IBM

developerWorks Россия Технические материалы Linux Статьи

Распределенная система управления версиями Git: Часть 2. Структура и философия Git

Во второй части цикла мы подробнее коснемся архитектуры и философии Git, а также рассмотрим базовый пример работы с данной СУВ.

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

23.09.2010

1. Введение

Продолжаем цикл "Распределенная система управления версиями Git". В предыдущей статье мы дали определение систем управления версиями, привели основные особенности и возможности GIT и рассмотрели базовую архитектуру. Во второй части подробнее коснемся архитектуры и философии Git, а также рассмотрим базовый пример работы с данной СУВ.

В следующих статьях будут рассмотрены особенности и тонкости ведения истории и взаимодействие с другими СУВ.

2. Философия и Архитектура Git

Настройка

Позволим себе пропустить тривиальный шаг установки Git. Рассмотрим нюансы, связанные с работой в терминале Git:

Любая команда в консоли/терминале записывается как ' git <команда> '.

Для вызова справки о команде и её ключах необходимо дописать в начало или конец вызова команды следующий префикс/суффикс: ' man git <команда> ' или ' git <команда> --help '.

Первое, что нам нужно сделать с репозиторием, — это правильно настроить его; команда, которая позволяет сделать это — ' config '. Все настройки системы хранятся в соответствующих файлах, в зависимости от ключей команда config позволяет задать иерархию конфигураций и настроек системы для разных пользователей и репозиториев:

' --system <параметр_настройки> <значение> ' — ключ задает конфигурацию для всех репозиториев и пользователей системы.

' --global <параметр_настройки> <значение> ' — ключ задает настройки конкретного пользователя.

' -f или --file <параметр_настройки> <значение> ' — ключ задает настройки конкретного репозитория.

Иерархия использования такова, что каждая последующая конфигурация заменяет предыдущую. Так, если вы пользуетесь системой на чужом компьютере и хотите подправить её под себя для удобства, вы можете задать конфигурацию пользователя, которая заменит глобальные настройки.

' -l ', ' --list ' — ключ позволяет просмотреть существующие настройки системы.

Репозиторий

Приступим к созданию истории наших файлов. Для этого необходимо получить уже существующий

репозиторий или создать новый, используя несколько команд:

' git init ' — создает пустой шаблон-каталог ' .git ' в текущей директории, содержащий необходимые конфигурационные данные для текущего репозитория.

' git clone <url> ' — создает каталог с именем удаленного репозитория в текущей директории, в которой инициализирует репозиторий ' .git '. Система копирует удаленный репозиторий из сети и забирает последний снимок истории репозитория.

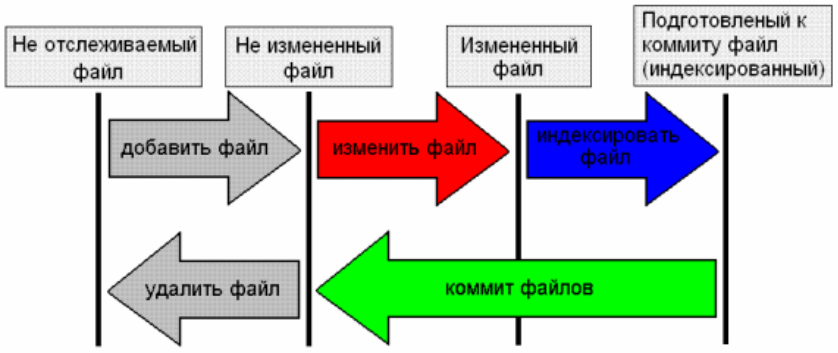
Удаленная ветка (remote branch) — это неизменяемая ветвь, которая имеет опрееленное имя — ' <имя удалённого репозитория>/<имя ветви> '. Для работы с удаленной веткой необходимо получить её локальную копию.

Удаленный репозиторий (remote repository) — это тип данных системы, который содержит сетевой путь к соответствующему репозиторию и информацию о содержащихся в нем ветвях — все удаленные ветви репозитория.

Представление места хранения информации об истории наших файлов неоднознщачно, для операционной системы это простой каталог, а для системы контроля версий это репозиторий. Каждый файл в репозитории может находиться в двух состояниях: под контролем и версий без него. В свою очередь, любой файл, контролируемый системой, может находиться в одном из трех состояний: не измененный, подготовленный к изменению (коммиту) и измененный.

Ситуации с изменениями состояний могут быть следующие: вы добавляете файл в ваш рабочий каталог, говорите системе, чтобы она добавила его под версионный контроль, по умолчанию ему будет присвоен статус неизменен. Затем работаете с файлом (изменяете его), вследствие чего он переходит в состояние изменен. Чтобы получить снимок файла в определенный момент времени, предварительно необходимо перевести файл в состояние подготовленный для коммита. При этом версия файла индексируется (запоминается текущее состояние), причем индекс для файла всегда один. Когда вы захотите зафиксировать историю файлов, Git сохранит именно те версии данных, которые будут проиндексированы.

Картина изменения состояний.



Формирование репозитория

Для того, чтобы управлять, просматривать содержимое репозитория, предусмотрен ряд разносторонних команд: ' add ', ' rm ', ' status ', …, которые в свою очередь имеют ряд ключей ' -f ', ' -a ', ' -u ', ... :

' git add ' <имя_файла> или <имя_директории> — команда переводит файл или каталог под версионный контроль в состояние подготовленный к коммиту.

' git rm --cached <имя_файла> или <имя_директории> ' — команда удаляет файл/каталог из-под контроля системы.

' -f ' или ' --force ' <имя_файла> или <имя_директории> — принудительлно удаляет файл/каталог

из файловой системы. Система сознательно требует ключи в качестве подтверждения, чтобы предостеречь пользователя от случайной потери информации.

' git status ' — команда позволяет просматривать состояния и наличие файлов текущей ветви.

' -u ' — включает информацию о наличии файлов репозитория, не находящихся под версионым контролем.

Манипуляции с историей

Коснемся задачи формирования содержимого нашего репозитория. В прошлом номере мы описали термин "Ветка" — ссылка на последний коммит в истории. Таких веток в истории проекта может быть несколько. Но, с другой стороны, каждая ветвь с помощью указателей объединяет в себе цепь из коммитов. Взаимное расположение данных цепей, как раз, и отражает картину проекта в зависимости от времени. На практике, в системе поддерживается ряд специальных ссылок для упрощения обращения к моментам истории. Одной из таких ссылок является ' HEAD ', она указывает на последний коммит текущего ветвления истории.

Ветвь, в смысле пути истории, достаточно закрытый абстрактный элемент, и с применением лишь тегов или хеш-имени достаточно неудобно оперировать коммитами цепи. Для того, чтобы развязать пользователю руки, в систему ввели механизм относительной адресации, позволяющий обращаться к коммитам данной ветви. Следующие правила предусматривают использование механизма:

' <ссылка_на_коммит>^ ' — ссылка на родителя текущего коммита.

' <ссылка_на_коммит>~x ' — ссылка на x-ого по порядку родителя коммита.

Также вспомним, что ссылки в Git могут использоваться как синонимы имен непосредственно.

Рассмотрим ряд команд формирования истории:

' git branch ' — команда показывает все существующие ветви и указывает текущую из них.

' git branch <имя_ветви> ' — создает новую ветвь.

' -d <имя_ветви> ' — удаляет указанную ветвь, если в ней нет неразрешенных конфликтов.

' -D <имя_ветви> ' — принудительно удаляет указанную ветвь.

' -m <имя_ветви> ' — переименовывает указанную ветвь.

' --contains <ссылка_на_коммит> ' — покажет те ветви, среди предков которых есть указанный коммит.

По умолчанию при инициализации репозитория создается первая ветка с именем master, которую нужно сделать основной. Оптимальный процесс ведения истории заключается в создании на некотором этапе истории дополнительных веток с целью решения необходимых задач именно там. После завершения локальных проблем и получения результатов, можно вернуться на некоторое время в основу и попытаться взглянуть вцелом на все модификации, проверить их работоспособность. После чего стоит вновь перенести локальные задачи на разные ветви. При этом, проект будет древовидным, и никто не мешает некоторой ветке создать своих потомков для решения уже своих локальных задач.

' git checkout <имя_файла> или <имя_директории> ' — команда переводит данный файл или файлы директории в состояние, соответствующее последнему коммиту.

' git checkout <ссылка_на_коммит> <имя_файла> или <имя_директории> ' — команда переводит

данный файл или файлы директории в состояние, соответствующее указанномму коммиту.

' git checkout <имя_ветви> ' — команда переключает систему на указанную ветвь, единственное ограничение — рабочие версии файлов ветви перед переключением не должны отличаться от версий последнего коммита, т.е. быть измененными.

' -f <имя_ветви> ' — переключает систему на указанную ветвь, игнорируя ограничения.

' -b <имя_ветви> ' — создает и переключает систему на указанную новую ветвь.

' git commit ' — команда создает слепок (коммит) индексированных файлов, к которому предлагается написать описание. Для его записи используется текстовый редактор, указанный в настройках системы (config).

' git commit <имя_файла> ' — команда индексирует указанный файл и создает новый коммит на его основе.

' -m "<текст_описания>"' — создает коммит индексированных файлов, прилагая к нему текст описания в качестве аргумента.

' -a ' — добавляет все измененные и проиндексированные файлы текущей ветви в новый коммит. Равносильно двум командам — ' git add <рабочая_директория> + git commit '.

' git reset ' — команда убирает индексацию со всех файлов текущей ветви.

' git reset <имя_файла> или <имя_директории> ' — команда рекурсивно удаляет индексацию всех файлов директории указанного файла.

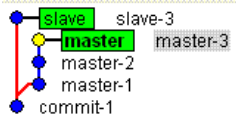
' --soft <ссылка_на_коммит> ' — команда позволяет осуществить так называемый "мягкий" откат, т.е. возврат ветви в указанный коммит, при этом сохраняются версии файлов до перехода.

' --hard <ссылка_на_коммит> ' — команда позволяет осуществить "жесткий" откат, т.е. возврат ветви в указанный коммит, но при этом теряются все текущие версии файлов. Можно сказать, что процедура возвращает вас во время и к версиям файлов, когда вы совершили указанный коммит. Откат произойдет с одной оговоркой, если стираемый момент истории не содержит слияния ветвей.

На практике, Git предоставляет механизм перехода к исходному слепку файлов после любого отката. Ссылка ' ORIG_HEAD ' указывает на тот коммит, откуда была вызвана команда reset --soft или --head. В случае ошибочных действий можно по ссылке перейти к последним по времени файлам.

Проект может начинать развитие с нескольких веток параллельно, по ходу дела они могут переплетаться, и на выходе снова могут быть несколько ветвей. Команды git commit, git reset с одной стороны и git checkout, git branch — с другой, отвечают за вертикальные и горизонтальные действия, т.е. выполняют операции/переходы между коммитами и ветками соответственно.

Обычная история проекта в Git.



Фильтрация файлов

По мере работы с системой, например, ведения программного продукта, в репозитории могут появляться ненужные для истории файлы (логи или временные файлы компиляции). Чтобы исключить их из-под версионного контроля, предусмотрена специальная схема:

Необходимо создать файл '.gitignore' в директории репозитория. Также как и с конфигурационными данными, файл будет отвечать за поддерево своего каталога, и он может быть замещен лежащим ниже .gitignore. Содержимым должен быть ряд регулярных выражений таких, что любое имя файла, ненужного для истории вашего проекта, подходило хотя бы под один из шаблонов. При чтении Git пользуется упрощенными регулярными выражениями по следующим правилам:

Пустые строки, а также строки, начинающиеся с символа ' # ', не просматриваются.

Можно инвертировать значение шаблона, использовав знак ' ! ' в начале выражения.

Синтаксис регулярных выражений:

' * ' — шаблон соответствует любому количеству любых символов. В том числе и нуль-символу(пустой символ). (c*a = caba, t*p = tp)

' ? ' — шаблон соответствует одному любому символу, но не нуль символу. (t?p = top, t?p <> tp)

' [ekh] ' — шаблон соответствует любому из перечисленных в скобках символу.

' [a-f] ' — шаблон соответствует любому символу из промежутка в скобках.

3. Первый пример использования.

```
cd ~/my_folder
mkdir my_project
cd my_project
git init
#Создаем директорию под проект, переходим в неё и инициализируем репозиторий
git config user.name Hamitov Roman
git config user.email tm-green@yandex.ru
#Настраиваем конфигурационные данные репозитория под себя
#vim – это простой текстовый редактор
vim .gitignore
readme
.git*
*.o
*~
*#
#Настраиваем файл игнорирования
vim first_step
```

По-моему, Git очень сложный

```
git status -u
git add .
git commit -m 'Моё первое мнение'
#Создаем новый файл, переводим его под версионный контроль
#индексируем, и затем создаем первый коммит
vim del_file
```

этот файл мы удалим

```
git add del_file
#В качестве примера создадим ненужный файл
vim test
```

Команды, с которыми мы познакомились:

```
init, add, commit
git add .
git rm -f del_file
#полностью удаляем ненужный файл
git commit -m 'Первое знакомство'
#Создали коммит достоверных и проверенных данных
git branch study_commands
git checkout study_commands
#Создаем ветку study_commands и переключаемся на неё; в этом появилась необходимость,
#т.к. мы хотим проверить нетривиальные, сложные команды
vim test
```

Команды, с которыми мы познакомились:

```
init, add, commit, rm, test, branch, add, checkout
#Сделали ошибку: add – написали два раза
git commit -a -m 'Наши знания растут'
#Записали все команды, которые мы на данный момент знаем
git reset --soft HEAD^
vim test
```

Команды, с которыми мы познакомились:

```
init, add, commit, rm, test, branch, checkout
```

```
git commit -a -m 'Исправили повторение ' add ' '
#коммит
#но test не является командой, это файл
vim test
```

Команды, с которыми мы познакомились:

```
init, add, commit, rm, branch, checkout
git commit -a -m 'Команды, которые мы точно знаем!!!'
#исправили test
git checkout master
vim first_step
```

Оказалось нетрудно.

```
git commit -a -m 'final'
#переходим обратно в ветку master и делаем последний завершающий коммит
```

4. Заключение

В следующей статье мы продолжим изучать философию системы, разберем практическую связь между командами, их ключами и ведением истории. Вопрос оптимальной настройки системы в зависимости от стиля разработки продукта остается открытым. Мы рассмотрим его решение для разных ситуаций.