Лекция 7. Настройка Git

Лекция 7. Настройка Git

До этого момента мы описывали основы того, как Git работает, и как его использовать. Также мы познакомились с несколькими предоставляемыми Git'ом инструментами, которые делают его использование простым и эффективным. В этой главе мы пройдёмся по некоторым действиям, которые вы можете предпринять, чтобы заставить Git работать в нужной именно вам манере. Мы рассмотрим несколько важных настроек и систему перехватчиков (hook). С их помощью легко сделать так, чтобы Git работал именно так как вам, вашей компании или вашей группе нужно.

Конфигурирование Git

В первой главе вкратце было рассказано, как можно изменить настройки Git с помощью команды git config. Одна из первых вещей, которую мы тогда сделали, это установили свои имя и e-mail адрес:

```
$ git config --global user.name "John Doe"

$ git config --global user.email johndoe@example.com
```

Теперь мы разберём пару более интересных опций, которые вы можете задать тем же образом, чтобы настроить Git под себя.

Мы уже рассмотрели некоторые детали настройки Git в первой главе, но давайте сейчас быстренько пройдёмся по ним снова. Git использует набор конфигурационных файлов для задания желаемого нестандартного поведения. Первым местом, в котором Git ищет заданные параметры, является файл /etc/gitconfig, содержащий значения, действующие для всех пользователей системы и всех их репозиториев. Когда вы передаёте git config опцию --system, происходит чтение или запись именно этого файла.

Следующее место, в которое Git заглядывает, это файл ~/.gitconfig, который для каждого пользователя свой. Вы можете заставить Git читать или писать этот файл, передав опцию --global.

И наконец, Git ищет заданные настройки в конфигурационном файле в Git-каталоге (.git/config) того репозитория, который вы используете в данный момент. Значения оттуда относятся к данному конкретному репозиторию. Значения настроек на новом уровне переписывают значения, заданные на предыдущем уровне. Поэтому, например, значения из .git/config перебивают значения в /etc/gitconfig. Позволяется задавать настройки путём редактирования конфигурационного файла вручную, используя правильный синтаксис, но, как правило, проще воспользоваться командой git config.

Основные настройки клиента

Настройки конфигурации, поддерживаемые Git'ом, можно разделить на две категории: клиентские и серверные. Большинство опций — клиентские, они задают предпочтения в вашей личной работе. Несмотря на то, что опций доступно великое множество, мы рассмотрим только некоторые из них — те, которые широко используются или значительно влияют на вашу работу. Многие опции полезны только в редких случаях, которые мы не будем здесь рассматривать. Если вы хотите посмотреть список всех опций, которые есть в вашем Git'е, выполните:

```
$ git config --help
```

В странице руководства для gitconfigвсе доступные опции описаны довольно подробно. core.editor Для создания и редактирования сообщений коммитов и меток Git по умолчанию использует тот редактор, который установлен текстовым редактором по умолчанию в вашей системе, или, как запасной вариант, редактор Vi. Чтобы сменить это умолчание на что-нибудь другое, используйте настройку core.editor:

```
$ git config --global core.editor emacs
```

Теперь неважно, что установлено в качестве вашего редактора по умолчанию в переменной оболочки, при редактировании сообщений Git будет запускать Emacs.

commit.template Если установить в этой настройке путь к какому-нибудь файлу в вашей системе, Git будет использовать содержимое этого файла в качестве сообщения по умолчанию при коммите. Например, предположим, что вы создали шаблонный файл $HOME/.gitmessage.txt, который выглядит следующим образом:

```
заголовок

что произошло

[карточка: X]
```

Чтобы попросить Git использовать это в качестве сообщения по умолчанию, которое будет появляться в вашем редакторе при выполнении git commit, задайте значение настройки commit.template:

```
$ git config --global commit.template $HOME/.gitmessage.txt

$ git commit
```

После этого, когда во время создания коммита запустится ваш редактор, в нём в качестве сообщения-заглушки будет находиться что-то вроде такого:

```
                                заголовок


что произошло

[карточка: X]
          # Please enter the commit message for your changes. Lines starting
         # with '#' will be ignored, and an empty message aborts the commit.
                              # On branch master
                            # Changes to be committed:
                  #   (use "git reset HEAD <file>..." to unstage)
                                         #
                             # modified:   lib/test.rb
                                         #
```

Если у вас существует определённая политика для сообщений коммитов, то задание шаблона соответствующего этой политике и настройка Git на использование его по умолчанию могут увеличить вероятность того, что этой политики будут придерживаться постоянно. core.pager Настройка core.pager определяет, какой пейджер использовать при постраничном отображении вывода таких команд, как log и diff. Вы можете указать здесь more или свой любимый пейджер (по умолчанию используется less), или можно отключить его, указав пустую строку:

```
$ git config --global core.pager ''
```

Если это выполнить, Git будет выдавать весь вывод полностью для всех команд вне зависимости от того насколько он большой.

user.signingkey Если вы делаете подписанные аннотированные метки (смотри Главу 2), то, чтобы облегчить этот процесс, можно задать свой GPG-ключ для подписи в настройках. Задать ID своего ключа можно так:

```
$ git config --global user.signingkey <id-gpg-ключа>
```

Теперь, чтобы подписать метку, не обязательно каждый раз указывать свой ключ команде git tag:

```
$ git tag -s <имя-метки>
```

core.excludesfile Чтобы Git не видел определённые файлы проекта как неотслеживаемые и не пытался добавить их в индекс при выполнении git add, можно задать для них шаблоны в файл .gitignore, как это описано Главе 2. Однако, если вам необходим другой файл, который будет хранить эти или дополнительные значения вне вашего проекта,

то вы можете указать Git'у расположение такого файла с помощью настройки core.excludesfile. Просто задайте там путь к файлу, в котором написано то же, что пишется в .gitignore.

help.autocorrect Эта опция доступна только в Git 1.6.1 и более поздних. Если вы неправильно наберёте команду в Git 1.6, он выдаст что-то вроде этого:

```
$ git com

git: 'com' is not a git-command. See 'git --help'.

Did you mean this?
        commit
```

Если установить help.autocorrectв 1, Git автоматически запустит нужную команду, если она была единственным вариантом при этом сценарии.

Цвета в Git

Git умеет раскрашивать свой вывод для терминала, что может помочь вам быстрее и легче визуально анализировать вывод. Множество опций в настройках помогут вам установить цвета в соответствии со своими предпочтениями.

color.ui Git автоматически раскрасит большую часть своего вывода, если вы его об этом попросите. Вы можете очень тонко задать, что вы хотите раскрасить и как. Но, чтобы просто включить весь предустановленный цветной вывод для терминала, установите color.uiв true:

```
$ git config --global color.ui true
```

Когда установлено это значение, Git раскрашивает свой вывод в случае, если вывод идёт на терминал. Другие доступные значения это: false, при котором вывод никогда не раскрашивается, и always, при котором цвета добавляются всегда, даже если вы перенаправляете вывод команд Git'а в файл или через конвейер другой команде. Эта настройка появилась в Git версии 1.5.5; если у вас версия старее, вам придётся задать каждую настройку для цвета отдельно.

Вам вряд ли понадобится использовать color.ui = always. В большинстве случаев, если вам нужны коды цветов в перенаправленном выводе, то вы можете просто передать команде флаг --color, чтобы заставить её добавить коды цветов. Настройка color.ui = true— это почти всегда именно то, что вам нужно.

color.* Если вам необходимо более точно задать какие команды и как должны быть раскрашены, или если вы используете старую версию, то в Git есть возможность задать настройки цветов для каждой команды отдельно. Каждая из этих настроек может быть установлена в true, falseили always:

```
   color.branch
    color.diff
color.interactive
  color.status
```

Кроме того, каждая из этих настроек имеет свои поднастройки, которые можно использовать для задания определённого цвета для какой-то части вывода, если вы хотите перезадать цвета. Например, чтобы получить мета-информацию в выводе команды diff в синем цвете с чёрным фоном и жирным шрифтом, выполните

```
$ git config --global color.diff.meta “blue black bold”
```

Цвет может принимать любое из следующих значений: normal, black, red, green, yellow, blue, magenta, cyan и white. Если вы хотите задать атрибут вроде bold, как мы делали в предыдущем примере, то на выбор представлены: bold, dim, ul, blink и reverse.

Если вам это интересно, загляните в страницу руководства для git config, чтобы узнать о всех доступных для конфигурации настройках.

Внешние утилиты merge и diff

Хоть в Git и есть внутренняя реализация diff, которой мы и пользовались до этого момента, вы можете заменить её внешней утилитой. И ещё вы можете установить графическую утилиту для разрешения конфликтов слияния, вместо того, чтобы разрешать конфликты вручную. Мы рассмотрим настройку Perforce Visual Merge Tool (P4Merge) в качестве замены diff и для разрешения конфликтов слияния, потому что это удобная графическая утилита и к тому же бесплатная.

Если вам захотелось её попробовать, то P4Merge работает на всех основных платформах, поэтому проблем с ней быть не должно. В примерах мы будем использовать пути к файлам, которые используются на Mac и Linux; для Windows вам надо заменить /usr/local/binна тот путь к исполняемым файлам, который используется в вашей среде.

Скачать P4Merge можно здесь:

http://www.perforce.com/perforce/downloads/component.html

Для начала сделаем внешние сценарии-обёртки для запуска нужных команд. Я буду использовать Mac’овский путь к исполняемым файлам; для других систем это будет тот путь, куда установлен ваш файл p4merge. Сделайте для слияния сценарий-обёртку с именем extMerge, он будет вызывать бинарник со всеми переданными аргументами:

```
$ cat /usr/local/bin/extMerge

#!/bin/sh

/Applications/p4merge.app/Contents/MacOS/p4merge $*
```

Обёртка для diff проверяет, что ей было передано семь аргументов, и передаёт два из них вашему сценарию для слияния. По умолчанию Git передаёт следующие аргументы программе выполняющей diff:

```
путь старый-файл старый-хеш старые-права новый-файл новый-хеш новые-права
```

Так как нам нужны только старый-файл и новый-файл, воспользуемся сценарием-обёрткой, чтобы передать только те аргументы, которые нам нужны:

```
$ cat /usr/local/bin/extDiff

#!/bin/sh

[ $# -eq 7 ] && /usr/local/bin/extMerge "$2" "$5"
```

Ещё следует убедиться, что наши сценарии имеют права на исполнение:

```
$ sudo chmod +x /usr/local/bin/extMerge

$ sudo chmod +x /usr/local/bin/extDiff
```

Теперь мы можем настроить свой конфигурационный файл на использование наших собственных утилит для разрешения слияний и diff'a. Для этого нам потребуется поменять несколько настроек: merge.tool, чтобы указать Git'у на то, какую стратегию использовать; mergetool.*.cmd, чтобы указать как запустить команду; mergetool.trustExitCode, чтобы указать Git'у, можно ли по коду возврата определить, было разрешение конфликта слияния успешным или нет; и diff.externalдля того, чтобы задать команду используемую для diff. Таким образом вам надо либо выполнить четыре команды git config

```
$ git config --global merge.tool extMerge

$ git config --global mergetool.extMerge.cmd \
  'extMerge "$BASE" "$LOCAL" "$REMOTE" "$MERGED"'

$ git config --global mergetool.trustExitCode false

$ git config --global diff.external extDiff
```

либо отредактировать свой файл ~/.gitconfigи добавить туда следующие строки:

```
[merge]

    tool = extMerge
[mergetool "extMerge"]

  cmd = extMerge "$BASE" "$LOCAL" "$REMOTE" "$MERGED"
                trustExitCode = false

                                    [diff]

                                external = extDiff
```

Если после того, как всё это настроено, вы выполните команду diff следующим образом:

```
$ git diff 32d1776b1^ 32d1776b1
```

то вместо того, чтобы получить вывод команды diff в терминал, Git запустит P4Merge, как это показано на Рисунке 7-1.

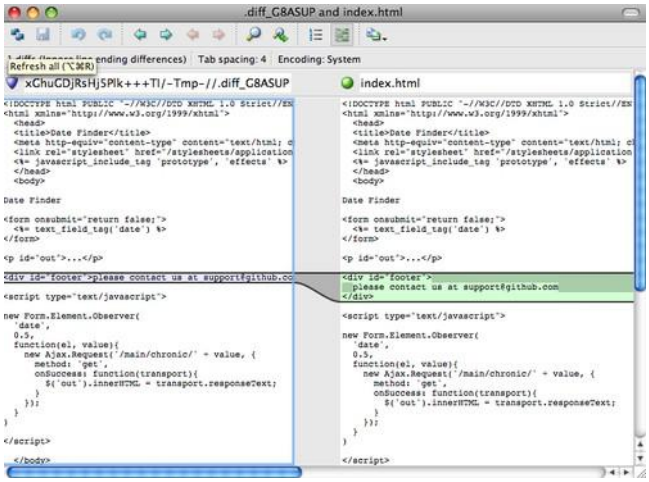


Рисунок 7.1: P4Merge.

Если при попытке слияния двух веток вы получите конфликт, запустите команду git mergetool — она запустит графическую утилиту P4Merge, с помощью которой вы сможете разрешить свои конфликты.

Что удобно в нашей настройке с обёртками, так это то, что вы с лёгкостью можете поменять утилиты для слияния и diff'а. Например, чтобы изменить свои утилиты extDiff и extMerge так, чтобы они использовали утилиту KDiff3, всё, что вам надо сделать, это отредактировать свой файл extMerge:

```
$ cat /usr/local/bin/extMerge

#!/bin/sh

/Applications/kdiff3.app/Contents/MacOS/kdiff3 $*
```

Теперь Git будет использовать утилиту KDiff3 для просмотра diff'ов и разрешения конфликтов слияния.

В Git уже есть предустановленные настройки для множества других утилит для разрешения слияний, для которых вам не надо полностью прописывать команду для запуска, а достаточно просто указать имя утилиты. К таким утилитам относятся: kdiff3, opendiff, tkdiff, meld, xxdiff, emerge, vimdiff и gvimdiff. Например, если вам не интересно использовать KDiff3 для diff'ов, а хочется использовать его только для разрешения слияний, и команда kdiff3 находится в пути, то вы можете выполнить

```
$ git config --global merge.tool kdiff3
```

Если вместо настройки файлов extMerge и extDiff вы выполните эту команду, Git будет использовать KDiff3 для разрешения слияний и обычный свой инструмент diff для diff'ов.

Форматирование и пробельные символы

Проблемы с форматированием и пробельными символами — одни из самых дурацких и трудно уловимых проблем из тех, с которыми сталкиваются многие разработчики при совместной работе над проектами, особенно если разработка ведётся на разных платформах. Очень просто внести малозаметные изменения с помощью пробельных символов при, например, подготовке патчей из-за того, что текстовые редакторы добавляют их без предупреждения, или в кросс- платформенных проектах Windows-программисты добавляют символы возврата каретки в конце изменяемых ими строк. В Git есть несколько опций для того, чтобы помочь с решением подобных проблем.

core.autocrlf Если вы пишите код на Windows или пользуетесь другой системой, но работаете с людьми, которые пишут на Windows, то наверняка рано или поздно столкнётесь с проблемой конца строк. Она возникает из-за того, что Windows использует для переноса строк и символ возврата каретки, и символ перехода на новую строку, в то время как в системах Mac и Linux используется только символ перехода на новую строку. Это незначительное, но невероятно раздражающее обстоятельство при кросс-платформенной работе.

Git может справиться с этим, автоматически конвертируя CRLF-концы строк в LF при коммите и в обратную сторону при выгрузке кода из репозитория на файловую систему. Данную функциональность можно включить с помощью настройки core.autocrlf. Если вы используете Windows, установите настройку в true, тогда концы строк из LF будут сконвертированы в CRLF при выгрузке кода:

```
$ git config --global core.autocrlf true
```

Если вы сидите на Linux или Mac, где используются LF-концы строк, вам не надо, чтобы Git автоматически конвертировал их при выгрузке файлов из репозитория. Однако, если вдруг случайно кто-то добавил файл с CRLF-концами строк, то хотелось бы, чтобы Git исправил это. Можно указать Git’у, чтобы он конвертировал CRLF в LF только при коммитах, установив настройку core.autocrlfв input:

```
$ git config --global core.autocrlf input
```

Такая настройка даст вам CRLF-концы в выгруженном коде на Windows-системах и LF- концы на Mac’ах и Linux, и в репозитории.

Если вы Windows-программист, пишущий проект, предназначенный только для Windows, то можете отключить данную функциональность и записывать символы возврата каретки в репозиторий, установив значение настройки в false:

```
$ git config --global core.autocrlf false
```

core.whitespace Git заранее настроен на обнаружение и исправление некоторых проблем, связанных с пробелами. Он может находить четыре основные проблемы с пробелами — две из них по умолчанию отслеживаются, но могут быть выключены, и две по умолчанию не отслеживаются, но их можно включить.

Те две настройки, которые включены по умолчанию — это trailing-space, которая ищет пробелы в конце строк, и space-before-tab, которая ищет пробелы перед символами табуляции в начале строк.

Те две, которые по умолчанию выключены, но могут быть включены — это indent-with-non-tab, которая ищет строки начинающиеся с восьми или более пробелов вместо символов табуляции, и cr-at-eol, которая сообщает Git'у, что символы возврата каретки в конце строк допустимы.

Вы можете указать Git'у, какие из этих настроек вы хотите включить, задав их в core.whitespace через запятую. Отключить настройку можно либо опустив её в списке, либо дописав знак -перед соответствующим значением. Например, если вы хотите установить все проверки, кроме cr-at-eol, то это можно сделать так:

```
$ git config --global core.whitespace \

trailing-space,space-before-tab,indent-with-non-tab
```

Git будет выявлять эти проблемы при запуске команды git diffи пытаться выделить их цветом так, чтобы можно было их исправить ещё до коммита. Кроме того, эти значения будут использоваться, чтобы помочь с применением патчей с помощью git apply. Когда будете принимать патч, можете попросить Git предупредить вас о наличии в патче заданных проблем с пробельными символами:

```
$ git apply --whitespace=warn <патч>
```

Или же Git может попытаться автоматически исправить проблему перед применением патча:

```
$ git apply --whitespace=fix <патч>
```

Данные настройки также относятся и к команде git rebase. Если вы вдруг сделали коммиты, в которых есть проблемы с пробельными символами, но ещё не отправили их на сервер, запустите rebasec опцией --whitespace=fix, чтобы Git автоматически исправил ошибки во время переписывания патчей.

Настройка сервера

Для серверной части Git доступно не так уж много настроек, но среди них есть несколько интересных, на которые следует обратить внимание.

receive.fsckObjects По умолчанию Git не проверяет все отправленные на сервер объекты на целостность. Хотя Git и может проверять, что каждый объект всё ещё совпадает со своей контрольной суммой SHA-1 и указывает на допустимые объекты, по умолчанию Git не делает этого при каждом запуске команды push. Эта операция довольно затратна и может значительно увеличить время выполнения git pushв зависимости от размера репозитория и количества отправляемых данных. Если вы хотите, чтобы Git проверял целостность объектов при каждой отправке данных, сделать это можно установив receive.fsckObjectsв true:

```
$ git config --system receive.fsckObjects true
```

Теперь Git, перед тем как принять новые данные от клиента, будет проверять целостность вашего репозитория, чтобы убедиться, что какой-нибудь неисправный клиент не внёс повреждённые данные.

receive.denyNonFastForwards Если вы переместили с помощью команды rebase уже отправленные на сервер коммиты, и затем пытаетесь отправить их снова, или, иначе, пытаетесь отправить коммит в такую удалённую ветку, которая не содержит коммит, на который на текущий момент указывает удалённая ветка — вам будет в этом отказано. Обычно это хорошая стратегия. Но в случае если вы переместили коммиты, хорошо понимая зачем это вам нужно, вы можете вынудить Git обновить удалённую ветку передав команде pushфлаг -f.

Чтобы отключить возможность принудительного обновления веток, задайте receive.denyNonFastForward

```
$ git config --system receive.denyNonFastForwards true
```

Есть ещё один способ сделать это — с помощью перехватчиков, работающих на приём (receive hooks), на стороне сервера, которые мы рассмотрим вкратце позднее. Такой подход позволит сделать более сложные вещи, такие как, например, запрет принудительных обновлений только для определённой группы пользователей.

receive.denyDeletes Один из способов обойти политику denyNonFastForwards — это удалить ветку, а затем отправить новую ссылку на её место. В новых версиях Git’а (начиная с версии 1.6.1) вы можете установить receive.denyDeletesв true:

```
$ git config --system receive.denyDeletes true
```

Этим вы запретите удаление веток и меток с помощью команды pushдля всех сразу — ни один из пользователей не сможет этого сделать. Чтобы удалить ветку на сервере, вам придётся удалить файлы ссылок с сервера вручную. Также есть и другие более интересные способы

добиться этого, но уже для отдельных пользователей с помощью ACL (списков контроля доступа), как мы увидим в конце этой главы.

Git-атрибуты

Некоторые настройки могут быть заданы для отдельных путей, и тогда Git будет применять их только для некоторых подкаталогов или набора файлов. Такие настройки специфичные по отношению к путям называются атрибутами и задаются либо в файле .gitattributes в одном из каталогов проекта (обычно в корне) или в файле .git/info/attributes, если вы не хотите, чтобы файл с атрибутами попал в коммит вместе с остальными файлами проекта.

Использование атрибутов позволяет, например, задать разные стратегии слияния для отдельных файлов или каталогов проекта, или объяснить Git'у, как сравнивать нетекстовые файлы, или сделать так, чтобы Git пропускал данные через фильтр перед тем, как выгрузить или записать данные в репозиторий. В этом разделе мы рассмотрим некоторые из доступных в Git'е атрибутов и рассмотрим несколько практических примеров их использования.

Бинарные файлы

Есть один клёвый трюк, для которого можно использовать атрибуты — можно указать Git'у, какие файлы являются бинарными (в случае если по-другому определить это не получается), и дать ему специальные инструкции о том, как с этими файлами работать. Например, некоторые текстовые файлы могут быть машинными — генерируемыми программой — для них нет смысла вычислять дельты, в то время как для некоторых бинарных файлов получение дельт может быть полезным. Дальше мы увидим, как сказать Git'у, какие файлы какие.

Определение бинарных файлов Некоторые файлы выглядят как текстовые, но по существу должны рассматриваться как бинарные данные. Например, проекты Xcode на Mac'ах содержат файл, оканчивающийся на .pbxproj, который по сути является набором JSON-данных (текстовый формат данных для javascript), записываемым IDE, в котором сохраняются ваши настройки сборки и прочее. Хоть технически это и текстовый файл, потому что содержит только ASCII- символы, но нет смысла рассматривать его как таковой, потому что на самом деле это легковесная база данных — вы не сможете слить её содержимое, если два человека внесут в неё изменение, получение дельт тоже, как правило, ничем вам не поможет. Этот файл предназначается для обработки программой. По сути, лучше рассматривать этот файл как бинарный.

Чтобы заставить Git обращаться со всеми pbxproj файлами как с бинарными, добавьте следующую строку в файл .gitattributes:

```
*.pbxproj -crlf -diff
```

Теперь Git не будет пытаться конвертировать CRLF-концы строк или исправлять проблемы с ними. Также он не будет пытаться получить дельту для изменений в этом файле при запуске git show или git diff в вашем проекте. Начиная с версии 1.6 в Git есть макрос, который означает то же, что и -crlf -diff:

```
*.pbxproj binary
```

Получение дельты для бинарных файлов В Git версии 1.6.x функциональность атрибутов может быть использована для эффективного получения дельт для бинарных файлов. Чтобы сделать это, нужно объяснить Git'у, как сконвертировать ваши бинарные данные в текстовый формат, для которого можно выполнить сравнение с помощью обычного diff.

Документы MS Word Так как эта довольно клёвая функция не особо широко известна, мы рассмотрим несколько примеров её использования. Для начала мы используем этот подход, чтобы решить одну из самых раздражающих проблем известных человечеству: версионный контроль документов Word. Всем известно, что Word это самый ужасающий из всех существующих редакторов, но, как ни странно, все им пользуются. Если вы хотите поместить документы Word

под версионный контроль, вы можете запихнуть их в Git-репозиторий и время от времени делать коммиты. Но что в этом хорошего? Если вы запустите git diff как обычно, то увидите только что-то наподобие этого:

```
$ git diff

diff --git a/chapter1.doc b/chapter1.doc
     index 88839c4..4afcb7c 100644

               Binary files a/chapter1.doc and b/chapter1.doc differ
```

У вас не получится сравнить две версии между собой, только если вы не выгрузите их обе и просмотрите их вручную, так? Оказывается, можно сделать это достаточно успешно, используя атрибуты Git. Поместите следующую строку в свой файл .gitattributes:

```
*.doc diff=word
```

Она говорит Git'у, что все файлы, соответствующие указанному шаблону (.doc) должны использовать фильтр «word» при попытке посмотреть дельту с изменениями. Что такое фильтр «word»? Нам нужно его изготовить. Сейчас мы настроим Git на использование программы strings для конвертирования документов Word в читаемые текстовые файлы, которые Git затем правильно сравнит:

```
$ git config diff.word.textconv strings
```

Этой командой в свой .git/configвы добавите следующую секцию:

```
   [diff "word"]
textconv = strings
```

Замечание: Существуют разные виды .docфайлов. Некоторые из них могут использовать кодировку UTF-16 или могут быть написаны не в латинице, в таких файлах strings не найдёт ничего хорошего. Полезность stringsможет сильно варьироваться. Теперь Git знает, что если ему надо найти дельту между двумя снимками состояния, и какие-то их файлы заканчиваются на .doc, он должен прогнать эти файлы через фильтр «word», который определён как программа strings. Так вы фактически сделаете текстовые версии своих Word-файлов перед тем, как получить для них дельту.

Рассмотрим пример. Я поместил Главу 1 настоящей книги в Git, добавил немного текста в один параграф и сохранил документ. Затем я выполнил git diff, чтобы увидеть, что изменилось:

```
                                    $ git diff

diff --git a/chapter1.doc b/chapter1.doc
      index c1c8a0a..b93c9e4 100644

                                  --- a/chapter1.doc

                                  +++ b/chapter1.doc

@@ -8,7 +8,8 @@ re going to cover Version Control Systems (VCS) and Git basics
  re going to cover how to get it and set it up for the first time if you don

                            t already have it on your system.

      In Chapter Two we will go over basic Git usage - how to use Git for the 80%

     -s going on, modify stuff and contribute changes. If the book spontaneously
```

Git коротко и ясно дал мне знать, что я добавил строку «Let’s see if this works», так оно и есть. Работает не идеально, так как добавляет немного лишнего в конце, но определённо работает. Если вы сможете найти или написать хорошо работающую программу для конвертации документов Word в обычный текст, то такое решение скорее всего будет невероятно эффективно. Тем не менее, strings доступен на большинстве Mac и Linux-систем, так что он может быть хорошим первым вариантом для того, чтобы сделать подобное со многими бинарными форматами.

Текстовые файлы в формате OpenDocument Тот же подход, который мы использовали для файлов MS Word (*.doc), может быть использован и для текстовых файлов в формате Open- Document, созданных в OpenOffice.org.

Добавим следующую строку в файл .gitattributes:

```
*.odt diff=odt
```

Теперь настроим фильтр odtв .git/config:

```
[diff "odt"]
binary = true

textconv = /usr/local/bin/odt-to-txt
```

Файлы в формате OpenDocument на самом деле являются запакованными zip'ом каталогами с множеством файлов (содержимое в XML-формате, таблицы стилей, изображения и т.д.). Мы напишем сценарий для извлечения содержимого и вывода его в виде обычного текста. Создайте файл /usr/local/bin/odt-to-txt (можете создать его в любом другом каталоге) со следующим содержимым:

```
#! /usr/bin/env perl
# Сценарий для конвертации OpenDocument Text (.odt) в обычный текст.
# Автор: Philipp Kempgen

if (! defined($ARGV[0])) {
print STDERR "Не задано имя файла!\n";
print STDERR "Использование: $0 имя файла\n"; exit 1;
}

my $content = '';
open my $fh, '-|', 'unzip', '-qq', '-p', $ARGV[0], 'content.xml' or die $!;
{
local $/ = undef;  # считываем файл целиком
$content = <$fh>;
}
close $fh;
$_ = $content;
s/<text:span\b[^>]*>//g;          # удаляем span'ы s/<text:h\b[^>]*>/\n\n***** /g;          # заголовки
s/<text:list-item\b[^>]*>\s*<text:p\b[^>]*>/\n  -- /g;          # элементы списков
s/<text:list\b[^>]*>/\n\n/g;          # списки
s/<text:p\b[^>]*>/\n  /g;          # параграфы
s/<[^>]+>//g;          # удаляем все XML-теги
s/\n{2,}/\n\n/g;          # удаляем подряд идущие пустые строки
s/\A\n+//;          # удаляем пустые строки в начале print "\n", $_, "\n\n";
```

Сделайте его исполняемым

```
chmod +x /usr/local/bin/odt-to-txt
```

Теперь git diffсможет сказать вам, что изменилось в .odtфайлах.

Изображения Ещё одна интересная проблема, которую можно решить таким способом, это сравнение файлов изображений. Один из способов сделать это — прогнать PNG-файлы через фильтр, извлекающий их EXIF-информацию — метаданные, которые дописываются в большинство форматов изображений. Если скачаете и установите программу exiftool, то сможете воспользоваться ею, чтобы извлечь из изображений текстовую информацию о метаданных, так чтобы diff хоть как-то показал вам текстовое представление произошедших изменений:

```
$ echo '*.png diff=exif' >> .gitattributes
$ git config diff.exif.textconv exiftool
```

Если вы замените в проекте изображение и запустите git diff, то получите что-то вроде такого:

```
diff --git a/image.png b/image.png
  index 88839c4..4afcb7c 100644
--- a/image.png
+++ b/image.png
@@ -1,12 +1,12 @@
  ExifTool Version Number         : 7.74
-File Size                       : 70 kB
-File Modification Date/Time     : 2009:04:21 07:02:45-07:00
+File Size                       : 94 kB
+File Modification Date/Time     : 2009:04:21 07:02:43-07:00
  File Type                       : PNG
  MIME Type                       : image/png
-Image Width                     : 1058
-Image Height                    : 889
```

Легко можно заметить, что размер файла, а также высота и ширина изображения поменялись.

Развёртывание ключа

Разработчики, привыкшие к SVN или CVS, часто хотят получить в Git возможность развёртывания ключа в стиле этих систем. Основная проблема с реализацией этой функциональности в Git это то, что нельзя записать в файл информацию о коммите после того,

как коммит был сделан, так как Git сначала считает контрольную сумму для файла. Не смотря на это, вы можете вставлять текст в файл во время его выгрузки и удалять его перед добавлением в коммит. Атрибуты Git предлагают два варианта сделать это.

Во-первых, вы можете внедрять SHA-1-сумму блоба в поле Idв файл автоматически. Если установить соответствующий атрибут для одного или нескольких файлов, то в следующий раз, когда вы будете выгружать данные из этой ветки, Git будет заменять это поле SHA-суммой блоба. Обратите внимание, что это SHA-1 не коммита, а самого блоба.

```
$ echo '*.txt ident' >> .gitattributes

$ echo '$Id$' > test.txt

$ git add test.txt
```

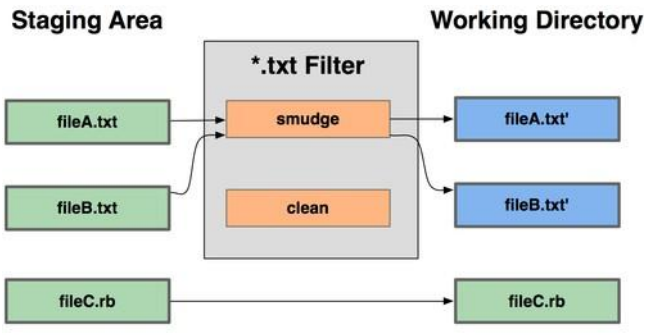
В следующий раз, когда вы будете выгружать этот файл, Git автоматически вставит в него SHA его блоба:

```
$ rm test.txt

$ git checkout -- test.txt

$ cat test.txt

$Id: 42812b7653c7b88933f8a9d6cad0ca16714b9bb3 $
```

Однако, такой результат мало применим. Если вы раньше пользовались развёртыванием ключа в CVS или Subversion, можете добавлять метку даты — SHA не особенно полезен, так как он довольно случаен, и к тому же, глядя на две SHA-суммы, никак не определить какая из них новее.

Как оказывается, можно написать свои собственные фильтры, которые будут делать подстановки в файлах при коммитах и выгрузке файлов. Для этого надо задать фильтры «clean» и «smudge».

В файле .gitattributes можно задать фильтр для определённых путей и затем установить сценарии, которые будут обрабатывать файлы непосредственно перед выгрузкой («smudge», смотри Рисунок 7-2) и прямо перед коммитом («clean», смотри Рисунок 7-3). Эти фильтры можно настроить на совершение абсолютно любых действий.



git checkout

Рисунок 7.2: Фильтр «smudge» выполняется при checkout.

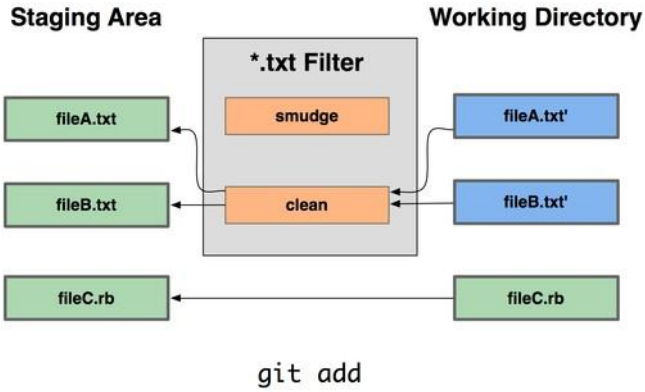


Рисунок 7.3: Фильтр «clean» выполняется при помещении файлов в индекс.

В сообщении первоначального коммита, добавляющего эту функциональность, дан простой пример того, как можно пропустить весь свой исходный код на C через программу indent перед коммитом. Сделать это можно, задав атрибут filter в файле .gitattributes так, чтобы он пропускал файлы *.cчерез фильтр «indent»:

```
*.c      filter=indent
```

Затем укажите Git'у, что должен делать фильтр «indent» при smudge и clean:

```
$ git config --global filter.indent.clean indent

$ git config --global filter.indent.smudge cat
```

В нашем случае, когда вы будете делать коммит, содержащий файлы, соответствующие шаблону *.c, Git прогонит их через программу indent перед коммитом, а потом через программу cat перед тем как выгрузить их на диск. Программа cat по сути является холостой — она выдаёт те же данные, которые получила. Фактически эта комбинация профильтровывает все файлы с исходным кодом на C через indentперед тем, как сделать коммит.

Ещё один интересный пример — это развёртывание ключа $Date$в стиле RCS. Чтобы сделать его правильно, нам понадобится небольшой сценарий, который принимает на вход имя файла, определяет дату последнего коммита в проекте и вставляет эту дату в наш файл. Вот небольшой сценарий на Ruby, который делает именно это:

```
#! /usr/bin/env ruby
  data = STDIN.read

                  last_date = `git log --pretty=format:"%ad" -1`

            puts data.gsub('$Date$', '$Date: ' + last_date.to_s + '$')
```

Всё, что делает этот сценарий, это получает дату последнего коммита с помощью команды git log, засовывает её во все строки $Date$, которые видит в stdin, и выводит результат — такое должно быть несложно реализовать на любом удобном вам языке. Давайте назовём этот файл expand_date и поместим в путь. Теперь в Git'е необходимо

настроить фильтр (назовём его dater) и указать, что надо использовать фильтр expand_dateпри выполнении smudge во время выгрузки файлов. Воспользуемся регулярным выражением Perl, чтобы убрать изменения при коммите:

```
$ git config filter.dater.smudge expand_date

$ git config filter.dater.clean 'perl -pe "s/\\\$Date[^\\\$]*\\\$/\\\$Date\\\$/"'
```

Этот фрагмент кода на Perl'е вырезает всё, что находит в строке $Date$ так, чтобы вернуть всё в начальное состояние. Теперь, когда наш фильтр готов, можете протестировать его, создав файл с ключом $Date$и установив для этого файла Git-атрибут, который задействует для него новый фильтр:

```
$ echo '# $Date$' > date_test.txt

$ echo 'date*.txt filter=dater' >> .gitattributes
```

Если мы сейчас добавим эти изменения в коммит и снова выгрузим файл, то мы увидим, что ключевое слово было заменено правильно:

```
$ git add date_test.txt .gitattributes

$ git commit -m "Testing date expansion in Git"

$ rm date_test.txt

$ git checkout date_test.txt

$ cat date_test.txt

# $Date: Tue Apr 21 07:26:52 2009 -0700$
```

Как видите, такая техника может быть весьма мощной для настройки проекта под свои нужды. Но вы должны быть осторожны, ибо файл .gitattributesвы добавите в коммит и будете его распространять вместе с проектом, а драйвер (в нашем случае dater) — нет. Так что не везде оно будет работать. Когда будете проектировать свои фильтры, постарайтесь сделать так, чтобы при возникновении в них ошибки проект не переставал работать правильно.

Экспорт репозитория

Ещё атрибуты в Git позволяют делать некоторые интересные вещи при экспортировании архива с проектом.

export-ignore Вы можете попросить Git не экспортировать определённые файлы и каталоги при создании архива. Если у вас есть подкаталог или файл, который вы не желаете включать в архив, но хотите, чтобы в проекте он был, можете установить для такого файла атрибут export-ignore.

Например, скажем, у вас в подкаталоге test/ имеются некоторые тестовые файлы, и нет никакого смысла добавлять их в тарбол при экспорте проекта. Тогда добавим следующую строку в файл с Git-атрибутами:

```
test/ export-ignore
```

Теперь, если вы запустите git archive, чтобы создать тарбол с проектом, этот каталог в архив включён не будет.

export-subst Ещё одна вещь, которую можно сделать с архивами, — это сделать какую-нибудь простую подстановку ключевых слов. Git позволяет добавить в любой файл строку вида $For- mat:$c любыми кодами форматирования, доступными в --pretty=format(многие из этих кодов мы рассматривали в Главе 2). Например, если вам захотелось добавить в проект файл с именем LAST_COMMIT, в который при запуске git archive будет автоматически помещаться дата последнего коммита, то такой файл вы можете сделать следующим образом:

```
$ echo 'Last commit date: $Format:%cd$' > LAST_COMMIT

$ echo "LAST_COMMIT export-subst" >> .gitattributes

$ git add LAST_COMMIT .gitattributes

$ git commit -am 'adding LAST_COMMIT file for archives'
```

После запуска git archive этот файл у вас в архиве будет иметь содержимое следующего вида:

```
$ cat LAST_COMMIT

Last commit date: $Format:Tue Apr 21 08:38:48 2009 -0700$
```

Стратегии слияния

Атрибуты Git могут также быть использованы для того, чтобы попросить Git использовать другие стратегии слияния для определённых файлов в проекте. Одна очень полезная возможность — это сказать Git'у, чтобы он не пытался слить некоторые файлы, если для них есть конфликт, а просто выбрал ваш вариант, предпочтя его чужому.

Это полезно в том случае, если ветка в вашем проекте разошлась с исходной, но вам всё же хотелось бы иметь возможность слить изменения из неё обратно, проигнорировав некоторые файлы. Скажем, у вас есть файл с настройками базы данных, который называется database.xml, и в двух ветках он разный, и вы хотите влить другую свою ветку, не трогая файл с настройками базы данных. Задайте атрибут следующим образом:

```
database.xml merge=ours
```

При вливании другой ветки, вместо конфликтов слияния для файла database.xml, вы увидите следующее:

```
$ git merge topic

Auto-merging database.xml
Merge made by recursive.
```

В данном случае database.xml остался в том варианте, в каком и был изначально.

Перехватчики в Git

Как и во многих других системах управления версиями, в Git есть возможность запускать собственные сценарии в те моменты, когда происходят некоторые важные действия. Существуют две группы подобных перехватчиков (hook): на стороне клиента и на стороне сервера. Перехватчики на стороне клиента предназначены для клиентских операций, таких как создание коммита и слияние. Перехватчики на стороне сервера нужны для серверных операций, таких как приём отправленных коммитов. Перехватчики могут быть использованы для выполнения самых различных задач. О некоторых из таких задач мы и поговорим.

Установка перехватчика

Все перехватчики хранятся в подкаталоге hooks в Git-каталоге. В большинстве проектов это .git/hooks. По умолчанию Git заполняет этот каталог кучей примеров сценариев, многие из которых полезны сами по себе, но кроме того в них задокументированы входные значения для каждого из сценариев. Все эти примеры являются сценариями для командной оболочки с вкраплениями Perl'а, но вообще-то будет работать любой исполняемый сценарий с правильным именем — вы можете писать их на Ruby или Python или на чём-то ещё, что вам нравится. В версиях Git'а старше 1.6 эти файлы с примерами перехватчиков оканчиваются на .sample; вам надо их переименовать. Для версий Git'а меньше чем 1.6 файлы с примерами имеют правильные имена, но не имеют прав на исполнение.

Чтобы активировать сценарий-перехватчик, положите файл в подкаталог hooks в Git-каталоге, дайте ему правильное имя и права на исполнение. С этого момента он будет вызываться. Основные имена перехватчиков мы сейчас рассмотрим.

Перехватчики на стороне клиента

Существует множество перехватчиков, работающих на стороне клиента. В этом разделе они поделены на перехватчики используемые при работе над коммитами, сценарии используемые в процессе работы с электронными письмами и все остальные, работающие на стороне клиента.

Перехватчики для работы с коммитами Первые четыре перехватчика относятся к процессу создания коммита. Перехватчик pre-commit запускается первым, ещё до того, как вы

наберёте сообщение коммита. Его используют для проверки снимка состояния перед тем, как сделать коммит, чтобы проверить не забыли ли вы что-нибудь, чтобы убедиться, что вы запустили тесты, или проверить в коде ещё что-нибудь, что вам нужно. Завершение перехватчика с ненулевым кодом прерывает создание коммита, хотя вы можете обойти это с помощью git commit --no-verify. Можно, например, проверить стиль кодирования (запускать lint или что-нибудь аналогичное), проверить наличие пробельных символов в конце строк (перехватчик по умолчанию занимается именно этим) или проверить наличие необходимой документации для новых методов.

Перехватчик prepare-commit-msg запускается до появления редактора с сообщением коммита, но после создания сообщения по умолчанию. Он позволяет отредактировать сообщение по умолчанию перед тем, как автор коммита его увидит. У этого перехватчика есть несколько опций: путь к файлу, в котором сейчас хранится сообщение коммита, тип коммита и SHA-1 коммита (если в коммит вносится правка с помощью git commit --amend). Как правило данный перехватчик не представляет пользы для обычных коммитов; он скорее хорош для коммитов с автогенерируемыми сообщениями, такими как шаблонные сообщения коммитов, коммиты-слияния, уплотнённые коммиты (squashed commits) и коммиты с исправлениями (amended commits). Данный перехватчик можно использовать в связке с шаблоном для коммита, чтобы программно добавлять в него информацию.

Перехватчик commit-msg принимает один параметр, и снова это путь к временному файлу, содержащему текущее сообщение коммита. Когда сценарий завершается с ненулевым кодом, Git прерывает процесс создания коммита. Так что можно использовать его для проверки состояния проекта или сообщений коммита перед тем, как его одобрить. В последнем разделе главы я продемонстрирую, как использовать данный перехватчик, чтобы проверить, что сообщение коммита соответствует требуемому шаблону.

После того, как весь процесс создания коммита завершён, запускается перехватчик post-commit. Он не принимает никаких параметров, но вы с лёгкостью можете получить последний коммит, выполнив git log -1 HEAD. Как правило, этот сценарий используется для уведомлений или чего-то в этом роде.

Сценарии на стороне клиента, предназначенные для запуска во время работы над коммитами, могут быть использованы при осуществлении практически любого типа рабочего процесса.

Их часто используют, чтобы обеспечить соблюдение определённых стандартов, хотя важно отметить, что данные сценарии не передаются при клонировании. Вы можете принудить к соблюдению правил на стороне сервера, отвергая присланные коммиты, если они не подчиняются некоторым правилам, но использование данных сценариев на клиентской

стороне полностью зависит только от разработчика. Итак, эти сценарии призваны помочь разработчикам, и это обязанность разработчиков установить и сопровождать их, хотя разработчики и имеют возможность в любой момент подменить их или модифицировать.

Перехватчики для работы с e-mail Для рабочих процессов, основанных на электронной почте, есть три специальных клиентских перехватчика. Все они вызываются командой git am, так что, если вы не пользуетесь этой командой в процессе своей работы, то можете смело переходить к следующему разделу. Если вы принимаете патчи, отправленные по e-mail и подготовленные с помощью git format-patch, то некоторые из них могут оказать для вас полезными.

Первый запускаемый перехватчик — это applypatch-msg. Он принимает один аргумент — имя временного файла, содержащего предлагаемое сообщение коммита. Git прерывает наложение патча, если сценарий завершается с ненулевым кодом. Это может быть использовано для того, чтобы убедиться, что сообщение коммита правильно отформатировано или, чтобы нормализовать сообщение, отредактировав его на месте из сценария.

Следующий перехватчик, запускаемый во время наложения патчей с помощью git am— это pre-applypatch. У него нет аргументов, и он запускается после того, как патч наложен, поэтому его можно использовать для проверки снимка состояния перед созданием коммита. Можно запустить тесты или как-то ещё проверить рабочее дерево с помощью этого сценария. Если чего-то не хватает, или тесты не пройдены, выход с ненулевым кодом так же завершает сценарий git amбез применения патча.

Последний перехватчик, запускаемый во время работы git am— это post-applypatch. Его можно использовать для уведомления группы или автора патча о том, что вы его применили. Этим сценарием процесс наложения патча остановить уже нельзя.

Другие клиентские перехватчики Перехватчик pre-rebase запускается перед перемещением чего-либо, и может остановить процесс перемещения, если завершится с ненулевым кодом.

Этот перехватчик можно использовать, чтобы запретить перемещение любых уже отправленных коммитов. Пример перехватчика pre-rebase, устанавливаемый Git’ом, это и делает, хотя он предполагает, что ветка, в которой вы публикуете свои изменения, называется next. Вам скорее всего нужно будет заменить это имя на имя своей публичной стабильной ветки.

После успешного выполнения команды git checkout, запускается перехватчик post-checkout. Его можно использовать для того, чтобы правильно настроить рабочий каталог для своей проектной среды. Под этим может подразумеваться, например, перемещение в каталог

больших бинарных файлов, которые вам не хочется включать под версионный контроль, автоматическое генерирование документации или что-то ещё в таком же духе.

И наконец, перехватчик post-merge запускается после успешного выполнения команды merge. Его можно использовать для восстановления в рабочем дереве данных, которые Git не может отследить, таких как информация о правах. Этот перехватчик может также проверить наличие внешних по отношению к контролируемым Git'ом файлов, которые вам нужно скопировать в каталог при изменениях рабочего дерева.

Перехватчики на стороне сервера

В дополнение к перехватчикам на стороне клиента вы, как системный администратор, можете задействовать пару важных перехватчиков на стороне сервера, чтобы навязать в своём проекте правила практически любого вида. Эти сценарии выполняются до и после отправки данных на сервер. Pre-перехватчики могут быть в любое время завершены с ненулевым кодом, чтобы отклонить присланные данные, а также вывести клиенту обратно сообщение об ошибке. Вы можете установить настолько сложные правила приёма данных, насколько захотите.

pre-receive и post-receive Первым сценарий, который выполняется при обработке отправленных клиентом данных, — это pre-receive. Он принимает на вход из stdin список отправленных ссылок; если он завершается с ненулевым кодом, ни одна из них не будет принята. Этот перехватчик можно использовать, чтобы, например, убедиться, что ни одна из обновлённых ссылок не выполняет ничего кроме перемотки, или, чтобы убедиться, что пользователь, запустивший gitpush, имеет права на создание, удаление или изменение для всех файлов модифицируемых этим push'ем.

Перехватчик post-receive запускается после того, как весь процесс завершился, и может быть использован для обновления других сервисов или уведомления пользователей. Он получает на вход из stdin те же данные, что и перехватчик pre-receive. Примерами использования могут быть: отправка писем в рассылку, уведомление сервера непрерывной интеграции или обновление карточки (ticket) в системе отслеживания ошибок — вы можете даже анализировать сообщения коммитов, чтобы выяснить, нужно ли открыть, изменить или закрыть какие-то карточки. Этот сценарий не сможет остановить процесс приёма данных, но клиент не будет отключён до тех пор, пока процесс не завершится; так что будьте осторожны, если хотите сделать что-то, что может занять много времени.

update Сценарий update очень похож на сценарий pre-receive, за исключением того, что он выполняется для каждой ветки, которую отправитель данных пытается обновить. Если отправитель пытается обновить несколько веток, то pre-receiveвыполнится только один раз, в

то время как update выполнится по разу для каждой обновляемой ветки. Сценарий не считывает параметры из stdin, а принимает на вход три аргумента: имя ссылки (ветки), SHA-1, на которую ссылка указывала до запуска push, и тот SHA-1, который пользователь пытается отправить. Если сценарий update завершится с ненулевым кодом, то только одна ссылка будет отклонена, остальные ссылки всё ещё смогут быть обновлены.

Пример навязывания политики с помощью Git

В этом разделе мы используем ранее полученные знания для организации в Git такого рабочего процесса, который проверяет сообщения коммитов на соответствие заданному формату, из обновлений разрешает только перемотки и позволяет только определённым пользователям изменять определённые подкаталоги внутри проекта. Мы создадим клиентские сценарии, которые помогут разработчикам узнать, будет ли их push отклонён, и серверные сценарии, которые будут действительно вынуждать следовать установленным правилам.

Для их написания я использовал Ruby, и потому что это мой любимый язык сценариев, и потому что из всех языков сценариев он больше всего похож на псевдокод; таким образом, код должен быть вам понятен в общих чертах, даже если вы не пользуетесь Ruby. Однако любой язык сгодится. Все примеры перехватчиков, распространяемые вместе с Git'ом, написаны либо на Perl, либо на Bash, так что вы сможете просмотреть достаточно примеров перехватчиков на этих языках, заглянув в примеры.

Перехватчик на стороне сервера

Вся работа для сервера будет осуществляться в файле update из каталога hooks. Файл update запускается по разу для каждой отправленной ветки и принимает на вход ссылку, в которую сделано отправление, старую версию, на которой ветка находилась раньше, и новую присланную версию. Кроме того, вам будет доступно имя пользователя, приславшего данные, если push был выполнен по SSH. Если вы позволили подключаться всем под одним пользователем (например, «git») с аутентификацией по открытому ключу, то вам может понадобиться создать для этого пользователя обёртку командной оболочки, которая на основе открытого ключа будет определять, какой пользователь осуществил подключение, и записывать этого пользователя в какой-нибудь переменной окружения. Тут я буду предполагать, что имя подключившегося пользователя находится в переменной окружения $USER, так что начнём наш сценарий со сбора всей необходимой информации:

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

$refname = ARGV[0]

$oldrev  = ARGV[1]

$newrev  = ARGV[2]

$user    = ENV['USER']

puts "Enforcing Policies... \n(#{$refname}) (#{$oldrev[0,6]}) (#{$newrev[0,6]})"
```

Да, я использую глобальные переменные. Не судите строго — в таком виде получается нагляднее.

Установка особого формата сообщений коммитов Первая наша задача — это заставить все сообщения коммитов обязательно придерживаться определённого формата. Просто чтобы было чем заняться, предположим, что каждое сообщение должно содержать строку вида «ref: 1234», так как мы хотим, чтобы каждый коммит был связан с некоторым элементом в нашей системе с карточками. Нам необходимо просмотреть все присланные коммиты, выяснить, есть ли такая строка в сообщении коммита, и, если строка отсутствует в каком-либо из этих коммитов, то завершить сценарий с ненулевым кодом, чтобы push был отклонён.

Список значений SHA-1 для всех присланных коммитов можно получить, взяв значения $newrevи $oldrevи передав их служебной команде gitrev-list. По сути, это команда git log, но по умолчанию она выводит только SHA-1 значения и больше ничего. Таким образом, чтобы получить список SHA для всех коммитов, сделанных между одним SHA коммита и другим, достаточно выполнить следующее:

```
$ git rev-list 538c33..d14fc7
d14fc7c847ab946ec39590d87783c69b031bdfb7
9f585da4401b0a3999e84113824d15245c13f0be

234071a1be950e2a8d078e6141f5cd20c1e61ad3
dfa04c9ef3d5197182f13fb5b9b1fb7717d2222a
17716ec0f1ff5c77eff40b7fe912f9f6cfd0e475
```

Можно взять этот вывод, пройти в цикле по SHA-хешам всех этих коммитов, беря их сообщения и проверяя с помощью регулярного выражения, совпадает ли сообщение с шаблоном.

Нам нужно выяснить, как из всех этих коммитов получить их сообщения, для того, чтобы их протестировать. Чтобы получить данные коммита в сыром виде, можно воспользоваться ещё одной служебной командой, которая называется git cat-file. Мы рассмотрим все эти служебные команды более подробно в Главе 9, но пока что, вот, что эта команда нам выдала:

```
                              $ git cat-file commit ca82a6

 tree cfda3bf379e4f8dba8717dee55aab78aef7f4daf
parent 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7

               author Scott Chacon <schacon@gmail.com> 1205815931 -0700

committer Scott Chacon <schacon@gmail.com> 1240030591 -0700

                 changed the version number
```

Простой способ получить сообщение коммита для коммита, чьё значение SHA-1 известно, это дойти в выводе команды git cat-filedo первой пустой строки и взять всё, что идёт после неё. В Unix-системах это можно сделать с помощью команды sed:

```
$ git cat-file commit ca82a6 | sed '1,/^$/d'
         changed the version number
```

Используйте приведённую ниже абракадабру, чтобы получить для каждого отправленного коммита его сообщение и выйти, если обнаружится, что что-то не соответствует требованиям. Если хотим отклонить отправленные данные, выходим с ненулевым кодом. Весь метод целиком выглядит следующим образом:

```
                              $regex = /\[ref: (\d+)\]/
```

```
# принуждает использовать особый формат сообщений
             def check_message_format

  missed_revs = `git rev-list #{$oldrev}..#{$newrev}`.split("\n")
                    missed_revs.each do |rev|

    message = `git cat-file commit #{rev} | sed '1,/^$/d'`
                  if !$regex.match(message)

      puts "[POLICY] Your message is not formatted correctly"
                               exit 1

    end
   end
```

Добавив это в свой сценарий update, мы запретим обновления, содержащие коммиты, сообщения которых не соблюдают наше правило.

Настройка системы контроля доступа для пользователей Предположим, что нам хотелось бы добавить какой-нибудь механизм для использования списков контроля доступа (ACL), где указано, какие пользователи могут отправлять изменения и в какие части проекта. Несколько людей будут иметь полный доступ, а остальные будут иметь доступ на изменение только некоторых подкаталогов или отдельных файлов. Чтобы обеспечить выполнение такой политики, мы запишем правила в файл acl, который будет находиться в нашем «голом» репозитории на сервере. Нам нужно будет, чтобы перехватчик update брал эти правила, смотрел на то, какие файлы были изменены присланными коммитами, и определял, имеет ли пользователь, выполнивший push, право на обновление всех этих файлов.

Первое, что мы сделаем, — это напишем свой ACL. Мы сейчас будем использовать формат очень похожий на механизм ACL в CVS. В нём используется последовательность строк, где первое поле — это availили unavail, следующее поле — это разделённый запятыми список пользователей, для которых применяется правило, и последнее поле — это путь, к которому применяется правило (пропуск здесь означает открытый доступ). Все эти поля разделяются вертикальной чертой (|).

В нашем примере будет несколько администраторов, сколько-то занимающихся написанием документации с доступом к каталогу doc и один разработчик, который имеет доступ только к каталогам libи tests, и наш файл aclбудет выглядеть так:

```
   avail|nickh,pjhyett,defunkt,tpw
avail|usinclair,cdickens,ebronte|doc
        avail|schacon|lib

                                avail|schacon|tests
```

Начнём со считывания этих данных в какую-нибудь пригодную для использования структуру.

В нашем случае, чтобы не усложнять пример, мы будем применять только директивы avail. Вот метод, который даёт нам ассоциативный массив, где ключом является имя пользователя, а значением — массив путей, для которых пользователь имеет доступ на запись:

```
                        def get_acl_access_data(acl_file)

                              # считывание данных ACL

 acl_file = File.read(acl_file).split("\n").reject { |line| line == '' }
                              access = {}

                              acl_file.each do |line|

   avail, users, path = line.split('|')
       next unless avail == 'avail'
     users.split(',').each do |user|

     access[user] ||= []
    access[user] << path

   end
  end
```

Для рассмотренного ранее ACL-файла, метод get_acl_access_data вернёт структуру данных следующего вида:

```
                                {"defunkt"=>[nil],

                                   "tpw"=>[nil],

       "nickh"=>[nil],
     "pjhyett"=>[nil],
"schacon"=>["lib", "tests"],
    "cdickens"=>["doc"],

                               "usinclair"=>["doc"],

                                "ebronte"=>["doc"]}
```

Теперь, когда мы разобрались с правами, нам нужно выяснить, какие пути изменяются присланными коммитами, чтобы можно было убедиться, что пользователь, выполнивший push, имеет ко всем ним доступ.

Мы довольно легко можем определить, какие файлы были изменены в одном коммите, с помощью опции --name-onlудля команды git log(мы упоминали о ней в Главе 2):

```
$ git log -1 --name-only --pretty=format:'' 9f585d

                    README

                                   lib/test.rb
```

Если мы воспользуемся ACL-структурой, полученной из метода get_acl_access_data, и сверим её со списком файлов для каждого коммита, то мы сможем определить, имеет ли пользователь право на отправку своих коммитов:

```
# некоторые подкаталоги в проекте разрешено модифицировать только определённым пользователям
                                def check_directory_perms

                            access = get_acl_access_data('acl')
```

```
# проверим, что никто не пытается прислать чего-то, что ему нельзя
  new_commits = `git rev-list #{$oldrev}..#{$newrev}`.split("\n")
                      new_commits.each do |rev|

   files_modified = `git log -1 --name-only --pretty=format:'' #{rev}`.split("\n")
                             files_modified.each do |path|

           next if path.size == 0
          has_file_access = false
     access[$user].each do |access_path|

                    if !access_path  # пользователь имеет полный доступ

         || (path.index(access_path) == 0) # доступ к этому пути
                         has_file_access = true

       end
      end

                                    if !has_file_access

       puts "[POLICY] You do not have access to push to #{path}"
                                  exit 1

      end
     end
```

Большую часть этого кода должно быть не сложно понять. Мы получаем список присланных на сервер коммитов с помощью git rev-list. Затем для каждого из них мы узнаём, какие файлы были изменены, и убеждаемся, что пользователь, сделавший push, имеет доступ ко всем изменённым путям. Один Ruby'изм, который может быть непонятен это path.index(access_path) == 0. Это условие верно, если path начинается с access_path — оно гарантирует, что access_path это не просто один из разрешённых путей, а что каждый путь, к которому запрашивается доступ, начинается с одного из разрешённых путей.

Теперь наши пользователи не смогут отправить никаких коммитов с плохо отформатированными сообщениями и не смогут изменить файлы вне предназначенных для них путей.

Разрешение только обновлений-перемоток Единственное, что нам осталось — это оставить доступными только обновления-перемотки. В Git версии 1.6 и более новых можно просто задать настройки receive.denyDeletes и receive.denyNonFastForwards. Но осуществление этого с помощью перехватчика будет работать и в старых версиях Git, и к тому же вы сможете изменить его так, чтобы запрет действовал только для определённых пользователей, или ещё как-то, как вам захочется.

Логика здесь такая — мы проверяем, есть ли такие коммиты, которые достижимы из старой версии и не достижимы из новой. Если таких нет, то сделанный push был перемоткой; в противном случае мы его запрещаем:

```
# разрешаем только обновления-перемотки
        def check_fast_forward

  missed_refs = `git rev-list #{$newrev}..#{$oldrev}`
    missed_ref_count = missed_refs.split("\n").size

                                if missed_ref_count > 0

               puts "[POLICY] Cannot push a non fast-forward reference"
    exit 1
    end

       end

check_fast_forward
```

Всё готово. Если вы выполните chmod u+x .git/hooks/update(а это тот файл, в который вы должны были поместить весь наш код) и затем попытаетесь отправить ссылку, для которой нельзя выполнить перемотку, то вы получите что-то типа такого:

```
$ git push -f origin master
Counting objects: 5, done.

   Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 323 bytes, done.
    Total 3 (delta 1), reused 0 (delta 0)
    Unpacking objects: 100% (3/3), done.

        Enforcing Policies...
(refs/heads/master) (8338c5) (c5b616)

[POLICY] Cannot push a non-fast-forward reference
  error: hooks/update exited with error code 1
error: hook declined to update refs/heads/master
          To git@gitserver:project.git

                ! [remote rejected] master -> master (hook declined)
```

Тут есть пара интересных моментов. Во-первых, когда перехватчик начинает свою работу, мы видим это:

```
        Enforcing Policies...
(refs/heads/master) (fb8c72) (c56860)
```

Обратите внимание, что мы выводили это в stdout в самом начале нашего сценария update. Важно отметить, что всё, что сценарий выводит в stdout, будет передано клиенту.

Следующая вещь, которую мы видим, это сообщение об ошибке:

```
[POLICY] Cannot push a non fast-forward reference
  error: hooks/update exited with error code 1
error: hook declined to update refs/heads/master
```

Первую строку напечатали мы, а в остальных двух Git сообщает, что сценарий update завершился с ненулевым кодом, и это именно то, что отклонило ваш push. И наконец мы видим это:

```
To git@gitserver:project.git

! [remote rejected] master -> master (hook declined)

error: failed to push some refs to 'git@gitserver:project.git'
```

Сообщение «remote rejected» будет появляться для каждой отклонённой перехватчиком ссылки. Оно сообщает нам, что ссылка была отклонена именно из-за сбоя в перехватчике.

Кроме того, при отсутствии отметки «ref» в каком-либо из коммитов, вы увидите сообщение об ошибке, которое мы для этого напечатали.

```
[POLICY] Your message is not formatted correctly
```

Или если кто-то попытается отредактировать файл, не имея к нему доступа, то отправив коммит с этими изменениями, он получит похожее сообщение. Например, если человек, пишущий документацию, попытается отправить коммит, вносящий изменения в файлы каталога lib, то увидит:

```
[POLICY] You do not have access to push to lib/test.rb
```

Вот и всё. С этого момента, до тех пор пока сценарий updateнаходится на своём месте и имеет права на исполнение, репозиторий никогда не будет откатан назад, в нём никогда не будет коммитов с сообщениями без вашего паттерна, и пользователи будут ограничены в доступе к файлам.

Перехватчики на стороне клиента

Обратная сторона такого подхода — это многочисленные жалобы, которые неизбежно появятся, когда отправленные пользователями коммиты будут отклонены. Когда чью-то тщательно оформленную работу отклоняют в последний момент, этот человек может быть сильно расстроен и смущён. Мало того, ему придётся отредактировать свою историю, чтобы откорректировать её, а это обычно не для слабонервных.

Решение данной проблемы — предоставить пользователям какие-нибудь перехватчики, которые будут работать на стороне пользователя и будут сообщать ему, если он делает что-то, что скорее всего будет отклонено. При таком подходе, пользователи смогут исправить любые проблемы до создания коммита и до того, как эти проблемы станет сложно исправить. Так как перехватчики не пересылаются при клонировании проекта, вам придётся распространять эти сценарии каким-то другим способом и потом сделать так, чтобы ваши пользователи скопировали их в свой каталог .git/hooks и сделали их исполняемыми. Эти перехватчики можно поместить

в свой проект или даже в отдельный проект, но способа установить их автоматически не существует.

Для начала, перед записью каждого коммита нам надо проверить его сообщение, чтобы быть уверенным, что сервер не отклонит изменения из-за плохо отформатированного сообщения коммита. Чтобы сделать это, добавим перехватчик commit-msg. Если мы сможем прочитать сообщение из файла, переданного в качестве первого аргумента, и сравнить его с шаблоном, то можно заставить Git прервать создание коммита при обнаружении несовпадения:

```
  #!/usr/bin/env ruby
message_file = ARGV[0]

                          message = File.read(message_file)


$regex = /\[ref: (\d+)\]/

if !$regex.match(message)

  puts "[POLICY] Your message is not formatted correctly"
                           exit 1

                                        end
```

Если этот сценарий находится на своём месте (в .git/hooks/commit-msg) и имеет права на исполнение, то при создании коммита с неправильно оформленным сообщением, вы увидите это:

```
                                $ git commit -am 'test'

                    [POLICY] Your message is not formatted correctly
```

В этом случае коммит не был завершён. Однако, когда сообщение содержит правильный шаблон, Git позволяет создать коммит:

```
$ git commit -am 'test [ref: 132]'
 [master e05c914] test [ref: 132]

                    1 files changed, 1 insertions(+), 0 deletions(-)
```

Далее мы хотим убедиться, что пользователь не модифицирует файлы вне своей области, заданной в ACL. Если в проекте в каталоге .gitуже есть копия файла acl, который мы использовали ранее, то сценарий pre-commitследующего вида применит эти ограничения:

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

                                   $user    = ENV['USER']

                          # [ insert acl_access_data method from above ]

# некоторые подкаталоги в проекте разрешено модифицировать только определённым пользователям
                                    def check_directory_perms

                             access = get_acl_access_data('.git/acl')

  files_modified = `git diff-index --cached --name-only HEAD`.split("\n")
                           files_modified.each do |path|

          next if path.size == 0
         has_file_access = false
    access[$user].each do |access_path|

    if !access_path || (path.index(access_path) == 0)
                  has_file_access = true
```

```
                                              end

                                     if !has_file_access

      puts "[POLICY] You do not have access to push to #{path}"
                                    exit 1

    end
  end

        end
```

Это примерно тот же сценарий, что и на стороне сервера, но с двумя важными отличиями. Первое — файл acl находится в другом месте, так как этот сценарий теперь запускается из рабочего каталога, а не из Git-каталога. Нужно изменить путь к ACL-файлу с этого:

```
                               access = get_acl_access_data('acl')
```

на этот:

```
                            access = get_acl_access_data('.git/acl')
```

Другое важное отличие — это способ получения списка изменённых файлов. Так как метод, действующий на стороне сервера, смотрит в лог коммитов, а сейчас коммит ещё не был записан, нам надо получить список файлов из индекса. Вместо

```
           files_modified = `git log -1 --name-only --pretty=format:'' #{ref}`
```

мы должны использовать

```
                files_modified = `git diff-index --cached --name-only HEAD`
```

Но это единственные два отличия — во всём остальном этот сценарий работает точно так же. Но надо предупредить, что он предполагает, что локально вы работаете под

тем же пользователем, от имени которого отправляете изменения на удалённый сервер. Если это не так, то вам необходимо задать переменную $userвручную.

Последнее, что нам нужно сделать, — это проверить, что пользователь не пытается отправить ссылки не с перемоткой, но это случается не так часто. Чтобы получились ссылки, не являющиеся перемоткой, надо либо переместить ветку за уже отправленный коммит, либо попытаться отправить другую локальную ветку в ту же самую удалённую ветку.

Так как сервер в любом случае сообщит вам о том, что нельзя отправлять обновления, не являющиеся перемоткой, а перехватчик запрещает принудительные push'и, единственная оплошность, которую вы можете попробовать предотвратить, это перемещение коммитов, которые уже были отправлены на сервер.

Вот пример сценария pre-rebase, который это проверяет. Он принимает на вход список всех коммитов, которые вы собираетесь переписать, и проверяет, нет ли их в какой-нибудь из ваших удалённых веток. Если найдётся такой коммит, который достижим из одной из удалённых веток, сценарий прервёт выполнение перемещения:

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

base_branch = ARGV[0]
      if ARGV[1]

  topic_branch = ARGV[1]
         else

  topic_branch = "HEAD"
         end

target_shas = `git rev-list #{base_branch}..#{topic_branch}`.split("\n")
      remote_refs = `git branch -r`.split("\n").map { |r| r.strip }

     target_shas.each do |sha|
  remote_refs.each do |remote_ref|

    shas_pushed = `git rev-list ^#{sha}^@ refs/remotes/#{remote_ref}`
                if shas_pushed.split("\n").include?(sha)

      puts "[POLICY] Commit #{sha} has already been pushed to #{remote_ref}"
                                      exit 1

    end
   end
```

Этот сценарий использует синтаксис, который мы не рассматривали в разделе «Выбор ревизии» в Главе 6. Мы получили список коммитов, которые уже были отправлены на сервер, выполнив это:

```
git rev-list ^#{sha}^@ refs/remotes/#{remote_ref}
```

Запись SHAˆ@означает всех родителей указанного коммита. Мы ищем какой-нибудь коммит, который достижим из последнего коммита в удалённой ветке и не достижим ни из

одного из родителей какого-либо SHA, который вы пытаетесь отправить на сервер — это значит, что это перемотка.

Главный недостаток такого подхода — это то, что проверка может быть очень медленной и зачастую избыточной — если вы не пытаетесь отправить данные принудительно с помощью -f, сервер и так выдаст предупреждение и не примет данные. Однако, это интересное упражнение и теоретически может помочь вам избежать перемещения, к которому потом придётся вернуться, чтобы исправить.

Итоги

Мы рассмотрели большинство основных способов настройки клиента и сервера Git'a с тем, чтобы он был максимально удобен для ваших проектов и при вашей организации рабочего процесса. Мы узнали о всевозможных настройках, атрибутах файлов и о перехватчиках событий, а также рассмотрели пример настройки сервера с соблюдением политики. Теперь вам должно быть по плечу заставить Git подстроиться под практически любой тип рабочего процесса, который можно вообразить