**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.

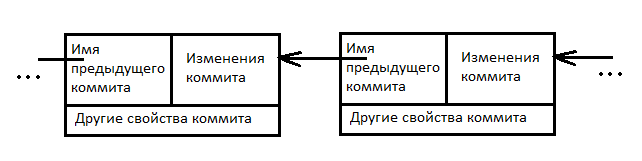
**Создание репозитория**

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.

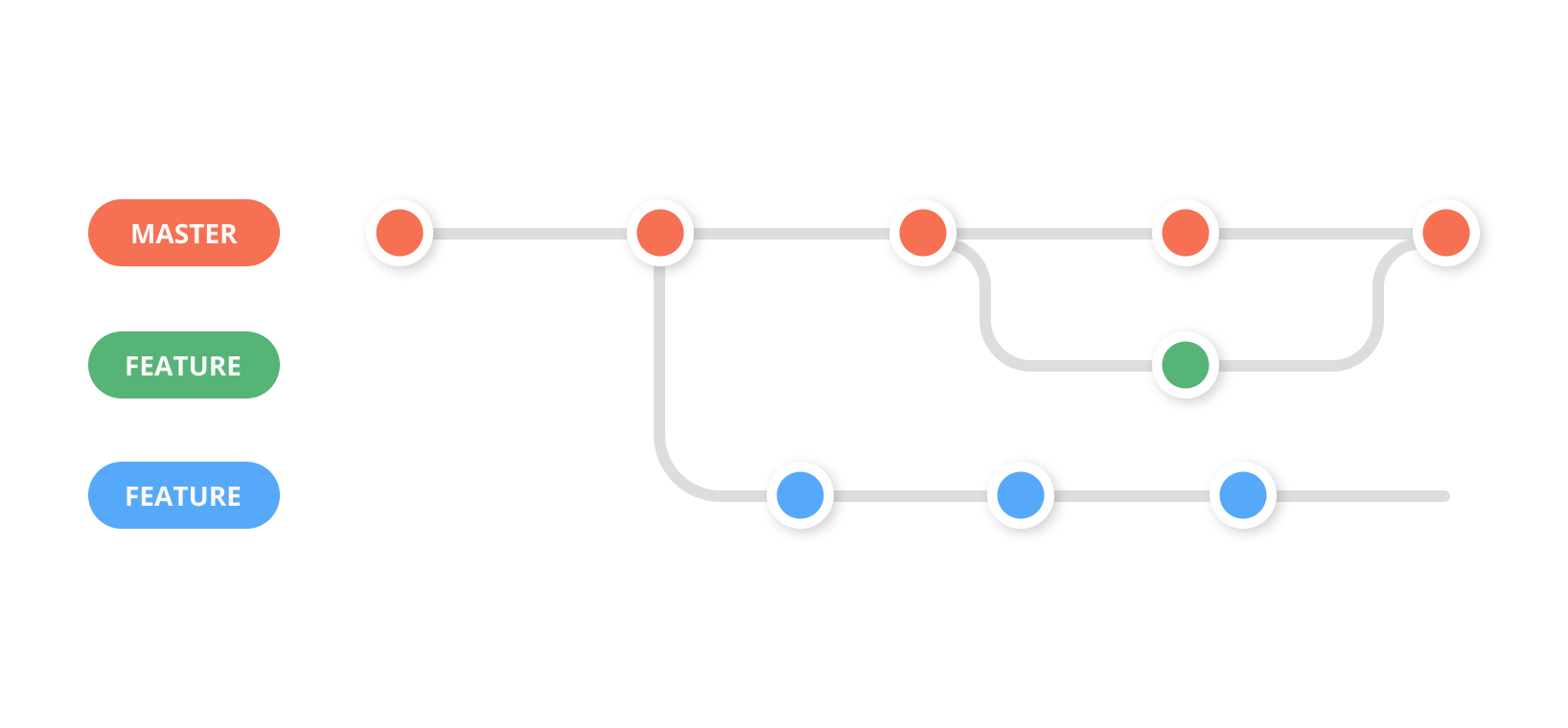
**Работа с ветками**

***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.

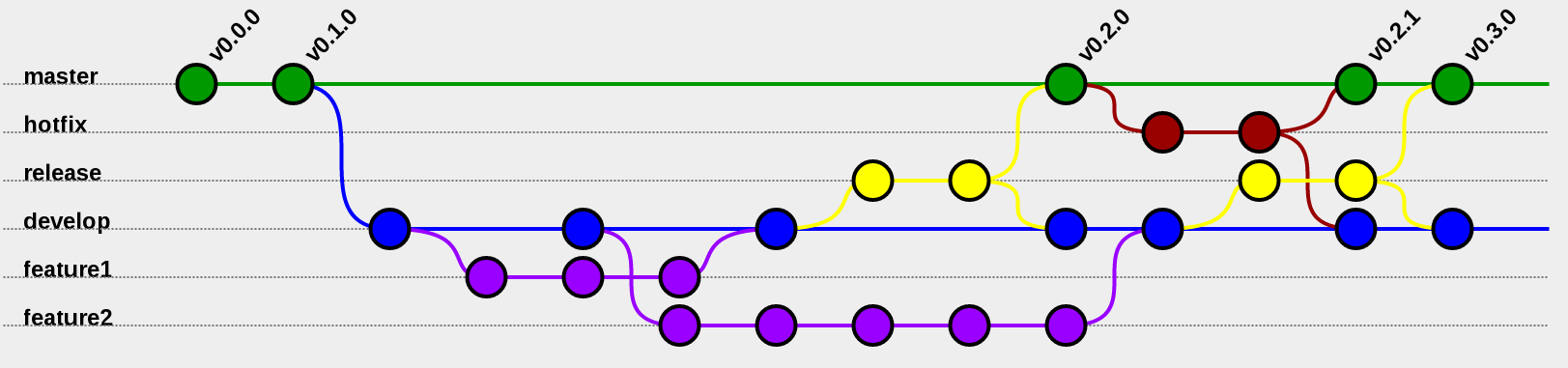
Во время создания проекта под управлением системы контроля версий создается ветка по умолчанию – master.

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.

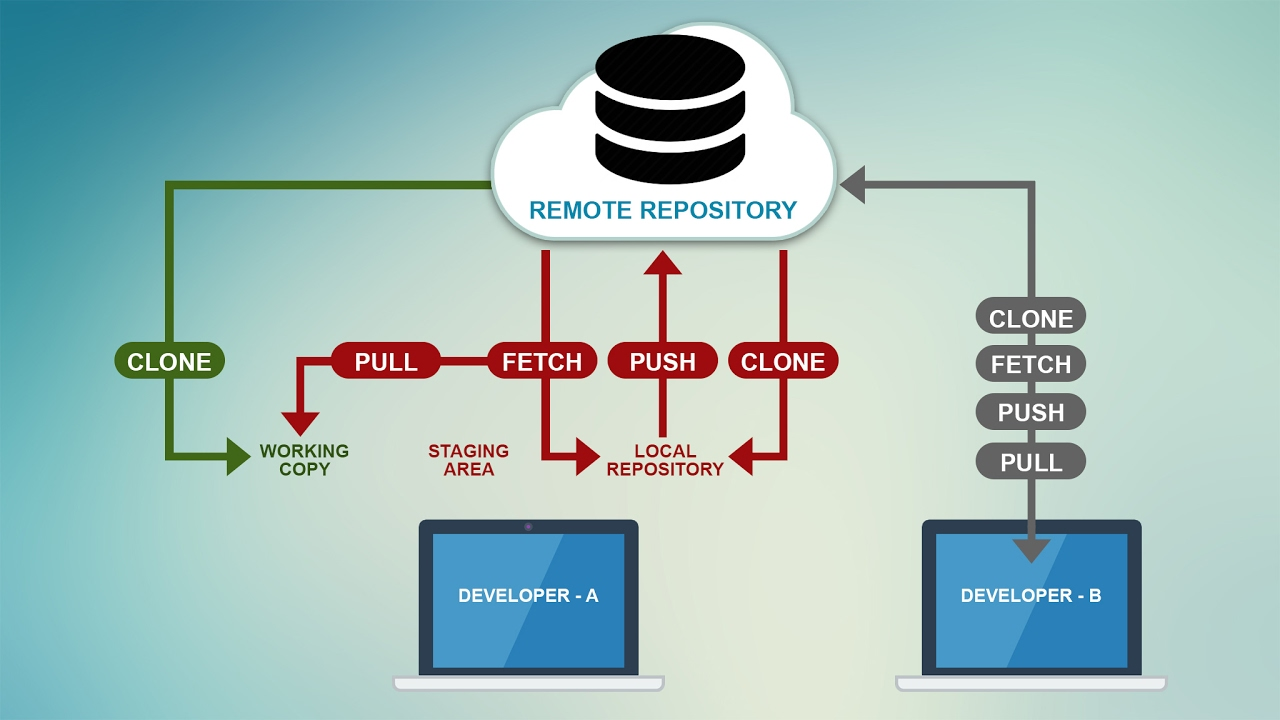
Важность команды заключается:

* 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**.