阿里云 | elastic

Elastic Stack 实战手册 (早鸟版)

从入门到应用，数十位大咖带你系统学习

- 开源技术界的一次大规模协作活动
- 聚集数十位技术创作人的一线经验
- 覆盖 Elastic Stack 全链路应用场景

阿里云开发者电子书系列

扫码申请成为

Elasticsearch 百人大作战联合创作人

扫码追踪书籍更新动态

认识优秀创作人

扫码加入 Elasticsearch 技术社区

参与技术交流

阿里云开发者“藏经阁”

海量电子书免费下载

推荐语

开源搜索引擎这十来年的发展，基本就是 Lucene 体系的发展。而基于 Lucene 的诸多搜索引擎中，Elasticsearch 以其极好的易用性、开箱即用的体验迅速折服了开发者。其边界也从最早的搜索引擎，扩展到了日志全观察、安全分析等场景，形成了今天的 Elastic Stack，具备从数据采集、处理、分析到展示的强大生态能力。

本书集成社区几十位开发者的心血，分入门、进阶、应用实践三大部分，从开发者最佳实践角度，循序渐进帮助大家更好使用、理解 Elastic Stack。Elasticsearch 应用十分广泛，成为大数据的必备工具之一，但我们发现，能熟练使用 Elasticsearch，熟悉 Elasticsearch 原理的人才非常稀缺，期待本书能帮助更多开发者成为 Elasticsearch 专家。

——阿里巴巴搜索推荐云服务技术负责人 - 郭瑞杰

祝贺本书的顺利出版，这是开源社区完美协助的又一次成功体现，数十位志愿者通力合作共同完成了本书的编写，也真切地感受到了各位社区小伙伴对 Elasticsearch 的喜爱之情。

本书从案例出发，不仅详细地介绍了 Elastic Stack 的各个功能组件的具体用法，还提供了不少来自一线的宝贵实践案例分享，不管是对于刚接触或是想要了解 Elasticsearch 更多知识的同学，本书都具有很好的参考价值。开卷有益，一起进入 Elastic 的精彩世界吧。

——Elastic 中文社区 创始人 - 曾勇

Elasticsearch 作为一个非常优秀的开源搜索引擎，已经被广泛应用到电商、APM、运维、企业搜索等业务场景中。《Elastic Stack 实战手册》是由不同业务领域的 Elasticsearch 专家参与编写的一部实战类的书籍，汇聚了各个业务场景下的 Elasticsearch 使用经验，包括 Elasticsearch 最新的一些特性，以及 Elasticsearch 在不同行业中的实践经验，无论是 Elasticsearch 的用户，还是在做搜索相关的技术选型，相信这本书都会给你带来一定的帮助。

——清博智能 技术 VP - 王欢

数据技术领域一直都受益于开源、开放，在信息过载的时代蓬勃发展。Elastic Stack 就是这数据激流中的标杆，基于开源，始于搜索，通过开箱即用的体验持续降低了搜索和大数据领域的门槛，如今已家喻户晓，成为优秀开发者的必备技能之一。

这份《Elastic Stack 实战手册》同样受益于社区的开放性，是集体智慧的结晶，相信定能帮助更多开发者学习、理解和掌握 Elastic Stack，并应用于各自的业务实践，用技术驱动业务增长并创建价值。在此我也呼吁更多开发者朋友积极参与进来，用建议和行动让“手册”得以持续进化。

—— Elastic 中文社区 深圳区负责人- 杨振涛

从 Lucene 到 Elasticsearch，到 ELK，再到当下 Elastic Stack，见证了这个产品体系逐步壮大发展到行业领先，也见证了大数据发展的技术迭代。Elastic Stack 已经越来越成为时代的标配，应用领域无处不在，没有使用 Elastic Stack 的企业不值得拥抱，没有掌握 Elastic Stack 的工程师不值得信任。

本书是迄今为止参与人数最多，综合方面最全的专业书籍，感谢社区集合了多方面

力量与资源，本人有幸参与审校，提前阅读了部分内容，推荐《应用实践部分》，毕竟能实战才是王道。

——力萌信息 数据领域专家 - 李猛

大数据近几年有了突飞猛进的发展，有数据的地方都离不开数据预处理、分析、检索、聚合、可视化分析等应用场景。以 Elasticsearch、Logstash、Kibana、Beats 等组成的 Elastic Stack，以其门槛低、上手快、版本迭代快、社区响应快等特点和优势，使得看似“遥不可及、高深莫测”的大数据存储、检索与分析技术“飞入寻常百姓家”。从一线大厂：阿里、腾讯、头条、滴滴、快手等到国内创业公司，甚至连婚庆网站都在使用 Elasticsearch。“Elastic 用的好，下班下的早”从一种半调侃的标语已然成为互联网实际人才衡量依据。

本人不才，从 1.X、2.X 版本接触 Elasticsearch，不觉中已有 6 年 + 的时间了，基本上是：以实际产品/项目驱动学习、以考试驱动学习、以社区互动输出、博客输出倒逼输入学习的方式。在 Elastic 中文社区、技术微信群、技术 QQ 群中经常被问到：“Elasticsearch 到底如何学？”，“Elasticsearch 基础进阶学习有没有书推荐？”我通常的建议是啃官方文档，必要时磕源码。但这种宽泛的建议不见得适合每一个人。今年上半年当我被邀请参加“Elasticsearch 百人大作战”的时候，我知道 Elastic 中文社区和阿里云要做这件事了。这是 Elasticsearch 圈子里的一件大事、幸事。

5 个月后，当我拿到《Elastic Stack 实战手册》这本电子书书稿的时候，我还是非常感动的。我深知多人异地协作创作的不易、感慨各位行业大佬的效率、感激各位创作者的“心血”。

全书以 7.10 版本（全网书籍最新）作为讲解版本，涵盖了应用场景、基础、应用

和实战案例。既包含了开源版本的核心基础功能，又创新性地包含了机器学习、高阶安全、APM、Elastic Cloud 等付费功能，难能可贵的是，这些高阶功能市面上的资料都非常少。本书的亮点和特色就在于：将企业实战业务场景的方案进行了全方位的解读，包含但不限于：基于舆情的全文检索场景、基于智能巡检、流媒体、面部识别等基础的日志分析场景，这些来源于实战的解读对于企业架构选型、开发、运维都非常有帮助。

在此，我向大家推荐这本电子书，期望大家和我一样读后有收获！“前事不忘后事之师”，也期望大家一起（包括我自己）多给 Elasitc 社区做贡献，期望我们的 Elastic 社区成为我们学习、进阶、成长的“伊甸园”！

最后，以“共和国勋章”获得者、中国工程院袁隆平院士 2019 年湖南农大演讲的一句话，“汗水指的是要能吃苦，任何一个科研成果都来自于深入细致的实干和苦干”和大家共勉。我坚信：获得科研成果要吃苦、掌握任何一门技术也要吃苦，Elasticsearch 也不例外！

——Elasticsearch 公众号作者 - 铭毅天下

目录

一、序言

Welcome reader!

On behalf of Elastic, I'm honoured to present this amazing work from hundreds of our Alibaba and Elastic community contributors, and we are proud to have worked alongside everyone involved in bringing it to fruition. We'd like to specially thank Alibaba for their assistance and expertise in preparing the project, the community members who shared their experiences and knowledge in providing content pieces, and our Elastic team who leveraged their expertise to editorialise the final out come.If you are new to the Elastic Stack, or an experienced practitioner, this book will give you many examples of how to deploy the Elastic Stack to Search, Observe and Protect your data and it's systems, and create valued insights and information.We also invite you to join our extended Alibaba and Elastic communities, both on and off line, where you can meet with peers and share knowledge around your use of the Elastic Stack on Alibaba Cloud. Alibaba and Elastic host recurring joint user group sessions, online webinars, technical blog posts, forums, conferences and more, to keep you educated on the work we do to progress the technology behind the Elastic Stack. The Alibaba and Elastic relationship grows stronger every day, and the future is bright as we continue to work together in helping our joint community embrace the Elastic Stack on Alibaba Cloud.

Mark Walkom

Community Team Lead - APJ

Elastic

二、导读

《Elastic Stack 实战手册》的创作发布，源自于阿里云和 Elastic 联合主办的三周年系列活动（https://developer.aliyun.com/topic/esanniv3rd）——Elasticsearch 百人大作战。这是一次有趣的大规模协作活动，集结了 Elasticsearch 技术圈百位开发者共创，旨在凝聚圈内优秀的创作人的实践经验和创作能力，输出一本能为开发者提供实践参考的书籍指南，推动技术应用和发展。

在活动过程中，本书得到了数十位在大数据搜索领域颇有经验的优秀开发者的支持，其中包括许多知名的业界精英，例如在社区有一定影响力的技术大咖，具有大数据及 Elastic Stack 相关书籍出版经验的作者等。这是一个自发所组成的团队努力的成果，也是一次成功的尝试。

本书涵盖了一个 Elastic Stack 开发者所需的大部分知识，尤其对于刚入门的开发者而言，是一本非常值得推荐的参考读物。本书内容由浅入深，从基础的 Elastic Stack 产品能力到后半部的应用实践，为开发者使用 ElasticStack 提供了必要的基础知识和应用参考。

全书基于 Elastic Stack 7.10 创作，包含完整的 Elastic 产品能力篇和应用实践篇。2021 年 6 月发布的版本为书籍第一期，囊括了全书的大部分内容，但仍有部分章节待补充，之后本书还将继续迭代，并随着 Elastic Stack 的版本升级持续更新。

书籍主要包含如下篇章：

Elastic 产品能力篇

本篇介绍了 Elastic 的三大产品能力和应用方法。为了便于阅读，满足不同阶段开发者的需求，本篇分为三个部分：

基础篇：如果你是一位对于 Elastic Stack 还不甚了解的开发者，非常建议阅读本章内容。本章包含了 Elastic Stack 的基础介绍、能力组成，以及技术优势等。同时，还介绍了 Elasticsearch 中必要了解的专用术语，以便正确理解后文的内容。

入门篇：本章是学习 Elastic Stack 夯实基础的部分，涵盖了非常广泛的内容，主要分为以下：

1、Elastic Stack 部署：本章节介绍了如何安装 Elastic Stack 的 4 个重要组件（包括 Elasticsearch, Kibana, Beats 及 Logstash）。安装环境包括本地部署，Docker 部署，以及如何使用阿里云快速部署一个 Elastic 集群。同时还包括必要的技能——如何部署一个多节点的集群。

通过学习如何为集群配置安全，创建用户并为不同的用户配置不同的角色，你将深刻体会到安全对一个集群的重要性。

2、Elasticsearch 基础应用：Elasticsearch 是整个 Elastic Stack 的核心，本章节也是理解 Elasticsearch 最重要的部分。

通过本章的阅读，你将学习如何创建 Elasticsearch 索引 （CURD），如何对索引创建 mapping，倒排索引到底是什么，如何分词，以及如何对数据进行扁平化、结构化，并对他们进行搜索。

同时了解不同的数据类型（object，nested，joined 等），动态 mapping，Index template，分布式计分，reindex，分页搜索，ingest pipeline，脚本编程，aggregation 等。

当你掌握了本章节的内容，相信你可以很自豪地对自己说：我也是一个dataengineer。

3、Kibana 基础应用：Kibana 是 Elastic Stack 的可视化窗口，担负着对 Elastic Stack 进行管理和监控的重任。本章节对于了解 Kibana 是非常有益的。

本章介绍如何使用 Kibana 对数据进行搜索、分析。重点学习如何使用 Discover 进行数据搜索，使用时间过滤器选择数据，创建 index pattern。同时，了解制作不同的可视化图（垂直条形图、地图、仪表盘、指标、数据表、折线图、饼图等），并组成面板（Dashboard）以及使用 Lens 制作可视化图的方法技巧。

进阶篇：本章建议读者在对 Elastic Stack 有基本了解的前提下，针对一些功能进行深入学习。掌握这个章节对于希望将 Elastic Stack 熟练运用到场景实践中的读者而言非常关键。

在本章节，你将了解 CCR/CCS，Rollup，Data stream，索引生命周期管理，分片管理（shard allocation/awareness），Snapshot，Watcher，Alerts，Transform，Graph，机器学习，Elasticsearch SQL，Canvas，APM，Uptime，Enterprise Search，SIEM 及 Endpoint Security，集群管理及监控，Elasticsearch 各类语言开发示例，ECS 等高级进阶功能。

应用实践篇

本篇也是全书的核心部分之一，主要以三大产品能力场景（企业搜索、可观测性和安全能力），以及性能优化场景划分。全书最终将呈现数十个各行各业应用 ElasticStack 产品能力的应用实践案例，每个案例自成一体，独立于其它案例。读者可以选择自己感兴趣或符合应用场景需求的案例进行阅读参考。

众所周知，Elastic Stack 的迭代速度非常快，每一两个月就会有一个小版本发布，因此，每位开发者都需要不断地学习，更新知识。希望通过本书的阅读，能够为读者学习 Elastic Stack 提供实用的参考意义。

同时，我们也深知，《Elastic Stack 实战手册》是一本由众多创作人共同书写的书籍，涉及了几十个人的视角、技术经验和创作习惯，这正是共创的趣味所在，也隐藏着许多不足。但这本书刚刚孵化起步，相信在未来的更新迭代中，它将无限趋近于更实用、更丰富。

我们也在此呼吁每一位开发者，将自己技术积累书写成文章，加入创作人群体，与我们一起学习交流、同频共振，这将不只是部分人的书籍，而是所有 Elastic Stack 开发者的实战指南。

学习拓展：

- 更多 Elastic Stack 知识，请访问 https://www.elastic.co/guide/index.html

- 阿里云 Elasticsearch 支持本书所需的学习环境，2c4g 3 节点免费试用 30 天 https://www.aliyun.com/product/bigdata/product/elasticsearch?utm_content=g_1000275202
- 成为联合创作人，请点击以下链接申请报名 https://survey.aliyun.com/apps/zhiliao/XlZHsCMxb

Elasticsearch 的前世今生

创作人：曾勇

需求的诞生

刘备一大早就来到了公司，一看张飞和关羽已经在公司了，就问道：“两位贤弟，今天来的还蛮早啊。”张飞一听就炸毛了，“大哥，你让我和二哥去做什么搜索功能，我们已经一晚没睡了，昨天就没回去好嘛。” 关羽也来气，“大哥，是啊，我们刚刚才上线电商网站，你这边又要加什么需求，现在用数据库检索不是好好的么，能不能让我们歇口气。”

“两位兄弟辛苦了，我也不想啊，最近咱们一单生意都没有啊。昨天我和一位朋友聊，他说我们的网站很不好用，找不到他想要的鞋，结果只好去别的地方买了。不过他给我推荐了一位黑客高手，叫诸葛亮的家伙，说是啥都得懂，我们今天找他取经去。”

三顾茅庐

三人一行来找诸葛亮，不过前面两次都碰了壁。据诸葛亮书童说，诸葛亮不在家，到了第三次，还是不在家。张飞仔细一听，明明是有人在家啊，而且玩游戏喊的声音还这么大，张飞怒了，搭梯子把诸葛亮家的保险给拔了。诸葛亮正郁闷呢，咋停电了呢？算了，今天没得玩了，于是让书童请他们进来。

“在下诸葛名亮，字孔明，不知三位...”，三人一说，是这么这么回事。诸葛亮一听，“哦，原来是这么这么回事啊，你们的网站我刚看了，你们家的草鞋品种确实不 Nan 少 Kan。如今客户上网站找东西，都是先用网站的搜索来搜一下，但是你们网站的搜索功能实在是太 La 弱 Ji，明摆在那里的商品我都搜不出来，实在是大问题啊。”

“这样啊，我看你们仨都是好人，给你们推荐一个好东西，叫做 Elasticsearch，这个肯定可以帮助你们。”

“翼德，把先生放下来吧。”

“是，大哥。二哥，你把刀也放下吧。”

关羽一听，好像在哪里听说过 Elasticsearch，“大哥，这个东西好像有点耳熟啊，

哦，诸葛亮先生这一说，我倒是记起来了，隔壁公司的吕布最近神神秘秘的，好像就是在用这个，难怪他们最近公司业务好的很”。

Elasticsearch 的故事

诸葛亮清了清嗓子，又从抽屉里摸出一把扇子，“还是让我来给你们讲讲吧”。

“Elasticsearch 以前叫 Elastic Search。顾名思义，就是“弹性的搜索”。很明显，它一开始是围绕着搜索功能，打造了一个分布式搜索引擎，底层是基于开源的搜索引擎库 Lucene，是由 Java 语言编写的，项目大概是 2010 年 2 月份在 Github 正式落户的。

咳咳，有必要首先给你介绍一下 Lucene。Lucene 是一个非常古老的搜索引擎工具包，也是用 Java 编写，主要用来构建倒排索引（一种数据结构）和对这些索引进行检索，从而实现全文检索功能。

Lucene 很强大，使用起来也非常灵活，缺点是它仅仅是一个基础类库，也没有考虑到高并发和分布式的场景。如果你想在自己的程序里面使用 Lucene，还是需要做很多工作，并且涉及很多搜索原理和索引数据结构的知识，这就给我们带来了不少挑战。所以，Lucene 的上手时间一般都比较长。”

关羽插了一句，“Lucene 我知道，确实贼难用，使用起来一堆问题啊，我之前试过来着。” 关羽说完，脸又红了。

诸葛亮接着说。“时间一晃来到 2004 年，有一个以色列小伙子，名字叫谢伊·班农（ Shay Banon），他成亲不久来到伦敦，因为当时他的夫人正好在伦敦学厨师。初来乍到，也没有找到工作，于是班农就打算写一个叫作 iCook 的小程序来管理和搜索菜谱，一来练练手，方便找工作；二来这个小工具还可以给其夫人用。

班农在编写 iCook 的过程中，使用了 Lucene，感受到了直接使用 Lucene 开发程序的各种暴击和痛苦，于是他在 Lucene 之上，封装了一个叫作 Compass 的程序框架，与 Hibernate 和 JPA 等 ORM 框架进行集成，通过操作对象的方式来自动地调用 Lucene 以构建索引。

这样做的好处是，可以很方便地实现对‘领域对象’进行索引的创建，并实现‘字段级别’的检索，以及实现‘全文搜索’功能。可以说，Compass 大大简化了给 Java 程序添加搜索功能的开发。Compass 开源出来，变得很流行。

在 Compass 编写到 2.x 版本的时候，社区里面出现了更多需求，比如需要有处理更多数据的能力以及分布式的设计。班农发现只有重写 Compass ，才能更好地实现这些分布式搜索的需求，于是 Compass 3.0 就没有了，取而代之的是一个全新的项目，也就是 Elasticsearch。”

让人砰然心动的 Elasticsearch

看到刘备三人听的入迷，诸葛亮轻挥羽扇，继续说了下去。

“得益于 Compass 项目的积累，Elasticsearch 问世之初就考虑到了功能的易用性。

Elasticsearch 作为一个独立的搜索服务器，提供了非常方便的搜索功能。用户完全不用关心底层 Lucene 的细节，只需要通过标准的 Http+RESTful 风格的 API，就可以进行索引数据的增删改查。数据的输入输出采用 JSON 格式，以文档和面向对象的方式，这样就能非常方便地理解和表达领域数据。”

张飞一拍桌子，“Elasticsearch 简直就是一个 Compass 的 RESTful 实现啊！”

“没错。同时，Elasticsearch 基于分片和副本的方式实现了一个分布式的 Lucene Directory，再结合 Map-reduce 的理念，实现了一个简单的搜索请求分发合并的策略，能轻松化解海量索引和分布式高可用的问题。

可以说，仅仅依靠这两点，Elasticsearch 就已经秒杀了当时市面上所有的搜索引擎服务或是程序库，我当时看到 Elasticsearch 也眼前一亮。

如今，Elasticsearch 基本上已经是搜索引擎市场排名第一的产品了，从 DB-Engines 网站的排名可以看到，Elasitcsearch 基本上是一骑绝红尘，拉开第二名远远一大截。”

17 systems in ranking, May 2018

Rank May 2018	Rank Apr 2018	Rank May 2017	DBMS	Database Model	Score May 2018	Score Apr 2018	Score May 2017
1.	1.	1.	Elasticsearch	Search engine	130.44	-0.92	+21.62
2.	2.	↑ 3.	Splunk	Search engine	65.09	+0.04	+8.40
3.	3.	↓ 2.	Solr	Search engine	61.51	-1.70	-2.26
4.	4.	4.	MarkLogic	Multi-model	10.39	-0.23	-1.09
5.	5.	5.	Sphinx	Search engine	6.32	+0.05	-0.92
6.	6.	6.	Microsoft Azure Search	Search engine	4.25	+0.00	+1.26
7.	7.	↑ 9.	Algolia	Search engine	3.16	+0.16	+0.91
8.	8.	↓ 7.	Google Search Appliance	Search engine	2.68	-0.08	-0.30
9.	9.	↓ 8.	Amazon CloudSearch	Search engine	2.28	+0.07	+0.01
10.	10.	↑ 11.	CrateDB	Multi-model	0.48	+0.01	-0.10
11.	11.	↓ 10.	Xapian	Search engine	0.46	+0.00	-0.12
12.	12.	12.	SearchBlox	Search engine	0.25	-0.01	-0.00
13.	13.	↑ 16.	DBSight	Search engine	0.00	-0.01	-0.08
13.	↑ 14.	↑ 14.	Exorbyte	Search engine	0.00	±0.00	-0.18
13.	↑ 14.	13.	Indica	Search engine	0.00	±0.00	-0.24
13.	↑ 14.		Manticore Search	Search engine	0.00	±0.00	
13.	↑ 14.	↑ 17.	searchxml	Multi-model	0.00	±0.00	±0.00

统计数据来源：https://db-engines.com/en/ranking/search+engine

ELK 横空出世

诸葛亮口水狂飙，显得很兴奋，“如果只是 Elasticsearch 单独使用，那我们的故事也就结束了，事实上好戏这才刚刚开始。俗话说，一个好汉三个帮，开源社区亦是如此。”

“这一个好汉三个帮，说的不就是咱仨嘛。” 刘备接过话茬。

“别打岔，”诸葛亮继续说，“这里我要说的是 ‘ELK’ 的出现，不过首先我要给你们讲讲 Logstash。”

“Logstash 是一个开源的日志处理工具，用 JRuby 写的，主要特点是基于灵活的 Pipeline 管道架构来处理数据。什么意思呢？可以理解为将数据放进一个管道内进行处理，并且就跟真正的自来水管一样，管道由一截一截管子组成，每一个小管代表着一个数据处理的流程，每一个流程只做一件事情，然后可以根据数据的处理需要，选择多个不同类型的管子灵活组装。

Logstash 社区非常活跃，支持多种输入数据源和多种输出数据源。一开始，Elasticsearch 只是作为其中一个输出的存储，主要用于日志数据的存储。

不过，随着大家把日志发送到 Elasticsearch 之后，大家发现这家伙用起来很方便嘛，不仅能够存储大量的数据，水平伸缩还很方便。更关键的是，你能够很方便地把数据找出来，也就是进行全文搜索。

全文搜索在日志分析里面是非常基础的一个功能，通过一个关键字就能定位具体的详细日志，相比存放到关系型数据库和普通的文件存储，Elasticsearch 优势非常明显。于是 Logstash 搭配 Elasticsearch 变得很受欢迎。

Kibana 的故事

不过 Logstash 自带的 UI 查询日志的界面有点简陋，于是有一个叫作 Rashid Khan 的运维工程师表示完全忍不了了，用 PHP 写了一个叫作 Kibana 的程序，一个更好看和更好用的前端界面。PHP 写完一版，他又用 Ruby 写一版，后面又用 AngularJS 写了一版，不仅有日志的搜索和查看，还加上了一些统计展示功能。

Kibana 的名字其实是俩个水果的名字的组合（Kiwi+Banana）。

张飞听到这里：“工作不饱和啊这家伙”。孔明瞪了他一眼，继续说道。

这个时候，Elasticsearch 已经有 Facet 概念，也就是分面统计（ 注：1.0 之后推出了 Aggregation 来代替 Facet），可以对数据里面的某个字段进行单个维度的统计，支持多种统计类型。比如， TermFacet 可以计算字段里面某些值出现了多少次；Histogram Facet 还可以按时间区间进行汇总统计等。这些统计功能在前端 UI 就可以被利用起来，展示一些饼图、时间曲线等等，在运维的分析里面自然也都是需要的。慢慢的 Kibana 越做越复杂，支持的功能越来越多，Kibana 3 变得流行起来。

于是乎，ELK 横空出世（Elasticsearch、Logstash 和 Kibana 这三个产品的首字母缩写），风靡了整个运维界。

故事讲到这里，相信你们对于 Elasticsearch 就有了一个大概的认识，可以用它做搜索，也可以用它做日志。”

张飞点点头，“还是相当的强悍嘛。”

Elastic Stack 平台的魅力

“不过，这还没完。”诸葛亮吞了吞口水，继续说。

“Elastic 后面又引入了 **Beats 家族**。这是一系列非常轻量级的数据收集端，我给

你介绍几个比较典型的，比如：

- Packetbeat

  可以实时监听网卡流量，并实时解析网络协议数据，可用来做 NPM 网络数据分析；

- Metricbeat

  可以用来收集服务器，以及服务器上部署的应用服务的各项监控指标数据，这样就可以替代 Zabbix 等传统的监控软件，来做服务器的性能指标分析；

- Auditbeat

  可以实时收集服务器的行为事件，用于安全方面的入侵检测和安全日志审计分析；

- Winlogbeat

  用于 Windows 平台的事件日志收集；

- Filebeat

  用于日志文件的收集等。

Elasticsearch、Logstash、Kibana、Beats ，这几个放在一起，就叫作 Elastic Stack。

如今，Elastic 的版图越来越大，前年，Elastic 收购 Opbeat，开源了业界第一个完整的 APM 解决方案，通过探针可以实现无侵入的代码级别的应用性能监控；去年 7 月又收购了代码搜索 Insight.IO，后续可以实现代码级别的语义检索。今年又收购了一

个做终端安全的厂商 Endgame。这样 Elastic Stack 这一个平台就可以同时做到：

- 日志分析
- 性能指标分析
- 安全日志分析
- APM 应用性能分析
- NPM 网络性能分析
- 网站站内搜索
- 企业级搜索
- 代码搜索
- 实时 BI 业务分析
- SIEM 解决方案
- 终端设备安全
- ……

试想一下：

在一个风和日丽的下午，你手机上收到一条告警短信，于是点击链接，打开 Kibana 的监控仪表盘，发现某台服务器的 CPU 达到 100% 了。

于是，你顺手点击过滤这台服务器的所有相关信息，可以看到相关的日志显示，是这台服务器上面部署的某一个业务服务的 QPS 有显著下降，然后过滤到这个业务的日志，发现有很多异常的日志信息，前端 Nginx 代理日志还显示有很多请求被拒绝，看样子是后端的微服务处理能力达到瓶颈。

这个时候，继续点击 APM 的分析面板，切换到事务和会话分析界面，看到有很多数据库链接处于开启状态。你点击查看调用代码，立马就找到了性能瓶颈的原因，原来是某个类的某个方法调用 MySQL 却没有及时释放链接造成了泄露，于是修改这行代码，提交上线，问题解决。然后，你可以若无其事地继续浏览相亲网站啦。

尽管这是一个假想的例子，但是可以看到，基于 Elastic Stack ，你可以覆盖一整套完整的，从全局性能监控到具体代码级别的排障和解决问题的过程，并且使用起来要比很多现有的方案更加高效和便捷。

好了，现在你们是否对 Elasticsearch 已经有了一个初步的了解呢？是不是也有跃跃欲试的打算？”

刘备点点头：“今天来先生这里真的是收获不少，之前多有冒犯，还请多多包涵啊。”

关羽也说：“大哥，明天我就和三弟开始研究 Elasticsearch，争取早日改造好咱们的网站。”

“刚说的相亲网站要不也发我一下”，张飞连忙问道。刘备没好气白了一眼张飞。

“天色已晚，告辞了！”

刘备三人作别孔明，各自高兴的回家了。

“慢走不送，有空来喝茶啊。”

孔明抹了一把额头，总算送走这仨了，恐怕从此江湖上估计要不平静喽。

创作人简介：

曾勇，Elastic 在中国的第一个员工，是中国最早接触 Elasticsearch 的一波人，也是 Elastic 中文社区的创始人和社区主席，目前在 Elastic 负责中国企业客户的项目落地与技术咨询，具有丰富的 Elastic 项目实施经验。

博客：http://medcl.com

三、产品能力

3.1 理解 Elastic Stack

3.1.1 从 Elasticsearch 到 Elastic Stack

编辑：葛丽丽

从早期开发的 Elasticsearch 到之后 ELK Stack 的发布，Elastic 在此期间经历了辉煌发展，也有混乱的时期，随后又推出了 Elastic Stack，并迎来了新的时代。

2000 年

源自查找菜谱的 APP

伦敦的公寓内，Shay Banon 正在忙着寻找工作，而他的妻子正在蓝带（Le CordonBleu）烹饪学校学习厨艺。在空闲时间，他开始编写搜索引擎来帮助妻子管理越来越丰富的菜谱。

他的首个迭代版本叫做 Compass，第二个迭代版本就是 Elasticsearch（基于 Apache Lucene 开发）。之后，他将 Elasticsearch 作为开源产品发布给公众，并创建了#Elasticsearch IRC 通道，剩下来就是静待用户出现了。

产品正式发布后，公众反响十分强烈，用户自然而然地就喜欢上了这款软件。由于

使用量急速攀升，此软件开始有了自己的社区，并引起了人们的高度关注，尤其引发了几位创始人 StevenSchuurman、Uri Boness 和 Simon Willnauer 的浓厚兴趣。最终，他们四人共同组建了一家搜索公司。

2012 年

Search Inc. 阶段

在 Elasticsearch Inc. 成立前后，另外两个开源项目也正在跨越式发展。

Jordan Sissel 当时正在开发 Logstash，这是一款开源的可插拔数据采集工具，可将日志文件发送至用户选择的“储藏库”。除此之外，他还在开发一款 UI，以实现日志数据的可视化，但这一产品的稳定性却实在让人难以恭维。

幸运的是，还有其他人也在潜心钻研可视化这个难题。这个人就是 Rashid Khan，他当时在开发一款名为 Kibana 的开源 UI。

Shay、Jordan 和 Rashid 彼此已认识了一段时间，对各自的产品也颇为了解，所以他们最终决定携手共同发展，ELK Stack 正式面世，即：Elasticsearch、Logstash 和 Kibana。

不久之后，Elasticsearch Inc.就推出了两个商用插件：一是用于监测的 Marvel，二是用于防护的 Shield。

2015 年

更名为 Elastic；喜纳 Found

在 2015 年于旧金山举行的 Elastic{ON} 大会上，Elastic 宣布了两项重要决定：第一，将公司品牌更名为 Elastic，新的品牌名称能够更好地代表逐渐扩大的产品生态系统和用例套件；第二，Elastic 与在 AWS 上提供 Elasticsearch 主机托管服务的公司 Found 实现了合作。通过这一合作，Elastic 能够提供市场上最简单、最全面的产品组合。

最初发展的问题

早期，Elastic 开发和发布软件时采用的是工程师各自为战的方法：工程师可以在任何时候推出任何喜欢的版本，唯一的要求就是产品要好。Kibana 有公测版，Logstash

采用里程碑，Elasticsearch 则采用数字编号。如果工程师高兴，还可以推出插件。尽管十分混乱，但是一切还算行得通，直到最后无法使用。

随着用户通过产品来完成越来越多的任务，Elastic 需要开发更好的产品来为用户提供更多帮助，所以添加了更多功能，开发了新插件和扩展。产品的确变得越来越好了，然而也越来越复杂，技术栈变得越来越混乱。

例如，如果运行的 Elasticsearch 是 1.7 版本，而运行的其他插件是 2.3 版本，则软件不能自动检测二者是否兼容，也无法验证插件是否在无预警的情况下已不能正常使用。

在 Elastic，也开始听到内部员工说：“如果想使用 Shield，需要使用 Elasticsearch 1.4.2，但前提是不能使用 Watcher。如果使用 Watcher 的话，则需要使用 Elasticsearch 1.5.2。而如果使用 Elasticsearch1.5.2 的话，其仅能与 Kibana 4.0.x、Logstash1.4.x、Shield 1.2.x 和 Watcher 1.0.x 兼容。”

Elastic 的版本控制做得一团糟，必须得研究对策；同时，支持矩阵也表现欠佳。

调整业务步伐，推出 Beats

就在产品团队为版本编号忙得团团转的时候，另外一个产品故事正在拉开序幕。Elastic 在 2015 年迎来了 Packetbeat，这是一家夫妻档公司，致力于开发一种轻量化方式来将网络数据发送至 Elasticsearch。

这启发了当时的 Elastic：如果开发一系列单一用途的轻量化数据传送工具，将网络数据、日志、指标、审计数据等从边缘机器传输到 Logstash 和 Elasticsearch，结果会怎样？就这样，Beats 应运而生了。

同步推出新产品版本

2015 年 10 月对 Elastic 来说是一个重大转折点，因为解决了产品版本编号问题，同时也降低了兼容性的复杂程度。

这一发布版本又称为“Bonanza 同步版”，是 Elastic 第一次在同一天面向公众发布全部产品：Elasticsearch 2.0、Logstash 2.0、Watcher 2.0、Shield 2.0 和 Kibana 4.2。

通过这次调整，用户得以更轻松地启用产品，同时也提高了产品的可靠性，帮助用户出色地完成任务。

一键部署，Elastic Cloud 隆重推出

几个月后，“Bonanza 同步版”不再仅仅局限于供人们下载的产品，通过 Elastic Cloud（即之前的 Found），在 AWS 上推出了 Elasticsearch 和 Kibana 服务。

2016 年

Elastic Stack 5.0

Elastic 致力于推出更为成熟的产品系列，通过发布 Elasticsearch 2.0 来统一发布步调就是第一步，5.0 的发布则是第二步。与之前的所有版本相比，用户通过这一版本可以体验集成性能更强，经过更严格测试且更加易于入门的产品。

5.0 发行版本的同时还将所有商用插件（当时被称为 Shield、Marvel 和 Watcher）整合为单一扩展，即 X-Pack，其包含了核心产品，诸如 security、monitoring 和 alerting 等的功能，并且随着 Prelert 公司也加入 Elastic，machine learning 也开始纳入其中。

模块应运而生

在 5.3 版本中，Filebeat 正式引入了“模块”的概念，可以将模块理解为用于在 Elastic Stack 中传输、解析、存储、分析常用日志格式（例如 Apache、Nginx 和 MySQL 等），并实现可视化的一组安全配置，它简化了用户从数据集至仪表板的入门体验。

Metricbeat 和 Packetbeat 的模块都各具特色，几个月后，Logstash 也将针对 ArcSight 和 NetFlow 数据引入自身模块。

2017 年

ECE 面世

这年，Elastic 采纳了管理自身 Elastic Cloud 服务时所用的技术，并发布了 Elastic Cloud Enterprise（又称 ECE），让所有规模的公司均能下载全部的托管产品，并独立运行，享受其带来的益处。有了 ECE，无论是一个集群，还是数千个集群，用户都能够顺利地对其进行管理，而且还可以简化在任何环境中对 Elastic 产品和解决方案的管理和编排工作。

Elastic 解决方案加速演进

随着模块数量的成倍增加，使用 Elastic Stack 来处理特定用例（例如日志或指标）开始变得越来越简单。几个月后，Elastic 并购了应用程序性能监测(APM) 公司 Opbeat，以及站点和企业搜索公司 Swiftype。

此时，Elastic 的发展已经日趋成熟，可以提供解决常见问题的精简方式。这些解决方案包含从 DIY 到更加一站式的体验，每套解决方案的背后都有真实的产品作为支撑，而且在几分钟内即可部署完毕。

2018 年

开放 X-Pack 代码

2018 年，Elastic 决定开放商用 X-Pack 功能的源码，从而加速开发周期，提高社区互动，并让每个人都能够贡献源码，对源码添加评论，并对其进行检查。

因此，用户能够更加轻松地使用 Elastic Stack，所有的 X-Pack 功能都默认提供 Elasticsearch、Kibana、Beats 和 Logstash。但这一改变并未删除任何 Apache 2.0 源码。恰恰相反，Elastic 在开放源码的发展道路上加大了力度。

纽交所敲钟

作为一家分布式公司，2018 年 10 月 5 日，Elastic 在纽约证券交易所敲钟，正式成为一家上市公司。交易大厅里多达 230 名 Elastic 员工聚在这里（创下了纪录），全球各地员工共同庆祝这一重要时刻。

Elastic 的发展历程还在继续，相信之后的探索发现之旅将会越来越精彩。

3.2 核心应用场景

3.2.1 企业搜索

创作人：朱永生

什么是企业搜索

企业搜索，顾名思义，就是企业使用的搜索服务或者说是企业提供的搜索服务。具体可以是企业的客户，使用企业提供的搜索服务，搜索企业提供的产品、服务等信息；比如电商企业提供搜索服务供客户搜索商品信息、应用市场提供搜素服务供用户查找 APP 等；也可能是企业内部各个部门成员，使用企业内部的搜索服务，搜索企业内的各种信息，比如项目信息、代码信息、文档信息等等。

企业搜索的特点

企业搜索因为不同的使用场景，具有其自己的特点。相较于大家熟悉和常用的百度、谷歌等互联网搜素，企业搜索有如下不同：

数据来源不同

众所周知，百度、谷歌等互联网搜索引擎，主要通过网络爬虫抓取互联网上的数据；

而企业搜索的数据主要来源于企业自身，由企业自己的数据源提供。

数据内容不同

互联网搜索引擎抓取的数据，主要是各个网站公开的各种网页、图片、音频、视频、文档等；而企业搜素处理的数据主要是企业内部提供的私有信息，如产品信息、项目信息、内部文档、办公软件、邮件、数据库等等。同时，企业搜索也可以包括公开的各种数据。

数据更新频率不同

互联网搜索抓取数据是爬虫被动执行的，抓取到新的数据需要一定的时间，数据更新频率由于各种因素存在不确定性，数据更新可能并不及时；而企业搜索的数据源是企业自主可控的，数据往往是企业主动生成的，数据更新基本是实时的。

数据完整性不同

互联网搜索抓取数据，因为各种因素，比如网站列表无法做到完整、网站 Robots 禁止抓取、法律政策等，无法做到抓取和显示所有数据，用户搜索不到需要的数据是正常现象；而企业搜索的数据都是企业预先设定的，用户搜索的结果应按照设计进行展现，搜索不到本该展示的数据是不可接受的。

面向的用户和需求不同

互联网搜索面向的是大众普通用户，搜素方式方法和搜索结果，一般都不会因个别用户或部分用户的需求而改变；企业搜索面向的是企业内部用户或是企业某项业务的客户，在搜索方式上要尽力贴近用户习惯，在搜索结果上要足够完整和准确，能确切表达业务诉求。

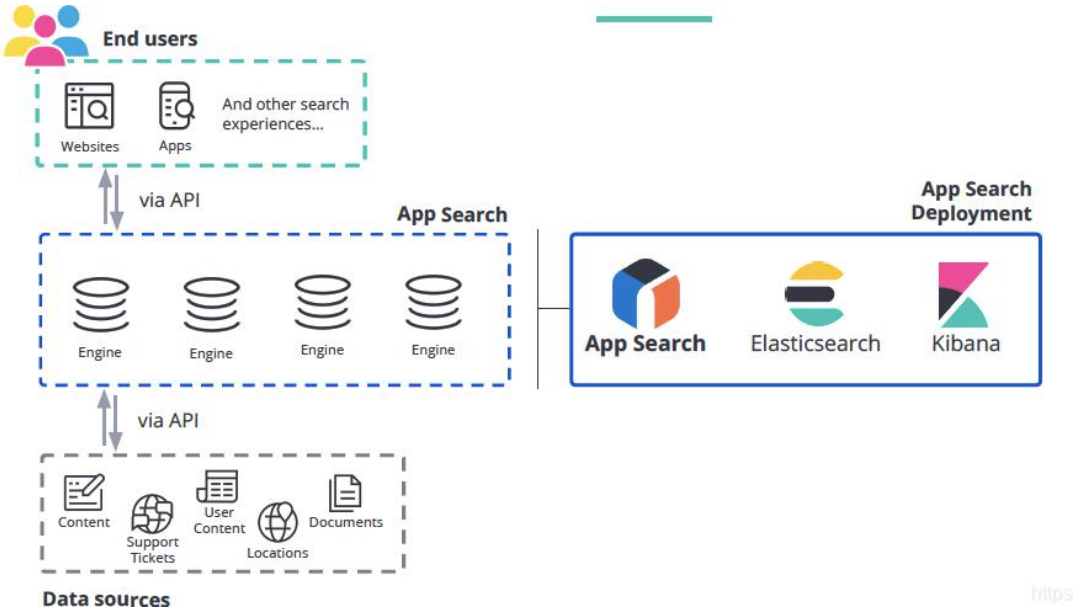
搜索结果的可控性不同

使用互联网搜索的用户，搜索出来的结果不会因用户的不同而不同，搜索结果均以 PageRank 算法为基础进行排序展示，所有用户可搜到的结果基本是一致的；而企业搜索的结果需要根据用户的权限进行控制，不同权限的用户搜索到的结果是不同的，不该对用户显示的结果不能显示；同时，企业搜索的结果需要能够进行显式控制，比如通过排序策略、权重策略等，甚至需要直接处理搜索结果从而控制搜索结果。

Elastic 企业搜索能力介绍

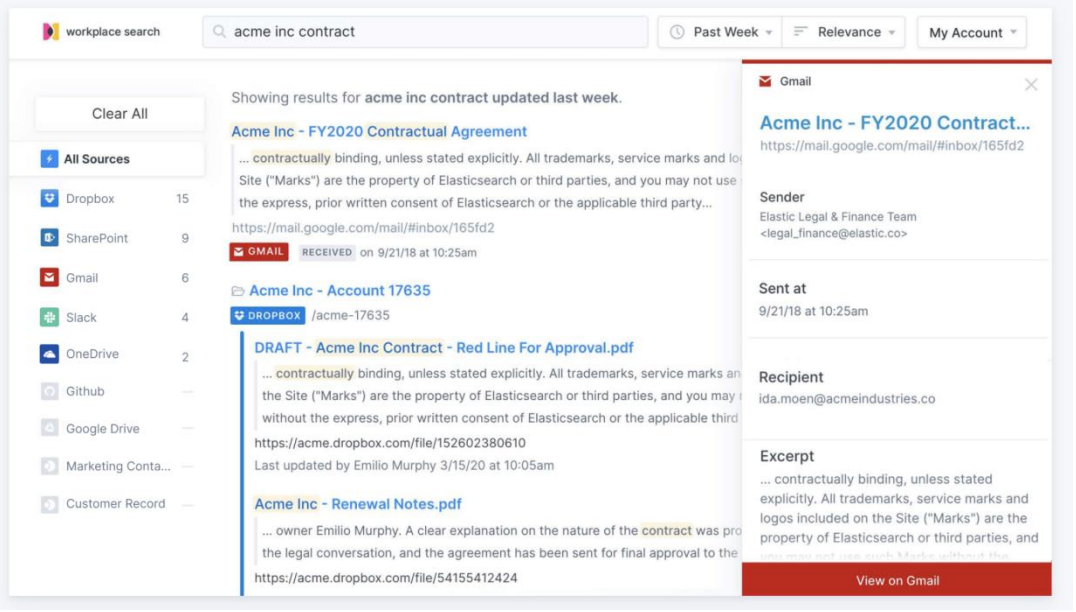
Elasticsearch 是基于 Apache Lucene 的分布式搜索引擎，本身就具有全文检索、多用户、近实时，可用于搜索各种文档的能力。而 Elastic 秉承让产品更易于使用的理念，在 Elastic Stack 7.2 中引入了 Elastic App Search，在 Elastic Stack 7.7 中推出 Elastic Workspace Search 正式版，并将 Elastic App Search 、 Elastic Workspace Search，Site Search 打包成了单独的解决方案，其名称就叫 Elastic Enterprise Search，也就是 Elastic 企业搜索。App Search 、Workspace Search、Site Search 基本覆盖了企业的所有搜索应用场景。

App Search 针对企业产品应用搜索场景，在 Elasticserach 强大的存储和分析功能之上，提供经过优化的 API、直观的仪表板、易用可调的相关控件以及可快速集成的客户端。



App Search 系统架构图

Workspace Search 针对企业内部办公搜索场景，提供无缝连接办公协作效率工具向导和 API，借助 Elasticsearch 构建集中信息源，对分散在各个办公软件中的信息和文档，设置自动同步并进行再组织和定制，解决团队协同办公过程中的信息孤岛问题。常见的办公软件如 Salesforce，Dropbox，Google docs，Sharepoint，Jira，Confluence 等都提供了友好的接入向导，当然也可以使用自定义源接入其他的系统。Workspace Search 可针对每名团队成员进行权限控制、相关性配置、个性化结果定制等，在安全可控的范围内，帮助团队提高获取信息的速度、完整性并提高信息利用率。



Workspace Search 自带搜索界面展示图

Site Search 的核心是网页爬虫，是一套帮助企业快速构建网站搜索功能的工具。只要输入网址，爬虫就可以自动采集内容并自行定期更新，也支持用户手动对特定页面或者整个网站重新索引。Site Search 可以通过自动更正、双连词匹配、词干提取、同义词等功能，提供复杂查询的支撑；也可以通过直观的界面快速调整页面排名、增减权重和同义词等。

虽然 App Search 、Workspace Search、Site Search 针对的应用场景有所不同，但都是企业搜索场景，并且相关支撑能力也是通用的或者类似的。下面我们就通过了解 Elastic Enterprise Search 解决方案来理解 Elastic 企业搜索能力。

快速部署能力

Elastic Enterprise Search 支持四种部署方式，分别是 Elastic 云实例、Elastic 云上 Kubernetes 集群部署、Linux/MacOS 包部署和 Docker 容器镜像部署。四种部署方式都非常简单快速，相对来说，阿里云 Elasticsearch 服务实例门槛最低且功能丰富，支持 30 天免费使用，适合快速学习了解产品功能；而 Linux/MacOS 包部署相对复杂一些，适合熟悉操作系统和想了解安装部署细节配置的用户；如果不想使用云服务也不想一步步下载和配置安装包，那么使用 Docker 部署是一个好的选择。

统一认证能力

Elastic App Search 和 Elastic Workspace Search 支持标准的用户名密码模式、Elasticsearch 本地域模式和 Elasticsearch SAML 第三方统一认证模式进行登陆认证和角色授权。其中标准用户名密码模式，由管理员在 Elastic App Search 或 Elastic Workspace Search 的面板上对用户进行管理；Elasticsearch 本地域模式 Elasticsearch Native Realm 由 Elasticsearch 直接管理和存储用户信息；Elasticsearch SAML 模式是 Elasticsearch 使用第三方统一认证进行用户的登陆认证，而 Elastic App Search 和 Elastic Workspace Search 直接继承了 Elasticearch 中的 SAML 配置。

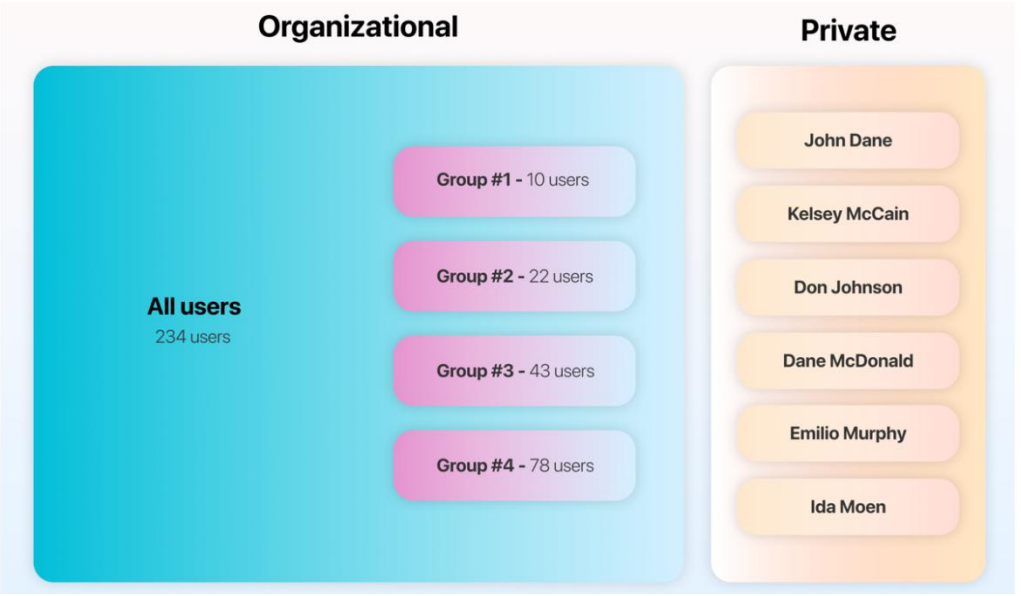
角色授权能力

不管使用哪种登陆认证模式，Elastic 企业搜索均支持按角色授权，不过针对每种认证模式，授权的方式略有区别。在标准的用户名密码认证模式下，Elastic App Search 使用基于角色的访问控制 (Role Based Access Control) 对用户进行授权，可授权的角色有：Owner、Admin、Dev、Editor、Analyst 等；而 Elastic Workspace 是基于数据

内容权限、用户所在部门等因素对用户进行分组，然后对分组进行授权，是基于用户组的访问控制对用户进行授权。在 Elasticsearch Native Realm 和 Elasticsearch SAML 认证模式下，Elastic App Search 和 Elastic Workspace Search 都使用角色映射对用户进行授权，先在 Elasticsearch 中创建角色，然后在 Elastic App Search 和 Elastic Workspace Search 中对 Elasticsearch 中创建的角色进行映射。Elastic App Search 中可映射的角色有：Owner、Admin、Dev、Editor、Analyst 等，Elastic Workspace Search 中可映射的角色有：Admin、User。

支持不同级别的内容源

Workspace Search 可以采集各种来源的数据内容，支持使用自定义 API 接入，同时针对 GitHub、Jira、Confluence、Google Driver、OneDriver、SharePoint Online、Gmail、Slack 等十几种常见办公应用，提供了方便进行接入的数据采集向导。另外，Workspace Search 支持 Organization Content Sources 组织内容源和 Private Content Sources 私有内容源，也支持 Standard Content Sources 标准内容源和 Standard Content Sources 远程内容源。组织内容源一般由管理员配置，供整个组织使用；而私有内容源可由个别用户自己配置并仅供自己使用。标准内容源中的所有源数据都将被进行采集并存储；而远程内容源仅采集部分信息，依赖数据源的搜索端点进行数据检索。因为标准内容源采集的是全量数据，如果有多个用户对同一个内容源建立了多个数据连接，那么数据就会被采集并存储多份，对 Elasticsearch 的存储容量影响很大；而远程数据源因为采集的数据非常少，在相同情况下，对 Elasticsearch 的影响非常小。当然，建立可检索的远程内容源有个前提条件，就是远程内容源本身是有检索端点的。



Site Search 的网页爬虫，只要输入网址，爬虫就可以自动采集内容并自行定期更新，并且支持用户手动对特定页面或者整个网站重新索引。

支持文档级别权限

Workspace Search 支持启用源文档权限同步，支持的应用包括：Jira Cloud、Confluence Cloud、Google Driver、OneDriver、SharePoint Online 等。其他自定义接入的内容源也可以使用 _allow_permissions 和 _deny_permissions fields 字段来配置文档级别权限。如下代码为文档配置权限:

```
{
    "_allow_permissions":[
        "permission1"
    ],
```

```
        "_deny_permissions":[

        ],
        "id":1235,
        "title":"The Meaning of Sleep",
        "body":"Rest, recharge, and connect to the Ether.",
        "url":"https://example.com",
        "created_at":"2019-06-01T12:00:00+00:00",
        "type":"list"
    }
```

如下代码为用户分配权限：

```
    curl -X POST \
    http://localhost:3002/api/ws/v1/sources/[CONTENT_SOURCE_ID]/permissions/[USER_NAME]
\
    -H "Authorization: Bearer [ACCESS_TOKEN]" \
    -H 'Content-Type: application/json' \
         -d '{
        "permissions":[
            "permission1"
        ]
    }'
```

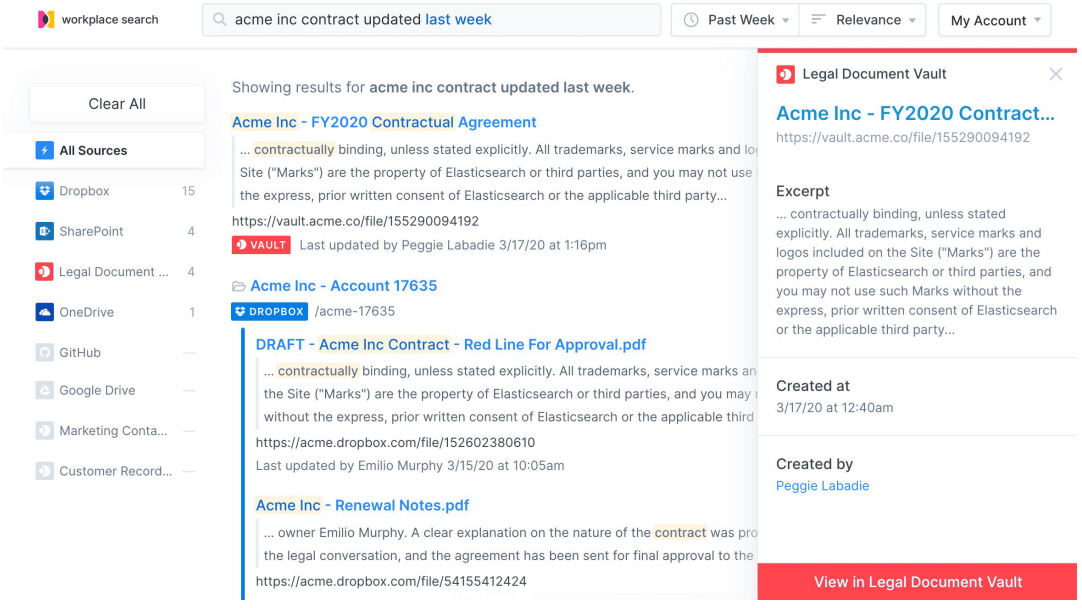
支持 Meta Engine

App Search 支持 Meta Engine。Meta Engine 本身不存储文档，是将多个源文档

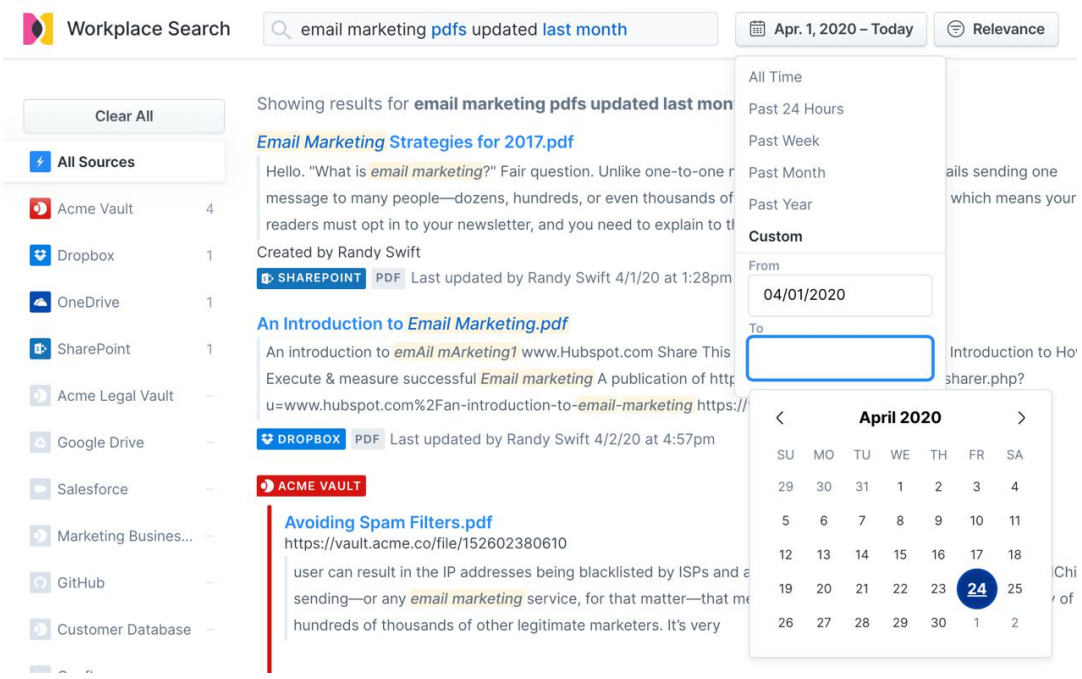
引擎进行结合，让用户可以通过搜索单个元引擎，搜索到多个源文档引擎中的内容。

支持自定义搜索体验

在 Workspace Search 搜索栏中输入关键字即可搜索，也可以将 Workspace Search 加入到浏览器搜索引擎中，用户在浏览器地址栏输入关键字即可搜索，搜索体验就像在浏览器中使用 Google 或者百度一样。



用户很容易查看可搜索的内容源、最新内容，也可以按日期检索内容。



用户可以按自己的需求，在搜索时设置搜索字段、结果字段、字段权重、字段值权重、过滤、排序、分页、构面、高亮显示等。

如下代码设置返回第一页，每页一条内容：

```
curl -X POST http://localhost:3002/api/ws/v1/search \
-H "Authorization: Bearer $ACCESS_TOKEN" \
-H "Content-Type: application/json" \
-d '{
"query": "denali",
"page": {
"size": 1,
"current": 1
```

```
}
}'
```

如下代码设置按 square_km 逆序，date_established 顺序排序：

```
curl -X POST http://localhost:3002/api/ws/v1/search \
-H "Authorization: Bearer $ACCESS_TOKEN" \
-H "Content-Type: application/json" \
-d '{
"query": "denali",
"sort": [
{ "square_km": "desc" },
{ "date_established": "asc" }
]
}'
```

如下代码设置在字段 Title 和 Description 中搜素，Title 的权重为 10:

```
curl -X POST http://localhost:3002/api/ws/v1/search \
-H "Authorization: Bearer $ACCESS_TOKEN" \
-H "Content-Type: application/json" \
-d '{
"query": "denali",
"search_fields": {
"title": {
"weight": 10
},
"description": {}
```

```
}
}'
```

如下代码根据字段 world_heritage_site 的值设置权重，当字段值为 true 时权重为 10:

```
curl -X GET 'https://[instance id].ent-search.[region].[provider].cloud.es.io/api/as/v1/engines/national-parks-demo/search' \
-H 'Content-Type: application/json' \
-H 'Authorization: Bearer search-soaewu2ye6uc45dr8mcd54v8' \
-d '{
"query": "old growth",
"boosts": {
    "world_heritage_site": [
    {
        "type": "value",
        "value": "true",
        "operation": "multiply",
        "factor": 10
    }
    ]
}
}
```

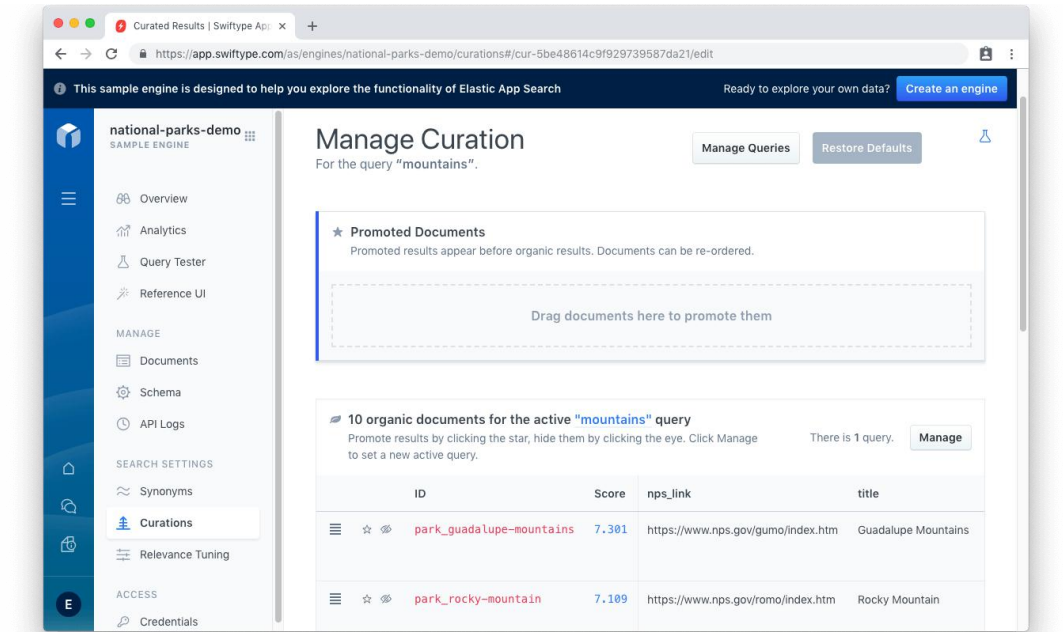
App Search 支持对查询进行加 tag 并按 tag 进行过滤查询：

```
curl -X GET 'https://[instance id].ent-search.[region].[provider].cloud.es.io/api/as/v1/engines/national-parks-demo/search' \
-H 'Content-Type: application/json' \
-H 'Authorization: Bearer search-soaewu2ye6uc45dr8mcd54v8' \
-d '{
"query": "everglade",
"analytics": {
    "tags": [
    "i-am-a-tag"
    ]
}
}'
```

```
curl -X GET 'https://[instance id].ent-search.[region].[provider].cloud.es.io/api/as/v1/engines/national-parks-demo/analytics/queries' \
-H 'Content-Type: application/json' \
-H 'Authorization: Bearer private-namt1hkv7ttsawuo452sxi6s' \
-d '{
"filters": { "tag": "i-am-a-tag" }
}'
```

支持对查询结果的控制

App Search 支持对查询结果直接进行控制，如下图，点击星号或者直接拖动结果可重新对查询结果进行排序，点击眼睛图标可隐藏查询结果。当然，所有操作也都是可以使用 API 进行设定，详情见 Curations API。



支持查询优化

App Search 创建引擎时可选择语言，App Search 会针对不同的语言自动进行优化，优化内容包括词干匹配、字符匹配、短语匹配、排版容忍度等。App Search 支持查询关键字推荐/自动完成功能。当用户输入部分关键字时，App Search 可根据引擎中已有的数据，推荐关键词，用户通过选用更合适的关键字，获取更精准的搜索结果。如下代码根据文档中 Title 和 States 字段内容提供 Car 关键词的推荐关键词：

```
curl -X POST 'https://[instance id].ent-search.[region].[provider].cloud.es.io/api/as/v1/engines/national-parks-demo/query_suggestion' \
-H 'Content-Type: application/json' \
-H 'Authorization: Bearer search-7eud55t7ecdmqzcanjsc9cqu' \
-d '{
"query": "car",
```

```
"types": {
      "documents": {
            "fields": [
            "title",
            "states"
            ]
      }
},
"size": 3
}'
```

结果提供 3 个推荐关键词为：carlsbad、carlsbad caverns、carolina

```
{
    "results":{
        "documents":[
            {
                "suggestion":"carlsbad"
            },
            {
                "suggestion":"carlsbad caverns"
            },
            {
                "suggestion":"carolina"
            }
        ]
    },
    "meta":{
```

```
        "request_id":"914f909793379ed5af9379b4401f19be"
    }
}
```

App Search 支持同义词配置，可通过设置同义词，使用同义词查询得到需要的结果。如下代码将 summit、peak、cliff、moutain 设置为同义词：

```
curl -X POST 'https://[instance id].ent-search.[region].[provider].cloud.es.io/api/as/v1/engines/national-parks-demo/synonyms' \
-H 'Content-Type: application/json' \
-H 'Authorization: Bearer private-xxxxxxxxxxxxxxxxxxxx' \
-d '{
"synonyms": ["summit", "peak", "cliff", "mountain"]
}
```

Enterprise Search 支持记录查询分析日志 API Log，方便对用户搜索情况，如搜索结果、搜素性能、搜索异常等进行分析和优化，不断改善用户搜素体验，形成正向反馈。

支持代码集成

Enterprise Search 提供了 Enterprise Search Python Client 和 Enterprise Search Ruby Client，可比较方便的使用 Python 和 Ruby 代码对 Enterprise Search 进行集成。

提供搜索框 UI

Elastic 企业搜索提供了连接 App Search 的 React 用户搜索交互界面，直接下载导入即可使用，省去了不少前端代码工作量，对有搜索框需求，但无特别要求的前端应用来说也是一个不错的选择。Site Search 则只需要用户在自己的网站上实施几行代码即可添加好由 Elasticsearch 提供支持的搜索框。

总结

企业搜索的业务场景决定了企业搜索的特点和需求，Elastic 在 Elasticsearch 强大功能的基础之上，构建了更加易用的企业搜索解决方案 Elastic Enterprise Search。Elastic Enterprise Search 针对企业搜索场景，提供了从自身部署到权限控制、从文档接入到查询优化、从前端 UI 到结果控制的全场景覆盖的支持能力，虽然其相比自己构建一套企业搜索系统的门槛已非常低，易用性也非常好，但毕竟是一套接口完善、功能众多、相对复杂的系统。以上内容仅简单介绍其基本能力，如需将其应用于生产环境，还需结合实际业务需求，仔细阅读相关文档并进行深入研究和实践。

注：本文中代码片段及配图来自 https://www.elastic.co/

参考链接：

- Enterprise Search Guide [7.12]
- Workplace Search Guide [7.12]
- App Search Guide [7.12].

创作人简介：

朱永生，现任上海闪马智能科技有限公司运维总监，拥有超过十五年技术运维经验，曾服务于多家上市公司和初创企业，负责网络游戏、广告、在线教育、金融科技等业务的全线运维工作和运维团队，擅长数据库和运维系统架构优化、运维自动化平台建设及 DevOps 相关的文化理念流程和技术系统。

博客：https://blog.csdn.net/yongsheng0550

3.2.2 可观测性

创作人：亢伟楠

审稿人：曾红

在开发技术越来越成熟便捷的今天，我们可以很轻松写出来一个程序，用来进行各种各样的业务流程。你能想象如果我们运营一个银行系统，但是不知道每天转账的成功率、取现的效率吗？

和银行系统一样，我们日常中的软件系统，都需要尽量良好的观测和测量，才能保证系统的健康。

正如管理大师彼得德鲁克的名言，”If you can't measure it,you can't manage it.” 我们必须对我们的计算系统进行测量和观测，才能进一步管理它。

业界对可观测性的定义由 Logging（日志），Metrics（指标）和 Tracing（跟踪）组成。其中大多数软件都仅在一个领域内发力，这导致了实施可观测性时的高昂成本。需要建设多个技术栈的软件，才能实现完整的可观测性。大多数企业基本都使用了 5 个 + 的技术栈，有的甚至能达到 10 个技术栈。

那有没有什么低成本便捷的方案能帮助我们在企业中实施可观测性？

Elastic 可观测性 ： 一站式低成本解决方案

Elastic Stack 的可观测性 (Observability) 产品是一个让人满意的答案。相较于市面上其他的可观测性系统，Elastic Stack 能提供一站式全栈的可观测性解决方案，而其他系统基本只能提供一个方法的功能，实际落地中，需要搭配多套不同技术栈的系统实现，繁琐且复杂。

Elastic Stack 提供开源的可观测性能力，并且在云原生计算基金会（CNCF）的 2020 年 9 月的可观测性技术雷达评测中，获得了“采纳（ADOPT）”评级。

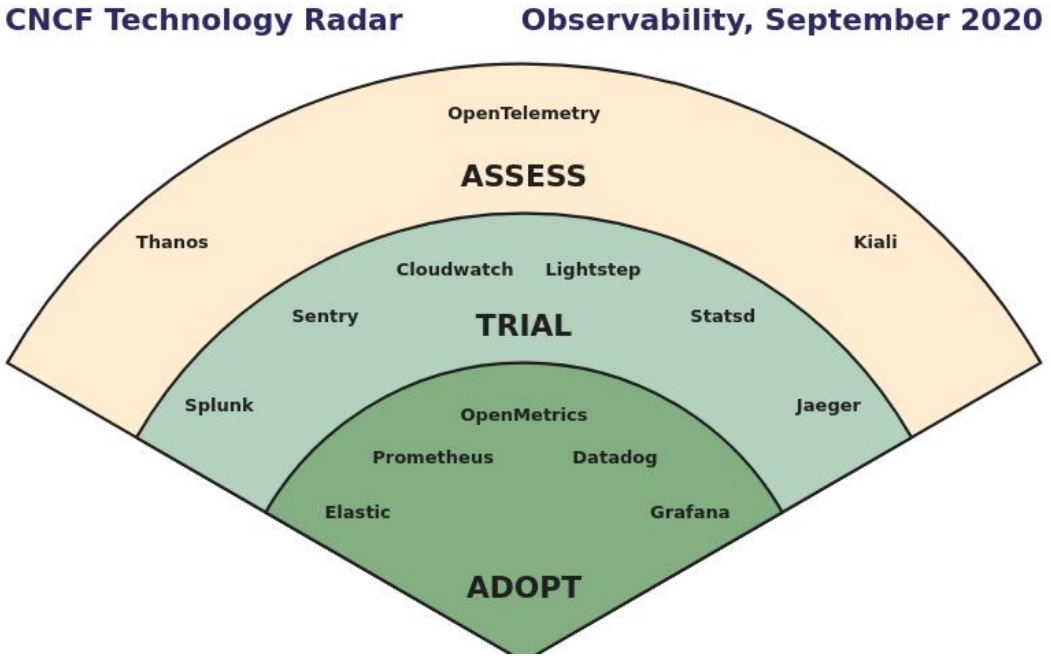


图 1 可观测性技术雷达

Elastic Stack 的可观测性由 Logs 、APM（application performance monitor）、Uptime、Metrics 四个模块组成，他们分别由四个组件提供支持。

组件对应情况如下:

- Logs -- Filebeat
- APM -- APM Server & APM agent
- Uptime -- Heartbeat
- Metrics -- Metricbeat

Logs

Kibana 中的 Logs 应用实现任何数据源日志的中心化，搜索。Elastic Stack 是天然的日志处理的集大成者。最新的 Logs 模块更是能实现在 web 的实时 tail、搜索、分类和异常检测功能。我们可以通过 Filebeat 或者 Logstash 把日志导入到 Elasticsearch 中。

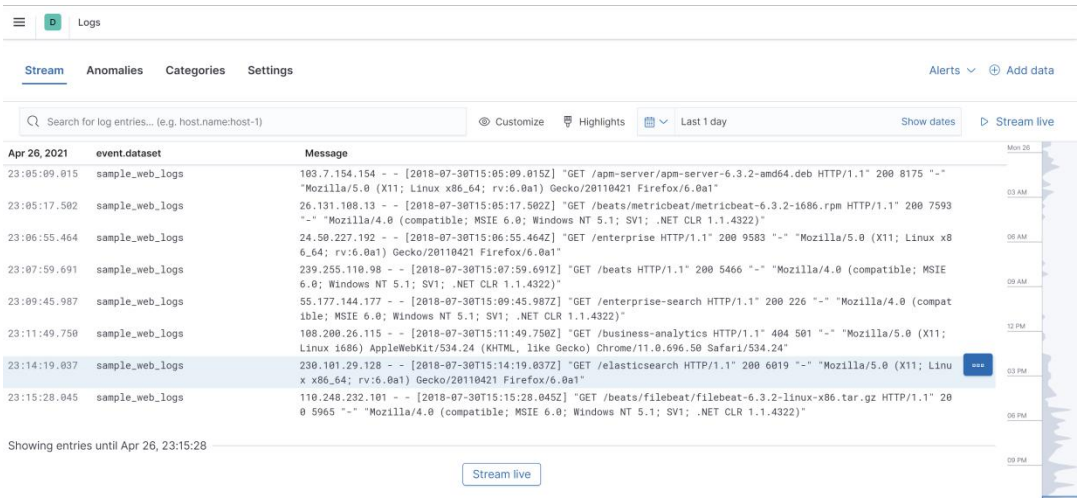


图 2 Logs 页面

APM（Application Performance Monitor）

Kibana 中的 APM 应用可以实现分布式链路跟踪、事务监控、依赖分析和基于真实用户体验的监控。该功能通过 APM Server 和 APM Agent 组件提供支持。

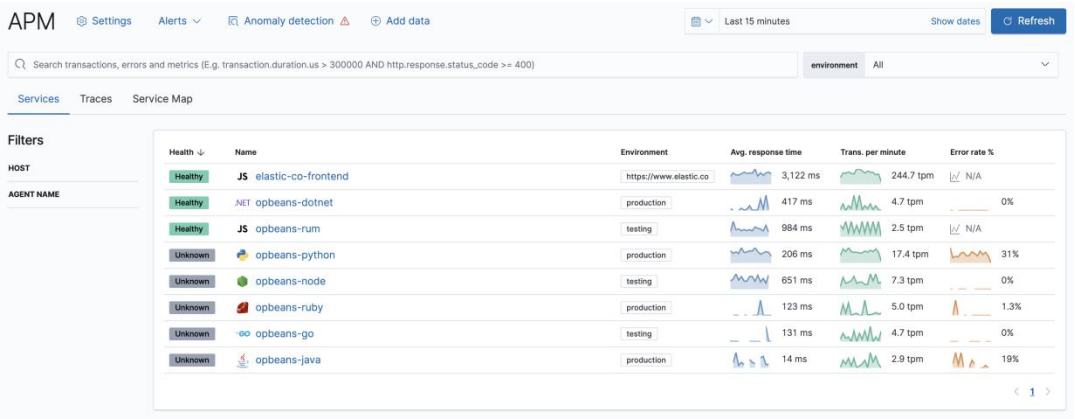


图 3 APM 页面

Uptime

为了帮助你在可用性问题，从而影响用户之前快速做出相应。Uptime 模块提供了对主机，网络设备以及第三方服务的整体可用性快照报告。根据其监控数据，可以查看出目前的监控点的 UP\DOWN.状态的监控点。该功能通过 Heartbeat 组件提供支持。

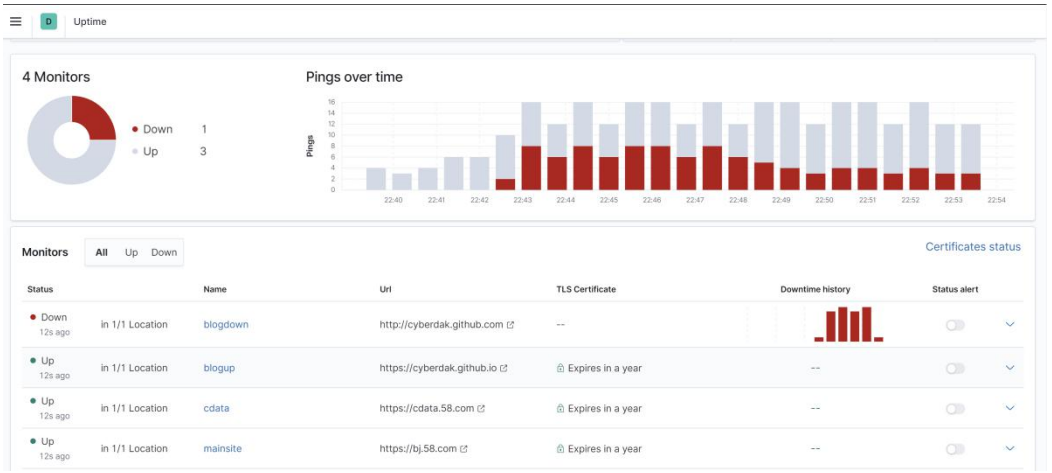


图 4 Uptime 页面

Metrics

Metricbeats 支持读取多大 50 种系统/数据源的 Metrics 采集，包括：数据库，消息队列，操作系统，文件系统和网关等系统。将 Metrics 数据采集完毕后，在 Metrics 应用中可实现一站式的统一查看、管理，甚至是基于机器学习的异常检测。

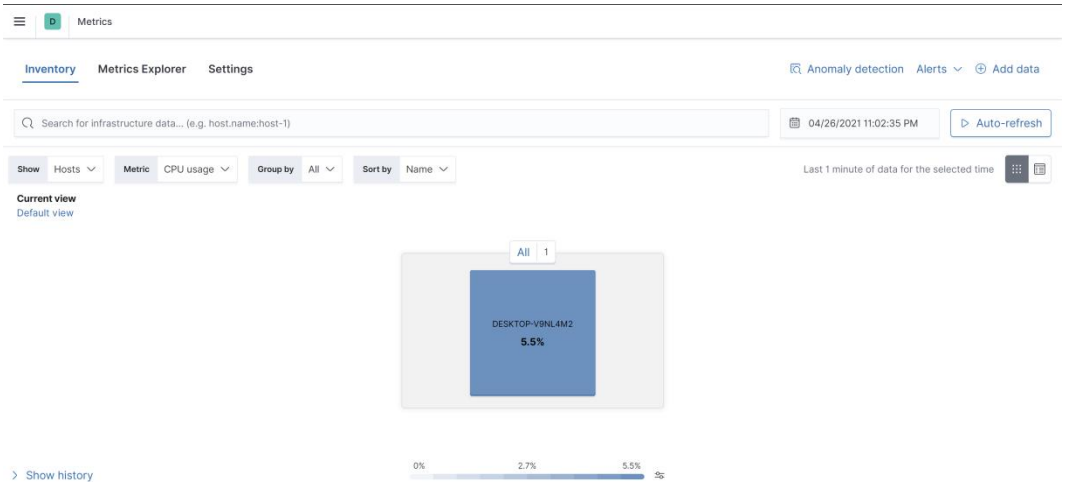


图 5 Metrics 页面

以上四个模块均支持告警功能。

Fleet：更方便的数据收集

Fleet Agent 提供了更简单方便的统一的 Logs 数据、Metrics 数据和主机的其他数据。不在需要安装多个 Beat 来实现对数据的收集。

Fleet 目前处于 Beta 阶段。

参考链接：

- CNCF End User Technology Radar https://radar.cncf.io/2020-09-observability

创作人简介：

亢伟楠，目前就职于 58 同城信息安全部，任架构师。 在 Elastic 社区任日报编辑，曾获得中文社区杰出贡献者奖项。平时对高并发、高可用等方向有较多关注，目前主要推动可观测性落地。

博客：https://cyberdak.github.io/

3.3 基础篇

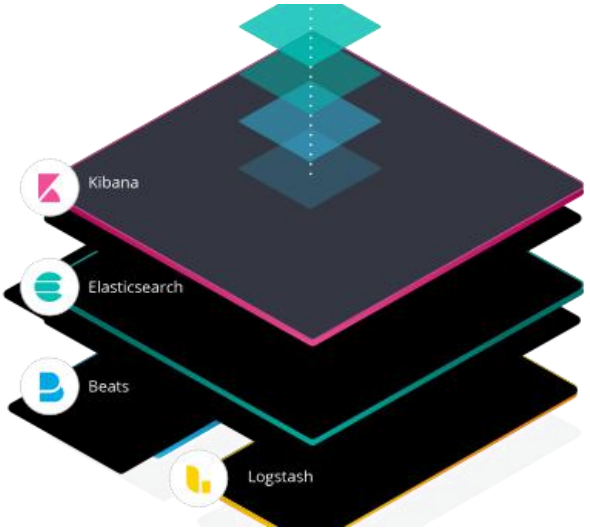
3.3.1 Elastic Stack 家族

创作人：左卫东

审稿人：田雪松

Elastic Stack 完整版图

Elastic Stack 是一系列由 Elastic 公司开发的产品组件，能够安全可靠地获取任何来源、任何格式的数据，然后实时地对数据进行搜索、分析和可视化。Elastic Stack 旧称 ELK Stack，主要有 Elasticsearch，Logstash，Kibana，Beats 四种组件组成。



Elastic Stack 功能丰富，他可以用来集中采集各类日志记录，这在识别 Web 服务器与应用程序相关的问题中起着重要作用。它使你可以在单个位置搜索所有日志，并通过在 IT 环境中找到的特定时间范围内，关联它们的日志来确定跨多个服务器的问题，这些特定时间范围包括 Web 分析，业务智能，合规性和安全性的用例。同样的，Elastic Stack 也可以将你海量的日志、指标和 APM 追踪集中到单一技术栈中，从而即时全面地监测环境中所发生的所有事件，并采取对策。

Elastic Stack 组件主要优点如下：

- **配置简单，开箱即用**

  上手简单，只需要修改几行配置，就能快速搭建起一套 Elastic Stack 平台。

- **多种数据源支持**

  Beats 组件支持多种数据类型，能够快速满足日志，指标，网络数据等多种数据源接入的需求。

- **性能优异**

  无论是数据写入还是实时检索，性能都相当优异。

- **集群扩展性强**

  Elasticsearch 和 Kibana 均可以进行灵活扩展，能够有效提高集群性能和高可用性。

- **多维度分析**

得益于 Elasticsearch 强大的搜索能力和 Kibana 优秀的可视化功能，能够从多个维度，对数据进行聚合分析并且展示。

Elastic Stack 应用场景

Elastic Stack 除了采集提供采集各类日志的能力外，它还提供了许多开箱即用的场景，如企业搜索，可观测性，安全解决方案等。

企业搜索

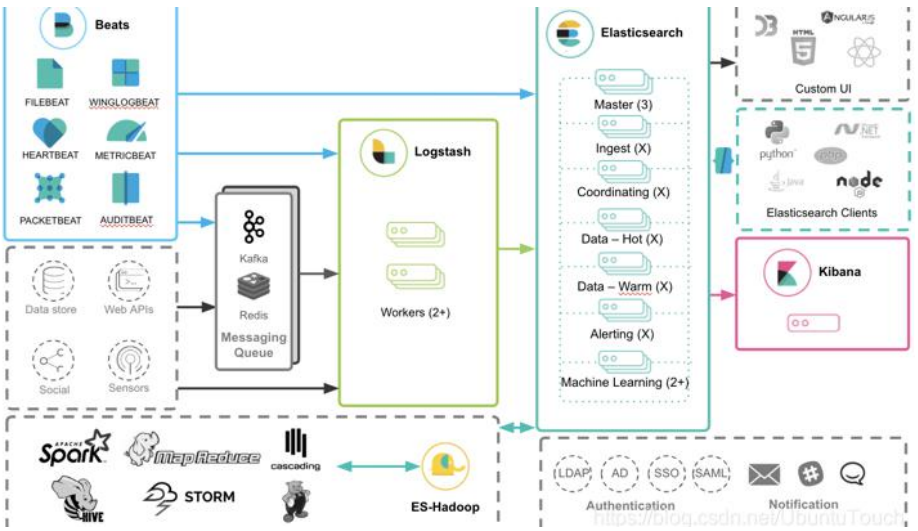
Elastic 企业搜索由 Elasticsearch 提供支持，性能优异，同时其采用的相关性模型已针对实际的自然搜索进行优化且得到了实践验证，因此能够快速投入应用。同时它又提供了灵活的定制选项，可以让客户快速根据自身需要打造精致又自然的搜索体验。

可观测性

Elastic Stack 将你的可观测性数据统一到一个强大的数据存储中，便于你实时搜索并应用交互式分析。凭借日志、指标和 APM 追踪之间的直观导航，便能依赖 Machine Learning 暴露异常数值，并对你系统中发生的所有事件采取对策。同时提供良好的可视化体验，能够以最直观的方式展示你的数据，完美掌握系统运行状态。

安全解决方案

Elastic 安全助力分析师防御、检测威胁，并就威胁做出响应。免费、开放的解决方案提供了 SIEM、Endpoint Security、威胁搜寻、云监测等功能。通过整个环境无死角的恶意软件和勒索软件防御体系，避免破坏和损失。同时为从业人员提供直观的 UI ，简化事件管理。利用可视化功能进行监测和搜寻，描绘出攻击的来源、程度和时间线—通过分析师驱动型的关联性将信息转变为见解。通过内置式案例管理和自动化操作加快响应速度。



Elasticsearch 在 Elastic Stack 中的位置及能力

Elasticsearch 是一个分布式的免费及开放搜索和分析引擎，适用于所有类型的数据，包括文本、数字、地理空间、结构化和非结构化数据。Elasticsearch 在 Apache Lucene 的基础上开发而成，以其简单的 RESTful API、相关性搜索，高性能和高可扩展性而闻名，是 Elastic Stack 的核心组件。

Elasticsearch 在 Elastic Stack 中的作用

在 Elastic Stack 体系中，Elasticsearch 提供数据的存储及检索能力，并且它是最核心的组件。外部数据采集到 Elasticsearch 后，用户便可针对他们的数据运行复杂的查询，并使用聚合来检索自身数据的复杂汇总。

Elasticsearch Index

一个 Elasticsearch Index 指相互关联的文档集合。Elasticsearch 以 JSON 文档的形式存储数据。每个文档都会在一组键（字段或属性的名称）和它们对应的值（字符串、数字、布尔值、日期、数组自值、地理位置或其他类型的数据）之间建立联系。

Elasticsearch 使用的是一种名为倒排索引的数据结构，这一结构的设计可以允许十分快速地进行全文本搜索。倒排索引会列出在所有文档中出现的每个特有词汇，并且可以找到包含每个词汇的全部文档。

在创建索引的过程中，Elasticsearch 会存储文档并构建倒排索引，这样用户便可以

近实时地对文档数据进行搜索。索引过程是在索引 API 中启动的，通过此 API 你既可向特定索引中添加 JSON 文档，也可更改特定索引中的 JSON 文档。

Elasticsearch 能力

Elasticsearch 很快

由于 Elasticsearch 是在 Lucene 基础上构建而成的，所以在全文本搜索方面表现十分出色。Elasticsearch 同时还是一个近实时的搜索平台，这意味着从文档索引操作到文档变为可搜索状态之间的延时很短，一般只有一秒（这个可以同过配置进行调整）。因此，Elasticsearch 非常适用于对时间有严苛要求的用例，例如安全分析和基础设施监测。

Elasticsearch 具有分布式的本质特征

Elasticsearch 中存储的文档分布在不同的容器中，这些容器称为分片，可以进行复制以提供数据冗余副本，以防发生硬件故障。Elasticsearch 的分布式特性使得它可以扩展至数百台（甚至数千台）服务器，并处理 PB 量级的数据。

“Elasticsearch 优异的相关性检索能力

Elasticsearch 底层采用倒排索引的数据结构，检索过程中采用优异的相关性计算算法，因此 Elasticsearch 具有优异的相关性检索能力，它能基于多个维度多搜索结果进行评分，因此它可以与业务需求进行结合，达到满足用户需求的搜索结果。

Elasticsearch 包含一系列广泛的功能

除了速度、可扩展性和弹性等优势以外，Elasticsearch 还有大量强大的内置功能（例如数据汇总和索引生命周期管理），可以方便用户更加高效地存储和搜索数据。

Logstash 在 Elastic Stack 中的位置及能力

Logstash 是免费且开放的服务器端数据处理管道，能够从多个来源采集数据，转换数据，然后将数据发送到你最喜欢的“存储库”中。

Logstash 在 Elastic Stack 中的作用

Logstash 是 Elastic Stack 的核心产品之一，可用来对数据进行聚合和处理，并将数据发送到 Elasticsearch。Logstash 是一个开源的服务器端数据处理管道，允许你在将数据索引到 Elasticsearch 之前同时从多个来源采集数据，并对数据进行丰富和转换。

Logstash 能力

采集各种样式、大小和来源的数据

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 支持多种输入选择，可以同时从众多常用来源捕捉事件。能够以连续的流式传输方式，轻松地从你的日志、指标、Web 应用、数据存储以及各种 AWS 服务采集数据。

实时解析和转换数据

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，以便进行更强大的分析和实现商业价值。

Logstash 能够动态地转换和解析数据，不受格式或复杂度的影响，比如：

- 利用 Grok 从非结构化数据中派生出结构
- 从 IP 地址破译出地理坐标
- 将 PII （Personal Identifiable Information）数据匿名化，完全排除敏感字段
- 简化整体处理，不受数据源、格式或架构的影响

选择你的存储库，导出你的数据

尽管 Elasticsearch 是我们的首选输出方向，能够为我们的搜索和分析带来无限可能，但它并非唯一选择。

Logstash 提供多种输出组件，你可以将数据发送你要指定的地方，并且能够灵活地解锁众多下游用例。

Logstash 扩展性

Logstash 采用可插拔框架，拥有 200 多个插件。你可以将不同的输入选择、过滤器和输出选择混合搭配、精心安排，让它们在管道中和谐地运行。

Kibana 在 Elastic Stack 中的位置及能力

Kibana 是一款适用于 Elasticsearch 的数据可视化和管理工具，可以提供实时的直方图、线形图、饼状图和表格等等。Kibana 同时还包括诸如 Canvas 和 Elastic Maps 等高级应用程序。Canvas 允许用户基于自身数据创建定制的动态信息图表，而 ElasticMaps 则可用来对地理空间数据进行可视化。

Kibana 在 Elastic Stack 中的作用

Kibana 可以被视作 Elastic Stack（之前称作 ELK Stack，分别表示 Elasticsearch、Logstash 和 Kibana）的制图工具，但也可将 Kibana 作为用户界面来监测和管理 Elastic Stack 集群并确保集群安全性，还可将其作为基于 Elastic Stack 所开发内置解决方案的汇集中心。

Kibana 能力

Kibana 与 Elasticsearch 和更广意义上的 Elastic Stack 紧密集成，这一点使其成为支持下列场景的理想之选：

搜索、查看并可视化 Elasticsearch 中所索引的数据，并通过创建柱状图、饼状图、表格、直方图和地图对数据进行分析。仪表板视图能将这些可视化元素集中到一起，然后通过浏览器加以分享，以提供有关海量数据的实时分析视图，为下列用例提供支持：

- 日志处理和分析
- 基础设施指标和容器监测
- 应用程序性能监测 (APM)
- 地理空间数据分析和可视化
- 安全分析
- 业务分析
- 机器学习

借助网络界面来监测和管理 Elastic Stack 实例并确保实例的安全。

针对基于 Elastic Stack 开发的内置解决方案（面向可观测性、安全和企业搜索应用程序），将其访问权限集中到一起。

Kibana 搜索及可视化流程

Kibana 允许对 Elasticsearch 索引中的数据进行可视化分析。当 Logstash（大型采集器）或 Beats（一系列单一用途的数据采集器）从日志文件和其他来源采集非结构化数据并将这些数据转化为结构化格式以用于 Elasticsearch 存储和搜索功能时，索引便会随之创建。

用户通过 Kibana 界面能够查询 Elasticsearch 索引中的数据，然后借助标准图表选项或诸如 Canvas 和 Maps 等内置应用对结果进行可视化。用户可在不同图表类型之中进行选择，更改数字的聚合方式，还可筛选出特定的数据片段。

Beats 在 Elastic Stack 中的位置及能力

Beats 是一个轻量的数据采集器，它集合了多种单一用途数据采集器。它们从成百上千或成千上万台机器和系统向 Logstash 或 Elasticsearch 发送数据。

Beats 在 Elastic Stack 中的作用

Beats 在 Elastic stack 中是数据的采集器，Beats 可以从你的各种环境中收集日志，指标，安全数据或者网路数据，然后通过来自主机、诸如 Docker 和 Kubernetes 等容器平台以及云服务提供商的必要元数据对这些内容进行记录，然后再传输到 ElasticStack 中。

Beats 诞生的原因

ELK 在最初仅包含 Elasticsearch，Kibana，Logstash。在旧有的日志采集系统中，数据管道包含 3 个主要阶段，数据采集，数据处理和存储，其中的前两个阶段均由 Logstash 进行承担。然后由于 Logstash 的设计导致的内在问题，常常发生性能问题，尤其是在有复杂的管道处理流程中。因此，转移 Logstash 的想法也应用而生，因此将数据提取任务抽离之后，就诞生了 Beats。

Beats 优点

即插即用

Beats 能够简化从关键数据源（例如云平台、容器和系统，以及网络技术）采集、解析和可视化信息的过程。只需一行命令，就能完成数据的采集。

扩展项强

Beats 提供了大量不同类型的采集器，同时提供了自定义协议所需的构建基石，方便扩展，同时 Beats 社区在不断状态，未来将会诞生满足更多场景的 Beats。

轻量易部署

相较于 Logstash，Beats 体积更小，性能更高，能够快速接入不同的数据源进行采集。

Beats 系列

Beats 提供了多种类型的采集器，方便你即插即用，搞定大多数数据类型的采集。

Filebeats ：采集日志文件

Filebeat 随附可观测性和安全数据源模块，这些模块简化了常见格式的日志的收集、解析和可视化过程，只需一条命令即可。之所以能实现这一点，是因为它将自动默认路径（因操作系统而异）与 Elasticsearch 采集节点管道的定义和 Kibana 仪表板组合在一起。不仅如此，Filebeat 的一些模块还随附了预配置的 Machine Learning 作业。

Metricbeat ：采集指标

将 Metricbeat 部署到你的所有 Linux、Windows 和 Mac 主机，并将它连接到 Elasticsearch 就大功告成了：你可以获取系统级的 CPU 使用率、内存、文件系统、磁盘 IO 和网络 IO 统计数据，还可针对系统上的每个进程获得与 top 命令类似的统计数据。

Packetbeat： 采集网络数据

HTTP 等网络协议能够让你密切监测应用程序延迟和错误、响应时间、SLA 性能、用户访问模式和趋势等等。 而 Packetbeat 则让你能够访问这些数据，了解流量的网络传输状态。这款工具完全采用被动模式，毫无延迟开销，并且不会妨碍你的基础架构。

Winlogbeat ：采集 Windows 事件日志

用于密切监控基于 Windows 的基础设施上发生的事件。使用 Winlogbeat，将 Windows 事件日志流式传输至 Elasticsearch 和 Logstash。

Auditbeat ：采集审计数据

你可以使用既有审计规则来轻而易举地收集数据，而无需重写规则。是谁在什么时间做了什么事情？Auditbeat 会记住所有这些原始的系统调用数据，以及相关联的路径，方便你了解所需的上下文信息。

Heartbeat ：采集运行时间监控

无论你要测试同一台主机上的服务，还是要测试开放网络上的服务，Heartbeat 都能轻松生成运行时间数据和响应时间数据。

Functionbeat ：无需服务器的采集器

你能够通过无服务器架构部署代码，省去了启动和管理额外的底层软件和硬件的麻烦。通过 Functionbeat，你能够同样简单地监测云端基础架构。

Journalbeat ： journald 日志采集器

开源社区一直在努力开发新的 Beats。

你可以通过链接查看其中的一些社区开发的 Beats：https://www.elastic.co/guide/en/beats/libbeat/current/community-beats.html

创作人简介：

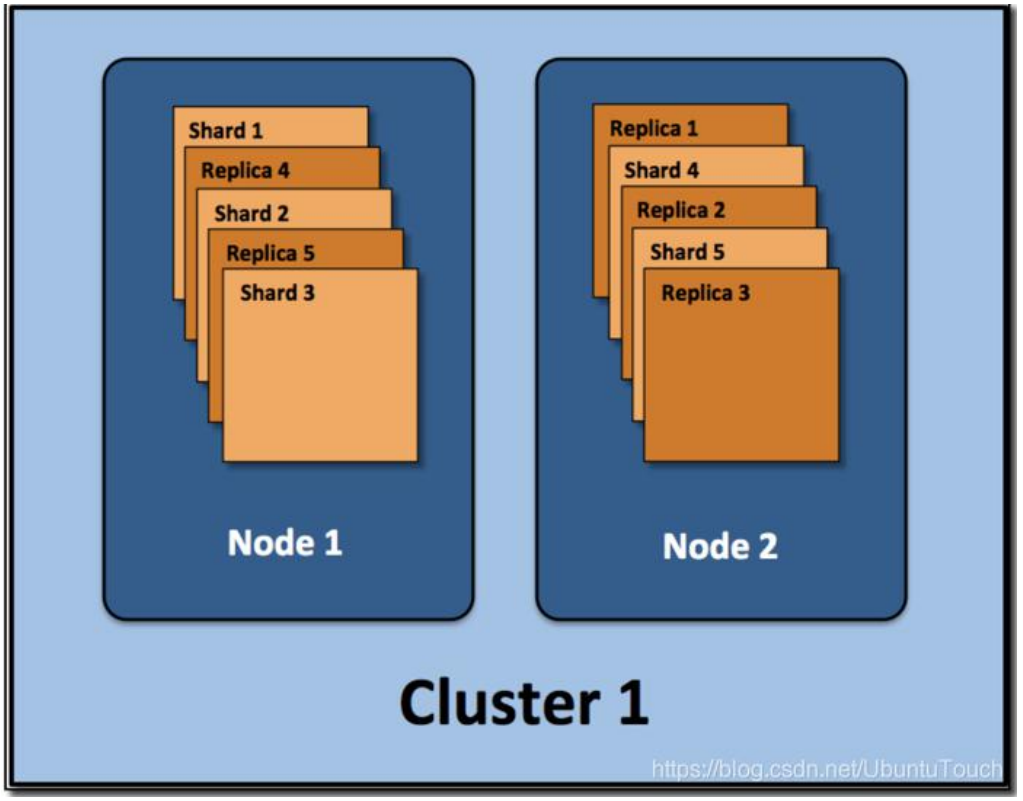
左卫东，Elasticsearch 认证工程师

3.3.2 专有名词解释

创作人：刘晓国

当我们开始使用 Elasticsearch 时，我们必须理解其中的一些重要的概念。这些概念的理解对于以后我们使用 Elastic Stack 是非常重要的。在今天的这篇文章里，我们先来介绍一下在 Elastic Stack 中最重要的一些概念。

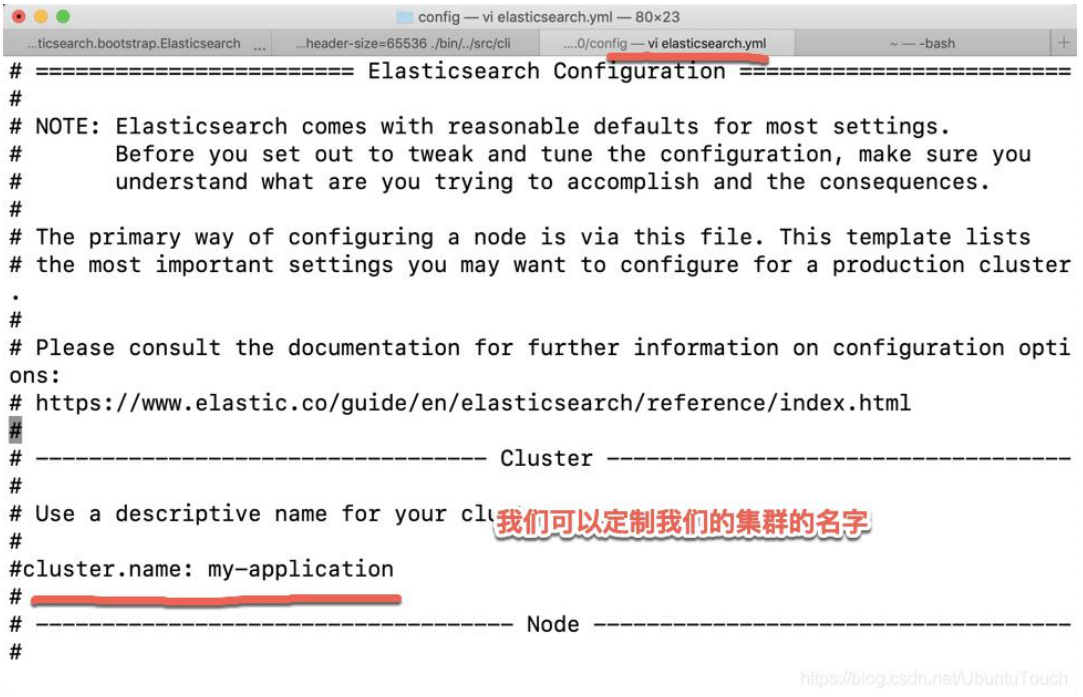
首先，我们来看下一下如下的这个图：



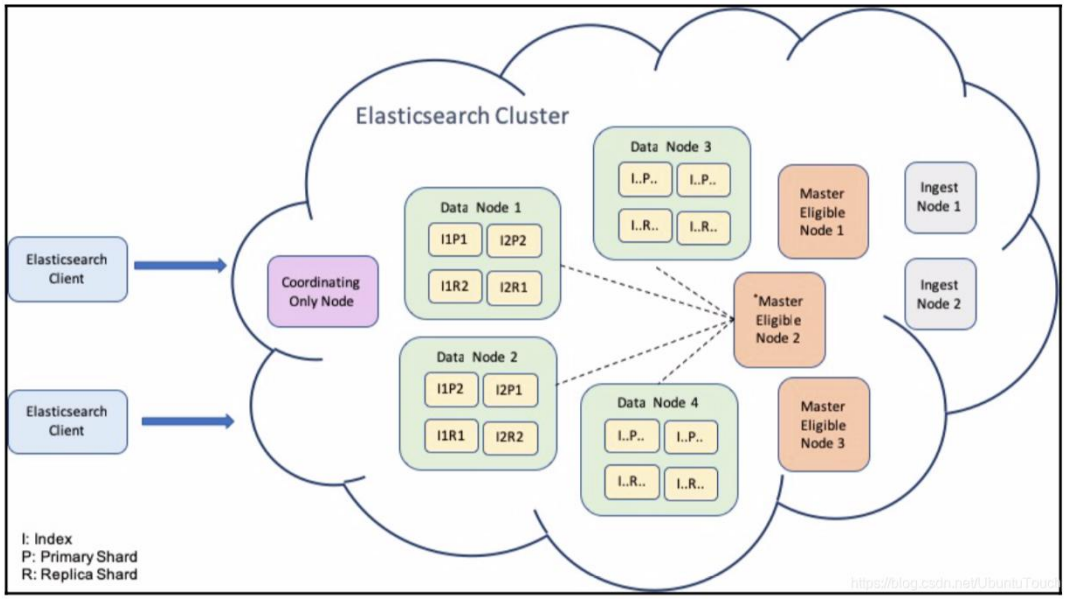
Cluster

Cluster 也就是集群的意思。

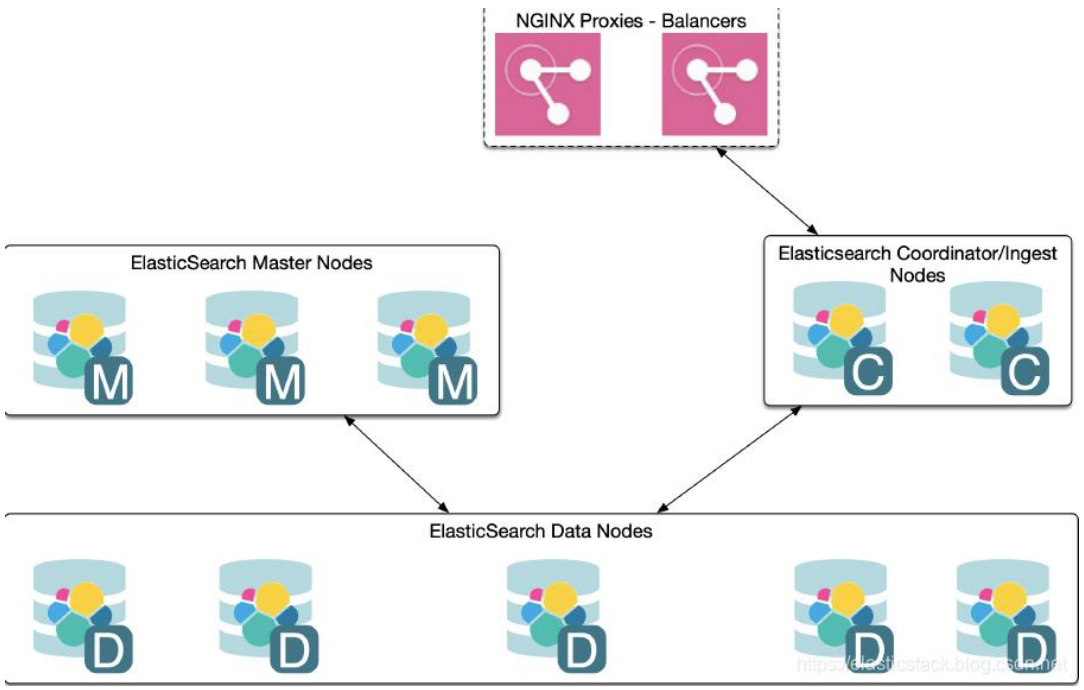
Elasticsearch 集群由一个或多个节点组成，可通过其集群名称进行标识。通常这个 Cluster 的名字是可以在 Elasticsearch 里的配置文件中设置的。在默认的情况下，如我们的 Elasticsearch 已经开始运行，那么它会自动生成一个叫做 “Elasticsearch” 的集群。我们可以在 config/elasticsearch.yml 里定制我们的集群的名字：



一个 Elasticsearch 的集群就像是下面的一个布局：



带有 NginX 代理及 Balancer 的架构图是这样的：



我们可以通过：

```
GET _cluster/state
```

来获取整个 cluster 的状态。这个状态只能被 master node 所改变。上面的接口返回的结果是：

```
{
  "cluster_name": "elasticsearch",
  "compressed_size_in_bytes": 1920,
  "version": 10,
  "state_uuid": "rPiUZXbURICvkPl8GxQXUA",
  "master_node": "O4cNlHDuTyWdDhq7vhJE7g",
  "blocks": {},
  "nodes": {...},
  "metadata": {...},
  "routing_table": {...},
  "routing_nodes": {...},
  "snapshots": {...},
  "restore": {...},
  "snapshot_deletions": {...}
}
```

Node

单个 Elasticsearch 实例。

在大多数环境中，每个节点都在单独的盒子或虚拟机上运行。一个集群由一个或多个 node 组成。在测试的环境中，我可以把多个 node 运行在一个 server 上。在实际的部署中，大多数情况还是需要一个 server 上运行一个 node。

根据 node 的作用，可以分为如下的几种：

- master-eligible：可以作为主 node。一旦成为主 node，它可以管理整个 cluster 的设置及变化：创建，更新，删除 index；添加或删除 node；为 node 分配 shard
- data：数据 node
- ingest: 数据接入（比如 pipepline）
- machine learning (Gold/Platinum License)

一般来说，一个 node 可以具有上面的一种或几种功能。我们可以在命令行或者 Elasticsearch 的配置文件（Elasticsearch.yml）来定义：

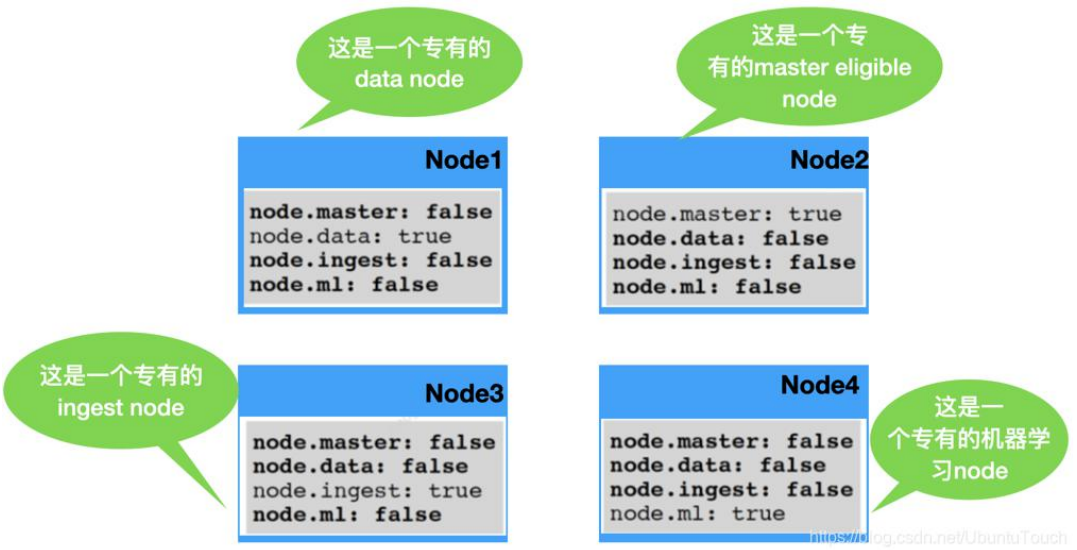
Node 类型	配置参数	默认值
master-eligible	node.master	true
data	node.data	true
ingest	node.ingest	true
machine learning	node.ml	true（除了 OSS 发布版）

你也可以让一个 node 做专有的功能及角色。如果上面 node 配置参数没有任何配置，那么我们可以认为这个 node 是作为一个 coordination node。在这种情况下，

它可以接受外部的请求，并转发到相应的节点来处理。针对 master node，有时我们需要设置 cluster.remote.connect: false。

在实际的使用中，我们可以把请求发送给 data 节点，而不能发送给 master 节点。

我们可以通过对 config/elasticsearch.yml 文件中配置来定义一个 node 在集群中的角色：



在有些情况中，我们可以通过设置 node.voting_only 为 true 从而使得一个 node 在 node.master 为真的情况下，只作为参加 voting 的功能，而不当选为 master node。这种情况为了避免脑裂情况发生。它通常可以使用一个 CPU 性能较低的 node 来担当。

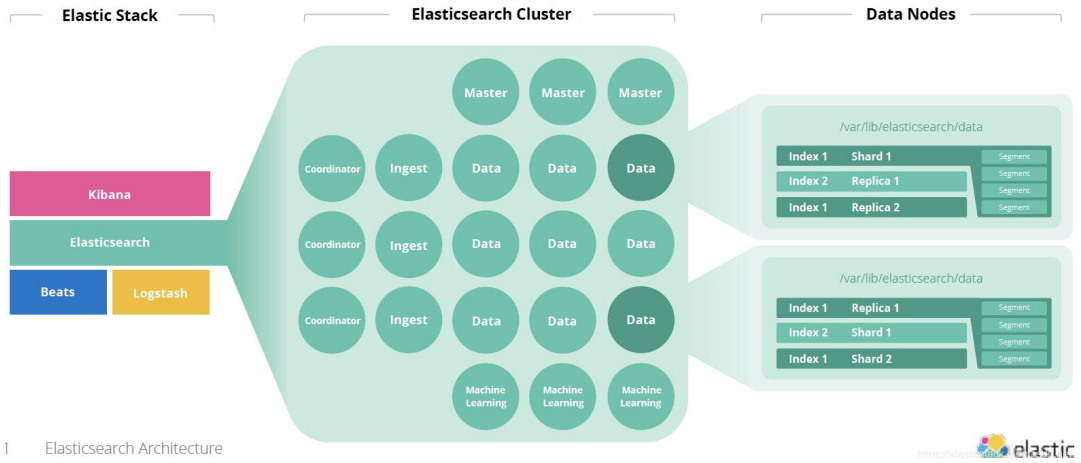
在一个集群中，我们可以使用如下的一个命令来获取当前可以进行 vote 的所有 master-eligible 节点：

```
GET /_cluster/state?filter_path=metadata.cluster_coordination.last_committed_config
```

你可能获得类似如下列表的结果：

```
{
  "metadata" : {
    "cluster_coordination" : {
      "last_committed_config" : [
        "Xe6KFUYCTA6AWRpbw84qaQ",
        "OvD79L1lQme1hi06Ouiu7Q",
        "e6KF9L1lQUYbw84CTAemQl"
      ]
    }
  }
}
```

在整个 Elastic 的架构中，Data Node 和 Cluster 的关系表述如下：



1 Elasticsearch Architecture

上面的定义适用于 Elastic Stack 7.9 发布版以前。在 Elastic Stack 7.9 之后，有了新的改进。

请详细阅读文章 “Elasticsearch：Node roles 介绍 - 7.9 之后版本”

Document

Elasticsearch 是面向文档的，这意味着你索引或搜索的最小数据单元是文档。

文档在 Elasticsearch 中有一些重要的属性：

- 它是独立的。文档包含字段（名称）及其值。
- 它可以是分层的。可以将其视为文档中的文档。字段的值可以很简单，就像位置字段的值可以是字符串一样。它还可以包含其他字段和值。例如，位置字段可能包含城市和街道地址。

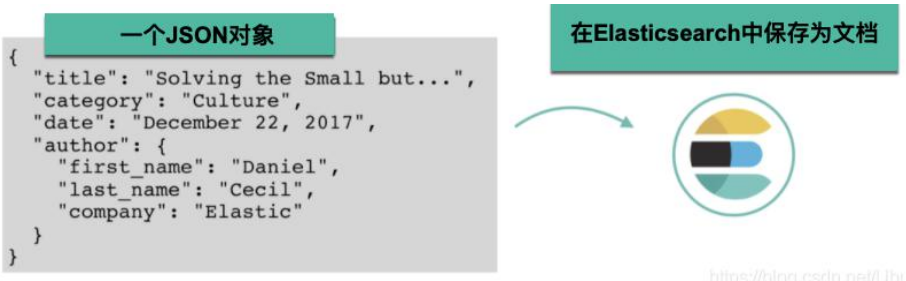
- 结构灵活。你的文档不依赖于预定义的架构。例如，并非所有事件都需要描述值，因此可以完全省略该字段。但它可能需要新的字段，例如位置的纬度和经度。

文档通常是数据的 JSON 表示形式。JSON over HTTP 是与 Elasticsearch 进行通信的最广泛使用的方式，它是我们在本书中使用的方法。

例如，你的聚会网站中的事件可以在以下文档中表示：

```
{
  "name": "Elasticsearch Denver",
  "organizer": "Lee",
  "location": "Denver, Colorado, USA"
}
```

很多人认为 Document 相比较于关系数据库，它相应于其中每个 record。



Type

类型是文档的逻辑容器，类似于表是行的容器。

你将具有不同结构（模式）的文档放在不同类型中。 例如，你可以使用一种类型来定义聚合组，并在人们聚集时为事件定义另一种类型。

每种类型的字段定义称为映射。 例如，name 将映射为字符串，但 location 下的 geolocation 字段将映射为特殊的 geo_point 类型。 每种字段的处理方式都不同。例如，你在名称字段中搜索单词，然后按位置搜索组以查找位于你居住地附近的组。

很多人认为 Elasticsearch 是 schema-less 的。大家都甚至认为 Elasticsearch 中的数据库是不需要 mapping 的。其实这是一个错误的概念。schema-less 在 Elasticsearch 中正确的理解是，我们不需要事先定义一个类型关系数据库中的 table 才使用数据库。

在 Elasticsearch 中，我们开始可以不定义一个 mapping，而直接写入到我们指定的 index 中。这个 index 的 mapping 是动态生成的 （当然我们也可以禁止这种行为）。其中的数据项的每一个数据类型是动态识别的。比如时间，字符串等，虽然有些数据类型，还是需要我们手动调整，比如 geo_point 等地理位置数据。

另外，它还有一个含义，同一个 type，我们在以后的数据输入中，可能增加新的数据项，从而生产新的 mapping。这个也是动态调整的。

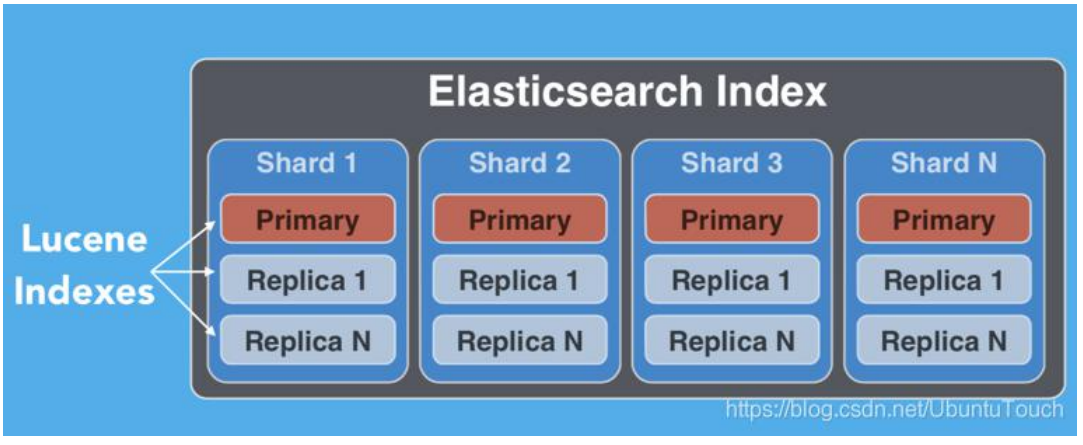
Elasticsearch 具有 schema-less 的能力，这意味着无需显式指定如何处理文档中可能出现的每个不同字段，即可对文档建立索引。 启用动态映射后，Elasticsearch 自动检测并向索引添加新字段。 这种默认行为使索引和浏览数据变得容易-只需开始建立

索引文档，Elasticsearch 就会检测布尔值，浮点数和整数值，日期和字符串，并将其映射到适当的 Elasticsearch 数据类型。

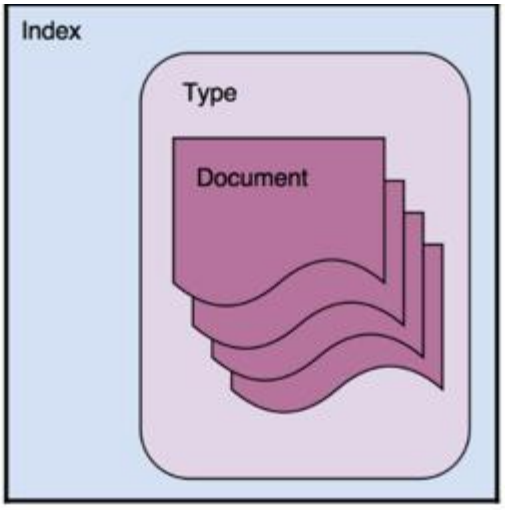
由于一些原因，在 Elasticsearch 6.0 以后，一个 Index 只能含有一个 type。这其中的原因是：相同 index 的不同映射 type 中具有相同名称的字段是相同； 在 Elasticsearch 索引中，不同映射 type 中具有相同名称的字段在 Lucene 中被同一个字段支持。在默认的情况下是 _doc。在未来 8.0 的版本中，type 将被彻底删除。

Index

在 Elasticsearch 中，索引是文档的集合。



每个 Index 一个或许多的 documents 组成，并且这些 document 可以分布于不同的 shard 之中。



Organization of Index, Type, and Document

很多人认为 index 类似于关系数据库中的 database。这中说法是有些道理，但是并不完全相同。其中很重要的一个原因是，在 Elasticsearch 中的文档可以有 object 及 nested 结构。一个 index 是一个逻辑命名空间，它映射到一个或多个主分片，并且可以具有零个或多个副本分片。

每当一个文档进来后，根据文档的 id 会自动进行 hash 计算，并存放于计算出来的 shard 实例中，这样的结果可以使得所有的 shard 都比较有均衡的存储，而不至于有的 shard 很忙。

```
shard_num = hash(_routing) % num_primary_shards
```

在默认的情况下，上面的 _routing 既是文档的 _id。如果有 routing 的参与，那么这些文档可能只存放于一个特定的 shard，这样的好处是对于一些情况，我们可以很快地综合我们所需要的结果而不需要跨 node 去得到请求。比如针对 join 的数据类型。

从上面的公式我们也可以看出来，我们的 shard 数目是不可以动态修改的，否则之后也找不到相应的 shard 号码了。必须指出的是，replica 的数目是可以动态修改的。

Shard

由于 Elasticsearch 是一个分布式搜索引擎，因此索引通常会拆分为分布在多个节点上的称为分片的元素。 Elasticsearch 自动管理这些分片的排列。 它还根据需要重新平衡分片，因此用户无需担心细节。

一个索引可以存储超出单个结点硬件限制的大量数据。比如，一个具有 10 亿文档的索引占据 1TB 的磁盘空间，而任一节点都没有这样大的磁盘空间；或者单个节点处理搜索请求，响应太慢。

为了解决这个问题，Elasticsearch 提供了将索引划分成多份的能力，这些份就叫做分片（shard）。当你创建一个索引的时候，你可以指定你想要的分片（shard）的数量。每个分片本身也是一个功能完善并且独立的“索引”，这个“索引”可以被放置到集群中的任何节点上。

分片之所以重要，主要有两方面的原因：

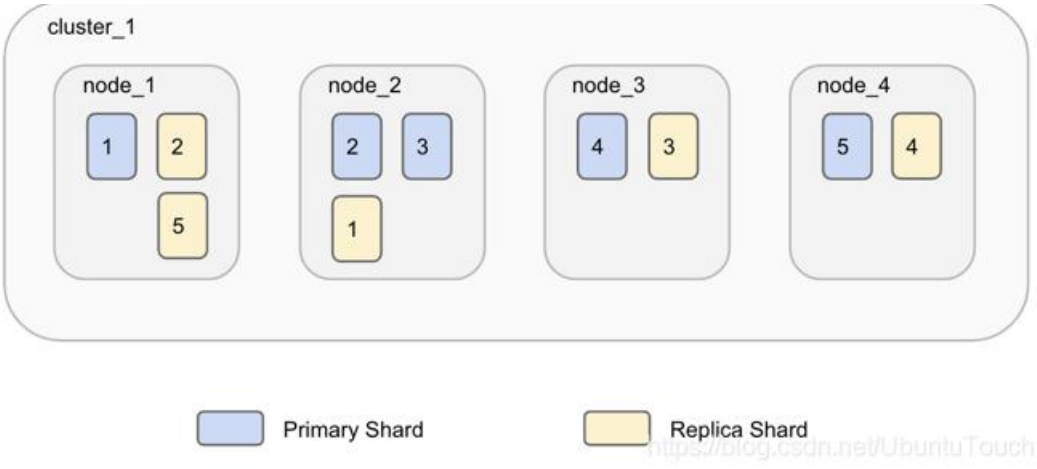
- 允许你水平分割/扩展你的内容容量。
- 允许你在分片（潜在地，位于多个节点上）之上进行分布式的、并行的操作，进而提高性能/吞吐量。

有两种类型的分片：Primary shard 和 Replica shard。

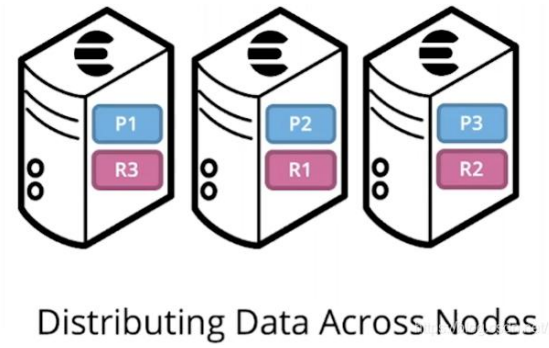
- Primary shard: 每个文档都存储在一个 Primary shard。 索引文档时，它首先在 Primary shard 上编制索引，然后在此分片的所有副本上（replica）编制索引。索引可以包含一个或多个主分片。 此数字确定索引相对于索引数据大小的可伸缩性。 创建索引后，无法更改索引中的主分片数。
- Replica shard: 每个主分片可以具有零个或多个副本。 副本是主分片的副本，有两个目的：
  - 增加故障转移：如果主要故障，可以将副本分片提升为主分片。
  - 提高性能：get 和 search 请求可以由主 shard 或副本 shard 处理。

默认情况下，每个主分片都有一个副本，但可以在现有索引上动态更改副本数。永远不会在与其主分片相同的节点上启动副本分片。

下面的图表示的是一个 index 有 5 个 shard 及 1 个 replica



这些 Shard 分布于不同的物理机器上：



我们可以为每个 Index 设置相应的 Shard 数值：

```
curl -XPUT http://localhost:9200/another_user?pretty -H 'Content-Type: application/json' -d '
{
    "settings" : {
        "index.number_of_shards" : 2,
        "index.number_of_replicas" : 1
    }
}
```

比如在上面的 REST 接口中，我们为 another_user 这个 index 设置了 2 个 shards，并且有一个 replica。一旦设置好 primary shard 的数量，我们就不可以修改了。这是因为 Elasticsearch 会依据每个 document 的 id 及 primary shard 的数量来把相应的 document 分配到相应的 shard 中。如果这个数量以后修改的话，那么每次搜索的时候，可能会找不到相应的 shard。

我们可以通过如下的接口来查看我们的 index 中的设置：

```
curl -XGET http://localhost:9200/twitter/_settings?pretty
```

上面我们可以得到 twitter index 的设置信息：

```
{
  "twitter" : {
    "settings" : {
      "index" : {
        "creation_date" : "1565618906830",
        "number_of_shards" : "1",
        "number_of_replicas" : "1",
        "uuid" : "rwgT8ppWR3aiXKsMHaSx-w",
        "version" : {
          "created" : "7030099"
        },
        "provided_name" : "twitter"
      }
    }
  }
}
```

Replica

默认情况下，Elasticsearch 为每个索引创建一个主分片和一个副本。这意味着每个索引将包含一个主分片，每个分片将具有一个副本。

分配多个分片和副本是分布式搜索功能设计的本质，提供高可用性和快速访问索引中的文档。主副本和副本分片之间的主要区别在于，只有主分片可以接受索引请求。副本和主分片都可以提供查询请求。

在上图中，我们有一个 Elasticsearch 集群，由默认分片配置中的两个节点组成。Elasticsearch 自动排列分割在两个节点上的一个主分片。有一个副本分片对应于每个主分片，但这些副本分片的排列与主分片的排列完全不同。

请允许我们澄清一下：请记住，number_of_shards 值与索引有关，而不是与整个群集有关。此值指定每个索引的分片数（不是群集中的主分片总数）。

我们可以通过如下的接口来获得一个 index 的健康情况：

```
http://localhost:9200/_cat/indices/twitter
```

上面的接口可以返回如下的信息：

更进一步的查询，我们可以看出：

如果一个 index 显示的是红色，表面这个 index 至少有一个 primary shard 没有被正确分配，并且有的 shard 及其相应的 replica 已经不能正常访问。 如果是绿色，表明 index 的每一个 shard 都有备份 （replica），并且其备份也成功复制在相应的 replica shard 之中。如果其中的一个 node 坏了，相应的另外一个 node 的 replica 将起作用，从而不会造成数据的丢失。

shard 健康：

- 红色：集群中未分配至少一个主分片
- 黄色：已分配所有主副本，但未分配至少一个副本
- 绿色：分配所有分片

创作人简介：

刘晓国，现为 Elastic 社区资深布道师。新加坡国立大学硕士，西北工业大学本硕。曾就职于新加坡科技，康柏电脑，通用汽车，爱立信，诺基亚，Linaro 非营利组织（Linux for ARM），Ubuntu，LinkMotion，Vantiq 等企业。从事过通信，电脑设计，计算机操作系统，物联网，汽车电子，云实时事件处理，大数据搜索等行业。从爱立信开始，到后来的诺基亚，Ubuntu 从事社区工作有超过 15 年以上经历。喜欢分享自己所学到的知识，希望和大家一起分享及学习。

博客：https://elasticstack.blog.csdn.net/

3.4 入门篇

3.4.1 Elastic Stack 安装部署

3.4.1.1 安装 Elasticsearch（本地及 docker)

创作人：陈晨

审稿人：刘帅

本章介绍 Elasticsearch （简称 ES ）的安装和部署，将会从以下几个方面进行阐述：

- 环境准备
- 系统级别参数配置
- 下载、安装及启动
- 常见问题及解决方案

环境准备

ES 和其他的服务一样，也是部署在服务器上进行使用的系统，为了能够让它更好的支持我们的实际使用，ES 节点/集群的部署环境就显得尤为重要。本节将从系统环境的选择，必须基础应用的安装等方面进行阐述。

环境选择策略

操作系统

由于绝大部分 ES 集群的部署环境都是基于公有云 Linux 的，所以本文主要以 CentOS 7 为基础进行阐述，同时也会针对 Windows 和 MacOS 环境的安装部署进行简要描述。

选择 CentOS 7 作为基础系统有以下一些考虑：

- ES 虽然是基于 JVM 上运行的 Java 项目，但它在启动、运行时会对一些环境参数，如虚拟内存数、文件句柄等有所要求。
- 国内的 ES 使用和部署中，以 CentOS 和 Debian 为主，存在少量的 Ubuntu 和极少量的 Windows 的服务器。

官方团队对 64 位系统进行了稳定性测试和系统性兼容，其中除了以下一些版本以外都可以较好的支持：

- CentOS 家族（6、7、8）中，CentOS6 不支持 7.9.2 版本捆绑的 JDK15+
- Debian 家族（7～10）中， Debian 7 从 5.x 版本开始不支持
- Ubuntu 家族（14、16、18）中，Ubuntu 14.04 从 7.x 版本开始不支持，但 Ubuntu 16，18 都支持 7.x 。
- Windows 家族（Windows server 2012/R2、2016、2019）中，只有 Windows server 2019 对 ES 7.7 之前的版本兼容性有限。

再结合通用云厂商和自建服务器的操作系统选型中，CentOS 7 得到了较好的支持和维护，所以此处我们选择以 CentOS 7 作为首选操作系统

内存、CPU

ES 节点启动的默认需求为 1C2G （1 核 CPU，2GB 内存）

通过调整 $ES_HOME/config/jvm.options 文件中的堆栈配置，也可以让 ES 实例在 1C1G 甚至更小资源的服务器上启动。

注意：更小的可用资源意味着更差的性能和节点稳定性，甚至节点启动失败。

实际生产中，更大的内存意味着更高的数据处理能力，更多的 CPU 核数可以支持更多的内部线程，但是无可避免的，更多的资源也意味着更高的系统开销。一般情况，内存和 CPU 的配比大致为 1:2 到 1:4 用以支持绝大部分数据存取、聚合等操作的使用，但是实际的内存、CPU 的用量和配比还需要用真实数据模拟真实生产环境的压测结果为准。

一般的内存、CPU 配置策略大致为以下几种：

- master 节点需要适当大小的内存
- coordination 节点需要较大的内存和 CPU
- ingest 节点需要较大的 CPU
- data 节点需要较大的内存和硬盘

实际生产中，需要通过压测来确定最佳的内存、CPU 配比，一般情况服务器的内存除了系统占用的固定内存之外，会建议设置为服务器可用内存的一半。

除了 ES 实例之外，ES 所维护的 Lucene 也是 Java 库，也需要占用相应的内存。此时，ES 的最大/小堆栈内存建议不超过 31G，否则会因为指针压缩的原因白白浪费内存资源，甚至可能出现数据存取更慢的问题。如果目标服务器的可用内存超过 64G 的话，可以考虑通过端口的配置部署多个 ES 实例。

磁盘

- 没做特殊配置的话，ES 会在写入及不断的查询过程中，将数据集中存在最新修改和召回的数据缓存在节点 /集群的各级内存（缓存）中。同时将绝大部分数据存在磁盘中的各种索引文件中，仅在内存中保留一部分索引文件的索引以加速数据的读写。
- 不同于内存中的文件，ES 放置在磁盘中的文件的读写是随机的不是顺序的，所以更快的随机读写速度将使 ES 提供更快的数据存取速度。
- 在在线搜索等高频存取的场景中，更建议使用固态硬盘以支持数据的高速读写。
- 在离线的日志存储等低频读取的场景中，则可以考虑用机械硬盘来节约成本。

JDK

- ES 作为一个 Java 应用，也需要运行在与之相匹配的 JVM 上。
- 相对于低版本的 ES，高版本的 ES 部署包会自带 JDK，所以只需要保证部署 ES 的系统可以支持对应的 JDK 就好。

- 目前 ES 7.10 所对应的 JDK 是 JDK11 至 15，所以部署的系统只要能支持 JDK 11 以上的 JDK 版本都可以用来做部署。
- ES 的部署建议尽量使用 ES 自带的 OpenJDK，因为 Elastic 团队会在每个版本中对对应的 OpenJDK 版本进行适配和调教，贸然使用其他版本的 JDK 可能会带来不可预见的问题。

实际系统配置

修改源并安装必要工具

```
    sudo sed -e 's|^mirrorlist=|#mirrorlist=|g' \
                -e 's|^#baseurl=http://mirror.centos.org/centos|baseurl=https://mirrors.ustc.edu.cn/centos|g' \
                -i.bak \
                /etc/yum.repos.d/CentOS-Base.repo &&  \
                yum makecache && \
                yum update -y && \
                yum install -y epel-release && \
                yum install -y curl wget htop unzip && \
                yum install -y docker docker-compose
```

开启 docker 服务：

- systemctl start docker
- systemctl enable docker

小结

本节对 ES 节点/集群部署所需环境对选择策略、必备软件等方面进行了阐述。

系统级别参数配置

ES 作为一个复杂的系统，对于服务器资源的要求相较于一般的服务要更严格，这样也能够保证 ，ES 节点可以更好的发挥其作用。本节将从各种系统级别的参数要求及修改意义方面进行阐述。

系统级别参数配置策略

ES 内部会启动包括而不仅限于 query 线程池、数据写入线程池、数据 refresh 线程池、segment merge 线程池等等。在启动时 ES 会要求系统中单个进程可用线程数超过 65535。

在 Linux 里万物皆文件，线程也可以看作一种特殊的文件。在启动时 ES 会要求系统中可打开的文件句柄数超过 65535。

在运行时，ES 会建议避免在运行过程中因为系统的缓存交换而产生的性能损耗。大部分操作系统有可能会将系统缓存中的数据交换到硬盘中。在 ES 节点部署的时候建议禁止这一交换行为。

在运行时，ES 会占用大量的内存进行一系列的数据处理。建议开启内存锁定的配置，将它所占用的内存进行锁定。

配置流程（太长不看版）

修改系统级别限制：

```
sed -e '/^vm.max_map_count/d' \
        -i.bak \
        /etc/sysctl.conf; \
    sed -e '$a vm.max_map_count=655360' \
        -i.bak \
        /etc/sysctl.conf; \
    sed -e '/^* soft nofile/d' \
        -e '/^* hard nofile/d' \
        -e '/^elasticsearch soft nofile/d' \
        -e '/^elasticsearch hard nofile/d' \
        -e '/^* soft memlock/d' \
        -e '/^* hard memlock/d' \
        -e '/^elasticsearch soft memlock/d' \
        -e '/^elasticsearch hard memlock/d' \
        -i.bak \
        /etc/security/limits.conf; \
    sed -e '$a * soft nofile 655350' \
        -e '$a * hard nofile 655350' \
        -e '$a elasticsearch soft nofile 655350' \
        -e '$a elasticsearch hard nofile 655350' \
```

```
        -e '$a * soft memlock unlimited' \
        -e '$a * hard memlock unlimited' \
        -e '$a elasticsearch soft memlock unlimited' \
        -e '$a elasticsearch hard memlock unlimited' \
        -i.bak \
        /etc/security/limits.conf; \
    swapoff -a;
```

使所有修改生效 reboot：

- 命令 sysctl -p 可以使 sysctl.conf 的配置生效。
- 重启 reboot 或者重新登陆 Ctrl + D 可以使 limits.conf 中的配置生效。

创建 ES 所用账号并切换 useradd -m elasticsearch; su elasticsearch;

配置流程（详解版）

调整机器中每个进程可以拥有的 VMA(虚拟内存区域)的数量

- 修改文件：/etc/sysctl.conf
- 添加/修改一行：vm.max_map_count=655360
- 否则可能会遇到报错：max virtual memory areas vm.max_map_count [65530] is too low, increase to at least [262144]

调整机器中每个进程可打开的文件句柄数量

- 修改文件：/etc/security/limits.conf
- 添加/修改两组：（*作用于所有用户，主要用于服务器直接部署 ES；elasticsearch 作用于 elasticsearch 用户，蛀牙用于服务器 rpm 包部署）

  - soft nofile 65535 => * soft nofile 655350
  - hard nofile 65535 => * hard nofile 655350
  - elasticsearch soft nofile 65535 => elasticsearch soft nofile 655350
  - elasticsearch hard nofile 65535 => elasticsearch hard nofile 655350
  - 否则可能遇到报错：max file descriptors [65535] for elasticsearch process is too low, increase to at least [65536]

开启内存锁定配置

- 修改文件：/etc/security/limits.conf
- 添加/修改两组：

  - soft memlock 65535 => * soft memlock unlimited
  - hard memlock 65535 => * hard memlock unlimited
  - elasticsearch soft memlock 65535 => elasticsearch soft memlock unlimited
  - elasticsearch hard memlock 65535 => elasticsearch hard memlock unlimited

- 否则可能在开启了内存锁定时（bootstrap.memory_lock: true）遇到报错：memory locking requested for elasticsearch process but memory is not locked

关闭内存交换区

```
$. swapoff -a
```

创建 ES 使用账号

- ES 在启动时默认不允许使用 root 账户，所以需要预先创建 ES 自己的账户
  - useradd -m elasticsearch

然后通过命令切换到 elasticsearch 账户中进行后续操作

- su elasticsearch

小结

本节对 ES 节点/集群部署所需环境参数及其意义进行了阐述，同时提供了最简初始化脚本和完整版初始化流程供参考。

安装实战

本节将对几个主流的 ES 安装部署的方式进行阐述，并会对节点的安装、部署、启动、停机等流程进行详细描述。

tar 包安装

下载链接（后面以 ES_DOWNLOAD_URL 指代）：

https://artifacts.elastic.co/downloads/elasticsearch/elasticsearch-7.10.0-linux-x86_64.tar.gz

下载并解压：

- mkdir -p /usr/local/elasticsearch
- cd /usr/local/elasticsearch
- wget -c ${ES_DOWNLOAD_URL}
- tar vxf elasticsearch-7.10.0-linux-x86_64.tar.gz

解压出来的文件：

```
[elasticsearch@esteam7001 elasticsearch]# ls -ltr elasticsearch-7.10.0
    总用量 584
    -rw-r--r--  1 elasticsearch elasticsearch  7313 11 月 10 05:28 README.asciidoc -> 项目说明文档
    -rw-r--r--  1 elasticsearch elasticsearch 13675 11 月 10 05:28 LICENSE.txt -> 协议
    drwxr-xr-x  2 elasticsearch elasticsearch  4096 11 月 10 05:32 plugins -> 插件文件夹，目前为空，自定义插件会放置在这里
```

```
drwxr-xr-x  2 elasticsearch elasticsearch    4096 11月 10 05:32 logs -> 日志文件夹
-rw-r--r--  1 elasticsearch elasticsearch  544318 11月 10 05:32 NOTICE.txt -> 一些协议的说明以及违反后果的警告
drwxr-xr-x  3 elasticsearch elasticsearch    4096 11月 10 05:34 lib -> 基础依赖库
drwxr-xr-x  2 elasticsearch elasticsearch    4096 11月 10 05:34 bin -> ES 内置的命令行工具，包括启动、密码生成等
drwxr-xr-x  9 elasticsearch elasticsearch    4096 11月 10 05:34 jdk -> ES 自带的 jdk
drwxr-xr-x 53 elasticsearch elasticsearch    4096 11月 10 05:34 modules -> ES 内置的各种功能模块，包括 Xpack 等
drwxr-xr-x  3 elasticsearch elasticsearch    4096 4月  15 19:02 config -> ES 的配置目录
```

最简启动

- 确认自己处于 非 root 用户，否则后续启动会报错
- cd elasticsearch-7.10.0
- ./bin/elasticsearch
- 如果要后台启动，只需在启动命令后面加上 -d
  - ./bin/elasticsearch -d
  - 完整路径 ./usr/local/elasticsearch/elasticsearch-7.10.0/bin/elasticsearch-d

在出现类似这些日志的时候，代表节点启动完成

```
[2021-04-15T19:57:24,720][INFO ][o.e.c.c.CoordinationState] [esteam7002] cluster UUID set to [rZ5dFfDWTTO3AUimscISOg] -> 声明集群 ID
```

[2021-04-15T19:57:24,931][INFO][o.e.c.s.ClusterApplierService] [esteam7002] master node changed {previous [], current [{esteam7002}{FBzZRCUQR1K0o8JEVpyfqg}{ikWQPL1TSAav5SSA0eqyZg}{127.0.0.1}{127.0.0.1:9300}{cdhilmrstw}{ml.machine_memory=1927176192, xpack.installed=true, transform.node=true, ml.max_open_jobs=20}]}, term: 1, version: 1, reason: Publication{term=1, version=1}

[2021-04-15T19:57:25,059][INFO][o.e.h.AbstractHttpServerTransport] [esteam7002] publish_address {127.0.0.1:9200}, bound_addresses {[::1]:9200}, {127.0.0.1:9200} -> 开始监听当前地址+端口

[2021-04-15T19:57:25,059][INFO][o.e.n.Node] [esteam7002] started -> 节点启动完成

ES 启动状态校验

You Know, for Search

```
[root@esteam7002 ~]# curl localhost:9200
{
    "name" : "esteam7002",
    "cluster_name" : "elasticsearch",
    "cluster_uuid" : "rZ5dFfDWTTO3AUimscISOg",
    "version" : {
        "number" : "7.10.0",
        "build_flavor" : "default",
        "build_type" : "tar",
        "build_hash" : "51e9d6f22758d0374a0f3f5c6e8f3a7997850f96",
        "build_date" : "2020-11-09T21:30:33.964949Z",
        "build_snapshot" : false,
        "lucene_version" : "8.7.0",
```

```
            "minimum_wire_compatibility_version" : "6.8.0",
            "minimum_index_compatibility_version" : "6.0.0-beta1"
        },
        "tagline" : "You Know, for Search"
    }
```

集群状态

```
[root@esteam7002 ~]# curl localhost:9200/_cat/health
1618488038 12:00:38 elasticsearch green 1 1 0 0 0 0 0 0 - 100.0%
```

节点停机

- ps -ef | grep elasticsearch | grep -v grep | awk '{ print$2 }' | xargs kill -15
- 没有后台启动的话，直接 ctrl + c 会输出类似以下的日志

```
[2021-04-15T20:05:47,962][INFO ][o.e.x.m.p.NativeController] [esteam7002] Native controller process has stopped - no new native processes can be started
[2021-04-15T20:05:47,964][INFO ][o.e.n.Node               ] [esteam7002] stopping ...
[2021-04-15T20:05:47,968][INFO ][o.e.x.w.WatcherService   ] [esteam7002] stopping watch service, reason [shutdown initiated]
[2021-04-15T20:05:47,969][INFO ][o.e.x.w.WatcherLifeCycleService] [esteam7002] watcher has stopped and shutdown
[2021-04-15T20:05:48,218][INFO ][o.e.n.Node               ] [esteam7002] stopped
[2021-04-15T20:05:48,218][INFO ][o.e.n.Node               ] [esteam7002] closing ...
[2021-04-15T20:05:48,234][INFO ][o.e.n.Node               ] [esteam7002] closed
```

rpm 包安装

下载链接（后面以 ES_DOWNLOAD_URL 指代）：

https://artifacts.elastic.co/downloads/elasticsearch/elasticsearch-7.10.0-x86_64.rpm

下载并安装：

- 切换到 root 账户（否则无法进行安装）sudo -i
- mkdir -p /usr/local/elasticsearch
- cd /usr/local/elasticsearch
- wget -c ${ES_DOWNLOAD_URL}
- rpm -ivh elasticsearch-7.10.0-x86_64.rpm

安装成功日志

日志会根据当前操作系统的语言而显示不同的语言提示，本示例系统为中文

```
[root@esteam7002 elasticsearch]# rpm -ivh elasticsearch-7.10.0-x86_64.rpm
警告：elasticsearch-7.10.0-x86_64.rpm: 头 V4 RSA/SHA512 Signature, 密钥 ID d88e42b4: NOKEY
准备中...                          ################################# [100%]
正在升级/安装...
```

```
1:elasticsearch-0:7.10.0-1 ################################# [100%]### NOT starting on installation, please execute the following statements to configure elasticsearch service to start automatically using systemd
sudo systemctl daemon-reload
sudo systemctl enable elasticsearch.service### You can start elasticsearch service by executing
sudo systemctl start elasticsearch.service
Created elasticsearch keystore in /etc/elasticsearch/elasticsearch.keystore
```

最简启动

- 确认自己处于 root 用户，否则命令需要加 sudo 前缀
- 通过命令 systemctl start elasticsearch 启动
- 没有报错表明节点启动完成

ES 启动状态校验

- 同上节

节点停机

- 通过命令 systemctl stop elasticsearch 停止服务
- 没有报错表明节点停机完成

Docker 安装

下载对应镜像

```
docker pull elasticsearch:7.10.1
```

(可选)如果目标机器无法上网，可以尝试通过其他机器下载并导入镜像

- 在宿主机下载镜像 docker pull elasticsearch:7.10.1
- 把镜像导出为文件 docker save -o elasticsearch-7.10.1-image.tar docker.io/elasticsearch:7.10.1
- 把导出的文件拷贝到目标机器 scp elasticsearch-7.10.1-image.tar root@192.168.10.221:/tmp
- 登陆目标机器 ssh root@192.168.10.221
- 导入目标镜像 docker load < elasticsearch-7.10.1-image.tar

镜像校验

```
docker images
```

```
[root@esteam7002 elasticsearch]# docker images
REPOSITORY                    TAG       IMAGE ID       CREATED         SIZE
docker.io/elasticsearch       7.10.1    558380375f1a   4 months ago    774 MB
```

最简启动

命令：

```
docker run -it \
    --rm \
    -p 9200:9200 \
    -p 9300:9300 \
    -e "discovery.type=single-node" \
    --name elasticsearch \
    elasticsearch:7.10.1 \
    bin/elasticsearch
```

如果需要开启包括禁止交换区、文件句柄限制等设置：

- 内存锁定：--ulimit memlock=-1:-1
- 打开文件上限：--ulimit nofile=655350:655350
- 关闭交换区：-e "bootstrap.memory_lock=true"
- 后台运行：-d
- 完整命令：

```
docker run -it \
    -d \
    --rm \
    --ulimit memlock=-1:-1 \
    --ulimit nofile=655350:655350 \
```

```
-e "bootstrap.memory_lock=true" \
-p 9200:9200 \
-p 9300:9300 \
-e "discovery.type=single-node" \
--name elasticsearch \
elasticsearch:7.10.1 \
bin/elasticsearch
```

- 通过命令查看日志 docker logs -f elasticsearch
- 出现以下日志内容代表启动成功

{"type": "server", "timestamp": "2021-04-19T09:58:55,399Z", "level": "INFO", "component": "o.e.n.Node", "cluster.name": "docker-cluster", "node.name": "adbdcd016771", "message": "initialized" } -> 节点初始化开始

{"type": "server", "timestamp": "2021-04-19T09:58:55,400Z", "level": "INFO", "component": "o.e.n.Node", "cluster.name": "docker-cluster", "node.name": "adbdcd016771", "message": "starting ..." }

{"type": "server", "timestamp": "2021-04-19T09:58:55,657Z", "level": "INFO", "component": "o.e.t.TransportService", "cluster.name": "docker-cluster", "node.name": "adbdcd016771", "message": "publish_address {172.17.0.2:9300}, bound_addresses {[::]:9300}" } -> 节点集群内部数据传输接口开启

{"type": "server", "timestamp": "2021-04-19T09:58:56,259Z", "level": "INFO", "component": "o.e.c.c.Coordinator", "cluster.name": "docker-cluster", "node.name": "adbdcd016771", "message": "setting initial configuration to VotingConfiguration{ScWOitRNSime0vB24lcmcA}" } -> 初始化投票配置

{"type": "server", "timestamp": "2021-04-19T09:58:56,605Z", "level": "INFO", "component": "o.e.c.s.MasterService", "cluster.name": "docker-cluster", "node.name": "adbdcd016771", "message": "elected-as-master ([1] nodes joined)[{adbdcd016771}{ScWOitRNSime0vB24lcmcA}{pKCm7

iZTR3eihceRySdZvA}{172.17.0.2}{172.17.0.2:9300}{cdhilmrstw}{ml.machine_memory=1927176192, xpack.installed=true, transform.node=true, ml.max_open_jobs=20} elect leader, _BECOME_MASTER_TASK_, _FINISH_ELECTION_], term: 1, version: 1, delta: master node changed {previous [], current [{adbdcd016771}{ScWOitRNSime0vB24lcmcA}{pKCm7iZTR3eihceRySdZvA}{172.17.0.2}{172.17.0.2:9300}{cdhilmrstw}{ml.machine_memory=1927176192, xpack.installed=true, transform.node=true, ml.max_open_jobs=20}]}" } -> 同集群节点发现

{"type": "server", "timestamp": "2021-04-19T09:58:56,838Z", "level": "INFO", "component": "o.e.c.c.CoordinationState", "cluster.name": "docker-cluster", "node.name": "adbdcd016771", "message": "cluster UUID set to [Of1IoqfaRMis216Bmgd5CQ]" } -> 集群 Id 初始化

{"type": "server", "timestamp": "2021-04-19T09:58:56,940Z", "level": "INFO", "component": "o.e.c.s.ClusterApplierService", "cluster.name": "docker-cluster", "node.name": "adbdcd016771", "message": "master node changed {previous [], current [{adbdcd016771}{ScWOitRNSime0vB24lcmcA}{pKCm7iZTR3eihceRySdZvA}{172.17.0.2}{172.17.0.2:9300}{cdhilmrstw}{ml.machine_memory=1927176192, xpack.installed=true, transform.node=true, ml.max_open_jobs=20}]}, term: 1, version: 1, reason: Publication{term=1, version=1}" } -> 集群选主完成

{"type": "server", "timestamp": "2021-04-19T09:58:57,039Z", "level": "INFO", "component": "o.e.h.AbstractHttpServerTransport", "cluster.name": "docker-cluster", "node.name": "adbdcd016771", "message": "publish_address {172.17.0.2:9200}, bound_addresses {[::]:9200}", "cluster.uuid": "Of1IoqfaRMis216Bmgd5CQ", "node.id": "ScWOitRNSime0vB24lcmcA" } -> 节点 restful 监听接口开启

{"type": "server", "timestamp": "2021-04-19T09:58:57,040Z", "level": "INFO", "component": "o.e.n.Node", "cluster.name": "docker-cluster", "node.name": "adbdcd016771", "message": "started", "cluster.uuid": "Of1IoqfaRMis216Bmgd5CQ", "node.id": "ScWOitRNSime0vB24lcmcA" } -> 节点初始完成

ES 启动状态校验

同上

节点停机

通过命令 docker ps -as 找到对应的 docker container

```
[root@esteam7002 elasticsearch]# docker ps -as
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES SIZE
adbdcd016771 elasticsearch:7.10.1 "/tini -- /usr/loc..." 2 hours ago Up 2 hours 0.0.0.0:9200->9200/tcp, 0.0.0.0:9300->9300/tcp elasticsearch 132 kB (virtual 774 MB)
```

关掉对应的 container docker stop elasticsearch

```
[root@esteam7002 elasticsearch]# docker stop elasticsearch
elasticsearch
```

Docker-compose 安装

- 下载镜像、校验等同 docker 安装 小结，本节略
- 创建配置文件 vi docker-compose.yml

```
# 声明 docker-compose 版本，Mac 等环境可以使用 3，但是在一些 Linux 环境中只支持到 2version: "2.2"
# 声明节点使用的网络空间 networks:
    bigdata:
        driver: bridge
# 声明节点使用的，从宿主机挂载进去的数据目录 volumes:
```

```
    es-data-01:
        driver: local
# 声明 ES 节点 services:
    # docker container
    es-node-01:
        # 使用的镜像及版本
        image: elasticsearch:7.10.1
        # container 是否自动重启
        restart: always
        # container 名称
        container_name: es-node-01
        # 环境参数
        environment:
            - node.name=es-node-01
            - cluster.name=docker-cluster
            - cluster.initial_master_nodes=es-node-01
            - discovery.seed_hosts=es-node-01
            - bootstrap.memory_lock=true
            - "ES_JAVA_OPTS=-Xms512m -Xmx512m"
        # 系统级别限制属性
        ulimits:
            memlock:
                soft: -1
                hard: -1
        # 宿主机挂载到 container 里的目录
        volumes:
            - es-data-01:/usr/share/elasticsearch/data
        # 开放端口映射
        ports:
```

```
      - 9200:9200
      - 9300:9300
    # 使用 docker 网络名称
    networks:
      - bigdata
```

最简启动

docker-compose up --build（如果要后台运行可以加参数 -d，如果目录里有多个配置文件可以通过 -f 参数指定目标配置文件）。

- 后台运行命令：docker-compose up -d --build
- 指定配置文件命令：docker-compose -f docker-compose.yml up -d --build

输出以下日志代表启动成功：

```
[root@esteam7002 elasticsearch]# docker-compose up --build
Creating network "elasticsearch_default" with the default driver -> 初次启动创建专用网络
Creating volume "elasticsearch_data01" with local driver -> 初次启动创建专用存储
Creating es-node-01 ... done -> 创建 ES 节点
Attaching to es-node-01
es-node-01 | {"type": "server", "timestamp": "2021-04-19T13:28:05,581Z", "level": "INFO",
"component": "o.e.n.Node", "cluster.name": "es-docker-cluster", "node.name": "es-node-01",
"message": "version[7.10.1], pid[6], build[default/docker/1c34507e66d7db1211f66f3513706fdf548
736aa/2020-12-05T01:00:33.671820Z], OS[Linux/3.10.0-1160.24.1.el7.x86_64/amd64], JVM[AdoptO
penJDK/OpenJDK 64-Bit Server VM/15.0.1/15.0.1+9]" } -> 节点启动系统环境信息
```

es-node-01 | {"type": "server", "timestamp": "2021-04-19T13:28:05,584Z", "level": "INFO",
"component": "o.e.n.Node", "cluster.name": "es-docker-cluster", "node.name": "es-node-01", "message": "JVM home [/usr/share/elasticsearch/jdk], using bundled JDK [true]" } -> 节点 JVM 使用信息

…… 中间省略

es-node-01 | {"type": "server", "timestamp": "2021-04-19T13:29:15,770Z", "level": "INFO", "component": "o.e.n.Node", "cluster.name": "es-docker-cluster", "node.name": "es-node-01", "message": "initialized" } -> 节点信息初始化

es-node-01 | {"type": "server", "timestamp": "2021-04-19T13:29:15,773Z", "level": "INFO", "component": "o.e.n.Node", "cluster.name": "es-docker-cluster", "node.name": "es-node-01", "message": "starting ..." } -> 节点启动开始

es-node-01 | {"type": "server", "timestamp": "2021-04-19T13:29:15,949Z", "level": "INFO", "component": "o.e.t.TransportService", "cluster.name": "es-docker-cluster", "node.name": "es-node-01", "message": "publish_address {172.18.0.2:9300}, bound_addresses {[::]:9300}" } -> 节点集群内部数据传输接口开启

es-node-01 | {"type": "server", "timestamp": "2021-04-19T13:29:16,441Z", "level": "INFO", "component": "o.e.c.c.Coordinator", "cluster.name": "es-docker-cluster", "node.name": "es-node-01", "message": "setting initial configuration to VotingConfiguration{s-6A5vqlQuKoZlyn-TKdnA}" } -> 初始化投票配置

es-node-01 | {"type": "server", "timestamp": "2021-04-19T13:29:16,834Z", "level": "INFO", "component": "o.e.c.s.MasterService", "cluster.name": "es-docker-cluster", "node.name": "es-node-01", "message": "elected-as-master ([1] nodes joined)[{es-node-01}{s-6A5vqlQuKoZlyn-TKdnA}{HD5TOEsDTUGWPFoR1pYUDg}{172.18.0.2}{172.18.0.2:9300}{cdhilmrstw}{ml.machine_memory=1927176192, xpack.installed=true, transform.node=true, ml.max_open_jobs=20} elect leader, _BECOME_MASTER_TASK_, _FINISH_ELECTION_], term: 1, version: 1, delta: master node changed {previous [], current [{es-node-01}{s-6A5vqlQuKoZlyn-TKdnA}{HD5TOEsDTUGWPFoR1pYUDg}{172.18.0.2}{172.18.0.2:9300}{cdhilmrstw}{ml.machine_memory=1927176192, xpack.installed=true, transform.node=true, ml.max_open_jobs=20}]}" } -> 同集群发现节点信息

```
es-node-01 | {"type": "server", "timestamp": "2021-04-19T13:29:17,047Z", "level": "INFO", "component": "o.e.c.c.CoordinationState", "cluster.name": "es-docker-cluster", "node.name": "es-node-01", "message": "cluster UUID set to [QDGTOlDiTm-9JkJrpg9q2Q]" } -> 集群 id 初始化
es-node-01 | {"type": "server", "timestamp": "2021-04-19T13:29:17,241Z", "level": "INFO", "component": "o.e.c.s.ClusterApplierService", "cluster.name": "es-docker-cluster", "node.name": "es-node-01", "message": "master node changed {previous [], current [{es-node-01}{s-6A5vqlQuKoZlyn-TKdnA}{HD5TOEsDTUGWPFoR1pYUDg}{172.18.0.2}{172.18.0.2:9300}{cdhilmrstw}{ml.machine_memory=1927176192, xpack.installed=true, transform.node=true, ml.max_open_jobs=20}]}, term: 1, version: 1, reason: Publication{term=1, version=1}" } -> 集群选主完成
es-node-01 | {"type": "server", "timestamp": "2021-04-19T13:29:17,315Z", "level": "INFO", "component": "o.e.h.AbstractHttpServerTransport", "cluster.name": "es-docker-cluster", "node.name": "es-node-01", "message": "publish_address {172.18.0.2:9200}, bound_addresses {[::]:9200}", "cluster.uuid": "QDGTOlDiTm-9JkJrpg9q2Q", "node.id": "s-6A5vqlQuKoZlyn-TKdnA" } -> 节点 restful 监听接口开启
es-node-01 | {"type": "server", "timestamp": "2021-04-19T13:29:17,316Z", "level": "INFO", "component": "o.e.n.Node", "cluster.name": "es-docker-cluster", "node.name": "es-node-01", "message": "started", "cluster.uuid": "QDGTOlDiTm-9JkJrpg9q2Q", "node.id": "s-6A5vqlQuKoZlyn-TKdnA" } -> 节点启动完成
```

ES 启动状态校验

同上

节点停机

- docker-compose down（docker-compose -f docker-compose.yml down）
- 当出现以下日志表明节点停机成功

```
[root@esteam7002 elasticsearch]# docker-compose -f docker-compose.yml down
Stopping es-node-01 ... done
Removing es-node-01 ... done
Removing network elasticsearch_default
```

MacOS 环境

MacOS 系统其实是一个类 Unix 系统，所以大部分的命令、环境属性等都和 Unix 系统类似。

tar 包安装和上文提到的，在 Linux 系统中基于 tar 包安装的过程大致一样。

- 需要注意的是，MacOS 作为一个较为特殊的操作系统，也会有一个专门针对 Mac OS 系统编译出来的安装包
- 路径上的主要区别为 Linux 系统的安装包被标记了 linux，MacOS 的安装包被标记了 darwin
- https://artifacts.elastic.co/downloads/elasticsearch/elasticsearch-7.10.0-darwin-x86_64.tar.gz

brew 安装

通过命令 brew tap elastic/tap 将 Elastic 原厂的仓库地址加入 homebrew 的配置：

```
brew tap elastic/tap
Updating Homebrew...
==> Tapping elastic/tap
Cloning into '/usr/local/Homebrew/Library/Taps/elastic/homebrew-tap'...
remote: Enumerating objects: 890, done.
remote: Counting objects: 100% (131/131), done.
remote: Compressing objects: 100% (104/104), done.
remote: Total 890 (delta 80), reused 57 (delta 26), pack-reused 759
Receiving objects: 100% (890/890), 206.46 KiB | 310.00 KiB/s, done.
Resolving deltas: 100% (666/666), done.
Tapped 17 formulae (51 files, 327.7KB).
```

通过命令 `brew install elastic/tap/elasticsearch-full` 进行安装

```
➜ ~ brew install elastic/tap/elasticsearch-full
==> Installing elasticsearch-full from elastic/tap
==> Downloading https://artifacts.elastic.co/downloads/elasticsearch/elasticsearch-7.12.1-d
arwin-x86_64.tar.gz?tap=elas############################################################
############## 100.0%
Warning: Tried to install empty array to /usr/local/etc/elasticsearch/jvm.options.d
==> codesign -f -s - /usr/local/Cellar/elasticsearch-full/7.12.1/libexec/modules/x-pack-ml/pl
atform/darwin-x86_64/contr
==> find /usr/local/Cellar/elasticsearch-full/7.12.1/libexec/jdk.app/Contents/Home/bin -type
 f -exec codesign -f -s - {
==> Caveats
Data:    /usr/local/var/lib/elasticsearch/elasticsearch_chenchen/ -> 数据目录
Logs:    /usr/local/var/log/elasticsearch/elasticsearch_chenchen.log -> 普通日志文件
Plugins: /usr/local/var/elasticsearch/plugins/ -> 插件目录
```

```
Config:  /usr/local/etc/elasticsearch/ -> 配置文件目录

To have launchd start elastic/tap/elasticsearch-full now and restart at login:
brew services start elastic/tap/elasticsearch-full
Or, if you don't want/need a background service you can just run:
elasticsearch
==> Summary
```

主要文件路径，也可以通过命令 brew info elasticsearch-full 查看（内容和安装日志类似）

```
➜ ~ brew info elasticsearch-full
elastic/tap/elasticsearch-full: stable 7.12.1
Distributed search & analytics engine
https://www.elastic.co/products/elasticsearch
Conflicts with:
elasticsearch
/usr/local/Cellar/elasticsearch-full/7.12.1 (960 files, 503MB) *
Built from source on 2021-05-14 at 23:45:38
From: https://github.com/elastic/homebrew-tap/blob/HEAD/Formula/elasticsearch-full.rb
==> Caveats
Data:     /usr/local/var/lib/elasticsearch/elasticsearch_chenchen/
Logs:     /usr/local/var/log/elasticsearch/elasticsearch_chenchen.log
Plugins: /usr/local/var/elasticsearch/plugins/
Config:  /usr/local/etc/elasticsearch/

To have launchd start elastic/tap/elasticsearch-full now and restart at login:
brew services start elastic/tap/elasticsearch-full
```

```
Or, if you don't want/need a background service you can just run:
elasticsearch
```

注意，通过常规方式安装 brew install elasticsearch 会安装 7.10.2，通过命令 brew info elasticsearch 也可以得到相应的信息，但是这个版本的 ES 存在一些启动上的问题

所以还是建议通过原厂建议的 brew info elasticsearch-full 命令进行安装和测试，目前通过这个命令会安装 7.12 版本，和本文中其它的版本不太一致

这些路径可能会因为系统的不同而有些许不同，也可能会有 /Users/steven/working/sourcecode/homebrew/etc/elasticsearch/ 格式的情况

服务启动

- 直接前台启动命令 elasticsearch
- 后台启动并随系统启动 brew services start elasticsearch

节点状态校验

- 同上

服务停机

- 前台启动时直接退出当前 terminal 窗口或者 control + c 结束进程

- 后台启动时通过命令 brew services stop elasticsearch

服务卸载（删除）

- brew uninstall elasticsearch-full

Windows 环境

Windows 的操作系统相对自成体系，所以在 Windows 平台中的安装可能会有些许不同。

zip 包安装

不同于 Linux 系统，Windows 系统用 zip 包进行压缩包安装：

https://artifacts.elastic.co/downloads/elasticsearch/elasticsearch-7.10.2-windows-x86_64.zip

主要的安装流程和 tar 包 安装类似，只是需要运行的是 $ES_HOME\bin\elasticsearch.bat 文件而非 Linux 系统中的 $ES_HOME/bin/elasticsearch

如果要在启动命令中添加参数（如指定节点名等），需要通过 cmd 工具或者其它的命令行工具进行操作

```
cd $ES_HOME\bin\
.\bin\elasticsearch.bat -Enode.name=my_node
```

msi 包安装

Windows 平台有自己的自引导安装包格式 msi 包（类似于前文的 rpm 包），可以将 ES 安装成 Windows 的系统服务：

https://artifacts.elastic.co/downloads/elasticsearch/elasticsearch-7.10.2.msi

msi 包安装是有 GUI 界面的，只需要双击打开并一步步进行配置就好，本文不再赘述，需注意以下几点：

不做修改的话，ES 的主要目录（data、config、log 等）都会放在 %ALLUSERSPROFILE%\Elastic\Elasticsearch\ 目录下（多半是在 C:\），当 C:\ 盘是系统安装盘又没有非常大空间的时候，建议将 ES 的安装路径换到其它盘符下。

安装过程中会有包括节点/集群信息、开放端口、内存使用等设置，配置思路和基于 Linux 系统的一致。

安装为系统级服务时，需要在安装时开启相应的命令参数。

- INSTALLASSERVICE：安装为系统服务
- STARTAFTERINSTALL：安装后启动
- STARTWHENWINDOWSSTARTS：随系统启动而启动

- 完整命令：start /wait msiexec.exe /i elasticsearch-7.10.2.msi /qn INSTALLASSERVICE=true STARTAFTERINSTALL=true STARTWHENWINDOWSSTARTS=true
- 安装完之后即可在系统服务中开启/关闭 ES 了

开发模式 VS 生产模式

本节中的 ES 是最简安装、启动，所以是以单节点（single-node）的方式启动。单节点启动默认是开发模式，会忽略绝大部分的启动校验。在不确定生产模式的强制校验项有哪些时，建议所有的部署节点的初始化流程都按上文中的配置流程逐一进行配置。

生产模式启动强制校验项：

配置文件	说明	配置项	默认值	期望值
conf/jvm.options	最大/小堆内存	-Xmx -Xms	-Xmx1g -Xms1g	部署节点可用内存的一半
elasticsearch.yml	内存锁定	bootstrap.memory_lock	NA	true
/etc/sysctl.conf	虚拟内存区域	vm.max_map_count	65530	> 262144
/etc/security/limits.conf	文件可打开句柄	soft/hard nofile	65536	655350

配置文件	说明	配置项	默认值	期望值
/etc/security/limits.conf	内存锁定	soft/hard memlock	NA	unlimited
/etc/security/limits.conf	虚拟内存	soft/hard as	NA	unlimited
/etc/security/limits.conf	最大进程数	soft/hard nproc	NA	10240
/etc/security/limits.conf	最大文件空间	soft/hard fsize	NA	unlimited

小结

本节列举了通过多种方式对 ES 节点进行最简安装、部署、停机等操作，希望读者可以找到适合自己的方式进行操作。

集群的组建

单个的 ES 节点可以支持普通的测试，但是对于生产的使用，特别是对数据安全性、可靠性、性能等维度有要求的使用中，应考虑使用 ES 集群支持生产的使用。本节将根据安装方式的不同，分别对集群的组建配置等进行阐述。

集群组建流程

ES 节点在启动时，会根据集群的信息以及自己的身份（配置在 $ES_HOME/config/elasticsearch.yml 里）尝试加入集群，其主要参数为。

- cluster.name 集群名称，节点会在同一网段中尝试找到和自己同一个集群的其他节点组建/加入现有集群。
- node.name 节点自身的标记，集群内唯一。
- node.master、node.data、node.ingest 节点身份，节点启动时会根据这个参数来进行注册。
- network.host 节点监听 IP 地址，一般建议本机 IP。
- http.port 节点监听端口，默认 9200
  - 不额外设置的话，集群节点间通信的端口会默认为 9300
  - 也可以通过参数 transport.port 来特殊指定为其它的端口
- cluster.initial_master_nodes 集群第一次初始化时的候选 master 节点列表。
- discovery.seed_hosts 集群节点发现会尝试访问的节点列表。
- ES 集群在启动/加入会根据节点的属性（master、master-eligible、data……）进行选主和状态同步，（注：集群选主流程将在后续章节进行详细阐述，本章只进行简要描述）
  - 集群初始化的时候（7.x），master 候选人（master）会尝试访问同一网段中同一个集群的所有节点。
  - 如果是新集群启动，配置在 cluster.initial_master_nodes 里的节点会优先成为 master 候选节点。

- 这些候选节点会发出投票及拉票请求给所有具有投票资格的节点（master、data、voting_only…）。
- 在获取足够多的票数之后，master 节点当选，否则 master 候选节点会在等待一段时间之后重新发起投票。请注意，为了防止集群脑裂的发生，这里这里建议在 cluster.initial_master_nodes 参数中设置奇数个（只需 1～3 个节点，不一定需要所有 master 候选节点）即可。
- master 会通过两段提交的方式将集群的信息、组织架构、模板等信息发布给集群中的所有节点。
- 当所有节点都成功同步集群状态之后，集群启动宣告完成。

tar 和 rpm 包安装方式

登陆每个 ES 节点，并修改配置文件并和其他节点组成集群。

- tar 包安装的配置文件 vi /usr/local/elasticsearch/config/elasticsearch.yml
- rpm 包安装的配置文件 vi /etc/elasticsearch/elasticsearch.yml

这里的 network.host 也可以配置为 _site_方便在节点批量初始化时进行配置。

```
# IP: 192.168.10.221
cluster.name: es-cluster
node.name: node-221
network.host: 192.168.10.221
http.port: 9200
discovery.seed_hosts: ["192.168.10.221", "192.168.10.222"]
cluster.initial_master_nodes: ["192.168.10.221"]
```

```
# IP: 192.168.10.222
cluster.name: es-cluster
node.name: node-222
network.host: 192.168.10.222
http.port: 9200
discovery.seed_hosts: ["192.168.10.221", "192.168.10.222"]
cluster.initial_master_nodes: ["192.168.10.221"]
```

节点启动

systemctl start elasticsearch

Docker 启动的配置方式

纯靠 docker run 命令方式启动 ES 集群会比较麻烦，建议通过 docker-compose 方式启动。

示例 docker-compose.yml 文件。

```
version: "2.2"
networks:
    bigdata:
        driver: bridge
volumes:
    es-data-01:
        driver: local
    es-data-02:
        driver: local
services:
    es-node-01:
```

```
        image: elasticsearch:7.10.1
        restart: always
        container_name: es-node-01
        environment:
            - node.name=es-node-01
            - cluster.name=docker-cluster
            - cluster.initial_master_nodes=es-node-01
            - discovery.seed_hosts=es-node-01,es-node-02
            - bootstrap.memory_lock=true
            - "ES_JAVA_OPTS=-Xms512m -Xmx512m"
        ulimits:
            memlock:
              soft: -1
              hard: -1
        volumes:
            - es-data-01:/usr/share/elasticsearch/data
        ports:
            - 9200:9200
            - 9300:9300
        networks:
            - bigdata

    es-node-02:
        image: elasticsearch:7.10.1
        restart: always
        container_name: es-node-02
        environment:
            - node.name=es-node-02
            - cluster.name=docker-cluster
            - cluster.initial_master_nodes=es-node-01
            - discovery.seed_hosts=es-node-01,es-node-02
            - bootstrap.memory_lock=true
            - "ES_JAVA_OPTS=-Xms512m -Xmx512m"
        ulimits:
            memlock:
              soft: -1
```

```
            hard: -1
        volumes:
            - es-data-02:/usr/share/elasticsearch/data
        networks:
            - bigdata
```

或者也可以在宿主机维护每个节点自己的 elasticsearch.yml 文件，并通过 -v $PATH/to/elasticsearch.yml:/usr/local/elasticsearch/config/elasticsearch.yml 的方式把这些配置文件映射到 docker container 里面进行使用。

这里需要注意在一个宿主机上，可以开放监听同一个端口的 container 只能有一个，所以如果需要整个集群里所有的节点都能监听/支持访问的话，需要把他们的 9200/9300 端口映射成宿主机里不同的端口，或者在 docker 环境中启动一个类似 NGINX 的网关来代理所有的节点。

小结

本节对 ES 集群组建流程进行了简要描述，并列举了通过多种方式对 ES 进行集群化配置。

常见问题及解决方案

本节将针对 ES 节点安装部署过程中经常遇到的一些问题进行分析，并提供一些简单的解决方案以供参考。

max virtual memory areas vm.max_map_count [65530] is too low, increase to at least [262144]，最大虚拟内存限制。

ES 在运行时会强依赖虚拟内存，在达不到 ES 要求限制时，ES 节点启动时会报错并启动失败。

修复方式：

- 在配置文件 /etc/sysctl.conf 中添加一行 vm.max_map_count=655360
- 运行命令 sudo sysctl -p 使配置生效

max file descriptors [65535] for elasticsearch process is too low, increase to at least [65536]，最大可持有文件句柄限制。

ES 在运行时会开启大量的线程（创建大量线程文件），在达不到 ES 要求限制时，ES 启动时会报错并启动失败。

修复方式：

- 修改配置文件 /etc/security/limits.conf
- 将 soft nofile 65535 和 hard nofile 65535 调大
- 如： soft nofile 655350 和 hard nofile 655350
- 如果使用 rpm 包安装，则需要单独对 elasticsearch 账号进行授权，elasticsearch oft nofile 655350 和 elasticsearch hard nofile 655350

memory locking requested for elasticsearch process but memory is not locked，内存锁定失败。

系统会默认关闭内存锁定，但是为了 ES 的使用性能，我们一般建议开启内存锁定（bootstrap.memory_lock: true）。

修复方式：

- 修改配置文件 /etc/security/limits.conf
- 将 soft memlock 65535 和 hard memlock 65535 调大
- 如： soft memlock unlimited 和 hard memlock unlimited
- 如果使用 rpm 包安装，则需要单独对 elasticsearch 账号进行授权，elasticsearch soft memlock unlimited 和 elasticsearch hard memlock unlimited

java.lang.RuntimeException: can not run elasticsearch as root，使用 root 账号启动。

在默认情况下，ES 是禁止使用 root 用户启动的。

修复方式：

- 切换成其他用户进行启动
- su elasticsearch

Error: Unable to find a match: docker-compose，找不到 docker-compose 对应安装包。

可能 yum 仓库中没有最新安装包信息或者精简版系统中没有对应的软件信息。

修复方式：

把源文件中应用市场的地址替换成中科大：

```
sudo sed -e 's|^mirrorlist=|#mirrorlist=|g' \
        -e 's|^#baseurl=http://mirror.centos.org/centos|baseurl=https://mirrors.ustc.edu.cn/centos|g' \
        -i.bak \
        /etc/yum.repos.d/CentOS-Base.repo
```

- 先安装 epel-release （拓展应用市场）
- 再进行后续安装

Cannot connect to the Docker daemon at unix:///var/run/docker.sock. Is the docker daemon running?docker 进程没启动。

docker 安装之后不会自动启动，在未设置之前，服务器重启之后 docker 多半也不会自动重启。

修复方式：

- 启动 docker 进程 systemctl start docker
- 设置 docker 随系统启动 systemctl enable docker

Get https://registry-1.docker.io/v2/: net/http: request canceled while waiting for connection (Client.Timeout exceeded while awaiting headers)，访问 docker 仓库失败。

在某些节点中可能无法直接访问外网进行 docker 镜像的下载。

修复方式：

- 开启外网访问
- （或者）在其他能够访问外网的节点中下载对应镜像 docker pull elasticsearch:7.10.1
- 把镜像导出为文件 docker save -o elasticsearch-7.10.1-image.tar docker.io/elasticsearch:7.10.1
- 把导出的文件拷贝到目标机器 scp elasticsearch-7.10.1-image.tar root@192.168.10.221:/tmp
- 登陆目标机器 ssh root@192.168.10.221

- 导入目标镜像 docker load < elasticsearch-7.10.1-image.tar

Error response from daemon: manifest for elasticsearch:7.9.11 not found: manifest unknown: manifest unknown，找不到目标镜像，可能在 docker 仓库中找

不到指定版本的镜像，登陆镜像仓库搜索合适版本（http://dockerhub.com/）（或者）通过命令搜索合适的镜像 docker search elasticsearch

ERROR: for es-node-01 Cannot start service es-node-01: driver failed programming external connectivity on endpoint es-node-01 (b483765a492a31517b047aae6e0b74c8c507851f7fd25850c87b504037262087): Bind for 0.0.0.0:9200 failed: port is already allocated

9200 端口已被占用，可能当前节点已经有其他服务绑定了 9200 端口，或者在 docker-compose 文件中设置了多个节点将 9200 端口映射到宿主机上。

修复方式：

通过 netstat -anp | grep 9200 命令寻找绑定 9200 端口的进程：

```
[root@esteam7002 elasticsearch-7.10.0]# netstat -anp | grep 9200
tcp6       0      0 :::9200                 :::*                    LISTEN      2130/docker-proxy-c
[root@esteam7002 elasticsearch-7.10.0]# netstat -anp | grep 9200
tcp6       0      0 127.0.0.1:9200          :::*                    LISTEN      2607/java
tcp6       0      0 ::1:9200                :::*                    LISTEN      2607/java
tcp6       0      0 ::1:45994               ::1:9200                TIME_WAIT   -
```

根据进程详细信息来决定是加入现有集群、节点重启或更换其他端口 ps -ef | grep 2607

```
[root@esteam7002 elasticsearch-7.10.0]# ps -ef | grep 2607
elastic+ 2607 1 29 17:08 pts/1 00:00:26 /usr/local/elasticsearch/elasticsearch-7.10.0/jdk/bin/java -Xshare:auto -Des.networkaddress.cache.ttl=60 -Des.networkaddress.cache.negative.ttl=10 -XX:+AlwaysPreTouch -Xss1m -Djava.awt.headless=true -Dfile.encoding=UTF-8 -Djna.nosys=true -XX:-OmitStackTraceInFastThrow -XX:+ShowCodeDetailsInExceptionMessages -Dio.netty.noUnsafe=true -Dio.netty.noKeySetOptimization=true -Dio.netty.recycler.maxCapacityPerThread=0 -Dio.netty.allocator.numDirectArenas=0 -Dlog4j.shutdownHookEnabled=false -Dlog4j2.disable.jmx=true -Djava.locale.providers=SPI,COMPAT -Xms512m -Xmx512m -XX:+UseG1GC -XX:G1ReservePercent=25 -XX:InitiatingHeapOccupancyPercent=30 -Djava.io.tmpdir=/tmp/elasticsearch-8284412088063757117 -XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=data -XX:ErrorFile=logs/hs_err_pid%p.log -Xlog:gc*,gc+age=trace,safepoint:file=logs/gc.log:utctime,pid,tags:filecount=32,filesize=64m -XX:MaxDirectMemorySize=268435456 -Des.path.home=/usr/local/elasticsearch/elasticsearch-7.10.0 -Des.path.conf=/usr/local/elasticsearch/elasticsearch-7.10.0/config -Des.distribution.flavor=default -Des.distribution.type=tar -Des.bundled_jdk=true -cp /usr/local/elasticsearch/elasticsearch-7.10.0/lib/* org.elasticsearch.bootstrap.Elasticsearch -d
elastic+ 2626 2607 0 17:08 pts/1 00:00:00 /usr/local/elasticsearch/elasticsearch-7.10.0/modules/x-pack-ml/platform/linux-x86_64/bin/controller
root 2688 703 0 17:10 pts/1 00:00:00 grep --color=auto 2607
```

ES 启动卡住，或者进行到一半就停止。

可能是系统可用内存过少或者配置的 ES 最大/最小内存过大。ES 在启动时会直接尝试申请 config/jvm.options 中申请的内存空间。

修复方式：

- 重新规划每个服务/程序的内存使用
- 调整 ES 的内存

创作人简介：

陈晨，十余年 IT 老兵，从售前做到运维，从后端做到 HR 和猎头，Hands on 过几乎 IT 生命周期的整个过程，不说样样精通，只希望能和不同岗位的同学尽可能站在一个 Baseline 上进行沟通和交流。希望能通过自己的一点努力，给更多的同学带来一些积极的影响，足矣。

博客：https://blog.csdn.net/weixin_40601534

3.4.1.2 Kibana（本地及 docker)

创作人：陈晨

审稿人：刘帅

环境准备

Kibana 是一个基于 Nodejs 构建出来的前端项目，它本身不包含数据存储功能，所以需要配合一个 Elasticsearch 节点/集群一起进行使用。本节将从系统环境的选择，必须的基础应用的安装等方面进行阐述。

环境选择策略

1. 操作系统

- 由于 Kibana 不能独立存在，需要绑定一个 Elasticsearch 节点/集群，所以本文主要会以一个 CentOS 7 系统来承载它配套的 Elasticsearch 节点。我们也将介绍其它常用操作系统的安装。
- Kibana 可以支持的系统和 Elasticsearch 类似，可以大致认为支持 Elasticsearch 的系统都可以承载 Kibana。

2. 内存、CPU

- Kibana 是一个前端系统，绑定的 Elasticsearch 可以认为是它用来存取数据用的数据库，所以不需要特别高的配置。

- 本文将以一个最小能够顺畅运行 Elasticsearch 节点的配置进行描述（1C2G）

实际系统配置

Kibana 的系统安装、配置和上一节 Elasticsearch 的安装一致，本节将不在赘述，只保留最简初始化方式。

修改源并安装必要工具

```
sudo sed -e 's|^mirrorlist=|#mirrorlist=|g' \
        -e 's|^#baseurl=http://mirror.centos.org/centos|baseurl=https://mirrors.ustc.edu.cn/centos|g' \
        -i.bak \
        /etc/yum.repos.d/CentOS-Base.repo &&  \
        yum makecache && \
        yum update -y && \
        yum install -y epel-release && \
        yum install -y curl wget htop unzip && \
        yum install -y docker docker-compose
```

开启 `docker` 服务：

- systemctl start docker
- systemctl enable docker

小结

本节对 Kibana 节点部署所需环境对选择策略、必备软件等方面进行了阐述。

下载、安装及启动

本节将对几个主流的 Kibana 安装部署的方式进行阐述，会对节点的安装、部署、启动、停机等流程进行详细描述。

tar 包安装

下载链接（后面以 KIBANA_DOWNLOAD_URL 指代）：

- https://artifacts.elastic.co/downloads/kibana/kibana-7.10.0-linux-x86_64.tar.gz

下载并解压：

- mkdir -p /usr/local/kibana
- cd /usr/local/kibana
- wget -c ${KIBANA_DOWNLOAD_URL}
- tar vxf kibana-7.10.0-linux-x86_64.tar.gz

解压出来的文件：

```
[elasticsearch@esteam7002 kibana]$ ls -ltr kibana-7.10.0-linux-x86_64
总用量 1324
-rw-rw-r--  1 elasticsearch elasticsearch    3968 11 月 10 06:16 README.txt -> 项目说明文档
drwxrwxr-x  2 elasticsearch elasticsearch    4096 11 月 10 06:16 plugins -> 插件文件夹，目前为空，自定义插件会放置在这里
-rw-rw-r--  1 elasticsearch elasticsearch     740 11 月 10 06:16 package.json -> 项目打包文件
-rw-rw-r--  1 elasticsearch elasticsearch 1263836 11 月 10 06:16 NOTICE.txt -> 一些协议的说明以及违反后果的警告
-rw-rw-r--  1 elasticsearch elasticsearch   13675 11 月 10 06:16 LICENSE.txt -> 协议
drwxrwxr-x  2 elasticsearch elasticsearch    4096 11 月 10 06:16 data -> Kibana 和它的插件写本地文件的文件夹
drwxrwxr-x  2 elasticsearch elasticsearch    4096 4 月  29 15:11 bin -> Kibana 内置命令行工具
drwxrwxr-x  4 elasticsearch elasticsearch    4096 4 月  29 15:11 x-pack
drwxrwxr-x 11 elasticsearch elasticsearch    4096 4 月  29 15:11 src
drwxrwxr-x  6 elasticsearch elasticsearch    4096 4 月  29 15:11 node
drwxrwxr-x 995 elasticsearch elasticsearch  36864 4 月  29 15:11 node_modules
drwxrwxr-x  2 elasticsearch elasticsearch    4096 4 月  29 15:12 config -> 配置文件目录
```

修改配置文件 ${KIBANA_HOME}/config/kibana.yml

- 添加 Elasticsearch 访问地址：elasticsearch.hosts: ["http://localhost:9200"]

服务启动：

- 通过命令./bin/kibana 启动
- kibana 不像 ES 有直接的后台运行参数，只能通过 nohup 配合 & 的方式后台运行
- 完整命令：nohup ./bin/kibana > kibana.log 2>&1 &
- 出现类似以下命令则代表启动完成

```
{"type":"log","@timestamp":"2021-04-29T07:21:08Z","tags":["info","plugins-service"],"pid":13089,"message":"Plugin \"visTypeXy\" is disabled."}
...中间省略...在配套的 ES 中创建了一些 Kibana 自己用的索引...
{"type":"log","@timestamp":"2021-04-29T07:21:11Z","tags":["info","plugins","watcher"],"pid":13089,"message":"Your basic license does not support watcher. Please upgrade your license."}
{"type":"log","@timestamp":"2021-04-29T07:21:11Z","tags":["info","plugins","monitoring","monitoring","kibana-monitoring"],"pid":13089,"message":"Starting monitoring stats collection"}
{"type":"log","@timestamp":"2021-04-29T07:21:12Z","tags":["listening","info"],"pid":13089,"message":"Server running at http://localhost:5601"}
{"type":"log","@timestamp":"2021-04-29T07:21:13Z","tags":["info","http","server","Kibana"],"pid":13089,"message":"http server running at http://localhost:5601"} -> 开启服务并可通过以下域名进行访问 http://localhost:5601
{"type":"log","@timestamp":"2021-04-29T07:21:15Z","tags":["warning","plugins","reporting"],"pid":13089,"message":"Enabling the Chromium sandbox provides an additional layer of protection."}
```

- 在浏览器里通过地址 `http://${node_ip}:5601` 进行访问

服务停机：

- Kibana 的进程是一个 nodejs 服务，所以不能像 Elasticsearch 一样通过 ps -ef | grep kibana 的方式获取进程 id
- 只能通过命令 netstat，通过查找监听端口的方式来找到 kibana 对应的 pid 并进行 kill 操作
  - 完整命令：netstat -anp | grep 5601 | awk '{ print $7 }' | cut -d '/' -f 1 | xargs kill -15

rpm 包安装

下载链接（后面以 KIBANA_DOWNLOAD_URL 指代）：

- https://artifacts.elastic.co/downloads/kibana/kibana-7.10.0-x86_64.rpm

下载并安装：

- 切换到 root 账户（否则无法进行安装）sudo -i
- mkdir -p /usr/local/kibana
- cd /usr/local/kibana
- wget -c ${KIBANA_DOWNLOAD_URL}
- rpm -ivh kibana-7.10.0-x86_64.rpm

安装成功日志：

日志会根据当前操作系统的语言而显示不同的语言提示，本示例系统为中文。

```
[root@esteam7002 kibana]# rpm -ivh kibana-7.10.0-x86_64.rpm
警告：kibana-7.10.0-x86_64.rpm: 头 V4 RSA/SHA512 Signature, 密钥 ID d88e42b4: NOKEY
准备中...                          ################################# [100%]
正在升级/安装...
   1:kibana-7.10.0-1                ################################# [100%]
```

修改配置文件 /etc/kibana/kibana.yml

- 添加 Elasticsearch 访问地址：elasticsearch.hosts: ["http://localhost:9200"]

服务启动：

- 确认自己处于 root 用户，否则命令需要加 sudo 前缀
- 通过命令 systemctl start kibana 启动
- 没有报错表明节点启动完成
- 在浏览器里通过地址 http://${node_ip}:5601 进行访问

服务停机：

- 通过命令 systemctl stop kibana 停止服务
- 没有报错表明服务停机完成

Docker/docker-compose 安装

下载对应镜像：

- docker pull kibana:7.10.1

(可选)如果目标机器无法上网，可以尝试通过其他机器下载并导入镜像：

- 在宿主机下载镜像 docker pull kibana:7.10.1
- 把镜像导出为文件 docker save -o kibana-7.10.1-image.tar docker.io/kibana:7.10.1
- 把导出的文件拷贝到目标机器 scp kibana-7.10.1-image.tar root@192.168.10.221:/tmp
- 登陆目标机器 ssh root@192.168.10.221
- 导入目标镜像 docker load < kibana-7.10.1-image.tar

镜像校验：

- docker images

```
[root@esteam7002 kibana]# docker images
REPOSITORY    TAG       IMAGE ID       CREATED         SIZE
kibana        7.10.1    3e014820ee3f   5 months ago    992 MB
```

服务启动：

- 不太建议直接命令行启动，因为需要和 Elasticsearch 节点配置共通网络之类的事情。

- 这里主要介绍通过 docker-compose 的方式进行管理。
- 修改配置文件 vi docker-compose.yml。

```
# 声明 docker-compose 版本，Mac 等环境可以使用 3，但是在一些 Linux 环境中只支持到 2
version: "2.2"

# 声明节点使用的网络空间
networks:
    bigdata:
        driver: bridge

# 声明 Kibana 节点
services:
    kibana:
        # kibana 版本要和 ES 相匹配，否则会报错甚至无法正常启动
        image: kibana:7.10.1
        container_name: kibana
        environment:
            # 如果 ES 节点和当前 kibana 节点在同一个 docker-compose 环境中
            # 可以直接写对应的 ES container_name，否则需要填完整的 URL
            ELASTICSEARCH_HOSTS: http://es01:9200
        depends_on:
            - es01
        ports:
            - 5601:5601
        networks:
            - bigdata
```

MacOS 环境

MacOS 系统其实是一个类 Unix 系统，所以大部分的命令、环境属性等都和 Unix 系统类似。

`tar 包` 安装和上文提到的，在 Linux 系统中基于 tar 包安装的过程大致一样。

- 需要注意的是，MacOS 作为一个较为特殊的操作系统，也会有一个专门针对 Mac OS 系统编译出来的安装包
- 路径上的主要区别为 Linux 系统的安装包被标记了 `linux`，MacOS 的安装包被标记了 `darwin`
- `https://artifacts.elastic.co/downloads/kibana/kibana-7.10.2-darwin-x86_64.tar.gzbrew` 安装

- Kibana 的 brew 安装和 Elasticsearch 类似，需要一些非常规的操作顺序
- 通过命令 `brew tap elastic/tap` 将 Elastic 原厂的仓库地址加入 homebrew 的配置。

```
brew tap elastic/tap
Updating Homebrew...
==> Tapping elastic/tap
Cloning into '/usr/local/Homebrew/Library/Taps/elastic/homebrew-tap'...
remote: Enumerating objects: 890, done.
remote: Counting objects: 100% (131/131), done.
remote: Compressing objects: 100% (104/104), done.
```

```
remote: Total 890 (delta 80), reused 57 (delta 26), pack-reused 759
Receiving objects: 100% (890/890), 206.46 KiB | 310.00 KiB/s, done.
Resolving deltas: 100% (666/666), done.
Tapped 17 formulae (51 files, 327.7KB).
```

通过命令 brew install elastic/tap/kibana-full 进行安装：

```
➜ ~ brew install elastic/tap/kibana-full
==> Installing kibana-full from elastic/tap
==> Downloading https://artifacts.elastic.co/downloads/kibana/kibana-7.12.1-darwin-x86_64.
tar.gz?tap=elastic/homebrew-t############################################################
############### 100.0%
==> Caveats
Config: /usr/local/etc/kibana/ -> 配置文件路径
If you wish to preserve your plugins upon upgrade, make a copy of
/usr/local/opt/kibana-full/plugins before upgrading, and copy it into the
new keg location after upgrading.

To have launchd start elastic/tap/kibana-full now and restart at login:
brew services start elastic/tap/kibana-full
Or, if you don't want/need a background service you can just run:
kibana
==> Summary
    /usr/local/Cellar/kibana-full/7.12.1: 53,439 files, 719.9MB, built in 1 minute 26 second
s
```

注意，通过常规方式安装 `brew install elasticsearch` 会安装 `7.10.2`，通过命令 `brew info elasticsearch` 也可以得到相应的信息

- 还是建议通过原厂建议的 `brew info elasticsearch-full` 命令进行安装和测试，目前通过这个命令会安装 7.12 版本，和本文中其它的版本不太一致。
- 这些路径可能会因为系统的不同而有些许不同，也可能会有 `/Users/steven/working/sourcecode/homebrew/etc/kibana/` 等格式的情况

主要文件路径，也可以通过命令 `brew info kibana-full` 查看

```
➜ ~ brew info kibana-full
elastic/tap/kibana-full: stable 7.12.1
Analytics and search dashboard for Elasticsearch
https://www.elastic.co/products/kibana
Conflicts with:
kibana
/usr/local/Cellar/kibana-full/7.12.1 (53,439 files, 719.9MB) *
Built from source on 2021-05-15 at 15:33:02
From: https://github.com/elastic/homebrew-tap/blob/HEAD/Formula/kibana-full.rb
==> Caveats
Config: /usr/local/etc/kibana/
If you wish to preserve your plugins upon upgrade, make a copy of
/usr/local/opt/kibana-full/plugins before upgrading, and copy it into the
new keg location after upgrading.

To have launchd start elastic/tap/kibana-full now and restart at login:
brew services start elastic/tap/kibana-full
```

```
Or, if you don't want/need a background service you can just run:
kibana
```

服务启动：

- 修改配置文件 $KIBANA_HOME/config/kibana.yml（本例为：/usr/local/etc/kibana/config/kibana.yml）
- 在里面配置 Elasticsearch 的 url，默认为 http://localhost:9200 或者 http://127.0.0.1:9200
- 前台启动命令：➜ ~ kibana
- 后台/作为系统服务启动：brew services start elastic/tap/kibana-full
- 在浏览器里通过地址 http://${node_ip}:5601 进行访问

Windows OS 环境

Windows 的操作系统相对自成体系，所以在 Windows 平台中的安装主要为.zip 包安装。

- 下载地址：https://artifacts.elastic.co/downloads/kibana/kibana-7.10.2-windows-x86_64.zip
- 主要的安装流程和 tar 包 安装类似，只是需要运行的是 $KIBANA_HOME\bin\kibana.bat 文件而非 Linux 系统中的 $KIBANA_HOME/bin/kibana
- Kibana 所需要的配置文件为 $KIBANA_HOME\config\kibana.yml

小结

本节列举了通过多种方式对 Kibana 节点进行最简安装、部署、停机等操作，希望读者可以找到适合自己的方式进行操作。

安全设置开启

Kibana 作为与 Elasticsearch 紧密相关的应用，在 Elasticsearch 开启了安全性认证的时候也需要相应的开启安全性认证。

- 在 Elasticsearch 集群中开启安全性设置
- 通过 bin/elasticsearch-setup-passwords 命令生成所需要账户的密码
- 修改配置文件 $KIBANA_HOME/config/kibana.yml 将 kibana 账号（7.x 之后可能会是 kibana_system 账号）对应的密码配置进相关参数中
  - elasticsearch.username: "kibana_system"
  - elasticsearch.password: "password"
- 在配置好安全设置之后初次登陆时需要以管理员（elastic）账号进行登陆，并进行后续的配置和操作

各环境的配置方式参考

- rpm 包安装：/etc/kibana/kibana.yml
- tar 包安装：$KIBANA_HOME/config/kibana.yml

- docker 安装：
  - 通过 -v 参数将本地配置文件映射到 docker 节点里：docker run -v $KIBANA_CONFIG_PATH/kibana.yml:/usr/share/kibana/config/kibana.yml docker.io/kibana:7.10.1
  - 修改 docker-compose.yml 文件里的 environment 配置

```
...
    environment:
        # 注意这几行
        ELASTICSEARCH_HOSTS: http://es01:9200
        ELASTICSEARCH_USERNAME: kibana
        ELASTICSEARCH_PASSWORD: kibana-password...
```

MacOS：

- tar 包安装：同上
- brew 安装：`brew info kibana-full` 中的配置地址，本例为：`/usr/local/etc/kibana/config/kibana.yml`

Windows：

- .zip 包安装：同上文中 tar 包安装

小结

本节中主要针对 Elasticsearch 集群开启了安全认证时，Kibana 需要进行的相关配置进行了阐述。

常见的参数优化

kibana 的功能已经较为完整，不太需要进行参数上的调整和优化，本节只探讨开启中文显示的方式。

- 同上节中的方式修改对应的配置文件
- 将参数 i18n.locale 设置成 zh-CN
  - 在 docker-compose.yml 中对应的参数为 I18N_LOCALE

常见问题及解决方案

本节将针对 Kibana 节点安装部署过程中经常遇到的一些问题进行分析，并提供一些简单的解决方案以供参考。

- Kibana should not be run as root. Use --allow-root to continue.
  - Kibana 和 ES 一样，不能直接通过 root 账号启动
  - 像提示中一样，可以通过添加参数 --allow-root 来启动
  - 完整命令：./bin/kibana --allow-root

- FATAL Error: Port 5601 is already in use. Another instance of Kibana may be running!，

- 5601 端口已被占用
- 修复方式：

通过 `netstat -anp | grep 5601` 命令寻找绑定 5601 端口的进程

```
[root@esteam7002 ~]# netstat -anp | grep 5601
tcp6       0      0 :::5601                 :::*                    LISTEN      3480/docker-proxy-c
```

1. 根据进程信息来决定是否需要关闭已有进程

无法连接到 ES：

```
log [10:08:28.980] [error][elasticsearch][monitoring] Request error, retrying
GET http://localhost:9200/_xpack => connect ECONNREFUSED 127.0.0.1:9200log [10:08:28.993] [warning][elasticsearch][monitoring] Unable to revive connection: http://localhost:9200/log [10:08:28.994] [warning][elasticsearch][monitoring] No living connectionslog [10:08:28.995] [warning][licensing][plugins] License information could not be obtained from Elasticsearch due to Error: No Living connections errorlog [10:08:29.016] [warning][monitoring][monitoring][plugins] X-Pack Monitoring Cluster Alerts will not be available: No Living connectionslog [10:08:29.025] [error][data][elasticsearch] [ConnectionError]: connect ECONNREFUSED 127.0.0.1:9200log [10:08:29.059] [error][savedobjects-service] Unable to retrieve version information from Elasticsearch nodes.log [10:08:31.476] [error][data][elasticsearch] [ConnectionError]: connect ECONNREFUSED 127.0.0.1:9200
```

- Kibana 无法直接连接 Elasticsearch url，可能有以下原因：

- 地址配置错误
- 当前节点和目标地址中间的网络不通
- Elasticsearch 配置的监听地址/端口有误

- 修复方式：
  - 检查该地址/域名是否正确
  - 通过 curl http://localhost:9200/ 命令测试一下当前节点是否能够正常的连接到目标地址
  - 调整并调试到正确的连接

- 认证失败

log [10:19:42.007] [error][data][elasticsearch] [security_exception]: missing authentication credentials for REST request [/_nodes?filter_path=nodes.*.version%2Cnodes.*.http.publish_address%2Cnodes.*.ip]log [10:19:42.042] [error][savedobjects-service] Unable to retrieve version information from Elasticsearch nodes.log [10:19:42.047] [warning][licensing][plugins] License information could not be obtained from Elasticsearch due to [security_exception] missing authentication credentials for REST request [/_xpack], with { header={ WWW-Authenticate="Basic realm=\"security\" charset=\"UTF-8\"" } } :: {"path":"/_xpack","statusCode":401,"response":"{\"error\":{\"root_cause\":[{\"type\":\"security_exception\",\"reason\":\"missing authentication credentials for REST request [/_xpack]\",\"header\":{\"WWW-Authenticate\":\"Basic realm=\\\"security\\\" charset=\\\"UTF-8\\\"\"}}],\"type\":\"security_exception\",\"reason\":\"missing authentication credentials for REST request [/_xpack]\",\"header\":{\"WWW-Authenticate\":\"Basic realm=\\\"security\\\" charset=\\\"UTF-8\\\"\"}},\"status\":401}","wwwAuthenticateDirective":"Basic realm=\"security\" charset=\"UTF-8\""} errorlog [10:19:42.050] [warning][monitoring][monitoring][plugins] X-Pack Monitoring Cluster Alerts will not be available: [security_exception]

```
missing authentication credentials for REST request [/_xpack], with { header={ WWW-Authenticate="Basic realm=\"security\" charset=\"UTF-8\"" } }log [10:19:44.442] [error][data][elasticsearch] [security_exception]: missing authentication credentials for REST request [/_nodes?filter_path=nodes.*.version%2Cnodes.*.http.publish_address%2Cnodes.*.ip]log [10:19:46.941] [error][data][elasticsearch] [security_exception]: missing authentication credentials for REST request [/_nodes?filter_path=nodes.*.version%2Cnodes.*.http.publish_address%2Cnodes.*.ip]
```

- Kibana 未配置安全性设置，以至于无法正常连接 Elasticsearch 节点/集群
- 当 Elasticsearch 节点/集群开启了安全性设置之后，所有的 restful 访问都需要添加认证设置，包括 kibana 的访问
- 修复方式：
- （如果没有设置）在 Elasticsearch 集群中选一个节点，运行 `bin/elasticsearch-setup-passwords` 命令生成所需要账户的密码
- 修改配置文件 `$KIBANA_HOME/config/kibana.yml` 将 `kibana` 账号（7.x 之后可能会是 `kibana_system` 账号）对应的密码配置进相关参数中

```
1. `elasticsearch.username: "kibana_system"`
1. `elasticsearch.password: "password"`
```

- 在配置好安全设置之后初次登陆时需要以管理员（`elastic`）账号进行登陆，并进行后续的配置和操作
- `Error: Unable to find a match: docker-compose`，找不到 `docker-compose` 对应安装包
  - 可能 yum 仓库中没有最新安装包信息或者精简版系统中没有对应的软件信息
  - 修复方式：

把源文件中应用市场的地址替换成中科大：

```
sudo sed -e 's|^mirrorlist=|#mirrorlist=|g' \
        -e 's|^#baseurl=http://mirror.centos.org/centos|baseurl=https://mirrors.ustc.edu.cn/centos|g' \
        -i.bak \
        /etc/yum.repos.d/CentOS-Base.repo
```

- 先安装 epel-release （拓展应用市场）
- 再进行后续安装
- Cannot connect to the Docker daemon at unix:///var/run/docker.sock. Is the docker daemon running?，docker 进程没启动
  - docker 安装之后不会自动启动，在未设置之前，服务器重启之后 docker 多半也不会自动重启
  - 修复方式：

  启动 docker 进程 systemctl start docker

  设置 docker 随系统启动 systemctl enable docker

- Get https://registry-1.docker.io/v2/: net/http: request canceled while waiting for connection (Client.Timeout exceeded while awaiting headers)，访问 docker 仓库失败
- 在某些节点中可能无法直接访问外网进行 docker 镜像的下载
- 修复方式：

1. 开启外网访问
1. （或者）在其他能够访问外网的节点中下载对应镜像 `docker pull kibana:7.10.1`
1. 把镜像导出为文件 `docker save -o kibana-7.10.1-image.tar docker.io/kibana:7.10.1`
1. 把导出的文件拷贝到目标机器 `scp kibana-7.10.1-image.tar root@192.168.10.221:/tmp`
1. 登陆目标机器 `ssh root@192.168.10.221`
1. 导入目标镜像 `docker load < kibana-7.10.1-image.tar`

- `Error response from daemon: manifest for kibana:7.9.11 not found: manifest unknown: manifest unknown`，找不到目标镜像
  - 可能在 docker 仓库中找不到指定版本的镜像
  - 登陆镜像仓库搜索合适版本（http://dockerhub.com/）
  - （或者）通过命令搜索合适的镜像 docker search kibana

小结

本节对 Kibana 部署和启动过程中常见的问题及参考解决方案进行了简要描述。

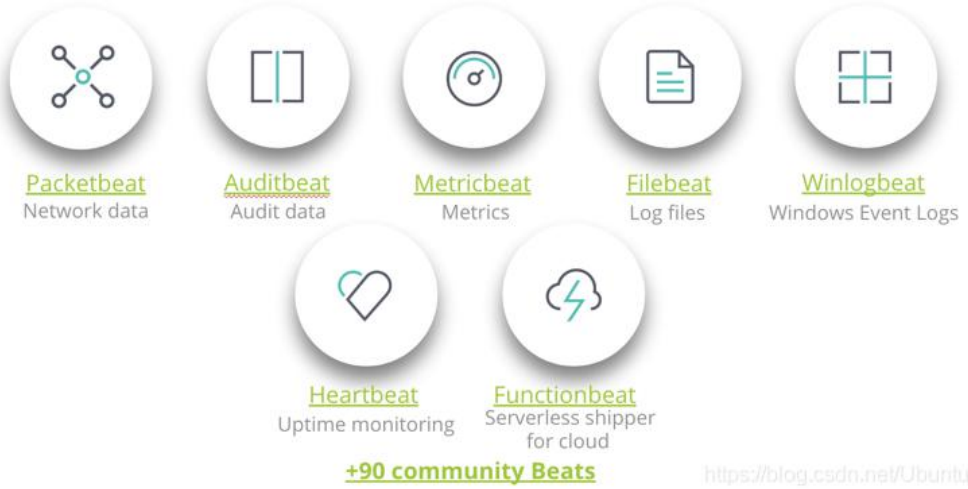
3.4.1.3 安装 Beats（本地及 docker)

创作人：冯江涛

审稿人：刘帅

Beats 是轻量级（资源高效，无依赖性，小型）和开放源代码日志发送程序的集合，这些日志发送程序充当安装在基础结构中不同服务器上的代理，用于收集日志或指标（Metrics）。这些可以是日志文件（Filebeat），网络数据（Packetbeat），服务器指标（Metricbeat）或 Elastic 和社区开发的越来越多的 Beats 可以收集的任何其他类型的数据。 收集后，数据将直接发送到 Elasticsearch 或 Logstash 中进行其他处理。Beats 建立在名为 libbeat 的 Go 框架之上，该框架用于数据转发，这意味着社区一直在开发和贡献新的 Beats。

Elastic Beats



环境准备

作为 Elastic Stack 的补充，在使用 Beats 之前，需要已安装好 Elasticearch 和 Kibana。Elasticsearch 用来存储，分析和检索数据，而 Kibana 作为可视化，监控和管理端。

接下来将基于 Elastic Stack 7.1.0 版本为基础，以 Metricbeat 组件为例，其他 Beats 组件使用方法类似。在安装 Beats 时，需要注意的一点是 Beats 的版本要和 Elasticsearch 及 Kibana 的版本一致，或至少是大版本是一致的。

Beats 组件的下载和安装

根据不同操作系统，选择合适的安装包。下面以 Elastic Stack 7.10 为例来展示 Metricbeat 的安装过程。如果你想安装其它版本的 Metricbeat，请替换命令行中的版本 7.10.0，然后按照同样的方法来进行安装。

deb:

```
curl -L -O https://artifacts.elastic.co/downloads/beats/metricbeat/metricbeat-7.10.0-amd64.deb
sudo dpkg -i metricbeat-7.10.0-amd64.deb
```

rpm:

```
curl -L -O https://artifacts.elastic.co/downloads/beats/metricbeat/metricbeat-7.10.0-x86_64.rpm
sudo rpm -vi metricbeat-7.10.0-x86_64.rpm
```

mac:

```
curl -L -O https://artifacts.elastic.co/downloads/beats/metricbeat/metricbeat-7.10.0-darwin-x86_64.tar.gz
tar xzvf metricbeat-7.10.0-darwin-x86_64.tar.gz
```

brew:

```
brew tap elastic/tap
brew install elastic/tap/metricbeat-full
```

linux:

```
curl -L -O https://artifacts.elastic.co/downloads/beats/metricbeat/metricbeat-7.10.0-linux-x86_64.tar.gz
tar xzvf metricbeat-7.10.0-linux-x86_64.tar.gz
```

win:

- 下载
  - zip 安装包：metricbeat-7.10.0-windows-x86_64.zip：https://artifacts.elastic.co/downloads/beats/metricbeat/metricbeat-7.10.0-windows-x86_64.zip

- MSI 安装包：metricbeat-7.10.0-windows-x86_64.msi：https://artifacts.elastic.co/downloads/beats/metricbeat/metricbeat-7.10.0-windows-x86_64.msi

- 解压并重命名 zip 安装包到 C:\Program Files\Metricbeat 。
- 以管理员身份打开 CDM 指令窗口。
- 进入 Metricbeat 目录执行安装.

```
# 1.进入目录
> cd C:\Program Files\Metricbeat
# 2.安装 Metricbeat 为 Windows 服务
> .\install-service-metricbeat.ps1
```

更多 Beats 组件可以前往下载页面 Beats Download：https://www.elastic.co/cn/downloads/beats

基础配置

进入解压后的 Metricbeat 目录可以看到，安装目录的根目录下如下文件及文件夹：

```
$ ls -l
fields.yml
kibana
LICENSE.txt
metricbeat# 完整的配置文件模板
metricbeat.reference.yml# 默认的配置文件
metricbeat.yml
Module
```

```
modules.d
NOTICE.txt
README.md
```

我们只修改使用 metricbeat.yml 这个配置文件。

如果你的 Elasticsearch 和 Kibana 都安装在同一台主机上，并配置了默认的端口，可以跳过此步骤不修改配置，Metricbeat 默认指定了 localhost。

```
output.elasticsearch:
  hosts: ["localhost:9200"]
  # 如果 ElasticSearch 启用了认证需要配置账号密码
  username: "YOUR_ACCOUNT"
  password: "YOUR_PASSWORD"setup.kibana:
  host: "localhost:5601"
  # 如果 kibana 启用了认证需要配置账号密码
  username: "YOUR_ACCOUNT"
  password: "YOUR_PASSWORD"
```

配置 Metricbeat，指定运行的模块。

```
# 查看所有支持的模块
./metricbeat modules list# 打开 system 模块
./metricbeat modules enable system
```

设置初始化环境，在此之前确保 Elasticsearch 和 Kibana 已经正常运行：

```
./metricbeat setup -e
```

上面的命令输出为：

```
./metricbeat setup
Overwriting ILM policy is disabled. Set `setup.ilm.overwrite: true` for enabling.

Index setup finished.
Loading dashboards (Kibana must be running and reachable)
```

如上所示。

在 setup 这个过程中，它将为 Beat 生成相应的 Dashboard，Index patterns，Index template，索引生命周期管理策略以及相应的 Ingest pipeline。这个命令的运行时间比较长。需要耐心等待。上面的命令针对一个 Beat 来说，只需要运行一次就可以了。

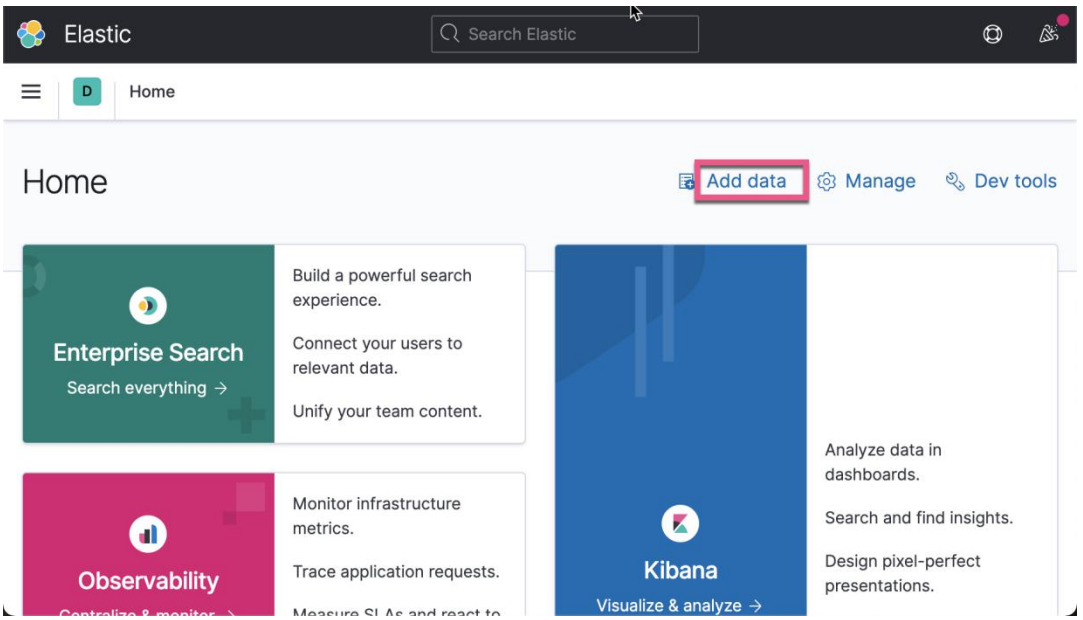
启动 Beats

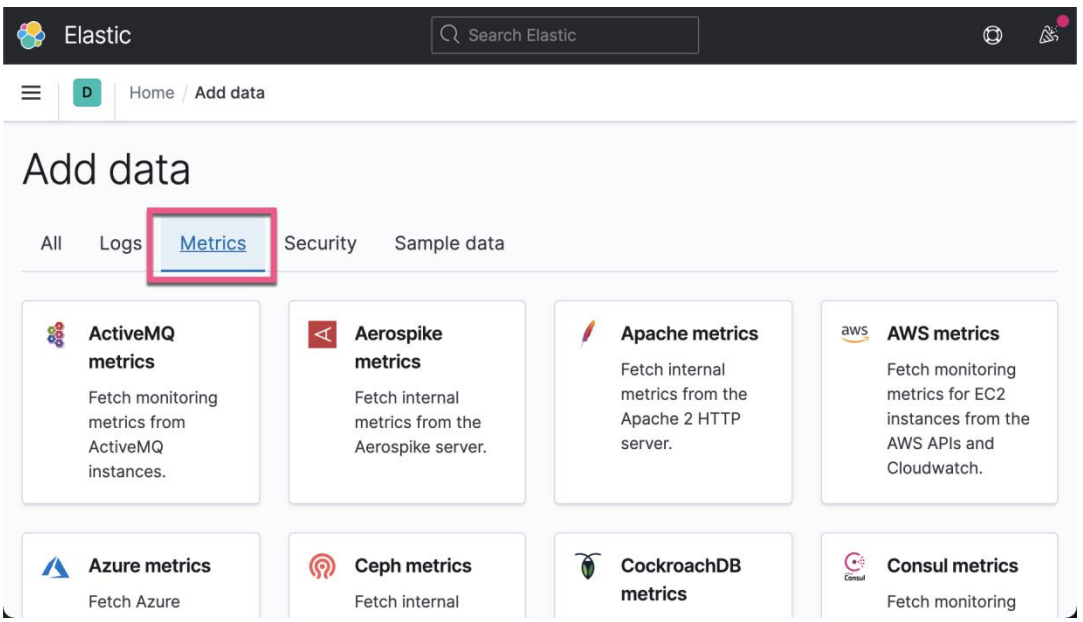
Metricbeat 启动后会发送 system metrics 数据到 Elasticsearch。

```
./metricbeat -e
```

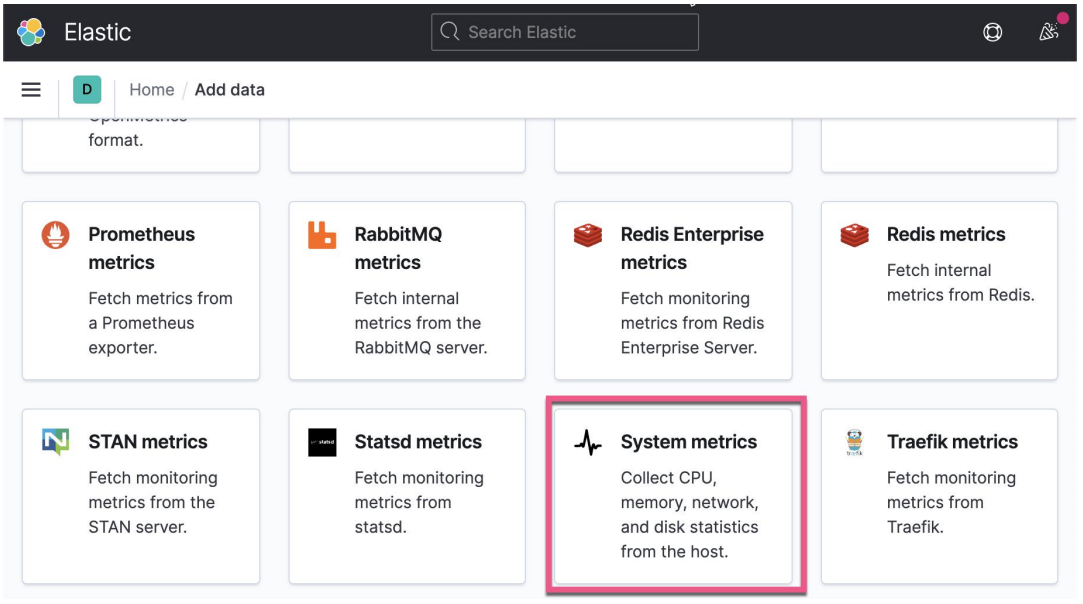
是不是觉得记忆上面的安装步骤很麻烦啊？在 Kibana 中，Elastic 已经为我们如何添加数据做了详细的描述，而且安装后的版本一定是和你安装的 Elasticsearch 及 Kibana 的版本是一样的。

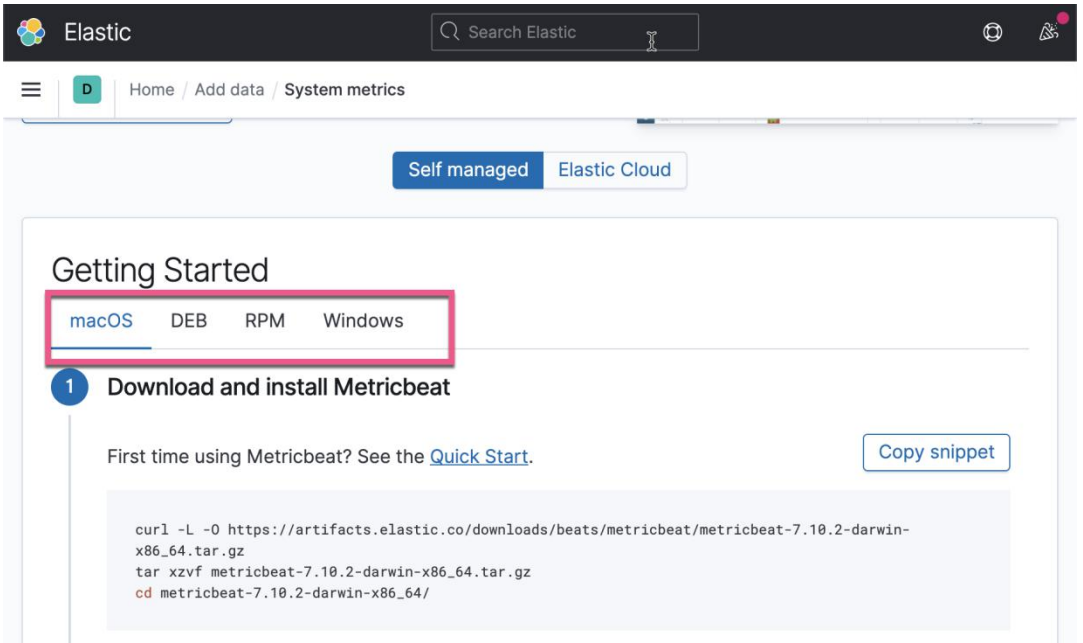
具体的操作步骤如下：


Elastic
Search Elastic
D
Home
Home
Add data
Manage
Dev tools
Enterprise Search
Search everything →
Build a powerful search experience.
Connect your users to relevant data.
Unify your team content.
Observability
Monitor infrastructure metrics.
Trace application requests.
Kibana
Visualize & analyze →
Analyze data in dashboards.
Search and find insights.
Design pixel-perfect presentations.


Elastic
Search Elastic
D
Home / Add data
Add data
All
Logs
Metrics
Security
Sample data
ActiveMQ metrics
Fetch monitoring metrics from ActiveMQ instances.
Aerospike metrics
Fetch internal metrics from the Aerospike server.
Apache metrics
Fetch internal metrics from the Apache 2 HTTP server.
AWS metrics
Fetch monitoring metrics for EC2 instances from the AWS APIs and Cloudwatch.
Azure metrics
Fetch Azure
Ceph metrics
Fetch internal
CockroachDB metrics
Consul metrics
Fetch monitoring

在上面显示了如下安装各种 Metrics 的具体步骤。以 System metrics 为例，在上面的页面中向下滚动：

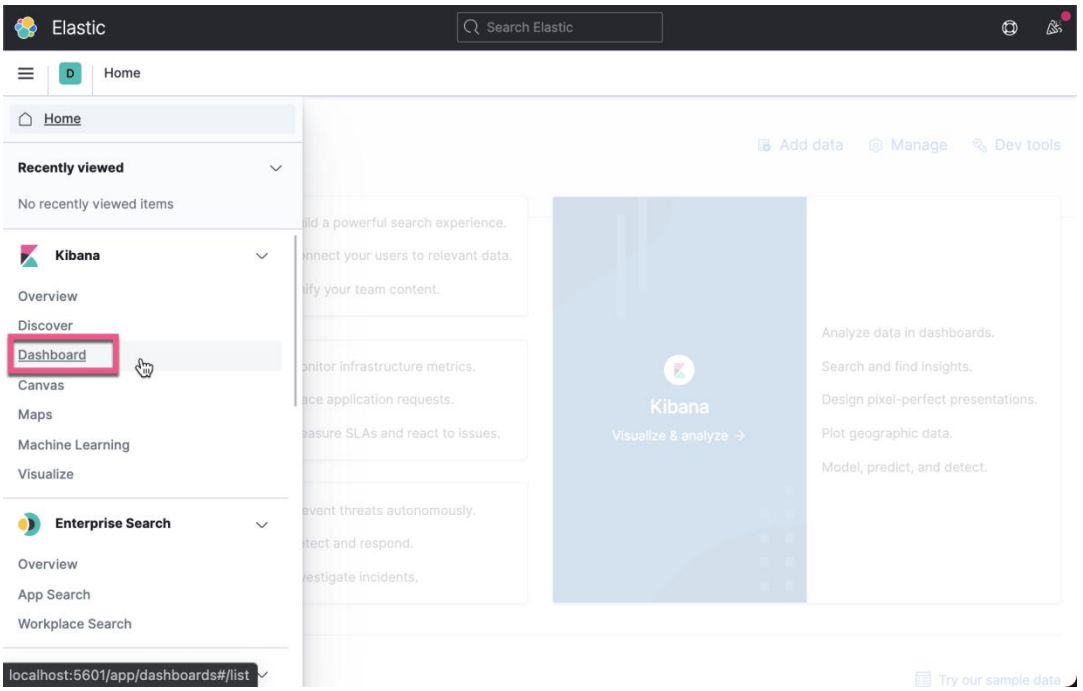


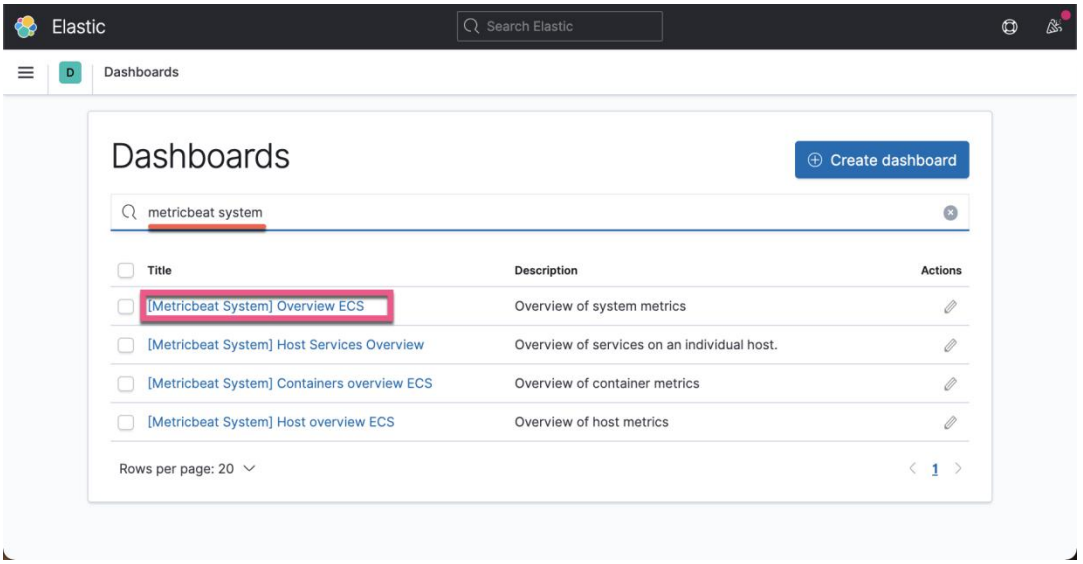


在上面，我们选择所需要的操作系统。再接着按照上面的安装步骤一步一步向下走。我们就可以完成所需要的 Beat 的安装。

检查收集到的数据

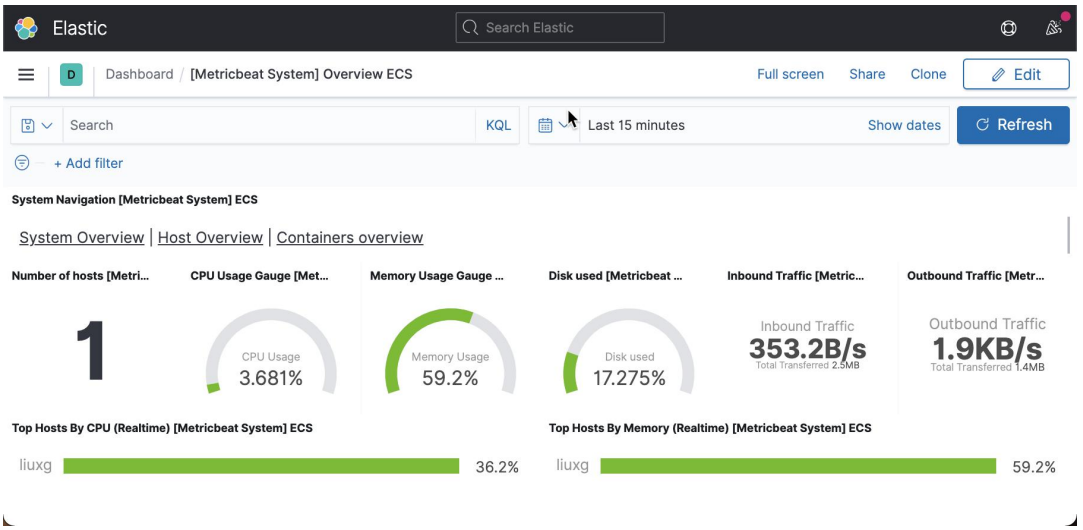
我们首先打开 Kibana，并启动 Dashboard：



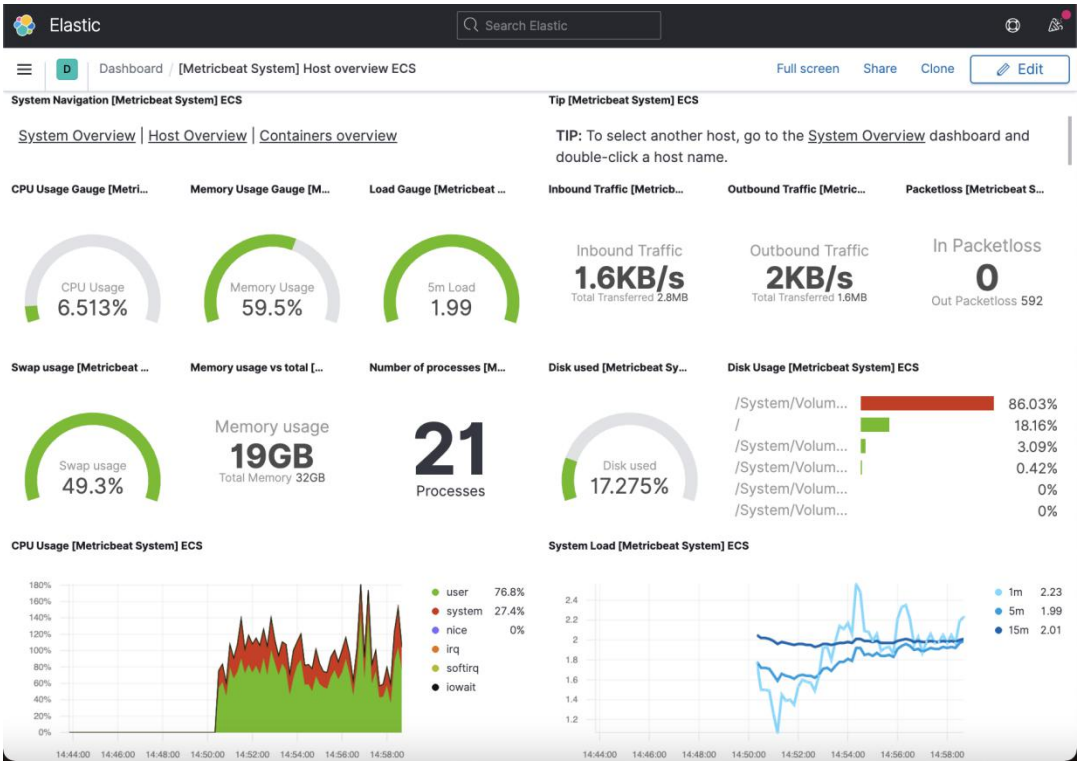


在上面，我们搜索 metricbeat system，我们就可以发现已经预置的 Dashboard。

我们选择 [Metricbeat System] Overview ECS：



点击 Host Overview 可以查看 Metrics 的详情：



Docker 方式安装

拉取镜像

```
docker pull docker.elastic.co/beats/metricbeat:7.10.0
```

启动 Docker 版 Metricbeat

通过 -E 设置 Elasticsearch 和 Kibana 的地址及其他参数(如果有必要)。

注意：如果是本机安装的服务，Docker 是无法通过 localhost 连接到 Elasticsearch 和 Kibana 的，可以通过增加参数 docker run --net=host，让 Docker 可以访问到宿主机的 hostname，或者可以通过 ip addr show docker0 查看 docker 的网关地址来访问宿主机。

```
docker run --net=host \
docker.elastic.co/beats/metricbeat:7.10.0 \
setup -E setup.kibana.host=elastichost:5601 \
-E output.elasticsearch.hosts=["elastichost:9200"]
```

创作人简介：

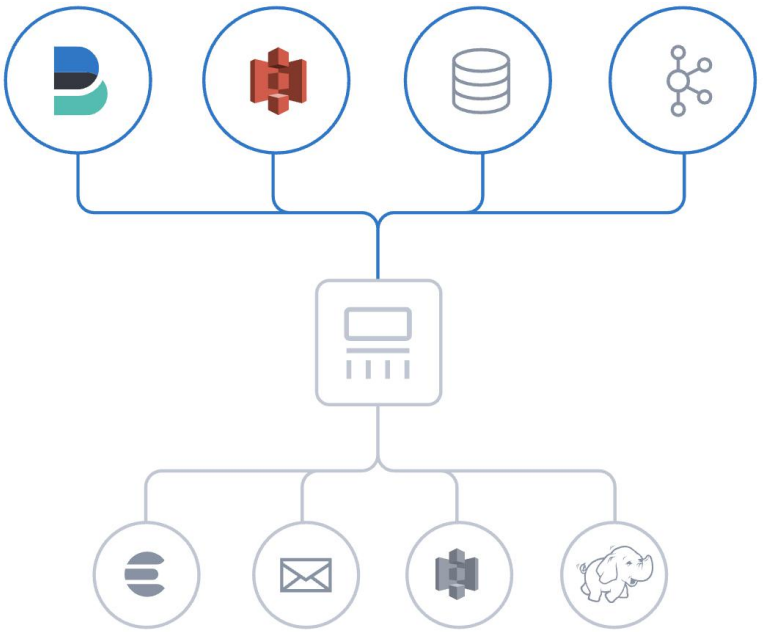
冯江涛，关注大数据相关技术栈，擅长 HDFS、Elasticsearch 存储相关，Spark、Flink 计算相关，以及大数据中台的开发。 Talk is cheap. Show me the code.

3.4.1.4 安装 Logstash（本地及 docker)

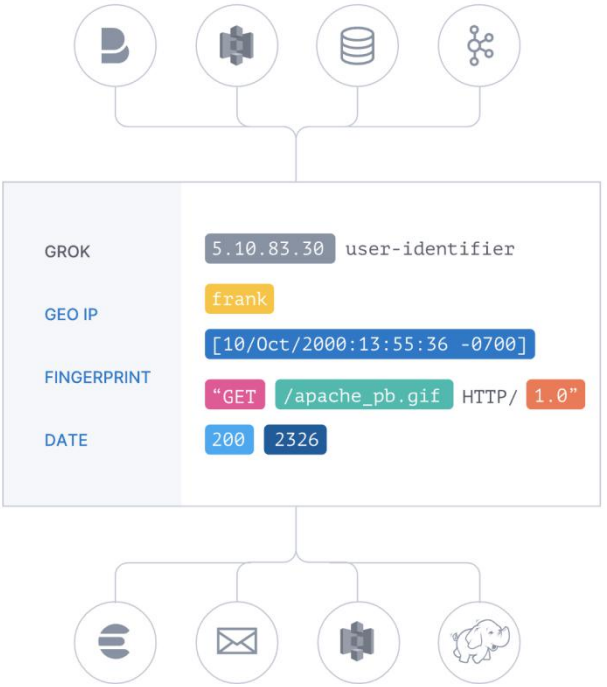
创作人：冯江涛

审稿人：刘帅

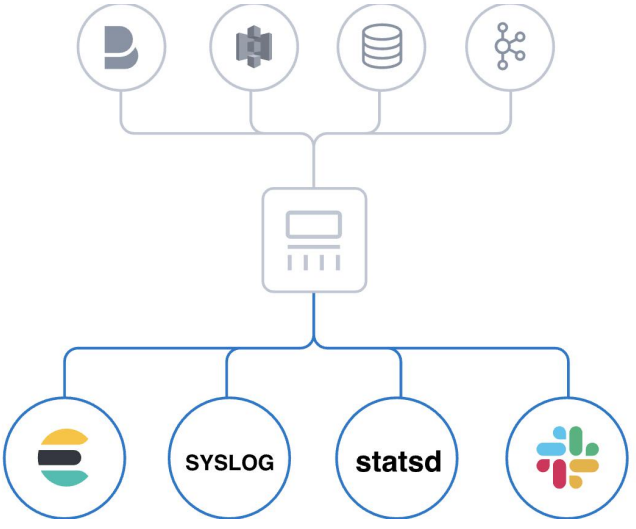
Logstash 是一个功能强大的工具，可与各种部署集成。 它提供了大量插件，可帮助你解析，丰富，转换和缓冲来自各种来源的数据。 如果你的数据需要 Beats 中没有的其他处理，则需要将 Logstash 添加到部署中。



Logstash 是 Elastic Stack 非常重要的一部分，但是它不仅仅为 Elasticsearch 所使用。它可以接受广泛的各种数据源。Logstash 可以帮利用它自己的 Filter 帮我们对数据进行解析，丰富，转换等。



最后，它可以把自己的数据输出到各种需要的数据储存地，这其中包括 Elasticsearch。



本章介绍 Logstash 的安装和部署，包括以下几个方面：

- 环境准备
- Logstash 的下载和安装
- 启动 Logstash
- 收集数据
- Docker 方式安装

环境准备

- JVM 运行环境 Logstash 依赖 JVM 运行环境，本文以 Java 8 版本进行介绍，支持以下 JVM 版本：8、11、15。

Logstash 的下载和安装

Logstash 的安装版本需要和 Elasticsearch 的版本一致，或至少是大版本号码一致。在下面以 7.10 版本为例来进行安装。如果你需要安装其它版本的 Logstash，那么请在相应的命令行替换相应的版本号码。

Linux:

```
> curl -L -O https://artifacts.elastic.co/downloads/logstash/logstash-7.10.0-linux-x86_64.tar.gz> tar xzvf logstash-7.10.0-linux-x86_64.tar.gz
```

APT

```
# 下载安装公钥
> wget -qO - https://artifacts.elastic.co/GPG-KEY-elasticsearch | sudo apt-key add -
> sudo apt-get install apt-transport-https
# 保存仓库地址到本地
> echo "deb https://artifacts.elastic.co/packages/7.x/apt stable main" | sudo tee -a /etc/apt/sources.list.d/elastic-7.x.list
# 安装 Logstash
> sudo apt-get update && sudo apt-get install logstash
```

YUM

```
# 下载安装公钥> sudo rpm --import https://artifacts.elastic.co/GPG-KEY-elasticsearch# 新建文件/etc/yum.repos.d/logstash.repo，并插入以下内容
[logstash-7.x]
name=Elastic repository for 7.x packages
baseurl=https://artifacts.elastic.co/packages/7.x/yum
gpgcheck=1
gpgkey=https://artifacts.elastic.co/GPG-KEY-elasticsearch
enabled=1
autorefresh=1
type=rpm-md
# 安装 logstash> sudo yum install logstash
```

Mac and Homebrew

181 > .3.4.1.4 安装 Logstash （本地及 docker)

```
# 安装 Elastic Homebrew 仓库> brew tap elastic/tap# 安装 Logstash> brew install elastic/tap/logstash-full# 通过 Homebrew 设置开机启动 logstash 服务> brew services start elastic/tap/logstash-full# 重启主机后，启动 Logstash> logstash
```

启动 Logstash

本文采用 tar 包安装方式进行阐述。

- 进入 Logstash 安装目录
- 最简配置启动 Logstash

```
# 通过控制台输入输出收集数据> bin/logstash -e 'input { stdin { } } output { stdout {} }'# 在控制台中输入 "Hello world!"，然后会看到控制台输出"Hello world!"
hello world
2013-11-21T01:22:14.405+0000 0.0.0.0 hello world
```

收集数据

Logstash 包含 3 个主要部分：输入（inputs），过滤器（filters）和输出（outputs）。下面以采集 log4j 日志并输出到 Elasticsearch 为例进行阐述。

创建收集数据的配置文件 bin/log4j2es.conf，插入以下内容：

```
input {
  file {
```

```
    # 要采集的 log 文件路径
      path => "/data/logs/springboot.log"
    }
}

filter {
}

output {
    stdout {
      codec => rubydebug
    }

    elasticsearch {
      hosts => ["localhost:9200"]
    }
}
```

指定配置启动 Logstash：

```
> bin/logstash -f bin/log4j2es.conf
# 或者后台启动
> nohup bin/logstash -f bin/log4j2es.conf >/dev/null 2>&1
&
```

查看收集到 ElasticSearch 索引的数据：

```
> curl http://localhost:9200/_cat/indices
```

默认 Logstash 生成以 Logstash 开头带有日期的索引：

```
green open logstash-2021.04.09-000001 3UhrpKMlRRCsJ7e5BRzHpA 1 1 0 0
208b 208b
```

查看索引中的数据：

```
> curl -XPOST 'http://localhost:9200/logstash-2021.04.09-000001/_search' -H 'Content-Typ
e: application/json' -d '{"query":{"match_all":{}}}'
```

返回如下结果：

```
{
    "took": 1,
    "timed_out": false,
    "hits": {
        "hits": [
            {
                "_index": "logstash-2021.04.09-000001",
                "_type": "_doc",
                "_id": "aTL3UHkBSh9MyZ_E_yVB",
                "_score": 1.0,
                "_source": {
                    "host": "elastichost",
                    "path": "/data/logs/springboot.log",
                    "message": "2021-04-09 17:58:47.172 INFO 23556 --- [ restartedMai
n] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Start completed3.",
```

```
                    "@version": "1",
                    "tags": [
                        "_grokparsefailure"
                    ],
                    "@timestamp": "2021-04-09T11:51:40.390Z"
                }
            }
        ]
    }
}
```

Docker 方式安装

拉取镜像

```
docker pull docker.elastic.co/logstash/logstash:7.10.0
```

Docker 模式运行 Logstash

```
# 1.参考 tar 包中 logstah/config 文件夹下所有配置拷贝一份放在宿主机 /usr/share/logstash/config/# 2.修改 pipeline.yml，增加以下配置
pipeline.id: main
path.config: /usr/share/logstash/config/log4j2es.conf# -v 挂载 Logstash 的配置/usr/share/logstash/config/到 docker 的路径~/settings/中
docker run --rm -it -v ~/settings/:/usr/share/logstash/config/ docker.elastic.co/logstash/logstash:7.10.0
```

3.4.1.5 配置集群安全访问

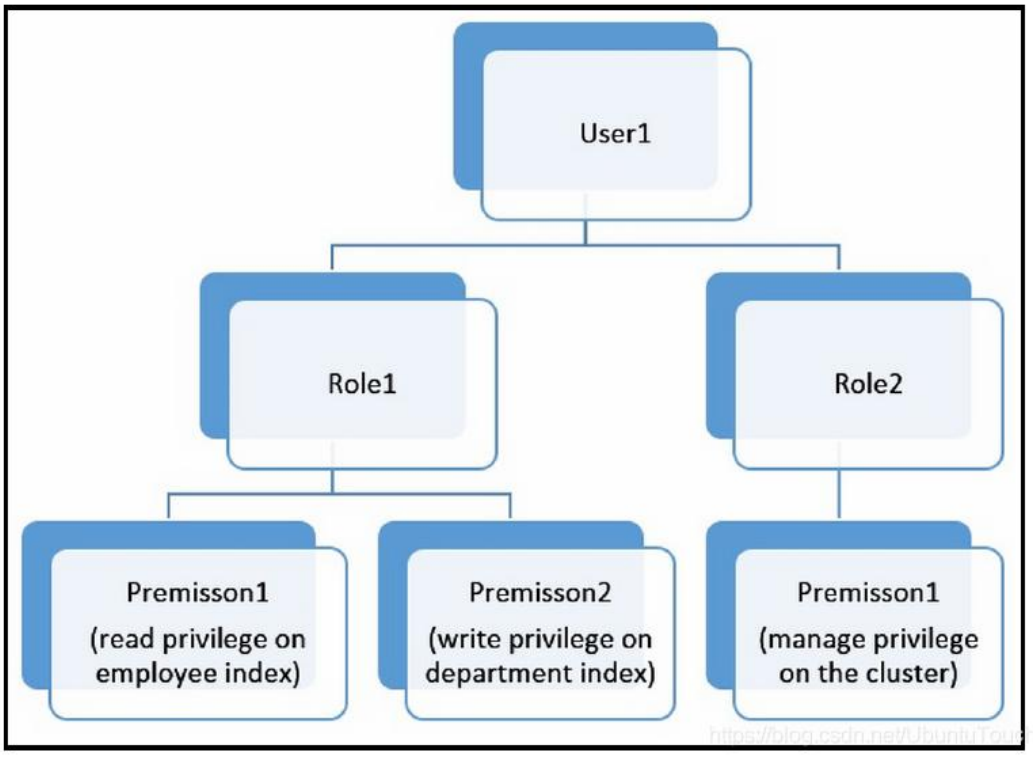
创作人：刘晓国

Elastic Stack 的组件是不安全的，因为它没有内置的固有安全性。 这意味着任何人都可以访问它。 在生产环境中运行 Elastic Stack 时，这会带来安全风险。 为了防止生产中未经授权的访问，采用了不同的机制来施加安全性，例如在防火墙后运行 Elastic Stack 并通过反向代理（例如 nginx，HAProxy 等）进行保护。 Elastic 提供商业产品来保护 Elastic Stack。 此产品是 X-Pack 的一部分，模块称为安全性。

本文中将介绍如何为我们的 Elastic 索引设置字段级的安全。这样有的字段对有些用户是可见的，而对另外一些用户是不可见的。我们也可以通过对用户安全的设置，使得不同的用户有不同的权限。

User authentication

在 X-Pack 安全性中，安全资源是基于用户的安全性的基础。 安全资源是需要访问以执行 Elasticsearch 集群操作的资源，例如索引，文档或字段。 X-Pack 安全性通过分配给用户的角色的权限来实现。 权限是针对受保护资源的一项或多项特权。 特权是一个命名的组，代表用户可以针对安全资源执行的一个或多个操作。 用户可以具有一个或多个角色，并且用户拥有的总权限集定义为其所有角色的权限的并集，如下图所示：

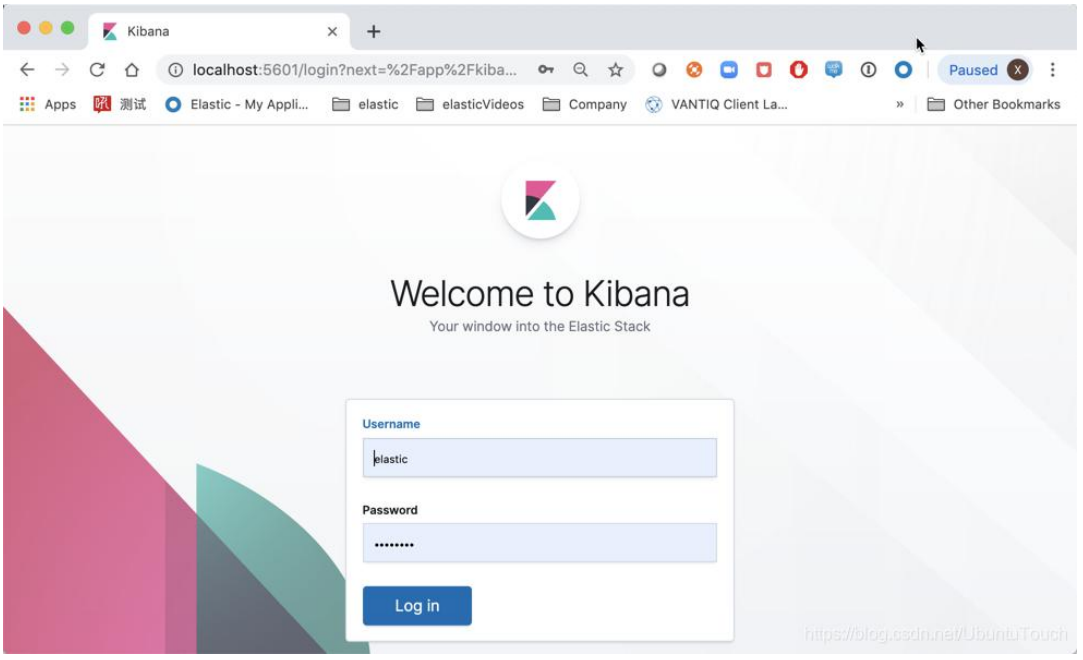


从上面的图上可以看出来：一个用户可以用多个 role，而每个 role 可以对应多个 permission（权限）。在接下来的练习中，我们来展示如何创建用户，role（角色）以及把 permission 分配到每个 role。通过这样的组合，我们可以实现对字段级的安全控制。

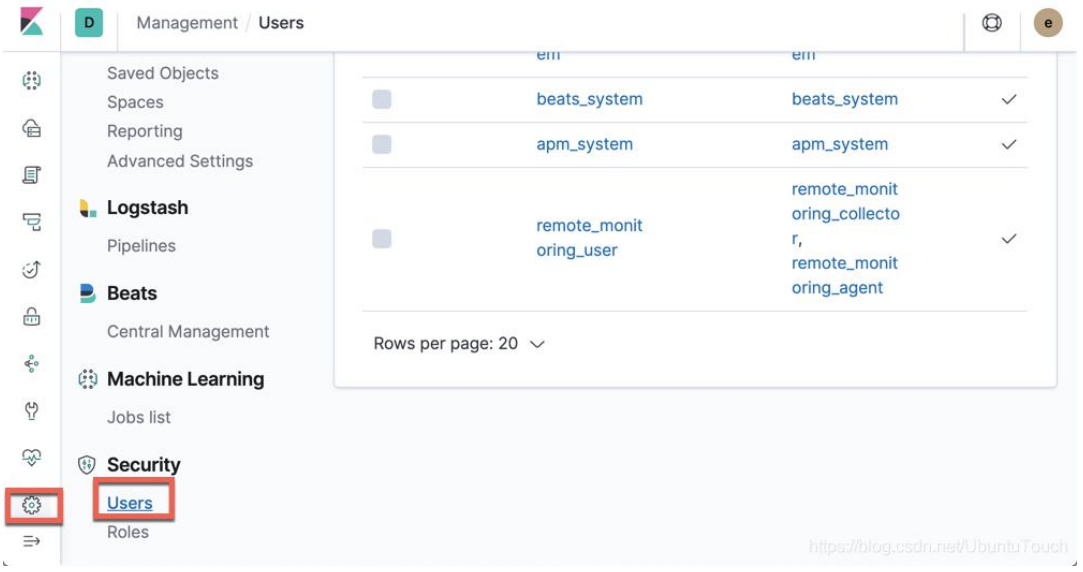
为 Elastic 设置安全及创建用户

当我们设置完我们的安全账户后，最开始我们使用最原始的 Elastic 的账号进行登录。

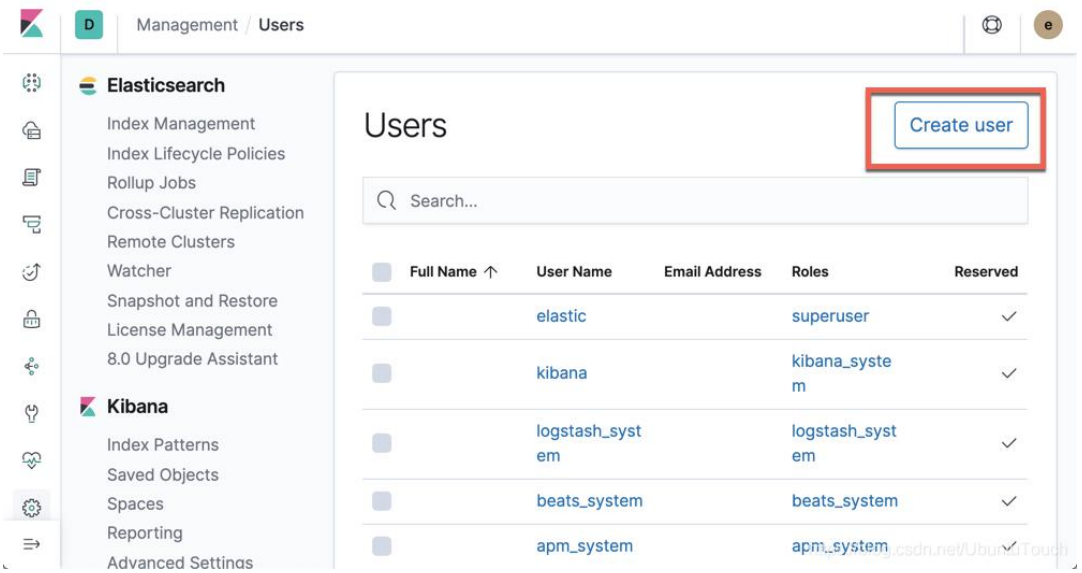
请注意这里的密码是我们设置 Elastic 账号的密码：



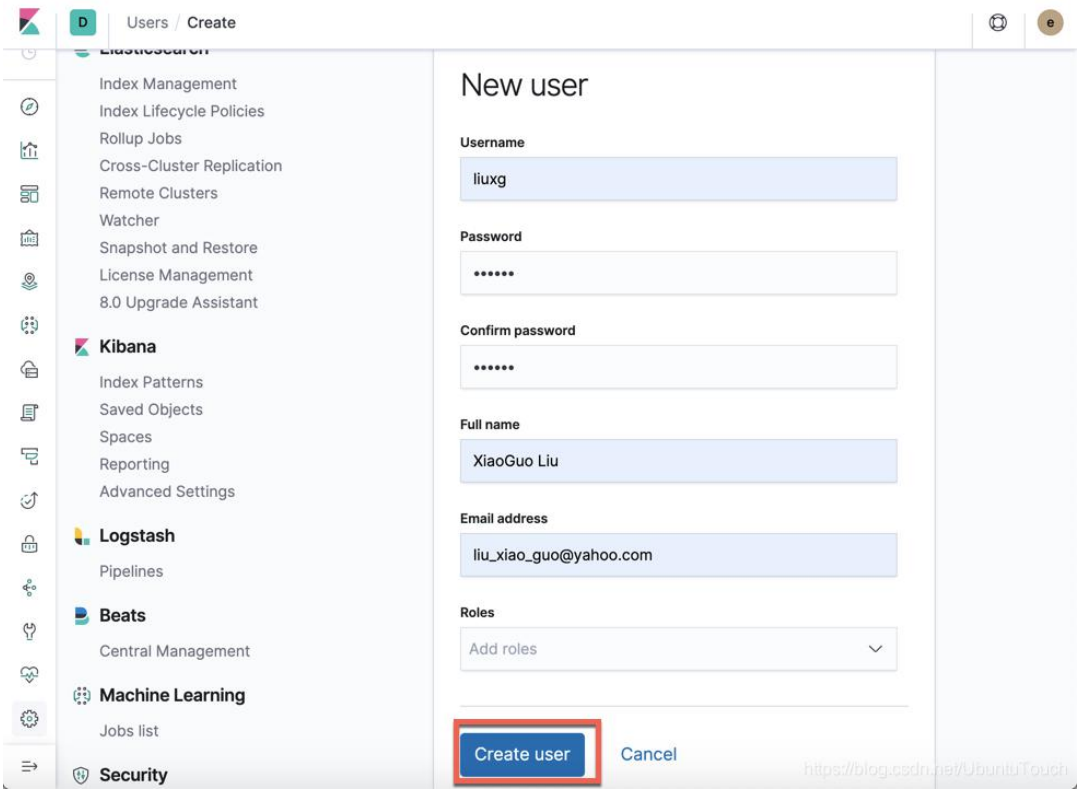
等登录进去之后，现在我们去 Manage/Sercurity/Users 页面：



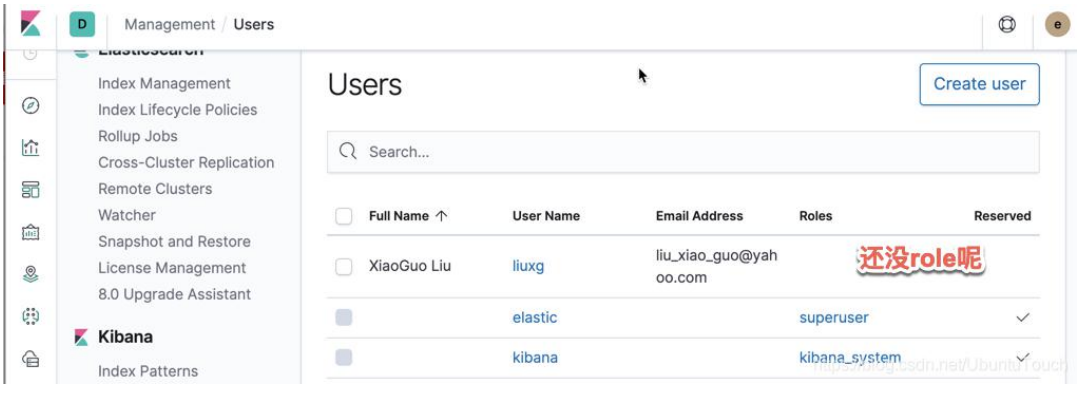
我们来创建一个新的账号。针对我的情况，我想创建一个叫做 liuxg 的用户名。点击当前页面的 Create User 按钮：



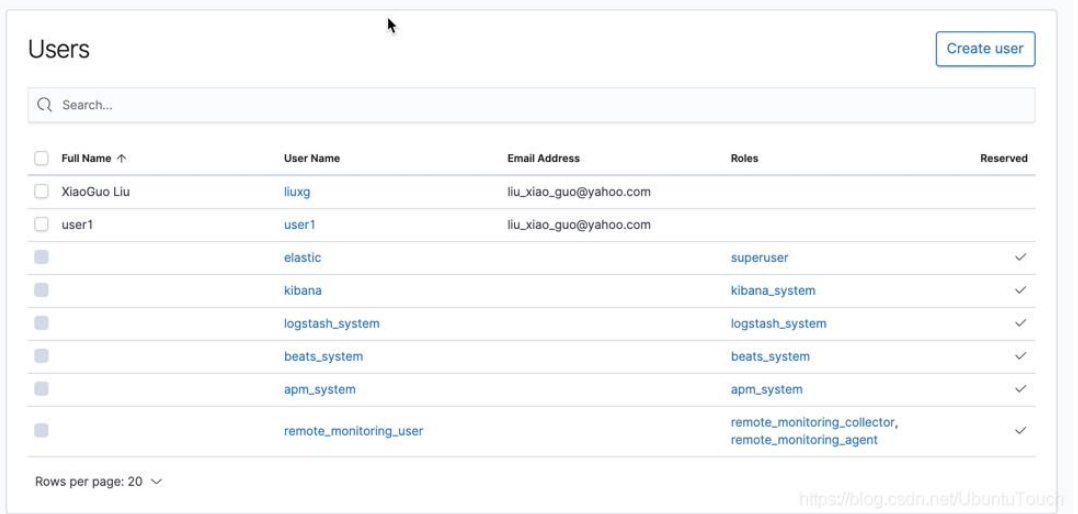
然后填入我们所需要的信息：



点击 Create User 按钮，这样我们就创建了我们的用户。



按照同样的步骤，我们来创建另外一个叫做 user1 的用户。

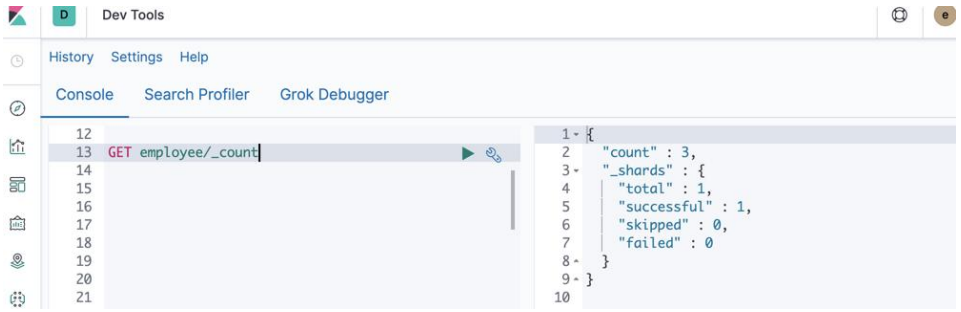


准备实验数据

在我们还没退出 elastic 用户的情况下，我们使用 bulk API 来把如下的文档输入到 Elasticsearch 中。

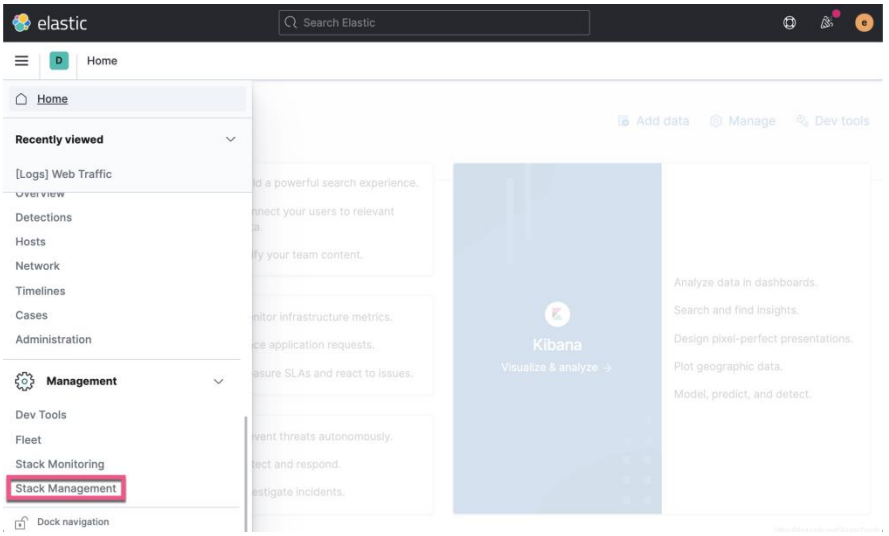
```
POST employee/_bulk
{"index":{"_index":"employee"}}
{"name":"user1","email":"user1@packt.com","salary":5000,"gender":"M","address1":"312 Main St","address2":"Walthill","state":"NE"}
{"index":{"_index":"employee"}}
{"name":"user2","email":"user2@packt.com","salary":10000,"gender":"F","address1":"5658 N Denver Ave","address2":"Portland","state":"OR"}
{"index":{"_index":"employee"}}
{"name":"user3","email":"user3@packt.com","salary":7000,"gender":"F","address1":"300 Quinterra Ln","address2":"Danville","state":"CA"}
```

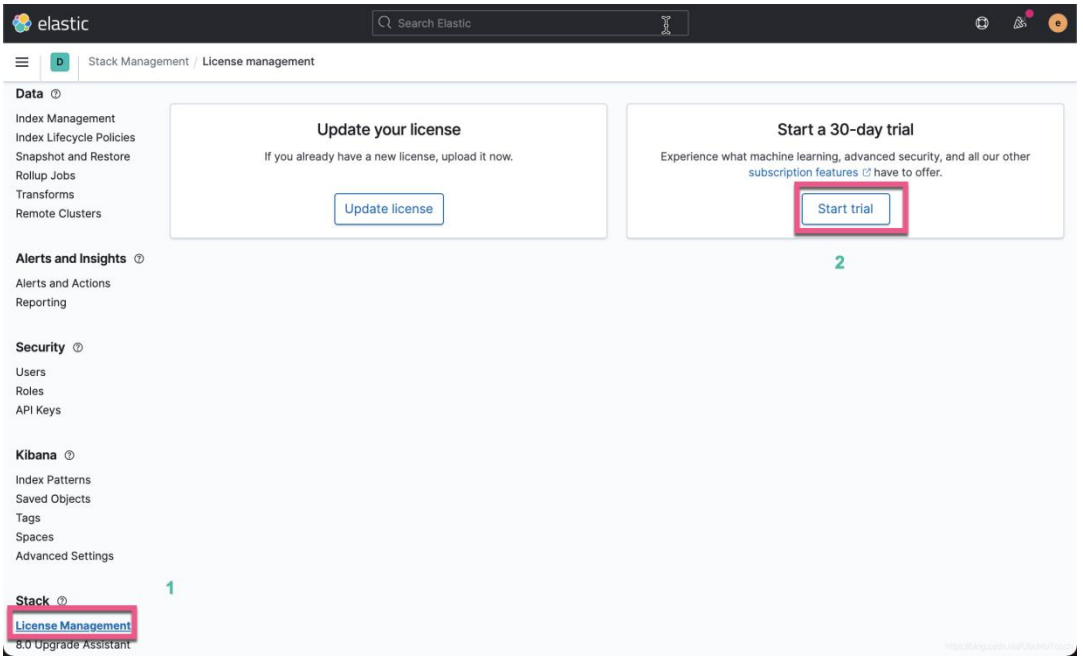
这样我们把三个文档存入到 employee 的索引之中。

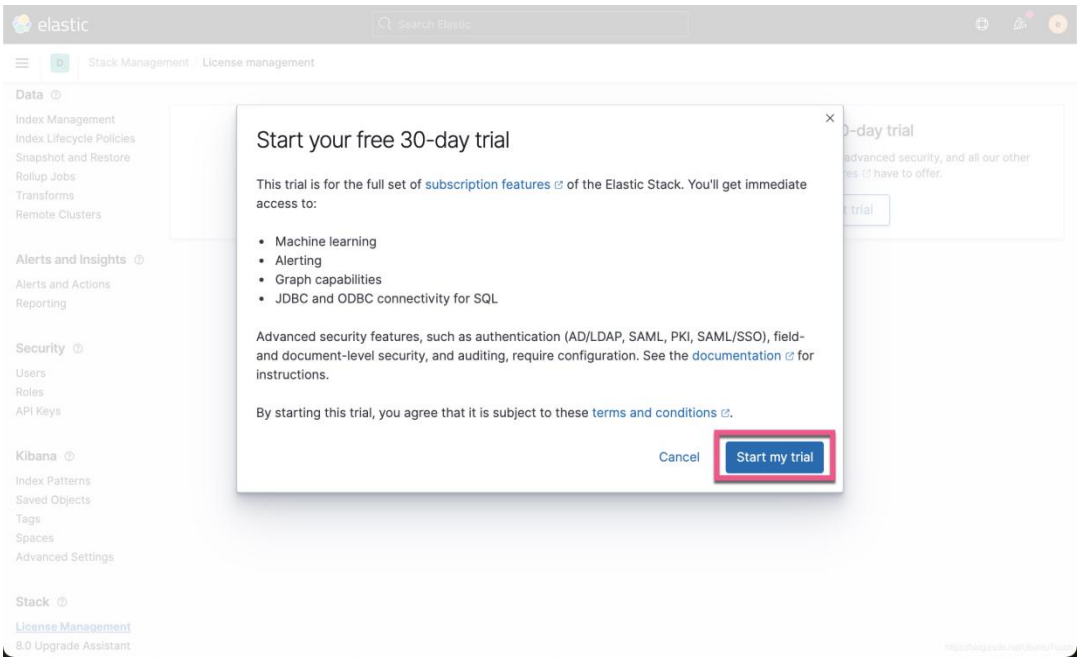


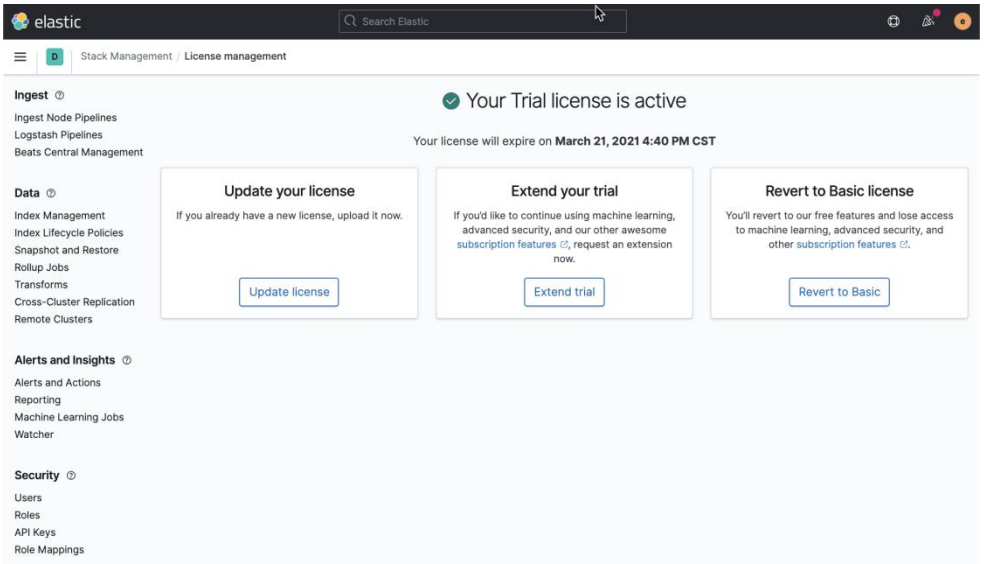
创建新的 role

由于字段级安全是白金版特有的功能，在一下的字段级安全配置中，我们需要启动白金版试用。更多有关关于订阅的信息，请参阅链接 https://www.elastic.co/cn/subscriptions. 对于不想使用字段级安全的用户来说，请忽略下面启动白金版试用的步骤。下面展示如何启动白金版试用：

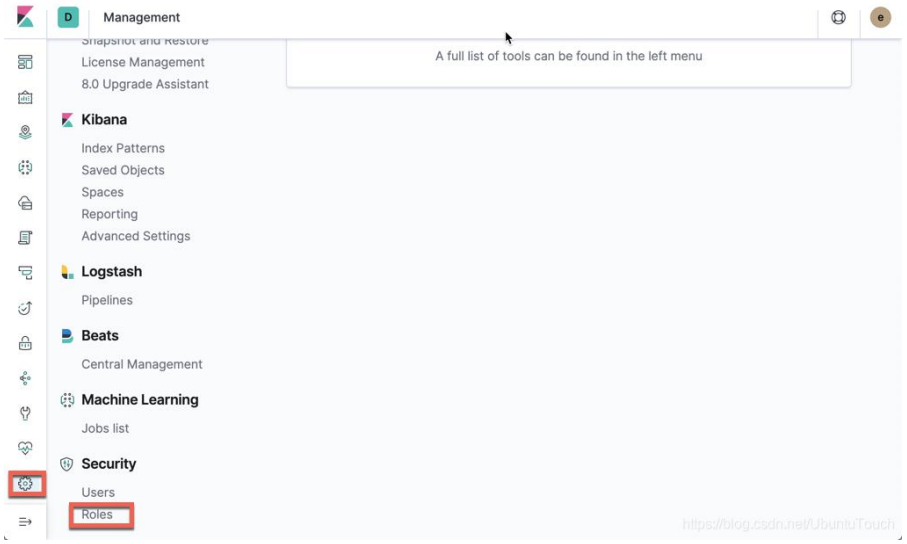



elastic
Search Elastic
Stack Management / License management
Data
Index Management
Index Lifecycle Policies
Snapshot and Restore
Rollup Jobs
Transforms
Remote Clusters
Alerts and Insights
Alerts and Actions
Reporting
Security
Users
Roles
API Keys
Kibana
Index Patterns
Saved Objects
Tags
Spaces
Advanced Settings
Stack
License Management
8.0 Upgrade Assistant
1
Update your license
If you already have a new license, upload it now.
Update license
Start a 30-day trial
Experience what machine learning, advanced security, and all our other subscription features have to offer.
Start trial
2

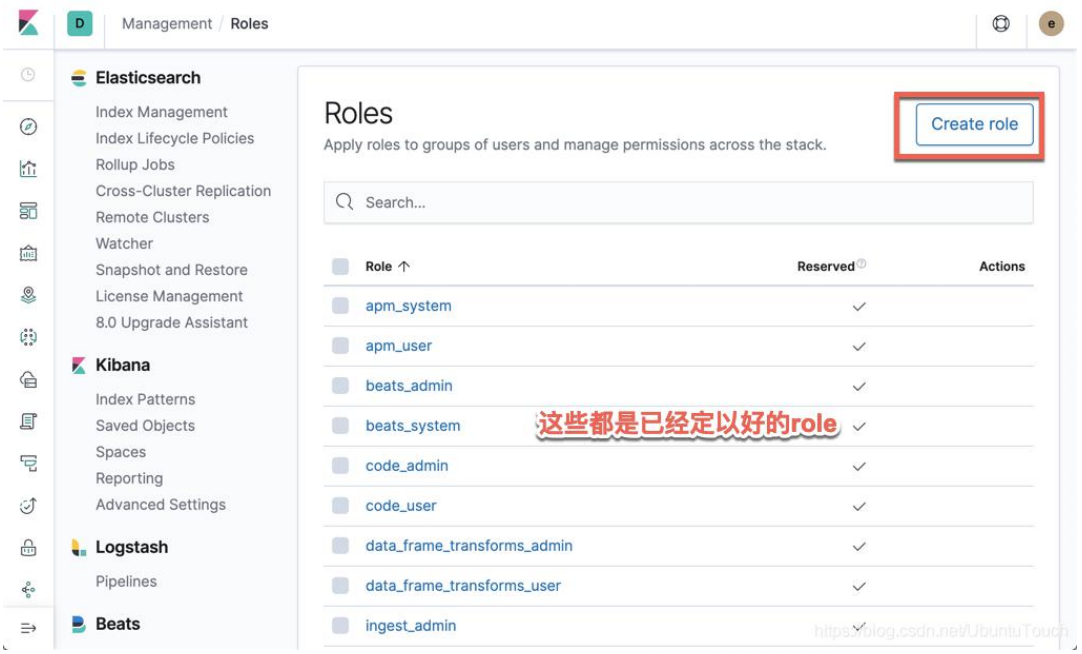

Start your free 30-day trial
This trial is for the full set of subscription features of the Elastic Stack. You'll get immediate access to:
Machine learning
Alerting
Graph capabilities
JDBC and ODBC connectivity for SQL
Advanced security features, such as authentication (AD/LDAP, SAML, PKI, SAML/SSO), field- and document-level security, and auditing, require configuration. See the documentation for instructions.
By starting this trial, you agree that it is subject to these terms and conditions.
Cancel
Start my trial



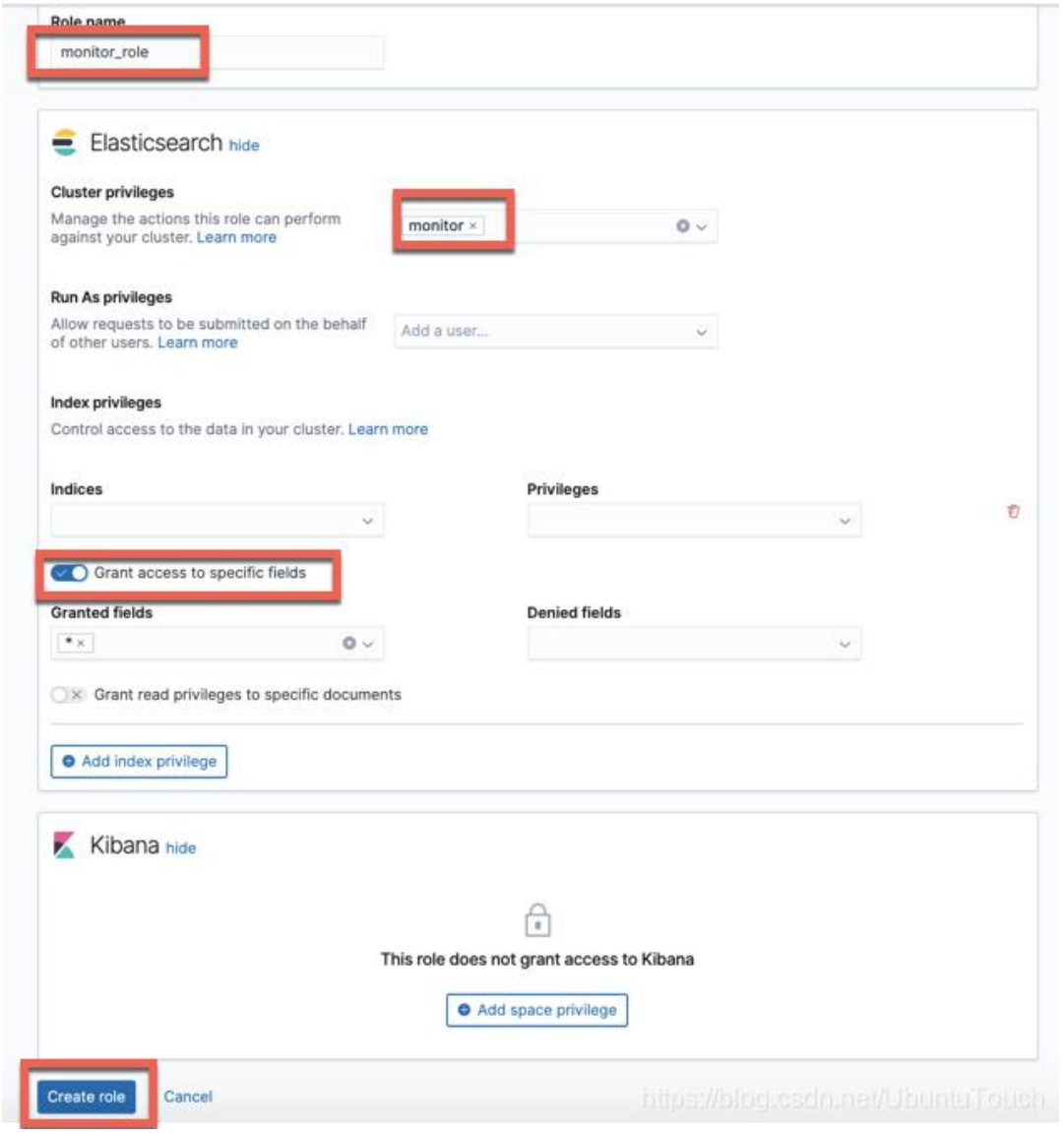
请注意：如下的操作是在 elastic 用户登录的情况下进行操作的。要创建新用户，请导航到 Stack Management 并在 “Security” 部分中选择 “role”，或者如果你当前在 “Users” 屏幕上，请单击 “Roles” 选项。 角色屏幕显示所有已定义/可用的角色:



当我们点击 roles 后：



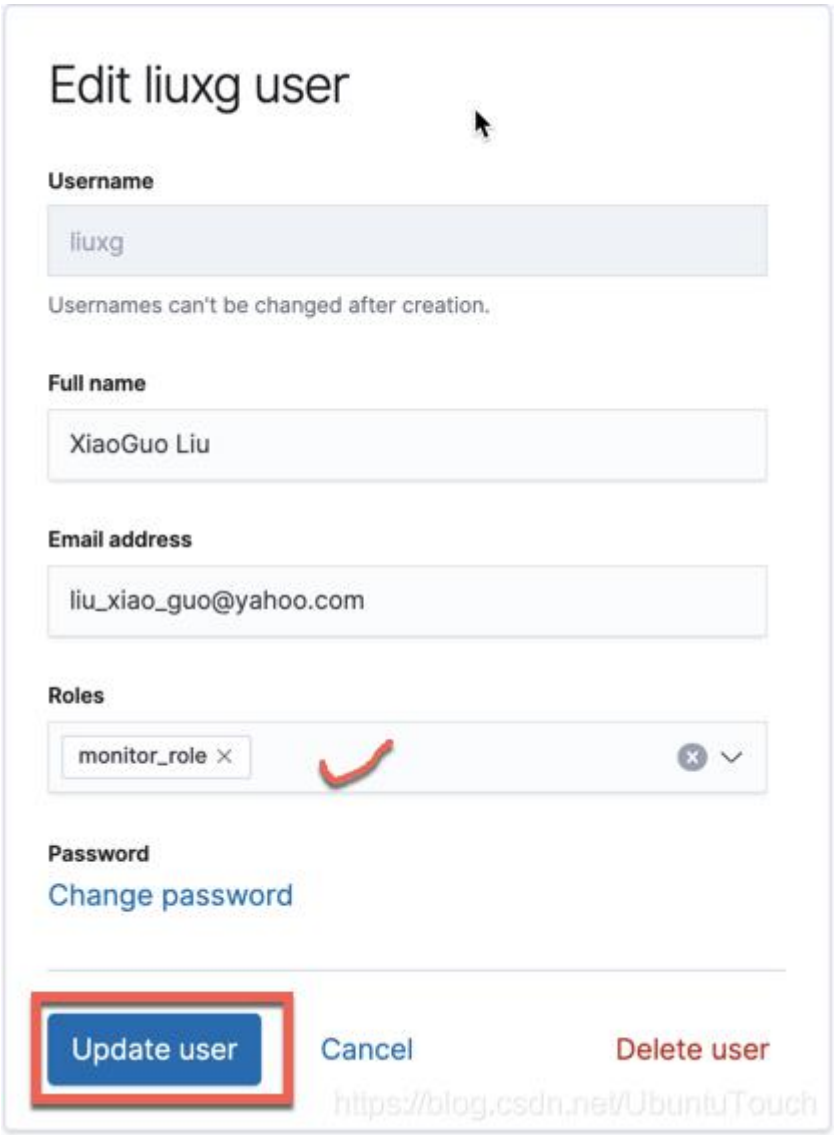
我们点击 Create role 按钮。



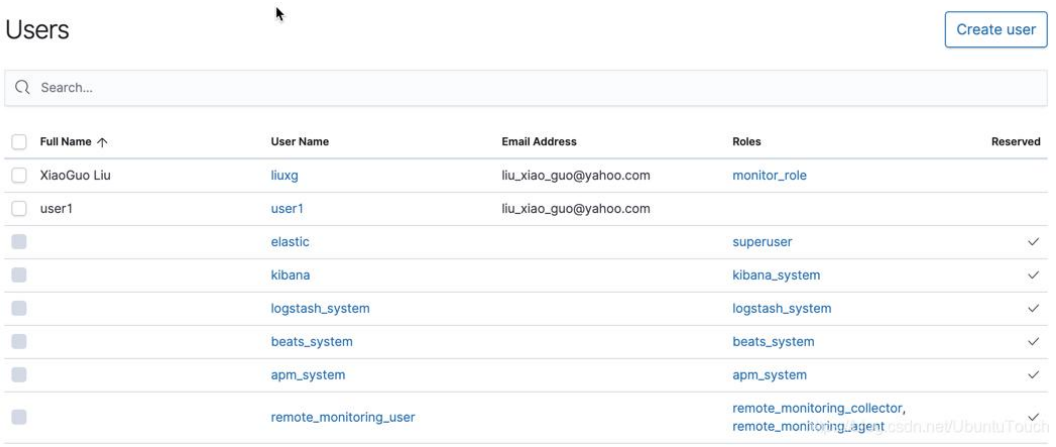
在这里，我们定义了一个叫做 monitor_role，它具有 monitor 的权限

把 role 赋予给用户

我们打开我们的用户列表。针对我的情况，我们打开 liuxg 用户：



我们修改 liuxg 账号的 Roles。把刚才创建的 monitor_role 赋予给 liuxg 用户。点击 Update User 按钮。这样我们的设定就好了。设定好的账号是这样的：



从上面，我们可以看出来 liuxg 账号是有 monitor_role 的，而 user1 账号是没有的。

下面我们来做一些基本的测试。我们在一个 terminal 中打入如下的命令：

```
curl -u liuxg:123456 "http://localhost:9200/_cluster/health?pretty"
```

注意这里的 123456 是 liuxg 的账号密码。执行上面的显示结果是：

```
}
liuxg-2:elasticsearch-7.4.0 liuxg$ curl -u liuxg:123456 "http://localhost:9200/_
cluster/health?pretty"
{
  "cluster_name" : "elasticsearch",
  "status" : "yellow",
  "timed_out" : false,
  "number_of_nodes" : 1,
  "number_of_data_nodes" : 1,
  "active_primary_shards" : 13,
  "active_shards" : 13,
  "relocating_shards" : 0,
  "initializing_shards" : 0,
  "unassigned_shards" : 4,
  "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" : 76.47058823529412
}
```


我们显然看到了结果。那么我们同样地对 User 1 账号来进行实验：

```
curl -u user1:123456 "http://localhost:9200/_cluster/health?pretty"
```

显示的结果是：

```
liuxg-2:elasticsearch-7.4.0 liuxg$ curl -u user1:123456 "http://localhost:9200/
cluster/health?pretty"
{
  "error" : {
    "root_cause" : [
      {
        "type" : "security_exception",
        "reason" : "action [cluster:monitor/health] is unauthorized for user [u
er1]"
      }
    ],
    "type" : "security_exception",
    "reason" : "action [cluster:monitor/health] is unauthorized for user [user1
"
  },
  "status" : 403
}
```

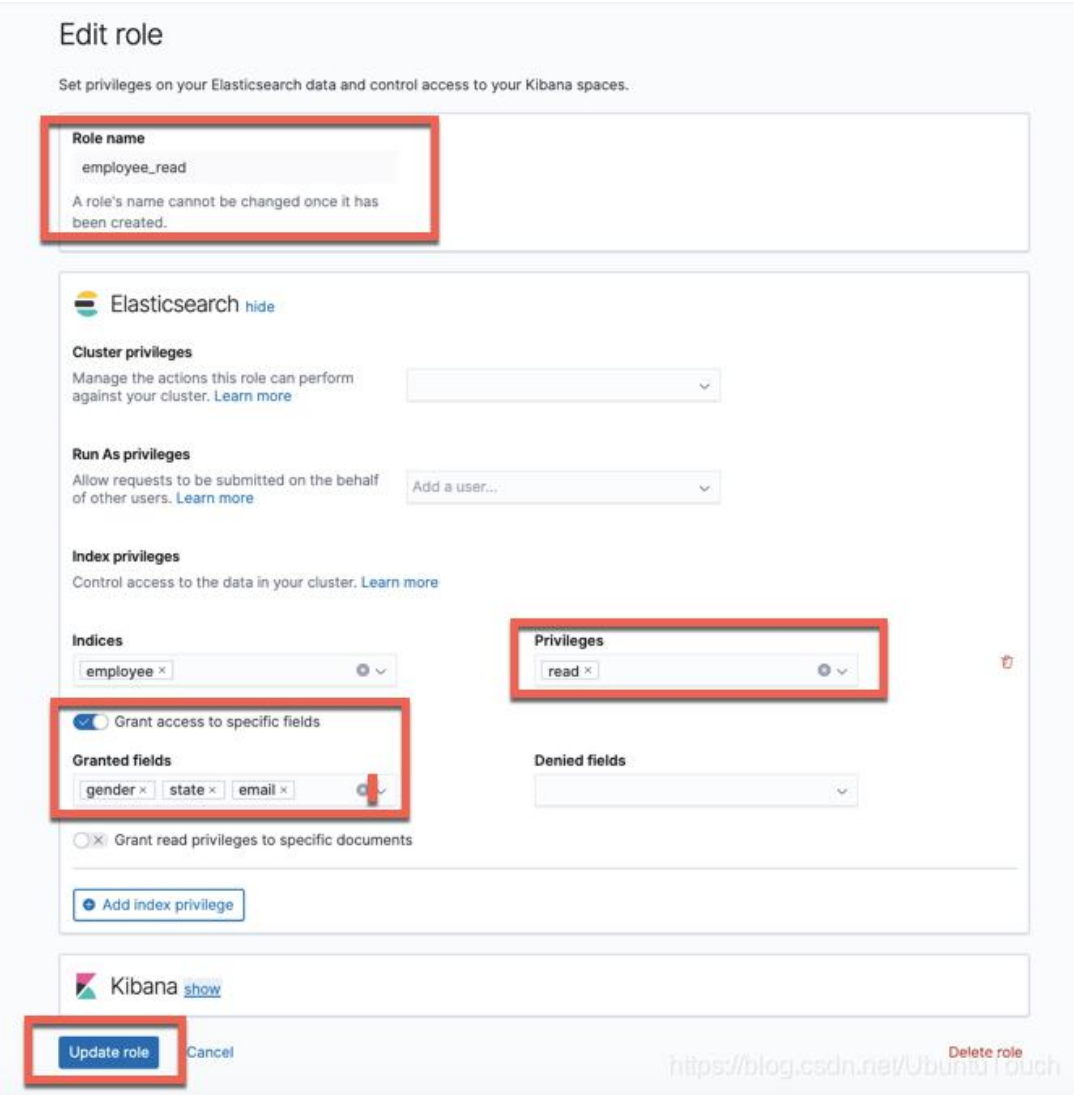
显然，user1 账号没有得到任何结果。这个根本的原因是因为这个账号没有相应的权限。

文档级或字段级安全

现在，我们知道了如何创建新用户，创建新角色以及将角色分配给用户，让我们探讨如何针对给定的索引/文档对文档和字段施加安全性。接下来，我们使用我之前给大家输入进的 employee 索引来展示。

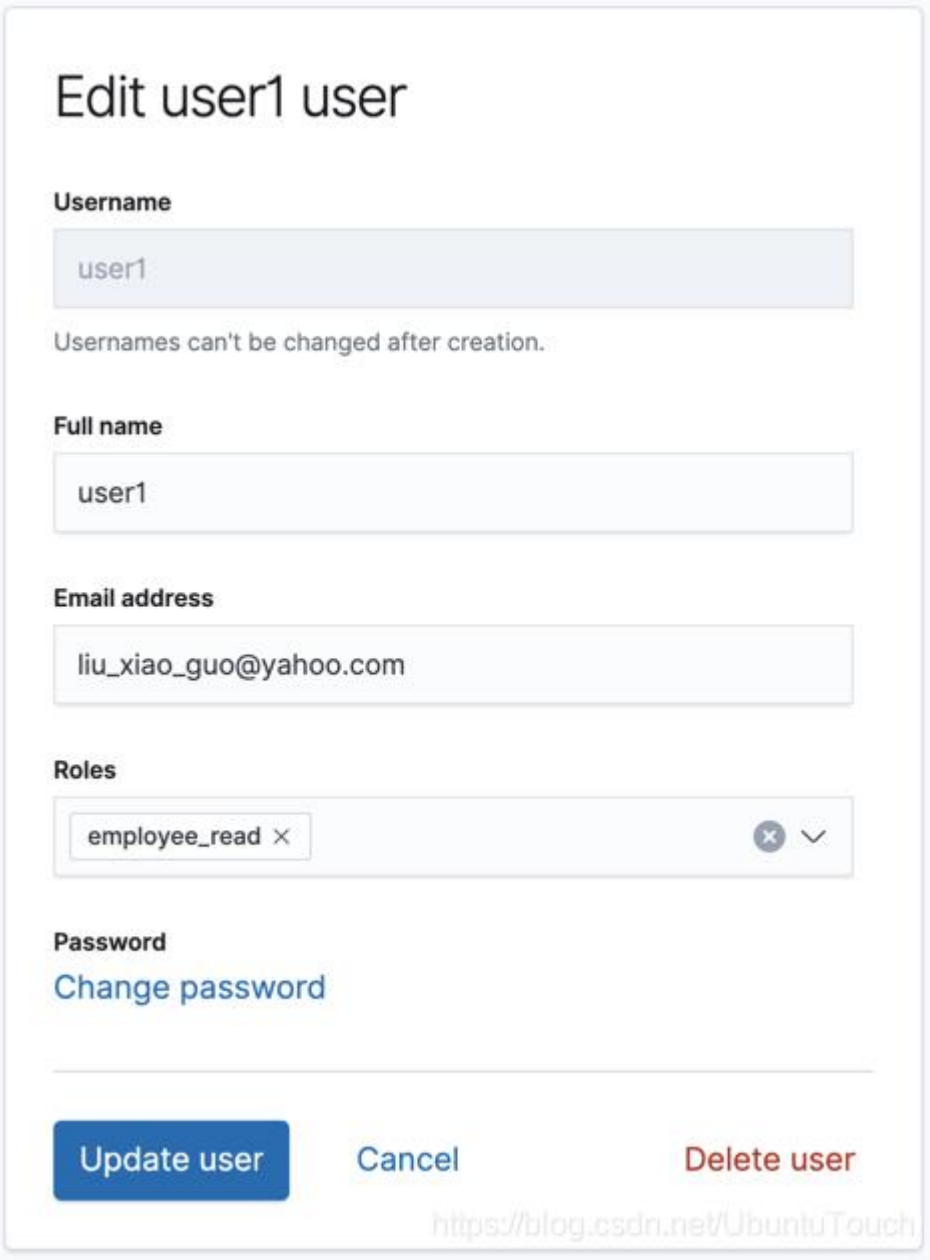
案例 1

当用户搜索员工详细信息时，该用户不允许包含在属于员工索引的文档中的薪水/地址详细信息。这就是我们所说的字段级安全。首先，让我们来创建一个叫做 employee_read 的 role。这个 role 只具有 employee 索引的 read 权限。为了限制字段，我们可以在设置里做相应的配置：

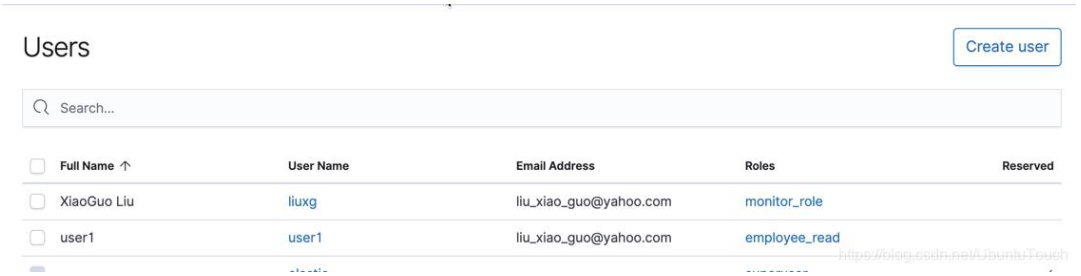


我们只允许这个 employee_read role 访问 gender，state 及 email 字段，而且只有 read 权限。

运用我们刚才设置的 employee_read role，我们赋予给我们的 user1 用户：


Edit user1 user
Username
user1
Usernames can't be changed after creation.
Full name
user1
Email address
liu_xiao_guo@yahoo.com
Roles
employee_read ×
Password
Change password
Update user
Cancel
Delete user
https://blog.csdn.net/UbuntuTouch

设置好的用户界面为：



上面显示我们的 user1 具有 employ_read 的 role。

在我们的一个 terminal 里打入如下的命令：

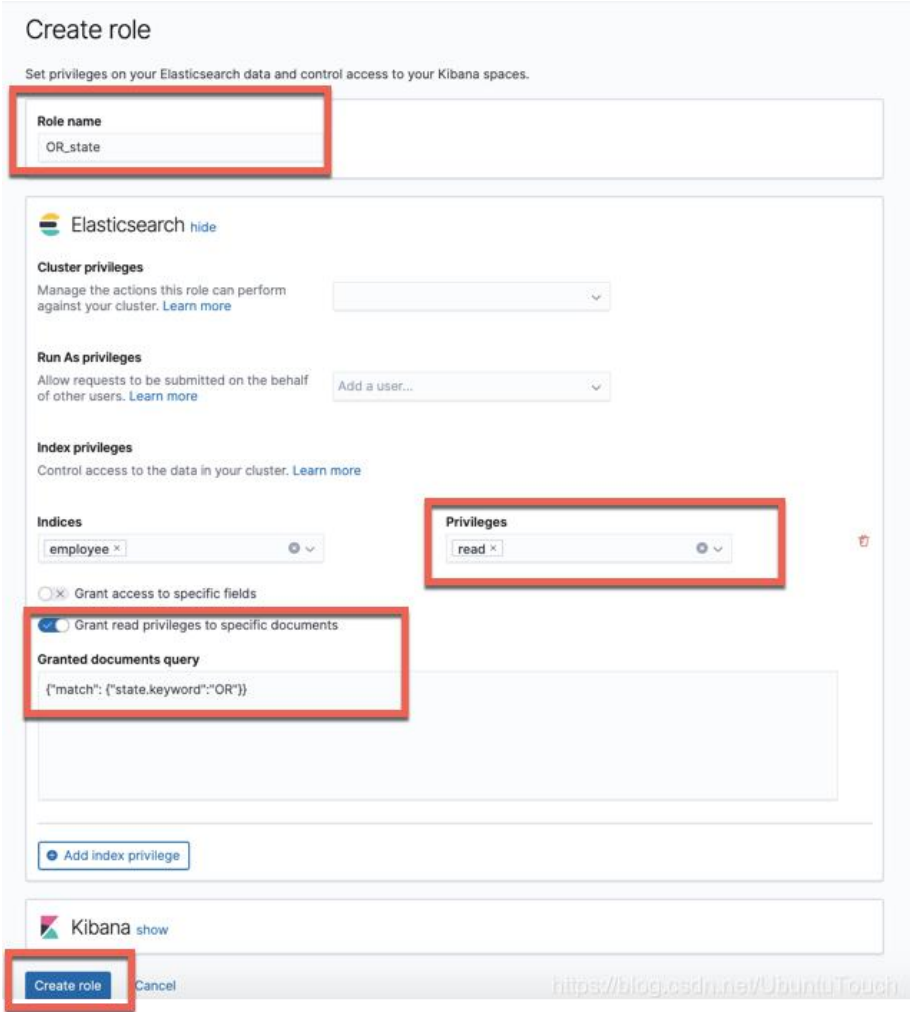
```
curl -u user1:123456 "http://localhost:9200/employee/_search?pretty"
```

请注意：这里的 123456 是 user1 用户的密码。上面命令显示的结果为：

显然，user1 只能访问在 employee_read 中的三个字段。

案例 2

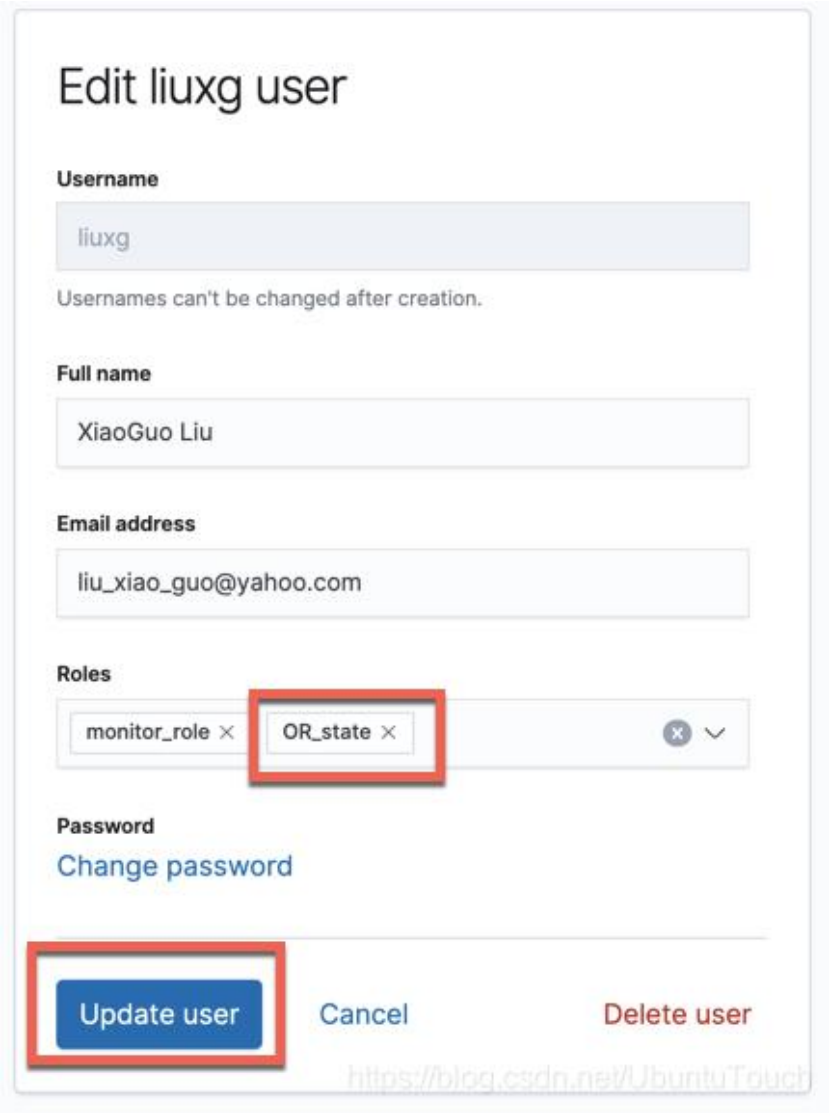
我们想定义一个 role。这个 role 具有 read 的权限，并且只能访问 state 为 OR 的那些文档。我们做一下的设置：



我们创建了一个叫做 OR_state 的 role。它通过一个 query:

```
{"match": {"state.keyword":"OR"}}
```

来匹配项对应的文档。我们接着把这个 role 赋予给 liuxg 用户：



在我们设置完后，我们接着在一个 terminal 中打入如下的命令：

```
curl -u liuxg:123456 "http://localhost:9200/employee/_search?pretty"
```

显示的结果：

```
liuxg-2:elasticsearch-7.4.0 liuxg$ curl -u liuxg:123456 "http://localhost:9200/e
mployee/_search?pretty"
{
  "took" : 0,
  "timed_out" : false,
  "_shards" : {
    "total" : 1,
    "successful" : 1,
    "skipped" : 0,
    "failed" : 0
  },
  "hits" : {
    "total" : {
      "value" : 1,
      "relation" : "eq"
    },
    "max_score" : 1.0,
    "hits" : [
      {
        "_index" : "employee",
        "_type" : "_doc",
        "_id" : "fkSLF24BajwQPgcY-55_",
        "_score" : 1.0,
```


我们可以看出来这次的显示的结果只有一个，而且这个文档的 state 是 OR。

3.4.1.6 配置多节点集群

创作人：刘晓国

如何使用 Elastic Helm Chart 来部署一个多节点的 Elasticsearch 集群。

Elastic Helm Chart:https://github.com/elastic/helm-charts/

安装

安装 VM

在今天的安装中，我选择使用 virtualbox 来部署我们的 ECK。我们也可以选择 Docker 及其它的容器来进行。你可以使用如下的命令

```
brew install virtualbox
```

安装 Minikube 及 kubectl

针对不同版本的 MasOS，安装的方法可能不能。对于有些版本的 MacOS 来说，你需要打入如下的命令：

```
brew cask install minikube
```

或者：

```
brew install minikube
```

一般来说在安装 minikube 的过程中，它也会同时帮我们安装 kubectl。如果 kubectl 命令不能正常被执行，那么你可以尝试使用如下的命令来进行安装：

```
brew install kubernetes-cli
```

当我们把 minikube 及 kubectl 完全进行安装后，我们可以通过打入如下的命令来检查安装是否已经完成：

```
$ minikube version
minikube version: v1.9.0
commit: 48fefd43444d2f8852f527c78f0141b377b1e42a
```

如果你能看到上面的信息，则表明我们的 minikube 的安装时正确的。同样：

```
$ kubectl version
Client Version: version.Info{Major:"1", Minor:"15", GitVersion:"v1.15.5", GitCommit:"20c265fef0741dd71a66480e35bd69f18351daea", GitTreeState:"clean", BuildDate:"2019-10-15T19:16:51Z", GoVersion:"go1.12.10", Compiler:"gc", Platform:"darwin/amd64"}
Server Version: version.Info{Major:"1", Minor:"16", GitVersion:"v1.16.0", GitCommit:"2bd9643cee5b3b3a5ecbd3af49d09018f0773c77", GitTreeState:"clean", BuildDate:"2019-09-18T14:27:17Z", GoVersion:"go1.12.9", Compiler:"gc", Platform:"linux/amd64"}
```

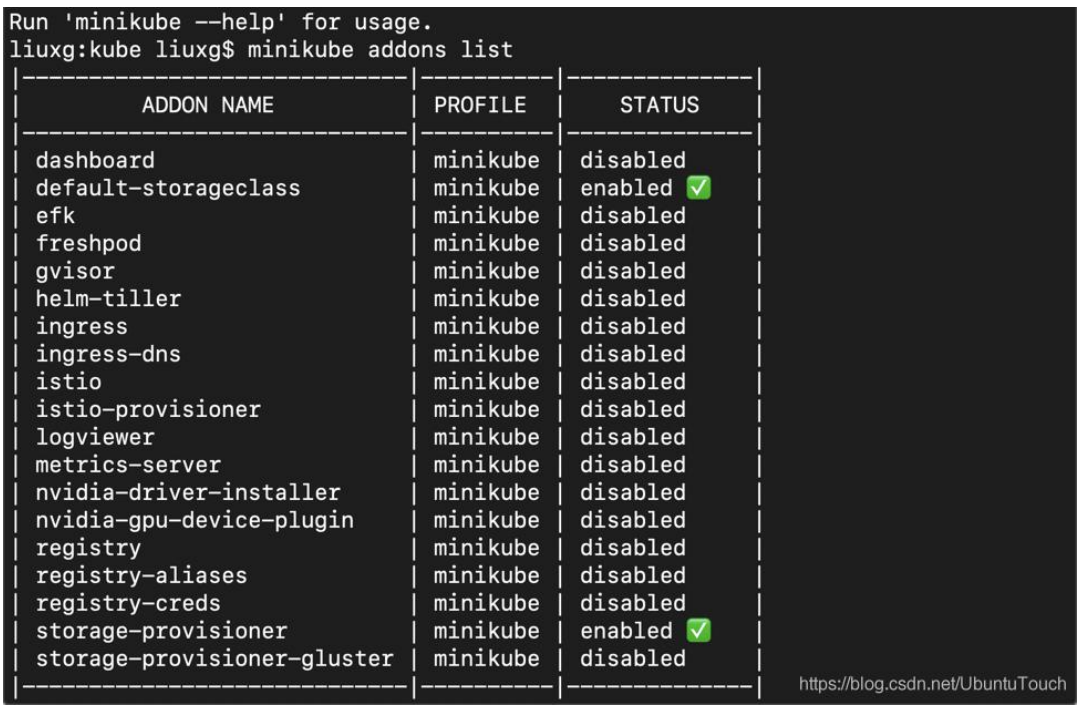
我们可以打入上面的命令来检查 kubectl 的版本信息。

目前 ECK 对于 kubernetes 的要求是：

- kubectl 1.11+
- Kubernetes 1.12+ or OpenShift 3.11+
- Elastic Stack: 6.8+, 7.1+

一旦 minikube 安装好后，我可以通过如下的命令来检查哪些 plugin 已经被加载

```
minikube addons list
```



我们可以通过如下的命令来 enable 一个 plugin:

```
minikube addons enable dashboard
```

或者使用如下的命令来 disable 一个 plugin:

```
minikube addons disable dashboard
```

安装 Helm

我们可以使用如下的命令来安装 Helm:

```
brew install helm
```

针对其它的操作系统，请参照链接：https://helm.sh/docs/intro/install/。

一旦完成我们们上面的安装后，我们也就可以开始来部署我们的 ECK 了。

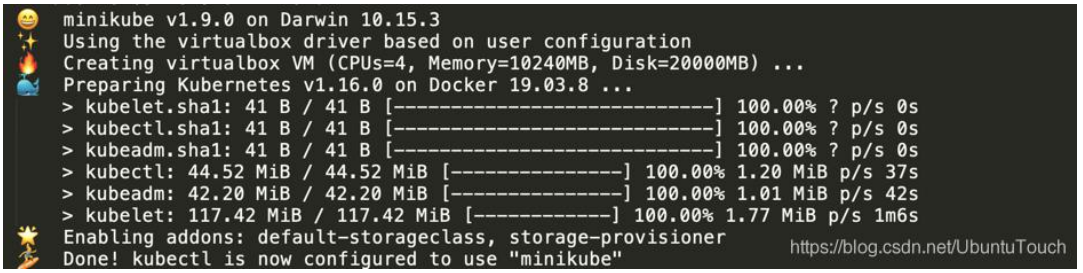
使用 Elastic Helm Chart 安装 Elasticsearch 集群

首先我们启动 minikube。在启动 minikube 之前，我们需要对 minikube 的启动参数来进行配置。我是这样来启动我的 minikube 的：

```
minikube start --driver=virtualbox --cpus 4 --memory 10240 --kubernetes-version 1.16.0
```

在上面，我使用 virtualbox 来作为驱动。你也可以使用 docker 来启动。在这里我配置了 cpu 的个数是 4，内存是 10G。同时由于一个一致的 bug，目前 minikube 不能和最新的 kubernetes 一起进行安装工作。在上面我特别指出了 kubernetes 的版本信息为 1.16.0。

上面的命令的输出为：



对于中国的开发者来说，由于网路的限制，那么在使用上面的命令时，可能会出现 k8s.gcr.io 网址不能被访问的情况。那么我们怎么解决这个问题呢？答案是我们可以使用使用阿里云服务器。我们可以尝试使用如下的命令：

```
$ minikube start --help |grep repo

        --image-repository='': Alternative image repository to pull docker images from. This can be used when you have limited access to gcr.io. Set it to "auto" to let minikube decide one for you. For Chinese mainland users, you may use local gcr.io mirrors such as registry.cn-hangzhou.aliyuncs.com/google_containers
```

根据上面的提示，我们可以修改我们上面的命令为：

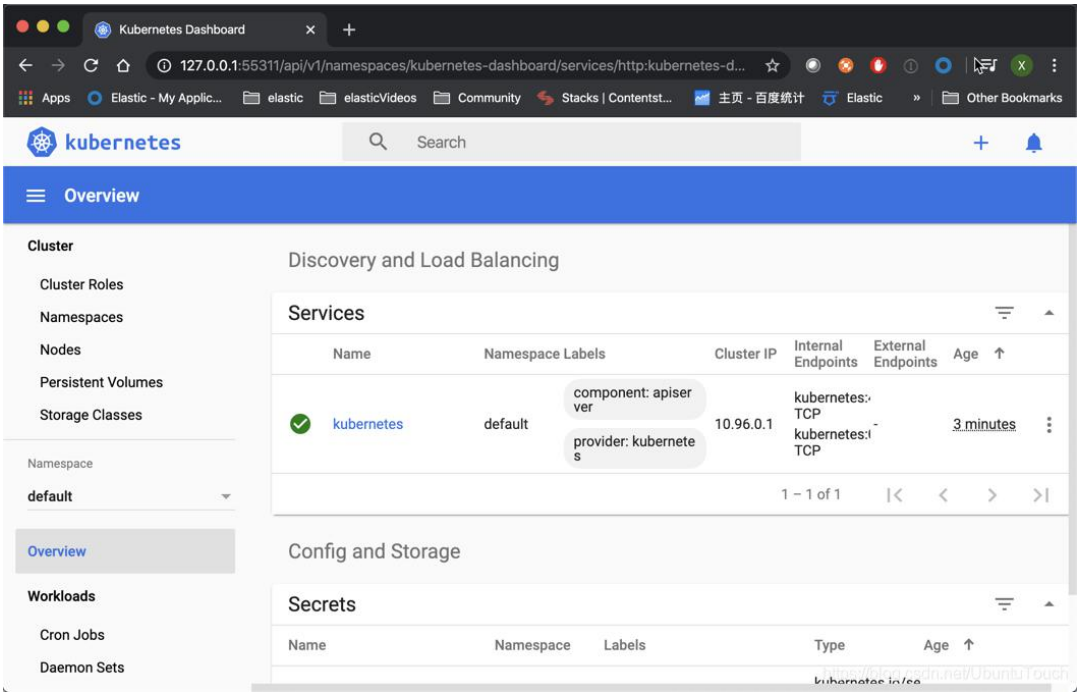
```
$ minikube start --driver=virtualbox --cpus 4 --memory 10240 --kubernetes-version 1.16.0 --image-repository='registry.cn-hangzhou.aliyuncs.com/google_containers'
```

那么根据上面的命令运行后的结果为：

一旦 minikube 被成功启动起来，我们可以使用如下的命令：

```
minikube dashboard
```

当上面的命令执行后，它就会启动一个 web 的接口让我们来对 kubernetes 监控和管理：



等我们上面的 Minikube 已经被启动后，我们使用如下的命令：

```
helm repo add elastic https://helm.elastic.co
```

上面的命令返回：

```
$ helm repo add elastic https://helm.elastic.co
"elastic" has been added to your repositories
```

接着我们使用如下的命令：

```
curl -O https://raw.githubusercontent.com/elastic/helm-charts/master/elasticsearch/examples/minikube/values.yaml
$ curl -O https://raw.githubusercontent.com/elastic/helm-charts/master/elasticsearch/examples/minikube/values.yaml
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   478  100   478    0     0    810      0 --:--:-- --:--:-- --:--:--   808
```

上面的 values.yaml 的内容如下：

```
---
# Permit co-located instances for solitary minikube virtual machines.
antiAffinity: "soft"

# Shrink default JVM heap.
esJavaOpts: "-Xmx128m -Xms128m"

# Allocate smaller chunks of memory per pod.
resources:
  requests:
    cpu: "100m"
    memory: "512M"
  limits:
    cpu: "1000m"
    memory: "512M"

# Request smaller persistent volumes.
volumeClaimTemplate:
```

```
  accessModes: [ "ReadWriteOnce" ]
  storageClassName: "standard"
  resources:
    requests:
      storage: 100M
```

我们使用 helm 来对我们的 Elasticsearch 进行安装：

```
helm install elasticsearch elastic/elasticsearch -f ./values.yaml
```

上面的命令的输出为：

```
$ helm install elasticsearch elastic/elasticsearch -f ./values.yaml
NAME: elasticsearch
LAST DEPLOYED: Sun Apr  5 19:04:46 2020
NAMESPACE: default
STATUS: deployed
REVISION: 1
NOTES:
1. Watch all cluster members come up.
  $ kubectl get pods --namespace=default -l app=elasticsearch-master -w
2. Test cluster health using Helm test.
  $ helm test elasticsearch --namespace=default
```

我们使用上面显示的命令来进行监控：

```
kubectl get pods --namespace=default -l app=elasticsearch-master -w
$ kubectl get pods --namespace=default -l app=elasticsearch-master -w
NAME                     READY   STATUS    RESTARTS   AGE
elasticsearch-master-0   0/1     Running   0          98s
elasticsearch-master-1   0/1     Running   0          98s
elasticsearch-master-2   0/1     Running   0          98s
elasticsearch-master-1   1/1     Running   0          106s
elasticsearch-master-2   1/1     Running   0          110s
elasticsearch-master-0   1/1     Running   0          116s
```

上面显示我们的 Elasticseach 已经被成功地部署好了。

接下来，我们来部署我们的 Kibana：

```
helm install kibana elastic/kibana
```

我们可以通过如下的命令来监控 Kibana 的创建过程：

```
kubectl get pods -w
$ kubectl get pods -w
NAME                             READY   STATUS              RESTARTS   AGE
elasticsearch-master-0           1/1     Running             0          4m12s
elasticsearch-master-1           1/1     Running             0          4m12s
elasticsearch-master-2           1/1     Running             0          4m12s
kibana-kibana-69f5ddd4bd-ql4s7   0/1     ContainerCreating   0          26s
kibana-kibana-69f5ddd4bd-ql4s7   0/1     Running             0          60s
```

如果我们看到 Kibana 的状态已经变为 Running，则表明我们的 Kibana 的创建是成功的。

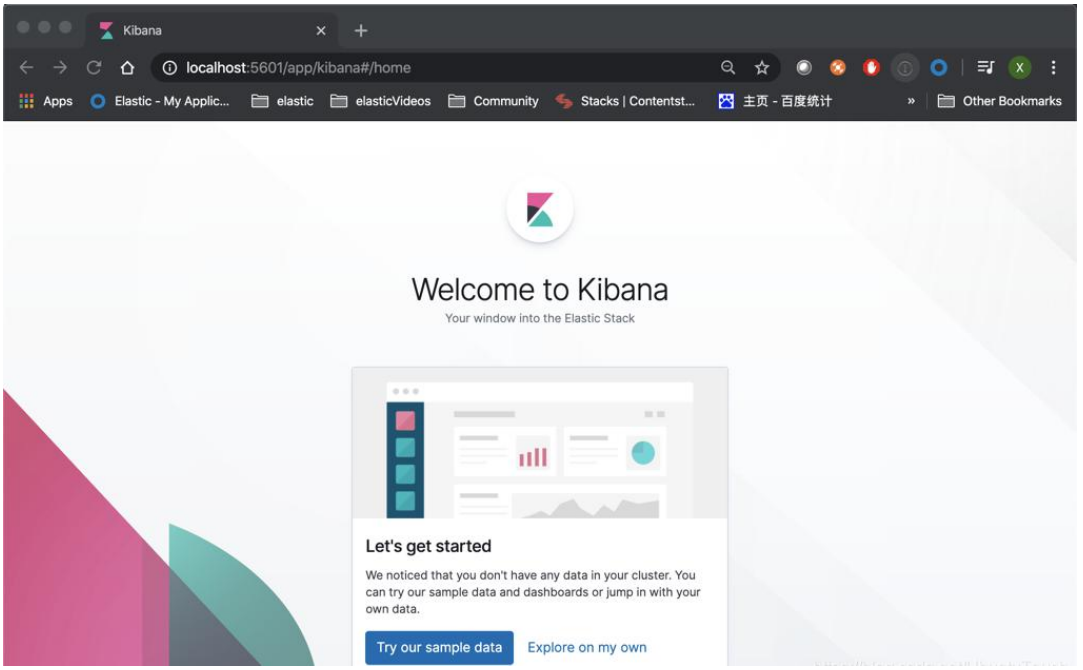
最后，我们需要把 Kibana 的 5601 口进行 port forward：

```
kubectl port-forward deployment/kibana-kibana 5601
```

上面命令的运行结果为：

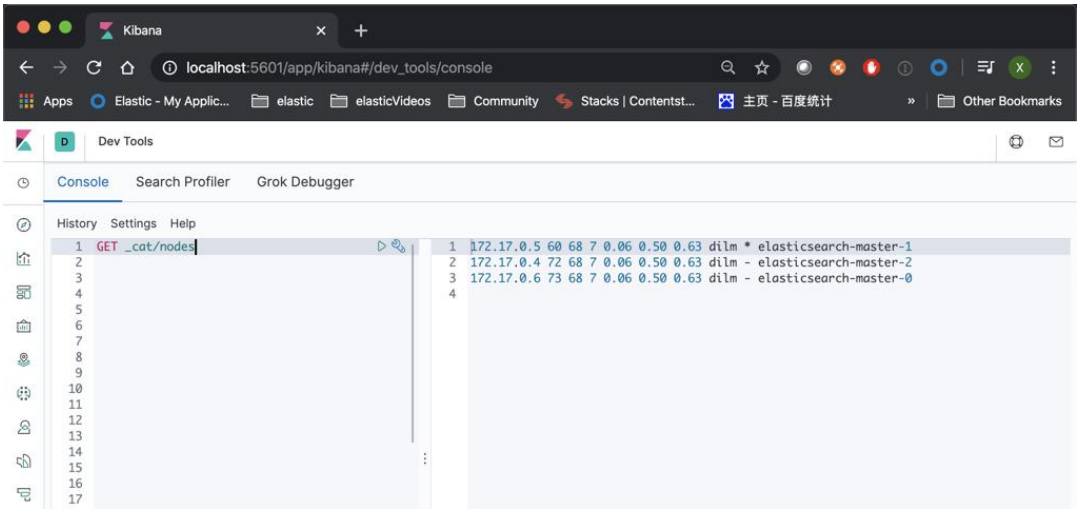
```
$ kubectl port-forward deployment/kibana-kibana 5601
Forwarding from 127.0.0.1:5601 -> 5601
Forwarding from [::1]:5601 -> 5601
Handling connection for 5601
```

这样我们的 Elasticsearch 已经被成功部署了。我们启动浏览器，并输入地址 localhost:5601



上面显示我们的 Elasticsearch 已经被成功部署了。

我们使用 Dev tools：



从上面我们可以看出来，我们已经成功地部署了 3 个 Elasticsearch 节点的集群。

如果你想了解 helm 更多，你可以使用如下的命令：

```
helm --help
```

helm 目前有很多的命令:

```
Available Commands:
  completion  Generate autocompletions script for the specified shell (bash or zsh)
  create      create a new chart with the given name
  dependency  manage a chart's dependencies
  env         Helm client environment information
  get         download extended information of a named release
  help        Help about any command
  history     fetch release history
  install     install a chart
  lint        examines a chart for possible issues
  list        list releases
  package     package a chart directory into a chart archive
  plugin      install, list, or uninstall Helm plugins
  pull        download a chart from a repository and (optionally) unpack it in local di
rectory
  repo        add, list, remove, update, and index chart repositories
  rollback    roll back a release to a previous revision
  search      search for a keyword in charts
  show        show information of a chart
```

```
  status          displays the status of the named release
  template        locally render templates
  test            run tests for a release
  uninstall       uninstall a release
  upgrade         upgrade a release
  verify          verify that a chart at the given path has been signed and is valid
  version         print the client version information
```

我们可以通过命令：

```
helm list
```

来显示已经发布了的：

```
$ helm list
NAME                NAMESPACE       REVISION        UPDATED
STATUS          CHART                       APP VERSION
elasticsearch       default         1               2020-04-05 19:04:46.629021 +0800 CST      depl
oyed        elasticsearch-7.6.2     7.6.2
kibana              default         1               2020-04-05 19:08:32.566291 +0800 CST      de
ployed      kibana-7.6.2            7.6.2
```

我们也可以使用如下的命令来卸载已经发布了的：

```
helm uninstall
```

比如：

```
$ helm uninstall kibana
release "kibana" uninstalled
liuxg:kube liuxg$ helm uninstall elasticsearch
release "elasticsearch" uninstalled
liuxg:kube liuxg$ kubectl get pods
No resources found.
```

3.4.1.7 阿里云 Elasticsearch 服务

编辑：大禹

本文将基于阿里云 Elasticsearch，通过快速创建、访问实例，并使用 RESTful API，完成创建索引、创建文档、插入数据、搜索数据、删除索引等操作，从而体验 Cloud 云服务。

阿里云 Elasticsearch 兼容 Elastic Stack 功能，提供免运维全托管服务的弹性云搜索与分析引擎，致力于数据库加速、数据分析、信息检索、智能运维监控等场景服务；独有的云原生高性能内核、达摩院 NLP 分词、向量检索、智能运维、免费 X-Pack 高级商业特性等能力，全面提升企业应用效率，降低成本。

前提条件

- 注册阿里云账号。

  具体操作，请参见账号注册：https://www.aliyun.com/product/bigdata/product/elasticsearch?utm_content=g_1000275202

- 创建专有网络和虚拟交换机。

  具体操作，请参见搭建 IPv4 专有网络：https://help.aliyun.com/document_detail/65430.htm#task-1512598

- 完成规格容量评估。

具体操作，请参见规格容量评估：https://help.aliyun.com/document_detail/72660.htm#concept-dq4-bmk-zgb

实践背景

某金融服务企业通过线上平台管理理财项目，之前使用传统数据库为客户提供理财产品的搜索功能。由于收益信用良好，得到了不少客户的青睐，但随之而来的是业务体系逐步扩大，客户信息也逐步增加，使得传统型数据库的缺陷越来越明显。为了改善搜索响应迟缓、精准性无法保障、数据服务设备性能降低等状况，该企业引入了阿里云 Elasticsearch 产品，为用户提供理财产品的搜索功能。阿里云 Elasticsearch 有效地解决了之前传统数据库存在的问题，同时提升了客户满意度。本文以此场景为例，为你介绍如何使用阿里云 Elasticsearch 搭建集群和搜索数据。

本场景假设该企业的理财产品信息如下所示。

```
{
"products":[
{"productName":"大健康天天理财","annual_rate":"3.2200%","describe":"180 天定期理财，最低 20000 起投，收益稳定，可以自助选择消息推送"}
{"productName":"西部通宝","annual_rate":"3.1100%","describe":"90 天定投产品，最低 10000 起投，每天收益到账消息推送"}
{"productName":"安详畜牧产业","annual_rate":"3.3500%","describe":"270 天定投产品，最低 40000 起投，每天收益立即到账消息推送"}
{"productName":"5G 设备采购月月盈","annual_rate":"3.1200%","describe":"90 天定投产品，最低 12000 起投，每天收益到账消息推送"}
```

{"productName":"新能源动力理财","annual rate":"3.0100%","describe":"30 天定投产品推荐，最低 8000 起投，每天收益会消息推送"}
{"productName":"微贷赚","annual_rate":"2.7500%","describe":"热门短期产品，3 天短期，无须任何手续费用，最低 500 起投，通过短信提示获取收益消息"}
]
}

步骤一：创建实例

前往实例创建页面：https://www.aliyun.com/product/bigdata/product/elasticsearch

在开通页面的前四个配置页面，完成实例启动配置。本教程使用的配置如下，未提及的配置保持默认。

配置页面	配置项	示例	说明
基础配置	付费模式	按量付费	在前期程序研发或功能测试期间，建议购买按量付费类型的实例测试。说明 开通包年包月类型的实例可以享受优惠条件。
	选择服务	通用商业版，7.10	阿里云 Elasticsearch 通用商业版包含全部 X-pack 高级特性，致力于数据分析和数据搜索等场景服务。

配置页面	配置项	示例	说明
集群配置	地域和可用区		阿里云 Elasticsearch 支持的地域和可用区，请参见地域和可用区：https://help.aliyun.com/document_detail/97672.htm#section-iaw-kz6-ha2 注意 所选可用区下必须存在专有网络和虚拟交换机。
	可用区数量	单可用区	
	实例规格	数据节点：	新用户可享受 2C4G 的首月 30 天免费试用
网络及系统配置	网络类型	专有网络	默认为专有网络，不可更改。
	专有网络	tf-testAcccn-hangzhou3274,vpc-bp16k1dvzxtmagcva**	选择对应区域下的专有网络。

配置页面	配置项	示例	说明
	虚拟交换机	tf-testAcccn-hangzhou3274 / vsw-bp1k4ec6s7sjdbudw**	只能显示所选专有网络中，与实例在相同可用区下的虚拟交换机。
	登录名	elastic	默认为 elastic，不可更改。
	登录密码	自定义密码	请记录该配置，在登录 Kibana 控制台时，需要输入该密码。
	场景初始化配置	通用场景	选择后，对应模板的配置会自动应用到集群中。

单击下一步：确认订单，然后预览实例配置。

本教程的实例配置预览如下图。

- 勾选服务协议，单击立即购买。
- 提示开通成功后，单击管理控制台，进入阿里云 Elasticsearch 的控制台概览页面。
- 在左侧导航栏，单击 Elasticsearch 实例。
- 在顶部菜单栏，选择资源和地域，然后在实例列表中查看创建成功的阿里云 Elasticsearch 实例。

步骤二：访问实例

等待实例状态变为正常，即可执行以下步骤，通过 Kibana 访问实例。

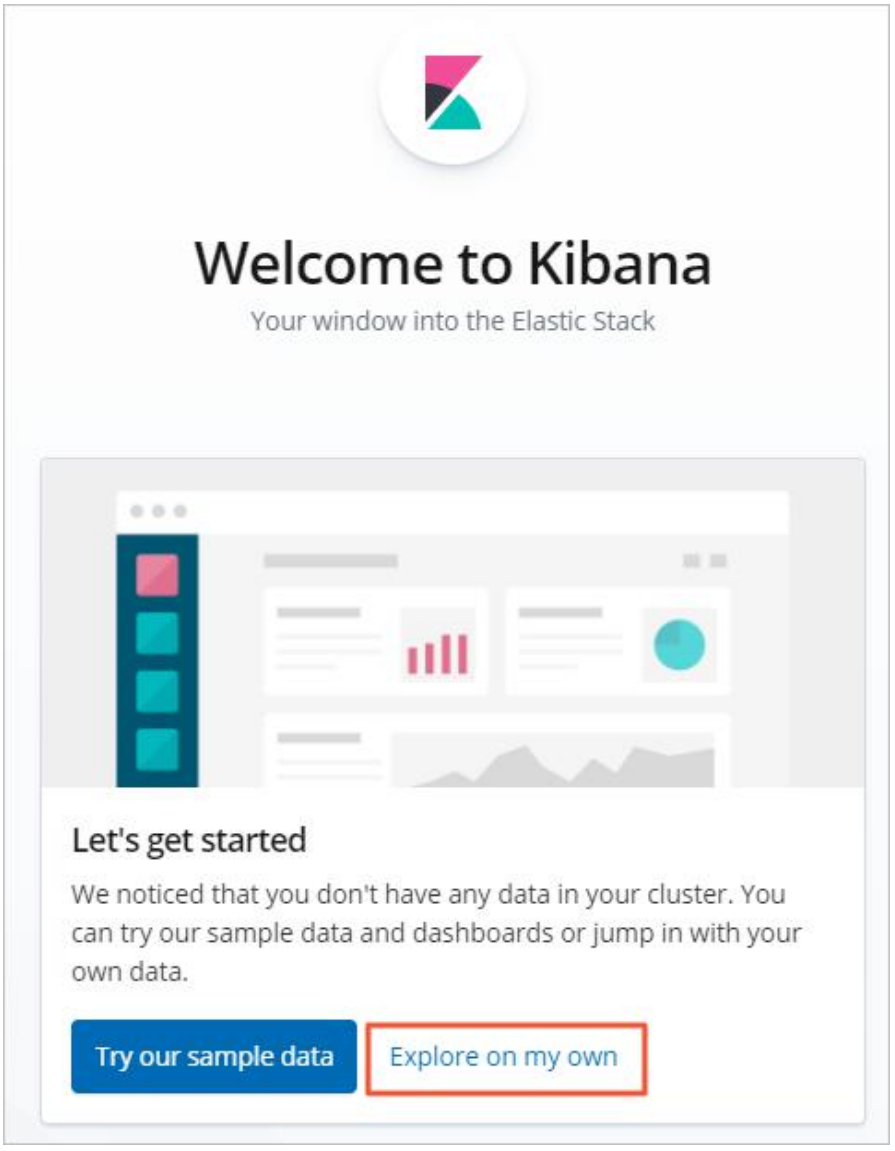
你也可以通过 curl 命令和客户端方式访问实例。

具体操作，请参见快速访问与配置：https://help.aliyun.com/document_detail/134862.htm?spm=a2c4g.11186623.2.21.33de9184Q0R30l#section-dy2-a9s-1ym

- 在实例列表中，单击目标实例 ID。
- 在左侧导航栏，单击可视化控制。
- 在 Kibana 区域中，单击进入控制台。
- 在登录页面输入账号和密码，单击登录。账号为 elastic，密码为你创建实例时设置的密码。

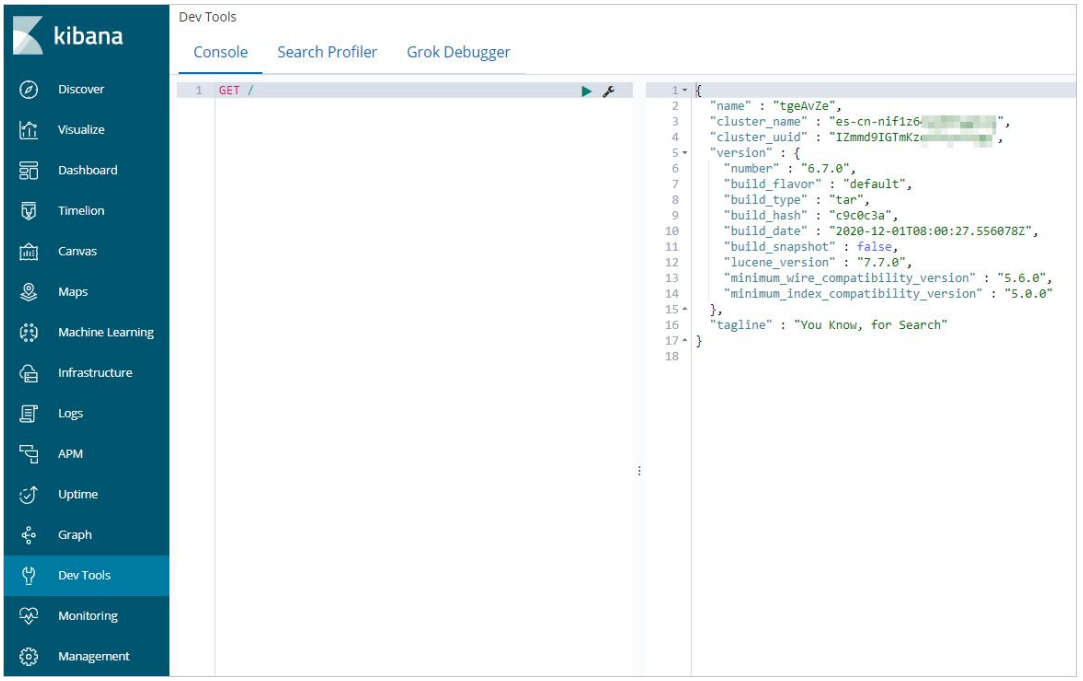
如果忘记可重置，重置密码的具体操作和注意事项，请参见重置实例访问密码：https://help.aliyun.com/document_detail/159883.htm#task-2458093

在登录成功页面，单击 Explore on my own。



- 在左侧导航栏，单击 Dev Tools（开发工具），再单击 Go to work。
- 在 Console 中，执行如下命令访问 Elasticsearch 实例。

```
GET /
```



访问成功后，结果如下。

```
{
  "name" : "tgeAvZe",
  "cluster_name" : "es-cn-nif1z64qj003g****",
  "cluster_uuid" : "IZmmd9IGTmKzqiZiyz****",
  "version" : {
    "number" : "6.7.0",
    "build_flavor" : "default",
    "build_type" : "tar",
    "build_hash" : "c9c0c3a",
    "build_date" : "2020-12-01T08:00:27.556078Z",
```

```
        "build_snapshot" : false,
        "lucene_version" : "7.7.0",
        "minimum_wire_compatibility_version" : "5.6.0",
        "minimum_index_compatibility_version" : "5.0.0"
    },
    "tagline" : "You Know, for Search"
}
```

步骤三：创建索引

创建一个名称为 product_info 的索引：

```
PUT /product_info
{
    "settings": {
        "number_of_shards": 5,
        "number_of_replicas": 1
    },
    "mappings": {
            "properties": {
                "productName": {
                    "type": "text",
                    "analyzer": "ik_smart"
                },
                "annual_rate":{
                    "type":"keyword"
                },
```

```
            "describe": {
              "type": "text",
              "analyzer": "ik_smart"
            }
        }
    }
}
```

以上示例创建了一个名称为 product_info 的索引。索引的类型为 products（7.0及以上版本为_doc），并包含了 productName、annual_rate 和 describe 字段。

创建成功后，返回结果如下。

```
{
  "acknowledged" : true,
  "shards_acknowledged" : true,
  "index" : "product_info"
}
```

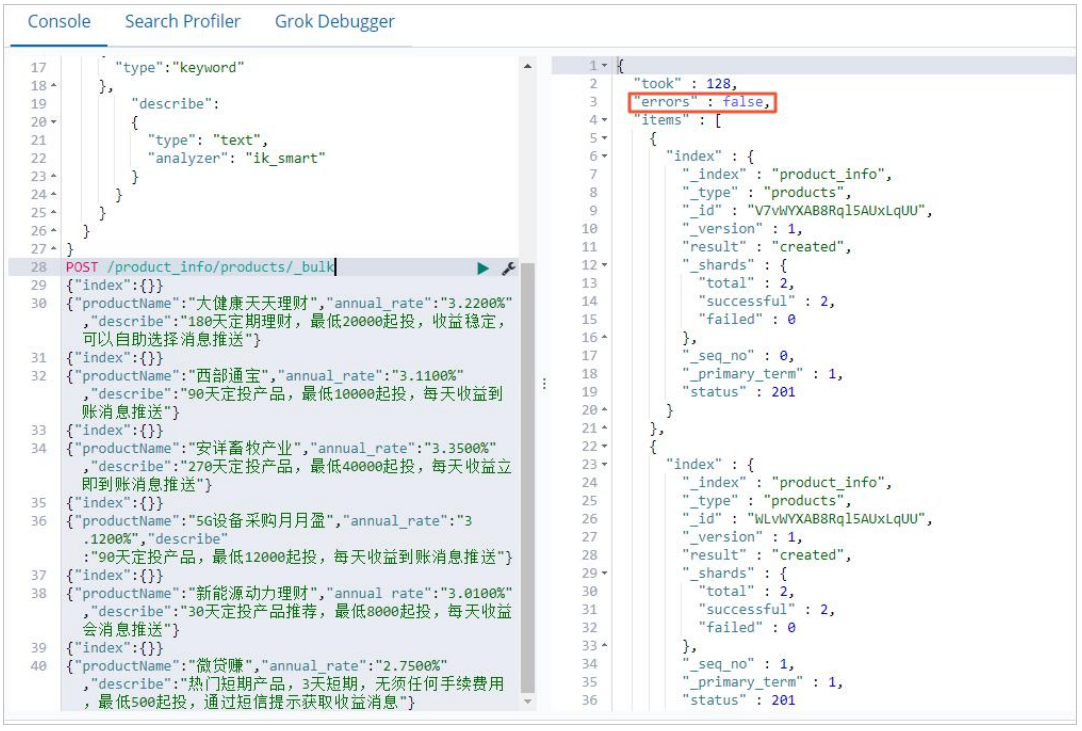
步骤四：创建文档并插入数据

使用 _bulk API，批量插入数据。

```
POST /product_info/_doc/_bulk
{"index":{}}
{"productName":"大健康天天理财","annual_rate":"3.2200%","describe":"180 天定期理财，最低
```

```
20000 起投，收益稳定，可以自助选择消息推送"}
{"index":{}}
{"productName":"西部通宝","annual_rate":"3.1100%","describe":"90 天定投产品，最低 10000 起投，每天收益到账消息推送"}
{"index":{}}
{"productName":"安详畜牧产业","annual_rate":"3.3500%","describe":"270 天定投产品，最低 40000 起投，每天收益立即到账消息推送"}
{"index":{}}
{"productName":"5G 设备采购月月盈","annual_rate":"3.1200%","describe":"90 天定投产品，最低 12000 起投，每天收益到账消息推送"}
{"index":{}}
{"productName":"新能源动力理财","annual rate":"3.0100%","describe":"30 天定投产品推荐，最低 8000 起投，每天收益会消息推送"}
{"index":{}}
{"productName":"微贷赚","annual_rate":"2.7500%","describe":"热门短期产品，3 天短期，无须任何手续费用，最低 500 起投，通过短信提示获取收益消息"}
```

如果返回显示"errors" ： false，说明数据插入成功。



步骤五：搜索数据

全文搜索搜索描述内容包含每天收益到账消息推送的所有产品。

```
GET  /product_info/_doc/_search
{
   "query": {
      "match": {
         "describe": "每天收益到账消息推送"
      }
   }
}
```

搜索成功后，返回结果如下。


```
{
  "took" : 21,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "skipped" : 0,
    "failed" : 0
  },
  "hits" : {
    "total" : 6,
    "max_score" : 1.3968885,
    "hits" : [
      {
        "_index" : "product_info",
        "_type" : "products",
        "_id" : "WLvWYXAB8Rql5AUxLqUU",
        "_score" : 1.3968885,
        "_source" : {
          "productName" : "西部通宝",
          "annual_rate" : "3.1100%",
          "describe" : "90 天定投产品，最低 10000 起投，每天收益到账消息推送"
        }
      },
      {
        "_index" : "product_info",
```

```
      "_type" : "products",
      "_id" : "WrvWYXAB8Rql5AUxLqUU",
      "_score" : 1.3968885,
      "_source" : {
        "productName" : "5G 设备采购月月盈",
        "annual_rate" : "3.1200%",
        "describe" : "90 天定投产品，最低 12000 起投，每天收益到账消息推送"
      }
    },
    {
      "_index" : "product_info",
      "_type" : "products",
      "_id" : "WbvWYXAB8Rql5AUxLqUU",
      "_score" : 1.3547361,
      "_source" : {
        "productName" : "安详畜牧产业",
        "annual_rate" : "3.3500%",
        "describe" : "270 天定投产品，最低 40000 起投，每天收益立即到账消息推送"
      }
    },
    {
      "_index" : "product_info",
      "_type" : "products",
      "_id" : "W7vWYXAB8Rql5AUxLqUU",
      "_score" : 1.1507283,
      "_source" : {
        "productName" : "新能源动力理财",
        "annual  rate" : "3.0100%",
        "describe" : "30 天定投产品推荐，最低 8000 起投，每天收益会消息推送"
```

```
            }
          },
          {
            "_index" : "product_info",
            "_type" : "products",
            "_id" : "XLvWYXAB8Rql5AUxLqUU",
            "_score" : 0.5753642,
            "_source" : {
              "productName" : "微贷赚",
              "annual_rate" : "2.7500%",
              "describe" : "热门短期产品，3 天短期，无须任何手续费用，最低 500 起投，通过短信提示获取收益消息"
            }
          },
          {
            "_index" : "product_info",
            "_type" : "products",
            "_id" : "V7vWYXAB8Rql5AUxLqUU",
            "_score" : 0.31854028,
            "_source" : {
              "productName" : "大健康天天理财",
              "annual_rate" : "3.2200%",
              "describe" : "180 天定期理财，最低 20000 起投，收益稳定，可以自助选择消息推送"
            }
          }
        ]
      }
    }
```

阿里云 Elasticsearch 支持通过分词器搜索数据，同时也支持评分排序。在上文的返回结果中，前两条商品信息中都出现了每天收益到账消息推送，后两条商品信息中只出现了关键词消息推送，所以越靠前的搜索结果的匹配度越高，分数也越高。

按查询条件搜索

搜索年化率在 3.0000%到 3.1300%之间的产品。

```
GET /product_info/_doc/_search
{
  "query": {
    "range": {
      "annual_rate": {
        "gte": "3.0000%",
        "lte": "3.1300%"
      }
    }
  }
}
```

执行成功后，返回结果如下。

```
{
  "took" : 10,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
```

```
    "successful" : 5,
    "skipped" : 0,
    "failed" : 0
  },
  "hits" : {
    "total" : 2,
    "max_score" : 1.0,
    "hits" : [
      {
        "_index" : "product_info",
        "_type" : "products",
        "_id" : "WLvWYXAB8Rql5AUxLqUU",
        "_score" : 1.0,
        "_source" : {
          "productName" : "西部通宝",
          "annual_rate" : "3.1100%",
          "describe" : "90 天定投产品，最低 10000 起投，每天收益到账消息推送"
        }
      },
      {
        "_index" : "product_info",
        "_type" : "products",
        "_id" : "WrvWYXAB8Rql5AUxLqUU",
        "_score" : 1.0,
        "_source" : {
          "productName" : "5G 设备采购月月盈",
          "annual_rate" : "3.1200%",
          "describe" : "90 天定投产品，最低 12000 起投，每天收益到账消息推送"
        }
      }
    ]
  } }
```

Elasticsearch 会根据条件搜索到符合预期的产品，同时以降序排列展示

更多搜索方式，请参见 Query DSL：https://www.elastic.co/guide/en/elasticsearch/reference/6.7/query-dsl.html?spm=a2c4g.11186623.2.27.33de9184S7oAZi

步骤六：删除索引（可选）

了解了阿里云 Elasticsearch 的使用方法后，你可以执行如下命令，删除对应索引，避免浪费资源。

索引删除后不可恢复，请谨慎操作。

```
DELETE /product_info
```

删除成功后，返回如下结果。

```
{
  "acknowledged" : true
}
```

步骤七：释放实例（可选）

如果你不再需要实例，可以将其释放。释放后，实例停止计费，数据不可恢复。释放操作只适用于按量付费实例。

实例释放后数据无法恢复，建议你在释放之前先备份数据。具体操作，请参见数据备份概述：https://help.aliyun.com/document_detail/106553.htm?spm=a2c4g.11186623.2.28.33de9184S7oAZi#concept-2038454

在实例列表中，单击操作列下的更多 > 释放实例。

在弹出的对话框中，单击确认。

3.4.2 Elasticsearch 基础应用

3.4.2.1 inverted index，doc_values，store 及 source

创作人：欧阳楚才

倒排索引

Elasticsearch 使用一种称为倒排索引的结构，它适用于快速的全文搜索。一个倒排索引由文档中所有不重复词的列表构成，对于其中每个词，有一个包含它的文档列表。

假设我们有两个文档，每个文档的正文字段包含如下内容：

- The quick brown fox jumped over the lazy dog
- Quick brown foxes leap over lazy dogs in summer

为了创建倒排索引，我们首先将每个文档的正文字段，拆分成单独的词（我们称它为词条或 Tokens），创建一个包含所有不重复词条的排序列表，然后列出每个词条出现在哪个文档。

结果如下所示：

词条	文档 1	文档 2
Quick		X
The	X	
brown	X	X
dog	X	
dogs		X
fox	X	
foxes		X
in		X
jumped	X	
lazy	X	X
leap		X
over	X	X
quick	X	
summer		X
the	X	

现在，如果我们想搜索 Quick brown，我们只需要查找包含每个词条的文档：

词条	文档 1	文档 2
brown	X	X
quick	X	
匹配词条数量	2	1

两个文档都匹配，但是第一个文档比第二个匹配度更高。如果我们使用，仅计算匹配词条数量的简单相似性算法，那么我们可以说，对于我们查询的相关性来讲，第一个文档比第二个文档更佳。

但是，我们目前的倒排索引有一些问题：

- Quick 和 quick 以独立的词条出现，然而用户可能认为它们是相同的词。
- fox 和 foxes 非常相似，就像 dog 和 dogs，他们有相同的词根。
- jumped 和 leap，尽管没有相同的词根，但他们是同义词。

使用前面的索引搜索 + Quick + fox 不会得到任何匹配文档。（记住，+ 前缀表明这个词必须存在）只有同时出现 Quick 和 fox 的文档才满足这个查询条件，但是第一个文档包含 quick fox，第二个文档包含 Quick foxes。

我们的用户可以合理的期望两个文档与查询匹配，我们可以做的更好。

如果我们将词条规范为标准模式，那么我们可以找到与用户搜索的词条不完全一致，但具有足够相关性的文档，例如：

- Quick 可以小写化为 quick
- foxes 可以词干提取 -- 变为词根的格式 -- 为 fox。类似的，dogs 可以为提取为 dog
- jumped 和 leap 是同义词，可以索引为相同的单词 jump

现在索引看上去像这样：

词条	文档 1	文档 2
brown	X	X
dog	X	X
fox	X	X
in		X
jump	X	X
lazy	X	X
over	X	X
quick	X	X
summer		X
the	X	

这还远远不够。我们搜索 +Quick +fox 仍然会失败，因为在我们的索引中，已经没有 Quick 了。

但是，如果我们对搜索的字符串，使用与正文字段相同的标准化规则，会变成查询 +quick +fox，这样两个文档都会匹配。

禁用索引

默认情况下，Elasticsearch 文档每个字段都会被索引。如果某些字段不需要支持查询，可以在映射中配置 "index": false ，减少存储空间占用，并且提升写入速度。尽管这个字段不能被搜索，但是它并不妨碍做聚合（如果该字段是可以聚合的字段）。

例如，文章的标题、正文、发布时间字段，需要创建索引，文章的 url 字段不需要被索引，创建索引映射时可以按以下方式禁用它：

```
PUT news
{
  "settings": {
    "number_of_shards": 5,
    "number_of_replicas": 1
  },
  "mappings": {
    "properties": {
      "title": {
        "type": "text",
        "analyzer": "ik_max_word"
      },
      "content": {
        "type": "text",
        "analyzer": "ik_max_word"
```

```
        },
        "createtime": {
          "type": "date"
        },
        "url": {
          "type": "keyword",
          "index": false
        }
      }
    }
  }
```

文档值

在 Elasticsearch 中，Doc Values 是一种列式存储结构。Doc Values 默认对所有字段启用，除了 text 和 annotated_text 类型字段。

Doc Values 是在索引时创建的，当字段索引时，Elasticsearch 为了能够快速检索，会把字段的值加入倒排索引中，同时它也会存储该字段的 Doc Values。

Elasticsearch 中的 Doc Values 常被应用到以下场景：

- 对一个字段进行排序
- 对一个字段进行聚合
- 某些过滤，比如地理位置过滤
- 某些与字段相关的脚本计算
- 使用 docvalue_fields 返回搜索结果部分字段值

因为文档值被序列化到磁盘，可以依靠操作系统的帮助来快速访问。当数据集远小于节点的可用内存，操作系统会自动将所有的 Doc Values 保存在内存中，使得其读写十分高速；当其远大于可用内存，操作系统会自动把 Doc Values 加载到系统的页缓存中，从而避免了 JVM 堆内存溢出异常。

列式存储的压缩

Doc Values 本质上是一个序列化的列式存储，适用于聚合、排序、脚本等操作。这种存储方式也非常便于压缩，特别是数字类型，这样可以节约磁盘空间，并且提高访问速度。

来看一组数字类型的 Doc Values，了解它如何压缩数据：

文档	词条
文档 1	100
文档 2	1000
文档 3	1500
文档 4	1200
文档 5	300
文档 6	1900
文档 7	4200

按列布局意味着我们有一个连续的数据块：[100,1000,1500,1200,300,1900,4200]。

因为我们已经知道他们都是数字，而不是像文档或行中看到的异构集合，所以我们可以使用统一的偏移来将他们紧紧排列。

而且，针对这样的数字有很多种压缩技巧。你会注意到这里每个数字都是 100 的倍数，Doc Values 会检测一个段里面的所有数值，并使用一个最大公约数，方便做进一步的数据压缩。

如果我们保存 100 作为此段的除数，我们可以对每个数字都除以 100，然后得到：[1,10,15,12,3,19,42]。

现在这些数字变小了，只需要很少的位就可以存储下，也减少了磁盘存放的大小。Doc Values 在压缩过程中使用如下技巧，它会按依次检测以下压缩模式:

- 如果所有的数值各不相同或缺失，设置一个标记并记录这些值
- 如果这些值小于 256，将使用一个简单的编码表
- 如果这些值大于 256，检测是否存在一个最大公约数
- 如果没有存在最大公约数，从最小的数值开始，统一计算偏移量进行编码

你会发现这些压缩模式不是传统的通用压缩算法，比如 DEFLATE 或者 LZ4。因为列式存储的结构是严格且良好定义的，我们可以通过使用专门的模式来达到，比通用压缩算法（如 LZ4）更高的压缩效果。

禁用 Doc Values

Doc Values 默认对所有字段启用，除了 text 和 annotated_text 类型字段。也就是说所有的数字、地理坐标、日期、IP 和 keyword 类型都会默认开启。

Text 类型字段不能使用 Doc Values，文本经过分析流程生成很多 Token，使得 Doc Values 不能高效运行。

因为 Doc Values 默认启用，你可以选择对你数据集里面的大多数字段，进行聚合和排序操作。如果你知道你永远也不会对某些字段进行聚合、排序或是使用脚本操作，你可以通过禁用特定字段的 Doc Values。这样不仅节省磁盘空间，也会提升索引的速度。

要禁用 Doc Values，在字段的映射 (mapping) 设置 doc_values: false 即可。例如，这里我们创建了一个新的索引，字段 "session_id" 禁用了 Doc Values：

```
PUT my_index
{
  "mappings": {
    "properties": {
      "session_id": {
        "type": "keyword",
        "doc_values": false
      }
    }
  }
}
```

通过设置 doc_values: false，这个字段将不能被用于聚合、排序以及脚本操作。

反过来也是可以进行配置的：让一个字段可以被聚合，通过禁用倒排索引，使它不能被正常搜索，例如：

```
PUT my_index
{
  "mappings": {
    "properties": {
      "customer_token": {
        "type": "keyword",
        "doc_values": true,
        "index": false
      }
    }
  }
}
```

通过设置 doc_values: true 和 index: false，我们得到一个只能被用于聚合/排序/脚本的字段。无可否认，这是一个非常少见的情况，但很有用。

存储

默认情况下，字段原始值会被索引用于查询，但是不会被存储。为了展示文档内容，通过一个叫 _source 的字段用于存储整个文档的原始值。

在字段的映射（mapping）设置 store: true，可以使索引单独保存这个字段。通常情况下，如果文档本身十分庞大，而一些字段又会经常单独使用，那么这样的字段，就可以设置为单独存储，然后可以使用 stored_fields 单独检索这些字段。

例如，如果你的文档包含标题、时间和一个很大的正文字段，你可能只需要检索标题、时间字段，没必要从很大的 _source 原文中解析出这些字段：

```
#创建索引,指定常用字段 store 属性
PUT /my-index-000001
{
  "mappings": {
    "properties": {
      "title": {
        "type": "text",
        "store": true
      },
      "date": {
        "type": "date",
        "store": true
      },
      "content": {
        "type": "text"
      }
    }
  }
}
```

```
#插入记录
PUT /my-index-000001/_doc/1
{
  "title":    "短文本标题",
  "date":     "2021-05-01",
  "content": "很长很长很长的正文字段..."
}

#查询结果返回 stored_fields 指定字段
GET /my-index-000001/_search
{
  "stored_fields": [ "title", "date" ]
}
```

注意：stored_fields 返回结果是数组格式。如果你需要获取原始文档，可以通过_source 字段替代。

原文

_source 字段包含索引时发送的原始 JSON 文档。_source 字段本身不建索引，但是存储原始文档，以便在执行查询请求时，可以将其返回。可以通过设置，禁用原文字段，或者只存储特定字段。

_source 在 Lucene 中是映射为一个特殊的字段：

Field	Index	IndexType	Analyzer	DocValues	Store
_source	No	No	No	No	Yes

Elasticsearch 中 _source 字段的主要目的，是通过 doc_id 读取该文档的原始内容，所以只需要存储 Store 即可。

Elasticsearch 中使用 _source 字段可以实现以下功能：

Update：

部分更新时，需要从读取文档保存在 _source 字段中的原文，然后和请求中的部分字段合并为一个完整文档。如果没有 _source，则不能完成部分字段的 Update 操作。

Reindex：

可以通过 Reindex API 完成索引重建，过程中不需要从其他系统导入全量数据，而是从当前文档的 _source 中读取。如果没有 _source，则不能使用 Reindex API。

Script：

不管是 Index 还是 Search 的 Script，都可能用到存储在 Store 中的原始内容，如果禁用了 _source，则这部分功能不再可用。

Summary：

摘要信息也是来源于 _source 字段。

禁用 _source

尽管使用非常方便，但是 _source 字段会导致占用更多的存储空间。如果业务上不需要存储原始文档，可以按以下方式禁用它：

```
PUT my-index-000001
{
  "mappings": {
    "_source": {
      "enabled": false
    }
  }
}
```

注意：禁用 _source 会导致更新、重建索引、摘要功能不可用，生产环境慎用。考虑节省存储空间，可以通过修改索引设置 index.codec 提高压缩效率。

包含/排除部分字段

包含/排除 _source 部分字段可以按以下方式设置它：

```
PUT logs
{
  "mappings": {
    "_source": {
```

```
        "includes": [
          "*.count",
          "meta.*"
        ],
        "excludes": [
          "meta.description",
          "meta.other.*"
        ]
      }
    }
}

PUT logs/_doc/1
{
  "requests": {
    "count": 10,
    "foo": "bar"
  },
  "meta": {
    "name": "Some metric",
    "description": "Some metric description",
    "other": {
      "foo": "one",
      "baz": "two"
    }
  }
}

GET logs/_search
{
  "query": {
    "match": {
      "meta.other.foo": "one"
    }
  }
}
```

3.4.2.2 理解 mapping

创作人：欧阳楚才

映射（mapping）就像数据库中的 Schema ，描述了文档可能具有的字段或属性、每个字段的数据类型，比如 text，keyword，integer 或 date ，以及 Lucene 是如何索引和存储这些字段的。

核心简单字段类型

Elasticsearch 支持如下简单字段类型：

- 字符串： text，keyword
- 整数：byte，short，integer，long
- 浮点数： float，double
- 布尔型： boolean
- 日期： date

更多的字段类型比如 geo_point，ip，nested 等可以在链接处查看：https://www.elastic.co/guide/en/elasticsearch/reference/current/mapping-types.html

当你索引一个包含新字段的文档之前，未曾出现 Elasticsearch 会使用动态映射，通过 JSON 中基本数据类型，尝试猜测字段类型，使用如下规则：

JSON 数据	字段类型
布尔型：true 或者 false	boolean
整数：123	long
浮点数：123.45	double
字符串，有效日期：2021-05-01	date
字符串：foo bar	text 和 keyword

注意：如果你通过引号（ "123" ）索引一个数字，它会被映射为字符串类型 text 和 keyword，而不是 long 。但如果这个字段已经映射为 long ，那么 Elasticsearch 会尝试将这个字符串转化为 long （在 coerce 设置为 true 的情况下），如果无法转化，则抛出一个异常。

查看映射

通过 /_mapping ，我们可以查看 Elasticsearch 在一个或多个索引中的映射。

Elasticsearch 文档写入示例：

```
PUT twitter/_doc/1
{
    "user": "kimchy",
    "post_date": "2009-11-15T13:12:00",
    "message": "Trying out Elasticsearch, so far so good?"
```

```
}

PUT twitter/_doc/2
{
    "user": "kimchy",
    "post_date": "2009-11-15T14:12:12",
    "message": "Another tweet, will it be indexed?"
}

PUT twitter/_doc/3
{
    "user": "elastic",
    "post_date": "2010-01-15T01:46:38",
    "message": "Building the site, should be kewl"
}
```

查看索引映射示例:

```
GET twitter/_mapping
```

Elasticsearch 根据我们索引的文档，为字段动态生成的映射：

```
{
  "twitter" : {
    "mappings" : {
      "properties" : {
        "message" : {
          "type" : "text",
          "fields" : {
```

```
            "keyword" : {
              "type" : "keyword",
              "ignore_above" : 256
            }
          }
        },
        "post_date" : {
          "type" : "date"
        },
        "user" : {
          "type" : "text",
          "fields" : {
            "keyword" : {
              "type" : "keyword",
              "ignore_above" : 256
            }
          }
        }
      }
    }
  }
```

注意：错误的映射，例如将年龄字段映射为 text 类型，而不是 integer ，会导致查询出现令人困惑的结果。

检查一下，而不是假设你的映射是正确的。

自定义字段映射

尽管在很多情况下基本字段数据类型已经够用，但你经常需要为单独字段自定义映射，特别是字符串字段。自定义映射允许你执行下面的操作：

- 全文字符串字段和精确值字符串字段的区别
- 使用特定语言分析器
- 优化字段以适应部分匹配
- 指定自定义数据格式
- 还有更多

字段最重要的属性是 type 。

```
{
    "number_of_clicks": {
        "type": "integer"
    }
}
```

字符串字段类型，包括全文字符串 text 和精确值字符串 keyword。

text 类型字段的最重要属性是分析器 analyzer，默认 Elasticsearch 使用 standard 分析器， 但你可以指定一个内置的分析器替代它，例如 whitespace 、simple 、english、cjk：

```
{
    "message": {
     "type": "text",
     "analyzer": "cjk"
   }
}
```

创建/更新映射

当你首次创建一个索引的时候，可以指定类型的映射。你也可以使用 /_mapping 更新映射。

我们可以更新一个映射来添加一个新字段，但不能更新一个现有的 mapping 把它的字段类型从一个变为另外一个，比如从 text 变为 keyword。我们可以在维持现有 mapping 的情况下，把一个字段变成一个 multi-field 字段。

为了描述指定映射的两种方式，我们先删除 twitter 索引：

```
DELETE twitter
```

创建一个新索引，指定 message 字段使用 cjk 分析器：

```
PUT twitter
{
  "settings": {
```

```
        "number_of_shards": "5",
        "number_of_replicas": "1"
      },
      "mappings": {
        "properties": {
          "user": {
            "type": "keyword"
          },
          "post_date": {
            "type": "date"
          },
          "message": {
            "type": "text",
            "analyzer": "cjk"
          }
        }
      }
    }
```

通过消息体中指定的 mappings 创建了索引映射，索引设置 settings 中通过 number_of_shards 指定分片数，number_of_replicas 指定副本数。

给映射增加一个新的名为 tag 的 keyword 类型字段，使用 _mapping ：

```
    PUT twitter/_mapping
    {
      "properties": {
        "tag": {
```

```
        "type": "keyword"
      }
    }
  }
```

我们不需要再次列出所有已存在的字段，因为无论如何我们都无法改变它们，新字段已经被合并到存在的映射中。

测试映射

可以使用 analyze API 测试字符串字段的映射，比较下面两个请求的输出：

```
GET /twitter/_analyze
{
  "field": "message",
  "text": "搜索索引引擎"
}

GET /twitter/_analyze
{
  "field": "tag",
  "text": "搜索引擎"
}
```

message体里面传入我们想要分析的文本。message 字段产生 3 个词条“搜索”、“索引”和“引擎”， tag 字段产生单独的词条”搜索引擎“，换句话说，我们的映射正常工作。

3.4.2.3 Search 通过 Kibana

创作人：李增胜

业务背景

在 TO B 行业，对商品的搜索展示，是有一定业务要求的，例如：存在合作关系的买家和供应商才能看到供应商店铺的商品，不存在合作关系的买家则不展示商品。另外，有些商品对客户甲展示一种价格，对客户乙则展示另外一种价格，从而区分不同的会员、分组对商品价格的区别。

一句话总结：TO B 行业的商品销售具有一定封闭性、特殊性。后续例子均在此背景下展开描述，以方便大家更加贴近业务场景来熟悉 Elasticsearch 对文档、索引、查询的一系列操作。

本文采用 IK 做分词器，下载的 IK 分词器版本必须和 Elasticsearch 版本一致
IK 下载地址：https://github.com/medcl/elasticsearch-analysis-ik/releases

- 在 Elasticsearch 的安装目录的 Plugins 目录下新建 IK 文件夹，然后将下载的 IK 安装包解压到此目录下。
- 重启 Elasticsearch 即可。

定义 Mapping

商品字段描述如下：

- goodsName: 商品名称
- skuCode：商品 sku 编码
- brandName：商品品牌名称
- channelType：渠道类型
- shopCode： 店铺编码
- publicPrice：售卖价格（基础价，对所有人开放价格）
- closeUserCode：封闭会员编码
- groupPrice：分组价格，其中使用嵌套类型存储，包括： 分组价格、 分组级别

定义商品 Mapping

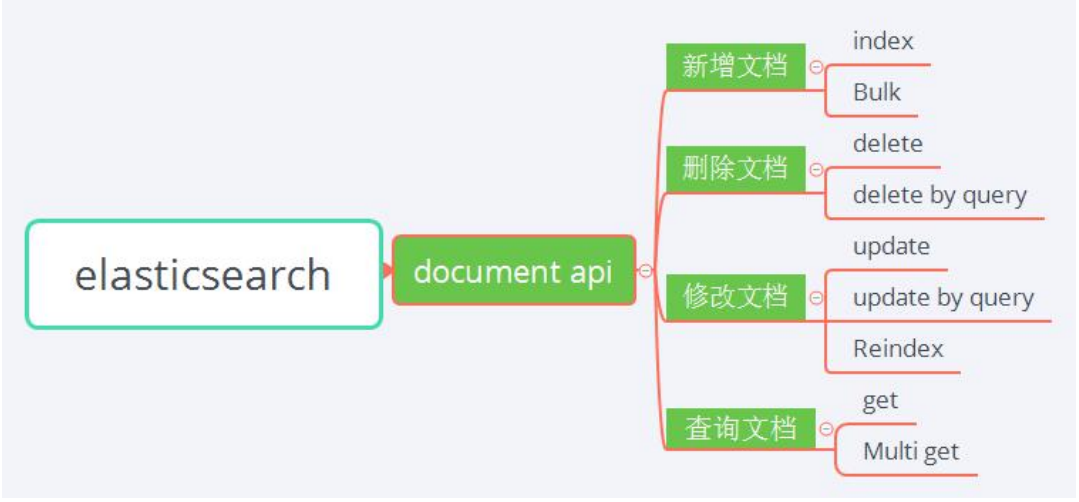
```
PUT my_goods
{
  "settings": {
    "index": {
      "number_of_shards": 1,
      "number_of_replicas": 1
    }
  },
  "mappings": {
    "properties": {
      "goodsName": {
        "type": "text",
        "analyzer": "ik_smart"
```

```
        },
        "skuCode": {
          "type": "keyword"
        },
        "brandName": {
          "type": "keyword"
        },
        "channelType": {
          "type": "keyword"
        },
        "shopCode": {
          "type": "keyword"
        },
        "publicPrice": {
          "type": "float"
        },
        "closeUserCode": {
          "type": "text",
          "analyzer": "standard"
        },
        "boostValue": {
          "type": "keyword"
        },
        "groupPrice": {
          "type": "nested",
          "properties": {
            "boxLevelPrice": {
              "type": "float"
            },
```

```
            "level": {
              "type": "text"
            }
          }
        }
      }
    }
  }
```

Document APIs

主要涉及以下几个核心功能：



Index

对文档的新增操作支持以下类型：

```
PUT /<target>/_doc/<_id>
POST /<target>/_doc/
PUT /<target>/_create/<_id>
POST /<target>/_create/<_id>
```

以 POST //_create/<_id>为例，以下将创建文档 ID 为 1 的商品信息：

```
POST /my_goods/_create/1
{
    "goodsName":"苹果 51 英寸 4K 超高清",
    "skuCode":"skuCode1",
    "brandName":"苹果",
    "closeUserCode":[
        "0"
    ],
    "channelType":"cloudPlatform",
    "shopCode":"sc00001",
    "publicPrice":"8188.88",
    "groupPrice":null,
    "boxPrice":null,
    "boostValue":1.8
}
```

Bulk

Elasticsearch 支持批量插入，_bulk 批量导入：

```
POST my_goods/_bulk
{"index":{"_id":1}}
{"goodsName":"苹果 51英寸 4K超高清","skuCode":"skuCode1","brandName":"苹果","closeUserCode":["0"],"channelType":"cloudPlatform","shopCode":"sc00001","publicPrice":"8188.88","groupPrice":null,"boxPrice":null,"boostValue":1.8}
{"index":{"_id":2}}
{"goodsName":"苹果 55英寸 3K超高清","skuCode":"skuCode2","brandName":"苹果","closeUserCode":["0"],"channelType":"cloudPlatform","shopCode":"sc00002","publicPrice":"6188.88","groupPrice":null,"boxPrice":null,"boostValue":1.0}
{"index":{"_id":3}}
{"goodsName":"苹果 UA55RU7520JXXZ 53英寸 4K高清","skuCode":"skuCode3","brandName":"美国苹果","closeUserCode":["0"],"channelType":"cloudPlatform","shopCode":"sc00001","publicPrice":"8388.88","groupPrice":null,"boxPrice":[{"boxType":"box1","boxUserCode":["htd003","uc004"],"boxPriceDetail":4388.88},{"boxType":"box2","boxUserCode":["uc005","uc0010"],"boxPriceDetail":5388.88}],"boostValue":1.2}
{"index":{"_id":4}}
{"goodsName":"山东苹果 UA55RU7520JXXZ 苹果 54 英寸 5K 超高清","skuCode":"skuCode4","brandName":"山东苹果","closeUserCode":["uc001","uc002","uc003"],"channelType":"cloudPlatform","shopCode":"sc00001","publicPrice":"8488.88","groupPrice":[{"level":"level1","boxLevelPrice":"2488.88"},{"level":"level2","boxLevelPrice":"3488.88"}],"boxPrice":[{"boxType":"box1","boxUserCode":["uc004","uc005","uc006","uc001"],"boxPriceDetail":4488.88},{"boxType":"box2","boxUserCode":["htd007","htd008","htd009","uc0010"],"boxPriceDetail":5488.88}],"boostValue":1.2}
{"index":{"_id":5}}
{"goodsName":"苹果 UA55R 苹果 U7 苹果 520JXXZ 55 英寸 5K 超高清","skuCode":"skuCode5","brandName":"三星苹果","closeUserCode":["uc001","uc002","uc003"],"channelType":"cloudPlatform","shopCode":"sc00001","publicPrice":"8488.88","groupPrice":[{"level":"level1","boxLevelPrice":"2500"},{"level":"level2","boxLevelPrice":"3500"}],"boxPrice":[{"boxType":"box1","boxUserCode":["uc004","uc005","uc006","uc001"],"boxPriceDetail":3588.88},{"boxType":"box2","boxUserCode":["htd0
```

07","htd008","htd009","uc0010"],"boxPriceDetail":5588.88}],"boostValue":1.2}
{"index":{"_id":6}}
{"goodsName":"三星 UA55RU7520JXXZ 51 英寸 4K 超高清","skuCode":"skuCode1","brandName":"三星","closeUserCode":["0"],"channelType":"cmccPlatform","shopCode":"sc00001","publicPrice":"8188.88","groupPrice":null,"boxPrice":null,"boostValue":1.2}
{"index":{"_id":7}}
{"goodsName":"三星 UA55RU7520JXXZ 52 英寸 4K 超高清","skuCode":"skuCode2","brandName":"三星","closeUserCode":["0"],"channelType":"cmccPlatform","shopCode":"sc00001","publicPrice":"8288.88","groupPrice":null,"boxPrice":[{"boxType":"box1","boxUserCode":["htd002"],"boxPriceDetail":4288.88}],"boostValue":1.2}
{"index":{"_id":8}}
{"goodsName":"三星 UA55RU7520JXXZ 52 英寸 4K 超高清","skuCode":"skuCode2","brandName":"三星","closeUserCode":["uc0022"],"channelType":"cloudPlatform","shopCode":"sc00001","publicPrice":"8288.88","groupPrice":null,"boxPrice":[{"boxType":"box1","boxUserCode":["uc0022"],"boxPriceDetail":4288.88}],"boostValue":1.2}
{"index":{"_id":9}}
{"goodsName":"三星 UA55RU7520JXXZ 52 英寸 4K 超高清","skuCode":"skuCode2","brandName":"三星","closeUserCode":["uc0022"],"channelType":"cloudPlatform","shopCode":"sc00001","publicPrice":"8288.88","groupPrice":null,"boxPrice":[{"boxType":"box1","boxUserCode":["uc0022"],"boxPriceDetail":4288.88}],"boostValue":1.2}
{"index":{"_id":10}}
{"goodsName":"三星 UA55RU7520JXXZ 52 英寸 4K 超高清","skuCode":"skuCode2","brandName":"三星","closeUserCode":["uc0022"],"channelType":"cloudPlatform","shopCode":"sc00001","publicPrice":"8288.88","groupPrice":null,"boxPrice":[{"boxType":"box1","boxUserCode":["uc0022"],"boxPriceDetail":4288.88}],"boostValue":1.8}

Delete

对文档的删除操作支持以下类型：

```
DELETE /<index>/_doc/<_id>
```

删除文档 ID 为 2 的数据：

```
DELETE /my_goods/_doc/2
```

Delete by query

另外，删除操作支持带多种条件的删除，可以使用 _delete_by_query

如下操纵，将删除店铺编码为 sc00002 的所有商品。

```
POST /my_goods/_delete_by_query
{
  "query": {
    "match": {
      "shopCode": "sc00002"
    }
  }
}
```

Update

对文档的修改操作支持以下类型：

```
POST /<index>/_update/<_id>
```

修改文档 ID 为 1 的文档信息：

新增字段：

```
POST /my_goods/_update/1
{
  "doc": {
    "shopName": "小王店铺"
  }
}
```

修改店铺名称为：“张三店铺”：

```
POST /my_goods/_update/1
{
  "doc": {
    "shopName": "张三店铺"
  }
}
```

```
{
  "goodsName" : "苹果 51 英寸 4K 超高清",
  "skuCode" : "skuCode1",
  "brandName" : "苹果",
  "closeUserCode" : [
```

```
      "0"
    ],
    "channelType" : "cloudPlatform",
    "shopCode" : "sc00001",
    "publicPrice" : "8188.88",
    "groupPrice" : null,
    "boxPrice" : null,
    "boostValue" : 1.8,
    "shopName" : "张三店铺"
}
```

另外还可以使用 PUT 进行修改，只不过需要罗列所有字段：

```
PUT my_goods/_doc/10
{
    "goodsName": "三星 UA55RU7520JXXZ 52 英寸 4K 超高清",
    "skuCode": "skuCode10",
    "brandName": "三星",
    "closeUserCode": [
      "uc0022"
    ],
    "channelType": "cloudPlatform",
    "shopCode": "sc00001",
    "publicPrice": "8288.88",
    "groupPrice": null,
    "boxPrice": [
      {
        "boxType": "box1",
```

```
        "boxUserCode": [
          "uc0022"
        ],
        "boxPriceDetail": 4288.88
      }
    ],
    "boostValue": 1.8
  }
```

用脚本同样能实现更新操作：

```
POST my_goods/_update/10
{
  "script": {
    "source": "ctx._source.city=params.channelType",
    "lang": "painless",
    "params": {
      "channelType": "cloudPlatform1"
    }
  }
}
```

Update by query

更新操作还可以使用 _update_by_query API，当店铺编码为 sc00002 时修改 publicPrice 为 5888.00 元。

插入文档 ID 为 2 的店铺商品信息：

```
POST /my_goods/_create/2
{
  "goodsName": "苹果 55 英寸 3K 超高清",
  "skuCode": "skuCode2",
  "brandName": "苹果",
  "closeUserCode": [
    "0"
  ],
  "channelType": "cloudPlatform",
  "shopCode": "sc00002",
  "publicPrice": "6188.88",
  "groupPrice": null,
  "boxPrice": null,
  "boostValue": 1
}
```

此时查询返回：

```
{
  "goodsName" : "苹果 55 英寸 3K 超高清",
  "skuCode" : "skuCode2",
  "brandName" : "苹果",
  "closeUserCode" : [
    "0"
  ],
```

```
    "channelType" : "cloudPlatform",
    "shopCode" : "sc00002",
    "publicPrice" : "6188.88",
    "groupPrice" : null,
    "boxPrice" : null,
    "boostValue" : 1.0
}
```

更新当店铺编码为 sc00002 时修改 publicPrice 为 5888.00 元：

```
POST /my_goods/_update_by_query
{
  "script": {
    "source": "ctx._source.publicPrice=5888.00",
    "lang": "painless"
  },
  "query": {
    "term": {
      "shopCode": "sc00002"
    }
  }
}
```

再次查询结果：

```
GET /my_goods/_source/2
```

```
{
  "shopCode" : "sc00002",
  "brandName" : "苹果",
  "closeUserCode" : [
    "0"
  ],
  "groupPrice" : null,
  "boxPrice" : null,
  "channelType" : "cloudPlatform",
  "boostValue" : 1.0,
  "publicPrice" : 5888.0,
  "goodsName" : "苹果 55 英寸 3K 超高清",
  "skuCode" : "skuCode2"
}
```


Reindex

当有业务需要重建索引时需要用到 _reindex API。

索引的来源和目的地，必须是已经存在的 index、index alias 或者 data stream。

你可以简单的将索引 A reindex 到索引 B，当然也可以带条件的 reindex 到索引 B。

如下所示，将 skuCode=skuCode2 的商品信息 reindex 到索引 my_goods_new 中：

```
POST _reindex
{
  "source": {
    "index": "my_goods",
    "query": {
      "match": {
        "skuCode": "skuCode2"
      }
    }
  },
  "dest": {
    "index": "my_goods_new"
  }
}
```

查询 my_goods_new 索引数据：

```
GET my_goods_new/_search/
```

```
{
  "took" : 5,
  "timed_out" : false,
  "_shards" : {
    "total" : 1,
    "successful" : 1,
    "skipped" : 0,
    "failed" : 0
  },
```

```
"hits" : {
  "total" : {
    "value" : 4,
    "relation" : "eq"
  },
  "max_score" : 1.0,
  "hits" : [
    {
      "_index" : "my_goods_new",
      "_type" : "_doc",
      "_id" : "7",
      "_score" : 1.0,
      "_source" : {
        "goodsName" : "三星 UA55RU7520JXXZ 52 英寸 4K 超高清",
        "skuCode" : "skuCode2",
        "brandName" : "三星",
        "closeUserCode" : [
          "0"
        ],
        "channelType" : "cmccPlatform",
        "shopCode" : "sc00001",
        "publicPrice" : "8288.88",
        "groupPrice" : null,
        "boxPrice" : [
          {
            "boxType" : "box1",
            "boxUserCode" : [
              "htd002"
            ],
            "boxPriceDetail" : 4288.88
```

```
            }
          ],
          "boostValue" : 1.2
        }
      },
      {
        "_index" : "my_goods_new",
        "_type" : "_doc",
        "_id" : "8",
        "_score" : 1.0,
        "_source" : {
          "goodsName" : "三星 UA55RU7520JXXZ 52 英寸 4K 超高清",
          "skuCode" : "skuCode2",
          "brandName" : "三星",
          "closeUserCode" : [
            "uc0022"
          ],
          "channelType" : "cloudPlatform",
          "shopCode" : "sc00001",
          "publicPrice" : "8288.88",
          "groupPrice" : null,
          "boxPrice" : [
            {
              "boxType" : "box1",
              "boxUserCode" : [
                "uc0022"
              ],
              "boxPriceDetail" : 4288.88
            }
          ],
```

```
        "boostValue" : 1.2
      }
    },
    {
      "_index" : "my_goods_new",
      "_type" : "_doc",
      "_id" : "9",
      "_score" : 1.0,
      "_source" : {
        "goodsName" : "三星 UA55RU7520JXXZ 52 英寸 4K 超高清",
        "skuCode" : "skuCode2",
        "brandName" : "三星",
        "closeUserCode" : [
          "uc0022"
        ],
        "channelType" : "cloudPlatform",
        "shopCode" : "sc00001",
        "publicPrice" : "8288.88",
        "groupPrice" : null,
        "boxPrice" : [
          {
            "boxType" : "box1",
            "boxUserCode" : [
              "uc0022"
            ],
            "boxPriceDetail" : 4288.88
          }
        ],
        "boostValue" : 1.2
      }
```

```
        },
        {
          "_index" : "my_goods_new",
          "_type" : "_doc",
          "_id" : "10",
          "_score" : 1.0,
          "_source" : {
            "goodsName" : "三星 UA55RU7520JXXZ 52 英寸 4K 超高清",
            "skuCode" : "skuCode2",
            "brandName" : "三星",
            "closeUserCode" : [
              "uc0022"
            ],
            "channelType" : "cloudPlatform",
            "shopCode" : "sc00001",
            "publicPrice" : "8288.88",
            "groupPrice" : null,
            "boxPrice" : [
              {
                "boxType" : "box1",
                "boxUserCode" : [
                  "uc0022"
                ],
                "boxPriceDetail" : 4288.88
              }
            ],
            "boostValue" : 1.8
          }
        }
      ]
    }
}
```

Get

对文档的查询操作支持以下类型：

```
GET <index>/_doc/<_id>
HEAD <index>/_doc/<_id>
GET <index>/_source/<_id>
HEAD <index>/_source/<_id>
```

查询文档 ID 为 1 的文档信息：

```
GET /my_goods/_doc/1
```

查询文档 ID 为 1 的文档是否存在，只判断文档是否存在，head 返回的信息更少、性能更高，满足特殊业务场景使用：

```
HEAD /my_goods/_doc/1
```

返回：

```
200 - OK
```

只返回文档信息:

查询时只返回 _source 信息:

```
GET /my_goods/_source/1
```

返回:

```
{
  "goodsName" : "苹果 51 英寸 4K 超高清",
  "skuCode" : "skuCode1",
  "brandName" : "苹果",
  "closeUserCode" : [
    "0"
  ],
  "channelType" : "cloudPlatform",
  "shopCode" : "sc00001",
  "publicPrice" : "8188.88",
  "groupPrice" : null,
  "boxPrice" : null,
  "boostValue" : 1.8
}
```

定制化返回参数:

只获取 _source 部分参数，类似数据库查询中的指定字段，而不是 select * 返回所有字段:

```
#GET 请求模式
GET my_goods/_source/1/?_source_includes=brandName,goodsName
```

```
#返回
{
  "brandName" : "苹果",
  "goodsName" : "苹果 51 英寸 4K 超高清"
}


#POST body 请求模式
POST my_goods/_search
{
  "query": {
    "match_all": {

    }
  },
  "fields": ["brandName", "goodsName"],
  "_source": false
}
#返回
"hits" : [
      {
        "_index" : "my_goods",
        "_type" : "_doc",
        "_id" : "2",
        "_score" : 1.0,
        "fields" : {
          "brandName" : [
            "苹果"
          ],
```

```
            "goodsName" : [
              "苹果 55 英寸 3K 超高清"
            ]
          }
        },
        {
          "_index" : "my_goods",
          "_type" : "_doc",
          "_id" : "3",
          "_score" : 1.0,
          "fields" : {
            "brandName" : [
              "美国苹果"
            ],
            "goodsName" : [
              "苹果 UA55RU7520JXXZ 53 英寸 4K 高清"
            ]
          }
        },
    ...
}
```

查询文档 ID 为 1 的文档是否存在。

只判断文档是否存在 ，Head 返回的信息更少、性能更高，满足特殊业务场景使用:

```
HEAD /my_goods/_doc/1
```

返回:

```
200 - OK
```

Mutil get

ES 同时支持批量查询，需要使用 _mget API，查询文档 ID 等于 1 和 2 的文档信息:

```
GET /my_goods/_mget
{
  "docs": [
    {
      "_id": "1"
    },
    {
      "_id": "2"
    }
  ]
}
```

返回:

```
{
  "docs" : [
    {
      "_index" : "my_goods",
      "_type" : "_doc",
```

```
        "_id" : "1",
        "_version" : 7,
        "_seq_no" : 8,
        "_primary_term" : 1,
        "found" : true,
        "_source" : {
          "goodsName" : "苹果 51 英寸 4K 超高清",
          "skuCode" : "skuCode1",
          "brandName" : "苹果",
          "closeUserCode" : [
            "0"
          ],
          "channelType" : "cloudPlatform",
          "shopCode" : "sc00001",
          "publicPrice" : "8188.88",
          "groupPrice" : null,
          "boxPrice" : null,
          "boostValue" : 1.8,
          "shopName" : "张三店铺"
        }
      },
      {
        "_index" : "my_goods",
        "_type" : "_doc",
        "_id" : "2",
        "found" : false
      }
    ]
}
```

Query DSL

查询索引包括全文本查询、组合查询、结构化查询等。

通常 Search 与 Filter 区别

二者的查询是有区别的：

- Query 查询

用于解答文档是否存在，并且告知返回文档与查询条件的匹配度，返回 _score 评分供用户选择。

- Filter 查询

只用于返回文档是否与查询匹配，但是不会告诉你匹配度，即不进行评分。在做聚合查询时，filter 经常发挥更大的作用。因为没有评分 Elasticsearch 的处理速度就会提高，提升了整体响应时间。同时 filter 可以缓存查询结果，而 Query 则不能缓存。

使用场景

如果涉及到全文检索以及评分相关业务使用 Query，其他场景推荐使用 Filter 查询。

组合查询

Boolean 查询

Boolean 查询包含 must、filter、must_not。

must ：必须匹配并且返回评分，filter 忽略评分，should 相当于数据库查询中的 or，针对 should 有一个特殊的情况，也就是所有的搜索只有 should ，那么必须满足 should 里的其中一个才会被搜索到。must_not 为不匹配，相当于不等于。

查询：店铺编码=sc00001 且渠道 channelType=cloudPlatform 且 publicPrice 价格区间不在 8288-8888 之间，或者品牌包含"果"。首先以下条件必须全部满足：

- 店铺编码=sc00001
- 渠道 channelType=cloudPlatform
- publicPrice 价格区间不在 8288-8888 之间

另外，由于还有 should 查询，满足品牌中包含“果”的也会被查询出来，另外匹配成功后的评分也会提高，相应的结果也会排在前面：

- 品牌包含"果"

2 者取并集的结果作为最终结果返回:

```
POST /my_goods/_search
{
  "query": {
    "bool": {
      "must": {
        "term":{
          "shopCode":"sc00001"
        }
      },
      "filter": {
        "term": {
          "channelType": "cloudPlatform"
        }
      },
      "must_not": [
        {
         "range": {
           "publicPrice": {
             "gte": 8288,
             "lte": 8888
           }
         }
        }
      ],
      "should": [
        {
          "term": {
            "brandName": {
```

```
                "value": "果"
              }
            }
          }
        ],
        "minimum_should_match" : 1
      }
    }
}
```

minimum_should_match 为最小匹配数量，如果 bool 查询包含至少一个 should 子句，并且没有 must 或 filter 子句，则默认值为 1，否则，默认值为 0。举例说明：

```
POST /my_goods/_search
{
  "query": {
    "bool": {
      "should": [
        {
          "term": {
            "brandName": {
              "value": "东"
            }
          }
        },
        {
          "term": {
            "brandName": {
```

```
                "value": "果"
              }
            }
          }
        ],
        "minimum_should_match" : 1
      }
    }
}
```

以上查询表示 brandName 包含“东” 和 “果” 至少匹配成功一次，查询结果如下：

```
"hits" : [
      {
        "_index" : "my_goods",
        "_type" : "_doc",
        "_id" : "4",
        "_score" : 1.5678144,
        "_source" : {
          "shopCode" : "sc00001",
          "brandName" : "山东苹果",
          "closeUserCode" : [
            "uc001",
            "uc002",
            "uc003"
          ],
          "skuCode_brandName" : "skuCode4 山东苹果",
          "channelType" : "cloudPlatform",
```

```
        "publicPrice" : 16977.76,
        "goodsName_length" : 31,
        "groupPrice" : [
          {
            "level" : "level1",
            "boxLevelPrice" : "2488.88"
          },
          {
            "level" : "level2",
            "boxLevelPrice" : "3488.88"
          }
        ],
        "boxPrice" : [
          {
            "boxType" : "box1",
            "boxUserCode" : [
              "uc004",
              "uc005",
              "uc006",
              "uc001"
            ],
            "boxPriceDetail" : 4488.88
          },
          {
            "boxType" : "box2",
            "boxUserCode" : [
              "htd007",
              "htd008",
              "htd009",
              "uc0010"
            ],
```

```
            "boxPriceDetail" : 5488.88
          }
        ],
        "boostValue" : 1.2,
        "goodsName" : "山东苹果 UA55RU7520JXXZ 苹果 54 英寸 5K 超高清",
        "skuCode" : "skuCode4"
      }
    },
    {
      "_index" : "my_goods",
      "_type" : "_doc",
      "_id" : "2",
      "_score" : 0.2792403,
      "_source" : {
        "shopCode" : "sc00002",
        "brandName" : "苹果",
        "closeUserCode" : [
          "0"
        ],
        "skuCode_brandName" : "skuCode2 苹果",
        "channelType" : "cloudPlatform",
        "publicPrice" : 12377.76,
        "goodsName_length" : 13,
        "groupPrice" : null,
        "boxPrice" : null,
        "boostValue" : 1.0,
        "goodsName" : "苹果 55 英寸 3K 超高清",
        "skuCode" : "skuCode2"
      }
    },
    {
```

```
        "_index" : "my_goods",
        "_type" : "_doc",
        "_id" : "1",
        "_score" : 0.2792403,
        "_source" : {
          "shopCode" : "sc00001",
          "brandName" : "苹果",
          "closeUserCode" : [
            "0"
          ],
          "skuCode_brandName" : "skuCode1 苹果",
          "channelType" : "cloudPlatform",
          "publicPrice" : 32755.52,
          "goodsName_length" : 13,
          "groupPrice" : null,
          "boxPrice" : null,
          "boostValue" : 1.8,
          "goodsName" : "苹果 51 英寸 4K 超高清",
          "skuCode" : "skuCode1"
        }
      },
      {
        "_index" : "my_goods",
        "_type" : "_doc",
        "_id" : "3",
        "_score" : 0.21222264,
        "_source" : {
          "shopCode" : "sc00001",
          "brandName" : "美国苹果",
          "closeUserCode" : [
            "0"
```

```
        ],
        "skuCode_brandName" : "skuCode3 美国苹果",
        "channelType" : "cloudPlatform",
        "publicPrice" : 16777.76,
        "goodsName_length" : 26,
        "groupPrice" : null,
        "boxPrice" : [
          {
            "boxType" : "box1",
            "boxUserCode" : [
              "htd003",
              "uc004"
            ],
            "boxPriceDetail" : 4388.88
          },
          {
            "boxType" : "box2",
            "boxUserCode" : [
              "uc005",
              "uc0010"
            ],
            "boxPriceDetail" : 5388.88
          }
        ],
        "boostValue" : 1.2,
        "goodsName" : "苹果 UA55RU7520JXXZ 53 英寸 4K 高清",
        "skuCode" : "skuCode3"
      }
    },
  ...
]
```

当我们调整 minimum_should_match 为 2 时观察结果返回：

```
POST /my_goods/_search
{
  "query": {
    "bool": {
      "should": [
        {
          "term": {
            "brandName": {
              "value": "东"
            }
          }
        },
        {
          "term": {
            "brandName": {
              "value": "果"
            }
          }
        }
      ],
      "minimum_should_match" : 2
    }
  }
}

#返回：
"hits" : [
      {
```

```
"_index" : "my_goods",
"_type" : "_doc",
"_id" : "4",
"_score" : 1.5678144,
"_source" : {
  "shopCode" : "sc00001",
  "brandName" : "山东苹果",
  "closeUserCode" : [
    "uc001",
    "uc002",
    "uc003"
  ],
  "skuCode_brandName" : "skuCode4 山东苹果",
  "channelType" : "cloudPlatform",
  "publicPrice" : 16977.76,
  "goodsName_length" : 31,
  "groupPrice" : [
    {
      "level" : "level1",
      "boxLevelPrice" : "2488.88"
    },
    {
      "level" : "level2",
      "boxLevelPrice" : "3488.88"
    }
  ],
  "boxPrice" : [
    {
      "boxType" : "box1",
      "boxUserCode" : [
        "uc004",
```

```
                "uc005",
                "uc006",
                "uc001"
              ],
              "boxPriceDetail" : 4488.88
            },
            {
              "boxType" : "box2",
              "boxUserCode" : [
                "htd007",
                "htd008",
                "htd009",
                "uc0010"
              ],
              "boxPriceDetail" : 5488.88
            }
          ],
          "boostValue" : 1.2,
          "goodsName" : "山东苹果 UA55RU7520JXXZ 苹果 54 英寸 5K 超高清",
          "skuCode" : "skuCode4"
        }
      }
    ]
```

可以看到，只有 goodsName 出现 “东” 和 “果” 2 次以及 2 次以上的结果被查询到。

Boosting 查询

Boosting 用于控制评分相关度相关，可以提升评分也可以降低评分。

可以看到 2 条文档记录评分一致："_score" ： 1.3862942 ，

当我们修改 negative_boost: 0.2 时，此时返回（省略部分无关字段）

```
POST /my_goods/_search
{
  "query": {
    "boosting": {
      "positive": {
        "term": {
          "skuCode": {
            "value": "skuCode1"
          }
        }
      },
      "negative": {
        "term": {
          "goodsName": {
            "value": "三星"
          }
        }
      },
      "negative_boost": 0.2
    }
  }
}
```

```
#返回
"hits" : [
        {
          "_index" : "my_goods",
          "_type" : "_doc",
          "_id" : "1",
          "_score" : 1.3862942,
          "_source" : {
            "goodsName" : "苹果 51 英寸 4K 超高清",
            "skuCode" : "skuCode1",
            "brandName" : "苹果",
            "closeUserCode" : [
              "0"
            ],
            "channelType" : "cloudPlatform",
            "shopCode" : "sc00001",
            "publicPrice" : "8188.88",
            "groupPrice" : null,
            "boxPrice" : null,
            "boostValue" : 1.8,
            "shopName" : "张三店铺"
          }
        },
        {
          "_index" : "my_goods",
          "_type" : "_doc",
          "_id" : "6",
          "_score" : 0.27725884,
          "_source" : {
```

```
            "goodsName" : "三星 UA55RU7520JXXZ 51 英寸 4K 超高清",
            "skuCode" : "skuCode1",
            "brandName" : "三星",
            "closeUserCode" : [
              "0"
            ],
            "channelType" : "cmccPlatform",
            "shopCode" : "sc00001",
            "publicPrice" : "8188.88",
            "groupPrice" : null,
            "boxPrice" : null,
            "boostValue" : 1.2
          }
        }
      ]
```

此时发现文档 ID=6 的评分下降到 _score : 0.27725884，因为在 negative 命中了查询条件，negative_boost 在 0 到 1 之间时，用于降低评分，相反，大于 1 用于提升评分。

Constant score query 查询

当查询不关心 TF（词频）时，就可以使用 constant score query 。

```
POST /my_goods/_search
{
  "query": {
    "constant_score": {
```

```
        "filter": {
          "term": {
            "goodsName": "苹果"
          }
        },
        "boost": 1.2
      }
    }
}
```

返回（省略部分无关字段）：

```
{
          "_index" : "my_goods",
          "_type" : "_doc",
          "_id" : "3",
          "_score" : 1.2,
          "_source" : {
            "goodsName" : "苹果 UA55RU7520JXXZ 53 英寸 4K 高清"
          }
        },
        {
          "_index" : "my_goods",
          "_type" : "_doc",
          "_id" : "4",
          "_score" : 1.2,
          "_source" : {
            "goodsName" : "山东苹果 UA55RU7520JXXZ 苹果 54 英寸 5K 超高清"
```

```
        }
      }
  }
```

可以看到，文档 ID =3 的评分和文档 ID =4 的评分一样，但是 ID=4 的匹配相关度更高，这是由于我们忽略了词频对打分的影响。

Disjunction max query 查询

Disjunction 查询也被理解为分离最大化查询，指的是将任何与任一查询匹配的文档，作为结果返回，但只将最佳匹配的评分，作为查询的评分结果返回。

例如查询商品名称和品牌名称中包含“苹果”的信息，当品牌的评分高于商品名称时，则返回品牌的评分做为总评分（忽略 tie_breaker 缓冲）。

```
GET /my_goods/_search
{
  "query": {
    "dis_max": {
      "tie_breaker": 0.7,
      "boost": 1.2,
      "queries": [
        {
          "term": {
            "goodsName": {
              "value": "苹果"
```

```
            }
          }
        },
        {
          "term": {
            "brandName": {
              "value": "苹果"
            }
          }
        }
        ]
      }
    }
}
```

返回结果（忽略无关字段）：

```
"max_score" : 3.0150018,
    "hits" : [
      {
        "_index" : "my_goods",
        "_type" : "_doc",
        "_id" : "1",
        "_score" : 3.0150018,
        "_source" : {
          "goodsName" : "苹果 51 英寸 4K 超高清",
          "brandName" : "苹果"
        }
```

```
    },
    {
      "_index" : "my_goods",
      "_type" : "_doc",
      "_id" : "5",
      "_score" : 1.3465583,
      "_source" : {
        "goodsName" : "苹果 UA55R 苹果 U7 苹果 520JXXZ 55 英寸 5K 超高清",
        "brandName" : "三星苹果"
      }
    },
    {
      "_index" : "my_goods",
      "_type" : "_doc",
      "_id" : "4",
      "_score" : 1.2337791,
      "_source" : {
        "goodsName" : "山东苹果 UA55RU7520JXXZ 苹果 54 英寸 5K 超高清",
        "brandName" : "山东苹果"
      }
    },
```

分析：

- ID=1 的记录，由于品牌只包含“苹果” 2 字，Elasticsearch 认为这种匹配度更高，所以此条记录评分排在第一位。
- ID=5 的记录，由于品牌中和 ID =4 的记录都包含苹果且字数一样，此时就要看 goodsName 包含苹果的词频数量了，ID=5 的品牌中，“苹果”出现了 3 次，而 ID=4 的值出现了 2 次，所以评分没有 ID=5 的高，符合我们的预期结果。

- tie_breaker 字段做什么用呢？它是起到了缓冲的作用（取值范围：0 到 1 之间），Disjunction 查询会将匹配度最高的字段得分，做为整个文档的得分返回，这种情况其他字段就不起作用了，难免有点走极端。此时就需要 tie_breaker 来做缓存，提升其他字段的影响力，最终的结果：brandName 评分+ goodsName 评分 *tie_breaker，作为总评分返回。

Function score query 查询

Function score 允许你控制查询评分，是用来控制评分过程的终极武器。最高效的用法是用过滤器对结果的子集应用不同的函数，同时运用了 filter 的缓存，并且达到控制评分的过程。

我们想让山东的苹果搜索出现在美国苹果之前，查询商品名称包含“苹果”，当品牌中包含“美国”时，权重设置为 2，当出现“山东”时，权重设置为 40 。

```
GET /my_goods/_search
{
  "query": {
    "function_score": {
      "query": {
        "term": {
          "goodsName": {
            "value": "苹果"
          }
        }
      },
```

```
        "boost": 2,
        "functions": [
          {
            "filter": {
              "match":{
                "brandName":"美国"
              }
            },
            "random_score": {

            },
            "weight": 2
          },
          {
            "filter": {
              "match":{
                "brandName":"山东"
              }
            },
            "weight": 40
          }
        ],
        "max_boost": 60,
        "score_mode": "max",
        "boost_mode": "multiply",
        "min_score": 2
      }
    }
}
```

返回主要信息:


```
 "max_score" : 2.2442641,
   "hits" : [
    {
       "_index" : "my_goods",
       "_type" : "_doc",
       "_id" : "4",
       "_score" : 2.0562985,
       "_source" : {
         "goodsName" : "山东苹果 UA55RU7520JXXZ 苹果 54 英寸 5K 超高清",
         "brandName" : "山东苹果"
       }
     },
     {
       "_index" : "my_goods",
       "_type" : "_doc",
       "_id" : "3",
       "_score" : 1.7582327,
       "_source" : {
         "goodsName" : "苹果 UA55RU7520JXXZ 53 英寸 4K 高清",
         "brandName" : "美国苹果",
       }
     }
   ]
```


解释几个参数:

- score_modemultiply: 默认，分数相乘
- avg：平均分数，第一个 function 的分数
- max：使用评分最大的分数
- min：使用评分最小的分数 avg

举例，如果 2 个函数返回的分数为 1 和 2，并且它们的权重分别为 3 和 4，则他们的评分为：(13+24)/(3+4)

其他详解请参考官方 score-functions 详解：

https://www.elastic.co/guide/en/elasticsearch/reference/7.10/query-dsl-function-score-query.html#score-functions

Full text 全文本查询

Match 查询

Match 查询是一种标准的查询，示例如下:

```
# 通过 highlight 对查询到的结果进行高亮显示
GET /my_goods/_search
{
  "query": {
    "match": {
      "goodsName": "苹果 高清 英寸"
    }
```

```
    },
    "highlight": {
      "fields": {
        "goodsName": {
          "pre_tags": [
            "<strong>"
          ],
          "post_tags": [
            "</strong>"
          ]
        }
      }
    }
}
```

Match 查询是一种 boolean 类型的查询，可以使用 operator 来控制 boolean 字句，operator 包含 and 和 or (默认为 or)。

```
GET /my_goods/_search
{
  "query": {
    "match": {
      "goodsName": {
        "query": "苹果 高清 英寸",
        "operator": "and"
      }
    }
  }
```

```
}
#返回结果:
{
  "took" : 1,
  "timed_out" : false,
  "_shards" : {
    "total" : 1,
    "successful" : 1,
    "skipped" : 0,
    "failed" : 0
  },
  "hits" : {
    "total" : {
      "value" : 0,
      "relation" : "eq"
    },
    "max_score" : null,
    "hits" : [ ]
  }
}
```

命中为 0，因为没有标题中包含 “苹果 高清 英寸” 词组的商品信息，这里的 and 是将查询条件做分词处理，然后查询结果时，必须全部包含 “苹果 高清 英寸” 分词词组才能被检索，下面再演示下 or 的例子:

```
GET /my_goods/_search
{
  "query": {
    "match": {
      "goodsName": {
        "query": "苹果 高清 英寸",
        "operator": "or"
      }
    }
  }
}

#返回

{
          "_index" : "my_goods",
          "_type" : "_doc",
          "_id" : "4",
          "_score" : 1.836855,
          "_source" : {
            "shopCode" : "sc00001",
            "brandName" : "山东苹果",
            "closeUserCode" : [
              "uc001",
              "uc002",
              "uc003"
            ],
            "skuCode_brandName" : "skuCode4 山东苹果",
            "channelType" : "cloudPlatform",
            "publicPrice" : 16977.76,
            "goodsName_length" : 31,
```

```
"groupPrice" : [
  {
    "level" : "level1",
    "boxLevelPrice" : "2488.88"
  },
  {
    "level" : "level2",
    "boxLevelPrice" : "3488.88"
  }
],
"boxPrice" : [
  {
    "boxType" : "box1",
    "boxUserCode" : [
      "uc004",
      "uc005",
      "uc006",
      "uc001"
    ],
    "boxPriceDetail" : 4488.88
  },
  {
    "boxType" : "box2",
    "boxUserCode" : [
      "htd007",
      "htd008",
      "htd009",
      "uc0010"
    ],
    "boxPriceDetail" : 5488.88
  }
```

```
        ],
        "boostValue" : 1.2,
        "goodsName" : "山东苹果 UA55RU7520JXXZ 苹果 54 英寸 5K 超高清",
        "skuCode" : "skuCode4"
      }
    },
    {
      "_index" : "my_goods",
      "_type" : "_doc",
      "_id" : "10",
      "_score" : 0.9227071,
      "_source" : {
        "goodsName" : "三星 UA55RU7520JXXZ 52 英寸 4K 超高清",
        "skuCode" : "skuCode10",
        "brandName" : "三星",
        "closeUserCode" : [
          "uc0022"
        ],
        "channelType" : "cloudPlatform",
        "shopCode" : "sc00001",
        "publicPrice" : "8288.88",
        "groupPrice" : null,
        "boxPrice" : [
          {
            "boxType" : "box1",
            "boxUserCode" : [
              "uc0022"
            ],
            "boxPriceDetail" : 4288.88
          }
        ],
```

```
            "boostValue" : 1.8,
            "city" : "cloudPlatform1"
          }
        }
```

可以看到，“三星 UA55RU7520JXXZ 52 英寸 4K 超高清” 由于包含 “高清” 所以能被查询到。

Match phrase query

用于匹配索引中是否存在所输入的查询条件数据:

```
GET /my_goods/_search
{
  "query": {
    "match_phrase": {
      "goodsName": "apple"
    }
  }
}
```

比较 match_phrase 与 match 区别。

- match_phrase

将查询条件的中的信息看做一个整体，如下面的 “goods t” 必须 goods 在前 t 在后。

- match

将查询中的条件做分词处理后，再去做查询。

#查询不到任何数据，因为不存在'goods t'的词组

```
GET /my_goods/_search
{
  "query": {
    "match_phrase": {
      "goodsName": "goods t"
    }
  }
}
#能查询到数据，因为文档中包含 goods 和 t 的词组
GET /my_goods/_search
{
  "query": {
    "match": {
      "goodsName": "goods t"
    }
  }
}
```

在 match_phrase 中，可以通过 slop 来控制单词中间的间隔，默认为 0，下面举例说明：

```
GET /my_goods/_search
{
  "query": {
    "match_phrase": {
      "goodsName": {
        "query": "apple test",
        "slop": 1
      }
    }
  }
}

#返回
{
  "took" : 10,
  "timed_out" : false,
  "_shards" : {
    "total" : 1,
    "successful" : 1,
    "skipped" : 0,
    "failed" : 0
  },
  "hits" : {
    "total" : {
      "value" : 1,
      "relation" : "eq"
    },
    "max_score" : 3.08089,
    "hits" : [
      {
        "_index" : "my_goods",
```

```
"_type" : "_doc",
"_id" : "21",
"_score" : 3.08089,
"_source" : {
  "goodsName" : "apple goods test",
  "skuCode" : "skuCode3",
  "brandName" : "美国苹果",
  "closeUserCode" : [
    "0"
  ],
  "channelType" : "cloudPlatform",
  "shopCode" : "sc00001",
  "publicPrice" : "8388.88",
  "groupPrice" : null,
  "boxPrice" : [
    {
      "boxType" : "box1",
      "boxUserCode" : [
        "htd003",
        "uc004"
      ],
      "boxPriceDetail" : 4388.88
    },
    {
      "boxType" : "box2",
      "boxUserCode" : [
        "uc005",
        "uc0010"
      ],
      "boxPriceDetail" : 5388.88
    }
```

```
            ],
            "boostValue" : 1.2
          }
        }
      ]
    }
  }
```

可以看到，我们设置了 1 个词条，apple 与 test 之间间隔 一个词条，故能查询到。

Match phrase prefix query

返回文档包含给定查询条件的文档，文档中必须包含给定条件的内容，且是按照 prefix 来进行匹配的，如 "apple goods test" ，商品名称包含 apple goods test 的数据将被查询到返回。

新增一条测试数据：

```
POST my_goods/_bulk
{"index":{"_id":13}}
{"goodsName":"apple and goods product ","skuCode":"skuCode3","brandName":"美国苹果","closeUserCode":["0"],"channelType":"cloudPlatform","shopCode":"sc00001","publicPrice":"8388.88","groupPrice":null,"boxPrice":[{"boxType":"box1","boxUserCode":["htd003","uc004"],"boxPriceDetail":4388.88},{"boxType":"box2","boxUserCode":["uc005","uc0010"],"boxPriceDetail":5388.88}],"boostValue":1.2}
{"index":{"_id":21}}
```

{"goodsName":"apple goods test","skuCode":"skuCode3","brandName":"美国苹果","closeUserCode":["0"],"channelType":"cloudPlatform","shopCode":"sc00001","publicPrice":"8388.88","groupPrice":null,"boxPrice":[{"boxType":"box1","boxUserCode":["htd003","uc004"],"boxPriceDetail":4388.88},{"boxType":"box2","boxUserCode":["uc005","uc0010"],"boxPriceDetail":5388.88}],"boostValue":1.2}

```
#只返回 goodsName : apple goods test 的数据
GET /my_goods/_search
{
  "query": {
    "match_phrase_prefix": {
      "goodsName": "apple goods t"
    }
  }
}
```

总结比较 match 这四种查询。

| Match | 返回匹配查询条件的文档内容，查询条件会在匹配之前会被分词处理。

|

Match boolean prefix	是一个 Boolean 查询，将分词后的短语按照 term 进行查询，最后一个词组按照 prefix 查询。
Match phrase	

| 将查询条件当做一个词组进行查询，不进行分词处理。

|

| Match phrase prefix

| 返回文档包含给定查询条件的文档，文档中必须包含给定条件的内容且是按照顺序的，与 match phrase 类似，对最后一个 token 会进行前缀匹配，可以通过 slop 来控制匹配 token 的位置差。 |

Multi-match

多字段匹配，可以在多个字段中匹配查询相关信息，通过 type 参数可以调整结果集：

```
#查询商品名称和品牌名称中包含苹果的文档信息
POST /my_goods/_search
{
  "query": {
    "multi_match": {
      "query": "苹果",
      "type": "best_fields",
      "fields": ["goodsName","brandName"],
      "tie_breaker": 0.3
    }
  }
}
```

type 参数类型详解：

- best_fields ：默认，匹配 fields，将评分最高的分数做为整个查询的分数返回；
- most_fields：查询匹配的文档，并且返回各个字段的分数之和的平均值；
- cross_fields：跨字段匹配，匹配多个字段中是否包含查询词组，对每个字段分别进行打分，然后执行 max 运算获取打分最高的；
- phrase：以 match_phrase 方式运行查询，并返回最佳匹配的评分做为总评分；
- phrase_prefix：以 match_phrase_prefix 方式运行查询，并返回最佳匹配的评分做为总评分；
- bool_prefix：在每个字段上运行 match_bool_prefix 查询，并组合每个字段的评分，详情参考 bool_prefix 以 cross_fields 为例进行实战讲解。

```
#插入测试数据
PUT my_shop
{
  "settings": {
    "number_of_shards": 1,
    "number_of_replicas": 1
  },
  "mappings": {
    "properties": {
      "firstName":{
        "type":"text"
      },
      "lastName":{
        "type":"text"
      }
    }
  }
```

```
}
POST my_shop/_bulk
{"index":{"_id":1}}
{"first_name":"Will","last_name":"Smith","age":25}
{"index":{"_id":2}}
{"first_name":"Smith","last_name":"hello","age":21}
{"index":{"_id":3}}
{"first_name":"Will","last_name":"hello","age":20}
```

```
#查询姓名为 Will Smith 的信息
GET /my_shop/_search
{
  "query": {
    "multi_match" : {
      "query":      "Will Smith",
      "type":       "cross_fields",
      "fields":     [ "first_name^2", "last_name" ],
      "operator":   "and"
    }
  }
}
#返回
    "max_score" : 1.9208363,
    "hits" : [
      {
        "_index" : "my_shop",
        "_type" : "_doc",
        "_id" : "1",
```

```
        "_score" : 1.9208363,
        "_source" : {
          "first_name" : "Will",
          "last_name" : "Smith",
          "age" : 25
        }
      }
    ]
```

另外，first_name 提升了权重，默认为 1。

Term - level 查询

可以使用 Term - level 查询结构化数据，结构化数据如日期范围、IP 地址、价格等,下面分别演示在业务场景中的实际使用。

Exists 查询

返回包含字段索引值的文档：

```
#返回包含 goodsName 字段的索引文档
GET /my_goods/_search
{
  "query": {
    "exists": {
      "field": "goodsName"
```

```
      }
    }
  }
```

Fuzzy 查询

返回包含与搜索字词相似的字词的文档，可以用于查询纠错功能。

Edit distance 指的是最小编辑距离，指的是两个字符串之间，由一个字符串转换为另外一个字符串，所需要的最少编辑次数，也叫：Levenshtein ，

参考地址：https://en.wikipedia.org/wiki/Levenshtein_distance

一些查询和 APIs 支持参数去做不精准查询操作，此时可以使用 fuzziness 参数

- 0、1、2 表示最大允许可编辑距离

AUTO 根据词项的长度确定可编辑距离数值，有两种可选参数，AUTO:[low] 和 [high]，用于分别表示短距离参数与长距离参数，未指定情况下，默认值是 3 和 6。

- 0..2 单词长度为 0 到 2 个字母之间时，必须要精确匹配
- 3..5 单词长度 3 到 5 个字母时，最大编辑距离为 1
- 5 单词长度大于 5 个字母时，最大编辑距离为 2

```
#以官网例子举例说明
POST /my_index/_bulk
{ "index": { "_id": 1 }}
{ "text": "Surprise me!"}
{ "index": { "_id": 2 }}
{ "text": "That was surprising."}
{ "index": { "_id": 3 }}
{ "text": "I wasn't surprised."}

GET /my_index/_search
{
  "query": {
    "fuzzy": {
      "text": {
        "value": "surprize",
        "prefix_length": 1
      }
    }
  }
}
#返回
"hits" : [
      {
        "_index" : "my_index",
        "_type" : "my_type",
        "_id" : "1",
        "_score" : 0.9559981,
        "_source" : {
          "text" : "Surprise me!"
```

```
      }
    },
    {
      "_index" : "my_index",
      "_type" : "my_type",
      "_id" : "3",
      "_score" : 0.69983494,
      "_source" : {
        "text" : "I wasn't surprised."
      }
    }
```

默认如果不设置，prefix_length 就是 0

- surprising 未被搜索到，原因是默认的 auto 只允许两个编辑错误，因为 surprize 的长度大于 5，确切地说有三个编辑距离（需要有三次编辑），不能纠错。
- surprize 拼写错误，s->z，错误在一个位置，在 2 个位置的纠错范围之内为提高性能，可以设置 max_expansions，将限制产生模糊文档的个数;
- prefix_length 不宜设置过大，也将影响查询性能，同时错误过多，也将导致查询结果不是用户期望的。

fuziness 实际上采用的是 auto，允许有两个编辑距离，假设采用如下的查询，将只有一个结果：

```
GET /my_index/_search
{
  "query": {
```

```
      "fuzzy": {
        "text": {
          "value": "surprize",
          "fuzziness": "1",
          "prefix_length": 1
        }
      }
    }
}

#返回:
{
  "took" : 19,
  "timed_out" : false,
  "_shards" : {
    "total" : 1,
    "successful" : 1,
    "skipped" : 0,
    "failed" : 0
  },
  "hits" : {
    "total" : {
      "value" : 1,
      "relation" : "eq"
    },
    "max_score" : 0.9559981,
    "hits" : [
      {
        "_index" : "my_index",
```

```
            "_type" : "my_type",
            "_id" : "1",
            "_score" : 0.9559981,
            "_source" : {
              "text" : "Surprise me!"
            }
          }
      ]
    }
}
```

Ids 查询

范围文档包含 ID 的文档信息：

```
GET /my_goods/_search
{
  "query": {
    "ids" : {
      "values" : ["1", "4", "5"]
    }
  }
}
```

Prefix 查询

返回在提供的字段中包含特定前缀的文档：

```
PUT my_shop_test
{
  "settings": {
    "number_of_shards": 1,
    "number_of_replicas": 1
  },
  "mappings": {
    "properties": {
      "shopName":{
        "type":"text"
      },
      "shopCode":{
        "type":"text"
      }
    }
  }
}

#添加测试数据
POST my_shop_test/_bulk
{"index":{"_id":1}}
{"shopName":"box","shopCode":"Smith"}
{"index":{"_id":2}}
{"shopName":"black","shopCode":"jack"}
{"index":{"_id":3}}
{"shopName":"fox","shopCode":"act"}
{"index":{"_id":4}}
{"shopName":"booex","shopCode":"act"}
```

```
#
GET /my_shop_test/_search
{
  "query": {
    "prefix": {
      "shopName": {
        "value": "bo"
      }
    }
  }
}
#返回
"hits" : [
      {
        "_index" : "my_shop_test",
        "_type" : "_doc",
        "_id" : "1",
        "_score" : 1.0,
        "_source" : {
          "shopName" : "box",
          "shopCode" : "Smith"
        }
      },
      {
        "_index" : "my_shop_test",
        "_type" : "_doc",
        "_id" : "4",
        "_score" : 1.0,
```

```
        "_source" : {
          "shopName" : "booex",
          "shopCode" : "act"
        }
      }
    ]
```

Range 查询

Range 查询类似数据库中的 大于、小于范围查询：

```
GET my_goods/_search
{
  "query": {
    "range": {
      "publicPrice": {
        "gte": 2000,
        "lte": 8488
      }
    }
  }
}
```

- gt：大于
- gte：大于等于
- lt：小于
- lte：小于等于

Regexp 查询

正则表达式查询，查询店铺编码以 's' 开头，中间包括任何字符，以及长度且以'1'结尾的数据：

```
GET my_goods/_search
{
  "query": {
    "regexp": {
      "shopCode": {
        "value": "s.*1",
        "flags": "ALL",
        "case_insensitive": true,
        "max_determinized_states": 10000,
        "rewrite": "constant_score"
      }
    }
  }
}
```

Term 查询

```
#返回确切的文档内容，避免对 text 字段类型使用 term
GET my_goods/_search
{
  "query": {
    "term": {
```

```
        "brandName": {
          "value": "三星",
          "boost": 1.0
        }
      }
    }
}
```

Terms 查询

Terms 返回一个或多个包含精确查询条件的文档信息：

```
GET /my_goods/_search
{
  "query": {
    "terms": {
      "brandName": [ "美国", "三星" ],
      "boost": 1.0
    }
  }
}
```

Terms_set 查询

返回最小精确匹配成功的文档信息，terms_set 类似 terms 查询，只不过 terms_se 多定义了返回最小匹配的数量。

```
#新定义商品信息
PUT /my_goods_info
{
  "mappings": {
    "properties": {
      "goodsName": {
        "type": "keyword"
      },
      "sale_property": {
        "type": "keyword"
      },
      "required_matches": {
        "type": "long"
      }
    }
  }
}
#添加 3 条商品测试数据
#销售属性 白色、64G、标品
PUT /my_goods_info/_doc/1?refresh
{
  "name": "apple",
  "sale_property": [ "white", "64","standard" ],
  "required_matches": 2
}
#黑色、32G、非标品
PUT /my_goods_info/_doc/2?refresh
{
```

```
  "name": "apple",
  "sale_property": [ "black", "32","no standard" ],
  "required_matches": 2
}
#黑色、64 非标品
PUT /my_goods_info/_doc/3?refresh
{
  "name": "apple",
  "sale_property": [ "black", "64","no standard" ],
  "required_matches": 2
}
#查询
GET /my_goods_info/_search
{
  "query": {
    "terms_set": {
      "sale_property": {
        "terms": [ "white", "64"],
        "minimum_should_match_field": "required_matches"
      }
    }
  }
}
#返回
"hits" : [
      {
        "_index" : "my_goods_info",
        "_type" : "_doc",
        "_id" : "1",
```

```
        "_score" : 1.1149836,
        "_source" : {
          "name" : "apple",
          "sale_property" : [
            "white",
            "64",
            "standard"
          ],
          "required_matches" : 2
        }
      }
    ]
```

Wildcard 查询

返回包含与通配符模式匹配的术语的文档：

```
GET /my_goods/_search
{
  "query": {
    "wildcard": {
      "shopCode": {
        "value": "sc*1",
        "boost": 1.0,
        "rewrite": "constant_score"
      }
    }
  }
}
```

Geo 查询

Elasticsearch 支持两种 geo 数据：geo_point 经纬度 和 geo_shape 点、线、圆、多边形等复杂图形

- Geo_point

用于查找距离另一个坐标范围内的所有坐标点，或者计算亮点之间的距离用于排序、打分、聚合等操作。

- Geo-shapes

常用于过滤，比如判断两个地理形状是否有重叠或者某个地形是否包含了其他的地理形状

查询分为 4 种类型：

- geo_bounding_box：查找具有落入指定矩形的地理位置的坐标点
- geo_distance：查找地理点在中心点指定距离内的坐标点
- geo_polygon：查找具有指定多边形内的地理点的坐标点
- geo_shape：查找具有以下内容的坐标点：
  - geo-shapes 与指定的几何形状相交，包含于其中或不与指定的几何形状相交的坐标点
  - geo-points 与指定的地理形状相交的坐标点

过滤器将所有文档载入内存，然后每个过滤器执行计算，判断坐标点是否落在指定区域。可见坐标过滤器的代价较昂贵。

最优的做法是先用简单的过滤器尽可能多的过滤掉文档，然后再交给地理坐标过滤器来处理数据。

Geo-bounding box 查询

定义索引对象店铺信息：

```
PUT /my_shop_info
{
  "mappings": {
    "properties": {
      "pin": {
        "properties": {
          "location": {
            "type": "geo_point"
          }
        }
      }
    }
  }
}

#添加 2 条测试数据

PUT /my_shop_info/_doc/1
{
  "pin": {
    "location": {
```

```
        "lat": 40.12,
        "lon": -71.34
      }
    }
}

PUT /my_shop_info/_doc/2
{
  "pin": {
    "location": {
      "lat": 50.12,
      "lon": -61.34
    }
  }
}

#查询指定范围内的数据

GET my_shop_info/_search
{
  "query": {
    "bool": {
      "must": {
        "match_all": {}
      },
      "filter": {
        "geo_bounding_box": {
          "pin.location": {
            "top_left": {
              "lat": 40.73,
              "lon": -74.1
            },
            "bottom_right": {
              "lat": 40.01,
              "lon": -71.12
            }
```

```
            }
          }
        }
      }
    }
}

#返回
"hits" : {
      "total" : {
        "value" : 1,
        "relation" : "eq"
      },
      "max_score" : 1.0,
      "hits" : [
        {
          "_index" : "my_shop_info",
          "_type" : "_doc",
          "_id" : "1",
          "_score" : 1.0,
          "_source" : {
            "pin" : {
              "location" : {
                "lat" : 40.12,
                "lon" : -71.34
              }
            }
          }
        }
      ]
  }
```

Geo-distance 查询

查询仅包含距某个地理点特定距离之内的匹配的坐标，如下所示，查询坐标：

```
#仍然以 my_shop_info 为例
GET /my_shop_info/_search
{
  "query": {
    "bool": {
      "must": {
        "match_all": {}
      },
      "filter": {
        "geo_distance": {
          "distance": "200km",
          "pin.location": {
            "lat": 40,
            "lon": -70
          }
        }
      }
    }
  }
}
```

创作人简介：

李增胜，Elasticsearch 认证工程师、PMP 项目管理认证，现就职于汇通达网络股份有限公司，任产业交易平台交易域技术经理，从事微服务架构、搜索架构方向开发与管理工作。技术关注：电商、产业互联网等领域。

博客：https://www.jianshu.com/u/59dceda66b57

3.4.2.4 分布式计分

创作人：赵震一

什么是打分

搜索引擎中的搜索与数据库中，常规的 SELECT 查询语句，都能帮你从一大堆数据中，找到匹配某个特定关键字的数据条目，但是这两者最大的区别在于，搜索引擎能够基于查询和结果的相关性，帮你做好结果集排序，即搜索引擎会将它认为最符合你查询诉求的数据条目，放在最前面，而数据库的 SELECT 语句却做不到。

那么搜索引擎是怎么做到的呢？其关键在于打分，搜索引擎在完成关键字匹配后，会基于一定的机制对每条匹配的数据（后称文档）进行打分，得分高的文档表示与本次查询相关度高，就会在最后的结果列表中排在靠前的位置，反之则排名靠后，从而帮助你快速找到你最想要的数据。

下面，我们来向 Elasticsearch 插入一些索引数据：

```
#删除已有索引
DELETE /my-index-000001

#创建索引，显示在 settings 中指定 2 个 shard:"number_of_shards": "2"
PUT /my-index-000001
{
  "settings": {
```

```
      "number_of_shards": "2",
      "number_of_replicas": "1"
    },
    "mappings": {
      "properties": {
        "title": {
          "type": "text"
        },
        "date": {
          "type": "date"
        },
        "content": {
          "type": "text"
        }
      }
    }
}

#插入记录
PUT /my-index-000001/_doc/1
{
  "title":    "三国志",
  "date":     "2021-05-01",
  "content": "国别体史书"
}

PUT /my-index-000001/_doc/2
{
  "title":    "红楼梦",
```

```
    "date":      "2021-05-02",
    "content": "黛玉葬花..."
}

PUT /my-index-000001/_doc/3
{
    "title":    "易中天品三国",
    "date":      "2021-05-03",
    "content": "草船借箭、空城计..."
}

PUT /my-index-000001/_doc/4
{
    "title":    "水浒传",
    "date":      "2021-05-03",
    "content": "梁山好汉被团灭..."
}

PUT /my-index-000001/_doc/5
{
    "title":    "三国演义",
    "date":      "2021-05-03",
    "content": "三国时代，群雄逐鹿..."
}
```

接下去，我们采用关键词“三国演义”进行搜索：

```
GET /my-index-000001/_search
{
  "query": {
    "query_string": {
      "query": "三国演义"
    }
  }
}
```

查看一下返回记录的排序以及打分情况：

```
{
  "took" : 4,
  "timed_out" : false,
  "_shards" : {
    "total" : 2,
    "successful" : 2,
    "skipped" : 0,
    "failed" : 0
  },
  "hits" : {
    "total" : {
      "value" : 3,
      "relation" : "eq"
    },
    "max_score" : 3.1212955,
    "hits" : [
      {
```

```
      "_index" : "my-index-000001",
      "_type" : "_doc",
      "_id" : "5",
      "_score" : 3.1212955,
      "_source" : {
        "title" : "三国演义",
        "date" : "2021-05-03",
        "content" : "三国时代，群雄逐鹿..."
      }
    },
    {
      "_index" : "my-index-000001",
      "_type" : "_doc",
      "_id" : "1",
      "_score" : 0.7946176,
      "_source" : {
        "title" : "三国志",
        "date" : "2021-05-01",
        "content" : "国别体史书"
      }
    },
    {
      "_index" : "my-index-000001",
      "_type" : "_doc",
      "_id" : "3",
      "_score" : 0.59221494,
      "_source" : {
        "title" : "易中天品三国",
        "date" : "2021-05-03",
```

```
            "content" : "草船借箭、空城计..."
          }
        }
      ]
    }
  }
```

可以看到 title 为“三国演义”的文档排在第一，分数（ _score ）是 3.1212955，其次是”三国志”，分数是 0.7946176，最后是“易中天品三国”，分数是 0.59221494，其余没有匹配的文档则没有出现，该搜索结果即打分排名基本符合预期。“三国志”的得分较 ”易中天品三国” 的原因是因为 “三国志” 词语较短。

Elasticsearch 搜索的打分机制

众所周知，Elasticsearch 是以 Lucene 作为其搜索引擎技术的核心基石的。为了适应大数据时代的搜索需求，Elasticsearch 对 Lucene 最大的增强在于，将原本的单机搜索能力扩展到了分布式的集群规模能力，即将原本单机无法支撑的索引数据，水平切分成多个可以独立部署在不同机器上的 Shard，每个 Shard 由独立的 Lucene 实例提供服务，从而以集群的形式对外提供搜索服务。

因此，为了便于理解 Elasticsearch 的分布式搜索的打分机制，我们先来简单回顾下单机情况下 Lucene 是如何打分的。

Lucene 打分机制

当我们向 Lucene 某个索引提交搜索请求后，Lucene 会基于查询完成匹配，并得到一个文档结果集，然后默认基于以下的评分公式，来对结果集中的每个条目计算相关度（评分公式可以基于配置调整）。

$$score(q,d) = \sum_{t \text{ in } q} \left(tf(t \text{ in } d) \cdot idf(t)^2 \cdot t.getBoost() \cdot norm(t,d) \right)$$

Lucene Practical Scoring Function

其中 q 表示查询，d 表示当前文档，t 表示 q 中的词条，tf(t in d) 是计算词条 t 在文档 d 中的词频，idf(t) 是词条 t 在整个索引中的逆文档频率。

我们介绍一下最关键的两个概念，即词频（ TF ）和逆文档频率(IDF)。

词频（ TF ）： 词条在文档中出现的次数

基于特定的 q 和文档 d 来说，词条 t 代指 q 分词后的其中一个词条，t 的词频指该 t 词条在文档 d 中的出现次数，出现次数越多，表示该文档相对于该词关联度更高。

逆文档频率（IDF ）： 在同一索引中存在该词条的文档数的倒数

包含某个词条的文档数越多，说明这个词条的词频在整个索引中的影响力越弱。

对于该公式的其他各项的含义，本小节不作深入介绍，我们仅需了解，一旦给定查询 q 和文档 d，其得分即为查询中每个词条 t 的得分总和。

而每个词条的得分，一个主要部分是该词条在文档 d 中的词频（TF）乘以逆文档频率（IDF）的平方。即词条在文档中出现的频率越高，则得分越高。而索引中存在该词条的文档越少，逆文档频率则越高，表示该词条越罕见，那么对应的分数也将越高。

回顾完 Lucene 的打分机制，我们再回过来看下 ES 的搜索及打分机制。

Elasticsearch 打分机制

Elasticsearch 的搜索类型有两种，默认的称为 QUERY_THEN_FETCH。顾名思义，它的搜索流程分为两个阶段，分别称之为 Query 和 Fetch 。

我们来看下 QUERY_THEN_FETCH 的流程：

Query 阶段：

- Elasticsearch 在收到客户端搜索请求后，会由协调节点将请求分发到对应索引的每个 Shard 上。
- 每个 Shard 的 Lucene 实例基于本地 Shard 内的 TF/IDF 统计信息，独立完成 Shard 内的索引匹配和打分（基于上述公式），并根据打分结果完成单个 Shard 内的排序、分页。

- 每个 Shard 将排序分页后的结果集的元数据（文档 ID 和分数，不包含具体的文档内容）返回给协调节点。
- 协调节点完成整体的汇总、排序以及分页，筛选出最终确认返回的搜索结果。

Fetch 阶段：

- 协调节点根据筛选结果去对应 shard 拉取完整的文档数据
- 整合最终的结果返回给用户客户端

分布式打分的权衡

我们再来看一个场景，先重建索引，但是我们将 Shard 建成 3：

```
#删除已有索引
DELETE /my-index-000001

#创建索引，显示在 settings 中指定 3 个 shard:"number_of_shards": "3"
PUT /my-index-000001
{
  "settings": {
    "number_of_shards": "3",
    "number_of_replicas": "1"
  },
  "mappings": {
    "properties": {
      "title": {
```

```
            "type": "text"
        },
        "date": {
            "type": "date"
        },
        "content": {
            "type": "text"
        }
      }
    }
}

#插入记录
PUT /my-index-000001/_doc/1
{
  "title":    "三国志",
  "date":      "2021-05-01",
  "content": "国别体史书"
}

PUT /my-index-000001/_doc/2
{
  "title":    "红楼梦",
  "date":      "2021-05-02",
  "content": "黛玉葬花..."
}

PUT /my-index-000001/_doc/3
{
```

```
  "title":    "易中天品三国",
  "date":     "2021-05-03",
  "content": "草船借箭、空城计..."
}

PUT /my-index-000001/_doc/4
{
  "title":    "水浒传",
  "date":     "2021-05-03",
  "content": "梁山好汉被团灭..."
}

PUT /my-index-000001/_doc/5
{
  "title":    "三国演义",
  "date":     "2021-05-03",
  "content": "三国时代，群雄逐鹿..."
}
```

然后再次执行相同的搜索：

```
GET /my-index-000001/_search
{
  "query": {
    "query_string": {
      "query": "三国演义"
    }
  }
}
```

查看本次搜索结果：


```
{
  "took" : 6,
  "timed_out" : false,
  "_shards" : {
    "total" : 3,
    "successful" : 3,
    "skipped" : 0,
    "failed" : 0
  },
  "hits" : {
    "total" : {
      "value" : 3,
      "relation" : "eq"
    },
    "max_score" : 1.6285465,
    "hits" : [
      {
        "_index" : "my-index-000001",
        "_type" : "_doc",
        "_id" : "3",
        "_score" : 1.6285465,
        "_source" : {
          "title" : "易中天品三国",
          "date" : "2021-05-03",
          "content" : "草船借箭、空城计..."
        }
```

```
        },
        {
          "_index" : "my-index-000001",
          "_type" : "_doc",
          "_id" : "5",
          "_score" : 1.1507283,
          "_source" : {
            "title" : "三国演义",
            "date" : "2021-05-03",
            "content" : "三国时代，群雄逐鹿..."
          }
        },
        {
          "_index" : "my-index-000001",
          "_type" : "_doc",
          "_id" : "1",
          "_score" : 0.5753642,
          "_source" : {
            "title" : "三国志",
            "date" : "2021-05-01",
            "content" : "国别体史书"
          }
        }
      ]
    }
}
```

搜索结果的排名竟然发生了变化，我们期望排第一的”三国演义”排到了第二，得分为 1.1507283，而“易中天品三国”竟然得分 1.6285465，跃居第一，这并不符合我们的搜索预期。

通过分析上面的 QUERY_THEN_FETCH 流程，我们不难发现：由于分布式系统天然的割裂性质，每个 shard 无法看到全局的统计信息，所以上述第 2 步中每个 Shard 的打分都是基于本地 Shard 内的 TF/IDF 统计信息来完成的。

在大多数的生产环境中，由于数据量多且在每个 Shard 分布均匀，这种方式是没有问题的。但是在极端情况下（如上例），3 个 shard 中的文档数相差较大，那么 IDF 在 3 个 Shard 中所起到的影响将截然不同，即单个 Shard 内打分汇总后的结果，与全局打分汇总的结果会有相当大的出入，造成我们在靠前的分页，搜到原本应该排名靠后的文档。

这也是分布式打分引入的实际问题，那么如何才能解决这类问题呢？

我们曾在上一小节提到，Elasticsearch 的搜索类型其实有两种，除了上面介绍的 QUERY_THEN_FETCH 之外，还有一种是 DFS_QUERY_THEN_FETCH。

DFS 在这里的意思是分布式频率打分，其思想是提前向所有 Shard 进行全局的统计信息搜集，然后再将这些统计信息，随着查询分发到各个 Shard，让各个 Shard 在本地采用全局 TF/IDF 来打分，具体的流程如下：

预统计阶段：

- Elasticsearch 在收到客户端搜索请求后，会由协调节点进行一次预统计工作，即先向所有相关 Shard 搜集统计信息

Query 阶段：

- 由协调节点整合所有统计信息，将全局的统计信息连同请求一起分发到对应索引的每个 Shard 上。
- 每个 Shard 的 Lucene 实例，基于全局的 TF/IDF 统计信息，独立完成 Shard 内的索引匹配和打分（基于上述公式），并根据打分结果，完成单个 Shard 内的排序、分页。
- 每个 Shard 将排序分页后的结果集的元数据（文档 ID 和分数，不包含具体的文档内容）返回给协调节点。
- 协调节点完成整体的汇总、排序以及分页，筛选出最终确认返回的搜索结果。

Fetch 阶段：

- 协调节点根据筛选结果去对应 shard 拉取完整的文档数据
- 整合最终的结果返回给用户客户端

综上可见，Elasticsearch 在分布式打分上做了权衡，如果要考虑绝对的精确性，那么需要牺牲一些性能来换取全局的统计信息。

让我们来看下如何切换到 DFS_QUERY_THEN_FETCH，只需在接口 URL 加上 search_type=dfs_query_then_fetch

```
GET /my-index-000001/_search?search_type=dfs_query_then_fetch
{
  "query": {
    "query_string": {
      "query": "三国演义"
    }
  }
}
```

可以看到，通过这种方式返回的结果又恢复了正常：

```
{
  "took" : 9,
  "timed_out" : false,
  "_shards" : {
    "total" : 3,
    "successful" : 3,
    "skipped" : 0,
    "failed" : 0
  },
  "hits" : {
    "total" : {
      "value" : 3,
      "relation" : "eq"
    },
    "max_score" : 3.7694218,
    "hits" : [
      {
```

```
      "_index" : "my-index-000001",
      "_type" : "_doc",
      "_id" : "5",
      "_score" : 3.7694218,
      "_source" : {
        "title" : "三国演义",
        "date" : "2021-05-03",
        "content" : "三国时代，群雄逐鹿..."
      }
    },
    {
      "_index" : "my-index-000001",
      "_type" : "_doc",
      "_id" : "1",
      "_score" : 1.1795839,
      "_source" : {
        "title" : "三国志",
        "date" : "2021-05-01",
        "content" : "国别体史书"
      }
    },
    {
      "_index" : "my-index-000001",
      "_type" : "_doc",
      "_id" : "3",
      "_score" : 0.8715688,
      "_source" : {
        "title" : "易中天品三国",
        "date" : "2021-05-03",
```

```
            "content" : "草船借箭、空城计..."
          }
        }
      ]
    }
}
```

“三国演义”的文档仍排在第一，分数（ _score ）变成了 3.7694218，其次是“三国志”，分数是 1.1795839，最后是“易中天品三国”，分数是 0.8715688，其余没有匹配的文档同样没有出现。

另外，根据返回的 took 数据，可以看到耗时较 query_then_fetch 的方式有略为增加，所以这种方式对性能会有折损，在生产环境中建议谨慎使用。

查看得分逻辑

为了在实际开发中了解得分逻辑，从而优化我们的查询条件或索引工作，我们需要关注例如“易中天品三国”为什么分数是 0.8715688，而不是 3.7694218。

我们可以通过在查询中增加 explain 来查看得分的说明信息。

```
GET /my-index-000001/_search?search_type=dfs_query_then_fetch
{
  "query": {
    "query_string": {
      "query": "三国演义"
```

```
      }
    },
    "explain": true
  }
```

通过增加 "explain": true，我们可以看到返回的结果集里增加了大量 _explanation 信息：

```
{
  "took" : 21,
  "timed_out" : false,
  "_shards" : {
    "total" : 3,
    "successful" : 3,
    "skipped" : 0,
    "failed" : 0
  },
  "hits" : {
    "total" : {
      "value" : 3,
      "relation" : "eq"
    },
    "max_score" : 3.7694218,
    "hits" : [
      {
        "_shard" : "[my-index-000001][0]",
        "_node" : "ydZx8i8HQBe69T4vbYm30g",
        "_index" : "my-index-000001",
```

```
            "_type" : "_doc",
            "_id" : "5",
            "_score" : 3.7694218,
            "_source" : {
              "title" : "三国演义",
              "date" : "2021-05-03",
              "content" : "三国时代，群雄逐鹿..."
            },
            "_explanation" : {
              "value" : 3.7694218,
              "description" : "max of:",
              "details" : [
                {
                  "value" : 3.7694218,
                  "description" : "sum of:",
                  "details" : [
                    {
                      "value" : 0.52763593,
                      "description" : "weight(title:三 in 0) [PerFieldSimilarity], result of:",
                      "details" : [
                        {
                          "value" : 0.52763593,
                          "description" : "score(freq=1.0), computed as boost * idf * tf fro
m:",
                          "details" : [
                            {
                              "value" : 2.2,
                              "description" : "boost",
                              "details" : [ ]
```

```
                    },
                    {
                      "value" : 0.5389965,
                      "description" : "idf, computed as log(1 + (N - n + 0.5) / (n +
 0.5)) from:",
                      "details" : [
                        {
                          "value" : 3,
                          "description" : "n, number of documents containing ter
m",
                          "details" : [ ]
                        },
                        {
                          "value" : 5,
                          "description" : "N, total number of documents with field
",
                          "details" : [ ]
                        }
                      ]
                    },
                    {
                      "value" : 0.4449649,
                      "description" : "tf, computed as freq / (freq + k1 * (1 - b +
b * dl / avgdl)) from:",
                      "details" : [
                        {
                          "value" : 1.0,
                          "description" : "freq, occurrences of term within docume
nt",
```

```
                "details" : [ ]
              },
              {
                "value" : 1.2,
                "description" : "k1, term saturation parameter",
                "details" : [ ]
              },
              {
                "value" : 0.75,
                "description" : "b, length normalization parameter",
                "details" : [ ]
              },
              {
                "value" : 4.0,
                "description" : "dl, length of field",
                "details" : [ ]
              },
              {
                "value" : 3.8,
                "description" : "avgdl, average length of field",
                "details" : [ ]
              }
            ]
          }
        ]
      }
    ]
  },
```

```
                    {
                      "value" : 1.357075,
                      "description" : "weight(title:演 in 0) [PerFieldSimilarity], result of:",
                      "details" : [
                        {
                          "value" : 1.357075,
                          "description" : "score(freq=1.0), computed as boost * idf * tf fro
m:",
                          "details" : [
                            {
                              "value" : 2.2,
                              "description" : "boost",
                              "details" : [ ]
                            },
                            {
                              "value" : 1.3862944,
                              "description" : "idf, computed as log(1 + (N - n + 0.5) / (n +
 0.5)) from:",
                              "details" : [
                                {
                                  "value" : 1,
                                  "description" : "n, number of documents containing ter
m",
                                  "details" : [ ]
                                },
                                {
                                  "value" : 5,
                                  "description" : "N, total number of documents with field
",
```

```
            "details" : [ ]
          }
        ]
      },
      {
        "value" : 0.4449649,
        "description" : "tf, computed as freq / (freq + k1 * (1 - b +
b * dl / avgdl)) from:",
        "details" : [
          {
            "value" : 1.0,
            "description" : "freq, occurrences of term within docume
nt",
            "details" : [ ]
          },
          {
            "value" : 1.2,
            "description" : "k1, term saturation parameter",
            "details" : [ ]
          },
          {
            "value" : 0.75,
            "description" : "b, length normalization parameter",
            "details" : [ ]
          },
          {
            "value" : 4.0,
            "description" : "dl, length of field",
            "details" : [ ]
```

```
                                },
                                {
                                  "value" : 3.8,
                                  "description" : "avgdl, average length of field",
                                  "details" : [ ]
                                }
                              ]
                            }
                          ]
                        }
                      ]
                    },
                    ...
                  ]
                },
                ...
              ]
            }
          },
          ...
        ]
      }
    }
```

通过分析 description 和 details 中信息的描述，我们可以进一步深挖 Elasticsearch 的打分逻辑和我们查询出来的每个文档的得分详情。

创作人简介：

赵震一，程序员，好奇技淫巧，关注大数据与分布式计算。

3.4.2.5 Object 数据类型

创作人：李增胜

在某些业务下，设计索引 Mapping 时，需要设计的对象中包含对象（非数组），此时就可以使用 Object 类型来存储对象。

以下定义了店铺对象，包含店铺名称、店铺编码、供应商信息，另外供应商信息中又包含供应商编码、供应商名称，同时供应商信息还包含自身的对象属性所在区域，所在区域又包含省和市，这种定义才能满足查询店铺信息、查询供应商所有店铺信息，以及查询某地区的所有店铺信息等等场景。

在如下示例中，supplier 是索引 my_shop 中的一个 Object，area 又是 supplier 的一个 Object

在访问 area 时，需要通过 supplier.area 才能访问：

```
#定义 mapping
PUT my_shop
{
  "settings": {
    "index": {
      "number_of_shards": 1,
      "number_of_replicas": 1
    }
```

```
    },
    "mappings": {
      "properties": {
        "shopName": {
          "type": "text",
          "analyzer": "ik_smart"
        },
        "shopCode": {
          "type": "keyword"
        },
        "supplier": {
          "properties": {
            "supplier_code": {
              "type": "keyword"
            },
            "supplier_name": {
              "type": "text",
              "analyzer": "ik_smart"
            },
            "area": {
              "properties": {
                "province": {
                  "type": "keyword"
                },
                "city": {
                  "type": "keyword"
                }
              }
            }
```

```
            }
          }
        }
      }
    }

    #插入测试数据
    POST my_shop/_bulk
    {"index":{"_id":1}}
    {"shopName":"苹果热销店铺","shopCode":"sc001","supplier":{"supplier_code":"001","supplier_name":"南京农村电商领导者","area":{"province":"江苏省","city":"南京市"}}}
    {"index":{"_id":2}}
    {"shopName":"美的热销店铺","shopCode":"sc002","supplier":{"supplier_code":"001","supplier_name":"南京农村电商领导者","area":{"province":"江苏省","city":"南京市"}}}
    {"index":{"_id":3}}
    {"shopName":"金沙酒热销店铺","shopCode":"sc003","supplier":{"supplier_code":"002","supplier_name":"山东农村电商领导者","area":{"province":"江苏省","city":"南京市"}}}
    {"index":{"_id":4}}
    {"shopName":"华为热销店铺","shopCode":"sc004","supplier":{"supplier_code":"002","supplier_name":"山东农村电商领导者","area":{"province":"山东省","city":"青岛市"}}}
```

测试数据包括 2 家供应商：

- 南京农村电商领导者 店铺：苹果热销店铺 + 美的热销店铺
- 山东农村电商领导者 店铺：金沙酒热销店铺 + 华为热销店铺

查询供应商 001 对应的所有店铺：

```
#访问时，需要 supplier.supplier_code 指定对象对应的字段
POST  my_shop/_search
{
  "query": {
    "match": {
      "supplier.supplier_code": "001"
    }
  }
}
#返回
{
  "took" : 1,
  "timed_out" : false,
  "_shards" : {
    "total" : 1,
    "successful" : 1,
    "skipped" : 0,
    "failed" : 0
  },
  "hits" : {
    "total" : {
      "value" : 2,
      "relation" : "eq"
    },
    "max_score" : 0.6931471,
    "hits" : [
      {
        "_index" : "my_shop",
```

```
      "_type" : "_doc",
      "_id" : "1",
      "_score" : 0.6931471,
      "_source" : {
        "shopName" : "苹果热销店铺",
        "shopCode" : "sc001",
        "supplier" : {
          "supplier_code" : "001",
          "supplier_name" : "南京农村电商领导者",
          "area" : {
            "province" : "江苏省",
            "city" : "南京市"
          }
        }
      }
    },
    {
      "_index" : "my_shop",
      "_type" : "_doc",
      "_id" : "2",
      "_score" : 0.6931471,
      "_source" : {
        "shopName" : "美的热销店铺",
        "shopCode" : "sc002",
        "supplier" : {
          "supplier_code" : "001",
          "supplier_name" : "南京农村电商领导者",
          "area" : {
            "province" : "江苏省",
```

```
                "city" : "南京市"
              }
            }
          }
        }
      ]
    }
}

#查询销售区域在南京的所有店铺
#通过 supplier.area.city 访问对应的字段值
POST my_shop/_search
{
  "query": {
    "match": {
      "supplier.area.city": "南京市"
    }
  }
}

#返回
{
  "hits" : {
    "total" : {
      "value" : 3,
      "relation" : "eq"
    },
    "max_score" : 0.35667494,
    "hits" : [
```

```
{
  "_index" : "my_shop",
  "_type" : "_doc",
  "_id" : "1",
  "_score" : 0.35667494,
  "_source" : {
    "shopName" : "苹果热销店铺",
    "shopCode" : "sc001",
    "supplier" : {
      "supplier_code" : "001",
      "supplier_name" : "南京农村电商领导者",
      "area" : {
        "province" : "江苏省",
        "city" : "南京市"
      }
    }
  }
},
{
  "_index" : "my_shop",
  "_type" : "_doc",
  "_id" : "2",
  "_score" : 0.35667494,
  "_source" : {
    "shopName" : "美的热销店铺",
    "shopCode" : "sc002",
    "supplier" : {
      "supplier_code" : "001",
      "supplier_name" : "南京农村电商领导者",
```

```
            "area" : {
              "province" : "江苏省",
              "city" : "南京市"
            }
          }
        }
      },
      {
        "_index" : "my_shop",
        "_type" : "_doc",
        "_id" : "3",
        "_score" : 0.35667494,
        "_source" : {
          "shopName" : "金沙酒热销店铺",
          "shopCode" : "sc003",
          "supplier" : {
            "supplier_code" : "002",
            "supplier_name" : "山东农村电商领导者",
            "area" : {
              "province" : "江苏省",
              "city" : "南京市"
            }
          }
        }
      }
    ]
  }
}
```

3.4.2.6 Join 数据类型

创作人：李增胜

Join 类型是一种特殊的数据类型，类似父子结构，一个子文档只能有一个父文档，一个父文档可以有多个子文档。

使用场景

Join 可以实现父子文档的关系存储，在什么情况下使用 Join 类型呢？假设我们存在这种场景，售卖的商品有评价信息，商品信息不会经常发生变更，但是评论信息就更新的比较频繁了，此时就可以使用 Join 数据类型来处理此种业务， 一对多关系存在多个文档中，父子文档更新性能高，可独立更新，互不影响。

在实际使用场景中，推荐使用 Data denormalization 来解决过多关联查询问题，字面解读就是”非规范化存储”，通过冗余存储多字段来达到过多关联的查询问题，避免使用 Join 数据类型，虽然带来了关联的方便性，但是会带来额外的查询开销影响搜索性能。

此外，Kibana 对 Join 以及 Nested 的支持也比较少：

```
#定义索引，my_goods_sale 为售卖的商品信息，my_goods_comment 为商品的评价信息
PUT my_goods_hot_sale
{
```

```
  "mappings": {
    "properties": {
      "my_id": {
        "type": "keyword"
      },
      "my_join_field": {
        "type": "join",
        "relations": {
          "my_goods_sale": "my_goods_comment"
        }
      }
    }
  }
}

#添加商品售卖 ID 为 1 的信息
PUT my_goods_hot_sale/_doc/1?refresh
{
  "my_id": "1",
  "text": "This is a my_goods_sale",
  "my_join_field": {
    "name": "my_goods_sale"
  }
}

#添加商品售卖 ID 为 2 的信息
PUT my_goods_hot_sale/_doc/2?refresh
{
  "my_id": "2",
```

```
  "text": "This is another my_goods_sale",
  "my_join_field": {
    "name": "my_goods_sale"
  }
}

#添加商品售卖 ID 为 3，父商品为 1，注意父子文档一定要在一个 shard 上
PUT my_goods_hot_sale/_doc/3?routing=1&refresh
{
  "my_id": "3",
  "text": "This is an comment",
  "my_join_field": {
    "name": "my_goods_comment",
    "parent": "1"
  }
}

#添加商品售卖 ID 为 4，父商品为 1
PUT my_goods_hot_sale/_doc/4?routing=1&refresh
{
  "my_id": "4",
  "text": "This is another comment",
  "my_join_field": {
    "name": "my_goods_comment",
    "parent": "1"
  }
}
```

根据父文档查询子文档：

```
GET my_goods_hot_sale/_search
{
  "query": {
    "has_parent": {
      "parent_type": "my_goods_sale",
      "query": {
        "match": {
          "text": "my_goods_sale"
        }
      }
    }
  }
}
```

根据子文档查询父文档：

```
GET my_goods_hot_sale/_search
{
  "query": {
    "has_child": {
      "type": "my_goods_comment",
      "query": {
        "match_all": {}
      }
    }
  }
}
```

3.4.2.7 Nested 数据类型

创作人：李增胜

Nested 是 Object 的专用版本，允许对象数组可以以彼此独立查询的方式进行索引。

Elasticsearch 中其实是没有内部对象的概念，因此它将对象层次结构，简化为字段名称和值，以列表的形式展现。

首先来比较 Nested 与 Join 以及 Object 的区别：

对比结果	Nested Object	Join	Object
优点	一对多关系存在一个文档中，查询速度较高	一对多关系存在多个文档中，父子文档更新性能高，可独立更新，互不影响	存储单个对象，性能高
缺点	无法单独更新父子文档，必须更新整个文档	维护关系需要占用更多内存，读取性能不高	当对象为数组时自动扁平化处理，无法满足多场景的查询条件，扩展性差
适用场景	查询性能高要求，子文档偶尔更新	子文档更新高频场景	对象非数组类型

小结：

Nested 类型使用场景：

- 含有 Object 数组。
- 需要对 Object 中的字段（至少两个及以上）同时进行查询，并维护这种关系。

Nested 类型允许相互独立地对对象数组进行索引和查询。如果需要维护数组中每个对象的关系，请使用 nested 数据类型。

以 B2B 电商行业的实际业务场景来举例说明，2B 行业的交易具有一定封闭性，只有签署合同、经常往来交易的会员，往往有更高资格的交易权、议价权。

定义商品索引，其中 groupPrice 标识分组价对象，对象里面包含了 boxLevelPrice 分组价格、level 分组级别。当前端业务线搜索时，传入用户所在组级别，即可查询对应的价格。

为了便于区分我们先定义为 Object 类型来观察下现象：

定义分组为 Object 类型

其中 groupPrice 为数组 Object 数据结构类型：

```
PUT goods_info_object
{
  "mappings": {
    "properties": {
      "goodsName": {
        "type": "text",
        "analyzer": "ik_smart"
      },
      "skuCode": {
        "type": "keyword"
      },
      "brandName": {
        "type": "keyword"
      },
      "shopCode": {
        "type": "keyword"
      },
      "publicPrice": {
        "type": "float"
      },
      "groupPrice": {
        "properties": {
          "boxLevelPrice": {
            "type": "keyword"
          },
          "level": {
            "type": "keyword"
          }
```

```
        }
      }
    }
  }
}

#插入测试数据，为了便于阅读 JSON 格式进行了展开
POST goods_info_object/_bulk
{
  "index": {
    "_id": 1
  }
}
{
  "goodsName": "美国苹果",
  "skuCode": "skuCode1",
  "brandName": "美国苹果",
  "shopCode": "sc00001",
  "publicPrice": "8388.88",
  "groupPrice": [
    {
      "boxLevelPrice": "4888.00",
      "level": "A"
    },
    {
      "boxLevelPrice": "6888.00",
      "level": "B"
    }
  ]
```

```
}
{
  "index": {
    "_id": 2
  }
}
{
  "goodsName": "山东苹果",
  "skuCode": "skuCode2",
  "brandName": "山东苹果",
  "shopCode": "sc00001",
  "publicPrice": "7388.88",
  "groupPrice": [
    {
      "boxLevelPrice": "5888.00",
      "level": "A"
    },
    {
      "boxLevelPrice": "4888.00",
      "level": "B"
    }
  ]
}

#检索 A 组且价格等于 4888.00 的商品
POST goods_info_object/_search
{
  "query": {
    "bool": {
```

```
        "must": [
          {
            "match": {
              "groupPrice.level": "A"
            }
          },
          {
            "match": {
              "groupPrice.boxLevelPrice": "4888.00"
            }
          }
        ]
      }
    }
}

#返回:
{
  "took" : 1,
  "timed_out" : false,
  "_shards" : {
    "total" : 1,
    "successful" : 1,
    "skipped" : 0,
    "failed" : 0
  },
  "hits" : {
    "total" : {
      "value" : 2,
```

```
    "relation" : "eq"
  },
  "max_score" : 0.45840856,
  "hits" : [
    {
      "_index" : "goods_info_object",
      "_type" : "_doc",
      "_id" : "1",
      "_score" : 0.45840856,
      "_source" : {
        "goodsName" : "美国苹果",
        "skuCode" : "skuCode1",
        "brandName" : "美国苹果",
        "shopCode" : "sc00001",
        "publicPrice" : "8388.88",
        "groupPrice" : [
          {
            "boxLevelPrice" : "4888.00",
            "level" : "A"
          },
          {
            "boxLevelPrice" : "6888.00",
            "level" : "B"
          }
        ]
      }
    },
    {
      "_index" : "goods_info_object",
```

```
            "_type" : "_doc",
            "_id" : "2",
            "_score" : 0.45840856,
            "_source" : {
              "goodsName" : "山东苹果",
              "skuCode" : "skuCode2",
              "brandName" : "山东苹果",
              "shopCode" : "sc00001",
              "publicPrice" : "7388.88",
              "groupPrice" : [
                {
                  "boxLevelPrice" : "5888.00",
                  "level" : "A"
                },
                {
                  "boxLevelPrice" : "4888.00",
                  "level" : "B"
                }
              ]
            }
          }
        ]
      }
    }
```

我们查询的数据，要满足分组等级是 A 级且价格为 4888.00 的数据信息。

如下图所示，只有文档 1 是满足的，但是却查询到了 2 条，其中包括不符合条件的文档 2：


```
      "skuCode" : "skuCode1",
      "brandName" : "美国苹果",
      "shopCode" : "sc00001",
      "publicPrice" : "8388.88",
      "groupPrice" : [
        {
          "boxLevelPrice" : "4888.00",
          "level" : "A"
        },
        {
          "boxLevelPrice" : "6888.00",
          "level" : "B"
        }
      ]
    }
  },
  {
    "_index" : "goods_info_object",
    "_type" : "_doc",
    "_id" : "2",
    "_score" : 1.0,
    "_source" : {
      "goodsName" : "山东苹果",
      "skuCode" : "skuCode2",
      "brandName" : "山东苹果",
      "shopCode" : "sc00001",
      "publicPrice" : "7388.88",
      "groupPrice" : [
        {
          "boxLevelPrice" : "5888.00",
          "level" : "A"
        },
        {
          "boxLevelPrice" : "4888.00",
          "level" : "B"
        }
      ]
    }
```

这是因为 Elasticsearch 中将 Object 数组打平了做存储导致，在 Elasticsearch 中，会将数据做如下存储：

```
{
        "goodsName" : "山东苹果",
        "skuCode" : "skuCode2",
        "brandName" : "山东苹果",
        "shopCode" : "sc00001",
        "publicPrice" : "7388.88",
        "groupPrice.boxLevelPrice" :["5888.00","4888.00"],
        "groupPrice.level" :["A","B"]
}
```

查询恰好 boxLevelPrice 为"4888.00" 并且 level 为"A"的文档 2 是能被检索到的，当需要对数组中两个字段进行查询时，就需要用 Nested 数据结构类型来解决此问题。

定义分组为 Nested 数据结构类型

```
PUT goods_info_nested
{
  "mappings": {
    "properties": {
      "goodsName": {
        "type": "text",
        "analyzer": "ik_smart"
```

```
        },
        "skuCode": {
          "type": "keyword"
        },
        "brandName": {
          "type": "keyword"
        },
        "shopCode": {
          "type": "keyword"
        },
        "publicPrice": {
          "type": "float"
        },
        "groupPrice": {
          "type": "nested",
          "properties": {
            "boxLevelPrice": {
              "type": "float"
            },
            "level": {
              "type": "keyword"
            }
          }
        }
      }
    }
}

#插入同样的测试数据
```

```
POST goods_info_nested/_bulk
{"index":{"_id":1}}
{"goodsName":"美国苹果","skuCode":"skuCode1","brandName":"美国苹果","shopCode":"sc00001","publicPrice":"8388.88","groupPrice":[{"boxLevelPrice":"4888.00","level":"A"},{"boxLevelPrice":"6888.00","level":"B"}]}
{"index":{"_id":2}}
{"goodsName":"山东苹果","skuCode":"skuCode2","brandName":"山东苹果","shopCode":"sc00001","publicPrice":"7388.88","groupPrice":[{"boxLevelPrice":"5888.00","level":"A"},{"boxLevelPrice":"4888.00","level":"B"}]}
#查询
POST goods_info_nested/_search
{
  "query": {
    "nested": {
      "path": "groupPrice",
      "query": {
        "bool": {
          "must": [
            {
              "match": {
                "groupPrice.level": "A"
              }
            },
            {
              "match": {
                "groupPrice.boxLevelPrice": "4888.00"
              }
            }
          ]
```

```
            }
        }
    }
  }
}
#返回:
"hits" : [
        {
          "_index" : "goods_info_nested",
          "_type" : "_doc",
          "_id" : "1",
          "_score" : 1.3862942,
          "_source" : {
            "goodsName" : "美国苹果",
            "skuCode" : "skuCode1",
            "brandName" : "美国苹果",
            "shopCode" : "sc00001",
            "publicPrice" : "8388.88",
            "groupPrice" : [
              {
                "boxLevelPrice" : "4888.00",
                "level" : "A"
              },
              {
                "boxLevelPrice" : "6888.00",
                "level" : "B"
              }
            ]
          }
        }
    ]
```

同样查询 groupPrice.boxLevelPrice 为"4888.00" 且 level 为"A"的数据，显然只有文档 1 满足，通过查询也验证了此结论，说明 Nested 查询生效，解决了嵌套查询的问题。

Nested 在 Aggregation 中的应用

在对 Nested Object 进行聚合操作时，我们需要使用到 Nested Aggregation，我们需要聚合查询最大的分组价格(groupPrice)。

```
POST /goods_info_nested/_search
{
  "query": {
    "match": {
      "goodsName": "苹果"
    }
  },
  "aggs": {
    "groupPrice": {
      "nested": {
        "path": "groupPrice"
      },
      "aggs": {
        "max_price": {
          "max": {
            "field": "groupPrice.boxLevelPrice"
          }
```

```
            }
          }
        }
      }
    }

    #返回
    {
        .....
        "aggregations" : {
            "groupPrice" : {
              "doc_count" : 4,
              "max_price" : {
                "value" : 6888.0
              }
            }
         }
    }
```

3.4.2.8 Index template

创作人：骆潇龙

Elasticsearch 本着让用户方便快捷的使用搜索功能的原则，对数据定义（索引定义）做了高度抽象，尽可能得避免了重复性定义工作，使之更加灵活。

Elasticsearch 在这方面做的工作主要体现是索引模板（Index template）和动态映射（Dynamic Mapping）两个功能。索引模板的主要功能，是允许用户在创建索引（index）时，引用已保存的模板来减少配置项。操作的一般过程是先创建索引模板，然后再手动创建索引或保存文档（Document）。而自动创建索引时，索引模板会作为配置的基础作用。对于 X-Pack 的数据流（Data Stream）功能，索引模板用于自动创建后备索引（Backing Indices）。该功能的意义是提供了一种配置复用机制，减少了大量重复作劳动。

目前 Elasticsearch 的索引模板功能以 7.8 版本为界，分为新老两套实现，新老两个版本的主要区别是模板之间复用如何实现。

老版本：

使用优先级（order）关键字实现，当创建索引匹配到多个索引模板时，高优先级会继承并铺盖低优先级的模板配置，最终多个模板共同起作用。

新版本：

删除了 order 关键字，引入了组件模板 Component template 的概念，是第一段可以复用的配置块。在创建普通模板时可以声明引用多个组件模板，当创建索引匹配到多个新版索引模板时，取用最高权重的那个。

下面将以生命周期（创建、查看、使用、删除）为切入点，分别介绍如何使用 Elasticsearch 新老两个版本的索引模板。

新版索引模板

在使用新版本的索引模板功能前，我们应当确认 Elasticsearch 是否开启了安全设置，如果是，那么我们操作的角色对于集群需要有 manage_index_templates 或 manage 权限才能使用索引模板功能，这个限制对新老版本都适用。

对于新版，Elasticsearch 为了方便用户调试，提供了模拟 API 来帮助用户测试创建索引最终的配置。该模拟 API 主要有 2 个

第一个模拟在现有索引模板下创建 1 个索引，最终使用的模板配置是什么样的。

第二个模拟在指定模板的配置下，最终模板配置是什么样的。

第一种模拟 API 使用范例如下：

```
# 创建 1 个组件模板 ct1
PUT /_component_template/ct1                    # 1
```

```
{
  "template": {
    "settings": {
      "index.number_of_shards": 2
    }
  }
}
# 创建 1 个组件模板 ct2
PUT /_component_template/ct2                    # 2
{
  "template": {
    "settings": {
      "index.number_of_replicas": 0
    },
    "mappings": {
      "properties": {
        "@timestamp": {
          "type": "date"
        }
      }
    }
  }
}
# 创建 1 个索引模板 final-template
PUT /_index_template/final-template          # 3
{
  "index_patterns": ["my-index-*"],
  "composed_of": ["ct1", "ct2"],
  "priority": 5,
```

```
  "template":{
  "settings": {
      "index.number_of_replicas": 1
    },
    "mappings": {
      "properties": {
        "name": {
          "type": "keyword"
        }
      }
    }
  }
}
# 验证创建名为 my-index-00000 的索引使用的配置是如何
POST /_index_template/_simulate_index/my-index-00000    # 4
#返回
{
  "template" : {     # 引用模板中的配置有 settings、mappings、aliase
    "settings" : {
      "index" : {
        "number_of_shards" : "2",
        "number_of_replicas" : "1"
      }
    },
    "mappings" : {
      "properties" : {
        "@timestamp" : {
          "type" : "date"
        },
```

```
            "name" : {
              "type" : "keyword"
            }
          }
        },
        "aliases" : { }
      },
      "overlapping" : [   # 5
        {
          "name" : "test-template",
          "index_patterns" : ["my-*"]
        }
      ]
}
```

- 在 #1 处创建了 1 个名为 cr1 的组件模板，该模板设置了索引分片数为 2
- 在 #2 处创建了 1 个名为 cr2 的组件模板，该模板设置了索引由 1 个@timestamp 属性，并且副本数是 0
- 在 #3 处创建了 1 个名为 final-template 的索引模板，它适用于所有以 my-index- 开头的索引
- 在 #4 处向/_index_template/_simulate_index/my-index-00000 发送 POST 请求，测试创建名为 my-index-00000 的索引，下面的 JSON 是使用索引模板的配置，可以看出 template 字段是组件 cr1、cr2 以及索引模板 final-template 全部配置的聚合。
- 在 #5 处的 overlapping 字段表示忽略了名为 test-template 模板的配置。

第二种模拟 API 使用范例如下：

```
# 默认组件模板 ct1、crt 已创建，内容如前
# 验证按照该模板创建的索引时会使用的配置
POST /_index_template/_simulate/<index-template> #1
{  # 2
  "index_patterns":["test-*"],
  "composed_of": ["ct1","ct2"],
  "priority": 5,
  "template": {
    "settings": {
      "index.number_of_replicas": 1
    },
    "mappings": {
      "properties": {
        "name": {
          "type": "keyword"
        }
      }
    }
  }
}
# 返回
{    # 3
  "template" : {
    "settings" : {
      "index" : {
        "number_of_shards" : "2",
```

```
            "number_of_replicas" : "1"
          }
        },
        "mappings" : {
          "properties" : {
            "@timestamp" : {
              "type" : "date"
            },
            "name" : {
              "type" : "keyword"
            }
          }
        },
        "aliases" : { }
      },
      "overlapping" : [
        {
         "name" : "test-template",
         "index_patterns" : ["my-*"]
        }
      ]
    }
```

- 在 #1 向/_index_template/_simulate/<index-template>发送 POST 请求，其中<index-template>为自定义索引模板的名词，该模板并不会实际创建
- 在 #2 处是请求的 body，与创建一个新版索引模板个格式完全一样，后续详细介绍，目前只需要知道它引用了 cr1 和 cr2 两个组件模板
- 在 #3 处为请求返回的 body，可以看出内容是组件模板 cr1、cr2 以及请求 body3 个配置的组合。

创建

新版本索引自动配置功能，主要依托组件模板（component template）和索引模板（index template）2 个概念来完成。下面依次来介绍他们是如何实现的。

组件模板的创建

组件模板是用来构建索引模板的特殊模板，它的内容一般是多个索引模板的公有配置，索引模板在创建时，可以声明引用多个组件模板。

在组件模板中可配置的索引内容有：别名 aliases、配置 settings、映射 mappings 3 个，具体配置方式与创建索引时一致。组件模板只有在被索引模板引用时，才会发挥作用。当我们需要创建或更新一个组件模板时，向/_component_template 路径发送 PUT 请求即可。

具体示例如下：

```
# 创建组件模板
PUT /_component_template/template_1?create=true&master_timeout=30s    #1
{
  "template": {   # 2
    "settings": {
      "number_of_shards": 1
    },
    "mappings": {
```

```
        "_source": {
          "enabled": false
        },
        "properties": {
          "name": {
            "type": "keyword"
          }
        }
      },
      "aliases": {
        "test-index": {}
      }
    },
    "version": 1,      #3
    "_meta": {          #4
      "description1": "for test",
      "description2": "create by phoenix"
    }
}
```

- 在 #1 处向/_component_template/template_1 发送 PUT 请求创建一个名为 template_1 的组件模板，模板名可以任意替换，需要注意的是，Elasticsearch 中预置有 6 个组件模板：logs-mappings、logs-settings、metrics-mappings、metrics-settings、synthetics-mapping、synthetics-settings，建议不要覆盖。同时 url 中还包含 2 个可选的查询参数 create 和 master_timeout
  - create，表示此次请求是否是创建请求，如果为 true 则系统中如果已有同名的会报错，默认为 false，表示请求可以是创建也可能是更新请求。

- master_timeout，表示可以容忍的连接 Elasticsearch 主节点的时间，默认是30s，如果超时则请求报错。

- 在 #2 处 template 的内容是对索引的设置，主要有别名，映射和配置，在本例中配置了索引分片数是 1，_source 不用保存，索引有名为 name 的属性，类型是keyword，同时索引别名有 test-index。
- 在 #3 处是用户指定的组件模板的版本号，为了方便外部管理，此为可选项，默认不会为组件模板增加版本号。
- 在 #4 处是用户为组件模板设置的 meta 信息，该对象字段可随意配置，但不要设置的过大。该字段也是可选的。

索引模板的创建

创建或更新一个索引模板的方式都是向/_index_template 发送 1 个 PUT 请求。索引模板可以配置的内容主要有 3 类，分别是别名 aliases、配置 settings、映射 mappings。这 3 部分的配置方式，与创建索引时的设置完全一样这里不再赘述。

创建索引模板的示例：

```
PUT /_index_template/test_template?create=false&master_timeout=30s #1
{
  "index_patterns" : ["te*"],      #2
  "priority" : 1,     #3
  "composed_of": ["template_1"], #4
  "template": {       #5
    "settings" : {
```

```
        "number_of_shards" : 2
      }
    },
    "version": 2,    #6
    "_meta": {         #7
      "user": "phoenix",
      "time": "2021/05/06"
    }
}
# 测试模板
POST /_index_template/_simulate_index/test      #8
{
    "template" : {
      "settings" : {
        "index" : {
          "number_of_shards" : "2"   # 索引模板覆盖了组件模板的配置
        }
      },
      "mappings" : {
        "_source" : {                  # 使用组件模板的配置
          "enabled" : false
        },
        "properties" : {
          "name" : {            # 使用组件模板的配置
            "type" : "keyword"
          }
        }
      },
      "aliases" : {
```

```
        "test-index" : { }      # 使用组件模板的配置
      }
    }
}
```

- 在 #1 处向/_index_template/test_template 发送 PUT 请求创建索引模板，模板名称为 test_template，名称可任意填写，与组件模板相同，有 2 个可选的查询参数：
  - create，表示此次请求是否是创建请求，如果为 true 则系统中如果已有同名模板则会报错，默认为 false，表示请求可以是创建也可能是更新请求。
  - master_timeout，表示可以容忍的连接 Elasticsearch 主节点的时间，默认是 30s，如果超时则请求报错。
- 在 #2 处 index_patterns 字段用于设置匹配索引的规则，目前仅支持使用索引名称匹配，支持*号作为通配符，该字段是必填字段。需要注意 Elasticsearch 自带了 3 种匹配规则的索引模板：logs-*-*、metrics-*-*、synthetics-*-*，建议不要做相同配置。
- 在 #3 处 priority 字段配置的是模板权重，当 1 个索引名符合多个模板的匹配规则时， 会使用该值最大的做为最终使用的模板，此值默认为 0，Elasticsearch 自带的模板此值是 100。
- 在 #4 处 composed_of 字段用于配置索引模板引用的组件模板，此处引用的组件模板配置自动会增加到该模板中。此例我们引用了之前创建的 template_1 组件。
- 在 #5 处 template 的内容是对索引的设置。
- 在 #6 处是用户指定的索引模板的版本号，为了方便外部管理，此为可选项，默认不会为组件模板增加版本号。
- 在 #7 处是用户为索引模板设置的 Meta 信息，该对象字段可随意配置，但不要设置的过大。该字段也是可选的。

- 在 #8 处用模拟 API 创建名为 test 的索引，我们发现返回的模板包含了模板 test_template 和组件 template_1 的所有配置。

查看

创建完索引模板或组件模板后，我们可以使用 GET 请求再次查看其内容，请求是可以指定名称表示精准找，也可以使用通配符*进行范围查找，如果不指定名称则会返回全部。

```
GET /_component_template/template_1?local=false&master_timeout=30s #1
GET /_component_template/template_* #2
GET /_component_template #3
GET /_index_template/test_template #4
GET /_index_template/test_* #5
GET /_index_template #6
```

- 在 #1 是查看名为 template_1 的组件模板。
- 在 #2 是查看名字以 template_开头的所有组件模板。
- 在 #3 是查看所有组件模板，通过该请求我们可以发现 Elasticsearch 默认创建了很多组件模板，使用时应尽量避免冲突。
- 在 #4 是查看名字为 test_template 的索引模板。
- 在 #5 是查看名字以 test_开头的所有索引模板。
- 在 #6 是查看所有索引模板，通过该请求我们可以发现 Elasticsearch 默认创建了。很多索引模板，使用时应尽量避免冲突。

如在 #1 所示，上述所有请求都可以增加 2 个可选的查询参数：

- local，如果为 true 模板的配置仅从本地节点中获取，默认为 false 表示此次查询结果是 master 节点返回的。
- master_timeout，表示可以容忍的连接 es 主节点的时间，默认是 30s，如果超时则请求报错。

使用

关于组件模板如何使用我们在创建索引模板时已经提及，增加 composed_of 字段声明索引模板引用的组件模板，这样组件模板的配置，就自动增加到了索引模板中。

索引模板的使用主要发生在创建索引的时候，如果创建的索引名与索引模板的 index_patterns 配置相匹配，那么该索引将会在此模板的基础上创建。

示例如下：

```
# 创建名为 test-my-template 的索引
PUT /test-my-template
{
  "settings": {
    "index": {
      "number_of_replicas": 3
    }
  },
```

```
  "mappings": {
    "properties": {
      "desc":{
        "type": "keyword"
      }
    }
  }
}
# 查看索引信息
GET /test-my-template
# 返回
{
  "test-my-template" : {
    "aliases" : {              # 组件模板 template_1 的配置
      "test-index" : { }
    },
    "mappings" : {
      "_source" : {            # 组件模板 template_1 的配置
        "enabled" : false
      },
      "properties" : {
        "desc" : {             # 创建索引时的配置
          "type" : "keyword"
        },
        "name" : {             # 组件模板 template_1 的配置
          "type" : "keyword"
        }
      }
    },
```

```
        "settings" : {
          "index" : {
            "routing" : {
              "allocation" : {
                "include" : {
                  "_tier_preference" : "data_content"
                }
              }
            },
            "number_of_shards" : "3",    # 覆盖了 索引模板 number_of_shards=2 的配置
            "provided_name" : "test-my-template",
            "creation_date" : "1620550940095",
            "number_of_replicas" : "1",
            "uuid" : "7UZKxK56SZ-bAwBeDL-Jvg",
            "version" : {
              "created" : "7100099"
            }
          }
        }
      }
    }
```

本例我们创建了名为 test-my-template 的索引，其索引名以 te 开头，匹配索引模板 test_template 的筛选规则。我们在创建时仅指定了分片数和一个 desc 属性，而在我们使用 GET 请求查看索引信息时可以看出，索引模板中的别名配置、_source 配置和 name 属性的配置被使用了。但是关于分片的设置，由于创建索引时明确指出了，所以模板中的配置 number_of_shards=2 被覆盖为了 3。

由于在向一个不存在的索引中插入数据时，Elasticsearch 会自动创建索引，如果新的索引名匹配某个索引模板的话，也会以模板配置来创建索引。在使用该功能前，请确保 Elasticsearch 集群的 action.auto_create_index 配置是允许你自动创建目标索引的。

示例如下：

```
# 向不存在的 test 索引中保存 1 条数据
PUT /test/_doc/1
{
  "name":"tom",
  "age": 18
}

# 查看 test 索引信息
GET /test
{
  "test" : {
    "aliases" : {            # 组件模板 template_1 的配置
      "test-index" : { }
    },
    "mappings" : {
      "_source" : {          # 组件模板 template_1 的配置
        "enabled" : false
      },
      "properties" : {
        "age" : {            # ES 自动推断创建的属性
```

```
            "type" : "long"
          },
          "name" : {          # 组件模板 template_1 的配置
            "type" : "keyword"
          }
        }
      },
      "settings" : {
        "index" : {
          "routing" : {
            "allocation" : {
              "include" : {
                "_tier_preference" : "data_content"
              }
            }
          },
          "number_of_shards" : "2",       # 使用索引模板的配置
          "provided_name" : "test",
          "creation_date" : "1620551933756",
          "number_of_replicas" : "1",
          "uuid" : "Oop40MPySjKbSKvs0OQwTA",
          "version" : {
            "created" : "7100099"
          }
        }
      }
    }
}
```

如上例所示，我们向不存在的 test 索引中保存 1 条数据，该索引名与索引模板

test_template 的规则相匹配。因此索引的配置与模板配置完全契合，除了模板中没有配置的 age 字段，是通过 Elasticsearch 属性自动创建功能生成的。

删除

当索引模板或组件模板完成了它们的使命，我们应该及时将其删除，避免因遗忘导致创建出的索引出现了预料之外的配置。删除操作非常简单，发送 DELETE 请求即可，删除同样可以使用通配符*一次删除多个，示例如下：

```
DELETE /_component_template/template_1?master_timeout=30s&timeout=30s #1
DELETE /_component_template/template_* #2
DELETE /_index_template/test_template #3
DELETE /_index_template/test_* #4
```

- 在 #1 表示删除名为 template_1 的组件模板。
- 在 #2 表示删除名字以 template_开头的组件模板。
- 在 #3 表示删除名为 test_template 的索引模板。
- 在 #3 表示删除名字以 test_开头的索引模板。

如在 #1 所示，上述所有请求都可以增加 2 个可选的查询参数：

- timeout，表示可以容忍的等待响应时间，默认是 30s，如果超时则请求报错。
- master_timeout，表示可以容忍的连接 Elasticsearch 主节点的时间，默认是 30s，如果超时则请求报错。

老版索引模板

Elasticsearch 7.8 版本之前的索引模板功能，与新版本基本相同，唯一的区别就是在模板的复用方式上。老版本允许在创建索引时，匹配到多个模板，多个模板间根据 order 配置的优先级从低到高依次覆盖。这种方式会造成用户在创建索引时，不能明确知道自己到底用了多少模板，索引配置在继承覆盖的过程中非常容易出错。

下面依然从模板的生命周期出发，介绍如何使用。

创建

与新版本一样，创建或更新一个老版索引模板，只需要向/_template 发送 PUT 请求即可，通过索引模板可配置的字段依然是：别名 aliases、配置 settings、映射 mappings3 个。

具体示例如下：

```
# 创建或更新老模板
PUT /_template/old_template?order=1&create=false&master_timeout=30s  # 1
{
  "index_patterns": ["te*", "bar*"],        #2
  "settings": {                  #3
    "number_of_shards": 1
  },
  "aliases": {                   #4
```

```
      "old-template-index": {}
    },
    "mappings": {                    #5
      "_source": {
        "enabled": false
      },
      "properties": {
        "host_name": {
          "type": "keyword"
        }
      }
    },
    "version": 0                     #6
}
```

- 在 #1 处向/_template/old_template 发送 PUT 请求创建或索引模板，模板名称为 old_template，名称可任意填写，该请求有 4 个可选的查询参数：
  - create，表示此次请求是否是创建请求，如果为 true 则系统中如果已有同名模板会报错，默认为 false，表示请求可以是创建也可能是更新请求。
  - master_timeout，表示可以容忍的连接 Elasticsearch 主节点的时间，默认是 30s，如果超时则请求报错。
  - order，该变量接受一个整数，表示模板的优先级，数字越大优先级越高，相关配置越可能被实际使用，强烈建议每个模板都根据实际情况配置该值，不要使用默认值。
- 在 #2 处 index_patterns 字段用于配置匹配索引的规则，目前仅支持使用索引名称匹配，支持*号作为通配符，该字段是必填字段，可配置多个值表示”或“的关系。

- 在 #3 处 settings 字段用于配置索引属性。

具体规则详见文档:https://www.elastic.co/guide/en/elasticsearch/reference/7.10/index-modules.html#index-modules-settings

- 在 #4 处 aliases 字段用于配置索引的别名。

具体规则详见文档:https://www.elastic.co/guide/en/elasticsearch/reference/7.10/indices-aliases.html

- 在 #5 处 mappings 字段用于配置索引的映射。

具体规则详见文档：https://www.elastic.co/guide/en/elasticsearch/reference/7.10/mapping.html

- 在 #6 处 version 字段是用户指定的索引模板的版本号，为了方便外部管理，此为可选项，Elasticsearch 默认不会为组件模板增加版本号。

查看

我们可以使用 GET 请求，查看其模板内容，同样可以使用名称精确查找，也可以使用通配符部分搜索以及搜索全部。老版本还支持使用 HEAD 请求快速验证 1 个模板是否存在：

```
GET /_template/old_template?local=false&master_timeout=30s&flat_settings=false #1
GET /_template/old_* #2
GET /_template #3
HEAD /_template/old_template #4
```

- 在 #1 是查看名为 old_template 的索引模板
- 在 #2 是查看名字以 old_开头的所有索引模板
- 在 #3 是查看所有索引模板，通过该请求我们可以发现 Elasticsearch 默认创建了很多组件模板，使用时应尽量避免冲突
- 在 #4 是验证名为 old_template 的索引模板是否存在，与上面 3 个请求返回模板内容不同，本请求不返回模板内容，以状态码为 200 表示存在，404 表示不存在

如在#1 所示，上述所有请求都可以增加 3 个可选的查询参数：

- local，如果为 true 组件模板的配置仅从本地节点中获取，默认为 false 表示此次查询结果是 master 节点返回的
- master_timeout，表示可以容忍的连接 Elasticsearch 主节点的时间，默认是 30s，如果超时则请求报错
- flat_settings，表示返回的配置中关于 settings 字段如何展示，如果为 false 则使用标准 JSON 格式展示，默认为 true 表示多级属性会压缩成 1 级属性名，以点分格式展示，比如关于索引分片的设置，如果该变量为 true 则返回为 index.number_of_shards:1

使用

索引模板的使用，主要发生在创建索引的时候，如果创建的索引名与索引模板相匹配，那么该索引将会在此模板的基础上创建。需要注意的是，如果同时匹配到新老两个版本的模板，那么默认使用新版本。如果仅匹配到多个老版模板则根据 order 字段依次覆盖。

```
# 创建索引
PUT /bar-test-old
{
  "settings": {
    "number_of_replicas": 2
  },
  "mappings": {
    "_source": {
      "enabled": true
    },
    "properties": {
      "ip":{
        "type": "ip"
      }
    }
  }
}
# 查看索引
GET /bar-test-old
# 返回
{
  "bar-test-old" : {
```

```
"aliases" : {
  "old-template-index" : { }    # old_template 模板中的配置
},
"mappings" : {
  "_source": {
    "enabled": true    # 创建时的配置覆盖模板的配置
  },
  "properties" : {
    "host_name" : {     #   old_template 模板中的配置
      "type" : "keyword"
    },
    "ip" : {
      "type" : "ip"
    }
  }
},
"settings" : {
  "index" : {
    "routing" : {
      "allocation" : {
        "include" : {
          "_tier_preference" : "data_content"
        }
      }
    },
    "number_of_shards" : "1",      # old_template 模板中的配置
    "provided_name" : "bar-test-old",
    "creation_date" : "1620615937000",
    "number_of_replicas" : "2",
```

```
            "uuid" : "jG3xiiB_S-iFrlwZ7df56g",
            "version" : {
              "created" : "7100099"
            }
          }
        }
      }
    }
```

本例我们创建了名为 bar-test-old 的索引，其索引名以 bar 开头，匹配索引模板 old_template 的筛选规则，我们在创建时仅指定了副本数和 1 个 ip 属性，而使用 GET 请求查看索引信息时可以看出，索引模板中的别名配置、host_name 属性和分片数的配置被使用了。但是关于_source 的设置，由于创建索引时明确指出了，所以模板中的 false 配置被覆盖为了 true。

当匹配到多个老版索引模板时，最终配置为多个模板的组合，当不同模板配置了相同字段时，那么以 order 高的模板配置为准。示例如下：

```
# 创建新版索引模板
PUT /_index_template/new_template    #1
{
    "index_patterns": ["template*"],
    "priority" : 100,
    "template": {
     "mappings": {
       "dynamic": "true"
     },
```

```
    "aliases": {
      "new-template": {}
    }
  }
}
# 创建老版索引模板，不配置 order
PUT /_template/old_template_1      #2
{
  "index_patterns": ["temp*"],
   "mappings": {
      "dynamic": "false"
    },
  "aliases": {
      "old_template_1": {}
    }
}
# 创建老版本索引模板，配置高 order
PUT /_template/old_template_2?order=10   #3
{
  "index_patterns": ["temp-*"],
   "mappings": {
      "dynamic": "strict"
    },
  "aliases": {
      "old_template_2": {}
    }
}
# 创建索引，名称会匹配 new_template 模板和 old_template_1 模板
PUT /template-test     #4
```

```
{
  "settings": {
    "number_of_shards": 2
  },
  "mappings": {
    "properties": {
      "ip":{
        "type": "ip"
      }
    }
  }
}
# 查看索引配置
GET  /template-test  #5
# 返回
{
  "template-test" : {
    "aliases" : {
      "new-template" : { }  #  仅有新版本模板的别名设置
    },
    "mappings" : {
      "dynamic" : "true",
      "properties" : {
        "ip" : {
          "type" : "ip"
        }
      }
    },
    "settings" : {
```

```
        "index" : {
          "routing" : {
            "allocation" : {
              "include" : {
                "_tier_preference" : "data_content"
              }
            }
          },
          "number_of_shards" : "2",
          "provided_name" : "template-test",
          "creation_date" : "1620617781794",
          "number_of_replicas" : "1",
          "uuid" : "0PKYANozRya9zG3LdMG1UA",
          "version" : {
            "created" : "7100099"
          }
        }
      }
    }
}
# 创建索引，名称会匹配 old_template_1 模板和 old_template_1 模板
PUT /temp-test #6
{
  "settings": {
    "number_of_shards": 2
  },
  "mappings": {
    "properties": {
      "ip":{
```

```
            "type": "ip"
          }
        }
      }
    }
# 查看索引配置
GET /temp-test
# 返回
{
  "temp-test" : {
    "aliases" : {
      "old_template_1" : { },   # old_template_1 的配置
      "old_template_2" : { }      # old_template_2 的配置
    },
    "mappings" : {
      "dynamic" : "strict",   # old_template_2 的配置
      "properties" : {
        "ip" : {
          "type" : "ip"
        }
      }
    },
    "settings" : {
        "refresh_interval" : "10s",
        "number_of_shards" : "2",
        "provided_name" : "temp-test",
        "creation_date" : "1620617979932",
        "number_of_replicas" : "1",
        "uuid" : "RwvrdGziT7iVmVutXYPwqA",
```

```
            "version" : {
              "created" : "7100099"
            }
          }
        }
      }
    }
```

- 在 #1 处我们创建了 1 个新版本的索引模板匹配名字，以 template 开头的索引，设置可以动态增加属性并设置别名 new-template。
- 在 #2 处我们创建了 1 个老版本的索引模板匹配名字，以 temp 开头的索引，设置不动态增加属性并设置别名 old_template_1。
- 在 #3 处我们创建了 1 个老版本的索引模板匹配名字，以 temp-开头的索引，设置有新增属性时报错并设置别名 old_template_2，同时配置 order 为 10。
- 在 #4 处我们创建了 1 个名为 template-test 的索引。
- 在 #5 处查看索引 template-test 的配置，该名称会匹配新版 new_template 模板，和老版 old_template_1 模板，根据索引的别名信息，只有新版模板配置的别名可以看出，该索引仅仅应用了 new_template 模板。
- 在 #6 处创建了 1 个名为 temp-test 的索引。
- 在 #7 处查看索引 temp-test 的配置，该名称会匹配老版 old_template_1 和 old_template_2 模板，根据索引的别名信息有 old_template_1 和 old_template_1 可以看出，2 个模板的配置都应用了。通过 dynamic 字段为 strict 我们可以判断该字段使用了 order 较高的模板 old_template_2 的配置。

删除

老版本的索引模板完成使命后，应该及时将其删除，避免创建索引时匹配到不必要的模板，导致最终创建的索引与预期不符。删除操作非常简单，发送 delete 请求即可，删除同样可以使用通配符*一次删除多个，示例如下：

```
DELETE /_template/old_template_1?master_timeout=30s&timeout=30s #1
DELETE /_template/old_template* #2
```

- 在 #1 表示删除名为 old_template_1 的索引模板。
- 在 #2 表示删除名字以 old_template 开头的索引模板。
- 在 如 #1 所示，上述 2 个请求都可以增加 2 个可选的查询参数。
- timeout，表示可以容忍的等待响应时间，默认是 30s，如果超时则请求报错。
- master_timeout，表示可以容忍的连接 es 主节点的时间，默认是 30s，如果超时则请求报错。

注意事项及技巧

目前新老版本的索引模板在 7.10 版本都可以使用，并且都是通过名称匹配的方式，来决定最后使用的配置，那么在使用的过程中难免会遇到，匹配到多个模板且配置有冲突的情况。下面总结几点规则来介绍，创建索引的最终配置是如何决定的。

- 如果同时匹配到新老 2 个版本的索引模板，那么使用新版模板。
- 如果仅匹配到多个新版模板，那么使用 priority 值最高的索引模板。

- 如果新版模板中配置了多个组件模板，且组件中有配置冲突，那么使用 composed_of 数组中靠后的组件模板的配置。
- 如果组件模板和索引模板有字段冲突，那么使用索引模板中的配置。
- 如果仅匹配到多个老版模板，那么最终配置由多个模板共同构成，如果有配置冲突，使用 order 值高的模板的配置。
- 如果创建索引语句中的配置与索引模板（不管新老版本）冲突，那么使用创建语句中的配置。

索引模板一般和动态映射结合使用，这样可以大大减少创建索引的语句，缩减索引创建频次。配置方式是在 mappings 字段中配置 dynamic_templates 的相关内容。

这个功能一般用于创建时序类数据的索引，比如日志数据，每天都会有新数据进入索引，数据量会持续增加，只用一个索引肯定不合适，需要按日或按月创建。

这种情况约定录入数据的索引名称与日期相关，再创建索引模板，这样数据持续录入时，索引也会按需增加，且不用人工干预，后期对不同索引还能做冷热处理。

创作人简介：

骆潇龙，高级 Java 开发工程师，关注大数据技术领域。

博客：https://blog.gaiaproject.club/

3.4.2.9 Search template

创作人：骆潇龙

Elasticsearch 允许使用模板语言 mustache 来预设搜索逻辑，在实际搜索时，通过参数中的键值，对来替换模板中的占位符，最终完成搜索。该方式将搜索逻辑封闭在 Elasticsearch 中，可以使下游服务，在不知道具体搜索逻辑的情况下完成数据检索。我们以 Kibana 自带的航班数据 kibana_sample_data_flights 为基础，以按航班号搜索为例，简单介绍搜索模板的使用。

第一步，创建 ID 为 testSearchTemplate 的搜索模板，语句如下：

```
POST _scripts/testSearchTemplate
{
  "script": {
    "lang": "mustache",   #使用 mustache 模板语言
    "source": {    # 脚本内容
      "query": {     # 搜索逻辑
        "term": {
          "FlightNum": {
            "value": "{{FlightNum}}"   # 占位符 FlightNum
          }
        }
      }
    }
  }
}
```

第二步，传参搜索数据，语句如下：

```
GET kibana_sample_data_flights/_search/template
{
  "id": "testSearchTemplate",       # 使用的模板 ID
  "params": {
    "FlightNum": "9HY9SWR"      # 占位符替换的值
  }
}
```

以上两步就是使用模板搜索数据，该逻辑等同于下面这个搜索：

```
GET kibana_sample_data_flights/_search
{
  "query": {
    "term": {
      "FlightNum": {
        "value": "9HY9SWR"
      }
    }
  }
}
```

API 介绍

下面我们从搜索模板的生命周期：创建、查看、使用、删除来展开介绍模板搜索相关 API。

准备

在正式介绍之前，我们先来说一说关于模板搜索的几个预备知识。

首先，如果使用的 Elasticsearch 集群开启了安全功能，那么角色对操作的索引必须要有 read 权限。

其次，搜索模板使用的语法是 Mustache

更多的关于该种脚本语言的介绍以及功能请查看其官方文档:https://mustache.github.io/mustache.5.html

最后，模板搜索属于 Elasticsearch 中 Script 功能的扩展， Script 的限定及用法基本都适用于模板搜索。比如，集群关于 Script 的配置也会影响模板搜索，配置项 script.allowed_types 可规范模板搜索接受的类型（ inline / stored / both ），script.allowed_contexts 也会限制模板搜索可进行的操作。

创建

搜索模板的创建与 Elasticsearch 其它脚本的创建一样，都是发送 1 个 POST 请求即可。

如下所示：

```
POST _scripts/<templateId>  # 1
{
  "script": {
    "lang": "mustache", # 2
    "source": {   # 3
      "query": {
          "term": {
            "FlightNum": {
              "value": "{{FlightNum}}"  # 可变参数 FlightNum
            }
          }
        }
    }
  }
}
```

- 向_scripts/<templateId>发送 POST 请求来创建搜索模板，其中<templateId>是你为该模板设置的 ID，搜索时会用到该 ID
- lang 参数配置的是搜索模板使用的脚本语言为 mustache
- source 参数配置的是搜索模板的具体内容，该部分的格式参照 Elasticsearch 搜索的请求 body，需要搜索时填充的值使用 mustache 语法，配置占位符即可，比如本例中的占位符就是{{FlightNum}}

查看

当我们想查看之前创建的模板内容，或者验证某个 ID 的模板是否存在时，可以向_scripts/<templateId>发送 GET 请求来获取模板的具体内容。

示例如下：

```
GET _scripts/<templateId>  # 1
{
  "_id" : "testSearchTemplate",  # 2
  "found" : true,  # 3
  "script" : {    # 4
    "lang" : "mustache",
    "source" : """{"query":{"term":{"FlightNum":{"value":"{{FlightNum}}"}}}}""",
    "options" : {  # 5
      "content_type" : "application/json; charset=UTF-8"
    }
  }
}
```

- 请求的 path 为_scripts/<templateId>，其中<templateId>为你要查询的模板 Id，请求类型为 GET
- 返回的 body 中，_id 属性再次表明此次查询的模板 ID，本示例查询的是之前创建的 testSearchTemplate 模板
- found 属性表明此次查询是否查到结果，如果模板 ID 存在则此值为 true，反之为 false
- script 就是该搜索模板的具体内容与保存时相同。核心有 lang 属性表示脚本语法，source 属性存放脚本具体内容

- script 属性中的 Options 属性是非必要其它脚本属性，默认会有 content_type 属性，该属性保存查询时 http 请求的 content-type ，默认为 application/json; charset=UTF-8

删除

在一个搜索模板完成了它的使命后，我们需要及时删除它，因为 Elasticsearch 默认缓存脚本的数据量是有上限的，删除的方式很简单，发送一个 DELETE 请求即可。

示例如下：

```
DELETE _scripts/<templateId>    #1
```

<templateId>为要删除的搜索模板的 ID，比如_scripts/testSearchTemplate 表示的就是删除 ID 为 testSearchTemplate 的搜索模板。

使用

搜索模板的使用就是在搜索时，直接发送占位符的值，即可执行预设搜索语句。由于还是在搜索的范畴，所以发送请求的 path 是_search/template。

下面是关于使用搜索模板进行查询的示例：

```
GET <index>/_search/template?<query_parameters> #1
{
  "source": """{"query": {"term": {"FlightNum": {"value": "{{FlightNum}}"}}}}""",  #2
  "id": "testSearchTemplate", # 3
  "params": {  # 4
    "FlightNum": "9HY9SWR"
  },
  "profile": true, # 5
  "explain": true  # 6
}
```

模板搜索发送的地址为<index>/_search/template，与搜索一样<index>处为选填参数，你可以指定搜索的索引，不指定则表示搜索全部索引。

因为本质上还是属于搜索的范畴，所以一些搜索参数在模板搜索是也可以使用，比如：

- scroll（可选，时长）：表示本搜索需要支持游标搜索，游标过期时间为配置值
- ccs_minimize_roundtrips(可选，布尔值)：如果为 true 则在跨集群搜索时最小化集群间交互。默认为 true
- expand_wildcards（可选，字符串）：表示索引通配符作用的范围，可配置为全部（all）、打开索引（open）、关闭索引（closed）、隐藏索引（hidden，需要与 open 或 closed 结合使用）、不允许通配符（none）
- explain（可选，布尔值）：表示返回结果是否带计算得分的详细信息，默认是 false

- ignore_throttled（可选，布尔值）：如果为 true 则表示查询忽略被限制的索引，被限制的索引一般指被冻结（freeze）的索引，该值默认是 true
- ignore_unavailable（可选，布尔值）：如果为 true 则表示关闭的索引不在搜索范围内，默认值为 true
- preference（可选，字符串）：指定执行该操作的节点或分片，默认是随机的
- rest_total_hits_as_int（可选，布尔值）：如果为 true 则 hits.total 将会是个数值而非一个对象，默认为 false
- routing(可选, 字符串)：配置搜索执行的路由
- search_type（可选，字符串）：这是搜索的类型，可选值有：query_then_fetch、dfs_query_then_fetch
- source 字段：用于配置搜索模板，该字段与 ID 字段冲突只能二选一，使用 source 表示不使用保存的模板而使用本模板
- id 字段：表示本次查询使用的搜索模板 ID，该字段与 source 字 段冲突只能二选一
- params 字段：配置的 key-value 值将替换模板中的占位符执行搜索
- profile 字段：是可选字段，表示返回结果中是否有 Elasticsearch 执行搜索的一些元信息
- explain 字段：是可选字段，与 http 中搜索参数配置的 explain 含义一样，表示结果是否带计算得分的详细信息

上述搜索返回结果如下：

```
{
  "took" : 1,
  "timed_out" : false,
```

```
  "_shards" : {
    "total" : 1,
    "successful" : 1,
    "skipped" : 0,
    "failed" : 0
  },
  "hits" : {
    "total" : {
      "value" : 1,
      "relation" : "eq"
    },
    "max_score" : 9.071844,
    "hits" : [
      {
        "_shard" : "[kibana_sample_data_flights][0]",
        "_node" : "ydZx8i8HQBe69T4vbYm30g",
        "_index" : "kibana_sample_data_flights",
        "_type" : "_doc",
        "_id" : "KPRFDHkB9LctWlE3WLqj",
        "_score" : 9.071844,
        "_source" : {
          "FlightNum" : "9HY9SWR",
          "DestCountry" : "AU",
          "OriginWeather" : "Sunny"
        },
        "_explanation" : {}   # 计算得分的逻辑
      }
    ]
  },
  "profile" : {} # 搜索细节信息
}
```

其它

本部分将介绍关于模板搜索的一些小技巧。通常情况下我们写的搜索模板，往往是很难一次就配置正确的，因此需要频繁的测试我们写的模板，与参数结合后是否是我们预期的搜索语句，这时我们就可以使用以下这个请求，来校验模板使用是否正确。

```
GET _render/template # 1
{
  "source": """{"query": {"term": {"FlightNum": {"value": "{{FlightNum}}"}}}}""" ,# 2
  "params": { # 3
    "FlightNum": "9HY9SWR"
  }
}

{  # 4
  "template_output" : {
    "query" : {
      "term" : {
        "FlightNum" : {
          "value" : "9HY9SWR"
        }
      }
    }
  }
}
```

- 向_render/template 发送 GET 请求来验证模板是否正确
- source 字段为要验证的搜索模板，该字段可以省略，如果省略需要在 path 处指定模板 iID，比如_render/template/testSearchTemplate
- params 字段为模板使用的参数
- 此 JSON 就是该请求的返回，template_output 字段就是在使用此 params 下搜索模板生成的查询语句

模板语言 mustache 有许多功能，这里再介绍几个比较常见的。

比如我们使用占位符替换的不是一个字符串，而是一个对象或数组对象，那么我们可以用{{#toJson}}{{/toJson}}来实现，具体如下：

```
GET _render/template
{
  "source": """{"query": {"term": {"FlightNum": {{#toJson}}FlightNum{{/toJson}}  }}}""", # 1
  "params": { # 2
    "FlightNum": {
      "value":"9HY9SWR"
    }
  }
}

{ # 3
  "template_output" : {
    "query" : {
      "term" : {
```

```
            "FlightNum" : {
              "value" : "9HY9SWR"
            }
          }
        }
      }
    }
```

在配置模板时，我们将 FlightNum 的 value 配置为{{#toJson}}FlightNum{{/toJson}}，即表示占位符 FlightNum 是一个对象

在配置 params 时，我们将 FlightNum 的值设置为一个 JSON 对象{ "value":"9HY9SWR"}

通过校验请求的返回，可以看到{{#toJson}}FlightNum{{/toJson}}被替换为对象{ "value":"9HY9SWR"}

Mustache 还能在将变量套入模板时做一些处理，比如将数组变量组合成字符串放入模板、设置占位符的默认值，以及对 URL 转码。

示例如下：

```
GET _render/template
{
  "source": {
    "query": {
```

```
        "term": {
          "FlightNum": "{{#join delimiter='||'}}FlightNums{{/join delimiter='||'}}", #1
          "DestCountry":"{{DestCountry}}{{^DestCountry}}AU{{/DestCountry}}",#2
          "Dest": "{{#url}}{{Dest}}{{/url}}"#3
        }
      }
    },
    "params": {
      "FlightNums": [
        "9HY9SWR",
        "adf2c1"
      ],
      "Dest":"http://www.baidu.com"
    }
}

{
  "template_output" : {
    "query" : {
      "term" : {
        "FlightNum" : "9HY9SWR||adf2c1", # 4
        "DestCountry" : "AU", #5
        "Dest" : "http%3A%2F%2Fwww.baidu.com" # 6
      }
    }
  }
}
```

第一个模板使用{{#join delimiter='||'}}{{/join delimiter='||'}}设置了数组合并的分割字符为 "||"，传参时 FlightNums 配置的为["9HY9SWR","adf2c1"]，而生成的则是 #4 处的 9HY9SWR||adf2c1

第二个模板使用{{^DestCountry}}AU{{/DestCountry}}设置了占位符 DestCountry 的默认值为 AU，这样我们在 params 中并未配置 DestCountry 的值，但生成的 #5 处自动用 AU 替换了占位符

第三个模板我们用{{#url}}{{/url}}声明了此处是一个 URL，需要进行转义，则在 #6 处配置的 http://www.baidu.com 变为了 http%3A%2F%2Fwww.baidu.com

3.4.2.10 Dynamic Mapping

创作人：骆潇龙

通常来说，搜索数据一般需要经过 3 个步骤：

- 定义数据（建表建索引）
- 录入数据
- 搜索数据

在现实使用中，定义数据往往是比较繁琐，并且有大量的重复操作。

Elasticsearch 本着让用户使用更方便快捷的原则，针对这个问题做了很多工作，使定义数据的方式更加抽象灵活，多个雷同的字段可使用 1 个配置完成。

比较有代表性的 2 个功能分别是：

- 索引模板（index template）：可以根据规则自动创建索引。
- 动态映射（dynamic mapping）：自动将新字段添加到映射中。

本小节我们着重介绍动态映射（dynamic mapping）

根据官方的定义动态映射可以自动检测和添加新字段（field）到映射（mapping）中。动态映射可以通过基础属性自动发现（Dynamic field mappings）以及复杂属性动

态生成（Dynamic templates）2 个方式实现此功能。

动态字段映射（Dynamic field mappings）

在默认情况下，当索引一个文档时有字段是在映射中没有配置的，那么 Elasticsearch 将会根据该属性的类型，自动将其增加到映射中。该功能可以通过配置 dynamic 来控制打开。

该配置可以接受以下 3 种选择：

- ture：默认配置，新字段将会自动加入映射中，并自动推断字段的类型。
- false：新字段不会增加到映射中，因此不能被搜索，但是内容依然会保存在 _source 中。如无特殊需要建议都配置为 false，这样可以避免写入流程经过 master 节点，从而提高性能。
- strict：索引文档时如果发现有新字段则报错，整个文档都不会被索引。

该配置可以在创建 mapping 时在根层配置，表示对所有属性适用。也可以每个内嵌对象（inner object）中配置，表示仅对该对象适用。

示例如下:

```
# 创建 test-dynamic-mapping
PUT test-dynamic-mapping
{
  "mappings": {
```

```
        "dynamic": false, # 1
        "properties": {
          "person":{
            "dynamic": true, # 2
            "properties": {
              "name":{
                "type":"keyword"
              }
            }
          },
          "company":{
            "dynamic": "strict",   # 3
            "properties": {
              "company_id":{
                "type":"keyword"
              }
            }
          }
        }
      }
    }
```

- 在 #1 处的配置索引 test-dynamic-mapping 整体是不自动增加字段的
- 在 #2 处对于内嵌对象 person 我们设置它可以自动发现字段
- 在 #3 处对于内嵌对象 company 我们设置它发现新字段会报错

```
# 插入文档
PUT test-dynamic-mapping/_doc/1
{
  "school":"test school", # 1
  "person":{
    "name":"tom",
    "age":"12" # 2
  },
  "company":{
    "company_id":"c001"
  }
}
```

- 传入文档的根层有个未定义的 school 字段
- 在 person 对象中增加 age 字段

```
# 再次查看索引 mapping
GET test-dynamic-mapping
{
  "test-dynamic-mapping" : {
    "mappings" : {
      "dynamic" : "false",
      "properties" : { # 1
        "company" : {
          "dynamic" : "strict",
          "properties" : {
            "company_id" : {
              "type" : "keyword"
```

```
            }
          }
        },
        "person" : {
          "dynamic" : "true",
          "properties" : {
            "name" : {
              "type" : "keyword"
            },
            "age" : { # 2
              "type" : "text",
              "fields" : {
                "keyword" : {
                  "type" : "keyword",
                  "ignore_above" : 256
                }
              }
            }
          }
        }
      }
      ..................
    }
}
```

- 1 处由于我们对整个 mapping 设置了 dynamic:false，所以 school 属性没有自动创建。

- 由于内嵌对象 person 的 dynamic:true，因此自动增加了 sex 属性，该属性派生出 2 个字段索引 person.age 其字段类型是 text 以及 person.age.keyword 其字段类型是 keyword。

```
# 再次查看索引 mapping
GET test-dynamic-mapping
{
  "test-dynamic-mapping" : {
    "mappings" : {
      "dynamic" : "false",
      "properties" : { # 1
        "company" : {
          "dynamic" : "strict",
          "properties" : {
            "company_id" : {
              "type" : "keyword"
            }
          }
        },
        "person" : {
          "dynamic" : "true",
          "properties" : {
            "name" : {
              "type" : "keyword"
            },
            "age" : { # 2
              "type" : "text",
              "fields" : {
```

```
                "keyword" : {
                  "type" : "keyword",
                  "ignore_above" : 256
                }
              }
            }
          }
        }
      }
      ...................
    }
  }
```

- 索引新文档时增加 company.company_name 字段。
- 由于 company 对象 dynamic:strict，所以创建文档的请求返回了 1 个 strict_dynamic_mapping_exception 错误。

对于 JSON 中的字段 遵循以下映射方式发现新属性。

JSON data type	Elasticsearch data type
null	不添加
true 或 false	boolean 类型
带小数的数字，如 1.1	float 类型
整数，如 3	long 类型

JSON data type	Elasticsearch data type
数组	ES 不特殊处理数组类型
字符串	如果配置了自动识别且通过则可被识别为 date、float、long 类型如果未配置则会识别为 text 类型且增加 keyword 子属性使用 keyword 类型

对于 JSON 中的字符串字段，我们可以通过配置 date_detection: true 和 numeric_detection: true 尝试将它们转化成数值类型或时间类型，date_detection 默认为 true，numeric_detection 默认为 false。

在识别数字时，所有整型字符串会识别成 long 型，带小数的字符串会识别成 float 类型。默认情况下 yyyy/MM/dd HH:mm:ss、yyyy/MM/dd、epoch_millis 格式的字符串会识别成 date 类型。

```
# 创建测试索引
PUT test-dynamic-mapping
{
  "mappings": {
    "dynamic": true,
    "numeric_detection": true, # 1
    "properties": {
      "field1":{
        "type": "keyword"
      }
    }
```

```
  }
}
# 插入数据
PUT test-dynamic-mapping/_doc/1
{
  "date":"2021/05/01", # 2
  "float":"1.1", # 3
  "long":"1" # 4
}
# 查看 mapping 变化
GET test-dynamic-mapping
{
  "test-dynamic-mapping" : {
    "mappings" : {
      "dynamic" : "true",
      "numeric_detection" : true,
      "properties" : {
        "date" : {    # 5
          "type" : "date",
          "format" : "yyyy/MM/dd HH:mm:ss||yyyy/MM/dd||epoch_millis"
        },
        "field1" : {
          "type" : "keyword"
        },
        "float" : {   #6
          "type" : "float"
        },
        "long" : {   # 7
          "type" : "long"
```

```
            }
          }
        }
      }
    }
```

- 在 #1 处在创建索引时设置字符串可以自动识别为数值类型
- 在 #2 处 date 字段条是符合时间格式的字符串
- 在 #3 处 float 字段是符合小数格式的字符串
- 在 #4 处 long 字段是符合整型格式的字符串
- 在 #5 处 date 字段加入 mapping 并被自动识别成了 date 类型
- 在 #6 处 float 字段加入 mapping 并被自动识别成了 float 类型
- 在 #7 处 long 字段加入 mapping 并被自动识别成了 long 类型

Elasticsearch 识别日期字符串的格式，是可以通过 dynamic_date_formats 来配置。该字段支持使用 y、m、d、h 等字符自定义格式，具体方式与 Java 中 DateTimeFormatter 对象实现的规则相同。

同时还能配置大量国际标准的时间格式比如：epoch_millis、basic_date、basic_date_time、strict_date_optional_time_nanos 等，

所有可选项可以参照官方文档:https://www.elastic.co/guide/en/elasticsearch/reference/7.10/mapping-date-format.html

```
# 创建测试索引
PUT test-dynamic-mapping
{
  "mappings": {
    "dynamic": true,
    "dynamic_date_formats": ["MM/dd/yyyy"]  # 识别 MM/dd/yyyy 格式的时间
    "properties": {
      "field1":{
        "type": "keyword"
      }
    }
  }
}
# 插入数据
PUT test-dynamic-mapping/_doc/1
{
  "date":"09/25/2015",
  "date1":"2015/09/25"
}
# 查看 mapping 变化
{
  "test-dynamic-mapping" : {
    "mappings" : {
      "dynamic" : "true",
      "dynamic_date_formats" : [
        "MM/dd/yyyy"
      ],
      "properties" : {
```

```
            "date" : {              # 符合 MM/dd/yyyy 格式的字符串识别为了 date 类型
              "type" : "date",
              "format" : "MM/dd/yyyy"
            },
            "date1" : {             # 符合默认 yyyy/MM/dd 格式的字符串识未正确识别
              "type" : "text",
              "fields" : {
                "keyword" : {
                  "type" : "keyword",
                  "ignore_above" : 256
                }
              }
            },
            "field1" : {
              "type" : "keyword"
            }
          }
        }
      }
    }
```

动态模板（Dynamic templates）

Elasticsearch 的动态字段映射（Dynamic field mappings）虽然使用简单，但往往不满足现实的业务场景，比如对于整型字段，往往用不着 long 类型，使用 integer 类型就足够了；对于字符串类型的字段，我们希望细化分词方式，而不是使用默认分词，以及对于不同字段采用不同的分词方式等。这时可以使用动态模板（Dynamic templates）功能来实现上述需求。

动态模板允许你在创建 mapping 时，设置自定义规则。当新字段满足该规则时，则按照预先的配置来创建字段。

Elasticsearch 允许用户通过 3 个角度来定义规则：新字段的数据类型，属性名和路径。创建 mapping 时可以通过 dynamic_templates 字段配置多个动态模板。

模板的整体结构如下：

```
{
  "mappings":{
    "dynamic_templates": [
      {
        "templateName":{    #1
          ……匹配规则……          # 2
          "mapping": { ... } #3
        }
      }
    ]
  }
}
```

- 在 #1 处定义了动态模板的名称，每个动态模板都需要配置名字，本例中配置的模板名称为 templateName
- 在 #2 处可以使用 match_mapping_type、match 、unmatch 、match_pattern、path_match、path_unmatch 来配置该模板的匹配规则，规则可以是多个，规则之间是与的关系

- 在 #3 处配置的是符合该规则的字段使用的 mapping 配置，此处与正常创建字段相同，主要需要配置 type，analyzer 等

匹配规则

下面我们通过几个例子来说明一下匹配规则中的各个关键字如何使用。

match_mapping_type

match_mapping_type 用于按照数据类型匹配，当用户想对 JSON 中具有某种数据类型的字段设置做特殊配置时，可以用此种匹配方式。该字段可配置的数据类型有如下几种：

- boolean，匹配值是 true 或 false 的字段。
- date，当字符串开启了时间类型识别且字符串符合预设日期格式则会被匹配
- double，匹配含有小数的字段
- long，匹配值是整型的字段
- object，匹配值是对象的字段
- string，匹配值是字符串的字段
- *，表示所有数据类型即匹配所有字段

之前我们提到 Elasticsearch 会自动将整型字段自动创建为 long 型，如果我们知道文档中所有数值都不会超过 int 范围，那么我们可以用如下配置，让所以非小数的数值字段自动创建为 integer 类型。

```
PUT test-dynamic-mapping
{
  "mappings": {
    "dynamic_templates": [
      {
        "test_float": {
          "match_mapping_type": "long", # 值是整型的字段会被匹配
          "mapping": {
            "type": "integer"  # 字段 type 统一设为 integer
          }
        }
      }
    ]
  }
}
```

match 、unmatch

在生产使用中最多的场景，是根据字段的名称进行匹配。这时就可以用 match 和 unmatch 这两种匹配方式。match 匹配的是符合设置的所有字段，unmatch 匹配的是不符合某种配置的所有字段。在设置匹配规则时可以使用*表 0 个或多个字符。

比如下面这个模板就表示所有属性名以 long_开头且不以_text 结尾的字段配置其 type 为 long。

```
PUT test-dynamic-mapping
{
  "mappings": {
    "dynamic_templates": [
      {
        "test_float": {
          "match": "long_*", # 属性名以 long_ 开头
          "unmatch": "*_test", # 属性名不以 _test 结尾
          "mapping": {
            "type": "long"  # 字段 type 设为 long
          }
        }
      }
    ]
  }
}
```

match_pattern

仅仅使用通配符，可能不能满足我们多变的匹配需求，那么我们可以将 match_pattern 设为 regex，这时 match 字段就可以用正则表达式了。

比如下面这个模板就表示所有以 profit_开头，后跟至少 1 位数字的属性，将它们的 type 设为 keyword。

```
PUT test-dynamic-mapping
{
  "mappings": {
    "dynamic_templates": [
      {
        "test_float": {
          "match_pattern": "regex",  # match 使用正则表达式
          "match": "^profit_\d+$"  # 标准正则
          "mapping": {
            "type": "keyword"  # 字段 type 设为 keyword
          }
        }
      }
    ]
  }
}
```

path_match 、 path_unmatch

在 Elasticsearch 中存储的文档允许有内嵌对象，当还有多层内嵌对象时，属性一般有路径的概念。属性的路径也可以作为匹配的条件。这个配置的用法与 match 和 unmatch 雷同，但需要注意的是 match 和 unmatch 仅作用于最后一级的属性名。

如下模板表示设置 person 内嵌对象除了 age 外其它所有以 long_开头的字段新增时类型设为 text。

```
PUT test-dynamic-mapping
{
  "mappings": {
    "dynamic_templates": [
      {
        "test_float": {
          "match_pattern": "long_",  # 以 long_ 开头
          "path_match": "person.*",  # 内嵌对象 person 所有字段
          "path_unmatch": "*.age"    # 排除 age 字段
          "mapping": {
            "type": "text"  # 字段 type 设为 text
          }
        }
      }
    ]
  }
}
```

其它技巧及注意事项

在日常生产中难免有这样的需求，字段是什么类型就将类型设为什么，字段名是什么就用什么解析器。对于这种需求我们在配置动态模板的 mapping 时，可以使用占位符 {name} 表示字段名，用 {dynamic_type}表示识别出的字段类型 。

比如下面 2 个模板一起表示的意思是，所有新增字符串类型字段，其解析器是字段的名称，所有其他类型字段新增时，类型就设为识别的字段类型，但是 doc_value 设为 false.

```
PUT test-dynamic-mapping
{
  "mappings": {
    "dynamic_templates": [
      {
        "named_analyzers": {  # 字段名即是该字段的解析器名称
          "match_mapping_type": "string", # 匹配所有 string 类型
          "match": "*",  # 匹配任意属性名
          "mapping": {
            "type": "text",
            "analyzer": "{name}"  # 解析器是字段名
          }
        }
      },
      {
        "no_doc_values": {
         # 匹配所有类型，但匹配 string 的在前,所以
实际匹配除 string 的其他所有字段
          "match_mapping_type":"*",
          "mapping": {
            "type": "{dynamic_type}", # 类型直接作为 type
            "doc_values": false

          }
        }
      }
    ]
  }
```

```
    }

    PUT test-dynamic-mapping/_doc/1
    {
      "english": "Some English text",  # 该字段是新字段，会在 mapping 中新增会用 english 解析
器
      "count":   5  # 该字段的类型会是 long
    , doc_values 为 false
    }
```

在使用动态模板时，还有以下几点需要注意。

- 所有 null 值及空数组属于无效值，不会被任何动态模板匹配，在索引文档时，只有字段第一次有有效值时，才会与各动态模板匹配，找出匹配的模板创建新字段。
- 规则匹配时，按照动态模板配置的顺序依次对比，使用最先匹配成功的模板，这就意味着如果有字段同时符合 2 个动态模板，那么会使用在 dynamic_templates 数组中靠前的那个。每个动态模板的匹配方式至少应包含 match、path_match 、match_mapping_type 中的一个，unmatch 和 path_unmatch 不能单独使用。
- mapping 的 dynamic_templates 字段是可以在运行时修改的，每次修改会整体替换 dynamic_templates 的所有值而非追加。

比如下面的请求就是将映射 test-dynamic-mapping 原来的动态模板配置删除，并配一个名为 newTemplate 的动态模板。

```
PUT test-dynamic-mapping/_mapping
{
  "dynamic_templates": [
    {
      "newTemplate": {
        "match": "abc*",
        "mapping": {
          "type": "keyword"
        }
      }
    }
  ]
```

3.4.2.11 Index alias

创作人：杨松柏

别名的优势

索引别名是一个非常好的”工具“，他可以帮助解决以下问题：

- 如果对写入 Elasticsearch 的数据进行极少的修改，索引别名+ Rollover 可以很好控制每个索引的大小，零停机切换索引；合适的索引大小可以提升数据的查询性能，数据恢复性能。
- 解耦 client 与索引的强耦合，Elasticsearch 维护人员可以对索引有更灵活的操作空间，且让用户侧无感知。
- 结合 Reindex 可以很方便的完成索引重建。
- 过滤别名和路由别名可以在一定程度上帮助提升查询性能。
- 一个绑定多个索引的别名，如果要查询一次查询这多个索引，别名可以使 uri 变的简洁。

什么是别名

别名，是为一个或多个索引而命名的第二名称，第二名称不得与集群中任何索引同名；只要把第二名称和真实索引建立绑定关系，便可以使用别名对索引进行相关的操作。

别名管理

别名创建

索引别名的 REST 语法如下：

```
#索引别名
PUT /<index>/_alias/<alias>?master_timeout=<time>&timeout=<time>
#过滤别名，路由别名
PUT /<index>/_alias/<alias>?master_timeout=<time>&timeout=<time>
{
  "routing" : "routing_value",
  "filter" : {
    "term" : {
      "filed" : value
    }
  }
}
#以下三种方式同上
POST /<index>/_alias/<alias>
PUT /<index>/_aliases/<alias>
POST /<index>/_aliases/<alias>
```

URI 参数释意

必填参数，参数类型 string；该参数可以由逗号分隔的索引，或者用通配符表达式。值也可为_all,表示作用于集群中的所有索引。

必填参数，参数类型 string，索引别名,建议名字使用有意义的单词和数字组成。

master_timeout

可选参数，value 值的单位可为 d、h、m、s、ms、micros、nanos； 等待连接到主节点的时间。如果在超时时间阈值之前没有收到响应，则请求失败并返回错误,默认值为 30s。

timeout

可选参数，value 值的单位可为 d、h、m、s、ms、micros、nanos；请求等待响应的时间。如果在超时时间阈值之前没有收到响应，则请求失败并返回错误,默认值为 30s。

请求体

创建一个索引别名的时候，通常路由别名和过滤别名需要指定请求体。

filter

必填参数，将过滤参数绑定到别名，使别名具有特定的查询功能；包含此参数的别名，通常将其称作为过滤别名。

routing

可选 string 类型参数，自定义路由值用于将操作路由到特定分片的；包含此参数的我们通常将其称作为路由别名。

批量创建别名

批量创建别名 REST 语法如下：

```
POST /_aliases?master_timeout=<time>&timeout=<time>
{
  "actions" : [
    { "<action>" : { "alias" : "index-alias", "<must_param>" : "value", "<option_param>" : "value",... } }
  ]
}
```

请求体参数释意

actions

必填参数，数组内包含一系列的动作<action>，支持的动作如下

add

为一个索引或多个索引添加一个别名

remove

将别名移除与索引的关联关系

remove_index

删除索引，等效于 delete index API。该动作只对索引别名生效，如果尝试删除索引别名，将会失败。

JSON 体内支持的参数包括必填参数和可选参数。

详情如下：

必填参数：

index

参数类型 string,支持通配符。如果 indices 没有指定，则该参数必须指定。

注意：不能向索引别名添加数据流。

indices

参数类型 string 数组,该数组内的索引将被执行相应的动作。如果 index 没有指定，则该参数必须指定。

注意事项与 index 参数相同。

alias

参数类型 string,为以逗号分隔或者通配符表示的索引， add,remove,delete 别名。如果 aliases 没有指定，则该参数必须指定。

aliases

参数类型 string array,需要进行 add,remove,delete 的索引别名组。如果 alias 没有指定，则该参数必须指定。

可选参数 ：

filter

query object 查询对象体，绑定了过滤查询的别名。如果指定，使用别名进行空查询，将只返回满足过滤条件的文档。

is_hidden

参数类型 bool 值，默认值为 false;如果设置为 true,使用通配符表达式别名进行搜索排除，该别名关联的数据将查询不到;除非在请求中使用 expand_wildcards 参数重写。对于共享同一个别名的所有索引，必须将此属性设置为相同的值。

must_exist

参数类型 bool 值，默认值为 false，如果设置为 ture，移除别名时，该别名必须存在。

is_write_index

参数类型 bool 值，默认值为 false;如果设置为 true，则可以直接使用该别名对关联的索引进行数据写入或者配置修改等操作；若别名绑定多个索引，则只能存在一个 is_write_index 值为 true 的绑定。注意：在同一个索引 is_hidden 和 is_write_index 不能同时设置为 true；当别名只绑定一个索引时无需现实设定该值为 true,该别名具有写权限，但是当别名再次绑定另外一个索引，则别名的写权限取消，除非现实指定 is_write_index 的值为 true

routing

参数类型 string，自定义值作为路由计算值，将操作路由到对应的分片上。

index_routing

参数类型 string，自定义值作为路由计算值，将写入操作路由到对应的分片上。

search_routing

参数类型 string，自定义值作为路由计算值，将查询操作路由到对应的分片上。

别名创建与修改示例

假设已经存在表 user1 和 user2,为他们绑定别名，示例子如下：

```
#为 users1 表创建 index-alias-name1 别名
PUT /users1/_alias/index-alias-name1

#为 users1,users2 表创建 index-alias-name2 别名
PUT /users1,users2/_alias/index-alias-name2

#为 users 开头的索引创建 index-alias-name3 别名
PUT /users*/_alias/index-alias-name3

#为 users1 索引添加具有路由和过滤功能的别名 routing-filter-index-alias
PUT users1/_alias/routing-filter-index-alias
{
  "routing" : "12",
  "filter" : {
```

```
      "term" : {
        "user_id" : 12
      }
    }
}
#为 users1 索引添加路由别名 routing-index-alias，路由计算值为 12
PUT users1/_alias/routing-index-alias
{
  "routing" : "12"
}

#为 users1 索引添加过滤别名 filter-index-alias，过滤 user_id 为 12
PUT users1/_alias/filter-index-alias
{
  "filter" : {
    "term" : {
      "user_id" : 12
    }
  }
}
```

别名创建成功之后，如果别名与索引的关系，为一个别名只对应一个索引，或者有一个绑定关系的 is_write_index (后文会介绍)值为 true ;那么我们可以通过别名往索引写如数据。

```
#通过别名，写入数据
PUT index-alias-name1/_bulk?refresh=true
{"index":{}}
```

```
{"user_id":"tom123456-user1"}

#通过路由别名，写入数据
PUT routing-index-alias/_bulk?refresh=true
{"index":{}}
{"user_id":"kimchy123456-routing"}

#通过索引，写入数据
PUT users2/_bulk?refresh=true
{"index":{}}
{"user_id":"kimchy123456-user2"}
{"index":{}}
{"user_id":"12"}
```

插入数据后可以通过别名或者索引查询

```
#以下三条查询语句的结果是等价的，返回 2 条数据
GET index-alias-name1/_search
GET users1/_search
GET routing-index-alias/_search

#以下三条查询语句的结果是等价的(如果没有其他 users 开头的索引)，返回四条数据
GET users1,users2/_search
GET users*/_search
GET index-alias-name2/_search

#查询 user2 的索引，返回 2 条数据
GET users2/_search
```

```
#返回值为空，因为索引 users1 插入的文档没有 user_id 值为 2 的
GET filter-index-alias/_search

# 返回值为一条，因为索引 users2 有一条 user_id 值为 2 的文档
GET index-alias-name3/_search
```

批量创建索引别名示例

批量创建索引别名,即使用 `POST /_aliases` 中定义多个 `action`。为 `test1` 和 `test2` 索引绑定一个名称为 `alias1` 的别名。

```
POST /_aliases
{
  "actions" : [
    { "add" : { "index" : "test1", "alias" : "alias1" } }
    { "add" : { "index" : "test2", "alias" : "alias1" } }
  ]
}
```

重命名别名

如果我们需要对一个索引进行别名替换，只需要在同一个 API 中简单的先 `remove` 掉旧别名，然后绑定新的别名即可；该操作为原子型操作，无需担心别名在短时间内不指向索引。

```
POST /_aliases
{
  "actions" : [
    { "remove" : { "index" : "test1", "alias" : "alias1" } },
    { "add" : { "index" : "test1", "alias" : "alias2" } }
  ]
}
```

将一个别名关联多个索引

为 test1 添加别名 alias1，test2 添加别名 alias2，代码块中的两种方式等效。为索引绑定多个别名的语法与之类似，只需将 alias 替换 aliases 数组。

```
POST /_aliases
{
  "actions" : [
    { "add" : { "index" : "test1", "alias" : "alias1" } },
    { "add" : { "index" : "test2", "alias" : "alias1" } }
  ]
}
# 等效于上面的方式
POST /_aliases
{
  "actions" : [
    { "add" : { "indices" : ["test1", "test2"], "alias" : "alias1" } }
  ]
}
```

除了以上两种方式外，我们还可以使用 glob 模式，将别名与多个索引相关联绑定，这种方式只会对集群中已存在的索引生效，不会对之后创建的索引生效。

```
POST /_aliases
{
  "actions" : [
    { "add" : { "index" : "test*", "alias" : "all_test_indices" } }
  ]
}
```

如果错误的创建了一个索引，同样可以通过别名的方式来解决。

例如：错误的创建了一个名称为 test 的索引，而实际需要的索引名称为 test_2，但是已经有数据往索引里面写入数据了；为了解决这个问题，首先创建正确的索引名称，然后用一个原子操作，将 test 别名绑定 test_2，同时删除索引 test。

集群状态中不会发生别名绑定不到索引的情况；但由于索引和搜索涉及多个步骤，正在运行或排队的请求，可能会由于临时不存在索引而失败。

```
# 创建索引 test
PUT test
#  创建索引 test_2
PUT test_2
POST /_aliases
{
  "actions" : [
```

```
        { "add":  { "index": "test_2", "alias": "test" } },
        { "remove_index": { "index": "test" } }
      ]
    }
```

新建索引时绑定别名

上述给索引绑定别名，均需要提前创建索引；若需要别名能给对新索引生效，可以在创建索引时进行指定。

```
PUT test1
{
  "aliases" : {
      "alias1" : { },
      "alias2" : { }
    },
    "mappings" : { },
    "settings" : {}
}
```

除此之外，还可以在索引模板里面进行指定；

如下代码块创建了一个模版名称为 `test`，`order` 权重为 `0` 索引模板，之后只要是以 `test` 开头的索引都会绑定别名 `alias1` 和 `alias2`。

```
PUT _template/test
{
    "order" : 0,
    "index_patterns" : [
        "test*"
     ],
      "aliases" : {
        "alias1" : { },
        "alias2" : { }
     },
      "mappings" : { },
     "settings" : {}
}
```

别名查看

Get index alias

别名的查看方式有如下几种方式

```
GET /_alias
GET /_alias/<alias>
GET /<index>/_alias/<alias>?allow_no_indices=true&expand_wildcards=all&local=false&ignore_unavailable=false
```

路径参数

可选参数，参数类型为 string;参数支持单个索引或者以逗号分隔的多个索引再或者通配符表达式形式

可选参数，参数类型为 string;参数支持单个别名或者以逗号分隔的多个别名再或者通配符表达式形式

查询参数

allow_no_indices

可选参数，参数类型为 Boolean，默认值为 true；如果设置为 false，任何通配符表达式、索引别名或_all 值只针对丢失或关闭的索引，则请求将返回一个错误。即使请求以其他开放索引为目标，此行为也适用。

expand_wildcards

可选参数，参数类型为 string,该参数主要用于控制哪些特性的索引的别名可以被查看，参数可取如下几种类型的值：

- all

  匹配所有数据流或索引，包括隐藏的数据流或索引。默认值为 all。

- open

  匹配索引状态为 open，非 hidden 的索引，以及非 hidden 的数据流。

- closed

匹配索引状态为 closed，非 hidden 的索引，以及非 hidden 的数据流（数据流不能够被关闭）。

- hidden

匹配隐藏的数据流和索引，且索引必须是打开或者关闭状态。

- none

不接受通配符表达式,即参数<index>中不能包含通配符表达式

ignore_unavailable

可选参数，参数类型 Boolean，默认值为 false。如果请求路径中<index>有索引不存在，则请求将会发生错误。

local

可选参数，参数类型 Boolean，默认值为 false。如果设置为 true，则仅仅从本地节点获取集群元信息（包括索引别名信息）；如果设置为 true，则从 master 节点获取，master 的信息最权威，可以避免因为网络等问题，造成的元信息下发不及时，造成的获取元信息有误，但也会增加网络开销。

_cat API

除了通过 Get index alias API 进行查看索引别名，还可以_cat API 进行索引别名查看。

```
GET _cat/aliases
GET _cat/aliases/<alias>
```

别名删除

Delete index alias

RESTful API 语法如下：

```
DELETE /<index>/_alias/<alias>?master_timeout=<time>&timeout=<time>
DELETE /<index>/_aliases/<alias>?master_timeout=<time>&timeout=<time>
```

参数释意

可选参数，参数类型为 string;参数支持单个索引或者以逗号分隔的多个索引再或者通配符表达式形式。_all 和*表示对集群中所有索引。

可选参数，参数类型为 string;参数支持单个别名或者以逗号分隔的多个别名再或者通配符表达式形式。_all 和*表示删除<index>的所有别名。master_timeout 与 timeout

的默认值 30s

bulk 删除

使用 bulk 的方式，同时为一个索引或者一组索引，移除关联的索引别名；假如已经为 test1 添加了别名 alias1，test2 添加了别名 alias2，现在需要进行别名移除，可以执行如下四种操作。

```
# 移除指定索引的指定别名
POST /_aliases
{
  "actions" : [
    { "remove": { "index" : "test1", "alias" : "alias1" } },
    { "remove": { "index" : "test2", "alias" : "alias2" } }
  ]
}
POST /_aliases
{
  "actions" : [
    {  "remove": { "indices" : ["test1","test2"], "aliases" : ["alias1","alias2"]}}
  ]
}
# 移除以 test 开头的索引的指定别名；使用通配符方式，需要注意影响范围
POST /_aliases
{
  "actions" : [
    { "remove": { "index" : "test*", "alias" : "alias1" } },
```

```
      { "remove": { "index" : "test*", "alias" : "alias2" } }
    ]
}

POST /_aliases
{
  "actions" : [
    { "remove": { "index" : "test*", "aliases" : ["alias1","alias2"] } }
  ]
}
```

别名的分类与应用

依据不同的使用场景，我们可以简单把别名分为三类：（1）索引别名（2）过滤别名（3）路由别名

过滤别名

一种创建同一索引的不同”视图“的简便方法。

通过将过滤条件绑定到对应别名，使用不同别名即获取满足不通条件的数据；使用 Query DSL 定义过滤器。

使用过滤别名，必须得保证过滤字段存在，因此提前创建好索引，并设定该字段的 schema。

如下示例，首先创建一个名为 my-index-000001 的索引：

```
PUT /my-index-000001
{
  "mappings": {
    "properties": {
      "user": {
        "properties": {
          "id": {
            "type": "keyword"
          }
        }
      }
    }
  }
}
```

再为索引添加上过滤别名，并批量插入三个文档：

```
#添加过滤别名
POST /_aliases
{
  "actions": [
    {
      "add": {
        "index": "my-index-000001",
        "alias": "alias2",
        "filter": { "term": { "user.id": "kimchy" } }
```

```
        }
      }
    ]
}
#批量插入文档
PUT my-index-000001/_bulk
{"index":{}}
{"user.id":"kimchy"}
{"index":{}}
{"user.id":"tom"}
{"index":{}}
{"user.id":"jerry"}
```

使用别名执行一个空搜索 `GET alias2/_search`，和预期的一致只返回 `user.id` 为 `kimchy` 的文档：

```
# 注意：返回内容中省略了与本节无关的内容
{
  "hits" : {
    "hits" : [
      {
        "_index" : "my-index-000001",
        "_type" : "_doc",
        "_id" : "b_VkIXkB9LctWlE3HOtS",
        "_score" : 1.0,
        "_source" : {
          "user.id" : "kimchy"
        }
```

```
        }
      ]
    }
  }
```

路由别名

将路由字段绑定到对应的别名，通过别名操作时，会执行默认的路由规则，也可以理解为索引的另外一种“视图”。使用路由别名，在一定程度提升写入和查询的性能，结合过滤别名，可以让操作发送到准确的分片上。

如下示例首先创建一个索引，因为本文演示的 Elasticsearch 集群只有两个数据节点，为了方便观察，设置副本数目为 0，主分片数目设置为 2。

```
PUT /routing-index-000001
{
  "settings" :{
    "index":{
      "number_of_shards":2,
      "number_of_replicas": 0
    }
  },
  "mappings": {
    "properties": {
      "user": {
        "properties": {
          "id": {
```

```
            "type": "keyword"
          }
        }
      }
    }
  }
}
```

创建一个路由别名 routing-index-alias1 绑定索引 routing-index-000001：

```
POST /_aliases
{
  "actions": [
    {
      "add": {
        "index": "routing-index-000001",
        "alias": "routing-index-alias1",
        "routing": "2"
      }
    }
  ]
}
```

创建成功之后，使用别名进行的所有操作，都将以 2 计算路由地址，使用别名写入一个文档。

```
PUT routing-index-alias1/_bulk
{"index":{}}
{"user.id":"kimchy"}
```

除了创建一个路由别名，也可以用索引在写入文档时，后面加上路由参数；往 routing-index-000001 索引写入两个文档，并指定路由参数为 12。

```
PUT routing-index-000001/_bulk?routing=12
{"index":{}}
{"user.id":"tom"}
{"index":{}}
{"user.id":"jerry"}
```

通过查看 GET _cat/shards/routing-index-000001?v&h=index,shard,prirep,docs,node，可以发现 routing=12 的文档被写到了 1 号分片，routing=2 的文档被写到了 0 号分片。

```
index                  shard prirep docs node
routing-index-000001 1     p      2 es-cn-n6w24fib900797tgz-29e2dafd-0003
routing-index-000001 0     p      1 es-cn-n6w24fib900797tgz-29e2dafd-0001
```

为了验证结果是不是这样的，我可以执行以下 3 种查询进行验证。前两个查询的是等效的，都是以 2 为计算路由值，将返回 user.id 为 kimchy 一个文档；第三个查询语句将返回以 routing=12 写入的文档。

```
GET /routing-index-alias1/_search
GET /routing-index-000001/_search?routing=2
GET /routing-index-000001/_search?routing=12
```

查询的时候，使用别名查询或者查询参数指定路由值，查询效果是等价的。routing 的值并不一定要等于 2，只要满足 hash 函数计算出来的结果一样，定位到分片结果一致。为了保证能够正确查询到文档，建议 routing 的值和写入的值保持一致。

除了统一设置一个 routing，还可以分别设置对不同动作生效的路由。

如下面分别设置查询路由（search_routing）和写入路由（index_routing），使用别名进行操作，查询请求会发送到路由值 12 和 2 对应的分片，写入操作只会写入路由 12 对应的分片。

```
POST /_aliases
{
  "actions": [
    {
      "add": {
        "index": "routing-index-000001",
        "alias": "routing-index-alias2",
        "search_routing": "12,2",
        "index_routing": "12"
      }
    }
  ]
}
```

如上代码块所示 search_routing 可以由多个以逗号分隔的路由值，index_routing 只能有一个值。

其实这也比较好理解，写入的时候，通过一个路由值计算数据，应该写到具体哪个分片，如果写入有多个路由值，将无法确定写入到哪一个分片；一个给定的分片上，可以有很多拥有不同路由值的文档，因此在查询的时候可以写多个路由值。

使用路由别名查询并且参数重新指定 routing 值；若 search_routing 的和路径参数 routing 的个数大于 2，则取两者的交集作为路由参数；若小于等于 2 且没有交集，则取 search_routing 的值（写入路由同理）。如下代码是查询不到任何文档的，因为其取交集后的路由值为 2，而包含 tom 这个文档写入的路由值为 12，所以将查询到对应的内容。

```
GET /routing-index-alias2/_search?q=user.id:tom&routing=2
```

正如所预期的一样返回内容如下：

```
{
  "took" : 2,
  "timed_out" : false,
  "_shards" : {
    "total" : 1,
    "successful" : 1,
    "skipped" : 0,
    "failed" : 0
  },
```

```
    "hits" : {
      "total" : {
        "value" : 0,
        "relation" : "eq"
      },
      "max_score" : null,
      "hits" : [ ]
    }
  }
```

索引别名（写权限）

当一个索引别名绑定较多的索引，这时若需要使用别名进行写操作，则需要对其中一对绑定关系进行标注，指定别名对特定索引具有写操作权限，没有标注具有写权限的索引别名即为普通索引别名。具有写权限的索引别名，操作索引别名时，会转化为对真实索引的操作。

索引别名（索引为动词）的应用场景主要包括，`reindex` 索引和 `Rollover` 索引。如下所示，将别名 `write-index-alias1` 同时绑定索引 `test` 和 `test2`，别名与 `test` 索引的绑定关系标注了写权限。

```
POST /_aliases
{
  "actions": [
    {
      "add": {
        "index": "test",
        "alias": "write-index-alias1",
```

```
            "is_write_index": true
          }
        },
        {
          "add": {
            "index": "test2",
            "alias": "write-index-alias1"
          }
        }
      ]
}
```

使用索引别名进行数据写入：

```
PUT /write-index-alias1/_doc/1
{
  "foo": "bar"
}
```

通过以下方式可以验证，数据写入到了 test 索引：

```
#可以获取得到对应文档
GET test/_doc/1

#可以查看到两个索引的文档数量，发现 test 表增加了一个文档
GET _cat/indices/write-index-alias1?v
```

在进行索引 Rollover 或者 Reindex 时，为了做到零停机切换索引；还可以通过 Bulk API 切换别名与索引绑定的写权限标注，该 API 为原子操作，actions 中的动作编写顺序不影响交换执行。

```
POST /_aliases
{
  "actions": [
    {
      "add": {
        "index": "test",
        "alias": "write-index-alias1",
        "is_write_index": false
      }
    },
    {
      "add": {
        "index": "test2",
        "alias": "write-index-alias1",
        "is_write_index": true
      }
    }
  ]
}
```

创作人简介：

杨松柏，目前就职于好未来教育科技集团，任数据平台资深研发工程师。 长期关注 ELK、TiDB、clickhouse 等分布式存储技术，对于 Elasticsearch 和 TiDB 都有深入的理解。

博客：https://blog.csdn.net/yang52017

3.4.2.12 Reindex API

创作人：杨松柏

什么是 Reindex

将文档从源索引复制到目的地索引，称之为 Reindex。

在 Reindex 时可以进行数据的丰富、缩减以及字段的变更等。Reindex 可以简单的理解为 Scroll+Bulk_Insert。source 和 dest 都可以是已存在的索引、索引别名或数据流(Data Stream)。

此外，使用 Reindex 需要注意以下几点：

- 源和目的不能相同，比如不能将数据流 Reindex 给它自身。
- 源索引的文档中 _source 字段必须开启。
- Reindex 不会复制源的 setting 和源所匹配的模板，因此在调用 _reindex 前，你需要设置好目的索引 (action.auto_create_index 为 false 或者 -.* 时)。
- 目标索引的 mapping，主分片数，副本数等推荐提前配置。

Reindex 的主要场景：

- 集群升级：将数据从旧集群远程 Reindex 到新集群
- 索引备份

- 数据重构

前置要求

如果 Elasticsearch 集群配置了安全策略和权限策略，则进行 Reindex 必须拥有以下权限：

- 读取源的数据流、索引、索引别名等索引级别权限。
- 对于目的数据流、索引、索引别名的写权限。
- 如果需要使用 Reindex API 自动创建数据流和索引，则必须拥有对目的数据流、索引、索引别名的 auto_configure、create_index 或者 manage 等索引级别权限。
- 如果源为远程的集群，则 source.remote.user 用户必须拥有集群监控权限，和读取源索引、源索引别名、源数据流的权限。

如果 Reindex 的源为远程集群，必须在当前集群的请求节点 elasticsearch.yml 文件配置远程白名单 reindex.remote.whitelist。

创建数据流，需要提前配置好数据流的匹配索引模板，详情可参看 Set up a data stream：https://www.elastic.co/guide/en/elasticsearch/reference/7.11/set-up-a-data-stream.html

API 介绍

RESTful API

```
POST /_reindex
```

Query parameters

refresh

可选参数，枚举类型 (true,false,wait_for)，默认值为 false。

如果设置为 true， Elasticsearch 刷新受当前操作影响的数据，能够被立即搜索(即立即刷新，但是会对 Elasticsearch 的性能有一定的影响)。如果为 wait_for，则等待刷新以使当前操作对搜索可见，等待时间为默认为 1s（index.refresh_interval）。如果为 false，本次请求不执行刷新。

timeout

可选参数，时间值（time units），默认值为 1 分钟；每个索引周期中等待索引自动创建、动态映射更新，和等待活跃健康分片等的时间。该参数可以确保 Elasticsearch 在失败之前，基本等待的超时时间。实际等待时间可能更长，特别是在发生多个等待时。

wait_for_active_shards

可选参数，参数类型 string，默认值为 1（即只要一个分片处于活跃就可以执行该

操作）。在执行 Reindex 之前索引必须处于活动状态的分片副本数，可以设置为 all 或者小于 number_of_replicas+1 的任何正整数，比如你的索引主分片数目为 3，副本设置为 2，那么可以设置的最大正整数为 3，即副本份数加 1 (主分片)。

```
#因为实操集群只有三个节点，如下索引将会出现副本分片无法分配，
#index.routing.allocation.total_shards_per_node
#控制每个该索引只允许每个节点分配一个分片
PUT reindex_index-name-2
{
  "settings" :{
    "index" :{
      "number_of_shards" : "3",
      "number_of_replicas" : "2"
    },
    "index.routing.allocation.total_shards_per_node":1
  }
}
#插入一条数据
PUT reindex_index-name-1/_bulk
{ "index":{ } }
{ "@timestamp": "2099-05-06T16:21:15.000Z", "message": "192.0.2.42 - - [06/May/2099:16:21:15 +0000] \"GET /images/bg.jpg HTTP/1.0\" 200 24736" }

#重建索引
POST _reindex?wait_for_active_shards=2&timeout=5s
{
  "source": {
    "index": "reindex_index-name-1"
```

```
  },
  "dest": {
    "index": "reindex_index-name-2"
  }
}
```

由于 reindex_index-name-2 只有主分片分配成功，所以上面的_reindex 将失败：

```
{
  "took" : 5002,
  "timed_out" : false,
  "total" : 1,
  "updated" : 0,
  "created" : 0,
  "deleted" : 0,
  "batches" : 1,
  "version_conflicts" : 0,
  "noops" : 0,
  "retries" : {
    "bulk" : 0,
    "search" : 0
  },
  "throttled_millis" : 0,
  "requests_per_second" : -1.0,
  "throttled_until_millis" : 0,
  "failures" : [
    {
      "index" : "reindex_index-name-2",
```

```
            "type" : "_doc",
            "id" : "U_i8VnkBYYHWy1KlBJjc",
            "cause" : {
              "type" : "unavailable_shards_exception",
              "reason" : "[reindex_index-name-2][0] Not enough active copies to meet shard count of [2] (have 1, needed 2). Timeout: [5s], request: [BulkShardRequest [[reindex_index-name-2][0]] containing [index {[reindex_index-name-2][_doc][U_i8VnkBYYHWy1KlBJjc], source[{ \"@timestamp\": \"2099-05-06T16:21:15.000Z\", \"message\": \"192.0.2.42 - - [06/May/2099:16:21:15 +0000] \\\"GET /images/bg.jpg HTTP/1.0\\\" 200 24736\" }]}]]"
            },
            "status" : 503
          }
        ]
      }
```


wait_for_completion

可选参数，参数类型 Boolean，默认为 true。如果为 true，则请求为阻塞同步方式，请求会等到操作完成才返回。

requests_per_second

可选参数，参数类型 integer，默认为 -1（不进行限制）；限制请求的每秒子请求数。

require_alias

可选参数，参数类型 Boolean，默认为 true。如果为 true，dest.index 必须为索引别名。

scroll

可选参数，参数类型为时间类型（ time units），指定滚动搜索时索引的一致视图应保持多长时间。

slices

可选参数，参数类型 integer，默认值为 1（不切分成多个子任务）；该参数表示将一个任务切分成多少个子任务，并行执行。

max_docs

可选参数，参数类型 integer，默认值为对应索引的所有文档；要处理的最大文档数。

Request Body

conflicts

可选参数，参数类型枚举类型，默认为 abort；设置为 proceed，即使发生文档冲

突也继续 reindexing。

source

- index

必填参数，参数类型 string；值可以为数据流、索引名字、索引别名，如果源有多个，也可以接受逗号分隔的数据源数组。

- max_docs

可选参数，参数类型 integer；要被重新索引的最大文档数目。

- query

可选参数，参数类型查询对象（query object），按查询 DSL 过滤需要重新索引的文档。

- remote

remote 的子参数可接受如下：

参数	是否必填	类型	说明
host	否	string	索引 pattern 名源索引所在远程 ES 集群中任一节点地址；如果是需要从远程 集群复制数据，则该参数必填。
username	否	string	与远程主机进行身份验证的用户名；当远程集群需要认证时必填。
password	否	string	与远程主机进行身份验证的密码；当远程集群需要认证时必填。
socket_timeout	否	时间类型	默认为 30 秒；远程套接字读取超时。
connect_timeout	否	时间类型	默认为 30 秒；远程连接超时时间。

- size

可选参数，参数类型 integer； 每批要索引的文档数（批处理），在远程索引时确保批处理能够放在堆上缓冲区，缓冲区的默认大小为 100 MB。

- slice

slice 的子参数可接受如下：

参数	是否必填	类型	说明
id	否	integer	进行手动切片时的，设置的切片 id
max	否	integer	切片总数。

- sort

可选参数，参数类型 list；以逗号分隔的 ： 对列表（比如 name:desc），用于在获取源索引文档时，按照 sort 中字段值的排序要求进行排序。通常与 max_docs 参数结合使用，以控制哪些文档需要被重新索引。

注意：sort 参数在 7.6 版本已经被标注弃用，不建议在 Reindex 中进行排序。Reindex 中的排序不能保证按顺序索引文档，并阻止 Reindex 的进一步发展，如恢复能力和性能改进。如果与结合使用 max_docs，请考虑改为使用查询过滤器。

- _source

可选参数，参数类型 string，默认值为 true。该参数可以用于选择文档中哪些字段需要进行重新索引。如果设置为 true，将会重索引文档中的所有字段。

dest

- index

必填参数，参数类型 string；该参数表示目的地的表，值可以为数据流、索引名字、索引别名。

- version_type

可选参数，参数类型枚举；用于索引操作的版本控制；枚举值包括：internal，external，external_gt，external_gte。

详情参看 Version types :https://www.elastic.co/guide/en/elasticsearch/reference/7.11/docs-index_.html#index-version-types

- op_type

可选参数，参数类型枚举，默认为 index，枚举值包括：index，create；如果设置为 create，则目标索引不存在该文档就创建（可用于 reindex 续传补偿）。注意：如果 dest 是数据流，必须设置为 create，因为数据流只做 append。

- type

可选参数，参数类型 string，默认值为_doc；被重建索引的文档中文档类型；注意：该参数在 Elasticsearch 6 版本中已经标记弃用，已经没有任何实际意义。

script

- source

可选参数，参数类型 string；重新索引时用于更新文档 source 或元数据的脚本 .

- lang

可选参数，参数类型枚举；支持的脚本语言：painless, expression, mustache, java

更多脚本语言，请参考 Scripting：https://www.elastic.co/guide/en/elasticsearch/reference/7.11/modules-scripting.html

Response Body

执行_reindex 时的，响应体参数释意：

字段	类型	说明
took	integer	整个操作花费的总毫秒数
timed_out	Boolean	如果在重新索引期间出现的任何请求超时，则此标志设置为 true。
total	integer	成功处理的文档数
updated	integer	已成功更新的文档数，即重新索引的文档，在 dest 索引中存在具有相同 ID 的文档，并且更新成功的
created	integer	成功创建的文档数
deleted	integer	成功删除的文档数
batches	integer	由重新索引回调的滚动响应数
noops	integer	由于重新索引的脚本为 ctx.op 返回 noop 值而被忽略的文档数
version_conflicts	integer	重新索引命中的版本冲突数

字段	类型	说明
retries	integer	重索引尝试的重试次数；bulk 是重试的批量操作数，search 是重试的搜索操作数
throttled_millis	integer	请求休眠以符合 requests_per_second
requests_per_second	integer	在重新索引期间每秒有效执行的请求数
throttled_until_millis	integer	此字段在 _reindex 响应中应始终等于零;该参数只有在使用任务 API（Task API）时才有意义，在任务 API 中，它指示下次再次执行限制请求的时间，以便符合每秒的请求数
failures	数组	如果进程中有任何不可恢复的错误，则返回失败数组。如果数组不为空，那么请求会因为这些失败而中止。重新索引是使用批处理实现的，任何失败都会导致整个进程中止，但当前批处理中的所有失败都会收集到数组中。你可以使用 conflicts 参数，避免因为版本冲突而中止重建索引 Reindex 的一些技巧

异步执行 Reindex

如果请求的查询参数 wait_for_completion 设置为 false，Elasticsearch 将会执行一些预检查，然后发起一个 task,来运行你的 Reindex 任务，并立即返回你一个 taskid，然后你可以通过这个 taskid，去查看任务的运行结果，运行结果记录在系统索引.tasks;

如果任务执行完成，你可以删除掉该文档，以使 Elasticsearch 释放空间。

```
#异步执行 reindex 任务
POST _reindex?wait_for_completion=false
{
  "source": {
    "index": "reindex_index-name-1"
  },
  "dest": {
    "index": "reindex_index-name-2"
  }
}
#返回立即返回的任务 id
{
  "task" : "ydZx8i8HQBe69T4vbYm30g:20987804"
}
#查看任务的运行情况
GET _tasks/ydZx8i8HQBe69T4vbYm30g:20987804#response 返回结果
{
  "completed" : true,
  "task" : {
    "node" : "ydZx8i8HQBe69T4vbYm30g",
    "id" : 20987804,
    "type" : "transport",
    "action" : "indices:data/write/reindex",
    "status" : {
      "total" : 2,
      "updated" : 0,
      "created" : 2,
      "deleted" : 0,
```

```
            "batches" : 1,
            "version_conflicts" : 0,
            "noops" : 0,
            "retries" : {
              "bulk" : 0,
              "search" : 0
            },
            "throttled_millis" : 0,
            "requests_per_second" : -1.0,
            "throttled_until_millis" : 0
          },
          "description" : "reindex from [reindex_index-name-1] to [reindex_index-name-2][_doc]
",
          "start_time_in_millis" : 1620539345400,
          "running_time_in_nanos" : 84854825,
          "cancellable" : true,
          "headers" : { }
        },
        "response" : {
          "took" : 82,
          "timed_out" : false,
          "total" : 2,
          "updated" : 0,
          "created" : 2,
          "deleted" : 0,
          "batches" : 1,
          "version_conflicts" : 0,
          "noops" : 0,
          "retries" : {
            "bulk" : 0,
            "search" : 0
```

```
      },
      "throttled" : "0s",
      "throttled_millis" : 0,
      "requests_per_second" : -1.0,
      "throttled_until" : "0s",
      "throttled_until_millis" : 0,
      "failures" : [ ]
    }
}
#实际任务运行结果会记录在 .tasks 索引
GET  .tasks/_doc/ydZx8i8HQBe69T4vbYm30g:20987804#返回值如下
{
  "_index" : ".tasks",
  "_type" : "_doc",
  "_id" : "ydZx8i8HQBe69T4vbYm30g:20987804",
  "_version" : 1,
  "_seq_no" : 1,
  "_primary_term" : 1,
  "found" : true,
  "_source" : {
    "completed" : true,
    "task" : {
      "node" : "ydZx8i8HQBe69T4vbYm30g",
      "id" : 20987804,
      "type" : "transport",
      "action" : "indices:data/write/reindex",
      "status" : {
        "total" : 2,
        "updated" : 0,
        "created" : 2,
        "deleted" : 0,
```

```
            "batches" : 1,
            "version_conflicts" : 0,
            "noops" : 0,
            "retries" : {
              "bulk" : 0,
              "search" : 0
            },
            "throttled_millis" : 0,
            "requests_per_second" : -1.0,
            "throttled_until_millis" : 0
          },
          "description" : "reindex from [reindex_index-name-1] to [reindex_index-name-2][_d
oc]",
          "start_time_in_millis" : 1620539345400,
          "running_time_in_nanos" : 84854825,
          "cancellable" : true,
          "headers" : { }
        },
        "response" : {
          "took" : 82,
          "timed_out" : false,
          "total" : 2,
          "updated" : 0,
          "created" : 2,
          "deleted" : 0,
          "batches" : 1,
          "version_conflicts" : 0,
          "noops" : 0,
          "retries" : {
            "bulk" : 0,
            "search" : 0
```

```
        },
        "throttled" : "0s",
        "throttled_millis" : 0,
        "requests_per_second" : -1.0,
        "throttled_until" : "0s",
        "throttled_until_millis" : 0,
        "failures" : [ ]
      }
    }
}
```

多源重建索引

如果有许多源需要重新索引，通常最好一次 Reindex 一个源的索引，而不是使用 glob 模式来选取多个源。这样如果出现任何的错误，你可以删除有问题部分，然后选择特定的源重新索引（dest 的 op_type 可以设置为 create 只重索引缺失的文档）；另外一个好处，你可以并行运行这些 reindex 任务。

```
#!/bin/bashfor index in i1 i2 i3 i4 i5; do
  curl -H Content-Type:application/json -XPOST localhost:9200/_reindex?pretty -d'{
    "source": {
      "index": "'$index'"
    },
    "dest": {
      "index": "'$index'-reindexed"
    }
  }'done
```

对 Reindex 限流

设置 requests_per_second 为任意的正十进制数（如 1.4，6，...1000 等），以限制批量操作_reindex 索引的速率。通过在每个批处理中，设置等待时间来限制请求；可以通过设置 requests_per_second=-1，来关闭限流操作。

限流是通过在每个批处理之间设置等待时间，因此 _reindex 在内部使用 scroll 的超时时间，应当将这个等待时间考虑进去。等待时间=批大小/requests_per_second - 批写入耗时；默认情况下，批处理大小为 1000，因此如果 requests_per_second 设置为 500：

```
target_time = 1000 / 500 per second = 2 seconds
wait_time = target_time - write_time = 2 seconds - 0.5 seconds = 1.5 seconds
```

由于批处理是作为单个_bulk 请求发出的，因此较大的批处理大小，会导致 Elasticsearch 创建许多请求，然后等待一段时间，再开始下一组请求；这种情况可能会造成 Elasticsearch 周期性的抖动。

动态调整限流

可以使用_rethrottle API 在正在运行的重新索引上更改 requests_per_second 的值：

```
POST _reindex/r1A2WoRbTwKZ516z6NEs5A:36619/_rethrottle?requests_per_second=-1
```

taskid 可以通过 task API 进行获取。重新调整 requests_per_second，如果是加快查询速度则可以立即生效，如果是降低查询速度，则需要在完成当前批处理后生效，这样可以避免 scroll 超时。

切片

Reindex 支持切片 scroll 以并行化重新索引过程，从而提高 Reindex 的效率。

注意：如果源索引是在远程的 Elasticsearch 集群，是不支持手动或自动切片的。

手动切片

通过为每个请求提供切片 ID 和切片总数。

示例如下：

```
POST _reindex
{
  "source": {
    "index": "my-index-000001",
    "slice": {
      "id": 0,
      "max": 2
    }
  },
  "dest": {
```

```
    "index": "my-new-index-000001"
  }
}

POST _reindex
{
  "source": {
    "index": "my-index-000001",
    "slice": {
      "id": 1,
      "max": 2
    }
  },
  "dest": {
    "index": "my-new-index-000001"
  }
}
```

可以通过以下方式验证此功能：

```
#避免还没有形成 segments，文档不可见
GET _refresh
#查看文档的个数
GET my-new-index-000001/_count
#或者
POST my-new-index-000001/_search?size=0&filter_path=hits.total
```

返回结果如下：

```
{
  "hits": {
    "total" : {
        "value": 120,
        "relation": "eq"
    }
  }
}
```

自动切片

还可以使用 Sliced scroll 基于文档_id 进行切片，让_reindex 自动并行化；通过设定 slices 参数的值来实现。

示例如下：

```
POST _reindex?slices=5&refresh
{
  "source": {
    "index": "my-index-000001"
  },
  "dest": {
    "index": "my-new-index-000001"
  }
}
```

可以通过以下方式验证此功能：

```
POST my-new-index-000001/_search?size=0&filter_path=hits.total
```

返回结果如下：

```
{
  "hits": {
    "total" : {
        "value": 120,
        "relation": "eq"
    }
  }
}
```

设置 slices 为 auto 会让 Elasticsearch 选择要使用的切片数。此设置将每个分片使用一个切片，直到达到某个限制。如果有多个源，它将基于分片数量最少的索引或 Backing index 确定切片数。

```
POST _reindex?slices=auto&refresh
{
  "source": {
    "index": "my-index-000001"
  },
  "dest": {
    "index": "my-new-index-000001"
  }
}
# 由源码可知，slices 实际被修改为 0
```

```
if (slicesString.equals(AbstractBulkByScrollRequest.AUTO_SLICES_VALUE)) {
                return AbstractBulkByScrollRequest.AUTO_SLICES;
            }
public static final int AUTO_SLICES = 0;
public static final String AUTO_SLICES_VALUE = "auto";
```

_reindex 中添加 slices，将会自动完成上面手动切片创建的请求；自动切片创建的子请求有些不一样的特征:

- 你可以使用 Tasks APIs 查看这些子请求，这些子请求是带有 slices 请求任务的”子”任务。
- 获取带有参数 slices 请求的任务状态，将只返回包含已完成切片的状态。
- 可以分别对这些子请求，进行任务取消或者重新调节限速。
- 重新调速带有 slices 参数的请求，将会按比例调速它的子任务。
- 取消带有 slices 参数的请求，将会取消它的所有子任务。
- 由于切片的性质，可能会每个切片的文档并不均匀，会出现某些切片某些切片可能比其他切片大；但是所有文档都会被划分到某个切片中。
- slices 请求如果含有 requests_per_second 和 max_docs，将会按比例分配给每个子请求。结合上面关于分布不均匀的观点，将 max_docs 与切片一起使用可能会出现满足条件的 max_docs 文档不被重新索引。
- 尽管这些快照几乎都是在同一时间获取，但是每个子请求，可能获取的源快照会有稍微的不同。

合理选择切片数目

自动切片，设置 slices 为 auto，将为大多数索引选择一个合理切片数目。如果手动切片或以其他方式调整自动切片，应当明白以下几个点：

- 当 slices 的数目等于索引的数目时，查询性能是最优的。设置 slices 高于分片数通常不会提高效率，反而会增加开销（CPU，磁盘 IO 等）。
- 索引性能会在可用资源与切片数之间线性地缩放。
- 查询或索引性能在运行时是否占主导地位，取决于重新索引的文档和集群资源。

重新索引的路由

默认情况下，如果_reindex 看到一个带有路由的文档，则路由将被保留，除非它被脚本更改。可以在 dest 的 JSON 体内上重新设置 routing，从而改变之前的路由值，routing 的可取值如下：

keep

keep 为默认值，将为每个匹配项发送的批量请求的路由，设置为匹配项旧的路由（就保持旧的路由方式）。

discard

将发送的批量请求的路由设置为 null

=

将发送的批量请求的路由设置为=之后的值

示例，使用以下请求将 source_index 中公司名称为 cat 的所有文档复制到 source_index 且路由设置为 cat：

```
POST _reindex
{
  "source": {
    "index": "source_index",
    "query": {
      "match": {
        "company": "cat"
      }
    }
  },
  "dest": {
    "index": "dest_index",
    "routing": "=cat"
  }
}
```

默认情况下，_reindex 使用滚动批处理 1000。你可以使用元素中的 size 字段更改批处理大小：

```
POST _reindex
{
  "source": {
    "index": "source_index",
    "size": 100
  },
  "dest": {
    "index": "dest_index",
    "routing": "=cat"
  }
}
```

重索引使用预处理 Pipeline

重索引也可以使用 ingest pipeline 的特性，来富化数据；示列如下：

```
POST _reindex
{
  "source": {
    "index": "source"
  },
  "dest": {
    "index": "dest",
     "pipeline": "some_ingest_pipeline" #提前定义的 pipeline
  }
}
```

实战示例

基于查询重新索引文档

可以通过在 source 中添加查询条件，对有需要的文档进行重新索引；

例如，复制 user.id 值为 kimchy 文档到 my-new-index-000001：

```
POST _reindex
{
  "source": {
    "index": "my-index-000001",
    "query": {
      "term": {
        "user.id": "kimchy"
      }
    }
  },
  "dest": {
    "index": "my-new-index-000001"
  }
}
```

基于 max_docs 重新索引文档

通过在请求体中设置 max_docs 参数，控制重建索引的文档的个数。

例如：从 my-index-000001 复制一个文档到 my-new-index-000001：

```
POST _reindex
{
  "max_docs": 1,
  "source": {
    "index": "my-index-000001"
  },
  "dest": {
    "index": "my-new-index-000001"
  }
}
```

基于多源重新索引

source 的 index 属性值可以是一个 list；这样可以允许复制多个源的文档到目的数据流或索引，但是需要注意多个源的文档中字段的类型必须一致。

例如复制 my-index-000001 和 my-index-000002 索引的文档：

```
POST _reindex
{
  "source": {
    "index": ["my-index-000001", "my-index-000002"]
  },
  "dest": {
    "index": "my-new-index-000002"
```

```
    }
}
```

选择字段重新索引

只重新索引每个文档筛选的字段；

例如，以下请求仅重新索引每个文档的 user.id 和_doc 字段：

```
POST _reindex
{
  "source": {
    "index": "my-index-000001",
    "_source": ["user.id", "_doc"]
  },
  "dest": {
    "index": "my-new-index-000001"
  }
}
```

通过重新索引修改文档中字段名字

_reindex 可用于复制源索引文档，在写入目的索引之前，重命名字段；假设 my-index-000001 索引有以下文档：

```
POST my-index-000001/_doc/1?refresh
{
  "text": "words words",
  "flag": "foo"
}
```

但是你想把字段名 flag 替换成 tag，处理手段如下（当然也可以用 ingest pipeline）：

```
POST _reindex
{
  "source": {
    "index": "my-index-000001"
  },
  "dest": {
    "index": "my-new-index-000001"
  },
  "script": {
    "source": "ctx._source.tag = ctx._source.remove(\"flag\")"
  }
}
```

现在获取新索引文档：

```
GET my-new-index-000001/_doc/1
```

返回值如下：

```
{
  "found": true,
  "_id": "1",
  "_index": "my-new-index-000001",
  "_type": "_doc",
  "_version": 1,
  "_seq_no": 44,
  "_primary_term": 1,
  "_source": {
    "text": "words words",
    "tag": "foo"
  }
}
```

重新索引每日索引

_reindex 结合 Painless 脚本来重新索引每日索引，将新模板应用于现有文档。假设你有如下索引并包含下列文档

```
PUT metricbeat-2021.05.10/_doc/1?refresh
{"system.cpu.idle.pct": 0.908}

PUT metricbeat-2021.05.11/_doc/1?refresh
{"system.cpu.idle.pct": 0.105}
```

通配 metricbeat-*索引的新模板，已经加载到 Elasticsearch 中，但是该模板只会对新建的索引生效。Painless 可用于重新索引现有文档，并应用新模板。下面的脚本从

索引名中提取日期，并创建一个新索引，新索引名添加 -1。metricbeat-2021.05.10 所有的数据将会被重建到 metricbeat-2021.05.10-1。

```
POST _reindex
{
  "source": {
    "index": "metricbeat-*"
  },
  "dest": {
    "index": "metricbeat"
  },
  "script": {
    "lang": "painless",
    "source": "ctx._index = 'metricbeat-' + (ctx._index.substring('metricbeat-'.length(), ctx._index.length())) + '-1'"
  }
}
```

之前 metricbeat 索引的数据，都能从新的索引从获取到：

```
GET metricbeat-2021.05.10-1/_doc/1
GET metricbeat-2021.05.11-1/_doc/1
```

提取源的随机子集

_reindex 可用于提取源的随机子集以进行测试：

```
POST _reindex
{
  "max_docs": 10,
  "source": {
    "index": "my-index-000001",
    "query": {
      "function_score" : {
        "random_score" : {},
        "min_score" : 0.9      #备注 1
      }
    }
  },
  "dest": {
    "index": "my-new-index-000001"
  }
}
```

可能需要根据从源中提取数据的相对数量，来调整 min_score 的值。

重新索引时修改文档

像_update_by_query 一样，_reindex 支持使用 script 修改文档；不同的是，_reindex 中使用脚本可以修改文档的元数据。

此示例增加了源文档的版本：

```
POST _reindex
{
  "source": {
    "index": "my-index-000001"
  },
  "dest": {
    "index": "my-new-index-000001",
    "version_type": "external"
  },
  "script": {
    "source": "if (ctx._source.foo == 'bar') {ctx._version++; ctx._source.remove('foo')}",
    "lang": "painless"
  }
}
```

与_update_by_query 一样，你可以设置 ctx.op 更改在 dest 上执行的操作：

noop

如果决定不必在目标中为文档重新索引，则需要在脚本中设置 ctx.op=“noop”。响应主体中的 noop 计数器将报告不做任何的操作。

delete

如果必须从目标（dest）中删除文档，则需要在脚本中设置 ctx.op=“delete”。删除将在响应正文中的已删除计数器中报告 。

设置 ctx.op 为其他任何值都将返回错误，就像设置中的其他任何字段一样 ctx。

同时还可以更改一些索引元信息，但是谨慎操作：

- _id
- _index
- _version
- _routing

如果将_version 设置为 null 或将其从 ctx 映射中清除，就像不在索引请求中发送版本一样；则无论目标上的版本或_reindex 请求中使用的版本类型如何，都会导致目标中的文档被覆盖。

远程重新索引

重新索引支持从远程 Elasticsearch 复制数据：

```
POST _reindex
{
  "source": {
    "remote": {
      "host": "http://otherhost:9200",
      "username": "user",
      "password": "pass"
    },
```

```
        "index": "my-index-000001",
        "query": {
          "match": {
            "test": "data"
          }
        }
      },
      "dest": {
        "index": "my-new-index-000001"
      }
    }
```

Host 参数必须包含 scheme，host，port（如：https://otherhost:9200)或者代理路径（https://otherhost:9200/proxy）。

_reindex 需要基本授权认证链接远程 Elasticsearch 集群，才需要用户名和密码参数;使用基本身份验证时请确保使用 协议，否则密码将以纯文本形式发送。

如果是 reindex 远程集群的数据，则必须在当前集群的某个节点（请求发送到的那个节点，即协调节点）配置白名单，在 elasticsearch.yml 文件中添加 reindex.remote.whitelist 属性，该属性的值为请求远程集群节点的 host:port，可以用逗号分隔配置多个，也可以使用通配符方式；

示例如下：

```
reindex.remote.whitelist: "otherhost:9200, another:9200, 127.0.10.*:9200, localhost:*"
```

此外在做远程 reindex 时，需要注意集群之前的版本兼容问题；Elasticsearch 不支持跨大版本的向前兼容，如不能从 7.x 群集重新索引到 6.x 群集。

从远程服务器重新索引时使用堆内缓冲区，默认最大为 100mb；如果远程索引的文档非常大，那么批的 size 就应该设置的小一点。

如下代码块设置 batch size 为 10：

```
POST _reindex
{
  "source": {
    "remote": {
      "host": "http://otherhost:9200"
    },
    "index": "source",
    "size": 10,
    "query": {
      "match": {
        "test": "data"
      }
    }
  },
  "dest": {
    "index": "dest"
  }
}
```

连接远程 Elasticsearch 集群，可以通过 socket_timeout 和 connect_timeout，分别设置 socket 读取超时时间和链接超时时间，在一定程度上保证 Reindex 的稳定性（网络延迟问题），两者的默认值均为 30s。

如下示例分别设置 socket 读取超时为 1 分钟和连接超时时间 10s：

```
POST _reindex
{
  "source": {
    "remote": {
      "host": "http://otherhost:9200",
      "socket_timeout": "1m",
      "connect_timeout": "10s"
    },
    "index": "source",
    "query": {
      "match": {
        "test": "data"
      }
    }
  },
  "dest": {
    "index": "dest"
  }
}
```

配置 SSL 参数

从远程集群 reindex 支持配置 ssl；这些参数是无法在_reindex 请求体配置；必须在 elasticsearch.yml 文件中指定，但在 Elasticsearch 密钥库中添加的安全设置除外，可支持的 ssl 参数。

配置如下：

参数	描述
reindex.ssl.certificate_authorities	应当信任的 PEM 编码证书文件的路径列表；但不能同时指定 reindex.ssl.certificate_authorities 和 reindex.ssl.truststore.path
reindex.ssl.truststore.path	要信任的证书的 Java Keystore 文件的路径；该密钥库可以采用”JKS”或“PKCS＃12”格式，但不能同时指定 reindex.ssl.certificate_authorities 和 reindex.ssl.truststore.path
reindex.ssl.truststore.password	信任库的密码（reindex.ssl.truststore.path）。此设置不能用于 reindex.ssl.truststore.secure_password
reindex.ssl.truststore.secure_password	信任库的密码（reindex.ssl.truststore.path）。此设置不能用于 reindex.ssl.truststore.password
reindex.ssl.truststore.type	信任库的类型（reindex.ssl.truststore.path）。必须为 jks 或 PKCS12。如果信任库路径以“ .p12”， “.pfx”或“ pkcs12”结尾，则此设置默认为 PKCS12。否则，默认为 jks。

参数	描述
reindex.ssl.verification_mode	表示防止中间人攻击和证书伪造的验证类型。其中一个 full（验证主机名和证书路径）， certificate （验证证书路径，而不是主机名）或 none（不执行任何验证-这是在生产环境中强烈反对）。默认为 full
reindex.ssl.certificate	指定用于 HTTP 客户端身份验证的 PEM 编码证书（或证书链）的路径（如果远程集群需要）。此设置 reindex.ssl.key 还需要设置。你不能同时指定 reindex.ssl.certificate 和 reindex.ssl.keystore.path。
reindex.ssl.key	指定与用于客户端身份验证（reindex.ssl.certificate）的证书相关联的 PEM 编码的私钥的路径。你不能同时指定 reindex.ssl.key 和 reindex.ssl.keystore.path。
reindex.ssl.key_passphrase	指定用于 reindex.ssl.key 加密 PEM 编码的私钥（reindex.ssl.key）的密码。不能与一起使用 reindex.ssl.secure_key_passphrase
reindex.ssl.secure_key_passphrase	指定用于 reindex.ssl.key 加密 PEM 编码的私钥（reindex.ssl.key）的密码。不能与一起使用 reindex.ssl.key_passphrase
reindex.ssl.keystore.path	指定密钥库的路径，该密钥库包含用于 HTTP 客户端身份验证的私钥和证书（如果远程集群需要）。该密钥库可以采用“ JKS”或“ PKCS＃12”格式。你不能同时指定 reindex.ssl.key 和 reindex.ssl.keystore.path

参数	描述
reindex.ssl.keystore.type	密钥库的类型（reindex.ssl.keystore.path）。必须为 jks 或 PKCS12。如果密钥库路径以“ .p12”，“.pfx” 或 “ pkcs12” 结尾，则此设置默认为 PKCS12。否则，默认为 jks
reindex.ssl.keystore.password	密钥库（reindex.ssl.keystore.path）的密码。此设置不能用于 reindex.ssl.keystore.secure_password
reindex.ssl.keystore.secure_password	密钥库（reindex.ssl.keystore.path）的密码。此设置不能用于 reindex.ssl.keystore.password
reindex.ssl.keystore.key_password	密钥库（reindex.ssl.keystore.path）中密钥的密码。默认为密钥库密码。此设置不能用于 reindex.ssl.keystore.secure_key_password
reindex.ssl.keystore.secure_key_password	密钥库（reindex.ssl.keystore.path）中密钥的密码。默认为密钥库密码。此设置不能用于 reindex.ssl.keystore.key_password

3.4.2.13 Rollover API

创作人：杨松柏

了解 Elasticsearch 的同学应该都知道，索引的主分片在设定之后，改变（shrink，split，reindex）主分片数目的成本相当大；因此在设计之初，一定要规划好索引的分片数目。如果集群的中节点数目固定，且写入的数据不会再有更新操作或者更新操作极其少；可以使用 Rollover index 的方式来限制每个索引的大小。

Rollover：

若 rollover-target 绑定的当前索引满足设定的条件，执行滚动操作将会为 rollover-target 创建新索引。滚动目标可以是索引别名或者数据流；

- 当滚动目标是别名时，执行滚动别名将指向新的索引。
- 当滚动目标是数据流时，数据流将数据写入到新的索引，且新索引名后缀自增 1。

```
POST /alias1/_rollover/my-index-000002
{
  "conditions": {
    "max_age":   "7d",
    "max_docs":  2,
    "max_size": "5gb"
  }
}
```

如果需要实现 Rollover 自动化，可以自行实现一个定时任务请求该 API 或者使用 Elasticsearch 的 index lifecycle management (ILM) 功能。

API 介绍

Rollover API

```
POST /<rollover-target>/_rollover/<target-index>?wait_for_active_shards=<number>
POST /<rollover-target>/_rollover?wait_for_active_shards=<number>
```

URI 参数

必填参数，参数类型 string。将已存的索引别名或数据流，分配给目标索引，来完成执行滚动。

可选参数，参数类型 string。代表要被创建和分配索引别名的目标索引名称；

目标索引名称必须遵循以下规则：

- 所有字符必须小写
- 不允许包含\, /, *, ?, ", <, >, |, (space character), ,, #
- 在 7.0 版本之前允许包含：，在 7.0+不再支持
- 不能够以-，_，+开头
- 名称不能够是.或者..

- 长度不能超过 255 字节(注意：这是字节数限制，如果是字符得看表示一个字符需要多少个字节)
- 名称以 . 开头已经过时；除了隐藏索引和由插件管理的内部索引

如果 rollover-target 为数据流，<target-index>参数是不被允许的；执行 Rollover 数据流会按照相应规则生产新的索引：格式为.ds-<rollover-target>-000001，其中-000001，数字必为 6 位数，左侧高位用 0 补充，每发生一次 Rollover 自增加 1。

如果 rollover-target 为一个索引别名，且别名绑定的索引以-<number>结尾；如果不指定<target-index>执行 Rollover ，将生成新的索引 indexName-<number>+1。

如果别名现绑定的索引名称不满足以-<number>形式结尾，则必须指定<target-index>。

查询参数

dry_run

可选参数，参数类型为 boolean，默认值为 false。该参数的作用为检测 index 是否满足提供的 rollover 条件。 如果设置为 true，将会完成检查工作，但不会真正执行 rollover 。

include_type_name

可选参数，参数类型为 boolean，默认值为 false。在 mappings 体内要求必须有 mapping type。

wait_for_active_shards

可选参数，参数类型为 string，默认值为 1（即只要一个主分片处于活跃便可以执行操作）。在执行 rollover,要求处于活跃状态的索引分片副本数目，以保证数据的可靠和安全性；设置为 all 或任何正整数，最多为索引分片的总数（number_of_replicas+1）。

详情可以参考 Active shards:https://www.elastic.co/guide/en/elasticsearch/reference/7.10/docs-index_.html#index-wait-for-active-shards

master_timeout

可选参数，value 值的单位可为 d、h、m、s、ms、micros、nanos，默认值为 30 s。等待连接到主节点的时间。如果在超时时间阈值之前没有收到响应，则请求失败并返回错误。

timeout

可选参数，value 值的单位可为 d、h、m、s、ms、micros、nanos，默认值为 30 s。请求等待响应的时间。如果在超时时间阈值之前没有收到响应，则请求失败并返回错误。

请求体参数

aliases

可选参数，alias object。包含索引的 Index aliases。详情可以参看索引别名，批量操作的相关内容（即_aliases API）。

conditions

可选参数,参数类型 object。如果设置条件，则只有现有索引达到条件集任何一个条件的阈值才会完成 rollover。如果省略，则无条件执行 rollover。条件集包括：

- max_age

可选参数，value 值的单位可为 d、h、m、s、ms、micros、nanos；依据索引的创建时间与现在时间差值作为阈值。

- max_docs

可选参数，参数类型为 integer；索引的最大文档数目；计数不包括自上次 refresh 之后新添加的文档和副本分片的文档。

- max_size

可选参数， byte units 值（单位 b、kb、mb、gb、tb、pb）。索引大小的最大值，只计算索引主分片的大小，副本不包括在内。

mappings

可选参数，参数 mapping object。为索引进行数据建模，定义字段 schema 等。如果设置，mapping 可能包含：

- 字段名字（Field names）
- 字段的数据类型（Field data types）
- mapping 中一些修饰参数（Mapping parameters）

settings

可选参数，参数 index setting object。索引的配置选项；比如设置滚动产生的新索引副本分片数目等等。

详情可以参看 Index Settings：https://www.elastic.co/guide/en/elasticsearch/reference/7.11/index-modules.html#index-modules-settings

Rollover 的分类

依据不同 rollover target 和别名情况，可以简单的分为三种执行情况：

别名只绑定一个索引

rollover target 为别名，且别名和索引一对一关系；滚动请求过程如下：

- 创建一个新索引
- 给新索引添加别名
- 将别名与之前绑定索引的关联关系移除

别名绑定多个索引

`rollover target` 为别名，且与多个索引建立了绑定关系，只有其中一组的绑定关系`**is_write_index**`的值为 true（如果对`**is_write_index**`有疑问，请参看索引别名小节）。在这种情况下，滚动请求过程如下：

- 创建一个新索引
- 新索引于别名的绑定时，将 is_write_index 设置为 true
- 修改别名与旧索引的绑定关系，将 is_write_index 设置为 false。

rollover-target 为数据流

`rollover target` 为数据流，滚动请求过程如下：

- 创建一个新索引
- 在数据流上添加新索引作为备份索引和写索引
- 增加数据流的 `generation` 属性

实战示例

基础示例

别名与索引是一对一关系，执行 _rollover 之后，别名将关联到新的索引：

```
PUT /logs-1   #备注 1
{
  "aliases": {
    "logs_write": {}
  }
}
# Add > 2 documents to logs-1
PUT logs-1/_bulk
{"index":{}}
{"user.id":"kimchy"}
{"index":{}}
{"user.id":"tom"}

POST /logs_write/_rollover   #备注 2
{
  "conditions": {
    "max_age":    "7d",
    "max_docs":   2,
    "max_size":   "5gb"
  }
}
```

创建索引 logs-1 且与别名 logs_write 绑定。

logs_write 指向的索引满足条件集中的任何一个条件（索引“年龄值”>=7 天或者主分片文档数目>=2 或者主分片 size>=5gb）;则 logs-000002 将会被创建，且别名 logs_write 将更新绑定到 logs-000002。

如上代码块 Rollover API 返回值如下：

```
{
  "acknowledged": true,
  "shards_acknowledged": true,
  "old_index": "logs-1",
  "new_index": "logs-000002",
  "rolled_over": true, #索引是否执行了滚动
  "dry_run": false, # 是否为 dry_run 模式
  "conditions": {  # 条件集触发结果
    "[max_age: 7d]": false,
    "[max_docs: 2]": true,
    "[max_size: 5gb]": false,
  }
}
```

通过查看索引与别名的绑定关系，可以看到 logs_write 只绑定 logs-000002：

```
# 查看命令
GET _cat/aliases/logs_write?v

#返回值如下
alias       index          filter routing.index routing.search is_write_index
logs_write logs-000002 -          -
```

基于数据流滚动

数据流必须要提前设定一个索引模板，否则无法创建数据流。

```
PUT _index_template/template
{
  "index_patterns": ["my-data-stream*"],
  "data_stream": { }
}
```

创建数据流

```
PUT /_data_stream/my-data-stream # 备注 1
```

创建一个名为 my-data-stream 的数据流，并且初始化一个名字为 my-data-stream-000001 的 backing 索引。

查看 GET /_data_stream/my-data-stream。返回值如下:

```
{
  "data_streams" : [
    {
      "name" : "my-data-stream",
      "timestamp_field" : {
        "name" : "@timestamp"
      },
```

```
      "indices" : [
        {
          "index_name" : ".ds-my-data-stream-000001",
          "index_uuid" : "Vir6yRm4S42k8n22mZ5YBw"
        }
      ],
      "generation" : 1,
      "status" : "GREEN",
      "template" : "my-index-template",
      "ilm_policy" : "my-lifecycle-policy"
    }
  ]
}
```

插入数据，然后执行滚动:

```
# Add > 2 documents to my-data-stream
#为了能够实时的看到效果插入数据时，加上 refresh 参数
PUT my-data-stream/_bulk?refresh
{ "create":{ } }
{ "@timestamp": "2099-05-06T16:21:15.000Z", "message": "192.0.2.42 - - [06/May/2099:16:21:15 +0000] \"GET /images/bg.jpg HTTP/1.0\" 200 24736" }
{ "create":{ } }
{ "@timestamp": "2099-05-06T16:25:42.000Z", "message": "192.0.2.255 - - [06/May/2099:16:25:42 +0000] \"GET /favicon.ico HTTP/1.0\" 200 3638" }
{ "create":{ } }
{ "@timestamp": "2099-05-06T16:27:42.000Z", "message": "192.0.2.255 - - [06/May/2099:16:25:42 +0000] \"GET /favicon.ico HTTP/1.0\" 200 3638" }
```

```
POST /my-data-stream/_rollover #备注 1
{
  "conditions" : {
    "max_age": "7d",
    "max_docs": 2,
    "max_size": "5gb"
  }
}
```

如果当前的写数据索引满足条件集中的任何一个条件；则滚动目标将创建一个新的 backing 索引 my-data-stream-000002，且新的 backing 索引将作为写入数据的索引。

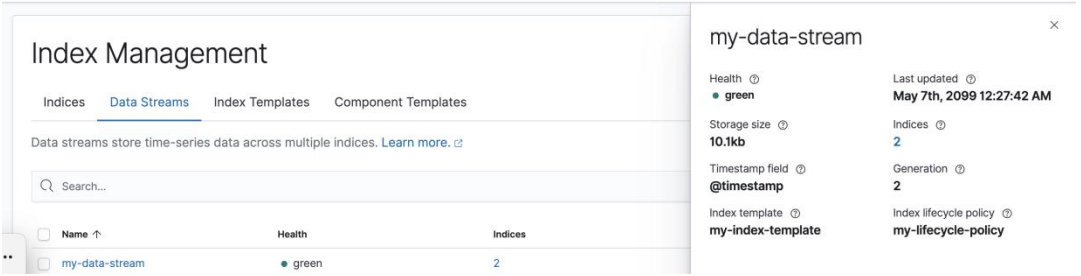
上面代码块的 Rollover API 返回值如下：

```
{
  "acknowledged" : false,
  "shards_acknowledged" : false,
  "old_index" : ".ds-my-data-stream-000001",#数据流之前对应的写数据索引
  "new_index" : ".ds-my-data-stream-000002",,#数据流新对应的写数据索引
  "rolled_over" : true,#索引是否执行滚动
  "dry_run" : false, # 是否为 dry_run 模式
  "conditions" : { # 滚动条件集，集触发情况
    "[max_size: 5gb]" : false,
    "[max_docs: 2]" : true,
    "[max_age: 7d]" : false
  }
}
```

再次查看数据流:

```
{
  "data_streams" : [
    {
      "name" : "my-data-stream", # 数据流名称
      "timestamp_field" : {
        "name" : "@timestamp"
      },
      "indices" : [
        {
          # backing 索引
          "index_name" : ".ds-my-data-stream-000001",
          "index_uuid" : "Vir6yRm4S42k8n22mZ5YBw"
        },
        {
          "index_name" : ".ds-my-data-stream-000002",
          "index_uuid" : "WcctdbIqSMqk3MmefRdfDQ"
        }
      ],
      "generation" : 2, #当前是那一个版本的后备索引作为写索引
      "status" : "GREEN",
      "template" : "my-index-template", # 适配的索引模板
      "ilm_policy" : "my-lifecycle-policy"
    }
  ]
}
```

也可以通过 Kibana 进行查看:



目标索引设定 setting

新索引的 setting、mapping 和 aliases 可以取自索引名匹配的任何索引模板。如果 <rollover-target>为索引别名，你可以在 rollover API 的请求体中指定 settings, mappings 和 aliases，与创建索引的 API（create index API）类似。如果请求体指定的值优先于匹配的索引模板。

如下示例重写了 index.number_of_shards setting:

```
PUT /logs-000001
{
  "aliases": {
    "logs_write": {}
  }
}

POST /logs_write/_rollover
```

```
{
  "conditions" : {
    "max_age": "7d",
    "max_docs": 2,
    "max_size": "5gb"
  },
  "settings": {
    "index.number_of_shards": 2
  }
}
```

指定目标索引名称

如果<rollover-target>是索引别名，并且绑定的索引名称以-<number>结尾；如果不指定目标索引，发生滚动时生成的新索引名称为 indexName-6 位数字，每滚动一次数字自增一次。

例如索引为 logs-1（或者 logs-000001），执行滚动后新的索引名字为 logs-000002，6 位数字低位自增高位补 0。

如果旧的索引名称不满足 indexName-<number>结尾，则执行滚动必须指定新索引名字；

示例如下：

```
# my_alias 绑定的索引索引名称不满足 indexName-<number>格式
#必须指定索引 my_new_index_name
POST /my_alias/_rollover/my_new_index_name
{
  "conditions": {
    "max_age":   "7d",
    "max_docs":  2,
    "max_size": "5gb"
  }
}
```

基于 date math 滚动

如果<rollover-target>是索引别名，使用 date math 根据索引滚动的当前日期来命名滚动索引；在某些场景中是非常有用的，比如定时按天删除索引、按天查询数据等。rollover API 支持 date math，但是要求索引名称必须以日期-<number>结尾，

例如：logstash-2021.05.06-1；以这种格式命名的主要是方便在任何时候执行滚动，索引名称自增。

示例如下：

```
# PUT /<logs-{now/d}-1> with URI encoding:
PUT /%3Clogs_%7Bnow%2Fd%7D-1%3E #备注 1
{
  "aliases": {
```

```
      "logs_write": {}
    }
}

PUT logs_write/_doc/1
{
  "message": "a dummy log"
}

POST logs_write/_refresh

# 立即执行 _rollover
POST /logs_write/_rollover #备注 2
{
  "conditions": {
    "max_docs":   1,
    "max_age":    "1d",
    "max_size": "5gb"
  }
}
```

创建一个索引取今天的日期，logs_2021.05.06-1 执行滚动操作，如果立即执行，则新索引名为 logs_2021.05.06-000002;如果等到 24 小时之后执行索引名称为 logs_2021.05.07-000002

你可以按照 date math 文档的说明去使用这些索引。

例如，你需要搜索过去三天索引的数据，你可以按照如下的书写方式：

```
# GET /<logs-{now/d}-*>,<logs-{now/d-1d}-*>,<logs-{now/d-2d}-*>/_search
GET /%3Clogs-%7Bnow%2Fd%7D-*%3E%2C%3Clogs-%7Bnow%2Fd-1d%7D-*%3E%2C%3Clogs-%7Bnow%2Fd-2d%7D-*%3E/_search
```

Dry run

rollover API 支持 dry_run 模式，该模式主要用于条件检测。比如想检测索引是否满足设定的条件完成滚动。

示例如下：

```
POST /logs_write/_rollover?dry_run
{
  "conditions" : {
    "max_age": "7d",
    "max_docs": 1000,
    "max_size": "5gb"
  }
}
```

返回值如下：

```
{
  "acknowledged" : false,
  "shards_acknowledged" : false,
  "old_index" : "logs-000002", #当前索引
  "new_index" : "logs-000003",# 将要新生成的索引
  "rolled_over" : false, #是否完成滚动
  "dry_run" : true,  # 是否为检测模式
  "conditions" : {
    "[max_size: 5gb]" : false,
    "[max_docs: 1000]" : false,
    "[max_age: 7d]" : false
  }
}
```

基于 write index 滚动

基于 write index 滚动主要是为了解决，使用别名查询能够获取到之前创建的索引数据，以及同一别名绑定多个索引滚动切换歧义问题。同一个别名具有查询和写入两种特性。

示例如下:

```
#创建一个索引，绑定别名 logs，并标注 is_write_index 为 true，即通过别名写入数据，实际是写入到 my_logs_index-000001 表中
PUT my_logs_index-000001
{
  "aliases": {
```

```
      "logs": { "is_write_index": true }
    }
  }
  #写入一个文档
  PUT logs/_doc/1
  {
    "message": "a dummy log"
  }
  #刷新，生成一个 segements,使文档变得可搜索
  POST logs/_refresh
  #执行滚动操作
  POST /logs/_rollover #备注 1
  {
    "aliases": {"search_all": {}}, #备注 2
    "conditions": {
      "max_docs":   "1"
    }
  }
  #再次基于别名写入
  PUT logs/_doc/2
  {
    "message": "a newer log"
  }
```

执行滚动操作 write index 切换成为新创建的索引，之后写入的数据将写入到新索引中，滚动时新产生的索引添加别名 search_all

上述代码块执行_rollover 的返回值如下:

```
{
  "_index" : "my_logs_index-000002",
  "_type" : "_doc",
  "_id" : "2",
  "_version" : 1,
  "result" : "created",
  "_shards" : {
    "total" : 2,
    "successful" : 1,
    "failed" : 0
  },
  "_seq_no" : 0,
  "_primary_term" : 1
}
```

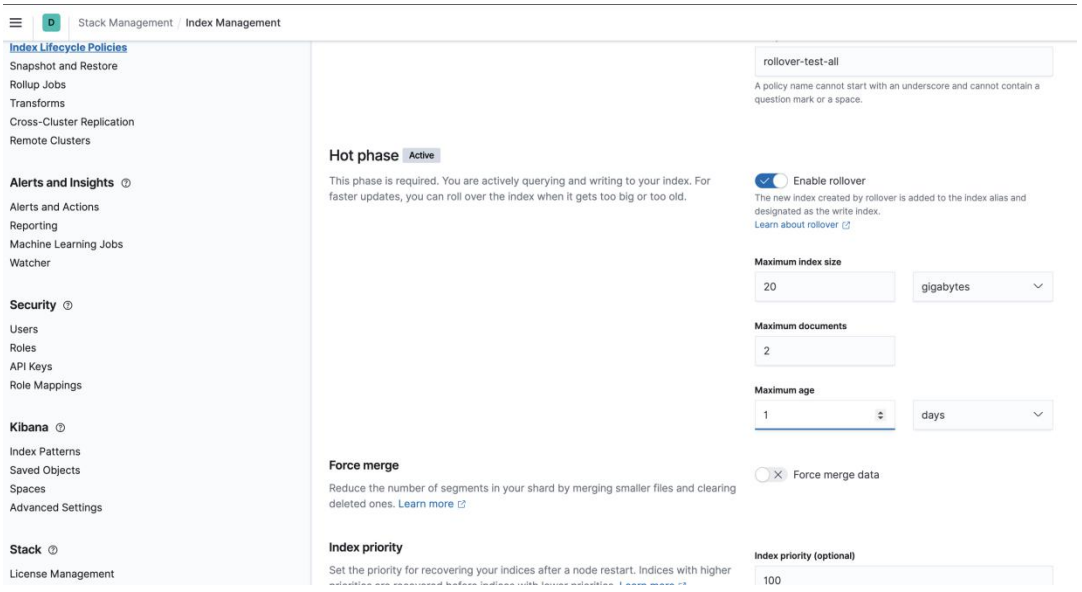
在完成滚动之后，`GET _alias/logs`，查看别名元信息，可以发现新索引已经作为`write index`。

```
{
  "my_logs_index-000002": {
    "aliases": {
      "logs": { "is_write_index": true }
    }
  },
  "my_logs_index-000001": {
    "aliases": {
      "logs": { "is_write_index" : false }
    }
  }
}
```

基于 ILM 实现自动 Rollover

首先在 Kibana ，定义生命周期策略(也可通过 API 方式);

下图规定了 Hot 阶段重新生成索引的规则，这部分功能其实就是对于 rollover API 的封装；下图中我们定义了一个名称为 rollover-test-all 的生命周期管理策略，其中滚动条件为 max_size，为 20gb,max_age 为 1 天，max_docs 为 2；



我们可以通过 API，查看 GET _ilm/policy/rollover-test-all。返回值如下：

```
{
  "rollover-test-all" : {
    "version" : 1,
    "modified_date" : "2021-05-11T15:23:27.155Z",
```

```
    "policy" : {
      "phases" : {
        "hot" : {
          "min_age" : "0ms",
          "actions" : {
            "rollover" : {
              "max_size" : "20gb",
              "max_age" : "1d",
              "max_docs" : 2
            },
            "set_priority" : {
              "priority" : 100
            }
          }
        }
      }
    }
  }
}
```

创建一个模板，模版中引用刚才定义的生命周期策略，并指定滚动的别名。

关于索引模板更多信息请参看 Simulate multi-component templates 。

```
# 这个是 legacy 的 index template，适用于 7.8 版本以前，
# 但在 7.11 版本，仍然可以用
PUT _template/iml-rollover_template
{
```

```
  "index_patterns": [
    "iml-rollover*"
  ],
  "aliases": {
    "iml-rollover_alias": {}
  },
  "settings": {
    "index": {
      "lifecycle": {
        "name": "rollover-test-all",
        "rollover_alias" : "iml-rollover_write_alias"
      },
      "refresh_interval": "2s",
      "number_of_shards": "1",
      "number_of_replicas": "0"
    }
  },
  "mappings": {
    "properties": {
      "name": {
        "type": "keyword"
      }
    }
  }
}

# 7.8 版本之后，可以使用 _index_template 定义模板;
PUT _index_template/iml-rollover_template
{
```

```
  "index_patterns": ["iml-rollover*"],
  "template": {
    "settings": {
      "lifecycle": {
        "name": "rollover-test-all",
        "rollover_alias" : "iml-rollover_write_alias"
      },
      "refresh_interval": "2s",
      "number_of_shards": "1",
      "number_of_replicas": "0"
    },
    "mappings": {
      "_source": {
        "enabled": true
      },
      "properties": {
      "name": {
        "type": "keyword"
      }
    }
    },
    "aliases": {
      "iml-rollover_alias": { }
    }
  },
  "priority": 500,
  "_meta": {
    "description": "my custom rollover test"
  }
}
```

创建索引：

```
PUT iml-rollover-000001
{
  "aliases": {
    "iml-rollover_write_alias": {  "is_write_index": true  }
  }
}
```

插入数据：

```
PUT iml-rollover_write_alias/_bulk?refresh=true
{"index":{}}
{"name":"kimchy"}
{"index":{}}
{"name":"tom"}
```

查看索引情况 `GET _cat/indices/iml-rollover-*?v&h=health,index,docs.count`。

结果如下：

```
health index                docs.count
green  iml-rollover-000001          2
green  iml-rollover-000002          0
```

iml-rollover-000001 的文档个数为 2，并且 iml-rollover-000002 索引已经被创建。查看别名的关联关系 `GET _cat/aliases/iml-rollover_*?format=json`。索引 iml-rollover-000002 的别名 iml-rollover_write_alias 被标记为具有写权限。

```
[
  {
    "alias" : "iml-rollover_alias",
    "index" : "iml-rollover-000001",
    "filter" : "-",
    "routing.index" : "-",
    "routing.search" : "-",
    "is_write_index" : "-"
  },
  {
    "alias" : "iml-rollover_write_alias",
    "index" : "iml-rollover-000001",
    "filter" : "-",
    "routing.index" : "-",
    "routing.search" : "-",
    "is_write_index" : "false"
  },
  {
    "alias" : "iml-rollover_alias",
    "index" : "iml-rollover-000002",
    "filter" : "-",
    "routing.index" : "-",
    "routing.search" : "-",
    "is_write_index" : "-"
```

```
    },
    {
      "alias" : "iml-rollover_write_alias",
      "index" : "iml-rollover-000002",
      "filter" : "-",
      "routing.index" : "-",
      "routing.search" : "-",
      "is_write_index" : "true"
    }
  ]
```

和预期是一致的，正确的完成滚动。这时通过 iml-rollover_write_alias 写入数据，数据被写入到 iml-rollover-000002 索引中。再次执行插入数据语句，然后查看索引文档。

结果如下：

```
health index                docs.count
green  iml-rollover-000001          2
green  iml-rollover-000002          2
```

3.4.2.14 分页搜索

创作人：张超

在查询场景中，从 Elasticsearch 中取得结果，根据不同场景，有多种不同的方式。

- 通过 from 、size 进行分页
- 通过 scroll 拉取大量数据
- 通过 search_after 拉取大量数据

每种方式有其各自的使用场景，或者说他们是为了解决某种场景而设计的。

from + size

搜索引擎的场景，类似 google 搜索，翻页操作一般是人为触发的，并且人的行为一般不会翻页太多，from+size 这种最经典的翻页模式是为了解决用户对于 TopN 的需求，用户希望找到 TopN 个最匹配的文档。其使用方式类似 SQL 中的 LIMIT 关键字，Elasticserach 使用 from 和 size 两个参数来控制翻页：

- size: 要返回的结果数量，默认为 10
- from: 要跳过的结果数量，默认为 0

如果每页显示 5 条结果，下面的命令可以得到 1-3 页的结果：

```
GET /_search?size=5
GET /_search?size=5&from=5
GET /_search?size=5&from=10
```

分页搜索实现方式是：

- 每个分片各自查询的时候先构建 from+size 的优先队列，然后将所有的文档 ID 和排序值返回给协调节点。
- 协调节点创建 size 为 number_of_shards * (from + size) 的优先队列，对数据节点的返回结果进行合并，取全局的 from+size 返回给客户端。

这种工作模式的主要代价在于，协调节点需要等待所有分片返回结果，然后再全局排序。因此会创建非常大的优先队列，需要控制分页的深度，Elasticsearch 默认最多返回 10000 个文档。但是有些时候，用户需要遍历取回所有文档，甚至可以不关心排序。在数据库中取回全部结果可以使用游标查询的方式，类似的概念在 Elasticsearch 中叫做 scroll。

scroll

scroll 可以用于拉取全量数据，他的工作模式不需要像 from + size 一样全局排序，因此没有深分页的代价，例如 reindex 本质上就是 scroll+bulk。使用 scroll 可以简单的在查询语句中添加 scroll 参数：

```
POST /my-index-000001/_search?scroll=1m
```

上面的查询语句会返回一个 ID，后面可以根据这个 ID 顺序拉取结果：

```
POST /my-index-000001/_search/scroll

{
  "scroll" : "1m",
  "scroll_id" : "DXF1ZXJ5QW5kRmV0Y2gBAAAAAAAAAD4WYm9laVYtZndUQlNsdDcwakFMNjU
1QQ=="
}
```

Elasticsearch 将查询后的上下文保存在服务端，因此客户端可依据这个 scroll_id 依次获取后续的数据。

这个上下文也必然有一个生命周期，因为他会一直占用服务端资源。在这个例子中，我们设置 scroll 窗口保存 1 分钟的时间。1 分钟之内，你都可以使用 scroll_id 来拉取数据。当然，你也可以在拉取完成之后根据 scroll_id 手工清理上下文。

search_after

如果说 scroll 是把上下文保存在服务端，而 search_after 要求数据中存在一个无重复，可以用于排序的字段，需要客户端每次传入上次查到的最后结果，然后获取其随后的数据。

由于随后的请求每次都是查询出来的，如果数据发生变化，就可能出现跨页面结果不一致的情况，为了防止这种情况，需要在请求中加一个参数来设置当前的索引状态保留时间。

```
POST /my-index-000001/_pit?keep_alive=1m
```

PIT 是 point in time 的简写，他是一个轻量级的视图。上述请求返回一个 ID：

```
{
  "id": "46ToAwMDaWR5BXV1aWQyKwZub2RlXzMAAAAAAAAAACoBYwADaWR4BXV1aWQxAgZub2RlXzEAAAAAAAAAAAEBYQADaWR5BXV1aWQyKgZub2RlXzIAAAAAAAAAAAwBYgACBXV1aWQyAAAFdXVpZDEAAQltYXRjaF9hbGw_gAAAAA=="
}
```

随后的请求中你需要带上他。可以通过下面的方式，获取第一页的结果，其中的特别之处在于，要指定进行排序的字段：

```
GET /_search
{
  "size": 10000,
  "query": {
    "match" : {
      "user.id" : "elkbee"
    }
  },
  "pit": {
      "id": "46ToAwMDaWR5BXV1aWQyKwZub2RlXzMAAAAAAAAAACoBYwADaWR4BXV1aWQxAgZub2RlXzEAAAAAAAAAAAEBYQADaWR5BXV1aWQyKgZub2RlXzIAAAAAAAAAAAwBYgACBXV1aWQyAAAFdXVpZDEAAQltYXRjaF9hbGw_gAAAAA==",
      "keep_alive": "1m"
  },
```

```
  "sort": [
    {"@timestamp": "asc"}
  ]
}
```

在返回结果中，会携带 sort 字段的值：

```
{
  "hits" : {
    "hits" : [
      {
        "sort" : [
          4294967298
        ]
      }
    ]
  }
}
```

你需要在下次请求的时候带上他：

```
GET /_search
{
  "size": 10000,
  "query": ...
  "pit": {
        "id":  ...
        "keep_alive": ...
```

```
    },
    "sort": [
      {"@timestamp": "asc"}
    ],
    "search_after": [
      4294967298
    ]
  }
```

类似 scroll，pit 请求返回的 ID 也可以手工清理。

最后我们总结一下每种分页方式的特点:

- from+size 支持跳页，不适合深分页；
- scroll 不支持跳页，适合拉取大量数据，不适合大量并发
- search_after 不支持跳页，适合拉取大量数据

Scroll 和 search_after 都可以用于深分页，search_after 需要提供一个主键字段进行排序，默认为 _shard_doc，它是 shard index 与 Lucene 内部 ID 的组合值。在服务端保存的上下文要比 scroll 小，目前官方推荐使用 search_after 代替 scroll。

3.4.2.15 ingest pipelines

创作人：李增胜

Elastic 提供了三种方式进行数据加工处理：Logstash、Beats Processors 以及 Ingest Pipeline，本文着重介绍 Ingest Pipeline，以下比较了 Logstash 与 Ingest Pipeline 的一些区别，便于在实际业务场景中选择：

种类	部署	数据缓冲	数据处理	数据源
Logstash	需要另外部署，增加复杂性	采用队列机制缓冲数据，多队列支持	支持大量 processors,远超 ingest	支持外部数据源，如 MYSQL、Kafka、Beats 等
Ingest pipeline	无需另外部署，易于扩展	无缓冲策略	支持超过 30 种 processors	Ingest 也可和 Beats 或者 Logstash 解决特定场景数据源问题

总结：

- 如果业务场景 Ingest pipeline 已经能处理完成，则无需使用 Logstash ，相反，如果业务处理数据场景要支持外部数据源，则选择 Logstash。
- 如果业务场景需要缓冲数据，则采用 Logstash 较优。
- 如果数据处理完成后需要输出到非 Elasticsearch 内部，则采用 Logstash。

- 在简化配置方便，如果想配置简单，则选择 Elasticsearch ingest pipeline 即可。

显然，Ingest pipeline 并非 Logstatsh 的替代品，需要根据自己的业务处理数据的要求和架构设计来选择对应的技术，并非二选一，也可以同时使用，对处理不同数据采用不同的技术架构。

Kibana Dev Tools 管理 Pipeline

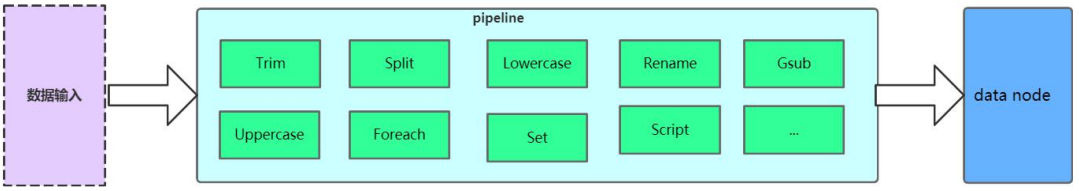
Ingest Pipeline

用于预处理数据，由 Elasticsearch Ingest Node 节点负责运行处理，如需要系统性能提升可单独部署 Ingest Node 节点。

优点：

- 由 Ingest Node 节点负责处理，职责清晰
- 更多 Processors 支持，扩展性强
- 轻量级，覆盖了 Logstash 大多常用场景

Ingest Pipeline 是一系列处理管道，由一系列的 Processors 组成处理，先来看下 pipeline 的处理过程：



在 Kibana 中也可以创建 Ingest pipeline，在稍微章节给出示例。

常用 的 Processors 如下：

更多 Pipeline Processors 参考更多;https://www.elastic.co/guide/en/elasticsearch/reference/master/processors.html

- Trim

  去除空格，如果是字符串类型的数组，数组中所有字符串都会被替换空格处理

- Split

  切分字符串，使用指定切分符，切分字符串为数组结构，只作用与字符串类型

- Rename

  重命名字段

- Foreach

  对一组数据进行相同的预处理，可以使用 Foreach

- Lowercase / Uppercase

  对字段进行大小写转换

- Script

  使用脚本语言进行数据预处理

- Gsub

  对字符串进行替换

- Append

  添加数据到数组

- Set

  设置字段值

- Remove

  移除字段

Trim

去除字符串中的空格：

```
PUT _ingest/pipeline/trim_pipeline
{
  "processors": [
```

```
    {
      "foreach": {
        "field": "message",
        "processor": {
          "trim": {
            "field": "_ingest._value"
          }
        }
      }
    }
  ]
}

POST _ingest/pipeline/trim_pipeline/_simulate
{
  "docs": [
    {
      "_source": {
        "message": [
          "car222 ",
          " auto2222 "
        ]
      }
    }
  ]
}

#返回:
{
  "docs" : [
    {
```

```
        "doc" : {
          "_index" : "_index",
          "_type" : "_doc",
          "_id" : "_id",
          "_source" : {
            "message" : [
              "car222",
              "auto2222"
            ]
          },
          "_ingest" : {
            "_value" : null,
            "timestamp" : "2021-04-28T13:19:13.542743Z"
          }
        }
      }
    ]
}
```

Split / Foreach

切分字符串，使用指定切分符，切分字符串为数组结构，只作用于字符串类型：

```
PUT _ingest/pipeline/split_pipeline
{
  "processors": [
    {
      "foreach": {
        "field": "message",
        "processor": {
```

```
            "split": {
              "field": "_ingest._value",
              "separator": " "
            }
          }
        }
      }
    ]
}

#测试
POST _ingest/pipeline/split_pipeline/_simulate
{
  "docs": [
    {
      "_source": {
        "message": [
          "car222 aaa",
          " auto2222 aaaa bbb"
        ]
      }
    }
  ]
}
#返回，可以看到 message 按照空格切分为了多个字符串数组
{
  "docs" : [
    {
      "doc" : {
        "_index" : "_index",
        "_type" : "_doc",
```

```
            "_id" : "_id",
            "_source" : {
              "message" : [
                [
                  "car222",
                  "aaa"
                ],
                [
                  "",
                  "auto2222",
                  "aaaa",
                  "bbb"
                ]
              ]
            },
            "_ingest" : {
              "_value" : null,
              "timestamp" : "2021-04-28T13:28:20.762312Z"
            }
          }
        }
      ]
}
```

Rename

重命名一个字段，rename 往往和 reindex 结合使用：

```
POST goods_info_comment_message/_bulk
{"index":{"_id":1}}
{"message":"美 国苹果 "}
{"index":{"_id":2}}
{"message":"山东 苹果 "}


#定义 rename_pipeline
PUT _ingest/pipeline/rename_pipeline
{
  "processors": [
    {
      "rename": {
        "field": "message",
        "target_field": "message_new"
      }
    }
  ]
}

#重建 index
POST _reindex
{
  "source": {
    "index": "goods_info_comment_message"
  },
  "dest": {
    "index": "goods_info_comment_message_new",
    "pipeline": "rename_pipeline"
  }
}
```

```
#查询 mapping
GET goods_info_comment_message_new/_mapping

#返回
{
  "goods_info_comment_message_new" : {
    "mappings" : {
      "properties" : {
        "message_new" : {
          "type" : "text",
          "fields" : {
            "keyword" : {
              "type" : "keyword",
              "ignore_above" : 256
            }
          }
        }
      }
    }
  }
}
```

Lowercase / Uppercase

将字符串修改为大写或者小写：

```
PUT _ingest/pipeline/lowercase_pipeline
{
  "description": "lowercase processor",
```

```
  "processors": [
    {
      "lowercase": {
        "field": "message"
      }
    }
  ]
}

#测试，部分字符大写
POST _ingest/pipeline/lowercase_pipeline/_simulate
{
  "docs": [
    {
      "_source": {
        "message": [
          "CAr222 aaa",
          " auto2222 aaaa Bbb"
        ]
      }
    }
  ]
}

#结果，全部输出为小写
{
  "docs" : [
    {
      "doc" : {
        "_index" : "_index",
        "_type" : "_doc",
```

```
        "_id" : "_id",
        "_source" : {
          "message" : [
            "car222 aaa",
            " auto2222 aaaa bbb"
          ]
        },
        "_ingest" : {
          "timestamp" : "2021-04-28T15:12:10.041308Z"
        }
      }
    }
  ]
}
```

Remove

移除已经存在的字段

```
#定义 remove pipelint
PUT _ingest/pipeline/remove_pipeline
{
  "description": "remove processor",
  "processors": [
    {
      "remove": {
        "field": "message"
      }
    }
  ]
}

#测试
POST _ingest/pipeline/remove_pipeline/_simulate
```

```
{
  "docs": [
    {
      "_source": {
        "message": [
          "CAr222 aaa",
          " auto2222 aaaa Bbb"
        ]
      }
    }
  ]
}
#返回，可以看到 message 字段已经被移除
{
  "docs" : [
    {
      "doc" : {
        "_index" : "_index",
        "_type" : "_doc",
        "_id" : "_id",
        "_source" : { },
        "_ingest" : {
          "timestamp" : "2021-04-28T15:15:27.811516Z"
        }
      }
    }
  ]
}
```

Set

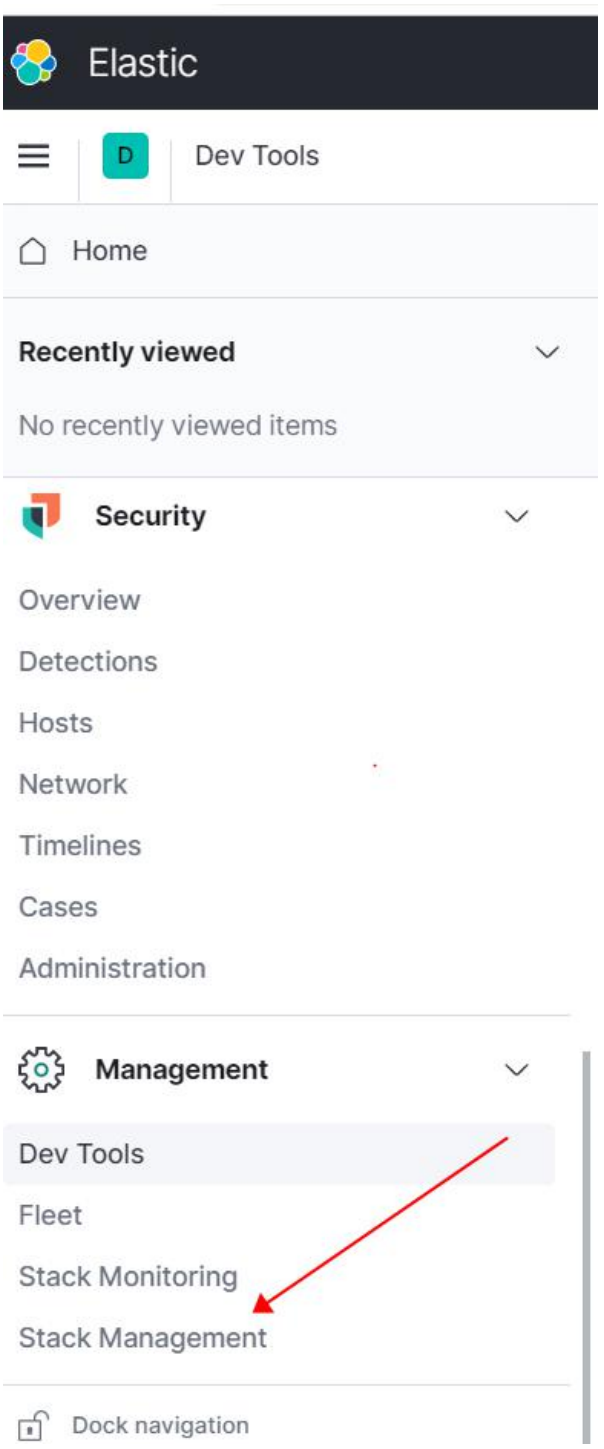
给已有字段进行赋值：

```
PUT _ingest/pipeline/set_pipeline
{
  "description": "set processor",
  "processors": [
    {
      "set": {
        "field": "message",
        "value": "this is a new message"
      }
    }
  ]
}


POST _ingest/pipeline/set_pipeline/_simulate
{
  "docs": [
    {
      "_source": {
        "message": "this"
      }
    }
  ]
}

#返回
{
  "docs" : [
```

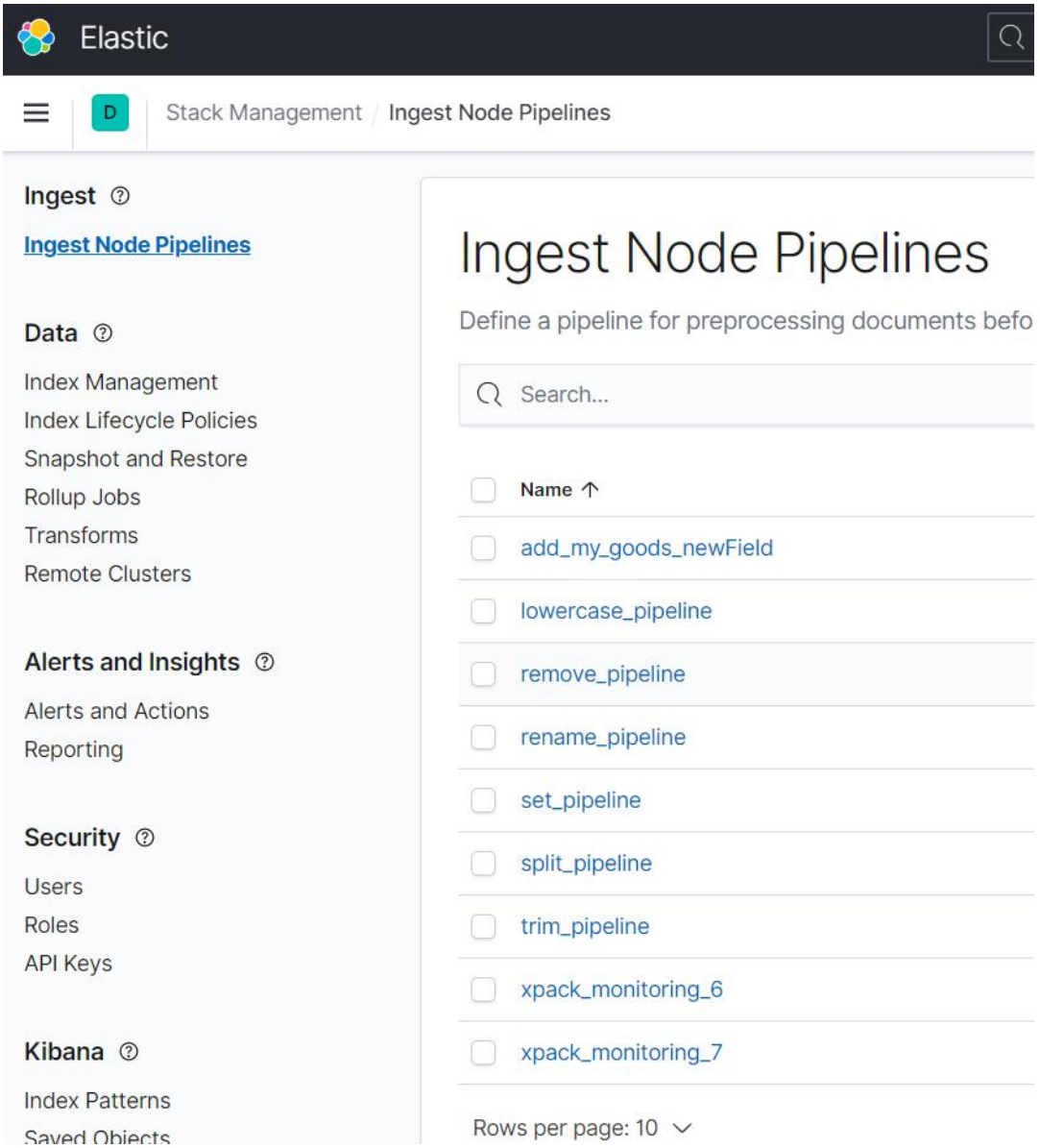
```
    {
      "doc" : {
        "_index" : "_index",
        "_type" : "_doc",
        "_id" : "_id",
        "_source" : {
          "message" : "this is a new message"
        },
        "_ingest" : {
          "timestamp" : "2021-04-28T15:21:28.928512Z"
        }
      }
    }
  ]
}
```

Kibana Dev Tools 管理 Pipeline

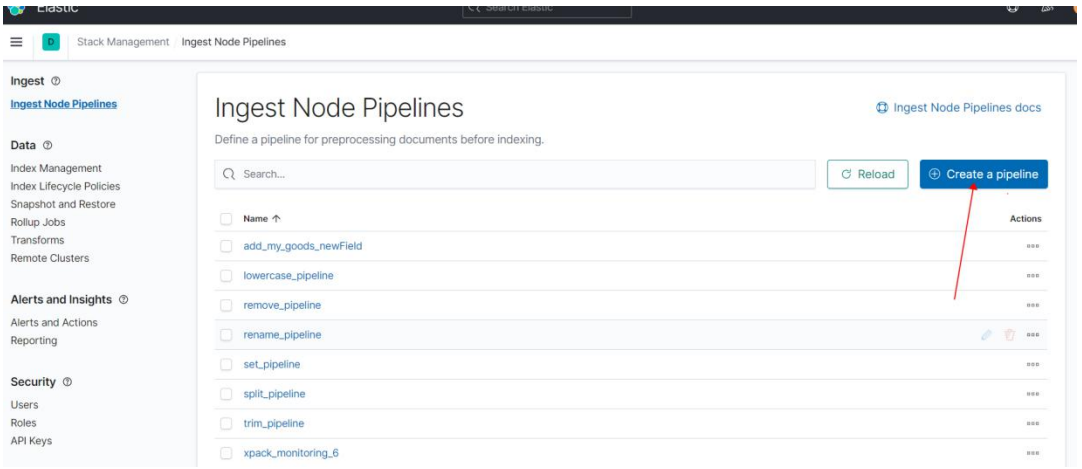
下面介绍如何在 kibana 中通过界面来创建 Pipeline，打开 Kibana 首页：

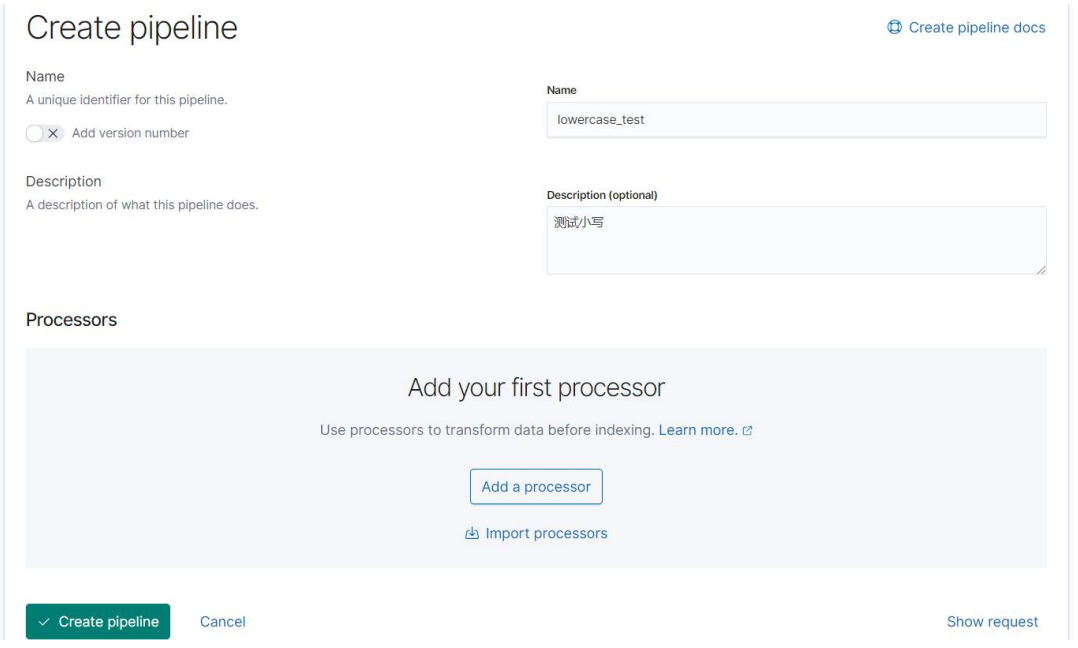

Elastic
D
Dev Tools
Home
Recently viewed
No recently viewed items
Security
Overview
Detections
Hosts
Network
Timelines
Cases
Administration
Management
Dev Tools
Fleet
Stack Monitoring
Stack Management
Dock navigation

选择 Ingest Node Pipelines，右边会展示已有的 Pipeline 列表：

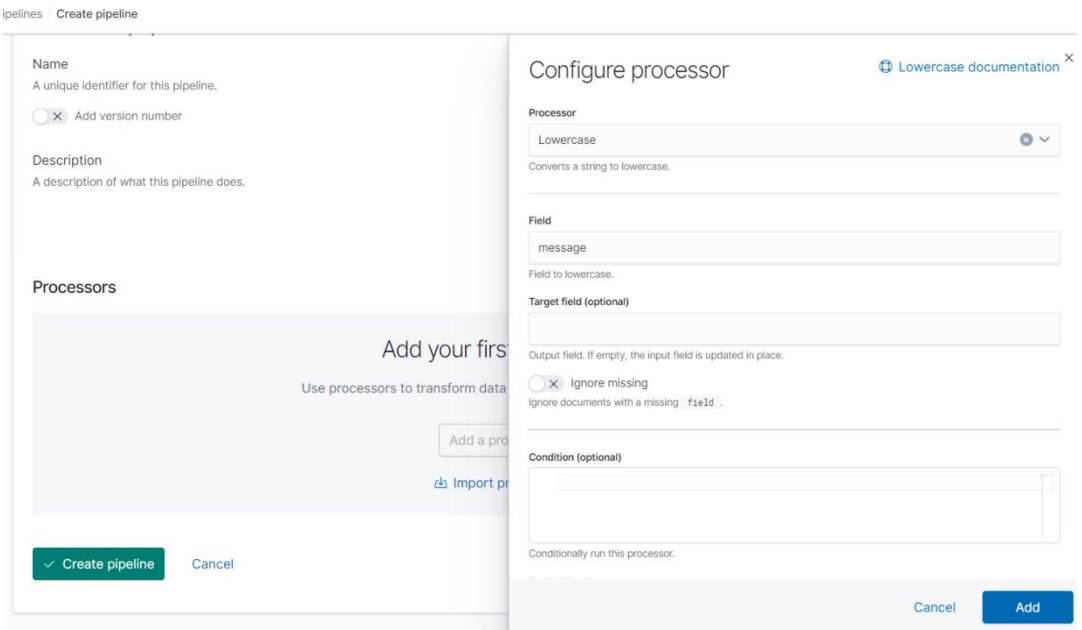


选择新创建 Ppipeline：

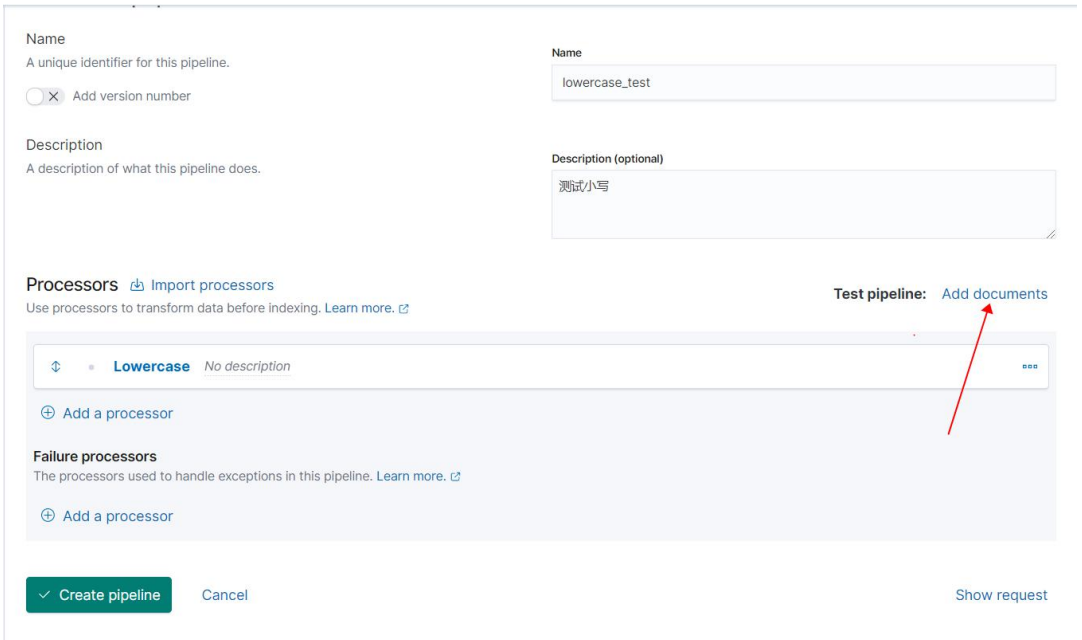




我们选择创建一个 lowercase processor：

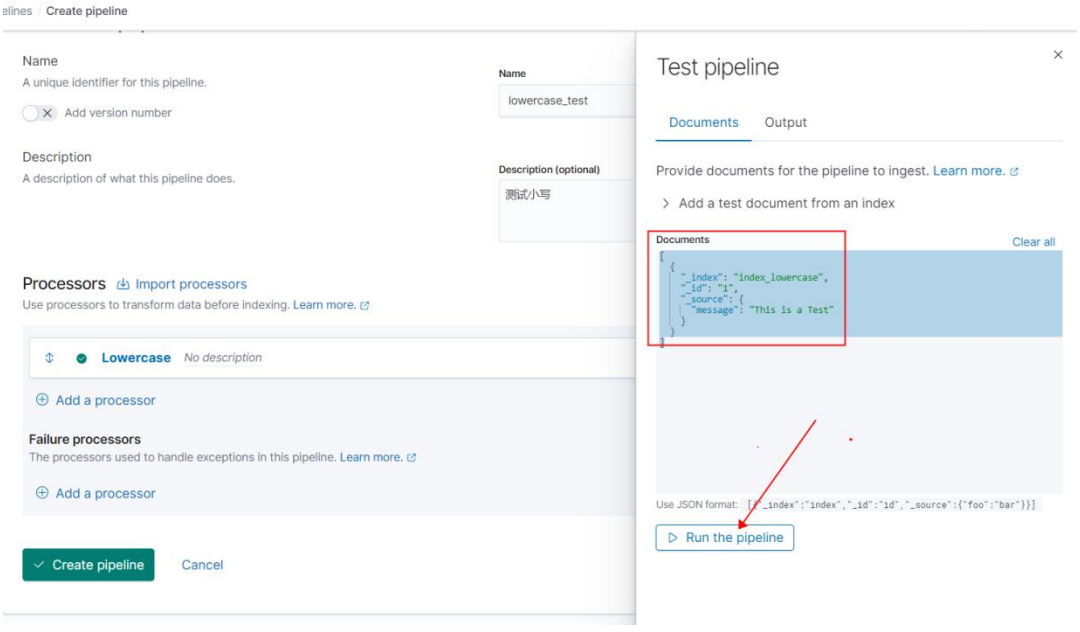


点击 Add documents 进行相关测试：

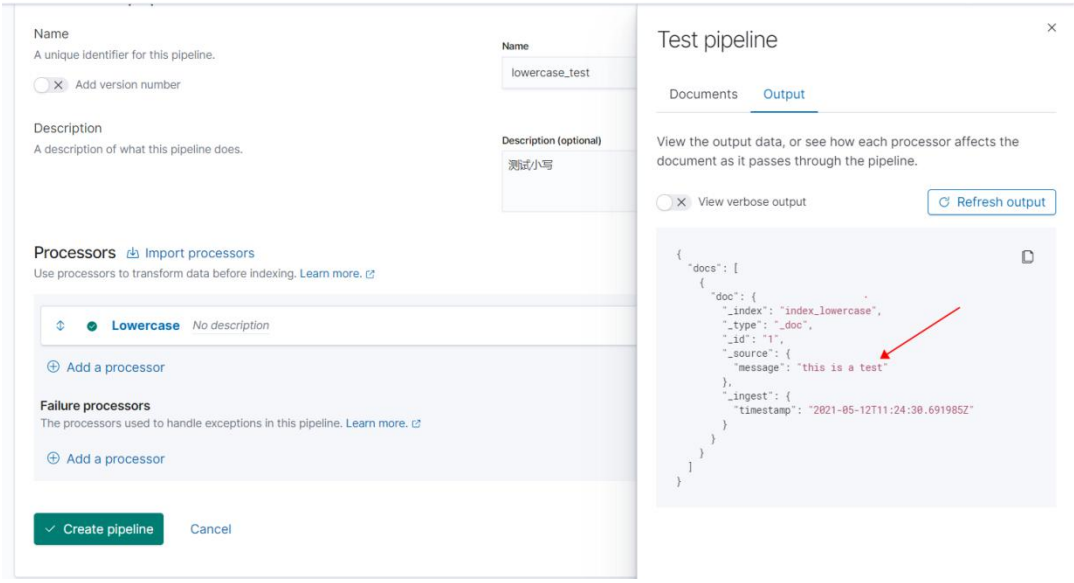


添加测试文档：

```
[
  {
    "_index": "index_lowercase",
    "_id": "1",
    "_source": {
      "message": "This is a Test"
    }
  }
]
```



可以看到，测试成功，字符串全部变为了小写：


Name
A unique identifier for this pipeline.
Add version number
Description
A description of what this pipeline does.
Name
lowercase_test
Description (optional)
测试小写
Processors
Import processors
Use processors to transform data before indexing. Learn more.
Lowercase
No description
Add a processor
Failure processors
The processors used to handle exceptions in this pipeline. Learn more.
Add a processor
Create pipeline
Cancel
Test pipeline
Documents
Output
View the output data, or see how each processor affects the document as it passes through the pipeline.
View verbose output
Refresh output
{
"docs": [
{
"doc": {
"_index": "index_lowercase",
"_type": "_doc",
"_id": "1",
"_source": {
"message": "this is a test"
},
"_ingest": {
"timestamp": "2021-05-12T11:24:30.691985Z"
}
}
}
]
}

3.4.2.16 Painless scripting

创作人：李增胜

Painless scripting 是一种简单的、安全的针对 Elasticsearch 设计的脚本语言，Painless 可以使用在任何可以使用 scripting 的场景。脚本提供了以下优点：

- 更高的性能，scripting 脚本比其他的可选脚本快数倍。
- 安全性高，更小颗粒度的字段授权机制，避免可能不必要的安全隐患安全。
- 可选类型，变量和参数可以使用显示或者动态类型编程方式。
- 语法，扩展 Java 的语法并兼容了其他脚本。
- 优化，专为 ES 设计的脚本语言。

常用关键字：

if、else、while、do、for、in，continue，break，return，new、try、catch、throw、this、instanceof。

常用举例

首先我们创建测试数据，商品信息。

```
#添加测试数据
POST my_goods/_bulk
{"index":{"_id":1}}
{"goodsName":"苹果 51英寸 4K超高清","skuCode":"skuCode1","brandName":"苹果","closeUserCode":["0"],"channelType":"cloudPlatform","shopCode":"sc00001","publicPrice":8188.88,"groupPrice":null,"boxPrice":null,"boostValue":1.8}
{"index":{"_id":2}}
{"goodsName":"苹果 55英寸 3K超高清","skuCode":"skuCode2","brandName":"苹果","closeUserCode":["0"],"channelType":"cloudPlatform","shopCode":"sc00002","publicPrice":6188.88,"groupPrice":null,"boxPrice":null,"boostValue":1.0}
{"index":{"_id":3}}
{"goodsName":"苹果 UA55RU7520JXXZ 53英寸 4K高清","skuCode":"skuCode3","brandName":"美国苹果","closeUserCode":["0"],"channelType":"cloudPlatform","shopCode":"sc00001","publicPrice":8388.88,"groupPrice":null,"boxPrice":[{"boxType":"box1","boxUserCode":["htd003","uc004"],"boxPriceDetail":4388.88},{"boxType":"box2","boxUserCode":["uc005","uc0010"],"boxPriceDetail":5388.88}],"boostValue":1.2}
{"index":{"_id":4}}
{"goodsName":"山东苹果 UA55RU7520JXXZ 苹果 54 英寸 5K 超高清","skuCode":"skuCode4","brandName":"山东苹果","closeUserCode":["uc001","uc002","uc003"],"channelType":"cloudPlatform","shopCode":"sc00001","publicPrice":8488.88,"groupPrice":[{"level":"level1","boxLevelPrice":"2488.88"},{"level":"level2","boxLevelPrice":"3488.88"}],"boxPrice":[{"boxType":"box1","boxUserCode":["uc004","uc005","uc006","uc001"],"boxPriceDetail":4488.88},{"boxType":"box2","boxUserCode":["htd007","htd008","htd009","uc0010"],"boxPriceDetail":5488.88}],"boostValue":1.2}
```

Inline script

少量代码跟随其他 DSL 一起执行的脚本，在下面的例子用会说明具体案例。

添加字段

如果我们想添加一个新字段，而新字段又依赖已有字段，如下所示，我们添加一个新品牌，品牌的名称为原有品牌的基础上拼接“新品”，就可以使用脚本来实现此业务。

```
POST my_goods/_update_by_query
{
  "script": {
    "source": "ctx._source.new_brandName = ctx._source.brandName + '新品'"
  }
}

#查询结果
GET my_goods/_search

#返回（省略部分无关字段）
"hits" : [
      {
        "_index" : "my_goods",
        "_source" : {
          "shopCode" : "sc00001",
          "new_brandName" : "苹果新品",
          "brandName" : "苹果",
          "closeUserCode" : [
            "0"
          ]
        }
      },
      {
```

```
            "_index" : "my_goods",
            "_type" : "_doc",
            "_id" : "2",
            "_score" : 1.0,
            "_source" : {
              "shopCode" : "sc00002",
              "new_brandName" : "苹果新品",
              "brandName" : "苹果",
              "closeUserCode" : [
                "0"
              ],
              "groupPrice" : null,
              "boxPrice" : null,
              "channelType" : "cloudPlatform",
              "boostValue" : 1.0,
              "publicPrice" : "6188.88",
              "goodsName" : "苹果 55 英寸 3K 超高清",
              "skuCode" : "skuCode2"
            }
          },
          ....
        ]

    #可以看到使用脚本新增的字段 new_brandName 已经生效
```

上面的 source 表示我们使用了 Painless 脚本代码，这种使用少量代码在 DSL 中的 Painless 脚本称为 Inline script 。

删除字段

当我们需要删除已有字段时，可以通过脚本来删除：

```
POST my_goods/_update_by_query
{
  "script": {
    "source": "ctx._source.remove('new_brandName')"
  }
}
```

更改字段值

在更改字段值时，我们使用了 params 参数的形式进行处理，使用 params 有一定优点，当脚本中 source 值一样时，ES 会视为同一个脚本，会进行缓存不需要重新编译，可以加快处理速度，在下次使用时可以拿出来直接使用而不用经过编译。

```
#性能较差，硬编码实现价格提升 2 倍
POST my_goods/_update/1
{
  "script": {
    "source": "ctx._source.publicPrice = ctx._source.publicPrice * 2",
    "lang": "painless"
  }
}

#性能较优，使用 params 将 ID 为 1 的商品的价格提高 2 倍
POST my_goods/_update/1
{
```

```
    "script": {
      "source": "ctx._source.publicPrice = ctx._source.publicPrice * params.promote_percen
t",
      "lang": "painless",
      "params": {
        "promote_percent": 2
      }
    }
  }

  #查询
  GET my_goods/_doc/1

  #返回
  {
    "_index" : "my_goods",
    "_type" : "_doc",
    "_id" : "1",
    "_version" : 2,
    "_seq_no" : 4,
    "_primary_term" : 1,
    "found" : true,
    "_source" : {
      "goodsName" : "苹果 51 英寸 4K 超高清",
      "skuCode" : "skuCode1",
      "brandName" : "苹果",
      "closeUserCode" : [
        "0"
      ],
      "channelType" : "cloudPlatform",
      "shopCode" : "sc00001",
```

```
        "publicPrice" : 16377.76,
        "groupPrice" : null,
        "boxPrice" : null,
        "boostValue" : 1.8
      }
    }
    #可以看到，在更新前价格为“8188.88”，通过脚本更新后价格变为 16377.76
```

在 Elasticsearch 中，以下的脚本会视为一个脚本：

```
"source": "ctx._source.publicPrice = ctx._source.publicPrice * params.promote_percent"
```

下面的会被认为是 2 个不同的脚本，运行时每次都需要编译，性能比上面使用 params 稍差：

```
"source": "ctx._source.publicPrice = ctx._source.publicPrice * 2"
"source": "ctx._source.publicPrice = ctx._source.publicPrice * 3"
```

排序

```
#修改 goodsName 可以被 doc 访问
PUT my_goods/_mapping
{
  "properties": {
    "goodsName":{
      "type":"text",
      "fielddata": "true"
    }
```

```
    }
  }
  #查询并排序，根据商品名称长度并添加干扰因子 1.1 倍为最终排序结果
  POST my_goods/_search
  {
    "query": {
      "match": {
        "brandName": "苹果"
      }
    },
    "sort": {
      "_script": {
        "type": "number",
        "script": {
          "lang": "painless",
          "source": "doc['goodsName'].value.length() * params.factor",
          "params": {
            "factor": 1.1
          }
        },
        "order": "asc"
      }
    }
  }
```

Stored script

先将脚本存储，在 DSL 查询时使用已经存储更好的脚本，叫做 stored script：

```
#定义 stored script,脚本名称为：promote_price
PUT _scripts/promote_price
{
  "script": {
    "source": "ctx._source.publicPrice = ctx._source.publicPrice * params.value",
    "lang": "painless"
  }
}
```

如上代码所示，我们定义了一个名称为 promote_price 的脚本，作用就是提升售卖价格（publicPrice）一定的倍数，这个倍数是在调用时传入的。

```
POST my_goods/_update_by_query
{
  "script": {
    "id": "promote_price",
    "params": {
      "value": 2
    }
  }
}
```

执行 stored script，将会看到价格提升了 2 倍。

Source 里字段访问

在使用 Painless 访问 Source 里的字段值时，需要根据运行时的上下文来确定使用的语法，Painless 常见的上下文有：update 、update_by_query、sort、ingest pipeline 等。

Context	访问字段
update	ctx._source.field_name
ingest node	ctx.field_name

分别举例使用 _source 与 ctx 来操作字段的值。

update

```
# 在上面的例子中，就曾使用过 ctx._source.field_name 来更新数据
POST my_goods/_update/1
{
  "script": {
    "source": "ctx._source.publicPrice = ctx._source.publicPrice * params.promote_percen
t",
    "lang": "painless",
    "params": {
      "promote_percent": 2
    }
  }
}
```

ingest node

在 ingest pipeline 中更新字段值：

```
#定义 pipeline
PUT _ingest/pipeline/add_my_goods_newField
{
  "processors": [
    {
      "script": {
        "lang": "painless",
        "source": "ctx.skuCode_brandName = ctx.skuCode + ctx.brandName"
      }
    }
  ]
}

#执行 pipeline
POST my_goods/_update_by_query?pipeline=add_my_goods_newField
{

}

#查询结果
GET my_goods/_search

#返回
"hits" : [
      {
        "_index" : "my_goods",
```

```
      "_type" : "_doc",
      "_id" : "2",
      "_score" : 1.0,
      "_source" : {
        "shopCode" : "sc00002",
        "brandName" : "苹果",
        "closeUserCode" : [
          "0"
        ],
        "skuCode_brandName" : "skuCode2 苹果",
        "channelType" : "cloudPlatform",
        "publicPrice" : 12377.76,
        "goodsName_length" : 13,
        "groupPrice" : null,
        "boxPrice" : null,
        "boostValue" : 1.0,
        "goodsName" : "苹果 55 英寸 3K 超高清",
        "skuCode" : "skuCode2"
      }
    },
    {
      "_index" : "my_goods",
      "_type" : "_doc",
      "_id" : "3",
      "_score" : 1.0,
      "_source" : {
        "shopCode" : "sc00001",
        "brandName" : "美国苹果",
        "closeUserCode" : [
          "0"
        ],
```

```
            "skuCode_brandName" : "skuCode3 美国苹果",
            "channelType" : "cloudPlatform",
            "publicPrice" : 16777.76,
            "goodsName_length" : 26,
            "groupPrice" : null,
            "boxPrice" : [
              {
                "boxType" : "box1",
                "boxUserCode" : [
                  "htd003",
                  "uc004"
                ],
                "boxPriceDetail" : 4388.88
              },
              {
                "boxType" : "box2",
                "boxUserCode" : [
                  "uc005",
                  "uc0010"
                ],
                "boxPriceDetail" : 5388.88
              }
            ],
            "boostValue" : 1.2,
            "goodsName" : "苹果 UA55RU7520JXXZ 53 英寸 4K 高清",
            "skuCode" : "skuCode3"
          }
        },
    ....
]
```

可以看到 ，skuCode_brandName 是通过 skuCode+brandName 拼接成功的，通过 ctx.field 访问字段成功。

Painless Debug

Elasticsearch 中为我们提供了脚本调试方法，使我们在使用时可以方便的进行脚本调试：

```
#定义用户信息，shop_id 为用户开的店铺 ID 信息
PUT /user_info/_doc/1?refresh
{
  "first": "Michael",
  "last": "Jordan",
  "shop_id": [
    100,
    102,
    103
  ],
  "time": "2021-05-09"
}

PUT /user_info/_doc/2?refresh
{
  "first": "Michael2",
  "last": "Jordan2",
  "shop_id": [
    110,
    112,
```

```
      113,
      114,
      115
    ],
    "time": "2021-05-08"
}


#查看 mapping
GET  user_info/_mapping

#返回

{
  "user_info" : {
    "mappings" : {
      "properties" : {
        "first" : {
          "type" : "text",
          "fields" : {
            "keyword" : {
              "type" : "keyword",
              "ignore_above" : 256
            }
          }
        },
        "last" : {
          "type" : "text",
          "fields" : {
            "keyword" : {
              "type" : "keyword",
```

```
                "ignore_above" : 256
              }
            }
          },
          "shop_id" : {
            "type" : "long"
          },
          "time" : {
            "type" : "date"
          }
        }
      }
    }
}
```

可以看到返回了很多字段类型，包括：long、date、keyword、text，每种类型有哪些方法可以操作呢？一种是查看官网文档，另外一种获取使用的方法就是通过调试来获取信息了，使用_explain 来看看效果：

```
POST /user_info/_explain/1
{
  "query": {
    "script": {
      "script": "Debug.explain(doc.shop_id)"
    }
  }
}

#返回:
{
```

```
"error": {
  "root_cause": [
    {
      "type": "script_exception",
      "reason": "runtime error",
      "painless_class": "org.elasticsearch.index.fielddata.ScriptDocValues.Longs",
      "to_string": "[100, 102, 103]",
      "java_class": "org.elasticsearch.index.fielddata.ScriptDocValues$Longs",
      "script_stack": [
        "Debug.explain(doc.shop_id)",
        "                       ^---- HERE"
      ],
      "script": "Debug.explain(doc.shop_id)",
      "lang": "painless",
      "position": {
        "offset": 17,
        "start": 0,
        "end": 26
      }
    }
  ],
  "type": "script_exception",
  "reason": "runtime error",
  "painless_class": "org.elasticsearch.index.fielddata.ScriptDocValues.Longs",
  "to_string": "[100, 102, 103]",
  "java_class": "org.elasticsearch.index.fielddata.ScriptDocValues$Longs",
  "script_stack": [
    "Debug.explain(doc.shop_id)",
    "                       ^---- HERE"
  ],
  "script": "Debug.explain(doc.shop_id)",
```

```
        "lang": "painless",
        "position": {
          "offset": 17,
          "start": 0,
          "end": 26
        },
        "caused_by": {
          "type": "painless_explain_error",
          "reason": null
        }
      },
      "status": 400
    }
```

可以看到是一个 runtime error 异常，那我们应该如何解决呢？

仔细观察，doc.shop_id 是这样的类提供支撑：

```
"painless_class": "org.elasticsearch.index.fielddata.ScriptDocValues.Longs"
"java_class": "org.elasticsearch.index.fielddata.ScriptDocValues$Longs"
```

通过 Painless Script 的 API 帮助：https://www.elastic.co/guide/en/elasticsearch/painless/7.10/painless-api-reference.html，

最终找到 Long 类型的 API 文档地址：https://www.elastic.co/guide/en/elasticsearch/painless/7.10/painless-api-reference-shared-org-elasticsearch-index-fielddata.html#painless-api-reference-shared-ScriptDocValues-Longs

ScriptDocValues.Longs

- List asList()
- int getLength()
- Collection asCollection()
- Long get(int)
- .....

我们通过观察数据知道 shop_id 存储的是一个 list 数据，加入我们要获取第一个数据，再次调整脚本：

```
GET user_info/_search
{
  "query": {
    "function_score": {
      "script_score": {
        "script": {
          "lang": "painless",
          "source": """
                return doc['shop_id'].getLength();
          """
        }
      }
    }
  }
}

#返回:
```

```
{
  "took" : 1,
  "timed_out" : false,
  "_shards" : {
    "total" : 1,
    "successful" : 1,
    "skipped" : 0,
    "failed" : 0
  },
  "hits" : {
    "total" : {
      "value" : 2,
      "relation" : "eq"
    },
    "max_score" : 5.0,
    "hits" : [
      {
        "_index" : "user_info",
        "_type" : "_doc",
        "_id" : "2",
        "_score" : 5.0,
        "_source" : {
          "first" : "Michael2",
          "last" : "Jordan2",
          "shop_id" : [
            110,
            112,
            113,
            114,
            115
```

```
            ],
            "time" : "2021-05-08"
          }
        },
        {
          "_index" : "user_info",
          "_type" : "_doc",
          "_id" : "1",
          "_score" : 3.0,
          "_source" : {
            "first" : "Michael",
            "last" : "Jordan",
            "shop_id" : [
              100,
              102,
              103
            ],
            "time" : "2021-05-09"
          }
        }
      ]
    }
  }
```

可以看到，得分最高的为 "max_score" : 5.0, 因为我们使用 script_score 调整了评分，以店铺 ID 个数为评分结果，文档 2 共计 5 个 ID，所以返回的是 5 。

通过以上案例，详细解读了 Painless Debug 在实际场景中的应用，通过一步步分析最终掌握了调试、看错误信息、找官方文档解决的方法，最终实现了掌握 Painless Debug 的目的。

3.4.3 Kibana 基础应用

创作人：郭海亮

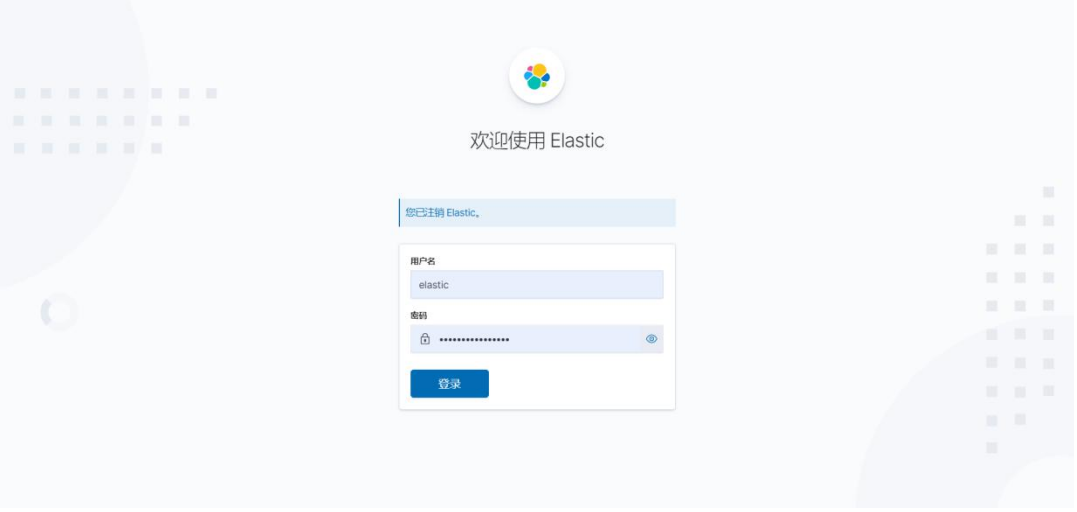
审稿人：杨振涛

查看样例数据

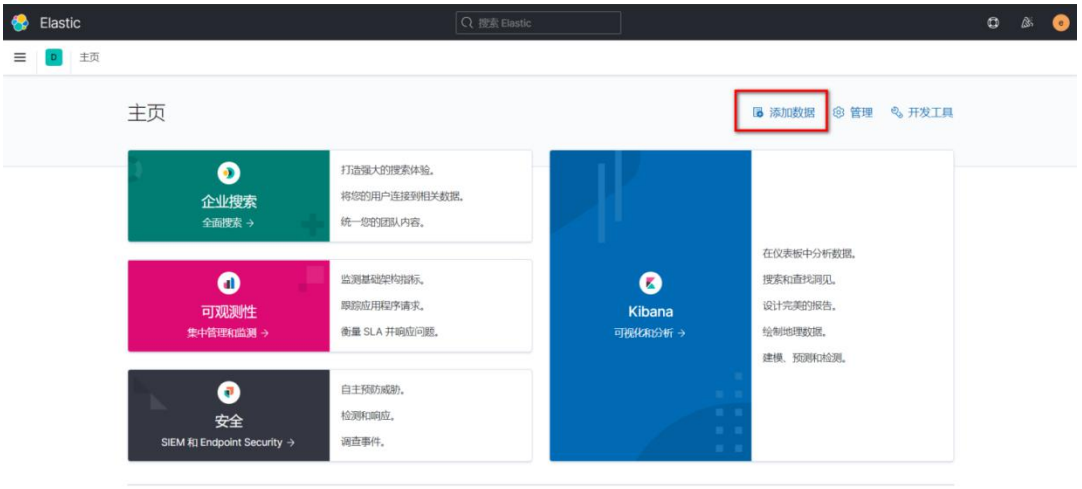
登录 Kibana

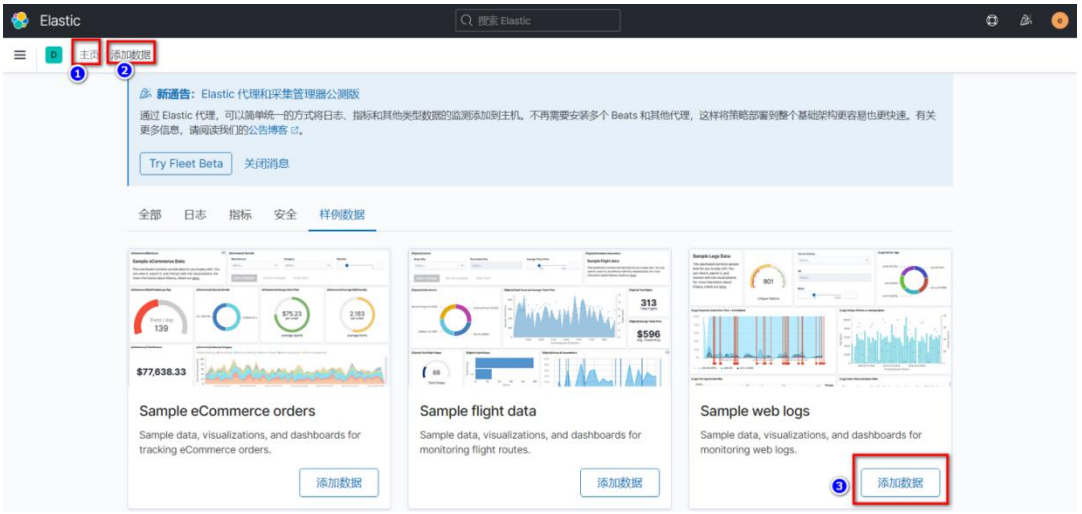
浏览器输入：http://localhost:5601

地址格式为：http://kibana 地址:kibana 所使用的端口，均为在 kibana.yml 配置文件中定义的。

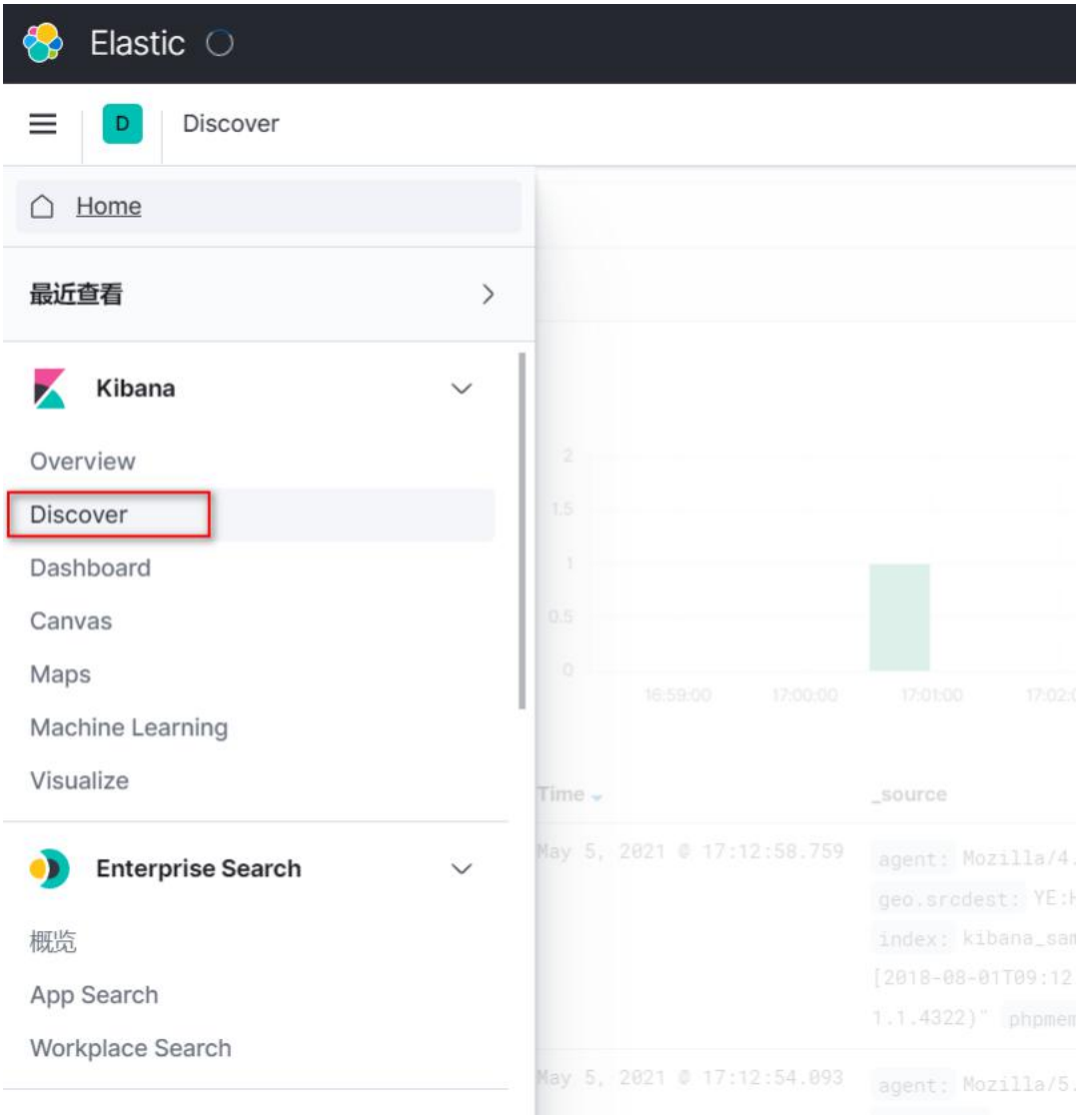


导入样例数据：

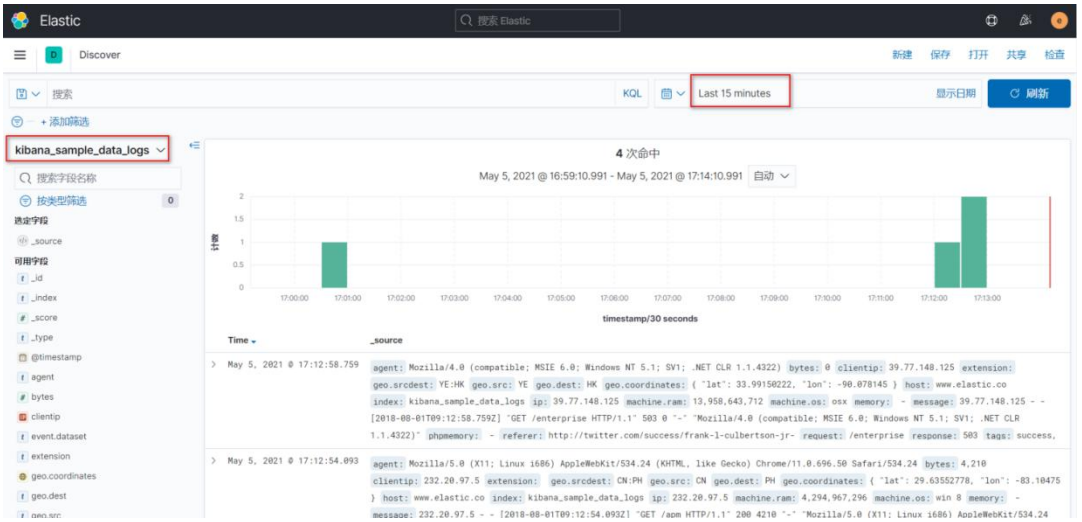




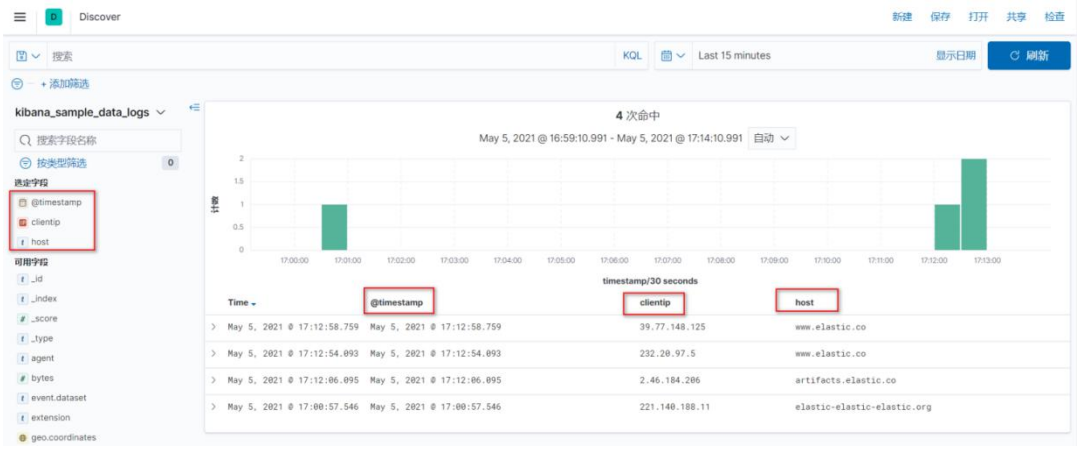
查看样例数据：



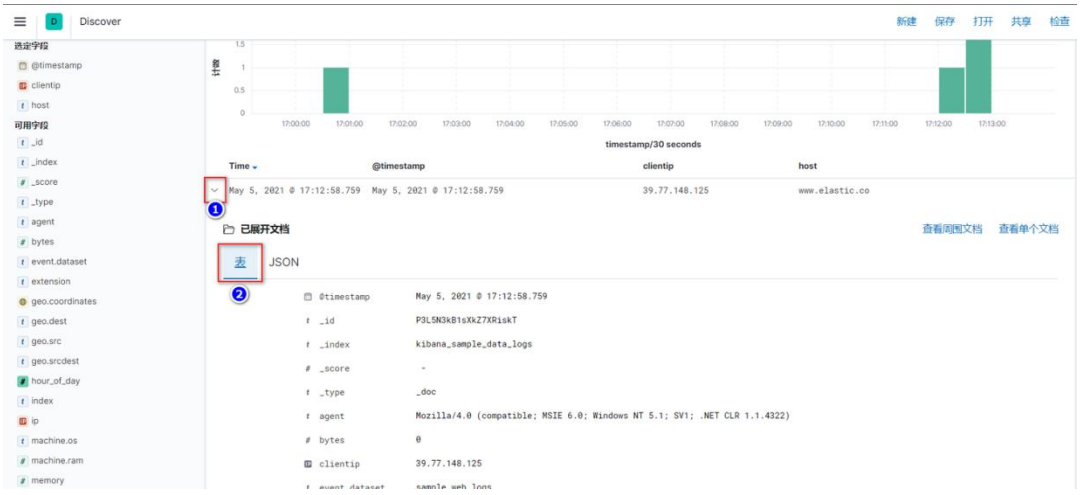
在左上方选择样例数据的索引，右上方选择需要查询的时间范围，即可看到我们需要的数据，如下：

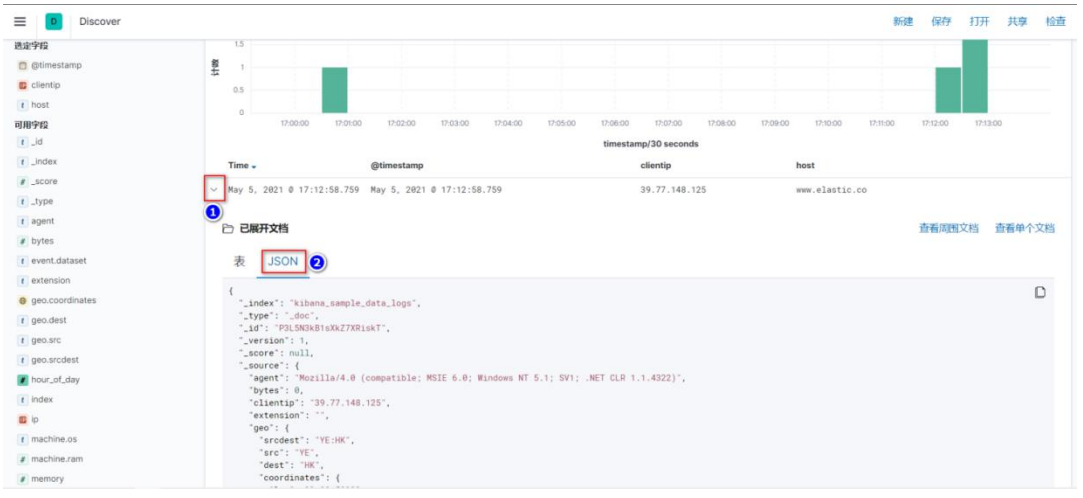


默认会展示所有字段，当然，也可以在左侧栏选择需要展示的字段，如下：

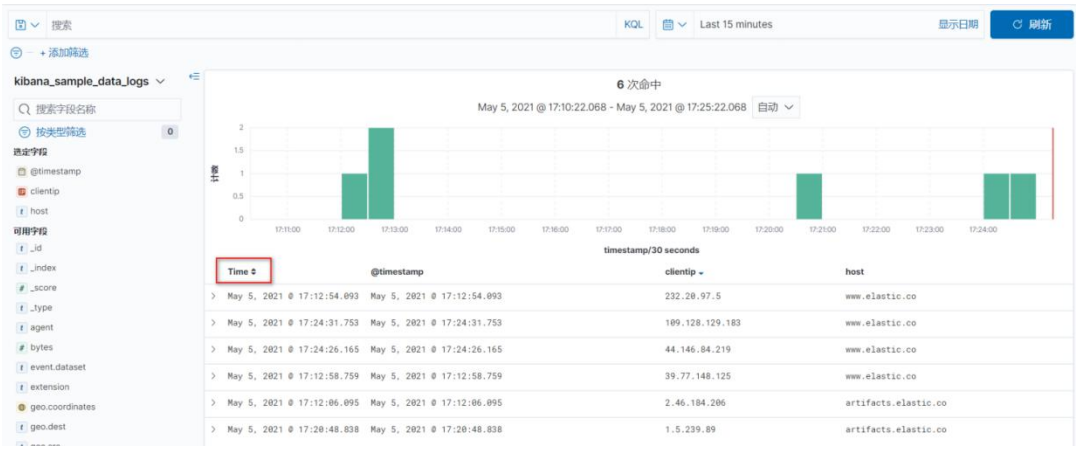


这些数据也可以以表或者以 JSON 形式展示，如下：

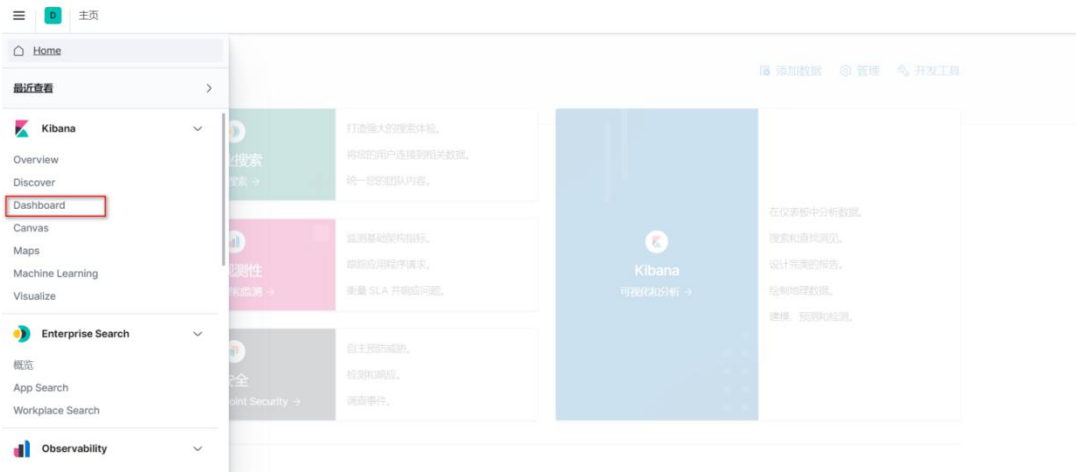


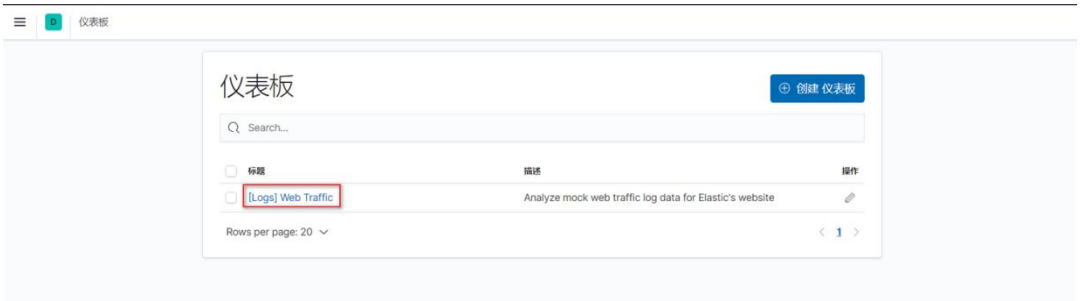


将鼠标移动到某个字段上，会出现上下的箭头，此时便可以根据箭头进行排序，如下：

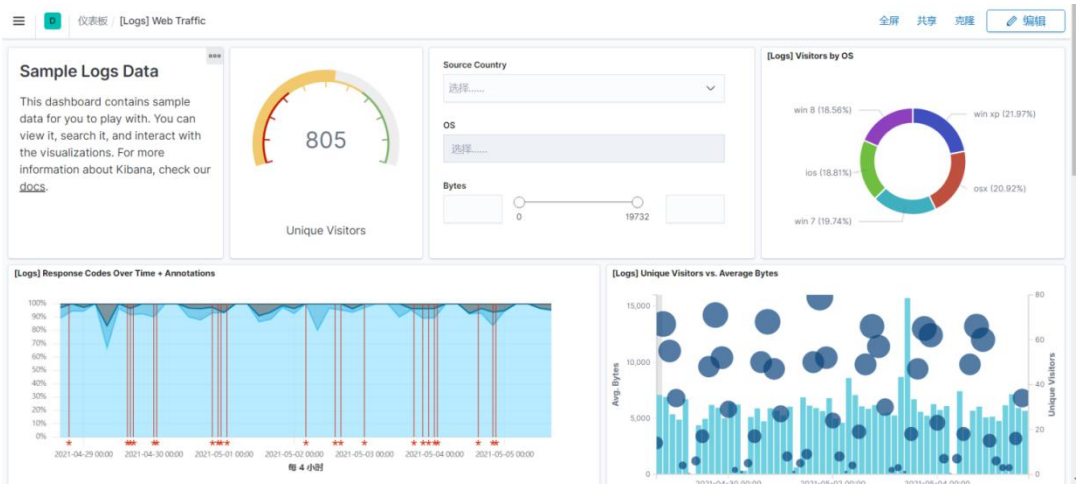

搜索
KQL
Last 15 minutes
显示日期
刷新
+ 添加筛选
kibana_sample_data_logs
搜索字段名称
按类型筛选
选定字段
@timestamp
clientip
host
可用字段
_id
_index
_score
_type
agent
bytes
event.dataset
extension
geo.coordinates
geo.dest
6 次命中
May 5, 2021 @ 17:10:22.068 - May 5, 2021 @ 17:25:22.068
自动
计数
timestamp/30 seconds
Time
@timestamp
clientip
host
May 5, 2021 @ 17:12:54.093 May 5, 2021 @ 17:12:54.093 232.20.97.5 www.elastic.co
May 5, 2021 @ 17:24:31.753 May 5, 2021 @ 17:24:31.753 109.128.129.183 www.elastic.co
May 5, 2021 @ 17:24:26.165 May 5, 2021 @ 17:24:26.165 44.146.84.219 www.elastic.co
May 5, 2021 @ 17:12:58.759 May 5, 2021 @ 17:12:58.759 39.77.148.125 www.elastic.co
May 5, 2021 @ 17:12:06.095 May 5, 2021 @ 17:12:06.095 2.46.184.206 artifacts.elastic.co
May 5, 2021 @ 17:20:48.838 May 5, 2021 @ 17:20:48.838 1.5.239.89 artifacts.elastic.co

查看样例数据图表


主页
Home
最近查看
Kibana
Overview
Discover
Dashboard
Canvas
Maps
Machine Learning
Visualize
Enterprise Search
概览
App Search
Workplace Search
Observability
Kibana

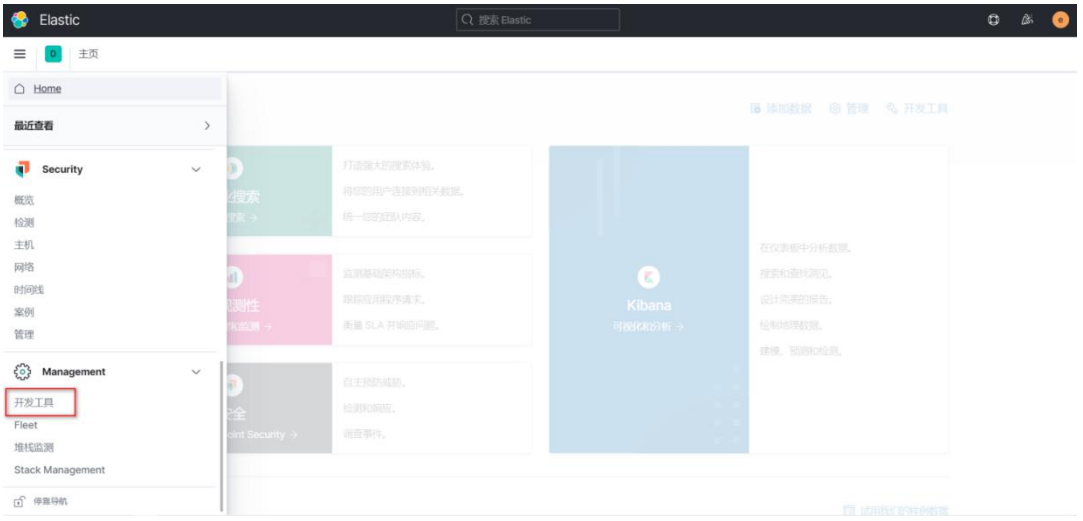

仪表板
仪表板
创建 仪表板
Search...
标题
描述
操作
[Logs] Web Traffic
Analyze mock web traffic log data for Elastic's website
Rows per page: 20
1

这是系统内置的一些数据图表方便我们快速了解 Kibana。

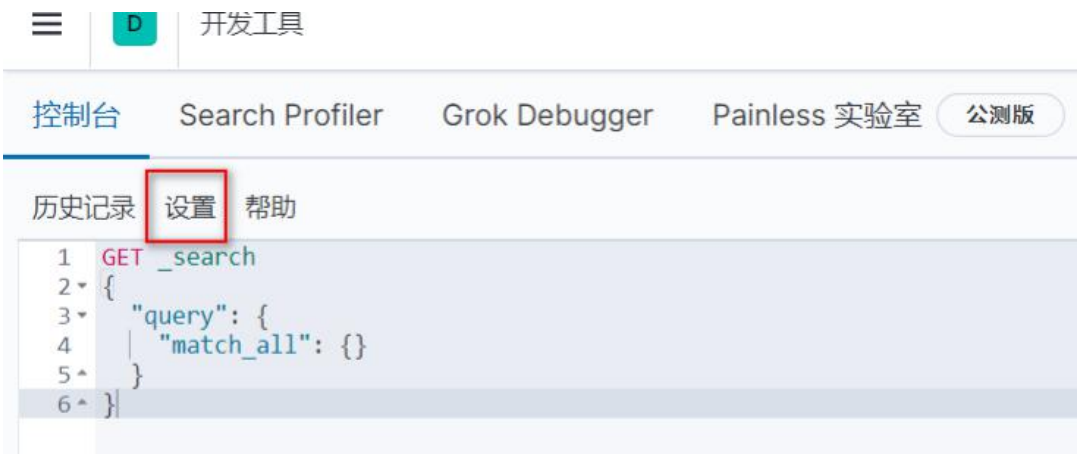


开发工具 Dev Tools

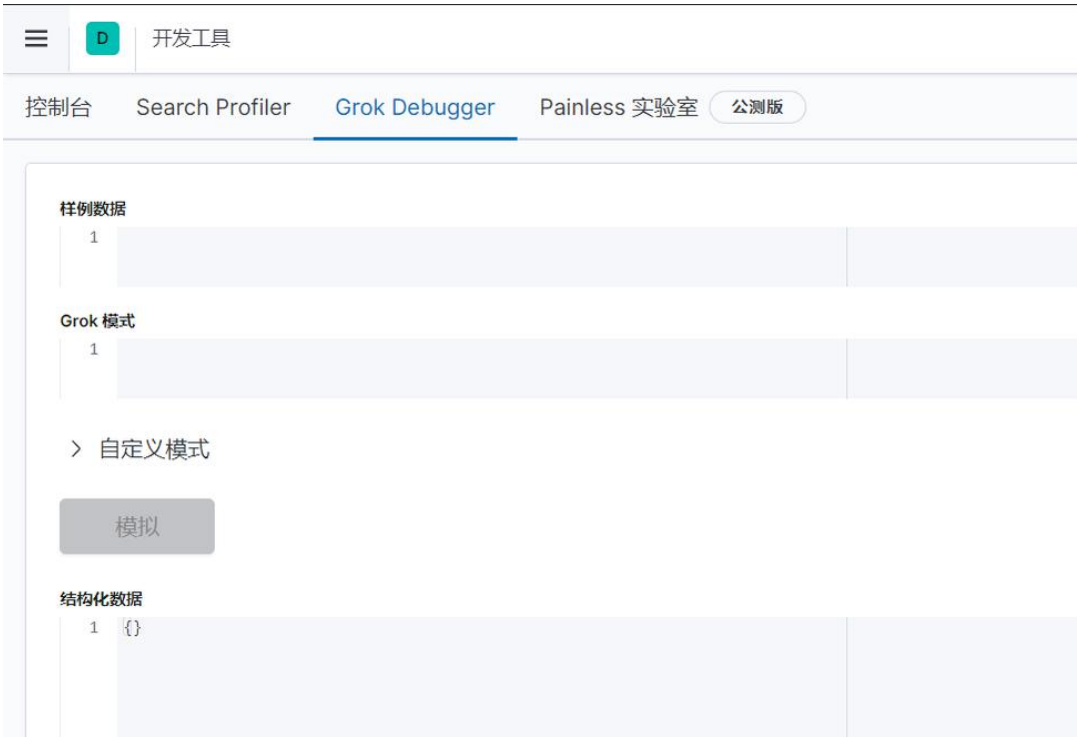
Dev Tools 是 Kibana 中最常用的功能，点击导航栏 -> Management -> 开发工具即可使用。



点击控制台 -> 设置，可以看到一些字体、换行等设置。



点击 Grok Debugger 可以进行一些正则调试，与 Logstash 结合解析日志。



基础查询

更加详细的查询语法参考：https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl.html

查看集群支持的选项

```
GET _cat
```

返回：

```
=^.^=
/_cat/allocation
/_cat/shards
/_cat/shards/{index}
/_cat/master
/_cat/nodes
/_cat/tasks
/_cat/indices
/_cat/indices/{index}
/_cat/segments
/_cat/segments/{index}
/_cat/count
/_cat/count/{index}
/_cat/recovery
/_cat/recovery/{index}
/_cat/health
/_cat/pending_tasks
/_cat/aliases
/_cat/aliases/{alias}
/_cat/thread_pool
/_cat/thread_pool/{thread_pools}
/_cat/plugins
/_cat/fielddata
/_cat/fielddata/{fields}
/_cat/nodeattrs
/_cat/repositories
```

```
/_cat/snapshots/{repository}
/_cat/templates
/_cat/ml/anomaly_detectors
/_cat/ml/anomaly_detectors/{job_id}
/_cat/ml/trained_models
/_cat/ml/trained_models/{model_id}
/_cat/ml/datafeeds
/_cat/ml/datafeeds/{datafeed_id}
/_cat/ml/data_frame/analytics
/_cat/ml/data_frame/analytics/{id}
/_cat/transforms
/_cat/transforms/{transform_id}
```

查看节点信息

```
GET _cat/nodes?v
```

返回：

```
ip heap.percent ram.percent cpu load_1m load_5m load_15m node.role master name
10.0.0.38 51 99 14 0.12 0.14 0.21 cdhilmrstw - 16195030170
01957332
10.0.0.32 47 99 9 0.70 0.28 0.26 cdhilmrstw - 161950301700
1957532
10.0.0.41 54 99 14 2.53 1.02 0.66 cdhilmrstw * 16195030170
01957432
```

查看 Master 节点信息

```
GET _cat/master?v
```

返回：

```
id                     host      ip        node
V_EuhAkbTS6T80mN3KX0XQ 10.0.0.41 10.0.0.41 1619503017001957432
```

查看所有节点上的热点线程

```
GET _nodes/hot_threads
```

返回：

```
::: {1619503017001957332}{26XvIqLSRlC2hEJ7-kAPUw}{9ytH1jFvTxWT8XHKILnWGg}{10.0.0.38}{10.0.0.38:9300}{cdhilmrstw}{ml.machine_memory=1959018496, rack=cvm_33_330001, xpack.installed=true, set=330001, transform.node=true, ip=9.27.21.20, temperature=hot, ml.max_open_jobs=20, region=33}
   Hot threads at 2021-05-05T12:44:05.242Z, interval=500ms, busiestThreads=3, ignoreIdleThreads=true:

::: {1619503017001957532}{Chd-cONFTwOTtZ5H-SdnpQ}{UgtOpFLURSa-Otaq5ECJnQ}{10.0.0.32}{10.0.0.32:9300}{cdhilmrstw}{ml.machine_memory=1959018496, rack=cvm_33_330001, xpack.installed=true, set=330001, transform.node=true, ip=9.27.19.91, temperature=hot, ml.max_open_jobs=20, region=33}
```

```
        Hot threads at 2021-05-05T12:44:05.266Z, interval=500ms, busiestThreads=3, ignoreIdle
Threads=true:

    ::: {161950301700195743 2}{V_EuhAkbTS6T80mN3KX0XQ}{VIcWTj5ERsmG_mY5jZSWtg}{10.0.0.
41}{10.0.0.41:9300}{cdhilmrstw}{ml.machine_memory=1959018496, rack=cvm_33_330001, xpack.i
nstalled=true, set=330001, transform.node=true, ip=9.27.16.243, temperature=hot, ml.max_ope
n_jobs=20, region=33}
        Hot threads at 2021-05-05T12:44:05.390Z, interval=500ms, busiestThreads=3, ignoreIdle
Threads=true:
```

查看不健康的分片或索引

```
GET _cluster/allocation/explain?pretty
```

返回：

```
{
  "error" : {
    "root_cause" : [
      {
        "type" : "illegal_argument_exception",
        "reason" : "unable to find any unassigned shards to explain [ClusterAllocationE
xplainRequest[useAnyUnassignedShard=true,includeYesDecisions?=false]"
      }
    ],
    "type" : "illegal_argument_exception",
    "reason" : "unable to find any unassigned shards to explain [ClusterAllocationExplai
nRequest[useAnyUnassignedShard=true,includeYesDecisions?=false]"
```

```
  },
  "status" : 400
}
```

查看线程池设置

```
GET _nodes/thread_pool/
```

返回：

```
{
  "_nodes" : {
    "total" : 3,
    "successful" : 3,
    "failed" : 0
  },
  "cluster_name" : "es-gcudgkos",
  "nodes" : {
    "Chd-cONFTwOTtZ5H-SdnpQ" : {
      "name" : "1619503017001957532",
      "transport_address" : "10.0.0.32:9300",
      "host" : "10.0.0.32",
      "ip" : "10.0.0.32",
      "version" : "7.10.1",
      "build_flavor" : "default",
      "build_type" : "tar",
      "build_hash" : "27aa98ee709dc860b4bec3994b44ba2e6c8dd73d",
```

```
        "roles" : [
          "data",
          "data_cold",
          "data_content",
          "data_hot",
          "data_warm",
          "ingest",
          "master",
          "ml",
          "remote_cluster_client",
          "transform"
        ],
......
```

查看集群全部信息

```
GET _cluster/stats?human&pretty
```

返回：

```
{
  "_nodes" : {
    "total" : 3,
    "successful" : 3,
    "failed" : 0
  },
  "cluster_name" : "es-gcudgkos",
  "cluster_uuid" : "UhtpZp9lScapLQIbid4gbw",
```

```
    "timestamp" : 1620218697400,
    "status" : "green",
    "indices" : {
      "count" : 29,
      "shards" : {
        "total" : 58,
        "primaries" : 29,
        "replication" : 1.0,
        "index" : {
          "shards" : {
            "min" : 2,
            "max" : 2,
            "avg" : 2.0
          },
          "primaries" : {
            "min" : 1,
            "max" : 1,
            "avg" : 1.0
          },
          "replication" : {
            "min" : 1.0,
            "max" : 1.0,
            "avg" : 1.0
          }
        }
      },
.....
```

查看集群状态

```
GET _cluster/health?pretty
```

返回：

```
{
  "cluster_name" : "es-gcudgkos",
  "status" : "green",
  "timed_out" : false,
  "number_of_nodes" : 3,
  "number_of_data_nodes" : 3,
  "active_primary_shards" : 29,
  "active_shards" : 58,
  "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" : 100.0
}
```

获取所有索引的信息

```
GET _cat/indices?v&pretty
```

返回：

```
health status index uuid pri rep docs.count docs.deleted store.size pri.store.size
green open .monitoring-kibana-7-2021.05.05 6q8JfbKET9WHaIm7cr2psg 1 1 18384 0 8.2mb 3.2mb
green open .monitoring-kibana-7-2021.05.04 mZpqpa90R22C3PSvpv_CnA 1 1 34556 0 10.3mb 5.1mb
green open .monitoring-kibana-7-2021.04.30 W2FAgGZ5TASJHYnoT5ofvg 1 1 34558 0 10.5mb 5.2mb
green open .items-default-000001 Se5hFqb7ThiiNbx8MMNQ3g 1 1 0 0 416b 208b
green open .monitoring-kibana-7-2021.05.03 mRH2eYPUTjOTWk0745d0Sw 1 1 34560 0 10.3mb 5.1mb
green open .monitoring-kibana-7-2021.05.02 G2_RTwVGRyii0pK9YPzp6w 1 1 34556 0 10.3mb 5.1mb
green open .monitoring-kibana-7-2021.05.01 GXj_YZZkSaWezvKDMJBTGA 1 1 34560 0 10.4mb 5.2mb
green open .apm-custom-link b3aD6BJ_TNOTEeDj6i6oDQ 1 1 0 0 416b 208b
green open .kibana_task_manager_1 IwHXIHnOSN6PJNN_kpvWLg 1 1 6 3516 3.5mb 578.2kb
green open logs-index_pattern_placeholder XR4BMRsgQzWgTX8-oXez-A 1 1 0 0 416b 208b
green open .monitoring-es-7-2021.04.29 NK-SLq7BSVaDUTmIl8rQcw 1 1 36357 0 46.1mb 20.7mb
green open .monitoring-kibana-7-2021.04.29 k7IXameTQIygrqKo6jz31w 1 1 34560 0 10.5mb 5.2mb
```

```
green open .lists-default-000001 6mx7-p1xSVef1uBunLAGgw 1 1
0 0 416b 208b
green open .apm-agent-configuration XxPQmjjfRQu_Qdl2jwZHxg 1 1
0 0 416b 208b
green open .kibana_1 xXlKVyfsSbSFY2ygRaalRw 1 1
140 61 8.9mb 4.4mb
green open .monitoring-es-7-2021.04.30 ULyYh6fTRdOBAUfvyKK2Kw 1 1
38994 0 43.4mb 21.7mb
green open .security-7 2rEeUg0vT8a6cQv18v73LA 1 1
46 0 306.9kb 106kb
green open .monitoring-es-7-2021.05.01 RrtO4Wp7Taad93D4awHNJg 1 1
41977 0 46.7mb 23.3mb
green open wfe 1UlifJS6Rsu4O8fVxaElGg 1 1
1085 1 16.5mb 8.2mb
green open .kibana-event-log-7.10.1-000001 ny0H9LiiQPuI8AH9zcPkfg 1 1
4 0 23.6kb 11.8kb
green open metrics-index_pattern_placeholder GQWH2XCFSq2gbIqGwgL_XA 1 1
0 0 416b 208b
green open kibana_sample_data_logs COoX4096S0az6IzJ6Mo7MA 1 1
14074 0 18.9mb 9.4mb
green open .async-search NsGrHYxWRK-C0iLvZ7THQQ 1 1
0 0 7.2kb 3.6kb
green open .monitoring-es-7-2021.05.03 xnk57_D-SSeqyW6kwpuSgw 1 1
47673 0 53.2mb 26.6mb
green open .monitoring-es-7-2021.05.02 Jii6AjJbTOmbBpT-zbM5RA 1 1
44778 0 50.2mb 25mb
green open .monitoring-es-7-2021.05.05 YtJgJS-tRSq39a18-oLLyQ 1 1 2
7816 44548 37.4mb 18.9mb
green open .monitoring-es-7-2021.05.04 j3UnA4odSlOLgT3AlAZMAQ 1 1
52027 0 55.8mb 27.8mb
```

查看集群状态

- green：所有功能完好；
- yellow：数据是可用的，但存在未被分配的副本；
- red：集群中存在不可用的数据；

```
GET _cat/health?v
```

返回：

```
epoch      timestamp cluster       status node.total node.data shards pri relo init unassign pending_tasks max_task_wait_time active_shards_percent
1620218868 12:47:48  es-gcudgkos   green           3         3     58  29    0    0        0             0                  -                100.0%
```

创建索引 test

- test：索引名；
- pretty：输出格式良好的 JSON 响应；

```
PUT test?pretty
```

返回：

```
{
  "acknowledged" : true,
  "shards_acknowledged" : true,
  "index" : "test"
}
```

查看 test 索引

```
GET test
```

返回：

```
{
  "test" : {
    "aliases" : { },
    "mappings" : {
      "dynamic_templates" : [
        {
          "message_full" : {
            "match" : "message_full",
            "mapping" : {
              "fields" : {
                "keyword" : {
                  "ignore_above" : 2048,
                  "type" : "keyword"
                }
              },
              "type" : "text"
            }
```

```
            }
          },
          {
            "message" : {
              "match" : "message",
              "mapping" : {
                "type" : "text"
              }
            }
          },
          {
            "strings" : {
              "match_mapping_type" : "string",
              "mapping" : {
                "type" : "keyword"
              }
            }
          }
        ]
      },
      "settings" : {
        "index" : {
          "routing" : {
            "allocation" : {
              "include" : {
                "_tier_preference" : "data_content"
              }
            }
          },
          "refresh_interval" : "10s",
          "number_of_shards" : "1",
```

```
            "translog" : {
              "sync_interval" : "5s",
              "durability" : "async"
            },
            "provided_name" : "test",
            "max_result_window" : "65536",
            "creation_date" : "1620218910183",
            "unassigned" : {
              "node_left" : {
                "delayed_timeout" : "5m"
              }
            },
            "number_of_replicas" : "1",
            "uuid" : "Xshcy1IyRemznHzcv3Focw",
            "version" : {
              "created" : "7100199"
            }
          }
        }
      }
}
```

判断索引 test 是否存在

```
HEAD test
```

返回：

```
200 - OK
```

打开索引 test

```
POST test/_open
```

返回：

```
{
  "acknowledged" : true,
  "shards_acknowledged" : true,
  "indices" : {
    "test" : {
      "closed" : true
    }
  }
}
```

关闭索引 test

```
POST test/_close
```

返回：

```
{
  "acknowledged" : true,
  "shards_acknowledged" : false,
  "indices" : { }
}
```

查看索引 test 状态

```
GET test/_stats
```

返回：

```
{
  "_shards" : {
    "total" : 2,
    "successful" : 2,
    "failed" : 0
  },
  "_all" : {
    "primaries" : {
      "docs" : {
        "count" : 0,
        "deleted" : 0
      },
      "store" : {
        "size_in_bytes" : 230,
        "reserved_in_bytes" : 0
      },
      "indexing" : {
        "index_total" : 0,
        "index_time_in_millis" : 0,
        "index_current" : 0,
        "index_failed" : 0,
        "delete_total" : 0,
```

```
            "delete_time_in_millis" : 0,
            "delete_current" : 0,
            "noop_update_total" : 0,
            "is_throttled" : false,
            "throttle_time_in_millis" : 0
        },
......
```

删除索引 test

- 根据索引名称删除

```
DELETE test?pretty
```

- 可以一次删除多个索引（以逗号间隔）删除所有索引 _all 或通配符 *

查看索引模板

```
GET _template                          #查看所有索引摸板
GET _template/temp*                    #查看以 temp 开头的索引模板
GET _template/template_1,template_2    #查看 template_1 和 template_2 索引摸板
GET _template/template_name            #查看名称为 template_name 的索引摸板
```

删除索引模板

```
DELETE _template/template_name
```

返回

Grok Debugger 调试

更多模式参考：http://grokdebug.herokuapp.com/patterns

例一

input

```
[2020-04-03T16:51:35,918] [DEBUG] [o.e.a.a.c.n.i.TransportNodesInfoAction] [data02-131-211] failed to execute on node [08GhVGGgRCqUE3qAdXf04g] org.elasticsearch.transport.NodeNotConnectedException: [master01-34.5][172.16.34.5:9300] Node not connected
```

pattern

```
(?<date>\d{4}-\d{2}-\d{2}T\d{2}:\d{2}:\d{2},\d{3})\] \[(?<loglevel>[A-Z \s]{4,5})] \[(?<service>[A-Za-z0-9/.]{4,40})\] \[(?<node>[A-Za-z0-9/-]{4,40})\] (?<msg>.*)
```

result

```
{
  "date": [
    [
      "2020-04-03T16:51:35,918"
    ]
  ],
  "loglevel": [
```

```
        [
          "DEBUG"
        ]
      ],
      "service": [
        [
          "o.e.a.a.c.n.i.TransportNodesInfoAction"
        ]
      ],
      "node": [
        [
          "data02-131-211"
        ]
      ],
      "msg": [
        [
          "failed to execute on node [08GhVGGgRCqUE3qAdXf04g] org.elasticsearch.transport.NodeNotConnectedException: [master01-34.5][172.16.34.5:9300] Node not connected"
        ]
      ]
    }
```

例二

input

```
[2020-04-03 09:04:20,446][INFO][Thread-16][c.h.jobhandler.ELKTestJobHandlervds.6665][ELKTestJobHandler.java : 32][elkTestJobHandler: 普通日志输出测试]
```

pattern

```
(?<date>\d{4}-\d{2}-\d{2}\s\d{2}:\d{2}:\d{2},\d{3})\]\[(?<loglevel>[A-Z]{4,5})\]\[(?<thread>[A-Za-z0-9-/-]{4,40})\]\[(?<class>[A-Za-z0-9/.]{4,40})\]\[(?<msg>.*)
```

result


```
{
  "date": [
    [
      "2020-04-03  09:04:20,446"
    ]
  ],
  "loglevel": [
    [
      "INFO"
    ]
  ],
  "thread": [
    [
      "Thread-16"
    ]
  ],
  "class": [
    [
      "c.h.jobhandler.ELKTestJobHandlervds.6665"
    ]
  ],
```

```
    "msg": [
      [
        "ELKTestJobHandler.java : 32][elkTestJobHandler: 普通日志输出测试]"
      ]
    ]
  }
```

例三

input

```
  2018/05/01 16:16:01.892 - OK - 759.2ms - 172.29.1.7:35184[485388]->172.7.1.39:3306[15251
62561129639717]:<DB>:select count(*) from test[];
```

pattern

```
  (?<date>\d{4}/\d{2}/\d{2}\s(?<datetime>%{TIME}))\s-\s(?<status>\w{2})\s-\s(?<respond_time>\
d+)\.\d+\w{2}\s-\s%{IP:client}:(?<client-port>\d+)\[\d+\]->%{IP:server}:(?<server-port>\d+).*:(?<data
bases><\w+>):(?<SQL>.*)
```

result

```
  {
    "date": [
      [
        "2018/05/01 16:16:01.892"
      ]
```

```
    ],
    "datetime": [
      [
        "16:16:01.892"
      ]
    ],
    "TIME": [
      [
        "16:16:01.892"
      ]
    ],
    "HOUR": [
      [
        "16"
      ]
    ],
    "MINUTE": [
      [
        "16"
      ]
    ],
    "SECOND": [
      [
        "01.892"
      ]
    ],
    "status": [
      [
        "OK"
      ]
    ],
```

```
"respond_time": [
  [
    "759"
  ]
],
"client": [
  [
    "172.29.1.7"
  ]
],
"IPV6": [
  [
    null,
    null
  ]
],
"IPV4": [
  [
    "172.29.1.7",
    "172.7.1.39"
  ]
],
"client-port": [
  [
    "35184"
  ]
],
"server": [
  [
    "172.7.1.39"
  ]
```

```
    ],
    "server-port": [
      [
        "3306"
      ]
    ],
    "databases": [
      [
        "<DB>"
      ]
    ],
    "SQL": [
      [
        "select count(*) from test[];"
      ]
    ]
  }
```

作图

以 Nginx 日志为例，插入数据，生产环境中可以通过 Beats 收集到 Elasticsearch 再作图。

插入 Nginx 日志测试数据

在 Kibana 的开发工具中执行：

```
POST nginx-access-logs/_bulk
{"index":{"_id":"1"}}
{"log_time":"2020-06-30T18:05:03+08:00","client_ip":"115.159.116.79","method":"POST","http_code":"200","size":"66","usersip":"119.85.16.64, 115.159.116.79","request_uri":"http://qdweb.zksf.com/xfjr-zfb/PhoneQry.do","req_time":"0.016","user_ua":"Mozilla/5.0 (iPhone; CPU iPhone OS 12_3_2 like Mac OS X) AppleWebKit/605.1.15 (KHTML, like Gecko) Mobile/15E148 MicroMessenger/7.0.4(0x17000428) NetType/4G Language/zh_CN"}
{"index":{"_id":"2"}}
{"log_time":"2020-06-30T18:05:04+08:00","client_ip":"123.206.205.161","method":"GET","http_code":"200","size":"11133","usersip":"117.136.84.181, 123.206.205.161","request_uri":"http://qdweb.zksf.com/static/wx/dist/htmls/applyCardMoneySuc/mod.js","req_time":"0.000","user_ua":"Mozilla/5.0 (Linux; Android 8.0.0; SM-G9550 Build/R16NW; wv) AppleWebKit/537.36 (KHTML, like Gecko) Version/4.0 Chrome/66.0.3359.126 MQQBrowser/6.2 TBS/044704 Mobile Safari/537.36 MMWEBID/1866 MicroMessenger/7.0.4.1420(0x2700043C) Process/tools NetType/4G Language/zh_CN"}
{"index":{"_id":"3"}}
{"log_time":"2020-06-30T18:05:06+08:00","client_ip":"123.206.107.139","method":"POST","http_code":"200","size":"3887","usersip":"117.136.44.137, 123.206.107.139","request_uri":"http://qdweb.zksf.com/xfjr-zfb/custLoanInfoQry.do","req_time":"0.028","user_ua":"Mozilla/5.0 (Linux; Android 8.1.0; PACM00 Build/O11019; wv) AppleWebKit/537.36 (KHTML, like Gecko) Version/4.0 Chrome/66.0.3359.126 MQQBrowser/6.2 TBS/044705 Mobile Safari/537.36 MMWEBID/908 MicroMessenger/7.0.4.1420(0x2700043C) Process/tools NetType/4G Language/zh_CN"}
{"index":{"_id":"4"}}
{"log_time":"2020-06-30T18:05:06+08:00","client_ip":"115.159.93.78","method":"POST","http_code":"200","size":"86","usersip":"218.26.54.246, 115.159.93.78","request_uri":"http://qdweb.zksf.com/xfjr-zfb/LoanAntiFraudQry.do","req_time":"0.022","user_ua":"Mozilla/5.0 (Linux; Android 8.1.0; vivo X21A Build/OPM1.171019.011; wv) AppleWebKit/537.36 (KHTML, like Gecko) Version/4.0 Chrome/66.0.3359.126 MQQBrowser/6.2 TBS/044705 Mobile Safari/537.36 MicroMessenger/6.7.2.1340(0x260702C5) NetType/4G Language/zh_CN"}
{"index":{"_id":"5"}}
```

{"log_time":"2020-06-30T18:05:31+08:00","client_ip":"123.206.205.161","method":"POST","http_code":"200","size":"110","usersip":"117.84.191.27, 123.206.205.161","request_uri":"http://qdweb.zksf.com/xfjr-zfb/WeixinForOpenId.do","req_time":"0.154","user_ua":"Mozilla/5.0 (iPhone; CPU iPhone OS 12_2 like Mac OS X) AppleWebKit/605.1.15 (KHTML, like Gecko) Mobile/15E148 MicroMessenger/7.0.4(0x17000428) NetType/WIFI Language/zh_CN"}
{"index":{"_id":"6"}}
{"log_time":"2020-06-30T18:05:32+08:00","client_ip":"123.206.205.161","method":"GET","http_code":"400","size":"2119","usersip":"117.84.191.27, 123.206.205.161","request_uri":"http://qdweb.zksf.com/static/wx/dist/htmls/applyCardMoney/applyCardMoney.html","req_time":"0.000","user_ua":"Mozilla/5.0 (iPhone; CPU iPhone OS 12_2 like Mac OS X) AppleWebKit/605.1.15 (KHTML, like Gecko) Mobile/15E148 MicroMessenger/7.0.4(0x17000428) NetType/WIFI Language/zh_CN"}
{"index":{"_id":"7"}}
{"log_time":"2020-06-30T18:05:32+08:00","client_ip":"123.206.205.161","method":"POST","http_code":"302","size":"150","usersip":"117.84.191.27, 123.206.205.161","request_uri":"http://qdweb.zksf.com/xfjr-zfb/LoginStatusQry.do","req_time":"0.014","user_ua":"Mozilla/5.0 (iPhone; CPU iPhone OS 12_2 like Mac OS X) AppleWebKit/605.1.15 (KHTML, like Gecko) Mobile/15E148 MicroMessenger/7.0.4(0x17000428) NetType/WIFI Language/zh_CN"}
{"index":{"_id":"8"}}
{"log_time":"2020-06-30T18:05:32+08:00","client_ip":"111.231.53.89","method":"POST","http_code":"200","size":"174","usersip":"117.136.67.251, 111.231.53.89","request_uri":"http://qdweb.zksf.com/xfjr-zfb/AntiFraudResultQry.do","req_time":"0.027","user_ua":"Mozilla/5.0 (Linux; Android 8.1.0; vivo Y83A Build/O11019; wv) AppleWebKit/537.36 (KHTML, like Gecko) Version/4.0 Chrome/66.0.3359.126 MQQBrowser/6.2 TBS/044705 Mobile Safari/537.36 MMWEBID/2371 MicroMessenger/7.0.4.1420(0x2700043C) Process/tools NetType/4G Language/zh_CN"}
{"index":{"_id":"9"}}
{"log_time":"2020-06-30T18:05:32+08:00","client_ip":"123.206.205.161","method":"GET","http_code":"200","size":"1306","usersip":"117.84.191.27, 123.206.205.161","request_uri":"http://qdweb.zksf.com/static/wx/dist/images/emApprove.png","req_time":"0.000","user_ua":"Mozilla/5.0 (iPhone; CPU iPhone OS 12_2 like Mac OS X) AppleWebKit/605.1.15 (KHTML, like Gecko) Mobile/15E148 MicroMessenger/7.0.4(0x17000428) NetType/WIFI Language/zh_CN"}
{"index":{"_id":"10"}}

{"log_time":"2020-06-30T18:05:32+08:00","client_ip":"122.152.197.50","method":"POST","http_code":"200","size":"110","usersip":"60.119.37.213, 122.152.197.50","request_uri":"http://qdweb.zksf.com/xfjr-zfb/CheckNotice.do","req_time":"0.015","user_ua":"Mozilla/5.0 (iPhone; CPU iPhone OS 12_2 like Mac OS X) AppleWebKit/605.1.15 (KHTML, like Gecko) Mobile/15E148 MicroMessenger/7.0.4(0x17000428) NetType/WIFI Language/zh_CN"}

返回：

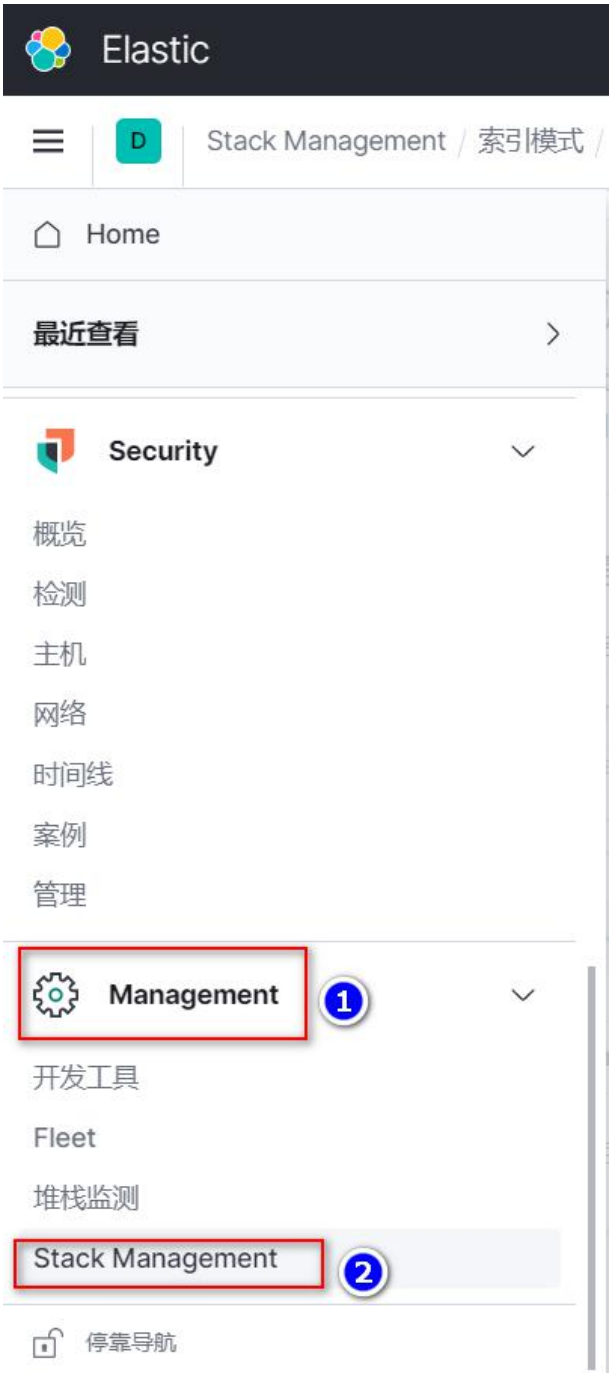

```
{
  "took" : 612,
  "errors" : false,
  "items" : [
    {
      "index" : {
        "_index" : "nginx-access-logs",
        "_type" : "_doc",
        "_id" : "1",
        "_version" : 1,
        "result" : "created",
        "_shards" : {
          "total" : 2,
          "successful" : 2,
          "failed" : 0
        },
        "_seq_no" : 0,
        "_primary_term" : 1,
        "status" : 201
      }
    },
```

```
{
  "index" : {
    "_index" : "nginx-access-logs",
    "_type" : "_doc",
    "_id" : "6",
    "_version" : 1,
    "result" : "created",
    "_shards" : {
      "total" : 2,
      "successful" : 2,
      "failed" : 0
    },
    "_seq_no" : 5,
    "_primary_term" : 1,
    "status" : 201
  }
},
{
  "index" : {
    "_index" : "nginx-access-logs",
    "_type" : "_doc",
    "_id" : "7",
    "_version" : 1,
    "result" : "created",
    "_shards" : {
      "total" : 2,
      "successful" : 2,
      "failed" : 0
    },
    "_seq_no" : 6,
    "_primary_term" : 1,
```

```
            "status" : 201
          }
        },
        {
          "index" : {
            "_index" : "nginx-access-logs",
            "_type" : "_doc",
            "_id" : "8",
            "_version" : 1,
            "result" : "created",
            "_shards" : {
              "total" : 2,
              "successful" : 2,
              "failed" : 0
            },
            "_seq_no" : 7,
            "_primary_term" : 1,
            "status" : 201
          }
        },
       ......
    ]
}
```

创建索引

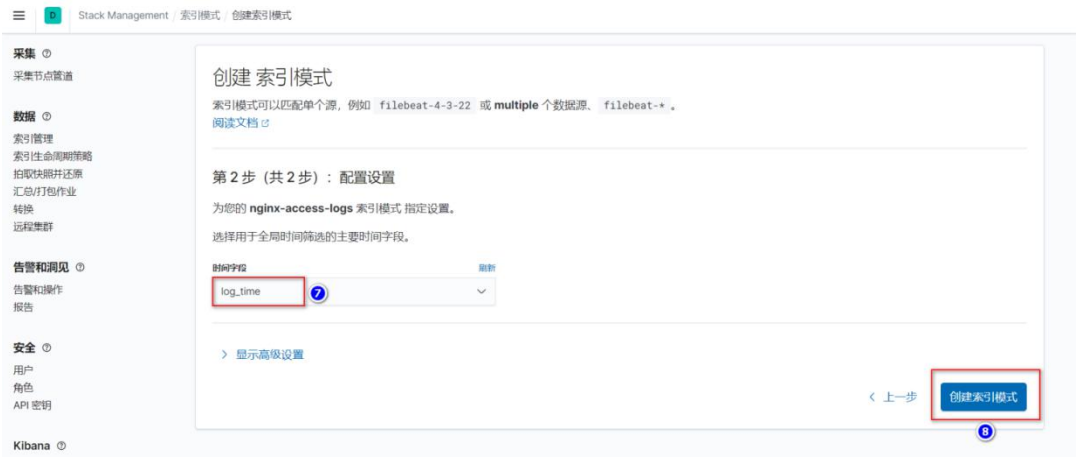
导航 -> Kibana -> Management -> Stack Management -> Index patterns（索引模式）-> Create index patterns（创建索引模式）-> nginx-access-logs -> Next step（下一步） -> log_time -> Create index patterns（创建索引模式）


Elastic
D
Stack Management / 索引模式 /
Home
最近查看
Security
概览
检测
主机
网络
时间线
案例
管理
Management
1
开发工具
Fleet
堆栈监测
Stack Management
2
停靠导航

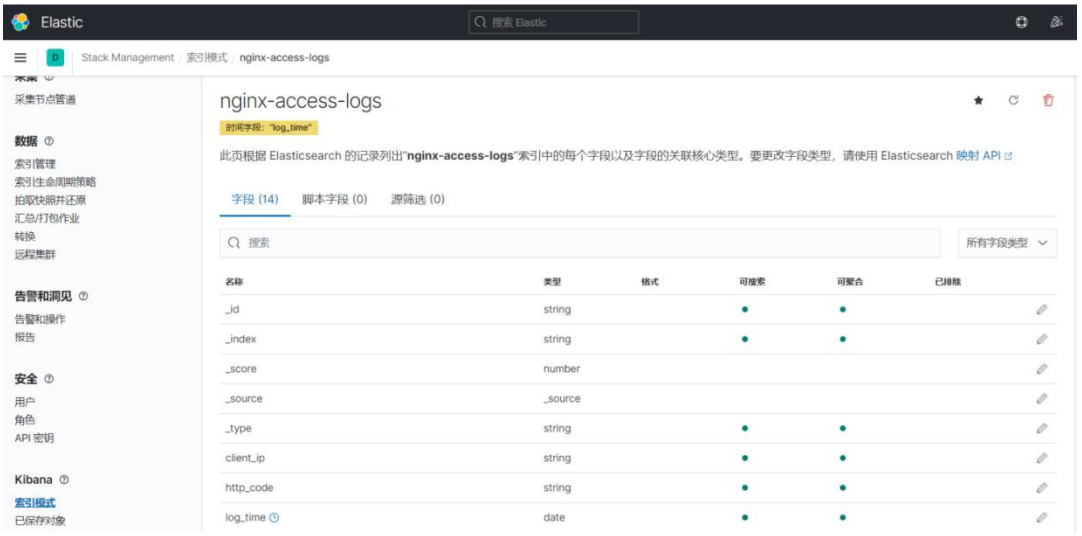

Elastic
D
Stack Management
汇总/打包作业
转换
远程集群
告警和洞见
告警和操作
报告
安全
用户
角色
API 密钥
Kibana
索引模式
3
已保存对象
工作区
高级设置
Stack
许可管理
8.0 升级助手

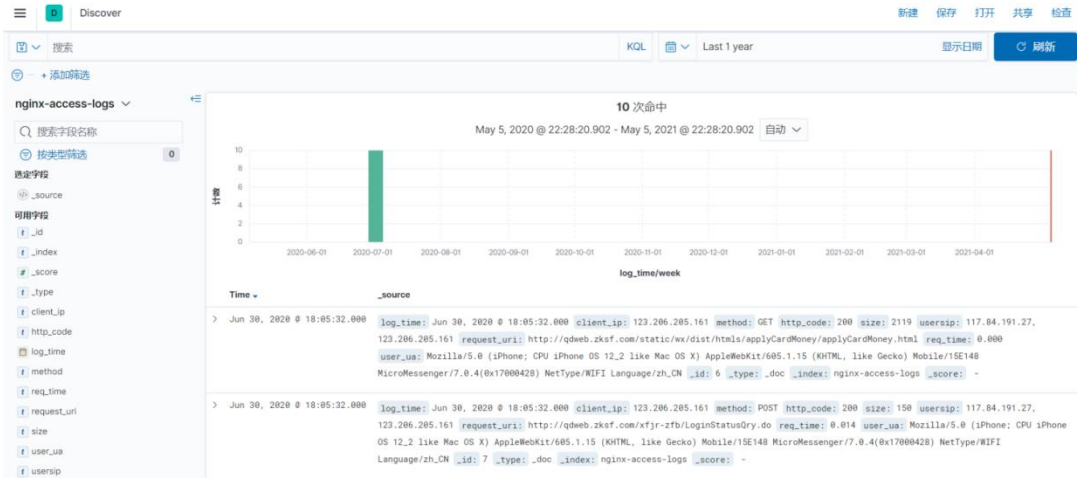

Elastic
搜索 Elastic
Stack Management / 索引模式
采集
采集节点管道
数据
索引管理
索引生命周期策略
拍取快照并还原
汇总/打包作业
转换
远程集群
告警和洞见
告警和操作
报告
索引模式
创建和管理帮助您从 Elasticsearch 中检索数据的索引模式。
4
创建索引模式
Search...
Pattern
wfe
默认值
kibana_sample_data_logs
Rows per page: 10
1


Stack Management / 索引模式 / 创建索引模式
采集
采集节点管道
数据
索引管理
索引生命周期策略
拍取快照并还原
汇总/打包作业
转换
远程集群
告警和洞见
告警和操作
报告
安全
用户
角色
API 密钥
创建 索引模式
索引模式可以匹配单个源，例如 filebeat-4-3-22 或 multiple 个数据源、filebeat-* 。
阅读文档
第 1 步（共 2 步）：定义索引模式
索引模式名称
nginx-access-logs
5
下一步 >
6
使用星号 (*) 匹配多个索引。不允许使用空格和字符 \, /, ?, ", <, >, |。
包括系统和隐藏索引
您的索引模式匹配 1 个源。
nginx-access-logs
索引
每页行数: 10


Stack Management / 索引模式 / 创建索引模式
采集
采集节点管道
数据
索引管理
索引生命周期策略
拍取快照并还原
汇总/打包作业
转换
远程集群
告警和洞见
告警和操作
报告
安全
用户
角色
API 密钥
Kibana
创建 索引模式
索引模式可以匹配单个源，例如 filebeat-4-3-22 或 multiple 个数据源、filebeat-* 。
阅读文档
第 2 步（共 2 步）：配置设置
为您的 nginx-access-logs 索引模式 指定设置。
选择用于全局时间筛选的主要时间字段。
时间字段
刷新
log_time
7
> 显示高级设置
< 上一步
创建索引模式
8

查看索引字段

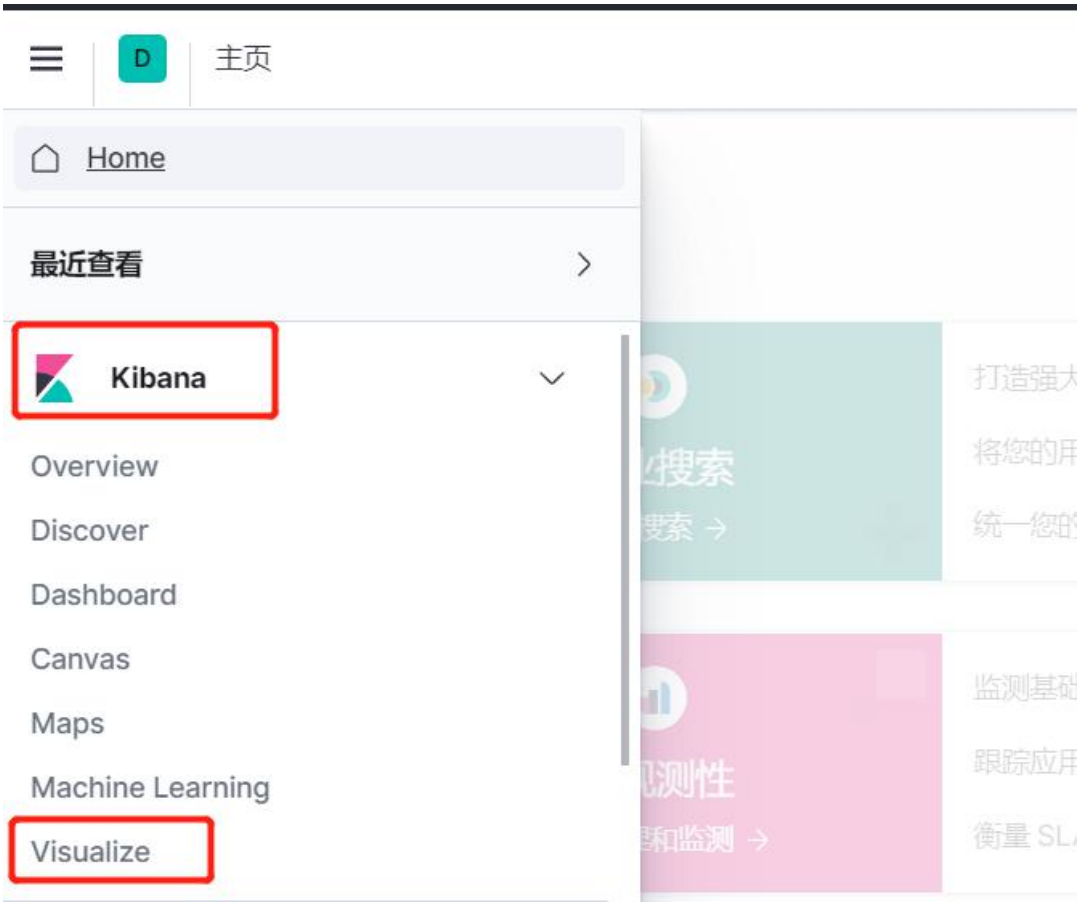




根据 Nginx 日志作图

状态码

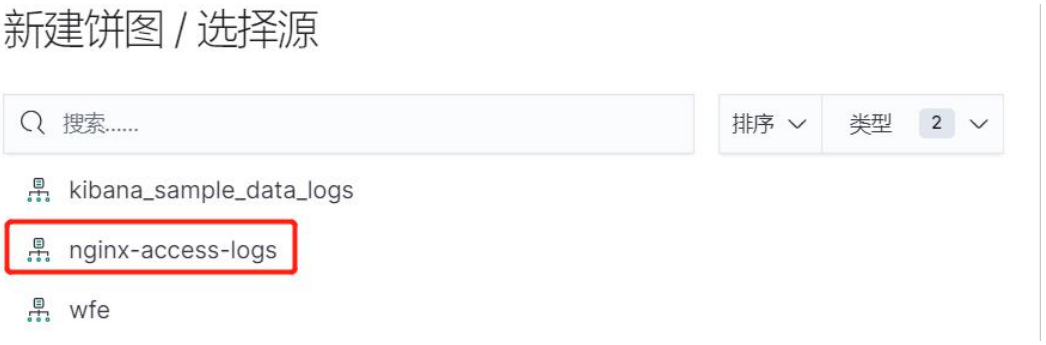
导航 -> Kibana -> Visualize -> Create visualization（创建可视化）



Create visualization（创建可视化）

Select a visualization type（选择可视化类型）-> Pie（饼图）

Choose a source（选择数据源）-> nginx-access-logs



Data（数据）-> Metrics（指标）-> Slice size（切片大小）-> Aggregation（聚合）->Count（计数）-> Custom label（定指标签）-> 状态码

Buckets（存储桶）-> Add（添加）

Split slices（拆分切片）

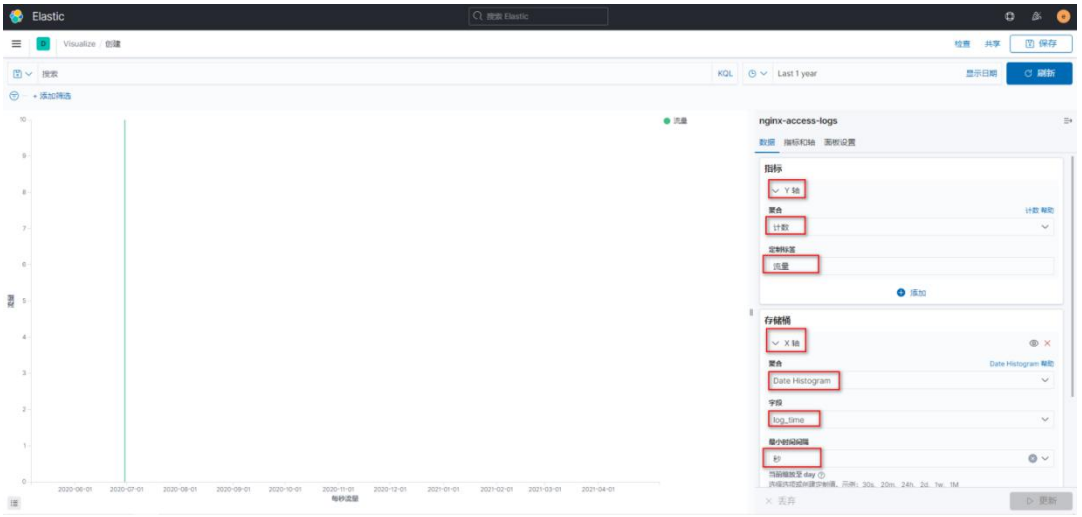
Aggregation（聚合）-> Terms（词）-> Field（字段）-> http_code -> Update（更新）

Save（保存）

状态码 -> 保存

流量

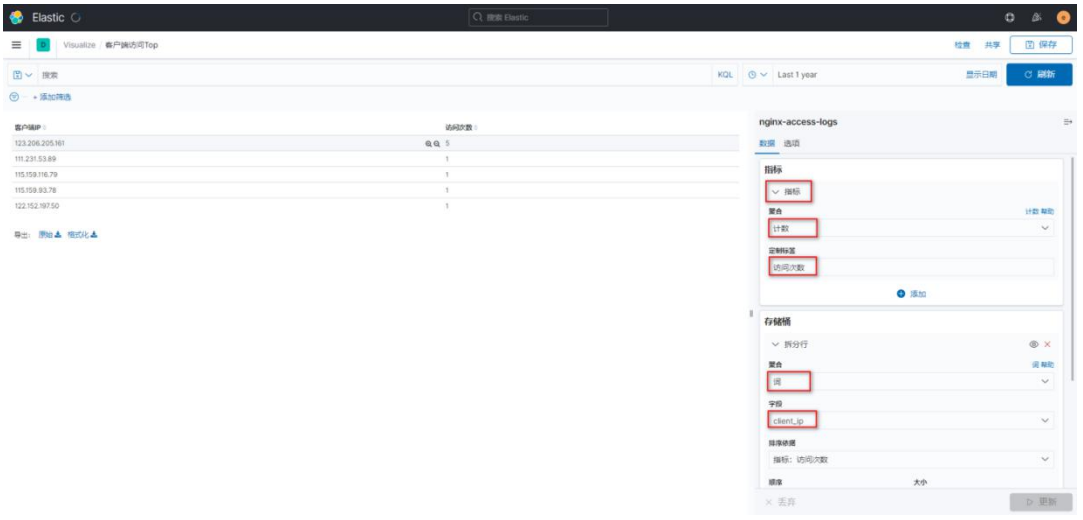
导航 -> Kibana -> Visualize -> Create visualization（创建可视化）-> Select a visualization type（选择可视化类型）-> Area（面积图）-> Choose a source（选择数据源）-> nginx-access-logs -> Data（数据）-> Y-axis（Y 轴）-> Aggregation（聚合）-> Count（计数）-> Custom label（定指标签）-> 流量 -> Buckets（存储桶）-> Add（添加）-> X-axis（X 轴）-> Aggregation（聚合）-> Date Histogram-> Field（字段）-> log_time -> Minimum interval（最小时间间隔）-> Second（秒）-> Custom label（每秒流量）-> Update（更新）-> Save（保存）-> 网络流量



客户端 IP

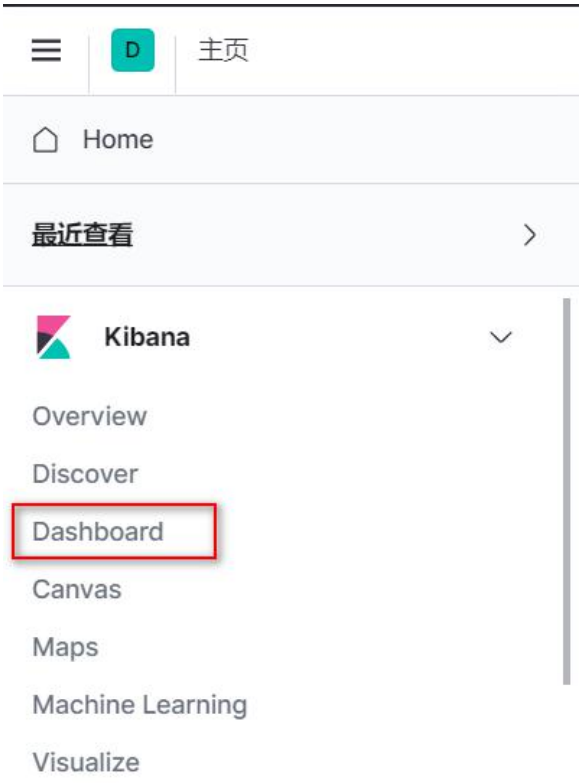
导航 -> Kibana -> Visualize -> Create visualization（创建可视化）-> Select a visualization type（选择可视化类型）-> Data Table（数据表）-> Choose a source

（选择数据源）-> nginx-access-logs -> Data（数据）-> Metrics（指标）-> Aggregation（聚合）-> Count（计数）-> Custom label（定指标签）-> 访问次数 -> Buckets（存储桶）-> Add（添加）-> Split rows（拆分行）->Aggregation（聚合）-> Terms（词）-> Field（字段）-> client_ip -> Custom label（客户端 IP）-> Update（更新）-> Save（保存）-> 客户端访问 Top

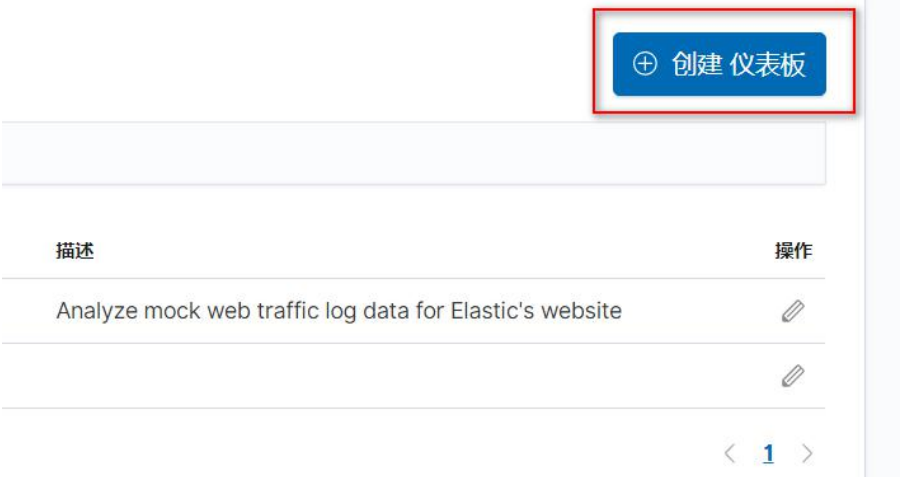


创建 Dashboard

导航 -> Kibana -> Dashboard（仪表盘）



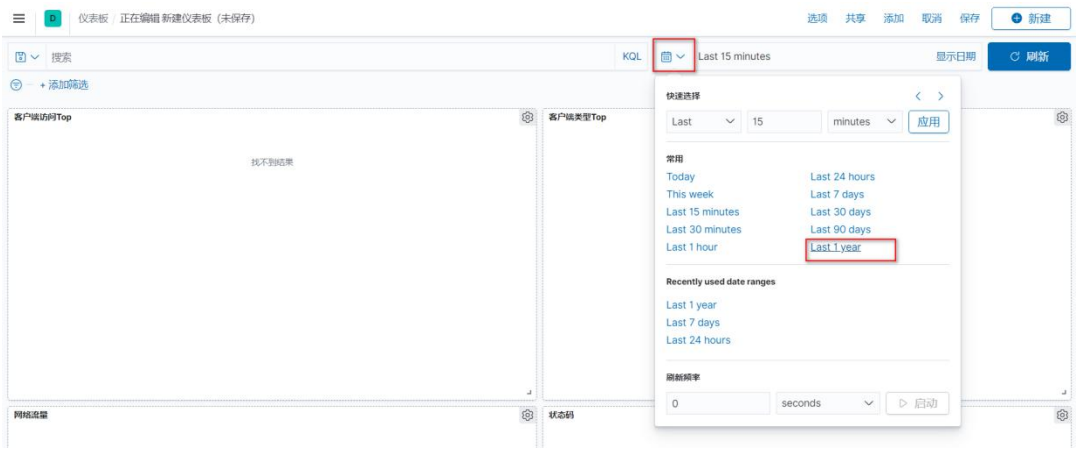
Create dashboard（创建仪表盘）



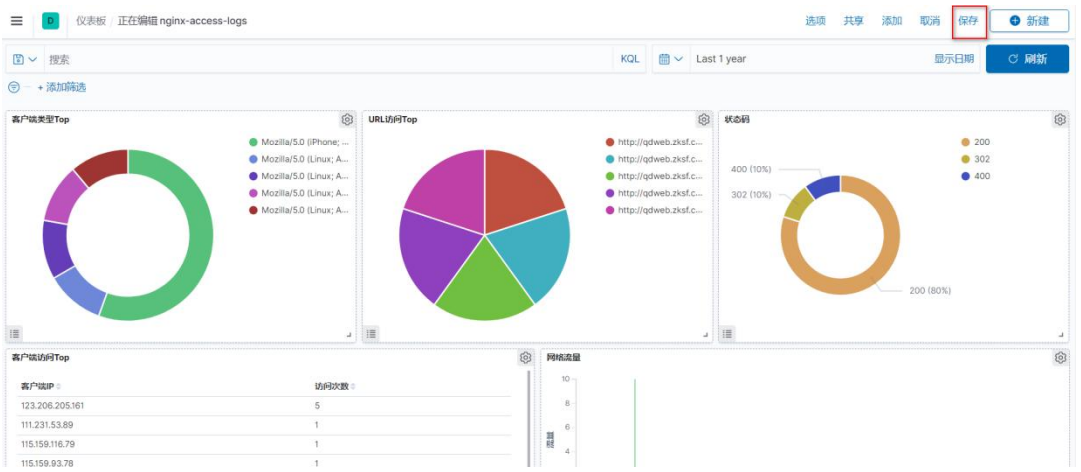
Add an existing（添加现有）

点击前面创建的图表

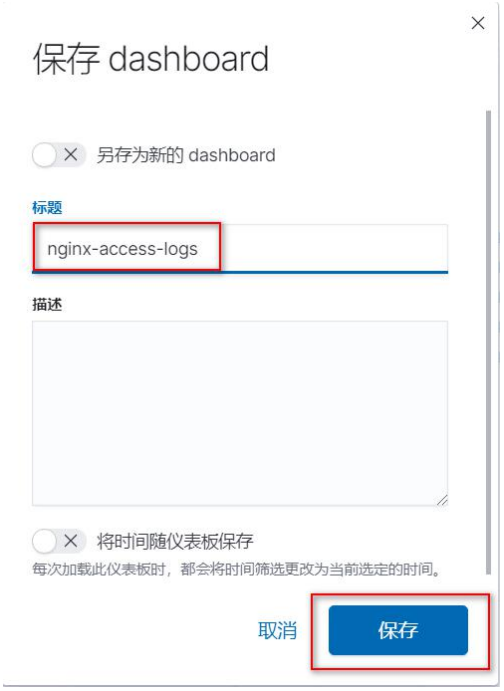
选择时间段，由于是假数据，直接选择 Last 1 year



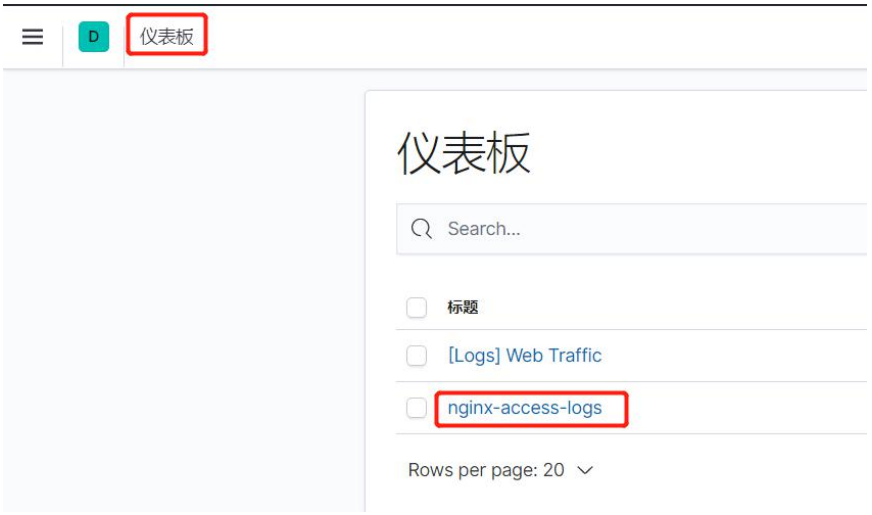
点击保存

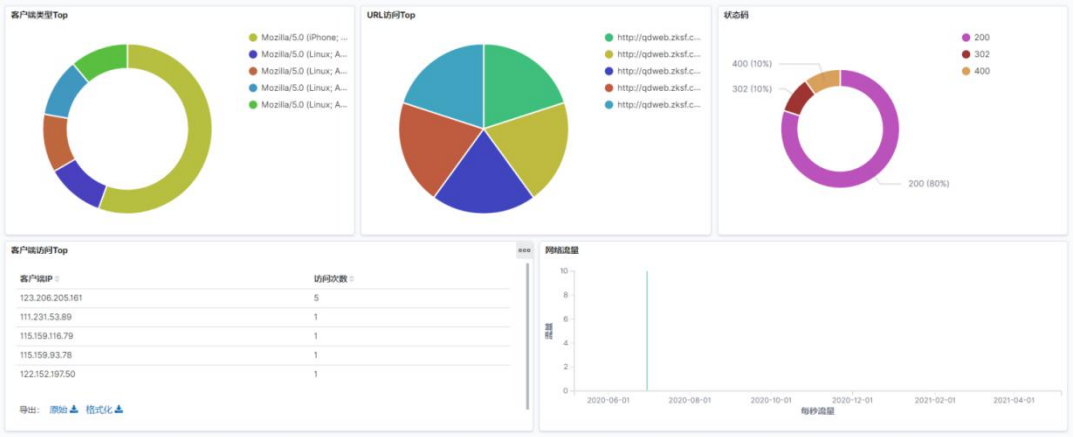


输入要保存的名称 nginx-access-logs 后点击保存



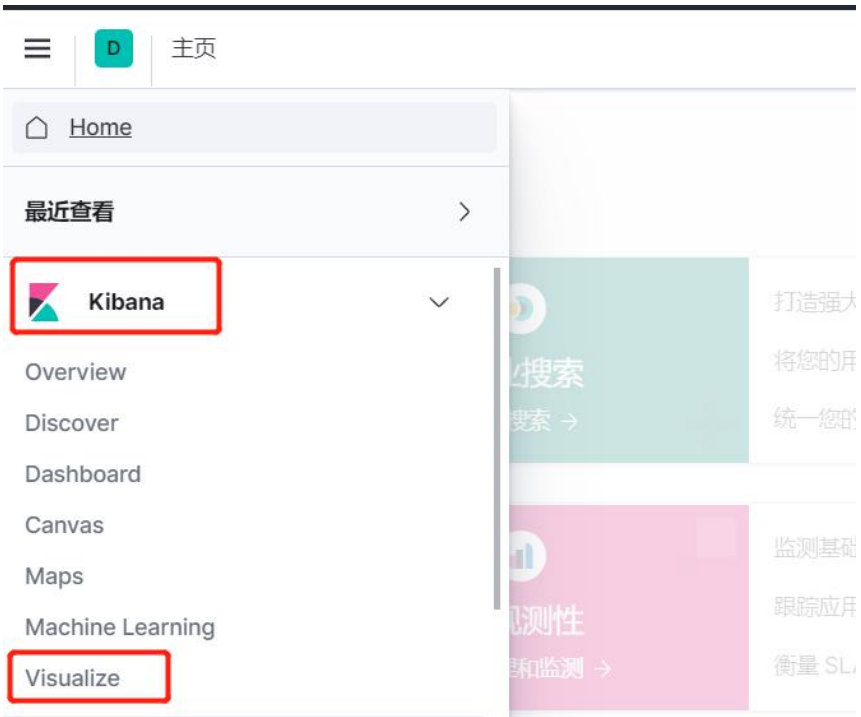
重新回到仪表盘，然后点击上面我们保存的 nginx-access-logs 名称就能看到这个仪表盘了。





Kibana Lens 可视化

导航 -> Kibana -> Visualize -> Create visualization（创建可视化）

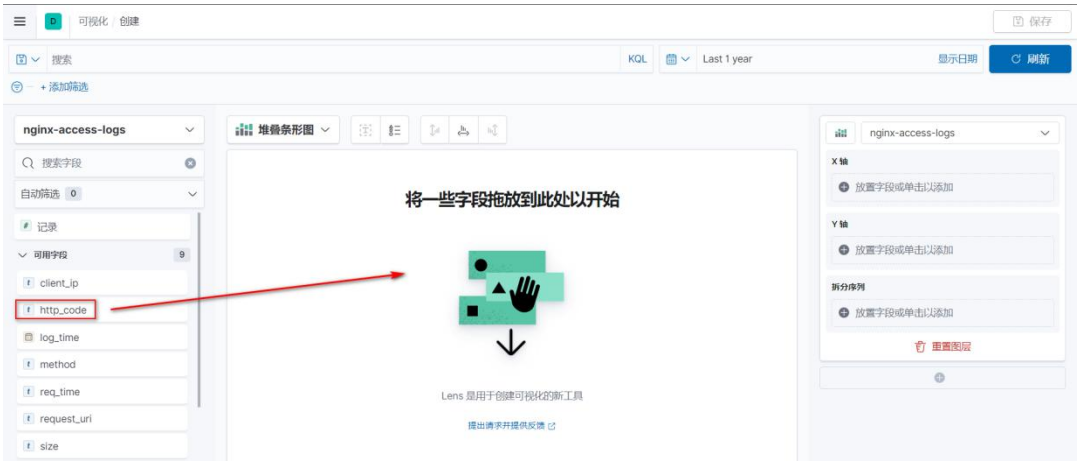


Create visualization（创建可视化）

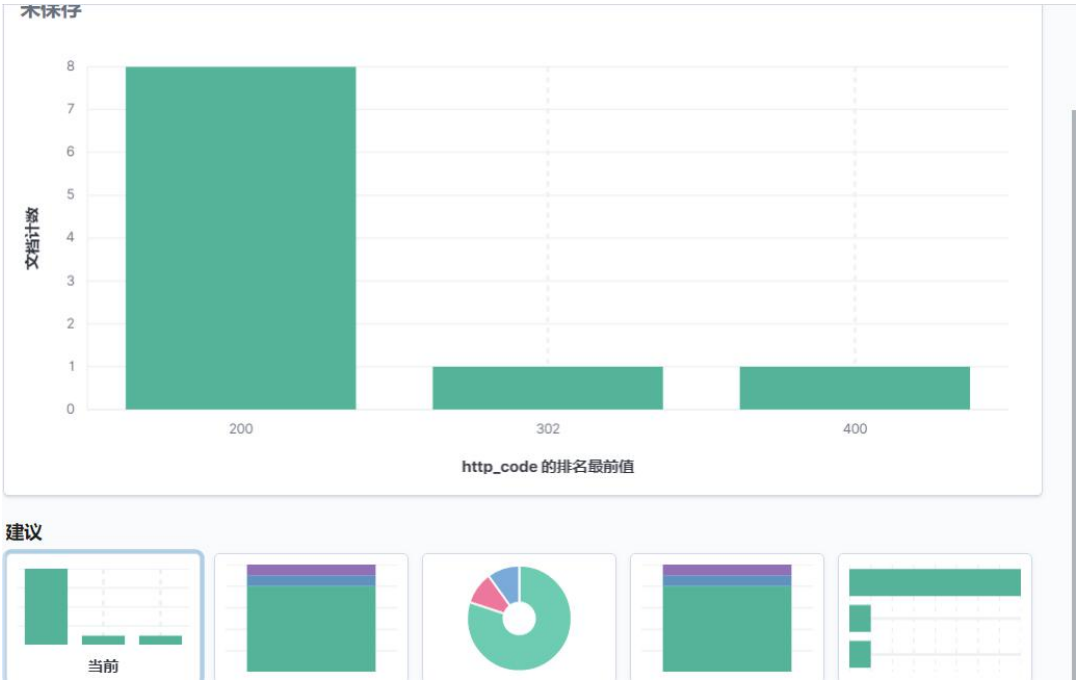
Select a visualization type（选择可视化类型）-> Lens 可视化

选择索引 nginx-access-logs -> 选择时间 Last 1 year

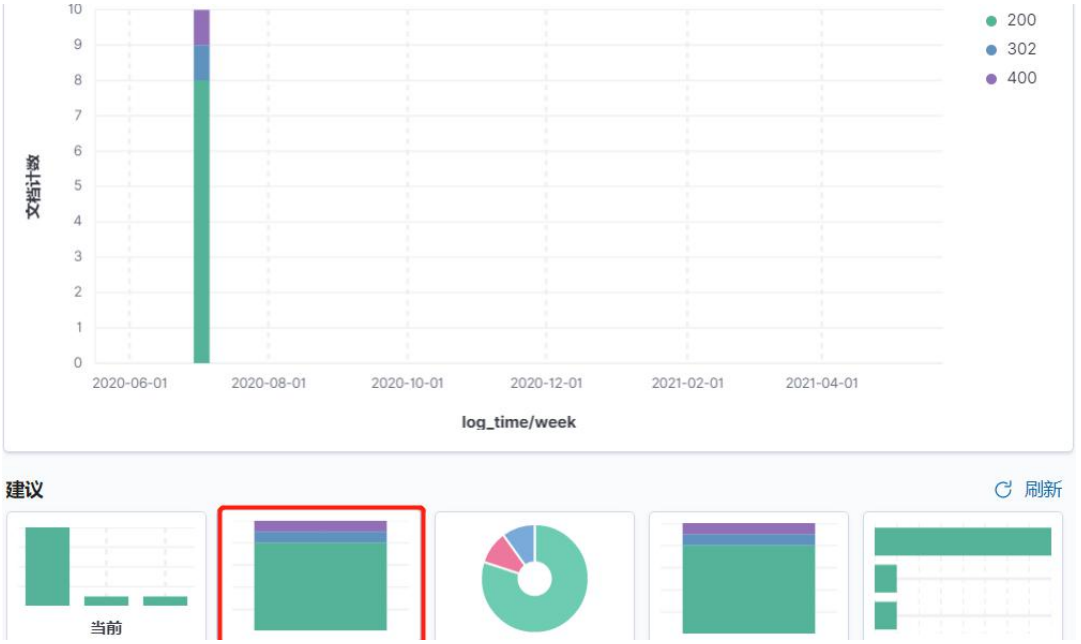
拖动字段 http_code 到中间

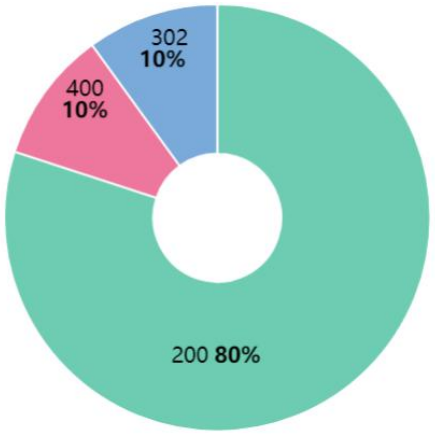


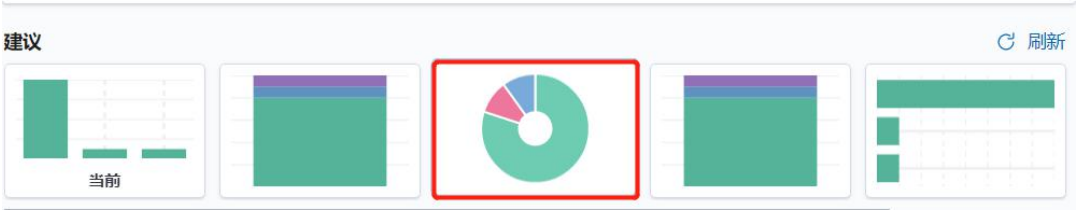
会自动根据此字段生成图表



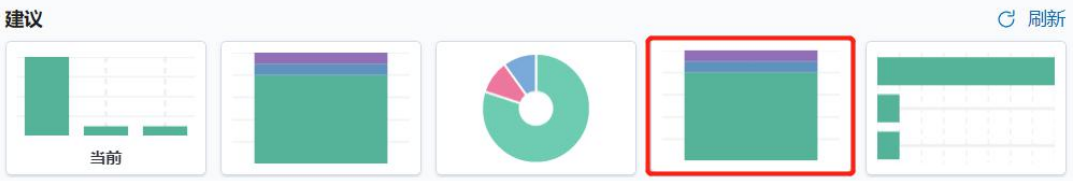
可以选择下方不同的图表进行展示



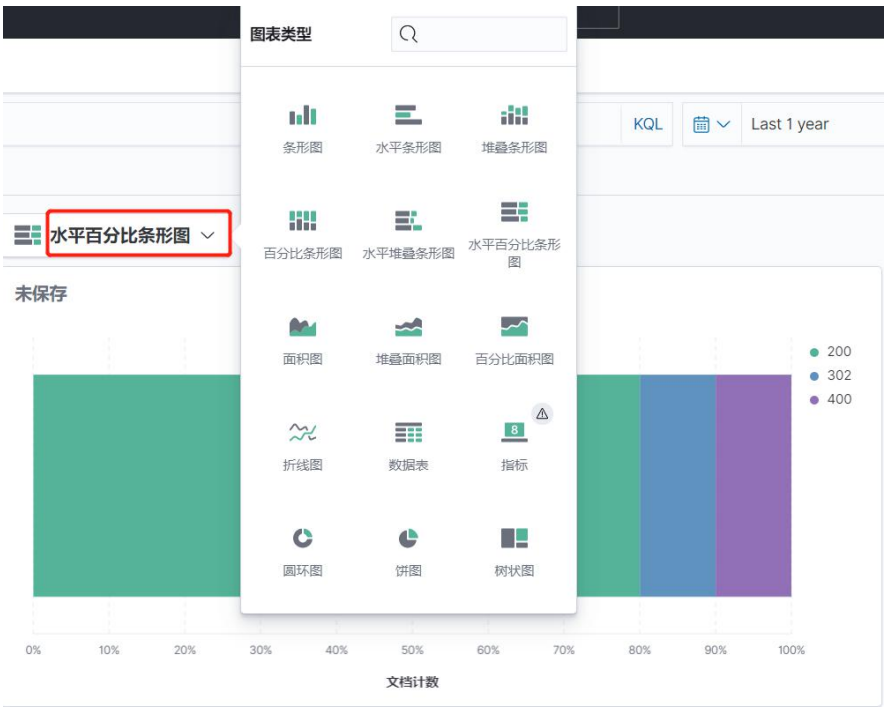

302
10%
400
10%
200 80%


建议
刷新
当前

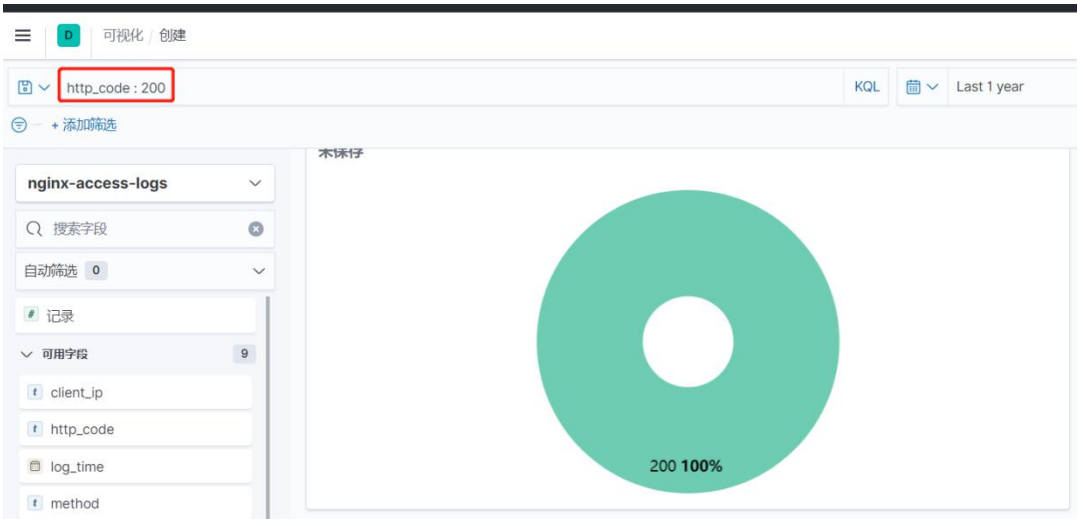

100%
90%
80%
70%
60%
50%
40%
30%
20%
10%
0%
文档计数
200
302
400


建议
刷新
当前

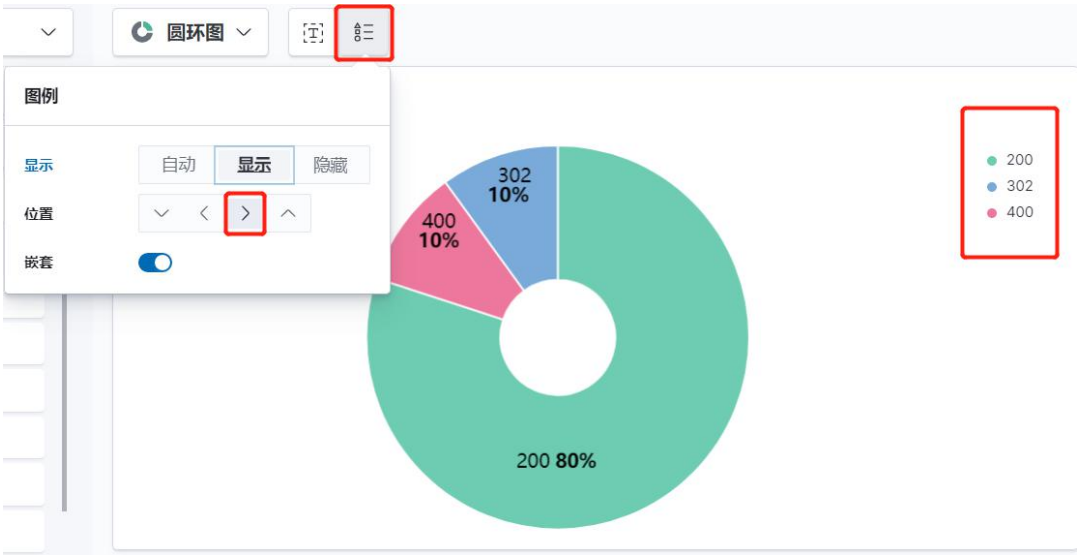
也可以在下拉菜单中选择不同的图像

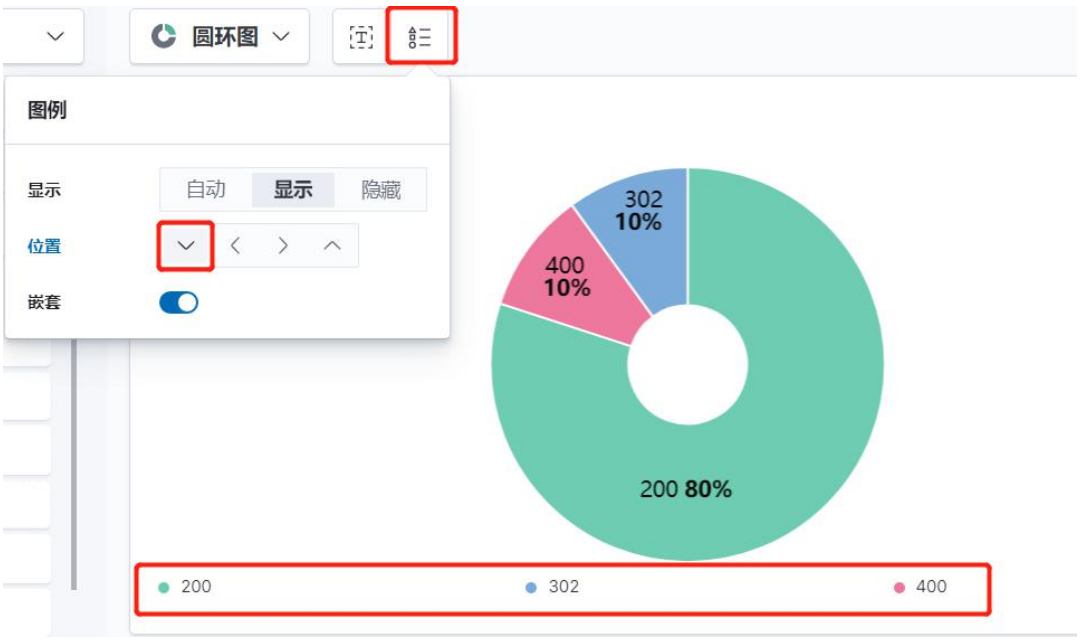


图表也会根据查询动态变化，比如要查询 http_code 的值为 200，图标就会变化为下图这样

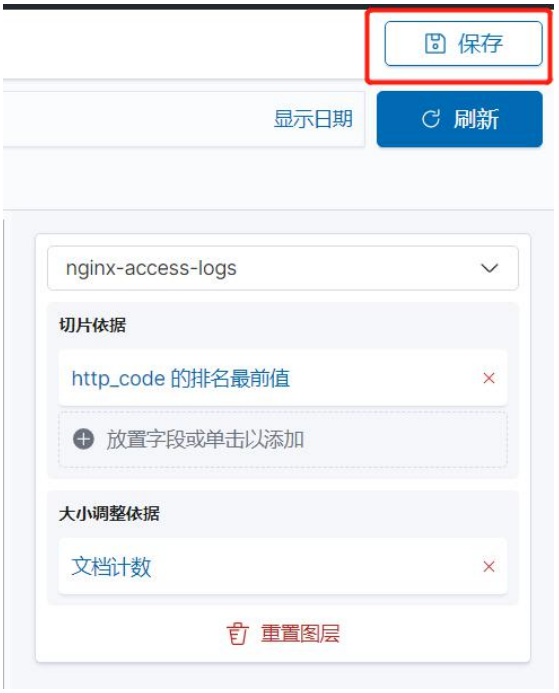


还可以选择图例展示的位置




圆环图
图例
显示
自动
显示
隐藏
位置
嵌套
302
10%
400
10%
200 80%
200
302
400

保存

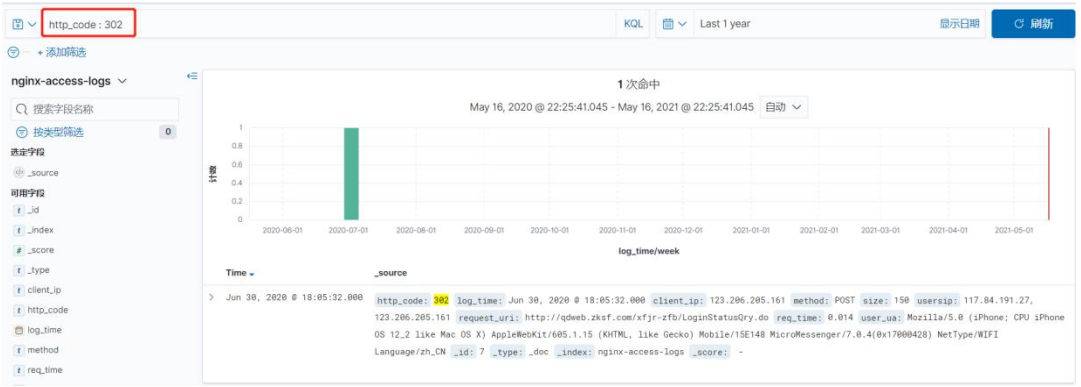

保存
显示日期
刷新
nginx-access-logs
切片依据
http_code 的排名最前值
放置字段或单击以添加
大小调整依据
文档计数
重置图层

KQL 查询

一般查询

查询 http_code 的值为 302 的数据

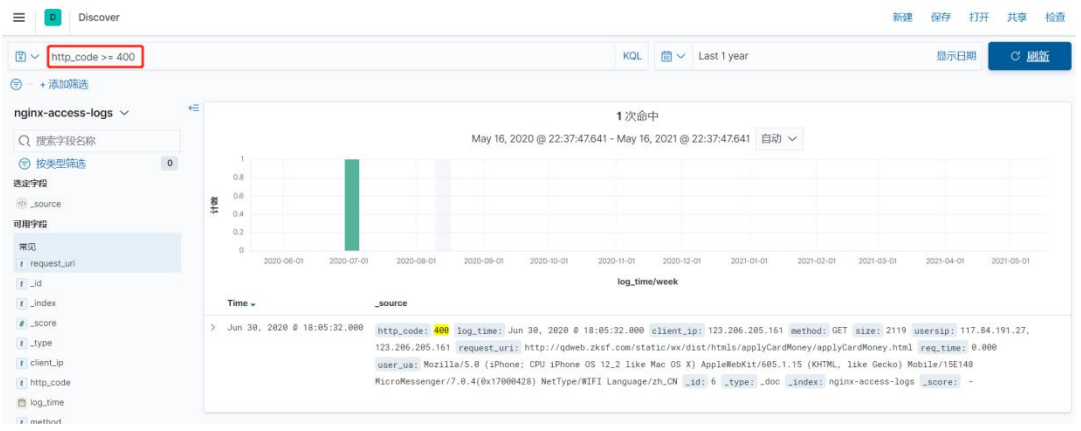
```
http_code : 302
```

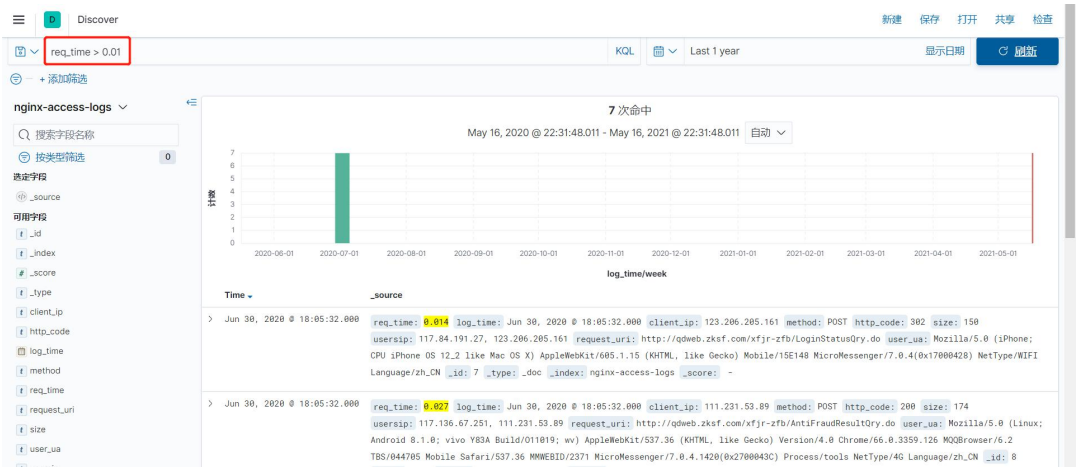


条件运算符查询

查询 http_code 的值是大于等于 400 的数据

http_code > = 400（代码格式会乱,正确的是> =中间没有空格）

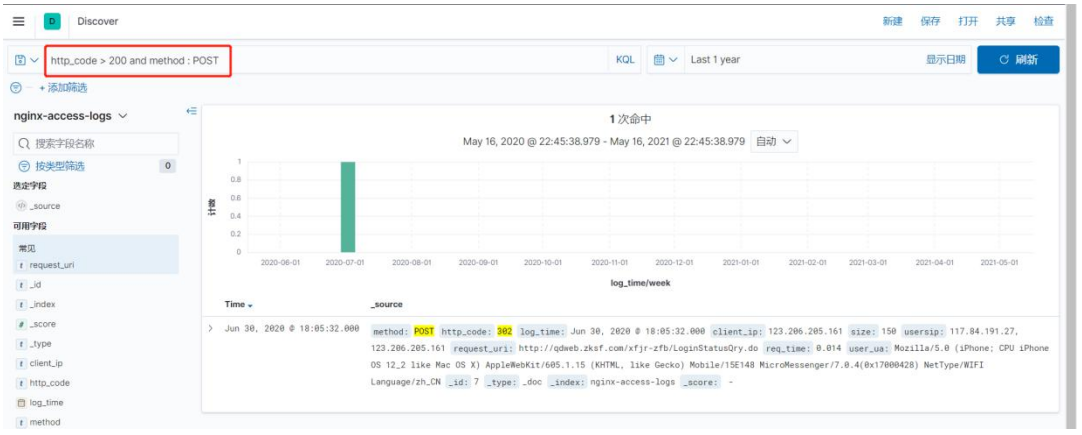




逻辑运算符查询

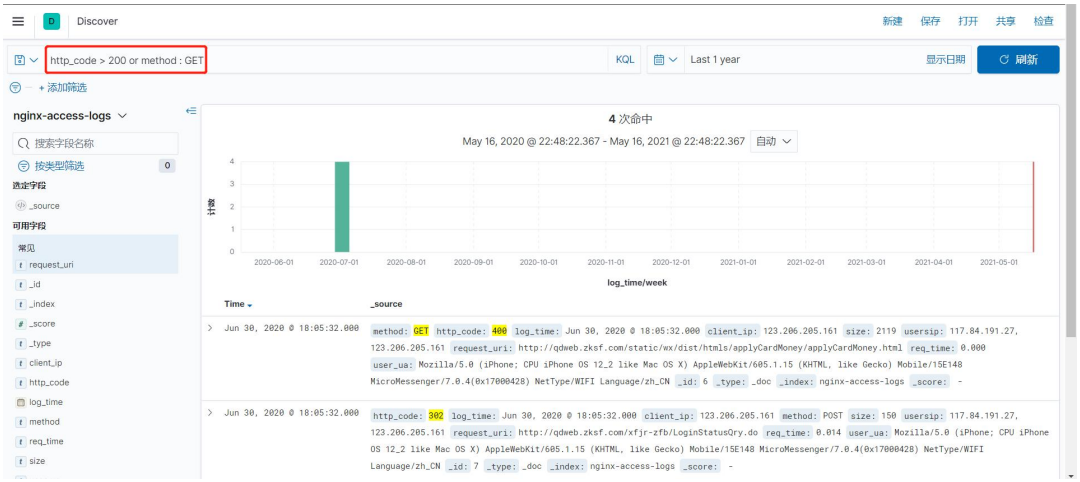
查询 http_code 的值为大于 200 并且 method 的值是 POST 的数据

```
http_code > 200 and method : POST
```



查询 http_code 的值为大于 200 或者 method 的值是 GET 的数据

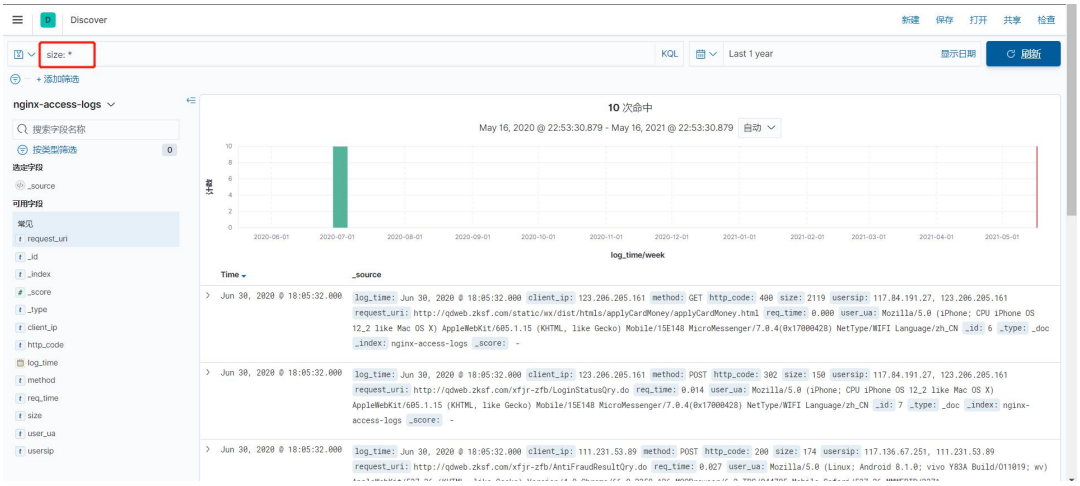
```
http_code > 200 or method : GET
```



通配符

查询某个字段的值存在的数据，存在则返回数据，不存在则返回为空

```
size: *
```



```
name: *
```

创作人简介：

郭海亮，落魄的运维搬砖人，工作近四年，爱折腾，对 k8s、Jenkins、Prometheus 等云原生技术有浓厚的兴趣，日拱一卒。

博客：https://github.com/ghl1024

3.5 进阶篇

3.5.1 跨集群操作

创作人：张刘毅

审稿人：吴斌

Elasticsearch 集群天然支持横向水平扩展，因此当业务规模扩大、对集群产生读写压力时，增加节点总是运维人员的“懒人选择”。但随着节点数增多，集群主节点要维护的 meta 信息也随之增多，这会导致集群更新压力增大，甚至无法提供正常服务。另外每个企业的数据中心都需要有灾备方案，在集群间同步数据，因为单点集群会存在隐患。

鉴于此，Elastic 官方提供了跨集群操作。主要包括：

- 跨集群搜索（CCS）：允许配置多个远程集群并且在所有已配置的集群中实现联合搜索。
- 跨集群复制（CCR）：允许跨多个集群复制索引，适合于做灾备方案和数据本地化场景。

跨集群配置

跨集群操作有两种配置模式连接远程的集群：嗅探模式（Sniff mode）或者代理模式（Proxy mode）。

- 在嗅探模式下，我们使用集群名称和种子节点列表注册远程集群。注册后，集群状态将被种子节点获取，该模式要求本地群集可以访问网关节点的发布地址。
- 在代理模式下，使用集群名称和单个代理地址注册远程集群。代理模式不需要远程集群节点具有可访问的发布地址。

我们可以在 Kibana 上动态配置远程集群，也可以在各个节点的 elasticsearch.yml 文件的上配置。

动态配置远程集群

我们在 Kibana 上使用 cluster update settings API 为每个节点动态配置远程集群。

例如：

```
PUT _cluster/settings
{
  "persistent": {
    "cluster": {
      "remote": {
        "cluster_one": {
          "seeds": [
```

```
            "127.0.0.1:9300"
          ],
          "transport.ping_schedule": "30s"
        },
        "cluster_two": {
          "mode": "sniff",
          "seeds": [
            "127.0.0.1:9301"
          ],
          "transport.compress": true,
          "skip_unavailable": true
        },
        "cluster_three": {
          "mode": "proxy",
          "proxy_address": "127.0.0.1:9302"
        }
      }
    }
  }
}
```

上面的配置中，当前集群是 cluster_one，一起联合远程访问的集群 cluster_two、cluster_three。其中 cluster_two 的连接方式是嗅探模式，cluster_three 的连接方式是代理模式，代理地址是 "127.0.0.1:9302"。

其中：

- transport.compress：网络传输的压缩参数
- transport.ping_schedule ：集群内部通信（tcp）的访问频率
- skip_unavailable：默认情况下如果请求中的任何集群都不可用则会返回错误。如果跳过不可用的集群，可以将 skip_unavailable 设置为 true。

以上这些参数是可以动态调整的，但必须要包括 seed 列表或者代理地址。

我们如果想关闭压缩、将 ping_schedule 由 30s 改成 60s 可以通过如下示例方式：

```
PUT _cluster/settings
{
  "persistent": {
    "cluster": {
      "remote": {
        "cluster_one": {
          "seeds": [
            "127.0.0.1:9300"
          ],
          "transport.ping_schedule": "60s"
        },
        "cluster_two": {
          "mode": "sniff",
          "seeds": [
            "127.0.0.1:9301"
          ],
          "transport.compress": false
```

```
            },
            "cluster_three": {
              "mode": "proxy",
              "proxy_address": "127.0.0.1:9302",
              "transport.compress": true
            }
          }
        }
      }
    }
```

静态配置远程集群

在节点的 elasticsearch.yml 中配置远程连接，只有在 YAML 文件中设置的节点，才能连接到远程集群，并处理远程集群请求。

举例：

```
cluster:
    remote:
        cluster_one:
            seeds: 127.0.0.1:9300
            transport.ping_schedule: 30s
        cluster_two:
            mode: sniff
            seeds: 127.0.0.1:9301
            transport.compress: true
```

```
            skip_unavailable: true
        cluster_three:
            mode: proxy
            proxy_address: 127.0.0.1:9302
```

其中，cluster_one，cluster_two 和 cluster_three 是表示与每个集群的连接的集群名称，用于区分本地索引和远程索引。

跨集群搜索

跨集群搜索可以针对一个或多个远程集群，运行单个搜索请求。例如，我们可以使用跨集群搜索，来过滤和分析存储，在不同数据中心的集群中的日志数据。

在 5.3.0 之前的版本，Elastic 官方提供了 Tribe Node 实现多集群访问的解决方案。

Tribe Node 是以 Client Node 的角色添加到集群中。但是由于不保留集群的 meta 信息，每次重启需要重新加载初始化。因此，在 5.3 版本中 Elastic 官方提供了 CCS 的功能，允许集群中的任何节点可以联合查询。

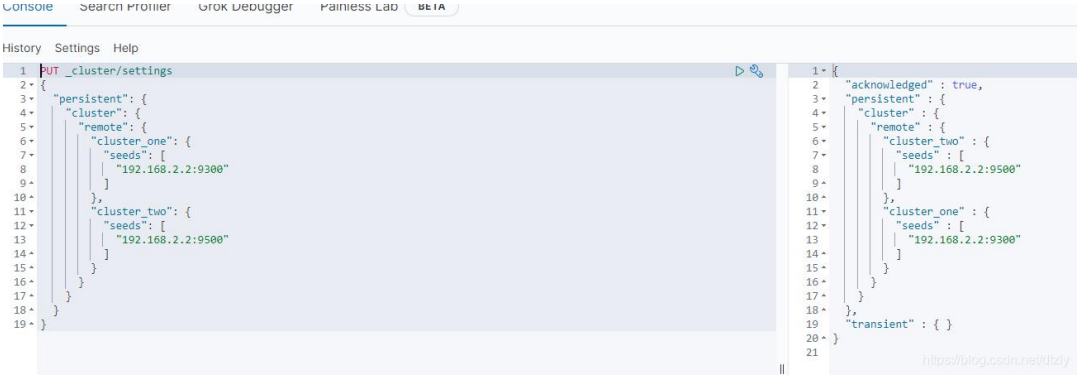
快速入门

下面以两个集群的跨集群搜索为例。我们预先启动了两个集群：cluster1、cluster2，当前集群是 cluster1。现在的任务是联合远程访问的集群 cluter2 进行跨集群搜索。

我们分别在两个集群上动态配置 remote cluster。

注意：在 seeds 列表中填写的是集群节点间通信的 TCP 端口而不是 HTTP 端口。

```
PUT _cluster/settings
{
  "persistent": {
    "cluster": {
      "remote": {
        "cluster_one": {
          "seeds": [
            "192.168.2.2:9300"
          ]
        },
        "cluster_two": {
          "seeds": [
            "192.168.2.2:9500"
          ]
        }
      }
    }
  }
}
```



在 cluster1 中插入数据：

```
PUT esfighttogether/_doc/1
{
  "teamname":"team 10"
}
```

在 cluster2 中插入数据：

```
 PUT esfighttogether/_doc/1
{
  "teamname":"team 1"
}

PUT esfighttogether/_doc/2
{
  "teamname":"team 7"
}
```

在两个集群上分别验证数据。因为写入时 Elasticsearch 自带的默认分词器会对数据进行分词，我们通过 team 就可以查询到所有数据。

查询语句如下：

```
GET esfighttogether/_search
{
```

```
  "query": {
    "match": {
      "teamname": "team"
    }
  }
}
```

执行跨集群搜索：

```
GET cluster_one:esfighttogether,cluster_two:esfighttogether/_search?timeout=5m
{
  "query": {
    "match": {
      "teamname": "team"
    }
  }
}
```

如上图所示，通过 CCS 查询到 3 条数据：cluster_one 的一条数据 team 10 以及 cluster_two 的两条数据 team 1 和 team 7，和之前写入的数据一致。

跨集群复制

跨集群复制是将特定索引，从一个 Elasticsearch 集群复制到另外的集群。

一般用于以下场景：

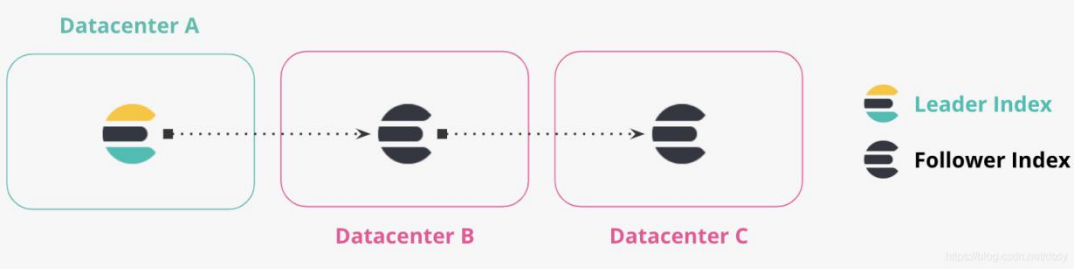
- 灾备：数据中心服务中断时，可以继续响应用户的搜索请求，防止用户大的查询影响到写入吞吐。
- 数据本地化：将数据复制到更靠近用户的位置以降低搜索延迟。

跨集群复制使用主动-被动模型，我们将索引写入领导者索引，数据会被复制到一个或多个只读跟随者索引。在将跟随者索引添加到集群之前，我们需要配置包含领导者索引的远程集群。

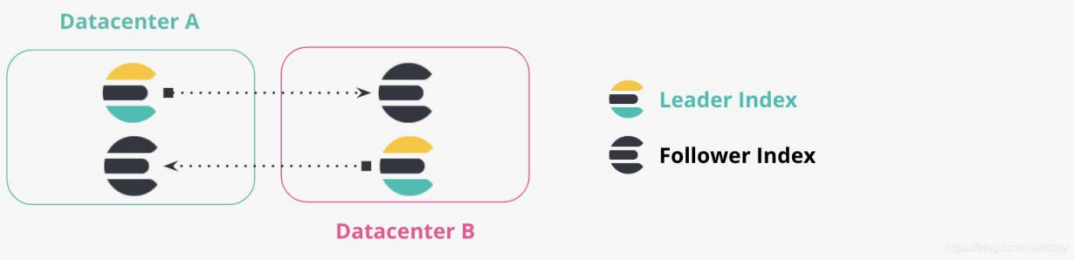
当领导者索引有写入请求时，跟随者索引从远程集群上的领导者索引中拉取增量数据。我们可以手动创建跟随者索引，或配置自动跟随模式（auto-follow patterns），创建跟随者索引。

我们可以单向或双向配置 CCR：

- 在单向配置中，一个集群仅包含领导者索引，而另一个集群仅包含跟随者索引。



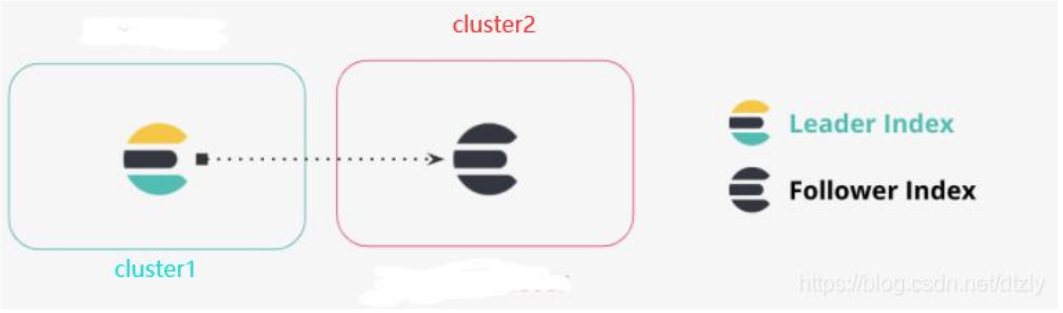
- 在双向配置中，每个集群都包含领导者索引和跟随者索引



快速入门

我们在服务器上启动两个集群来模拟不同地区数据中心的集群：

- “cluster1”：在端口 9200 上运行。我们会将文档从 cluster1 复制到 cluster2。
- “cluster2”：在端口 9400 上运行。cluster2 将维护一个来自 cluster1 的复制索引。

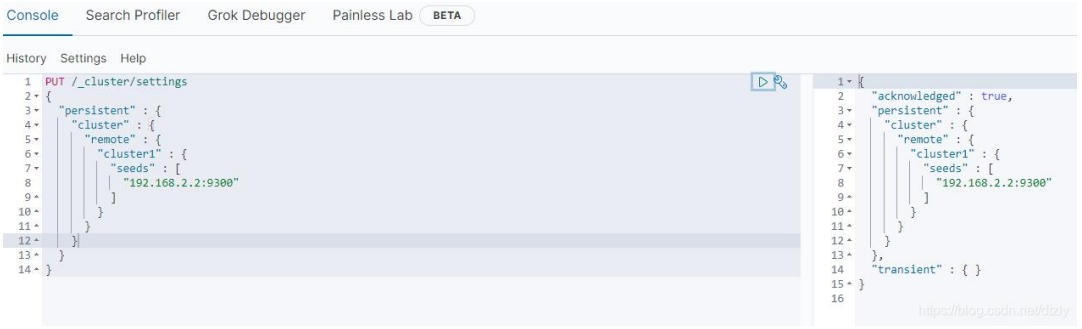


配置远程集群

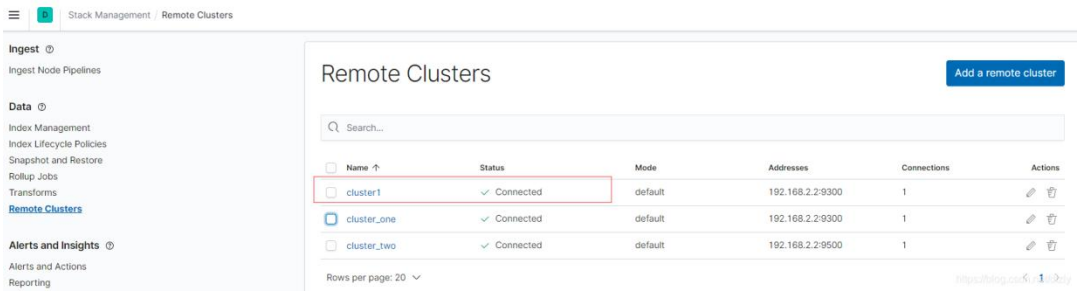
我们选择 CCR 的单向配置，因此 CCR 是基于拉取模式，所以我们只需要确保 cluster2 连接到 cluster1 ，而不需要指定从 cluster2 到 cluster1 的连接。

下面让我们通过 cluster2 上的 API 调用来定义 cluster1:

```
PUT /_cluster/settings
{
  "persistent": {
    "cluster": {
      "remote":{
        "cluster1":{
          "seeds":["192.168.2.2:9300"]
        }
      }
    }
  }
}
```



Kibana 中远程集群管理的 UI，单击左侧导航面板中的 “Management”（齿轮图标），然后点击“Stack Management” ，导航到 Elasticsearch 部分中的“Remote Clusters”（远程集群）。



创建要复制的索引：

```
PUT esfightalone
{
  "settings": {
    "index": {
      "number_of_shards": 1,
      "number_of_replicas": 0,
      "soft_deletes": {
```

```
            "enabled": true
          }
        }
      },
      "mappings": {
        "properties": {
          "name": {
            "type": "keyword"
          }
        }
      }
    }
```

CCR 的 Leader 索引需要使用软删除（soft_deletes），无论何时删除或更新现有文档，都可以将操作历史记录保留在领导者分片上，等重新操作历史记录时，供追随者分片任务使用。

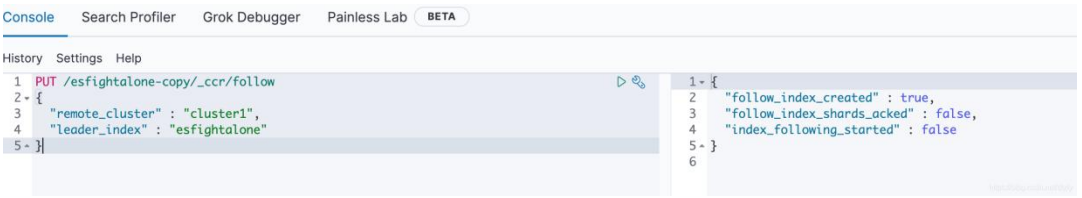
当追随者分片从领导者复制数据时，会在领导者分片上留下标记，这样领导者就知道追随者在历史记录中的所在位置。基于这些标记，领导者分片将会保留这些操作，直到分片历史记录保留到设定时间结束（默认为 12 个小时）。

我们已经为远程集群创建了一个别名，并创建了一个我们要复制的索引，接下来我们启动复制。在 cluster2 上，执行：

```
    PUT /esfightalone-copy/_ccr/follow
    {
      "remote_cluster" : "cluster1",
```

```
    "leader_index" : "esfightalone"
}
```

注意，复制索引是只读的，不能接受写操作。至此，我们已配置了要从一个 Elasticsearch 集群复制到另一个集群的索引。



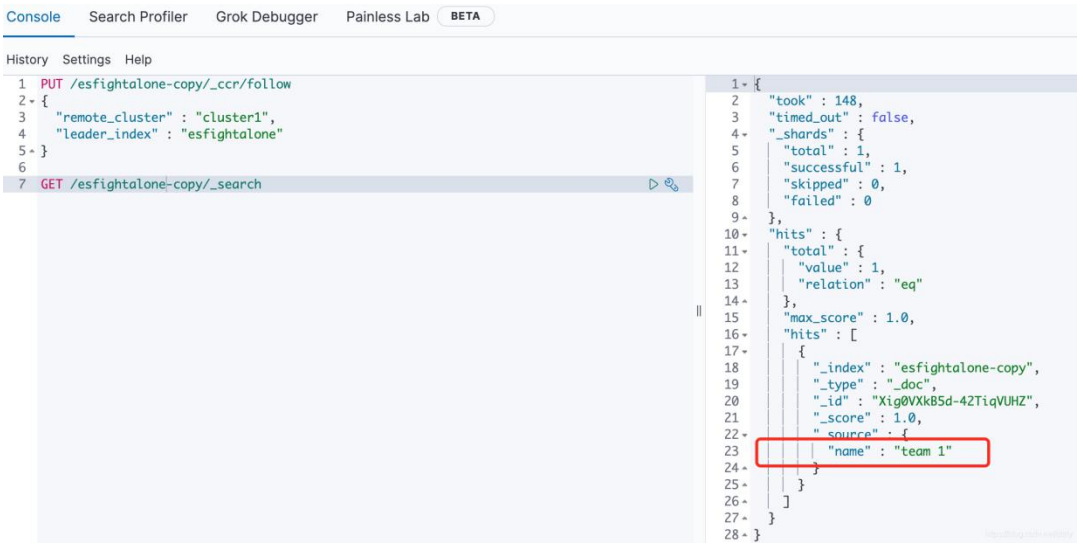
测试 CCR 复制

我们在 cluster1 上写入数据：

```
POST /esfightalone/_doc
{
  "name" :"team 1"
}
```

然后在 cluster2 上查询，验证索引数据是否同步，发现此时数据已实时同步到 cluster2 中了：

```
GET /esfightalone-copy/_search
```



CCR 属于 Elastic 官方的白金付费（Platinum License）的功能。一般企业还是会选择自研数据同步工具，来同步集群间的数据。

不过，需要体验的小伙伴可以在 Elastic 官网或阿里云 Elasticsearch 申请 30 天免费使用或者在自己的本地安装中启用试用功能。

创作人简介：

张刘毅，存储研发工程师，曾经做过 AI 大数据平台研发，负责过 80+ES 集群。目前专注于生物医药和大数据。

博客：https://blog.csdn.net/dtzly

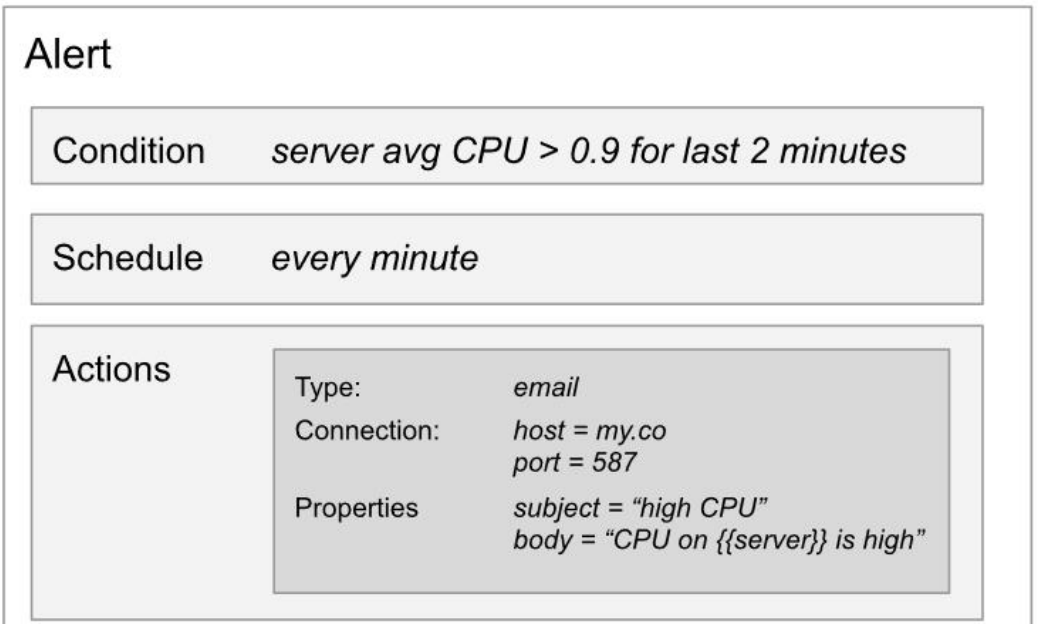
3.5.2 Kibana 的 Alert

创作人：金端

审稿人：周海清

Kibana 的 Alert 模块主要用于 Elastic Stack 的监控告警。以一种相对较低的使用成本，将复杂的查询条件，编辑完成后监控不同的 Elastic Stack 的技术产品中产生的数据，最终把符合条件的告警信息以需要的方式反馈给用户。

Alert 的组成



Alert 主要由三个部分组成：

Condition 条件

也是 Alert Type，检测需要执行的查询或者统计。

Condition 的概念执行主要由 alert type 承担。Alert Type 通过简单的参数设置将涉及 Elasticsearch 的查询结果和其它数据源的复杂计算完美实现。比如：监控的程序 APM 数据 2 分钟内 CPU 使用均值高于 0.9；某个业务数据索引中，10 分钟内购买失败次数占比超总量的 30%等等.

Schedule 检测周期

Alert 的执行周期。Schedule 的设置按照每隔多久循环来设置，从每秒到每月不等，但并不能设置具体哪月哪天。

Action 告警动作

即满足告警条件后需要执行的操作，主要是将需要的告警信息发送给第三方系统，在 Kibana 后台执行。

Action 可以分解为三个要素：

- action type，发送的第三方系统类型定义。
- connection，告警发送的连接信息，比如 email 的 host 和端口等
- properties，告警信息所需要引用的参数值。

Alert 与 Elasticsearch 的 Watcher 不同的是，Alert 运行在 Kibana 而不是 Elasticsearch，相关任务数据也是存储在 Kibana 的索引中，而 Elasticsearch 的 Watcher 数据则是在 Watcher 的索引中。

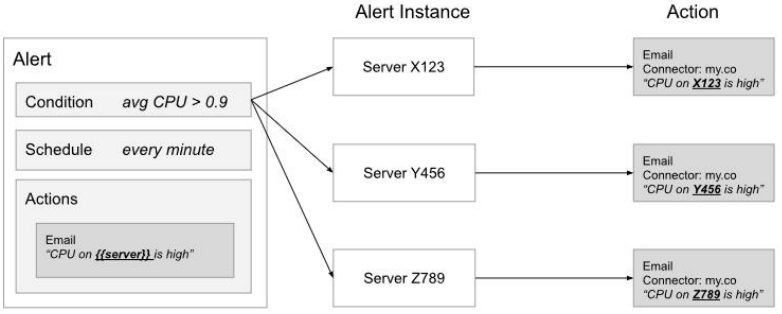
在更高的层次上，Kibana Alert 允许跨用例（如 APM、Metrics、Scurity 和 Uptime）进行丰富的集成。预先打包的 Alert Type 简化了设置，隐藏了复杂的检测细节，同时提供了跨 Kibana 的一致接口。

Alert Instances 的概念和抑制重复告警

在检查一个 condition 条件时，Alert 可能会识别该条件的多次出现。每出现一次符合 condition 条件的情况，Kibana 就生成一个 Alert Instances，即警报实例。那么 Kibana 分别跟踪每个警报实例，并对每个实例采取行动。

重复告警

以下面的图示例，将每个平均 CPU 为> 0.9 的服务器作为 Alert Instance 进行跟踪。然后将每个超过阈值的服务器都将发送单独的电子邮件。



那就会带来一个问题，当 Alert Instances 过多的时候，就会造成大量通知重复发送，即 Alert Noise 的现象。

比如一个警报每分钟监控三个服务器的 CPU 使用情况> 0.9，就发送邮件通知工作人员:

- 第一分钟：服务器 X123 的 CPU > 0.9。

  其中一封邮件发送通知工作人员服务器 X123 的 CPU 过高。

- 第二分钟：X123 和 Y456 的 CPU > 0.9。

  发送了两封邮件，一封是关于 X123 的，一封是关于 Y456 的。

- 第三分钟：X123, Y456, Z789 的 CPU > 0.9。

  发送了三封邮件，分别是 X123，Y456，Z789。

在上面的例子中，对于相同的条件，在 3 分钟的时间内，向服务器 X123 发送了 3 封邮件。

抑制重复告警

Kibana 针对这个情况做了抑制重复通知的优化，主要是通过设置通知间隔来抑制重复多余的告警。比如在上面的例子中，将警报重新通知间隔设置为 5 分钟，那么 Alert 发送通知的情况则如下:

- 第一分钟：服务器 X123 > 0.9

  邮件发送报告服务器 X123 的 CPU 过高；

- 第二分钟：X123 和 Y456 > 0.9

  邮件发送报告服务器 Y456 的 CPU 过高；

- 第三分钟：X123, Y456, Z789 > 0.9

  邮件发送报告服务器 Z789 的 CPU 过高。

当然过了五分钟后，如果服务器 X123 > 0.9 还是存在，那么继续会发送邮件，报告服务器 X123 的 CPU 过高。

Kibana Alert 的实现机制

Kibana Alert 将 Alert Check 的信息和 Action 的信息，持久化在 Elasticsearch 在后台执行。这有两个主要好处:

- 持久性：所有的任务相关的信息都存储在 Elasticsearch 中，所以如果 Kibana 重新启动，Alert 和 Action 将从它们停止的地方恢复；
- 伸缩性：多个 Kibana 实例可以从 Elasticsearch 中读取和更新相同的任务队列，允许 Alert 和 Action 跨实例分布。如果现有的 Alert 执行数量超出了现有的 Kibana 实例的容量上限，可以增加额外的 Kibana 实例。

Kibana 后台任务的执行机制

- 每隔 3 秒轮循 Elasticsearch 任务索引以查找过期任务；
- 任务执行后在 Elasticsearch 索引中更新，使用乐观并发控制来防止冲突；
- 任务在 Kibana 服务器上运行。每个 Kibana 实例最多可以运行 10 个并发任务，因此每个间隔最多可以声明 10 个任务；
- 对于重复后台检查的 Alert，任务完成后将按照检查间隔再次调度。

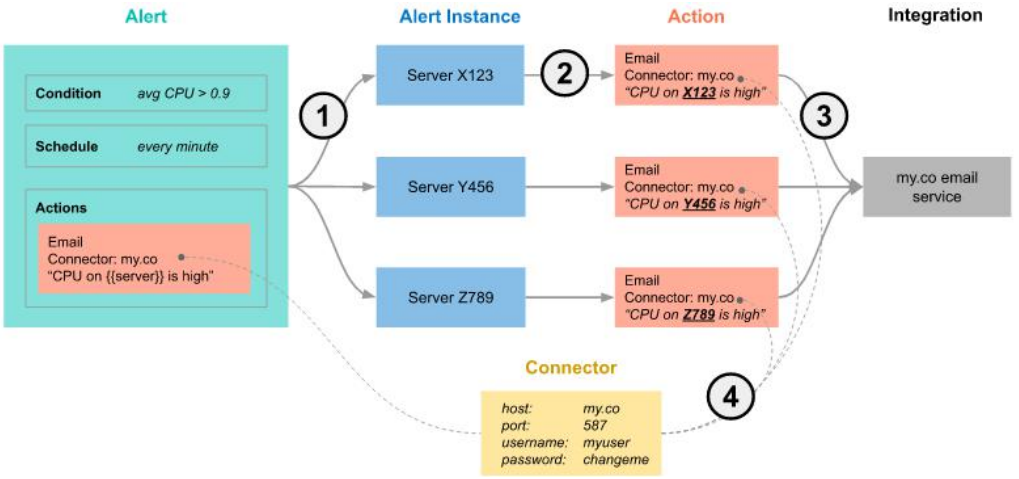
因为每 3 秒轮询一次任务，并且每个 Kibana 实例只能同时运行 10 个任务，所以 Alert 和 Action 任务可能会在以下情况延迟运行：

- 警报使用较小的检查间隔。最低间隔时间可能是 3 秒，但建议间隔时间为 30 秒或更高。
- 许多警报或操作必须同时运行。在这种情况下，挂起的任务将在 Elasticsearch 中排队，并且每隔 3 秒从队列中取出 10 个任务。
- 长时间运行的任务占用槽位的时间较长，留给其他任务的槽位较少。

完整的 Alert 流程

Alert 由 Condition（条件）、Action 和 Schedule 组成。当条件满足时，就会创建警报实例来呈现和调用操作。为了使 Action 设置和更新更容易，Action 包含了与第三方连接交互的 Connector 。

下面的例子将这些概念联系在一起:



- 只要警报的条件得到满足，就会创建一个警报实例。这个示例检查平均 CPU 为 > 0.9 的服务器。三个服务器满足条件，因此创建了三个实例；
- Action 执行时，警报中设置的模板将被实际值填充。在这个示例中，创建了三个操作，模板字符串 {{server}} 被替换为每个实例的服务器名；
- Kibana 调用这些 Action，将它们发送给第三方集成，比如邮件服务；
- 发送这些信息时，Action 会结合 Connector 中设置的信息发送。比如：邮件的 host/port/用户名/密码。

如何配置 Alert

目前 Kibana 提供了一种内置的警报类型：索引阈值类型（index threshold）。索引阈值警报类型，允许你指定要查询的索引、聚合字段和时间窗口。但底层 Elasticsearch 查询的详细信息是隐藏的。根据设定的查询条件，将结果与阈值进行比较，并在满足阈值时进行后续的调度执行。

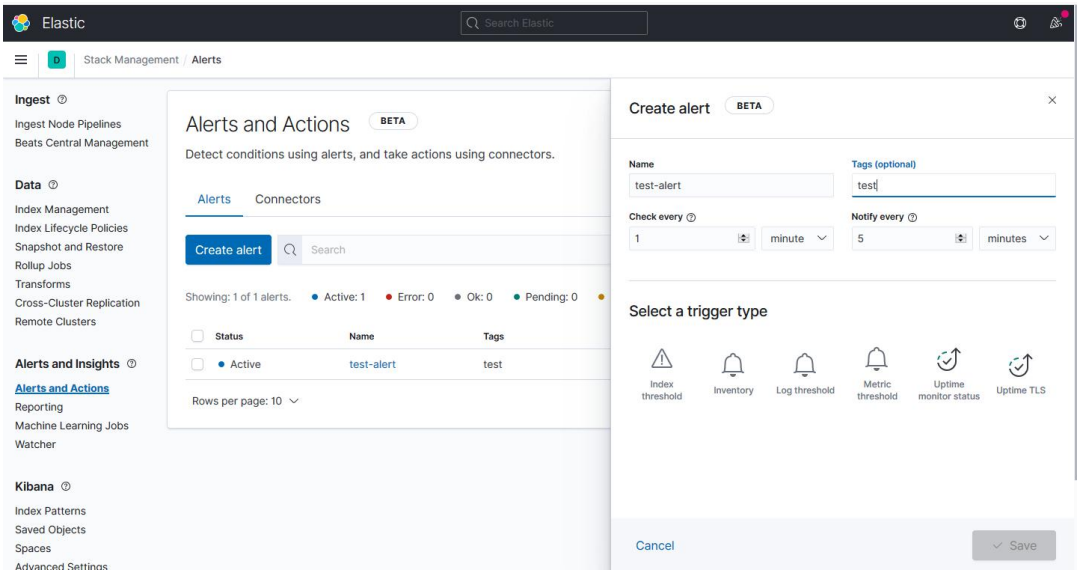
操作示范

在 Stack management 的 Alert 中 Create alert，新建一个名为 test-alert 的告警。该告警用于检查索引 test-es 中 fail_num 的累计数量，test-alert 每分钟检查一次，最多 5 分钟告警通知一次，标签为 test。

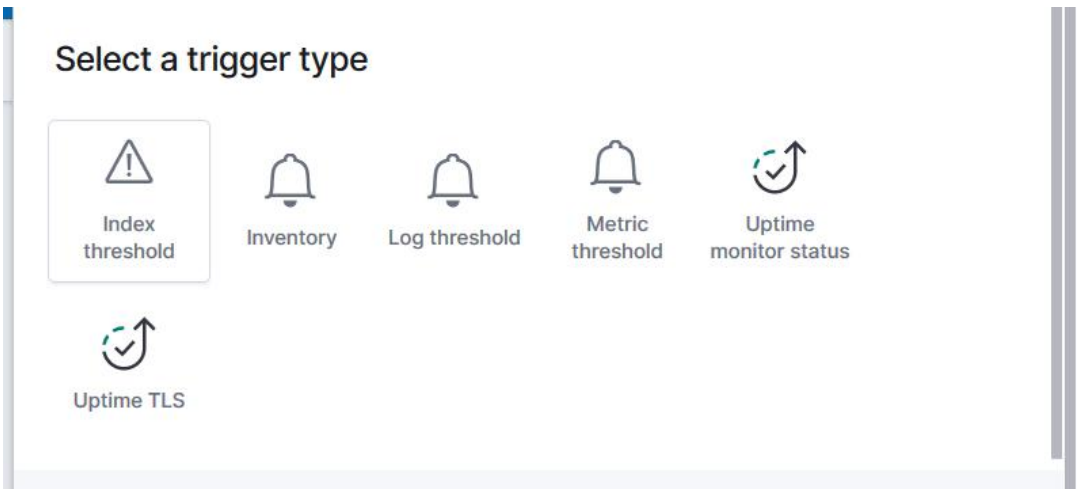
索引 test-es 的数据如下：

```
PUT test-es

POST test-es/_mapping
{
        "properties" : {
          "@timestamp" : {
            "type" : "date",
            "format" : "yyyy/MM/dd HH:mm:ss||yyyy/MM/dd||epoch_millis"
          },
          "fail_num" : {
            "type" : "long"
          },
          "user" : {
            "properties" : {
              "id" : {
                "type" : "text",
                "fields" : {
                  "keyword" : {
                    "type" : "keyword",
```

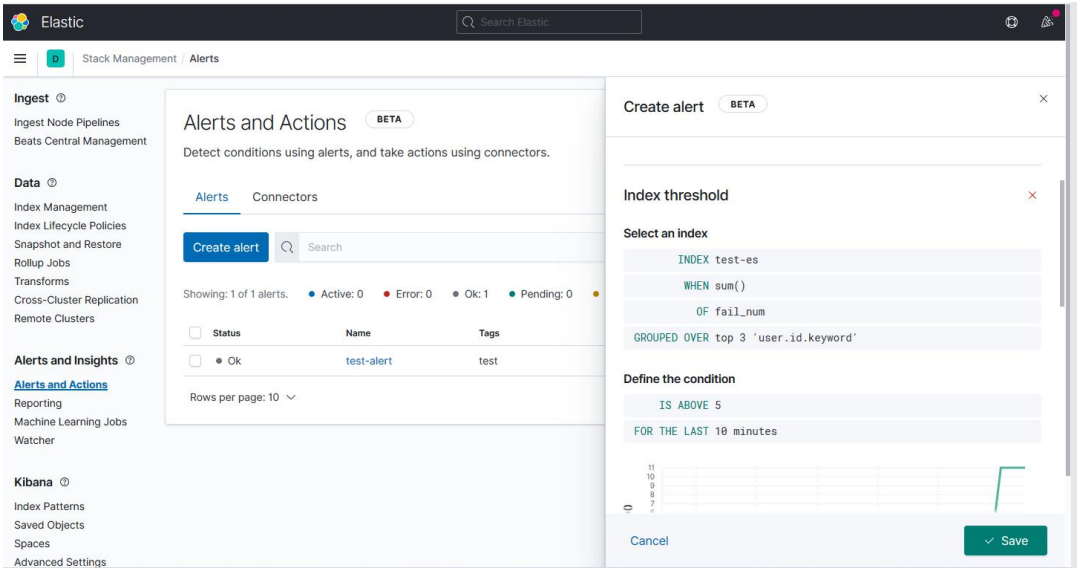
```
                    "ignore_above" : 256
                }
            }
        }
    }
}
}

POST _bulk
{ "create" : { "_index" : "test-es", "_id" : "1" } }
{ "@timestamp" : "2021/04/20 23:30:30" ,"user.id":"may","fail_num":"1"}
{ "create" : { "_index" : "test-es", "_id" : "2" } }
{ "@timestamp" : "2021/04/20 23:33:31" ,"user.id":"may","fail_num":"4"}
{ "create" : { "_index" : "test-es", "_id" : "4" } }
{ "@timestamp" : "2021/04/20 23:46:30" ,"user.id":"jack","fail_num":"1"}
{ "create" : { "_index" : "test-es", "_id" : "5" } }
{ "@timestamp" : "2021/04/20 23:50:30" ,"user.id":"may","fail_num":"6"}
{ "create" : { "_index" : "test-es", "_id" : "6" } }
{ "@timestamp" : "2021/04/20 23:49:30" ,"user.id":"jack","fail_num":"3"}
{ "create" : { "_index" : "test-es", "_id" : "7" } }
{ "@timestamp" : "2021/04/20 23:49:30" ,"user.id":"jack","fail_num":"3"}
{ "create" : { "_index" : "test-es", "_id" : "8" } }
{ "@timestamp" : "2021/04/20 23:50:30" ,"user.id":"bill","fail_num":"9"}
{ "create" : { "_index" : "test-es", "_id" : "9" } }
{ "@timestamp" : "2021/04/20 23:52:30" ,"user.id":"jack","fail_num":"1"}
{ "create" : { "_index" : "test-es", "_id" : "10" } }
{ "@timestamp" : "2021/04/20 23:53:30" ,"user.id":"jack","fail_num":"1"}
```



选择 Index threshold



配置 condition 条件为：监控索引 test-es 中 10 分钟内，如果 fail_num 总计超过 5 次则告警。

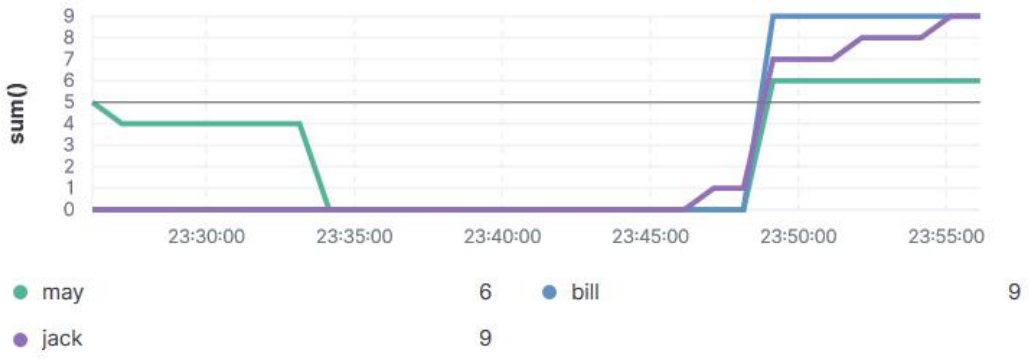


其中，WHEN 条件可选择为 count，average，sum，min 和 max。

count 为统计文档数，不需要填写字段，其余方法会自动匹配出可计算类型的字段，比如 long 类型。

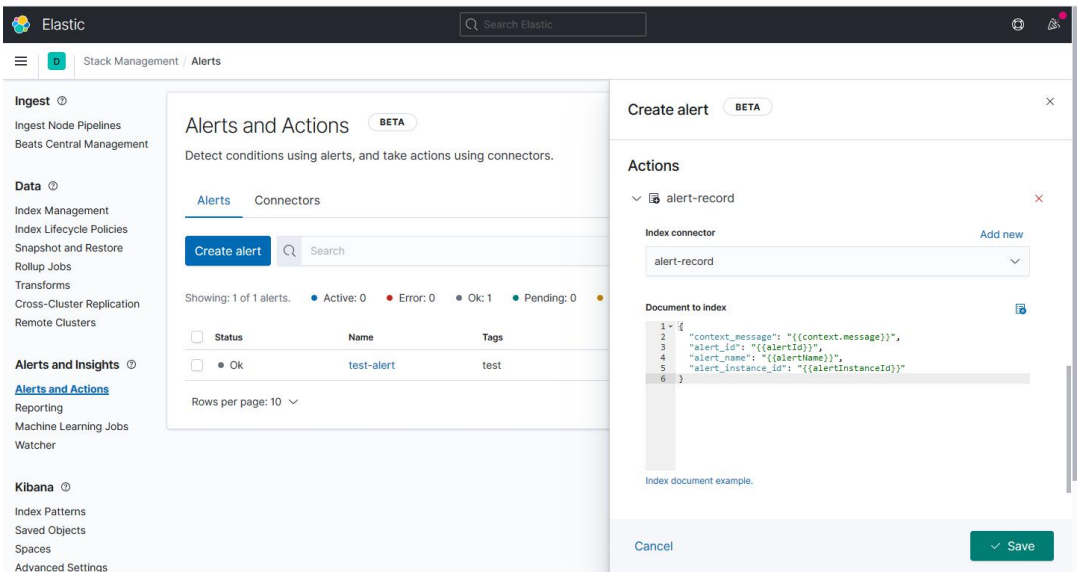
OVER 条件可以配置聚合全部文档或者分组。如果使用了分组，即 top，那么当每个组超过阈值时，将为每个组创建一个 Alert Instance。在配置中，top 会设定分组数量，限制高基数字段上的实例数量。比如上图中只检查 user.id.keyword 数量最多的 3 组。

相关的查询结果会有一张时序图来体现，如下图：

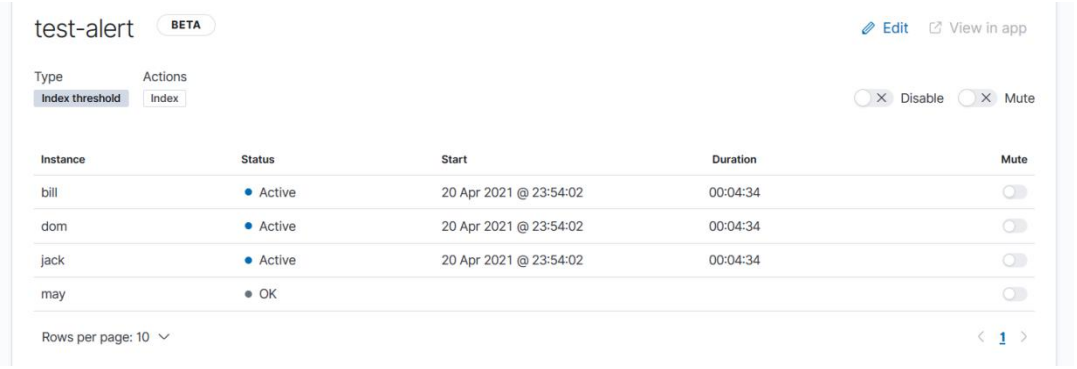


从图中看出前三个是 may 6 次、jack 9 次、bill 9 次。

再设置一个 Action，此处是写进索引 alert-record



保存后，过段时间 Alert 的详情里就有了 Alert Instance 的相关信息。



其中 Status 为 Active 的是待执行 Action 操作的。不管 Alert Instance 在被监控判断的期间是否被静音(Mute)，最终状态都为 OK。

关于 Actions

预设置 Connector 和 Action

可以在 kibana.yml 中预设值 Connector 或者 Action Type。

预设 Connector 需要将配置和相关证书设置清楚，不可以数组对象的形式设置，且配置完以后不可修改。相比之下在 Kibana 启动后，再设置 Connector 相对灵活许多。

Connector 和 Action 的具体配置可参考官方网站。

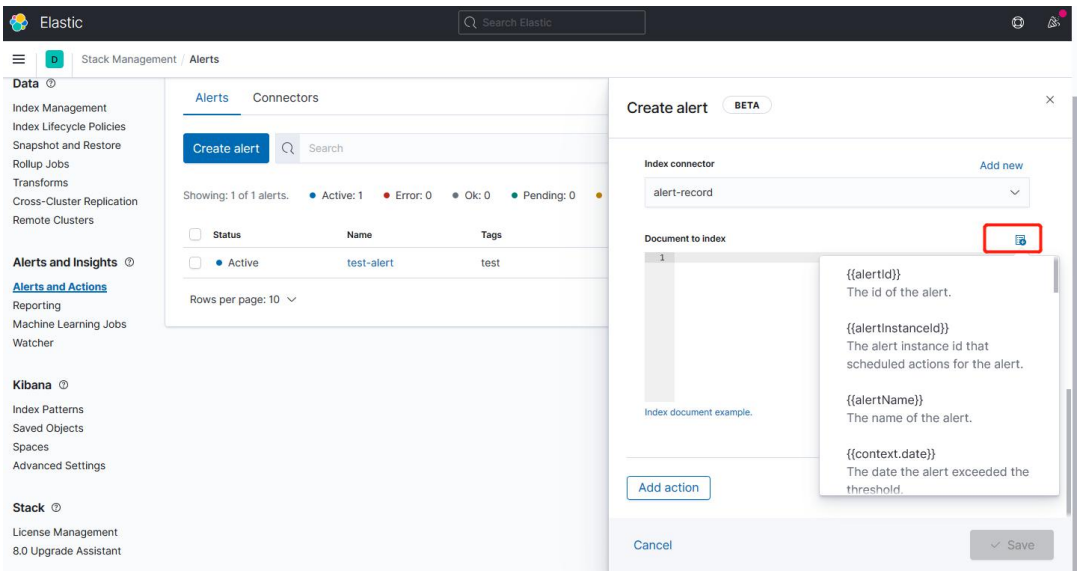
Action 的变量传入

在设置 Action 时，根据 Action Type 的不同，会设置不同的 properties。比如，email 配置 Subject 和 Message；Index 提供 document。具体细节参考每个类型的具体 action 配置。

虽然各个 Properties 不一样，但是在配置时都可以把 Alert 中监控到的数据，作为变量传入报警信息中。使用 Mustache 模板语法 {{variable name}}，可以在检测到一个 Condition 时将警报值传递给一个 Action。

可用的变量因 Alert Type 不同而不同，可以使用 "Add variable" 按钮来访问列表。

下图是 email 的可传入参数：



3.5.3 Rollup

创作人：杨景江

审稿人：朱永生

汇总作业（ rollup jobs ）是周期性执行的任务，通过汇总作业，可以将某些索引中的数据进行周期性自定义化聚合，然后将聚合后的数据写入到新的索引中，整个流程叫做 Rollup 。

使用场景：

汇总历史数据：

由于历史数据数据量大，占用磁盘成本高，相关业务方只关心近期几天的原始数据，历史数据不关心原始数据，只关心固定指标统计。为了节省成本，就可以通过 Rollup 操作将历史数据进行汇总，写入到新的索引，之后将历史索引删除（ ILM 功能），进而节省大量成本

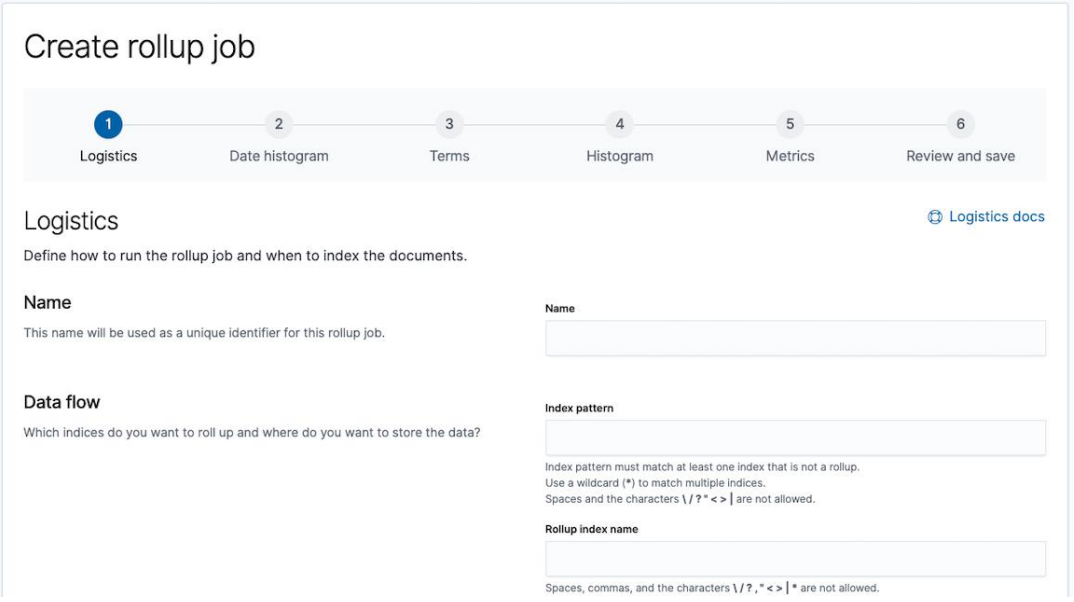
转换最佳时间：

由于数据量或机器硬件等原因，导致实时聚合查询耗时较长，可以通过在夜间或者准实时进行 Rollup 操作，将前一天索引或者几分钟前的数据进行汇总，写入到新索引（将毫秒级别数据汇总，转换为秒级甚至分钟级别），用户查询 Rollup 后新索引的数据，进而提升查询效率。

汇总历史数据功能限制：

汇总功能只允许使用以下聚合方式对字段进行分组：

- Date Histogram aggregation
- Histogram aggregation
- Terms aggregation （使用较多）



数字字段只可以进行如下指标聚合：

- Min aggregation
- Max aggregation
- Sum aggregation

- Average aggregation
- Value Count aggregation

每个功能都要结合具体业务场景来使用，切忌为了使用功能而设计。

API 介绍

此处以 Elasticsearch 慢查原始数据统计功能为例进行介绍(敏感信息已经替换)

数据准备

索引 mapping 结构:

```
PUT es-slowlog-2021-04-21
{
    "mappings": {
      "_field_names": {
        "enabled": false
      },
      "dynamic_templates": [
        {
          "strings": {
            "match_mapping_type": "string",
            "mapping": {
              "ignore_above": 512,
              "type": "keyword"
            }
```

```
        }
      }
    ],
    "properties": {
      "@timestamp": {
        "type": "date"
      },
      "cluster": {
        "type": "keyword",
        "ignore_above": 512
      },
      "host": {
        "properties": {
          "name": {
            "type": "keyword",
            "ignore_above": 512
          }
        }
      },
      "elasticsearch": {
        "properties": {
          "index": {
            "properties": {
              "name": {
                "type": "keyword",
                "ignore_above": 512
              }
            }
          }
        }
      },
```

```
            "timestamp_local": {
              "type": "date"
            }
          }
        }
    }
```

单条数据 demo 样例（与上边的 mapping 对应）：

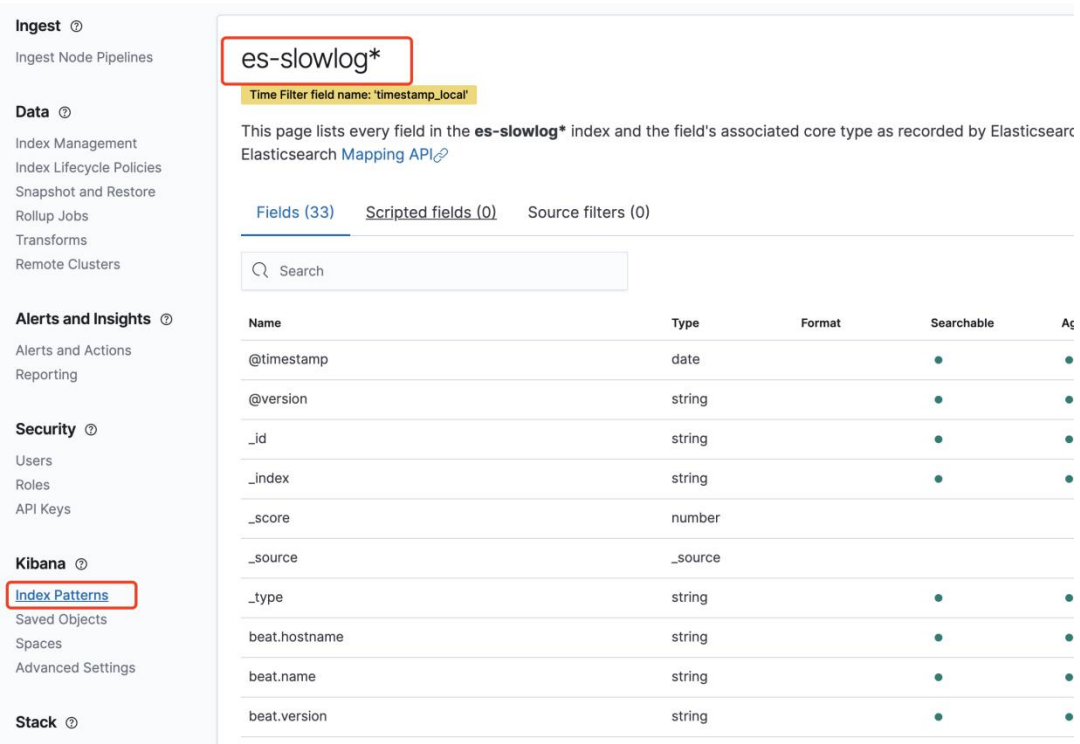
```
POST es-slowlog-2021-04-21/_doc
{
  "cluster": "clustername-demo",
  "offset": 0,
  "log": {
    "level": "WARN"
  },
  "prospector": {
    "type": "log"
  },
  "source": "/home/elasticsearch/clustername-demo_index_search_slowlog.log",
  "message": "[2021-04-21T14:03:06,896][WARN ][i.s.s.query ] [host_name-demo] [basiclog-slowlog_2021-04-02][2] took[2.3s], took_millis[2307], total_hits[23129 hits], types[], stats[], search_type[QUERY_THEN_FETCH], total_shards[4], source[{\"size\":0,\"query\":{\"bool\":{\"filter\":[{\"match_all\":{\"boost\":1.0}},{\"match_phrase\":{\"logtype.keyword\":{\"query\":\"server\",\"slop\":0,\"zero_terms_query\":\"NONE\",\"boost\":1.0}}},{\"range\":{\"@timestamp\":{\"from\":\"2021-04-02T15:48:04.138Z\",\"to\":\"2021-04-02T16:03:04.138Z\",\"include_lower\":true,\"include_upper\":true,\"format\":\"strict_date_optional_time\",\"boost\":1.0}}}],\"adjust_pure_negative\":true,\"boost\":1.0}},\"_source\":{\"includes\":[],\"excludes\":[]},\"stored_fields\":\"*\",\"docvalue_fields\":[{\"field\":\"@timestamp\",\"format\":\"date_time\"},{\"field\":\"time\",\"format\":\"date_time\"}],\"script_fields\":{},\"track_total_hits\":2147483647,\"aggregations\":{\"2\":{\"terms\":{\"field\":\"cluster.
```

keyword\",\"size\":20,\"min_doc_count\":1,\"shard_min_doc_count\":0,\"show_term_doc_count_error\":false,\"order\":[{\"_count\":\"desc\"},{\"_key\":\"asc\"}]}}}}}], id[],",

```
"input": {
  "type": "log"
},
"logtype": "slowlog",
"log_type": "basic-slowlog",
"timestamp_local": "2021-04-21T14:03:06.896+08:00",
"@timestamp": "2021-04-21T14:03:06.896Z",
"elasticsearch": {
  "node": {
    "name": "host_name-demo"
  },
  "slowlog": {
    "took": "2.3s",
    "logger": "i.s.s.query "
  },
  "index": {
    "name": "basiclog-slowlog_2021-04-02"
  },
  "shard": {
    "id": "2"
  }
},
"host": {
  "name": "host_name-demo"
},
"beat": {
  "hostname": "beathostname-demo",
  "name": "beathostname-demo",
  "version": "6.5.4"
```

```
    },
    "@version": "1",
    "event": {
      "duration": 2307000000,
      "created": "2021-04-21T06:59:11.934Z",
      "kind": "event",
      "category": "database",
      "type": "info"
    }
  }
```

在 Kibana 中配置 Index Patterns



注：最新版本 API 请参考官方文档：https://www.elastic.co/guide/en/elasticsearch/reference/master/xpack-rollup.html

基础 API

创建汇总任务:

请求：PUT _rollup/job/<job_id>

参数	必选	类型	说明
index_pattern	是	string	索引 pattern 名称
rollup_index	是	string	目标索引，部分版本限制索引名以 rollup 开头
cron	是	string	定时任务执行周期，与汇总数据的时间间隔无关。
page_size	是	integer	汇总索引每次迭代中处理的存储桶的结果数。值越大，执行越快，但是处理过程中需要更多的内存。
groups	是	object	为汇总作业定义日期直方图聚合
-date_histogram	是	object	定义 日期直方图聚合

参数	必选	类型	说明
--calendar_interval	是	object	时间桶大小，1m 代表一分钟一个桶
--field	是	string	聚合依据的时间字段
--time_zone	否	string	时区，default：UTC
--delay	否	time units	汇总延时，多久之前的数据可以进行汇总，因为部分数据写入可能会有延时，汇总任务前要将数据全部写入并且可查询
-terms	否	object	分组的字段属性
--fields	是	string	定义 terms 字段集。此数组字段可以是 keyword 也可以是 numerics 类型，无顺序要求。
-histogram	否	object	直方图组将一个或多个数字字段聚合为数字直方图间隔
--fields	是	array	构建直方图的字段，必须是数字
--interval	是	integer	汇总时要生成的直方图存储桶的间隔
metrics	否	object	定义汇总数据的方式

参数	必选	类型	说明
-field	是	string	定义需要采集的指标的字段。例如以上示例是分别对，进行采集。
-metrics	是	array	定义聚合算子。设置为 sum,表示对某个指标进行 sum 运算。仅支持 min、max、sum、avg、value_count。
timeout	否	string	请求超时时间

```
PUT _rollup/job/es-slowlog-agg-id
{
  "index_pattern": "es-slowlog*", //索引 pattern 名称
  "rollup_index": "rollup-es-slowlog-agg", //目标索引，rollup-开头必须明确指定
  "cron": "0 * * * * ?", //定时任务执行周期，与汇总数据的时间间隔无关。
  "groups": {
    "date_histogram": { //定义 日期直方图聚合
      "calendar_interval": "1m", // 时间桶大小，一分钟一个桶
      "field": "timestamp_local", //聚合的时间字段
      "delay": "1m", //汇总延时，多久之前的数据可以进行汇总，因为部分数据写入可能会有延时，汇总任务前要将数据全部写入并且可查询
      "time_zone": "UTC" // 时区 eg: GMT+8
    },
    "terms": {
      "fields": [ //汇总字段
        "cluster", // 集群的名称
```

```
            "elasticsearch.index.name", //索引名称
            "host.name" //主机名
        ]
      }
    },
    "metrics": [], //默认是 count 数，可以指定 min、max、sum、average、value count
    "timeout": "20s", // 超时时间
    "page_size": 10000 // 单页数量，较大的值会更快地汇总，但也会耗费更多内存
}
```

查询所有汇总任务:

```
GET _rollup/job/*
```

获取单个汇总任务详情:

请求：GET _rollup/job/<job_id>

```
GET _rollup/job/es-slowlog-agg-id
{
    "jobs": [
      {
        "config": {
            "id": "es-slowlog-agg-id",
            "index_pattern": "es-slowlog*",
            "rollup_index": "rollup-es-slowlog-agg",
            "cron": "0 * * * * ?",
```

```
      "groups": {
        "date_histogram": {
          "calendar_interval": "1m",
          "field": "timestamp_local",
          "delay": "1m",
          "time_zone": "UTC"
        },
        "terms": {
          "fields": [
            "cluster",
            "elasticsearch.index.name",
            "host.name"
          ]
        }
      },
      "metrics": [

      ],
      "timeout": "20s",
      "page_size": 10000
    },
    "status": {
      "job_state": "stopped",
      "upgraded_doc_id": true
    },
    "stats": {
      "pages_processed": 0,
      "documents_processed": 0,
      "rollups_indexed": 0,
```

```
            "trigger_count": 0,
            "index_time_in_ms": 0,
            "index_total": 0,
            "index_failures": 0,
            "search_time_in_ms": 0,
            "search_total": 0,
            "search_failures": 0,
            "processing_time_in_ms": 0,
            "processing_total": 0
          }
        }
    ]
}
```

开始汇总任务:

请求：POST _rollup/job/<job_id>/_start

```
POST _rollup/job/es-slowlog-agg-id/_start
//执行后获取当前任务状态，关注下 status、stat,status 中
GET _rollup/job/es-slowlog-agg-id
{
  "jobs": [
    {
      "config": {
        "id": "es-slowlog-agg-id",
        "index_pattern": "es-slowlog*",
        "rollup_index": "rollup-es-slowlog-agg",
```

```
        "cron": "0 * * * * ?",
        "groups": {
          "date_histogram": {
            "calendar_interval": "1m",
            "field": "timestamp_local",
            "delay": "1m",
            "time_zone": "UTC"
          },
          "terms": {
            "fields": [
              "cluster",
              "elasticsearch.index.name",
              "host.name"
            ]
          }
        },
        "metrics": [

        ],
        "timeout": "20s",
        "page_size": 10000
      },
      "status": {
        "job_state": "started",  //如果停止的任务，此处显示 stopped
        "current_position": { //当前 rollup 任务执行的位置，及 term 结果
          "cluster.terms": "clustername-demo",
          "elasticsearch.index.name.terms": "basiclog-slowlog_2021-04-02",
          "host.name.terms": "host_name-demo",
          "timestamp_local.date_histogram": 1618984980000
```

```
            },
            "upgraded_doc_id": true
          },
          "stats": {//执行状态
            "pages_processed": 2,
            "documents_processed": 1,
            "rollups_indexed": 1,
            "trigger_count": 1,
            "index_time_in_ms": 103,
            "index_total": 1,
            "index_failures": 0,
            "search_time_in_ms": 6,
            "search_total": 2,
            "search_failures": 0,
            "processing_time_in_ms": 0,
            "processing_total": 2
          }
        }
      ]
    }
```

status.job_state 描述:

- stopped

  表示任务已暂停。

- started

  表示任务正在运行，但没有主动汇总数据。当 cron 间隔触发时，作业的任务将开始处理数据。

- indexing

意味着正在处理数据并创建新的汇总文档。在此状态下，任何后续的 cron 间隔触发器都将被忽略，因为该作业已经与先前的触发器一起处于活动状态。

- abort

是一种瞬态，通常用户不会看到。如果由于某种原因需要关闭任务（已删除作业，遇到不可恢复的错误等）。abort 状态后不久，作业将自己从群集中删除。

停止汇总任务:

请求：POST _rollup/job/<job_id>/_stop

```
POST _rollup/job/es-slowlog-agg-id/_stop
```

删除汇总任务:

请求：DELETE _rollup/job/<job_id>

```
删除操作需谨慎
DELETE /_rollup/job/es-slowlog-agg-id
```

_rollup_search 查询

因为在原始文档和汇总文档中使用的文档结构不同。 Rollup 搜索会将标准查询 DSL 重写为与汇总文档相同的结构，然后获取响应并将其重写回客户端。

使用方式：

GET **<target>**/_rollup_search

<target>参数规则（必需，字符串）：

- 必须指定索引或通配符表达式。
- 可以指定多个非汇总索引。
- 只能指定一个汇总索引。如果提供多个，则会发生异常。
- 可以使用通配符表达式，但是，如果它们匹配多个汇总索引，则会发生异常。

eg: es-slowlog*,rollup-es-slowlog-agg1/_rollup_search。

请求体支持常规 Search API 的功能的子集。它支持：

- query 用于指定 DSL 查询的参数，但受一些限制

请参阅：

汇总搜索限制：https://www.elastic.co/guide/en/elasticsearch/reference/7.x/rollup-search-limitations.html

汇总聚合限制：https://www.elastic.co/guide/en/elasticsearch/reference/7.x/rollup-agg-limitations.html

- aggregations 用于指定聚合的参数

不可用的功能：

- size：无法获取原始数据，如果想获取原始数据，请使用 _search 查询汇总索引。
- highlighter，suggestors，post_filter，profile，explain：不允许使用。

原始数据和汇总索引同时查询实现原理：

Elasticsearch 接收到原始数据和汇总数据联合 _rollup_search 查询响应后， 会重写汇总响应，并将两者合并在一起。在合并过程中，如果两个响应之间的存储桶中有任何重叠，则使用非汇总索引中汇总的桶数据。

样例：

创建新的复杂任务，具体任务信息如下：

```
//创建复杂任务，汇总多个指标，任务详情如下
{
  "config": {
    "id": "es-slowlog-agg-id1",
    "index_pattern": "es-slowlog*",
```

```
"rollup_index": "rollup-es-slowlog-agg1",
"cron": "0 * * * * ?",
"groups": {
  "date_histogram": {
    "calendar_interval": "1m",
    "field": "timestamp_local",
    "delay": "1m",
    "time_zone": "UTC"
  },
  "histogram": {
    "interval": 8,
    "fields": [
      "event.duration"
    ]
  },
  "terms": {
    "fields": [
      "cluster",
      "elasticsearch.index.name",
      "host.name"
    ]
  }
},
"metrics": [
  {
    "field": "event.duration",
    "metrics": [
      "avg",
      "max",
      "min",
      "sum",
```

```
            "value_count"
          ]
        }
      ],
      "timeout": "20s",
      "page_size": 10000
    },
    "status": {
      "job_state": "started",
      "current_position": {
        "cluster.terms": "clustername-demo",
        "elasticsearch.index.name.terms": "basiclog-slowlog_2021-04-02",
        "event.duration.histogram": 2307000000,
        "host.name.terms": "host_name-demo",
        "timestamp_local.date_histogram": 1618984980000
      },
      "upgraded_doc_id": true
    },
    "stats": {
      "pages_processed": 6,
      "documents_processed": 1,
      "rollups_indexed": 1,
      "trigger_count": 5,
      "index_time_in_ms": 115,
      "index_total": 1,
      "index_failures": 0,
      "search_time_in_ms": 21,
      "search_total": 6,
      "search_failures": 0,
      "processing_time_in_ms": 0,
      "processing_total": 6
    }
  }
```

_search 查询汇总目标索引中的原始数据:

```
GET rollup-es-slowlog-agg1/_search
{
  "size":10,
  "query": {
    "bool": {
      "must": [],
      "filter": [
        {
          "match_all": {}
        }
      ],
      "should": [],
      "must_not": []
    }
  }
}

返回结果
{
  "took": 2,
  "timed_out": false,
  "_shards": {
    "total": 1,
    "successful": 1,
    "skipped": 0,
    "failed": 0
  },
  "hits": {
```

```
        "total": {
          "value": 1,
          "relation": "eq"
        },
        "max_score": 1,
        "hits": [
          {
            "_index": "rollup-es-slowlog-agg1",
            "_type": "_doc",
            "_id": "es-slowlog-agg-id1$5uzfGmyS2uAb3XRznkZBgA",
            "_score": 1,
            "_source": {
              "cluster.terms.value": "bj-ali-xueyan-oa-es-cluster",
              "event.duration.avg._count": 1,
              "event.duration.max.value": 2377000000,
              "event.duration.histogram.value": 2377000000,
              "timestamp_local.date_histogram.time_zone": "UTC",
              "elasticsearch.index.name.terms.value": "basiclog-slowlog_2400-2021-04-02",
              "host.name.terms._count": 1,
              "cluster.terms._count": 1,
              "host.name.terms.value": "bj-sjhl-university-es-online-99-62",
              "event.duration.avg.value": 2377000000,
              "elasticsearch.index.name.terms._count": 1,
              "event.duration.histogram.interval": 8,
              "timestamp_local.date_histogram._count": 1,
              "timestamp_local.date_histogram.timestamp": 1618995780000,
              "_rollup.version": 2,
              "event.duration.histogram._count": 1,
              "timestamp_local.date_histogram.interval": "1m",
              "event.duration.sum.value": 2377000000,
              "event.duration.min.value": 2377000000,
```

```
            "event.duration.value_count.value": 1,
            "_rollup.id": "es-slowlog-agg-id1"
          }
        }
      ]
    }
}
```

_rollup_search 查询数据（可以把原始数据和汇总数据联合查询）：

```
GET es-slowlog*,rollup-es-slowlog-agg1/_rollup_search
{
  "size": 0,
  "aggregations": {
    "avg_event.duration": {
      "avg": {
        "field": "event.duration"
      }
    }
  }
}


//返回值
{
  "took": 740,
  "timed_out": false,
  "terminated_early": false,
```

```
    "num_reduce_phases": 2,
    "_shards": {
      "total": 5,
      "successful": 5,
      "skipped": 0,
      "failed": 0
    },
    "hits": {
      "total": {
        "value": 0,
        "relation": "eq"
      },
      "max_score": 0,
      "hits": [

      ]
    },
    "aggregations": {
      "avg_event.duration": {
        "value": 2311777445.714286
      }
    }
  }
```

获取汇总信息

根据 Rollup 配置中的 index_pattern 获取对应的任务,支持 _all 查询所有

请求：GET _rollup/data/

```
//查询所有
GET _rollup/data/_all
//查询指定目标
GET _rollup/data/es-slowlog*
{
  "es-slowlog*": {
    "rollup_jobs": [
      {
        "job_id": "es-slowlog-agg-id",
        "rollup_index": "rollup-es-slowlog-agg",
        "index_pattern": "es-slowlog*",
        "fields": {
          "cluster": [
            {
              "agg": "terms"
            }
          ],
          "timestamp_local": [
            {
              "agg": "date_histogram",
              "delay": "1m",
              "time_zone": "UTC",
              "calendar_interval": "1m"
            }
          ],
          "elasticsearch.index.name": [
```

```
            {
              "agg": "terms"
            }
          ],
          "host.name": [
            {
              "agg": "terms"
            }
          ]
        }
      },
      {
        "job_id": "es-slowlog-agg-id1",
        "rollup_index": "rollup-es-slowlog-agg",
        "index_pattern": "es-slowlog*",
        "fields": {
          "cluster": [
            {
              "agg": "terms"
            }
          ],
          "timestamp_local": [
            {
              "agg": "date_histogram",
              "delay": "1m",
              "time_zone": "UTC",
              "calendar_interval": "1m"
            }
          ],
          "elasticsearch.index.name": [
            {
```

```
            "agg": "terms"
          }
        ],
        "host.name": [
          {
            "agg": "terms"
          }
        ]
      }
    },
    {
      "job_id": "es-slowlog-agg-id1",
      "rollup_index": "rollup-es-slowlog-agg1",
      "index_pattern": "es-slowlog*",
      "fields": {
        "event.duration": [
          {
            "agg": "histogram",
            "interval": 8
          },
          {
            "agg": "avg"
          },
          {
            "agg": "max"
          },
          {
            "agg": "min"
          },
          {
            "agg": "sum"
```

```
        },
        {
          "agg": "value_count"
        }
      ],
      "cluster": [
        {
          "agg": "terms"
        }
      ],
      "timestamp_local": [
        {
          "agg": "date_histogram",
          "delay": "1m",
          "time_zone": "UTC",
          "calendar_interval": "1m"
        }
      ],
      "elasticsearch.index.name": [
        {
          "agg": "terms"
        }
      ],
      "host.name": [
        {
          "agg": "terms"
        }
      ]
    }
  },
  {
```

```
            "job_id": "es-slowlog-agg-id3",
            "rollup_index": "rollupes-slowlog-agg",
            "index_pattern": "es-slowlog*",
            "fields": {
              "cluster": [
                {
                  "agg": "terms"
                }
              ],
              "timestamp_local": [
                {
                  "agg": "date_histogram",
                  "delay": "1m",
                  "time_zone": "UTC",
                  "calendar_interval": "1m"
                }
              ],
              "elasticsearch.index.name": [
                {
                  "agg": "terms"
                }
              ],
              "host.name": [
                {
                  "agg": "terms"
                }
              ]
            }
          }
        ]
      }
    }
```

根据 Rollup 目标索引查询对应的任务，支持 * 匹配

请求：GET /_rollup/data

```
GET rollupes-slowlog-*/_rollup/data
GET rollupes-slowlog-agg/_rollup/data
{
  "rollupes-slowlog-agg": {
    "rollup_jobs": [
      {
        "job_id": "es-slowlog-agg-id3",
        "rollup_index": "rollupes-slowlog-agg",
        "index_pattern": "es-slowlog*",
        "fields": {
          "cluster": [
            {
              "agg": "terms"
            }
          ],
          "timestamp_local": [
            {
              "agg": "date_histogram",
              "delay": "1m",
              "time_zone": "UTC",
              "calendar_interval": "1m"
            }
          ],
          "elasticsearch.index.name": [
            {
              "agg": "terms"
```

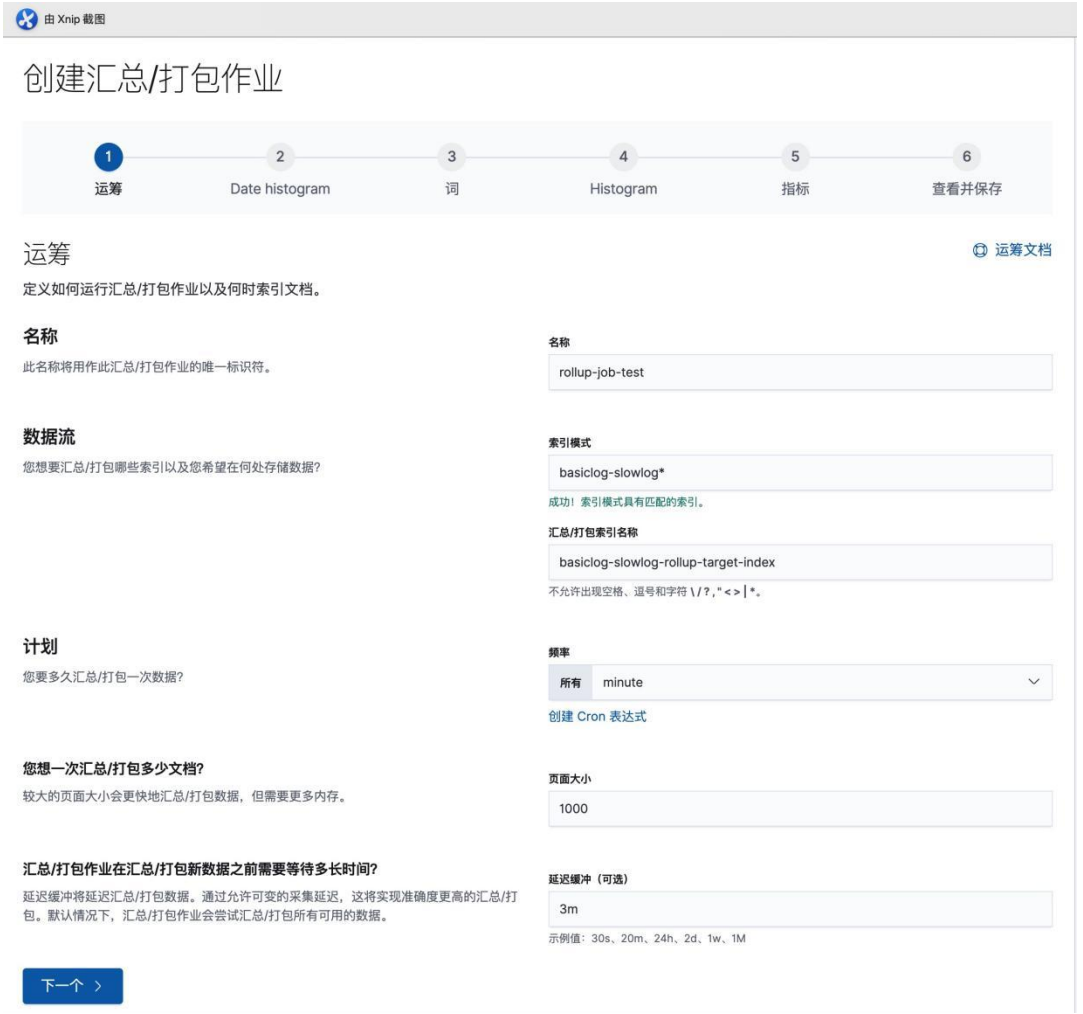
```
            }
          ],
          "host.name": [
            {
              "agg": "terms"
            }
          ]
        }
      }
    ]
  }
}
```

Kibana 使用介绍

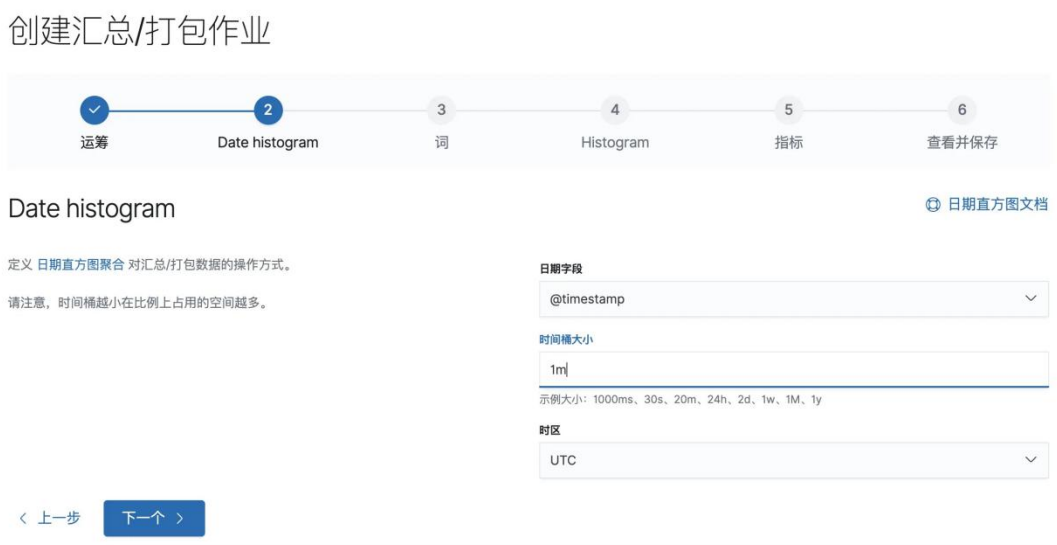
对 API 有了一定了解之后，再来通过 Kibana 创建对应 Elasticsearch 集群的慢查统计就比较简单了

Kibana 使用中文的部分功能有 bug（例如 Rollup 选择指标时，会出现异常的情况），建议 Kibana 语言选择英文。

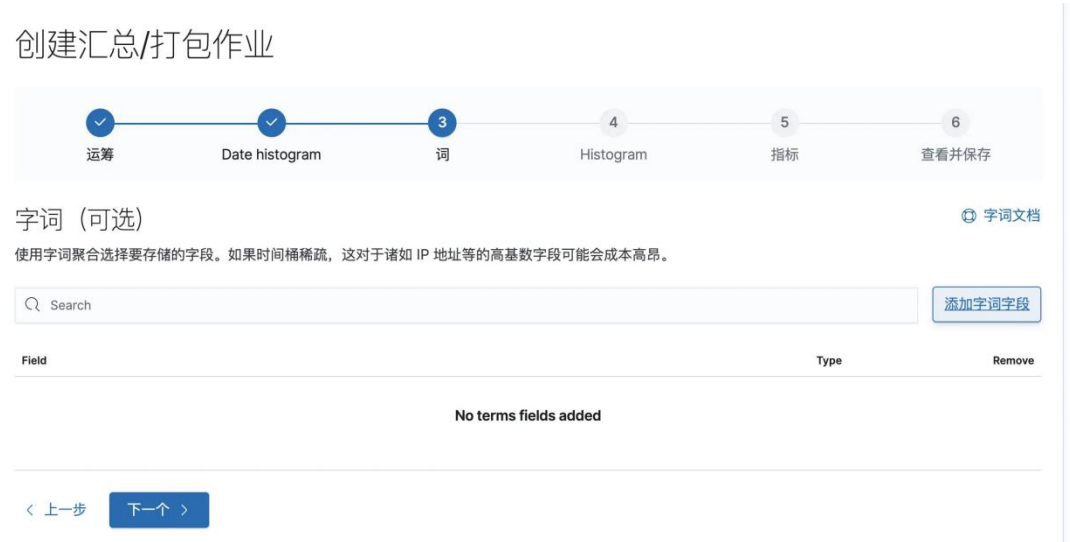
填写 Logistics

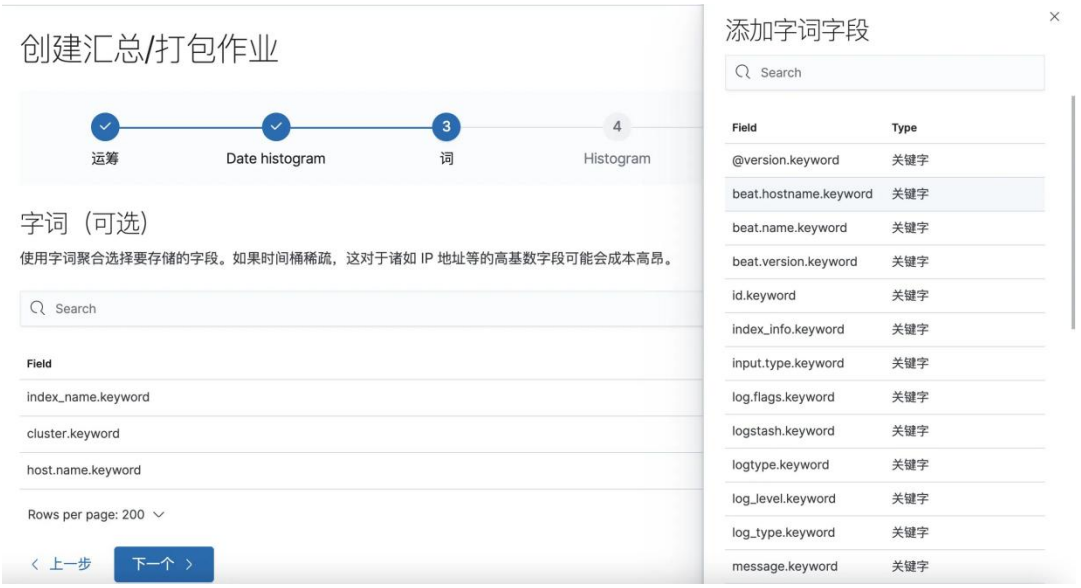


选择 Date histogram（必填）

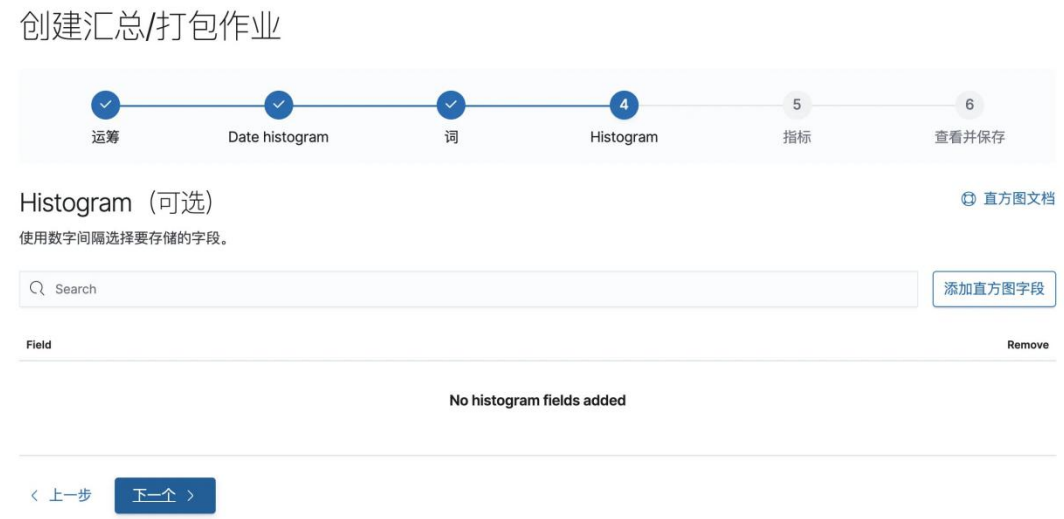


选择 Terms ，此处选择集群名称、索引名称、节点名称（选填）





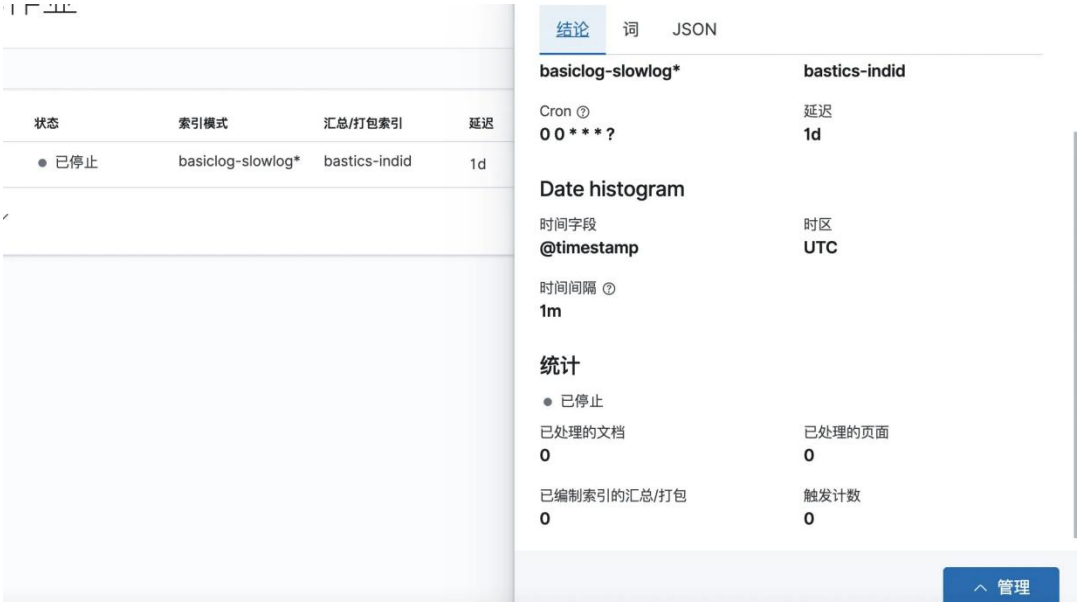
根据需求选择 Histogram（选填），本次样例中的 Elasticsearch 慢查 Rollup 只需要统计 Count 数，此处不需要选择，直接下一步。



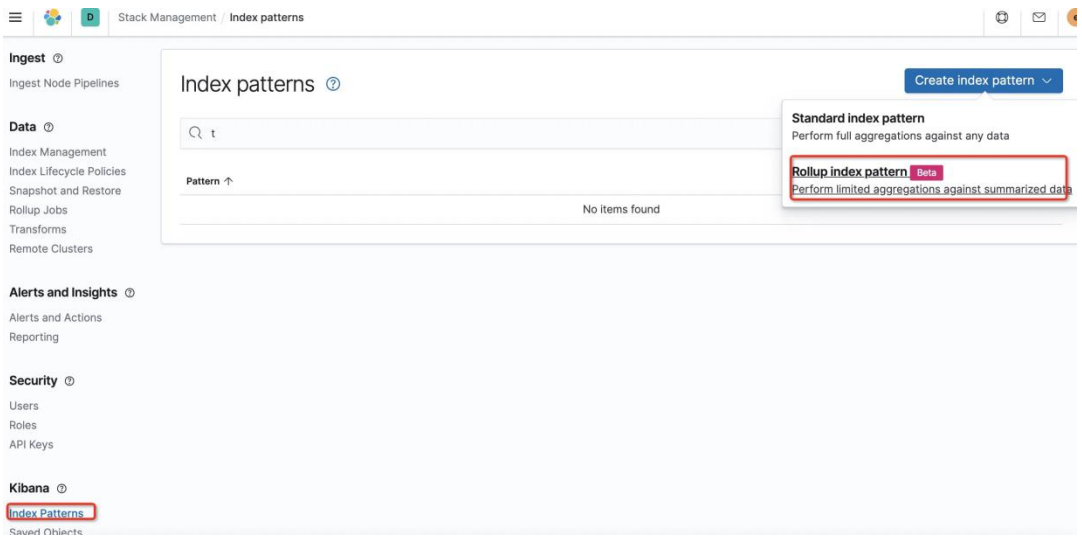
根据需求填写 Metrics（选填），本次样例中的 Elasticsearch 慢查 Rollup 只需要统计 Count 数，此处不需要选择，直接下一步。

操作完成，保存。

查看状态：



配置 Index Pattern 注意选择的是 Rollup index pattern，图表配置和普通没有区别：



创作人简介：

杨景江，关注研究中间件，比如 ES，Redis，RocketMQ 等技术领域。

博客：https://blog.csdn.net/xiaoyanghapi/article/month/2016/08

3.5.4 Graph

创作人：杨丛聿

审稿人：朱荣鑫

图作为一种现实中广泛存在的结构，与我们的生活息息相关，如社交网络、交通网络等。如何抽象的描述这些结构，并对其进行分析，获取潜在价值是一个普遍存在的问题。

相比于我们熟悉的关系数据库的建模方式，从 Graph 的角度出发，可以让一些复杂问题和场景的处理变得简单，如行为分析、个性化推荐、知识图谱等。

通常来讲，对于图的研究，分为图存储和图计算分析两个场景，二者解决的问题不同，相应的技术栈也有一些差异。

图存储

更强调图结构的建模，以及解决诸如邻居、路径等查询问题。

图计算

则是从全局或子图的角度为出发点，经过大量计算，来发现更深层次的问题，其关注的技术点，更多的是资源调度，以及各类经典图分析算法的实现。

Elasticsearch 的 Graph 功能介于二者之间，其定位是对已有的结构化数据进行分析，从图的角度去审视已有数据，并发现潜在价值。

图通常是以 vertex 和 edge 的模型管理数据，Elasticsearch 也不例外，只是区别于图数据库的先建模后存储，Elasticsearch 中 Graph 的建模逻辑，是在查询中动态填入的，其底层实现逻辑是基于 Elasticsearch 的 terms 和 aggregations 相关功能。

相比于图数据库通常只提供基于 vertex 和 edge 的基础查询，Elasticsearch 由于本身在构建倒排索引时会统计词频，这使其在查询层面，兼具一定的分析能力。可见 Elasticsearch Graph 本身的定位并不是图结构数据的管理（如知识图谱），而是在已有的结构化数据中，以图的视角来发现一些问题。如库中已存在用户的听歌记录，则可以根据听众偏好，为其进行歌曲的推荐。

其他图数据库概览

由于 Elasticsearch 本身从逻辑上更接近图数据库，不管是 vertices + connections 的数据模型还是查询模式，本节先介绍下目前主流图数据库的类别和实现逻辑，具体细节不进行展开。

目前主流的图数据库，以属性图和 RDF 模型为主；

RDF 相对小众，仅有 DGraph 属于此范畴；

属性图是由节点、边、属性三者构成的有向图，相比于其他数据库，其关键特征之一是边（或连接）与顶点一同被视为模型的核心组件；

基于这些理念，一般属性图数据的结构可以归纳如下：

每个顶点（vertex）包括：

- 唯一标识符
- 一组出边
- 一组入边

一组属性（键值对

每条边（edge）包括：

- 唯一标识符
- 边的起点
- 边的终点
- 描述两个顶点间关系类型的标签

一组属性（键值对）

对于各类图数据库而言，数据建模的核心便是如何组织这些数据，从存储和索引的角度来看，目前主要分为自行实现和 Graph 层两种：

自行实现存储&索引

代表如 neo4j，其使用了免索引邻接，即每个节点都会维护其相邻节点的引用，免去了基于索引查找相邻节点的开销，这个设计理念在很多图数据库中都可以看到；

以 Graph 层的形式构建，使用外置 Nosql 存储和外置索引

代表 JanusGraph、HugeGraph，对于此类图数据库，Elasticsearch 便可作为其索引层存在。

除此以外，如何在分布式场景下，对存储和计算进行优化，是各类图数据库遇到的最大问题，针对诸如分布式场景下的分片、删除等问题，逐步诞生了 NebulaGraph、Dgraph 等。

Elasticsearch 的实现方式

Elasticsearch 的 Graph 功能始于 5.5 版本，属于 X-pack 的扩展功能组，从 API 的路径 /_graph/explore 可以看出，其定位更倾向于是探索分析，即在已有的索引上通过聚合的方式进行分析。

常规的 graph 查询可以理解为，先进行两层嵌套的 terms 聚合，再将查询结果以 vertexs 和 connections 的数据模型进行返回。

在使用 Graph 功能时，有 3 个核心要素，分别是 vertices、connections、controls，前两者主要用于确定图查询中，前者通过哪些字段产生，后者用于控制一些查询细节。

如希望通过点击日志探查用户的搜索词和点击的产品间的关联，可通过如下查询：

```
POST clicklogs/_graph/explore
{
  "query": {
    "match": {
      "query.raw": "midi"
    }
  },
  "vertices": [
    {
      "field": "product"
    }
  ],
  "connections": {
    "vertices": [
      {
        "field": "query.raw"
      }
    ]
  }
}
```

查询语句中

- vertices 用于指定对哪些字段的内容感兴趣，后续会作为 target 节点处理，对应的字段必须是已索引的字段。
- connections 用于指定希望哪些字段和 vetices 的内容进行关联，后续会作为 source 节点处理，connections 也支持使用 query 缩小关联内容的范围。

这个查询会以聚合的方式得到最终结果，会产生类似两层嵌套 aggs 的查询效果：

```
"aggs": {
    "vertices": {
      "terms": {
        "field": "product"
      },
      "aggs": {
        "connections": {
          "terms": {
            "field": "query.raw"
          }
        }
      }
    }
  }
```

查询结果会返回 vertices 和 connections 两个数组，通过这两部分数据，即可构成一个有向图（方向是由 connections 中设定的 query.raw 指向 vertices 中设定的 product）。

为了能更直观的反映每个节点，和每条边的重要程度，vertices 和 connections 的元素中均会额外返回一个 weight，这点也是 Elasticsearch Graph 区别于其他图数据库查询的功能点之一。

```
"vertices": [
        {
            "field": "query.raw",
            "term": "midi cable",
            "weight": 0.08745858139552132,
            "depth": 1
        },
        {
            "field": "product",
            "term": "8567446",
            "weight": 0.13247784285434397,
            "depth": 0
        }
],"connections": [
        {
            "source": 0,
            "target": 1,
            "weight": 0.04802242866755111,
            "doc_count": 13
        }
]
```

除了上述例子所举的常规查询模式外，Elasticsearch 的 Graph API 还可以通过 controls 控制采样规模和 weight 的计算：

- use_significance（默认 false）

如设置为 true，第二层的聚合将会变为 Significant Terms，得到的 weight 会是通过前置和后置频率算出的，可用来发现一些有趣，或不寻常的节点或关系。

- sample_size（默认 100）

通过减少采样规模，可以有效提高检索效率。

除此以外，Elasticsearch 还支持通过 Kibana 配置 Graph 相关的可视化页面，借此可以快速将上述查询得到的结果以图形化的方式进行展现。以下示例基于 7.10 操作：

- 在 Kibana 中选择 Graph 进入导航页面；
- 选取一个数据源，即库中已存在的索引；

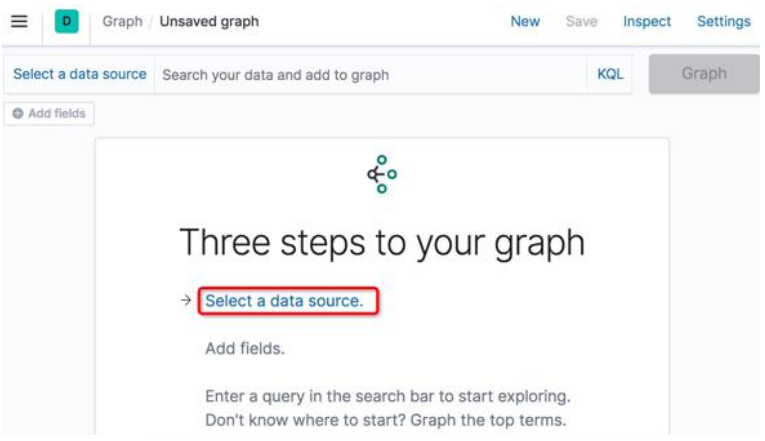


图 1 Kibana 操作-选取数据源

添加字段并触发查询即可得到如下的图结构分析页面 （可变更颜色区分节点），上文提及的 weight 值将通过节点间连线的粗细体现。

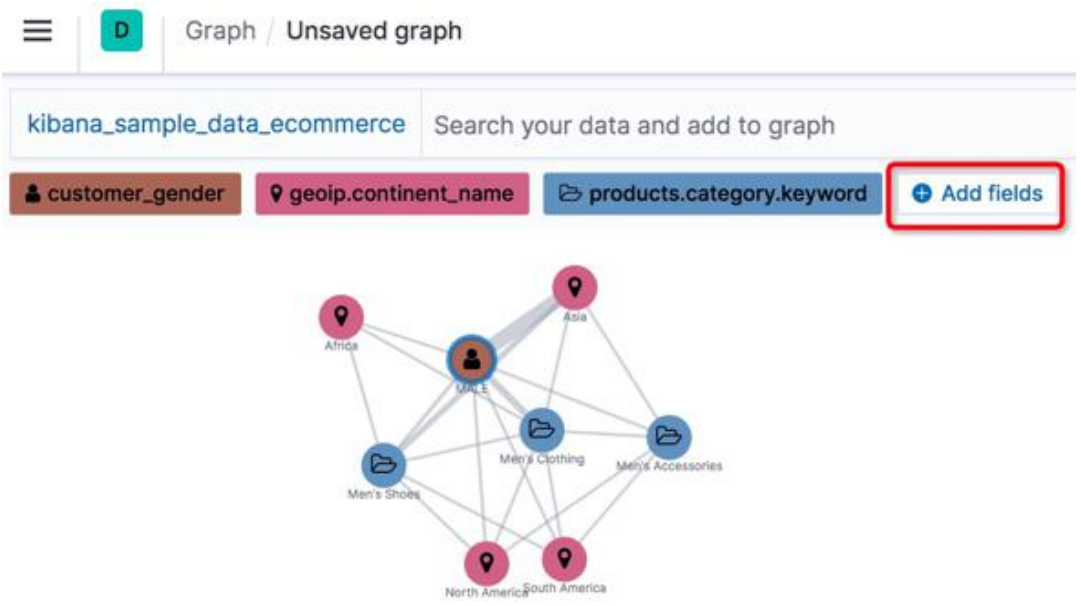


图 2 Kibana 操作-以图的视角分析索引不同字段

总结

高可用和近实时的能力

Elasticsearch 的 Graph 功能通过 X-pack 提供，其优点是可以对库中已有的数据，快速进行分析，且具备可视化能力。其天然具备高可用和近实时的能力。

千万以及亿级的数据进行 Graph 分析

依托于已有的倒排索引和聚合功能，可以快速以 vertices 和 connections 的数据模型呈现结果。由于聚合本身需要消耗较大的内存和计算量，在使用时需要留意数据量级和系统资源，在资源允许的情况下，可以支持对千万以及亿级的数据进行 Graph 分析。

当然 Elasticsearch 的 Graph 也存在一定局限性，它本身定位是对已有索引进行分析，因此不适合需要建模管理图结构数据的场景，且无法对多跳的路径查询进行有效支持。

并且 Elasticsearch Graph 内部的处理逻辑，是通过对索引进行字段级的聚合，因此产生关系的上下游节点仅限于同一索引中的不同字段。

综上，如果你希望发现索引中不同字段的一些潜在价值，不妨使用 Elasticsearch Graph 功能来进行探索。

参考资料：

- https://www.elastic.co/guide/en/elasticsearch/reference/master/graph-explore-api.html
- https://www.elastic.co/guide/en/kibana/current/graph-getting-started.html
- https://nebula-graph.io/posts/review-on-graph-databases/

3.5.5 Shard allocation

创作人：毛夏军

审稿人：刘帅

什么是分片分配 (shard allocation)

分片分配 (shard allocation)，是指在索引创建、副本增减、节点增减、分片重平衡等，将索引分片落实到实际的物理节点的过程，分片分配可以分为，集群级分配和索引级分配两种，集群级分配常见的包括：

- 要求热索引的分片不要去往低配机器。
- 商品、订单索引的分片不要分配到同一个节点等。

将分片分布到不同的节点，一方面是为了提高系统的可用性，如当集群中一台机器宕机，使得该节点上的分片不可用时，分布在其他机器上的分片，能通过重新选举继续工作 (但是仍要保证同一分片的主从副本不全在宕机节点上) 。

另一方面是为了提高系统的容量和读写性能，如通过增加节点横向扩容，将集群中部分分片 Rebalance 到新节点，既可以利用新节点的存储容量，提升索引存储容量，迁移过来的分片，可以利用新节点增加的算力提供服务。

总结一下分片分配过程，有两个基本要素：

- 分片：来自集群内的全部索引，包括主分片和副本分片。
- 节点：组成集群的各个 Elasticsearch 进程，能够通过一定的标识来识别个体或划分成组。

那么，集群内的各节点是如何被识别和标记的呢？

节点属性 (node attributes)

节点属性 (node attributes)，包括内置属性和自定义属性两种。

Elasticsearch 会将常见用于区分不同机器的标记，如主机名 (_host)、IP 地址 (_ip)、节点名称 (_name) 等作为内置属性，供分片分配时区分节点的标记使用，具体包括：

- _name：节点名称，即在 elasticsearch.yml 中定义的 node.name 属性
- _host_ip、_publish_ip、_ip：节点的 IP 地址，一般情况下使用 _ip 即可，具体含义可以查阅官方帮助文档
- _host：主机名
- _id：集群为节点自动分配的唯一标识符，手动调控时使用较少
- _tier：节点的数据角色，比如存储冷热数据的 data_cold、data_hot 等，可以在 elasticsearch.yml 中指定 node.roles 属性

内置属性可以用来区分不同节点，但是对于将节点划分成组来说不是很便利，比如我们希望将索引分配到高配节点。如果使用内置属性，比如 _ip 的话，需要在索引设置

index.routing.allocation.include._ip 中，指定多个机器的 IP 地址，在集群增删高配节点的情况下，需要同时调整对应索引的分片分配设置，显得不太便捷。

我们很自然的希望除了内置属性之外，还可以根据机器配置的高低、是否同属于一个网段等情况来标记节点，方便我们将不同类型的节点划分为一个个组，继而将索引的分配配置到节点组，这样在集群增删节点时，只要节点配置了对应的组，集群就会根据对应组的节点变化自动的将分片重新调整，而不需要我们再手动的同步每一个索引的分配设置。

自定义节点属性解决的便是这个问题，我们可以通过：

- 在 elasticsearch.yml 中新增配置项，如 node.attr.zone=zone1
- 或在启动命令中增加变量，如 bin/elasticsearch -Enode.attr.zone=zone1

以上方式来为节点增加自定义属性。

小结

分布式架构为我们带来了众多容量、性能和可用性方面的优势，但是相应的也提高了保障难度，因为分片在集群自动分配的情况下不一定能达到我们期望的"平衡"状态，需要我们对分片分配机制有较高的掌握程度来调控集群内分片的分配状况，比如主从副本不分布在同一物理节点、解决节点数据倾斜导致新分片集中于某几台负载较低节点的热点问题等。

调控分片分配

Elasticsearch 集群中 master 节点的一项重要功能，就是决定分片如何以最佳的方式，均衡分布到集群内的各个节点上。除了自动分配之外，我们也可以从粗粒度的集群维度和细粒度的索引维度，手动调控分片在各节点的分配。

集群维度（cluster level）的分片分配，是将所有分片纳入一起考虑，不会单独考虑某个索引的分片分配情况。

举个例子：我们有两个索引，每个索引包含两个分片，将分片分配到两个节点组成的集群。

状态一：

同一个索引的全部分片分配到同一个节点：

```
{
  "node_1": ["index_1_shard_1", "index_1_shard_2"],
  "node_2": ["index_2_shard_1", "index_2_shard_2"]
}
```

状态二 ：

同索引的分片均匀分配到各节点：

```
{
  "node_1": ["index_1_shard_1", "index_2_shard_2"],
  "node_2": ["index_2_shard_1", "index_1_shard_2"]
}
```

从集群维度，都可以被认为是"平衡"状态，但是从实际角度看，只有状态二是我们期望的平衡状态。因为如果 index_1 和 index_2 负载不均，在状态一下，很可能导致集群内节点负载不均，使得服务整体表现不能达到预期。

要达到我们期望的状态二，就可以使用索引维度（index level）的分片分配控制方法，比如通过 index.routing.allocation.total_shards_per_node 参数控制每个节点的分片数量为 1 ，就可以达到我们的目的了。

集群维度分片分配

从集群层面来说，分片分配控制的是集群内各索引的分片，集合在各节点的分布，并且在分片分配过程中，添加一些硬性限制以控制集群负载在合理范围内。

以一个实际操作中可能会遇到的情况来说，公司准备新上一批业务，需要用到大规模的数据检索功能，首先需要处理搭建集群的任务，在开始搭建集群之前，让我们先看看实际的业务场景。

系统用于收集应用打点日志，提供一周内数据供查询。

分片平衡的启发式参数

日志类型数据存储，带有非常明显的时效性，一般情况下当日数据的读写频繁，非当日数据几乎不会有写操作，离当前时间越久的数据读操作也越少。

通过上述判断，可以认为索引的分片，越是平均分布于集群内各节点越好，因为可以充分利用全部节点的算力，来分摊当日数据的高频读写负载。

为了达到这个目的，我们可以通过 Elasticseach 提供的部分启发式参数，让 master 在决策分片如何分配时，更多的向我们期望的方向考虑：

- cluster.routing.allocation.balance.shard 节点中分片总数对权重的影响因子，默认为 0.45，该值越大则各节点的分片数越趋向于相等。
- cluster.routing.allocation.balance.index 节点中来自不同索引的分片数对权重的影响因子，默认为 0.55，该值越大，则各索引的分片更倾向于均匀分配到各节点。
- cluster.routing.allocation.balance.threshold 默认为 1.0，该值越大，则集群对不平衡状态的容忍程度越高。

我们可以适当放大 cluster.routing.allocation.balance.index 的权重来使得集群在分配分片时，更倾向于将一个索引的不同分片，均匀分布到各个节点。不过需要注意的是，这只是一个启发式参数，更多的是"建议"，而不是"命令"，要完全达到我们的期望，还需要借助索引维度的分配调控手段。

分片迁移的流量控制

回到本节起始提到的日志业务，系统上线运行一段时间后，随着索引量的不断增加，

我们需要适时的清理掉过期数据，清理过程中自然的会删除过期数据所在的索引，释放存储空间供新的索引使用。

在集群删除索引时，因为集群内分片总数发生了变化，自然的分片在各节点的分配状态也随之发生变化，可能会出现分片的"不平衡"状态。这时默认情况下集群，会自动触发分片的重平衡操作，将分片在各节点间适当的迁移，以使得分片在集群重新达到"平衡"状态。

在日志类数据情况下，单个分片包含的数据量可能会较大，达到若干 GB。这样在分片发生迁移时，节点必然会触发大量的 IO 操作，为了避免大量的 IO 操作对节点造成冲击，使得集群服务发生抖动，我们可以通过分片迁移的流量控制参数进行干预：

- cluster.routing.allocation.node_concurrent_incoming_recoveries

用于控制可同时在一个节点上进行初始化或恢复的最大分片数，默认为 2，设置过大可能导致节点负载过高 (同时写入大量数据)，调整时需要考虑节点的硬件配置。

- cluster.routing.allocation.node_concurrent_outgoing_recoveries

用于控制该节点可同时为其他节点分片恢复或迁移提供数据源的最大分片数，默认为 2，调整同理。

注 1: 从节点的角度，分片会出现两种流向：流入 和 流出，其中，流入是指来自某个索引的分片新落入到该节点，流出是指该在其他节点的分片以该节点所属的分片为数据源进行副本恢复或者数据迁移。

节点的水位线

随着打点应用的接入越来越多，单日的日志索引量上涨迅速，节点的磁盘水位吃紧，我们希望新的分片在分配时，能考虑到节点存储容量的状态，避免将新分片分配到磁盘容量快满的节点。

这个情况下如果要自行通过节点属性来调控，至少需要：

- 自动监测磁盘水位，并为节点打上 low/medium/high 的属性，而且更改属性还需要重启节点。
- 为新分配的索引设置 index.routing.allocation.require.* 属性，来让索引避开高水位节点。
- 在节点的磁盘水位属性变更时，自动为集群内的索引更新 allocation 配置来避免自动平衡。

看上去就很麻烦，那么有没有简便方法呢？

答案是使用内置的水位 cluster.routing.allocation.disk.watermark.* 属性。

水位限制分为高低两种，其中：

- cluster.routing.allocation.disk.watermark.low

低水位，默认为磁盘容量的 85%，Elasticsearch 会避免将分片分布至磁盘容量超过低水位的节点。但是新创建索引的主分片 (primary shards)，仍然可以分配到超过低水位的节点。

- cluster.routing.allocation.disk.watermark.high

高水位，默认为磁盘容量的 90%，Elasticsearch 会将磁盘容量超过高水位节点上的分片迁移至其他节点。

- cluster.routing.allocation.disk.watermark.flood_stage

警戒水位，默认为磁盘容量的 95%，当节点磁盘容量超过警戒水位时，该节点所属分片所在的索引，都会被执行写禁止操作，即索引变为只读状态。

比如 A 索引的 shard 1 分布在 N1 节点，shard 2 分布在 N2 节点，如果 N1 节点磁盘容量超过警戒水位，索引 A 即被执行写禁止操作，成为只读索引。但是容量绝对值和百分比不能混用，比如指定了磁盘低水位为 500mb，则高水位相应的也必须使用绝对值表示。

索引的增、删、改都会对所在节点的磁盘水位产生影响，为了动态的感知磁盘水位，相应的就有了水位采集参数：

- cluster.info.update.interval

磁盘水位采集频率，即每隔多久去检查一次磁盘用量，默认为 30 秒。

- cluster.routing.allocation.disk.include_relocations

是否将正在迁移到当前节点的分片磁盘用量 (将占用的磁盘空间) ，计入当前节点的磁盘用量，默认打开。

热点问题

虽然根据实际数据更替情况，合理配置了节点的高低水位，但是随着时间推移，我们发现集群发生了热点数据倾斜问题，由于冷数据占用了大量的存储空间，导致热点数据 (当日新创建的索引)， 被迫分配到空间用量相对较少的几个节点，使得集群的负载不均。

针对日志服务等索引频繁创建、删除的场景，数据带有明显的时效性，可以考虑集群分组，对冷热数据使用不同的分配标记 (allocation attributes) ，来隔离冷热数据 (或者使用 data tier allocation) ，目的是避免访问较少的冷数据，占用磁盘容量，导致集群将新创建的索引分配到少数几个"看起来比较合适"的节点，导致热点问题出现。

Elasticsearch 新版本中已经将类似的功能集成为 data_tier 插件，详见下一小节。

针对索引创建、删除不频繁的场景，比如电商后端常见的商品搜索、订单搜索等，一方面可以考虑将集群节点分组，或者部署多个集群，将不同业务进行资源隔离。

另一方面，创建索引时需要考虑到，未来数据量的增长情况，以设置合理的分片数量，将分片尽量均匀分配到每个节点，以更合理的利用节点硬件资源；

一般来说，商品、订单等业务数据长尾效应比较明显，针对热点的店铺、类目等引起的数据倾斜问题，可以将热点数据单独拆出一个索引，配合前端的引擎代理将请求路由到对应的索引。

管理节点

系统资源吃紧，需要通过横向扩容增加集群容量。

节点扩/缩容

当向集群扩容节点时，其他节点会迁移部分分片到新节点，如果并发迁移的分片过多，可能造成瞬时的高 IO 负载，引起服务抖动。

我们可以通过 `cluster.routing.allocation.node_concurrent_incoming_recoveries` 参数控制分片迁移的速度，如果是在集群负载较高的情况下横向扩容新节点，建议分开两步操作：

- 关闭 `cluster.routing.rebalance.enable` (主要是考虑到分片移出后可能会引起集群重平衡操作)

- 通过手动对目标索引进行 index.routing.allocation.include 配置，将新节点纳入到分片的分布范围，逐个迁移索引分片。目的是减少大规模持续的分片迁移，导致集群负载继续升高，甚至发生雪球效应。

同样的，在缩容集群时，如果直接关闭节点，可能存在两个风险点：

- 在集群规模较大情况下，会有大量索引同时进行分片主从切换和分片重新分配操作，瞬时对 master 节点带来很大的负载，尤其是日志类数据的大集群。因为分片数较多，可能导致 master 节点 CPU 飙高，使得新创建索引等操作被阻塞；
- 小概率情况下。如果其他节点发生意外宕机，索引将存在数据丢失风险。

对于类似缩容，存在一定风险性的主动操作，建议与扩容类似，首先设置全部索引的 index.routing.allocation.exclude 或者直接在集群范围内设置 cluster.routing.allocation.exclude 属性将待下线的节点排除，待分片全部移出之后再关闭节点进程。

节点重启

除了横向扩容外，对节点纵向扩容，或者升级 Elasticsearch 版本，都需要对节点进行重启操作。

在重启节点时，自然会有节点的上下线操作，节点下线同时会让该节点所属的分片处于 unassigned 状态。正常情况下，集群会将这些未分配分片，重新分配到集群内其他节点上，在日常运维的节点重启操作中，这显然不是我们期望的，无端的带来了大量的 IO 操作。

正常的滚动重启操作中，建议是：

- 通过 cluster.routing.allocation.enable: none 关闭分片分配；
- 重启节点；
- 将 cluster.routing.allocation.enable 重置为 all 打开分片分配。

调控分片的物理分配

因为日志检索方面表现良好，公司决定将商品、订单等系统的检索功能也迁移到 Elasticsearch 集群，并提供了高配机器用于集群搭建。

分片在单台物理机的分配

在实际的部署过程中，有时会遇到大容量高配机器，比如 32 核 128GB 内存，我们可以考虑单节点部署，将对内存数据量扩大到 32GB 以上 (一个是对象指针压缩技术不再可用，造成内存空间膨胀，另一个是堆内存回收压力也增大，可能造成 gc 停顿时间变长)；

也可以考虑单机多节点的部署方式，在这种情况下，为了数据可用性的考虑，索引内同一分片的主副本数据，需要分配到不同的物理节点上，这时可以使用如下参数：

- cluster.routing.allocation.same_shard.host

阻止同一分片的多个实例 (主副本) 落到同一个主机，同一主机的判定条件为相同的

主机名 (host name) 和主机地址 (host address)，默认情况下该参数为关闭状态，强烈建议在单机多节点部署的情况打开该配置，避免可能的数据丢失风险。

对商品、订单的搜索场景，一般单节点的负载会控制在最大值的 50% 左右，以提供足够的余量来承载瞬时的流量高峰，为了达到这个目的，在分片分配层面，可以设置单节点的分片数上限：

- cluster.routing.allocation.total_shards_per_node

单节点最多能支撑的分片数，当节点包含的分片数高于该数值时，新分片不会在该节点创建。

分片在机房内的分配

为了数据高可用的考虑，我们在管理集群的时候，可能会考虑索引的主从分片，不要都落到某一台宿主机，或者同一个机架上，这个时候可以通过一些提示性参数，让集群在选择节点时有一定的倾向性：

- cluster.routing.allocation.awareness.attributes

设置分片分布时，会考虑将分布交叉分布到属性不同的节点上，比如集群包含两个节点 N1 node.attr.zone=zone1、N2 node.attr.zone=zone2，如果我们在 elasticsearch.yml 中设置 cluster.routing.allocation.awareness.attributes: zone，则我们新建带一个副本的索引时，集群会将同一分片的主副本交叉分布在不同的节点上。节点属性可通过

上述自定义属性方式设置。

- cluster.routing.allocation.awareness.force.zone.values

假设集群节点仍然是 N1、N2，如果 N2 因为故障宕机，默认情况下，N2 所属的分片会在 N1 节点重新恢复出来，但是在同一个节点运行相同分片的主副本并没有实际意义，这时我们在 elasticsearch.yml 中设置 cluster.routing.allocation.awareness.force.zone.values： zone1,zone2 来避免这种操作，集群会在 zone2 属性节点恢复后再将相应的副本分片恢复到该节点。

集群分片上限

系统跑起来了，那么集群最多能负载的分片数是多少呢？

对日志类的数据，通常会以应用、时间、日志级别等多个维度建立索引，这样一来集群内的总分片数变得相当可观，那么集群最多能负载的分片数是多少呢？

答案也很简单，就是节点数与单个节点能最多能支撑的分片的积，但是这里的单节点最多支撑的分片数：

- cluster.max_shards_per_node

用于限制整个集群最多能支撑的分片数，当集群内活跃分片数大于 cluster.max_shards_per_node * number_of_data_node 时，集群会阻止新索引的创建，直到有索引

被删除或者关闭（closed），使得活跃分片总数低于阈值。这里的活跃分片数是指非 closed 状态的分片，包括 unassigned / initializing / relocating / started。

不会像 cluster.routing.allocation.total_shards_per_node 真正的限制单节点的分片数。

这里列举了部分较为常用的配置，更多参数可以参阅官方文档。

小结

我们调整或干预集群的分片参数，从根本上说，是为了在集群稳定的情况下将性能最大化，从分片分配（allocation）的角度来说，稳定意味着分片尽量少的移动，性能意味着同一个索引的分片尽量均匀分布到各节点，而不要集中到少数几个节点。

索引维度分片分配

如同设计系统需要先从架构角度考虑系统模块设计，再细化到每一个应用设计应用自身的功能模块，索引分片的调整也需要先从集群角度，设定大的分配策略导向，如节点分布（shard awareness）、平衡（shard rebalance）等，再细化到每一个索引考虑单索引分片如何调整以达到最佳的表现。

再以上一节中的商品、订单检索系统为例，因为这分属两个子系统，符合我们预期的理解是即使商品检索系统负载很高，也不应该影响订单系统的检索耗时。

隔离不同索引

对不同业务所属索引进行物理隔离，实现 A 索引在执行大负载操作时，不会对 B 产生影响，前提是使用节点属性，将集群按照需要分割为多个群组。

比如集群包含以下设置的节点：

通过 index.routing.allocation.* 配置可以启用索引的分片控制，具体的：

- index.routing.allocation.include.{attribute}

  将索引分片分配到包含任一指定属性的节点上。{attribute} 可以指定为节点内置属性，如 _ip、_host 等，也可以指定为自定义属性，如 zone 等，也可以混用。

- index.routing.allocation.require.{attribute}

  将索引分片分配到指定节点上，节点必须包含指定的全部属性。

- index.routing.allocation.exclude.{attribute}

  不要将索引分配到包含任一指定属性的节点上。

如果设置索引的分片控制参数为：如果参数设置为：如果参数设置为：则索引分片不会被分配到 Node C。

排查分片分配

当我们在创建索引后，发现索引分片不能被正常分配时，可以通过 explain 接口来查看原因，如下：

```
curl -XGET '{host:port}/_cluster/allocation/explain'
```

在 response 中可以看到具体分片未被正常分配的原因，如：

```
{
  "index": "test_v1",
  "current_state": "unassigned",
  "unassigned_info": {
    "reason": "INDEX_CREATED",
    "last_allocation_status": "no"
  },
  "can_allocate": "no",
  "allocate_explanation": "cannot allocate because allocation is not permitted to any of the nodes",
  "node_allocation_decisions": [
    {
      "node_decision": "no",
      "deciders": [
        {
          "decider": "filter",
          "decision": "NO",
          "explanation": "node does not match index setting [index.routing.allocation.require] filters [node:\"xxx\",_ip:\"1.1.1.1\"]"
        }
      ]
    }
  ]
}
```

表示因为没有节点能够同时满足 node.attr.node: xxx 且 IP 地址为 1.1.1.1 的节点(response 内容适当删减了非必要信息)。

平衡索引分片在各节点的分配

在索引按业务分组隔离之后，以商品检索为例，后续又追加了商品评价索引，用于存放商品的评价记录，由于用量较低，我们希望将分组内机器资源，主要用来承载商品检索服务。

不考虑分片数据倾斜的问题，即每个分片的负载一致，我们可以将索引的分片数（主分片和副本分片的总和），设置为与节点个数一致，并通过设置索引分片，在各节点的分配个数来强迫索引在各节点间均衡分配。

要控制索引在单个节点的数量，可以通过 index.routing.allocation.total_shards_per_node 参数设置。

比如现有 6 个节点，我们可以将索引的分片数 index.number_of_shards 设置为 3，副本数 index.number_of_replicas 设置为 2，同时将索引在每个节点上的最大分片数 index.routing.allocation.total_shards_per_node 设置为 1，即可以保证索引在每个节点分配一个分片，充分利用每个节点的算力。

冷热隔离/归档

回到日志数据的问题，为了便于回溯问题，我们期望将数据的存档时间，从一周延

长为半年，但是不要求全部的数据，都有同样的可用性和响应时间：

- 一周内需要有高性能的读写能力
- 一月内数据仅需保证秒级的查询 RT 即可
- 一年内的数据不需要实时保证可读，只要能保证数据在必要情况下可恢复使用即可。

对于冷热时效明显的数据场景（比如日志类），热数据（如当日数据）的读写频率都要明显高于冷数据（如一周前数据），这个时候从成本的角度出发，我们期望将冷数据存放于容量较大、IO 及 CPU 性能较弱的存储类机器，将热数据存放于 IO、CPU 性能较好的计算类机器。

我们可以将节点按照存储优化型、计算优化型、通用型等定向优化的类型分组，使用 node.roles 将节点分组，可使用的角色包括：

- data_content 存储用户定义的业务数据，需要保证高性能的读写。
- data_hot 存储新的时序数据，读写频繁，CPU、IO 敏感型数据。
- data_warm 读写频率降低，写很少，可容忍读 RT 变高。
- data_cold 保留只读数据，比如历史记录。
- data_frozen 用于保留数据快照，比如对冷数据将其副本作为 searchable snapshot 存放到 data_frozen 节点，并关闭原始索引的副本，在保证数据可用性的情况下进一步减少空间冗余，降低数据成本。

实际应用中，通常会搭配 ILM（index lifecycle management）来使用这类 data tier 属性，如：

```
{
  "phases" : {
    "hot" : {
      "actions" : {
        "rollover" : {
          "max_age" : "1d",
          "max_size" : "5gb"
        }
      }
    },
    "warm" : {
      "min_age" : "7d",
      "actions" : {
        "forcemerge" : {
          "max_num_segments" : 1
        }
      }
    },
    "cold" : {
      "min_age" : "30d",
      "actions" : {
        "searchable_snapshot": {
          "snapshot_repository" : "snapshot_house"
        }
      }
    }
  }
}
```

将一周内数据作为热点数据，存储于 data_hot 节点，其他一月内数据作为温数据，存储于 data_warm 节点，超过一个月的数据，启用 searchable_snapshot，集群内只

保留主分片，副本备份到指定的远端路径下。

Elasticsearch 内部使用 index.routing.allocation.include._tier_preference 属性来实现这个操作，与 index.routing.allocation.include._tier 不同，_tier_preference 属性不是仅限定一种角色的节点，比如设置

```
{
    "index.routing.allocation.include._tier_preference": "data_hot,data_warm,data_cold"
}
```

表示将索引优先存储于 data_hot 节点，如果不存在 data_hot 次选 data_warm，最后使用 data_cold 兜底存储。

在索引的生命周期中，新创建的索引 _tier_preference 先设置为 data_hot，在超过热点周期后，更新 _tier_preference 为 data_warm。data_hot 将其迁移到存在的 data_warm 节点。

通过这样的手段，我们可以根据索引数据的使用场景，合理的调配硬件资源，达到成本利用的最优化。

创作人简介：

毛夏君，关注研究中间件，比如 ES，Redis，RocketMQ 等技术领域。

博客：微信公众号——tiaotiaoba_abc

3.5.6 Data stream

创作人：赵凯

审稿人：周海清

Data stream 的概念

时序性数据

时间序列数据（ time series data ）是在不同时间上收集到的数据，用于所描述现象随时间变化的情况。这类数据反映了某一事物、现象等，随时间的变化状态或程度。

总的来说，这类数据主要基于时间特性明显，随着时间的流逝，往往过去时间的数据没有现在时间的重要或者敏感。

对于 Elastisearch 处理时序性数据，有人总结了主要有以下特点：

- 由时间戳 + 数据组成。基于时间的事件，可以是服务器日志或者社交媒体流。
- 通常搜索最近事件，旧文件变得不太重要。
- 索引的使用主要基于时间，但是数据并不一定随着时间均衡分布。
- 时序性数据一旦存入后很少修改。
- 时序性数据随着时间的增加，数据量会很大。

Elastisearch 在时序性数据的使用中，往往会有以下的缺点：

- 索引随着时间增加而数目较多。
- 索引大小无法均衡。
- 管理索引成本较高，需要维护 merge 合并删除等一系列任务。
- 节点资源与冷热数据分布不匹配。

在这样的一个场景下，数据流 Data stream 应运而生。

Data stream （数据流）是 Elastic Stack 7.9 的一个新的功能。Data stream 可以跨多个索引存储只追加时序性数据，同时为查询写入等请求提供唯一的一个命名资源。Data stream 非常适合日志，事件，指标以及其他持续生成的数据。

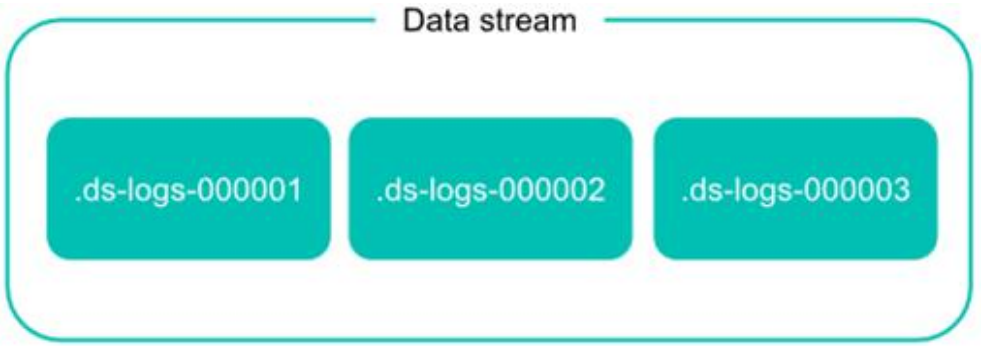
简单来说，Data stream 根据模板生成存储数据的后备索引，然后自动将搜索或者索引请求路由到存储流数据的后备索引。而这些后备索引则根据索引生命周期管理（ILM）来自动管理。

例如，你可以使用 ILM 自动将较旧的后备索引移动到较便宜的硬件上（冷热数据处理），根据索引大小自动 Rollover 出新的后备索引，或者删除到时间限制的索引。

在一定程度上，Data stream 的管理优势是利用了 ILM 的特性。但是 ILM 在普通场景下需要根据索引的别名（ alias ）逐个设置，而 Data stream 则是抛弃了 alias 的限制，可以直接批量化设置相似名称的索引，大大增加了 ILM 的使用范围。

Data stream 的组成

数据流在 Elasticsearch 集群中由一个或多个隐藏的、自动生成的后备索引组成。

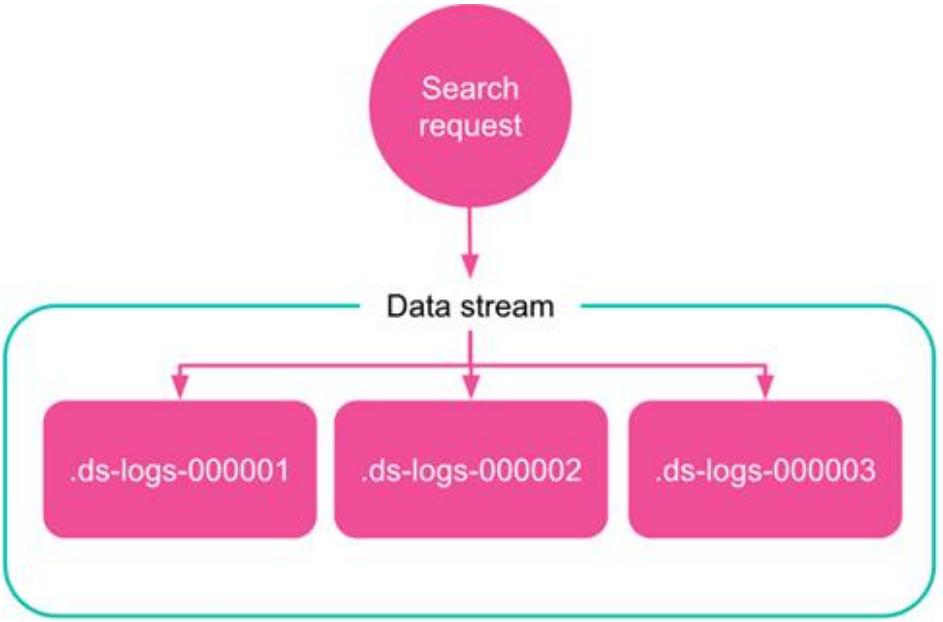


在实际的 Elasticsearch 操作中，数据流依靠索引模板来设定数据流实体的后备索引。

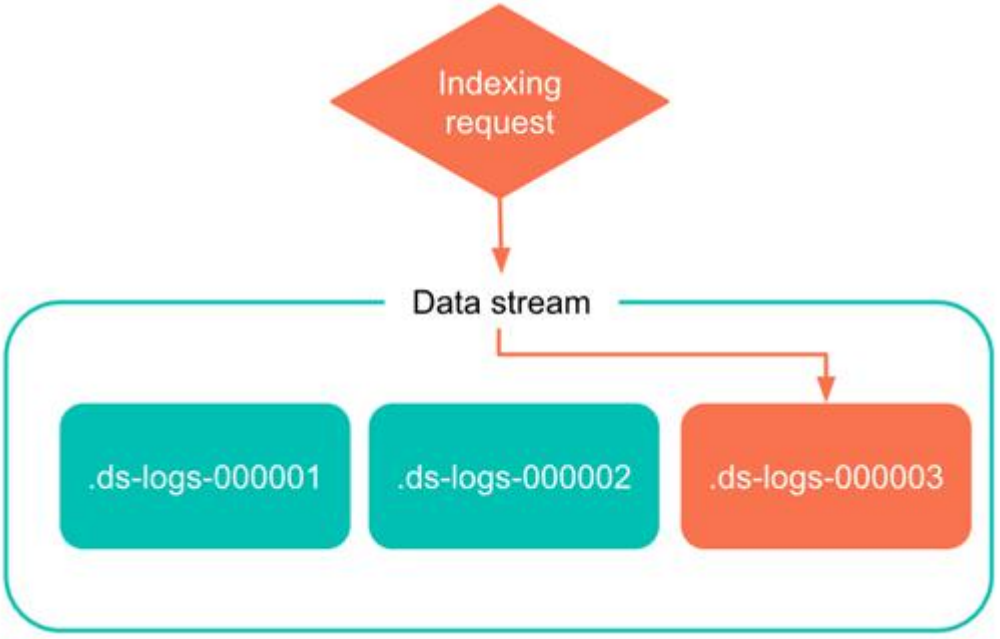
- 模板包含用于配置流的后备索引的映射和设置。
- 同一个索引模板可用于多个数据流。
- 不能删除数据流正在使用的索引模板。

每个索引到数据流的文档，必须包含一个 @timestamp 字段，映射为 date 或 date_nanos 字段类型。如果索引模板没有为 @timestamp 字段指定映射，Elasticsearch 将 @timestamp 映射为带有默认选项的日期字段。

Data stream 的读请求主要如下图，数据流自动将请求路由到其所有后备索引。



而对于写请求，数据流则将该请求自动转发给最新的后备索引。



对于写请求，有两点需要注意：

- 不能将新文档添加到其他非最新后备索引，即使直接将请求发送到这些索引也不行。
- 不能对正在写入的索引做 Clone/Close/Delete/Freeze/Shrink/Split 相关操作。

注：7.12 版本可以 Close

Data stream 的特性

生成

每个 Data stream 的后备索引都有一个 generation 数，一个六位数，零填充的整数，从 000001 开始，用作该流的 rollover 的计数。

后备索引名主要依照以下格式：

.ds--

Generation 越大，后备索引包含的数据越新。 例如，web-server-logs 数据流最新的 generation 为 34。该流的最新后备索引名为 .ds-web-server-logs-000034。

注意：某些操作（例如 shrink 或 restore ）可以更改后备索引的名称。 这些名称更改不会从其数据流中删除后备索引。

Rollover

在 Data stream 的使用中，rollover 是必不可少的条件。

创建数据流时，Elasticsearch 会自动为该 Data stream 根据 template 创建一个后备索引。 该索引还充当流的第一个写入索引。当满足一定条件时， rollover 会创建一个新的后备索引，该后备索引将成为 Data stream 的新写入索引。

当然 rollover 的条件设置主要依靠 ILM。 如果需要，你还可以手动将数据 rollover 。

追加

由于时序性数据的特征，Data stream 的设计场景中，数据是只追加的，极少需要修改删除。如果实际需要修改删除，则可以考虑以下操作：

- 对于数据流只能通过 update by query 或者 delete by query 操作，不能进行 update 或者 delete 文档。
- 需要 delete 或者 update 文档，则直接对后备索引操作。
- 需要经常删除或者修改文档的，请用索引别名或者索引模板，不要对 Data stream 操作。

Data stream 的使用

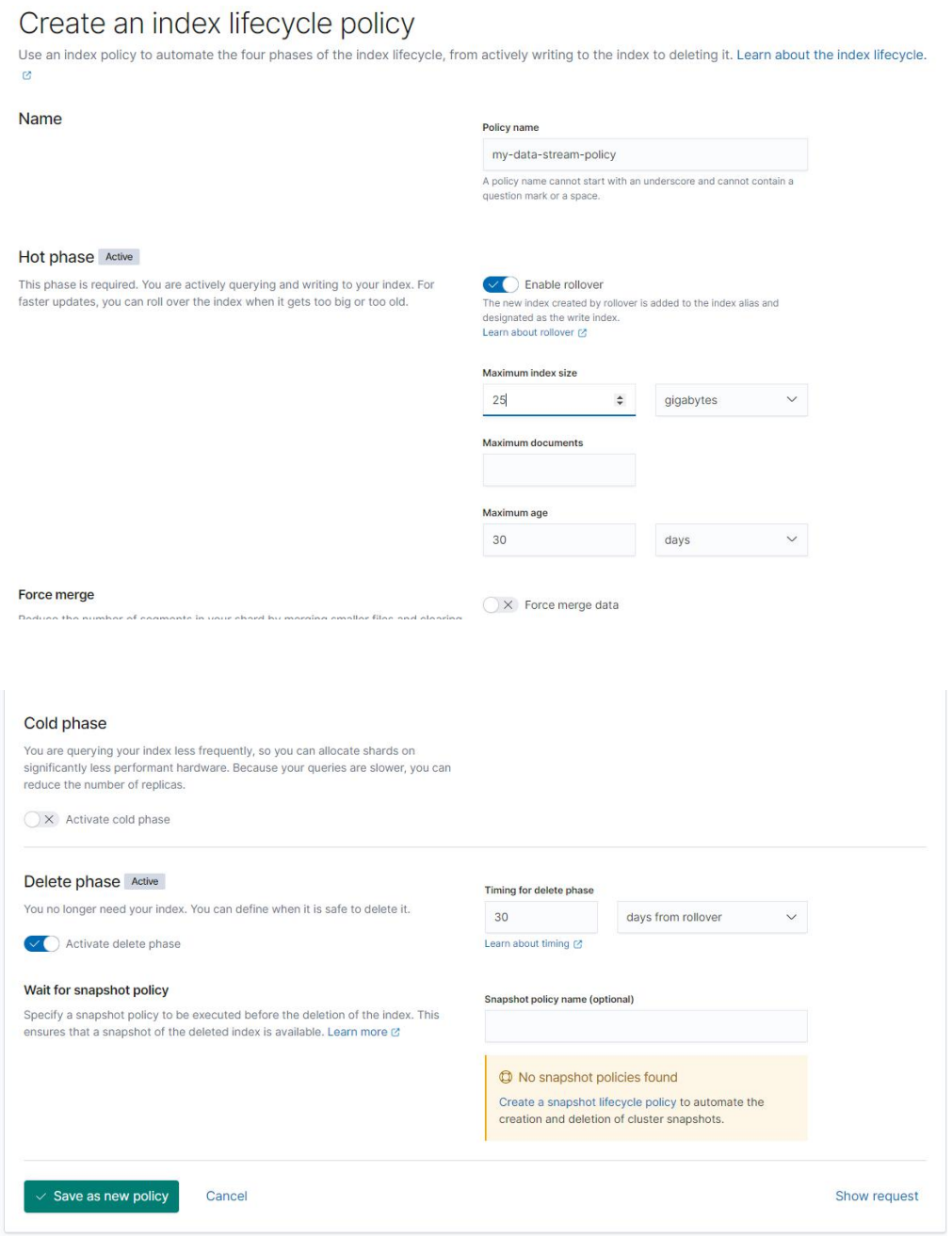
创建索引生命周期管理策略 ILM

索引生命周期管理策略 ILM 的主要配置细节见索引周期管理一章，此处主要做 hot 和 delete 阶段的设置，用于 rollover 的引用。

相关命令：

```
PUT /_ilm/policy/my-data-stream-policy
{
  "policy": {
    "phases": {
      "hot": {
        "actions": {
          "rollover": {
            "max_size": "25GB"
          }
        }
      },
      "delete": {
        "min_age": "30d",
        "actions": {
          "delete": {}
        }
      }
    }
  }
}
```

Kibana 图形界面： Stack Management -> Index Lifecycle Policies -> Create policy



注意：

- rollover 设置中，文档数和最大存在时间是相对敏感的配置参数，由于 Elasticsearch 并不是实时监控 ILM 的执行任务（默认十分钟），最终结果并不一定完全一致。
- ILM 任务判断中，max_size 判断的是主分片的大小，而不是整个索引的大小。
- 新版本下，max_size 的判断并不敏感，可能是因为索引的主分片 size 大小会被 merge 后收缩，需要有一定时间的观察。如下图。测试之下，200MB 之下的 max_size 会失效。建议 max_size 设置参数不要太小。

创建索引模板

索引模板是后备索引设置，以及 mapping 的主要配置来源，此处不展开延伸。主要设置 Data stream 相关的部分。

相关命令：

```
PUT /_index_template/my-data-stream-template
{
  "index_patterns": [ "my-data-stream*" ],
  "data_stream": { },
  "priority": 200,
  "template": {
    "settings": {
      "index.lifecycle.name": "my-data-stream-policy"
    }
  }
}
```

注意：

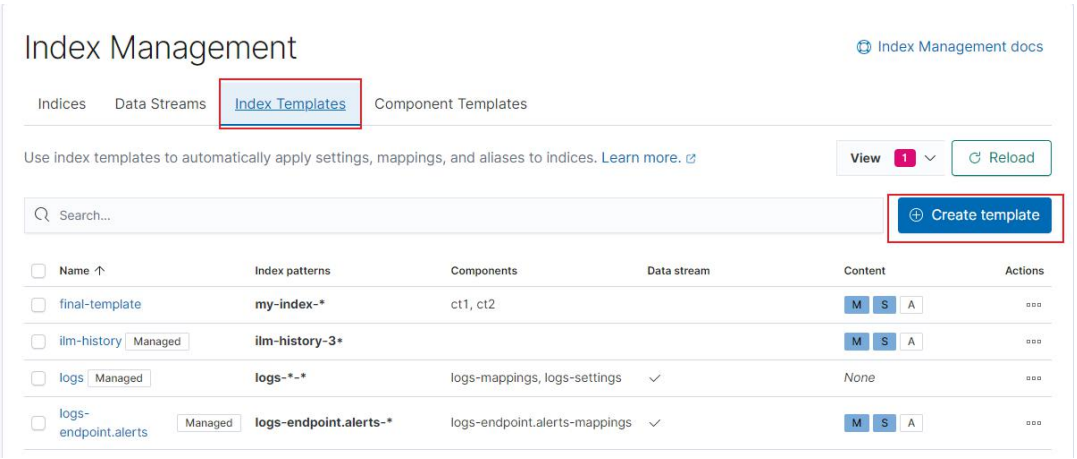
- 定义 data_stream 为一个空的 object ，这是必要的。
- Template 中使用了上一步创建的 ILM 策略 my-data-stream-policy。

此外，还需要注意两点：

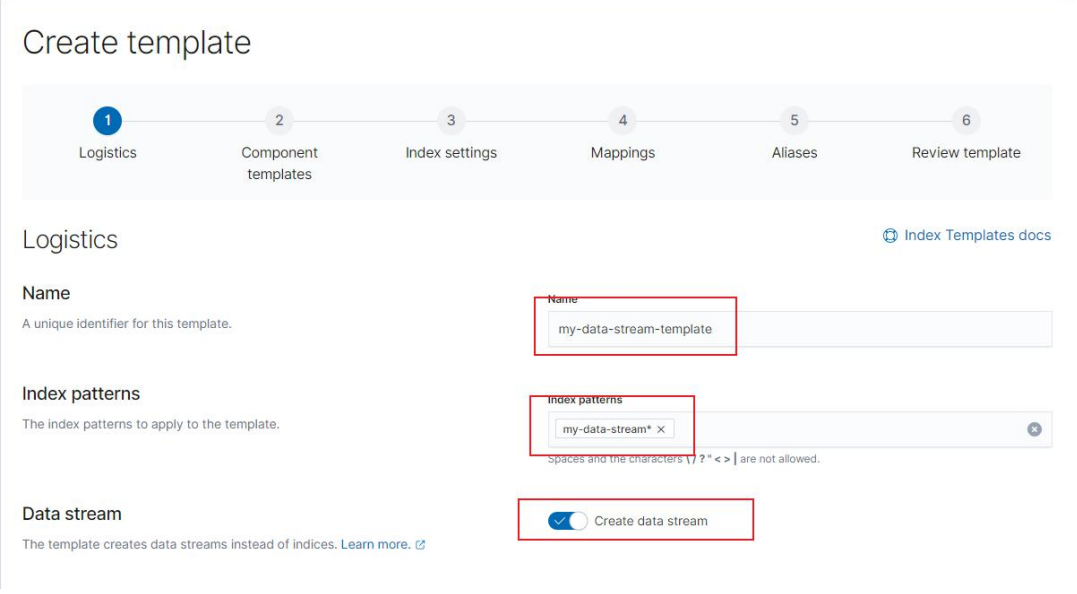
- Elasticsearch 有一些内置索引模板如 metric-- 和 logs-- ，默认优先级 priority 是 100。如果有重名使用，则可以调高优先级，防止被默认的覆盖。
- 索引模板默认将 @timestamp 字段设置为 date 属性。

Kibana 界面:

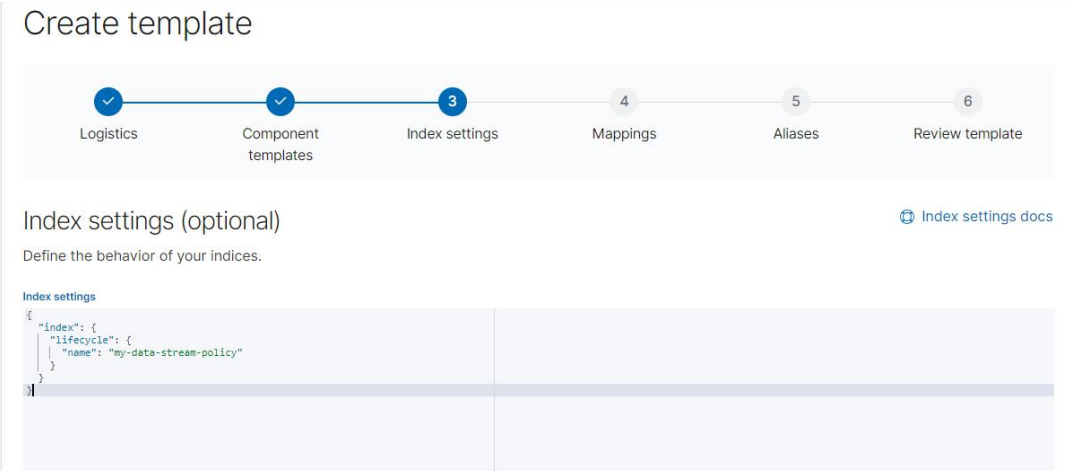
Stack Management -> Index Management -> Index Templates -> Create template



创建 template，不要创建旧版索引，并打开数据流标签：



设置生命周期管理策略，其他设置此处省略，一直下一步至创建完成。



创建 Data stream

可以自动利用 template 的匹配模式新增文档创建：

```
POST /my-data-stream/_doc/
{
  "@timestamp": "2020-12-06T11:04:05.000Z",
  "user": {
    "id": "vlb44hny"
  },
  "message": "Login attempt failed"
}
```

Response:

```
{
  "_index" : ".ds-my-data-stream-000001",
  "_type" : "_doc",
  "_id" : "8ZadZXkBkhA9X9yUbI17",
  "_version" : 1,
  "result" : "created",
  "_shards" : {
    "total" : 2,
    "successful" : 1,
    "failed" : 0
  },
  "_seq_no" : 0,
  "_primary_term" : 1
}
```

也可以直接 PUT 创建一个空的 Data stream。

```
PUT /_data_stream/my-data-stream
```

删除

删除命令：

```
DELETE /_data_stream/my-data-stream
```

删除数据流会将数据流的后备索引一起删除。

使用 Data stream

此处对数据流的操作主要以命令为主，Kibana 界面支持较少。

新增数据

Data stream 在新增数据时是只追加的模式，因此在固定 id 和 bulk 的模式下，op_type 是指定 create 的。

如下面命令：

```
POST my-data-stream/_create/1
{"@timestamp":"2020-12-07T11:06:07.000Z","test":1}
```

或者：

```
PUT /my-data-stream/_bulk?refresh
{"create":{ }}
{ "@timestamp": "2020-12-08T11:04:05.000Z", "user": { "id": "vlb44hny" }, "message": "Log in attempt failed" }
{"create":{ }}
{ "@timestamp": "2020-12-08T11:06:07.000Z", "user": { "id": "8a4f500d" }, "message": "Log in successful" }
{"create":{ }}
{ "@timestamp": "2020-12-09T11:07:08.000Z", "user": { "id": "l7gk7f82" }, "message": "Log out successful" }
```

如果并不指定，文档的 id，则可以使用默认的 _doc ，如下：

```
POST my-data-stream/_doc/
{"@timestamp":"2020-12-07T11:06:07.000Z","test":1}
```

获取 Data stream 状态

使用 Data stream stats API 查看 Data stream 的状态。

```
GET /_data_stream/my-data-stream/_stats?human=true
```

Response:

```
{
  "_shards" : {
    "total" : 4,
    "successful" : 2,
    "failed" : 0
  },
  "data_stream_count" : 1,
  "backing_indices" : 1,
  "total_store_size" : "5kb",
  "total_store_size_bytes" : 5151,
  "data_streams" : [
    {
      "data_stream" : "my-data-stream",
      "backing_indices" : 1,
```

```
        "store_size" : "5kb",
        "store_size_bytes" : 5151,
        "maximum_timestamp" : 1607252645000
      }
    ]
}
```

可见 my-data-stream 的大下和后备索引数量。

同时需要用 _ilm/explain 获取 Data stream 后备索引所在的 ILM 策略状态。

```
GET my-data-stream/_ilm/explain
```

Response:

```
{
  "indices" : {
    ".ds-my-data-stream-000001" : {
      "index" : ".ds-my-data-stream-000001",
      "managed" : true,
      "policy" : "my-data-stream-policy",
      "lifecycle_date_millis" : 1620907943375,
      "age" : "7.95s",
      "phase" : "hot",
      "phase_time_millis" : 1620907943567,
      "action" : "rollover",
      "action_time_millis" : 1620907943661,
```

```
        "step" : "check-rollover-ready",
        "step_time_millis" : 1620907943661,
        "phase_execution" : {
          "policy" : "my-data-stream-policy",
          "phase_definition" : {
            "min_age" : "0ms",
            "actions" : {
              "rollover" : {
                "max_size" : "25gb"
              }
            }
          },
          "version" : 1,
          "modified_date_in_millis" : 1620907939978
        }
      }
    }
}
```

上图可见，这个数据的 000001 索引主要处于 hot 阶段，策略名称是 logs 等信息。具体参数可见于 ILM 的相关定义。

手动 rollover Data stream

使用 rollover API，手动 rollover Data stream。

```
POST my-data-stream/_rollover
```

Response:

```
{
  "acknowledged" : true,
  "shards_acknowledged" : true,
  "old_index" : ".ds-my-data-stream-000001",
  "new_index" : ".ds-my-data-stream-000002",
  "rolled_over" : true,
  "dry_run" : false,
  "conditions" : { }
}
```

再 GET 相关 Data stream 状态，后备索引增加。

```
GET /_data_stream/my-data-stream/
```

Response:

```
{
  "data_streams" : [
    {
      "name" : "my-data-stream",
      "timestamp_field" : {
        "name" : "@timestamp"
      },
      "indices" : [
        {
```

```
            "index_name" : ".ds-my-data-stream-000001",
            "index_uuid" : "AJBi0g3fRyG8-1tiH2UD2Q"
          },
          {
            "index_name" : ".ds-my-data-stream-000002",
            "index_uuid" : "AgOLGMSBSYWb4X-ID8uwtg"
          }
        ],
        "generation" : 2,
        "status" : "GREEN",
        "template" : "my-data-stream-template",
        "ilm_policy" : "my-data-stream-policy",
        "hidden" : false
      }
    ]
}
```

Reindex Data stream

使用 reindex API 去复制数据到一个 Data stream。由于 Data stream 的只追加特性，在 op_type 中要选择为 create 。

```
POST /_reindex
{
  "source": {
    "index": "test"
  },
  "dest": {
```

```
      "index": "my-data-stream",
      "op_type": "create"
    }
  }
```

Response:

```
{
  "took" : 80,
  "timed_out" : false,
  "total" : 1,
  "updated" : 0,
  "created" : 1,
  "deleted" : 0,
  "batches" : 1,
  "version_conflicts" : 0,
  "noops" : 0,
  "retries" : {
    "bulk" : 0,
    "search" : 0
  },
  "throttled_millis" : 0,
  "requests_per_second" : -1.0,
  "throttled_until_millis" : 0,
  "failures" : [ ]
}
```

Delete/Update by query

针对 Data stream 只能 delete/update by query 。

相关命令：

```
POST /my-data-stream/_update_by_query
{
  "query": {
    "match": {
      "user.id": "l7gk7f82"
    }
  },
  "script": {
    "source": "ctx._source.user.id = params.new_id",
    "params": {
      "new_id": "XgdX0NoX"
    }
  }
}

POST /my-data-stream/_delete_by_query
{
  "query": {
    "match": {
      "user.id": "vlb44hny"
    }
  }
}
```

Delete update 后备索引数据

在后备索引删除或者修改，需要注意下面三个要素:

- 文档 id。
- 文档所在的后备索引。
- 如果是修改文档，则需要其 _seq_no 和 _primary_term 两个参数。

主要操作如下:

先获取文档所需的要素信息，设置 seq_no_primary_term 为 true。

```
GET /my-data-stream/_search
{
  "seq_no_primary_term": true,
  "query": {
    "match": {
      "message": "Login attempt failed"
    }
  }
}
```

获得结果:

```
{
  "took" : 621,
  "timed_out" : false,
  "_shards" : {
    "total" : 2,
    "successful" : 2,
    "skipped" : 0,
    "failed" : 0
  },
  "hits" : {
    "total" : {
      "value" : 1,
      "relation" : "eq"
    },
    "max_score" : 0.8630463,
    "hits" : [
      {
        "_index" : ".ds-my-data-stream-000001",
        "_type" : "_doc",
        "_id" : "9ZakZXkBkhA9X9yUZo2P",
        "_seq_no" : 0,
        "_primary_term" : 1,
        "_score" : 0.8630463,
        "_source" : {
          "@timestamp" : "2020-12-06T11:04:05.000Z",
          "user" : {
            "id" : "vlb44hny"
          },
```

```
            "message" : "Login attempt failed"
          }
        }
      ]
    }
  }
```

然后修改命令：

```
PUT /.ds-my-data-stream-000001/_doc/9ZakZXkBkhA9X9yUZo2P?if_seq_no=0&if_primary_term=1
{
  "@timestamp": "2020-12-07T11:06:07.000Z",
  "test": 4
}
```

Response:

```
{
  "_index" : ".ds-my-data-stream-000001",
  "_type" : "_doc",
  "_id" : "9ZakZXkBkhA9X9yUZo2P",
  "_version" : 2,
  "result" : "updated",
  "_shards" : {
    "total" : 2,
    "successful" : 1,
    "failed" : 0
```

```
    },
    "_seq_no" : 1,
    "_primary_term" : 1
}
```

或者删除命令：

```
DELETE /.ds-logs-1-1-000002/_doc/3
```

关于修改 Data stream 的 mapping 和 setting

Data stream 的 setting 和 mapping 修改主要还是基于 Elasticsearch 默认的修改规则。总结一下，主要有以下几点：

- 新增字段不影响。
- 已存在的配置不可更改。
- 修改的 template 只能应用于未来新增的索引。

因此，如果需要修改不可更改的配置，可以考虑 reindex 或者修改 template 后手工 Rollover Data stream。

关于 Data tiers

Data tiers 也称数据层，是一个在 7.10 版本的一个新概念。

Data tiers 主要的一个特点是将节点角色（ node roles ）与索引生命周期所需要的节点属性（ attribute ）结合，直接可以在制定 Elasticsearch 节点角色时配置，不需要再去设置 attribute 。Data tiers 的概念也是对时序性数据分层管理的优化配置。

Data tiers 的数据节点默认是都配置的，即 data_content/data_hot/data_warm/data_cold（ chsw ）都具备。

Tiers 的定义

- Content tier

Content tier 节点存储的数据，往往定义为与时序性数据相反的常态化数据，比如商品种类这种随着时间推移保持相对不变。这种数据并不能根据冷热数据性质分层存储。

此类数据有以下特点：

- Content tier 节点通常需要较高的计算性能，要求处理能力比 IO 吞吐能力高，需要处理复杂的搜索和聚合并快速返回结果。
- 对数据内容的获取，即文档内容本身获取比时序性数据要少。
- 这类数据索引需要配置为一个或多个副本。

- Hot tier

Hot tier，热层是时间序列数据的 Elasticsearch 入口点，最新存储的时间序列数据。Hot tier 的数据也是会被查询最多的数据。因此热层中的节点在读取和写入时都需要快

速，这需要更多的硬件资源和更快的存储（ SSD）。属于数据流 （ Data stream ）的新索引会自动分配给热层。

- Warm tier

即温层，一旦查询时间序列数据的频率低于 hot tier 中最近索引的数据，便可以将其移至 warm tier 。 warm tier 通常保存最近几周的数据。 仍然允许进行更新，但可能很少。通常，warm tier 中的节点不需要像 hot tier 中的节点一样快。

- Cold tier

冷层的数据一般查询频率非常低，且不会被更新。 但是 cold tier 仍然是响应查询层。 随着数据过渡到 cold tier，可以对其进行压缩和去副本。Cold tier 节点的机器配置可以相对较低。

tier_preference

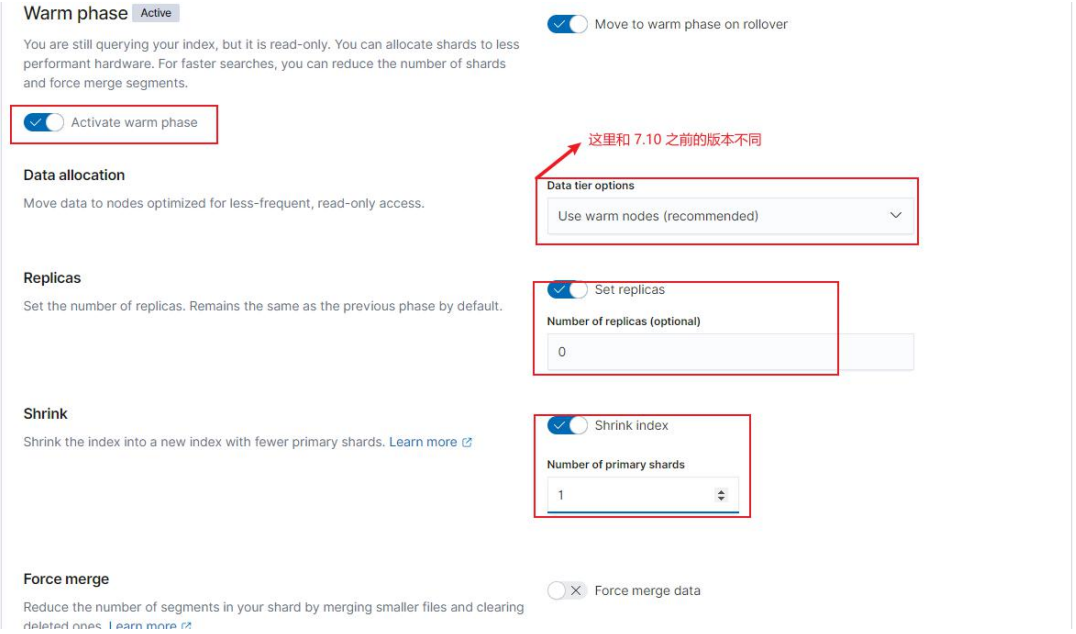
index.routing.allocation.include._tier_preference 是 Data tiers 的主要配置方式，在分片数据的时候使用 tier_preference 指定数据节点的分配。

tier_preference 的设置会有三种情况：

- 创建正常索引时，默认情况下，Elasticsearch 将 index.routing.allocation.include._tier_preference 设置为 data_content ，以将索引分片自动分配给内容层。

- 创建数据流时，Elasticsearch 会将后备索引的 index.routing.allocation.include._tier_preference 设置为 data_hot，以自动将索引分片分配给热层。
- 显式设置 index.routing.allocation.include._tier_preference，选择索引需要的数据节点。 如果将层首选项设置为 null，则 Elasticsearch 在分配期间将忽略数据层角色，依照其它参数分配。

相关的图形和命令配置如下：



上图时在索引生命周期管理中选择 Data tiers 节点。

```
PUT _index_template/template_demo
{
  "index_patterns": ["demo-*"],
```

```
    "data_stream": {},
    "priority": 200,
    "template": {
      "settings": {
        "number_of_shards": 2,
        "index.lifecycle.name": "demo",
        "index.routing.allocation.include._tier_preference": "data_hot"
      }
    }
  }
```

上面命令中设置索引模板匹配 demo-* 的索引的分配策略为 "index.routing.allocation.include._tier_preference":"data_hot"

创作人简介：

赵凯，平时喜欢阅读 elastic 官网，对 Elasticsearch 较为熟悉。 学习一门技术，官网永远是最好的学习文档。 在西安，我们也建立了自己的圈子，欢迎西安的小伙伴们一起交流，共同进步。。

博客：https://dr-kyle.github.io/

3.5.7 索引生命周期管理

创作人：赵凯

审稿人：朱永生

Elasticsearch 在 6.7 版本正式加入索引生命周期管理，旨在管理 Elasticsearch 中的索引。

通常我们使用 elasticsearch 的时候，index 命名都是 xxx-YYYY.MM.dd 类似这样的格式，每天创建一个 index，这需要我们自己创建 index，或者通过自动创建。

- 每天创建一个 index，但是每天的数据量又非常少，这对集群来说是不利的。
- 如果是自动创建的话，集群 index 和 shard 数过多，那么在每天的 00:00 时，大量的 index 同时创建，这时我们就会发现集群的写入速度会变慢，可能会发生 index 写入拒绝的情况。
- 集群需要对冷热数据进行分离，性能好的机器放最近频繁查询的数据，随着时间推移，数据查询不在频繁，需要将数据迁移到性能较差的机器上。

以上这些我们可以使用 Elasticsearch 提供的索引生命周期管理功能能很好的解决，接下来我们了解一下 索引生命周期管理。

索引生命周期的四个阶段

- Hot:

  index 正在查询和更新，一般性能好的机器会设置为 Hot 节点来进行数据的读写。

- Warm:

  index 不再更新，但是仍然需要查询，节点性能一般可以设置为 Warm 节点。

- Cold:

  index 不再被更新，且很少被查询，数据仍然可以搜索，但是能接受较慢的查询，节点性能较差，但有大量的磁盘空间。

- Delete:

  数据不需要了，可以删除。

  节点的类型可以通过一下两种方式设置，推荐第二种，第一种后续可能会弃用。

  第一种：

```
# elasticsearch.yml
# node.attr.xxx: xxx
node.attr.data: warm
```

  第二种（推荐）：

```
# elasticsearch.yml
# data_content, data_hot, data_warm, data_cold
```

```
# 配置该节点既属于内容层又属于热层
node.roles: ["data_hot", "data_content"]
```

这四个阶段按照 Hot，Warm，Cold，Delete 顺序执行，上一个阶段没有执行完成是不会执行下一个阶段的，对于不存在的阶段，会跳过该阶段进入到下一个阶段。

生命周期默认每 10 分钟检测一次，可以通过集群的配置动态修改，如下：

```
PUT _cluster/settings
{
  "transient": {
    "indices.lifecycle.poll_interval": "10m"
  }
}
```

生命周期管理 API

每个阶段支持的行为会在下一章节进行介绍，此章节仅仅为了介绍 API。

创建生命周期管理策略

min_age 参数指定从 index 创建后多长时间进入到该阶段。

以下示例是指从当 index 创建时间超过 10 天后，进入到 warm 阶段，将 segment 数量 merge 为 1，warm 阶段完成后，进入 delete 阶段，index 创建时间超过 30 天后，将 index 删除。

```
PUT _ilm/policy/my_policy
{
  "policy": {
    "phases": {
      "warm": {
        "min_age": "10d",
        "actions": {
          "forcemerge": {
            "max_num_segments": 1
          }
        }
      },
      "delete": {
        "min_age": "30d",
        "actions": {
          "delete": {}
        }
      }
    }
  }
}
```

查看生命周期管理策略

```
# 查看所有的生命周期管理策略
GET _ilm/policy
# 查看特定的生命周期管理策略# GET _ilm/policy/<policy_id>
```

删除生命周期管理策略

```
DELETE _ilm/policy/<policy_id>
```

触发生命周期策略中特定步骤的执行

current_step

```
- phase 当前阶段的名称
- action 当前行为的名称
- name 当前步骤的名称
```

next_step

```
- phase 想要执行阶段的名称
- action 想要执行行为的名称
- name 想要执行步骤的名称
```

```
POST _ilm/move/my-index-000001
{
  "current_step": {
    "phase": "new",
    "action": "complete",
    "name": "complete"
  },
  "next_step": {
```

```
        "phase": "warm",
        "action": "forcemerge",
        "name": "forcemerge"
      }
}
```

移除生命周期管理策略

```
# POST <target>/_ilm/remove
POST my-index-000001/_ilm/remove
```

生命周期重试

```
# POST <index>/_ilm/retry
POST my-index-000001/_ilm/retry
```

查看当前索引生命周期管理状态

```
GET /_ilm/status
```

查看一个或多个索引的当前生命周期状态

```
# GET <target>/_ilm/explain
GET my-index-000001/_ilm/explain
```

启动索引生命周期管理插件

```
POST _ilm/start
```

停止索引生命周期管理插件

```
POST /_ilm/stop
```

四个阶段支持的行为

索引生命周期每个阶段支持的行为如下：

- Hot
  - Set Priority
  - Unfollow
  - Force Merge
  - Rollover
- Warm
  - Set Priority
  - Unfollow
  - Read only
  - Allocate
  - Shrink
  - Force Merge
  - Migrate

- Cold
  - Set Priority
  - Unfollow
  - Allocate
  - Migrate
  - Freeze
  - Searchable Snapshot

- Delete
  - Wait For Snapshot
  - Delete

行为

Set Priority

设置索引的优先级，一旦进入到某阶段，就设置索引的优先级，节点重新启动后，优先级较高的索引将会优先恢复。

参数:

- priority: 正整数。

例如：设置 warm 阶段 index 的优先级为 50

```
PUT _ilm/policy/my_policy
{
  "policy": {
    "phases": {
      "warm": {
        "actions": {
          "set_priority" : {
            "priority": 50
          }
        }
      }
    }
  }
}
```

Rollover

当 index 满足三个条件中的任何一个时，会将别名指向新生成的索引。

参数：

- max_age

  达到索引创建的最大时间

- max_docs

  达到指定的文档数后

- max_size

  index 达到指定的大小时，主分片的大小，不包含副本。

以上三个参数至少应该存在一个

例如：当前 index 主分片大小达到 100GB 或文档数超过 100000000 或者 index 创建超过 7 天 生产新的 index：

```
PUT _ilm/policy/my_policy
{
  "policy": {
    "phases": {
      "hot": {
        "actions": {
          "rollover" : {
            "max_size": "100GB",

"max_docs": 100000000,
            "max_age": "7d"
          }
        }
      }
    }
  }
}
```

Unfollow

将 follow 索引转换为正常索引。

例如：

```
PUT _ilm/policy/my_policy
{
  "policy": {
    "phases": {
      "hot": {
        "actions": {
          "unfollow" : {}
        }
      }
    }
  }
}
```

Allocate

指定 index 的副本数，迁移 index 到某些节点，冷热节点数据迁移依赖此步骤。

参数：

- number_of_replicas

  指定 index 的副本数

- include

  将 index 迁移到具有指定属性之一的节点

- exclude

  将 index 迁移到不包含指定属性的节点

- require

  将 index 迁移到具有所有指定属性的节点

Note: include 满足其中一个就可以， require 必须全部满足。

例如：到达 warm 阶段将 index 的备份数设置为 2，并且将 index 迁移至属性 box_type 包含 hot, warm 且不包含 cold 的节点。

```
PUT _ilm/policy/my_policy
{
  "policy": {
    "phases": {
      "warm": {
        "actions": {
          "allocate" : {
            "number_of_replicas" : 2,
            "include" : {
```

```
                    "box_type": "hot"
                },
                "exclude" : {
                    "box_type": "cold"
                },
                "require" : {
                    "box_type": "hot,warm"
                }
            }
        }
    }
  }
}
```

Force Merge

指定 index 合并后 segment 数量，在 hot 阶段使用时，必须包含 rollover ，merge 时会将 index 设置为只读。

参数：

- max_num_segments

  segment 最大数量

- index_codec

  压缩文件存储， default: LZ4

例如：warm 阶段将 index 的 segments 合并为 1。

```
PUT _ilm/policy/my_policy
{
  "policy": {
    "phases": {
      "warm": {
        "actions": {
          "forcemerge" : {
            "max_num_segments": 1
          }
        }
      }
    }
  }
}
```

Read only

将 index 设置为只读。

例如：

```
PUT _ilm/policy/my_policy
{
  "policy": {
    "phases": {
```

```
        "warm": {
          "actions": {
            "readonly" : { }
          }
        }
      }
    }
}
```

Shrink

index 设置为只读，然后将 index 缩小为具有更少的的 shard， 缩小后的 index 名称为 shrink-

参数：

- number_of_shards

  合并后的主分片数

例如：warm 阶段将 index 的 shard 数合并为 1 个。

```
PUT _ilm/policy/my_policy
{
  "policy": {
    "phases": {
      "warm": {
```

```
            "actions": {
              "shrink" : {
                "number_of_shards": 1
              }
            }
          }
        }
      }
```

Freeze

最大程度减少 index 的内存占用。

例如：cold 阶段将 index freeze，释放内存。

```
PUT _ilm/policy/my_policy
{
  "policy": {
    "phases": {
      "cold": {
        "actions": {
          "freeze" : { }
        }
      }
    }
  }
}
```

Migrate

通过更新 index.routing.allocation.include._tier_preference 设置，将 index 移动到对应的数据层，如果指定了 allocate，会在迁移前先将副本数减少。如果在热阶段和冷阶段没有指定 allocate 分配选项，ILM 会自动注入迁移操作，如果要禁用可以将 enabled 设置为 false。

参数：

- enabled

  default: true, 控制 ILM 在此阶段是否自动迁移索引

例如：warm 阶段禁用迁移操作， 主动将 index 备份数设置为 1，并且将 index 迁移至属性 rack_id 为 one 或者 two 的节点。

```
PUT _ilm/policy/my_policy
{
  "policy": {
    "phases": {
      "warm": {
        "actions": {
          "migrate" : {
            "enabled": false
          },
          "allocate": {
            "number_of_replicas": 1,
```

```
                "include" : {
                  "rack_id": "one,two"
                }
              }
            }
          }
        }
      }
    }
```

Searchable Snapshot

生成可搜索快照，在 7.10 版本还处于 beta，在新版可能会有所更改。

在 delete action 步骤中默认会删除快照，如果需要保留，在 delete action 中将 delete_searchable_snapshot 设置 false

参数：

- snapshot_repository

  Required，指定存储快照的位置

- force_merge_index

  Boolean，default: true，如果索引在先前的操作中已经使用了 force merge， 则可搜索快照操作不会执行强制合并。

例如：在 cold 阶段生成快照。

```
PUT _ilm/policy/my_policy
{
  "policy": {
    "phases": {
      "cold": {
        "actions": {
          "searchable_snapshot" : {
            "snapshot_repository" : "backing_repo"
          }
        }
      }
    }
  }
}
```

Wait For Snapshot

等待制定的 SLM 策略执行，然后在删除索引，为了确保删除的索引快照是可用的。

参数：

- policy

  required， SML 策略的名字

例如：delete 阶段等待 SLM 策略执行，然后删除索引。

```
PUT _ilm/policy/my_policy
{
  "policy": {
    "phases": {
      "delete": {
        "actions": {
          "wait_for_snapshot" : {
            "policy": "slm-policy-name"
          }
        }
      }
    }
  }
}
```

Delete

删除 index

参数：

- delete_searchable_snapshot

  boolean, default: true, 是否删除 cold 阶段创建的 searchable snapshot。

例如：index 创建 90 天后，删除 index

```
PUT _ilm/policy/my_policy
{
  "policy": {
    "phases": {
      "delete": {
        "min_age" : "90d",
        "actions": {
          "delete" : { }
        }
      }
    }
  }
}
```

通过 alias 使用 ILM

创建生命周期策略

warm 阶段将 index 分配给节点属性 data 为 warm 的节点，cold 阶段将 index 分配给节点属性 data 为 cold 的节点。

节点属性可以通过 elasticsearch.yml 进行配置或环境变量设置。

```
# 启动命令
bin/elasticsearch -Enode.attr.data=warm
# elasticsearch.yml# node.attr.xxx: xxx# 建议使用 node.roles 进行配置，可以参考 通过 data tiers 使用 ILM 这一章节# node.attr 后续版本可能不在使用

node.attr.data: warm
```

创建生命周期策略，在 index 创建 1 天后进入 hot 阶段，设置优先级为 100，当 index 主分片大小超过 50gb 或者 index 文档数超过 500000000 或者 index 创建超过 2 天生成新的 index

warm 阶段将 index 迁移至属性 data 为 warm 的节点。

cold 阶段将 index 副本数设置为 1 并将 index 迁移至属性 data 为 cold 的节点。

当 hot ，warm，cold 阶段的动作都完成并且 index 创建达到 7 天，删除 index。

```
PUT _ilm/policy/logx_policy
{
  "policy": {
    "phases": {
      "hot": {
        "min_age": "1d",
        "actions": {
          "set_priority": {
            "priority": 100
```

```
        },
        "rollover": {
          "max_age": "2d",
          "max_docs": 500000000,
          "max_size": "50gb"
        }
      }
    },
    "warm": {
      "min_age": "1d",
      "actions": {
        "set_priority": {
          "priority": 50
        },
        "allocate": {
          "include": {
            "data" : "warm"
          }
        }
      }
    },
    "cold": {
      "min_age": "1d",
      "actions": {
        "set_priority": {
          "priority": 0
        },
        "allocate": {
          "number_of_replicas": 1,
          "include" : {
            "data": "cold"
```

```
            }
          }
        }
      },
      "delete": {
        "min_age": "7d",
        "actions": {
          "delete": {}
        }
      }
    }
  }
}
```

创建索引模板，将生命周期应用到 index

设置 shard 数为 2， 备份数 为 1， 生命周期策略为 logx_policy，滚动别名为 logx

```
PUT _index_template/logx-template
{
  "index_patterns" : ["logx-*"],
  "priority" : 1,
  "template": {
    "settings" : {
      "index" : {
        "number_of_shards" : "2",
        "number_of_replicas" : "1",
        "lifecycle.name": "logx_policy",
```

```
            "lifecycle.rollover_alias": "logx"
          }
        }
      }
    }
```

创建第一个 index，以下两种形式任选一种即可， index 格式必须满足该正则 ^.-\d+$* ，example：logs-000001

```
PUT logx-000001
{
  "aliases": {
    "logx": {
      "is_write_index": true
    }
  }
}# OR 带创建日期的 index# PUT /<logx-{now/d}-1> with URI encoding:
PUT /%3Clogx-%7Bnow%2Fd%7D-1%3E
{
  "aliases": {
    "logx": {
      "is_write_index": true
    }
  }
}
```

后续的数据读写均使用固定别名 logx

通过 Data stream 使用 ILM

创建生命周期策略

创建生命周期策略，在 index 创建 1 天后进入 hot 阶段，设置优先级为 100，当 index 主分片大小超过 50gb 或者 index 文档数超过 500000000 或者 index 创建超过 2 天生成新的 index，warm 阶段将 index 迁移至属性 data 为 warm 的节点，cold 阶段将 index 副本数设置为 1 并将 index 迁移至属性 data 为 cold 的节点，当 hot ，warm，cold 阶段的动作都完成并且 index 创建达到 7 天，删除 index。

```
PUT _ilm/policy/logx_policy
{
  "policy": {
    "phases": {
      "hot": {
        "min_age": "1d",
        "actions": {
          "set_priority": {
            "priority": 100
          },
          "rollover": {
            "max_age": "2d",
            "max_docs": 500000000,
            "max_size": "50gb"
          }
        }
```

```
        },
        "warm": {
          "min_age": "1d",
          "actions": {
            "set_priority": {
              "priority": 50
            },
            "allocate": {
              "include": {
                "data" : "warm"
              }
            }
          }
        },
        "cold": {
          "min_age": "1d",
          "actions": {
            "set_priority": {
              "priority": 0
            },
            "allocate": {
              "number_of_replicas": 1,
              "include" : {
                "data": "cold"
              }
            }
          }
        },
        "delete": {
      "min_age": "7d",
```

```
            "actions": {
              "delete": {}
            }
          }
        }
      }
    }
```

创建索引模板，将生命周期应用到 index

与通过 alias 的形式区别：模板不需要指定 index.rollover_alias，也不需要手动创建第一个 index，直接将数据写入符合模板的 index 即可，至于这个 index 在 Elasticsearch 中对应几个 index，我们无需关注。

```
PUT _index_template/logx-template
{
  "index_patterns" : ["logx-*"],
  "priority" : 1,
  "data_stream": { },
  "template": {
    "settings" : {
      "index" : {
        "number_of_shards" : "2",
        "number_of_replicas" : "1",
        "lifecycle.name": "logx_policy"
      }
    }
  }
}
```

创建 data stream

```
POST /logx-business/_doc/
{
    "@timestamp":"2021-04-13T11:04:05.000Z",
    "message":"Loginattemptfailed"
}# OR
PUT /_data_stream/logx-business
```

后续的数据读写均使用固定 index: logx-business

通过 Data tiers 使用 ILM

data tiers （ 数据层 ）是具有相同数据角色的节点的集合：

- Content tier （ 内容层 ）节点处理诸如产品目录之类的内容的索引和查询负载。
- Hot tier （ 热层 ） 节点处理诸如日志或指标之类的时间序列数据的索引负载，并保存你最近，最常访问的数据。
- Warm tier （ 温层 ）节点保存的时间序列数据访问频率较低，并且很少需要更新。
- Cold tier （ 冷层 ）节点保存时间序列数据，这些数据偶尔会被访问，并且通常不会更新。

推荐冷热分离采用 data tiers 这种方式，节点可以通过如下配置方式配置：

```
# elasticsearch.yml
# data_content, data_hot, data_warm, data_cold
# 配置该节点既属于内容层又属于热层
node.roles: ["data_hot", "data_content"]
```

创建生命周期策略

warm 阶段将 index 迁移至 warm 节点，cold 阶段禁用 migrate，将 index 分配给 rack_id 为 one 或 two 的节点。

创建生命周期策略，在 index 创建 1 天后进入 hot 阶段，设置优先级为 100，当 index 主分片大小超过 50gb 或者 index 文档数超过 500000000 或者 index 创建超过 2 天生成新的 index，warm 阶段将 index 迁移至 warm 节点，cold 阶段将 index 副本数设置为 1，禁用 migrate， 并将 index 迁移至属性 data 为 cold 的节点，当 hot ，warm，cold 阶段的动作都完成并且 index 创建达到 7 天，删除 index。

```
PUT _ilm/policy/logx_policy
{
  "policy": {
    "phases": {
      "hot": {
        "min_age": "1d",
        "actions": {
          "set_priority": {
            "priority": 100
          },
          "rollover": {
```

```
            "max_age": "2d",
            "max_docs": 500000000,
            "max_size": "50gb"
          }
        }
      },
      "warm": {
        "min_age": "1d",
        "actions": {
          "set_priority": {
            "priority": 50
          },
          "migrate" : {
          }
        }
      },
      "cold": {
        "min_age": "1d",
        "actions": {
          "set_priority": {
            "priority": 0
          },
          "allocate": {
            "number_of_replicas": 1,
            "include" : {
              "data": "cold"
            }
          },
          "migrate" : {
            "enabled": false
          }
```

```
        }
      },
      "delete": {
        "min_age": "7d",
        "actions": {
          "delete": {}
        }
      }
    }
  }
}
```

创建索引模板，将生命周期应用到 index

设置 shard 数为 2， 备份数 为 1， 生命周期策略为 logx_policy

```
PUT _index_template/logx-template
{
  "index_patterns" : ["logx-*"],
  "priority" : 1,
  "data_stream": { },
  "template": {
    "settings" : {
      "index" : {
        "number_of_shards" : "2",
        "number_of_replicas" : "1",
        "lifecycle.name": "logx_policy"
      }
    }
  }
}
```

```
创建 data stream
POST /logx-business/_doc/
{
    "@timestamp":"2021-04-13T11:04:05.000Z",
    "message":"Loginattemptfailed"
}# OR
PUT /_data_stream/logx-business
```

后续的数据读写均使用固定 index: logx-business

3.5.8 Canvas

创作人：王涛

审稿人：吴斌

创建演示资料是一个十分耗时费力的过程，因为除了使用 JSON 代码外，还需要对演示数据进行额外处理，才能方便受众查看和理解，所以花费的时间会更长。

即使对于从柱状图截图，并将图片放到演示资料中这样简单的工作，如果需要对柱状图中的数据进行定期更新，这项工作也会变得十分枯燥乏味。令人遗憾的是，你可能经常需要做这些工作：导出数据，对数据进行清理，将结果粘贴到演示资料中，添加图片，等等。这个过程很快就会让你到无比痛苦和抓狂。

Canvas 是什么？

Canvas 是 Kibana 中内置的一项演示工具。

通过 Canvas，用户可创建既能直接从 Elasticsearch 提取实时数据、且符合完美像素要求的演示资料和幻灯片文档。这意味着你无需对演示资料进行手动更新，便可以获得基于最新数据的图片、图形元素和图表。并且，Canvas 功能十分灵活，绝不局限于本篇文章中所提到的主要用例。

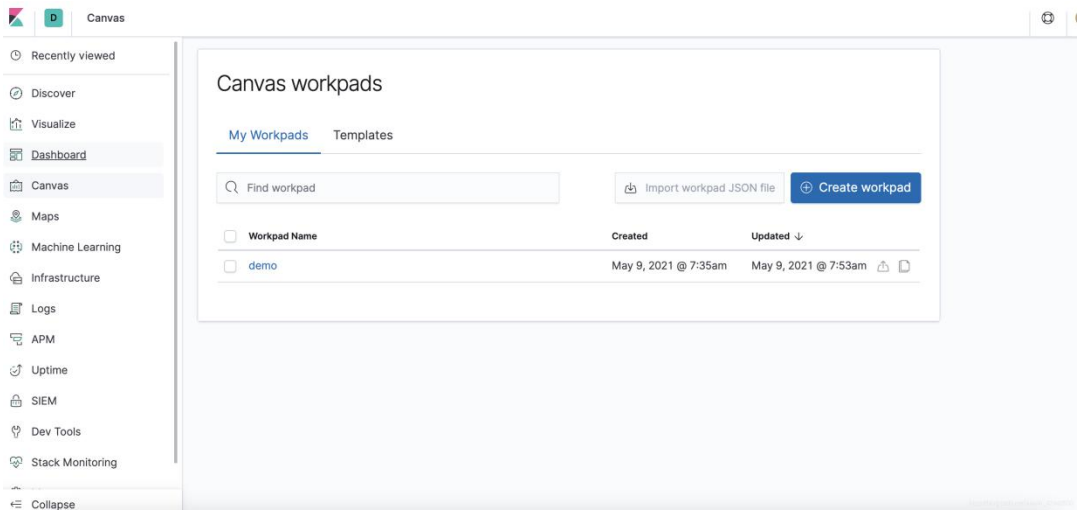
使用步骤

准备工作

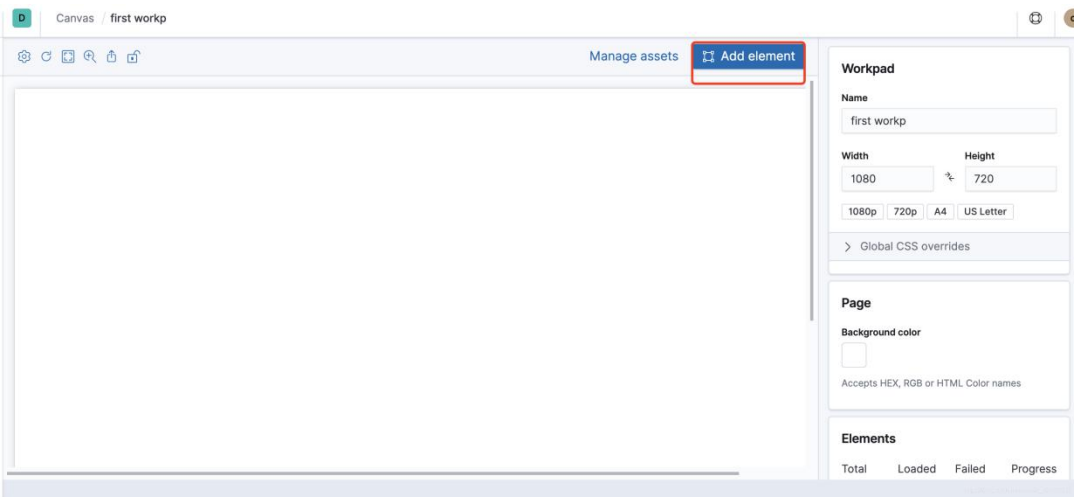
如要开始使用 Canvas，你需要安装下列两项：

- Elasticsearch，用于存储数据并对数据进行索引
- Kibana，用作 UI

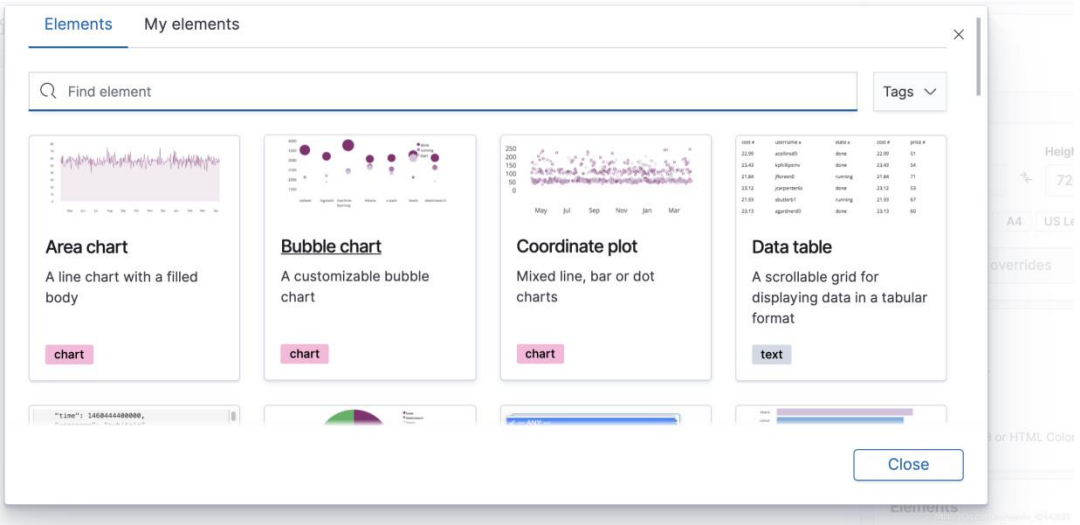
打开 Kibana，我们点击 Canvas:



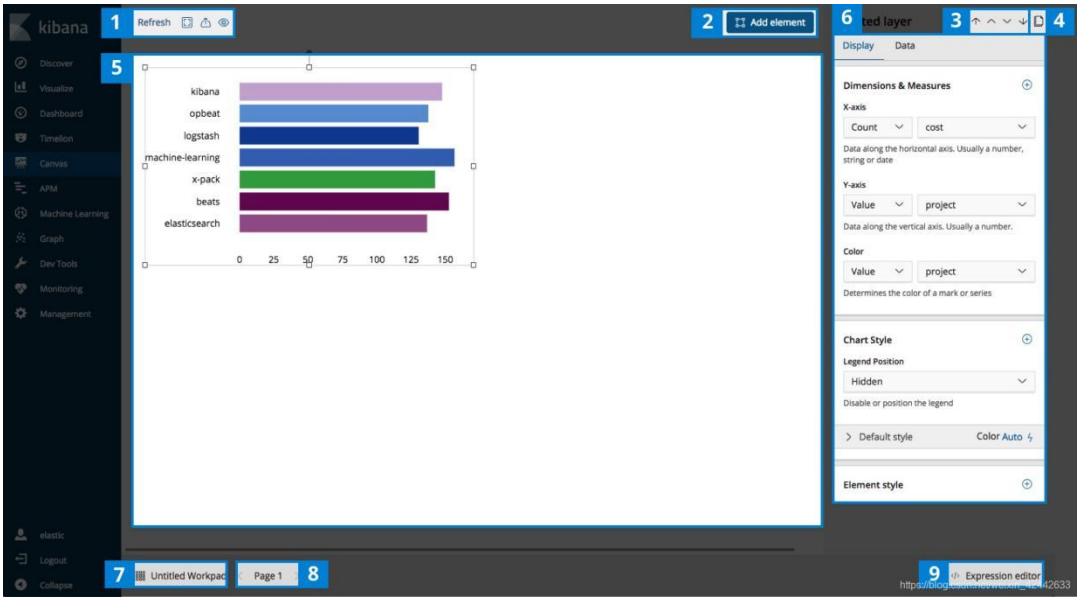
点击 Create workpad ，为你的 Workpad 命名，名称不可重复：



选择我们第一个元素：



图解



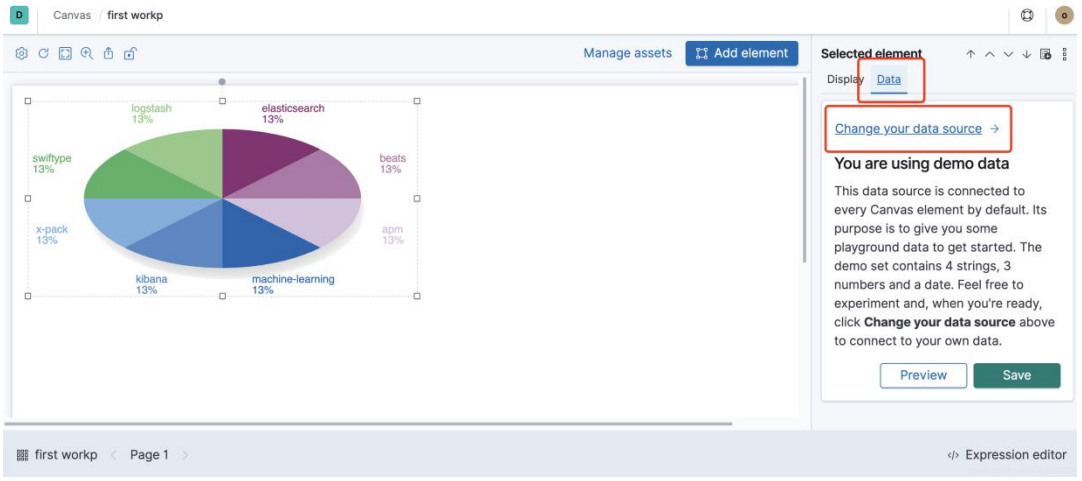
按钮

- 数据刷新间隔 - 设置 Canvas 多长时间检查一次 Elasticsearch 中的数据是否有更新。
- 切换全屏 - 切换 “演示模式”。
- 导出 Workpad - 将 Workpad 导出为 PDF 文件。
- 切换编辑侧栏 - 隐藏上图中的第 6 个区块。

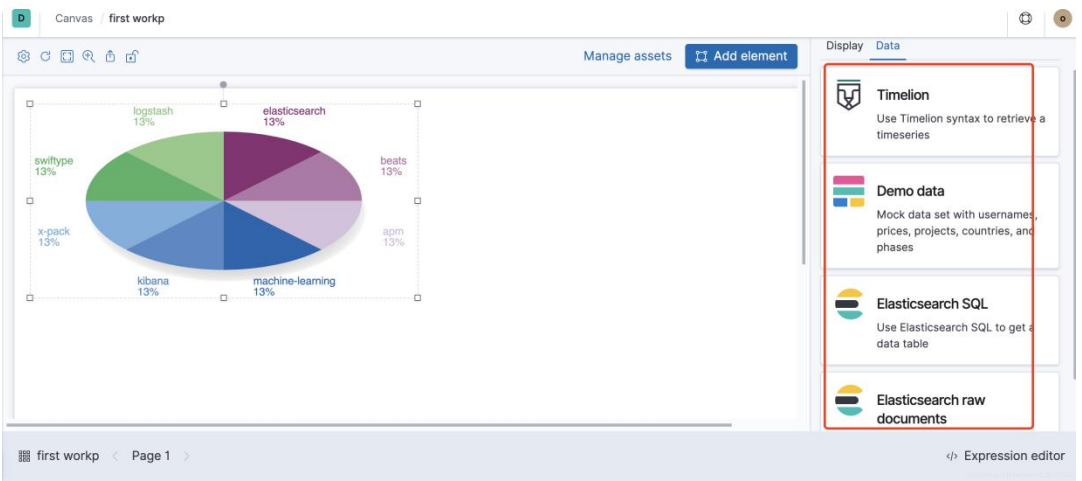
1. 添加元素 - 这将会打开元素选择器，以便你向 Canvas Workpad 中添加图形、图表、图像等内容。
2. 元素层次控件 - 选择将哪一元素置于顶层，允许你进行组合和隐藏。
3. 复制元素 - 注意：必须选中某项元素，然后方可使用此功能。
4. Canvas Workpad - 主要工作区域。

5. 编辑控件 - 上下文感知面板，该面板可以针对所选中的元素（例如字体、颜色、定制样式表等）显示属性编辑项。
6. 打开 Canvas 主菜单 - 查看不同的 Workpad，复制、导入、导出以及删除 Workpad。
7. 页面控件 - 添加新页面或者在现有页面之间进行切换。
8. 元素代码编辑器 - 如果拿不准的话，那就编写代码吧。你可以无比灵活地调整查询、格式和管道。

数据源选择

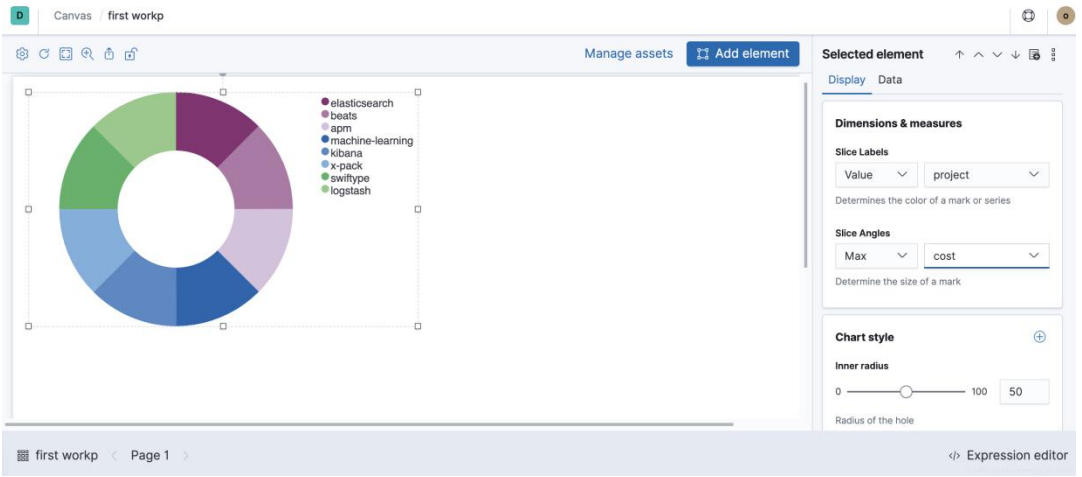


点击 Change your data source



- Demo Data（演示数据）： 这是一个小型的样本数据集，你最初创建新的 Canvas 元素时使用的便是这一数据源。主要用于测试目的。
- Elasticsearch Raw Documents（Elasticsearch 原始文档）： 此项能够让你访问
- Elasticsearch 中的原始数据。
- Timelion： Timelion 为你在 Elasticsearch 中处理时序数据提供了一种专用方法。
- Elasticsearch SQL：与原始文档类似，此项能够让你在 Elasticsearch 中使用 SQL 语法功能访问数据。

我们选择 Elasticsearch SQL 设置我们的 sql 这样就可以展示我们的数据了。



再也不用手动更新演示资料啦！

总结

如果使用 Canvas 的话，用户不仅可以省去持续更新演示资料中可视化的枯燥工作，而且可以使用强大的工具和筛选器来实时调整数据，从而奉上一场灵活、动态且富有说服力的精彩演示。

3.5.9 Space

创作人：齐乐

审稿人：吴斌

Space 功能介绍

Space 功能可以将 Kibana 划分为多个工作空间，并基于权限控制使不同的用户看到不同的工作空间。空间可管理的对象包含 Dashboards 等可视化内容以及 Kibana 自带的标签页功能如 Dev 和 Monitoring 等。

Space 功能默认自动开启，且会自动创建 Default 空间，当创建了其他空间之后，登录 Kibana 时会要求你选择工作空间。如果想要关闭 Space 功能，只需将 kibana.yml 配置文件中的 xpack.spaces.enabled 设置为 false 即可。

Space 实现原理

Space 的权限控制由 Elasticsearch 的 Security 模块提供的 Application privileges 实现。此功能对 Elasticsearch 存储的数据无任何权限控制，是专门为其他自定义应用程序，存储其权限在 Elasticsearch 的 Role 中而设置。主要包含三部分结构：

- application：表示应用程序名称，Space 权限命名为 kibana-.kibana。
- privileges：表示权限，Space 细分为各个小功能的三种权限：ALL、READ 和 NONE。
- resources：表示权限应用范围，Space 用于区分空间。

示例如下图所示：通过 GET _security/role/a 命令获取 Role a 的权限，拥有此 Role 的用户具有 A 空间的部分权限和 B 空间的全部权限。

```
"applications" : [
  {
    "application" : "kibana-.kibana",
    "privileges" : [
      "feature_discover.read",
      "feature_canvas.all",
      "feature_maps.all",
      "feature_ml.all",
      "feature_visualize.all"
    ],
    "resources" : [
      "space:a"
    ]
  },
  {
    "application" : "kibana-.kibana",
    "privileges" : [
      "space_all"
    ],
    "resources" : [
      "space:b"
    ]
  }
],
```

图 1 Role 中存储的 Space 权限

负责管理 Space 功能的管理员用户，需要赋予 kibana_admin 角色的权限，其设置如下图所示：

```
{
  "kibana_admin" : {
    "cluster" : [ ],
    "indices" : [ ],
    "applications" : [
      {
        "application" : "kibana-.kibana",
        "privileges" : [
          "all"
        ],
        "resources" : [
          "*"
        ]
      }
    ],
    "run_as" : [ ],
    "metadata" : {
      "_reserved" : true
    },
    "transient_metadata" : {
      "enabled" : true
    }
  }
}
```

图 2 kibana_admin 的权限

Space 操作实践

Space 的创建、修改和删除

通过 Kibana 界面的 Stack Management >> Kibana >> Spaces 标签页进行相关操作：

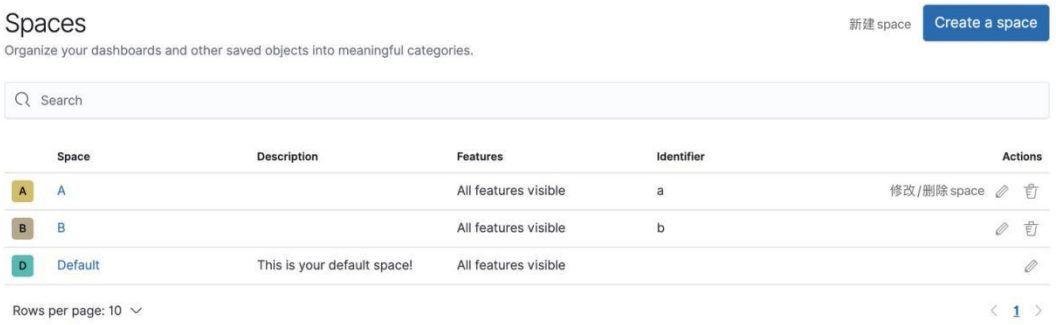


图 3 Space 管理界面

创建或修改 Space 信息包含如下内容：

- Space 基本信息：包含名字、URL 地址信息（切换 Space 工作空间会修改 Kibana 的 URL 地址为对应 Space，一旦设置不可修改）、描述信息（可为空）和头像照片。

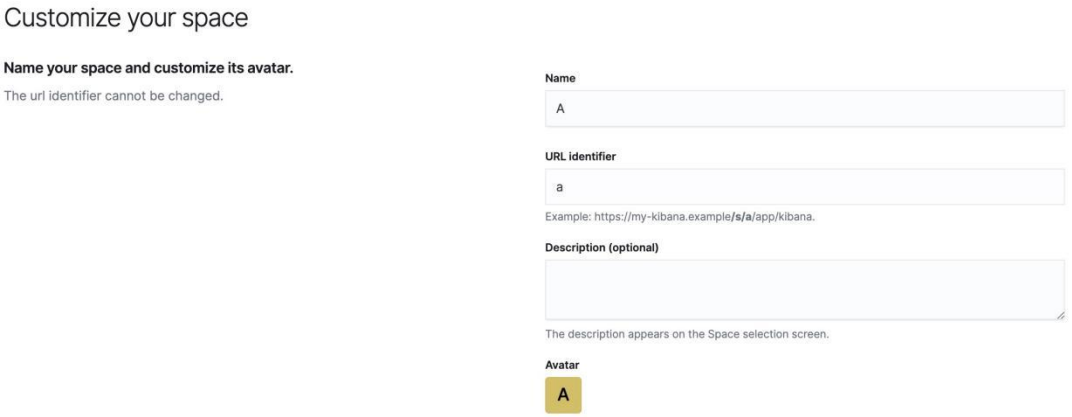


图 4 Space 基本信息

- Space 功能可见性：控制当前 Space 空间中 Kibana 各个功能标签页的可见性，可以将其隐藏以防止用户使用，如果想禁用需要控制 Role 权限实现。

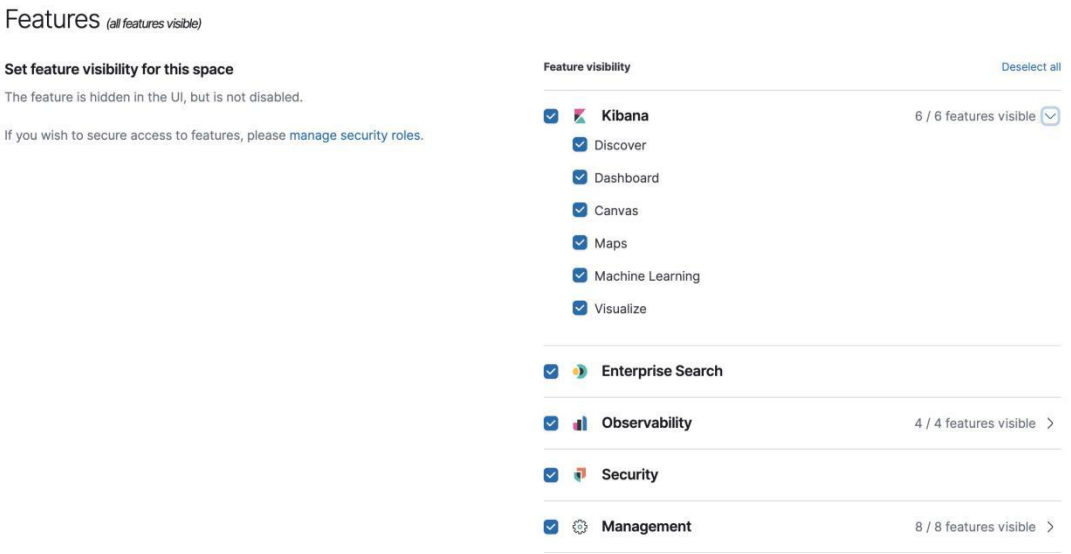


图 5 Space 功能可见性

需要注意的是删除 Space 会同时删除空间内存储的工作对象。

设置角色的 Space 权限

通过 Kibana 界面的 Stack Management >> Security >> Roles 标签页进行相关操作。

此标签页除了可以设置 Role 的 Elasticsearch 相关权限（cluster / index 等）外，还可以设置 Kibana 的 Space 权限。如上文所述，其信息存在 Role 的 Application 之中，且可以为不同的 Space 设置不同的权限如下图所示

图 6 Role 中的多个 Space

具体 Space 权限设置如图 8 所示：可以直接设置全部权限为 All 或者 Read ，也可以使用自定义方式设置每个小功能独立权限。

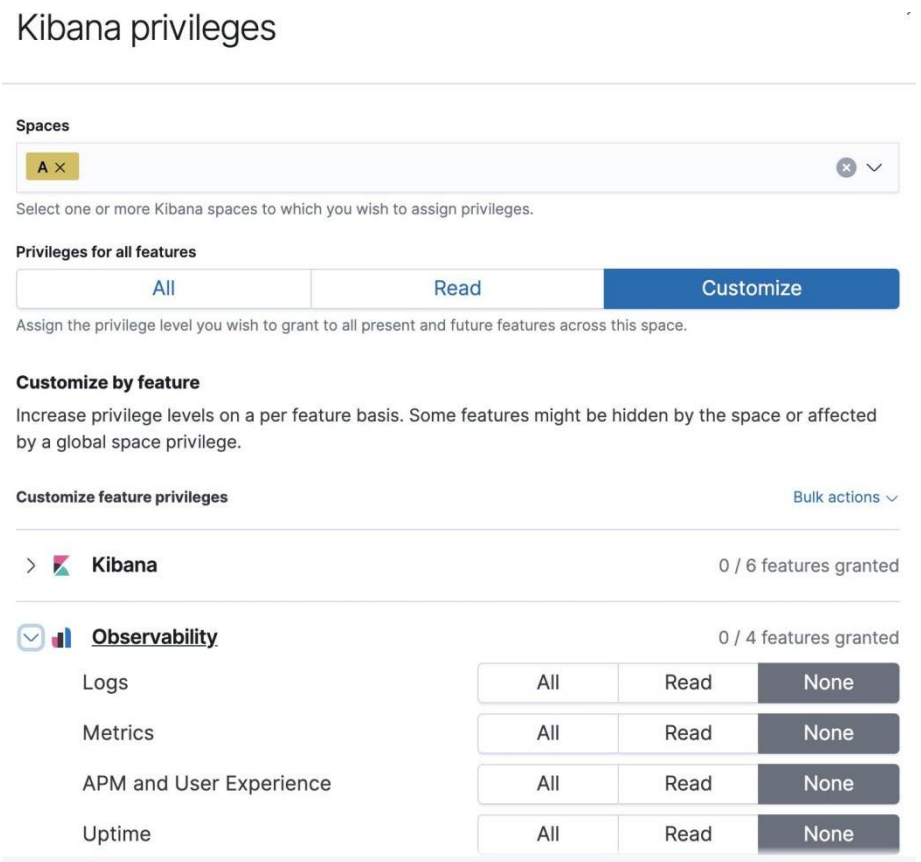


图 7 Role 中 Space 权限设置

管理并分享 Spaces 之间的工作对象

通过 Kibana 界面的 Stack Management >> Kibana >> Saved Objects 标签页进行相关操作。

Space 功能的目的就是将 Kibana 的工作对象分到不同空间，所以多个 Spaces 之间的对象复制是必不可少的操作，具体操作如下图：选择对象的 Copy to space 选项，然后选择要复制到的目的 Space 。

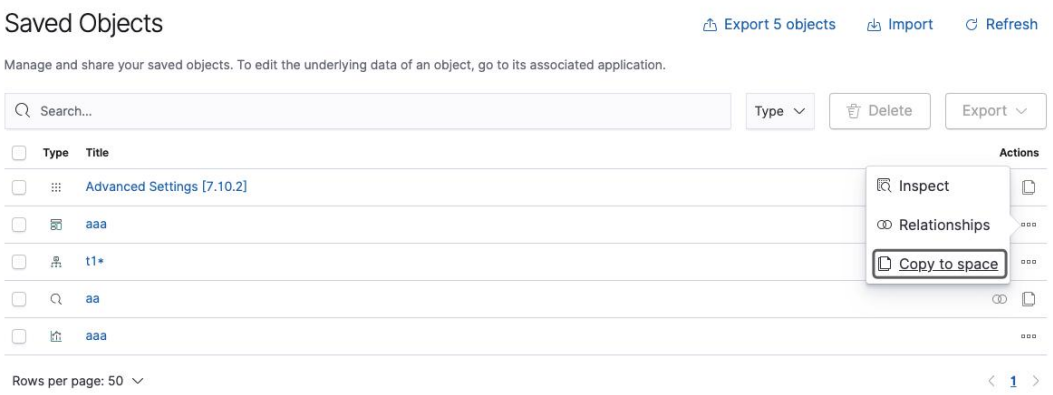


图 8 选择要迁移的对象

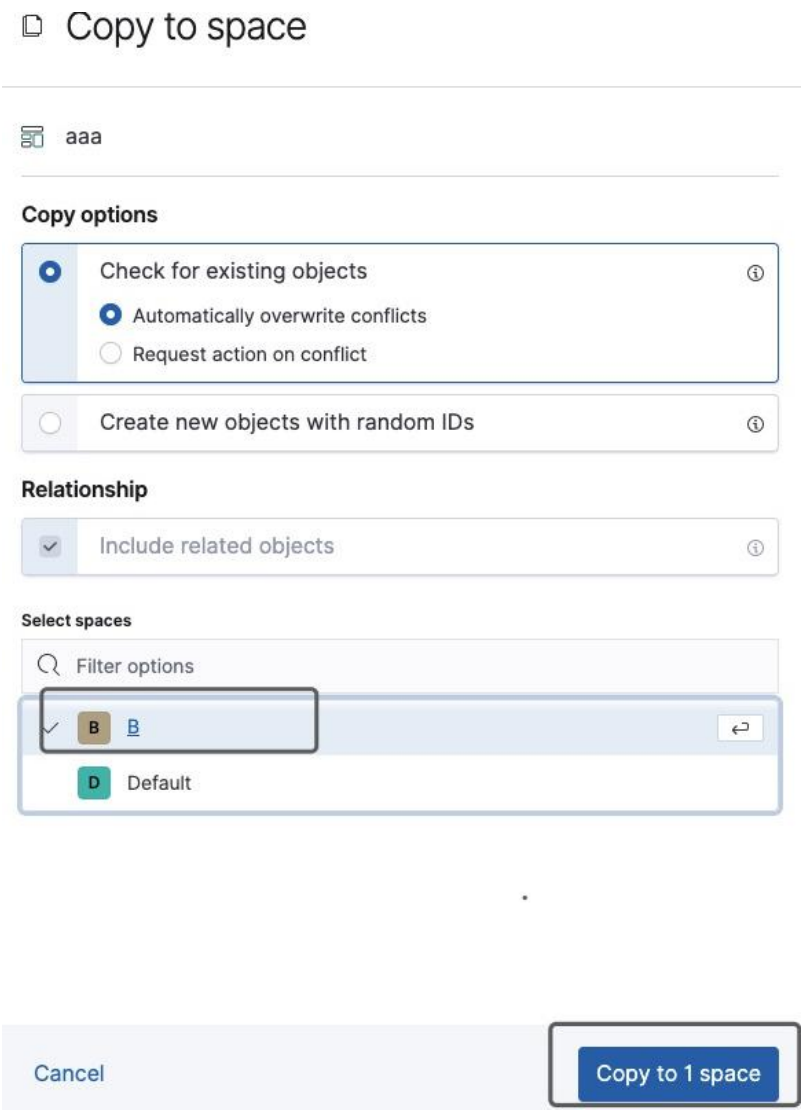


图 9 选择迁移的目的空间

并且还可以将工作对象导出到文件，用于批量迁移工作对象到其他 Space 或者迁移到其他的 Kibana 实例。具体操作如下图：选择要导出的对象，然后点击 Export 导出到文件；切换 Space 或者在其他 Kibana 实例中通过 Import 可以导入文件存储的 Kibana 工作对象。

图 10 选择要导出的工作对象

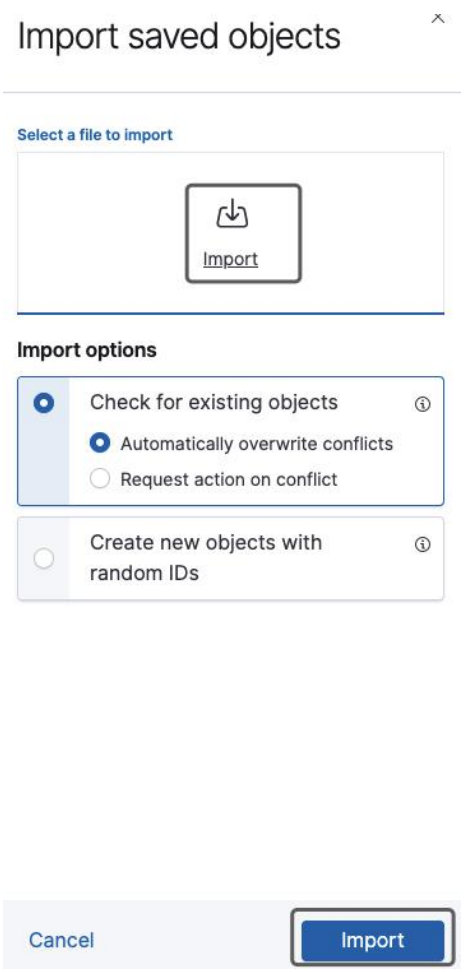


图 11 Import 选择文件进行对象导入

Space 自定义配置

通过 Kibana 界面的 Stack Management >> Kibana >> Advanced Settings 标签页进行相关操作。

我们可以修改一些 Kibana 的默认配置，但大部分配置只应用于当前 Space。例如可以修改进入 Space 的初始路由界面：从默认 Home 界面修改为 Dashboards 界面：

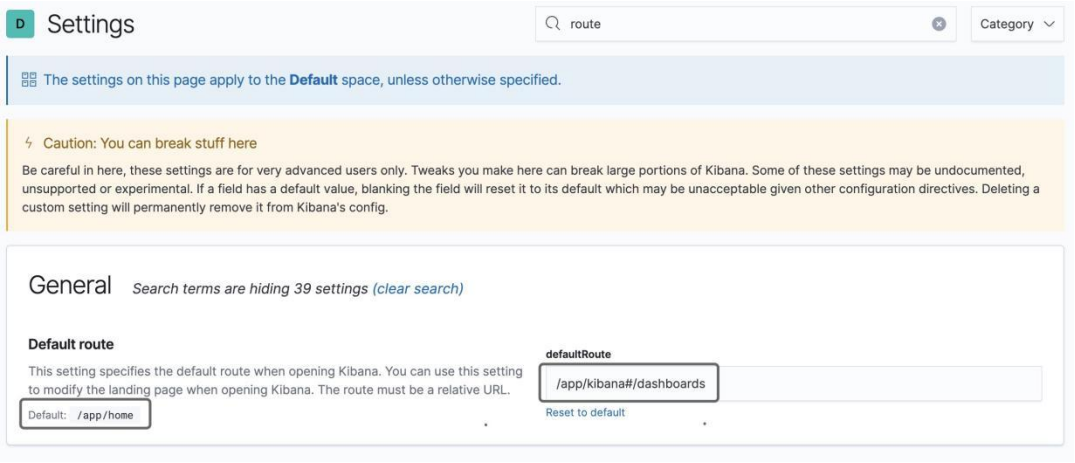


图 12 Space 默认配置修改

创作人简介：

齐乐，从事大数据开发工作近四年，主要方向为 ES，已通过 ECE 认证，近期在研究 Spark 和 Flink。 欢迎一起学习，一起讨论，共同进步。

博客：https://www.jianshu.com/u/cc7ee7454afc

3.5.10 APM

创作人：胡征南

审稿人：杨振涛

应用程序性能管理（Application Performance Management）简称 APM。主要功能为监视和管理软件应用程序性能和可用性。

Elastic APM 是一款基于 Elastic 技术栈的免费及开放的性能监控系统。用于实时监控软件服务和应用程序的各项性能指标，如：请求访问的各项指标、访问耗时、数据库查询、缓存调用、外部 HTTP 请求等。便于开发人员快速排查和修复各种性能问题。

Elastic APM 也会自动采集各种系统异常。开发人员可以通过追踪异常的程序栈，识别系统异常发生的时间和次数。

自动采集的系统指标也是 Elastic APM 一个非常重要的数据源。采集维度主要包括，宿主机级别和服务代理（Agent）级别，如：使用 Java 语言系统编写的应用程序的 JVM 指标，Go 语言运行环境的运行指标等。

Elastic APM 由 4 个组件组成：APM 代理、APM 服务端、Elasticsearch 和 Kibana。

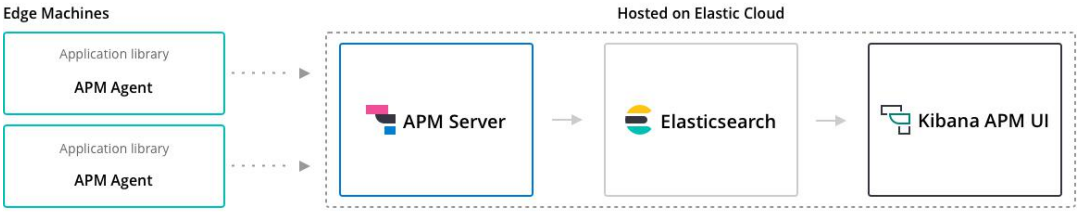


图 1

APM Agent

APM Agent 为通过与应用服务相同的语言编写的开源类库。通过按照其他应用程序服务一样安装代理。安装完成之后，即可通过代理收集各项应用程序指标和运行时异常。收集的数据将在本都缓存较短时间后发送至 APM Server。

支持的语言主要包括：Go、Java、.NET、Node.js、Python、Ruby、JavaScript。

APM Server

APM Server 是一款收集 APM Agent 数据、可独立部署的开源应用程序。APM Server 使得代理变得轻量级、防范安全风险、并提高了 Elastic 技术栈的跨语言能力。APM Server 将通过 APM Agent 收集的数据进行验证、处理后存入对应的 Elasticsearch 索引。经过几秒种后，就可以通过 Kibana APM 应用观测到可视化后的应用程序性能数据了。

Elasticsearch

Elasticsearch 是一款高度可伸缩的全文检索和分析引擎。用于近实时存储、搜索和分析大量数据。在 Elastic APM 中，Elasticsearch 用于存储和聚合性能监控指标。

Kibana

Kibana 是一款将 Elasticsearch 数据进行分析和可视化的免费及开放的数据分析平台。用于对检索、展示、操作 Elasticsearch 中存储的数据。

APM 术语

Service

在 APM agent 配置中进行设置，以将特定的 APM agent 组标识为单个服务，这是一种逻辑上标识一组事务的方法。

Transaction

组成一个服务的请求和响应，例如登录 api 调用，每个调用由单独的 span 组成。

Span

事务中的单个事件，例如方法调用，数据库查询或缓存插入或检索，即需要花费时间才能完成的任何事件。

Erorrs

具有匹配的异常或日志消息的异常组。

Trace

代表请求的整个过程

它们之间的关系可以用如下的图来表示：

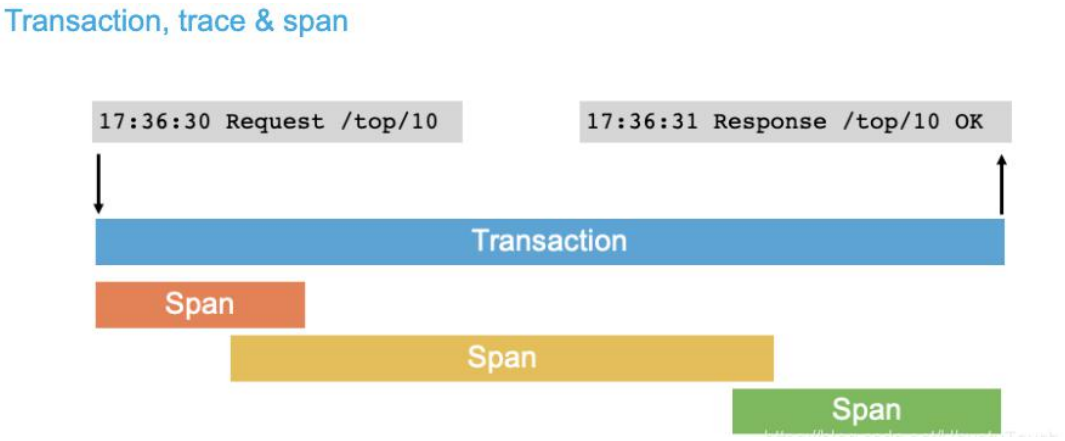


图 2

分布式 Tracing

当请求从一个微服务流向另一个微服务时，追踪器添加逻辑以创建唯一的追踪识别代码， 跨度 id

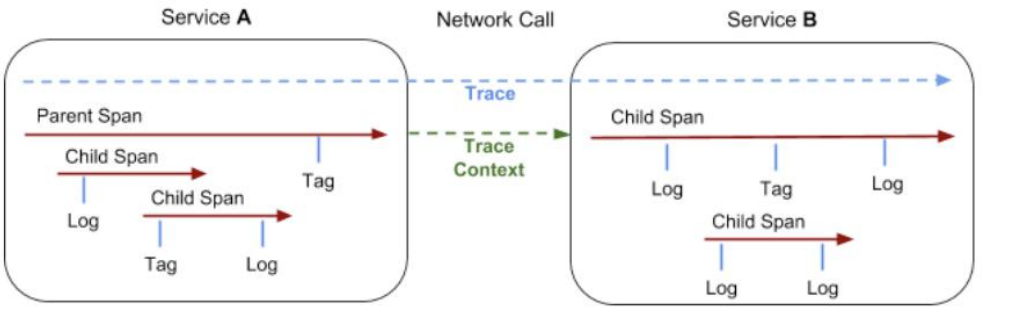

Service A
Network Call
Service B
Trace
Parent Span
Child Span
Tag
Log
Child Span
Tag
Log
Trace
Context
Child Span
Log
Tag
Log
Child Span
Log
Log

图 3

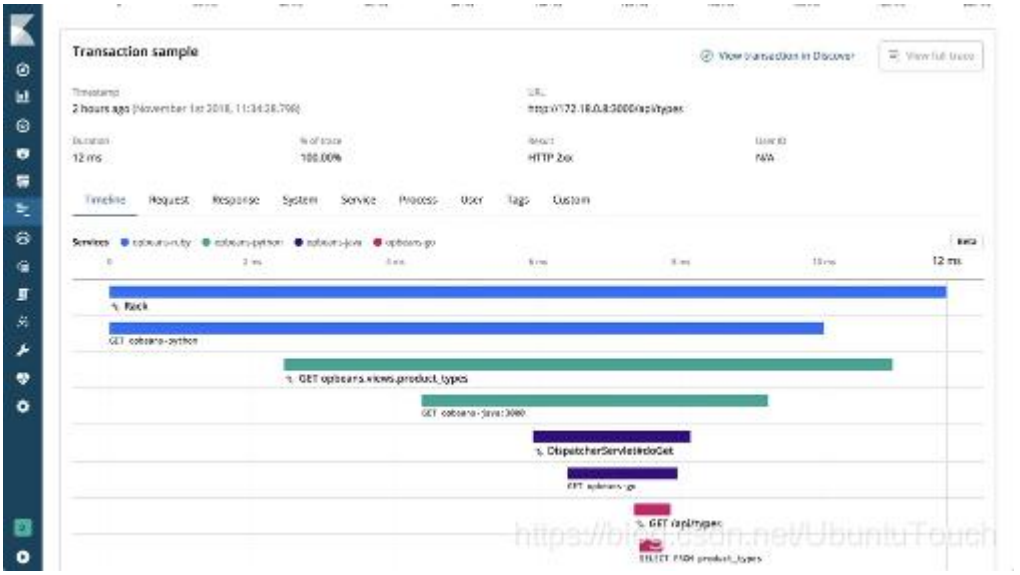

Transaction sample
Timeline
Request
Response
System
Service
Process
User
Tags
Custom
12 ms
Rack
GET opbeans.views.product_types
GET /api/types

图 4

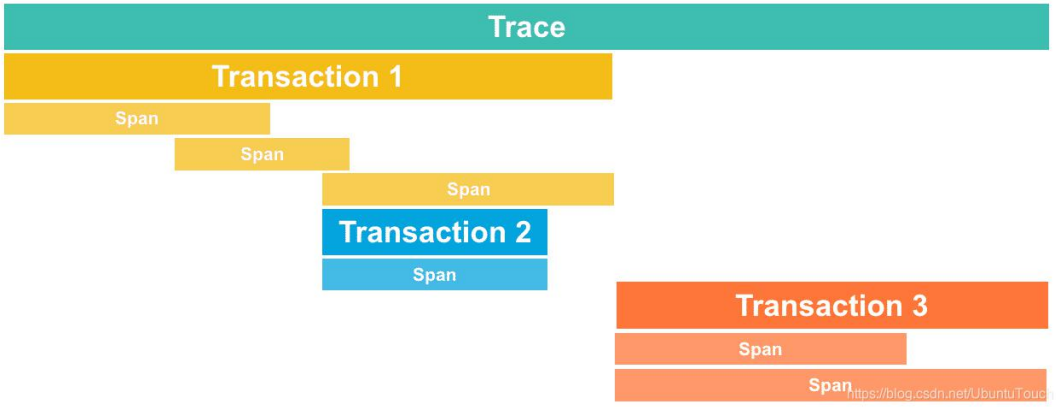


图 5

APM 的安装和使用

启动步骤

配置启动 APM 主要包含三个步骤：安装 APM Server、配置 APM Agent、在 Kibana 中配置可视化面板。

步骤一：安装 APM Server

- 安装

通过 APM 服务端地址下载合适的服务端安装包。

下载地址 :https://www.elastic.co/cn/downloads/apm

- 设置和配置

  通过 apm-server setup [FLAGS] 可以设置 APM 服务端

  FLAGS ：

- -h, --help 查看帮助。
- --index-management 设置关联 Elasticsearch 索引管理，包括：索引模板、生命周期管理策略、写入别名。
- --pipelines 注册定义在 ingest/pipeline/definition.json 中的管道。

  配置示例：

```
apm-server setup --index-management
apm-server setup --pipelines
```

- 启动

  通过启动命令启动 APM 服务端，启动命令：

```
./apm-server -e
```

  指定输出 Elasticsearch 及 APM 服务端：

```
./apm-server -e -E output.elasticsearch.hosts=ElasticsearchAddress:9200 -E apm-server.host=localhost:8200
```

步骤二：配置 APM Agent

下载 APM 代理 可以通过 Maven Central 下载代理 Jar 包，不需要再项目中引入依赖，

下载地址:https://search.maven.org/search?q=a:elastic-apm-agent

- 使用 javaagent 参数启动应用，并设置好对应的配置项。

  `elastic.apm.service_name` 为服务名称

  `elastic.apm.server_urls` 为服务端请求地址

  `elastic.apm.application_packages` 为项目包路径

- 启动参数示例：

```
java -javaagent:/path/to/elastic-apm-agent-<version>.jar \
     -Delastic.apm.service_name=my-application \
     -Delastic.apm.server_urls=http://localhost:8200 \
     -Delastic.apm.secret_token= \
     -Delastic.apm.application_packages=org.example \
     -jar my-application.jar
```

步骤三：在 Kibana 查看可视化面板

- 启动 Kibana
- 在 Kibana 中 可观测性(Observability) 菜单下选择 APM，如图 1

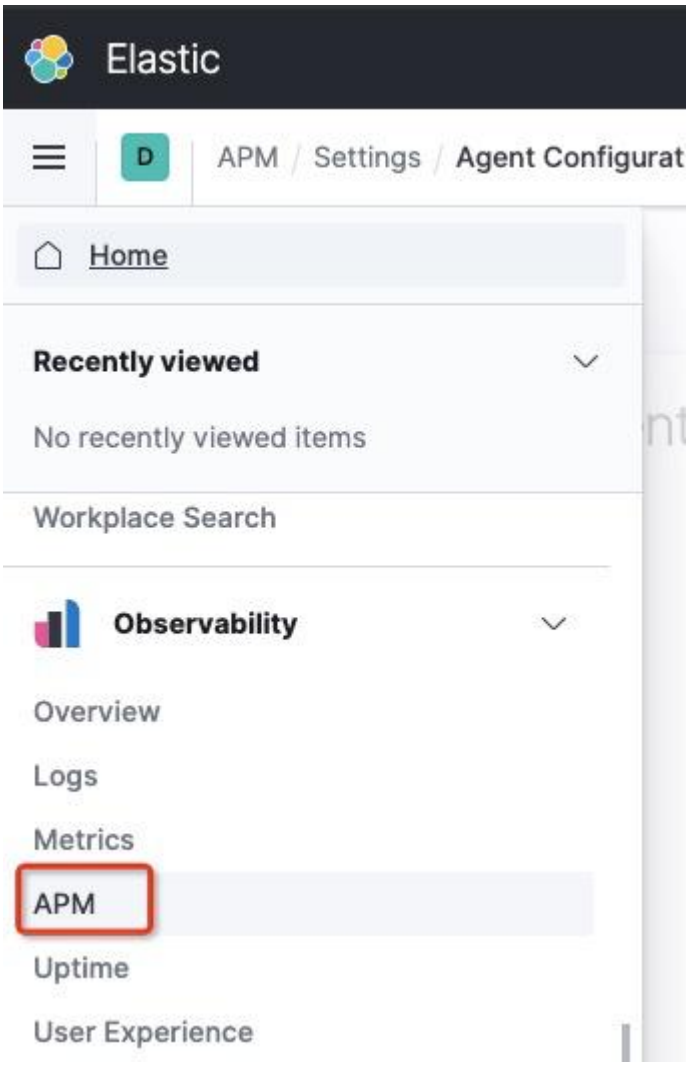


图 6 Observability 菜单中选择 APM

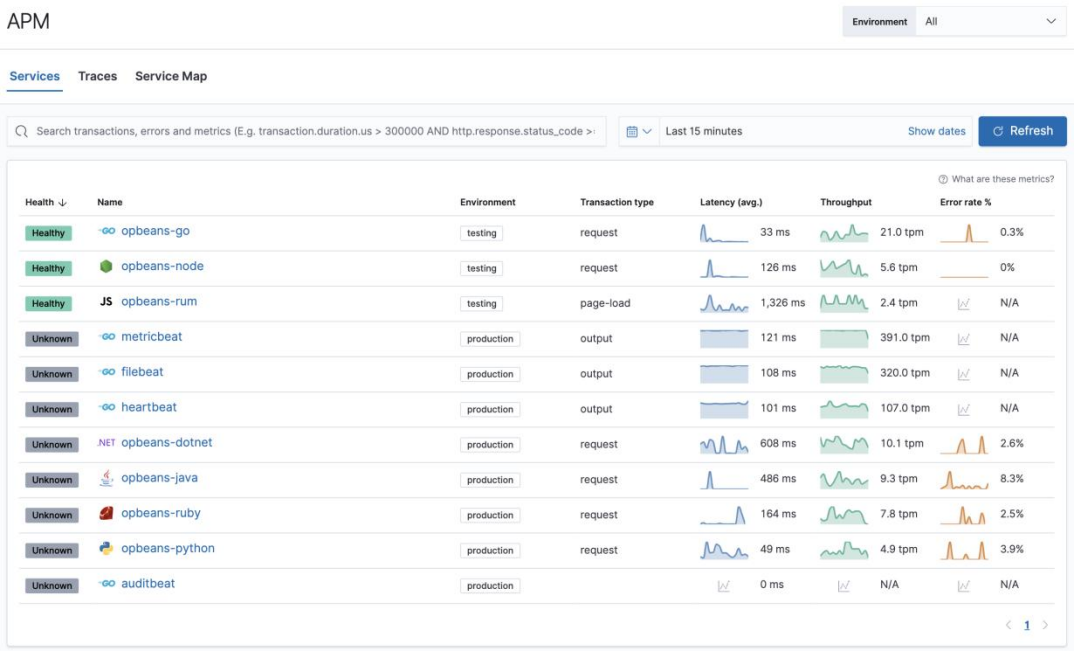


图 7 服务健康状态视图

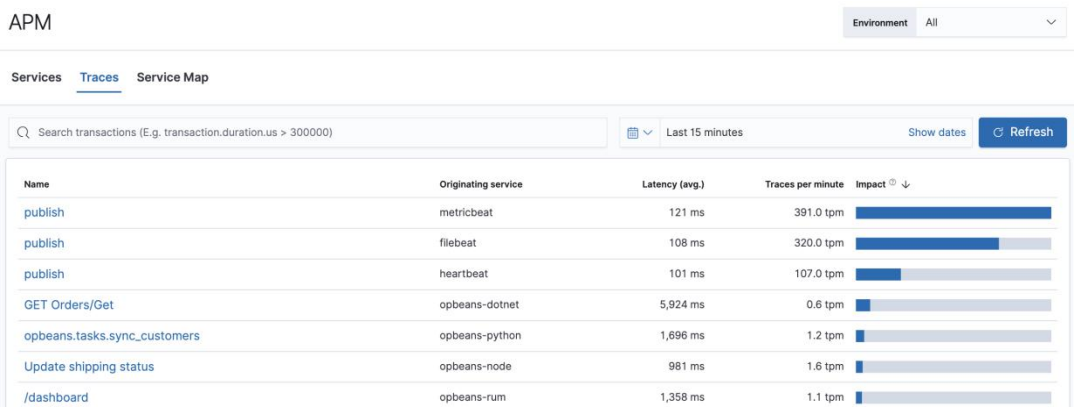


图 8 服务链路追踪视图

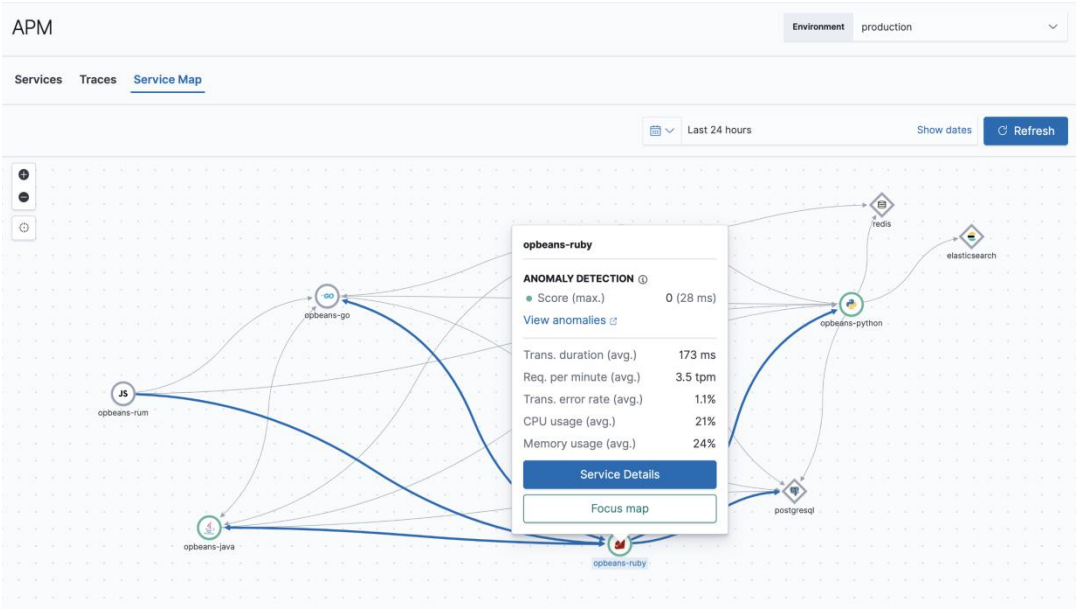


图 9 服务拓扑图视图

Service Map

APM 的 服务拓扑(Service map)功能可以实时显示应用架构的服务拓扑图。可以展示服务是如何连接及服务相关指标，如：事务处理耗时、每分钟请求数量、每分钟错误数量等。APM 服务拓扑图也可以结合机器学习功能，基于异常检测得得分的实时健康状态显示。以上功能可用于帮助开发人员快速、可视化的观测到服务的状态和健康程度。

APM 服务拓扑图有两种显示方式，如下图所示：

- 全局显示：所有安装了 APM 客户端的服务及它们的连接关系。
- 指定服务显示：高亮选中服务的连接关系。

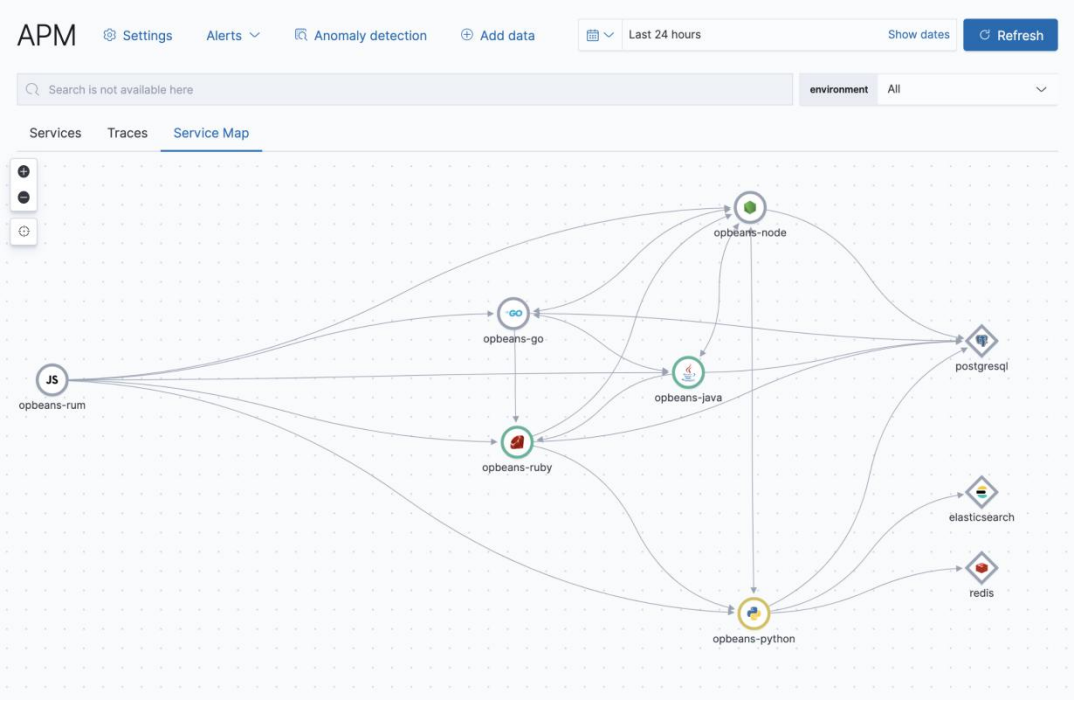


图 10

服务拓扑是通过开箱即用的服务链路追踪实现的。即：未正确配置追踪服务的项目不会被追踪到，也不会在服务拓扑图中显示。

服务拓扑图中选择关注的服务，可以查看对应服务的详细信息。

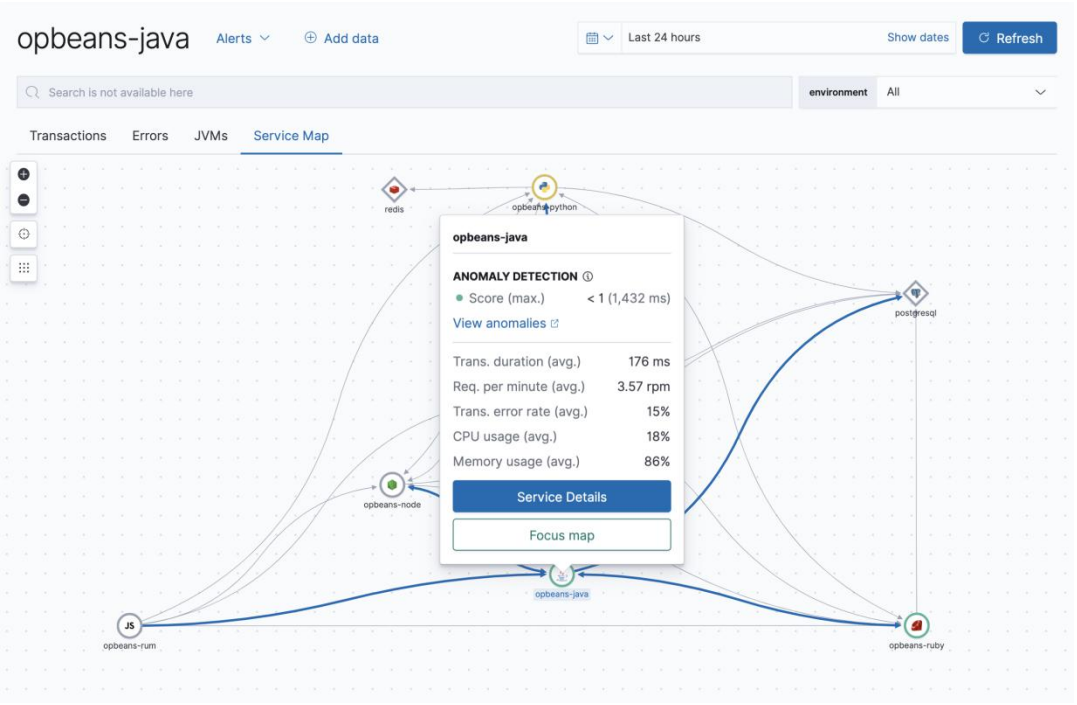


图 11

通过机器学习自动检测事务调用得分，并在服务拓扑图中通过图例显示。

	异常得分小于 25，服务处于健康状态
	最大异常得分介于 26-74，服务可能处于亚健康状态
	最大异常得分大于 75，服务出于不健康状态

表 1

服务拓扑图中有两种形状的节点。

圆形：检测到的服务，图标取决于使用了那种代理。

钻石形：数据库，外部系统和消息。根据已知的实体显示对于的图标，如 Elasticsearch 的图标。

参考文献：

- https://elasticstack.blog.csdn.net/article/details/102844900

创作人简介：

胡征南，杭州光云科技 Java 开发专家，获 ECE 认证、CNCF K8S 认证。

3.5.11 Uptime

创作人：程序员历小冰

审稿人：朱荣鑫

现在互联网架构随着用户的增加，而越来越复杂，可能要有成千上万个不同的组件和不同的实例，对这些组件可用性的监控是提供高可用服务的关键之一，Elastic 为此推出了 Uptime App。

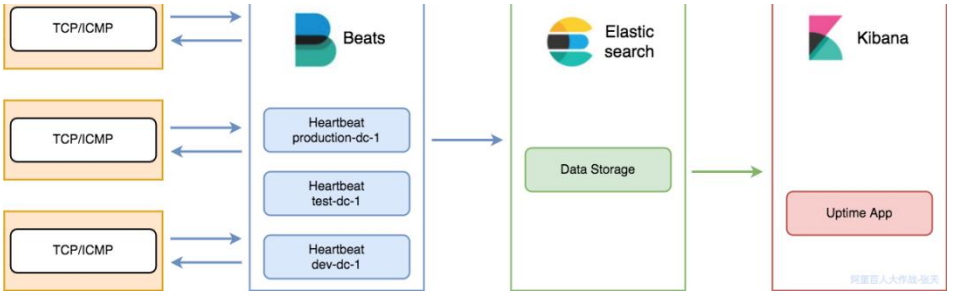
Elasticsearch 使用 Heartbeat 进行组件的监控。

Heartbeat 也就是我们通常所说的心跳，通过 Hearteat 我们可以判断一个网络组件，当前是否存活，是否可以对外正常提供服务。

Heartbeat 是一个轻量级的数据收集器。它用来帮我们进行 Uptime 的健康监控。它可以定期通过 HTTP、TCP 或 ICMP 等方式验证组件是否处于运行状态，然后将收集到的状态和信息上报给 Elasticsearch。

而 Kibana 中的 Uptime app 则为我们提供了查看可用性数据的仪表板，以监控服务器或服务的正常运行，并提供了报警功能支持。

Elasticsearch 使用 Heartbeat 来进行 Uptime 的监控的架构可以表述如下：

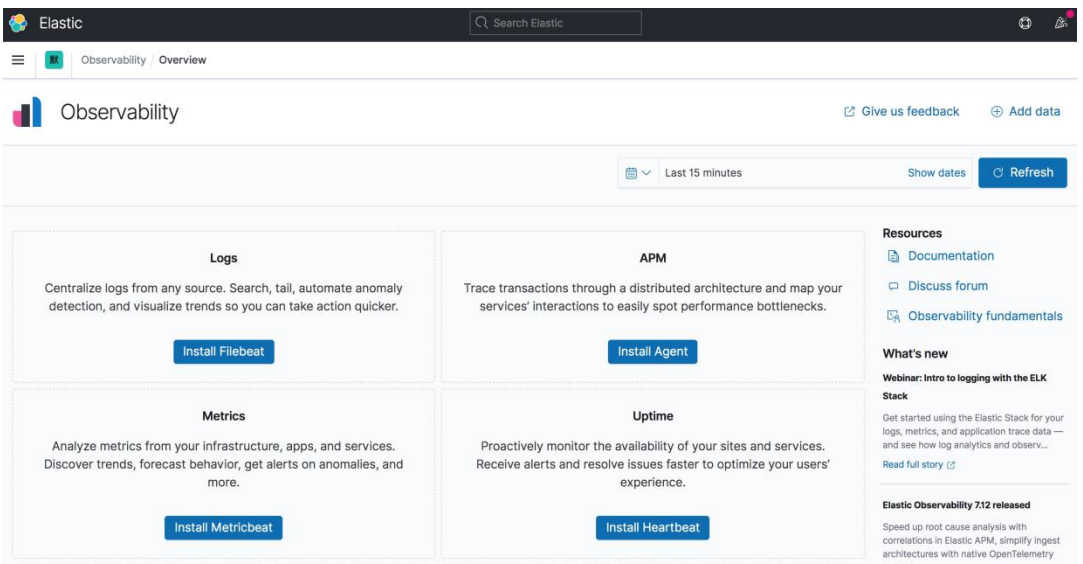


Uptime 监控示意图

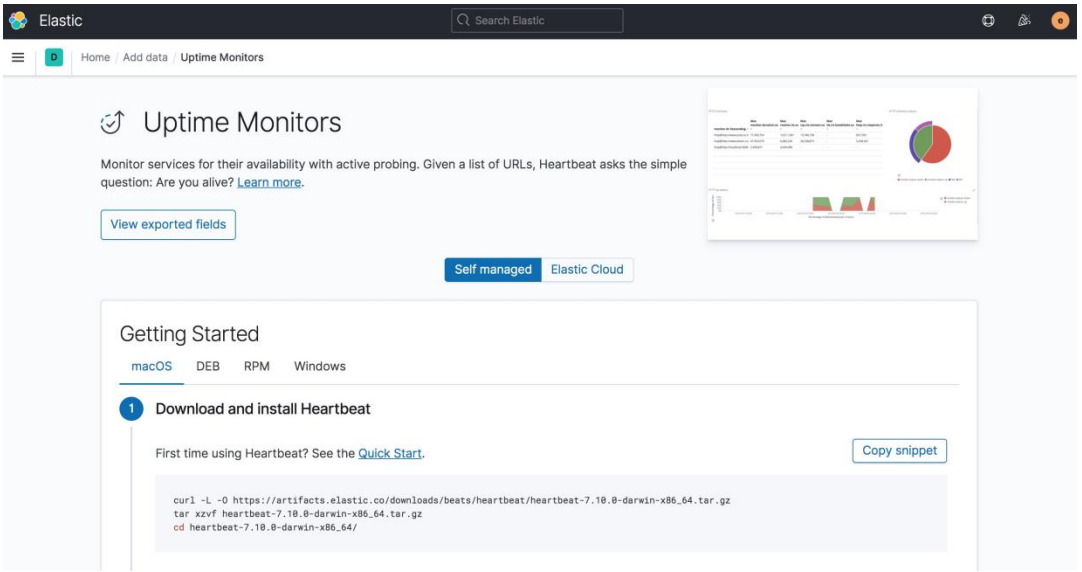
下面，我们将依次讲解 Uptime App 的安装，Heartbeat 的配置和各类监控组件的配置。

安装 Uptime App

如果我们打开我们的 Kibana 并点击 Uptime 应用，那么第一次打开的时候，我们可以看到，如下的界面。



点击 Install Heartbeat，就会跳转到配置 Uptime Monitors 的文档界面，你可以按照这个界面上的步骤进行 Heartbeat 的安装，配置，启动和测试 Kibana 是否接收到 Heartbeat 上传的数据。



Heartbeat 在不同平台有多种安装方式，比如说 macOS、DEB、RPM 和 Windows 等，我们这里介绍最为常用的 Docker 安装方式，其后续部署和启动步骤则大同小异，读者可以自行根据需要进行实践。

需要注意的是，安装的 Heartbeat 必须和 Elasticsearch 或 Kibana 版本相同，所以我们这里选取 heartbeat:7.10.0 版本的镜像。

```
docker pull docker.elastic.co/beats/heartbeat:7.10.0
```

接着，我们可以使用如下命令启动 Heartbeat 容器。

```
docker run -d --name=heartbeat --user=heartbeat
--volume="/tmp/heartbeat.docker.yml:/usr/share/heartbeat/heartbeat.yml:ro"
docker.elastic.co/beats/heartbeat:7.10.0 --strict.perms=false
```

这里使用了 docker 的 --volume 参数，挂载了宿主机文件系统路径下的 heartbeat.docker.yml 文件到容器的对应路径下，这是在为 Heartbeat 提供配置文件。具体配置文件内容后续继续讲解，我们这里先演示完整个 Uptime 安装流程。

启动 Heartbeat 容器后，通过 docker ps 和 docker exec 命令可以进入到相应的容器内部。

```
docker ps
docker exec -it 5b3785357c26(要替换为自己 ps 命令输出的 CONTAINER ID) bash
```

然后，通过 ls 命令，我们可以看到 Heartbeat 的整体文件结构。

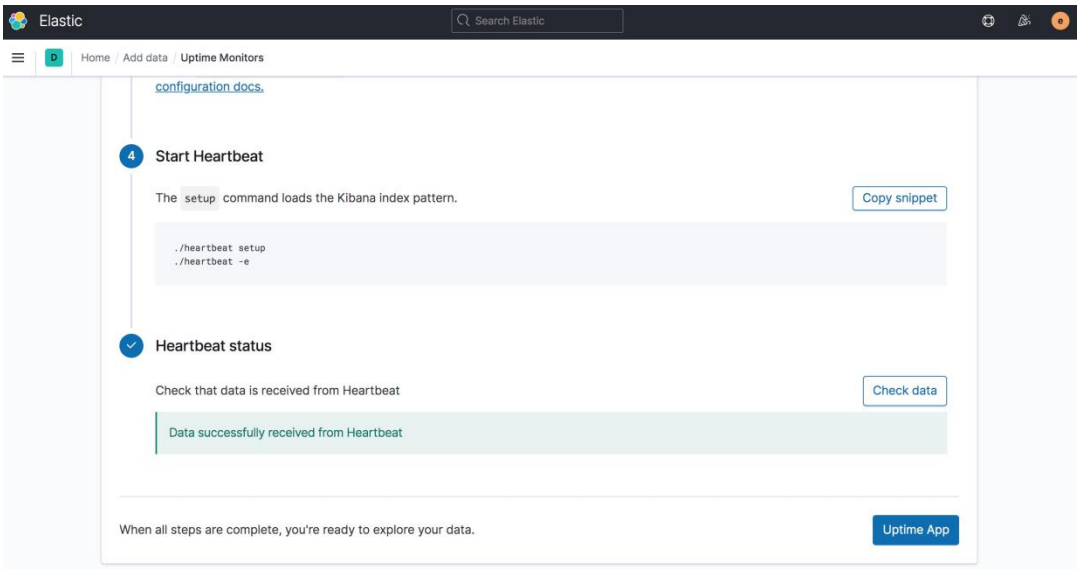
```
bash-4.2$ ls
LICENSE.txt NOTICE.txt README.md data fields.yml heartbeat
heartbeat.reference.yml heartbeat.yml kibana logs monitors.d
```

在目录中，有一个叫做 heartbeat.yml 的配置文件，这个文件就是上边通过 --volume 参数挂载进来的。同时在 monitor.d 目录中，有一些不同监控器配置的配置文件案例可供大家参考，heartbeat.reference.yaml 中则是最全的配置案例。

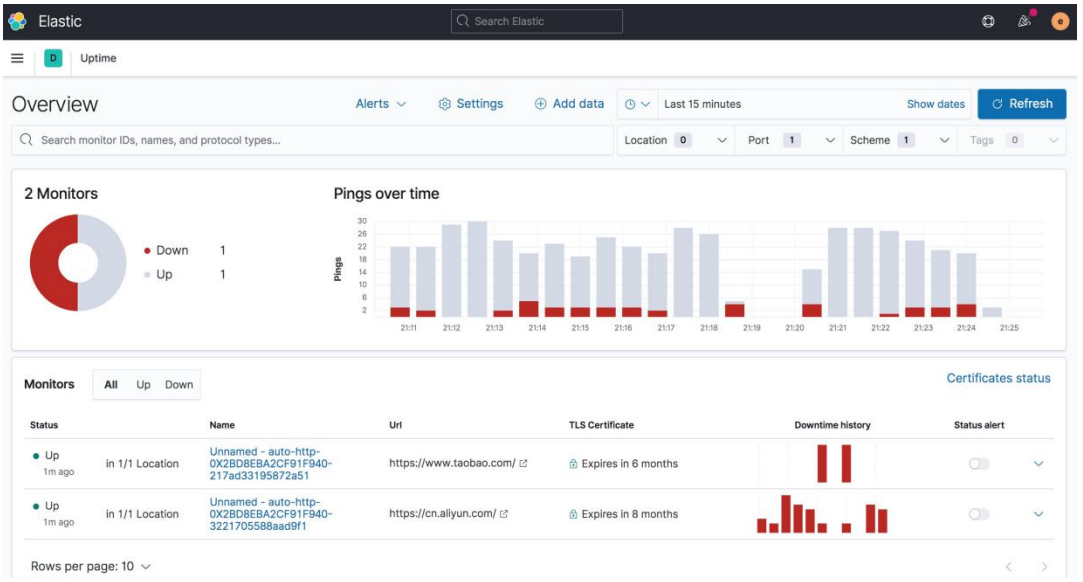
接着，我们要使用如下命令来启动 Heartbeat，让它开始收集数据并向配置文件中指定的 Elasticsearch 中上报数据。

```
./heartbeat setup
./heartbeat -e
```

查看上述命令的输出日志没有什么异常后，可以再次来到 Uptime Monitors 界面，点击其 Check data 按钮检查是否接收到了数据，如果接受到了数据，则可以点击 Uptime App 按钮，前往 Uptime App 界面查看详细数据。



运行过一段时间的 Uptime App 界面如下图所示。



我们可以看到界面分为两大部分，上半部分是统计区，通过饼图和柱状图展示了当前监控器 Monitor 的状态和过去一段时间中 Monitor 的状态。而下半部分是具体的 Monitor 列表，一共有两个 Monitors，分别是监听 taobao 网和 aliyun 网站，目前两个都是 Up 状态。

配置 Heartbeat

上边讲解了安装 Heartbeat 和 Uptime 的整体流程，本小节详细解决一下 Heartbeat 的配置，也就是 heartbeat.yml 文件的配置。

heartbeat.yml 文件一般有两部分组成：

- 监控器配置 heartbeat.monitors，配置要监控的目标和监控的方式；

- 输出配置 output.elasticsearch，配置数据上报的 Elasticsearch 的地址，用户名和密码。

比如说，上一小节我们启动 docker 时指定的 heartbeat.yaml 文件如下所示：

```
heartbeat.monitors:- type: http # 使用 http 方式监控，还可以使用 TCP 和 ICMP
  schedule: '@every 5s' # 每 5s 抓取一次
  urls: # 需要监控的 url 地址
    - https://cn.aliyun.com/
    - https://www.taobao.com/
output.elasticsearch:
  hosts: '${ELASTICSEARCH_HOSTS:http://es-cn-n6w24fib900797tgz.public.elasticsearch.aliyuncs.com:9200}'
  username: '${ELASTICSEARCH_USERNAME:111}'
  password: '${ELASTICSEARCH_PASSWORD:111}'
```

为了使 Heartbeat 知道要检查的服务，它需要一个 URL 列表。

heartbeat.yaml 中的 heartbeat.monitors 中指定了此配置。 如上的 heartbeat.yaml 配置文件，对 cn.aliyun.com 和 www.taobao.com 两个网址每隔 5s 进行一次 HTTP 检查。

除了 HTTP 监视器，Heartbeat 还可以进行 TCP 和 ICMP 类型的检查。

```
heartbeat.monitors:- type: icmp
  schedule: '@every 5s'
```

```
    hosts:
      - http://cn.aliyun.com/
      - http://www.taobao.com/- type: tcp
    schedule: '@every 5s'
    hosts:
      - 127.0.0.1:8080
```

此外，它还支持定义不同的检查语句，例如，使用 HTTP 监视器，可以检查响应代码(code)、正文（body）和标头（header）。 使用 TCP 监视器，能定义端口检查和字符串检查。

```
heartbeat.monitors:- type: http
  schedule: '@every 5s'
  urls:
    - https://cn.aliyun.com/
  # request details:
  check.request:
        method: GET
  check.response:
        body: "aliyun"
```

如上的配置， Heartbeat 会每 5s 使用 GET 调用一次 https://cn.aliyun.com/ ，并在其 Response 的 Body 中寻找字符串 aliyun。如果没有找到这个字符串，则本次检查未通过。

其他更加详细的配置，你可以参考 heartbeat.reference.yml 文件。

创作人简介：

程序员历小冰，专注于探讨后端生态的点点滴滴，包括微服务、分布式、数据库、性能调优和各类中间件源码分析。

博客：http://remcarpediem.net/

3.5.12 Monitoring 及 Central Management

创作人：高冬冬

审稿人：刘帅

Monitoring

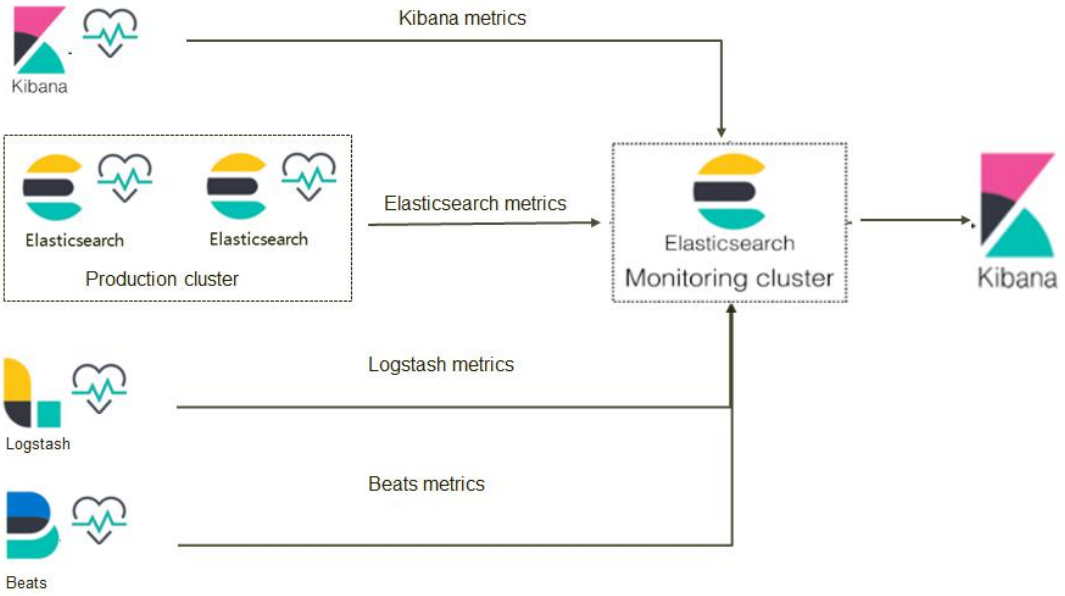
Monitoring 就是跟踪和监控 Elastic Stack 各个组件的实时运行状况和性能指标；当监控一个集群时，不仅要采集 Elasticsearch 节点的指标，而且要采集集群相关的 Logstash 节点，Kibana 实例以及各种 Beats 节点的性能指标甚至还要通过 Filebeat 采集集群日志，存储在 Elasticsearch 集群中，以便可以通过 Kibana 可视化，实时监控各种组件和节点的实时运行状态。

两种监控方案：

- 组件自身监控
- Metricbeat 监控

组件自身监控

开启快捷简单，无需额外组件，收集采集指标会占用组件自身资源；



默认情况下，每一个 Elastic Stack 组件自身都包含一个内置的 agent 负责采集数据

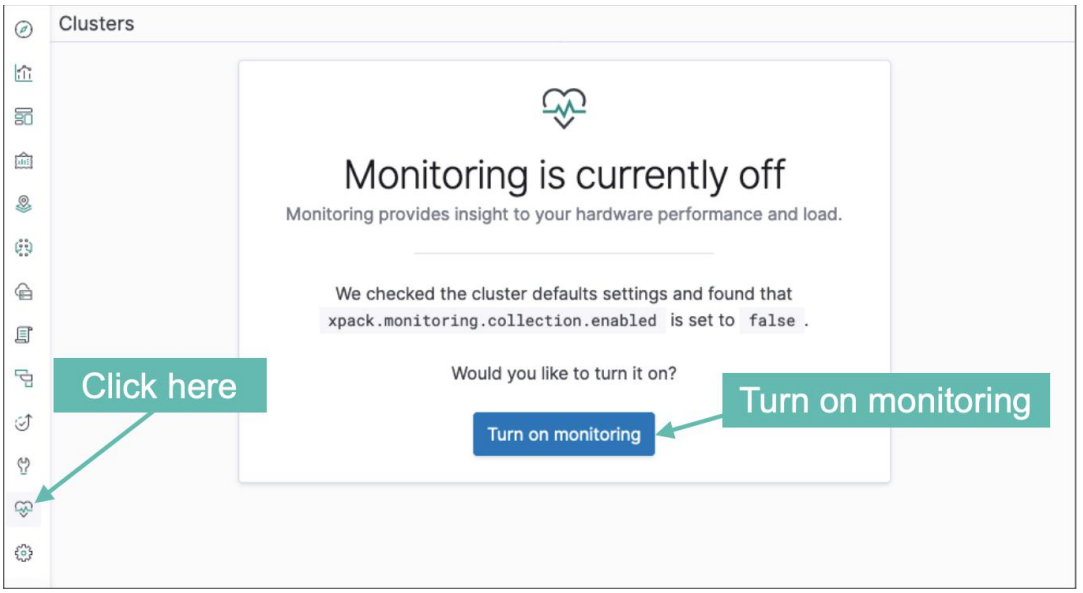
配置方式

Elasticsearch

在 Elasticsearch 集群中监控采集配置默认关闭的 xpack.monitoring.collection.enabled : false

通过 Kibana 开启

- 打开 Kibana
- 进入 Management-->Stack Monitoring
- 点击 Turn on monitoring



通过 API 开启

```
GET _cluster/settings
PUT _cluster/settings
{
```

```
    "persistent": {
      "xpack.monitoring.collection.enabled": true
    }
}
```

Elasticsearch 的其他配置

在节点的配置文件 elasticsearch.yml 更多配置

xpack.monitoring.collection.indices	指定要监控的索引名，默认是监控所有的，指定	The indices to collect data from. Defaults to all indices, but can be a comma-separated list.
xpack.monitoring.collection.interval	数据采集频率，默认是10s	How often data samples are collected. Defaults to 10s
xpack.monitoring.exporters	指标数据的存储位置，默认存储在自身集群，可以通过此配置指定远端集群。	Configures where the agent stores monitoringdata. By default, the agent uses a localexporter that indexes monitoring data on thecluster where it is installed.

参考文献：https://www.elastic.co/guide/en/elasticsearch/reference/7.10/monitoring-settings.html

Kibana

在配置文件 kibana.yml 开启：

```
#是否开启 Kibana NodeJS server 指标采集
monitoring.kibana.collection.enabled: true
#采集频率(ms),默认 10s
monitoring.kibana.collection.interval: 10000
#指定监控指标存储远程 ES 集群
monitoring.ui.elasticsearch.hosts: ["https://es1:9200", "https://es2:9200"]
#远程 ES 集群的账号和密码
monitoring.ui.elasticsearch.username: elasticsearch
monitoring.ui.elasticsearch.password: changeme
#控制 monitoring 后端的运行和 kibana 运行状态的监控
monitoring.enabled: true
#在 kibana 中隐藏 Stack Monitoring 功能。
monitoring.ui.enabled: true
```

参考文档：https://www.elastic.co/guide/en/kibana/7.10/monitoring-settings-kb.html#monitoring-general-settings

Logstash

在配置文件 logstash.yml 开启：

```
# X-Pack Monitoring
# https://www.elastic.co/guide/en/logstash/current/monitoring-logstash.html
xpack.monitoring.enabled: true
xpack.monitoring.elasticsearch.hosts: ["https://es1:9200", "https://es2:9200"]
```

```
xpack.monitoring.elasticsearch.username: elasticsearch
xpack.monitoring.elasticsearch.password: password
```

Beats：Filebeat、Metricbeat

在配置文件 filebeat.yml 或 metricbeat.yml 中开启：

```
monitoring.enabled: true
#monitoring.cluster_uuid:
monitoring.elasticsearch.hosts: ["https://es1:9200"]
monitoring.elasticsearch.username: filebeat_system
monitoring.elasticsearch.password: password
```

APM

在配置文件 apm-server.yml 中开启：

```
monitoring.enabled: true
monitoring.elasticsearch.hosts: ["https://es1:9200"]
monitoring.elasticsearch.username: filebeat_system
monitoring.elasticsearch.password: password
```

从某种程度上讲 AMP Server 其实就是另外一种 Beat。对于它的监控和 Beats 完全是一样的。

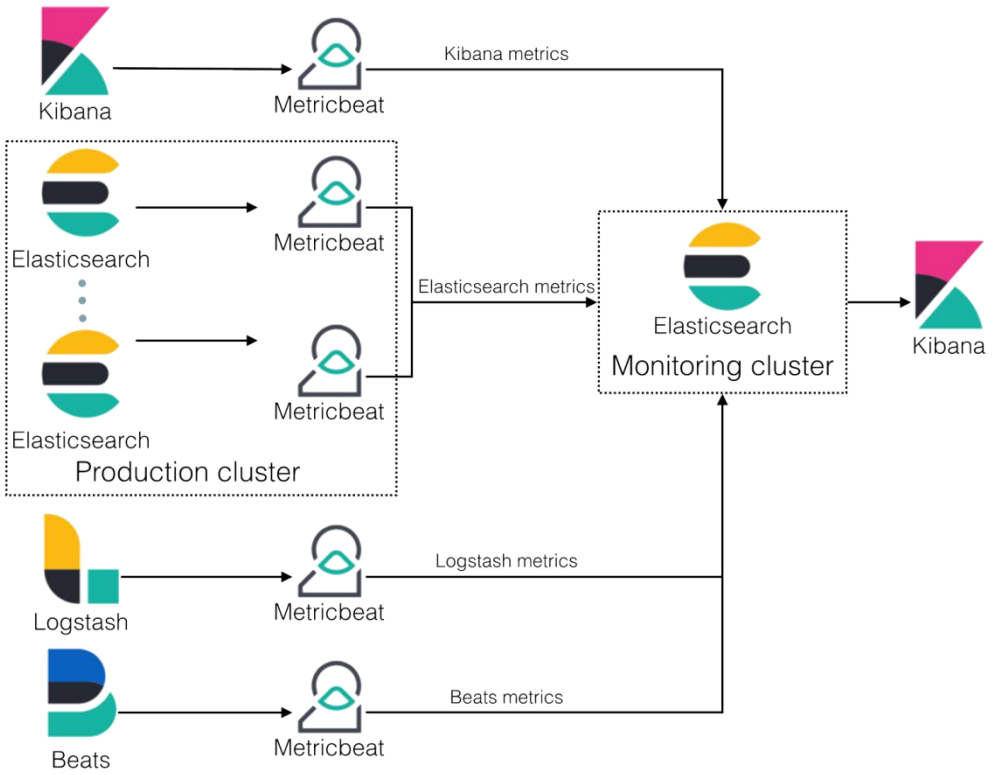
Metricbeat 监控

使用 Metricbeat 采集 Elastic Stack 监控指标，需单独部署监控 Metricbeat 及单独的监控集群。开启对应的 metricbeat modules，避免由于采集数据对组件自身带来压力，而影响组件运行性能。

监控方案

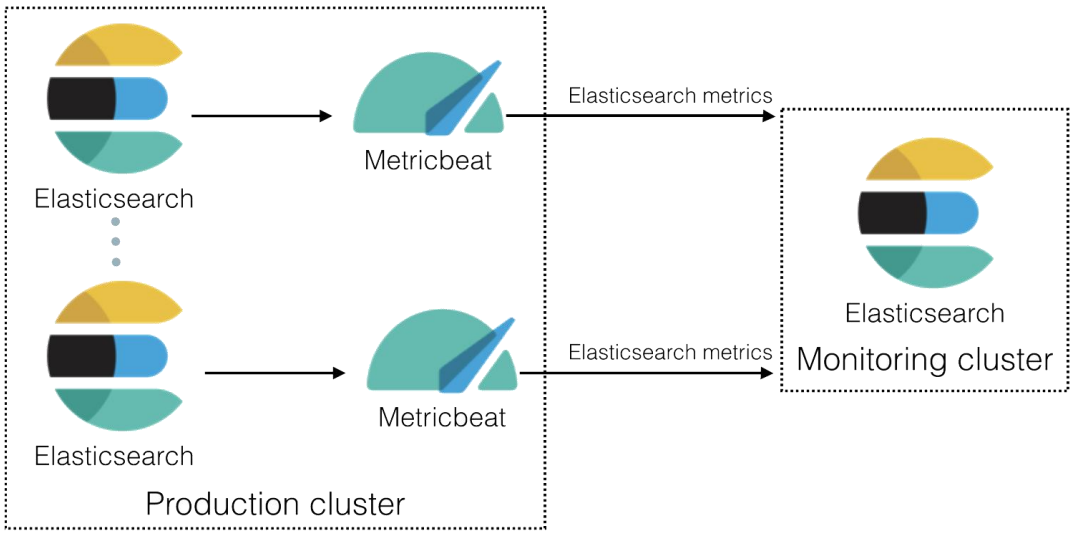
可用来监控 Elastic Stack 的所有类型组件：

- 未来版本中默认的监控方案
- 采集性能比内置采集更好



配置方式

注意在 6.5 版本以及以后，才可以通过 Metricbeat 采集 Elasticsearch 监控指标，并可指定专用的监控集群。



开启监控数据采集

在生产集群中 xpack.monitoring.collection.enabled 默认为 false，可以通过以下 API 进行开启和关闭。针对 Metricbeat 监控，这个设置应为 false。

```
GET _cluster/settings

PUT _cluster/settings
{
  "persistent": {
```

```
    "xpack.monitoring.collection.enabled": false
  }
  "transient" : { }
}
```

在生产集群的每个 node 上安装 Metricbeat，保证每个 node 都安装

在每个 Elasticsearch node 的 Metricbeat 上开启 Elasticsearch X-Pack module

```
metricbeat modules enable elasticsearch-xpack
```

在每个 Elasticsearch node 上配置 Elasticsearch X-Pack module

```
- module: elasticsearch
    xpack.enabled: true
    period: 10s
    hosts: ["http://localhost:9200"]
```

指定监控数据存储的集群

在 Metricbeat 的配置文件（metricbeat.yml）中配置 Elasticsearch output 信息。

```
output.elasticsearch:
  hosts: ["http://es-mon-1:9200", "http://es-mon-2:9200"]
  #protocol: "https"
  #username: "elastic"
  #password: "changeme"
```

在每个 Elasticsearch node 节点启动 Metricbeat

```
nohup ./metricbeat -c metricbeat.yml >/dev/null 2>&1 &
```

关闭默认的 Elasticsearch 监控数据采集

在生产集群中配置 xpack.monitoring.elasticsearch.collection.enabled 为 false

通过以下 API 进行配置：

```
PUT _cluster/settings
{
  "persistent": {
    "xpack.monitoring.elasticsearch.collection.enabled": false
  }
}
```

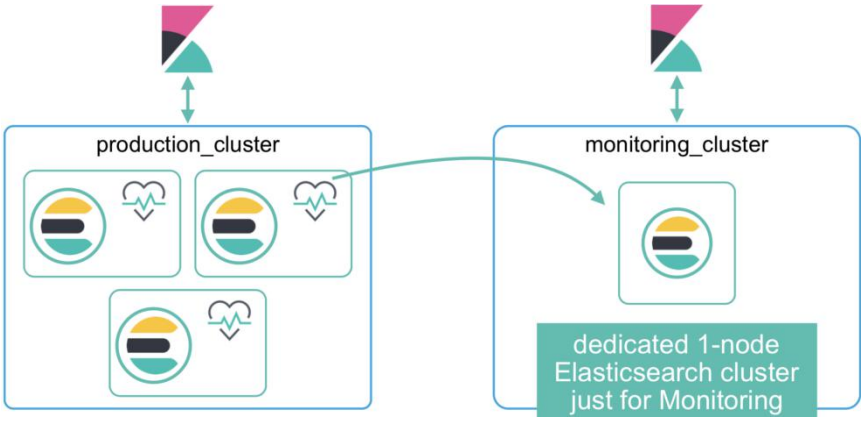
在监控集群的 Kibana 中查看监控页面

参考文档：https://www.elastic.co/guide/en/elasticsearch/reference/7.10/monitoring-overview.html

专用的监控集群

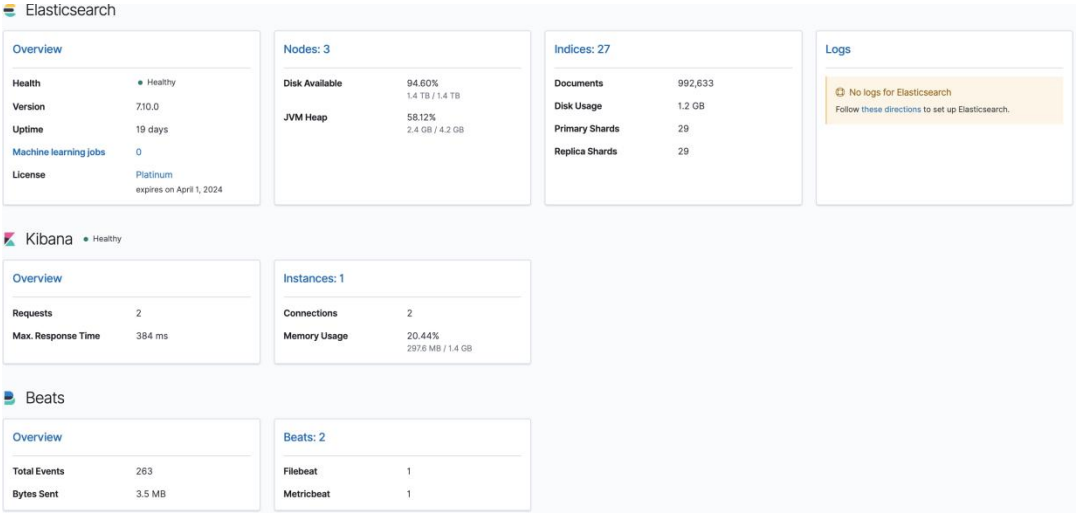
在生产环境推荐部署专用的监控集群来实现集群的指责分离：

- 减少被监控的业务集群的负载和存储压力。
- 防止被监控集群的故障影响监控功能。
- 实现职责隔离，比如监控集群和业务集群可配置不同的安全策略，保障级别等。



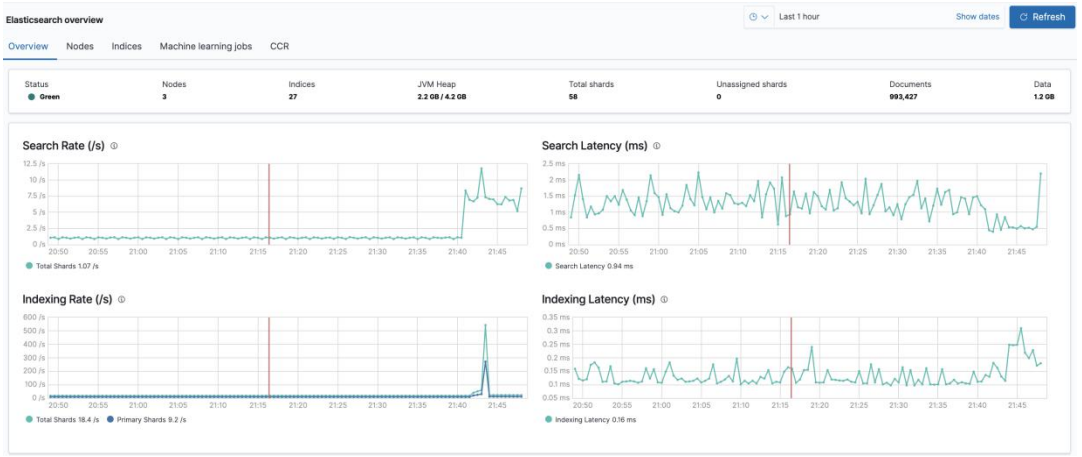
统一监控 UI

Monitoring 概览



概览页面展示了所有正在被监控的 Elastic Stack 组件，包括但不限于 Elasticseach，Kibana，Beats，Logstash，APM 等等。

Elasticsearch 监控



概览页面主要展示了四块内容：

- 集群状态

包括集群状态，节点总数，索引总数，JVM 使用状态，分片总数，未分配的分片数，索引文档总数，存储的数据总大小。

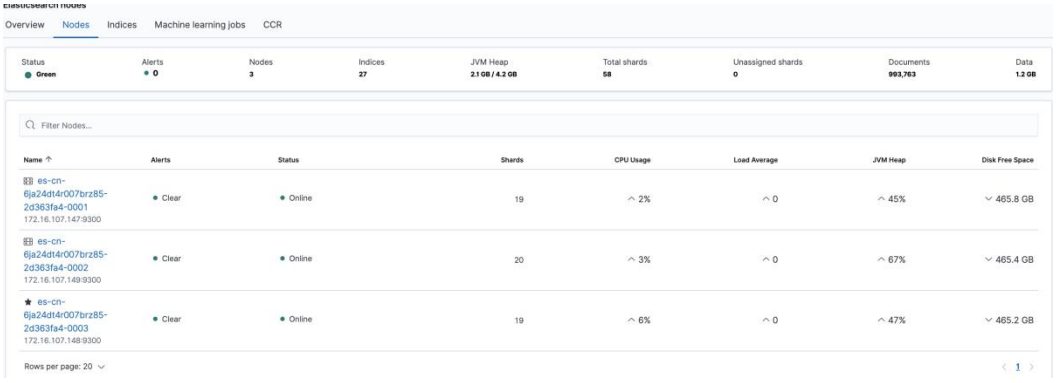
- 集群吞吐量

查询 TPS，查询耗时，索引 TPS 以及索引耗时等时序图。

- 集群日志列表。
- 活动分片迁移列表。

采集的 Elasticsearch 的监控数据包括：Cluster Stats，Index Stats，Index Recovery，Shards,Jobs，Node Stats 等等。

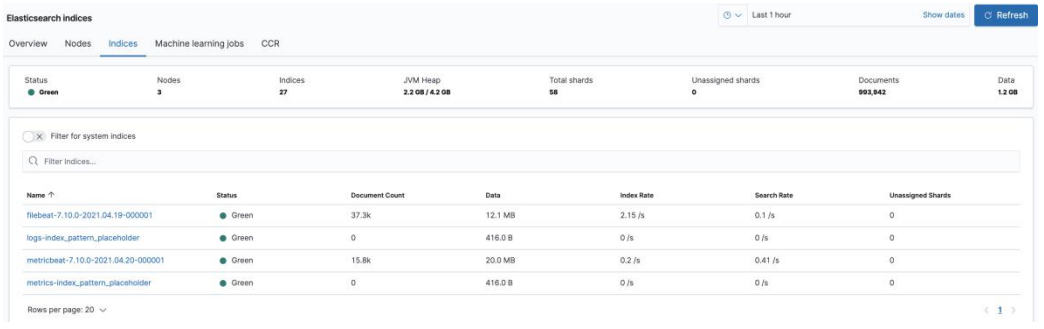
节点监控



主要包含节点列表，同时可以实时查看节点的告警数，在线状态，分片数， 使用率，节点负载，JVM 使用情况以及当前节点空余的存储空间。

点击节点名可以下钻到更详细全面的节点监控以及时序图。

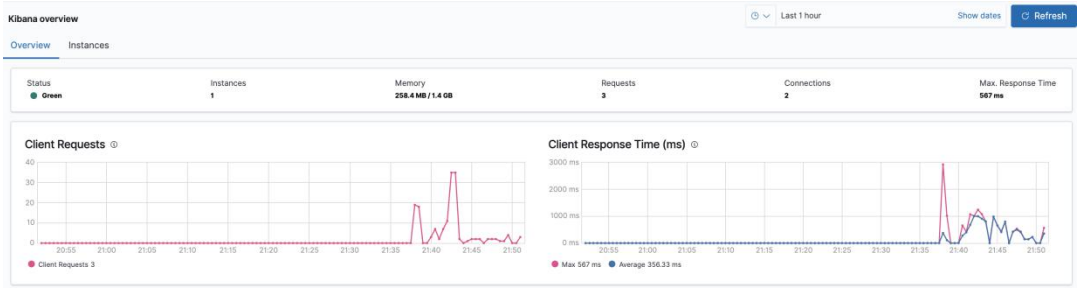
索引监控



主要包含索引列表，可以实时查看到当前索引的状态，文档数，索引大小，索引 TPS，搜索 TPS 以及未分配的分片数。

点击索引名可下钻到更详细全面的索引监控以及时序图。

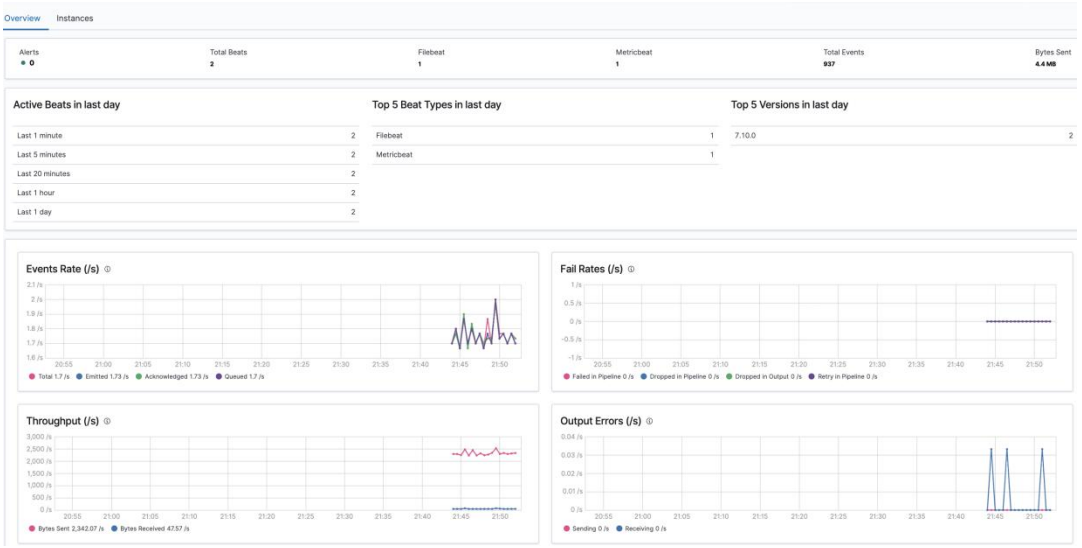
Kibana 监控



在此页面中可以监控到访问 Kibana 的 client request 次数和 client 请求平均耗时的时序图

点击 Instances 还可以监控到所有 Kibana 实例列表以及当前运行状况。

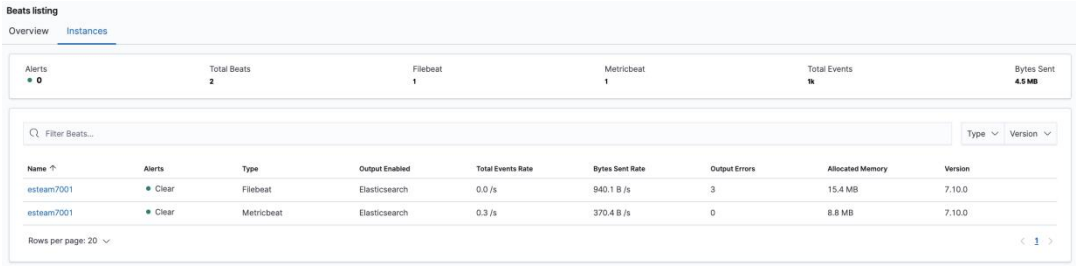
Beats 监控



统计 Beats 的告警数，Beats 总数，Metricbeat 数，总共生产的事件数和发送的事件字节数，最近一天活跃的 Beats，Beats 类型的 Top5 以及 Beats 版本的 Top5。

成功事件数/秒，失败事件数/秒，平均的吞吐量 bytes/second，以及输出失败数/秒。

Beats 实例监控



在实例下可以看到该集群监控到所有 Beats 列表和其运行状态。

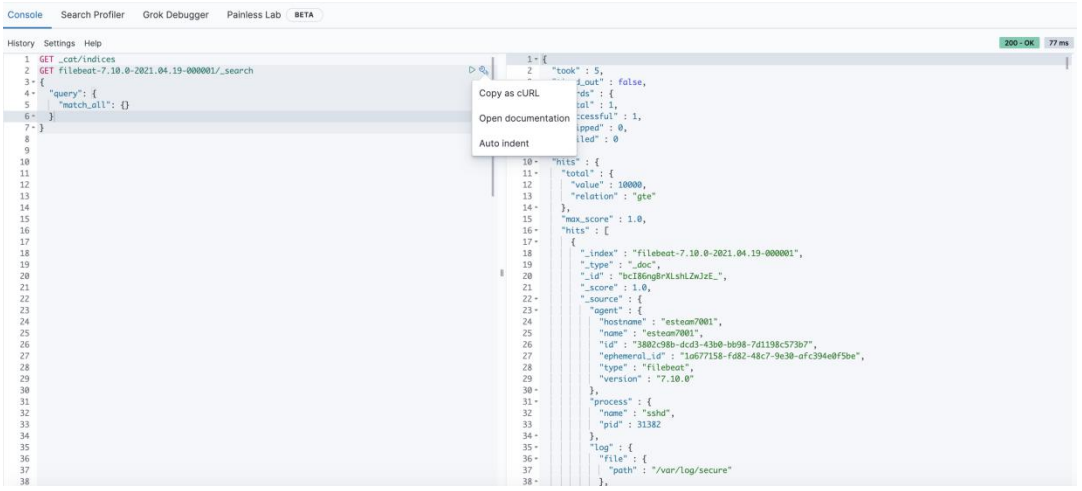
点击实例名可下钻到更详细全面的 Beats 实例监控以及时序图。

Central Management

开发工具，包括一些可以方便用户和集群中数据进行交互探索分析的工具

Console 终端

使用 Elasticsearch 的 REST API 进行交互，包含发送请求和查看 API 文档



如上图所示，在 console 里面执行类似 CRUL 的命令，点击语句后面的➡执行，即可在右侧栏中实时查看结果；

支持 GET，PUT，POST，DELETE 四种命令；

- 支持包含写入，搜索，集群，节点和索引状态等等 Elasticsearch 相关的所有 API；
- 输入命令行时，支持自动补全；
- 支持自动格式化（Auto indent）；
- 支持查看 API 文档（Open documention）；
- 支持查看执行命令历史记录（点击 History）；
- 设置 console 配置。

关闭 console：

在 Kibana 的配置文件 kibana.yml 配置如下：

```
console.enabled: false
```

然后重启 Kibana 即可。

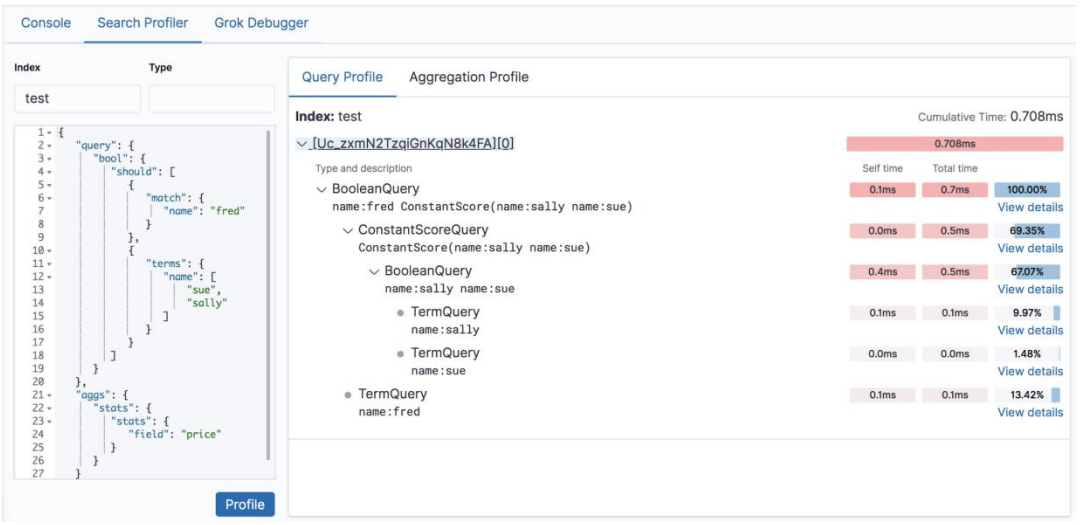
查询分析器

检查并分析你的搜索查询。

Elasticsearch 具有功能强大的 Profile API，可用于检查和分析你的搜索查询。响应返回一个较大的 JSON Blob，可能很难手动对其进行分析。

Search Profiler 工具可以将此 JSON 输出转换为易于浏览的可视化文件，从而使你可以更快地诊断和调试效果不佳的查询。

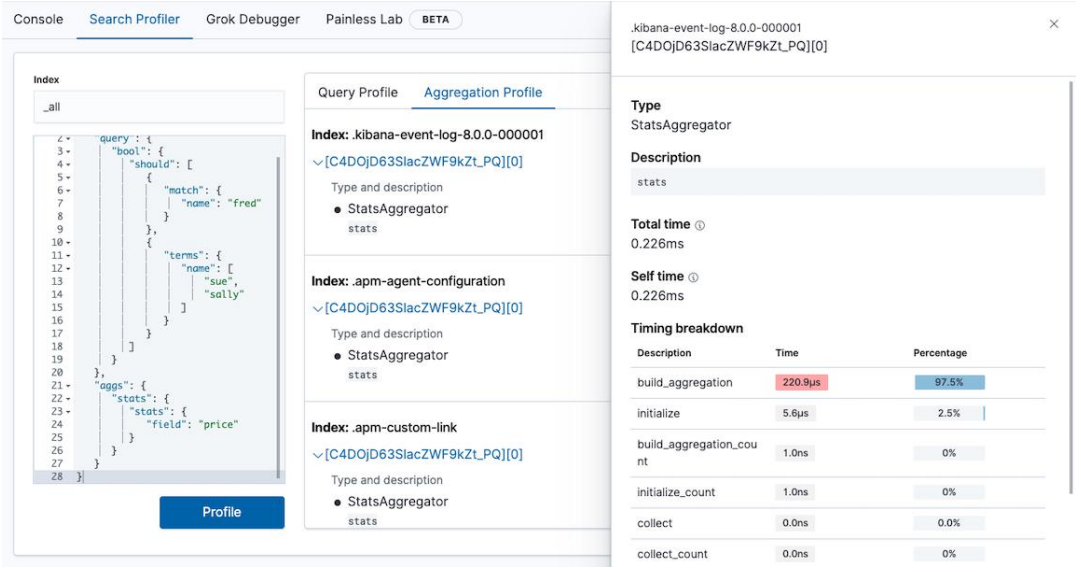
Query Profile



- 顶级 BooleanQuery 组件和 query 中 bool 相对应。
- 第二个 BooleanQuery 对应于条件查询，该条件在内部转换为一个 Boolean 的 should 子句，它有两个子查询，分别与 terms query 中的 “sue” 和 “sally” 相对应。
- 在 TermQuery 标有 “name:fred” 这是对应于 match: fred 在查询中。

如图你可以看到每一行的 Self time 和 Total time 都是不一样的，Self time 表示查询组件执行所需的时间。Total time 是查询组件及其所有子组件执行所花费的时间。因此，诸如 Boolean queries 之类的查询的总时间通常比 Self time 长。

Aggregation



点击 Aggregation Profile 去查看聚合的性能统计信息，选择 shard 查看聚合详细信息和时序分布。

Grok 调试器

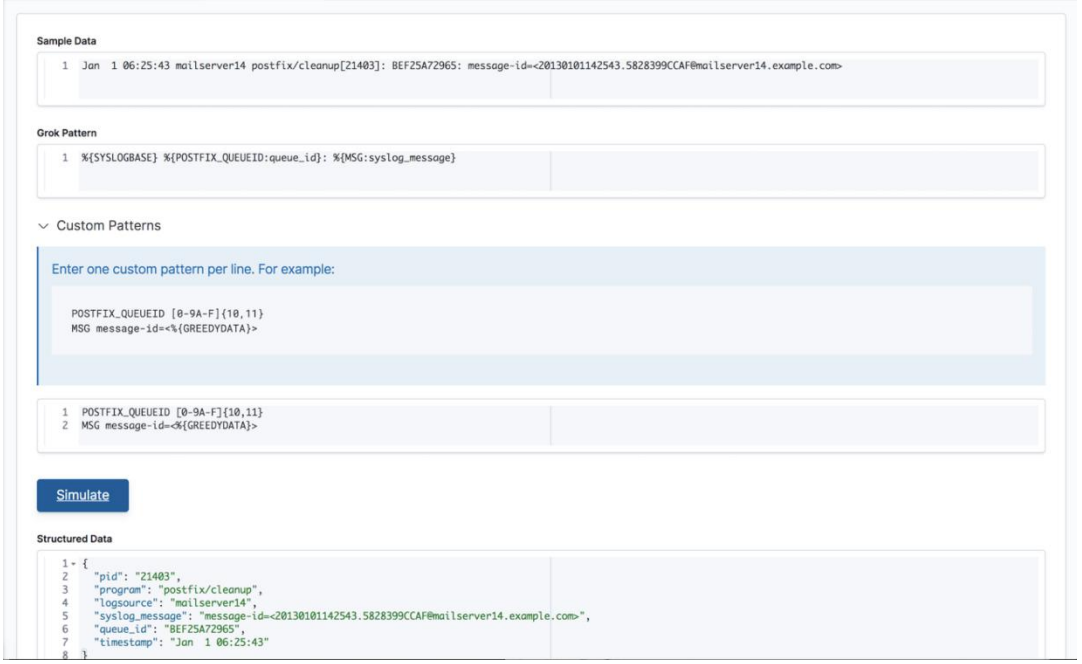
在数据处理管道中使用 grok 模式之前，请先对其进行构建和调试。

你可以在 Kibana Grok 调试器中构建和调试 grok 模式， 然后再在数据处理管道中使用它们。Grok 是一种模式匹配语法，可用于解析任意文本并对其进行结构化。Grok 非常适合解析 syslog，apache 和其他 Web 服务器日志，mysql 日志，以及具备可读性的任何日志格式。

使用实例：

自定义模式：

如果默认的 grok 模式字典不包含你所需的模式，则可以使用 Grok Debugger 定义，测试和调试自定义模式。



Painless 实验室

实时实验和调试 Painless 脚本（beta 功能）

此处不做过多的介绍。

采集管理

Logstash Pipelines 管理

Logstash Pipelines 管理就是集中管理配置 Logstash pipeline，使其事件处理和结果调试可视化。

准备步骤如下：

- 安装对应版本的 Logstash（和 Elasticsearch 版本保持一致）
- 开启 Logstash 监控和集中管理
- 创建 Logstash Pipeline
- 验证 Pipelines 集中管理

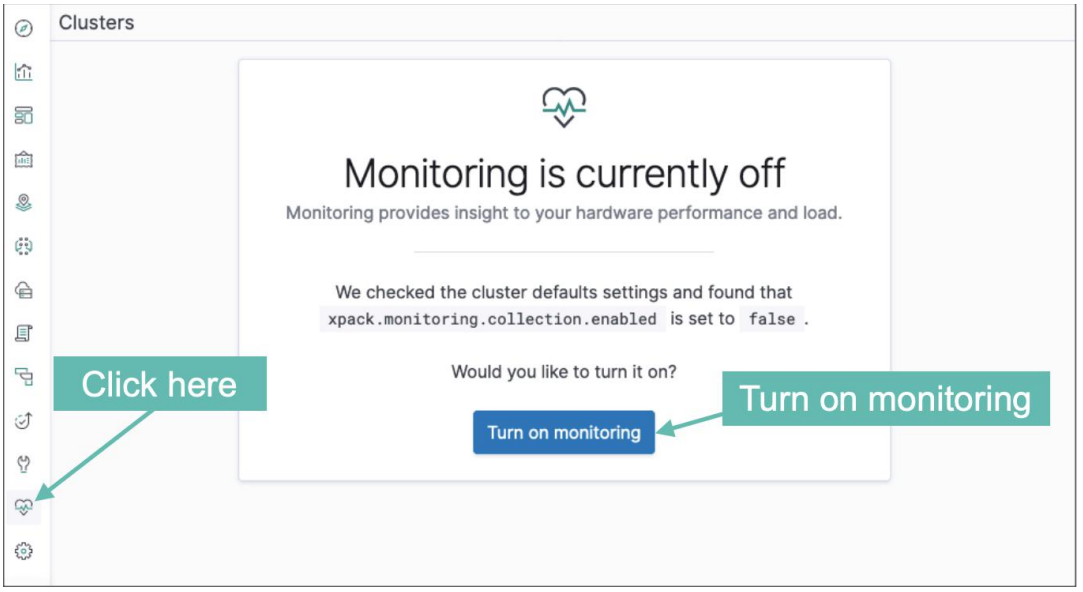
安装 Logstash

```
#下载安装包
wget https://artifacts.elastic.co/downloads/logstash/logstash-7.10.0-linux-x86_64.tar.gz
#解压到指定目录下
tar -zxvf logstash-7.10.0-linux-x86_64.tar.gz -C /opt/
#启动安装包
cd /opt/logstash-7.10.0
./bin/logstash
第一次启动会启动失败，这个不用担心，因为没有对 logstash 进行任何配置。
```

开启监控和集中管理

开启 Logstash 监控的前提，请确保在 Elasticsearch 集群中启动 xpack.monitoring.collection.enabled ： true

通过 Kibana 启动方式如下：



设置 Logstash 监控

在配置文件 logstash.yml 添加如下配置：

```
xpack.monitoring.enabled: true
xpack.monitoring.elasticsearch.username: "elastic"
xpack.monitoring.elasticsearch.password: "esTeam123456"
xpack.monitoring.elasticsearch.hosts: ["http://es-cn-6ja24dt4r007brz85.public.elasticsearch.aliyuncs.com:9200"]
xpack.monitoring.collection.interval: 10s
xpack.monitoring.collection.pipeline.details.enabled: true
```

设置 Logstash 集中管理

创建用于管理 Logstash 的用户和角色：

```
POST _xpack/security/role/logstash_writer
{
  "cluster": ["manage_index_templates", "monitor", "manage_ilm"],
  "indices": [
    {
      "names": [ "test1_*", ".monitoring-logstash-*" ],
      "privileges": ["write","create","create_index","manage","manage_ilm"]
    }
  ]
}

POST _xpack/security/user/logstash_internal
{
  "password" : "123456",
  "roles" : [ "logstash_writer", "logstash_admin", "logstash_system"],
  "full_name" : "Internal Logstash User"
}
```

创建一个 logstash_writer 角色，并给其分配相应 cluster 和 indices 的权限。

在配置文件 logstash.yml 添加如下配置：

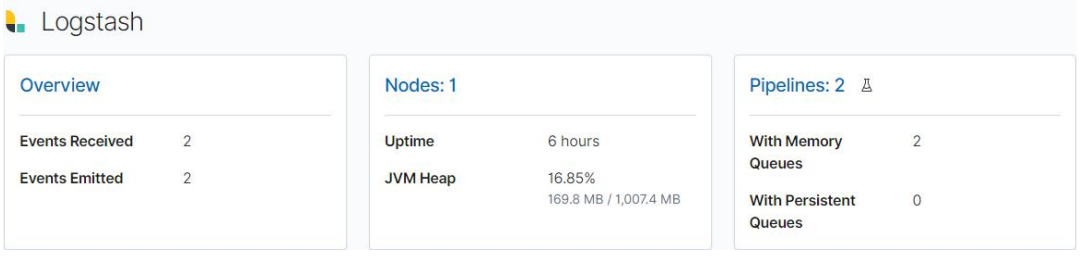
```
xpack.management.enabled: true
xpack.management.pipeline.id: ["Team7_test"]
xpack.management.elasticsearch.username: "logstash_internal"
xpack.management.elasticsearch.password: "123456"
xpack.management.elasticsearch.hosts: ["http://es-cn-6ja24dt4r007brz85.public.elasticsearch.aliyuncs.com:9200"]
xpack.management.logstash.poll_interval: 1s
```

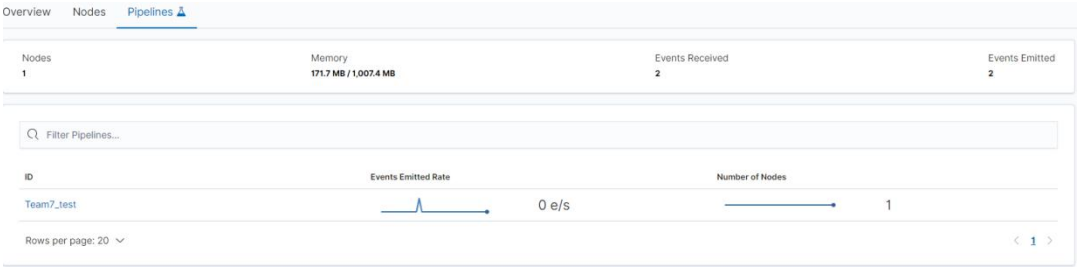
注意：X-pack.management.pipeline.id 中支持配置多个 pipeline.id，请保证此处填入的和 Kibana 创建发布的保持一致。

启动 Logstash

```
[root@master01 logstash-7.10.0]# ./bin/logstash --debug
Using JAVA_HOME defined java: /opt/jdk1.8.0_251
WARNING, using JAVA_HOME while Logstash distribution comes with a bundled JDK
```

在 Kibana monitor 查看 Logstash 监控

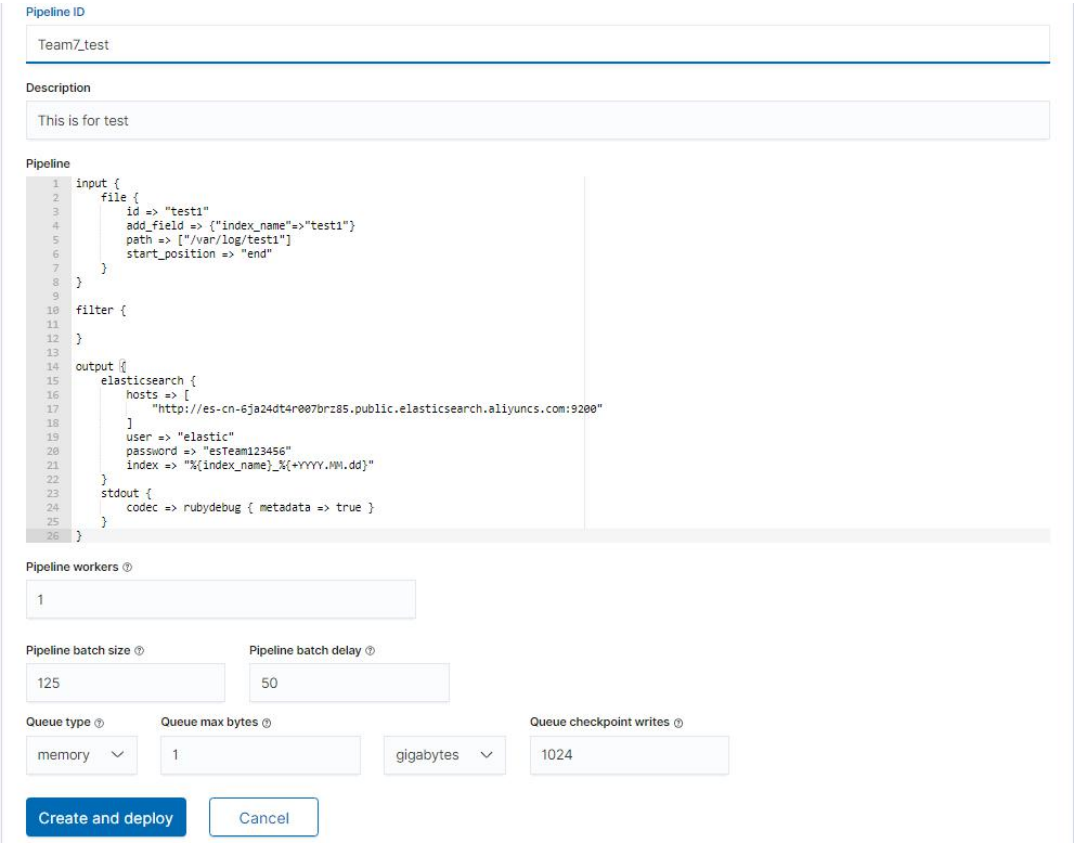




依次可以查看 Logstash 的概览，Logstash 节点和 Logstash pipeline 监控。

创建 Logstash Pipeline

在 Kibana 上操作，依次点击 Mangement-> Stack Management ->Logstash Pipelines->Create pipeline，输入配置参数后，点击 Create and Deploy 即可创建和发布成功。



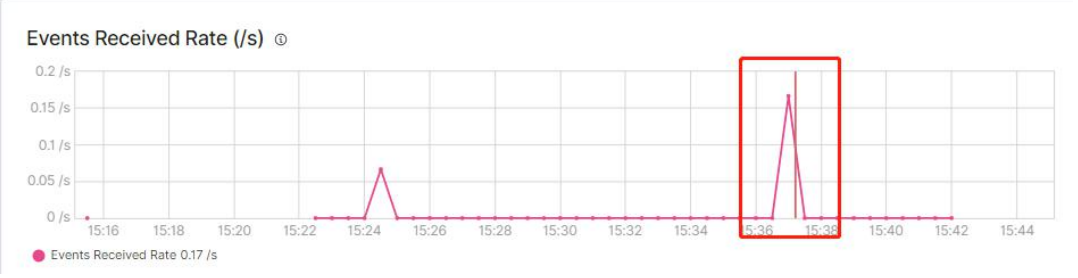
验证 Pipelines 集中管理

首先往 /var/log/test1 文件中写入多条日志：

```
[root@master01 ~]# echo "test1" >> /var/log/test1
[root@master01 ~]# echo "test2" >> /var/log/test1
[root@master01 ~]# echo "test3" >> /var/log/test1
[root@master01 ~]# echo "test4" >> /var/log/test1
```

在 Elasticsearch 中可查询到刚刚 test1_20210501 写入的 4 条日志。

也可以在 Logstash 的监控中，看到在 15:36-15:38 之间采集到数据。



使用总结

该功能属于 X-pack 的高级特性，基础版本的 License 不能体验；

使用该功能时无需在 Logstash 节点配置任何 pipeline；

创建和发布的 pipeline 会自动下发至该集群管理的所有 Logstash 实例，不能进行单独管控。

Beats 集中管理

功能概述

在 6.5 的版本中，Elastic 官方在 Kibana 中引入了一个新功能：Beats 的集中管理。主要用来解决

Beats 的集中配置管理，通过此功能，你无需重复的登录每台机器上调整配置，这样可以大大减轻配置管理相关的工作量。

Beats 的集中管理功能很简单，包含 3 个步骤：

- 注册 Beats：

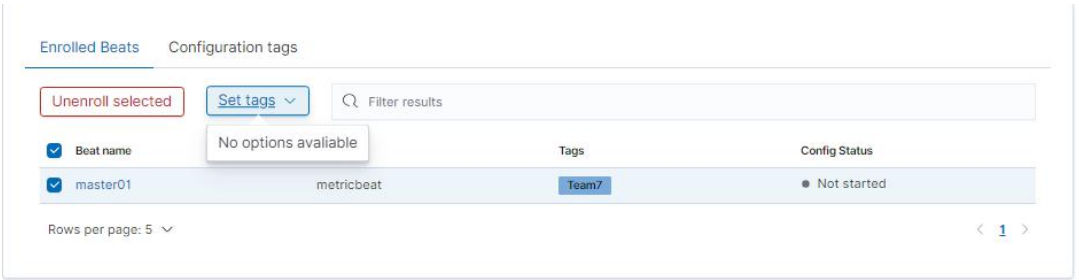
注册 Beats 就是通过 Beats enroll 命令将 Beats 注册集群中，目前只支持两种 beats（filebeats 和 metricbeat），注册成功以后才能被管控。

- 配置标签

使用一种使用配置标签的机制来对相关配置进行分组，详细的相关配置在配置标签中进行添加

目前支持 4 种类型。Filebeat input，Filebeat module, Metricbeat module 和 output。

最后在 Beats 列表中绑定配置标签，即可自动批量下发配置标签中的相关配置，如下图所示。



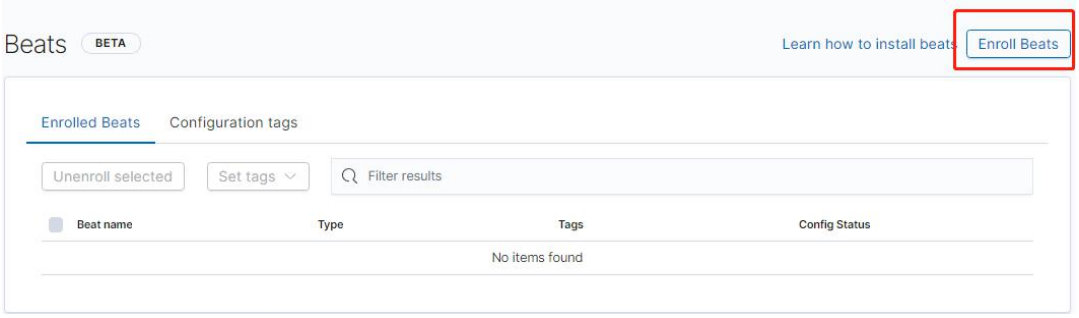
验证方案

安装 Metricbeat：

```
wget https://artifacts.elastic.co/downloads/beats/metricbeat/metricbeat-7.10.0-linux-x86_64.tar.gz
tar -zxvf metricbeat-7.10.0-linux-x86_64.tar.gz
cd /opt/metricbeat-7.10.0-linux-x86_64
```

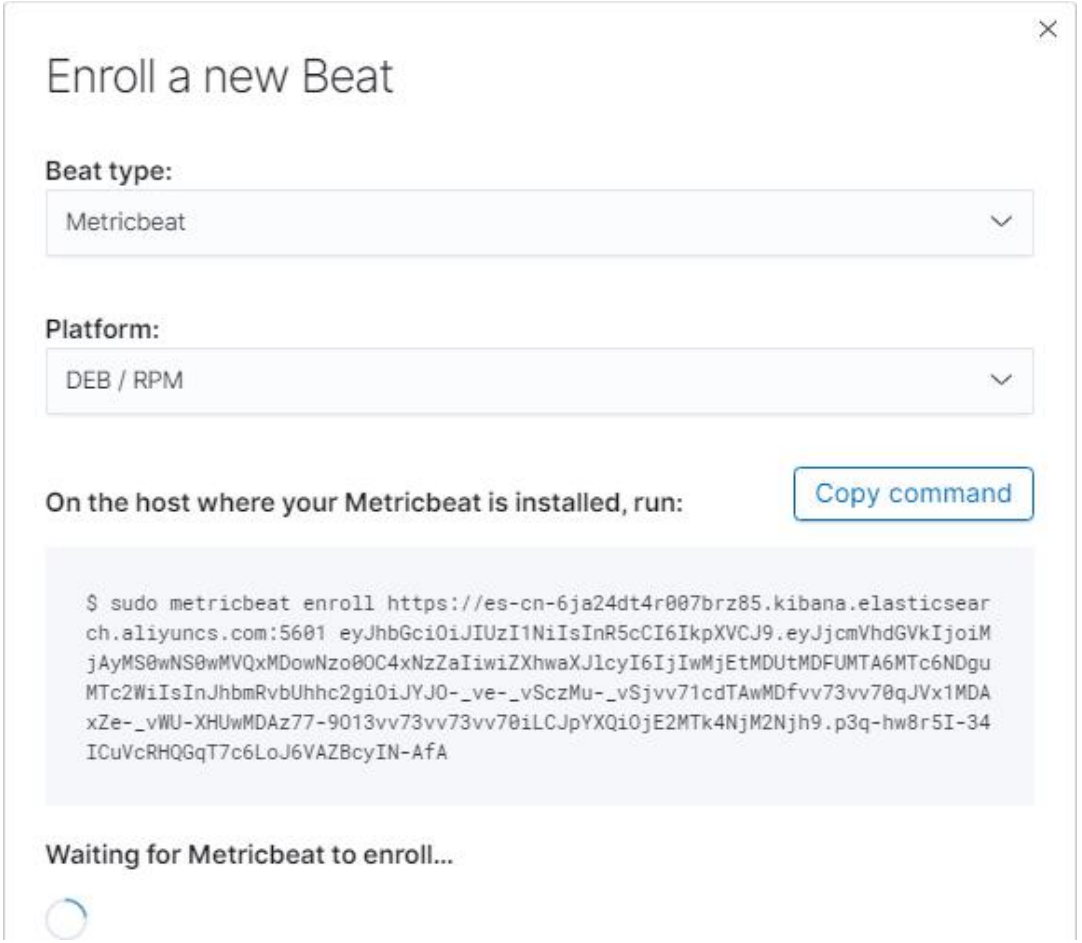
创建 Enroll Beats

在 Kibana 上操作，依次点击 Mangement-> Stack Management ->Beats Central Management->Enroll Beats



选择 Beat type=Metricbeat 和 Platform=DEB/RPM

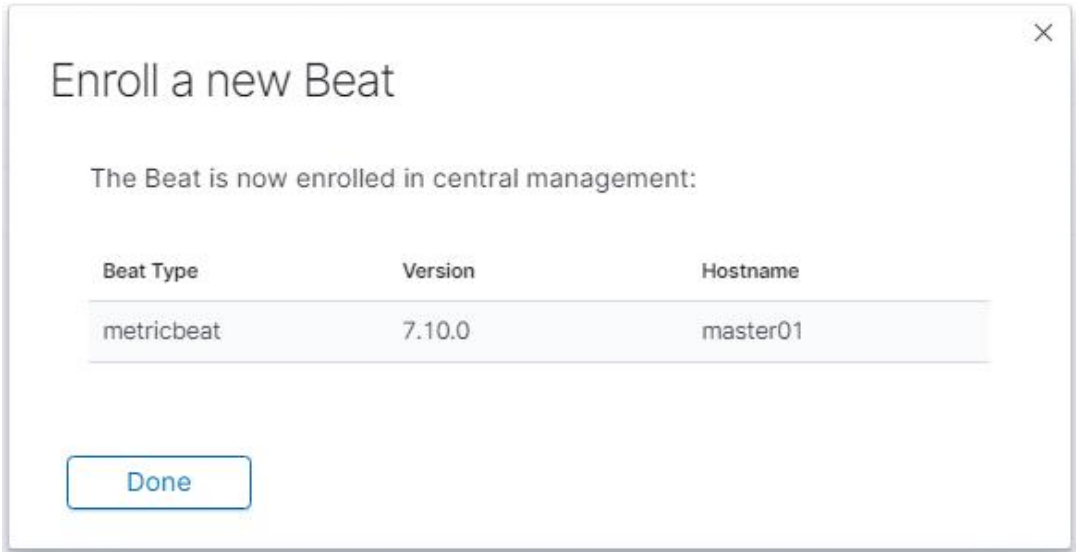
即可得到 Metricbeat Enroll 命令：



在 Metricbeat 安装节点执行：

```
[root@master01 metricbeat-7.10.0-linux-x86_64]# ./metricbeat enroll https://es-cn-6ja24dt4r007brz85.kibana.elasticsearch.aliyuncs.com:5601 eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJjcmVhdGVkIjoiMjAyMS0wNS0wMVQxMDowNzo0OC4xNzZaIiwiZXhwaXJlcyI6IjIwMjEtMDUtMDFUMTA6MTc6NDguMTc2WiIsInJhbmRvbUhhc2giOiJYJO-_ve-_vSczMu-_vSjvv71cdTAwMDfvv73vv70qJVx1MDAxZe-_vWU-XHUwMDAz77-9O13vv73vv73vv70iLCJpYXQiOjE2MTk4NjM2Njh9.p3q-hw8r5I-34ICuVcRHQGqT7c6LoJ6VAZBcyIN-AfA
This will replace your current settings. Do you want to continue? [Y/n]:y
Saving a copy of current settings to metricbeat.yml.2021-05-01T18-12-22.bak
Enrolled and ready to retrieve settings from Kibana
[root@master01 metricbeat-7.10.0-linux-x86_64]#
```

即可在显示在 Kibana 注册成功。

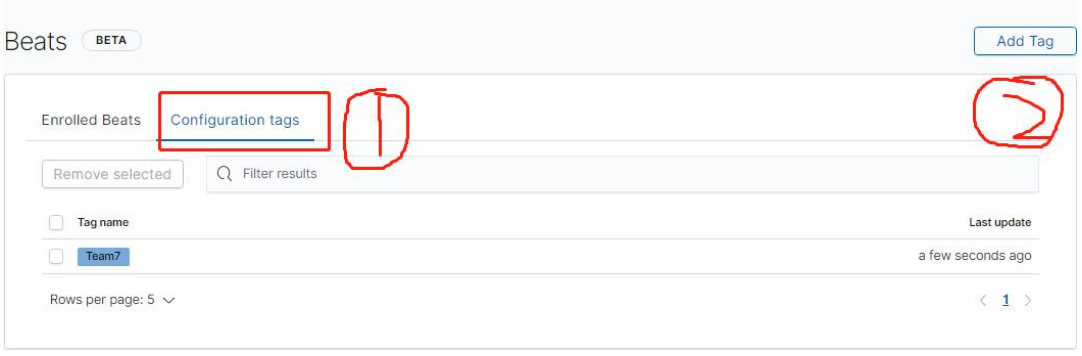


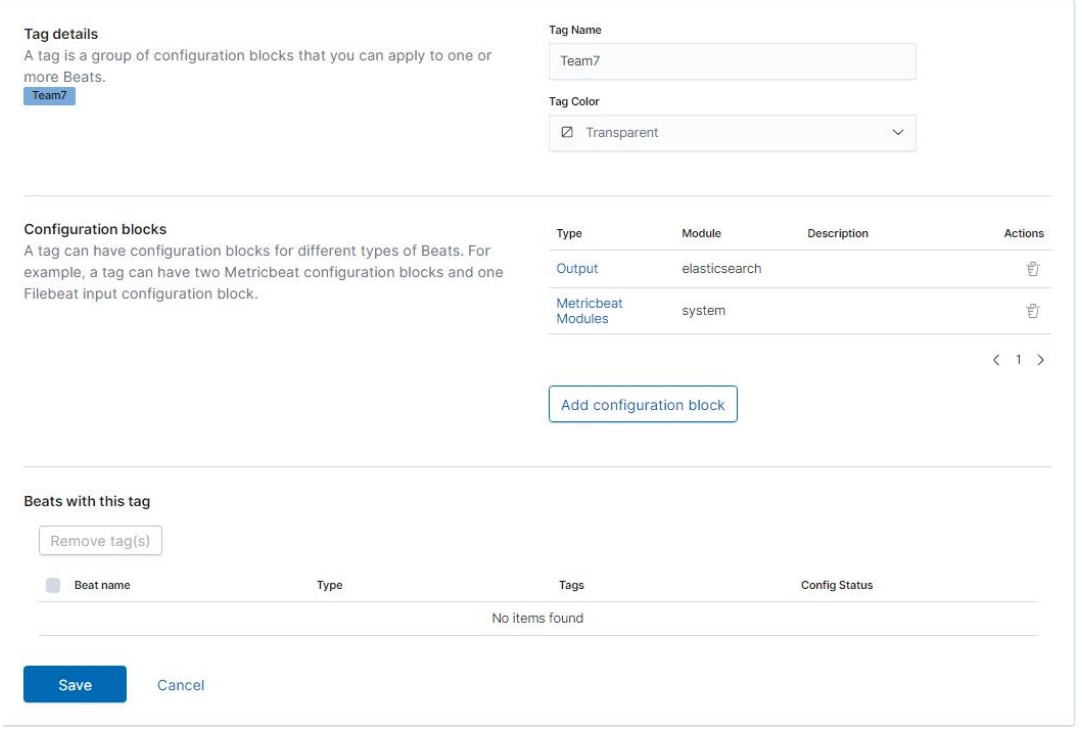
点击 Done 即可。

启动 metricbeat：

```
./metricbeat -e -c metricbeat.yml
```

创建 Configuration tags

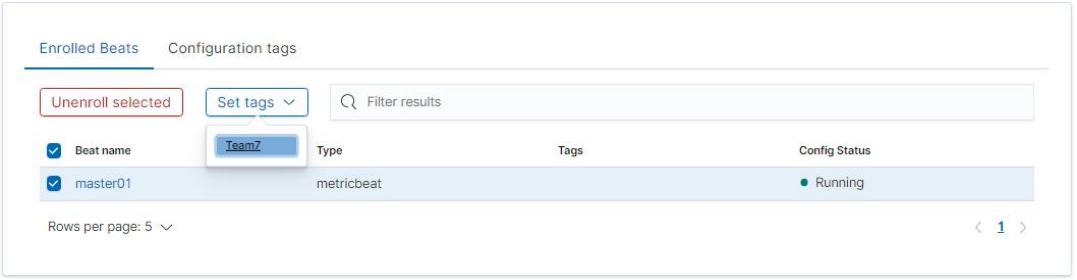


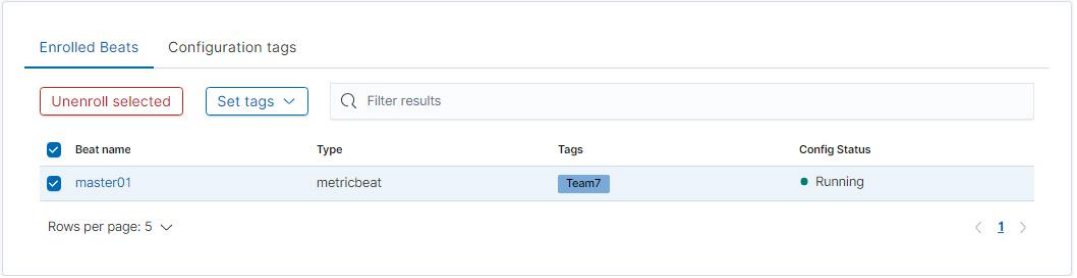


创建一个 Team7 的 Configuration tags，并在其中配置了两个 Configuration Blocks。

- 输出到 Elasticsearch
- 开启 system 的 metricbeat modules

应用配置到 Beats





验证数据

Metricbeat 索引的文档数持续增加：

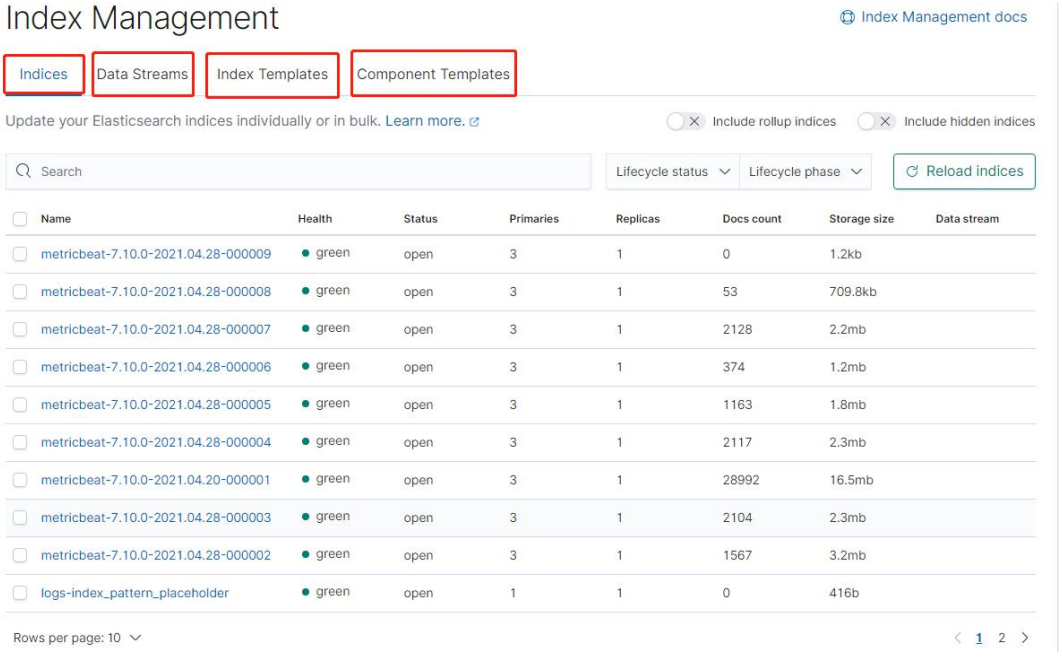
```
GET metricbeat-7.10.0-2021.05.01/_count
{
  "count" : 405,
  "_shards" : {
    "total" : 3,
    "successful" : 3,
    "skipped" : 0,
    "failed" : 0
  }
}
```

在 Kibana 的 discover 页面分析 Metricbeat 索引数据：

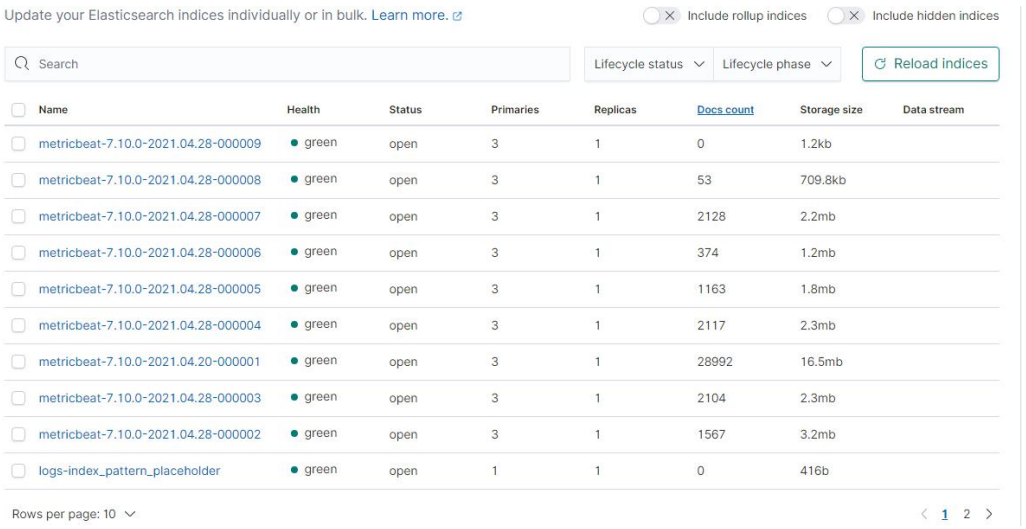
数据管理

索引管理

Kibana 的索引管理菜单中，主要管理着和索引相关的四类数据：



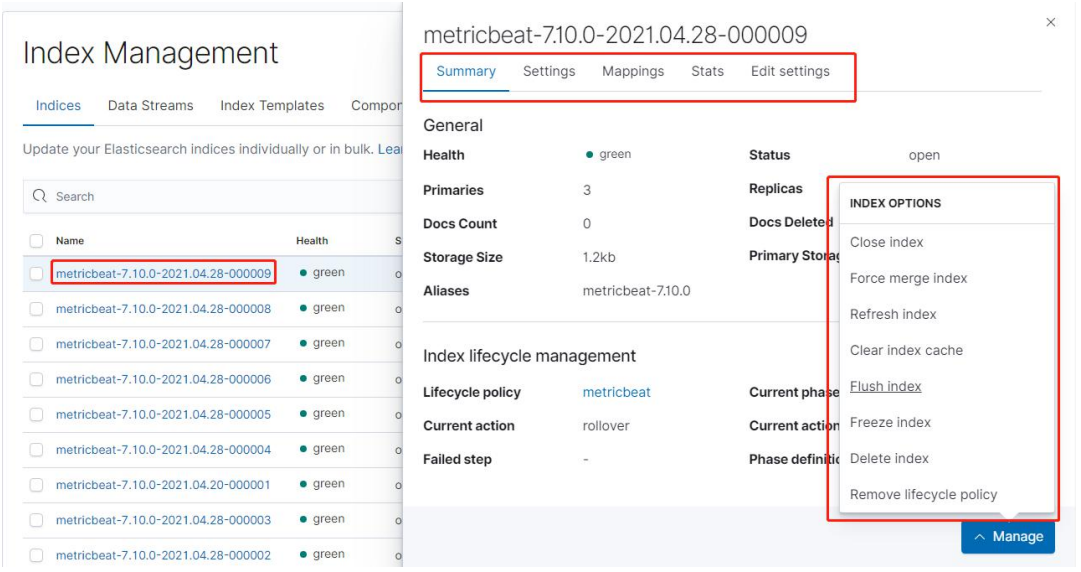
索引列表：



管理着集群中的所有 indices，包含 Rollup 索引（通过 Rollup 聚合索引）和隐藏索引（名字以.开头的索引）。

同时还可以根据是否被 ILM 管理（Managed 和 Unmanaged）和处于 ILM 管理阶段（Hot，warm，Frozen，Cold 和 Delete）进行查询和过滤。

点击索引名可以查看索引的详细：

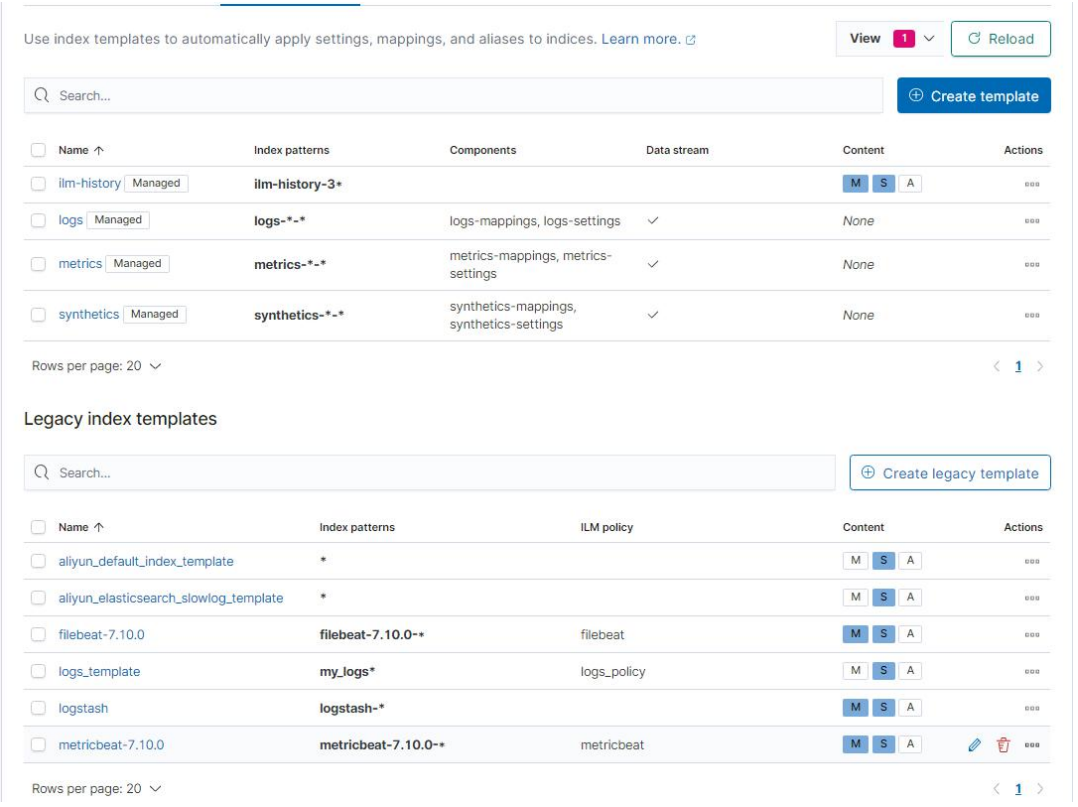


在索引详情中，可以查看索引概览信息，settings，mappings，当前索引状态数据以及对该索引的 settings 进行修改。

同时你可以对该索引进行关闭，segment 强制合并，Refresh，清空索引缓存，Flush，冻结，删除和解除 ILM 管理等操作

索引模板管理

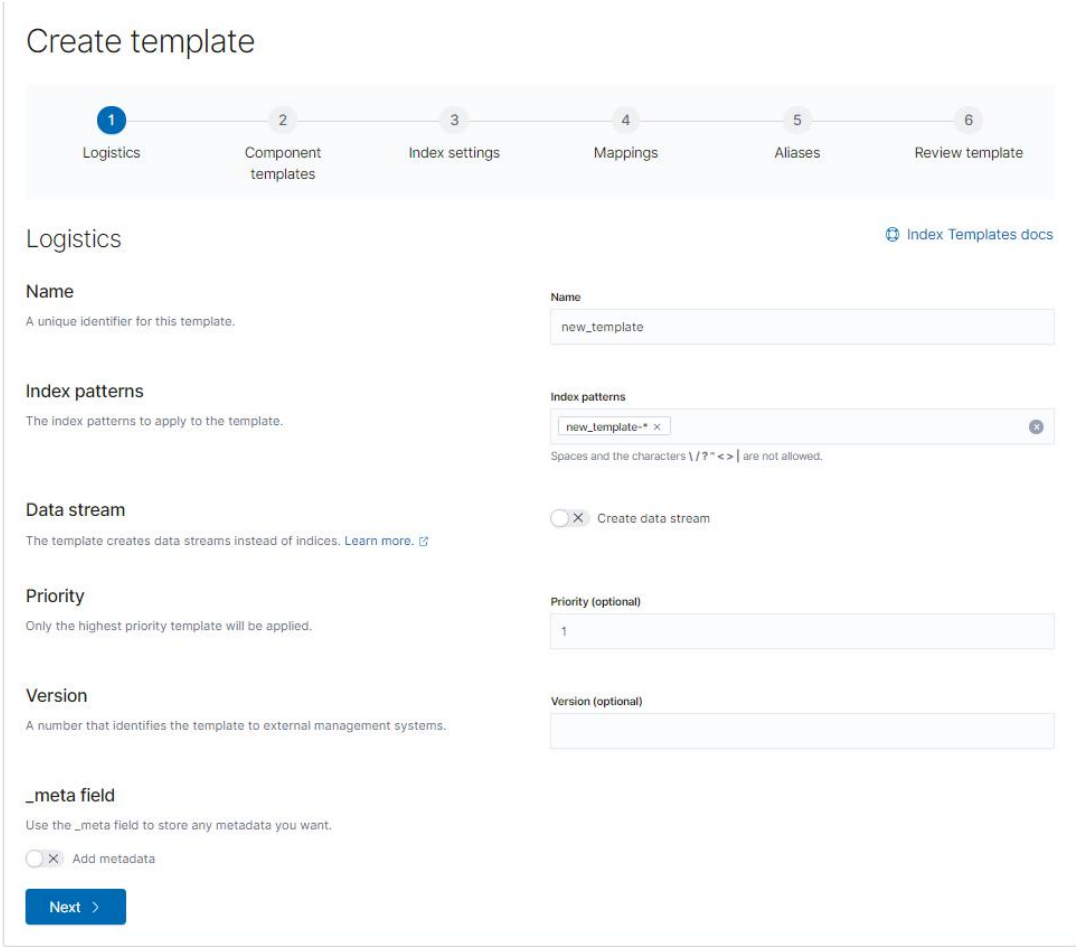
在创建索引时，根据索引名匹配符合条件的索引自动设置索引的 settings，mappings 和 aliases



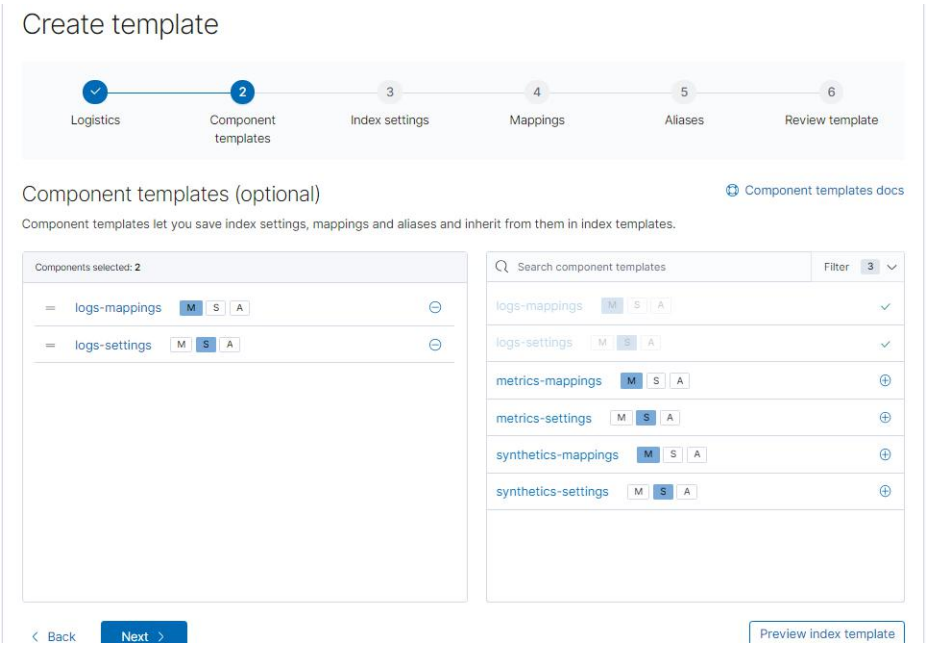
在此处创建索引模板，支持创建两种方式创建索引模板。

新版模板创建流程：

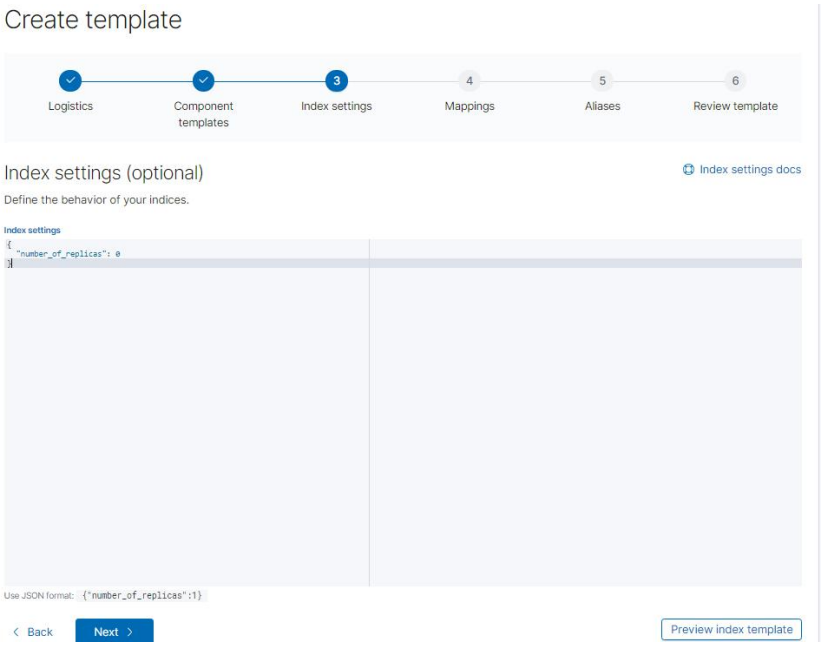
设置模板总览信息



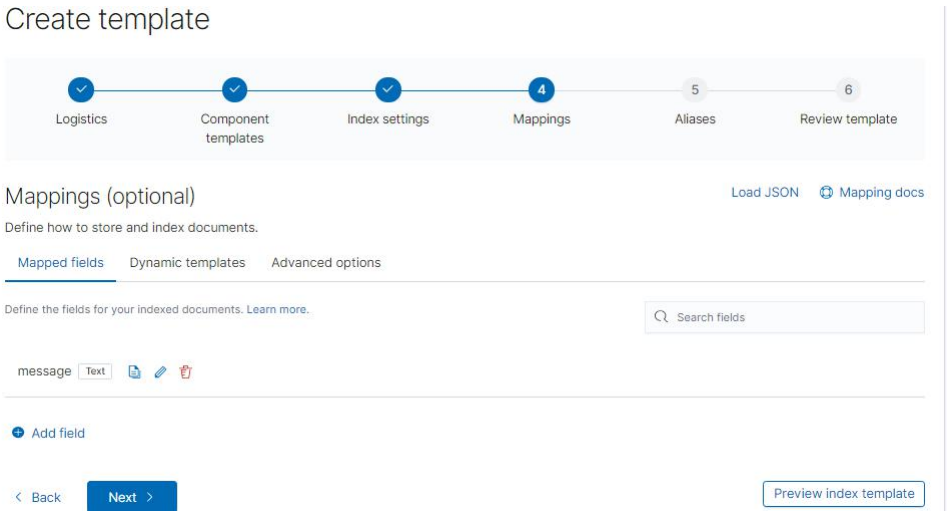
选择 templates 组件，包括 settings，mappings，aliases：



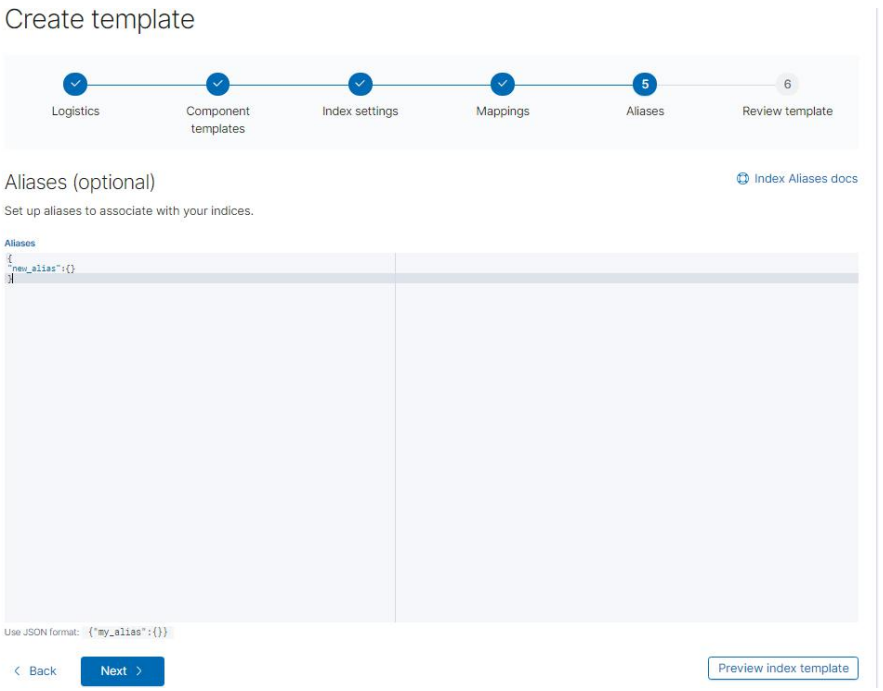
自定义修改 index settings：



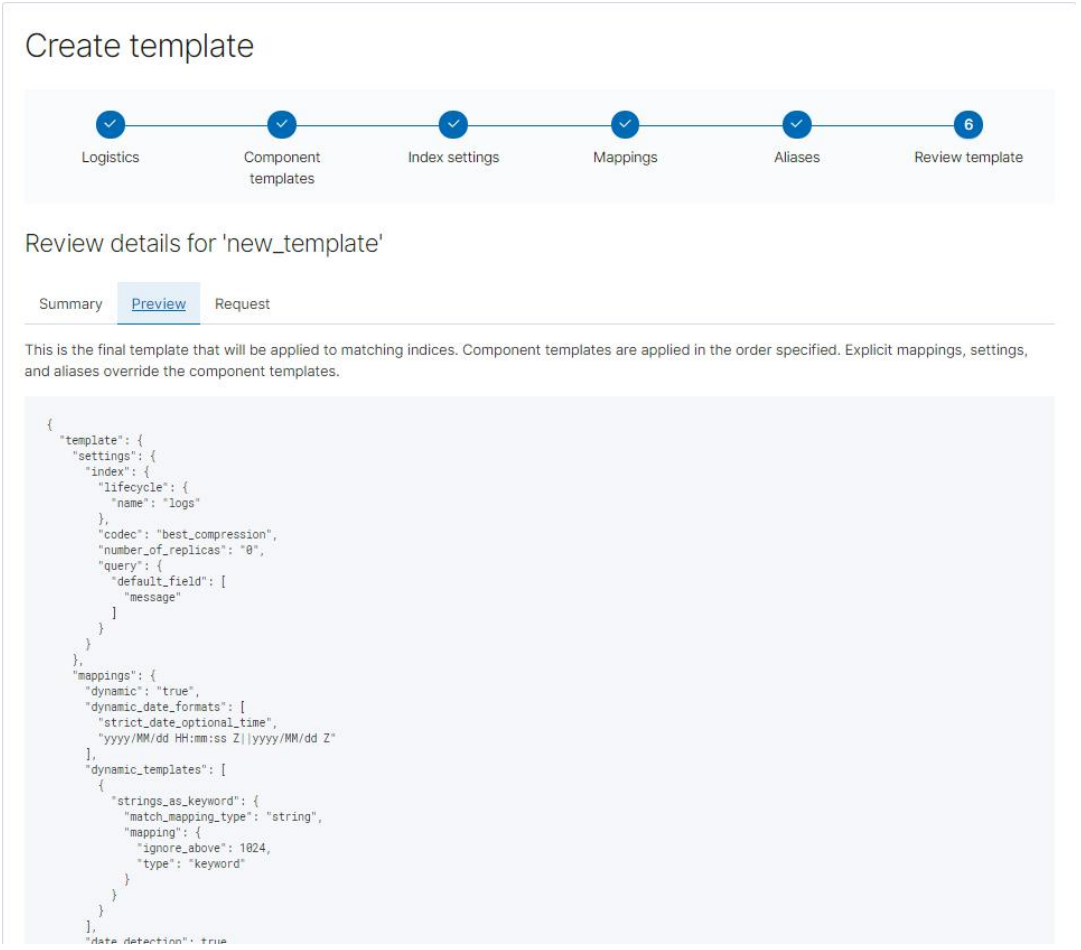
自定义 mappings：



添加 Aliases 信息：

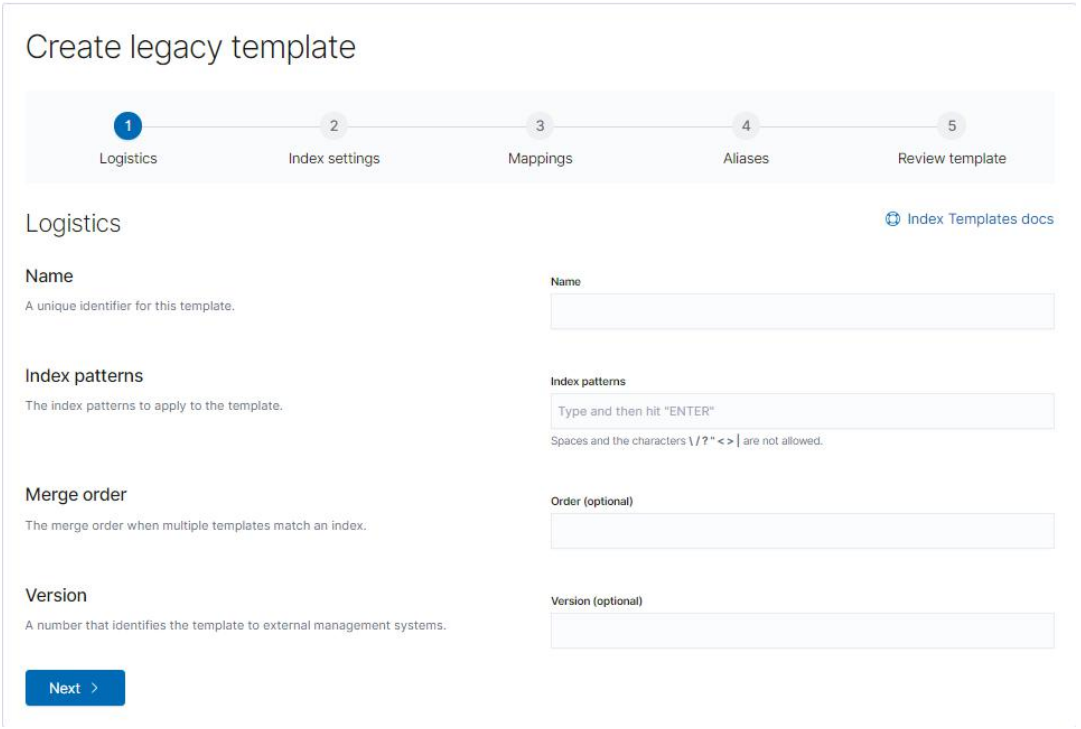


检查设置好的 template 信息：



点击创建按钮即可完成模板创建。

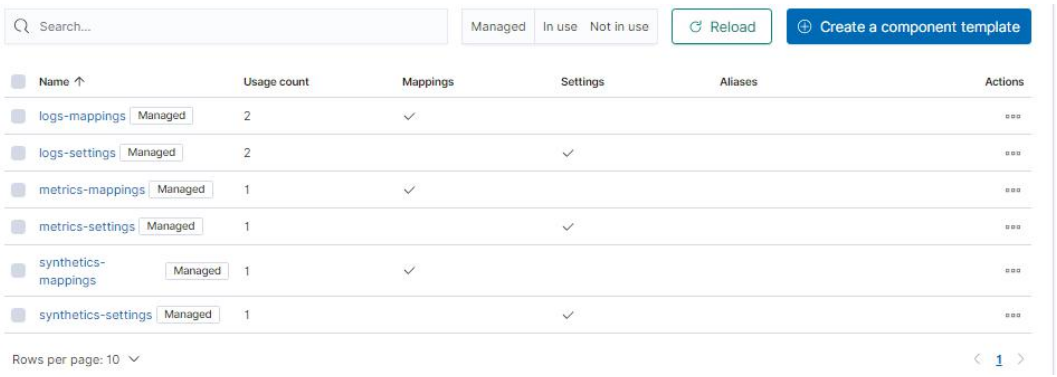
旧版模板创建流程：



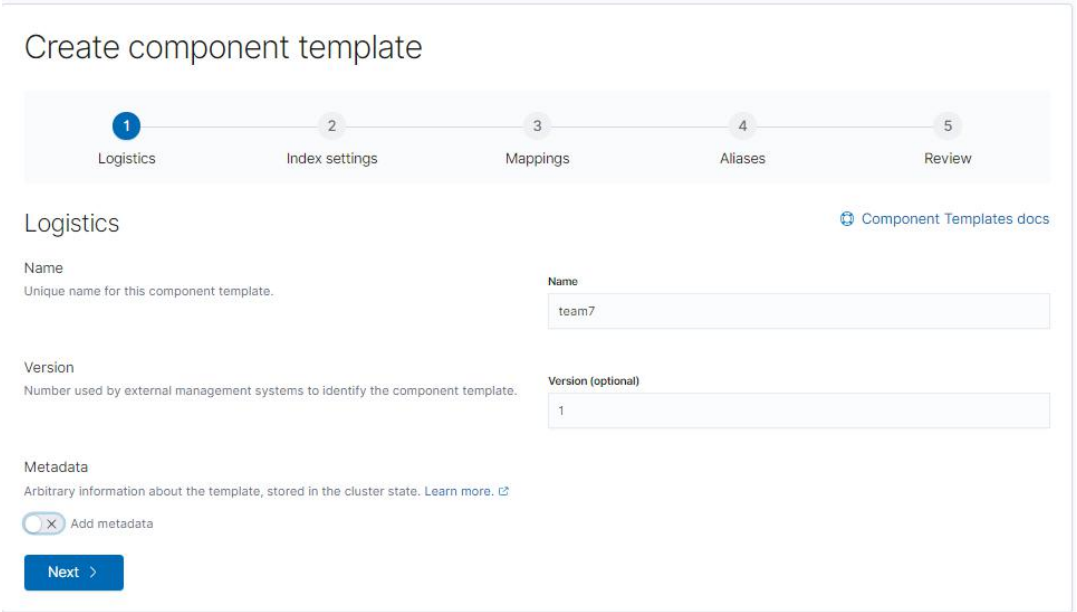
和新版本的创建流程基本相同，少了可选择已有 template 组件这一步。

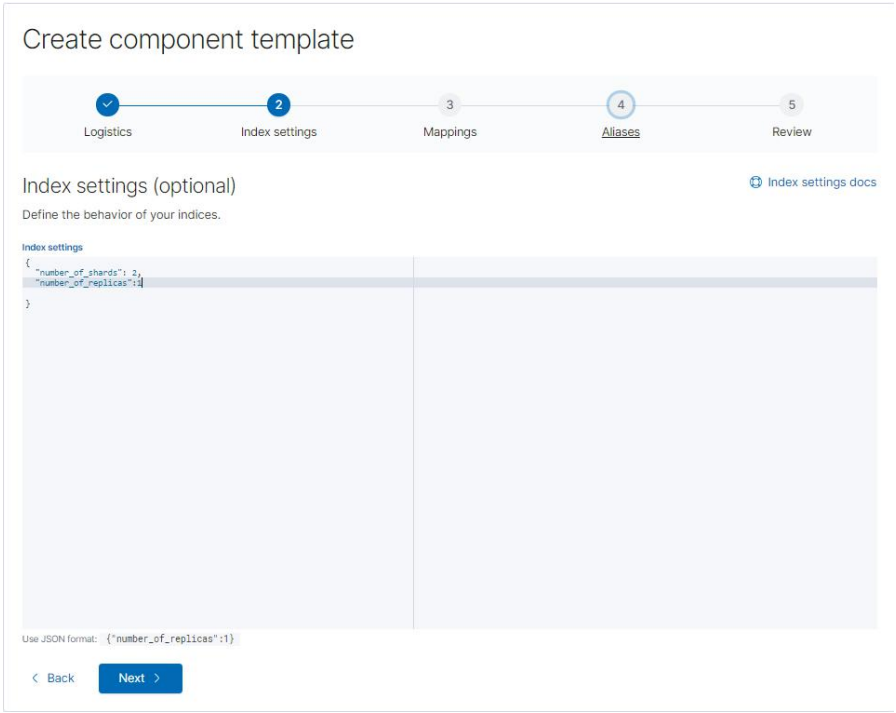
模板组件管理

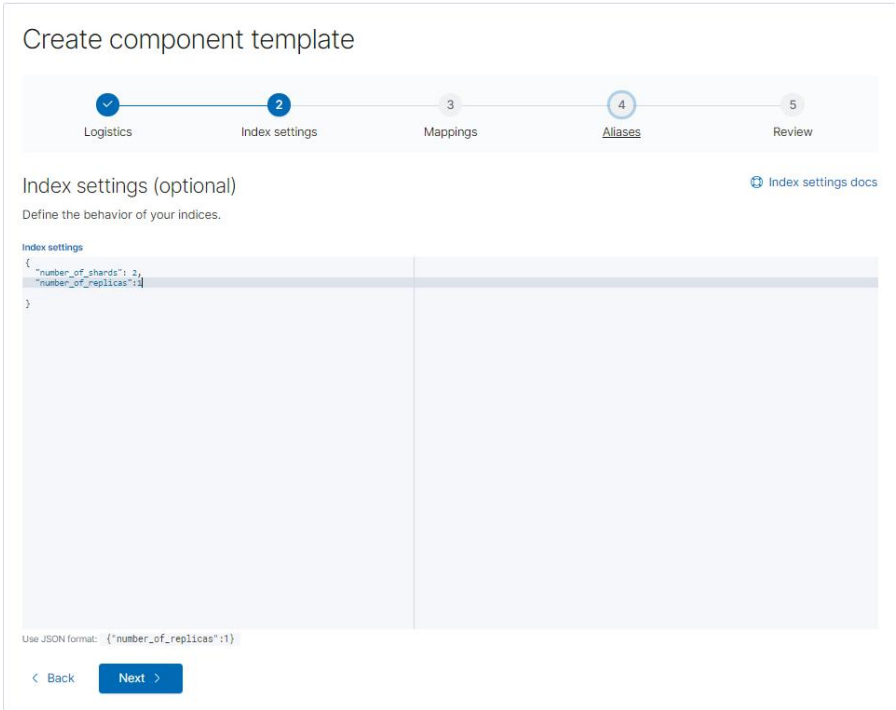
使用组件模板可以在多个索引模板中重用设置，映射和别名配置：

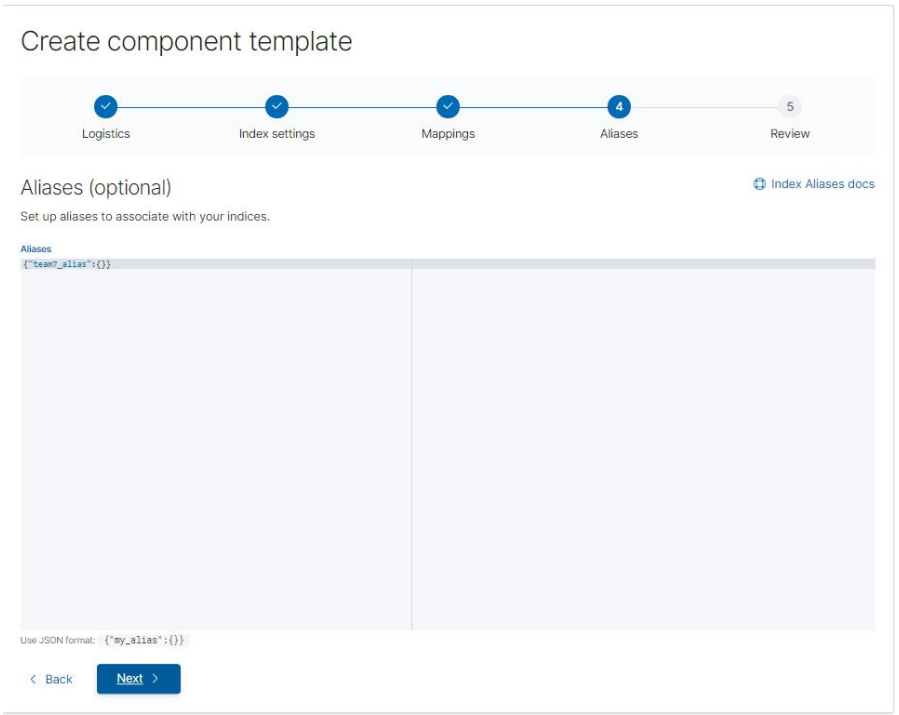


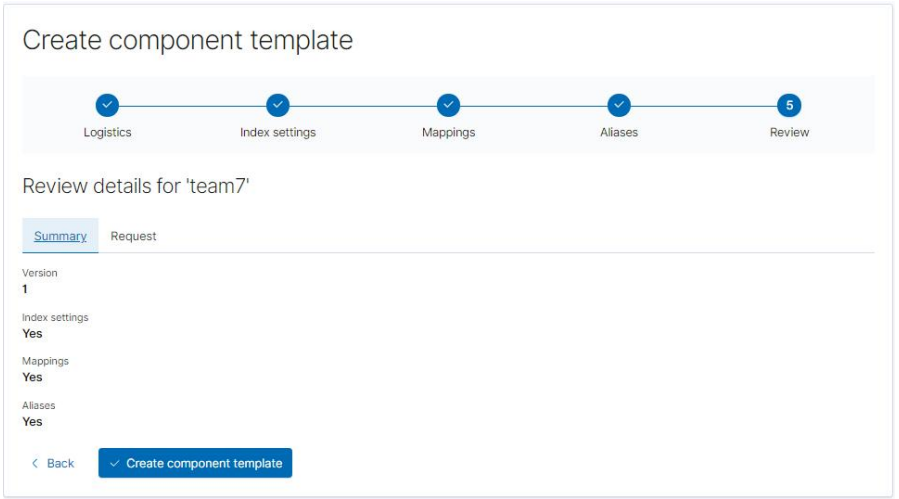
创建索引组件模板的流程如下：




Create component template
Logistics
2
Index settings
3
Mappings
4
Aliases
5
Review
Index settings (optional)
Index settings docs
Define the behavior of your indices.
Index settings
{
"number_of_shards": 2,
"number_of_replicas":1
}
Use JSON format: {"number_of_replicas":1}
Back
Next


Create component template
Logistics
2
Index settings
3
Mappings
4
Aliases
5
Review
Index settings (optional)
Index settings docs
Define the behavior of your indices.
Index settings
{
"number_of_shards": 2,
"number_of_replicas":1
}
Use JSON format: {"number_of_replicas":1}
Back
Next


Create component template
Logistics
Index settings
Mappings
4
Aliases
5
Review
Aliases (optional)
Index Aliases docs
Set up aliases to associate with your indices.
Aliases
{"team7_alias":{}}
Use JSON format: {"my_alias":{}}
Back
Next


Create component template
Logistics
Index settings
Mappings
Aliases
5
Review
Review details for 'team7'
Summary
Request
Version
1
Index settings
Yes
Mappings
Yes
Aliases
Yes
Back
Create component template

创建成功的效果：

创建成功后，该索引组件模板，就可以在创建索引模板时进行复用。

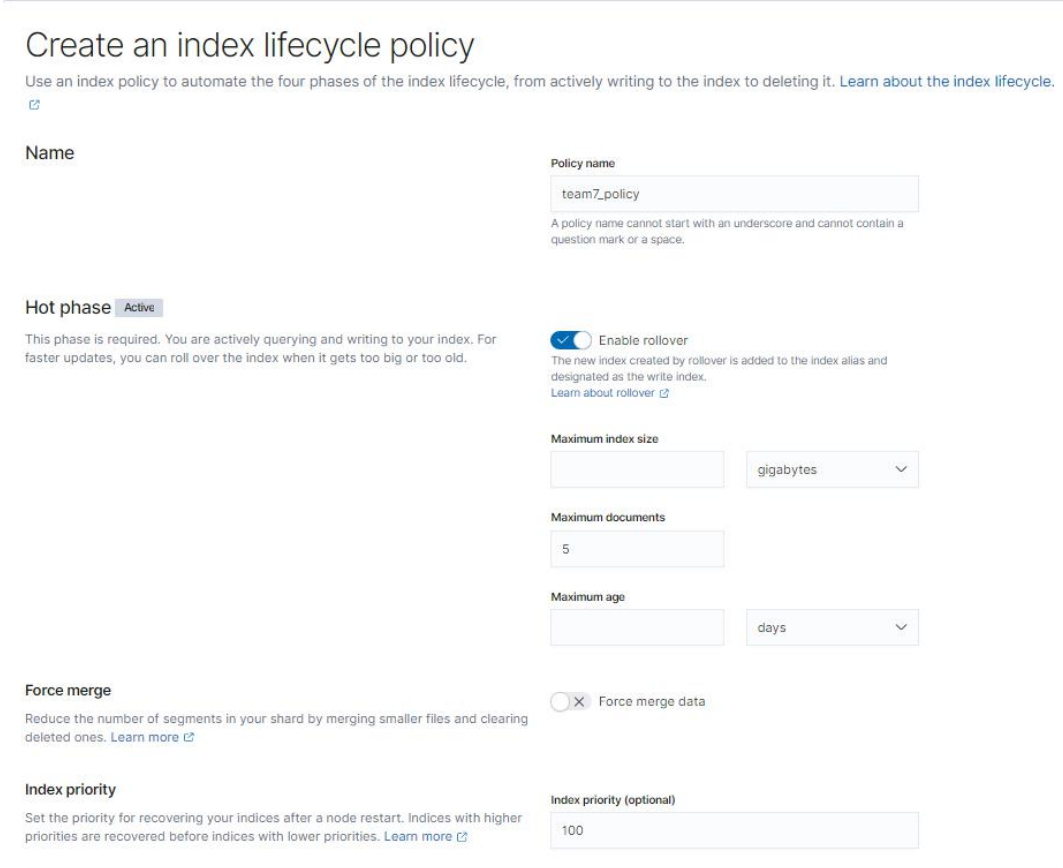
索引生命周期

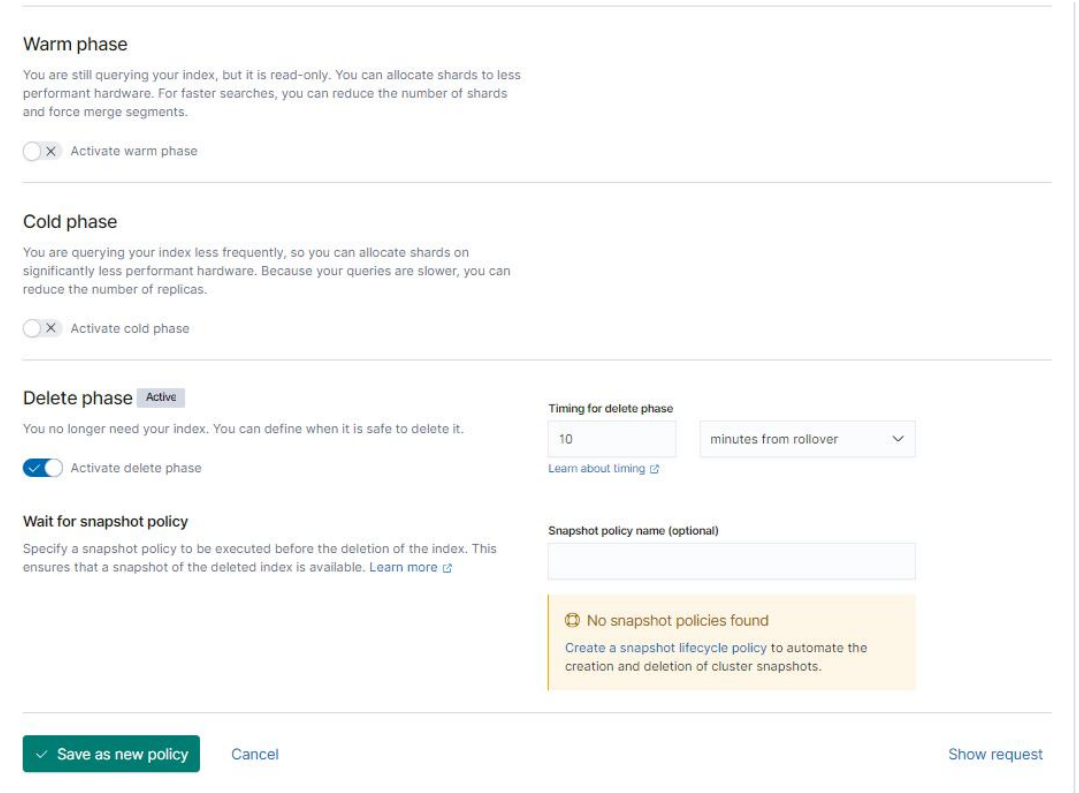
ElasticSearch 在 6.7 版本推出的索引生命周期管理(index lifecycle management，简称 ILM），生命周期把索引分为四个阶段，Hot，Warm，Cold，和 Delete。这也是 Elastic 目前官方比较推荐的索引管理方法。

hot	索引可写入，也可查询，也就是我们通常说的热数据。
warm	索引通常不会被写入，但仍然会被查询。
cold	索引不再被更新，并且很少被查询。这些信息仍然需要可搜索，但如果查询速度较慢也没关系。
delete	索引不再需要，可以安全地删除。

创建一个 Index Lifecycle policy

在 Kibana 上操作，依次点击 Mangement-> Stack Management -> Index Lifecycle Policies->Create policy，输入配置参数后，点击 Save as new policy 可以生成一个新的策略。


Create an index lifecycle policy
Use an index policy to automate the four phases of the index lifecycle, from actively writing to the index to deleting it. Learn about the index lifecycle.
Name
Policy name
team7_policy
A policy name cannot start with an underscore and cannot contain a question mark or a space.
Hot phase
Active
This phase is required. You are actively querying and writing to your index. For faster updates, you can roll over the index when it gets too big or too old.
Enable rollover
The new index created by rollover is added to the index alias and designated as the write index.
Learn about rollover
Maximum index size
gigabytes
Maximum documents
5
Maximum age
days
Force merge
Reduce the number of segments in your shard by merging smaller files and clearing deleted ones. Learn more
Force merge data
Index priority
Set the priority for recovering your indices after a node restart. Indices with higher priorities are recovered before indices with lower priorities. Learn more
Index priority (optional)
100



点击 Show request，可得到创建此 Policy 的请求语句：

```
PUT _ilm/policy/team7_policy
{
  "policy": {
    "phases": {
      "hot": {
        "min_age": "0ms",
        "actions": {
          "rollover": {
            "max_docs": 5
          },
```

```
            "set_priority": {
              "priority": 100
            }
          }
        },
        "delete": {
          "min_age": "10m",
          "actions": {}
        }
      }
    }
}
```

在集群中验证创建的策略：

配置 lifecycle 检测时间：

```
PUT _cluster/settings
{
  "transient": {
    "indices.lifecycle.poll_interval": "5s"
  }
}
```

默认为十分钟，为了测试效果，改为 5 秒钟。

创建索引模板：

```
PUT _template/team7_template
{
  "index_patterns": [
    "my_team7*"
  ],
  "settings": {
    "number_of_shards": 1,
    "number_of_replicas": 1,
    "index.lifecycle.name": "team7_policy",
    "index.lifecycle.rollover_alias": "my_team7",
    "index.default_pipeline": "indexed_at"
  }
}
```

索引以 my_team7-开头的自动采用 settings 的配置。

index.lifecycle.name 表示采用 team7_policy 的策略，index.lifecycle.rollover_alias 表示创建使用该模版创建的索引

统一用 my_team7 的别名进行管理。

创建索引：

```
PUT my_team7-000001
{
  "aliases": {
    "my_team7": {
      "is_write_index": true
```

```
      }
    }
  }
```

创建一个开始的索引，并设置当前索引可通过索引别名写入。

验证功能:

一切准备就绪，我们开始验证。

首先执行下面的新建文档操作 5 次

```
POST my_team7/_doc
{
  "message": "this is team7 test"
}
```

```
health status index          uuid                   pri rep docs.count docs.deleted store.size pri.store.size
green  open   my_team7-000001 LpOrrKNmRY6a55fz7xhXOA   1   1          0            0       416b           208b
```

之后 Rollover 执行，新的索引创建，如下所示:

```
1 health status index          uuid                   pri rep docs.count docs.deleted store.size pri.store.size
2 green  open   my_team7-000002 mfSAtoHnTTKiTekT1SfEwg   1   1          1            0       416b           208b
3 green  open   my_team7-000001 LpOrrKNmRY6a55fz7xhXOA   1   1          5            0      9.5kb          4.7kb
```

10m 以后， my_team7-000001 删除至此，一个完整的 ILM Policy 执行的流程就结束了，而后续 my_team7-000002 也会按照这个设定进行流转。

索引快照和恢复

- console 开发工具
  - console 终端
  - Profiler 分析
  - Grok 调试工具

- 采集管理
  - Logstash 管道
  - Beats 集中管理

- 数据管理
  - 索引管理
  - 索引生命周期
  - 索引快照和恢复

创作人简介：

高冬冬，从事运维架构，参与私有云平台运维体系平台开发，金融行业统一日志平台(PB级)的规划和建设，正在进行包含监控体系和智能运维的统一监控告警平台的规划和设计。

自 2016 年开始使用和研究 Elastic Stack 相关技术栈，擅长使用 Elastic Stack 解决日志分析和可观测性相关问题，对 PB 级日志平台的规划，部署，优化和运维有丰富的经验和实践，喜欢学习运用新技术，解决工作中问题和提高生产力。希望以后有更多的机会，分享输出更多有价值的东西给大家。

四、应用实践

4.1 企业搜索应用场景

4.1.1 ES 在舆情搜索中的实践

创作人：王欢

审稿人：杨振涛

业务背景

网络舆情监测，主要是利用互联网信息采集技术，以及自然语言处理等智能信息处理技术，通过对互联网公开数据进行自动化抓取，然后对信息进行结构化、自动分类、文本聚类、主题发现与跟踪等，提供信息检索、多维度统计、敏感信息预警、信息简报、自动化报告等功能，帮助用户及时发现危害品牌形象的观点，并为用户分析关注对象在网络中的形象提供依据。

在舆情 SAAS 系统（以下简称：舆情系统）中，用户设置关注的关键词，就可以快速检索对应的舆情数据，以及对提及关键词的数据提供统计图表，包括舆情走势、词云图、情感分布、情绪走势等。

这里的信息检索与统计，都离不开 Elasticsearch（以下简称：ES）的 Query 以及 aggregation 功能，下面详细介绍如何使用 Elasticsearch 实现这些功能，以及在实践过程中遇到的一些问题及解决方案。

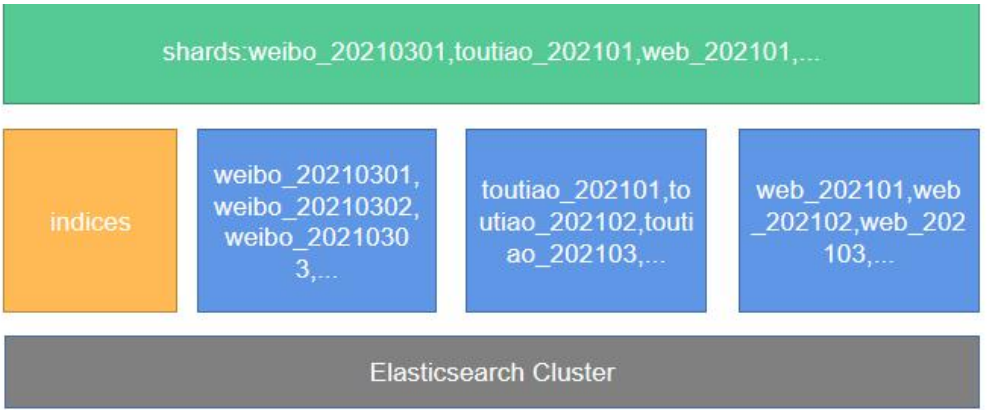
舆情系统多维度检索

索引设计

采集的数据源包括微博、微信、新闻网页、论坛、自媒体平台、短视频等平台的数据，每天新增去重数据量在 1 亿+，每条数据在经过结构化，以及经过 NLP（自然语言处理）之后，超过 150 个字段，比如，文章标题、发布时间、发布作者、发布平台、新闻分类、新闻提及地域、新闻情绪等。由于业务端需要对这些数据进行实时检索，对不同平台的数据实时聚合，各平台的数据量分布也有很大的差异，所以按照平台进行拆分，而不是把所有的数据放到一个大的索引里面。

由于不同平台的数据量差异很大，一般地，微博占每日总采集量的 80%，而新闻网页、微信、自媒体平台的占比相对较少。为了避免由于索引的大小不一样，导致每个 shard 的差异过大，最终导致落在不同节点上 shard 占用空间分布不均匀而出现数据倾斜。

所以，在实现上对微博的索引按照日期做了进一步拆分，微博每日一个索引，而自媒体平台每月一个索引。



索引划分示意图

为了方便业务检索，对按天分索引的微博设置别名，比如 `alias weibo_202101` 对应 weibo_20210101,weibo_20210102,...,weibo_20210131

分词器设计

索引膨胀对比

不同于英文分词器，大部分使用空格作为分隔符。针对不同的检索场景，中文有更多的分词器可供选择，不同中文分词器的选择，会有索引大小，检索性能，以及检索数据的召回率与准确率上的不同。

分词器类型	磁盘占用(GB)	数据条数	单条记录平均大小（KB）
ik(ik_max_word)	52.3	8294805	6.61
standard	55.1	8294805	6.96
ngram-3	110.6	8294805	13.98
ngram-4	153.8	8294805	19.44
ngram-5	204.7	8294805	25.88

不同分词器的磁盘占用对比

在同样的数据条件下，通过对不同分词器下索引占用磁盘空间对比，我们发现：

- IK（ik_max_word）分词器，占用磁盘空间最小。
- Standard 分词器，与 ik_max_word 相差不大，比 ik_max_word 分词方式只增加了 5% 左右。
- N-gram 占用空间比较大，相对于 ik_max_word， 当 n=3，4，5 时，占用空间，分别是 ik_max_word 的 2 倍、3 倍、4 倍 左右。

检索性能对比

	两个字	三个字	四个字
ik(ik_max_word)	91.94ms	91.09ms	110.19ms
standard	159.34ms	237.86ms	268.06ms
ngram-5	26.65ms	31.05ms	35.16ms

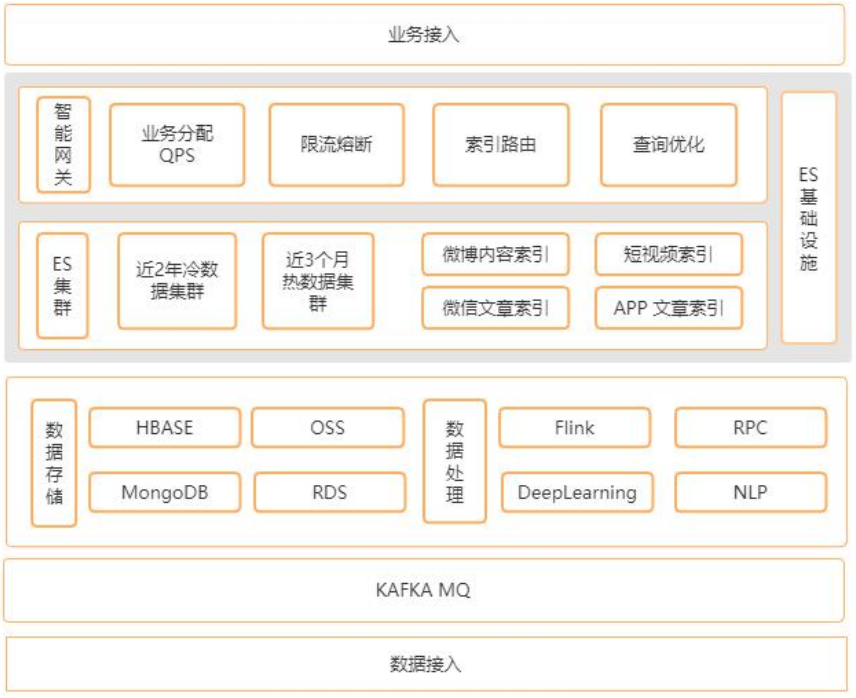
不同分词器的检索耗时对比

在同样的数据条件下，通过对不同的分词下检索性能的对比，我们发现：

- IK（ik_max_word）分词器，检索性能是 standard 分词器的 2-3 倍 左右。
- N-gram（n=5）分词器，检索性能是 standard 分词器的 7 倍 左右，是 ik_max_word 分词器的 3 倍 左右。

通过综合对比，虽然 n-gram（n=5）分词器具备更高的检索性能，但是占用更多磁盘空间，在舆情业务上，索引是百亿级别（保留近 3 个月），基于成本考虑，这里选择了 ik_max_word 分词器。

基于 ES 的数据中台



系统架构图

整个数据中台也是分层的架构体系，分为：

- 数据接入层
- 消息总线
- 数据处理与存储层
- 数据索引层（ES 集群）
- 智能网关层
- 业务接入层

这里重点介绍数据索引层与智能网关层。

数据索引层

在数据索引层，按照业务特点，以及成本综合考虑。分为近 2 年 数据的冷数据集群，以及近 3 个月的热数据集群。

- 冷数据集群，选择价格相对低廉的 SAS 盘作为索引的存储介质， 提供离线的数据下载，以及对响应时间不敏感，且时间周期跨度较长的检索、聚合统计等。
- 热数据集群，选择 SSD 盘作为索引的存储介质，每个节点 16C、64G 内存， 为了降低运维成本，以及动态扩缩容，我们选择了 阿里云 Elasticsearch 服务。

在索引设计方面，根据业务特点，经常要检索特定平台的数据，对索引按照文章发布平台，以及发布日期做了拆分，使每个索引不至于过大，以及导致每个节点上的数据分布不均匀。提高磁盘的利用率与检索性能。

智能网关层

智能网关层避免了业务端直连 ES，无法做到访问并发限制，以细粒度的权限限制。智能网关主要解决了一下几个痛点：

1、并发控制

网关为每个业务分配对应的独立的 TOKEN，并且设置相应的 QPS，防止某个业务的高频访问，影响了其他业务的访问，最终因为级联效应， 导致整个 ES 集群无法提供服务。另外， 网关提供了熔断限流的功能，在 ES 集群负载比较高的况下，对低优先级的 TOKEN 进行限流。

2、权限控制

网关为业务分配对应的权限，比如，读写权限、访问特定索引权限、查询时间跨度权限等，对查询语句进行解析，禁止访问超出权限的数据。

3、SQL 查询

网关提供了 SQL 转换成 DSL 的功能，访问业务端通过标准的 SQL 进行快速的查询对应的数据，提高业务开发效率，降低使用 ES 的门槛。

4、动态路由

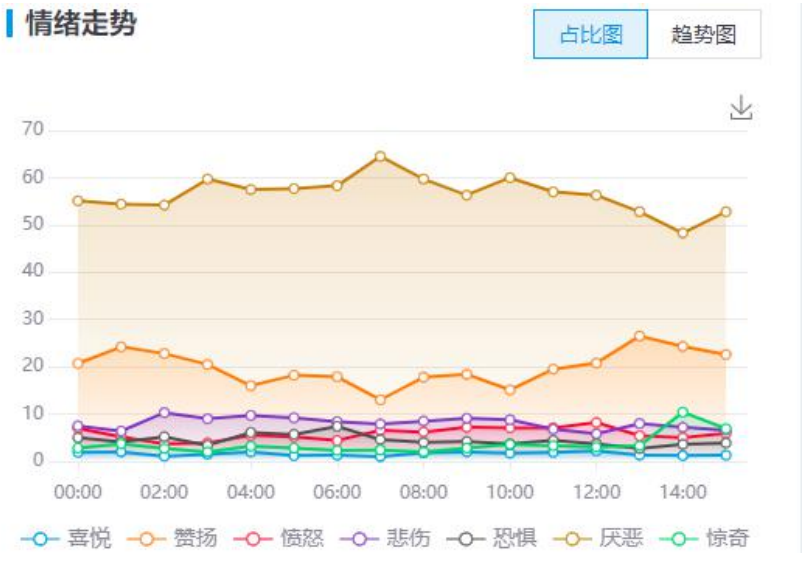
在索引设计阶段，把索引按天、按照发文平台进行了拆分，为了方便业务端查询，根据业务端查询的时间段、发文平台，自动定位到对应的索引，业务端不用关心具体的索引名称，提高业务端的开发效率， 同时，根据查询时间范围，动态查找对应的索引，并在查询时指定到具体的索引，提高查询的速度，避免无效的索引扫描。

5、降低升级成本

由于业务不是直连 ES，后端切换 ES 集群，可做到业务端无感知，降低了 ES 集群的升级切换对业务端的影响， 降低了升级带来的开发成本。

典型业务场景

情绪走势



情绪走势图

通过“情绪走势”图，可以看出一个舆情事件，在一段时间，不同情感表达上的数据分布情况，为了方便使用 ES 的聚合统计功能，对每篇文档的发布时间，设置了 news_posthour（文档发布时间所在的小时）冗余字段，文档的情绪 news_mood_pri 字段，通过在 DSL 中嵌套使用 aggregation ，就可以在一次查询中获得对应的图表数据。

比如查询在微博上与疫情相关的博文在某一天每小时的情绪走势，查询语法如下：

```
GET weibo_2021-04-08/_search{
  "query": {"bool": {"must": [
    {"match_phrase": {
      "news_title": "疫情"
    }}
  ]}},
  "size": 0,
  "aggs": {
    "NAME": {
      "terms": {
        "field": "news_mood_pri",
        "size": 10
      },
      "aggs": {
        "NAME": {
          "terms": {
            "field": "news_posthour",
            "size": 24
          }
        }
      }
    }
  }}
```

通过上面的嵌套查询 DSL 语句，就可以快速获取到不同情绪分类下，每个小时段的数据量。

热门主题词

热门主题词

通过一个事件的热门主题词，可以直观的了解到一个事件的大概内容。这里也是通过 ES 的聚合功能实时获取主题词的统计数据。为了能够实时获取主题词的统计数据，这里用一个事件中提到每个主题词的文档数量，来当作主题词的数量（相当于默认每个主题词在文档中只出现一次），并没有用每篇文档的主题词的绝对量。这样做有一个好处，可以使用 ES 的 aggregation 功能实时聚合获取统计数据，再配合 TF-IDF 算法，计算每个词的相对权重。

在设置索引 schema 时，定义了 news_keywords_list 字段， 用于保存单篇文档的分词结果列表，然后使用如下的语法，就可以快速统计每个词对应的文档数量：

```
GET weibo_2021-04-08/_search{
  "query": {"bool": {"must": [
    {"match_phrase": {
      "news_title": "疫情"
    }}
  ]}},
  "size": 0,
  "aggs": {
    "NAME": {
      "terms": {
        "field": "news_keywords_list",
        "size": 10
      }
    }
  }}
```

通过上面的 DSL 语句，可以快速统计出高频词以及与其相关的文档数量。

创作人简介：

王欢，近 10 年内容大数据领域从业经验，安徽云计算产业促进会开发者工作委员会发起人之一，阿里云 MVP。擅长高并发系统设计、数据中台构建等，目前在一家人工智能企业担任技术 VP，主要关注 AI 算法平台构建、AI 算法在内容分析领域落地等。

博客：https://juejin.cn/user/2981531263175213/posts

4.1.2 实现主流搜索引擎广告置顶显示效果

创作人：铭毅天下

审稿人：李捷

本应用实践，主要针对 Elasticsearch 如何实现类似百度广告置顶显示，给定商品数据的效果展开介绍，例如实现置顶显示某特定数据，像搜索某关键词，出现关联广告置顶显示的效果。

举例：某搜索引擎 “电动汽车”，结果如下：

上面实现的本质：

返回结果的第一页头 1 条或多条数据是服务端（如电商网站、主流搜索引擎）指定的数据，而非按照相关度评分计算得出的结果数据。

这时候，不禁要问 Elasticsearch 能实现类似功能不 ？

拆解实现

Elasticsearch from + size 分页实现机制的原理（大致意思）：

page 1：from 0，size:10——返回第 0 到 第 9 条数据。

page 2：from 10，size:10——返回第 0 到 第 19 条数据，截取第 10 到 第 19 条数据；

page 3：from 20，size:10——返回第 0 到 第 29 条数据，截取 第 20 到 第 29 条数据。

.....

本质是深度分页，肯定越往后翻页响应越慢。

要实现根据固定关键词，添加特定数据置顶显示的效果，探讨方案如下：

方案一：不重新分页，牺牲首页部分数据

不再做重新分页，强制将 page 1 部分数据，换成：类【广告位】置顶显示数据。

显然，会有数据丢失，导致搜索精准率下降，用户一般不会接受。

方案二：重新内存分页

将类【广告位】置顶显示数据 + 已有返回的前 10 页（举例：100 条数据）重新组合后，再分页。

需要内存维护一堆数据，有较大内存开销。用户期望翻页越深（比如：100 页+），维护数据越大，处理越慢、延时会越明显。

方案三：其他方案

类主流搜索引擎实现的方法或者新的实现机制。

但此时要想，有没有更简洁的实现方式呢？

Elastic 官方没有考虑这个用户需求吗？

有的，从 Elasticsearch 7.4.0 新增的 pinned query 就能实现这种功能。

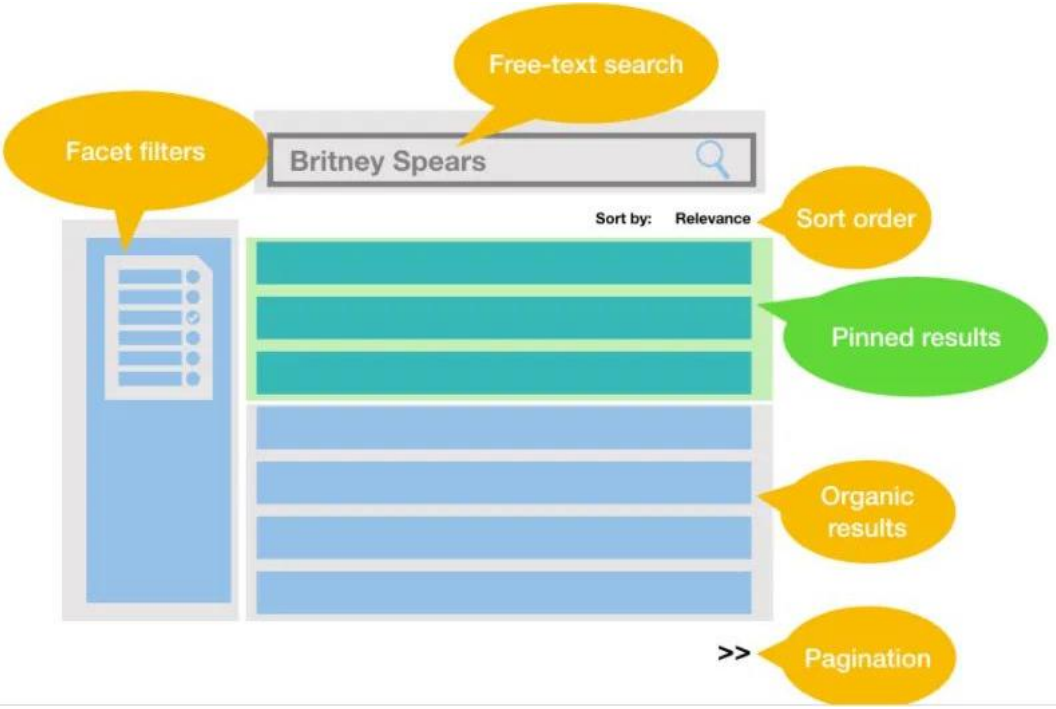
Pinned query 介绍

Pinned query 是 Elasticsearch 7.4.0 版本之后，实现的增强检索功能。

Pinned：中文翻译为“固定”。

Pinned query 则可以解释为——固定某些结果，首页置顶显示的检索方式。

下图能形象的说明：绿色的 Pinned results 就是要首页置顶显示的结果。



Pinned query 实战实现

基础数据 Demo 如下，直接拿文章开头的截图示例模拟一下，假设 id 为 1、2、3 的 3 条数据是需要特意置顶显示的数据：

```
PUT index_001
{
  "mappings": {
    "properties": {
      "title":{
        "type": "text",
        "analyzer": "ik_max_word",
        "fields": {
          "keyword":{
            "type":"keyword"
          }
        }
      }
    }
  }
}



PUT index_001/_bulk
{"index":{"_id":1}}
{"title":"大众汽车首款纯电动 ID.4_预售进行时_先订先享"}
{"index":{"_id":2}}
{"title":"保时捷首款纯电动跑车 Taycan - 现已到店 - 电驰神往"}
{"index":{"_id":3}}
```

```
{"title":"纯电动电动汽车?英国国际贸易部_邀你来投资英国汽车工业"}
{"index":{"_id":4}}
{"title":"四轮电动车_ 电动汽车报价_阿里巴巴采购批发_超多品类低价批发"}
{"index":{"_id":5}}
{"title":"电动汽车之家,为新能源汽车而生 - 第一电动网"}
{"index":{"_id":6}}
{"title":"中国电动汽车网_新能源汽车_电动汽车网"}
{"index":{"_id":7}}
{"title":"电车之家_领先的电动汽车及新能源汽车行业门户网站"}
```

如果要召回既包含：“电动汽车” 完全匹配，又包含“电动”或“汽车” 分词匹配的全量数据。大致的检索语句如下：

```
POST index_001/_search
{
  "query": {
    "bool": {
      "should": [
        {
          "match_phrase": {
            "title": {
              "query": "电动汽车",
              "boost": 5
            }
          }
        },
        {
          "bool": {
```

```
                "should": [
                  {
                    "match": {
                      "title": "电动"
                    }
                  },
                  {
                    "match_phrase": {
                      "title": "汽车"
                    }
                  }
                ],
                "minimum_should_match": 2
              }
            }
          ]
        }
      }
    }
```

如上检索部分：完全匹配加了 boost 提升权重。

返回结果如下：

```
  "_index" : "index_001",
  "_type" : "_doc",
  "_id" : "5",
  "_score" : 3.9220803,
  "_source" : {
    "title" : "电动汽车之家,为新能源汽车而生 - 第一电动网"
  }
},
{
  "_index" : "index_001",
  "_type" : "_doc",
  "_id" : "7",
  "_score" : 3.522773,
  "_source" : {
    "title" : "电车之家_领先的电动汽车及新能源汽车行业门户网站"
  }
},
{
  "_index" : "index_001",
  "_type" : "_doc",
  "_id" : "3",
  "_score" : 3.3853958,
  "_source" : {
    "title" : "纯电动电动汽车?英国国际贸易部_邀您来投资英国汽车工业"
  }
},
{
  "_index" : "index_001",
  "_type" : "_doc",
  "_id" : "6",
  "_score" : 0.422661,
  "_source" : {
    "title" : "中国电动汽车网_新能源汽车_电动汽车网"
  }
},
{
  "_index" : "index_001",
  "_type" : "_doc",
  "_id" : "4",
  "_score" : 0.273311,
  "_source" : {
    "title" : "四轮电动车_ 电动汽车报价_阿里巴巴采购批发_超多品类低价批发"
  }
},
{
  "_index" : "index_001",
  "_type" : "_doc",
```


返回结果按照评分由高到低顺序排列，_id 序列为：5、7、3、6、4 .....

置顶显示_id 为 1、2、3 的数据，pinned query 实现如下：

```
GET index_001/_search
{
  "query": {
    "pinned": {
      "ids": [
        "1",
        "2",
        "3"
      ],
      "organic": {
        "bool": {
          "should": [
            {
              "match_phrase": {
                "title": {
                  "query": "电动汽车",
                  "boost": 5
                }
              }
            },
            {
              "bool": {
                "should": [
                  {
                    "match": {
                      "title": "电动"
                    }
                  },
                  {
```

```
                    "match_phrase": {
                      "title": "汽车"
                    }
                  }
                ],
                "minimum_should_match": 2
              }
            }
          ]
        }
      }
    }
  }
```

本质是在原来检索语句的基础上，添加了如下部分代码：

```
"pinned": {
    "ids": [
      "1",
      "2",
      "3"
    ],
    "organic": {
```

第一：置顶显示的 id ，写法如下：

第二：除了置顶数据之外的其余正常检索语句块内容。只是加了“organic” 包裹一层。其中的检索语句还是原来的写法 ，拷贝过来即可。

返回结果如下：


```
  "value" : 7,
  "relation" : "eq"
},
"max_score" : 1.7014126E38,
"hits" : [
  {
    "_index" : "index_001",
    "_type" : "_doc",
    "_id" : "1",
    "_score" : 1.7014126E38,
    "_source" : {
      "title" : "大众汽车首款纯电动ID.4_预售进行时_先订先享"
    }
  },
  {
    "_index" : "index_001",
    "_type" : "_doc",
    "_id" : "2",
    "_score" : 1.7014124E38,
    "_source" : {
      "title" : "保时捷首款纯电动跑车Taycan - 现已到店 - 电驰神往"
    }
  },
  {
    "_index" : "index_001",
    "_type" : "_doc",
    "_id" : "3",
    "_score" : 1.7014122E38,
    "_source" : {
      "title" : "纯电动电动汽车?英国国际贸易部_邀您来投资英国汽车工业"
    }
  },
  {
    "_index" : "index_001",
    "_type" : "_doc",
    "_id" : "5",
    "_score" : 3.9220803,
    "_source" : {
      "title" : "电动汽车之家,为新能源汽车而生 - 第一电动网"
```


返回结果已 pinned（类似做了“广告位”定制），_id 序列为：1、2、3、5 ……. 实现了类百度置顶显示广告的效果。

Pinned query 源码解读

认知前提：源码中最大评分计算方法

```
float MAX_ORGANIC_SCORE = Float.intBitsToFloat((0xfe << 23)) - 1;
```

本质下面代码等价：

```
float max_rst  = (float)Math.pow(2,127);//1.7014118E38
```

也就是说：MAX_ORGANIC_SCORE 大小为：2 的 127 次幂，是 Elasticsearch float 最大值。

最大评分作用

正常查询的评分得分不会超过 MAX_ORGANIC_SCORE， 将固定查询（pinned query）的评分设定为：MAX_ORGANIC_SCORE。

pinned query 保证置顶显示解密

原理：将置顶显示的数据通过 bool 组合查询 + boost 提升权重的方式给设置了 float 最大值评分，这样就能保证置顶显示了。

核心源码实现如下：

```
@Override
protected Query doToQuery(SearchExecutionContext context) throws IOException {
    MappedFieldType idField = context.getFieldType(IdFieldMapper.NAME);
    if (idField == null) {
        return new MatchNoDocsQuery("No mappings");
    }
    if (this.ids.isEmpty()) {
        return new CappedScoreQuery(organicQuery.toQuery(context), MAX_ORGANIC_SCORE);
    } else {
        BooleanQuery.Builder pinnedQueries = new BooleanQuery.Builder();

        // Ensure each pin order using a Boost query with the relevant boost factor
        int minPin = NumericUtils.floatToSortableInt(MAX_ORGANIC_SCORE) + 1;
        int boostNum = minPin + ids.size();
        float lastScore = Float.MAX_VALUE;
        for (String id : ids) {
            float pinScore = NumericUtils.sortableIntToFloat(boostNum);
            assert pinScore < lastScore;
            lastScore = pinScore;
            boostNum--;
            // Ensure the pin order using a Boost query with the relevant boost factor
            Query idQuery = new BoostQuery(new ConstantScoreQuery(idField.termQuery(id, context)), pinScore);
            pinnedQueries.add(idQuery, BooleanClause.Occur.SHOULD);
        }

        // Score for any pinned query clause should be used, regardless of any organic clause score, to preserve pin order.
        // Use dismax to always take the larger (ie pinned) of the organic vs pinned scores
        List<Query> organicAndPinned = new ArrayList<>();
        organicAndPinned.add(pinnedQueries.build());
        // Cap the scores of the organic query
        organicAndPinned.add(new CappedScoreQuery(organicQuery.toQuery(context), MAX_ORGANIC_SCORE));
        return new DisjunctionMaxQuery(organicAndPinned, 0);
    }
```

注意细节没有深究，比如：置顶返回的结果显示的是原始评分。

小结

读者可能会问：这并没有实现基于特定关键词返回特定数据的需求？其实有了 pinned query 再将特定关键词与待置顶显示文章 _id ，建立个一对多的映射关系就可以实现。映射关系可以自己内存维护或者借助 redis 实现都可以。

你我发现的新需求，很可能别人早就发现，且已经提交 Git 了。更可怕的是：官方新版本已经实现了！

要注重基础夯实的同时，多关注一下技术动态。两手抓、两手都要硬！

参考链接：

- https://www.elastic.co/guide/en/elasticsearch/reference/7.4/release-notes-7.4.0.html

创作人简介：

铭毅天下，Elastic 认证工程师、Elastic 官方合作培训讲师、阿里云 MVP、CSDN 博客专家、铭毅天下 Elasticsearch 公众号作者、死磕 Elasticsearch 知识星球星主。近 10 年工作经验，关注 Elastic Stack 技术栈、大数据技术领域。

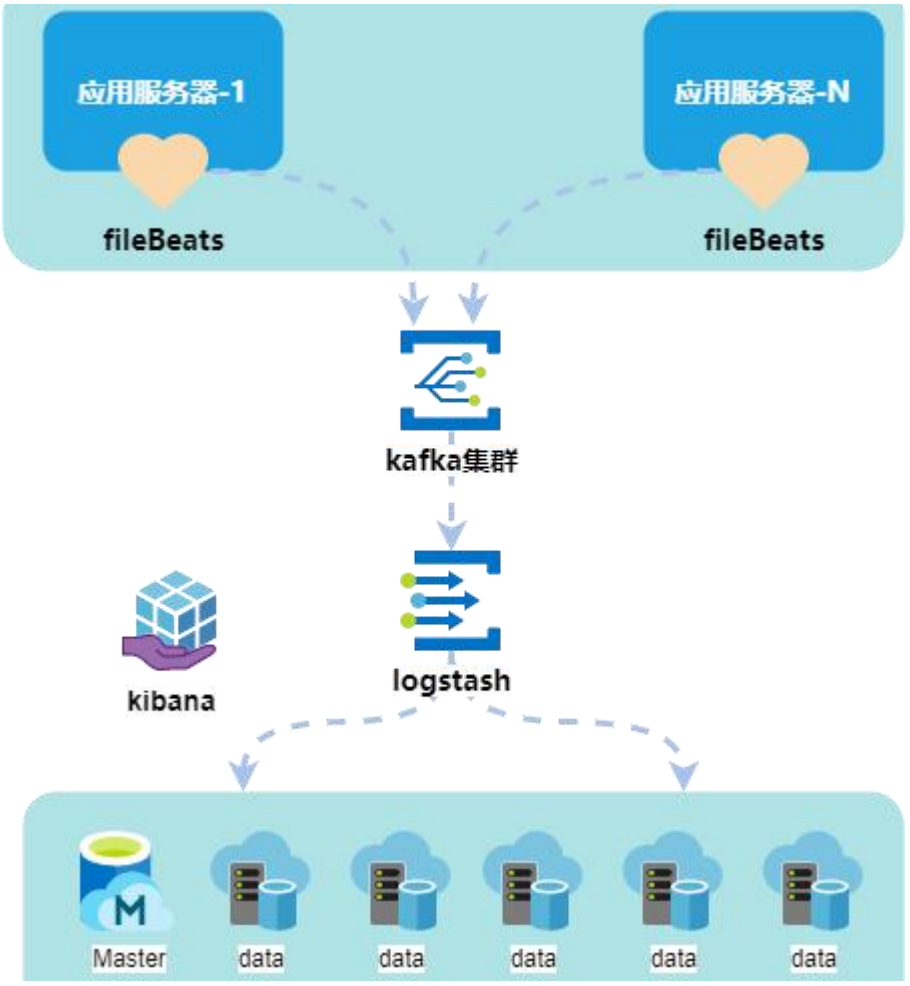
博客：https://elastic.blog.csdn.net/

4.1.3 企业 ELK 日志搜索引擎

创作人：朱祝元

创作人：朱永生

技术架构



- 物理部署：

  1master；5 Data；1 Logstash+kibana；3 kafka 3 主 3 从交叉部署

- 应用框架：

  项目采用 springboot 作为基础框架开发分布式应用；

- 实施方案：

  通过每个应用服务器上部署 filebeat，上传到 kafka；由 kafka 分发消息到 logstash； Logstatsh 写入日志到 Elasticsearch 集群；

- 应用目标：

  收集 50 台机器的日志，可以及时发现日志中的错误日志以及日志对应的上下文。

日志解决方案的演进

阶段一、项目上线一切刚开始

每个程序员通过 ssh 将数据 copy 到堡垒机。然后把数据从堡垒机下载到本地处理数据，分析日志；

遇到的问题：

- 下载日志到本地，文件太大难以处理：每个日志文件大概 500M，这种体量，Windows 上任何文本工具打开都很吃力，还要下载多个文件，下载速率也有很大影响；
- 远程服务器上查找，服务器关联多：同一个服务部署的有多个节点，那么找一个需要的日志就要多个服务器都执行类似于下面的命令来查找蛛丝马迹：

```
more INFO-2020-12-17.0.log |grep -C 5 'scanRecord'
```

如果遇到关联的服务日志查询，还会让事情的复杂度变的更高。

阶段二、测试环境建立 ELK 环境

实践过程：

刚开始的时候 1 master+ 3 data；有一个普遍的认知就是，单个 Elasticsearch data 节点的每个分片数据大小：30GB-50GB。因为我们的系统是 4 核 8G 的配置，因此我们采用了下限，也就是每个 Shard 30G。这样子运行了 3 个月。

采用策略：

按天产生 index，一些 IP，APP 应用名等不需要分词查询的字段都禁用了 index (这样可以节省磁盘)，只保留一周的日志回溯，3 天的日志 alive 查询，4 天的日志 close。一周以上的 index 直接 delete ，晚上 12 点 定时执行 forcemerge。

遇到瓶颈，系统扩容：

因为随着系统票件量的提升，日志数据逐步增加。慢慢就会感到系统查询非常慢，磁盘空间慢慢的无法做到保留一周日志回溯，立马进行了系统扩容。

扩容后：

系统会自动进行索引分片重分配，会把分片均匀的分布到所有的节点上。比如刚开始 3 台 data 节点 6 个分片，平均每个机器会有 2 个分片，那么系统扩容一倍后，会变成 6 个 data 节点，那么这 6 个分片，会自动平均分布到 6 个 data 节点上。每个节点有一个 shard。

扩容步骤

修改配置文件：

主要修改所有 Elasticsearch 节点的 elasticsearch.yml 中的 IP 地址，如果一个机器上部署多个节点，记得将端口号加上。

一个机器上部署三个节点实例：

```
discovery.zen.ping.unicast.hosts:["192.168.207.43:9300","192.168.207.43:9301","192.168.207.43:9302"]
```

配合的属性：

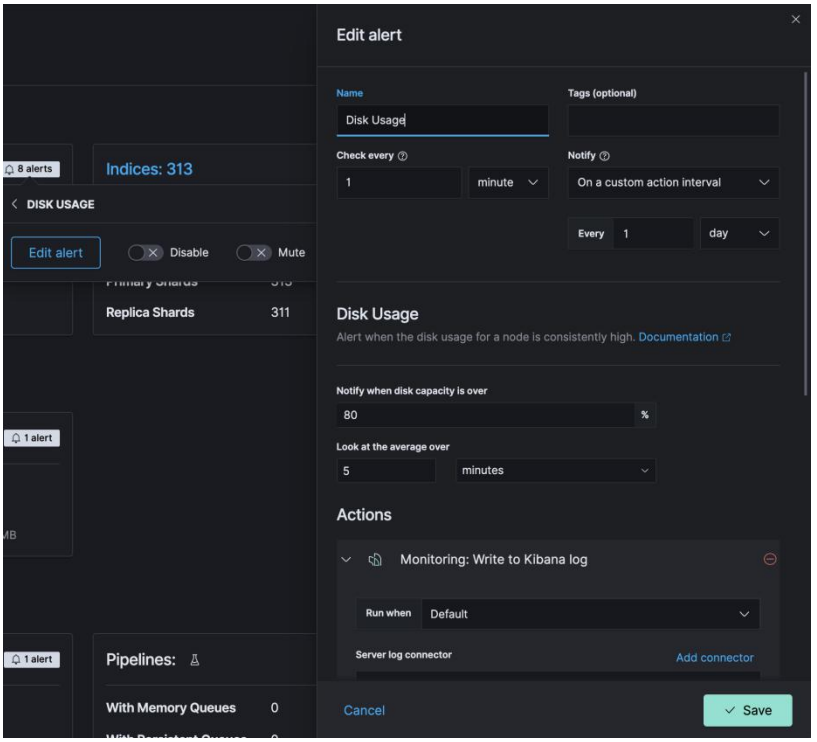
```
http.port: 9202transport.tcp.port: 9302
```

分批启动 ES：

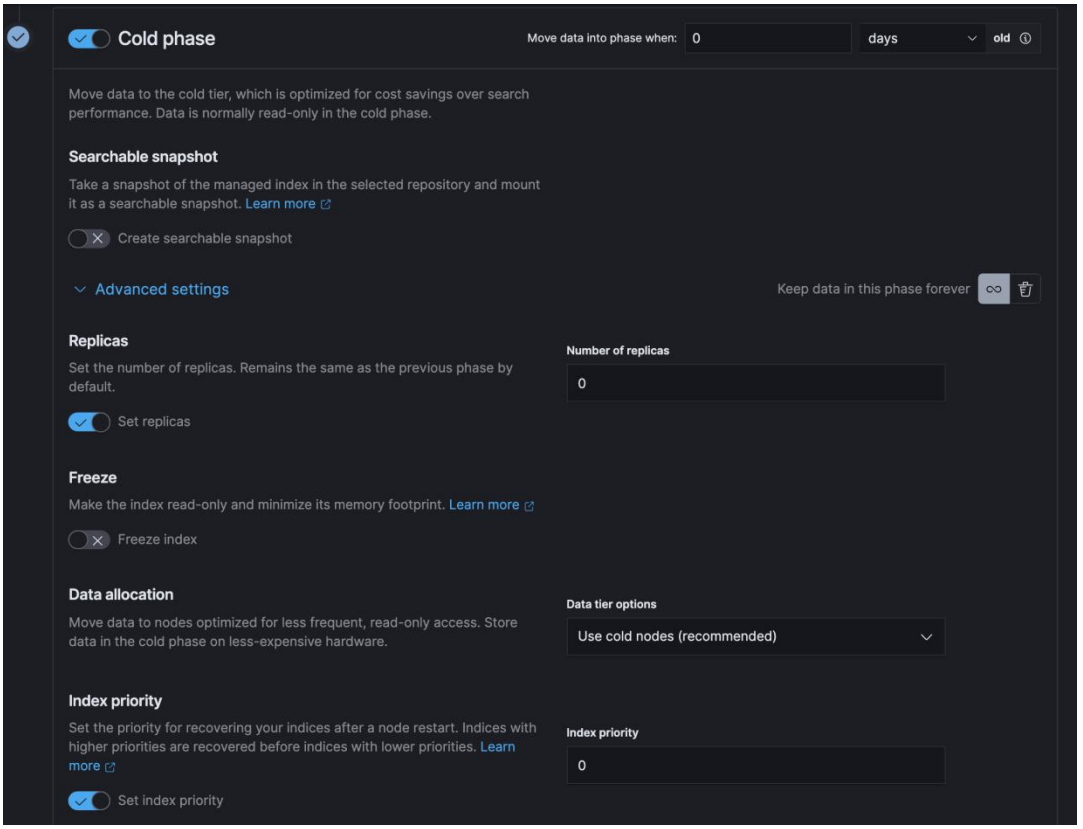
- 启动顺序：先启动 master 节点，再启动其他类型的节点。
- 启动命令：nohup ./bin/elasticsearch > nohup.out 2>&1 &

心路旅程

1、资源并不是充裕的。可以使用 Stack Monitoring 上的磁盘监控功能，随时监控磁盘的剩余空间。



并且，可以在数据可靠性要求允许的情况下，在索引生命周期管理中，把冷数据的 index.number_of_replicas 设置为 0。



2、最佳的 Kafka 分发效率。如果使用了 Kafka，注意 Kafka 的 Partition 与 Topic 的配置关系，通常来说 Logstash 中 Worker 的数量应该等于或大于 Kafka Partition 的数量，以便于达到最优的分发效率

3、SSD 的取舍。数据量过大。磁盘 IO 也真的能成为瓶颈，对比集群没有数据和集群数据量达到磁盘容量的 50%的时候，写入的速率差别很大。业务需求需要实时查询的场景能上 SSD 就上 SSD。

创作人简介：

朱祝元，从事 JAVA 企业级应用开发十余年，获得 pmp，acp 项目管理认证。有扎实的企业级开发经验，以及分布式应用开发架构经验，参与了千万级的复杂项目数据场景业务处理。

4.2 可观测性应用场景

4.2.1 基于 Elasticsearch 实现预测系统

创作人：田雪松

审稿人：李捷

业务背景介绍

当代商业组织面临的最基本挑战，是互联网已经不再是一个替代或可选渠道，它已经成为许多企业最主要的、甚至是惟一的销售平台。网上店面在现实中往往比实体店面还要重要，所以人们就必须要像监视实体店面一样，监控网上应用。

监控系统通常会以推送（Push）或拉取（Pull）的方式，从服务或应用中获取监控指标（Metrics），并在监控指标出现异常时，发送警报或恢复服务，以实现网上店面实时可用的目标。

监控系统在提升了应用可用性同时，还在日积月累中形成海量监控数据，而从这些监控数据中挖掘出巨大的商业价值。监控指标会包含一些特定业务场景下的业务数据，借助大数据分析工具对它们进行处理和建模后，就可以辅助人们作出正确的市场决策。

本章要介绍的案例就是基于 Elasticsearch 对监控数据进行商业挖掘的一次尝试，我们内部称这套系统为 Prophet（大预言家）。Prophet 通过机器学习对历史监控数据

进行处理，并创建预测模型，最终可以实现对单次用户请求处理的全面预测，包括处理结果是否成功、处理执行时长等等。

Prophet 预测系统架构

Prophet 虽然只是监控数据最终的分析处理服务，但支撑 Prophet 实现智能预测的整个系统却涵盖了四个主要部分：

- 第一部分是产生海量监控数据的监控系统，它从被监控的业务系统中拉取监控数据，并提供实时告警服务。
- 第二部分则是将监控数据，从监控系统中导入到 Elasticsearch 集群的数据系统，它还负责对监控数据进行清洗、转换；
- 第三部分是用于存储历史监控数据的 Elasticsearch 集群，我们借助 Elasticsearch 冷热模型，延长了监控数据的保存时间；
- 而最后一个才是最终处理这些数据的 Prophet 服务，它通过 Spark 运算集群，实现智能预测功能，未来还会添加更多数据分析功能。

四个系统之间的数据流动关系如图 1-1 所示：

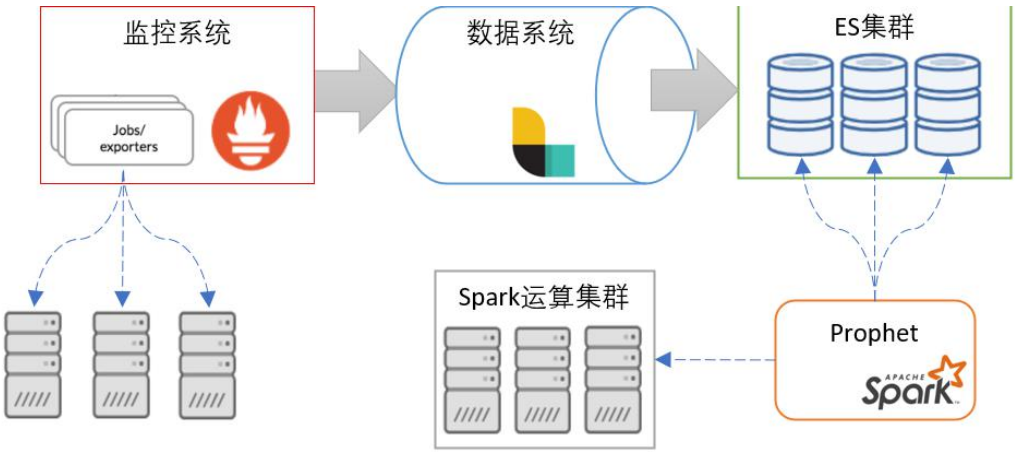


图 1-1 Prophet 周边生态

除了 Elasticsearch 以外，HDFS 也是存储历史监控数据的一种不错选项，并且 HDFS 更容易与 Hadoop 的大数据生态系统集成。

我们之所以采用 Elasticsearch 主要还是被 Elasticsearch 强大的检索能力所吸引。同时，Elastic Stack 家族中的 Kibana 还提供了强大的数据可视化能力，至于数据分析则可以利用 elasticsearch-hadoop 插件整合 Hadoop 生态。

使用 Elasticsearch 存储监控数据

在 Prophet 诞生之前，我们的业务系统已经使用 Prometheus 做到了实时监控，同时还通过 Grafana 对监控数据做了可视化处理。

但 Prometheus 监控数据存储，在其内置 TSDB（Time Serie Database）中，Prometheus TSDB 中并没有分片和副本等概念，而只是简单地将数据保存在本地硬盘

上。这意味着 Prometheus 无法支持数据高可用，同时也意味着 Prometheus TSDB 中的数据不能保存太久，否则本地硬盘迟早会被积累的数据撑爆。所以 Prometheus 默认只保存监控数据 15 天，超过这个时间后监控数据就会被直接删除。

对于一个监控系统来说这并不算什么大的问题，因为监控系统通常对实时数据更感兴趣，它只要能够实时快速地反映，被监控系统的异常就足够了。而历史数据分析和处理，并不能算是监控系统的职责范畴，应该交由 Elasticsearch 这样更专业的数据检索与分析工具来实现。

从另一个角度来说，由于预测系统，必须要基于历史数据创建预测模型，而如果直接在 Prometheus TSDB 上进行高频度的模型运算，则有可能会对监控系统本身造成性能上的影响。从系统监控与数据分析的职责角度来看，监控系统的稳定性无疑比数据分析更为重要。所以预测系统使用的历史数据，应该与 Prometheus 隔离开来，这就要求监控数据得从 Prometheus 中同步到 Elasticsearch 。一旦数据从 Prometheus 进入到 Elasticsearch ，我们就可以利用 Elasticsearch 冷热架构对数据进行优化以保存更长时间。

Prometheus 与 Elasticsearch 的数据同步

Prometheus 支持一种称为远程读写（Remote Read/Write）的功能，可以在 Prometheus 拉取监控数据时将数据写入到远程的数据源中，或者从远程数据源中读取监控数据做处理。

Prometheus 目前支持的远程数据源多达几十种，所有数据源都支持远程写入，但

并不是所有数据源都同时支持远程读取，比如我们这里要使用的 Elasticsearch 就只支持远程写入。

注：7.10 版本是支持的，可以通过 metricbeat 对 Prometheus 进行远程拉取，参见：[__https://www.elastic.co/guide/en/beats/metricbeat/current/metricbeat-metricset-prometheus-query.html__]

Prometheus 为远程读取和写入，分别定义了独立的通信与编码协议，第三方存储组件需要向 Prometheus 提供兼容该协议的 HTTP 接口地址。但由于众多第三方存储组件接收数据的方式并不相同，支持 Prometheus 读写协议的接口地址并不一定存在。为此，Prometheus 采用了适配器的方式屏蔽不同存储组件之间的差异，

如图 1-2 所示：

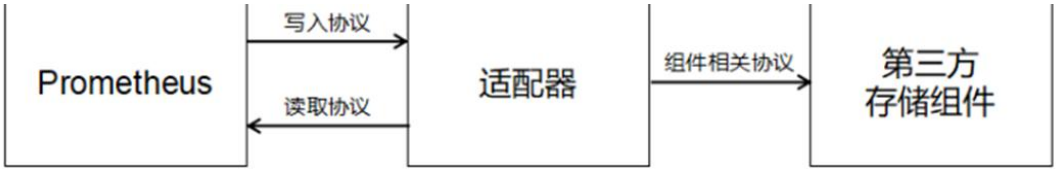


图 1-2 Prometheus 远程读写

只要提供了适配器，Prometheus 读写数据就可以转换成第三方存储组件支持的协议和编码。Elasticsearch 也需要使用适配器，这个适配器就是由 Elastic Stack 家族中的 Beat 组件担任。早期 Prometheus 官方推荐的适配器是 Infonova 开发的 Prometheusbeat，而在最新版本中已经转而推荐 Elastic Stack 官方的 Metricbeat。

无论是 Prometheusbeat 还是 Metricbeat，它们都可以向 Prometheus 开放 HTTP 监听地址，并且在接收到 Prometheus 监控数据后将它们存储到 Elasticsearch 中。Metricbeat 采用 Module 的形式开启面向 Prometheus 的 HTTP 接口，具体配置如示例 1.1 所示：

```
- module: prometheus
  metricsets: ["remote_write"]
  host: "localhost"
  port: "9201"
```

示例 1.1 Metricbeat 配置

添加示例所示配置内容后重启 Metricbeat，它就会在本机 9201 端口开始监听 Prometheus 的写入数据。与此同时还要将这个监听地址告诉 Prometheus，这样它才知道在拉取到数据后向哪里写入。具体来说就是在 prometheus.yml 中 添加如下内容：

```
remote_write:
  - url: "https://localhost:9201/write"
```

示例 1.2 Prometheus 配置

使用 MetricBeat 组件保存监控数据，监控指标的标签（Label）会转换为 Elasticsearch 的文档字段。Metricbeat 还会附加一些 Metricbeat 自身相关的数据，所以文档中的字段会比实际监控指标的标签要多一些。

除了指标名称和标签以外，还包括主机地址、操作系统等信息，这可能会导致样本数据量急剧膨胀。但 Prometheus 并不支持在写入远程存储前过滤样本，所以只能通过

Beat 组件处理。

[注：最高效的做法是在 metricbeat 上设置 drop_fields 的 processor，直接过滤，避免无效的网络传输]

比较理想的方法是将 Beat 组件的输出设置为 Logstash，然后再在 Logstash 中做数据过滤，最后再由 Logstash 转存到 Elasticsearch 中。

在我们的系统中，除了要过滤以上 Metricbeat 附加的字段以外，还需要要组合、修正一些业务相关的字段。所以为了更好的处理数据，我们在整个数据系统中加入了 ogstash 组件.

最终整个数据系统的大致结构如图 1-3 所示：

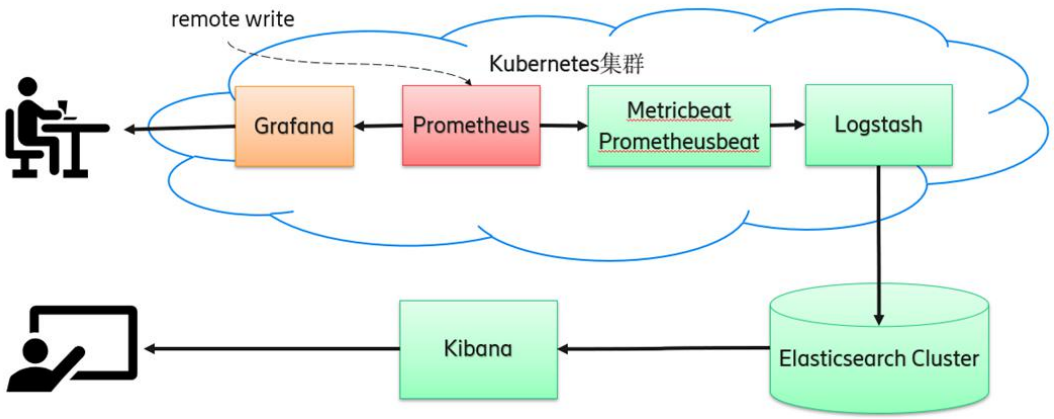


图 1-3 数据系统

Prometheus 在拉取到监控指标时，将数据同时发送给 Metricbeat 或 Prometheusbeat，它们在这里职责就是一个监控数据接收的端点。

Metricbeat 或 Prometheusbeat 再将数据发送给 Logstash，而 Logstash 则负责清洗和整理监控数据，并将它们存储到 Elasticsearch 中。我们将整个监控系统和数据系统部署在 Kubernetes 集群中，而最终用于机器学习的数据源 Elasticsearch 则被部署为独立的集群。

这样就将监控系统与数据分析系统隔离开来，从而保证了监控系统，不会受到大量数据分析运算的影响而出现故障。实际上图 1-3 就是对图 1-1 中 所展示的监控系统、数据系统以及 Elasticsearch 集群的细化，但在实际应用中还有许多细节没有展现出来，需要读者根据项目实际情况做适当调整。

集成 Elasticsearch 与 Spark

事实上，新版 Kibana 组件中已经具备了非常强大的机器学习能力，但这项功能还未开放在基础授权中。此外我们的业务系统还有一些自身的特殊性，所以最终我们决定采用 Spark 编写代码实现这套智能预测系统。

一种显而易见的办法是利用 Elasticsearch 客户端接口，先将数据从 Elasticsearch 中读取出来再传给 Spark 进行分析处理。但 Spark 数据处理是先将任务分解成子任务，再将它们分发到不同计算节点上并行处理，以此提升海量数据分析处理的速度。而任务分解的同时是数据也要分解，否则最终数据读取，就会成为所有子任务的瓶颈，这种分解后的数据在 Spark 中称为分区（Partition）。每个任务在各自的数据分区上运算

处理，最终再将各自的处理结果按 Map/Reduce 的思想合并起来。

Elasticsearch 中的分片（Shard）刚好与 Spark 分区相契合，如果能让任务运算于分片之上，这无疑可以更充分地利用 Elasticsearch 存储特征。但如果是先从 Elasticsearch 读取数据再发送给 Spark 做处理，数据就是经由分片合并后的数据，Spark 在运算时就需要对数据再次进行分区。

Elasticsearch 包含一个 elasticsearch-hadoop 项目，可以很好地解决数据分区问题。elasticsearch-hadoop 可以与包括 Spark 在内的 Hadoop 生态良好集成，Spark 分区数据也可以与 Elasticsearch 分片数据直接对应起来。我们所要做只是将 elasticsearch-hadoop 的 Spark 依赖添加到项目中，并使用其提供的接口将数据从 Elasticsearch 中读取进来即可。智能预测系统采用 Java 语言开发，所以使用 Maven 引入如下依赖：

```
<dependency>
    <groupId>org.elasticsearch</groupId>
    <artifactId>elasticsearch-spark-${spark.major.version}_${scala.version}</artifactId>
    <version>${elasticsearch.version}</version>
</dependency>
<dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-sql_${scala.version}</artifactId>
    <version>${spark.version}</version>
</dependency>
<dependency>
    <groupId>org.apache.spark</groupId>
```

```
        <artifactId>spark-mllib_${scala.version}</artifactId>
        <version>${spark.version}</version>
    </dependency>
```

示例 1.3 引入依赖

如示例 1.3 所示，elasticsearch-hadoop 的 Spark 依赖为 elasticsearch-spark。由于需要用到 Spark ML 相关库，示例也将 spark-sql 和 spark-mllib 依赖也一并引入了进来。elasticsearch-spark 在 org.elasticsearch.spark.rdd.api.java 包中定义了类 JavaEsSpark，它提供了大量用于读取和写入 Elasticsearch 的静态方法。其中，静态方法 esRDD 用于从 Elasticsearch 中读取索引，而 saveToEs 则用于向 Elasticsearch 中存储数据。

示例展示了从 Elasticsearch 中读取索引数据的代码片段：

```
List<Row> jobs = esRDD(sparkContext, metricIndex, queryString).values().map(
                    map -> {
                        Object[] values = new Object[featureFields.length+1];
                        for (int i = 0; i < featureFields.length; i++) {
                            values[i] = map.get(featureFields[i]);
                        }
                        values[featureFields.length] = map.get(labelField);
                        return RowFactory.create(values);
                    }
            ).collect();

            StructField[] structFields = new StructField[featureFields.length + 1];
            for (int i = 0; i < featureFields.length; i++) {
```

```
                structFields[i] = new StructField(featureFields[i], DataTypes.StringType, true,
Metadata.empty());
            }
            structFields[featureFields.length] = new StructField(labelField, DataTypes.StringTy
pe, true, Metadata.empty());

            StructType schema = new StructType(structFields);
            Dataset<Row> dataset = sparkSession.createDataFrame(jobs, schema);
```

示例 1.4 使用 Spark 读入数据

示例中，sparkContext 是 SparkContext 实例，代表了与 Spark 运算环境相关的一些基本信息。MetricIndex 则是 esRDD 方法要读取文档数据的索引名称，而 queryString 则是检索文档的查询条件。esRDD 返回的类型为 JavaPairRDD，这是类似于一个 Pair 的集合。

每一个 Pair 代表索引中的一个文档，Pair 的 key 是字符串类型，代表了当前文档的标识符；而 value 则是一个 Map 类型，代表了整个文档数据。通过 JavaPairRDD 的 keys 方法将只返回所有文档的标识符，而通过 values 方法则会只返回所有文档的 Map 对象。

Map 的键为文档字段名称，而值则为文档字段对应的具体数值。示例中的代码就是通过调用 values 方法只获取文档的 Map 对象，然后再通过 map 和 collect 方法将它们转换成 Row 的集合。

分类与回归

机器学习本质上是基于统计学原理，构建数学模型的过程，而构建出来的模型又可以用于数据分析与数据预测。机器学习一般分为监督学习和无监督学习两大类，它们分别针对有标注（Label）数据和无标注数据构建数学模型。

标注可以认为是对数据在某一维度上的识别，比如让计算机识别声音或图像中的文字，就需要先对它们做标注，然后才可以根据标注对声音或图像构建数学模型。具体来说，人们先要提供一组与文字关联好的音频或图片，计算机再根据这些标注好的数据，找出从声音频率、图片像素到文字的映射关系。由于不同人对文字的发音和书写存在着巨大的差异，所以需要通过海量音频和图像进行统计分析，找出这种映射关系中最为本质的转换模型。所以监督学习的本质是学习输入到输出的统计学规律，只不过这些映射关系并不像加减法那样直观，需要借助计算机统计与分析才能完成。

无监督学习则是根据无标注数据进行数学建模的过程，它不存在输入与输出，本质上是学习数据中天然的统计规律或潜在结构。比如用户的消费行为就很难根据他们的年龄、性别、学历等特征做区分，而且由于涉及隐私这些数据也不一定会在系统中查到。换句话说，即使两个人的硬性特征完全相同，但在人生经历、思想认识等方面存在着的差异，依然可能使他们的消费行为不同。所以这时就只能通过用户消费行为产生的数据对他们进行聚类，相同聚类的用户往往有着相类似的消费习惯，聚类结果也就可以应用于商品推荐系统之中了。

本章介绍的智能预测系统是一种通过监督学习，构建预测模型的机器学习系统，本质上就是发现已知条件与未知结果之间的统计规律。我们之所以要建立这样一个系统，是因为在以往对业务系统日志的统计分析中发现，在某些特定条件下某些用户请求一定会失败。这让我们坚信在已知请求条件与未知运行结果之间，一定存在着某种映射关系，

我们希望借由这样一个系统找到这种神秘的映射关系。

在监控系统的数据中原本就包含了请求执行结果，这个数据就可以作为整个请求与处理相关数据的标注。在监督学习中，作为输入的数据一般称为特征（Feature），而输出的结果则一般称为标注（Label）。在示例中定义的 featureFields 定义的就是索引上那些可以作为特征的字段，而 labelField 则是索引上可以作为标注的字段。示例中的代码就是将特征字段的值和标注字段的值从 Map 中取出，并将它们统一放置到一个数组中并转换为 Spark 运算需要使用的 DataFrame。

正如前面所介绍的那样，监督学习的本质是学习输入到输出的统计学规律。如果从输入到输出的产出结果是有限的，那么这时机器学习生成的模型称为分类器（Classifier），而它解决的问题就是分类问题。比如我们正在介绍智能预测系统，它通过已知的请求条件预测执行结果，输入是在请求执行前已知的请求条件，而输出则是该请求的执行结果。

执行结果只有两种结果，要么成功要么失败，所以计算机学习出来的数学模型就是一个分类器。除了分类问题以外，监督学习可以解决的另一种类问题是回归问题。与分类问题不同，回归问题输出的不是有限数量的结果，而是数值连续的结果，它更像我们在数学中学习到常规函数。

同样是以请求条件作为输入，如果预测的不是执行结果而是执行时间，那么这时要解决的就不再是分类问题而是回归问题。因为请求的执行时间并不是离散的，它可能是任意一个非负的数值。

Spark 机器学习支持多种实现分类与回归问题的算法，包括逻辑回归、决策树、随机森林、梯度提升树等等。我们很难在一个章节中将上述算法完全讲解清楚，但所幸应用这些算法做预测也并不需要深入了解算法原理。本章仅以梯度提升树为例解决分类问题，也就是通过梯度提升树实现对请求执行结果的预测，读者可根据本章所列方法实现对请求执行时间的预测。

Spark 在实现机器学习时一般会将数据分成两组，一组用于构建数学模型，而另一组则用于验证已构建的模型。

示例中的代码就是将示例中生成的数据按 8：2 分成两组，8 成的数据用于生成模型，而 2 成的数据用于验证模型：

```
        Dataset<Row>[] splits = featurized.randomSplit(new double[]{0.8, 0.2});
            GBTClassifier gbt = new GBTClassifier()
                    .setLabelCol(LABEL_COL)
                    .setFeaturesCol(FEATURES_COL)
                    .setMaxIter(10);
            GBTClassificationModel model = gbt.fit(splits[0]);

            Dataset<Row> predictions = model.transform(splits[1]);
            MulticlassClassificationEvaluator evaluator = new MulticlassClassificationEvaluator
()
                    .setLabelCol(LABEL_COL)
                    .setPredictionCol(PREDICTION_COL)
                    .setMetricName("accuracy");
            double accuracy = evaluator.evaluate(predictions);
```

```
log.info("Test Error = " + (1.0 - accuracy));
model.write().overwrite().save(path);
```

示例 1.5 生成预测模型

如示例所示，GBTClassifier 就是基于 GBT 算法的分类器，通过它可以生成基于 GBT 算法的分类模型。setLabelCol 和 setFeaturesCol 分别设置了数据的标注和特征，而 setMaxIter 则设置了生成模型时迭代的次数。有了 GBT 分类器后，调用其 fit 方法并将 8 成的数据 splits[0] 传入就会触发模型创建的运算。

模型生成的运算通常比较慢，具体时间长度取决于参与运算的数据量大小，但一般都是分钟级别的。由于模型一旦生成往往可以复用，所以在数据变化不是特别剧烈的情况下，可以将模型保存到硬盘。这样在需要预测时就不用再通过匹配数据生成模型，通过硬盘中保存的现成模型数据就可迅速生成模型。所以在我们的实现中，会每天定时更新一次模型到硬盘，但在实例应用模型时，都是直接从硬盘中加载模型以节省运算时间。

示例展示了从硬盘中加载模型，并根据输入的特征向量进行预测的代码片段：

```
GBTClassificationModel model = GBTClassificationModel.load(path);
Dataset<Row> predictions = model.transform(featurized);
predictions.show(false);
```

示例 1.6 实现预测

示例中使用的 tranform 方法在这里就是最终用于预测的方法，传入的 featurized 则是经过向量化的已知数据。

当然，以上示例代码中省略大量的代码细节，在实际应用中还有许多更为具体的问题要解决。根据我们应用的效果来看，预测结果的准确率可以达到 95% 以上。

回归类问题的处理过程与本章所介绍内容类似，感兴趣的读者可根据本章内容尝试实现对请求处理时间的预测。

4.2.2 ES 智能巡检开发设计实践

创作人：张妙成

审稿人：田雪松

项目背景

PaaS 下管理了大量集群，监控和告警能快速的让开发维护人员，知道系统已经发生故障，并且辅助高效排障。

但是无法提前预知集群的健康状况，开发人员和维护人员均无法在故障前及时作出调整。为了帮助用户及时的知道集群的健康状态，更好使用 Elasticsearch 集群，可以定期对集群进行指标检查并给出相应报告。巡检作业及时发现集群的健康问题，集群的配置是否合理，提前主动发现问题，能有效保证集群的稳定性、健状性，从而减少业务中断时间保证服务质量。

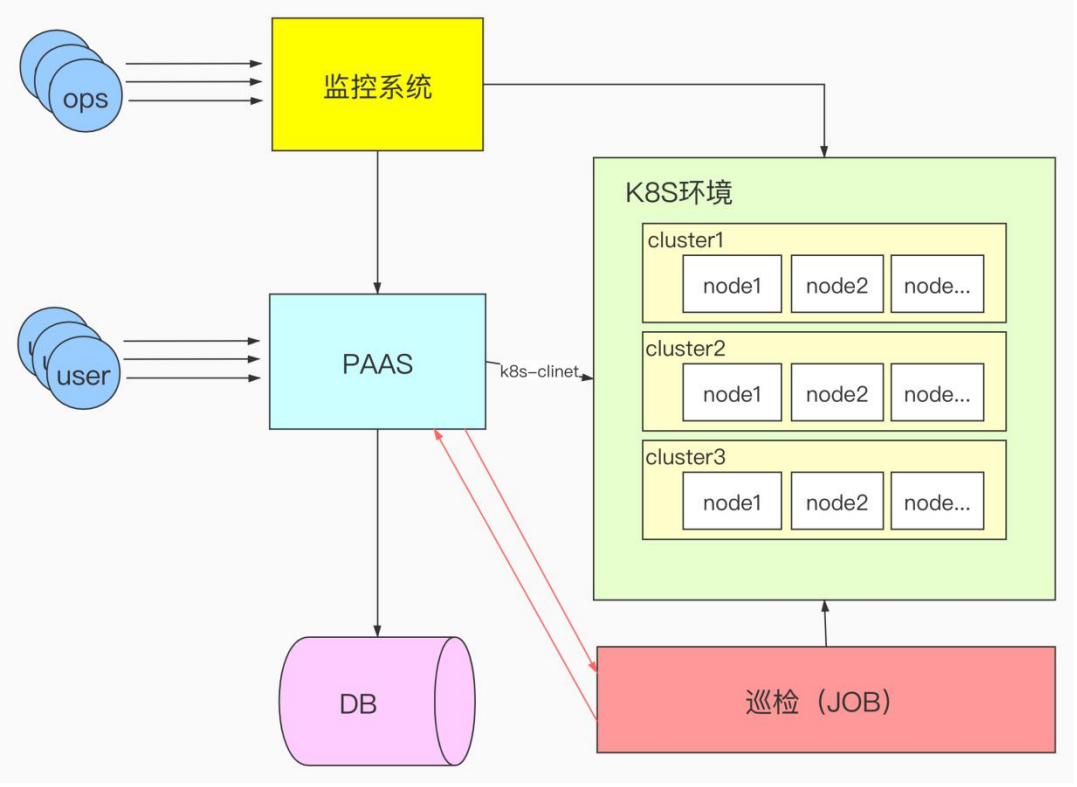
为了解决集群健康状态提前预知困难的问题，可以通过抽取一些指标，进行定时检查达到健康诊断的目的。

巡检主要是对集群的各个指标检查，给出一份全方位的报告，并提供一定的推荐解决、优化方案。如阿里的 EYOU 平台（阿里云 Elasticsearch 智能诊断系统）会系统的在 Elasticsearch 公有云进行各个指标的检查，并给出相应报告，极大的减小了风险，降低了维护成本。

智能管理系统不是一个独立的检查系统，而是一个与其他系统相结合的闭环系统，独立的巡检模块对各项指标进行检查分析，将结果通过 PaaS 系统展示给用户，并在 PaaS 中给予入口，用以帮助用户手动再次触发检查，增强实时性，提高用户体验。

本文将介绍智能巡检系统在整个 Elasticsearch 相关系统中的位置与意义，并从指标分析选取、异常标准的角度，主要阐述智能巡检系统的设计与实现。

巡检系统的结构



整个应用的框架如上：

- Elasticsearch 集群在 K8S 环境中（实际生产大多是 K8S 环境与物理机、虚拟机环境共存，这里简化成最终要达到的统一环境），由 PaaS 平台进行统一管理。
- PaaS 的信息数据主要是与 DB 交互（PaaS 是与 DB 的唯一交互入口），用户主要与 PaaS 平台进行交互。
- 智能巡检系统信息收集模块（一组 Python Job）主要是 K8S 环境中的 Elasticsearch cluster、宿主机进行交互，数据报告信息通过 PaaS 平台存入 DB。
- 监控使用 VictoriaMetrics（ Prometheus 的高可用方案）作为存储，grafana 作为前端展示页面。监控可以配置 Elasticsearch 各项指标，其中与智能巡检相关是巡检异常数量的监控面板，用来给 OPS 观察巡检亚健康集群异常点的修复（优化）情况。
- PaaS 提供入口手动触发再次检查。

指标选取简介

巡检的指标、异常阈值与告警配置的主要区别是，检的指标项会更加关心可能引发故障的某些现象和配置，参考阈值相对告警配置会更加宽松。巡检主要是通过指标的采集分析，得出一份相对全面的报告和推荐解决方案。为了报告的全面性与分析的准确性，巡检的指标项会与告警配置有一定相似或重复。

告警与巡检需要解决的问题不同，告警的目的是将异常指标恢复到正常状态，响应的实时性要求较高，而巡检的目的是预防故障、消除隐患、优化集群性能，以报告的形式推到平台和用户，不需要用户主动响应，只需解决问题后重新触发巡检。为推进优化，可将巡检报告中非健康指标配置成监控面板、告警。

集群健康程度可以从几个方面表现：cluster 层面、node 层面、shard 层面、index 层面、jvm 层面、threadpool 层面。如下为参考指标：

模块	指标项	异常阈值（参考）
cluster 层面	1.cluster status 2.pengding_task3.cpu_util（极差）4.query（极差）5.一次 bulk 请求的数量	1.health 为 red、yellow2. pending_task 数量 count > 100 3.cpu_util(max) - cpu_util(min) > 50%4.query(max)/query(min) > 25.indexing_total(stacked) /s > 1000 且 indexing_total / thread_pool_write_completed < 100
node 层面	1.uptime2.free_disk3.cpu_util 4.node 上 shard 数量	1.date_now - uptime < 1h2.free_disk < 30%3.cpu_util > 90%4.count(shard) = 0
shard 层面	1.number2.size of per shard	1.每 GB 的 heap 超过 20 个 shards2.搜索类集群（tag）单个 shard_size > 20g，日志类集群单个 shard_size > 50g
index 层面	1.replica2.dynamic mapping3.refresh_interval4.indices.refresh.total5.max_result_window	1.存在 index 无 replica2.dynamic != false && dynamic != strict3.refresh_interval = -14.refresh_count > 80/min5.max_result_window > 10000
jvm 层面	1.jvm heap 使用率 2.jdk version 一致性 3.heap segment memory4.Full GC	1.heap_util > 90%2.各个节点 jdk 版本出现不一致 3.segment memory 占用 heap > heap_size 的 20% 4.出现 full gc

模块	指标项	异常阈值（参考）
thread pool 层面	1.bulk reject 数量 2.search reject 数量	1.bulk rejected>0 2.search rejected>0

Elasticsearch 功能强大、使用方便，也就意味着对用户来说有很多的默认设置，用户使用的自由度很高，也就意味着开放的能力丰富，用户的使用对集群健康程度有着很大的影响。

所以指标选取需要从两个角度，一是现有的现象指标，二是常用不合理的配置指标。接下来对选取的指标进行简要逐一分析。

指标分析

cluster 层面指标分析

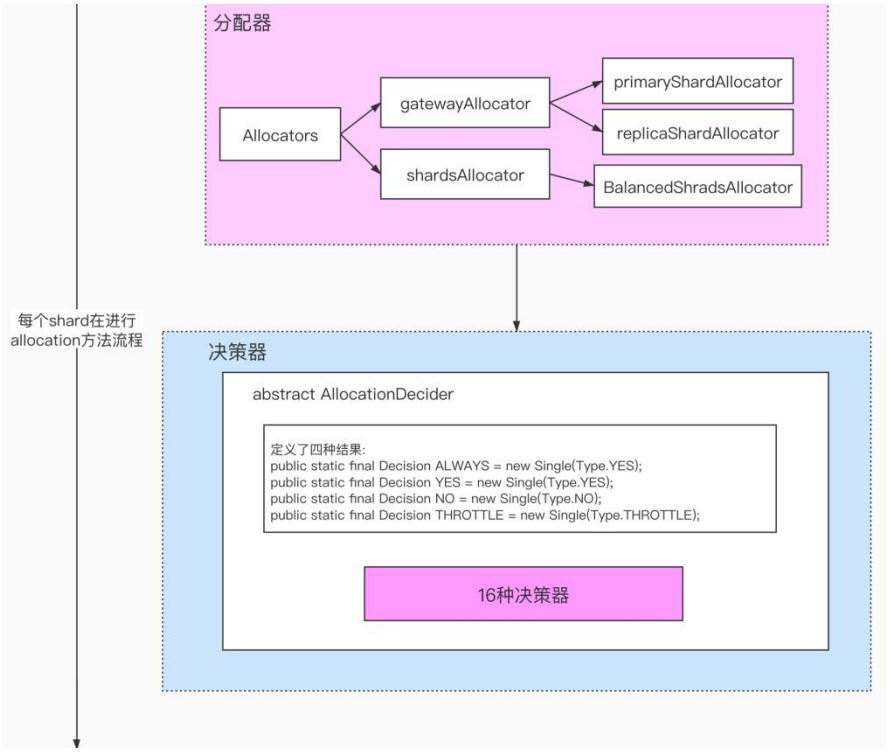
cluster status

集群健康状态，检测到集群状态非 green，则说明巡检异常结果未处理，或者突发情况导致，此时巡检的意义是快速的给出推荐解决方案，让运维、集群 owner（开发）能够有处理的方案，并非一味依赖告警等待运维处理，最大限度的减少异常带来的影响；

非 green 状态下，首先检查的是节点个数是否符合预期，节点数量正常情况下，通常通过 explain API 进行分析。

```
GET /_cluster/allocation/explain
{
  "index":"index_name",
  "shard": 1,
  "primary":false
}
```

分片 allocation 是通过分配器和决策器来决定的，explain API 通过决策器的信息来体现 unassigned shard 的异常原因，allocation 原理图如下：



详细流程图链接：https://www.processon.com/view/link/5f43b268e401fd5f24852544

该部分只要非 green 状态即为异常，智能巡检系统会分析出异常原因，并给出推荐解决方案到最终的报告中，如果是出现 red 场景或者 OOM 导致掉节点场景，则会由告警平台即时通知到用户和 OPS，用户和 OPS 可以通过在 PAAS 平台记录的报告，快速查看状态异常的分析；由于巡检系统非实时，可以手动触发智能巡检系统，快速得到最新分析报告与推荐解决方案。

推荐解决方案由两部分组成：

- 一些特定场景分析后的经验建议。
- 该集群所有 unassigned shard 通过 explain API 的查到的结果集。

示例代码见示例代码：https://www.teambition.com/project/601f63c9997fc9a15fb8e683/app/5eba5fba6a92214d420a3219/workspaces/601f659743c5e10046459556/docs/606285c1eaa1190001e688b1#608588d89645b900464132e3

pengding_task

pending_task 反应了 master 节点尚未执行的集群级别的更改任务（例如：创建索引，更新映射，分配分片）的列表。pending_task 的任务是分级别的（优先级排序：IMMEDIATE>URGENT>HIGH>NORMAL>LOW>LANGUID），只有当上一级别的任务执行完毕后，才会执行下一级别的任务，即当出现 HIGH 级别以上的 pending_task 任务时，

备份和创建索引等低级别任务将延迟执行。

pending_task 过多会给 master 节点造成压力，大集群（大数据量、高并发）情况下，容易造成节点被踢出集群，甚至集群不响应的情况。pending_task 积压的场景一般出现在大集群中，由于 task 不能快速处理完，会长时间处于积压状态，且会越积压越多，所以异常阈值可以设置一个较大值，根据经验值，设置 pending_task 数量大于 100 为异常；

```
GET _cat/pending_tasks
```

节点最大 cpu_util 与最小 cpu_util 之差（极差）

cpu_util 是判断 Elasticsearch 集群健康程度的重要指标，直接影响集群的吞吐量、请求的响应时间。一般情况下集群各个节点 cpu_util 普遍过高的场景比较好处理，增加配置即可让集群恢复健康状态。

而单节点/部分节点出现 cpu_util 过高的情况则不容易定位，如 shard 分配不均匀、routing 设置不合理、宿主机异常都有可能引发该现象，且可能在程序上线一段时间后才出现，这时巡检结果对预警与分析有重大意义。

```
GET _cat/nodes?h=ip,cpu
```

为了能够有效起到预警作用，cpu_util 极差的阈值不宜过大，同时应该考虑到角色分离、冷热分离等场景，将极差的计算范围限制到同角色节点之间。

data 节点最大 qps 与最小 qps 之差（极差）

节点 qps 指标受到 shard 分配、routing 的影响，极差过大代表集群数据分配不均，或有参数干预，集群负载不均衡。

为该指标设置阈值时，同样需要考虑到角色分离场景，同时由于业务的不同，不同集群 qps 相差巨大，可以通过换算成瞬时流量的百分比来做极差计算。

```
GET /_nodes/stats/indices,ingest/search
```

一次 bulk 请求的数量

bulk 请求适用于大写入场景，由于减少了大量的连接，bulk 的效率远高于单条 index。一次 bulk 请求涉及到的 doc 数量对性能有较大影响，过小容易造成线程池堆积，过大容易造成超时。

该指标的阈值设置，仅针对写入/更新频率较大的集群，由于集群配置的差异，可给每个配置区间内设置一个最小值作为巡检的阈值。

node 层面指标分析

uptime

集群节点重启往往是对集群性能有着较大影响，接收到的流量会异常，同时减少了

一部分吞吐量，对于大集群而言，重建缓存、shard allocation 会对集群的响应造成影响。集群重启的原因以及重启带来的影响对于集群都是一个隐患。

```
GET _cat/nodes?h=ip,name,node.role,master,uptime
```

如果启动时间距离当前时间 1h 以内，则节点发生重启，提示用户检查重启原因、评估带来的影响。

free_disk

磁盘剩余空间最直观的就是影响数据写入，到达水位线（默认 95%）后会进入 read only 状态，其次磁盘空间还会带来其他隐患，比如无法操作 forcemerge、甚至 deleteByQuery 也无法完成等。

```
GET _cat/allocation
```

出于成本与利用率的考虑，预期状态磁盘占比不应过低，一般 50% 以内为安全值，由于不同集群的特性可能会有差异，可以设置 70% 作为异常阈值。

cpu_util

CPU 对集群性能影响极大，高 CPU 的场景下，响应时间增加，可能出现大量超时（读写异常）。所以需要平衡 cpu 利用率（一般通过超分、调整配置两种方式）。

```
GET _cat/nodes?h=ip,name,cpu
```

由于业务流量存在周期性波峰波谷，所以阈值需要在安全范围内尽量设置大一点，例如设置阈值为"cpu_util = 90%"。

node 上 shard 数

节点未分配 shard，则需要确认资源容量分配是否合理，该指标主要针对分片设置不合理导致的资源浪费。一般发生在分片数量设置不合理、迁移过程（存在 exclude）的场景下。故阈值可以设置为 "count(shard) = 0"。

```
GET _cat/allocation
```

shard 层面指标分析

number

过多或过少的 shard 在一定场景下都会影响查询和写入性能。官方建议的合理的设置数量：每 GB 的 heap 不超过 20 个 shards; 比如 20GB heap，400 个 shards, 30 GB heap，不超过 600 个 shards。Elasticsearch 7.0 版本开始，集群中每个节点默认限制 1000 个 shard。

阈值按照官方建议值设定即可。

```
GET _cat/shards?h=index,shard,prirep,state,docs,store,ip,node
```

size of per shard

大量的小 shard 会影响写入和查询性能，且在同数据量情况下占用更多的内存和磁盘。单 shard 过大则有更多的弊端，例如查询耗时变长、不易于恢复、迁移、容易造成集群压力不均衡等。

通常单个 shard 的大小建议在 10GB - 65GB 之间 （经验值参考：搜索类控制在 20GB，日志类控制在 50BG），查询 API 同上。

官方建议 shard size、count 值参考：https://www.elastic.co/guide/en/elasticsearch/reference/current/size-your-shards.html

Index 层面指标分析

replica

所有集群的 index 都应该有副本分片，没有副本分片的 index 在节点 crash 时会丢失数据。

```
GET index_name/_settings
```

当 number_of_replicas 为 0 时候异常情况。

动态 mapping

dynamic mapping 设置为 true 会使得 mapping 变得不可维护，且 mapping 源数据由 master 维护、分发，大量变更可能导致 master 压力过大，在高峰情况下，可能会使得积压大量 task，引发集群不响应、踢出节点等问题。

```
GET /*/_mapping?format=json
```

巡检需要检查出 "dynamic=true（或默认）"的索引的集群，标记为异常。

refresh_interval

索引 refresh 频率是影响性能的一个因素，受到 refresh_interval 参数与 buffer 大小的影响，由于业务场景的差异，对 refresh 的设置可能大不相同，可将集群类型大致分为搜索类型与数据分析类型，根据类型的不同设置差异化的阈值，且集群不应该出现"refresh_interval = -1"的设置。

```
GET /*/_settings?include_defaults=true
```

indices.refresh.total

refresh 的频率影响着 segment 的生成速度与大小，而 segment 过多往往影响查询性能，并且需要消耗更多的内存和磁盘空间。由于默认值为"refresh_interval = 1s"，不考虑 buffer 的影响可以认为 refresh 频率为 60/min，故巡检阈值可以设置到比默认值稍高，例如：count(refresh) = 80/min。

```
GET /_nodes/stats/indices,ingest/refresh
```

max_result_window

max_result_window 为单次请求返回 doc 的最大值，默认为 10000，该默认值的限制可以覆盖到所有正常的业务场景。一般是深度分页、全量查询、job 查询可能导致返回 doc 数大于 10000，触发异常，而这些场景可以由 scroll、search after 来完成。故该指标阈值可以设置成该参数默认值。

```
GET /*/_settings?include_defaults=true
```

jvm 层面指标分析

jvm heap 使用率

jvm 堆的使用率过高有着 OutOfMemory 的风险，并使得 GC 频率过高，影响请求响应时间。由于使用的 G1 收集器，首次 GC 收集会在预估 GC 时间达到预定值的时候开始触发，则 heap 使用率的稳定值也随着参数设置而产生较大差异。而该参数主要是为了预防 OutOfMemory 异常，所以该指标阈值可以设置一个较大值，例如 "heap > 90"。

```
GET /_nodes/stats/indices,jvm
```

jdk version 一致性

由于 Elasticsearch 的分布式属性，集群存在多节点，每个节点一个单独的实例，需要保证 jdk 版本一致。

jvm heap segment memory

segment memory 常驻 heap 内存，所以 segment memory 的增长会压缩其他对象的内存空间。segment memory 是每个 segment 倒排词典上层的一个前缀索引，即 FST 结构，该前缀索引会在 segment 不断的累积下逐渐增多。

为了防止其对 heap 内存过多的占用，需要对该值继续检查限制，由于 FST 结构对前缀索引进行大量压缩，正常状态下对 heap 占用较低，巡检阈值也可以设置较低，例如 20% heap_size。

```
GET /_nodes/stats/indices,ingest/segments
```

full gc

Elasticsearch 7.x 默认使用的 G1 垃圾收集器，所以一般会是 Young GC 或 Mixed GC，如果 mixed GC 无法跟上新对象分配内存的速度，导致老年代填满无法继续进行 Mixed GC，于是使用 full GC 来收集整个 heap。G1 不提供 full GC，使用的是 serial old GC。所以该 full GC 是单线程串行的，且 stop the world，这对业务来说是致命的。所以该巡检的阈值为"count(full gc) > 0"。

```
GET /_nodes/stats/indices,jvm
```

threadpool 层面指标分析

bulk reject 数量

bulk 出现 reject 意味着线程池中线程被完全占用，且队列也已经占满。该指标阈值可设置为 "count(bulk rejected)>0"。

```
GET _cat/thread_pool
```

search reject 数量

search 出现 reject 意味着线程池中线程被完全占用，且队列也已经占满。该指标阈值可设置为 "count(search rejected)>0"，查询 API 同上。

结语

本章详细介绍了智能巡检系统的结构与指标选取、阈值确认，并给出 cluster status 指标采集分析的完整示例代码。有兴趣的读者可以将智能巡检系统通过 python 脚本或者 operator 的方式实现。由于 PAAS 系统下管理的大量差异巨大的 Elasticsearch 集群，巡检系统实现主要难点与重点，是抽象出合理的指标以及精细化实现。

上述介绍中可以看到部分指标项是固定场景下必现，且解决方案唯一且简单，例如 shard_limit 场景，一般通过 API 调整 total_shards_per_node 参数即可恢复，因此这些场景可以设计成自动化修复，达到简单的“自愈”，解放开发维护人员的生产力。

示例代码

```
# -*- coding:utf-8 -*-"""
analyse cluster status exception,
匹配常用的 case, 返回 explain 完整结果
"""import espaas_apiimport argparseimport tracebackimport jsonimport loggingimport ge
ventimport timeimport datetimeimport requestsfrom libs.log import initlogfrom gevent impor
t monkey
monkey.patch_all()
timeout = 10# 7.10 elasticsearch 有 16 种决策器
dict = {
    'same_shard': '默认配置下一个节点不允许分配同一 shard 的多个副本分片', # 不参与统一
处理
    'shards_limit': '节点限制单索引 shard 数，调整相应索引的 total_shards_per_node 参数',
    'disk_threshold': '磁盘空间达到水位线无法分配 shard，调整磁盘容量、节点数或者清理磁
盘',
    'max_retry': '达到 allocate 最大重试次数，尝试调用 API 手动 retry',
    'awareness': '副本分配过多，大于 awareness 的配置',
    'cluster_rebalance': '集群正在 Rebalance，可忽略',
    'concurrent_rebalance': '集群当前正在 Rebalance，可忽略',
    'node_version': '分配到的节点版本不一致，请联系管理员调整',
    'replica_after_primary_active': '主分片未 active，等待并观察主分片',
    'filter': '未通过 filter，请检查配置',
    'enable': '集群 enable 限制',
    'throttling': '集群正在恢复，需要按照优先级恢复 primary > replica',
    'rebalance_only_when_active': 'Only allow rebalancing when all shards are active wi
thin the shard replication group',
```

```
    'resize': 'An allocation decider that ensures we allocate the shards of a target index fo
r resize operations next to the source primaries',
        'restore_in_progress': 'This allocation decider prevents shards that have failed to be
 restored from a snapshot to be allocated',
        'snapshot_in_progress': 'This allocation decider prevents shards that are currently b
een snapshotted to be moved to other nodes'
    }
    def check_exception_status_reason_advice(cluster):
        # 省略结果数据入库代码
        res = {
            "status": "",
            "reason": "",
            "advice": "",
            "detail": ""
        }
        # 1.health API
        # 2.node count check
        url = gen_url(cluster) + "/_cat/health?format=json"
        r = requests.get(url, auth=(cluster["username"], cluster["password"]), timeout=timeout)
        if r.status_code == 200:
            try:
                content = json.loads(r.content)
                cluster_status_res = content[0]
                res["status"] = cluster_status_res["status"]
                if cluster_status_res["status"] == "green":
                    return res
                if cluster_status_res["node.total"] != cluster["nodes_count"]:
                    res["reason"] = "node_left"
```

```
                    res["advice"] = "请检查 k8s 环境中 node 数，使用`kubectl describe po pod_name`检查 crash reason"
                    return res
            except:
                traceback.print_exc()
        else:
            res["status"] = "unknown"
            return res
        # 3.get UNASSIGNED shard
        get_all_shard_url = gen_url(cluster) + "_cat/shards?format=json"
        get_all_shard_res = requests.get(get_all_shard_url, auth=(cluster["username"], cluster["password"]),
                                         timeout=timeout)
        unassigned_shard_list = []
        unassigned_shard_final_list = []
        if get_all_shard_res.status_code == 200:
            try:
                shards = json.loads(get_all_shard_res.content)
                for shard in shards:
                    if shard["state"] == "UNASSIGNED":
                        unassigned_shard_list.append(shard)
                if len(unassigned_shard_list) > 20:
                    unassigned_shard_final_list = unassigned_shard_list[0:20]
                else:
                    unassigned_shard_final_list = unassigned_shard_list
            except:
                traceback.print_exc()
        # 4.explain
        # 5.经验值提取出常用建议
```

```
        decider_list = []
        explanation_list = []
        has_advice = 0
        explain_res_all = []
        for shard_tmp in unassigned_shard_final_list:
            index_name_tmp = shard_tmp["index"]
            shard_pos = shard_tmp["shard"]
            shard_type = shard_tmp["prirep"]
            is_primary = False
            if shard_type == "p":
                is_primary = True
            explain_url = gen_url(cluster) + "/_cluster/allocation/explain"
            payload = {
                "index": index_name_tmp,
                "shard": shard_pos,
                "primary": is_primary
            }
            explain_res = requests.post(explain_url, json.dumps(payload),
                                        auth=(cluster["username"], cluster["password"]),
                                        headers={"Content-Type": "application/json"}).json()
            explain_res_all.append(explain_res)
            # 整理单个分片所有的 decider 信息
            same_shard_count = 0
            for decisions_tmp in explain_res["node_allocation_decisions"]:
                for decider_tmp in decisions_tmp['deciders']:
                    decider_list.append(decider_tmp['decider'])
                    explanation_list.append(decider_tmp['explanation'])
                    if decider_tmp['decider'] == "same_shard":
                        same_shard_count = same_shard_count + 1
```

```
            tmp_string = ''
            explanation_list_string = tmp_string.join(explanation_list)
            if explanation_list_string.find("ik_max_word") != -1:
                res["reason"] = "max_retry"
                res["advice"] = "可能触发 6.8.5Ik 分词器 bug，请使用其他兼容版本或根据 issue f
ix"
                has_advice = 1
                break
            if explanation_list_string.find("cluster.routing.allocation.enable=primaries") == -1 \
                    and explanation_list_string.find("cluster.routing.allocation.enable=replicas")
 == -1:
                print '正常'
            else:
                res["reason"] = "enable"
                res["advice"] = "cancel cluster.routing.allocation.enable=replicas/primaries set
tings"
                has_advice = 1
                break
            if 'same_shard' in decider_list and same_shard_count == cluster["nodes_count"]:
                res["reason"] = "same_shard"
                res["advice"] = "shard 副本设置过多，调整副本数量"
                has_advice = 1
                break
            # 取 decider 与 ES 决策器 keys 的交集
            decider_and = list(set(decider_list).intersection(set(dict.keys())))
            reason_and = ''
            advice_and = ''
            if list(set(decider_list).intersection(set(dict.keys()))):
```

```
                for x in decider_and:
                    if x != 'same_shard':
                        reason_and += x + ';'
                        advice_and += dict[x] + ';'
                res["reason"] = reason_and
                res["advice"] = advice_and

        if has_advice == 0:
            res["detail"] = explain_res_all
        return res
    def gen_url(cluster):
        url = 'http://' + cluster["domain"] + ':' + str(cluster["http_port"]) + cluster["http_path
"]
        if url.endswith("/"):
            url = url[:-1]
        return url
    def main(cluster_id=None):
        try:
            if cluster_id:
                clusters = espaas_api.get("cluster", "/api/get_cluster_by?cluster_id=%s" % cl
uster_id)["data"]
            else:
                clusters = espaas_api.get("cluster", "/api/all_status_exception_cluster")["data
"]
            jobs = []
            for cluster in clusters:
                try:
                    jobs.append(gevent.spawn(check_exception_status_reason_advice, cluste
r))
```

```
                except:
                    traceback.print_exc()
            logging.info("send job number: %s" % len(jobs))
            gevent.joinall(jobs)
        except:
            traceback.print_exc()
if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument("-l", default="-", help="log file")
    parser.add_argument("--level", default="info")
    args = parser.parse_args()
    initlog(level=args.level, log=args.l)
    main()
```

创作人简介：

张妙成，ES PAAS 平台开发，云计算技术爱好者。

个人博客：https://blog.csdn.net/qq_33999844?spm=1001.2014.3001.5343

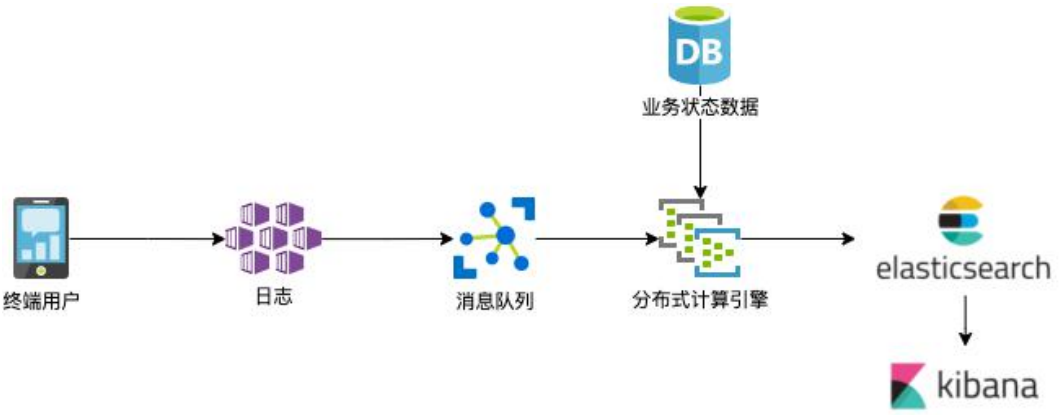
4.2.3 CDN 流媒体服务实时分析 ES 实践

创作人：吴斌

审稿人：李捷

发挥 Elastic Stack 在日志和实时数据分析计算领域的一些优势，对流媒体服务这样规模较大、实时性要求偏高，且分析、业务探索流程要求灵活的业务是一个比较百搭的选择。

数据逻辑架构如下：



整体架构比较直观简单。我们省去了业务组建和存储层可能会用到的其他引擎，把目光主要集中在 Elasticsearch 上。

日志采集

日志的收集在正式进入数据管道前，可以选择落地或者直接吐到消息队列。这里采集的内容也主要分成 2 大部分：

- CDN/网络访问日志
- 业务打点数据

业务打点的数据可以根据需求采集，实时部分主要会聚焦在，例如卡顿等这样的用户体验指标。网络访问的日志通常比较通用，这里我们也先给出一个例子，相信大家看上去会比较熟悉。

那么根据这里样例的数据，Elasticsearch 可以轻松的利用内置的 processor 和聚合功能做快速的分析，后面我们会举例说明。

```
{
    "receiveTimestamp": "2021-04-28T14:30:17.90993285Z",
    "spanId": "blahblah",
    "trace": "blah/f5c7578feaf277dd9a8d96",
    "@timestamp": "2021-04-28T14:30:17.549287Z",
    "logName": "logs/requests",
    "jsonPayload": {
      "@type": "loadbalancing.type.LoadBalancerLogEntry",
      "latencySeconds": "0.001749s",
      "statusDetails": "response_from_cache",
      "cacheIdCityCode": "ABC",
      "cacheId": "ABC-abcabc123"
    },
    "httpRequest": {
```

```
"remoteIp": "10.0.0.1",
"remoteIpIsp": {
  "ip": "10.0.0.2",
  "organization_name": "China Telecom",
  "asn": 8346,
  "network": "10.0.0.0/15"
},
"requestMethod": "GET",
"responseSize": "125621",
"userAgent": "Mozilla/5.0 (Linux; Android 10) Bindiego/7.1-1840",
"frontendSrtt": 0.124,
"cacheLookup": true,
"geo": {
  "continent_name": "Asia",
  "country_iso_code": "CN",
  "country_name": "China",
  "location": {
    "lon": 123,
    "lat": 321
  }
},
"backendLatency": 0.001749,
"requestUrl": "http://bindiego.com/vid/bindigo.m4s",
"requestDomain": "bindiego.com",
"cacheHit": true,
"requestSize": "671",
"requestProtocol": "http",
"user_agent": {
  "original": "Mozilla/5.0 (Linux; Android 10) Bindiego/7.1-1840",
  "os": {
    "name": "Android",
```

```
            "version": "10",
            "full": "Android 10"
          },
          "name": "Android",
          "device": {
            "name": "Generic Smartphone"
          },
          "version": "10"
        },
        "status": 200,
        "resourceType": "m4s"
      }
    }
```

这里就是最终导入到 Elasticsearch 里可分析的网络性能数据。针对这个数据，我们分别对它经过的管道和处理做一个简单快速的剖析。

消息队列

日志采集器会直接把数据打到消息队列，这里主要起到一个抗反压缓冲的作用。有些队列有很多附加的功能，例如存储和窗口计算，这里我们只使用最单纯的功能。因为后面我们选取了分布式计算引擎来做这这部分。

分布式计算引擎

分布式计算引擎，其实在整体实时数据分析业务里，扮演的着实是非常重要的角色。

例如实时指标的窗口计算，迟报数据的修正等等。但在我们这个简单的场景下为了后面在 Elasticsearch 内更方便快捷的分析、过滤数据。

我们这里主要做了 ETL 和补全。例如把请求资源的域和资源类型提取出来，还有 CDN 缓存节点的区域代码等等。但是例如 IP 地址地理位置、用户设备类型和运营商（ISP）的反查，方便起见，我们利用了 Elasticsearch Ingest 节点预置的 Pipeline 去做。

这里要注意的就是，如果你的集群配置是全角色的节点，会对数据节点的性能有影响。建议使用独立的 ingest node 去做，且如果是在 K8S 上部署的话，还可以弹性扩容这组 nodeSet。

下面是 Ingest 节点配置举例：

完整代码戳这里：https://github.com/cloudymoma/raycom/blob/gcp-lb-log/scripts/elastic/index-gclb-pipeline.json

```
{
    "description": "IP & user agent lookup",
    "processors": [
      {
        "user_agent" : {
          "field" : "httpRequest.userAgent",
          "target_field" : "httpRequest.user_agent",
          "ignore_missing": true
        }
      },
```

```
        {
          "geoip" : {
            "field" : "httpRequest.remoteIp",
            "target_field" : "httpRequest.geo",
            "ignore_missing": true
          }
        },
        {
          "geoip" : {
            "field" : "httpRequest.remoteIp",
            "target_field" : "httpRequest.remoteIpIsp",
            "database_file" : "GeoLite2-ASN.mmdb",
            "ignore_missing": true
          }
        }
      ]
}
```

数据安全

数据安全这块顺带提一下，现在 Elasticsearch 的认证、授权都可以在 Basic License 里使用了，非常方便。这里简单提一下通讯这块，很多小伙伴用的是自签的证书。这个问题不大，经常被问到在使用 RestClient 开发的时候如何绕过去（例如在写计算引擎最后入库的时候）。其实方法也很简单，这里就给大家上个代码片段说明看下.

配置:https://github.com/elasticsearch-cn/elastic-on-gke#option-2-regional-tcp-lb

完整代码:https://github.com/cloudymoma/raycom/blob/streaming/src/main/java/bindiego/io/ElasticsearchIO.java#L273-L296

```
try {
    SSLContext context = SSLContext.getInstance("TLS");

    context.init(null, new TrustManager[] {
        new X509TrustManager() {
            public void checkClientTrusted(X509Certificate[] chain, String authType) {}

            public void checkServerTrusted(X509Certificate[] chain, String authType) {}

            public X509Certificate[] getAcceptedIssuers() { return null; }
        }
    }, null);

    httpAsyncClientBuilder.setSSLContext(context)
        .setSSLHostnameVerifier(NoopHostnameVerifier.INSTANCE);
} catch (NoSuchAlgorithmException ex) {
    logger.error("Error when setup dummy SSLContext", ex);
} catch (KeyManagementException ex) {
    logger.error("Error when setup dummy SSLContext", ex);
} catch (Exception ex) {
    logger.error("Error when setup dummy SSLContext", ex);
}
```

索引管理

索引生命周期管理 Elasticsearch 也提供了非常便利的工具。

生命周期配置，这里应该根据业务需求和节点规模综合考量：

```
{
    "policy": {
        "phases": {
            "hot": {
                "actions": {
                    "rollover": {
                        "max_size": "20GB",
                        "max_docs": 20000000,
                        "max_age": "7d"
                    }
                }
            },
            "delete": {
                "min_age": "30d",
                "actions": {
                    "delete": {}
                }
            }
        }
    }
}
```

索引模版

完整版:https://github.com/cloudymoma/raycom/blob/gcp-lb-log/scripts/elastic/index-gclb-template.json

模版为每次生成的索引应用相同的配置，且指定了生命周期的政策文件和注入别名。

```
{
    "index_patterns": [
        "bindiego*"
    ],
    "order": 999,
    "settings": {
        "number_of_shards": 2,
        "number_of_replicas": 1,
        "final_pipeline": "bindiego",
        "index.lifecycle.name": "bindiego-policy",
        "index.lifecycle.rollover_alias": "bindiego-ingest"
    },
    "mappings": {
```

最后我们配置了脚本一次性把上述配置应用，且在 Kibana 里为我们建立好查询的 index pattern

详细戳这里：https://github.com/cloudymoma/raycom/blob/gcp-lb-log/scripts/elastic/init.sh

数据面板

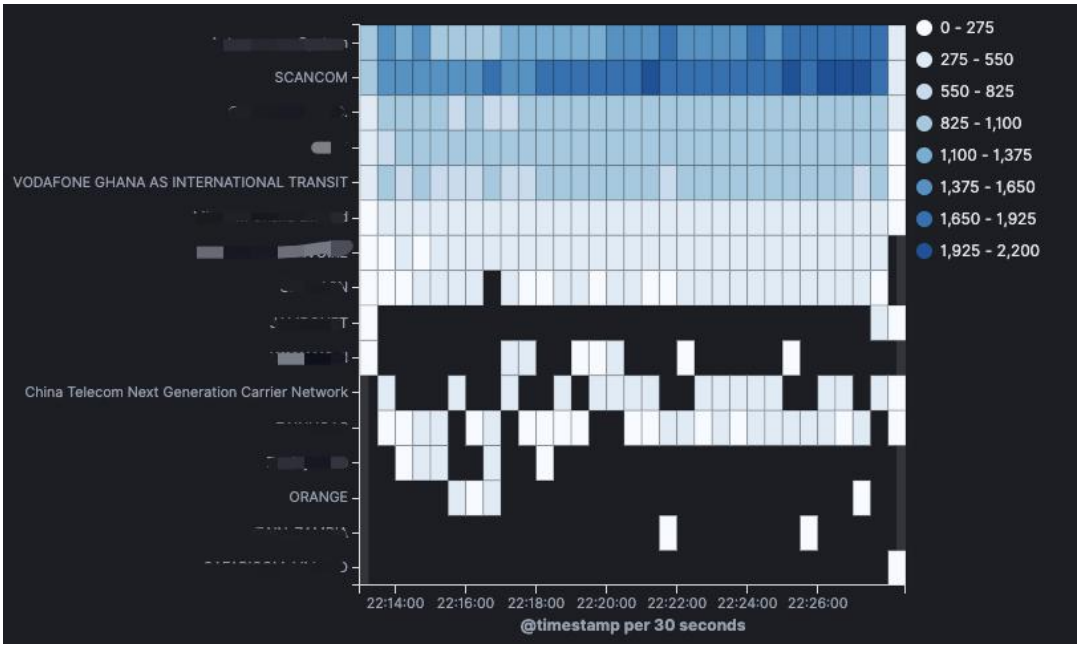
这里虽然是个人弱项，但是借助 Kibana 强大的可视化功能，可以根据第一部分整理出来的数据绘制实时面板。

完整可复用面板：https://github.com/cloudymoma/raycom/blob/gcp-lb-log/scripts/elastic/gclb_dashboard.ndjson

部分截图：https://github.com/cloudymoma/raycom/tree/gcp-lb-log#dashboards-in-kibana

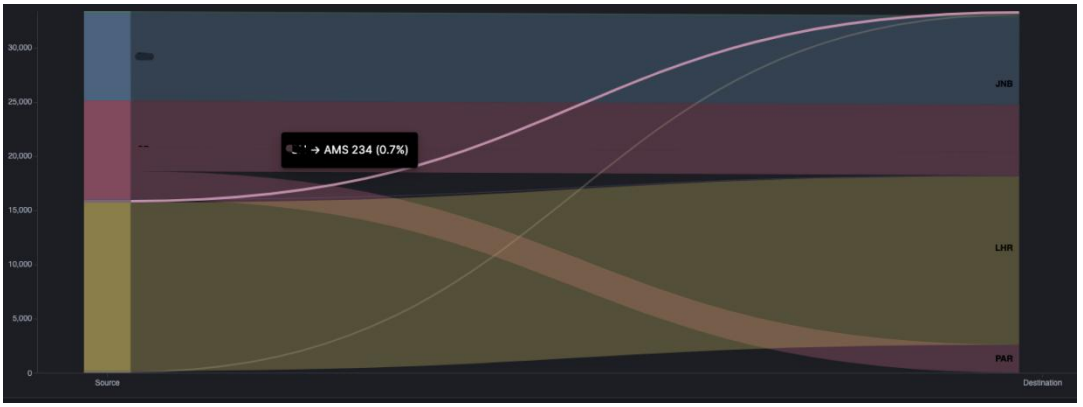
下面我举一些可能常被忽视的好用功能给大家打个样。

IP 反查出的 Geo 和 ISP 信息：



通过这些信息，可以快速反映出各个运营商网络的情况，甚至一些盗链的线索初判断。

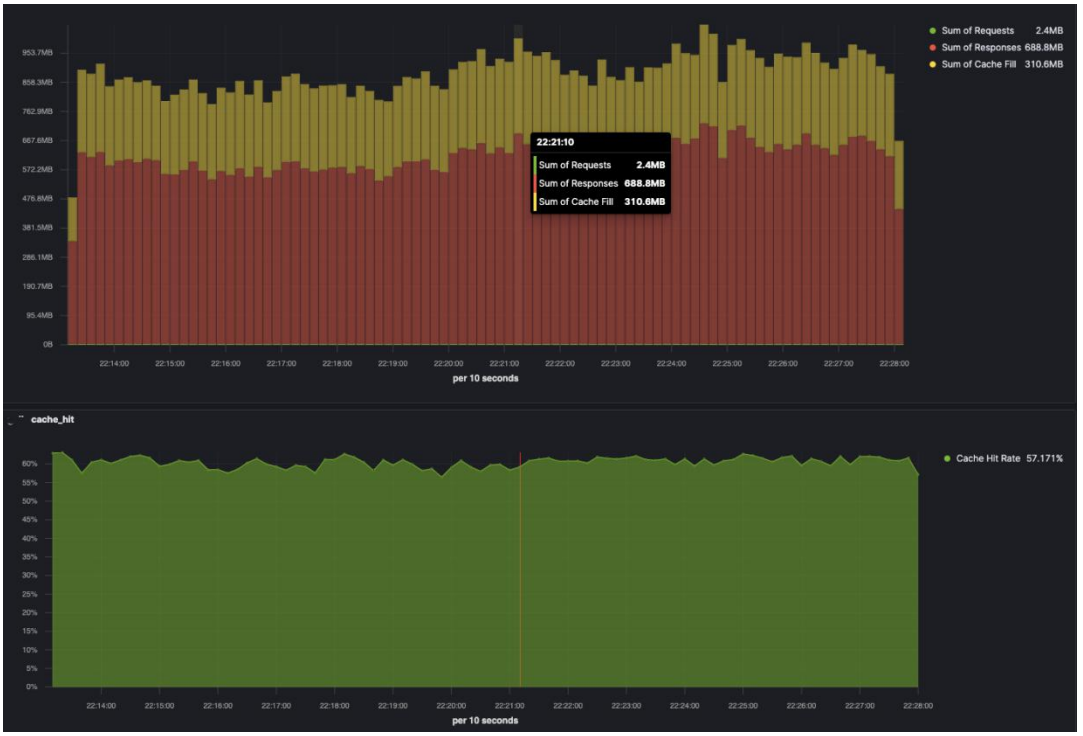
Vega 在 Kibana 里绘制数据：



当我们觉得 Kibana 自身图表不够丰富的时候，可以借助 Vega。上面这个图就展示了来自不同地区的用户，分别命中 CDN 缓存点的流量分配。数据通过用 Elasticsearch 的 Composite Aggregation 提取。

Kibana TSVB

这个是我个人最喜欢的绘图方法了，可以非常灵活的对指标进行计算。下面这两个图表就展示过滤出直播业务的缓存命中、请求返回和缓存填充的数据量这些信息。



总结

由于业务数据的敏感性，这里就不列举细节了。但数据管道和治理，都依旧遵循同样的原则。整体数据管道的选型也非常灵活，采集部分即可以是 Beats 生态中的产品，也可以是自己开发的 agent。队列常用的有 Kafka 或者云上托管服务。分布式计算层因为业务比较简单，我比较推荐使用 Apache Beam，这样执行引擎可以在比如 Flink、Spark Streaming 和任何 Beam 支持的平台上相对灵活的切换。

今天我们给出的案例是一个非常简单，且可以快速复用的开源项目。

大家有任何需求和疑问也欢迎到社区一起交流、学习。

创作人简介：

吴斌，Elastic 中文社区副主席，现就职于大型互联网公司任职云架构师。专注于海量数据处理、挖掘、分析和企业级搜索领域。十分熟悉分布式应用，高可用架构和自动化技术。曾在海外世界百强大学计算机学院任教 6 年。更是一位开源软件社区的积极贡献者和组织者。

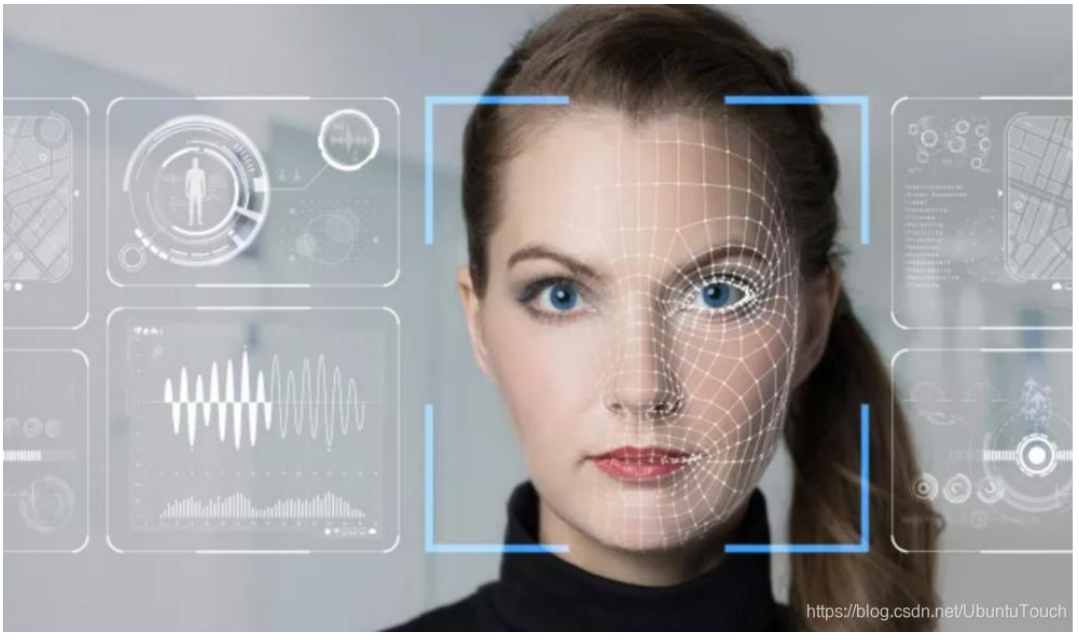
博客：https://gist.github.com/bindiego

4.2.4 Elasticsearch 和 Python 构建面部识别系统

创作人：刘晓国

你是否曾经尝试在图像中搜索目标？ Elasticsearch 可以帮助你存储，分析和搜索图像或视频中的目标。

在本文中，我们将向你展示如何构建一个使用 Python 进行面部识别的系统。 了解有关如何检测和编码面部信息的更多信息-并在搜索中找到匹配项。



我们将参照代码：https://github.com/liu-xiao-guo/face_detection_elasticsearch。你可以把这个代码下载到本地的电脑：

```
$ pwd
/Users/liuxg/python/face_detection
$ tree -L 2
.
├── README.md
├── getVectorFromPicture.py
├── images
│   ├── shay.png
│   ├── simon.png
│   ├── steven.png
│   └── uri.png
├── images_to_be_recognized
│   └── facial-recognition-blog-elastic-founders-match.png
└── recognizeFaces.py
```

在上面的代码中，有如下的两个 python 文件：

- getVectorFromPicture.py：导入在 images 目录下的图像。这些图像将被导入到 Elasticsearch 中。
- recognizeFaces.py：识别位于 images_to_be_recognized 目录下的图像文件。

基础知识

面部识别

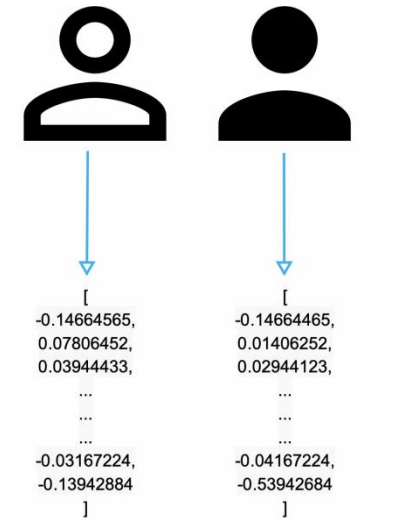
面部识别是使用面部特征来识别用户的过程，例如，为了实现身份验证机制（例如解锁智能手机）。 它根据人的面部细节捕获，分析和比较模式。 此过程可以分为三个步骤：

- 人脸检测：识别数字图像中的人脸
- 人脸数据编码：将人脸特征转换为数字表示
- 脸部比对：搜寻和比较脸部特征

在示例中，我们将引导你完成每个步骤。

128 维向量

可以将面部特征转换为一组数字信息，以便进行存储和分析。



Vector data type

Elasticsearch 提供了 dense_vector 数据类型来存储浮点值的 dense vectors。向量中的最大尺寸数不应超过 2048，这足以存储面部特征表示。

现在，让我们实现所有这些概念。

准备

要检测面部并编码信息，你需要执行以下操作：

- Python：在此示例中，我们将使用 Python 3
- Elasticsearch 集群：你可以免费使用 阿里云 Elasticsearch 来启动集群。本文中，我将进行一个本地的部署 Elasticsearch 及 Kibana。
- 人脸识别库：一个简单的人脸识别 Python 库。
- Python Elasticsearch 客户端：Elasticsearch 的官方 Python 客户端。

客户端下载：https://elasticsearch-py.readthedocs.io/en/v7.10.1/

Python 教程：https://elasticstack.blog.csdn.net/article/details/111573923

Python 下载：:https://www.python.org/downloads/

注意，我们已经在 Ubuntu 20.04 LTS 和 Ubuntu 18.04 LTS 上测试了以下说明。根据你的操作系统，可能需要进行一些更改。尽管下面的安装步骤是针对 Ubuntu 操作系统的，但是我们可以按照同样的步骤在 Mac OS 上进行同样的顺序进行安装（部分

指令会有所不同）。

安装 Python 和 Python 库

随 Python 3 的安装一起提供了 Ubuntu 20.04 和其他版本的 Debian Linux。

如果你的系统不是这种情况，则可以点击下载并安装 Python：https://www.python.org/downloads/

要确认你的版本是最新版本，可以运行以下命令：

```
sudo apt update
sudo apt upgrade
```

确认 Python 版本为 3.x：

```
python3 -V
```

或者：

```
python --version
```

安装 pip3 来管理 Python 库：

```
sudo apt install -y python3-pip
```

安装 face_recognition 库所需的 cmake：

```
pip3 install CMake
```

将 cmake bin 文件夹添加到 $PATH 目录中：

```
export PATH=$CMake_bin_folder:$PATH
```

在我的测试中，上述步骤可以不需要。你只要在任何一个 terminal 中打入 cmake 命令，如果能看到被执行，那么就可以不用上面的命令了。

最后，在开始编写主程序脚本之前，安装以下库：

```
pip3 install dlib
pip3 install numpy
pip3 install face_recognition
pip3 install elasticsearch
```

从图像中检测和编码面部信息

使用 face_recognition 库，我们可以从图像中检测人脸，并将人脸特征转换为 128 维向量。

为此，我们创建一个叫做 getVectorFromPicture.py:

getVectorFromPicture.py

```
import face_recognition
import numpy as np
import sys
import os
from pathlib import Path
from elasticsearch import Elasticsearch

es = Elasticsearch([{'host':'localhost','port':9200}])

cwd = os.getcwd()
print("cwd: " + cwd)

# Get the images directory
rootdir = cwd + "/images"
print("rootdir: " + rootdir)
```

```
for subdir, dirs, files in os.walk(rootdir):
    for file in files:
        print(os.path.join(subdir, file))
        file_path = os.path.join(subdir, file)

        image = face_recognition.load_image_file(file_path)

        # detect the faces from the images
        face_locations = face_recognition.face_locations(image)

        # encode the 128-dimension face encoding for each face in the image
        face_encodings = face_recognition.face_encodings(image, face_locations)

        # Display the 128-dimension for each face detected
        for face_encoding in face_encodings:
            print("Face found ==>  ", face_encoding.tolist())
            print("name: " + Path(file_path).stem)
            name = Path(file_path).stem
            face_encoding = face_encoding.tolist()

            # format a dictionary to be indexed
            e = {
                "face_name": name,
                "face_encoding": face_encoding
            }
            res = es.index(index = 'faces', doc_type ='_doc', body = e)
```

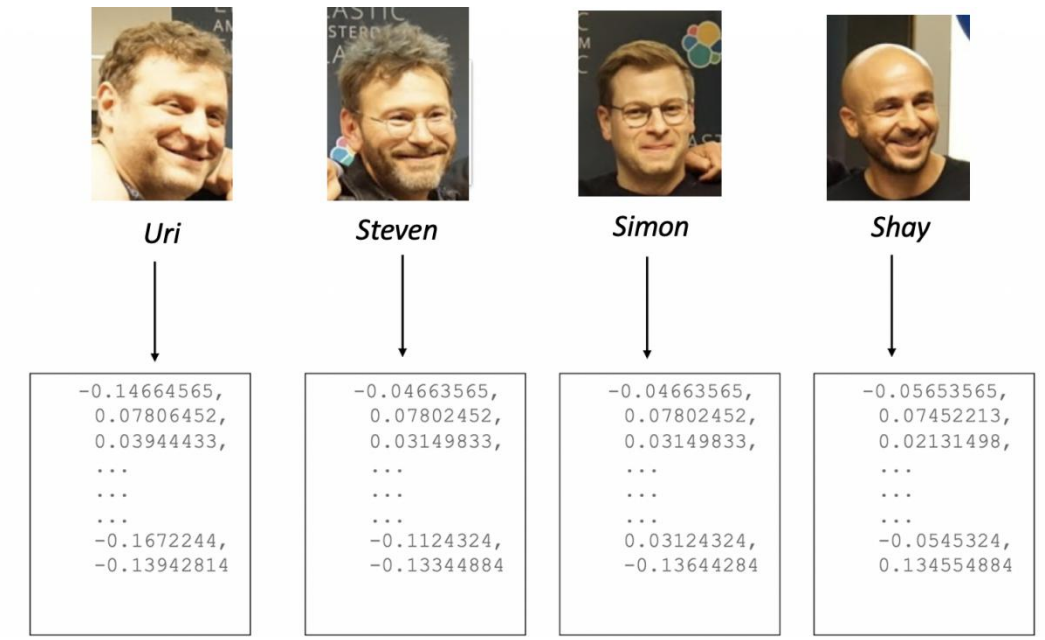
首先，我们需要声明的是：你需要修改上面的 Elasticsearch 的地址，如果你的 Elasticsearch 不是运行于 localhost:9200。上面的代码非常之简单。它把当前目录下的子目录 images 下的所有文件都扫描一遍，并针对每个文件进行编码。我们使用 Python client API 接口把数据导入到 Elasticsearch 中去。在我们的 images 文件夹中，有四个文件。

在导入数据之前，我们需要在 Kibana 中创建一个叫做 faces 的索引：

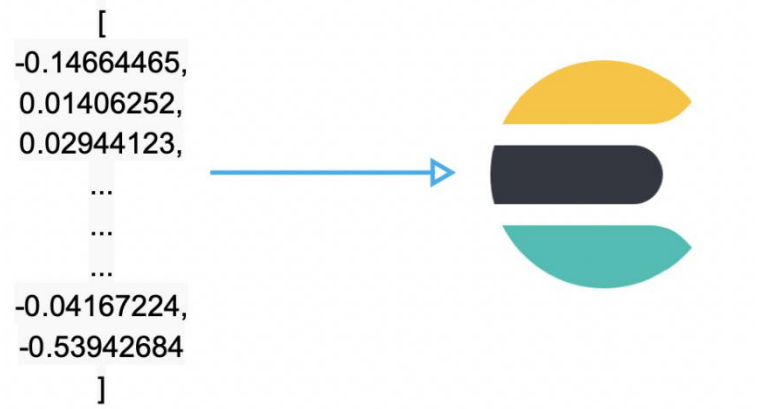
```
PUT faces
{
  "mappings": {
    "properties": {
      "face_name": {
        "type": "keyword"
      },
      "face_encoding": {
        "type": "dense_vector",
        "dims": 128
      }
    }
  }
}
```

让我们执行 getVectorFromPicture.py 以获取 Elastic 创始人图像的面部特征表示。

```
python3 getVectorFromPicture.py
```



现在，我们可以将面部特征表示存储到 Elasticsearch 中。



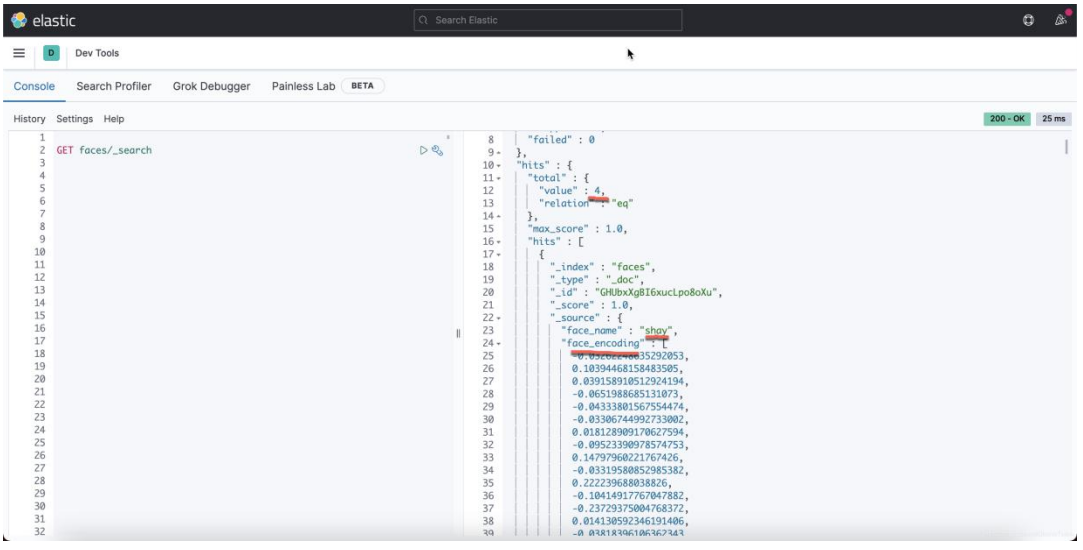
我们可以在 Elasticsearch 中看到四个文档：

```
GET faces/_count
```

```
{
  "count" : 4,
  "_shards" : {
    "total" : 1,
    "successful" : 1,
    "skipped" : 0,
    "failed" : 0
  }
}
```

我们也可以查看 faces 索引的文档：

```
GET faces/_search
```



匹配面孔

假设我们在 Elasticsearch 中索引了四个文档，其中包含 Elastic 创始人的每个面部表情。 现在，我们可以使用创始人的其他图像来匹配各个图像。

为此，我们需要创建一个叫做 recognizeFaces.py 的文件。

recognizeFaces.py

```
import face_recognition
import numpy as np
from elasticsearch import Elasticsearch
import sys
import os
```

```
from elasticsearch import Elasticsearch

es = Elasticsearch([{'host': 'localhost', 'port': 9200}])

cwd = os.getcwd()
# print("cwd: " + cwd)

# Get the images directory
rootdir = cwd + "/images_to_be_recognized"
# print("rootdir: {0}".format(rootdir))

for subdir, dirs, files in os.walk(rootdir):
    for file in files:
        print(os.path.join(subdir, file))
        file_path = os.path.join(subdir, file)

        image = face_recognition.load_image_file(file_path)

        # detect the faces from the images
        face_locations = face_recognition.face_locations(image)

        # encode the 128-dimension face encoding for each face in the image
        face_encodings = face_recognition.face_encodings(image, face_locations)

        # Display the 128-dimension for each face detected
        i = 0
        for face_encoding in face_encodings:
            i += 1
            print("Face", i)
            response = es.search(
```

```
                index="faces",
                body={
                    "size": 1,
                    "_source": "face_name",
                    "query": {
                        "script_score": {
                            "query": {
                                "match_all": {}
                            },
                            "script": {
                                "source": "cosineSimilarity(params.query_vector, 'face_e
ncoding')",
                                "params": {
                                    "query_vector": face_encoding.tolist()
                                }
                            }
                        }
                    }
                }
            )

            # print(response)

            for hit in response['hits']['hits']:
                # double score=float(hit['_score'])
                print("score: {}".format(hit['_score']))
                if float(hit['_score']) > 0.92:
                    print("==> This face  match with ", hit['_source']['face_name'], ",the
 score is", hit['_score'])
                else:
                    print("==> Unknown face")
```

这个文件的写法也非常简单。它从目录 images_to_be_recognized 中获取需要识别的文件，并对这个图片进行识别。我们使用 cosineSimilarity 函数来计算给定查询向量和存储在 Elasticsearch 中的文档向量之间的余弦相似度。

```
    # Display the 128-dimension for each face detected
        i = 0
        for face_encoding in face_encodings:
            i += 1
            print("Face", i)
            response = es.search(
                index="faces",
                body={
                    "size": 1,
                    "_source": "face_name",
                    "query": {
                        "script_score": {
                            "query": {
                                "match_all": {}
                            },
                            "script": {
                                "source": "cosineSimilarity(params.query_vector, 'face_e
ncoding')",
                                "params": {
                                    "query_vector": face_encoding.tolist()
                                }
                            }
                        }
                    }
                }
            )
```

假设分数低于 0.92 被认为是未知面孔：

```
    for hit in response['hits']['hits']:
                        # double score=float(hit['_score'])
                        print("score: {}".format(hit['_score']))
                        if float(hit['_score']) > 0.92:
                            print("==> This face  match with ", hit['_source']['face_name'], ",the
score is", hit['_score'])
                        else:
                            print("==> Unknown face")
```

执行上面的 Python 代码：

```
$ pwd
/Users/liuxg/python/face_detection
$ python3 recognizeFaces.py
/Users/liuxg/python/face_detection/images_to_be_recognized/facial-recognition-bl
og-elastic-founders-match.png
Face 1
score: 0.9590121
==> This face  match with  simon ,the score is 0.9590121
Face 2
score: 0.939491
==> This face  match with  uri ,the score is 0.939491
Face 3
score: 0.960626
==> This face  match with  shay ,the score is 0.960626
Face 4
score: 0.92384624
==> This face  match with  steven ,the score is 0.92384624
```


该脚本能够检测出得分匹配度高于 0.92 的所有面孔。

搜寻进阶

面部识别和搜索可以结合使用，以用于高级用例。 你可以使用 Elasticsearch 构建更复杂的查询，例如 geo_queries，query-dsl-bool-query 和 search-aggregations。

例如，以下查询将 cosineSimilarity 搜索应用于 200 公里半径内的特定位置：

```
GET /_search
{
  "query": {
    "script_score": {
      "query": {
    "bool": {
      "must": {
        "match_all": {}
      },
      "filter": {
        "geo_distance": {
          "distance": "200km",
          "pin.location": {
            "lat": 40,
            "lon": -70
          }
        }
      }
    }
  },
       "script": {
                  "source": "cosineSimilarity(params.query_vector, 'face_encoding')",
                   "params": {
                   "query_vector":[
                            -0.14664565,
```

```
                        0.07806452,
                        0.03944433,
                        ...
                        ...
                        ...
                        -0.03167224,
                        -0.13942884
                    ]
                }
            }
        }
    }
}
```

将 cosineSimilarity 与其他 Elasticsearch 查询结合使用，可以无限地实现更复杂的用例。

结论

面部识别可能与许多用例相关，并且你可能已经在日常生活中使用了它。 上面描述的概念可以推广到图像或视频中的任何对象检测，因此你可以将用例扩展到非常大的应用场景。

参考链接：

- https://www.elastic.co/blog/how-to-build-a-facial-recognition-system-using-elasticsearch-and-python

4.2.5 在 Docker 上使用 Elastic Stack 和 Kafka

创作人：刘晓国

你是否考虑分析和可视化地理数据？ 为什么不尝试 Elastic Stack？ 也就是所谓的 ELK（Elasticsearch + Logstash + Kibana）或 Elatic Stack 不仅是 NoSQL 数据库。 它是一个整体系统，可以实时存储，搜索，分析和可视化来自任何来源的数据。在这种情况下，我们将使用有关华沙公共交通位置的开放数据。

本文中，我将介绍如何使用 Elastic Stack 和 Kafka 来监控公共交通的车辆。我们将使用 Docker 来部署所有需要的组件。下面是整个系统的框架图：

整个应用的框架如上：

- 汽车或公交的数据上传到一个数据平台。它提供 REST API 接口来被调用。

- Python 应用定时从 data portal 进行抓取数据，并同时发送到 Kafka
- Kafaka 的数据发送到 Logstash 进行加工，并导入到 Elasticsearch 中
- 在 Kibana 中对数据进行呈现

安装

Python

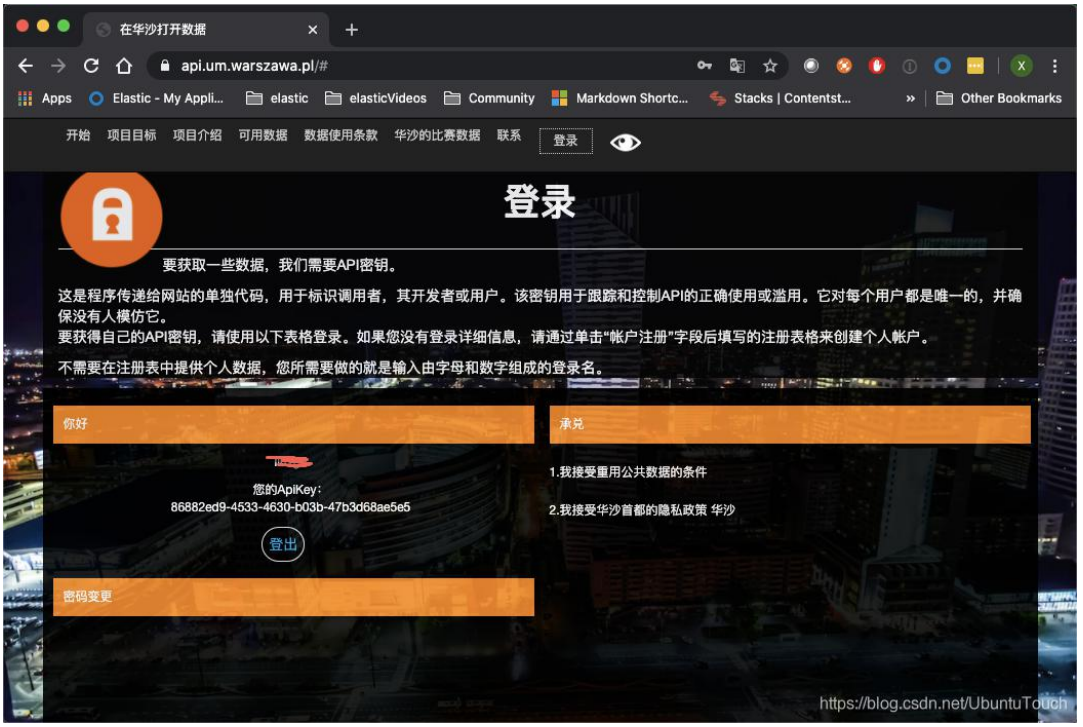
我们有一个应用是用 python 语言写的。你需要安装 python3 来运行该应用。

API key

为了测试这个应用，我们必须得到相应的华沙公共交通信息的 API key。

我们可以在地址 https://api.um.warszawa.pl/# 进行申请。

点击上面的 “登录” 链接，并进行脑力测试：



最终得到如上所示的 API key：86882ed9-4533-4630-b03b-47b3d68ae5e5。这个 key 将在一下的 python 应用中使用。

Elastic Stack 及 Kafka

你需要安装 Docker 来实现 Elastic Stack 及 Kafka 的安装。

本展示的所有的源码可以在地址 https://github.com/liu-xiao-guo/wiadro-danych-kafka-to-es-ztm 进行下载。

docker-compose 包含 Elasticsearch，Kibana，Zookeeper，Kafka，Logstash 和应用程序 Kafka Streams （由于一些原因，在本展示中将不被采用）。

docker-compose.yml

```
version: '3.3'
services:
    elasticsearch:
        image: docker.elastic.co/elasticsearch/elasticsearch:7.7.0
        restart: unless-stopped
        environment:
        - discovery.type=single-node
        - bootstrap.memory_lock=true
        - "ES_JAVA_OPTS=-Xms512m -Xmx512m"
        ulimits:
            memlock:
                soft: -1
                hard: -1
        volumes:
        - esdata:/usr/share/elasticsearch/data
        ports:
        - 9200:9200

    kibana:
        image: docker.elastic.co/kibana/kibana:7.7.0
        restart: unless-stopped
        depends_on:
            - elasticsearch
        ports:
            - 5601:5601
```

```
        volumes:
            - kibanadata:/usr/share/kibana/data

    zookeeper:
        image: 'bitnami/zookeeper:3'
        ports:
            - '2181:2181'
        volumes:
            - 'zookeeper_data:/bitnami'
        environment:
            - ALLOW_ANONYMOUS_LOGIN=yes

    kafka:
        image: 'bitnami/kafka:2'
        ports:
            - '9092:9092'
            - '29092:29092'
        volumes:
            - 'kafka_data:/bitnami'
        environment:
            - KAFKA_CFG_ZOOKEEPER_CONNECT=zookeeper:2181
            - ALLOW_PLAINTEXT_LISTENER=yes
            - KAFKA_CFG_LISTENERS=PLAINTEXT://:9092,PLAINTEXT_HOST://:29092
            - KAFKA_CFG_LISTENER_SECURITY_PROTOCOL_MAP=PLAINTEXT:PLAINTEXT,P
LAINTEXT_HOST:PLAINTEXT
            - KAFKA_CFG_ADVERTISED_LISTENERS=PLAINTEXT://kafka:9092,PLAINTEXT_H
OST://localhost:29092
        depends_on:
            - zookeeper

    ztm_kafka_streams:
```

```
        image: "maciejszymczyk/ztm_stream:1.0"
        environment:
          - APPLICATION_ID_CONFIG=awesome_overrided_ztm_stream_app_id
          - BOOTSTRAP_SERVERS_CONFIG=kafka:9092
        depends_on:
          - kafka

    logstash:
        image: docker.elastic.co/logstash/logstash:7.7.0
        volumes:
            - "./pipeline:/usr/share/logstash/pipeline"
        environment:
            LS_JAVA_OPTS: "-Xmx256m -Xms256m"
        depends_on:
            - elasticsearch
            - kafka

volumes:
    esdata:
        driver: local
    kibanadata:
        driver: local
    zookeeper_data:
        driver: local
    kafka_data:
        driver: local
```

我们在自己电脑的 console 中打入如下的命令：

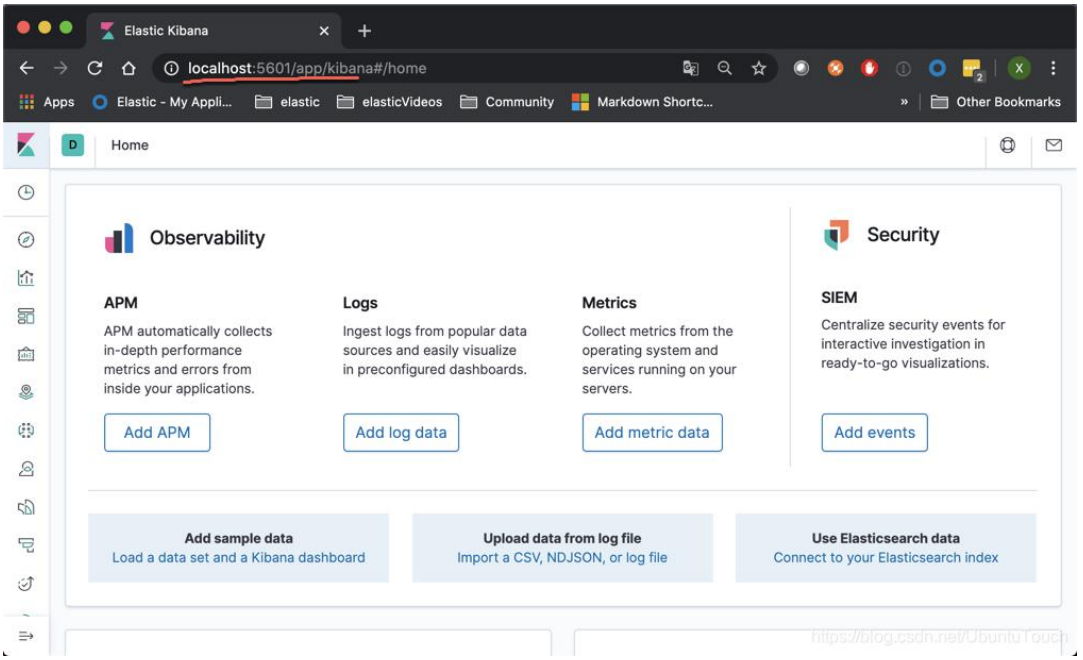
```
docker-compose up
```

我们可以看到如下的画面：

```
 ][.monitoring-logstash] 'pipeline.ordered' is enabled and is likely less effici
ent, consider disabling if preserving event order is not necessary
logstash_1           | [2020-06-02T13:42:14,990][INFO ][logstash.javapipeline
 ][.monitoring-logstash] Starting pipeline {:pipeline_id=>".monitoring-logstash"
, "pipeline.workers"=>1, "pipeline.batch.size"=>2, "pipeline.batch.delay"=>50, "
pipeline.max_inflight"=>2, "pipeline.sources"=>["monitoring pipeline"], :thread=
>"#<Thread:0x4f94e0cd run>"}
logstash_1           | [2020-06-02T13:42:15,007][INFO ][org.apache.kafka.clients
.consumer.internals.ConsumerCoordinator][main][1557c4b996dceab36aaeef298b22a3343
3772413442fe3d18f9cb9d0853b9250] [Consumer clientId=logstash-0, groupId=logstash
] Setting offset for partition ztm-input-0 to the committed offset FetchPosition
{offset=493285, offsetEpoch=Optional[0], currentLeader=LeaderAndEpoch{leader=kaf
ka:9092 (id: 1001 rack: null), epoch=0}}
logstash_1           | [2020-06-02T13:42:15,128][INFO ][logstash.javapipeline
 ][.monitoring-logstash] Pipeline started {"pipeline.id"=>".monitoring-logstash"
}
logstash_1           | [2020-06-02T13:42:15,144][INFO ][logstash.agent
 ] Pipelines running {:count=>2, :running_pipelines=>[:main, :".monitoring-logst
ash"], :non_running_pipelines=>[]}
logstash_1           | [2020-06-02T13:42:15,673][INFO ][logstash.agent
 ] Successfully started Logstash API endpoint {:port=>9600}
elasticsearch_1      | {"type": "server", "timestamp": "2020-06-02T13:42:17,864Z
", "level": "INFO", "component": "o.e.c.r.a.AllocationService", "cluster.name":
"docker-cluster", "node.name": "a122e65f252e", "message": "Cluster health status
```

从上面我们可以看出来 Logstash 已经被成功地启动。

我们在浏览器的地址栏中输入地址 http://localhost:5601



我们可以看到 Kibana 已经成功启动，这也意味着 Elasticsearch 被成功地运行起来了。

配置及运行

Logstash

我们使用如下的 pipeline 来实现对数据的处理：

pipeline/kafka_to_es.conf

```
input {
    kafka {
        topics => "ztm-input"
        bootstrap_servers => "kafka:9092"
        codec => "json"
    }
}

filter {
     mutate {
        convert => {"Lat" => "float"}
        convert => {"Lon" => "float"}

        add_field => ["location", "%{Lat},%{Lon}"]
        remove_field => ["Lat", "Lon"]
     }
}
output {
     stdout {
        codec => rubydebug
     }

     elasticsearch {
        hosts => ["elasticsearch:9200"]
        index => "ztm"
    }
}
```

它从 Kafaka 的 "ztm-input" topic 获取数据，并把相应的 Lat 及 Lon 字段合并成为一个 location 字段。在 output 的部分，我们把数据导入到 Elasticsearch 之中。

Elasticsearch

我们使用了索引生命周期管理机制， 而不是将记录放入诸如 ztm-2020.05.24 之类的索引中。 它使你可以自动执行索引的寿命。 它会自动进行汇总，并根据你配置策略的方式更改索引属性（热-热-冷架构）。 假设我希望在索引达到 1GB 或 30 天过去后进行 rollover，我们在 Kibana 中执行如下的命令：

```
PUT _ilm/policy/ztm_policy
{
  "policy": {
    "phases": {
      "hot":{
        "actions": {
          "rollover": {
            "max_size": "1gb",
            "max_age": "30d"
          }
        }
      }
    }
  }
}
```

你还需要一个模板，该模板具有 ztm_policy 将连接到的适当 mapping。 如果没有 mapping，Elasticsearch 将不会猜测到 location 字段为 geo_point 的数据类型，并且时间字段将是纯文本。

```
PUT _template/ztm_template
{
  "index_patterns": ["ztm-*"],
  "settings": {
    "number_of_shards": 1,
    "number_of_replicas": 0,
    "index.lifecycle.name":"ztm_policy",
    "index.lifecycle.rollover_alias": "ztm"
  },
  "mappings": {
    "properties": {
      "@timestamp": {
        "type": "date"
      },
      "@version": {
        "type": "text",
        "fields": {
          "keyword": {
            "type": "keyword",
            "ignore_above": 256
          }
        }
      },
      "bearing": {
        "type": "float"
      },
```

```
        "brigade": {
          "type": "keyword"
        },
        "distance": {
          "type": "float"
        },
        "lines": {
          "type": "keyword"
        },
        "location": {
          "type": "geo_point"
        },
        "speed": {
          "type": "float"
        },
        "time": {
          "type": "date",
          "format":"MMM dd, yyyy K:mm:ss a"
        },
        "vehicleNumber": {
          "type": "keyword"
        }
      }
    }
}
```

现在该使用适当的别名创建第一个索引了。

```
PUT ztm-000001
{
  "aliases": {
    "ztm": {
      "is_write_index":true
    }
  }
}
```

我们在 Kibana 中运行上面的三个命令。

Python 脚本

首先，我们必须获得所需要的 API key。这个在上面我们已经讲述了。

ztm.py

```
import requests
import json
import time
from kafka import KafkaProducer

token = '86882ed9-4533-4630-b03b-47b3d68ae5e5'
url = 'https://api.um.warszawa.pl/api/action/busestrams_get/'
resource_id = 'f2e5503e927d-4ad3-9500-4ab9e55deb59'
sleep_time = 15
```

```
    bus_params = {
        'apikey':token,
        'type':1,
        'resource_id': resource_id
        }
    tram_params = {
        'apikey':token,
        'type':2,
        'resource_id': resource_id
        }

    while True:
        try:
            r = requests.get(url = url, params = bus_params)
            data = r.json()
            producer = KafkaProducer(bootstrap_servers=['localhost:29092'],
                                     value_serializer=lambda x: json.dumps(x).encode('utf-8'),
                                     key_serializer=lambda x: x
                                     )

            print('Sending records...')
            for record in data['result']:
                print(record)
                future = producer.send('ztm-input', value=record, key=record["VehicleNumber
"].encode('utf-8'))
                result = future.get(timeout=60)
        except:
            print("¯\_(ツ)_/¯")
        time.sleep(sleep_time)
```

上面的代码其实是蛮简单的。它定时从 API portal 获取公交系统的位置信息，并转发到 Kafka。

我们使用如下的命令来运行上面的应用：

python3 ztm.py

```
 10:16:50', 'Lat': 52.1972982, 'Brigade': '012'}
{'Lines': '141', 'Lon': 21.00733, 'VehicleNumber': '2209', 'Time': '2020-06-02 1
6:02:36', 'Lat': 52.195278, 'Brigade': '1'}
{'Lines': '706', 'Lon': 20.905991, 'VehicleNumber': '2210', 'Time': '2020-06-02
16:01:01', 'Lat': 52.13681, 'Brigade': '1'}
{'Lines': '187', 'Lon': 21.048264, 'VehicleNumber': '2212', 'Time': '2020-06-02
16:01:42', 'Lat': 52.178185, 'Brigade': '7'}
{'Lines': '401', 'Lon': 20.862062, 'VehicleNumber': '2215', 'Time': '2020-06-02
16:01:52', 'Lat': 52.190331, 'Brigade': '010'}
{'Lines': '317', 'Lon': 21.040525, 'VehicleNumber': '2216', 'Time': '2020-06-02
16:02:48', 'Lat': 52.1707, 'Brigade': '06'}
{'Lines': '187', 'Lon': 20.861298, 'VehicleNumber': '2217', 'Time': '2020-06-02
16:01:53', 'Lat': 52.191597, 'Brigade': '1'}
{'Lines': '189', 'Lon': 21.040316, 'VehicleNumber': '2218', 'Time': '2020-06-02
16:02:52', 'Lat': 52.171017, 'Brigade': '10'}
{'Lines': '105', 'Lon': 21.024334, 'VehicleNumber': '2219', 'Time': '2020-06-02
16:01:45', 'Lat': 52.242054, 'Brigade': '4'}
{'Lines': '184', 'Lon': 20.970509, 'VehicleNumber': '2220', 'Time': '2020-06-02
16:02:32', 'Lat': 52.219357, 'Brigade': '4'}
{'Lines': '188', 'Lon': 21.05415, 'VehicleNumber': '2221', 'Time': '2020-06-02 1
6:02:29', 'Lat': 52.226475, 'Brigade': '1'}
{'Lines': '188', 'Lon': 21.082222, 'VehicleNumber': '2222', 'Time': '2020-06-02
16:02:40', 'Lat': 52.240078, 'Brigade': '3'}
{'Lines': '188', 'Lon': 20.978231, 'VehicleNumber': '2224', 'Time': '2020-06-02
```

这个时候，我们可以在屏幕上看到所获得很多的关于公交系统车辆的信息。

我们可以转到运行 docker-compopse up 命令的那个 console，我们可以看到如下的信息：

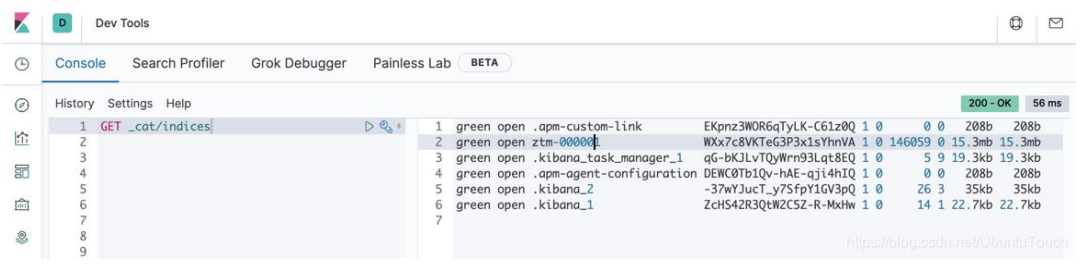
```
logstash_1  | }
logstash_1  | {
logstash_1  |             "Lines" => "195",
logstash_1  |              "Time" => "2020-06-02 16:00:15",
logstash_1  |          "@version" => "1",
logstash_1  |     "VehicleNumber" => "1062",
logstash_1  |          "location" => "52.148994,21.059315",
logstash_1  |        "@timestamp" => 2020-06-02T14:03:27.699Z,
logstash_1  |            "Brigade" => "3"
logstash_1  | }
logstash_1  | {
logstash_1  |             "Lines" => "328",
logstash_1  |              "Time" => "2020-06-02 16:02:48",
logstash_1  |          "@version" => "1",
logstash_1  |     "VehicleNumber" => "1063",
logstash_1  |          "location" => "52.150848,20.994638",
logstash_1  |        "@timestamp" => 2020-06-02T14:03:27.699Z,
logstash_1  |            "Brigade" => "03"
logstash_1  | }
logstash_1  | {
logstash_1  |             "Lines" => "365",
logstash_1  |              "Time" => "2020-06-02 16:02:49",
logstash_1  |          "@version" => "1",
logstash_1  |     "VehicleNumber" => "1064",
```


它表明我们的 Logstash 是在正常工作。

在 Kibana 中展示

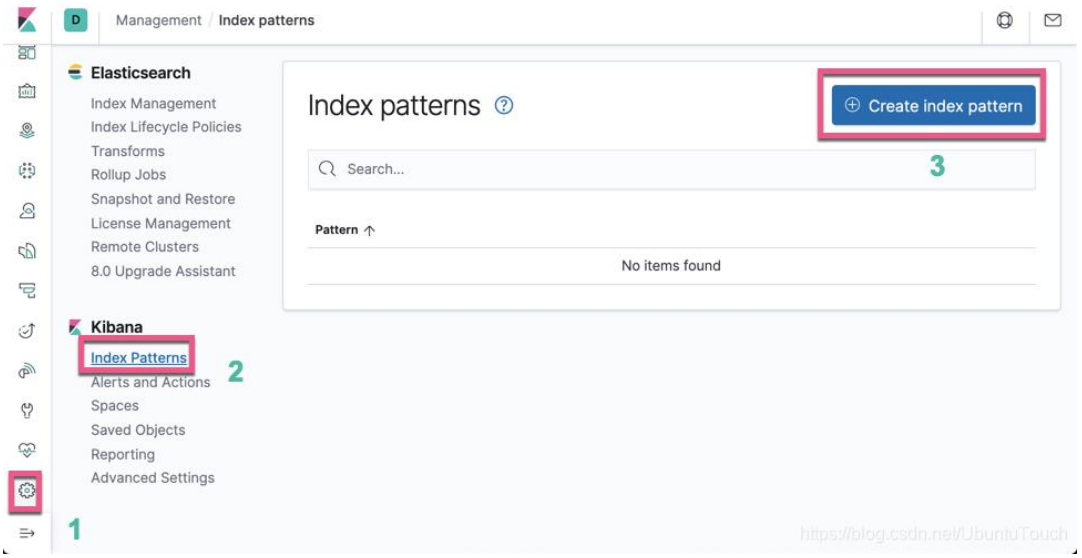
打开 Kibana，并使用如下的命令：

```
GET _cat/indices
```

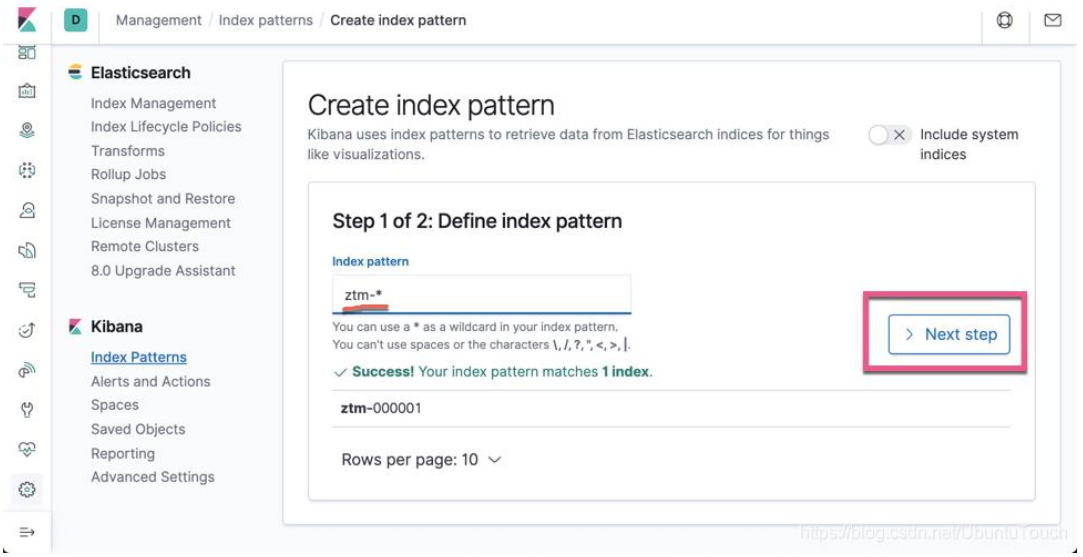


从上面，我们可以看到一个叫做 ztm-000001 的索引，并且它里面含有已经收集上来的车辆信息。

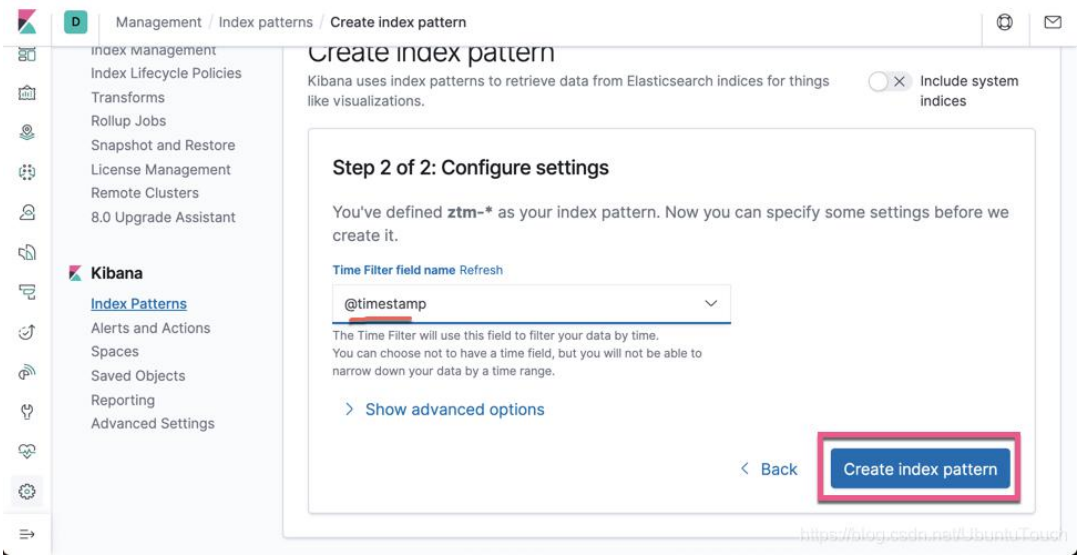
为了分析这个索引，我们必须创建一个 index pattern:



点击 Create index pattern:

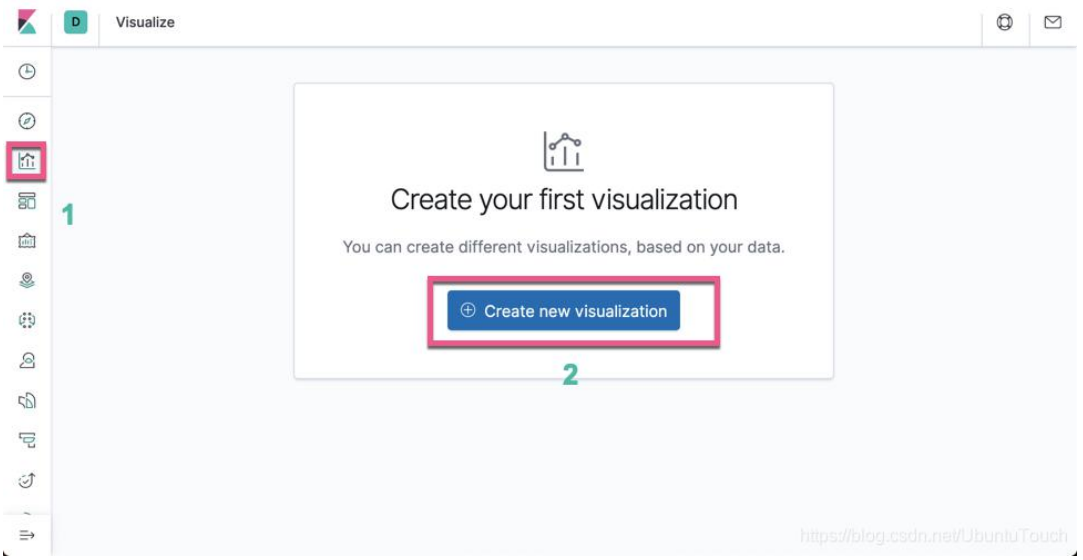


点击 Next step：

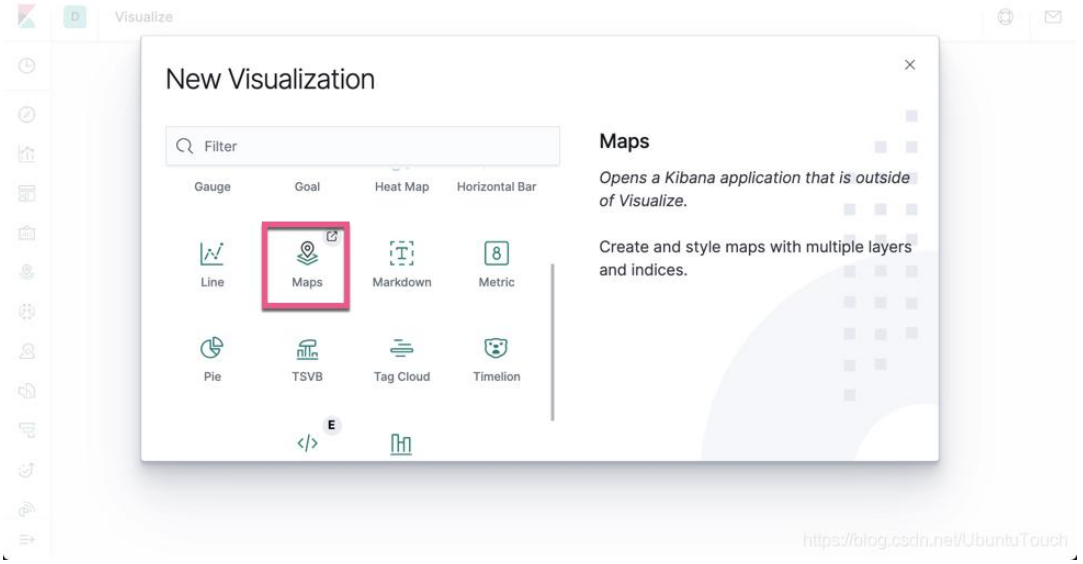


点击上面的 Create index pattern 按钮。这样就完成了创建 index pattern。

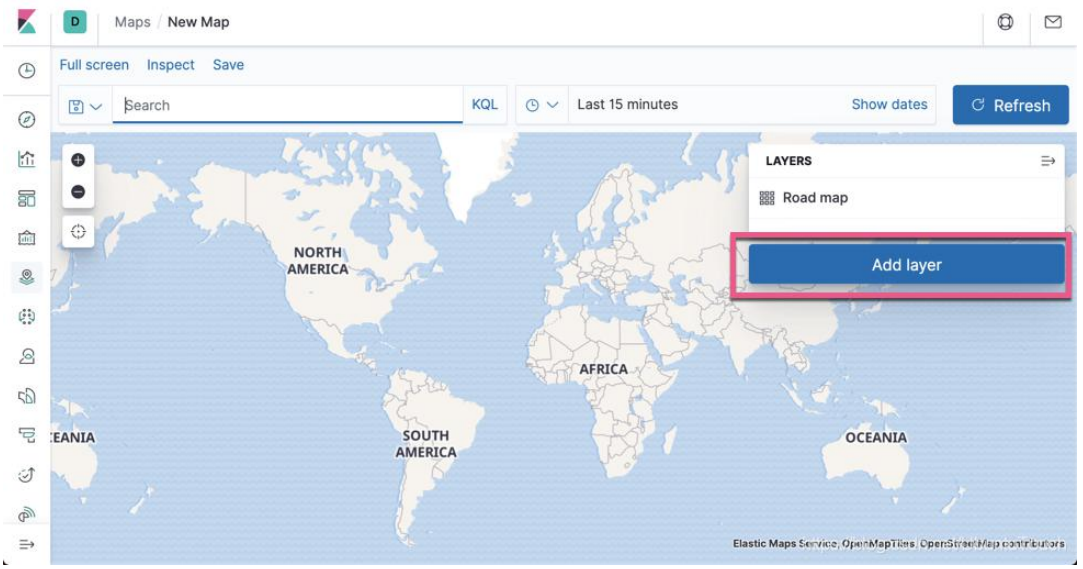
为了对数据可视化，我们点击 Visualization:



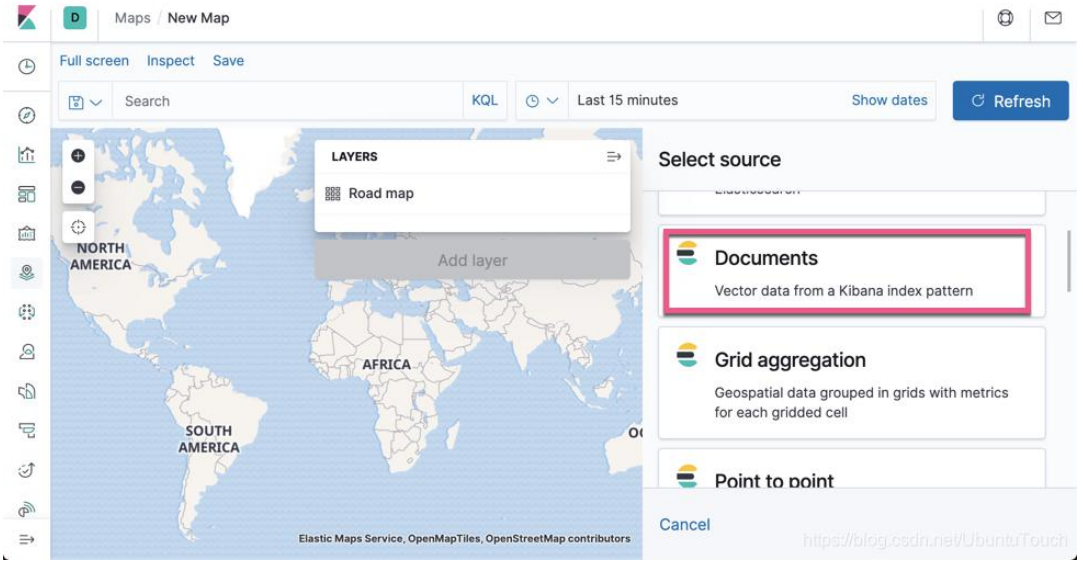
点击上面的 Create new visualization：



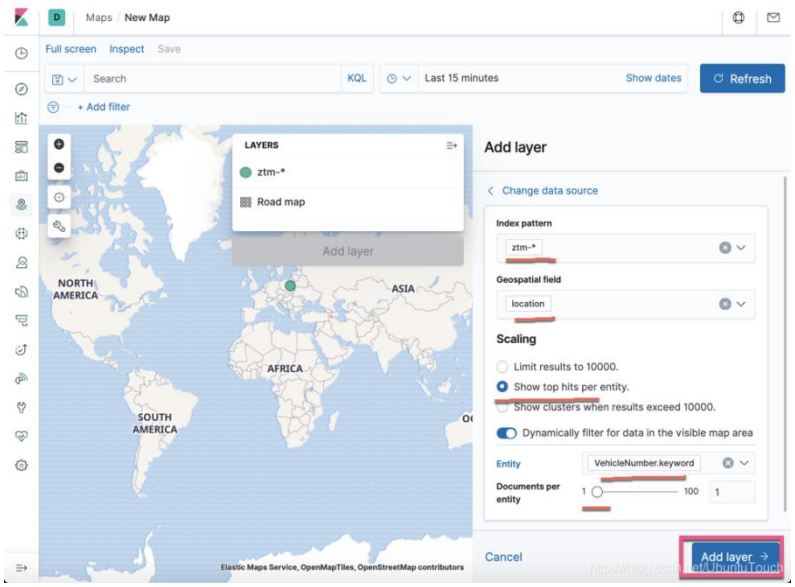
点击 Maps：



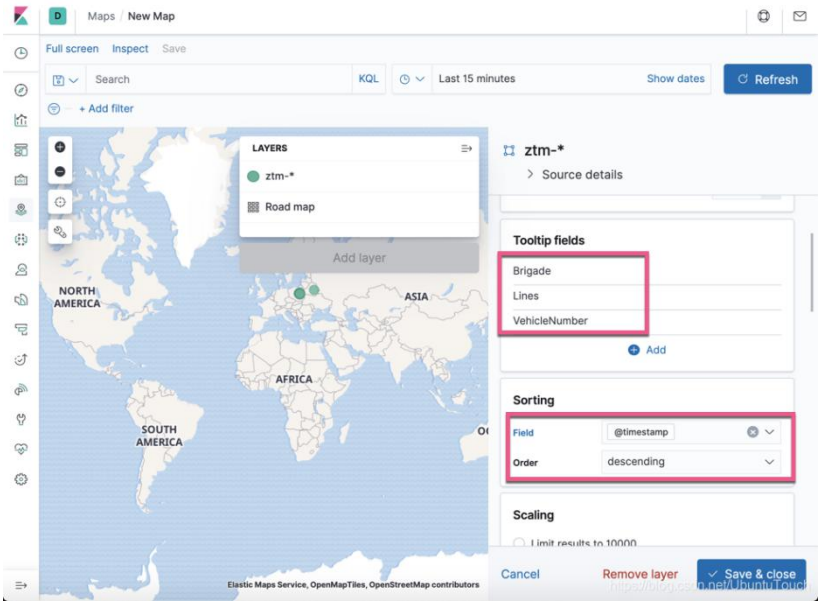
点击 Add layer:



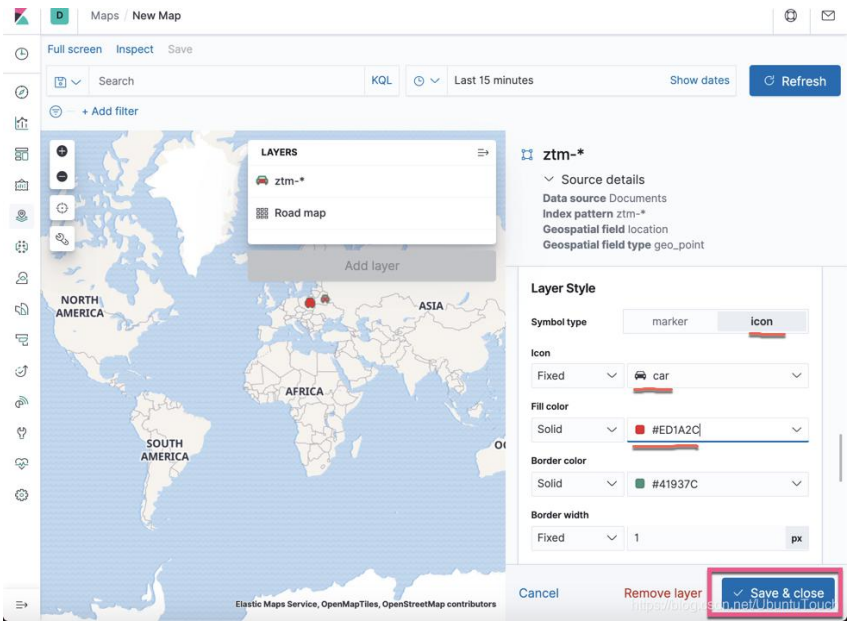
点击 Documents:



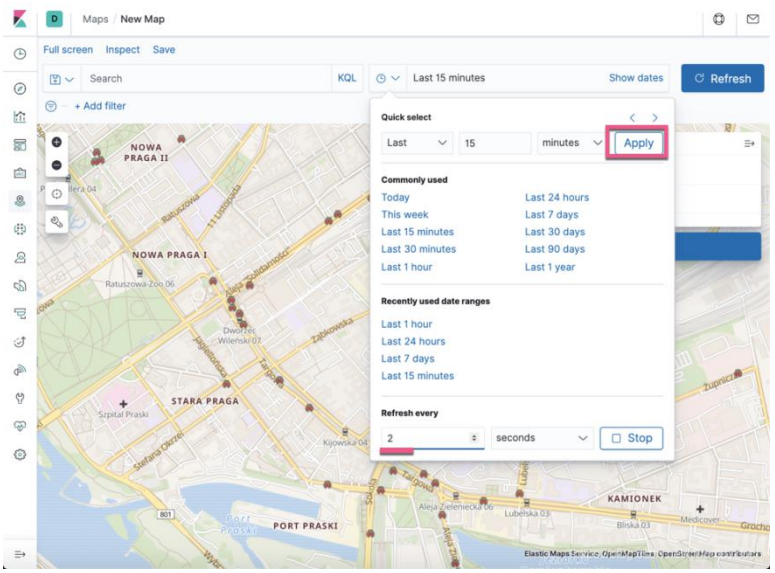
点击 Add layer:



向下滚动：

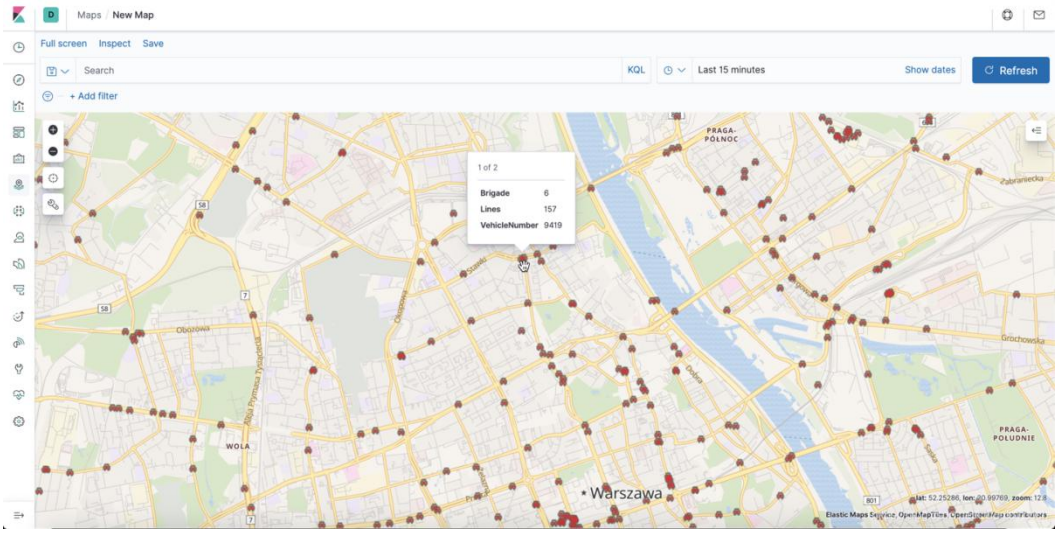


点击上面的 Save & close 按钮：

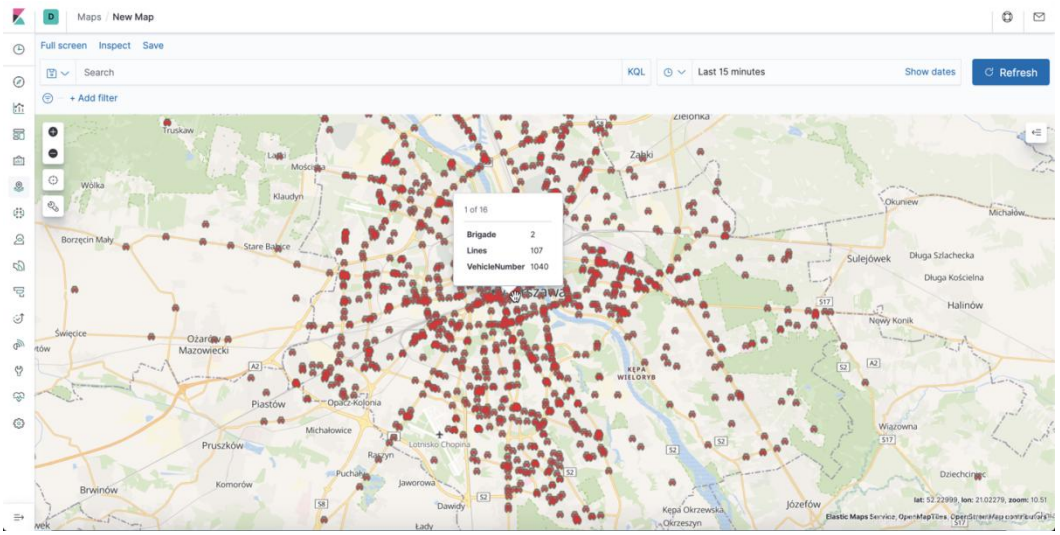


在上面，我们配置每隔 2 秒自动获取数据。点击 Apply 按钮。

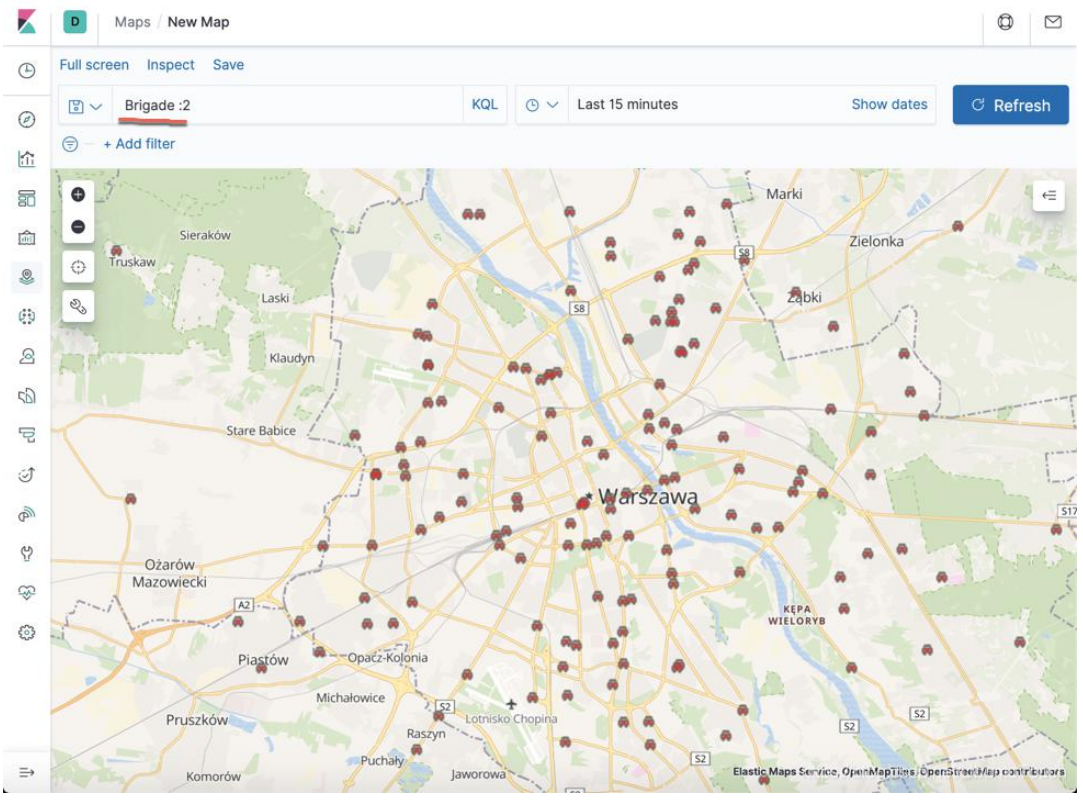
我们聚焦华沙地区：



这样在地图上，我们可以清楚地看到每个车辆的运行情况。



我们甚至可以针对一个 Brigade 进行搜索：



参考链接：

- https://medium.com/@zorteran/how-to-visualize-public-transport-using-kibana-elasticserach-logstash-elastic-stack-and-kafka-eabc6975255a

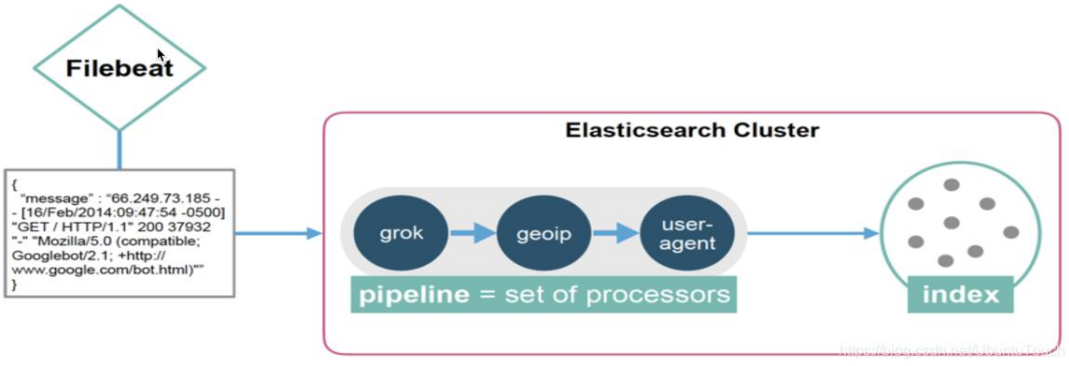
4.2.6 运用 Elastic Stack 分析 COVID-19 数据

创作人：刘晓国

目前 COVID-19 有很多的地方发表数据。

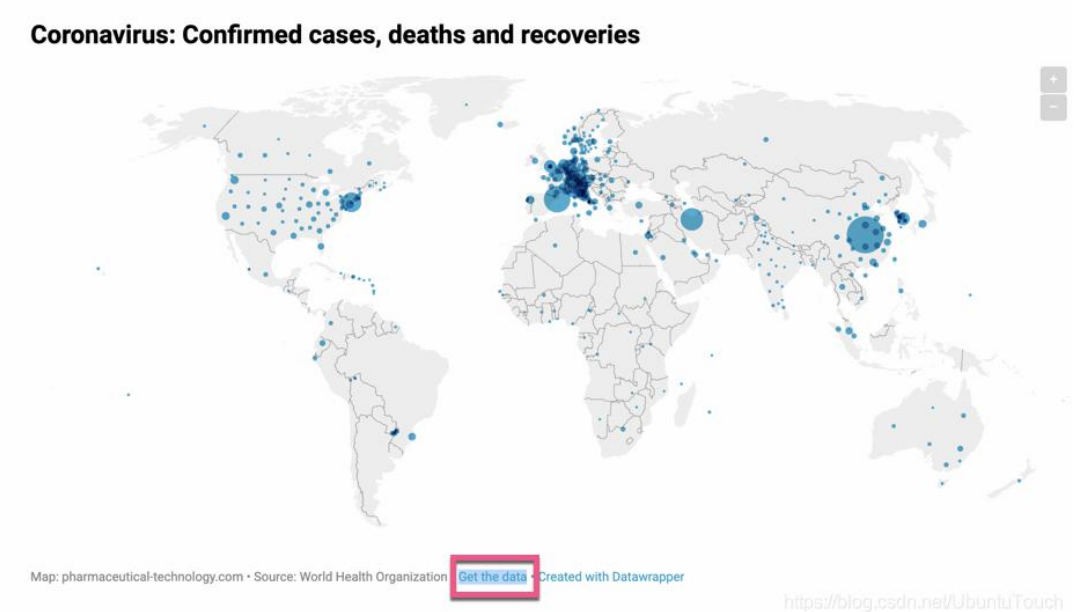
在本文中，我们使用 Elastic Stack 来分析相关的数据，并对数据进行分析。我们将使用 Kibana 来对所有的数据进行可视化分析。

特别指出的是：我们将使用 Elasticsearch 的 processors 来对我们的数据进行解析。



数据来源

我们将使用在地址：https://www.datawrapper.de/_/Gnfyw/下载我们想要的数据。我们点击链接 Get the data 来下载我们想要的数据。



我们下载后的数据就像是下面的：

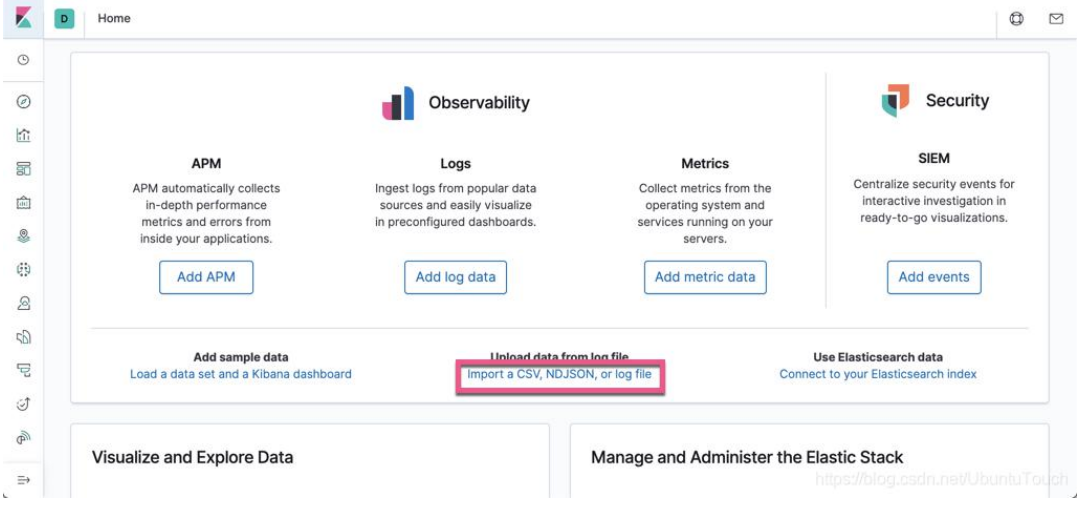
Lat	Lon	Title	Province/State	X.1	Country/Region	Confirmed	Deaths	Recovered
31.1517252	112.8783222	Hubei, China	Hubei	,	China	67801	3160	60323
29.0000001	119.9999999	Zhejiang, China	Zhejiang	,	China	1240	1	1221
23.1357694	113.1982688	Guangdong, China	Guangdong	,	China	1428	8	1333
34.0000001	113.9999999	Henan, China	Henan	,	China	1274	22	1250
27.6662087	111.7487063	Hunan, China	Hunan	,	China	1018	4	1014
32	117	Anhui, China	Anhui	,	China	990	6	984
28	116	Jiangxi, China	Jiangxi	,	China	936	1	934
29.5585712	106.5492822	Chongqing, China	Chongqing	,	China	578	6	570
30.5000001	102.4999999	Sichuan, China	Sichuan	,	China	545	3	536
36.0000001	118.9999999	Shandong, China	Shandong	,	China	768	7	752
33.0000001	119.9999999	Jiangsu, China	Jiangsu	,	China	636	0	631
31.2252985	121.4890497	Shanghai, China	Shanghai	,	China	414	4	330
39.906217	116.3912757	Beijing, China	Beijing	,	China	558	8	401
26.5450001	117.842778	Fujian, China	Fujian		China	318	1	296

上面的数据是一个 csv 格式的文件。我们把下载后的数据存入到一个我们喜欢的目录中，并命令为 covid19.csv 。针对 CSV 格式的数据导入，我们可以采用 Logstash 来把它导入到 Elasticsearch 中。具体如何操作，我们可以参照以下文章：

- Logstash：导入 zipcode CSV 文件和 Geo Search 体验
  https://blog.csdn.net/UbuntuTouch/article/details/100606857

- Logstash: 应用实践 - 装载 CSV 文档到 Elasticsearch
  https://blog.csdn.net/UbuntuTouch/article/details/100606857

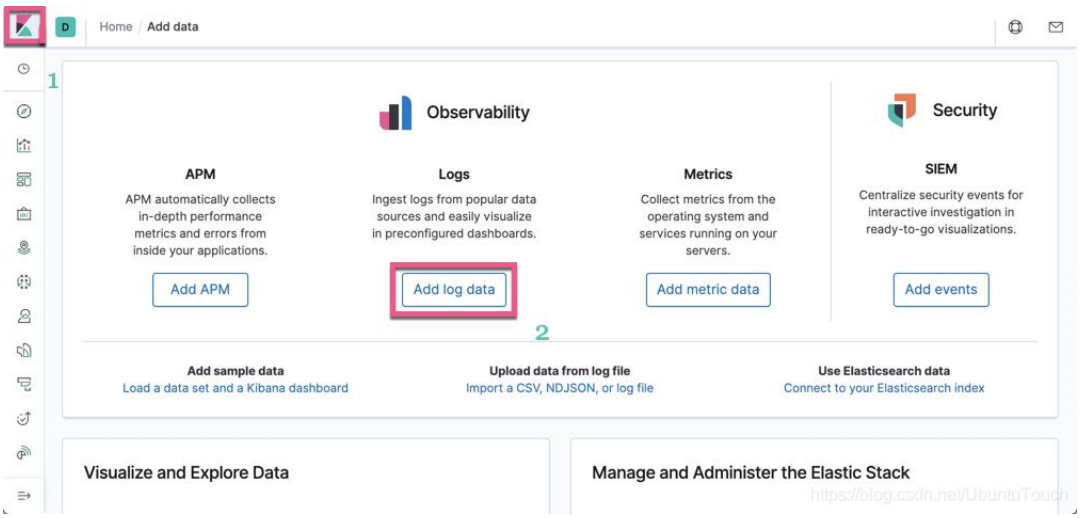
当然我们也可以直接通过 Kibana 提供的便利，直接把数据导入到 Elasticsearch 中：



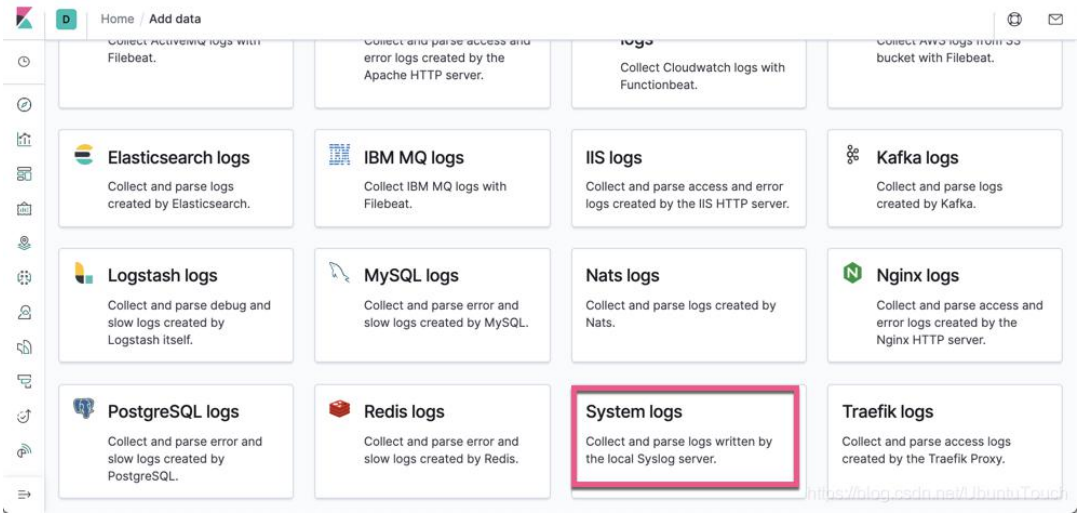
针对我们今天的这个 COVID-19 数据，我们将使用 Filebeat 来对它处理。

安装

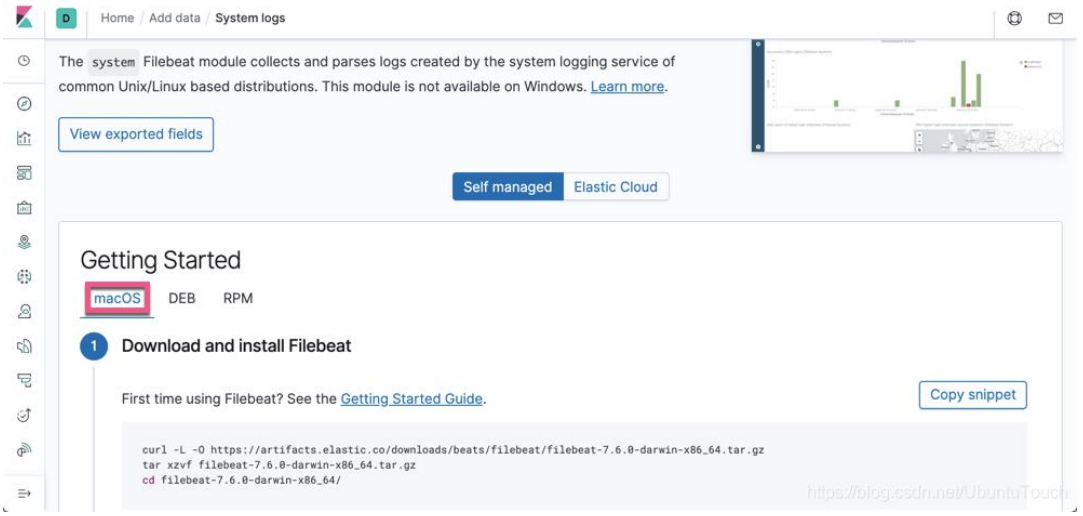
- 安装 Elasticsearch 及 Kibana
- 安装 Filebeat
- 打开 Kibana



虽然 csv 部署 log 范畴，但是在上面的 logs 里含有针对各个操作系统的 filebeat 的安装指南。我们点击上面的 Add log data 按钮：



我们点击 System logs：



在上面我们选择我们的操作系统，并按照上面的安装指令来完成和我们 Elasticsearch 版本相匹配的 Filbeat 的安装。我们可以不启动相应的模块，只做相应的安装即可。

配置 Filebeat 及导入数据到 Elasticsearch

为了能够把我们的数据导入到 Elasticsearch 中，我们可以采用 Filebeat。为此，我们必须来配置 filebeat。我们首先来创建一个定制的 filebeat_covid19.yml 配置文件，并把这个文件存于和我们上面的 covid19.csv 同一个目录中：

filebeat_covid19.yml

```
filebeat.inputs:
- type: log
  paths:
    - /Users/liuxg/data/covid19/covid19.csv
  exclude_lines: ['^Lat']

output.elasticsearch:
  hosts: ["http://localhost:9200"]
  index: covid19

setup.ilm.enabled: false
setup.template.name: covid19
setup.template.pattern: covid19
```

在上面我们定义了一个 type 为 log 的 filebeat.inputs。我们定了我们的 log 的路径。你需要根据自己的实际路径来修改上面的路径。

我们定义了数据的 index 为 covid19 的索引。值得注意的是，由于 csv 文件的第

一行是数据的 header，我们需要去掉这一行。为此，我们采用了 exclude_lines: ['^Lat'] 来去掉第一行。

等我们定义好上面的配置后，我们运行如下的命令：

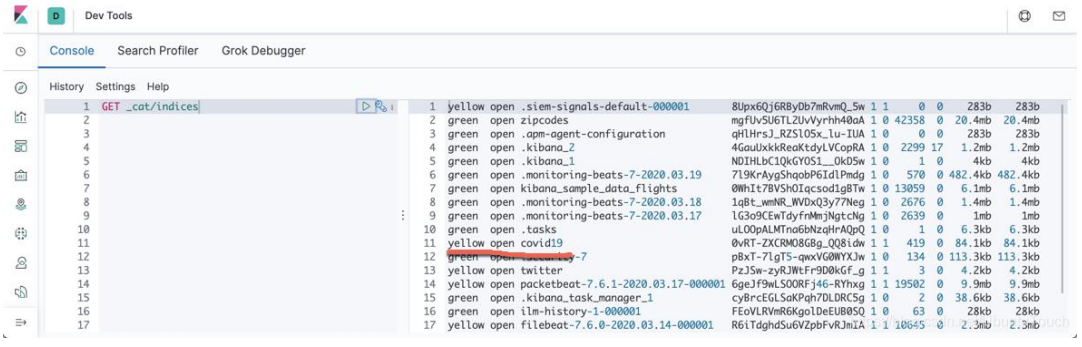
```
./filebeat -e -c ~/data/covid19/filebeat_covid19.yml
```

上面的命令将把我们的数据导入到 Elasticsearch 中。

Filebeat 的 registry 文件存储 Filebeat 用于跟踪上次读取位置的状态和位置信息。如果由于某种原因，我们想重复对这个 csv 文件的处理，我们可以删除如下的目录：

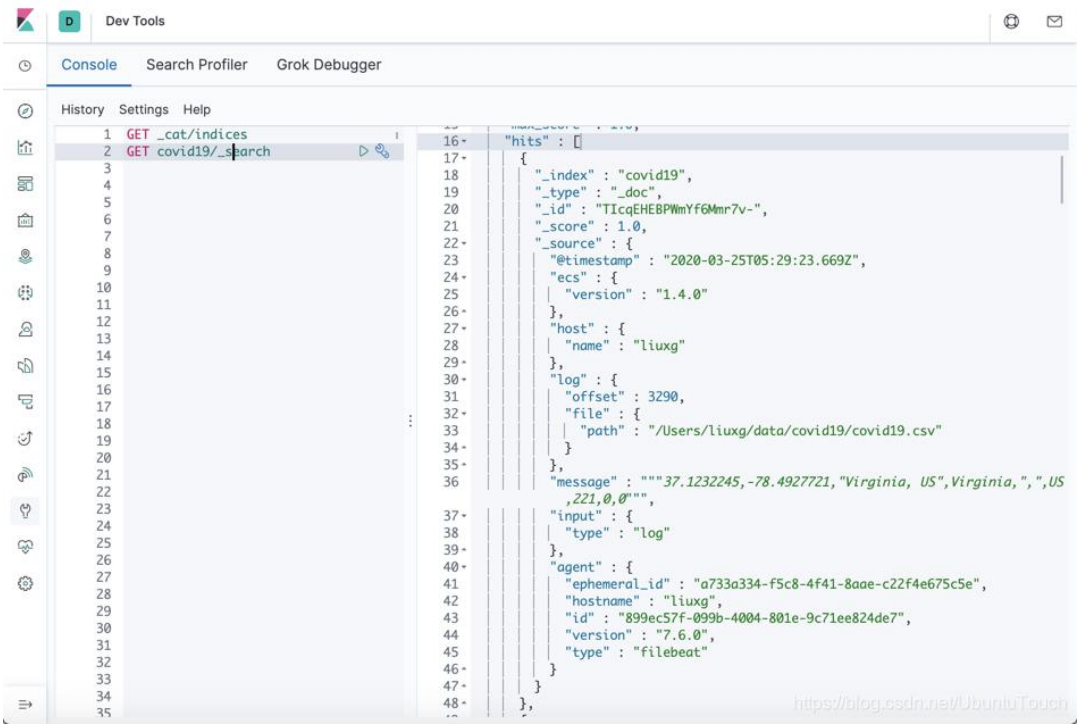
- data/registry 针对 .tar.gz and .tgz 归档文件安装
- /var/lib/filebeat/registry 针对 DEB 及 RPM 安装包
- c:\ProgramData\filebeat\registry 针对 Windows zip 文件

如果上面的命令成功后，我们可以在 Kibana 中查看新生产的 covid19 索引：



如果可以对数据进行查询：

```
GET covid19/_search
```



在上面，它显示了 message 字段。

显然这个数据是最原始的数据，它并不能让我们很方便地对这个数据进行分析。那么我们该如何处理呢？我们可以通过 Elasticsearch 的 ingest node 提供的强大的 processors 来帮我们处理这个数据。

利用 Processors 来加工数据

去掉无用的字段

在我们的文档里，我们可以看到有很多我们并不想要的字段，比如 ecs， host，log 等等。我们想把这些字段去掉，那么我们该如何做呢？我们可以通过定义一 个 pipleline 来帮我们处理。为此，我们定义一个如下的 pipeline:

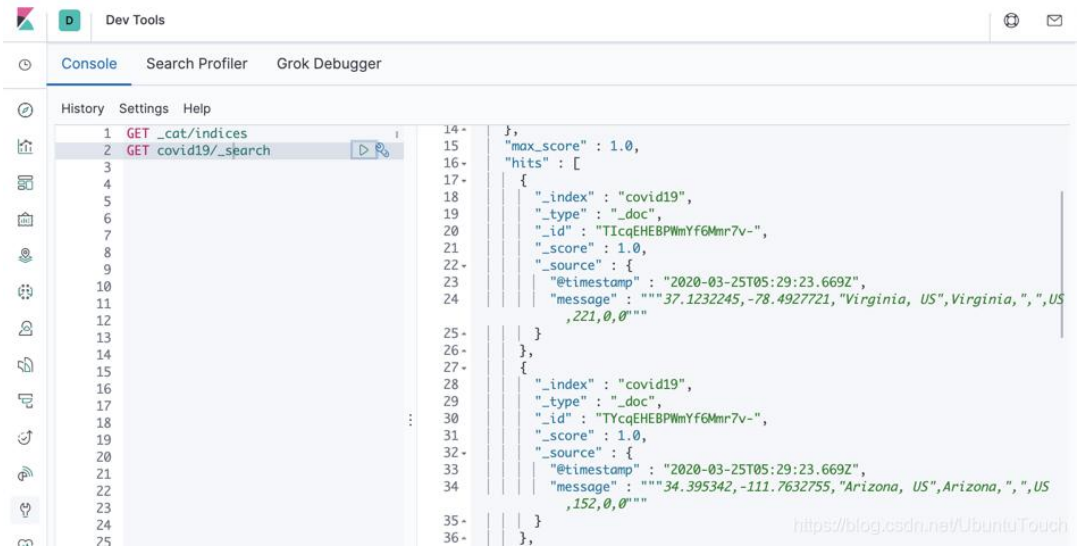
```
PUT _ingest/pipeline/covid19_parser
{
  "processors": [
    {
      "remove": {
        "field": ["log", "input", "ecs", "host", "agent"],
        "if": "ctx.log != null && ctx.input != null && ctx.ecs != null && ctx.host != null &
& ctx.agent != null"
      }
    }
  ]
}
```

上面的 pipeline 定义了一个叫做 remove 的 processor。它检查 log，input, ecs, host 及 agent 都不为空的情况下，删除字段 log, input，ecs, host 及 agent。我们在 Kibana 中执行上面的命令。

为了能够使得我们的 pipleline 起作用，我们通过如下指令来执行：

```
POST covid19/_update_by_query?pipeline=covid19_parser
```

当我们执行完上面的指令后，我们重新查看我们的文档：



在上面我们可以看出来，所有的我们不想要的字段都已经被去掉了

替换引号

我们可以看到导入的 message 数据为：

```
"""37.1232245,-78.4927721,"Virginia, US",Virginia,",",US,221,0,0"""
```

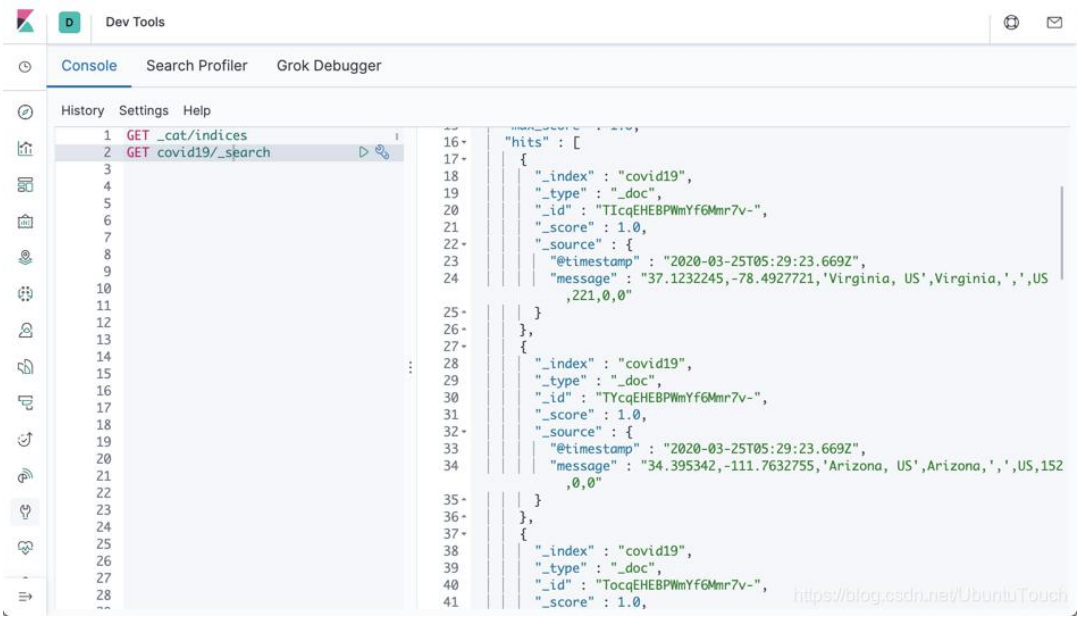
显然，这里的数据有很多的引号"字符，我们想把这些字符替换为符号'。为此，我们需要 gsub processors 来帮我们处理。我重新修改我们的 pipeline:

```
PUT _ingest/pipeline/covid19_parser
{
  "processors": [
    {
      "remove": {
        "field": ["log", "input", "ecs", "host", "agent"],
        "if": "ctx.log != null && ctx.input != null && ctx.ecs != null && ctx.host != null &
& ctx.agent != null"
      }
    },
    {
      "gsub": {
        "field": "message",
        "pattern": "\"",
        "replacement": "'"
      }
    }
  ]
}
```

在 Kibana 中运行上面的指令，并同时执行：

```
POST covid19/_update_by_query?pipeline=covid19_parser
```

经过上面的 pipeline 的处理后，我们重新来查看我们的文档：



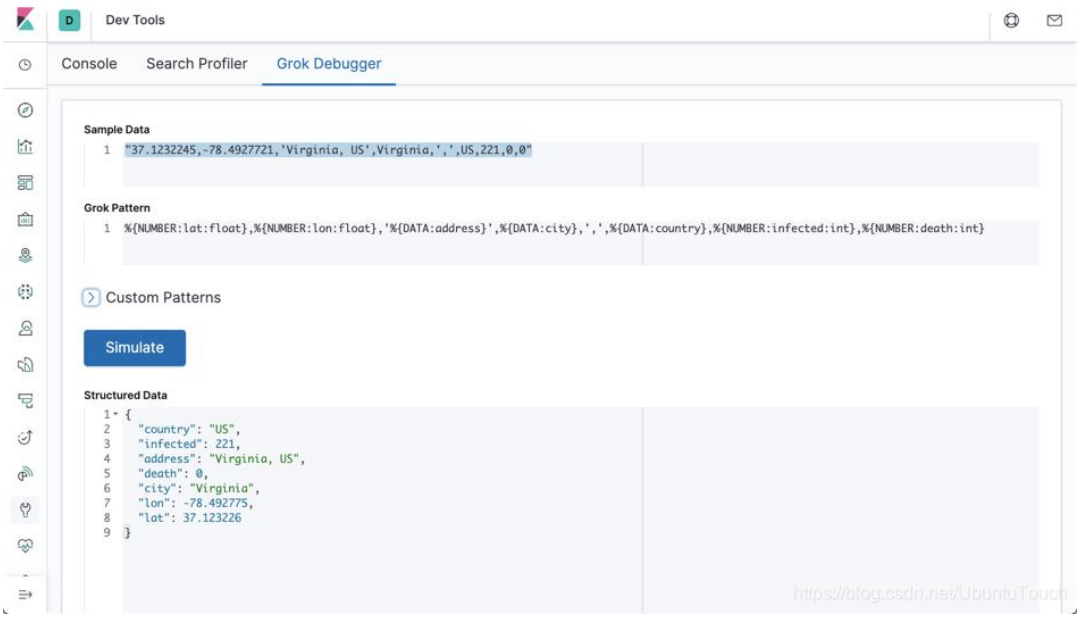
从上面的显示中，我们也看出来我们已经成功都去掉了引号。我们的 message 的信息如下：

```
"37.1232245,-78.4927721,'Virginia, US',Virginia,',',US,221,0,0"
```

解析信息

在上面我们已经很成功地把我们的信息转换为我们所希望的数据类型。接下来我们来使用 grok 来解析我们的数据。grok 的数据解析，基本上是一种正则解析的方法。我们首先使用 Kibana 所提供的 Grok Debugger 来帮助我们分析数据。我们将使用如下的 grok pattern 来解析我们的 message:

%{NUMBER:lat:float},%{NUMBER:lon:float},'%{DATA:address}',%{DATA:city},',',%{DATA:country},%{NUMBER:infected:int},%{NUMBER:death:int}



我们点击 Grok Debugger，并把我们的相应的文档拷入到相应的输入框中，并用上面的 grok pattern 来解析数据。上面显示，它可以帮我们成功地解析我们想要的数据。显然这个被解析的信息更适合我们做数据的分析。为此，我们需要重新修改 pipeline：

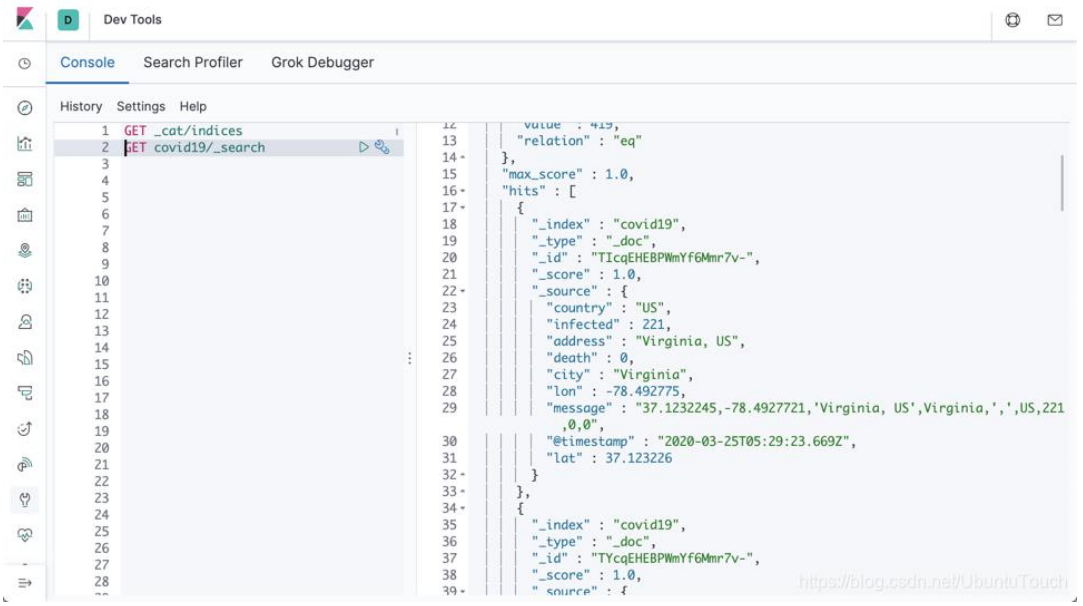
```
PUT _ingest/pipeline/covid19_parser
{
  "processors": [
    {
      "remove": {
        "field": ["log", "input", "ecs", "host", "agent"],
        "if": "ctx.log != null && ctx.input != null && ctx.ecs != null && ctx.host != null && ctx.agent != null"
```

```
        }
      },
      {
        "gsub": {
          "field": "message",
          "pattern": "\"",
          "replacement": "'"
        }
      },
      {
       "grok": {
          "field": "message",
          "patterns": [
            "%{NUMBER:lat:float},%{NUMBER:lon:float},'%{DATA:address}',%{DATA:city},',',%{DATA:country},%{NUMBER:infected:int},%{NUMBER:death:int}"
          ]
        }
      }
    ]
}
```

我们运行上面的 pipeline，并使用如下的命令来重新对数据进行分析：

```
POST covid19/_update_by_query?pipeline=covid19_parser
```

我们重新来查看文档：



在上面我们可以看到新增加的 country，infected，address 等等的字段。

添加 location 字段

在上面我们可以看到 lon 及 lat 字段。这些字段是文档的经纬度信息。这些信息并不能为我们所使用，因为首先他们是分散的，并不处于一个通过叫做 location 的字段中。为此，我们需要创建一个新的 location 字段。为此我们更新 pipeline 为：

```
PUT _ingest/pipeline/covid19_parser
{
  "processors": [
    {
      "remove": {
        "field": ["log", "input", "ecs", "host", "agent"],
```

```
            "if": "ctx.log != null && ctx.input != null && ctx.ecs != null && ctx.host != null &
& ctx.agent != null"
          }
        },
        {
          "gsub": {
            "field": "message",
            "pattern": "\"",
            "replacement": "'"
          }
        },
        {
         "grok": {
            "field": "message",
            "patterns": [
              "%{NUMBER:lat:float},%{NUMBER:lon:float},'%{DATA:address}',%{DATA:city},',',%{D
ATA:country},%{NUMBER:infected:int},%{NUMBER:death:int}"
            ]
          }
        },
        {
          "set": {
            "field": "location.lat",
            "value": "{{lat}}"
          }
        },
        {
          "set": {
            "field": "location.lon",
            "value": "{{lon}}"
          }
```

```
      }
    ]
  }
```

在上面我们设置了一个叫做 location.lat 及 location.lon 的两个字段。它们的值分别是 {{lat}} 及 {{lon}}。我们执行上面的命令。

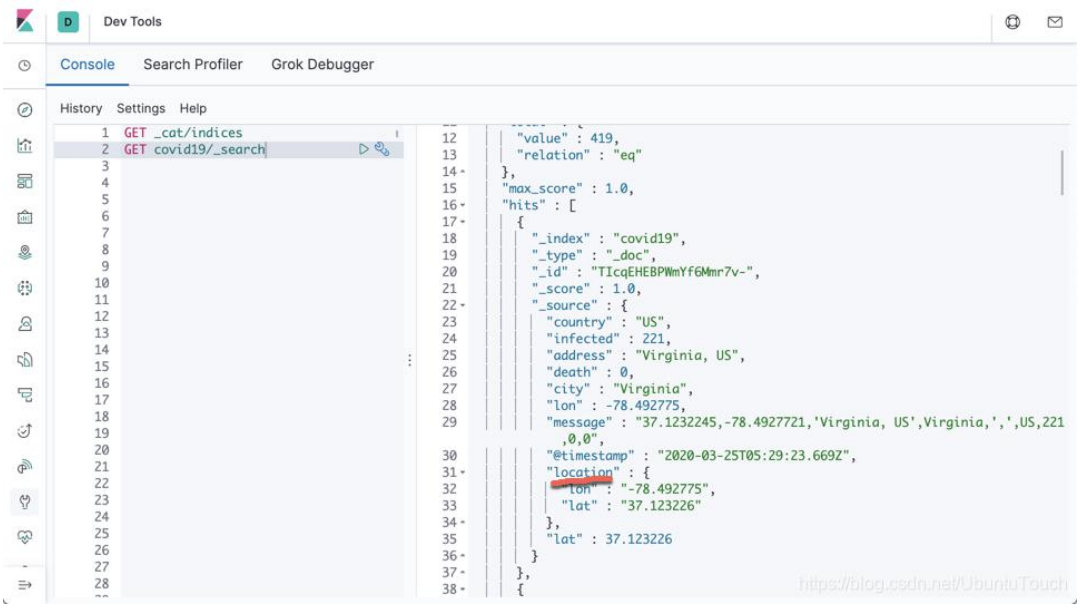
由于 location 是一个新增加的字段，在默认的情况下，它的两个字段都会被 Elasticsearch 设置为 text 的类型。为了能够让我们的数据在地图中进行显示，它必须是一个 geo_point 的数据类型。为此，我们必须通过如下命令来设置它的数据类型：

```
PUT covid19/_mapping
{
  "properties": {
    "location": {
      "type": "geo_point"
    }
  }
}
```

执行上面的指令，我们再使用如下的命令来对我们的数据重新进行处理：

```
POST covid19/_update_by_query?pipeline=covid19_parser
```

等执行完上面的命令后，我们重新来查看我们的文档：



从上面我们可以看到一个叫做 location 的新字段。它含有 lon 及 lat 两个字段。我们同时也可以查看 covid19 的 Mapping。

```
GET covid19/_mapping
```

我们可以发现 Location 的数据类型为：

```
"location" : {
      "type" : "geo_point"
 }
```

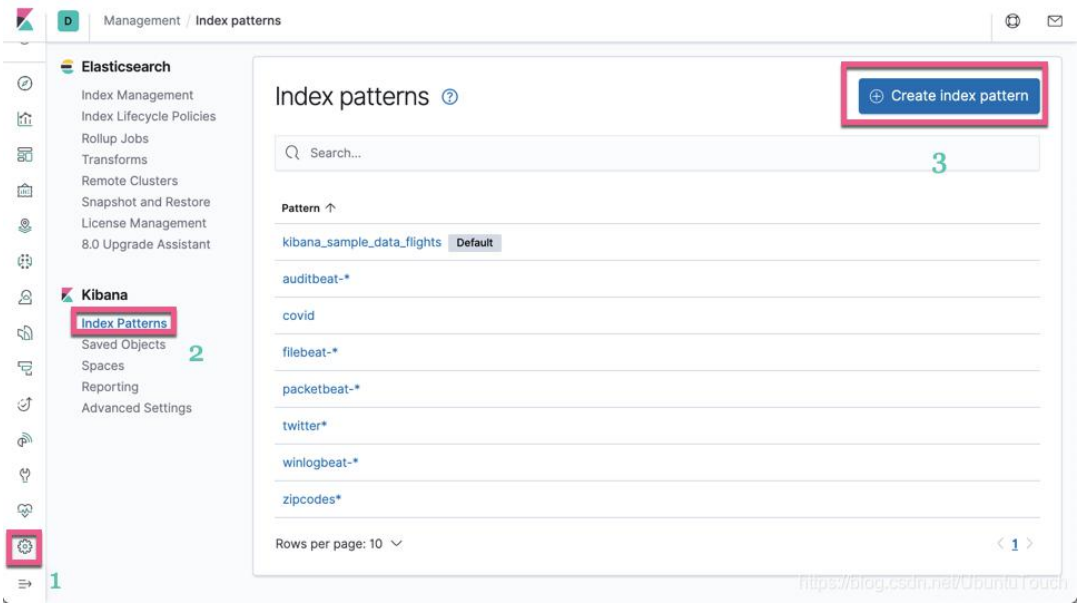
它显示 location 的数据类型是对的。

到目前为止，我们已经成功地把数据导入到 Elasticsearch 中。我们接下来针对 covid19 来进行数据分析。

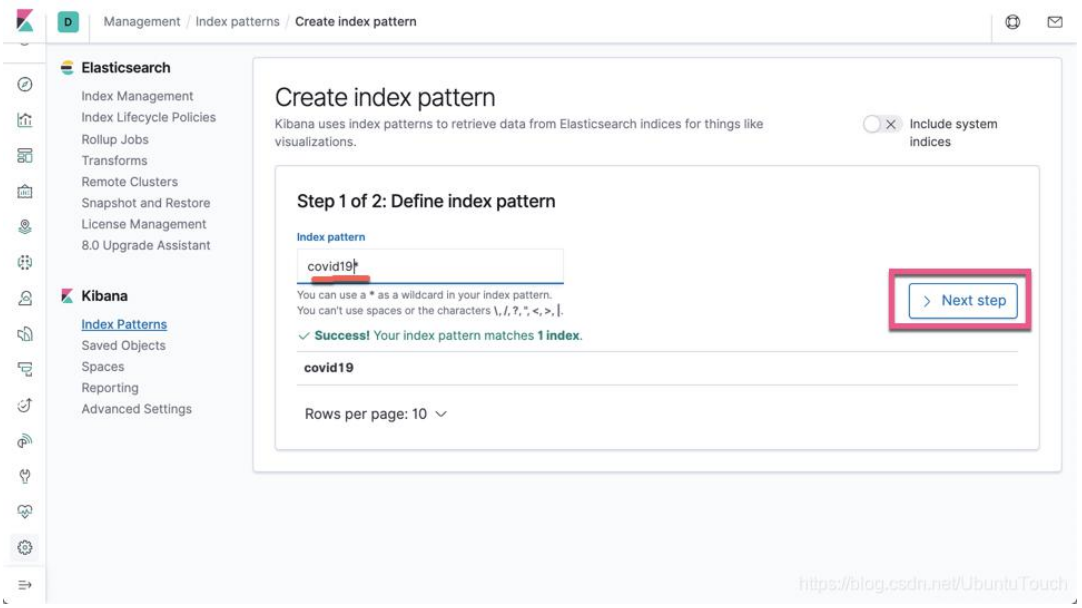
展示并分析数据

创建 index pattern

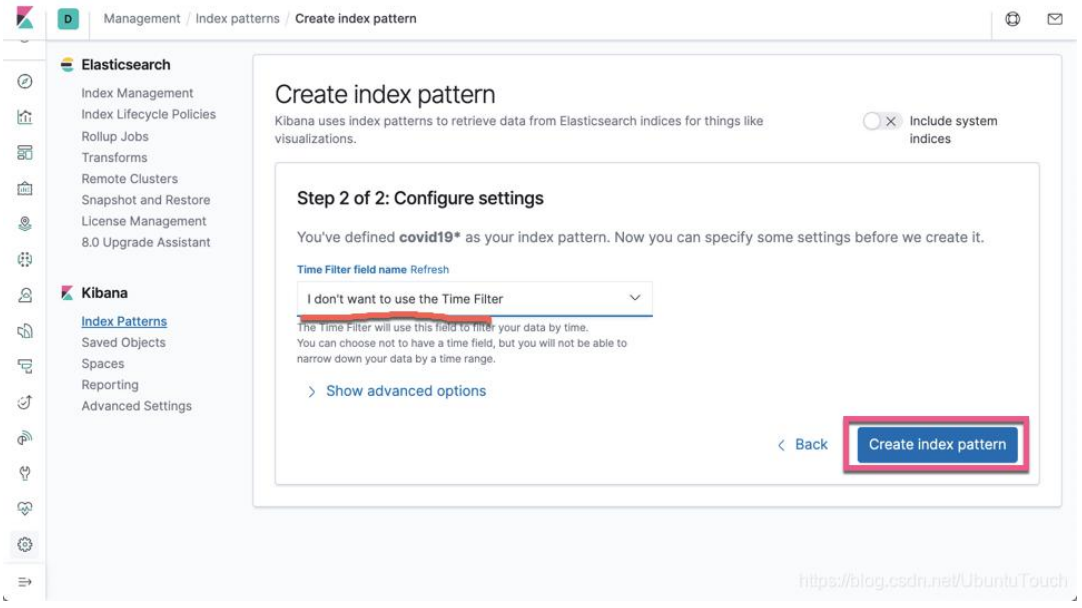
为了对 covid19 进行分析，我们必须创建一个 index pattern：



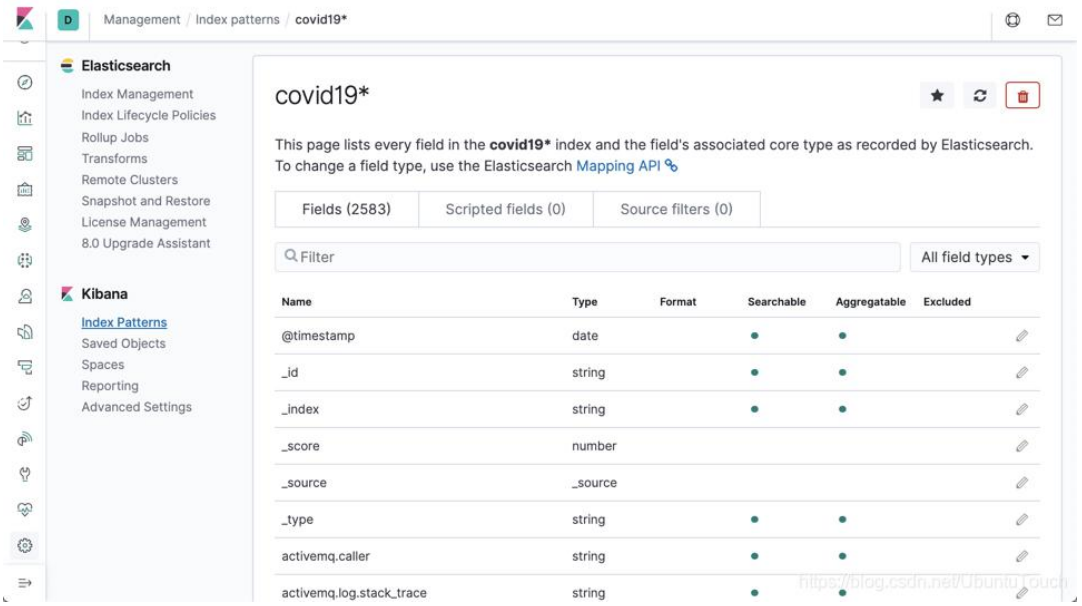
点击上面的 Create index pattern 按钮：


Management / Index patterns / Create index pattern
Elasticsearch
Index Management
Index Lifecycle Policies
Rollup Jobs
Transforms
Remote Clusters
Snapshot and Restore
License Management
8.0 Upgrade Assistant
Kibana
Index Patterns
Saved Objects
Spaces
Reporting
Advanced Settings
Create index pattern
Kibana uses index patterns to retrieve data from Elasticsearch indices for things like visualizations.
Include system indices
Step 1 of 2: Define index pattern
Index pattern
covid19*
You can use a * as a wildcard in your index pattern.
You can't use spaces or the characters \, /, ?, ", <, >, |.
Success! Your index pattern matches 1 index.
Next step
covid19
Rows per page: 10

点击 Next step:


Management / Index patterns / Create index pattern
Elasticsearch
Index Management
Index Lifecycle Policies
Rollup Jobs
Transforms
Remote Clusters
Snapshot and Restore
License Management
8.0 Upgrade Assistant
Kibana
Index Patterns
Saved Objects
Spaces
Reporting
Advanced Settings
Create index pattern
Kibana uses index patterns to retrieve data from Elasticsearch indices for things like visualizations.
Include system indices
Step 2 of 2: Configure settings
You've defined covid19* as your index pattern. Now you can specify some settings before we create it.
Time Filter field name Refresh
I don't want to use the Time Filter
The Time Filter will use this field to filter your data by time.
You can choose not to have a time field, but you will not be able to narrow down your data by a time range.
Show advanced options
Back
Create index pattern

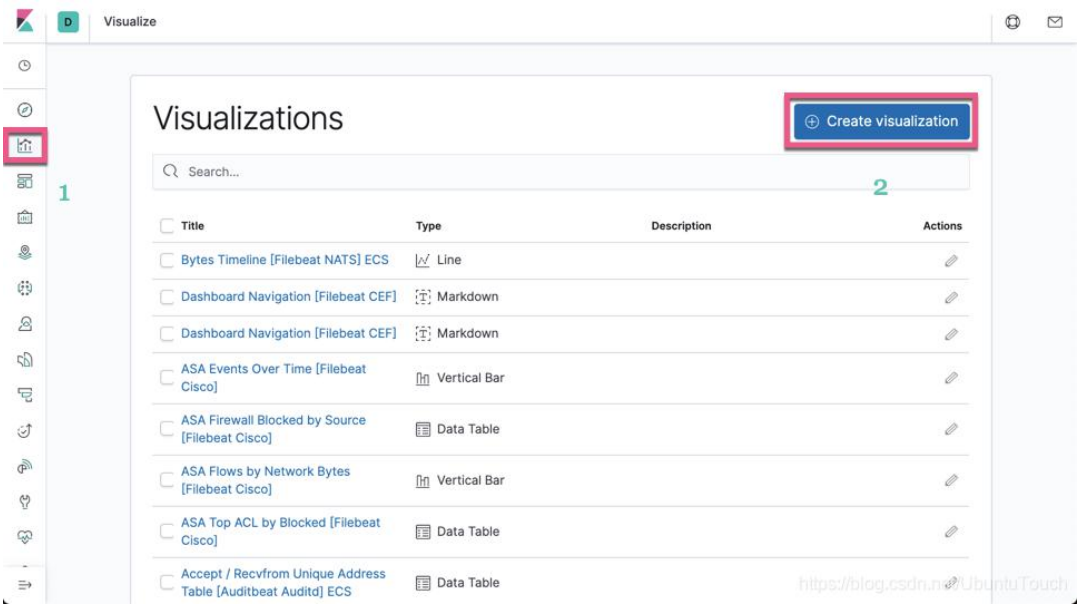
点击 Create index pattern:



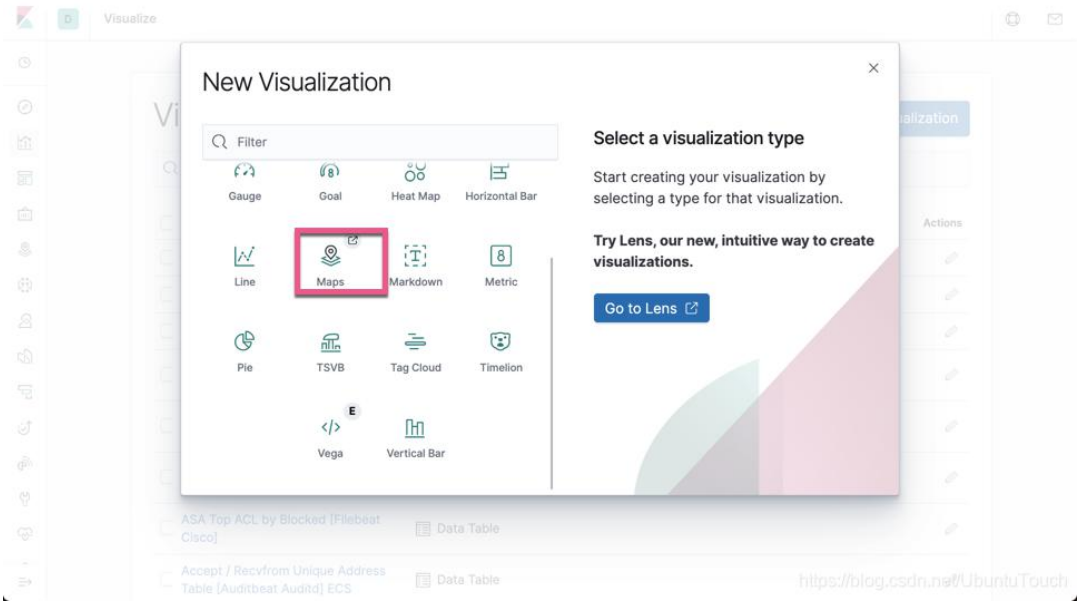
分析数据

创建 Maps visualization

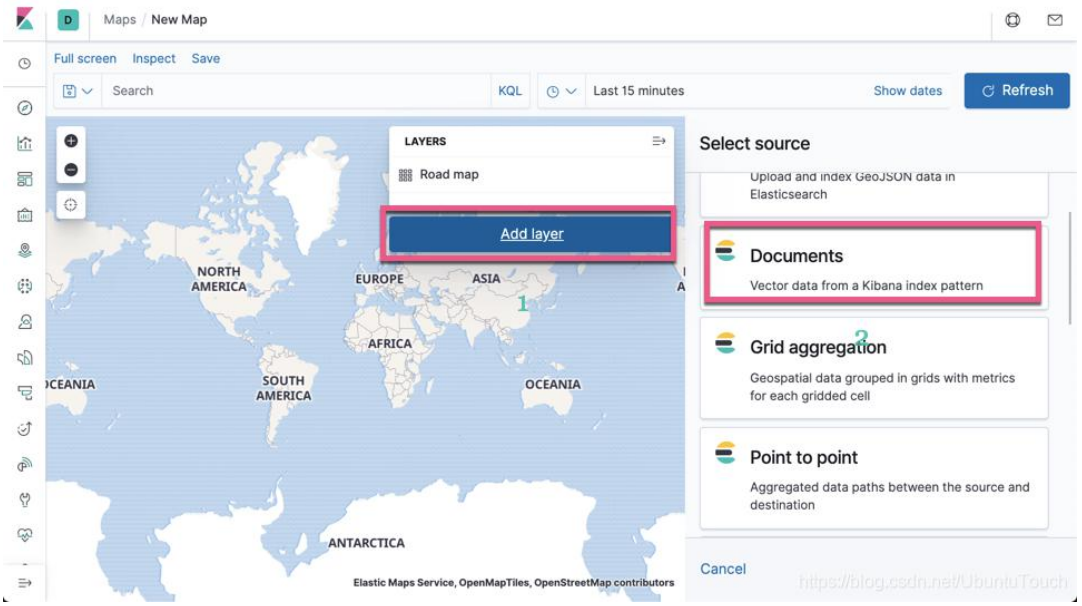
我们打开 Visualization:



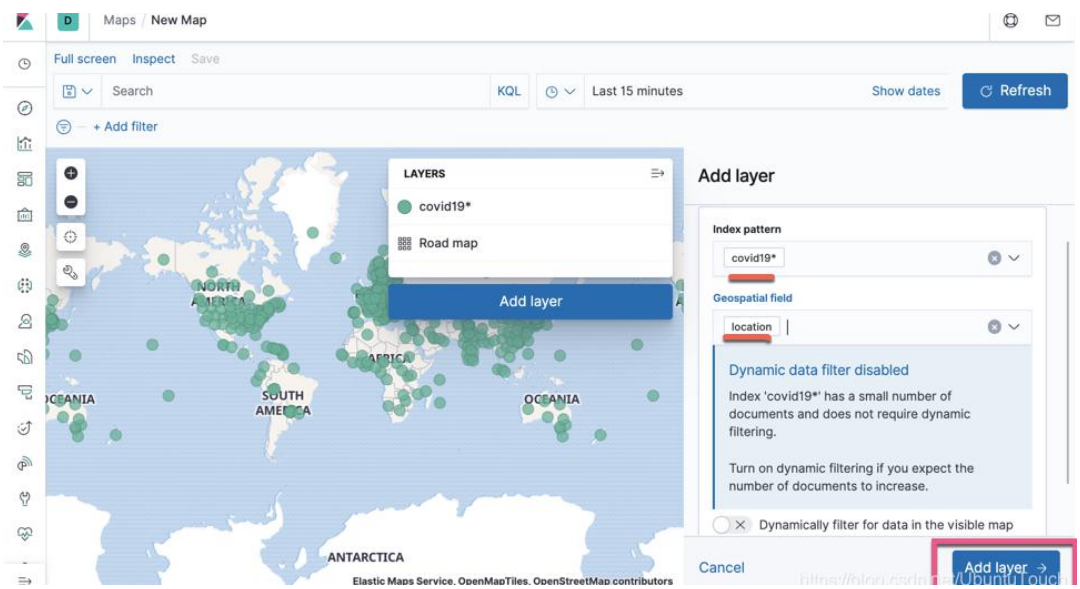
点击 Create visualization:

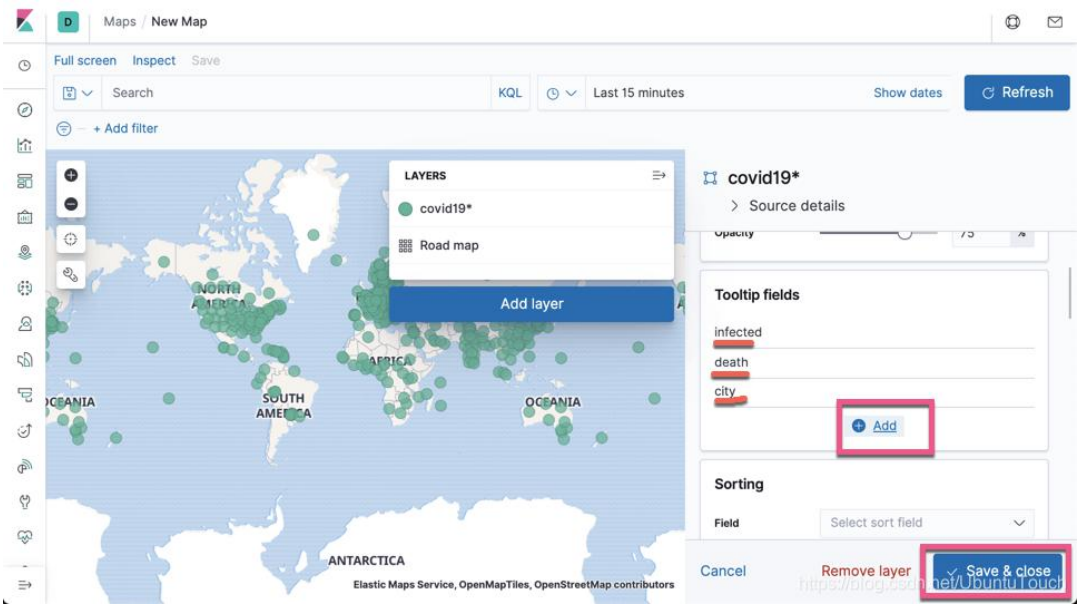


选择 Maps：

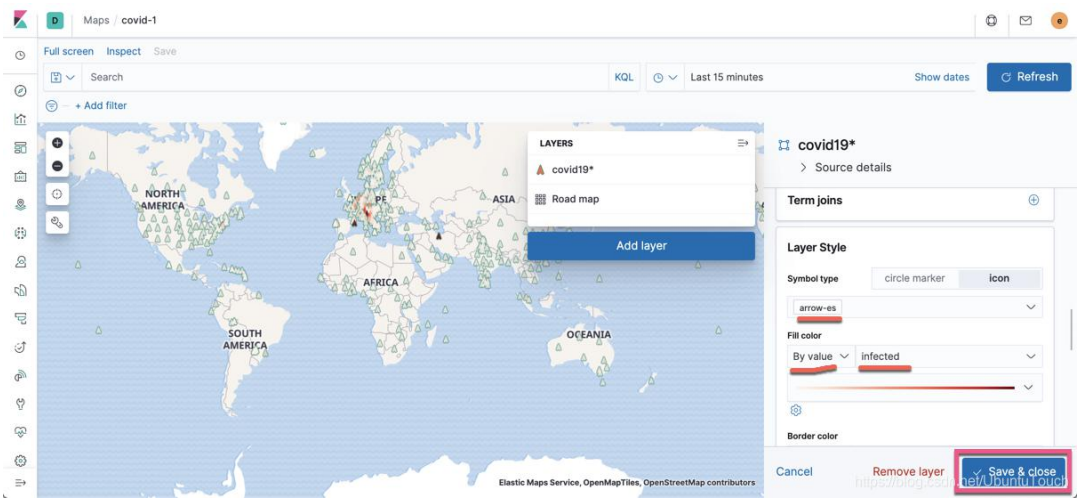


选择 Documents:

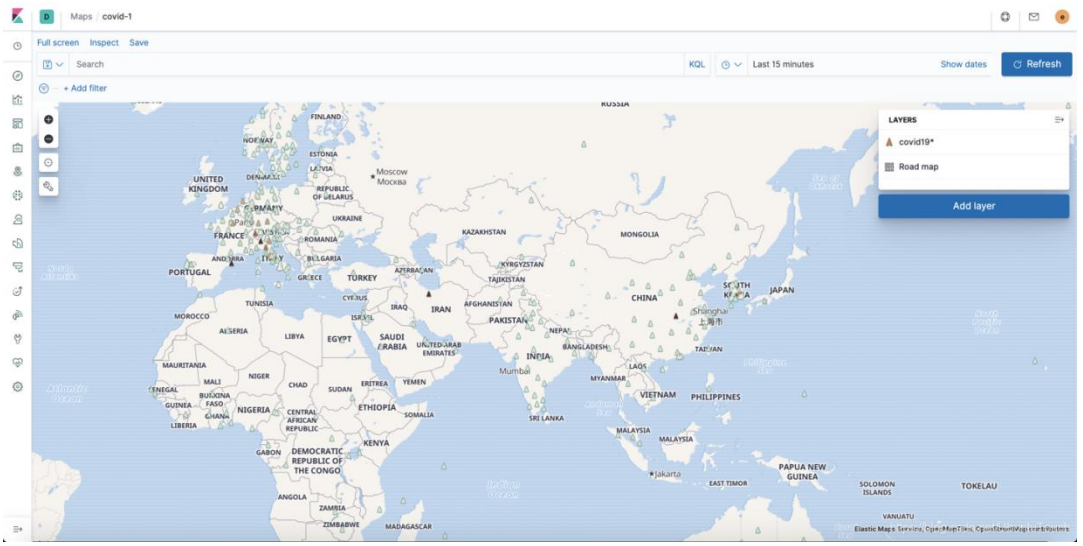




我们接着定制这个 Map：



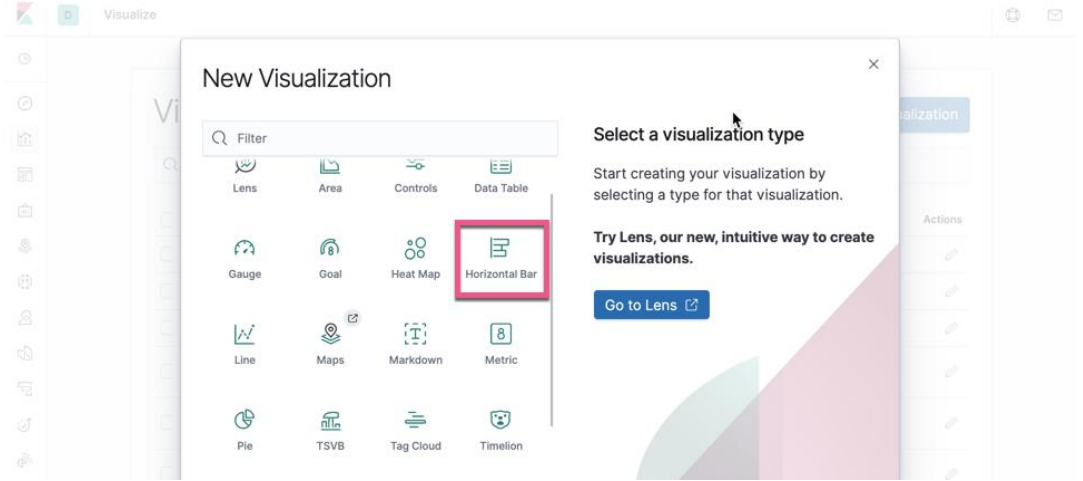
这样颜色越深，代表感染的越严重。点击 Save & close:



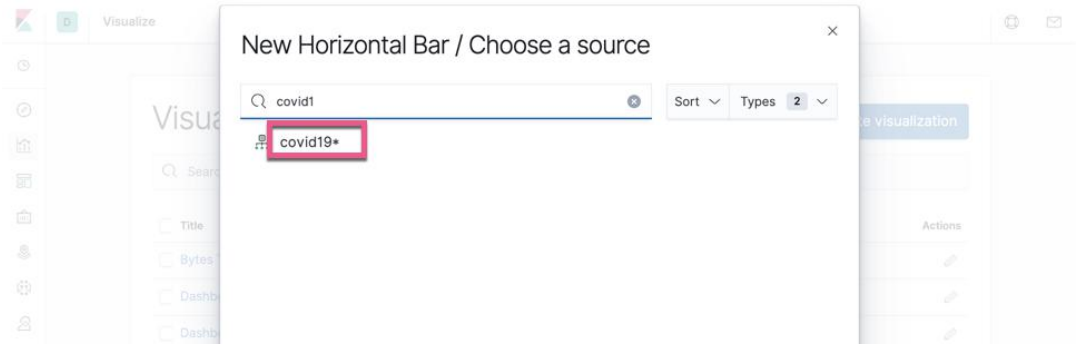
我们把上面的 Visualization 保存为 covid-1。

找出感染和死亡最多的国家

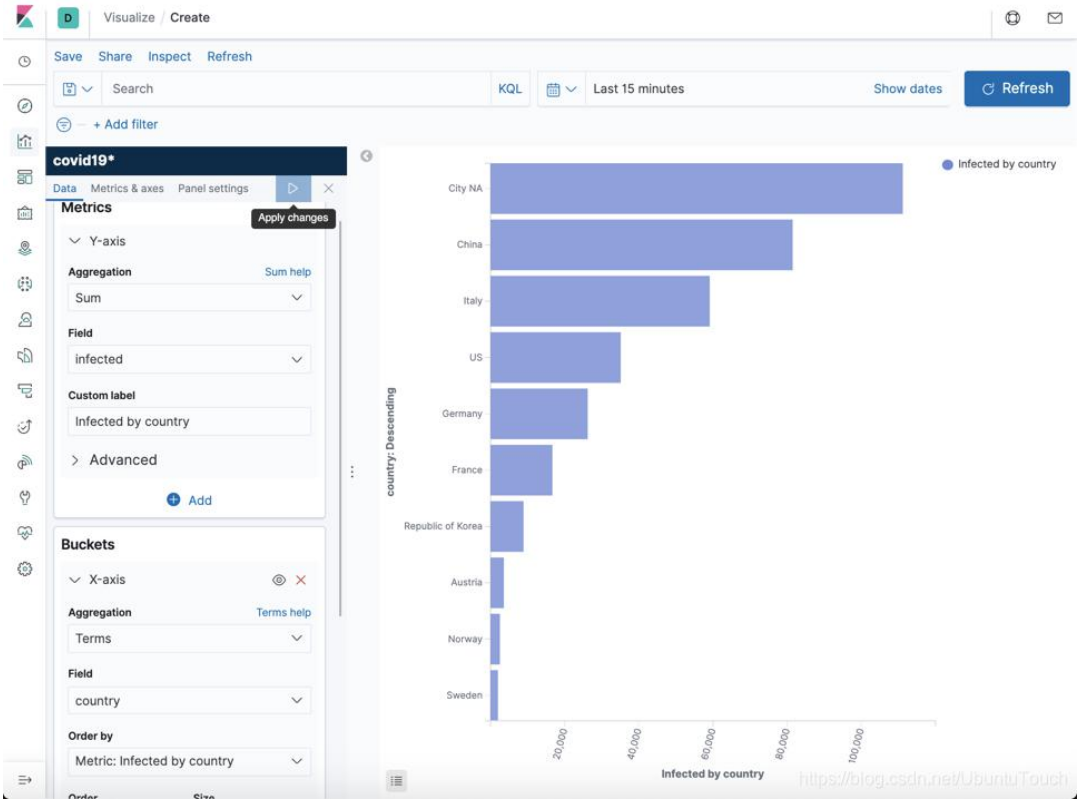
同样地，我们创建一个叫做 Horizontal Bar 的



点击 Horizontal Bar:

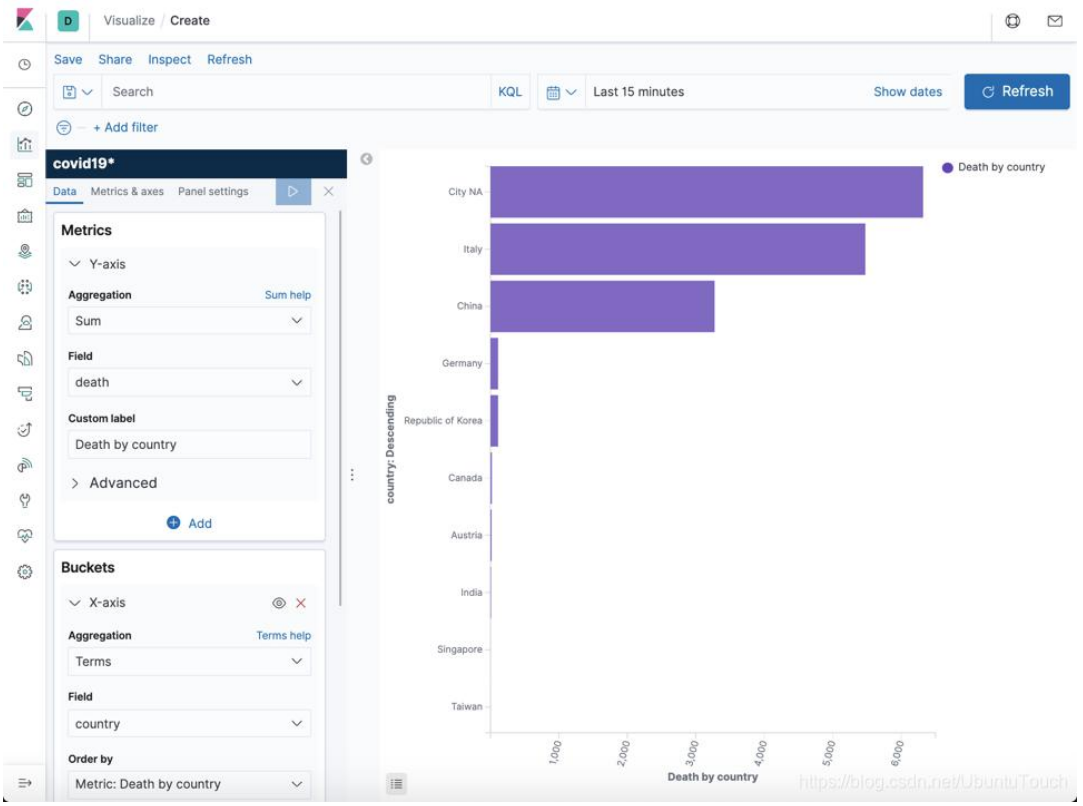


选择 covid-19:



我们保存该 Visualization 为 covid-2。

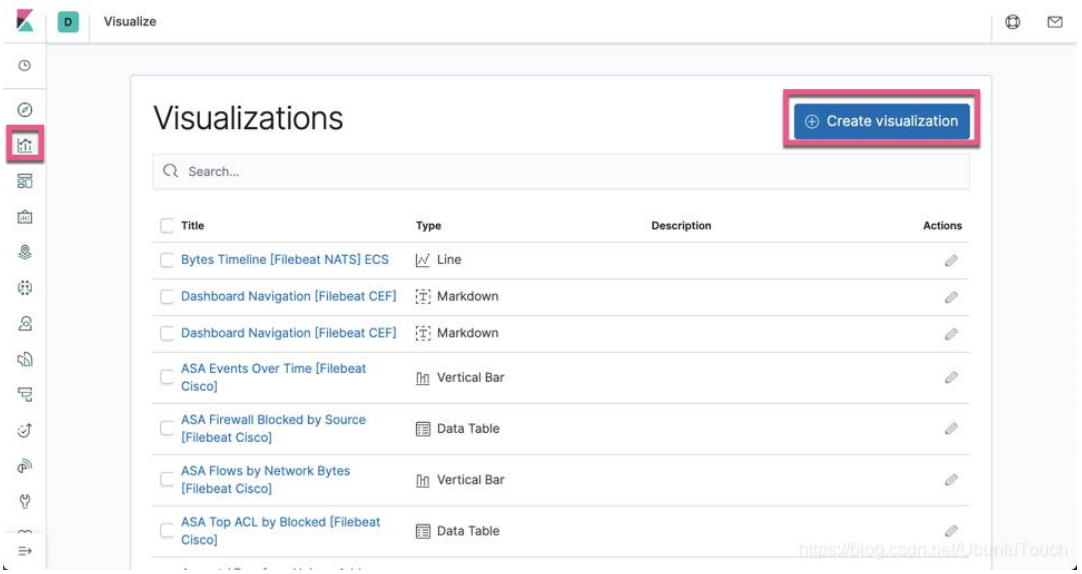
同样地，我们得到死亡最多的前十个国家：

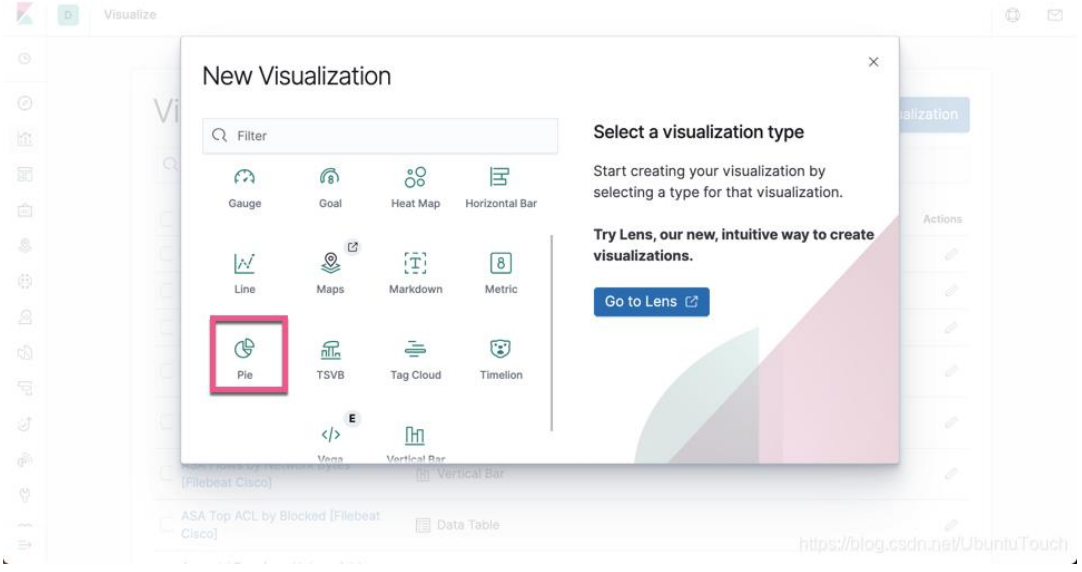


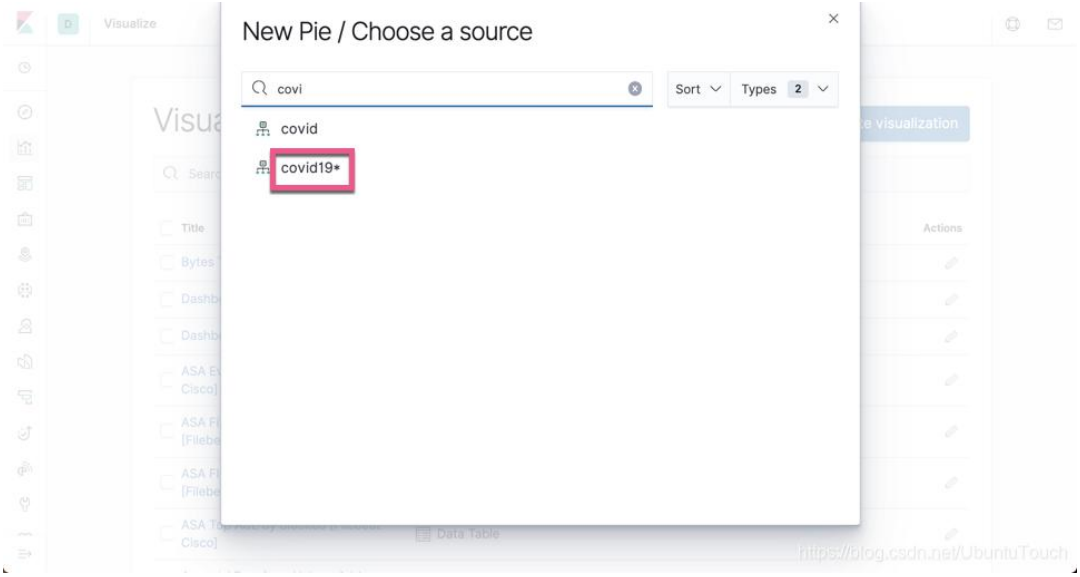
我们保存该 Visualization 为 covid-3。

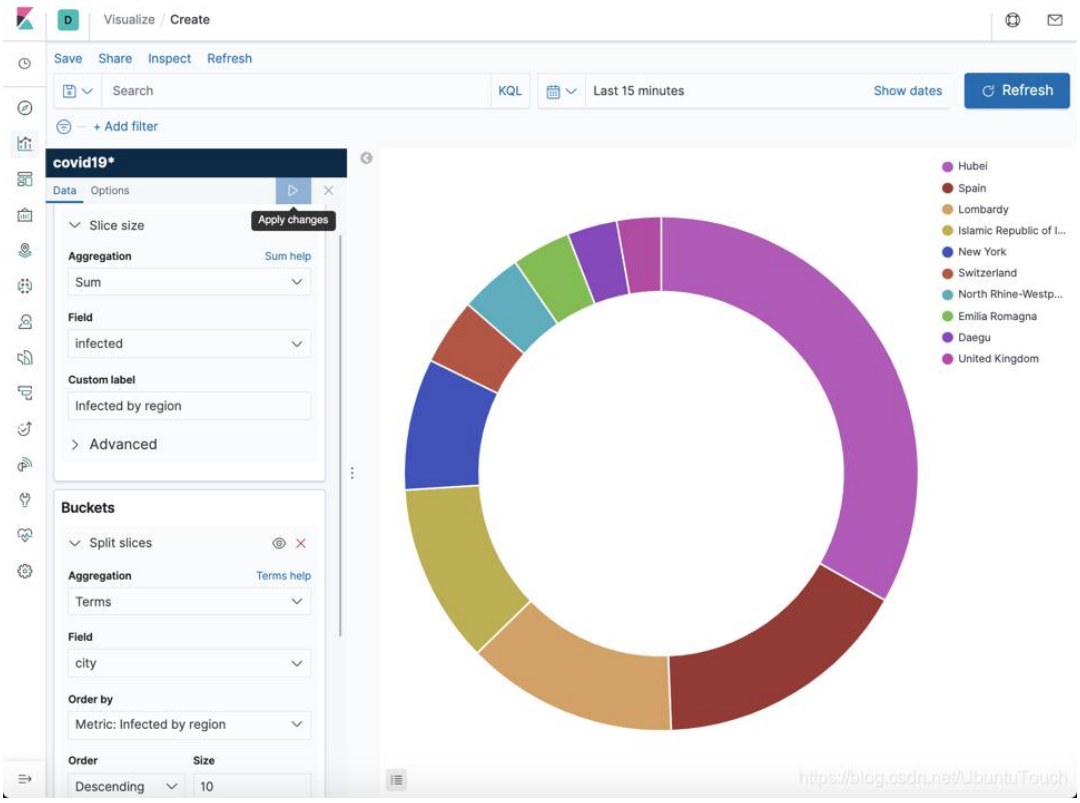
找出感染人数最多的地区

我们可以选择一个 pie 统计：


Visualize
Visualizations
Create visualization
Search...
Title
Type
Description
Actions
Bytes Timeline [Filebeat NATS] ECS
Line
Dashboard Navigation [Filebeat CEF]
Markdown
Dashboard Navigation [Filebeat CEF]
Markdown
ASA Events Over Time [Filebeat Cisco]
Vertical Bar
ASA Firewall Blocked by Source [Filebeat Cisco]
Data Table
ASA Flows by Network Bytes [Filebeat Cisco]
Vertical Bar
ASA Top ACL by Blocked [Filebeat Cisco]
Data Table


New Visualization
Filter
Gauge
Goal
Heat Map
Horizontal Bar
Line
Maps
Markdown
Metric
Pie
TSVB
Tag Cloud
Timelion
Select a visualization type
Start creating your visualization by selecting a type for that visualization.
Try Lens, our new, intuitive way to create visualizations.
Go to Lens

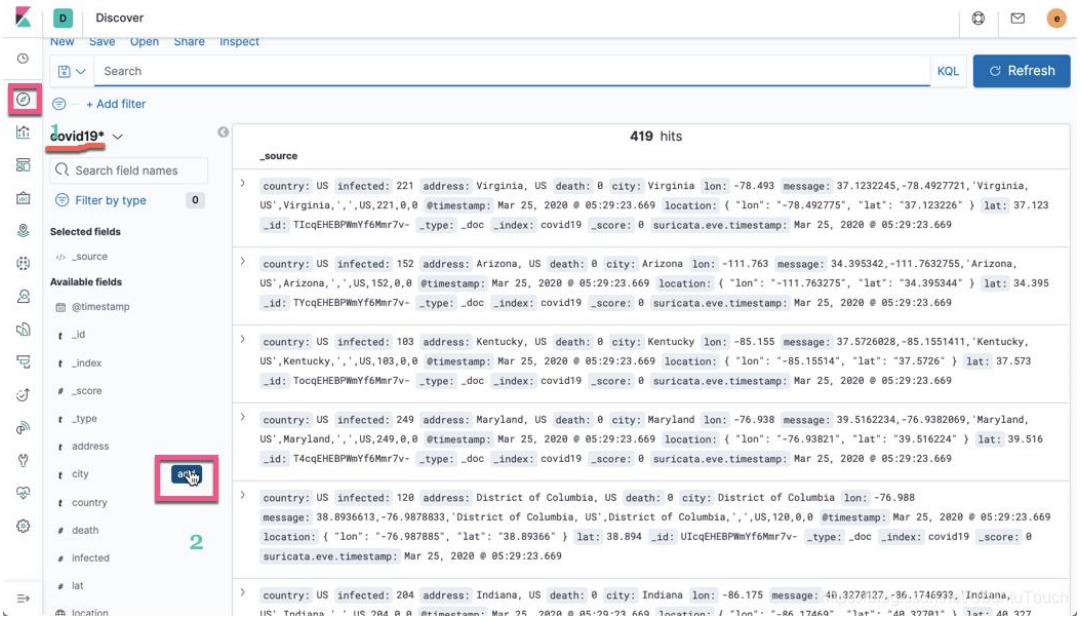

New Pie / Choose a source
covi
Sort
Types 2
covid
covid19*
https://blog.csdn.net/UbuntuTouch


Visualize / Create
Save Share Inspect Refresh
Search
KQL
Last 15 minutes
Show dates
Refresh
+ Add filter
covid19*
Data Options
Apply changes
Slice size
Aggregation
Sum help
Sum
Field
infected
Custom label
Infected by region
Advanced
Buckets
Split slices
Aggregation
Terms help
Terms
Field
city
Order by
Metric: Infected by region
Order
Descending
Size
10
Hubei
Spain
Lombardy
Islamic Republic of I...
New York
Switzerland
North Rhine-Westp...
Emilia Romagna
Daegu
United Kingdom
https://blog.csdn.net/UbuntuTouch

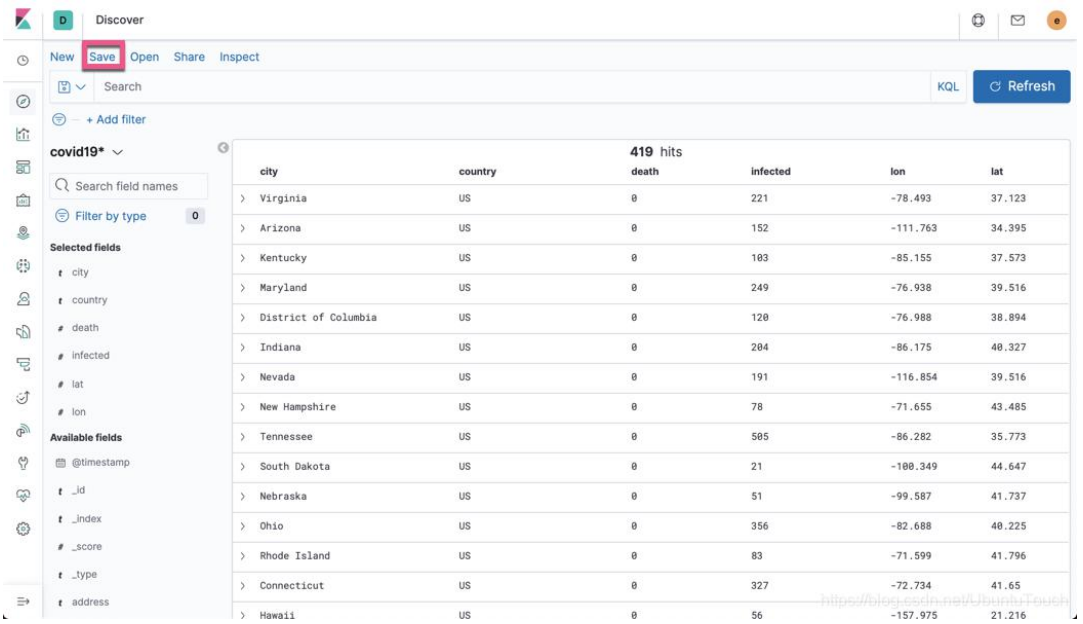
我们保存上面的 Visualization 为 covid-3。

建立一个表格

我们最早得到的表格也是非常不错的。我们想把之前的那个表格的内容也放到我们的 Dashboard 里，这样，我们可以在 Dashboard 里进行搜索。我们首先点击 Discover:



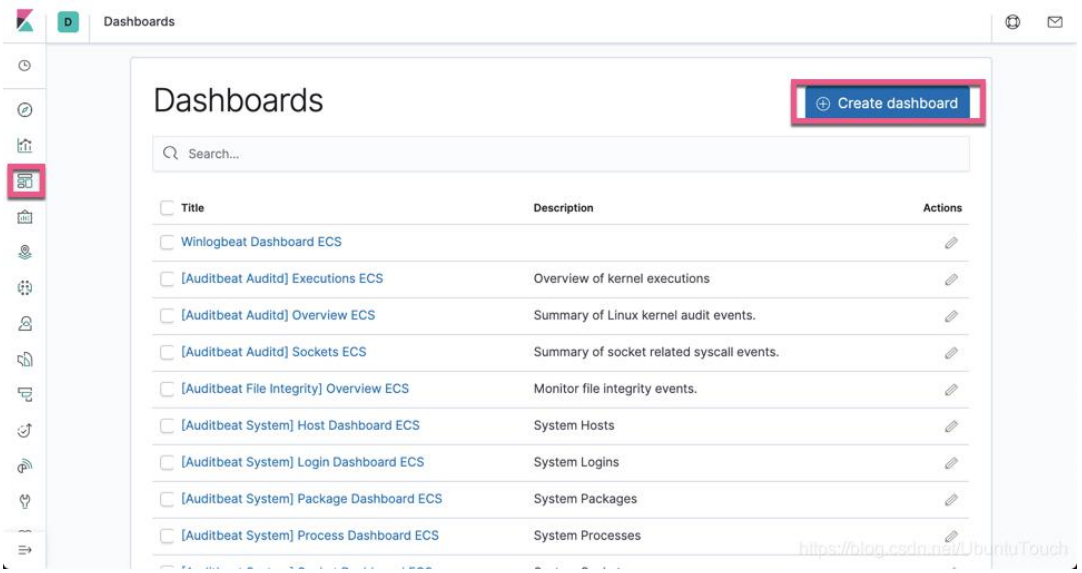
我们选中我们的 Index pattern covid19*。同时在左下方选择我们想要的字段，并点击 add。我们添加 city, country, death, infected 等字段：

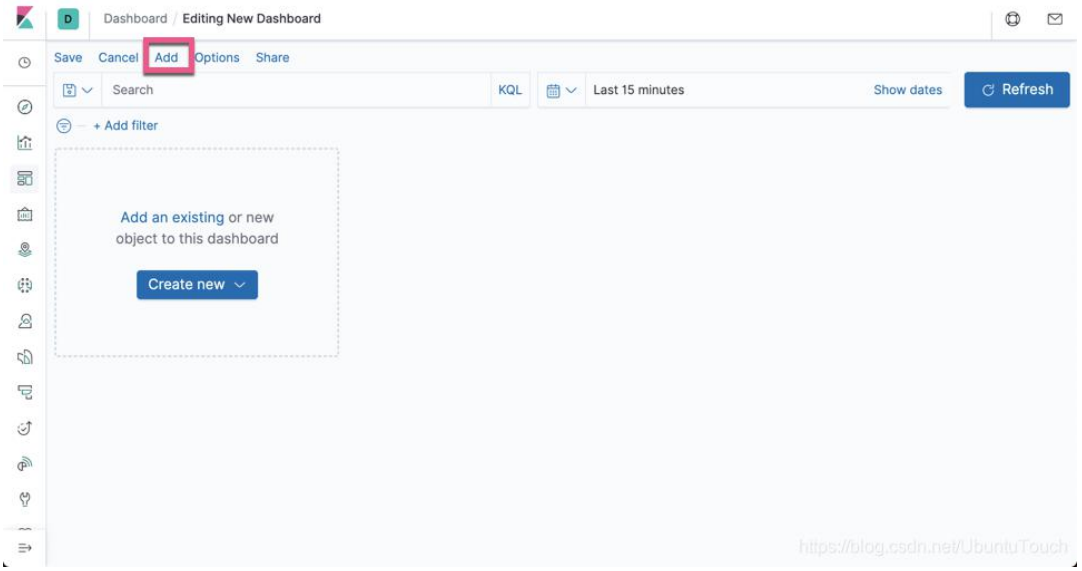


我们保存这个表格为 covid-4。

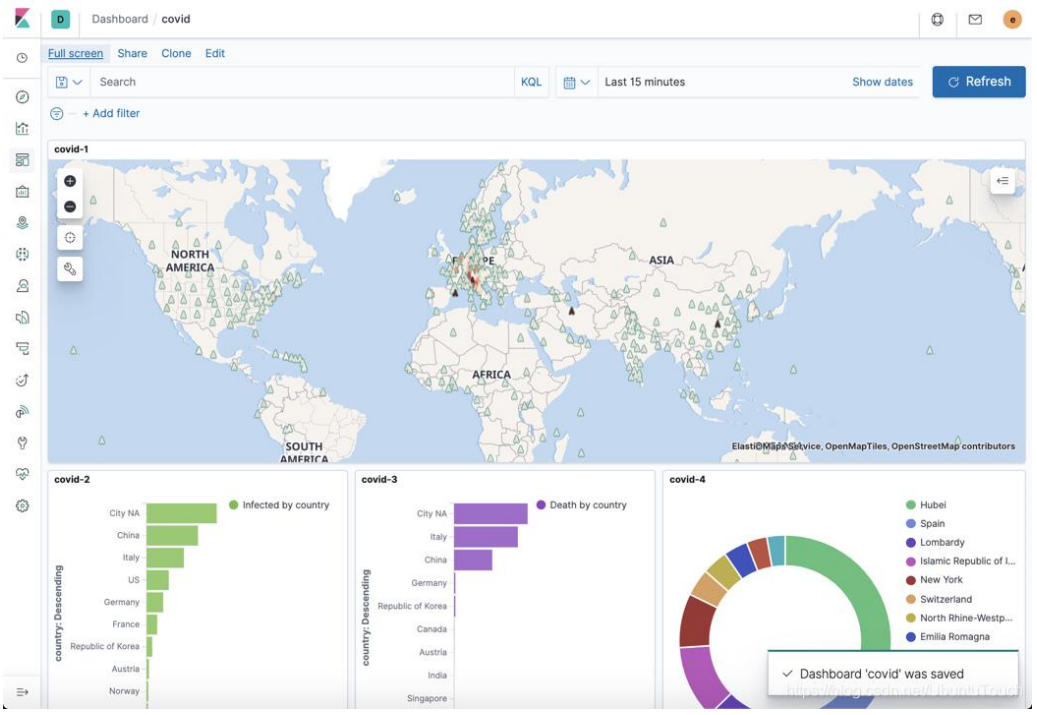
创建 Dashboard

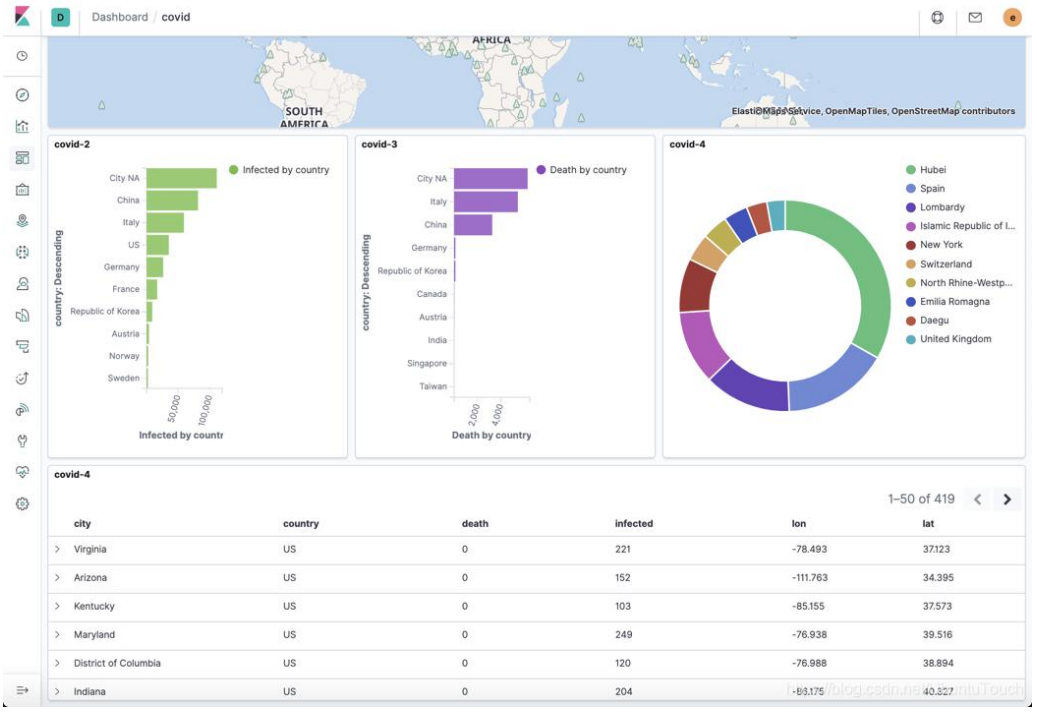
根据之前的 Visualization，我们来创建一个 Dashboard：



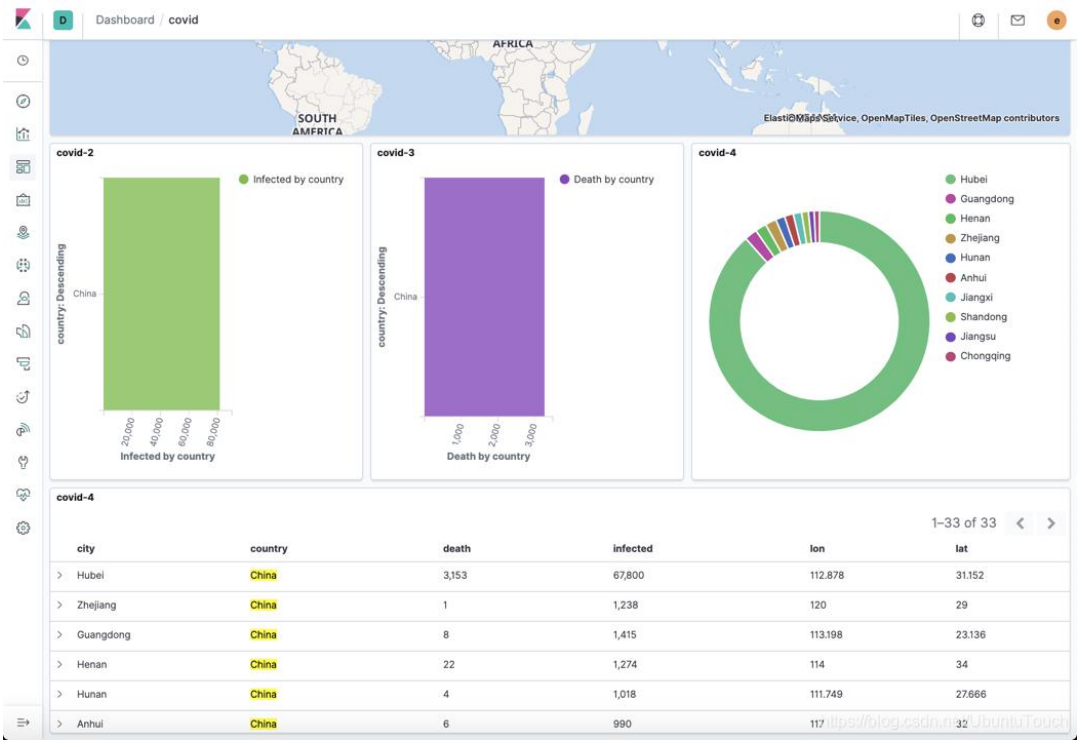


我们把之前创建的 Visualization 一个一个地添加进来，并形成我们最终的 Dashboard：


Dashboard / covid
Full screen
Share
Clone
Edit
Search
KQL
Last 15 minutes
Show dates
Refresh
+ Add filter
covid-1
NORTH AMERICA
ASIA
AFRICA
SOUTH AMERICA
Elastic Maps Service, OpenMapTiles, OpenStreetMap contributors
covid-2
Infected by country
City NA
China
Italy
US
Germany
France
Republic of Korea
Austria
Norway
country: Descending
covid-3
Death by country
City NA
Italy
China
Germany
Republic of Korea
Canada
Austria
India
Singapore
country: Descending
covid-4
Hubei
Spain
Lombardy
Islamic Republic of I...
New York
Switzerland
North Rhine-Westp...
Emilia Romagna
Dashboard 'covid' was saved


Dashboard / covid
AFRICA
SOUTH AMERICA
Elastic Maps Service, OpenMapTiles, OpenStreetMap contributors
covid-2
Infected by country
City NA
China
Italy
US
Germany
France
Republic of Korea
Austria
Norway
Sweden
50,000
100,000
Infected by countr
country: Descending
covid-3
Death by country
City NA
Italy
China
Germany
Republic of Korea
Canada
Austria
India
Singapore
Taiwan
2,000
4,000
Death by country
country: Descending
covid-4
Hubei
Spain
Lombardy
Islamic Republic of I...
New York
Switzerland
North Rhine-Westp...
Emilia Romagna
Daegu
United Kingdom
1–50 of 419
city
country
death
infected
lon
lat
Virginia US 0 221 -78.493 37.123
Arizona US 0 152 -111.763 34.395
Kentucky US 0 103 -85.155 37.573
Maryland US 0 249 -76.938 39.516
District of Columbia US 0 120 -76.988 38.894
Indiana US 0 204 -86.175 40.327

在上面图中，如果我们点击 China，那么所有的图形将变为:



从上面我们可以看出来整个表格都发送了变化，而且我们的饼状图也发生了相应的变化。

参考链接:

- https://www.pharmaceutical-technology.com/special-focus/covid-19/coronavirus-covid-19-outbreak-latest-information-news-and-updates/
- https://www.siscale.com/importing-covid-19-data-into-elasticsearch/

4.2.7 IP 地址分布地图可视化

创作人：铭毅天下

本文将介绍如何基于 Elasticsearch + Kibana 实现 IP 地址分布地图可视化

实践背景

- 有一批特定用途（文末揭晓）的 IP 地址。
- 想通过地图形式可视化展示 IP 地址对应的经纬度坐标的分布。

方案探讨

基础方案如下：

- 第一步：IP 地址转经纬度坐标。

  实现借助第三方工具：https://ipstack.com/

- 第二步：经纬度坐标借助可视化工具（如：echarts）渲染展示。

  这时候不免进一步思考：

有没有更快捷的方案呢？ELK 能实现不？

已知的知识点：

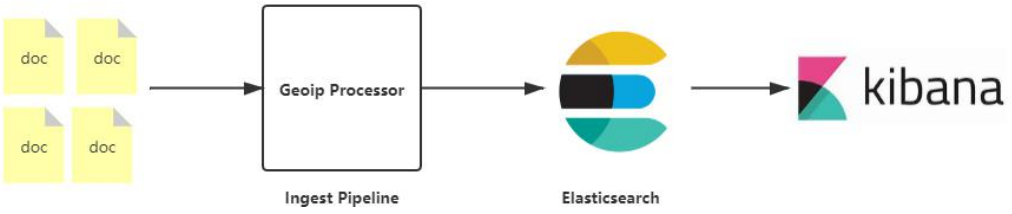
- Elasticsearch 支持 Geo-point、Geo-shape 数据类型。
- Kibana 支持 Coordinate Map（坐标图）、Region Map（区域地图）可视化地图展示。

两个已知知识点一整合不就是基于 Elasticsearch + Kibana 的可视化展示方案吗？

且慢，有没有更快捷的 IP 地址转经纬度坐标的信息呢？

有的。Ingest 数据预处理管道的 GeoIP Processor （处理器）就能达到这个目的。

整体架构图如下图所示：



Geoip processor 介绍

官方解读如下：Geoip processor 根据来自 Maxmind 数据库的数据添加有关 IP 地址地理位置的信息。

默认情况下，GeoIp processor 将此信息添加到 geoip 字段下。GeoIp processor 可以解析 IPv4 和 IPv6 地址。

更多 Maxmind 数据库信息参见：https://dev.maxmind.com/geoip/geoip2/geolite2/

在 Elasticsearch 早期版本中 GeoIp processor 需要安装插件才能使用。7.X 版本后，Elasticsearch 已自带，不需要安装。

导入一条数据实战一把

步骤 1：创建预处理管道

```
PUT _ingest/pipeline/geoip_pipeline
{
  "description" : "Add geoip info",
  "processors" : [
    {
      "geoip" : {
        "field" : "ip"
      }
    }
  ]
}
```

该预处理的目的就是：将输入的 IP 字段转换为：Geoip 类型。具体 Geoip 类型张什么样？后面会揭晓。

步骤 2：创建索引

```
DELETE niu_20210215
PUT niu_20210215
{
  "settings": {
    "index.default_pipeline": "geoip_pipeline",
    "number_of_shards": 1,
    "number_of_replicas": 0
  },
  "mappings": {
    "properties": {
      "geoip": {
        "properties": {
          "location": {
            "type": "geo_point"
          }
        }
      },
      "ip":{
        "type":"keyword"
      }
    }
  }
}
```

考虑到后面要批量导入数千条 + 数据，我们采用了取巧的方式。

使用了在创建索引的时候指定缺省管道（index.default_pipeline）的方式。

这样的好处是：

- 灵活：用户只关心 bulk 批量写入数据。
- 零写入代码修改：甚至写入数据的代码一行都不需要改就可以。

步骤 3：写入一条数据

```
PUT niu_20210215/_doc/1
{
  "ip": "8.8.8.8"
}
```

这时候，我们查看一下完整的 Mapping 长什么样？

```
{
  "niu_20210215" : {
    "mappings" : {
      "properties" : {
        "geoip" : {
          "properties" : {
            "city_name" : {
              "type" : "text",
              "fields" : {
                "keyword" : {
```

```
            "type" : "keyword",
            "ignore_above" : 256
          }
        }
      },
      "continent_name" : {
        "type" : "text",
        "fields" : {
          "keyword" : {
            "type" : "keyword",
            "ignore_above" : 256
          }
        }
      },
      "country_iso_code" : {
        "type" : "text",
        "fields" : {
          "keyword" : {
            "type" : "keyword",
            "ignore_above" : 256
          }
        }
      },
      "location" : {
        "type" : "geo_point"
      },
      "region_iso_code" : {
        "type" : "text",
        "fields" : {
          "keyword" : {
            "type" : "keyword",
```

```
                    "ignore_above" : 256
                  }
                }
              },
              "region_name" : {
                "type" : "text",
                "fields" : {
                  "keyword" : {
                    "type" : "keyword",
                    "ignore_above" : 256
                  }
                }
              }
            },
            "ip" : {
              "type" : "keyword"
            }
          }
        }
      }
    }
```

写入后的数据，查看返回如下：

```
  "_source" : {
    "geoip" : {
      "continent_name" : "North America",
      "country_iso_code" : "US",
      "location" : {
```

```
        "lon" : -97.822,
        "lat" : 37.751
      }
    },
    "ip" : "8.8.8.8"
}
```

有点长，铭毅解读一下：

第一：geoip 是 object 类型，它有几个子字段，含义如下：

- geoip.city_name：城市
- geoip.continent_name：大陆名称
- geoip.country_iso_code：国家编码
- geoip.location：经纬度坐标，必须是：geo_point 类型
- geoip.region_iso_code：地域编码
- geoip.region_name：地域名称

第二：为节省存储，Mapping 可以优化。

- 比如：所有的默认字符串类型改成：keyword 类型。

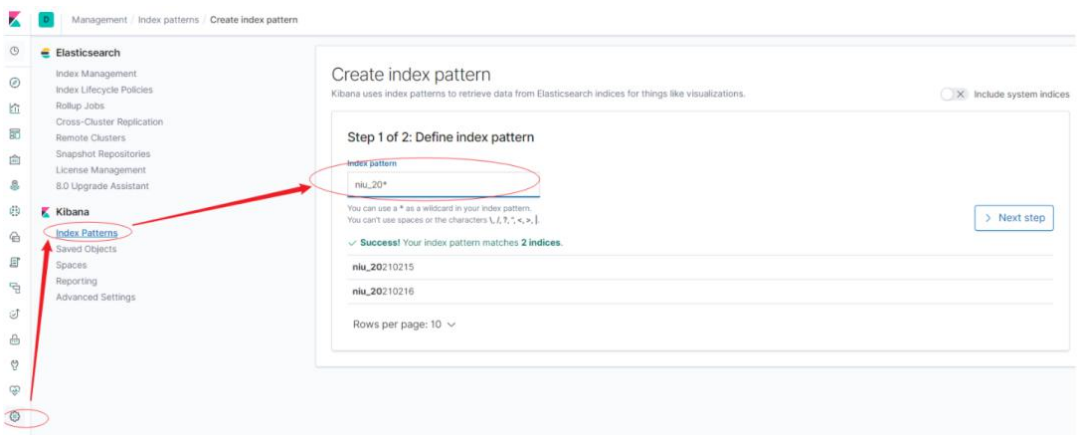
第三：为了后面的作图必须将 location 设置为 geo_point 类型。

以上三个步骤：就完成了单条数据的写入。

步骤 4：Kibana 可视化展示

创建关联索引模板

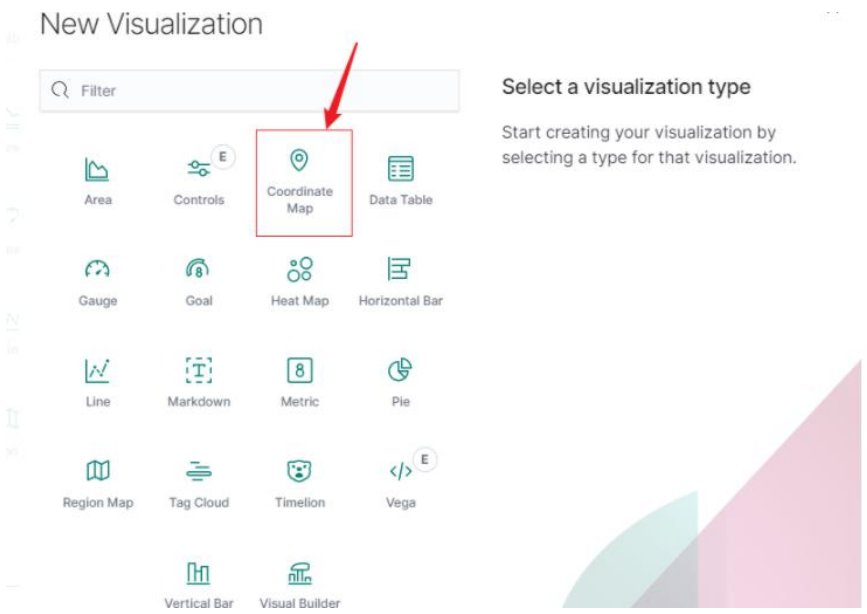
目的：创建可视化需要关联的索引数据。



创建坐标图

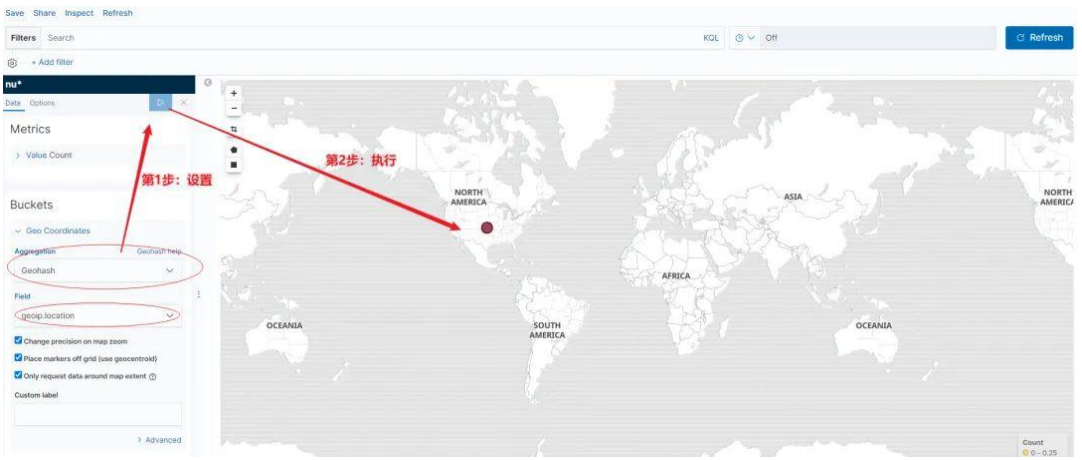
选择左侧导航栏的 Visualize，然后选择右侧 Create new visualization，然后再选择：Coordinate Map 即可。

本文 Elasticsearch + kibana 均选用 7.2 版本。



可视化基础设置，执行后，就能看到可视化结果。

如前所述，这里要强调的是：geoip.location 必须是 geo_point 类型。

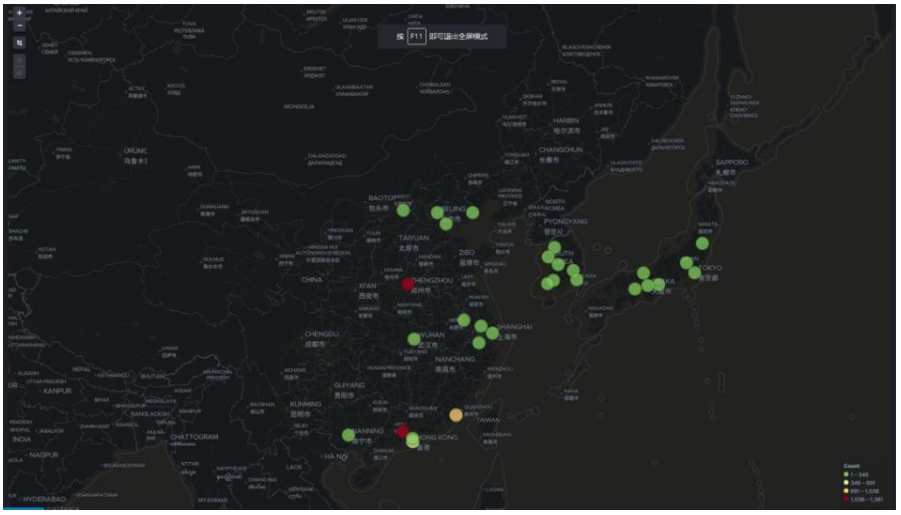


批量导入数据后可视化展示

基于上一段导入一条数据，python 批量 bulk 导入本地文件数据后，可视化效果如下图所示：

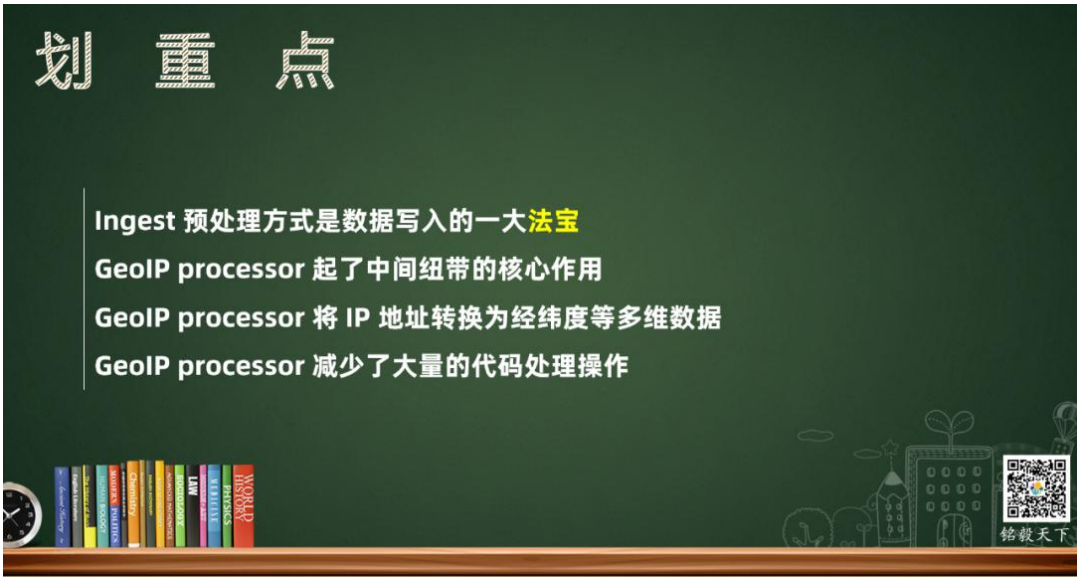
因为全局设置了 default_pipeline，写入数据不需要做任何特殊处理了。

换 dark 风格显示如下：



PS：文章开头提到的：特定用途——通过模拟 port scan 获取的全网部分开放 9200 端口的公网 IP（仅个人学习用，未任何其他用途）。

小结



基础方案大家都能想到，有没有更简单的、更快捷的方式呢？是需要我们考虑的。

本文抛砖引玉，Kibana 新版本的可视化功能更强大，需要学习的点还有很多.....

参考链接：

- https://blog.ruanbekker.com/blog/2018/09/12/using-the-geoip-processor-plugin-with-elasticsearch-to-enrich-your-location-based-data/

4.3 安全能力应用场景

4.3.1 基于 Elastic Stack 构建 SOC 能力

创作人：冯钰妍

审稿人：李捷

本文将介绍如何使用 ELK 在网络安全分析领域中的实际应用。

Elasticsearch 做为一款功能强大的分布式搜索引擎，支持近实时的存储、搜索数据，具有扩展性好，检索速度快等优势，依托于 Elasticsearch 的这些优势，其不仅广泛地应用于各种搜索场景，如日志检索，聚合分析等，在安全分析等领域也开始逐渐展现其强大的能力。

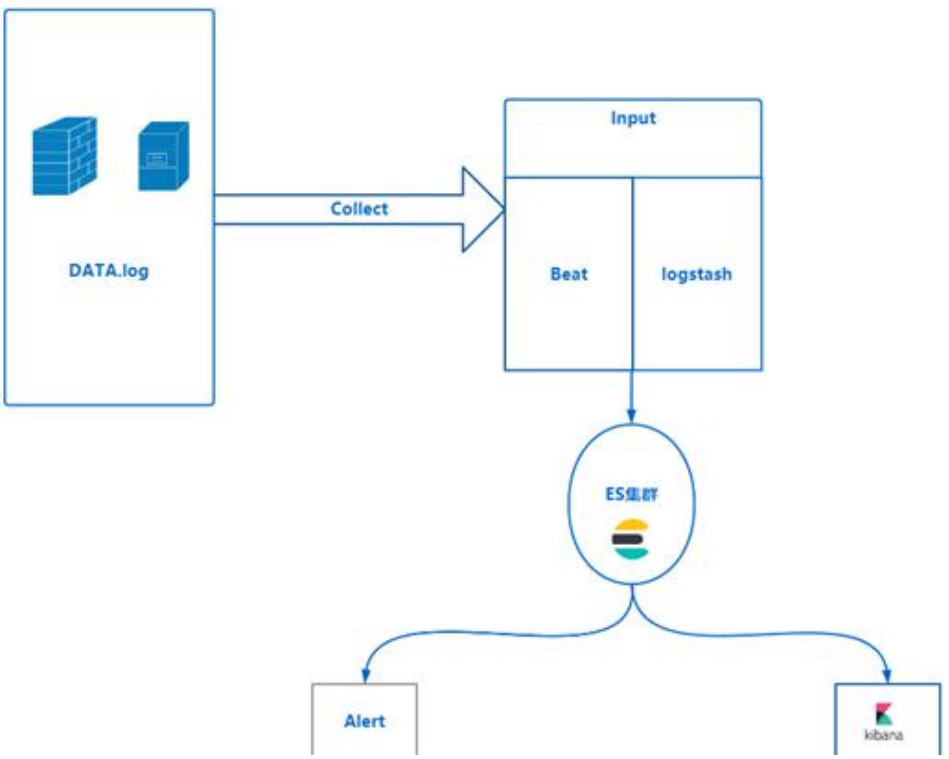
在传统安全领域，企业通常会借助防火墙，杀毒软件等，为企业构造起一套固若金汤的安全防御体系，然而即使在如此严密的防护之下，仍然无法完全保证内部数据的安全，尤其是当面临内部威胁时。

这时，根据已有安全数据进行安全分析，及时发现并处理威胁，就需要使用到可以大量实现实时日志存储及数据可视化于一体的平台——Elastic Stack。

然而，现代企业的安全数据，已随着日益蓬勃发展的信息网络技术而迅速膨胀，对海量安全数据的采集，处理，存储，查询等正日益困扰着企业安全分析团队。而

Elasticsearch 正是为应对海量数据的采集和检索而生的，将 Elasticsearch 应用于安全分析领域，可以非常便捷高效地解决安全分析领域，海量数据的存储和检索问题。

- 第一部分是先配置好不同安全产品或服务器的监控日志的 syslog 转发；
- 第二部分则是将监控数据从监控系统中收集到 Logstash 或 Beat，它还负责对监控数据进行拆分、转换的职责；
- 第三部分是用于存储监控数据的到 Elasticsearch 集群，我们借助 Elasticsearch 冷热模型，延长了监控数据的保存时间；
- 第四部分是将处理过的数据进行可视化展示，通过分析和关联多种安全产品数据，将庞大的安全事件降噪为真实的威胁事件，并且实现定时或实时告警，方便企业的安全运维人员可以在第一时间发现威胁并采取相应处理措施。



数据收集

安全分析的第一步当然是建立各类数据收集，而数据来源的多种多样，决定了我们收集时使用的方式。

轻量级的数据采集工具 Beat

Elastic Stack 中的 Beat 工具包含了丰富的数据采集工具，可以十分轻量且便捷的应用于各种数据的采集场景，下表是目前安全分析中需要采集的各类数据对应该采用的 Beats 方式：

数据类型	来源	采集方式
网络数据	网络监控，抓包等	Packetbeat，Filebeat
应用数据	日志	Filebeat
云端数据	接口，日志	Functionbeat，Filebeat
系统数据	系统调用，日志	Metricbeat,Auditbeat，Winlogbeat，Filebeat Osquery module
设备运行状态数据	日志	Heartbeat

Elastic Stack 的免费及开放性质决定了它没有太大的局限性，所以 Elastic 不仅仅提供了官方 Beats，还有大量的社区贡献的 Beats，用户还可以根据自己的需要开发自定义 Beats，完全可以满足安全分析对各种数据源的采集需求。

ELK 中不可或缺的 L（Logstash）

在网络安全中数据的多种多样，往往会分散或集中地存在于不同的系统中，或常见或不常见的。Logstash 作为一个日志聚合器，可以收集和处理来自几乎任何数据源的数据，它可以过滤、处理、关联，并且增强它收集的任何日志数据。

以下是一段 360 天擎的日志。

<6>{"version":"\u5929\u64ce6.6.0.4000","log_name":"\u6f0f\u6d1e\u7ba1\u7406","log_id":"82a40dbab698e5b6049d22071db2c517","create_time":"2020-02-28 10:40:11","ip":"42.101.64.215","report_ip":"192.168.1.107","mac":"3cf0117b5d27","gid":16777228,"work_group":"abfchina.local","content":{"name":null,"type":null,"action":"\u7528\u6237\u5378\u8f7d\u6f0f\u6d1e\u8865\u4e01","kbid":"4504282"}}

```
{
    "version": "天擎 6.6.0.4000",
    "log_name": "漏洞管理",
    "log_id": "82a40dbab698e5b6049d22071db2c517",
    "create_time": "2020-02-28 10:40:11",
    "ip": "42.101.64.215",
    "report_ip": "192.168.1.107",
    "mac": "3cf0117b5d27",
    "gid": 16777228,
    "work_group": "bayu.local",
    "content": {
```

```
            "name": null,
            "type": null,
            "action": "用户卸载漏洞补丁",
            "kbid": "4504282"
        }
    }
```

针对这种未解码的 JSON 嵌套的日志，首先我们可在数据进去时，设置好解码格式，再使用到 JSON 模块进行对字段的拆分，再选择存放到 Elasticsearch 中，解析如下：

```
    nput {
      syslog {
        timezone => "UTC"
        port => "514"
            tags => "syslog"
        codec => plain{
                    charset=>"GBK"
            }
      }
    }
    filter {
            grok {
                    match => ["message", "<\d?>%{GREEDYDATA:log_json}"]
      }

              json {
                    source => "log_json"
      }
```

```
            mutate {
                add_field => {"[source][ip]" => "%{report_ip}"}
                add_field => {"[destination][ip]" => "%{report_ip}"}
    }
    }
    output{
            elasticsearch {
                    index => "360tq-%{+yyyy.MM}"
                    hosts => ["localhost:9200"]
                    user => "xxxx"
                    password => "xxxx"
            }
    }
```

数据标准化

不同来源的数据中表示相同含义的字段，在名称、类型上各不相同，这就导致了在进行数据检索分析时，为了检索不同数据源中的同类数据，可能要兼容性地写多个查询条件，这给数据分析带来了不小的麻烦。

为了解决这个问题，Elastic 建立 ECS(Elastic Common Schema) 项目，采用专业的分类方法，对字段进行统一命名，且 Elastic 生态相关组件，均遵循这一命名规格，使得对不同数据源的检索得以简化。

同样是以上 360 天擎的日志，其日志字段被 Logstash 的 JSON 解析器拆分之后的字段与 ECS 后的字段对比如下：

原生字段	ECS 后
create_time	event.created
log_id	event.id
log_name	event.category
ip	source.nat.ip
client_name	host.name
login_user	user.name

经过对字段的处理，字段的统一命名，便大大的提高了各类日志源之间的关联性。

ECS 不仅对字段的名称进行了规范，对字段的类型也进行了定义。在安全分析中，对 IP 的关联查询频次较高，以下是 ECS 对 source 的部分字段的定义：

字段	定义	类型
source.ip	IP address of the source (IPv4 or IPv6).	ip
source.port	Port of the source.	long
source.domain	Source domain.	keyword
source.mac	MAC address of the source.	keyword

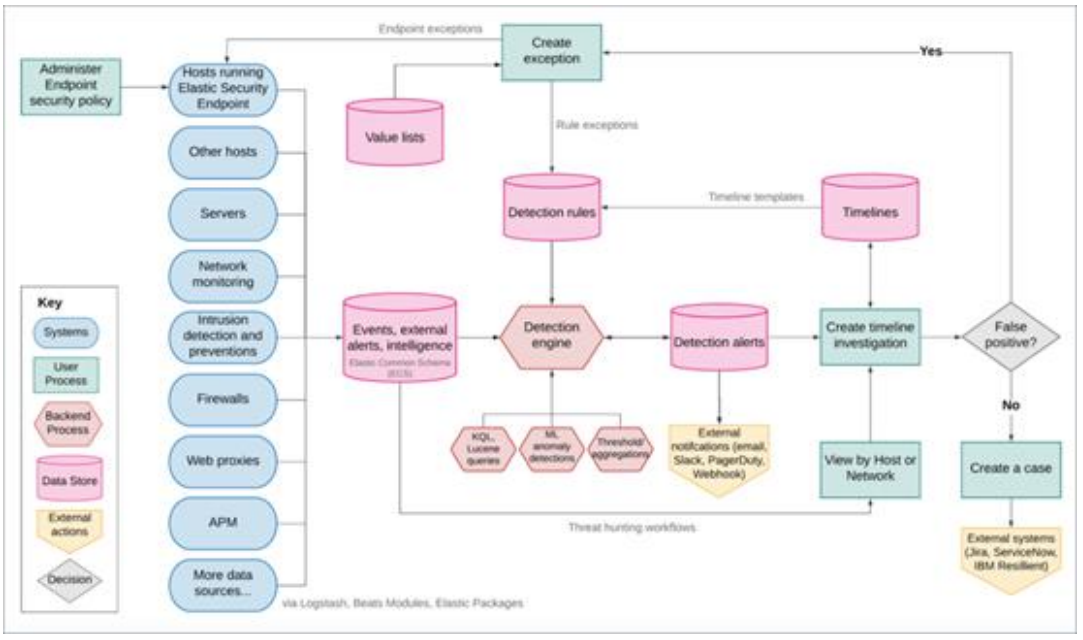
威胁事件与告警

实现异常自动发现，最直接的方式便是基于阈值规则的监控和告警。

Kibana Detection 模块：

Elastic 官方在 Kibana v7.3 之后推出了 SIEM APP 后，更名为 Security app ，为安全分析团队提供了一个专用的集成化交互工作空间，数据可通过 Beats 模块和 Endpoint agent 发送到 Elasticsearch。使用 Detection 功能创建和管理规则，并查看这些规则触发的告警。

除了可以使用 Elastic 预先构建的规则之外，还可以自定义规则，创建规则警报时，可以使用 Kibana 的 Alerts 和 Actions 发送通知，通知可以通过 Email、PagerDuty、Slack 和 Webhook 发送。



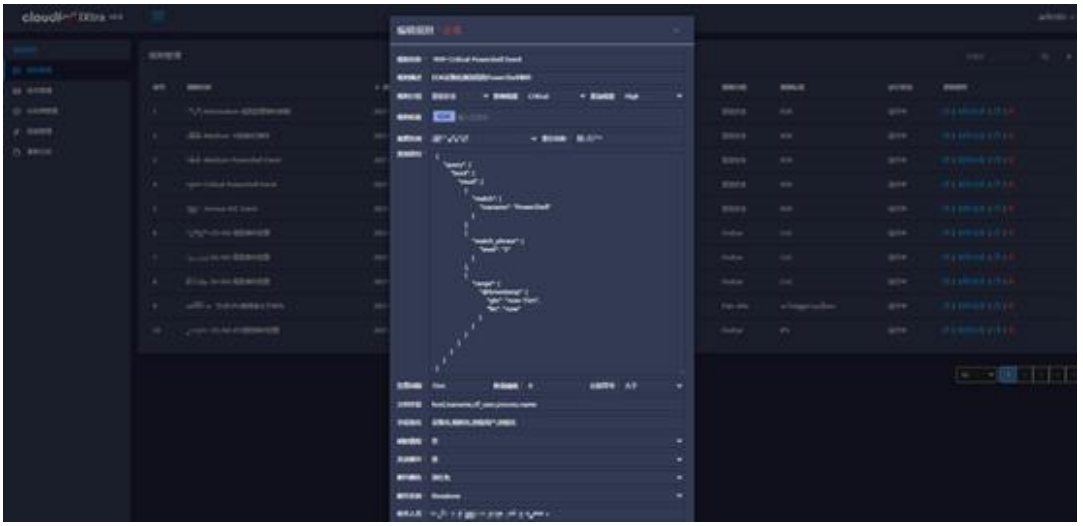
IXtra 邮件告警：

最早期为了实现定期查询 Elasticsearch 数据、判断是否满足特定条件、发送 HTML 邮件，衍生了 IXtra Web 应用。

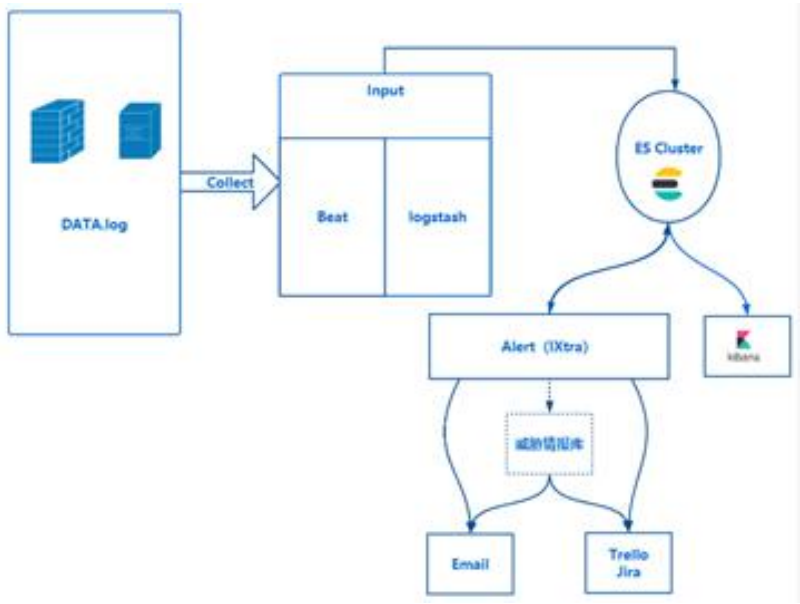
IXtra 应用是由一组邮件告警 Python 脚本发展而来，经过不断地升级迭代，现阶段 IXtra 的主要功能是事件监控，通过连接多个不同的 Elasticsearch 集群，可以集中地对这些集群进行监控和告警。

当集群内某个索引满足告警阈值内条件时，即产生告警。但一旦这些告警产生大量噪音时，IXtra 的聚合阈值型查询（聚合后重复出现次数大于阈值），则可以发挥到降噪作用。

除此之外 IXtra 也可以在一个 Elasticsearch 集群内建立两个查询（同时满足），进行组合型查询。并且所有查询后的结果，都会自动根据预设的规则分组、影响程度、紧急程度和标签等通知到安全运维人员，另外 Ixtra 还可将告警的内容写入至指定索引、自动将告警内容，提交到威胁情报库进行调查，并且可以自动将告警事件通过 API 方式在工单（Trello、Jira 等）系统创建 ticket 等功能。



具体的工作流程图如下：



大大简化了安全运维人员的手动操作部分、提升安全运维人员的工作效率。

扮演的角色

经过以上的阐述，希望你大致可以了解到，其实 Elastic Stack 本身并不是一个传统意义上的 SIEM（Security information and event management）平台，而是我们这些具有专业素养的安全服务行业的人员，利用 Elasticsearch 本身的性质，DIY 搭建一个具有 SIEM 基础功能的平台。完整的 SIEM 解决方案，包含从各种数据源收集信息，长时间保留信息，在不同事件之间关联，创建关联规则或警报，分析数据并使用可视化和仪表板监控数据的能力。

随着安全数据的不断暴增，传统的安全分析方法，对于企业安全问题的定位和解决越来越显得力不从心。Elastic 作为当前主流的大数据存储和检索引擎，由于免费及开放具有最佳的价格，其优势使得其对解决当前安全分析领域面临的问题，有得天独厚的优势。

它简化了安全操作工作流程，并通过从业人员驱动的关联性，帮助从业人员最大化数据见解。

安全团队受益于涵盖广泛安全用例的多种检测和调查方法。将基于 EQL（弹性安全检测引擎中高级关联背后的技术）的关联与基于机器学习的检测，指标匹配类型检测规则，以及云规模的第三方上下文相结合，可以实现更全面的安全策略。

而 Elastic 团队也在这方面不断发现、提供了一套完整的解决方案，使得安全分析团队在使用 Elastic 进行安全分析变得更得心应手。

结语

无论一款工具多么强大，都可能存在这样或那样的不足，从来都不存在可以“一劳永逸”的工具。想要真正最大化实现工具的性能，还需要根据自身需求和具备的实际条件来进行选择，而 Elastic Stack 则更适合那些拥有员工，技能和耐心的企业，自行创建解决方案。

当然，寻找一个实力相当的 SIEM 供应商，提供的全面 SOCaaS（SOC-as-a-Service）的快速部署和响应事件，也是个不错的选择。

近来，越来越的企业为避免过多一次性投资的基础上，实现全方位全时段的安全保障，更愿意寻求第三方安全托管机构的 724 SOCaaS 服务。同时，由于夜间被打扰的概率更小，其实 724SOCassS 服务更适合做一些专注需求高的特殊工作，例如漏洞的深度分析、网络情报分析等。

另外，中型市场公司与大型企业，具有相同的安全需求，而没有大型团队和预算，SOC 可以很好的解决这一难题。

4.3.2 读《长安十二时辰》有感——SIEM/SOC 建设要点

创作人：李捷

最近读了马伯庸老师的小说《长安十二时辰》（也有改为《长安二十四时辰》的网剧，之所以改成二十四时辰，我觉得也是非常的不认可原著里面的时间观念吧?

别说是十二时辰，即便是二十四时辰，我还是认为也不可能这么短的时间内构筑这么多事情，这是一部以唐朝为背景，讲述短短二十四小时内发生在长安城内，攻防双方围绕入侵 & 防御、检查 & 规避、攻击 & 应对 等系列主题，展开的一场场惊心动魄故事的小说。

这不仅让我想到了最近一直在研究的 SIEM/SOC 的建设，特此有感，写下本文。

攻防演练的悠久历史

围绕我们安全的主题，特别是具体到企业内部的安全建设，其实对我们绝大多数人来说，并不像想象中的那么的陌生和遥远。细想一下，如果把企业想象成一座城堡的话，攻击方和防守方之间的反复较量，其实就相当于一场持续的、没有硝烟的城堡攻防战争。就像小说《长安十二时辰》里面的故事一样，事情都会围绕长安城这座城市展开。

现实中的现代网络攻防，和历史上的古典城堡攻防事件是类似，对于一个“城堡”的攻防双方来说，从理念和需具备的能力上来讲没有什么新的东西，一切都有迹可循，我们完全能从历史的长河中发现已有的案例，来对标正在发生的事情；也可以在历史中进行归纳总结，得出城堡攻防的关键因素，用于指导现代的网络攻防。

安全分析&网络攻防的本质

我相信，虽然马伯庸老师在准备这本小说的时候，查阅了大量的资料，参考文献、古籍，甚至是去西安实地考察了很多回，但是他一定没有去学习过，如何构建一个安全信息和事件管理系统（SIEM），或是如何建设一个安全运营中心（SOC）。

但让我特别惊奇的是，马老师在写这样一部历史玄幻类小说的时候，居然完完全全的描述出了现代网络攻防双方，所需要到的各种能力和技术。特别是守城一方，在《长安十二时辰》中，马老师大量描述了作为守城方的靖安司所构建的系统，非常独创的杜撰出了一些在唐代不可能具备的能力，但又是现代攻防中必须具备的实实在在的能力。

安全是一个数据问题

首先，安全的本质是一个数据的问题，我们只有对企业内外部，所有的安全关联数据都拥有可见性的时候，才能够有效检查、有效的追踪、有效的分析，然后才能揪出漏洞、渗透、入侵，进而采取行动进行补救、缓解、修复和应对。

因此，我们可以看到，在《长安十二时辰》中，马伯庸老师为主管城防的靖安司配备的第一个能力就是大数据，其最大的特点之一是它是一个巨大的文库，可以随时查阅

朝廷各部的文书资料，施展大案牍术做大数据的搜索。

而在安全里面，构建一个安全的大数据系统也是第一要务，这也是现在 SIEM 和 SOC 的底层核心。安全大数据具有以下要求。

安全数据 —— 不能有盲点

在小说里面，我们可以看到，细到每天进出长安城的人口，货物都有记录，分门别类，记录登录时间，停留地点，货物类型。并且，这些数据都是长久存储的历史数据。因此，我们看到靖安司的全面数据，帮助主角快速查到了突厥狼崽行踪、查到猛火雷原料的去向。

因此，对于企业攻防来说，有全面的数据是开展安全工作的主要前提，无论是即时发现，还是事后追溯，还是可疑事件的威胁捕获，还是用于建模分析的异常检测，都需要全面的实时数据和全量的历史数据。

因此，全面和不能有盲点是指：

- 采集的数据要全
- 保存的数据要久

安全数据 —— 要有规范，且能关联

在小说里面，靖安司不仅有大量的门禁数据，而且可以随时查阅朝廷各部的文书资

料，这里涉及到大量数据的理解和关联的问题，其中有个情节，就是将户部的物品采购记录，结合城门的出入记录，发现了突厥狼卫偷运的猛火雷原料的问题。

通过小说我们不难发现，数据的整合分析是一件非常重要的事情。在现在企业内部，我们有不同的域，也有大量的网络设备，安全设备，数据来自不同的厂商、不同的部门。安全部门要能够通读这些数据，首先就要有一个统一的数据规范和标准，对不同日志提取出来的字段，有一致的解释、命名和类型，然后，要具备跨数据源的关联分析能力。这样，数据才不会成为一个个的数据孤岛，能够协同。《长安十二时辰》里面虽然没有特别强调和描述，但是其设定和从历史古代文书的规范看，就是靖安司（朝廷）必须具备良好规范的数据，并且能够随时调取，并关联分析。

数据系统要有快速分析和处理数据的能力

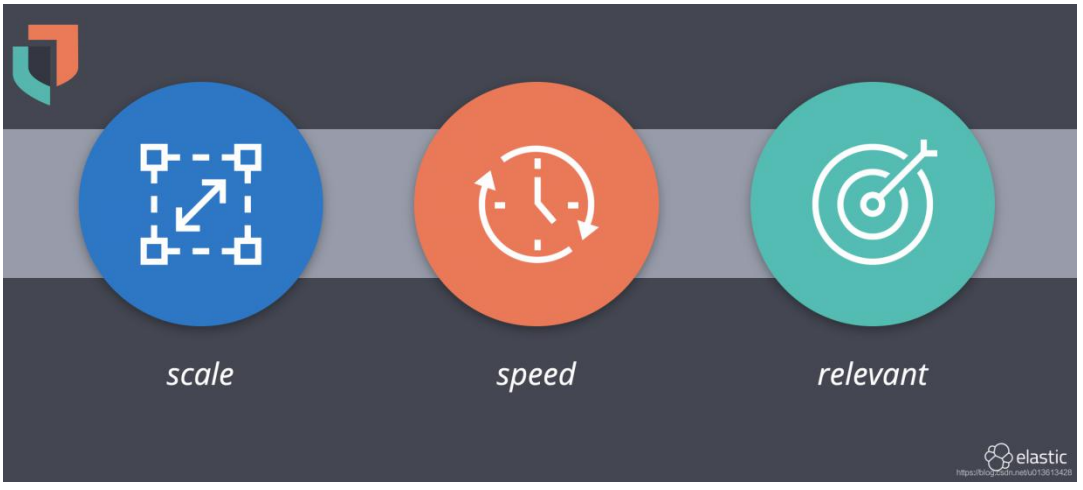
在小说中设定我们拥有足够的数据是比较轻松的事情，但大数据的处理能力，却无法通过大数据系统来提供。马老师在《长安十二时辰》中的设定，是通过最强大脑来提供大数据的处理能力，靖安司中的徐宾能够过目不忘，能施展大案牍术做大数据的搜索，他作为小说中前期的关键人物，在数个场景中通过强大的脑力，快速的找到数据里关于案情相关信息的记录，是推动剧情的关键。因此，安全分析系统必备的能力是要能在海量数据中进行快速检索。

注意，这里强调的点有两个：一是海量数据，前面我们说过了，数据要全面且要保存足够长时间。第二个是速度，速度是关键，天下武功唯快不破，对于安全攻防，速度直接代表了金钱，代表了挽救大厦于将倾的能力。无论是规则引擎的事件扫描，还是威胁捕获时的即兴搜索，都需要查询速度作为支撑

Elastic Security 的大数据基础

整个 Elastic Security 以 Elasticsearch 为基础来构建。我们通过 Elastic Security 来构建企业 SIEM 或者 SOC 的时候，就可以很轻松的应对安全大数据的问题。

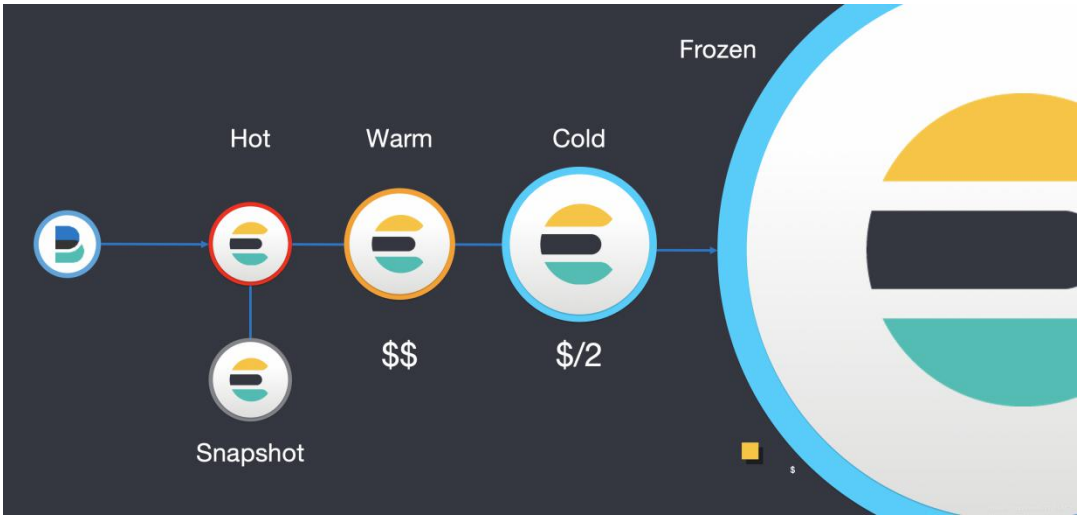
- 首先，Elasticseach 本身是一个分布式的大数据搜索引擎，天然具备横向扩容的能力，可以存储海量的安全数据；
- 并且 Elasticseach 以 json document 的形式存储数据，具备动态可变的 schema/mapping，可以灵活存储不同类型的结构化数据和非结构化数据，并随意添加新的字段，使其能够吸纳任意来源、格式、类型的安全数据；
- 而对比其他具备搜索能力的大数据系统，Elasticseach 的优势在于首屈一指的全文检索、模糊查询能力，结合丰富的多维分析能力，内置的机器学习能力，高并发的支撑能力，再加上海量数据的毫秒级查询响应，使得 Elasticseach 能轻松应对安全大数据分析的工作。



除了避免数据盲点和提供足够的数据分析能力，其实对于构建 SIEM 或者 SOC 的一个重要考虑因素是单位数据的存储成本。

可以明确的一点是，安全运维数据，绝对是一个非常庞大的数据集，它包含来自不同域的数据（生产域，办公域，互联网域等），来自不同设备的数据（应用，硬件，网络，数据库等），并且因为回溯和审计的需求，可能需要保存很长的时间。迫于成本，我们可能会放弃部分数据，进而造成某些事件不可见或不可回溯、审计的问题。

而在最新的 Elasticsearch 7.10 版本上，我们提出了一个完美的解决方案，通过数据层和可搜索快照，我们能以极低的成本来维护所需的所有安全数据。



安全建设需要对内外部环境梳理

马伯庸老师为靖安司配备的第二个能力，就是将其设为一个内外部数据和情报的中枢。

内部环境的梳理

小说中有一重要的概念，就是长安城“108 坊”，靖安司内有长安城的沙盘模型，坐在屋子里，就能俯瞰长安全城。要把整个长安城的所有建筑，通道，马路，下水道都梳理清楚不是一件容易的事，并且这些资产还是可变的，可能有变动，有新增，有删除，虽然不易，但却必须。靖安司构建的这个沙盘模型对于快速的检查安全隐患，分析对手意图、潜入方式、攻击手段等，都有至关重要的作用。

同样，企业内部要能够梳理所有的安全资产、拓扑和漏洞：

- 知道自己拥有哪些资产，动态掌握资产的信息
- 分清楚企业内部的各种边界和域环境
- 明了这些资产在不同域中的上下游网络链路和通道
- 同时，能进行漏洞扫描，知道哪些资产上存在隐患

通过梳理这些数据，再结合我们之前谈到的安全数据（安全信息和事件），我们才能够做到知己知彼。了解自身的结构，有针对性的进行补救和补强，再去分析对手的可能行为，对核心资产和边缘资产进行不同级别的有效防护。

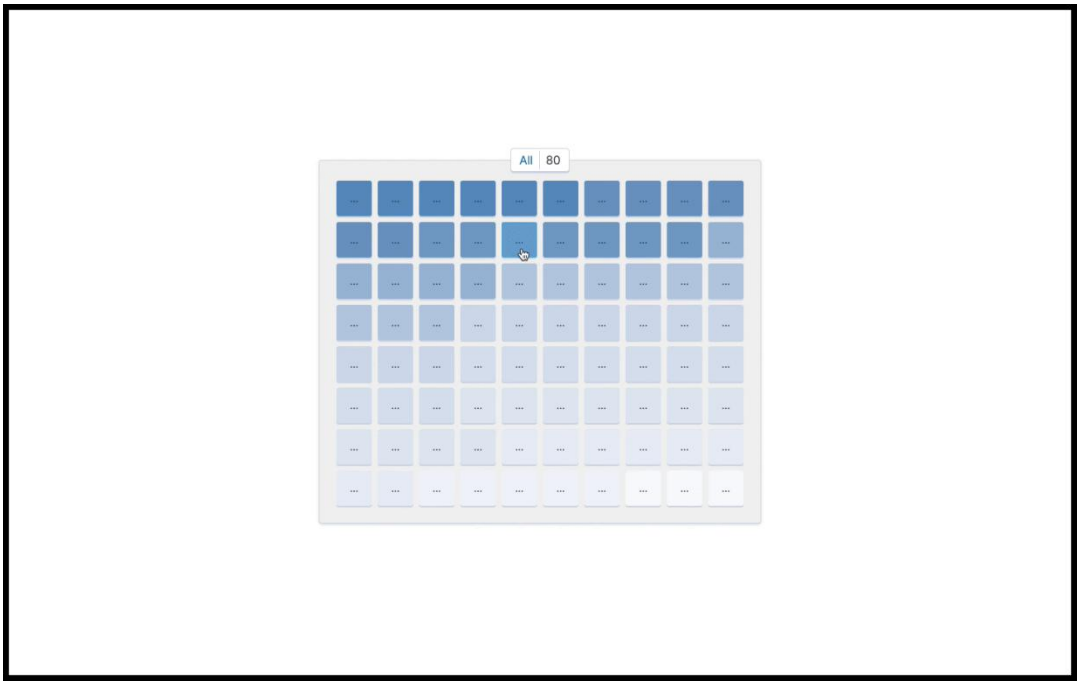
外部情报的汇入

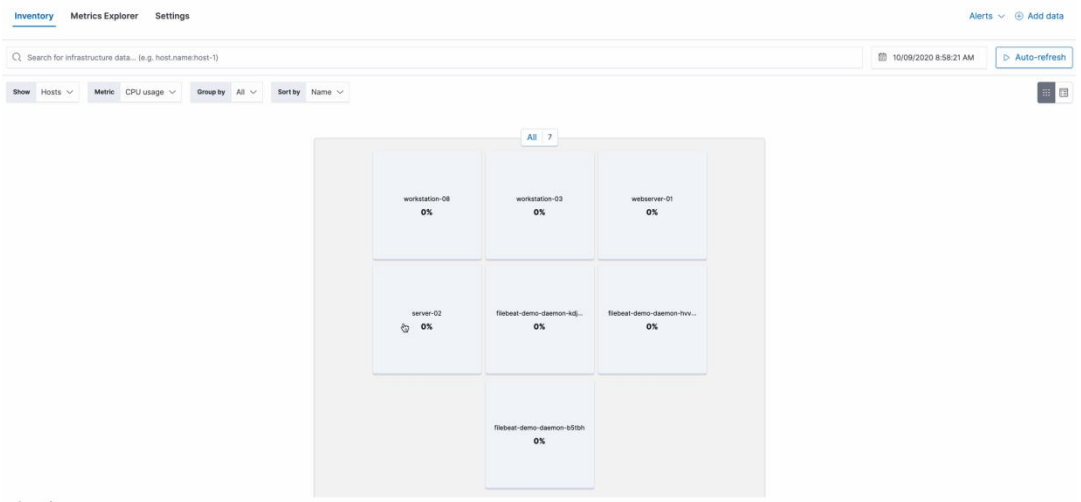
仅了解自己是不够的，还需要足够的了解对手，方能做到百战不殆。小说中，靖安司不仅有长安城的所有信息，还掌握着长安城外的所有情报（对手情报），在突厥狼卫的剧情中，能够及时调阅和突厥相关的信息，了解到突厥可汗，突厥宰相和突厥狼卫的

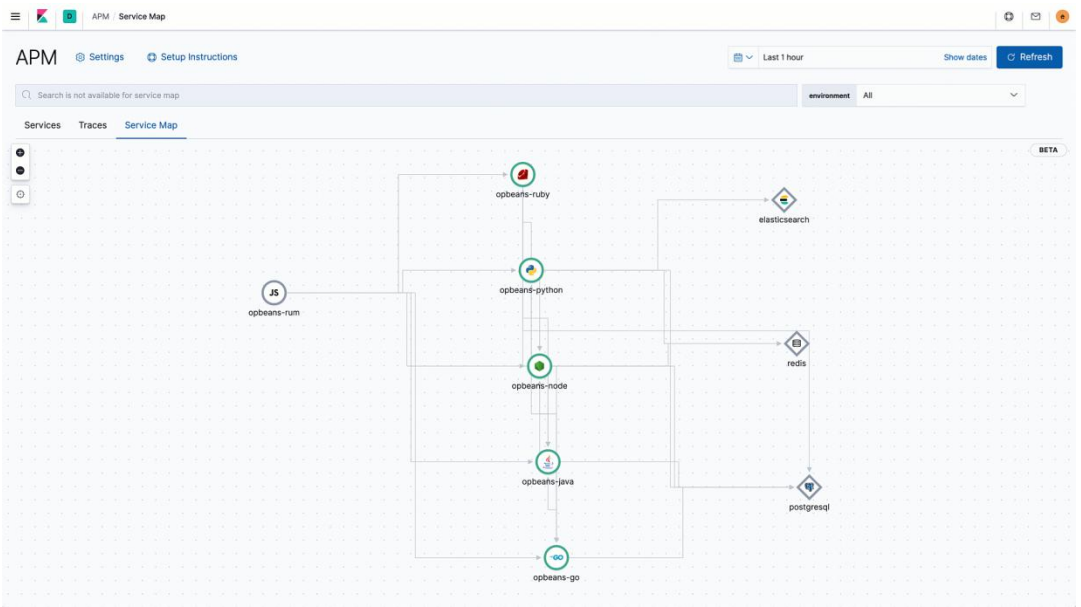
关系，方才知道幕后另有黑手。在现代安全攻防中，外部的威胁情报同样重要，比如，最新发布的零日漏洞，最新威胁的检测方式，最新的黑 ip，黑 url 等各种威胁情报，都是我们了解对手攻击，进行有针对性防御的重要信息

Elastic Security 对安全环境的梳理

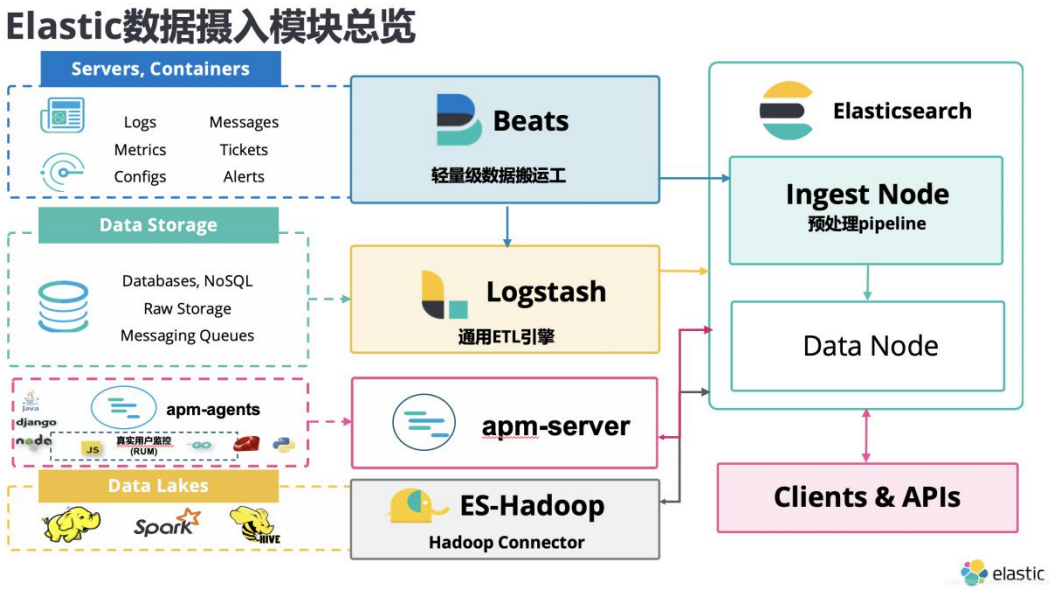
Elastic Security 通过 Elasticsearch 强大的数据吸纳能力，以及丰富的开箱即用的数据源，和 Kibana 上构建的基础架构分析能力，能够有效的帮助掌握内部动态、实时的安全资产信息

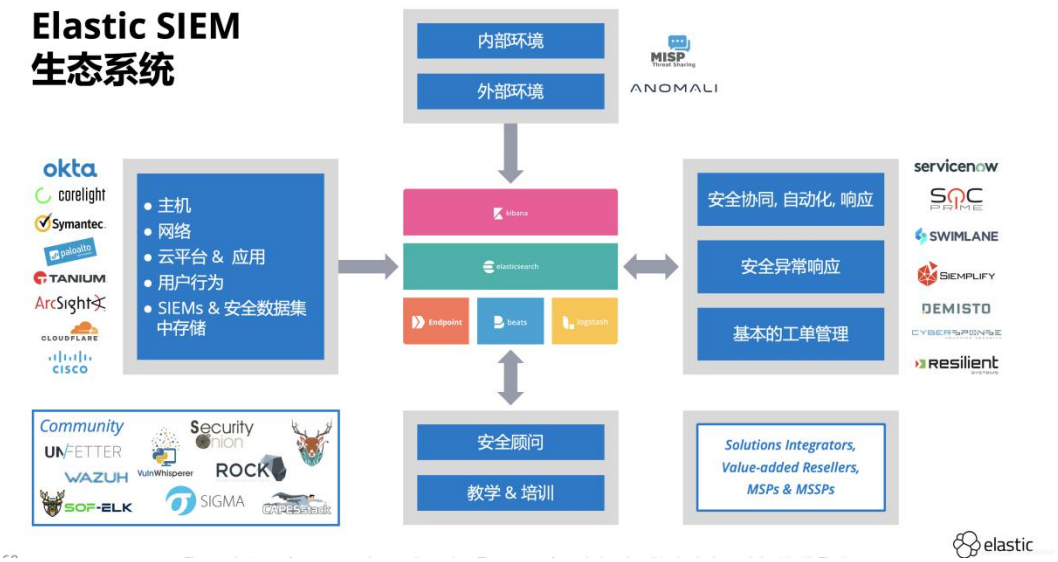






同时，通过功能强大的数据摄入模块，可有效的丰富和完善内外部威胁情报。





完善的监控体系和有效的信息传导

马伯庸老师为靖安司配备的第三个能力是望楼体系。

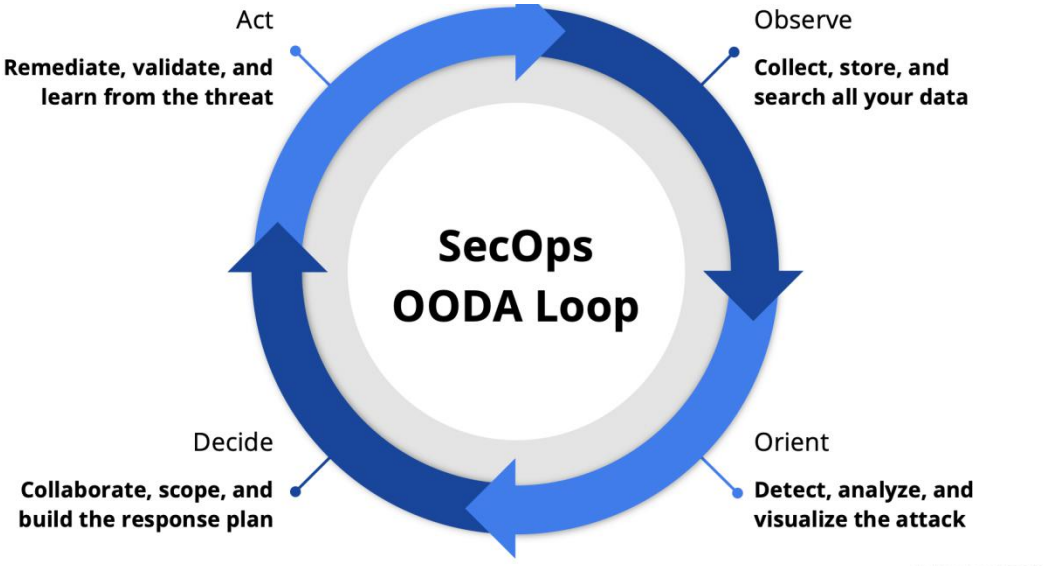
靖安司在长安城的中心，有一座非常高大的建筑叫大望楼，它是望楼体系的核心，围绕大望楼，在整个长安城内一层层的分布着很多小望楼，每个望楼上都有士兵，可以通过旗和鼓声打出暗号，在整个长安城内传递信息。正是有了这个诡异的设定，整个故事才可以架设在十二个时辰的设定上。

这里给我们带来的启示点是：

- 是安全的检测和响应必须要具有实时性，安全事件能够在不同的端点，部门进行快速的流转，并形成闭环。
- 另外要有安全的处理中心，能够统一的快速发起安全相关事件的处置和响应。

因此，构建一个 SIEM/SOC 就是我们企业的靖安司，能够汇聚跟安全相关的信息，一旦检测出安全问题，能够通过这个安全运营中心中的“大望楼”，快速的下发相关操作。

比如，告警快速触达相关的安全分析人员进行告警分析，确认是否是需要处理的安全事件；当确认安全事件后，可以快速的升级，触达到对应的业务，网络，机架等部门进行安全处置和响应，有一个整体且闭环的完成安全运维流程



SIEM/SOC 三要素 —— 人、技术、流程

保卫长安光有个靖安司是不够的，静安司的核心并不是这个机构本身，而是其中的人，技术和流程。

企业安全需要仰仗具备不同能力的安全人员

在小说中，如果没有李泌（主角，Security Director/SOC manager），张小敬（主角，Security Analyst），徐宾，姚汝能（Security Architect/Engineer）等人去规划，执行对应的安全工作，是不可能有最终的对抗胜利的。

我们要始终把安全建设，看成是一种攻防对抗的行为。所有的安全设施都只是工具，是让防御方用来抗衡入侵方的工具，不是依仗的主体。关键还是人和人之间的对抗，安

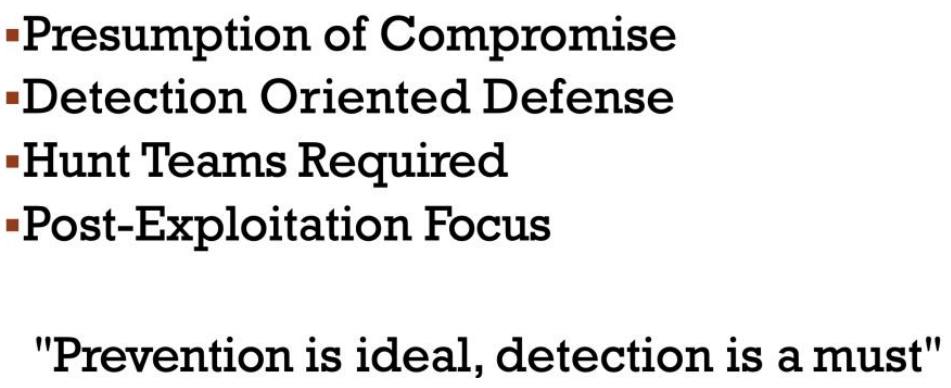
全人员必须划分不同的角色，具备不同的技能。我们既要有安全的主管，又要有捕快；既要有防御工事的工人，又要有巡逻的士兵；

安全人员的思维方式

在小说中，主角李泌和张小敬，从突厥狼卫，到守捉郎，再到蚍蜉，再到最后的幕后黑手。一直的思维方式就是居安思危，一直假设有更加深层次的威胁，才最终能让幕后黑手浮出水面。做安全的，无论是在古代还是现代，永远不能有万事具备，高枕无忧的想法。

以下几点是现在面向各种 APT 攻击的安全人员必须具备的思维方式：

- 假设侵入已经存在
- 以检测（Detection）为导向的防御
- 聚焦于漏洞利用阶段 （post-exploitation focus）



其实，这也是警示安全人员，千万不要沉迷于对已知威胁的防御，那只是工具该干的事。一个城堡总会有意向不到的漏洞，安全设备的边界防御固然重要，但这只是基础，而被渗透是必然，安全人员的工作重心应该是对可疑事件和未知威胁的捕获

必须组建威胁捕获团队

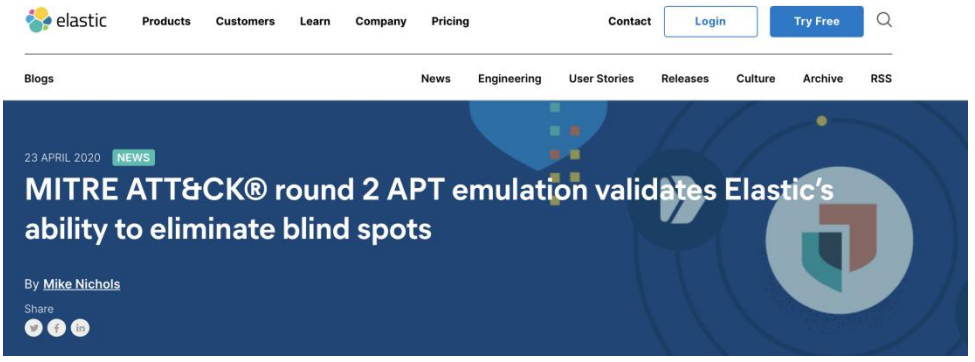
因此，我们看到，在《长安十二时辰》中，整个故事是以李泌和张小敬为主的安全团队，主动出击为引线而展开的。最有效的防御从来不是被动式的，所谓进攻就是最好的防御，无论是以前还是现代，我们都需要有四大名捕这样的捕快能主动出击，针对各种可以的事件，主动探查蛛丝马迹背后隐藏的信息，在一次有计划的攻击（Kill chain）完整完成之前，将其找出，并终止。

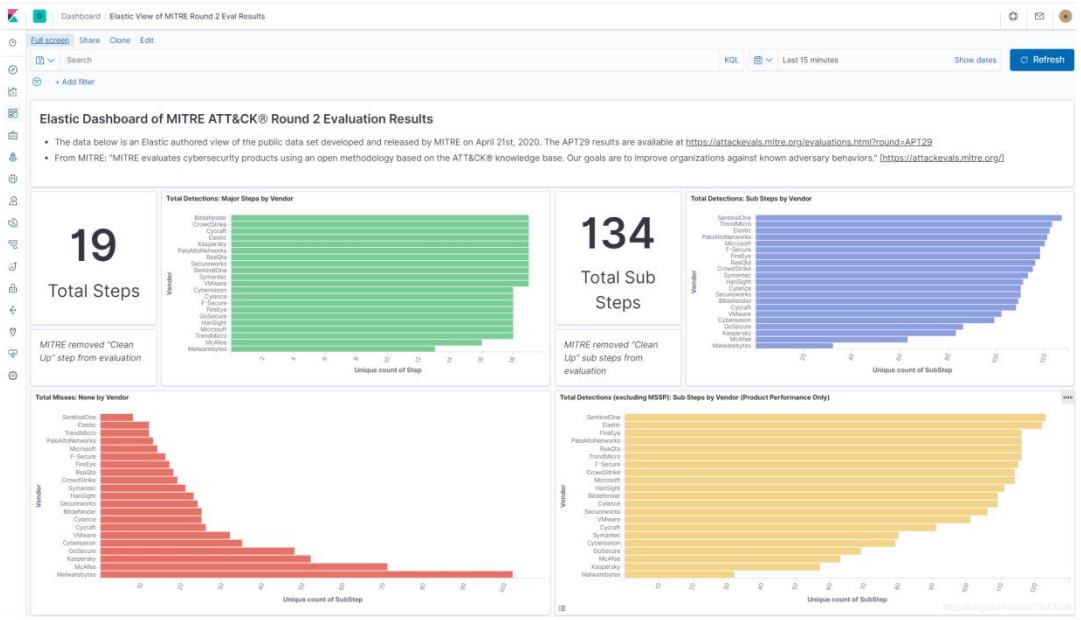
当然，现在不是每个企业都有足够的人力和资源去组建威胁捕获团队，但在财力允许的情况下，这是必须放在日程上的一件事。因为你的对手可能是处心积虑以企业为目标的黑客，这是一场脑力的对决，一个猫鼠的游戏，不是仅仅依赖购买几个设备和工具（捕鼠夹）就能稳操胜券的对局。

当然，我们仍然要为威胁捕获团队配备即兴搜索所需要的高效分析工具：

技术的对决

在现代的安全攻防中，不仅仅是脑力的对决，也是一场技术的对决，攻防双方都可能使用最新的技术，穷尽不同的方法，尝试去达到各自的目标。黑客也会使用云技术，也会用大数据，用机器学习生产 DGA 算法，会发动大量的僵尸机、肉鸡进行饱和攻击，同理，防守方也需要掌握对应的技术，进行有效的检测和响应。EDR，NTA，UEBA 都是构建 SOC 必须具备的技术能力，而 MITRE ATT&CK 则是知识库，帮助我们更好的了解对手的策略，技术和流程。



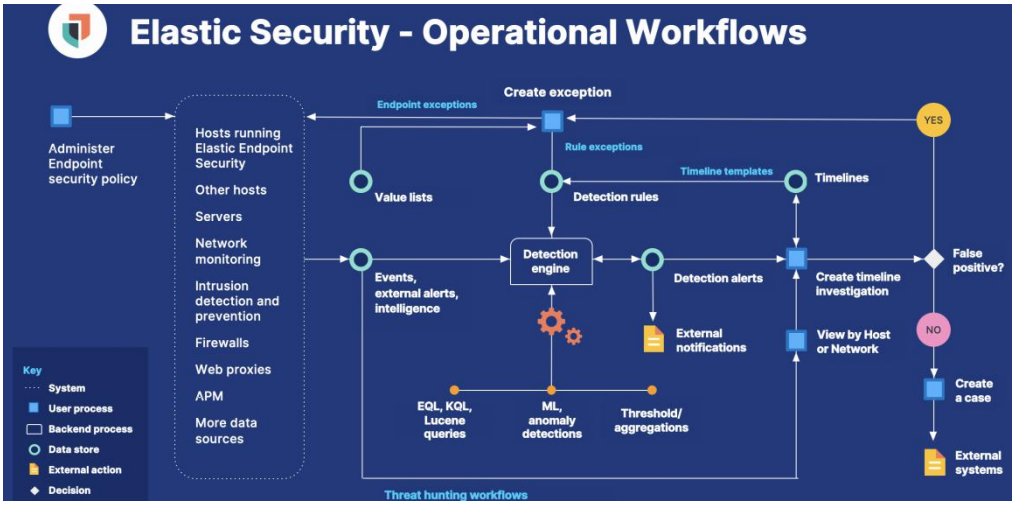


能够闭环处置的流程

在《长安十二时辰》中，我们也看到，整个长安城的安全，绝不仅仅是靖安司一个部门就能保证的。在不同的场景需要不同部门的参与，比如最终的灯会，我们看到有户部，工部，旅贲军，右骁卫，龙武卫，禁军等各个部门都需要参与其中，户部安排民众的出游时间，路线；工部负责灯会工程的建设，旅贲军，右骁卫，龙武卫，禁军有各自的防御任务，靖安司处置安全事件，需要其他部门的配合，否则就根本没有办法在熙熙攘攘，复杂运转的长安城中有效保证安全。

回到我们的现代网络攻防，当我们发生一个安全事件的时候，绝不仅仅是安全部门的事情，会涉及到业务，机架，网络，中间件，运维等不同部门，只有形成一个能闭环处置的流程，由安全发起，到最终安全检验完成，中间通过很多部门的配合才能够完成时间的处置。因此，有一个行之有效的闭环流程非常的重要，我们在构建一个 SIEM/

SOC 的时候，就需要工具有这样的能力：能发起工单，对接企业内部的流程系统（ITOM），能设置合理的权限，能让不同的部门在统一的数据上协作。才能让流程有效运行起来



而在单一系统上构建这样的能力，更是能够事半功倍。

SIEM/SOC 的建设的注意事项：

通过分析《长安十二时辰》这个小说里面描写的惊心动魄的古代长安城的攻防场景，我们可以更加直观的了解到现今企业安全攻防，特别是现在国家级别黑客组织的兴起，各种 APT 组织，使用更为完善的攻击工具，更为系统的社交工程，更为丰富、复杂、耐心的攻击方式和链路，让我们的企业安全面临极大的挑战，就像《长安十二时辰》里面，一层套一层的攻击，不到最后，你都不知道究竟是谁在攻击你，究竟他潜伏了多久。

对手在持续的进步，持续的发展，因此，一个 SIEM/SOC 的建设也是一个持续优化和改进的过程。并且这里必须是人、技术和流程三位一体的持续改进，如果没有持续的投入是很难获得成效的。

可以查看 Elastic Security 7.10 上的最新功能，了解 Elastic Security 带给我们的新能力。

Elastic Security 7.10：https://www.elastic.co/blog/whats-new-elastic-security-7-10-0-correlation-cloud-visibility-detection

创作人简介：

李捷，专注于 Elastic Stack 的解决方案的设计和咨询。13 年软件行业从业经验，从嵌入式开发，到后端 J2EE 应用和前端界面开发。从爬虫脚本，到区块链和大数据分析。从开发工程师，到测试工程师和项目经理。 拥有全栈开发经验和丰富的项目实施经验，同时也是一个活跃的知识分享者和社区活动者。

博客：https://lex-lee.blog.csdn.net/

4.4 性能优化场景

4.4.1 Elasticsearch 生产环境集群部署最佳实践

创作人：铭毅天下

在生产环境搭建或维护 Elasticsearch 集群和个人搭建集群的小打小闹有非常大的不同。

本文的最佳实践基于每天增量数亿+ 的线上环境。

内存

Elasticsearch 和 Lucene 都是 Java 语言编写，这意味着我们必须注意堆内存的设置。

Elasticsearch 可用的堆越多，它可用于过滤器（filter）和其他缓存的内存也就越多，更进一步讲可以提高查询性能。

但请注意，过多的堆可能会使垃圾回收暂停时间过长。请勿将堆内存的最大值设置为 JVM 用于压缩对象指针（压缩的 oops）的临界值之上，确切的临界值有所不同，但不要超过 32 GB

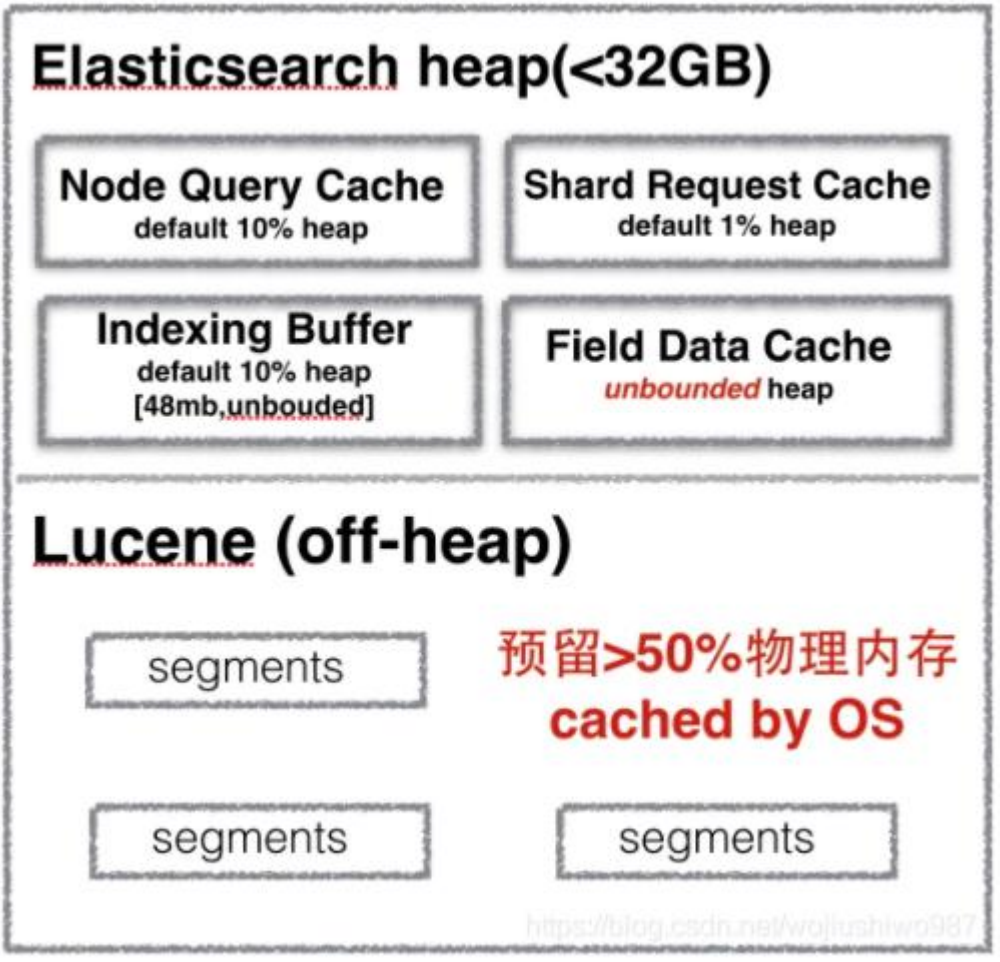
常见内存配置坑 1：堆内存设置过大

举例：Elasticsearch 宿主机：64 GB 内存，堆内存恨不得设置为 64 GB。

这忽略了堆的另一部分内存使用大户：OS 文件缓存。Lucene 旨在利用底层操作系统来缓存内存中的数据结构。Lucene 段存储在单独的文件中。由于段是不可变的（immutable），因此这些文件永远不会更改。这使它们非常易于缓存，并且底层操作系统很乐意将热段驻留在内存中，以加快访问速度。

这些段包括倒排索引（用于全文搜索）和 doc values 正排索引（用于聚合）。Lucene 的性能取决于与 OS 文件缓存的交互。如果你将所有可用内存分配给 Elasticsearch 的堆，则 OS 文件缓存将不会剩下任何可用空间。这会严重影响性能。

官方标准建议是：将 50% 的可用内存（不超过 32 GB，一般建议最大设置为：31 GB）分配给 Elasticsearch 堆，而其余 50% 留给 Lucene 缓存。



可以通过以下方式配置 Elasticsearch 堆：

方式一：堆内存配置文件 jvm.options

```
# Xms represents the initial size of total heap space
# Xmx represents the maximum size of total heap space
-Xms16g
-Xmx16g
```

方式二：启动参数设置

```
ES_JAVA_OPTS="-Xms10g -Xmx10g" ./bin/elasticsearch
```

CPU

运行复杂的缓存查询、密集写入数据都需要大量的 CPU，因此选择正确的查询类型以及渐进的写入策略至关重要。

一个节点使用多个线程池来管理内存消耗。与线程池关联的队列，使待处理的请求得以保留（类似缓冲效果）而不是被丢弃。

由于 Elasticsearch 会做动态分配，除非有非常具体的要求，否则不建议更改线程池和队列大小。

推荐阅读：https://www.elastic.co/guide/en/elasticsearch/reference/current/modules-threadpool.html

分片数

分片是 Elasticsearch 在集群内分发数据的单位。集群发生故障再恢复平衡的速度，取决于分片的大小、分片数量、网络以及磁盘性能。

在 Elasticsearch 中，每个查询在每个分片的单个线程中执行。但是，可以并行处理多个分片。针对同一分片的多个查询和聚合也可以并行处理。这意味着在不涉及缓存的情况下，最小查询延迟将取决于数据、查询类型以及分片的大小三个因素。

设置很多小分片 VS 设置很少大分片？

- 查询很多小分片，导致每个分片能做到快速响应，但是由于需要按顺序排队和处理结果汇集。因此不一定比查询少量的大分片快。
- 如果存在多个并发查询，那么拥有大量小分片也会降低查询吞吐量。

所以，就有了下面的分片数如何设定的问题？

分片数设定

选择正确数量的分片是一个复杂问题，因为在集群规划阶段以及在数据写入开始之前，一般不能确切知道文档数。

对于集群而言，分片数多了以后，索引和分片管理，可能会使主节点超载，并可能会导致集群无响应，甚至导致集群宕机。

建议：为主节点（Master 节点）分配足够的资源，以应对分片数过多可能导致的问题。

必须强调的是：主分片数是在索引创建时定义的，不支持借助 update API 实现类

副本数更新的动态修改。创建索引后，更改主分片数的唯一方法是重新创建索引，然后将原来索引数据 reindex 到新索引。

官方给出的合理的建议：每个分片数据大小：30GB-50GB。

- 推荐 1：Elasticsearch 究竟要设置多少分片数?
  https://elastic.blog.csdn.net/article/details/78080602

- 推荐 2：Elasticsearch 之如何合理分配索引分片
  https://qbox.io/blog/optimizing-elasticsearch-how-many-shards-per-index

副本

Elasticsearch 通过副本实现集群的高可用性，数据在数据节点之间复制，以实现主分片数据的备份，因此即便部分节点因异常下线，也不会导致数据丢失。

默认情况下，副本数为 1，但可以根据产品高可用要求将其增加。副本越多，数据的容灾性越高。

副本多的另一个优点是，每个节点都拥有一个副本分片，有助于提升查询性能。

铭毅提醒：

- 实际副本数增多，提高查询性能建议结合集群做下测试，我实测过效果不明显。
- 副本数增多，意味着磁盘存储要加倍，也考验硬盘空间和磁盘预算。

建议：根据业务实际综合考虑设置副本数。普通业务场景（非精准高可用）副本设置为 1 足够了。

冷热集群架构配置

根据产品业务数据特定和需求，我们可以将数据分为热数据和冷数据，这是冷热集群架构的前提。

访问频率更高的索引，可以分配更多更高配（如：SSD）的数据节点，而访问频率较低的索引可以分配低配（如：机械磁盘）数据节点。

冷热集群架构对于存储，诸如应用程序日志或互联网实时采集数据（基于时间序列数据）特别有用。

数据迁移策略：通过运行定时任务，来实现定期将索引移动到不同类型的节点。

具体实现：Curator 工具或借助 ILM 索引生命周期管理。

热节点

热节点是一种特定类型的数据节点，关联索引数据是：最近、最新、最热数据。

因为这些热节点数据通常倾向于最频繁地查询。热数据的操作会占用大量 CPU 和 IO 资源，因此对应服务器需要功能强大（高配）并附加 SSD 存储支持。

针对集群规模大的场景，建议：至少运行 3 个热节点以实现高可用性。

当然，这也和你实际业务写入和查询的数据量有关系，如果数据量非常大，可能会需要增加热节点数目。

冷节点（或称暖节点）

冷节点是对标热节点的一种数据节点，旨在处理大量不太经常查询的只读索引数据。由于这些索引是只读的，因此冷节点倾向于使用普通机械磁盘而非 SSD 磁盘。

与热节点对标，也建议：最少 3 个冷节点以实现高可用性。同样需要注意的是，若集群规模非常大，可能需要更多节点才能满足性能要求。

甚至需要更多类型，如：热节点、暖节点、冷节点等。

强调一下：CPU 和 内存的分配最终需要你通过使用与生产环境中类似的环境借助 esrally 性能测试工具测试确定，而不是直接参考各种最佳实践拍脑袋而定。

有关热节点和热节点的更多详细信息，请参见：https://www.elastic.co/blog/hot-warm-architecture-in-elasticsearch-5-x

节点角色划分

Elasticsearch 节点核心可分为三类：主节点、数据节点、协调节点。

主节点

主节点：如果主节点是仅是候选主节点，不含数据节点角色，则它配置要求没有那么高，因为它不存储任何索引数据。

如前所述，如果分片非常多，建议主节点要提高硬件配置。

主节点职责：存储集群状态信息、分片分配管理等。

同时注意，Elasticsearch 应该有多个候选主节点，以避免脑裂问题。

数据节点

数据节点职责：CURD、搜索以及聚合相关的操作。

这些操作一般都是 IO、内存、CPU 密集型。

协调节点

协调节点职责：类似负载平衡器，主要工作是将搜索任务分发到相关的数据节点，

并收集所有结果，然后再将它们汇总并返回给客户端应用程序。

节点配置参考

下表参见官方博客 PPT：

角色	描述	存储	内存	计算	网络
数据节点	存储和检索数据	极高	高	高	中
主节点	管理集群状态	低	低	低	低
Ingest	节点 转换输入数据	低	中	高	中
机器学习节点	机器学习	低	极高	极高	中
协调节点	请求转发和合并检索结果	低	中	中	中

不同节点角色配置如下

必须配置到：elasticsearch.yml 中。

主节点：

```
node.master:true  node.data:false
```

数据节点：

```
node.master:false  node.data:true
```

协调节点：

```
node.master:false  node.data:false
```

故障排除提示

Elasticsearch 的性能在很大程度上取决于宿主机资源情况。CPU、内存使用率和磁盘 IO 是每个 Elasticsearch 节点的基本指标。建议你在 CPU 使用率激增时查看 Java 虚拟机（JVM）指标。

堆内存使用率高

高堆内存使用率压力以两种方式影响集群性能：

堆内存压力上升到 75% 及更高

- 剩余可用内存更少，并且集群现在还需要花费一些 CPU 资源以通过垃圾回收来回收内存。
- 在启用垃圾收集时，这些 CPU 周期不可用于处理用户请求。结果，随着系统变得越来越受资源约束，用户请求的响应时间增加。

堆内存压力继续上升并达到接近 100%

- 将使用更具侵略性的垃圾收集形式，这将反过来极大地影响集群响应时间。
- 索引响应时间度量标准表明，高堆内存压力会严重影响性能。

非堆内存使用率增长

JVM 外非堆内存的增长，吞噬了用于页面缓存的内存，并可能导致内核级 OOM。

监控磁盘 IO

由于 Elasticsearch 大量使用存储设备，磁盘 IO 的监视是所有其他优化的基础，发现磁盘 IO 问题并对相关业务操作做调整可以避免潜在的问题。

应根据引起磁盘 IO 的情况评估对策，常见优化磁盘 IO 实战策略如下：

- 优化分片数量及其大小
- 段合并策略优化
- 更换普通磁盘为 SSD 磁盘
- 添加更多节点

合理设置预警

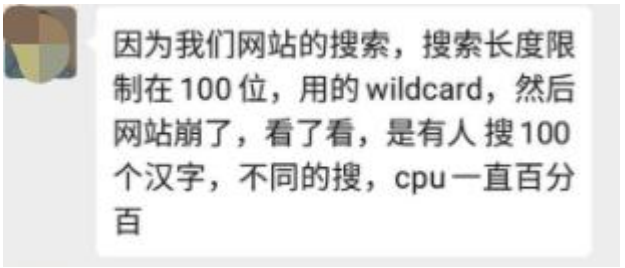
对于依赖搜索的应用程序，用户体验与搜索请求的等待时间长短相关。

有许多因素会影响查询性能，例如：

- 构造查询方式不合理
- Elasticsearch 集群配置不合理
- JVM 内存和垃圾回收问题
- 磁盘 IO 等

查询延迟是直接影响用户体验的指标，因此请确保在其上放置一些预警操作。

举例：线上实战问题：



如何避免？以下两个核心配置供参考：

```
PUT _cluster/settings
{
  "transient": {
    "search.default_search_timeout": "50s",
    "search.allow_expensive_queries": false
  }
}
```

需要强调的是："search.allow_expensive_queries" 是 7.7+ 版本才有的功能，早期版本会报错。

推荐阅读：

- https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl-wildcard-query.html
- https://www.elastic.co/guide/en/elasticsearch/reference/current/search-your-data.html

合理配置缓存

默认情况下，Elasticsearch 中的大多数过滤器都是高速缓存的。

这意味着在第一次执行过滤查询时，Elasticsearch 将查找与过滤器匹配的文档，并使用该信息构建名为“bitset”的结构。

存储在 bitset 中的数据包含文档标识符以及给定文档是否与过滤器匹配。

具有相同过滤器的查询的后续执行将重用存储在 bitset 中的信息，从而通过节省 IO 操作和 CPU 周期来加快查询的执行速度。

建议在查询中使用 filter 过滤器。

合理设置刷新频率

刷新频率（refresh_interval）和段合并频率与索引性能密切相关，此外，它们还会影响整个集群的性能。

刷新频率需要根据业务需要合理设置，尤其频繁写入的业务场景。

启动慢查询日志

启用慢查询日志记录将有助于识别哪些查询慢，以及可以采取哪些措施来改进它们，这对于通配符查询特别有用。

增大 ulimit 大小

增加 ulimit 大小以允许最大文件数，这属于非常常规的设置。

在 /etc/profile 下设置：

```
ulimit -n 65535
```

合理设置交互内存

当操作系统决定换出未使用的应用程序内存时，ElasticSearch 性能可能会受到影响。

通过 elasticsearch.yml 下配置：

```
bootstrap.mlockall: true
```

禁用通配符模糊匹配删除索引

禁止通过通配符查询删除所有索引。

为确保某人不会对所有索引（* 或 _all）发出 DELETE 操作，设置如下：

```
PUT /_cluster/settings
{
  "persistent": {
    "action.destructive_requires_name": true
  }
}
```

此时如果我们再使用通配符删除索引，举例执行如下操作：

```
DELETE join_*
```

会报错如下：

```
{
  "error" : {
    "root_cause" : [
```

```
        {
          "type" : "illegal_argument_exception",
          "reason" : "Wildcard expressions or all indices are not allowed"
        }
      ],
      "type" : "illegal_argument_exception",
      "reason" : "Wildcard expressions or all indices are not allowed"
    },
    "status" : 400
}
```

常用指标监视 API

集群健康状态 API

```
GET _cluster/health?pretty
```

索引信息 API

```
GET _cat/indices?pretty&v
```

节点状态 API

```
GET _nodes?pretty
```

主节点信息 API

```
GET _cat/master?pretty&v
```

分片分配、索引信息统计 API

```
GET _stats?pretty
```

节点状态信息统计 API

统计节点的 jvm，http，IO 统计信息。

```
GET _nodes/stats?pretty
```

大多数系统监视工具（如 Kibana、cerebro 等）都支持 Elasticsearch 的指标聚合。

建议使用此类工具持续监控集群状态信息。

小结

Elasticsearch 具有很好的默认配置以供新手快速上手、入门。但是，一旦到了线上业务实战环境，就必须花费一些时间来调整设置以满足实际业务功能要求以及性能指标要求。

建议你参考本文建议并结合官方文档修改相关配置，以使得集群整体部署最优。

4.4.2 Elasticsearch 开发人员最佳实践指南

创作人：铭毅天下

几个月以来，我一直在记录自己开发 Elasticsearch 应用程序的最佳实践。本文梳理的内容试图传达 Java 的某些思想，我相信其同样适用于其他编程语言。我尝试尽量避免重复教程和 Elasticsearch 官方文档中已经介绍的内容。

本文梳理的内容都是从线上实践问题和个人总结的经验汇总得来的。

文章从以下几个维度展开讲解：

- 映射（Mapping）
- 设置（Setting）
- 查询方式（Querying）
- 实战技巧（Strategy）

映射（Mapping）

避免使用 Nested 类型

每个 Elasticsearch 文档都对应一个 Lucene 文档。

Nested 类型是个例外，对于 nested 类型，每个字段都作为单独的文档存储与父 Lucene 的关联。

其影响是：

- Nested 与父文档中的字段相比，查询字段的速度较慢
- 检索匹配 Nested 字段会降低检索速度
- 一旦更新了包含 Nested 字段的文档的任何字段（与是否更新嵌套字段无关，则所有基础 Lucene 文档（父级及其所有 Nested 子级）都需要标记为已删除并重写）。除了降低更新速度外，此类操作还会产生大量垃圾文件，直到通过段合才能进行清理。

在某些情况下，你可以将 Nested 字段展平。

例如，给定以下文档：

```
{
    "attributes": [
        {"key": "color", "val": "green"},
        {"key": "color", "val": "blue"},
        {"key": "size", "val": "medium"}
    ]
}
```

展平如下：

```
{
    "attributes": {
        "color": ["green", "blue"],
        "size": "medium"
    }
}
```

Mapping 设置 strict

实际业务中，如果不明确设定字段类型，Elasticsearch 有动态映射机制，会根据插入数据自动匹配对应的类型。

假定本来准备插入浮点型数据，但由于第一个插入数据为整形，Elasticsearch 自定会判定为 long 类型，虽然后续数据也能写入，但很明显“浮点类型”只阉割保留了整形部分。

铭毅给个 Demo 一探究竟：

```
POST my_index03/_doc/1
{
  "tvalue":35
}

POST my_index03/_doc/2
{
  "tvalue":3.1415
```

```
}

GET my_index03/_mapping

GET my_index03/_search
{
  "query": {
    "term": {
      "tvalue": {
        "value": 3.1415
      }
    }
  }
}
```

注意：term 查询是不会返回结果的。

所以，实战环境中，Mapping 设定要注意如下节点：

- 显示的指定字段类型
- 尽量避免使用动态模板（dynamic-templates）
- 禁用日期检测 （date_detection），默认情况下处于启用状态。“strict” 实践举例：

```
PUT my_index
{
  "mappings": {
```

```
    "dynamic": "strict",
    "properties": {
      "user": {
        "properties": {
          "name": {
            "type": "text"
          },
          "social_networks": {
            "dynamic": "strict",
            "properties": {
              "network_id": {
                "type": "keyword"
              },
              "network_name": {
                "type": "keyword"
              }
            }
          }
        }
      }
    }
}
```

合理的设置 string 类型

Elasticsearch 5.X 之后，String 被分成两种类型，text 和 keyword。两者的区别：

- text：适用分词全文检索场景
- keyword：适用字符串的精准匹配场景

默认，如果不显示指定字段类型，字符串类型自定映射后的 Mapping 如下所示：

```
"cont" : {
            "type" : "text",
            "fields" : {
              "keyword" : {
                "type" : "keyword",
                "ignore_above" : 256
              }
            }
```

而公司实战的业务场景，通常会面临：

- 需不需要分词，不需要的话仅保留 keyword 即可。
- 需要用什么分词？英文分词还是中文分词？
- 分词后是否还需要排序和聚合，即 fielddata 是否需要开启？
- 是否需要精准匹配，即是否需要保留 keyword？

所以，回答了如上几个问题，再有针对的显示设定 string 类型的 Mapping 方为上策。

设置（Setting）

在这里我分享了 Elasticsearch 集群设置相关的技巧。

避免过度分片

分片是 Elasticsearch 的最大优势之一，即将数据分散到多个节点以实施并行化。关于这个主题有过很多讨论。

但请注意，索引的主分片一旦设置便无法更改（除非重建索引或者 reindex）。对于新来者来说，过度分片是一个非常普遍的陷阱。在做出任何决定之前，请确保先通读官方的这篇博文：

我在 Elasticsearch 集群内应该设置多少个分片？

https://www.elastic.co/cn/blog/how-many-shards-should-i-have-in-my-elasticsearch-cluster

铭毅提示：

主分片数过多:

- 批量写入或者查询请求被分割成过多的子写入、子查询，导致索引的写入、查询拒绝率上升。

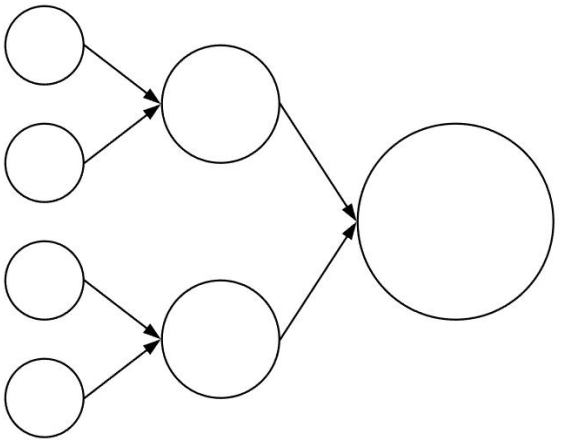
主分片数过少：

- 尤其对于数据量非常庞大的索引，若分片数过少或者就 1 个分片，会导致无法利用集群多节点资源（也就是分布式特性），造成资源利用率不高或者不均衡，影响写入或者查询效率。
- 并且，一旦该大的主分片出现问题，恢复起来耗时会非常长。

取消学习任何段合并的技巧

从本质上讲，Elasticsearch 是另一种分布式 Lucene 产品，就像 Solr 一样 。在底层，大多数时候，每个 Elasticsearch 文档都对应一个 Lucene 文档（ nested 除外）。在 Lucene 中，文档存储在 segment 中。后台的 Elasticsearch 通过以下两种模式连续维护这些 Lucene 段：

- 在 Lucene 中，当你删除或更新文档时，旧文档被标记为已删除，而新文档被创建。Elasticsearch 会跟踪这些标记为 deleted 的文档，适时对其段合并。
- 新添加的文档可能会产生大小不平衡的段。Elasticsearch 可能会出于优化目的而决定将它们合并为更大的段。



实战中一定要注意：段合并是高度受磁盘 I/O 和 CPU 约束的操作。作为用户，我们不想让段合并破坏 Elasticsearch 的查询性能。

事实上，在某些情况下可以完全避免使用它们：一次构建索引，不再更改它。尽管在许多应用场景中可能很难满足此条件。一旦开始插入新文档或更新现有文档，段合并就成为不可避免的一部分。

正在进行的段合并可能会严重破坏集群的总体查询性能。在 Google 上进行随机搜索，你会发现许多人发帖求助求助：“在段合并中减少对性能的影响的配置“，还有许多人共享某些适用于他们的配置。但很多配置都是早期 1.x，2.X 版本的设置，新版本已经废弃。

综上我进行段合并的经验法则如下：

- 取消学习任何段合并的技巧。早期版本的段合并配置是与 Elasticsearch 的内部紧密耦合的操作，新版本一般不再兼容。几乎没有“神秘”的底层配置修改可以使它运行得更快。
- 找到 translog flush 的最优配置 。尝试调整 index.translog.sync_interval 和 index.translog.flush_threshold_size 设置。

详见：https://www.elastic.co/guide/en/elasticsearch/reference/current/index-modules-translog.html

动态调整 index.refresh_interval 以满足业务需求。如果实时性要求不高，可以调大刷新频率（默认是 1s，可以调到 30s 甚至更大）。

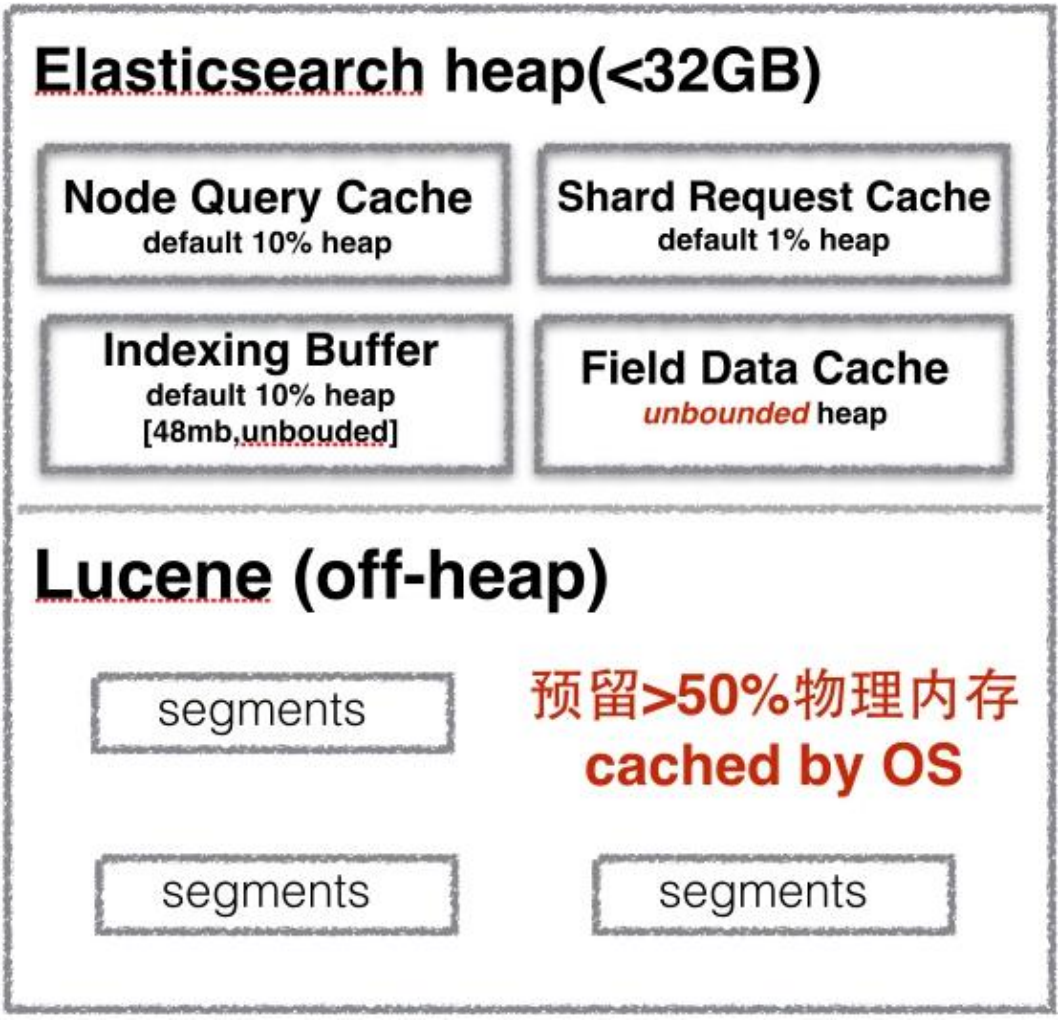
```
PUT /twitter/_settings
{
"index" : {
"refresh_interval" : "30s"
}
}
```

注意 JVM 内存设置

Elasticsearch 可以根据两个主要内存设置产生引人注目的性能特征：

JVM 堆空间——主要用途：缓存（节点缓存、分片请求缓存、Field data 缓存以及索引缓存）

堆外内存空间—— Lucene 段文件缓存



提醒你不要根据过去的非 Elasticsearch JVM 应用程序经验来盲目设置 Elasticsearch JVM 堆大小。

详见官方文档：https://www.elastic.co/guide/en/elasticsearch/reference/current/heap-size.html

查询方式（Querying）

下面我收集了一些技巧，你可以在 Elasticsearch 查询时使用它们。

Elasticseach 里面多线程修改如何保证数据准确性？

1,用如下两个参数校验冲突

```
UT products/_doc/1?if_seq_no=1&if_primary_term=1
{ "title":"iphone", "count":100 }
```

2,用 version 避免冲突

```
PUT products/_doc/1?version=30000&version_type=external
{ "title":"iphone", "count":100 }
```

尝试分割复杂的查询，并行执行提升性能

如果你同时具有过滤器和聚合组件的复杂查询，则在大多数情况下，可以将它们拆分为多个查询并并行执行它们可以提高查询性能。

也就是说，在第一个查询中，仅使用过滤器获取匹配，然后在第二个查询中，仅获取聚合结果而无需再获取检索结果，即 size: 0。

了解你的数字类型，防止被优化导致精度损失

许多 JSON 解析器可以进行各种优化，以提供有效的读/写性能。但可能造成了精度的损失，所以在选型 Jackson json 解析器时：优先使用 BigDecimal 和 BigInteger

不要使用 Elasticsearch Transport / Node 客户端

TransportClient 可以支持 2.x，5.x 版本，TransportClient 将会在 Elasticsearch 7.X 版本弃用并在 8.X 版本中完成删除.

官方推荐使用 Java High Level REST Client，它使用 HTTP 请求而不是 Java 序列化请求。为了安全起见，坚持使用 HTTP 上的 JSON 格式，而不使用 SMILE（二进制格式）

使用官方的 Elasticsearch High-level REST 客户端

非官方客户端一般更新太慢，几乎无法跟上 Elasticsearch 新版本的特性，如：Jest 客户端近一年几乎没有更新，只支持到 6.X 版本。

相比之下，官方 REST 客户端仍然是你相对最好的选择。https://www.elastic.co/guide/en/elasticsearch/client/java-rest/current/index.html

不要使用 HTTP 缓存来缓存 Elasticsearch 响应结果

由于便利性和低进入门槛，许多人陷入了将 HTTP 缓存（例如 Varnish http://varnish-cache.org/）置于 Elasticsearch 集群前面的陷阱。使用 HTTP 缓存缺点如下：

在生产环境中使用 Elasticsearch 时，由于各种原因如：弹性扩展、测试和线上环境分离、零停机升级等，你很有可能最终会拥有多个集群。

（1）一旦为每个集群提供专用的 HTTP 缓存，99%的缓存内容是重复的。

（2）如果你决定对所有集群使用单个 HTTP 缓存，那么很难以编程方式配置 HTTP 缓存以适应不断变化的集群状态的需求。

- 如何传达集群负载以使缓存平衡流量？
- 如何配置计划内或手动停机时间？
- 在维护时段期间，如何使缓存逐渐从一个集群迁移到另一个集群？

这些都是亟待考虑的问题。如上所述，HTTP 缓存很难以编程方式进行实现。当你需要手动删除一个或多个条目时，它并不总是像 DELETE FROM cache WHERE keysIN (...) 查询那样容易。还得通过手动实现。

铭毅提示：这一条我实际没有用过，有用过的童鞋可以留言讨论。

使用基于 _doc 排序的 slice scroll 遍历数据

Scrolls 是 Elasticsearch 提供的一种遍历工具，用来扫描整个数据集，以获取大量

甚至全量数据。它在功能上及内部实现上，与 RDBMS 游标非常相似。但是，大多数人在第一次尝试中都没有用正确。以下是一些基本知识：

- 如果你接触到 scrolls，你可能正在读取大量数据。slicing 很可能会帮助你显著提高读取性能。
- 使用 _doc 进行排序，读取速度就会提高 20%+，而无需进行其他任何更改。（_doc 是一个伪字段）
- scrollId 调用之后会有变化。因此，请确保你始终使用最新检索的滚动 scrollId。
- 在 Reindex 的时候使用 slicing 也能提升索引数据迁移效率。

单文档检索 优先使用 GET /index/type/{id} 而非 POST /index/_search

Elasticsearch 使用不同的线程池来处理 GET /index/type/{id}和 POST /index/_search 查询。

使用 POST /index/_search 与有效载荷 {query: {"match": {"_id": "123"}}}（或类似的东西）占据搜索专用线程池。

在高负载下，这将同时降低搜索和单个文档的获取性能。

所以，单文档坚持使用：GET /index/type/{id}。

使用 size: 0 和 includes/ excludes 限定字段返回

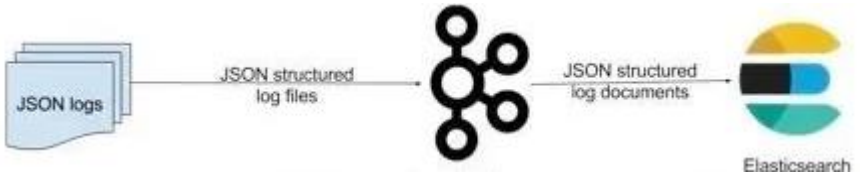
Elasticsearch 在添加 size: 0 子句前后会带来显著的性能差异 。

除非业务需要，才返回必要字段，无需返回的字段通过 includes 和 excludes 控制。

提前做好压力测试，了解系统支持的上限

分享我的个人最佳实践：

- 使用应用程序的性能基准（ performance benchmarks）测试来估计应用程序能提供支持的性能负载上限。如基于 esrally 测试。
- 避免将线程池与无限制的任务队列一起使用。队列的过度增长会对内存增加压力。
- 如果你的应用程序是借助第三方引擎中转或写入数据（例如，从 kafka 队列到 Elasticsearch 集群写入数据），请确保你的生产者对消费者的压力做出反应。也就是说，如果消费者延迟开始增加，则最好开始降低生产者的速度。



在查询中提供明确的超时

几乎所有的 Elasticsearch API 都允许用户指定超时。

找出并摆脱耗时长的操作，节省相关资源，建立稳定的服务，这将对你的应用程序和 Elasticsearch 集群都有帮助。

不要使用注入变量的 JSON 模板

永远不要这样做：

```
{
    "query": {
        "bool": {
            "filter": [
                {
                    "term": {
                        "username": {
                            "value": {{username}}
                        }
                    }
                },
                {
                    "term": {
                        "password": {
                            "password": {{password}}
                        }
                    }
                },
            ]
        }
    }
}
```

防止 SQL 注入，只要有人通过恶意 username 和 password 输入，将暴露你的整个数据集，这只是时间问题。

我建议使用两种安全的方法来生成动态查询：

- 使用 Elasticsearch 官方客户端提供的查询模型。（这在 Java 上效果很好）。
- 使用 JSON 库（例如 Jackson）构建 JSON 树并将其序列化为 JSON。

实战技巧（Strategy）

在最后一节中，我收集了解决上述未解决问题的便捷的实战技巧。

始终（尝试）坚持使用最新的 JVM 和 Elasticsearch 版本

Elasticsearch 是一个 Java 应用程序。像其他所有 Java 应用程序一样，它也有 hot paths 和垃圾回收问题。几乎每个新的 JVM 版本都会带来很多优化，你可以不费吹灰之力利用这些优化。

Elasticsearch 有一个官方页面，列出了支持的 JVM 版本和垃圾收集器。在尝试任何 JVM 升级之前，请务必先翻一翻如下文章清单：

- https://www.elastic.co/guide/en/elasticsearch/guide/current/_don_8217_t_touch_these_settings.html
- https://www.elastic.co/cn/support/matrix#matrix_jvm

注意：Elasticsearch 升级也是免费获得性能提升的来源。

使用 Elasticsearch 完整和部分快照进行备份

Elasticsearch 可以便捷的实现全部索引的全量快照，或者部分索引数据的增量快照。

根据你的更新模式和索引大小，找到适合你的用例的快照最佳组合。

也就是说，例如，在 00:00 时有 1 个完整快照，在 06:00、12:00 和 18:00 时有 3 个 局部增量快照。将它们存储在第三方存储也是一种好习惯。

有一些第三方 插件 可以简化这些情况。

举例：https://www.elastic.co/guide/en/elasticsearch/plugins/master/repository.html

与每份备份方案一样，安全起见，请确保快照可以还原并反复练习几次。

有一个持续的性能测试平台

像任何其他数据库一样，Elasticsearch 在不同条件下显示不同的性能：

- 索引，文档大小；

- 更新，查询/检索模式；
- 索引，集群设置；
- 硬件，操作系统，JVM 版本等。

很难跟踪每个设置的改变以观察其对整体性能的影响。确保你（至少）进行每日性能测试，以帮助缩小范围，快速定位最近引入的、导致性能下降的可能的原因。

这种性能测试说起来容易做起来难。你需要确保测试环境：

- 能有代表性的生产环境数据
- 配置和生产环境一致
- 完全覆盖用例
- 考虑包括操作系统缓存的测试的影响。

使用别名

告诉你一些颇有见地的实操经验：永远不要查询索引，而要查询 别名。

别名是指向实际索引的指针。你可以将一个或多个索引归为一个别名。

许多 Elasticsearch 索引在索引名称上都有内部上下文，例如 events-20190515 代表 20190515 这一天的数据。

现在，在查询 events-* 索引时，应用程序代码中有两个选择：

选择 1：通过特定日期格式即时确定索引名称：events-YYYYMMDD。

这种方法有两个主要缺点：

- 需要回退到特定日期的索引，因此需要对整个代码库进行相应的设计以支持这种操作。
- 撇开所有时钟同步问题，在凌晨，你需要用程序或者脚本控制索引切换，确保数据写入下一天索引。

选择 2：创建一个 events 别名，指向 events-* 相关的索引。负责创建新索引的组件如： curator 或者 ILM（索引生命周期管理）可以自动将别名切换到新索引。

这种方法将带来两个明显的好处：

- 它没有以前方法的缺点。
- 只需指向 events 别名，代码就会更简洁。

避免拥有大量同义词

Elasticsearch 支持索引阶段和查询阶段指定 同义词。

没有同义词，搜索引擎是不完整的，但实战使用环境，注意如下问题：

- 索引阶段同义词增加了索引大小，并增加了运行时开销。

- 查询阶段同义词不会增加索引的大小，但顾名思义，这会增加运行时开销。
- 使用同义词，很容易在尝试修复其他问题时无意间破坏某些其他内容。

所以，要持续监视同义词对性能的影响，并尝试为添加的每个同义词编写测试用例。

同义词官方文档：https://www.elastic.co/guide/en/elasticsearch/reference/current/analysis-synonym-tokenfilter.html

在启用副本之前强制段合并及增加带宽

一个非常常见的 Elasticsearch 用例是：定期（每两小时一次）创建一个索引。

关于如何实现最佳性能，SoundCloud 上有一篇非常不错的 文章。从该文中引用，我特别发现以下几项“必须”。

- 在完成索引创建后，务必启用副本。
- 在启用副本之前，请确保：

通过强制合并来缩小索引大小:

```
POST /twitter/_forcemerge
```

推荐阅读：https://developers.soundcloud.com/blog/how-to-reindex-1-billion-documents-in-1-hour-at-soundcloud

记录应用程序级别指标

Kibana 对 Elasticsearch 性能提供了多维监控指标仪表盘：

- indexing,
- search latency and throughput,
- flush
- merge operations
- GC pauses
- heap size
- OS (CPU usage, disk I/O
- kernel caches 等.....

但，这还不够。如果由多个应用程序使用，Elasticsearch 将受到各种访问模式的影响。

想象一下，你的应用程序 A 试图删除 1000 万 个不太重要的用户文档，而另一个组

件 B 试图更新用户帐户详细信息。

如果你查看 Elasticsearch 监控指标，一切都是绿色正常。

但是，此时更新账户的用户可能不满意他们尝试更新帐户时的延迟。

因此，始终为你的 Elasticsearch 查询提供额外的应用程序级指标。

尽管 Elasticsearch 结合 Kibana 或者 cerbro 已经为整体集群性能提供了足够的指标，但它们缺乏特定于操作的上下文监控，需要结合实际业务特事特办。

重视 CPU 的配置选型和使用率监控

怎么强调 CPU 都不过分。

从我过去的经验来看，无论是写负载还是读负载场景，CPU 一直是我们的瓶颈。

谨慎编写自定义的 Elasticsearch 插件

- 许多 Elasticsearch 版本包含重大的内部更改。你的插件所基于的公共 API 很可能会向后不兼容。
- 你需要调整部署过程，不能再使用原始的 Elasticsearch 工作。
- 由于你的应用程序依赖于于插件提供的特定功能，因此在集成测试过程中运行的 Elasticsearch 实例也需要包含插件。你也就不能再使用原始的 Docker 镜像。

五、致谢

本书源自阿里云和 Elastic 联合主办的大规模协作活动——Elasticsearch 百人大作战，历时近 2 个月成型发布。在创作过程中，我们深切感受到 Elasticsearch 技术爱好者们的热情和利他精神。期间，我们收到过创作人在忙碌之后凌晨时分，提交发送的文章信息，目睹过跨越数天的内容沟通讨论，以及创作人在临近发布时，对文章一遍遍精益求精的修改优化。

这本书承载着数十位技术圈开发者的共同努力和坚持。对每一位创作人，我们都有一万分的感谢。茫茫技术圈，有幸同行，合作完成一件之前没有多少人尝试过的事情；技术学习创作之路，道阻且长，愿与所有创作人共勉。

成书不易，特此感谢以下每一位涉及本书创作的参与者。

主编

刘晓国（Elastic）

专家团（排名不分先后）

曾勇（Elastic）

刘征（Elastic）

朱杰（Elastic）

李捷（Elastic）

郭瑞杰（阿里巴巴）

李京梅（阿里巴巴）

高雪峰（阿里巴巴）

郭雪梅（阿里巴巴）

出品人（排名不分先后）

刘帅（企查查）

李猛（力萌信息）

田雪松（Elastic 中文社区）

杨振涛（vivo）

周海清（观想科技）

朱永生（闪马智能）

李捷（Elastic）

欧阳楚才（阿里巴巴）

吴斌（Elastic 中文社区）

张超（Elastic 中文社区）

朱荣鑫（源图信息）

曾红（妙想技术）

创作人（排名不分先后）

陈晨

高冬冬

亢伟楠

骆潇龙

齐乐

吴斌

张超

赵震一

曾勇

程序员历小冰

郭海亮

李捷

毛夏君

田雪松

杨丛聿

张刘毅

朱永生

冯江涛

胡征南

李增胜

铭毅天下

王欢

杨景江

张妙成

朱祝元

冯钰妍

金端

刘晓国

欧阳楚才

王涛

杨松柏

赵凯

左卫东

项目策划组（排名不分先后）

是溪	葛丽丽	王佳玲	潘禹丞	洪阳
辰悠	莫孤			

鸣谢（排名不分先后）

染天	万喜	城破	濒湖	昭坤
杨青	王媛	邱国峰	汪兴	

扫码申请成为

Elasticsearch 百人大作战联合创作人

扫码追踪书籍更新动态

认识优秀创作人

扫码加入 Elasticsearch 技术社区

参与技术交流

阿里云开发者“藏经阁”

海量电子书免费下载