

العربية (ar) </lang/ar/> Български (bg) </lang/bg/> català (ca) </lang/ca/> čeština (cs) </lang/cs/>
Dansk (da) </lang/da/> Deutsch (de) </lang/de/> Ελληνικά (el) </lang/el/> English (en) </>
español (es) </lang/es/> فارسی (fa) </lang/fa/> français (fr) </lang/fr/> עברית (he) </lang/he/>
हिन्दी (hin) </lang/hin/> hrvatski (hr) </lang/hr/> magyar (hu) </lang/hu/> Հայերեն (hy) </lang/hy/>
Bahasa Indonesia (id) </lang/id/> italiano (it) </lang/it/> 日本語 (ja) </lang/ja/>
ქართული (ka) </lang/ka/> taqbaylit (kab) </lang/kab/> 한국어 (ko) </lang/ko/>
Nederlands (nl) </lang/nl/> polski (pl) </lang/pl/> português brasileiro (pt-BR) </lang/pt-BR/>
Română (ro) </lang/ro/> русский (ru) </lang/ru/> slovensky (sk) </lang/sk/>
slovenščina (sl) </lang/sl/> srpski (sr) </lang/sr/> svenska (sv) </lang/sv/> Türkçe (tr) </lang/tr/>
українська (uk) </lang/uk/> Tiếng Việt (vi) </lang/vi/> 简体中文 (zh-CN) </lang/zh-CN/>
繁體中文 (zh-TW) </lang/zh-TW/>

2.0.0-rc.2 </lang/es/spec/v2.0.0-rc.2.html>

Versionado Semántico 2.0.0

Resumen

Dado un número de versión MAYOR.MENOR.PARCHE, se incrementa:

1. La versión MAYOR cuando realizas un cambio incompatible en el API,
2. La versión MENOR cuando añades funcionalidad compatible con versiones anteriores, y
3. La versión PARCHE cuando reparas errores compatibles con versiones anteriores.

Hay disponibles etiquetas para prelanzamiento y metadata de compilación como extensiones al formato MAYOR.MENOR.PARCHE.

Introducción

En el mundo de la administración de software existe un temido lugar llamado “Infierno de Dependencias”. Mientras más crece tu sistema y más paquetes integras dentro de tu software, más probable se hace que un día te encuentres en este pozo de desesperación.

En sistemas con muchas dependencias, lanzar nuevas versiones de los paquetes puede convertirse en una pesadilla. Si las especificaciones de la dependencias son muy estrictas, estarás en peligro de bloquear una versión (la inhabilidad de actualizar un paquete sin tener que publicar una nueva versión de cada otro paquete dependiente). Si las dependencias son especificadas de forma muy relajada, inevitablemente serás mordido por versiones promiscuas (asumir la compatibilidad con próximas versiones más allá de lo razonable). El Infierno de Dependencias es donde estás cuando una versión bloqueada y/o promiscua previenen que muevas tu proyecto adelante de forma fácil y segura.

Como solución a este problema, propuse un conjunto simple de reglas y requerimientos que dicten cómo asignar e incrementar los números de la versión. Estas reglas están basadas en prácticas preexistentes de uso generalizado tanto en software de código cerrado como de código abierto, pero no necesariamente limitadas a éstas. Para que este sistema funcione, primero debes declarar un API público. Éste puede consistir en documentación aparte o ser impuesto por el código mismo. Independientemente de lo anterior, es importante que este API sea claro y preciso. Una vez identifiques tu API público, debes comunicar los cambios realizados a éste con incrementos específicos a tu número de versión. Considera un formato de versión X.Y.Z (Mayor.Menor.Parche). Las correcciones de errores que no afectan el API incrementan la versión parche. Adiciones o sustracciones compatibles con versiones anteriores incrementan la versión menor, y cambios en el API incompatibles con versiones anteriores incrementan la versión mayor.

Llamo a este sistema “Versionado Semántico”. Bajo este esquema, los números de versión y la forma en la que cambian transmiten el sentido del código y lo que ha sido modificado de una versión a otra.

Especificación del Versionado Semántico (SemVer)

Las palabras clave “DEBE”, “NO DEBE”, “OBLIGATORIO”, “DEBERÁ”, “NO DEBERÁ”, “DEBERÍA”, “NO DEBERÍA”, “RECOMENDADO”, “PUEDE” y “OPCIONAL” en este documento serán interpretadas como se describe en [RFC 2119-es <https://www.rfc-es.org/rfc/rfc2119-es.txt>](https://www.rfc-es.org/rfc/rfc2119-es.txt).

1. El Software que usa Versionado Semántico DEBE declarar un API público. Este API puede ser declarado en el propio código o debe existir estrictamente en la documentación. Sin importar como sea realizado, este DEBERÍA ser preciso y comprensivo.

2. Un número de versión normal DEBE tener la forma de X.Y.Z donde X, Y y Z son números enteros no negativos, y NO DEBEN ser precedidos de ceros. X es la versión mayor, Y es la versión menor, y Z es la versión parche. Cada elemento DEBE incrementarse numéricamente. Por ejemplo: 1.9.0 -> 1.10.0 -> 1.11.0
3. Una vez que el paquete versionado ha sido publicado, el contenido de esa versión NO DEBE ser modificado. Cualquier modificación DEBE ser publicada como una nueva versión.
4. Una versión mayor en cero (0.y.z) se considera como desarrollo inicial. Todo PUEDE cambiar en cualquier momento. El API público NO DEBERÍA ser considerado estable.
5. La versión 1.0.0 define el API público. La manera en que cada número de versión es incrementado después de esta publicación dependerá de su API público y cómo cambia.

- | | | |
|----|----------------------------|--|
| 6. | La versión parche Z (x.y.Z | x > 0) DEBE ser incrementada si solamente se introducen correcciones de errores compatibles con versiones anteriores. Una corrección de error se define como un cambio interno que corrige un comportamiento incorrecto. |
| 7. | La versión menor Y (x.Y.z | x > 0) DEBE ser incrementada si se introduce funcionalidad nueva y compatible con la versión anterior del API público. Ésta DEBE ser incrementada si se introduce cualquier funcionalidad al API público o mejora al código privado. Este PUEDE incluir cambios a nivel de parches. La versión parche DEBE reiniciarse a 0 cuando una versión menor se incrementa. |
| 8. | La versión mayor (X.y.z | X > 0) DEBE ser incrementada solamente si se introducen cambios incompatibles con la versión anterior del API público. Este PUEDE incluir cambios de nivel menor y parches. Versiones parche y menores DEBEN ser reiniciadas a 0 cuando una versión mayor es incrementada. |
9. Una versión de prelanzamiento PUEDE ser denotada agregando un guión y una serie de identificadores separados por puntos, inmediatamente seguida de la versión parche. Los identificadores DEBEN ser compuestos sólo de caracteres alfanuméricos ASCII y guión ([0-9A-Za-z-]). Los identificadores NO DEBEN estar vacíos. Identificadores numéricos NO DEBEN ser precedidos de cero. Versiones de prelanzamiento tienen una precedencia inferior que una versión normal. Una versión de prelanzamiento indica que esa versión no es estable y puede no satisfacer los requerimientos de compatibilidad destinados como se denota en la versión normal asociada. Por ejemplo: 1.0.0-alpha, 1.0.0-alpha.1, 1.0.0-0.3.7, 1.0.0-x.7.z.92.
 10. Metadatos de compilación PUEDEN ser denotados agregando el signo más y una serie de identificadores separados por puntos, inmediatamente seguido de la versión parche o prelanzamiento. Los identificadores DEBEN ser compuestos de sólo caracteres alfanuméricos ASCII y guión ([0-9A-Za-z-]). Los identificadores NO DEBEN estar vacíos. Identificadores numéricos NO DEBEN ser precedidos de cero. Los metadatos de

compilación DEBEN ser ignorados cuando se determina la precedencia de la versión. Por lo tanto, dos versiones que difieren solamente en los metadatos de compilación tienen la misma precedencia. Por ejemplo: 1.0.0-alpha+001, 1.0.0+20130313144700, 1.0.0-beta+exp.sha.5114f85.

11. La precedencia se refiere a cómo las versiones se comparan entre ellas cuando se ordenan.

1. La precedencia DEBE ser calculada separando los identificadores de la versión en mayor, menor, parche y prelanzamiento (dejando de lado los metadatos de compilación).
2. La precedencia es determinada por la primera diferencia al comparar cada uno de los identificadores de izquierda a derecha como se indica: mayor, menor y parche son siempre comparadas numéricamente.

Por ejemplo: $1.0.0 < 2.0.0 < 2.1.0 < 2.1.1$.

3. Cuando la versión mayor, menor y parche son iguales, la versión de prelanzamiento tiene menor precedencia que la versión normal.

Por ejemplo: $1.0.0\text{-alpha} < 1.0.0$.

4. La precedencia para dos versiones de prelanzamiento con la misma versión mayor, menor y parche DEBE ser determinada comparando cada identificador separado por punto, de izquierda a derecha, hasta que una diferencia sea encontrada como se indica:

1. identificadores compuestos solamente por números son comparados numéricamente
2. los identificadores con letras o guiones son comparados léxicamente en orden ASCII.
3. Identificadores numéricos siempre tienen menor precedencia que identificadores no numéricos.
4. Un conjunto de campos de prelanzamiento más numeroso tiene mayor precedencia que un conjunto menos numeroso, si todos los identificadores anteriores son iguales.

Por ejemplo: $1.0.0\text{-alpha} < 1.0.0\text{-alpha}.1 < 1.0.0\text{-alpha}.beta < 1.0.0\text{-beta} < 1.0.0\text{-beta}.2 < 1.0.0\text{-beta}.11 < 1.0.0\text{-rc}.1 < 1.0.0$.

Forma Gramatical Backus–Naur para Versiones Válidas SemVer

```

<semver válido> ::= <versión núcleo>
    | <versión núcleo> "-" <prelanzamiento>
    | <versión núcleo> "+" <compilación>
    | <versión núcleo> "-" <prelanzamiento> "+" <compilación>

<versión núcleo> ::= <mayor> "." <menor> "." <parche>

<mayor> ::= <identificador numérico>

<menor> ::= <identificador numérico>

<parche> ::= <identificador numérico>

<prelanzamiento> ::= <identificadores prelanzamiento separados-por-puntos>

<identificadores prelanzamiento separados-por-puntos> ::= <identificador prelanzamiento>
    | <identificador prelanzamiento> " " <identificador prelanzamiento>

<compilación> ::= <identificadores compilación separados-por-puntos>

<identificadores compilación separados-por-puntos> ::= <identificador compilación>
    | <identificador compilación> "." <identificador compilación>

<identificador prelanzamiento> ::= <identificador alfanumérico>
    | <identificador numérico>

<identificador compilación> ::= <identificador alfanumérico>
    | <dígitos>

<identificador alfanumérico> ::= <no-dígito>
    | <no-dígito> <caracteres identificadores>
    | <caracteres identificadores> <no-dígito>
    | <caracteres identificadores> <no-dígito> <caracteres identificadores>

<identificador numérico> ::= "0"
    | <dígito positivo>

```

```

| <dígito positivo> <dígitos>

<caracteres identificadores> ::= <carácter identificador>
| <carácter identificador> <caracteres identificadores>

<identificador caracter> ::= <dígito>
| <no-dígito>

<no-dígito> ::= <letra>
| "-"

<dígitos> ::= <dígito>
| <dígito> <dígitos>

<dígito> ::= "0"
| <dígito positivo>

<dígito positivo> ::= "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"

<letra> ::= "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J"
| "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" | "S" | "T"
| "U" | "V" | "W" | "X" | "Y" | "Z" | "a" | "b" | "c" | "d"
| "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m" | "n"
| "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x"
| "y" | "z"

```

¿Por qué usar Versionado Semántico?

Esto no es una idea nueva o revolucionaria. De hecho, probablemente ya tengas algo parecido a ésto. El problema es que “parecido” no significa suficientemente bueno. Sin el cumplimiento de alguna especie de especificación formal, los números de versión son esencialmente inútiles para el manejo de dependencias. Dándole un nombre y una definición clara a las ideas anteriores, se hace fácil comunicar tus intenciones a los

usuarios de tu software. Una vez estas intenciones son claras y flexibles (pero no tanto) las especificaciones de dependencia finalmente se pueden generar.

Un simple ejemplo demuestra cómo el Versionado Semántico puede hacer el Infierno de Dependencias una cosa del pasado. Considera una librería llamada “Camión de bomberos”. Este requiere un paquete versionado semánticamente llamado “Escalera”. Al mismo tiempo que el “Camión de bomberos” es creado, “Escalera” está en su versión 3.1.0. Dado que “Camión de bomberos” usa una funcionalidad que fue introducida en 3.1.0, puedes especificar con seguridad la dependencia “Escalera” igual o mayor a 3.1.0 pero menor a 4.0.0. Ahora, cuando la versión “Escalera” 3.1.1 y 3.2.0 se hagan disponibles, puedes recibirlas en tu gestor de paquetes y saber que son compatibles con tu software existente.

Como desarrollador responsable vas a querer verificar, como deberías, que cada actualización de paquetes funciona como se indica. El mundo real es un lugar desordenado; no hay nada que podamos hacer excepto estar atentos. Lo que puedes hacer es dejar que el Versionado Semántico te otorgue una sana manera de publicar y actualizar paquetes sin tener que crear nuevas versiones de paquetes dependientes, ahorrándote tiempo y molestias.

Si todo esto suena demasiado bueno como para ocuparlo, todo lo que tienes que hacer para usar Versionado Semántico es declarar que los vas a hacer y seguir las reglas. Vincula este sitio web en tu LÉEME para que otros sepan estas reglas y puedan beneficiarse de ellas.

PREGUNTAS FRECUENTES

¿Cómo debería lidiar con las revisiones en la fase inicial de desarrollo 0.y.z?

La manera más simple de hacerlo es comenzar publicando tu desarrollo inicial en 0.1.0 y luego incrementar la versión menor por cada siguiente lanzamiento.

¿Cómo voy a saber cuando publique la versión 1.0.0?

Si tu software está siendo usado en producción, probablemente ya debería ser 1.0.0. Si tienes un API estable del cual los usuarios dependen, debería ser 1.0.0. Si te estás preocupando mucho por la retrocompatibilidad, probablemente ya debería ser 1.0.0.

¿Acaso esto no desalienta el desarrollo rápido y la iteración rápida?

La versión mayor en cero se trata de desarrollo rápido. Si estás cambiando el API cada día entonces aún deberías estar en la versión 0.y.z o en una rama separada de desarrollo trabajando en la siguiente versión.

Si incluso el más pequeño cambio incompatible con versiones anteriores del API público requiere incrementar la versión mayor, ¿Acaso no voy a terminar pronto con una versión 42.0.0?

Esto es una pregunta de desarrollo responsable y precaución. Cambios incompatibles no deberían ser introducidos a la ligera en un software que tiene mucho código del cual depender. El costo que se incurre en actualizar puede ser significativo. Tener que incrementar versiones mayores para publicar cambios incompatibles quiere decir que tendrás que pensar sobre el impacto de tus cambios, y evaluar la proporción costo/beneficio involucrado.

¡Documentar todo el API público es demasiado trabajo!

Es tu responsabilidad como desarrollador profesional el documentar apropiadamente el software destinado a ser utilizados por otros. Administrar la complejidad del software es una parte muy importante para mantener un proyecto eficiente, y eso es difícil de hacer si nadie sabe cómo se usa tu software, o qué métodos son seguros de llamar. A largo plazo, el Versionado Semántico, y la insistencia de un API público bien definido puede mantener a todos y todo funcionando sin contratiempos.

¿Qué pasa si accidentalmente publico una versión incompatible como versión menor?

Tan pronto como te des cuenta que has roto las especificaciones del Versionado Semántico, arregla el problema y publica una nueva versión menor que corrija el problema y restaure la retrocompatibilidad. Incluso bajo estas circunstancias, es inaceptable modificar publicaciones versionadas. Si es apropiado, documenta la versión problemática e informa a tus usuarios sobre el percance para que todos sepan de esta versión problemática.

¿Qué debería hacer si actualizo mis propias dependencias sin cambiar el API público?

Eso sería considerado compatible dado que no afecta el API público. El software que explícitamente dependa de las mismas dependencias que tu paquete deberían tener su propias especificaciones de dependencias y el autor se dará cuenta de cualquier conflicto. Determinar si el cambio requiere modificar la versión parche o la versión menor dependerá en si has actualizado tus dependencias para arreglar un error o introducir nueva funcionalidad. Usualmente sospecho de código adicional sobre lo último, en ese caso es obviamente un incremento a la versión menor.

¿Qué pasa si altero el API público sin darme cuenta, de una manera que deja de cumplir con el número de la versión, como cuando el código incorrectamente introduce un cambio importante en una versión parche?

Usa tu mejor juicio. Si tienes una gran audiencia que será drásticamente impactada al devolver el comportamiento a lo que pretendía el API público, entonces quizás sea mejor realizar un lanzamiento de una versión mayor, incluso si una corrección pudiera ser estrictamente considerada una versión parche. Recuerda, el Versionado Semántico se trata de darle sentido a cómo un número de versión cambia. Si estos cambios son importantes para tus usuarios, usa el número de versión para informarles.

¿Cómo debería manejar funcionalidad obsoleta?

Desechar funcionalidad existente es algo normal en el desarrollo de software y es usualmente requerido para progresar. Cuando desechas parte de tu API público, deberías hacer dos cosas: (1) actualizar tu documentación para avisar a tus usuarios sobre el cambio, (2) lanzar una nueva versión menor manteniendo la obsolescencia en su lugar. Antes que remuevas completamente la funcionalidad en una nueva versión mayor debería haber al menos una versión menor anterior que contenga la obsolescencia para que todos los usuarios puedan planear una transición sin sobresaltos al nuevo API.

¿Tiene SemVer una longitud límite en el texto de la versión?

No, pero usa tu juicio. Por ejemplo, una versión de 255 caracteres probablemente sea exagerada. También, algunos sistemas en específico pueden imponer sus propios límites en la longitud del texto.

¿Es “v1.2.3” una versión semántica?

No, “v1.2.3” no es una versión semántica. Sin embargo, anteponer una versión semántica con un “v” es una forma muy generalizada (en inglés) de indicar que es un número de versión. Abreviar “version” como “v” es visto muy a menudo en programas de control de versiones. Por ejemplo: `git tag v1.2.3 -m "Release version 1.2.3"`, en donde “v1.2.3” es el nombre de la etiqueta y la versión semántica es “1.2.3”.

¿Hay alguna expresión regular (Regex) sugerida para verificar un texto SemVer?

Hay dos. Uno incluye grupos con nombre para sistemas que los soportan (PCRE [Perl Compatible Regular Expressions, i.e. Perl, PHP and R], Python y Go)


Mira: <https://regex101.com/r/Ly7O1x/3/> <<https://regex101.com/r/Ly7O1x/3/>>

```
^(?P<major>0|[1-9]\d*)\.(?P<minor>0|[1-9]\d*)\.(?P<patch>0|[1-9]\d*)(?:-(?P<prerelease>(?:
```

Y el otro, en cambio, incluye grupos capturados numéricamente (así que cg1 = mayor, cg2 = menor, cg3 = parche, cg4 = prelanzamiento y cg5 = metadatos de compilación) que es compatible con ECMA Script (JavaScript), PCRE (Perl Compatible Regular Expressions, i.e. Perl, PHP and R), Python y Go.

Mira: <https://regex101.com/r/vkijKf/1/> <<https://regex101.com/r/vkijKf/1/>>

```
^(0|[1-9]\d*)\.(0|[1-9]\d*)\.(0|[1-9]\d*)(?:-((?:0|[1-9]\d*|\d*[a-zA-Z-][0-9a-zA-Z-]*))(?:\
```



Acerca de

La especificación de Versionado Semántico ha sido escrita por [Tom Preston-Werner](http://tom.preston-werner.com) <<http://tom.preston-werner.com>>, inventor de Gravatar y co-fundador de GitHub.

Si quieres dejar comentarios, porfavor [abre un problema \(issue\) en Github](https://github.com/semver/semver/issues) <<https://github.com/semver/semver/issues>>.

Traducción

Esta traducción ha sido realizada por [Italo Baeza Cabrera](https://italobc.com) <<https://italobc.com>>. Si deseas editar esta traducción, puedes [dirigirte al respectivo repositorio en Github](https://github.com/semver/semver.org/blob/gh-pages/lang/es/index.md) <<https://github.com/semver/semver.org/blob/gh-pages/lang/es/index.md>>

Licencia

[Creative Commons — CC BY 3.0](http://creativecommons.org/licenses/by/3.0/) <<http://creativecommons.org/licenses/by/3.0/>> ([en español](https://creativecommons.org/licenses/by/3.0/deed.es) <<https://creativecommons.org/licenses/by/3.0/deed.es>>)