О КОНТРОЛЕ ВЕРСИЙ

Система контроля версий (СКВ) — система, регистрирующая изменения в одном или нескольких файлах с тем, чтобы в дальнейшем была возможность вернуться к определённым старым версиям этих файлов. Под версионный контроль можно поместить файлы практически любого типа. СКВ даёт возможность возвращать отдельные файлы к прежнему виду, возвращать к прежнему состоянию весь проект, просматривать происходящие со временем изменения, определять, кто последним вносил изменения во внезапно переставший работать модуль, кто и когда внёс в код какую-то ошибку, и многое другое. Вообще, если, пользуясь СКВ, вы всё испортите или потеряете файлы, всё можно будет легко восстановить. Вдобавок, накладные расходы за всё, что вы получаете, будут очень маленькими.

ЛОКАЛЬНЫЕ СИСТЕМЫ КОНТРОЛЯ ВЕРСИЙ

Многие предпочитают контролировать версии, просто копируя файлы в другой каталог (как правило, добавляя текущую дату к названию каталога). Такой подход очень распространён, потому что прост, но он и чаще даёт сбои. Очень легко забыть, что файлы находятся не в том каталоге, и случайно изменить не тот файл, либо скопировать файлы не туда, куда необходимо, и затереть нужные файлы. Чтобы решить эту проблему, программисты уже давно разработали локальные СКВ с простой базой данных, в которой хранятся все изменения нужных файлов (рисунок 1).

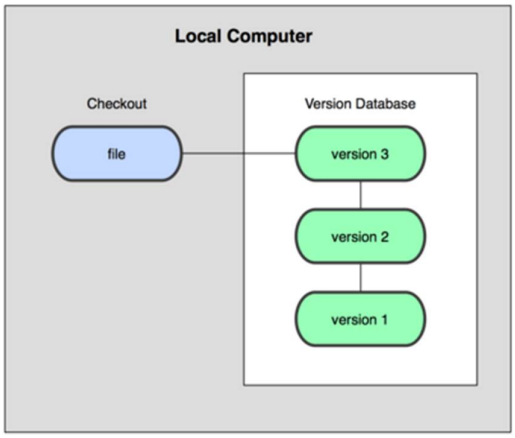

Рисунок 1 – Схема локальной СКВ

Одной из наиболее популярных СКВ такого типа является rcs, которая до сих пор устанавливается на многие компьютеры. Даже в современной операционной системе Mac OS X утилита rcs устанавливается вместе с Developer Tools. Эта утилита основана на работе с наборами патчей между парами версий (патч — файл, описывающий различие между файлами), которые хранятся в специальном формате на диске. Это позволяет пересоздать любой файл на любой момент времени, последовательно накладывая патчи.

ЦЕНТРАЛИЗОВАННЫЕ СИСТЕМЫ КОНТРОЛЯ ВЕРСИЙ

Следующей основной проблемой оказалась необходимость сотрудничать с разработчиками за другими компьютерами. Чтобы решить её, были созданы централизованные системы контроля версий (ЦСКВ). В таких системах, например CVS, Subversion и Perforce, есть центральный сервер, на котором хранятся все файлы под версионным контролем, и ряд клиентов, которые получают копии файлов из него. Много лет это было стандартом для систем контроля версий (рисунок 2).

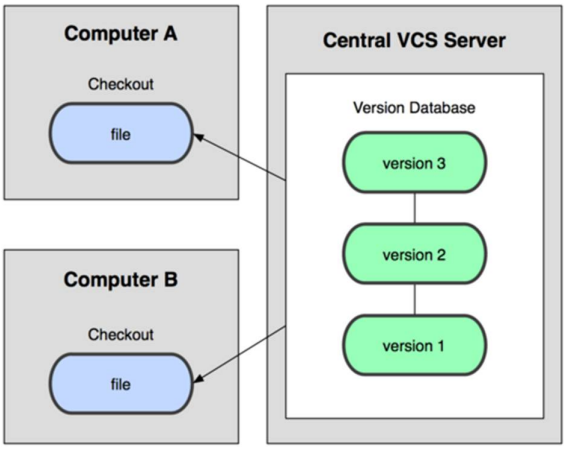

Рисунок 2 – Схема централизованного контроля версий

Такой подход имеет множество преимуществ, особенно над локальными СКВ. К примеру, все знают, кто и чем занимается в проекте. У администраторов есть чёткий контроль над тем, кто и что может делать, и, конечно, администрировать ЦСКВ намного легче, чем локальные базы на каждом клиенте.

Однако при таком подходе есть и несколько серьёзных недостатков. Наиболее очевидный — централизованный сервер является уязвимым местом всей системы. Если сервер выключается на час, то в течение часа разработчики не могут взаимодействовать, и никто не может сохранить новой версии своей работы. Если же повреждается диск с центральной базой данных и нет резервной копии, то теряется абсолютно всё — вся история проекта, разве что за исключением нескольких рабочих версий, сохранившихся на рабочих машинах пользователей. Локальные системы контроля версий подвержены той же проблеме: если вся история проекта хранится в одном месте, то высок риск потерять всё.

РАСПРЕДЕЛЁННЫЕ СИСТЕМЫ КОНТРОЛЯ ВЕРСИЙ

И в этой ситуации в игру вступают распределённые системы контроля версий (РСКВ). В таких системах как Git, Mercurial, Bazaar или Darcs клиенты не просто выгружают последние версии файлов, а полностью копируют весь репозиторий. Поэтому в случае, когда недоступен сервер, через который шла работа, любой клиентский репозиторий может быть

скопирован обратно на сервер, чтобы восстановить базу данных. Каждый раз, когда клиент забирает актуальную версию файлов, он создаёт себе полную копию всех данных (рисунок 3).

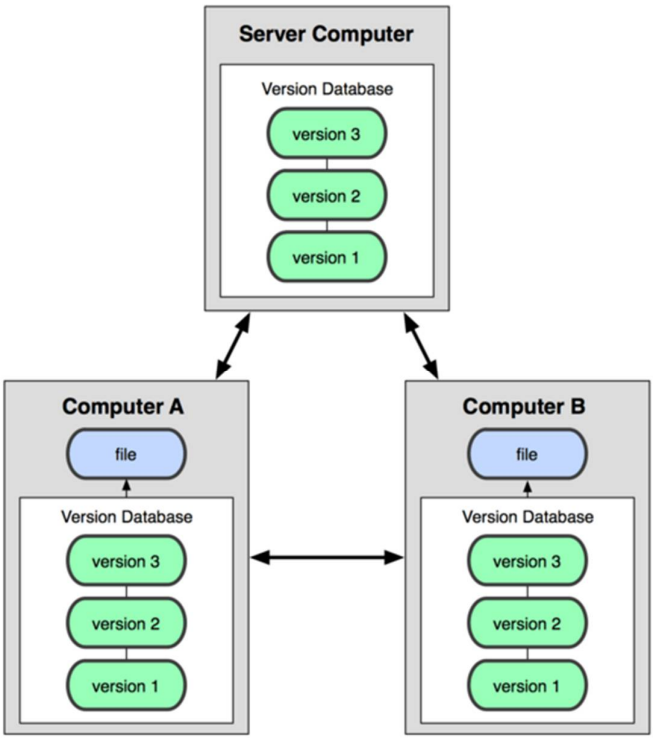

Рисунок 3 – Схема распределённой системы контроля версий

Кроме того, в большей части этих систем можно работать с несколькими удалёнными репозиториями, таким образом, можно одновременно работать по-разному с разными группами людей в рамках одного проекта. Так, в одном проекте можно одновременно вести несколько типов рабочих процессов, что невозможно в централизованных системах.

КРАТКАЯ ИСТОРИЯ GIT

Ядро Linux — очень большой открытый проект. Бо́льшую часть существования ядра Linux (1991-2002) изменения к нему распространялись в виде патчей и заархивированных файлов. В 2002 году проект перешёл на проприетарную РСКВ BitKeeper. В 2005 году отношения между сообществом разработчиков ядра Linux и компанией, разрабатывавшей BitKeeper, испортились, и право бесплатного пользования продуктом было отменено. Это подтолкнуло разработчиков Linux (и в частности Линуса Торвальдса, создателя Linux) разработать собственную систему, основываясь на опыте, полученном за время использования BitKeeper. Основные требования к новой системе были следующими:

- скорость
- простота дизайна
- поддержка нелинейной разработки (тысячи параллельных веток)

- полная распределённость
- возможность эффективной работы с такими большими проектами, как ядро Linux (как по скорости, так и по размеру данных)

С момента рождения в 2005 году Git развивался и эволюционировал, становясь проще и удобнее в использовании, сохраняя при этом свои первоначальные качества. Он невероятно быстр, очень эффективен для больших проектов, а также обладает превосходной системой ветвления для нелинейной разработки.

ОСНОВЫ GIT

Главное отличие Git от любых других СКВ (например, Subversion) — это то, как Git смотрит на свои данные. Большинство других систем хранит информацию как список изменений (патчей) для файлов. Эти системы (CVS, Subversion, Perforce, Bazaar и др.) относятся к хранимым данным как к набору файлов и изменений, сделанных для каждого из этих файлов во времени, как показано на рисунке 4.

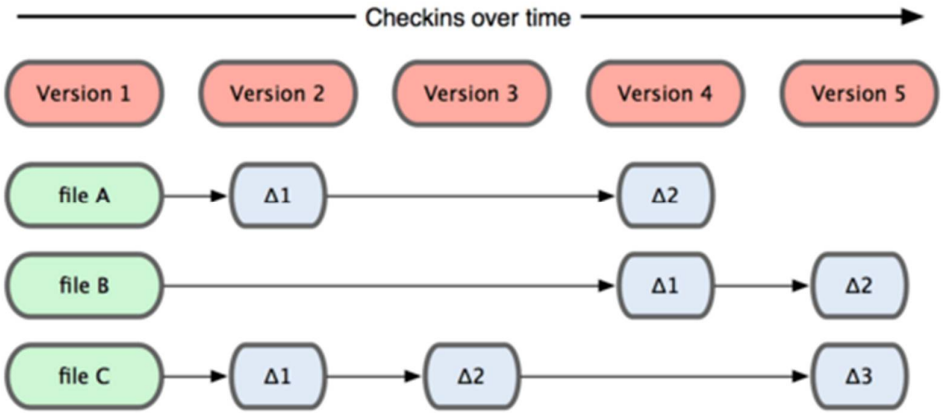

Рисунок 4 –Системы, которые хранят данные как изменения к базовой версии для каждого файла

Git не хранит свои данные в таком виде. Вместо этого Git считает хранимые данные набором слепков небольшой файловой системы. Каждый раз, когда вы фиксируете текущую версию проекта, Git, по сути, сохраняет слепок того, как выглядят все файлы проекта на текущий момент. Ради эффективности, если файл не менялся, Git не сохраняет файл снова, а делает ссылку на ранее сохранённый файл. То, как Git подходит к хранению данных, похоже на рисунок 5.

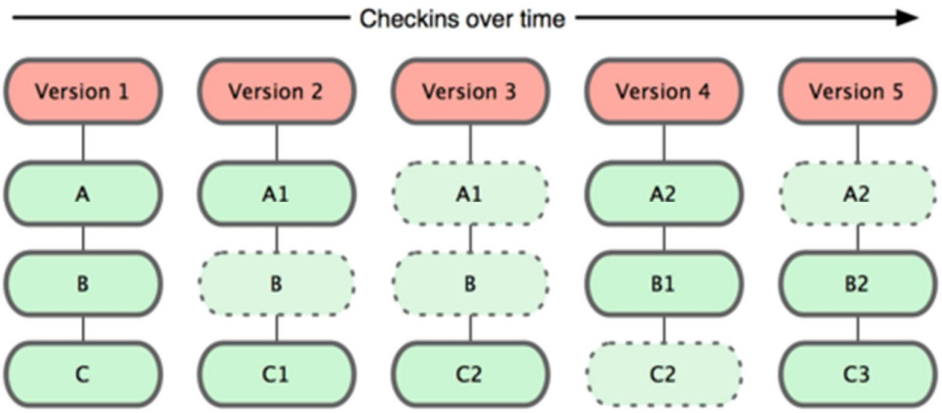

Рисунок 5 – Git хранит данные как слепки состояний проекта во времени

Это важное отличие Git от практически всех других систем контроля версий. Из-за него Git вынужден пересмотреть практически все аспекты контроля версий, которые другие системы переняли от своих предшественниц. Git больше похож на небольшую файловую систему с невероятно мощными инструментами, работающими поверх неё, чем на просто СКВ.

ПОЧТИ ВСЕ ОПЕРАЦИИ — ЛОКАЛЬНЫЕ

Для совершения большинства операций в Git необходимы только локальные файлы и ресурсы, т.е. обычно информация с других компьютеров в сети не нужна. В централизованных системах практически на каждую операцию накладывается сетевая задержка. Поскольку вся история проекта хранится локально у вас на диске, большинство операций кажутся практически мгновенными.

К примеру, чтобы показать историю проекта, Git не нужно скачивать её с сервера, он просто читает её прямо из вашего локального репозитория. Поэтому историю вы увидите практически мгновенно. Если вам нужно просмотреть изменения между текущей версией файла и версией, сделанной месяц назад, Git может взять файл месячной давности и вычислить разницу на месте, вместо того чтобы запрашивать разницу у СКВ-сервера или качать с него старую версию файла и делать локальное сравнение.

Кроме того, работа локально означает, что мало чего нельзя сделать без доступа к Сети или VPN. Если вы в самолёте или в поезде и хотите немного поработать, можно спокойно делать коммиты, а затем отправить их, как только станет доступна сеть. Если вы пришли домой, а VPN-клиент не работает, всё равно можно продолжать работать. Во многих других системах это невозможно или крайне неудобно. Например, используя Perforce, вы мало что можете сделать без соединения с сервером. Работая с Subversion и CVS, вы можете редактировать файлы, но сохранить изменения в вашу базу данных нельзя (потому что она отключена от репозитория).

GIT СЛЕДИТ ЗА ЦЕЛОСТНОСТЬЮ ДАННЫХ

Перед сохранением любого файла Git вычисляет контрольную сумму, и она становится индексом этого файла. Поэтому невозможно изменить содержимое файла или каталога так, чтобы Git не узнал об этом. Эта функциональность встроена в сам фундамент Git. Если информация потеряется при передаче или повредится на диске, Git всегда это выявит.

Механизм, используемый Git для вычисления контрольных сумм, называется SHA-1 хешем. Это строка из 40 шестнадцатеричных символов (0-9 и a-f), вычисляемая в Git на основе содержимого файла или структуры каталога. SHA-1 хеш выглядит примерно так:

```
24b9da6552252987aa493b52f8696cd6d3b00373
```

Работая с Git, вы будете встречать хеши повсюду, поскольку он их очень широко использует. Фактически, в своей базе данных Git сохраняет всё не по именам файлов, а по хешам их содержимого.

ЧАЩЕ ВСЕГО ДАННЫЕ В GIT ТОЛЬКО ДОБАВЛЯЮТСЯ

Практически все действия, которые вы совершаете в Git, только добавляют данные в базу. Очень сложно заставить систему удалить данные или сделать что-то неотменяемое. Можно, как и в любой другой СКВ, потерять данные, которые вы ещё не сохранили, но как только они зафиксированы, их очень сложно потерять, особенно если вы регулярно отправляете изменения в другой репозиторий.

ТРИ СОСТОЯНИЯ

В Git файлы могут находиться в одном из трёх состояний: *зафиксированном, изменённом и подготовленном*. **"Зафиксированный"** значит, что файл уже сохранён в вашей локальной базе. К **изменённым** относятся файлы, которые поменялись, но ещё не были зафиксированы. **Подготовленные** файлы — это изменённые файлы, отмеченные для включения в следующий коммит. Таким образом, в проектах, использующих Git, есть три части: каталог Git (Git directory), рабочий каталог (working directory) и область подготовленных файлов (staging area).

Рисунок – Картина изменения состояний

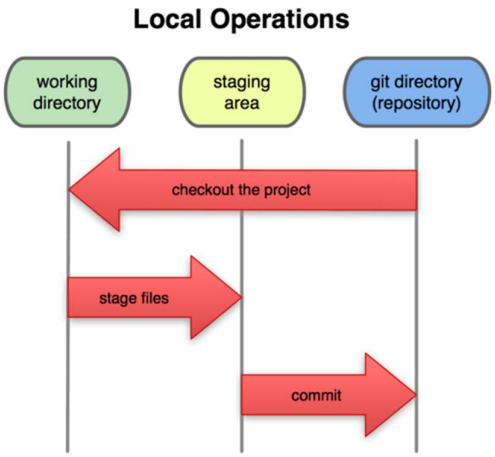

Рисунок 6 – Рабочий каталог, область подготовленных файлов, каталог Git

Каталог Git — место, где Git хранит метаданные и базу данных объектов вашего проекта. Это наиболее важная часть Git, и именно она копируется, когда вы клонируете репозиторий с другого компьютера.

Рабочий каталог — извлечённая из базы копия определённой версии проекта. Эти файлы достаются из сжатой базы данных в каталоге Git и помещаются на диск для того, чтобы вы их просматривали и редактировали.

Область подготовленных файлов — файл, обычно хранящийся в каталоге Git, который содержит информацию о том, что должно войти в следующий коммит. Иногда его называют индексом (index), но в последнее время становится стандартом называть его областью подготовленных файлов (staging area).

Стандартный рабочий процесс с использованием Git выглядит примерно так:

1. Вы вносите изменения в файлы в своём рабочем каталоге.
2. Подготавливаете файлы, добавляя их слепки в область подготовленных файлов.
3. Делаете коммит, который берёт подготовленные файлы из индекса и помещает их в каталог Git на постоянное хранение.

Если рабочая версия файла совпадает с версией в каталоге Git, файл считается зафиксированным. Если файл изменён, но добавлен в область подготовленных данных, он подготовлен. Если же файл изменился после выгрузки из БД, но не был подготовлен, то он считается изменённым.

УСТАНОВКА GIT. УСТАНОВКА ИЗ ИСХОДНИКОВ

Если есть возможность, то, как правило, лучше установить Git из исходных кодов, поскольку так вы получите самую актуальную версию. Каждая новая версия Git обычно включает полезные улучшения пользовательского интерфейса. Для установки Git 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'a:

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, чтобы получить обновления:

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

УСТАНОВКА В WINDOWS

Установить Git в Windows очень просто. У проекта msysGit процедура установки — одна из самых простых. Просто скачайте exe-файл инсталлятора:

http://msysgit.github.com/

После установки у вас будет как консольная версия (включающая SSH-клиент, который пригодится позднее), так и стандартная графическая. Рекомендуется использовать Git только из командой оболочки, входящей в состав msysGit, потому что так вы сможете запускать сложные команды, приведённые в примерах в настоящем руководстве. Командная оболочка Windows использует иной синтаксис, из-за чего примеры в ней могут работать некорректно.