IBM

developerWorks Россия Технические материалы Linux Статьи

Распределенная система управления версиями Git. Часть 1: Введение

В этой статье в общих чертах, разобранны характеристики систем контроля, рассказано об их архитектуре и основных особенностях рассматриваемого приложения. Кроме того, произведен обзор ныне существующих интерфейсов для работы с Git.

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

26.01.2010

1. Введение

Во время работы над проектом его участники часто сталкиваются с проблемами синхронизации и ведения истории файлов, решить которые помогают системы управления версиями (СУВ). Цель этой серии статей – познакомить читателя с принципами работы СУВ и подробно рассмотреть одну из них, а именно Git. Почему Git? В последнее время эта система набирает популярность, и ее важность для свободного ПО (и для проекта GNU/Linux, в частности) сложно переоценить.

Мы последовательно, в общих чертах, разберем характеристики систем контроля, расскажем об их архитектуре и основных особенностях рассматриваемого приложения. Кроме того, сделаем обзор ныне существующих интерфейсов для работы с Git.

Автор сознательно опускает терминологию функций, ключей и прочих тонкостей, чтобы четко, ясно и в общем виде представить вам картину. Данная статья предполагает, что читатель знаком с Unix-подобными операционными системами (ОС), а также имеет базовые знания в области алгоритмики и информатики в целом.

В следующих материалах мы углубимся в структуру и философию Git, специфику этой системы и тонкости практической работы с ней. Завершит цикл статья о взаимодействии Git с другими СУВ (такими как Subversion, CVS, Mercurial и др.).

2. Git – это ...

Git – это распределённая система управления версиями файлов. Код программы написан в основном на языке C. Проект был создан Линусом Торвальдсом в 2005 году для управления разработкой ядра Linux и, как и GNU/Linux, является свободным программным обеспечением (ПО), при этом стороннее использование подчиняется лицензии GNU GPL версии 2. Вкратце данное соглашение можно охарактеризовать как ПО со свободным кодом, которое должно развиваться открыто, т.е. любой программист вправе продолжить совершенствование проекта на любом его этапе. За свое недолгое время существования данная система была введена многими ведущими разработчиками. Git используется в таких известных Linux-сообществу проектах, как Gnome, GNU Core Utilities, VLC, Cairo, Perl, Chromium, Wine.

3. Системы управления версиями

Системы управления версиями (Version Control Systems) – это программное обеспечение, призванное автоматизировать работу с историей файла (или группы файлов), обеспечить мониторинг изменений, синхронизацию данных и организовать защищенное хранилище проекта.

Короче говоря, основная задача систем управления версиями – упростить работу с изменяющейся информацией. Разберем общий вид разработки на примере.

Предположим, есть некий проект, который вы разрабатываете, несколько отделов программистов и вы – координатор (или руководитель). По отношению к системе контроля, будь то сервер (если речь идет о централизованной системе) или локальная машина, любой разработчик проекта ограничен только правами доступа на изменение и/или чтение версий файлов данного хранилища. В любой момент вы можете сделать откат данных до необходимой вам версии. Вы, как координатор, можете ограничить доступ определенным пользователям на обновление версии файла. Также СУВ предоставляет интерфейс наблюдения и поиска версий файлов. Например, можно создать запрос: “Где и когда менялся данный кусок кода?”.

Система предполагает защищенное хранение данных, т.е. любой хранимый в ней блок имеет множество клонов. Так, например, при повреждении какого-либо файла вы своевременно можете заменить его копией. Для уменьшения объема данных проекта часто используется дельта-компрессия – такой вид хранения, при котором хранятся не сами версии файла, а только изменения между последовательными ревизиями.

4. Отличия распределённых систем управления версиями

Распределённые системы управления версиями – это СУВ, главной парадигмой которых является локализация данных каждого разработчика проекта. Иными словами, если в централизованных СУВ все действия, так или иначе, зависят от центрального объекта (сервер), то в распределенных СУВ каждый разработчик хранит собственную ветвь версий всего проекта. Удобство такой системы в том, что каждый разработчик имеет возможность вести работу независимо, время от времени обмениваясь промежуточными вариантами файлов с другими участниками проекта. Рассмотрим эту особенность, продолжая предыдущий пример.

У каждого разработчика на машине есть свой локальный репозиторий – место хранения версий файлов. Работа с данными проекта реализуется над вашим локальным репозиторием, и для этого необязательно поддерживать связь с остальными (пусть даже и главными) ветвями разработки. Связь с другими репозиториями понадобится лишь при изменении/чтении версий файлов других ветвей. При этом каждый участник проекта задает права собственного хранилища на чтение и запись. Таким образом, все ветви в распределенных СУВ равны между собой, и главную из них выделяет координатор. Отличие главной ветви лишь в том, что на неё мысленно будут равняться разработчики.

5. Основные возможности и особенности Git

Стоит сказать, что система если и не произвела фурор, то немного всколыхнула сообщество в области СУВ своей новизной и предложила новый путь развития. Git предоставляет гибкие и простые в использовании инструменты для ведения истории проекта.

Особенностью Git является то, что работа над версиями проекта может происходить не в хронологическом порядке. Разработка может вестись в нескольких параллельных ветвях, которые могут сливаться и разделяться в любой момент проектирования.

Git – довольно гибкая система, и область её применения ограничивается не только сферой разработки. Например, журналисты, авторы технической литературы, администраторы, преподаватели вузов вполне могут использовать её в своем роде деятельности. К таковым задачам можно отнести контроль версий какой-либо документации, доклада, домашних заданий.

Выделим основные отличия Git от других распределенных и централизованных СУВ.

Архитектура Git

SHA1 (Secure Hash Algorithm 1) – это алгоритм криптографического хеширования. Каждый файл вашего проекта в Git состоит из имени и содержания. Имя – это первые 20 байтов данных, оно наглядно записывается сорока символами в шестнадцатеричной системе счисления. Данный ключ получается хешированием содержимого файла. Так, например, сравнив два имени, мы можем почти со стопроцентной вероятностью сказать, что они имеют одинаковое содержание. Также, имена идентичных объектов в разных ветвях (репозиториях) – одинаковы, что позволяет напрямую оперировать данными. Хорошим дополнением сказанному выше служит ещё то, что хеш позволяет точно определить поврежденность файлов. Например, сравнив хеш содержимого с именем, мы можем вполне точно сказать, повреждены данные или нет. Далее под именем мы будем понимать имя файла, а строку символов будем называть SHA1-хешем.

Стоит упомянуть о так называемых коллизиях. “Вполне точно определить поврежденность” означает, что существуют такие файлы, различные по содержанию, SHA1-хеш которых совпадает. Вероятность таких коллизий очень мала, и по предварительной оценке равна 2 в -80-й степени (~ 10 в -25-й степени). Точной оценки нет, так как на данный момент мировому сообществу не удалось эффективно расшифровать данную криптографическую схему.

Объекты Git

Работу с версиями файлов в Git можно сравнить с обычными операциями над файловой системой. Структура состоит из четырех типов объектов: Blob, Tree, Commit и References; некоторые из них, в свою очередь, делятся на подобъекты.

Blob (Binary Large Object) – тип данных, который вмещает лишь содержимое файла и собственный SHA1-хеш. Blob является основным и единственным носителем данных в структуре Git. Можно провести параллель между данным объектом и инодами (inodes) в файловых системах, поскольку их структура и цели во многом схожи.

Дерево (Tree) – тип данных, который содержит:

- собственный SHA1-хеш;
- SHA1-хеш blob’ов и/или деревьев;
- права доступа Unix-систем;
- символьное имя объекта (название для внутреннего использования в системе).

По своей сути объект является аналогом директории. Он задает иерархию файлов проекта.

Commit – тип данных, который содержит:

- собственный SHA1-хеш;
- ссылку ровно на одно дерево;
- ссылку на предыдущий commit (их может быть и несколько);
- имя автора и время создания commit’a;
- имя коммитера (commiter – человек, применивший commit к репозиторию, он может отличаться от автора) и время применения commit’a;
- произвольный кусок данных (блок можно использовать для электронной подписи или, например, для пояснения изменений commit’a).

Данный объект призван хранить снимок (версию) группы файлов в определенный момент времени, можно сравнить его с контрольной точкой. Commit'ы можно объединять (merge), разветвлять (branch) или, например, установить линейную структуру, тем самым отражая иерархию версий проекта.

Reference – тип данных, содержащий ссылку на любой из четырех объектов (Blob, Tree, Commit и References). Основная цель его – прямо или косвенно указывать на объект и являться синонимом файла, на который он ссылается. Тем самым повышается понимание структуры проекта. Очень неудобно оперировать бессмысленным набором символов в названии, ссылку же, в отличие от SHA1-хеша, можно именовать так, как удобнее разработчику.

Из ссылок, в свою очередь, можно выделить ряд подобъектов, имеющих некоторые различия: Ветвь, Тег. Рассмотрим их.

Ветвь (Head, Branch) – символьная ссылка (Symbolic link), которая указывает на последний в хронологии commit определенной ветви и хранит SHA1-хеш объекта. Является типом данных журналируемых файловых систем. Данный вид объекта определяется не в самом Git, а наследуется от операционной и файловой систем. Ветвь используется как синоним файла, на который она ссылается, т.е. Git позволяет оперировать ею напрямую. Можно позволить себе не задумываться о том, работаете ли вы с последней версией или нет.

Тег (tag) – тип данных, который в отличие от ветвей неизменно ссылается на один и тот же объект типа blob, tree, commit или tag. Его, в свою очередь, можно разделить на легковесный (light tag) и тяжеловесный или аннотированный (annotated tag). Легкий тег, кроме неизменности ссылки, ничем не отличается от обычных ветвей, т.е. содержит лишь SHA1-хеш объекта, на который ссылается, внутри себя. Аннотированный тег состоит из двух частей:

первая часть содержит собственный SHA1-хеш;

вторая часть состоит из:

- SHA1 объекта, на который указывает аннотированный тег;
- тип указываемого объекта (blob, tree, commit или tag);
- символьное имя тега;
- дата и время создания тега;
- имя и e-mail создателя тега;
- произвольный кусок данных (данный блок можно использовать для электронной подписи или для пояснения тега).

Иными словами, проект в Git представляет собой набор blob'ов, которые связаны сетью деревьев. Полученная иерархическая структура может, в зависимости от времени, быть отражена в виде commit'ов – версий, а для понимания их структуры в Git присутствуют такие объекты, как ссылки. Исключая действия со ссылками, почти вся работа с объектами системы максимально автоматизирована изнутри. Отталкиваясь от механизма ссылок, мы приходим к следующей идее – работать именно над группами файлов. По мнению автора, мысль является ключевой в философии Git. Задав, например, операцию для данного commit'а, она рекурсивно отработает свою часть по дереву, на которое ссылается. Являясь расширением общепринятого взгляда "действие над каждым файлом", нововведение упрощает реализацию и подход со стороны программиста над повседневными задачами СУВ, такими как слияние/разделение ветвей, опять же рекурсивно автоматизируя процесс. Данный подход прост для понимания, быстро работает и

гибок в реализации своих целей. Многие из этих черт достигаются благодаря Unix-ориентированности системы, т.е. оперируя стандартными устройствами, Git опирается на уже имеющиеся в операционной системе решения.

Проясним момент хранения данных. Содержание файлов разных версий в хронологии занимает довольно много памяти. Так, например, в проекте из двадцати файлов двадцати версий архив будет весить в 20 раз больше (возможно, порядка сотни мегабайтов), а что будет, если количество и тех и других в 10 раз больше (вроде бы не намного)? Размер занятого пространства возрастет в 100 раз (т.е. примерно 1 ГБ). В реальных задачах скорость роста занимаемой памяти далеко не линейно зависит от времени. Для решения данной проблемы существует несколько оптимизаций:

каждый объект Git хранится в виде обыкновенного архива (tar.gz);

для всей иерархии файлов применяется последовательная дельта-компрессия.

Разберем на примере.

У вас есть трехлетняя история вашего проекта, в ней порядка тысячи файлов и ста версий. Если в определенный момент нужно будет обратиться к самой ранней версии, Git придется разархивировать дельта-компрессию всей истории файла. Неутешительно, но на данный процесс может уйти до полудня. Git предлагает делать так называемые контрольные точки, т.е. хранить недельта-архивированный файл через некоторое количество версий, которое назовем глубиной компрессии. Тогда в нашем примере вся история сужается до некоторого наперед заданного количества дельта-компрессий , разархивировав которые, можно взглянуть на любую версию в хронологии. Заметим, что дельта-компрессию наиболее целесообразно использовать над одними видами ближайших в иерархии объектов, для этого репозиторий необходимо отсортировать соответственно по типу и размеру. Данный ряд операций, описанных в этом пункте, выполняет функция git-repack (и git-gc, которая её содержит).

Слияние и разделение ветвей

Данный вопрос очень трудоемок и насыщен, в связи с чем введем понятия слияния и разделения только в общих чертах. Снова обратимся к примеру.

Представим себе момент разработки проекта, когда главной поставленной целью является скорость работы программы. Один из возможных тактических вариантов решения – разбить разработчиков на две группы, каждая из которых будет решать одну и ту же задачу. При этом ветвь истории проекта должна раздвоиться. Данная процедура называется ветвление (branch). Действие разветвления ветви – это простое создание её копии, которая впоследствии будет иметь свою историю.

Пусть мы получили два уже законченных результата одной и той же задачи, над которой работали две группы программистов. Как нам быть? Посмотреть, чей код быстрее и надежнее? Это слишком просто, но не всегда лучший выход. Хорошее решение – это, немного разобравшись в коде и файлах, разбить их на подзадачи или блоки кода. И только тогда уже выявлять сильные и слабые стороны данных кусочков. Конечно, этот вариант подходит только в том случае, когда вы заранее предусмотрели, что впоследствии сможете собрать все эти частицы воедино. Случай, когда вы сами разрабатываете код, улучшая и исправляя некоторые ошибки, равнозначен приведенному примеру. Данный процесс объединения двух целых в одно называется слияние (merge). Процесс объединения двух версий и есть ключевой момент ведения проекта. Как бы то ни было, стоит избегать автоматизированного исполнения данной операции. Отличительная черта Git – это максимально достоверный и довольно быстрый способ решения задачи ветвления.

К достоинствам системы можно отнести:

1. Unix-ориентированность.
2. Идеологическая выдержанность (следуя правилам использования системы, очень сложно попасть в безвыходную ситуацию или получить то, чего вы не ожидали).
3. Высокая производительность (это одно из самых явных достоинств системы, плата за которое есть «Идеологическая выдержанность» и «Unix-ориентированность»).
4. Интеграция Git со сторонними СУВ, такими как Subversion, Mercurial, …
5. Управление группой файлов (системе нет необходимости рассматривать изменения в каждом файле по отдельности, она запоминает любые изменения всего проекта, и если вдруг вам понадобится проследить единичные изменения, она выдаст ровно ту часть, которая связана с данным файлом).
6. Операция слияния (максимально автоматизированная реализация сложной задачи).

К недостаткам отнесем:

1. Unix-ориентированность (стоит отметить отсутствие зрелой реализации Git на не Unix-системах).
2. Необходимость периодического выполнения команды git-gc (пакует группы файлов и удаляет те, которые не связанны ссылками).
3. Коллизии хеширования (совпадение SHA1 хеша различных по содержанию файлов).

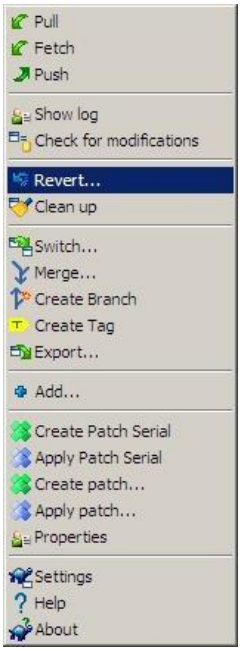
6. Интерфейсы Git

«Сколько людей, столько и мнений». Попробуем выделить ряд типов интерфейсов для работы с системой. Для определенных целей по-своему лучше каждое из приведеных ниже видов приложений.

Для людей, которые не занимаются разработкой вплотную, для «консерваторов» – тех, кто любит “кнопочки и галочки” и сознательно хочет оградить себя от непомерных усилий запоминания функций, ключей и многих тонкостей, больше подойдет вариант в стиле TortoiseGit или Git Extensions – простые интерфейсы. Они позволяют действовать преимущественно мышью и работают в привычной для многих ОС Windows.

Рисунок 1. Контекстное меню приложения.

Рисунок 2. Контекстное меню приложения.



ИЛИ

Рисунок 3. Контекстное меню приложения.

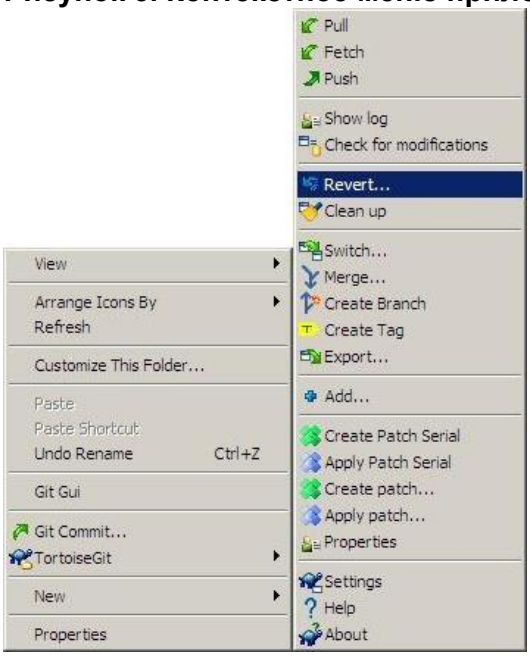


Рисунок 4. Окно Commit приложения.

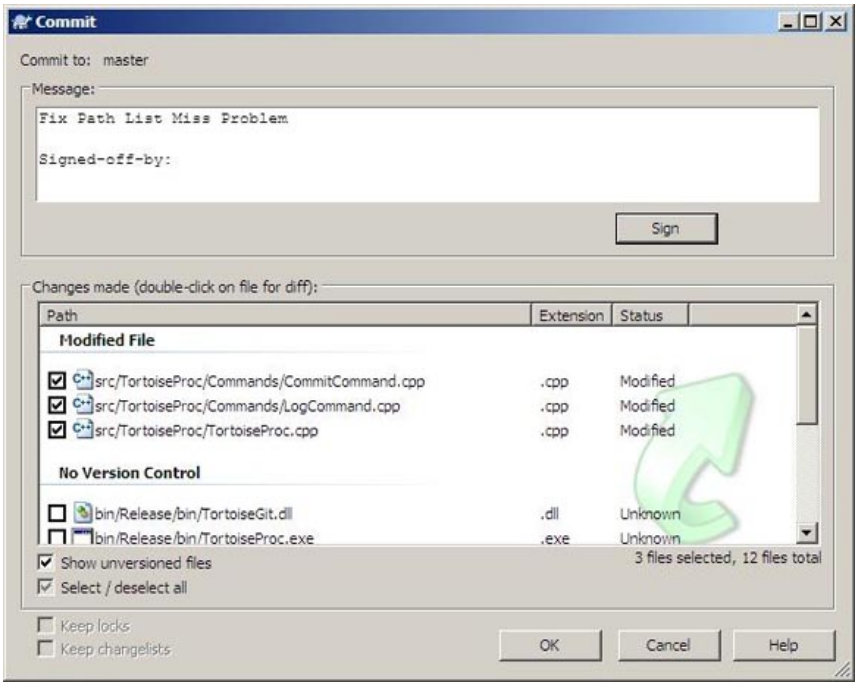
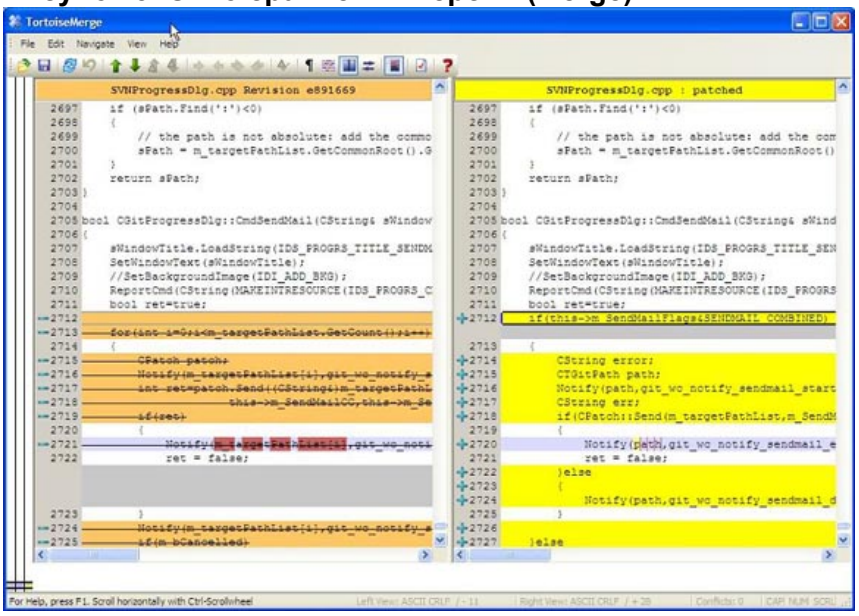
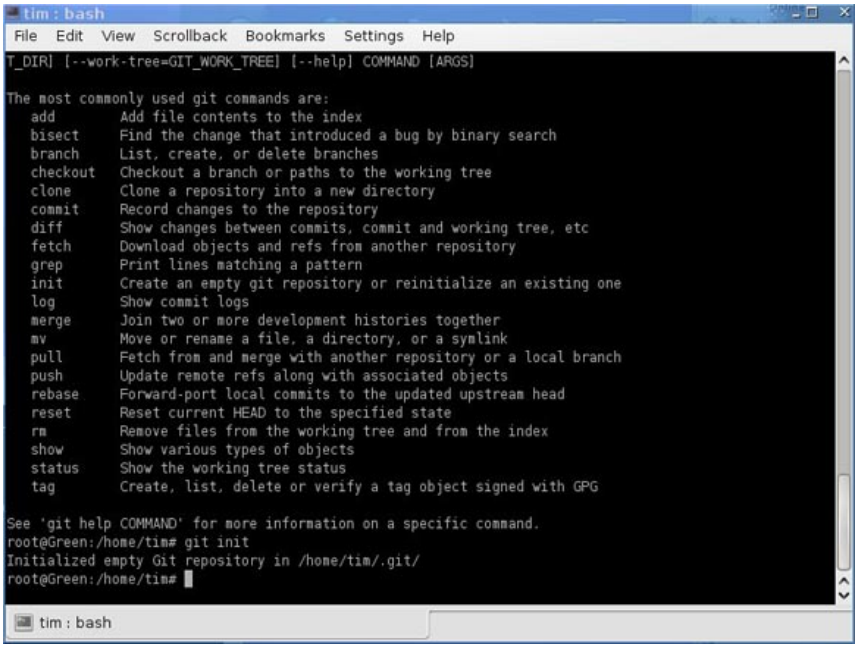


Рисунок 5. Окно сравнения версий (Merge).



Ровно противоположный тип интерфейса. Для программистов, которым постоянно необходимо взаимодействовать с сотрудниками, решать типичные задачи контроля именно кода, для людей, которые привыкли работать в Unix-like системах, используя терминал, лучше всего подойдет консольный вид приложений. Они так же просты в обращении, немного быстрее и функциональнее, но им придется уделить время, для того чтобы разобраться в использовании.

Рисунок 6. Консольный интерфейс Git.



Можно выделить и третий тип интерфейсов – смешение первых двух. Т.е. у вас есть консольное приложение, например, “родная” оболочка git. Вы можете использовать ряд дополнительных утилит, таких как Gitk или QGit, для отображения деревьев, упрощения обзора иерархии версий, различий между версиями, поиска нужных объектов.

Рисунок 7. Git + Gitk.

7. Заключение

Итак, читатели уже представляют себе, как работают современные системы контроля версий. Кроме того, нами была рассмотрена архитектура одной из самых популярных систем – Git. В следующей статье мы попробуем на практике познакомиться с некоторыми ее особенностями – рассмотрим функции и ключи к ним. В статье будет дан ряд наглядных примеров ведения истории файлов, а также изложена философия распределенных систем. Автор попытается показать читателям, как можно правильно использовать Git и рассмотрит некоторые типичные ошибки использования этой СУВ.