Engineer

Semantisk versionering

versionsnummer som visar om en uppdatering är kompatibel

Vad det är

Semantisk versionering, på engelska semantic versioning och oftast förkortat SemVer, är en uppsättning regler för hur releaser numreras, så att själva numret berättar vilken sorts ändring en release innehåller. I specifikationens svenska översättning kallas det semantisk versionshantering. Specifikationen, som nu är i version 2.0.0 och ursprungligen skrevs av Tom Preston-Werner, använder tre tal på formen MAJOR.MINOR.PATCH. En release som bryter det publika API:t höjer major-versionen, en som lägger till funktionalitet på ett bakåtkompatibelt sätt höjer minor-versionen, och en som bara rättar buggar utan att röra API:t höjer patch-versionen. När ett tal höjs, nollställs talen till höger om det, och varje tal jämförs som ett tal och inte som text, så `1.10.0` kommer efter `1.9.0` .

Allt vilar på ett krav: projektet måste deklarera ett publikt API, i koden eller i dokumentationen, eftersom talen beskriver ändringar i just det API:t och inget annat. När en version väl är släppt, får dess innehåll inte ändras; en rättelse är en ny version. Major-version noll, `0.y.z` , är till för den inledande utvecklingen, där vad som helst kan ändras när som helst och API:t inte ska betraktas som stabilt. Version `1.0.0` är den punkt där det publika API:t definieras, och därifrån följer talen hur det ändras.

Två valfria tillägg utökar formatet. En förhandsversion (pre-release) skrivs efter ett bindestreck och sorteras före den release den leder fram till, så `1.0.0-alpha` kommer före `1.0.0-alpha.1` , som i sin tur kommer före `1.0.0-beta` , `1.0.0-rc.1` och till sist `1.0.0` . Byggmetadata skrivs efter ett plustecken, som i `1.0.0+20130313144700` , och ignoreras när versioner sorteras, så två versioner som bara skiljer sig där har samma prioritet. Ett inledande v, som i Git-taggen `v1.2.3` , är inte en del av den semantiska versionen; specifikationen kallar det ett vanligt sätt att visa att det rör sig om ett versionsnummer. Go gör det till ett krav: en kanonisk version av en Go-modul är bokstaven v följd av en semantisk version.

Använd begreppet i ett beslut

Använd ”Vad är semantisk versionering?” i ett konkret beslut. Vi kan knyta begreppet till systemet ni bygger eller köper. Testade, släppta och möjliga att backa, så att ändringar når kunderna en tisdagseftermiddag i stället för sent en fredagskväll.

Se våra tjänster Hör av er →

Granska antagandena bakom ordet

Använd förklaringen och det länkade arbetet för att formulera frågor till projektet. Vi kan undersöka antagandena tillsammans innan ett verktyg väljs. Automatiserade GIS-driftsättning i GitHub Actions — driftsättning, data och rapportering, helt utan manuellt arbete

Automatiserade GIS-driftsättning i GitHub Actions Hör av er →

Varför det spelar roll

Specifikationen utgår från det den kallar beroendeträsket (dependency hell). Anger du beroenden för snävt, hamnar du i versionslås, där en uppgradering av ett paket kräver nya versioner av allt som är beroende av det. Anger du dem för löst, hamnar du i versionsnaivitet: du antar kompatibilitet med fler framtida versioner än vad som är rimligt. Ett versionsnummer med en överenskommen betydelse låter ett projekt säga vad det accepterar: ett bibliotek som behöver en funktion som tillkom i 3.1.0 kan acceptera allt från 3.1.0 upp till, men inte med, 4.0.0, och få senare rättelser och tillägg utan att någon redigerar kravet.

Pakethanterare agerar på det löftet utan att fråga. I Cargo, pakethanteraren för Rust, är ett vanligt krav som `"1.2.3"` ett caret-krav och betyder minst 1.2.3 men under 2.0.0. När Cargo skriver `Cargo.lock` , väljer det den högsta versionen inom det intervallet, och det stannar på 1.x även efter att 2.0.0 har publicerats, eftersom det betraktar 2.0.0 som inkompatibel. I node-semver, som beskriver sig som verktyget för semantisk versionering i npm, betyder `^1.2.3` detsamma som `>=1.2.3 <2.0.0-0` , och `~1.2.3` tillåter bara ändringar på patch-nivå, `>=1.2.3 <1.3.0-0` . Go bygger in regeln i importsökvägarna: från major-version 2 och uppåt måste en moduls sökväg sluta med ett suffix som `/v2` , så två inkompatibla major-versioner blir två olika importsökvägar. Minimal version selection i Go tar sedan, för varje modul, den högsta version som något krav begär, och det är den lägsta version som uppfyller alla krav. Resultatet sparas inte i någon låsfil; varje kommando som arbetar med moduler räknar fram det på nytt.

Inget av detta gör numret till en garanti. Det är ett löfte från dem som publicerar paketet, och det är aldrig bättre än deras bedömning av vad som räknas som brytande. Därför publicerar Cargo en lång kompatibilitetsguide som sorterar ändringar i major, minor och möjligen brytande (possibly-breaking). Det mesta i guiden handlar om ändringar som gör att kod som kompilerade i går inte

kompilerar i dag. Guiden påpekar själv att nästan varje ändring medför en viss risk för beteendet vid körning och att frågan om en sådan ändring är inkompatibel oftast avgörs efter underhållarnas eget omdöme.

Så ser det ut i praktiken

Följ ett påhittat Rust-bibliotek, tally, genom dess releaser. Version 0.4.2 är fortfarande inledande utveckling. Version 1.0.0 förklarar det publika API:t stabilt. Version 1.0.1 rättar ett avrundningsfel och är en patch. Version 1.1.0 lägger till en funktion, `parse_strict`, och markerar den gamla `parse` som föråldrad (deprecated): specifikationen kräver en minor-release för det och ber om minst en sådan release innan funktionen tas bort. Version 2.0.0 tar bort `parse` och är en major-release. Version 2.1.0-rc.1 är en förhandsversion av nästa minor-version.

En applikation som är beroende av tally och tre andra påhittade crates anger sina krav i `Cargo.toml`. Kommentarerna visar det intervall som Cargo läser ut ur varje rad:

```
[dependencies]
tally = "1.0.1"         # >=1.0.1, <2.0.0: takes 1.1.0, never 2.0.0
ledger = "0.4.2"        # >=0.4.2, <0.5.0: 0.5.0 counts as breaking
stamp = "~1.2.3"        # >=1.2.3, <1.3.0: patch releases only
quill = "=2.1.0-rc.1"   # exactly this pre-release
```

Raden för `ledger` visar en konvention i verktygen och inte i specifikationen: för Cargo är det första talet från vänster som inte är noll det som signalerar brott, medan specifikationen säger att allt under 1.0.0 kan ändras när som helst. Cargo håller förhandsversioner utanför ett krav om inte kravet nämner en, så ett krav på `"2.0"` skulle aldrig plocka upp releasekandidaten för tally. Den exakta `=`-formen på sista raden är det fall där Cargos dokumentation säger att ett fullständigt angivet krav kan vara motiverat: att vara beroende av en förhandsversion och samtidigt undvika brytande ändringar mellan förhandsversioner. När versionerna har valts, registreras de i `Cargo.lock`, och `cargo update` flyttar dem inom de här intervallen. Med `--locked` returnerar ett bygge som skulle behöva ändra låsfilen ett fel i stället.

Det numret inte täcker är allt utanför API:t. Licensgranskningen av 644 Rust-crates, utförd i september 2026, kontrollerar licensen för varje crate vid varje bygge mot en lista över villkor som projektet godtar, eftersom en rutinmässig versionshöjning djupt ned i beroendegrafen i tysthet kan ändra under vilka villkor

en applikation får distribueras, i en låsfil som ingen läser rad för rad. För att bevisa att kontrollen fungerar, fick den fallera med flit, två gånger. Den ena gången skrevs en crates licensuttryck om, så som det ser ut när en crate ändrar sina villkor uppströms mellan versioner, något som ingen mänsklig granskning fångar, eftersom craten i sig inte är ny. Den andra gången flyttades den fastlåsta kärnrevisionen utan att licensgranskningen genererades om. En semantisk version kan berätta att 1.4.0 är tänkt att fungera med kod skriven mot 1.0.0. Den kan inte berätta om du fortfarande får leverera den.

Var det går fel

En brytande ändring i en minor-release. Det händer av misstag eller för att man ser olika på vad som är brytande. Cargos guide räknar upp ändringar som ser harmlösa ut men inte är det: att lägga till en variant i en publik enum bryter kod som matchar uttömmande på den, om inte enumen märktes `#[non_exhaustive]` från början, och i sällsynta fall med glob-importer orsakar till och med ett nytt publikt element ett kompileringsfel. Specifikationens svar, när misstaget väl har upptäckts, är att släppa en ny minor-version som återställer kompatibiliteten, aldrig att ändra den redan publicerade releasen, och att tala om för användarna vilken version som var fel. På den konsumerande sidan är det en låsfil som är incheckad i repot och tester som körs vid varje uppdatering som fångar felet innan det når produktion.

Att bli kvar på 0.x i åratal. Enligt specifikationen har major-version noll inget stabilt API alls. Dess FAQ säger det rakt ut: används mjukvaran redan i produktion, borde den med största sannolikhet redan ha version 1.0.0. Verktygen fyller tomrummet med en egen konvention: Cargo och node-semver behandlar det första talet från vänster som inte är noll som det brytande, så `^0.2.3` accepterar bara fler 0.2.x-releaser och `^0.0.3` accepterar inga uppdateringar alls. Ett bibliotek som har stått på 0.x i åratal har användare som är beroende av det men inget definierat löfte till dem. Släpp 1.0.0 när folk väl förlitar sig på API:t; det ändrar ett nummer, inte koden.

Att läsa in mer i numret än det säger. Kompatibilitet enligt specifikationen handlar om det publika API:t. Cargos guide klassar en höjning av den lägsta Rust-version som en crate kräver som möjligen brytande och rekommenderar i allmänhet att den behandlas som en minor-ändring, så en minor-uppdatering kan stoppa bygget på en äldre verktygskedja. När `package.rust-version` är satt, kan Cargos beroendeupplösare pröva äldre versioner av craten vid behov.

Licensvillkor, prestanda och beteende som API:t aldrig har dokumenterat ligger helt utanför numret. Kontrollera dem separat, med en licensgranskning och tester, i stället för att sluta dig till dem utifrån en minor-release.

Två major-versioner i ett bygge. När två beroenden behöver inkompatibla versioner av samma crate, till exempel 0.6 och 0.7, bygger Cargo båda. Kompilatorn behandlar deras typer som olika, även när namnen är desamma. Oftast visar det sig som ett kompileringsfel och ibland först vid körning: Cargos dokumentation beskriver en downcast som misslyckas eftersom objektet skapades av ett bibliotek som använde version 1.0.0, medan den anropande koden försöker göra en downcast till typen från 2.0.0. Kommandot `cargo tree -d` listar dubblerade versioner och var de kommer ifrån. En biblioteksförfattare kan mildra en major-release med det som Cargos dokumentation kallar ”the semver trick”: publicera den nya major-versionen och sedan en punktrelease av den gamla som återexporterar typerna från den nya.

Läs mer

- Semantic Versioning 2.0.0 — specifikationen och dess FAQ, bland annat om vad man gör efter att av misstag ha släppt en brytande ändring. Finns även i svensk översättning.
- SemVer Compatibility — Cargo-bokens guide till vilka ändringar i ett Rust-bibliotek som är major, minor eller möjligen brytande, var och en med ett exempel.
- Specifying Dependencies — Cargos kravsyntax: standard-, caret-, tilde- och jämförelsekrav, och hur förhandsversioner matchas.
- Dependency Resolution — hur Cargo väljer versioner, vad låsfilen innehåller och riskerna med två inkompatibla versioner i en graf.
- node-semver — npm:s verktyg för semantisk versionering, med reglerna för caret, tilde och intervall med förhandsversioner.
- Go Modules Reference — suffix för major-versioner, regeln för importkompatibilitet och minimal version selection.

Ordlista

Välj utifrån era faktiska förutsättningar

Ta med driftsättningsvägen, driftsriskerna och dem som har jour. Vi kan avgränsa en förändring med kontroller, en väg tillbaka och tydligt ansvar.

Se vad det kostar Hör av er →

Gör begreppen praktiska

Fråga hur ”Vad är semantisk versionering?” gäller i er organisation. Ett konkret exempel från ert nuvarande system räcker för att börja samtalet.

Se våra tjänster

welcome@engineer.company +45 50 10 61 84 Köpenhamn, Danmark

Ta med dig den här sidan

Visitkort

Omsorgsfullt skapad med kärlek av D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Upphovsrätt © 2025 – nu · Engineer ApS · Alla rättigheter förbehållna.

Utskrivet från https://engineer.company/sv/glossary/semantic-versioning/