Engineer

versionado semántico

Numeración de versiones que indica cambios incompatibles

Qué es

El versionado semántico, normalmente abreviado SemVer, es un conjunto de reglas para numerar versiones publicadas de manera que el propio número indique qué clase de cambio contienen. La especificación, actualmente en su versión 2.0.0 y escrita originalmente por Tom Preston-Werner, utiliza tres números con la forma MAJOR.MINOR.PATCH. Una versión que rompe la API pública aumenta el número mayor; una que añade funcionalidad de forma retrocompatible aumenta el menor; y una que solo corrige errores sin tocar la API aumenta el de parche. Cuando aumenta un número, los que quedan a su derecha vuelven a cero. Cada componente se compara como un número, no como texto, de modo que `1.10.0` viene después de `1.9.0` .

Todo se apoya en un requisito: el proyecto debe declarar una API pública, en el código o en su documentación, porque los números describen los cambios de esa API y nada más. Una vez publicada una versión, su contenido no debe modificarse; una corrección constituye una versión nueva. La versión mayor cero, `0.y.z` , se reserva al desarrollo inicial: cualquier cosa puede cambiar en cualquier momento y la API no debe considerarse estable. La versión `1.0.0` marca el punto en el que se define la API pública; desde entonces, los números siguen sus cambios.

Dos etiquetas opcionales amplían el formato. Una versión preliminar se coloca tras un guion y se ordena antes de la versión a la que precede: `1.0.0-alpha` va antes de `1.0.0-alpha.1` , que va antes de `1.0.0-beta` , `1.0.0-rc.1` y, por último, `1.0.0` . Los metadatos de compilación se añaden tras un signo más, como en `1.0.0+20130313144700` , y se ignoran al ordenar versiones: dos versiones que solo difieran en ellos tienen la misma precedencia. La v inicial, como en la etiqueta Git `v1.2.3` , no forma parte de la versión semántica; la especificación la describe

como una forma habitual de marcar un número de versión. Go la hace obligatoria: una versión canónica de un módulo Go es la letra v seguida de una versión semántica.

Utilice el concepto en una decisión

Aplique «¿Qué es el versionado semántico?» a una decisión real. Podemos ayudarle a relacionar el concepto con el sistema que está construyendo o comprando. Cambios probados, publicados y reversibles, para que lleguen a sus clientes un martes por la tarde en vez de un viernes por la noche.

Explore nuestros servicios Póngase en contacto →

Compruebe los supuestos detrás del término

Utilice la explicación y los trabajos enlazados para formular preguntas sobre su proyecto. Podemos examinar los supuestos con usted antes de elegir una herramienta. Despliegue GIS SaaS con GitHub Actions — despliegue, datos e informes automatizados

Despliegue GIS SaaS con GitHub Actions Póngase en contacto →

Por qué importa

La especificación parte de lo que llama el infierno de las dependencias. Si se restringen demasiado, se produce un bloqueo de versiones: actualizar un paquete exige publicar versiones nuevas de todo lo que depende de él. Si se dejan demasiado abiertas, aparece lo que la especificación llama promiscuidad de versiones: aceptar versiones futuras sin una base sólida para esperar compatibilidad. Un número de versión con un significado acordado permite declarar lo que aceptará un proyecto: una biblioteca que necesita una función añadida por primera vez en 3.1.0 puede aceptar cualquier versión desde 3.1.0 hasta, sin incluirla, 4.0.0, y recibir correcciones y ampliaciones posteriores sin que nadie edite el requisito.

Los gestores de paquetes actúan sobre esa promesa sin preguntar. En Cargo, el gestor de paquetes de Rust, un requisito simple como `"1.2.3"` es un requisito con acento circunflejo: significa al menos 1.2.3, pero menos de 2.0.0. Al escribir `Cargo.lock`, Cargo escoge la versión más alta de ese intervalo y permanece en la rama 1.x después de publicarse 2.0.0, porque considera 2.0.0 incompatible. En node-semver, que se presenta como una herramienta de versionado semántico para npm, `^1.2.3` significa `>=1.2.3 <2.0.0-0` y `~1.2.3` solo permite cambios de parche, `>=1.2.3 <1.3.0-0`. Go incorpora la regla a las rutas de importación: desde la versión mayor 2, la ruta de un módulo debe terminar con un sufijo como `/v2`, de modo que dos versiones mayores incompatibles tengan rutas de importación diferentes. La selección de versiones mínimas de Go toma entonces, para cada módulo, la versión más alta solicitada por algún requisito, que es la mínima que los satisface todos. El resultado no se guarda en un archivo de bloqueo: cada comando que trabaja con módulos lo vuelve a calcular.

Nada de esto convierte el número en una garantía. Es una promesa de quienes publican el paquete y depende de su criterio sobre qué constituye una incompatibilidad. Por eso Cargo publica una extensa guía que clasifica los cambios en mayores, menores y potencialmente incompatibles. La mayor parte de esa guía trata de cambios que impiden compilar hoy código que compilaba ayer. La propia guía advierte que casi todo cambio conlleva algún riesgo para el comportamiento en ejecución y que decidir si un cambio así es incompatible suele corresponder a los responsables del proyecto.

Cómo se aplica en la práctica

Siga las versiones de una biblioteca Rust inventada, tally. La versión 0.4.2 sigue siendo desarrollo inicial. La 1.0.0 declara estable la API pública. La 1.0.1 corrige un error de redondeo y es un parche. La 1.1.0 añade la función `parse_strict` y marca la antigua `parse` como obsoleta: la especificación exige una versión menor para declarar obsoleta una función y pide al menos una versión de ese tipo antes de retirarla. La 2.0.0 elimina `parse` y es una versión mayor. La 2.1.0-rc.1 es una versión preliminar de la siguiente versión menor.

Una aplicación que depende de tally y de otras tres crates inventadas declara sus requisitos en `Cargo.toml` . Los comentarios indican el intervalo que Cargo interpreta en cada línea:

```
[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
```

La línea de `ledger` muestra una convención de las herramientas, no de la especificación: Cargo trata el primer número distinto de cero empezando por la izquierda como el que señala una incompatibilidad, mientras que la especificación permite que todo cambie en cualquier momento por debajo de 1.0.0. Cargo excluye las versiones preliminares de un requisito salvo que este nombre una; por tanto, un requisito `"2.0"` nunca elegiría la versión candidata de tally. La forma exacta `=` de la última línea corresponde al caso en el que la documentación de Cargo considera que podría justificarse un requisito completamente especificado: depender de una versión preliminar y evitar cambios incompatibles entre versiones preliminares. Tras resolver las

dependencias, las versiones se registran en `Cargo.lock`, y `cargo update` las mueve dentro de estos intervalos. Con `--locked`, una compilación que necesitara cambiar el archivo de bloqueo devuelve un error.

El número no cubre nada que quede fuera de la API. La auditoría de licencias de 644 crates Rust, realizada en septiembre de 2026, comprueba en cada compilación la licencia de todas las crates frente a una lista de licencias permitidas, porque una actualización rutinaria de versión en lo profundo del grafo de dependencias puede cambiar las condiciones bajo las que se permite distribuir una aplicación, silenciosamente, en un archivo de bloqueo que nadie lee entero. Para demostrar que el control funciona, se provocaron dos fallos deliberados. Una prueba reescribió la expresión de licencia de una crate, reproduciendo el caso de que una crate cambie sus condiciones entre versiones en el proyecto de origen, algo que la revisión humana pasa por alto porque la propia crate no es nueva. La otra cambió la revisión fijada del núcleo sin regenerar la auditoría. Una versión semántica puede decir que se espera que 1.4.0 funcione con código escrito para 1.0.0. No puede decir que siga teniendo permiso para distribuirla.

Dónde surgen los problemas

Un cambio incompatible aparece en una versión menor. Puede ocurrir por accidente o por una interpretación distinta de qué se considera incompatible. La guía de Cargo enumera cambios que parecen inocuos y no lo son: añadir una variante a un enum público rompe el código que trata exhaustivamente todas sus variantes, salvo que el enum se hubiera marcado con `#[non_exhaustive]` desde el principio; y, en casos poco frecuentes relacionados con importaciones con comodín, incluso añadir un elemento público nuevo provoca un error de compilación. Cuando se detecta el error, la especificación pide publicar una nueva versión menor que restablezca la compatibilidad, nunca modificar la ya publicada, e informar a los usuarios de cuál era la versión defectuosa. Del lado consumidor, un archivo de bloqueo incorporado al repositorio y pruebas en cada actualización lo detectan antes de que llegue a producción.

Un proyecto permanece años en 0.x. Según la especificación, la versión mayor cero no ofrece ninguna API estable. Sus preguntas frecuentes son claras: «Si su software se utiliza en producción, probablemente ya debería ser 1.0.0». Las herramientas cubren el vacío con su propia convención: Cargo y node-semver consideran que el primer número distinto de cero empezando por la izquierda indica incompatibilidad, de modo que `^0.2.3` solo acepta más versiones 0.2.x y `^0.0.3` no acepta ninguna actualización. Una biblioteca que lleva años en 0.x

tiene usuarios que dependen de ella, pero ninguna promesa definida para ellos. Publique 1.0.0 cuando haya personas que dependan de la API: cambia un número, no el código.

Se atribuye al número más significado del que tiene. La compatibilidad de la especificación se refiere a la API pública. La guía de Cargo clasifica como potencialmente incompatible elevar la versión mínima de Rust que necesita una crate y, por lo general, recomienda tratarlo como un cambio menor. Por eso una actualización menor puede impedir compilar con una cadena de herramientas antigua. Definir `package.rust-version` permite al resolutor de Cargo intentar elegir versiones anteriores de la crate cuando haga falta. Las condiciones de licencia, el rendimiento y los comportamientos que la API nunca documentó quedan completamente fuera del número. Compruébelos por separado con una auditoría de licencias y pruebas, en vez de deducirlos de una versión menor.

Hay dos versiones mayores en una compilación. Cuando dos dependencias necesitan versiones incompatibles de la misma crate, como 0.6 y 0.7, Cargo compila ambas. El compilador considera distintos sus tipos aunque los nombres coincidan. Esto suele manifestarse como un error de compilación y a veces solo durante la ejecución: la documentación de Cargo describe una conversión descendente que falla porque una biblioteca que utiliza la versión 1.0.0 creó el objeto, mientras el código que realiza la conversión utiliza el tipo de 2.0.0. `cargo tree -d` enumera las versiones duplicadas y su procedencia. El autor de una biblioteca puede suavizar el impacto de una versión mayor con lo que la documentación de Cargo llama el truco semver: publicar la nueva versión mayor y después una actualización puntual de la antigua que vuelva a exportar los tipos de la nueva.

Lecturas adicionales

- Versionado semántico 2.0.0 — la especificación y sus preguntas frecuentes, incluido qué hacer si se publica por error un cambio incompatible.
- Compatibilidad SemVer — la guía del libro de Cargo para clasificar los cambios de una biblioteca Rust como mayores, menores o potencialmente incompatibles, con un ejemplo de cada uno.
- Especificar dependencias — la sintaxis de requisitos de Cargo: requisitos predeterminados, con acento circunflejo, con virgulilla y de comparación, y cómo se seleccionan versiones preliminares.
- Resolución de dependencias — cómo elige versiones Cargo, qué contiene el archivo de bloqueo y los riesgos de dos versiones incompatibles en un grafo.

- node-semver — la herramienta de versionado semántico para npm, con las reglas de intervalos con acento circunflejo, virgulilla y versiones preliminares.
- Referencia de módulos de Go — sufijos de versión mayor, regla de compatibilidad de importaciones y selección de versiones mínimas.

Glosario

Elija según sus restricciones reales

Describa la ruta de despliegue, los riesgos operativos y quiénes se encargan de las guardias. Podemos definir el alcance de un cambio con comprobaciones, una vía de recuperación y responsabilidades claras.

Vea cuánto cuesta Póngase en contacto →

Haga práctica la terminología

Pregunte cómo se aplica «¿Qué es el versionado semántico?» a su organización. Basta un ejemplo concreto de su sistema actual para iniciar la conversación.

Explore nuestros servicios

welcome@engineer.company +45 50 10 61 84 Copenhague, Dinamarca

Llévese esta página

Tarjeta de contacto

Creado con esmero y cariño por D.

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

Derechos de autor © 2025 – actualidad · Engineer ApS · Todos los derechos reservados.

Impreso desde https://engineer.company/es/glossary/semantic-versioning/