语义化版本 宣讲

Semantic Versioning 2.0.0

2023.08.31 vviz.

X.Y.Z-<pre-release>+<build>

e.g.

1.0.0-beta.0+exp.sha.5114f85

什么是语义化版本

语义化版本是一种规范的版本控制方法，用简单易懂的数字标识软件的更新程度，方便开发者和用户了解软件的改动程度和兼容性情况。

简单概括：

1. 一套运作软件版本号的规范
2. 通过版本号就能了解软件的更新程度

为什么使用语义化版本

以前，软件更新不像现在这样频繁，版本号通常只是简单的数字或者字母。

后来，软件构建依赖大量第三方软件包，当依赖包发布了新版本，是否更新依赖包存在巨大的挑战。

因为仅通过简单的数字或者字母，开发者不知道依赖包是破坏性的改变，还是向下兼容的功能性增强或者错误修复。

当版本号被人为赋予语义时，就具备了这些重要的信息。

就像变量命名，有语义，才好理解

```
// bad
const v = '1.0.0'
// vague
const version = '1.0.0'
```

```
// good
const semanticVersioning= '1.0.0'
// nice
const semver = '1.0.0'
```

常见的版本规范

常见的版本规范

- 语义化版本（Semantic Versioning）：MAJOR.MINOR.PATCH-<pre-release>+<build>
- 四部分版本号（Four-Part Version Numbers）：MAJOR.MINOR.BUILD.PATCH，如 Chrome，Windows
- 日期版本（Date Versioning）：YYYY.MM.DD，如 Ubuntu
- 其他以食物、动物或地名作为版本，如 Android，Ubuntu，Mac OS，Windows

语义化版本的结构

MAJOR.MINOR.PATCH-<pre-release>+<build>

or

X.Y.Z-<pre-release>+<build>

MAJOR.MINOR.PATCH-<pre-release>+<build>

- MAJOR：主版本号
- MINOR：次版本号
- PATCH：修订号
- pre-release：先行版本号
- build：编译信息 (Build metadata)

e.g.

1.0.0-beta.0+exp.sha.5114f85
1.0.0-beta.0+20230827

实际上，版本号中的语义是软件包的开发者想要传达给软件包的使用者的。

两者都需要理解软件包采用的语义化版本的规则才能理解版本号的含义。

语义化版本的规则

`MAJOR.MINOR.PATCH` 规则

1. 三个版本号必须是非负的整数，且前面不能补 0
2. 在合适的时机递增某个版本号后，后面的版本号必须重置为 0

递增

1. 软件包的开发者，做了向下兼容的问题修正，会递增 `PATCH`
2. 软件包的开发者，做了向下兼容的功能性新增或修改，会递增 `MINOR`
3. 软件包的开发者，做了不兼容的 API 修改，会递增 `MAJOR`

所以，`MAJOR.MINOR.PATCH` 有一个非常容易理解的非官方解释：`BREAKING.FEATURE.FIX`。

e.g. 软件包当前版本为 1.0.1

1. 做了向下兼容的问题修正，应升级为 1.0.2
2. 做了向下兼容的功能性新增或修改，1.1.0
3. 做了不兼容的 API 修改，2.0.0

`pre-release` 规则

> 当我们开发或者维护一个软件包，在发布它的下一个（无论是递增 X、Y 还是 Z）正式版本前，一般来说需要经过内测 (alpha)、公测 (beta)、候选发布 (rc) 版本，此前都称作 `pre-release` (先行版本或预发布版本)。

1. `Z` 版本后面需要先加上一个减号 `-`，才能跟 `pre-release`
2. `pre-release` 是一连串以英文句点分隔的标识符
3. 每串被分隔的标识符只能由 `[0-9A-Za-z-]` 组成

e.g. Vue 团队在 2.6.11 后发布了 3.0.0-alpha.0，一直到 3.0.0-alpha.13 才发布 3.0.0-beta.1，一直到 3.0.0-beta.24 才发布 3.0.0-rc.1，一直到 3.0.0-rc.13 才正式发布 3.0.0。

可能是觉得从 0 开始不太好数，Vue 团队后面没再发布过 alpha.0、beta.0 或 rc. 0，直接从 1 开始。

`build` 规则

> 用来向版本号附加编译信息。

1. `Z` 或 `pre-release` 之后需要先加上一个加号`+`，才能跟 `build`
2. `build` 是一连串以英文句点分隔的标识符
3. 每串被分隔的标识符只能由 `[0-9A-Za-z-]` 组成

e.g. 1.0.0-alpha+001、1.0.0+20130313144700、1.0.0-beta.0+exp.sha.5114f85

语义化版本的优先级

同一软件包，优先级越高的版本越新，反之亦然。

e.g.

1. 1.0.0 < 2.0.0 < 2.1.0 < 2.1.1
2. 1.0.0-alpha < 1.0.0
3. 1.0.0-alpha < 1.0.0-alpha.1 < 1.0.0-alpha.beta < 1.0.0-beta < 1.0.0-beta.2 < 1.0.0-beta.11 < 1.0.0-rc.1 < 1.0.0

大概就是：从左到右开始比较，X > Y > Z > pre-release，具体的排序规则参考：版本排序

⚠注意：`build` 不参与优先级排序

程序上可使用 npm semver 来验证

```
const semver = require('semver')

semver.gt('1.2.3', '9.8.7') // false
semver.lt('1.2.3', '9.8.7') // true
semver.minVersion('>=1.0.0') // '1.0.0'
semver.satisfies('1.2.3', '^1.2.0') // true
semver.satisfies('1.2.3', '1.x || >=2.5.0 || 5.0.0 - 7.2.3') // true
semver.inc('1.2.3', 'prerelease', 'beta') // '1.2.4-beta.0'
semver.inc('1.2.4-beta.0', 'prerelease', 'beta') // '1.2.4-beta.1'
```

冷知识：当运行 `npm install` 时，install 程序内部就是依据 `semver` 来确保要安装的包的正确版本。

FAQ

FAQ

1. 在 0.y.z 初始开发阶段，如何进行版本控制?

   以 0.1.0 作为你的初始化开发版本，并在后续的每次发行时递增 Y 版本号。0.y.z 阶段很特殊，意味着版本 API 不稳定，随时可能会改变，在 API 稳定之前，不应该草率发布 1.0.0 版本

2. 如何处理即将弃用的 API?

   1. 更新你的文档让使用者知道这个改变
   2. 在适当的时机将弃用的功能透过新的 Y 版本号发布
   3. 在发布 X 版本，完全移除弃用功能前，至少要有一个 Y 版本包含这个弃用信息

3. 万一不小心破坏了语义化版本，发布了不正确的版本怎么办?

   发布新的 Y 版本恢复向下兼容，必要时发布 X 版本

实用 CLI

yarn

```
# 发布 beta 版本, yarn add <package> 不会安装此版本
# 除非 yarn add <package>@beta
yarn publish --tag beta

# 查看 package 信息
yarn info <package>

# 查看 package 所有版本
yarn info <package> versions
```

pnpm

```
# 发布 beta 版本, pnpm add <package> 不会安装此版本
# 除非 pnpm add <package>@beta
pnpm publish --tag beta

# 查看 package 信息
pnpm view <package>

# 查看 package 所有版本
pnpm view <package> versions
```

一些良好的习惯

一些良好的习惯

1. `lock` 文件需 git push，否则 install 时，可能会安装到未正确发布版本且存在 BUG 的依赖包（因为 ^ ~ >= 这些范围标识符）
2. 手动修改了 `package.json` 文件中的依赖项，请执行 install 更新 lock 文件，并 git push
3. 作为一位负责任的开发者，理当确保每次的 commit message 与语义化版本一致
4. 作为一位负责任的开发者，理当确保每次包升级的运作与版本号的语义一致
5. 不兼容的改变不应该轻易被加入到有许多依赖代码的软件中
6. 开发公共库，至少需要发布一个测试版本验证后，再发布正式版本

Thanks