# 绪论

本章介绍了本文的研究背景，并对当前日志系统和云计算系统的研究和实际应用现状进行了综合概述。从当前日志系统和云计算平台的发展趋势着手，提出了本文的研究工作，并对全文的行文进行了说明。

## 研究背景和意义

在互联网产业高速发展的现代，随着电子商务应用，移动互联网，以及社交媒体的普及。每个用户在与应用交互的过程中都会产生大量的数据。这些数据中的大多数都以日志的形式在前台或是后台被记录了下来。在各种互联网应用系统中日志系统往往数作为基础组件而存在。在各种系统的运行过程中，需要根据系统的日志来分析系统运行状态。在系统异常发生时，系统管理员需要根据系统的日志记录来推断故障发生的原因，日志数据是非常重要的调试依据。对于广告服务等数据驱动型的应用来说以日志形式固定下来的业务数据拥有巨大的价值，基于这些数据进行的分析和挖掘能够进一步提高应用广告投放，推广等的精度和成本分配的合理性。当日志数据达到一定的数量级时，这些数据就发生了价值上从量变到质变的转变，从而可以从中挖掘出许多规律性的行为。从而使得应用提供商更进一步地了解用户，为向用户提供个性化的服务提供了依据。

传统的企业应用是单块式程序，在一个单一的程序包中包含了企业应用所需全部逻辑，其承担了企业服务的所有部分。任何对这个系统的改动都需要完全地重新构建和部署一个服务器端应用程序的新版本。对于这样的应用，IBM，EMC等厂商一直在提供完善的集中式的日志解决方案。

随着虚拟化和云计算技术的发展，单块应用的缺点进一步显露了出来。无论是以Amazon Web Service，IBM Bluemix和Microsoft Azure 为代表的的公有云平台还是以OpenStack为代表的私有云解决方案，他们能够提供的大多是生命周期短（相对于生命周期以年计算的物理服务器），性能有限，稳定性欠佳，基于x86架构的虚拟机。巨大的单块企业应用在这样的环境下往往显得水土不服。

单块应用的这些不足催生了微服务风格应用架构：多个松耦合的服务（称为微服务）相互协作来构建应用程序。每一个服务运行在自己独立的进程中，微服务之间使用轻量而通用的方法，通常是HTTP或者RPC API来实现通信。各个服务能够独立地被部署和弹性地缩放，对应用的更改只会影响到一个微服务。当前企业应用已经开始向着微服务架构演进，在微服务场景之下，业务应用一般运行在一些云计算平台上，日志数据不再集中地存储在一个或者几个服务器上而是分散在数量庞大的虚机或是容器中。传统的日志管理方法在这种架构下不再合适。

本日志管理系统立足云计算管理平台，利用分布式的思想和先进的Elastic技术栈对日志管理平台设计和实现了一种可以快速有效水平扩展，高可用，高弹性，高容错的日志管理解决方案。本方案可以有效收集来自于异构应用的日志数据，提供了高效开放的API可供用户接入自己的数据分析系统，提供了友好的前端页面使得用户无论是通过PC还是手机都可以方便地查询日志数据，同时实现了良好的数据隔离，保护了用户的隐私和利益。

## 研究工作主要内容

本课题主要调研了作为云计算平台一部分存在的日志管理系统的相关内以及目前存在的优秀的日志系统实现。在此过程中，学习了kubernetes，Elastic技术栈，消息队列以及Restful API等先进技术。结合传统的日志管理系统，联系实际工作需要，设计并实现了一种被称为MSCP的容器化私有云平台上的基于Elastic技术栈的日志管理系统。并进行了如下的研究工作：

1. 首先查阅相关资料和文献，调研云计算领域的基本概念，基础知识和云计算业务基本特点。分析和研究MSCP平台上日志系统的特点和用户需求。根据传统日志管理系统的优势和缺陷，结合自身项目的实践经验，探索和总结日志管理系统的功能和性能需求，规划日志系统的技术实现路径。
2. 结合云计算平台的优势，针对MSCP平台上的海量数据。采用ElasticSearch集群对日志数据进行索引和存储。为了应对巨大数据量的挑战，支持各个层面上的水平扩展。
3. 针对日志系统要收集来自于不同来源和应用的日志数据，要做到租户之间，应用之间数据的有效隔离隔离。各个不同的应用进行写入日志，读取日志，获取统计数据等操作时都要经过鉴权过程。
4. 日志收集系统应当实现对用户应用零入侵，不对用户应用性能带来损失。同时支持用户对日志数据结构的自定义设计。
5. 日志平台需要提供开放的接口以供用户接入自己的日志分析系统。

本课题所述日志系统的数据源是来自于MSCP平台上部署的用户应用docker容器内部各各种日志信息，加上MSCP平台自身的日志数据，通过日志收集组件对这些数据源进行信息采集，收集到的日志信息主要来自于了实际生产和测试系统中的各种日志文件和控制台输出。同时第三方应用可以组织成JSON文档的key-value的形式向日志系统通过HTTP API推送自定义的日志数据。日志系统能够对大规模的日志数据进行解析，并按字段进行检索，持久化到存储系统中。 通过ElasticSearch来对所有日志进行解析处理，将所有数据进行规范化。通过开放的API来实现对历史记录的总结和回溯，对第三方日志分析系统提供接入手段。提供用户友好的UI方便用户对日志数据的查询。从而实现一个实际生产上可用的日志管理系统。

## 论文组织架构

本文的组织架构如下：

第一章，介绍本文的研究背景、日志系统的研究工作和调研日志系统的当前研究成果等内容。

第二章，主要对日志管理系统的技术背景和重要技术进行了详细介绍，包括微服务云平台的基本概念，MSCP平台的基本知识，MSCP平台上日志数据的特点。并对系统中用到的主要技术如Elastic stack，Journald和kubernetes等进行说明。

第三章，主要阐述MSCP平台上的日志系统的总体设计工作，包括需求分析、在云平台上的部署架构设计、日志系统自身架构和各个模块的等的设计。首先阐述整个系统的设计目标和需要成的工作。完成整体的架构设计，为实际的系统实现提供了架构依据。

第四章，主要阐述了MSCP平台上的日志系统的具体实现，对各个组件的实现进行了详细的阐述，综合前面介绍的关键技术，将整个系统继续搭建和集成。展示了数据收集，处理，存储，展示以及轮转的整个生命周期的过程。

第五章，对日志系统的主要功能进行了测试的设计，阐述了测试所用的环境和测试用例的步骤设计。除了测试了日志系统各个接口和功能，还测试了系统的性能，可扩展性，高可用性等诸多方面。

第六章，对论文进行总结，对未来的工作进行展望。

# 微服务日志管理技术概述

## 日志管理系统概述

## 微服务云平台

随着信息技术的发展，各行各业都在试图通过信息化来提高企业的工作效率。在此过程中，越来越多的企业业务被信息化，成为互联网上的一种服务。企业级应用程序通常由三个部分组成：一个客户界面（也称UI（User interface）由用户机器上的浏览器中的HTML页面和javscript代码或者是专用的客户端软件组合而成），一个数据库，以及一个服务端应用程序。这个服务端应用程序处理HTTP请求，运行逻辑，从数据库获取和更新数据，然后生成合适的HTML返回给用户的浏览器。这个服务器端的应用程序就是单块应用（monolithic application）——一个包含了企业应用所需全部逻辑的单一程序。任何对这个系统的改动都需要完全地重新构建和部署一个服务器端应用程序的新版本。 单块应用以其部署简洁，易于使用的优势在早期被绝大多数的企业所采用。然而，随着业务数量的增加和复杂度的提高，单块应用的问题渐渐显露出来。一方面，企业应用越来越臃肿，任何一点改动都会牵一发而动全身。哪怕仅仅是一个小小的修改都会使得整个应用需要被重新构建和部署。而不断膨胀的应用又使得重新构建和部署的成本变得越来越高，有时甚至需要在部署新版本的应用时暂停服务，这对许多企业来说是不可接受的。另一方面如果要实现对应用的横向扩展——通过启动更多的应用实例来提高性能，则需要扩展整个应用程序而不是应用中的某个部分，这将会造成资源的浪费。

随着虚拟化和云计算技术的发展，单块应用的缺点进一步显露了出来。无论是以Amazon Web Service，IBM Bluemix和Microsoft Azure 为代表的的公有云平台还是以OpenStack为代表的私有云解决方案，他们能够提供的大多是生命周期短（相对于生命周期以年计算的物理服务器），性能有限，稳定性欠佳，基于x86架构的虚拟机。巨大的单块企业应用在这样的环境下往往显得水土不服。

单块应用的这些不足催生了微服务风格应用架构：多个松耦合的服务（称为微服务）相互协作来构建应用程序。每一个服务运行在自己独立的进程中，微服务之间使用轻量的方法，通常是HTTP API来实现通信。各个服务能够独立地被部署和缩放，对应用的更改只会影响到一个微服务。通过灰度更新的方式，企业应用可以实现真正的在线升级。通过各个微服务自身的横向扩展，企业应用可以实现在稳定性不足的设备上的高可用和对资源的高效利用。各个微服务甚至可以采用不同的编程语言编写，由不同的团队来管理。 当前企业应用已经开始向着微服务架构演进，如何构建起一个合适的微服务平台以承载微服务化的应用已成为当前急需解决的问题。

### MSCP平台

MSCP平台是XXX公司正在开发的一套用于开发，部署，运维微服务化企业应用的私有云平台。该平台应当能为企业应用微服务化提供便利，能提高企业应用的开发，部署，运维的效率，提高企业应用的健壮性。为企业节省成本，帮助企业更好地实现信息化。

该系统具备以下能力：

* 支持15-20个节点（300-450应用）
* 全自动化部署
* 不同维度的自动弹性扩展
* 高度自动化运维
* 用户权限控制，数据隔离

MSCP平台基于kubernetes系统构建，可以运行在物理服务器，虚拟机，IaaS平台等各种基础平台之上。为了实现上述功能，我们的MSCP平台提供了以下的功能。

微服务运行基础架构：微服务化的应用需要一个合适的环境来运行，大量异构的微服务需要共享有限的物理资源从而实现对资源的最大化利用。而同时各个服务之间又需要适当的隔离以保护用户的数据隐私。为了实现这样的需求，我们通过虚拟化技术来为微服务提供基础环境。

应用编排系统：在微服务架构中，企业应用由大量的微服务构成。应用编排系统通常包括容器管理、调度、集群定义和服务发现等。通过容器编排引擎，容器被有机的组合成微服务应用，实现业务需求。

分布式存储：分布式文件系统是一种实际数据资源的物理存储资源不一定直接存在于节点节点上，而是分布在许多不同设备上，通过高速网络与节点相连。分布式存储利用分布式技术将多个服务器上的本地磁盘，固态存储等存储介质等组织成一个大规模资源池。同时，对上层的虚拟机和应用提供工业界标准的块存储和对象存储访问接口。在容器化微服务平台上，有状态的应用依赖分布式存储实现状态和数据的持久化以及容器间的数据共享。

网络虚拟化：网络虚拟化是一种重要的网络技术，该技术可在对物理网络进行重新分配，从而使得处于不同网络内无法互相通信的节点能相互连通或者是在不同用户之间实现网络隔离，给各个用户提供独立的网络资源切片，从而提高网络资源的利用效率，实现“有弹性的网络”。

DevOps平台：随着软件发布迭代的频率越来越高，传统的「瀑布型」（开发—测试—发布）模式已经不能满足快速交付的需求。2009 年左右被称为 DevOps 的新的模式应运而生。DevOps将开发，测试，运维的流程优化，运用大量的自动化工具将开发和运维一体化。通过CI/CD使得软件构建、测试、发布更加快捷、频繁和可靠。我们的微服务平台将提供帮助企业实施DevOps的工具。

监控系统：为了方便用户对平台和应用的性能和资源使用进行监控，本平台设计了监控系统。监控系统将会不断收集来自于各个容器，应用以及底层基础设施的性能数据。该系统将对用户提供HTTP API，以方便用户将其与分析工具或其他现有系统集成。

日志系统：日志是用户了解应用工作状态，分析故障原因。该系统提供了对日志收集，分发，储存，分析，提取的端到端的解决方案并可以进一步地和数据挖掘服务对接。从而帮助客户迅速定位问题，分析故障。

本文中介绍的便是其中的日志系统部分。

### MSCP平台日志特点和需求

日志系统的数据源是来自于MSCP平台上部署的用户应用docker容器内部各各种日志信息，加上MSCP平台自身的日志数据，通过日志收集组件对这些数据源进行信息采集，收集到的日志信息主要来自于实际生产和测试系统中的各种日志文件和控制台输出。同时第三方应用可以将日志数据组织成JSON文档的key-value的形式向日志系统通过HTTP API推送自定义的日志数据。日志系统要能够对大规模的日志数据进行解析，并按字段进行检索，持久化到存储系统中。

## 日志管理平台

日志是计算机系统同维护人员以及开发人员交流的重要途径。特别是对于现代互联网应用系统来说，每小时产出数以万计的日志。这些日志中记录了对系统操作大量一般或是异常信息，比如系统工作过程中的关键的操作记录，软硬件错误或是异常信息，用户对系统更改的操作记录等。这些信息不仅仅给开发，运维人员提供了维护的依据，更能够挖掘出如用户特征，行为习惯，关注热点等诸多更深层次的信息。可以说，现代信息系统的日志是企业用户的数据金矿。日志管理就是通过对不同来源的日志统一收集，处理，储存，展示，方便用户从中提取有用的信息。

目前，在市面上已经有了许多开源或是商业的日志管理解决方案，每一个系统管理员也都会有自己管理日志的独特方案。最简单的，系统管理员可以通过shell，python等脚本语言过滤，处理和分析，利用Linux系统自身的特性管理日志数据的生命周期。这种简单有效的方式得到了许多开发者和管理员的青睐。但是这种简单的方式有着显而易见的局限性，那就是只能处理部署在单个服务器上个的单个应用产生的日志。对于微服务化应用产生的高度分散的日志无法有效地处理。当然，目前亦有许多成熟的日志分析工具，如Awstats，Webalizer等等就是专门为分析日志而生的。同时，百度，谷歌等公司亦提供了一些在线工具可以用于对日志的分析。但是这些工具虽然拥有出众的分析能力但是却无法和我们的微服务平台相集成，同时由于这些工具都无法实现集群化部署，其性能无法满足微服务云平台的需求。另外，在信息安全越来越受重视的当下，企业用户并不愿意将包含有自己业务信息的日志上传到公用服务器上进行处理。面对部署在由数百台服务器构成的微服务平台上，短生命周期的数以千计的微服务产生的每秒万条以上的日志，以及企业级应用对日志管理的特殊需求，这些传统的解决方案都已难以满足需求。

对日志分析应用来说，其处理的数据规模十分庞大。在这个移动互联网高速发展的时代，数据的规模高速扩张，一个应用可能要服务于数以万计的用户，而一个终端用户可能在一天内进行数十次交易，单单一次交易便能够生成数十条日志记录。这样的服务只能由运行在云平台上的微服务化的应用来实现。

在云平台上的应用数量众多种类各异，不仅要求运行日志管理服务拥有可靠的日志存储能力，同时还要能够高性能地接受来自分布式应用的日志数据。此外，用户往往还要求云平台的日志管理服务能够提供一定的日志分析和报警功能。这些日志数据直接来自于用户应用，这就造成了这些数据的结构往往只有应用开发者自己能够理解的状况，这样的（非结构化的）数据无法被有效地保存到传统的结构化数据库中，我们的日志系统需要将这些非结构化的数据尽可能地保存下来。为了满足这些需求，我们的日志系统应当能够实现如下的特性：

* 对用户应用零入侵
* 通用的应用程序接口（api）
* 覆盖常见日志形式的日志收集器
* 允许用户自行开发日志收集组件并且接入我们云服务系统
* 用户，租户间数据隔离
* 高效的数据查询和过滤接口，同时和至少和某一种流行的查询方式相兼容
* 日志服务能够实现分散部署，高可用，可扩展
* 多数据中心的数据能够统一查询
* 日志数据能够提供热备份

### Elastic stack

Elastic stack是Elastic公司以ElasticSearch全文搜索引擎为核心打造的由一系列软件工具构成的数据分析系统。由于该系统中最重要的三个部分是Elasticsearch，Logstash以及Kibana所以该系统也曾被称为ELK Stack。后来Elastic 团队收购了 Packetbeat 团队，就建立了 Beat，Beat 是一个轻量级的数据收集平台，可以将不同的数据发送 ELK 系统，例如日志、网络数据、系统信息等等。Elastic 团队在命名时最终将 ELK + Beat 命名为 Elastic Stack，并将整个产品线的版本提升至 5.0。

Elastic Stack中最重要最核心的部分是ElasticSearch。ElasticSearch是Elastic公司开发的基于Apache Lucene的开源搜索引擎服务器软件。它一方面可以作为一个拥有良好的写入性能的NoSQL数据库使用，另一方面拥有第一无二的强大的近似的实时全文搜索和分析能力。这样的能力使得用户能够对海量的数据进行复杂的查询。经过清洗的数据在被输送到ElasticScarch中后，Elasticsearch会对这些数据建立索引，根据配置将数据分发到不同的分片上。同时，Elasticsearch也会对这些分片建立副本以降低结点失效时数据丢失的风险。这些分片和分片的副本会被平衡地分配到不同的数据节点上，从而降低了单节点的压力。这样的设计也为集群的建立提供了可能。当系统压力增加时，只需要将新的数据节点加入集群。整个集群的的性能便能够通过横向扩展几乎无限地扩张。

Logstash 是开源的服务器端数据处理管道，能够同时从多个来源采集数据、转换数据，然后将数据发送到包含ElasticSearch，mangoDB，kafka等在内的多种数据后端中，多个logstash亦可串联使用从而实现更为复杂的数据处理过程。数据往往以各种各样的形式，或分散或集中地存在于很多系统中。Logstash 支持各种输入选择，可以在同一时间从众多常用来源捕捉事件。能够以连续的流式传输方式，轻松地从各种服务采集数据。 数据从源传输到存储库的过程中，Logstash 过滤器能够解析各个事件，识别已命名的字段以构建结构，并将它们转换成通用格式，以便更轻松、更快速地分析和实现商业价值。

Beats平台集合了多种单一用途数据采集器。这些采集器安装后可用作轻量型代理，从成百上千或成千上万台机器向Logstash或 Elasticsearch发送数据。Beats是数据采集的得力工具，它们能够把数据汇总到 Elasticsearch。如果需要更加强大的处理性能，Beats 还能将数据输送到 Logstash 进行转换和解析。 各种不同的Beats收集器都是以用于转发数据的通用库 libbeat 为基石。Beats社区提供了多种不同用途的收集器如用于收集日志文件的FileBeat，用于收集来自于Journald的日志数据的JournalBeat以及用于收集网络数据的PacketBeat等。

### kubernetes

kubernetes是Google开发的基于golang开发的容器化应用编排和管理工具。其可以实现在跨主机的环境上的应用编排，部署，弹性扩展，负载均衡等等一系列的功能。同时，这些这个kubernetes集群都可以通过restful的API来实现管理和使用。 kubernetes实现了以下功能：

1. 利用对容器化的应用进行包装，实例化以及运行。支持从公用和私有服务器上下载image。
2. 对容器化应用集群进行管理，实现了容器之间跨主机的网络互联互通。
3. 实现了对kubernetes集群内的资源的管理。可以通过配置资源需求限制容器使用的资源，同时自动地根据可用资源将workload分配到各个计算节点上。
4. 通过对服务(Service)的定义，实现了服务发现，负载均衡以及服务依赖管理。从而实现了对服务的生命周期管理以及对服务的调度。
5. kubernetes会自动管理应用的健康状态，当应用发生意外退出时可以将其自动重新部署，以保持应用的可用性。
6. 支持多种多样的基础平台，可以在AWS，IBM Cloud等云计算平台上工作，也可以直接部署在物理机上。
7. 提供各种监控接口，可供用户接入自己的监控平台。

kubernetes的设计思想在于维护应用集群始终处于用户计划的状态。它建立了一套完整的机制提供了对应用的全生命周期管理。

Pod在kubernetes中是最小的可以部署单元，每一个Pod都可以包含一个或多个containers。这些containre一般是同一个应用的组成部分，他们相互依赖共同提供同一服务。Pod的设计思想是想在contaltier的环境中模拟出一个抽象的逻辑主机。pod中的各个containers运行在同一计算节点上，同一个pod里的contianer共享同一组volumes，同一个network namespace/IP和以及port space。同一个pod里的contianers可以通过locaihost互相访问。kubernetes在调度任务时，会将Pod作为一个整体来考虑。kubernetes会考虑这个节点是否拥有足够的资源来提供给整个pod从而避免免资源不足。当一个pod所在的计算节点掉线pod会被调度到新的节点。

Service是一个pod服务池的代理抽像，由于pods是短生命周期的对象，其有动态性给服务访问带来了困难。因此kubernetes提供了代理抽象的概念service, pods的访问由service代理，对外部透明。目前的实现方法是通过一个固定的虚拟IP来定义，然后通过各个计算节点上的IPtable规则来将服务请求传递给各个真正提供服务的Pod。

每个Node都会运行一个kube-proxy服务, kube-proxy会持续访问kubernetes api以获取service的新动态，每一个pod建立时都会有一组自定义的labels，同时向kubernetes master进行注册，记录相关的信息。kube-proxy会将本地运行的容器的label和service的设置进行比较。从而维护用于数据包转发的iptaible规则。

Replication Controllers可以保证任意时刻都有指定数量提供同样服务的pods在运行， pods可运行在不同的物理主机上，当pod的数量少于指定值，会自动创建pods,否则删除多出来的pods. Replication Controllers保证了服务的高可用性。Replication Controllers同时还拥有以下功能：

1. Rescheduling，确保指定数目的pods副本运行。当副本不足或者原本的计算节点失效时安排，在可用的计算节点上安排新的pod。
2. Scalinit，可实现pods副木的水平扩展。
3. Rolling updates，当应用容器的镜像需要更新时kubernetes会根据配置在不停机的情况下完成应用的更新工作。

### kafka

Kafka是由Linkedin开源的一种高吞吐量的分布式发布订阅消息系统，它可以处理消费者规模的网站中的所有动作流数据。其设计目标是高性能地Append写日志，存取的代价为O(1),直接由sendfile系统调用持久化文件，绕过多次内存的数据进出, 可以在普通的服务器上支持每秒万条数量的消息持久化。 Kafka专注于大吞吐的消息队列，另外一个著名的消息队列RabbitMQ，专注于消息传递的事务性保障．一般用于实现高可靠性的消息系统，RabbitMQ与Kafka的区别如下表所示：

|  |  |  |
| --- | --- | --- |
|  | RabbitMQ | Kafka |
| Language | Erlang | Scala |
| Implementation | AMQP model | Basic MQ |
| Producer/Broker | Push | Push |
| Broker/Consumer | Push | Pull |
| Availability | Mirror queue | Multiple replicas |
| Parallelism | Multi queues | Partitions |
| CAP | AP/CP | CA |
| Throughput | Low | High |

Kafka和RabbitMQ的consumer消费消息的方式不一样，Kafka是pull的方式，可以大大减轻Kafka服务器的压力，Kafka的吞吐量远远超过了RabbitMQ。

Kafka是根据topic进行消息分类，Kafka为了实现对同一个topic的消息读写的高吞吐性能，对消息进行分区存储如下图所示。并行读写提高性能。



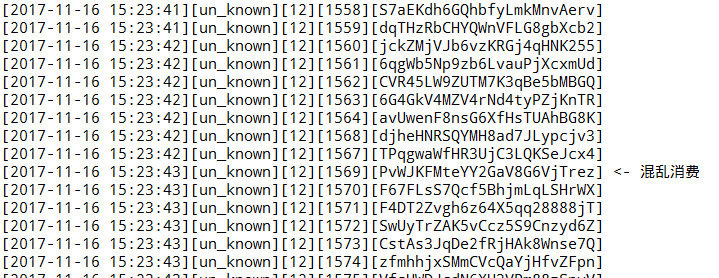
“Kafka partitions”

一个topic保存在多个partitions上，来突破单机磁盘文件尺寸的限制，有效的提升系统的并发读写能力。

Kafka的消息日志文件会根据broker中的配置保留一定时间后删除，到达规定期限后，无论日志是否被消费，文件都会被清除，Kafka通过过期日志清理策略来释放磁盘空间。Kalka集群的Producer和consumer的状态信息都由zooKeeper保存，这样可以使组件解耦，ZooKeeper的高可用性保证了Kafka集群的高可用性。

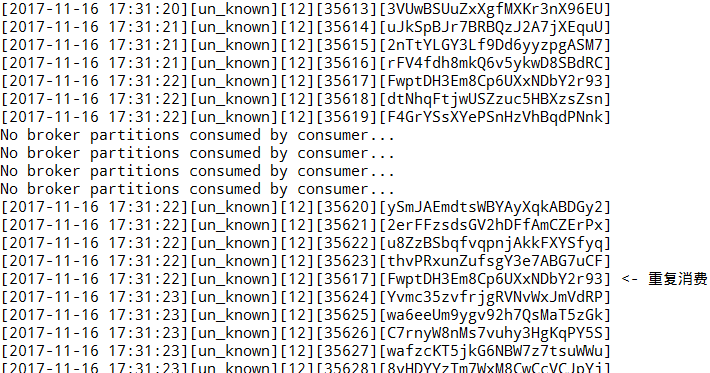
当前很多消息队列服务提供可靠交付保证（RabbitMQ等），默认是即时消费，不适合离线。Kafka系统的设计目标是高吞吐，不是高可靠性，但并不代表Kafka不可靠，只是使用场景上不同。

Kafka并不是消费一条消息就确认一条消息，简化了消息确认过程，以此来达到高 吞吐的目的。下图展示Kafka消费消息不是严格按照时间顺序到达。



“Kafka Mass”

客户端在消费Kafka的时候，如果中问过程中与Kafka集群的网络出现了中断，在重启消费数据的时候有可能会重复消费之间己经消费过的数据，这是kafka采取的at-least-one delivery的策略所造成的，kafka仅保证一条消息至少被消费一次。当consumer掉线后，有些消息可能会被重复delivery。下图展示了在断开重连的情况下消息的重复消费。



“Kafka multi”

Kafka通过协调组件ZooKeeper维护集群状态和消息读写的偏移值，不存在单点故障。满足分布式系统的要求。

## 本章小结

本章主要对日志管理系统的技术背景和重要技术进行了详细介绍，包括微服务云平台的基本概念，MSCP平台的基本知识，MSCP平台上日志数据的特点。并对系统中用到的主要技术如Elastic stack，kafka和kubernetes等进行了说明。

# 日志系统设计

## 需求分析

本系统的设计需求主要是在MSCP云服务平台的实际生产中提出的统一处理各微服务日志的需求。结合当前十分流行的Elastic技术栈，在kubernetes平台上，结合kafka等技术。搭建一个高可用，高性能的日志管理平台，从而统一管理部署在MSCP平台上的各种应用产生的服务。完成对各个应用的日志数据的收集、整理、汇总、检索等工作，同时方便用户对日志数据的进一步处理。

日志系统的数据源是来自于MSCP平台上部署的用户应用docker容器内部各各种日志信息，加上MSCP平台自身的日志数据，通过日志收集组件对这些数据源进行信息采集，收集到的日志信息主要提供了实际生产和测试系统中的各种日志文本。同时第三方应用可以接受以组织成JSON文档的key-value的形式向日志系统通过HTTP API推送自定义的日志数据。日志系统能够对大规模的日志数据进行解析，并按字段进行检索，持久化到存储系统中。

ElasticSearch技术栈中的各个部分均能够在kubernetes集群之上实现良好的扩展性，和高可用性。通过ElustkSearch来对所有日志进行解析处理，将所有数据进行规范化。通过开放的API来实现对历史记录的总结和回溯，对第三方日志分析系统提供接入手段。提供用户友好的UI方便用户对日志数据的查询。

### 日志系统特性

日志系统有以下特性：

* 系统各个组件之间实现松耦合，用户自身的应用程序和日志收集，日志存储，日志分析之间解耦。
* 对用户应用零入侵。
* 扩展性，支持迅速调整节点数量以实现水平扩展。同时能够实现根据负载情况自动调整系统规模。
* 高可用，集群化部署。少数几个的节点的失效不会造成整个集群服务的中断也不会造成数据丢失。
* 高性能，高并发，低延时地接受和查询日志支持全文检索。
* 提供Rest风格的API接口
* 提供用户友好的UI

### 分布式日志系统处理流程

日志按来源分为三个不同种类，来自容器内部stdout/stderro的容器日志，来自容器内部日志文件的应用日志，和来自Http API的应用日志，用户可在此基础上结合自己的应用的部署情况和日志记录时间对日志进行过滤检索。

日志的处理流程如下：

* 应用主动将日志通过日志推送API传输到日志系统中，或者本日志系统之中的收集组件也能够主动地收集应用本地的日志，然后同步到日志系统中。
* 日志系统将数据落盘到数据库中。
* 日志系统需要提供日志查询的接口。

### 分布式日志系统功能需求和性能需求

日志系统的功能需求：

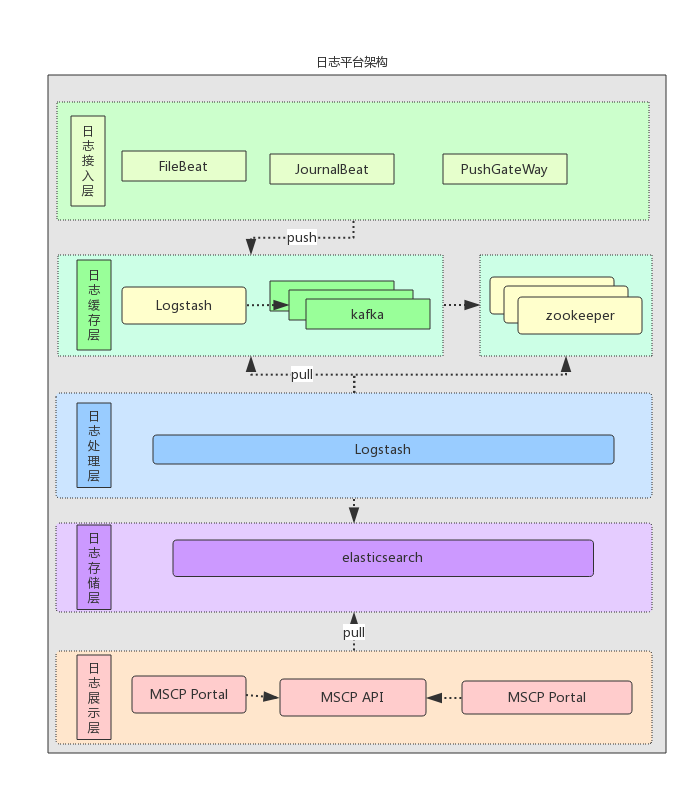
* 能够零入侵地收集容器内的日志数据。
* 拥有高性能的日志写入、读取API。
* 提供日志的查询以及日志系统管理的API接口。同时兼容ElasticSearch的查询语法。
* UI上提供日志按照时间正序查询以及逆序直询接口，并支持全文检索。
* 实现应用和应用之间，租户和租户之间的数据隔离并兼容MSCP系统的Role based access control模型。

分布式日志系统的性能需求：

* 在单region支持12krps以上的日志落盘能力。
* 单region支持TB级的数据的存储能力。
* 查询平均在5秒以内完成，简单查询在2秒以内完成。
* 日志落盘延时在10秒以内。在region间实现无限水平扩展。

## 日志系统总体架构设计

日志系统架构如下图所示：



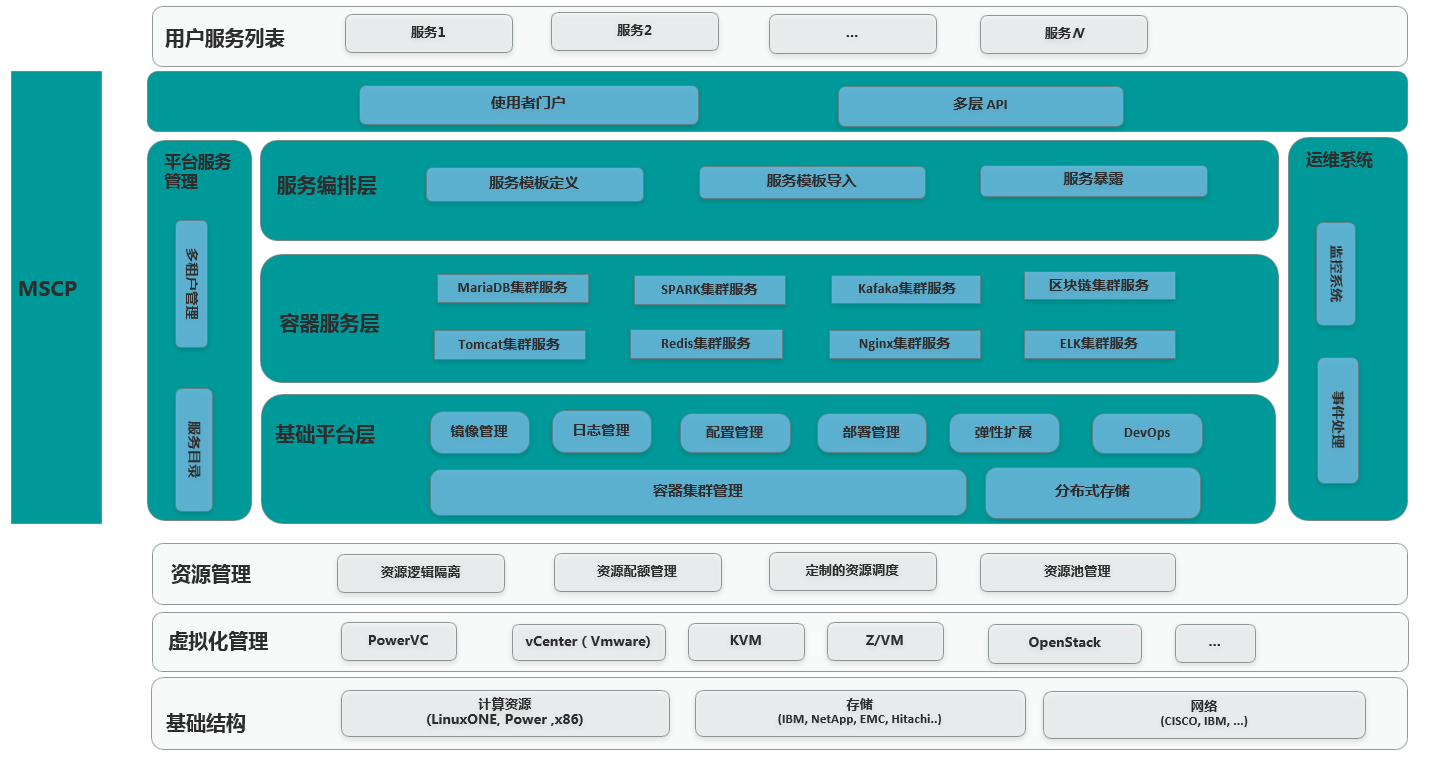
“Arch log-system”

* 数据收集组件：日志收集器，HTTP Restful API接口
* 数据清洗组件：logstash分析器
* 数据缓存组件：高速消息队列缓存kafka
* 数据存储组件：ElasticSearch
* 数据查询和集群管理组件：数据查询和集群管理API Server
* 前端UI

数据收集组件从docker容器内部收集包括控制台输出，日志文件等在内的日志数据，并且开放HTTP API接口以供用户应用直接推送日志数据。该API接口能够通过分析数据来源自动将这些数据归入某用户名下从而实现数据隔离。logstash分析器对来自各个数据源的数据进行清洗，删除一些不需要的字段，同时对数据来源作分析，形成半结构化的数据，传递给高速缓存kafka。由于logstash作为无状态的服务，可以几乎无限地水平扩展，其性能往往远远超过ElasticSearch落盘的性能。Kafka高速缓存可以有效地将到达的数据缓存起来，等待ElasticSearch空闲时再将数据落盘。这样的设计一方面挡住了高并发的访问减少了日志数据落盘时的阻塞，另一方面有效地利用了ElasticSearch的性能。作为MSCP平台API接口的一部分，日志服务也提供了restful的API接口，通过该接口能够查询和管理日志服务的状态，同时其查询接口兼容ElasticSearch查询语法。这个API接口也能够和MSCP云平台的role based access control相配合有效地对用户数据进行隔离。在MSCP前端页面上我们提供了一个前端页面，用户可以方便地通过这个页面查询自己应用的日志；管理员用户还可以管理和查询日志服务的状态。下面详细说明各个组件的设计方案。

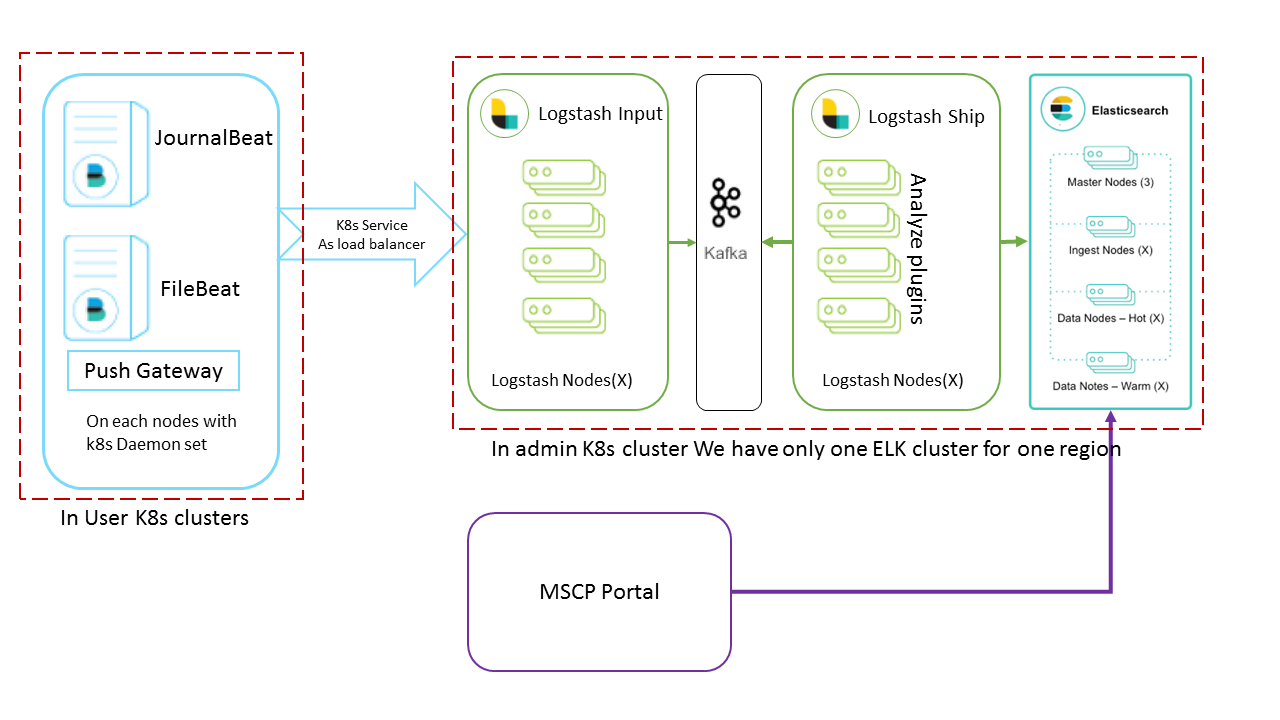
## 日志系统云平台层设计

MSCP微服务云平台的整体架构如下图所示



“Arch mscp”

在MSCP平台上，每个zone对应着一个kubernetes集群，这个集群一般被安置在同一个服务器机房的内部；而同一个region中的各个不同zone一般部署在同一个数据中心内部，其间一般由高速网络相连接（一般有多个千兆网络）。而region一般对应用户部署在不同地理位置上的数据中心，如用户的北京数据中心，纽约数据中心等等。其间一般通过公用网络相连接，这些链接一般容量不足，同时价格昂贵，无法支持大量日志数据的传输需求。针对以上情况我们对日志系统在云平台上的部署作了如下图的规划：



“Topology log”

每个region拥有自己的ElasticSearch数据库和kafka集群，这些集群与MSCP平台的其他基础组件一同作为一些deployment被安装在一个被称为Admin zone的kubernetes集群之中。用户无法在此zone中部署自己的应用。这样的设计保证了MSCP平台的稳定性和用户体验。同一个region中的日志收集器和分析器均将数据发送到region中的ElasticSearch中。

而日志收集组件由于需要部署到kubernetes中的每一台计算节点上。所以作为DaemonSet被部署到每一个kubernetes集群之中。

而日志查询API Server作为MSCP平台API Server的一部分能够根据用户查询的不同从不同的数据源中查询日志数据。

日志前端UI亦作为MSCP UI的一部分，与其一同部署。

## 日志系统各组件设计

下面介绍日志系统各个组件的设计。

### 数据收集组件

日志收集组件主要由两个部分构成，日志文件收集器和HTTP API接口。

日志收集器收集docker容器中的应用发送到控制台以及日志文件中的日志。这也是应用最常见的日志记录方式。通过这种方式收集日志不需要用户对自己的应用作出任何更改，做到了对用户应用的零入侵。对于容器化应用来说，其最佳实践是将所有日志发送到控制台，所以这些来自控制台的日志实际也是日志数据中最重要的部分。对于这部分日志我们将其重定向到系统中的journald服务之中，然后通过部署在每一台计算节点上的journalbeat收集器统一发送到日志服务之中。

另外，传统上应用开发者一般会选择将各种不同的日志，如服务器日志，系统日志，业务日志等输出到不同的日志文件之中保存到服务器本地。对于这一部分日志我们将其挂载到容器外部，从而暴露给负责收集日志文件的filebeat收集器，完成收集工作。

此外我们还提供了一个HTTP API接口以供愿意修改自己应用，或者是有特殊需求的用户使用。用户可以直接通过这个API向我们的日志系统推送日志数据。这个API充分考虑到了用户自定义日志数据结构的需求。用户除了可以向我们的系统提交日志文本和时间戳等必须的信息职位，还可以以key-value的方式描述一些自定义的字段，从而大大增强了日志系统可以接受的数据的描述能力。

考虑到HTTP协议是一个比较“重”的协议，其链接建立，断开以及头部都需要消耗额外的资源，通过这样的协议传输如而日志数据这样的段数据会造成数据传输效率低下的问题，我们的HTTP API还提供了批量写入的功能。一个HTTP请求可以包含多组日志数据。

### 数据清洗组件

该组件主要负责对日志的数据进行清洗，为了能够在云服务平台上实现用户间的数据隔离并且保证用户查询的结果真实可信，我们通过建立Logstash服务（在日志系统中被称为phaser）对日志数据进行加工处理。来自于不同源头的日志数据会由不同的phaser服务进行处理，通过对特定字段进行切分，类型转换，字段重命名等操作，原始的来源不同的日志数据被统一转换为符合设计结构的数据项。同时这个过程中，一些对日志系统来说价值较低或者重复的数据项目，如在有微秒级时间戳和毫秒级时间戳的情况下我们仅保留微秒级时间戳，而诸如系统版本，进程ID，进程优先级等字段在这个过程中将被删除。而对于来自API Server的日志数据来说，数据清洗组件也能够过滤掉不符合系统要求的日志存储请求。这一过程降低了ElasticSearch需要分析和持久化的数据量，从结果上提高了系统的性能。在完成数据清洗之后，该组件将会将日志数据传递给数据缓存组件。

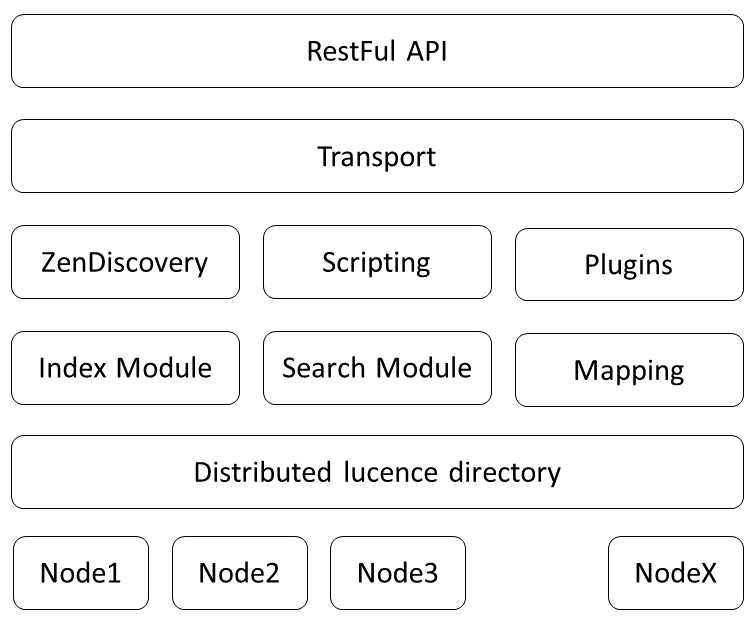
### 数据缓存组件

日志数据通常是短文本数据，其尺寸一般很小，通常有效数据不会超过1KiB。而在一个生产系统上用户应用每秒就可能产生数以千计的日志数据。这样的业务特点要求我们的数据缓存组件能够拥有很高的数据吞吐和处理并发请求的能力。Apache Kafka是由Java和Scala编写的开源流处理平台。它提供了一个统一，高吞吐量，低延迟平台来处理实时数据流。它十分适合用于实现一个大规模，可扩展的消息队列。我们将其集成到日志系统之中并通过第二个logstash服务（在日志系统中被称为shipper）将kafka中缓存的数据“运输”到ElasticSearch之中。

### 数据存储组件

数据存储组件主要由ElasticSearch数据库实现，这也是本系统中最为核心的部分。ElasticSearch是Elastic公司开发的基于Apache Lucene的开源搜索引擎服务器软件。它一方面可以作为一个拥有良好的写入性能的NoSQL数据库使用，另一方面拥有第一无二的强大的近似的实时全文搜索和分析能力。这样的能力使得用户能够对海量的数据进行复杂的查询。经过清洗的数据在被shipper服务输送到ElasticScarch中后，Elasticsearch会对这些数据建立索引，根据配置将数据分发到不同的分片上。同时，Elasticsearch也会对这些分片建立副本以降低结点失效时数据丢失的风险。这些分片和分片的副本会被平衡地分配到不同的数据节点上，从而降低了单节点的压力。这样的设计也为集群的建立提供了可能。当系统压力增加时，只需要将新的数据节点加入集群。整个集群的的性能便能够通过横向扩展几乎无限地扩张。

日志系统默认按日期建立索引，对于每天产生的新的数据利用当天日期取名。为了降低单个索引的压力，来自于各个不同主机的日志也会被安置到不同的索引之中。由于ElasticSearch对内存占用相当大，需要部署ElasticSearch的节点有较高的性能。为了不让ElasticSearch集群影响到用户应用的正常工作，我们选择将其部署到一个单独的Kubernetes集群之中。针对集群管理的需求，我们选择了自己开发自动伸缩工具来管理集群规模。同时利用一些第三方工具来实现对日志数据的冷热分离机制。当有ElasticSearch节点失效时，集群会自行利用副本中的数据来恢复改节点上的内容，同时将负载分配给其他节点，从而保证了节点失效不会造成数据丢失，这一特性在集群收缩过程中也扮演着重要角色。整个ElasticSearch集群内部架构如下图所示，用户的日志数据被分配到了集群中的各个节点中。在查询数据时，ElasticSearch集群将查询请求分配给相关数据所在的数据节点，这些节点分别进行查询统计，最后由主节点完成汇总将结果返回给用户。



“Arch ES”

### 数据查询组件

日志系统提供Http API接口以供用户接入自己的日志分析系统。从而使得应用的开发人员可以集中精力于其业务逻辑而将日志分析的任务交给专业团队。同时MSCP前端UI对日志的访问也需要通过该接口获取数据。该API同MSCP平台的其他API一样是基于Http的Rest风格API。为了减少用户的学习成本，同时兼容一些现有的分析工具。日志查询API直接采用了ElasticSearch的查询语法和返回格式。

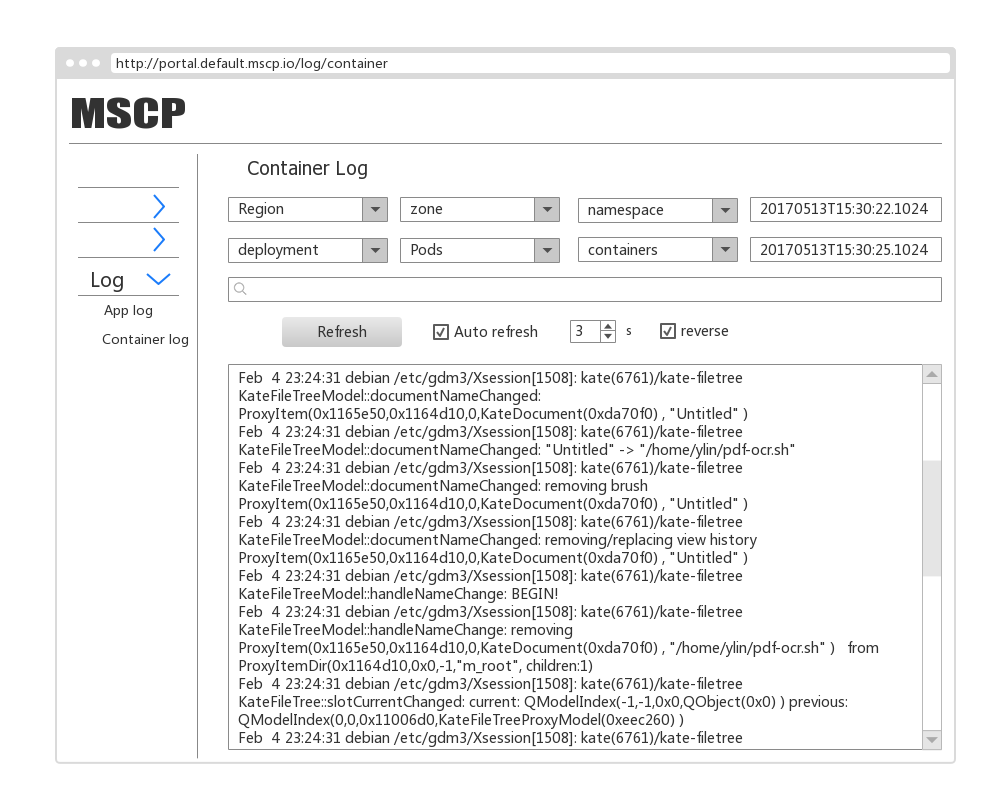
为了提供对第三方系统的集成能力，MSCP平台的API直接暴露在公网之上。为了保护用户的隐私和数据的安全，日志系统需要提供一定的鉴权方式和加密手段来验证客户端身份。由于MSCP平台的API Server会被分布式部署，如果使用传统的基于cookie-session方案，则cookie内仅包含一个session标识符，而诸如用户信息、授权信息等都保存在服务端内存中。用户每次访问时服务端都需要查询数据库来确定用户身份。应用实例之间无法直接共享用户状态。针对这一问题，我们选择了利用JWT token机制实现用户鉴权。用户信息直接包含在此token之中，MSCP平台在签发token时使用一个secretkey对这些用户信息进行数字签名。客户端在每一次向服务器发送请求时都会把这个token作为一个请求参数随同请求数据一同发送给服务器。服务器在收到请求时会然后通过验证数字签名的有效性来验证token。token短期有效，从而降低了token遗失带来的安全隐患，也提高了对抗“回放攻击”的能力。

此外日志系统也为管理员用户提供了查询日志系统状态和更改日志系统设置的能力。 该组件使用Go语言基于BeeGo框架实现。

### 前端UI

日志系统的前端UI作为MSCP系统前端UI的一部分实现。这个UI是一个基于AngularJS实现的单页应用。该应用通过Http API和后台进行交互，交换所需数据。日志查询UI由两部分构成，一是容器日志，负责显示容器中stdout/stderro中的日志信息。二是应用日志，负责显示用户自定义的日志文件和通过Http API推送的日志信息。用户按照自己的需求，按Region，Zone，Namespace，App查询自己部署的应用日志。

页面设计如下图所示：



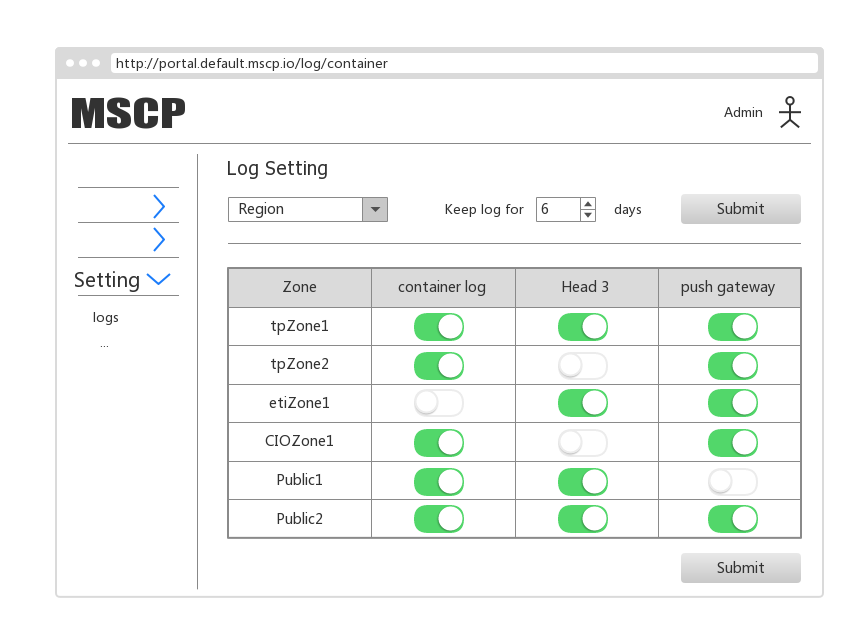
“Container Log pencil”



“App Log pencil”

另一方面，管理员用户可以访问一个日志系统管理页面，从而查询自己的管理的kubernetes集群中日志系统的运行状态。并对日志系统进行一些设置。

页面设计如下图所示：



“Log Setting pencil”

## 本章小结

本章主要完成了MSCP平台上的日志系统的总体设计工作，包括需求分析、在云平台上的部署架构设计、日志系统自身架构和各个模块的等的设计。首先阐述整个系统的设计目标和需要成的工作。完成了整体的架构设计，在此基础之上，将整个系统分为了一下6个组件数据收集组件、数据清洗组件、数据缓存组件、数据存储组件、数据查询和集群管理组件、前端UI。将6大组件分别进行了设计，为实际的系统实现提供了架构依据。

# 日志系统的实现

本文前述章节已经阐述了日志系统的总体设计，包括需求分析、在云平台上的部署架构设计、日志系统自身架构和各个模块的等的设计。本章将介绍各个组件的具体实现。

## 用户鉴权的实现

由于后面日志收集API，日志查询API都需要用到用户鉴权所以我们在此首先介绍我们的用户鉴权的实现。MSCP平台上的用户鉴权机制通过基于jwt的令牌机制来实现。JSON Web Token（JWT）是一个非常轻巧的规范。一个JWT实际上就是一个字符串，它经过Base64编码的头部、载荷与签名三部分组成。

载荷是jwt中最重要的部分，用户的信息都保存在载荷之中。MSCP平台上的jwt载荷（Payload）一般与如下所示的json文本的结构相同：

{  
 "iat": 1516792419,  
 "exp": 1548334419,  
 "user\_id": "AVMSR362",  
 "email": "linyzbj@cn.linyz.net",  
 "role": ["cloud","tenant","user"],  
 "access": [  
 {  
 "method": ["GET","POST","DELETE",...],  
 "url":["[mscp\_api\_url]/[region]/system/log",...],  
 "other":{}  
 }, ...  
 ]  
}

其中最重要的是以下字段：

* exp(expires): 什么时候过期，这里是一个Unix时间戳；
* iat(issued at): 签发时间戳(UNIX时间)；
* user\_id: 用户id
* role: 用户角色数组，包含了用户的扮演的所有角色
* access: 该token所授权的资源信息，下文中会具体介绍。

这个载荷会经过Base64编码，成为jwt的第一部分，下文称编码1。

jwt令牌还需要一个头部，这个头部被用于描述关于该JWT的基本信息，例如其类型以及签名算法等。这也被表示成一个JSON文档。

{  
 "typ": "JWT",  
 "alg": "HS256"  
}

这个头部指明了，这个请求通过JWT鉴权，签名算法是HS256算法。JWT一般可以选择使用非对称的RSA或者对称的HMAC算法对token签名。MSCP系统选择使用HMAC算法来进行签名。这是一种对称加密，加密操作和解密操作使用同一个密钥，其优势在于计算量比较小。缺点在于无法支持第三方签发token的场景。但是MSCP系统上的token都由MSCP API自己签发自己使用，所以HMAC算法即可以满足需求。这个头部经过Base64编码，成为jwt的第二部分，下文称编码2。

MSCP系统在签发token时会使用来自数据库中的一组密钥，通过HMAC-SHA256算法来对上文中编码1和编码2用句号.连接在一起（头部在前）形成的形如编码2.编码1的字符串通过加密和哈希来进行签名，这个签名下文中称为编码3。此密钥由系统自行生成，并且会经常更改以降低泄露风险。

最后将这一部分签名也拼接在被签名的字符串后面，我们就得到了完整的JWT，形如编码2.编码1.编码3

这一组信息会在在HTTP头部的Authorization字段或者直接在请求URL的jwt参数中被传输。我们一般推荐用户将这部分数据作为HTTP头部发送，以避免出现url过长的问题。API Server在收到请求后会通过验证签名来验证token的有效性。

MSCP平台中的用户在第一次使用API或者是当前token已经过期的情况下需要访问GET [mscp\_api\_url]/auth/token\_rootAPI，然后使用自己的邮箱和密码通过basic认证方式来获取一个自己的根token。这个根token不包含任何的access信息，只包含用户的基本信息。之后，用户若要访问其他API则需要使用根token来访问GET [mscp\_api\_url]/authAPI接口并在请求体中加入需要访问的API的信息，也就是上文中的access字段内容来申请实际用于访问API的token。

access字段中method指的是HTTP方法，url是API的路径，other是一个可以包含其他附加信息的结构体。MSCP API Server会在收到这个请求后通过查询数据库中的内容觉得是否签发这个token。如果申请通过则会向用户返回包含授权信息的token，否则返回401授权失败错误。

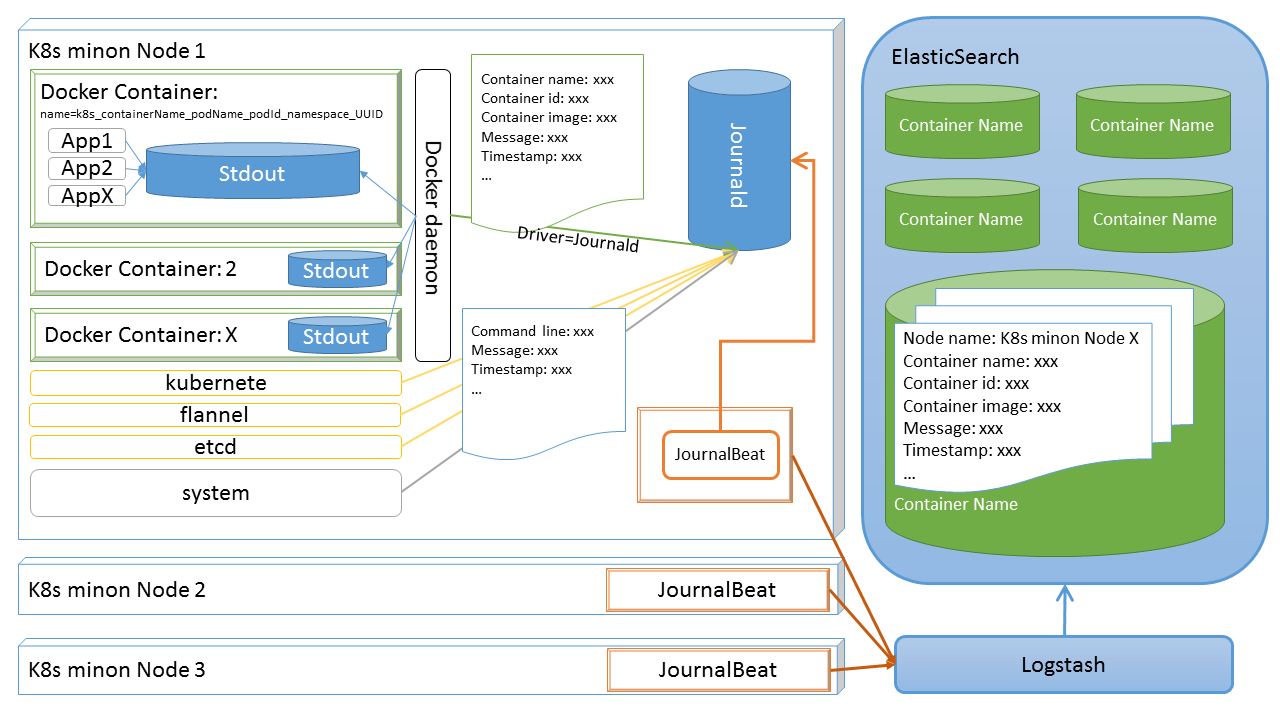
MSCP系统中的每个API都会通过检查用户提供的jwt来决定是否提供服务。

## 数据收集组件的实现

本组件主要包括三部分，第一部分是基于journald和JournalBeat对容器内控制台数据的收集的实现。第二部分是基于FileBeat对容器内日志文件的收集的实现。第三部分是日志推送接口的实现。

### 容器控制台日志的收集

容器内日志控制台日志的收集过程如下图所示：



“Container log collection”

为了让docker能给将容器中控制台中的日志数据发送到journald。首先需要对kubernetes计算节点上运行的docker service进行设置。修改，docker service的定义文件，在我们基于centos7.2的计算节点上该文件是/etc/systemd/system/docker.service 在docker的启动参数中加入或者修改参数--log-driver的值为journald然后通过systemctl restart docker.service重启docker守护程序，以使配置生效。 docker容器发送到journald中的数据可以通过journalctl程序读出，journalctl支持将日志数据组织成json形式输出，其典型结构如下（与本日志系统关联不大的字段已被删除）：

{  
 "\_\_CURSOR": "s=d8478c3862d347e59d28104ba4edb24d;i=144dd0;b=b4d5f19eb0fa4460a1155cdb19fcdcf9;m=f1125097aa;t=54f14690b9edf;x=19866ec227042b69",  
 "\_\_REALTIME\_TIMESTAMP": "1494323963862751",  
 "\_\_MONOTONIC\_TIMESTAMP": "1035394389930",  
 "\_HOSTNAME": "kvm-009660.novalocal",  
 "MESSAGE": "2017-05-09 09:59:23.854320 I | flags: recognized and used environment variable ",  
 "CONTAINER\_NAME": "k8s\_kube-controller-manager.905f3855\_kube-controller-manager-kvm-009660\_kube-system\_7cc96a847ece6fad70a98b9db56bd08d\_01130b8b",  
 "\_SOURCE\_REALTIME\_TIMESTAMP": "1494323963855249",  
 ...  
}

其中MESSAGE字段是应用实际提交的日志信息，\_\_CURSOR字段是一个对日志条目的唯一标记，该标记可以用于在日志存储中定位日志，该字段对帮助日志收集器确定收集数据的起点有重要作用。CONTAINER\_NAME字段提供容器名称。其中由kubernetes管理的应用的名称一般包含了关于日志来源的信息如容器所属的pod，deployment，namespace的名称等。我们将利用这些信息来确定日志来源，为日志入库归档提供依据。\_REALTIME\_TIMESTAMP字段记录了日志数据被journald接收到的时间点。这是一个微秒级的时间戳，标识了自UNIX纪元（1970-01-01 00:00:00 UTC）开始已经过的微秒数。根据linux man7的解释，这个时间点一般略晚于由应用自己汇报的日志产生时间\_SOURCE\_REALTIME\_TIMESTAMP。但是其优势在于，首先这个数值由journald直接生成不会被用户应用“欺骗”，其次其有更好的唯一性，比较不容易出现诸如两条日志拥有同样的时间戳的情况。我们选择这一字段作为日志排序的主要依据。\_HOSTNAME字段提供了日志文件来源的计算节点名称，这一信息将用于决定日志数据在ElasticSearch中的存储位置。

由于日志数据主要在ElasticSearch中存储，journald中存储的数据可以尽量少。我们对journald进行一些额外的配置，以使得journald中的数据在达到一定数量的情况下能够自动滚动删除。journald的配置文件/etc/systemd/journald.conf中的重要配置如下：

[Journal]  
Storage=volatile #为了提高系统性能，日志数据保存在内存中  
SystemMaxUse=1G #日志最多占用1GiB的空间  
SystemKeepFree=4G #至少为其他应用保留4GiB空间  
MaxRetentionSec=7day #最多保留7天的日志数据

在日志数据到达宿主机内部的journald之后我们使用一个被称为JournalBeat的工具将journald中的内容收集起来然后发送给数据清洗组件logstash，logstash的地址会以环境变量的形式被配置到JournalBeat容器内部。JournalBeat作为一个daemonSet被部署在每一个kubernetes集群之中。就结果上来说每一个计算节点都会部署上一个JournalBeat实例。

JournalBeat的重要配置如下：

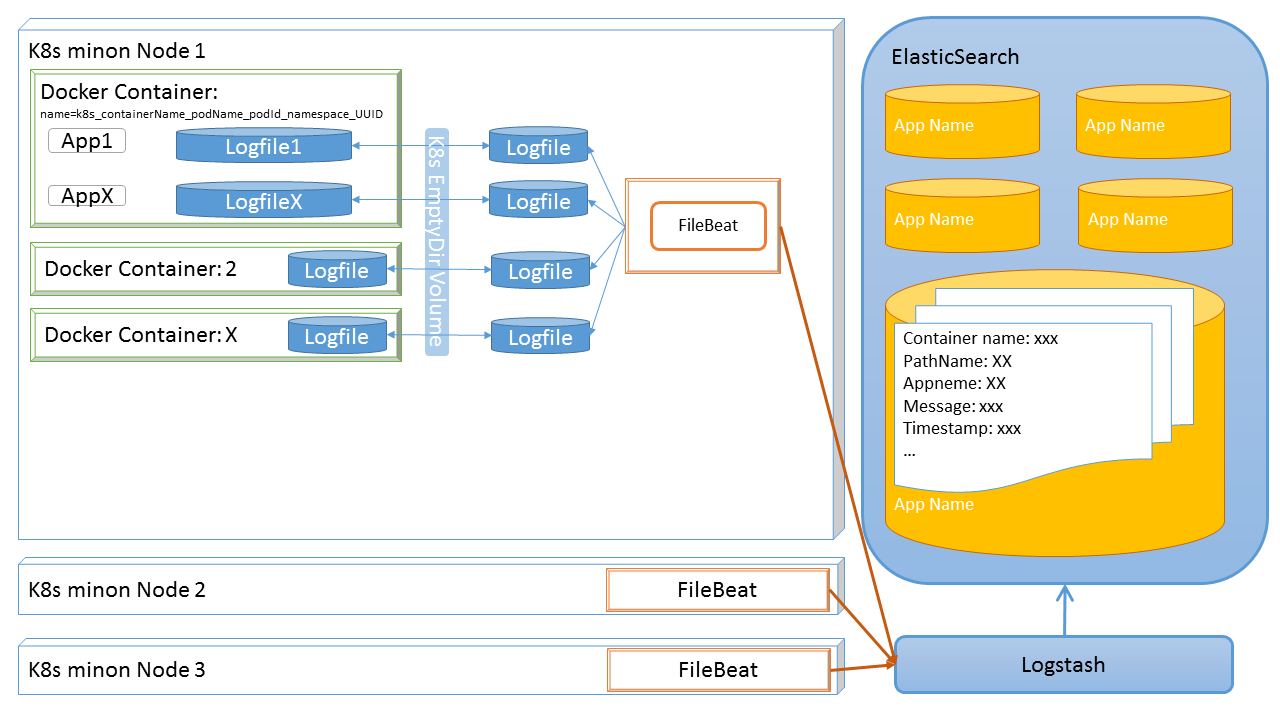
journalbeat:  
 #保存的日志日志坐标处开始收集  
 seek\_position: cursor   
 #如果没有找到保存的坐标则从尾部开始收集，这一般发生在第一次部署时  
 cursor\_seek\_fallback: tail   
 #保存日志位置坐标  
 write\_cursor\_state: true   
 cursor\_state\_file: /data/journalbeat-cursor-state  
 cursor\_flush\_period: 5s  
 clean\_field\_names: true  
 convert\_to\_numbers: true  
 move\_metadata\_to\_field: journal  
name: journalbeat  
#进行一些预处理，只保留来自docker容器的日志，和感兴趣的字段  
processors:  
- drop\_event:  
 when:  
 not:  
 regexp:  
 journal.container\_name: "^.+"  
- include\_fields:  
 fields: [ "journal.container\_name",  
 "journal.hostname",  
 "journal.container\_id",  
 "journal.realtime\_timestamp",  
 "message",  
 "@metadata"  
 ]  
output.logstash:  
 enabled: true  
 hosts: ["${LOGSTASH\_HOST}"]  
logging.level: info

JournalBeat在kubernetes集群上的定义如下（仅重要部分）：

apiVersion: extensions/v1beta1  
kind: DaemonSet  
spec:  
 ...  
 spec:  
 containers:  
 - env:  
 - name: LOGSTASH\_HOST  
 value: #Logstash服务的地址和监听端口  
 image: [dockerReg]/journalbeat-container  
 name: journalbeat-container  
 #为了能够访问带宿主机上的日志数据我们需要将一系列的路径挂载到容器内部。  
 volumeMounts:  
 - mountPath: /var/log/journal  
 name: var-journal  
 readOnly: true  
 - mountPath: /run/log/journal  
 name: run-journal  
 readOnly: true  
 - mountPath: /etc/machine-id  
 name: machine-id  
 readOnly: true  
 - mountPath: /data/  
 name: data  
 - mountPath: /journalbeat.yml  
 name: config  
 subPath: journalbeat.yml  
 volumes:  
 - hostPath:  
 path: /var/journalBeatContainerData  
 name: data  
 - hostPath:  
 path: /var/log/journal  
 name: var-journal  
 - hostPath:  
 path: /run/log/journal  
 name: run-journal  
 - hostPath:  
 path: /etc/machine-id  
 name: machine-id  
 #通过configMap将配置文件提供给journalBeat  
 - configMap:  
 defaultMode: 420  
 items:  
 - key: journalbeat.yml  
 path: journalbeat.yml  
 name: journalbeat-container-config  
 name: config

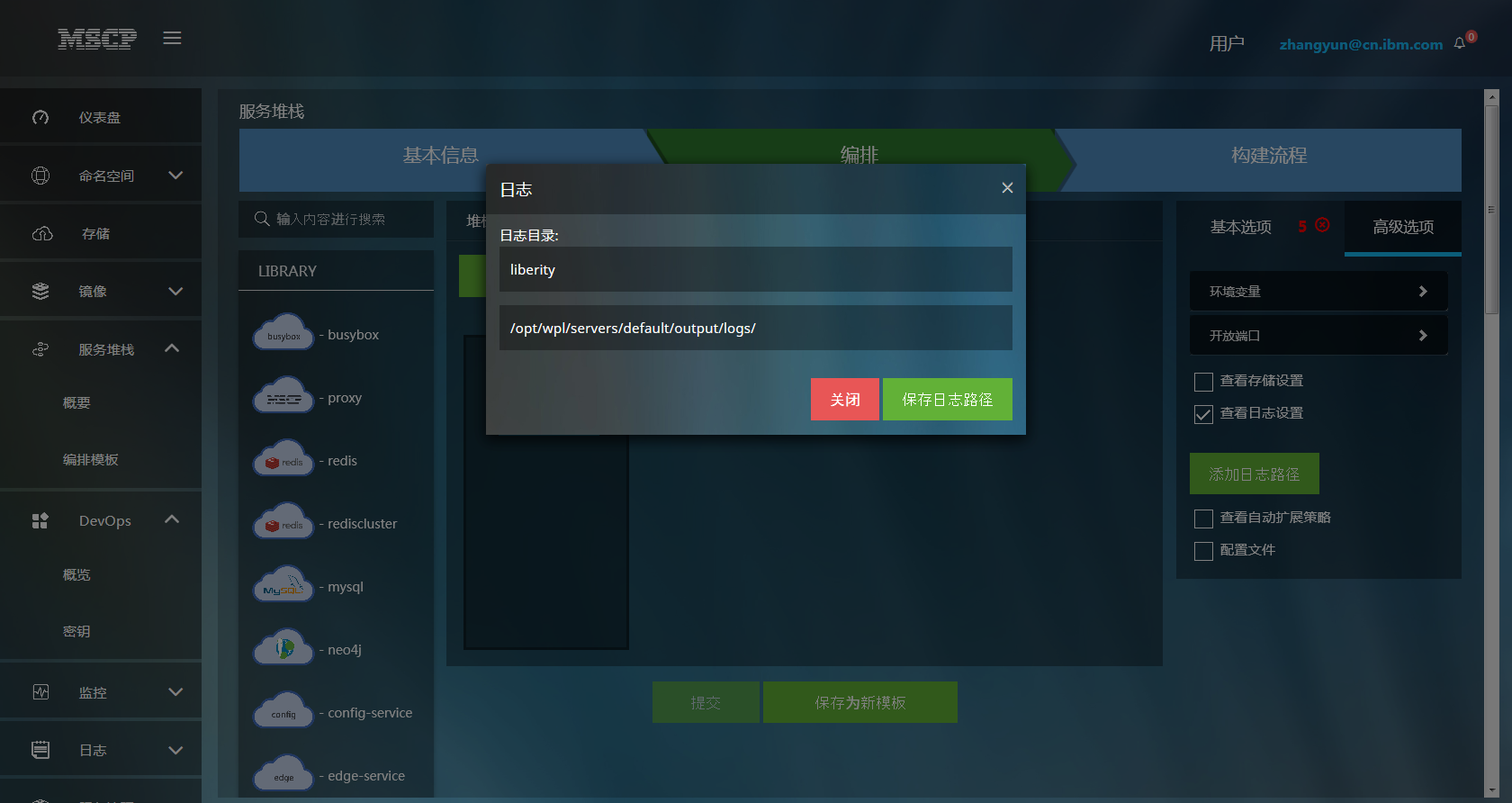
### 容器内日志文件的收集

容器内日志文件的收集过程如下图所示：



“Container logfile collection”

为了能够收集到容器内部的日志文件中的日志数据我们需要将容器内部的日志挂载到容器外，从而让用于收集日志文件的FileBeat工具能够完成对其的访问。同时，在容器消亡后这些日志文件需要一同被回收。用户如果需要使用该功能则在MSCP平台上部署应用时需要在编排应用的阶段，设置日志文件所在的位置。MSCP日志系统将会收集所有位于该位置下的\*.log文件。



“mscp orth log setting”

如以下代码所示，在编排应用的过程中MSCP平台将会，建立一个名为applog-{serviceUUID}-endname-{logDirName}的空目录。通过这个名称中包含的信息，我们能够建立起主机上的日志目录与用户应用间的联系。

//Compose empty dir for log collecting  
func composeLogEmptyDir(appInfo \*Service) (volumes []k8model.Volume) {  
 if !(len(appInfo.LogDirList) > 0) {  
 return nil  
 }  
 logDirList := appInfo.LogDirList  
 for key := range logDirList {  
 volume := new(k8model.Volume)  
 logDir := logDirList[key]  
 volume.Name = fmt.Sprintf("applog-%s-endname-%s", appInfo.ServiceUUID, logDir.Name)  
 volume.EmptyDir.Medium = "{}"  
 volumes[key] = \*volume  
 }  
 return volumes  
}

用户需要提供该日志目录的标题和路径。MSCP系统在部署该应用时会利用kubernetes的EmptyDir存储类型将容器内部的目录和主机上的一个空目录建立联系。kubernetes中的EmptyDir是一个空目录，他的生命周期和所属的Pod是完全一致的。在Pod被建立起来的同时，kubernetes会在宿主机的/var/lib/kubelet/pods/{containerUUID}/volumes/kubernetes.io~empty-dir/建立一个名字和EmptyDir的名字相同的目录，然后将这个目录挂载进容器内部。在容器被销毁时，主机上的这个目录也会同时被销毁。由于EmptyDir一定是一个空目录，所以用户应当注意应为日志数据单独建立一个目录来进行操作而不能与其他数据或程序混杂在同一目录之中。 在每一台宿主机上我们都会通过部署FileBeat组件来收集这些日志文件。

FileBeat是elastic stack生态圈的一部分，它可以收集来自日志文件的日志数据，同时发送到下一级进行下一步的处理。同JournalBeat一样这些FileBeat组件也是作为DaemonSet部署在kubernetes集群上的。这些FileBeat实例的重要配置如下：

filebeat.prospectors:  
- input\_type: log  
 paths:  
 - /k8spods/\*/volumes/kubernetes.io~empty-dir/applog-\*/\*.log  
output.logstash:  
 hosts: ["${LOGSTASH\_HOST}"]  
output.file:  
 path: "/tmp/filebeat"  
 filename: filebeat

他们将会收集/k8spods/\*/volumes/kubernetes.io~empty-dir/applog-\*/目录下的所有.log文件中的日志数据，然后将其发送到指定的logstash中进行处理。FileBeat在kubernetes集群上的定义如下（仅重要部分）：

apiVersion: extensions/v1beta1  
kind: DaemonSet  
spec:  
 template:  
 spec:  
 containers:  
 - env:  
 - name: LOGSTASH\_HOST  
 value: [LOGSTASH\_HOST]  
 image: [dockerReg]/filebeat-app:[version]  
 name: filebeat-app  
 volumeMounts:  
 - mountPath: /k8spods/  
 name: k8spods  
 readOnly: true  
 volumes:  
 - hostPath:  
 path: /var/lib/kubelet/pods/  
 name: k8spods

### 日志推送接口

日志推送接口是一个基于beego实现的API接口，用户可以直接通过这个API向我们的日志系统推送日志数据。这个API充分考虑到了用户自定义日志数据结构的需求。用户除了可以向我们的系统提交日志文本和时间戳等必须的信息职位，还可以以key-value的方式描述一些自定义的字段，从而大大增强了日志系统可以接受的数据的描述能力。我们的HTTP API还提供了批量写入的功能。一个HTTP请求可以包含多组日志数据。

在我们的日志平台上用户向我们推送的日志数据应当有以下的数据结构：

[{"timestemp": 1516792419231221,  
 "appid": "caa0c2f0-1116-495f-8f09-3cd023cc14a2",  
 "type": "applog",  
 "message":"Message to send.",  
 "annotations":[  
 {"key":"level", "value":"info"},...]  
},...]

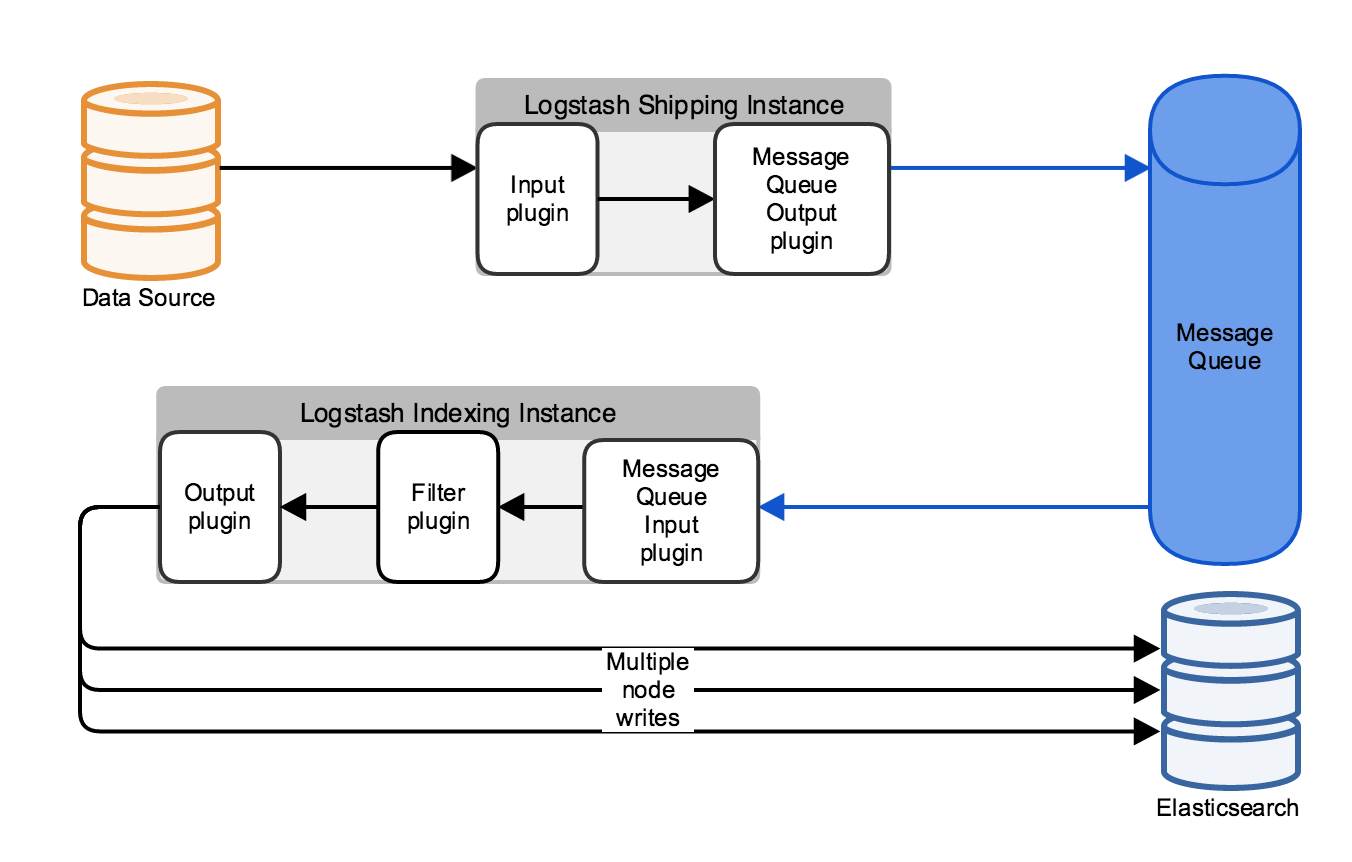
其中timestemp，appid，type以及message是必须的字段，如果这些字段有缺失，则服务器会拒绝这个请求并返回400(bad request)错误。一个请求之中可以包含多个日志数据条目。这个API接口会通过上述的jwt方式验证用户的权限。服务器会检查token中access字段中是否有形如

{  
 "method": ["POST"],  
 "url":["{mscp\_api\_url}/apps/{appid:UUID}/log"],  
}

的记录，如果有的话则认为用户完成了鉴权过程，服务器会将这些日志记录推送入数据缓存服务然后返回202(Accepted)，否则返回403(Forbidden)错误。

## 数据清洗组件

数据清洗组件主要由logstash实现，将接受来自于数据收集组件的日志数据然后推送给数据缓存组件。数据清洗组件的工作流程如下图所示：



“Logstash data flow”

上图中虚线框内部的是logstash进行的操作。其主要过程包含Input，Filter，Output三个关键步骤。这三个过程分别完成了从收集组件接收数据，数据处理和数据向缓存组件输出的工作。

这个数据清洗工作的核心是Filter过程，在logstash的处理过程中，filter完成了数据处理工作。其主要工作是对数据进行字段切分，合并编辑和合并等。由于接收到的JSON数据结构中许多数据字段包含有混合的信息。所以必须对数据进行字段解析，按需分切从而获取需要字段和值。同时将这些值转换成正确的类型。

在logstash中，我们利用Grok对数据字段通过正则表达式进行解析。主要是根据kubernetes的产生的container name进行切分从而获得container\_name，container\_id，controler，podid，namespace，uuid等字段。然后利用muate等工具对字段进行处理整合，从而生成其他需要的字段。

我们还会利用grok工具对container name进行检查，检查其是否满足kubernetes的命名规范。对于诸如etcd，kubelet等不由kubernetes管理的docker容器，他们的容器名称不符合kubernetes的命名规范。我们手工产生这些字段从而使得进入ElasticSearch的数据结构能够统一。

最后，kubernetes还会对每一个Pod产生一个PAUSE容器用于管理同一个pod中的多个容器。这个容器产生的日志数据对我们的系统没有意义。我们将其丢弃。

我们的日志系统在ElasticSearch中每天为来自每一个宿主机的日志数据建立一个index。这个index名字也来自于filter中获取的字段。

logstash的重要配置如下：

filter {  
 if [journal][container\_name] == "kubelet" or [journal][container\_name] =~ "etcd[0-9]+" {  
 grok {  
 match => { "journal[container\_name]" => ["(?<controler>.+)"] }  
 add\_field => {  
 "container" => "%{controler}"  
 "podid" => "%{journal[hostname]}"  
 "namespace" => "default"  
 "uuid" => "00000000-0000-0000-0000-000000000000"  
 "resourceid" => "00000000"  
 "containerid" => "00000000"  
 }  
 }  
 mutate {  
 replace => { "journal[container\_name]" => "%{journal[container\_name]}\_%{journal[hostname]}\_%{journal[container\_id]}" }  
 }  
 } else {  
 grok { match => { "journal[container\_name]" => [  
 "k8s\_(?<container>[a-zA-Z0-9\-]+)\.(?<containerid>[0-9a-f]{6,8})\_(?<controler>[a-zA-Z0-9\-]+)-(?<podid>[a-z0-9]+)\_(?<namespace>[a-zA-Z0-9\-]+)\_(?<uuid>[0-9a-f]{8}-{0,1}[0-9a-f]{4}-{0,1}[0-9a-f]{4}-{0,1}[0-9a-f]{4}-{0,1}[0-9a-f]{12})\_(?<resourceid>[0-9a-f]{8})"  
 ] } }  
 }  
 # Drop event from "pause" containers   
 if [container] == "POD" {   
 drop {}   
 }   
 mutate { add\_field => { "pod" => "%{controler}-%{podid}" } }  
 #create index name  
 mutate{ add\_field => { "index" => "logstash-%{[@metadata][beat]}-%{journal[hostname]}" } }  
 mutate{ lowercase => [ "index" ] }  
 mutate{ gsub => [ "index", "[^a-z0-9\-\_]", "\_" ] }  
}

由于数据传输速度和数据处理速度远远大于ElasticSearch落盘的速度。数据会在清洗组件形成堆积，这是我们加入数据缓存组件的根本原因。通过加入消息队列kafka我们可以在数据处理过程结束以后，logstash会将数据推送到作为数据缓存组件的kafka集群之中。从而增加对数据高峰的处理能力。

在我们的日志系统上，logstash被作为一个deployment部署在admin zone上，我们可以利用kubernetes来管理logstash的实例数量，从而管理数据清洗组件的性能。

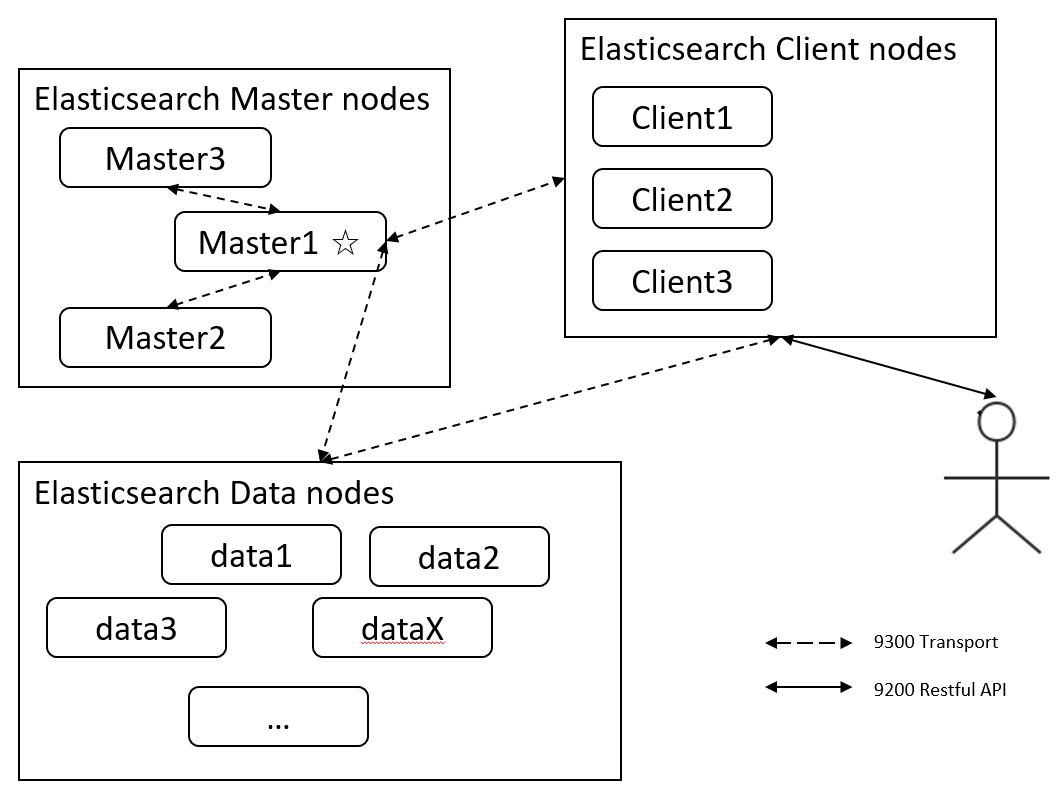
## 数据缓存组件

我们使用kafka来实现日志数据的缓存。Apache Kafka是由Java和Scala编写的开源流处理平台。它提供了一个统一，高吞吐量，低延迟平台来处理实时数据流。它十分适合用于实现一个大规模，可扩展的消息队列。我们将其集成到日志系统之中并通过第二个logstash服务（在日志系统中被称为shipper）将kafka中缓存的数据“运输”到ElasticSearch之中。kafka作为MSCP平台提供的开箱即用服务直接部署。 shipper的脚本极为简单，只是简单地将kafka中的数据搬运到Elasticsearch中：

input { kafka{  
 bootstrap\_servers => "scale-kafka-service:9092"  
 topics => "logstash-journal"  
 codec => "json" }  
}  
filter {}  
output { elasticsearch {  
 hosts => "http://elasticsearch.elk.svc.cluster.local:9200"  
 manage\_template => false  
 index => "%{index}-%{+YYYY.MM.dd}"  
 document\_type => "%{[@metadata][type]}" }  
}

## 数据存储组件

我们的日志系统采用ElasticSearch来实现我们的数据存储组件。这个组件实际上还承担着日志检索的工作所以，实际上是我们整个日志系统的核心。ElasticSearch作为一个基于Lucene的搜索引擎，其性能优异，功能强大。其已经在实际的生产中得到了广泛的认可和应用。下图展示了ElasticSearch集群的部署架构。



“Arch ES”

ElasticSearch集群由多个ElasticSearch实例构成。每一个ElasticSearch实例都可以实现三种主要功能：作为Master节点管理集群状态，作为Data节点实现数据存储以及作为client节点提供HTTP API访问接口。一个ElasticSearch实例在集群中可以实际承担这其中的一种或多种功能。

ElasticSearch集群一般都会选择一个启用了master功能的实例作为master节点，除了该节点之外集群中的其他节点都是slave节点。其他启用master功能的ElasticSearch实例都从master节点中备份集群的信息。以备Master节点失效时恢复集群状态。站在用户的角度上来看，master节点在ElasticSearch集群中并不占据一个重要的位置。这同用户一般对其他其它系统的认识(比如传统的分布式数据库系统)不同。实际上用户要知道主节点在哪里，甚至不需要知道自己需要访问的数据在哪里；所有操作都可以被分派到任意一个节点，ElasticSearch集群内部会自动将这些请求发送到各个data节点上来完成。在必要的时候下，任何一个节点都可以并发地把查询计算workload分发到其它的节点上，然后合并来自于各个不同节点的查询结果。最后再返回给用户一个完整结果集。所有这些工作都不需要通过主节点的协调和汇总，data节点之间能够通过P2P的方式通信，自动地协调集群的工作。

在我们的日志平台上，为了实现更好的可扩展性和获得性能ElasticSearch节点都只扮演Master，Data或者Http API server中的一者。

按照ElasticSearch的默认配置，节点之间会通过多播(multicast)寻找集群中的其它节点，并与之建立连接。然而在kubernetes集群中，容器之间通过flannel虚拟网络相互通信，这种虚拟网络不支持多播通信。我们需要配置ElasticSearch集群中的节点使之通过单播的方式实现集群发现。在此过程中我们首先要建立起一个kubernetes service，这个service中包含了所有的ElasticSearch容器。下面是这个service在kubernetes中的定义：

apiVersion: v1  
kind: Service  
metadata:  
 name: elasticsearch-discovery  
 labels:  
 component: elasticsearch  
 role: master  
spec:  
 selector:  
 component: elasticsearch  
 role: master  
 ports:  
 - name: transport  
 port: 9300  
 protocol: TCP

ElaticSearch中的各个节点在试图通过这个service访问其他节点集群管理端口（9300）时请求会被等概率地分发到各个节点之上。节点之间会通过这一特性渐渐地相互发现。ElasticSearch集群有时会发生脑裂的问题。这是由于ElasticSearch集群启动过程中多个Master节点分别获取Data节点建立集群造成故障。常发于ElasticSearch安装部署过程中和集群发送故障后自动恢复的过程。可造成出现Log无法查询等问题。为了避免这个问题的发生，我们可以将discovery.zen.minimum\_master\_nodes的值设置到master节点数量的二分之一以上。这个参数所控制的是一个master节点在建立一个集群之前需要联合到的具有master节点资格的节点的最小数量，然后才能在集群中做操作。这样的设计可以保证最多只有一组节点达到足够建立集群的规模。从而在避免master过早停止节点发现过程同时也能够保证整个集群在少量主节点失效时依然能够提供服务，真正实现了高可用部署。

cluster.name: "elasticsearch"  
network.host: 0.0.0.0  
# discovery service name in k8s  
discovery.zen.ping.unicast.hosts: elasticsearch-discovery  
# should > nb\_node/2  
discovery.zen.minimum\_master\_nodes: 2   
xpack.security.enabled: false  
xpack.monitoring.enabled: false  
node.master: false # true for master node  
node.data: false # true for data node  
http.enabled: false # true for HTTP API server

我们的系统中为了提高日志落盘性能，使用了ElasticSearch中新的MMap文件系统来存储日志数据。这种存储类型使用Apache Lucene MMapDirectory实现类。它使用linux中的mmap功能完成文件随机读写操作。它将原本直接存储到文件磁盘的文件数据映射到同样尺寸的虚拟内存空间中。由于没有涉及到任何的锁操作，在多线程共同存取数据文件时程序就具有可伸缩性了(这里可伸缩性是指当增加线程数量时，程序的处理能力相应增加)。我们使用mmap来读取索引文件时，在linux系统看来，该文件已被缓存(文件会被映射到虚拟内存中)。因此，ElasticSearch在从Lucene索引中读取文件时，数据可以直接从内存缓存中被读到，读取速度会快很多。这基本就是允许ElasticSearch直接操作I/O缓存，索引文件的读写效率会提高很多。

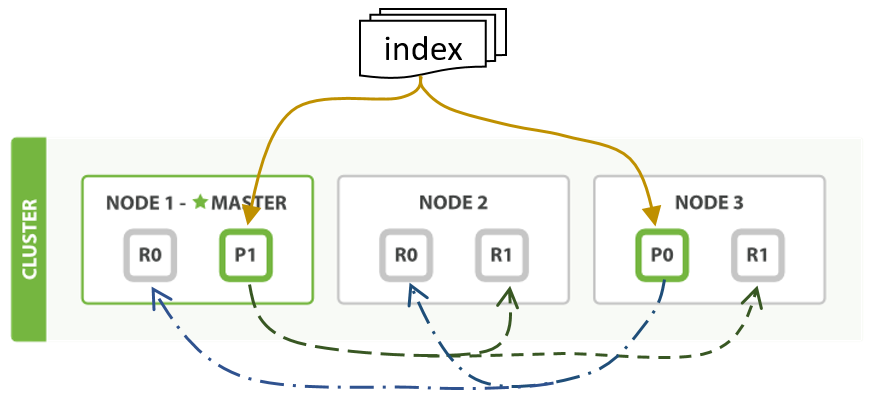
由于Mmap要求每一个线程都能够访问足够大的内存，这一数值受两个配置的影响。首先是JAVA Heap的大小，我们将JAVA Heap的最大和最小数值设为一致，从而JVM可以在启动时就获取做够的内存。其次linux系统的max\_map\_count参数决定了允许一个进程在VMAs(虚拟内存区域)拥有最大数量。linux发行版的max\_map\_count参数配置一般较低。我们在启动实际的ElasticSearch实例之前首先启动一个busybox容器来配置这个参数。

ElasticSearch在kubernetes上的定义如下：

apiVersion: extensions/v1beta1  
kind: Deployment  
spec:  
 replicas: 2  
 selector:  
 matchLabels:  
 component: elasticsearch  
 role: client/master/data  
 template:  
 metadata:  
 annotations:  
 pod.beta.kubernetes.io/init-containers: '[{  
 "name": "sysctl",  
 "image": "busybox",  
 "imagePullPolicy": "IfNotPresent",  
 "command": ["sysctl", "-w", "vm.max\_map\_count=262144"],  
 "securityContext": {"privileged": true}  
 }]'  
 spec:  
 containers:  
 - name: es-client/data/master  
 securityContext:  
 capabilities:  
 add:  
 - IPC\_LOCK  
 image: [dockerReg]/elasticsearch:6.1.3  
 env:  
 - name: "ES\_JAVA\_OPTS"  
 value: "-Xms2048m -Xmx2048m"  
 ports: ...  
 volumeMounts: ...  
 - name: config  
 mountPath: /usr/share/elasticsearch/config/elasticsearch.yml  
 subPath: elasticsearch.yml  
 volumes:  
 - name: config  
 configMap:  
 name: elasticsearch-config  
 items:  
 - key: client/master/data.yml  
 path: elasticsearch.yml

ElasticSearch将Logstash发送过来的数据建立索引并存储起来。

ElasticSearch的储存模型如下图所示：



“ES Stoage”

根据上图所示，ElasticSearch中的一个索引(index)可以被理解为传统关系型数据库中的一个数据库，所有数据都保存在这些索引之中。一个索引一般可以被分为多个分片，每一个分片都存储了索引中数据的一部分。一个分片一般有一主多副，多个副本。在默认配置下ElasticSearch之中的一个索引被分为5个分片，这5个分片分别拥有一个副本。这样的配置可以保证在有一个数据节点失效的情况下数据不丢失。

分片会被分配到各个不同的数据节点上，上图中可以看到三个ElasticSearch节点中各有一些主分片和一些分片副本。这些分片共同构成了一个完整的索引的数据。

考虑到设备的存储能力有限，一般需要对ElasticSearch中的数据作轮转处理，定期删除集群中的旧数据。ElasticSearch中数据的删除方式一般是使用HTTP API对集群发出指令，如调用DELETE '[ES\_API]/old-index'可以删除old-index索引。从ElasticSearch中删除索引的速度很快，一般与删除的数据量无关。日志系统中的轮转策略一般由用户定义，根据实际需求选择如何处理已经收集到的数据。在我们的日志系统中管理员用户可以配置数据存储的时间。

在我们的日志系统上ElasticSearch的数据轮转采用了curator工具，该工具主要用于对ElasticSearch中的索引进行处理，定期删除旧的索引。curator工具所用的脚本如下所示：

actions:  
 1:  
 action: delete\_indices  
 description: "Delete selected indices"  
 options:  
 timeout\_override: 300  
 continue\_if\_exception: True  
 filters:  
 - filtertype: pattern  
 kind: regex  
 value: '^logstash-journalbeat-(([a-zA-Z]|[a-zA-Z][a-zA-Z0-9\-]\*[a-zA-Z0-9])\.)\*([A-Za-z]|[A-Za-z][A-Za-z0-9\-]\*[A-Za-z0-9])-.\*$'  
 - filtertype: age  
 source: field\_stats  
 direction: older  
 unit: days  
 unit\_count: [NUMBER\_OF\_DAYS\_TO\_DELETE]  
 field: '@timestamp'  
 stats\_result: max\_value

该脚本首先通过正则表达式过滤需要删除的索引名。其中的actions[1][filters][1][unit\_count]字段控制了数据的寿命，创建时间早于这个时间的索引将会被删除。这个值可以通过集群管理API更改。curator会通过kubernetes中的cronJob每天运行一次，每次运行时curator都会使用最新配置。

## 数据查询和集群管理组件

数据查询和集群管理组件是一组rest风格的api，和其他的MSCP API一样，这个api通过基于jwt的用户鉴权服务完成rbac。

在完成鉴权之后查询日志的请求会被原样转发给ElasticSearch集群。然后将ElasticSearch集群的回复原样转发给用户。通过这样的操作我们的日志系统实现了查询接口以及语法和ElasticSearch相兼容。从而降低了用户的学习成本也方便了用户将自己的日志分析服务接入我们的日志系统。

我们的日志系统向管理员用户提供了查询日志系统状态和管理日志系统配置的接口。这些配置接口被安排在[mscp\_api\_url]/[region]/system/log路径下。

GET [mscp\_api\_url]/[region]/system/log/status接口提供了当前日志系统的状态，其返回的数据结构如下：

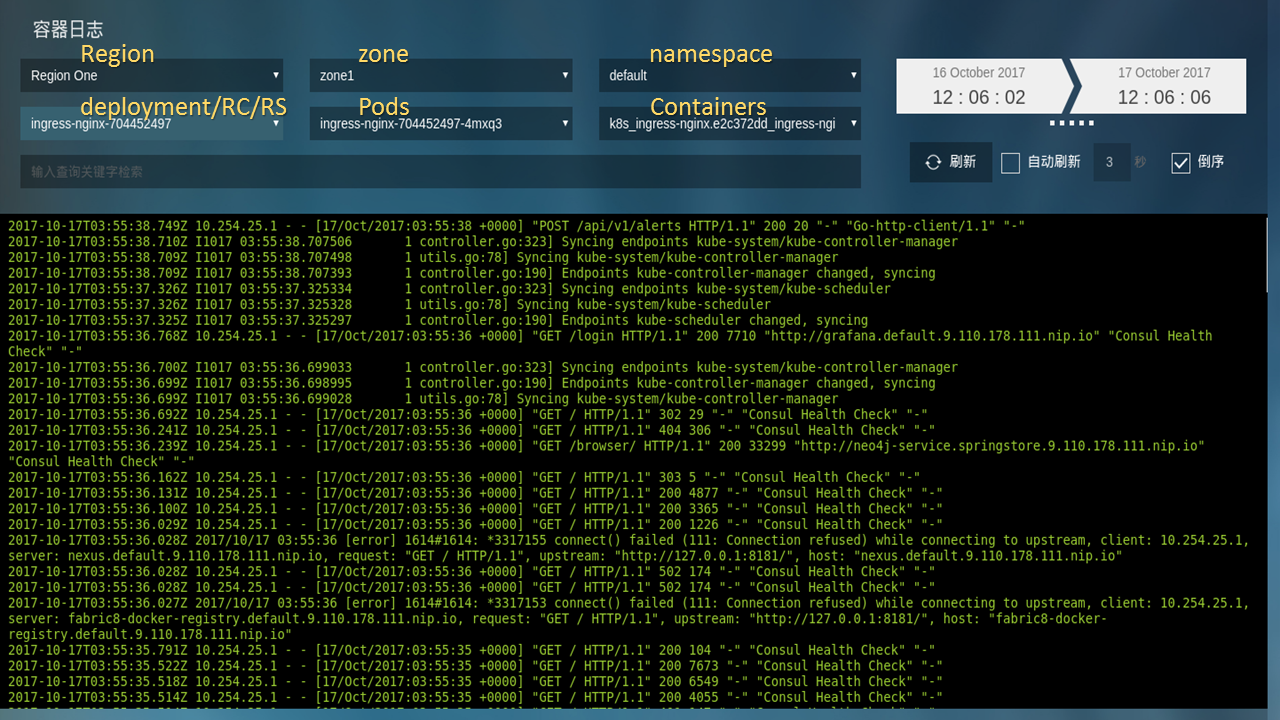
[{"status": "green/yellow/red",  
 "timestemp": 0,  
 "rotate":{  
 "enabled": true,"age": 7  
 },  
 "collector": {  
 "journalbeat": {  
 "enabled": true,  
 "status": {  
 "scheduled": 10, "up": 10  
 },  
 "instances": [{"node": "string",  
 "pod\_id": "string","status": "up"  
 }]  
 },  
 "filebeat": {"enabled": true,"status": {...},"instances": [...]  
 },  
 "http\_gateway": {"enabled": true,"status": {...}}  
 },  
 "parser": {"status": {...}},  
 "shipper": {"status": {...}},  
 "kafka": {  
 "health\_check": true,  
 "zookeeper": {"status": {...}},  
 "broker": {"status": {...}}  
 },  
 "elasticsearch": {  
 "status": {  
 "master": {  
 "scheduled": 3, "up": 3  
 },  
 "client": {...},"data": {...}  
 },  
 "health": {  
 "cluster\_name": "elasticsearch",  
 "status": "green",  
 "timed\_out": false,  
 "number\_of\_nodes": 12,  
 "number\_of\_data\_nodes": 10,  
 "active\_primary\_shards": 2006,  
 "active\_shards": 4012,  
 "relocating\_shards": 0,  
 "initializing\_shards": 0,  
 "unassigned\_shards": 0,  
 "delayed\_unassigned\_shards": 0,  
 "number\_of\_pending\_tasks": 0,  
 "number\_of\_in\_flight\_fetch": 0,  
 "task\_max\_waiting\_in\_queue\_millis": 0,  
 "active\_shards\_percent\_as\_number": 100  
}}}]

POST [mscp\_api\_url]/[region]/system/log接口可供管理员修改日志系统的配置。用户可以将上面数据结构中的部分配置根据自己的需求进行修改，然后将这个修改过的json文档作为request body发送。用户可以通过此接口启用或停用日志轮转功能，改变ElasticSearch集群的配置，启用或停用某种日志收集器等。

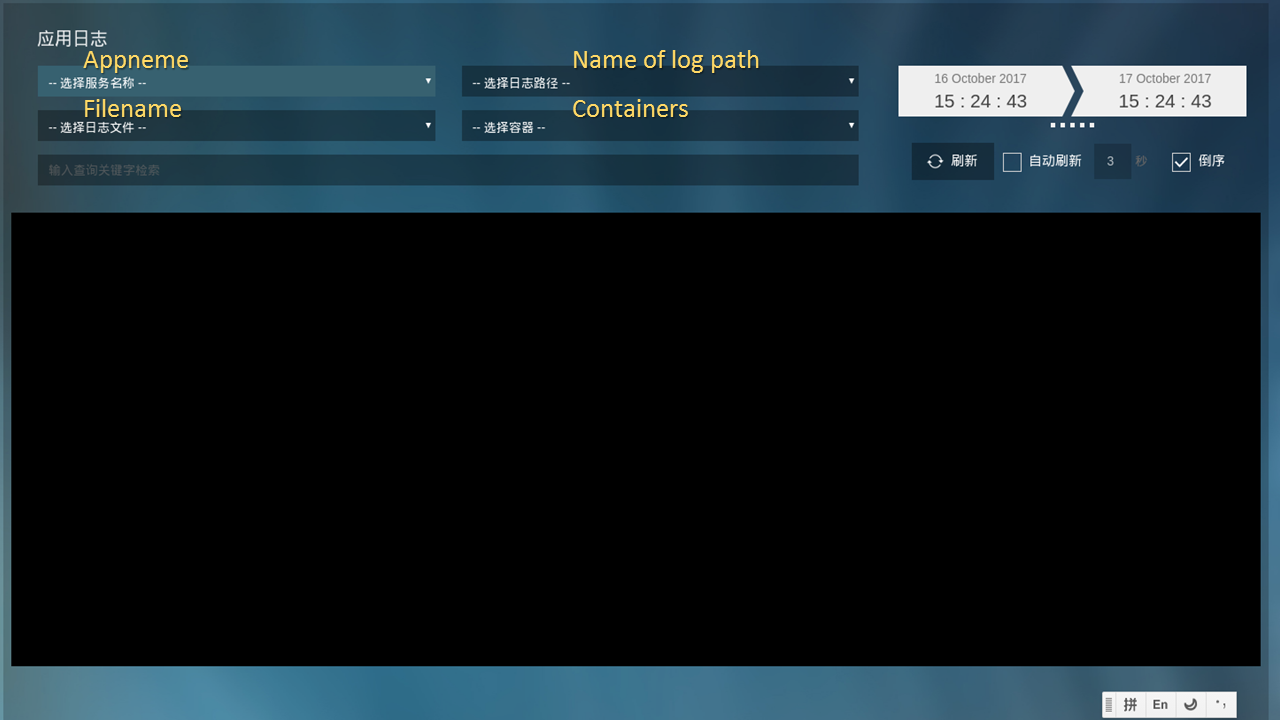
## 前端UI

日志系统的前端UI作为MSCP系统前端UI的一部分实现。这个UI是一个基于AngularJS实现的单页应用。该通过Http API和后台进行交互，交换所需数据。日志查询UI由两部分构成，一是容器日志，负责显示容器中stdout/stderro中的日志信息。二是应用日志，负责显示用户自定义的日志文件和通过Http API推送的日志信息。用户按照自己的需求，按Region，Zone，Namespace，App查询自己部署的应用日志。

线上部署的系统上的前端UI如下图所示：



“Container Log”



“App Log”

## 本章小结

本章主要完成了MSCP平台上的日志系统的设计工作完成后的具体实现，云平台上的部署以及包括数据收集组件、数据清洗组件、数据缓存组件、数据存储组件、数据查询和集群管理组件、前端UI在内的6个组件的实现。对各个组件的实现进行了详细的阐述，综合前面介绍的关键技术，将整个系统继续搭建和集成。展示了数据收集，处理，存储，展示以及轮转的整个生命周期的过程。实际上，在真实生产环境上运行的ElasticSearch集群还有一组服务用来维持其状态，同时根据请求数量自动地改变集群规模，但是这个部分不在本文讨论范围以内。

# 日志系统测试

本章主要是针对前述的系统实现，对整个日志系统进行功能测试和性能测试。测试各个组件的工作性能，集成后的整体效果，各个API的工作情况，前端展示和使用等各方面的功能，并发现其中可能存在的问题，或者需要优化的技术问题，对整体细节方面进行完善。

## 测试环境

本节首先对软硬件配置进行介绍。

### 软件环境

日志系统作为MSCP平台的一部分存在，MSCP平台各模块所用到的与本系统相关的软件如下：

|  |  |  |
| --- | --- | --- |
| 模块 | 软件 | 版本 |
| MSCP | kubernetes | 1.5.3 |
| flannel | v0.9.0-rc1 |
| portal | beego | 1.9.0 |
| angularJS | 1.6.2 |
| 存储 | ceph | 11.2.0 |
| 日志 | ElasticSearch | 5.2.2 |
| Logstash | 5.2.2 |
| kibana | 5.2.2 |
| journalBeat | 5.2.2 |
| fileBeat | 5.2.2 |
| kafka | 0.10.1.0 |
| zookeeper | 3.4.9 |

### 硬件环境

受测试环境限制本测试在单region环境下进行。该region中有一个运行了ElasticSearch集群和其他MSCP系统组件的admin zone和15个运行模拟用户负载的worker zone。admin zone由三台28核56线程，256GiB内存，拥有SSD存储的高性能服务器构成。每个worker zone分别由20台来自于云计算提供商的4核4线程，8GiB内存的KVM虚拟机构成。整个测试平台共有203台计算节点可用。实际测试时系统实际可用的计算节点数在183-195之间波动。

## 系统安装

### 安装ES集群（文件位于./kafka/elasticsearch）

以下操作在Namespace ELK中进行

#### 导入configmap

kubectl --namespace=elk create -f elasticsearch-config.yaml

#### 创建service

kubectl --namespace=elk create -f es-discovery-service.yaml  
 kubectl --namespace=elk create -f es-service.yaml

#### 部署容器

1. 部署master节点

kubectl --namespace=elk create -f es-master-deployment.yaml

可将rep数量改到1，内存（ES\_JAVA\_OPTS环境变量）改到2GiB（不可低于1GiB）

1. 部署client节点

kubectl --namespace=elk create -f es-client-deployment.yaml

可将rep数量改到1，内存改到2GiB（不可低于1GiB）

1. 部署data节点 在部署data节点前请保证master和client节点均已部署完成，可用 curl -s http://elasticsearch:9200/\_cluster/health检查“number\_of\_nodes”是否和client+master数量一致，number\_of\_data\_nodes是否为0

kubectl --namespace=elk create -f es-data-deployment.yaml

可将rep数量改到2，内存改到6GiB（不可低于2GiB）

### 部署kafka（文件位于./kafka/kafka）

1. 部署zookeeper

kubectl --namespace=elk create -f zookeeper.yaml

1. 部署kafka

kubectl --namespace=elk create -f kafka.yaml

### 部署logstash（文件位于./kafka/logstash）

1. 导入configmap

kubectl --namespace=elk create -f logstash-config.yaml  
 kubectl --namespace=elk create -f logstash-ship/logstash-pipline-config-ship.yaml  
 kubectl --namespace=elk create -f logstash-direct/logstash-pipeline-direct-config.yaml

这里请检查hosts => “http://elasticsearch.elk.svc.cluster.local:9200”一行是否合适，此处应指向ES API。

1. 建立service

kubectl --namespace=elk create -f logstash-service.yaml

1. 建立容器 rep数量均可降为1

* Logstash for container logs

kubectl --namespace=elk create -f logstash-container/logstash-deployment.yaml

* Logstash for app log files

kubectl --namespace=elk create -f logstash-app/logstash-deployment-app.yaml

* Logstash shipper 用于从kafka中向ES搬运数据

kubectl --namespace=elk create -f logstash-ship/logstash-ship-deployment.yaml

### 部署Beats（文件位于./kafka/beats）

1. JournalBeat

kubectl --namespace=elk create -f journalbeat-container-config.yaml  
 kubectl --namespace=elk create -f journalbeat-container.yaml

此处请将环境变量LOGSTASH\_HOST的值和节点的IP相匹配

1. FileBeat

kubectl --namespace=elk create -f filebeat-app.yaml

此处请将环境变量LOGSTASH\_HOST的值和节点的IP相匹配

## 性能测试

### ElasticSearch性能测试

#### 测试方法

通过journalbeat和logstash不断以最大速度向ElasticSearch写入数据从而测试ElasticSearch的写入性能。数据来源是系统journald中积攒的系统日志。根据经验，journalBeat，logstash产生日志的速度大大高于ElasticSearch的落盘速度。

#### 测试基准

ElasticSearch 配置：

* 1个Master Node，JVM：heap size设置为2048MB
* 1个Client Node，JVM：heap size设置为2048MB
* 2个Data Node，JVM：heap size设置为2048MB

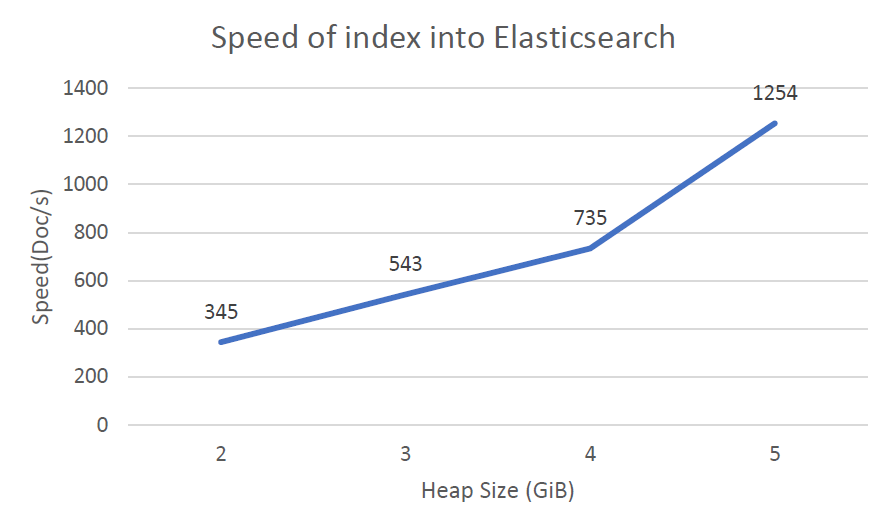
Indice 配置：5分片1备份

基准数据：ElasticSearch落盘速度为172 Doc/s

对ElasticSearch Data节点的Log发现有大量超过1s 的full gc发生。进一步分析其gc log发现gc耗时大约占了总运行时间的40%（1267.59s/3786s）由此分析，我们认为系统的瓶颈在于ES Data的内存不足。

#### 纵向扩展

我们在维持Data节点数量为2不变的情况下提高JVM heap size测试结果如下



JVM heap size

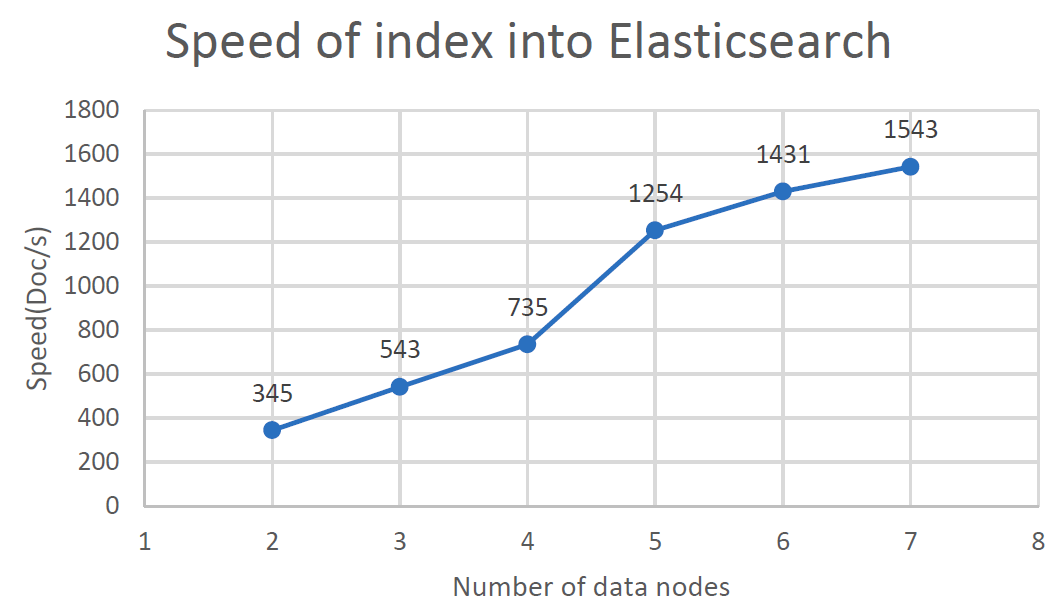
我们可以看到，随着数据Heap size的提高Elasticsearch集群的写入性能不断提高。

通过对Gc log的分析我们发现，在Heap size到达6Gb时gc耗时<10%（321.03s/3723s），运行过程中没有发生full gc， heap占用峰值被良好的控制在80%左右，平均Heap占用维持在3.7GiB左右。我们由此认为6GiB的Heap Size是一个合适的值。

在Heap size达到8GiB之后，虽然集群性能依然能得到少许提高，但是根据对Gc log的分析我们发现ElasticSearch Data节点对Heap的平均使用只有4GiB左右。我们认为这样的操作是得不偿失的。

#### 水平扩展

在确定6GiB的Heap Size为合适值后我们尝试对ElasticSearch Data节点进行水平扩展，提高ElasticSearch Data节点的数量来提高性能。下图展示了我们的测试结果：



nb\_node

由上图可见在Data Node的个数达到5时集群性能有了大幅提升，我们认为这是由于分片的个数被设置为5造成的。根据Elastic.io提供的指南，Data节点的个数在被设置为分片数量的倍数时比较合适。所以在这里我们在安装脚本中将ElasticSearch Data 节点的个数默认设置为5。

#### 其他可以提高Log系统性能的方案

1. 降低log Document的大小，在我们的测试中Log Document的大小大约为2KiB其中包括了30种以上的属性。而log信息则包含在其中一个叫message的属性中。这样的Document效率比较低，可以考虑删除部分不会用到的属性。
2. 调整ElasticSearch中的Mapping设置对一些字段不进行分词等操作，降低集群的压力。
3. 调整ElasticSearch中分片和备份的数量，降低备份数量有利于提高性能但是ElasticSearch集群将会丧失节点失效时自动恢复和Scale In的能力。
4. 在Logstash和ElasticSearch集群中加入Message queue。

### Kafka性能测试

#### 测试方法

利用kafka自身提供的性能测试工具和grafana可视化平台测试kafka集群的能力。 #### 硬件配置

* 集群服务器数量：8
* CPU核数：8
* RAM数量：32G
* 磁盘：200GB机械磁盘

#### 测试场景规划

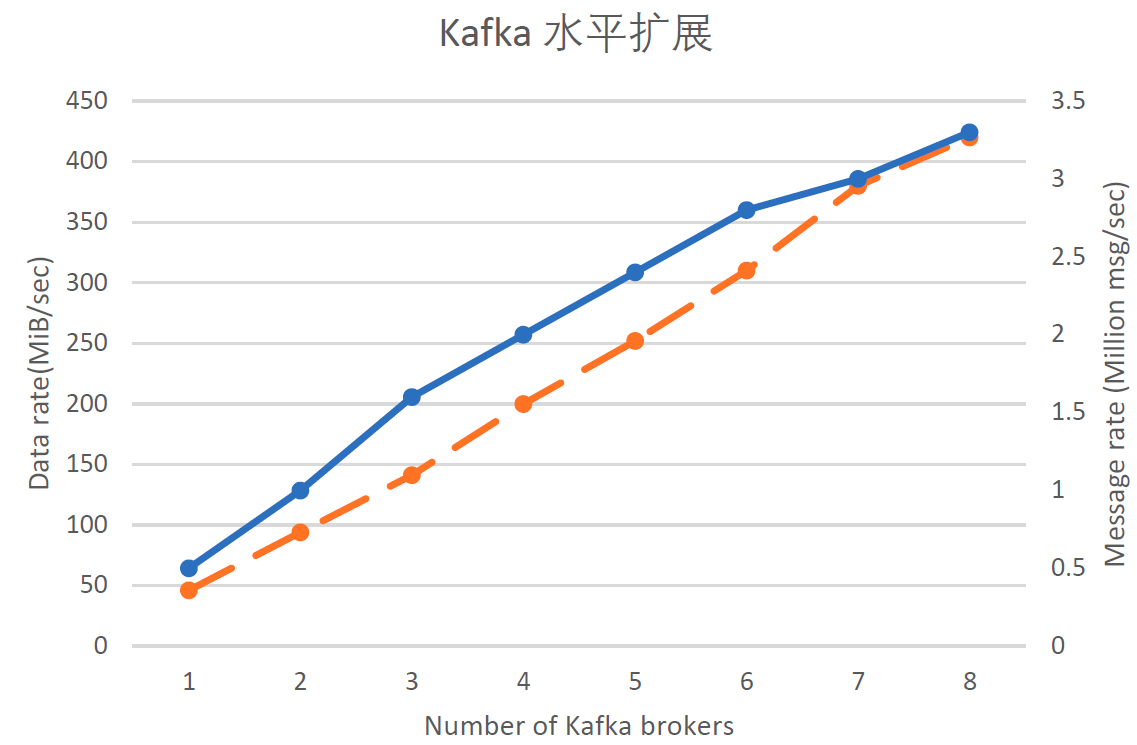
* 最大消息数：270Million
* 最大总数据量：42GB
* 总计运行时间：120s
* 最大Broker数：8
* 客户端最大线程数：9

#### 测试结果

在每条消息1KiByte的场景下最大每秒消息数达到了大约3.3Million Msg/Sec，数据吞吐率最高达到了420MiB/Sec.

#### 水平扩展测试

我们同时测试了kafka集群的水平扩展能力。在增加broker数量的情况下kafka的数据吞吐能力得到了明显提高。



sacle\_kafka

## 功能测试

### Journald中的日志收集

Journald中的日志收集测试情况如下表所示：

|  |  |
| --- | --- |
| 测试项目： | 日志收集 |
| 测试分项： | Journald中的日志收集 |
| 测试目的： | 测试日志系统是否能够顺利收集journald中的日志 |
| 预置条件： | 日志系统部署完毕，运行正常。journalbeat已经安装部署 |
| 测试步骤： | 1）改变logstash中的配置，使之将日志处理状态通过console显示出来  2）部署作为监控用途的kibana服务  3）重新启动journalbeat容器  4）查看日志收集情况 |
| 预期结果： | 1）journalbeat的console中有关于日志收集和发送的信息  2）logstash中的console中能够查询到关于日志收集和发送的信息  3）kibana中能够看到收集到的日志 |
| 测试结果： | Pass |

### Logfile中的日志收集

Logfile中的日志收集测试情况如下表所示：

|  |  |
| --- | --- |
| 测试项目： | 日志收集 |
| 测试分项： | 日志文件中的日志收集 |
| 测试目的： | 测试日志系统是否能够顺利收集日志文件中的日志 |
| 预置条件： | 日志系统部署完毕，运行正常。filebeat已经安装部署 |
| 测试步骤： | 1）改变logstash中的配置，使之将日志处理状态通过console显示出来  2）部署作为监控用途的kibana服务  3）部署一个测试用应用  4）查看日志收集情况 |
| 预期结果： | 1）filebeat的console中有关于日志收集和发送的信息  2）logstash中的console中能够查询到关于日志收集和发送的信息  3）kibana中能够看到收集到的日志 |
| 测试结果： | Pass |

### 日志推送

日志推送测试情况如下表所示：

|  |  |
| --- | --- |
| 测试项目： | 日志收集 |
| 测试分项： | 日志推送接口日志收集 |
| 测试目的： | 测试日志系统是否能够顺利收集来自日志推送接口的日志 |
| 预置条件： | 日志系统部署完毕，运行正常。日志推送接口API Server已经安装部署 |
| 测试步骤： | 1）部署作为监控用途的kibana服务  2）部署一个测试用应用，该应用会定时向日志推送接口推送日志  3）查看日志收集情况 |
| 预期结果： | 1）API Server的console中有关于日志收集和发送的信息  3）kibana中能够看到收集到的日志 |
| 测试结果： | Pass |

日志推送接口日志格式验证测试情况如下表所示：

|  |  |
| --- | --- |
| 测试项目： | 日志收集 |
| 测试分项： | 日志推送接口日志格式验证 |
| 测试目的： | 测试日志系统是否能够验证日志推送接口的日志的格式 |
| 预置条件： | 日志系统部署完毕，运行正常。日志推送接口API Server已经安装部署 |
| 测试步骤： | 1）部署作为监控用途的kibana服务  2）部署一个测试用应用，该应用会定时向日志推送接口推送日志，但是其中缺少timestemp信息  3）查看日志收集情况 |
| 预期结果： | 1）API Server的console中有关于拒绝日志转发信息 |
| 测试结果： | Pass |

### 日志批量写入

日志批量写入测试情况如下表所示：

|  |  |
| --- | --- |
| 测试项目： | 日志收集 |
| 测试分项： | 日志推送接口批量日志收集 |
| 测试目的： | 测试日志系统是否能够顺利收集来自日志推送接口的批量日志 |
| 预置条件： | 日志系统部署完毕，运行正常。日志推送接口API Server已经安装部署 |
| 测试步骤： | 1）部署作为监控用途的kibana服务  2）部署一个测试用应用，该应用会定时向日志推送接口批量推送日志  3）查看日志收集情况 |
| 预期结果： | 1）API Server的console中有关于日志收集和发送的信息  3）kibana中能够看到收集到的日志 |
| 测试结果： | Pass |

### 日志查询

日志查询测试情况如下表所示：

|  |  |
| --- | --- |
| 测试项目： | 日志查询 |
| 测试分项： | 日志查询API |
| 测试目的： | 测试日志系统是否能够顺利根据查询条件返回结果 |
| 预置条件： | 日志系统部署完毕，运行正常。 |
| 测试步骤： | 1）通过API查询某应用的所有日志  2）通过API查询某应用某时间段内的所有日志  3）通过API查询某应用某时间段内日志条数按时间的直方图  4）查询某不存在应用的日志 |
| 预期结果： | 1）步骤一查询到某应用的所有日志  2）步骤二查询到某应用某时间段内的所有日志  3）步骤三查询到某应用某时间段内日志条数按时间的直方图  4）步骤四返回空数据集 |
| 测试结果： | Pass |

### 用户token签发

用户token签发测试情况如下表所示：

|  |  |
| --- | --- |
| 测试项目： | 鉴权系统 |
| 测试分项： | token签发 |
| 测试目的： | 测试鉴权系统是否能够按需求正确签发token |
| 预置条件： | 日志系统部署完毕，运行正常。 |
| 测试步骤： | 1）使用正确的用户名密码请求签发根token  2）使用错误的用户名密码请求签发根token  3）使用正确的根token申请签发关于某应用的token  4）使用过期的根token申请签发关于某应用的token  5）使用正确的根token申请签发关于某不存在应用的token |
| 预期结果： | 1）步骤一返回202 Accept和新token  2）步骤二返回403 Forbidden  3）步骤三返回202 Accept和新token  4）步骤四返回403 Forbidden  5）步骤五返回404 Not Found |
| 测试结果： | Pass |

### 用户隔离日志写入

用户隔离日志写入测试情况如下表所示：

|  |  |
| --- | --- |
| 测试项目： | 日志系统 |
| 测试分项： | 日志推送接口的用户隔离 |
| 测试目的： | 测试日志推送接口的用户隔离是否工作正常 |
| 预置条件： | 日志系统部署完毕，运行正常。 |
| 测试步骤： | 1）使用正确的token向日志推送接口推送数据  2）使用过期token向日志推送接口推送数据  3）不使用token向日志推送接口推送数据  4）向日志系统推送“appid”和token中access字段不一致的数据  5）推送关于某不存在应用的日志数据 |
| 预期结果： | 1）步骤一返回202 Accept  2）步骤二返回403 Forbidden  3）步骤三返回401 Unauthorized  4）步骤四返回403 Forbidden  5）步骤五返回404 Not Found |
| 测试结果： | Pass |

### 用户隔离日志读取

用户隔离日志读取测试情况如下表所示：

|  |  |
| --- | --- |
| 测试项目： | 日志系统 |
| 测试分项： | 日志推送接口的用户隔离 |
| 测试目的： | 测试日志推送接口的用户隔离是否工作正常 |
| 预置条件： | 日志系统部署完毕，运行正常。 |
| 测试步骤： | 1）使用正确的token查询某应用的日志数据  2）使用过期token查询某应用的日志数据  3）不使用token查询某应用的日志数据  4）查询“appid”和token中access的应用的日志数据  5）查询某不存在应用的日志数据 |
| 预期结果： | 1）步骤一返回200 OK和日志数据  2）步骤二返回403 Forbidden  3）步骤三返回401 Unauthorized  4）步骤四返回403 Forbidden  5）步骤五返回404 Not Found |
| 测试结果： | Pass |

## 其他测试

此节我们将对日志系统其它方面的测试进行介绍，其中主要是高可用测试和对kafka的水平扩展测试。由于对ElasticSearch的水平扩展测试已在上文“性能测试”一节中加以阐述，所以在此不再重复。

### 高可用测试

当日志系统因网络或机器原因导致部分服务无法达到计划的规模时，整体的服务不应中断。待服务恢复后，恢复的节点应当被重新加入到集群之中。

|  |  |
| --- | --- |
| 测试项目： | 日志系统 |
| 测试分项： | 日志系统高可用测试 |
| 测试目的： | 测试日志系统部分失效后的自我恢复能力 |
| 预置条件： | 日志系统部署完毕，运行正常。 |
| 测试步骤： | 1）通过日志系统管理API降低ElasticSearch的data节点数量，使得集群中部分节点被停止  2）通过日志系统管理API降低kafka的broker数量，使得集群中部分节点被停止  3）过一段时间恢复以上步骤中关掉的节点 |
| 预期结果： | 1）步骤一结束后通过日志系统管理API检查日志系统状态可以看到状态由green变为yellow。ElasticSearch中的索引重新分配。一段时间后状态变回green。  2）步骤二结束后通过日志系统管理API检查日志系统状态可以看到kafka broker的数目降低。  3）步骤三结束后通过日志系统管理API检查日志系统状态可以看到kafka broker和ElasticSearch data节点的数量恢复。ElasticSearch中的索引重新分配。 |
| 测试结果： | Pass |

### kafka水平扩展测试

kafka水平扩展测试情况如下表所示：

|  |  |
| --- | --- |
| 测试项目： | kafka集群 |
| 测试分项： | kafka水平扩展测试 |
| 测试目的： | 测试kafka水平扩展能力 |
| 预置条件： | 日志系统部署完毕，运行正常。 |
| 测试步骤： | 1）通过日志系统管理API提高kafka broker的数量 |
| 预期结果： | 1）通过日志系统管理API检查日志系统状态可以看到kafka broker数量提高  2）通过kafka cli发现新节点加入集群 |
| 测试结果： | Pass |

## 本章小结

本章对日志系统的主要功能进行了测试的设计，阐述了测试所用的环境和测试用例的步骤设计。除了测试了日志系统各个接口和功能，还测试了系统的性能，可扩展性，高可用性等诸多方面。

测试结果表明，本课题实现的日志管理系统，提供了高吞吐，高可用的数据收集，存储，分析，查询等能力。同时提供了友好的用户界面。符合各项既定要求。

# 总结

## 论文总结

信息技术和信息系统已经深入到社会的各个行业和领域，人们的生活也越来越离不开来自于网络和信息系统的服务。人们在系统中进行的每一次操作都会留下痕迹，而日志数据就是记录这些操作痕迹的最主要方式。网络应用的日志数据中往往包含着丰富的用户访问信怠和用户特征，通过对日志数据的充分挖掘，我们可以获取到系统运营状况、用户行为习惯等等诸多有价值的信息。而随着数据量的爆炸式增长，如何有效的存储、处理和分析大规模日志数据已然成了我们需要迫切解决的问越。云计算和云服务的出现为解决日志数据的存储和处理提供了一种行之有效的方式。通过在云计算平台上建立计算和存储集群，我们可以获得对海量数据的存储和处理能力。

本文主要是从基于Elastic Stack的微服务日志管理系统的设计与实现入手，对每一步进行展开说明。近几年随着ElasticSearch的飞速发展，包含了ElasticSearch，logstash，各种Beats收集器在内的Elastic Stack系统在实际的生产中得到了广泛的应用。本系统主要使用ElasticSearch作为日志数据存储组件，利用Beats收集数据，利用logstash进行数据传递和处理，利用kafka作为数据缓存组件并且实现了Reatful的日志管理API以及其数据鉴权。最终完成了一个从数据采集到分析展示的完善系统，同时与MSCP微服务管理系统完成了集成工作，实现了一个用户友好的UI。在最后对系统进行了功能测试，证明了本系统的可用性。

本系统的出现，为MSCP云计算平台用户提供了统一的日志分析能力，降低了日志分析和处理的成本。同时方便了用户将其自己的日志系统接入。用户在本系统的基础上可以开发数据监控，行为分析，分类统计等多种功能。高吞吐，分布式的设计为数据产生价值提供了基础。

## 下一步的研究工作

本文设计并实现的日志管理系统符合本文既定的各项需求，开放的数据接口为用户进行数据分析提供了方便。但是本文所述的日志管理系统还存在不少需要完善的地方。下阶段需要研究的重点主要有：

1. 提高系统在高压力下的稳定性，提供更好的维护性能和在故障场景下的集群自愈能力。
2. 提供更多的日志分析和处理能力。应用人工智能等新技术从日志数据中挖掘出更多有价值的结果。
3. 改进系统的自动弹性伸缩能力。提高系统应对压力高峰的能力。
4. 提高系统UI的功能，在UI上提供更多的可操作选项。
5. 改进系统管理接口，提供对系统本身的实时监控。

综上所述，本人还将继续深入研究日志管理系统的基础组件和日志分析的各种方法，以进一步完善日志管理系统的设计。随着微服务化的不断发展，越来越多的应用开始迁移到云平台上。用户也越来越需要云平台能够提供更多的日志管理和分析能力。目前我们的日志管理组件还比较简单，下一步将继续深入丰富系统功能，改善系统设计，以更加满足客户对日志管理的需求。