**Git**

**Система управления версиями (от англ. Version Control System, VCS)**

— программное обеспечение для облегчения работы с изменяющейся информацией. Система управления версиями позволяет хранить несколько версий одного и того же документа, при необходимости возвращаться к более ранним версиям, определять, кто и когда сделал то или иное изменение, и многое другое.

**Контроль** **версий** — это отслеживание изменений программного кода и управление ими.

В перечень задач, выполняемых системой контроля версий файлов, входят:

* **Сохранение и защита исходного кода.** Информация сваливается на удаленный сервер и в репозитории остаются даже файлы, удаленные с компьютера разработчика.
* **Командная работа,** возможность привлекать группу программистов, не покупая отдельно специальные инструменты для командной работы. Каждый решает свою задачу на персональном компьютере, обновляя файлы, когда это нужно.
* **Отмена внесенных изменений**. Всегда есть возможность вернуться к контрольной точке, провести ревью исходного кода и текущего, а затем обновить основную ветку.
* **Распределенная работа** над проектом. То есть, программисты могут создавать видоизмененный плагин, пока основная его версия спокойно функционирует на сайте.

**Разновидности архитектур**

**Локальная система контроля версий**

Локальная система контроля версий представляет из себя простейшую базу данных, хранящую записи обо всех изменениях в файлах. Локальная система контроля версий хорошо решает поставленную перед ней задачу, однако ее проблемой является основное свойство — локальность. Она совершенно не предназначена для коллективного использования.

**Преимущества:**

а) не требуется помощь от сторонних серверов;

б) история изменений хранится локально, без глобальной сети;

в) возможность индивидуальной работы с проектом.

**Недостатки:**

а) возможность потери данных при неисправности компьютера;

б) исключено коллективное использование, так как данные доступны только самому пользователю.

**Распределенная система контроля версий**

Данная архитектура имеет важную особенность, состоящую в том, что клиенты хранят у себя полную копию всех версий проекта, а не какую-то малую часть. При выходе из строя сервера, работа не остановится, а как только сервер восстановится на него можно будет загрузить все версии с компьютеров сотрудников. При потере копий файла проекта их легко можно восстановить с компьютера любого разработчика.

К таким системам относятся **Git**, которая на данный момент является самой быстрой распределенной системой, использующей компактное хранилище ревизий, **Mercurial**, **Bazaar** и другие.

Основными **преимуществами** распределенной VCS являются:

а) работа над проектами не зависит от работы сервера;

б) возможность работы с несколькими удаленными хранилищами;

в) гибкость общей системы и повышение надежности.

Но, если нет хорошей организации проекта, отсутствие одного центрального хранилища может стать**недостатком** данной системы.

**Централизованная система контроля версий**

Чтобы обеспечить возможность коллективной работы с проектом, была разработана централизованная система контроля версий.  Суть централизованной VCS заключается в том, что клиенты обращаются к единственному центральному серверу, который хранит все версии файлов, и получают копии файлов из него.

Примерами таких систем являются **Subversion** и **Perforce**.

**Преимущества:**

а) наличие работы в команде с другими разработчиками;

б) человеку, которые ответственен за данный проект, легче следить за работой подчиненных;

в) по сравнению с локальной администрировать централизованную систему намного легче.

**Недостатки:**

а) выход из строя или поломка сервера может остановить работу над проектом или привести к потере всех файлов;

б) при работе необходимо наличие высокоскоростной бесперебойной работы интернета.

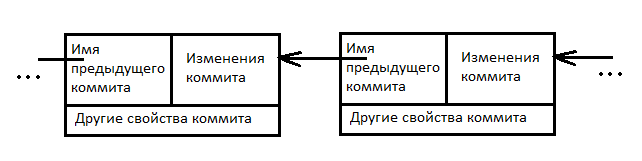
**Создание репозитория**

Все файлы проекта, находящиеся под контролем, вместе с историей их изменения и другой служебной информацией образуют ***репозиторий*** проекта.

**Изменения и их фиксация**

После создания репозитория возникает вопрос: что делать, если файл изменить?

Здесь возникает понятие ***коммит*** -  это основной объект в любой системе управления версиями. В нем содержится описание тех изменений, которые вносит пользователь в код приложения. В Git коммит состоит из нескольких так называемых объектов. Для простоты понимания можно считать, что коммиты это односвязный список, состоящий из объектов в которых содержаться измененные файлы, и ссылка на предыдущий коммит.



**Работа с GitHub**

GitHub неразрывно связан с системой контроля версий Git, которую разработчики устанавливают на персональный компьютер.

В чем разница между Git и GitHub?

С помощью Git программисты и разработчики ориентируются в коде и отслеживают изменения. Git помогает вернуть файлы в исходное состояние и видеть изменения, внесённые в определённый период. Разработчик выполняет разные команды, а все изменения синхронизируются с центральным репозиторием.

***Git*** — это система контроля версий, а ***GitHub*** — онлайн-сервис, по сути социальная сеть.

Одна из основных ***целей GitHub***— быть единым местом для проектов с исходным кодом. Предполагается, что пользователь делится чем-то полезным, а другие люди смогут участвовать в разработке.

Ещё один вариант — использовать GitHub как хранилище проектов для портфолио: легко дать на них ссылку.

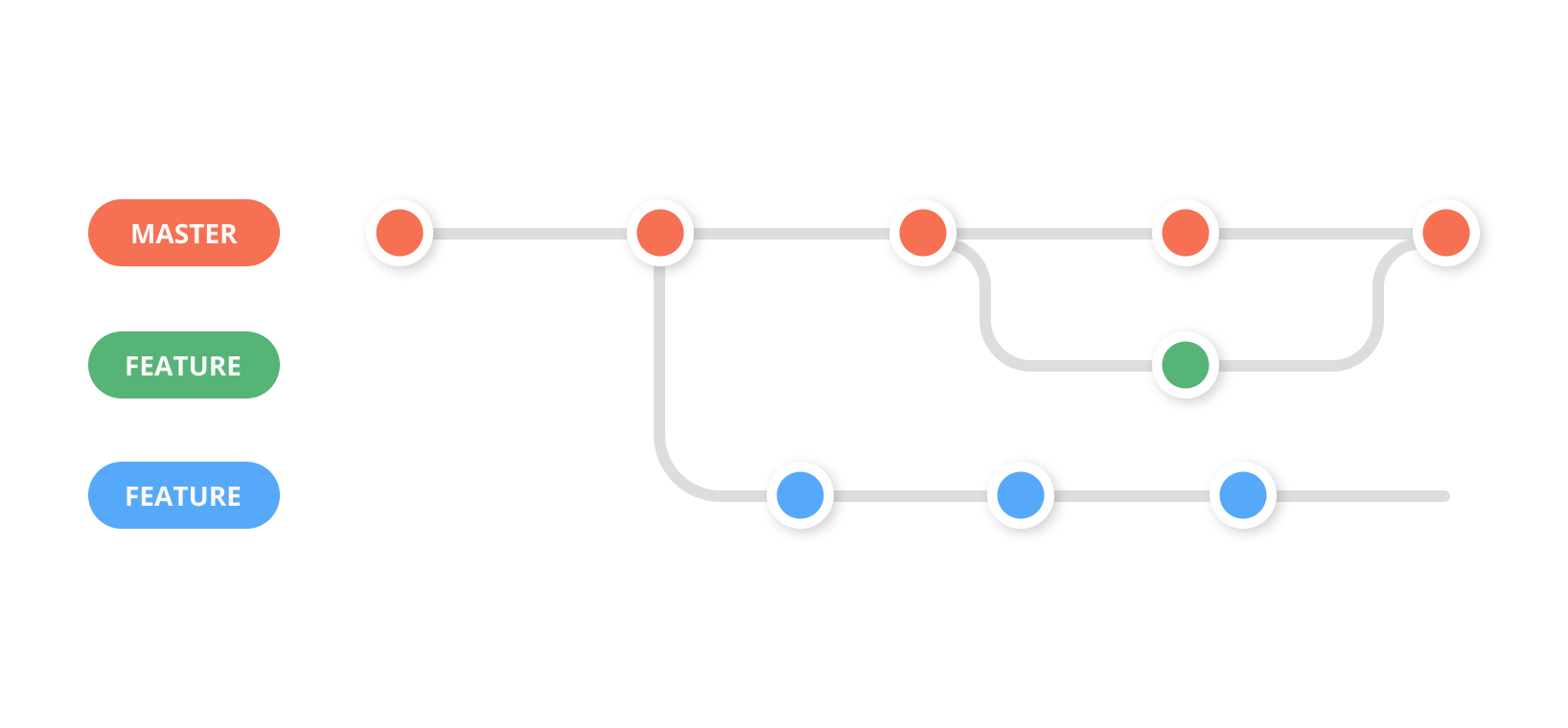
**Работа с ветками**

***Ветка в Git***это подвижный указатель на один из коммитов. Обычно ветка указывает на последний коммит в цепочке коммитов. Ветка берет свое начало от какого-то одного коммита.

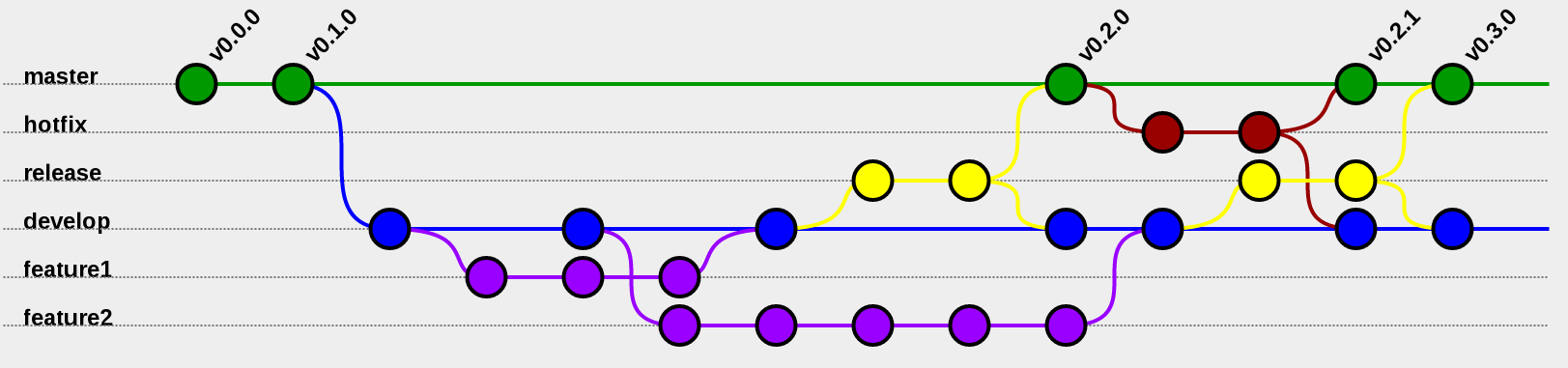
Во время создания проекта под управлением системы контроля версий создается ветка по умолчанию – master.

Ветка по сути — это ответвление от текущей версии проекта. Все изменения, которые будет вносить разработчик, будут сохраняться в ветке и когда функция будет полностью реализована и проверена, разработчик сможет слить все свои изменения с веткой master.

Наглядно это можно представить с помощью следующего рисунка:



На самом деле в большом проекте может быть огромное количество веток и они будут ветвиться и сливаться на протяжении всей работы:



**Работа в удаленном репозитории**

После клонирования репозитория можно работать с данными - это коммиты и ветки. Для работы с удаленным репозиторием применяются две команды: pull и push.

***Push (пуш)*** - это отправка данных на сервер, в удаленный репозиторий, на github.

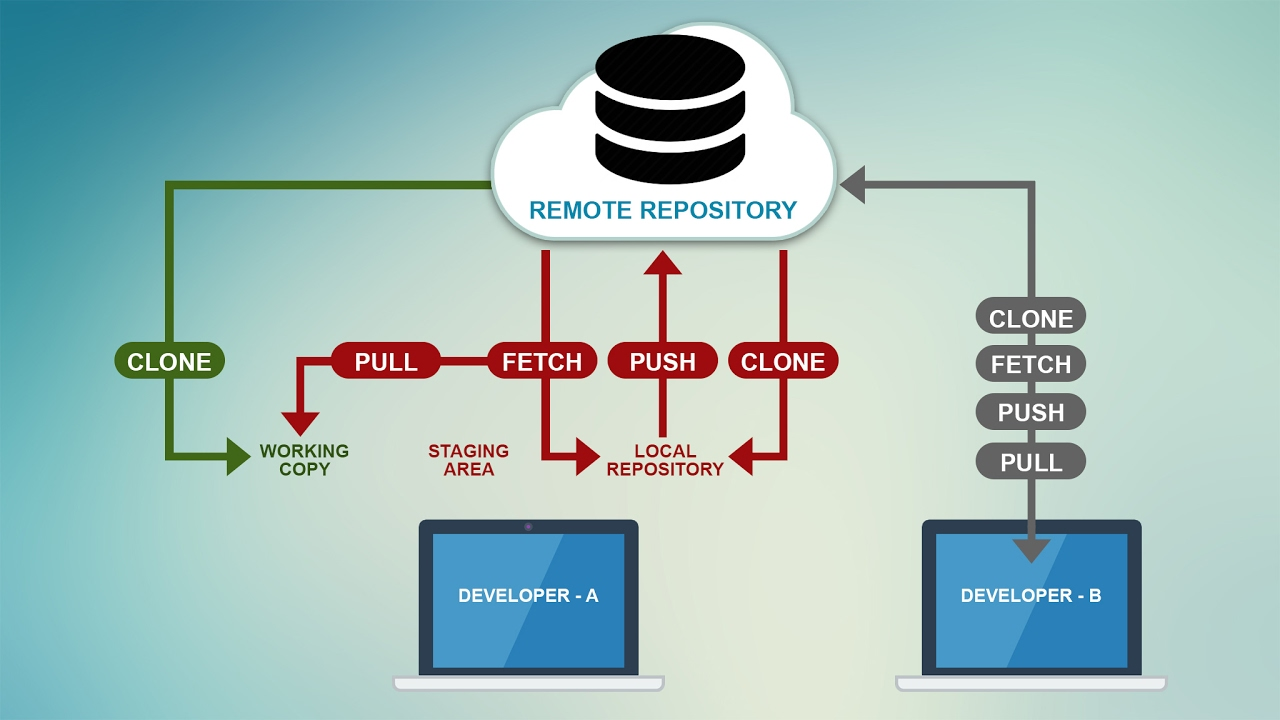
Важность команды заключается:

* делиться своим кодом с коллегами по команде
* иметь резервную копию на случай потери данных на своей машине

***Pull (пулл)*** - это скачивание данных с сервера.

Возникает вопрос: в чем же отличие пулл от клонирования? При написании пулл скачиваются не все коммиты, а только новые. Команда нужна чтобы получать изменения от других участников, если вы работаете в команде, или от себя самого, если работаете на разных машинах.

Эта команда является связкой команд git fetch и git merge. Так, отдельная команда git pull получает изменения и обновляет рабочую копию, которая станет соответствовать удаленному репозиторию.



Рассмотрим возможности работы с репозиториями, которые дает GitHub:

* Если мы хотим добавить чей-то репозиторий в собственный аккаунт, то используем функцию **fork** этого репозитория.

Со своим репозиторием мы можем делать все что угодно, а оригинал задет не будет.

* При работе над коллективным проектом можно послать **pull-request** (предложение о изменении кода), сделанные вами, другому пользователю.

Автор либо примет ваши изменения в оригинал, либо отклонит.

**Терминология**

**Дистрибутив** программы – это совокупность компонентов, необходимых для ее установки (инсталляции).

**Github Desktop** — программы под Windows 7+ и OS X, которая дублирует функциональность сайта github.com, но при этом работает локально на компьютере разработчика.

**Git Bash** — это приложение для сред Microsoft Windows, эмулирующее работу командной строки Git.

**Репозиторий** - специальное хранилище файлов и папок проекта, изменения в которых отслеживаются. Каталог, в котором хранится файловая система проекта. Для каждого проекта создаётся отдельный репозиторий. Существуют локальные и удалённые репозитории. В первом осуществляется работа над проектом на компьютере, а второй выступает в роли хранилища.

**Индекс в Git** — это специальная промежуточная область, в которой хранятся изменения файлов на пути от рабочей директории до репозитория. При выполнении коммита в него попадают только те изменения, которые были добавлены в индекс.

Один **коммит** — это пакет изменений, хранящий информацию с добавленными, отредактированными или удалёнными файлами кода. Операция позволяет зафиксировать текущее состояние проекта.

**Ветка**– это последовательность коммитов. очерняя версия основного репозитория. Она входит в его состав, но не влияет на работу. После того, как разработчики закончат работу над новой функцией или исправят все баги, можно совместить дочерний и родительский репозитории.

Описанные выше понятия встречаются очень часто, но также есть и определения, которые нужны не меньше:

**Форк**- копия репозитория, которую можно использовать для изменения исходного кода без отправки изменений в основной репозиторий.

**Пул и пуш** - первая операция позволяет выкачивать содержимое репозитория на компьютер, а вторая отправляет измененные файлы на сервер.

**Мастер** - основная ветка репозитория, в которой хранится ядро проекта. В неё добавляют изменения только после тщательного тестирования.

**Кодревью** - процесс проверки кода на соответствие техническому заданию или требованиям внутри команды. Когда один разработчик хочет добавить свой код в ядро, остальные члены команды проверяют его и если проблем нет, происходит обновление главной ветки.

**Пулреквест (Pull Request)** — запрос на слияние форка репозитория с основным репозиторием. Пулреквест может быть принят или отклонён вами, как владельцем репозитория.

**Checkout** - это команда, которая используется для переключения между ветвями.

**Merge** - это команда, которая используется для объединения изменений из одной ветки в другую.

**Fetch** - это команда, которая используется для извлечения изменений из удаленного репозитория в локальный репозиторий.

## **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.

Чудеса ветвления

## **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 "Я хитрее моего босса" > myfile.txt

$ git init

$ git add .

$ git commit -m "Начальный коммит"

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

$ git checkout -b boss # вероятно, это последнее изменение

$ echo "Мой босс умнее меня" > myfile.txt

$ git commit -a -m "Другой коммит"

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

$ git checkout master # переключиться на оригинальную версию файла

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

$ git checkout boss # перейти на версию, подходящую для глаз босса

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 "Ошибка исправлена"

$ 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 # часть2

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 # Возвращаемся к первой части.

$ вносим\_исправления

$ git commit -a # Фиксируем изменения

$ git checkout part2 # Возвращаемся ко второй части.

$ git merge master # Вливаем сделанные исправления.

В конечном счете, первая часть утверждена:

$ git checkout master # Возвращаемся к первой части.

$ отправка файлов # Выпускаем в мир!

$ git merge part2 # Вливаем вторую часть.

$ git branch -d part2 # Удаляем ветку 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 # Переименовываем ветку master в part2.

$ git branch master HEAD~7 # Создаем новую ветку master семью коммитами выше.

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 # Создаем ветку и переключаемся на нее.

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 # Создаем ветку для очищенных коммитов.

$ git checkout -b medley # Создаем ветку для работы и переключаемся на нее.

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.

## **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 # Возможно, понадобится устранить возникшие конфликты.

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.

## **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 Добавил ссылку repo.or.cz

pick a311a64 Переставил аналогии в «Работай как хочешь»

pick 100834f Добавил цель для push в 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 совершенно/секретный/файл' 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.

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.

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.

## **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

веб.сервер:/путь/к/proj.git master

и кто угодно сможет взять ваш проект с помощью

$ git clone <http://веб.сервер/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](http://www-cs-students.stanford.edu/~blynn/gitmagic/intl/ru/ch06.html" \l "makinghistory)**[и](http://www-cs-students.stanford.edu/~blynn/gitmagic/intl/ru/ch06.html" \l "makinghistory)**[git fast-import](http://www-cs-students.stanford.edu/~blynn/gitmagic/intl/ru/ch06.html" \l "makinghistory)**[могут преобразовать хранилища в один файл и обратно](http://www-cs-students.stanford.edu/~blynn/gitmagic/intl/ru/ch06.html" \l "makinghistory). 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 некий-файл HEAD

Затем передает «пакет», некий-файл, 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 некий-файл

Получатель может сделать это даже в пустом хранилище. Несмотря на свой небольшой размер, некий-файл 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 некий-файл 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 последний-пакет HEAD

и создавайте обновленные пакеты с помощью

$ git bundle create новый-пакет HEAD ^последний-пакет

## **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 < мой.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://новый.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://пример.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://пример.com/некое\_хранилище.git

$ git pull other некая\_ветка

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/некая\_ветка~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 # Перенести из origin, по

умолчанию.

$ git fetch other # Перенести от

второго программиста.

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**.

## **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 минут назад, прим. пер.

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 # отображает текущие алиасы

alias.co checkout

$ git co foo # то-же, что и «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 существующее/хранилище новый/каталог

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 мертвая\_ветка # вместо -d

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

$ git branch -M источник цель # вместо -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 # В старых версиях 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](http://www.kernel.org/pub/software/scm/git/docs/user-manual.html).

## **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](http://lkml.org/lkml/2005/4/6/121) 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 > ВАШЕ\_ИМЯ\_ФАЙЛА

$ 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](http://www.zlib.net/zpipe.c" \t "_top), 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 # Введите какое-нибудь описание

$ 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 ВАШЕ\_ИМЯ\_ФАЙЛА 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 соответстия тому, что мы имеем, мы должны немного «подкрутить» Git:

$ git commit --amend -m Shakespeare # Изменим описание коммита.

$ 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"' # Подделаем временные метки и авторов.

$ 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 надеждности. 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](http://cygwin.com/), Linux-подобная среда для Windows, содержащая [порт Git на Windows](http://cygwin.com/packages/git/).
* [Git для Windows](https://gitforwindows.org/), 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 А и Б значения выражений «А..Б» и «А…Б» зависят от того, ожидает ли команда указания двух конечных точек или промежутка. Смотрите **git help diff** и **git help rev-parse**.