Engineer

Семантичне версіонування

номери версій, що показують, чи сумісне оновлення

Що це таке

Семантичне версіонування, англійською semantic versioning або скорочено SemVer, — це набір правил нумерації релізів, за якими сам номер показує, яку зміну містить реліз. Специфікація, нині у версії 2.0.0, яку первісно написав Tom Preston-Werner, використовує три числа у форматі MAJOR.MINOR.PATCH: мажорну, мінорну й патч-версію. Реліз, що ламає публічний API, збільшує мажорну версію; реліз, що додає функціональність, сумісну з попередньою версією, збільшує мінорну; реліз, що лише виправляє помилки й не зачіпає API, збільшує патч-версію. Коли якесь число зростає, числа праворуч від нього скидаються до нуля, а порівнюють їх як числа, а не як текст, тож `1.10.0` іде після `1.9.0`.

Усе тримається на одній вимозі: проєкт мусить оголосити публічний API, у коді чи в документації, бо номери описують зміни саме цього API і нічого більше. Випущену версію змінювати не можна; виправлення — це вже нова версія. Нульова мажорна версія, `0.y.z`, призначена для початкової розробки: будь-що може змінитися будь-коли, і API не слід вважати стабільним. Версія `1.0.0` — це момент, коли публічний API визначено, і відтоді номери стежать за тим, як він змінюється.

Формат доповнюють дві необов’язкові позначки. Передрелізну версію пишуть після дефіса, і вона стоїть перед релізом, до якого веде: `1.0.0-alpha` іде перед `1.0.0-alpha.1`, далі йдуть `1.0.0-beta`, `1.0.0-rc.1` і нарешті `1.0.0`. Метадані збірки пишуть після знака плюс, як у `1.0.0+20130313144700`, і під час упорядкування версій їх ігнорують, тож дві версії, що відрізняються лише ними, мають однаковий пріоритет. Літера v на початку, як у тегу Git `v1.2.3`, до семантичної версії не належить; специфікація називає її поширеним способом позначити номер версії. Go робить її обов’язковою: канонічна версія модуля Go — це літера v, за якою йде семантична версія.

Застосуйте поняття до рішення

Застосуйте «Що таке семантичне версіонування?» до конкретного рішення. Ми можемо пов'язати поняття із системою, яку ви створюєте або купуєте. Перевірено, випущено і з можливістю відкату, тож зміни доходять до клієнтів у вівторок удень, а не пізно в п'ятницю.

Переглянути наші послуги Зв'язатися з нами →

Перевірте припущення за терміном

Використайте пояснення та пов'язані роботи, щоб сформулювати запитання для проєкту. Ми можемо перевірити припущення разом до вибору інструмента. Автоматизувала розгортання GIS SaaS через CI/CD — розгортання, дані та звітність — усе без ручної роботи

Автоматизувала розгортання GIS SaaS через CI/CD Зв'язатися з нами →

Чому це важливо

Специфікація відштовхується від того, що вона називає пеклом залежностей. Якщо задати залежності надто жорстко, виникає блокування версій: оновлення одного пакета вимагає нових версій усього, що від нього залежить. Якщо надто вільно, виникає безлад у версіях: розраховують на сумісність із більшою кількістю майбутніх версій, ніж розумно. Номер версії з узгодженим значенням дає проєкту змогу сказати, що він прийме: бібліотека, якій потрібна функція, вперше додана у 3.1.0, може приймати будь-що від 3.1.0 до 4.0.0, не включно, і отримувати пізніші виправлення та доповнення без жодної правки вимоги.

Менеджери пакетів діють на підставі цієї обіцянки, нічого не питаючи. У Cargo, менеджері пакетів Rust, звичайна вимога на кшталт `"1.2.3"` — це caret-вимога, яка означає щонайменше 1.2.3, але нижче за 2.0.0. Записуючи `Cargo.lock`, Cargo бере найбільшу версію в цьому діапазоні й лишається на версіях 1.x і після публікації 2.0.0, бо вважає 2.0.0 несумісною. У node-semver, що описує себе як засіб семантичного версіонування для npm, `^1.2.3` означає `>=1.2.3 <2.0.0-0`, а `~1.2.3` дозволяє лише зміни рівня патча, `>=1.2.3 <1.3.0-0`. Go вбудовує правило у шляхи імпорту: починаючи з мажорної версії 2, шлях модуля мусить закінчуватися суфіксом на зразок `/v2`, тож дві несумісні мажорні версії — це два різні шляхи імпорту. Мінімальний вибір версій (minimal version selection) у Go бере для кожного модуля найвищу версію, якої вимагає будь-яка з вимог, і саме вона є мінімумом, що задовольняє їх усі. Результат не зберігається в lock-файлі; кожна команда, що працює з модулями, обчислює його заново.

Ніщо з цього не робить номер гарантією. Це обіцянка тих, хто публікує пакет, і вона варта рівно стільки, скільки їхнє судження про те, яку зміну вважати несумісною. Саме тому Cargo публікує довгий посібник із сумісності, який розкладає зміни на мажорні, мінорні та потенційно несумісні (possibly-breaking). Здебільшого посібник говорить про зміни, через які код, що вчора

компілювався, сьогодні не компілюється. Сам посібник зазначає, що майже будь-яка зміна несе певний ризик для поведінки під час виконання і що питання, чи є така зміна несумісною, мейнтейнери зазвичай вирішують на власний розсуд.

Як це виглядає на практиці

Простежмо релізи вигаданої бібліотеки Rust під назвою tally. Версія 0.4.2 — ще початкова розробка. Версія 1.0.0 оголошує публічний API стабільним. Версія 1.0.1 виправляє помилку округлення й є патчем. Версія 1.1.0 додає функцію `parse_strict` і позначає стару `parse` як застарілу (deprecated): специфікація вимагає для цього мінорного релізу й просить випустити принаймні один такий реліз, перш ніж функцію видалять. Версія 2.0.0 видаляє `parse` і є мажорним релізом. Версія 2.1.0-rc.1 — передрелізна версія наступної мінорної.

Застосунок, що залежить від tally і ще трьох вигаданих крейтів, записує свої вимоги в `Cargo.toml`. Коментарі показують діапазон, який Cargo вичитує з кожного рядка:

```
[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
```

Рядок `ledger` показує домовленість інструментів, а не специфікації: Cargo вважає сигналом несумісності крайнє ліве ненульове число, тоді як специфікація каже, що до 1.0.0 будь-що може змінитися будь-коли. Передрелізні версії Cargo до вимоги не включає, якщо вона прямо не називає одну з них, тож вимога `"2.0"` ніколи не підхопила б реліз-кандидат tally. Точна форма `=` в останньому рядку — саме той випадок, коли, за документацією Cargo, повністю задана вимога може бути виправданою: залежність від передрелізної версії без ризику несумісних змін між передрелізами. Знайдені версії записуються в `Cargo.lock`, а `cargo update` пересуває їх у межах цих діапазонів. З параметром `--locked` збірка, якій довелося б змінити lock-файл, натомість завершується помилкою.

Чого номер не охоплює, то це всього, що лежить поза API. Перевірка ліцензій 644 крейтів Rust, запроваджена у вересні 2026 року, під час кожної збірки звіряє ліцензію кожного крейта зі списком умов, які приймає проєкт, бо рутинне оновлення версії глибоко в графі залежностей може непомітно змінити умови, на яких застосунок можна розповсюджувати, у lock-файлі, який ніхто не читає рядок за рядком. Щоб довести, що перевірка працює, її двічі навмисно змусили спрацювати. Перший раз ліцензійний вираз крейта переписали: так виглядає ситуація, коли автори крейта змінюють його умови між версіями, а рев'ю цього не помічає, бо сам крейт не новий. Другий раз закріплену ревізію ядра змінили без повторної генерації перевірки. Семантична версія може сказати, що 1.4.0 має працювати з кодом, написаним під 1.0.0. Але вона не скаже, чи вам досі можна її постачати.

Де виникають проблеми

Несумісна зміна в мінорному релізі. Таке стається випадково або через інше розуміння того, що вважати несумісним. Посібник Cargo перелічує зміни, які здаються нешкідливими, але такими не є: новий варіант у публічному enum ламає код, що зіставляє його вичерпно, якщо enum від початку не позначили `#[non_exhaustive]`, а в рідкісних випадках із glob-імпортами навіть новий публічний елемент спричиняє помилку компіляції. Відповідь специфікації, щойно помилку помічено, — випустити нову мінорну версію, яка відновлює сумісність, ніколи не змінювати вже опублікований реліз і повідомити користувачам, яка версія порушила правила. На боці споживача таку помилку ловлять lock-файл, збережений у репозиторії, і тести, що запускаються при кожному оновленні, ще до того, як її помітять у продакшні.

Роки на 0.x. За специфікацією нульова мажорна версія взагалі не має стабільного API. Її FAQ каже про це прямо: якщо програмне забезпечення вже працює в продакшні, йому, найімовірніше, вже час мати версію 1.0.0. Інструменти заповнюють цю прогалину власною домовленістю: Cargo і node-semver вважають несумісним крайнє ліве ненульове число, тож `^0.2.3` приймає лише наступні релізи 0.2.x, а `^0.0.3` не приймає жодних оновлень. Бібліотека, що роками стоїть на 0.x, має користувачів, які від неї залежать, але не має перед ними жодної визначеної обіцянки. Випустіть 1.0.0, щойно на API почали покладатися: це змінює номер, а не код.

Вичитувати з номера більше, ніж у ньому є. Сумісність за специфікацією стосується публічного API. Посібник Cargo відносить підвищення мінімальної версії Rust, потрібної крейту, до потенційно несумісних змін і загалом радить

вважати його мінорною зміною, тож мінорне оновлення може зупинити збірку зі старішим набором інструментів Rust. Поле `package.rust-version` дає змогу механізму розв'язання залежностей Cargo за потреби спробувати старіші версії крейта. Умови ліцензії, продуктивність і поведінка, яку API ніколи не документував, лежать поза номером цілком. Перевіряйте їх окремо, перевіркою ліцензій і тестами, а не виводьте з мінорного релізу.

Дві мажорні версії в одній збірці. Коли двом залежностям потрібні несумісні версії одного крейта, скажімо 0.6 і 0.7, Cargo збирає обидві. Компілятор вважає їхні типи різними, навіть коли назви збігаються. Зазвичай це закінчується помилкою компіляції, а іноді проявляється лише під час виконання: документація Cargo описує невдале приведення типу (downcast), коли об'єкт створила бібліотека з версією 1.0.0, а код, що її викликає, приводить його до типу з 2.0.0. Команда `cargo tree -d` показує дубльовані версії і те, звідки вони беруться. Автор бібліотеки може пом'якшити мажорний реліз прийомом, який документація Cargo називає «semver trick»: опублікувати нову мажорну версію, а потім точковий реліз старої, який реекспортує типи з нової.

Що почитати далі

- Semantic Versioning 2.0.0 — специфікація та її FAQ, зокрема що робити, якщо несумісну зміну випустили помилково. Є й український переклад.
- SemVer Compatibility — посібник Cargo Book про те, які зміни в бібліотеці Rust є мажорними, мінорними чи потенційно несумісними, кожна з прикладом.
- Specifying Dependencies — синтаксис вимог Cargo: типові, caret-, tilde- і порівняльні вимоги та те, як зіставляються передрелізні версії.
- Dependency Resolution — як Cargo обирає версії, що зберігає lock-файл і чим небезпечні дві несумісні версії в одному графі.
- node-semver — засіб семантичного версіонування для npm із правилами діапазонів caret, tilde і передрелізних версій.
- Go Modules Reference — суфікси мажорних версій, правило сумісності імпорту та мінімальний вибір версій.



Словник

Обирайте за фактичними умовами

Опишіть шлях розгортання, експлуатаційні ризики та чергову команду. Ми можемо визначити зміну з перевірками, шляхом відновлення та чіткою відповідальністю.

Дивитися, скільки це коштує Зв'язатися з нами →

Зробіть термінологію практичною

Запитайте, як «Що таке семантичне версіонування?» стосується вашої організації. Конкретного прикладу з поточної системи достатньо для початку розмови.

Переглянути наші послуги

welcome@engineer.company +45 50 10 61 84 Копенгаген, Данія

Візьміть цю сторінку із собою

Візитівка

Дбайливо зроблено з любов'ю — D.

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

Авторське право © 2025 – дотепер · Engineer ApS · Усі права захищено.

Надруковано з https://engineer.company/uk/glossary/semantic-versioning/