ for b in $(git for-each-ref --format='%(refname:short)' refs/remotes); do git branch $b refs/remotes/$b && git branch -D -r $b; done
```

It may happen that you'll see some extra branches which are suffixed by `@xxx` (where xxx is a number), while in Subversion you only see one branch. This is actually a Subversion feature called "peg-revisions", which is something that Git simply has no syntactical counterpart for. Hence, `git svn` simply adds the svn version number to the branch name just in the same way as you would have written it in svn to address the peg-revision of that branch. If you do not care anymore about the peg-revisions, simply remove them:

```
$ for p in $(git for-each-ref --format='%(refname:short)' | grep @); do git branch -D $p; done
```

Now all the old branches are real Git branches and all the old tags are real Git tags.

There's one last thing to clean up. Unfortunately, `git svn` creates an extra branch named `trunk`, which maps to Subversion's default branch, but the `trunk` ref points to the same place as `master`. Since `master` is more idiomatically Git, here's how to remove the extra branch:

```
$ git branch -d trunk
```

The last thing to do is add your new Git server as a remote and push to it. Here is an example of adding your server as a remote:

```
$ git remote add origin git@my-git-server:myrepository.git
```

Because you want all your branches and tags to go up, you can now run this:

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

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

Mercurial

Since Mercurial and Git have fairly similar models for representing versions, and since Git is a bit more flexible, converting a repository from Mercurial to Git is fairly straightforward, using a tool called "hg-fast-export", which you'll need a copy of:

```
$ git clone https://github.com/frej/fast-export.git
```

The first step in the conversion is to get a full clone of the Mercurial repository you want to convert:

```
$ hg clone <remote repo URL> /tmp/hg-repo
```

The next step is to create an author mapping file. Mercurial is a bit more forgiving than Git for what it will put in the author field for changesets, so this is a good time to clean house. Generating this is a one-line command in a `bash` shell:

```
$ cd /tmp/hg-repo
$ hg log | grep user: | sort | uniq | sed 's/user: *//' > ../authors
```

This will take a few seconds, depending on how long your project's history is, and afterwards the `/tmp/authors` file will look something like this:

```
bob
bob@localhost
bob <bob@company.com>
bob jones <bob <AT> company <DOT> com>
Bob Jones <bob@company.com>
Joe Smith <joe@company.com>
```

In this example, the same person (Bob) has created changesets under four different names, one of which actually looks correct, and one of which would be completely invalid for a Git commit. Hg-fast-export lets us fix this by turning each line into a rule: `"<input>"="<output>"`, mapping an `<input>` to an `<output>`. Inside the `<input>` and `<output>` strings, all escape sequences understood by the python `string_escape` encoding are supported. If the author mapping file does not contain a matching `<input>`, that author will be sent on to Git unmodified. If all the usernames look fine, we won't need this file at all. In this example, we want our file to look like this:

```
"bob"="Bob Jones <bob@company.com>"
"bob@localhost"="Bob Jones <bob@company.com>"
"bob <bob@company.com>"="Bob Jones <bob@company.com>"
"bob jones <bob <AT> company <DOT> com>"="Bob Jones <bob@company.com>"
```

The same kind of mapping file can be used to rename branches and tags when the Mercurial name is not allowed by Git.

The next step is to create our new Git repository, and run the export script:

```
$ git init /tmp/converted
$ cd /tmp/converted
$ /tmp/fast-export/hg-fast-export.sh -r /tmp/hg-repo -A /tmp/authors
```

The `-r` flag tells hg-fast-export where to find the Mercurial repository we want to convert, and the `-A` flag tells it where to find the author-mapping file (branch and tag mapping files are specified by the `-B` and `-T` flags respectively). The script parses Mercurial changesets and converts them into a script for Git's "fast-import" feature (which we'll discuss in detail a bit later on). This takes a bit (though it's *much* faster than it would be over the network), and the output is fairly verbose:

```
$ /tmp/fast-export/hg-fast-export.sh -r /tmp/hg-repo -A /tmp/authors
Loaded 4 authors
master: Exporting full revision 1/22208 with 13/0/0 added/changed/removed files
master: Exporting simple delta revision 2/22208 with 1/1/0 added/changed/removed files
master: Exporting simple delta revision 3/22208 with 0/1/0 added/changed/removed files
[…]
master: Exporting simple delta revision 22206/22208 with 0/4/0 added/changed/removed files
master: Exporting simple delta revision 22207/22208 with 0/2/0 added/changed/removed files
```

```
master: Exporting thorough delta revision 22208/22208 with 3/213/0 added/changed/removed files
Exporting tag [0.4c] at [hg r9] [git :10]
Exporting tag [0.4d] at [hg r16] [git :17]
[…]
Exporting tag [3.1-rc] at [hg r21926] [git :21927]
Exporting tag [3.1] at [hg r21973] [git :21974]
Issued 22315 commands
git-fast-import statistics:
---------------------------------------------------------------------
Alloc'd objects:     120000
Total objects:       115032 (    208171 duplicates                 )
      blobs  :        40504 (    205320 duplicates      26117 deltas of      39602 attempts)
      trees  :        52320 (      2851 duplicates      47467 deltas of      47599 attempts)
      commits:        22208 (         0 duplicates          0 deltas of          0 attempts)
      tags   :            0 (         0 duplicates          0 deltas of          0 attempts)
Total branches:         109 (         2 loads     )
      marks:        1048576 (     22208 unique    )
      atoms:           1952
Memory total:          7860 KiB
       pools:          2235 KiB
     objects:          5625 KiB
---------------------------------------------------------------------
pack_report: getpagesize()            =       4096
pack_report: core.packedGitWindowSize = 1073741824
pack_report: core.packedGitLimit      = 8589934592
pack_report: pack_used_ctr            =      90430
pack_report: pack_mmap_calls          =      46771
pack_report: pack_open_windows        =          1 /          1
pack_report: pack_mapped              =  340852700 /  340852700
---------------------------------------------------------------------

$ git shortlog -sn
   369  Bob Jones
   365  Joe Smith
```

That's pretty much all there is to it. All of the Mercurial tags have been converted to Git tags, and Mercurial branches and bookmarks have been converted to Git branches. Now you're ready to push the repository up to its new server-side home:

```
$ git remote add origin git@my-git-server:myrepository.git
$ git push origin --all
```

Bazaar

Bazaar is a DVCS tool much like Git, and as a result it's pretty straightforward to convert a Bazaar repository into a Git one. To accomplish this, you'll need to import the `bzr-fastimport` plugin.

Getting the bzr-fastimport plugin

The procedure for installing the fastimport plugin is different on UNIX-like operating systems and on Windows. In the first case, the simplest is to install the `bzr-fastimport` package that will install all the required dependencies.

For example, with Debian and derived, you would do the following:

```
$ sudo apt-get install bzr-fastimport
```

With RHEL, you would do the following:

```
$ sudo yum install bzr-fastimport
```

With Fedora, since release 22, the new package manager is dnf:

```
$ sudo dnf install bzr-fastimport
```

If the package is not available, you may install it as a plugin:

```
$ mkdir --parents ~/.bazaar/plugins     # creates the necessary folders for the plugins
$ cd ~/.bazaar/plugins
$ bzr branch lp:bzr-fastimport fastimport   # imports the fastimport plugin
$ cd fastimport
$ sudo python setup.py install --record=files.txt   # installs the plugin
```

For this plugin to work, you'll also need the `fastimport` Python module. You can check whether it is present or not and install it with the following commands:

```
$ python -c "import fastimport"
Traceback (most recent call last):
  File "<string>", line 1, in <module>
ImportError: No module named fastimport
$ pip install fastimport
```

If it is not available, you can download it at address https://pypi.python.org/pypi/fastimport/.

In the second case (on Windows), `bzr-fastimport` is automatically installed with the standalone version and the default installation (let all the checkboxes checked). So in this case you have nothing to do.

At this point, the way to import a Bazaar repository differs according to that you have a single branch or you are working with a repository that has several branches.

Project with a single branch

Now `cd` in the directory that contains your Bazaar repository and initialize the Git repository:

```
$ cd /path/to/the/bzr/repository
$ git init
```

Now, you can simply export your Bazaar repository and convert it into a Git repository using the following command:

```
$ bzr fast-export --plain . | git fast-import
```

Depending on the size of the project, your Git repository is built in a lapse from a few seconds to a few minutes.

Case of a project with a main branch and a working branch

You can also import a Bazaar repository that contains branches. Let us suppose that you have two branches: one represents the main branch (myProject.trunk), the other one is the working branch (myProject.work).

```
$ ls
myProject.trunk myProject.work
```

Create the Git repository and `cd` into it:

```
$ git init git-repo
$ cd git-repo
```

Pull the `master` branch into git:

```
$ bzr fast-export --export-marks=../marks.bzr ../myProject.trunk | \
git fast-import --export-marks=../marks.git
```

Pull the working branch into Git:

```
$ bzr fast-export --marks=../marks.bzr --git-branch=work ../myProject.work | \
git fast-import --import-marks=../marks.git --export-marks=../marks.git
```

Now `git branch` shows you the `master` branch as well as the `work` branch. Check the logs to make sure they're complete and get rid of the `marks.bzr`and `marks.git` files.

Synchronizing the staging area

Whatever the number of branches you had and the import method you used, your staging area is not synchronized with `HEAD`, and with the import of several branches, your working directory is not synchronized either. This situation is easily solved by the following command:

```
$ git reset --hard HEAD
```

Ignoring the files that were ignored with .bzrignore

Now let's have a look at the files to ignore. The first thing to do is to rename `.bzrignore` into `.gitignore`. If the `.bzrignore` file contains one or several lines starting with "!!" or "RE:", you'll have to modify it and perhaps create several `.gitignore` files in order to ignore exactly the same files that Bazaar was ignoring.

Finally, you will have to create a commit that contains this modification for the migration:

```
$ git mv .bzrignore .gitignore
$ # modify .gitignore if needed
$ git commit -am 'Migration from Bazaar to Git'
```

Sending your repository to the server

Here we are! Now you can push the repository onto its new home server:

```
$ git remote add origin git@my-git-server:mygitrepository.git
$ git push origin --all
$ git push origin --tags
```

Your Git repository is ready to use.

Perforce

The next system you'll look at importing from is Perforce. As we discussed above, there are two ways to let Git and Perforce talk to each other: git-p4 and Perforce Git Fusion.

Perforce Git Fusion

Git Fusion makes this process fairly painless. Just configure your project settings, user mappings, and branches using a configuration file (as discussed in Git Fusion), and clone the repository. Git Fusion leaves you with what looks like a native Git repository, which is then ready to push to a native Git host if you desire. You could even use Perforce as your Git host if you like.

Git-p4

Git-p4 can also act as an import tool. As an example, we'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
```

Note In order to follow along, you'll need a Perforce depot to connect with. We'll be using the public depot at public.perforce.com for our examples, but you can use any depot you have access to.

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 //guest/perforce_software/jam@all p4import
Importing from //guest/perforce_software/jam@all into p4import
Initialized empty Git repository in /private/tmp/p4import/.git/
Import destination: refs/remotes/p4/master
Importing revision 9957 (100%)
```

This particular project has only one branch, but if you have branches that are configured with branch views (or just a set of directories), you can use the `--detect-branches` flag to `git p4 clone` to import all the project's branches as well. See Branching for a bit more detail on this.

At this point you're almost done. If you go to the `p4import` directory and run `git log`, you can see your imported work:

```
$ git log -2
commit e5da1c909e5db3036475419f6379f2c73710c4e6
Author: giles <giles@giles@perforce.com>
Date:   Wed Feb 8 03:13:27 2012 -0800

    Correction to line 355; change </UL> to </OL>.

    [git-p4: depot-paths = "//public/jam/src/": change = 8068]

commit aa21359a0a135dda85c50a7f7cf249e4f7b8fd98
Author: kwirth <kwirth@perforce.com>
Date:   Tue Jul 7 01:35:51 2009 -0800

    Fix spelling error on Jam doc page (cummulative -> cumulative).

    [git-p4: depot-paths = "//public/jam/src/": change = 7304]
```

You can see that `git-p4` has left an identifier in each commit message. 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 e5da1c909e5db3036475419f6379f2c73710c4e6 (125/125)
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 b17341801ed838d97f7800a54a6f9b95750839b7
Author: giles <giles@giles@perforce.com>
Date:   Wed Feb 8 03:13:27 2012 -0800

    Correction to line 355; change </UL> to </OL>.

commit 3e68c2e26cd89cb983eb52c024ecdfba1d6b3fff
Author: kwirth <kwirth@perforce.com>
Date:   Tue Jul 7 01:35:51 2009 -0800

    Fix spelling error on Jam doc page (cummulative -> cumulative).
```

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

A Custom Importer

If your system isn't one of the above, you should look for an importer online – quality importers are available for many other systems, including CVS, Clear Case, Visual Source Safe, even a directory of archives. If none of these tools works for you, you have a more obscure 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 Git Internalsfor 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_2014_01_02
back_2014_01_04
back_2014_01_14
back_2014_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 we did in An Example Git-Enforced Policy, we'll write this in Ruby, because it's what we 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`. And, if you are running on Windows, this means you'll need to take special care to not introduce carriage returns at the end your lines – `git fast-import` is very particular about just wanting line feeds (LF) not the carriage return line feeds (CRLF) that Windows uses.

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 = 'John Doe <john@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
```

Note: If you are running on Windows you'll need to make sure that you add one extra step. As mentioned before, Windows uses CRLF for new line characters while `git fast-import` expects only LF. To get around this problem and make `git fast-import` happy, you need to tell ruby to use LF instead of CRLF:

```
$stdout.binmode
```

That's it. Here's the script in its entirety:

```
#!/usr/bin/env ruby

$stdout.binmode
$author = "John Doe <john@example.com>"

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

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

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

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

def print_export(dir, last_mark)
    date = convert_dir_to_date(dir)
    mark = convert_dir_to_mark(dir)

    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

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

```
        inline_data(file)
    end
    mark
end

# Loop through the directories
last_mark = nil
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
```

If you run this script, you'll get content that looks something like this:

```
$ ruby import.rb /opt/import_from
commit refs/heads/master
mark :1
committer John Doe <john@example.com> 1388649600 -0700
data 29
imported from back_2014_01_02deleteall
M 644 inline README.md
data 28
# Hello

This is my readme.
commit refs/heads/master
mark :2
committer John Doe <john@example.com> 1388822400 -0700
data 29
imported from back_2014_01_04from :1
deleteall
M 644 inline main.rb
data 34
#!/bin/env ruby

puts "Hey there"
M 644 inline README.md
(...)
```

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:           13 (         6 duplicates                  )
      blobs  :            5 (         4 duplicates          3 deltas of          5 attempts)
      trees  :            4 (         1 duplicates          0 deltas of          4 attempts)
      commits:            4 (         1 duplicates          0 deltas of          0 attempts)
      tags   :            0 (         0 duplicates          0 deltas of          0 attempts)
Total branches:           1 (         1 loads     )
      marks:           1024 (         5 unique    )
      atoms:              2
Memory total:          2344 KiB
       pools:          2110 KiB
     objects:           234 KiB
---------------------------------------------------------------------
pack_report: getpagesize()            =       4096
pack_report: core.packedGitWindowSize = 1073741824
pack_report: core.packedGitLimit      = 8589934592
pack_report: pack_used_ctr            =         10
pack_report: pack_mmap_calls          =          5
pack_report: pack_open_windows        =          2 /          2
pack_report: pack_mapped              =       1457 /       1457
---------------------------------------------------------------------
```

As you can see, when it completes successfully, it gives you a bunch of statistics about what it accomplished. In this case, you imported 13 objects total for 4 commits into 1 branch. Now, you can run `git log` to see your new history:

```
$ git log -2
commit 3caa046d4aac682a55867132ccdfbe0d3fdee498
Author: John Doe <john@example.com>
Date:   Tue Jul 29 19:39:04 2014 -0700

    imported from current

commit 4afc2b945d0d3c8cd00556fbe2e8224569dc9def
Author: John Doe <john@example.com>
Date:   Mon Feb 3 01:00:00 2014 -0700

    imported from back_2014_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 3caa046 imported from current
$ ls
README.md main.rb
```

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.

prev | next