Волшебство Git

Ben Lynn

Август 2007

От редактора перевода

Не буду долго вас задерживать перед интересным чтением, лишь дам небольшие пояснения по переводу терминологии.

Приводя текст к единому стилю, я старался в первую очередь сохранить его цельность и легкость восприятия, а уже затем следовать чистоте языка. Поэтому на русский переведены лишь устоявшиеся термины; в тех случаях, когда общепринятого русского слова нет, была оставлена калька с английского. Например, используется слово «каталог» вместо «директория»; «хранилище» вместо «репозиторий» или «репозитарий»; «слияние» вместо «мерж»; и «ветка» вместо «бранч». Обратные примеры: «коммит», а не «фиксация»; «хук», а не «крюк»; «патч», а не «заплатка». Единственное исключение сделано для фразы «буферная зона» вместо «область стейджинг» и, соответственно, слова «буфер» вместо «стейдж»: поскольку здесь уже не только перевод, но и калька не есть общеупотребительные термины, то лучше было попытаться объяснить смысл понятия.

Надеюсь, эти краткие пояснения не оставят для вас неровностей в переводе и позволят погрузиться в текст книги без помех. Приятного чтения.

Предисловие

Git это швейцарский нож управления версиями — надежный универсальный многоцелевой инструмент, чья черезвычайная гибкость делает его сложным в изучении даже для многих профессионалов.

Как говорил Артур Кларк, любая достаточно развитая технология неотличима от волшебства. Это отличный подход к Git: новички могут игнорировать принципы его внутренней работы и рассматривать Git как нечто восхищающее друзей и приводящее в бешенство врагов своими чудесными способностями.

Вместо того, чтобы вдаваться в подробности, мы предоставим приблизительные инструкции для получения конкретных результатов. При частом использовании

вы постепенно поймете, как работает каждый трюк и как приспосабливать рецепты под ваши нужды.

- Китайский (упрощенный): JunJie, Meng и JiangWei.
- Испанский: Rodrigo Toledo.
- Немецкий: Benjamin Bellee и Armin Stebich. Armin также разместил немецкий перевод на его сайте.
- Русский: Тихон Тарнавский, Михаил Дымсков и другие.
- Украинский: Владимир Боденчук.
- Французский: Alexandre Garel. Также размещён на itaapy.
- Португальский: Leonardo Siqueira Rodrigues [в формате ODT].
- HTML одной страницей: чистый HTML без CSS.
- PDF файл: для печати.
- Пакет Debian, пакет Ubuntu: получите локальную копию этого сайта. Придется кстати, если этот сервер будет недоступен.

Благодарности

Я очень ценю, что столь многие люди работали над переводами этих строк. Я благодарен названным выше людям за их усилия, расширившие мою аудиторию.

Dustin Sallings, Alberto Bertogli, James Cameron, Douglas Livingstone, Michael Budde, Richard Albury, Tarmigan, Derek Mahar, Frode Aannevik, Keith Rarick, Andy Somerville, Ralf Recker, Øyvind A. Holm, Miklos Vajna, Sébastien Hinderer, Thomas Miedema, Joe Malin и Tyler Breisacher содействовали в правках и доработках.

François Marier сопровождает пакет Debian, изначально созданный Daniel Baumann.

Мои благодарности остальным за вашу поддержку и похвалы. Мне очень хотелось процитировать вас здесь, но это могло бы возвысить ваше тщеславие до невообразимых высот.

Если я случайно забыл упомянуть вас, пожалуйста, напомните мне или просто вышлите патч.

- http://repo.or.cz/ хостинг свободных проектов. Первый сайт Git-хостинга. Основан и поддерживается одним из первых разработчиков Git.
- http://gitorious.org/ другой сайт Git-хостинга, нацеленный на проекты с открытым кодом.

- http://github.com/ хостинг для проектов с открытым кодом; а также для закрытых проектов (на платной основе).

Большое спасибо каждому из этих сайтов за размещение этого руководства.

Лицензия

Введение

Чтобы объяснить, что такое управление версиями, я буду использовать аналогии. Если нужно более точное объяснение, обратитесь к статье википедии.

Работа - это игра

Я играл в компьютерные игры почти всю свою жизнь. А вот использовать системы управления версиями начал уже будучи взрослым. Полагаю, я такой не один, и сравнение этих двух занятий может помочь объяснению и пониманию концепции.

Представьте, что редактирование кода или документа — игра. Далеко продвинувшись, вы захотите сохраниться. Для этого вы нажмете на кнопку «Сохранить» в вашем любимом редакторе.

Но это перезапишет старую версию. Это как в древних играх, где был только один слот для сохранения: конечно, вы можете сохраниться, но вы больше никогда не сможете вернуться к более раннему состоянию. Это досадно, так как прежнее сохранение могло указывать на одно из очень интересных мест в игре, и может быть, однажды вы захотите вернуться к нему. Или, что еще хуже, вы сейчас находитесь в безвыигрышном положении и вынуждены начинать заново.

Управление версиями

Во время редактирования вы можете «Сохранить как…» в другой файл, или скопировать файл куда-нибудь перед сохранением, чтобы уберечь более старые версии. Может быть, заархивировав их для экономии места на диске. Это самый примитивный вид управления версиями, к тому же требующий интенсивной ручной работы. Компьютерные игры прошли этот этап давным-давно, в большинстве из них есть множество слотов для сохранения с автоматическими временны́ми метками.

Давайте немного усложним условия. Пусть у вас есть несколько файлов, используемых вместе, например, исходный код проекта или файлы для вебсайта. Теперь, чтобы сохранить старую версию, вы должны скопировать весь каталог. Поддержка множества таких версий вручную неудобна и быстро становится дорогим удовольствием.

В некоторых играх сохранение — это и есть каталог с кучей файлов внутри. Игры скрывают детали от игрока и предоставляют удобный интерфейс для управления различными версиям этого каталога.

В системах управления версиями всё точно так же. У них у всех есть приятный интерфейс для управления каталогом с вашим скарбом. Можете сохранять состояние каталога так часто, как пожелаете, а затем восстановить любую из предыдущих сохраненных версий. Но, в отличие от компьютерных игр, они существенно экономят дисковое пространство. Обычно от версии к версии изменяется только несколько файлов, и то ненамного. Хранение лишь различий вместо полных копий требует меньше места.

Распределенное управление

А теперь представьте очень сложную компьютерную игру. Ее настолько сложно пройти, что множество опытных игроков по всему миру решили объединиться и использовать общие сохранения, чтобы попытаться выиграть. Прохождения на скорость — живой пример. Игроки, специализирующиеся на разных уровнях игры, объединяются, чтобы в итоге получить потрясающий результат.

Как бы вы организовали такую систему, чтобы игроки смогли легко получать сохранения других? А загружать свои?

В былые времена каждый проект использовал централизованное управление версиями. Какой-нибудь сервер хранил все сохраненные игры. И никто больше. Каждый держал лишь несколько сохранений на своей машине. Когда игрок хотел пройти немного дальше, он выкачивал самое последнее сохранение с главного сервера, играл немного, сохранялся и закачивал уже свое сохранение обратно на сервер, чтобы остальные могли им воспользоваться.

А что если игрок по какой-то причине захотел использовать более старую сохраненную игру? Возможно, нынешнее сохранение безвыигрышно, потому что

кто-то забыл взять некий игровой предмет еще на третьем уровне, и нужно найти последнее сохранение, где игру всё еще можно закончить. Или, может быть, хочется сравнить две более старые сохраненные игры, чтобы установить вклад конкретного игрока.

Может быть много причин вернуться к более старой версии, но выход один: нужно запросить ту старую сохраненную игру у центрального сервера. Чем больше сохраненных игр требуется, тем больше понадобится связываться с сервером.

Системы управления версиями нового поколения, к которым относится Git, известны как распределенные системы, их можно понимать как обобщение централизованных систем. Когда игроки загружаются с главного сервера, они получают каждую сохраненную игру, а не только последнюю. Они как бы зеркалируют центральный сервер.

Эти первоначальные операции клонирования могут быть ресурсоемкими, особенно при длинной истории, но сполна окупаются при длительной работе. Наиболее очевидная прямая выгода состоит в том, что если вам зачем-то потребуется более старая версия, взаимодействие с сервером не понадобится.

Глупые предрассудки

Широко распространенное заблуждение состоит в том, что распределенные системы непригодны для проектов, требующих официального централизованного хранилища. Ничто не может быть более далеким от истины. Получение фотоснимка не приводит к тому, что мы крадем чью-то душу. Точно так же клонирование главного хранилища не уменьшает его важность.

В первом приближении можно сказать, что все, что делает централизованная система управления версиями, хорошо сконструированная распределенная система может сделать лучше. Сетевые ресурсы просто дороже локальных. Хотя дальше мы увидим, что в распределенном подходе есть свои недостатки, вы вряд ли ошибетесь в выборе, руководствуясь этим приближенным правилом.

Небольшому проекту может понадобиться лишь частица функционала, предлагаемого такой системой. Но использование плохо масштабируемой системы для маленьких проектов подобно использованию римских цифр в расчетах с небольшими числами.

Кроме того, проект может вырасти сверх первоначальных ожиданий. Использовать Git с самого начала — это как держать наготове швейцарский нож, даже если вы всего лишь открываете им бутылки. Однажды вам безумно понадобится отвертка и вы будете рады, что под рукой есть нечто большее, чем простая открывалка.

Конфликты при слиянии

Для этой темы аналогия с компьютерной игрой становится слишком натянутой. Вместо этого, давайте вернемся к редактированию документа.

Итак, допустим, что Алиса вставила строчку в начале файла, а Боб — в конце. Оба они закачивают свои изменения. Большинство систем автоматически сделает разумный вывод: принять и соединить их изменения так, чтобы обе правки — и Алисы, и Боба — были применены.

Теперь предположим, что и Алиса, и Боб внесли разные изменения в одну и ту же строку. В этом случае невозможно продолжить без человеческого вмешательства. Тот из них, кто вторым закачает на сервер изменения, будет информирован о *конфликте слияния* (merge conflict), и должен либо предпочесть одно изменение другому, либо скорректировать всю строку.

Могут случаться и более сложные ситуации. Системы управления версиями разрешают простые ситуации сами и оставляют сложные для человека. Обычно такое их поведение поддается настройке.

Базовые операции

Прежде чем погружаться в дебри многочисленных команд Git, попробуйте воспользоваться приведенными ниже простыми примерами, чтобы немного освоиться. Каждый из них полезен, несмотря на свою простоту. На самом деле первые месяцы использования Git я не выходил за рамки материала этой главы.

Сохранение состояния

Собираетесь попробовать внести некие радикальные изменения? Предварительно создайте снимок всех файлов в текущем каталоге с помощью команд

```
$ git init
$ git add .
$ git commit -m "                "
```

Теперь, если новые правки всё испортили, можно восстановить первоначальную версию:

```
$ git reset --hard
```

Чтобы вновь сохранить состояние:

```
$ git commit -a -m "             "
```

Добавление, удаление, переименование

Приведенный выше пример отслеживает только те файлы, которые существовали при первом запуске **git add**. Если вы создали новые файлы или подкаталоги, придется сказать Git'у:

```
$ git add readme.txt Documentation
```

Аналогично, если хотите, чтобы Git забыл о некоторых файлах:

```
$ git rm   .h     .c
$ git rm -r    /
```

Git удалит эти файлы, если вы не удалили их сами.

Переименование файла — это то же, что удаление старого имени и добавления нового. Для этого есть **git mv**, которая имеет тот же синтаксис, что и команда **mv**. Например:

```
$ git mv bug.c feature.c
```

Расширенные отмена/возврат

Иногда просто хочется вернуться назад и забыть все изменения до определенного момента, потому что все они были неправильными. В таком случае

```
$ git log
```

покажет список последних коммитов и их хеши SHA1:

```
commit 766f9881690d240ba334153047649b8b8f11c664
Author: Bob <bob@example.com>
Date:   Tue Mar 14 01:59:26 2000 -0800

         printf()   write().

commit 82f5ea346a2e651544956a8653c0f58dc151275c
Author: Alice <alice@example.com>
Date:   Thu Jan 1 00:00:00 1970 +0000

              .
```

Для указания коммита достаточно первых нескольких символов его хеша, но можете скопировать и весь хеш. Наберите:

```
$ git reset --hard 766f
```

для восстановления состояния до указанного коммита и удаления всех последующих безвозвратно.

Возможно, в другой раз вы захотите быстро перескочить к старому состоянию. В этом случае наберите

```
$ git checkout 82f5
```

Эта команда перенесет вас назад во времени, сохранив при этом более новые коммиты. Однако, как и в фантастических фильмах о путешествиях во времени, если теперь вы отредактируете и закоммитите код, то попадете в альтернативную реальность, потому что ваши действия отличаются от тех, что были в прошлый раз.

Эта альтернативная реальность называется «веткой» (branch, прим. пер.), и чуть позже мы поговорим об этом подробнее. А сейчас просто запомните, что команда

```
$ git checkout master
```

вернет вас обратно в настоящее. Кроме того, чтобы не получать предупреждений от Git, всегда делайте commit или сбрасывайте изменения перед запуском checkout.

Еще раз воспользуемся аналогией с компьютерными играми:

- **git reset --hard**: загружает ранее сохраненную игру и удаляет все версии, сохраненные после только что загруженной.
- **git checkout**: загружает старую игру, но если вы продолжаете играть, состояние игры будет отличаться от более новых сохранений, которые вы сделали в первый раз. Любая игра, которую вы теперь сохраняете, попадает в отдельную ветку, представляющую альтенативную реальность, в которую вы попали. Мы обсудим это позже.

Можно также восстановить только определенные файлы и подкаталоги, перечислив их имена после команды:

```
$ git checkout 82f5    - .       .
```

Будьте внимательны: такая форма **checkout** может молча перезаписать файлы. Чтобы избежать неприятных неожиданностей, выполняйте commit перед checkout, особенно если вы только изучаете Git. Вообще, если вы не уверены в какой-либо операции, будь то команда Git или нет, выполните предварительно **git commit -a**.

Не любите копировать и вставлять хеши? Используйте

```
$ git checkout :/"        "
```

для перехода на коммит, чье описание начинается с приведенной строки.

Можно также запросить 5-ое с конца сохраненное состояние:

```
$ git checkout master~5
```

Откаты

В зале суда пункты протокола могут вычеркиваться прямо во время слушания. Подобным образом и вы можете выбирать коммиты для отмены.

```
$ git commit -a
$ git revert 1b6d
```

отменит коммит с заданным хешем. Откат будет сохранен в виде нового коммита. Можете запустить **git log**, чтобы убедиться в этом.

Создание списка изменений

Некоторым проектам нужен список изменений (changelog, прим. пер.). Создайте его такой командой:

```
$ git log > ChangeLog
```

Скачивание файлов

Получить копию проекта под управлением Git можно, набрав

```
$ git clone git://   /  / /
```

Например, чтобы получить все файлы, которые я использовал для создания этого документа,

```
$ git clone git://git.or.cz/gitmagic.git
```

Позже мы поговорим о команде **clone** подробнее.

Держа руку на пульсе

Если вы уже загрузили копию проекта с помощью **git clone**, можете обновить ее до последней версии, используя

```
$ git pull
```

Безотлагательная публикация

Допустим, вы написали скрипт, которым хотите поделиться с другими. Можно просто предложить им скачивать его с вашего компьютера, но если они будут делать это когда вы дорабатываете его или добавляете экспериментальную функциональность, у них могут возникнуть проблемы. Очевидно, поэтому и существуют циклы разработки. Разработчики могут постоянно работать над

проектом, но общедоступным они делают свой код только после того, как приведут его в приличный вид.

Чтобы сделать это с помощью Git, выполните в каталоге, где лежит ваш скрипт,

```
$ git init
$ git add .
$ git commit -m "       "
```

Затем скажите вашим пользователям запустить

```
$ git clone   .     :/  / /
```

чтобы загрузить ваш скрипт. Здесь подразумевается, что у них есть доступ по ssh. Если нет, запустите **git daemon** и скажите пользователям запустить эту команду вместо вышеприведенной:

```
$ git clone git://  .     /  / /
```

С этих пор всякий раз, когда ваш скрипт готов к релизу, выполняйте

```
$ git commit -a -m "         "
```

и ваши пользователи смогут обновить свои версии, перейдя в каталог, с вашим скриптом и набрав

```
$ git pull
```

Ваши пользователи никогда не наткнутся на версию скрипта, которую вы не хотите им показывать.

Что я сделал?

Выясните, какие изменения вы сделали со времени последнего коммита:

```
$ git diff
```

Или со вчерашнего дня:

```
$ git diff "@{yesterday}"
```

Или между определенной версией и версией, сделанной 2 коммита назад:

```
$ git diff 1b6d "master~2"
```

В каждом случае на выходе будет патч, который может быть применен с помощью **git apply**. Попробуйте также:

```
$ git whatchanged --since="2 weeks ago"
```

Часто вместо этого я использую для просмотра истории qgit, из-за приятного интерфейса, или tig с текстовым интерфейсом, который хорошо работает через

медленное соединение. Как вариант, установите веб-сервер, введите **git instaweb** и запустите любой веб-браузер.

Упражнение

Пусть A, B, C, D — четыре последовательных коммита, где B отличается от A лишь несколькими удаленными файлами. Мы хотим вернуть эти файлы в D. Как мы можем это сделать?

Существует как минимум три решения. Предположим, что мы находимся на D.

1. Разница между A и B — удаленные файлы. Мы можем создать патч, отражающий эти изменения, и применить его:

   ```
   $ git diff B A | git apply
   ```

2. Поскольку в коммите A мы сохранили файлы, то можем восстановить их:

   ```
   $ git checkout A foo.c bar.h
   ```

3. Мы можем рассматривать переход от A к B как изменения, которые хотим отменить:

   ```
   $ git revert B
   ```

Какой способ лучше? Тот, который вам больше нравится. С помощью Git легко получить желаемое, и часто существует много способов это сделать.

Все о клонировании

В старых системах управления версиями стандартная операция для получения файлов — это checkout. Вы получаете набор файлов в конкретном сохраненном состоянии.

В Git и других распределенных системах управления версиями стандартный способ — клонирование. Для получение файлов вы создаете «клон» всего хранилища. Другими словами, вы фактически создаете зеркало центрального сервера. При этом всё, что можно делать с основным хранилищем, можно делать и с локальным.

Синхронизация компьютеров

Я вполне приемлю создание архивов или использование **rsync** для резервного копирования и простейшей синхронизации. Но я работаю то на ноутбуке, то на стационарном компьютере, которые могут никак между собой не взаимодействовать между этим.

Создайте хранилище Git и закоммитьте файлы на одном компьютере. А потом выполните на другом

```
$ git clone     .     :/  / /
```

для создания второго экземпляра файлов и хранилища Git. С этого момента команды

```
$ git commit -a
$ git pull     .     :/  / /    HEAD
```

будут «втягивать» состояние файлов с другого компьютера на тот, где вы работаете. Если вы недавно внесли конфликтующие изменения в один и тот же файл, Git даст вам знать, и нужно будет сделать коммит заново после разрешения ситуации.

Классическое управление исходным кодом

Создайте хранилище Git для ваших файлов:

```
$ git init
$ git add .
$ git commit -m "          "
```

На центральном сервере создайте так называемое «голое» (bare) хранилище Git в неком каталоге:

```
$ mkdir proj.git
$ cd proj.git
$ git init --bare
$  #      «         »: GIT_DIR=proj.git git init
```

Запустите Git-демон, если необходимо:

```
$ git daemon --detach #
```

Для создания нового пустого хранилища Git на публичных серверах следуйте их инструкциям. Обычно, нужно заполнить форму на веб-странице.

Отправьте ваши изменения в центральное хранилище вот так:

```
$ git push git://      .    /  / /proj.git HEAD
```

Для получения ваших исходников разработчик вводит

```
$ git clone git://      .    /  / /proj.git
```

После внесения изменений разработчик сохраняет изменения локально:

```
$ git commit -a
```

Для обновления до последней версии:

```
$ git pull
```

Любые конфликты слияния нужно разрешить и закоммитить:

```
$ git commit -a
```

Для выгрузки локальных изменений в центральное хранилище:

```
$ git push
```

Если на главном сервере были новые изменения, сделанные другими разработчиками, команда push не сработает. В этом случае разработчику нужно будет вытянуть к себе (pull) последнюю версию, разрешить возможные конфликты слияний и попробовать еще раз.

Голые (bare) хранилища

Голое (bare) хранилище называются так потому, что у него нет рабочего каталога. Оно содержит только файлы, которые обычно скрыты в подкаталоге .git. Другими словами, голое хранилище содержит историю изменений, но не содержит снимка какой-либо определенной версии.

Голое хранилище играет роль, похожую на роль основного сервера в централизованной системе управления версиями: это дом вашего проекта. Разработчики клонируют из него проект и закачивают в него свежие официальные изменения. Как правило, оно располагается на сервере, который не делает почти ничего кроме раздачи данных. Разработка идет в клонах, поэтому домашнее хранилище может обойтись и без рабочего каталога.

Многие команды Git не работают в голых хранилищах, если переменная среды GIT_DIR не содержит путь до хранилища и не указан параметр --bare.

Push или pull?

Зачем вводится команда push, вместо использования уже знакомой pull? Прежде всего, pull не работает в голых хранилищах, вместо нее нужно использовать команду fetch, которая будет рассмотрена позже. Но даже если держать на центральном сервере нормальное хранилище, использование команды pull в нем будет затруднительным. Нужно будет сначала войти на сервер интерактивно и сообщить команде pull адрес машины, с которой мы хотим забрать изменения. Этому могут мешать сетевые брандмауэры (firewall), но в первую очередь: что если у нас нет интерактивного доступа к серверу?

Тем не менее, не рекомендутся push-ить в хранилище помимо этого случая – из-за путаницы, которая может возникнуть, если у целевого хранилища есть рабочий каталог.

Короче говоря, пока изучаете Git, push-те только в голые хранилища. В остальных случаях pull-те.

Создание форка проекта

Не нравится путь развития проекта? Думаете, можете сделать лучше? Тогда на вашем сервере выполните

```
$ git clone git://     .    /  / /
```

Теперь расскажите всем о форке (ответвлении, прим. пер.) проекта на вашем сервере.

Позже вы сможете в любой момент втянуть к себе изменения из первоначального проекта:

```
$ git pull
```

Максимальные бэкапы

Хотите иметь множество защищенных, географически разнесенных запасных архивов? Если в вашем проекте много разработчиков, ничего делать не нужно! Каждый клон — это и есть резервная копия; не только текущего состояния, но и всей истории изменений проекта. Благодаря криптографическому хешированию, повреждение какого-либо из клонов будет обнаружено при первой же попытке взаимодействия с другими клонами.

Если ваш проект не такой популярный, найдите как можно больше серверов для размещения клонов.

Особо беспокоящимся рекомендуется всегда записывать самый последний 20-байтный SHA1 хеш HEAD в каком-нибудь безопасном месте. Оно должно быть безопасным, а не тайным. Например, хороший вариант — публикация в газете, потому что атакующему сложно изменить каждый экземпляр газеты.

Многозадачность со скоростью света

Скажем, вы хотите работать над несколькими функциями параллельно. Тогда закоммитьте ваши изменения и запустите

```
$ git clone . /   /   /
```

Благодаря жёстким ссылкам создание локального клона требует меньше времени и места, чем простое копирование.

Теперь вы можете работать с двумя независимыми функциями одновременно. Например, можно редактировать один клон, пока другой компилируется. В любой момент можно сделать коммит и вытянуть изменения из другого клона:

```
$ git pull /   /   HEAD
```

Партизанское управление версиями

Вы работаете над проектом, который использует другую систему управления версиями, и вам очень не хватает Git? Тогда создайте хранилище Git в своем рабочем каталоге:

```
$ git init
$ git add .
$ git commit -m "          "
```

затем склонируйте его:

```
$ git clone . /   /   /
```

Теперь перейдите в этот новый каталог и работайте в нем вместо основного, используя Git в свое удовольствие. В какой-то момент вам понадобиться синхронизировать изменения со всеми остальными — тогда перейдите в изначальный каталог, синхронизируйте его с помощью другой системы управления версиями и наберите

```
$ git add .
$ git commit -m "                "
```

Теперь перейдите в новый каталог и запустите

```
$ git commit -a -m "               "
$ git pull
```

Процедура передачи изменений остальным зависит от другой системы управления версиями. Новый каталог содержит файлы с вашими изменениями. Запустите команды другой системы управления версиями, необходимые для загрузки файлов в центральное хранилище.

Subversion (вероятно, наилучшая централизованная система управления версиями) используется неисчислимым множеством проектов. Команда **git svn** автоматизирует описанный процесс для хранилищ Subversion, а также может быть использована для экспорта проекта Git в хранилище Subversion.

Mercurial

Mercurial — похожая система управления версиями, которая может работать в паре с Git практически без накладок. С расширением hg-git пользователь Mercurial может без каких либо потерь push-ить и pull-ить из хранилища Git.

Получить hg-git можно с помощью Git:

```
$ git clone git://github.com/schacon/hg-git.git
```

или Mercurial:

```
$ hg clone http://bitbucket.org/durin42/hg-git/
```

К сожалению, мне неизвестен аналогичное расширение для Git. Поэтому я рекомендую использовать Git, а не Mercurial, для центрального хранилища, даже если вы предпочитаете Mercurial. Для проектов, использующих Mercurial, обычно какой-нибудь доброволец поддерживает параллельное хранилище Git для привлечения пользователей последнего, тогда как проекты, использующие Git, благодаря hg-git автоматически доступны пользователям Mercurial.

Хотя расширение может сконвертировать хранилище Mercurial в Git путем push'а в пустое хранилище, эту задачу легче решить, используя сценарий hg-fast-export.sh, доступный как

```
$ git clone git://repo.or.cz/fast-export.git
```

Для преобразования выполните в пустом каталоге

```
$ git init
$ hg-fast-export.sh -r /hg/repo
```

после добавления сценария в ваш $PATH.

Bazaar

Упомянем вкратце Bazaar, так как это самая популярная свободная распределенная система управления версиями после Git и Mercurial.

Bazaar относительно молод, поэтому у него есть преимущество идущего следом. Его проектировщики могут учиться на ошибках предшественников и избавиться от исторически сложившихся неровностей. Кроме того, его разработчики заботятся о переносимости и взаимодействии с другими системами управления версиями.

Расширение bzr-git позволяет (в какой-то степени) пользователям Bazaar работать с хранилищами Git. Программа tailor конвертирует хранилища Bazaar в Git и может делать это с накоплением, тогда как bzr-fast-export хорошо приспособлена для разовых преобразований.

Почему я использую Git

Изначально я выбрал Git потому, что слышал, что он в состоянии справиться с совершенно неуправляемыми исходными текстами ядра Linux. Я никогда не ощущал потребности сменить его на что-то другое. Git работает замечательно и мне еще только предстоит напороться на его недостатки. Так как я в основном использую Linux, проблемы на других системах меня не касаются.

Я также предпочитаю программы на C и сценарии на bash исполняемым файлам вроде сценариев на Python-е: у них меньше зависимостей, и я привык к быстрому выполнению.

Я думал о том, как можно улучшить Git, вплоть до того, чтобы написать собственный инструмент, похожий на Git; но только как академическое упражнение. Завершив проект, я бы все равно продолжил пользоваться Git, потому что выигрыш слишком мал, чтобы оправдать использование самодельной системы.

Естественно, ваши потребности и пожелания вероятно отличаются от моих и вы, возможно, лучше уживетесь с другой системой. И всё же вы не слишком ошибетесь, используя Git.

Чудеса ветвления

Возможности мгновенного ветвления и слияния — самые замечательный особенности Git.

Задача: внешние факторы неизбежно влекут переключение внимания. Серьезная ошибка в уже выпущенной версии обнаруживается без предупреждения. Срок сдачи определённой функциональности приближается. Разработчик, помощь которого нужна вам в работе над ключевой частью проекта, собирается в отпуск. Одним словом, вам нужно срочно бросить все, над чем вы трудитесь в настоящий момент, и переключиться на совершенно другие задачи.

Прерывание хода ваших мыслей может серьезно снизить эффективность работы, и чем сложнее переключение между процессами, тем больше будет потеря. При централизованном управлении версиями мы вынуждены скачивать свежую рабочую копию с центрального сервера. Распределенная система лучше: мы можем клонировать нужную версию локально.

Однако клонирование все же предполагает копирование всего рабочего каталога, как и всей истории изменений до настоящего момента. Хотя Git и снижает затратность этого действия за счет возможности совместного использования файлов и жестких ссылок, но все файлы проекта придется полностью воссоздать в новом рабочем каталоге.

Решение: у Git есть более удобный инструмент для таких случаев, который сэкономит и время, и дисковое пространство по сравнению с клонированием — это **git branch** (branch — ветка, прим. пер.).

Этим волшебным словом файлы в вашем каталоге мгновенно преобразуются от одной версии к другой. Это изменение позволяет сделать намного больше, чем просто вернуться назад или продвинуться вперед в истории. Ваши файлы могут изменится с последней выпущенной версии на экспериментальную, с

экспериментальной — на текущую версию в разработке, с нее — на версию вашего друга и так далее.

Кнопка босса

Играли когда-нибудь в одну из таких игр, где при нажатии определеной клавиши («кнопки босса»), на экране мгновенно отображается таблица или что-то вроде того? То есть, если в офис зашел начальник, а вы играете в игру, вы можете быстро ее скрыть.

В каком-нибудь каталоге:

```
$ echo "             " > myfile.txt
$ git init
$ git add .
$ git commit -m "          "
```

Мы создали хранилище Git, содержащее один текстовый файл с определенным сообщением. Теперь выполните

```
$ git checkout -b boss #      ,
$ echo "             " > myfile.txt
$ git commit -a -m "        "
```

Это выглядит так, будто мы только что перезаписали файл и сделали коммит. Но это иллюзия. Наберите

```
$ git checkout master #
```

Вуаля! Текстовый файл восстановлен. А если босс решит сунуть нос в этот каталог, запустите

```
$ git checkout boss #            ,
```

Вы можете переключаться между двумя версиями этого файла так часто, как вам хочется и делать коммиты каждой из них независимо.

Грязная работа

Допустим, вы работаете над некой функцией, и вам зачем-то понадобилось вернуться на три версии назад и временно добавить несколько операторов вывода, чтобы посмотреть как что-либо работает. Тогда введите

```
$ git commit -a
$ git checkout HEAD~3
```

Теперь вы можете добавлять временный черновой код в любых местах. Можно даже закоммитить эти изменения. Когда закончите, выполните

```
$ git checkout master
```

чтобы вернуться к исходной работе. Заметьте, что любые изменения, не внесенные в коммит, будут перенесены.

А что, если вы все-таки хотели сохранить временные изменения? Запросто:

```
$ git checkout -b dirty
```

а затем сделайте коммит перед возвращением в ветку master. Всякий раз, когда вы захотите вернуться к черновым изменениям, просто выполните

```
$ git checkout dirty
```

Мы говорили об этой команде в одной из предыдущих глав, когда обсуждали загрузку старых состояний. Теперь у нас перед глазами полная картина: файлы изменились к нужному состоянию, но мы должны покинуть главную ветку. Любые коммиты, сделанные с этого момента, направят файлы по другому пути, к которому можно будет вернуться позже.

Другими словами, после переключения на более старое состояние Git автоматически направляет вас по новой безымянной ветке, которой можно дать имя и сохранить ее с помощью **git checkout -b**.

Быстрые исправления

Ваша работа в самом разгаре, когда вдруг выясняется, что нужно все бросить и исправить только что обнаруженную ошибку в коммите «1b6d…»:

```
$ git commit -a
$ git checkout -b fixes 1b6d
```

После исправления ошибки сделайте

```
$ git commit -a -m "          "
$ git checkout master
```

и вернитесь к работе над вашими исходными задачами.

Вы можете даже «влить» только что сделанное исправление ошибки в основную ветку:

```
$ git merge fixes
```

Слияния

В некоторых системах управления версиями создавать ветки легко, а вот сливать их воедино трудно. В Git слияние столь тривиально, что вы можете его не заметить.

На самом деле мы сталкивались со слияниями уже давно. Команда **pull** по сути получает коммиты, а затем сливает их с вашей текущей веткой. Если у вас нет

локальных изменений, слияние произойдет само собой, как вырожденный случай вроде получения последней версии в централизованной системе управления версиями. Если же у вас есть локальные изменения, Git автоматически произведет слияние и сообщит о любых конфликтах.

Обычно у коммита есть один «родитель», а именно предыдущий коммит. Слияние веток приводит к коммиту как минимум с двумя родителями. Отсюда возникает вопрос: к какому коммиту на самом деле отсылает HEAD~10? Коммит может иметь несколько родителей, так за которым из них следовать далее?

Оказывается, такая запись всегда выбирает первого родителя. Это хороший выбор, потому что текущая ветка становятся первым родителем во время слияния. Часто вас интересуют только изменения, сделанные вами в текущей ветке, а не те, которые влились из других веток.

Вы можете обращаться к конкретному родителю с помощью символа «^». Например, чтобы показать запись в журнале от второго родителя, наберите

```
$ git log HEAD^2
```

Для первого родителя номер можно опустить. Например, чтобы показать разницу с первым родителем, введите

```
$ git diff HEAD^
```

Вы можете сочетать такую запись с другими. Например,

```
$ git checkout 1b6d^^2~10 -b ancient
```

создаст новую ветку «ancient» («древняя», прим. пер.), отражающую состояние на десять коммитов назад от второго родителя первого родителя коммита, начинающегося с 1b6d.

Непрерывный рабочий процесс

В производстве техники часто бывает, что второй шаг плана должен ждать завершения первого шага. Автомобиль, нуждающийся в ремонте, может тихо стоять в гараже до прибытия с завода конкретной детали. Прототип может ждать производства чипа, прежде чем разработка будет продолжена.

И в разработке ПО может быть то же. Вторая порция новой функциональности может быть вынуждена ожидать выпуска и тестирования первой части. Некоторые проекты требуют проверки вашего кода перед его принятием, так что вы должны дождаться утверждения первой части, прежде чем начинать вторую.

Благодаря безболезненным ветвлению и слиянию, мы можем изменить правила и работать над второй частью до того, как первая официально будет готова. Допустим, вы закоммитили первую часть и выслали ее на проверку. Скажем, вы в ветке master. Теперь смените ветку:

```
$ git checkout -b part2 #    2
```

Затем работайте над второй частью, попутно внося коммиты ваших изменений. Человеку свойственно ошибаться, и часто вы хотите вернуться и поправить что-то в первой части. Если вы везучи или очень искусны, можете пропустить эти строки.

```
$ git checkout master  #                 .
$    _
$ git commit -a        #
$ git checkout part2   #                  .
$ git merge master     #                  .
```

В конечном счете, первая часть утверждена:

```
$ git checkout master  #                 .
$                  #          !
$ git merge part2      #             .
$ git branch -d part2  #          part2.
```

Теперь вы снова в ветке master, а вторая часть — в вашем рабочем каталоге.

Этот прием легко расширить на любое количество частей. Столь же легко сменить ветку задним числом. Предположим, вы слишком поздно обнаружили, что должны были создать ветку семь коммитов назад. Тогда введите:

```
$ git branch -m master part2 #              master   part2.
$ git branch master HEAD~7   #             master            .
```

Теперь ветка master содержит только первую часть, а ветка part2 — всё остальное. В последней мы и находимся. Мы создали ветку master, не переключаясь на нее, потому что хотим продолжить работу над part2. Это непривычно: до сих пор мы переключались на ветки сразу же после их создания, вот так:

```
$ git checkout HEAD~7 -b master  #                        .
```

Изменяем состав смеси

Предположим, вам нравится работать над всеми аспектами проекта в одной и той же ветке. Вы хотите закрыть свой рабочий процесс от других, чтобы все видели ваши коммиты только после того, как они будут хорошо оформлены. Создайте пару веток:

```
$ git branch sanitized    #                       .
$ git checkout -b medley  #                               .
```

Далее делайте всё что нужно: исправляйте ошибки, добавляйте новые функции, добавляйте временный код и так далее, при этом почаще выполняя коммиты. После этого

```
$ git checkout sanitized
```

```
$ git cherry-pick medley^^
```

применит коммит «пра-родителя» головы ветки «medley» к ветке «sanitized». Правильно подбирая элементы, вы сможете создать ветку, в которой будет лишь окончательный код, а связанные между собой коммиты будут собраны вместе.

Управление Ветками

Для просмотра списка всех веток наберите

```
$ git branch
```

По умолчанию вы начинаете с ветки под названием «master». Кому-то нравится оставлять ветку «master» нетронутой и создавать новые ветки со своими изменениями.

Опции **-d** и **-m** позволяют удалять и перемещать (переименовывать) ветки. Смотрите **git help branch**.

Ветка «master» — это удобная традиция. Другие могут предполагать, что в вашем хранилище есть ветка с таким именем и что она содержит официальную версию проекта. Хотя вы можете переименовать или уничтожить ветку «master», лучше соблюсти общее соглашение.

Временные Ветки

Через какое-то время вы можете обнаружить, что создаете множество временных веток для одной и той же краткосрочной цели: каждая такая ветка всего лишь сохраняет текущее состояние, чтобы вы могли вернуться назад и исправить серьезную ошибку или сделать что-то еще.

Это похоже на то, как вы переключаете телевизионные каналы, чтобы посмотреть что показывают по другим. Но вместо того, чтобы нажать на пару кнопок, вам нужно создавать, выбирать (checkout), сливать (merge) а затем удалять временные ветки. К счастью, в Git есть сокращенная команда, столь же удобная, как пульт дистанционного управления.

```
$ git stash
```

Эта команда сохранит текущее состояние в во временном месте («тайнике», stash) и востановит предыдущее состояние. Ваш каталог становиться точно таким, каким был до начала редактирования, и вы можете исправить ошибки, загрузить удаленные изменения и тому подобное. Когда вы хотите вернуться назад в состояние «тайника», наберите:

```
$ git stash apply #     ,                          .
```

Можно создавать несколько тайников, используя их по-разному. Смотрите **git help stash**. Как вы могли догадаться, Git оставляет ветки «за кадром» при выполнении этого чудесного приема.

Работайте как вам нравится

Возможно, вы сомневаетесь, стоят ли ветки таких хлопот. В конце концов, клоны почти столь же быстрые и вы можете переключаться между ними с помощью **cd** вместо загадочных команд Git.

Посмотрим на веб-браузеры. Зачем нужна поддержка вкладок вдобавок к окнам? Поддержка и тех, и других позволяет приспособиться к широкому разнообразию стилей работы. Некоторым пользователям нравится держать открытым единственное окно и использовать вкладки для множества веб-страниц. Другие могут впасть в другую крайность: множество окон без вкладок вообще. Третьи предпочтут нечто среднее.

Ветки похожи на вкладки для рабочего каталога, а клоны — на новые окна браузера. Эти операции быстры и выполняются локально, так почему бы не поэкспериментировать и не найти наиболее удобную для себя комбинацию? Git позволяет работать в точности так, как вам нравится.

Уроки истории

Вследствие распределенной природы Git, историю изменений можно легко редактировать. Однако, если вы вмешиваетесь в прошлое, будьте осторожны: изменяйте только ту часть истории, которой владеете вы и только вы. Иначе, как народы вечно выясняют, кто же именно совершил и какие бесчинства, так и у вас будут проблемы с примирением при попытке совместить разные деревья истории.

Некоторые разработчики убеждены, что история должна быть неизменна со всеми огрехами и прочим. Другие считают, что деревья нужно делать презентабельными перед выпуском их в публичный доступ. Git учитывает оба мнения. Переписывание истории, как и клонирование, ветвление и слияние, — лишь еще одна возможность, которую дает вам Git. Разумное ее использование зависит только от вас.

Оставаясь корректным

Только что сделали коммит и поняли, что должны были ввести другое описание? Запустите

```
$ git commit --amend
```

чтобы изменить последнее описание. Осознали, что забыли добавить файл? Запустите **git add**, чтобы это сделать, затем выполните вышеуказанную команду.

Захотелось добавить еще немного изменений в последний коммит? Так сделайте их и запустите

```
$ git commit --amend -a
```

…И кое-что еще

Давайте представим, что предыдущая проблема на самом деле в десять раз хуже. После длительной работы вы сделали ряд коммитов; но вы не очень-то довольны тем, как они организованы, и кое-какие описания коммитов надо бы слегка переформулировать. Тогда запустите

```
$ git rebase -i HEAD~10
```

и последние десять коммитов появятся в вашем любимом редакторе (задается переменной окружения $EDITOR). Например:

```
pick 5c6eb73          repo.or.cz
pick a311a64              «          »
pick 100834f            push   Makefile
```

Теперь вы можете:

- Убирать коммиты, удаляя строки.
- Менять порядок коммитов, переставляя строки.
- Заменять «pick» на:
  - «edit» для внесения правок в коммиты;
  - «reword» для изменения описания в журнале;
  - «squash» для слияния коммита с предыдущим;
  - «fixup», чтобы слить коммит с предыдущим, отбросив его описание.

Сохраните файл и закройте редактор. Если вы отметили коммит для исправлений, запустите

```
$ git commit --amend
```

Если нет, запустите

```
$ git rebase --continue
```

Одним словом, делайте коммиты как можно раньше и как можно чаще — вы всегда сможете навести порядок при помощи rebase.

Локальные изменения сохраняются

Предположим, вы работаете над активным проектом. За какое-то время вы делаете несколько коммитов, затем синхронизируетесь с официальным деревом через слияние. Цикл повторяется несколько раз, пока вы не будете готовы влить изменения в центральное дерево.

Однако теперь история изменений в локальном клоне Git представляет собой кашу из ваших и официальных изменений. Вам бы хотелось видеть все свои изменения непрерывной линией, а затем — все официальные изменения.

Это работа для команды **git rebase**, описанной выше. Зачастую, имеет смысл использовать флаг **--onto** и убрать переплетения.

Также смотрите **git help rebase** для получения подробных примеров использования этой замечательной команды. Вы можете расщеплять коммиты. Вы можете даже переупорядочивать ветки.

Переписывая историю

Время от времени вам может понадобиться в системе управления версиями аналог «замазывания» людей на официальных фотографиях, как бы стирающего их из истории в духе сталинизма. Например, предположим, что мы уже собираемся выпустить релиз проекта, но он содержит файл, который не должен стать достоянием общественности по каким-то причинам. Возможно, я сохранил номер своей кредитки в текстовый файл и случайно добавил его в проект. Удалить файл недостаточно: он может быть доступен из старых коммитов. Нам надо удалить файл из всех ревизий:

```
$ git filter-branch --tree-filter 'rm       /     /   ' HEAD
```

Смотрите **git help filter-branch**, где обсуждается этот пример и предлагается более быстрый способ решения. Вообще, **filter-branch** позволяет изменять существенные части истории при помощи одной-единственной команды.

После этой команды каталог .git/refs/original будет описывать состояние, которое было до ее вызова. Убедитесь, что команда filter-branch сделала то, что вы хотели, и если хотите опять использовать эту команду, удалите этот каталог.

И, наконец, замените клоны вашего проекта исправленной версией, если собираетесь в дальнейшем с ними взаимодействовать.

Создавая Историю

Хотите перевести проект под управление Git? Если сейчас он находится под управлением какой-либо из хорошо известных систем управления версиями,

то вполне вероятно, что кто-нибудь уже написал необходимые скрипты для экспорта всей истории проекта в Git.

Если нет, то смотрите в сторону команды **git fast-import**, которая считывает текстовый ввод в специальном формате для создания истории Git с нуля. Обычно скрипт, использующий эту команду, бывает слеплен наспех для единичного запуска, переносящего весь проект за один раз.

В качестве примера вставьте такие строки во временный файл, вроде */tmp/history*:

```
commit refs/heads/master
committer Alice <alice@example.com> Thu, 01 Jan 1970 00:00:00 +0000
data <<EOT
         .
EOT

M 100644 inline hello.c
data <<EOT
#include <stdio.h>

int main() {
  printf("Hello, world!\n");
  return 0;
}
EOT

commit refs/heads/master
committer Bob <bob@example.com> Tue, 14 Mar 2000 01:59:26 -0800
data <<EOT
     printf()   write()
EOT

M 100644 inline hello.c
data <<EOT
#include <unistd.h>

int main() {
  write(1, "Hello, world!\n", 14);
  return 0;
}
EOT
```

Затем создайте хранилище Git из этого временного файла при помощи команд:

```
$ mkdir project;  cd project;  git  init  $  git  fast-import  --date-format=rfc2822 < /tmp/history
```

Вы можете извлечь последнюю версию проекта с помощью

```
$ git checkout master .
```

Команда **git fast-export** преобразует любое хранилище в формат, понятный команде **git fast-import**. Ее вывод можно использовать как образец для написания скриптов преобразования, или для переноса хранилищ в понятном человеку формате. Конечно, с помощью этих команд можно пересылать хранилища текстовых файлов через каналы передачи текста.

Когда же все пошло не так?

Вы только что обнаружили, что кое-какой функционал вашей программы не работает, но вы совершенно отчетливо помните, что он работал всего несколько месяцев назад. Ох… Откуда же взялась ошибка? Вы же это проверяли сразу как разработали.

В любом случае, уже слишком поздно. Однако, если вы фиксировали свои изменения достаточно часто, то Git сможет точно указать проблему:

```
$ git bisect start
$ git bisect bad HEAD
$ git bisect good 1b6d
```

Git извлечет состояние ровно посередине. Проверьте работает ли то, что сломалось, и если все еще нет,

```
$ git bisect bad
```

Если же работает, то замените «bad» на «good». Git снова переместит вас в состояние посередине между «хорошей» и «плохой» ревизиями, сужая круг поиска. После нескольких итераций, этот двоичный поиск приведет вас к тому коммиту, на котором возникла проблема. После окончания расследования, вернитесь в исходное состояние командой

```
$ git bisect reset
```

Вместо ручного тестирования каждого изменения автоматизируйте поиск, запустив

```
$ git bisect run my_script
```

По возвращаемому значению заданной команды, обычно одноразового скрипта, Git будет отличать хорошее состояние от плохого. Скрипт должен вернуть 0, если нынешний коммит хороший; 125, если его надо пропустить; и любое другое число от 1 до 127, если он плохой. Отрицательное возвращаемое значение прерывает команду bisect.

Вы можете сделать многим больше: страница помощи поясняет, как визуализировать bisect, проанализировать или воспроизвести ее журнал, или исключить заведомо хорошие изменения для ускорения поиска.

Из-за кого все пошло не так?

Как и во многих других системах управления версиями, в Git есть команда blame (ответственность, прим. пер.):

```
$ git blame bug.c
```

Она снабжает каждую строку выбранного файла примечаниями, раскрывающими, кто и когда последним ее редактировал. В отличие же от многих других систем управления версиями, эта операция происходит без соединения с сетью, выбирая данные с локального диска.

Личный опыт

В централизованных системах управления версиями изменения истории — достаточно сложная операция, и доступна она лишь администраторам. Клонирование, ветвление и слияние невозможны без взаимодействия по сети. Так же обстоят дела и с базовыми операциями, такими как просмотр истории или фиксация изменений. В некоторых системах сетевое соединение требуется даже для просмотра собственных изменений, или открытия файла для редактирования.

Централизованные системы исключают возможность работы без сети и требуют более дорогой сетевой инфраструктуры, особенно с увеличением количества разработчиков. Что важнее, все операции происходят медленнее, обычно до такой степени, что пользователи избегают пользоваться «продвинутыми» командами без крайней необходимости. В радикальных случаях это касается даже большинства базовых команд. Когда пользователи вынуждены запускать медленные команды, производительность страдает из-за прерываний рабочего процесса.

Я испытал этот феномен на себе. Git был моей первой системой управления версиями. Я быстро привык к нему и стал относится к его возможностям как к должному. Я предполагал, что и другие системы похожи на него: выбор системы управления версиями не должен отличаться от выбора текстового редактора или браузера.

Когда немного позже я был вынужден использовать централизованную систему управления версиями, я был шокирован. Ненадежное интернет-соединение не имеет большого значения при использовании Git, но делает разработку невыносимой, когда от него требуют надежности как у жесткого диска. Вдобавок я обнаружил, что стал избегать некоторых команд из-за задержек в их выполнении, что помешало мне следовать предпочтительному рабочему процессу.

Когда мне было нужно запустить медленную команду, нарушение хода моих мыслей оказывало несоизмеримый ущерб разработке. Ожидая окончания связи с сервером, я вынужден был заниматься чем-то другим, чтобы скоротать

время; например, проверкой почты или написанием документации. К тому времени, как я возвращался к первоначальной задаче, выполнение команды было давно закончено, но мне приходилось тратить уйму времени, чтоб вспомнить, что именно я делал. Люди не очень приспособлены для переключения между задачами.

Кроме того, есть интересный эффект «трагедии общественных ресурсов»: предвидя будущую перегруженность сети, некоторые люди в попытке предотвратить грядущие задержки начинают использовать более широкие каналы, чем им реально требуется для текущих задач. Суммарная активность увеличивает загрузку сети, поощряя людей задействовать всё более высокоскоростные каналы для предотвращения еще больших задержек.

Многопользовательский Git

Сначала я использовал Git для личного проекта, в котором был единственным разработчиком. Среди команд, относящихся к распределенным свойствам Git, мне были нужны только **pull** и **clone**, чтобы хранить один и тот же проект в разных местах.

Позднее я захотел опубликовать свой код при помощи Git и включать изменения помощников. Мне пришлось научиться управлять проектами, в которых участвуют многие люди по всему миру. К счастью, в этом сильная сторона Git и, возможно, сам смысл его существования.

Кто я?

Каждый коммит содержит имя автора и адрес электронной почты, которые выводятся командой **git log**. По умолчанию Git использует системные настройки для заполнения этих полей. Чтобы установить их явно, введите

```
  $ git config --global user.name "John
Doe"
  $ git config --global user.email
johndoe@example.com
```

Чтобы установить эти параметры только для текущего хранилища, опустите флаг --global.

Git через SSH, HTTP

Предположим, у вас есть SSH доступ к веб-серверу, но Git не установлен. Git может связываться через HTTP, хотя это и менее эффективно, чем его собственный протокол.

Скачайте, скомпилируйте, установите Git в вашем аккаунте; создайте хранилище в каталоге, доступном через web:

```
 $ GIT_DIR=proj.git git init
 $ cd proj.git
 $ git --bare update-server-info
 $ cp hooks/post-update.sample
hooks/post-update
```

Для старых версий Git команда копирования не сработает, и вы должны будете запустить

```
$ chmod a+x hooks/post-update
```

Теперь вы можете публиковать свои последние правки через SSH с любого клона:

```
 $ git push
  .    :/  / /proj.git master
```

и кто угодно сможет взять ваш проект с помощью

```
$ git clone http://  .    /proj.git
```

Git через что угодно

Хотите синхронизировать хранилища без серверов или вообще без сетевого подключения? Вынуждены импровизировать на ходу в непредвиденной ситуации? Мы видели, как **git fast-export** и **git fast-import** могут преобразовать хранилища в один файл и обратно. Посредством обмена такими файлами мы можем переносить хранилища git любыми доступными средствами, но есть более эффективный инструмент: **git bundle**.

Отправитель создает пакет (bundle):

```
$ git bundle create    -    HEAD
```

Затем передает «пакет»,   -   , другой команде любыми средствами, как то: электронная почта, флешка, **xxd** печать и последующее распознавание текста, надиктовка битов по телефону, дымовые сигналы и так далее. Получатель восстанавливает коммиты из пакета, введя

```
$ git pull    -
```

Получатель может сделать это даже в пустом хранилище. Несмотря на свой небольшой размер,   -   содержит всё исходное хранилище Git.

В больших проектах для устранения излишков объема пакетируют только изменения, которых нет в других хранилищах. К примеру, пусть коммит «1b6d…» – последний общий для обеих групп:

```
$ git bundle create    -    HEAD ^1b6d
```

Если это делается часто, можно легко забыть, какой коммит был отправлен последним. Справка предлагает для решения этой проблемы использовать теги. А именно, после передачи пакета введите

```
$ git tag -f      -    HEAD
```

и создавайте обновленные пакеты с помощью

```
$ git bundle create    -    HEAD ^     -
```

Патчи: общее применение

Патчи это тексты изменений, вполне понятные как человеку, так и компьютеру. Это делает их очень привлекательным форматом обмена. Патч можно послать разработчикам по электронной почте, независимо от того, какую систему управления версиями они используют. Вашим корреспондентам достаточно возможности читать электронную почту, чтобы увидеть ваши изменения. Точно так же, с Вашей стороны требуется лишь адрес электронной почты: нет нужды в настройке онлайн хранилища Git.

Вспомним из первой главы:

```
$ git diff 1b6d
```

выводит патч, который может быть вставлен в письмо для обсуждения. В Git хранилище введите

```
$ git apply <   .patch
```

для применения патча.

В более формальных случаях , когда нужно сохранить имя автора и подписи, создавайте соответствующие патчи с заданной точки, набрав

```
$ git format-patch 1b6d
```

Полученные файлы могут быть отправлены с помощью **git-send-email** или вручную. Вы также можете указать диапазон коммитов:

```
$ git format-patch 1b6d..HEAD^^
```

На принимающей стороне сохраните письмо в файл и введите:

```
$ git am < email.txt
```

Это применит входящие исправления и создаст коммит, включающий имя автора и другую информацию.

С web-интерфейсом к электронной почте вам, возможно, потребуется нажать кнопку, чтобы посмотреть электронную почту в своем первоначальном виде перед сохранением патча в файл.

Для клиентов электронной почты, использующих mbox, есть небольшие отличия; но если вы используете один из них, то вы, по всей видимости, можете легко разобраться в этом без чтения описаний!

Приносим извинения, мы переехали

После клонирования хранилища команды **git push** или **git pull** автоматически отправляют и получают его по первоначальному адресу. Каким образом Git это делает? Секрет кроется в настройках, заданных при создании клона. Давайте взглянем:

```
$ git config --list
```

Опция remote.origin.url задает исходный адрес; origin — имя первоначального хранилища. Как и имя ветки master, это соглашение. Мы можем изменить или удалить это сокращённое имя, но как правило, нет причин для этого.

Если оригинальное хранилище переехало, можно обновить его адрес командой

```
$ git config remote.origin.url git://   .url/proj.git
```

Опция branch.master.merge задает удаленную ветку по умолчанию для **git pull**. В ходе первоначального клонирования она устанавливается на текущую ветку исходного хранилища, так что даже если HEAD исходного хранилища впоследствии переместится на другую ветку, pull будет верно следовать изначальной ветке.

Этот параметр обращается только к хранилищу, которое мы изначально клонировали и которое записано в параметре branch.master.remote. При выполнении pull из других хранилищ мы должны указать нужную ветку:

```
$ git pull git://    .com/other.git master
```

Это объясняет, почему некоторых из наших предыдущих примеров push и pull не имели аргументов.

Удаленные ветки

При клонировании хранилища вы также клонируете все его ветки. Вы можете не заметить этого, потому что Git скрывает их: вы должны запросить их явно. Это предотвращает противоречие между ветками в удаленном хранилище и вашими ветками, а также делает Git проще для начинающих.

Список удаленных веток можно посмотреть командой

```
$ git branch -r
```

Вы должны увидеть что-то вроде

```
origin/HEAD
origin/master
origin/experimental
```

Эти имена отвечают веткам и «голове» в удаленном хранилище; их можно использовать в обычных командах Git. Например, вы сделали много коммитов, и хотели бы сравнить текущее состояние с последней загруженной версией. Вы можете искать в журналах нужный SHA1 хеш, но гораздо легче набрать

```
$ git diff origin/HEAD
```

Также можно увидеть, для чего была создана ветка experimental:

```
$ git log origin/experimental
```

Несколько удаленных хранилищ

Предположим, что над нашим проектом работают еще два разработчика, и мы хотим следить за обоими. Мы можем наблюдать более чем за одним хранилищем одновременно, вот так:

```
$ git remote add other git://   .com/   _     .git
$ git pull other    _
```

Сейчас мы сделали слияние с веткой из второго хранилища. Теперь у нас есть легкий доступ ко всем веткам во всех хранилищах:

```
 $ git diff origin/experimental^
other/   _   ~5
```

Но что если мы просто хотим сравнить их изменения, не затрагивая свою работу? Иными словами, мы хотим изучить чужие ветки, не давая их изменениям вторгаться в наш рабочий каталог. Тогда вместо pull наберите

```
 $ git fetch #         origin,
     .
 $ git fetch other #
           .
```

Так мы лишь переносим их историю. Хотя рабочий каталог остается нетронутыми, мы можем обратиться к любой ветке в любом хранилище команды, работающей с Git, так как теперь у нас есть локальная копия.

Держим в уме, что pull это просто **fetch**, а затем **merge**. Обычно мы используем **pull**, потому что мы хотим влить к себе последний коммит после получения чужой ветки. Описанная ситуация — примечательное исключение.

О том, как отключить удаленные хранилища, игнорировать отдельные ветки и многом другом смотрите в **git help remote**.

Мои Настройки

Я предпочитаю, чтобы люди, присоединяющиеся к моим проектам, создавали хранилища, из которых я смогу получать изменения с помощью pull. Некоторые хостинги Git позволяют создавать собственные форки проекта в одно касание.

После получения дерева из удаленного хранилища я запускаю команды Git для навигации и изучения изменений, в идеале хорошо организованных и описанных. Я делаю слияние со своими изменения и возможно вношу дальнейшие правки. Когда я доволен результатом, я заливаю изменения в главное хранилище.

Хотя со мной мало сотрудничают, я верю, что этот подход хорошо масштабируется. Смотрите эту запись в блоге Линуса Торвальдса.

Оставаться в мире Git несколько удобнее, чем использовать файлы патчей, так как это избавляет меня от преобразования их в коммиты Git. Кроме того, Git управляет деталями вроде сохранения имени автора и адреса электронной почты, а также даты и времени, и просит авторов описывать свои изменения.

Гроссмейстерство Git

Теперь вы уже должны уметь ориентироваться в страницах **git help** и понимать почти всё. Однако точный выбор команды, необходимой для решения конкретной проблемы, может быть утомительным. Возможно, я сберегу вам немного времени: ниже приведены рецепты, пригодившиеся мне в прошлом.

Релизы исходников

В моих проектах Git управляет в точности теми файлами, которые я собираюсь архивировать и пускать в релиз. Чтобы создать тарбол с исходниками, я выполняю:

```
$ git archive --format=tar --prefix=proj-1.2.3/ HEAD
```

Коммит изменений

В некоторых проектах может быть трудоемко оповещать Git о каждом добавлении, удалении и переименовании файла. Вместо этого вы можете выполнить команды

```
$ git add .
$ git add -u
```

Git просмотрит файлы в текущем каталоге и сам позаботится о деталях. Вместо второй команды add, выполните **git commit -a**, если вы собираетесь сразу

сделать коммит. Смотрите **git help ignore**, чтобы узнать как указать файлы, которые должны игнорироваться.

Вы можете выполнить все это одним махом:

```
$ git ls-files -d -m -o -z | xargs -0 git update-index --add --remove
```

Опции **-z** и **-0** предотвращают неверную обработку файловых имен, содержащих специальные символы. Поскольку эта команда добавляет игнорируемые файлы, вы возможно захотите использовать опции -x или -X.

Мой коммит слишком велик

Вы пренебрегали коммитами слишком долго? Яростно писали код и вспомнили об управлении исходниками только сейчас? Внесли ряд несвязанных изменений, потому что это ваш стиль?

Нет поводов для беспокойства. Выполните

```
$ git add -p
```

Для каждой сделанной вами правки Git покажет измененный участок кода и спросит, должно ли это изменение попасть в следующий коммит. Отвечайте «y» (да) или «n» (нет). У вас есть и другие варианты, например отложить выбор; введите «?» чтобы узнать больше.

Когда закончите, выполните

```
$ git commit
```

для внесения именно тех правок, что вы выбрали («буферизованных» изменений). Убедитесь, что вы не указали опцию **-a**, иначе Git закоммитит все правки.

Что делать, если вы изменили множество файлов во многих местах? Проверка каждого отдельного изменения становится удручающей рутиной. В этом случае используйте **git add -i**. Ее интерфейс не так прост, но более гибок. В несколько нажатий кнопок можно добавить или убрать из буфера несколько файлов одновременно, либо просмотреть и выбрать изменения лишь в отдельных файлах. Как вариант, запустите **git commit --interactive**, которая автоматически сделает коммит когда вы закончите.

Индекс — буферная зона Git

До сих пор мы избегали знаменитого «индекса» Git, но теперь мы должны рассмотреть его, для пояснения вышесказанного. Индекс это временный буфер. Git редко перемещает данные непосредственно между вашим проектом и его историей. Вместо этого Git сначала записывает данные в индекс, а уж затем копирует их из индекса по месту назначения.

Например, **commit -a** на самом деле двухэтапный процесс. Сначала слепок текущего состояния каждого из отслеживаемых файлов помещается в индекс. Затем слепок, находящийся в индексе, записывается в историю. Коммит без опции **-a** выполняет только второй шаг, и имеет смысл только после выполнения команд, изменяющих индекс, таких как **git add**.

Обычно мы можем не обращать внимания на индекс и делать вид, что взаимодействуем напрямую с историей. Но в данном случае мы хотим более тонкого контроля, поэтому управляем индексом. Мы помещаем слепок некоторых (но не всех) наших изменений в индекс, после чего окончательно записываем этот аккуратно сформированный слепок.

Не теряй «головы»

Тег HEAD (англ. «голова», прим. пер.) — как курсор, который обычно указывает на последний коммит, продвигаясь с каждым новым коммитом. Некоторые команды Git позволяют перемещать этот курсор. Например,

```
$ git reset HEAD~3
```

переместит HEAD на три коммита назад. Теперь все команды Git будут работать так, как будто вы не делали последних трех коммитов, хотя файлы останутся в текущем состоянии. В справке описано несколько способов использования этого приема.

Но как вернуться назад в будущее? Ведь предыдущие коммиты о нем ничего не знают.

Если у вас есть SHA1 изначальной «головы», то:

```
$ git reset 1b6d
```

Но допустим, вы его не записывали. Не беспокойтесь: для комнад такого рода Git сохраняет оригинальную «голову» как тег под названием ORIG_HEAD, и вы можете вернуться надежно и безопасно:

```
$ git reset ORIG_HEAD
```

Охота за «головами»

Предположим ORIG_HEAD недостаточно. К примеру, вы только что осознали, что допустили громадную ошибку, и вам нужно вернуться к древнему коммиту в давно забытой ветке.

По умолчанию Git хранит коммиты не меньше двух недель, даже если вы приказали уничтожить содержащую их ветку. Проблема в нахождении соответствующего хеша. Вы можете просмотреть все значения хешей в

.git/objects и методом проб и ошибок найти нужный. Но есть путь значительно легче.

Git записывает каждый подсчитанный им хеш коммита в .git/logs. В подкатлоге refs содержится полная история активности на всех ветках, а файл HEAD содержит каждое значение хеша, которое когда-либо принимал HEAD. Последнее можно использовать чтобы найти хеши коммитов на случайно обрубленных ветках.

Команда reflog предоставляет удобный интерфейс работы с этими журналами. Используйте

```
$ git reflog
```

Вместо копирования хешей из reflog, попробуйте

```
$ git checkout "@{10 minutes ago}" # 10        ,    .   .
```

Или сделайте чекаут пятого с конца из посещенных коммитов с помощью

```
$ git checkout "@{5}"
```

Смотрите раздел «Specifying Revisions» в **git help rev-parse** для дополнительной информации.

Вы можете захотеть удлинить отсрочку для коммитов, обреченных на удаление. Например,

```
$ git config gc.pruneexpire "30 days"
```

означает, что удаляемые коммиты будут окончательно исчезать только по прошествии 30 дней и после запуска **git gc**.

Также вы можете захотеть отключить автоматический вызов **git gc**:

```
$ git config gc.auto 0
```

В этом случае коммиты будут удаляться только когда вы будете запускать **git gc** вручную.

Git как основа

Дизайн Git, в истинном духе UNIX, позволяет легко использовать его как низкоуровневый компонент других программ: графических и веб-интерфейсов; альтернативных интерфейсов командной строки; инструментов управления патчами; средств импорта или конвертации, и так далее. Многие команды Git на самом деле — скрипты, стоящие на плечах гигантов. Небольшой доработкой вы можете переделать Git на свой вкус.

Простейший трюк — использование алиасов Git для сокращения часто используемых команд:

```
$ git config --global alias.co checkout
$ git config --global --get-regexp alias       #
alias.co checkout
$ git co foo #  - ,     «git checkout foo»
```

Другой пример: можно выводить текущую ветку в приглашении командной строки или заголовке окна терминала. Запуск

```
$ git symbolic-ref HEAD
```

выводит название текущей ветки. На практике вы скорее всего захотите убрать «refs/heads/» и сообщения об ошибках:

```
$ git symbolic-ref HEAD 2> /dev/null | cut -b 12-
```

Подкаталог contrib это целая сокровищница инструментов, построенных на Git. Со временем некоторые из них могут становиться официальными командами. В Debian и Ubuntu этот каталог находится в /usr/share/doc/git-core/contrib.

Один популярный инструмент из этого каталога — workdir/git-new-workdir. Этот скрипт создает с помощью символических ссылок новый рабочий каталог, имеющий общую историю с оригинальным хранилищем:

```
$ git-new-workdir        /         /
```

Новый каталог и файлы в нем можно воспринимать как клон, с той разницей, что два дерева автоматически остаются синхронизированными ввиду общей истории. Нет необходимости в merge, push и pull.

Рискованные трюки

Нынешний Git делает случайное уничтожение данных очень сложным. Но если вы знаете, что делаете, вы можете обойти защиту для распространенных команд.

Checkout: Наличие незакоммиченных изменений прерывает выполнение checkout. Чтобы перейти к нужному коммиту, даже уничтожив свои изменения, используйте «принуждающий» (force, прим. пер.) флаг **-f**:

```
$ git checkout -f HEAD^
```

С другой стороны, если вы укажете checkout конкретные пути, проверки на безопасность не будет: указанные файлы молча перезапишутся. Будьте осторожны при таком использовании checkout.

Reset: сброс также прерывается при наличии незакоммиченных изменений. Чтобы заставить его сработать, запустите

```
$ git reset --hard 1b6d
```

Branch: Удаление ветки прервётся, если оно привело бы к потере изменений. Для принудительного удаления введите

```
$ git branch -D     _    #     -d
```

Аналогично, попытка перезаписи ветки путем перемещения будет прервана, если может привести к потере данных. Для принудительного перемещений ветки введите

```
$ git branch -M          #     -m
```

В отличии от checkout и reset, эти две команды дают отсрочку в удалении данных. Изменения остаются в каталоге .git и могут быть возвращены восстановлением нужного хеша из .git/logs (смотрите выше раздел «Охота за „головами“»). По умолчанию они будут храниться по крайней мере две недели.

Clean: Некоторые команды могут не сработать из опасений повредить неотслеживаемые файлы. Если вы уверены, что все неотслеживаемые файлы и каталоги не нужны, то безжалостно удаляйте их командой

```
$ git clean -f -d
```

В следующий раз эта досадная команда сработает!

Предотвращаем плохие коммиты

Глупые ошибки загрязняют мои хранилища. Самое ужасное это проблема недостающих файлов, вызванная забытым **git add**.

Примеры менее серьезных проступков: завершающие пробелы и неразрешённые конфликты слияния. Несмотря на безвредность, я не хотел бы, чтобы это появлялось в публичных записях.

Если бы я только поставил защиту от дурака, используя *хук*, который бы предупреждал меня об этих проблемах:

```
$ cd .git/hooks
$ cp pre-commit.sample pre-commit #          Git: chmod +x pre-commit
```

Теперь Git отменит коммит, если обнаружит лишние пробелы или неразрешенные конфликты.

Для этого руководства я в конце концов добавил следующее в начало хука **pre-commit**, чтобы защититься от своей рассеянности:

```
if git ls-files -o | grep \.txt$; then echo ПРЕРВАНО! Неотслеживаемые .txt
файлы. exit 1 fi
```

Хуки поддерживаются несколькими различными операциями Git, смотрите **git help hooks**. Мы использовали пример хука **post-update** раньше, при обсуждении использования Git через http. Он запускался при каждом перемещении «головы». Пример скрипта **post-update** обновляет файлы, которые

нужны Git для связи через не считающиеся с ним средства сообщения, такие как HTTP.

Раскрываем тайны

Мы заглянем под капот и объясним, как Git творит свои чудеса. Я опущу излишние детали. За более детальными описаниями обратитесь к руководству пользователя.

Невидимость

Как Git может быть таким ненавязчивым? За исключением периодических коммитов и слияний, вы можете работать так, как будто и не подозреваете о каком-то управлении версиями. Так происходит до того момента, когда Git вам понадобится, и тогда вы с радостью увидите, что он наблюдал за вами все это время.

Другие системы управления версиями вынуждают вас постоянно бороться с загородками и бюрократией. Файлы могут быть доступны только для чтения, пока вы явно не укажете центральному серверу, какие файлы вы намереваетесь редактировать. С увеличением количества пользователей большинство базовых команд начинают выполняться всё медленнее. Неполадки с сетью или с центральным сервером полностью останавливают работу.

В противоположность этому, Git просто хранит историю проекта в подкаталоге .git вашего рабочего каталога. Это ваша личная копия истории, поэтому вы можете оставаться вне сети, пока не захотите взаимодействовать с остальными. У вас есть полный контроль над судьбой ваших файлов, поскольку Git в любое время может легко восстановить сохраненное состояние из .git.

Целостность

Большинство людей ассоциируют криптографию с содержанием информации в секрете, но другой столь же важной задачей является содержание ее в сохранности. Правильное использование криптографических хеш-функций может предотвратить случайное или злонамеренное повреждение данных.

SHA1 хеш можно рассматривать как уникальный 160-битный идентификатор для каждой строки байт, с которой вы сталкиваетесь в вашей жизни. Даже больше того: для каждой строки байтов, которую любой человек когда-либо будет использовать в течение многих жизней.

Так как SHA1 хеш сам является последовательностью байтов, мы можем получить хеш строки байтов, содержащей другие хеши. Это простое наблюдение на

удивление полезно: ищите «hash chains» (цепочки хешей). Позднее мы увидим, как Git использует их для эффективного обеспечения целостности данных.

Говоря кратко, Git хранит ваши данные в подкаталоге ”.git/objects”, где вместо нормальных имен файлов вы найдете только идентификаторы. Благодаря использованию идентификаторов в качестве имен файлов, а также некоторым хитростям с файлами блокировок и временны́ми метками, Git преобразует любую скромную файловую систему в эффективную и надежную базу данных.

Интеллект

Как Git узнаёт, что вы переименовали файл, даже если вы никогда не упоминали об этом явно? Конечно, вы можете запустить **git mv**; но это то же самое, что **git rm**, а затем **git add**.

Git эвристически находит файлы, которые были переименованы или скопированы между соседними версиями. На деле он может обнаружить, что участки кода были перемещены или скопированы между файлами! Хотя Git не может охватить все случаи, он всё же делает достойную работу, и эта функция постоянно улучшается. Если она не сработала, попробуйте опции, включающие более ресурсоемкое обнаружение копирования и подумайте об обновлении.

Индексация

Для каждого отслеживаемого файла, Git записывает такую информацию, как размер, время создания и время последнего изменения, в файле, известном как «индекс». Чтобы определить, был ли файл изменен, Git сравнивает его текущие характеристики с сохраненными в индексе. Если они совпадают, то Git не станет перечитывать файл заново.

Поскольку считывание этой информации значительно быстрее, чем чтение всего файла, то если вы редактировали лишь несколько файлов, Git может обновить свой индекс почти мгновенно.

Мы отмечали ранее, что индекс это буферная зона. Почему набор свойств файлов выступает таким буфером? Потому что команда add помещает файлы в базу данных Git и в соответствии с этим обновляет эти свойства; тогда как команда commit без опций создает коммит, основанный только на этих свойствах и файлах, которые уже в базе данных.

Происхождение Git

Это сообщение в почтовой рассылке ядра Linux описывает последовательность событий, которые привели к появлению Git. Весь этот тред — привлекательный археологический раскоп для историков Git.

База данных объектов

Каждая версия ваших данных хранится в «базе данных объектов», живущей в подкаталоге .git/objects. Другие «жители» .git/ содержат вторичные данные: индекс, имена веток, теги, параметры настройки, журналы, нынешнее расположение «головного» коммита и так далее. База объектов проста и элегантна, и в ней источник силы Git.

Каждый файл внутри .git/objects это «объект». Нас интересуют три типа объектов: объекты «блобов», объекты деревьев и объекты коммитов.

Блобы

Для начала один фокус. Выберите имя файла — любое имя файла. В пустом каталоге:

```
$ echo sweet >   _ _
$ git init
$ git add .
$ find .git/objects -type f
```

Вы увидите .git/objects/aa/823728ea7d592acc69b36875a482cdf3fd5c8d.

Откуда я знаю это, не зная имени файла? Это потому, что SHA1 хеш строки

«blob» SP «6» NUL «sweet» LF

равен aa823728ea7d592acc69b36875a482cdf3fd5c8d, где SP это пробел, NUL — нулевой байт и LF — перевод строки. Вы можете проверить это, набрав

```
$ printf "blob 6\000sweet\n" | sha1sum
```

Git использует «адресацию по содержимому»: файлы хранятся в соответствии не с именами, а с хешами содержимого, — в файле, который мы называем «блоб-объектом». Хеш можно понимать как уникальный идентификатор содержимого файла, что означает обращение к файлам по их содержимому. Начальный «blob 6» — лишь заголовок, состоящий из типа объекта и его длины в байтах и упрощающий внутренний учет.

Таким образом, я могу легко предсказать, что вы увидите. Имя файла не имеет значения: для создания блоб-объекта используется только его содержимое.

Вам может быть интересно, что происходит с одинаковыми файлами. Попробуйте добавить копии своего файла с какими угодно именами. Содержание .git/objects останется тем же независимо от того, сколько копий вы добавите. Git хранит данные лишь единожды.

Кстати, файлы в каталоге .git/objects сжимаются с помощью zlib поэтому вы не сможете просмотреть их напрямую. Пропустите их через фильтр zpipe -d, или введите

```
$ git cat-file -p aa823728ea7d592acc69b36875a482cdf3fd5c8d
```

что выведет указанный объект в читаемом виде.

Деревья

Но где же имена файлов? Они должны храниться на каком-то уровне. Git обращается за именами во время коммита:

```
$ git commit  #         -
$ find .git/objects -type f
```

Теперь вы должны увидеть три объекта. На этот раз я не могу сказать вам, что из себя представляют два новых файла, так как это частично зависит от выбранного вами имени файла. Далее будем предполагать, что вы назвали его «rose». Если это не так, то вы можете переписать историю, чтобы она выглядела как будто вы это сделали:

```
$ git filter-branch --tree-filter 'mv    _  _    rose'
$ find .git/objects -type f
```

Теперь вы должны увидеть файл .git/objects/05/b217bb859794d08bb9e4f7f04cbda4b207fbe9, так как это SHA1 хеш его содержимого:

«tree» SP «32» NUL «100644 rose» NUL 0xaa823728ea7d592acc69b36875a482cdf3fd5c8d

Проверьте, что этот файл действительно содержит указанную строку, набрав

```
$ echo 05b217bb859794d08bb9e4f7f04cbda4b207fbe9 | git cat-file --batch
```

С zpipe легко проверить хеш:

```
$ zpipe -d < .git/objects/05/b217bb859794d08bb9e4f7f04cbda4b207fbe9 | sha1sum
```

Проверка хеша с помощью cat-file сложнее, поскольку ее вывод содержит не только «сырой» распакованный файл объекта.

Этот файл — объект «дерево» (tree, прим. пер.): список цепочек, состоящих из типа, имени файла и его хеша. В нашем примере: тип файла — 100644, что означает, что «rose» это обычный файл; а хеш — блоб-объект, в котором находится содержимое «rose». Другие возможные типы файлов: исполняемые файлы, символические ссылки или каталоги. В последнем случае, хеш указывает на объект «дерево».

Если вы запускали filter-branch, у вас есть старые объекты которые вам больше не нужны. Хотя по окончании срока хранения они будут выброшены автоматически, мы удалим их сейчас, чтобы было легче следить за нашим игрушечным примером:

```
$ rm -r .git/refs/original
$ git reflog expire --expire=now --all
$ git prune
```

Для реальных проектов обычно лучше избегать таких команд, поскольку вы уничтожаете резервные копии. Если вы хотите иметь чистое хранилище, то обычно лучше сделать свежий клон. Кроме того, будьте осторожны при непосредственном вмешательстве в каталог .git: что если другая команда Git работает в это же время, или внезапно произойдет отключение питания? Вообще говоря, ссылки нужно удалять с помощью **git update-ref -d**, хотя обычно ручное удаление refs/original безопасно.

Коммиты

Мы рассмотрели два из трех объектов. Третий объект — «коммит» (commit). Его содержимое зависит от описания коммита, как и от даты и времени его создания. Для соответствия тому, что мы имеем, мы должны немного «подкрутить» Git:

```
$ git commit --amend -m Shakespeare  #               .
$ git filter-branch --env-filter 'export
    GIT_AUTHOR_DATE="Fri 13 Feb 2009 15:31:30 -0800"
    GIT_AUTHOR_NAME="Alice"
    GIT_AUTHOR_EMAIL="alice@example.com"
    GIT_COMMITTER_DATE="Fri, 13 Feb 2009 15:31:30 -0800"
    GIT_COMMITTER_NAME="Bob"
   GIT_COMMITTER_EMAIL="bob@example.com"'  #                    .
$ find .git/objects -type f
```

Теперь вы должны увидеть .git/objects/49/993fe130c4b3bf24857a15d7969c396b7bc187 который является SHA1 хешем его содержимого:

```
«commit 158» NUL
«tree 05b217bb859794d08bb9e4f7f04cbda4b207fbe9» LF
«author Alice <alice@example.com> 1234567890 -0800» LF
«committer Bob <bob@example.com> 1234567890 -0800» LF
LF
«Shakespeare» LF
```

Как и раньше, вы сами можете запустить zpipe или cat-file, чтобы увидить это.

Это первый коммит, поэтому здесь нет родительских коммитов, но последующие коммиты всегда будет содержать хотя бы одну строку, идентифицирующую родительский коммит.

Неотличимо от волшебства

Секреты Git выглядят слишком простыми. Похоже, что вы могли бы объединить несколько shell-скриптов и добавить немного кода на C, чтобы сделать всё это в считанные часы: смесь базовых операций с файлами и SHA1-хеширования,

приправленная блокировочными файлами и fsync для надежности. По сути, это точное описание ранних версий Git. Тем не менее, помимо гениальных трюков с упаковкой для экономии места и с индексацией для экономии времени, мы теперь знаем, как ловко Git преображает файловую систему в базу данных, идеально подходящую для управления версиями.

Например, если какой-либо файл в базе данных объектов поврежден из-за ошибки диска, то его хеш теперь не совпадет, что привлечет наше внимание к проблеме. С помощью хеширования хешей других объектов, мы поддерживаем целостность на всех уровнях. Коммиты атомарны, так что в них никогда нельзя записать лишь часть изменений: мы можем вычислить хеш коммита и сохранить его в базу данных только сохранив все соответствующие деревья, блобы и родительские коммиты. База данных объектов нечувствительна к непредвиденным прерываниям работы, таких как перебои с питанием.

Мы наносим поражение даже самым хитрым противникам. Предположим, кто-то пытается тайно изменить содержимое файла в древней версии проекта. Чтобы база объектов выглядела неповрежденной, он также должен изменить хеш соответствующего блоб-объекта, поскольку это теперь другая последовательность байтов. Это означает, что нужно поменять хеши всех объектов деревьев, ссылающихся на этот файл; что в свою очередь изменит хеши всех объектов коммитов с участием таких деревьев; а также и хеши всех потомков этих коммитов. Вследствие этого хеш официальной головной ревизии будет отличаться от аналогичного хеша в этом испорченном хранилище. По цепочке несовпадающих хешей мы можем точно вычислить искаженный файл, как и коммит, где он изначально был поврежден.

Одним словом, невозможно подделать хранилище Git, оставив невредимыми двадцать байт, отвечающие последнему коммиту.

Как насчет известных характерных особенностей Git? Ветвление? Слияние? Теги? Очевидные подробности. Текущая «голова» хранится в файле `.git/HEAD`, содержащем хеш объекта коммита. Хеш обновляется во время коммита, а также при выполнении многих других команд. С ветками всё аналогично: это файлы в `.git/refs/heads`. То же и тегами: они живут в `.git/refs/tags`, но их обновляет другой набор команд.

Недостатки Git

Есть некоторые проблемы Git, которые я спрятал под сукно. Некоторые из них можно легко решить с помощью скриптов и хуков, некоторые требуют реорганизации или пересмотра проекта, а несколько оставшихся неприятностей придется потерпеть. А еще лучше — взяться за них и решить!

Слабости SHA1

Со временем криптографы обнаруживают всё больше и больше слабостей в SHA1. Уже сейчас обнаружение коллизий хешей осуществимо для хорошо финансируемой организации. Спустя годы, возможно, даже типичный ПК будет иметь достаточную вычислительную мощность, чтобы незаметно испортить хранилище Git.

Надеюсь, Git перейдет на лучшую хеш-функцию прежде чем дальнейшие исследования уничтожат SHA1.

Microsoft Windows

Git на Microsoft Windows может быть громоздким:

- Cygwin, Linux-подобная среда для Windows, содержащая порт Git на Windows.
- Git для Windows, вариант, требующий минимальной рантайм поддержки, хотя некоторые комманды нуждаются в доработке.

Несвязанные файлы

Если ваш проект очень велик и содержит много несвязанных файлов, которые постоянно изменяются, Git может оказаться в невыгодном положении по сравнению с другими системами, поскольку отдельные файлы не отслеживаются. Git отслеживает изменения всего проекта, что обычно бывает выгодным.

Решение — разбить проект на части, каждая из которых состоит из взаимосвязанных файлов. Используйте **git submodule** если вы все же хотите держать все в одном хранилище.

Кто и что редактировал ?

Некоторые системы управления версиями вынуждают вас явным образом пометить файл перед редактированием. Хотя такой подход особенно раздражает, когда подразумевает работу с центральным сервером, однако он имеет два преимущества:

1. Diff’ы быстры, так как нужно проверить только отмеченные файлы.
2. Можно обнаружить, кто еще работает с этим файлом, спросив центральный сервер, кто отметил его для редактирования.

С помощью соответствующих скриптов, вы можете добиться того же с Git. Это требует сотрудничества со стороны другого программиста, который должен запускать определенный скрипт при редактировании файла.

История файла

Поскольку Git записывает изменения всего проекта, воссоздание истории единичного файла требует больше работы, чем в системах управления версиями, следящими за отдельными файлами.

Потери как правило незначительны, и это неплохая цена за то, что другие операции невероятно эффективны. Например, git checkout быстрее, чем cp -a, а дельта всего проекта сжимается лучше, чем коллекция по-файловых дельт.

Начальное Клонирование

Создание клона хранилища дороже обычного чекаута в других системах управления версиями при длинной истории.

Первоначальная цена окупается в долгосрочной перспективе, так как большинство последующих операций будут быстрыми и автономными. Однако в некоторых ситуациях может быть предпочтительным создание мелких клонов с опцией --depth. Это намного быстрее, но у полученного клона будет урезанная функциональность.

Изменчивые Проекты

Git был написан, чтобы быть быстрым при относительно небольших изменениях. Люди вносят незначительные правки от версии к версии. Однострочное исправление ошибки здесь, новая функция там, исправленные комментарии и тому подобное. Но если ваши файлы радикально различаются в соседних ревизиях, то с каждым коммитом ваша история неизбежно увеличится на размер всего проекта.

Никакая система управления версиями ничего не может с этим сделать, но пользователи Git страдают больше, поскольку обычно истории клонируются.

Причины, по которым эти изменения столь велики, нужно изучить. Возможно, надо изменить форматы файлов. Небольшие правки должны приводить к небольшим изменений не более чем в нескольких файлах.

Возможно, вам была нужна база данных или система резервного/архивного копирования, а не система управления версиями. Например, управление версиями может быть плохо приспособлено для обращения с фотографиями периодически получаемыми с веб-камеры.

Если файлы действительно должны постоянно изменяться и при этом версироваться, может иметь смысл использовать Git централизованным образом. Можно создавать мелкие клоны, с небольшой историей или без истории вообще. Конечно, многие инструменты Git будут недоступны, и исправления придется представлять в виде патчей. Возможно, это и хорошо,

так как неясно, зачем кому-либо понадобится история крайне нестабильных файлов.

Другой пример — это проект, зависимый от прошивки, принимающей форму огромного двоичного файла. Ее история неинтересна пользователям, а обновления плохо сжимаются, потому ревизии прошивки будут неоправдано раздувать размер хранилища.

В этом случае исходный код стоит держать в хранилище Git, а бинарные файлы — отдельно. Для упрощения жизни можно распространять скрипт, использующий Git для клонирования кода и rsync или мелкий клон Git для прошивки.

Глобальный счетчик

Некоторые централизованные системы управления версиями содержат натуральное число, увеличивающееся при поступлении нового коммита. Git идентифицирует изменения по их хешам, что лучше во многих обстоятельствах.

Но некоторым людям нравятся эти целые числа повсюду. К счастью, легко написать такой скрипт, чтобы при каждом обновлении центральное хранилище Git увеличивало целое число, возможно, в теге, и связывало его с хешем последнего коммита.

Каждый клон может поддерживать такой счетчик, но это, видимо, будет бесполезным, поскольку только центральное хранилище и его счетчик имеет значение для всех.

Пустые подкаталоги

Пустые подкаталоги не могут отслеживаться. Создавайте подставные файлы, чтобы обойти эту проблему.

В этом виноват не дизайн Git, а его текущая реализация. Если повезет и пользователи Git будут поднимать больше шума вокруг этой функции, возможно она будет реализована.

Первоначальный коммит

Шаблонный компьютерщик считает с 0, а не с 1. К сожалению, в отношении коммитов Git не придерживается этого соглашения. Многие команды недружелюбны до первоначального коммита. Кроме того, некоторые частные случаи требуют специальной обработки, к примеру rebase ветки с другим начальным коммитом.

Git’у было бы выгодно определить нулевой коммит: при создании хранилища HEAD был бы установлен в строку, состоящую из 20 нулевых байтов. Этот

специальный коммит представлял бы собой пустое дерево, без родителей, которое предшествует каждому хранилищу Git.

Тогда запуск **git log**, например, показывал бы пользователю, что коммиты еще не были сделаны, вместо того чтобы завершаться с фатальной ошибкой. Аналогично для других инструментов.

Каждый первоначальный коммит — неявный потомок этого нулевого коммита.

Однако здесь, к сожалению, есть некоторые проблемные случаи. Если несколько ветвей с различными начальными коммитами сливаются, то rebase результата требует значительного ручного вмешательства.

Причуды интерфейса

Для коммитов A и Б значения выражений «A..Б» и «A…Б» зависят от того, ожидает ли команда указания двух конечных точек или промежутка. Смотрите **git help diff** и **git help rev-parse**.

Перевод этого руководства

Я советую следующий способ для перевода этого руководства, чтобы мои скрипты могли быстро создавать HTML и PDF версии, а все переводы находились в одном хранилище.

Клонируйте исходные тексты, затем создайте каталог, отвечающий тегу IETF целевого языка: смотрите статью W3C по интернационализации. К примеру, английский язык это «en», а японский — «ja». Скопируйте в каталог файлы txt из каталога «en» и переведите их.

К примеру, для перевода руководства на клингонский язык, вы можете набрать:

```
$ git clone git://repo.or.cz/gitmagic.git
$ cd gitmagic
$ mkdir tlh # «tlh» -         IETF            .
$ cd tlh
$ cp ../en/intro.txt .
$ edit intro.txt  #          .
```

и так с каждым файлом.

Отредактируйте Makefile и добавьте код языка в переменную TRANSLATIONS. Теперь вы сможете просматривать вашу работу по ходу дела:

```
$ make tlh
$ firefox book.html
```

Почаще делайте коммиты, а когда ваш перевод будет готов, сообщите мне об этом. На GitHub есть веб-интерфейс, облегчающий описанные действия: сделайте форк проекта «gitmagic», залейте ваши изменения и попросите меня сделать слияние.