Системы контроля версий

Что такое управление версиями и зачем оно?

Итак, предположим, вы разрабатываете программу. Для простоты сначала предположим, что вы разрабатываете ее в одиночку и для собственного удовольствия. Как это выглядит?

Создаем директорию MyApplication, в ней - собственно файлы программы (скажем, \*.c). Далее начинается итерационный процесс: написали некоторое количество кода → скомпилировали → запустили → обнаружили ошибки или недоработки → отредактировали исходники → … Всё просто, в целом. Но через некоторое время начинаются нетривиальные вещи. Например:

* Последнее изменение сделало программу хуже, чем она была до того. Хочется вернуть всё "как было". Хорошо, если не успели закрыть редактор - делаем Undo. А если закрыли?
* Хотим приделать к программе новую функциональность, но сразу ясно, что, во-первых, сразу после написания она будет глючить (т.к. еще не отлажена), а во-вторых, в сложной программе добавление новой функциональности может добавить ошибок в ранее работавшую часть программы. Хочется, в случае чего, откатиться к заведомо работающему варианту и начать всё сначала.
* Если программа предназначена не только для вас лично, то вам придется делать "релизы", т.е. выпускать заведомо работающие версии программы. Для подготовки релиза вам придется фиксить некоторое количество багов. Это довольно скучный процесс, хочется разбавлять его добавлением новых возможностей в программу (это, как правило, интереснее). Но нельзя: добавление новых фич обязательно добавит багов, и вы никогда не доедете до релиза.

Первое, что приходит в голову для решения всех этих проблем - резервные копии. Скажем, каждую работающую версию программы упаковывать в MyApplication\_v0.001.tar.gz, в случае чего - откатываться. Для подготовки к релизу - сделать копию директории MyApplication, в которой только фиксить баги, а в основной - добавлять функционал. При этом подходе через некоторое время разработки у вас появится куча \*.tar.gz и не меньше копий рабочей директории. Очень быстро вы начнете путаться, где что.

Если вы разрабатываете программу не в одиночку, а, скажем, с группой товарищей, появляются новые неочевидности:

* Вы добавили в программу (в свою копию) возможность A, а ваш товарищ (в свою) - возможность B. Хотелось бы, чтобы у обоих была версия с обоими возможностями.
* Вы добавили в программу некоторую функцию, ваш товарищ добавил ее же, но реализовал по-другому. Какую версию выпускать и поддерживать дальше?
* Ваш товарищ в какой-то момент разработки понял, что программу можно улучшить и упростить, если изменить ее архитектуру, что-то обобщить, что-то вынести в отдельный модуль и т.п., и начал переделку (это называется умным словом рефакторинг). Вы же в это время в старой версии программы находите баги и фиксите их. Т.к. в версии вашего товарища в основном тот же код, эти баги наверняка есть и там - хотелось бы их сразу пофиксить.

Даже в случае двух разработчиков, при "ручном" согласовании изменений после определенного объема программы эти неочевидности превращаются в окончательную неразбериху. При разработке сколько-нибудь большой программы в конторе "ручное" согласование изменений видится почти невозможным.

Все эти проблемы призваны решить специальные программы - системы контроля версий (version control systems, vcs).

Общие свойства VCS

Если вы работаете с vcs, то у вас есть место, где хранится вся история изменений в программе - это место называется репозиторий (repository). Для внесения изменений в программу вам нужно получить копию текущей разрабатываемой версии - она называется вашей рабочей копией (working copy). Процесс получения рабочей копии традиционно называется словом "чекаут" (checkout). В случае коллективной разработки, вам придется поддерживать актуальность вашей рабочей копии (проверять, что там остальные наработали) - это обычно называется "апдейт" (update). Чтобы ваши изменения попали в репозиторий, вам нужно сделать коммит (commit) (еще это называется словом checkin). Часто словом commit называют собственно сам набор изменений, зафиксированный этой операцией (более точный термин - changeset). Каждое зафиксированное изменение называют ревизией. Вы можете в любой момент откатить свою рабочую копию к любой предыдущей ревизии - это называется revert.

Для того чтобы отслеживать такие вещи, как релизы, или варианты рефакторинга программы, или тестировать версии с новыми функциями - в репозитории создаются ветви разработки (branches). Для согласования хода разработки в разных ветках (она может вестись, и обычно ведется, параллельно) время от времени делают слияние веток (merge) - когда к одной ветке применяются те же изменения, что были проделаны в другой. Если какое-то место какого-то файла было по-разному изменено в разных ветках, при слиянии возникает конфликт (merge conflict) - его приходится решать вручную, выбирая один из вариантов или как-то логично их объединяя.

Так как ревизий в репозитории обычно довольно много, возникает необходимость некоторые из них как-то помечать (например, выпущенные релизы, или просто состояния программы, в которых она заведомо работала). Такие пометки называют тегами (tags). Таким образом, теги - это поименованные ревизии в репозитории.

Централизованные vs распределенные VCS

Всё множество VCS можно разделить на два класса - централизованные и распределенные.

В случае с централизованной VCS репозиторий хранится на одном сервере, и все разработчики работают с ним. Очевидное преимущество: простое управление выпуском релизов и вообще ходом развития программы, раз весь код в одном месте. Очевидный недостаток: если с сервером что-то случится, работа всех разработчиков пропадет (даже в случае регулярных бэкапов - пропадет работа всех разработчиков, скажем, за последнюю неделю). Известные примеры централизованных vcs - CVS, Subversion, Perforce.

В случае с распределенной VCS, каждый разработчик имеет полную копию репозитория и работает с ним, время от времени синхронизируясь с репозиториями остальных. Таким образом, для организации целенаправленной разработки потребуются какие-то не технические, а организационно-административные средства. Зато получаем множество преимуществ:

* Раз у каждого есть полная копия репозитория, работа всех разработчиков пропасть не может вообще.
* Часто выполняемые операции - прежде всего, commit - происходят почти мгновенно, т.к. не требуют соединения по сети.
* Каждый разработчик может создавать в своем репозитории ветки для каких-то экспериментов, всем остальным даже не нужно знать об этом.
* Т.к. в распределенных VCS предполагается регулярная синхронизация репозиториев, в них гораздо более эффективно реализована операция слияния веток (здесь это одна из базовых операций, в отличие от централизованных VCS, где это делается нечасто).
* Каждый разработчик может взять ("утянуть") у другого один или несколько коммитов, применив их к своему коду.

Система управления версиями Subversion.

(www.subversion.tigris.org)

Subversion – эта централизованная система управления версиями, созданная в 2000 году и основанная на технологии клиент-сервер. Она обладает всеми достоинствами CVS и решает основные ее проблемы (переименование и перемещение файлов и каталогов, работа с двоичными файлами и т.д.). Часто ее называют по имени клиентской части – SVN.

Принцип работы с Subversion очень походит на работу с CVS. Клиенты копируют изменения из репозитория и объединяют их с локальным проектом пользователя. Если возникают конфликты локальных изменений и изменений, сохраненных в репозитории, то такие ситуации разрешаются вручную. Затем в локальный проект вносятся изменения, и полученный результат сохраняется в репозитории.

При работе с файлами, не позволяющими объединять изменения, может использоваться следующий принцип:

1. Файл скачивается из репозитория и блокируется (запрещается его скачивание из репозитория).
2. Вносятся необходимые изменения.
3. Загружается файл в репозиторий и разблокируется (разрешается его скачивание из репозитория другим клиентам).

Во многом, из-за простаты и схожести в управлении с CVS, но в основном, из-за своей широкой функциональности, Subversion с успехом конкурирует с CVS и даже успешно ее вытесняет.

Однако, и у Subversion есть недостатки. Давайте рассмотрим ее слабые и сильные стороны для сравнения с другими системами управления версиями.

Достоинства:

1. Система команд, схожая с CVS.
2. Поддерживается большинство возможностей CVS.
3. Разнообразные графические интерфейсы и удобная работа из консоли.
4. Отслеживается история изменения файлов и каталогов даже после их переименования и перемещения.
5. Высокая эффективность работы, как с текстовыми, так и с бинарными файлами.
6. Встроенная поддержка во многие интегрированные средства разработки, такие как KDevelop, Zend Studio и многие другие.
7. Возможность создания зеркальных копий репозитория.
8. Два типа репозитория – база данных или набор обычных файлов.
9. Возможность доступа к репозиторию через Apache с использованием протокола WebDAV.
10. Наличие удобного механизма создания меток и ветвей проектов.
11. Можно с каждым файлом и директорией связать определенный набор свойств, облегчающий взаимодействие с системой контроля версии.
12. Широкое распространение позволяет быстро решить большинство возникающих проблем, обратившись к данным, накопленным Интернет-сообществом.

Недостатки:

1. Полная копия репозитория хранится на локальном компьютере в скрытых файлах, что требует достаточно большого объема памяти.
2. Существуют проблемы с переименованием файлов, если переименованный локально файл одним клиентом был в это же время изменен другим клиентом и загружен в репозиторий.
3. Слабо поддерживаются операции слияния веток проекта.
4. Сложности с полным удалением информации о файлах попавших в репозиторий, так как в нем всегда остается информация о предыдущих изменениях файла, и непредусмотрено никаких штатных средств для полного удаления данных о файле из репозитория.

Выводы:

Subversion – современная система контроля версий, обладающая широким набором инструментов, позволяющих удовлетворить любые нужды для управления версиями проекта с помощью централизованной системы контроля. В Интернете множество ресурсов посвящено особенностям Subversion, что позволяет быстро и качественно решать все возникающие в ходе работы проблемы.

Простота установки, подготовки к работе и широкие возможности позволяют ставить subversion на одну из лидирующих позиций в конкурентной гонке систем контроля версий.

Система управления версиями Git.

(www.git-scm.com)

С февраля 2002 года для разработки ядра Linux’а большинством программистов стала использоваться система контроля версий BitKeeper. Довольно долгое время с ней не возникало проблем, но в 2005 году Лари МакВоем (разработчик BitKeeper’а) отозвал бесплатную версию программы.

Разрабатывать проект масштаба Linux без мощной и надежной системы контроля версий – невозможно. Одним из кандидатов и наиболее подходящим проектом оказалась система контроля версий Monotone, но Торвальдса Линуса не устроила ее скорость работы. Так как особенности организации Monotone не позволяли значительно увеличить скорость обработки данных, то 3 апреля 2005 года Линус приступил к разработке собственной системы контроля версий – Git.

Практически одновременно с Линусом (на три дня позже), к разработке новой системы контроля версий приступил и Мэтт Макал. Свой проект Мэтт назвал Mercurial, но об этом позже, а сейчас вернемся к распределенной системе контроля версий Git.

Git – это гибкая, распределенная (без единого сервера) система контроля версий, дающая массу возможностей не только разработчикам программных продуктов, но и писателям для изменения, дополнения и отслеживания изменения «рукописей» и сюжетных линий, и учителям для корректировки и развития курса лекций, и администраторам для ведения документации, и для многих других направлений, требующих управления историей изменений.

У каждого разработчика, использующего Git, есть свой локальный репозиторий, позволяющий локально управлять версиями. Затем, сохраненными в локальный репозиторий данными, можно обмениваться с другими пользователями.

Часто при работе с Git создают центральный репозиторий, с которым остальные разработчики синхронизируются. Пример организации системы с центральным репозиторием – это проект разработки ядра Linux’a (http://www.kernel.org).

В этом случае все участники проекта ведут свои локальны разработки и беспрепятственно скачивают обновления из центрального репозитория. Когда необходимые работы отдельными участниками проекта выполнены и отлажены, они, после удостоверения владельцем центрального репозитория в корректности и актуальности проделанной работы, загружают свои изменения в центральный репозиторий.

Наличие локальных репозиторием также значительно повышает надежность хранения данных, так как, если один из репозиториев выйдет из строя, данные могут быть легко восстановлены из других репозиториев.

Работа над версиями проекта в Git может вестись в нескольких ветках, которые затем могут с легкостью полностью или частично объединяться, уничтожаться, откатываться и разрастаться во все новые и новые ветки проекта.

Можно долго обсуждать возможности Git’а, но для краткости и более простого восприятия приведем основные достоинства и недостатки этой системы управления версиями

Достоинства:

1. Надежная система сравнения ревизий и проверки корректности данных, основанные на алгоритме хеширования SHA1 (Secure Hash Algorithm 1).
2. Гибкая система ветвления проектов и слияния веток между собой.
3. Наличие локального репозитория, содержащего полную информацию обо всех изменениях, позволяет вести полноценный локальный контроль версий и заливать в главный репозиторий только полностью прошедшие проверку изменения.
4. Высокая производительность и скорость работы.
5. Удобный и интуитивно понятный набор команд.
6. Множество графических оболочек, позволяющих быстро и качественно вести работы с Git’ом.
7. Возможность делать контрольные точки, в которых данные сохраняются без дельта компрессии, а полностью. Это позволяет уменьшить скорость восстановления данных, так как за основу берется ближайшая контрольная точка, и восстановление идет от нее. Если бы контрольные точки отсутствовали, то восстановление больших проектов могло бы занимать часы.
8. Широкая распространенность, легкая доступность и качественная документация.
9. Гибкость системы позволяет удобно ее настраивать и даже создавать специализированные контроля системы или пользовательские интерфейсы на базе git.
10. Универсальный сетевой доступ с использованием протоколов http, ftp, rsync, ssh и др.

Недостатки:

1. Unix – ориентированность. На данный момент отсутствует зрелая реализация Git, совместимая с другими операционными системами.
2. Возможные (но чрезвычайно низкие) совпадения хеш - кода отличных по содержанию ревизий.
3. Не отслеживается изменение отдельных файлов, а только всего проекта целиком, что может быть неудобно при работе с большими проектами, содержащими множество несвязных файлов.
4. При начальном (первом) создании репозитория и синхронизации его с другими разработчиками, потребуется достаточно длительное время для скачивания данных, особенно, если проект большой, так как требуется скопировать на локальный компьютер весь репозиторий.

Выводы:

Git – гибкая, удобная и мощная система контроля версий, способная удовлетворить абсолютное большинство пользователей. Существующие недостатки постепенно удаляются и не приносят серьезных проблем пользователям. Если вы ведете большой проект, территориально удаленный, и тем более, если часто приходится разрабатывать программное обеспечение, не имея доступа к другим разработчикам (например, вы не хотите терять время при перелете из страны в страну или во время поездки на работу), можно делать любые изменения и сохранять их в локальном репозитории, откатываться, переключаться между ветками и т.д.). Git – один из лидеров систем контроля версий.