# **Optiver低延时硬件运维实践**

原创 愚夫 [愚夫一得](javascript:void(0);)

 2025年04月19日 06:56 浙江

本文总结于optiver多篇基础设施运维管理相关博客[1]

现在搞软件，云是标配。点几下鼠标，或者跑一段 Terraform/CloudFormation 代码，服务器、网络、数据库就都来了。这种便利性，归功于所谓的“基础设施即代码”（Infrastructure as Code, IaC）。它让基础设施管理变得像软件开发一样，可版本化、可重复、可审计。

但如果你的业务不是基于云的呢？比如，Optiver在高频交易（HFT）这个行当里，时间单位是纳秒（nanosecond），必须要用裸金属机器。你没法把核心交易系统放在 AWS 或 Google Cloud 上。

怎么办？答案是：回归本源，自己动手。你得在靠近交易所的数据中心里托管主机（Colocation），塞满自己的裸金属服务器、顶级网络交换机，用最短的光纤连起来。

但问题马上就来了。当你需要在全球多个这样的数据中心里，管理几十几百套物理设备时，复杂性会爆炸式增长。

* • 怎么保证芝加哥数据中心的交换机配置，和法兰克福的一致？
* • 新来一个工程师，怎么快速理解这套复杂的物理网络拓扑？
* • 某台服务器需要更换，如何确保新服务器的配置和旧的一模一样？
* • 出了问题，怎么快速回滚到上一个稳定状态？

传统的运维方式——Excel 表格、Wiki 文档、老师傅的口传心授——在这种规模和速度要求下，根本行不通。你会陷入无尽的“救火”和不确定性中。规模大了，就必须引入工程化的方法。

所以，即使在没有云的物理世界里，IaC 的思想也是我们迫切需要的。我们需要一种方法，能像代码一样管理这些硬件和它们的配置，确保：

1. 1. **一致性 (Consistency):** 同样的角色，配置必须一样。
2. 2. **可重复性 (Repeatability):** 部署一百次，结果都相同。
3. 3. **可审计性 (Auditability):** 随时能检查当前状态是否符合预期。
4. 4. **自动化 (Automation):** 减少手动操作，降低错误率，提高效率。

终极目标是，你能自信地回答那个问题：“**这整套基础设施，我能从零开始重建吗？**” 如果答案是“Yes”，那你就真的掌控了它。

### 挑战1：没有现成的云 API

云厂商之所以能让 IaC 变得简单，是因为他们提供了丰富的 API。你想创建虚拟机？调用 CreateInstance API。你想配置负载均衡？调用 ConfigureLoadBalancer API。Terraform 这些工具，本质上就是这些 API 的封装。

但在我们自己的数据中心里，面对的是一堆“哑巴”硬件。交换机、服务器，它们有自己的命令行接口（CLI）或一些有限的 API，但五花八门，各不相同。没有一个统一的、高层次的抽象接口，让你能说“我想要一个这样的网络”或者“给我部署一套这样的计算集群”。

既然没有现成的，那就自己造。这是解决很多底层问题的根本思路——**构建自己的抽象层**。

### 挑战1应对：声明式配置系统 (Intent System)

我们设计并实现了一个内部系统，我们称之为“声明式配置系统”（Intent System）。这个系统是我们对基础设施期望状态的“**唯一真实来源**”（Single Source of Truth）。它不是直接去操作硬件，而是用来**描述**和**存储**我们想要的基础设施应该是什么样子。

系统架构图如下：

这个系统包含几个关键部分，就像搭积木一样：

1. 1. **标准模型 (Standard):**  
   这是基石。没有标准，自动化就是空中楼阁。我们必须先定义好一切：机柜布局、线缆颜色和标签、网络架构（IP 地址规划、VLAN 划分、路由协议等）、服务器的操作系统配置、固件版本等等。标准模型一旦定下来，后续的自动化才有章可循。当然，标准模型也需要演进，系统需要支持多版本标准共存，因为改造现有数据中心是个漫长的过程。
2. 2. **高层定义 (High-level Definition):**  
   工程师不需要关心底层的命令细节。他们只需要用一种简单、易读的方式（我们选择了 YAML）来描述他们想要部署的“声明式配置”。比如，“我需要在这个机架的这个位置放一台服务器，它的管理 IP 是多少，连接到哪个交换机的哪个端口，运行哪个版本的操作系统”。
3. 3. **声明式配置代码 (Code to Implement Intent):**  
   我们编写了一些代码（提供了 SDK 和 CLI 工具），这些代码能读取工程师写的高层定义 YAML 文件，理解其声明式配置，并将其转换成对“Intent System API “的一系列调用。这些调用会在Intent System的数据库里创建或更新相应的记录，比如服务器、交换机、端口、连接关系等。
4. 4. **Intent System核心 (The Intent System Core):**  
   这是一个数据库（比如 PostgreSQL）加上一套 API（比如用 Django 和 FastAPI 构建）。数据库里存储了所有基础设施组件的模型及其关系（机架、设备、端口、线缆、IP 地址、VLAN 等）。API 允许查询这些信息，也允许通过上面的代码来写入期望状态。你可以把它想象成一个关于你基础设施的、活的、结构化的数据库，你也可以把它看成是CMDB ( Configuration Management Database )。
5. 5. **配置生成与下发流水线 (Provisioning Pipelines):**  
   Intent System只存储“声明”。要让“声明”变成现实，需要有东西去读取声明式配置，然后真正地去配置物理设备。这就是“配置流水线”的工作。它可以是一个生成交换机配置文件的脚本，也可以是一个自动化安装操作系统（比如使用 PXE Boot 和 Kickstart/Preseed）并进行后续配置的流程。  
   **关键点：** 这些流水线与Intent System是**解耦**的。它们只负责读取Intent System里的数据，然后使用具体的工具（可能是 Ansible，可能是自研脚本，可能是调用厂商 API，比如通过 NAPALM 库）去完成配置。这样做的好处是，如果将来我们想换一种配置工具，只需要修改流水线，而不需要动Intent System。
6. 6. **现状收集器 (Truth Collectors):**  
   只看“声明式配置”是不够的。现实世界总会发生偏差：有人可能手动改了配置、设备可能出故障、线缆可能插错了。我们需要知道基础设施的**实际状态**。因此，我们有“现状收集器”，它们会定期连接到设备上，拉取实际的配置信息、状态信息。
7. 7. **稽核审计 (Audits):**  
   这是闭环的关键。稽核审计程序会比较“Intent System”里的**期望状态**和“现状收集器”拿到的**实际状态**。如果两者有差异，系统就会发出告警，通知工程师去检查。这就像给你的基础设施写了单元测试，持续验证它的正确性。

### 一个实际例子：部署新的 Colocation

想象一下，我们要在芝加哥一个新的数据中心部署一套交易系统：

1. 1. **定义声明式配置:** 工程师编写 YAML 文件，描述需要多少机架、服务器、交换机，它们的型号、位置、连接关系、IP 地址、基础配置等。
2. 2. **注入声明式配置:** 运行我们的 SDK/CLI 工具，读取 YAML，调用Intent System API，将所有这些期望状态写入数据库。
3. 3. **物理部署:** Intent System可以生成一份详细的“跳线表”（Patching Instructions），指导数据中心的工程师如何安装设备、连接线缆。
4. 4. **连接验证:** 物理连接完成后，可以运行一个特殊的审计流水线，检查线缆是否都按声明式配置连接正确（比如通过 LLDP 协议）。
5. 5. **设备配置:** 验证通过后，运行配置流水线。流水线从Intent System读取每台设备的期望配置（比如主机名、IP 地址、NTP 服务器、交换机 VLAN 设置等），然后自动登录设备完成配置。
6. 6. **持续稽核:** 系统上线后，现状收集器和稽核程序会持续监控，确保实际状态与声明式配置一致。

### 挑战2：账实相符的稽核

Optiver为确保声明式配置和物理部署的一致性，采用定期从设备收集数据，并将其与Intent System中的预期状态进行比较的方法。该稽核审计方案也颇具特色。

从高层设计的角度来看，架构图如下：

涉及到两部分：

* • NetPipe：网络设备的硬件抽象层，它是一个 Python 应用程序，可以通过在 NAPALM（一个开源的网络设备描述抽象层） 和一些自定义管道之上构建的定制抽象层与来自不同供应商的许多设备进行交互。即可向下推送配置，也可向上收集配置。
* • Intent System：声明式配置系统，以 Django 应用程序的形式实现，存放了所有的设备配置信息。

### 挑战2应对：网络配置的“提升”与“降低”

上面的框架听起来不复杂，但在网络设备配置这块，水尤其深。

网络配置通常是大段的、有层级结构的文本，而且不同厂商（比如 Arista, Cisco, Juniper）的语法和特性差异很大。特别是像 BGP 路由策略这种复杂配置，用简单的键值对很难完整表达，也很难做精确的稽核审计。

这里Optiver借鉴了编译器理论的一些思想：

* • **“提升” (Lifting) 配置 - 理解现实:**  
  为了能准确地稽核审计网络设备的真实状态，我们需要把厂商特定的、复杂的文本配置，转换成一种标准化的、结构化的数据。
  1. 1. **解析 (Parsing):** 我们不能只靠正则表达式。对于复杂的配置，我们为每个厂商的配置语法编写了**解析器**（Parser），使用了像 Lark 这样的库。解析器能把配置文本转换成一棵**抽象语法树**（Abstract Syntax Tree, AST）。
  2. 2. **转换 (Transformation):** AST 仍然带有厂商的烙印。我们接着用一个“转换器”（Transformer），遍历 AST，把它转换成一种我们自己定义的、与厂商无关的**中间表示**（Intermediate Representation, IR）。这个 IR 是标准化的，能准确描述网络配置的语义（比如一个接口的 IP 地址、一个 BGP Peer 的配置）。  
     这个从设备原始配置到标准化 IR 的过程，我们称之为“提升”，因为它把底层的、具体的配置细节，“提升”到了一个更高层、更抽象、统一的数据模型。有了这个 IR，稽核审计就变得容易多了：比较Intent System里的期望 IR 和从设备“提升”上来的实际 IR。
* • **“降低” (Lowering) 配置 - 应用声明式配置:**  
  反过来，当我们需要根据声明式配置去配置设备时，就需要一个“降低”的过程。
  1. 1. **生成配置:** 配置流水线从Intent System获取目标的 IR。
  2. 2. **模板渲染:** 使用模板引擎（比如 Jinja2），根据这个 IR 和目标设备的厂商类型，生成该厂商特定的配置命令文本。
  3. 3. **差异比较:** 在真正下发配置前，我们生成一个“差异”（Diff），比较将要下发的配置和设备当前配置（也通过“提升”得到 IR 再比较），让工程师可以清晰地看到将要发生的改动。我们甚至为此做了一个基于 Monaco Editor（VS Code 的编辑器核心）的可视化 Diff 工具。
  4. 4. **下发配置:** 工程师确认无误后，流水线再通过 NAPALM 等库，把生成的配置命令推送到设备上。

### 这样做的好处是什么？

* • **真正解耦:** Intent System只关心标准化的 IR，不关心具体厂商。增加对新厂商设备的支持，主要是编写新的解析器、转换器和模板。
* • **处理复杂性:** 基于解析器的方法，能更好地处理复杂和嵌套的网络配置，比正则表达式或简单的键值对匹配要健壮得多。
* • **可测试性:** 解析器和转换器本身就是代码，可以编写单元测试来保证其正确性。

# 总结

你看，即使在严苛的、非云的物理环境中，IaC 不仅是可行的，而且对于管理规模化、低延迟的基础设施来说，几乎是**必须**的。关键在于，你不能指望现成的工具能解决所有问题。你需要：

1. 1. **回归第一性原理：** 真正理解问题的本质（我们需要管理状态、确保一致性、自动化变更）。
2. 2. **构建合适的抽象：** 设计你自己的“Intent System”和数据模型（比如我们的 IR）。
3. 3. **拥抱基础技术：** 不要害怕使用像解析器这样的“硬核”技术，它们是解决复杂问题的利器。
4. 4. **建立闭环：** 不仅要能下发配置，还要能持续稽核审计，确保现实与声明式配置一致。

这听起来比在云上点点鼠标要复杂得多，也确实需要投入更多的研发力量。但对于像 Optiver 这样，业务成败系于基础设施性能和稳定性的公司来说，这种投入是值得的。它给了我们对自身基础设施的深度掌控力，让我们能够快速、可靠地响应业务需求，最终在竞争激烈的市场中获得优势。

这套方法论和实践，不仅仅适用于交易行业。任何需要自己管理物理基础设施，并且对可靠性、一致性有高要求的场景，都可以从中借鉴。核心思想是共通的：**像管理软件版本一样管理基础设施**。

点击**关注**，共同进步

卡通画

AI 生成的内容可能不正确。

**愚夫一得**

某交易所技术人。在持续学习中和你聊聊交易、研发、运维、数字化、管理那些事。

135篇原创内容

公众号

#### 引用链接

[1] *https://optiver.com/working-at-optiver/career-hub/infrastructure-as-code-in-a-non-cloud-environment/*

[2] *https://optiver.com/working-at-optiver/career-hub/optivers-approach-to-network-devices-and-infrastructure-as-code/*