IBM

developerWorks Россия Технические материалы Linux Статьи

Распределенная система управления версиями Git: Часть 3. Философия Git

Продолжая разговор о философии Git, её архитектуре и идеологии, мы рассмотрим ряд важных команд и закрепим изученный материал на конкретных примерах.

Свободный журналист, разработчик ПО. С 2006 года является студентом университета Информационных Технологий Механики и Оптики кафедры Компьютерных технологий (СПбГУ ИТМО, КТ).

24.03.2011

1. Введение

В первой и второй статьях цикла "Распределенная система управления версиями Git" мы выяснили, что из себя представляют СУВ и чем они отличаются. Также мы рассмотрели основные команды и ключи к ним, основательно разобрали архитектуру системы контроля версий Git и уделили внимание ее философии.

Продолжая разговор о философии Git, её архитектуре и идеологии, мы рассмотрим ряд важных команд и закрепим изученный материал на конкретных примерах, а также попытаемся найти ответ на вопрос: "почему именно так устроен Git?".

2. Философия Git

Система управления версиями Git отталкивается в своем поведении от двух постулатов:

1. "Ни один момент истории отменить нельзя, однако в будущем можно сделать такое изменение, которое исправит необходимые данные".
2. "Распределенное поведение — самое оптимальное поведение".

Прежде чем понять как менять прошлое, стоит задуматься над вопросом — "а что же тогда делать с настоящим?", которое основано на прошлом. Если мы что-то сломали, то кроме как заново построить и склеить, мы ничего сделать с этим не сможем. Такова политика системы Git. Данный подход, наверное, единственный, не содержащий противоречий.

Под распределенным поведением понимается разумное разделение разрабатываемого проекта на ветви (или на репозитории). Такое деление предполагает возможность раздельной разработки каждой из частей и возможность собрать в нужный момент воедино работоспособный релиз проекта.

Попробуем определить, к чему приводит подобный подход.

3. Идеология и Архитектура Git

Журналирование

В паре с графическими утилитами (например, "git gui") команда "git log" вполне наглядно отражает картину истории проекта, помогает найти необходимые развилки, ключевые моменты иерархии версий. Любой объемный проект требует хранения истории разработки программного продукта. *Git log* может заменить историю логов проекта лишь упрощенно, но имеющихся инструментов, зачастую, достаточно.

"git log" - выводит список описаний коммитов(без описания содержимого), созданных в данном

репозитории в обратном хронологическом порядке:

"-p" — (от англ. *patch*) ключ позволяет добавить разницу между коммитами к описанию.

"--stat" – ключ позволяет добавить количественную информацию об изменении файлов.

"--shortstat" – ключ позволяет выводить только *измененные/добавленные/удаленные* строки формата вывода *--stat*.

"--name-only" - ключ позволяет выводить список измененных файлов каждого коммита.

"--name-status" — ключ позволяет выводить список файлов с информацией *добавлен/изменен/удален*.

"--abbrev-commit" — ключ позволяет выводить короткую форму *SHA-1* хеша коммита.

"--relative-date" — ключ позволяет выводить даты относительно текущего времени("час назад").

"--graph" — ключ позволяет вывести *граф* истории коммитов в текстовом виде.

"--pretty=<переменная>" – ключ выступает в роли переменной, значения "oneline", "short", "full" ,"fuller" и "format" которой позволяют изменять формат вывода информации о коммите.

Значение ключа *format* позволяет задать свой собственный формат вывода вида:

```
$ git log –pretty=format:<my-format>
```

Где "my-format" является строкой, содержащей маску информации о соответствующем коммите. В качестве переменных маски можно выбрать следующие значения:

Параметр	Значение в маске
%H	Хэш коммита
%h	Сокращенный хэш коммита
%T	Хэш дерева
%t	Сокращенных хэш дерева
%P	Хэши родительских коммитов
%p	Сокращенные хэши родительских коммитов
%an	Имя автора
%ae	Электронная почта автора
%ad	Дата автора
%ar	Дата автора, относительная
%cn	Имя коммиттера
%ce	Электронная почта коммиттера
%cd	Дата коммиттера
%cr	Дата коммиттера, относительная
%s	Комментарий

Также ключами можно задать фильтр вывода информации:

"-<n>" - ключ позволяет отображать последние n коммитов.

"--grep=<регулярное_выражение>" - ключ позволяет выводить информацию о всех коммитах, совпадающих с указанным регулярным выражением.

"--since=<время>" или "--after=<время>" – ключи позволяют задать начало указываемого

промежутка времени вывода информации.

"--until=<время>" или "--before=<время>" – ключи позволяют задать конец указываемого промежутка времени вывода информации.
Досадное ограничение: данными ключами нельзя указывать несколько промежутков времени.

"--author=<автор>" – ключ позволяет выводить сообщения указанного автора.

"--committer=<коммитер>" – ключ позволяет выводить сообщения указанного коммитера.

Замечательная сторона в том, что все ключи могут сочетаться, и почти все из них сочетаются разумно (т.е. сочетать их имеет смысл).

Работа с удаленными репозиториями

Начало работы с новым проектом почти всегда требует выполнения следующей команды. Вы входите в систему и создаете локальную копию разрабатываемого проекта:

"git remote" – команда выводит имена всех зарегистрированных в системе удаленных репозиториев.

"-v" или "--verbose" – ключ добавляет к выводу информации об удаленных репозиториях *url*-путь.

"add <имя_репозитория> <url-путь_репозитория>" — ключ позволяет добавить удаленный репозиторий.

"rename <старое_имя_репозитория> <новое-имя_репозитория>" — ключ позволяет переименовать удаленный репозиторий.

"show <имя_репозитория>" — ключ позволяет вывести информацию о репозитории.

"update <имя-репозитория>" — ключ позволяет обновить ветви указанного удаленного репозитория.

Каждый день разработчика должен начинаться с выполнения этих двух команд -"*git fetch*" и "*git pull*", возможно, не единожды в день. Войдя в систему, Вы должны быть уверены, что работаете над последней версией проекта. Предположим обратное. Вы не обновились в нужный момент и одна из частей программы изменилась(был исправлен старый *баг*, добавилась новая функциональность). Ваши изменения вашей же ветви никак не могут навредить разработке. Но вы, к сожалению, потеряете время, "изобретая колесо", т.е. делая либо то, чего уже не требуется, либо то, что за вас уже кто-то сделал:

"git fetch <удаленный_репозиторий>" – команда обновляет информацию об указанном удаленном репозитории (но не копирует содержимое). Для того, чтобы получить локальную копию обновленных данных, необходимо слить нужную ветвь удаленного репозитория с текущей рабочей ветвью.

"git pull <удаленный_репозиторий> <удаленная_ветвь>" – команда обновляет указанную ветвь удаленного репозитория, одновременно сливая её с текущей ветвью.

Каждый день разработчика должен заканчиваться выполнением этой команды - "git push", возможно, не единожды в день. Аналогично логике предыдущей команды, выполнение этой - поднимает мобильность разработки проекта. Она нужна для заливки своих версий данных в сторонние ветви:

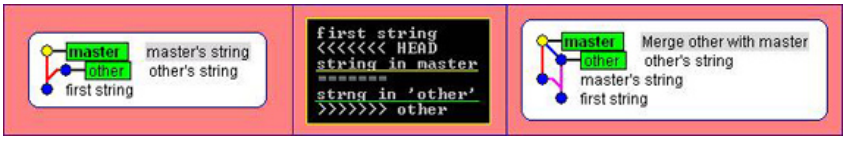
"git push <удаленный_репозиторий> <локальная_ветвь>:<удаленная_ветвь>" – команда отсылает информацию о локальной ветви в указанную ветвь удаленного репозитория.

Ветвление

"git merge <ветвь>" – команда сливает историю указанной и текущей ветви.

"git merge <имя_удаленного_репозитория>/<ветвь_удаленного_репозитория>" - команда сливает историю указанной ветви удаленного репозитория и текущей ветви.

Слияние ветвей означает слияние коммитов, помеченных как *HEAD* (т.е. последних коммитов соответствующей ветви). Заметим один факт. Из предыдущих статей мы знаем, что ведение истории проекта может отличаться от истории в реальном времени (например, добавив новый коммит в давно забытую ветвь и слить её с самой новой). В таком случае может возникнуть конфликт слияний. Например, когда между одинаковыми строками в разных коммитах стоят раличные измененные данные. Система не в силах сама разрешить такой конфликт. Но процесс слияния на этом не останавливается, а такие неоднозначности, которые необходимо будет разрешить вручную, просто помечаются некоторым образом.



Слева на рисунке общий вид истории ветвей до слияния, в середине — получившаяся неоднозначность, справа — после слияния.

"git merge" – одна из самых часто используемых операций в Git. Рассмотрим, в какие моменты её разумно применять:

Структура разработки программного обеспечения во многих компаниях построена на некотором делении разрабатываемого продукта на части, каждая из которых реализует свою задачу. И над каждым разделом трудится своя команда разработчиков. По мере того как та или иная часть подходит к логическому завершению, возникает необходимость соединить их воедино.

Кроме функционального деления разрабатываемого продукта, существует необходимость в разделении проекта по методологическим принципам. Таким как "Debug"(отладка и поиск багов), тестирование приложения, экспериментальное деление(появляется новая идея, необходимо её проверить).

Рассмотрим использование ветвлений на примере:

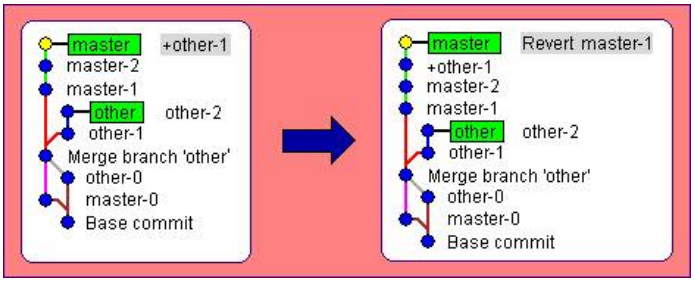
Предположим, нужно решить арифметическую задачу. Как бы Вы стали это делать? Вы смотрите на условие задачи и пытаетесь найти подходящие и заведомо доказанные формулы. Но в формуле в качестве аргумента может быть также формула и т.д.. Для того, чтобы посчитать её, нужно знать значение аргумента. Приходим к тому, что нужно дойти до самой последней формулы, в которой в качестве параметров будут константы. А после — возвращаемся обратно, попутно разрешая текущие формулы. В результате, решение задачи будет эквивалентно обходу всех вершин в структуре данных *дерево* методом DFS(поиск в глубину).

Аналогичным образом обстоит ситуация с версиями файлов в Git. В истории важны именно те моменты, в которых данные будут иметь определенные свойства. И для того, чтобы не потерять эти свойства, мы сохраняем определенный снимок файлов. Если вдруг наталкиваемся на объемную и сложную работу, то нам нужно разветвить историю в этот момент (аналогично подсчету формулы в формуле, с той, лишь, разницей, что мы не будем возвращаться тем же путем после проделанной работы назад). Зачем? Во-первых, над создаваемым проектом вы работаете, наверняка, не в одиночку. И пока вы будете изменять свою часть кода, остальные смогут работать над другими. Во-вторых, работа с программным кодом — это сплошной эксперимент и неизвестно, какой код в итоге окажется лучше. В данном случае, попав в тупик, можно просто вернуться к развилке и начать заново, а можно и вовсе удалить ветвь, если идея оказалась неудачной (при этом вы ничего не потеряете и не испортите в основной ветви проекта). Если задумка удалась и вы корректно закончили локальную работу, необходимо вернуться назад (слить законченную работу с основой), т.е. к той ветви из которой производилось ветвление. Обобщая сказанное, никто не мешает вам от второстепенной ветви, выделенной под конкретные действия, ответвить ещё одну, для которой первая локальная будет основной. Таким образом, рекурсивно мы будем ветвить историю до тех пор, пока вся работа не будет выполнена. С точки зрения алгоритмики такой иерархии версий соответствует структура данных *сеть*.

Исправление истории

"git revert <коммит>" — команда создает коммит, отменяющий изменения указанного коммита. Единственное ограничение - чтобы все файлы ветвей проекта были в состоянии *неизменен*.

Очевидно, если кто-то во время работы случайно или ошибочно совершил какие-то изменеия, то их можно исправить данной командой — *git revert*.



Слева на рисунке показана история двух ветвей "*other*" и "*master*" до выполнения команды "git revert <имя_коммита>". Справа показана история ветвей после успешного выполнения команды.

"git stash" – команда создает временную (*stash*) ветвь, куда переносит все изменения данных текущей ветви (файлы имеющие состояние отличное от *неизменен*).

"git stash apply" — команда создает новый коммит в текущей ветви на основе созданной ранее *stash*-ветви.

"Stash" – от англ. "прятать". Предположим, Вы пришли на работу и начачали писать проект. Но

вдруг, под конец рабочего дня, обнаруживается, что все изменения делались не в нужной ветви СУВ. Тогда Вы можете использовать команду *git stash*, переключиться на нужную ветвь и применить все те же сделанные за день изменения в неё.

Корректировка истории

"git rebase <имя_патч-ветви> <имя_ветви_слияния>" — команда создает последовательный набор *патч*-коммитов, включающий в себя все изменения указанной *патч*-ветви, вплоть до пересечения её истории с ветвью слияния, в которую мы вливаем *патч*-коммиты.

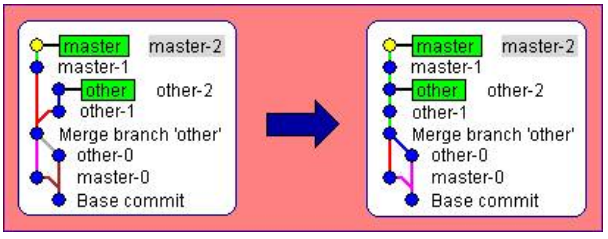
"git rebase <имя_патч-ветви>" — команда создает последовательный набор *патч*-коммитов, включающий в себя все изменения указанной *патч*-ветви, вплоть до пересечения её истории с текущей ветвью, в которую мы вливаем *патч*-коммиты.

"--continue" – ключ позволяет продолжить работу команды после её остановки, вызванной конфликтом слияний.

"--skip" – ключ позволяет продолжить работу команды после её остановки, вызванной конфликтом слияний, пропустив при этом слияние текущего коммита.

"--abort" – ключ позволяет остановить выполение и отменить все изменения, сделанные командой *git rebase*.

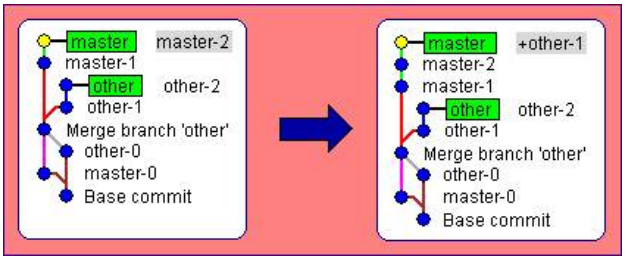
Предположим, в процессе нашей работы некоторая команда разработчиков отделилась от основной ветви, с целью, скажем, исправления ошибок предыдущего кода. Иногда этот процесс затягивается, и за то время, пока одни люди корректировали код, основная команда разработала новую функциональность, которая завязана на предыдущей работе. В такой ситуации очень просто прийти к релизу текущей версии проекта - при помощи команды *git rebase*. Ее использование повышает мобильность проекта.



Слева на рисунке показана история двух ветвей "*other*" и "*master*" до выполнения команды "git rebase other master". Справа показана история ветвей после успешного выполнения команды.

"git cherry-pick <коммит>" — команда создает новый коммит в текущей ветви, накладывающий изменения указанного коммита.

Данная команда и *git rebase* выполняют схожие действия. Отличие их в том, что git cherry-pick оперирует одним коммитом, накладывая при этом изменения на вершину стека коммитов ветви.



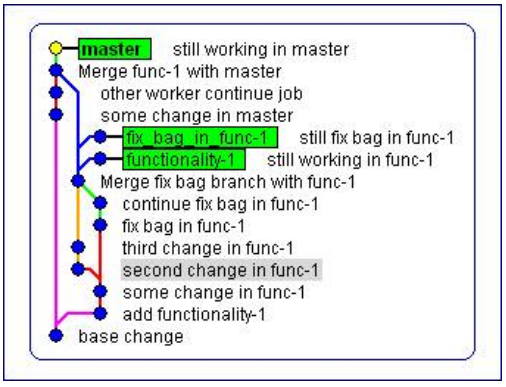
Слева на рисунке показана история двух ветвей "other" и "master" до выполнения команды "git cherry-pick <имя_коммита_other-1>". Справа показана история ветвей после успешного выполнения команды.

4. Философия на практике

Опишем простой пример разработки при помощи Git:

```
git init                                    #создаем репозиторий
vim file                                    #создаем файл, который будет составлять суть
проекта
git add file                                #индексируем файлы
git commit -a -m 'first working version'    #создаем первую рабочую версию проекта
vim file                                    #изменяем небольшие огрехи работы
git commit -a -m 'second working version'   #создаем вторую рабочую версию проекта
git checkout -b functionality               #создаем отдельную ветвь для реализации
                                            #некоторой добавляемой функциональности
                                            #программы и переходим в неё (ветвь)
vim file                                    #делаем первые попытки создать функциональность
git commit -a -m 'add func-1'               #сохранение новой функциональности
vim file                                    #осуществляем незначительные изменеия с
                                            #файлами функциональности
git commit -a -m 'to do func-1 in order'    #применение изменений новой функциональности
git checkout master                         #переходим в основную ветвь
git merge functionality                     #создаем релиз программы с новой
                                            #функциональностью
git checkout functionality                  #переходим в ветвь работы с функциональностью
git checkout -b fix_bag_in_func'            #создаем новую ветвь для исправления
                                            #ошибок функциональности и переключаемся на
                                            #неё (ветвь)
vim file                                    #исправляем ошибку
git commit -a -m 'Correct bag №1 in func'   #закрепляем в истории исправления
git merge  functionality                    #вливаем ветвь функциональности в ветвь
                                            #исправления ошибок
git checkout functionality                  #переходим в ветвь функциональности.
git merge  fix_bag_in_func                  #вливаем ветвь исправления ошибок в ветвь
                                            #функциональности
git checkout master                         #переходим в основную ветвь
git merge functionality                     #вливаем ветвь функциональности в основную ветвь
vim file                                    # делаем необходимые исправления в ветви master
git commit -a -m 'still working in master'#продолжаем работу с ветвью
vim file                                    #делаем необходимые исправления в ветви
                                            #functionality
git commit -a -m 'still working in func'    #продолжаем работу с ветвью
vim file                                    #делаем необходимые исправления в ветви
                                            #fix_bag_in_func
git commit -a -m 'still working in fix_bag_in_func' #продолжаем работу с ветвью
```

Заметим, что работа в каждой созданной нами ветви может не прекращаться даже после достижения результата и слияния с основой. Можно улучшать проект бесконечно.



На рисунке показан приближенный вид получившейся истории версий проекта.

5. Заключение

Подведем промежуточные итоги. Мы узнали основную часть команд и их ключей системы управления версиями Git. Познакомились с методологией ведения истории и основательно разобрали архитектуру, также подчеркнули суть принципа разработки на примере.

Следующая статья цикла будет посвящена взаимодействию Git с её аналогами (такими, как SVN, …), также мы продолжим изучение архитектуры системы.

Авторы и их статьи
Статьи на русском языке специалистов в сфере ИТ технологий, сотрудничающих с IBM developerWorks.

Облачные вычисления
Статьи, руководства, обзоры

для ИТ специалистов.

Библиотека документов
Более трех тысяч статей, обзоров, руководств и других полезных материалов.