Tartalomjegyzék

1. fejezet

Vajna Miklós: Verziókezelés a Git használatával

1.1. Miért jó a verziókezelés?

Mielőtt megpróbálnánk megválaszolni ezt a meglehetősen összetett kérdést, próbáljuk meg részekre bontani. Bevezetésként legyen elég annyi, hogy jóval több ember használ verziókezelést, mint ahány tud erről. Gondoljunk csak arra, mikor kedvenc szövegszerkesztőnk *mentés másként* funkcióját használjuk. Meglehetősen kezdetleges módszer, de valójában máris verziókezelésről van szó: érintetlenül hagyjuk az eredeti példányt, és egy újat hozunk létre. Márpedig pont ez a verziókezelés lényege: egy projekt különböző állapotait nyilvántartani.

De hogy ne ragadjunk le a legtriviálisabb esetnél, a verziókezelés gyakorlatilag elengedhetetlen minden olyan munka esetén ahol több ember dolgozik ugyanazon a projekten. Ha ugyanahhoz a fájlhoz ketten nyúlnak hozzá egyszerre, akkor a verziókezelő ezt vagy automatikusan kezeli, vagy segítséget nyújt az ilyen jellegű probléma feloldásához. Fontos megjegyezni, hogy a jó verziókezelő nem feltétlenül old meg mindent helyettünk, cserébe viszont saját maga kérdés nélkül nem hibázhat. Ha már hosszabb ideig használjuk a gitet, észrevehetjük, hogy vannak olyan esetek amikor a git ütközést észlel, és a segítségünket kéri, míg ugyanazt a patch-et pl. a *patch(1)* egy figyelmeztetés mellett elfogadja. Ez pontosan azért van, mert jobb, ha nekünk kell feloldani egy ütközést, mint azt hinni, hogy minden problémamentesen lefutott, majd jóval később észrevenni (ha egyáltalán észrevesszük) az inkorrekt eredményt.

A verziókezelés ezen kívül segíti a hibakeresést. Ha van egy szkriptünk ami reprodukálja a hibát, akkor a jó verziókezelő bináris kereséssel gyorsan megtalálja az első hibás commitot.

Végezetül a verziókezelő dokumentációs eszköz. Igényes commit üzenetek esetén egy-egy kiadás esetén a változások listáját már generálni lehet, illetve később a forráskód minden egyes sorához részletes többlet-információt találhatunk, ha a forráskódbeli megjegyzések nem lennének elegendőek.

Természetesen verziókezelő nélkül is lehet élni, legfeljebb nem érdemes. Ennek legkezdetlegesebb kiküszöbölése mikor áttelefonálunk a kollégának, hogy *légyszi ne nyúljál most hozzá, mert dolgozom rajta.* Vagy ha az OpenOffice.org *változások követése* funkcióját használjuk, mely pár száz változtatás esetén már

teljes bizonyossággal használhatatlan. Szövegfájloknál megoldható a kézi 3-utas (3-way) merge, de egy idő után ezt kézzel csinálni szintén unalmas játék. Itt jegyezném meg, hogy ha belegondolunk, alapvető igény, hogy más-más típusú fájlokat más algoritmussal merge-öljünk, mégis a legtöbb verziókezelőből kispórolták ezt a funkciót, és a merge algoritmus a verziókezelő egyik leginkább bedrótozott modulja. Természetesen a git szakít ezzel a hagyománnyal és kedvenc programozási nyelvünkön írhatunk hozzá merge meghajtókat.

Hasonlóan érdekes ámde értelmetlen próbálkozás a CVS névre hallgató, verziókezelőnek csúfolt program. A git szempontjából elemezve a legnagyobb probléma vele, hogy nem állítható vissza maradéktalanul a projekt egy-egy időpillanatban fennálló állapota, abból következően, hogy nem a projekt teljes állapotát tárolja, hanem egy-egy fájl változásait. A *cvsps(1)* ezt próbálja meg korrigálni – több-kevesebb sikerrel.

Még egy általános problémát említenék, amiben a git szintén nem érintett. Elosztott környezetben verziószámokat használni egy-egy állapotra nem túl okos megfontolás. Ha több fejlesztő dolgozik egy projekten akkor tipikusan egy ember feladata szokott lenni hivatalos kiadásokat készíteni, azoknak verziószámot adni már van értelme. De a köztes állapotokat valamilyen egyedi módon kell megcímezni, hiszen az 1.0 verzió után ha két fejlesztő is commitolt egy-egy változtatást, nem hívhatjuk mind a kettőt 1.0.1-nek, hiszen a kettő nem azonos.

1.2. Az elosztott verziókezelők előnyei

A git legnagyobb előnye, hogy elosztott. Sok más elosztott verziókezelő is létezik, és legtöbbjük sokkal több funkcionalitást nyújt, mint bármelyik központi verziókezelő. Sok olyan gitről szóló leírás látott napvilágot, amely nagyrészt az elosztott verziókezelők előnyeit hangsúlyozza, úgy feltüntetve, mintha ez a git kizárólagos előnye lenne. Ezzel két probléma van. Egyrészt ha erre a csúsztatásra rájön az olvasó, valószínűleg tovább sem olvassa a cikket, mert nem hiteles. A másik probléma, hogy a git még az elosztott verziókezelők mezőnyében is hihetetlen előnyökkel bír, és az ilyesfajta cikkek, ezeket az előnyöket nem fejtik ki, pedig a git valójában ettől igazán innovatív, nem azért, mert sikerült a Linux kernel atyjának egy újabb elosztott verziókezelőt implementálnia.

Ennek ellenére nem feltételezhetjük, hogy mindenki tisztában van az elosztott verziókezelők előnyeivel, így röviden összefoglaljuk ezeket is. Talán a legfontosabb különbség, hogy egy repó letöltésekor nem csak az utolsó verzió kerül letöltésre, hanem a teljes repó, így minden repó egyenlő, ettől elosztott a rendszer, hogy nincs egy kijelölt központ. Ennek a következménye, hogy számos gyakran használt művelet (blame, log, diff, merge) nagyságrendekkel gyorsabb. Ez nagyon fontos, például ha a blame funkció 10 másodpercnél több időt vesz igénybe, akkor gyakorlatilag értelmetlen, a legtöbb eseten ennél több időt nem érdemes az egészre vesztegetni, akkor már inkább megnézzük az adott fájl történetét, abból is ki lehet bogarászni a számunkra érdekes információt. Tehát ha bizonyos funkciók lassúak, a felhasználók nem fogják használni, felesleges volt időt vesztegetni az implementálásukra.

Az előzőekből következik, hogy a legtöbb művelet így nem igényel hálózati kapcsolatot, eltűnik a központi repó, mint egyetlen hibalehetőség (single point of failure) problémája, megszűnik a commiter fogalma (hiszen mindenki commitolhat a saját repójába), minden egyes letöltés implicit módon egy backupot

készít a teljes repóról. Az előzőekből nem feltétlenül következik, de a git esetén a branch létrehozása és merge-ölése is nagyon egyszerűvé válik, egyrészt mert erre a kezdetektől odafigyeltek, másrészt a helyi commitok miatt.

1.3. A git előnyei

Csak a git előnyeiről külön könyvet lehet írni, így a teljesség igénye nélkül említünk párat, ami remélhetőleg arra már elég lesz, hogy az olvasó motivációt érezzen a git kipróbálásához.

Központi verziókezelők (például Subversion) esetén is létezik a branch és a merge fogalma, de meglehetősen korlátozottan. Készíthetünk egy branch-et, abban dolgozhatunk, a trunk-ot merge-ölhetjuk bele sokszor, majd ha készen vagyunk, akkor egy külön merge paranccsal merge-ölhetjük a branch-ünket a trunk-ba, és innentől hozzá ne nyúljunk a branch-ünkhöz, mert összedől a világ. A gitnél természetesen minden branch egyenlő és bármelyik branch-et bármelyik branch-be annyiszor merge-ölhetjük ahányszor jólesik. Összehasonlításképp például a darcs merge algoritmusa is enged elvileg ilyesmit, de nagyobb számú ütközés esetén általában végtelen ciklusba kerül. A bzr merge algoritmusa nem szenved ilyen problémával, de szinten be van drótozva a verziókezelőbe, az algoritmust nem cserélhetjük le a sajátunkra egykönnyen.

A rerere nevű szolgáltatás azt biztosítja, hogyha egyszer feloldottunk egy ütközést, akkor ha legközelebb egy ugyanolyan ütközést kapunk, akkor már automatikusan feloldja a rendszer. Ez rendkívül hasznos funkció patchsetek karbantartásakor.

A git implicit módon, futási időben és utólagosan ismeri fel a fájlok másolását és átnevezését. Mi több, ezt nem csak teljes fájlokkal, hanem nagyobb méretű kódblokkokkal is meg tudja tenni. Jelen pillanatban (2009 október) tudomásunk szerint nincs még egy verziókezelő, amely ilyen funkcionalitást nyújtana.

Példa, melyben az *a.c* fájlból átmásoltuk az *ip_get* függvényt a *b.c* fájlba, majd a git blame felismeri, hogy annak az utolsó tényeleges módosítása még akkor volt, mikor az régi fájlban volt:

```
$ git blame -C b.c
...
607001b8 b.c (Miklos 02:21:22 32)         g_free( t );
607001b8 b.c (Miklos 02:21:22 33) }
607001b8 b.c (Miklos 02:21:22 34)
^56bbfb0 a.c (Miklos 02:21:14 35) ipst_t *ip_get( char *ip_txt )
^56bbfb0 a.c (Miklos 02:21:14 36) {
^56bbfb0 a.c (Miklos 02:21:14 37)         unsigned int ip;
^56bbfb0 a.c (Miklos 02:21:14 38)         ipstats_t *l;
^56bbfb0 a.c (Miklos 02:21:14 39)         int p[4];
```

A combined diff egy olyan patch szintaxis, mely merge-ök eredményét tudja bemutatni, egyszerre összehasonlítva az eredeti saját, az eredeti másik, és a végső verziót.

A git grep hasonlóan működik a *grep(1)* programhoz, viszont rekurzív grepelés esetén csak a követett fájlokat veszi figyelembe.

1.4. Felhasználóbarátság és pokol

A git tipikus UNIX eszköz, meredek tanulási görbével, azonban a szükséges tanulási szakasz után hihetetlenül hatékony eszközt kapunk. Ha az olvasó találkozott a *vim*, *emacs*, *mutt* vagy hasonló szoftverekkel, csak, hogy néhány példát említsünk, akkor ismeri ezt a szituációt. Ha jobban megvizsgáljuk az okokat, az egyik leginkább szembetűnő a bőség zavara. A git 1.6.4-es verziójához összesen több, mint 600-an küldtek be módosításokat, így számos munkafolyamatot és speciális eseten támogat, melyek között elsőre nehéz lehet az eligazodás. A hivatalos dokumentáció elsősorban referencia jellegű, bár ez az utóbbi 2 évben jelentősen javult. Ezen kívül szintén az utóbbi egy-két évben több gittel foglalkozó könyv is megjelent, javítva az arányt.

1.5. Adatszerkezetek

A projekt történetét tároló objektum-adatbázis adatszerkezetei meglepően egyszerűek. Négy féle objektum-típus van. A blob egy fájl egy adott változatát tárolja. A tree (fa) egy pozitív elemszámú listát tárol, melynek elemei tree-k, vagy blobok lehetnek. Ezzel már el is tudjuk tárolni a projekt egy pillanatbeli állapotát. Mivel az állapotokat össze akarjuk kötni, bevezetésre került a commit, mely pontosan egy tree-re mutat, és nulla vagy több szülője lehet. Az utolsó típus a tag, ennek neve van, valamint egy objektumra mutat, ami tipikusan commit szokott lenni.

Látjuk tehát, hogy minden egyes commit tárolja a projekt teljes állapotát, valamint az egyes állapotok közötti változások (diff) mindig futási időben kerül kiszámításra. Ennek ellenére néhány esetben praktikus mégis úgy gondolni a commitra, mintha csak egy patch lenne az előző commithoz képest, a rebase kapcsán ez a szemléletmód még hasznos lesz.

A mutatók minden esetben a hivatkozott tartalom sha1 értékét tartalmazzák, aminek több előnye is van: ha két commit között csak egy fájl változott, akkor a legtöbb tree és egy kivételével az összes blob objektum újra felhasználható; a módosítás nélküli fájlmásolások és átnevezések detektálása triviális; valamint a legutolsó commit sha1-ét meghatározza a projekt korábbi összes állapota, így ha digitálisan aláírunk egy kiadás alkalmával létrehozott taget, akkor az egyben hitelesíti a teljes korábbi történetet (kriptográfiai biztonságosság).

A git egyik legnagyobb előnye a fenti adatszerkezetek robusztussága Ez olyannyira igaz, hogy ezeket az adatszerkezeteket kezelő könyvtárat (libgit), amely C nyelven sose íródott meg önálló formában, megírták már számos nyelven (Python, Ruby, Java, C#).

Az adatszerkezeteken kívül egy repóban még vannak referenciák, melyik a tag-hez hasonló módon egy-egy commitra mutatnak, viszont egyezményesen ha egy új commit születik, akkor automatikusan az új commitra illik állítani a referenciát; symref-ek, amik olyan mutatók amik ref-ekre mutatnak; hook-ok (hurkok) melyek bizonyos események bekövetkeztekor automatikusan végrehajtódnak; reflogok, melyek dokumentálják, hogy a referenciák milyen objektumokra mutattak korábban – ez kifejezetten hasznos patchsetek karbantartásakor; egy config (beállítási) fájl, valamint az index, melyről később még részletesen szó lesz.

Példa a reflog használatára:

```
$ git checkout "@{10 seconds ago}"
$ git checkout master
$ git log -g --pretty=oneline
402de8e HEAD@{0}: checkout: moving from 402de8e to master
402de8e HEAD@{1}: checkout: moving \
from master to @{10 seconds ago}
402de8e HEAD@{2}: commit: B
c820060 HEAD@{3}: commit (initial): A
```

Egy gyakori kérdés, hogy mi a különbség a merge és a rebase között. Ha a git adatszerkezeteit nem ismerjük, akkor erre nehéz is válaszolni. A fenti bekezdések alapján már viszont megérthetjük a különbséget, az alábbi ábrák alapján. Vegyünk egy kiindulási állapotot:

```
      A---B---C topic
     /
D---E---F---G master
```

Tehát a D..G commitok a projekt master branch-ét jelképezik, a fejlesztő pedig akkor, mikor az E commit volt a legutolsó a master branch-ben, nyitott egy új topic (téma) branch-et és ott létrehozott 3 commitot. Az ilyen topic branch-ek azért nagyon hasznosak, mert elindíthatjuk őket egy olyan állapotból amikor tudjuk, hogy a master branch biztosan stabil, dolgozhatunk nyugodtan, úgy, hogy mások nem zavarják a munkánkat, majd mikor az adott témát befejeztük, merge-ölhetjük a topic branch-et a masterbe.

Nézzük mi történik rebase esetén:

```
             A'--B'--C' topic
            /
D---E---F---G master
```

Azt látjuk, hogy az A..C commitokat patch-ként tekintettük, elmentettük, az A..C commitokat eldobtuk, majd a G commitra raktuk rá a patch-eket, ezzel új A..C commitokat létrehozva. Mivel a régi A szülője az E volt, az újé a G, emiatt az A commit sha1-e más lesz. Ez akkor hasznos, ha például az A..C commitok egy patchsetet képeznek, és a projekt 1.0 verziójáról a 2.0 verzióra akarjuk azt frissíteni. A rebase közben a git minden olyan patchnél amit nem sikerült alkalmazni megáll, és lehetőséget biztosít, hogy feloldjuk az ütközéseket. Ez a megoldás szép historyt generál, hiszen úgy tűnik, mintha eredetileg is a 2.0-hoz készült volna a patchset, vagy például egy hibát is javíthattunk közben a B commitban, és azt hihetik a többiek, hogy eredetileg is úgy (tökéletesen) sikerült.

Felmerülhet a kérdés, hogy jó-e, ha az ilyen hibákat takargatják. Ha belegondolunk, hogy a későbbi hibakereséshez fontos, hogy minden *végső* (ami egy logikai fogalom, tehát például a master branch-be kerülő) commit olyan kell legyen, hogy lefordul a forráskód, akkor el kell ismernünk, hogy ez egy hasznos funkció. Ha a karbantartó kap egy jó patchsetet amit be szeretne olvasztani, de van egy olyan patch ami fordítási hibát okozna, akkor ezt így ki tudja javítani.

Nézzük mi lesz merge esetén:

```
      A---B---C---H topic
     /           /
D---E---F---G master
```

Azt látjuk, hogy egyetlen új commit született, aminek két szülője van. Így maradandó nyoma marad, hogy az A..C commitok egy külön branch-ben készültek. Ennek is megvan a maga előnye. Ha például valaki egy másik patchsetet készít a régi C commitra alapozva, akkor azt triviális rebase-elni a H commitra, míg ha a régi C-ről akar rebase-elni az új C-re azt kézzel kell megadnia, hiszen a repónak nincs információja arról, hogy az új C commitnak volt régi változata is.

1.6. Használat külsősként

Ez az a szituáció, mikor találtunk egy projektet, tetszik ahogy működik, de pár funkciót meg akarunk benne valósítani. Ilyenkor sorban megvalósítjuk a funkciókat, például minden egyes funkciót egy-egy commitban. Ahhoz viszont nem lesz jogunk, hogy a saját repónkból ezeket a commitokat a projekt repójába írjuk. Tehát külsősök vagyunk. Ezt a módot is nagyon jól támogatja a git, a git format-patch paranccsal tudunk patchsetet generálni például egy branch csak helyben elérhető commitjaiból. Egy külsős sose merge-öl, hanem mindig rebase-el. Ez is jól támogatott, a helyi commitok sorrendjét át tudjuk rendezni, darabolni tudjuk őket, összeolvasztani. A karbantartó oldalán pedig az emailben vagy fájlként megérkezett patchsetet a git am parancs tudja újra commitokká alakítani.

1.7. Parancsok

A git 1.6.4 145 paranccsal érkezik, ezt elsőre nehéz áttekinteni. A legszűkebb részhalmaz az a néhány, amit a git help jelenít meg, először ezekkel érdemes megismerkedni. Egy tágabb halmaz a *porcelain* nevet viseli, mely a magasszintű parancsokat tartalmazza. Ezek azok, amiket egy tipikus verziókezelőben elvártnak gondolunk. Végül egy másik nagy halmaz a *plumbing* (csővezeték) nevet viseli, mely alacsony-szintű parancsokat tartalmaz. Ezek paraméterezése, valamint a parancsok kimenete visszafele mindig kompatibilis, szkriptekből ezeket érdemes használni. Ezen parancsok létezésének eredménye az, hogy számos alternatív felhasználói felület is készült a githez.

Egy érdekes kezdeményezés támogatása a gitben a fast-import illetve a fast-export parancs. Ezek a repó tartalmát egy verziókezelő-független folyammá alakítják, és ilyen importer és exporter létezik más rendszerekhez is, például a bzr-hez, darcs-hoz, mercurialhoz. Nyilvánvaló előnye, hogy így N verziókezelő esetén csak 2N programot kell írni, és nem N*N-et.

1.8. Az index

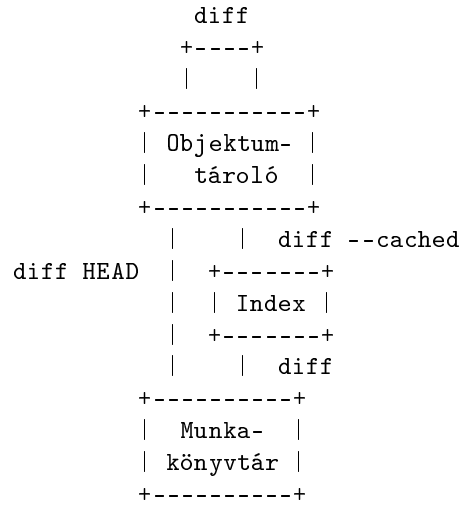
Az index egy köztes réteg a munkakönyvtár és az objektum-adatbázis között. A munkakönyvtárban változtatjuk meg a fájlokat, majd ezen változtatások egy részét az indexbe rakjuk (staging), végül a commit csupán csak az index tartalmát másolja az objektum-adatbázisba.

Új felhasználók gyakran elfelejtik ezt a fontos részletet. Ennek következménye például az, hogyha a git add paranccsal egy létező, a munkakönyvtárban

módosított fájlt az indexhez adunk, a fájlt újra módosítjuk, majd commitolunk, akkor a fájl indexbeli verziója lesz commitolva, nem a munkakönyvtárbeli!

Ez több szempontból is hasznos. Akkor például, ha egy fájlban két külön helyen két változtatást eszközöltünk, de a következő commitba csak az egyiket szeretnénk berakni. Vagy a karbantartónak is hasznos: ha merge-öl és sok fájl változott, de csak egyben van ütközés akkor a többi fájl bekerül az indexbe és ha a munkakönyvtárat összehasonlítjuk az indexszel akkor csak a számunkra érdekes részt, az egyetlen ütköző fájl változásait fogjuk látni, a többi változást nem.

A három réteg (objektum-adatbázis, index, munkakönyvtár) közötti diffelés eszköze a git diff, git diff –cached és a git diff HEAD parancs:



1.9. Elérhetőségek

A szerző ezúton is elnézést kér, hogy sok – a cikkben szereplő – témát csak érintőlegesen említett, mint az korábban szóba került, a témáról vastag könyvet lehetne írni, a cél leginkább a figyelemfelkeltés volt. Az alábbi linkek további kérdések esetén remélhetőleg segítséget nyújtanak.

GIT honlap: http://git-scm.com/

Levelezési lista: http://vger.kernel.org/vger-lists.html#git

IRC: #git @ irc.freenode.net

A diák és ezen cikk elérhetősége: http://vmiklos.hu/odp/