Разпределени системи за управление на код

Зорница Атанасова Костадинова, 4 курс, КН, фн: 80227,
Искрен Ивов Чернев, 4 курс, КН, фн: 80246

5 февруари 2011 г.

Цел на проекта

Да запознае читателя с историята и развитието на системите за управление на код (SCM). Ще бъдат сравнени централизираните и дистрибутираните системи както на високо ниво - предимства, недостатъци, сфери на приложение - така и на ниско ниво - обща архитектура, представяне на данните, използвани алгоритми и структури от данни. Ще бъде обърнато специално внимание на дистрибутираните системи за управление на код, тъй като те са по-нови като концепция и вече успешно заместват централизираните системи във все повече проекти, най-забележимо тези с отворен код, но също и в големи компании които по исторически причини са използвали централизирани системи (google, facebook).

Съдържание

1 Увод **3**

2 История и развитие на SCM **3**
2.1 Ръчно управление 3
2.2 Локално управление 4
2.3 Централизирано управление 4
2.4 Разпределено управление 4

3 Сравнение на архитектурно ниво **5**
3.1 Архитектура на централизираните системи 5
3.2 Архитектура на дистрибутираните системи 9

4 Сравнение на функционално ниво **19**
4.1 Предимства на централизираните системи 19
4.2 Недостатъци на централизираните системи 19
4.3 Предимства на дистрибутираните системи 20
4.4 Недостатъци на дистрибутираните системи 21

5 Заключение **21**

1 Увод

Системите за управление на код (SCM) играят важна роля в процеса на разработване на софтуер. Те съхраняват историята на развитие на файловете като по този начин позволяват на потребителя да прегледа направените промени по различни критерии (времеви период, потребител направил промяната). Правенето на промени по кодовата база също е благоприятствано от факта, че те винаги могат да се игнорират и състоянието на проекта да бъде върнато към предишно стабилно такова. Историята на развитие на проекта може да бъде използвана и от хора интересуващи се от прогреса по проекта (мениджъри, клиенти) с цел създаване на план за по-нататъшно развитие и оценка на свършената работа.

Софтуерните проекти обикновено се развиват в няколко направления:

- едно или няколко стабилни направления - използват се от обикновения потребител;
- направление за тестване (beta версия) - използва се от по-напреднали потребители, които искат да получат нововъведенията колкото се може по-скоро, дори на цената на по-нестабилно поведение;
- направление за развитие - използва се от програмистите докато разработват най-новите промени по кода.

SCM предоставят възможности за управление на отделните направления като по този начин може да се разграничи кои версии от историческото развитие на проекта се използват от програмисти, от тестери и от обикновени потребители.

2 История и развитие на SCM

Нуждата от SCM се появява през $60^{те}$ и $70^{те}$ години на XX век. През това време се появяват първите по-големи софтуерни проекти и става ясно, че поддържането на историята на развитие на проекта е крайно наложително. Развитието преминава през няколко периода:

2.1 Ръчно управление

В началото всеки сам е решавал проблема с пазене на история на проект или отделен файл. Нужен е бил механизъм за запаметяване на състоянието на един или няколко файла в даден момент, за да може при нужда да се върне старо състояние. Това може да става със запазване на стара версия на файла с различно разширение (например `filename.old.1`, `filename.old.2` и т.н.) или създаване на архив на цяла директория. Тези подходи вършат работа при малки начинания, но имат големия недостатък, че изразходват много памет. Ако един файл има дължина 1 Kb и е бил запазен 20 пъти от момента на своето създаване, тогава заеманата памет е приблизително $1000 * 20/2 = 10\text{Kb}$. Това означава, че необходимата памет нараства линейно с броя на запаметените ревизии. На практика обаче всяка ревизия се различава с малко от предходната. Това означава, че независимо от броя на ревизиите заеманата памет теоретично би могла да бъде пропорционална на размера на проекта.

2.2 Локално управление

Първите софтуерни продукти за управление на код са работели на ниво отделен файл. Запазвали са историята на един файл независимо от останалите. Това може да бъде постигнато като за всяка нова ревизия бъде запазена разликата с предишната. Първата система, която реализира това е SCCS[1]. Тя използва механизмът на припокриващи се разлики (interleaved deltas) за да запази различията между версиите на един файл. Системата е била разработвана от Bell Labs и се разпространявала чрез заплащане. RCS[2] е наследник на SCCS и набрала голяма популярност през 80^{те} години, защото била безплатен и усъвършенстван еквивалент на SCCS. RCS също предоставяла възможност за следене на история на всеки файл по отделно. Освен работната версия на файла се пазел и файл с историята, съдържащ списък с промените приложени върху файла от началното до крайното му състояние.

2.3 Централизирано управление

Представените до момента системи дават възможност за пазене на историята на отделни файлове на компютъра на който са били създадени. Възникнала необходимостта тази история да се предоставя на група хора - екип програмисти. Разпределянето на файла с историята между няколко компютъра се оказало нелека задача, особено при наличието на много файлове. Нужна била система която е в състояние да управлява всички файлове на проекта и да осигури лесен и бърз начин за достъпване на историята на проекта от всеки член на екипа. С тези идеи била създадена CVS[3] - една от най-известните системи за управление на код, използвана и в момента от немалко проекти. В нея точно както в RCS е имплементирано запазването на история за всеки файл. Присъства и централен сървър на който се съхранява информация за историята на всички файлове. При клиента се намира само една текуща версия на всички файлове и при необходимост от други версии той се свързва със сървъра и изтегля нужната му информация. За да поправи много от проблемите открити с времето в CVS бил създаден Subversion[4]. Subversion има мото “CVS направен както трябва”. Тъй като решава доста проблеми на CVS и е лесен за използване много проекти мигрират от CVS на Subversion. При Subversion се запазва идеята за един централен сървър с който потребителите се свързват и актуализират файловете си.

2.4 Разпределено управление

Най-съществено различие на дистрибутираните системи за управление на код спрямо централизираните е, че всеки клиент притежава на локалната си машина пълната история на развитие на проекта. Тъй като всеки добавя сам нова история трябва да съществува механизъм за обмяна на направените промените между самите клиента. Това означава, че може лесно да се наподоби моделът на работа на централизираните системи като се избере

[1] Source Code Control System[1]
[2] Revision Control System[2]
[3] Concurrent Versions System[3]

един централен клиент и останалите се синхронизират с него. На практика, в зависимост от ситуацията, се използва както централен клиент, така и синхронизиране на два нецентрални клиента.

Най-известните примери за разпределени системи за управление са Git[5] и Mercurial[6]. Те набират голяма популярност сред open source обществото защото улесняват споделянето на промени между хора които не са се познавали предварително и следователно нямат общ централен сървър.

3 Сравнение на архитектурно ниво

Модел за едновременен достъп Описва как в случай на едновременна редакция на файлове се избягва въвеждането в хранилището на неверни данни.

- `Схема със заключване` – Промени не са позволени докато потребителят не поиска и получи изключителните права над файла от основното хранилище. Файлът е заключен докато трае текущата редакция.
- `Схема със сливане` – Потребителите са свободни да редактират всички файлове, но при опит да добавят промените си в хранилището са информирани за евентуалните конфликти. В такъв случай системата за контрол на версиите може да разреши конфликта или ако не успее да остави потребителят сам да го разреши.

Разпределените системи почти винаги предполагат модела със сливане.

3.1 Архитектура на централизираните системи

3.1.1 Архитектура на CVS

CVS (съкратено от Concurrent Versions System) е една от първите широко достъпни системи за управление на код с архитектура от тип "клиент-сървър"[7]. Сървърът пази информация за цялата история на всички файлове от проекта, а клиентът може да свали произволна версия от сървъра (check-out), да направи локални промени, след което да качи промените на сървъра (check-in).

Системата е паралелна (concurrent) защото позволява на произволен брой програмисти да работят едновременно – да правят промени върху един и същ проект по едно и също време. Единственото условие е при check-in след последния check-out да няма check-in на друг потребител. Ако версията свалена от сървъра е най-новата и не се появи по-нова заради check-in на някой друг, качването е успешно. Ако междувременно е създадена нова версия, то останалите трябва първо да свалят най-актуалната такава, да обединят промените си с нея, след което да запишат своите промени. В повечето случаи тази операция се извършва автоматично от CVS. Проблем

създават само ситуациите при които е редактиран един и същи файл на еднакви места – тогава се налага намесата на програмиста.

Запазване на историята Историята на файловете в CVS repository се запазва за всеки файл поотделно като се използва RCS формат – списък с промените, довели от началното до крайното състояние. Когато се създава нова версия файловете които са променени увеличават своята версия (промяната им се записва във файла с историята). Файловете участващи във всяка версия не са групирани по никакъв начин. Това е проблем ако искаме да видим състоянието на проекта във фиксиран минал момент – всеки файл пази своята история, но информацията за това кои версии на файловете са свързани в ревизиите на целия проект липсва. Това налага ръчното маркиране на ревизии със специални имена, за да може лесно да се възстанови исканото предишно състояние на всички файлове.

Връзка със сървъра При качване на промени към сървъра е възможно част от файловете да бъдат обновени, а останалите не – например при липса на стабилна мрежова свързаност. Това създава проблеми, защото в по-голям проект със сигурност ще се случи, оставя хранилището в невярно състояние, а поправянето става единствено ръчно от администратора.

3.1.2 Архитектура на Subversion

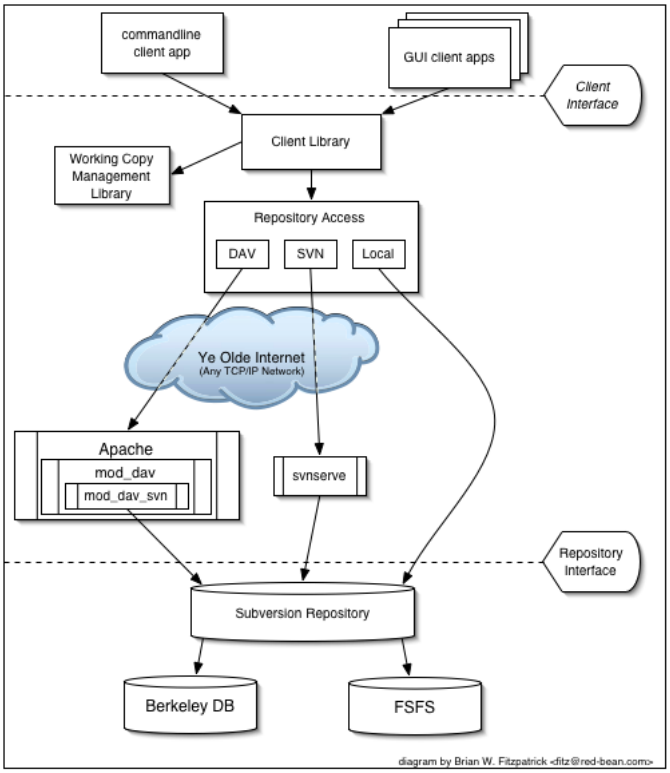
Apache Subversion (или само SVN по името на командата svn) е система за контрол на версиите създадена с цел да бъде силно подобна на CVS, поправяйки грешките и имплементирайки някои липсващи нейни функционалности. Използва се широко в средите почитащи отворения код. Сред примерите са Apache Software Foundation, FreeBSD, GCC, Django, Ruby, Tigris.org, PHP, Python и MediaWiki. Google Code и SourceForge предоставят Subversion хостинг за проекти с отворен код.

Функционалности, които го правят предпочитан пред CVS:

- `Атомарни commit-и` - В смисъла на транзакциите – или цялото множество от промени се записва на сървъра, или абсолютно нищо.
- `Пълна история` на преименовани, копирани или преместени файлове и директории.
- `Мета информация` - Може да се съхраняват мета данни за файлове и директории под формата на двойки ключ - стойност.
- `Двоични файлове` - Съхранението им е ефективно, като от ревизия до ревизия се пази само binary-diff.
- `Branching` е евтина операция независеща от размера на файловете. Механизмът е подобен на хард-линковете в UNIX.

- `Обмен на малко информация` - Протоколът между клиента и сървъра изпраща само diff-ове на файловете в двете посоки.
- `Резултатите` са удобни за парсене например от скриптов език. Има възможност за получаване на log във вид на XML.
- `Конфликти` - За да се избегне необходимостта от разрешаване на конфликти, файловете могат да се заключват. Така програмистът си запазва изключителното право само той да ги редактира (reserved checkout).
- `Интеграция` - SVN е написан на C, но поддържа интеграция също със C#, PHP, Python, Perl, Ruby и Java.

Архитектура Моделът е клиент-сървър, дизайнът на библиотеките е слоест.



В долната част на схемата се намира SVN хранилището, което съдържа всички данни регистрирани за контрол на версиите. В горната част е представена клиентската програма която управлява локалните копия на тези данни (наречени работни копия). Помежду им са множество маршрути през различни слоеве за достъп до repository-то. Някои от тях минават през мрежата и през сървъри, които от своя страна вземат данните от централния сървър. Други нямат нужда да използват мрежата и директно достъпват хранилището.

Видове хранилища

- `Berkeley DB`
- `FSFS` - Предпочитаният метод за съхранение на данните, тъй като работи по-бързо и заема по-малко дисково пространство.

Достъп до хранилищата

- `Местна или мрежова файлова система` — Използва протокола `file:///path`.
- `WebDAV / Delta-V` — Чрез модула `mod_dav_svn` за Apache. Протоколът е http(s).
- `Специализиран svn протокол` — Предава обикновен текст (`svn://host/path`) или криптиран през ssh (`svn+ssh://host/path`).

Клиенти

- В командния ред
  - Subversion предоставя собствен клиент за командния ред.
- В мениджъра на прозорците
  - TortoiseSVN е разширение на Windows shell което информира за състоянието на файловете в revision системата като модифицира иконите в Windows Explorer.
  - SmartSVN работи на подобен принцип.
- Интегрирани в средата за разработване на код[4]
  - AnkhSVN е предвидена за ползване с Microsoft Visual Studio.
  - Subclipse, Subversive работят заедно с Eclipse.
  - Xcode е редактор, включен в Mac OS X 10.5 (Leopard), поддържащ SVN.
- Уеб-базирани
  - Trac - вдъхновена е от CVSTrac и първоначално е наречена "svntrac" заради интеграцията си със Subversion.

Слоеве

- `Файлова система` – Това е най-ниското ниво на което се съхраняват потребителските данни.
- `Хранилище` – Контролира скриптовете които изпълняват действията по контрол на версиите. Тези два слоя заедно съставляват интерфейса на файловата система.
- `mod_dav_svn` – Осигурява WebDAV / Delta-V достъп чрез Apache 2.

[4] IDE — Integrated Development Environment

- `Достъп до хранилището` – Този слой контролира както местния така и отдалечения достъп. На това ниво към хранилищата се обръщаме чрез URL:
  - `местен достъп` — `file:///path/`
  - `WebDAV` — `http://host/path/`, `https://host/path/`
  - `протокол SVN` — `svn://host/path/`, `svn+ssh://host/path/`
- `Клиент, Работно копие на проекта` – Най-високото ниво абстрахира достъпа до хранилището и предоставя интерфейс за общи клиентски задачи като автентикация на потребители и сравнение на версии.

3.2 Архитектура на дистрибутираните системи

3.2.1 Архитектура на Mercurial[8]

Обща информация Mercurial е многоплатформена, дистрибутирана система за управление на код. Тя е написана главно на Python, но съдържа алгоритъм за двоично запазване на разликите разработен на C. Mercurial работи на всички основни операционни системи: Windows, Unix-like – Linux, OS X, *BSD[5]. При разработването на Mercurial основни цели са били:

- производителност
- скалируемост
- децентрализация
- пълна дистрибутираност
- с адекватно обработване на текстови и двоични файлове
- развити средства за разклоняване и сливане
- идейно проста

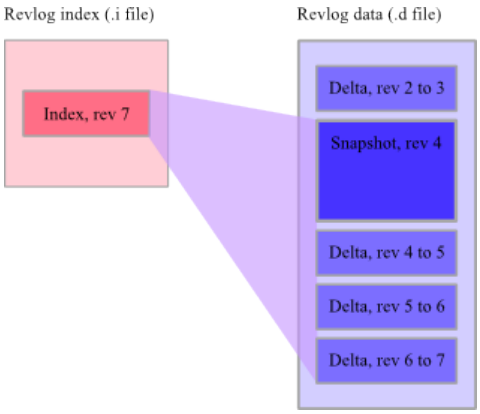
Основни концепции

revlog Mercurial пази историята на всеки файл в `revlog` файлове – един индекс и един за данни. Във файла с данни са записани или последователни разлики, или целия файл. Това зависи от обемът данни, които ще са необходими за да се възстанови ревизията при нужда – както видео кодерите запазват цял фрейм, последван от няколко разлики с предишния фрейм. Данните се компресират ако компресираният вариант е по-малък от оригинала. В индекс файла са записани местата във файла с данни в който

[5] NetBSD, FreeBSD, OpenBSD

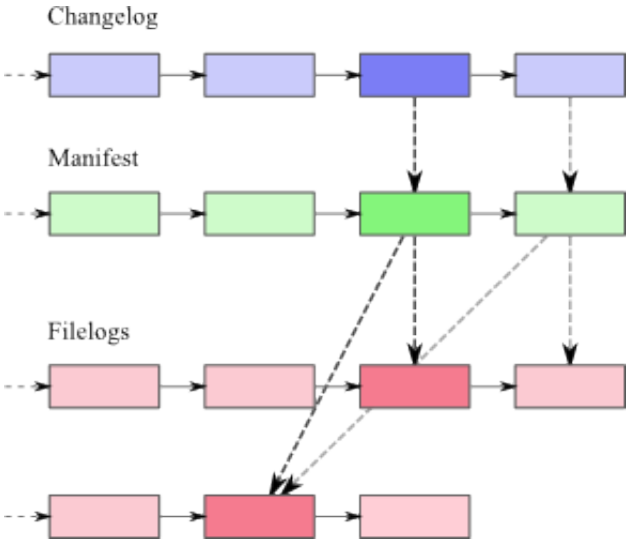
започва описанието на всяка нова версия (било то разлика с предходната или целия файл), заедно с мета данни.

Всяка ревизия на всеки файл има уникален идентификатор наречен `nodeid`. Той се получава като се сметне SHA1[9] сумата на файла и се конкатенира с `nodeid`-то на предишната версия на файла. Това гарантира, че всеки файл в процеса на своето разработване ще получава за различните си версии винаги различни `nodeid` дори съдържанието на файла да е напълно еднакво.



manifest Информацията за това кои версии на всеки файл участват в определена ревизия на проекта се запазва в manifest файл. Една и съща версия на файл може да участва в няколко последователни версии на manifest-а, ако файлът не е бил променян в тези ревизии на проекта. Информацията се пази в двоичен формат и представлява `nodeid` на версията на файла заедно с пълното му име. Тъй като manifest-ът е файл който се променя с времето, различните му ревизии се пазят в revlog – точно както се пазят промените в обикновените файлове. Това е удачно, тъй като големи части от manifest файла сочат към едни и същи версии на файловете, защото сравнително малко файлове биват променени при всяка нова версия на целия проект. Така последователните ревизии на манифеста се различават минимално, което значително намалява необходимата памет за запазване на всички версии.

changelog Информацията за самите ревизии на проекта се съхраняват в changelog. Той съдържа датата, потребител направил ревизията, предишна ревизия (или ревизии, в случай на сливане), списък с променените файлове и `nodeid` на manifest файла, съдържащ информация за файловите версии на тази ревизия. Форматът на changelog файла също е revlog.



dirstate Състоянието на текущата директория се пази в dirstate. То се използва при записване на промените при създаване на нова ревизия. Пази се кога за последно Mercurial е променял всеки файл и колко е бил размера на файла в този момент. Когато потребителят се интересува от състоянието на работната директория (посредством `hg status`) Mercurial проверява последното време на промяна на всеки файл в проекта и го сравнява с информацията в dirstate. Ако timestamp-овете съвпадат се приема, че файлът не е бил променян. Ако не съвпадат се гледа текущия размер на файла. Ако размера е различен значи файлът е бил променен. Само в такъв случай се налага сравняване на целия файл с последната ревизия запазена от Mercurial.

Начин на работа Когато се наложи да се разгледа състоянието на проекта в определена (минала) ревизия – например с командата `hg update`, се случват следните неща:

1. прочита се индекс файлът на changelog и се намира мястото в data файла където стои информация за дадената ревизия на changelog-а
2. възстановява се желаната ревизия от changelog data файла – това включва акумулирането на няколко разлики върху пълна версия[6] – всички взети от data файла
3. прочита се nodeid на manifest-а от changelog за съответната версия
4. възстановява се manifest-а със съответното nodeid, отново чрез използване на index и data файловете му
5. от manifest-а се взима информация за nodeid на всички файлове участвали в проекта за фиксираната версия

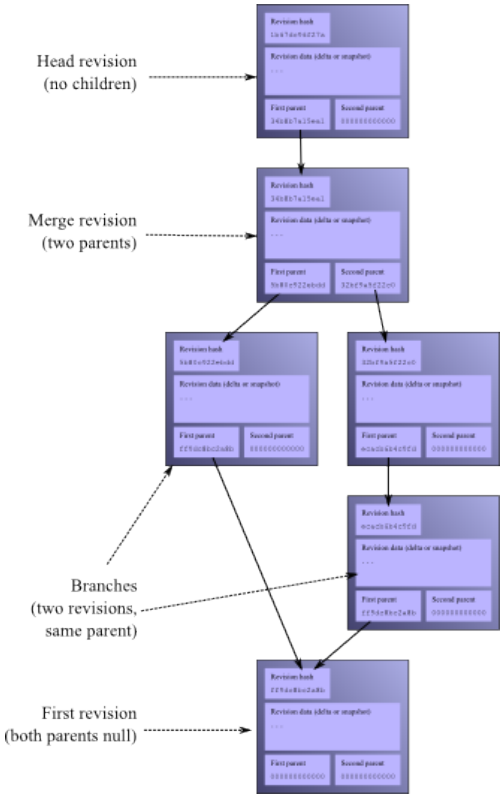
[6] snapshot

6. от index и data файловете за всеки файл посредством nodeid се възстановяват и самите файлове и се записват в текущата работна директория

Разклонения и сливания Основна разлика между дистрибутираните и централизираните системи е лекотата с която дистрибутираните се справят с разклоненията и сливанията в проекта. При централизираните системи всяка ревизия на проект има точно една предишна. Това налага проектът да се развива праволинейно. При дистрибутираните системи е често срещано много хора едновременно да работят върху една и съща начална ревизия (всеки от тях създава собствено разклонение). Това означава, че всички техни промени няма да включват промените на всички останали. Това се случва често и следователно е нужен механизъм по който да бъдат обединени различните промени, за да се получи версия на проекта, в която всички промени са извършени (това е сливане).

В Mercurial всяка ревизия на проекта може да има един или два предшественика. Само един предшественик означава, че върху някаква ревизия са направени промени и така се е образувала нова ревизия (повечето ревизии в един стандартен проект са точно такива). При наличие на ревизии с по един предшественик е възможно дървото на ревизиите да се разклони. Това става когато две промени са базирани на една и съща начална ревизия. За да се слеят две ревизии които споделят обща бащина ревизия се използва нова ревизия с два предшественика. На базата на графа на ревизиите Mercurial успява да слее повечето файлове без нужда от програмистка намеса. Преглежда се какви промени са правени по пътя на първия предшественик и какви промени са правени по пътя на втория предшественик до най-близката обща ревизия. Ако промените са по не пресичащи се части от файла може да бъде направено автоматично сливане[7]. Ако промените засягат едни и същи части на даден файл – например по едната верига е изтрит даден ред, а по другата той е модифициран, се налага потребителят да редактира файла на ръка.

[7] auto merge



Компресия Когато Mercurial запазва файлове на диска се използва deflate[10] алгоритъма, който дава добър баланс между използвано място и време за компресиране / декомпресиране. Ако обаче се предават данни по мрежата компресираното съдържание се декомпресира, след което се компресира отново с друг алгоритъм, който е по-бавен, но за сметка на това по-ефективен. Компресира се целият поток от данни което е по-ефективно по време и памет отколкото компресиране на отделните файлове. Ако за преноса на данни се използва `ssh`[11], то данните въобще не се компресират, защото `ssh` има вградена поддръжка за компресиране. С това Mercurial постига по-добра ефективност върху повече видове мрежи.

Нареждания на входно/изходните операции и атомарност Mercurial гарантира, че клиент четящ от хранилището никога няма да види частични данни въведени от клиент който записва по същото време. Това означава, че във всеки един момент може да има един пишещ клиент и много четящи, което значително подобрява качеството на достъпа. За да се постигне това се използват няколко техники:

файловете се дописват само в края Както вече стана ясно, цялата информация в едно repository се съхранява в множество revlog файлове. Записването на нова информация в тях (добавянето на нова ревизия) е позволено да се извършва само в края на файла. Това подобрява производителността (пред методи, презаписващи цялото съдържание) и гарантира

атомарност на операциите.

подреждане на входно/изходните операции Когато се записва нова ревизия на проекта, редът на записване е следният:

- записват се revlog data файловете на променените файлове от проекта
- записват се revlog index файловете на променените файлове от проекта
- записва се revlog data файла на манифеста
- записва се revlog index файла на манифеста
- записва се revlog data файла на changelog
- записва се revlog index файла на changelog

Както вече обяснихме, редът при четене е обратен – това означава, че клиент четящ в даден момент от хранилището прочита данни сочещи (чрез nodeid) към вече записани данни в следващия файл.

едновременен достъп Подреждането на входно / изходните операции гарантира, че четящ потребител няма нужда да заключва repository-то. Това е и класически механизъм за защита срещу deadlock. Това означава, че четящите потребители имат само нужда от права за четене, което значително улеснява администрирането на repository-то.

Пишещите клиенти трябва да заключват repository-то. Механизмът за заключване работи на всякакви файлове системи, включително NFS[12], която по принцип създава всевъзможни проблеми със заключванията. Ако пишещ клиент завари заключено repository, той изчаква определено време, проверявайки периодично дали се е отключило. Ако времето изтече и repository-то още е заключено, процесът завършва с неуспех. Това е полезно, ако са настроени скриптове които изпълняват периодичен запис – няма начин нещо да забие и акумулиращите чакащи за запис скриптове да изразходят ресурсите на системата.

3.2.2 Архитектура на Git [13]

Git е създадена от Линус Торвалдс за целите на разработването на ядрото на операционната система Линукс. При писането ѝ се е обърнало специално внимание на скоростта и лесното включване на код от много участници в проекта. Всяка работна директория в Git е пълноправно хранилище с пълна история и възможности за преглеждането ѝ, независещи от мрежови достъп до централен сървър.

Дизайн Вдъхновен от BitKeeper и Monotone, Git е бил замислен като engine за система за контрол на версиите на ниско ниво. Над него други щели да надграждат front-end. Проектът обаче се развил дотолкова, че е напълно функционален и за директно използване. Създателят на Git има широки познания върху работата на файловите системи и опит с поддръжката на голям разпределен проект какъвто е операционната система Линукс. Това предопределя следните решения:

- `Разпределена разработка` – Git предоставя на всеки разработчик местно копие на цялата база. Промените се включват от едно хранилище в друго като нови разклонения, които могат да бъдат слени по същия начин като местните разклонения.
- `Ефективност при работа с големи проекти` – По думите на създателя си, Git е бърз и скалируем. Тестове проведени от Mozilla доказват, че е в пъти по-бърз от повечето системи за контрол на версиите, а достъпът до историята на ревизиите от местното хранилище е 2 пъти по-бърз отколкото достъпването им през отдалечен сървър. Git не става по-бавен с увеличаването на кодовата база или акумулираната история на ревизиите.
- `Сериозна поддръжка при нелинейна разработка` – Важно допускане при дизайна на Git е, че една промяна по-често ще бъде присъединявана към проектите на участниците отколкото редактирана. Системата поддържа бързо разклоняване и сливане на код и предоставя инструменти за визуализиране и управление на сложна история на разработка.
- `Съвместимост с протоколи и системи`
  Медия
  - `ssh`
  - `sockets`

  Протоколи
  - `http`
  - `ftp`
  - `git`
  - `rsync`

  Други системи
  - `cvs` — Git има възможност за емулиране на CVS сървър и позволява достъпа до хранилища чрез съществуващи cvs клиенти и плъгини за среди за разработка.
  - `svn` — Съществуващи subversion хранилища могат да се достъпват с командата git-svn.

- `Криптографска автентикация на историята` – Името на всеки commit зависи от (криптографската SHA1 сума на) пълната история, довела до него. Веднъж публикуван commit-а не е възможно да се промени без това да се отрази на хеш сумата, а следователно и на името на ревизията. В този смисъл промените по Git са немодифицируеми[8].

- `Модулен дизайн` – Git е създаден като множество програми на C, свързани с shell скриптове - wrapper-и около тези програми. Макар и сега повечето от скриптовете да са пренаписани на C като част от усилието да се направи Git напълно функционален под Windows, дизайнът е останал, което прави лесно навързването и замяната на отделни компоненти.

- `Стратегиите за сливане са лесно заменими` – Като част от модулния дизайн, Git ясно дефинира незавършено сливане и има множество алгоритми за справяне с него. В случай на невъзможност да се слеят автоматично промените е необходима ръчна редакция от страна на потребителя.

- `BitKeeper[14] като модел` – Използваната в първите години на разработване на Линукс ядрото система е модел за интерфейса и потока на действия в Git. Технически двете са много различни, което е била една от целите които създателят на Git си е поставил за да бъде очевидно, че макар и създаден да го замести, Git не е копие на BitKeeper.

- `CVS като анти-модел` – Централизираната система се оказва неприложима за целите на Линукс ядрото.

- `Сигурност` – Гаранции срещу повреда на данните, била тя случайна или злонамерена.

Имплементация

blob Всички файлове в едно Git repository се представят вътрешно чрез blob. Най-важният атрибут на един blob е неговата SHA1 сума. Ако два blob-а имат една и съща SHA1 сума repository-то ще пази само един обект. В blob се намира само съдържанието на файла. Това означава, че два файла с различно име имащи еднакво съдържание ще се представят вътрешно като точно един blob.

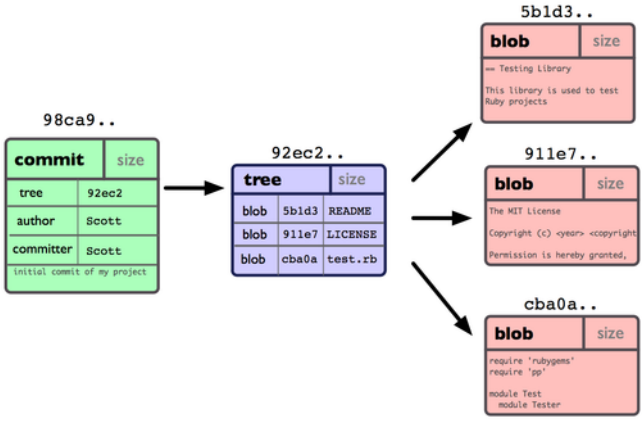
tree Директориите се пазят в структура наречена tree. Тя съдържа информация за обектите които се съдържат в нея – списък от тип на обекта, SHA1 сума и име. В tree обектите се помещават други tree обекти и blob обекти.

commit Създаването на нова ревизия на проекта се отбелязва в commit обект. Той съдържа:

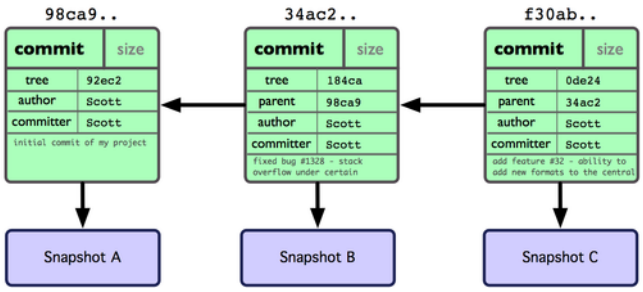
- SHA1 сумата на главната директория на проекта (tree обект)

[8] immutable

- SHA1 суми на предшестващи commit обекти – в случай на сливане е повече от един
- информация за автора на ревизията
- информация за човека създаващ ревизията в самото repository (различен е от автора ако например е получил patch от него и в момента го записва като нова ревизия)
- съобщение за новата ревизия



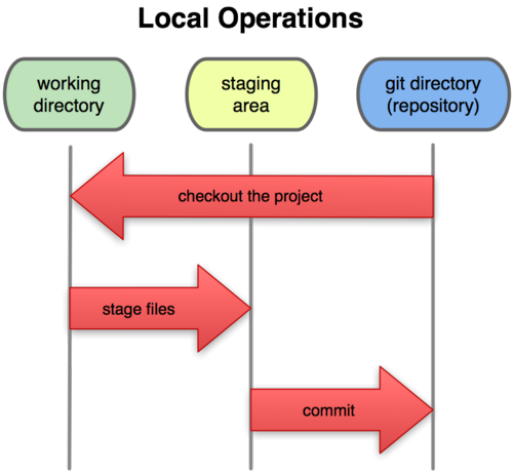
разлики vs snapshot За разлика от почти всички останали системи за управление на код, Git разглежда отделните ревизии на проекта като съвкупност от файловете и директориите му. Идеята е да се пази архив на главната директория за всяка отделна версия. Останалите системи опитват да използват умни алгоритми за следене на файлове по отделните ревизии, като запазват или пълната им версия, или разликата[9] с предходната. На практика следенето на разликите на ниво файл не винаги е достатъчно – ако няколко реда се преместят от един файл в друг, то в първия файл те ще се отбележат като изтрити, а във втория - като добавени, но физическа връзка между тях никога няма да съществува. В Git промените се засичат на ниво repository от сложни евристични алгоритми. Следователно системата може лесно да се справи с проблеми които биха възникнали от per file следене на промени.



[9] changeset

запазване на blob обекти Git запазва blob обектите по два начина – опакован и не опакован. При не опакования вариант всеки blob се архивира и записва върху диска като отделен файл. Това е стандартния начин по който се запиват всички blob обекти при нормална работа с repository-то. Това прави git толкова бърз – не се налага да се изчисляват разлики с предишни версии и да се акумулират при поискване на определена версия. Ако продължително се използва repository-то то ще увеличи много размера си. За да се справи с този проблем Git въвежда команда с която се оптимизира размера (`git gc`). Тя прилага сложни евристични алгоритми за да намери добър компромис между заето място и време за достъп. За намаляване на размера се използват разлики с предишни версии – точно както почти всички други системи. Разликата е, че в Git това не е основен идиом а е по-скоро оптимизация, която се прилага само при нужда от алгоритъм, имащ информация за всички файлове и ревизии, а не само за един файл. На практика с тази допълнителна информация алгоритъма може да се справи много по-добре.

staging area Staging area или index е мястото където се натрупват промените преди те да бъдат commit-нати (пази се tree обект на основната директория). Това е различно от работната директория, която съдържа файловете с които потребителят работи. Полезно е когато са направени няколко логически несвързани промени които трябва да се запазят като отделни ревизии. Index-ът позволява да се изберат файловете които ще се запишат или дори отделните промени в тях (на ниво hunk). Staging area съдържа и мета информация позволяваща на Git да намери бързо разликите между работната директория и tree обекта който ще бъде записан (например време на модификация и размер, подобно на Mercurial). В случай на сливане index-ът пази и информация за бързо разрешаване на конфликти на ниво файл и на ниво директория.



4 Сравнение на функционално ниво

Сравнение на разгледаните SCM

Система	Интерактивен commit	Потребителят избира mergetool	Атомарен commit	Промените са на ниво	Схема на ревизиите
CVS	не	не	не	файл	множество промени
SVN	не	да	да	дърво	множество промени
Git	да	да	да	дърво	копие на състоянието
Mercurial	да	да	да	дърво	множество промени

4.1 Предимства на централизираните системи

- Познати са на много голям брой програмисти.
- Съществува изобилие от клиенти (графични, конзолни, web).
- Имат много добра поддръжка в други приложения (IDE-та).
- Концепция им е проста.
- Многократно са доказани в разработване на големи, комерсиални проекти.
- Недостатъците им и методите за справяне с тях са добре известни.

4.2 Недостатъци на централизираните системи

- Създаването на ново repository изисква повече време и посветен на това сървър.
- При проблеми със сървъра целия процес на разработка не може да продължи.
- Включването на външен човек към екипа е тромаво и несигурно – трябва да му се дадат привилегии за писане, което му дава неограничени права. Това е особено неприятно при проектите с отворен код, които разчитат най-вече на доброволци.
- Основните операции са в пъти по-бавни от еквивалентите им при дистрибутираните системи.
- При повечето централизирани системи няма утвърден начин за защита на кода от злонамерени и непреднамерени промени.
- Създаването на разклонения и сливането им е бавен, сложен, неефективен и за това избягван процес.

4.3 Предимства на дистрибутираните системи

- Позволяват ефективна работа дори когато потребителите не са свързани в мрежа.
- Включването в проект е лесно - не изисква позволение за писане от привилегировани потребители.
- Използването за лични проекти е лесно и удобно. Подходящи са за началните фази на проект, когато потребителите все още нямат нищо готово за публикуване.
- Не се създава единична централизирана контролираща система, която да създаде проблем в случай на срив. Всяко работно копие на базата може да служи за отдалечен backup на базата и историята на промените ѝ, намаляващо риска от загуба на данни.
- Все пак се позволява централизиран контрол когато е необходимо да се издаде официална версия на проекта. Възможно е да се прилага и напълно централизиран модел на употреба.
- Повечето операции са много по-бързи отколкото в централизираните системи, тъй като не използват мрежата.
- Създаването на ново repository е изключително лесен и бърз процес. Това прави дистрибутираните системи много удобни за малки проекти.
- Разклоняването и сливането е естествен и утвърден процес. Това означава, че програмистите ще са по-склонни да започват експериментални промени отколкото при централизирано repository.

Отворени системи Дистрибутираните VCS са подходящи за използване от отворени системи заради:

- `независимите разклонения`
- `лесното сливане на отдалечените хранилища`

При отворените системи:

- Всяко разклонение практически се имплементира като работно копие. Сливанията се извършват чрез размяна на patch-ове между отделните разклонения.
- Програмистите имат по-голяма готовност да създават нови версии на проекта[10] когато е необходимо. Всъщност работното копие на хранилището само по себе си е потенциална нова версия. В случай, че неразбирателствата се изгладят и се стигне до консенсус, обединяването на кода в едно е лесно.
- Възможно е да се вземат само отделни промени[11], подбрани от различни потребители.

[10] forks
[11] cherry-picking

- Не е задължително да съществува основно отправно и "вярно" копие на базата с код. Вместо това съществуват множество работни копия. В този ред на мисли е лесно да се създадат и множество "централни" хранилища.
- Код от отделните хранилища се слива на базата на т.нар. "мрежа на доверието"[12]. Исторически е доказано, че това подобрява качеството на кода.

4.4 Недостатъци на дистрибутираните системи

- Начинът на работа на дистрибутираните хранилища е по-сложен, което води и до по-сложен интерфейс на инструментите за работа с тях.
- В момента дистрибутираният подход не е набрал голяма популярност, затова в нов проект трябва да се инвестира повече време за обучение на екипа.
- Съществуват среди за разработка (IDE) в които не са имплементирани разширения за комуникация с дистрибутирани хранилища.
- Първоначалното клониране на хранилището на локалната машина е по-бавно в сравнение с централизираните системи тъй като се копират всички разклонения с цялата им история.
- Липсват механизми за заключване, които традиционно са част от централизираните системи. Това е важно при двоични файлове, които не подлежат на сливане (например изображения, музика, изходни изпълними файлове).

5 Заключение

Системите за управление на код са изминали дълъг път от първоначалното си създаване. Както централизираните системи са взели поуки от своите предшественици и са се адаптирали към нуждите на общността по онова време, така и разпределените системи са замислени като последователи на централизираните, обогатяващи възможностите на потребителя и вземащи предвид изискванията на open source обществото. Познатите модели на работа от централизираните системи са приложими и даже предпочитани при работа в големи екипи, но те не ограничават архитектурата на системата, за да може да се използва и в други среди.

[12] web of trust

Литература

[1] http://sccs.berlios.de/

[2] http://www.gnu.org/software/rcs/

[3] http://savannah.nongnu.org/projects/cvs

[4] http://subversion.apache.org/

[5] http://git-scm.com/

[6] http://mercurial.selenic.com/

[7] http://en.wikipedia.org/wiki/Client-server

[8] http://mercurial.selenic.com/wiki/Design

[9] http://en.wikipedia.org/wiki/SHA-1

[10] http://en.wikipedia.org/wiki/DEFLATE

[11] http://en.wikipedia.org/wiki/Secure_Shell

[12] http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.14.473

[13] http://ftp.newartisans.com/pub/git.from.bottom.up.pdf

[14] http://www.bitkeeper.com/