架构师

ARCHITECT

特刊

微服务监控实践

SPECIAL ISSUE

Jan . 2019

架构师特刊

Geekbang. 极客邦科技 | InfoQ

CONTENTS / 目录

本期主编：张婵　　内容投稿：editors@geekbang.com

流程编辑：丁晓昀　　业务合作：hezuo@geekbang.org

发行人：　霍泰稳　　反馈投诉：feedback@geekbang.org

卷首语

流水的架构，铁打的监控

作者 张婵

随着云计算为应用打造好分布式的基础设施，软件架构也从 SOA 逐渐进化到微服务。微服务粒度更细，每种服务只做一件事，是一种松耦合的能够被独立开发和部署的无状态化服务。微服务的灵活性和高可用性能让部署更便捷，缩短交付周期。但是同时，引入微服务也会带来更多的技术挑战，对于架构设计和业务梳理要求更高，现如今业务快速发展，服务器越来越多，中间件、应用、微服务、数据库等也越来越多样化，监控是微服务控制系统的关键部分，软件越复杂，就越难了解其性能及问题排障。

微服务架构下的监控有如下难点：

- 监控对象动态可变，无法进行预先配置；
- 监控范围非常繁杂，各类监控难以互相融合；
- 微服务实例间的调用关系非常复杂，故障排查会很困难；
- 微服务架构仍在快速发展，难以抽象出稳定的通用监控模型。

而在监控系统的设计上，技术选型也是一个不容忽视的问题。很多公司都是根据自己的系统架构和业务需求自研监控系统，但同时现在也有很多成熟的开源监控软件，在它们的基础之上进行二次开发就能满足自己的需求，还能节省不少开发和后期维护成本。

目前我们正处在云原生的技术浪潮之中，Kubernetes 及围绕其的生态系统能很好地适应现在软件架构。而 Kubernetes 社区推崇的监控工具 Prometheus 也是微服务架构下比较理想的监控系统。Prometheus 使用 Pull 模型，具有灵活的数据模型和强大的查询能力，其生态系统也比较健全。

在今年 8 月份，Prometheus 正式从 CNCF 毕业，表示它本身已经具备一定的成熟度和稳定性，项目在 GitHub 上也获得了超过 2 万的 Star 数，足以见其受欢迎的程度。即便如此，Prometheus 也还有其欠缺的地方难以满足微服务监控，比如数据持久化方面做得不好；仅适用于维度监控，不能用于日志监控、分布式追踪等范围等等。所以目前使用 Prometheus 的团队多在其基础之上与 ELK 技术栈合用，并对监控系统进行高可用的架构设计。

另外，在微服务架构里，软件系统通常会被拆分为数十甚至数百个微服务，这种拆分会使得监控数据爆炸增长，监控系统必须具备处理和展示这些数据的能力。因此在微服务监控中，也需要用到一些可观察性工具，用数据面板展示一些指标，方便进行报警及后续问题排查。

现在微服务模式正在逐步走向巅峰，也会有越来越多的团队向微服务架构转型。但是监控系统的设计思路到底如何理清？在监控方面有丰富的经验，开源实时监控系统 CAT 的作者吴其敏曾在采访中说，监控首先要解决的是目标设定，到底要解决什么问题，关注什么指标，不管现存的还是潜在的，如果没有问题就不需要监控。如果目标明确了，第二步就可以指定方案，怎么样通过已有的机制解决，或者使用新的创造性的方法，达到目标。接下来就是方案的落地实施了，数据在哪里，需要通过什么手段拿到。最后一步，等系统上线有结果后，需要去做验证，是不是真的达到了预期，还是目标设定需要调整。这四步可以不断迭代，直到各种监控问题都可以被解决。

主办方 Geekbang. 极客邦科技 InfoQ

大咖助阵

联席主席：程立（鲁肃）
蚂蚁金服 / 首席技术官

联席主席：于旸(TK)
腾讯 / 玄武实验室总监

联席主席：祁安龙
百度搜索公司
首席架构师 / 技术委员会主席

联席主席：洪强宁
爱因互动 / 创始人兼CTO

专题出品人：程劭非（寒冬）
自由职业 / 前端工程师

专题出品人：董志强（Killer）
腾讯 / 云鼎实验室
云基础安全中心负责人

分享嘉宾

Stuart Douglas
Red Hat
Undertow project leader

Boris Scholl
Microsoft
Azure Compute
Product Architect

Emily Jiang
（蒋丰慧）
IBM MicroProfile
CDI首席架构师

王明刚
英特尔
软件工程师

李鹏
阿里云
容器服务资深架构师

姜承尧
腾讯金融科技
副总监

赵健博
快手 高级架构师
大数据架构团队负责人

姜宝琦
百度
资深工程师

优惠

100+技术大咖 实战解析

如果您有任何需要或问题，请联系我们：

购票热线：010-53935761 票务微信：qcon-0410

网易云基于 Prometheus 的微服务监控实践

作者 陈咨余，王添

一、当监控遇上微服务

在过去数年里，微服务的落地一直都是业界重点关注的问题，其始终面临着部署、监控、配置和治理等方面的挑战。轻舟微服务平台是网易云为企业提供的一套微服务解决方案，其中微服务监控是其关注的重点问题之一。与传统监控相比，微服务监控面临着更多难点，包括：

1. 监控对象动态可变，无法进行预先配置；
2. 监控范围非常繁杂，各类监控难以互相融合；
3. 微服务实例间的调用关系非常复杂，故障排查会很困难；
4. 微服务架构仍在快速发展，难以抽象出稳定的通用监控模型。

在工程角度也面临着不少考验，如：

- 在微服务架构里，软件系统通常会被拆分为数十甚至数百个微服务，这种拆分会使得监控数据爆炸增长，监控系统必须具备处理和展示这些数据的能力；
- 监控系统必须要保证可靠性，具体而言：保证不会因为单点故障而全局失效，监控数据有备份机制，系统各服务的实例均可通过备份数据得到恢复；
- 监控系统必须支持云上部署及快速水平扩容，这既是云原生的基本要求，也符合企业系统微服务化演进的实际情况。

二、微服务监控的技术选型

微服务监控的诸多挑战使得我们不得不慎重地进行技术选型。选择开源还是再造轮子，这个问题在项目初期一直困扰着我们，经过一段时间的调研和论证，开源项目 Prometheus 成了最终的答案。

Prometheus 是 CNCF 旗下的项目，该项目是一个用于系统和应用服务监控的软件，它能够以给定的时间间隔从给定目标中收集监控指标，并能够通过特定查询表达式获取查询结果。

选择 Prometheus 的主要原因是：

1. 灵活的数据模型：在 Prometheus 里，监控数据是由值、时间戳和标签表组成的，其中监控数据的源信息是完全记录在标签表里的；同时 Prometheus 支持在监控数据采集阶段对监控数据的标签表进行修改，这使其具备强大的扩展能力；
2. 强大的查询能力：Prometheus 提供有数据查询语言 PromQL。从表现上来看，PromQL 提供了大量的数据计算函数，大部分情况下用户都可以直接通过 PromQL 从 Prometheus 里查询到需要的聚合数据；
3. 健全的生态 : Prometheus 能够直接对常见操作系统、中间件、数据库、硬件及编程语言进行监控；同时社区提供有 Java/Golang/Ruby 语言客户端 SDK，用户能够快速实现自定义监控项及监控逻辑；
4. 良好的性能：在性能方面来看，Prometheus 提供了 PromBench 基准测试，从最新测试结果来看，在硬件资源满足的情况下，Prometheus 单实例在每秒采集 10w 条监控数据的情况下，在数据处理和查询方面依然有着不错的性能表现；
5. 更契合的架构：采用推模型的监控系统，客户端需要负责在服务

端上进行注册及监控数据推送；而在 Prometheus 采用的拉模型架构里，具体的数据拉取行为是完全由服务端来决定的。服务端是可以基于某种服务发现机制来自动发现监控对象，多个服务端之间能够通过集群机制来实现数据分片。推模型想要实现相同的功能，通常需要客户端进行配合，这在微服务架构里是比较困难的；

6. 成熟的社区：Prometheus 是 CNCF 组织第二个毕业的开源项目，拥有活跃的社区；成立至今，社区已经发布了一百多个 P 版本，项目在 github 上获得的 star 数超过了 2 万。

Prometheus 虽然在上述六方面拥有优势，但其仍然难以满足微服务监控的所有需求，具体而言：

- 仅适用于维度监控，不能用于日志监控、分布式追踪等范围；
- 告警规则和告警联系人仅支持通过静态文件配置；
- 原生支持的数据聚合函数有限且不支持扩展；

这些不足都说明了一个事实，Prometheus 社区版并非微服务监控的最终答案。

三、我们的答案 - 轻舟微服务监控系统的设计

从大的方面来看，我们将微服务监控划分为维度监控、日志监控、分布式追踪等三部分。其中维度监控在整个微服务监控里最为重要，所占比例也最大，此类监控的层级有如下划分：

1. 基础设施监控：主要对各个微服务实例所在的基础设施进行监控，具体包括这些设施的运行状态、资源使用情况及系统日志进行监控，一般而言微服务应用实例会运行在容器里，因此这个维度的监控对象也通常包含有容器编排系统、持续构建系统、镜像仓库等，这些对象的具体监控指标的范围包括对象的健康状态、运行状态、资源使用情况等；
2. 微服务通用监控：主要针对微服务通用指标进行监控，包括服务实例处理请求的情况及实例调用其它服务的情况，具体而言包括请求总数、请求处理时延（中位数，包括有 90、95 和 99 值）、请求结果（成功、失败、熔断、限流、超时和拒绝）统计、调用其它服务的结果（成功、失败、熔断、限流、超时和拒绝）统计及时延（中位数，包括有 90、95 和 99 值）；
3. 应用监控：主要对具体的微服务实例进行性能监控，通过数据自动化收集、数据可视化展示，使用户能够及时、全面地掌控各个

实例的性能情况，定位性能瓶颈。这一维度重点在于提供丰富的应用性能展示及性能问题定位功能，包括应用响应时间、吞吐量和状态的展示，慢响应和错误明细的查询。

4. 通用中间件：我们没有预置这个维度的监控到系统里，不过得益于 Prometheus 完善的生态，系统保留有对常用数据库、消息队列及缓存进行监控的能力，具体包括 MySQL、Redis、Memcached、Consul、RabbitMQ 及 Kafka 等。

在工程实现方面，我们进行了如下设计：

- 用 Prometheus 原生的联邦集群部署模式，使得全部监控数据分片处理；分片处理机制使得只需要增加实例个数就能够应对海量监控数据问题；
- 多 Prometheus 实例作用于同一监控对象，使得单一实例失效也不会影响到此对象的监控，满足高可用的要求；
- 监控系统所有组件及配置均实现容器化并由 Kubernetes 编排；理论上，在任意 Kubernetes 集群里都能够一键部署；系统需要变更时，仅需修改相关编排文件，即可完成改变。

对上文提到的几个 Prometheus 不足之处，我们进行了如下设计：

- 引入 ELK 实现日志监控，Logstash 负责采集日志，日志数据被保存到 Elasticsearch 里，用户则可以通过 Kibana 查询到具体应用的日志；
- 基于 OpenTracing 实现分布式追踪，最终完成了应用拓扑关系展示，调用链查询等功能；
- 对 Netflix Turbine 进行了二次开发，将微服务框架的秒级监控纳入到系统能力集里。

四、多场景多维度 - 轻舟微服务监控系统的实现细节

从架构上来讲，轻舟微服务监控系统在设计时考虑到有多种用户场景，并为此设计了多种模式，包括精简模式、读写优化模式及多环境模式。

图 4-1 描述了精简模式的架构，精简模式的主要特点在于部署简单，容易维护。从整体上来看，我们使用了 Prometheus 经典的联邦集群部署方案，处于叶子节点的 Prometheus 分片采集处理监控数据；处于根节点的 Prometheus 则直接从各个叶子节点上拉取处理后的监控数据并负责处理外部的查询请求；告警服务则定期从位于根节点的 Prometheus 里查询监控数据，在发现数据达到阈值时发送告警通知至对应联系人。这个模式

基本上解决了微服务监控的数据分片处理、多维度及系统可靠性问题，同时 ELK 系统及轻舟 APM 服务在日志监控和分布式追踪方面进行功能补充，在规模不大的时候是能够满足用户需求的。

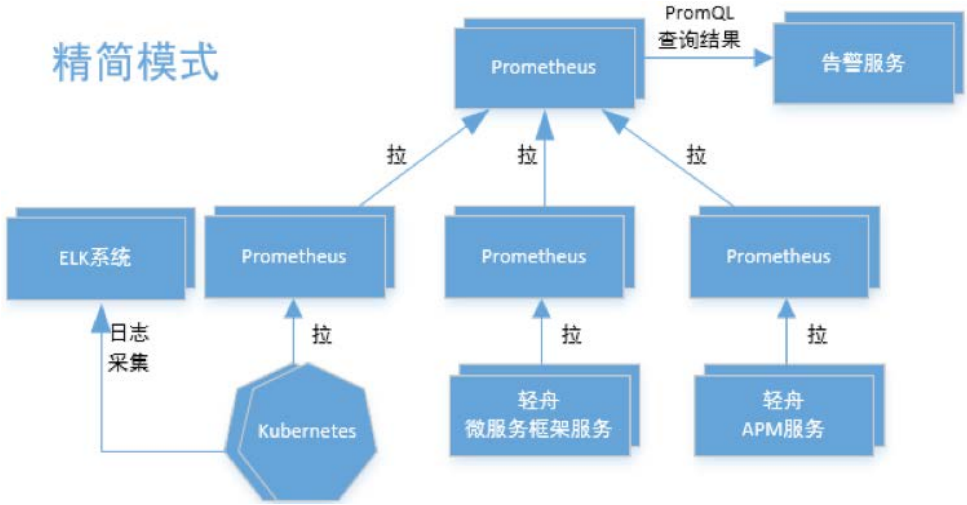


图 4-1 精简模式架构

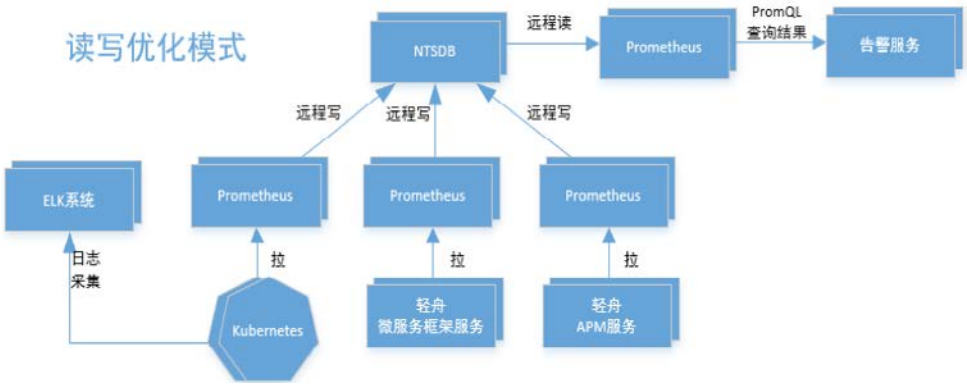


图 4-2 读写优化模式架构

在精简模式下，所有的维度监控数据都保存在本地磁盘里面，当本地磁盘发生问题时，数据会有丢失的风险；同时精简模式的可靠性主要靠多个 Prometheus 实例执行相同的监控任务来保证，多个实例之间实际上是没有数据同步的，这使得数据有不一致的风险。为了解决上述问题，我们在读写优化模式里加入了网易自研的分布式时序数据库 NTSDB，利用 Prometheus 的 Remote Write/Read 机制将监控数据存取操作实际交由 NTSDB 来处理。由于 NTSDB 自带数据同步机制，所以采用这种模式的数据安全性要高于第一种。

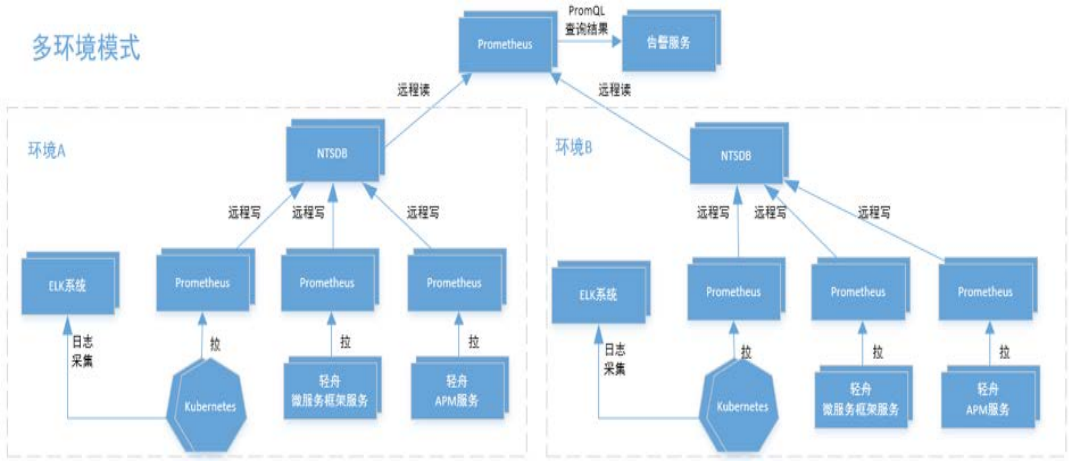


图 4-3 多环境模式架构

对于规模较大的用户而言，还会存在多个物理隔离的机房。这些机房之间通常仅能够通过网络专线通信。针对这种情况，我们设计了多环境模式，在这个模式里，每个环境的监控数据都保存在对应环境的 NTSDB 集群里，仅当需要进行数据查询时才会跨环境通信。这个模式在前两个模式之外，解决了微服务监控的多数据中心及多 AZ 问题。

维度监控是轻舟微服务监控系统的主要部分，其实现细节如下所述：

1. 基础设施监控：就轻舟微服务平台的具体情况来看，主要指的是容器监控。轻舟微服务的容器编排系统是 Kubernetes，Prometheus 则原生支持 Kubernetes 服务发现机制，这使得我们解决了监控对象发现问题；同时 Kubernetes 各组件原生支持 Prometheus，开源社区也提供了 Node exporter、kube-state-metrics exporter 及 Ceph Exporter，这些组件已经能够满足全部功能需求，所以在基础设施监控上，系统完全采用了开源方案。
2. 微服务框架监控：图 4-4 显示了这一维度监控的实现。在这一维度里，我们自研了两个组件，nsf-agent 和 nsf-turbine。nsf-agent 主要负责从服务实例里收集并上报原始监控数据；nsf-turbine 则主要负责接收 nsf-agnet 推送的监控数据，同时对原始监控数据进行聚合及通过暴露这些监控数据给 Prometheus；Prometheus 定期拉取 nsf-turbine 暴露的监控数据并为这些数据提供持久化及数据查询能力。另外 ,nsf-turbine 也提供了相对简单的监控数据查询接口，用户能够通过这个接口查询到当日的实时统计数据及秒级监控数据。

3. 应用监控：从总的结构上来讲，应用监控分为客户端、Collector 及 WEB 服务端部分；其中客户端收集并上报应用的监控数据，这部分支持使用网易云自研的 APM 客户端或者开源的 Zipkin 及 Jaeger 客户端，自研的 APM 客户端能够以无代码侵入的方式进行数据采集，采集到的数据是满足 OpenTracing 规范的；各个客户端采集的监控数据将被上报到 Collector 里进行处理，处理后的数据将被保存到 MySQL、ElasticSearch 或 Redis 里；WEB 服务端部分则负责提供标准接口给 Prometheus 拉取数据。

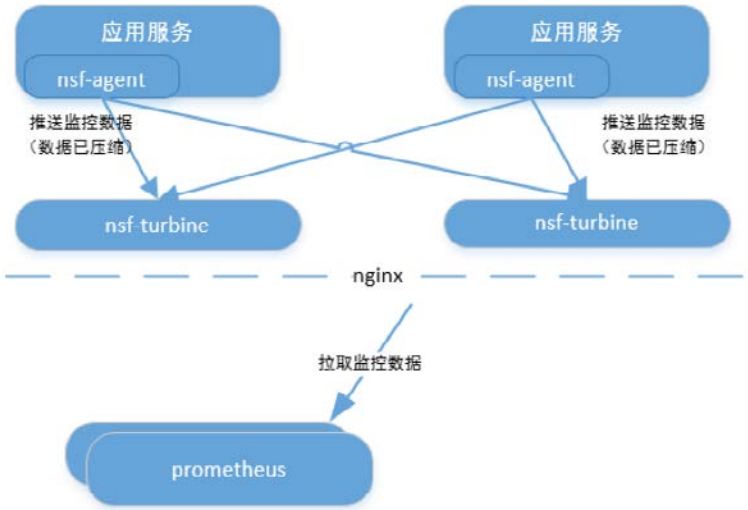


图 4-4 微服务框架监控具体架构

当然，在基于 Prometheus 实现轻舟微服务监控系统的过程里，我们也踩了一些坑，如：

- Prometheus 的各种计算函数都会对结果进行一定预估处理，其返回值通常都不是精确值。例如当聚合规则为获取过去一小时的监控值之和，但实际只收集到十五分钟监控数据时，这时候聚合出来的数据就是预估的值。如果需求非常精确的结果，需要通过客户端来聚合计算。
- Prometheus 不支持定时整点进行聚合计算，只能计算过去一段时间的值；无法获取到诸如当天零点到次日零点这种规则的聚合数据。如有类似于这种的需求，需要通过客户端直接聚合。
- Prometheus 预定义的计算规则、查询表达式是非常多的，而且会根据具体需求进行变动，如果不采用版本管理工具来维护，是非常容易出错的。

五、新的起点 - 我们的进展以及未来

目前轻舟微服务监控系统已经具备了下面的特性：

- 高可用：在精简模式里，同一份监控数据至少由两个 Prometheus 实例来采集；在读写优化和多环境模式里，监控数据保存在分布式时序数据库 NTSDB 里；任意一个 Prometheus 失效都不会影响到系统的整体功能。
- 全局立体化：系统已经集成了基础设施、微服务及应用等三个维度的监控告警；在日志监控和分布式追踪等方面也提供了相应的日志及调用链查询审计功能；这些已经基本上涵盖了微服务监控的全部功能需求。
- 可动态调整：在前文提到的各种部署模式里，我们对监控数据的采集和处理进行了分片。目前系统支持通过调整数据分片配置及 Prometheus 实例数，来满足各种规模的微服务系统的监控需求。

另外，在不远的将来，我们还会在下面几个方面持续改进轻舟微服务监控系统：

1. 系统自监控、智能监控及分布式追踪能力强化；
2. 结合 Thanos、Druid 等组件，扩充部署模式及增强聚合能力；
3. 增强监控及告警响应速度。

通过这些优化，轻舟微服务监控系统能够更好地为企业的微服务系统保驾护航。

51 信用卡在微服务架构下的监控平台架构实践

作者 杨帆

一、背景介绍

51 信用卡的技术架构是基于 Spring Cloud 所打造的微服务体系，随着业务的飞速发展，不断增多的微服务以及指标给监控平台带来了极大的挑战。监控团队在开源 vs 自研，灵活 vs 稳定等问题上需要不断做出权衡，以应对飞速发展的需求。本次将会分享我们在微服务下的白盒监控思考，以及如何将时下社区流行的 Spring Cloud，K8S，Prometheus 等开源技术在企业落地。

本文整理自杨帆在 QCon 2018 北京站上的演讲，原标题为《51 信用卡在微服务架构下的监控平台架构实践》

这次主要讲的是关于微服务的监控，微服务看起来很美话，但实践起

来却有很多坑，希望这次的分享能给大家一些收获或者思考。

二、传统的监控分层

传统的监控一般会将监控分层，比如我们常用的分层方式是将监控分成基础设施、系统、应用、业务和用户端这几层，分完层后将每层的监控做到位。

而在传统的监控里，zabbix 是最常用的开源软件，zabbix 的优点主要是成熟可靠，社区非常强大，几乎你的需求，社区都有一套对应的解决方案，但 zabbix 的缺点也很明显，就是太难用，很多监控配置加起来成本很高，甚至很多运维用了很久的 zabbix，还没学会怎么配置 HTTP 监控，这在应用较少的时候，还不是很明显的问题，但是到了微服务时代，这个问题就暴露得非常明显了，而且 zabbix 以机器为维度的监控也无法适用微服务时代的理念。

三、以服务为维度的监控

微服务监控比起传统应用的监控，最明显的改变就是视角的改变，

我们把监控从分层 + 机器的视角转换成以服务为中心的视角，在微服务的视角下，我们的监控可以分为指标监控、链路监控和日志监控，在开源社区，这些监控也都有对应的解决方案，比如指标监控有 prometheus、influxdb，链路监控有 zipkin、pinpoint，日志则有 elk。

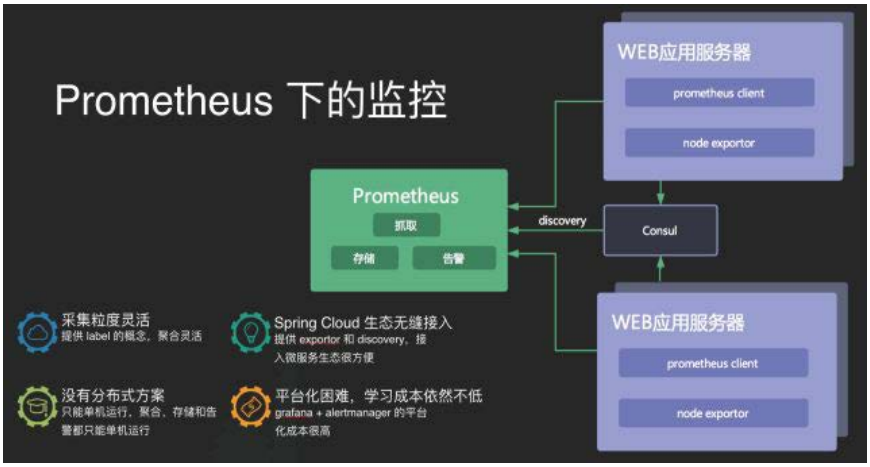
在 51 信用卡发展起步的时候，我们也同样使用这些开源方案来解决我们的监控问题，但当我们业务快速发展的时候，我们开始不断碰到监控上的挑战，其中有部分是互联网金融特有的，另一部分是微服务所带给我们的。

微服务监控有什么特点？用一句话概括就是服务特别多，服务间的调用也变得非常复杂。我们其实是微服务的受害者，其实业内很多人做的架构只是服务化，并不够「微」，而我们做的比较彻底，我们线上很多服务都只有一个 API，但这样造成线上指标非常多，告警也非常多，读和写的压力都非常大。

互联网金融是一个跟钱息息相关的行业，所以互联网金融对监控也有自己的要求。首先是对故障的容忍程度很低，监控的有效性需要被反复确认，其次是对监控的覆盖度，黑盒监控在互联网金融里很难行得通，白盒监控变得越来越重要，开发们迫切需要对自己的应用有全面的了解。然后是对告警的及时以及快速诊断有更高的需求，告警以及诊断信息在 10 分钟内发出与 5 分钟内发出有很大的差别，举个例子，如果有个活动有个漏洞被黑产行业抓住，如果能早一分钟确定问题关闭后门，就能给公司挽回巨大的损失。

四、Prometheus 下的监控

51 信用卡在早期也同样使用 Prometheus，其实 Prometheus 是个很棒的产品，白盒监控的理念也很先进，自带告警以及 PromQL，稍微学习之



后便能上手，作为 CNCF 的项目与 K8S 等开源产品结合得也很好。

在随着服务的增长，我们开始不断地踩坑，首先突出的问题就是 Prometheus 没有现成的分布式方案，性能遇到单机瓶颈之后只能手动给业务划分集群并且之间的数据不能共享，然后拉模式在兼容多数据源上也显得力不从心，比如我们有场景需要指定精确的时间，还有比如我们有些数据是从日志来的或是从 Kafka 来的，这些都没有现成的方案。

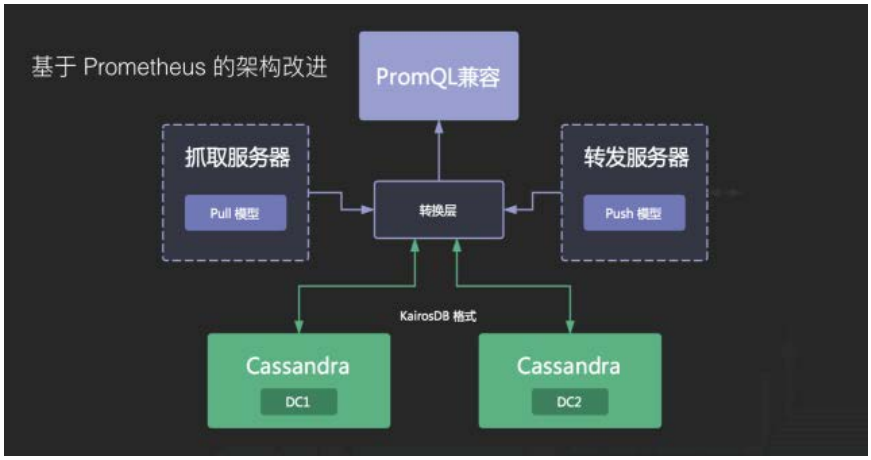
微服务的指标增长其实比想像得要快很多，因为微服务架构下，我们总是迫切想要把应用的每个细节都搞清楚，比如主机指标、虚拟机指标、容器指标、应用性能指标、应用间调用指标、日志指标以及自定义的业务指标等等，甚至在这些指标下，我们还会给指标打上更多的标签，比如是哪个进程，哪个机房，我们大致算过一笔账，一个服务即使开发什么都不做，他通过基础框架就自带了 5000 个指标。

我们内部也讨论过为什么指标会这么多，能不能把一些指标去掉，但很快我们就否决了去指标的想法，我们觉得业界的趋势是白盒监控会变得越来越重要，APM 的概念会变得越来越重要，devops 会和白盒监控不断发生化学反应，变成一种潮流。

而在 51 信用卡，我们是怎么解决的呢，其实很简单，用三个字概况就是「平台化」，平台化的好处很多，最直观的好处就是提供了一个统一的平台去处理监控问题，并给开发带来了统一的使用体验。

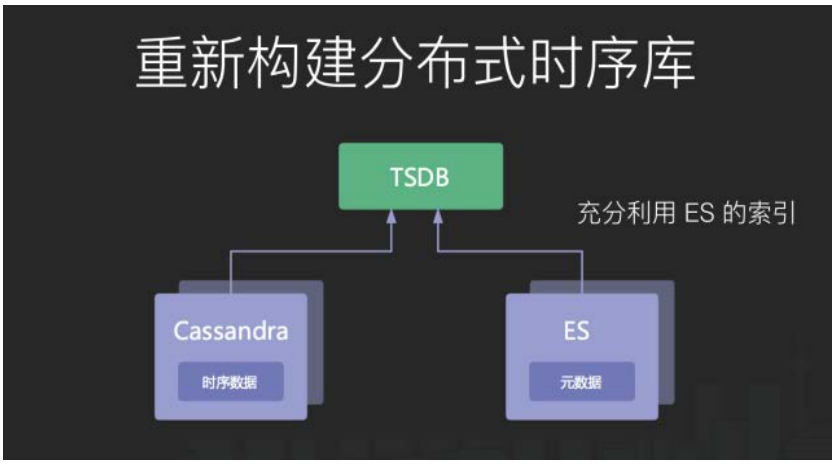
五、基于 Prometheus 的架构改进

我们首先要解决的问题是如何构建对上层统一的存储，一开始我们基于 Prometheus 的生态做了一些架构改进，将底层换成分布式的列式存储 Cassandra，并开发了推送服务和拉取服务来兼容原先的数据模型，在上层，我们开发了兼容 PromQL 的界面提供给开发使用。

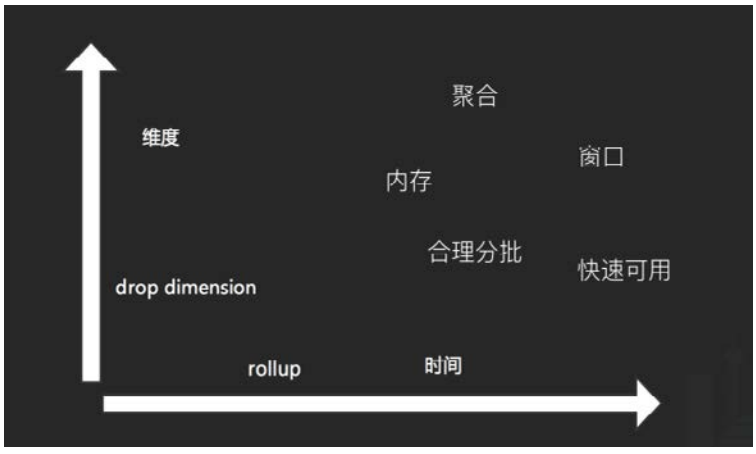


但很快，我们就碰到了新的问题。

首先是 labels 的匹配效率问题，当指标名相同的时候，由于 label 是自由组合的，在匹配部分 label 的时候，我们需要先将 labels 的元数据全部读出来，然后进行过滤，这样的效率会显得很低，我们的做法是在 label 元数据上面加上倒排索引，因为我们是分布式方案，倒排索引本身也需要分布式，所以我们直接使用 ES 来帮我们构建元数据。



第二个问题是预聚合，比如我们只想看 API 的整体访问量，但做这个查询的时候，我们会在底层读到 4 份数据，然后再聚合再显示出来，这样无疑也是一种浪费。我们的做法是引入预聚合机制，在纵向，我们需要舍弃维度，在横向，我们需要聚合时间轴。对于分布式而言，预聚合会显得比较麻烦，因为我们需要考虑的东西比较多，比如需要在内存里完成，需要合理将数据分批分配到不同机器，需要有一个窗口机制保证数据的及时有效又高性能。业内常用的做法是引入一个 Storm 这样的流式计算或者 Spark Streaming 这样的微批计算，然后将计算完的结果推入缓存或者内存



供告警来使用。

第三个问题是 Metric 的长度，因为在列式存储的底层，我们直接借鉴 Kairosdb 的存储格式，兼容 UTF8 的 Metric 而直接讲 Metric 转换成二进制存储在数据库里，如果指标很少，这个问题不大，但如果指标非常多，这会造成底层存储的存储浪费，而且影响索引的效率，这个问题的解决办法也很简单，直接引入一个 Bitmap 机制就可以解决。

第四个问题是维度重复，比如我们有三个指标，这三个指标的维度都一样，但这三个指标完全不同，代表不同的值，但存储在数据库的时候，它会占用三个 series 的空间，查找的时候也不够高效，因为往往三个指标会同时查询同时展示。这个解决办法是将数据库里原本的 value 定义成多类型支持，不只是双精度浮点或是整型，增加 Map 的支持类型，并在数据库的上层做兼容。

当初我们在解决这些问题的时候，我们发现社区已经有一个比较好的解决方案了，就是 Druid，不是阿里那个数据库连接池 Druid，而是 druid.io。它比较好地满足了 Bitmap、预聚合、复合类型、倒排索引、冷热数据这些需求。

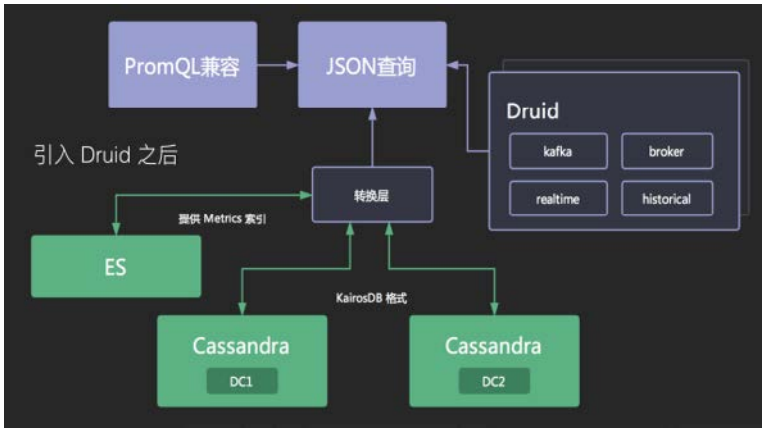
我们将拉服务和收服务做了进一步的改进，可以自动将数据做转换，兼容原先数据模型的同时，将数据也投递一份到 Druid 里。

至此我们基本完成了一个能够满足需求的存储架构改进。

最后限于时间关系，和大家分享两个非常有用又容易实践的告警智能诊断：

第一个是和日志监控的联动，当一个告警发生的时候，我们以时间、服务为维度去匹配 ERROR 或是 Exception 日志，并以 simhash 之类的相似算法排序日志，就会非常快速地找到问题的直接原因，不一定是 Root Cause，但告警的时候如果附上这个日志，对开发排查问题的效率会有很

大的帮助。



第二个是和链路监控的联动，当一个告警发生的时候，我们同样以时间、服务查询链路监控，并从日志监控里排名靠前的日志提取 trace id 后进行过滤，能很快发现故障的关联原因，这同样不一定是 Root Cause（很有可能是），但同样对开发排查问题很有帮助。

六、未来

最后展望一下未来，我们会继续在 3 个方向发力。

第一个是更好的底层存储，Cassandra 毕竟是一个通用的列式数据库，对时序数据来说，有很多不好优化的地方，我们期望能够自研一个时序数据库来满足我们的业务需求。

第二个是智能化的监控和告警，运用合适的算法并加上机器学习或是深度学习，探索出无阈值的告警体系，并自动分析出告警之间的关联关系，给出根因。

第三个是 APM 和监控的更紧密结合，将链路监控、日志监控和指标监控直接合并，更深度地诊断系统，系统没有无法探查的秘密。

360 容器平台基于 Prometheus 的监控实践

作者 王希刚

背景

360 在做容器化平台之前，有一个基于小米开源的 Open-Falcon 进行二次开发的老监控系 (Wonder)，这个系统承揽了公司所有的物理机和虚拟机的监控任务。随着容器技术的普及，以容器的方式在创建应用时，由于 Kubernetes 容器编排系统部署的服务具有弹性扩容的特性，而老的监控系统无法感知这些动态创建的服务，已经不适合容器化的场景，所以 360 团队就搭建了一套可以支持服务发现的新监控系统。

目前 360 一共有 5 个 k8s 线上集群，分布在北京上海和深圳，此外还有一些 GPU 集群。

容器平台构建以后带来了以下几个方面的好处：

- 节省资源。传统一台物理机只能部署一个实例，虚拟机能部署几个服务，而使用容器一台机器上能部署几十个服务。
- 提高效率。1. 缩短流程。老系统要增加机器需要提前申请，而使用 Kubernetes 容器平台只要整个资源池里有充足的资源，不用提交预算就可以直接使用。2. 弹性扩容。在流量高峰期，容器平台可以快速扩容；在流量不多的时段，空闲的资源可以处理其他离线任务，对资源的利用率高。
- 高可用。容器平台可以保证运行的服务数量总是能达到预期。
- 减轻运维负担。之前所有的部署上线活动都是运维来做。容器平台上线后，开发人员可以直接在程序完成之后将其制作成镜像，自己就可以进行部署。

当然任何事情不可能只有好的一面，容器平台也会带来相应的难度。容器平台对服务的调度具有动态性，这就需要监控系统能动态感知服务实例落在哪里，这是监控工作中的一个难点。

360 容器平台的监控维度

360 容器平台的监控系统对业务的技术指标进行监控，产品是从三个维度进行设计的：

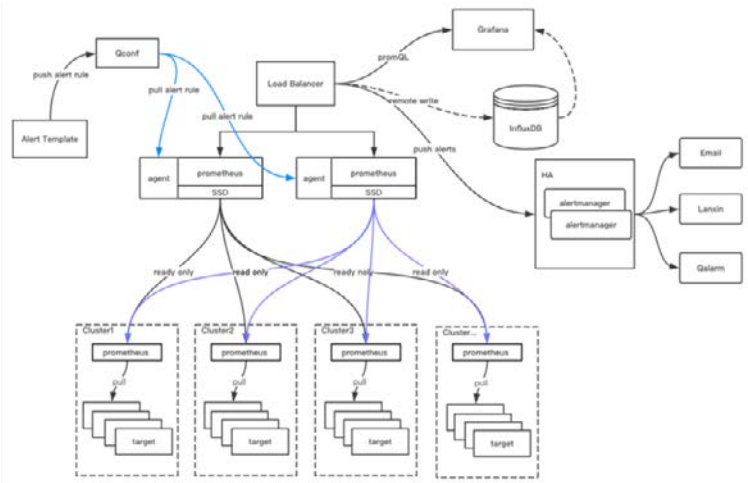
- 容器 : 这是最小的粒度。
- Pod: 一个 Pod 里面可以有多个容器。
- 应用 : 一个应用里可能会有多个 Pod。

除了这些技术指标监控，360 容器平台的监控系统还支持自定义监控。有些业务可能需要在自己的程序里打点，比如要调第三方接口，查看延时和调用的次数等。大平台上不同业务对数据的需求不同，为了达到业务需求，支持自定义监控还是很必要的。新开发的业务可以在自己的程序里引入 Prometheus 相关的 SDK 进行打点，360 的容器平台就会将这些 metrics 收集上来。

另外，没有使用 Prometheus 的老系统在开发时没有在程序里打点的功能，这时业务可以针对自己的需求以 sidecar 的方式在程序的边缘写 exporter 来采集该进程所有的监控数据，exporter 以 metrics 的形式报告给 Prometheus，Prometheus 进行抓取。

监控系统设计

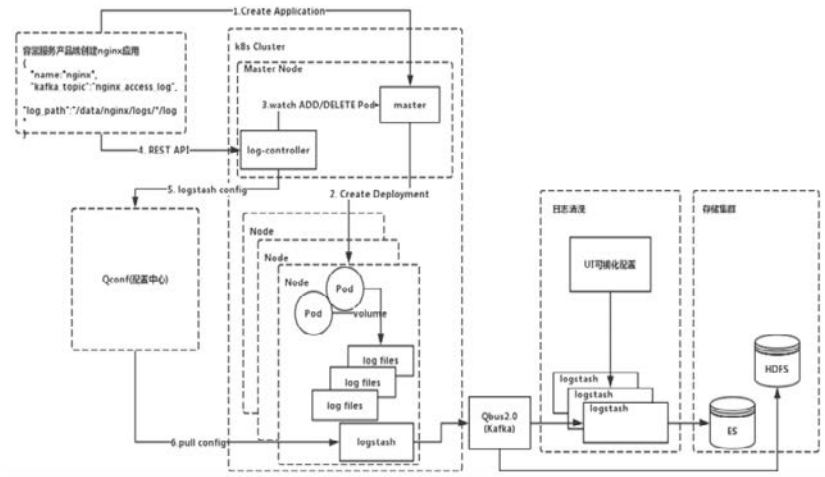
360 容器平台监控系统架构图如下：



日志监控

360 容器平台监控系统的日志监控使用的是 ELK 技术栈，但是进行了二次开发。团队自己写了 Log controller 组件，该组件会实时地 watch Kubernetes API Server 的 Deployment 资源对象的状态变化。当业务在创建应用时，是以 Kubernetes deployment 的资源对象来创建的。Log controller 会感知到这个资源的创建，知道这个 deployment 下有多少 Pod 已经处于 Ready 状态。当 Log controller 感知到新创建的 Pod 已经处于 Ready 状态以后，会根据用户在创建 Pod 时指定的真实日志收集路径拼接成它在物理机上的绝对路径，然后将组装好的配置并推送到 360 自研的配置中心 QConf 中。

360 在 Kubernetes 集群的每个 Node 节点都会部署一个二次开发的 Logstash，该 Logstash 会定期（每隔十秒或者五秒，时间可配置）去配置中心 (Qconf) 拉取最新的配置。拉取完配置之后，Logstash 会根据最新的配置把相关的的日志收集到公司的 Qbus（对 Kafka 进行包装的产品）中，也就是数据已经进入到 Kafka 了。之后用户可以在 HULK 私有云平台 (内部的私有云平台) 开通 ES 服务，对日志进行检索分析或对其他日志指定数据进行监控。



数据面板

360 容器平台监控系统的数据面板使用的是 Grafana，并对其增加了 LDAP 用户认证。可以显示内存，访问量，SIO, GPU 等监控数据指标。

应用级别基础监控指标 :

Pod 级别基础监控指标 :

容器级别基础监控指标 :

Prometheus 的基本原理是通过 HTTP 协议周期性抓取被监控组件的状态，任意组件只要提供对应的 HTTP 接口就可以接入监控。不需要任何 SDK 或者其他的集成过程。这样做非常适合做虚拟化环境监控系统，比如 VM、Docker、Kubernetes 等。输出被监控组件信息的 HTTP 接口被叫做 exporter 。目前常用的组件大部分都有 exporter 可以直接使用，比如 HAproxy、Nginx、MySQL、Linux 系统信息 (包括磁盘、内存、CPU、网络等等)。

报警

报警系统使用的是 Prometheus 自带的 Alertmanager 组件，但是对其进行二次开发，集成了 360 自己的报警系统 (Qalram)，并且可以通过 360 自己的即时通信工具（蓝信）实时接收报警信息，友好方便快捷。

360 容器平台监控系统选型对比

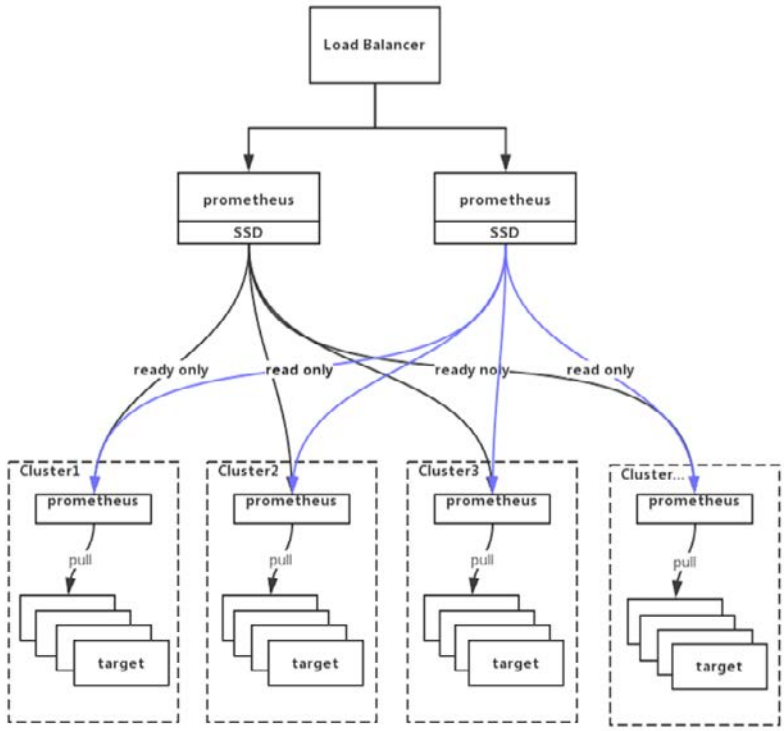
在构建容器平台的过程中，360 团队对监控系统的几种常用的开源方案进行过选型对比，主要调研了 K8s 社区的 Heapster（K8s 安装过程中默认安装的插件）和 Prometheus。

Heapster+InlfuxDB+Heapster 自带报警系统的方案有个缺点：它是针对 K8s 容器级别以及 Code 级别的技术指标监控，无法实现业务系统数据的监控，可扩展性不太友好，并且当未来数量级大时，不方便扩容。

最后团队选择了 Prometheus 云原生监控方案。虽然 Prometheus 在持久化方面做的还不够好，但是 Prometheus 更适合云原生生态。因为容器是动态变化的，微服务架构下一个实例的多个副本也是动态变化的，Prometheus 的 Pull 方式更适合动态变化的场景。

Prometheus 的应用实践

高可用



现在由于每个机房的机器数量不算太多，当前的设计方式是在每个机房部署一套 Prometheus 实例用于抓取目标的监控数据，同时在每个机房的上一层会部署两套 Prometheus 来做高可用 (HA). 这样上层的 Prometheus 只要去下层抓取关心的 metrics 就可以。这样做的好处 :

过滤掉上层不关心的监控数据，减轻数据存储的压力。

对各个机房的数据进行聚合，并统一对外提供统一的报警，视图查询接口。

由于 promethues 实例运行在本地，本地的磁盘空间有限，默认只保留最近 15 天的监控数据，但是想查询最近一个月，或者半年的数据，这个需求是做不到的。所以将 prometheus 的数据又入到了远程存储 InfluxDB 一份，用于数据的查询使用。

报警

Prometheus 的报警基于配置文件形式的。这种情况下，基础性指标可以产品化。360 团队定制了配置模版，用户只需要填具体的监控值就可以了。

针对业务监控，用户需要学习 Prometheus 的 promQL，这是由于不同的业务监控的需求不同，没有办法给业务统一的产品化服务，只能按业务自己制定报警规则去下发报警。

对于自定义业务，让业务自己写告警，统一的配置文件会上报到 QConf。

运行在两台 Prometheus 机器上 Agent 会实时的 watch 配置中心 (Qconf) 是否有新的配置生成，如果存在则会 Pull 报警规则，并 reload Prometheus 实例。

为什么要使用开源系统

一般公司都有自研的监控系统，大多数监控系统都是基于开源项目开发的。为什么 360 要使用 Prometheus ?

在 K8s 大规模使用的今天，整个容器云的生态推荐使用 Prometheus。

节省人力成本。公司开发一套监控系统需要投入人力，如果后期开发人员流失，后期的维护是个大问题。但是使用开源监控系统，在社区活跃的情况下项目也有保证，并且我们也会在使用过程中回馈社区。

360 容器平台监控系统演进方向

目前的监控系统整体架构对于容器平台来说可扩展性比较好。未来随着集群规模不断扩大，单个集群只有一个 Prometheus 实例要处理的 Job 数量爆发时，Prometheus 的性能就会变差。这个时候可以针对不同的任务类型，在一个集群里部署多个 Prometheus，每一个 Prometheus 只关心自己的任务，可控性会大大提升，同时整个架构的横向扩展比较方便。

另外 Prometheus 存在持久化的问题，未来还需要对 Prometheus 进行

数据远程存储（在上面的架构图中使用的虚线标注部分）。

除了数据，360 也在探索 AIOps，可以对监控系统收集到的数据进行处理，探索故障定位和根因分析等。

普通团队如何构建监控系统？

360 监控团队认为，公司的监控系统要根据公司的规模和业务场景来定，不可能一套方案直接拿来就直接使用。

对于使用公有云并且规模较小的公司，公有云本身就有监控系统。如果公司自己有服务器，但是量不大，简单搭建一个小型 HA 的监控系统就完全可以满足监控的需求了。架构的演变是随着业务规模的增长而不断的演变改进的。这时就需要根据具体的情况，去选择适合自己平台的架构方案。

作者简介

王希刚，目前在 360 做运维开发工作，有着多年 PaaS 设计及开发经验，对 Kubernetes，Docker 等有较深入的研究。目前主要负责 360 HULK 容器云平台的研发工作。

专访 Zipkin 项目 Leader：如何用 Zipkin 做好分布式追踪？

作者 姜宁

现代微服务架构由于业务系统模型日趋复杂，分布式系统中需要一套链路追踪系统来帮助我们理解系统行为，明确服务间调用。最近作者请到了 Zipkin 项目的主要开发维护人员 Adrian Cole 来介绍有关 Zipkin 项目的细节内容，可以让大家了解到如何在分布式追踪系统中用好 Zipkin。

Adrian 一直在从事云计算相关开源项目的开发，是开源项目 Apache jclouds 和 OpenFeign 的创始人。最近几年，他专注于分布式跟踪领域，是 OpenZipkin 项目的主要开发维护人员。Adrian 目前在 Pivotal Spring Cloud OSS 团队工作。在加入 Pivotal 之前，他还在 Twitter，Square，Netflix 工作过。

总所周知 Zipkin 项目起源于 Twitter，您能给我们介绍一下项目的相关背景吗？

Adrian：Zipkin 是由 Twitter 内部构建的分布式追踪项目，于 2012 年

开源。它最初被称为 BigBrotherBird，因此即使在今天，http 标头也被命名为“B3”。Twitter 早期因为业务发展迅猛，经常会出现系统过载情况。每当出现这种情况时，用户会看到一条搁浅的鲸鱼显示在页面上。Zipkin 这个名字也与鲸鱼有关，因为 Zipkin 是鱼叉的意思，构建初衷是为了追踪“搁浅的鲸鱼”相关的系统延迟问题。Zipkin 的成功除了和 Twitter 的的品牌加持有关，其他因素也起到很大作用：一是 Zipkin 包含了客户端、服务器、用户界面等所有你需要的部分；此外，Twitter Engineering 上一篇介绍 Zipkin 的博客也引起了轰动。随着时间的推移，Zipkin 成为了数十种可观察性工具之一。

分布式调用追踪对微服务监控有什么价值，为什么大家要为微服务建这样的追踪系统？

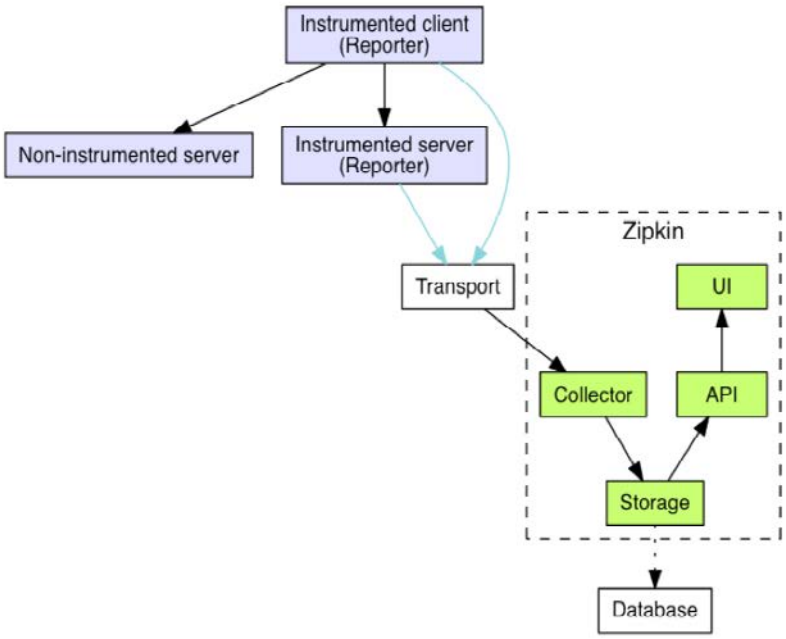
Adrian：这里我想引用 Netflix 的局点文档（Zipkin 的用户使用报告）中的描述来总结分布式追踪系统的价值：

其商业价值在于为系统提供了一个操作层面的可视化界面，同时提升开发人员的生产效率。

对于微服务系统来说，如果没有分布式追踪工具的帮助，要了解各个服务之间的调用关系是相当困难的。即使对于非常资深的工程师来说，如果只依靠追踪日志，分析系统监控信息，也是很难快速定位和发现微服务系统问题的。分布式调用追踪系统可以帮助开发人员或系统运维人员及时了解应用程序及其底层服务的执行情况，以识别和解决性能问题或者是发现错误的根本原因。分布式追踪系统可以在请求通过应用系统时提供端到端的视图，并显示应用程序底层组件相关调用关系，从而帮助开发人员分析和调试生产环境下的分布式应用程序。

在业界已经大量采用微服务架构的今天，Zipkin 在工业界微服务系统问题追踪方面扮演了很重要的角色，您能给我们介绍一下业界是如何使用 Zipkin 项目的吗？

Adrian：在回答这个问题之前，我们先简单介绍一下 Zipkin 的系统架构。Zipkin 包含了前端收集以及后台存储展示两部分。为了追踪应用的调用情况，我们需要在应用内部设置相关的追踪器来记录调用执行的时间以及调用操作相关的元数据信息。一般来说这些追踪器都是植入到应用框架内部的，用户应用程序基本感知不到它的存在。如下图所示当被监控的客户端向被监控的服务器端发送消息时，被监控客户端和被监控服务器端的追踪器会分别生成一个叫做 Span 的信息通过报告器（Report）发送到 Zipkin 后台。一次跨多个服务的调用会包含多个 Span 信息，这些 Span 信息是通过客户端与服务器之间传输的消息头进行关联的。后台会通过收集器（collector）接收 Span 消息，并进行相关分析关联，然后将数据通过存储模块存储起来供 UI 展示。



Zipkin 在项目开源之初就包含了一个完整的追踪解决方案方便大家上手，同时通过传输协议以及相关消息头开源，帮助 Zipkin 很容集地成到用户的监控系统中。这让 Zipkin 能够在众多的追踪系统中脱颖而出，成为工业界最常用的追踪工具。

在 2015 年的时候我们发现不是所有人都有分布调用追踪的系统使用

经验，他们不太了解如何成功搭建分布式追踪系统。因此我们整理了很多业界使用 Zipkin 的相关资料，并在 Google Drive 上创建了相关文档目录存放这些资料。考虑到大家访问 Google Drive 可能会不太方便，后来我们创建了 wiki 来分享如何使用 Zipkin，让更多人了解其他人是如何使用 Zipkin 来解决追踪系统中存在的问题的。

在这里你可以看到像 Netflix 这样的大厂是如何使用 Zipkin 以及 Elasticsearch 处理每天高达 5TB 的追踪数据，Ascend Money 是如何将公有云和私有云结合在一起使用 Zipkin，Line 是如何将 Zipkin 与 armeria 结合进行异步调用追踪的，以及 Sound Cloud 是如何结合 Kubernetes Pod 的元数据信息改善 Zipkin 追踪的。这里我们也欢迎更多的中国用户能够在此分享你们使用 Zipkin 的经验。

对了，如果你是 Zipkin 用户，记着给我们的代码仓库加星，这是对我们最大的鼓励。

我们知道 Zipkin 项目目前在生产环境中大量使用，他们是直接使用这些项目还是依据自己的需要对项目进行了相关的修改？

Adrian：因为 Zipkin 项目运作是建立在用户吃自己的狗粮（使用自己开发的软件）的基础上，很多新工具都是先在用户内部使用，然后开源变成通用项目。例如 zipkin-forwarder 这个项目就是 Ascend 结合自己的业务需要实现的跨多个数据中心转发数据实验性项目。Yelp 实现了一个 Zipkin 代理用来读取多个 Zipkin 集群的数据。LINE 在内部开发了一个代号为“project lens”的项目来替换 Zipkin UI。

现在 Zipkin 的使用案例很多，不同使用案例对应的架构也不相同。这并不是说用户没有直接采用 OpenZipkin 组织下面的项目。例如 Mediata 就在他们的局点文档中描述了他们直接使用 Zipkin 的发行版，没有进行任何修改。

Zipkin 项目缺省是支持中等规模的使用场景，为了支持更大规模的使用场景，我们需要做些什么？

Adrian：大规模的使用场景意味着更多的数据量，大规模用户通常会从成本以及处理或清理数据的能力来考虑这个问题。例如，SoundCloud 实现了基于 kubernetes 元数据清理数据的工具。以往几乎所有 Zipkin 大型局点都将追踪数据存储在 Cassandra。现在，Netflix 等大型局点采用 Elasticsearch 也能取得成功。几乎所有大型局点都使用 Kafka 来进行更大的数据传输。出于效率的原因，较小的站点会采用不同的传输方式。例如，Infostellar 使用 Armeria（一个类似于 gRPC 的异步效 RPC 库）来传输数据。

大多数成熟局点也会担心数据大小而使用采样的方式来解决问题。因为我们很难为不同类型的局点设计一个通用工具，Zipkin 为大家提供了多种采样选择，其中包括基于 http 请求的表达式来进行配置，也有新的开发客户端速率限制采样器。例如，使用 Spring 的局点有时会使用配置服务器实时推送采样率。这样可以在不干扰通常的低采样率的情况下抓取有趣的痕迹。

大多数局点使用客户端采样来避免启动无用的跟踪。例如，spring cloud sleuth 的默认策略不会为健康检查等管理流量创建跟踪。这些是通过 http 路径表达式完成的。一些基于百分比的采样（例如 1％或更少的流量）在数据激增时仍然存在问题。即使采样率为 1％，当流量达到 1000％的时候也可能会出现问题。因此，我们始终建议对系统进行冗余配置，也就是说不仅要为正常流量分配带宽和存储空间，还要提供一定的余量。但这不是一个最好的解决方案，我们开玩笑的将其称为 OPP （over-provision and pray，美国流行歌曲名），通过冗余保障还有祈祷来帮我们应对数据激增的问题。

值得注意的是许多局点站并不打算“演进”到自适应（自动）采样。例如，Yelp 在定制的无索引的廉价存储中进行 100％的数据消费。这样可以在不干扰通常的低采样率的情况下直接抓取有趣的痕迹。在 Zipkin 社区中有相关自适应采样配置的设计讨论，以突出问题区域而不会增加基本采样率的复杂性。自适应是一个有趣的选择，绝对可以应对数据激增问题，但大家通常会用不同的方法来实现自适应。

目前 Zipkin 在线局点日常处理 span 的数据流大概是多少？ Zipkin 缺省能支持存储一个月的数据吗？在 Zipkin 的存储方面你有什么好的建议？

Adrian：Yelp 处理的 Span 数据可能是最多的，因为它们会将 100％的数据集中到专用存储群集中。但是他们也没有公布数字。Netflix 的系统

大约会将 240 MB /s 的 span 数据推送到后台（大家可以算算这里有多少条 span 信息），一般来说一条 Span 数据通常不会超过 1KiB，甚至远远低于 1KiB。

关于数据存储，大多数网站出于成本的考虑只保留数天的数据。不过有些用户自己提供“最喜欢的追踪”功能来长时间保留追踪数据。关于 Span 大小，我们建议大家使用 brave 或 zipkin-go 等工具中自带的默认值，然后根据大家的具体需求添加一些数据标签（决定是否保留数据）来提升资源利用率。

我们也不建议把 span 作为日志记录器。 OpenTracing 开了这个坏头，他们把日志工具和分布式追踪 API 相混淆，甚至在 Span 中的 API 定义了“microlog”。据我所知没有一个追踪系统能在系统内部把常规日志记录工作做得很好的，因为追踪系统和常规日志记录系统是两个完全不同的系统，这样做只会损害追踪系统运行效率。

我们建议大家根据自己的需要选择最佳的存储方式。例如，Infostellar 直接将追踪数据转发到 Google Stackdriver 上进行存储。如果你更熟悉 Elasticsearch 而不是 Cassandra，那么请使用 Elasticsearch，反之亦然。但是我们建议大家不要使用 MySQL，因为我们的 MySQL 架构不是为了高性能而编写的。

Zipkin 是如何对接分析系统（如 Amazon X-Ray, Apache SkyWalking 以及 Expedia HayStack）的？对此你有什么好的建议？

Adrian：你提到的所有项目都是通过接收 Zipkin 数据实现集成的，Infostellar 在这方面有比较好的网站文档可以参考。

Skywalking 也提供了接入 Zipkin 数据的集成方式。 Skywalking 可以接受 Zipkin 格式的数据，这样无论是 Zipkin 的探针，还是其他工具，如 Jaeger，只要使用相同格式的工具都可以接入到 Skywalking 中。需要指出的是现在很多追踪工具都支持 Zipkin 格式，通过支持 Zipkin 格式可以很容易完成对其他追踪系统的集成工作。

我对 Haystack 与 Zipkin 集成工作的有一定的了解。他们通过 Hotels.com 提供的 Pitchfork 这个工具，将数据分别发送给 Zipkin 和 Haystack。Haystack 的系统可以对 Zipkin 数据进行服务图聚合等处理，这样就不需要使用 Zipkin 的 UI 来处理数据了。

OpenZipkin 是什么？这些年 Zipkin 社区是怎么发展起来的？如何加入到 Zipkin 社区中？

Adrian：2015 年，社区有人呼吁把项目迁移到一个更开放的地方，

以便更快地发展项目。我们经过三个月的努力，于 2015 年 7 月在 Github 上成立了“OpenZipkin”小组。社区在此之后快速发展，大量用户在社区中交流他们在构建分布式追踪系统所遇到的问题和挑战。当社区决定把首选语言定为 Java 而不是 Scala 后，我们重新编写了服务器。我们有许多示例项目帮助大家上手，而且我们认为与他人交流是最好的学习方式。如果你不熟悉追踪，最容易参与的方法就是参与到我们的 gitter 讨论中来。

Zipkin 最近加入 ASF 孵化器，这对于 Zipkin 来说意味着什么？

Adrian：Zipkin 发展了一段时间后，CNCF 联系我们加入，而我们考虑的是加入阿帕奇软件基金会（ASF) 或者什么也不加入。当时的社区看不到加入基金会的好处，我们当时选择了什么不加入基金会。然而，社区不断发展壮大，我们也有责任成长。特别是当我们通过 SkyWalking 的在 ASF 孵化对 ASF 有了更深入的了解。因为基金会要求旗下项目站在厂商中立的角度来考虑问题的，这样可以帮助 Zipkin 站在社区的角度上考虑如何独立发展。ASF 的文化和我们也更匹配，因此 Zipkin 于今年 8 月份刚成为 Apache 孵化器项目。

作者简介

姜宁，华为开源能力中心技术专家，前红帽软件首席软件工程师，Apache 软件基金会 Member，有十余年企业级开源中间件开发经验，有丰富的 Java 开发和使用经验，函数式编程爱好者。从 2006 年开始一直从事 Apache 软基金会开源中间件项目的开发工作，先后参与 Apache CXF，Apache Camel，Apache ServiceMix，Apache ServiceComb 的开发。对微服务架构，WebServices，Enterprise Integration Pattern，SOA， OSGi 有比较深入的研究。

微博 ID：https://weibo.com/willemjiang

个人博客地址：https://willemjiang.github.io/

微服务架构下的监控需要注意哪些方面？

作者 张婵

微服务架构虽然诞生的时间并不长，却因为适应现今互联网的高速发展和敏捷、DevOps 等文化而受到很多企业的推崇。微服务架构在带来灵活性、扩展性、伸缩性以及高可用性等优点的同时，其复杂性也给运维工作中最重要的监控环节带来了很大的挑战：海量日志数据如何处理，服务如何追踪，如何高效定位故障缩短故障时常……

InfoQ 记者采访了京东云应用研发部门运维负责人，来谈一谈微服务架构下的监控应该注意哪些方面。

微服务架构带来的变化

在京东云运维专家看来，微服务架构给 IT 系统和团队带来了以下显著的变化：

- 基础设施的升级，需要引入虚拟化（如 Docker），现存基础设施也需要与之进行适配；
- 系统架构的升级，需要引入服务注册（如 Consul），服务间的交互方式也需要与之进行适配；
- 运维平台的升级，建议引入日志收集（如 Fluentd），分布式跟踪（如 Zipkin）和仪表盘（如 Vizceral/Grafana）等；
- 运维效率和自动化水平的提升也迫在眉睫，否则无法应对实例数量，变更频率，系统复杂度的快速增长；
- 观念的转变，基础设施，系统架构和运维平台等的大幅升级，犹如小米加步枪换成飞机大炮，相应的战略战术也需要与之相适配才行。

微服务架构下用户面临的监控问题

在转型到微服务架构以后，用户在监控方面主要会面临以下问题。

首先，监控配置的维护成本增加。某个在线系统大概有 106 个模块，每个模块都需要添加端口监控，进程监控，日志监控和自定义监控；不同服务的监控指标，聚合指标，报警阈值，报警依赖，报警接收人，策略级别，处理预案和备注说明也不完全相同；如此多的内容，如何确保是否有效，是否生效，是否完整无遗漏。

当前针对维护成本，业界常用的几种方法有：

- 通过变量的方式尽量减少人工输入；
- 通过监控配置文件解析做一些可标准化的校验；
- 通过故障演练验证报警是否符合预期。

其次，第三方依赖越来越多。例如 Docker 的可靠性很大程度上取决于宿主机，如果所在的宿主机发生资源争用，网络异常，硬件故障，修改内核参数，操作系统补丁升级等，都可能会让 Docker 莫名其妙地中招。

第三，服务故障的定位成本增加。假设故障是因为特定服务处理耗时增大导致的，那么如何快速从 106 个服务以及众多的第三方依赖中把它找出来，进一步，又如何确认是这个服务的单个实例还是部分实例的异常，这些都让故障定位变得更复杂。

在微服务架构下，提高故障定位效率的常用方法有：基于各类日志分析，通过仪表盘展示核心指标：数据流，异常的监控策略，变更内容，线上登录和操作记录，文件修改等内容。

微服务监控的难点及解决思路

在微服务架构下，监控系统在报警时效性不可改变的前提下，采集的指标数量是传统监控的三倍以上，如果是万台以上的规模，监控系统整体都面临非常大的压力，在监控方面的挑战主要来源于：

首先，存储功能的写入压力和可用性都面临巨大挑战。每秒写入几十万采集项并且需要保证 99.99% 的可用性，对于任何存储软件来讲，都不是一件轻松的事情。

对于写入和可用性的压力，业界常见的解决思路主要是基于如下方式的组合：

- 集群基于各种维度进行拆分（如地域维度、功能维度和产品维度等）；
- 增加缓存服务来降低 Hbase 的读写压力；
- 调整使用频率较低指标的采集周期；
- 通过批量写入降低 Hbase 的写入压力；
- 通过写入两套 Hbase 避免数据丢失并做到故障后快速切换。

其次，监控的生效速度也面临巨大挑战。微服务架构下，基于弹性伸缩的加持，从服务扩容或者迁移完毕到接入流量的耗时降低到 1Min 左右，且每时每刻都在不断发生着。对于复杂监控系统来讲，支持这样的变更频率绝非易事，而且实例变更如此频繁，对监控系统自身来讲，也会面临可用性的风险。

常见的提高监控生效速度的思路主要有如下的几种方式：

- 实时热加载服务注册信息；
- 对监控配置的合规性进行强校验；
- 增加实例数量的阈值保护；
- 支持配置的快速回滚。

第三，基础设施的故障可能导致报警风暴的发生。基础设施如 IDC 故障，可能会在瞬时产生海量报警，进而导致短信网关拥塞较长时间。

解决这类问题的思路主要是如下方式：

- 基于报警接收人通过延时发送进行合并；
- 报警策略添加依赖关系；
- 优先发送严重故障的报警短信；
- 增加多种报警通知方式如电话，IM 等。

微服务监控原则

对于采用微服务的团队，京东云运维专家建议在做监控时可以参考 Google SRE 的理论，结合自己长期的运维实践经验，他总结了几点可以参考的原则：

- 首先，所有系统和第三方依赖的核心功能必须添加黑盒监控；
- 第二，所有模块必须添加白盒监控的四个黄金指标（饱和度，错误，流量和延时）；
- 第三，所有的变更都需要进行采集，包括但不限于程序，配置，数据，网络，硬件，操作系统以及各类基础设施。

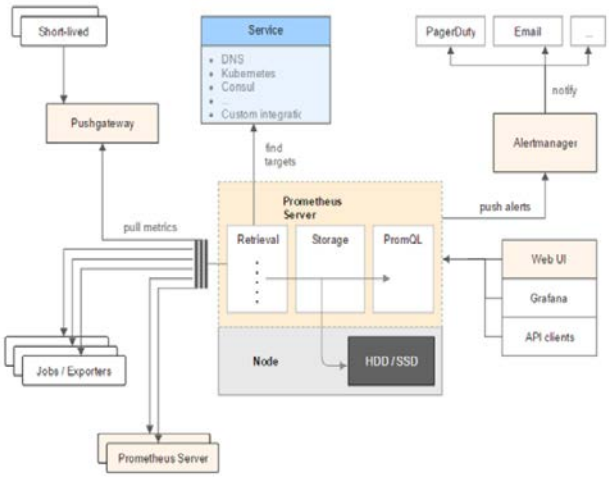
另外，京东云运维专家也给大家提供了一些黑盒监控的实施经验：

首先，应该监控哪些功能？建议将系统接入层的访问日志，TOP-10 的 URL 添加黑盒监控。那 TOP-9 的 URL 是否一定需要监控？ TOP-11 的 URL 是否一定不需要监控？这取决于其访问量是否和前面的 URL 在一个数量级以及人工评估其接口的重要性程度，这里提供的更多是一个思路，而非可量化的方法。

第二，应该使用多少个样本 / 节点对一个功能进行黑盒监控？建议样本应该覆盖到对应模块的所有实例，这样也能发现由少数实例导致的小规模故障。

微服务架构下的理想监控系统

从用户的角度看，Prometheus 的整体架构设计参考了 Google BorgMon，系统具有高度的灵活性，围绕其开放性现在也慢慢形成了一个生态系统。具体来说，Prometheus 使用的是 Pull 模型，Prometheus Server 通过 HTTP 的 Pull 方式到各个目标拉取监控数据。HTTP 协议的支持能够更加方便的进行定制化开发，服务注册、信息采集和数据展示均支持多种形式 / 开源软件。



结合目前国内正在兴起的智能运维，也许不久的将来，上面提到的监控的各种问题也就迎刃而解了。监控策略不在需要人工定义，转由机器学习负责，诸如策略添加，阈值设定，异常检测，故障定位，自动止损等逐步由系统负责，运维人员不再是“救火队长”。

京东云监控响应实践

京东云运维平台为数万台机器提供监控，部署，机器管理，权限管理，安全管理，审计和运营分析等功能，为京东云所有的业务在各类异构网络环境下提供标准和统一的运维支撑能力。

基于产品所处的发展阶段，用户规模的不同，报警频率也不尽相同。理想情况下，报警频率应该等同于故障频率，这里面体现了报警的准确度和召回率两个指标，如果每个报警都对应一个服务故障，则准确度为100%，同理，如果每次服务故障均有报警产生，则召回率为 100%。大家可以基于上述两个指标，来衡量自己团队的现状，并针对性的制定提升计划即可。

对于响应流程，京东云有几个做的好的地方可以给大家参考。

- 首先，所有核心报警均有可靠的应对预案和处理机制，并通过定期的破坏演练持续进行完善。
- 其次，公司的监控中心会 7x24 值守，他们也会和业务线运维同学一样，接收所有影响核心系统稳定性的报警，收到报警后会进行通报，确保核心报警在发生后第一时间内有人处理并在规定的时间内处理完毕。如果未在规定的时间内处理完毕，监控中心会进行报警升级，通报该系统的管理人员，从而确保该报警可以得到更高的重视度和支持力度。

结语

对于监控系统的未来发展，京东云的运维专家认为长期来看，依托于 Kubernetes 的发展，在基础设施的各个领域，都会从百花齐放到几家独大，从而将标准化落地到基础设施的各个领域，进而促进整个生态的繁荣。

在监控方向，Prometheus 在未来一段时间后，也许会是一个很好的选择。在 Prometheus 等工具解决了通用的监控场景并标准化之后，在其上的各类应用场景，如容量规划，流量监控，故障定位以及各种基于大数据和人工智能场景的落地等，就会出现百花齐放之势。

版权声明

InfoQ 中文站出品

出版 : 北京极客邦科技有限公司
北京市朝阳区来广营叶青大厦北园 5 层

欢迎共同参与 InfoQ 中文站的内容建设工作，包括原创投稿和翻译，请联系 editors@geekbang.com。
网址 : www.infoq.cn

扫码关注InfoQ公众号

Geekbang. 极客邦科技 | InfoQ