Лекция 8. Git и другие системы управления версиями

Лекция 8. Git и другие системы управления версиями

Наш мир несовершенен. Как правило, вы не сможете моментально перевести любой проект, в котором вы участвуете, на использование Git. Иногда вам придется иметь дело с проектами, использующими другую систему контроля версий, и, в большинстве случаев, этой системой будет Subversion. Первая часть этого раздела научит вас обращаться с git svn — встроенным в Git двухсторонним интерфейсом обмена с Subversion.

В какой-то момент, вы, возможно, захотите перевести свой существующий проект на Git. Вторая часть раздела расскажет о том, как провести миграцию: сначала с Subversion, потом с Perforce, и наконец, с помощью написания собственного сценария для нестандартных вариантов миграции.

Git и Subversion

В настоящее время большинство проектов с открытым исходным кодом, а также большое число корпоративных проектов, используют Subversion для управления своим исходным кодом. Это самая популярная на текущий момент система управления версиями с открытым исходным кодом, история её использования насчитывает около 10 лет. Кроме того, она очень похожа на CVS, систему, которая была самой популярной до Subversion.

Одна из замечательных особенностей Git — возможность двустороннего обмена с Sub- version через интерфейс, называемый git svn. Этот инструмент позволяет вам использовать Git в качестве корректного клиента при работе с сервером Subversion. Таким образом, вы можете пользоваться всеми локальными возможностями Git, а затем сохранять изменения на сервере Subversion так, как если бы использовали Subversion локально. То есть вы можете делать локальное ветвление и слияние, использовать индекс, перемещение и отбор патчей для переноса из одной ветви в другую (cherry-picking) и т.д., в то время, как ваши коллеги будут продолжать использовать в разработке подход времён каменного века. Это хороший способ протащить Git в рабочее окружение своей компании, чтобы помочь коллегам разработчикам стать более эффективными, в то время как вы будете лоббировать переход полностью на Git. Интерфейс обмена с Subversion это ворота в мир распределённых систем управления версиями.

git svn

Базовой командой в Git для всех команд работающих с мостом к Subversion является git svn. Ей предваряется любая команда. Она принимает довольно порядочное число команд, поэтому мы изучим из них, те которые наиболее часто используются, рассмотрев несколько небольших вариантов работы. Важно отметить, что при использовании git svn, вы

взаимодействуете с Subversion системой, которая намного менее «продвинута», чем Git. Хоть вы и умеете с лёгкостью делать локальное ветвление и слияние, как правило лучше всего держать свою историю в как можно более линейном виде, используя перемещения (rebase) и избегая таких вещей, как одновременный обмен с удалённым репозиторием Git.

Не переписывайте свою историю, попробуйте отправить изменения ещё раз, а также не отправляйте изменения в параллельный Git-репозиторий, используемый для совместной работы, одновременно с другими разработчиками, использующими Git. Subversion может иметь только одну единственную линейную историю изменений, сбить с толку которую очень и очень просто. Если вы работаете в команде, в которой некоторые разработчики используют Git, а другие Subversion, убедитесь, что для совместной работы все используют только SVN-сервер — это сильно упростит вам жизнь.

Настройка

Для того, чтобы попробовать этот функционал в действии, вам понадобится доступ с правами на запись к обычному SVN-репозиторию. Если вы хотите повторить рассматриваемые примеры, вам нужно сделать доступную на запись копию моего тестового репозитория. Это можно сделать без труда с помощью утилиты svnsync, входящей в состав последних версий Subversion (по крайней мере после версии 1.4). Для этих примеров, я создал новый Subversion- репозиторий на Google Code, который был частичной копией проекта protobuf(утилита шифрования структурированных данных для их передачи по сети).

Чтобы мы могли продолжить, прежде всего создайте новый локальный репозиторий Sub- version:

```
$ mkdir /tmp/test-svn

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

Затем разрешите всем пользователям изменять revprops — самым простым способом сделать это будет добавление сценария pre-revprop-change, который просто всегда завершается с кодом 0:

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

#!/bin/sh
 exit 0;

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

Теперь вы можете синхронизировать проект со своей локальной машиной, вызвав svn-sync init с параметрами задающими исходный и целевой

репозиторий:

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

Эта команда подготовит процесс синхронизации. Затем склонируйте код выполнив:

```
$ 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.

                                          ...
```

Хотя выполнение этой операции и может занять всего несколько минут, однако, если вы попробуете скопировать исходный репозиторий в другой удалённый репозиторий, а не в локальный, то процесс займёт почти час, хотя в этом проекте менее ста коммитов. Subversion вынужден клонировать ревизии по одной, а затем отправлять их в другой репозиторий — это чудовищно неэффективно, однако это единственный простой способ выполнить это действие.

Приступим к работе

Теперь, когда в вашем распоряжении имеется SVN-репозиторий, для которого вы имеете право на запись, давайте выполним типичные действия по работе с СУВ. Начнём с команды git svn clone, которая импортирует весь SVN-репозиторий в локальный Git-репозиторий. Помните, что если вы производите импорт из настоящего удалённого SVN-репозитория, вам надо заменить file:///tmp/test-svn на реальный адрес вашего SVN-репозитория:

```
          $ 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
```

Эта команда эквивалентна выполнению для указанного вами URL двух команд — git svn init, а затем git svn fetch. Процесс может занять некоторое время. Тестовый проект имеет всего лишь около 75 коммитов, и кода там не очень много, так что скорее всего, вам придётся подождать всего несколько минут. Однако, Git должен по отдельности проверить и выполнить коммит для каждой версии. Для проектов, имеющих историю с сотнями и тысячами изменений, этот процесс может занять несколько часов или даже дней.

Часть команды -T trunk -b branches -t tagsсообщает Git, что этот SVN- репозиторий следует стандартным соглашениям о ветвлении и метках. Если вы используете не стандартные имена: trunk, branches и tags, а какие-то другие, то должны изменить эти параметры соответствующим образом. В связи с тем, что такие соглашения являются общепринятыми, вы можете использовать короткий формат, заменив всю эту часть на -s, заменяющую собой все эти параметры. Следующая команда эквивалента предыдущей:

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

Таким образом, вы должны были получить корректный Git-репозиторий с импортированными ветками и метками:

```
$ 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
```

Важно отметить, что эта утилита именует ваши ссылки на удалённые ресурсы по-другому. Когда вы клонируете обычный репозиторий Git, вы получаете все ветки с удалённого сервера на локальном компьютере в виде: origin/[branch] — в пространстве имён с именем удалённого сервера. Однако, git svnполагает, что у вас не будет множества удалённых источников данных и сохраняет все ссылки на всякое, находящееся на удалённом сервере, без пространства имён. Для просмотра всех имён ссылок вы можете использовать служебную команду Git show-ref:

```
$ 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
```

Обычный Git-репозиторий выглядит скорее так:

```
$ git show-ref

83e38c7a0af325a9722f2fdc56b10188806d83a1 refs/heads/master
```

```
3e15e38c198baac84223acfc6224bb8b99ff2281    refs/remotes/gitserver/master
 0a30dd3b0c795b80212ae723640d4e5d48cabdff    refs/remotes/origin/master
25812380387fdd55f916652be4881c6f11600d6f    refs/remotes/origin/testing
```

Здесь два удалённых сервера: один с именем gitserver и веткой master, и другой с именем originc двумя ветками: masterи testing.

Обратите внимание, что в примере, где ссылки импортированы из git svn, метки добавлены так, как будто они являются ветками, а не так, как настоящие метки в Git. Импортированные из Subversion данные выглядят так, как будто под именами меток с удалённого ресурса скрываются ветки.

Коммит в Subversion

Теперь, когда у вас есть рабочий репозиторий, вы можете выполнить какую-либо работу с кодом и выполнить коммит в апстрим, эффективно используя Git в качестве клиента SVN. Если вы редактировали один из файлов и закоммитили его, то вы внесли изменение в локальный репозиторий Git, которое пока не существует на сервере Subversion:

```
$ 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(-)
```

После этого, вам надо отправить изменения в апстрим. Обратите внимание, как Git изменяет способ работы с Subversion — вы можете сделать несколько коммитов оффлайн, а затем отправить их разом на сервер Subversion. Для передачи изменений на сервер Subversion требуется выполнить команду git svn dcommit:

```
$ git svn dcommit

      M       README.txt

Committed r79  README.txt
r79 = 938b1a547c2cc92033b74d32030e86468294a5c8 (trunk)

No changes between current HEAD and refs/remotes/trunk
```

Это действие возьмёт все коммиты, сделанные поверх того, что есть в SVN-репозитории, выполнит коммит в Subversion для каждого из них, а затем перепишет ваш локальный коммит в Git, чтобы добавить к нему уникальный идентификатор. Это важно, поскольку это означает, что изменятся все SHA-1 контрольные суммы ваших коммитов. В частности и поэтому работать с одним и тем же проектом одновременно и через Git, и через

Subversion это плохая идея. Взглянув на последний коммит, вы увидите, что добавился новый git-svn-id:

```
$ 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
```

Обратите внимание — контрольная сумма SHA, которая начиналась с 97031e5 когда вы делали коммит, теперь начинается с 938b1a5. Если вы хотите отправить изменения как на Git-сервер, так и на SVN-сервер, вы должны отправить их (dcommit) сначала на сервер Subversion, поскольку это действие изменит отправляемые данные.

Получение новых изменений

Если вы работаете вместе с другими разработчиками, значит когда-нибудь вам придётся столкнуться с ситуацией, когда кто-то из вас отправит изменения на сервер, а другой, в свою очередь, будет пытаться отправить свои изменения, конфликтующие с первыми. Это изменение не будет принято до тех пор, пока вы не сольёте себе чужую работу. В git svnэта ситуация выглядит следующим образом:

```
$ 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
```

Для разрешения этой проблемы, вам нужно выполнить команду git svn rebase, которая получит все изменения, имеющиеся на сервере, которых ещё нет на вашей локальной машине и переместит все ваши недавние изменения поверх того, что было на сервере:

```
$ git svn rebase

    M      README.txt

  r80 = ff829ab914e8775c7c025d741beb3d523ee30bc4 (trunk)
First, rewinding head to replay your work on top of it...
        Applying: first user change
```

Теперь все ваши изменения находятся сверху того, что есть на SVN-сервере, так что вы можете спокойно выполнить 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
```

Следует помнить, что в отличие от Git'a, который требует сливать себе изменения в апстриме, которых у вас ещё нет локально, перед тем как отправить свои изменения, git svn заставляет делать такое только в случае конфликта правок. Если кто-либо внесёт изменения в один файл,

а вы внесёте изменения в другой, команда dcommitcработает без ошибок:

```
$ git svn dcommit

M      configure.ac

Committed r84 autogen.sh
r83 = 8aa54a74d452f82eee10076ab2584c1fc424853b (trunk)

M      configure.ac

r84 = cdbac939211ccb18aa744e581e46563af5d962d0 (trunk)

W: d2f23b80f67aaaa1f6f5aaef48fce3263ac71a92 and refs/remotes/trunk differ, \
using rebase:

:100755 100755 efa5a59965fbbb5b2b0a12890f1b351bb5493c18 \
```

Это важно помнить, поскольку последствием этих действий может стать такое состояние проекта, которого нет ни на одном из ваших компьютеров. Если изменения несовместимы, но не ведут к конфликту изменений у вас могут возникнуть проблемы, которые трудно будет диагностировать. Это отличается от работы с Git-сервером — в Git вы можете полностью проверить состояние проекта на клиентских машинах до публикации, в то время, как в SVN вы не можете даже быть уверены в том, что состояние проекта непосредственно перед коммитом и после него идентично.

Кроме того, вам нужно выполнить следующую команду для получения изменений с сервера Subversion, даже если вы не готовы сами сделать коммит. Вы можете выполнить git svn fetch для получения новых данных, но git svn rebase и извлечёт новые данные с сервера, и обновит ваши локальные коммиты.

```
$ git svn rebase
       M       generate_descriptor_proto.sh
  r82 = bd16df9173e424c6f52c337ab6efa7f7643282f1 (trunk)

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

Выполняйте команду git svn rebase периодически, чтобы быть уверенным в том, что ваш код имеет самую свежую версию. Перед выполнением этой команды убедитесь, что ваш рабочий каталог чист. Если нет, вы должны либо «спрятать» свои изменения, либо временно закоммитить их перед выполнением git svn rebase, иначе выполнение этой команды прекратится, если она обнаружит возникновение конфликта слияния.

Проблемы с ветвлением в Git

После того как вы привыкли к работе с Git, вы наверняка будете создавать ветки для работы над отдельными задачами, а затем сливать их. Если вы отправляете изменения на сервер Subversion через git svn, вам скорее всего потребуется перемещать свою работу каждый раз в одну ветку, а не сливать ветки вместе. Причина, по которой предпочтение должно быть отдано именно такому подходу, заключается в том, что Subversion имеет линейную историю изменений и не может обрабатывать слияния так, как это делает Git. Таким образом git svn проходит только по первым родителям при конвертации снимков состояния в коммиты Sub- version.

Допустим, что история изменений выглядит следующим образом: вы создали ветку experiment, сделали два коммита, а затем слили их в ветку master. Если вы выполните dcommit, результат будет следующим:

```
              $ git svn dcommit

       M      CHANGES.txt

Committed r85  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

       M      COPYING.txt

       M

Committed r86  INSTALL.txt
          M
r86 = 2647f6b86ccfcaad4ec58c520e369ec81f7c283c (trunk)

No changes between current HEAD and refs/remotes/trunk
```

Выполнение dcommitдля ветки с объединённой историей не вызовет никаких проблем. Однако, если вы посмотрите на историю проекта в Git, то увидите, что ни один из коммитов, которые вы сделали в ветке experimentне были переписаны — вместо этого, все эти изменения появятся в SVN версии как один объединённый коммит.

Когда кто-нибудь склонирует себе эту работу, всё, что он увидит — это коммит, в котором все изменения слиты воедино; он не увидит данных о том, откуда они взялись и когда они были внесены.

Ветвление в Subversion

Работа с ветвями в Subversion отличается от таковой в Git; если у вас есть возможность избегать её, то это наверное лучший вариант. Хотя, вы можете создавать и вносить изменения в ветки Subversion используя git svn.

Создание новой ветки в SVN Для того, чтобы создать новую ветку в Subversion, выполните

git svn branch [имя ветки]:

```
                              $ 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)
```

Эта команда эквивалентна команде Subversion svn copy trunk branches/opera и выполняется на сервере Subversion. Важно отметить, что эта команда не переключает вас на указанную ветку. Так что, если вы сейчас сделаете коммит, он попадёт на сервере в trunk, а не в opera.

Переключение активных веток

Git определяет ветку, куда вносятся ваши коммиты, путём выбора самой последней Subversion- ветки в вашей истории — она должна быть единственной и она должна быть последней в текущей истории веток, имеющей метку git-svn-id.

Если вы хотите работать одновременно с несколькими ветками, вы можете настроить локальные ветки на внесение изменений через dcommit в конкретные ветки Subversion, начиная их на основе импортированного SVN-коммита для нужной ветки. Если вам нужна ветка opera, в которой вы можете поработать отдельно, можете выполнить:

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

Теперь, если вы захотите слить ветку opera в trunk (вашу ветку master), вы можете сделать это с помощью обычной команды git merge. Однако вам потребуется добавить подробное описание к коммиту (через параметр -m), иначе при слиянии комментарий будет иметь вид «Merge branch opera», вместо чего-нибудь полезного.

Помните, что хотя вы и используете git merge для этой операции, и слияние скорее всего произойдёт намного проще, чем было бы в Subversion (потому, что Git автоматически определяет подходящую основу для слияния), это не является обычным коммитом-слиянием Git. Вы должны передать данные обратно на сервер Subversion, который не способен справиться с коммитом, имеющим более одного родителю, так что после передачи этот коммит будет выглядеть как один коммит, в который затолканы все изменения с другой ветки. После того, как вы сольёте одну ветку в другую, вы не сможете просто так вернуться к работе над ней, как вы могли бы в Git. Команда dcommit удаляет всю информацию о том, какая ветка была влита, так что последующие вычисления базы слияния будут неверными — команда dcommit сделает результаты выполнения git mergeтакими же, какими они были бы после выполнения git merge --squash. К сожалению, избежать подобной ситуации вряд ли удастся — Subver- sion не способен сохранять подобную информацию, так что вы всегда будете связаны этими ограничениями. Во избежание проблем вы должны удалить локальную ветку (в нашем случае opera) после того, как вы вольёте её в trunk.

Команды Subversion

Набор утилит git svn предоставляет в ваше распоряжение несколько команд для облегчения перехода на Git, путём предоставления функциональности, подобной той, которую вы имеете

в Subversion. Ниже приведены несколько команд, которые дают вам то, что вы имели в Sub- version.

Просмотр истории в стиле SVN Если вы привыкли к Subversion и хотите просматривать историю в стиле SVN, выполните команду git svn log, чтобы увидеть историю коммитов в формате таком же как в SVN:

```
$ 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
```

Вы должны знать две важные вещи о команде git svn log. Во-первых, она работает в оффлайне, в отличие от оригинальной команды svn log, которая запрашивает информацию с сервера Subversion. Во-вторых, эта команда отображает только те коммиты, которые были переданы на сервер Subversion. Локальные коммиты Git, которые вы ещё не отправили с помощью dcommit не будут отображаться, равно как и коммиты, отправленные на сервер Subversion другими людьми с момента последнего выполнения dcommit. Результат действия этой команды скорее похож на последнее известное состояние изменений на сервере Subver- sion.

SVN-Аннотации Так же как команда git svn log симулирует в оффлайне команду svn log, эквивалентом команды svn annotate является команда git svn blame [ФАЙЛ]. Её вывод выглядит следующим образом:

```
$ 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:
```

Опять же, эта команда не показывает коммиты, которые вы сделали локально в Git или те, которые за то время были отправлены на Subversion-сервер.

Информация о SVN-сервере Вы можете получить ту же информацию, которую даёт выполнение команды svn info, выполнив команду 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)
```

Так же, как blame и log, эта команда выполняется оффлайн и выводит информацию, актуальную на момент последнего вашего обращения к серверу Subversion.

Игнорирование того, что игнорирует Subversion Если вы клонируете репозиторий Subver- sion, в котором где-то установлены свойства svn:ignore, скорее всего вы захотите создать соответствующие им файлы .gitignore, чтобы ненароком не добавить в коммит те файлы, которые не стоит добавлять. Для решения этой проблемы в git svn имеется две команды. Первая — git svn create-ignore— автоматически создаст соответствующие файлы .gitignore, и затем вы можете добавить их в свой следующий коммит.

Вторая команда — git svn show-ignore, которая выводит на стандартный вывод строки, которые вы должны включить в файл .gitignore. Таким образом вы можете перенаправить вывод этой команды в файл исключений вашего проекта:

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

Поступая таким образом, вы не захламляете проект файлами .gitignore. Это правильный подход, если вы являетесь единственным пользователем Git в команде, использующей Subversion, и ваши коллеги выступают против наличия файлов .gitignoreв проекте.

Заключение по Git-Svn

Утилиты git svn полезны в том случае, если ваша разработка по каким-то причинам требует наличия рабочего сервера Subversion. Однако, вам стоит смотреть на Git использующий мост в Subversion как на урезанную версию Git. В противном случае вы столкнётесь с проблемами в преобразованиях, которые могут сбить с толку вас и ваших коллег. Чтобы избежать неприятностей, старайтесь следовать следующим рекомендациям:

•Держите историю в Git линейной, чтобы она не содержала коммитов-слияний, сделанных с помощью git merge. Перемещайте всю работу, которую вы выполняете вне основной ветки обратно в неё; не выполняйте слияний.

•Не устанавливайте отдельный Git-сервер для совместной работы. Можно иметь один такой сервер для того, чтобы ускорить клонирование для новых разработчиков, но не отправляйте на него ничего, не имеющего записи git-svn-id. Возможно, стоит даже добавить перехватчик pre-receive, который будет проверять каждый коммит на наличие git-svn-id и отклонять git push, если коммиты не имеют такой записи.

При следовании этим правилам, работа с сервером Subversion может быть более-менее сносной. Однако, если возможен перенос проекта на реальный сервер Git, преимущества от этого перехода дадут вашему проекту намного больше.

Миграция на Git

Если вы решили начать использовать Git, а у вас уже есть база исходного кода в другой СУВ, вам придётся как-то мигрировать свой проект. Этот раздел описывает некоторые из инструментов для импортирования проектов, включённых в состав Git для самых распространённых систем, в конце описывается создание вашего собственного инструмента для импортирования.

Импортирование

Вы научитесь импортировать данные из двух самых популярных систем контроля версий Subversion и Perforce — поскольку они охватывают большинство пользователей,

которые переходят на Git, а также потому, что для обеих систем созданы высококлассные инструменты, которые поставляются в составе Git.

Subversion

Если прочли предыдущий раздел об использовании git svn, вы можете с лёгкостью использовать все инструкции, имеющиеся там для клонирования репозитория через git svn clone. Затем можете отказаться от использования сервера Subversion и отправлять изменения на новый Git-сервер, и использовать уже его. Вытащить историю изменений, можно так же быстро, как получить данные с сервера Subversion (что, однако, может занять какое-то время). Однако, импортирование не будет безупречным. И так как оно занимает много времени, стоит сделать его правильно. Первая проблема это информация об авторах. В Subversion каждый коммитер имеет свою учётную запись в системе, и его имя пользователя отображается

в информации о коммите. В примерах из предыдущего раздела выводилось schacon в некоторых местах, например, в выводе команд blameи git svn log. Если вы хотите преобразовать эту информацию для лучшего соответствия данным об авторах в Git, вам потребуется отобразить пользователей Subversion в авторов в Git. Создайте файл users.txt, в котором будут содержаться данные об этом отображении в таком формате:

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

Для того, чтобы получить список авторов, который использует SVN, вы можете выполнить следующее:

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

Это даст вам на выходе журнал в формате XML — в нём вы можете просмотреть информацию об авторах, создать из неё список с уникальными записями и избавиться от XML разметки. (Конечно, эта команда сработает только на машине с установленными grep, sort, и perl).

Затем перенаправьте вывод этого скрипта в свой файл users.txt, чтобы потом можно было добавить к каждой записи данные о соответствующих пользователях из Git.

Вы можете передать этот файл как параметр команде git svn, для более точного преобразования данных об авторах. Кроме того, можно дать указание git svn не включать метаданные, обычно импортируемые Subversion, передав параметр --no-metadata команде cloneили init. Таким образом, команда для импортирования будет выглядеть так:

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

Теперь в вашем каталоге my_projectбудут находиться более приятно выглядящие данные после импортирования. Вместо коммитов, которые выглядят так:

```
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
```

они будут выглядеть так:

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

fixed install - go to trunk
```

нет.

Теперь не только поле Author выглядит намного лучше, но и строк с git-svn-idбольше

Вам потребуется сделать небольшую «уборку» после импорта. Сначала вам нужно убрать странные ссылки, оставленные git svn. Сначала мы переставим все метки так, чтобы они были реальными метками, а не странными удалёнными ветками. А затем мы переместим остальные ветки так, чтобы они стали локальными.

Для приведения меток к корректному виду меток Git выполните:

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

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

Эти действия переместят ссылки, которые были удалёнными ветками, начинающимися с tag/и сделают их настоящими (легковесными) метками.

Затем, переместите остальные ссылки в refs/remotes так, чтобы они стали локальными ветками:

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

$ rm -Rf .git/refs/remotes
```

Теперь все старые ветки стали реальными Git-ветками, а все старые метки — реальными метками Git. Последнее, что осталось сделать, это добавить свой Git-сервер в качестве удалённого ресурса и отправить на него данные. Вот пример добавления сервера как удалённого источника:

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

Так как вы хотите, чтобы все ваши ветви и метки были переданы на этот сервер, выполните:

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

Теперь все ваши ветки и метки должны быть импортированы на новый Git-сервер в чистом и опрятном виде.

Perforce

Следующей системой, для которой мы рассмотрим процедуру импортирования, будет Per- force. Утилита импортирования для Perforce также входит в состав Git, но только в секции contribисходного кода — она не доступна по умолчанию, как git svn. Для того, чтобы запустить её, вам потребуется получить исходный код Git, располагающийся на git.kernel.org:

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

$ cd git/contrib/fast-import
```

В каталоге fast-import вы найдёте исполняемый скрипт Python, с названием git- p4. Вы должны иметь на вашем компьютере установленный Python и утилиту p4для того, чтобы эта утилита смогла работать. Допустим, например, что вы импортируете проект Jam из Perforce Public Depot. Для настройки вашей клиентской машины, вы должны установить переменную окружения P4PORT, указывающую на депо Perforce:

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

Запустите команду git-p4 clone для импортирования проекта Jam с сервера Perforce, передав в качестве параметров депо и путь к проекту, а также путь к месту, куда вы хотите импортировать проект:

```
   $ 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%)
```

Если вы теперь перейдёте в каталог /opt/p4import и выполните команду git log, вы увидите импортированную информацию:

```
                                    $ 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]
```

Как видите, в каждом коммите есть идентификатор git-p4. Оставить этот идентификатор будет хорошим решением, если позже вам понадобится узнать номер изменения в Perforce.

Однако, если вы всё же хотите удалить этот идентификатор — теперь самое время это сделать, до того, как вы начнёте работать в новом репозитории. Можно воспользоваться командой git filter-branchдля одновременного удаления всех строк с идентификатором:

```
$ git filter-branch --msg-filter '
          sed -e "/^\[git-p4:/d"

                                          '

Rewrite 1fd4ec126171790efd2db83548b85b1bbbc07dc2 (123/123)
          Ref 'refs/heads/master' was rewritten
```

Если вы теперь выполните git log, то увидите, что все контрольные суммы SHA-1 изменились, и что строки содержащие git-p4больше не появляются в сообщениях коммитов:

```
                                    $ 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
```

Ваш импортируемый репозиторий готов к отправке на новый Git-сервер.

Собственная утилита для импорта

Если вы используете систему отличную от Subversion или Perforce, вы можете поискать утилиту для импорта под свою систему в интернете — для CVS, Clear Case, Visual Source Safe и даже для простого каталога с архивами уже существуют качественные инструменты для импортирования. Если ни один из этих инструментов не подходит для ваших целей, либо если вам нужен больший контроль над процессом импортирования, вам стоит использовать утилиту git fast-import. Эта команда читает простые инструкции со стандартного входа,

управляющие процессом записи специфичных данных в Git. Намного проще создать необходимые объекты в Git используя такой подход, чем запуская базовые команды Git, либо пытаясь записать сырые объекты (см. главу 9). При использовании git fast-import, вы можете создать сценарий для импортирования, который считывает всю необходимую информацию из импортируемой системы и выводит прямые инструкции на стандартный вывод. Затем вы просто запускаете этот скрипт и используя конвейер (pipe) передаёте результаты его работы на вход git fast- import.

Чтобы быстро продемонстрировать суть этого подхода, напишем простую утилиту для импорта. Положим, что вы работаете в каталоге current, и время от времени делаете резервную копию этого каталога добавляя к имени дату — back_YYYY_MM_DD, и вы хотите импортировать это всё в Git. Допустим, ваше дерево каталогов выглядит таким образом:

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

Для того, чтобы импортировать всё это в Git, надо вспомнить, как Git хранит данные. Как вы помните, Git в своей основе представляет собой связный список объектов коммитов, указывающих на снимки состояния их содержимого. Всё, что вам требуется, это сообщить команде fast-import что является данными снимков состояния, какие данные коммитов указывают на них и порядок их следования. Стратегией наших действий будет обход всех снимков состояния по очереди и создание соответствующих коммитов с содержимым каждого каталога, с привязкой каждого коммита к предыдущему.

Так же как и в главе 7 в разделе «Пример создания политики в Git», мы напишем сценарий на Ruby, поскольку это то, с чем я обычно работаю, и кроме того он легко читается. Но вы можете создать его на любом другом языке, которым владеете — он просто

должен выводить необходимую информацию на стандартный вывод. Если вы работаете под Windows, то должны особым образом позаботиться о том, чтобы в конце строк не содержались символы возврата каретки — git fast-import принимает только символ перевода строки (LF), а не символ перевода строки и возврата каретки (CRLF), который используется в Windows.

Для того, чтобы начать, вы должны перейти в целевой каталог и идентифицировать каждый подкаталог, являющийся снимком состояния, который вы хотите импортировать в виде коммита. Основной цикл будет выглядеть следующим образом:

```
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
```

Вы запускаете функцию print_export внутри каждого каталога, она берёт запись и отметку предыдущего снимка состояния и возвращает запись и отметку текущего; таким образом они соединяются нужным образом между собой. «Отметка» — это термин утилиты fast- import, обозначающий идентификатор, который вы даёте коммиту; когда вы создаёте коммиты, вы назначаете каждому из них отметку, которую можно использовать для связывания с другими коммитами. Таким образом, первая операция, которую надо включить в метод print_export, это генерация отметки из имени каталога:

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

Мы сделаем это путём создания массива каталогов и используя значение порядкового номера каталога в массиве, как его отметку, поскольку отметка должна быть целым числом:

```
$marks = []

def convert_dir_to_mark(dir)
   if !$marks.include?(dir)

    $marks << dir
        end

  ($marks.index(dir) + 1).to_s
            end
```

Теперь, когда мы имеем целочисленное представление нашего коммита, нам нужны даты, чтобы указывать их в метаданных коммитов. Поскольку дата записана в имени каталога, мы выделяем её оттуда. Следующей строкой в сценарии print_exportбудет:

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

где метод convert_dir_to_dateопределён как:

```
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
```

Этот метод возвращает целочисленное значение даты для каждого каталога. Последняя часть метаданных, которая нам нужна для всех коммитов это данные о коммитере, которые мы жёстко задаём в глобальной переменной:

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

Теперь мы готовы приступить к выводу данных коммита в своём сценарии импорта. Дадим начальную информацию говорящую, что мы задаём объект коммита, ветку, на которой он находится, затем отметку, которую мы ранее сгенерировали, информацию о коммитере и сообщение коммита, а затем предыдущий коммит, если он есть. Код выглядит следующим образом:

```
 # 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
```

Мы жёстко задаём часовой пояс (-0700), поскольку так проще. Если вы импортируете данные из другой системы, вы должны указать часовой пояс в виде смещения. Сообщение коммита должно быть представлено в особом формате:

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

Формат состоит из слова data, размера данных, которые требуется прочесть, символа переноса строки и, наконец, самих данных. Поскольку нам потребуется использовать такой же формат позже, для описания содержимого файла, создадим вспомогательный метод export_data:

```
def export_data(string)

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

Всё что нам осталось, это описать содержимое файла для каждого снимка состояния.

Это просто, поскольку каждый из них содержится в каталоге: мы можем вывести команду deleteall, за которой следует содержимое каждого файла в каталоге. После этого Git соответствующим образом позаботится о регистрации каждого снимка:

```
puts 'deleteall'

Dir.glob("**/*").each do |file|
    next if !File.file?(file)
        inline_data(file)

end
```

Примечание: поскольку многие системы рассматривают свои ревизии как изменения от одного коммита до другого, fast-import также может принимать команды задающие для каждого коммита, какие файлы были добавлены, удалены или модифицированы, а также что является новым содержимым файлов. В нашем примере вы могли бы вычислить разность между снимками состояния и предоставить только эти данные, но это сложнее. С таким же успехом можно предоставить Git все данные для того, чтобы он сам вычислил разницу. Если с вашими данными проще предоставлять разницу между снимками состояния, обратитесь к странице руководства fast-import для получения подробностей о том как предоставлять данные таким способом.

Формат для задания содержимого нового файла, либо указания нового содержимого изменённого файла следующий:

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

(file contents)
```

Здесь, 644 — это права доступа (если в проекте есть исполняемые файлы, вам надо выявить их и назначить им права доступа 755), а параметр inlineговорит о том, что содержимое будет выводиться непосредственно после этой строки. Метод inline_dataвыглядит следующим образом:

```
def inline_data(file, code = 'M', mode = '644')
           content = File.read(file)

  puts "#{code} #{mode} inline #{file}"
         export_data(content)

end
```

Мы повторно используем метод export_data, определённый ранее, поскольку он работает тут так же, как и при задании сообщений коммитов.

Последнее, что вам осталось сделать, это вернуть текущую отметку, чтобы её можно было передать для использования в следующую итерацию:

```
return mark
```

ПРИМЕЧАНИЕ: Если вы работаете под Windows, то должны убедиться, что добавили ещё один дополнительный шаг. Мы уже упоминали, что Windows использует CRLF для перехода на новую строку, тогда как gitfast-importожидает только LF. Для того, чтобы избежать этой проблемы и сделать процесс импорта безошибочным, вам нужно сказать Ruby использовать LF вместо CRLF:

```
$stdout.binmode
```

Это всё. Если вы теперь запустите этот сценарий, то получите примерно следующее содержимое:

```
$ 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
```

Для того, чтобы запустить утилиту импорта, перенаправьте этот вывод на вход git fast-import, находясь в каталоге Git, в который хотите совершить импортирование. Вы можете создать новый каталог, а затем выполнить в нём git initи потом запустить свой сценарий:

$ 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: Total ob

blobs :

trees : commits: tags :

Total branches:

marks: atoms:

Memory total:

pools: objects:

```
5000
  18 (         1 duplicates                 )
   7 (         1 duplicates          0 deltas)
   6 (         0 duplicates          1 deltas)
   5 (         0 duplicates          0 deltas)
   0 (         0 duplicates          0 deltas)
   1 (         1 loads     )
1024 (         5 unique    )
   3
2255 KiB
2098 KiB
 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

Как видите, после успешного завершения, Git выдаёт большое количество информации о проделанной работе. В нашем случае мы на итог импортировали 18 объектов для 5 коммитов в одну ветку. Теперь выполните git log, чтобы увидеть свою новую историю изменений:

```
$ 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
```

Ну вот, вы получили чистый и красивый Git-репозиторий. Важно отметить, что пока у вас нет никаких файлов в рабочем каталоге — вы должны сбросить свою ветку на ветку master:

```
$ ls

$ git reset --hard master

HEAD is now at 10bfe7d imported from current

$ ls

file.rb  lib
```

С помощью утилиты fast-import можно делать намного больше — манипулировать разными правами доступа, двоичными данными, несколькими ветками, совершать слияния, назначать метки, отображать индикаторы прогресса и многое другое. Некоторое количество примеров более сложных сценариев содержится в каталоге contrib/fast-import исходного кода Git; один из самых лучших из них — сценарий git-p4, о котором я уже рассказывал.

Итоги

После всего вышесказанного, вы должны чувствовать себя уверенно при совместной работе с Git и Subversion и при выполнении импортирования практически любого существующего репозитория в репозиторий Git без потерь данных. Следующая глава раскроет перед вами внутреннюю механику Git, так что вы будете способны создать каждый необходимый байт данных, если потребуется.