

云计算与虚拟化技术丛书

深入理解 ElasticSearch

Mastering ElasticSearch

[美]拉斐尔·酷奇 (Rafał Kuć) 马雷克·罗戈任斯基 (Marek Rogoziński)著

张世武 余洪森 商旦 译



机械工业出版社
China Machine Press

图书在版编目 (CIP) 数据

深入理解 ElasticSearch/ (美) 酷奇 (Kuć, R.), (美) 罗戈任斯基 (Rogoński, M.) 著;
张世武, 余洪森, 商旦译. —北京: 机械工业出版社, 2016.1
(云计算与虚拟化技术丛书)
书名原文: Mastering ElasticSearch

ISBN 978-7-111-52416-8

I. 深… II. ①酷… ②罗… ③张… ④余… ⑤商… III. 互联网络—情报检索
IV. ①G354.4 ②TP391.3

中国版本图书馆 CIP 数据核字 (2015) 第 309541 号

本书版权登记号: 图字: 01-2014-2032

Rafał Kuć and Marek Rogoński: *Mastering ElasticSearch* (ISBN: 978-1-78328-143-5)

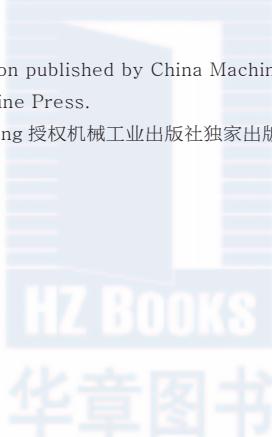
Copyright © 2013 Packt Publishing. First published in the English language under the title "Mastering ElasticSearch".

All rights reserved.

Chinese simplified language edition published by China Machine Press.

Copyright © 2016 by China Machine Press.

本书中文简体字版由 Packt Publishing 授权机械工业出版社独家出版。未经出版者书面许可, 不得以任何方式复制或抄袭本书内容。



深入理解 ElasticSearch

出版发行: 机械工业出版社 (北京市西城区百万庄大街 22 号 邮政编码: 100037)

责任编辑: 秦 健

责任校对: 董纪丽

印 刷:

版 次: 2016 年 1 月第 1 版第 1 次印刷

开 本: 186mm×240mm 1/16

印 张: 16.75

书 号: ISBN 978-7-111-52416-8

定 价: 69.00 元

凡购本书, 如有缺页、倒页、脱页, 由本社发行部调换

客服热线: (010) 88379426 88361066

投稿热线: (010) 88379604

购书热线: (010) 68326294 88379649 68995259

读者信箱: hzit@hzbook.com

版权所有 · 侵权必究

封底无防伪标均为盗版

本书法律顾问: 北京大成律师事务所 韩光 / 邹晓东

The Translator's Words 译者序

随着互联网时代的来临，人类面临着前所未有的信息过载问题。为了方便人们从海量数据中快速精准地检索感兴趣的信息，Web 搜索引擎应运而生。在互联网发展的早期，数据量比较小，单机索引就能支撑一个完整的应用。此时 Apache Lucene 凭借其精巧的代码设计、优异的性能、丰富的查询接口，以及众多的衍生搜索产品（如 Apache Solr、Nutch 等），在开源搜索领域大放异彩。随着互联网的发展，数据量快速膨胀，此时对搜索引擎提出了分布式、准实时、高容错、可扩展、易于交互等诸多要求。基于 Lucene 的简单二次开发已经满足不了日常的搜索需求，ElasticSearch 的诞生则很好地满足了上述大数据时代的搜索产品需求。

ElasticSearch 是一款基于 Apache Lucene 的开源搜索引擎产品，最早发布于 2010 年。之后 ElasticSearch 的开发团队成了专门的商业公司，持续进行开发并提供服务和技术支持。ElasticSearch 具有开源、分布式、准实时、RESTful、便于二次开发等特点，代码实现精巧，系统稳定可靠，已经被国内外众多知名组织和公司广泛采用。

本书内容丰富，不仅深入介绍了 Apache Lucene 的评分机制、查询 DSL、底层索引控制，而且介绍了 ElasticSearch 的分布式索引机制、系统监控及性能优化、用户体验的改善、Java API 的使用，以及自定义插件的开发。本书文笔优雅，辅以大量翔实的实例，能帮助读者快速提高 ElasticSearch 水平。需要提醒读者的是，本书的目标读者是 ElasticSearch 的中高级用户，如果读者对 ElasticSearch 的基础概念诸如 Mapping、Types 等缺乏了解的话，可先阅读作者的另外一本针对初学者的书籍《ElasticSearch Server》。

本书的译文经过精心组织，结合了译者的 ElasticSearch 使用经验，并参考了 IBM、微软、百度、腾讯等多位业界专业人士的意见。其中，张世武负责第 1～3 章的翻译及全书的审校，余洪森负责第 4～5 章的翻译，商旦负责第 6～9 章的翻译。在本书交稿之前，翻译团队经过多次讨论、审校，力求翻译准确、优雅。由于本书涉及很多新概念，业界尚无统一术语，另外译者水平有限，难免会出现一些翻译问题。欢迎广大读者朋友及业内同行批评指正。

前　　言 *Preface*

欢迎来到 ElasticSearch 的世界。通过阅读本书，我们将带你接触与 ElasticSearch 紧密相关的各种话题。本书会从介绍 Apache Lucene 及 ElasticSearch 的基本概念开始。即使读者熟悉这些知识，简略的介绍也是很有必要的，掌握背景知识对于全面理解集群构建、索引文档、搜索这些操作背后到底发生了什么至关重要。

之后，读者将学习 Lucene 的评分过程是如何工作的，如何影响评分，以及如何让 ElasticSearch 选择不同的评分算法。本书也将介绍什么是查询重写以及进行查询重写的原因。除此之外，本书还将介绍如何修改查询来影响 ElasticSearch 的缓存功能以及如何最大限度地使用缓存。

接着你将学习索引控制的相关知识：如何通过设置不同的倒排表格式（posting format）来改变索引字段的写入模式；索引的段合并机制和段合并的重要性，以及如何调整段合并来适应应用场景；深入探讨索引分片（shard）的分配机制、路由机制，以及当数据量、查询量日渐增长时的应对策略。

当然本书也不会遗漏垃圾收集的相关内容，包括垃圾收集的工作原理、触发时间以及如何调整垃圾收集的行为。此外，本书也将涉及 ElasticSearch 状态诊断的介绍，例如，描述系统段合并状况，ElasticSearch 在高级 API 背后是如何工作以及如何限制 I/O 操作的。然而，本书并不仅限于讨论 ElasticSearch 的底层机制，同时也涵盖了如何改进用户搜索体验，例如处理拼写检查，高效地输入自动提示以及如何改进查询等内容。

除了前面介绍的那些，本书还将指导读者熟悉 ElasticSearch 的 Java API，并演示它的使用方法，其中不仅包含 CRUD（增删查改）等基本功能，同时也包含集群、索引的维护与操作等高级功能。最后，读者将通过开发一个用于数据索引的自定义 river 插件，以及一个在检索期和索引期用于数据分析的自定义分析插件来深入了解 ElasticSearch 的扩展机制。

本书主要内容

第 1 章介绍 Apache Lucene 的工作方式，以及 ElasticSearch 的基本概念，并演示 Elastic-

Search 的内部工作机制。

第 2 章描述 Lucene 评分过程是如何工作的，为什么要进行查询重写，以及查询二次评分（rescore）是如何工作的。除此之外，还将介绍 ElasticSearch 的批处理 API，以及如何使用过滤器（filter）来优化查询。

第 3 章描述如何修改 Lucene 评分，并使用不同的倒排索引格式来改变索引字段的结构。此外还会介绍 ElasticSearch 的准实时搜索和索引，事务日志的使用，理解索引的段合并以及如何调整段合并来适应应用场景。

第 4 章介绍以下技术：如何选择恰当的索引分片及复制（replicas）数量，路由是如何工作的，索引分片机制是如何工作的以及如何影响分片行为。同时还介绍 ElasticSearch 如何进行系统初始配置，以及当数据量和查询量急剧增长时如何调整系统配置。

第 5 章介绍如何为具体应用选择正确的目录（directory）实现，什么是发现（Discovery）、网关（Gateway）、恢复（Recovery）模块，如何配置这些模块，以及有哪些令人困扰的疑难点。最后介绍如何通过 ElasticSearch 来查看索引段信息，以及如何进行 ElasticSearch 缓存机制的调优。

第 6 章介绍 JVM 垃圾收集的工作原理和重要意义，以及如何对它进行调优。同时还介绍如何控制 ElasticSearch 的 I/O 操作数量，什么是预热器（warmer）以及如何使用它，最后介绍如何诊断 ElasticSearch 中的问题。

第 7 章介绍查询建议（suggester），它能帮助修正查询中的拼写错误以及构建高效的自动完成（autocomplete）机制。除此之外，将通过实际的案例展示如何使用不同查询类型和 ElasticSearch 的其他功能来提高查询相关性。

第 8 章覆盖 ElasticSearch 的 Java API，不仅包括一些基本 API，诸如连接到 ElasticSearch 集群、单条索引或批量索引、检索文档等，而且涵盖 ElasticSearch 暴露的一些用于控制集群的 API。

第 9 章通过演示如何开发你自己的河流（river）和语言处理（language）插件来介绍 ElasticSearch 的插件开发。

阅读本书的必备资源

本书基于 ElasticSearch 0.90.x 版本，所有范例代码均能在该版本下正常运行。除此之外，读者需要一个能发送 HTTP 请求的命令行工具，如 curl，该工具在绝大多数操作系统上是可用的。请记住，本书的所有范例都使用了 curl，如果读者想使用其他工具，请注意检查请求的格式从而保证所选择的工具能正确解析它。

除此之外，为了运行第 8 章和第 9 章的范例，要求已安装 JDK，并且需要一个编辑器来开发相关代码（或者类似 Eclipse 的 Java IDE）。书中这两章都使用 Apache Maven 进行代码的管理与构建。

本书的目标读者

本书的目标读者是那些虽然熟悉 ElasticSearch 基本概念但又想深入了解其本身，同时也对 Apache Lucene、JVM 垃圾收集感兴趣的 ElasticSearch 用户和发烧友。除此之外，想了解如何改进查询相关性，如何使用 ElasticSearch Java API，如何编写自定义插件的读者，也会发现本书的趣味性和实用性。

如果你是 ElasticSearch 的初学者，对查询和索引这些基本概念都不熟悉，那么你会发现本书的绝大多数章节难以理解，因为这些内容假定读者已经具备了相关背景知识。这种情况下，建议参考 Packt 出版社上一本关于 ElasticSearch 的图书《ElasticSearch Server》。

客户支持

亲爱的读者，请随时浏览 <http://www.elasticsearchserverbook.com>，这里列出了本书最新的勘误表，以及相关的扩展阅读。

范例代码下载

如果读者通过 <http://www.packtpub.com> 账号购买了 Packt 图书，可直接在本网站下载范例代码。如果你采用了其他购买方式，可登录 <http://www.packtpub.com/support> 并注册账号，我们将通过 E-mail 将代码发给你。



Acknowledgements 致 谢

Rafał Kuć 的致谢

本书正是我在完成《ElasticSearch Server》一书以后的下一个写作目标。幸运的是，我顺利实现了这个目标。我并不想逐一介绍所有主题，而是精选了一部分来阐述和分享我所了解的知识。与《ElasticSearch Server》类似，我也不会在本书中囊括所有的主题，毕竟很多小细节并不是那么重要（这依赖具体的使用案例），因此会忽略这部分内容。尽管如此，我还是希望读者能轻松获取所有 ElasticSearch、Apache Lucene 的相关知识细节，并能轻松快速地掌握感兴趣的知识。

在此，我想感谢我的家庭，我在电脑屏幕前全身心投入本书写作的那些日日夜夜里，他们表现出极大的耐心，他们是我最坚强的后盾。

同样也要感谢 Sematext 所有的同事，尤其是 Otis，感谢他为我付出时间，并让我深刻认识到 Sematext 是一个非常适合我的公司。

最后，非常诚挚地感谢所有 ElasticSearch、Lucene 项目的创建者和开发者，感谢他们杰出的工作和对开源项目的热情。没有他们，就没有本书的诞生，没有他们，开源搜索引擎就不会有现在这种活力。再次感谢！

Marek Rogoziński 的致谢

像往常一样，撰写本书是件非常艰巨的任务。这本书不仅涉及更多的高级话题，同时 ElasticSearch 的代码也在随时改进。ElasticSearch 的开发速度并不会变缓，可以毫不夸张地说，每天都会有新东西呈现。请记住，本书是前一本著作的补充和延续，因此，这意味着我们会忽略上一本著作中已经涉及的内容，并补充该书遗漏的内容。现在看看你是否会成功吧！感谢大家。

感谢 ElasticSearch、Lucene 及所有相关产品的创建者。

同时也要感谢本书的写作和出版团队。尤其要感谢帮助检查错误、校稿、消除表达歧义的伙伴们。

最后，感谢在本书写作期间给予我坚定支持的所有的朋友。



About the Authors 作者简介

Rafał Kuć 是一个很有天资的团队领袖及软件开发人员，现任 Sematext 集团公司的咨询专家及软件工程师，专注于开源技术，如 Apache Lucene、Solr、ElasticSearch 和 Hadoop stack 等，拥有超过 11 年的软件研发经验，涉及领域广阔，从银行软件到电子商务产品。他主要侧重于 Java 平台，但对能提高研发效率的任何其他工具或编程语言都抱有极高的热情。同时他也是 solr.pl 网站的创始人之一，该网站致力于帮助人们解决 Solr 和 Lucene 的相关问题。他还是世界范围内各种会议热邀的演讲嘉宾，曾受邀出席过 Lucene Eurocon、Berlin Buzzwords、ApacheCon、Lucene Revolution 等会议。

Rafał 最早于 2002 年接触 Lucene，一开始他并不喜欢这个开源产品，然而在 2003 年再次使用 Lucene 时，他改变了自己的看法，并看到了搜索技术的巨大潜力，随后 Solr 诞生了。Rafał 于 2010 年开始使用 ElasticSearch，目前主要关注 Lucene、Solr、ElasticSearch 和信息检索等方面。

Rafał 是《 Solr 3.1 Cookbook 》一书及其后续版本《 Solr 4.0 Cookbook 》的作者，同时也是 Packt Publishing 出版的所有版本的《 ElasticSearch Server 》的合著者之一。

Marek Rogoziński 是一个有着 10 多年经验的软件架构师和咨询师，专注基于开源搜索引擎（如 Solr、ElasticSearch 等）的解决方案和大数据分析技术（Hadoop、HBase、Twitter Storm 等）。

他是 solr.pl 网站的联合创始人之一，该网站致力于提供 Solr 和 Lucene 的相关资讯，同时他也是 Packt Publishing 出版的《 ElasticSearch Server 》的作者之一。

Marek Rogoziński 还是一家提供流式大数据处理和分析产品的公司的 CTO。

评审者简介 *About the Reviewers*

Ravindra Bharathi 有着 10 多年的软件工业从业经验，涉及多个领域，如教育、数字媒体营销 / 广告、企业级搜索、能源管理系统等。兴趣涉及基于搜索的应用软件，包括数据的可视化、插件定制、数据报表等。个人博客地址：<http://ravindrabharathi.blogspot.com>。

感谢我的妻子 Vidy，感谢她对我事业的所有默默付出。

Surendra Mohan 目前在一个印度知名软件咨询公司担任 Drupal 咨询师和架构师。他在加入该公司之前，曾在印度的一些跨国公司服务，并担任过各种角色，如程序员、技术 Leader、项目 Leader、项目经理、解决方案架构师、服务发布负责人等。他拥有 9 年左右的 Web 技术研发经验，涉及媒体、娱乐、房地产、旅游、出版、在线学习、企业级架构等多个领域。他是知名的演讲者，技术涉及 Drupal、开源产品、PHP、Moodle 等。同时他也是印度孟买各种 Drupal 科技会议、活动的组织者和发布者。

Surendra Mohan 也是一些书籍的评审者，这些书包括《Drupal 7 Multi Site Configuration》《Drupal Search Engine Optimization》《Building e-commerce Sites with Drupal Commerce Cookbook》。除了做技术评审之外，他还撰写了一本关于 Apache Solr 的著作。

感谢我的家人和朋友们，正是他们对我的不懈支持和鼓励，我才能保质保量完成我的图书评审工作。

Marcelo Ochoa 现任教于阿根廷布宜诺斯艾利斯省中部国立大学精确科学与自然科学院的系统实验室，也是 Scotas.com 公司的 CTO，该公司致力于提供基于 Solr 和 Oracle 的准实时搜索解决方案。他在高校任职的同时，也参与了一些与 Oracle、大数据相关的外部项目。其中 Oracle 相关项目有：Oracle 手册文档翻译、多媒体培训等。技术背景涉及数

据库、Web、Java 等。在 XML 领域，他因为参与 Apache Cocoon 中的 DB Generator，开源项目 DBPrism、DBPrism CMS，基于 Oracle JVM Directory 的 Lucene-Oracle 集成方案，Restlet.org 项目中的 Oracle XDB Restlet Adapter（一个能在基于数据库驻存的 JVM 内部生成本地 REST Web 服务的解决方案）等项目或模块的开发而为业界所熟知。

从 2006 年开始，他参与了 Oracle ACE 计划，这是 Oracle 公司官方推出的一个计划，旨在认可和奖励 Oracle 技术社区中技术娴熟并愿意分享他们的知识和经验的成员为该社区所做的贡献。



目 录 *Contents*

译者序	2.1.3 ElasticSearch 如何看评分.....	16
前言	2.2 查询改写.....	17
致谢	2.2.1 前缀查询范例	17
作者简介	2.2.2 回顾 Apache Lucene	19
评审者简介	2.2.3 查询改写的属性	20
第 1 章 ElasticSearch 简介.....	2.3 二次评分.....	21
1.1 Apache Lucene 简介.....	2.3.1 理解二次评分	21
1.1.1 熟悉 Lucene.....	2.3.2 范例数据	21
1.1.2 Lucene 的总体架构.....	2.3.3 查询	22
1.1.3 分析你的数据	2.3.4 二次评分查询的结构	22
1.1.4 Lucene 查询语言.....	2.3.5 二次评分参数配置	23
1.2 ElasticSearch 简介	2.3.6 小结	24
1.2.1 ElasticSearch 的基本概念.....	2.4 批量操作	24
1.2.2 ElasticSearch 架构背后的	2.4.1 批量取	24
关键概念	2.4.2 批量查询	26
1.2.3 ElasticSearch 的工作流程.....	2.5 排序	27
1.3 小结.....	2.5.1 基于多值字段的排序	28
第 2 章 查询 DSL 进阶.....	2.5.2 基于多值 geo 字段的排序.....	28
2.1 Apache Lucene 默认评分公式解释.....	2.5.3 基于嵌套对象的排序	30
2.1.1 何时文档被匹配上	2.6 数据更新 API	31
2.1.2 TF/IDF 评分公式	2.6.1 简单字段更新	31
	2.6.2 使用脚本按条件更新	32

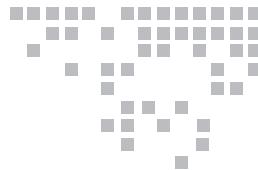
2.6.3 使用更新 API 创建或删除文档	33	3.5.2 范例的使用	65
2.7 使用过滤器优化查询	33	3.5.3 索引期更换分词器	67
2.7.1 过滤器与缓存	34	3.5.4 搜索时更换分析器	68
2.7.2 词项查找过滤器	36	3.5.5 陷阱与默认分析	68
2.8 ElasticSearch 切面机制中的过滤器与作用域	40	3.6 控制索引合并	68
2.8.1 范例数据	40	3.6.1 选择正确的合并策略	69
2.8.2 切面计算和过滤	41	3.6.2 合并策略配置	70
2.8.3 过滤器作为查询的一部分	42	3.6.3 调度	72
2.8.4 切面过滤器	44	3.7 小结	73
2.8.5 全局作用域	45		
2.9 小结	47		
第3章 底层索引控制	48	第4章 分布式索引架构	74
3.1 改变 Apache Lucene 的评分方式	48	4.1 选择合适的分片和副本数	74
3.1.1 可用的相似度模型	49	4.1.1 分片和过度分配	75
3.1.2 为每字段配置相似度模型	49	4.1.2 一个过度分配的正面例子	75
3.2 相似度模型配置	50	4.1.3 多分片与多索引	76
3.2.1 选择默认的相似度模型	51	4.1.4 副本	76
3.2.2 配置被选用的相似度模型	52	4.2 路由	76
3.3 使用编解码器	53	4.2.1 分片和数据	77
3.3.1 简单使用范例	53	4.2.2 测试路由功能	77
3.3.2 工作原理解释	54	4.2.3 索引时使用路由	80
3.3.3 可用的倒排表格式	55	4.2.4 别名	83
3.3.4 配置编解码器	56	4.2.5 多个路由值	83
3.4 准实时、提交、更新及事务日志	58	4.3 调整默认的分片分配行为	84
3.4.1 索引更新及更新提交	59	4.3.1 分片分配器简介	84
3.4.2 事务日志	60	4.3.2 even_shard 分片分配器	84
3.4.3 准实时读取	62	4.3.3 balanced 分片分配器	85
3.5 深入理解数据处理	62	4.3.4 自定义分片分配器	85
3.5.1 输入并不总是进行文本分析	62	4.3.5 裁决者	86
		4.4 调整分片分配	88
		4.4.1 部署意识	89
		4.4.2 过滤	91

4.4.3 运行时更新分配策略	92	6.2 关于 I/O 调节	136
4.4.4 确定每个节点允许的总分片数	93	6.2.1 控制 IO 节流	136
4.4.5 更多的分片分配属性	96	6.2.2 配置	136
4.5 查询执行偏好	97	6.3 用预热器提升查询速度	138
4.6 应用我们的知识	99	6.3.1 为什么使用预热器	138
4.6.1 基本假定	99	6.3.2 操作预热器	138
4.6.2 配置	100	6.3.3 测试预热器	141
4.6.3 变化来了	104	6.4 热点线程	144
4.7 小结	105	6.4.1 澄清热点线程 API 的用法 误区	145
第 5 章 管理 ElasticSearch	106	6.4.2 热点线程 API 的响应信息	145
5.1 选择正确的目录实现 – 存储模块	106	6.5 现实场景	146
5.2 发现模块的配置	109	6.5.1 越来越差的性能	146
5.2.1 Zen 发现	109	6.5.2 混杂的环境和负载不平衡	148
5.2.2 亚马逊 EC2 发现	111	6.5.3 我的服务器出故障了	149
5.2.3 本地网关	114	6.6 小结	150
5.2.4 恢复配置	115	第 7 章 改善用户搜索体验	151
5.3 索引段统计	116	7.1 改正用户拼写错误	151
5.3.1 segments API 简介	116	7.1.1 测试数据	152
5.3.2 索引段信息的可视化	118	7.1.2 深入技术细节	152
5.4 理解 ElasticSearch 缓存	119	7.1.3 completion suggester	168
5.4.1 过滤器缓存	119	7.2 改善查询相关性	172
5.4.2 字段数据缓存	121	7.2.1 数据	172
5.4.3 清除缓存	126	7.2.2 改善相关性的探索之旅	174
5.5 小结	127	7.3 小结	188
第 6 章 故障处理	129	第 8 章 ElasticSearch Java API	189
6.1 了解垃圾回收器	129	8.1 ElasticSearch Java API 简介	189
6.1.1 Java 内存	130	8.2 代码	190
6.1.2 处理垃圾回收问题	131	8.3 连接到集群	191
6.1.3 在类 UNIX 系统中避免内存 交换	135	8.3.1 成为 ElasticSearch 节点	191

8.3.2 使用传输机连接方式	192	8.7.4 Multi Search.....	212
8.3.3 选择合适的连接方式	193	8.8 Percolator	213
8.4 API 剖析	194	8.9 explain API	214
8.5 CRUD 操作	195	8.10 构造 JSON 格式的查询和文档.....	214
8.5.1 读取文档	195	8.11 管理 API	216
8.5.2 索引文档	197	8.11.1 集群管理 API	216
8.5.3 更新文档	199	8.11.2 索引管理 API	219
8.5.4 删除文档	201	8.12 小结	226
8.6 ElasticSearch 查询	203		
8.6.1 准备查询请求	203		
8.6.2 构造查询	203	9.1 建立 Apache Maven 项目结构.....	227
8.6.3 分页	206	9.1.1 了解基本知识	228
8.6.4 排序	207	9.1.2 Maven Java 项目的结构	228
8.6.5 过滤	207	9.1.3 POM 的理念.....	228
8.6.6 切面计算	208	9.1.4 运行构建过程	229
8.6.7 高亮	209	9.1.5 引入 Maven 装配插件	230
8.6.8 查询建议	209	9.2 创建一个自定义 river 插件.....	232
8.6.9 计数	210	9.2.1 实现细节	232
8.6.10 滚动	211	9.2.2 测试 river.....	238
8.7 批量执行多个操作	211	9.3 创建自定义分析插件.....	240
8.7.1 批量操作	211	9.3.1 实现细节	240
8.7.2 根据查询删除文档	212	9.3.2 测试自定义分析插件	247
8.7.3 Multi GET.....	212	9.4 小结.....	249

第 9 章 开发 ElasticSearch 插件.....227

9.1 建立 Apache Maven 项目结构.....	227
9.1.1 了解基本知识	228
9.1.2 Maven Java 项目的结构	228
9.1.3 POM 的理念.....	228
9.1.4 运行构建过程	229
9.1.5 引入 Maven 装配插件	230
9.2 创建一个自定义 river 插件.....	232
9.2.1 实现细节	232
9.2.2 测试 river.....	238
9.3 创建自定义分析插件.....	240
9.3.1 实现细节	240
9.3.2 测试自定义分析插件	247
9.4 小结.....	249



第1章

Chapter 1

ElasticSearch 简介



我们希望读者通过阅读本书能获取和拓展关于 ElasticSearch 的基本知识，并假设读者已经知道如何使用 ElasticSearch 进行单次或批量索引创建，如何发送请求检索感兴趣的文档，如何使用过滤器缩减检索返回文档的数量，以及使用切面 / 聚合 (faceting/aggregation) 机制来计算数据的一些统计量。不过，在接触 ElasticSearch 提供的各种令人激动的功能之前，仍然希望读者能对 Apache Lucene 有一个快速了解，因为 ElasticSearch 使用开源全文检索库 Lucene 进行索引和搜索，此外，我们还希望读者能了解 ElasticSearch 的一些基础概念，以及为了加快学习进程，牢记这些基础知识，当然，这并不难掌握。同时，我们也需要确保读者能按 ElasticSearch 所需要的那样正确理解 Lucene。本章主要涵盖以下内容：

- Apache Lucene 是什么。
- Lucene 的整体架构。
- 文本分析过程是如何实现的。
- Apache Lucene 的查询语言及其使用方法。
- ElasticSearch 的基本概念。
- ElasticSearch 内部是如何通信的。

1.1 Apache Lucene 简介

为了全面理解 ElasticSearch 的工作原理，尤其是索引和查询处理环节，对 Apache Lucene 的理解显得至关重要。揭开 ElasticSearch 神秘的面纱，你会发现它在内部不仅使用 Apache Lucene 创建索引，同时也使用 Apache Lucene 进行搜索。因此，在接下来的内容中，我们将展示 Apache Lucene 的基本概念，特别是针对那些从未使用过 Lucene 的读者们。

1.1.1 熟悉 Lucene

读者也许会好奇，为什么 ElasticSearch 的创始人决定使用 Apache Lucene 而不是开发自己的全文检索库。对于这个问题，笔者并不是很确定，毕竟我们不是这个项目的创始人，但我们猜想是因为 Lucene 的以下特点而得到了创始人的青睐：成熟、高性能、可扩展、轻量级以及强大的功能。Lucene 内核可以创建为独立的 Java 库文件并且不依赖第三方代码，用户可以使用它提供的各种所见即所得的全文检索功能进行索引和搜索操作。当然，Lucene 还有很多扩展，它们提供了各种各样的功能，如多语言处理、拼写检查、高亮显示等。如果不需要这些额外的特性，可以下载单个的 Lucene 内核库文件，直接在应用程序中使用。

1.1.2 Lucene 的总体架构

尽管我们可以直接探讨 Apache Lucene 架构的细节，但有些概念还是需要提前了解，以便更好地理解 Lucene 的架构，它们包括：

- 文档 (document)：索引与搜索的主要数据载体，它包含一个或多个字段，存放将要写入索引或将从索引搜索出来的数据。
- 字段 (field)：文档的一个片段，它包括两个部分：字段的名称和内容。
- 词项 (term)：搜索时的一个单位，代表文本中的某个词。
- 词条 (token)：词项在字段中的一次出现，包括词项的文本、开始和结束的位移以及类型。

Apache Lucene 将写入索引的所有信息组织成一种名为倒排索引 (inverted index) 的结构。该结构是一种将词项映射到文档的数据结构，其工作方式与传统的关系数据库不同，你大可以认为倒排索引是面向词项而不是面向文档的。接下来我们看看简单的倒排索引是什么样的。例如，我们有一些只包含 title 字段的文档，如下所示：

- ElasticSearch Server (文档 1)
- MasteringElasticSearch (文档 2)
- Apache solr 4 Cookbook (文档 3)

而索引后的结构示意图如下所示。

词项	计数	文档
4	1	<3>
Apache	1	<3>
Cookbook	1	<3>
ElasticSearch	2	<1><2>
Mastering	1	<1>
Server	1	<1>
Solr	1	<3>

正如你所见，每个词项指向该词项所出现过的文档数。这种索引组织方式支持快速有效的搜索操作，例如基于词项的查询。除了词项本身以外，每个词项还有一个与之关联的计数（即文档频率），用来告诉 Lucene 这个词项在多少个文档中出现过。

当然，实际中 Lucene 创建的索引更为复杂，也更先进，因为索引中还存储了很多其他信息，如**词向量**（为单个字段创建的小索引，存储该字段中所有的词条）、各字段的原始信息、文档删除标记等。然而，你只需知道 Lucene 索引中数据是如何组织的即可，而不用知道到底存储了哪些东西。

每个索引由多个段（segment）组成，每个段只会被创建一次但会被查询多次。索引期间，段经创建就不会再被修改。例如，文档被删除以后，删除信息被单独保存在一个文件中，而段本身并没有修改。

多个段会在一个叫作**段合并**（segments merge）的阶段被合并在一起，而且要么强制执行，要么由 Lucene 的内在机制决定在某个时刻执行，合并后段的数量更少，但是更大。段合并非常耗 I/O，且合并期间有些不再使用的信息也将被清理掉，例如，被删除的文档。对于容纳相同数据的索引，段的数量较少时，搜索速度更快。尽管如此，还是需要强调一下：因为段合并非常耗 I/O，请不要强制进行段合并，你只需要仔细配置段合并策略，剩余的事情 Lucene 会自行搞定。



如果你想知道段由哪些文件组成以及每个文件都存储了什么信息，请参考 Apache Lucene 的官方文档：http://lucene.apache.org/core/4_5_0/core/org/apache/lucene/codecs/lucene45/package-summary.html。

1.1.3 分析你的数据

读者也许会好奇，文档中的数据是如何转化为倒排索引的，而查询串又是如何转换为可用于搜索的词项的？这个转换过程称为**分析**（analysis）。

文本分析由分析器来执行，而分析器由分词器（tokenizer）、过滤器（filter）和字符映射器组成（character mapper）。

Lucene 的分词器用来将文本切割成词条，其中携带各种额外信息的词项，这些信息包括：词项在原始文本中的位置、词项的长度。分词器的工作成果称为词条流，因为这些词条被一个接一个地推送给过滤器处理。

除了分词器，过滤器也是 Lucene 分析器的组成部分，过滤器数额可选，可以为 0 个、1 个或多个，用于处理词条流中的词条。例如，它可以移除、修改词条流中的词条，甚至可以创造新的词条。Lucene 提供了很多现成的过滤器，你也可以根据需要实现新的过滤器。以下是一些过滤器的例子：

- 小写过滤器：将所有词条转化为小写。
- ASCII 过滤器：移除词条中所有非 ASCII 字符。

□ **同义词过滤器**: 根据同义词规则, 将一个词条转化为另一个词条。

□ **多语言词干还原过滤器**: 将词条的文本部分归约到它们的词根形式, 即词干还原。

当分析器中有多个过滤器时, 会逐个处理, 理论上可以有无限多个过滤器。

最后我们介绍字符映射器, 它用于调用分词器之前的文本预处理操作, 如 HTML 文本的去标签处理。

索引与查询

也许读者会好奇, Lucene 以及所有基于 Lucene 的软件, 是如何控制索引和查询操作的。在索引期, Lucene 会使用你选择的分析器来处理文档中的内容, 并可以对不同的字段使用不同的分析器, 例如, 文档的 title 字段与 description 字段就可以使用不同的分析器。

在检索时, 如果使用了某个查询分析器 (query parser), 那么查询串就会被分析。当然, 你也可以选择不进行查询分析。有一点需要牢记, Elasticsearch 中有些查询会被分析, 而有些则不会。例如, 前缀查询 (prefix query) 不会被分析, 而匹配查询 (match query) 会被分析。

你还应该记住, 索引期与检索期的文本分析要采用同样的分析器, 只有查询分析出来的词项与索引中词项能匹配上, 才会返回预期的文档集。例如, 如果在索引期使用了词干还原与小写转换, 那么在查询期也应该对查询串做相同的处理, 否则查询可能不会返回任何结果。

1.1.4 Lucene 查询语言

ElasticSearch 提供的一些查询类型 (query type) 支持 Apache Lucene 的查询解析语法, 因此, 我们应该深入了解 Lucene 的查询语言并加以描述。

理解基本概念

在 Lucene 中, 一个查询通常被分割为词项与操作符。Lucene 中的词项可以是单个词, 也可以是一个短语 (用双引号括起来的一组词)。如果设置了查询分析过程, 那么预先选定的分析器将会对查询中的所有词项进行处理。

查询中也可以包含布尔操作符, 用于连接多个词项, 使之构成从句 (clause)。有以下这些布尔操作符:

- **AND** : 它的含义是, 文档匹配当前从句当且仅当 AND 操作符左右两边的词项都在文档中出现。例如, 执行 apache AND lucene 这样的查询, 只有同时包含 apache 和 lucene 这两个词项的文档才会返回给用户。
- **OR** : 它的含义是, 包含当前从句中任意词项的文档都会被视为与该从句匹配。例如, 执行 apache OR lucene 这样的查询, 任意包含词项 apache 或词项 lucene 的文档都会返回给用户。
- **NOT** : 它的含义是, 与当前从句匹配的文档必须不包含 NOT 操作符后面的词项。

例如，执行 lucene NOT elasticsearch 这样的查询，只有包含词项 lucene 且不包含词项 elasticsearch 的文档才会返回给用户。

此外，我们还可以使用以下操作符：

□ +：它的含义是，只有包含 + 操作符后面词项的文档才会被认为是与从句匹配。例如，查找那些必须包含 lucene，但是 apache 可出现可不出现的文档，可执行查询：
+lucene apache。

□ -：它的含义是，与从句匹配的文档不能出现 - 操作符后的词项。例如，查找那些包含 lucene 但不包含 elasticsearch 的文档，可以执行查询：+lucene-elasticsearch。

如果查询中没有出现前面提到过的任意操作符，那么默认使用 OR 操作符。

另外，你还可以使用圆括号对从句进行分组，以构造更复杂的从句，例如：

```
elasticsearch AND (mastering OR book)
```

在字段中查询

就像 ElasticSearch 的处理方式那样，Lucene 中所有数据都存储在字段（field）中，而字段又是文档的组成单位。为了实现针对某个字段的查询，用户需要提供字段名称，再加上冒号以及将要在该字段中执行查询的从句。例如要查询所有在 title 字段中包含词项 elasticsearch 的文档，可执行以下查询：

```
title:elasticsearch
```

也可以在一个字段中同时使用多个从句，例如，要查找所有在 title 字段中同时包含词项 elasticsearch 和短语 mastering book 的文档，可执行如下查询：

```
title:(+elasticsearch +"mastering book")
```

当然，该查询也可以写成下面这种形式：

```
+title:elasticsearch +title:"mastering book"
```

词项修饰符

除了使用简单词项和从句的常规字段查询以外，Lucene 还允许用户使用修饰符（modifier）修改传入查询对象的词项。毫无疑问，最常见的修饰符就是通配符（wildcard）。Lucene 支持两种通配符：? 和 *。前者匹配任意一个字符，而后者匹配多个字符。



请记住，出于对性能的考虑，通配符不能作为词项的第一个字符出现。

除通配符之外，Lucene 还支持模糊（fuzzy and proximity）查询，通过使用 ~ 字符以及一个紧随其后的整数值。当使用该修饰符修饰一个词项时，意味着我们想搜索那些包含该词项近似词项的文档（所以这种查询称为模糊查询）。~ 字符后的整数值确定了近似词项与原始词项的最大编辑距离。例如，当我们执行查询：writer~2，意味着包含词项 writer 和 writers 的文档都能与查询匹配。

当修饰符 ~ 用于短语时，其后的整数值用于告诉 Lucene 词项之间可以接受的最大距离。例如，执行如下查询：

```
title:"mastering elasticsearch"
```

在 title 字段中包含 mastering elasticsearch 的文档被视为与查询匹配，而包含 mastering book elasticsearch 的文档则不匹配。如果执行这个查询：title:"mastering elasticsearch"~2，则这两个文档都被认为与查询匹配。

此外，还可以使用 ^ 字符并赋以一个浮点数对词项加权（boosting），以提高该词项的重要程度。如果都被加权，则权重值较大的词项更重要。默认情况下词项权重为 1。可以参考 2.1 节进一步了解什么是权重值，以及它在文档评分中的作用。

我们也可以使用方括号和花括号来构建范围查询。例如，要在一个数值类型的字段上执行一个范围查询，执行如下查询即可：

```
price:[10.00 TO 15.00]
```

上述查询所返回文档的 price 字段的值大于等于 10.00 并小于等于 15.00。

当然，我们也可以在字符串类型的字段上执行范围查询，例如：

```
name:[Adam TO Adria]
```

上面查询所返回文档的 name 字段包含按字典顺序介于 Adam 和 Adria 之间（包括 Adam 和 Adria）的词项。

如果想执行范围查询同时又想排除边界值，则可使用花括号作为修饰符。例如，查找 price 字段值大于等于 10.00 但小于 15.00 的文档，可使用如下查询：

```
price:[10.00 TO 15.00}
```

特殊字符处理

很多应用场景中，需要搜索某个特殊字符（这些特殊字符包括 +、-、&&、||、!、(,)、{}、[], ^、"、~、*、?、:、\、/），这时先使用反斜杠对这些特殊字符进行转义。例如，搜索 abc"efg 这个词项，需要按如下方式处理：

```
abc\"efg
```

1.2 ElasticSearch 简介

虽然读者可能已经对 ElasticSearch 有所了解，至少已经了解了它的一些核心概念和基本用法。然而，为了全面理解该搜索引擎是如何工作的，我们最好简略地讨论一下它。

ElasticSearch 是一个可用于构建搜索应用的成品软件^Θ。它最早由 Shay Banon 创建并

^Θ 区别于 Lucene 这种中间件。——译者注

于2010年2月发布。之后的几年ElasticSearch迅速流行开来，成为商业解决方案之外且开源的一个重要选择，也是下载量最多的开源软件之一，每月下载量超过20万次。

1.2.1 ElasticSearch的基本概念

现在，让我们了解一下ElasticSearch的基本概念及其特征。

索引

ElasticSearch将它的数据存储在一个或多个索引(index)中。用SQL领域的术语来类比，索引就像数据库，可以向索引写入文档或者从索引中读取文档，并通过在ElasticSearch内部使用Lucene将数据写入索引或从索引中检索数据。需要注意的是，ElasticSearch中的索引可能由一个或多个Lucene索引构成，具体细节由ElasticSearch的索引分片(shard)、复制(replica)机制及其配置决定。

文档

文档(document)是ElasticSearch世界中的主要实体(对Lucene来说也是如此)。对所有使用ElasticSearch的案例来说，它们最终都可以归结为对文档的搜索。文档由字段构成，每个字段有它的字段名以及一个或多个字段值(在这种情况下，该字段被称为是多值的，即文档中有多个同名字段)。文档之间可能有各自不同的字段集合，且文档并没有固定的模式或强制的结构。另外，这些规则也适用于Lucene文档。事实上，ElasticSearch的文档最后都存储为Lucene文档了。从客户端的角度来看，文档是一个JSON对象(想了解更多关于JSON格式细节，请参考<http://en.wikipedia.org/wiki/JSON>)。

映射

正如你在1.1节所了解的那样，所有文档在写入索引前都需要先进行分析。用户可以设置一些参数，来决定如何将输入文本分割为词条，哪些词条应该被过滤掉，或哪些附加处理是有必要被调用的(如移除HTML标签)。此外，ElasticSearch也提供了各种特性，如排序时所需的字段内容信息。这就是映射(mapping)扮演的角色，存储所有这种元信息。虽然ElasticSearch能根据字段值自动检测字段的类型，但有时候(事实上，几乎是所有时候)用户还是想自行配置映射，以避免出现一些令人不愉快的意外。

类型

ElasticSearch中每个文档都有与之对应的类型(type)定义。这允许用户在一个索引中存储多种文档类型，并为不同文档类型提供不同的映射。

节点

单个的ElasticSearch服务实例称为节点(node)。很多时候部署一个ElasticSearch节点就足以应付大多数简单的应用，但是考虑到容错性或在数据膨胀到单机无法应付这些状况时，你也许会更倾向于使用多节点的ElasticSearch集群。

集群

当数据量或查询压力超过单机负载时，需要多个节点来协同处理，所有这些节点组成的系统称为集群（cluster）。集群同时也是无间断提供服务的一种解决方案，即便在某些节点因为宕机或执行管理任务（如升级）不可用时。ElasticSearch 几乎无缝集成了集群功能。在我们看来，这是它胜过竞争对手的最主要的优点之一。而且，在 ElasticSearch 中配置一个集群是再容易不过的事了。

分片

正如我们之前提到的那样，集群允许系统存储的数据总量超过单机容量。为了满足这个需求，ElasticSearch 将数据散布到多个物理 Lucene 索引上。这些 Lucene 索引称为分片（shard），而散布这些分片的过程叫作分片处理（sharding）。ElasticSearch 会自动完成分片处理，并且让这些分片呈现出一个大索引的样子。请记住，除了 ElasticSearch 本身自动进行分片处理外，用户为具体的应用进行参数调优也是至关重要的，因为分片的数量在索引创建时就已经配置好，而且之后无法改变，至少对目前的版本是这样的。

副本

分片处理允许用户向 ElasticSearch 集群推送超过单机容量的数据。副本（replica）则解决了访问压力过大时单机无法处理所有请求的问题。思路很简单，即为每个分片创建冗余的副本，处理查询时可以把这些副本用作最初的主分片（primary shard）。请记住，我们并未付出额外的代价。即使某个分片所在的节点宕机，ElasticSearch 也可以使用其副本，从而不会造成数据丢失，而且支持在任意时间点添加或移除副本，所以一旦有需要可随时调整副本的数量。

网关

在 ElasticSearch 的工作过程中，关于集群状态，索引设置的各种信息都会被收集起来，并在网关（gateway）中被持久化。

1.2.2 ElasticSearch 架构背后的关键概念

ElasticSearch 架构遵循了一些设计理念。通常开发团队希望这个搜索引擎产品易于使用和扩展，并能在 ElasticSearch 的各个地方体现出来。从架构的角度出发，ElasticSearch 具有下面这些主要特征：

- 合理的默认配置，使得用户在简单安装以后能直接使用 ElasticSearch 而不需要任何额外的调试，这包括内置的发现（如字段类型检测）和自动配置功能。
- 默认的分布式工作模式。每个节点总是假定自己是某个集群的一部分或将是在某个集群的一部分，一旦工作启动节点便会加入某个集群。
- 对等架构（P2P）可以避免单点故障（SPOF）。节点会自动连接到集群中的其他节点，进行相互的数据交换和监控操作。这其中就包括索引分片的自动复制。

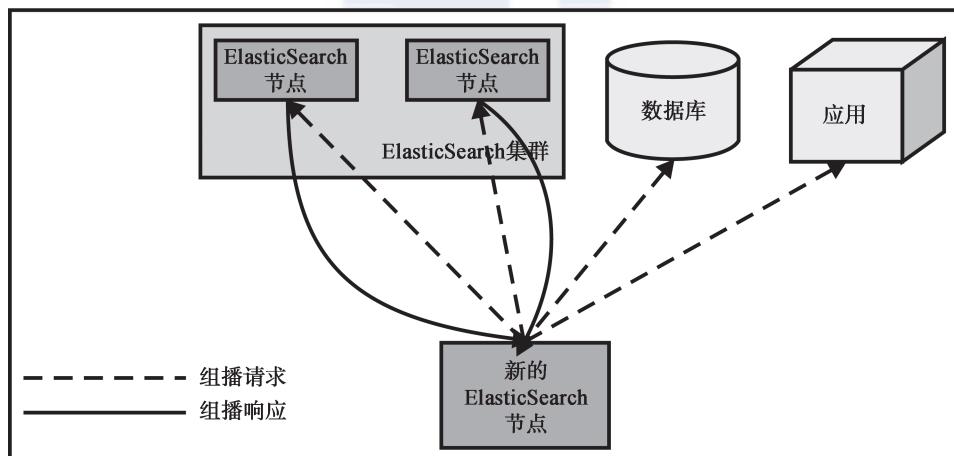
- 易于向集群扩充新节点，不论是从数据容量的角度还是数量角度。
- ElasticSearch 没有对索引中的数据结构强加任何限制，从而允许用户调整现有的数据模型。正如之前描述的那样，ElasticSearch 支持在一个索引中存在多种数据类型，并允许用户调整业务模型，包括处理文档之间的关联（尽管这种功能非常有限）。
- 准实时（Near Real Time, NRT）搜索和版本同步（versioning）。考虑到 ElasticSearch 的分布式特性，查询延迟和节点之间临时的数据不同步是难以避免的。ElasticSearch 尝试消除这些问题并且提供额外的机制用于版本同步。

1.2.3 ElasticSearch 的工作流程

现在，让我们简单地讨论一下 ElasticSearch 是如何工作的。

启动过程

当 ElasticSearch 节点启动时，它使用广播技术（也可配置为单播）来发现同一个集群中的其他节点（这里的关键是配置文件中的集群名称）并与它们连接。读者可以通过下图的描述来了解相关的处理过程：



集群中会有一个节点被选为管理节点（master node）。该节点负责集群的状态管理以及在集群拓扑变化时做出反应，分发索引分片至集群的相应节点上。



请记住，从用户的角度来看，ElasticSearch 中的管理节点并不比其他节点重要，这与其他某些分布式系统不同（如数据库）。实际上，你不需要知道哪个节点是管理节点，所有操作可以发送至任意节点，ElasticSearch 内部会自行处理这些不可思议的事情。如果有需要，任意节点可以并行发送子查询给其他节点，并合并搜索结果，然后返回给用户。所有这些操作并不需要经过管理节点处理（请记住，ElasticSearch 是基于对等架构的）。

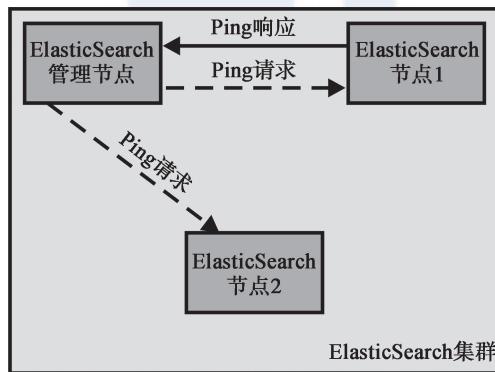
管理节点读取集群的状态信息，并在必要时进行恢复处理。在该阶段，管理节点会检查所有索引分片并决定哪些分片将用于主分片。然后，整个集群进入黄色状态。

这意味着集群可以执行查询，但是系统的吞吐量以及各种可能的状况是未知的（这种状况可以简单理解为所有的主分片已经分配出去了，而副本没有），因而接下来就是要寻找到冗余的分片并用作副本。如果某个主分片的副本数过少，管理节点将决定基于某个主分片创建分片和副本。如果一切顺利，集群将进入绿色状态（这意味着所有主分片和副本均已分配好）。

故障检测

集群正常工作时，管理节点会监控所有可用节点，检查它们是否正在工作。如果任何节点在预定义的超时时间内没有响应，则认为该节点已经断开，然后开始启动错误处理过程。这意味着要在集群 – 分片之间重新做平衡，因为之前已断开节点上的那些分片不可用了，剩下的节点要肩负起相应的责任。换句话说，对每个丢失的主分片，一个新的主分片将会从原来的主分片的副本中脱颖而出。新分片和副本的放置策略是可配置的，用户可以根据具体需求进行配置。更多信息请参见第 4 章（索引分布架构）的内容。

为了描述故障检测（failure detection）是如何工作的，我们用一个只有三个节点的集群为例，即包含一个管理节点，两个数据节点。管理节点会发送 ping 请求至其他节点，然后等待响应。如果没有响应，则该节点会从集群中移除。如下图所示：



与 Elasticsearch 通信

前面已经讨论过 Elasticsearch 是如何构建的了，然而，对普通用户来说，最重要的还是如何向 Elasticsearch 推送数据并构建查询。为了提供这些功能，Elasticsearch 对外公开了一个设计精巧的 API。这个 API 是基于 REST（REST 细节请参考：http://en.wikipedia.org/wiki/Representational_state_transfer）的，并在实践中能轻松整合到任何支持 HTTP 协议的系统中去。

Elasticsearch 假设数据由 URL 携带或者以 JSON（JSON 细节请参考 <http://en.wikipedia.org/wiki/JSON>）。文档的形式由 HTTP 消息体携带。使用 Java 或基于 JVM 语言的用户，

应该了解一下 Java API，它除了 REST API 提供的所有功能以外还有内置的集群发现功能。

值得一提的是，ElasticSearch 在内部也使用 Java API 进行节点间通信。读者可以在第 8 章中了解更多 Java API 的细节，而这里只是简略地了解 Java API 提供了哪些功能。本书假设读者已经使用过这些功能了，只是在此做一点小小的提示。如果用户还没有使用过，强烈建议阅读相关材料，其中《ElasticSearch Server》一书覆盖了所有这些内容。

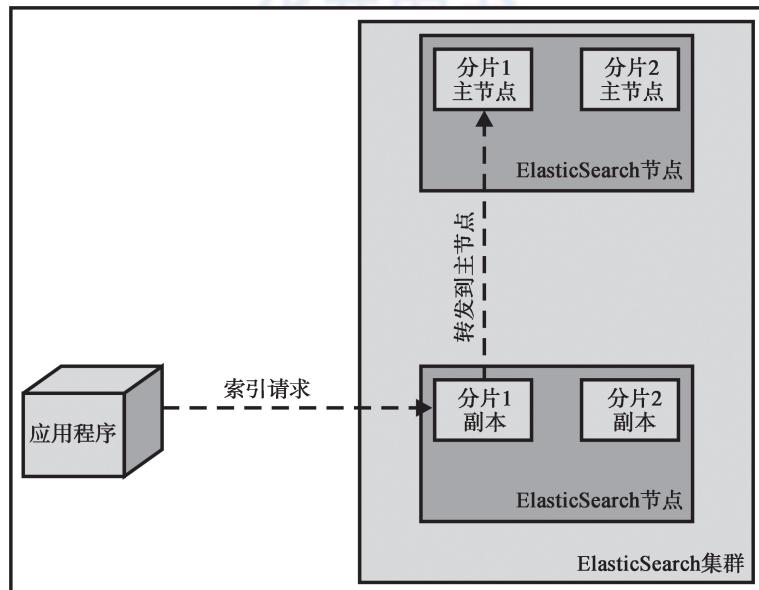
索引数据

ElasticSearch 提供了四种方式来创建索引。最简单的方式是使用索引 API，它允许用户发送一个文档至特定的索引。例如，使用 curl 工具（详见 <http://curl.haxx.se/>），并用如下命令创建一个文档：

```
curl -XPUT http://localhost:9200/blog/article/1 -d '{"title": "New
version of Elastic Search released!", "content": "...", "tags":
["announce", "elasticsearch", "release"] }'
```

第二种或第三种方式允许用户通过 bulk API 或 UDP bulk API 来一次性发送多个文档至集群。两者的区别在于网络连接方式，前者使用 HTTP 协议，后者使用 UDP 协议，且后者速度快，但是不可靠。第四种方式使用插件发送数据，称为河流（river），河流运行在 ElasticSearch 节点上，能够从外部系统获取数据。

有一件事情需要记住，建索引操作只会发生在主分片上，而不是副本上。当把一个索引请求发送至某节点时，如果该节点没有对应的主分片或者只有副本，那么这个请求会被转发到拥有正确的主分片的节点（如下图所示）。

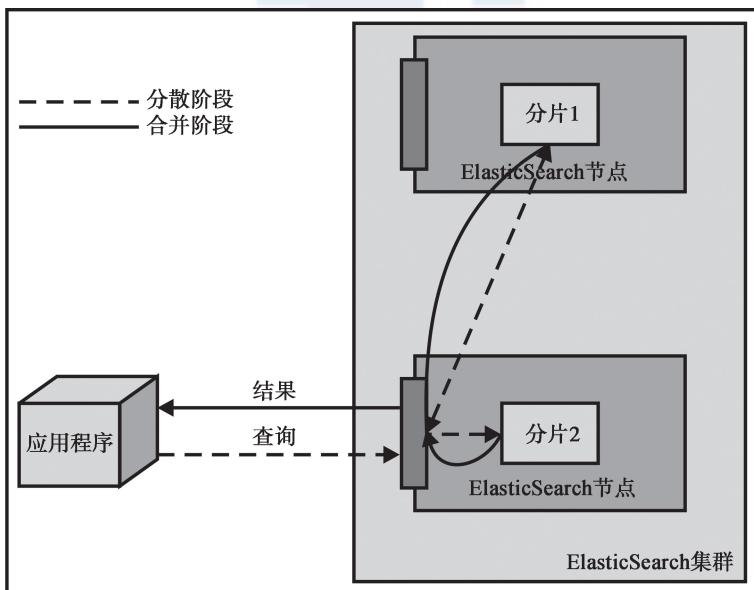


查询数据

查询 API 占据了 ElasticSearch API 的大部分内容。使用查询 DSL (基于 JSON 的可用于构建复杂查询的语言)，我们可以做下面这些事情：

- ❑ 使用各种查询类型，包括：简单的词项查询、短语查询、范围查询、布尔查询、模糊查询、区间查询、通配符查询、空间查询等。
- ❑ 组合简单查询构建复杂查询。
- ❑ 文档过滤，在不影响评分的前提下抛弃那些不满足特定查询条件的文档。
- ❑ 查找与特定文档相似的文档。
- ❑ 查找特定短语的查询建议和拼写检查。
- ❑ 使用切面构建动态导航和计算各种统计量。
- ❑ 使用预搜索 (prospective search) 并查找与指定文档匹配的 query 集合。

对于查询操作，读者应该要重点了解：查询并不是一个简单的、单步骤的操作。一般来说，查询分为两个阶段：分散阶段 (scatter phase) 和合并阶段 (gather phase)。分散阶段将 query 分发到包含相关文档的多个分片中去执行查询，合并阶段则从众多分片中收集返回结果，然后对它们进行合并、排序、后续处理，然后返回给客户端。该机制可以由下图描述。



 ElasticSearch 对外提供了 6 个系统参数，任何一个都可以用来定制分散 / 合并机制。关于这个问题可参阅本书的上一版《ElasticSearch Server》(Packt 出版社)。

索引配置

前面已经讨论过 ElasticSearch 的自动索引配置以及发现识别文档字段类型和结构的功能。当然，ElasticSearch 也提供了一些功能使得用户能手动配置，例如用户想通过映射来配置自定义的文档结构，或者想设置索引的分片和副本数，抑或定制文本分析过程，种种这些需求都可以通过手动配置解决。

系统管理和监控

ElasticSearch 中系统管理和监控相关的 API 允许用户改变集群的设置，如调节集群发现机制和索引放置策略等。此外，你还可得到关于集群状态信息或每个节点、每个索引的统计信息。集群监控 API 非常全面，我们将在第 5 章学习相关 API 的使用范例。

1.3 小结

在本章中，我们了解了 Apache Lucene 的一般架构，例如它的工作原理，文本分析过程是如何完成的，如何使用 Apache Lucene 查询语言。此外，我们还讨论了 ElasticSearch 的一些基本概念，例如它的基本架构和内部通讯机制。

在下一章，我们将学习 Apache Lucene 的默认评分公式，什么是查询重写过程（query rewrite process）以及它是如何工作的。除此之外，还将讨论 ElasticSearch 的一些功能，例如查询的二次评分（query rescore）、准实时批量获取（multi near real-time get）、批量搜索操作（bulk search operations）。接着将学习到如何使用 update API 来部分地改变文档，如何对数据进行排序，如何使用过滤功能（filtering）来改进查询的性能。最后，我们将了解如何在切面机制中使用过滤器（filters）和作用域（scope）。

查询 DSL 进阶

上一章我们了解了什么是 Apache Lucene，它的整体架构，以及文本分析过程是如何完成的。之后，我们介绍了 Lucene 的查询语言及其用法。除此之外，我们还讨论了 ElasticSearch 及其架构和一些核心概念。在本章，我们将深入研究 ElasticSearch 的查询 DSL (Domain Specific Language)。然而，在了解那些高级查询之前，我们先来了解 Lucene 评分公式的工作原理。本章将涵盖以下内容：

- ❑ Lucene 默认评分公式是如何工作的。
- ❑ 什么是查询重写。
- ❑ 查询二次评分是如何工作的。
- ❑ 如何在单次请求中实现批量准实时读取操作。
- ❑ 如何在单次请求中发送多个查询。
- ❑ 如何对包括嵌套文档和多值字段的数据排序。
- ❑ 如何更新已索引的文档。
- ❑ 如何通过使用过滤器来优化查询。
- ❑ 如何在 ElasticSearch 的切面计算机制中使用过滤器和作用域。

2.1 Apache Lucene 默认评分公式解释

对于查询相关性，重点是去理解文档对查询的得分是如何计算出来的。什么是文档得分？它是一个刻画文档与查询匹配程度的参数。本节我们就将了解 Apache Lucene 的默认评分机制：TF/IDF（词频 / 逆文档频率）算法以及它是如何影响文档召回的。了解评分公式的工作原理对构造复杂查询以及分析查询中因子的重要性都是很有价值的。

2.1.1 何时文档被匹配上

当一个文档经 Lucene 返回，则意味着该文档与用户提交的查询是匹配的。在这种情况下，每个返回的文档都有一个得分。得分越高，文档相关度更高，至少从 Apache Lucene 及其评分公式的角度来看是这样的。显而易见，同一个文档针对不同查询的得分是不同的，而且比较某文档在不同查询中的得分是没有意义的。读者需要注意，同一文档在不同查询中的得分不具备可比较性，不同查询返回文档中的最高得分也不具备可比较性。这是因为文档得分依赖多个因子，除了权重和查询本身的结构，还包括匹配的词项数目，词项所在字段，以及用于查询规范化的匹配类型等。在一些比较极端的情况下，同一个文档在相似查询中的得分非常悬殊，仅仅是因为使用了自定义得分查询或者命中词项数发生了急剧变化。

现在，让我们再回到评分过程。为了计算文档得分，需要考虑以下这些因子：

- **文档权重 (document boost)**: 索引期赋予某个文档的权重值。
- **字段权重 (field boost)**: 查询期赋予某个字段的权重值。
- **协调因子 (coord)**: 基于文档中词项命中个数的协调因子，一个文档命中了查询中的词项越多，得分越高。
- **逆文档频率 (inverse document frequency)**: 一个基于词项的因子，用来告诉评分公式该词项有多么罕见。逆文档频率越低，词项越罕见。评分公式利用该因子为包含罕见词项的文档加权。
- **长度范数 (length norm)**: 每个字段的基于词项个数的归一化因子（在索引期计算出来并存储在索引中）。一个字段包含的词项数越多，该因子的权重越低，这意味着 Apache Lucene 评分公式更“喜欢”包含更少词项的字段。
- **词频 (term frequency)**: 一个基于词项的因子，用来表示一个词项在某个文档中出现了多少次。词频越高，文档得分越高。
- **查询范数 (query norm)**: 一个基于查询的归一化因子，它等于查询中词项的权重平方和。查询范数使不同查询的得分能相互比较，尽管这种比较通常是困难且不可行的。

2.1.2 TF/IDF 评分公式

现在我们来看评分公式。请记住，为了调节查询相关性，你并不需要深入理解这个公式的来龙去脉，但是了解它的工作原理是非常重要的。

Lucene 理论评分公式

TF/IDF 公式的理论形式如下：

$$score(q, d) = coord(q, d) * queryBoost(q) * \frac{V(q) * V(d)}{|V(q)|} * lengthNorm(d) * docBoost(d)$$

上面的公式糅合了布尔检索模型和向量空间检索模型。现在，我们并不讨论理论评分公式，而是直接跳到 Lucene 实际评分公式，因为在 Lucene 内部就是这么实现并使用的。



关于布尔检索模型和向量空间检索模型的知识远远超出了本书的讨论范围，想了解更多相关知识，请参考 http://en.wikipedia.org/wiki/Standard_Boolean_model 和 http://en.wikipedia.org/wiki/Vector_Space_Model。

Lucene 实际评分公式

现在让我们看看 Lucene 实际使用的评分公式：

$$score(q, d) = coord(q, d) * queryNorm(q) * \sum_{t \in q} (tf(t \text{ in } d) * idf(t)^2 * boost(t) * norm(t, d))$$

也许你已经看到了，得分公式是一个关于查询 q 和文档 d 的函数，有两个因子 $coord$ 和 $queryNorm$ 并不直接依赖查询词项，而是与查询词项的一个求和公式相乘。

求和公式中每个加数由以下因子连乘所得：词频、逆文档频率、词项权重、范数。范数就是之前提到的长度范数。

这个公式听起来很复杂。请别担心，你并不用记住所有的细节，而只需意识到哪些因素是跟评分有关即可。从前面的公式我们可以导出一些基本规则：

- 越多罕见的词项被匹配上，文档得分越高。
- 文档字段越短（包含更少的词项），文档得分越高。
- 权重越高（不论是索引期还是查询期赋予的权重值），文档得分越高。

正如你所见，Lucene 将最高得分赋予同时满足以下条件的文档：包含多个罕见词项，词项所在字段较短（该字段索引了较少的词项）。该公式更“喜欢”包含罕见词项的文档。



如果你想了解更多关于 Apache Lucene TF/IDF 评分公式的信息，请参考 Apache lucene 中 TFIDFSimilarity 类的文档：http://lucene.apache.org/core/4_5_0/core/org_apache_lucene_search_similarities_TFIDFSimilarity.html。

2.1.3 ElasticSearch 如何看评分

总而言之，是 ElasticSearch 使用了 Lucene 的评分功能，但好在我们可以替换默认的评分算法（更多细节请参考 3.1 节）。还有一点请记住，ElasticSearch 使用了 Lucene 的评分功能但不仅限于 Lucene 的评分功能。用户可以使用各种不同的查询类型以精确控制文档评分的计算（如 `custom_boost_factor` 查询、`constant_score` 查询、`custom_score` 查询等），还可以通过使用脚本（scripting）来改变文档得分，还可以使用 ElasticSearch 0.90 中出现的二次评分功能，通过在返回文档集之上执行另外一个查询，重新计算前 N 个文档的文档得分。



想了解更多 Apache Lucene 查询类型，请参考 http://lucene.apache.org/core/4_5_0/queries/org_apache_lucene_queries_package-summary.html 上的相关文档。

2.2 查询改写

如果你之前使用过诸如前缀查询或通配符查询之类的查询类型，那么你会发现这些都是基于多词项的查询，且都涉及查询改写。ElasticSearch（实际上是 Lucene 执行该操作）使用查询改写是出于对性能的考虑。从 Lucene 的角度来看，所谓的查询改写操作，就是把费时的原始查询类型实例改写成一个性能更高的查询类型实例。

2.2.1 前缀查询范例

演示查询改写过程的最好方式莫过于通过范例深入了解该过程的内部实现机制，尤其是要了解原始查询中的词项是如何改写成目标查询中那些词项的。假设索引了下面这些文档中的数据：

```
curl -XPUT 'localhost:9200/clients/client/1' -d
'{
  "id": "1", "name": "Joe"
}'
curl -XPUT 'localhost:9200/clients/client/2' -d
'{
  "id": "2", "name": "Jane"
}'
curl -XPUT 'localhost:9200/clients/client/3' -d
'{
  "id": "3", "name": "Jack"
}'
curl -XPUT 'localhost:9200/clients/client/4' -d
'{
  "id": "4", "name": "Rob"
}'
curl -XPUT 'localhost:9200/clients/client/5' -d
'{
  "id": "5", "name": "Jannet"
}'
```



也许用户想找出索引中所有 name 字段以字母 j 开头的文档。简单起见，我们在 clients 索引中执行以下查询：

```
curl -XGET 'localhost:9200/clients/_search?pretty' -d '{
  "query" : {
    "prefix" : {
      "name" : "j",
      "rewrite" : "constant_score_boolean"
    }
  }
}'
```

这里使用了一个简单的前缀查询，想检索出所有 name 字段以字母 j 开头的文档。我们

同时也设置了查询改写属性以确定执行查询改写的具体方法，不过现在先跳过该参数，具体的参数值将在本章的后续部分讨论。

执行前面的查询，将得到以下结果：

```
{
  ...
  "hits" : {
    "total" : 4,
    "max_score" : 1.0,
    "hits" : [ {
      "_index" : "clients",
      "_type" : "client",
      "_id" : "5",
      "_score" : 1.0, "_source" : {"id":"5", "name":"Jannet"}
    }, {
      "_index" : "clients",
      "_type" : "client",
      "_id" : "1",
      "_score" : 1.0, "_source" : {"id":"1", "name":"Joe"}
    }, {
      "_index" : "clients",
      "_type" : "client",
      "_id" : "2",
      "_score" : 1.0, "_source" : {"id":"2", "name":"Jane"}
    }, {
      "_index" : "clients",
      "_type" : "client",
      "_id" : "3",
      "_score" : 1.0, "_source" : {"id":"3", "name":"Jack"}
    } ]
  }
}
```

如你所见，返回结果中有 3 个文档，这些文档的 name 字段以字母 j 开头。我们并没有显式设置待查询索引的映射，因此 ElasticSearch 猜测 name 字段的映射，并将其设置为字符串类型并进行文本分析。可使用下面的命令：

```
curl -XGET 'localhost:9200/clients/client/_mapping?pretty'
```

ElasticSearch 会返回类似下面的结果：

```
{
  "client" : {
    "properties" : {
      "id" : {
        "type" : "string"
      },
      "name" : {
        "type" : "string"
      }
    }
  }
}
```

2.2.2 回顾 Apache Lucene

现在回顾一下 Lucene。如果你还记得 Lucene 倒排索引是如何构建的，就知道倒排索引中包含了词项，词频以及文档指针（如果忘了，请重新阅读 1.1 节）。我们看看之前存储到 clients 索引中的数据是如何组织的：

词项	计数	文档
jack	1	<3>
jane	2	<2><5>
joe	1	<1>
rob	1	<4>

Term 这一列非常重要。如果去探究 ElasticSearch 和 Lucene 的内部实现，就会发现前缀查询已经改写为下面这种查询：

```
ConstantScore(name:jack name:jane name:joe)
```

这意味着我们的前缀查询已经改写为常数得分查询（constant score query），该查询由一个布尔查询构成，而这个布尔查询又由三个词项查询构成。Lucene 所做的事情就是：枚举索引中的词项，并利用这些词项的信息来构建新的查询。当我们比较改写前后的两个查询的执行效果，会发现改写后的查询性能有所提升，尤其是当索引中有大量不同词项时。

如果想手动构建这个改写后的查询，可执行类似下面的代码（本范例代码存储在 constant_score_query.json 文件中）：

```
{
  "query" : {
    "constant_score" : {
      "query" : {
        "bool" : {
          "should" : [
            {
              "term" : {
                "name" : "jack"
              }
            },
            {
              "term" : {
                "name" : "jane"
              }
            },
            {
              "term" : {
                "name" : "joe"
              }
            }
          ]
        }
      }
    }
}
```

```

        }
    }
}
}
```

现在，让我们看看有哪些可以配置的查询改写属性。

2.2.3 查询改写的属性

之前已经说过可以对任何多词项查询（如 ElasticSearch 中的前缀和通配符查询）使用 rewrite 参数来控制查询改写。我们可以将 rewrite 参数存放在代表实际查询的 JSON 对象中，例如，下面的代码：

```

{
  "query" : {
    "prefix" : {
      "name" : "j",
      "rewrite" : "constant_score_boolean"
    }
  }
}
```

现在我们来看看 rewrite 参数有哪些选项可以配置：

- ❑ scoring_boolean：该选项将每个生成的词项转化为布尔查询中的一个或从句（should clause）。这种处理方法比较耗 CPU（因为要计算和保存每个词项的得分），而且有些查询生成的词项太多从而超出了布尔查询的限制，默认为 1024 个从句。改写后的查询会保存计算出来的得分。默认的布尔查询限制可以通过设置 elasticsearch.yml 文件的 index.query.bool.max_clause_count 属性来修改。但需谨记，改写后的布尔查询的从句越多，查询性能越低。
- ❑ constant_score_boolean：该选项与前面提到的 scoring_boolean 类似，但是 CPU 消耗较少，这是因为该过程并不计算每个从句的得分，而是每个从句得到一个与查询权重相同的常数得分，默认情况下等于 1，当然我们也可以通过设置查询权重来改变这个默认值。与 scoring_boolean 类似，该选项也有布尔从句数的限制。
- ❑ constant_score_filter：正如 Lucene 的 Javadoc 描述的那样，该选项按如下方式改写原始查询：通过顺序遍历每个词项来创建一个私有的过滤器，标记跟每个词项相关的所有文档。命中的文档被赋予一个跟查询权重相同的常量得分。当命中词项数或文档数较大时，该方法比 scoring_boolean 和 constant_score_boolean 执行速度更快。
- ❑ top_terms_N：该选项将每个生成的词项转化为布尔查询中的一个或从句，并保存计算出来的查询得分。与 scoring_boolean 不同之处在于，该方法只保留了最佳的前 N 个词项，从而避免超出布尔从句数的限制。

- `top_terms_boost_N`：该选项与 `top_terms_N` 类似，不同之处在于该选项产生的从句类型为常量得分查询，得分为从句的权重。



当 `rewrite` 属性设置为 `constant_score_auto` 或者根本不设置时，`constant_score_filter` 或 `constant_score_boolean` 属性的取值依赖于查询类型及其构造方式。

现在，让我们再看一个例子。如果想在范例查询中使用 `top_terms_N` 选项，并且 N 的值设置为 10，那么查询看起来与下面的代码类似：

```
{
  "query" : {
    "prefix" : {
      "name" : "j",
      "rewrite" : "top_terms_10"
    }
  }
}
```

结束本节之前，读者应该会产生一个疑问，即如何决定何时采用何种查询改写方法？该问题的答案更多取决于具体的应用场景。简单来说，如果你能接受低精度（往往伴随着高性能），那么可以采用 `top N` 查询改写方法。如果你需要更高的查询精度（往往伴随着低性能），那么应该使用布尔方法。

2.3 二次评分

有的时候，改变查询返回文档的顺序是很有好处的。这么做的理由有很多，其中之一便是出于对性能的考虑，例如，在整个文档集上计算文档顺序是非常耗时的，而在原始查询的返回文档的子集上做这种计算则非常省事。你可以想象一下，二次评分给了用户很多机会来定制业务逻辑。现在，让我们来看看这个功能，以及它能带给我们哪些便利。

2.3.1 理解二次评分

ElasticSearch 中的二次评分指的是重新计算查询返回文档中指定个数文档的得分。这意味着 ElasticSearch 会截取查询返回文档的前 N 个，并使用预定义的二次评分方法来重新计算它们的得分。

2.3.2 范例数据

我们的范例数据保存在 `documents.json` 文件中（本书提供了这些代码），并可用以下命令添加到索引中：

```
curl -XPOST localhost:9200/_bulk?pretty --data-binary @documents.json
```

2.3.3 查询

让我们从下面这个简单查询入手：

```
{
  "fields" : ["title", "available"],
  "query" : {
    "match_all" : {}
  }
}
```

执行该查询将返回索引中的所有文档。因为使用了 `match_all` 查询类型，所以每个返回文档的得分都等于 1.0，这充分体现了二次评分对查询返回文档集的影响。值得一提的是，我们在查询中指定只返回文档的 `title` 和 `available` 字段。

2.3.4 二次评分查询的结构

二次评分查询范例如下所示：

```
{
  "fields" : ["title", "available"],
  "query" : {
    "match_all" : {}
  },
  "rescore" : {
    "query" : {
      "rescore_query" : {
        "custom_score" : {
          "query" : {
            "match_all" : {}
          },
          "script" : "doc['year'].value"
        }
      }
    }
  }
}
```

前面的范例中，你可以在 `rescore` 对象中看到一个 `query` 对象。在本书中 `query` 只是该对象的唯一选项，但在将来的版本中可能会有其他方式来影响二次评分。因而这里的范例中，只是使用了一个简单的查询返回所有的文档，并且将每个文档的得分改写为该文档的 `year` 字段中的值（本范例只是用来演示二次评分，并没有实际意义）。

如果将该查询保存在 `query.json` 文件中，并使用下面的命令执行该查询：curl localhost:9200/library/book/_search?pretty -d @query.json，则会看到下面这些文档（这里忽略了响应消息的结构）：

```
"_score" : 1962.0,
```

```

"title" : "Catch-22",
"available" : false
"_score" : 1937.0,
"title" : "The Complete Sherlock Holmes",
"available" : false
"_score" : 1930.0,
"title" : "All Quiet on the Western Front",
"available" : true
"_score" : 1887.0,
"title" : "Crime and Punishment",
"available" : true

```

正如大家所见，ElasticSearch 找到了与原始查询匹配的所有文档。现在让我们看看各个文档的得分。ElasticSearch 截取了前 N 个文档，并对它们使用了第二个查询。这么做的结果是，文档的得分等于两个查询的得分之和。

读者应该知道，执行脚本性能低下，因此我们在第二个查询上再使用它。范例中的第一个 match_all 查询可能会返回成千上万的文档，如果直接在这里使用基于脚本的评分，那么性能会非常低下。由于二次评分使得我们可以在原始查询返回文档的前 N 个文档上重新计算得分，因而降低了对性能的影响。

现在，让我们看看如何对二次评分调优以及有哪些选项可供配置。

2.3.5 二次评分参数配置

在 rescore 对象中的查询对象中，必须配置下面这些参数：

- ❑ window_size：窗口大小，该参数默认设置为 from 和 size 参数值之和。它提供了之前提到的 N 个文档的相关信息。该参数值指定了每个分片上参与二次评分的文档个数。
- ❑ query_weight：查询权重值，默认等于 1，原始查询的得分与二次评分的得分相加之前将乘以该值。
- ❑ rescore_query_weight：二次评分查询的权重值，默认等于 1，二次评分查询的得分在与原始查询得分相加之前，将乘以该值。
- ❑ rescore_mode：二次评分的模式，默认设置为 total，ElasticSearch 0.90.3 引入了该参数（之前版本中类似的行为，该参数只能设置为 total），它定义了二次评分中文档得分的计算方式，可用的选项有 total、max、min、avg 和 multiply。当我们设置该参数值为 total 时，文档得分为原始查询得分与二次评分得分之和。当该参数值设置为 max，文档得分为原始查询得分与二次评分得分中的最大值。与 max 选项类似，当该参数值设置为 min 时，文档得分为两次查询得分中的最小值。以此类推，参数值为 avg 时，文档得分为两次查询得分的平均值。当参数值为 multiply 时，文档得分为两次查询得分的乘积。

例如，当 rescore_mode 参数值为 total 时，文档得分按照下面的公式计算：

```
original_query_score * query_weight + rescore_query_score *
rescore_query_weight
```



请记住，在ElasticSearch 0.90.3之前，还不支持`rescore_mode`参数，二次评分机制只能使用默认的`total`选项。

2.3.6 小结

有的时候，我们会需要显示查询结果，并使页面（page）上靠前文档的顺序能受一些额外的规则控制。但遗憾的是，我们并不能通过二次评分来实现。也许读者会第一时间想到`window_size`参数，然而实际上这个参数与返回列表中靠前文档并无关联，它只是指定了每个分片应该返回的文档数。除此之外，`window_size`不能小于页面大小（如果小于页面大小，ElasticSearch 会不予警告地将`window_size`设置为页面大小）。另外，有个非常重要的事实：二次评分功能并不能与排序一起使用，这是因为排序发生在二次评分之前，所以排序没有考虑后续新计算出来的文档得分。

前面提到的各种二次评分功能的限制和缺陷（例如，对返回文档的前3个文档及后续的5个文档分别采用不同的二次评分定义）可能限制了该功能的使用，因而用户在使用之前应三思。

2.4 批量操作

在本书的几个范例中，我们使用了批量索引格式携带数据，这种方式允许我们高效地发送数据至ElasticSearch。ElasticSearch提供了批量操作功能来读取数据和检索。值得一提的是，这些操作与批量索引类似，允许用户将多个请求归到一组，尽管每个请求可能有各自的目标索引和类型。现在，让我们看看都有哪些批量操作功能。

2.4.1 批量取

批量取（MultiGet）可以通过`_mget`端点（endpoint）操作，它允许使用一个请求获取多个文档。与实时获取功能类似，文档获取也是实时的，ElasticSearch会返回那些被索引的文档，而不论这些文档可用于搜索还是暂时对查询不可见。请查看下面的操作：

```
curl localhost:9200/library/book/_mget?fields=title -d '{
  "ids" : [1,3]
}'
```

该操作获取了两个特定的文档，其中索引名和类型在URL中定义。同时在前面的范例中，我们也设置了要获取的字段名（通过使用字段请求参数，即`request`参数）。ElasticSearch返回了如下形式的文档集：

```
{
  "docs" : [ {
    "_index" : "library",
    "_type" : "book",
    "_id" : "1",
    "_version" : 1,
    "exists" : true,
    "fields" : {
      "title" : "All Quiet on the Western Front"
    }
  }, {
    "_index" : "library",
    "_type" : "book",
    "_id" : "3",
    "_version" : 1,
    "exists" : true,
    "fields" : {
      "title" : "The Complete Sherlock Holmes"
    }
  } ]
}
```

前面的范例也可以写成这种更紧凑的形式：

```
curl localhost:9200/library/book/_mget?fields=title -d '{
  "docs" : [{ "_id" : 1}, { "_id" : 3}]
}'
```

这种形式更便于批量获取具有如下特点的文档集：不同文档有不同的目标索引及类型，或者不同文档返回的字段组合不同。在当前范例中，URL 中包含的信息被看作默认值。例如，我们查看下面这个查询：

```
curl localhost:9200/library/book/_mget?fields=title -d '{
  "docs" : [
    { "_index": "library_backup", "_id" : 1, "fields": ["otitle"]},
    { "_id" : 3}
  ]
}'
```

该查询返回了 ID 为 1 跟 3 的两个文档，但是第一个文档从索引 library_backup 中获取，而第二个文档则从索引 library 中获取（因为 URL 中定义的索引名为 library，因此将它作为默认值）。除此之外，在第一个文档中，我们限制只返回文档的 ottitle 字段。

 随着 ElasticSearch 1.0 的发布，MultiGet API 允许用户设置要操作文档的版本。如果文档的版本与请求版本不一致，则 ElasticSearch 不会执行相应操作。另外，还有两个参数：version，该参数允许用户传递感兴趣的版本信息；version_type，拥有 internal 和 external 两个取值。

2.4.2 批量查询

与批量取类似，批量查询允许用户将多个查询请求打包到一组。不过它的分组略有不同，更像批量索引操作。ElasticSearch 将输入解析成一行一行的文本，而文本行（每一对）包含了目标索引、其他参数以及查询串等信息。请查看下面这个简单的范例：

```
curl localhost:9200/library/books/_msearch?pretty --data-binary '
{
  "type" : "book"
}
{
  "filter" : { "term" : { "year" : 1936} }
}
{
  "search_type": "count"
}
{
  "query" : { "match_all" : {} }
}
{
  "index" : "library-backup", "type" : "book"
}
{
  "sort" : ["year"]
}'
```

正如你所见，查询请求被发送到 `_msearch` 端点。URL 中的索引名及类型是可选的，并且会作为剩余输入行的默认参数。剩余行可用于存储搜索类型信息（`search_type`）以及查询执行的路由或提示信息（`preference`）。因为这些参数并不是必需的，在某些特殊情况下，行中可以包含空对象（`{}`）甚至行本身为空。请求的偶数行负责携带真正的查询。现在，让我们看看该请求的查询结果：

```
{
  "responses" : [
    {
      "took" : 2,
      "timed_out" : false,
      "_shards" : {
        "total" : 5,
        "successful" : 5,
        "failed" : 0
      },
      "hits" : [
        {
          "total" : 1,
          "max_score" : 1.0,
          "hits" : [ {
            ...
          } ]
        }
      ],
      ...
    },
    ...
  {
    "took" : 2,
    "timed_out" : false,
    "_shards" : {
      "total" : 5,
      "successful" : 5,
      "failed" : 0
    },
    "hits" : [
      {
        "total" : 4,
        "max_score" : null,
      }
    ]
  }
]
```

```

    "hits" : [ {
      ...
    } ]
  }
}

```

返回的 JSON 结果包含了与批量搜索中的查询相对应的响应对象 (response object) 数组。如前所述，批量查询允许我们将多个独立的查询打包到一个请求中，因此，与之对应的，不同查询的返回文档可能具有不同的结构（本范例省略了）。



请记住，批量搜索就像批量索引一样，请求中不允许包含任何多余的值。每一行都有明确的用途，因此，请确保每行后面都紧随换行符，并且确保用于发送查询的工具没有对发送的数据做任何修改。这就是为什么我们在使用 curl 命令行时，使用了 `--data-binary` 选项而不是 `-d` 选项，因为后者并不保留换行符。

2.5 排序

当你发送查询至 Elasticsearch，返回文档默认按文档得分降序排序（文档评分计算详见 2.1 节）。这种排序策略通常是我们想要的，即第一个文档是与查询最相关的。然而，有时候我们希望能改变这种排序方式。下面这个例子就很容易做到，因为我们在查询中只使用了简单的词项字符串。请查看下面的范例：

```

{
  "query" : {
    "terms" : {
      "title" : [ "crime", "front", "punishment" ],
      "minimum_match" : 1
    }
  },
  "sort" : [
    { "section" : "desc" }
  ]
}

```

该查询范例会返回所有在 `title` 字段上至少命中一个词项的文档，并基于 `section` 字段数据排序。

也可以通过添加查询的 `sort` 部分的 `missing` 属性为那些 `section` 字段有缺失值的文档定制排序行为。例如，这样设置之前查询范例的 `sort` 配置，将 `section` 字段值缺失的文档排在最后：

```
{ "section" : { "order" : "asc", "missing" : "_last" }}
```

2.5.1 基于多值字段的排序

在 0.90 版本之前，ElasticSearch 在基于多值字段排序时存在一些问题。尝试这种排序时，通常会导致下面这种错误：[Can't sort on string types with more than one value per doc, or more than one token per field]。事实上，基于多值字段排序没有任何意义，因为 ElasticSearch 并不知道依照字段的哪个值进行排序。随着 ElasticSearch 0.90 的出现，基于多值字段的排序变得可行了。例如，某些文档的 release_dates 字段里存储了多个电影上映日期（同一部电影在不同国家的上映日期不同）。如果我们使用了 ElasticSearch 0.90，我们可以如此构造查询请求：

```
{
  "query" : {
    "match_all" : {}
  },
  "sort" : [
    {"release_dates" : { "order" : "asc", "mode" : "min" }}
  ]
}
```

请注意，在本小节的范例中，查询请求的 query 部分是多余的（它与默认设置相同），后面的范例将不再赘述。

例子中，ElasticSearch 将基于每个文档的 release_dates 字段的最小值进行排序。mode 参数可以设置为以下这些值：

- min：升序排序的默认值，ElasticSearch 将依照该字段的最小值进行排序。
- max：降序排序的默认值，ElasticSearch 将依照该字段的最大值进行排序。
- avg：ElasticSearch 将依照该字段的平均值进行排序。
- sum：ElasticSearch 将依照该字段的总和进行排序。

请注意，后面两个选项只对数值类型字段有效。尽管目前的版本能对文本类型字段使用 avg 和 sum 选项，但是返回结果可能并不符合用户的预期，因此并不鼓励使用这种用法。

2.5.2 基于多值 geo 字段的排序

ElasticSearch 的 0.92.0RC2 版本提供了基于多维坐标数据的排序。从用户角度来看，这种排序与前面提到的排序没有什么区别。现在，我们通过一个真实的例子来进一步了解这种类型的排序。例如，要查找特定国家里离自己最近的一个机构。假设我们使用下面这个映射：

```
{
  "mappings": {
    "poi": {
      "properties": {
```

```

        "country": { "type": "string" },
        "loc": { "type": "geo_point" }
    }
}
}
}
```

同时，使用下面这条数据记录：

```
{ "country": "UK", "loc": ["51.511214,-0.119824", "53.479251,
-2.247926", "53.962301,-1.081884"] }
```

我们的查询也很简单，如下所示：

```
{
  "sort": [
    {
      "_geo_distance": {
        "loc": "51.511214,-0.119824",
        "unit": "km",
        "mode" : "min"
      }
    }
  ]
}
```

正如你所见，我们有一个文档，文档中存储了多个地理位置的坐标。现在，我们执行查询并查看结果：

```
{
  "took" : 21,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 1,
    "max_score" : null,
    "hits" : [ {
      "_index" : "map",
      "_type" : "poi",
      "_id" : "1",
      "_score" : null, "_source" : {
        "country": "UK", "loc": ["51.511214,-0.119824",
        "53.479251,-2.247926", "53.962301,-1.081884"] }
    },
    "sort" : [ 0.0 ]
  }
}
```

从中可以看到，返回结果包含这个值：“sort”：[0.0]。这是因为返回文档中的地理坐标

与查询中的坐标精确匹配。如果我们在查询中配置 mode 属性的值为 max，则会返回一个不同的值，返回结果的高亮部分就会是下面的值：

```
"sort" : [ 280.4459406165739 ]
```



在 ElasticSearch 0.90.1 版本中，可以设置 mode 属性的值为 avg，此时可以基于字段中的地理坐标与查询中坐标的距离的均值排序。

2.5.3 基于嵌套对象的排序

本节将介绍最后一种排序技术，即在 ElasticSearch 0.90 及以上版本中，用户可以基于字段中定义的嵌套对象排序。基于嵌套文档的字段排序，对以下两种情形都适用：使用了显式嵌套映射（在映射中配置 type="nested"）的文档以及使用了对象类型的文档。但是，两者之间的一些细微区别仍需要注意。

假设我们索引了如下数据：

```
{
  "country": "PL", "cities": { "name": "Cracow", "votes": {
    "users": "A" } }
}
{
  "country": "EN", "cities": { "name": "York", "votes": [{ "users": "B" }, { "users": "C" } ] }
}
{
  "country": "FR", "cities": { "name": "Paris", "votes": {
    "users": "D" } }
}
```

正如你所见，文档中有嵌套对象，并且某些字段中有多个值（如存在多张选票）。请查看下面这个查询：

```
{
  "sort": [ { "cities.votes.users": { "order": "desc", "mode": "min" } } ]
}
```

查询返回结果按嵌套对象的 users 字段最小值降序排序。如果我们将嵌套文档中的子文档视为一种数据类型，则可以将查询简化为如下形式：

```
{
  "sort": [ { "users": { "order": "desc", "mode": "min" } } ]
}
```

当使用对象类型（object type）时，可以简化查询，这是因为整个对象结构被当成一个 Lucene 文档进行存储的。而在使用嵌套类型的时候，ElasticSearch 需要更多精确的字段信息，这是因为这些文档确实是独立的 Lucene 文档。但有些时候，使用 nested_path 属性会更加便捷，我们按照下面这种方式构造一个查询：

```
{
  "sort": [{"users": {"nested_path": "cities.votes", "order": "desc", "mode": "min"}}]
}
```

请记住，我们也可以使用 nested_filter 参数，只是该参数仅对嵌套文档有效（即那些被显式标记为嵌套文档的）。利用这个参数，我们可以在排序前就已经通过一个过滤器在检索期排除了某些文档，而不是在检索结果文档集中过滤它们。

2.6 数据更新 API

在索引一个新文档的时候，Lucene 会对每个字段进行分析并产生词条流，词条流中的词条可能会经过滤器的额外处理，而没有过滤掉的词条会写入倒排索引中。索引过程中，一些不需要的信息可能会被抛弃，这些信息包括：某些特殊词条的位置（当词项向量没有存储时），特定词汇（停用词或同义词），词条的变形（如词干还原）。因此，我们无法更新索引中的文档，并且在每次修改文档时不得不向索引发送文档所有字段的数据。但 Elasticsearch 可以通过使用 _source 伪字段存储和检索文档的原始数据来解决这个问题。当用户需要更改文档时，Elasticsearch 会获取 _source 字段中的值，做相应的修改，然后向索引提交一个新文档。当然，为了使这个特性生效，_source 字段必须是可用的。更新命令一个很大的局限性就是它只能更新单个的文档，目前还不支持通过查询实现批量更新。



如果你不熟悉 Apache Lucene 的文本分析处理机制或任何前面提到的术语，请参考 1.1 节（Apache Lucene 简介）。

从 API 的角度来看，文档更新可以通过执行发送至端点的更新请求来实现，也可以通过在更新请求的 url 中添加 _update 参数来更新某个特定的文档，如 /library/book/1/_update。现在，我们来看看 Elasticsearch 提供了哪些更新功能。

作为示例，本节的其余部分都将使用下面命令所索引的文档：

```
curl -XPUT localhost:9200/library/book/1 -d '{
  "title": "The Complete Sherlock Holmes", "author": "Arthur Conan Doyle", "year": 1936, "characters": ["Sherlock Holmes", "Dr. Watson", "G. Lestrade"], "tags": [], "copies": 0, "available": false, "section": 12
}'
```

2.6.1 简单字段更新

在本节的第一个案例里，我们尝试更新指定文档的字段。例如，使用下面的命令：

```
curl -XPOST localhost:9200/library/book/1/_update -d '{
  "doc": {
    "title": "The Complete Sherlock Holmes Book",
    "available": true
  }
}'
```

```

        "year" : 1935
    }
}'

```

其中，我们更新了文档中的两个字段，title 字段与 year 字段。作为响应，ElasticSearch 将返回一个与建索引操作类似的回复：

```
{"ok":true,"_index":"library","_type":"book","_id":"1","_version"
2}
```

现在，如果我们想从索引中获取刚才修改的文档，查看那两个字段是否真得修改了，可执行下面的命令：

```
curl -XGET localhost:9200/library/book/1?pretty
```

该命令的响应如下所示：

```
{
  "_index" : "library",
  "_type" : "book",
  "_id" : "1",
  "_version" : 2,
  "exists" : true, "_source" : {"title":"The Complete Sherlock
    Holmes Book","author":"Arthur Conan
    Doyle","year":1935,"characters":["Sherlock Holmes","Dr.
    Watson","G.
    Lestrade"],"tags":[],"copies":0,"available":false,
  "section":12}
}
```

正如我们所见，_source 字段中，title 字段与 year 字段的值已经被修改过了。接下来，我们查看下一个范例，该范例通过脚本来更新文档。

2.6.2 使用脚本按条件更新

有些时候，在修改文档的时候添加一些额外的逻辑是很有好处的，基于这点考虑，ElasticSearch 允许用户结合脚本使用更新 API。例如，我们发送下面这样的请求：

```
curl localhost:9200/library/book/1/_update -d '{
  "script" : "if(ctx._source.year == start_date) ctx._source.year
    = new_date; else ctx._source.year = alt_date;",
  "params" : {
    "start_date" : 1935,
    "new_date" : 1936,
    "alt_date" : 1934
  }
}'
```

正如你所见，script 字段定义了要对文档进行的操作，这可以是任何脚本。在范例中我们指派了 ctx 变量来引用源文档，但一般来说，脚本中会定义多个变量。通过使用 ctx._source，我们可以修改当前字段或创建新字段（如果引用不存在的字段，ElasticSearch 会自

动创建这个字段), 这正是范例中 `ctx._source.year = new_date` 语句产生的动作。此外, 也可以使用 `remove()` 方法来移除某些字段, 例如:

```
curl localhost:9200/library/book/1/_update -d '{
  "script" : "ctx._source.remove(\"year\");"
}'
```

2.6.3 使用更新 API 创建或删除文档

更新 API 不仅仅可以用来修改字段, 也可以用来操作整个文档。`upsert` 属性允许用户在当 URL 中地址不存在时创建一个新的文档。请查看下面这个命令:

```
curl localhost:9200/library/book/1/_update -d '{
  "doc" : {
    "year" : 1900
  },
  "upsert" : {
    "title" : "Unknown Book"
  }
}'
```

该命令修改了某个已有文档的 `year` 字段 (该文档位于索引 `library` 中, `book` 类型, 文档 ID 为 1)。如果该文档不存在, 将会创建一个新文档, 并且该文档会创建一个新字段 `title`, 如请求命令中 `upsert` 部分定义的那样。此外, 前面的命令也可以使用脚本重写为以下形式:

```
curl localhost:9200/library/book/1/_update -d '{
  "script" : "ctx._source.year = 1900",
  "upsert" : {
    "title" : "Unknown Book"
  }
}'
```

最后一个有趣的特性是有条件地移除整个文档, 具体可以通过设置 `ctx.op` 的值为 `delete` 来实现。例如, 可以通过下面的命令从索引中删除文档:

```
curl localhost:9200/library/book/1/_update -d '{
  "script" : "ctx.op = \"delete\""
}'
```

当然, 我们可以使用脚本实现更复杂的逻辑来删除满足特定规则的文档。

2.7 使用过滤器优化查询

ElasticSearch 允许用户创建他们熟知的各种不同的查询类型。当需要决定哪些文档与查询匹配并应该返回时, 仅有查询本身是不够的。ElasticSearch 查询 DSL 提供的大多数查询类型都有它们的相似物, 并且能将相似物包装 (wrapping) 成以下这些查询类型使用:

- constant_score
- filtered
- custom_filters_score

那么问题来了：“什么时候使用过滤器，什么时候仅需使用查询类型？”。现在，让我们揭晓它的答案。

2.7.1 过滤器与缓存

首先，过滤器是很好的缓存候选方案，正如你所预料的那样，ElasticSearch 提供了一种特殊的缓存，即过滤器缓存 (filter cache)，用来存储过滤器的结果。而且，被缓存的过滤器并不需要消耗过多的内存（因为它们只存储了哪些文档能与过滤器相匹配的相关信息），而且可供后续所有与之相关的查询重复使用，从而极大地提高了查询性能。想象一下，你执行了下面这个简单的查询：

```
{
  "query" : {
    "bool" : {
      "must" : [
        {
          "term" : { "name" : "joe" }
        },
        {
          "term" : { "year" : 1981 }
        }
      ]
    }
  }
}
```

该查询返回了在 name 字段包含 joe 以及在 year 字段包含 1981 的所有文档。尽管这个查询很简单，但是它能查询出满足指定姓名和出生年代条件的足球运动员。

在当前这种情形下，只有同时满足这两个条件的查询才会被缓存起来。因此，如果我们想搜索名字相同而出生年代不同的足球运动员，那么之前的查询就不再适用了。所以现在让我们想想如何来优化这个查询：人名有太多种可能，因此它不是完美的缓存候选对象，而出生年代则是 (year 字段中并没有很多不同的值，难道不是吗？)。于是，我们使用另外一种查询方法，该查询组合了查询类型与过滤器：

```
{
  "query" : {
    "filtered" : {
      "query" : {
        "term" : { "name" : "joe" }
      },
      "filter" : {
        "term" : { "year" : 1981 }
      }
    }
  }
}
```

```

    }
}
```

我们已经使用了 `filtered` 查询来同时包含查询和过滤器元素。第一次执行该查询以后，过滤器就会被 Elasticsearch 缓存起来，如果后续的其他查询也要使用该过滤器，则它将会被重复利用，从而避免 Elasticsearch 重复加载相关数据。

并不是所有过滤都默认被缓存

缓存很有用，但事实上 Elasticsearch 并不是默认缓存所有过滤器。这是因为在 Elasticsearch 中某些过滤器使用了字段数据缓存，这是一种特殊的缓存，它可以在基于字段数据的排序时使用，也能在计算切面结果时使用。以下过滤器默认不缓存：

- `numeric_range`
- `script`
- `geo_bbox`
- `geo_distance`
- `geo_distance_range`
- `geo_polygon`
- `geo_shape`
- `and`
- `or`
- `not`

上面提到的过滤器中，最后三个本身并不使用字段缓存，但由于它们操作其他过滤器，因而它们不缓存。而且，它们操作的过滤器在必要的时候已经被缓存起来了。

改变 Elasticsearch 的缓存行为

如果有需要，Elasticsearch 允许用户通过设置 `_cache` 和 `_cache_key` 属性来开启或关闭过滤器的缓存机制。回到先前的例子，进行相关配置从而缓存词项过滤器结果，缓存的 key 为 `year_1981_cache`：

```
{
  "query" : {
    "filtered" : {
      "query" : {
        "term" : { "name" : "joe" }
      },
      "filter" : {
        "term" : {
          "year" : 1981,
          "_cache_key" : "year_1981_cache"
        }
      }
    }
}
```

```

        }
    }
}
```

现在，关闭该查询的词项过滤器缓存：

```
{
  "query" : {
    "filtered" : {
      "query" : {
        "term" : { "name" : "joe" }
      },
      "filter" : {
        "term" : {
          "year" : 1981,
          "_cache" : false
        }
      }
    }
  }
}
```

为什么要为 cache 的 key 命名

也许读者会有疑问，为什么非要手动设置 `_cache_key` 属性，难道 ElasticSearch 不会自动处理它吗？当然，必要时可以让 ElasticSearch 自动处理，但有时候需要更精细地控制缓存行为，就需要手动处理了。例如，已知某些查询很少执行，但又想周期性地清除之前查询的缓存。如果不设置 `_cache_key`，那么将不得不强制清理全部的过滤器缓存；而如果设置了 `_cache_key`，只需执行下面的命令：

```
curl -XPOST 'localhost:9200/users/_cache/clear?filter_keys=year_1981_cache'
```

何时改变 ElasticSearch 过滤器缓存的行为

有些时候，用户比 ElasticSearch 更清楚自己想要什么。例如，你也许想在查询时使用 `geo_distance` 过滤器使之只返回距离较近的文档，同时确保该过滤器能与其他多个具有相同脚本过滤器（script filter）参数的查询一起使用，从而将多次使用同一个脚本。在这种场景中，就值得开启这些过滤器的缓存。每时每刻用户都应该问自己：“我会多次使用该过滤器还是只使用一次？”由于将数据存放在缓存中会消耗资源，因而在不需要时应及时清理数据。

2.7.2 词项查找过滤器

缓存和各种标准查询并不是 ElasticSearch 的全部家当。随着 ElasticSearch 0.90 的发布，又一个精巧的过滤器可供我们使用了，它用于给一个具体的查询传递从 ElasticSearch 取回的多个词项（与 SQL 的 IN 操作符类似）。

现在看一个简单的例子。假设有一个在线书店，并且存储了书店客户所购买书籍的相关信息。索引 book 看起来非常简单（索引映射信息存储在 books.js 文件中）：

```
{
  "mappings" : {
    "book" : {
      "properties" : {
        "id" : { "type" : "string", "store" : "yes", "index" :
          "not_analyzed" },
        "title" : { "type" : "string", "store" : "yes", "index" :
          "analyzed" }
      }
    }
  }
}
```

前面的代码并没有什么稀奇之处，它仅仅列出了书籍的 ID 和标题而已。

现在，我们再看看 clients.json 文件，这里存储了用于描述索引 clients 结构的映射信息：

```
{
  "mappings" : {
    "client" : {
      "properties" : {
        "id" : { "type" : "string", "store" : "yes", "index" :
          "not_analyzed" },
        "name" : { "type" : "string", "store" : "yes", "index" :
          "analyzed" },
        "books" : { "type" : "string", "store" : "yes", "index" :
          "not_analyzed" }
      }
    }
  }
}
```

我们有了客户 ID、姓名以及客户所购买书籍的 ID 列表。除此之外，还索引了一些范例数据：

```
curl -XPUT 'localhost:9200/clients/client/1' -d '{
  "id":"1", "name":"Joe Doe", "books":["1","3"]
}'
curl -XPUT 'localhost:9200/clients/client/2' -d '{
  "id":"2", "name":"Jane Doe", "books":["3"]
}'
curl -XPUT 'localhost:9200/books/book/1' -d '{
  "id":"1", "title":"Test book one"
}'
curl -XPUT 'localhost:9200/books/book/2' -d '{
  "id":"2", "title":"Test book two"
}'
curl -XPUT 'localhost:9200/books/book/3' -d '{
```

```
"id": "3", "title": "Test book three"
}'
```

现在，考虑一下如何获取某个特定用户购买的所有书籍，例如，ID 为 1 的用户。当然，我们可以执行下面这个 get 请求：curl -XGET 'localhost:9200/clients/client/1'，来获取存有该客户信息的文档，然后利用文档中的 books 字段信息来执行下面这个查询：

```
curl -XGET 'localhost:9200/books/_search' -d '{
  "query" : {
    "ids" : {
      "type" : "book",
      "values" : [ "1", "3" ]
    }
  }
}'
```

幸运的是，我们有更简捷的做法，ElasticSearch 0.90 新提供了一个查找过滤器，它允许用户将前面的两个查询合并为一个过滤查询 (filtered query)，如下面代码所示：

```
curl -XGET 'localhost:9200/books/_search' -d '{
  "query" : {
    "filtered" : {
      "query" : {
        "match_all" : {}
      },
      "filter" : {
        "terms" : {
          "id" : {
            "index" : "clients",
            "type" : "client",
            "id" : "1",
            "path" : "books"
          },
          "_cache_key" : "terms_lookup_client_1_books"
        }
      }
    }
  }
}'
```

请注意参数 _cache_key 的值。正如你所见，它被设置为 terms_lookup_client_1_books，并包含有用户的 ID。但值得警惕的是，当在多个不同的查询中重用该 _cache_key 的值时，你可能会得到错误的或者是预料之外的检索结果。这是因为 ElasticSearch 会在某个特定的 key 下缓存某个查询的结果，并在其他查询执行时重用这些结果。

现在，来查看一下前面那个查询的响应结果：

```
{
  ...
  "hits" : {
    "total" : 2,
```

```

    "max_score" : 1.0,
    "hits" : [ {
        "_index" : "books",
        "_type" : "book",
        "_id" : "1",
        "_score" : 1.0, "_source" : {"id":"1", "title":"Test book
                                     one"}
    }, {
        "_index" : "books",
        "_type" : "book",
        "_id" : "3",
        "_score" : 1.0, "_source" : {"id":"3", "title":"Test book
                                     three"}
    } ]
}
}

```

这正是我们想要的。看起来有点不可思议吧？

它是如何工作的

查看一下我们发送至 ElasticSearch 的查询。它只是一个简单的带过滤器的查询，即在一个匹配所有文档的查询上外加了一个 terms 过滤器。但是在这里，词项过滤器的使用方式略有不同，我们并没有显式确定具体的词项，而是让 ElasticSearch 去索引中加载它们。

正如你所见，我们感兴趣的是查询的 id 对象，这是因为它包含多个属性，且过滤器的执行也依赖于它。具体包含的属性有：index、type、id 和 path。index 属性指明了所要加载词项所在的索引（本范例中是 clients 索引）。type 属性告诉 ElasticSearch 我们对哪些文档类型感兴趣（本范例中是 client）。id 属性指示了需要加载对象所在文档的 ID。path 字段用来告诉 ElasticSearch 从哪个字段加载词项，在我们的范例中，指的是 clients 索引的 books 字段。

总而言之，ElasticSearch 所做的就是从 clients 索引中 ID 为 1，类型为 client 的文档的 books 字段中加载词项，并将这些词项用于词项过滤器中，从而筛选那些 books 索引（因为我们在该索引中执行查询）中 id 字段包括这些词项的文档（请注意词项过滤器的名字：id）。



请记住，为了使词项查找功能生效，需要先存储 _source 字段。

性能考虑

前面的查询执行在 ElasticSearch 内部已经通过缓存机制优化了，即相关词项已经被加载到过滤器缓存中且由查询指定的 key 下了。除此之外，一旦将这些词项（即那些书籍的 ID）加载到缓存，后续的查询就不会再重复加载，这意味着 ElasticSearch 能提高此类查询的性能。

如果在词项查找过程中使用到的数据量不大，建议索引（如范例中的索引 clients）只创

建一个分片，并且在所有出现了 books 索引的节点上为该分片保存一个副本。之所以这么建议是因为 ElasticSearch 优先在本地执行词项查询以避免不必要的网络开销和延迟，进而能提高查询的性能。

从内部对象中加载词项

如果客户信息的 books 字段类型由数值类型数组变为内部对象数组，那么查询也应该做相应的修改，此时应修改查询的 id 属性，使之包含对象嵌套的相关信息。例如，将 "id": "books" 修改为 "id": "books.book"。

词项查询过滤器缓存设置

前面提到过，为了提供词项查找功能，ElasticSearch 引进了一种新的缓存类型，它使用了一种快速的 LRU（最近最少使用算法）缓存来处理词项缓存。



想了解更多关于 LRU 缓存及其工作机制的内容，请参考：http://en.wikipedia.org/wiki/Page_replacement_algorithm#Least_recently_used 中的文档。

为了配置这种缓存，用户可以在 elasticsearch.yml 文件中配置下面这些属性：

- ❑ indices.cache.filter.terms.size：默认设置 ElasticSearch 词项查找缓存的最大内存使用量为 10mb。对于大多数案例来说，这个默认值够用了，但如果要加载更多的数据至缓存中，那么可以调大该参数值。
- ❑ indices.cache.filter.terms.expire_after_access：该属性配置了自上次查询以来的超时长。默认不超时。
- ❑ indices.cache.filter.terms.expire_after_write：该属性配置了数据加载至缓存以后多长时间将超时。默认不超时。

2.8 ElasticSearch 切面机制中的过滤器与作用域

当使用 ElasticSearch 的切面机制时，有几件事情需要注意。首先要记住的是，系统只在查询结果之上计算切面结果。如果你在 filter 对象内部且在 query 对象外部包含了过滤器，那么这些过滤器将不会对参与切面计算的文档产生影响。另外一个需要注意的事情是作用域（scope），它能扩充用于切面计算的文档。接下来我们来看几个范例。

2.8.1 范例数据

先回忆一下查询、过滤器、切面是如何一起工作的。为了演示该功能，我们先使用下面的命令索引一些文档至索引 books 中：

```
curl -XPUT 'localhost:9200/books/book/1' -d '{
  "id": "1", "title": "Test book 1", "category": "book",
  "price": 29.99
```

```

}'
curl -XPUT 'localhost:9200/books/book/2' -d '{
  "id": "2", "title": "Test book 2", "category": "book",
  "price": 39.99
}'
curl -XPUT 'localhost:9200/books/book/3' -d '{
  "id": "3", "title": "Test comic 1", "category": "comic",
  "price": 11.99
}'
curl -XPUT 'localhost:9200/books/book/4' -d '{
  "id": "4", "title": "Test comic 2", "category": "comic",
  "price": 15.99
}'

```

2.8.2 切面计算和过滤

让我们查看一下当使用查询和过滤器的时候，切面计算是如何工作的。我们先执行一个简单的查询，该查询返回索引 books 中的所有文档。为了缩小返回结果集，过程中使用了一个过滤器，用来限制查询只返回 category 字段值为 book 的文档。此外，我们还在 price 字段上使用了一个简单的基于范围的切面计算，用来查看有多少书籍的 price 低于 30，多少书籍的 price 高于 30。整个查询如下所示（代码存储在 query_with_filter.json 中）：

```

{
  "query" : {
    "match_all" : {}
  },
  "filter" : {
    "term" : { "category" : "book" }
  },
  "facets" : {
    "price" : {
      "range" : {
        "field" : "price",
        "ranges" : [
          { "to" : 30 },
          { "from" : 30 }
        ]
      }
    }
  }
}

```

查询执行以后，将得到下面的返回结果：

```
{
  ...
  "hits" : {
    "total" : 2,
    "max_score" : 1.0,
    "hits" : [ {
      "_index" : "books",
      "_type" : "book",
      "_id" : "1",
      ...
    }, {
      ...
    } ]
  }
}
```

```

    "_score" : 1.0, "_source" : {"id":"1", "title":"Test book
    1", "category":"book", "price":29.99}
  },
  {
    "_index" : "books",
    "_type" : "book",
    "_id" : "2",
    "_score" : 1.0, "_source" : {"id":"2", "title":"Test book
    2", "category":"book", "price":39.99}
  }
]
},
"facets" : {
  "price" : {
    "_type" : "range",
    "ranges" : [ {
      "to" : 30.0,
      "count" : 3,
      "min" : 11.99,
      "max" : 29.99,
      "total_count" : 3,
      "total" : 57.97,
      "mean" : 19.32333333333334
    }, {
      "from" : 30.0,
      "count" : 1,
      "min" : 39.99,
      "max" : 39.99,
      "total_count" : 1,
      "total" : 39.99,
      "mean" : 39.99
    } ]
  }
}
}

```

尽管查询被限制为只返回 category 字段值为 book 的文档，但是对切面计算来说并不是这样。事实上，切面作用于 books 索引的所有文档（因为使用了 match_all 查询）。因此，读者需要意识到 ElasticSearch 的切面在进行计算时并不考虑过滤器的因素。当过滤器作为是查询的一部分时，例如使用 filtered 查询类型时，将会发生哪些动作呢？我们来进一步探究。

2.8.3 过滤器作为查询的一部分

再次尝试前面的例子，只是这次使用过滤查询类型。于是，我们再次结合 filtered 检索所有 category 字段值为 book 的文档，并且在 price 字段上使用了一个简单的基于范围的切面计算，以查看有多少书籍的价格高于 30 以及多少书籍的价格低于 30。为了实现这个功能，我们执行下面这个查询（代码存储在 filtered_query.json 中）：

```
{
  "query" : {
    "filtered" : {
      "query" : {

```

```

        "match_all" : {}
    },
    "filter" : {
        "term" : {
            "category" : "book"
        }
    }
},
"facets" : {
    "price" : {
        "range" : {
            "field" : "price",
            "ranges" : [
                { "to" : 30 },
                { "from" : 30 }
            ]
        }
    }
}
}
}

```

该查询返回结果如下所示：

```
{
    ...
    "hits" : {
        "total" : 2,
        "max_score" : 1.0,
        "hits" : [ {
            "_index" : "books",
            "_type" : "book",
            "_id" : "1",
            "_score" : 1.0, "_source" : {"id":"1", "title":"Test book
                1", "category":"book", "price":29.99}
        }, {
            "_index" : "books",
            "_type" : "book",
            "_id" : "2",
            "_score" : 1.0, "_source" : {"id":"2", "title":"Test book
                2", "category":"book", "price":39.99}
        } ]
    },
    "facets" : {
        "price" : {
            "_type" : "range",
            "ranges" : [ {
                "to" : 30.0,
                "count" : 1,
                "min" : 29.99,
                "max" : 29.99,
            }
        }
    }
}
```

```
        "total_count" : 1,
        "total" : 29.99,
        "mean" : 29.99
    }, {
        "from" : 30.0,
        "count" : 1,
        "min" : 39.99,
        "max" : 39.99,
        "total_count" : 1,
        "total" : 39.99,
        "mean" : 39.99
    } ]
}
}
}
}
```

正如你所见，切面计算作用于查询返回结果上，且跟事前预期的结果一样，这正是因为过滤器成为了查询的一部分！在我们的案例中，切面计算结果包含两个范围，每个范围只有一个文档。

2.8.4 切面过滤器

考虑一下，如果只想为 title 字段包含词项 2 的书籍计算分组，我们应该怎么做。如果在查询使用第二个过滤器，虽然能减少查询返回结果，但这并不是我们想要的。一种更明智的做法是使用切面过滤器 (facet filter)。

在切面类型的相同层级上使用 facet_filter 过滤器，能通过使用过滤器减少计算切面时的文档数，就像在查询时使用过滤器那样。例如，如果用户想通过使用 facet_filter 将范围切面计算过滤成只包含 title 字段值为 2 的文档，那么需要将查询中切面计算相关部分修改为下面这样（完整的查询保存在 filtered_query_facet_filter.json 中）：

```
{
...
"facets" : {
    "price" : {
        "range" : {
            "field" : "price",
            "ranges" : [
                { "to" : 30 },
                { "from" : 30 }
            ]
        },
        "facet_filter" : {
            "term" : {
                "title" : "2"
            }
        }
    }
}
```

```

    }
}
```

正如你所见，我们新引入了一个名为 term 的简单过滤器。修改后的查询的返回结果看起来与下面的类似：

```
{
  ...
  "hits" : {
    "total" : 2,
    "max_score" : 1.0,
    "hits" : [ {
      "_index" : "books",
      "_type" : "book",
      "_id" : "1",
      "_score" : 1.0, "_source" : {"id":"1", "title":"Test book
1", "category":"book", "price":29.99}
    }, {
      "_index" : "books",
      "_type" : "book",
      "_id" : "2",
      "_score" : 1.0, "_source" : {"id":"2", "title":"Test book
2", "category":"book", "price":39.99}
    } ]
  },
  "facets" : {
    "price" : {
      "_type" : "range",
      "ranges" : [ {
        "to" : 30.0,
        "count" : 0,
        "total_count" : 0,
        "total" : 0.0,
        "mean" : 0.0
      }, {
        "from" : 30.0,
        "count" : 1,
        "min" : 39.99,
        "max" : 39.99,
        "total_count" : 1,
        "total" : 39.99,
        "mean" : 39.99
      } ]
    }
  }
}
```

查看第一个返回结果，你就能发现不同之处。通过在查询中使用切面过滤器，虽然能限制切面计算只作用于单个文档，但我们的查询仍然能返回两个文档。

2.8.5 全局作用域

如果我们想查询所有书名中包含词项 2 的文档，但同时又要显示索引中所有文档的基

于范围的切面计算结果，应该如何处理呢？幸运的是，此处并不需要强制运行第二个查询，这是因为可以使用全局切面作用域（global faceting scope）来达成目的，并具体通过将切面类型的 global 属性配置为 true 来实现。

例如，修改我们前面用过的第一个查询。在本小节，我们并不包含过滤器，取而代之的是一个词项查询。除此之外，还需添加 global 属性，从而查询看起来与下面的查询类似（该查询保存在 query_global_scope.json 中）：

```
{
  "query" : {
    "term" : { "category" : "book" }
  },
  "facets" : {
    "price" : {
      "range" : {
        "field" : "price",
        "ranges" : [
          { "to" : 30 },
          { "from" : 30 }
        ]
      },
      "global" : true
    }
  }
}
```

现在，查看该查询的返回结果：

```
{
  ...
  "hits" : {
    "total" : 2,
    "max_score" : 0.30685282,
    "hits" : [ {
      "_index" : "books",
      "_type" : "book",
      "_id" : "1",
      "_score" : 0.30685282, "_source" : { "id": "1", "title": "Test book 1", "category": "book", "price": 29.99 }
    }, {
      "_index" : "books",
      "_type" : "book",
      "_id" : "2",
      "_score" : 0.30685282, "_source" : { "id": "2", "title": "Test book 2", "category": "book", "price": 39.99 }
    } ]
  },
  "facets" : {
    "price" : {
      "_type" : "range",
      "global" : true
    }
  }
}
```

```
"ranges" : [ {
    "to" : 30.0,
    "count" : 3,
    "min" : 11.99,
    "max" : 29.99,
    "total_count" : 3,
    "total" : 57.97,
    "mean" : 19.32333333333334
}, {
    "from" : 30.0,
    "count" : 1,
    "min" : 39.99,
    "max" : 39.99,
    "total_count" : 1,
    "total" : 39.99,
    "mean" : 39.99
} ]
}
}
```

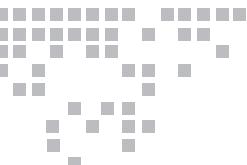
尽管查询只返回了两个文档，但后面依然对索引中所有文档进行了切面计算，这归功于查询中的 `global` 属性。

我们再来看一个关于 `global` 属性的使用案例，即展示基于切面计算的导航。想象一下这种场景：在用户输入他的查询之后显示一个顶级导航，例如，使用基于词项的切面计算来枚举所有电子商务网站的顶级目录。这种案例中 `global` 作用域就显得尤为有用了。

2.9 小结

在本章中，我们了解了 Apache Lucene 是如何工作的，查询改写是什么，如何使用二次评分影响文档的评分。接下来，了解了如何在一个 HTTP 请求中发送多个查询和实时读取请求，以及如何对包含多值字段和嵌套文档的数据进行排序。另外，还介绍了数据更新 API 和使用过滤器来优化查询的方法。最后，学习了如何使用过滤器和作用域来缩减或扩大切面计算返回的文档集。

下一章中，我们将学习如何选择不同的评分公式，以及在索引期选择不同的倒排索引格式（postings format），了解多语言数据处理和事务日志（transaction log）配置，以及深入理解 ElasticSearch 缓存工作机制。



底层索引控制



在上一章，我们了解了 Apache Lucene 如何为文档评分，什么是查询重写，如何利用 ElasticSearch 0.90 中的新特性，即二次评分来影响搜索返回文档的得分。同时我们也讨论了如何使用单个 HTTP 请求发送多个查询或准实时读取请求，以及如何对数据进行基于多值字段或嵌套文档的排序。除此之外，还介绍了如何使用数据更新 API 以及如何通过使用过滤器优化查询。最后，我们介绍了如何通过使用过滤器和作用域来缩减或增加用于切面计算的文档数量。本章涵盖以下内容：

- 如何使用不同的评分公式及其特性。
- 如何使用不同的倒排表格式及其特性。
- 如何处理准实时搜索、实时读取，以及搜索器重新打开之后发生动作。
- 深入理解多语言数据处理。
- 配置搜索事务日志以满足应用需求，并查看它对部署的影响。
- 段合并、各种索引合并策略和合并调度方式。

3.1 改变 Apache Lucene 的评分方式

自 2012 年 Apache Lucene 4.0 发布以后，用户便可以改变默认的基于 TF/IDF 的评分算法了，这是因为 Lucene 的 API 做了一些改变，使得用户能轻松地修改和扩展该评分公式。然而，这并不是 Lucene 在改变文档评分计算方面仅有的改进。Lucene 4.0 提供了更多的相似度模型，从而允许我们采用不同的评分公式。本节中，我们将深入了解 Lucene 4.0 带来了哪些变化以及如何整合这些特性至 ElasticSearch 中。

3.1.1 可用的相似度模型

前面已经说过，Apache Lucene 4.0 之前，除了最原始和默认的相似度模型以外，TF/IDF 模型也是可用的。详细内容请参见 2.1 节。

而现在，又新增了以下三种相似度模型可供使用：

- ❑ Okapi BM25 模型：这是一种基于概率模型的相似度模型，可用于估算文档与给定查询匹配的概率。为了在 ElasticSearch 中使用它，你需要使用该模型的名字，BM25。一般来说，Okapi BM25 模型在短文本文档上的效果最好，因为这种场景中重复词项对文档的总体得分损害较大。
 - ❑ 随机偏离（Divergence from randomness）模型：这是一种基于同名概率模型的相似度模型。为了在 ElasticSearch 中使用它，你需要使用该模型的名字，DFR。一般来说，随机偏离模型在类似自然语言的文本上效果较好。
 - ❑ 基于信息的（Information based）模型：这是最后一个新引入的相似度模型，与随机偏离模型类似。为了在 ElasticSearch 中使用它，你需要使用该模型的名字，IB。同样，IB 模型也在类似自然语言的文本上拥有较好的效果。



前面提到的相似度模型所涉及的数学知识已经远远超出本书的讨论范围。如果想深入了解这些模型以及拓展相关知识，请参考 http://en.wikipedia.org/wiki/Okapi_BM25 (Okapi BM25 模型)，以及 http://terrier.org/docs/v3.5/dfr_description.html (随机偏离模型)。

3.1.2 为每字段配置相似度模型

自 ElasticSearch 0.90 以后，用户可以在映射中为每字段设置不同的相似度模型。例如，假设我们有下面这个映射，用于索引博客的回帖（该映射存储在 posts_no_similarity.json 文件中）：

```
{
  "mappings" : {
    "post" : {
      "properties" : {
        "id" : { "type" : "long", "store" : "yes", "index" : "no" },
        "precision_step" : "0",
        "name" : { "type" : "string", "store" : "yes", "index" : "analyzed" },
        "contents" : { "type" : "string", "store" : "no", "index" : "analyzed" }
      }
    }
  }
}
```

我们希望的是，在 name 字段和 contents 字段中使用 BM25 相似度模型。为了实现这个目的，我们需要扩展当前的字段定义，即添加 similarity 字段，并将该字段的值设置为相应的相似度模型的名字。修改后的映射（该映射存储在 posts_similarity.json 文件中）如下所示：

```
{
  "mappings" : {
    "post" : {
      "properties" : {
        "id" : { "type" : "long", "store" : "yes",
                  "precision_step" : "0" },
        "name" : { "type" : "string", "store" : "yes", "index" :
                  "analyzed", "similarity" : "BM25" },
        "contents" : { "type" : "string", "store" : "no", "index" :
                  "analyzed", "similarity" : "BM25" }
      }
    }
  }
}
```

以上更改就足够了，并不需要额外的信息。经过前面的处理，Apache Lucene 将在搜索期在 name 字段和 contents 字段上使用 BM25 相似度模型来计算文档得分。

 对于随机偏离模型和基于信息的相似度模型，我们需要配置一些额外属性，用于控制这些相似度模型的行为。相关知识后面会详细讲述。

3.2 相似度模型配置

我们已经知道如何为索引中各个字段配置相似度模型了，现在来了解如何按需求配置它们。事实上，这相当容易。我们所要做的就是，在索引配置相关部分提供相应的相似度模型配置信息，就像下面的代码这样（本范例存储在 posts_custom_similarity.json 文件中）：

```
{
  "settings" : {
    "index" : {
      "similarity" : {
        "mastering_similarity" : {
          "type" : "default",
          "discount_overlaps" : false
        }
      }
    }
  },
  "mappings" : {
    "post" : {
      "properties" : {
```

```

    "id" : { "type" : "long", "store" : "yes",
    "precision_step" : "0" },
    "name" : { "type" : "string", "store" : "yes", "index" :
    "analyzed", "similarity" : "mastering_similarity" },
    "contents" : { "type" : "string", "store" : "no", "index"
    : "analyzed" }
}
}
}
}

```

尽管用户可以配置多个相似度模型，但此时还是先回到前面的范例。我们定义了一个名为 `mastering_similarity` 的新的相似度模型，它基于默认的 TF/IDF 相似度模型。接着将它的 `discount_overlaps` 属性值设置为 `false`，指定该相似度模型用于 `name` 字段。关于不同相似度模型都有哪些属性本章后面会详细描述，现在，我们先讨论如何改变 ElasticSearch 的默认相似度模型。

3.2.1 选择默认的相似度模型

为了设置默认的相似度模型，我们需要提供关于一个名为 `default` 的相似度模型的配置信息。例如，要使用 `mastering_similarity` 模型作为默认的相似度模型，需要将前面的配置文件修改为如下形式（该范例存储在 `posts_default_similarity.json` 文件中）：

```

{
  "settings" : {
    "index" : {
      "similarity" : {
        "default" : {
          "type" : "default",
          "discount_overlaps" : false
        }
      }
    }
  },
  ...
}

```

虽然所有的相似度模型都全局使用了 `query norm` 和 `coord` 这两个评分因子（详见 2.1 节），但是它们又从 `default` 相似度模型的配置中移除出去了。然而，ElasticSearch 允许用户根据需要改变这种状况。为了实现该目的，用户需要另外定义一个名为 `base` 的相似度模型，它的定义方式与前面的范例如出一辙，将相似度模型的名字由 `default` 改为 `base` 即可。参考下面的代码（该范例代码保存在 `posts_base_similarity.json` 文件中）：

```

{
  "settings" : {
    "index" : {

```

```

    "similarity" : {
      "base" : {
        "type" : "default",
        "discount_overlaps" : false
      }
    }
  },
...
}

```

如果 base 相似度模型出现在索引配置中，那么当 Elasticsearch 使用其他相似度模型计算文档得分时，会使用 base 相似度模型来计算 query norm 和 coord 评分因子。

3.2.2 配置被选用的相似度模型

每个新增的相似度模型都可以根据用户需求进行配置，而 Elasticsearch 还允许用户不加配置地直接使用 default 和 BM25 相似度模型。因为它们是预先配置好的，而 DFR 和 IB 模型则需要进一步配置才能使用。现在，我们来看看各个相似度模型都提供了哪些可配置的属性。

配置 TF/IDF 相似度模型

在 TF/IDF 相似度模型案例中，我们可以只设置一个参数：discount_overlaps 属性，其默认值为 true。默认情况下，位置增量（position increment）为 0（即该词条的 position 计数与前一个词条相同）的词条在计算评分时并不会被考虑进去。如果在计算文档时需要考虑这类词条，则需要将相似度模型的 discount_overlaps 属性值设置为 false。

配置 Okapi BM25 相似度模型

在 Okapi BM25 相似度模型案例中，有如下参数可供配置：

- k1：该参数为浮点数，控制饱和度（saturation），即词频归一化中的非线性项。
- b：该参数为浮点数，用于控制文档长度对词频的影响。
- discount_overlaps：与 TF/IDF 相似度模型中的 discount_overlaps 参数作用相同。

配置 DFR 相似度模型

在 DFR 相似度模型案例中，有如下参数可供配置：

- basic_model：该参数值可设置为 be、d、g、if、in 和 ine。
- after_effect：该参数值可设置为 no、b 和 l。
- normalization：该参数值可设置为 no、h1、h2、h3 和 z。

如果 normalization 参数值不是 no，则需要设置归一化因子。归一化因子的设置依赖于所选的 normalization 参数值。参数值为 h1 时，使用 normalization.h1.c 属性；参数值为 h2 时，使用 normalization.h2.c 属性；参数值为 h3 时，使用 normalization.h3.c 属性；参数值为

`z` 时，使用 `normalization.z.z` 属性。这些属性值的数据类型均为浮点型。下面的代码展示了如何配置相似度模型：

```
"similarity" : {
    "esserverbook_dfr_similarity" : {
        "type" : "DFR",
        "basic_model" : "g",
        "after_effect" : "l",
        "normalization" : "h2",
        "normalization.h2.c" : "2.0"
    }
}
```

配置 IB 相似度模型

在 IB 相似度模型案例中，有如下参数可供配置：

- `distribution`: 该参数值可设置为 `ll` 或 `spl`。
- `lambda`: 该参数值可设置为 `df` 或 `tff`。

此外，IB 模型也需要配置归一化因子，它的配置方式与 DFR 模型相同，这里不再赘述。

下面的代码展示了如何配置 IB 相似度模型：

```
"similarity" : {
    "esserverbook_ib_similarity" : {
        "type" : "IB",
        "distribution" : "ll",
        "lambda" : "df",
        "normalization" : "z",
        "normalization.z.z" : "0.25"
    }
}
```

3.3 使用编解码器

Lucene 4.0 的另一个显著变化是允许用户改变索引文件编码方式。在此之前，只能通过修改 Lucene 内核代码来实现。而 Lucene 4.0 出现以后，这个不再是难题，它提供了灵活的索引方式，允许用户改变倒排索引的写入方式。

3.3.1 简单使用范例

读者也许会有疑问，这个功能真的有用吗？这个问题问得很好，为什么用户需要改变 Lucene 索引写入格式？理由之一是性能。某些字段需要特殊处理，如主键字段。相较于包含多个重复值的标准数值类型或文本类型而言，主键字段中的数值并不重复，只要借助一些技术，主键值就能被快速搜索到。用户还可以使用 `SimpleTextCodec` 来调试代码，以便了解到底是什么格式的数据写入了 Lucene 索引之中（请注意，编解码器是 Lucene 提供的功能，ElasticSearch 并没有相应的接口）。

3.3.2 工作原理解释

假设 posts 索引有下面这个映射 (该映射保存在 posts.json 文件中):

```
{
  "mappings" : {
    "post" : {
      "properties" : {
        "id" : { "type" : "long", "store" : "yes",
                  "precision_step" : "0" },
        "name" : { "type" : "string", "store" : "yes", "index" :
                  "analyzed" },
        "contents" : { "type" : "string", "store" : "no", "index" :
                      "analyzed" }
      }
    }
  }
}
```

编解码器需要逐字段配置。为了配置某个字段使用特定的编解码器，需要在字段配置文件中添加一个 postings_format 属性，并将具体的编解码器所对应的属性值赋给它，例如 pulsing。因此，我们需要对前面的映射文件进行修改 (该映射保存在 post_codec.json 文件中)，代码如下所示：

```
{
  "mappings" : {
    "post" : {
      "properties" : {
        "id" : { "type" : "long", "store" : "yes", "precision_step" :
                  "0", "postings_format" : "pulsing" },
        "name" : { "type" : "string", "store" : "yes", "index" :
                  "analyzed" },
        "contents" : { "type" : "string", "store" : "no", "index" :
                      "analyzed" }
      }
    }
  }
}
```

现在，执行下面的命令：

```
curl -XGET 'localhost:9200/posts/_mapping?pretty'
```

检查该命令的执行结果，看看编解码器配置是否在 ElasticSearch 中生效，如下所示：

```
{
  "posts" : {
    "post" : {
      "properties" : {
        "contents" : {
          "type" : "string"
```

```

        },
        "id" : {
          "type" : "long",
          "store" : true,
          "postings_format" : "pulsing",
          "precision_step" : 2147483647
        },
        "name" : {
          "type" : "string",
          "store" : true
        }
      }
    }
  }
}

```

正如我们所见，id 字段的编解码器类型已经变成我们所期望的那样了。



因为编解码器在 Lucene4.0 以后才出现，所以 ElasticSearch0.90 之前的版本并不支持相关功能。

3.3.3 可用的倒排表格式

我们可以使用下面这些倒排表格式。

- default：当没有显式配置时，倒排表使用该格式。该格式提供了存储字段（stored field）和词项向量压缩功能。如果想了解更多关于索引压缩的知识，可参考 <http://solr.pl/en/2012/11/19/solr-4-1-stored-fields-compression/>。
- pulsing：该编解码器将高基（high cardinality）字段[⊕]中的倒排表编码为词项数组，这会减少 Lucene 在搜索文档时的查找操作。使用该编解码器，可以提高在高基字段中的搜索速度。
- direct：该编解码器在读索引阶段将词项载入词典，且词项在内存中为未压缩状态。该编解码器能提升常用字段的查询性能，但也需要谨慎使用，由于词项和倒排表数组都需要存储在内存中，从而导致它非常消耗内存。



因为词项保存在 byte 数组中，所以每个索引段最多可以使用 2.1GB 的内存来存储这些词项。

- memory：顾名思义，该编解码器将所有数据写入磁盘，而在读取时则使用 FST（Finite State Transducers）结构直接将词项和倒排表载入内存。如果想了解更多 FST 相关信息，请参考 Mike McCandless 的博客：<http://blog.mikemccandless.com/2010/12/>

[⊕] 即包含大量不同值的字段。——译者注

[using-finite-state-transducers-in.html](#)。因为使用该编解码器时，数据都在内存中，因而它能加速常见词项的查询。

- ❑ bloom_default：是 default 编解码器的一种扩展，即在 default 编解码器处理基础上又加入了 bloom filter 的处理，且 bloom filter 相关数据会写入磁盘中。当读入索引时，bloom filter 相关数据会被读入内存，用于快速判断某个特定值是否存在。该编解码器在处理主键之类的高基字段时非常有用。想了解更多 bloom filter 信息请参考 http://en.wikipedia.org/wiki/Bloom_filter。
 - ❑ bloom_pulsing：它是 pulsing 编解码器的扩展，在 pulsing 编解码器处理基础上又加入了 bloom filter 的处理。

3.3.4 配置编解码器

默认配置中提供的倒排索引格式已足以应付大多数应用了，但偶尔还是需要定制倒排索引格式以满足具体的需求。这种情况下，ElasticSearch 允许用户更改索引映射中的 codec 部分来配置编解码器。例如，我们想配置 default 编解码器，并且将其命名为 custom_default，便可以通过定义下面这个映射来实现（该范例存储在 posts_codec_custom.jsonfile 文件中）：

```
{  
    "settings" : {  
        "index" : {  
            "codec" : {  
                "postings_format" : {  
                    "custom_default" : {  
                        "type" : "default",  
                        "min_block_size" : "20",  
                        "max_block_size" : "60"  
                    }  
                }  
            }  
        }  
    }  
},  
"mappings" : {  
    "post" : {  
        "properties" : {  
            "id" : { "type" : "long", "store" : "yes",  
                      "precision_step" : "0" },  
            "name" : { "type" : "string", "store" : "yes", "index" :  
                       "analyzed", "postings_format" : "custom_default" },  
            "contents" : { "type" : "string", "store" : "no", "index"  
                          : "analyzed" }  
        }  
    }  
}
```

如你所见，我们已经更改了 default 编解码器的 min_block_size 和 max_block_size 参数值，并且将新配置的编解码器命名为 custom_default。在这之后，我们对索引的 name 字段使用该倒排索引格式。

default 编解码器属性

当使用 default 编解码器时，可以配置下面这些参数：

- min_block_size：该参数确定了 Lucene 将词项词典（term dictionary）中的多个词项编码为块（block）时，块中的最小词项数。默认值为 25。
- max_block_size：该参数确定了 Lucene 将词项词典（term dictionary）中的多个词项编码为块（block）时，块中的最大词项数。默认值为 48。

direct 编解码器属性

当使用 direct 编解码器时，可以配置下面这些参数：

- min_skip_count：该参数确定了允许写入跳表（skip list）指针的具有相同前缀的词项的最小数量。默认值为 8。
- low_freq_cutoff：编解码器使用单个数组对象来存储那些文档频率（document frequency）低于该参数值的词项的倒排链及位置信息。默认值为 32。

memory 编解码器属性

当使用 memory 编解码器时，可以配置下面这些参数：

- pack_fst：该参数为布尔类型，默认设置为 false，用来确认保存倒排链的内存结构是否被打包为 FST（Finite State Transducers）类型。而打包为 FST 类型能减少保存数据所需的内存量。
- acceptable_overhead_ratio：该参数为浮点型，指定了内部结构的压缩率，默认值为 0.2。当该参数值为 0 时，虽然没有额外的内存消耗，但是这种实现方式会导致较低的性能。当该参数值为 0.5 时，将会多付出 50% 的内存消耗，但是这种实现方式能提升性能。参数值超过 1 的设置也是可行的，只是会导致更多的内存开销。

pulsing 编解码器属性

当使用 pulsing 编解码器时，除了可以设置 default 编解码器的那些参数，还有另外一个参数可供配置，请参考下面的描述：

- freq_cut_off：该参数默认设置为 1，是设置的一个文档频率阈值，若词项对应的文档频率小于等于该阈值，则将该词项的倒排链写入词典中。

基于 bloom filter 的编解码器属性

如果要配置基于 bloom filter 的编解码器，可使用 bloom_filter 类型并设置下面这些属性：

- delegate：该属性值用来确定将要被 bloom filter 包装（wrap）的编解码器。

□ ffp：该属性值介于 0 与 1.0 之间，用来确定期望的假阳率（false positive probability）。

我们可以依据每个索引段中的文档数设置多个 ffp 值。例如，默认情况下，10k 个文档时为 0.01，1m 个文档时为 0.03。这种配置的含义是：当索引段中文档数大于 10 000 个时，ffp 值使用 0.01，而当文档数超过 1 000 000 时，ffp 值使用 0.03。

例如，我们想在 direct 编解码器上包装基于 bloom filter 的编解码器（范例存储在 posts_bloom_custom.json 文件中）：

```
{
  "settings" : {
    "index" : {
      "codec" : {
        "postings_format" : {
          "custom_bloom" : {
            "type" : "bloom_filter",
            "delegate" : "direct",
            "ffp" : "10k=0.03,1m=0.05"
          }
        }
      }
    }
  },
  "mappings" : {
    "post" : {
      "properties" : {
        "id" : { "type" : "long", "store" : "yes",
                  "precision_step" : "0" },
        "name" : { "type" : "string", "store" : "yes", "index" :
                  "analyzed", "postings_format" : "custom_bloom" },
        "contents" : { "type" : "string", "store" : "no", "index" :
                      "analyzed" }
      }
    }
  }
}
```

3.4 准实时、提交、更新及事务日志

一个理想的搜索解决方案中，新索引的数据应该能立即搜索到。ElasticSearch 给人的第一印象仿佛就是如此工作的，即使是在多服务器环境下，然而事实并非如此（至少不是任何场景都能保证新索引的数据能被实时检索到），原因我们后面会讲解。接下来的案例中，我们将使用下面的命令将一篇文档索引到新创建的索引中：

```
curl -XPOST localhost:9200/test/test/1 -d '{ "title": "test" }'
```

现在，更新该文档，并尝试立即搜索它。为实现该目的，我们将串行执行下面的两个命令：

```
curl -XPOST localhost:9200/test/test/1 -d '{ "title": "test2" }' ; curl
localhost:9200/test/test/_search?pretty
```

前面的命令将返回类似下面的结果：

```
{"ok":true,"_index":"test","_type":"test","_id":"1","_version":2} {
  "took" : 1,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 1,
    "max_score" : 1.0,
    "hits" : [ {
      "_index" : "test",
      "_type" : "test",
      "_id" : "1",
      "_score" : 1.0, "_source" : { "title": "test" }
    } ]
  }
}
```

第一个返回结果对应索引命令的操作，正如你所见，一切正常。第二个返回结果是查询对应的结果，其中 title 字段的值应该是 test2，然而返回的却是修改前的那个文档，这是怎么回事？

关于上面这个问题，在给出答案之前，不妨去回顾一下 Lucene 的内部机制，探究一下 Lucene 是如何让新索引的文档在搜索时可用的。

3.4.1 索引更新及更新提交

在阅读 1.1 节时我们已经知道，在索引期新文档会写入索引段。索引段是独立的 Lucene 索引，这意味着查询是可以与索引并行的，只是不时会有新增的索引段被添加到可被搜索的索引段集合之中。Apache Lucene 通过创建后续的（基于索引只写一次的特性）segments_N 文件来实现此功能，且该文件列举了索引中的索引段。这个过程称为提交（committing），Lucene 以一种安全的方式来执行该操作，能确保索引更改以原子操作方式写入索引，即便有错误发生，也能保证索引数据的一致性。

让我们回到之前的例子，尽管第一个操作向索引中添加了文档，但它并没有执行提交 commit 操作，这就是返回结果令人惊讶的原因。然而，一次提交并不足以保证新索引的数据能被搜索到，这是因为 Lucene 使用了一个叫作 Searcher 的抽象类来执行索引的读取。如果索引更新提交了，但 Searcher 实例并没有重新打开，那么它觉察不到新索引段的加入。Searcher 重新打开的过程叫作刷新（refresh）。出于性能考虑，Lucene 推迟了耗时的刷新，因此它不会在每次新增一个文档（或批量增加文档）的时候刷新，但 Searcher 会每秒刷新一

次。这种刷新已经非常频繁了，然而有很多应用却需要更快的刷新频率。如果碰到这种情况，要么使用其他技术，要么审视需求是否合理。ElasticSearch 提供了强制刷新的 API。例如，在例子中可以使用下面的命令：

```
curl -XGET localhost:9200/test/_refresh
```

如果在搜索前执行该命令，就会得到我们预期的结果。

更改默认的刷新时间

Searcher 自动刷新的时间间隔可以通过以下手段改变：更改 ElasticSearch 配置文件中的 index.refresh_interval 参数值或者使用配置更新相关的 API。例如：

```
curl -XPUT localhost:9200/test/_settings -d '{
  "index" : {
    "refresh_interval" : "5m"
  }
}'
```

上面的命令将 Searcher 的自动刷新时间间隔更改为 5 分钟。请注意，上次刷新后的新增数据并不会被搜索到。

 刷新操作是很耗资源的，因此刷新间隔时间越长，索引速度越快。如果需要长时间高速建索引，并且在建索引结束之前暂不执行查询，那么可以考虑将 index.refresh_interval 参数值设置为 -1，然后在建索引结束以后再将该参数恢复为初始值。

3.4.2 事务日志

Apache Lucene 能保证索引的一致性，这非常棒，但是这并不能保证当往索引中写数据失败时不会损失数据（如磁盘空间不足、设备损坏，或没有足够的文件句柄供索引文件使用）。另外，频繁提交操作会导致严重的性能问题（因为每提交一次就会触发一个索引段的创建操作，同时也可能触发索引段的合并）。ElasticSearch 通过使用事务日志（transaction log）来解决这些问题，它能保存所有的未提交的事务，而 ElasticSearch 会不时创建一个新的日志文件用于记录每个事务的后续操作。当有错误发生时，就会检查事务日志，必要时会再次执行某些操作，以确保没有丢失任何更改信息。而且，事务日志的相关操作都是自动完成的，用户并不会意识到某个特定时刻触发的更新提交。事务日志中的信息与存储介质之间的同步（同时清空事务日志）称为事务日志刷新（flushing）。

 请注意事务日志刷新与 Searcher 刷新的区别。大多数情况下，Searcher 刷新是你所期望的，即搜索到最新的文档。而事务日志刷新用来确保数据正确写入了索引并清空了事务日志。

除了自动的事务日志刷新以外，也可以使用对应的 API。例如，可以使用下面的命令，强制将事务日志中涉及的所有数据更改操作同步到索引中，并清空事务日志文件：

```
curl -XGET localhost:9200/_flush
```

我们也可以使用 flush 命令对特定的索引进行事务日志刷新（如 library 索引）：

```
curl -XGET localhost:9200/library/_flush
```

```
curl -XGET localhost:9200/library/_refresh
```

上面第二行命令中，我们紧接着在事务日志刷新之后，调用 Searcher 刷新操作，打开一个新的 Searcher 实例。

事务日志相关配置

如果事务日志的默认配置不能满足用户需要，ElasticSearch 还支持默认配置修改功能以满足特定需求。以下参数既可以通过修改 `elasticsearch.yml` 文件来配置，也可以通过索引配置更新 API 来更改。

- ❑ `index.translog.flush_threshold_period`：该参数的默认值为 30 分钟，它控制了强制自动事务日志刷新的时间间隔，即便是没有新数据写入。强制进行事务日志刷新通常会导致大量的 I/O 操作，因此当事务日志涉及少量数据时，才更适合进行这项操作。
- ❑ `index.translog.flush_threshold_ops`：该参数确定了一个最大操作数，即在上次事务日志刷新以后，当索引更改操作次数超过该参数值时，强制进行事务日志刷新操作，默认值为 5000。
- ❑ `index.translog.flush_threshold_size`：该参数确定了事务日志的最大容量，当容量大于等于该参数值，就强制进行事务日志刷新操作，默认值为 200MB。
- ❑ `index.translog.disable_flush`：禁用事务日志刷新。尽管默认情况下事务日志刷新是可用的，但对它临时性地禁用能带来其他方面的便利。例如，向索引中导入大量文档的时候。

 尽管前面提及的所有参数都被指定用于用户选定的某个索引，但它们同时也定义了该索引各个分片的事务日志处理方式。

当然，除了通过修改 `elasticsearch.yml` 文件来配置上述参数，我们也可以使用设置更新 API 来更改相关配置。例如：

```
curl -XPUT localhost:9200/test/_settings -d '{
  "index" : {
    "translog.disable_flush" : true
  }
}'
```

前述命令会在向索引导入大量数据之前执行，大幅提高索引的速度。但是请记住，当数据导入完毕之后，要重新设置事务日志刷新相关参数。

3.4.3 准实时读取

事务日志给我们带来一个免费的特性：实时读取（real-time GET），该功能让返回文档各种版本（包括未提交版本）成为可能。实时读取操作从索引中读取数据时，会先检查事务日志中是否有可用的新版本。如果近期索引没有与事务日志同步，那么索引中的数据将会被忽略，事务日志中最新版本的文档将被返回。为了演示实时读取的工作原理，我们用下面的命令替换范例中的搜索操作：

```
curl -XGET localhost:9200/test/test/1?pretty
```

ElasticSearch 将返回类似下面的结果：

```
{
  "_index" : "test",
  "_type" : "test",
  "_id" : "1",
  "_version" : 2,
  "exists" : true, "_source" : { "title": "test2" }
}
```

如代码所示，这正是我们想要的结果，而且这里并没有使用 Searcher 刷新技巧就得到了最新版本的文档。

3.5 深入理解数据处理

刚开始使用 ElasticSearch 的时候，各种搜索方式和查询类型往往会令人感到头疼。查询类型之间行为各异，尽管有些差别非常细微，例如范围查询和前缀查询。了解这些差别对理解查询的工作原理至关重要，尤其是要在 ElasticSearch 提供的默认查询类型之外做些额外事情的时候，例如，处理多语言信息。

3.5.1 输入并不总是进行文本分析

在讨论查询分析之前，我们先使用以下命令创建一个索引：

```
curl -XPUT localhost:9200/test -d '{
  "mappings" : {
    "test" : {
      "properties" : {
        "title" : { "type" : "string", "analyzer" : "snowball" }
      }
    }
  }
}'
```

```

    }
}
}'
```

如你所见，这个索引非常简单。文档只包含一个字段，该字段使用 snowball 分析器进行处理。现在，我们通过执行下面这个命令来索引一个简单的文档：

```
curl -PUT localhost:9200/test/test/1 -d '{
  "title" : "the quick brown fox jumps over the lazy dog"
}'
```

此时索引中已经有文档了，我们不妨用一些查询来做一下测试，请仔细查看下面的两个命令：

```
curl localhost:9200/test/_search?pretty -d '{
  "query" : {
    "term" : {
      "title" : "jumps"
    }
  }
}'
```

```
curl localhost:9200/test/_search?pretty -d '{
  "query" : {
    "match" : {
      "title" : "jumps"
    }
  }
}'
```

第一个查询返回了目标文档，而第二个查询没有返回任何结果，两种情况对比悬殊！也许你已经知道（或猜测到）产生这种差异的原因，即该现象与文本分析有关。我们来比较一下，索引中存储了什么，又想搜索到什么。为实现该目的，可以通过执行下面的命令来使用文本分析（Analyze）API：

```
curl 'localhost:9200/test/_analyze?text=the+quick+brown+fox+jumps+over+the+lazy+dog&pretty&analyzer=snowball'
```

端点 _analyze 允许我们查看 ElasticSearch 是如何处理 text 参数中的输入的，同时也能指定要使用哪个分析器（通过 analyzer 参数）。



更多文本分析 API 特性请参考 <http://www.elasticsearch.org/guide/reference/api/admin-indices-analyze/>。

前面的命令经 ElasticSearch 处理后会返回类似下面的结果：

```
{  
  "tokens" : [ {  
    "token" : "quick",  
    "start_offset" : 4,  
    "end_offset" : 9,  
    "type" : "<ALPHANUM>",  
    "position" : 2  
  }, {  
    "token" : "brown",  
    "start_offset" : 10,  
    "end_offset" : 15,  
    "type" : "<ALPHANUM>",  
    "position" : 3  
  }, {  
    "token" : "fox",  
    "start_offset" : 16,  
    "end_offset" : 19,  
    "type" : "<ALPHANUM>",  
    "position" : 4  
  }, {  
    "token" : "jump",  
    "start_offset" : 20,  
    "end_offset" : 25,  
    "type" : "<ALPHANUM>",  
    "position" : 5  
  }, {  
    "token" : "over",  
    "start_offset" : 26,  
    "end_offset" : 30,  
    "type" : "<ALPHANUM>",  
    "position" : 6  
  }, {  
    "token" : "lazi",  
    "start_offset" : 35,  
    "end_offset" : 39,  
    "type" : "<ALPHANUM>",  
    "position" : 8  
  }, {  
    "token" : "dog",  
    "start_offset" : 40,  
    "end_offset" : 43,  
    "type" : "<ALPHANUM>",  
    "position" : 9  
  } ]  
}
```

从中可以看到 ElasticSearch 是如何将输入转化为词条流的。请回顾 1.1 节，并注意这个事实：每个词条都携带了它在原始文本中的位置信息、类型信息（尽管用户对该信息不感兴趣，但是可能会被过滤器使用到）、词项信息（即一个词，它存储在索引中，在检索期用于与查询中的词项匹配）。为什么原始文本 the quick brown fox jumps over the lazy dog 被转化成了这些词项：quick、brown、fox、jump、over、lazi（这里发生了有趣的变化），dog，

我们可以总结一下 snowball 分词器都做了哪些事情：

- 过滤非重要词（如 the）。
- 将单词转换为词干形式（如 jump）。
- 有时会进行糟糕的转换（如 lazi）。

第三种情况看起来并没有那么糟，只要同一个词的不同形式得到了统一转化。如果这种事情发生了，词干还原的目的就达到了，即 Elasticsearch 会对查询和索引中的词项进行匹配，而不管它最初的单词形态如何。现在，回过头来看看我们的查询。该查询旨在搜索一个简单的词项（如这个例子中的 jumps），然而索引中并没有该词项（索引中只有 jump）。因此，query 在搜索前应该先用分析器进行处理，此时会将 jumps 转换为 jump，然后再进行搜索操作。

现在，请看第二个范例：

```
curl localhost:9200/test/_search?pretty -d '{
  "query" : {
    "prefix" : {
      "title" : "lazy"
    }
  }
}'
curl localhost:9200/test/_search?pretty -d '{
  "query" : {
    "match_phrase_prefix" : {
      "title" : "lazy"
    }
  }
}'
```

范例中的两个查询看起来很相似，但查询结果却大相径庭。第一个查询什么也没返回（因为查询中的 lazy 文本与索引中的 lazi 并不相同），而第二个查询经过分词器处理，返回了我们预期的文档。

3.5.2 范例的使用

所有这些范例都比较有趣，且能从中发现，有些查询经过了文本分析处理，而有些没有。对我们来说最重要的事情是，如何自觉利用这些知识改进具体的搜索应用。

假如要搜索本书的内容，可能某些用户会搜索书中的章节名、地名或某个片段。因为我们没有自然语言分析工具，所以不能理解用户输入的这些短语。然而，从概率的角度来看，与查询短语在文本上精确匹配的文档应该是用户最感兴趣的。而从另外一个重要指标来看，与用户输入短语中词语精确匹配的文档才是用户感兴趣的。这里的词语精确匹配既

可以是语义上相同也可以是同一个词的不同形态。

为了演示，我们先用下面的命令创建一个只包含单字段文档的索引：

```
curl -XPUT localhost:9200/test -d '{
  "mappings" : {
    "test" : {
      "properties" : {
        "lang" : { "type" : "string" },
        "title" : {
          "type" : "multi_field",
          "fields" : {
            "i18n" : { "type" : "string", "index" : "analyzed",
                        "analyzer" : "english" },
            "org" : { "type" : "string", "index" : "analyzed",
                      "analyzer" : "standard" }
          }
        }
      }
    }
  }
}'
```

尽管只有一个字段，但却使用了两个分析器进行文本分析处理，这是因为 title 字段是 multi_field 类型的缘故，其中对 title.org 子字段使用了 standard 分析器，而对 title.i18n 子字段使用了 english 分析器（该分词器会将用户输入转换为词干形式）。

如果我们使用如下命令向索引中添加一个文档：

```
curl -XPUT localhost:9200/test/test/1 -d '{ "title" : "The quick brown
fox jumps over the lazy dog." }'
```

此时，索引中的 title.org 字段已有 jumps 词项，而 title.i18n 字段中也有了 jump 词项。然后我们执行下面的查询：

```
curl localhost:9200/test/_search?pretty -d '{
  "query" : {
    "multi_match" : {
      "query" : "jumps",
      "fields" : ["title.org^1000", "title.i18n"]
    }
  }
}'
```

我们的文档由于跟查询完美匹配而获得了较高的得分，这归功于对 field.org 字段的命中做了加权处理。field.i18n 字段的命中也贡献了部分得分，只是它对总得分的影响要小很多，因为我们并没有对该字段的命中做加权处理，它还是默认权值 1。

3.5.3 索引期更换分词器

另外一件值得一提的事情是，在处理多语言数据的时候，可能要在索引期中动态更换分词器。比如说，我们修改前面的映射，添加 _analyzer 相关配置：

```
curl -XPUT localhost:9200/test -d '{
  "mappings" : {
    "test" : {
      "_analyzer" : {
        "path" : "lang"
      },
      "properties" : {
        "lang" : { "type" : "string" },
        "title" : {
          "type" : "multi_field",
          "fields" : {
            "i18n" : { "type" : "string", "index" : "analyzed" },
            "org" : { "type" : "string", "index" : "analyzed",
                      "analyzer" : "standard" }
          }
        }
      }
    }
  }
}'
```

我们仅做了少许修改，首先是允许 ElasticSearch 在处理文本时根据文本内容决定采用何种分析器。其中，path 参数为文档中的字段名，该字段中保存了分析器的名称。其次是移除了 field.i18n 字段所用分析器的定义。现在，我们可以用下面这条命令来创建索引：

```
curl -XPUT localhost:9200/test/test/1 -d '{ "title" : "The quick brown
fox jumps over the lazy dog.", "lang" : "english" }'
```

上面的例子中，ElasticSearch 从索引中提取 lang 字段的值，并将该值代表的分析器置于当前文档的文本分析器处理。总之，当你想对不同文档采用不同分析器时，该设置非常有用（例如，在文档中移除或保留非重要词）。

3.5.4 搜索时更换分析器

也可以在搜索时更换分词器，并通过配置 analyzer 属性来实现。例如，下面这个查询：

```
curl localhost:9200/test/_search?pretty -d '{
  "query" : {
    "multi_match" : {
      "query" : "jumps",
      "fields" : ["title.org^1000", "title.i18n"],
      "analyzer": "english"
    }
  }
}'
```

如代码所示，ElasticSearch 会采用我们显式提到过的分析器。

3.5.5 陷阱与默认分析

索引期与检索期能针对文档更换分词器的机制是一个非常有用的特性，但它也会引入很多非常隐蔽的错误。其中之一就是没有定义分析器。针对这种情况，虽然 ElasticSearch 会选用一个所谓的默认分析器，但这往往并不是我们想要的。这是因为默认分析器有时候会被文本分析插件模块重定义。此时，有必要指定 ElasticSearch 的默认分析器。为了实现该目的，我们还是像平常那样定义分析器，只是将自定义分析器的名称替换为 default。

 作为一种备选方案，你可以定义 default_index 分析器和 default_search 分析器，并将它们作为索引期和检索期的默认分析器。

3.6 控制索引合并

读者知道（我们已经在第 1 章中讨论过），在 ElasticSearch 中每个索引都会创建一到多个分片以及零到多个副本，也知道这些分片或副本本质上都是 Lucene 索引，而 Lucene 索引又基于多个索引段构建（至少一个索引段）。索引文件中绝大部分数据都是只写一次，读多次，而只有用于保存文档删除信息的文件才会被多次更改。在某些时刻，当某种条件满足时，多个索引段会被拷贝合并到一个更大的索引段，而那些旧的索引段会被抛弃并从磁盘中删除，这个操作称为段合并（segment merging）。

也许你会有疑问，为什么非要进行段合并？这是因为：首先，索引段的个数越多，搜索性能越低且耗费内存更多。另外，索引段是不可变的，因而物理上你并不能从中删除信息。也许你碰巧从索引中删除了大量文档，但这些文档只是做了删除标记，物理上并没有被删除。而当段合并发生时，这些标记为删除的文档并没有复制到新的索引段中。如此一来，这减少了最终索引段中的文档数。



频繁的文档更改操作会导致大量的小索引段，从而导致文件句柄打开过多的问题。我们必须要对这种情况有所准备，如修改系统配置，或设置合适的大文件打开数。

从用户角度来看，段合并可快速概括为如下两个方面：

- 当多个索引段合并为一个的时候，会减少索引段的数量并提高搜索速度。
- 同时也会减少索引的容量（文档数），因为在段合并时会移除被标记为已删除的那些文档。

尽管段合并有这些好处，但用户也应该了解到段合并的代价，即主要是 I/O 操作的代价。在速度较慢的系统中，段合并会显著影响性能。基于这个原因，ElasticSearch 允许用户选择段合并策略（merge policy）及存储级节流（store level throttling）。本章后续部分将会讨论段合并策略，而存储级节流则安排在 6.2 节中讨论。

3.6.1 选择正确的合并策略

尽管段合并是 Lucene 的责任，ElasticSearch 也允许用户配置想用的段合并策略。到目前为止，有三种可用的合并策略：

- tiered（默认）
- log_byte_size
- log_doc

前面提到的每一种段合并策略都有各自的参数，而这些参数定义了各自的行为特点，并且它们的默认值都是可以覆写的（请阅读后续章节来了解这些参数）。

为了告知 ElasticSearch 我们想使用的段合并策略，可以将配置文件的 index.merge.policy.type 字段配置成我们期望的段合并策略类型。例如下面这样：

```
index.merge.policy.type: tiered
```



一旦使用特定的段合并策略创建了索引，它就不能被改变。但是，可以使用索引更新 API 来改变该段合并策略的参数值。

接下来我们来了解这些不同的段合并策略，以及它们提供的功能，并讨论这些段合并策略的具体配置。

tiered 合并策略

这是 ElasticSearch 的默认选项。它能合并大小相似的索引段，并考虑每层允许的索引段的最大个数。读者需要清楚单次可合并的索引段的个数与每层允许的索引段数的区别。在索引期，该合并策略会计算索引中允许出现的索引段个数，该数值称为 **阈值**（budget）。如果正在构建的索引中的段数超过了阈值，该策略将先对索引段按容量降序排序（这里考虑了被标记为已删除的文档），然后再选择一个成本最低的合并。合并成本的计算方法倾向于回收更多删除文档和产生更小的索引段。

如果某次合并产生的索引段的大小大于 index.merge.policy.max_merged_segment 参数值，则该合并策略会选择更少的索引段参与合并，使得生成的索引段的大小小于阈值。这意味着，对于有多个分片的索引，默认的 index.merge.policy.max_merged_segment 则显得过小，会导致大量索引段的创建，从而降低查询速度。用户应该根据自己具体的数据量，观察索引段的状况，不断调整合并策略以满足应用需求。

log byte size 合并策略

该策略会不断地以字节数的对数为计算单位，选择多个索引来合并创建新索引。合并过程中，时不时会出现一些较大的索引段，然后又产生出一些小于合并因子（merge factor）的索引段，如此循环往复。你可以想象，时而有一些相同数量级的索引段，其个数会变得比合并因子还少。当碰到一个特别大的索引段时，所有小于该级别的索引段都会被合并。索引中的索引段个数与下次用于计算的字节数的对数成正比。因此，该合并策略能够保持较少的索引段数量并且极小化段索引合并的代价。

log doc 合并策略

该策略与 log_byte_size 合并策略类似，不同的是前者基于索引的字节数计算，而后者基于索引段的文档数计算。以下两种情况中该合并策略表现良好：文档集中的文档大小类似或者你期望参与合并的索引段在文档数方面相当。

3.6.2 合并策略配置

我们现在已经知道索引的段合并策略的工作原理了，只是还缺乏配置方面的相关知识，所以现在就来讨论每种合并策略及其提供的配置选项。请记住，大多数情况下默认选项是够用的，除非有特殊的需求才需要修改。

配置 tiered 合并策略

当使用 tiered 合并策略时，可配置以下这些选项：

- ❑ index.merge.policy.expunge_deletes_allowed：默认值为 10，该值用于确定被删除文档的百分比，当执行 expungeDeletes 时，该参数值用于确定索引段是否被合并。
- ❑ index.merge.policy.floor_segment：该参数用于阻止频繁刷新微小索引段。小于该参数值的索引段由索引合并机制处理，并将这些索引段的大小作为该参数值。默认值为 2MB。
- ❑ index.merge.policy.max_merge_at_once：该参数确定了索引期单次合并涉及的索引段数量的上限，默认为 10。该参数值较大时，也就能允许更多的索引段参与单次合并，只是会消耗更多的 I/O 资源。
- ❑ index.merge.policy.max_merge_at_once_explicit：该参数确定了索引优化（optimize）操作和 expungeDeletes 操作能参与的索引段数量的上限，默认值为 30。但该值对索引期参与合并的索引段数量的上限没有影响。

- ❑ index.merge.policy.max_merged_segment：该参数默认值为 5GB，它确定了索引期间单次合并中产生的索引段大小的上限。这是一个近似值，因为合并后产生的索引段的大小是通过累加参与合并的索引段的大小并减去被删除文档的大小而得来的。
- ❑ index.merge.policy.segments_per_tier：该参数确定了每层允许出现的索引段数量的上限。越小的参数值会导致更少的索引段数量，这也意味着更多的合并操作以及更低的索引性能。默认值为 10，建议设置为大于等于 index.merge.policy.max_merge_at_once，否则你将遇到很多与索引合并以及性能相关的问题。
- ❑ index.reclaim_deletes_weight：该参数值默认为 2.0，它确定了索引合并操作中清除被删除文档这个因素的权重。如果该参数设置为 0.0，则清除被删除文档对索引合并没有影响。该值越高，则清除较多被删除文档的合并会更受合并策略青睐。
- ❑ index.compound_format：该参数类型为布尔型，它确定了索引是否存储为复合文件格式（compound format），默认值为 false。如果设置为 true，则 Lucene 会将所有文件存储在一个文件中。这样设置有时能解决操作系统打开文件处理器过多的问题，但是也会降低索引和搜索的性能。
- ❑ index.merge.async：该参数类型为布尔型，用来确定索引合并是否异步进行。默认为 true。
- ❑ index.merge.async_interval：当 index.merge.async 设置为 true（因此合并是异步进行的），该参数值确定了两次合并的时间间隔，默认值为 1s。请记住，为了触发真正的索引合并以及索引段数量缩减操作，该参数值应该保持为一个较小值。

配置 log byte size 合并策略

当采用 log_byte_size 合并策略时，可配置以下选项：

- ❑ merge_factor：该参数确定了索引期间索引段以多大的频率进行合并。该值越小，搜索的速度越快，消耗的内存也越少，而代价则是更慢的索引速度。如果该值越大，情形则正好相反，即更快的索引速度（因为索引合并更少），搜索速度更慢，消耗的内存更多。该参数默认为 10，对于批量索引构建，可以设置较大的值，对于日常索引维护则可采用默认值。
- ❑ min_merge_size：该参数定义了索引段可能的最小容量（段中所有文件的字节数）。如果索引段大小小于该参数值，且 merge_factor 参数值允许，则进行索引段合并。该参数默认值为 1.6MB，它对于避免产生大量小索引段是非常有用的。然而，用户应该记住，该参数值设置为较大值时，将会导致较高的合并成本。
- ❑ max_merge_size：该参数定义了允许参与合并的索引段的最大容量（以字节为单位）。默认情况下，参数不做设置，因而在索引合并时对索引段大小没有限制。
- ❑ maxMergeDocs：该参数定义了参与合并的索引段的最大文档数。默认情况下，参数没有设置，因此当索引合并时，对索引段没有最大文档数的限制。
- ❑ calibrate_size_by_deletes：该参数为布尔值，如果设置为 true，则段中被删除文档

的大小会用于索引段大小的计算。

- ❑ `index.compund_format`：该参数为布尔值，它确定了索引文件是否存储为复合文件格式，默认为 `false`。可参考 `tiered` 合并策略配置中该选项的解释。

配置 `log doc` 合并策略

当使用 `log_doc`（文档数对数）合并策略时，可配置以下这些选项：

- ❑ `merge_factor`：与 `log_byte_size` 合并策略中该参数的作用相同，请参考前面的解释。
- ❑ `min_merge_docs`：该参数定义了最小索引段允许的最小文档数。如果某索引段的文档数低于该参数值，且 `merge_factor` 参数允许，就会执行索引合并。该参数默认值为 1000，它对于避免产生大量小索引段是非常有用的。但是用户需要记住，将该参数值设置过大将增大索引合并的代价。
- ❑ `max_merge_docs`：该参数定义了可参与索引合并的索引段的最大文档数。默认情况下，该参数没有设置，因而对参与索引合并的索引段的最大文档数没有限制。
- ❑ `calibrate_size_by_deletes`：该参数为布尔值，如果设置为 `true`，则段中被删除文档的大小会在计算索引段大小时考虑进去。
- ❑ `index.compund_format`：该参数为布尔值，它确定了索引文件是否存储为复合文件格式，默认为 `false`。可参考 `tiered` 合并策略配置中该选项的解释。

与前面介绍的合并策略类似，上面提及的属性需要以 `index.merge.policy` 为前缀。例如，要设置 `min_merge_docs` 属性，则应该设置 `index.merge.policy.min_merge_docs` 属性。

除此之外，`log_doc` 合并策略支持 `index.merge.async` 和 `index.merge.async_interval` 属性，就像 `tiered` 合并策略那样。

3.6.3 调度

除了可以影响索引合并策略的行为之外，ElasticSearch 还允许我们定制合并策略的执行方式。索引合并调度器（scheduler）分为两种，默认的是并发合并调度器 `ConcurrentMergeScheduler`。

并发合并调度器

该调度器使用多线程执行索引合并操作，其具体过程是：每次开启一个新线程直到线程数达到上限，当达到线程数上限时，必须开启新线程（因为需要进行新的段合并），那么所有索引操作将被挂起，直到至少一个索引合并操作完成。

为了控制最大线程数，可以通过修改 `index.merge.scheduler.max_thread_count` 属性来实现。一般来说，可以按如下公式来计算允许的最大线程数：

```
maximum_value(1, minimum_value(3, available_processors / 2))
```

如果我们的系统是 8 核的，那么调度器允许的最大线程数可以设置为 4。

顺序合并调度器

该调度器非常简单，它使用同一个线程执行所有的索引合并操作。在执行合并时，该线程的其他文档处理都会被挂起，从而索引操作会延迟进行。

设置合并调度

为了设置特定的索引合并调度器，用户可将 `index.merge.scheduler.type` 的属性值设置为 `concurrent` 或 `serial`。例如，为了使用并发合并调度器，用户应该如此设置：

```
index.merge.scheduler.type: concurrent
```

如果想使用顺序合并调度器，用户则应该像下面这样设置：

```
index.merge.scheduler.type: serial
```

 在讨论索引合并策略和调度器时，可视化演示是再好不过的方式。如果你想了解在 Lucene 之中索引合并是如何执行的，不妨参阅 Mike McCandless 的博客：<http://blog.mikemccandless.com/2011/02/visualizing-lucenes-segment-merges.html>。
除此之外，也可以使用一个叫作 SegmentSpy 的插件。可参考这个 URL 中的内容：<https://github.com/polyfractal/elasticsearch-segmentspy>。

3.7 小结

在本章，我们学习了如何使用不同的评分公式以及它们带来的好处。同时也了解了不同的倒排索引格式及其优点。除此之外，还介绍了准实时搜索和实时读取以及 Searcher 刷新对 ElasticSearch 的意义。另外，还讨论了多语言数据处理和如何按需配置事务日志。最后介绍了索引的段合并、合并策略以及调度。

在下一章，我们将近距离观察 ElasticSearch 提供了哪些索引分片控制功能，也将了解如何为我们的索引选择合适的索引分片数和副本数，如何操纵索引分片的放置，以及了解何时创建多于我们实际所需的索引分片数。我们还将讨论索引分片的分配机制，并在最后使用之前所学知识创建容错并可扩展的集群。