Grundläggande Git-teori

Henning Nåbo

21 april 2016

Git är ett versionshanteringsverktyg som skapades av Linus Torvalds 2005 för att versionshantera Linuxkärnan. Det är ett distribuerat verktyg som använder sig av hashvärden för att lagra historik.

1 Repository

Ett repository, eller repo som det ofta kallas, är ett mappsystem som är versionshanterat av git. Repot innehåller information om hur mappsystemet ser ut, hur det har sett ut och vem som har gjort vilka ändringar.

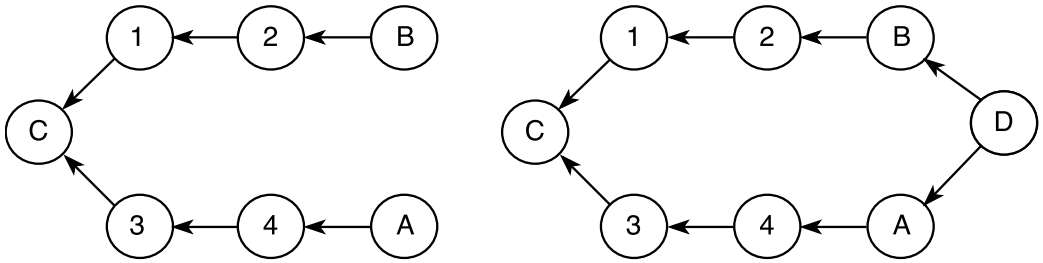
Kopior av ett repo kan finnas distribuerat på flera datorer och vanligtvis hålls dessa synkroniserade genom att ha en kopia på en server och sedan låta alla distributioner synkronisera sig med denna. Gitlab och github är exempel på tjänster som tillhandahåller denna sortens servrar.

2 Hashvärde

En hashfunktion är en funktion som tar data av godtycklig storlek och genererar ett hashvärde av specifik storlek. Hashfunktionen bör minimera risken att två set av data får samma hashvärde samtidigt som den minimerar antalet tecken i hashvärdet. Git använder en hashfunktion som genererar hashvärden bestående av 40 hexadecimala tecken.

3 Commit

En commit kan ses som ett fotografi av ett repo, ett sätt som repot såg ut vid ett vist tillfälle. När en commit görs beräknas ett hashvärde ut för samtliga filer i repot, redan beräknas hashvärden för mappar baserat på de hashvärden som de filer och mappar den innehåller har. Committen innehåller också ett



Figur 1: En commit-historik före och efter en merge. (Observera att pilarna går bakåt i tiden då en commit har en pekare till sin förälder)

meddelande och vem som gjort den. Hashvärdet av meddelandet blir namnet på committen. Dessutom innehåller en commit namnet på sin förälder, det vill säga den commit som repot var i innan committen gjordes.

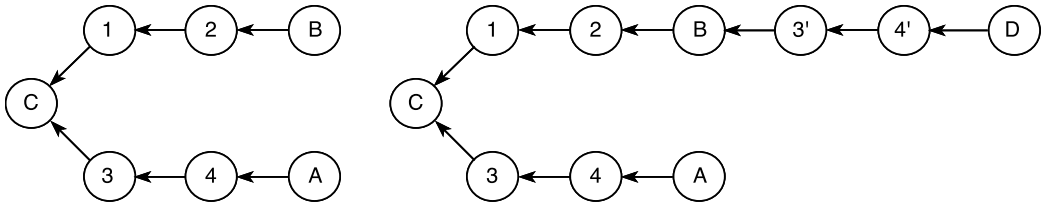
4 Branch

En branch är kort och gott ett mänskligt läsbart namn på en commit. I mer detalj kan man beskriva det som ett objekt som pekar på en commit och som kan flyttas längs historiken. För att hålla koll på vilken branch som beskriver repots nuvarande utseende finns en specialbranch som heter HEAD och som endast kan peka på en annan branch. Står du i branchen master (HEAD pekar alltså på master) och gör en commit så kommer master att peka ut den nya committen istället.

5 Merge

Ofta när flera personer arbetar på samma projekt kommer arbetet ej ske linjärt. För att synkronisera arbete krävs att man slår ihop commits, en så kallad merge. Är den ena commiten förfader till den andra är mergen trivial, då flyttas endast eventuell branch till den yngre commiten. Denna sorts merge kallas för en “fast-forward”

Har commitsen olika historik skapas en ny merge commit får två eller fler föräldrar som applicerar alla ändringar som gjorts sedan föräldrarnas senaste gemensamma förfader på en av commitsen. Har föräldrarna modifierat olika områden kan detta ske automatiskt men annars orsakas konflikter som måste lösas manuellt.



Figur 2: En commit-historik före och efter en rebase. Notera att 3' och 4' inte är samma commit som 3 och 4 även om de gör samma ändringar då de appliceras på en annan commit och därför kommer få andra hashvärden. Observera att pilarna går bakåt i tiden då en commit har en pekare till sin förälder.

6 Rebase

Ett alternativ till en merge är en så kallad rebase. Detta liknar en merge mellan två commits men orsakar en linjär historik, detta genom att användaren försäkrar git om att den ena committens ändringar kan appliceras (eller gjordes) efter den andra committen.

Ska commit A rebasas på commit B så kommer alla förändringar gjorda av commits från den senaste gemensamma förfadern C och fram till A appliceras en efter en på B och på så sätt skapa nya commits fram tills den förändringen gjord av commit A, denna commit kan vi kalla D. Sedan kommer eventuella branchhuvuden att flyttas till D. p Detta innebär att alla commits från C till A kommer "glömmas bort". Det är därför viktigt att samtliga användare använder commiten D och inte arbetar vidare från committen A.