Engineer

Versionnage sémantique

Numérotation des versions signalant les incompatibilités

Ce que c’est

Le versionnage sémantique, généralement abrégé en SemVer, est un ensemble de règles de numérotation des publications, afin que le numéro lui-même indique la nature des changements qu’elles contiennent. La spécification, actuellement en version 2.0.0 et rédigée à l’origine par Tom Preston-Werner, utilise trois nombres sous la forme MAJOR.MINOR.PATCH. Une publication qui rompt la compatibilité de l’API publique augmente le numéro majeur ; celle qui ajoute des fonctionnalités de manière rétrocompatible augmente le numéro mineur ; celle qui corrige uniquement des bogues sans toucher à l’API augmente le numéro de correctif. Lorsqu’un nombre augmente, ceux à sa droite reviennent à zéro. Chacun est comparé comme un nombre plutôt que comme du texte : `1.10.0` vient donc après `1.9.0` .

Tout repose sur une exigence : le projet doit déclarer une API publique, dans le code ou dans sa documentation, car les numéros décrivent les changements de cette API et rien d’autre. Une fois une version publiée, son contenu ne doit plus changer ; une correction constitue une nouvelle version. La version majeure zéro, `0.y.z` , correspond au développement initial, où tout peut changer à tout moment et où l’API ne doit pas être considérée comme stable. La version `1.0.0` est le moment où l’API publique est définie. Dès lors, les numéros suivent son évolution.

Deux indications facultatives complètent le format. Une préversion suit un trait d’union et se classe avant la version qu’elle prépare : `1.0.0-alpha` précède `1.0.0-alpha.1` , qui précède `1.0.0-beta` , puis `1.0.0-rc.1` et enfin `1.0.0` . Les métadonnées de construction suivent un signe plus, comme dans `1.0.0+20130313144700` , et sont ignorées lors du classement : deux versions qui ne diffèrent que par elles ont donc le même rang. Un v initial, comme dans l’étiquette Git `v1.2.3` , ne fait pas partie de la version sémantique ; la

spécification le présente comme une manière courante de marquer un numéro de version. Go le rend obligatoire : une version canonique de module Go est la lettre v suivie d'une version sémantique.

Utilisez le concept pour prendre une décision

Utilisez « Qu'est-ce que le versionnage sémantique ? » pour éclairer une décision réelle. Nous pouvons vous aider à relier ce concept au système que vous construisez ou achetez. Des changements testés, publiés et réversibles, pour livrer aux clients un mardi après-midi plutôt qu'un vendredi soir tard.

Découvrir nos prestations Contactez-nous →

Vérifiez les hypothèses derrière le terme

Appuyez-vous sur l'explication et les réalisations liées pour formuler les questions de votre projet. Nous pouvons examiner les hypothèses avec vous avant le choix d'un outil. Déploiement SaaS GIS automatisé : GitHub Actions — déploiement, données et reporting automatisés

Déploiement SaaS GIS automatisé : GitHub Actions Contactez-nous →

Pourquoi c'est important

La spécification part de ce qu'elle appelle l'enfer des dépendances. Des contraintes trop strictes provoquent un blocage de versions : mettre à jour un paquet impose de publier de nouvelles versions de tout ce qui en dépend. Des contraintes trop lâches produisent ce qu'elle appelle la promiscuité des versions : accepter de futures publications sans fondement solide pour en attendre la compatibilité. Un numéro dont le sens est convenu permet au projet d'énoncer ce qu'il accepte. Une bibliothèque qui a besoin d'une fonction ajoutée en 3.1.0 peut accepter toute version à partir de 3.1.0, jusqu'à 4.0.0 exclue, et recevoir les corrections et ajouts ultérieurs sans modifier la contrainte.

Les gestionnaires de paquets agissent sur cette promesse sans demander confirmation. Dans Cargo, le gestionnaire de paquets de Rust, une contrainte simple comme `"1.2.3"` est une contrainte avec accent circonflexe et signifie au moins 1.2.3, mais moins de 2.0.0. Lorsqu'il écrit `Cargo.lock` , Cargo choisit la version la plus élevée de cet intervalle. Il reste sur la branche 1.x après la publication de 2.0.0, car il considère 2.0.0 comme incompatible. Dans node-semver, qui se présente comme l'outil de versionnage sémantique de npm, `^1.2.3` signifie `>=1.2.3 <2.0.0-0` , et `~1.2.3` n'autorise que des changements de niveau correctif, `>=1.2.3 <1.3.0-0` . Go inscrit la règle dans les chemins d'importation : à partir de la version majeure 2, le chemin d'un module doit se terminer par un suffixe comme `/v2` . Deux versions majeures incompatibles ont donc deux chemins d'importation différents. La sélection minimale de versions de Go retient ensuite, pour chaque module, la plus élevée des versions demandées par les différentes contraintes, soit la version minimale qui les satisfait toutes. Le résultat n'est pas enregistré dans un fichier de verrouillage : chaque commande prenant en charge les modules le recalcule.

Rien de cela ne fait du numéro une garantie. C'est une promesse des auteurs du paquet, qui vaut seulement ce que vaut leur jugement sur les ruptures de compatibilité. C'est pourquoi Cargo publie un long guide de compatibilité, classant les changements en majeurs, mineurs et potentiellement incompatibles. La majeure partie de ce guide concerne les changements qui empêchent de compiler aujourd'hui un code qui compilait hier. Le guide précise lui-même que presque tout changement comporte un risque pour le comportement à l'exécution, et que son éventuelle incompatibilité relève généralement de l'appréciation des responsables du projet.

À quoi cela ressemble en pratique

Suivons une bibliothèque Rust fictive, tally, au fil de ses versions. La version 0.4.2 est encore en développement initial. La version 1.0.0 déclare l'API publique stable. La version 1.0.1 corrige un bogue d'arrondi et constitue un correctif. La version 1.1.0 ajoute la fonction `parse_strict` et marque l'ancienne fonction `parse` comme obsolète : la spécification exige une version mineure pour signaler une obsolescence et demande au moins une telle publication avant de supprimer la fonction. La version 2.0.0 supprime `parse` et constitue une publication majeure. La version 2.1.0-rc.1 est une préversion de la prochaine version mineure.

Une application qui dépend de tally et de trois autres crates fictives énonce ses contraintes dans `Cargo.toml`. Les commentaires donnent l'intervalle que Cargo déduit de chaque ligne :

```
[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 ligne `ledger` montre une convention des outils plutôt que de la spécification : Cargo considère que le nombre non nul le plus à gauche signale une rupture, tandis que la spécification dit que tout peut changer à tout moment en dessous de 1.0.0. Cargo exclut les préversions d'une contrainte, sauf si celle-ci en nomme une : une contrainte `"2.0"` ne choisirait donc jamais la version candidate de tally. La forme exacte avec `=` sur la dernière ligne correspond au cas où la documentation de Cargo considère qu'une contrainte entièrement spécifiée peut se justifier : dépendre d'une préversion tout en évitant les ruptures entre

préversions. Après résolution, les versions sont consignées dans `Cargo.lock`, et `cargo update` les fait évoluer à l'intérieur de ces intervalles. Avec `--locked`, une construction qui devrait modifier le fichier de verrouillage renvoie une erreur.

Le numéro ne couvre rien de ce qui est extérieur à l'API. L'audit de licence de 644 crates Rust, réalisé en septembre 2026, vérifie la licence de chaque crate à chaque construction par rapport à une liste d'autorisation. Une mise à jour banale au fond du graphe de dépendances peut en effet modifier les conditions de distribution d'une application, sans que cela se remarque, dans un fichier de verrouillage que personne ne lit en entier. Pour prouver le fonctionnement de la vérification, deux échecs ont été provoqués volontairement. Un test a réécrit l'expression de licence d'une crate : c'est le cas d'une crate qui change ses conditions en amont entre deux versions, ce qui échappe à une revue humaine parce que la crate elle-même n'est pas nouvelle. L'autre a changé la révision fixée du cœur sans régénérer l'audit. Une version sémantique peut vous dire que 1.4.0 est censée fonctionner avec du code écrit pour 1.0.0. Elle ne peut pas vous dire si vous avez encore le droit de la distribuer.

Où les problèmes surviennent

Une rupture de compatibilité dans une version mineure. Cela arrive par accident ou parce qu'on ne définit pas la rupture de la même manière. Le guide de Cargo cite des changements apparemment anodins qui ne le sont pas : ajouter une variante à une énumération publique casse le code qui en examine tous les cas, sauf si l'énumération portait `#[non_exhaustive]` dès l'origine. Dans de rares cas impliquant des importations avec joker, même l'ajout d'un nouvel élément public provoque une erreur de compilation. Une fois l'erreur constatée, la spécification demande de publier une nouvelle version mineure rétablissant la compatibilité, de ne jamais modifier la publication existante et d'indiquer aux utilisateurs la version fautive. Du côté du projet qui l'utilise, un fichier de verrouillage versionné et des tests exécutés à chaque mise à jour détectent le problème avant la production.

Rester des années en 0.x. Selon la spécification, la version majeure zéro ne garantit aucune stabilité de l'API. Sa FAQ est directe : « Si votre logiciel est utilisé en production, il devrait probablement déjà être en 1.0.0. » Les outils comblent ce vide par leur propre convention : Cargo et node-semver prennent le nombre non nul le plus à gauche comme celui qui signale une rupture. Ainsi, `^0.2.3` n'accepte que les versions ultérieures de la branche 0.2.x, et `^0.0.3` n'accepte aucune mise à jour. Une bibliothèque restée des années en 0.x a des utilisateurs

qui en dépendent, mais aucune promesse définie envers eux. Publiez la version 1.0.0 lorsque des personnes dépendent de l'API : cela change un numéro, pas le code.

Faire dire au numéro plus qu'il ne dit. La compatibilité au sens de la spécification concerne l'API publique. Le guide de Cargo classe l'augmentation de la version minimale de Rust requise par une crate comme potentiellement incompatible et recommande généralement de la traiter comme un changement mineur. Une mise à jour mineure peut donc empêcher la construction avec une ancienne chaîne d'outils. Définir `package.rust-version` permet au résolveur de Cargo d'essayer au besoin des versions plus anciennes de la crate. Les conditions de licence, les performances et les comportements jamais documentés par l'API sont entièrement hors du champ du numéro. Vérifiez-les séparément par un audit de licence et des tests, plutôt que de les déduire d'une publication mineure.

Deux versions majeures dans une construction. Lorsque deux dépendances requièrent des versions incompatibles d'une même crate, par exemple 0.6 et 0.7, Cargo construit les deux. Le compilateur considère leurs types comme différents, même si les noms coïncident. Cela se manifeste généralement par une erreur de compilation, et parfois seulement à l'exécution : la documentation de Cargo décrit une conversion vers un type plus précis qui échoue, car l'objet a été créé par une bibliothèque utilisant 1.0.0 alors que le code appelant tente de le convertir vers le type de 2.0.0. `cargo tree -d` énumère les versions en double et leurs origines. Un auteur de bibliothèque peut atténuer l'effet d'une version majeure par ce que la documentation de Cargo appelle l'astuce semver : publier la nouvelle version majeure, puis une petite mise à jour de l'ancienne branche qui réexporte les types de la nouvelle.

Pour aller plus loin

- Versionnage sémantique 2.0.0 — la spécification et sa FAQ, notamment la conduite à tenir après une rupture de compatibilité publiée par erreur.
- Compatibilité SemVer — le guide du Cargo Book qui classe les changements d'une bibliothèque Rust en majeurs, mineurs ou potentiellement incompatibles, avec un exemple pour chacun.
- Spécifier les dépendances — la syntaxe des contraintes Cargo : par défaut, avec accent circonflexe, tilde ou comparaison, et la sélection des préversions.
- Résolution des dépendances — comment Cargo choisit les versions, ce que contient le fichier de verrouillage et les risques liés à deux versions incompatibles dans un même graphe.

- node-semver — l’outil de versionnage sémantique de npm, avec ses règles d’intervalles pour accents circonflexes, tildes et préversions.
- Référence des modules Go — les suffixes de version majeure, la règle de compatibilité des importations et la sélection minimale de versions.

Glossaire

Choisissez selon vos contraintes réelles

Présentez le parcours de déploiement, les risques opérationnels et les personnes d'astreinte. Nous pouvons définir un changement avec des vérifications, une procédure de reprise et une responsabilité claire.

Consulter les tarifs Contactez-nous →

Rendez la terminologie concrète

Demandez comment « Qu’est-ce que le versionnage sémantique ? » s'applique à votre organisation. Un exemple concret tiré de votre système actuel suffit pour engager la discussion.

Découvrir nos prestations

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

Emportez cette page avec vous

Fiche de contact

Créé avec soin et amour par D.

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

Droit d'auteur © 2025 – aujourd'hui · Engineer ApS · Tous droits réservés.

Imprimé depuis https://engineer.company/fr/glossary/semantic-versioning/