МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение

высшего профессионального образования «Саратовский государственный

технический университет имени Гагарина Ю.А.»

Реферат

По дисциплине «Вычислительные системы»

На тему «Распределённая система управления версиями файлов Git»

|  |  |
| --- | --- |
|  | Выполнил  Студент группы мИВЧТ-21  № зачетной книжки 111096  Ваденеев Даниил Михайлович |
|  | Проверил  Доцент кафедры ИСТ  Кинцель Дмитрий Александрович |

Саратов 2013

Оглавление

[Введение 3](#_Toc345545177)

[Системы управления версиями файлов 4](#_Toc345545178)

[Сраспределенные системы управления версиями файлов 5](#_Toc345545178)

[Терминология систем управления версиями файлов 8](#_Toc345545180)

[Особенности Git 11](#_Toc345545180)

[Установка Git 14](#_Toc345545179)

[Использование Git 22](#_Toc345545180)

[Заключение 33](#_Toc345545181)

[Информационные источники 34](#_Toc345545182)

Введение

Git (произн. «гит») — распределённая система управления версиями файлов. Проект был создан Линусом Торвальдсом для управления разработкой ядра Linux, первая версия выпущена 7 апреля 2005 года. На сегодняшний день поддерживается Джунио Хамано.

Примерами проектов, использующих Git, являются ядро Linux, Android, Drupal, Cairo, GNU Core Utilities, Mesa, Wine, Chromium, Compiz Fusion, FlightGear, jQuery, PHP, NASM, MediaWiki и некоторые дистрибутивы Linux (см. ниже).

Программа является свободной и выпущена под лицензией GNU GPL версии 2.

Скачать последнюю версию Git можно с официального веб сайта по адресу http://git-scm.com/.

Системы управления версиями файлов

Система управления версиями (от англ. Version Control System, VCS или Revision Control System) — 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.

Такие системы наиболее широко используются при разработке программного обеспечения для хранения исходных кодов разрабатываемой программы. Однако они могут с успехом применяться и в других областях, в которых ведётся работа с большим количеством непрерывно изменяющихся электронных документов. В частности, системы управления версиями применяются в САПР, обычно в составе систем управления данными об изделии (PDM). Управление версиями используется в инструментах конфигурационного управления (Software Configuration Management Tools).

Ситуация, в которой электронный документ за время своего существования претерпевает ряд изменений, достаточно типична. При этом часто бывает важно иметь не только последнюю версию, но и несколько предыдущих. В простейшем случае можно просто хранить несколько вариантов документа, нумеруя их соответствующим образом. Такой способ неэффективен (приходится хранить несколько практически идентичных копий), требует повышенного внимания и дисциплины и часто ведёт к ошибкам, поэтому были разработаны средства для автоматизации этой работы.

Традиционные системы управления версиями используют централизованную модель, когда имеется единое хранилище документов, управляемое специальным сервером, который и выполняет бо́льшую часть функций по управлению версиями. Пользователь, работающий с документами, должен сначала получить нужную ему версию документа из хранилища; обычно создаётся локальная копия документа, т. н. «рабочая копия». Может быть получена последняя версия или любая из предыдущих, которая может быть выбрана по номеру версии или дате создания, иногда и по другим признакам. После того, как в документ внесены нужные изменения, новая версия помещается в хранилище. В отличие от простого сохранения файла, предыдущая версия не стирается, а тоже остаётся в хранилище и может быть оттуда получена в любое время. Сервер может использовать т. н. дельта-компрессию — такой способ хранения документов, при котором сохраняются только изменения между последовательными версиями, что позволяет уменьшить объём хранимых данных. Поскольку обычно наиболее востребованной является последняя версия файла, система может при сохранении новой версии сохранять её целиком, заменяя в хранилище последнюю ранее сохранённую версию на разницу между этой и последней версией. Некоторые системы (например, ClearCase) поддерживают сохранение версий обоих видов: большинство версий сохраняется в виде дельт, но периодически (по специальной команде администратора) выполняется сохранение версий всех файлов в полном виде; такой подход обеспечивает максимально полное восстановление истории в случае повреждения репозитория.

Иногда создание новой версии выполняется незаметно для пользователя (прозрачно), либо прикладной программой, имеющей встроенную поддержку такой функции, либо за счёт использования специальной файловой системы. В этом случае пользователь просто работает с файлом, как обычно, и при сохранении файла автоматически создаётся новая версия.

Часто бывает, что над одним проектом одновременно работают несколько человек. Если два человека изменяют один и тот же файл, то один из них может случайно отменить изменения, сделанные другим. Системы управления версиями отслеживают такие конфликты и предлагают средства их решения. Большинство систем может автоматически объединить (слить) изменения, сделанные разными разработчиками. Однако такое автоматическое объединение изменений, обычно, возможно только для текстовых файлов и при условии, что изменялись разные (непересекающиеся) части этого файла. Такое ограничение связано с тем, что большинство систем управления версиями ориентированы на поддержку процесса разработки программного обеспечения, а исходные коды программ хранятся в текстовых файлах. Если автоматическое объединение выполнить не удалось, система может предложить решить проблему вручную.

Часто выполнить слияние невозможно ни в автоматическом, ни в ручном режиме, например, если формат файла неизвестен или слишком сложен. Некоторые системы управления версиями дают возможность заблокировать файл в хранилище. Блокировка не позволяет другим пользователям получить рабочую копию или препятствует изменению рабочей копии файла (например, средствами файловой системы) и обеспечивает, таким образом, исключительный доступ только тому пользователю, который работает с документом.

Распределенные системы управления версиями файлов

Распределенные системы управления версиями файлов - Distributed Version Control System, DVCS. Такие системы используют распределённую модель вместо традиционной клиент-серверной. Они, в общем случае, не нуждаются в централизованном хранилище: вся история изменения документов хранится на каждом компьютере, в локальном хранилище, и при необходимости отдельные фрагменты истории локального хранилища синхронизируются с аналогичным хранилищем на другом компьютере. В некоторых таких системах локальное хранилище располагается непосредственно в каталогах рабочей копии.

Когда пользователь такой системы выполняет обычные действия, такие как извлечение определённой версии документа, создание новой версии и тому подобное, он работает со своей локальной копией хранилища. По мере внесения изменений, хранилища, принадлежащие разным разработчикам, начинают различаться, и возникает необходимость в их синхронизации. Такая синхронизация может осуществляться с помощью обмена патчами или так называемыми наборами изменений (англ. change sets) между пользователями.

Описанная модель логически близка созданию отдельной ветки для каждого разработчика в классической системе управления версиями (в некоторых распределённых системах перед началом работы с локальным хранилищем нужно создать новую ветвь). Отличие состоит в том, что до момента синхронизации другие разработчики этой ветви не видят. Пока разработчик изменяет только свою ветвь, его работа не влияет на других участников проекта и наоборот. По завершении обособленной части работы, внесённые в ветви изменения сливают с основной (общей) ветвью. Как при слиянии ветвей, так и при синхронизации разных хранилищ возможны конфликты версий. На этот случай во всех системах предусмотрены те или иные методы обнаружения и разрешения конфликтов слияния.

С точки зрения пользователя распределённая система отличается необходимостью создавать локальный репозиторий и наличием в командном языке двух дополнительных команд: команды получения репозитория от удалённого компьютера (pull) и передачи своего репозитория на удалённый компьютер (push). Первая команда выполняет слияние изменений удалённого и локального репозиториев с помещением результата в локальный репозиторий; вторая — наоборот, выполняет слияние изменений двух репозиториев с помещением результата в удалённый репозиторий. Как правило, команды слияния в распределённых системах позволяют выбрать, какие наборы изменений будут передаваться в другой репозиторий или извлекаться из него, исправлять конфликты слияния непосредственно в ходе операции или после её неудачного завершения, повторять или возобновлять неоконченное слияние. Обычно передача своих изменений в чужой репозиторий (push) завершается удачно только при условии отсутствия конфликтов. Если конфликты возникают, пользователь должен сначала слить версии в своём репозитории (выполнить pull), и лишь затем передавать их другим.

Обычно рекомендуется организовывать работу с системой так, чтобы пользователи всегда или преимущественно выполняли слияние у себя в репозитории. То есть, в отличие от централизованных систем, где пользователи передают свои изменения на центральный сервер, когда считают нужным, в распределённых системах более естественным является порядок, когда слияние версий инициирует тот, кому нужно получить его результат (например, разработчик, управляющий сборочным сервером).

Основные преимущества распределённых систем — их гибкость и значительно бо́льшая (по сравнению с централизованными системами) автономия отдельного рабочего места. Каждый компьютер разработчика является, фактически, самостоятельным и полнофункциональным сервером, из таких компьютеров можно построить произвольную по структуре и уровню сложности систему, задав (как техническими, так и административными мерами) желаемый порядок синхронизации. При этом каждый разработчик может вести работу независимо, так, как ему удобно, изменяя и сохраняя промежуточные версии документов, пользуясь всеми возможностями системы (в том числе доступом к истории изменений) даже в отсутствие сетевого соединения с сервером. Связь с сервером или другими разработчиками требуется исключительно для проведения синхронизации, при этом обмен наборами изменений может осуществляться по различным схемам.

К недостатком распределённых систем можно отнести увеличение требуемого объёма дисковой памяти: на каждом компьютере приходится хранить полную историю версий, тогда как в централизованной системе на компьютере разработчика обычно хранится лишь рабочая копия, то есть срез репозитория на какой-то момент времени и внесённые изменения. Менее очевидным, но неприятным недостатком является то, что в распределённой системе практически невозможно реализовать некоторые виды функциональности, предоставляемые централизованными системами. Это:

Блокировка файла или группы файлов (для хранения признака блокировки нужен общедоступный и постоянно находящийся в онлайне центральный сервер). Это вынуждает применять специальные административные меры, если приходится работать с бинарными файлами, непригодными для автоматического слияния.

Слежение за определённым файлом или группой файлов (изменения файлов происходят на разных серверах, слияния и выделения ветвей происходят локально, об изменениях становится известно только при синхронизации, причём не всем разработчикам, а только тем, кто в данной синхронизации участвует).

Единая сквозная нумерация версий системы и/или файлов, в которой номер версии монотонно возрастает (такая нумерация также требует наличия главного сервера, задающего номера версий для всех остальных). В распределённых системах приходится обходиться локальными обозначениями версий и применять теги, назначение которых определяется соглашением между разработчиками или корпоративными стандартами фирмы.

Локальная работа пользователя с отдельной, небольшой по объёму выборкой из значительного по размеру и внутренней сложности хранилища на удалённом сервере.

Можно выделить следующие типичные ситуации, в которых использование распределённой системы даёт заметные преимущества:

- Периодическая синхронизация нескольких компьютеров под управлением одного разработчика (рабочего компьютера, домашнего компьютера, ноутбука и так далее). Использование распределённой системы избавляет от необходимости выделять один из компьютеров в качестве сервера, а синхронизация выполняется по необходимости, обычно при «пересадке» разработчика с одного устройства на другое.

- Совместная работа над проектом небольшой территориально распределённой группы разработчиков без выделения общих ресурсов. Как и в предыдущем случае, реализуется схема работы без главного сервера, а актуальность репозиториев поддерживается периодическими синхронизациями по схеме «каждый с каждым».

- Крупный распределённый проект, участники которого могут долгое время работать каждый над своей частью, при этом не имеют постоянного подключения к сети. Такой проект может использовать централизованный сервер, с которым синхронизируются копии всех его участников. Возможны и более сложные варианты — например, с созданием групп для работы по отдельным направлениям внутри более крупного проекта. При этом могут быть выделены отдельные «групповые» серверы для синхронизации работы групп, тогда процесс окончательного слияния изменения становится древовидным: сначала отдельные разработчики синхронизируют изменения на групповых серверах, затем обновлённые репозитории групп синхронизируются с главным сервером. Возможна работа и без «групповых» серверов, тогда разработчики одной группы синхронизируют изменения между собой, после чего любой из них (например, руководитель группы) передаёт изменения на центральный сервер.

В традиционной «офисной» разработке проектов, когда группа разработчиков относительно невелика и целиком находится на одной территории, в пределах единой локальной компьютерной сети, с постоянно доступными серверами, централизованная система может оказаться лучшим выбором из-за своей более жёсткой структуры и наличия функциональности, отсутствующей в распределённых системах (например, уже упомянутой блокировки). Возможность фиксировать изменения без их слияния в центральную ветвь в таких условиях легко реализуется путём выделения незавершённых работ в отдельные ветви разработки.

Терминология систем управления версиями файлов

Общепринятой терминологии не существует, в разных системах могут использоваться различные названия для одних и тех же действий. Ниже приводятся некоторые из наиболее часто используемых вариантов. Приведены английские термины, в литературе на русском языке используется тот или иной перевод или транслитерация.

**branch**

Ветвь — направление разработки, независимое от других. Ветвь представляет собой копию части (как правило, одного каталога) хранилища, в которую можно вносить свои изменения, не влияющие на другие ветви. Документы в разных ветвях имеют одинаковую историю до точки ветвления и разные — после неё.

**changeset, activity**

Набор изменений. Представляет собой поименованный набор правок, сделанных в локальной копии для какой-то общей цели. В системах, поддерживающих наборы правок, разработчик может объединять локальные правки в группы и выполнять фиксацию логически связанных изменений одной командой, указывая требуемый набор правок в качестве параметра. При этом прочие правки останутся незафиксированными. Типичный пример: ведётся работа над добавлением новой функциональности, а в этот момент обнаруживается критическая ошибка, которую необходимо немедленно исправить. Разработчик создаёт набор изменений для уже сделанной работы и новый — для исправлений. По завершении исправления ошибки отдаётся команда фиксации только второго набора правок.

**check-in, commit, submit**

Создание новой версии, фиксация изменений. Распространение изменений, сделанных в рабочей копии, на хранилище документов. При этом в хранилище создаётся новая версия изменённых документов.

**check-out, clone**

Извлечение документа из хранилища и создание рабочей копии.

**conflict**

Конфликт — ситуация, когда несколько пользователей сделали изменения одного и того же участка документа. Конфликт обнаруживается, когда один пользователь зафиксировал свои изменения, а второй пытается зафиксировать и система сама не может корректно слить конфликтующие изменения. Поскольку программа может быть недостаточно разумна для того, чтобы определить, какое изменение является «корректным», второму пользователю нужно самому разрешить конфликт (resolve).

**head**

Основная версия — самая свежая версия для ветви/ствола, находящаяся в хранилище. Сколько ветвей, столько основных версий.

**merge, integration**

Слияние — объединение независимых изменений в единую версию документа. Осуществляется, когда два человека изменили один и тот же файл или при переносе изменений из одной ветки в другую.

**rebase**

Перенос точки ветвления (версии, от которой начинается ветвь) на более позднюю версию основной ветви. Например, после выпуска версии 1.0 проекта в стволе продолжается доработка (исправление ошибок, доработка имеющегося функционала), одновременно начинается работа над новой функциональностью в новой ветви. Через какое-то время в основной ветви происходит выпуск версии 1.1 (с исправлениями); теперь желательно, чтобы ветвь разработки новой функциональности включала изменения, произошедшие в стволе. Вообще, это можно сделать базовыми средствами, с помощью слияния (merge), выделив набор изменений между версиями 1.0 и 1.1 и слив его в ветвь. Но при наличии в системе поддержки перебазирования ветви эта операция делается проще, одной командой: по команде rebase (с параметрами: ветвью и новой базовой версией) система самостоятельно определяет нужные наборы изменений и производит их слияние, после чего для ветви базовой версией становится версия 1.1; при последующем слиянии ветви со стволом система не рассматривает повторно изменения, внесённые между версиями 1.0 и 1.1, так как ветвь логически считается выделенной после версии 1.1.

**repository, depot**

Хранилище документов — место, где система управления версиями хранит все документы вместе с историей их изменения и другой служебной информацией.

**revision**

Версия документа. Системы управления версиями различают версии по номерам, которые назначаются автоматически.

**shelving**

Откладывание изменений. Предоставляемая некоторыми системами возможность создать набор изменений (changeset) и сохранить его на сервере без фиксации (commit’а). Отложенный набор изменений доступен на чтение другим участникам проекта, но до специальной команды не входит в основную ветвь. Поддержка откладывания изменений даёт возможность пользователям сохранять незавершённые работы на сервере, не создавая для этого отдельных ветвей.

**tag, label**

Метка, которую можно присвоить определённой версии документа. Метка представляет собой символическое имя для группы документов, причем метка описывает не только набор имен файлов, но и версию каждого файла. Версии включенных в метку документов могут принадлежать разным моментам времени.

**trunk, mainline, master**

Ствол — основная ветвь разработки проекта. Политика работы со стволом может отличаться от проекта к проекту, но в целом она такова: большинство изменений вносится в ствол; если требуется серьёзное изменение, способное привести к нестабильности, создаётся ветвь, которая сливается со стволом, когда нововведение будет в достаточной мере испытано; перед выпуском очередной версии создаётся «релизная» ветвь, в которую вносятся только исправления.

**update, sync**

Синхронизация рабочей копии до некоторого заданного состояния хранилища. Чаще всего это действие означает обновление рабочей копии до самого свежего состояния хранилища. Однако при необходимости можно синхронизировать рабочую копию и к более старому состоянию, чем текущее.

**working copy**

Рабочая (локальная) копия документов.

Особенности Git

Система спроектирована как набор программ, специально разработанных с учётом их использования в скриптах. Это позволяет удобно создавать специализированные системы контроля версий на базе Git или пользовательские интерфейсы. Например, Cogito является именно таким примером фронтенда к репозиториям Git, а StGit использует Git для управления коллекцией патчей.

Git поддерживает быстрое разделение и слияние версий, включает инструменты для визуализации и навигации по нелинейной истории разработки. Как и Darcs, BitKeeper, Mercurial, Bazaar и Monotone, Git предоставляет каждому разработчику локальную копию всей истории разработки, изменения копируются из одного репозитория в другой.

Удалённый доступ к репозиториям Git обеспечивается git-daemon, SSH- или HTTP-сервером. TCP-сервис git-daemon входит в дистрибутив Git и является наряду с SSH наиболее распространённым и надёжным методом доступа. Метод доступа по HTTP, несмотря на ряд ограничений, очень популярен в контролируемых сетях, потому что позволяет использовать существующие конфигурации сетевых фильтров.

Ядро Git представляет собой набор утилит командной строки с параметрами. Все настройки хранятся в текстовых файлах конфигурации. Такая реализация делает Git легко портируемым на любую платформу и даёт возможность легко интегрировать Git в другие системы (в частности, создавать графические git-клиенты с любым желаемым интерфейсом).

Репозиторий Git представляет собой каталог файловой системы, в котором находятся файлы конфигурации репозитория, файлы журналов, хранящие операции, выполняемые над репозиторием, индекс, описывающий расположение файлов и хранилище, содержащее собственно файлы. Структура хранилища файлов не отражает реальную структуру хранящегося в репозитории файлового дерева, она ориентирована на повышение скорости выполнения операций с репозиторием. Когда ядро обрабатывает команду изменения (неважно, при локальных изменениях или при получении патча от другого узла), оно создаёт в хранилище новые файлы, соответствующие новым состояниям изменённых файлов. Существенно, что никакие операции не изменяют содержимого уже существующих в хранилище файлов.

По умолчанию репозиторий хранится в подкаталоге с названием «.git» в корневом каталоге рабочей копии дерева файлов, хранящегося в репозитории. Любое файловое дерево в системе можно превратить в репозиторий git, отдав команду создания репозитория из корневого каталога этого дерева (или указав корневой каталог в параметрах программы). Репозиторий может быть импортирован с другого узла, доступного по сети. При импорте нового репозитория автоматически создаётся рабочая копия, соответствующая последнему зафиксированному состоянию импортируемого репозитория (то есть не копируются изменения в рабочей копии исходного узла, для которых на том узле не была выполнена команда commit).

Преимущества и недостатки git по сравнению с централизованными системами управления версиями (такими как, например, Subversion) типичны для любой распределённой системы и описаны в статье «Система управления версиями». Если же сравнивать git с «родственными» ей распределёнными системами, можно отметить, что git изначально идеологически ориентирован на работу с изменениями, а не с файлами, «единицей обработки» для него является набор изменений, или патч. Эта особенность прослеживается как в структуре самой системы (в частности — в структуре репозитория), так и в принципах построения команд; она отражается на производительности системы в различных вариантах её использования и на достоинствах и недостатках git по сравнению с другими DVCS.

Часто называемые преимущества git перед другими DVCS:

Высокая производительность.

Развитые средства интеграции с другими VCS, в частности, с CVS, SVN и Mercurial. Помимо разнонаправленных конвертеров репозиториев, имеющиеся в комплекте программные средства позволяют разработчикам использовать git при размещении центрального репозитория в SVN или CVS, кроме того, git может имитировать cvs-сервер, обеспечивая работу через клиентские приложения и поддержку в средах разработки, специально не поддерживающих git.

Продуманная система команд, позволяющая удобно встраивать git в скрипты.

Качественный веб-интерфейс «из коробки».

Репозитории git могут распространяться и обновляться общесистемными файловыми утилитами архивации и обновления, такими как rsync, благодаря тому, что фиксации изменений и синхронизации не меняют существующие файлы с данными, а только добавляют новые (за исключением некоторых служебных файлов, которые могут быть автоматически обновлены с помощью имеющихся в составе системы утилит). Для раздачи репозитория по сети достаточно любого веб-сервера.

В числе недостатков git обычно называют:

Отсутствие переносимой на другие операционные системы поддержки путей в кодировке Unicode в Microsoft Windows. Если путь содержит символы, отличные от ANSI, то их поддержка из командной строки требует специфических настроек, которые не гарантируют правильного отображения файловых имён при пользовании тем же репозиторием из других ОС. Одним из способов решения проблемы для git 1.7 является использование специально пропатченного консольного клиента. Другой вариант — использование графических утилит, работающих напрямую через API, таких как TortoiseGit.

Некоторое неудобство для пользователей, переходящих с других VCS. Команды git, ориентированные на наборы изменений, а не на файлы, могут вызвать недоумение у пользователей, привыкших к файл-ориентированным VCS, таким как SVN. Например, команда «add», которая в большинстве систем управления версиями производит добавление файла к проекту, в git подготавливает к фиксации сделанные в файлах изменения. При этом сохраняется не патч, описывающий изменения, а новая версия целевого файла.

Использование для идентификации ревизий хэшей SHA1, что приводит к необходимости оперировать длинными строками вместо коротких номеров версий, как во многих других системах (хотя в командах допускается использование неполных хэш-строк).

Бо́льшие накладные расходы при работе с проектами, в которых делаются многочисленные несвязанные между собой изменения файлов. При работе в таком режиме размеры наборов изменений становятся достаточно велики и происходит быстрый рост объёма репозиториев.

Бо́льшие затраты времени, по сравнению с файл-ориентированными системами, на формирование истории конкретного файла, истории правок конкретного пользователя, поиска изменений, относящихся к заданному месту определённого файла.

Отсутствие отдельной команды переименования/переноса файла, которая отображалась бы в истории как соответствующее единое действие. Существующий скрипт git mv фактически выполняет переименование, копирование файла и удаление его на старом месте, что требует специального анализа для определения, что в действительности файл был просто перенесён (этот анализ выполняется автоматически командами просмотра истории).

Система работает только с файлами и их содержимым, и не отслеживает пустые каталоги.

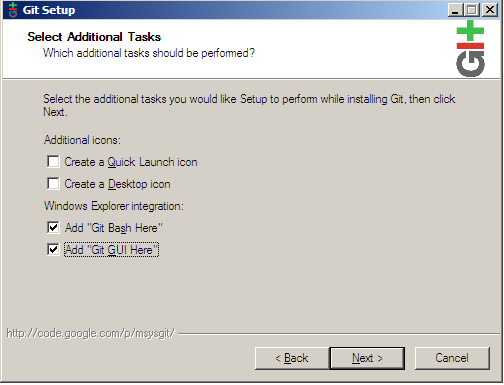
Некоторые команды работают неожиданно, в частности, могут приводить к неочевидным ошибкам или требовать для правильной работы указания специальных параметров, когда применяются к исходно пустому репозиторию или к репозиторию, в котором ещё не было сделано ни одного коммита.

В ряде публикаций, относящихся преимущественно к 2005—2008 годам можно встретить также нарекания в отношении документации git, отсутствия удобной windows-версии и удобных графических клиентов. В настоящее время эта критика неактуальна: существует версия git на основе MinGW («родная» сборка под Windows), и несколько высококачественных графических клиентов для различных операционных систем, в частности, под Windows имеется клиент TortoiseGit, идеологически очень близкий к широко распространённому TortoiseSVN — клиенту SVN, встраиваемому в оболочку Windows.

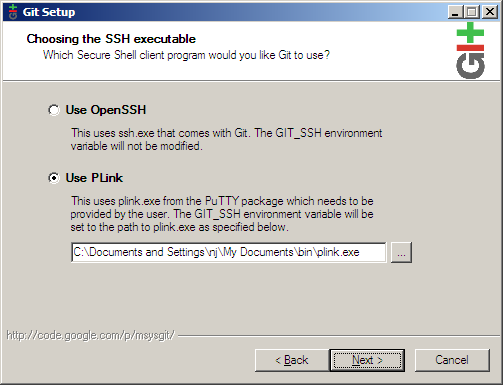
Установка Git

Рассмотрим процесс установки msysgit – системы Git с графическим интерфейсом, предназначенного для операционной системы Windows. Для работы в Git необходимо наличие SSH программу программы, позволяющей однозначно идентифицировать себя при размещении измений (коммитов) в публичных или иных репозиториях. В данном случае рассматривается использование отличной от входящей в поставку SSH программы PuTTY Agent.

Для начала, скачайте msysgit. После завершения скачивания необходимо запустить единственный загруженный файл, который произведет установку всего необходимого программного обеспечения. При прохождении шагов установочной программы, вы можете отметить опцию интеграции с Windows Explorer.



Далее следует диалоговое окно выбора SSH программы, в данном случае мы выбираем установленную ранее PuTTY Agent.



На этом процесс установки графиечского интерфейса Git для Windows завершен.

Также дополнительно опишем процесс установки для операционных систем семейства Linux:

Если вы хотите установить Git под Linux как бинарный пакет, это можно сделать, используя обычный менеджер пакетов вашего дистрибутива. Если у вас Fedora, можно воспользоваться yum'ом:

*$ yum install git-core*

Если же у вас дистрибутив, основанный на Debian, например, Ubuntu, попробуйте apt-get:

*$ apt-get install git-core*

Проблема данного процесса установки заключается в том, что многие дистрибутивы содержат весьма старые версии программного продукта Git, поэтому одним из решений данного вопроса может быть сборка из исходного кода:

Для установки Git'а вам понадобятся библиотеки, от которых он зависит: curl, zlib, openssl, expat и libiconv. Например, если в вашей системе менеджер пакетов ― yum (Fedora), или apt-get (Debian, Ubuntu), можно воспользоваться следующими командами, чтобы разрешить все зависимости:

*$ yum install curl-devel expat-devel gettext-devel \*

*openssl-devel zlib-devel*

*$ apt-get install libcurl4-gnutls-dev libexpat1-dev gettext \*

*libz-dev libssl-dev*

Установив все необходимые библиотеки, можно идти дальше и скачать последнюю версию с сайта Git'а:

http://git-scm.com/download

Теперь скомпилируйте и установите:

*$ tar -zxf git-1.7.2.2.tar.gz*

*$ cd git-1.7.2.2*

*$ make prefix=/usr/local all*

*$ sudo make prefix=/usr/local install*

После этого вы можете скачать Git с помощью самого Git'а, чтобы получить обновления:

*$ git clone git://git.kernel.org/pub/scm/git/git.git*

Использование Git

При использовании Git возможно два направление: управление при помощи графического интерфейса или управление командной строкой.

Особенности использования в консаольной версии:

Создание нового репозитория

Создание нового репозитория – это наверное первая из задач, с которой сталкивается любой разработчик. К данной проблеме есть несколько подходов.

Во-первых, можно использовать только локальный репозиторий, но смысла особого я в этом не вижу.

Во-вторых, можно создать открытый репозиторий, с которым смогут работать другие люди. Рассмотрим этот вариант. Предположим, что у вас есть некий сервер example.com и ваша рабочая машинка.

Для начала вам необходимо установить git. Сделать это можно как вручную скомпилировав его из исходных кодов, так и установив с помощью менеджера пакетов. Будем считать что с этой операцией вы уже справились.

Далее вам надо создать репозиторий. Предположим что вы будете хранить все git-репозитории в /var/git:

*cd /var/git*

*mkdir example.git*

*cd example.git*

*git init --bare --share*

Теперь у вас есть пустой git-репозиторий. Это, собственно говоря, все команды, которые требуется выполнить на сервере.

Теперь идем на вашу рабочую машину и пишем:

*git init*

*git remote add origin ssh://username@example.com/var/git/example.git*

*touch changelog*

*git add changelog*

*git commit -a -m"Initital commit"*

*git push origin master*

Этими командами мы создали новый репозиторий, связали его с основной веткой example.git, закомитили туда файл changelog и отослали это на сервер. В результате выполнения этих команд, вы должны увидеть чтото примерно такое:

*Counting objects: 3, done.*

*Writing objects: 100% (3/3), 224 bytes, done.*

*Total 3 (delta 0), reused 0 (delta 0)*

*To ssh://username@example.git/var/git/example.git*

*\* [new branch] master -&gt; master*

Получение и оправка репозитория

Разумеется, после создания репозитория разумно узнать “а как же получить эти данные на свой компьютер?”.

Для данной операции используется команда clone. Она создает полную копию удаленного репозитория у вас. Под полной копией понимается именно полная копия, со всеми ветками, удаленными файлами и т.д.

*git clone ssh://username@example.com/var/git/имя-репозитория*

Разумеется, /var/git может меняться, в зависимости от того, где располагаются файлы данного репозитория на удаленной машине.

Обновление данных

С таким локальным репозиторием можно работать долгое время, однако, когда-то у вас все же возникнет необходимость передать сделанные вами изменения в удаленный репозиторий, чтобы результатами вашего труда могли воспользоваться другие участники разработки, и получить из удаленного репозитория новую версию. Для этого служат команды pull и push.

*git pull*

Данная команда получает обновленную версию из удаленного репозитория, при этом она проверяет на наличие различных проблем при объединении репозиториев и сообщает об этом.

*git push*

Данная команда целиком передает все сделанные вами изменения, уже закомиченные в локальный репозиторий, в удаленный репозиторий. Для передачи тегов необходимо использовать аргумент –tags

*git push --tags*

Базовые операции

Для базовой работы с любой системой контроля версий требуется не особенно большой набор операций: добавление файла в репозиторий, удаление файла из репозитория, комит изменений в репозиторий, отмена незакомиченных изменений и получение списка изменений.

Добавление списка файлов в комит:

*git add file1 file2 ... fileN*

Добавление всех недобавленных файлов в комит:

*git add -a*

Удаление файла из комита:

*git rm file1 file2 ... fileN*

Удаление файла из комита и с жесткого диска:

*git rm -f file1 file2 ... fileN*

Комит в локальный репозиторий (надо отметить, что в таком случае закомитятся только файлы, которые были обработаны с помощью git add/rm):

*git commit*

Комит всех изменений в локальный репозиторий:

*git commit -a*

Отмена всех изменений, сделанных в дереве, до состояния, которое было при последнем комите в локальный репозиторий( очень опасная команда, подумайте прежде чем пользоваться ею):

*git reset --hard*

Создание дифа относительно последнего комита:

*git diff*

Использование веток

Рано или поздно в любом проекте возникает ситуация, когда требуется заморозить изменения, но продолжать работать, а на замороженные изменения накладывать только баг-фиксы. Для этой цели служат ветви (branch)

В гите можно создать ветку от любого места. Для создания ветки от основного дерева надо выполнить следующую команду:

*git checkout --track -b name\_of\_newbranch origin/master*

В результате этой команды вы увидите примерно такое сообщение:

*Branch name\_of\_newbranch set up to track remote branch refs/remotes/origin/master.*

*Switched to a new branch "name\_of\_newbranch"*

Это значит, что в локальном репозитории у вас создалась новая ветка.

Если в этой команде заменить origin/master на origin/remote\_branch\_name то вы создадите ветку от другой ветки.

Чтобы ваша ветка была видна всем, ее требуется пропихнуть в удаленный репозиторий. Делается это так:

*git push origin local\_branch\_name:remote\_branch\_name*

Разумеется, надо также уметь и получать ветки в свое распоряжение:

*git branch local\_branch\_name origin/remote\_branch\_name*

*git checkout local\_branch\_name*

В результате вы получите искомую ветку после следующего сообщения:

*Switched to branch "local\_branch\_name"*

Работа с тегами

Как правило, кроме веток разработчики используют теги – чтобы запомнить состояние кода в какой-то момент. Тег – это своеобразный слепок, точно идентифицирующий состояние кода. Гит умеет работать с подписанными GPG тегами и с неподписанными. Здесь я рассмотрю только неподписанные теги.

Для создания такого тега необходимо выполнить команду:

*git tag*

Чтобы убрать тег необходимо выполнить:

*git tag -d*

Для того, чтобы тег стал виден всем, необходимо отправить его в удаленный репозиторий:

*git push --tags*

Чтобы получить версию с конкретного тега необходимо создать от него локальную ветку и расчекаутить эту ветку:

*git fetch origin tag*

*git branch*

*git checkout*

Разумеется, в будущем эту ветку можно будет сделать глобальной и выслать в удаленной репозиторий.

Настройка git

Для ускорения некоторых операций и увеличения удобства работы можно провести пару настроек:

Настройка цветного вывода:

*git config --global color.ui "auto"*

Настройка имени пользователя и почтового адреса

*git config user.name "FirstName LastName"*

*git config user.email "user@example.com"*

Заключение

Распределенная система управления версиями файлов Git является мощным универсальным, кроссплатформенным инструментом для качественной полноценной разработки программного обеспечения группой разработчиков. В данной работе отражены базовые элементы управления Git , процесс установки для различных типов операционных систем.

Информационные источники

1. <http://git-scm.com/book/ru>
2. <http://ru.wikipedia.org/wiki/Система_управления_версиями>
3. <http://tutorials.assembla.com/git-guide-for-windows-users/tour.ru.html>
4. <http://kb.etersoft.ru/Работа_с_git>