迁移到 Git

如果在其他版本控制系统中保存了某项目的代码而后决定转而使用 Git，那么该项目必须经历某种形式的迁移。本节将介绍 Git 中包含的一些针对常见系统的导入脚本，并将展示编写自定义的导入脚本的方法。

导入

你将学习到如何从专业重量级的版本控制系统中导入数据—— Subversion 和 Perforce —— 因为据我所知这二者的用户是（向 Git）转换的主要群体，而且 Git 为此二者附带了高质量的转换工具。

Subversion

读过前一节有关 git svn 的内容以后，你应该能轻而易举的根据其中的指导来 git svn clone 一个仓库了；然后，停止 Subversion 的使用，向一个新 Git server 推送，并开始使用它。想保留历史记录，所花的时间应该不过就是从 Subversion 服务器拉取数据的时间（可能要等上好一会就是了）。

然而，这样的导入并不完美；而且还要花那么多时间，不如干脆一次把它做对！首当其冲的任务是作者信息。在 Subversion，每个提交者在都在主机上有一个用户名，记录在提交信息中。上节例子中多处显示了 schacon ，比如blame 的输出以及 git svn log。如果想让这条信息更好的映射到 Git 作者数据里，则需要 从 Subversion 用户名到 Git 作者的一个映射关系。建立一个叫做 user.txt 的文件，用如下格式表示映射关系：

schacon = Scott Chacon <schacon@geemail.com>

selse = Someo Nelse <selse@geemail.com>

通过该命令可以获得 SVN 作者的列表：

$ svn log ^/ --xml | grep -P "^<author" | sort -u | \

perl -pe 's/<author>(.\*?)<\/author>/$1 = /' > users.txt

它将输出 XML 格式的日志——你可以找到作者，建立一个单独的列表，然后从 XML 中抽取出需要的信息。（显而易见，本方法要求主机上安装了grep，sort 和 perl.）然后把输出重定向到 user.txt 文件，然后就可以在每一项的后面添加相应的 Git 用户数据。

为 git svn 提供该文件可以然它更精确的映射作者数据。你还可以在 clone 或者 init后面添加 --no-metadata来阻止 git svn 包含那些 Subversion 的附加信息。这样 import 命令就变成了：

$ git svn clone http://my-project.googlecode.com/svn/ \

--authors-file=users.txt --no-metadata -s my\_project

现在 my\_project 目录下导入的 Subversion 应该比原来整洁多了。原来的 commit 看上去是这样：

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

现在是这样：

commit 03a8785f44c8ea5cdb0e8834b7c8e6c469be2ff2

Author: Scott Chacon <schacon@geemail.com>

Date: Sun May 3 00:12:22 2009 +0000

fixed install - go to trunk

不仅作者一项干净了不少，git-svn-id 也就此消失了。

你还需要一点 post-import（导入后） 清理工作。最起码的，应该清理一下 git svn 创建的那些怪异的索引结构。首先要移动标签，把它们从奇怪的远程分支变成实际的标签，然后把剩下的分支移动到本地。

要把标签变成合适的 Git 标签，运行

$ git for-each-ref refs/remotes/tags | cut -d / -f 4- | grep -v @ | while read tagname; do git tag "$tagname" "tags/$tagname"; git branch -r -d "tags/$tagname"; done

该命令将原本以 tag/ 开头的远程分支的索引变成真正的（轻巧的）标签。

接下来，把 refs/remotes 下面剩下的索引变成本地分支：

$ git for-each-ref refs/remotes | cut -d / -f 3- | grep -v @ | while read branchname; do git branch "$branchname" "refs/remotes/$branchname"; git branch -r -d "$branchname"; done

现在所有的旧分支都变成真正的 Git 分支，所有的旧标签也变成真正的 Git 标签。最后一项工作就是把新建的 Git 服务器添加为远程服务器并且向它推送。下面是新增远程服务器的例子：

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

为了让所有的分支和标签都得到上传，我们使用这条命令：

$ git push origin --all

$ git push origin --tags

所有的分支和标签现在都应该整齐干净的躺在新的 Git 服务器里了。

Perforce

你将了解到的下一个被导入的系统是 Perforce. Git 发行的时候同时也附带了一个 Perforce 导入脚本，不过它是包含在源码的 contrib 部分——而不像 git svn 那样默认可用。运行它之前必须获取 Git 的源码，可以在 git.kernel.org 下载：

$ git clone git://git.kernel.org/pub/scm/git/git.git

$ cd git/contrib/fast-import

在这个 fast-import 目录下，应该有一个叫做 git-p4 的 Python 可执行脚本。主机上必须装有 Python 和 p4 工具该导入才能正常进行。例如，你要从 Perforce 公共代码仓库（译注： Perforce Public Depot，Perforce 官方提供的代码寄存服务）导入 Jam 工程。为了设定客户端，我们要把 P4PORT 环境变量 export 到 Perforce 仓库：

$ export P4PORT=public.perforce.com:1666

运行 git-p4 clone 命令将从 Perforce 服务器导入 Jam 项目，我们需要给出仓库和项目的路径以及导入的目标路径：

$ 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%)

现在去 /opt/p4import 目录运行一下 git log ，就能看到导入的成果：

$ 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]

每一个 commit 里都有一个 git-p4 标识符。这个标识符可以保留，以防以后需要引用 Perforce 的修改版本号。然而，如果想删除这些标识符，现在正是时候——在开启新仓库之前。可以通过 git filter-branch 来批量删除这些标识符：

$ git filter-branch --msg-filter '

sed -e "/^\[git-p4:/d"

'

Rewrite 1fd4ec126171790efd2db83548b85b1bbbc07dc2 (123/123)

Ref 'refs/heads/master' was rewritten

现在运行一下 git log，你会发现这些 commit 的 SHA-1 校验值都发生了改变，而那些 git-p4 字串则从提交信息里消失了：

$ 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 <rmg@perforce.com>

Date: Tue Apr 22 20:51:34 2003 -0800

Update derived jamgram.c

至此导入已经完成，可以开始向新的 Git 服务器推送了。

自定导入脚本

如果先前的系统不是 Subversion 或 Perforce 之一，先上网找一下有没有与之对应的导入脚本——导入 CVS，Clear Case，Visual Source Safe，甚至存档目录的导入脚本已经存在。假如这些工具都不适用，或者使用的工具很少见，抑或你需要导入过程具有更多可制定性，则应该使用 git fast-import。该命令从标准输入读取简单的指令来写入具体的 Git 数据。这样创建 Git 对象比运行纯 Git 命令或者手动写对象要简单的多（更多相关内容见第九章）。通过它，你可以编写一个导入脚本来从导入源读取必要的信息，同时在标准输出直接输出相关指示。你可以运行该脚本并把它的输出管道连接到 git fast-import。

下面演示一下如何编写一个简单的导入脚本。假设你在进行一项工作，并且按时通过把工作目录复制为以时间戳back\_YY\_MM\_DD 命名的目录来进行备份，现在你需要把它们导入 Git 。目录结构如下：

$ ls /opt/import\_from

back\_2009\_01\_02

back\_2009\_01\_04

back\_2009\_01\_14

back\_2009\_02\_03

current

为了导入到一个 Git 目录，我们首先回顾一下 Git 储存数据的方式。你可能还记得，Git 本质上是一个 commit 对象的链表，每一个对象指向一个内容的快照。而这里需要做的工作就是告诉 fast-import 内容快照的位置，什么样的 commit 数据指向它们，以及它们的顺序。我们采取一次处理一个快照的策略，为每一个内容目录建立对应的 commit ，每一个 commit 与之前的建立链接。

正如在第七章 "Git 执行策略一例" 一节中一样，我们将使用 Ruby 来编写这个脚本，因为它是我日常使用的语言而且阅读起来简单一些。你可以用任何其他熟悉的语言来重写这个例子——它仅需要把必要的信息打印到标准输出而已。同时，如果你在使用 Windows，这意味着你要特别留意不要在换行的时候引入回车符（译注：carriage returns，Windows 换行时加入的符号，通常说的 \r ）—— git fast-import 对仅使用换行符（LF）而非 Windows 的回车符（CRLF）要求非常严格。

首先，进入目标目录并且找到所有子目录，每一个子目录将作为一个快照被导入为一个 commit。我们将依次进入每一个子目录并打印所需的命令来导出它们。脚本的主循环大致是这样：

last\_mark = nil

# 循环遍历所有目录

Dir.chdir(ARGV[0]) do

Dir.glob("\*").each do |dir|

next if File.file?(dir)

# 进入目标目录

Dir.chdir(dir) do

last\_mark = print\_export(dir, last\_mark)

end

end

end

我们在每一个目录里运行 print\_export ，它会取出上一个快照的索引和标记并返回本次快照的索引和标记；由此我们就可以正确的把二者连接起来。"标记（mark）" 是 fast-import 中对 commit 标识符的叫法；在创建 commit 的同时，我们逐一赋予一个标记以便以后在把它连接到其他 commit 时使用。因此，在 print\_export 方法中要做的第一件事就是根据目录名生成一个标记：

mark = convert\_dir\_to\_mark(dir)

实现该函数的方法是建立一个目录的数组序列并使用数组的索引值作为标记，因为标记必须是一个整数。这个方法大致是这样的：

$marks = []

def convert\_dir\_to\_mark(dir)

if !$marks.include?(dir)

$marks << dir

end

($marks.index(dir) + 1).to\_s

end

有了整数来代表每个 commit，我们现在需要提交附加信息中的日期。由于日期是用目录名表示的，我们就从中解析出来。print\_export 文件的下一行将是：

date = convert\_dir\_to\_date(dir)

而 convert\_dir\_to\_date 则定义为

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

它为每个目录返回一个整型值。提交附加信息里最后一项所需的是提交者数据，我们在一个全局变量中直接定义之：

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

我们差不多可以开始为导入脚本输出提交数据了。第一项信息指明我们定义的是一个 commit 对象以及它所在的分支，随后是我们生成的标记，提交者信息以及提交备注，然后是前一个 commit 的索引，如果有的话。代码大致这样：

# 打印导入所需的信息

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

时区（-0700）处于简化目的使用硬编码。如果是从其他版本控制系统导入，则必须以变量的形式指明时区。 提交备注必须以特定格式给出：

data (size)\n(contents)

该格式包含了单词 data，所读取数据的大小，一个换行符，最后是数据本身。由于随后指明文件内容的时候要用到相同的格式，我们写一个辅助方法，export\_data：

def export\_data(string)

print "data #{string.size}\n#{string}"

end

唯一剩下的就是每一个快照的内容了。这简单的很，因为它们分别处于一个目录——你可以输出 deleeall 命令，随后是目录中每个文件的内容。Git 会正确的记录每一个快照：

puts 'deleteall'

Dir.glob("\*\*/\*").each do |file|

next if !File.file?(file)

inline\_data(file)

end

注意：由于很多系统把每次修订看作一个 commit 到另一个 commit 的变化量，fast-import 也可以依据每次提交获取一个命令来指出哪些文件被添加，删除或者修改过，以及修改的内容。我们将需要计算快照之间的差别并且仅仅给出这项数据，不过该做法要复杂很多——还如不直接把所有数据丢给 Git 然它自己搞清楚。假如前面这个方法更适用于你的数据，参考 fast-import 的 man 帮助页面来了解如何以这种方式提供数据。

列举新文件内容或者指明带有新内容的已修改文件的格式如下：

M 644 inline path/to/file

data (size)

(file contents)

这里，644 是权限模式（加入有可执行文件，则需要探测之并设定为 755），而 inline 说明我们在本行结束之后立即列出文件的内容。我们的 inline\_data 方法大致是：

def inline\_data(file, code = 'M', mode = '644')

content = File.read(file)

puts "#{code} #{mode} inline #{file}"

export\_data(content)

end

我们重用了前面定义过的 export\_data，因为这里和指明提交注释的格式如出一辙。

最后一项工作是返回当前的标记以便下次循环的使用。

return mark

注意：如果你在用 Windows，一定记得添加一项额外的步骤。前面提过，Windows 使用 CRLF 作为换行字符而 git fast-import 只接受 LF。为了绕开这个问题来满足 git fast-import，你需要让 ruby 用 LF 取代 CRLF：

$stdout.binmode

搞定了。现在运行该脚本，你将得到如下内容：

$ 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

(...)

要运行导入脚本，在需要导入的目录把该内容用管道定向到 git fast-import。你可以建立一个空目录然后运行git init 作为开头，然后运行该脚本：

$ 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

---------------------------------------------------------------------

你会发现，在它成功执行完毕以后，会给出一堆有关已完成工作的数据。上例在一个分支导入了5次提交数据，包含了18个对象。现在可以运行 git log 来检视新的历史：

$ 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

就它了——一个干净整洁的 Git 仓库。需要注意的是此时没有任何内容被检出——刚开始当前目录里没有任何文件。要获取它们，你得转到 master 分支的所在：

$ ls

$ git reset --hard master

HEAD is now at 10bfe7d imported from current

$ ls

file.rb lib

fast-import 还可以做更多——处理不同的文件模式，二进制文件，多重分支与合并，标签，进展标识等等。一些更加复杂的实例可以在 Git 源码的 contib/fast-import 目录里找到；其中较为出众的是前面提过的 git-p4 脚本。