Лекция 4 (25. Октомври, 2011)

Връзки

"The problem with quotes on the Internet is that you can't always be sure of their authenticity."
Abraham Lincoln

http://en.wikipedia.org/wiki/Revision_control
http://en.wikipedia.org/wiki/Comparison_of_revision_control_software
http://en.wikipedia.org/wiki/Distributed_revision_control_system
http://en.wikipedia.org/wiki/Git_(software)
http://en.wikipedia.org/wiki/Mercurial
http://en.wikipedia.org/wiki/Pseudorandom_number_generator
http://en.wikipedia.org/wiki/Linear_congruential_generator
http://www.cplusplus.com/reference/clibrary/cstdlib/rand/
http://download.oracle.com/javase/1.4.2/docs/api/java/util/Random.html

Първи час

Поради важната си роля в разработването на софтуера, както казахме миналия път, source control системите са широко-разпространени както в малките, така и в големите софтуерни компании. Освен SVN, който вече разгледахме, съществуват редица негови алтернативи, които се различават освен откъм user interface, така и в цялостната си идеология как точно да се осъществява процесът на пазене на данните. Няколко от най-основните системи за контрол на кода са Subversion, CVS, Git, Mercurial, SourceSafe, Team Foundation Server, Bazaar, BitKeeper. Може би най-фундаменталното разграничаване между тях се базира на това дали операциите се извършват на централизиран сървър (Centralized Version Control Systems) или не (Distributed Version Control Systems). От горните примери централизирани са Subversion, CVS, Microsoft-ските SourceSafe и Team Foundation Server. От друга страна, децентрализирани са Git, Mercurial, Bazaar и BitKeeper.

Нека разгледаме какви са основните разлики при един децентрализиран модел в сравнение с досега разглеждания централизиран такъв. Основната идея на дистрибутирания тип системи за контрол на кода е да се извършва възможно най-малко интеракция използвайки мрежата. Това налага пазенето на цялото repository при всеки от програмистите. Разполагайки с цялата информация на своя собствен компютър, времето за достъп до сорсовете, логовете, и всъщност почти всички възможни операции, става мигновено. Така всеки от програмистите прави промени върхо свое собствено repository, като така възникват по два вида commit и update – такива към/от локалното репозитори, и такива към/от нечие чуждо repository (всички други репозиторита). Това дублиране се налага естествено от самата идеология на дистрибутираните version control системи. Въвеждат се термините push и fetch (вместо commit и update съответно) за качване и сваляне на локалната версия на repository-то към тези на останалите участници в проекта. Забележете, че отново може да се реализира „централизиран" сървър като просто този сървър се включи като фиктивен разработчик. Така се получава хибриден модел между централизирания и децентрализирания модел. Този начин на складиране на информацията се радва на редица предимства пред

традиционния метод с централизиран сървър. Някои от тях са:

1. Сигурност: Няма опасност централния сървър да изгори и данните да се загубят безвъзвратно – все пак такъв сървър няма – всеки от участниците в проекта може да възстанови изгубените данни, тъй като пази версия на repository-то (с евентуално малки промени).
2. Бързина: Както вече споменахме, почти всички операции не изискват интеракция с мрежата и съответно се извършват много по-бързо.
3. Контрол на достъпа: Всеки е администратор на собствената версия на repository-то. Тоест той може да налага специфични правила кой може да commit-ва там и кой не; също така той може да извършва произволни промени без code review от администратор – такова ще се наложат чак при обединяването на неговото repository с нечие друго (или централно такова, ако е реализиран хибридният модел).
4. В този модел няма нужда от branch-ове тъй като всичко е branch. Това може да се окаже предимство ако различни екипи разработват различни части от кода при съществуването на отделни branch-ове програмист от даден екип може лесно да merge-ва своите нововъведения само с кода на остасналите хора от своя team, а не с кода на всички останали (като при централизирания модел). Това, разбира се, може да бъде реализирано и чрез branch в централизирания модел, но би станало по-сложно.
5. Реализирана е по-добра организация за пазене на версиите, което спестява известно място на харддисковете. Докато при SVN всяка версия е реализирана като отделен файл с промените (всъщност това е малко остаряла информация, в момента се пазят по 1000 версии в един файл), то в някои от децентрализираните алтернативи има само няколко файла, които се грижат за ВСИЧКИ версии, независимо колко много са те (примерно в Git има 2 файла, които се грижат за versioning-а).

Version Control системите с централизиран сървър също обаче имат своите предимства пред дистрибутираните такива:

1. Леснота на ползване: Отчасти поради това, че са възникнали по-рано, отчасти защото това е била идеята им, системи като Subversion и CVS са значително по-прости за употреба.
2. Масово разпространение: предимно заради точка 1 повечето централизирани системи за контрол на кода могат да се похвалят със значителен брой потребители (съответно фирми, които ги използват).
3. Интеграция в IDE-та: Поради точка 2 редица development среди (примерно Eclipse, DevC++, и т.н.) могат да интегрират в себе си пряк достъп до някоя от тези системи. Това опростява (и ускорява) значително работата с тях, като прави ползването им доста по-удобно от стандартния начин.
4. Интеграция в OS: Отново поради точка 2 има варианти за интеграция на Subversion директно в операционната система (пример: TortoiseSVN, който се интегрира в windows explorer). Като цяло User Interface-ът на, примерно, Git, е много далеч откъм красота и удобство от някои от предлаганите интерфейси за SVN дори в операционни системи, различни от Windows.

5. Единствено repository: макар и да носи някои недостатъци, е доста по-интуитивно за ползване – програмистът винаги знае къде е всичко и е сигурен къде да commit-ва. Ако му трябва последната версия на софтуера знае къде да я намери.
6. Контрол на достъпа: Тук предимството е, че шефовете на фирмата (или администраторите на проекта) могат да наложат строги правила кое да се променя и кое не (както и от кого). Също така е нужна една единствена промяна при приемане на нова политика за достъп на даден файл, директория или branch. В другия вариант тя трябва да бъде apply-ната във всяко от repository-тата.
7. Частичен checkout: едно важно предимство (особено при големи проекти) е възможността за частичен checkout при централизираните revision control системи. Примерно, ако голям проект има версии за Windows/Linux/Mac и вие искате да променяте нещо само по Windows частта, можете да я изтеглите без изобщо да докосвате тези за Linux или Mac. Това е невъзможно при дистрибутирания вариант, където трябва да имате цялото repository при себе си.
8. Номериране на ревизиите: при недистрибутираните системи за контрол на кода е доста по-ясно какво значат номерата на версиите. Примерно ако имате revision 42 (от, да кажем, 1337) ще ви е ясно, че това е една от първите версии на проекта, докато ако имате revision 1313 то това би била една от последните. При дистрибутирания вариант такова пазене на версии би било невъзможно, затова се ползва някакъв вид хеш на данните в ревизията (SHA-1 за Git и Mercurial). Този вид начин за revision numbering не носи почти никаква информация за самата ревизия – ако ви дадат просто два хеша много трудно можете да определите дори нещо елементарно като това коя ревизия е по-нова, какво да говорим за това колко по-нова и приблизително откога е.

Няколко думи за Git

Създаден е от небезизвестния Линус Торвалдс (човекът, създал Linux). Името произлиза от жаргонната дума „git", която може да се преведе нещо от сорта на „гад" (trivia: самият Торвалдс казва: "I'm an egotistical bastard, and I name all my projects after myself. First Linux, now git.").

Проектът е Open Source и се разработва активно. Всяка работна Git директория е пълно repository с цялостна история, която не зависи от достъп до мрежата. Разработен е с идеята да поддържа непоследователна разработка на кода (тоест улеснява случая, в който различни групи работят по различни части от кода и merge-ването на тези части). Когато все пак трябва да се push-ва или fetch-ва това е възможно по няколко много стандартни мрежови протокола (HTTP, FTP, rsync). Има вътрешно-изградена емулация на CVS, която позволява да „мимикрира" централизиран сървър и да работи със CVS плъгини (някой по-човешки user interface). Ефективен при разработване на големи проекти.

Някои от проектите, които го ползват (уточнявам – не задължително за целите проекти, но поне за някои части от тях) са: Android, Debian, Eclipse, Fedora, ffmpeg, GIMP, GNOME, Kate, KDevelop, Linux Kernel, Wine, някои SQL-и.

Няколко думи за Mercurial
Имплементирана предимно на Python (с дребни изключения, където скоростта е от жизнено значение, като например diff). Първоначално е била разработена за Linux, но в последствие бива портната за повечето други масови операционни системи. Също като Git работата с Mercurial е предимно през конзола (command line), но има и няколко графични интерфейси, разработени за него. Отново отличителни белези са ефективност, независимост от големината на проекта и децентрализираност. Също така има добра поддръжка както за текстови файлове (source) така и за двоични такива (картинки, изпълними файлове, др.). Като повечето децентрализирани системи за контрол на кода и при Mercurial има добре-направен поток при branch-ване и merge-ване. Също така той съдържа вграден web интерфейс. Както повечето такива системи и той е безплатен (GNU GPL).
Някои проекти, които го ползват са: Mozilla, OpenJDK, OpenSolaris, OpenOffice, Symbian, Video4Linux, Mercurial, Netbeans, Vim.

В някои фирми има интегрирани „транслатори" от SVN към Git или Mercurial. Тоест е възможно да ползвате интерфейса на SVN (стандартните команди, с които сте свикнали), като програма се грижи да изпълни съответните такива на Git, което значително улеснява ползването му от начинаещи хора. Такива транслатори, обаче, си имат своите ограничения – примерно, макар и да е възможно да бъдат реализирани за почти всички команди, те често са реализирани така, че да НЕ позволяват commit (тъй като тази команда е с различна идеология при централизираните и децентрализираните version control системи). Това, макар и недостатъчно за един програмист (developer), е напълно приемливо за някой QA (Quality Assurance, тоест човек, чиято работа е да тества кода, написан от другите) или пък дизайнер. Това са позиции, които рядко (да се разбира почти никога) не се месят в кода на developer-ите, но пък им е нужна последната версия на кода, за да вършат своята собствена работа. Затова ограничената функционалност (липса на commit) в дадените случаи е напълно окей.

Другата независима тема, която разгледахме в часа беше как можем да генерираме произволни числа с компютър. На теория това е невъзможно, но на практика са постигнати доста добри резултати, които в общия случай са напълно приемливи (за повечето възможни неща, за които биха ни трябвали). Най-лесният pseudo-random number generator (PRNG) е базиран на (сравнително проста) теория на числата и поради своята ефективност и сравнително задоволителни резултати бива широко разпространен (на него са базирани random функциите в C, Java и Delphi).
Идеята зад него е да се генрира псевдо-произволна редица от числа, зададена чрез рекурентната зависимост r[i] = (r[i-1] * A + B) % C, където A, B и C са някакви константи, а r[0] (често наричан „seed") може да се задава от потребителя (или да се ползва default-на такава). Математически доказано е, че при добре избрани A, B и C (погледнете статията за LCG за повече информация) се генерират всички числа в интервала от 0 до C – 1, включително, точно по веднъж, преди да започнат да се повтарят. Това, макар и не истински рандом, често напълно покрива нуждите на потребителя. Проблем обаче възниква от това, че следващите C на брой числа

са същите като предходните C, при това в същия ред! (тоест тези генератори създават една и съща пермутация).
Обикновено в компютрите за модул (числото C) се ползва 2^{32} или 2^{64} като просто при сметките се оставят числата да overflow-ват, което де факто представлява именно остатък при деление на тях. Ако ви се наложи да генерирате по-произволни числа се препоръчва да се ползва някой по-сложен метод за това (например Mersenne Twister, който влиза в новия стандарт на C++).

В C/C++ функциите за рандом числа са (в библиотеката stdlib.h/cstdlib):

1. rand() - връщаща число между 0 и константата RAND_MAX, включително.
2. srand(int) – променя seed-а на редицата (тоест променя началната ѝ стойност).
3. lrand() – в някои имплементации има lrand, който връща 32 битово число, докато rand() едва 16 битово, но не е особено стандартизирано (тоест може да попаднете на компилатор, в който го няма).

Забележете, че константата RAND_MAX е зависима както от компилатора, така и от платформата. Примерно под Windows тази константа е 32767 ($2^{15} – 1$) както на MinGW (GCC за Windows), така и на Visual Studio (VC 8.0), докато под (повечето) Linux-и (в GCC) е 2147483647 ($2^{31} – 1$).

В Java се ползва класът Random:

1. Можете да подадете seed-а в конструктора или да не подавате аргумент, което ще доведе до ползване на default-ен seed.
2. В случай, че вече сте създали обект от класа, но искате да промените seed-а, можете да ползвате setSeed(long seed).
3. Тук имате доста по-удобни методи в зависимост какво точно число ви трябва: nextBoolean(), nextBytes(byte[] bytes), nextDouble(), nextFloat(), nextInt(), nextLong().
4. Ако ви трябва произволно число в някакви граници (всъщност това е най-честият случай) можете да ползвате nextInt(int n), което ви връща число между 0 и n – 1.