



华南理工大学

South China University of Technology

专业学位硕士学位论文

基于 ELK 与 kafka 的分布式工业日志系统的设计与实现

作者姓名 巫辉强

学位类别 工程硕士（电子与通信工程）

指导教师 向友君 副教授

王界兵 教授级高工

所在学院 电子与信息学院

论文提交日期 2019 年 4 月

Design and Implementation of Distributed Industrial Log System Based on Kafka and ELK

A Dissertation Submitted for the Degree of Master

Candidate: Wu Huiqiang

Supervisor: Associate Prof. Xiang Youjun

South China University of Technology

Guangzhou, China

分类号：TP393

学校代号：10561

学 号：201621009766

华南理工大学硕士学位论文

基于 ELK 与 kafka 的分布式工业日志系统的设计与实现

作者姓名：巫辉强

指导教师姓名、职称：向友君 副教授

申请学位级别：工程硕士

学科专业名称：电子与通信工程

研究方向：智能信息处理

论文提交日期：2019 年 4 月 10 日

论文答辩日期：2019 年 5 月 31 日

学位授予单位：华南理工大学

学位授予日期： 年 月 日

答辩委员会成员：

主席：傅予力 教授

委员：向友君 副教授、余卫宇 副教授、李波 副教授、周斯宁 高工

华南理工大学

学位论文原创性声明

本人郑重声明：所呈交的论文是本人在导师的指导下独立进行研究所取得的研究成果。除了文中特别加以标注引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写的成果作品。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。本人完全意识到本声明的法律后果由本人承担。

作者签名：巫辉强

日期：2019年5月31日

学位论文版权使用授权书

本学位论文作者完全了解学校有关保留、使用学位论文的规定，即：研究生在校攻读学位期间论文工作的知识产权单位属华南理工大学。学校有权保存并向国家有关部门或机构送交论文的复印件和电子版，允许学位论文被查阅（除在保密期内的保密论文外）；学校可以公布学位论文的全部或部分内容，可以允许采用影印、缩印或其它复制手段保存、汇编学位论文。本人电子文档的内容和纸质论文的内容相一致。

本学位论文属于：

☐ 保密（校保密委员会审定涉密学位论文时间：____年__月__日），于____年__月__日解密后适用本授权书。

☒ 不保密，同意在校园网上发布，供校内师生和与学校有共享协议的单位浏览；同意将本人学位论文编入有关数据库进行检索，传播学位论文的全部或部分内容。

（请在以上相应方框内打“√”）

作者签名：巫辉强

日期：2019年5月31日

指导教师签名：阿古

日期：2019年5月31日

作者联系电话：

电子邮箱：

联系地址(含邮编)：

摘 要

在工业生产过程中，企业通常会遇到生产设备故障的问题，设备日志在分析设备故障原因以及了解设备损耗情况方面发挥着重要的作用，对设备日志的存储和分析工作由工业日志系统完成。近年来，我国提出中国制造 2025，这意味着对工业日志系统提出了更高的要求，然而，传统的工业日志系统并不能满足其需求，这体现在不能高并发处理日志存储请求、缺乏对日志数据的实时分析以及对原始日志数据的处理方式较为简陋。因此，设计实现一个分布式工业日志系统来解决上述问题具有重要意义。

本文以 A 公司的日志系统需求为背景，针对传统工业日志系统的不足，设计实现了基于 ELK 和 kafka 的分布式工业日志系统，采用分层设计的思想，将该日志系统分为收集层、缓冲层以及检索层，主要完成的工作包括：

(1) 针对传统工业日志系统无法高并发处理日志请求的问题，本文设计实现了一个日志收集服务子系统，该子系统采用 Reactor 事件驱动模式和线程池技术实现了高并发处理日志请求，设计实现了环形内存缓冲降低了日志文件磁盘 I/O 对服务器性能的影响。

(2) 针对传统工业日志系统缺乏对日志数据的实时分析的问题，本文设计实现了一个日志计算中心，日志计算中心基于 Elasticsearch 实现了日志数据的实时检索分析，并提供一个日志检索的可视化界面。

(3) 针对传统工业日志系统对原始日志数据的处理方式较为简陋的问题，本文设计实现了一个日志缓存服务子系统，该子系统采用 logstash 对实现了日志数据处理规则的灵活配置，并基于分布式消息系统 kafka 实现了日志数据的缓存队列。

最后，本文对设计的工业日志系统进行测试分析，验证了日志系统能够高并发处理工业设备的日志存储请求，对原始日志数据的处理规则实现了可灵活配置，并实现了对日志数据的实时检索分析。

关键字： 工业日志系统；实时检索；高并发；原始日志处理

Abstract

In the industrial production process, enterprises often encounter the problem of production equipment malfunction. The equipment log plays an important role in analyzing the cause of equipment malfunction and understanding the equipment loss. The storage and analysis of the equipment log is completed by the industrial log system. In recent years, China has proposed to manufacture 2025 in China, which means higher requirements for industrial log systems. However, the traditional industrial log system cannot meet its needs, which is reflected in the inability to process log storage requests concurrently, lack of real-time analysis of log data, and the crude handling of raw log data. Therefore, it is important to design and implement a distributed industrial log system to solve the above problems.

Based on the requirements of the log system of Company A, this paper designs and implements a distributed industrial log system based on ELK and kafka for solving the the problems existing in the traditional industrial log system. The idea of hierarchical design is adopted, and the log system is divided into collection layer and buffer Layer and search layer, The main work completed includes:

(1) Aiming at the problem that the traditional industrial log system can't handle the log request concurrently, this paper designs and implements a log collection service subsystem. The subsystem uses Reactor event-driven mode and thread pool technology to implement high-consistent processing log request, and realizes the ring memory buffer, which reduces the impact of log file disk I/O on server performance.

(2) Aiming at the lack of real-time analysis of log data in traditional industrial log system, this paper designs and implements a log computing center. The log computing center realizes real-time retrieval of log data based on Elasticsearch and provides a visual interface for log retrieval.

(3) Aiming at the problem that the traditional industrial log system handles the original log data is relatively simple, this paper designs and implements a log cache service subsystem, which uses logstash to implement flexible configuration of log data processing rules and is implements a cache queue for log data based on distributed message Kafka system.

Finally, this paper tests and analyzes the designed industrial log system, and verifies that the log system can process the log storage request of the industrial equipment concurrently, and the processing rules of the raw log data can be flexibly configured and the real-time retrieval of log data is realized..

Keywords: industrial log system; real-time retrieval; high concurrent; raw log processing

目录

摘 要.....	I
Abstract.....	II
第一章 绪论.....	1
1.1 项目背景.....	1
1.2 研究现状.....	1
1.2.1 工业日志系统的发展现状.....	1
1.2.2 开源日志系统.....	3
1.2.3 存在的问题.....	6
1.3 本文的工作.....	7
1.4 论文组织架构.....	8
第二章 系统需求与总体架构设计.....	9
2.1 工业日志系统.....	9
2.2 系统设计目标.....	10
2.3 系统架构设计.....	10
2.4 核心子系统概要设计.....	12
2.4.1 日志收集服务子系统.....	12
2.4.2 日志缓存服务子系统.....	13
2.4.3 日志计算中心.....	14
2.5 本章小结.....	15
第三章 日志收集服务子系统的设计与实现.....	16
3.1 日志收集服务子系统的架构设计.....	16
3.2 基于 Reactor 事件驱动的网络模块的实现.....	17
3.2.1 事件驱动模型.....	17
3.2.2 Reactor 模式的具体实现.....	19
3.2.3 worker 线程池的具体实现.....	24
3.3 基于环形内存缓冲 ringBuffer 的业务模块的实现.....	26
3.4 基于轻量级日志采集组件 filebeat 的上报模块的实现.....	30
3.5 本章小结.....	33
第四章 日志缓存服务子系统的设计与实现.....	34

4.1 日志缓存服务子系统的架构设计.....	34
4.2 数据处理模块的实现.....	35
4.2.1 开源数据处理组件对比.....	35
4.2.2 数据处理模块的具体实现.....	35
4.3 缓存模块的实现.....	38
4.3.1 技术选型.....	38
4.3.2 kafka 集群协调一致的实现.....	39
4.3.2 日志数据高可用的实现.....	41
4.3.3 kafka 集群的部署.....	42
4.4 本章小结.....	44
第五章 日志计算中心的设计与实现.....	45
5.1 日志计算中心的架构设计.....	45
5.2 日志检索模块的实现.....	46
5.2.1 Elasticsearch 概述.....	46
5.2.2 Elasticsearch 集群的部署.....	47
5.2.3 Elasticsearch 集群数据高可用的具体实现.....	49
5.2.4 Elasticsearch 性能调优.....	50
5.3 展示模块的实现.....	53
5.4 本章小结.....	54
第六章 系统测试与结果分析.....	55
6.1 测试内容.....	55
6.2 测试环境.....	55
6.2.1 测试环境部署.....	55
6.2.2 测试工具.....	56
6.3 基本功能测试.....	56
6.4 日志收集层性能测试.....	59
6.4.1 网络模块性能测试.....	59
6.4.2 业务模块性能测试.....	61
6.5 日志缓冲层性能测试.....	62
6.5.1 读写性能测试.....	62

6.5.2 高可用性测试.....	66
6.6 日志检索层性能测试.....	68
6.6.1 查询性能测试.....	68
6.6.2 高可用性测试.....	73
6.7 测试分析总结.....	74
总结与展望.....	76
总结.....	76
展望.....	77
参考文献.....	78
攻读硕士学位期间取得的研究成果.....	81
致谢.....	82

第一章 绪论

1.1 项目背景

改革开放 40 年以来，我国工业发展迅速，由中国信息通信研究院于 2018 年 11 月发布的《2018 年中国工业发展研究报告》显示，1978-2017 年 GDP 年均增速为 9.5%，全部工业增加值年均增速为 11%。在工业方面，我国拥有所有工业门类制造能力，200 多种工业品产量稳居世界首位，是世界第一制造大国。

在工业生产活动中，工业设备的正常运转以及定期维护对于保证生产任务如期完成有重要意义，设备的日志数据在这里起到了十分重要的作用，工业日志系统负责存储和分析生产过程中设备输出的日志数据，同时，工业大数据是未来工业在全球市场竞争中发挥优势的关键^[1]，而工业日志系统存储的设备日志是工业大数据的重要来源。因此，工业日志系统成为诸多学者和技术人员研究的热点问题之一。

本文依托于本人所在的实验室与广东 A 智能装备有限公司的合作项目，设计实现了基于 ELK 和 kafka 的分布式工业日志系统。该系统能够低延迟地处理设备的高并发日志存储请求，实现了日志数据处理规则的灵活配置，具有日志数据的实时检索分析能力，并保证数据的高可用。该系统符合 A 智能装备有限公司的日志系统需求，其实时的设备日志分析能够提升 A 公司对设备故障情况的处理效率。

1.2 研究现状

本节首先介绍工业日志系统的发展现状，然后选取了典型的传统日志系统进行分析，最后提出传统工业日志系统存在的问题。

1.2.1 工业日志系统的发展现状

我国工业日志系统起源于 20 世纪 90 年代，工业日志系统的发展可以分为两个时期，分别是单节点日志系统时期和分布式日志系统时期^[2]。在单节点日志系统时期，开始阶段由于设备输出日志速率较低，人们直接将日志数据保存在磁盘文件中。随后由于产生了对日志数据进行分类的需求，于是日志数据的载体由磁盘文件变为数据库，且随着工业设备自动化程度不断提高，对工业日志系统的要求也不断提高，人们通过提高节点的硬件配置，以及不断优化提升数据库的性能来满足与日俱增的需求。随着分布式技术的兴起，且节点的硬件配置以及数据库性能的提升已经很难取得突破，工业日志系统进入了分布式日志系统时期，该阶段的工业日志系统通过横向扩展节点即可提高日志系统的

性能，且节点硬件配置要求不高，价格低廉。

2012 年，美国通用电气公司提出工业互联网的概念，旨在利用蓬勃发展的互联网技术对传统的工业进行融合、改造和升级，工业互联网的提出引发了一轮新的工业革命^[3]。我国为了在工业互联网领域占领制高点，也相继提出了“中国制造 2025”、“两化融合”等战略规划，并与德国开展工业 4.0 合作^[4]。

在工业 4.0 的背景下，人们通过互联网信息技术来调控工业生产活动，这意味着企业能够节省更多的人力，进行更大规模的工业生产活动^[5]。在工业生产过程中，通常会发生设备故障事件。在早期工业设备数量较少的情况下，技术人员进入服务器节点通过 vim 等工具手动查找故障设备日志，然而在工业 4.0 下，生产设备更多，传统的人力查找设备日志并不可取。另外，传统的工业日志系统通常将每天的设备日志数据存放在 HDFS、HBase 中，然后对每天的日志数据运行 MapReduce 任务进行日志数据处理，并产生一个统计报表。然而这种日志处理方案只能进行离线处理，缺乏日志数据的实时处理，这对于实时检修故障设备并没有太大帮助。

与此同时，工业 4.0 的提出也推动了工业设备自动化程度的不断提高，设备日志输出的速度随之提高^[6]。早期的工业设备自动化程度不高，设备日志产生的速度也相对较低，传统的工业日志系统足以应付其日志存储请求，而在工业 4.0 下，需要考虑工业设备高并发的日志存储请求，比如在注塑行业，多个生产阶段可能每分钟产生的日志达到上千条，传统的工业日志系统并不能承载高并发的日志存储请求。

近几年来，工业大数据、云计算等的概念悄然兴起，中国制造 2025 中也指出，希望能够通过工业大数据和云计算使得传统制造业向工业 4.0 转变^[7]。工业大数据是指在工业领域中围绕整个产品全生命周期各个环节所产生的各类数据及相关技术和应用的总称，其主要来源通常分为三类：生产经营相关业务数据、设备物联数据以及外部数据。工业设备日志属于其中的设备物联数据，大多数采集得到的原始设备数据为“脏”数据，在进行数据分析之前需要经过规范处理，传统的工业日志系统中数据处理模块通常需要设备端为日志数据附加一些额外的字段，来协助其进行数据处理，这给网络传输带宽带来了一定的负担，同时当引进新设备时，数据处理逻辑的修改可能使得所有的工业设备为了适配新处理逻辑而附加新的字段，这对于技术人员无疑是一种重复的无意义的劳动。

综合以上，工业 4.0 对工业日志系统提出了更高的需求。然而，传统的工业日志系统远不能满足其提出的需求，主要涉及日志数据的实时处理、高并发日志存储请求以及设备日志数据处理等方面的问题。因此，本文研究设计实现一个分布式工业日志系统，

解决上述传统工业日志系统遇到的问题，有助于提高企业生产效率，减少不必要的人力物力，使得工业生产向信息化发展，从而推动制造业向工业 4.0 的转变。

1.2.2 开源日志系统

日志系统分为商用日志系统和开源日志系统。商用日志系统通常是专业的技术团队为一个企业或厂商定制研发，并为该系统提供后续的维护服务，但商用日志系统造价不菲，同时商用日志系统的代码并不开源，无法深入其源码进行研究。因此，本文主要对现有的开源日志系统进行分析，并说明这些开源日志系统实现过程中存在的问题。

(1) Scribe

Scribe 是 Facebook 公司开源的一个用于对流数据聚合的日志收集系统^[8, 9]，能够从多种数据源中收集日志，最终日志数据流向目标存储系统中。Scribe 实现了一个高可用性、节点可灵活扩展的日志收集系统。

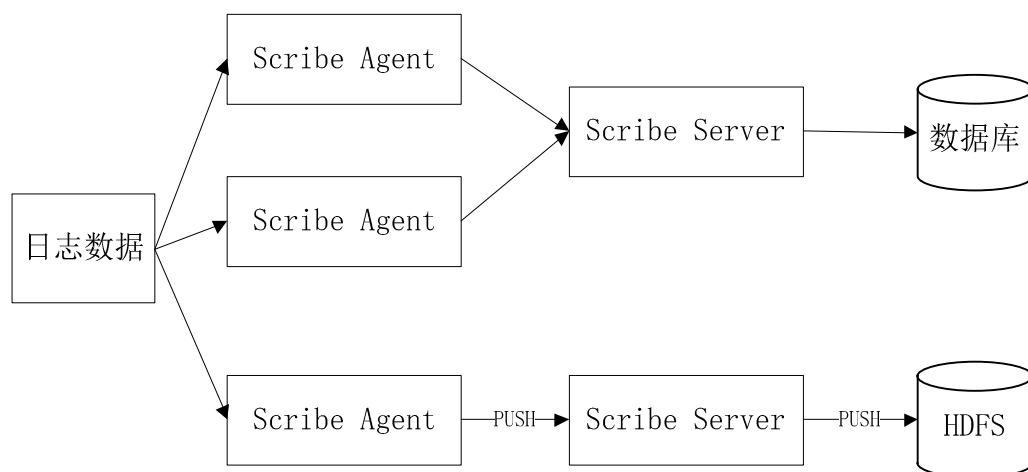


图 1-1 Scribe 架构图

Scribe 架构图如图 1-1 所示，分为 Scribe Agent、Scribe Server 和存储系统三个部分。

Scribe Agent 作为客户端向 Scribe Server 发送日志数据，日志数据除了日志自身携带的内容，还需要附加一个 category 字段的值，表示该日志的类型。

Scribe Server 作为服务端接收日志数据，在启动 Scribe 服务前通过配置 Scribe.conf 文件来实现对不同 category 值日志数据的处理逻辑。

存储系统是日志数据流向的目的地址，对应于 Scribe 配置文件中的 store。store 的配置实现了 Scribe Server 对不同类型日志的处理逻辑和其目的存储系统，store 的配置有两个重要的选项：category 和 type，category 指明该 store 处理的日志类型，type 指定

该 store 的存储类型。Scribe 目前支持 7 种 store 类型, 分别为 file、buffer、network、bucket、null、thirtfile 以及 multi。

Scribe 在 Scribe Server 上实现了数据的高容错性, 当存储系统出现故障时, Scribe Server 将日志数据暂时写到节点的本地磁盘中, 待到日志存储系统恢复可用之后重新发送日志数据。

但 Scribe 并没有实现 Scribe Agent 和 Scribe Server 的数据容错性, 需要用户自己实现, 且 Scribe 只提供了日志数据的存储功能, 并不具备对日志数据的实时分析能力。

(2) Chukwa

Chukwa 是 Hadoop 的一个子项目^[10, 11]。Hadoop MapReduce 框架的最初目的是日志处理, 然而使用 MapReduce 来直接处理来自许多机器发送的海量日志是不合适的, 因为 MapReduce 框架在少量的大文件处理上效果最佳, 但由于日志在机器上随着时间会不断地产生, 大量对日志处理结果的合并为 Hadoop 在数据管理上带来了一定的开销^[12]。

Chukwa 的提出便是为了弥合日志处理与 Hadoop 生态系统之间的差距, chukwa 通过对不同数据源的日志进行汇聚再转交给 MapReduce 处理, 使得 MapReduce 能够发挥出最佳性能。

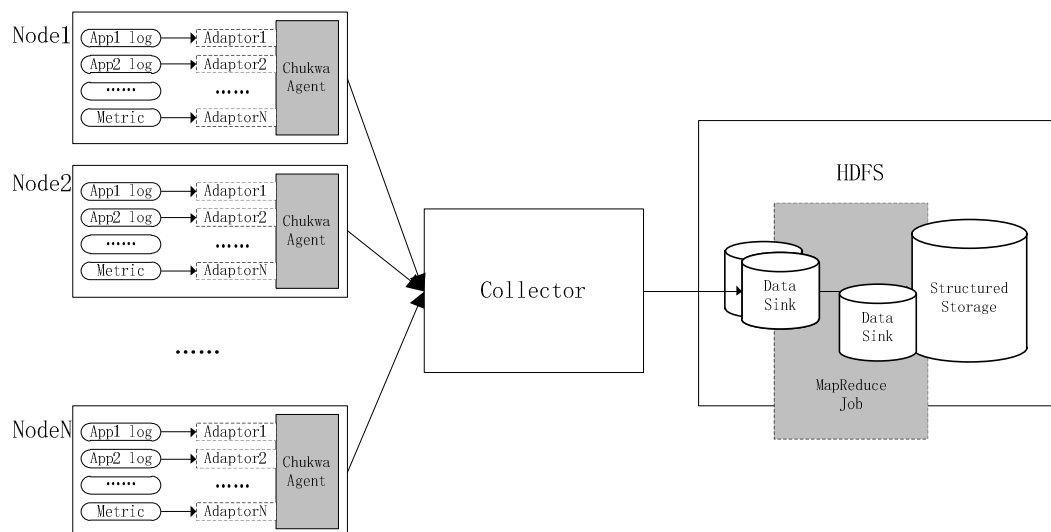


图 1-2 Chukwa 架构图

Chukwa 架构图如图 1-2 所示, 由 Adaptor、Agent、以及 Collector 构成, Agent 分布在数据的产生端, 每个 Agent 内含有一个或者多个 Adaptor。Collector 负责收集 Agent 节点发送的日志数据, 并将其发送到 HDFS 存储系统中。

Adaptor 负责实际的数据收集工作, 不同的数据源采用与之适配的 Adaptor 来收集, Chukwa 目前提供了一些标准 Adaptor: FileAdaptor、ExecAdaptor、UDPAdaptor 和

SocketAdaptor, 用户也可以实现一个 adaptor 来满足需求。这为用户增加了数据收集过程的灵活性, 但同时也增加了开发与维护的成本。

Agent 负责 Adaptor 的管理, Adaptor 运行在 Agent 进程内, 是一个可动态加载的模块。Agent 可以通过执行命令来实现对 Adaptor 模块的启动和结束。Agent 定期记录已发送给 Collector 的数据偏移量, 当发生故障时, 可以根据偏移量继续发送数据, 数据的容错性高。

Collector 处于 Chukwa Agent 与 HDFS 存储系统之间, Collector 接收多达数百个 Chukwa Agent 节点发送的数据, 并将所有的数据写入到一个 sink file。Collector 定时关闭 sink file, 重命名后将 sink file 标记为可处理的, 并重新开始写入新的 sink file, 该过程大大减少了 MapReduce 需要处理的 HDFS 文件。

然而, Chukwa 并不能达到实时处理日志数据的要求, Collector 需要汇聚一定量的日志后才形成一个 sink 文件, 这个过程会有一定的延时。同时, chukwa 的存储系统高度依赖 Hadoop。

(3) Flume

Apache Flume 为日志数据的高效收集、聚合和移动提供了一种分布式的、可靠的服务^[13, 14], 它旨在将日志数据从应用程序移动到 Apache Hadoop 的 HDFS。Flume 在 2011 年 10 月经历了一次核心组件的重构, 我们将 Flume 在 1.0 之前的版本统称 Flume OG, 将重构之后的版本统称为 Flume NG, 这里主要分析 Flume NG。

如图 1-3 为 Flume NG 的架构图。一个 Flume Agent 节点包括 Source、Channel 和 Sink 三个组件, 数据从一个 Flume Agent 节点的 sink 组件发送到另一个 Flume Agent 节点的 Source 组件, 多个 Flume Agent 节点实现了不同的数据源汇聚到最终的目的节点。Flume NG 传输数据的基本单元是 Event, 大小约为几个字节到几千字节。一个 Event 由键/值对形式的头部和字节数组形式的实体组成。

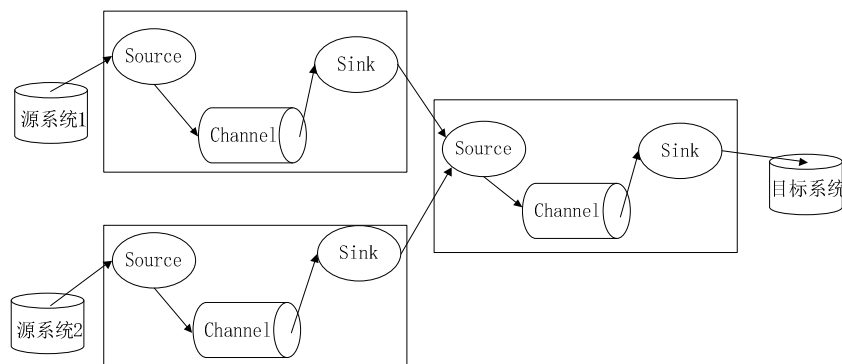


图 1-3 Flume NG 架构图

Flume NG 从原来的分布式数据收集系统转变成为一个配置简单的传输工具，使得 Flume 插件化，并且通过事务保证了数据的完整和一致性，通过部署 Agent 节点实现数据的分流和复制，容易扩展，Flume NG 支持多种存储系统。

然而 Flume NG 的客户端方面需要用户自己实现代码，且实时数据的收集是通过命令行方式，可靠性较差，并没有实现对文件新增记录的监控，对数据的过滤和解析能力较差。

1.2.3 存在的问题

在 1.2.2 节中，通过对多个开源的日志系统的分析，传统的开源日志系统存在以下的问题：

(1) 传统日志系统缺乏实时的数据分析。传统的日志系统更多关注于日志的落地存储，如存储到 HDFS、HBase 等，大多都不具备对日志数据进行即时分析的能力。传统的日志处理方案通常是将日志收集到一个存储系统上，然后对一段时间内的日志数据进行离线的数据分析工作。但是在工业生产活动中，面对突发的设备故障，传统的日志系统无法回答诸如“这个设备故障前几分钟发生了什么情况”这种即时性的问题。设备故障是一个企业或工厂经常遇到的事故，技术人员需要设备日志来确定设备问题，所以对日志数据的实时分析应该是工业日志系统的重要内容之一，实现实时性的日志查询分析，对于一个企业或工厂在发生故障时迅速修复故障设备恢复生产、减少企业经济损失有重要的意义。

(2) 传统日志系统的数据处理方式较为简陋。在 1.2.2 节对传统的开源日志系统的分析中，Scribe 中实现对日志数据的处理是通过 Scribe Agent 在日志内容上额外附加一个 category 字段值，通过判断 category 值来进行分类或者丢弃，该方式增加了网络传输的额外负担，同时添加的 category 字段只用于日志的分类，并没有对实际的日志内容进行处理；chukwa 中对不同的数据源有对应的 Adaptor，用户也可以自定义实现自己的 Adaptor，但 Adaptor 对于来自同一个数据源的不同日志数据并不具备处理能力；Flume NG 官方则提供了拦截器 (Interceptor) 来进行数据处理，如 Timestamp Interceptor 可以对一个 Event 头部的 timestamp 字段进行检查，Host Interceptor 可以对 Event 头部的 host 字段进行检查等，但这类拦截器需要日志产生端对日志内容添加 timestamp、host 等字段，日志产生端需要去适应日志系统的规则，对日志产生端并不友好。在工业生产中，不同设备类型的设备可能有不同的日志输出格式，工业日志系统应该在日志收集端实现

对不同类型日志的数据处理，而无需工业设备端为了适配收集端的数据处理规则而添加额外的字段，同时数据处理规则实现灵活配置。

(3)在工业 4.0 背景下，工业日志系统还需要具备高并发、低延迟以及数据高可用的特性，但传统的日志系统并不能具备所有特性。Scribe 在高可用方面只保证了数据在 Scribe Server 与存储系统之间的高可用，数据在 Scribe Agent 与 Scribe Server 的高可用性需要用户自己来实现。chukwa 通过汇聚多个数据源的数据形成一个大文件 sink file，这个过程决定了 chukwa 不能达到低延迟的目标。Flume NG 通过命令行形式实现实时数据收集，其数据容易丢失，不能保证数据的可靠性。

1.3 本文的工作

本文针对传统日志系统的不足，设计并实现了基于 ELK 和 Kafka 技术的分布式工业日志收集系统，该系统支持实时的日志数据分析，具备高并发、低延迟和高可用的特性，并提供了日志数据处理规则可配置的功能。主要完成了如下工作：

(1)研究并分析传统的开源日志系统的架构和其优缺点，明确本文设计的日志系统需要解决的问题。

(2)针对传统日志系统缺乏实时的数据分析的问题，本文在日志检索层设计并实现了基于 elasticsearch 的日志数据实时检索模块，为 A 公司提供实时的日志数据检索服务。在工业生产活动中，工业设备产生日志数据速率快，数据量大，技术人员进入系统通过 vim 等工具搜寻设备日志效率十分低下，实时检索模块能够帮助技术人员快速检索到所需的设备日志，并提供了日志检索的可视化界面。

(3)针对传统日志系统的数据处理方式较为简陋的问题，本文在日志缓冲层采用 logstash 实现对日志数据的处理，logstash 支持从多种数据源收集数据，并能够简便的实现日志数据的处理，它无需日志产生端为了适配收集端的日志处理规则而添加额外的字段，其丰富的插件生态系统为用户编写日志数据处理规则提供了便利。

(4)针对工业日志系统所需求的高并发、低延迟以及高可用特性，本文在日志收集层设计并实现了能够低延迟处理高并发日志存储请求、数据高可用的工业日志收集系统。在日志收集层基于 Reactor 事件驱动模型设计实现了一个能够高并发处理日志存储请求的服务器，并通过设计环形内存缓冲减少了日志数据的磁盘写入次数，降低了磁盘 I/O 对服务器性能的影响。本文设计的分布式工业日志系统在日志的收集层、缓冲层以及检索层均考虑了数据的高可用。

(5) 针对本文设计实现的日志系统进行具体的功能测试以及性能测试，并对测试结果进行分析，验证该日志系统已经完成设计目标，能够提供实时的数据检索服务，提供日志数据过滤规则的可配置功能，并具有高并发、低延迟和高可用的特性。

1.4 论文组织架构

第一章为绪论，简述了本文的项目背景，阐述了工业日志系统的发展现状，并通过分析当前三个典型的开源日志系统，总结了开源日志系统存在的问题，明确了本文的工作方向。

第二章为系统需求与总体架构设计，根据先进制造业所需求工业日志系统实现的要点，明确了本文工业日志系统的需求，然后提出了工业日志系统的总体架构，分析了总体架构中各层在系统中的作用，并概述了各层核心子系统的设计目标和需要实现的模块。

第三章为日志收集服务子系统的设计与实现，针对日志收集层高并发性能以及高并发场景下减少磁盘 I/O 对服务器性能影响的设计目标，深入阐述了基于 Reactor 事件驱动的网络模块、基于环形内存缓冲的业务模块以及轻量级日志上报模块的设计与实现过程。

第四章为日志缓存服务子系统的设计与实现，针对日志数据处理规则可灵活配置以及日志数据的缓冲队列的设计目标，着重阐述了数据处理模块与缓存模块的设计与实现过程。

第五章为日志计算中心的设计与实现，针对日志数据的实时检索以及可视化的日志数据检索界面的设计目标，详细阐述了日志检索模块与展示模块的设计与实现过程。

第六章为系统测试与结果分析，通过测试来验证本文设计实现的工业日志系统达到本文提出的设计目标，能够在日志收集层实现高并发处理请求以及减少磁盘 I/O 对服务器性能的影响，在日志缓冲层实现数据处理规则的可灵活配置以及日志数据的队列缓冲，在日志检索层实现日志数据的实时检索以及可视化界面展示。

最后对本文进行总结，并指出本文的后续工作方向。

第二章 系统需求与总体架构设计

本章首先阐述工业日志系统在现代制造业下需要实现的要点，进而明确本文的分布式工业日志系统的具体设计需求，然后由此提出日志系统的总体架构，并对架构中各层的核心子系统进行概要设计。

2.1 工业日志系统

工业日志系统是一个主要负责收集工业设备日志数据的日志系统。在 20 世纪 90 年代初期，传统工业的自动化程度不高，工业设备在单位时间内产生的日志量较少，单台机器的磁盘文件即可存储所有的日志数据。之后随着我国工业技术的飞速发展，设备自动化程度不断提高，日志数据的载体也经历了单机文件、单机数据库以及分布式系统的演变。近年来，我国提出中国制造 2025，并推出政策鼓励传统工业向先进制造业转变^[15]，相比于传统工业，先进制造业的自动化程度更高，原有的传统工业日志系统已经不能满足先进制造业的需求。新的工业日志系统需要实现以下要点：

(1) 日志存储请求的高并发处理。

随着工业设备的自动化程度不断提高，单位时间内设备输出的日志量增多，且工业设备数量也越来越多，日志系统需要能够高并发处理设备日志存储请求的能力。同时，设备日志不断地由而频繁的磁盘写入会导致日志系统的磁盘 I/O 过多，进而影响节点的服务器性能。实现高并发处理日志存储请求以及减少磁盘 I/O 对服务器性能的影响，是工业日志系统的难点之一。

(2) 日志数据处理规则的可灵活配置。

日志数据通常需要经过数据处理阶段，然后再写入到存储设备中。在传统日志系统中，绪论提及的开源系统中日志产生端需要根据数据处理规则来额外添加字段值。且通常在企业引进一种新工业设备时，数据处理模块需要为此重新编写日志数据处理规则，并对该模块重新进行编译等过程。在高并发的场景下，设备端为适配数据处理规则添加的额外字段会导致一定的数据传输负担，且重新编写数据处理模块的处理逻辑并不利于系统的维护。实现日志数据处理规则的灵活配置以及无需日志产生端添加额外字段，是工业日志系统的难点之一。

(3) 日志数据的高可用性。

日志数据的高可用性指的是当系统某个节点发生故障时，系统仍能正常地对外提供

服务^[16]。在先进制造业中，企业可以通过设备日志了解设备状态，提前预测可能出现的损坏零件，也可以利用大数据技术对海量设备日志进行数据分析，设备日志数据对于工业生产活动的指导有重要意义。实现日志数据的高可用性，是工业日志系统的要点之一。

(4) 日志数据的实时检索。

在传统的工业日志系统中，日志数据通常被收集到日志系统中，然后存入到 HDFS、数据库等数据载体中，并在一段时间后集中对这些数据进行离线式的日志分析。这种数据离线式集中处理的日志处理方案对于实际工业生产活动的作用收效甚微，在工业生产活动中通常遇到的设备故障问题需要日志系统能对日志数据进行实时的检索分析，而传统的日志处理方案并不能处理即时性的问题。在现代制造业中，快速修复故障设备有助于企业迅速恢复生产活动，这需要日志系统具有对日志数据的实时检索能力。因此，实现日志数据的实时检索，是工业日志系统的要点之一。

2.2 系统设计目标

根据 2.1 中对于新工业日志系统需要实现的要点，本文的分布式工业日志系统需要实现的组件包括：

(1) 实现一个日志收集组件，该组件能够实现对高并发日志请求的低延迟处理，并且能够减少日志文件的磁盘 I/O 对服务器响应性能的影响。具体指标为在数万台工业设备的并发数下保证 100MB/S 以上的吞吐率，并能在数秒内完成百万级日志数的磁盘写入。

(2) 实现一个日志缓冲组件，该组件是日志收集服务与日志检索分析服务的中间组件，该组件接收来自日志收集组件传输的日志数据，对日志数据进行处理后写入到缓冲队列，其中数据处理规则可以灵活配置，无需日志产生端添加额外的字段。缓冲队列用于缓冲日志数据，缓解上游日志写入速度与日志消费速度不一致的矛盾，并实现日志收集服务与日志检索分析服务的解耦合。

(3) 实现一个日志检索组件，该组件提供对日志数据的实时检索分析服务，并提供了一个用于日志检索的可视化界面。具体指标为在工业生产场景下，能够在千量级的实时检索并发数下保证不超过 1s 的延迟，以提供良好的客户体验。

2.3 系统架构设计

根据 2.2 节提出的系统需求，本文设计的分布式工业日志系统的总体架构如图 2-1 所示。本文对分布式工业日志系统进行分层设计，将日志系统分为日志收集层、日志缓冲层和日志处理层。日志系统中的每一层都有一个核心子系统，负责实现该层的功能逻辑。

辑。日志收集服务子系统是日志收集层的核心子系统，负责监听来自设备端的日志存储请求，接收设备日志，并将日志数据上报到日志缓冲层；日志缓存服务子系统是日志缓冲层的核心子系统，负责接收来自日志收集层的设备日志数据，并对原始的设备日志进行处理，其日志处理规则可由用户自定义配置，最后将处理后的日志写入到日志缓冲队列，供日志检索层消费；日志计算中心是日志检索层的核心子系统，负责对日志进行实时的检索分析服务，并提供了用于检索的可视化界面。本文的日志系统分成三层进行设计和实现，有利于实现各层之间模块的解耦，各层的模块只需要关注实现自身的功能，也有利于日后的维护和改良。

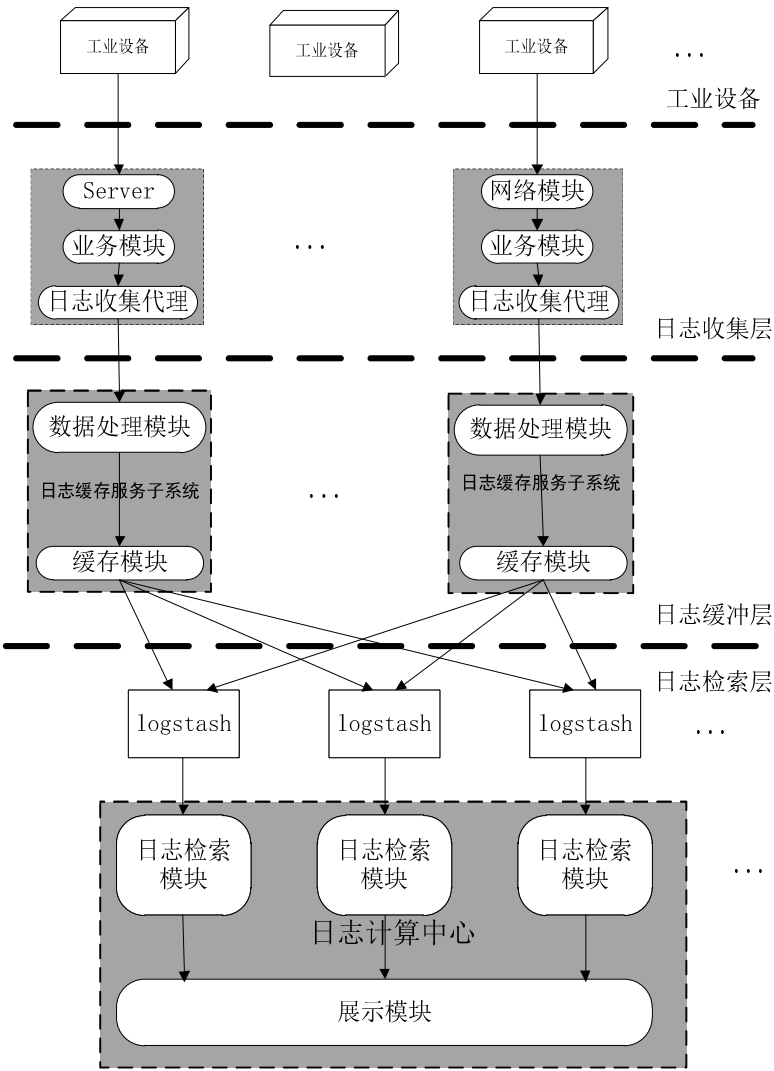


图 2-1 分布式工业日志系统架构图

本文设计的日志系统能够满足系统的需求，日志收集层关注实现低延迟处理高并发的日志存储请求，并将日志数据上报到缓冲层；日志缓冲层关注实现对日志数据的处理和缓冲队列，实现日志收集服务与日志检索服务的解耦；日志检索层主要关注实现日志

数据的实时检索，并提供日志数据的可视化检索界面。在数据高可用方面，收集层通过将数据写入磁盘保证数据不丢失，日志缓冲层和日志检索层通过副本机制保证数据的高可用性。

2.4 核心子系统概要设计

本文设计的分布式工业日志系统分为三层，每一层都有一个核心子系统负责实现该层主要功能。日志收集服务子系统收集设备日志数据，将日志数据上报到日志缓存服务子系统，日志缓存服务子系统接受收集层的设备日志，对原始的设备日志进行处理后，将日志数据加入到缓存队列，等待日志计算中心的消费，日志计算中心提供日志数据的实时检索服务，并提供数据的可视化检索界面。本文主要阐述各层的核心子系统的设计与实现，本节阐述对每一层的核心子系统进行概要设计。

2.4.1 日志收集服务子系统

日志收集服务子系统是日志收集层的核心子系统，负责实现低延迟响应工业设备端的高并发日志存储请求，将日志数据上报到日志缓冲层。日志收集层可以通过线性添加多个日志收集服务子系统节点来提升收集层的负载能力，具有高可扩展性。日志收集服务子系统的功能模块图如图 2-2 所示。

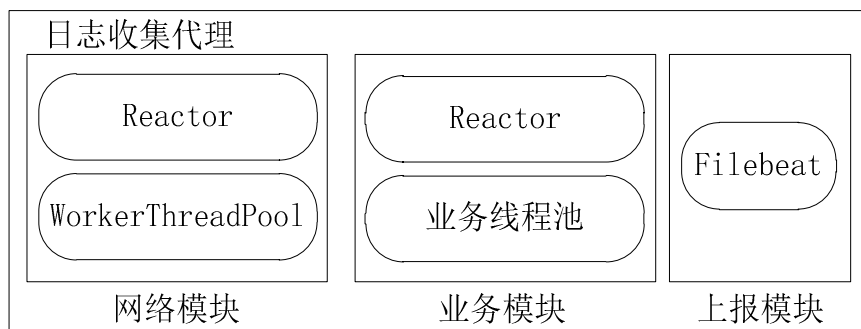


图 2-2 日志收集服务子系统功能模块图

该核心子系统负责日志数据的接收和磁盘写入，并将日志数据传输到日志缓冲层。写入到磁盘文件一方面保证数据不丢失，另一方面作为上报模块的数据源传输到日志缓冲层。然而磁盘写入操作需要陷入系统调用，而高并发下的日志存储请求会导致频繁的写入磁盘，大量的系统调用会使得日志收集服务子系统对设备日志请求的响应速度变得极为缓慢。

因此，为了保证对设备日志存储请求的低延迟处理，该核心子系统划分为网络模块、业务模块和上报模块，分离了日志数据接收和磁盘写入过程，避免日志数据的磁盘写入

处理影响节点与设备通信的性能，实现日志收集服务子系统对设备日志存储请求的高并发、低延迟处理。

(1) 网络模块

网络模块负责监听设备的日志存储请求，接受来自设备端的日志数据，网络模块采用 **Reactor** 事件驱动模型，实现对高并发请求的快速响应，将数据传输阶段交给 **worker** 线程负责，避免对外服务阻塞在数据传输逻辑上。在 **worker** 线程池实现上，由于生产者消费者模式实现的线程池涉及到频繁的加锁解锁，并不适用于高并发场景，因此本文采用在 **worker** 线程运行事件循环，通过将已连接套接字注册在 **worker** 线程的 **epoll** 上，避免了消耗系统资源的加锁解锁操作。

(2) 业务模块

业务模块负责日志数据的磁盘写入，工业设备的日志收集场景是高频率的日志存储请求，但每次传输的日志量不大，若每次日志接收对应一次磁盘写入，那大量的磁盘 I/O 将会导致大量的系统调用，系统调用是用户态到内核态的切换，频繁的用户态内核态切换会影响整个节点的性能。业务模块设计实现了一个环形内存缓冲池 **ringBuffer**，将多次的日志内容先存放在 **ringBuffer** 中，在保证实时性的情况下将多次的日志数据一次写入到磁盘，这极大的减少了磁盘 I/O 的次数。

(3) 上报模块

上报模块负责将日志数据传输到日志缓冲层，该模块采用轻量级日志传输组件 **Filebeat**，**Filebeat** 能够监听文件中新增记录，占用的系统资源极少，能够保证日志数据的可靠传输。

2.4.2 日志缓存服务子系统

日志缓存服务子系统是日志缓冲层的核心子系统，负责日志内容的缓存，供日志检索层消费。该服务子系统需实现对日志的处理规则可灵活配置，缓存日志数据，并保证数据的高可用。其功能模块图如下图 2-3 所示。

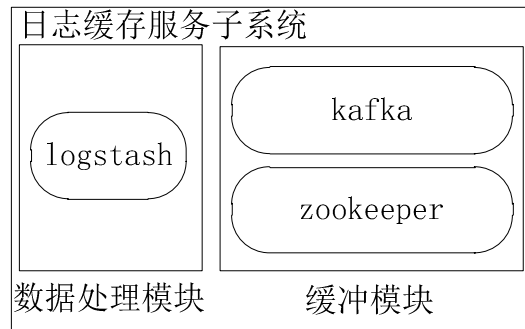


图 2-3 日志缓存服务子系统功能模块图

日志缓存服务子系统分为数据处理模块和缓存模块。

(1) 数据处理模块

数据处理模块负责接收日志收集层的日志数据，并对日志数据进行处理，其中日志数据处理规则实现可灵活配置。

(2) 缓存模块

缓存模块负责实现对日志数据的缓存，提升系统对突发日志请求高峰的应对能力，并保证数据的高可用。

2.4.3 日志计算中心

日志计算中心是日志处理层的核心子系统，负责提供日志内容的实时检索服务，保证数据的高可用性，并提供可视化的日志检索界面。其功能模块图如图 2-4 所示。

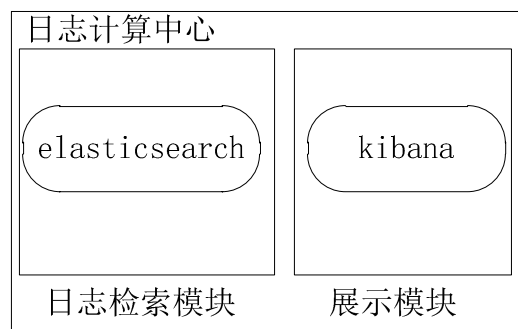


图 2-4 日志计算中心功能模块图

日志计算中心分为日志检索模块和展示模块。

(1) 日志检索模块

日志检索模块负责接收日志检索层 logstash 写入的日志数据，并对日志数据进行分析，实现日志数据的实时检索。

(2) 展示模块

展示模块负责对接日志检索模块，并将日志数据以可视化的形式展示给用户，提供可视化的检索界面。

2.5 本章小结

本章首先阐述了现代制造业下工业日志系统需要实现的要点：日志存储请求的高并发处理、日志数据处理规则的可灵活配置、日志数据的实时检索以及高可用性；随后，根据提出的实现要点，明确本文的分布式工业日志系统的需求，并提出日志系统的系统架构，分为日志收集层、日志缓冲层以及日志检索层；最后，本章对日志系统中各层的核心子系统进行了功能模块设计，确定了各层子系统功能模块的设计目标。

第三章 日志收集服务子系统的设计与实现

日志收集服务子系统是日志收集层的核心子系统，该子系统的设计目标是高并发处理日志存储请求，并降低磁盘 I/O 对服务器性能的影响。本文设计将该子系统分成网络模块、业务模块以及上报模块。本章主要阐述日志收集服务子系统中网络模块、业务模块和上报模块的设计与实现过程。

3.1 日志收集服务子系统的架构设计

日志收集服务子系统负责监听工业设备端的日志存储请求，接收设备日志数据，将日志数据写入磁盘文件，并传输到日志缓冲层。然而，大量工业设备不断产生设备日志，服务端需要能够低延迟地处理高并发的日志存储请求，同时高频率的日志传输请求导致大量日志文件的磁盘写入，大量的磁盘 I/O 会影响对工业设备端日志请求的响应性能。因此，日志收集服务子系统设计实现了基于 Reactor 模式的网络模块，将网络 I/O 事件分为连接建立事件和数据传输事件，监听到日志存储请求后，迅速响应并将之后的数据传输事件交由 worker 线程池处理，数据接收后先缓存在业务模块的 ringBuffer 中，然后再将多次缓存的日志数据一次写入磁盘，最后由上报模块将其上报到日志缓冲层中。日志收集服务子系统的架构图如图 3-1 所示。

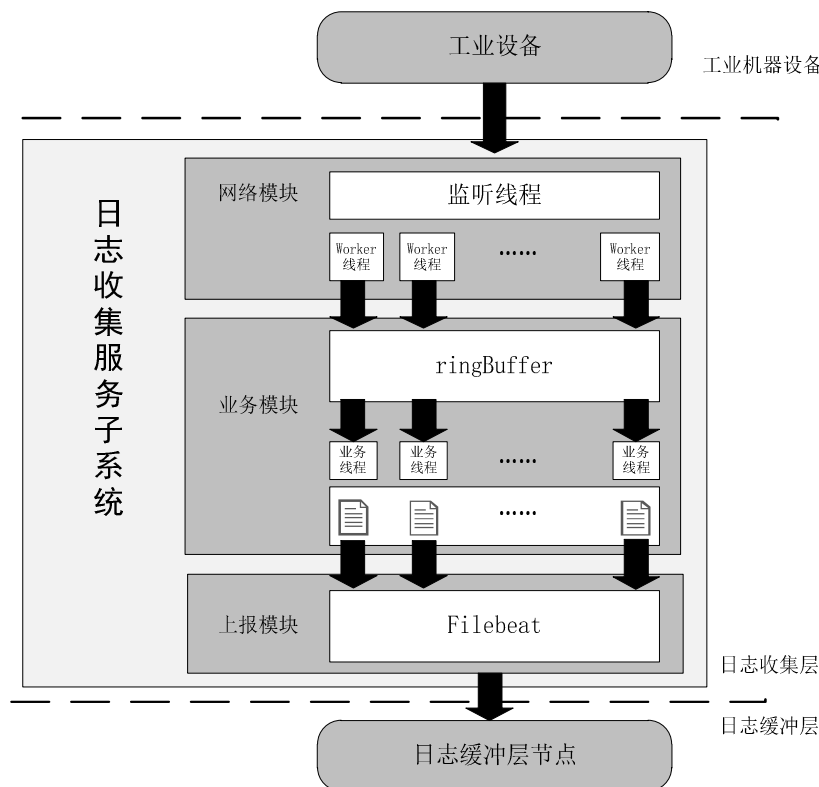


图 3-1 日志收集服务子系统架构图

日志收集服务子系统分为网络模块、业务模块和上报模块三个模块。网络模块设计了基于 Reactor 事件驱动的服务器，负责网络 I/O 事件，并将网络 I/O 事件分为与设备端的 TCP 连接建立事件和数据接收事件，采用多线程编程和线程池技术^[17]，实现高并发处理日志存储请求；业务模块设计实现了一个环形内存缓冲 ringBuffer，接收的日志先缓存在 ringBuffer 中，在保证实时性的情况下将多次接收的日志内容一次写入磁盘，极大减少了磁盘 I/O 次数。上报模块监控磁盘文件新增的日志记录，并将日志数据上报到日志缓冲层。

3.2 基于 Reactor 事件驱动的网络模块的实现

3.2.1 事件驱动模型

网络模块负责与工业设备端的网络 I/O 事件，主要分为连接建立事件和数据传输事件。网络模块需要能够高并发处理设备的日志存储请求，因此服务端不能阻塞在某个日志存储请求，导致其他请求得不到响应处理。网络模块采用事件驱动模型，来实现高并发处理工业设备的日志存储请求。事件驱动模型分为事件源、事件分发器和事件处理器三个构件，事件源产生事件对象，事件处理器在事件分发器上注册感兴趣的事件，并在事件发生时处理事件对象，事件分发器负责接收事件源的事件对象，并将事件分发到事件处理器进行事件处理^[18]。本文中工业设备端作为事件源，产生网络 I/O 事件，网络模块负责实现事件分发器和事件处理器，事件分发器将网络 I/O 事件分为连接建立事件和数据接收事件，建立连接事件由事件分发器快速响应，数据接收事件分发给事件处理器完成，事件驱动模型如图 3-2 所示。

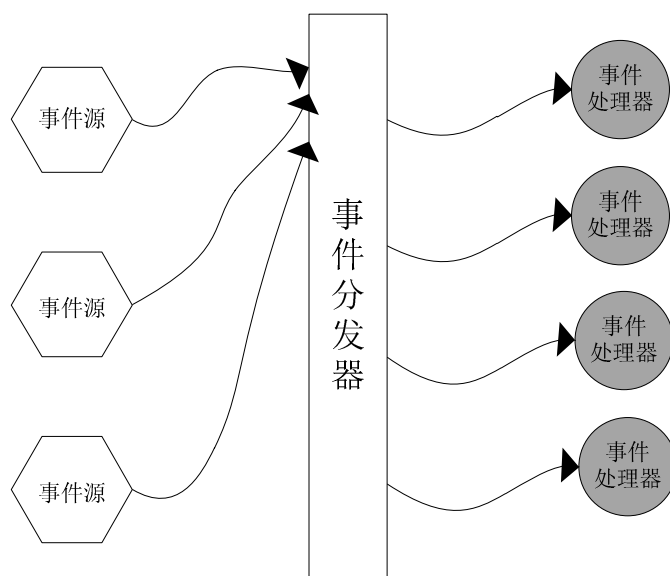


图 3-2 事件驱动模型

事件驱动模式分为基于同步 I/O 模型的 Reactor 模式和基于异步 I/O 模型的 Proactor 模式。在 Reactor 模式中，事件处理器定义事件到来时的事件处理逻辑，并在 Reactor 上注册，Reactor 上监听是否有事件到来，并在事件到来时将事件分发到事件处理器中进行处理^[19]。而在 Proactor 模式中，事件处理器主动调用异步操作接口函数，来指示事件到来时异步操作处理器的处理逻辑，并定义事件完成处理逻辑，在 Proactor 上注册，事件到来之后由异步操作处理器执行异步操作，异步操作完成后写入完成事件队列，Proactor 循环检测是否有异步事件完成，并从完成事件队列取出执行注册的事件完成处理逻辑^[20]。Proactor 使用异步 I/O，在性能方面比 Reactor 更为高效，但是 Proactor 需要操作系统支持异步 I/O。目前 windows IOCP 支持纯异步操作，但 windows 系统并不是服务器主流操作系统，linux 操作系统作为服务器主流操作系统，对纯异步的支持有限，因此，本文在网络模块的事件驱动模式上选择 Reactor 模式。

实现 Reactor 事件驱动模型，需要实现 Reactor 模式中的五个重要组件，分别为句柄 (Handle)、同步事件分离器 (Synchronous Event Demultiplexer)、事件处理器接口 (Event Handler)、Reactor 管理器 (Reactor) 以及具体事件处理接口 (Concrete Event Handler)^[21]。这些组件的相互关系如图 3-3 所示。

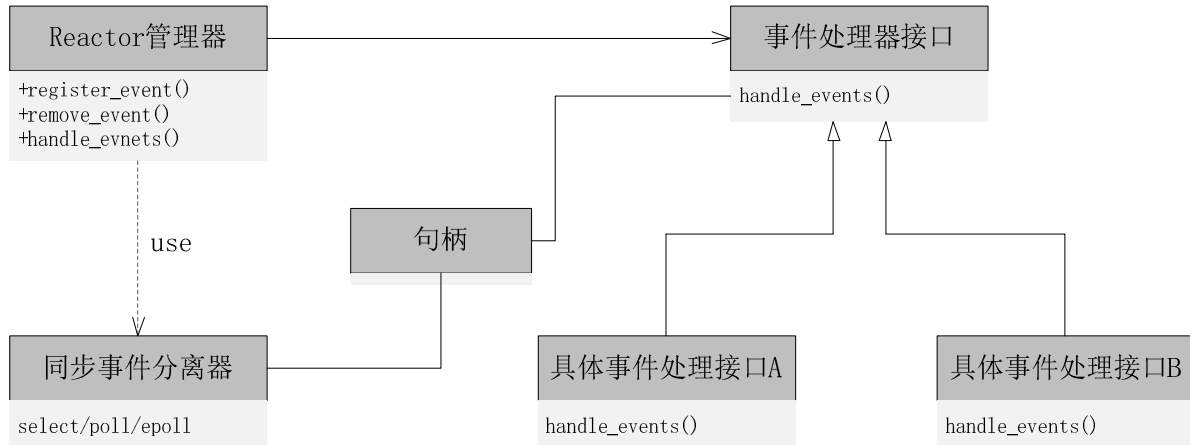


图 3-3 Reactor 模型关键组件

Reactor 模式中每个组件的详细说明如表 3-1 所示。

表 3-1 Reactor 模型关键组件说明

组件	说明
句柄	操作系统的句柄，注册到同步事件分离器上，关注句柄的可读、可写事件等
同步事件分离器	封装了操作系统的 I/O 复用，阻塞等待，监听注册的句柄的事件是否发生，当有事件发生时返回并通知相应句柄事件发生
事件处理器接口	亦称回调方法或者钩子函数，当句柄有事件发生时，执行回调函数
具体事件处理接口	事件处理器接口的具体实现，事件的具体处理逻辑
Reactor 管理器	Reactor 管理器管理句柄，句柄通过 Reactor 管理器向同步事件分离器注册、删除事件回调方法，当同步事件分离器检测到有事件发生，也会通知 Reactor 管理器调用相应注册的回调方法。

Reactor 模式中，应用向 Reactor 管理器注册句柄时，需要标识出该句柄关注的事件类型及其相应的具体事件处理接口^[22, 23]。当所有具体事件处理接口都已经注册完毕，Reactor 进入事件循环，在事件循环中，同步事件分离器阻塞等待事件到来，当有注册句柄关注的事件到来时，同步事件分离器通知 Reactor 管理器调用相应的具体事件处理接口^[24]。其时序图如图 3-4 所示。

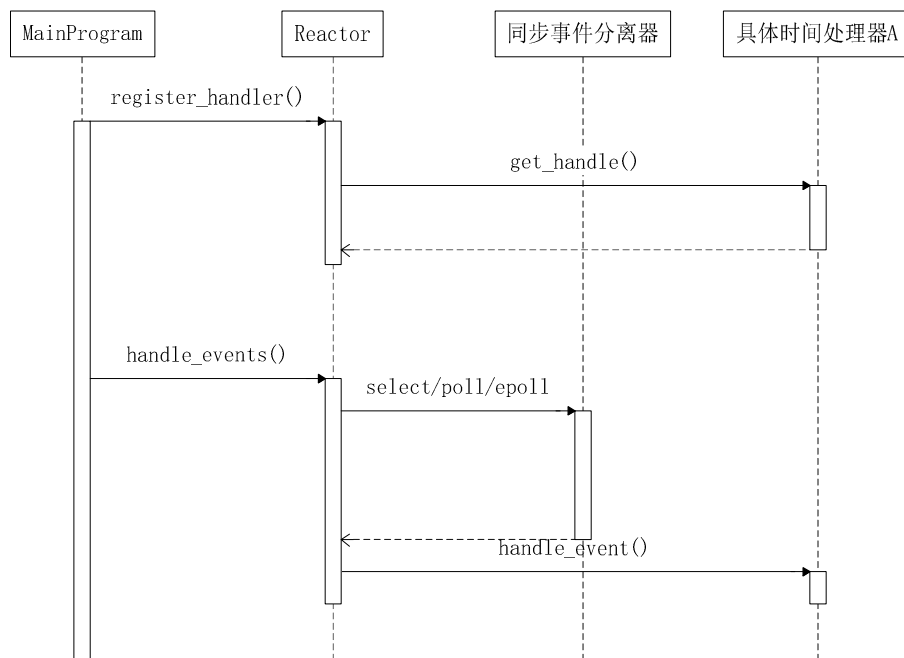


图 3-4 Reactor 模式时序图

3.2.2 Reactor 模式的具体实现

网络模块使用 C++ 语言编程，采用基于对象的编程方法，实现了 Reactor 事件驱动

模式。网络模块的类图如图 3-5 所示。

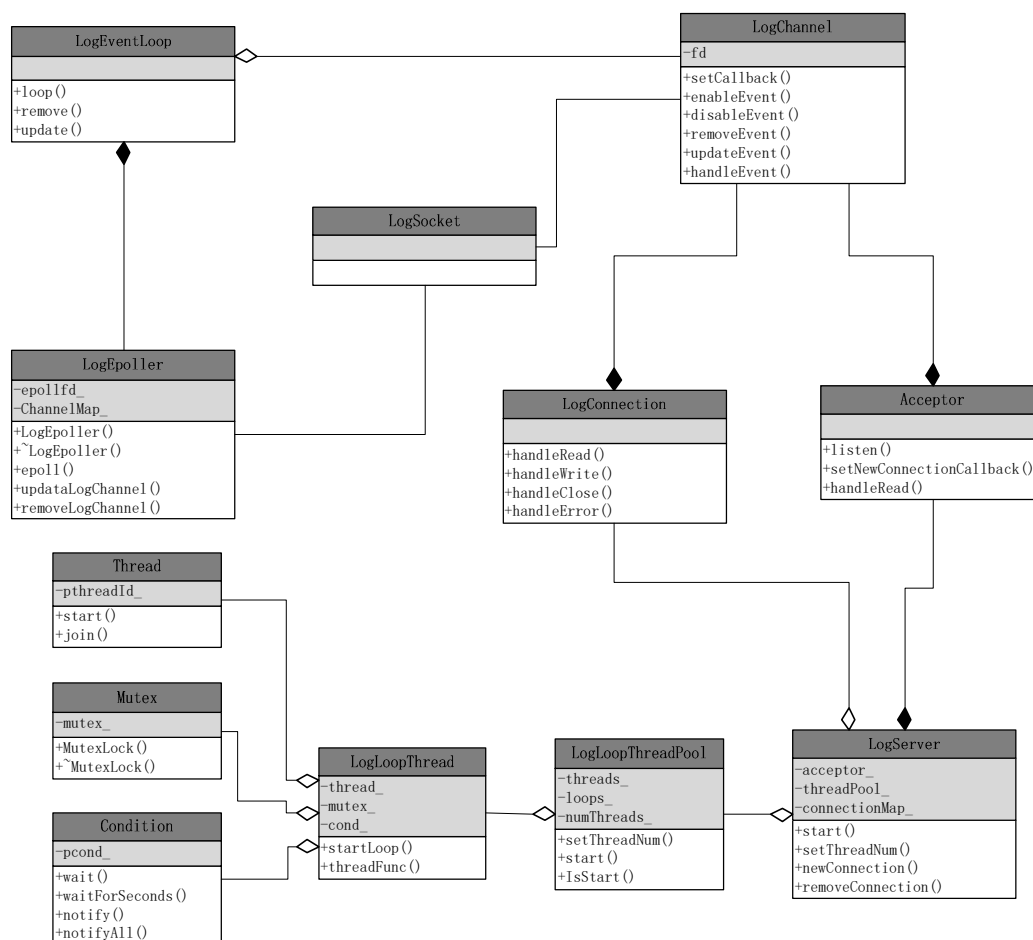


图 3-5 网络模块类图

LogEventLoop 类负责实现 Reactor 管理器, LogEpoller 封装了 linux 下的 epoll 接口, 负责实现同步事件分离器, LogSocket 类负责封装句柄, LogChannel 类负责实现事件处理器接口, Acceptor 类、LogConnection 类负责实现具体事件处理接口, LogServer 类负责封装网络模块服务端, LogLoopThreadPool 类封装 worker 线程池。其中 LogSocket 主要封装了与文件描述符相关的 linux 系统调用, 在此不再赘述。

(1) 同步事件分离器

同步事件分离器由 LogEpoller 类实现, 负责封装操作系统的 I/O 复用, linux 系统下的 I/O 复用有 select、poll 以及 epoll (linux 内核 2.6 版本之后提供)。epoll 的效率在大多数情况更为高效, epoll 的注册事件由内核提供一个内核事件表维护, 而 select、poll 的注册事件需要由用户空间传递到内核空间; 当有事件发生, select、poll 需要线性遍历所有的注册句柄来判断哪个句柄有事件发生, 而内核为 epoll 在内核准备了一张句柄就绪列表, epoll 只需判断列表是否为空即可。因此, 本文的 LogEpoller 类选择封装 epoll。

LogEpoller 类的主要函数如表 3-2 所示。

表 3-2 LogEpoller 类主要函数

函数	作用
LogEpoller	构造函数, 初始化, 创建 epoll 句柄等
~LogEpoller	析构函数, 关闭 epoll 句柄等
epoll	进入 epoll, 阻塞等待事件发生
updateLogChannel	修改/注册事件
removeLogChannel	移除事件

(2) 事件处理器接口

事件处理器接口由 LogChannel 类实现, 负责实现事件的注册/更新与删除、事件发生时回调方法的逻辑模板等, 该类的主要成员如下所示。

```
typedef boost::function<void()> EventCallback;
const int logfd_;
int events_;
int revents_;
EventCallback readCallback_;
EventCallback writeCallback_;
EventCallback closeCallback_;
EventCallback errorCallback_;
```

logfd_为监听该事件的句柄; events_是关注的事件类型, 该值在进入事件循环前设置; revents_是已发生事件类型, 该值由 LogEpoller 在事件发生时设置, readCallback_、writeCallback_、closeCallback_与 errorCallback_分别是该事件的读事件回调方法、写事件回调方法、连接关闭事件回调方法以及错误事件回调方法, 该值由具体事件处理接口类具体实现。LogChannel 类的主要函数表 3-3 所示,

表 3-3 LogChannel 类的主要函数

函数	作用
setCallback	设置回调方法
enableEvent	使关注某个类型事件(事件类型通过参数传递)
disableEvent	取消关注某个类型事件(事件类型通过参数传递)
removeEvent	移除关注事件类型(事件类型通过参数传递)
updateEvent	更新关注事件类型(事件类型通过参数传递)
handleEvent	事件发生时根据事件类型选择执行相应的回调函数

其中 handleEvent 负责实现事件回调的处理逻辑模板, 根据 revents_的值来选择相应

的事件回调方法。revents_值与选取的事件回调方法对应关系如表 3-4 所示。

表 3-4 revents_值与选取的事件回调方法对应表

revents_值	事件回调方法
POLLIN POLLPRI POLLRDHUP	readCallback_
POLLOUT	writeCallback_
POLLHUP	closeCallback_
POLLERR POLLNVAL	errorCallback_

(3) 具体事件处理接口

具体事件处理接口实现事件处理器接口中事件的具体处理逻辑，Acceptor 类和 LogConnection 类通过基于对象的编程方法实现具体事件处理接口，这两个类中均有一个 Channel 类成员，并在对象初始化过程中，对该 Channel 类成员初始化，对 Channel 类中的成员 readCallback_、writeCallback_、closeCallback_与 errorCallback_传递其实际的事件处理方法。

Acceptor 类负责连接事件处理逻辑, 其主要的类成员如图所示。Acceptor 类在初始化时首先绑定 IP 地址和监听端口, 在监听线程的 LogEventLoop 注册关注端口的读事件, 将读事件处理逻辑 newConnectionCallback 传递到 acceptChannel_的类成员 readCallback_。当读事件发生, acceptChannel_回调读事件回调处理逻辑 newConnectionCallback。

```
LogEventLoop* loop_;
LogSocket acceptSocket_;
LogChannel acceptChannel_;
```

LogConnection 类负责数据接收处理逻辑, 其主要的类成员如图所示。该类对象的生命周期从 Acceptor 回调读事件回调方法开始, 此时 Acceptor 返回一个连接套接字, LogConnection 初始化时在一个 worker 线程中注册监听该连接套接字的读写事件, 初始化 Connchannel_类对象的成员。inputBuffer_ 和 outputBuffer_负责存放接收的数据和待写入到 ringBuffer 中的数据。当数据传输完毕后设备通信端通过 close() 关闭连接后, LogConnection 类对象回调 closeCallback() 方法后由 LogServer 负责结束该对象生命周期。

```
LogEventLoop* loop_;
boost::scoped_ptr<LogSocket> Connsocket_;
boost::scoped_ptr<LogChannel> Connchannel_;
typedef Buffer std::vector<char>;
Buffer inputBuffer_,outputBuffer_;
```

(4) Reactor 管理器

Reactor 管理器由 `LogEventLoop` 类实现。监听线程与 `worker` 线程中都有一个 `LogEventLoop` 类对象，`LogEventLoop` 类对象初始化时对其 `LogEpoller` 类成员初始化，并提供注册 `LogEpoller` 类成员的接口 `update()` 和 `remove()`。初始化完成后进入事件循环 `loop()`，`loop()` 时序图如图 3-6 所示。`loop()` 首先调用 `poller_` 的 `epoll()` 方法阻塞等待注册事件到来，当监听到注册事件发生时，返回发生的注册事件集，回调事件集每个事件的事件处理逻辑。

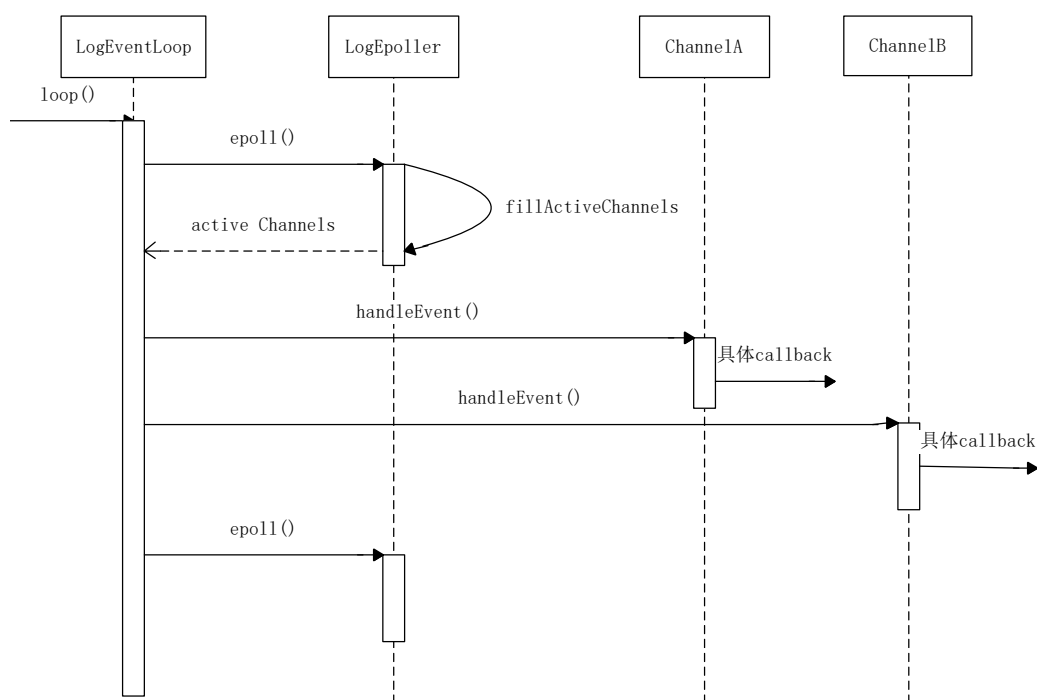


图 3-6 `loop` 方法时序图

`LogServer` 类负责封装网络模块的服务端，其包含一个 `Acceptor` 类成员和一个 `worker` 线程池，使用 `std::map` 结构管理了若干个 `worker` 线程与工业设备端建立的 TCP 连接 `LogConnection`。如图 3-7 所示，初始化过程中，`Acceptor` 在监听线程的 `LogEpoller` 注册关注其持有句柄的可读事件，该句柄使用系统调用 `listen` 指定监听服务端端口，当 `LogEpoller` 检测到服务端端口有可读事件时，回调 `Acceptor` 读事件处理函数 `newConnectionCallback`，该处理函数调用 `accept` 系统调用后返回一个连接套接字 `connfd`，创建一个关注 `connfd` 套接字可读可写事件的 `LogConnection` 对象，并从 `worker` 线程池选取一个 `worker` 线程，在该 `worker` 线程的 `LogEpoller` 注册关注 `connfd` 的读写事件，之后的数据传输过程交由该 `worker` 线程负责。

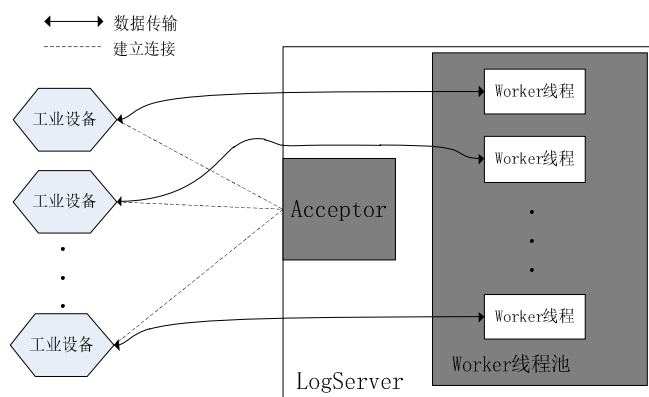


图 3-7 LogServer 内部结构图

3.2.3 worker 线程池的具体实现

在 Reactor 事件驱动模型中，网络模块若采用单线程来处理连接建立事件和日志数据传输事件，则线程会因为停留在数据传输事件过长导致高延迟处理连接事件，因此网络模块采用多线程编程。网络模块中将数据传输事件交由 worker 线程池处理。

常见的线程池实现方法大多基于生产者消费者模式，程序初始预先创建多个线程，作为消费者等待执行任务 Task，并设计一个任务队列，当有业务逻辑事件时，事件分发器作为生产者向任务队列写入一个任务 Task，线程池中的某一个线程争用到互斥锁后执行该任务 Task。然而该生产者消费者模式并不适用于高并发的业务场景，原因在于高并发场景下导致频繁的临界区加锁解锁操作，这极大的消耗系统资源，降低系统对外部请求的响应性能。

为了避免临界区加锁解锁的操作，本文创新地将 worker 线程与事件循环结合起来，每一个 worker 线程中都运行着一个事件循环 LogEventLoop。当 worker 线程被分配负责某个已连接套接字的设备日志接收事件时，只需要将该套接字注册到事件循环中即可。当该套接字有读事件发生时，事件循环会回调用户设置的读事件回调函数处理，从而避免了生产者消费者模式中频繁的加锁解锁问题。

如类图 3-5 所示，worker 线程池由 Thread、Mutex、Condition、LogLoopThread 以及 LogLoopThreadPool 类实现，Thread 类和 Condition 类分别封装线程相关的系统调用和条件变量，Mutex 类使用 RAII 技法封装互斥锁^[25]，关于 Thread、Condition 以及 Mutex 类具体的实现亦有很多成熟的开源方案，本节不再赘述。

实现 worker 线程池的核心类为 LogLoopThreadPool 类和 LogLoopThread 类。LogLoopThreadPool 类的主要成员如下所示。

```
int numThreads_;
boost::ptr_vector<LogLoopThread> threads_;
std::vector<LogEventLoop*> loops_;
```

numThreads_记录了线程池中的线程个数，threads_存储了所有 worker 线程指针，loops_存储了所有 worker 线程中的 LogEventLoop 类成员的指针，其中在实现时 threads_[i] 线程中的 LogEventLoop 类成员的指针保存在 loops_[i] 上，即 threads_ 与 loops_ 一一对应。

LogLoopThreadPool 类在构造函数中并不预先创建 worker 线程，该类创建 worker 线程的处理逻辑在 start() 方法，该类的核心实现代码片段如下所示。该方法是根据用户设置的线程个数，创建多个 LogLoopThread 类实例，并将实例保存在 threads_，将 LogLoopThread 实例负责事件循环的 LogEventLoop 类成员保存在 loops_。其中代码中调用 startLoop() 方法后启动一个 worker 线程并且进入事件循环，该方法为 LogLoopThread 类的核心处理逻辑。

```
void LogLoopThreadPool::start(){
    .....
    for (int i = 0; i < numThreads_; ++i){
        LogLoopThread* thread = new LogLoopThread();
        threads_.push_back(thread);
        loops_.push_back(thread->startLoop());
    }
    .....
}
```

在阐述 startLoop() 方法前，首先介绍 LogLoopThread 类的主要成员。

```
LogEventLoop* loop_;        //事件循环
Thread thread_;              //线程
MutexLock mutex_;            //互斥锁
Condition cond_;              //条件变量
```

startLoop() 的处理逻辑是首先在主线程中调用 thread_ 的 start() 方法启动一个 worker 线程，然后进入临界区后检查 loop_ 是否为空，若为空则进入条件等待。worker 线程首

先创建一个局部的 `LogEventLoop` 类变量 `lop`, 然后进入临界区后将类成员 `loop_` 指向 `lop`, 并唤醒条件变量, 退出临界区后对 `lop` 变量调用 `loop` 函数进入事件循环。主线程被条件唤醒后返回 `loop_` 变量。

3.3 基于环形内存缓冲 `ringBuffer` 的业务模块的实现

(1) 高并发场景下磁盘 I/O 的问题与技术选型

业务模块负责日志数据的磁盘写入。业务模块涉及磁盘 I/O 读写, 传统同步日志在每次接收日志数据后陷入一次 `write` 系统调用方法, 在并发量不多的情况下, 这种传统同步日志的方法对服务器性能影响极小。但在高并发的场景下, 频繁的磁盘读写会陷入等量的系统调用, 用户态与内核态之间的切换造成一定的系统开销, 影响系统性能。

通常采用增加缓存使得磁盘写入次数减少的方法提高磁盘 I/O 性能。因此, 在高并发的工业场景下, 本文设计实现了一个环形内存缓冲 `ringBuffer` 作为日志数据缓冲, 上游网络模块的 `worker` 线程作为生产者将接收的日志数据写入到 `ringBuffer`, 业务模块的业务线程池作为消费者读取 `ringBuffer` 中的日志数据并写入到磁盘。考虑到在业务模块下由于使用环形内存缓冲使得争用锁的频率并不高, 因此业务线程池的设计实现采用传统的生产者消费者模型, 基于生产者消费者模型的线程池已经有许多优秀的开源实现方案, 本局不再赘述。本节主要阐述 `ringBuffer` 的设计与实现过程。

(2) 环形内存缓冲 `ringBuffer` 的具体实现

设计环形内存缓冲的目的是提供一个日志数据缓冲, 将多次接收的设备日志一次写入磁盘, 减少磁盘 I/O 的次数, 从而降低盘 I/O 对服务器性能的影响。内存缓冲可以通过线性表、链表等结构实现, 考虑到内存空间可由程序随时向操作系统申请, 内存缓冲可能动态扩展, 这会涉及到元素的添加和删除操作, 在线性表删除元素的复杂度为 $O(N)$, 而链表上添加或删除节点的操作均只有 $O(1)$ 的复杂度, 同时考虑到内存缓冲会被重复使用, 因此本文采用环形链表结构作为内存缓冲的数据结构。

环形内存缓冲的模型图如图 3-8 所示。环形缓冲维护一个生产指针和一个消费指针, 网络模块中的 `worker` 线程将接收的日志内容写入到生产指针指向的节点, 业务线程拉取消费指针指向的节点中缓存的日志内容。环形缓冲使用互斥锁机制来保证对环形缓冲操作的完整性, `worker` 线程与业务线程要读写环形缓冲需要先获得锁, 才能对环形缓冲进行读写操作。生产指针和消费指针均往所在节点的后继节点方向移动。

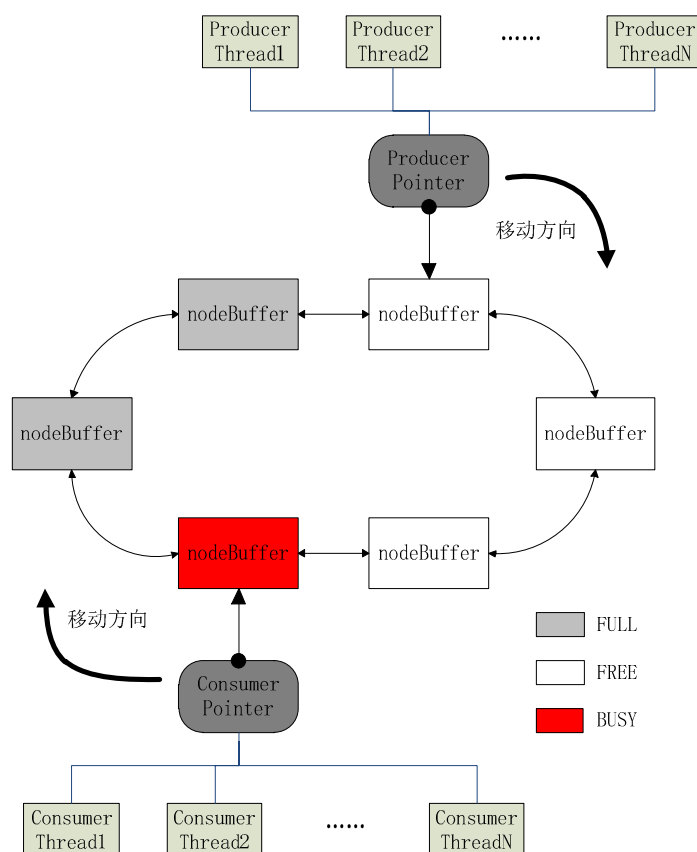


图 3-8 环形内存缓冲 ringBuffer 模型图

环形链表上的节点由 `nodeBuffer` 类定义，`nodeBuffer` 类定义如下所示。`nodeBuffer` 类中保存了节点在链表上的前驱节点的 `nodeBuffer` 指针 `prev` 和后继节点的 `nodeBuffer` 指针 `next`，变量 `total_len` 和 `used_len` 分别表示节点所能保存的总字节数和已被使用的总字节数，`char` 指针变量 `date` 指向节点缓冲区起始点，定义了表示节点状态的枚举类型 `buffer_status`，枚举值有 `FREE`、`FULL` 和 `BUSY`，其中 `FULL` 状态表示该节点不能再缓存更多的数据，等待写入到磁盘，`FREE` 状态表示节点还能够继续缓存日志数据，`BUSY` 状态表示该节点的数据正在写入到磁盘，节点状态保存在变量 `statu`。

```
class nodeBuffer{
    public:
        enum buffer_status    //节点状态
        {
            FREE,
            FULL,
            BUSY
        }
};
```

```

};

nodeBuffer* prev;    //前驱节点
nodeBuffer* next;    //后继节点

private:

uint32_ttotal_len;    //节点缓冲区总字节数
uint32_tused_len;     //节点缓冲区已使用字节数
char* data;           //节点缓冲区起始点
buffer_statusstatu;   //节点状态
}
    
```

网络模块的 worker 线程往缓冲池写入日志内容的流程如图 3-9 所示。

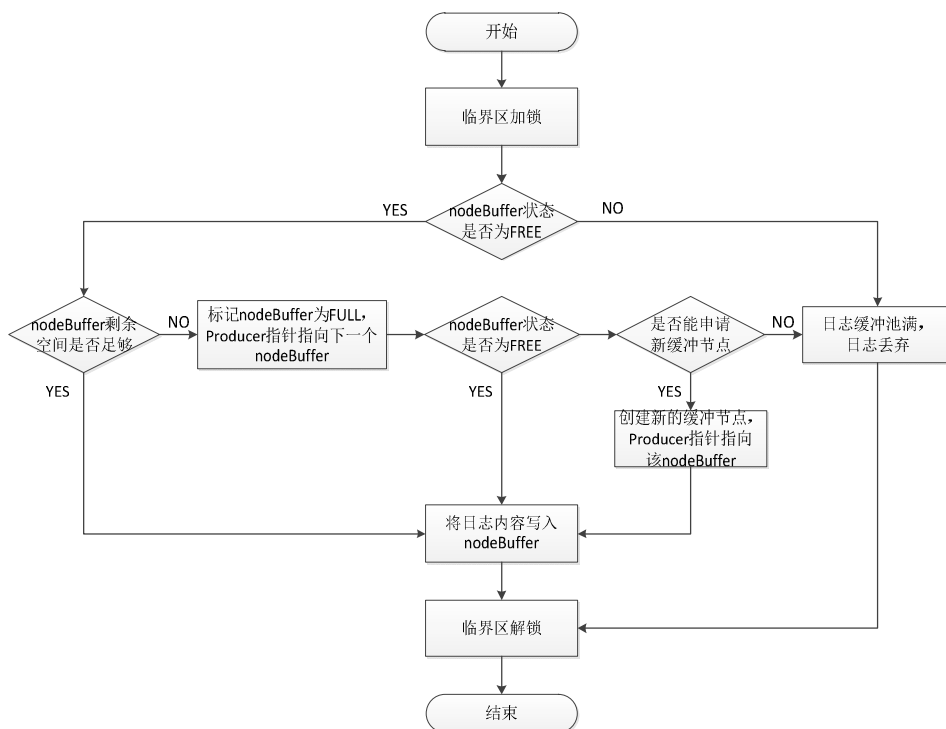


图 3-9 往缓冲池写入日志数据流程图

如图 3-9 所示,网络模块的 worker 线程获取互斥锁后进入临界区,判断当前 producer 生产指针指向的 nodeBuffer 节点是否为 FREE 状态,若不是 FREE 状态,表示当前环形内存缓冲所能申请的内存空间已经达到上限,且环形内存缓冲无 FREE 节点用来保存日志数据,退出临界区;若 nodeBuffer 节点为 FREE 状态,判断当前 producer 指针指向的节点缓冲区是否足够容纳需要存入的日志内容,若足够容纳,将日志内容写入到该节点,更新记录该节点的状态变量后退出临界区;若不足以容纳,标记当前节点状态为 FULL,produce 指针指向下一个 nodeBuffer 节点,然后继续判断该节点状态是否为 FREE,若为

FREE，表示该 `nodeBuffer` 能够使用，将日志内容写入到该节点，更新记录该节点的状态变量后退出临界区；若不为 FREE，表示当前环形缓冲已经无 FREE 节点可供数据缓存，尝试向系统申请内存，若能够申请内存，则创建新缓冲节点，`producer` 指向该节点，将日志内容写入到该节点，更新记录该节点的状态变量后退出临界区；若无法申请，则直接退出临界区。

业务线程拉取环形内存缓冲的日志数据的流程如图 3-10 所示。

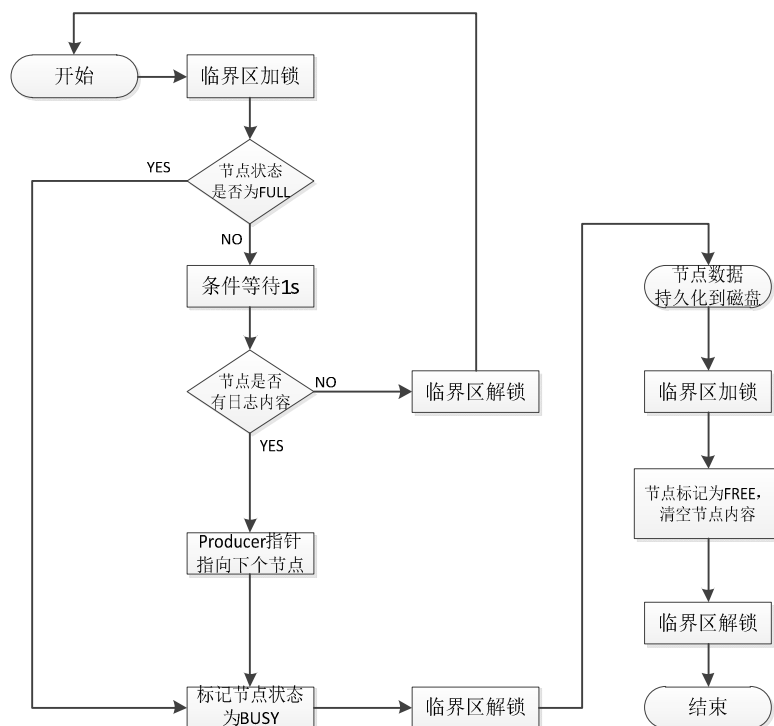


图 3-10 拉取缓冲池日志数据流程图

如图 3-10 所示，流程分为以下六个步骤来执行。

- 步骤 1：临界区上锁，判断 `consumer` 指针指向的节点状态是否为 FULL
 - 1) 若为 FULL，进入步骤 4。
 - 2) 若不为 FULL，条件等待 1 秒后进入步骤 2。
- 步骤 2：再次判断 `consumer` 指针指向的节点状态是否为 FULL
 - 1) 若为 FULL，进入步骤 4。
 - 2) 若不为 FULL，进入步骤 3。
- 步骤 3：判断节点是否存有待写入磁盘的日志内容
 - 1) 若有日志内容，`producer` 指针指向下一个节点，进入步骤 4。
 - 2) 若没有日志内容，临界区解锁，回到步骤 1。
- 步骤 4：标记 `consumer` 指针指向的节点状态为 BUSY，`consumer` 指针指向下一个节点

点，解锁临界区。进入步骤 5。

- 步骤 5：节点数据持久化到磁盘，进入步骤 6。
- 步骤 6：临界区加锁，标记节点状态为 FREE，清空该节点的日志数据，然后退出临界区。

3.4 基于轻量级日志采集组件 filebeat 的上报模块的实现

(1) 技术选型

上报模块负责将业务模块写入磁盘的日志数据上报到日志缓冲层。上报模块需要实现监测磁盘文件新增的日志记录，且尽可能减少运行时系统资源的开销。

目前开源的日志采集组件有 logstash、logtail、fluentd 以及 filebeat 等可供选择，本文主要从功能和性能方面来评估这四款日志采集组件，在功能方面，本节主要从采集组件是否实现日志数据的采集功能、数据处理功能以及功能扩展来评估；在性能方面，本节主要从采集组件的采集性能和消耗资源来评估。表 3-5 进行了以上四款开源日志采集组件的功能方面和性能方面的对比。

表 3-5 开源日志采集组件的功能方面和性能方面对比

		Filebeat	Logstash	Logtail	Fluent
功能	文件采集	支持	支持	支持	支持
	过滤	不支持	插件扩展	正则	插件扩展
	压缩	支持	插件扩展	lz4	插件扩展
	功能扩展	支持	支持插件	支持插件	支持插件
性能	采集性能	单核 15M/s	单核 2M/s	单核 160M/s 正则 20M/s	单核 3~5M/s
	消耗资源	内存消耗极低	内存占用极大	内存消耗中等	内存占用极大

本文设计的上报模块在功能上无需实现过滤功能，同时更关注上报模块能够占用较少的硬件资源。根据表 3-4 中对日志采集组件的功能和性能方面的对比，上报模块采用 Filebeat 来实现将日志文件传输到日志缓冲层。

(2) 上报模块的具体实现

Filebeat 的架构如图 3-11 所示，它由 prospectors(探查器)和 harvester(采集器)两个主要组件组成^[26]。当启动 filebeat 时，filebeat 读取指定的配置文件，启动一个或多个 prospectors，prospectors 会探查配置文件中指定的每一个本地路径下的日志文件。对于

prospectors 探查到每一个符合条件的日志文件, filebeat 为其指定一个 harvester。harvester 读取日志文件获取新增的数据记录, 并将其发送给 spooler。spooler 汇聚所有的日志记录, 然后将这些数据传输到配置文件中指定的目的地。

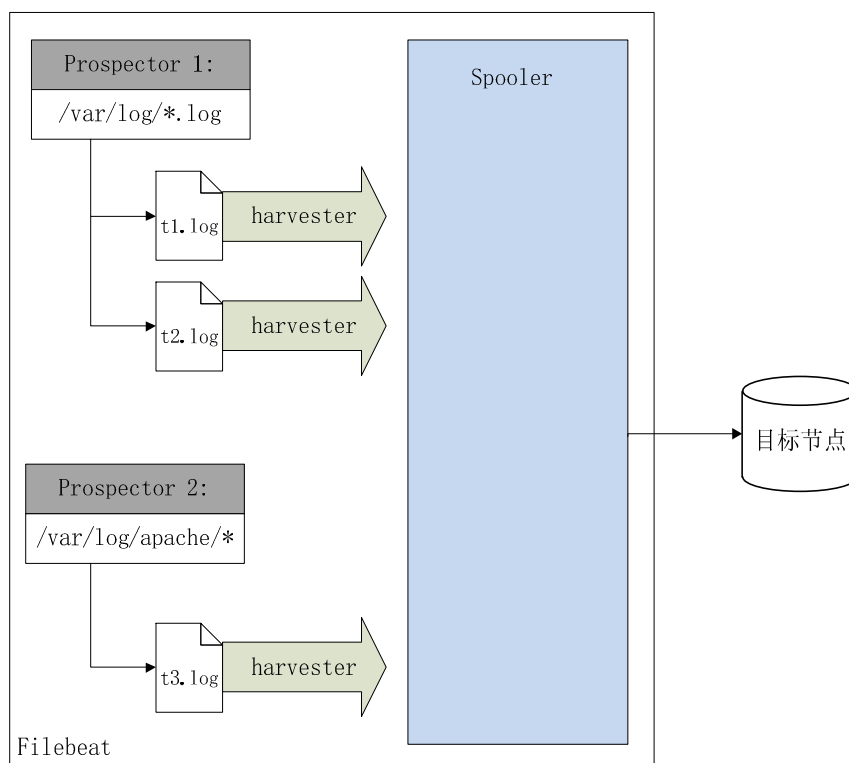


图 3-11 Filebeat 架构图

harvester 负责实现监测磁盘文件新增的日志记录, 其监测流程如图 3-12 所示。filebeat 将文件上一次的读取状态记录到 registry 文件, harvester 打开文件时首先从 registry 文件获取文件上一次的读取状态, 然后逐行采集数据直到读取完文件的最后一行, 并将数据输出到目标地点。当读取完文件最后一行后, 启动计时器 counter, 在 close_inactive 时间内若文件若有更新, 则计时器 counter 清零并继续采集数据到最后一行, 否则关闭该文件句柄。关闭文件句柄后等待一定的时间间隔 backoff 后重新打开该文件进行采集。时间间隔不是固定的, 有 backoff、max_backoff、backoff_factor 三个参数参与设置时间间隔的值, 参数单位为秒。backoff 初始值为 1, max_backoff 定义了最大的时间间隔, 默认为 10。当文件有更新时, backoff 值还原为 1; 当文件未有更新时, 每关闭一次采集器, backoff 的值设置为 backoff 当前值与 backoff_factor 的乘积。

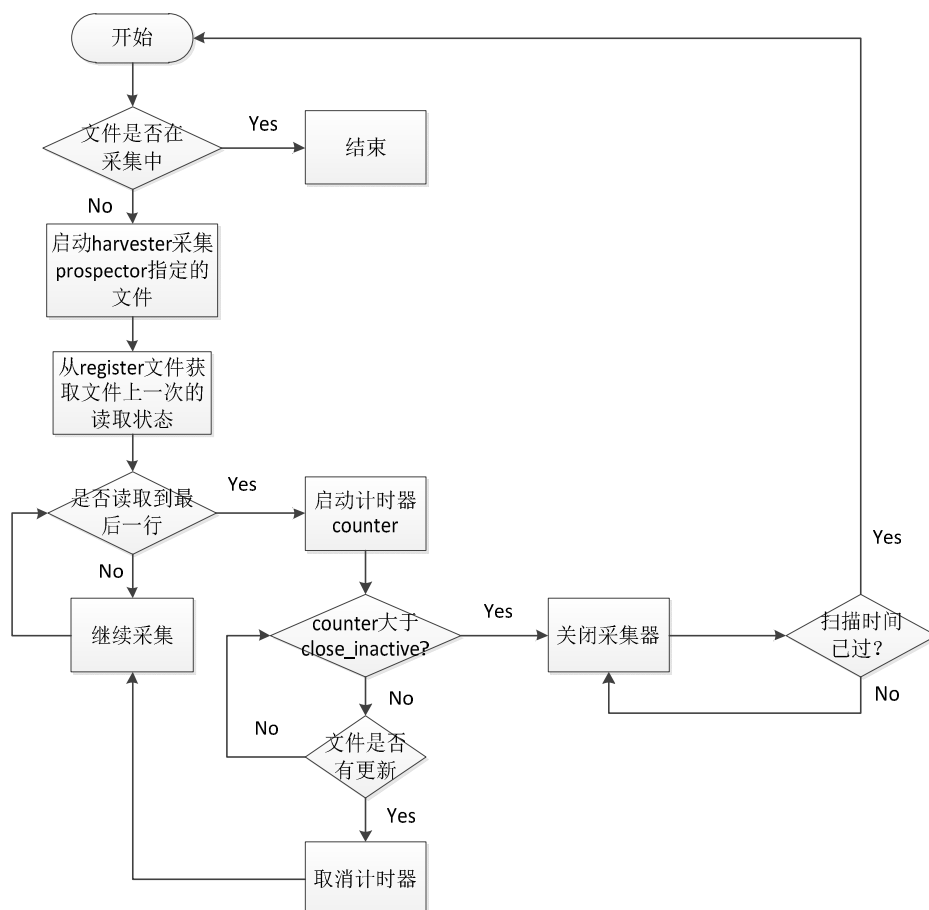


图 3-12 harvester 监测磁盘文件流程图

Filebeat 与缓冲层数据处理模块的 logstash 对接，filebeat 使用背压敏感协议，可以自动与下游的 logstash 匹配传输速度。当感知下游 logstash 处于忙碌状态时，则会告诉 filebeat 减慢速度。一旦拥堵解决，filebeat 会恢复速度传输数据。

filebeat 的配置文件 filebeat.yml 在安装目录下，本文上报模块对 filebeat 的配置文件如下。

```

filebeat.prospectors:
- input_type: log
  paths:
    - /home/hqw/logDir/*.log
close_inactive: 5m
output.logstash:
  hosts: ["192.168.234.13:5044"]
    
```

该 filebeat.yml 文件中 filebeat.prospectors，用以指定一个 prospectors 列表，这些 prospectors 用以确定需要监测是否有新增记录的日志文件。其中 input_type 指定输入类

型，其值为“log”或“stdin”，前者表示读取日志文件的每一行，后者表示从标准输入中读取。`paths` 指定输入日志的文件路径，`close_inactive` 设置一个最大时间为 5 分钟，当超过该时间 `harvester` 关闭读取该文件。`output.logstash` 表示输出到 `logstash`，`hosts` 指定 `logstash` 服务所在的主机和端口。

3.5 本章小结

针对高并发处理日志存储请求以及减少磁盘 I/O 对服务器性能的影响的设计目标，日志收集服务子系统设计实现了网络模块、业务模块以及上报模块。网络模块基于 `Reactor` 事件驱动，将网络事件分为连接事件与数据传输事件，并创建监听线程和 `worker` 线程池，监听线程负责快速响应连接事件，`worker` 线程池完成数据传输任务，实现了低延迟处理高并发日志请求；针对高频率的日志数据写入磁盘请求，业务模块基于环形内存缓冲 `ringBuffer`，将日志数据先存放在内存空间，将多次的日志内容一次写入磁盘，极大地减少了磁盘 I/O 的次数，降低磁盘 I/O 对服务器性能的影响；上报模块采用轻量级日志采集组件 `filebeat`，能够监控文件新增日志记录，将日志数据上报到日志缓冲层。

第四章 日志缓存服务子系统的设计与实现

日志收集服务子系统将日志数据上报到日志缓存服务子系统，日志缓存服务子系统是日志缓冲层的核心子系统，该子系统的设计目标是实现日志数据处理规则的灵活配置以及对日志数据的缓存队列。本文设计将该子系统分数据处理模块和缓存模块。本章主要阐述日志缓存服务子系统中数据处理模块和缓存模块的设计与实现过程。

4.1 日志缓存服务子系统的架构设计

日志缓存服务子系统负责接收日志收集层的数据，对日志数据处理后写入缓存队列，其中日志数据处理规则可以灵活配置。该核心子系统的架构图如图 4-1 所示。

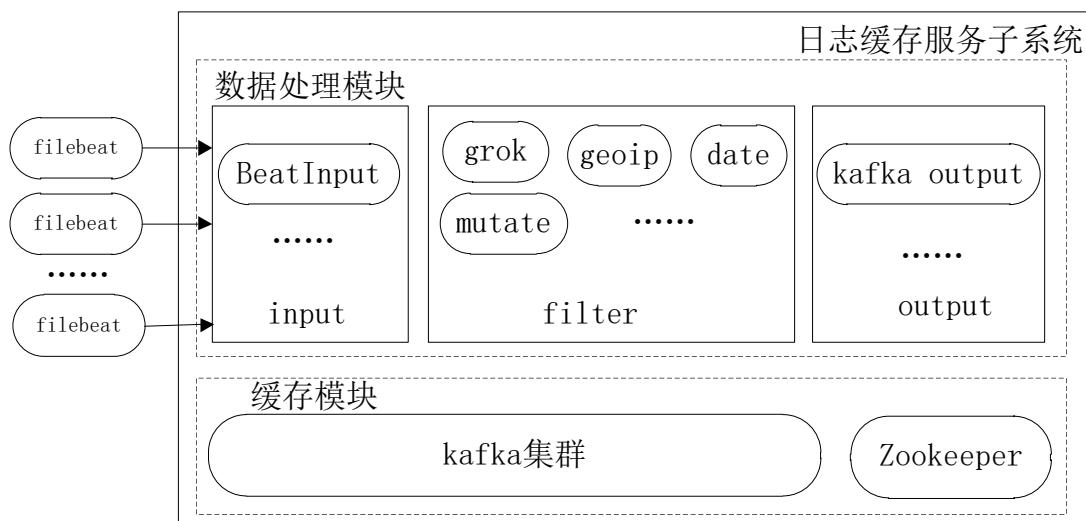


图 4-1 日志缓存服务子系统架构图

该服务子系统分为数据处理模块和缓存模块两个模块。数据处理模块负责接收来自日志收集层的日志数据，对日志数据进行处理。该模块使得产生日志的设备端只需要发送设备原始格式的日志数据即可，而无需为了适配处理模块的数据处理规则而添加额外的字段，对于 A 公司新引进设备，技术人员只需要了解新设备的原始日志输出格式，并在数据处理模块的配置文件中动态地添加匹配规则即可对新设备的日志进行处理，添加匹配规则应该尽可能做到简单易配置。缓存模块负责实现对处理后的日志数据进行缓存，并保证数据的高可用性。

4.2 数据处理模块的实现

4.2.1 开源数据处理组件对比

数据处理模块涉及对日志数据的处理，目前著名的日志收集系统有 Scribe、Flume NG、Chukwa 以及 logstash，这些系统在收集端均实现了对日志数据进行处理，然而在数据处理的实现方式上有所不同，Scribe 需要日志产生端在日志内容附加一个额外的 category 字段，其数据处理端根据 category 字段值不同来进行分类处理；Flume NG 提供了拦截器 (Interceptor) 来进行数据处理，拦截器通常也需要日志数据中存在相应的字段才能进行数据拦截，这同样可能需要日志产生端附加一些额外的字段^[27]；Chukwa 对不同的数据源可以采用不同的 Adaptor 来收集，然而 Adaptor 对于来自同一个数据源的不同类型日志并不具备处理能力，同时 chukwa 的 collector 端时延性高，与本文日志数据的实时检索存在矛盾；logstash 无需日志产生端附加额外的字段，其能够通过丰富的插件生态系统对非结构化数据进行数据处理，且数据处理规则的配置较为简洁灵活^[28]。

本文考虑在工业场景下，工业设备端为输出的设备日志附加额外的字段并不合理，若数据处理规则新添加了一个处理规则，则所有的工业设备端均需要为此再添加一个额外的字段，这对于技术人员无疑是一个相当繁琐的工作。因此，数据处理模块采用数据处理开源组件 logstash。

4.2.2 数据处理模块的具体实现

logstash 作为一款服务端数据处理工具，常用于对日志数据的传输、格式化处理和输出^[29]。Logstash 的作者是世界著名的虚拟主机托管商 DreamHost 的运维工程师 Jordan Sissel，2013 年 Logstash 被 Elasticsearch 公司收购。

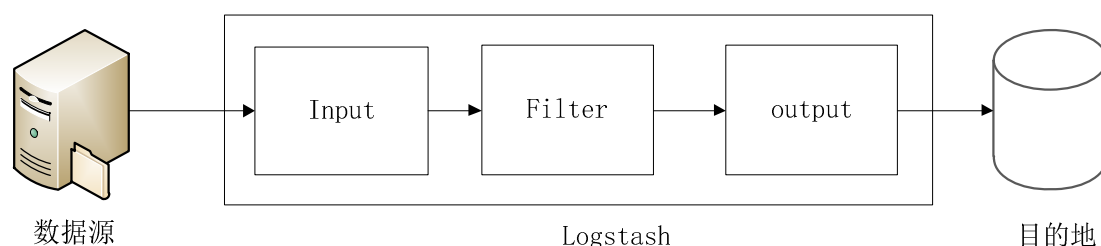


图 4-2 logstash 数据处理过程

如图 4-2 所示，Logstash 将数据的处理过程分成三个事件阶段，分别为输入阶段 (input)、处理阶段 (filter) 和输出阶段 (output)。其中输入阶段和输出阶段分别配置数据源和数据去向，在本文设计中，输入阶段采用 Beat input 插件来实现对 filebeat 传输的

数据的接收，输出阶段采用 Kafka output 插件实现日志数据输出到缓存模块的 kafka 集群。本节重点阐述 logstash 处理阶段的详细设计。

编写 logstash 配置文件，并在 logstash 启动时指定，logstash 会按照配置文件设置的规则来对数据进行处理。logstash 配置文件的格式如下：

```
input{
    .....
}
filter{
    .....
}
output{
    .....
}
```

开发人员通过对 filter 块进行配置数据处理规则，实现 logstash 对数据的处理。logstash 能够对非结构化的原始日志数据进行结构化，并按照实际需求拼接、增添字段。本节通过一个具体的设备日志处理过程来阐述数据处理模块如何实现数据处理规则的灵活配置。以下是一工业设备的原始日志数据，该日志数据由日志产生时间、设备 ip、日志等级、设备节点编号和日志信息组成。

```
2019-02-2611:13:36.576 181.144.250.19 WARN 49599 "High Voltage Load!"
2019-02-2611:13:37.576 233.140.14.70 ERROR 27500 "Machine Malfunction"
2019-02-2611:13:38.576 168.125.234.93 INFO 26740 "Normally running."
```

首先采用 logstash 的 grok 插件可以对非结构化的日志记录进行结构化，根据设备输出日志格式在 grok 中编写正则表达式匹配规则，grok 将设备的原始日志记录拆分成 log_date、machine_ip、log_level、machine_id 和 log_info 五个字段。

切分字段后，machine_ip 字段值由 geoip 插件处理，geoip 能根据 ip 地址来得到 ip 地址所在的地理位置，在这里通过 machine_ip 能够得知设备的地理位置。logstash 在处理时会自动加上 timestamp 字段，其中保存着 logstash 处理该条日志记录的时间，该字段在此处并无参考价值，本文采用 date 插件将该字段值替换为 log_date 字段的值，然后使用 mutate 插件删除 log_date 字段。

logstash 的配置文件如下所示。

```
input {
  beats { port =>5044 }
}
filter{
  grok{
    match=>{
      "message"=>"%{TIMESTAMP_ISO8601:log_date}%{WORD:log_level}%{IP:machine_ip} %{NUMBER:machine_id}%{QUOTEDSTRING:log_info}"
    }
  }
  geoip{
    source =>"machine_ip"
  }
  date{
    match => ["log_date", "yyyy-MM-dd HH:mm:ss,SSS"]
    target =>"@timestamp"
  }
  mutate{
    remove=>"log_date"
  }
}
output {
  kafka{
    bootstrap_servers=>"192.168.234.12:9092,192.168.234.13:9092,192.168.234.15:9092"
    topic_id =>"devicelog"
  }
}
```

在 input 块中，beats 表示使用 Beat input 插件，port 选项表示 Beat input 插件监听的端口，因为 filebeat 配置数据输出的端口为 5044，所以 port 选项值设置为 5044。

在 filter 块中, grok 插件对保存日志消息的 message 字段进行正则化匹配, 正则匹配的语法为 “%{SYNTAX:SEMANTIC}”。其中 SYNTAX 是定义的正则式表达名字, 代表一种正则匹配规则, grok 插件已经提供了许多日志文件中常使用的正则匹配规则。SEMANTIC 表示匹配结果的标识, 通过该字段可以得到匹配后的结果。切分字段后, 使用 geoip 插件, 可以根据 IP 地址获取设备的地理位置, logstash 提供了一个免费的 IP 地理定位数据库, source 选项指向需要查询的 ip 地址, 设置为 machine_ip 字段。date 插件实现修改 logstash 附加的 timestamp 字段的值, 用 log_date 字段值覆盖, 在 match 中对 log_date 的格式进行指定, target 指向需要被修改的字段, 此处指向 timestamp 字段。timestamp 字段已经保存了 log_date 字段的值, 因此 filter 块最后使用 mutate 插件删除 log_date 字段, remove 指定需要被删除的字段名。

在 output 块中, 采用 Kafka output 插件实现将日志数据输出到缓冲模块的 kafka 集群中, bootstrap_servers 字段用来配置数据发送的目的地, 默认为本地地址的 9092 端口, 在此处设置成 kafka 集群的三个节点。topic_id 是必选项, 用来设置日志消息在 kafka 集群中所属的 topic。

以上是工业设备的原始日志数据的处理过程, 该过程采用 grok、geoip、date、mutate 等插件实现了对日志数据字段切分、IP 地址定位、字段转换以及删除等数据处理操作, 可以看出, 对于不同输出格式的设备日志, 数据处理模块能够通过添加在数据处理规则中加入相应处理规则实现对原始设备日志的处理, 且编写的数据处理规则的行数极少, 这得益于 logstash 提供的丰富的插件, 技术人员也可以编写代码实现适合工业数据的自定义插件。同时, logstash 支持动态修改日志数据处理配置文件, 在第一次启动时加入参数 “--config.reload.automatic”, 当配置文件发生更改时, logstash 能自动重新加载配置文件而无需手动重新启动 logstash。

4.3 缓存模块的实现

4.3.1 技术选型

缓存模块负责实现日志数据的队列缓存, 保证日志数据的高可用。Redis 和 Kafka 均能实现对日志数据的队列缓存, redis 将日志数据存放在内存中, 因此处理速度比 kafka 快, 然而 redis 作为日志数据的缓存可能会发生数据丢失, 且吞吐量较低, 适用于数据持久性需求不高的场景^[30]; kafka 默认提供了日志数据的持久化, 保证数据的高可靠, 并具有高吞吐量的特点。考虑在工业场景下, 工业设备不断产生设备日志, 提供一个高

吞吐、数据高可用的缓存模块更适合该场景，因此本文的缓存模块采用 kafka 实现日志数据的缓存，保证数据的高可用。本文通过 zookeeper 管理 kafka 集群配置，实现 kafka 集群中节点之间的协调一致，对同一个 topic 下设置多个分区提高 kafka 集群的吞吐量，采用 kafka 副本机制保证数据的高可用性。下面将详细阐述 kafka 集群协调一致、日志数据高可用的设计和实现过程。

4.3.2 kafka 集群协调一致的实现

kafka 集群通过 zookeeper 管理集群配置，实现 kafka 集群各个节点的协调一致。Kafka 集群结构图如图 4-3。Producer 端通过 zookeeper 获取 Broker.list 列表和与 Topic 相关的元数据，然后与 Topic 下每个分区 leader 节点建立 socket 来传输信息。Broker 端在启动 kafka 服务时会向 zookeeper 注册自己的 broker 信息，并通过 zookeeper 来监控分区 leader 节点的存活状态。Consumer 端通过 zookeeper 注册 Consumer 相关的信息，比如该 Consumer 节点消费的 partition 列表，同时也通过 zookeeper 获取 Broker.list 列表，并和分区 leader 节点建立 socket 进行通信。

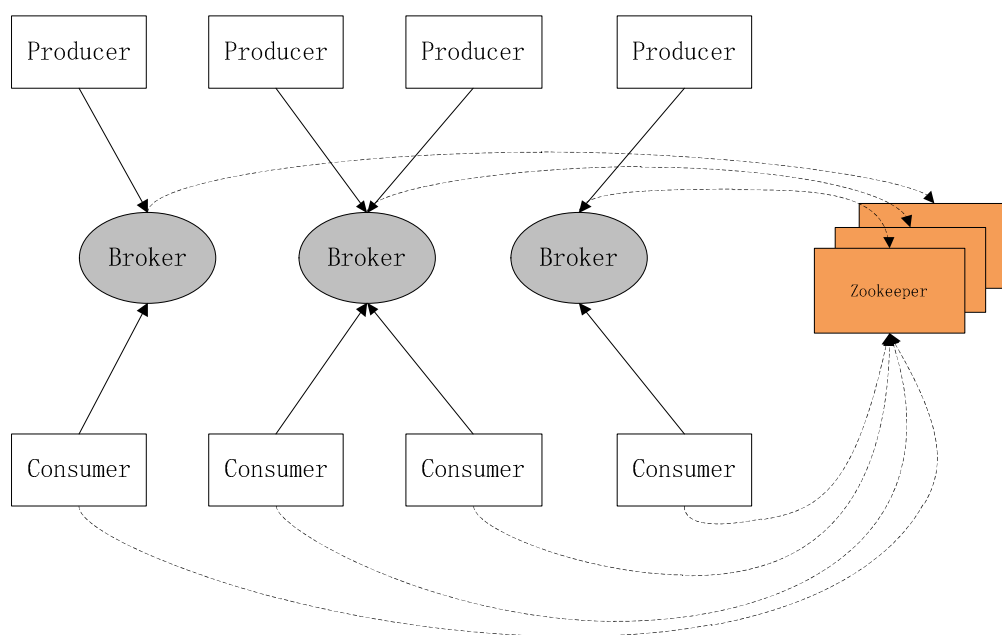


图 4-3 kafka 架构图

Zookeeper 是一个分布式协调服务框架，以 Paxos 算法为基础，通常用于解决分布式集群中管理协调的问题，如分布式配置管理、统一命名服务、集群节点状态协调等问题^[31, 32]。

zookeeper 使用 zab 协议(zookeeper Atomic Broadcast, 原子广播协议)实现分布式数据一致性^[33]。zab 协议中 zookeeper 模式分为恢复模式和广播模式。恢复模式主要进行

leader 节点选举，然后进行 follower 节点与 leader 节点的状态同步，之后进入广播模式。

在广播模式下,zookeeper集群中节点扮演 leader 角色或者 follower 角色^[34]。其中 leader 节点有且仅有一个，其余均为 follower 节点。leader 节点负责发起投票，并根据 followers 的投票结果进行决议以及集群状态的更新。follower 节点负责响应客户端请求、向客户端返回结果以及对事务进行投票。

zookeeper 为 kafka 集群维持了一个树型结构的集群状态信息，该树型结构被称为命名空间，类似于一个标准文件系统，如图 4-4 所示。zookeeper 命名空间的根路径为“/”，根到树叶的路径上每个节点称为 znode，每个 znode 都存储着一份协调数据，这些数据维护着 kafka 集群状态、配置、topic 等信息，znode 维护的数据主要为了分布式集群的协调一致，数据非常小，因此 zookeeper 命名空间保存在内存中。zookeeper 通过共享该命名空间组织的 znodes 节点来实现分布式集群的相互协调。

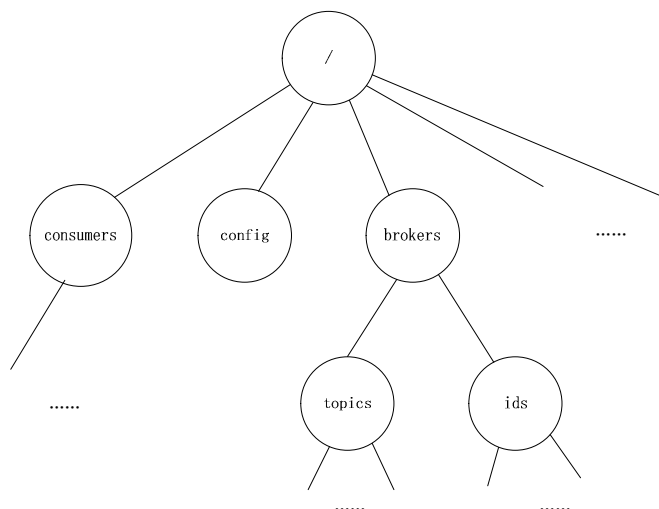


图 4-4 树型集群状态信息图

zookeeper 保证当集群状态信息发生更新时，kafka 集群各节点的信息能保持一致。当 kafka 集群某节点发出一个写请求，该请求会改变 zookeeper 命名空间中某一个 znode 的数据信息，比如通过命令行新增加一个 topic，zookeeper 节点对此请求的处理流程如图 4-5 所示。此时发出写请求的某 kafka 节点作为客户端向 zookeeper 中的 follower 节点发出写请求，follower 节点将该请求发送给 leader 节点，leader 节点收到该请求后，通过原子广播向所有的 follower 节点发起投票 (proposal)，follower 节点将投票结果回复给 leader 节点，leader 节点统计投票结果后，若超过一半的 follower 节点同意该操作，则开始写入该 topic，同时将写入操作通知 (commit) 给所有的 follower 节点，follower 节点将请求结果返回给客户端。

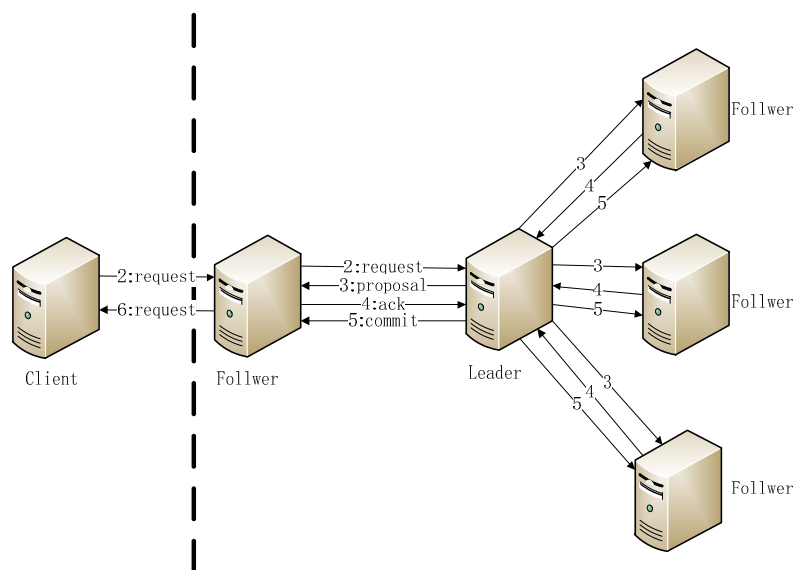


图 4-5 集群状态信息变更请求处理流程图

4.3.2 日志数据高可用的实现

日志数据进入 kafka 集群前需要标识该数据所属的 topic, topic 可以认为是该日志数据的类别。Kafka 集群中每一个 topic 都会有一个或者多个分区(partition), 如图 4-6 所示。每一个分区都是一个有序的且不可变的消息队列, 分区中的每个消息数据都被分配一个称为偏移(offset)的顺序 ID 号, 每个消息都被该顺序 ID 号唯一标识。

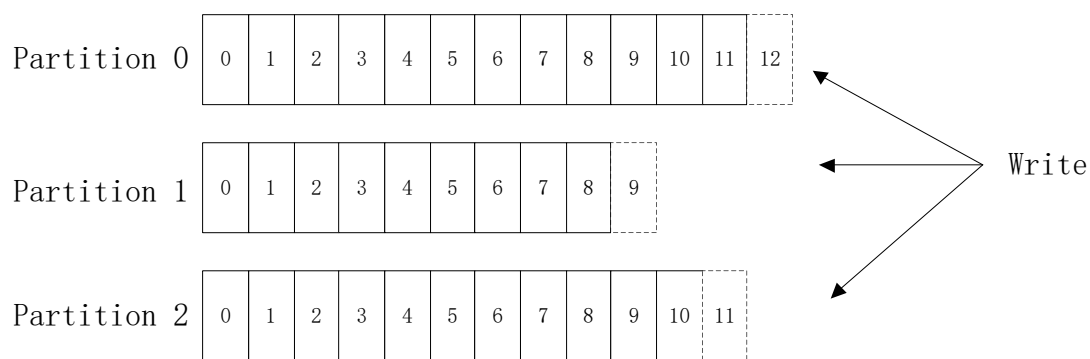


图 4-6 kafka 集群数据写入

本文为保证数据的高可用, 在写入过程中加入数据副本机制, Kafka 集群中的副本机制如图 4-7 所示, 即对于一个 topic 下的多个分区, 对每个分区进行拷贝并将副本保存在不同的 Broker 节点上, 对于每一个分区选取一个 Broker 节点作为该分区的主节点, 存有该分区副本的其余 Brokers 作为该分区的从节点, 并维持从节点分区副本与主节点分区的状态同步。当 kafka 集群中某节点发生故障, 以故障节点为主节点的分区会重新选择一个保存该分区副本的 Broker 节点作为主节点, 从而保证数据的高可用。

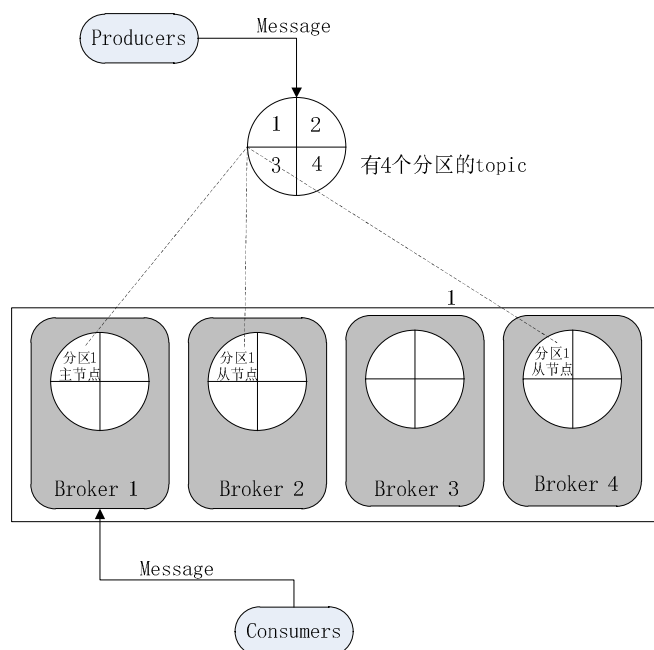


图 4-7 副本写入机制示意图

Kafka 集群分区的副本与主节点分区需要保持状态一致，本文在实现分区副本与主节点分区状态一致时采用同步复制模式。producer 往 kafka 集群写入消息时，首先通过与 zookeeper 通信识别出分区主节点，然后向分区主节点发送信息，主节点将消息写入本地后，向分区的所有从节点发送该消息，从节点收到消息后写入本地，并向主节点发送 ack 表示消息已经写入，主节点收到所有从节点的 ack 消息后，向 producer 回传 ack 消息。

4.3.3 kafka 集群的部署

本文使用三台主机搭建 kafka 集群，相应在这三个节点的 Zookeeper 配置文件如下所示。

```
# 心跳时间，以毫秒为单位
tickTime=2000

# Follower 节点与 Leader 节点之间初始连接时可以容忍的最大心跳数量
initLimit=10

# Follower 节点与 Leader 节点进行通信时可以容忍的的最大心跳数量
syncLimit=5

# Zookeeper 服务端的端口
clientPort=2181
```

```
# 数据存储路径
dataDir=/home/hqwu/elk/zookeeper-3.4.12/zkdata
# 日志文件存储路径
dataLogDir=/home/hqwu/elk/zookeeper-3.4.12/zkdataLog
# zookeeper 节点信息
server.1=192.168.234.12:2888:3888
server.2=192.168.234.13:2888:3888
server.3=192.168.234.15:2888:3888
```

zookeeper 集群中节点服务器的信息采用“server. N=Y:A:B”的格式来设置，其中 N 设置服务器编号，Y 设置服务器的 IP 地址，A 代表 Y 所设置的服务器与集群 leader 节点的通信端口，B 设置选举端口，表示 zookeeper 集群处于恢复模式选举新 leader 时，服务器与服务器之间相互通信的端口。zookeeper 配置文件配置完毕后，还需要在集群中每个节点的 dataDir 目录下建立名为 myid 的空文件，并将节点的服务器编号写入该文件中。

Kafka 集群部署三个 Broker 节点，每个节点的 kafka 安装路径下的 conf 目录下存在一个 server.properties 文件，Kafka 集群的配置主要集中在该文件。本文的 Kafka 集群主机地址为 192.168.234.12 的 broker 节点的配置如下所示，其余节点的配置与此大致相同，只需相应修改其中 broker.id、host.name 选项值即可。

```
# broker 在集群中的唯一标识
broker.id=1
# broker 节点的服务端口
port=9092
# broker 节点的主机地址
host.name=192.168.234.12
# kafka 日志数据文件的存储路径
log.dirs=/home/hqwu/elk/kafka_log
#日志文件在 kafka 集群中的最长保存时间
log.retention.hours=1
# zookeeper 集群的地址
zookeeper.connect=192.168.234.12:2181,192.168.234.13:2181,192.168.234.15:2181
# 消息的默认备份数量
default.replication.factor=2
```

kafka 集群的三个 Broker 节点的 server. properties 配置文件按上图设置之后, 在每个 Broker 节点的 kafka 安装目录, 启动 Kafka 程序, 即可建立 Kafka 集群。

```
>bin/kafka-server-start.shconf/server.properties
```

4.4 本章小结

针对实现日志数据处理规则的灵活配置以及对日志数据的缓存队列的设计目标, 日志缓存服务子系统设计实现了数据处理模块和缓存模块。数据处理模块采用 logstash 将数据处理过程分成输入阶段、处理阶段和输出阶段, 本章重点阐述了 logstash 在处理阶段通过丰富的插件来实现对设备日志的处理过程, 由处理过程可以得出对于不同类型设备的日志只需要编写极少行数的处理规则即可实现相应设备日志数据的处理, 无需日志产生端添加额外的字段, 实现了数据处理规则的灵活配置, 且 logstash 运行的过程中若发生数据处理规则修改, 数据处理模块能够重新加载日志数据处理逻辑而无需重启模块; 缓存模块采用 kafka 集群实现日志数据的缓存队列, 采用 zookeeper 保证了集群节点状态的协调一致, 并通过数据副本机制保证了 kafka 集群中数据的高可用性。

第五章 日志计算中心的设计与实现

日志检索层的 logstash 拉取日志缓存服务子系统的日志数据，并写入到日志计算中心，日志计算中心是日志检索层的核心子系统，该子系统的设计目标是实现日志数据的实时检索以及提供一个可视化的日志检索界面。本文设计将日志计算中心分为日志检索模块和展示模块。本章主要阐述日志检索模块和展示模块的设计与实现过程。

5.1 日志计算中心的架构设计

日志计算中心负责实现日志数据的实时检索分析，并提供可视化的数据检索界面。传统的工业日志系统通常对工业日志数据进行离线式数据分析，无法解决工业生产中遇到的即时性问题，本文采用 Elasticsearch 实现日志数据的实时检索，Elasticsearch 基于著名的全文搜索引擎 Apache Lucene 实现的搜索引擎^[35, 36]，使用倒排索引建立存储结构，检索性能可以达到百毫秒级，并保证了数据的高可用。日志计算中心的架构图如下所示。

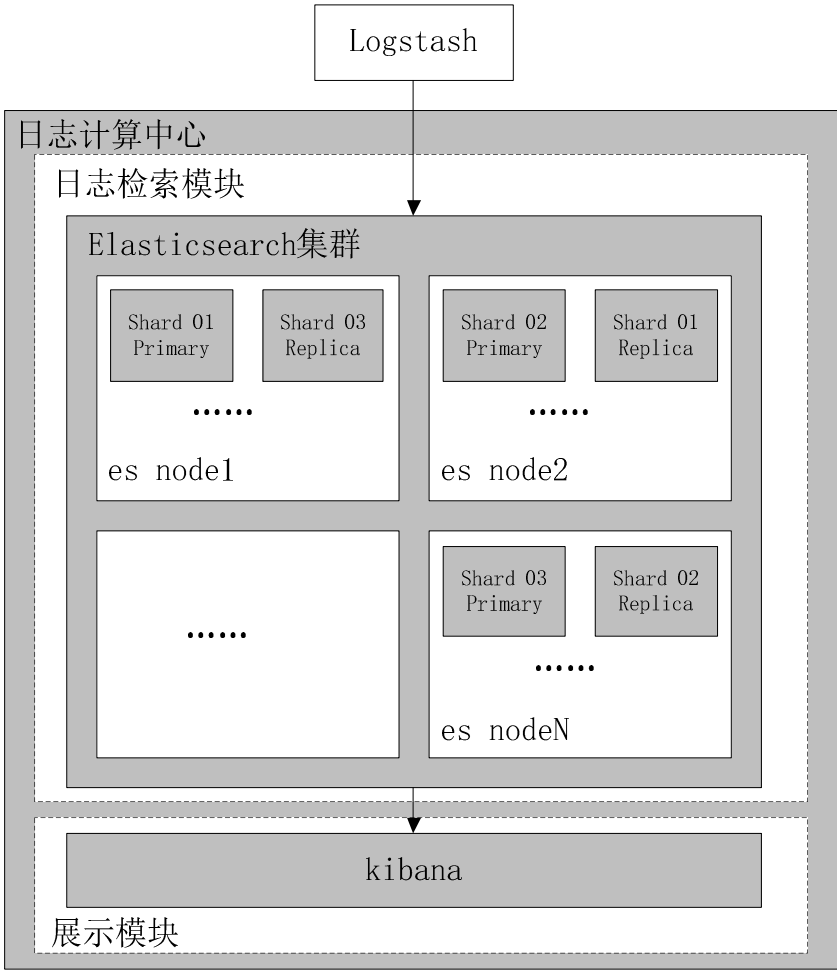


图 5-1 日志计算中心架构图

日志计算中心分为两个模块：日志检索模块和展示模块。日志检索模块负责实现日

志数据的实时检索功能，展示模块负责提供可视化的日志检索界面。日志检索层的 logstash 通过拉取日志缓冲层中 kafka 集群的日志数据，并写入到日志检索模块。日志检索模块部署 Elasticsearch 集群，对日志数据进行实时数据分析，并对日志数据通过副本机制保存在不同的集群节点上，保证数据的高可用性。展示模块使用开源软件 kibana，通过与集群通信获取集群中分片的状态等数据，并提供可视化的数据检索界面。下面主要阐述日志检索模块与展示模块的设计与实现过程。

5.2 日志检索模块的实现

本节首先介绍 Elasticsearch 的概况，然后进行 Elasticsearch 集群的部署，并实现日志数据在 Elasticsearch 集群的高可用性，最后对集群进行性能调优，提升日志系统对日志数据的实时检索性能。

5.2.1 Elasticsearch 概述

Elasticsearch 是一个可扩展性高的、开源的全文搜索分析引擎，可以近乎实时地对海量数据进行存储、搜索和分析^[37]。它通常用作底层引擎/技术，为具有复杂搜索功能和要求的应用程序提供支持。Elasticsearch 使用 Java 语言开发，并将 Lucene 库作为其核心来实现所有索引和搜索的功能，并提供简单的 RESTful API 接口，从而隐藏了 Lucene 的复杂性^[38]。表 5-1 是对 Elasticsearch 集群相关专业术语的解释。

表 5-1 Elasticsearch 集群专业术语说明

术语	说明
文档	Elasticsearch 中的基本信息单元。文档以 JSON 格式来表示，每个 JSON 格式的文档包括了零个或多个字段。
索引	具有类似特征的文档集合。
文档类型	在索引中，使用者可以定义一个或者多个文档类型。文档类型是索引的逻辑类别，其语义由使用者来定义。通常情况下，人们为具有一组公共字段的文档归为同一个类型。
节点	一个服务器，节点作为集群的成员，负责提供数据的存储、索引和搜索功能。节点由名称标识，默认情况下节点启动时被分配一个随机的通用唯一标识符(UUID)，也可以通过配置定义节点的名称。
集群	一个或者多个节点的集合，集群由唯一的名称标识。
分片	Elasticsearch 将索引细分为多个单元，这里的单元称为分片，是一个单一的 Lucene 的实例。

分片、节点和集群在物理上构成了 Elasticsearch 集群，索引、文档类型以及文档是

Elasticsearch 集群中数据的逻辑组织形式。表 5-2 将 Elasticsearch 与人们更为熟悉的关系型数据库 MySQL 的概念上的对比。

表 5-2 Elasticsearch 与 MySQL 概念对比

Elasticsearch	MySQL
索引 (index)	数据库 (database)
文档类型 (type)	表 (table)
文档 (document)	行记录 (row)
字段 (field)	列 (column)

5.2.2 Elasticsearch 集群的部署

Elasticsearch 集群是一个 P2P 类型、去中心化的分布式系统,采用 gossip 协议^[39, 40]。除了与集群状态管理有关的请求以外,集群的节点在发送其他请求时先计算该请求需要转发给集群中的哪些节点,然后直接与这些节点进行通信。Elasticsearch 提供了两种集群节点自动发现策略,第一种是 multicast 方式,即组播方式,该方式通过保证节点配置文件中的 cluster.name 的值相同即可。第二种是 unicast 方式,即单播方式,该方式通过 Elasticsearch 集群中节点在配置文件中提供集群中其他节点的地址,来完成集群的发现。在实际场景中,路由交换设备不能保证一定支持且开启组播信息传输,且多网卡的节点发送的组播信息可能在操作系统内核层面被丢弃,因此本文设计实现的 Elasticsearch 集群采用单播方式来完成集群的发现。

Elasticsearch 集群的节点有三种类型: master 节点、data 节点以及 client 节点,这三种类型的 Elasticsearch 节点所担负的职责如表 5-3 所示。

表 5-3 Elasticsearch 节点及其职责

节点类型	职责
master 节点	主要用于元数据 (metadata) 的处理,比如索引的新增、删除、分片分配等
data 节点	负责数据相关操作,比如分片的 CRUD,以及搜索和整合操作。这些操作都比较消耗 CPU、内存和 I/O 资源
client 节点	主要起到路由请求的作用,实际上可以认为是负载均衡器

本文使用三台主机部署 Elasticsearch 集群,三台主机的主机信息如表 5-4 所示。

表 5-4 Elasticsearch 集群节点信息

主机名	IP 地址	操作系统
node-1	192.168.234.14	CentOS6.5
node-2	192.168.234.16	CentOS6.5
node-3	192.168.234.18	CentOS6.5

实现 Elasticsearch 集群的部署，需要对各节点 Elasticsearch 安装目录下的配置文件 `elasticsearch.yml`。各节点启动 Elasticsearch 服务时会首先读取该配置文件，通过配置文件对 Elasticsearch 进行配置。对 `node-1` 节点的 `elasticsearch.yml` 文件配置如下。

```
# Elasticsearch 集群名字
cluster.name: log-system

# 节点名
node.name: node-1

# 数据文件路径
path.data: /home/hqw/es_data

# 日志文件路径
path.logs: /home/hqw/es_log

# 是否有资格被选举为 master 节点
node.master: true

# 是否存储索引数据
node.data: true

# 主机 IP 地址
network.host: 192.168.234.14

# 对外提供的 http 访问端口
http.port: 9200

# 集群中其他 master 节点的主机地址列表
discovery.zen.ping.unicast.hosts: ["192.168.234.16","192.168.234.17"]
```

在 `node-2` 和 `node-3` 节点的配置文件中，需要修改 `node.name`、`network.host` 以及 `discovery.zen.ping.unicast.hosts` 选项的值，其余选项值不变。

本文通过安装 `Elasticsearch-head` 插件用于可视化管理和操作 Elasticsearch 集群状态。当 Elasticsearch 集群部署且启动后，启动 `Elasticsearch-head`，通过浏览器访问主机 9100 端口能够查看到 Elasticsearch 集群的状态信息。其界面如图 5-2 所示。

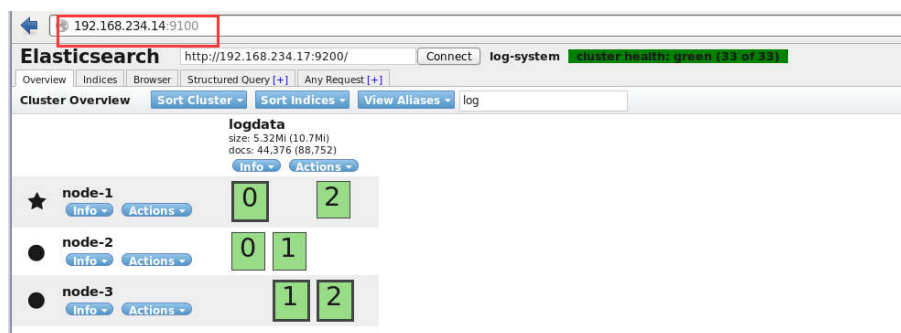


图 5-2 Elasticsearch-head 界面图

5.2.3 Elasticsearch 集群数据高可用的具体实现

本文为保证在 Elasticsearch 集群上数据的高可用性，为 Elasticsearch 集群中的数据提供了副本机制。写入到 Elasticsearch 集群的数据会被指定写入到 Elasticsearch 集群中的一个特定索引(index)中，每个索引都被细分为多个分片(shard)，数据保存在其中的一个分片中，分片通常分开保存在不同的集群节点。为保证某个 Elasticsearch 集群节点出现故障时，分片不可用导致数据丢失，本文对每个主分片均拷贝一份副本分片保存在与主分片不同的集群节点上，当主分片所在的集群节点发生故障时，副本分片所在的节点查看 master 节点发送的集群状态信息，检测该分片的主分片节点出现故障后，将副本分片提升为主分片，使得数据仍然能对外可用。在实现上，可以在创建索引时通过 curl 命令发送的实体内容中设置副本数，也可以通过 elasticsearch.yml 文件配置索引默认的副本分片个数。在 elasticsearch.yml 文件配置副本分片默认值如下所示。

```
index.number_of_shards: 3    //默认索引主分片个数为 3
index.number_of_replicas: 1 //默认副本分片数为 1
```

副本分片需要与主分片保持一致性，才能保证主分片节点发生故障时数据不会丢失。图 5-3 展示了集群的数据写入流程。图示中分片副本数为 2，客户端首先向 Node1 节点发送写入数据的请求，Node1 节点通过计算得出数据需要存储在编号为 0 的分片上，并通过集群状态信息发现分片 0 的主分片节点为 Node3 节点，于是 Node1 节点将客户端请求转发给 Node3 节点。Node3 节点接收到该请求后将数据存入到主分片 0，同时将数据转发到拥有主分片 0 副本的 Node1 和 Node2 上。默认情况下，对于有 N 个副本数的索引，当收到 $(N/2+1)$ 个副本分片节点写入成功，主分片节点既可以认为数据写入成功。因此，在图示的集群中，Node3 节点收到其中一个节点返回数据写入成功时，Node3 节点会通知给向其转发客户端请求的 Node1 节点该请求数据写入成功，Node1 节点接收到后向客户端返回写入成功。

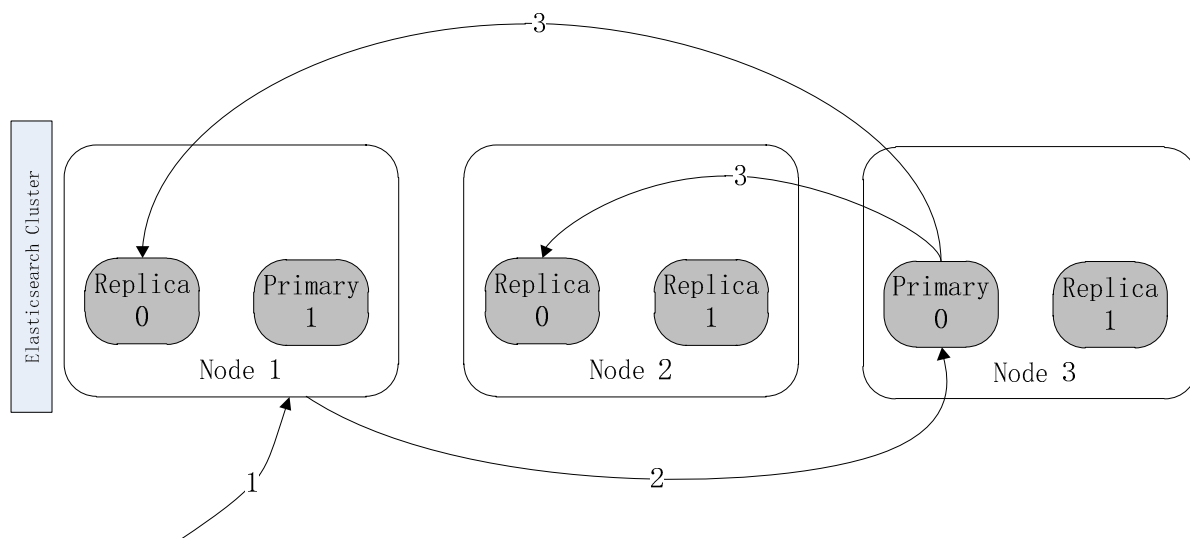


图 5-3 Elasticsearch 集群数据写入流程

5.2.4 Elasticsearch 性能调优

本文的分布式工业日志系统要实现日志数据的近实时检索，主要通过检索模块的 Elasticsearch 集群对日志数据的实时更新处理，因此对 Elasticsearch 集群进行性能调优，能够显著提升日志系统对日志数据的检索性能。

(1)内存调优

Elasticsearch 基于 lucene 构建，而 lucene 出色的设计正是在于它能高效的利用系统内存来缓存索引数据。lucene 的索引文件 segments 需要占用系统内存，以提供数据的快速访问。同时，Elasticsearch 本质上是 java 程序，JAVA 程序的运行需要建立在 JVM(Java Virtual Machine, Java 虚拟机)上，JVM 也需要占用系统内存。在内存调优上，本文设置为 JVM 和 lucene 各占节点 50%的内存。

在实际节点上，只需要在 Elasticsearch 安装目录下的 jvm.options 文件设置 JVM 所占系统内存即可。本文测试搭建的 Elasticsearch 集群节点内存为 4G，因此将 JVM 大小设置为 2G。

```
-Xms2g
-Xmx2g
```

(2)刷新闻隔优化

Elasticsearch 使用倒排索引结构实现了快速查找，但要做到数据实时的动态更新，Elasticsearch 还使用了一个策略：将新收到的数据写到新的索引文件。

Elasticsearch 把生成的倒排索引称为一个段(segment)，并另外使用一个 commit 文

件，来记录所有的 segment。实现数据的近实时分析需要避免磁盘的频繁写入，Elasticsearch 对新收到的文档写入到新的 segment 文件，并采用文件系统缓存来缓存新的 segment，并在之后一定条件下进行全量持久化到磁盘。但文件系统缓存在断电或者程序退出时可能会导致数据丢失，Elasticsearch 采用事务日志记录每一次在 Elasticsearch 上的操作，当发生异常时，文件系统缓存的数据丢失，Elasticsearch 会从 commit 位置开始，恢复整个事务日志文件中的记录，保证索引数据的一致性。

如图 5-4 为 Elasticsearch 实现数据动态更新的过程，初始索引的 segment 均可用，并都已经持久化到磁盘，随后，新接收的文档进入内存缓冲，与此同时该文档被追加到事务日志，内存缓冲生成一个新的 segment，并刷新(refresh)到文件系统缓存中，此时内存缓冲被清空，事务日志不会清空，会继续累积文档，每隔一段时间或者事务日志的文件大小超过一定大小，会进行一次 flush 操作，此时文件系统缓存真正同步到磁盘上，commit 文件更新，事务日志文件清空。

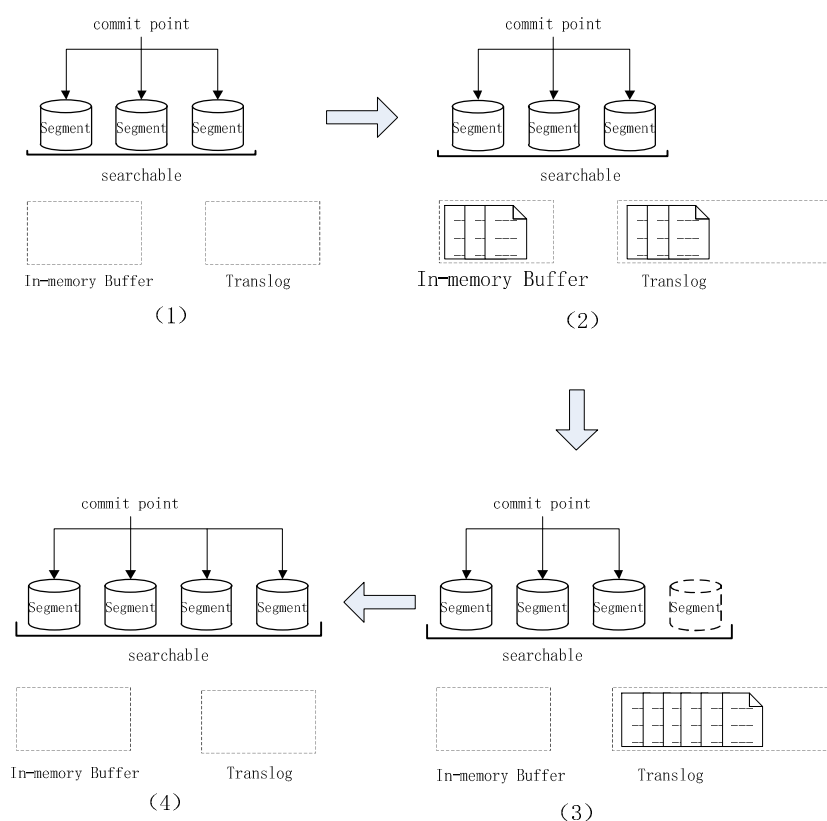


图 5-4 Elasticsearch 实现数据动态更新原理图

从系统内存刷新到文件系统缓存的时间间隔称为 `refresh_interval`，Elasticsearch 默认该时间间隔为 1s，这对于实时性要求不高的应用来说相当于实时可搜索，但同时意味着每一秒都会产生一个新文件，而打开一个新文件会消耗节点的文件句柄、内存以及 CPU

等资源，这给服务器带来一定的负载压力。考虑到 A 公司需求的日志系统对于日志数据的更新速度并不需要 1s 这个级别的实时性，而且在高并发场景下，日志系统需要的是更快的写入性能。因此，本文通过 `elasticsearch.yml` 文件，将 `refresh_interval` 值设置为 30s，

```
index.refresh_interval: 30s
```

同时，Elasticsearch 的实时动态更新过程需要不断地打开新文件，而在常见的 linux 服务器中进程默认打开的最大文件句柄数为 1000，这对于 Elasticsearch 来说是远远不够的。本文在每个 Elasticsearch 节点设置最大文件句柄数为 65535，通过修改 `/etc/security/limit.conf` 文件并重启使配置生效。

```
* softnfile 65535
* hardnfile 131072
```

(3) 内存锁定

操作系统有一个虚拟内存页交换机制，该机制会通过算法决策该将哪一个内存页交换到磁盘文件中，而内存换出会极大地影响 Elasticsearch 的查询性能。因此，本文设置在 Elasticsearch 运行过程中禁止内存交换，通过在 Elasticsearch 的配置文件 `elasticsearch.yml` 中添加：

```
bootstrap.memory_lock: true
```

同时，通过在 Elasticsearch 节点的 `/etc/sysctl.conf` 文件设置一个进程能拥有的最大的虚拟内存区域，并执行“`sysctl -p`”命令生效：

```
vm.max_map_count=262144
```

(4) 段合并优化

在刷新闻隔优化中，本文设置 `refresh_interval` 为 30s，这意味着每经过 30s 事件，Elasticsearch 会产生一个新的 `segment` 文件，`segment` 文件数量不断增加，每次请求都需要扫描一遍所有的 `segment` 文件，响应速度随着文件数量增多而不断变慢。因此，Elasticsearch 在后台会主动将零散的 `segment` 文件进行段合并，使得索引内只保留少量的 `segment` 大文件，段合并的进行由独立的线程进行，并不影响新 `segment` 文件的产生。

在段合并中，Elasticsearch 首先读取 `segment`，进行归并计算，并将几个较小的 `segment` 合并到较大的 `segment` 中；当合并完成，其中较大的 `segment` 刷新到磁盘中，被合并的较小的 `segment` 被删除，从而减少了 `segment` 索引文件的数量。

因此，本文通过修改 `elasticsearch.yml` 文件，设置与段合并相关的参数值。

```
index.merge.policy.floor_segment: 2mb
index.merge.policy.max_merge_at_once: 10
index.merge.policy.max_merged_segment: 5gb
```

以上参数的说明如表 5-5 所示，其中省略了 index.merge.policy 前缀。

表 5-5 段合并相关参数值说明

参数	说明
floor_segment	设置一个 segment 大小，小于该值的 segment 优先被合并
max_merge_at_once	段合并合并一次最多的 segmeng 个数
max_merged_segment	设置一个 segment 大小，大于该值的 segemnt 不参与合并。

此外，在本文的日志系统，会存在 segment 文件超过 max_merged_segment 设置的大小，因此这些文件永远不会被合并，这会极大地浪费文件句柄、系统内存等系统资源。本文通过 optimize 接口，选择在负载较低的时间段，如凌晨时间段，强制合并这些大的 segment 文件。

```
>POST http://192.168.234.14:9200/indexname/_optimize?max_num_segements=1
```

5.3 展示模块的实现

日志检索模块实现了对日志数据的实时分析，但技术人员仍需要通过在终端使用命令来获取设备日志信息，因此本文通过展示模块将日志检索模块对日志数据实时分析的结果以可视化的形式展示出来。展示模块采用 kibana 实现，kibana 是一个开源软件，kibana 原名为 elasticsearch-dashboard，其在初始设计便是为了支持 Elasticsearch 数据的可视化展示^[41]。Elasticsearch 集群的数据接入 kibana 中，用户通过在界面设置检索过滤条件即可获取需要的设备日志数据。

kibana 通过访问 Elasticsearch 集群获取并展示集群中的设备日志数据。在 kibana 官网下载安装好 kibana 后，通过对 kibana.yml 文件的配置集群主节点的主机地址和其端口号，即可通过 kibana 实现 Elasticsearch 集群数据的可视化展示。如图 5-5 是在 kibana 在浏览器提供的可视化界面中设定索引匹配模式，匹配 Elasticsearch 集群中的 systemfunctiontest 索引。如图 5-6 在 discover 标签上输入过滤条件来检索该索引的设备日志数据。

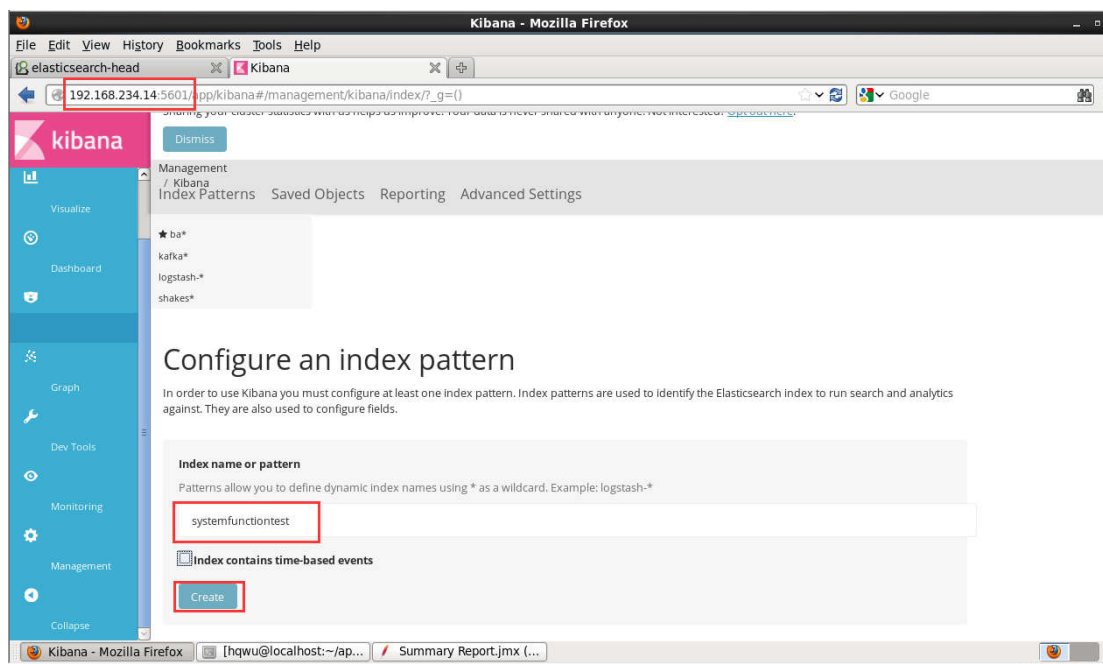


图 5-5 kibana 设定索引匹配模式

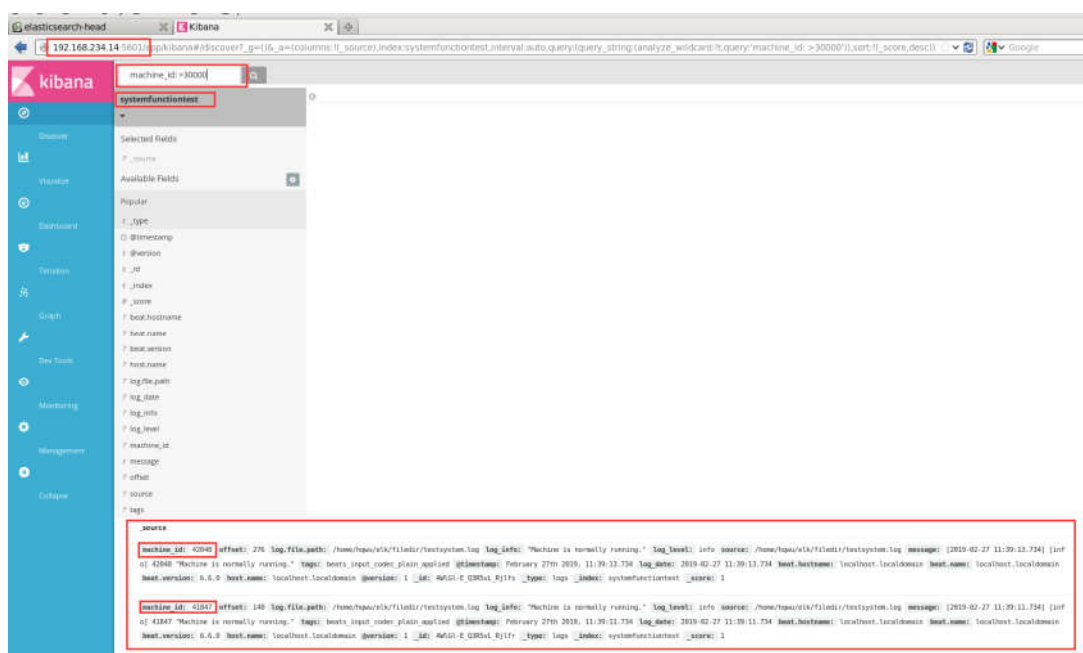


图 5-6 kibana 检索索引日志数据

5.4 本章小结

针对日志数据的实时检索以及可视化的日志检索界面的设计目标，日志计算中心设计实现了日志检索模块和展示模块。日志检索模块基于 Elasticsearch 集群，部署了 Elasticsearch 集群，通过数据副本机制实现 Elasticsearch 集群数据的高可用性，并对集群进行了性能调优；展示模块采用 kibana 实现日志检索模块中数据的可视化展示。

第六章 系统测试与结果分析

本章对基于 ELK 和 kafka 的分布式工业日志收集系统进行系统测试，首先对日志系统进行基本的功能测试，验证日志系统能够实现日志数据收集、数据处理规则可配置以及检索功能。随后，本章对日志系统的收集层、缓冲层以及检索层分别进行性能测试，并对测试结果进行分析，验证日志系统各层已经达到设计目标，即收集层高并发处理日志请求，减少磁盘 I/O 对服务器性能的影响；缓冲层实现日志数据的缓冲队列，并保证了日志数据的高可用性；检索层实现了日志数据的实时检索功能，且保证了数据的高可用性。

6.1 测试内容

本文设计的分布式工业日志系统由日志收集层、日志缓冲层以及日志检索层组成，本章首先对日志系统进行功能测试，然后对实现系统各层的核心子系统进行性能测试，以下是本章的测试内容：

1) 基本功能测试：验证日志系统是否实现日志数据的收集、可配置处理以及实时检索功能。

2) 日志收集层性能测试：分为网络模块与业务模块的性能测试，网络模块性能测试用于验证网络模块能够高并发高吞吐地处理日志存储请求，业务模块性能测试用于验证业务模块能够极大减少磁盘 I/O 的次数，从而减少磁盘 I/O 对服务器性能的影响。

3) 日志缓冲层性能测试：分为读写性能测试和数据高可用性测试，读写性能测试用于验证缓存模块具有高吞吐读写的能力，数据高可用性测试验证缓存模块能够保证日志数据的高可用。

4) 日志检索层性能测试：分为查询性能测试和数据高可用性测试，查询性能测试验证日志检索模块具有日志数据的实时检索能力，数据高可用性测试验证日志检索模块能够保证日志数据的高可用。

6.2 测试环境

6.2.1 测试环境部署

本文的分布式工业日志系统为分布式集群，需要多个服务节点。本文通过在 VMware 创建多个虚拟机节点来搭建整个分布式集群。测试环境中的虚拟机配置如表 6-1 所示。为简洁叙述，本章将 IP 为 192.168.234.11 的节点简称为节点 11，以此类推，本文测试

环境由节点 11~18 的机器组成。

表 6-1 测试环境节点配置图

编号	IP	CPU	内存	硬盘	说明
1	192.168.234.11	1 虚拟内核	4G	50GB	日志收集层节点
2	192.168.234.12	1 虚拟内核	4G	100GB	日志缓冲层节点
3	192.168.234.13	1 虚拟内核	4G	100GB	日志缓冲层节点
4	192.168.234.15	1 虚拟内核	4G	100GB	日志缓冲层节点
5	192.168.234.14	1 虚拟内核	4G	100GB	日志检索层节点
6	192.168.234.16	1 虚拟内核	4G	100GB	日志检索层节点
7	192.168.234.17	1 虚拟内核	4G	100GB	日志检索层节点
8	192.168.234.18	1 虚拟内核	4G	50GB	测试节点

虚拟机系统均为 Centos 6.5 系统, Java 版本均为 1.8.0_201, gcc/g++ 版本为 5.4.0, Filebeat 版本为 6.6.0, logstash 版本为 5.3.0, zookeeper 版本为 3.4.12, kafka 版本为 0.10.2, elasticsearch 版本为 5.3.0, kibana 版本为 5.3.0。

6.2.2 测试工具

本章测试使用的测试工具如下表所示, 表 6-2 展示了工具在测试过程中终端使用时需要使用的选项及其选项说明。

表 6-2 测试工具

工具名	说明	选项	选项说明
kafka-producer-perf-test.sh	生产者写入 kafka 集群性能测试	--topic	消息写入的 topic
		--throughput	每秒写入消息数
		--num-records	写入的消息总数
		--record-size	单个消息字节数
		--producer-props	生产者的配置属性
kafka-consumer-perf-test.sh	用于 kafka 集群读取性能测试	--topic	消息所属的 topic
		--messages	消费的消息总数
		--zookeeper	zookeeper 列表
		--fetch-size	单个消息字节数
		--threads	线程数
ab	压测工具	-n	请求总数
		-c	单次的请求数
		-p	指定 POST 数据文件
		-q	不显示进度计数

6.3 基本功能测试

本节进行日志系统的功能测试, 功能测试是测试该分布式系统性能测试的前提, 只

有验证了系统能够正常运行和处理设备日志数据，之后的性能测试才有测试意义。日志收集层接收来自工业设备端传输的设备日志数据，并将日志数据上报到日志缓冲层，日志缓冲层的数据处理模块对日志数据进行处理后，写入到 `kafka` 集群，日志检索层拉取缓冲层的数据，并将数据交由 `elasticsearch` 集群进行日志的实时检索分析。若日志收集层、日志缓冲层以及日志检索层正常运行，则能够在检索层的展示模块正常检索和显示。本节通过是否能在展示层检索并展示相关数据，来验证日志系统的基本功能是否实现。

(1) 测试方法

本文通过在测试节点上模拟工业设备，向日志收集层节点发送日志数据，测试数据在日志收集层被接收后，通过 `filebeat` 传入日志缓冲层节点，在写入 `kafka` 集群前由缓冲层节点的数据处理模块 `logstash` 进行数据处理，切分字段后并写入 `topic` 为 `systemFunctionTest` 的 `kafka` 消息队列缓冲。日志检索层的 `logstash` 消费该日志数据，并将日志数据写入到 `elasticsearch` 集群中名为 `systemfunctiontest` 的索引进行实时检索，最后由展示模块 `kibana` 以界面化的形式展示数据。本文通过在展示层是否能检索测试节点发送的日志数据，来验证系统的功能是否正常。

(2) 测试过程及结果分析

测试节点发送的设备日志测试数据如下：

```
[2019-02-27 11:39:09.733] [info] 11892 "Machine is normally running."
[2019-02-27 11:39:10.734] [info] 19057 "Machine is normally running."
[2019-02-27 11:39:11.734] [info] 41847 "Machine is normally running."
[2019-02-27 11:39:12.734] [critical] 27032 "Machine Malfunction"
[2019-02-27 11:39:13.734] [info] 42048 "Machine is normally running."
```

测试数据匹配 “`[${log_date}] [${log_level}] ${machine_id} ${log_info}`” 格式。

日志层接收测试节点传输的测试数据，并上报到日志缓冲层。由于本节的功能测试目的在于验证日志系统的基本功能是否实现，所以日志缓冲层的数据处理模块只使用了 `grok` 对测试数据进行了切分字段，并设置写入 `kafka` 集群中的数据格式为 `json` 格式。配置文件如图 6-1 所示。日志检索层的 `logstash` 消费 `kafka` 集群中 `systemFunctionTest` 主题的日志数据，并写入到 `elasticsearch` 集群。

```

1 input {
2   beats {
3     port => 5044
4   }
5 }
6 filter{
7   grok{
8     match => { "message" => "\[%{TIMESTAMP_ISO8601:log_date}\] \[%{WORD:log_level}\] %{NUMBER:machine_id} %{QUOTEDSTRING:log_info}"
9   }
10 }
11 output {
12   kafka {
13     bootstrap_servers => "192.168.234.12:9092,192.168.234.13:9092,192.168.234.15:9092"
14     topic_id => "systemFunctionTest"
15     codec => "json"
16   }
17 }

```

图 6-1 测试环境下日志数据处理规则

启动日志收集层、日志缓冲层以及日志检索层的模块服务，当测试环境下分布式日志系统的所需模块都已经能够提供对外服务时，测试节点向日志收集层节点发送设备数据。测试数据发送完毕后，在节点 14 使用浏览器打开 <http://192.168.234.14:5601>，kibana 在 5601 端口提供了一个可视化界面。如图 6-2 所示，首先在 kibana 中建立索引匹配模式，然后在检索界面选择该匹配模式，在检索该索引时输入“machine_id: >30000”以匹配设备日志数据中 machine_id 字段大于 30000 的日志数据，检索结果如图 6-3 所示。

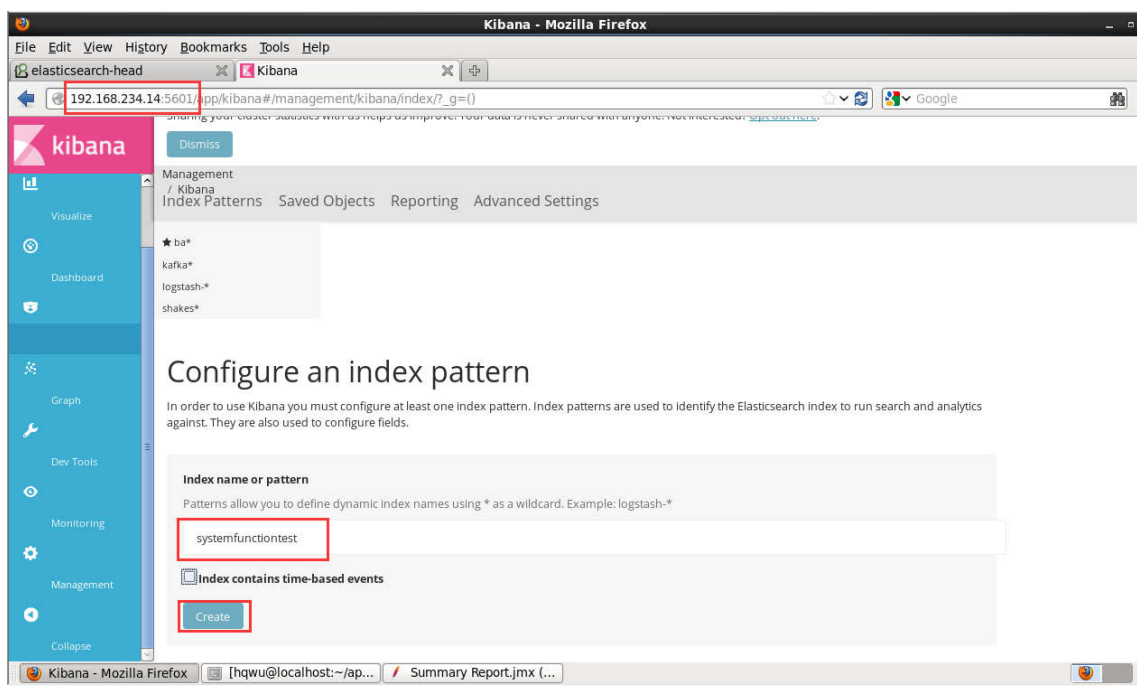


图 6-2 建立索引匹配模式

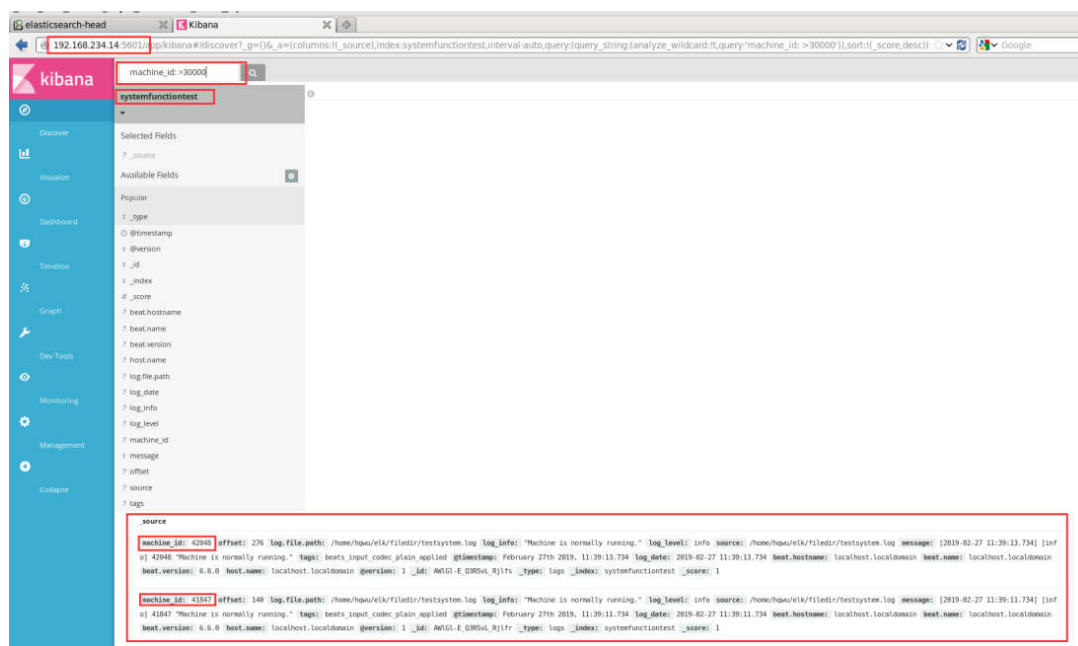


图 6-3 检索设备日志消息

展示模块查询得到测试数据，与实际测试数据中字段 `machine_id` 值大于 30000 的数据符合，验证了本文设计实现的分布式系统节点之间通信正常，通过配置日志数据的处理规则，数据能够被正确处理，展示模块通过输入查询条件能够检索出所需要的测试数据，基本功能测试通过。

6.4 日志收集层性能测试

6.4.1 网络模块性能测试

(1) 测试方法

本小节主要通过测试在不同的并发请求数下网络模块的吞吐量，测试网络模块是否拥有高并发处理外部请求的能力。测试网络模块的吞吐量采用 `ping pong` 协议，即客户端与服务端均实现对数据的回显处理逻辑，客户端向服务端发送的数据，服务端接收后将数据返回给客户端，接着客户端将服务端返回的数据重新发送给服务端，数据在服务端和客户端之间如乒乓球一样来回接收发送，直到有一方断开与对端的连接。通过该过程中相互传输的数据总量与该过程持续的时间，可以得出网络模块的吞吐量。

(2) 测试过程及结果分析

节点 11 的网络模块开启服务，监听端口 9002。网络模块中的 `worker` 线程数设置为 1 个，即采用单线程来处理接收的数据，处理方法是数据重新发送给客户端。考虑到网络带宽可能是影响吞吐量大小的一个潜在因素，因此在日志收集层节点启动测试脚本，即客户端与提供服务的网络模块在同一节点。测试脚本向日志收集层节点的监听端口发

起 TCP 连接。

本小节中采用不同的消息长度和不同的并发连接数来测试网络模块的吞吐量，表 6-3 是在不同消息长度和不同的并发数下测试的吞吐量数据。为了更直观的反映吞吐量与并发连接数、消息大小的关系，绘制了吞吐量折线图，如图 6-4。

表 6-3 不同消息长度和不同并发数的吞吐量测试结果

并发连接数	1	10	100	1000	10000
4KB 消息吞吐率 (MB/S)	29.9	88.2	87.8	78.9	77.1
8KB 消息吞吐率 (MB/S)	48.6	138.5	137.8	130.3	128.6
16KB 消息吞吐率 (MB/S)	84.7	196.5	182.5	169.4	170.1

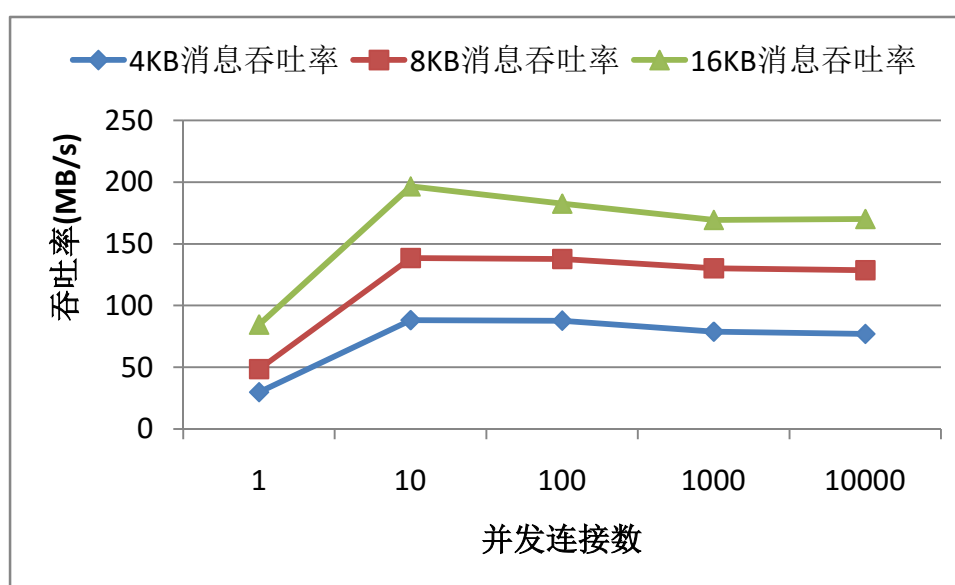


图 6-4 吞吐量折线图

由图 6-4 能较明显的看出，在并发连接数相同的情况下，随着消息大小的增大，吞吐率也随着不断增大。当工业设备通信端与网络模块单次传输数据在 16KB 时，网络模块的吞吐率最高。因此后续可以考虑优化工业设备通信端的数据处理逻辑，可以考虑在不影响实时性的情况下，将日志数据累积到 16KB 再发送到网络模块。

由图 6-4 可以看出，在开始时随着并发连接数的不断增大，网络模块吞吐率的增长较为明显，并发连接数超过 10 之后，吞吐率在不同消息大小的情况都基本趋于平稳。在消息大小为 16KB、并发量为 10000 的情况下，网络模块吞吐率能维持在 170.1MB/s，且吞吐率保持平稳的上下浮动状态，若每条设备日志记录为 100 字节，那么网络模块每秒钟能够处理 170 万条设备日志记录，通常情况下，工业设备端产生日志的频率远远没有到达这个数量级，这说明网络模块能够为日志系统提供较为良好的高并发处理能力。

6.4.2 业务模块性能测试

(1) 测试方法

业务模块在高并发场景下通过设计实现 ringBuffer 环形缓冲来降低磁盘 I/O 对网络模块的性能影响。本文通过对比业务模块写入日志 (以下简称 ringLog) 与传统同步日志 (以下简称 syncLog), 来验证业务模块能够优化磁盘读写, 减少磁盘 I/O 对网络模块的性能影响。

(2) 测试过程及结果分析

为减少线程因素对于磁盘写入的影响, 本文对 syncLog 和 ringLog 均采用单线程来进行磁盘读写, syncLog 即单线程对于每一个写入请求都执行一次磁盘读写, ringLog 使用环形内存缓冲来暂存日志, 线程从环形内存缓冲拉取日志进行磁盘写入。测试写入的每条日志大小为 100 个字节, 测试写入 10k、100k、1000k 以及 10000k 日志记录时 ringLog 与 syncLog 所用的时间。图 6-5 是 ringLog 的测试代码。

```
7   pthread_t tid;
8   pthread_create(&tid, NULL, be_thdo, NULL);
9   for (int i = 0; i < 10000000; ++i)
10  {
11      LOG_ERROR("****The number of this device log is %d****", i);
12  }
13  pthread_join(tid, NULL);
14
```

图 6-5 ringLog 测试代码

表 6-4 记录了 ringLog 与 syncLog 在写入不同日志数下所用的时间。

表 6-4 不同日志数下 ringLog 与 syncLog 所用磁盘时间

写入日志数(千条)	10	100	1000	10000
ringLog 写入磁盘时间(s)	0.012	0.113	1.247	13.602
syncLog 写入磁盘时间(s)	0.047	0.511	4.977	50.131

将表格中的测试结果绘制成折线图, 以便更直观的进行 ringLog 与 syncLog 的性能比较。折线图如图 6-6 所示。

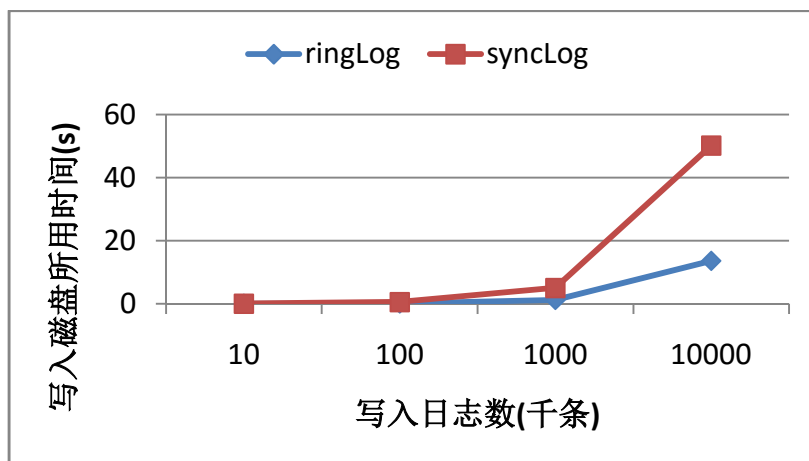


图 6-6 ringLog 与 syncLog 写入磁盘时间折线图

由折线图可以看出，随着写入日志数不断增加，ringLog 与 syncLog 在写入磁盘所花费的时间上的差距越来越明显。这主要是因为 syncLog 中每条日志记录对应一次磁盘写入，而 ringLog 通过 ringBuffer 将多次的日志数据一次写入磁盘中，这极大地减少了磁盘读写的次数。从表格中可以大致估算出，在相同的日志数下，ringLog 花费的时间只有 syncLog 的 30% 左右。因此，该测试能够验证业务模块极大的优化了磁盘读写的性能，减少了磁盘 I/O 的次数，降低了磁盘读写对网络 I/O 的性能影响。

6.5 日志缓冲层性能测试

6.5.1 读写性能测试

(1) 测试方法

本节使用 kafka 安装包中提供的测试脚本 kafka-producer-perf-test.sh 和 kafka-consumer-perf-test.sh 对 kafka 集群进行读写测试，通过测试工具提供的命令选项来测试 kafka 集群在不同情况下的读写性能。

(2) 测试过程及结果分析

1) 写入测试

如图 6-7 所示，本文创建用于测试的 test01 主题，该主题有 3 个分区，每个分区有三个副本。

```
[hqwu@localhost kafka_2.11-0.10.2.2]$ bin/kafka-topics.sh --create --zookeeper 192.168.234.13:2181,192.168.234.12:2181,192.168.234.15:2181 --replication-factor 3 --partitions 3 --topic test01
Created topic "test01".
[hqwu@localhost kafka_2.11-0.10.2.2]$ bin/kafka-topics.sh --describe --zookeeper 192.168.234.13:2181 --topic test01
Topic: test01 PartitionCount: 3 ReplicationFactor: 3 Configs:
Topic: test01 Partition: 0 Leader: 3 Replicas: 3,1,2 Isr: 3,1,2
Topic: test01 Partition: 1 Leader: 1 Replicas: 1,2,3 Isr: 1,2,3
Topic: test01 Partition: 2 Leader: 2 Replicas: 2,3,1 Isr: 2,3,1
[hqwu@localhost kafka_2.11-0.10.2.2]$
```

图 6-7 kafka 集群创建 test01 主题

如下所示是终端写入到 kafka 集群消息的命令，表示每秒对 kafka 集群的 test01 主

题写入 2000 个消息，其中每个消息大小为 1000 字节，总共发送 100000 个消息。表 6-5 记录了测试环境下的 kafka 集群在不同消息写入速度下的集群吞吐量和平均延迟，其中消息大小固定为 1000 字节。

```
>bin/kafka-producer-perf-test.sh --topic test01 --num-records 100000 --throughput 2000
--record-size 1000 --producer-propsbootstrap.servers=192.168.234.12:9092,192.168.234.1
3:9092,192.168.234.15:9092
```

表 6-5 不同消息写入速度下 kafka 集群吞吐量和平均延迟

消息总数	设置每秒消息数 (千条/s)	实际每秒消息数	吞吐率 (MB/sec)	平均延迟 (ms)
100000	1.0	999.7800	0.95	12.23
100000	2.0	1997.6827	1.91	39.11
100000	2.5	2498.6257	2.38	62.58
100000	3.0	2998.4108	2.86	108.00
100000	3.5	3231.6442	3.08	274.37
100000	4.0	3810.9756	3.63	139.78
100000	4.5	4435.5733	4.23	392.03
100000	5.0	4407.6163	4.20	646.82
100000	5.5	2575.3947	2.46	1967.22

将该表的数据绘制成如图 6-8 所示的折线图，以更直观地反映吞吐率和平均延迟与每秒消息数的关系。

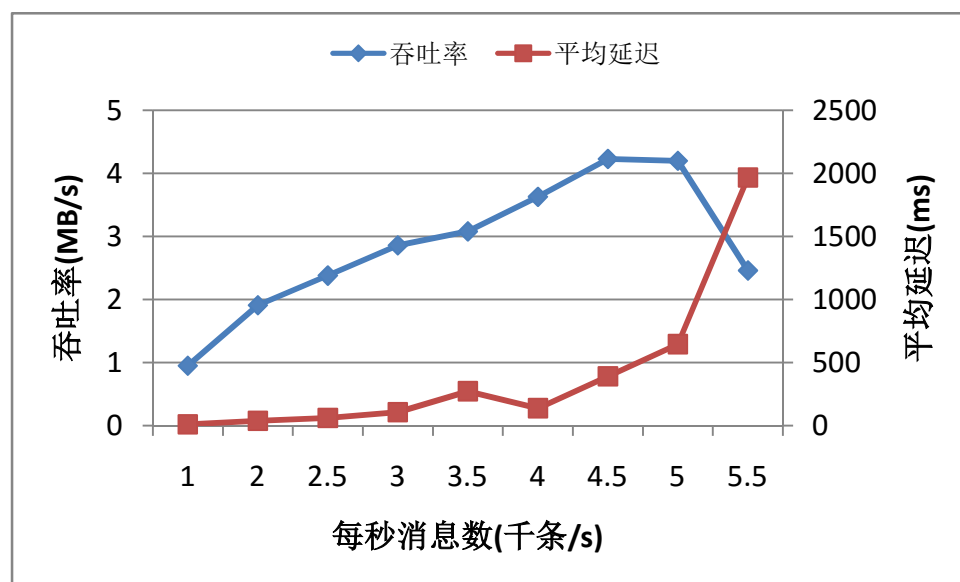


图 6-8 吞吐率和平均延迟折线图

由图 6-8 可以看出，每秒消息数在 4500 条/s 之前，随着每秒消息数的不断增加，吞吐率也在近乎线性增长，说明此时测试环境中 kafka 集群节点的资源正在不断地被充分利用，且此时平均延迟均在 1s 以内；当每秒消息数超过 5000 条/s 时，受限于测试环

境的硬件配置,平均延迟急剧上升,单位时间内处理的写入请求减少,因此吞吐率降低。测试环境中节点均为 4G 的 kafka 集群在保证平均延迟在百毫秒级的情况下,能达到 4.5MB/s 的写入速度,而实际生产环境中 kafka 机器节点的内存通常为 32G 以上,以单纯的倍数估算亦能达到 36MB/s,若单条日志记录为 100 字节, kafka 集群在实际生产中每秒写入的消息数能达到 36 万条/s,并且实际生产节点在 CPU 核数、固态硬盘等配置还存在提升空间,因此能够验证缓存模块的写入性能能够满足实际生产中工业日志写入的需求。

2) 读取测试

如下所示是在终端读取 kafka 集群中消息的命令,表示总共在 kafka 集群的 test01 主题上消费 100 万个消息,其中每次消费的数据为 1048576 个字节,即 1MB。表 6-6 记录了 kafka 集群读取不同消息总数时的消费速度和消息数速度。

```
>bin/kafka-consumer-perf-test.sh --zookeeper 192.168.234.12:2181 --topic test01 --fetch-size 1048576 --messages 1000000 --threads 1
```

表 6-6 kafka 集群消息消费速度

设置消费 总数(万条)	消费线程数	消费数据 (MB)	消费速度 (MB/s)	实际消费消 息数	消息数速度 (条/s)
10	1	95.7222	1.8546	100372	1944.6662
	10	95.7222	46.1757	100372	48418.7168
	100	95.7222	101.6159	100372	106552.0170
	1000	95.7222	96.8848	100372	101591.0931
100	1	953.6829	8.7842	1000009	9210.8866
	10	953.6829	15.9963	1000009	16773.3273
	100	954.1483	134.1483	1000497	140756.4716
	1000	954.1483	142.9436	1000497	149887.1910
1000	1	1076.4894	7.1301	1128781	7476.4977
	10	1076.4894	22.0796	1128781	23152.1075
	100	1059.1774	144.1450	1110628	151146.9788
	1000	1059.1774	127.5810	1110628	133778.3667

将表 6-6 中的数据绘制成折线图,图 6-9 为消费速度与消费线程数的折线图,图 6-10 为消费消息数速度与消费线程数的折线图。

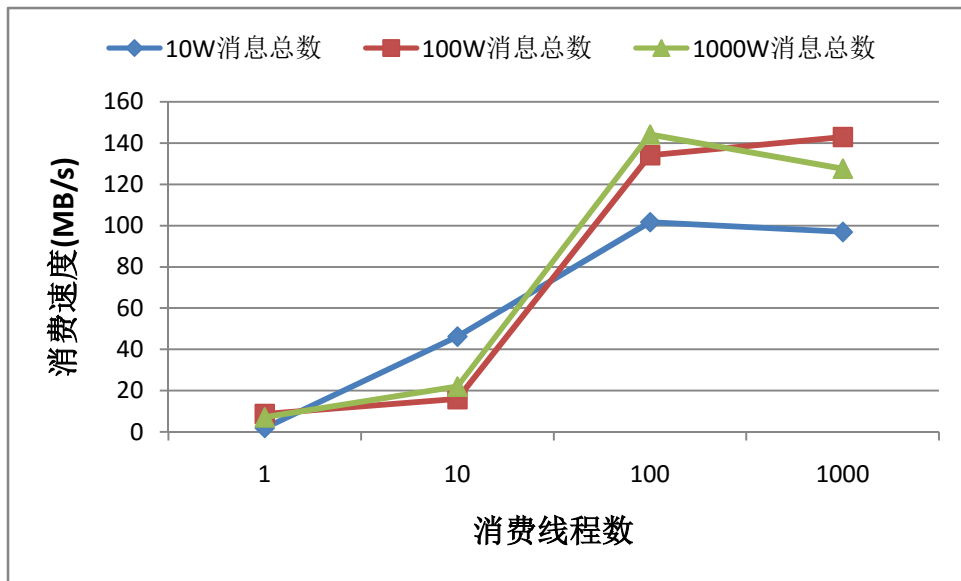


图 6-9 消费速度折线图

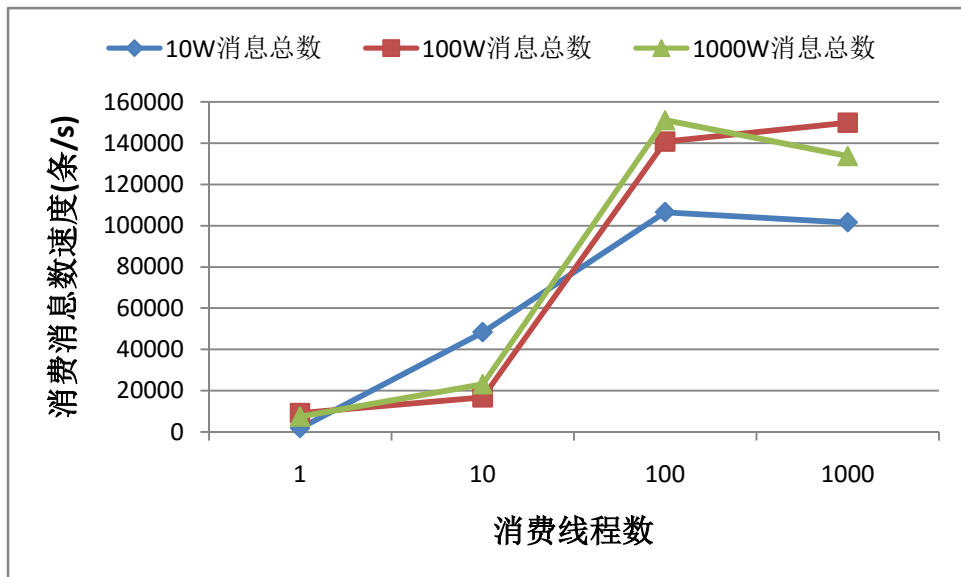


图 6-10 消费消息数速度折线图

由图 6-9 和图 6-10 可以清晰的看出,无论消费的消息总数是 10 万、100 万还是 1000 万,消费线程数为 100 个与 1000 个的消费能力相差不大。同时,总消费消息个数为 10 万时,线程数在 1 至 100 的范围内,消费能力呈线性增长,而总消费消息个数为 100 万和 1000 万时,并发消费线程数为 1 和 10 时消费能力较低,这是因为消费线程数太少并不足以完全发挥出 kafka 集群的性能。当消费日志消息的并发线程数达到 100 之后,消费的消息数速度达到 14 万条/s 左右,且测试环境中节点的机器配置相对配置较低,因此能够验证缓存模块被消费消息的性能能够满足实际生产需求。

6.5.2 高可用性测试

(1) 测试方法

本文通过模拟其中一个提供 Kafka 服务的节点发生故障，对比 Kafka 集群在正常运行和节点出现故障时，Kafka 对节点故障的感知与数据迁移，来验证 Kafka 集群的高可用性。

(2) 测试过程及结果分析

在节点 15 创建一个分区数为 3、副本数为 3 的 topic，名为 HighAvailabilityTest。图 6-11 是该索引在集群中的详细信息，由图 6-11 可以知道，该 topic 在集群中有三个分区：分区 1 的 leader 节点为节点 12，分区 2 的 leader 节点为节点 13，分区 3 的 leader 节点为节点 15。此时集群中三个节点正常工作，三个分区的副本同步序列 (Isr, In-Sync Replicas) 在三个节点都是可用的。

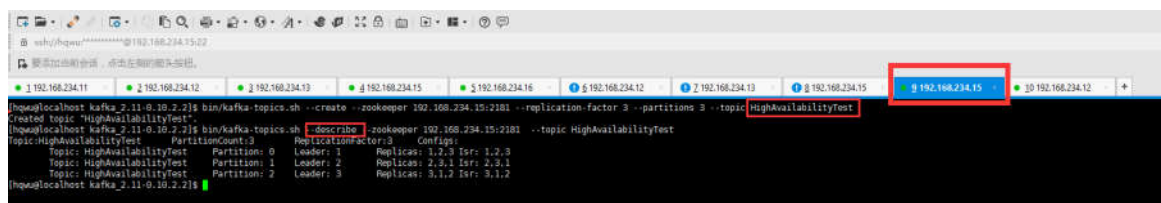


图 6-11 HighAvailabilityTest 索引详细信息

如图 6-12 所示，在 IP 为节点 12 上生产一个测试索引的消息，图 6-13 在节点 13 上消费该消息。说明 kafka 集群正常工作。

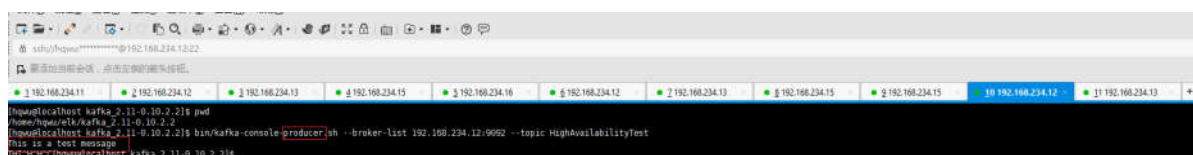


图 6-12 生产一个消息

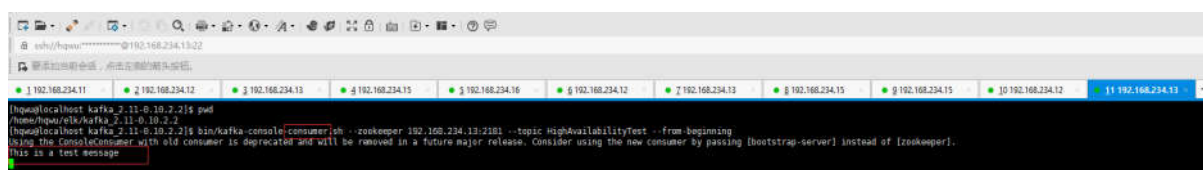


图 6-13 消费一个消息

数据的高可用性是指当某一节点发生故障，该节点的数据均不可用时，系统依然能对外提供正常的的数据服务，外界并不能感知到系统节点发生故障。本节通过使用 kill 命令结束节点上的 kafka 进程来模拟 kafka 节点出现故障，如图 6-14 所示，在 IP 为节点 12 上使用 kill 命令结束 kafka 进程。

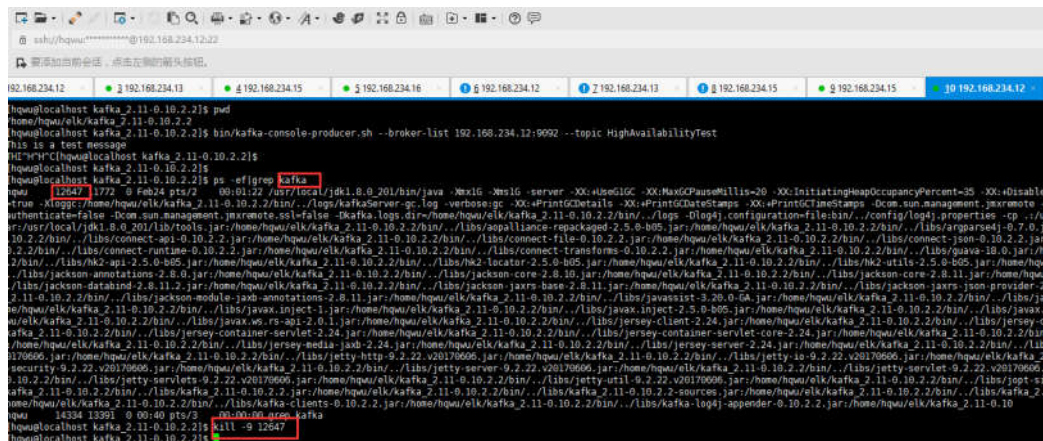


图 6-14 结束 kafka 进程

此时，节点 13 和节点 15 检测到节点故障。如图 6-15 为节点 13 发现节点 12 出现故障，图 6-16 为节点 15 发现节点 12 出现故障。

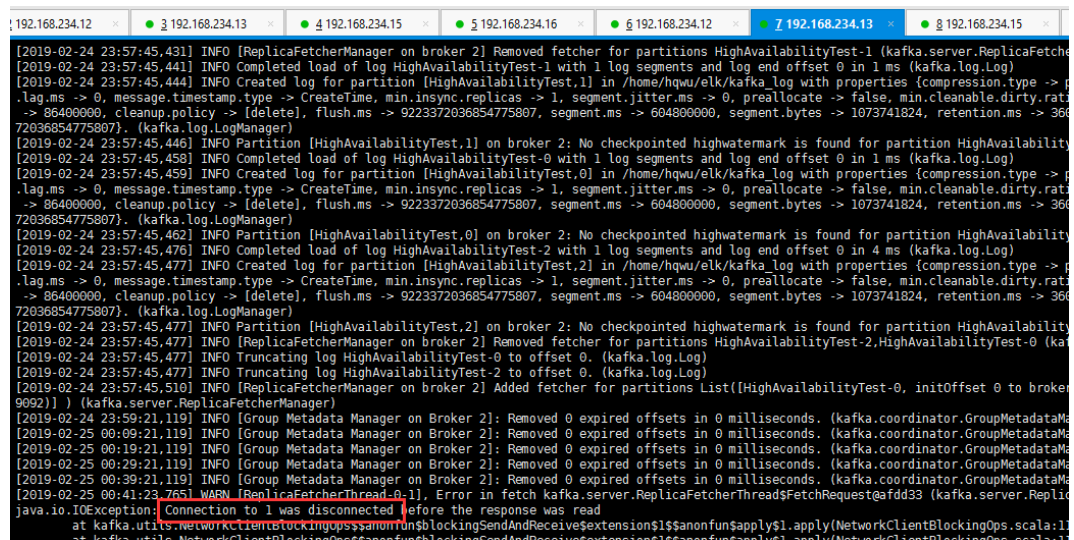


图 6-15 节点 13 发现节点 12 故障

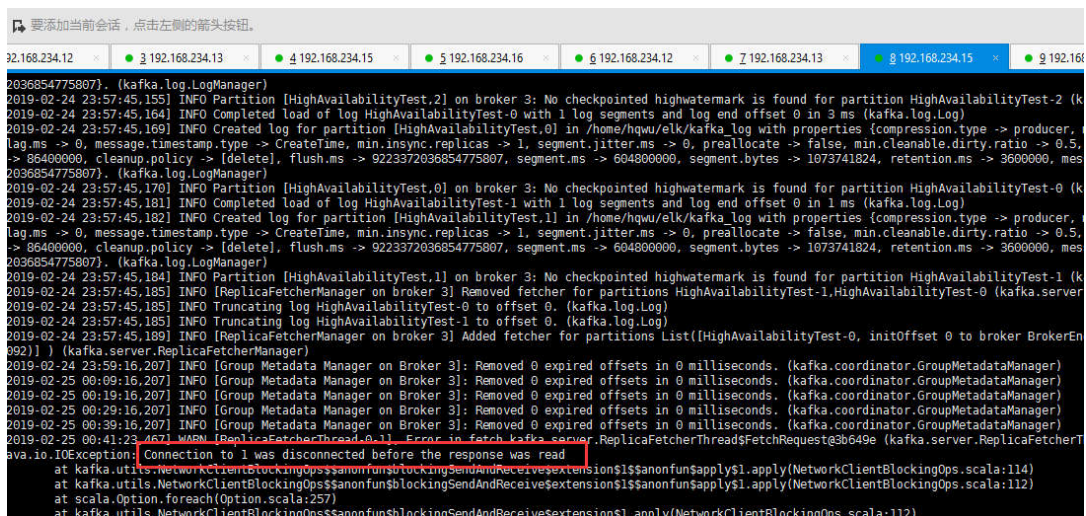


图 6-16 节点 15 发现节点 12 故障

发生故障后，HighAvailabilityTest 在集群中的详细信息如图 6-17。分区 1 原来的 leader 节点 12 出现故障，重新选择节点 13 为该分区的 leader 节点，此时三个分区的副本同步队列只有 broker 2 和 broker 3。同时，在节点 15 终端仍能够消费之前在节点 12 写入的消息。这说明，在节点 12 写入消息时，另外两个 kafka 节点对其该消息副本保存在自己的节点上，kafka 集群通过副本机制保证了数据的高可用性。因此验证了缓存模块能够保证日志数据在缓存模块中的高可用性。

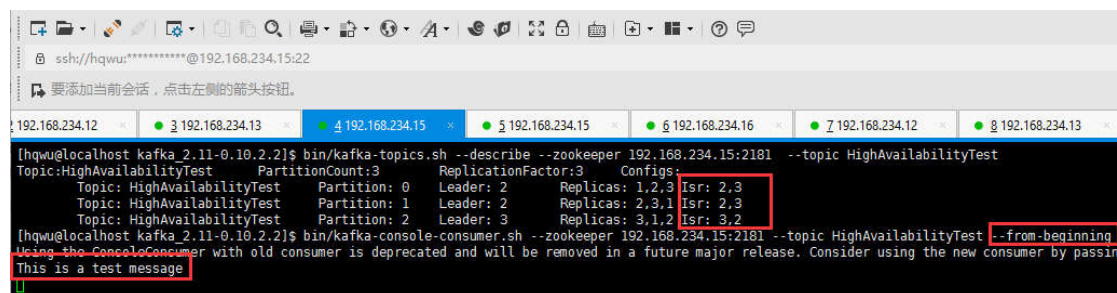


图 6-17 节点 12 故障后 HighAvailabilityTest 索引详细信息

6.6 日志检索层性能测试

6.6.1 查询性能测试

(1) 测试方法

本小节对 Elasticsearch 集群的查询测试分为关键字查询测试和组合查询测试，采用一款用于性能评测的开源软件 ApacheBenchmark(以下简称 ab)来对 Elasticsearch 集群进行查询测试。ab 的工作原理是创建多个线程，模拟多用户访问被测节点，其中 ab 对节点的通信协议为 HTTP 协议，且一个线程可以发送多次 HTTP 请求。

查询测试中测试的性能主要包括吞吐率和平均等待时间。吞吐率指的是每秒能处理的请求数，等待时间指的是客户端从发送请求到接收到结果的时间间隔。在高并发场景中，吞吐率越大，平均等待时间越少，则服务器对外服务的性能越好。

本小节主要做了两个测试：在关键字查询进行了测试环境下 Elasticsearch 集群的性能测试，并对相同配置的 Elasticsearch 单机服务做性能测试用于对比；对 Elasticsearch 集群进行组合查询测试，并进行了关键字查询与组合查询的集群性能对比。

(2) 测试过程及结果分析

关键字查询中，linux 终端使用 ab 的具体命令如下：

```
>ab -n 500 -c 500 -p search01 -q http://192.168.234.16:9200/logdata/product/_search
```

集群多节点的关键字查询测试的性能测试表如表 6-7 所示，单节点的关键字查询测

试的性能测试表如表 6-8 所示。

表 6-7 集群多节点关键字查询测试结果

总并发数	总请求数	吞吐率 (req/s)	平均等待时间 (ms)
200	200	725.73	275.584
300	300	945.26	317.374
400	400	1179.09	339.246
500	500	1402.87	359.413
600	600	1357.31	442.051
700	700	1334.31	524.614
800	800	1295.63	617.461
900	900	1304.02	690.176
1000	1000	1356.20	737.357
1100	1100	1378.70	797.854
1200	1200	1071.03	1120.422
1300	1300	1049.73	1182.527
1400	1400	988.57	1301.494
1500	1500	977.36	1354.635
1600	1600	975.40	1380.814

表 6-8 单节点关键字查询测试结果

总并发数	总请求数	吞吐率 (req/s)	平均等待时间 (ms)
200	200	1600.12	124.991
300	300	1387.74	232.966
400	400	1232.28	324.601
500	500	1114.68	448.561
600	600	1088.74	559.739
700	700	1054.39	663.889
800	800	1114.11	718.059
900	900	855.63	1051.859
1000	1000	918.30	1088.965
1100	1100	913.37	1204.337
1200	1200	911.03	1317.190
1300	1300	920.82	1353.746
1400	1400	905.68	1430.480
1500	1500	916.79	1460.327
1600	1600	920.47	1482.704

根据表格的测试数据绘制曲线图，可以更直观的反映吞吐率与并发数的关系。吞吐率与并发数的曲线图如图 6-18 所示。

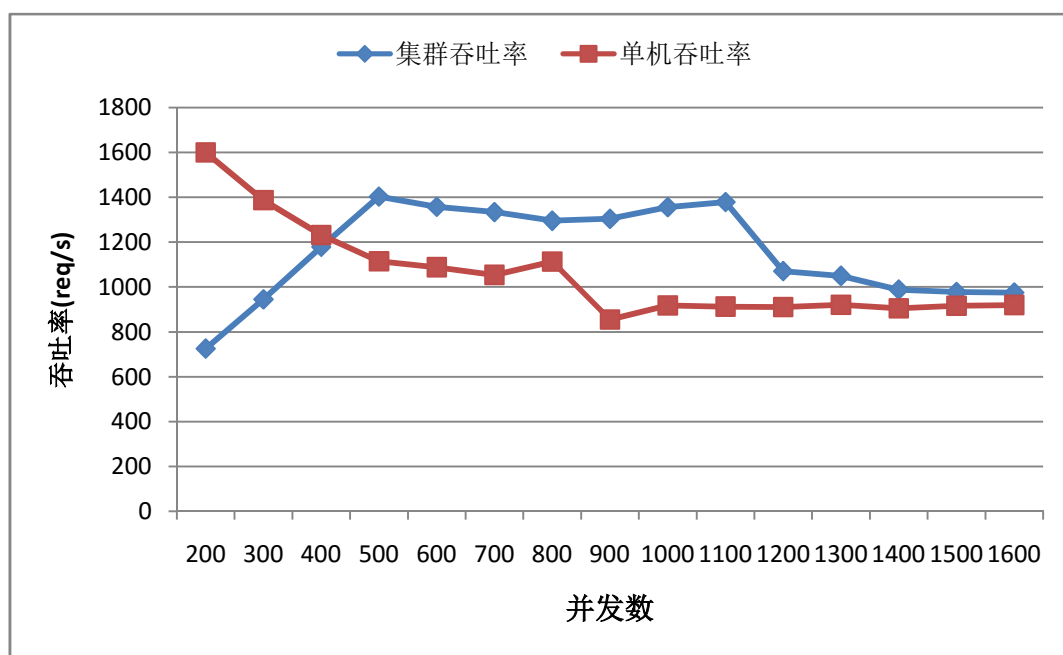


图 6-18 吞吐率与并发数折线图

由图 6-18 可以看出，在开始时随着并发数的增加，集群吞吐量也在线性增长，这反映了集群服务的资源在逐渐地被充分利用。受限于测试环境下机器的硬件配置，并发数在 500 至 1100 的范围时，集群吞吐率在 1300req/s 附近波动。当并发数超过 1200 时，受限于测试环境中节点的硬件资源，吞吐率急剧下降，原因是节点硬件资源并不足以及时处理并发请求，平均延迟在此时也急剧上升到 1120.422ms。本次测试环境下由单核 4G 内存的机器部署的 elasticsearch 集群在并发数 1100 时吞吐率达到 1300req/s 左右，通常实际生产过程中会采用多核 32G 内存等高配置的机器节点，且集群性能并不是简单的倍数关系，因此能够验证 Elasticsearch 集群在关键字查询上能够处理高并发查询请求。

在图 6-18 上，对比集群吞吐率和单机吞吐率，在开始并发量较小的时候，单机吞吐率远高于集群吞吐率，这可能是因为并发量太小，并不足以发挥出集群服务的性能，此时单机优势比较明显。当并发量不断增大时，单机的吞吐率逐渐降低，最终维持在 1000req/s 左右；而集群性能开始是线性增长，在集群能够处理的并发量下，吞吐率在 1300req/s 波动。

如图 6-19 所示是平均等待时间与并发数的曲线图，绘制该图同样能更直观的反映平均等待时间与并发数的关系。通过图 6-19 可以看出，平均等待时间随着并发数不断增大呈近似的线性增长。在开始时，单机的平均等待时间略低于集群多节点，并发数从 400 开始，集群处理高并发查询请求的优势开始体现，平均等待时间低于单机。

客户端的等待时间不超过 1s 时，用户不会有服务器卡顿的体验。在高并发场景集

群不仅需要保证吞吐率，也需要良好的客户体验，通常在这两者中寻找一个平衡点，使得既能保证高吞吐率，同时不会给客户带来卡顿的体验。在本文的测试环境中，当并发数为 1100 时，既能保证 1300req/s 的吞吐率，也能提供低于 1s 的平均等待时间，此时 Elasticsearch 集群既能保证高吞吐率，也能提供用户良好的查询体验。

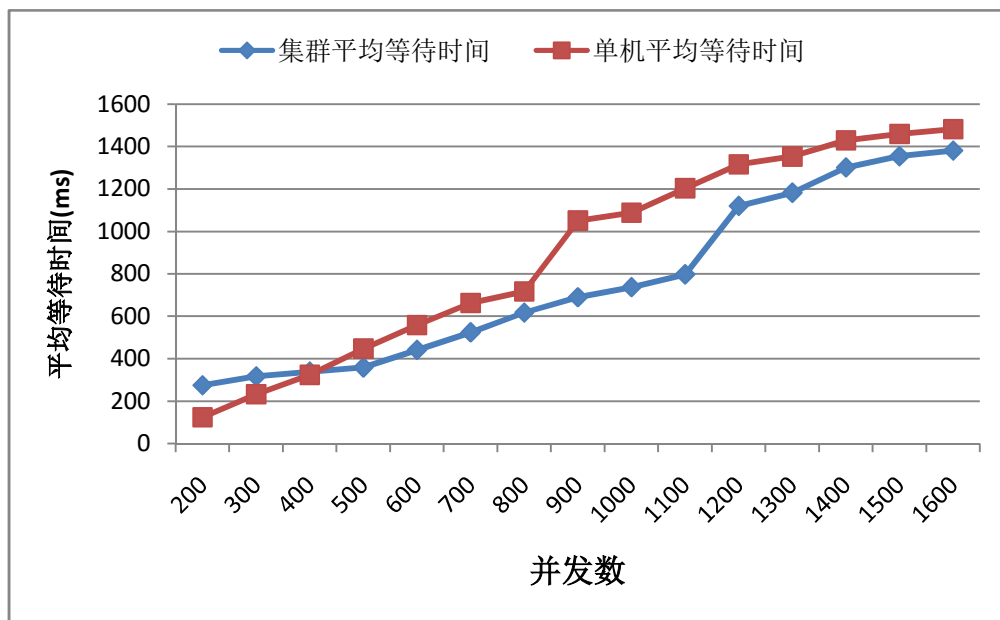


图 6-19 平均等待时间与并发数折线图

集群多节点的组合查询测试的性能测试表如表 6-9 所示。

表 6-9 多节点组合查询性能测试结果

总并发数	总请求数	吞吐率 (req/s)	平均等待时间 (ms)
200	200	351.31	569.299
300	300	405.84	739.202
400	400	412.58	969.513
500	500	375.55	1332.727
600	600	386.95	1550.568
700	700	401.84	1741.993
800	800	389.58	2053.478
900	900	402.41	2236.505
1000	1000	398.91	2506.849
1100	1100	351.07	3133.240
1200	1200	320.03	3649.621

将测试环境下 Elasticsearch 集群在关键字查询和组合查询的测试数据绘制在同一个曲线图，可以更直观的反映两种不同查询在性能上的差别。图 6-20 为两种查询类型的吞吐率与并发数的折线图。图 6-21 为两种查询类型的平均等待时间与并发数的折线图

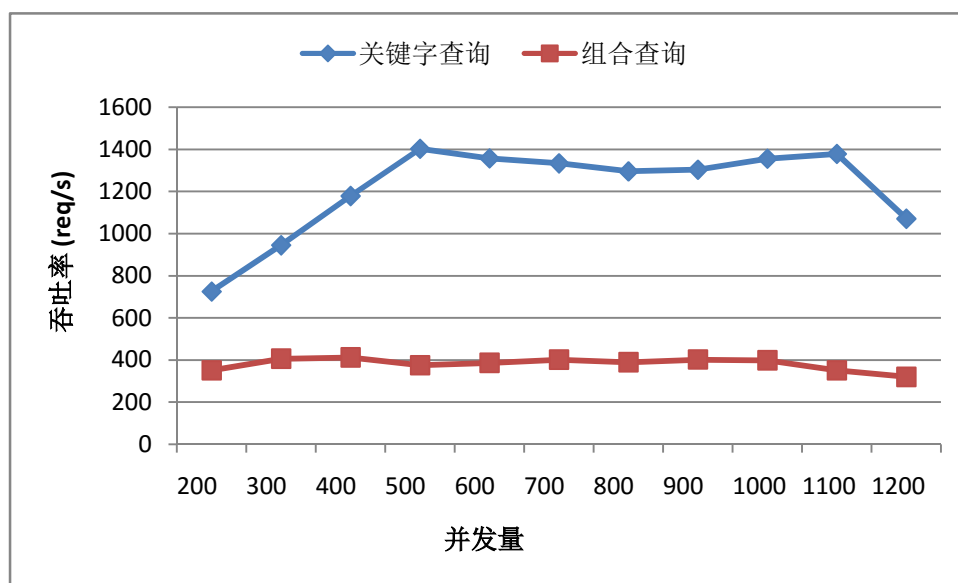


图 6-20 吞吐率与并发数折线图

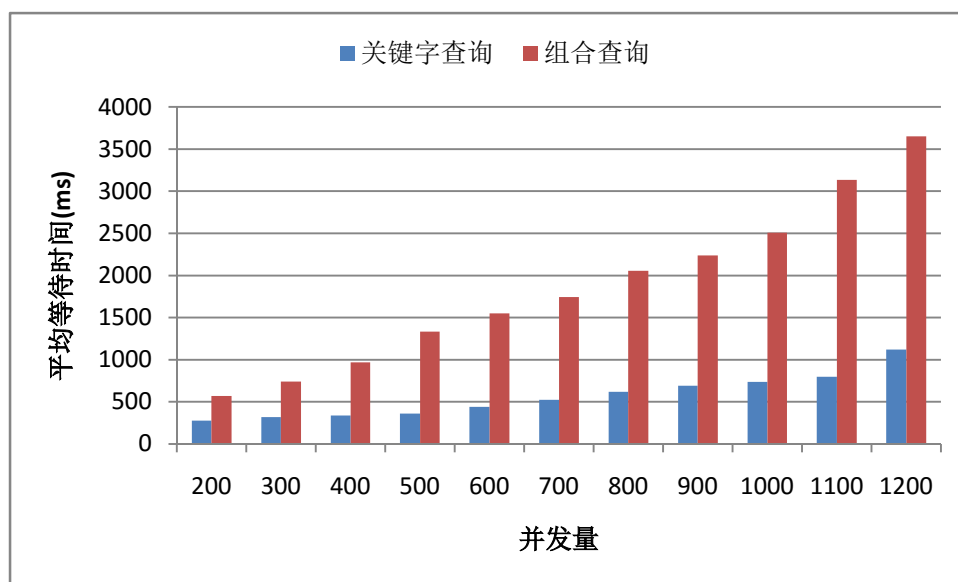


图 6-21 平均等待时间与并发数折线图

从图 6-20 可以很明显的看出，在相同并发数上，组合查询在吞吐率上远低于关键字查询，在平均等待时间上远高于关键字查询，原因在于组合查询加入了相关性打分机制，与单一的关键字查询不同，组合查询会评估每一个数据与查询条件的相关性，因此需要耗费较多的检索时间，导致吞吐率降低。受限于测试节点的机器配置，组合查询的吞吐率最终维持在 400req/s 上下。

从图 6-21 可以看出，在组合查询下，随着并发量的不断增加，平均等待时间也不断增加。当并发数超过 500 时，平均等待时间已经超过 1s，这会让用户有明显的卡顿体验。考虑到测试环境的节点配置较低，且实际工业生产活动的查询并发量尚未达到千量

级的体量，因此能够验证 Elasticsearch 集群能够满足工业日志系统的实时查询需求。

6.6.2 高可用性测试

(1) 测试方法

测试环境的 elasticsearch 集群有 3 个节点，本文首先在 elasticsearch 集群创建一个主分片数为 3，副本数为 2 的索引用于测试，通过对比集群正常工作和集群中的一个节点出现故障时索引分片分布的情况，来验证 elasticsearch 集群的高可用。本文采用 elasticsearch-head 插件来监控索引的分布情况，安装该插件后通过在浏览器访问所在节点的 9100 端口，可以查看 elasticsearch 集群的索引分片的分布情况。

(2) 测试过程及结果分析

图 6-22 创建一个主分片数为 3、副本数为 2 的索引，索引名为 highavailabilitytest。在 elasticsearch 集群索引分片的分布如图 6-23。图中加粗的边框为主分片，不加粗的边框为副本分片，框内的数字代表分片编号。通过图 6-23 可以看到，node-1 为主节点，主分片 0 在 node-1 节点，主分片 1 在 node-2 节点，主分片 3 在 node-3 节点，副本分片分布在与主分片不同节点的节点上。此时集群健康状态为 GREEN，表示集群中的主分片和副本分片都已经分配，集群 100%可用。

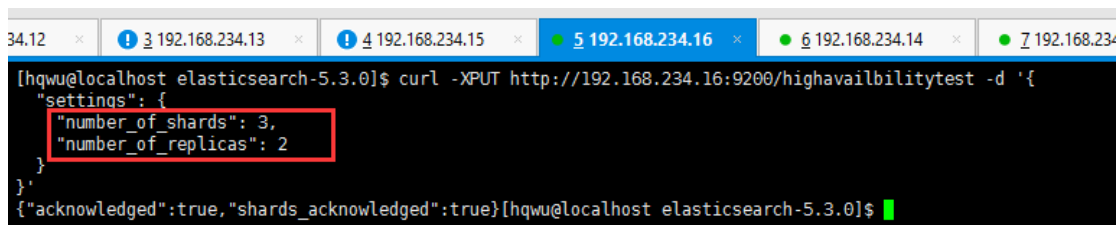


图 6-22 创建 highavailabilitytest 索引

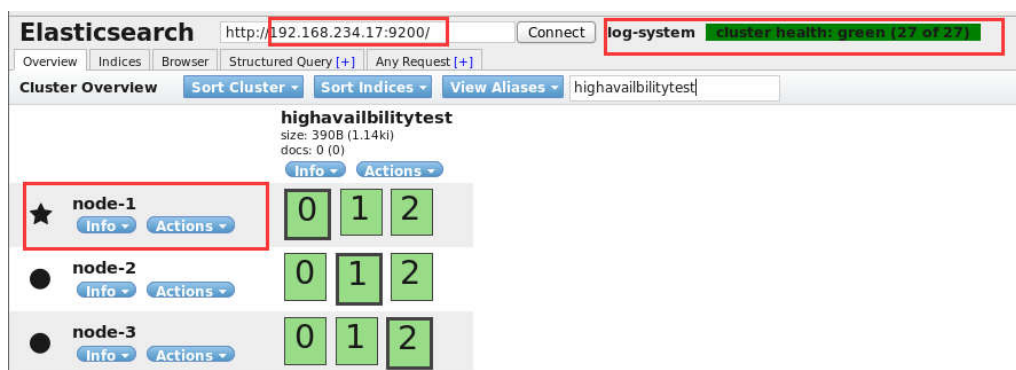


图 6-23 初始索引分片分布

通过 kill 命令结束 node-1 节点的 elasticsearch 进程，来模拟 node-1 节点发生故障。此时集群分片的分布情况如图 6-24。通过图 6-24 可以知道，master 节点变成 node-2，

集群健康状态变为 YELLOW，表示主分片都已经分片了，但至少有一个副本分片是缺失的。node-1 的主分片 0 丢失，此时 Elasticsearch 集群通过选举决定分片 0 的主节点变成 node-2 节点，node-2 节点原本的副本分片 0 成为主分片 0，此时仍能保证在分片 0 的日志数据可用，从而保证了数据的高可用性。



图 6-24 node-1 节点故障时索引分片分布

重新启动 node-1 节点的 elasticsearch 进程，node-1 节点通过单播机制加入 elasticsearch 集群。此时集群分片的分布情况如图 6-25。通过图 6-25 可以知道，node-2 仍然是 master 节点，集群健康状态重新变为 GREEN。node-1 节点分配的均为副本分片。

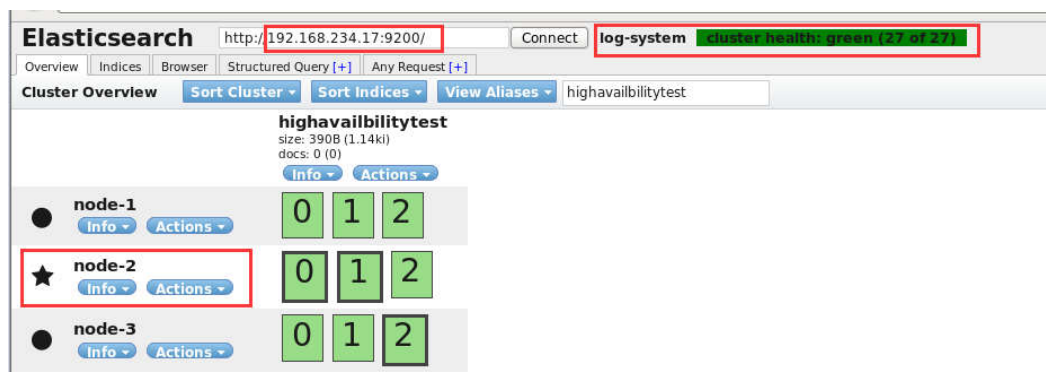


图 6-25 node-1 节点恢复后索引分片分布

因此，本节通过测试索引分片在 Elasticsearch 集群正常运行和故障时的分布情况，验证了 Elasticsearch 集群能够保证日志数据的高可用性。

6.7 测试分析总结

本章首先对分布式工业日志系统进行基本的功能测试，然后对日志系统各层进行性能测试以及结果分析。由基本功能测试的结果可以得出，日志系统中节点之间通信正常，能够根据配置的日志数据处理规则来处理日志数据，并经过日志检索模块的实时检索分析，在展示模块中输入查询条件得到实时的日志数据；由日志收集层性能测试结果可以得出，网络模块在模拟的高并发场景下能够保持高吞吐量，具备高并发处理日志存储请

求的能力，业务模块通过与传统同步日志对比，在相同的日志写入量测试中，所需的磁盘写入时间减少了 70% 左右，降低了磁盘 I/O 对于服务器性能的影响；由日志缓冲层性能测试结果可以得出，日志缓冲层 **kafka** 集群写入消息速度能够达到 36MB/s，被读取消息速度能够达到 130MB/s 左右，能够满足实际工业生产活动的需求，同时保证了日志数据的高可用性；由日志检索层性能测试结果可以得出，检索层能够提供不错的并发查询服务，且平均延迟在百毫秒级，能够满足实际工业生产的实时查询需求，且保证了日志数据的高可用性。

将本文设计的工业日志系统与绪论中的开源日志系统 **Scribe**、**Chukwa** 以及 **Flume** 相比，在数据高可用性方面，**Scribe**、**Flume** 在该方面并不能保证数据高可用，而本文的日志系统在收集层、缓冲层以及检索层均保证了数据的高可用性；在日志数据处理方面，**Scribe** 需要为日志内容额外附加一个 **category** 字段，**Flume** 的拦截器也需要日志内容附加额外字段才能对日志内容进行过滤，而本文的日志系统通过 **logstash** 开源组件使得日志产生端无需为日志内容附加额外的字段，且丰富的插件生态系统为用户编写日志数据处理规则提供了便利；在日志数据实时检索方面，**Scribe** 仅提供了日志数据存储功能，**Chukwa** 和 **Flume** 将日志数据写入到 **Hadoop** 中，而 **Hadoop** 更擅长于离线日志数据分析，本文的日志系统采用 **Elasticsearch** 实现了日志数据的实时检索。

因此，本文设计的日志系统解决了绪论 1.2.3 节对传统开源日志系统分析的三个问题，能够高并发处理日志存储请求，提供日志处理规则的灵活配置，保证了日志数据的高可用性，并提供了日志数据的实时检索功能。

总结与展望

总结

工业日志在工业生产活动中发挥着越来越重要的作用，然而，传统工业日志系统难以应对先进制造业中高并发的日志存储请求，对原始工业日志数据处理上较为简陋，缺乏对日志数据的实时检索分析，难以进行先进制造业的日志收集。本文针对传统工业日志系统的不足，分析了工业日志系统的实现要点，设计实现了基于 ELK 和 kafka 的分布式工业日志系统，主要工作内容包括：

(1) 设计了一个日志收集服务子系统，能够高并发的处理工业设备的日志存储请求，并降低了高并发场景下日志文件磁盘 I/O 对服务器节点的性能的影响。首先提出了该子系统的总体架构，分成网络模块、业务模块以及上报模块。网络模块将网络 I/O 事件分为连接事件和数据传输事件，首先研究了事件驱动模型，并该驱动模型的 Reactor 与 Proactor 两种模式进行比较，选择 Reactor 模型实现网络模块，同时，设计实现了 worker 线程池负责数据传输事件。业务模块通过设计实现环形内存缓冲 ringBuffer，解决了高并发场景下频繁的磁盘 I/O 写入问题。上报模块采用轻量级工具 filebeat，监控文件的新增记录，并上报到日志缓冲层。

(2) 设计了日志缓存服务子系统，提供日志数据处理规则的灵活配置，并实现对日志数据的缓存队列。首先提出了该子系统的总体架构，分成数据处理模块和缓存模块。数据处理模块采用开源组件 logstash，通过 logstash 丰富的插件实现对设备日志数据的简洁高效的处理，且系统运行时若修改日志数据处理规则能够重新加载处理规则。缓存模块采用 kafka 集群实现日志数据的缓存队列，采用 zookeeper 保证了集群节点状态的协调一致，并通过数据副本机制保证了 kafka 集群中数据的高可用性。

(3) 设计了日志计算中心，实现了日志数据的实时检索分析，并提供日志数据的可视化展示界面。首先提出了日志计算中心的总体架构，分成日志检索模块和展示模块。日志检索模块基于 Elasticsearch 集群，通过数据副本机制实现集群数据的高可用性，并进行性能调优。展示模块采用 kibana 实现日志检索模块中数据的可视化展示。

(4) 本文对设计的工业日志系统进行了系统测试，通过测试验证了本文日志系统较好地完成了设计目标，即本文日志系统实现了高并发处理日志存储请求，降低了高并发场景下日志磁盘 I/O 对于服务器节点性能的影响，提供了日志数据处理规则的灵活配置，实现了工业日志数据的实时检索分析，同时日志系统各层能够保证日志数据的高可用性。

展望

本文所设计的工业日志系统虽然能够很好地解决了传统工业日志系统存在的问题，然而，本文的日志系统仍有许多方面能够进行改进：

(1) 日志收集服务子系统中网络模块采用 `linux` 系统提供的线程机制实现，随着高并发服务框架技术的发展，少部分服务器采用一种基于协程的高并发服务器解决方案。协程也被称为用户态的线程，协程与线程不同之处在于协程无需陷入内核进行切换，减少了一定的内核开销，从而使得服务器能够实现更强的并发处理能力。因此，本文可以研究实现使用协程技术来实现网络模块，使得网络模块能有更好的高并发处理能力。

(2) 在数据处理模块中，本文目前只使用了 `logstash` 现成提供的插件来进行日志数据的处理，本文后续会在 `logstash` 中研究实现适配工业日志的自定义插件，使得日志系统中对日志数据的处理更为简便和高效。

(3) 工业日志系统还需要考虑保证工业日志数据的安全，本文设计的日志系统尚未对接入系统的节点进行身份验证，在未来还需要在系统安全方面做进一步研究。

参考文献

- [1]王晨, 宋亮, 李少昆. 工业互联网平台:发展趋势与挑战[J]. 中国工程科学, 2018,20(02):15-19.
- [2]王普刚. 基于HBase的工业日志系统设计与实现[D]. 大连理工大学, 2016.
- [3]葛岩. “工业4.0”时代的MES设计[J]. 电子技术与软件工程, 2015(09):14.
- [4]李培楠, 万劲波. 工业互联网发展与“两化”深度融合[J]. 中国科学院院刊, 2014,29(02):215-222.
- [5]吴智慧. 工业4.0:传统制造业转型升级的新思维与新模式[J]. 家具, 2015,36(01):1-7.
- [6]文宗瑜. 着力打通二三产“连接通道” 助推中国工业自动化智能化[J]. 发展研究, 2018(01):23-28.
- [7]周济. 智能制造——“中国制造2025”的主攻方向[J]. 中国机械工程, 2015,26(17):2273-2284.
- [8]Scribe[EB/OL]. <https://github.com/facebookarchive/scribe/wiki>.
- [9]郑荣. 统一日志系统中的日志获取模块与日志检索模块的设计与实现[Z]. 2018.
- [10]Chukwa[EB/OL]. <http://chukwa.apache.org>.
- [11]Jia B, Wlodarczyk T W, Rong C. Performance Considerations of Data Acquisition in Hadoop System[Z]. 2010545-549.
- [12]王玉芹. 基于Hadoop MapReduce作业调度方法研究[J]. 电脑知识与技术, 2018,14(27):10-11.
- [13]Flume[EB/OL]. <http://flume.apache.org>.
- [14]郝璇. 基于Apache Flume的分布式日志收集系统设计与实现[J]. 软件导刊, 2014,13(07):110-111.
- [15]邵安菊. “中国制造2025”与“工业4.0”的比较及推进路径研究[J]. 上海市经济管理干部学院学报, 2017,15(02):36-42.
- [16]黄振豪. 适配电商平台的分布式日志采集系统关键技术研究 and 系统实现[D]. 华南理工大学, 2017.
- [17]吉利, 潘林云, 刘姚. 线程池技术在网络服务器中的应用[J]. 计算机技术与发展, 2017,27(08):149-151.
- [18]林志勇. 面向RPC服务监控的集群监控系统的设计与实现[D]. 华南理工大学, 2017.

- [19]黄志龙. 面向超市电子货架标签的高并发服务器系统[D]. 华南理工大学, 2015.
- [20]叶柏龙, 刘蓬. Proactor模式的NIO框架的设计与实现[J]. 计算机应用与软件, 2014,31(09):110-113.
- [21]张苗. 面向分布式计算的编程框架设计与实现[D]. 电子科技大学, 2015.
- [22]乔平安, 颜景善, 周敏. 基于Linux系统的构建高性能服务器的研究[J]. 计算机与数字工程, 2016,44(04):653-657.
- [23]Guo X, Li Q. Fixed Bed Reactor Design Program Development Based on Java[J]. Journal of Software, 2014(5):1263-1269.
- [24]肖凯. 基于Reactor模式的Muduo网络框架技术研究[D]. 武汉邮电科学研究院, 2016.
- [25]Ramananandro T, Reis G D, Leroy X. A mechanized semantics for C++ object construction and destruction, with applications to resource management[J]. ACM SIGPLAN Notices, 2012(1):521-532.
- [26]Fay R, Bland J, Jones S. Next Generation Monitoring: Tier 2 Experience[J]. Journal of Physics: Conference Series, 2017(9).
- [27]于秦. 基于Apache Flume的大数据日志收集系统[J]. 中国新通信, 2016,18(18):41.
- [28]葛诗颖. Origin商城日志系统的日志管理子系统的设计与实现[D]. 南京大学, 2014.
- [29]王裕辰. 基于ELK Stack的实时日志分析系统的设计与实现[D]. 北京邮电大学, 2018.
- [30]孙超. Redis内存数据库在智慧消防系统设计中的应用[J]. 网络安全技术与应用, 2018(08):103-105.
- [31]刘芬, 王芳, 田昊. 基于Zookeeper的分布式锁服务及性能优化[J]. 计算机研究与发展, 2014,51(S1):229-234.
- [32]Dang H T, Canini M, Pedone F, et al. Paxos Made Switch-y[J]. ACM SIGCOMM Computer Communication Review, 2016(2):18-24.
- [33]Reed B, Junqueira F P. A simple totally ordered broadcast protocol[Z]. 20081-6.
- [34]袁子淇. 基于ZooKeeper的集群应用配置管理的设计与实现[D]. 内蒙古大学, 2015.
- [35]岳绍敏, 李万龙, 王璐, 等. 基于Lucene索引的数据库全文检索[J]. 吉林大学学报(理学版), 2014,52(05):995-1000.
- [36]姜康, 冯钧, 唐志贤, 等. 基于ElasticSearch的元数据搜索与共享平台[J]. 计算机与现代化, 2015(02):117-121.

- [37] 邹康. 基于Nutch的分布式搜索引擎的研究与实现[D]. 湖北工业大学, 2015.
- [38] 刘召明. 基于Elasticsearch的新闻实时词云系统设计与实现[D]. 华中科技大学, 2016.
- [39] Johansson T. Gossip spread in social network Models[J]. Physica A: Statistical Mechanics and its Applications, 2017:126-134.
- [40] 张仕将, 柴晶, 陈泽华, 等. 基于Gossip协议的拜占庭共识算法[J]. 计算机科学, 2018,45(02):20-24.
- [41] 周映, 韩晓霞. ELK日志分析平台在电子商务系统监控服务中的应用[J]. 信息技术与标准化, 2016(07):67-70.

攻读硕士学位期间取得的研究成果

一、已发表（包括已接受待发表）的论文，以及已投稿、或已成文打算投稿、或拟成文投稿的论文情况（只填写与学位论文内容相关的部分）：

序号	作者（全体作者，按顺序排列）	题 目	发表或投稿刊物名称、级别	发表的卷期、年月、页码	相当于学位论文的哪一部分（章、节）	被索引收录情况

注：在“发表的卷期、年月、页码”栏：

1 如果论文已发表，请填写发表的卷期、年月、页码；

2 如果论文已被接受，填写将要发表的卷期、年月；

3 以上都不是，请据实填写“已投稿”，“拟投稿”。

不够请另加页。

二、与学位内容相关的其它成果（包括专利、著作、获奖项目等）

[1] 巫辉强、向友君、吴宗泽.一种自适应均衡日志存储请求的方法（发明专利，已受理，申请号：201910168181.3）

[2] 巫辉强、向友君、吴宗泽.基于 C++异步日志存储系统 V1.0（软件版权，已登记，登记号：2019SR372794）

致谢

时光荏苒，研究生生涯一眨眼已经过了三年。三年的研究生时光，在 501、215 实验室的学习经历，感触良多。在此，我想对那些一直帮助我、鼓励我、支持我的人说声感谢。

首先，向我的导师向友君老师和吴宗泽老师表达最真挚的感谢！很幸运能成为你们的学生，在研究生期间给我们提供了一个广阔的项目平台，让我在平台上得到了许多历练的机会，受益匪浅。在生活中，你们也热心地帮我们解决在生活中遇到的问题，给予生活上的关心和指导。感谢向老师和吴老师在毕业论文过程中给予的全面细致的指导，对论文提出细致的修改意见，对我毕业论文的完成给予了极大的帮助。

同时感谢 501 实验室傅予力老师、周智恒老师、李波老师以及唐杰老师在学习以及生活中给予的关心和指导，老师们的言传身教，我铭记在心。

感谢实验室林志勇、张莉婷、黄振豪、蔡旭坤、苏春晨、李凯鑫、杨达、何煦、郑瑶强等师兄师姐的关心和帮助，感谢实验室同窗朱叶、胡卉馨、何家成、陈宏尘、周玉龙、刘涵、劳志辉、陈培林、罗怀烨、张隆琴、代雨鲲的一路陪伴与共同成长，感谢邱昱、容汉铿、董庆洲等师弟师妹们的支持与关怀！

最后我要感谢我的爸爸妈妈，正是他们对我无微不至地关怀和默默地付出，才有了我今天的小小成绩，我的健康成长离不开他们悉心的指导和教育。即将步入社会之际，我也准备好担负更多的责任，不辜负大家对我的期望。

最后，衷心祝愿在 501、215 的所有老师和小伙伴们身体健康，一切顺利！

巫辉强

2019 年 4 月于华南理工大学

3.答辩委员会对论文的评语

(主要包括: 1.对论文的综合评价; 2.对论文主要工作和创造性成果的简要介绍; 3.对作者掌握基础理论、专业知识程度、独立从事科研工作能力以及在答辩中表现的评价; 4.存在的不足之处和建议; 5.答辩委员会结论意见等)

工业日志在工业生产活动中发挥着尤为重要的作用,然而,传统工业日志系统难以应对先进制造业中高并发的日志存储请求,缺乏对工业日志的实时检索,数据处理方式较为简陋,难以对工业日志进行收集。作者针对工业日志系统展开研究,选题具有很好的工程实际意义和实用价值。

作者在对工业日志系统的发展现状进行了详细的调查和分析的基础上,研究了传统开源日志系统的日志收集方式,针对传统日志系统的不足,分析了工业日志系统的实现要点,采用分层设计的思想,设计并实现了一个基于 ELK 和 Kafka 的分布式工业日志系统。通过测试表明,日志系统整体实现了高并发的日志收集、日志处理规则的灵活配置、日志的实时检索的功能,并在各层保证了日志数据的高可用性。

该论文结构组织合理,撰写规范,工程量适中,具有一定创新性,表明作者已经具备相应的基础理论知识和专业水平。答辩过程中,作者准备充分,能够较好地回答评阅人及答辩委员提出的问题,展示出其对论文所属领域做了深入研究,具有较好的解决问题的能力。经答辩委员会讨论,一致认为论文已经达到硕士学位论文水平,通过答辩,建议学位评定委员会授予硕士学位。

论文答辩日期: 2019 年 5 月 31 日 答辩委员会委员 5 人


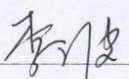
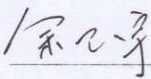
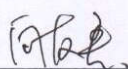
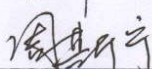
表决票数: 同意毕业及授予学位 (5) 票;

同意毕业,但不同意授予学位 (0) 票;

不同意毕业 (0) 票

表决结果 (打“√”): 通过 (√); 不通过 ()

决议: 同意授予硕士学位 (√) 不同意授予硕士学位 ()

答辩成员 签名	 (主席)		
			
答辩秘书 签名	