

10 | 配置管理：最容易被忽视的DevOps工程实践基础

2019-11-02 石雪峰

DevOps实战笔记

[进入课程 >](#)



讲述：石雪峰

时长 17:06 大小 15.67M



你好，我是石雪峰。从今天开始，专栏正式进入了工程实践的部分。在 DevOps 的体系中，工程实践所占的比重非常大，而且和我们的日常工作息息相关。正因为如此，DevOps 包含了大量的工程实践，很多我们都耳熟能详，比如持续集成、自动化测试、自动化部署等等，这些基本上是实践 DevOps 的必选项。

可是，还有一些实践常常被人们所忽视，但这并不代表它们已经被淘汰或者是不那么重要了。恰恰相反，它们同样是 DevOps 能够发挥价值的根基，配置管理（Configuration Management）就是其中之一。它的理念在软件开发过程中无处不在，可以说是整个 DevOps 工程实践的基础。所以今天我们就来聊一聊配置管理。

说了这么多，那软件配置管理到底是个啥呢？

熟悉运维的同学可能会说，不就是类似 Ansible、Saltstack 的环境配置管理工具吗？还有人会说，CMDB 配置管理数据库也是配置管理吧？这些说法都没错。配置管理这个概念在软件开发领域应用得非常普遍，几乎可以说无处不在，但是刚刚提到的这些概念，都是细分领域内的局部定义。

我今天要讲到的配置管理，是一个宏观的概念，是站在软件交付全生命周期的视角，对整个开发过程进行规范管理，控制变更过程，让协作更加顺畅，确保整个交付过程的完整、一致和可追溯。

看到这里，我估计你可能已经晕掉了。的确，配置管理的理论体系非常庞大。但是没关系，你只需要把四个核心理念记在心中就足够了。这四个理念分别是：**版本变更标准化，将一切纳入版本控制，全流程可追溯和单一可信数据源。**

1. 版本变更标准化

版本控制是配置管理中的一个非常核心的概念，而对于软件来说，最核心的资产就是源代码。现在很多公司都在使用类似 Git、SVN 之类的工具管理源代码，这些工具其实都是版本控制系统。版本描述了软件交付产物的状态，可以说，从第一行软件代码写下开始，版本就已经存在了。

现代软件开发越来越复杂，往往需要多人协作，所以，如何管理每个开发者的版本，并把它们有效地集成到一起，就成了一个难题。实际上，版本控制系统就是为了解决这个问题的。试想一下，如果没有这么一套系统的话，所有代码都在本地，不要说其他人了，就连自己都会搞不清楚哪个是最新代码。那么，当所有人的代码集成到一起的时候，那该是多么混乱啊！

不仅如此，如果线上发生了严重问题，也找不到对应的历史版本，只能直接把最新的代码发布上去，简直就是灾难。

配置管理中的另一个核心概念是变更。我们对软件做的任何改变都可以称之为一次变更，比如一个需求，一行代码，甚至是一个环境配置。**版本来源于变更。**对于变更而言，核心就是要记录：**谁，在什么时间，做了什么改动，具体改了哪些内容，又是谁批准的。**

这样看来，好像也没什么复杂的，因为现代版本控制系统基本都具备记录变更的功能。那么，是不是只要使用了版本控制系统，就做到变更管理了呢？

的确，版本控制系统的出现，大大简化了管理变更的成本，至少是不用人工记录了。但是，从另一方面来看，用好版本控制系统也需要有一套规则和行为规范。

比如，版本控制系统需要打通公司的统一认证系统，也就是任何人想要访问版本控制系统，都需要经过公司统一登录的认证。同时，在使用 Git 的时候，你需要正确配置本地信息，尤其是用户名和邮箱信息，这样才能在提交时生成完整的用户信息。另外，系统本身也需要增加相关的校验机制，避免由于员工配置错误导致无效信息被提交入库。

改动说明一般就是版本控制系统的提交记录，一个完整的提交记录应该至少包括以下几个方面的内容：

提交概要信息：简明扼要地用一句话说明这个改动实现了哪些功能，修复了哪些问题；

提交详细信息：详细说明改动的细节和改动方式，是否有潜在的风险和遗留问题等；

提交关联需求：是哪次变更导致的这次提交修改，还需要添加上游系统编号以关联提交和原始变更。

这些改动应该遵循一种标准化的格式，并且有相关的格式说明和书写方式，比如有哪些关键字，每一行的长度，变更编号的区隔是使用逗号、空格还是分号等等。如果按照这个标准来书写每次的变更记录，其实成本还是很高的，更不要说使用英文来书写的话，英文的表达方式和内容展现形式又是一个难题。

我跟你分享一个极品的提交注释，你可以参考一下。

```
switch to Flask-XML-RPC dependency
```

```
CR: PBX-2222
```

```
The Flask-XML-RPC-Re fork has Python 3 support, but it has a couple  
other problems.
```

```
1. test suite does not pass
```

```
2. latest code is not tagged
```

```
3. uncompiled source code is not distributed via PyPI
```

```
The Flask-XML-RPC module is essentially dead upstream, but it is  
packaged in EPEL 7 and Fedora. This module will get us far enough to
```

the

point that we can complete phase one for this project.

When we care about Python 3, we can drop XML-RPC entirely and get the service consumers to switch to a REST API instead.

(Note, with this change, the Travis CI tests will fail for Python 3.

The

solution is to drop XML-RPC support.)

这时，肯定有人会问，花这么大力气做这个事情，会不会有点得不偿失呢？从局部来看，的确如此。但是，换个角度想，当其他人看到你的改动，或者是评审你的代码的时候，如果通过提交记录就能清晰地了解你的意图，而不是一脸蒙地把你叫过来，让你再讲一遍，这样节约的时间比当时你书写提交记录的时间要多得多。

所以你看，一套标准化的规则和行为习惯，可以降低协作过程中的沟通成本，一次性把事情做对，这也是标准和规范的重要意义。

当然，如果标准化流程要完全依靠人的自觉性来保障，那就太不靠谱了。毕竟，人总是容易犯错的，会影响到标准的执行效果。所以，当团队内部经过不断磨合，逐步形成一套规范之后，最好还是用自动化的手段保障流程的标准化。

这样做的好处有两点：一方面，可以降低人为因素的影响，如果你不按标准来，就会寸步难行，也减少了人为钻空子的可能性。比如，有时候因为懒，每次提交都写同样一个需求变更号，这样的确满足了标准化的要求，但是却产生了大量无效数据。这时候，你就可以适当增加一些校验机制，比如只允许添加你名下的变更，或者是只允许开放状态的变更号等等。另一方面，在标准化之后，很多重复性的工作就可以自动化完成，标准化的信息也方便计算机分析提取，这样就可以提升流程的流转效率。

可以说，标准化是自动化的前提，自动化又是 DevOps 最核心的实践。这样看来，说配置管理是 DevOps 工程实践的基础就一点不为过了吧。

2. 将一切纳入版本控制

如果说，今天这一讲的内容，你只需要记住一句话，那就是将一切纳入版本控制，这是配置管理的金科玉律。你可能会问，需要将什么样的内容纳入版本控制呢？我会毫不犹豫地回答

你：“一切都需要！”比如软件源代码、配置文件、测试编译脚本、流水线配置、环境配置、数据库变更等等，你能想到的一切，皆有版本，皆要被纳入管控。

这是因为，软件本身就是一个复杂的集合体，任何变更都可能带来问题，所以，全程版本控制赋予了我们全流程追溯的能力，并且可以快速回退到某个时间点的版本状态，这对于定位和修复问题是非常重要的。

之前，我就遇到过一个问题。一个 iOS 应用发灰度版本的时候一切正常，但是正式版本就遇到了无法下载的情况。当时因为临近上线，为了查这个问题，可以说是全员上阵，团队甚至开始互相抱怨，研发说代码没有变化，所以是运维的问题；运维说环境没动过，所以是研发的问题。结果到最后才发现，这是由于一个工具版本升级，某个参数的默认值从“关闭”变成了“打开”导致的。

所以你看，如果对所有内容都纳入版本控制，快速对比两个版本，列出差异点，那么，解决这种问题也就是分分钟的事情，大不了就把所有改动都还原回去。

纳入版本控制的价值不止如此。实际上，很多 DevOps 实践都是基于版本控制来实现的，比如，环境管理方面推荐采用基础设施即代码的方式管理环境，也就是说把用代码化的方式描述复杂的环境配置，同时把它纳入版本控制系统中。这样一来，任何环境变更都可以像提交代码一样来完成，不仅变更的内容一目了然，还可以很轻松地实现自动化。**把原本复杂的事情简单化，每一个人都可以完成环境变更。**

这样一来，开发和运维之间的鸿沟就被逐渐抹平了，DevOps 的真谛也是如此。所以，现在行业内流行的“什么什么即代码”，其背后的核心都是版本控制。

不过，这里我需要澄清一下，纳入版本控制并不等同于把所有内容都放到 Git 中管理。有些时候，我们很容易把能力和工具混为一谈。Git 只是一种流行的版本控制系统而已，而这里强调的其实是一种能力，工具只是能力的载体。比如，Git 本身不擅长管理大文件，那么可以把这些大文件放到 Artifactory 或者其他自建平台上进行管理。

对自建系统来说，实现版本控制的方式有很多种，比如，可以针对每次变更，插入一组新的数据，或者直接复用 Git 这种比较成熟的工具作为后台。唯一不变的要求就是，无论使用什么样的系统和工具，都需要把版本控制的能力考虑进去。

另外，在实践将一切纳入版本控制的时候，你可以参考一条小原则。如果你不确定是否需要纳入版本控制，有一个简单的判断方法就是：**如果这个产物可以通过其他产物来重现，那么就可以作为制品管理，而无需纳入版本控制。**

举个例子，软件包可以通过源代码和工具重新打包生成，那么，代码、工具和打包环境就需要纳入管控，而生成的软件包可以作为制品；软件的测试报告如果可以通过测试管理平台重新自动化生成，那么同样可以将其视为制品，但前提是，测试管理平台可以针对每一个版本重新生成测试报告。

3. 全流程可追溯

对传统行业来说，全流程可追溯的能力从来不是可选项，而是必选项。像航空航天、企业制造、金融行业等，对变更的管控都是非常严谨的，一旦出现问题，就要追溯当时的全部数据，像软件源代码、测试报告、运行环境等等。如果由于缺乏管理，难以提供证据证明基于当时的客观情况已经做了充分的验证，就会面临巨额的罚款和赔偿，这可不是闹着玩的事情。像最近流行的区块链技术，除了发币以外，最典型的场景也是全流程可追溯。所以说，**技术可以日新月异，但很多理念都是长久不变的。**

对于配置管理来说，除了追溯能力以外，还有一个重要的价值，就是记录关联和依赖关系。怎么理解这句话呢？我先提个问题，在你的公司里面，针对任意一个需求，你们是否能够快速识别出它所关联的代码、版本、测试案例、上线记录、缺陷信息、用户反馈信息和上线监控数据呢？对于任意一个应用，是否可以识别出它所依赖的环境，中间件，上下游存在调用关系的系统、服务和数据呢？

如果你的回答是“yes”，那么恭喜你，你们公司做得非常好。不过，绝大多数公司都是无法做到这一点的。因为这不仅需要系统与系统之间的关联打通、数据联动，也涉及到一整套完整的管理机制。

DevOps 非常强调价值导向，强调团队内部共享目标，这个目标其实就是业务目标。但实际情况是，业务所关注的维度，和开发、测试、运维所关注的维度都各不相同。业务关心的是提出的需求有没有上线，而开发关心的是这个需求的代码有没有集成，运维关心的是包含这个代码的版本是否上线。所以，如果不能把这些信息串联打通，就没有真正做到全流程可追溯。

关于这个问题，我给你的建议是**把握源头，建立主线**。所谓源头，对于软件开发而言，最原始的就是需求，**所有的变更都来源于需求**。所以，首先要统一管理需求，无论是开发需求、测试需求还是运维需求。

接下来，要以需求作为抓手，去关联下游环节，打通数据，这需要系统能力的支持，也需要规则的支持。比如，每次变更都要强制关联需求编号，针对不同的需求等级定义差异化流程，这样既可以减少无意义的审批环节，给予团队一定的灵活性，也达到了全流程管控的目标。这是一个比较漫长的过程，但不积跬步，无以至千里，DevOps 也需要一步一个脚印地建设才行。

4. 单一可信数据源

最后，我想单独谈谈单一可信数据源。很多人不理解这是什么东西，我举个例子你就明白了。

有一个网络热词叫作“官宣”，也就是官方宣布的意思。一般情况下，官宣的信息都是板上钉钉的，可信度非常高。可问题是，如果有多个官宣的渠道，信息还都不一样，你怎么知道要相信哪一个呢？这就是单一可信数据源的意义。

试想一下，我们花了很大力气来建设版本控制的能力，但如果数据源本身不可靠，缺乏统一管控，那岂不是白忙一场吗？所以，对于软件开发来说，必须要有统一的管控：

对于代码来说，要有统一的版本控制系统，不能代码满天飞；

对于版本来说，要有统一的渠道，不能让人随便本地打个包就传到线上去了；

对于开发依赖的组件来说，要有统一的源头，不能让来路不明的组件直接集成到系统中。这不仅对于安全管控来说至关重要，对于企业内部的信息一致性也是不可或缺的。

同时，单一可信数据源也要能覆盖企业内部元数据的管控。比如，企业内部经常出现这种情况，同样是应用，在 A 部门的系统中叫作 123，在 B 部门的系统中叫作 ABC，在打通两边平台的时候，这就相当于“鸡同鸭讲”，完全对不上。再比如，信息安全团队维护了一套应用列表，但实际上，在业务系统中，很多应用都已经下线且不再维护了，这样一来，不仅会造成资源浪费，还伴随着非常大的安全风险。

很多时候，类似的这些问题都是因为缺乏统一的顶层规划和设计导致的，这一点，在建立配置管理能力的时候请你格外关注一下。

总结

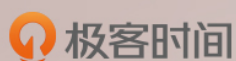
今天我给你介绍了 DevOps 工程实践的基础配置管理，以及配置管理的四大理念，分别是版本变更标准化、将一切纳入版本控制、全流程可追溯和单一可信数据源，希望能帮你掌握配置管理的全局概念。

虽然配置管理看起来并不起眼，但是就像那句经典的话一样：“岁月静好，是因为有人替你负重前行。”对于任何一家企业来说，信息过载都是常态，而**配置管理的最大价值正是将信息序列化**，对信息进行有效的整理、归类、记录和关联。而软件开发标准和有序，也是协同效率提升的源头，所以，配置管理的重要性再怎么强调都不为过。

思考题

你在企业中遇到过哪些配置管理方面的难题呢？你们的配置管理体系又是如何建立的呢？你遇到过因为缺乏单一可信数据源而导致“鸡同鸭讲”的有趣故事吗？

欢迎在留言区写下你的思考和答案，我们一起讨论，共同学习进步。如果你觉得这篇文章对你有帮助，欢迎你把文章分享给你的朋友。




DevOps 实战笔记

精要 30 计，让 DevOps 快速落地

石雪峰

京东商城工程效率专家



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

上一篇 09 | 精益看板（下）：精益驱动的敏捷开发方法

精选留言 (5)

写留言



阿硕

2019-11-03

石老师，您好，请教下什么样的角色适合担当起配置管理呢？考核是统一管理还是各自独立呢？

作者回复: 你好，在以往公司内部都有专职的配置管理员，不过近些年来这个岗位本身有些模糊，在互联网公司随着流程和工具平台的成熟度提升，慢慢都会转到工程效率，版本团队来兼任，这个还是要看你们公司的业务形态，一般组织级会统一建设配置管理能力和规则。



albertz

2019-11-02

提交记录那部分让我想到这一点：Code Review的起始点应该是commit message，从这里可以看出一个开发人员对自己的工作成果是否有信心和责任心，以及表达能力。也是一种综合素质的体现。我们就是做软件配置管理的，每天看到的无数提交记录，真的是千奇百怪，良莠不齐，有的干脆是不知所云，或者就是写一串无意义的字符。现在我们通过工具强制填写JIRA号和任务信息，算是略有改善。所以我想说，对自己的代码负责，想让...
展开

作者回复: 没错，每个人写下的每一行代码和提交注释都是自身credit的体现，良好的习惯也要通过流程来培养，为了加速这个习惯的养成，适当的规则建立也是很有必要的。



Robert小七

2019-11-02

老师是否有落地的方案呢？这篇专栏都是讲的概念，在devops研发运营一体化中都有相应的介绍！但是对于如何落地，我觉得这个才是重点

展开

作者回复: 你好, 每个企业面临的实际问题都不相同, 很难给出一个通用的落地方案, 但从配置管理的角度来说, 最重要的就是识别和管理配置项, 也就是将一切纳入版本控制, 并且是要遵循规则纳入管理的, 我的建议是你把软件交付过程中涉及的这些配置项通盘梳理一下, 看看是否都满足这个要求。纳入管理之后, 就要建立流程方面的上下游关联和联动, 比如开始可以从需求和代码的关联做起, 再逐步扩展到其他配置项内容。当然, 也欢迎你提出你的实际问题来, 这样可能会更加聚焦一些, 谢谢。

1



leslie

2019-11-02

其实“版本变更标准化”和“全流程可追溯”是一个典型问题: 都是靠人的大致记忆或者去搜, 很多会去抱怨干嘛写那么多/详细的注释; 领导不强行推或者说各自一个标准, 其实这是后面出现问题排错的一个巨坑。

“单一可信数据源”其实还好控制: 测试测完了才能放上去, 让软件测试去把关; 彻底没有问题的才能放上去, 这样就能避免不少问题。开发不写注释或者注释的问题确实很难...

展开

作者回复: 可能随着企业规模的扩大, 单一可信数据源的问题会越来越明显吧, 至少在我所在的公司, 这就是一个非常大的通病, 各个系统之间难以有一套标准的语言来实现打通。

1



Brian

2019-11-02

四大理念里面, 对于我们公司而言, 最难的是第一条—版本变更标准化。

问题在于有了标准之后刚开始一段时间大家按部就班得按标准化流程去执行, 后期处于疲态后完全忽视标准的存在, 仅仅将成果物做了版本管理而已。主要原因我觉得是领导也睁一只眼闭一只眼

展开

作者回复: 是的, 如果标准不能固化在工具平台之中, 的确容易流于形式, 标准能够被绕过, 说明还是没有重视这块的价值吧, 这么细节的内容, 可能领导也的确关注不到。

1



