72 | 发布单元与版本管理

2020-01-10 许式伟

讲述：姚迪迈

时长 11:47　大小 10.81M

你好，我是七牛云许式伟。

前面我们在 “68 | 软件工程的宏观视角” 一讲中谈到：一个软件工程往往是生命周期以数年甚至数十年计的工程。对于传统工程，我们往往把一个工程同时也称之为项目，项目工程。但软件工程不同，虽然我们平常也有项目的概念，但软件工程并不是一个项目，而是无数个项目。每个项目只是软件工程中的一个里程碑（Milestone）。

这意味着软件工程终其完整的生命周期中，是在反复迭代与演进的。这种反复迭代演进的工程，要保证其质量实际上相当困难。

源代码版本管理

怎么确保软件工程的质量？

很容易想到的一个思路是，万一出问题了，就召回，换用老版本。

这便是版本管理的来由。当然，如果仅仅只是为了召回，只需要对软件的可执行程序进行版本管理就好了。但我们如果要进一步定位软件质量问题的原因，那就需要找到一个方法能够稳定再现它。

这意味着我们需要对软件的源代码也进行版本管理，并且它的版本与可执行程序的版本保持一一对应。

但实际上这事并没有那么简单。

从软件的架构设计可知，软件是分模块开发的，不同模块可能由不同团队开发，甚至有些模块是外部第三方团队开发。这意味着，从细粒度的视角来看，一个软件工程的生命周期中，包含着很多个彼此完全独立的子软件工程。这些子软件工程它们有自己独立的迭代周期，我们软件只是它们的 “客户”。

这种拥有独立的迭代周期的软件实体，我们称之为 “发布单元”。你可能直觉认为它就是模块，但是实际上两者有很大的不同。

对于一个发布单元，我们直观的一个感受是它有自己独立的源代码仓库（repo）。

发布单元的输出不一定是可执行程序，它有如下可能：

可执行程序，或某种虚拟机的字节码程序；

动态库（so/dylib/dll）；

某种虚拟机自己定义的动态库，比如 JVM 平台下的 jar 包；

静态库（.a 文件），它通常实际上是可执行程序的半成品，比较严谨来说的编译过程是先把每个模块编译成半成品，然后由链接器把各个模块组装成成品；

源代码本身，一些语言的价值主张是源代码发布，比如 Go 语言。

发布单元的输入，常规理解主要包含以下两部分的内容：

若干自己独立演进的模块，也就是源代码仓库（repo）托管的代码；

自己依赖的发布单元列表，这些外部的发布单元有自己独立的迭代周期。

源代码仓库管理系统，比如 svn、git 等等，一般只能管到第一部分。它让我们对自己独立演进的代码可以有很好的质量跟踪。

我们以 github 为例，它提供了以下源代码质量的管理手段。

其一，团队成员开发活动的独立性。每个人可以极低成本地建立一个开发分支（branch），一个开发分支做一个功能（feature），这个工作没有完成时，他的工作对所有其他人不可见，所以团队成员有很好的并行开发的能力，彼此完全独立。

其二，完善的代码质量检查机制。当一个团队成员完成他某项功能（feature）开发时，他可以提交一个功能合并请求（pull request），以求将代码合并进主代码库。但在此之前，我们需要对这项新功能的代码质量进行检查。常见的手段如下：

自动化运行单元测试案例（unit test）；

单元测试覆盖率检查（code coverage）；

静态代码质量检查（lint）；

人工的代码互审（code review）；

……

代码质量检查过程，需求显然比较易变。所以在这里 github 做了开放设计。我们再一次感受到了开闭原则的威力。

其三，完善的回滚机制（revert）。在代码已经合并到主代码库后，如果我们突然发现它有 Bug，这时候并不是落子无悔，而是可以自己对某次有 Bug 的 pull request 做回滚（revert），这样主干就可以得到去除了该功能后的一个新的发行版本。

对于第二部分，也就是发布单元的外部依赖管理，通常不同语言有自己的惯例。例如，Go 语言早期并没有官方的版本管理手段，所以导致有很多社区版本的实现方案。直到最新的 go mod 机制终于统一了这一纷争。

从基本原理来说，所有外部依赖管理无非要达到这样一个目标：指定我这个发布单元依赖的各个模块（嗯，这是通俗说法，其实是指依赖的发布单元）的建议版本是什么。

这样，我们理论上就可以稳定持续地通过源代码构建出相同能力的输出结果。

注意，这里有一个前提假设，是要求所有人都自觉遵循的：一个打好了版本号的发布单元是只读的，我们不能对其做任何改动。这句话的意思包括：

其一，我们不能修改发布单元自身包含的各个模块的的代码。这很容易理解，我们不展开。

其二，我们不能修改发布单元依赖的外部模块（同样地，其实指依赖的发布单元）的版本。比如我们依赖 opencv，把依赖的版本号从 v1.0 升级到 v2.0，这是不行的，这也是一次变更，需要修改我们的版本号。

如果有人破坏了版本的只读语义，就会导致所有依赖它的发布单元的版本只读语义也被破坏。这是我们需要极力去避免发生的事情。

从严谨意义来说，仅保证发布单元自身的源代码和依赖的外部模块只读，仍然不足以保证输出结果的确定性。为什么这么说，因为还有两个东西没有做到只读：

其一，操作系统内核。不同版本的操作系统内核行为不完全一致，它的一些动态库可能行为不完全一致，这些都可能会导致我们的软件行为有所不同。

其二，编译器。不同版本的编译器同样存在理论上与编译的结果行为上不一样的可能。

为什么没有把它们纳入到源代码版本管理的范畴管起来？这当然是因为操作系统和编译器大部分情况下质量是有所保证的，所以当软件在不同版本的操作系统下行为不一致时，这会被看做软件 Bug 记录下来，而不是修改操作系统。

软件发布的版本管理

但并不是在所有时刻，我们都能够相信操作系统和编译器。从源代码版本管理的角度，它的好处是软件构建（build）过程是一个相对封闭可预期的环境，这个环境我们甚至直接规定操作系统的种类和版本、编译器的版本，系统预装哪些软件等等。

但是软件发布过程却并非如此。

我们大家可能都接触过各种软件发布的管理工具，比如 apt、rpm、brew 等等。在这些管理工具的使用过程中，我们每个人或多或少都有过不少 “失败教训” 。并不是每一次软件安装过程都能够如愿。

这些软件发布的管理工具，背后有不少实际上基于的就是源代码的版本管理。但是为什么这个时候它会不 work 呢？因为用户之间系统环境的差异太大了。让每个软件的发布者都能够想到多样化的环境并加以适配，这是非常高的要求。

所以，软件安装有时会不成功，实在是在所难免。

怎么才能彻底解决这个问题？

答案是，容器化。

容器的镜像（image），不只是包含了软件发布的可执行程序本身，也完整包含了运行它的所有环境，包括依赖的动态库和运行时，甚至包括了它依赖的 “操作系统” 。这意味着容器的镜像（image）的版本管理，比之源代码的版本管理更进一步，实现完完全全的自描述，不再依赖任何外部环境。

这给我们线上服务的版本管理带来了巨大的便捷性。新版本的服务有缺陷？回滚到老版本即可。

只读设计的确定性

版本的只读设计，带来巨大的收益，这是因为版本是一个 “基线” ，对于这个基线，我们心理上对它的预期是确定性的。这种确定性非常重要。

在 “68 | 软件工程的宏观视角” 一讲中我们提到：

> 软件项目的管理期望达到确定性。但软件工程本身是快速变化的，是不确定的。这就是软件工程本身的矛盾。我们的目标是在大量的不确定性中找到确定性，这其实就是软件工程最核心的点。

只读设计提升了软件工程的确定性，所以只读思想被广泛运用。前面我们说开闭原则背后的架构治理哲学，也是模块，或者说软件实体，其业务范畴只读。在业务只读，接口稳定的预期下，模块与模块之间就可以自由组合，构建越来越复杂的系统。

往小里说，我们开发的时候，有时候会倾向于变量只读，以提高内心对确定性的预期。我并没有去用严谨的方式实证过变量只读的收益究竟有多大，但它的确成为了很重要的一种编程流派，即函数式编程。

函数式编程从编程范式来说比较小众，但是其只读思想被广泛借鉴。

这里面最典型的就是大数据领域的 Spark。Spark 的核心是建立在统一的抽象弹性分布式数据集（Resiliennt Distributed Datasets，RDD）之上。

而 RDD 的核心思想正是只读。对一个只读的 RDD 施加一个变换（transform），即得到另一个 RDD，这不就是函数式编程么？但这种只读设计，让我们的分布式运算在重试、延迟计算、缓存等过程都变得极其简单。

版本的兼容问题

版本管理的最后一个问题是兼容性。让一个模块依赖另一个模块（严谨来说是发布单元）的特定版本，这解决了版本的确定性问题。

但是，在某个特定的时刻，我们总是会希望将依赖的模块升级到新版本。无论是基于我们需要使用该模块的新功能，又或者是为了修复的 Bug，或者纯粹是心理上想要更好的东西。

更换到新版本多多少少冒了一些风险。这里面最大风险是所依赖的模块完成了一次重构。

为什么依赖模块的重构会给我们的系统带来未知风险？这其中的原因就在于版本兼容的难度。

兼容一个模块的主体功能并不复杂，既然我们重构了，这部分肯定是得到了解决。但兼容的难度全在细节上。错误码、低频的分支行为等等，这些都需要兼容。

如果这种分支兼容太麻烦，我们干脆就放弃兼容，连软件实体（如函数）的名字都改了。这倒是干脆，客户升级版本后一看，编译不过了，老老实实用新的接口进行重写，重新测试。

但有时候我们无法放弃兼容。这发生在我们在做一个互联网服务时。一旦我们发布了一个 api，它就很难收回，因为使用这个 api 的客户端可能有很多。如果我们放弃这个 api 就意味着我们放弃了很多用户，这是不可接受的。

为了应对这个问题，比较常见的做法是为所有 api 引入版本号，如 “/v2/foo/bar”。当我们对 api 发生不兼容的修改时，就升级版本号，比如 “/v3/foo/bar”。

这样做有一个额外的好处。如果我们对某个复杂模块进行了全局重构，并且兼容老版本的行为细节非常困难时，我们可以直接升级所有 api 的版本号。这样在线上我们可以保留两个版本的服务同时存在。这通过前面放 nginx 作为 api 分派的网关来做到。

这样两个版本服务并行，就不需要重构时做太细节的行为兼容。但应当注意，这也是不得已的办法，如果能够兼容，还是鼓励尽可能去兼容。毕竟客户端在升级版本之后，不兼容的地方越多，修改的心智负担就越大。

结语

今天我们聊的是怎么做版本管理。一个复杂的软件，总可以被分割为若干个独立迭代的发布单元，以便分而治之。发布单元的切割不宜过细，应该以一个小团队负责起来比较舒服为宜，不太小但也不太大。

版本的只读设计提高了系统的确定性预期，这是非常非常好的收益。但我们也应注意版本兼容上带来的坑。

如果你对今天的内容有什么思考与解读，欢迎给我留言，我们一起讨论。下一讲我们谈谈“软件质量管理：单元测试、持续构建与发布”。

如果你觉得有所收获，也欢迎把文章分享给你的朋友。感谢你的收听，我们下期再见。

上一篇　71 | 如何阅读别人的代码?

精选留言 (5)

写留言

Aaron Cheung
2020-01-12

补打卡 很受益的文章

展开

靠人品去赢
2020-01-10

其实容器化很方便，但是这种serverless带来的就是如何监控等一些新问题也挺头疼的。

Geek007
2020-01-10

许老师思维缜密，很多文章读起来仿佛是一个大的switch case，基本所有的条件都考虑到

了，同时又能够把握主次。非常佩服，是我的榜样。
版本管理看似简单但其实复杂。有两个问题：
1. 对go mod了解不深，想了解go mod 对外部依赖的管理的层次，比如发布单元依赖外部包A，但其实外部包A又依赖外部包B, B 又依赖C...go mod会把这个依赖的chain: A->...
展开

作者回复: 1、支持的
2、在Go mod中，就算指定的是latest，对于具体的一个版本，也是依赖于具体版本，也不是latest。
3、两者都有。趋势是后者，直接开源版本就是内部版本。

霜花香似海
2020-01-10

多模块开发，需要做好模块边界上下文问题。只要是架构师确定了各个边界之间的上下文问题，也就统一了版本兼容，或者说统一了各个模块之间的标准了。不晓得这么说对不对


leslie
2020-01-10

多人协作的标准化其实是引发这个问题的原因：当前就碰到。1个项目开发工期偏紧，直接就多个团队同时参与，各自团队内部的开发标准又不一样，开发完成进入测试改错的环节时就发现一堆标准不一致的问题-直接导致部分代码重写。
如何在发布单元和版本控制中解决当下问题：这是将来相关项目时将会碰到的问题。多个小团队一起迭代发布就明显碰到了版本管理的问题。
展开