语义化版本 2.0.0

摘要

版本格式：主版本号.次版本号.修订号，版本号递增规则如下：

1. 主版本号：当你做了不兼容的 API 修改，
2. 次版本号：当你做了向下兼容的功能性新增，
3. 修订号：当你做了向下兼容的问题修正。

先行版本号及版本编译信息可以加到“主版本号.次版本号.修订号”的后面，作为延伸。

简介

在软件管理的领域里存在着被称作“依赖地狱”的死亡之谷，系统规模越大，加入的套件越多，你就越有可能在未来的某一天发现自己已深陷绝望之中。

在依赖高的系统中发布新版本套件可能很快会成为恶梦。如果依赖关系过高，可能面临版本控制被锁死的风险（必须对每一个相依套件改版才能完成某次升级）。而如果依赖关系过于松散，又将无法避免版本的混乱（假设兼容于未来的多个版本已超出了合理数量）。当你专案的进展因为版本相依被锁死或版本混乱变得不够简便和可靠，就意味着你正处于依赖地狱之中。

作为这个问题的解决方案之一，我提议用一组简单的规则及条件来约束版本号的配置和增长。这些规则是根据（但不局限于）已经被各种封闭、开放源码软件所广泛使用的惯例所设计。为了让这套理论运作，你必须先有定义好的公共 API。这可以透过文件定义或代码强制要求来实现。无论如何，这套 API 的清楚明了是十分重要的。一旦你定义了公共 API，你就可以透过修改相应的版本号来向大家说明你的修改。考虑使用这样的版本号格式：X.Y.Z（主版本号.次版本号.修订号）修复问题但不影响 API 时，递增修订号；API 保持向下兼容的新增及修改时，递增次版本号；进行不向下兼容的修改时，递增主版本号。

我称这套系统为“语义化的版本控制”，在这套约定下，版本号及其更新方式包含了相邻版本间的底层代码和修改内容的信息。

语义化版本控制规范（SemVer）

以下关键词 MUST、MUST NOT、REQUIRED、SHALL、SHALL NOT、SHOULD、SHOULD NOT、RECOMMENDED、MAY、OPTIONAL 依照 RFC 2119 的叙述解读。（译注：为了保持语句顺畅， 以下文件遇到的关键词将依照整句语义进行翻译，在此先不进行个别翻译。）

1. 使用语义化版本控制的软件“必须 MUST ”定义公共 API。该 API 可以在代码中被定义或出现于严谨的文件内。无论何种形式都应该力求精确且完整。

2. 标准的版本号“必须 MUST ”采用 XYZ 的格式，其中 X、Y 和 Z 为非负的整数，且“禁止 MUST NOT”在数字前方补零。X 是主版本号、Y 是次版本号、而 Z 为修订号。每个元素“必须 MUST ”以数值来递增。例如：1.9.1 -> 1.10.0 -> 1.11.0。

3. 标记版本号的软件发行后，“禁止 MUST NOT ”改变该版本软件的内容。任何修改都“必须 MUST ”以新版本发行。

4. 主版本号为零（0.y.z）的软件处于开发初始阶段，一切都可能随时被改变。这样的公共 API 不应该被视为稳定版。

5. 1.0.0 的版本号用于界定公共 API 的形成。这一版本之后所有的版本号更新都基于公共 API 及其修改内容。

6. 修订号 Z（x.y.Z ｜ x > 0）“必须 MUST ”在只做了向下兼容的修正时才递增。这里的修正指的是针对不正确结果而进行的内部修改。

7. 次版本号 Y（x.Y.z ｜ x > 0）“必须 MUST ”在有向下兼容的新功能出现时递增。在任何公共 API 的功能被标记为弃用时也“必须 MUST ”递增。也“可以 MAY ”在内部程序有大量新功能或改进被加入时递增，其中“可以 MAY ”包括修订级别的改变。每当次版本号递增时，修订号“必须 MUST ”归零。

8. 主版本号 X（X.y.z ｜ X > 0）“必须 MUST ”在有任何不兼容的修改被加入公共 API 时递增。其中“可以 MAY ”包括次版本号及修订级别的改变。每当主版本号递增时，次版本号和修订号“必须 MUST ”归零。

9. 先行版本号“可以 MAY ”被标注在修订版之后，先加上一个连接号再加上一连串以句点分隔的标识符号来修饰。标识符号“必须 MUST ”由 ASCII 码的英数字和连接号 [0-9A-Za-z-] 组成，且“禁止 MUST NOT ”留白。数字型的标识符号“禁止 MUST NOT ”在前方补零。先行版的优先级低于相关联的标准版本。被标上先行版本号则表示这个版本并非稳定而且可能无法达到兼容的需求。范例：1.0.0-alpha、1.0.0-alpha.1、1.0.0-0.3.7、1.0.0-x.7.z.92。

10. 版本编译信息“可以 MAY ”被标注在修订版或先行版本号之后，先加上一个加号再加上一连串以句点分隔的标识符号来修饰。标识符号“必须 MUST ”由 ASCII 的英数字和连接号 [0-9A-Za-z-] 组成，且“禁止 MUST NOT ”留白。当判断版本的优先层级时，版本编译信息“可 SHOULD ”被忽略。因此当两个版本只有在版本编译信息有差别时，属于相同的优先层级。范例：1.0.0-alpha+001、1.0.0+20130313144700、1.0.0-beta+exp.sha.5114f85。

11. 版本的优先层级指的是不同版本在排序时如何比较。判断优先层级时，“必须 MUST ”把版本依序拆分为主版本号、次版本号、修订号及先行版本号后进行比较（版本编译信息不在这份比较的列表中）。由左到右依序比较每个标识符号，第一个差异值用来决定优先层级：主版本号、次版本号及修订号以数值比较，例如：1.0.0 < 2.0.0 < 2.1.0 < 2.1.1。当主版本号、次版本号及修订号都相同时，改以优先层级比较低的先行版本号决定。例如：1.0.0-alpha < 1.0.0。有相同主版本号、次版本号及修订号的两个先行版本号，其优先层级“必须 MUST ”透过由左到右的每个被句点分隔的标识符号来比较，直到找到一个差异值后决定：只有数字的标识符号以数值高低比较，有字母或连接号时则逐字以 ASCII 的排序来比较。数字的标识符号比非数字的标识符号优先层级低。若开头的标识符号都相同时，栏位比较多的先行版本号优先层级比较高。范例：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。

为什么要使用语义化的版本控制?

这并不是一个新的或者革命性的想法。实际上，你可能已经在做一些近似的事情了。问题在于只是“近似”还不够。如果没有某个正式的规范可循，版本号对于依赖的管理并无实质意义。将上述的想法命名并给予清楚的定义，让你对软件使用者传达意向变得容易。一旦这些意向变得清楚，弹性（但又不会太弹性）的依赖规范就能达成。

举个简单的例子就可以展示语义化的版本控制如何让依赖地狱成为过去。假设有个名为“救火车”的函式库，它需要另一个名为“梯子”并已经有使用语义化版本控制的套件。当救火车创建时，梯子的版本号为 3.1.0。因为救火车使用了一些版本 3.1.0 所新增的功能， 你可以放心地指定相依于梯子的版本号大等于 3.1.0 但小于 4.0.0。这样，当梯子版本 3.1.1 和 3.2.0 发布时，你可以将直接它们纳入你的套件管理系统，因为它们能与原有相依的软件兼容。

作为一位负责任的开发者，你理当确保每次套件升级的运作与版本号的表述一致。现实世界是复杂的，我们除了提高警觉外能做的不多。你所能做的就是让语义化的版本控制为你提供一个健全的方式来发行以及升级套件，而无需推出新的相依套件，节省你的时间及烦恼。

如果你对此认同，希望立即开始使用语义化版本控制，你只需声明你的函式库正在使用它并遵循这些规则就可以了。请在你的 README 文件中保留此页连结，让别人也知道这些规则并从中受益。

FAQ

在 0.y.z 初始开发阶段，我该如何进行版本控制?

最简单的做法是以 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次版本号。

如何判断发布 1.0.0 版本的时机?

当你的软件被用于正式环境，它应该已经达到了 1.0.0 版。如果你已经有个稳定的 API 被使用者依赖，也会是 1.0.0 版。如果你很担心向下兼容的问题，也应该算是 1.0.0 版了。

这不会阻碍快速开发和迭代吗?

主版本号为零的时候就是为了做快速开发。如果你每天都在改变 API，那么你应该仍在主版本号为零的阶段（0.y.z），或是正在下个主版本的独立开发分支中。

对于公共 API，若即使是最小但不向下兼容的改变都需要产生新的主版本号，岂不是很快就达到 42.0.0 版?

这是开发的责任感和前瞻性的问题。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。升级所付出的代价可能是巨大的。要递增主版本号来发行不兼容的改版，意味着你必须为这些改变所带来的影响深思熟虑，并且评估所涉及的成本及效益比。

为整个公共 API 写文件太费事了!

为供他人使用的软件编写适当的文件，是你作为一名专业开发者应尽的职责。保持专案高效一个非常重要的部份是掌控软件的复杂度，如果没有人知道如何使用你的软件或不知道哪些函数的调用是可靠的，要掌控复杂度会是困难的。长远来看，使用语义化版本控制以及对于公共 API 有良好规范的坚持，可以让每个人及每件事都运行顺畅。

万一不小心把一个不兼容的改版当成了次版本号发行了该怎么办?

一旦发现自己破坏了语义化版本控制的规范，就要修正这个问题，并发行一个新的次版本号来更正这个问题并且恢复向下兼容。即使是这种情况，也不能去修改已发行的版本。可以的话，将有问题的版本号记录到文件中，告诉使用者问题所在，让他们能够意识到这是有问题的版本。

如果我更新了自己的依赖但没有改变公共 API 该怎么办?

由于没有影响到公共 API，这可以被认定是兼容的。若某个软件和你的套件有共同依赖，则它会有自己的依赖规范，作者也会告知可能的冲突。要判断改版是属于修订等级或是次版等级，是依据你更新的依赖关系是为了修复问题或是加入新功能。对于后者，我经常会预期伴随着更多的代码，这显然会是一个次版本号级别的递增。

如果我变更了公共 API 但无意中未遵循版本号的改动怎么办呢？（意即在修订等级的发布中，误将重大且不兼容的改变加到代码之中）

自行做最佳的判断。如果你有庞大的使用者群在依照公共 API 的意图而变更行为后会大受影响，那么最好做一次主版本的发布，即使严格来说这个修复仅是修订等级的发布。记住， 语义化的版本控制就是透过版本号的改变来传达意义。若这些改变对你的使用者是重要的，那就透过版本号来向他们说明。

我该如何处理即将弃用的功能?

弃用现存的功能是软件开发中的家常便饭，也通常是向前发展所必须的。当你弃用部份公共 API 时，你应该做两件事：（1）更新你的文件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次版本号发布。在新的主版本完全移除弃用功能前，至少要有一个次版本包含这个弃用信息，这样使用者才能平顺地转移到新版 API。

语义化版本对于版本的字串长度是否有限制呢?

没有，请自行做适当的判断。举例来说，长到 255 个字元的版本已过度夸张。再者，特定的系统对于字串长度可能会有他们自己的限制。

关于

语义化版本控制的规范是由 Gravatars 创办者兼 GitHub 共同创办者 Tom Preston-Werner 所建立。

如果您有任何建议，请到 GitHub 上提出您的问题。

授权