Git 与 Subversion

当前，大多数开发中的开源项目以及大量的商业项目都使用 Subversion 来管理源码。作为最流行的开源版本控制系统，Subversion 已经存在了接近十年的时间。它在许多方面与 CVS 十分类似，后者是前者出现之前代码控制世界的霸主。

Git 最为重要的特性之一是名为 git svn 的 Subversion 双向桥接工具。该工具把 Git 变成了 Subversion 服务的客户端，从而让你在本地享受到 Git 所有的功能，而后直接向 Subversion 服务器推送内容，仿佛在本地使用了 Subversion 客户端。也就是说，在其他人忍受古董的同时，你可以在本地享受分支合并，使暂存区域，衍合以及 单项挑拣等等。这是个让 Git 偷偷潜入合作开发环境的好东西，在帮助你的开发同伴们提高效率的同时，它还能帮你劝说团队让整个项目框架转向对 Git 的支持。这个 Subversion 之桥是通向分布式版本控制系统（DVCS, Distributed VCS ）世界的神奇隧道。

git svn

Git 中所有 Subversion 桥接命令的基础是 git svn 。所有的命令都从它开始。相关的命令数目不少，你将通过几个简单的工作流程了解到其中常见的一些。

值得警戒的是，在使用 git svn 的时候，你实际是在与 Subversion 交互，Git 比它要高级复杂的多。尽管可以在本地随意的进行分支和合并，最好还是通过衍合保持线性的提交历史，尽量避免类似与远程 Git 仓库动态交互这样的操作。

避免修改历史再重新推送的做法，也不要同时推送到并行的 Git 仓库来试图与其他 Git 用户合作。Subersion 只能保存单一的线性提交历史，一不小心就会被搞糊涂。合作团队中同时有人用 SVN 和 Git，一定要确保所有人都使用 SVN 服务来协作——这会让生活轻松很多。

初始设定

为了展示功能，先要一个具有写权限的 SVN 仓库。如果想尝试这个范例，你必须复制一份其中的测试仓库。比较简单的做法是使用一个名为 svnsync 的工具。较新的 Subversion 版本中都带有该工具，它将数据编码为用于网络传输的格式。

要尝试本例，先在本地新建一个 Subversion 仓库：

$ mkdir /tmp/test-svn

$ svnadmin create /tmp/test-svn

然后，允许所有用户修改 revprop —— 简单的做法是添加一个总是以 0 作为返回值的 pre-revprop-change 脚本：

$ cat /tmp/test-svn/hooks/pre-revprop-change

#!/bin/sh

exit 0;

$ chmod +x /tmp/test-svn/hooks/pre-revprop-change

现在可以调用 svnsync init 加目标仓库，再加源仓库的格式来把该项目同步到本地了：

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

这将建立进行同步所需的属性。可以通过运行以下命令来克隆代码：

$ 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.

...

别看这个操作只花掉几分钟，要是你想把源仓库复制到另一个远程仓库，而不是本地仓库，那将花掉接近一个小时，尽管项目中只有不到 100 次的提交。 Subversion 每次只复制一次修改，把它推送到另一个仓库里，然后周而复始——惊人的低效，但是我们别无选择。

入门

有了可以写入的 Subversion 仓库以后，就可以尝试一下典型的工作流程了。我们从 git svn clone 命令开始，它会把整个 Subversion 仓库导入到一个本地的 Git 仓库中。提醒一下，这里导入的是一个货真价实的 Subversion 仓库，所以应该把下面的 file:///tmp/test-svn 换成你所用的 Subversion 仓库的 URL：

$ 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

这相当于针对所提供的 URL 运行了两条命令—— git svn init 加上 git svn fetch 。可能会花上一段时间。我们所用的测试项目仅仅包含 75 次提交并且它的代码量不算大，所以只有几分钟而已。不过，Git 仍然需要提取每一个版本，每次一个，再逐个提交。对于一个包含成百上千次提交的项目，花掉的时间则可能是几小时甚至数天。

-T trunk -b branches -t tags 告诉 Git 该 Subversion 仓库遵循了基本的分支和标签命名法则。如果你的主干(译注：trunk，相当于非分布式版本控制里的master分支，代表开发的主线），分支或者标签以不同的方式命名，则应做出相应改变。由于该法则的常见性，可以使用 -s 来代替整条命令，它意味着标准布局（s 是 Standard layout 的首字母），也就是前面选项的内容。下面的命令有相同的效果：

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

现在，你有了一个有效的 Git 仓库，包含着导入的分支和标签：

$ 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

值得注意的是，该工具分配命名空间时和远程引用的方式不尽相同。克隆普通的 Git 仓库时，可以以origin/[branch] 的形式获取远程服务器上所有可用的分支——分配到远程服务的名称下。然而 git svn 假定不存在多个远程服务器，所以把所有指向远程服务的引用不加区分的保存下来。可以用 Git 探测命令 show-ref 来查看所有引用的全名。

$ 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

1c4cb508144c513ff1214c3488abe66dcb92916f refs/remotes/tags/release-2.0.2rc1

1cbd4904d9982f386d87f88fce1c24ad7c0f0471 refs/remotes/trunk

而普通的 Git 仓库应该是这个模样：

$ git show-ref

83e38c7a0af325a9722f2fdc56b10188806d83a1 refs/heads/master

3e15e38c198baac84223acfc6224bb8b99ff2281 refs/remotes/gitserver/master

0a30dd3b0c795b80212ae723640d4e5d48cabdff refs/remotes/origin/master

25812380387fdd55f916652be4881c6f11600d6f refs/remotes/origin/testing

这里有两个远程服务器：一个名为 gitserver ，具有一个 master分支；另一个叫 origin，具有 master 和testing 两个分支。

注意本例中通过 git svn 导入的远程引用，（Subversion 的）标签是当作远程分支添加的，而不是真正的 Git 标签。导入的 Subversion 仓库仿佛是有一个带有不同分支的 tags 远程服务器。

提交到 Subversion

有了可以开展工作的（本地）仓库以后，你可以开始对该项目做出贡献并向上游仓库提交内容了，Git 这时相当于一个 SVN 客户端。假如编辑了一个文件并进行提交，那么这次提交仅存在于本地的 Git 而非 Subversion 服务器上。

$ 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(-)

接下来，可以将作出的修改推送到上游。值得注意的是，Subversion 的使用流程也因此改变了——你可以在离线状态下进行多次提交然后一次性的推送到 Subversion 的服务器上。向 Subversion 服务器推送的命令是 git svn dcommit：

$ 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

所有在原 Subversion 数据基础上提交的 commit 会一一提交到 Subversion，然后你本地 Git 的 commit 将被重写，加入一个特别标识。这一步很重要，因为它意味着所有 commit 的 SHA-1 指都会发生变化。这也是同时使用 Git 和 Subversion 两种服务作为远程服务不是个好主意的原因之一。检视以下最后一个 commit，你会找到新添加的 git-svn-id （译注：即本段开头所说的特别标识）：

$ 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

注意看，原本以 97031e5 开头的 SHA-1 校验值在提交完成以后变成了 938b1a5 。如果既要向 Git 远程服务器推送内容，又要推送到 Subversion 远程服务器，则必须先向 Subversion 推送（dcommit），因为该操作会改变所提交的数据内容。

拉取最新进展

如果要与其他开发者协作，总有那么一天你推送完毕之后，其他人发现他们推送自己修改的时候（与你推送的内容）产生冲突。这些修改在你合并之前将一直被拒绝。在 git svn 里这种情况形似：

$ 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

为了解决该问题，可以运行 git svn rebase ，它会拉取服务器上所有最新的改变，再次基础上衍合你的修改：

$ git svn rebase

M README.txt

r80 = ff829ab914e8775c7c025d741beb3d523ee30bc4 (trunk)

First, rewinding head to replay your work on top of it...

Applying: first user change

现在，你做出的修改都发生在服务器内容之后，所以可以顺利的运行 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

需要牢记的一点是，Git 要求我们在推送之前先合并上游仓库中最新的内容，而 git svn 只要求存在冲突的时候才这样做。假如有人向一个文件推送了一些修改，这时你要向另一个文件推送一些修改，那么 dcommit 将正常工作：

$ 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.

这一点需要牢记，因为它的结果是推送之后项目处于一个不完整存在与任何主机上的状态。如果做出的修改无法兼容但没有产生冲突，则可能造成一些很难确诊的难题。这和使用 Git 服务器是不同的——在 Git 世界里，发布之前，你可以在客户端系统里完整的测试项目的状态，而在 SVN 永远都没法确保提交前后项目的状态完全一样。

即使还没打算进行提交，你也应该用这个命令从 Subversion 服务器拉取最新修改。sit svn fetch 能获取最新的数据，不过 git svn rebase 才会在获取之后在本地进行更新 。

$ 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.

不时地运行一下 git svn rebase 可以确保你的代码没有过时。不过，运行该命令时需要确保工作目录的整洁。如果在本地做了修改，则必须在运行 git svn rebase 之前或暂存工作，或暂时提交内容——否则，该命令会发现衍合的结果包含着冲突因而终止。

Git 分支问题

习惯了 Git 的工作流程以后，你可能会创建一些特性分支，完成相关的开发工作，然后合并他们。如果要用 git svn 向 Subversion 推送内容，那么最好是每次用衍合来并入一个单一分支，而不是直接合并。使用衍合的原因是 Subversion 只有一个线性的历史而不像 Git 那样处理合并，所以 git svn 在把快照转换为 Subversion 的 commit 时只能包含第一个祖先。

假设分支历史如下：创建一个 experiment 分支，进行两次提交，然后合并到 master 。在 dcommit 的时候会得到如下输出：

$ 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

在一个包含了合并历史的分支上使用 dcommit 可以成功运行，不过在 Git 项目的历史中，它没有重写你在experiment 分支中的两个 commit ——另一方面，这些改变却出现在了 SVN 版本中同一个合并 commit 中。

在别人克隆该项目的时候，只能看到这个合并 commit 包含了所有发生过的修改；他们无法获知修改的作者和时间等提交信息。

Subversion 分支

Subversion 的分支和 Git 中的不尽相同；避免过多的使用可能是最好方案。不过，用 git svn 创建和提交不同的 Subversion 分支仍是可行的。

创建新的 SVN 分支

要在 Subversion 中建立一个新分支，需要运行 git svn branch [分支名] ：

$ 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)

这相当于在 Subversion 中的 svn copy trunk branches/opera 命令，并会对 Subversion 服务器进行相关操作。值得注意的是它没有检出和转换到那个分支；如果现在进行提交，将提交到服务器上的 trunk， 而非 opera。

切换当前分支

Git 通过搜寻提交历史中 Subversion 分支的头部来决定 dcommit 的目的地——而它应该只有一个，那就是当前分支历史中最近一次包含 git-svn-id 的提交。

如果需要同时在多个分支上提交，可以通过导入 Subversion 上某个其他分支的 commit 来建立以该分支为 dcommit目的地的本地分支。比如你想拥有一个并行维护的 opera 分支，可以运行

$ git branch opera remotes/opera

然后，如果要把 opera 分支并入 trunk （本地的 master 分支），可以使用普通的 git merge。不过最好提供一条描述提交的信息（通过 -m），否则这次合并的记录是 Merge branch opera ，而不是任何有用的东西。

记住，虽然使用了 git merge 来进行这次操作，并且合并过程可能比使用 Subversion 简单一些（因为 Git 会自动找到适合的合并基础），这并不是一次普通的 Git 合并提交。最终它将被推送回 commit 无法包含多个祖先的 Subversion 服务器上；因而在推送之后，它将变成一个包含了所有在其他分支上做出的改变的单一 commit。把一个分支合并到另一个分支以后，你没法像在 Git 中那样轻易的回到那个分支上继续工作。提交时运行的 dcommit 命令擦除了全部有关哪个分支被并入的信息，因而以后的合并基础计算将是不正确的—— dcommit 让 git merge 的结果变得类似于 git merge --squash。不幸的是，我们没有什么好办法来避免该情况—— Subversion 无法储存这个信息，所以在使用它作为服务器的时候你将永远为这个缺陷所困。为了不出现这种问题，在把本地分支（本例中的opera）并入 trunk 以后应该立即将其删除。

对应 Subversion 的命令

git svn 工具集合了若干个与 Subversion 类似的功能，对应的命令可以简化向 Git 的转化过程。下面这些命令能实现 Subversion 的这些功能。

SVN 风格的历史

习惯了 Subversion 的人可能想以 SVN 的风格显示历史，运行 git svn log 可以让提交历史显示为 SVN 格式：

$ 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

关于 git svn log ，有两点需要注意。首先，它可以离线工作，不像 svn log 命令，需要向 Subversion 服务器索取数据。其次，它仅仅显示已经提交到 Subversion 服务器上的 commit。在本地尚未 dcommit 的 Git 数据不会出现在这里；其他人向 Subversion 服务器新提交的数据也不会显示。等于说是显示了最近已知 Subversion 服务器上的状态。

SVN 日志

类似 git svn log 对 git log 的模拟，svn annotate 的等效命令是 git svn blame [文件名]。其输出如下：

$ 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

同样，它不显示本地的 Git 提交以及 Subversion 上后来更新的内容。

SVN 服务器信息

还可以使用 git svn info 来获取与运行 svn info 类似的信息：

$ 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)

它与 blame 和 log 的相同点在于离线运行以及只更新到最后一次与 Subversion 服务器通信的状态。

略 Subversion 之所略

假如克隆了一个包含了 svn:ignore 属性的 Subversion 仓库，就有必要建立对应的 .gitignore 文件来防止意外提交一些不应该提交的文件。git svn 有两个有益于改善该问题的命令。第一个是 git svn create-ignore，它自动建立对应的 .gitignore 文件，以便下次提交的时候可以包含它。

第二个命令是 git svn show-ignore，它把需要放进 .gitignore 文件中的内容打印到标准输出，方便我们把输出重定向到项目的黑名单文件：

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

这样一来，避免了 .gitignore 对项目的干扰。如果你是一个 Subversion 团队里唯一的 Git 用户，而其他队友不喜欢项目包含 .gitignore，该方法是你的不二之选。

Git-Svn 总结

git svn 工具集在当前不得不使用 Subversion 服务器或者开发环境要求使用 Subversion 服务器的时候格外有用。不妨把它看成一个跛脚的 Git，然而，你还是有可能在转换过程中碰到一些困惑你和合作者们的迷题。为了避免麻烦，试着遵守如下守则：

* 保持一个不包含由 git merge 生成的 commit 的线性提交历史。将在主线分支外进行的开发通通衍合回主线；避免直接合并。
* 不要单独建立和使用一个 Git 服务来搞合作。可以为了加速新开发者的克隆进程建立一个，但是不要向它提供任何不包含 git-svn-id 条目的内容。甚至可以添加一个 pre-receive 挂钩来在每一个提交信息中查找git-svn-id 并拒绝提交那些不包含它的 commit。

如果遵循这些守则，在 Subversion 上工作还可以接受。然而，如果能迁徙到真正的 Git 服务器，则能为团队带来更多好处。