返回 码云 首页(目录) 章节列表 ▾ Pro Git（中文版）

8 Git 与其他系统

世界不是完美的。大多数时候，将所有接触到的项目全部转向 Git 是不可能的。有时我们不得不为某个项目使用其他的版本控制系统（ VCS, Version Control System ），其中比较常见的是 Subversion 。你将在本章的第一部分学习使用 `git svn` ，Git 为 Subversion 附带的双向桥接工具。

或许现在你已经在考虑将先前的项目转向 Git 。本章的第二部分将介绍如何将项目迁移到 Git：先介绍从 Subversion 的迁移，然后是 Perforce，最后介绍如何使用自定义的脚本进行非标准的导入。

8.1 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。

8.2 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

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

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：

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

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
```

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 的 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。

8.3 总结

现在的你应该掌握了在 Subversion 上使用 Git 以及把几乎任何先存仓库无损失的导入为 Git 仓库。下一章将介绍 Git 内部的原始数据格式，从而是使你能亲手锻造其中的每一个字节，如果必要的话。

下一节上一节首页（目录）| 返回 码云