Engineer

semantische Versionierung

Versionsnummern, die inkompatible Änderungen kennzeichnen

Was es ist

Semantische Versionierung, meist zu SemVer verkürzt, ist ein Regelwerk für Versionsnummern. Die Nummer selbst soll ausdrücken, welche Art von Änderung eine Veröffentlichung enthält. Die ursprünglich von Tom Preston-Werner verfasste Spezifikation liegt bei Version 2.0.0 und verwendet drei Zahlen in der Form MAJOR.MINOR.PATCH. Eine Veröffentlichung mit einer inkompatiblen Änderung der öffentlichen API erhöht die Hauptnummer. Abwärtskompatible neue Funktionalität erhöht die Nebennummer, reine Fehlerkorrekturen ohne Änderung der API erhöhen die Patchnummer. Steigt eine Zahl, werden alle rechts davon auf null zurückgesetzt. Verglichen wird jede als Zahl und nicht als Text: `1.10.0` folgt deshalb auf `1.9.0` .

Alles beruht auf einer Voraussetzung: Das Projekt muss eine öffentliche API im Code oder in der Dokumentation deklarieren. Die Zahlen beschreiben Änderungen an dieser API und sonst nichts. Der Inhalt einer veröffentlichten Version darf anschließend nicht verändert werden; eine Korrektur ist eine neue Version. Hauptversion null, `0.y.z` , ist für die anfängliche Entwicklung vorgesehen. Dabei darf sich jederzeit alles ändern, und die API soll nicht als stabil gelten. Version `1.0.0` markiert die Definition der öffentlichen API. Von da an richten sich die Zahlen nach deren Änderungen.

Zwei optionale Zusätze erweitern das Format. Eine Vorabversion steht hinter einem Bindestrich und wird vor der dazugehörigen Veröffentlichung einsortiert: `1.0.0-alpha` vor `1.0.0-alpha.1` , diese vor `1.0.0-beta` , `1.0.0-rc.1` und schließlich `1.0.0` . Build-Metadaten folgen auf ein Pluszeichen, etwa in `1.0.0+20130313144700` . Für die Sortierung werden sie ignoriert, sodass zwei nur darin verschiedene Versionen denselben Rang haben. Ein vorangestelltes v wie im Git-Tag `v1.2.3` gehört nicht zur semantischen Version. Die Spezifikation

nennt es eine verbreitete Kennzeichnung einer Versionsnummer. Go macht es zur Pflicht: Eine kanonische Go-Modulversion besteht aus dem Buchstaben v und einer semantischen Version.

Das Konzept für eine Entscheidung nutzen

Nutzen Sie „Was ist semantische Versionierung?“ für eine konkrete Entscheidung. Wir können helfen, das Konzept mit dem System in Verbindung zu bringen, das Sie entwickeln oder einkaufen. Geprüft, veröffentlicht und rückgängig zu machen: Änderungen erreichen Ihre Kunden am Dienstagnachmittag statt spät am Freitagabend.

Unsere Leistungen entdecken Kontakt aufnehmen →

Die Annahmen hinter dem Begriff prüfen

Nutzen Sie die Erklärung und die verlinkten Arbeiten, um Fragen für Ihr Projekt zu formulieren. Wir können die Annahmen gemeinsam mit Ihnen prüfen, bevor ein Werkzeug ausgewählt wird. GIS-SaaS-Deployments mit GitHub automatisiert — Bereitstellung, Daten und Berichte ohne Handarbeit

GIS-SaaS-Deployments mit GitHub automatisiert Kontakt aufnehmen →

Warum es wichtig ist

Die Spezifikation beginnt bei der sogenannten Abhängigkeitshölle. Zu enge Anforderungen führen zur Versionsblockade: Um ein Paket zu aktualisieren, müssen neue Versionen aller davon abhängigen Pakete erscheinen. Zu lockere Anforderungen führen zu dem, was die Spezifikation Versionspromiskuität nennt: künftige Versionen zu akzeptieren, ohne tragfähigen Grund, ihre Kompatibilität zu erwarten. Eine Versionsnummer mit vereinbarter Bedeutung lässt ein Projekt ausdrücken, was es akzeptiert. Eine Bibliothek, die eine erstmals mit 3.1.0 eingeführte Funktion braucht, kann alles von 3.1.0 bis ausschließlich 4.0.0 annehmen und spätere Korrekturen und Ergänzungen erhalten, ohne dass jemand die Anforderung bearbeitet.

Paketmanager handeln nach diesem Versprechen, ohne nachzufragen. In Cargo, Rusts Paketmanager, ist eine einfache Anforderung wie `"1.2.3"` eine Caret-Anforderung: mindestens 1.2.3, aber kleiner als 2.0.0. Beim Schreiben von `Cargo.lock` wählt Cargo die höchste Version innerhalb dieses Bereichs. Nach Veröffentlichung von 2.0.0 bleibt es auf der Linie 1.x, weil es 2.0.0 als inkompatibel betrachtet. Bei node-semver, das sich als semantische Versionsverwaltung für npm bezeichnet, bedeutet `^1.2.3` den Bereich `>=1.2.3 <2.0.0-0` . `~1.2.3` erlaubt nur Patchänderungen, `>=1.2.3 <1.3.0-0` . Go verankert die Regel in Importpfaden: Ab Hauptversion 2 muss ein Modulpfad mit einem Suffix wie `/v2` enden. Zwei inkompatible Hauptversionen haben damit verschiedene Importpfade. Gos minimale Versionsauswahl nimmt dann für jedes Modul die höchste von irgendeiner Anforderung verlangte Version. Das ist die

niedrigste Version, die alle Anforderungen erfüllt. Das Ergebnis wird nicht in einer Lockdatei gespeichert; jeder Befehl, der mit Modulen arbeitet, berechnet es neu.

Nichts davon macht die Nummer zur Garantie. Sie ist ein Versprechen der Paketautoren und nur so gut wie deren Urteil darüber, was eine inkompatible Änderung ist. Deshalb veröffentlicht Cargo einen langen Kompatibilitätsleitfaden, der Änderungen nach Hauptversion, Nebenversion und möglicher Inkompatibilität einteilt. Der größte Teil behandelt Änderungen, durch die gestern kompilierbarer Code heute nicht mehr kompiliert. Der Leitfaden selbst hält fest, dass fast jede Änderung ein Risiko für das Laufzeitverhalten mitbringt. Ob eine solche Änderung inkompatibel ist, müssen meist die Betreuer beurteilen.

Wie es in der Praxis aussieht

Verfolgen Sie die Veröffentlichungen einer erfundenen Rust-Bibliothek namens tally. Version 0.4.2 gehört noch zur anfänglichen Entwicklung. Version 1.0.0 erklärt die öffentliche API für stabil. Version 1.0.1 behebt einen Rundungsfehler und ist ein Patch. Version 1.1.0 ergänzt die Funktion `parse_strict` und kennzeichnet das bisherige `parse` als veraltet. Die Spezifikation verlangt dafür eine Nebenversion und empfiehlt mindestens eine solche Veröffentlichung vor dem Entfernen der Funktion. Version 2.0.0 entfernt `parse` und ist eine Hauptversion. Version 2.1.0-rc.1 ist eine Vorabversion der nächsten Nebenversion.

Eine Anwendung, die tally und drei weitere erfundene Crates verwendet, hält ihre Anforderungen in `Cargo.toml` fest. Die Kommentare nennen den Bereich, den Cargo aus jeder Zeile liest:

```
[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
```

Die Zeile für `ledger` zeigt eine Konvention der Werkzeuge statt der Spezifikation: Cargo behandelt die erste von links gelesene Zahl ungleich null als Kennzeichen einer inkompatiblen Änderung. Laut Spezifikation darf sich hingegen unterhalb von 1.0.0 jederzeit alles ändern. Cargo schließt

Vorabversionen aus einer Anforderung aus, sofern sie keine nennt. Eine Anforderung `"2.0"` würde deshalb tallys Release Candidate nie auswählen. Die exakte Form `=` in der letzten Zeile entspricht dem Fall, in dem laut Cargos Dokumentation eine vollständig festgelegte Anforderung sinnvoll sein kann: von einer Vorabversion abhängen und zugleich inkompatible Änderungen zwischen Vorabversionen vermeiden. Nach der Auflösung stehen die Versionen in `Cargo.lock`. `cargo update` verschiebt sie innerhalb dieser Bereiche. Mit `--locked` liefert ein Build stattdessen einen Fehler, wenn er die Lockdatei ändern müsste.

Alles außerhalb der API deckt die Nummer nicht ab. Die im September 2026 durchgeführte Lizenzprüfung von 644 Rust-Crates prüft bei jedem Build jede Crate-Lizenz gegen eine Positivliste. Eine gewöhnliche Versionsänderung tief im Abhängigkeitsgraphen kann unbemerkt die Bedingungen für die Weitergabe einer Anwendung ändern. Sie steht in einer Lockdatei, die niemand vollständig liest. Zum Nachweis ihrer Wirksamkeit wurde die Prüfung zweimal absichtlich zum Scheitern gebracht. Ein Test schrieb den Lizenzausdruck einer Crate um. Das entspricht dem Fall, dass eine Crate zwischen Versionen im Ursprungsprojekt ihre Bedingungen ändert, was eine menschliche Prüfung übersieht, weil die Crate selbst nicht neu ist. Der andere änderte die festgeschriebene Kernrevision, ohne das Audit neu zu erzeugen. Eine semantische Version kann sagen, dass 1.4.0 mit Code kompatibel sein soll, der für 1.0.0 geschrieben wurde. Sie kann nicht sagen, ob Sie die Anwendung weiterhin ausliefern dürfen.

Wo es schiefgeht

Eine inkompatible Änderung in einer Nebenversion. Das geschieht versehentlich oder durch ein anderes Verständnis von Inkompatibilität. Cargos Leitfaden nennt harmlos wirkende Änderungen, die es nicht sind: Eine neue Variante in einem öffentlichen Enum bricht Code, der alle Varianten vollständig abdeckt, sofern das Enum nicht von Anfang an mit `#[non_exhaustive]` gekennzeichnet war. In seltenen Fällen mit Glob-Importen erzeugt sogar ein neues öffentliches Element einen Compilerfehler. Wird der Fehler bemerkt, verlangt die Spezifikation eine neue Nebenversion, die die Kompatibilität wiederherstellt, niemals eine Änderung der bereits veröffentlichten Version. Nutzer sollen erfahren, welche Version fehlerhaft war. Bei den Projekten, die die Bibliothek verwenden, fangen eine eingecheckte Lockdatei und Tests bei jeder Aktualisierung das Problem ab, bevor es die Produktion erreicht.

Jahrelang bei 0.x bleiben. Nach der Spezifikation bringt Hauptversion null überhaupt keine stabile API mit. Die FAQ sagt deutlich: „Wenn Ihre Software in Produktion verwendet wird, sollte sie wahrscheinlich bereits 1.0.0 sein.“ Die Werkzeuge füllen die Lücke mit einer eigenen Konvention. Cargo und node-semver behandeln die erste Zahl ungleich null von links als Kennzeichen einer inkompatiblen Änderung. `^0.2.3` akzeptiert daher nur weitere Versionen der Linie 0.2.x, `^0.0.3` überhaupt keine Aktualisierung. Eine seit Jahren auf 0.x stehende Bibliothek hat Nutzer, die von ihr abhängen, aber kein definiertes Versprechen ihnen gegenüber. Veröffentlichen Sie 1.0.0, sobald Menschen sich auf die API verlassen. Das ändert eine Nummer, nicht den Code.

Mehr in die Nummer hineinlesen, als sie aussagt. Kompatibilität betrifft nach der Spezifikation die öffentliche API. Cargos Leitfaden stuft eine höhere benötigte Rust-Mindestversion einer Crate als möglicherweise inkompatibel ein und empfiehlt meist, sie als Änderung einer Nebenversion zu behandeln. Eine Aktualisierung auf eine neue Nebenversion kann deshalb den Build mit einer älteren Werkzeugkette verhindern. Mit `package.rust-version` kann Cargo bei der Auflösung bei Bedarf ältere Versionen der Crate ausprobieren. Lizenzbedingungen, Leistung und von der API nie dokumentiertes Verhalten liegen vollständig außerhalb der Nummer. Prüfen Sie sie durch Lizenzprüfung und Tests gesondert, statt sie aus einer Nebenversion abzuleiten.

Zwei Hauptversionen in einem Build. Brauchen zwei Abhängigkeiten inkompatible Versionen derselben Crate, etwa 0.6 und 0.7, baut Cargo beide. Der Compiler behandelt ihre Typen auch bei gleichem Namen als verschieden. Meist zeigt sich das als Compilerfehler, manchmal erst zur Laufzeit. Cargos Dokumentation beschreibt einen scheiternden Downcast: Das Objekt wurde von einer Bibliothek erzeugt, die Version 1.0.0 der Crate verwendet, während der aufrufende Code einen Downcast auf den Typ aus 2.0.0 ausführt. `cargo tree -d` zeigt doppelte Versionen und ihre Herkunft. Ein Bibliotheksautor kann eine neue Hauptversion mit dem sogenannten Semver-Trick aus Cargos Dokumentation abfedern: erst die neue Hauptversion veröffentlichen, dann eine kleine Folgeversion der bisherigen Reihe, die die Typen aus der neuen erneut exportiert.

Weiterführende Lektüre

- Semantische Versionierung 2.0.0 — Spezifikation und FAQ, einschließlich des Umgangs mit einer versehentlich veröffentlichten inkompatiblen Änderung.

- SemVer-Kompatibilität — Cargos Leitfaden dazu, welche Änderungen einer Rust-Bibliothek eine Hauptversion oder Nebenversion erfordern oder möglicherweise inkompatibel sind, jeweils mit Beispiel.
- Abhängigkeiten festlegen — Cargos Anforderungssyntax: Standard-, Caret-, Tilde- und Vergleichsanforderungen sowie die Auswahl von Vorabversionen.
- Abhängigkeiten auflösen — Wie Cargo Versionen auswählt, was die Lockdatei enthält und welche Risiken zwei inkompatible Versionen in einem Graphen bringen.
- node-semver — Die semantische Versionsverwaltung für npm mit Caret-, Tilde- und Vorabversionsbereichen.
- Go-Modulreferenz — Hauptversionssuffixe, Importkompatibilitätsregel und minimale Versionsauswahl.

Glossar

Für Ihre tatsächlichen Rahmenbedingungen auswählen

Beschreiben Sie den Bereitstellungsweg, die Betriebsrisiken und die Bereitschaftsverantwortlichen. Wir können eine Änderung mit Prüfungen, einem Wiederherstellungsweg und klaren Zuständigkeiten abgrenzen.

Preise ansehen Kontakt aufnehmen →

Die Begriffe praktisch nutzbar machen

Fragen Sie, wie „Was ist semantische Versionierung?“ auf Ihre Organisation zutrifft. Ein konkretes Beispiel aus Ihrem bestehenden System reicht aus, um das Gespräch zu beginnen.

Unsere Leistungen entdecken

welcome@engineer.company +45 50 10 61 84 Kopenhagen, Dänemark

Nehmen Sie diese Seite mit

Kontaktkarte

Mit Sorgfalt und Liebe gestaltet von D.

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

Copyright © 2025 – heute · Engineer ApS · Alle Rechte vorbehalten.

Gedruckt von https://engineer.company/de/glossary/semantic-versioning/