

Apollo+ES源码改造，构建民生银行的ELK日志平台配置管理中心

原创 高效开发运维 架构头条 2019-02-28



作者 | 中国民生银行大数据基础平台运维组团队

编辑 | 张婵

随着 IT 业务系统的迅速发展，中国民生银行需要考虑实现一套完整并适用于民生银行的日志文件智能分析与处理的解决方案。本文详细介绍了中国民生银行大数据基础平台运维组团队通过改造 Apollo 和 ES 的源码，构建了自己的天眼实时智能日志管理分析平台。

背景

随着中国民生银行的 IT 业务系统的迅速发展，主机、设备、系统、应用软件数量不断增多，业务资源访问、操作量不断增加，对于应用整体系统智能分析与处理的要求不断提高，在解

解决分布式大数据搜索、日志挖掘和可视化的过程中，需要考虑实现一套完整并适用于民生银行的日志文件智能分析与处理的解决方案。

基于此天眼实时智能日志管理分析平台应运而生，通过日志的收集、传输、储存，对海量系统日志进行集中管理和实时搜索分析，帮助运维人员进行业务实时监控、故障定位及排除、业务趋势分析、安全与合规审计等工作，深度挖掘日志的大数据价值，提升了应用整体系统智能分析与处理效率，达到了汇总、检索、展示应用日志和串联事件、快速定位问题等全方位功能要求。

目前日志平台纳管的服务器超过 1000 台，覆盖了民生银行所有操作系统类型：SuSE Linux (11/12)、AIX (7)、HP_UX (11) 和 RedHat (5.5)，除了应用日志以外，系统软件日志类型覆盖 DB2、Oracle、Mysql、Redis、Weblogic、Activemq、Kafka、Tomcat 等等，同时采集存储、操作系统、管理口相关指标日志。每天接入日志量在 10T 到 20T 之间，平均日接入量在 15T 左右。

在此如此量级的日志接入下，对平台本身吞吐量、端到端全链路的写入延迟、查询响应时间等都提出了比较高的要求，因此对于 ES 集群本身的参数调优成为了一项持续进行的长期性工作，这时 ES 物理节点过多导致的配置文件分散、角色参数差异、版本管理混乱、配置监控缺失等问题就集中暴露了出来。而使用脚本或者文件分发管理又不够直观和友好，出现问题排查困难，易用性较差，也不具备分布式和高可用功能。

为了能够界面化、集中化管理 ES 集群不同角色、不同类型的配置并且在配置修改后能够在 ES 中实时生效，所有配置信息的修改具备规范的权限、流程治理等特性，民生银行天眼 ELK 日志平台最终采用携程开源的 Apollo（阿波罗）作为技术选型，本篇以民生银行天眼日志平台实际需求为中心，逐步展开介绍我们是如何通过 Apollo+ES 源码改造构建民生银行天眼 ELK 日志平台配置管理中心的。

Apollo 功能介绍

Apollo 目前在 GitHub 的 Star 数量超过 10000，社区活跃度和版本更新效率都比较高，首先简单介绍一下 Apollo 本身的功能点和其在天眼 ELK 日志平台配置管理中心中的实际运用。

Apollo 是携程的开源配置管理中心，可以从应用、环境、集群、命名空间 4 个维度集中管理配置，并能够实时推送至客户端，优点如下：

统一管理不同环境、不同集群的配置

Apollo 提供了一个统一界面集中式管理不同环境（environment）、不同集群（cluster）、不同命名空间（namespace）的配置。同一份代码部署在不同的集群，可以有不同的配置，通过命名空间（namespace）可以很方便地支持多个不同应用共享同一份配置，同时还允许应用对共享的配置进行覆盖。

由于民生银行开发、测试、生产环境均是网络隔离的，所以在进行 Apollo 部署时我们去掉了 4 个环境标识 DEV、FAT、UAT、PRO 中的后面 3 个，只保留了 DEV 环境，测试和生产均完整的部署一套 Apollo 环境。另外运用集群（cluster）实现了 ES 角色配置文件分离，运用 namespace 实现了 ES 不同种类配置文件分析，这些在后面 Apollo+ES 设计中会详细讲到。

配置修改实时生效（热发布）

用户在 Apollo 修改完配置并发布后，客户端能实时（1 秒）接收到最新的配置，并通知到应用程序。

这个功能比较诱人，我们在其它项目中也使用到了热发布，但是在 ES 中没有配置，一方面是由于 ES 读取的很多配置的模式都是读且仅读一次，热配置无法生效。另一方面 ES 代码较复杂，其在读取配置文件前后都需要初始化很多参数，很多配置参数是扩散到整个项目代码中的，核心代码、插件代码可能都会有涉及，一些如监听端口配置修改需要重起相关线程模块，实现热配置风险太高，最后热配置在 ES 源码改造过程中我们只是用于配置参数变化的日志输出打印，完成配置生效还是需要重起节点进程。

版本发布管理

所有的配置发布都有版本概念，从而可以方便地支持配置的回滚。

这个是配置中心基本功能，在日志平台进行 ES 参数优化我们依赖 Apollo 进行版本管理，使用起来比较方便。

灰度发布

支持配置的灰度发布，比如点了发布后，只对部分应用实例生效，等观察一段时间没问题后再推给所有应用实例。

在日志平台实际应用中我们经常使用灰度发布的功能去验证某些通用参数是否可以配置在角色节点或者数据节点可以单独生效，在 default 集群参数中让某些参数的发布推送到特定的

master 节点或者 data 节点，不过最后根据默认参数全部配置一致原则会将 default 中修改过的参数配置推送到全部的 ES 节点当中。

权限管理、发布审核、操作审计

应用和配置的管理都有完善的权限管理机制，对配置的管理还分为了编辑和发布两个环节，从而减少人为的错误。所有的操作都有审计日志，可以方便地追踪问题。

在权限控制上生产环境要求必须日志平台 A 岗负责人有最终的发布权限，ES 参数修改对集群整个的稳定性影响比较大，必须将风险降到最低。另外目前日志平台配置管理中心也承担着一些项目组自己开发的工具的配置管理工作，所以单独也进行了权限的划分和处理。

客户端配置信息监控

可以在界面上方便地看到配置在被哪些实例使用。

这个功能在日志平台中某种程度上也起到了实例心跳检测的功能，同时根据角色的划分可以清楚地看到这个 ES 的逻辑节点配置位置，比如 default 参数肯定是所有节点都会读取，而单独的 master、hot、warm、client 节点只有相关的角色节点才会读取其配置。值得一提的是由于有缓存的功能，可以看到 ES 相关节点读取配置的时间实际上是参数改变后节点第一次读取配置的时间，如果 ES 节点单独重启但是相关参数没有发生变化和发布，那么界面上看到的配置参数读取时间是不会发生改变的。

提供 Java 和 .Net 原生客户端

提供了 Java 和 .Net 的原生客户端，方便应用集成，同时提供了 Http 接口，非 Java 和 .Net 应用也可以方便地使用。

ES 本身就是通过 Java 语言编写的，所以 **通过修改 ES 源码可以比较方便地将 Apollo 客户端集成到 ES 中去**。后续我们打算将 Logstash 处理的配置文件也通过 Apollo 纳管，由于 Logstash 是用 ruby 语言，所以可能就会用到其提供的 Http 接口。

提供开放平台 API

Apollo 自身提供了比较完善的统一配置管理界面，支持多环境、多数据中心配置管理、权限、流程治理等特性。不过 Apollo 出于通用性考虑，不会对配置的修改做过多限制，只要符合基本的格式就能保存，不会针对不同的配置值进行针对性的校验，如数据库用户名、密

码，Redis 服务地址等。对于这类应用配置，Apollo 支持应用方通过开放平台 API 在 Apollo 进行配置的修改和发布，并且具备完善的授权和权限控制。

目前天眼 ELK 日志平台主要是还是使用 Apollo 本身提供的配置管理界面进行管理，不过后续可以考虑通过其 API 将部分功能在大数据集中的管控平台上去实现。

部署简单

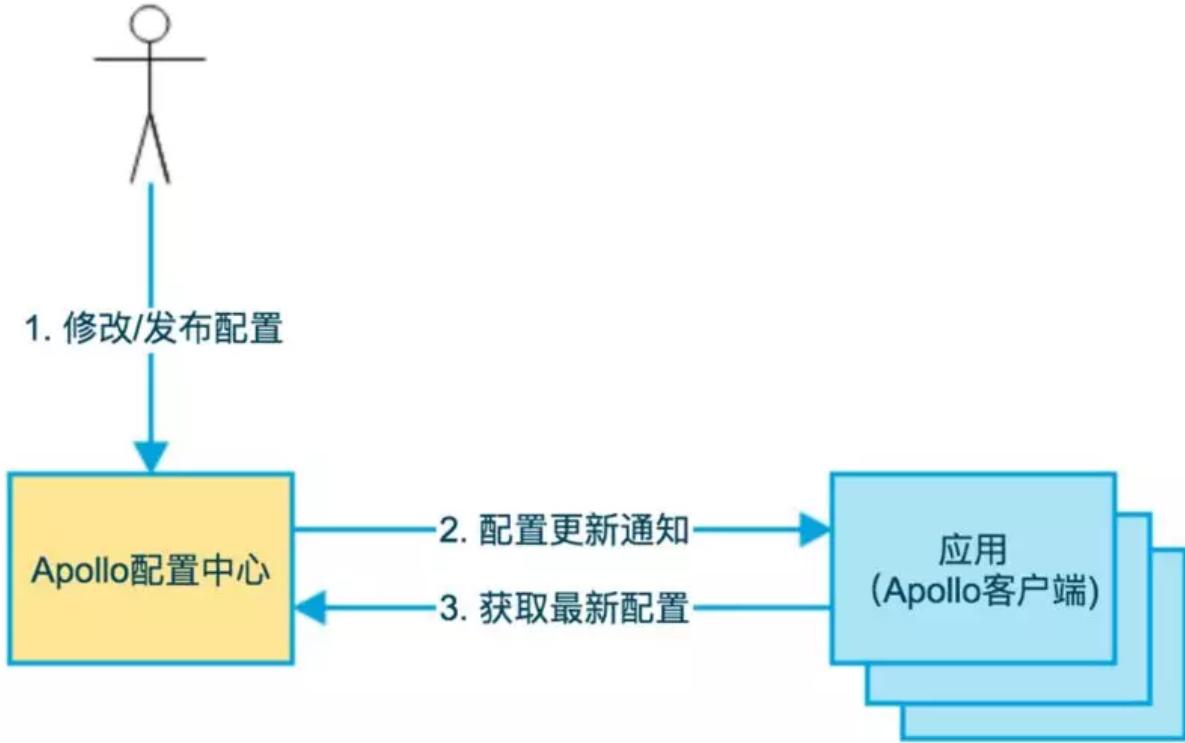
配置中心作为基础服务，可用性要求非常高，这就要求 Apollo 对外部依赖尽可能地少，目前唯一的外部依赖是 MySQL，所以部署非常简单，只要安装好 Java 和 MySQL 就可以让 Apollo 跑起来。Apollo 还提供了打包脚本，一键就可以生成所有需要的安装包，并且支持自定义运行时参数。

日志平台基本按照官方文档进行标准部署，稍微有一点改动的地方是 Apollo 本身的日志路径存放地址是以进程号为目录层级的，不太直观，通过修改源码将其增加 `adminservice`、`configservice` 和 `portal` 目录，使得不同模块的日志路径比较清晰易区分。

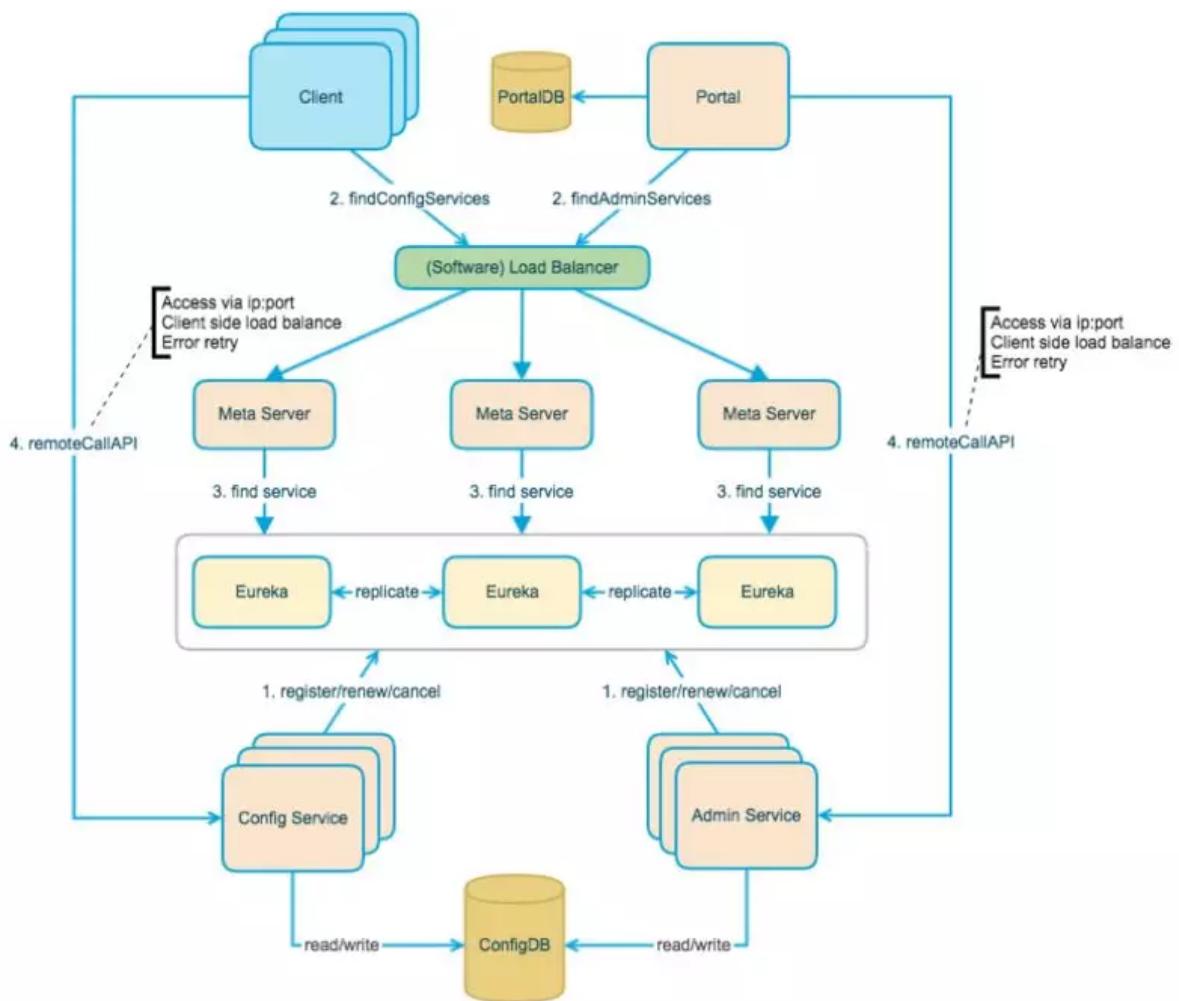
Apollo 架构解析

在日志平台生产实际使用 Apollo 之前，必须对其架构进行深入了解和剖析，在架构设计的过程中有一个基本点必须明确，那就是 Apollo 的某个服务异常不应该影响整个配置管理中心的服务，Apollo 所有模块的异常不应该影响 ES 集群的正常运行。

Apollo 的基础模型如下：



如上图所示，用户在配置中心配置 / 修改并发布配置，然后 Apollo 通知客户端有配置更新，最后客户端向 Apollo 配置中心请求最新配置。总体设计如下图所示：



其中 Apollo 各模块的功能简单介绍如下：

- config Service： 提供配置获取接口， 提供配置更新推送接口， 接口服务对象为 Apollo 客户端；
- Admin Service： 提供配置管理接口， 提供配置修改、发布等接口， 接口服务对象为 Portal；
- Meta Server： Portal 通过访问 Meta Server 获取 Admin Service 服务列表；
(IP+Port) , Client 通过访问 Meta Server 获取 Config Service 服务列表；

(IP+Port) , Meta Server 从 Eureka 获取 Config Service 和 Admin Service 的服务信息， 相当于是一个 Eureka Client， 增设一个 Meta Server 的角色主要是为了封装服务发现的细节， 对 Portal 和 Client 而言， 永远通过一个 Http 接口获取 Admin Service 和 Config Service 的服务信息， 而不需要关心背后实际的服务注册和发现组件， Meta Server 只是一个逻辑角色，在部署时和 Config Service 是在一个 JVM 进程中的；

- Eureka： 基于 Eureka 和 Spring Cloud Netflix 提供服务注册和发现， Config Service 和 Admin Service 会向 Eureka 注册服务，并保持心跳，为了简单起见，目前 Eureka 在部署时和 Config Service 是在一个 JVM 进程中的；
- Portal： 提供 Web 界面供用户管理配置，通过 Meta Server 获取 Admin Service 服务列表 (IP+Port) ， 通过 IP+Port 访问服务，在 Portal 侧做 load balance、错误重试；
- Client： Apollo 提供的客户端程序，为应用提供配置获取、实时更新等功能，通过 Meta Server 获取 Config Service 服务列表 (IP+Port) ， 通过 IP+Port 访问服务，在 Client 侧做 load balance、错误重试。

关于 Apollo 各个组件的相关作用和实现原理不是本文的重点，我就不再这里过多赘述了，感兴趣的同学可以去 Apollo 的 github 进行详细了解。

在实际部署中我们是将 Apollo 部署在三台服务器上做高可用和负载均衡，每台服务器上都会启动独立的 config Service、Admin Service 和 Portal 三个模块进程，后端连接统一的 mysql 数据库，通过修改数据库中 serverconfig 字段使得 Eureka 注册服务构成三节点的集群，client 端和 portal 统一配置 3 台服务 ip 地址进行连接。

下面就开始介绍我们构建天眼 ELK 日志管理平台的两块核心工作：Apollo ES 架构设计和 ES 源码改造。

Apollo ES 架构设计

在 Apollo ES 架构设计这块其实包含了两部分工作：

1. 一部分工作是通过 Apollo 的集群分离功能将通过 ES 中的角色将配置进行拆分，方便统一管理；
2. 另一部分工作是根据实际需求将 elasticsearch 需要频繁修改的配置文件进行分类并分别实现客户端开发。

这两部分工作都包含了一个共有的工作内容——页面配置设计实现以增加功能使用体验。

ES 角色配置拆分

日志平台目前在生产环境使用的 ELK 版本是 5.5 的，在进行 ES 集群设计部署时 data 节点采用了冷热数据分离技术，引入了 SSD 来提升 ES 的读写性能。单台 ES 存储有 2 块 SD 盘和若干 SATA 盘，所以每台 ES server 都启动了 3 个 ES 节点，2 个 hot 节点和 1 个 warm 节点。Indexer 中指配置了 hot 节点的端口，通过 ES 中的模板定义保证实时数据只写入 hot 节点。通过 ES 官方推荐的 curator 工具定时将数据从 hot 节点搬迁到 warm 节点，SSD 数据保留周期为一周。同时本身 ES 集群我们启动了 3 个 master 节点和 3 个 client 节点，那么实际上 ES 集群中一共是 4 种：master, client, hot 和 warm，节点类型都是通过配置文件 elasticsearch.yml 的参数进行区分的。

我们知道 ES 主要的配置参数都是在 elasticsearch.yml 中进行定义的，最开始设计时我们计划是通过主机名的方式进行配置文件区分，Apollo 本身也提供了通过 hostname 定义不同的 namespace 的内置方法，但是最后达到的效果非常不理想。日志平台生产集群物理节点目前是 28 个，总节点数为 90 个，当需要修改参数时需要每个节点的 namespace 配置都在界面上进行修改并发布，重复工作太大；另外 namespace 过多也导致管理页面特别冗长，下拉很久也拖不到页面底部，用户体验很差，当时就想这样还不如直接脚本分发修改配置文件来的快些，失去了日志平台配置管理中心建设的初衷。

于是就想能不能通过角色进行配置拆分，相同的角色的配置是否可以放到一个 namespace 中去处理。Apollo 本身在 DEV 的生产模式下提供创建多集群的配置方式，由于在日志平台

中不存在多集群的情况，于是我们就将集群模式的功能来区分 ES 中的节点角色。最终在角色中实际上创建了 5 种角色参数分类：default、hot、warm、master 和 client。

The screenshot shows the Apollo configuration center interface. On the left, there's a sidebar with '环境列表' (Environment List) containing 'default' (selected), 'hot', 'warm', 'master', and 'client'. Below it is '项目信息' (Project Information) with fields like AppId: elasticsearch, 项目名: elasticsearch, 部门: 系统管理部, 负责人: 张三, 邮箱: zhangsan@***.com. The main area has tabs for 'application' and 'properties'. Under 'application', there are sections for 'elasticsearch-pool-yml' and 'jvm-pool-options'. Under 'elasticsearch-pool-yml', there are two rows of configuration: 'elasticsearch-pool-yml' with key 'ES-POOL-YML' and value 'elasticsearch-pool-yml'; and 'jvm-pool-options' with key 'JVM-PUBLIC' and value 'jvm-pool-options'. Both rows have columns for '备注' (Remarks), '最后修改人' (Last Modified By), '最后修改时间' (Last Modified Time), and '操作' (Operations). The 'elasticsearch-pool-yml' row was modified by 'spidle' on 2018-11-28 10:15:47. The 'jvm-pool-options' row was modified by 'spidle' on 2018-11-28 10:23:06.

在 ES 的官方文档中，没有详细且特意的介绍 `elasticsearch.yml` 中哪些参数需要配置在 master 节点，哪些需要配置在 data 节点。我大概翻了一下，其中 Local Gateway 参数明确提到需要配置在 master 节点，Fielddata 和 Node Query Cache、Indexing Buffer 类参数是需要每个 datanode 节点都部署的。

实际上我个人认为 ES 也是努力使越来越多的参数可以动态修改，所以从 5.x 以后将很多参数都提供了 API 可以在模板中进行配置，配置文件中的参数越来越少。也简单翻了一下源码，基本上没有特意区分节点角色参数的逻辑处理。这就说明了基本上 ES 的 `elasticsearch.yml` 参数配置最好是所有节点全部配置，不管是什角色，除非是真正区分角色的配置。这就正好适配了配置中心的意义，通过一次修改而使得全部节点配置统一生效。

最初就是按照 ES 集角色群设计了 4 个集群，但是我发现其实每种角色节点有很多参数配置也是重复的，参数变更修改要操作 4 次，所以最后抽象出来了一个 default 集群，default 中配置的都是通用的参数，只有具有角色特征的参数才会存放在各自不同 namespace 中。那么问题来了，集群做参数优化的时候修改那个角色集群的参数呢？没错，只需要修改 default 中的即可，所有和优化相关配置参数都是在 default 集群中的。

发布状态	Key	Value	备注	最后修改人	最后修改时间	操作
已发布	cluster.name	es	集群名称	apollo	2019-01-11 10:28:39	<input checked="" type="checkbox"/> <input type="checkbox"/>
已发布	node.max_local_storage_nodes	6	一台机器最多可以启动多少个ES节点	apollo	2019-01-11 10:28:39	<input checked="" type="checkbox"/> <input type="checkbox"/>
已发布	get.gateway.remove_stale_nodes	9	设置集群中小数据节点自动删除过期索引数	apollo	2019-01-11 10:01:54	<input checked="" type="checkbox"/> <input type="checkbox"/>
已发布	get.gateway.remove_stale_time	2m	设置老化数据索引过期的删除时间， 默认是5分钟，部分节点存储容量数据	apollo	2019-01-11 10:02:27	<input checked="" type="checkbox"/> <input type="checkbox"/>
已发布	gateway.expected_nodes	12	设置这个集群中节点的数量， 默认为12，一旦发现节点启动，就会立即进行数据恢复	apollo	2019-01-11 10:30:30	<input checked="" type="checkbox"/> <input type="checkbox"/>
已发布	indices.recovery.max_bytes_per_sec	60mb	设置恢复时每秒的带宽，约10 mb， 默认为0，即无限制	apollo	2019-01-11 10:37:02	<input checked="" type="checkbox"/> <input type="checkbox"/>
已发布	cluster.routing.allocation.node_initial_primaries_recoveries	4	在初始化时分配给每一个节点的副本数	apollo	2019-01-11 10:42:16	<input checked="" type="checkbox"/> <input type="checkbox"/>
已发布	cluster.routing.allocation.node_concurrent_recoveries	2	在添加、故障节点及调整时一个节点的最大允许同时运行分片副本数的个数， 默认为2，如果集群中有两个分片在重新分配	apollo	2019-01-11 10:56:40	<input checked="" type="checkbox"/> <input type="checkbox"/>
已发布	cluster.routing.allocation.cluster_level_rebalance	1	设置集群中最大允许同时运行分片副本数的个数， 默认为1，也就是说，整个集群最多有两个分片在重新分配	apollo	2019-01-11 10:56:56	<input checked="" type="checkbox"/> <input type="checkbox"/>
已发布	discovery.zen.minimum_master_nodes	2	设置一个集群中主节点数量的最小值	apollo	2019-01-11 10:59:07	<input checked="" type="checkbox"/> <input type="checkbox"/>
已发布	discovery.tls.timeout	10s	设置连接其他节点的检测时间，间隔比较快时可能会增加	apollo	2019-01-11 11:00:06	<input checked="" type="checkbox"/> <input type="checkbox"/>
已发布	discovery.tls.ping_interval	6s	节点间存活检测间隔	apollo	2019-01-11 11:02:36	<input checked="" type="checkbox"/> <input type="checkbox"/>
已发布	discovery.tls.ping_retries	10	存活检测时重试次数	apollo	2019-01-11 11:03:59	<input checked="" type="checkbox"/> <input type="checkbox"/>

在 hot、warm、master、client 中的配置实际上均是一些特有配置，如角色定义、日志路径、端口号等等，那么这里就有一个问题，如果在 hot 集群中和 default 集群中配置有冲突怎么办。实际上在这块设计中我们设置了优先级，也就是说 hot 集群中的配置优先级是大于 default 集群中的优先级的，当发生冲突时默认选择以角色集群中配置的参数为准。

另外对于一些具有节点特征的参数，如节点名称、节点绑定 ip、网卡等等，那么这类参数就保留在配置文件当中，不在 Apollo 中进行配置。所以最后配置的读取优先级是：角色配置 > default 配置 > 配置文件，这种设计一方面保证了有用的配置集中管理，层次划分清晰，另一方面可以使得配置迁移的安全性和健壮性，避免配置中心漏掉某些配置导致 ES 集群出现问题。

发布状态	Key	Value	备注	最后修改人	最后修改时间	操作
已发布	ES-PUBLIC	elasticsearch-cluster	elasticsearch集群配置	apollo	2019-11-28 11:05:03	<input checked="" type="checkbox"/> <input type="checkbox"/>
已发布	ES-PRIVATE	elasticsearch-cluster	elasticsearch单机集群配置	apollo	2019-11-28 11:05:16	<input checked="" type="checkbox"/> <input type="checkbox"/>
已发布	TW-PUBLIC	tw-pod-type-test	elasticsearch测试集群配置	apollo	2019-11-28 11:05:22	<input checked="" type="checkbox"/> <input type="checkbox"/>
已发布	JW-PRIVATE	jw-pod-type-test	elasticsearch私有集群配置	apollo	2019-11-28 11:05:58	<input checked="" type="checkbox"/> <input type="checkbox"/>

发布状态	Key	Value	备注	最后修改人	最后修改时间	操作
未发布	node.master	false	节点是否成为主节点	赵勇	2019-11-29 10:17:02	<input checked="" type="checkbox"/> <input type="checkbox"/>
未发布	node.data	true	节点是否成为数据节点	赵勇	2019-11-29 10:17:02	<input checked="" type="checkbox"/> <input type="checkbox"/>
未发布	node.ingest	false	节点是否成为摄入节点	赵勇	2019-11-29 10:17:02	<input checked="" type="checkbox"/> <input type="checkbox"/>
未发布	path.data	/logstash/es/elasticsearch/data/testdata	节点数据目录	赵勇	2019-11-29 10:17:02	<input checked="" type="checkbox"/> <input type="checkbox"/>
未发布	path.logs	/logstash/es/elasticsearch/logs/test	节点日志目录	赵勇	2019-11-29 10:17:02	<input checked="" type="checkbox"/> <input type="checkbox"/>
未发布	transport.tcp.port	9302	节点端口	赵勇	2019-11-29 10:17:02	<input checked="" type="checkbox"/> <input type="checkbox"/>
未发布	http.port	9202	节点端口	赵勇	2019-11-29 10:17:02	<input checked="" type="checkbox"/> <input type="checkbox"/>
未发布	node.attr.type	hot	节点属性	赵勇	2019-11-29 10:17:02	<input checked="" type="checkbox"/> <input type="checkbox"/>

参数文件拆分

除了 elasticsearch.yml 配置文件以外，ES 每个节点的 jvm.options 文件我们也经常需要修改，最基本的就是通过 -Xms 和 -Xmx 来调整 ES 节点分配的堆内存大小。根据冷热分离需求和节点角色的特性，hot、warm、master、client 分配的内存都是不一样的，所以很自然的 -Xms 和 -Xmx 就单独放到了每个角色的专有属性当中，其他的配置参数都放到 default 集群中。

Key	Value
-XX:+CMSInitiatingOccupancyFraction	75
-Djava.net.hostname	true
-Dfile.encoding	UTF-8
-Djava.nio	true
-Djdk.io.permissionsUseCanonicalPath	true
-Dio.netty.noUnsafe	true
-Dio.netty.allocator.optimization	true
-Dio.netty.recycler.maxCapacityPerThread	0
-Dlog4j.shutdownHookEnabled	false

Key	Value
-Dps	-Xms1g
-Dps	-Xmx2g

在 Apollo 页面配置设计上角色集群中我们没有在 application 中去加入配置参数，而是通过配置集群中的 namespace 名称作了一次链接操作。在每个角色的 application 中统一都配置 4 个 key 值来分别指向不同的 namespace，public 的 key 值都是 default 集群中的 namespace 名称，private 都指向该角色集群下的 namespace 名称。default 中为了保持统一和方便管理，在 application 中也配置了本地的 namespace 名称，也就是两个 public 的配置项，实际上这两个配置项是用不到的。这样设计的好处是当 namespace 名称需要改变时不需要更改 ES 中 Apollo 的代码而达到解耦的目的。

App ID	Cluster Role	Data Center	最后修改时间
elasticsearch	master		2018-12-21 16:00:37
elasticsearch	wara		2019-12-23 16:00:38
elasticsearch	master		2019-12-23 16:00:38
elasticsearch	hot		2018-12-23 16:00:39
elasticsearch	wara		2019-12-23 16:00:39
elasticsearch	hot		2018-12-23 16:00:39
elasticsearch	wara		2019-12-23 16:00:39
elasticsearch	hot		2019-12-23 16:00:39

App ID	Cluster Role	Data Center	最后修改时间
elasticsearch	hot		2018-11-21 20:41:00
elasticsearch	hot		2018-11-21 20:41:00
elasticsearch	hot		2018-11-21 20:41:00
elasticsearch	hot		2018-11-21 20:41:00

在实际运行的过程中，可以实时监测那些运行的 ES 节点读取到了当前配置，通过上图可以看到在 default 集群中实际上所有的节点都会进行读取其中的参数，但是和 application 中的配置无关，这个是用来保持统一方便管理的。而角色集群中的配置只有 ES 中的角色节点才会进行读取。这里需要注意的是 Apollo 客户端本身是由缓存机制的，其在本地会创建一个 cache 目录保留从 Apollo 中读取的配置，如果配置没有修改并发布过的话，那么就算 ES 节点重启它还是会显示的是你最后一次发布配置的时间。

最后值得一提的是 Apollo 本身提供了 properties、xml、json、yml 和 yaml 五种配置文件录入方式，需要在创建对应 namespace 时指定，虽然 elasticsearch.yml 本身是 yml 格式配置文件，但 yml 方式并没有带来客户端程序开发的优势，同时也不如 properties 在页面上看起来美观，最后我们还是进行了一下转换，统一使用 properties 格式进行录入，如下图所示 Apollo 支持文本方式进行批量导入。另外所有的 namespace 我们都设置为私有的，因为目前暂时没有和其他项目共享参数的需求。

```

1. cluster.name = elk
2. node.max_local_storage_nodes = 8
3. gateway.recover_after_nodes = 9
4. gateway.recover_after_time = 2m
5. gateway.expected_nodes = 12
6. index.translog.flush_byterate_per_sec = 20mb
7. cluster.routing.allocation.node_initial_primaries_recoveries = 4
8. cluster.routing.allocation.node_concurrent_recoveries = 1
9. cluster.routing.allocation.cluster_concurrent_rebalance = 1
10. discovery.zen.minimum_master_nodes = 2
11. discovery.zen.ping.timeout = 30s
12. discovery.zen.fd.ping_interval = 5s
13. discovery.zen.fd.ping_retries = 10
14. discovery.zen.ping.unicast.hosts.enabled = false
15. discovery.zen.ping.unicast.hosts = ["197.3.84.120:9300", "197.3.84.121:9300"]
16. http.cors.allow-origin = "/"
17. http.cors.enabled = true
18. Indices.fielddata.cache.size = 75%
19. Indices.breaker.fielddata.limit = 25%
20. bootstrap.mlockall=true

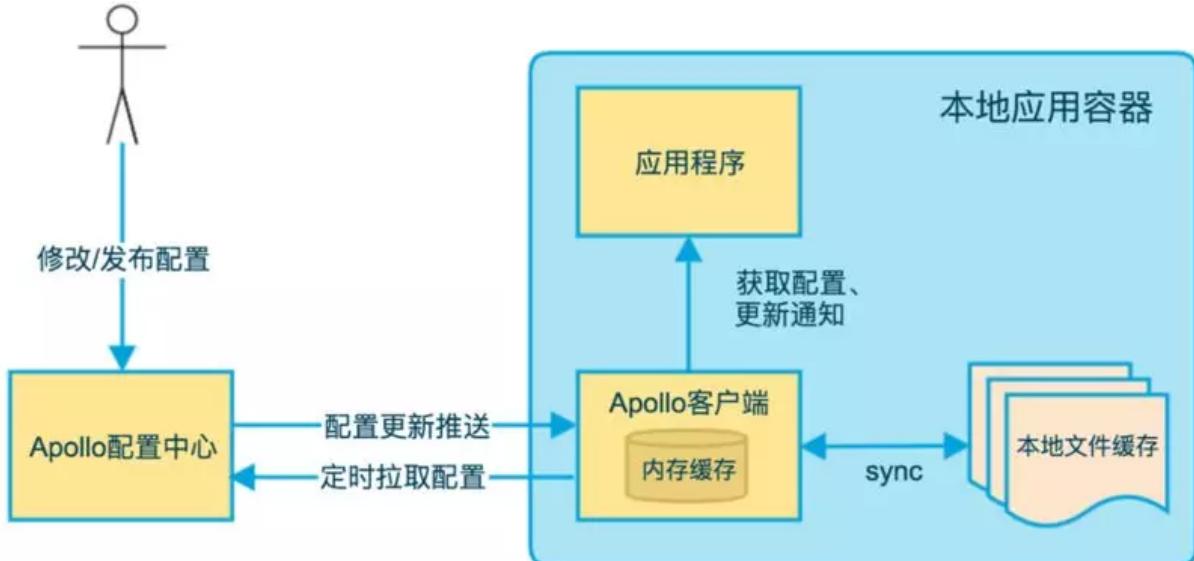
```

ES 源码改造

Apollo ES 架构设计结束之后就可以进行 ES 源码改造这块的工作了，实际上在我们在 ES 源码改造这块的所花费的时间远远没有设计的时候多，一旦设计定型源码改造就比较容易。结合 Apollo 客户端进行 ES 源码改造主要分为两部分工作：

Elasticsearch 配置源码改造和 JVM 配置功能开发。

Elasticsearch 配置源码改造



上图简要描述了 Apollo 客户端的实现原理：

1. 客户端和服务端保持了一个长连接，从而能第一时间获得配置更新的推送。（通过 Http Long Polling 实现）；
2. 客户端还会定时从 Apollo 配置中心服务端拉取应用的最新配置；
- 这是一个 fallback 机制，为了防止推送机制失效导致配置不更新；

- 客户端定时拉取会上报本地版本，所以一般情况下，对于定时拉取的操作，服务端都会返回 304 - Not Modified;
- 定时频率默认为每 5 分钟拉取一次，客户端也可以通过在运行时指定 System Property: Apollo.refreshInterval 来覆盖，单位为分钟。

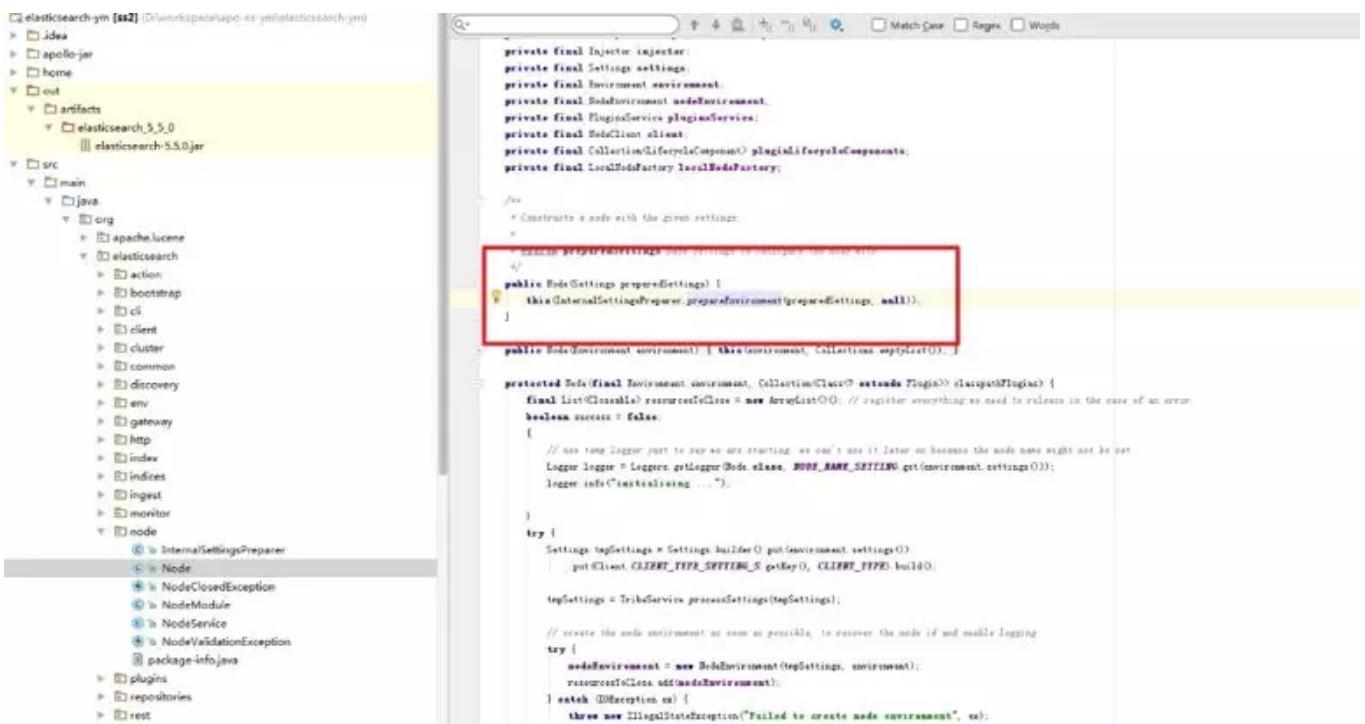
3. 客户端从 Apollo 配置中心服务端获取到应用的最新配置后，会保存在内存中；

4. 客户端会把从服务端获取到的配置在本地文件系统缓存一份；

- 在遇到服务不可用，或网络不通的时候，依然能从本地恢复配置。

5. 应用程序可以从 Apollo 客户端获取最新的配置、订阅配置更新通知。

ES 的主要核心包在 core 包里，通过 Bootstrap 类去启动实例化的 node 节点，在 node 的构造参数中，可以找到配置文件的加载，我们就是通过修改 InternalSettingsPreparer 这个类来实现 ES 通过 Apollo 客户端读取 elasticsearch.yml 的相关配置。



The screenshot shows the file structure of the `elasticsearch-gm` project in the left-hand sidebar. The `src/main/java` directory contains several packages: `org.apache.lucene`, `org.elasticsearch.action`, `org.elasticsearch.bootstrap`, `org.elasticsearch.client`, `org.elasticsearch.cluster`, `org.elasticsearch.common`, `org.elasticsearch.discovery`, `org.elasticsearch.env`, `org.elasticsearch.gateway`, `org.elasticsearch.http`, `org.elasticsearch.index`, `org.elasticsearch.indices`, `org.elasticsearch.ingest`, `org.elasticsearch.monitor`, and `org.elasticsearch.node`. Within `node`, there is a `InternalSettingsPreparer` class.

The right-hand pane displays the Java code for `InternalSettingsPreparer`. A specific method, `public NodeSettings preparedSettings()`, is highlighted with a red rectangle. The code within this method calls `this.InternalSettingsPreparer.prepareEnvironment(preparedSettings, null);`.

```

private final Injector injector;
private final Settings settings;
private final Environment environment;
private final NodeEnvironment nodeEnvironment;
private final PluginsService pluginService;
private final NodeClient client;
private final Collection<LifecycleComponent> pluginLifecycleComponents;
private final LocalNodeFactory localNodeFactory;

// ...
// Construct a node with the given settings.
// ...
// Create the environment
// ...
public NodeSettings preparedSettings() {
    this.InternalSettingsPreparer.prepareEnvironment(preparedSettings, null);
}

public NodeEnvironment environment() { return environment; }

protected void final Environment environment, Collection<Class<? extends Plugin>> elasticsearchPlugins() {
    final List<Class<?>> resourcesInClass = new ArrayList<Class<?>>; // register everything we need to values in the case of an error
    boolean success = false;
    {
        // use this logger just to say we are starting - we can't use it later on because the node name might not be yet
        Logger logger = LoggerFactory.getLogger(Node.class);
        NODE_NAME_SETTING.get(environment.settings());
        logger.info("initializing ...");
    }
    try {
        Settings topSettings = Settings.builder().put(environment.settings())
            .putClient(CLIENT_TYPE_SETTING.getKey(), CLIENT_TYPE.build());
        topSettings = TribeService.praseSetting(topSettings);
        // create the node environment as soon as possible, to recover the node id and enable logging
        try {
            nodeEnvironment = new NodeEnvironment(topSettings, environment);
            resourcesInClass.addAll(nodeEnvironment);
        } catch (Exception ex) {
            throw new IllegalStateException("Failed to create node environment", ex);
        }
    }
}

```

在 `InternalSettingsPreparer` 中 `PrepareEnvironment` 方法构建了一个 `Settings` 实例来加载给定或某认路径下的 `elasticsearch.yml` 文件，加载的配置都存放到 `output` 中，所以按照设计只要读取 Apollo 替换 `output` 中的值即可，`output` 底层实现的原理就是 map。代码很简单，就是首先读取 `elasticsearch.yml` 配置文件中的值，然后读取 Apollo 中 default 集群中的值，相同的 key 值进行覆盖，最后读取 Apollo 每个角色配置值，相同的 key 值覆盖就可以了。如下就是增加的代码块：

```

//start add apollo
try{
    //启动日志
    logger.info("开始加载apollo中配置的elasticsearch.yml的参数信息...");

    //System.setProperty("env", "DEV");//apollo environment set
    //System.setProperty("apollo.cluster", "xx");
    //获得本地服务器的ip
    // 读取配置机名
    //String name = InetAddress.getLocalHost().getHostName();
    Config config = ConfigService.getAppConfig();
    //读取当前服务根据对应的apollo-namespace
    //String nameSpace = config.getProperty(name, "necconfig");
    String nameSpace_private = config.getProperty("ES-PRIVATE", "necconfig");
    String nameSpace_public = config.getProperty("ES-PUBLIC", "necconfig");
    //logger.info("当前节点的nameSpace" + nameSpace);
    if(nameSpace_private==null || "necconfig".equals(nameSpace_private)){
        throw new RuntimeException("ES-PRIVATE当前服务器在apollo-application没有配置");
    }
    if(nameSpace_public==null || "necconfig".equals(nameSpace_public)){
        throw new RuntimeException("ES-PUBLIC当前服务器在apollo-application没有配置");
    }

    //根据所需要的nameSpace，获取其对应的参数
    System.out.println("展示xx中配置文件中版本的参数配置");
    //output底层实现原理是Map
    output.build().getHashMap().forEach((k,v)->{
        logger.info("old:k={{}},v={{}},k,v:");
        System.out.println("k=" + k);
    });
    logger.info("apollo参数开始put-public");
    Config apolloConfig_public = ConfigService.getConfig(nameSpace_public);
    Set<String> apolloSet_public = apolloConfig_public.getPropertyNames();
    Map<String, Integer> map = new HashMap<>();
    for(String key: apolloSet_public){
        String value = apolloConfig_public.getProperty(key, "necconfig");
        if(value.indexOf("[")>0){
            String split[] = value.replaceAll("\\\\", "\\").replace("[", "\\[").replace("]", "\\]").split("\\[\\]");
            for(int i = 0; i < split.length; i++) {
                output.put(key+"."+i, split[i].trim());
                map.put(key, i);
            }
        }else{
            output.put(key, value);
        }
    }
}

```

注意 ES 中 discovery.zen.ping.unicast.hosts 这种参数的值是中括号形式括起来的，底层实际上是一个数组，ES 在处理时会以参数 +. 数字的方式进行标识，所以需要特殊处理一下，最后要加上逻辑判断防止 default 中数组参数元素个数大于角色集群而造成的配置参赛个数溢出。另外 ES 参数中有些参数在 elasticsearch.yml 配置时需要加上双引号，这个双引号在 ES 真正加载时不会读取，所以在 Apollo 中配置时不要带双引号。我在配置 http.cors.allow-origin 曾经直接将双引号粘贴过去，结果导致了 elasticsearch-head 插件不可用。

修改源码集成了 Apollo 客户端的 ES 启动日志如下图所示，在新的日志中打印了读取配置的信息。从日志中可以清晰地看到首先 ES 会读取本地的配置文件，所有从本地配置文件读取的配置参数都记录为 old 参数，然后再从 Apollo 中读取配置，如果有同名参数则选择 Apollo 参数直接覆盖掉。

```

开始读取vm.options配置文件中的参数
line 22: -Xms1变成了[-Xmx1]
line 23: -Xmx1变成了[-Xmax1]
line 37: -XX:CMSInitiatingOccupancyFraction=74变成了【-XX:CMSInitiatingOccupancyFraction=74】
开始启动民生组件的elasticsearch-5.5.0版本
14:46:46.813 [main] INFO org.elasticsearch.node.InternalSettingsPreparer - 开始获取apollo中配置的elasticsearch.yml的参数信息...
提示在配置文件中原本的参数配置
14:46:47.684 [main] INFO org.elasticsearch.node.InternalSettingsPreparer - old:k=[bootstrap.memory_lock],v=[true]
bootstrap.memory_lock=true
14:46:47.685 [main] INFO org.elasticsearch.node.InternalSettingsPreparer - old:k=[cluster.name],v=[elk]
cluster.name=elk
14:46:47.685 [main] INFO org.elasticsearch.node.InternalSettingsPreparer - old:k=[cluster.routing.allocation.cluster_concurrent_rebalance],v=[1]
cluster.routing.allocation.cluster_concurrent_rebalance=1
14:46:47.685 [main] INFO org.elasticsearch.node.InternalSettingsPreparer - old:k=[cluster.routing.allocation.disk.threshold_enabled],v=[true]
cluster.routing.allocation.disk.threshold_enabled=true
14:46:47.685 [main] INFO org.elasticsearch.node.InternalSettingsPreparer - old:k=[cluster.routing.allocation.disk.watermark.high],v=[.99]
cluster.routing.allocation.disk.watermark.high=.99
14:46:47.685 [main] INFO org.elasticsearch.node.InternalSettingsPreparer - old:k=[cluster.routing.allocation.disk.watermark.low],v=[.95]
cluster.routing.allocation.disk.watermark.low=.95
14:46:47.685 [main] INFO org.elasticsearch.node.InternalSettingsPreparer - old:k=[cluster.routing.allocation.node.concurrent_recoveries],v=[2]
cluster.routing.allocation.node.concurrent_recoveries=2
14:46:47.685 [main] INFO org.elasticsearch.node.InternalSettingsPreparer - old:k=[cluster.routing.allocation.node.initial_primaries_recoveries],v=[4]
cluster.routing.allocation.node.initial_primaries_recoveries=4
14:46:47.686 [main] INFO org.elasticsearch.node.InternalSettingsPreparer - old:k=[discovery.zen.fd.ping_interval],v=[5s]
discovery.zen.fd.ping_interval=5s
14:46:47.686 [main] INFO org.elasticsearch.node.InternalSettingsPreparer - old:k=[discovery.zen.fd.ping_retries],v=[10]
discovery.zen.fd.ping_retries=10
14:46:47.686 [main] INFO org.elasticsearch.node.InternalSettingsPreparer - old:k=[discovery.zen.minimum_master_nodes],v=[2]
discovery.zen.minimum_master_nodes=2

14:48:25.494 [main] INFO org.elasticsearch.node.InternalSettingsPreparer - apollo参数开始put
14:48:25.495 [main] INFO org.elasticsearch.node.InternalSettingsPreparer - apollo参数put完毕
14:48:25.495 [main] INFO org.elasticsearch.node.InternalSettingsPreparer - new:k=[bootstrap.memory_lock],v=[true]
14:48:25.495 [main] INFO org.elasticsearch.node.InternalSettingsPreparer - new:k=[cluster.name],v=[elk]
14:48:25.496 [main] INFO org.elasticsearch.node.InternalSettingsPreparer - new:k=[cluster.routing.allocation.cluster_concurrent_rebalance],v=[1]
14:48:25.496 [main] INFO org.elasticsearch.node.InternalSettingsPreparer - new:k=[cluster.routing.allocation.disk.threshold_enabled],v=[true]
14:48:25.496 [main] INFO org.elasticsearch.node.InternalSettingsPreparer - new:k=[cluster.routing.allocation.disk.watermark.high],v=[.99]
14:48:25.496 [main] INFO org.elasticsearch.node.InternalSettingsPreparer - new:k=[cluster.routing.allocation.disk.watermark.low],v=[.95]
14:48:25.496 [main] INFO org.elasticsearch.node.InternalSettingsPreparer - new:k=[cluster.routing.allocation.node.concurrent_recoveries],v=[2]
14:48:25.496 [main] INFO org.elasticsearch.node.InternalSettingsPreparer - new:k=[cluster.routing.allocation.node.initial_primaries_recoveries],v=[4]
14:48:25.496 [main] INFO org.elasticsearch.node.InternalSettingsPreparer - new:k=[discovery.zen.fd.ping_interval],v=[5s]
14:48:25.496 [main] INFO org.elasticsearch.node.InternalSettingsPreparer - new:k=[discovery.zen.fd.ping_retries],v=[10]
14:48:25.496 [main] INFO org.elasticsearch.node.InternalSettingsPreparer - new:k=[discovery.zen.minimum_master_nodes],v=[2]
14:48:25.496 [main] INFO org.elasticsearch.node.InternalSettingsPreparer - new:k=[discovery.zen.ping.unicast.hosts.0],v=[██████████]
14:48:25.496 [main] INFO org.elasticsearch.node.InternalSettingsPreparer - new:k=[discovery.zen.ping.unicast.hosts.1],v=[██████████]
14:48:25.496 [main] INFO org.elasticsearch.node.InternalSettingsPreparer - new:k=[discovery.zen.ping.timeout],v=[10s]
14:48:25.496 [main] INFO org.elasticsearch.node.InternalSettingsPreparer - new:k=[gateway.expected_nodes],v=[12]
14:48:25.497 [main] INFO org.elasticsearch.node.InternalSettingsPreparer - new:k=[gateway.recover_after_nodes],v=[9]
14:48:25.497 [main] INFO org.elasticsearch.node.InternalSettingsPreparer - new:k=[gateway.recover_after_time],v=[2m]
14:48:25.497 [main] INFO org.elasticsearch.node.InternalSettingsPreparer - new:k=[http.cors.allow-origin],v=[/*]
14:48:25.497 [main] INFO org.elasticsearch.node.InternalSettingsPreparer - new:k=[http.cors.enabled],v=[true]
14:48:25.497 [main] INFO org.elasticsearch.node.InternalSettingsPreparer - new:k=[http.port],v=[9200]
14:48:25.497 [main] INFO org.elasticsearch.node.InternalSettingsPreparer - new:k=[indices.breaker.fielddata.limit],v=[85%]
14:48:25.497 [main] INFO org.elasticsearch.node.InternalSettingsPreparer - new:k=[indices.fielddata.cache.size],v=[75%]
14:48:25.500 [main] INFO org.elasticsearch.node.InternalSettingsPreparer - new:k=[indices.recovery.max_bytes_per_sec],v=[60mb]
14:48:25.501 [main] INFO org.elasticsearch.node.InternalSettingsPreparer - new:k=[network.host.0],v=[_local_]
14:48:25.501 [main] INFO org.elasticsearch.node.InternalSettingsPreparer - new:k=[network.host.1],v=[_eth1_]

```

这里需要提醒的是在 Apollo 客户端集成到 ES 的过程中，客户端所依赖 jar 包有可能和 ES 的某些插件的包有冲突，如果有冲突需要提前将这些包替换掉保留一份即可。

高可用也是我们特别关注的功能点，必须保证哪怕 Apollo 所有服务都挂掉也不影响 ES 的正常启动和运行。Apollo 客户端在启动后会在启动用户的根目录下创建一个 config-cache 缓存目录，目录中会存放所有 Apollo 的 namespace 配置文件，文件内容就是配置的 key-value 键值对，保证 ES 在遇到 Apollo 服务不可用，或者网络不通的时候依然能从本地恢复配置。

```

logger@192.168.1.11:/config-cache> ll
total 60
-rw-r--r-- 1 logger logger 178 Jan  9 08:30 elasticsearch+hot+application.properties
-rw-r--r-- 1 logger logger 266 Jan  8 20:36 elasticsearch+hot+elasticsearch-pri-yml.properties
-rw-r--r-- 1 logger logger 1031 Jan  8 20:36 elasticsearch+hot+elasticsearch-pub-yml.properties
-rw-r--r-- 1 logger logger 82 Jan  9 08:30 elasticsearch+hot+jvm-pri-options.properties
-rw-r--r-- 1 logger logger 401 Jan  9 08:30 elasticsearch+hot+jvm-pub-options.properties
-rw-r--r-- 1 logger logger 178 Jan  9 08:30 elasticsearch+master+application.properties
-rw-r--r-- 1 logger logger 249 Jan  9 08:30 elasticsearch+master+elasticsearch-pri-yml.properties
-rw-r--r-- 1 logger logger 1031 Jan  9 08:30 elasticsearch+master+elasticsearch-pub-yml.properties
-rw-r--r-- 1 logger logger 82 Jan  9 08:30 elasticsearch+master+jvm-pri-options.properties
-rw-r--r-- 1 logger logger 401 Jan  9 08:30 elasticsearch+master+jvm-pub-options.properties
-rw-r--r-- 1 logger logger 178 Jan  9 08:30 elasticsearch+warm+application.properties
-rw-r--r-- 1 logger logger 269 Jan  9 08:29 elasticsearch+warm+elasticsearch-pri-yml.properties
-rw-r--r-- 1 logger logger 1031 Jan  9 08:29 elasticsearch+warm+elasticsearch-pub-yml.properties
-rw-r--r-- 1 logger logger 82 Jan  9 08:30 elasticsearch+warm+jvm-pri-options.properties
-rw-r--r-- 1 logger logger 401 Jan  9 08:30 elasticsearch+warm+jvm-pub-options.properties
logger@192.168.1.11:/config-cache> cat elasticsearch+hot/elasticsearch-pri-yml.properties
#Persisted by DefaultConfig
#Tue Jan 08 20:36:20 CST 2019
node.master=false
transport.tcp.port=9302
path.logs=/loggerfiles/elasticsearch/log/hot
http.port=9202
node.data=true
node.ingest=false
path.data=/loggerfiles/elasticsearch/data/hotdata
node.attr.box.type=hot
logger@192.168.1.11:/config-cache>

```

下表是 Apollo 官方提供的高可用说明，我们在实践中完全按照官方推荐的架构搭建了 3 节点的 Apollo 集群，后端使用的 MySQL 数据库是主备模式，同时额外通过优先级读取策略在最坏的情况下会读取本地的配置文件实现了双保险（本地缓存文件失效也不影响 ES）。

场景	影响	降级	原因
某台 Config Service 下线	无影响		Config Service 无状态，客户端重连其它 Config Service
所有 Config Service 下线	客户端无法读取最新配置，Portal 无影响	客户端重启时，可以读取本地缓存配置文件。如果是新扩容的机器，可以从其它机器上获取已缓存的配置文件	
某台 Admin Service 下线	无影响		Admin Service 无状态，Portal 重连其它 Admin Service
所有 Admin Service 下线	客户端无影响，Portal 无法更新配置		
某台 Portal 下线	无影响		Portal 域名通过 SLB 绑定多台服务器，重试后指向可用的服务器
全部 Portal 下线	客户端无影响，Portal 无法更新配置		

某个数据中心下线	无影响		多数据中心部署，数据完全同步，Meta Server/Portal 域名通过 SLB 自动切换到其它存活的数据中心
数据库宕机	客户端无影响，Portal 无法更新配置	Config Service 开启配置缓存后，对配置的读取不受数据库宕机影响	

JVM 配置功能开发

ES 读取 jvm.options 配置文件是在启动脚本中直接实现的，所以无法通过直接修改 ES 源码来进行 Apollo 客户端开发。如果直接修改 elasticsearch 启动脚本来读取 jvm 配置比较复杂，而且这样修改对原来的启动脚本改造较大，不是我们所想要的。但 jvm 参数本身对于 ES 来讲还是具有一定的调优价值，尤其是内存分配上，在单台物理机上启动多个不同角色节点可能需要频繁的调整。最后我们在实现上采用了单独开发一个 jar 包在 elasticsearch 启动脚本之前执行，读取 Apollo 配置如果有 key 值不同在文件中进行替换。Apollo 客户端上的处理逻辑与 elasticsearch.yml 完全一样，优先级都是角色配置>default 配置>配置文件，区别就是配置文件不一致就直接替换，最终落地还是以配置文件中的参数配置为准。代码就不粘了，启动命令如下：

```
java -jar /logger/Apollo/Apollo-jvm.jar -DApollo.cluster=$APOLLO
```

这里有几点需要特殊说明一下。

首先 jvm 参数在 jvm.options 配置文件中是有两种格式的，一种就是传统的以冒号分隔的 key-value 键值对，这种在 Apollo 中直接 properties 方式处理就好，还有一种参数既是 key 值也是 value 值，如 -Xms2g，这种类型参数我们想了好几种解决方案，如把 key 值设置成和 value 值一样、对数字进行特殊处理、对特殊参数单独命名等等，但是最后一旦多次修改就会产生一些逻辑 bug，最终我们采取了一种折中的方式，首先如果是 key-value 键值对类型那么正常处理，如果是单一键值的话那么采用前缀字符串匹配的方式实现，也就是说比如 -Xms2g 这种参数实际上在 Apollo 配置的是 -Xms: -Xms2g，当 Apollo 客户端进行文件扫描时如果是非 key-value 且符合字符串前缀的就进行参数值覆盖，这就要求这种配置参数在进行 Apollo jvm 录入时保证是唯一的，这种方案的优势是一方面 jvm 参数一般 value 值直接做 key 值或者前缀到数字之前就可以保证唯一性，另一方面这种方式避免了自己命名参数导致含义不清，前缀 + 注释的方式在页面上显示清晰，有较好的用户体验。

其次因为我们在一台服务器上实际上是启动了多个 ES 实例的，所以在 Apollo 启动命令中是通过环境变量来区分角色集群和 jvm 配置文件路径的，环境变量通过进程管理工具 supervisord 来进行传递的。

最后需要提到的就是在 jvm 的启动实例列表中经常显示为 0，原因是单独开发的一个 jar 包在 ES 启动脚本前执行，执行后这个客户端就退出了，前面提过 Apollo 的客户端和服务端回保持一个长连接，这个长连接可以保证服务端获取客户端的心跳信息，一旦客户端退出了 Apollo 默认只展示最近一天访问过 Apollo 的实例，那么一天之后数据库就会清空对应字段，jvm 启动实例列表就会显示为 0。

发布状态	Key	Value	备注	最后修改人	最后修改时间	操作
已禁用	-Xms	-Xms1g	jvm最小分配堆内存	apollo	2018-11-28 11:06:16	<input checked="" type="checkbox"/>
已启用	-Xmx	-Xmx1g	jvm最大分配堆内存	apollo	2018-11-28 11:06:34	<input checked="" type="checkbox"/>

Supervisord 工具结合

进行 ES Apollo 客户端改造后我们发现了一个问题：在源码中按照 ES 的方式输出的日志无法打印到 ES 的系统日志中。这个问题困扰了我们很久也找不到原因，只能一点点去梳理源码。通过 ES 入口方法类找到了读取配置文件的 EnvironmentAwareCommand 中的执行方法，该方法是一个抽象方法，其对应的实现方法如下图：

```

    ...
    @Override
    protected void execute(Terminal terminal, OptionSet options) throws Exception {
        final Map<String, String> settings = new HashMap<>();
        for (final KeyValuePair kvp : settingOption.values(options)) {
            if (kvp.value.isEmpty()) {
                throw new UserException(ExitCodes.USAGE, "setting [" + kvp.key + "] must not be empty");
            }
            if (settings.containsKey(kvp.key)) {
                final String message = String.format(
                    Locale.ROOT,
                    "setting [%s] already set, saw [%s] and [%s]",
                    kvp.key,
                    settings.get(kvp.key),
                    kvp.value);
                throw new UserException(ExitCodes.USAGE, message);
            }
            settings.put(kvp.key, kvp.value);
        }

        putSystemPropertyIfSettingIsMissing(settings, "path.conf", "es.path.conf");
        putSystemPropertyIfSettingIsMissing(settings, "path.data", "es.path.data");
        putSystemPropertyIfSettingIsMissing(settings, "path.home", "es.path.home");
        putSystemPropertyIfSettingIsMissing(settings, "path.logs", "es.path.logs");

        execute(terminal, options, createEnv(terminal, settings));
    }

    /**
     * Create an {@link Environment} for the command to use. Overrideable for tests.
     */
    protected Environment createEnv(Terminal terminal, Map<String, String> settings) {
        return InternalSettingsPreparer.prepareEnvironment(Settings.EMPTY, terminal, settings);
    }
}

```

由上图可以看到，该 execute 方法需要先执行 createEnv 方法，即需要执行 InternalSettingsPreparer.prepareEnvironment(Settings.EMPTY, terminal, settings); (即红色箭头指向的 1) 该方法，而该方法就是 Apollo 的主要代码修改与存放区域。在该方法内部，会进行对环境的初始化，即对 yml 配置文件参数的加载。但该方法区并没有对 log4j2.properties 配置文件进行加载和其他处理。然后其执行一个新的 execute 方法 (即红色箭头所标识的 2)。该抽象方法所对应的实现方法如下：

```

    ...
    protected void execute(Terminal terminal, OptionSet options, Environment env) throws UserException {
        if (options.nonOptionArguments().isEmpty() == false) {
            throw new UserException(ExitCodes.USAGE, "Positional arguments not allowed, found " + options.nonOptionArguments());
        }
        if (options.has(versionOption)) {
            if (options.has(daemonizeOption) || options.has(pidfileOption)) {
                throw new UserException(ExitCodes.USAGE, "Elasticsearch version option is mutually exclusive with any other option");
            }
            terminal.println("Version: " + org.elasticsearch.Version.CURRENT
                + ", Build: " + Build.CURRENT.shortHash() + "/" + Build.CURRENT.date()
                + ", JVM: " +JvmInfo.jvmInfo().version());
            return;
        }

        final boolean daemonize = options.has(daemonizeOption);
        final Path pidfile = pidfileOption.value(options);
        final boolean quiet = options.has(quietOption);

        try {
            init(daemonize, pidfile, quiet, env);
        } catch (NodeValidationException e) {
            throw new UserException(ExitCodes.CONFIG, e.getMessage());
        }

        void init(final boolean daemonize, final Path pidfile, final boolean quiet, Environment initialEnv)
        throws NodeValidationException, UserException {
        try {
            Bootstrap.init(daemonize, pidfile, quiet, initialEnv);
        } catch (BootstrapException | RuntimeException e) {
            // format exceptions to the console in a special way
            // to avoid NPE stacktraces from guice, etc.
            throw new StartupException(e);
        }
    }
}

```

由上图可以看出，在执行 execute 的过程中，elasticsearch 开始对 BootStrap 这个类进行初始化。

```
/*
 * 
 */
static void init(
    final boolean foreground,
    final Path pidFile,
    final boolean quiet,
    final Environment initialEnv) throws BootstrapException, NodeValidationException, UserException
// Set the system property before anything has a chance to trigger its use
initLoggerPrefix()

// force the class initializer for BootstrapInfo to run before
// the security manager is installed
BootstrapInfo.init();

INSTANCE = new Bootstrap(); 安装安全器

final SecureSettings keystore = loadSecureSettings(initialEnv);
Environment environment = createEnvironment(foreground, pidFile, keystore, initialEnv.settings());
try {
    LogConfigurator.configure(environment);
} catch (IOException e) { 加载log4j2.properties配置文件, 对日志环境进行配置
    throw new BootstrapException(e);
}
checkForCustomConfigFile();
checkConfigExtension(environment.configExtension());

if (environment.pidFile() != null) {
    try {
        PidFile.create(environment.pidFile(), true);
    } catch (IOException e) {
        throw new BootstrapException(e);
    }
}
```

rg
apache.lucene
elasticsearch
action
bootstrap
Bootstrap
BootstrapCheck
BootstrapChecks
BootstrapException
BootstrapInfo
BootstrapSettings
ConsoleCtrlHandler
Elasticsearch
ElasticsearchUncaughtExceptionHandler
ESPolicy
JarHell
JavaVersion
JNACLlibrary
JNAKernel32Library
JNANatives
Natives
Security
Spawner
StartupException

从上图可以看出，因为我们修改的 ES 源码是在 InternalSettingsPreparer 类中，prepareEnvironment(Settings input, Terminal terminal, Map<String, String> properties) 方法加载配置 yml 文件的时候，此时 log4j2.properties 中的配置还没有进行加载，在 ES 对 yml 文件加载完成后，才开始对日志的配置文件进行的加载。故而出现了在 InternalSettingsPreparer 类中修改 prepareEnvironment 方法，添加的日志提示信息没有打印在 elasticsearch 的 logs 目录下的日志文件中的现象。如果想将我们手动添加的日志打印信息打印在 elasticsearch 的 logs 目录下的日志文件中，只能在 BootStrap 加载之后的文件中添加才可以实现，所以 ES 本身的日志在 BootStrap 加载之前是没有任何日志记录的。

找到了问题原因之后，基本可以确定在 ES 的 Apollo 客户端中如果需要打印日志需要引入第三方工具，同时逻辑上配置参数的读取是有优先级的，所以在日志中打印相关信息确定最终读取配置值这个需求是客观存在的。基于此我们在实际生产中引入了 Supervisord 进程管理工具，一方面可以批量的维护 ES 的运行状态，另一方面通过 Supervisord 提供的重定向功能将 ES 的屏幕输出到一个单独的文件当中，这样在 ES 的 Apollo 客户端只需要将需要打印的日志信息直接 print 出来即可。Supervisord 的配置样例如下：

```
[program:elkwarm]
environment=ES_JVM_OPTIONS=%(ENV_ELK_WARM_JVM_OPTIONS)s,APOLLO=%(ENV_APOLLO)s
```

```

command = /logger/elasticsearch/bin/elasticsearch -Epath.conf=/1
username = logger
autostart=true
autorestart=false
startsecs=3
priority=1003
stdout_logfile=/loggerfiles/elasticsearch/log/warm/cmbc_elk_warm.
stdout_logfile_maxbytes=10MB
stdout_logfile_backups=10
stdout_events_enabled=true

```

如上所示，在配置中需要将 elasticsearch 和 jvm 的 Apollo 角色环境变量传递进去，这样才能区分出同一台服务器不同 ES 实例的集群角色和配置文件路径，并且由于 hot 节点单台服务器上有两个实例环境变量还需要数字来区分一下。

此外，为了方便管理我们还使用 Supervisord-monitor 开源工具将 ES 集群所有服务器集成到一个页面上进行统一管理，除了具备启停功能之外还可以通过页面以 tail -f 的方式来查看我们单独生成的 cmbc_elk_warm.log 日志文件。



总结与展望

这是民生银行大数据团队对 Apollo 和 ES 两种开源产品的一次深入学习和探索，在业界没有相关案例的前提下，我们团队摸着石头过河，没有盲目地为了使用而使用，而是根据在生产环境中 ES 集群运行的实际情况有针对性的进行了 Apollo 架构设计和 ES 源码改造。

在构建天眼 ELK 日志平台配置管理中心的整个过程当中，Apollo 架构设计实际上要比 ES 源码改造花的时间多的多，无论多么优秀的开源产品、多么牛逼的技术框架都不能脱离实际应

用场景而存在，设计的合理必然会导致开发模块内部的高聚合和模块之间的低耦合，从而提高程序开发的效率，减少改造风险和增加落地的可行性。

后续我们一方面会推进日志平台本身配置集中工作，设计出比较合理的模式接入日志平台其他组件配置信息，如 Logstash，另一方面大数据产品均有服务器数量多、配置文件多难以集中管理的问题，如何让其满足更多的大数据产品是我们下一步所面临的挑战，最终目标是天眼 ELK 日志平台配置管理中心能够推广成为整个大数据平台的集中配置管理中心。

团队介绍

本文作者为中国民生银行大数据基础平台运维组团队：赵蒙、詹玉林、文乔、黄鹏程、焦媛、武文齐、孙玺。

民生银行大数据基础平台运维组团队是一个热衷技术的年轻团队，采用先进的技术架构及成熟的开源产品建设了民生银行一系列大数据基础平台，如天眼实时日志平台，实现了秒级延时的海量日志实时接入；大数据平台计算和非计算集群，支持非结构化数据的对象存储，同时支持机器学习和数据挖掘；Kafka 平台，实现了数据的分布式消息订阅功能，支持数据的实时处理等，为全行提供了全面和高效的大数据基础服务。

活动推荐

逐年增加的业务场景与数据量，对于系统的稳定性提出了更高的挑战。搭载自动化运维平台，甚至尝试实践 AIOps，是众多技术团队的现在进行时以及未来发展趋势。

QCon 全球软件开发大会广州站，特设「DevOps 最佳实践」专题，5 月 25-28 日邀你面聊大牛，收获实战经验。目前，大会 8 折报名中，立减 1360 元，团购更多惊喜！扫描下方二维码或点击阅读原文了解，有任何问题欢迎咨询鱼丸同学，电话：13269078023（微信同号）。

QCon全球软件开发大会 | 广州站 · 2019

高效运维最佳实践

技术大牛的经典与创新案例复盘

8折购票中 报名立减1360



[阅读原文](#)