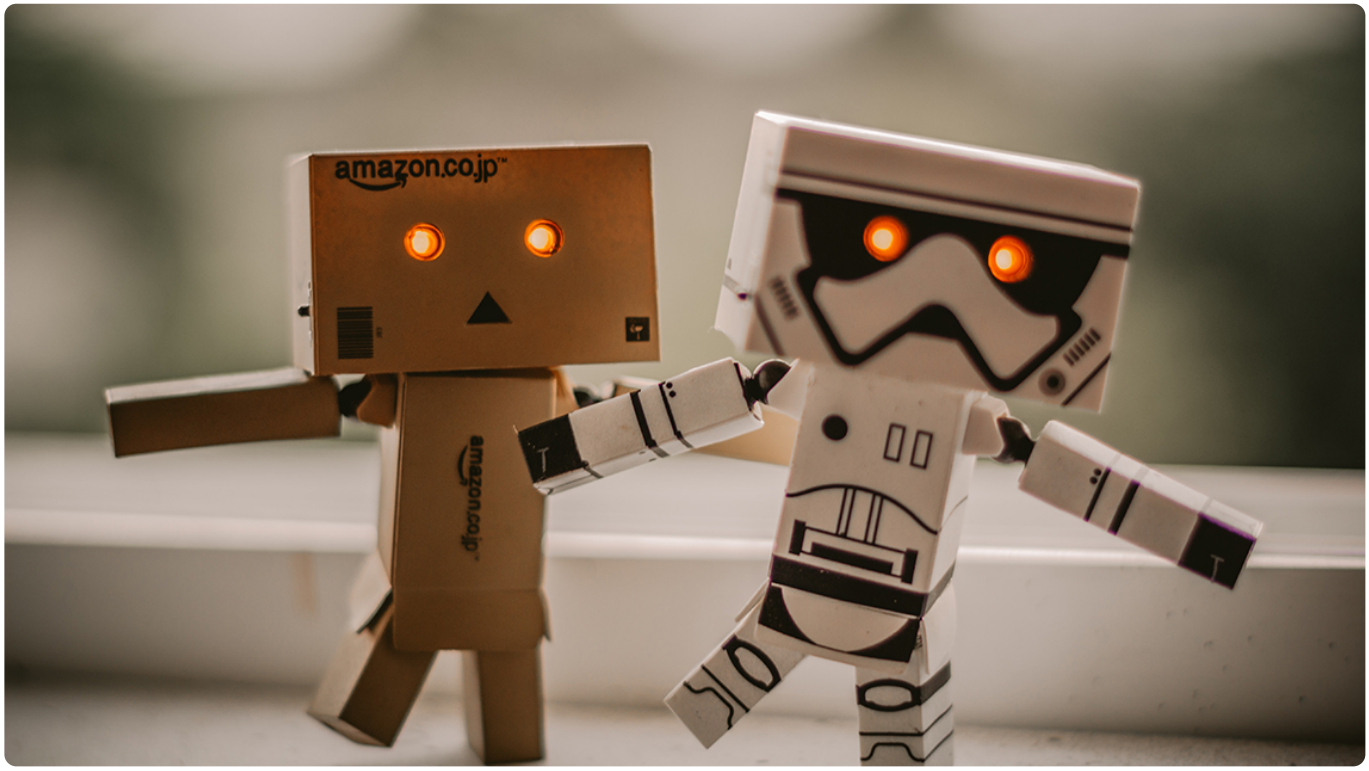
# 28 | Ops三部曲之一：配置管理

2019-11-13 四火

全栈工程师修炼指南 进入课程 >



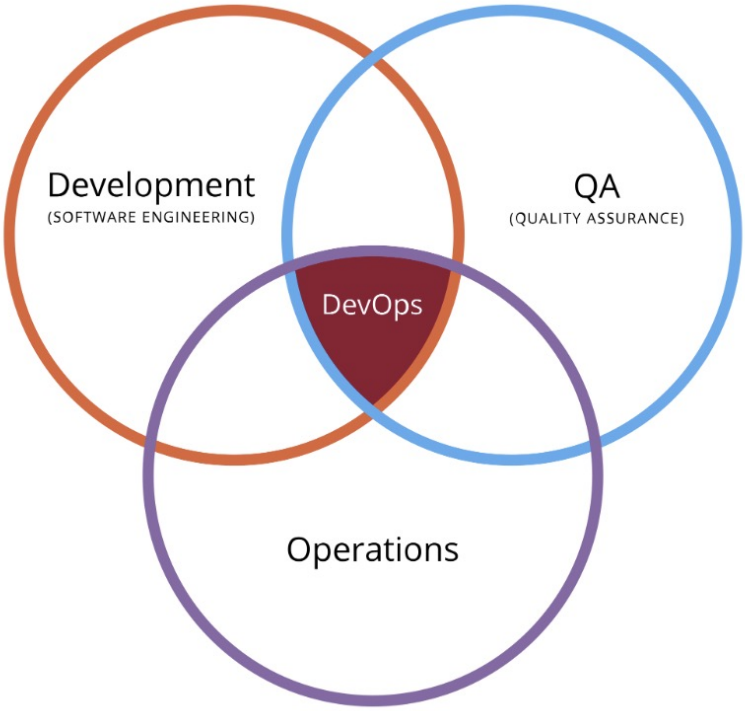

**讲述：四火**
时长 20:50 大小 14.32M

你好，我是四火。

欢迎进入第五章：寻找最佳实践。本章我们会讲到 Ops，由于 Ops 的范围实在太广了，因此从今天开始，接连三讲，我们会讨论 Ops 的三个常见话题，今天要谈的就是其中的第一个——配置管理。

我们总在谈论 Ops，所谓 Ops，指的就是 Operations，如果你专门去搜索，你恐怕也很难找到一个准确的定义。在中文翻译上看，“运维”这个词用得比较多，但是我认为 Ops 的含义明显更为广泛。我的理解是，和实际的软件开发和质量保证相对应的，环境搭建、配置管理、自动化流程、部署发布，等等的实践活动，无论线上还是线下，无论它们是在开发测试环节还是产品发布环节，都可以算作 Ops。

当然，**我们并不需要执着去找出 Ops 和开发、测试的分界线，它们实际是互有重叠的**，我觉得维基百科上的这张图就很好地展示了这一点。




我想很多程序员不喜欢 Ops，也不愿意谈 Ops，这都可以理解，毕竟，和开发本身比起来，它没有那么多的创造性，有时甚至属于“重要但无趣”的工作。但是，我必须要强调的是，不只对于全栈工程师而言，Ops 能力对于每一个通常意义上的软件工程师来说，都是必须要锻炼的重要方面。

需要明确的是，我今天要讲的配置管理，我认为就是 Ops 的一个重要部分，它关注的是软件代码以各种形式，在项目的各个阶段存在的配置。几乎每个做运维的人都要经常去修改大量的线上配置，无论是来自操作系统还是应用本身。这个话题很少有人谈及，但却存在于每个程序员的日常工作中。

## 常见的配置方式

在实际的项目开发过程中，当我们讲到“配置”的时候，其实隐含了许多不同的方式，这些方式有时候看起来差不多，实际却有着最适合的场景。

### 1. 源代码中的常量

代码常量是一种最典型的配置方式，它的特点是定义和使用方便，且只对开发人员友好。每次变更的时候，都需要改动代码，经过一定的测试和 Code Review，之后还要通过指定流

程的部署，才能发布到产品线上。

通常来说，项目中常量类的定义有着明确的规约，比如这样的示例代码：

```
复制代码
1 public final class ProductConstants {
2     private ProductConstants() {}
3
4     public static int MAX_NUMBER = 9999;
5     public static String DEFAULT_CODE = "123";
6 }
```

你看，上面的常量类包含的要点有：

> 类名直白且具体，这个 Product 修饰很重要，避免了过于一般化的常量类定义，从而限制了它的职责范围；而这个 Constants 则说明了类的属性，一眼就知道它是干什么的。
>
> 被定义为无法实例化的（即将构造器私有化），无法被继承的（即使用 final 修饰）。
>
> 常量都使用 public static 修饰，这样就可以直接访问，而不需要实例化这个类。

值得一提的是，在我参与过的项目中，有的程序员为了强制编码的时候只能定义 static 属性来表示常量，会使用 interface 这样的方式，即使用接口来存放常量。效果自然是能达到的，但是这个方法有些投机取巧（毕竟和 interface 关键字所应该表示的含义有明显偏差了），我认为这是一种反模式。

## 2. 代码中的配置文件

代码中的配置文件也很常见，它们从 Java、Python 这样的编程语言中脱离出来，但是依然作为源代码的一部分而存在。那为什么要这样做呢？

简单来说，就是为了“解耦”。

而就是这个“解耦”，带来了诸多好处。例如最小职责，单一职责，即配置文件做且只能做配置；再例如可以将同一类的资源放在更合适的地方统一管理了。

举例来说：

```
MessagesBundle_en_US.properties
MessagesBundle_fr_CN.properties
MessagesBundle_de_DE.properties
```

你看，这样的 i18n（Internationalization）的特定语言的资源文件，就从编程语言的代码中独立出来了，更容易统一管理和维护。

这种把相关代码或配置单独拿出来的方式，不知道你能否联想到我们在 [第 10 讲] 中介绍的 MVC 控制器各个子功能的配置方式，当时不但讨论了横向和纵向两种实现形式，还讨论了各自的优劣，如果忘记了的话，你可以阅读回顾一下。比如 URL 映射，既可以通过注解的方式，和实际的 Controller 代码放在一起；也可以通过 XML 配置的方式，从 Java 代码中拿出去。

### 3. 环境配置文件和环境变量

接着我们来讲讲环境下的配置。这里的“环境”（Environment），指的是代码部署的环境，可以是用于开发的笔记本电脑，也可以是生产线上的机器。对于不同的环境，虽说代码是同一份，但是可以通过不同的环境配置文件和环境变量，让代码执行不同的逻辑。

举一个环境配置文件的例子，例如环境下有这样一个文件，定义了这个环境所属的“地区”，而通过这个地区，可以从代码中寻找到相应正确的配置文件来读取配置：

```
/etc/region
```

在某些系统中，这样的配置可以通过一个 web 工具指定，并通过一定的部署工具将其同步到目标环境中去。

还有一种类似的方式是环境变量，环境变量和环境配置文件，有时允许同时存在，但是有一定的优先级顺序。比如说，如果环境变量使用 REGION 也指定了上述文件中的 region，发生了冲突，这就要根据优先级的关系来决定使用哪一个配置，通常来说，环境变量比环境配置文件优先级更高。

那为什么要允许配置不同方式的覆写呢？这是为了给应用的部署赋予灵活性，举例来说，/etc/region 已经确定了值为 EU（欧洲），那么如果需要在该物理机器上部署两份环境，第二份就可以通过环境变量 REGION 强制指定为 NA（北美）。

### 4. 运行参数

运行参数，可以说是最为具体的一种配置方式，真正做到了应用级别，即一个应用可以配置一个参数。这种情况其实对于开发人员来说非常熟悉，比如在项目中应用 log4j2 这个日志工具时，启动应用的 java 命令上，增加如下参数：

```
-Dlog4j.configurationFile=xxx/log4j2.xml
```

复制代码

通过这样的运行参数指定的方式，指定 log4j2 配置文件的位置。

当然，也可以通过规约优于配置的方式，不显式指定位置，而在 classpath 中放置一个名为 log4j2.component.properties 的文件，里面指定类似的配置，而 log4j2 在系统启动的时候可以自动加载。

### 5. 配置管理服务

常见的配置方式中，我们最后来讲一下配置管理服务。尤其对一个较大的系统来说，配置管理是从整个系统的层面上抽取并统一管理配置项的方式。通常来说，这样的配置管理系统会被包装成一个服务，当然，也有少数是单纯放到数据库的某张表里，不过这种数据库访问层面的耦合通常并不推荐。

一旦配置管理成为了独立的服务，就说明这个系统已经复杂到一定程度了，通常也意味着这个服务的用户，不再只是开发人员了，往往还有运维人员，甚至是一些非技术的管理人员。

## 配置的层级关系

资源文件，本质上也算是代码的一部分，**通过合理的设计，可以让资源文件具备编程语言代码一般的继承关系。**比如这样的配置文件组织结构：

```
conf/rules.conf
conf/CN/rules.conf
conf/CN/Zhejiang/rules.conf
conf/US/rules.conf
```

conf 目录下，rules.conf 文件就像是基类，存放了默认的规则配置；下面的 CN 目录下的 rules.conf 则存放了中国区的增量配置，就像是子类，里面的配置项优先级高于“基类”的配置，起到一个有选择性地覆写的作用；而再下面的 Zhejiang 目录下的 rulels.conf 则表示浙江省的规则配置，优先级更高。

在这种方式下，配置代码不但清晰易懂，而且减少了重复，易于维护。

## 规约优于配置

不知道你是否还记得我们在 [第 10 讲] 中介绍过的“终极偷懒大法”——规约优于配置（CoC，Convention over Configuration）。在这种方式下，系统和配置的用户会建立“隐性的契约”，通过遵从一定的规则，比如命名规则，达到自动应用配置的目的。

当时我们举了一个 Spring 的 ControllerClassNameHandlerMapping 的例子来说明，现在我们来举另外一个 Grails 的例子。这里使用 Grails 举例，是因为 Grails 是我接触过的应用 CoC 原则应用得最好的 Web 应用框架了，使用它搭建起一个 Web 应用极其简洁。如果你使用过 Spring Boot 并对它印象还不错的话，你可以尝试这个将快速和简洁履行得更为彻底的 Grails。

```
class BooksController {
    def index() { ... }
}
```

你看，如此简单的控制器定义，就可以自动把路径为 /books 的请求映射到 index 方法上，通过规约将 BooksController 映射到 /books 上，而 index 又代表了默认的 GET 方法的访问。

## 配置模板

对于某些复杂或灵活的软件系统来说，配置会变成实际上的 DSL（Domain Specific Language），复杂程度可以不亚于一门编程语言写的代码。于是，有一种常见的帮助使用者理解和修改配置的方法就出现了，它就是创建配置模板，写好足量的配置默认值和配置说明，这样使用者就可以复制一份模板，并在其之上按需修改，比如 Nginx 的配置模板。

Nginx 是一个高性能的反向代理服务器，反向代理可以挡在 Web 服务器前面响应用户的请求，根据不同的规则来处理请求，比如认证、限流、映射到其它路径，访问特定的服务器等等，将这些复杂的访问逻辑和服务器节点都隐藏起来，给用户提供一个单一的 IP 地址。

下面我们来动动手，亲自配置一下 Nginx 并让它工作起来。

在 Mac 上你可以使用 brew 来安装 Nginx，或者去官网上找一个适合的版本下载安装：

```
1 brew install nginx
```

复制代码

在安装成功以后，你应该能看到命令行输出了默认 Nginx 配置文件的位置，例如 /usr/local/etc/nginx/nginx.conf。现在打开看一下，你会发现它本质上就是一个配置模板，有些配置选项为了便于程序员理解和使用，使用注释包围起来了。在同一路径下，还有一个 nginx.conf.default 作为备份和参考。

还记得在我们在 [第 12 讲] 中自己动手配置的过滤器吗？确认一下它还能正确运行：

```
1 catalina run
```

复制代码

接着访问 http://localhost:8080/books?category=art，你应该能看到类似如下的输出：

```
1 Category name: art, date: 2019-10-4 Count: 1
```

复制代码

好，如果能正确运行，我们继续往下；否则，请回头看看那一讲是怎样把这一个过滤器配置起来的。现在我们根据前面提示的 Nginx 配置文件的路径来稍加修改，比如，把 http 部分的 #access_log 开始的一行修改为类似如下路径：

```
1 access_log  /logs/nginx_access.log;
```

复制代码

接着打开一个新命令行窗口运行：

```
1 tail -f /logs/nginx_access.log
```

复制代码

这样就可以监视 Nginx 的请求访问日志了。

继续修改配置文件，在接下去的 server 部分，将开头部分的 listern、server_name 修改为如下配置，表示同时捕获访问 localhost 的 80 端口和 9000 端口的请求：

```
1 listen 80;
2 listen 9000;
3 server_name  localhost;
```

复制代码

紧接着，在它下方增加路径映射的配置：

```
1 location ~ /books.* {
2     proxy_pass  http://localhost:8080;
3 }
```

复制代码

这表示如果 URI 是以 /books 开头，就映射到 8080 的端口上面去。

好了，如果是第一遍启动 Nginx，你可以直接执行 sudo nginx，如果已经启动，但是修改了配置文件，你可以重新加载 sudo nginx -s reload。

现在，去访问如下两个链接，你都应该看到前面见到的那个熟悉的页面：

```
http://localhost/books?category=art
http://localhost:9000/books?category=art
```

这证明端口映射成功了，并且，切换回访问日志的那个窗口，你应该可以看到类似这样的访问日志：

```
127.0.0.1 - - [04/Oct/2019:20:36:43 -0700] "GET /books?category=art HTTP/1.1" 
```

## 总结思考

今天我们介绍了 Ops 中配置管理的一些常见的方式，以及一些配置文件常见的组织形式。内容本身并不复杂，也没有介绍配置文件的格式（它在后面的文章中会有介绍），但是配置管理确实是程序员每天都在打交道的对象，自然全栈工程师也不例外，遵从好的实践可以养成良好的 Ops 习惯。

现在，提问时间又到了，我来提两个问题吧：

> 我们今天比较了常见的配置方式，其中一个是使用源代码中的常量，另一个是使用 Web 应用的运行参数，你觉得这两个各有什么优劣，各适合怎样的场景？
>
> 有程序员朋友认为，大型 Web 应用应该尽量少用代码层面的配置，而是把这些变化的部分放到独立的配置服务中，这样软件会比较灵活，修改简便，适合各式各样的场景。你觉得这个说法对吗？

好，今天的正文内容就到这里，下面是选修课堂。今天的选修课堂我想继续顺着 Ops 往下谈，来讲一讲程序员的“独立性”，而 Ops，恰好就是独立性的一大标志。

## 选修课堂：程序员的“独立性”

几乎所有的软件工程师，都会写代码，都会做测试，但是做项目的“独立性”是大相径庭的，这也是容易让人忽略的部分，但却是一个“程序员”向“工程师”蜕变的标志。“独立

性”在一定程度上决定了软件工程师的单兵作战能力，而对于全栈工程师来说，尤其如此。我们在技术上精进的同时，也需要提升独立完成项目的能力。而今天我们开始接触的 Ops，恰恰是程序员独立性提升的一个重要部分。我认为，软件工程的“独立性”可以分成这样几个阶段（再次强调一下这并不是从编程能力上来分的）。

### 第一阶段：编码工作者

拿到详细的设计文档，上面连许多方法接口都定义好了。写一些实现，调用一些既定的 API，然后根据设计文档来实现测试。这种情况很常见，比如在某些外包公司，就是如此。编码能力得到了一定的锻炼，但这扼杀了大部分的创造力，长此以往也许最终只能成为一个熟练工。我相信短期内这是一个可行的选择，但是从程序员成长的角度来看，这个阶段是一定要迈过去的。

### 第二阶段：需求的独立实现者

拿到了粗略的设计文档，需求和业务也已经大致描述清楚，接下去要做的就是发挥聪明才智把软件设计好，把代码写好，通过测试。在具备简单沟通的基础上，这项工作可以在安静和独立的环境中完成，因为项目经理、架构师和产品经理已经把那些复杂的技术或业务难题搞定了。这样的环境下，可以诞生许许多多代码设计优秀、实现逻辑清晰简洁的程序员，但是这始终只是在做一个“残缺”的项目而已。在大厂，无论国内外，很多程序员新手都是从这个阶段开始的。那从这个阶段开始，Ops 的工作就显山露水了，每天大量的安装、配置、部署工作，无论是在开发还是测试环境。

### 第三阶段：项目沟通者和管控者

程序员要和产品经理，甚至客户澄清需求；需要自行分析可行性，明确项目中的技术和业务难点；参与决定和管理迭代周期和计划表；组织和参与项目组内运作跟踪会议。编码以外的事情会占用相当多的时间，而且这些时间往往会用在各种沟通上。到了这个阶段的程序员，通常已经成为了团队中的顶梁柱。

### 第四阶段：从做项目到做产品

从做项目到做产品，区别是什么？做项目只需要做好一次或者很少的几次交付就可以了，而产品，则是要倾注心血于它的整个生命周期。做项目需要更多倾听用户需求，但是做产品更注重思考，思考用户的痛点和产品的定位远重于倾听用户表述，要把更多的精力花在产品定义、设计，思考怎样把技术、业务落地到产品实现上。在发布以后，如果幸运的话，产品会

有一个漫长的迭代和维护周期，Ops 工作也很可能成为你的重心，你会把主要的时间都投入到这里。

### 第五阶段：产品成长的见证人

也许很少人能够参与从零开始，经过创意、市场分析到产品设计的过程。在明确要做什么之前，程序员有大量的时间会花在产品探索性的工作上面。也许会做很多的产品原型，也许某些版本和功能在 A/B 测试之后就被放弃了，更有些产品在流传开来以前就销声匿迹了，或者很快就死在了抄袭和山寨手里。产品的更迭和换代总是千辛万苦，而看得到的部分往往如此简单，但是谁又知道它的历史有多曲折呢？

好，到这里，我想问一下，正在阅读的你，处于程序员“独立性”的哪个阶段呢？

## 扩展阅读

当 Operation 和 Development 相遇，DevOps 就是它们碰撞产生的火花。你可能已经听过 DevOps 无数次了，但是，如果你并不是很清楚它意味着什么的话，那么我推荐你阅读什么是 DevOps? 以及 DevOps: Breaking the Development-Operations barrier 这两篇文章。

文中简单介绍了反向代理，而它是几乎每个基于 Web 的全栈工程师都会接触的，维基百科的页面介绍了它常见的功能。

关于 Nginx，对于它配置的具体含义，如果想了解的话，除了官方的文档例子以外，nginxconfig.io 这个网站可以通过简单的配置，直观、清晰地生成和比较相应的配置文件。

有一篇 Top 5 configuration management tools 文章介绍了 5 种常见的配置管理工具，推荐阅读，中文译文在这里。




上一篇　27 | 特别放送：聊一聊代码审查

下一篇　29 | Ops三部曲之二：集群部署

## 精选留言 (1)

写留言

**tt**

2019-11-13

自己处在独立性的哪个阶段?

在银行的省级分行工作，因为人手不多，所以五个阶段哪个都干过。

第四个阶段回忆一下，应该是从16年发端的。项目诞生之初，出发点很简单。但是随着...

展开

作者回复: 👍