<http://habrahabr.ru/post/60030/>

1 Вступление

В топике освещаются не столько подробности работы с git, сколько его отличия от схемы разработки других систем контроля версий, и общий подход (выработанный по большей части личным опытом и Git Community Book) к работе.

2 Распределенный подход

Среди открытых разработок на данную тему можно вспомнить git, Mercurial и Bazaar. Первый проект особенно интересен, он используется в некоторых из сложнейших современных программных систем(Linux Kernel, Qt, Wine, Gnome, Samba и многие другие), крайне быстро работает с любым объемом кода и сейчас набирает популярность в открытом мире. Какое-то время на распространении этой программы негативно сказывался недостаток документации; но сейчас этот недостаток можно считать устраненным.  
  
Итак, в чем заключается глобальное отличие git? Во-первых, как следует из самого названия, не существует главного (в том смысле, который его понимают разработчики, привыкшие к SVN) репозитария. У каждого разработчика имеется собственный полноценный репозитарий, с которым и ведется работа; периодически проводится синхронизация работы с (чисто условно!) центральным репозитарием. Во вторых, операции ветвления и слияния веток (merging) ставятся во главу угла при работе программиста, и поэтому они очень легковесны.

3 Работа с репозитарием

Для создания нового репозитария достаточно просто зайти в папку проекта и набрать:  
  
git init  
  
Был создан пустой репозитарий — папка .git в корне проекта, в которой и будет собираться вся информация о дальнейшей работе. Предположим, уже существует несколько файлов, и их требуется проиндексировать командой git add:  
  
git add .  
  
Внесем изменения в репозитарий:  
  
git commit -m "Первоначальный коммит"  
  
Готово! Имеется готовый репозитарий с единственной веткой. Допустим, потребовалось разработать какой-то новый функционал. Для этого создадим новую ветку:  
  
git branch new-feature  
  
И переключимся на нее:  
  
git checkout new-feature  
  
Вносим необходимые изменения, после чего смотрим на них, индексируем и коммитимся:  
  
git status   
git add .   
git commit -m "new feature added"  
  
Теперь у нас есть две ветки, одна из которых (master) является условно (технически же ничем не отличается) основной. Переключаемся на нее и включаем изменения (сливаем с другой веткой):

git checkout master   
git merge new-feature  
  
Легко и быстро, не находите? Веток может быть неограниченное количество, из них можно создавать патчи, определять diff с любым из совершенных коммитов.  
  
Теперь предположим, что во время работы выясняется: нашелся небольшой баг, требующий срочного внимания. Есть два варианта действий в таком случае. Первый состоит из создания новой ветки, переключения в нее, слияния с основой… Второй — команда git stash. Она сохраняет все изменения по сравнению с последним коммитом во временной ветке и сбрасывает состояние кода до исходного:  
  
git stash  
  
Исправляем баг и накладываем поверх произведенные до того действия (проводим слияние с веткой stash):  
  
git stash apply  
  
Вот и все. Очень удобно. На самом деле таких «заначек» (stash) может быть сколько угодно; они просто нумеруются.  
  
При такой работе появляется необычная гибкость; но среди всех этих веточек теряется понятие ревизии, характерное для линейных моделей разработки. Вместо этого каждый из коммитов (строго говоря, каждый из объектов в репозитарии) однозначно определяется хэшем. Естественно, это несколько неудобно для восприятия, поэтому разумно использовать механизм тэгов для того, чтобы выделять ключевые коммиты:

git tag

просто именует последний коммит;

git tag -a

также дает имя коммиту, и добавляет возможность оставить какие-либо комментарии (аннотацию). По этим тегам можно будет в дальнейшем обращаться к истории разработки.  
  
Плюсы такой системы очевидны! Вы получаете возможность колдовать с кодом как душе угодно, а не как диктует система контроля версий: разрабатывать параллельно несколько «фишек» в собственных веточках, исправлять баги, чтобы затем все это дело сливать в единую кашу главной ветки. Замечательно быстро создаются, удаляются или копируются куда угодно папочки .git с репозитарием.  
  
Гораздо удобней такую легковесную систему использовать для хранения версий документов, файлов настроек и т.д, и т.п. К примеру, настройки и плагины для Емакса я храню в директории ~/site-lisp, и держу в том же месте репозитарий; и у меня есть две ветки: work и home; иногда бывает удобно похожим образом управлять настройками в /etc. Естественно, что каждый из моих личных проектов тоже находит под управлением git.

4 Общественные репозитарии

Общественный репозитарий — способ обмениваться кодом в проектах, где участвует больше двух человек. Лично я использую сайт github.com, настолько удобный, что многие начинают из-за него пользоваться git.  
  
Итак, создаем у себя копию удаленного репозитария:  
  
git clone git://github.com/username/project.git master  
  
Команда создала у вас репозитарий, и внесла туда копию ветки master проекта project. Теперь можно начинать работу. Создадим новую ветку, внесем в нее изменения, закоммитимся:  
  
git branch new-feature   
edit README   
git add .   
git commit -m "Added a super feature"  
  
Перейдем в основную ветку, заберем последние изменения в проекте, и попробуем добавить новую фишку в проект:  
  
git checkout master   
git pull   
git merge new-feature  
  
Если не было неразрешенных конфликтов, то коммит слияния готов.  
  
Команда git pull использует так называемую удаленную ветку (remote branch), создаваемую при клонировании удаленного репозитария. Из нее она извлекает последние изменения и проводит слияние с активной веткой.  
  
Теперь остается только занести изменения в центральный (условно) репозитарий:  
  
git push  
  
Нельзя не оценить всю гибкость, предоставляемую таким средством. Можно вести несколько веток, отсылать только определенную, жонглировать коммитами как угодно.  
  
В принципе, никто не мешает разработать альтернативную модель разработки. Например, использовать иерархическую систему репозитариев, когда «младшие» разработчики делают коммиты в промежуточные репозитарии, где те проходят проверку у «старших» программистов и только потом попадают в главную ветку центрального репозитария проекта.  
  
При работе в парах возможно использовать симметричную схему работы. Каждый разработчик ведет по два репозитария: рабочий и общественный. Первый используется в работе непосредственно, второй же, доступный извне, только для обмена уже законченным кодом.

<http://habrahabr.ru/post/60347/>

1 Введение

В своей [**прошлой**](http://habrahabr.ru/blogs/Git/60030/)заметке я постарался осветить в общих чертах стиль работы с  
распределенной системой контроля версий git и указать на отличия по сравнению с  
классическими централизованными СКВ. Целью было прежде всего обобщение опыта  
работы с системой без упоминания тонкостей синтаксиса отдельных команд.  
  
Данный же топик задумывался как непосредственное введение в работу с git, нечто  
среднее между tutorial и обобщенной справкой, до которого все же рекомендуется  
прочитать упомянутое выше введение. Сознательно избегаются технические  
подробности работы git, употребляются только общие для СКВ термины и  
ограничивается список упоминаемых команд.

2 Работа с локальным репозитарием

Сила любых распределенных систем — в наличии у каждого разработчика локального  
репозитария, в котором возможно организовывать произвольную личную схему  
разработки. В git есть несколько основных команды для ведения работы на месте и  
множество вспомогательных.

2.1 Базовые команды

Базовые команды — те, без которых невозможно обойтись в разработке.

2.1.1 *git init* — создание репозитария

Команда git init создает в директории пустой репозитарий в виде директория  
.git, где и будет в дальнейшем храниться вся информация об истории коммитов,  
тегах — ходе разработки проекта:  
  
mkdir project-dir  
  
cd project-dir  
  
git init  
  
Другой способ создать репозитарий — команда git clone, но о ней чуть позже.

2.1.2 *git add* и *git rm* — индексация изменений

Следующее, что нужно знать — команда git add. Она позволяет внести в индекс — временное хранилище — изменения, которые затем войдут в коммит. Примеры  
использования:  
  
git add EDITEDFILE — индексация измененного файла, либо оповещение о  
создании нового.  
  
git add. — внести в индекс все изменения, включая новые файлы.  
  
Из индекса и дерева одновременно проекта файл можно удалить командой git rm:  
  
git rm FILE1 FILE2 — отдельные файлы  
  
git rm Documentation/\\*.txt — хороший пример удаления из документации к git,  
удаляются сразу все файлы txt из папки.  
  
Сбросить весь индекс или удалить из него изменения определенного файла можно  
командой git reset:  
  
git reset — сбросить нафиг весь индекс.  
  
git reset — EDITEDFILE — удалить из индекса конкретный файл.  
  
Команда git reset используется не только для сбрасывания индекса, поэтому дальше  
ей будет уделено гораздо больше внимания. 

2.1.3 *git status* — состояние проекта, измененные и не добавленные файлы, индексированные файлы

Команда git status, пожалуй, можно считать самой часто используемой наряду с  
командами коммита и индексации. Она выводит информацию обо всех изменениях,  
внесенных в дерево директорий проекта по сравнению с последним коммитом рабочей  
ветки; отдельно выводятся внесенные в индекс и неиндексированные  
файлы. Использовать ее крайне просто:  
  
git status  
  
Кроме того, git status указывает файлы с неразрешенными конфликтами слияния и  
файлы, игнорируемые git.

2.1.4 *git commit* — совершение коммита

Коммиты — базовое понятие во всех системах контроля версий, поэтому совершатся  
он должен легко и по возможности быстро. В самом своем простом виде достаточно  
после индексации набрать:   
  
git commit  
  
Если индекс не пустой, то на его основе будет совершен коммит, после чего  
пользователя попросят прокомментировать вносимые изменения вызовом команды  
edit(например, в Ubuntu обычно вызывается простенький текстовый редактор nano, у  
меня же — emacs). Сохраняемся, и вуала! Коммит готов.  
  
Есть несколько ключей, упрощающих работу с git commit:  
  
git commit -a — совершит коммит, автоматически индексируя изменения в файлах  
проекта. Новые файлы при этом индексироваться **не будут**! Удаление же файлов  
будет учтено.  
  
  
git commit -m «commit comment» — комментируем коммит прямо из командной строки  
вместо текстового редактора.  
  
git commit FILENAME — внесет в индекс и создаст коммит на основе изменений  
единственного файла. 

2.1.5 *git reset* — возврат к определенному коммиту, откат изменений, «жесткий» или «мягкий»

Помимо работы с индексом (см. выше), git reset позволяет сбросить состояние  
проекта до какого-либо коммита в истории. В git данное действие может быть двух  
видов: «мягкого»(soft reset) и «жесткого» (hard reset).  
  
  
«Мягкий» (с ключом "--soft") резет оставит нетронутыми ваши индекс и все дерево  
файлов и директорий проекта, вернется к работе с указанным коммитом. Иными  
словами, если вы обнаруживаете ошибку в только что совершенном коммите или  
комментарии к нему, то легко можно исправить ситуацию:

1. git commit… — некорректный коммит;
2. git reset --soft HEAD^ — переходим к работе над уже совершенным коммитом,  
   сохраняя все состояние проекта и проиндексированные файлы
3. edit WRONGFILE
4. edit ANOTHERWRONGFILE
5. add.

6.1. git commit -c ORIG\_HEAD — вернутся к последнему коммиту, будет предложено  
редактировать его сообщение. Если сообщение оставить прежним, то  
достаточно изменить регистр ключа -с:  
  
6.2. git commit -C ORIG\_HEAD  
  
Обратите внимание на обозначение HEAD^, оно означает «обратиться к предку  
последнего коммита». Подробней описан синтаксис такой относительной адресации  
будет ниже, в разделе «Хэши, тэги, относительная адресация». Соответственно,  
HEAD — ссылка на последний коммит. Ссылка ORIG\_HEAD после «мягкого» резета  
указывает на оригинальный коммит.  
  
  
Естественно, можно вернуться и на большую глубину коммитов,  
  
«Жесткий» резет (ключ --hard) — команда, которую следует использовать с  
осторожностью. Git reset --hard вернет дерево проекта и индекс в состояние,  
соответствующее указанному коммиту, удалив изменения последующих коммитов:  
  
git add.  
  
git commit -m «destined to death»  
  
git reset --hard HEAD~1 — больше никто и никогда не увидит этот позорный коммит.  
  
git reset --hard HEAD~3 — вернее, три последних коммита. Никто. Никогда.  
  
  
Если команда достигнет точки ветвления, удаления коммита не произойдет.   
  
Для команд слияния или выкачивания последних изменений с удаленного репозитария  
примеры резета будут приведены в соответствующих разделах.

2.1.6 *git revert* — отмена изменений, произведенных в прошлом отдельным коммитом

Возможна ситуация, в которой требуется отменить изменения, внесенные отдельным  
коммитом. Git revert создает новый коммит, накладывающий обратные изменения:  
  
  
git revert config-modify-tag — отменяем коммит, помеченный тегом.  
  
git revert 12abacd — отменяем коммит, используя его хэш.  
  
Для использования команды необходимо, чтобы состояние проекта не отличалось от  
состояния, зафиксированного последним коммитом.

2.1.7 *git log* — разнообразная информация о коммитах в целом, по отдельным файлам и различной глубины погружения в историю

Иногда требуется получить информацию об истории коммитов, коммитах, изменивших  
отдельный файл; коммитах за определенный отрезок времени и так далее. Для этих  
целей используется команда git log.  
  
Простейший пример использования, в котором приводится короткая справка по всем  
коммитам, коснувшимся активной в настоящий момент ветки (о ветках и ветвлении  
подробно узнать можно ниже, в разделе «Ветвления и слияния»):  
  
git log   
  
Получить подробную информацию о каждом в виде патчей по файлам из коммитов  
можно, добавив ключ -p (или -u):  
  
git log -p  
  
Статистика изменения файлов, вроде числа измененных файлов, внесенных в них  
строк, удаленных файлов вызывается ключом --stat:  
  
  
git log --stat  
  
За информацию по созданиям, переименованиям и правам доступа файлов отвечает ключ  
--summary:  
  
git log --summary  
  
Для исследования истории отдельного файла достаточно указать в виде параметра  
его имя (кстати, в моей старой версии git этот способ не срабатывает,  
обязательно добавлять " — " перед «README»):  
  
git log README  
  
или, если версия git не совсем свежая:  
  
git log — README  
  
Далее будет приводится только более современный вариант синтаксиса. Возможно  
указывать время, начиная в определенного момента («weeks», «days», «hours», «s»  
и так далее):  
  
git log --since=«1 day 2 hours» README   
  
git log --since=«2 hours» README   
  
git log --since=«2 hours» dir/ — изменения, касающиеся отдельной папки.  
  
  
Можно отталкиваться от тегов:  
  
git log v1… — все коммиты, начиная с тега v1.  
  
git log v1… README — все коммиты, включающие изменения файла README, начиная с  
тега v1.  
  
git log v1..v2 README — все коммиты, включающие изменения файла README, начиная с  
тега v1 и заканчивая тегом v2.  
  
Создание, выведение списка, назначение тегов будет приведено в соответствующем  
разделе ниже.  
  
Интересные возможности по формату вывода команды предоставляет ключ --pretty:  
  
  
git log --pretty=oneline — выведет на каждый из коммитов по строчке, состоящей из хэша  
(здесь — уникального идентификатора каждого коммита, подробней — дальше).  
  
git log --pretty=short — лаконичная информация о коммитах, приводятся только  
автор и комментарий  
  
git log --pretty=full/fuller — более полная информация о коммитах, с именем  
автора, комментарием, датой создания и внесения коммита  
  
В принципе, формат вывода можно определить самостоятельно:   
  
git log --pretty=format:'FORMAT'  
  
Определение формата можно поискать в разделе по git log из Git Community Book  
или справке. Красивый ASCII-граф коммитов выводится с использованием ключа  
--graph.

2.1.8 *git diff* — отличия между деревьями проекта; коммитами; состоянием индекса и каким-либо коммитом.

Своего рода подмножеством команды git log можно считать команду git diff,  
определяющую изменения между объектами в проекте: деревьями (файлов и  
директорий):  
  
git diff — покажет изменения, не внесенные в индекс.  
  
git diff --cached — изменения, внесенные в индекс.  
  
  
git diff HEAD — изменения в проекте по сравнению с последним коммитом  
  
git diff HEAD^ — предпоследним коммитом  
  
Можно сравнивать «головы» веток:  
  
git diff master..experimental  
  
Ну или активную ветку с какой-либо:  
  
git diff experimental

2.1.9 *git show* — показать изменения, внесенные отдельным коммитом

Посмотреть изменения, внесенные любым коммитом в истории можно командой git  
show:  
  
git show COMMIT\_TAG

2.1.10 *git blame* и *git annotate* — вспомогательные команды, помогающие отслеживать изменения файлов

При работе в команде часто требуется выяснить, кто именно написал конкретный  
код. Удобно использовать команду git blame, выводящую построчную информацию о  
последнем коммите, коснувшемся строки, имя автора и хэш коммита:  
  
git blame README  
  
Можно указать и конкретные строки для отображения:  
  
git blame -L 2,+3 README — выведет информацию по трем строкам, начиная со второй.  
  
Аналогично работает команда git annotate, выводящая и строки, и информацию о  
коммитах, их коснувшихся:  
  
git annotate README

2.1.11 *git grep* — поиск слов по проекту, состоянию проекта в прошлом

git grep, в целом, просто дублирует функционал знаменитой юниксовой  
команды. Однако, он позволяет слова и их сочетания искать в прошлом проекта, что  
бывает очень полезно:  
  
git grep tst — поиск слова tst в проекте.  
  
git grep -с tst — подсчитать число упоминаний tst в проекте.  
  
  
git grep tst v1 — поиск в старой версии проекта.  
  
Команда позволяет использовать логическое И и ИЛИ:  
  
git grep -e 'first' --and -e 'another' — найти строки, где упоминаются и первое  
слово, и второе.  
  
git grep --all-match -e 'first' -e 'second' — найти строки, где встречается хотя  
бы одно из слов.

2.2 Ветвление

Операции ветвления и слияния — сердце и душа git, именно эти возможности делают такой  
удобной работу с системой.

2.2.1 *git branch* — создание, перечисление и удаление веток

Работа с ветками — очень легкая процедура в git, все необходимые механизмы  
сконцентрированы в одной команде:  
  
git branch — просто перечислит существующие ветки, отметив активную.  
  
git branch new-branch — создаст новую ветку new-branch.  
  
git branch -d new-branch — удалит ветку, если та **была залита** (merged) с  
разрешением возможных конфликтов в текущую.  
  
git branch -D new-branch — удалит ветку **в любом** случае.  
  
git branch -m new-name-branch — переименует ветку.  
  
  
git branch --contains v1.2 — покажет те ветки, среди предков которых есть  
определенный коммит.

2.2.2 *git checkout* — переключение между ветками, извлечение отдельных файлов из истории коммитов

Команда git checkout позволяет переключаться между последними коммитами (если  
упрощенно) веток:  
  
checkout some-other-branch  
  
checkout -b some-other-new-branch — создаст ветку, в которую и произойдет  
переключение.  
  
Если в текущей ветке были какие-то изменения по сравнению с последним коммитом в  
ветке(HEAD), то команда откажется производить переключение, дабы не потерять  
произведенную работу. Проигнорировать этот факт позволяет ключ -f:  
  
checkout -f some-other-branch  
  
В случае, когда изменения надо все же сохранить, используют ключ -m. Тогда команда  
перед переключением попробует залить изменения в текущую ветку и, после  
разрешения возможных конфликтов, переключиться в новую:  
  
checkout -m some-other-branch  
  
  
Вернуть файл (или просто вытащить из прошлого коммита) позволяет команда вида:  
  
git checkout somefile — вернуть somefile к состоянию последнего коммита  
git checkout HEAD~2 somefile — вернуть somefile к состоянию на два коммита назад по ветке.

2.2.3 *git merge* — слияние веток (разрешение возможных конфликтов).

Слияние веток, в отличие от обычной практики централизованных систем, в git  
происходит практически каждый день. Естественно, что имеется удобный интерфейс к  
популярной операции:  
  
git merge new-feature — попробует объединить текующую ветку и ветку new-feature.  
  
В случае возникновения конфликтов коммита не происходит, а по проблемным файлам  
расставляются специальные метки а ля svn; сами же файлы отмечаются в индексе как  
«не соединенные» (unmerged). До тех пор пока проблемы не будут решены, коммит совершить  
будет нельзя.  
  
Например, конфликт возник в файле TROUBLE, что можно увидеть в git status:  
  
git merge experiment — произошла неудачная попытка слияния.  
  
git status — смотрим на проблемные места.  
  
edit TROUBLE — разрешаем проблемы.  
  
git add. — индексируем наши изменения, тем самым снимая метки.  
  
git commit — совершаем коммит слияния.  
  
Вот и все, ничего сложного. Если в процессе разрешения вы передумали разрешать  
конфликт, достаточно набрать:  
  
git reset --hard HEAD — это вернет обе ветки в исходные состояния.  
  
  
Если же коммит слияния был совершен, используем команду:  
  
git reset --hard ORIG\_HEAD

2.2.4 *git rebase* — построение ровной линии коммитов

Предположим, разработчик завел дополнительную ветку для разработки отдельной  
возможности и совершил в ней несколько коммитов. Одновременно по какой-либо  
причине в основной ветке также были совершены коммиты: например, в нее были  
залиты изменения с удаленного сервера; либо сам разработчик совершал в ней  
коммиты.  
  
В принципе, можно обойтись обычным *git merge*. Но тогда усложняется сама линия  
разработки, что бывает нежелательно в слишком больших проектах, где участвует  
множество разработчиков.   
  
Предположим, имеется две ветки, master и топик, в каждой из которых было совершенно несколько коммитов начиная с момента ветвления.  
Команда git rebase берет коммиты из ветки topic и накладывает их на последний коммит ветки  
master:

1. git-rebase master topic — вариант, в котором явно указывается, что и куда  
   прикладывается.
2. git-rebase master — на master накладывается активная в настоящий момент  
   ветка.

После использования команды история становится линейной. При возникновении   
конфликтов при поочередном накладывании коммитов  
работа команды будет останавливаться, а в проблемные местах файлов появятся  
соответствующие метки. После редактирования — разрешения конфликтов — файлы  
следует внести в индекс командой git add и продолжить наложение следующих  
коммитов командой git rebase --continue. Альтернативными выходами будут команды  
git rebase --skip (пропустить наложение коммита и перейти к следующему) или git  
rebase --abort (отмена работы команды и всех внесенных изменений).  
  
С ключом -i (--interactive) команда будет работать в интерактивном  
режиме. Пользователю будет предоставлена возможность определить порядок внесения  
изменений, автоматически будет вызывать редактор для разрешения конфликтов и так  
далее.

2.2.5 *git cherry-pick* — применение к дереву проекта изменений, внесенных отдельным коммитом

Если ведется сложная история разработки, с несколькими длинными ветками  
разработками, может возникнуть необходимость в применении изменений, внесенных  
отдельным коммитом одной ветки, к дереву другой (активной в настоящий момент).   
  
git cherry-pick BUG\_FIX\_TAG — изменения, внесенные указанным коммитом будут  
применены к дереву, автоматически проиндексированы и станут коммитом в активной  
ветке.  
  
git cherry-pick BUG\_FIX\_TAG -n — ключ "-n" показывает, что изменения надо  
просто применить к дереву проекта без индексации и создания коммита.

2.3 Прочие команды и необходимые возможности

Для удобства работы с git было введено дополнительное понятие: тэг. Кроме того  
дальше будет пояснена необходимость в хэшах, и его применение; показан способ  
обращаться к коммитам при помощи относительной адресации.

2.3.1 Хэш — уникальная идентификация объектов

В git для идентификации любых объектов используется уникальный (то есть с  
огромной вероятностью уникальный) хэш из 40 символов, который определяется  
хэшируюшей функцией на основе содержимого объекта. Объекты — это все: коммиты,  
файлы, тэги, деревья. Поскольку хэш уникален для содержимого, например, файла,  
то и сравнивать такие файлы очень легко — достаточно просто сравнить две строки  
в сорок символов.  
  
Больше всего нас интересует тот факт, что хэши идентифицируют коммиты. В этом  
смысле хэш — продвинутый аналог ревизий Subversion. Несколько примеров  
использования хэшей в качестве способа адресации:  
  
git diff f292ef5d2b2f6312bc45ae49c2dc14588eef8da2 — найти разницу текущего  
состояния проекта и коммита за номером… Ну сами видите, каким.  
  
git diff f292ef5 — то же самое, но оставляем только шесть первых символов. Git  
поймет, о каком коммите идет речь, если не существует другого коммита с таким  
началом хэша.  
  
git diff f292 — иногда хватает и четырех символов.  
  
git log febc32...f292 — читаем лог с коммита по коммит.  
  
  
Разумеется, человеку пользоваться хэшами не так удобно, как машине, именно поэтому были  
введены другие объекты — тэги.

2.3.2 *git tag* — тэги как способ пометить уникальный коммит

Тэг (tag) — это объект, связанный с коммитом; хранящий ссылку на сам коммит, имя  
автора, собственное имя и некоторый комментарий. Кроме того, разработчик может  
оставлять на таких тегах собственную цифровую подпись.  
  
Кроме этого в git представленные так называемые «легковесные тэги» («lightweight  
tags»), состоящие только из имени и ссылки на коммит. Такие тэги, как правило,  
используются для упрощения навигации по дереву истории; создать их очень легко:  
  
git tag stable-1 — создать «легковесный» тэг, связанный с последним  
коммитом. Если тэг уже есть, то еще один создан не будет.  
  
git tag stable-2 f292ef5 — пометить определенный коммит.   
  
git tag -d stable-2 — удалить тег.  
  
git tag -l — перечислить тэги.  
  
git tag -f stable-1.1 — создать тэг для последнего коммита, заменить  
существующий, если таковой уже был.  
  
После создания тэга его имя можно использовать вместо хэша в любых командах  
вроде git diff, git log и так далее:  
  
  
git diff stable-1.1...stable-1  
  
Обычные тэги имеет смысл использовать для приложения к коммиту какой-либо  
информации, вроде номера версии и комментария к нему. Иными словами, если в  
комментарии к коммиту пишешь «исправил такой-то баг», то в комментарии к тэгу по  
имени «v1.0» будет что-то вроде «стабильная версия, готовая к использованию»:  
  
git tag -a stable — создать обычный тэг для последнего коммита; будет вызван  
текстовый редактор для составления комментария.  
  
git tag -a stable -m «production version» — создать обычный тэг, сразу указав в  
качестве аргумента комментарий.  
  
Команды перечисления, удаления, перезаписи для обычных тэгов не отличаются от  
команд для «легковесных» тэгов.

2.3.3 Относительная адресация

Вместо ревизий и тэгов в качестве имени коммита можно опираться на еще один  
механизм — относительную адресацию. Например, можно обратиться прямо к предку  
последнего коммита ветки master:  
  
git diff master^  
  
Если после «птички» поставить цифру, то можно адресоваться по нескольким предкам  
коммитов слияния:  
  
git diff HEAD^2 — найти изменения по сравнению со вторым предком последнего  
коммита в master. HEAD здесь — указатель на последний коммит активной ветки.  
  
Аналогично, тильдой можно просто указывать, насколько глубоко в историю ветки  
нужно погрузиться:  
  
git diff master^^ — что привнес «дедушка» нынешнего коммита.  
  
git diff master~2 — то же самое.  
  
Обозначения можно объединять, чтобы добраться до нужного коммита:  
  
git diff master~3^~2  
  
git diff master~6

2.3.4 *файл .gitignore* — объясняем git, какие файлы следует игнорировать

Иногда по директориям проекта встречаются файлы, которые не хочется постоянно  
видеть в сводке git status. Например, вспомогательные файлы текстовых  
редакторов, временные файлы и прочий мусор.   
  
Заставить git status игнорировать можно, создав в корне или глубже по дереву  
(если ограничения должны быть только в определенных директория) файл  
.gitignore. В этих файлах можно описывать шаблоны игнорируемых файлов  
определенного формата.  
  
Пример содержимого такого файла:  
  
>>>>>>>Начало файла  
  
#комментарий к файлу .gitignore  
  
#игнорируем сам .gitignore  
  
  
.gitignore  
  
#все html-файлы…  
  
  
\*.html  
  
  
#… кроме определенного  
  
  
!special.html  
  
  
#не нужны объектники и архивы  
  
\*.[ao]  
  
>>>>>>>>Конец файла  
  
Существуют и другие способы указания игнорируемых файлов, о которых можно узнать  
из справки git help gitignore.

3 «Вместе мы — сила», или основы работы с удаленным репозитарием

Естественно, что большая часть проектов все-таки подразумевает работу по крайней  
мере двух разработчиков, которым требуется обмениваться кодом. Далее будут  
перечислены команды, требующиеся для совместной — возможно удаленной — работы.

3.1 Удаленные ветки (remote tracking branches)

Новое понятие здесь — удаленные ветки. Удаленные ветки соответствуют какой-либо  
ветке (чаще master) на удаленном сервере. Одна такая создается автоматически при  
создании копии удаленного репозитария; все команды, связанные с удаленной  
работой, будут по умолчанию использовать именно эту удаленную ветку (обычно  
называется «origin»).  
  
Рассмотрим эти команды.

3.2 *git clone* — создание копии (удаленного) репозитария

Для начала работы с центральным репозитарием, следует создать копию  
оригинального проекта со всей его историей локально:  
  
git clone /home/username/project myrepo — клонируем репозитарий с той же машины  
в директорию myrepo.  
  
git clone ssh://user@somehost:port/~user/repository — клонируем репозитарий,  
используя безопасный протокол ssh (для чего требуется завести у себя на машине  
эккаунт ssh).  
  
git clone git://user@somehost:port/~user/repository/project.git/ — у git имеется  
и собственный протокол.

3.3 *git fetch* и *git pull* — забираем изменения из центрального репозитария (из удаленной ветки)

Для синхронизации текущей ветки с репозитарием используются команды git fetch и  
git pull.  
  
git fetch — забрать изменения удаленной ветки из репозитария по умолчания,  
основной ветки; той, которая была использована при клонировании  
репозитария. Изменения обновят удаленную ветку (remote tracking branch), после  
чего надо будет провести слияние с локальной ветку командой git merge.  
  
git fetch /home/username/project — забрать изменения из определенного  
репозитария.  
  
Возможно также использовать синонимы для адресов, создаваемые командой git remote:  
  
git remote add username-project /home/username/project  
  
git fetch username-project — забрать изменения по адресу, определяемому  
синонимом.   
  
Естественно, что после оценки изменений, например, командой git diff, из надо  
создать коммит слияния с основной:  
  
git merge username-project/master  
  
Команда git pull сразу забирает изменения и проводит слияние с активной веткой:  
  
git pull — забрать из репозитария, для которого были созданы удаленные ветки по  
умолчанию.  
  
git pull username-project — забрать изменения из определенного репозитария.  
  
  
Как правило, используется сразу команда git pull.

3.4 *git push* — вносим изменения в удаленный репозитарий (удаленную ветку)

После проведения работы в экспериментальной ветке, слияния с основной,  
необходимо обновить удаленный репозитарий (удаленную ветку). Для этого  
используется команда git push:  
  
git push — отправить свои изменения в удаленную ветку, созданную при  
клонировании по умолчанию.  
  
git push ssh://yourserver.com/~you/proj.git master:experimental — отправить изменения  
из ветки master в ветку experimental удаленного репозитария.  
  
git push origin :experimental — в удаленном репозитарии origin удалить ветку experimental.  
  
git push origin master:master — в удаленную ветку master репозитария origin (синоним  
репозитария по умолчанию) ветки локальной ветки master.

4 git-о-день

В этом разделе будут показаны и разобраны подробно несколько обычных и чуть  
меньше необычных для работы с git ситуаций.

4.1 Обычный workflow при работе с локальным репозитарием

Git обладает необычайной легкостью в использовании не только как распределенная  
система контроля версий, но и в работе с локальными проектами. Давайте разберем  
обычный цикл — начиная с создания репозитария — работы разработчика git над  
собственным персональным проектом:

1. mkdir git-demo
2. cd git-demo
3. git init
4. git add.
5. git commit -m «initial commit»
6. git branch new-feature
7. git checkout new-feature
8. git add.
9. git commit -m «Done with the new feature»
10. git checkout master
11. git diff HEAD new-feature
12. git merge new-feature
13. git branch -d new-feature
14. git log --since=«1 day»

Разберем каждое из действий. 1-2 — просто создаем рабочую директорию  
проекта. 3 — создаем репозитарий в директории. 4 — индексируем все существующие  
файлы проекта (если, конечно, они вообще были). 5 — создаем инициализирующий  
коммит. 6 — новая ветка, 7 — переключение в нее (можно сделать в один шаг  
командой git checkout -b new-feature). Далее, после непосредственной работы с  
кодом, индексируем внесенные изменения(8), совершаем коммит(9). Переключаемся в  
основную ветку(10), смотрим отличия между последним коммитом активной ветки и  
последним коммитом экспериментальной (11). Проводим слияние (12) и, если не было  
никаких конфликтов, удаляем ненужную больше ветку (13). Ну и на всякий случай  
оценим проведенную за последний день работу (14).  
  
Почему именно так? Зачем отказываться от линейной модели? Хотя бы даже потому,  
что у программиста появляется дополнительная гибкость: он может переключаться  
между задачами (ветками); под рукой всегда остается «чистовик» — ветка  
master; коммиты становятся мельче и точнее.

4.2 Workflow при работе с удаленным репозитарием

Предположим, что вы и несколько ваших напарников создали общественный  
репозитарий, чтобы заняться неким общим проектом. Как выглядит самая  
распространенная для git модель общей работы?

1. git clone <http://yourserver.com/~you/proj.git>  
   … возможно, прошло некоторое время.
2. git pull
3. git diff HEAD^
4. git checkout -b bad-feature  
   … работаем некоторое время.
5. git commit -a -m «Created a bad feature»
6. git checkout master
7. git pull
8. git merge bad-feature
9. git commit -a
10. git diff HEAD^  
    … запускаем тесты проекта, обнаруживаем, что где-то произошла ошибка. Упс.
11. git reset --hard ORIG\_HEAD
12. git checkout bad-feature  
    … исправляем ошибку.
13. git -m bad-feature good-feature
14. git commit -a -m «Better feature»
15. git checkout master
16. git pull
17. git merge good-feature
18. git push
19. git branch -d good-feature

Итак, первым делом создаем (1) создаем копию удаленного репозитария (по  
умолчанию команды вроде git pull и git push будут работать с ним). «Вытягиваем»  
последние обновления (2); смотрим, что же изменилось(3); создаем новую ветвь и  
переключаемся в нее (4); индексируем все изменения и одновременно создаем из них  
коммит (5); переключаемся в главную ветвь (6), обновляем ее (7); проводим  
слияние с веткой bad-feature(8) и, обнаружив и разрешив конфликт, делаем коммит  
слияния (9).  
  
После совершения коммита отслеживаем изменения(10), запускаем, например,  
юнит-тесты и с ужасом обнаруживаем, что после слияния проект валится на большей  
части тестов.   
  
В принципе, тесты можно было прогнать и до коммита, в момент  
слияния (между пунктами 8 и 9); тогда бы хватило «мягкого» резета.  
  
Таким образом, приходится совершить «жесткий» (11) сброс произошедшего слияния,  
ветки вернулись в исходное до состояние. После чего переключаемся в неудачную  
ветку (12), вносим необходимые изменения и переименовываем ветку (13). Совершаем  
коммит (14); переходим в главную ветку(15), опять ее обновляем (16). На этот раз  
бесконфликтно делаем слияние (17), закидываем изменения в удаленный репозитарий  
(18) и удаляем ненужную теперь ветку (19). Закрываем ноутбук, одеваемся и идем  
домой под утро.