# 实战 Elasticsearch、Logstash、Kibana

## ——分布式大数据搜索与日志挖掘及可视化解决方案

高 凯 编著

清华大学出版社

## 实战Elasticsearch、Logstash、Kibana
### ——分布式大数据搜索与日志挖掘及可视化解决方案

Elasticsearch是一个基于Lucene的开源分布式信息检索架构和全文搜索工具。构建在Elasticsearch基础上的日志处理工具Logstash和信息可视化组件Kibana能有效衔接并高效处理由Elasticsearch索引的分布式数据，三者优势互补，各司其职，共同完成从网络大数据分布式存储、倒排索引、全文检索、Web日志处理、挖掘结果可视化等一整套信息处理流程。本书将Elasticsearch、Logstash、Kibana联袂奉献给广大读者，可使读者尽快熟悉ELK架构，并构建自己的Web应用程序，完成分布式信息检索与分析工作。

——文益民博士，桂林电子科技大学教授

大数据时代，建立一个网站或应用程序，搜索功能是必备的。本书通过对大数据搜索、日志挖掘和可视化利器Elasticsearch、Logstash、Kibana相关知识的阐述，介绍了ELK架构在搜索领域中的实际应用方法。本书强调实践，内容新颖，条理清晰，组织合理，能让读者更好地了解到ELK架构的实现细节，可以给需要做类似产品的读者带来不小的帮助，值得推荐。

——邵华钢博士，上海京颐科技股份有限公司执行副总裁

ISBN 978-7-302-39984-1
9 787302 399841
定价：49.00元

# 实战 Elasticsearch、Logstash、Kibana

## ——分布式大数据搜索与日志挖掘及可视化解决方案

高 凯 编著

清华大学出版社
北 京

## 内 容 简 介

对大数据的搜索与挖掘，在当今网络时代是很有必要的。本书提出的分布式大数据搜索与日志挖掘及可视化解决方案是基于 Elasticsearch、Logstash 和 Kibana 而形成的，它能有效应对海量大数据所带来的分布式存储与处理、全文检索、日志挖掘、可视化等的挑战。构建在全文检索开源软件 Lucene 之上的 Elasticsearch，不仅能对海量规模的数据完成分布式索引与检索，还能提供数据聚合分析；Logstash 能有效处理来源于各种不同数据源的日志信息；Kibana 能得出可视化分析结果。本书讲解有关 Elasticsearch、Logstash、Kibana 的使用，相关内容以模块化的方式进行组织，注重实战，强调实践，内容新颖，组织合理。

本书可为高校相关专业（如计算机科学与技术、软件工程、情报学、图书馆学、信息管理与信息系统）学生的学习和科研工作提供帮助，同时对于从事大数据搜索与挖掘、信息检索与智能处理技术的工程技术人员和希望了解网络信息检索与分析技术的爱好者也具有较高的参考价值。

**本书封面贴有清华大学出版社防伪标签，无标签者不得销售。**
**版权所有，侵权必究。侵权举报电话：010-62782989　13701121933**

**图书在版编目(CIP)数据**

实战 Elasticsearch、Logstash、Kibana——分布式大数据搜索与日志挖掘及可视化解决方案/高凯编著.--北京：清华大学出版社，2015

ISBN 978-7-302-39984-1

Ⅰ. ①实…　Ⅱ. ①高…　Ⅲ. ①互联网络—情报检索　Ⅳ. ①G354.4

中国版本图书馆 CIP 数据核字(2015)第 086511 号

**责任编辑：**焦　虹　李　晔
**封面设计：**傅瑞学
**责任校对：**徐俊伟
**责任印制：**杨　艳

**出版发行：**清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　邮　　编：100084
社 总 机：010-62770175　　邮　　购：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质 量 反 馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
课 件 下 载：http://www.tup.com.cn，010-62795954

**印 装 者：**北京鑫海金澳胶印有限公司
**经　　销：**全国新华书店
**开　　本：**185mm×260mm　　**印　　张：**15.25　　**字　　数：**371 千字
**版　　次：**2015 年 6 月第 1 版　　**印　　次：**2015 年 6 月第 1 次印刷
**印　　数：**1～2000
**定　　价：**49.00 元

产品编号：062546-01

Preface

# 序言

云计算、智慧城市、移动互联网、大数据与物联网已经成为大数据时代的前瞻技术，实现了人、机器与实物的多维互联互通，监测数据、内容数据、社交数据、关系数据裂变式增长，大数据时代全方位地到来。大数据具有多（体量大）、快（生成速度快）、好（价值大）、省（高效）的特征，传统的信息搜索、数据挖掘与知识呈现理论技术难以满足当下多样化的需求。大数据的理念与理论已经成为了人所共知的科学常识，但是大数据搜索、挖掘与可视化等落地的工程实践尚有较大距离，也是当下的工程急需。

本书从分布式大数据搜索、日志挖掘与可视化三个角度出发，以非结构化文本信息、半结构化的日志数据为处理对象，进行宏观解决方案与微观方法技巧全面阐释。具体地说，如何利用在全文检索开源软件 Lucene 之上的 Elasticsearch 对大数据进行分布式计算与全文检索；如何利用 Logstash 对日志文件智能分析与处理；如何利用 Web 接口 Kibana 对日志进行高效的搜索、可视化、分析等各种操作是，是本书的论述重点。

从工程实践的角度掌握 ElasticSearch、Logstash、Kibana 的基本使用方法和技巧，很有必要。目前，国内专门针对 ElasticSearch、Logstash、Kibana 进行介绍的书很少，本书是目前国内较早的一本综合介绍 ELK 架构的编著，涉及范围广泛，内容新颖，条理清晰，组织合理。

高凯老师是我多年的朋友，我们都在大数据搜索与挖掘方向上从事教学、科研与开发工作。高凯博士严谨的治学态度、理论联系实际的做法以及敬业的态度也一直为我所学习。非常荣幸能够有这个机会来为高老师的新著作序，认真拜读后，我以为本书实战性很强，是大数据搜索与挖掘所需的上乘之作，是大数据“知著、见微、晓意”的必备工具，值得推荐！

2015.5.5

（张华平博士，副教授，北京理工大学大数据搜索挖掘实验室主任，ICTCLAS 及 NLPIR 分词软件发明者）

Foreword

建立在分布式系统之上的大数据搜索与挖掘应用，是当今 IT 业的研究与工程实践热点之一。在 DB-Engines 公布的 2015 年度最受欢迎的数据库系统中，Elasticsearch 名列前茅。作为开源分布式检索与数据处理平台，Elasticsearch 不仅仅是一个数据库，它还是一个基于 Lucene 构建的开源、分布式、RESTful 信息检索框架。基于 Elasticsearch＋Logstash＋Kibana 的信息处理架构，为编程人员提供了一种分布式可扩展的信息存储和全文检索机制以及基于 Logstash 的日志处理机制、基于 Kibana 的挖掘结果可视化机制。它不仅能对海量规模的数据完成分布式索引与检索，还能提供数据聚合分析和可视化。因此，从实战的角度掌握 Elasticsearch、Logstash、Kibana 的基本使用方法和技巧，很有必要。

大数据这个术语的出现，大概可追溯到基于 Lucene 的 Apache 开源项目 Nutch。从 2009 年开始，大数据开始成为互联网行业的流行词汇，也吸引了越来越多的关注。物联网、云计算、移动互联网、手机与平板电脑、PC 以及遍布各个角落的各种各样的传感器，无一不是大数据的来源方或承载方。可以说，大数据就在我们身边。从阿里巴巴、1 号店、京东商城等电子商务数据，到 QQ 等即时聊天内容，再到 Google、Bing、百度，又到社会网络与微博、微信等，都在生产、承载着大数据。随着信息处理量的增大，对大数据的分布式存储、快速搜索与挖掘显得特别必要。例如，挖掘用户的行为习惯和喜好，从凌乱纷繁的大数据背后找到符合用户兴趣和习惯的产品和服务，并对产品和服务进行有针对性的调整和优化，本身就蕴含着巨大的商机。但是，传统的基于关系型数据库管理系统的方法，在高效处理大数据时显得有些力不从心。虽然开源的全文检索工具 Lucene 能处理非结构化和半结构化的信息，但其某些版本在分布式处理方面的不足限制了它在大数据方面的应用。我们希望找到一个快速的分布式信息检索解决方案，希望它是一个零配置和易于上手的全文检索模式，希望它能够简单地使用 JSON 通过 HTTP 索引数据，更希望它支持分布式处理并支持系统扩展，能够实时搜索，并且稳定、可靠。

Elasticsearch 是一个基于 Lucene 的开源分布式信息检索架构和全文搜索工具。构建在 Elasticsearch 基础上的日志处理工具 Logstash 和信息可视化组件 Kibana，能有效衔接并高效处理由 Elasticsearch 索引的分布式数据，三者优势互补，各司其职，共同完成网络大数据分布式存储、倒排索引、全文检索、Web 日志处理、挖掘结果可视化这一整套的信息处理流程。目前，国内这方面的资料很少，仅有的几部译著所提及的 Elasticsearch 版本较低，且没有任何有关 Logstash 和 Kibana 的书籍。因此，我们萌发了一个想法，将 Elasticsearch、Logstash、Kibana(统称为 ELK)联袂奉献给广大软件开发者，帮助他们尽快熟悉 ELK 架构，并构建自己的 Web 应用程序，完成对分布式信息的检索与分析工作。

本书强调实践，内容新颖，条理清晰，组织合理。通过实战讲解的方式，让读者更好地了解 ELK 架构的实现细节。全书内容涵盖 ELK 简介、文档索引与处理、信息检索与过滤、信息统计与分析、基于 Java 客户端的 Elasticsearch 功能实现、Elasticsearch 配置与管理、基于 Logstash 的网络日志处理、基于 Kibana 的分析结果可视化、应用实例等多个部分。

全书由高凯提出写作大纲，第 1 章和第 6 章由高凯撰写并完成全书通稿和审校工作，其余各章均由高莘撰写，其中，第 1 章概述 Elasticsearch、Logstash、Kibana 的主要功能，对涉及到的一些概念进行简介，并从实用的角度出发，通过对实例的讲解，介绍索引、检索的实现机制；第 2 章对 Elasticsearch 中的索引、映射等进行说明；第 3 章介绍 Elasticsearch 中的检索功能；第 4 章介绍基于 Facets、Aggregations 的数据聚合与统计功能；第 5 章从工程实践的角度，介绍面向 Java 客户端的 Elasticsearch 部分功能的设计与实现；第 6 章介绍 Elasticsearch 的配置及一些高级功能、监控等的使用；第 7 章介绍日志处理及 Logstash 的应用；第 8 章介绍基于 Kibana 的可视化技术；第 9 章给出一个综合应用实例，该实例从网页采集、处理、存储、索引、日志处理、可视化展示等入手，介绍了基于 ELK 的分布式信息检索与日志挖掘解决方案。

本书的顺利完成也得益于参阅了大量的相关工作及研究成果，部分内容源自 Elasticsearch、Logstash、Kibana 的官方文档。在写作过程中，也参考了相关文献和互联网上众多热心网友提供的素材，在此谨向这些文献的作者、热心网友以及为本书提供帮助的老师，特别是那些由于篇幅所限未及在参考文献中提及的相关文献的作者和网站，致以诚挚的谢意和崇高的敬意。

由于我们的学识、水平均有限，书中不妥之处在所难免，恳请广大读者批评指正。

编　者

2015 年 5 月

Contents

# 目录


**第1章　概述** ………………………………………………………… 1

1.1　Elasticsearch 的安装与简单配置 ……………………………… 2

1.2　走近 Elasticsearch ……………………………………………… 6

1.2.1　Elasticsearch 是什么 ……………………………………… 6

1.2.2　Elasticsearch 中涉及到的相关概念 …………………………… 7

1.2.3　Elasticsearch API 的简单使用方式 …………………………… 9

1.2.4　Elasticsearch RTF 版本中的部分插件简介 ……………… 10

1.2.5　Elasticsearch 基本架构 ……………………………………… 12

1.3　Elasticsearch 索引及其构建 ……………………………………… 13

1.3.1　概述 ………………………………………………………… 13

1.3.2　借助 Head 工具构建索引 ……………………………………… 13

1.3.3　Mapping 简述 ……………………………………………… 15

1.4　信息检索及其构建 ……………………………………………… 15

1.5　实例 …………………………………………………………… 16

1.6　扩展知识与阅读 ……………………………………………… 21

1.7　本章小结 ……………………………………………………… 22

**第2章　文档索引及管理** ……………………………………………… 23

2.1　文档索引概述 ………………………………………………… 23

2.2　建立索引 ……………………………………………………… 24

2.3　通过映像 Mapping 配置索引 …………………………………… 28

2.3.1　在索引中使用映像 ………………………………………… 28

2.3.2　管理/配置映像 …………………………………………… 29

2.3.3　获取映像信息 ……………………………………………… 30

2.3.4　删除映像 …………………………………………………… 31

2.4　管理索引文件 ………………………………………………… 31

2.4.1　打开、关闭、检测、删除索引文件 ………………………… 31

2.4.2　清空索引缓存 ……………………………………………… 32

2.4.3　刷新索引数据 ……………………………………………… 32

2.4.4 优化索引数据 …… 32
2.4.5 Flush 操作 …… 33
2.5 设置中文分词器 …… 33
2.6 对文档的其他操作 …… 34
2.6.1 获取指定的文档信息 …… 34
2.6.2 删除文档中的信息 …… 36
2.6.3 数据更新 …… 36
2.6.4 基于 POST 方式批量获取文档 …… 39
2.6.5 删除部分文档 …… 40
2.7 扩展知识与阅读 …… 40
2.8 本章小结 …… 41

**第3章 信息检索与结果过滤 …… 42**

3.1 实验数据集描述 …… 42
3.2 简单检索 …… 44
3.3 基本检索 …… 45
3.3.1 设置不同字段的排序权重 …… 45
3.3.2 指定返回的字段子集 …… 46
3.3.3 Term 查询、Terms 查询、Wildcard 通配符查询 …… 48
3.3.4 Match、Match_all、Match_phrase 查询 …… 49
3.3.5 Query_string 查询 …… 50
3.3.6 Prefix、Range 查询 …… 51
3.3.7 More_like_this、Fuzzy_like_this 查询 …… 52
3.3.8 跨字段检索 …… 54
3.4 Filter 概述 …… 54
3.5 常用 Filter 及其应用 …… 56
3.5.1 And Filter 及 Or Filter …… 56
3.5.2 Bool Filter …… 57
3.5.3 Exists Filter 和 Missing Filter …… 57
3.5.4 Type Filter …… 58
3.5.5 Match_all Filter …… 58
3.5.6 Not Filter …… 59
3.5.7 Query Filter …… 59
3.6 复合查询 …… 60
3.7 结果排序 …… 62
3.8 扩展知识与阅读 …… 63
3.9 本章小结 …… 63

**第 4 章 信息统计分析与搜索提示** ………… 64

4.1 Facets 概述 ………… 64
4.2 各种不同的 Facets 统计 ………… 66
4.2.1 Terms Facets：指定字段的分布情况统计 ………… 66
4.2.2 Range Facets：在某个范围的分布情况统计 ………… 70
4.2.3 Histogram Facets ………… 72
4.2.4 Date_histogram Facets ………… 75
4.2.5 Statistical Facets ………… 77
4.2.6 Terms_stats Facets ………… 79
4.3 Aggregations ………… 80
4.3.1 概述 ………… 80
4.3.2 最值、求和、均值统计 ………… 82
4.3.3 Stats Aggregations 及 Extended Stats Aggregations ………… 84
4.3.4 Terms Aggregations ………… 85
4.3.5 Range Aggregations ………… 89
4.3.6 Date_range Aggregations ………… 92
4.3.7 Histogram Aggregations ………… 93
4.3.8 Date_histogram Aggregations ………… 96
4.3.9 Filter Aggregations ………… 98
4.3.10 Missing Aggregations ………… 101
4.4 搜索提示 ………… 101
4.5 扩展知识与阅读 ………… 102
4.6 本章小结 ………… 102

**第 5 章 Elasticsearch 部分功能的 Java 客户端实现** ………… 103

5.1 Elasticsearch 节点实例化 ………… 103
5.1.1 通过 Maven 添加对 Elasticsearch 依赖 ………… 103
5.1.2 初始化 Elasticsearch Client ………… 105
5.2 索引数据 ………… 107
5.2.1 准备 JSON 数据 ………… 107
5.2.2 索引 JSON 数据 ………… 108
5.3 对索引文档的操作 ………… 110
5.3.1 获取索引文档 ………… 110
5.3.2 删除索引文档 ………… 111
5.3.3 更新索引文档 ………… 112

5.3.4 批量操作索引文件 …… 113
5.3.5 简单的统计操作 …… 113
5.4 信息检索 …… 114
5.4.1 概述 …… 114
5.4.2 MultiSearch …… 115
5.4.3 Query DSL 概述 …… 116
5.4.4 MatchQuery …… 117
5.4.5 MatchAllQuery …… 117
5.4.6 MultiMatchQuery …… 118
5.4.7 BoolQuery …… 118
5.4.8 TermQuery …… 120
5.4.9 WildcardQuery …… 120
5.4.10 QueryString …… 121
5.4.11 MoreLikeThis …… 121
5.4.12 Filter 概述 …… 123
5.4.13 TermFilter …… 123
5.4.14 ExistsFilter …… 123
5.4.15 MatchAllFilter …… 124
5.4.16 QueryFilter …… 124
5.4.17 RangeFilter …… 125
5.4.18 TypeFilter …… 126
5.4.19 过滤器间的组合：BoolFilter、NotFilter、OrFilter、AndFilter …… 126
5.5 统计分析 …… 127
5.5.1 Facets …… 127
5.5.2 Aggregations …… 129
5.6 对检索结果的进一步处理 …… 130
5.6.1 控制每页的显示数量及显示排序依据 …… 130
5.6.2 基于 Scroll 方法的检索结果及其分页 …… 131
5.6.3 高亮显示检索词 …… 133
5.7 扩展知识与阅读 …… 134
5.8 本章小结 …… 134

**第6章 Elasticsearch 配置与集群管理 …… 136**

6.1 Elasticsearch 部分基本配置及其说明 …… 136
6.2 提高索引和查询效率的策略 …… 139
6.3 监控集群状态 …… 140

6.4 控制索引分片与副本分配 …… 143
6.5 扩展知识与阅读 …… 144
6.6 本章小结 …… 144

**第7章 基于 Logstash 的日志处理 …… 145**

7.1 概述 …… 145
7.2 Input：处理输入的日志数据 …… 148
7.2.1 处理基于 File 方式输入的日志信息 …… 148
7.2.2 处理基于 Generator 产生的日志信息 …… 149
7.2.3 处理基于 Log4j 的日志信息 …… 150
7.2.4 处理基于 Redis 的日志信息 …… 151
7.2.5 处理基于 Stdin 方式输入的信息 …… 154
7.2.6 处理基于 TCP 传输的日志数据 …… 154
7.2.7 处理基于 UDP 传输的日志数据 …… 157
7.3 Codecs：格式化日志数据 …… 159
7.3.1 JSON 格式 …… 159
7.3.2 Rubydebug 格式 …… 161
7.3.3 Plain 格式 …… 162
7.4 基于 Filter 的日志处理与转换 …… 162
7.4.1 JSON Filter …… 163
7.4.2 Grok Filter …… 164
7.4.3 Kv Filter …… 166
7.5 Output：处理输出的日志数据 …… 167
7.5.1 将处理后的日志输出到 Elasticsearch 中 …… 168
7.5.2 将处理后的日志输出至文件中 …… 169
7.5.3 将处理后的部分日志输出到 csv 格式的文件中 …… 170
7.5.4 将处理后的日志输出到 redis 中 …… 171
7.5.5 将处理后的部分日志通过 UDP 协议输出 …… 173
7.5.6 将处理后的部分日志通过 TCP 协议输出 …… 175
7.5.7 将收集到的日志信息传输到自定义的 HTTP 接口中 …… 178
7.6 扩展知识与阅读 …… 178
7.7 本章小结 …… 178

**第8章 基于 Kibana 的数据分析可视化 …… 180**

8.1 安装 Kibana …… 181
8.2 Kibana 概述 …… 182

8.2.1 在仪表盘上添加新行 …… 183
8.2.2 在行中添加新面板 …… 183
8.2.3 设置 Query 和 Filtering …… 185
8.3 常用面板类型 …… 187
8.3.1 Histogram …… 187
8.3.2 Table …… 189
8.3.3 Map 和 Bettermap …… 190
8.3.4 Terms …… 191
8.3.5 Text …… 192
8.3.6 Sparklines …… 193
8.3.7 Trends …… 194
8.4 网站性能监控可视化应用的设计与实现 …… 195
8.4.1 概述 …… 195
8.4.2 Page View …… 196
8.4.3 响应/请求时间 …… 197
8.4.4 流量走势与统计 …… 198
8.4.5 状态码监控 …… 200
8.4.6 UA 行 …… 203
8.5 Kibana V4 简介 …… 205
8.5.1 新建视图 …… 205
8.5.2 建立 Dashboard …… 207
8.5.3 配置 …… 208
8.6 扩展知识与阅读 …… 208
8.7 本章小结 …… 209

**第9章 网络信息检索与分析实践 …… 210**

9.1 信息采集 …… 210
9.2 基于 Python 的信息检索及 Web 端设计 …… 214
9.2.1 安装 Python 及 Django …… 214
9.2.2 安装 Elasticsearch 的 Python 插件 …… 215
9.2.3 Web 页面设计 …… 216
9.3 基于 Logstash 的日志处理 …… 219
9.3.1 安装和配置 Nginx …… 219
9.3.2 设计面向日志文件的 Pattern …… 220
9.3.3 在 Logstash 中进行相关配置 …… 220
9.4 基于 Kibana 的日志分析结果可视化设计与实现 …… 222

9.4.1 图表1：状态码走势分析 …… 222
9.4.2 图表2：查询词分析 …… 224
9.4.3 图表3：分析各状态码随时间的变迁情况 …… 224
9.4.4 集成上述图表 …… 226
9.5 扩展知识与阅读 …… 226
9.6 本章小结 …… 227
参考文献 …… 228

# 第1章

# 概　述

> "Elasticsearch is a flexible and powerful open source, distributed, real-time search and analytics engine. Elasticsearch gives you the ability to move easily beyond simple full-text search.
>
> Logstash helps you take logs and other time based event data from any system and store it in a single place for additional transformation and processing. Logstash will scrub your logs and parse all data sources into an easy to read JSON format.
>
> Kibana is Elasticsearch's data visualization engine, allowing you to natively interact with all your data in Elasticsearch via custom dashboards."
>
> ——http://www.elasticsearch.org/overview/

随着大数据、大型综合网站以及 Web 2.0 技术的普及，越来越多的软件开发者需处理海量异构信息的索引、检索、日志挖掘、可视化等和信息检索与大数据挖掘相关的业务。虽然 Lucene 是许多互联网公司的标准信息检索工具，但它无法在一个合理的时间内存储和检索海量的大数据，不具备良好的可扩展性，一般也不适合分布式大数据搜索、挖掘和云计算环境。而在 DB-Engines 公布的 2015 年 5 月份最受欢迎的数据库系统（含 NoSQL 数据库）中，Elasticsearch 名列前茅[Db-engines, 2015]。

作为开源分布式搜索与数据处理平台，Elasticsearch 不仅仅是一个数据库，它还是一个基于 Lucene 构建的开源、分布式、RESTful 信息检索框架，能够实时搜索，并且稳定、可靠，使用方便，支持通过 HTTP 使用 JSON 进行数据索引。基于 ELK（注：本书中的 ELK 是指 Elasticsearch＋Logstash＋Kibana，后文同）的架构为编程人员提供了一个分布式的可扩展的信息存储和基于 Lucene 的信息检索机制、基于 Logstash 的日志处理机制、基于 Kibana 的挖掘结果可视化的机制。在一个典型的使用场景，一般用 Elasticsearch 作为后台数据的分布式存储和全文检索，Kibana 用来前端的可视化展示，Logstash 在其过程中担任相关日志加工和“搬运工”的角色。ELK 架构为数据分布式存储、可视化查询和日志解析创建了一个功能强大的管道链。三者互相配合，取长补短，共同完成分布式大数据处理工作。

首先，Elasticsearch是一个开源的分布式信息检索框架，具备高可靠性，它提供多种管理工具，各种相关插件也可方便地集成到Elasticsearch中。它对外提供一系列基于Java和HTTP的API，可用于分布式索引、检索、日志分析与数据挖掘等，且大多数配置是可以修改的。因此，很多国际知名企业都在使用Elasticsearch完成分布式数据处理工作。例如，Github已升级了其代码搜索程序，并将核心架构由Solr转向Elasticsearch；Wikimedia也启用了由Elasticsearch为基础的全新搜索框架。

其次，Logstash可以对相关的网络日志进行收集、分析、转换等处理工作，并将其存储在Elasticsearch供以后使用。其实，Logstash本身并不产生日志，它仅仅是一个可接收多种多样的日志输入、经处理后转发到多个不同目的地的“管道”。最后，Kibana可以帮助可视化数据日志，并提供友好的可视化界面。

学习ELK，对于大数据处理、信息检索及搜索引擎研发、日志处理与分析、挖掘信息可视化等，对于设计高效的大型商业网站，都具有重要的现实意义。本书主要介绍Elasticsearch分布式信息存储与检索解决方案，并结合Logstash、Kibana，介绍面向大数据搜索与挖掘的处理方法。作为全书的“引子”，本章介绍Elasticsearch的背景和简单使用，并通过一个例子介绍Elasticsearch的索引、检索流程和实现方法。有关Logstash和Kibana的相关内容，在本书后续章节中进行介绍。

## 1.1 Elasticsearch的安装与简单配置

“工欲善其事，必先利其器。”要想了解Elasticsearch，要从该软件的安装入手。Elasticsearch的安装非常简单，几乎是“开箱即用”的。当然，前提是需要先下载JDK，并配置相应的环境变量，同时确保系统可用内存大于2GB。

> Tips：建议使用JDK7或JDK8，低版本的JDK会对Elasticsearch的使用造成不利影响；建议设置JAVA_HOME环境变量。

下面对Elasticsearch的安装进行说明。进入Elasticsearch官网http://www.elasticsearch.org，找到对应的Elasticsearch软件版本下载。如果是在Windows系统下使用，可以下载ZIP格式的安装包。对于工程开发人员来说，往往需要在Elasticsearch软件中集成一些其他插件和工具等。因此，建议初学者可以首先从Elasticsearch的RTF版本入手。可以到https://github.com/medcl/elasticsearch-rtf去下载Elasticsearch的RTF版本。

> Tips：RTF是Ready To Fly的缩写，这是一个集成了基本插件（如服务封装、中文分词、mapper-attachments、transport-thrift、tools.carrot2等插件）的并带有示例程序的可直接上手的简易工程版本。

解压后会看到其目录结构。Elasticsearch 包含的主要文件夹及功能如下(以 RTF 版为例):

- bin——含有运行 Elasticsearch 实例和管理插件的一些脚本。
- config——主要是一些设置文件,如:elasticsearch. yml 和 logging. yml 等。对 elasticsearch. yml 和 logging. yml 文件中相关配置的说明,可参阅本书后续章节。
- lib——包含一些相关的包文件等。
- plugins——包含相关的插件文件等。
- logs——日志文件。
- data——Elasticsearch 中存放数据的地方。
- works——临时文件。

Tips:如果 Elasticsearch 运行在专用服务器上,一般经验是分配一定的内存给 Elasticsearch,可以通过修改 ES_HEAP_SIZE 环境变量来改变这个设定,它控制堆大小。在启动 Elasticsearch 之前应该把这个变量改到预期值。关于这个数据的设置可参见相关手册。一般来说,如果在日志文件中发现带有 OutOfMemoryError 错误的输出记录,则应考虑将环境变量 ES_HEAP_SIZE 取值调大,建议该值不应超过总可用物理内存的 50%(剩余内存可用作高速缓存,提高检索性能),这样可以极大地提高搜索性能[Rafa,2015]。

可选择 Elasticsearch 使用的中文分词器。打开 config / elasticsearch. yml 文件(注:yml 是一种简单的数据描述语言,语法比 XML 简单,适合用来表达或编辑数据结构,并完成各种设定等)。在这个文件中的 index. analysis. analyzer. default. type 部分指定使用的中文分词器。代码段 1. 1 是选择使用 IK 分词器并对其进行设置(注:在 Elasticsearch、Logstash、Kibana 等相关的配置文件中用#表示注释信息。为便于统一表述形式,本书多用//表示注释说明信息,仅在第 7 章有时用#表示注释说明信息)。

```
//代码段 1.1: 在 elasticsearch.yml 中设置中文分词算法
index:
  analysis:
    analyzer:
      ik:
        alias:
        -ik_analyzer
        type: org.elasticsearch.index.analysis.IkAnalyzerProvider
      ik_max_word:
        type: ik
        use_smart: false
      ik_smart:
        type: ik
        use_smart: true
index.analysis.analyzer.ik.type: ik
index.analysis.analyzer.default.type: ik
```

> Tips：在文本被索引前需要经过分词处理，这项工作一般由 Analyzer 类完成。Analyzer 类是个抽象类，对应不同语言的文本，应该从 Analyzer 派生出特定的 Analyzer。IK Analyzer 是一个开源的基于 Java 语言开发的轻量级的中文分词工具包。最初它是以开源项目 Lucene 为应用主体的、结合词典分词和文法分析算法的中文分词组件。从 IK Analyzer 3.0 起，发展为面向 Java 的公用分词组件，且独立于 Lucene 项目。另外，yml 配置文件多数内容是被#注释起来的，需要修改某部分时，可以删掉注释标记并进行相关的配置即可。yml 内容要求严格执行规定的缩进格式。

进入 Elasticsearch 的 bin 文件夹，运行 elasticsearch.bat 文件，启动 Elasticsearch。

> Tips：若关闭 Elasticsearch，可在 shell 环境输入命令（Elasticsearch 默认占用 9200 端口）：

```
curl -xPOST http://localhost:9200/_cluster/nodes/_shutdown
```

之后，打开浏览器，输入 http://locaohost:9200，会显示类似图 1.1 的内容。其中：

- status——发出请求后的 HTTP 的状态代码，显示“200”表示正常。
- name——Elasticsearch 实例的名字，默认情况下它将从名字列表中随机选择一个。其设置同样是在 config / elasticsearch.yml 文件中。
- version——版本号，以 JSON 格式表示了一组信息，其中的 number 字段代表了当前运行 Elasticsearch 的版本号，build_snapshot 字段代表了当前运行的版本是否是从源代码构建而来，lucene_version 表示 Elasticsearch 所基于的 Lucene 的版本（图 1.1 显示该版本是基于 Lucene4.10.2 而构建的）。
- tagline——包含了 Elasticsearch 的第一个 tagline："You Know，for Search."。

localhost:9200

```
{
  "status" : 200,
  "name" : "Gremlin",
  "cluster_name" : "gsElasticSearch",
  "version" : {
    "number" : "1.4.0",
    "build_hash" : "bc94bd81298f81c656893ab1ddddd30a99356066",
    "build_timestamp" : "2014-11-05T14:26:12Z",
    "build_snapshot" : false,
    "lucene_version" : "4.10.2"
  },
  "tagline" : "You Know, for Search"
}
```

图 1.1 Elasticsearch 启动后的界面

图 1.1 中出现了 JSON 格式的数据。JSON（JavaScript Object Notation）是基于 Javascript 的轻量级的数据交换格式，是独立于语言的文本格式。在 Javascript 中，处理 JSON 数据不需要任何特殊的 API 或工具包。利用 JSON 可简单地表示半结构化数据，而

且目前多数编程语言支持对 JSON 数据的解析。JSON 的基本语法表示是：

- 数据在用双引号表示的名称/值对中，中间用冒号隔开，如："Name"："Smith"。
- 可创建包含多个"名称 / 值对"的记录，如：{"Name"："Smith"，"email"："abc@sjtu.org"}等，它表示以上两个值是同一记录的一部分，数据由逗号分隔，花括号保存对象，方括号保存数组。

如下的代码是对同样数据的 XML 和 JSON 表示形式。JSON 和 XML 的可读性可谓不相上下，但在 Javascript 范围内，JSON 优势要大于 XML。在 Elasticsearch 应用中，可以在很多地方看到 JSON 的身影。

XML 示例代码：

```
<?xml version="1.0" encoding="utf-8"?>
<book>
    <name>Elasticsearch Searching</name>
    <author>
        <name>Gao</name>
        <sex>male</sex>
        <age>45</age>
        <country>China</country>
    </author>
    <price>10</peice>
</book>
```

JSON 示例代码：

```
{
    "book": {
        "name": " Elasticsearch Searching ",
        "author": {
            "name": "Gao",
            "sex": "male",
            "age": 45,
            "country": "China"
        },
      "price": 10,
    }
}
```

Tips：Elasticsearch 的配置文件中，elasticsearch.yml 负责设置服务器的默认配置；logging.yml 定义了多少信息写入系统日志、定义日志文件，并定期创建新文件，当需要适配监视环境或备份解决方案，抑或在系统调试时，可能需要更改该文件。

## 1.2 走近 Elasticsearch

构建在全文检索开源软件 Lucene 之上的 Elasticsearch,其实已经蜕变为一个完整的分布式大数据分析平台。它能对海量规模数据完成实时分析,完成分布式索引与检索,提供数据聚合功能,应用非常广泛,图 1.2 是部分采用 Elasticsearch 作为其搜索业务基础的公司。下面是国外一些著名企业使用 Elasticsearch 的例子[CSDN,2014]。

图 1.2 使用 Elasticsearch 的部分知名 IT 企业

- Github: Git 是一个分布式的版本控制系统,Github 使用 Elasticsearch 搜索数以几十亿的文件和上千亿行的代码。
- Foursquare: Foursquare 是一家基于用户地理位置信息的手机服务网站,鼓励手机用户同他人分享自己当前所在地理位置等信息。Foursquare 使用 Elasticsearch 做地点信息搜索。
- SoundCloud: SoundCloud 是一家德国网站,提供音乐分享社区服务。SoundCloud 使用 Elasticsearch 为数以几亿计的用户提供即时精准的音乐搜索服务。
- Sony: Sony 公司使用 Elasticsearch 作为信息搜索引擎。
- StackOverflow: 将全文搜索与地理位置和相关信息进行结合,以提供 more-like-this 相关问题的展现。
- 每天,Goldman Sachs 使用 Elasticsearch 来处理 5TB 数据的索引,还有很多投行使用 Elasticsearch 来分析股票市场的变动。

### 1.2.1 Elasticsearch 是什么

Elasticsearch 是由 Shay Banon 发起的一个开源搜索服务项目,并于 2010 年 2 月公开了源码。Elasticsearch 不仅仅能提供全文检索功能,它还能提供高效的分布式数据存储、索引、搜索,能完成对大数据的自动分片、自动负载索引,并提供 RESTful Web 的风格接口。

Tips: REST(REpresentational State Transfer)意即"表现层状态转化",RESTful是目前流行的一种互联网软件架构,具有结构清晰、符合标准、易于理解、扩展方便等特点,这种架构下的每一个URI代表一种资源,客户端通过GET(获取资源)、POST(新建或更新资源)、PUT(更新资源)、DELETE(删除资源)方式来对服务器端资源进行操作。

Elasticsearch是面向文档型的NoSQL数据库,可以在Elasticsearch中索引、搜索、排序和过滤这些文档。它不仅继承了Lucene的简洁语法表述,同时还可以进行以下工作:

- 分布式实时文件存储,可将每一个字段都编入索引,使其可以被检索到。
- 实时分析的分布式搜索引擎。
- 可以扩展到上百台服务器,处理PB级别的结构化或非结构化数据(当然它也可以运行在单台PC上)。

另外,Elasticsearch支持插件机制,如与mongoDB、couchDB同步的River插件、中文分词插件、Hadoop插件、脚本支持插件等。在使用Elasticsearch时,有很多基础服务可以用插件的方式来提供。

其实,Elasticsearch的优点很多,比如它执行搜索的速度更快,可以简单地通过HTTP方式,使用JSON来操作数据,并支持对分布式集群的搜索。这也是很多Lucene用户在面对大数据时转而使用Elasticsearch的原因之一。Elasticsearch的一个主要的特点是对分布式的支持,其索引能分拆为多个分片,每个分片可有零或多个副本,集群中的每个数据节点都可承载一个或者多分片,并且能协调和处理各种操作;负载再平衡(Rebalancing)和路由(Routing)在大多数情况下都是自动完成的。

### 1.2.2 Elasticsearch中涉及到的相关概念

参照文献[百度,2014],这里给出有关Elasticsearch中相关概念的解释。

(1) Cluster和Node——Elasticsearch中的Cluster是对外提供搜索服务的集群,组成这个Cluster的各个节点叫做Node。节点Node是Elasticsearch运行的实例;集群Cluster是一组有着同样cluster.name的节点,它们协同工作,互相分享数据,提供了故障转移和扩展的功能。Node又可分为IndexNode(提供读写)、DataNode(只提供数据存储和访问,负载均衡)等。节点之间是对等关系的(去中心化),而弱化的Master节点只不过多了维护集群状态的功能。每个节点上面的集群状态数据都是实时同步的。如果Master节点出故障,按照预定的程序,其他一台Node机器会被选举成为新的Master。从这点来看,它就比某些大数据处理平台的鲁棒性强[Fu, 2015]。

(2) Shards——Elasticsearch将一个完整的索引分成若干部分,每个部分就是一个Shards,每个Shard实际上就是一个基于Lucene的索引。Shards存储在相同或不同的Node上;Shards的数量一般在索引创建前指定,且索引创建后不能更改(其初始配置也是可以修改的)。检索时,Elasticsearch会将查询发送到不同的Shards上并将返回结果合并,这个过程对用户来说是透明的。

(3) Replicas——Replicas是索引的冗余备份，可用于防止数据丢失或用来做负载均衡。一般地，Elasticsearch会自动对搜索请求进行负载均衡。

(4) Recovery——在有节点加入或退出集群Cluster或故障节点重新启动时，Elasticsearch会根据机器的负载情况，对索引分片Shards进行重新分配。

(5) River——River是一个运行在Elasticsearch集群内部的插件，主要用来从外部获取异构数据，然后在Elasticsearch里创建索引。常见的有couchD Briver Plugin、RabbitMQ river Plugin、Twitter river Plugin、Wikipedia river Plugin、MongoDB river Plugin、JDBC river Plugin等。

(6) Gateway——是Elasticsearch索引数据快照的存储方式，当Elasticsearch集群关闭再重新启动时，就会从Gateway中读取索引数据快照。Elasticsearch支持多种类型的Gateway，有本地文件系统（像普通的Lucene索引一样，这也是默认方式）、分布式文件系统（如freeds）、Hadoop的HDFS和Amazon的S3云存储服务等。

> Tips：Gateway与workDir的区别：Gateway存储完整的索引数据，workDir对外提供相应查询操作；Gateway可以是本地文件系统、共享文件系统或HDFS等云存储，workDir可以是内存、本地文件系统或两者结合；Gateway被假设是可靠的，持久化的数据存储，workDir被假设是不安全的运行环境，数据允许丢失。

(7) Discovery. zen——Discovery. zen代表Elasticsearch的自动发现节点机制。Zen用来实现节点自动发现和Master节点选举，Master节点负责处理节点的加入和退出以及分片Shard的重新分配。Master不是固定不变的，当前Master出故障后，其他节点自动选举产生新的Master。只有当节点准备就绪以后，该节点才会被通知可以被使用。在config/elasticsearch. yml中可以进行相应参数的设置。

> Tips：Elasticsearch是一个基于P2P的系统，它通过广播机制寻找存在的节点，再通过多播协议来进行节点间的通信，同时也支持点对点的交互。因此，需要节点发现与Master选举机制。

(8) Transport——Transport代表Elasticsearch内部节点或集群与客户端的交互方式，默认内部是使用TCP协议进行交互，同时它支持HTTP协议（JSON格式）、Thrift、Servlet、Memcached、ZeroMQ等的传输协议（通过插件方式集成）。

(9) Index、Type、Document、Field——Index是Elasticsearch存储数据的地方，可以快速高效地对索引中的数据进行全文索引，类似于RDBMS数据库中的Database；在Index下一般会有多个存放数据的Type，Type类似于Database中的Table，用来存放具体数据；Document是类似关系数据库的一行数据，在一个Type里的每一Document都有一个唯一的ID作为区分，这里与RDBMS不同的是，Document不需要有固定的结构，不同文档可以

具有不同的字段集合;Field 类似关系数据库的某一列,是 Elasticsearch 数据存储的最小单位。

为了更好地理解 Elasticsearch 中的相应概念,文献[吴晓刚,2015]中给出相关概念在传统的关系型数据库(如 SQL Server)和 Elasticsearch 中的对应关系,见表 1.1。

表 1.1 相关概念在关系型数据库和 Elasticsearch 中的对应关系

| 关系型数据库(如 SQL Server) | Elasticsearch |
|---|---|
| 数据库 Database | 索引 Index,支持全文检索 |
| 表 Table | 类型 Type |
| 数据行 Row | 文档 Document,但不需要固定结构,不同文档可以具有不同字段集合 |
| 数据列 Column | 字段 Field |
| 模式 Schema | 映像 Mapping |

(10) Mapping——Mapping 定义索引下 Type 的字段处理规则,如索引如何建立、索引数据类型、是否保存原始索引 JSON 文档、是否压缩原始 JSON 文档、是否需要分词处理、如何进行分词处理等。一般地,一个索引文件下能存储不同映像(Mapping)的类型文件(Types)。Mapping 也可通过语句删除,此时对应的类型文件下所有数据也会被删除。

## 1.2.3 Elasticsearch API 的简单使用方式

### 1. 非客户端方式

(1) 通过 HTTP 方式的 JSON 格式进行调用。关于 HTTP 的相关参数设置可在 elasticsearch.yml 中进行相关设置(出于安全考虑,也可禁用 HTTP 接口,只需在配置文件中将 http.enabled 设置为 false 即可)。本书在介绍 Elasticsearch 的使用时,除第 5 章外,都是基于这种方式使用 Elasticsearch API 的。

(2) 通过 Thrift 软件框架方式。Thrift 结合了功能强大的软件堆栈和代码生成引擎,以构建在 C++、Java 等编程语言间的无缝、高效服务。可通过 Thrift 方式实现跨语言操作,参见本书第 5 章基于 Java 客户端的编程实现方法。

(3) 通过 Memcached 方式。Memcached 是分布式内存对象缓存系统,用于在动态系统中减少数据库负载,提升性能。它适合大型的分布式系统,对于单台服务器的应用,此种方式有时可能是不合适的。

### 2. 客户端方式

另外,对应 Java 编程者来说,Elasticsearch 内置了两个客户端:

(1) 节点客户端。它以一个无数据节点的身份加入到集群中。换句话说,它自身是没有任何数据的,但是知道什么数据在集群中的哪一个节点上,然后就可以将请求转发到正确的节点上并进行连接。

(2) 传输客户端。更加轻量的传输客户端可被用来向远程集群发送请求。它并不加入集群本身,而是把请求转发到集群中的节点。

这两个客户端都使用 Elasticsearch 的传输协议,通过 9300 端口与 Java 客户端进行通信。集群中的各个节点也是通过 9300 端口进行通信。

Tips：Elasticsearch 的 9200 端口是 HTTP 接口，9300 端口是 Transport 接口。

### 1.2.4 Elasticsearch RTF 版本中的部分插件简介

Elasticsearch 的社区支持力度是比较大的。众多 Java 开源社区提供的丰富插件可以有效地帮助 Elasticsearch 初学者和中高级用户有效使用 Elasticsearch。有些插件是可以自己安插到 Elasticsearch 中的，如对于 Head 插件，用户可以从 http://mobz.github.io/elasticsearch-head 下载，之后进入 Elasticsearch 的 bin 文件夹，执行如下命令即可把 Head 安装到 Elasticsearch 环境中：

```
Plugin  -install  mobz/elasticsearch-head
```

其实，在 Elasticsearch 的 RTF 版本中，已经集成了部分常用的插件，部分插件是可以直接在浏览器中看到其作用效果的。RTF 中的 plugins 文件夹中就是已经安装好的插件，下面对部分插件功能进行说明。

- Analysis-ansj——ansj 中文分词器。
- Analysis-ik——IK 中文分词器。
- Analysis-mmseg——mmseg 中文分词。
- Analysis-pinyin——拼音分词器。
- Analysis-string2int——字符串转整型工具，可用在 Facets、Aggregations 功能上。
- Bigdesk——监控 Elasticsearch 状态。
- Head——对 Elasticsearch 进行多种操作的客户端工具。
- Inquisitor——调试查询。
- Kopf——网络管理。

Tips：汉语自动分词是中文自然语言处理的重要一环。构建于词之上的各种后续语言分析手段才有展示身手的舞台。例如，Ansj 是基于 ICTCLAS 中文分词算法而开发的开源的 Java 中文分词工具；IK Analyzer 是一个开源的基于 Java 语言开发的轻量级的中文分词工具包；MMSEG 是基于中文分词之最大匹配法扩展而来的中文分词工具。几种分词算法各有优势。

#### 1. Head

Head 是一个用来监控 Elasticsearch 状态的客户端插件。用户在浏览器中输入 http://localhost:9200/_plugin/head/，会打开类似图 1.3 的界面。图中显示了具有两个节点的结果（分别是 Jackal 和 Vargas，见左下角），每个节点拥有不同 index 的数据（这里的两份 index 分别为 myfirstindex 和 page）。

Head 插件可以为用户（特别是初学者）提供很多便利，例如可以使用 Head 提供的

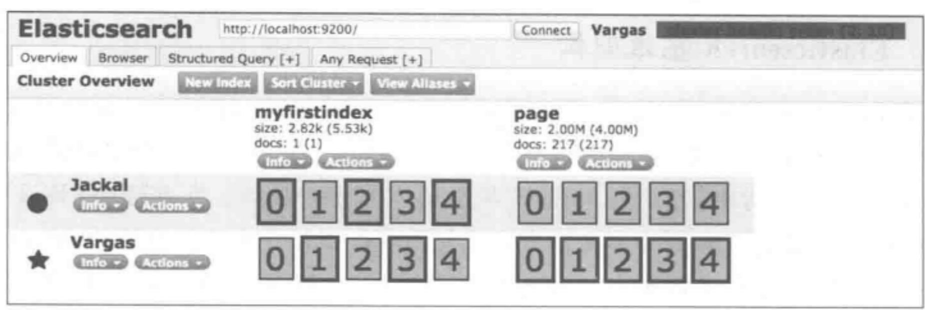


图 1.3 Head 界面

HTTP 客户端,通过 HTTP 的方式来操作 Elasticsearch。在 Head 中可以单击界面的 Any Request 标签来构建 HTTP 请求,辅以必要的 PUT、GET 等操作就能实现对应的功能。本书后续给出的例子有一部分就是基于 Head 插件来实现的。

### 2. Marvel

Marvel 是 Elasticsearch 的图形化监控客户端,可以通过 Marvel 来查看 Elasticsearch 当前的各项状态指标,使用方法是在浏览器中输入 http://localhost:9200/_plugin/marvel/,会显示出类似图 1.4 的内容。

图 1.4 Marvel

### 3. Health

Health 是用来查看集群目前健康状态的。在浏览器中输入 http://localhost:9200/_cluster/health 会显示当前集群的部分信息,如图 1.5 所示。

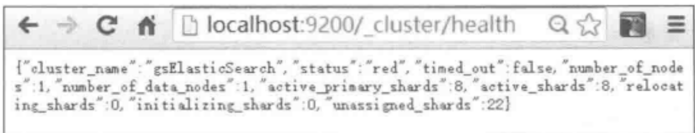


图 1.5 Health 界面

### 1.2.5 Elasticsearch 基本架构

Elasticsearch 的基本架构如图 1.6 所示。从图中可以看出，Elasticsearch 可以接受本地、共享及云平台上的数据；在 Lucene 提供的基本功能上，通过构建分布式索引，完成对大数据的加工处理。其中，Zen 用来做节点自动发现和 Master 节点选举；EC2（Elastic Compute Cloud）借由提供 Web 服务的方式让使用者可以弹性地运行自己的 Amazon 机器映像，提供可调整的云计算能力。通过提供的 Thrift、Memcached、HTTP 等方式使用 Elasticsearch 的 API。在顶层，用户可以基于 RESTful 和客户端（如 Java 客户端）的方式来通过 Elasticsearch API 完成数据操作、管理等工作。

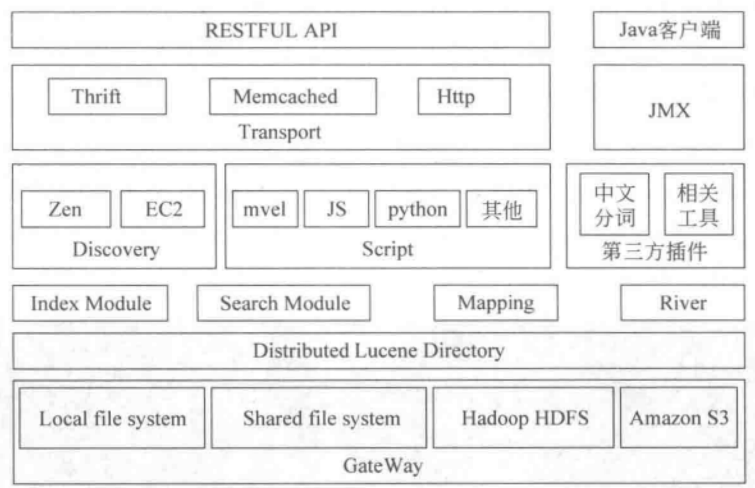


图 1.6　Elasticsearch 的基本架构

RESTful 接口 URL 的格式是：

```
http://localhost:9200/<index>/<type>/[<id>]
```

其中，index、type 是必须提供的（index 可以理解为数据库；type 理解为数据表）；id 是可选的（相当于数据库表中记录的主键，是唯一的。如果不提供，Elasticsearch 会自动生成）。增、删、改分别对应 HTTP 请求的 PUT、GET、DELETE 方法。对于 PUT 方法来说，调用时如果数据不存在，就创建；如果数据已存在，就更新。Elasticsearch 可能会返回一个 HTTP 状态码（类似如图 1.1 所示）以及一个 JSON 格式的主体。

Tips：Elasticsearch 的语句包括可能的如下几个部分[Fu，2015]：

- 相应的 HTTP 请求方法或者变量，如 GET、POST、PUT、DELETE。
- 集群中任意一个节点的访问协议、主机名以及端口。
- 请求的路径。
- 查询后再加上“? pretty”，可增强可读性。
- 一个 JSON 编码的请求主体（如果需要的话）。

## 1.3 Elasticsearch 索引及其构建

### 1.3.1 概述

对信息检索过程而言，建立索引是最基础的工作。当前主流的文档索引方法有倒排索引、后缀数组索引、签名文件索引等。Elasticsearch 是基于 Lucene 构建的，它采用倒排索引机制，将文件"封装"为索引，将文本信息切分成称为 Token 的信息单元，再利用这些 Token 构造倒排索引。这里所言的索引并非普通意义上的文档，而是一个可看作不同文档内容的集合。Elasticsearch 中的索引类似于数据库，而其中的类型类似于数据表，每个类型由多个字段组成，实际文档的具体内容可分别作为一个字段添加到某个类型中去。在 Elasticsearch 中设置好的分析器 Analyzer 的作用下，可以把某些字段的值进行分隔，并得到独立元素。

简单来说，在 Elasticsearch 中建立索引的简要流程如下：

(1) 准备待处理的文档。一般而言，文档准备可以采用 JSON 格式，也可以使用第三方的工具协助进行处理。本章后续给出的例子就是采用 JSON 格式来组织的。

(2) 将准备好的数据提交文档给 Elasticsearch。

(3) 完成索引。

(4) 返回索引结果。如果启动了 Elasticsearch，可以在前端浏览器中以如下方式来查看其细节信息：

```
http://localhost:9200/index名称/type名称/文档唯一标识
```

> Tips：上面例子是 GET 操作。其中的"index 名称"相当于 RDBMS 的数据库名称；"type 名称"相当于该数据库中的某个数据表名称；一个 index 中可以存在有多个 type。

### 1.3.2 借助 Head 工具构建索引

构建索引与检索时，可以利用 Elasticsearch 自带的 Head 工具来完成。本节介绍借助 Head 工具构建索引的方法。

首先，启动 Elasticsearch。之后，在浏览器的 URL 栏中输入如下 URL，打开 Elasticsearch 中的 Head 工具：

```
http://localhost:9200/_plugin/head/
```

其次，单击 New Index 按钮来创建一个新的索引，在弹出的框中输入索引名称，如图 1.7 所示，其中，Number of Shards 是分片数，对应如图 1.8 所示的 5 个数据分片(依次为 0,1,2,3,4)；Number of Replicas 是数据副本数，在图 1.8 中带有粗框的分片副本是正在提供数据服务的副本。如果能在 Head 中看到类似图 1.8 的结果，说明成功创建了索引。

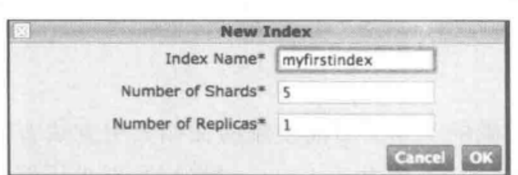


图 1.7 建立索引

图 1.8 建立好的索引

第三，在 Head 工具的 Any Request 标签下，打开 Query 选项，如图 1.9 所示。输入拟提交数据的索引文件名称(这里是指新建立的索引文件，即 myfirstindex)以及 Type(这里是 share)、索引文档的唯一标识(即索引的 ID 号，这里是 1)等信息，并选择操作方式(POST、GET、PUT、DELETE 等，这里选择 POST 方法)。接着，在下面的文本输入框中输入拟添加到该索引文件中的具体数据信息，数据以 JSON 格式表示。需要注意的是，JSON 数据格式有严格规定，一般来说，JSON 数据的书写格式是“名称/值对”。“名称/值对”组合中的名称写在前面(在双引号中)，值对写在后面(同样在双引号中)，中间用冒号隔开，如图 1.9 所示。提交后，就会在索引文件 myfirstindex 的类型文件 share 中，写入指定的信息。图 1.9 右侧表示写入成功。

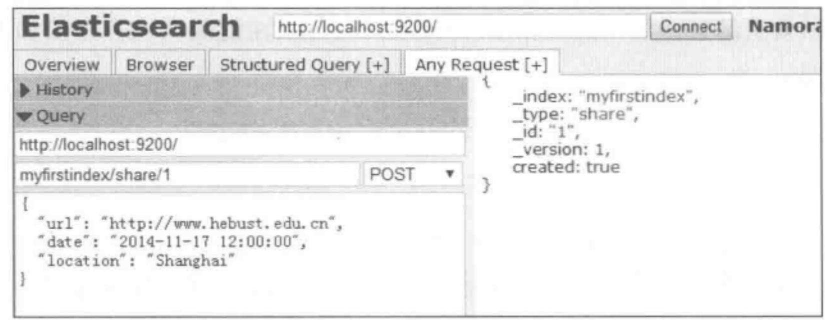


图 1.9 向索引文件中添加信息

如何查看索引文件中的数据呢？只需在图 1.9 的 HEAD 界面的左侧选择操作类型 GET，就可以看到该索引文件的详细信息，如图 1.10 右侧所示，图中右侧部分就是返回的以 JSON 形式组织的索引文件中的具体数据。

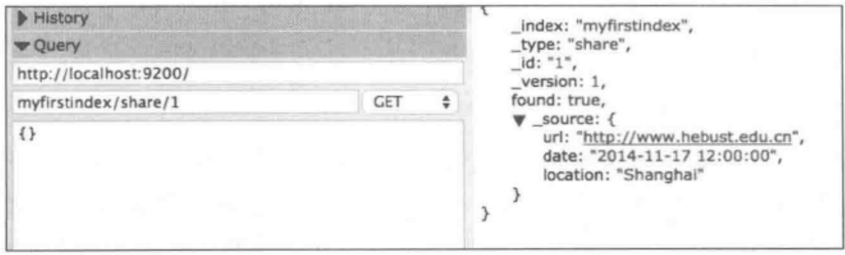


图 1.10 查看索引文件中的数据信息

> Tips：标准 HTTP 协议支持 6 种请求方法：POST、GET、PUT、DELETE、HEAD、OPTIONS。在 Elasticsearch 的 Head 中只用到前 4 个，其中：POST 是向服务器提交数据；GET 是发送一个请求来取得服务器上的某一资源；PUT 和 POST 都是向服务器发送数据，但 PUT 通常指定了资源的存放位置，而 POST 则没有(POST 的数据存放位置由服务器自己决定)；DELETE 是用来删除服务器上的某个资源。

### 1.3.3 Mapping 简述

Elasticsearch 中 Mapping 类似于静态语言中的数据类型。但同语言的数据类型相比，映像还有一些其他的含义。Elasticsearch 中的映像的作用就是执行一系列的指令，将输入的数据转成可搜索的索引项。映像不仅告诉 Elasticsearch 一个字段中是什么类型的值，还告诉 Elasticsearch 如何索引数据以及数据是否能被搜索到。借助于映像，可以很方便地观察到索引中的数据及其属性信息。

在 Elasticsearch 中使用映像是很容易的，只有在 URL 后指定某个索引文件名，后面跟 _mapping 参数即可，如图 1.11 所示。从图右侧可以清晰地看到索引文件 myfirstindex 中类型文件 share 中的各个字段的类型等信息。

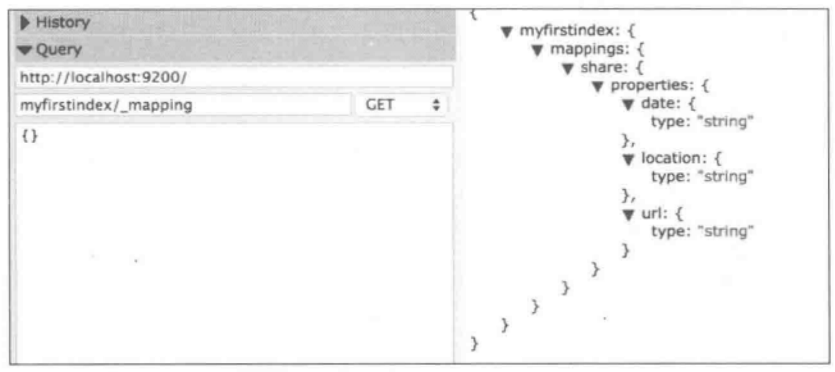


图 1.11 借助 Mapping 了解索引文件的相关信息

## 1.4 信息检索及其构建

用户可利用 Head 工具，通过 HTTP 传递参数的方式来构造一个简单的信息检索语句。图 1.12 展示了在指定的 myfirstindex 索引的 share 中，搜索字段为 location 且其值为 Shanghai 的检索请求的构建方式。在图右侧的 Hits 结果中可以看到返回的结果。其中，hits 表示命中的检索集合，total 表示命中 1 条记录，max_score 是其评分。另外，在图中的 URL 构建查询语句时，_search 表示搜索 RESTful 接口；q 后代表查询条件；q 后的＝是基于 Lucene 语法的查询表达式；中文需要 urlencode 字符编码，特殊字符需要转义处理。

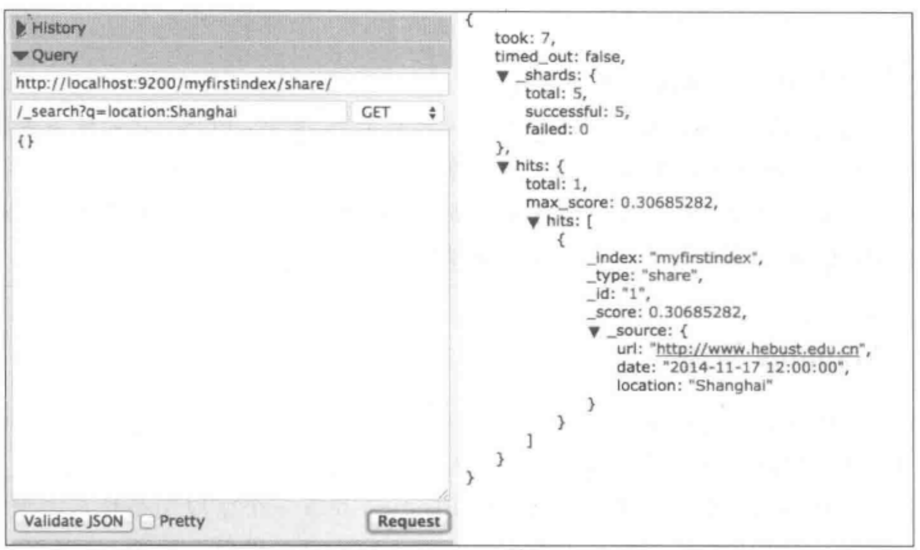


图 1.12　检索结果

> Tips：在基于 Lucene 的查询语法中，查询域 Field 指定从哪个域中寻找查询词；查询域和查询词之间用分号分隔(如 title："中国")；支持两种通配符？和 * 的查询；支持模糊查询。

总的来说，查询流程包括：

(1) 构造查询条件——可以基于 Lucene 的查询语法进行构建，也可以基于 Query DSL 查询表达式进行构建。有关后一种方式的介绍参加见第 2 章中的说明。

(2) 提交给 Elasticsearch。

(3) 返回 Hits 结果。

> Tips：Query DSL 是用表达式的类似 JSON 的方式来描述查询语句，是一个通用的查询框架，可以通过一组通用的查询 API 为用户构建出适合不同类型 ORM(Object Relational Mapping)框架或查询语句，Query DSL 支持的平台包括 Elasticsearch、SQL、RDF、Lucene 等。ORM 框架采用元数据来描述对象-关系映射细节，元数据一般采用 XML 格式，并且有的在专门对象-映射文件中。

## 1.5　实例

文献[Open，2014b]给出了从创建数据索引、构建映像、录入信息、构建检索表达式、返回检索结果等比较全面的信息检索过程。在得出检索结果后，可将得到的检索结果发送到

页面前端，在通过相应的 UI 渲染后，就能得到类似普通搜索引擎那样的结果了。为方便看到中间结果，下面的例子仍是借助 Head 插件实现的。

第一步，创建示例索引文件 test，注意用 PUT 方法实现，如图 1.13 所示。

Elasticsearch http://localhost:9200/

| Overview | Browser | Structured Query [+] | Any Request [+] |
|---|---|---|---|

History

Query

http://localhost:9200/

/test PUT

{}

图 1.13 创建索引示例 test

第二步，创建数据映像文件，主要代码如代码段 1.2 所示，处理结果如图 1.14 所示。注意，本例中采用的是 IK 分析器。

Tips：为便于书中表述风格的统一，在下面代码中参照 C 语言的注释方式增加了注释，但这种注释风格在 Head 和实际应用环境中是不允许的，这里只是为了方便说明而已。

```
//代码段 1.2：创建数据 mapping
{
  "news": {                              //建立 news 的 Type
    "properties": {                      //下面是定义各个字段
      "content": {                       //对 content 字段的结构设计
        "type": "string",                //当前字段为字符串类型
        "store": "no",                   //当前字段不存储
        "term_analyzer": "with_positions_offsets",
                                         //指示不仅保存分隔后的词 还保存词之间的距离
        "index_analyzer": "ik",          //设定索引时用的分词器是 IK 分析器
        "search_analyzer": "ik"          //设定搜索该字段时用的分词器是 IK 分析器
      },
      "title": {                         //对 title 字段的结构设计
        "type": "string",
        "store": "no",
        "term_analyzer": "with_positions_offsets",
        "index_analyzer": "ik",
        "search_analyzer": "ik",
        "boost": 5
      },
```

```
        "author": {                    //对 author 字段的结构设计
          "type": "string",
          "index": "not_analyzed"      //设定该字段为不分词
        },
        "publish_date": {              //对 publish_date 字段的结构设计
          "type": "date",
          "format": "yyyy/mm/dd"       //设定该日期字段的格式
        },
        "category": {                  //对 category 字段的结构设计
          "type": "string",
          "index": "not_analyzed"
        }
      }
    }
}
```

图 1.14 创建映像

第三步，录入数据信息。可以在 Head 中采用如图 1.15 的方法添加数据。添加数据后，可以在 Head 中看到数据情况，如图 1.16 所示。

第四步，构建检索表达式，其内容可包括如下几个主要部分：

- 分页(from/size)
- 字段(fields)

图 1.15 添加数据

Searched 5 of 5 shards. 1 hits. 0.002 seconds

| _index | _type | _id | _score | content | title | author | pub |
| --- | --- | --- | --- | --- | --- | --- | --- |
| test | news | C3D5nQgLTI--jvX8-CAE6A | 1 | 中国总理李克强访问德国,并会见了德国总理默克尔 | 中国总理出访 | 高凯 | 2014 |

图 1.16 查看添加好的索引数据

- 排序(sort)
- 查询(query)
- 过滤(filter)
- 高亮(highlight)
- 统计(facets)

有关构造检索的示例代码详见代码段 1.3(对其中部分代码含义的解释参见第 3 章)。检索结果如图 1.17 所示。

```
//代码段 1.3: 构建检索表达式
{
  "from": 0,                          //检索结果集合的起始位置
  "size": 10,                         //返回的检索结果数量
  "fields":[                          //指定检索结果 document 集合中含有哪些域
    "title",
    "content",
    "publish_date",
    "category",
    "author"
  ],
  "sort":[                            //定义排序方式
    {
      "publish_date": {               //搜索结果按照 publish_date 进行排序
        "order": "asc"                //按正序排序,即从小到大
```

```
        }
      },
      "_score"                              //先按照 publish_date,再按照_score 排序
    ],
    "query": {
      "bool": {                             //构建布尔查询
        "should": [
          {
            "term": {
              "title": "中国"
            }
          },
          {
            "term": {
              "content": "中国"
            }
          }
        ]
      }
    },
    "filter": {                             //创建过滤规则
      "range": {                            //按照区间进行过滤
        "publish_date": {
          "from": "2014/01/01",             //起始时间
          "to": "2014/12/31",               //结束时间
          "include_lower": true,
          "include_upper": false
        }
      }
    },
    "highlight": {                          //对高亮进行相关设置
      "pre_tags": [                         //高亮的前置标签
        "< tag1> ",
        "< tag2> "
      ],
      "post_tags": [                        //高亮的后置标签
        "< /tag1> ",
        "< /tag2> "
      ],
      "fields": {                           //需高亮的域
        "title": {},
        "content": {}
```

```
      }
    },
    "facets": {              //构建聚合
      "cate": {
        "terms": {
          "field": "category"
        }
      }
    }
  }
```

图 1.17 检索结果

## 1.6 扩展知识与阅读

从广义上说，Elasticsearch 也属于数据库范畴，但它能够轻松地进行大规模的横向扩展，以支持对 PB 级的结构化和非结构化海量数据的处理。由于关系型数据库对全文检索功能的支持相对不足(如传统关系型数据库并不能很好地解决大数据带来的问题，单机的统计和可视化工具也往往比较欠缺)，所以一些实际项目需要将关系型数据库中的大数据同步到 Elasticsearch 中，以提供更加强大的全文检索功能。进入 2015 年，Elasticsearch 的应用范围在进一步扩大。根据 DB-Eignes 的统计，截至 2015 年 5 月，ElasticSearch 在搜索引擎类数据库中排名第 2，如图 1.18 所示[Db-engines,2015]。

众所周知，Elasticsearch 和 Solr、Nutch 等都是基于 Lucene 构建的。了解 Lucene 的前世今生，更加有助于学好 Elasticsearch。有关 Lucene 的更详细的内容，参见文献[Michael,2011]。文献[韩陆,2014]介绍了基于 Java 标准规范实现 REST 的方法，这对于深刻理解

| 17 systems in ranking, May 2015 | | | | | | | |
|---|---|---|---|---|---|---|---|
| Rank | | | DBMS | Database Model | Score | | |
| May 2015 | Apr 2015 | May 2014 | | | May 2015 | Apr 2015 | May 2014 |
| 1. | 1. | 1. | Solr | Search engine | 82.93 | +0.92 | +15.77 |
| 2. | 2. | 2. | Elasticsearch | Search engine | 64.83 | +0.17 | +32.77 |
| 3. | 3. | 3. | Splunk | Search engine | 40.72 | +2.70 | +16.04 |
| 4. | 4. | ↑6. | MarkLogic | Multi-model | 9.90 | -0.31 | +2.09 |
| 5. | 5. | ↓4. | Sphinx | Search engine | 9.80 | +0.22 | +0.75 |
| 6. | 6. | ↓5. | Endeca | Search engine | 8.53 | +0.23 | -0.02 |
| 7. | 7. | 7. | Google Search Appliance | Search engine | 3.70 | +0.02 | -0.22 |
| 8. | 8. | 8. | Amazon CloudSearch | Search engine | 1.73 | -0.05 | +1.12 |
| 9. | 9. | | Microsoft Azure Search | Search engine | 0.74 | +0.02 | |
| 10. | 10. | ↓9. | Xapian | Search engine | 0.55 | +0.00 | +0.11 |
| 11. | 11. | 11. | Indica | Search engine | 0.30 | +0.01 | +0.12 |
| 12. | 12. | ↓10. | Compass | Search engine | 0.21 | -0.01 | +0.01 |
| 13. | 13. | | Crate.IO | Multi-model | 0.17 | +0.02 | |
| 14. | 14. | ↓12. | SearchBlox | Search engine | 0.08 | -0.01 | +0.04 |
| 15. | ↑16. | ↓13. | Srch$^2$ | Search engine | 0.01 | +0.01 | +0.01 |
| 16. | ↓15. | | DBSight | Search engine | 0.00 | -0.05 | |
| 16. | 16. | ↓13. | Exorbyte | Search engine | 0.00 | ±0.00 | ±0.00 |

图 1.18 ElasticSearch 的使用情况

JAX-RT 标准和 API 设计，了解 Jersey 的使用要点和实现原理，以及基于 REST 的 Web 服务的设计思想，是很有帮助的。文献[杨传辉，2014]系统讲述了构建大规模存储系统的核心技术和原理，分析了 Google、Amazon、Microsoft 和阿里巴巴的大规模分布式存储系统的原理。有关现代信息检索发展、倒排索引构建、搜索引擎系统研发等可以参阅文献[Ricardo，2012][高凯，2010]，而有关信息检索、搜索引擎等核心技术的通俗叙述，可以参阅文献[吴军，2013]。另外，Elasticsearch 扩展性非常好，有很多官方和第三方开发的插件，文献[云端分布式搜索技术，2013]对分词、同步、数据传输、脚本支持、站点等类别进行了划分。

## 1.7 本章小结

本章简介了 Elasticsearch 的背景知识，对其中出现的概念进行了说明。借助于可视化插件 Head，通过一个综合性例子，介绍了 Elasticsearch 的索引、检索流程和实现方法。通过学习本章内容，应该了解 JSON、RESTful 以及 Elasticsearch 相应插件等的用法。

Chapter 2

# 文档索引及管理

> “Elasticsearch is API driven. Almost any action can be performed using a simple RESTful API using JSON over HTTP. An API already exists in the language of your choice. Storing complex real world entities in Elasticsearch as structured JSON documents, all fields are indexed by default, and all the indices can be used in a single query, to return results at breath taking speed.”
>
> ——http://www.elasticsearch.org/overview/ elasticsearch.

在 Elasticsearch 中，对文档的索引等进行操作时，既可以通过 RESTful 接口进行操作，也可以通过 Java 客户端进行操作。本章介绍基于 RESTful 的文档索引与管理方法，面向 Java 客户端的编程方法参见第 5 章。

## 2.1 文档索引概述

在信息检索过程中，文本数据首先要经过加工处理后才能被检索到，而这个加工处理过程就是建立索引文件(通常是倒排索引文件)的过程。一般的信息检索过程是用户通过接口提交查询请求，在索引文件中检索出相关结果，并按相关度排序之后返回给用户。可见，建立文档索引是最基础的加工过程。由于检索通常要面向大量用户的查询，因此索引文件的设计要尽量高效，以便由索引项快速定位到相应文档。当前文档索引方法有倒排索引、后缀数组索引、签名文件索引等，其中倒排索引是应用最广泛的。在 Elasticsearch 中，采用的就是经典的倒排索引技术。目前成熟的信息检索系统几乎普遍采用这种索引方法。可以说，倒排索引是检索技术的基础。

倒排文件可看作是一种描述了一个词项集合 TERMS 元素和一个文档集合 DOCS 元素对应关系的数据结构。若记 $N$ 为文档集合的大小，$M$ 为词项集合的大小，DOCS$=\{d_1, d_2, \cdots, d_N\}$，TERMS$=\{t_1, t_2, \cdots, t_M\}$，则倒排文件可给出 $t_j$ 出现在哪些 $d_i$ 中(或 $t_j$ 在 $d_i$ 中出现在哪些位置)的信息。一般地，它从逻辑上看可分为两个部分：一部分用于表示文档的索引项；另一部分则由多个位置表组成，每个位置表和索引中的某个索引项相对应，并记录所有出现过该索引项的文档及其在文档中的具体位置，以便计算出索引项之间的相邻关系。

图 2.1 是一个倒排索引结构示意图，表示在文档 $d_1$ 和 $d_2$ 中出现过“应用”这个索引项，而在文档 $d_2$ 和 $d_3$ 出现过“技术”索引项。由于 $d_2$ 文档中上述两个索引项是相邻的（分别位于 51 和 52 位置，见图 2.1），因此“应用技术”就会出现在 $d_2$ 文档中。如果要检索“应用技术”一词，则首先在倒排索引中找到包含“应用”一词的候选文档集，然后从候选文档集合中筛选出在这个检索词后紧跟“技术”的文档（是 $d_2$，而非 $d_1$ 或 $d_3$）。这种方式可以加快检索速度[高凯,2010][高凯,2014]。

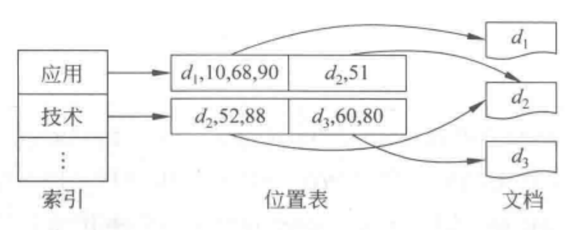


图 2.1 倒排文件示意图

## 2.2 建立索引

在 Elasticsearch 中，可以通过 Index API 来对文档进行索引操作。在建立索引文件时，可以设置分布式索引文件的 Shards 数量和 Replicas 数量，如可通过使用{index} / _settings（指定 Index 名称的配置）子句，修改索引文件的配置。

代码段 2.1 实现了对名为 myweibo2 的索引文件的创建工作（注意是通过 PUT 方式向系统提交索引请求），方法中指定了索引数据的 shards 数量和 replicas 数量（如果不指定它们，系统会采用默认值）。注意，这里的参数-d 后面的是提交的数据。

```
//代码段 2.1：使用 JSON 数据格式来创建索引。执行后会新建一个名为 myweibo2 的新的索引文件
curl -XPUT 'http://localhost:9200/myweibo2/' -d '{
"settings": {
    "index": {
        "number_of_shards": 5,
        "number_of_replicas": 1
    }
}
}'
```

也可以采用_settings 子句实现其他相应功能。下面的代码段 2.2 修改 myweibo1 索引文件，并将其 replicas 副本量改为指定的数据，图 2.2 显示该索引的副本已变成设定的数值了。

```
//代码段 2.2: 使用_settings 子句修改索引文件
curl - XPUT 'localhost:9200/myweibo1/_settings' - d '
{
    "index": {
                "number_of_replicas": 7
            }
}'
```

Tips: 上述语句中写 number_of_replicas 参数的地方也可换成如下参数：

- number_of_replicas——设置当前索引的副本数量。
- blocks. read_only——如设为 true，则当前索引只允许读，不允许写或更新。
- blocks. read——如设为 true，则禁止读取操作。
- blocks. write——如设为 true，则禁止写操作。
- blocks. metadata——如设为 true，则禁止对 metadata 操作。

```
    aliases: [ ]
},
 myweibo1: {
  state: "open",
    settings: {
       index: {
        uuid: "yw_5Fr9CTomlf12w7Na9pQ",
        number_of_replicas: "7",
        number_of_shards: "5",
          version: {
           created: "1000099"
          }
       }
    },
    mappings: {
       my1: {
          properties: {
             content: {
              type: "string"
```

图 2.2 索引文件属性信息

对于{index}/_settings 子句，如果选择的 HTTP 操作类型是 GET（见下述语句），则可以获取当前索引文件 weibo 的较为详细的配置信息。

```
curl -XGET 'http://localhost:9200/weibo/_settings'
```

返回值如下：

```
{
    "weibo": {
    "settings": {
        "index": {
              "uuid": "Z3_EcCZHRb255LdLHol1ww",
               "number_of_replicas": "2",
               "number_of_shards": "5",
               "version": {
```

```
                    "created": "1000099"
                        }
                    }
                }
            }
    }
```

类似地,采用如下的类似语句,可以一次性获得多个索引文件(例子是返回 weibo、weibo2 这两个索引文件)的配置信息:

```
curl -XGET 'http://localhost:9200/weibo, weibo2/_settings'
```

还可以使用_all 参数来获取所有的索引的配置信息:

```
curl -XGET 'http://localhost:9200/_all/_settings'
```

也可以使用通配符来获取一批索引的配置参数,语句如下所示:

```
curl -XGET 'http://localhost:9200/.marvel*/_settings
```

另外,也可通过指定 JSON 格式的数据来向指定的索引文件中插入数据,并建立相应的索引。代码段 2.3 新建了一个索引(名为 myweibo3)及其 type 文件(名为 example),并向其中写入指定 JSON 格式的字段信息(这些 Fields 包括 user、post_date、mymessage 等)。执行完毕后的结果如图 2.3 所示。

```
//代码段 2.3: 指定 JSON 格式的数据来向指定的 Index 文件中插入数据。自动生成 ID 号
curl -XPUT 'http://localhost:9200/myweibo3/example/_create -d '{
    "user": "Alan",
    "post_date": "2014-11-26T08:00:00",
    "mymessage": "this is an example on operation type on create"
}'
```

Searched 5 of 5 shards. 1 hits. 0.001 seconds

| _index | _type | _id | _score ▼ | user | post_date | mymessage |
|---|---|---|---|---|---|---|
| myweibo3 | example | _create | 1 | Alan | 2014-11-26T08:00:00 | this is an example on operation type on create |

图 2.3　向新建的索引文件中插入指定的信息

Tips: Elasticsearch 的内置字段主要有_uid、_id、_type、_source、_all、_analyzer、_boost、_parent、_routing、_index、_size、_timestamp、_ttl 等,字段类型主要有 String、Integer/long、Float/double、Boolean、Null、Date 等。在图 2.3 中已经看到了_index、_type、_id、_score 这几个内置字段(单词本身所表示的含义此处不再赘述)。图中显示的其他字段(即 user、post_date、memessage)是用户以 JSON 格式自定义的字段。

代码段 2.4 是向刚才新建立的索引文件 myweibo3 的类型文件 example 中添加文档的语句。这个文档共有 3 个字段，它们分别是 user、post_date、mymessage。和前述语句不同的是，在 URL 最后的参数 3 是指定了这个新插入的 Document 的 ID 号。该语句的执行结果如图 2.4 所示。

```
//代码段 2.4: 指定新插入索引数据的 ID 号
curl -XPUT 'http://localhost:9200/myweibo3/example/3' -d '{
"user": "LiMing",
"post_date": "2014-11-24T14:12:12",
"message": "Hello Tom"
}'
```

Searched 5 of 5 shards. 3 hits. 0.001 seconds

| _index | _type | _id | _score ▼ | user | post_date | mymessage |
| --- | --- | --- | --- | --- | --- | --- |
| myweibo3 | example | 3 | 1 | LiMing | 2014-11-24T14:12:12 | Hello Tom |

图 2.4 向指定的索引文件中插入指定内容的信息

Tips: 上面例子中指定了这个 Document 的 ID 为 3，但是也可以不指定 ID。这样，Elasticsearch 会自动生成一个 ID 号并在结果中返回。

在建立索引文件后，可以通过语句获取指定索引文件的状态信息，如 curl-XGET' localhost:9200/indexfile/_stats'，注意这里使用了/_stats 参数。图 2.5 给出查询索引文件返回的状态信息。

在图 2.5 返回的对象中，可以看到几个对象：primaries（包含当前节点之上的所有主分片的信息）、total（包含所有分片及副本的信息）。另外，所有这些对象都包含如下对象：

- docs——显示被索引文档的信息，其中的 count 值表示所描述的索引中的文档数量。
- store——反映索引的大小，以及 throttling 信息等。
- indexing——索引操作信息。
- get——实时获取操作信息。
- search——搜索操作信息。

Tips: 也可以同时给出多个索引的统计信息，如“/索引 1，索引 2，索引 3/_stats”。

```
{
  "_shards" : {
    "total" : 10,
    "successful" : 10,
    "failed" : 0
  },
  "_all" : {
    "primaries" : {
      "docs" : {
        "count" : 2145821,
        "deleted" : 0
      },
      "store" : {
        "size_in_bytes" : 15818405646,
        "throttle_time_in_millis" : 42
      },
      "indexing" : {
        "index_total" : 11,
        "index_time_in_millis" : 219,
        "index_current" : 0,
        "delete_total" : 0,
        "delete_time_in_millis" : 0,
        "delete_current" : 0
      },
      "get" : {
        "total" : 0,
        "time_in_millis" : 0,
        "exists_total" : 0,
        "exists_time_in_millis" : 0,
        "missing_total" : 0,
        "missing_time_in_millis" : 0,
        "current" : 0
      },
      "search" : {
        "open_contexts" : 24,
        "query_total" : 13706,
        "query_time_in_millis" : 8068055,
        "query_current" : 0,
        "fetch_total" : 2710,
        "fetch_time_in_millis" : 1374981,
        "fetch_current" : 24
      },
      "merges" : {
        "current" : 0,
```


图 2.5 查询索引文件状态信息

## 2.3 通过映像 Mapping 配置索引

类似于在 RDBMS 中建立数据表时要定义其字段名及其数据类型那样，映像 Mapping 是对 Elasticsearch 中索引字段名及其数据类型的定义。但映像要比 RDBMS 中的结构定义灵活得多。其实，在使用 Elasticsearch 时，不指定映射也是可以的，因为 Elasticsearch 会自动根据数据格式定义它的类型。但如果需要对某些字段添加特殊属性（如：该字段是否分词、是否存储、使用什么样的分词器等），就必须添加和设置映像。

为索引文件添加映像有两种方式：一种是定义在配置文件中，另一种是在运行时手动提交映像。

### 2.3.1 在索引中使用映像

下面的代码段 2.5 就是手动提交映像的例子。代码运行后，会创建名为 weibo 的索引文件（Index），其内部会包含一个名为 wb 的类型（Type），这个 Type 中有一个名为 user 的

字段，其数据类型为 string，且设置不对该字段内容进行分词。请注意在代码段 2.5 中，也同时指定了索引文件的 shards 数和 replicas 数。

```
//代码段 2.5: 通过 Mapping 设置 Index 中某个 Type 下的 Field 中的细节信息
curl -XPOST localhost:9200/weibo -d '{
"settings": {
    "number_of_shards": 5,
    "number_of_replicas": 1
},
"mappings": {
    "wb": {
        "properties": {
            "user": {  "type": "string",
                       "index": "not_analyzed"
                    }
        }
    }
}
}'
```

## 2.3.2 管理/配置映像

一般地，可以使用类似下面的方法完成管理/配置映射文件。

```
PUT /{index}/_mapping/{type}
```

其中，PUT 代表 HTTP 方法，{index}表示对应的索引文件名称，{type}表示类型文件名称。针对{index}和{type}，可以使用下面的语法进行扩充：

```
对{index}部分的扩展: blank | * | _all | glob pattern | name1, name2, ...
对{type}部分的扩展: 需要配置 Mapping 的 type 名称
```

下面的代码段 2.6 展示了如何管理/配置索引文件的映像。此例中，在名为 weibo 的索引文件中，针对其中的某个类型文件(此例为 wb)，对名为 mymessage 的字段进行了设置。这里定义 memessage 的数据类型为字符串，且它需要在 Elasticsearch 中存储。

```
//代码段 2.6: 通过 Mapping 配置 Type 文件中的细节属性
curl -XPUT 'http://localhost:9200/weibo/wb/_mapping' -d '{
    "wb": {
        "properties": {
                    "mymessage": {"type": "string",
                                  "store": true
                                 }
                 }
          }
}'
```

### 2.3.3 获取映像信息

可通过 GET 方法来获取映像中的信息，相关命令为：

```
GET /{index}/_mapping/{type}
```

下面的代码给出了面对具体类型文件的获取信息的方法。

```
curl -XGET 'http://localhost:9200/weibo/_mapping/wb'
```

和前述情况类似，在{index}中写入索引文件名称（如果有多个索引文件，可以使用逗号隔开，也可以使用_all 参数来匹配所有索引）；在{type}中写入具体的类型文件名称（如果有多个也可以使用逗号来分隔它们）。例如，如果获取所有索引文件的所有类型文件的映像信息，可使用下面方法：

```
curl -XGET 'http://localhost:9200/_all/_mapping'
curl -XGET 'http://localhost:9200/_mapping'
```

如果指定了具有的类型，则可用类似下面的语句来获取相应类型中的信息（下面的做法可以获取两个类型文件——即 wb 和 pages——中的信息）。

```
curl -XGET 'http://localhost:9200/_all/_mapping/wb, pages'
```

代码段 2.7 演示了如何查看索引文件名为 weibo、类型文件为 wb、字段为 user 的映像配置信息的方法。

```
//代码段 2.7: 查看指定 Field 的 Mapping 的方法
curl -XGET 'http://localhost:9200/weibo/_mapping/wb/field/user'
```

针对上述语句的返回值如下：

```
{"weibo": {
    "mappings": {
        "wb": {
            "user": {  "full_name": "user",
                       "mapping": {"user": {"type":"string", "index": "not_
                       analyzed"}}
                    }
            }
        }
    }
}
```

代码段 2.8 演示了如何查看跨索引（即多 Index）的 field 的方法，以及在所有索引中针对某些指定类型中的指定字段内容的方法，注意在第 3 行代码中的通配符的用法。

```
//代码段 2.8: 在多个 Index 或多个 Type 中查询 Mapping 的方法
curl -XGET 'http://localhost:9200/weibo, weibo2/_mapping/field/time'
curl -XGET 'http://localhost:9200/_all/_mapping/wb,pages/field/time,meaasge'
curl -XGET 'http://localhost:9200/_all/_mapping/wei*/field/*.id'
```

### 2.3.4 删除映像

映像也可以通过 DELETE 方法删除。同样，这里也允许使用通配符和_all 等参数。注意在下面给出的方法中，方括号里面代表需要使用的 HTTP 方法(这里是 DELETE 方法)。

```
[DELETE] /{index}/{type}
[DELETE] /{index}/{type}/_mapping
[DELETE] /{index}/_mapping/{type}
```

在上述语句后面的{index}参数和{type}参数列表中可使用的参数如下，如果存在多个名称，用逗号分隔开它们即可。

```
{index}部分:    * | _all | glob pattern | name1, name2, …
{type}部分:     * | _all | glob pattern | name1, name2, …
```

## 2.4 管理索引文件

### 2.4.1 打开、关闭、检测、删除索引文件

一个关闭的索引将禁止写入或读取对其中数据的操作，而一个打开的索引文件可以允许用户对其中的数据文件进行操作。通过使用 /{index 名称}/_open 和 /{index 名称}/_close 语句，可以打开或关闭指定的索引文件，具体方法见代码段 2.9。

```
//代码段 2.9: 打开及关闭索引文件 my_index 的方法
curl -XPOST 'localhost:9200/my_index/_open'
curl -XPOST 'localhost:9200/my_index/_close'
```

如下代码段 2.10 会检测索引文件 weibo 是否存在(通过返回的 HTTP 状态码进行检查)。如果返回的状态码是 200，则表示存在相应的索引文件；如果返回的状态码是 404，则表示文件不存在。

```
//代码段 2.10: 检测索引文件 weibo 的状态
curl -XHEAD 'http://localhost:9200/weibo' -v
```

类似地，通过删除索引的 DELETE API，可以删除一个或者多个索引，如下代码段 2.11 删除名为 weibo1 的索引文件。

```
//代码段 2.11: 删除索引 weibo1
curl -XDELETE 'http://localhost:9200/weibo1/'
```

还可使用通配符来批量删除名称相似的索引文件。如下代码段 2.12 会删除以 weibo 字符开头的一组索引文件。这里，也可使用_all 参数来删除全部索引文件。

```
//代码段 2.12: 使用通配符批量删除索引文件
curl -XDELETE 'http://localhost:9200/weibo*/'
```

### 2.4.2 清空索引缓存

通过下面的 API，可以清空指定索引中的缓存(使用 clear 参数)，参见代码段 2.13。

```
//代码段 2.13: 清空索引 weibo 中的缓存
curl -XPOST 'http://localhost:9200/weibo/_cache/clear'
```

如果指定了多个索引文件名称，也可以一次清空多个索引缓存，代码段 2.14 可以清空两个索引文件 weibo1、weibo2 中的缓存。

```
//代码段 2.14: 清空多个索引缓存
curl -XPOST 'http://localhost:9200/weibo1, weibo2/_cache/clear'
```

### 2.4.3 刷新索引数据

通过 POST 方法中的_refresh 参数语句，可以刷新一个或多个索引文件的状态，以便反映最新变化，参见代码段 2.15。

```
//代码段 2.15: 刷新索引数据
curl -XPOST 'http://localhost:9200/weibo/_refresh'
```

这里也可以通过指定多个索引名称，完成一次刷新多个索引文件(分别指定索引文件名)或全部索引(没指出索引文件名)的任务，方法如代码段 2.16 所示。

```
//代码段 2.16: 一次刷新多个索引文件或全部索引文件
curl -XPOST 'http://localhost:9200/weibo1, weibo2/_refresh'
curl -XPOST 'http://localhost:9200/_refresh'
```

### 2.4.4 优化索引数据

相对于 Lucene 的索引，Elasticsearch 索引过程多了分布式数据的扩展，它主要是用 Tranlog 进行各节点间的数据平衡。通过代码段 2.17 中的_optimize 参数，可以优化一个或

多个索引。

```
//代码段 2.17: 优化索引数据
curl -XPOST 'http://localhost:9200/weibo/_optimize'
```

类似地，也可以通过指定多个 index 名称，完成一次优化多个索引文件或全部索引的任务，此处不再赘述。

### 2.4.5 Flush 操作

Flush 操作将暂存于内存中的临时数据送至索引文件，并清空内部操作日志等，如代码段 2.18 所示。

```
//代码段 2.18: Flush 索引数据
curl -XPOST 'http://localhost:9200/weibo/_flush'
```

类似地，也可以通过指定多个 Index 名称，完成一次 Flush 多个索引或全部索引的任务，此处不再赘述。

## 2.5 设置中文分词器

全文检索往往需要对中文进行分词。有关中文分词的背景知识可参见 2.7 节。Lucene 提供了分析器用于处理分词，此外也有一些开源的分词处理模块可供方便地集成到信息检索系统中。如果需要为当前的索引文件定义一个新的分词器，需要先关闭当前索引，然后在更改分词器后，再次打开这个索引文件。相关语句如代码段 2.19 所示。

```
//代码段 2.19: 设置索引分析器
curl -XPOST 'localhost:9200/weibo/_close'                 //关闭 weibo 索引
curl -XPUT 'localhost:9200/weibo/_settings' -d '{         //设置 weibo 索引
    "analysis": {                                           //更改分词器
        "analyzer":{
      "content":{
            "type":"custom",                                //设置应用分词器的字段
            "tokenizer":"standard "                         //采用的分词器
}
        }
    }
}'
curl -XPOST 'localhost:9200/weibo/_open'                  //重新打开 weibo 索引
```

代码段 2.20 演示了通过特定分词器对指定文字进行分词并观察分词结果的方法。

```
//代码段 2.20: 使用 standard 分词器对指定文字进行分词
curl -XGET 'localhost:9200/_analyze?analyzer=standard&pretty' -d 'this is a
test'
```

上述代码执行后的返回值如下：

```
{
  "tokens": [ {
    "token": "this",                        //第 1 个词
    "start_offset": 0,
    "end_offset": 4,
    "type": "<ALPHANUM>",
    "position": 1
  }, {
    "token": "is",                          //第 2 个词
    "start_offset": 5,
    "end_offset": 7,
    "type": "<ALPHANUM>",
    "position": 2
  }, {
    "token": "a",                           //第 3 个词
    "start_offset": 8,
    "end_offset": 9,
    "type": "<ALPHANUM>",
    "position": 3
  }, {
    "token": "book",                        //第 4 个词
    "start_offset": 10,
    "end_offset": 14,
    "type": "< ALPHANUM> ",
    "position": 4
  } ]
}
```

## 2.6 对文档的其他操作

除索引文档外，Elasticsearch 也提供多种途径对文档进展诸如获取信息、增删改、更新等相关操作。

### 2.6.1 获取指定的文档信息

可以通过 GET 方法来获取指定文档的详细信息，代码段 2.21 演示了查看索引 weibo 下类型文件名为 wb 且 ID 号为 2 的记录信息的方法，注意这里的 HTTP 方式是 GET。

```
//代码段 2.21: 获取指定文档文档信息
curl -XGET 'http://localhost:9200/weibo/wb/2'
```

返回结果如下,包括在_source 段中的内容就是文档中的详细内容(含各字段及其内容),具体如下:

```
{
"_index": "weibo",                    //索引名
"_type": "wb",                        //类型名
"_id": "2",                           //ID号
"_version": 1,                        //版本号
"found": true,
"_source": {"user": "LiMing",         //详细信息
"post_date": "2014-10-15T14:12:12",
"message": "Hello Tom"
}
}
```

除了上面的方法外,还可设置想要显示或屏蔽的结果。代码段 2.22 演示了关闭_source 过滤器后的效果(注:语句中的问号?表示其后是参数,pretty 表示返回的结果中显示缩进以方便阅读)。执行上该语句后,在输出的结果中将不再带有 source 内容。

```
//代码段 2.22: source 过滤器
curl -XGET 'http://localhost:9200/weibo/wb/2? pretty&_source= false'
```

在下面的代码段 2.23 中,演示了如何只显示特定字段的方法(此例只显示 fields 为 user 的内容):

```
//代码段 2.23: 显示特定字段的方法
curl -XGET 'http://localhost:9200/weibo/wb/2? pretty&fields= user'
```

针对上述语句的返回结果将只带有 user 这个字段,如下所示:

```
{
"_index": "weibo",
"_type": "wb",
"_id": "2",
"_version": 1,
"found": true,
"fields": {
"user": [ "LiMing" ]
}
}
```

### 2.6.2 删除文档中的信息

可以使用 DELETE 方法来删除文档中的相应信息。执行代码段 2.24 中的语句，会删除 weibo 索引中类型文件为 wb 且 ID 号为 2 的信息。

```
//代码段 2.24: 删除指定的文档信息
curl -XDELETE 'http://localhost:9200/weibo/wb/2'
```

针对上述语句的返回值如下，表明删除成功：

```
{
  "found": true,
  "_index": "weibo",
  "_type": "wb",
  "_id": "2",
  "_version": 2
}
```

### 2.6.3 数据更新

如果需要对索引文档中的类型或其中的文档进行更新，需要用 Elasticsearch 提供的 UPDATE API 来实现。它的处理过程是先取出文档，运行指定的脚本，之后更新文档。下面的代码段 2.25 是在索引 weibo 中类型文件名为 wb 中增加 ID 为 3 的文档信息（给定了 4 个 field，即 user、post_date、message、like，这里的 like 字段表示针对此条微博的点赞的数量）。

```
//代码段 2.25: 直接向索引文件中指定的 ID 中录入信息
curl-XPUT 'http://localhost:9200/weibo/wb/3'     //直接指定 ID 号，这里的 HTTP 方
                                                   法是 PUT
{
  "user": "LiMing",
  "post_date": "2014-10-15T14:12:12",
  "message": "Hello Tim",
  "like": 3
}
```

可以通过使用 Update API 将上述这个文档中的 like 字段数值增加到 4，实现方法如代码段 2.26 所示，注意这里的 ctx._source. 表示本文档的内容，ctx._source.like 表示文档中某个具体的字段（这里是指 like 字段），而 params 是拟使用的新参数值。

```
//代码段 2.26: 通过_update 方式更新数据
curl-XPOST http://localhost:9200/weibo/wb/3/_update
                        //请注意这里的_update,HTTP 方法是 POST
```

```
{
  "script": "ctx._source.like + = count"
  "params": {
    "count": 4
  }
}
```

针对上面的例子，执行完上述语句后的索引数据如图 2.6 所示，注意字段中的 like 字段值已经由原先是 3 变为 7。

```
_score: ,
_source: {
    user: "LiMing",
    post_date: "2014-10-15T14:12:12",
    message: "Hello Tim",
    like:
},
```

图 2.6　更新 like 字段后的索引数据

> Tips：上面的代码中，"ctx._source."表示 document 内容，ctx._source.like 表示该 document 中具体的 fields 是 like 字段；params 表示为变量赋值为 4。图 2.6 中的结果可以通过语句 curl XPUT GET http://localhost:9200/weibo/wb/3 命令得到。

上述语句是针对于数值型数据的增删改操作。类似地，也可以完成对字符型数据的更新。代码段 2.27 录入了 ID 号为 5 的针对索引文件为 weibo 的类型文件名为 wb 的部分微博数据字段(user、post_data、message、tags)。

```
//代码段 2.27：直接向指定的 ID 中录入信息
curl-XPOST http://localhost:9200/weibo/wb/5   //注意这里使用的 HTTP 方法是 POST
{
    "user": "LiMing",
    "post_date": "2014-10-15T14:12:12",
    "message": "Hello Lily",
    "tags": [
            "Hello"
            ]
}
```

代码段 2.28 的作用是修改了上述语句执行结果中的 tag 信息并增加了新的内容(即：在 tag 中增加了新的内容)。修改后的文档信息如图 2.7 所示(可通过 curl XGET http://localhost:9200/weibo/wb/5 命令得到结果)。

```
//代码段 2.28：通过_update 方式更新信息
curl-XPOST http://localhost:9200/weibo/wb/5/_update    //注意这里的 POST 方法
{
```

```
    "script": "ctx._source.tags+=tag",
    "params": {
        "tag": "Today is a nice day!"
    }
}
```

```
post_date: "2014-10-15T14:12:12",
message: "Hello Lily",
  tags: [
   "Hello",
   "Today is a nice day!"
]
```

图 2.7　更新 Tag 后的索引数据

由于 Elasticsearch 处理的文档是非结构化的信息，因此可能与关系型数据库不同，文档中各记录的字段有可能不一样。利用_update，不仅可以像上述那样更新数据，也可以修改文档结构(如只在某一 ID 的记录上增加新的字段)。代码段 2.29 在指定的文档中增加新的字段，而这个新的字段的名字是 name_of_new_field，其值是 new_field，注意这里对引号需转义处理。

```
//代码段 2.29：通过 update 方式修改文档结构并增加新的字段
curl-XPOST http://localhost:9200/weibo/wb/5/_update
{
    "script": "ctx._source.name_of_new_field=\"new_field\""
}
```

代码段 2.30 表示在使用_update 参数时，如果该记录(即指定的 ID 号，此例中是 7)不存在，则通过参数体中的 upsert 创建这个文档，并且加入新的字段(其名为 counter，其值为 1)；如果有该记录(即指定的 ID 号，例子中是 7)，就把指定的字段 like 值增加 4(在代码中的 params 中表明拟增加的偏移量)。

```
//代码段 2.30：通过 update 方式修改 document 结构
curl-XPOST http://localhost:9200/weibo/wb/7/_update
{
    "script": "ctx._source.like+=count",
    "params": {
        "count": 4
    },
    "upsert": {
        "counter": 1
    }
}
```

### 2.6.4 基于 POST 方式批量获取文档

Elasticsearch 支持一次操作多条记录。代码段 2.31 可返回 ID 号为 5 和 7 的记录信息，注意语句中的_mget 参数的使用。

```
//代码段 2.31：基于 post 方式批量获取文档
curl-XPOST http://localhost:9200/weibo/wb/_mget?
{
  "docs": [
    {
      "_index": "weibo",
      "_type": "wb",
      "_id": "5"
    },
    {
      "_index": "weibo",
      "_type": "wb",
      "_id": "7"
    }
  ]
}
```

针对代码段 2.31 的返回结果如图 2.8 所示。

```
{
  ▼ docs: [
      {
          _index: "weibo",
          _type: "wb",
          _id: "5",
          _version: 3,
          found: true,
          ▼ _source: {
              user: "LiMing",
              post_date: "2014-10-15T14:12:12",
              message: "Hello Lily",
              ▼ tags: [
                  "Hello",
                  "Today is a nice day!"
              ],
              name_of_new_field: "new_field"
          }
      },
      {
          _index: "weibo",
          _type: "wb",
          _id: "7",
          _version: 1,
          found: true,
          ▼ _source: {
              counter: 1
          }
      }
  ]
}
```

图 2.8 基于 POST 方式批量获取文档数据

在代码中也可使用_source 过滤器。代码段 2.32 只获取其中的一个文档信息。

```
//代码段 2.32: 使用_source 过滤器获取文档
curl-XPOST 'localhost:9200/_mget?pretty' -d '{
    "docs": [
              { "_index": "weibo",
                "_type": "wb",
                "_id": "3",
                "_source": false
              }
          ]
}'
```

针对上述语句的返回值如下：

```
{    "docs": [ {
    "_index": "weibo",
    "_type": "wb",
    "_id": "3",
    "_version": 1,
    "found": true
    } ]
}
```

### 2.6.5 删除部分文档

可以通过联合使用_query 方法和 DELETE API，将命中的文档中的部分或全部内容删掉。代码段 2.33 检索到用户名为 LiMing 的文档，之后通过 DELETE 方法将其删掉。

```
//代码段 2.33: 删除文档 LiMing
curl -XDELETE 'http://localhost:9200/weibo/wb/_query?q=user:LiMing'
```

## 2.7 扩展知识与阅读

实现人机间自然语言的交互，意味着要使计算机既能理解自然语言的意义，也能以自然语言文本来表达给定的意图、思想等。前者一般称为自然语言理解或自然语言处理(Natural Language Processing，NLP)，后者是自然语言生成。NLP 包括词法分析(涉及自动分词、分词歧义消解、未登录词识别与获取等)、语法分析(涉及自动标注、语法表示、语法分析等)、语义分析(涉及语义表示、语义分析、语义消歧等)，其应用包括文本分类、信息抽取、自动文摘、统计对齐与机器翻译、聚类等。研究自然语言处理，是希望计算机能够在某种程度上理解并处理人类的自然语言。目前各种自然语言处理技术都相继出现并已应用到某

些领域[张华平,2014]。

国内外关于自然语言处理的研究曾长期专注于语法层次。在 20 世纪末期人们就认识到单纯从语法层次上进行研究是不能解决实际问题的。目前,国内外在信息检索领域内,对自然语言理解的研究已有很多成果。研究方法有基于语言学规则分析的方法和基于统计的方法两种[高凯,2014]。基于规则分析的方法是以建立形式化的知识系统来阐述语言知识,规则多是通过内省的方式得到,其本质是一种确定性的演绎推理,虽然这种方法可以实现对单个句子的分析,但却难以覆盖各种复杂的语言现象。同时,如果要添加新的规则,还需要注意到与已有规则间的相容性。正因为基于规则的方法的诸多缺陷的存在,再加之后来大量语料库的涌现,使得基于统计的方法逐渐兴起,其最大优点是可以使语言现象数量化,再加之其他的一些优势,这种方法的应用范围非常广泛,但是该方法可能会掩盖一些小概率事件的发生,所以该方法也有局限性。总之,两种方法各有各的优缺点。将二者结合起来,优势互补,并借此实现面向大规模真实文本的信息处理,是可行的。

在进行自然语言处理时,分词——特别是对像中文这样的亚洲语言来说——是必要的。相比较以空格自然分开的英文来说,中文分词处理要复杂得多,因为在英文中广泛存在的空格就是最简单的分词标志(两个空白之间的字符串即被定义为一个所谓的 Token),但是对中文而言,问题就没有这么简单了。比如中文中广泛存在着的切分歧义等问题,在西文中就基本没有。中文的切分歧义包括交叉歧义(如"乒乓球拍卖完了"等)、组合歧义(需要根据上下文,甚至整个句子来判断)、真歧义(由人来判断也不知道应该怎样切分合适)等。又比如,中文信息处理中对未登录词的处理也是一个难点。由于全文检索需要对没有分隔标记的文本进行分析和处理,因此基本的分词处理是必须的,特别是在对文本数据建立倒排索引时。文献[高凯,2010]介绍了 Lucene 中常用分词器及分词效果。

倒排索引是一种数据结构,常见的搜索引擎系统多是采用"词"作为检索项,可根据某个词,找到这个词出现的所有文档及出现位置。简单地说,倒排索引类似一个字典,该字典中的每一个条目都指向一个列表,列表中的每一项指向一个此条目在某个文档中出现的位置。这个条目的值应该是词,而不是单独的字。例如要查询出现"计算机"三个字的文档,当然不希望查询的结果中出现"计"或"算"或"机"这三个单个的字,因为"计算机"是一个具有单独意思的词,所以分词的准确度直接影响搜索引擎的可用性。一般认为,对文档的分词处理,是对其进行分类等处理的前期步骤。

## 2.8 本章小结

本章主要介绍了 Elasticsearch 中和索引相关的知识,内容包括基于 RESTful 的接口进行文档索引的方法,内容涉及索引建立、文档操作等的 API 用法。Elasticsearch 的映像 Mapping 类似于静态语言中的数据类型。映像不仅告诉 Elasticsearch 一个字段中是什么类型的值,它还告诉 Elasticsearch 如何索引数据以及数据是否能被搜索到。

Chapter 3

# 信息检索与结果过滤

> "Elasticsearch uses Lucene under the covers to provide the most powerful full text search capabilities available in any open source product. Search comes with multi-language support, a powerful query language, support for geolocation, context aware did-you-mean suggestions, autocomplete and search snippets. The Search API allows to execute a search query and get back search hits that match the query. It can be executed across indices and types. The query can either be provided using a simple query string as a parameter, or using a request body"________ www.elasticsearch.org

在 Elasticsearch 中的 RESTful 接口方式中，完成信息检索功能的关键词是_search，通过 POST 的方式发送到 Elasticsearch，其后再跟"?q=查询词"等。其形式化表示方式是：

```
http://ipaddress:port/index_name/type_name/_search?q=
```

例如，可以以"curl -XGET 'http://localhost:9200/_search?q=hello+world'"的方式完成简单的检索，但 Elasticsearch 的信息检索功能很强大，其功能远远不止于此。信息检索时，可简单地使用基于 Lucene 的通用检索语法，也可以使用灵活的基于 JSON 格式的 Query DSL(Domain Special Language)来构造各种检索请求。在检索表达式中，有时也可同时包含查询条件和过滤器，可以使用复合查询，可以改变查询结果的排序，也可以在 Elasticsearch 中使用脚本。需要说明的是，第 2 章中介绍的文档索引是对其进行检索的前提和基础。在处理字段、类型和查询时可以指定分析器。和 Lucene 类似，索引和检索时使用的分析器(是带有零个或多个过滤器的分词器，如中文分词工具)应该是一样的，否则可能导致检索失败。

## 3.1 实验数据集描述

首先介绍本书后续章节中可能用到的 3 个主要数据文件的结构：

(1) 索引文件 whale 下类型文件为 log(日志信息)的结构如下(部分数据略)。

```
_index: whale                        //索引文件名称
_type: log                           //type 名称
_id: xxx                             //某 document 的 ID 号
_version: x                          //版本
_score: x                            //评分
_source: {                           //数据字段描述
    event_type:                      //事件类型
    @ timestamp: xxx                 //时间戳
    login: true or false             //是否为登录用户
    uid: xxx                         //登录用户 ID 号
    userAgent: xxx                   //用户使用的客户端代理软件及版本号
    browser: xxx                     //浏览器名称
    device: xxx                      //设备类型(如 PC 机或智能手机)
    os: xxx                          //客户端使用的操作系统,如 windows 7.0
    size: 0                          //HTTP 响应大小
    http_method: xxx                 //HTTP 方法,如 GET、POST 等
    ip:  xxx                         //客户端 IP 地址
    location: xxx                    //客户端所在地理位置
    reqTime: xxx                     //发布请求时间
    respTime: xxx                    //返回结果时间
    statusCode: xxx                  //网络状态码
    referer: xxx                     //是 header 的一部分,当浏览器向 web 服务器发送请
                                     //求时一般会带上 referer,告诉服务器是从哪个页面
                                     //链接过来的,服务器借此可以获得一些信息,用于分
                                     //析处理
    refererDomain: xxx               //referer 的域名
    uri: xxx                         //uri 标识
}
```

(2) 索引文件 page 下的类型文件 pages(采集到的网页信息)的结构如下(部分数据略):

```
_index: page                         //索引文件名称
_type: pages                         //type 名称
_id: xxx                             //某个 ID 号
_version: x                          //版本
_score: x                            //评分
_source: {                           //数据字段描述
    url: xxx                         //网页 URL
    title:  xxx                      //网页标题
    summary: xxx                     //针对该网页所抽取出的自动文摘
    category: xxx                    //网页分类
    spiderID: xxx                    //网页采集器 ID
    domain: xxx                      //该网页所属的类别域
    gatherTime: xxx                  //采集时间
    keywords: xxx                    //针对该网页内容所抽取出的关键词列表
    my_boost: xxx                    //网站权重
}
```

(3) 索引文件 baike 下类型文件 baike(采集到的百度百科信息,采集方法见本书第 9 章)的结构如下(部分数据略):

```
_index: baike                //针对百度百科数据的索引文件名称
_type: baike                 //针对百度百科数据的类型文件名称
_id: xxx                     //ID 号
_version: x                  //版本号
_score: x                    //排序分值
_source: {                   //数据字段描述
    title: (略)               //标题
    url:(略)                  //URL,如 http://baike.baidu.com/view/6505879.htm
    content: (略)             //网页内容
    lastModifyTime: (略)      //更新时间
    taglist: (略)             //内容分类,如"历史人物"等
}}
```

## 3.2 简单检索

如果需要构造一个简单的查询语句,含对结果分页、控制返回数据集大小、指定返回字段、限制结果分数、使用 script_fields、选择合适的搜索类型等,可以使用 Elasticsearch 的简单检索(对结果分页、排序、限制结果分数、使用脚本等将在后面介绍)。

在 Elasticsearch 启动的前提下,可以在浏览器中直接使用 URL 输入(如果同时提供有 &pretty=ture 子句,则输出结果是有缩进的),如:

```
http://localhost:9200/index_file_name/type_name/_search?q=field_name:
Hello&pretty=true
```

上述方法是可以在指定的索引文件 index_file_name、指定的类型文件 type_name 中,在指定的字段 field_name 中,查找包含 Hello 字符串的结果集。可以为查询指定明确的索引名和类型名,但也不是必需的。如果只给出索引名没有指定类型名,则检索该索引下的所有类型文件;也可以指定多个索引,查询其中的所有或特定的类型文件。下面列出几种不同的情况:

(1) 查询指定索引和指定类型下的信息(指定一个 index 和一个 type 名):

```
curl -XGET 'localhost:9200/page/pages/_search?q=field_name:Hello&pretty=true'
```

(2) 查询指定索引下所有类型中的信息(指定一个 index 名,没指定 type 名):

```
curl -XGET 'localhost:9200/page/_search?q=field_name:Hello&pretty=true'
```

(3) 查询所有索引中的信息(没指定 index 和 type 名):

```
curl -XGET 'localhost:9200/_search?q=field_name:Hello&pretty=true'
```

(4) 查询多个索引下所有类型中的信息(指定多个 index 名,没指定 type 名):

```
curl -XGET 'localhost:9200/page, whale/ _search?q=field_name:Hello&pretty=true'
```

(5) 查询多个索引下多个类型中的信息(指定多个 index 名和多个 type 名):

```
curl - XGET 'localhost: 9200/page, whale/pages, log/ _search? q = field _ name:
Hello&pretty=true'
```

在检索过程中,可以控制结果的规模以及从哪个结果开始返回,在请求中可以设置相应的属性,其中:

- from——该属性指定了从那个结果开始返回(默认是 0)。
- size——该属性指定了查询的结果集中包含的最大文档数(默认是 10)。

基于上述内容的查询体(不含 HTTP 方法、索引 index 名、类型 type 名等信息,限于篇幅,本章后文多数采取这种表示方法)如代码段 3.1 所示。

```
//代码段 3.1: 含有 from、size 的查询体
{
  "from": 18,
  "size": 30,
  "query": {
    "term": {
      "title": "中国"
    }
  }
}
```

## 3.3 基本检索

基本检索涉及 term 查询、terms 查询、match 查询、match_all 查询、query_string 查询、prefix 查询、range 查询、more_like_this 查询等。在介绍各种查询前,先介绍如何设置排序权重以及控制返回结果集的方法。

### 3.3.1 设置不同字段的排序权重

信息检索结果集合的排序策略对用户和网站开发者双方来说都是非常重要的,因为很少有人对排名很靠后的检索结果给与充分的重视,Top-10 的检索结果对所有用户来说都很重要。有统计表明,约有 58%的用户只对检索结果中排名前 10 位的内容感兴趣,而只有不超过 12%的用户才会对排名 30 位以后的内容感兴趣。

其实,在 Elasticsearch 的检索结果中,也可以设置不同字段对应的排序权重,在设置好权重以后,在检索结果排序时会用到的。例如,在 baike 索引的 baike 类型文件下,在 title、content 两个字段中搜索给定的关键字,如果需要设置在 title 字段中出现其权重是 2,而在

content 中出现其权重默认是 1,可以参照图 3.1 中的方法(图左侧是代码实现,图右侧是针对 baike 数据集的检索结果)。

```
▶ 历史
▼ Query
http://ip.hebust.edu.cn/es/
baike/baike/_search
POST
{
  "query": {
    "multi_match": {
      "query": "袁世凯",
      "fields": [
        "title^2",
        "content"
      ]
    }
  }
}
```

```
took: 88,
timed_out: false,
▼ _shards: {
    total: 5,
    successful: 5,
    failed: 0
},
▼ hits: {
    total: 237435,
    max_score: 9.340397,
    ▼ hits: [
        {
            _index: "baike",
            _type: "baike",
            _id: "M4kJGouKSBKDi71EZMQEEA",
            _score: 9.340397,
            ▼ _source: {
                title: "袁世凯",
                url: "http://baike.baidu.com/view/8447.htm",
                content: "袁世凯出身河南项城的一个大家族他的叔祖袁甲三官至漕运总
```

图 3.1 设置不同字段的权重

## 3.3.2 指定返回的字段子集

在默认情况下,检索是返回完整的 JSON 格式的数据。用户可以通过_source 来引用想要返回的检索结果(即搜索结果 hits 中的_source 字段,如图 3.1 右侧所示)。也就是说,如果不想返回完整的源文档结构,可以返回指定的部分字段子集,这有点类似于针对 RDBMS 的 SQL 查询语句中的"Select 部分字段名 from 某数据表"中的"部分字段"列表部分。

代码段 3.2 是简化了的_source 字段,它仍会返回一个叫做_source 的字段,但是这里仅包含指定的几个字段。代码段 3.3 则是在排序后返回指定数量的指定字段子集的检索集,而针对实际数据集 baike 的运行效果如图 3.2 所示(有关排序的其他内容参见本章最后部分)。在图 3.2 的例子中,针对给的检索词"世界历史人物"进行了分词处理,也就是说,会检索到在指定字段中包含"世界"、"世界历史"、"历史人物"等特征词的结果集。

```
//代码段 3.2: 通过_source 参数指定返回的检索字段集
{
  "query": {
    "match_all": {}                      //查询所以内容
  },
  "_source": [                           //指定返回的结果集字段列表
    "url",
    "taglist"
  ]
}
```

```
//代码段 3.3: 通过 match 参数指定字段和检索词,排序后返回指定数量的指定字段的检索集
{
   "query": {
      "match": {
```

```
        "字段名称": "检索词"
      },
    "sort": {                        //按照这里指定的 lastModifyTime 字段升序排序
      "lastModifyTime": {
          "order": "asc"
      }
    },
    "_source": ["url", "taglist"],   //显示的结果集
    "size": 10                       //返回的结果集数量
    }
}
```

图 3.2　指定字段子集的检索

如果需要匹配全部字段时，可用_all 参数来替代特定的字段子集，如代码段 3.4 所示。

```
//代码段 3.4: match 检索子句和_all 参数的使用
{
  "query": {
    "match": {
      "_all": "检索词"                //匹配全部字段的检索
    }
  }
}
```

如果在指定的字段中匹配特定的检索词，也可以采用代码段 3.5 中的方法来实现。

```
//代码段 3.5: match_phrase 子句及其应用
{
    "query": {
```

```
        "match_phrase": {
          "taglist": "世界"          //taglist 是 baike 中的一个字段
        }
      }
    }
```

### 3.3.3 Term 查询、Terms 查询、Wildcard 通配符查询

term 查询仅匹配在给定字段有某个词项的文档，如代码段 3.6 所示即是 term 查询，term 查询中的词项不再被解析。另外，如果希望提升该 term 的重要性，可以在代码中增加 boost 属性以便提升其重要性，实现方法如代码段 3.6 所示。

```
//代码段 3.6：含有 boost 的 term 查询
{
  "query": {
    "term": {                    //term 查询
      "title": {                 //查询字段，给定值及 boost
        "value": "中国",
        "boost": 10              //设置 term 的 boost 属性
      }
    }
  }
}
```

terms 查询允许匹配包含某些词项的文档。例如，如果想查询在文档的 title 字段中包含字符串“中国”或“日本”的文档，可以采用类似代码段 3.7 中的方法。注意这里的 minimum_match 的用法。其值设为 1，表示至少应匹配一个词项；如果其值为 2，则查询仅匹配在文档中同时包含这两个词项的文档。

```
//代码段 3.7：terms 和 minimum_match 的用法
{
  "query": {
    "terms": {
      "title": [
        "中国",
        "日本"
      ],
      "minimum_match": 1          //设置最小匹配集的大小
    }
  }
}
```

wildcard 通配符查询允许在要查询的内容中使用通配符 * 和?（通配符 * 表示任意多个任意字符，? 表示一个任意字符）。除此之外，它和 term 查询相似。实现时，只需把代码段 3.6 中的 "term"换为"wildcard"，在查询字段中根据需要嵌入 * 或? 即可，此处不再赘述。

### 3.3.4 Match、Match_all、Match_phrase 查询

match 查询子句可接受文字、数字和日期等类型的数据，match_all 查询是查询指定索引下的所有文档（相当于 SQL 语句的"select * from 某数据表"，如代码段 3.8 所示）。也可以利用 size 指定返回结果集的大小（如代码段 3.1 所示的那样，如果没有指定 size 的值，相当于 SQL 语句的"Select top 10 * from 某数据表"）。

```
//代码段 3.8: match_all 相当于 select * from 某数据表
{
  "query": {
    "match_all": { }
    }
}
```

代码段 3.9 做了一次 match_all 查询并且返回第 11 到第 15 个文档。

```
//代码段 3.9: 匹配所有文档且返回 top11-top15 的记录集
{
    "query": {
      "match_all": { }
    },
    "from": 10,            //起始数据
    "size": 5              //返回的检索集合的大小
}
```

代码段 3.10 做了一次 match_all 查询并按指定字段（lastModifyTime）的升序排序，返回满足条件的 3 条记录（在 RDBMS 中，相当于 SQL 语句"select top 3 * from 某数据表 order by lastModifyTime desc"）。

```
//代码段 3.10: 匹配所有文档且检索结果按指定的字段排序，返回前 3 个记录集
{
  "query": {
    "match_all": { }
  },
  "sort": {
    "lastModifyTime": {        //指定的排序字段
      "order": "asc"           //排序策略
    }
  },
  "size": 3                    //返回的结果集大小
}
```

match_phrase 查询对查询文本分析后构建一个短语查询，其中的 slop 参数定义了在查询文本的词项之间应间隔多少个未知单词才视为短语匹配成功。代码实现见代码段 3.11。

```
//代码段 3.11: match_phrase 查询
{
  "query": {
    "match_phrase": {
      "title": {
        "query": "中国 and 日本",
        "slop": 2              //定义了在查询文本的词项间的间隔词数量
      }
    }
  },
  "_source": [
    "title"
  ]
}
```

### 3.3.5 Query_string 查询

与其他的查询类型相比，query_string 查询支持 Lucene 所有的查询语法。对于给定的内容，query_string 查询使用查询解析器来构造实际的查询。示例代码如代码段 3.12 所示[Rafa, 2015]。

```
//代码段 3.12: query_string 查询
{
  "query": {
    "query_string": {
      "query": "title:中国^2+title:日本 -content:美国",
                                  //注意这里的加号和减号前空格不能省略
      "default_field": "title"    //查询所作用的默认字段，默认为_all
    }
  }
}
```

Tips：代码段 3.12 中出现的查询是典型的 Lucene 查询模式，其中，“title:中国^2”是指在 title 字段中要包含“中国”字符且其权重为 2，请注意这个表示方法在图 3.1 左侧图中亦有使用；“+title:日本”是指在 title 字段中还要同时包含“日本”字符串，只不过该字符串的权重为 1，这些权重会影响到最终结果排序；“-content:美国”是说在 content 字段中不能有“美国”字符串。

### 3.3.6 Prefix、Range 查询

prefix 查询能够找到某个字段以给定前缀开头的文档。同样,这里也支持 boost 属性来影响其排序结果。基于 prefix 查询实现的方法见代码段 3.13。

```
//代码段 3.13: prefix 查询
{
  "query": {
    "prefix": {                    //prefix 查询
      "title": {                   //在 title 中查询,可指定值和权重
        "value": "中华",
        "boost": 2
      }
    }
  },
  "_source": [
    "title"
  ]
}
```

range 查询是范围查询。range 查询只作用在单个字段上,并且查询的参数要封装在字段名称中,它也支持如下参数(注:这些参数在其他查询中也可以使用,参见本章其他内容)。

- from——范围下界。
- to——范围上界。
- include_lower——是否包含下界,默认是 true。
- include_upper——是否包含上界,默认是 true。
- boost——查询的权重。

有关 range 查询的用法见代码段 3.14。

```
//代码段 3.14: range 查询,其中 gatherTime 字段是指抓取网页时间,参见 3.1 节对 pages
类型文件的说明
{
  "query": {
    "range": {                     //range 查询
      "gatherTime": {              //指定查询字段及其范围
        "from": "2015-1-1",
        "to": "2015-2-28"
      }
    }
  }
}
```

下面的代码段 3.15 展示了在类型文件 pages 中，对日期型或时间型字段(这里是对 gatherTime 字段)进行范围检索的方法，它表示查找的是在从现在开始两个小时之前的检索集(注：当然这里还能改为"now-3d"等，意为从现在开始往前 3 天起算)，到当前时间的筛选数据子集。

```
//代码段 3.15：在 range 子句中设定 gatherTime 的上下限
{
  "query": {
    "range": {
      "gatherTime": {
        "from": "now-2h",
        "to": "now"
      }
    }
  }
}
```

### 3.3.7 More_like_this、Fuzzy_like_this 查询

more_like_this 查询得到与所提供的文本相似的文档。在这里可以使用的部分参数如下[Rafa, 2015]：

- fields——查询所作用的字段的数组，默认是_all。
- like_text——指明文档应比较的内容。
- precent_terms_to_match——指明一个文档必须匹配多大比例词项才视为相似，默认是 30%。
- min_term_freq——文档中词项出现的最低频次，低于该值的词项将被忽略，默认是 2。
- min_doc_freq——词项应至少在多少个文档中出现才不会被忽视，默认是 5。
- min_word_len——指明单个单词的最小长度，低于该值的单词将被忽视，默认是 0。
- max_query_terms——指明在生成的查询中查询词项的最大数目，默认是 25。
- max_doc_freq——指明出现词项的文档的最大数目，以避免词项被忽视，默认是无界的。
- max_word_len——指明单个单词的最大长度，高于该值的单词将被忽视，默认无界。
- stop_words——指明忽略词集。
- boost——提升一个查询的权重时使用的权重，默认是 1。
- analyzer——指明用于分析内容的分词器。

more_like_this 查询示例代码如代码段 3.16 所示。

```
//代码段 3.16: more_like_this 查询
{
  "query": {
    "more_like_this": {                    // more_like_this 查询
      "fields": [                          //查询字段
        "title",
        "keywords"
      ],
      "like_text": "中国和平发展",          //指定待比较的内容
      "min_term_freq": 1,                  //词项出现的最低频次
      "min_doc_freq": 1                    //词项应至少在多少个文档中出现才不会被忽视
    }
  },
  "_source": [
    "title"
  ]
}
```

类似地,fuzzy_like_this 查询得到与给定内容相似的所有文档,其查询是基于模糊串并选择其产生的最好的区分词项。该查询支持的部分参数如下[Rafa, 2015]:

- fields——字段数组,指明在哪些字段上执行,默认是_all。
- like_text——指明文档应比较的内容。
- ignore_tf——指定忽略词项的频次,默认为 false。
- max_query_terms——指明在生成的查询中查询词项的最大数目,默认是 25。
- min_similarity——指明区分词项应具有的最小相似度,默认是 0.5。
- prefix_length——指明区分词项的共同前缀的长度,默认是 0。
- boost——默认是 1。
- analyze——指明用于分析给定内容的分析器。

有关 fuzzy_like_this 示例代码如代码段 3.17 所示。

```
//代码段 3.17: fuzzy_like_this 查询
{
  "query": {
    "fuzzy_like_this": {
      "fields": [
        "title",
        "keywords"
      ],
      "like_text": "中国和平发展",
      "min_similarity": 0.8,             //指明区分词项应具有的最小相似度
      "prefix_length": 0.2               //指明区分词项的共同前缀的长度
```

```
      }
    },
    "_source": [
      "title"
    ]
}
```

### 3.3.8 跨字段检索

Elasticsearch 支持使用 multi_match 子句在多个字段中进行检索。multi_match 子句是跨字段的检索(只需写明需要检索的多个目的字段即可)。下面的代码段 3.18 提交的检索词是“中国”,而通过 multi_match,会同时从“字段 1”和“字段 2”中匹配包含指定检索词的内容。

```
//代码段 3.18: 利用 multi_match 在多个字段中检索
{
  "query": {
    "multi_match": {
      "query": "中国",
      "fields": [                          //指定在哪些字段集合中进行检索
        "字段 1",  "字段 2"
      ]
    }
  }
}
```

## 3.4 Filter 概述

Elasticsearch 在执行带有 filter 的查询时,会打开索引的每个 segment 段文件,然后去判断里面的文档是否符合该 filter 要求,并且这个匹配的结果用一个很大的只有两个状态的数组 BitSet 来存储,如果一个文档和 filter 查询匹配,那么其对应的 bit 位就设置为 1,否则设置为 0。下次如果面对同样的 filter 查询,直接使用内存里面的 Bitset 来进行判断即可,而不需要再打开索引的 segment 文件了,这样就可以大大提高了查询处理的速度。

可见,在 Elasticsearch 的检索过程中可以使用 filter 子句,其主要作用有两点:首先,它能够过滤满足条件的部分结果;其次,filter 子句可很好地将数据缓存在内存中,这样可大大加快下一次的检索速度(需要添加_cache 参数),因此,建议在可能的情况下尽量用 filter 代替一般的查询以便进一步提高效率。这样做的代价是存储空间的消耗和第一次执行过滤时的查询时间。一般地,应该把经常使用的过滤结果缓存,对于 and、bool、or 过滤器也可开启

缓存功能。

在使用 filter 子句时，既可以在 query 属性中直接使用 filter 子句，也可以最后再由 filtered 子句将它们合并，代码段 3.19 是由 filtered 子句将普通的 query 查询和 filter 子句结合了起来，表示检索在字段中带有指定的查询词且时间字段是从现在往前 6 天范围的检索集合。针对实际索引类型文件 pages 中的检索结果如图 3.3 所示。

```
//代码段 3.19：通过 filtered 子句将多个查询条件合并在一起的形式化方法
//此例中一部分为通常的 query，另一部分为 filter 部分
{
  "query": {
    "filtered": {
      "query": {                        //query 部分
        "match": {                      //match query 查询
          "字段": "检索词"
        }
      },
      "filter": {                       //filter 部分
        "range": {
          "时间字段": {                  //这里要替换为指定的时间类型字段
            "gte": "now-6d"
          }
        }
      }
    }
  }
}
```

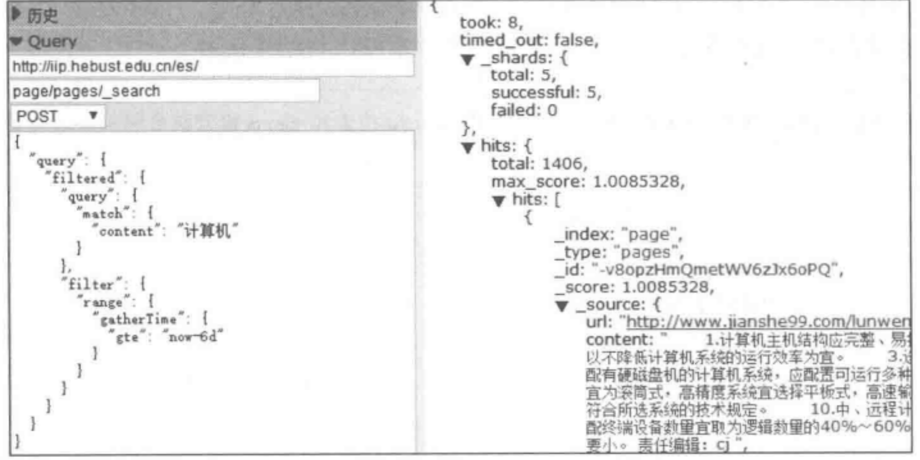


图 3.3 用 filtered 子句将多个查询体合并在一起

Tips: Lucene 在做大量 term 值查询时,如超过 1024 个 term,可能在某些版本下会出现"TooManyClauses[maxClauseCount is set to 1024]"的异常,因此建议在 term 过多的情况下采用 filter。

## 3.5 常用 Filter 及其应用

### 3.5.1 And Filter 及 Or Filter

逻辑关系操作符 and 和 or 可以用在对检索结果的 filter 操作上。and filter 和 or filter 可以将检索结果用指定的逻辑关系运算符连接起来。也就是说,这几个过滤器之间满足逻辑与,抑或满足逻辑或的关系。代码段 3.20 给出了 or filter 的形式化用法。

```
//代码段 3.20: 基于 or filter 过滤的形式化方法
"filter": {
    "or": [
        {
            "term": { "字段 1": "检索词 1" }
        },
        {
            "term": { "字段 2": "检索词 2" }
        }
    ]
}
```

代码段 3.21 展示了 and filter 的用法,它包含两个不同的检索条件:一个是 range 查询;另一个是 term 查询。and filter 返回两个条件都满足的结果集。一般情况下,检索结果默认是不缓存的。只有在添加了_cache 参数之后才可以将结果缓存。

```
//代码段 3.21: 基于 and filter 的过滤,将 range 检索和 term 检索结合起来并缓存结果
{"filter": {
    "and": [                        //and filter
        {
          "range": {
              "时间字段": {
                 "from": "now-10d",
                 "to": "now"
              }
          }
        },
        {
```

```
            "term": {
              "字段": "检索词"
            }
          }
      ],"_cache": true              //_cache 参数的用法
}}
```

### 3.5.2 Bool Filter

代码段 3.22 展示了 bool filter 的使用。它可将多个查询块通过 must、must_not 等连接词整合在一起。类似地,也可以使用_cache 参数,其含义同上。

```
//代码段 3.22: 基于 bool filter 的过滤
{
    "filter": {
      "bool": {
        "must": {                   //必须要满足的条件,这里以 term query 为例
            "term": {
              "字段 1": "检索词"
            }
        },
        "must_not": {               //必须要排除的条件,这里以时间范围为例
            "range": {
              "时间型字段": {
                "from": "now-1d",
                "to": "now"
              }
            }
        }
      }
    }
}
```

### 3.5.3 Exists Filter 和 Missing Filter

Elasticsearch 所处理的文档可以是非结构化的,也就是说,不同记录的结构有可能不一样。此时,可以用 exists filter 来过滤搜索结果,使其必须存在某个指定的字段,如代码段 3.23 所示。

```
//代码段 3.23: 基于 exists filter 的过滤
"filter": {
    "exists": {
        "field": "字段名称"
    }
}
```

而 missing filter 的作用与 exists filter 正好相反。它除了可以选择那些指定字段缺失的文档外，还可以指定 Elasticsearch 将哪些值作为空值处理，这适合于输入数据中包含空值等的情形。代码段 3.24 找到 type 类型文件 baike(文档名略)中 taglist 字段为空的文档集合。

```
//代码段 3.24: missing filter 过滤
{
  "filter": {
    "missing": {
      "field": "taglist",
      "null_value": [ ]
    }
  }
}
```

### 3.5.4 Type Filter

type filter 返回指定 type 的文档。当查询被定向到多个索引或者一个有大量数据类型的索引上时，可以使用这种类型的过滤器。实现方法见代码段 3.25。

```
//代码段 3.25: type filter 过滤
{
  "filter": {
    "type": {
      "value": "pages"                    //返回类型文件名为 pages 的结果集
    }
  }
}
```

### 3.5.5 Match_all Filter

match_all filter 的作用是选中全部数据，相当于 SQL 语句的“Select * ”部分。示例代码如代码段 3.26 所示。

```
//代码段 3.26: 基于 match_all filter 的过滤
"filter": {
    "match_all": {
  }
  }
```

### 3.5.6 Not Filter

not filter 的作用是过滤掉所有满足该过滤器的结果。代码段 3.27 找到类型文件 pages 中，从现在开始往前推 23 小时（即找到从那个时刻到现在）这段时间区间中采集的数据。

```
//代码段 3.27: 基于 not filter 的过滤
{
  "filter": {
    "not": {
      "range": {
        "gatherTime": {
          "from": "now-23h",
          "to": "now"
        }
      }
    }
  }
}
```

### 3.5.7 Query Filter

query filter 将一个可以含有逻辑运算符的查询用 filter 包裹起来并当作一个过滤器来用。代码段 3.28 在是索引的所有字段上查询同时含有“中国”和“美国”字符串的内容。如果指定在特定字段上进行检索，则需要给出以 JSON 格式表示的字段信息。如图 3.4 是在指定的 title 字段上完成类似检索的结果，代码实现参见图 3.4 的左侧。

```
//代码段 3.28: query filter 及其使用
{
  "filter": {
    "query": {
      "query_string": {
        "query": "中国 AND 美国"
      }
    }
  }
}
```

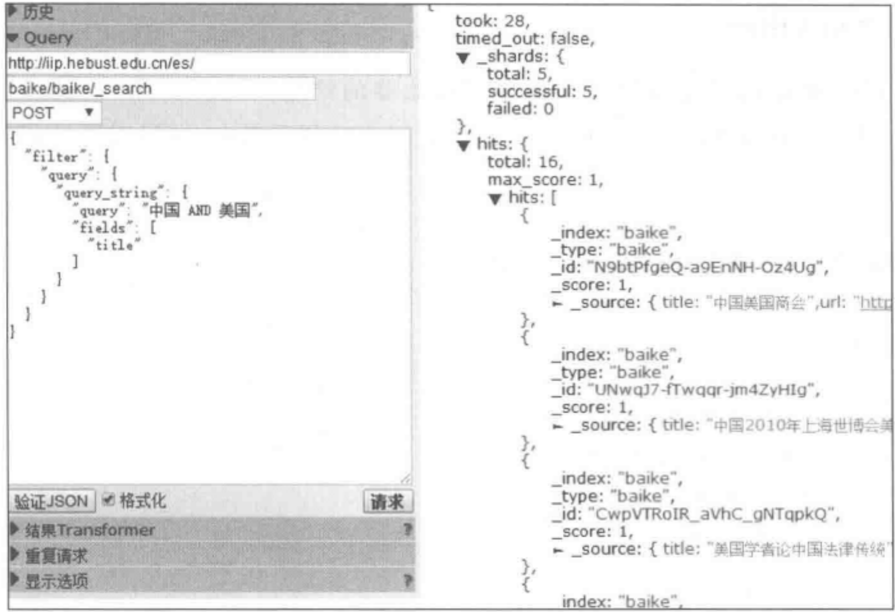


图 3.4　基于 query filter 的检索

## 3.6　复合查询

复合查询可以将各个子查询封装在一起，并通过下面的一些选项连接起来。

- should——封装在该选项中的查询可以匹配也可以不匹配。
- must——封装在该选项下的查询必须在返回的文档中被匹配上。
- must_not——封装在该选项下的查询不能在返回的文档中被匹配上。

代码段 3.29 完成了基于 must 的复合查询。针对实际数据集 baike 的执行效果如图 3.5 所示。

```
//代码段 3.29: 基于 must 完成复合查询
{
    "query": {
      "bool": {
        "must": [
          { "match": { "字段 1": "检索词 1" } },
          { "match": { "字段 2": "检索词 2" } }
        ]
      }
    }
  }
```

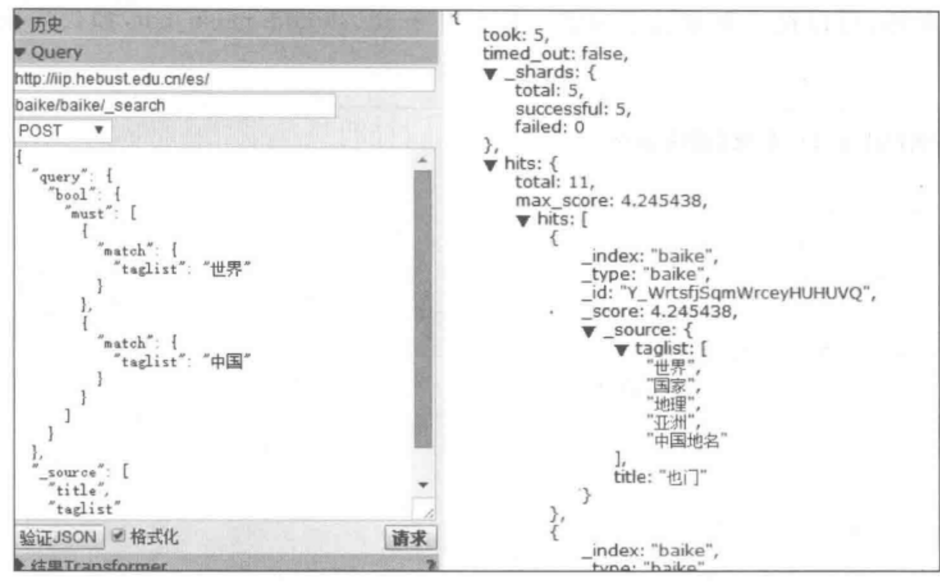


图 3.5 复合查询

代码段 3.30 完成的是逻辑“或”的布尔查询。这里采用了 should 选项完成复合查询。

```
//代码段 3.30: 基于 should 完成复合查询
{
      "query": {
        "bool": {
          "should": [
            { "match": { "字段 1": "检索词 1" } },
            { "match": { "字段 2": "检索词 2" } }
          ]
        }
      }
}
```

代码段 3.31 中采用的 must_not 语句指明对于一个文档,查询列表中的所有查询必须都不为真,这个文档才被认为是匹配的。

```
//代码段 3.31: must_not 复合查询
{
      "query": {
        "bool": {
          "must_not": [
            { "match": { "字段 1": "检索词 1" } },
            { "match": { "字段 2": "检索词 2" } }
          ]
        }
      }
}
```

类似地，可以在一个布尔查询里一起使用 must、should、must_not，如代码段 3.32 所示。

```
//代码段 3.32：较复杂的复合查询
{
  "query": {
    "bool": {
      "must": [
        {
          "match": {
            "taglist": "人物"
          }
        }
      ],
      "must_not": [
        {
          "match": {
            "taglist": "中国"
          }
        }
      ]
    }
  }
}
```

## 3.7 结果排序

在前面的例子中，有的地方已经用到了检索结果排序，可以在 query 查询体外指定排序方式，代码段 3.3 和代码段 3.10 给出了针对具体字段的排序（desc 和 asc 分别对于逆序和顺序排序）。也可以不针对某个具体的字段，而是针对默认的排序分数_score 进行顺序或逆序的排序。代码段 3.33 给出对基于 terms 的查询结果的排序方法。

```
//代码段 3.33：基于_score 对结果排序
{
  "query": {
    "terms": {
      "title": [
        "中国",
        "日本"
      ],
      "minimum_match": 3
    }
  },
```

```
  "sort": [                                  //排序方式
    {
      "_score": "desc"
    }
  ]
}
```

而对于指定字段的排序而言，除了代码段 3.3 和代码段 3.10 给出的方法外，也可以采用代码段 3.34 中的方法，这里指定_last 是将无值的结果放在检索集的最后（如果指定为_first 则是将其放在检索集的首位）。

```
//代码段 3.34：指定缺省值的排序策略
{
  "query": {
    "match_all": {}
  },
  "sort": [
    {
      "title": {
        "order": "desc",
        "missing": "_last"               //对于无值的，放在最后显示
      }
    }
  ]
}
```

## 3.8 扩展知识与阅读

有关 Lucene 的检索，可参考文献[Michael，2011]。传统的 Web 服务是基于 RPC 风格的，其实现技术主要包含 SOAP、WS 标准栈等。RPC 风格的 Web 服务用在分布式、开放的环境中会带来一些问题，如技术架构复杂、可伸缩性差等。而 RESTful 风格的 Web 服务可以解决上述问题。有关 RESTful 的内容，可以参阅文献[韩陆，2014]。文档[Rafa，2015]对 Elasticsearch 的搜索有更详细的说明，除此之外，还对扩展结构与搜索进行了说明。

## 3.9 本章小结

本章对基于 RESTful 的 Elasticsearch 信息检索方法进行了说明，分别从基本搜索、搜索结果过滤、复合搜索等多个方面进行了说明，并给出部分实际运行效果。

Chapter 4

# 信息统计分析与搜索提示

"The usual purpose of a full-text search engine is to return a small number of documents matching your query. Facets provide aggregated data based on a search query. In the simplest case, a terms facet can return facet counts for various facet values for a specific field. Elasticsearch supports more facet implementations, such as statistical or date histogram facets. The field used for facet calculations must be of type numeric, date/time or be analyzed as a single token. You can give the facet a custom name and return multiple facets in one request. Facets are deprecated and will be removed in a future release. You are encouraged to migrate to aggregations instead." ——— http://www.elasticsearch.org/

在面向关系型数据库管理系统中的 SQL 语句中，可以使用集函数完成简单的统计分析工作。其实，在 Elasticsearch 检索过程中，往往也需要进行信息统计和分析。和 SQL 语句不同的是，这里的统计和分析工作是由 facets(注：在新版本中由 aggregations 替代)完成的。Facets(或 aggregations)是 Elasticsearch 的重要功能之一，在聚合统计、数据分析、电商营销统计等许多领域都有实际应用，如分析在最近一小时发送的数据量大小、所有用户中使用 iPhone 浏览网站的用户有多少、在过去两天内哪些网站的点击量最大、统计交易额最多的 top-N 用户情况等。需要注意的是，用来进行统计分析的字段必须是数字类型、时间日期类型或可被处理为一个简单 token 的字段。facets 包含多个不同的功能，如 term facets、statistical 和 date histogram 等多种。本章首先介绍 facets 统计的使用。由于在新推出的 Elasticsearch 版本中已不再支持使用 facets，而是使用 aggregations 来完成统计分析任务，因此在本章也用相当的篇幅介绍基于 aggregations 的统计。为方便说明统计结果，本章基于 Elasticsearch 的 Head 工具，提供了部分实际运行效果，方便读者理解统计和聚合功能的实际效果。在本章最后介绍一种简单的搜索提示实现方法。

## 4.1 Facets 概述

基于 facets，可以得到各种统计结果(可以是以数组形式返回的)，如图 4.1 所示(注：此例是对 whale 索引中类型文件为 log 的统计，数组中的 term 和 count 分别代表各种状态码

及其数量)。

在图 4.1 中,除了返回必要的统计结果外,还能得到如下信息:

- _type——使用的统计类型。
- missing——在计算统计时那些数据不全的文档的个数。
- total——在计算中使用的词条的个数。
- other——在返回的计数中未包含的统计值的个数。

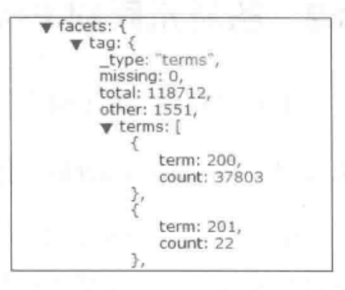


图 4.1 返回结果

下面通过一个例子来看看 facets 的执行情况。代码段 4.1 的前半部分是普通的 query 查询(细节不再赘述),后半部分就是 facets,terms 部分包含了需要的统计,统计字段是 content,size 是满足条件的统计数量(这里是要返回统计数据排名前两位的结果)。针对类型文件 pages 的效果如图 4.2 所示。

```
//代码段 4.1: 简单的 term 统计示例
{
  "query": {
    "query_string": {
      "query": "中国 AND 美国"
    }
  },
  "facets": {
    "my results": {                  //名称
      "terms": {
        "field": "content",
        "size": 2                    //返回结果集大小
      }
    }
  }
}
```

```
"facets": {
  "my results": {
    "terms": {
      "field": "content",
      "size": 2
    }
  }
 }
}
验证JSON  格式化  请求
结果Transformer
重复请求
```

```
▼ facets: {
  ▼ my results: {
    _type: "terms",
    missing: 0,
    total: 3505458,
    other: 3490963,
    ▼ terms: [
      {
        term: "美国",
        count: 7508
      },
      {
        term: "中国",
        count: 6987
      }
    ]
  }
}
}
```

图 4.2 基于 terms facets 的统计

## 4.2 各种不同的 Facets 统计

本节将对常用的不同类型的 facets 统计进行说明。

### 4.2.1 Terms Facets：指定字段的分布情况统计

terms facets 能够指定一个字段进行统计，Elasticsearch 将返回在指定字段中使用最多的词项及其分布。这里以日志信息索引类型文件 log 为例看看 term facets 的统计效果。

在代码段 4.2 中，通过 field 来设置需要进行统计的字段（这里是统计状态码 statusCode 字段）；size 返回聚合结果的前多少个结果；order 参数设置返回的聚合结果按哪个字段进行排序。处理结果如图 4.3 所示，图中显示对不同的状态码 statusCode 的分类统计结果（这里如果搭配 range query 的使用，就可以得到指定时间段的各状态码出现的次数）。

```
//代码段 4.2: term facets
{
    "query": {
        "match_all": {  }
    },
    "facets": {
        "tag": {                              //tag 是这个统计的名称
            "terms": {                        //terms facets
                "field": "statusCode",        //统计字段
                "size": 10,                   //返回结果集大小
                "order": "term"               //排序依据
            }
        }
    }
}
```

另外，在 terms facets 中，还可以使用如下参数：

- fields——在统计中允许指定多字段的数组，Elasticsearch 会返回多字段之间的统计结果。
- exclude——在统计计算中需要被排除的词项数组。
- regex——控制在计算中需要包含的词项的正则表达式。
- size——指定了最多返回多少个出现最频繁的词项，包含其他词项的文档将被在结果的 other 字段中计数。
- order——指定统计结果排序，可用值包括：

▶历史
▼Query
http://ip.hebust.edu.cn/es/
whale/log/_search
POST ▼

```
{
  "query": {
    "match_all": {}
  },
  "facets": {
    "tag": {
      "terms": {
        "field": "statusCode",
        "size": 3,
        "order": "term"
      }
    }
  }
}
```

```
{
    took: 2,
    timed_out: false,
  ▼ _shards: {
        total: 5,
        successful: 5,
        failed: 0
    },
  ▼ hits: {
        total: 17739,
        max_score: 1,
      ▸ hits: [ { _index: "whale", _type:
    },
  ▼ facets: {
      ▼ tag: {
            _type: "terms",
            missing: 0,
            total: 17739,
            other: 308,
          ▼ terms: [
                {
                    term: 200,
                    count: 2011
                },
                {
                    term: 302,
                    count: 137
```

图 4.3 term facets 效果

- ◆ count——默认排序规则，按照出现频率排序，从最频繁的开始。
- ◆ reverse_count——按照出现频率排序，从最不频繁的开始。
- ◆ term——按照字母顺序排序。
- ◆ reverse_term——按照字母降序顺序排序。

- all_terms——设置为 true 时，结果中将返回所有词项。
- script——指定在统计结果中用于对词项进行处理的脚本。
- _script_field——提供用于计算的实际词项的脚本。

其中的 size、order 参数的用法见代码段 4.1 和代码段 4.2，此处不再赘述。下面对其他部分参数的使用进行说明。

### 1. Fields 参数：跨字段统计

可以通过 fields 参数使用多字段统计，也就是说，可实现在多个字段间的使用最多的词项(即 term)统计。代码段 4.3 是在“字段 1”和“字段 2”中进行的词项统计并返回 Top-N 结果集。图 4.4 是对在指定类型文件 log 在两个不同字段(即 uri 和 referer)中统计出现次数最多的词项个数，并按它们出现次数的逆序排列。

```
//代码段 4.3：对多字段进行的统计
"terms": {
        "fields": ["字段 1", "字段 2"],      //字段列表
        "size": N                           //给出具体数字
    }
```

### 2. Exclude 参数：排除项

通过 exclude 参数可声明哪些 terms 是不被接受的。这些被声明为 exclude 的词项将

```
{
  "query": {
    "match_all": {}
  },
  "facets": {
    "tag": {
      "terms": {
        "fields": [
          "uri",
          "referfer"
        ],
        "size": 5
      }
    }
  }
}
```

验证JSON ✔格式化 请求

▸结果Transformer ?

```
▸ hits: { total: 62847,max_score: 1,hits: [ { _index: "whale",_type
▼ facets: {
    ▼ tag: {
        _type: "terms",
        missing: 0,
        total: 125694,
        other: 69579,
        ▼ terms: [
            {
                term: "-",
                count: 22327
            },
            {
                term: "http://iip.hebust.edu.cn/admin/home",
                count: 16381
            },
            {
                term: "http://iip.hebust.edu.cn/admin/homeContent
                count: 10699
            },
            {
                term: "/cas/login;jsessionid=E9F727751AF25808D6
                count: 3448
```

图 4.4 多字段统计

不会参与在最终的统计中,例如在代码段 4.4 中可以设置屏蔽字段 statusCode 取值为 200 和 302 的状态码的统计。

```
//代码段 4.4: exclude: 排除指定项
"tag": {
    "terms": {
        "field": "statusCode",
        "exclude": ["200", "302"]
    }
}
```

### 3. Regex 参数:正则表达式

regex 参数控制在计算中需要包含的词项的正则表达式,可以通过 regex 参数来设置正则表达式以匹配指定的内容。在代码段 4.5 中,regex 参数部分写相应的正则表达式,如“.*js”即是匹配所有以 js 字符结尾的值;regex_flags 参数遵循 Java Pattern API。处理结果如图 4.5 所示(图左侧是完整代码实现),此例统计在 whale 索引中类型文件名为 log 的文档中所有 uri 中以字符串 js 结尾的文档的个数。

```
//代码段 4.5: 正则表达式的使用
"terms": {
      "field": "tag",
      "regex": "正则表达式",
      "regex_flags": "DOTALL"
    }
```

图 4.5 regex pattern 示例

不仅仅是在 facets 中，在 Elasticsearch 随后推出的 aggregations 中也是支持正则表达式应用的。本章后文不再对正则表达式进行说明(详情可参见 4.5 节内容)，只在 4.3.9 节中给出一种在 aggregations 中使用正则表达式进行统计的方法。

**4. Scripts 参数：脚本**

可通过设置 scripts 参数，以便在统计时进行自定义的处理。代码段 4.6 中的 script 指定在统计中用于对词项进行处理的脚本。

```
//代码段 4.6: 脚本的使用
"terms": {
      "field": "统计字段",
      "script": "脚本"
   }
```

图 4.6 是在 log 类型中仅对于字段 statusCode 取值等于 304 的所有文档中的 uri 字段进行聚合，其聚合结果如图 4.6 右侧所示。

图 4.6 term scripts 示例

### 4.2.2 Range Facets：在某个范围的分布情况统计

range facets 可以允许指定区间并在这个区间内进行统计并获得指定字段的统计数据。代码段 4.7 可以统计 size 字段值在[10,50)以及[100,1000)的文档个数(包括下界但不包括上界),也可以使用 fields 指定多个字段名称。针对 whale 索引中的 log 类型文件,实际效果如图 4.7 所示。从图中可以清楚地看到在两个指定范围内的 size 字段的分布情况。

```
//代码段 4.7: range facets 统计
{
    "query": {
        "match_all": {}
    },
    "facets": {
        "range_facet": {          //名称
            "range": {
                "field": "size",//待统计的字段,这里的 size 是字段名,不是排序关键词
                "ranges": [
                    { "from": 10, "to": 50 },
                    { "from": 100, "to": 1000 }
                ]
            }
        }
    }
}
```

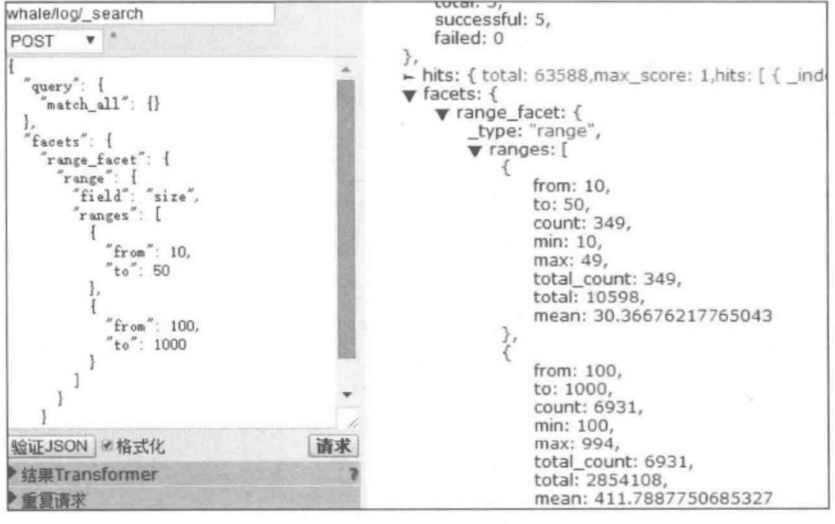

图 4.7 range facets

图 4.7 中显示的部分测度的含义如下：

- min 和 max——分别表示在给的范围内用于统计的字段取值的最小和最大值。
- count——表示给定字段取值属于指定范围的文档个数。
- total_count——表示给定字段中属于指定范围的取值总数(对于单字段，该取值与 count 取值相同，对于多字段，两者取值可能不一样)。
- total——给定字段中属于指定范围的取值的总和。
- mean——针对给定字段的指定范围进行 range 统计所计算出的平均值。

> Tips：每个范围都可以使用 to 或 from 属性或者同时使用这两个属性来指定。对于每个范围返回了部分统计信息，其中 from 表示范围的下边界，to 表示范围的上边界。

除上述形式外，range facets 还可以选取不同字段进行数据分析。如果需要对计算取值范围的字段之外的其他字段进行数据统计，可以使用 key_field 和 key_value，或者 key_script 和 value_script 两个允许使用脚本的属性形式(注意这里使用了下划线)。key_field 属性指定了应对哪个字段取值检查是否属于指定范围，value_field 属性指明应对哪个字段取值进行聚合计算。以 key_field 和 key_value 为例，这时会将 key 字段按指定的 ranges 进行分隔，然后计算其在 value 字段上的聚合结果。下面的示例代码段 4.8 展示了 log 类型文件中 HTTP 状态码(即 statusCode 字段)从 200 到 300、从 301 到 400、400 以上的各个段的发送数据量的统计情况。

```
//代码段 4.8: 基于 key-value 形式的 range facets
"facets": {
    "range1": {
        "range": {
            "key_field": "statusCode",
            "value_field": "size",
            "ranges": [
                { "from": 200, "to": 300 },
                { "from": 301, "to": 400 },
                { "from": 401 }
            ]
        }
    }
}
```

针对上述代码的返回结果如图 4.8 所示，下方的 count、min 等测度都是对 value 的计算结果。从结果中可以看到 HTTP 状态码在 200～300 发送的数据量最多。需要说明的是，这里的 key 和 value field 也是支持脚本 scripts 的，对应的名称分别为 key_script 和 value_script，具体实现方法不再赘述。

```
whale/log/_search
POST
"query": {
  "match_all": {}
},
"facets": {
  "range1": {
    "range": {
      "key_field": "statusCode",
      "value_field": "size",
      "ranges": [
        {
          "from": 200,
          "to": 300
        },
        {
          "from": 301,
          "to": 400
        },
        {
          "from": '401
        }
验证JSON  格式化  请求
结果Transformer
重复请求
显示选项
```

```
  successful: 5,
  failed: 0
},
hits: { total: 63608,max_score: 1,hits: [ { _index: "whale
facets: {
  range1: {
    _type: "range",
    ranges: [
      {
        from: 200,
        to: 300,
        count: 18623,
        min: 0,
        max: 717499,
        total_count: 18623,
        total: 176613982,
        mean: 9483.648284379531
      },
      {
        from: 301,
        to: 400,
        count: 42012,
        min: 0,
        max: 284,
        total_count: 42012,
        total: 783277,
        mean: 18.644125487955822
      },
      {
        from: 401,
        count: 2886,
        min: 0,
        max: 27649,
        total_count: 2886,
        total: 4685634,
        mean: 1623.5738045738046
```

图 4.8 基于 key_value 的统计

### 4.2.3 Histogram Facets

histogram facets 是一种可以根据其返回值(针对数值型或日期型的字段)生成可以将来用于柱状图的统计数据。代码段 4.9 是在 log 类型文件中,计算 size 字段每间隔 4000b 的统计分布情况。针对 log 实际数据集的返回结果如图 4.9 所示,下方的统计默认是采用了 count 测度。通过上面的分析,可以得到 HTTP 请求的 size 在各区段的数量分布情况。在这个统计数据基础上,可以利用一些可视化框架(如 D3、Echarts 等),对上述统计结果生成一张可视化的图表。

```
//代码段 4.9: histogram facets 对数字型字段的统计
"facets": {
    "histo1": {                            //图表名称
        "histogram": {
            "field": "size",               //待计算的字段名称
            "interval": 4000               //统计间隔
        }
    }
}
```

上述代码中的 field 参数不仅支持数字类型的字段,也支持时间类型的字段,此时可以通过 time_interval 来设置时间的步长值(可以设置为 1s、30m、1.5h 等)。代码段 4.10 是针对类型文件 log 中对时间字段的统计结果,这里指定的时间间隔是 30 分钟,处理结果如

```
whale/log/_search
POST ▼
{
  "query": {
    "match_all": {}
  },
  "facets": {
    "histo1": {
      "histogram": {
        "field": "size",
        "interval": 4000
      }
    }
  }
}
```

```
    total: 5,
    successful: 5,
    failed: 0
},
► hits: { total: 63616,max_score: 1,hits: [ { _inde
▼ facets: {
  ▼ histo1: {
      _type: "histogram",
    ▼ entries: [
        {
          key: 0,
          count: 50970
        },
        {
          key: 4000,
          count: 9195
        },
        {
          key: 8000,
          count: 1338
        },
        {
          key: 12000,
          count: 597
```

图 4.9 histogram facets 对数字型字段的统计

图 4.10 右侧所示。从结果中可以看到每隔 30 分钟内的访问量是多少(注：图 4.10 中显示的 key 值来源于代码段 4.10 中的@timestamp 字段,采用的是系统默认的格式,在实际工程中可以根据情况转换为可读的公历时间,转换方法见代码段 4.31 中的 format 子句)。

```
//代码段 4.10: histogram facets 对时间字段的统计
"histo1": {
    "histogram": {
        "field": "@ timestamp",            //统计时间类型字段
        "time_interval": "30m"             //时间间隔
    }
}
```

```
► 历史
▼ Query
http://iip.hebust.edu.cn/es/
whale/log/_search
POST ▼
{
  "query": {
    "match_all": {}
  },
  "facets": {
    "histo1": {
      "histogram": {
        "field": "@timestamp",
        "time_interval": "30m"
      }
    }
  }
}
```

```
{
took: 37,
timed_out: false,
▼ _shards: {
    total: 5,
    successful: 5,
    failed: 0
},
► hits: { total: 63624,max_score: 1,hits: [ { _index:
▼ facets: {
  ▼ histo1: {
      _type: "histogram",
    ▼ entries: [
        {
          key: 1421065800000,
          count: 53
        },
        {
          key: 1421067600000,
          count: 114
        },
        {
          key: 1421071200000,
          count: 81
        },
        {
          key: 1421073000000,
          count: 30
```

图 4.10 histogram facets 对时间字段的统计

如果要统计的内容不是可以由已知的字段直接得到的，而是需要进行一些简单计算才能得到的话，就需要在 key 和 value 中使用脚本（设置 key_script 和 value_script）。代码段 4.11 在 key_script 脚本中的语句是得到当前文档中“@timestamp”字段的日期（用 date 获取日期对象，dayOfMonth 方法得到对应的日期月份，类似地，minuteOfHour 方法可以得到对应的分钟数等）；这里的 doc 表示当前正在被处理的文档。同理，在 value_script 脚本中可以得到当前记录的 size 值。针对代码段 4.11 的处理结果如图 4.11 所示。图 4.11 中右侧显示的结果表示针对用户提出的请求，系统返回的 size 字段的统计结果，其中的 count、min 等均是对 value 的统计，其含义前面已有叙述。

> Tips：Elasticsearch 中的脚本可以用于很多功能。脚本中一般有如下几个对象可用：
> - doc（也可以是_doc）——用于访问基于计算分值或字段取值找到的当前文档。
> - _source——可访问当前文档的源，以及在其中定义的取值。
> - _fields——用于访问文档中的字段取值。

```
//代码段 4.11：在 key 和 value 中写入脚本
"facets": {
  "histo1": {
    "histogram": {
      "key_script": "doc['@timestamp'].date.dayOfMonth",  //得到时间戳中的日期
      "value_script": "doc['size'].value * 1"
                          //这里的 value 值乘 1 无实际意义，只是演示可进行脚本计算
    }
  }
}
```

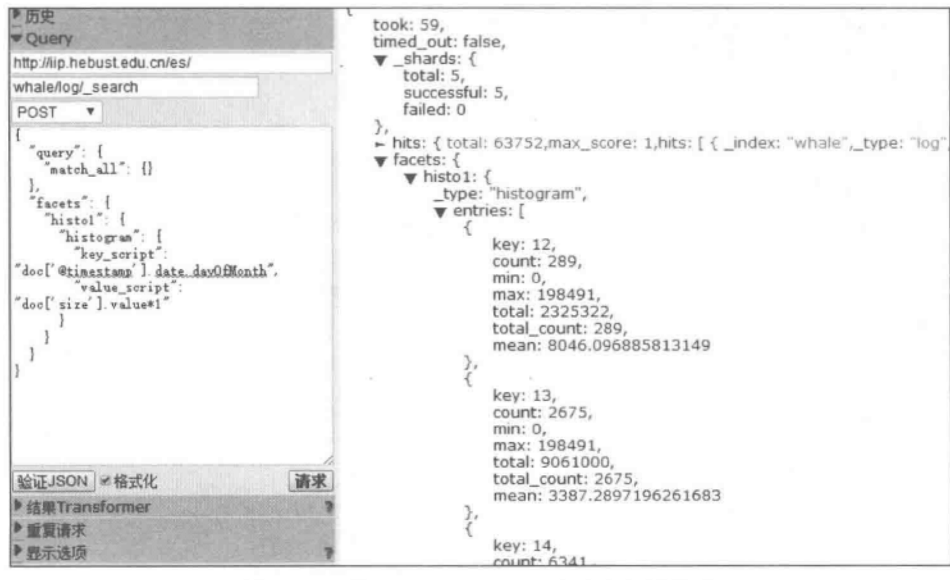


图 4.11　在 histogram facets 统计中使用脚本

更进一步地，在 script 中也支持指定的参数(参数写在"params"里)，参见代码段 4.12，其中 key_script 的作用是取出文档中的@timestamp 字段，得到其数据对应的小时值，再乘以因子 3，value_script 的写法与此类似，此处不再赘述。针对 log 类型文件的实际运行效果如图 4.12 所示。从图中可以看出，key 值都是 3 的倍数(这和我们在代码段 4.12 的设计思路是相符的)，而其下的 count、min、max、total 等都是对 value 的统计，其含义不再赘述。

```
//代码段 4.12: 在 script 脚本中写入参数
"facets": {
  "histo1": {
    "histogram": {
      "key_script": "doc['@ timestamp'].date.minuteOfHour * factor_x",
      "value_script": "doc['size'].value+factor_y",
      "params": {
        "factor_x": 3,              //因子倍数
        "factor_y": 6
      }
    }
  }
}
```

```
▼ facets: {
  ▼ histo1: {
      _type: "histogram",
    ▼ entries: [
        {
          key: 0,
          count: 1456,
          min: 6,
          max: 195768,
          total: 7851918,
          total_count: 1456,
          mean: 5392.800824175824
        },
        {
          key: 3,
          count: 1452,
          min: 6,
          max: 195768,
          total: 6652620,
          total_count: 1452,
          mean: 4581.694214876033
        },
        {
          key: 6,
          count: 1086,
          min: 6,
          max: 195768,
          total: 6059109,
          total_count: 1086,
          mean: 5579.290055248619
        },
```

图 4.12 在 histogram facets 脚本中使用参数后的结果

### 4.2.4 Date_histogram Facets

date_histogram facets 是一个增强型的专门针对于日期型字段进行统计的 facets。date_histogram facets 类型允许使用 year、month、week、day、hour、minute 等作为 interval 属性

的取值。可以在形式化的代码段 4.13 中，在 field 中填写一个日期类型的字段名称，在 interval 中写一个步长值。

```
//代码段 4.13: date_histogram facets
{
"query": {
  "match_all": {}
},
"facets": {
  "histo1": {
    "date_histogram": {
      "field": " 字段名 ",                 //统计字段
        "interval": "步长"                 //间隔
      }
    }
  }
}
```

相似地，date_histogram facets 也支持使用 value 参数。代码段 4.14 针对类型文件 log，用时间类型的字段作为 key_field，然后使用其他的字段作为 value_field(代码段 4.14 是对指定的 size 字段进行计算)，结果展示的是在@timestamp 时间段内，对 HTTP 请求的发送数据大小(即 size 字段)进行统计，内容包括计数、最大值、最小值、求和、求平均数等，如图 4.13 所示。

```
▼ facets: {
  ▼ histo1: {
      _type: "date_histogram",
    ▼ entries: [
        {
            time: 1421066884000,
            count: 44,
            min: 0,
            max: 0,
            total: 0,
            total_count: 44,
            mean: 0
        },
        {
            time: 1421066922000,
            count: 1,
            min: 4514,
            max: 4514,
            total: 4514,
            total_count: 1,
            mean: 4514
```

图 4.13　基于 value_field 方式的 date_histogram facets 统计

请注意图 4.13 和图 4.10 的不同。图 4.13 的统计值来源于类型文件 log 中的 size 字段并给出了诸如 count、min 等测度值，而图 4.10 结果对应的代码中没有指明 value 字段，只能给出 count 测度。另外，图 4.13 中的时间格式不是可读的有意义时间格式，可以对其进行转换，转换方法见代码段 4.31 中的 format 参数。

类似地，date_histogram facets 也支持对于 key 和 value 的 script 脚本(对应的参数名为 key_script 和 value_script)，其用法类似于代码段 4.11 和代码段 4.12。类似地，代码段 4.14

给出 key_field 和 value_field 在 date_histogram facets 中的应用。

```
//代码段 4.14: key_field 和 value_field 在 date_histogram facets 中的应用
"facets": {
  "histo1": {
    "date_histogram": {
      "key_field": "@ timestamp",          //统计@ timestamp 字段
      "value_field": "size",               //用 size 字段作为 value
      "interval": "second"
    }
  }
}
```

### 4.2.5 Statistical Facets

statistical facets 可以对数字类型的字段进行统计。具体地,可以完成计数、求和、求平方和、求平均数、求最大和最小值、求方差、求标准差等。代码段 4.15 完成了对 log 类型文件中 size 字段的统计。

```
//代码段 4.15: 对指定字段进行统计
"facets": {
  "stat1": {
    "statistical": {
      "field": "size"          //待统计的字段
    }
  }
}
```

针对类型文件 log 的实际运行结果如图 4.14 所示。这里,返回的统计字段及其含义如下:

- _type——统计类型(这里是进行 statistical 统计)。
- count——给定字段中包含指定取值的文档个数。
- total——给定字段中所有取值的总和。
- min——字段取值的最小值。
- max——字段取值的最大值。
- mean——指定字段中取值的均值。
- sum_of_squares——指定字段中取值的平方和。
- variance——指定字段中取值的方差。
- std_deviation——指定字段中取值的标准差。

相似地,statistical facets 也支持使用 script 和 fields 属性,其用法和 histogram facets 等类似。代码段 4.16 展示了对于类型文件 log 中的请求时间 respTime 和响应时间

```
历史
Query
http://iip.hebust.edu.cn/es/
whale/log/_search
POST
{
  "query": {
    "match_all": {}
  },
  "facets": {
    "stat1": {
      "statistical": {
        "field": "size"
      }
    }
  }
}
```

```
took: 6,
timed_out: false,
▼ _shards: {
    total: 5,
    successful: 5,
    failed: 0
},
► hits: { total: 64387,max_score: 1,hits: [ { _index: "whale",_typ
▼ facets: {
  ▼ stat1: {
      _type: "statistical",
      count: 64387,
      total: 182912951,
      min: 0,
      max: 717499,
      mean: 2840.836675105223,
      sum_of_squares: 11575141843529,
      variance: 171704164.25639454,
      std_deviation: 13103.593562698537
    }
  }
}
```

图 4.14　statistical facets 示例

reqTime 字段相加的统计结果(注：这两个字段值相加没有实际工程价值，这里只是示范相应代码的用法)，实际运行效果如图 4.15 所示。

```
//代码段 4.16: 在 script 脚本中可以完成对指定数值字段的运算
"facets": {
  "stat1": {
    "statistical": {
      "script": "doc['respTime'].value+doc['reqTime'].value"
                                              //在脚本中可以完成计算工作
    }
  }
}
```

```
历史
Query
http://iip.hebust.edu.cn/es/
whale/log/_search
POST
{
  "query": {
    "match_all": {}
  },
  "facets": {
    "stat1": {
      "statistical": {
        "script": "doc['respTime'].value +
doc['reqTime'].value"
      }
    }
  }
}
```

```
took: 90,
timed_out: false,
▼ _shards: {
    total: 5,
    successful: 5,
    failed: 0
},
► hits: { total: 64395,max_score: 1,hits: [ { _index: "whale",_type: "log"
▼ facets: {
  ▼ stat1: {
      _type: "statistical",
      count: 64395,
      total: 2515.833999999963,
      min: 0,
      max: 120.056,
      mean: 0.03906877863188078,
      sum_of_squares: 142182.02645999636,
      variance: 2.206440498460685,
      std_deviation: 1.4854092023616539
    }
  }
}
```

图 4.15　在 statistical 统计中使用脚本的示例

类似地，field 参数也支持对多字段的统计(此时 field 参数应改用 fields，后面的值也应该是一个数组类型，参见代码段 4.3)；脚本 script 也是支持参数的(可以用 param 参数指

定，参见代码段 4.12)，此处不再赘述。

### 4.2.6 Terms_stats Facets

terms_stats facets 是 terms facets 和 statistical facets 的组合形式，它提供了在一个字段上基于另一个字段获得的取值进行统计的能力。代码段 4.17 展示了针对类型文件 log 在不同的 HTTP 状态码(即 statusCode 字段)下响应数据量大小(即 size 字段)的变化情况，也就是说，对字段 size 进行统计，同时根据 statusCode 字段对统计值进行划分。针对该段代码在类型文件 log 上的实际运行效果如图 4.16 所示。

```
//代码段 4.17: 在不同的 statusCode 字段下统计 size 字段的变化情况
"facets": {
  "tag_price_stats": {
    "terms_stats": {
      "key_field": "statusCode",
      "value_field": "size"
    }
  }
}
```

图 4.16 terms_stats facets 示例

从上面的结果可以看出，在状态码为 200 的情况下，响应数据的大小平均约为 9381。terms stats facets 也可以指定 size 参数来控制返回结果的数量，可以通过指定 order 参数来选择返回结果的排序依据(order 参数可选的值有 term、reverse_term、count、reverse_count、total、reverse_total、min、reverse_min、max、reverse_max、mean、reverse_mean 等，默

认是按 count 进行统计的，有关排序参数的说明参见 4.2.1 节)。同样地，terms_stats facets 也是支持脚本 script 的，方法可参见代码段 4.11 和代码段 4.12，此处不再赘述。

## 4.3 Aggregations

### 4.3.1 概述

aggregations 是 facets 的升级版。在 Elasticsearch 的官方文档中已指出，新版本中将废弃 facets 功能(Elasticsearch V1.0 版后开始用 aggregations 取代 facets)，相应功能转而由 aggregations 代替。从本节开始，介绍有关 aggregations 的使用方法。其实，aggregations 的用法和 facets 十分类似，一般来说，aggregations 的通用形式如下：

```
//代码段 4.18: aggregations 形式化通用形式
"aggregations": {
  "<aggregation_name>": {
    "<aggregation_type>": {
      <aggregation_body>
    }
    [,"aggregations": { [<sub_aggregation>]+} ]?
  }
  [,"<aggregation_name_2>": { ... } ] *
}
```

简单说来，aggregations 和 facets 都可以看作是对查询结果的二次汇总，比如先查询出某个时间段的 HTTP 请求，然后统计每天的数据，但 facets 多数情况下只提供一级汇总，而 aggregations 能嵌套聚合并实现多级汇总。

在 aggregations 中会用到“桶”(buckets)的概念。所谓的“桶”，是满足某个条件的文档集合，它和关系型数据库中的 SQL 语句中的 grouping by 子句的作用相似(但又不一样)。例如，在校大学生要么属于本科生群的 buckets，要么属于研究生群的 buckets。另外，metrics(测度)是为某个桶中的文档计算得到的统计信息，它和关系型数据库中的 SQL 语句中的集函数如 count()、max()等的作用相似。可见，aggregations 聚合是由一个或多个 buckets、零个或者多个 metrics 组合而成的统计结果。每个文档中的值会被计算来决定它们是否匹配了某些 buckets 的条件，如匹配成功，那么该文档会被置入该 buckets 中。一个桶也能够嵌套在其他的桶中(即桶是可以嵌套的)。而对 metrics 而言，多数仅使用文档中的值进行简单计算。另外，aggregations 也支持排序等属性，由于它们多数和 facets 的用法差不多，因此不再单独列出，而是嵌入在相关的例子中进行说明。

代码段 4.19 给出在类型文件 log 中基于 terms aggregations 的例子。由于 Elasticsearch 允许在代码中将 aggregations 简写为 aggs，因此本章后续内容中也有多处采用这种简记

法。从代码中可以看出，aggregations 和 facets 有着很多相似之处。由于本章主要关注的是 aggregations 的返回结果，因此在命令行_search 参数的后面，可以通过输入“? search_type=count”来指定不返回搜索结果（即 hit 结果）而只返回统计结果，如图 4.17 左上角所示，也就是说，可以在 URL 输入框中输入类似的 URL：“http://localhost:9200/index_file_name/type_name/_search? search_type=count&q=field_name:查询词 &pretty=true”（注：这里的 IP 地址、索引文件名 index_file_name、类型文件名 type_name、查询字段 field_name、查询词等均应替换为符合实际要求的字符串）。本章后续截图多数都是基于这种方式（即只返回 facet 或 aggregations 的统计结果而不返回 hit 结果集）。

```
//代码段 4.19: 统计指定字段出现的次数
{
  "query": {
    "match_all": {}
  },
  "aggregations": {                    //可以简写为 aggs,后文同
    "my_agg": {                        //名称
      "terms": {                       //terms aggregations
        "field": "uri"                 //统计字段,此处是以 uri 为例
      }
    }
  }
}
```

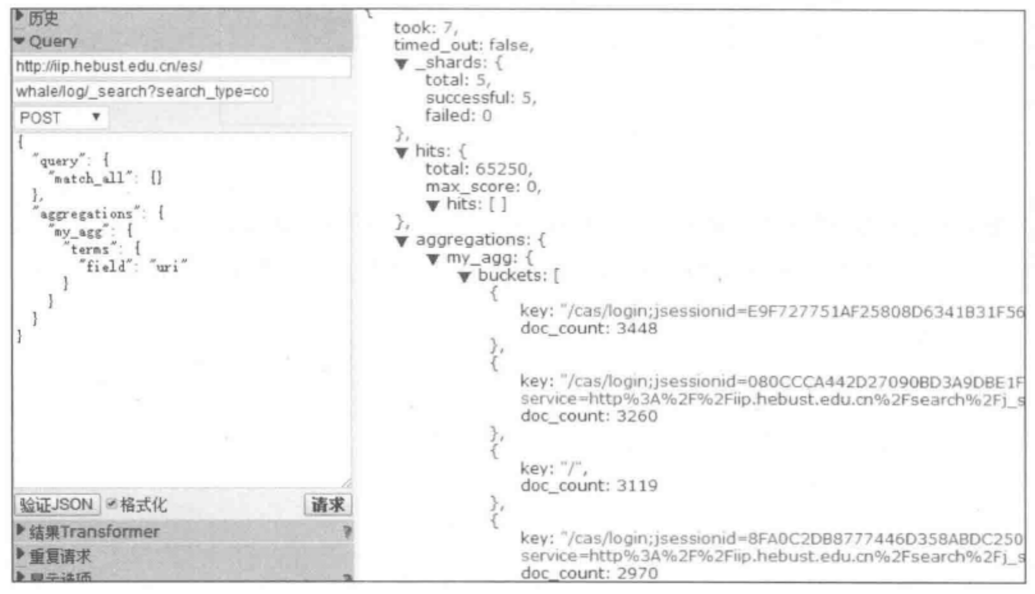

图 4.17 terms aggregations 示例

### 4.3.2 最值、求和、均值统计

在 aggregation 中可以方便地完成对最值、求和、均值等的统计。代码段 4.20 完成对指定字段的最小值统计，示例结果如图 4.18 所示（当统计最大值时，将代码段 4.20 中的统计函数换成 max 即可，此处不再赘述）。

```
//代码段 4.20: 最小值统计
{
  "query": {
    "match_all": {}
  },
  "aggs": {
    "min_size": {              //aggs 的名称
      "min": {                 //统计最小值。当统计最大值时这里换成 max 即可
        "field": "字段名称"    //统计字段,如 size 等
      }
    }
  }
}
```

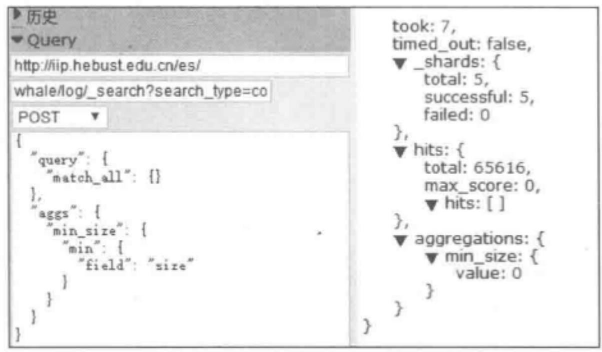


图 4.18 最小值统计

类似地，求和、求平均统计只需在相应函数处写上 sum 及 avg 即可。代码段 4.21 完成对指定字段的平均值统计，而针对返回 HTTP 响应大小 size 字段的实际效果如图 4.19 所示。

```
//代码段 4.21: 求均值
{
  "query": {
    "match_all": {}
  },
  "aggs": {
    "avg_aggs": {                 //统计的名称
      "avg": {                    //统计的测度函数
        "field": "字段名称"       //待统计的字段,如 size 等
```

```
            }
        }
    }
}
```

```
▶历史
▼Query
http://iip.hebust.edu.cn/es/
whale/log/_search?search_type=co
POST
{
  "query": {
    "match_all": {}
  },
  "aggs": {
    "avg_aggs": {
      "avg": {
        "field": "size"
      }
    }
  }
}
```


```
took: 5,
timed_out: false,
▼ _shards: {
    total: 5,
    successful: 5,
    failed: 0
},
▼ hits: {
    total: 65632,
    max_score: 0,
  ▼ hits: [ ]
},
▼ aggregations: {
  ▼ avg_aggs: {
      value: 2810.357462823013
    }
  }
}
```


图 4.19 均值统计

在完成上述计算时,同样支持脚本的使用。代码段 4.22 中使用了脚本,用来计算 reqTime 和 respTime 之和的最大值,示例显示如图 4.20 所示。

```
//代码段 4.22:脚本的使用
{
  "query": {
    "match_all": {}
  },
  "aggs": {
    "aggsname": {
      "max": {              //统计的测度
        "script": "doc['reqTime'].value+doc['respTime'].value"       //脚本
      }
    }
  }
}
```

```
▶历史
▼Query
http://iip.hebust.edu.cn/es/
whale/log/_search?search_type=co
POST
{
  "query": {
    "match_all": {}
  },
  "aggs": {
    "aggsname": {
      "max": {
        "script":
"doc['reqTime'].value+doc['respTime'].value"
      }
    }
  }
}
```


```
took: 18,
timed_out: false,
▼ _shards: {
    total: 5,
    successful: 5,
    failed: 0
},
▼ hits: {
    total: 65628,
    max_score: 0,
  ▼ hits: [ ]
},
▼ aggregations: {
  ▼ aggsname: {
      value: 120.056
    }
  }
}
```


图 4.20 最大值统计中脚本的应用示例

### 4.3.3 Stats Aggregations 及 Extended Stats Aggregations

stats aggregation 是一个多值统计，返回值包括最大值、最小值、求和、计数、均值等。代码段 4.23 演示了对相应字段进行多值统计的方法，针对类型文件 log 的返回结果如图 4.21 所示。同样，stats aggregation 也支持脚本 script 和参数，此处不再赘述。

```
//代码段 4.23：多值统计
{
  "query": {
    "match_all": {}
  },
  "aggs": {
    "multi_stats_size": {
      "stats": {
        "field": "size"          //统计字段，这里以 size 为例
      }
    }
  }
}
```

```
历史
Query
http://iip.hebust.edu.cn/es/
whale/log/_search?search_type=co
POST
{
  "query": {
    "match_all": {}
  },
  "aggs": {
    "multi_stats_size": {
      "stats": {
        "field": "size"
      }
    }
  }
}
```

```
took: 5,
timed_out: false,
_shards: {
    total: 5,
    successful: 5,
    failed: 0
},
hits: {
    total: 65636,
    max_score: 0,
    hits: [ ]
},
aggregations: {
    multi_stats_size: {
        count: 65636,
        min: 0,
        max: 717499,
        avg: 2810.20350112743,
        sum: 184450517
    }
}
}
```

图 4.21 多值统计

extended stats aggregation 是对一般的 stats aggregation 的功能扩展，可以在上述输出结果上添加平方和、方差和标准差等测度，参见代码段 4.24。同样地，extended stats aggregation 也支持脚本 script 和 script 参数，此处不再赘述。

```
//代码段 4.24：扩展的多值统计
{
  "query": {
    "match_all": {}
  },
  "aggs": {
```

```
      "extended stats of size": {
        "extended_stats": {          //采用相应的函数
          "field": "size"            //统计字段,这里以 size 为例
        }
      }
    }
  }
```

### 4.3.4 Terms Aggregations

和 facets 相似,terms aggregation 用于对指定字段的内容进行分布统计。代码段 4.25 给出了实现方法,而针对类型文件 log 的实际效果如图 4.22 所示。

```
//代码段 4.25: terms aggregations
{
  "query": {
    "match_all": {}
  },
  "aggs": {
    "Name Of Aggs": {             //名称
      "terms": {                  //terms aggsregation
        "field": "os"             //对用户登录系统时使用的操作系统分布情况进行统计
      }
    }
  }
}
```

```
▼ aggregations: {
  ▼ Name Of Aggs: {
    ▼ buckets: [
        {
          key: "Windows 7",
          doc_count: 33990
        },
        {
          key: "unknown",
          doc_count: 23366
        },
        {
          key: "Windows 8",
          doc_count: 4763
        },
        {
          key: "OS X 10.9 Mavericks",
          doc_count: 1941
        },
        {
          key: "Windows 8.1",
          doc_count: 1012
        },
        {
          key: "Windows XP",
          doc_count: 384
        },
        {
          key: "Android 4.3 Jelly Bean",
          doc_count: 113
```

图 4.22 对客户端使用的操作系统情况进行统计(未排序)

可以通过设置 size、order 等参数来指定返回结果的大小和排序方式等，可以按结果的 count 数排序（使用_count 参数），也可以根据 term 字段来排序（要将_count 换成_term，此时是按照字母升序排序，如图 4.23 所示）。比较图 4.22 和图 4.23 可以清楚地看到，图 4.22 给出的各个 key 是没有排序的，而图 4.23 给出的 key 则是按照字母顺序排序的，这就是 order 参数的作用。

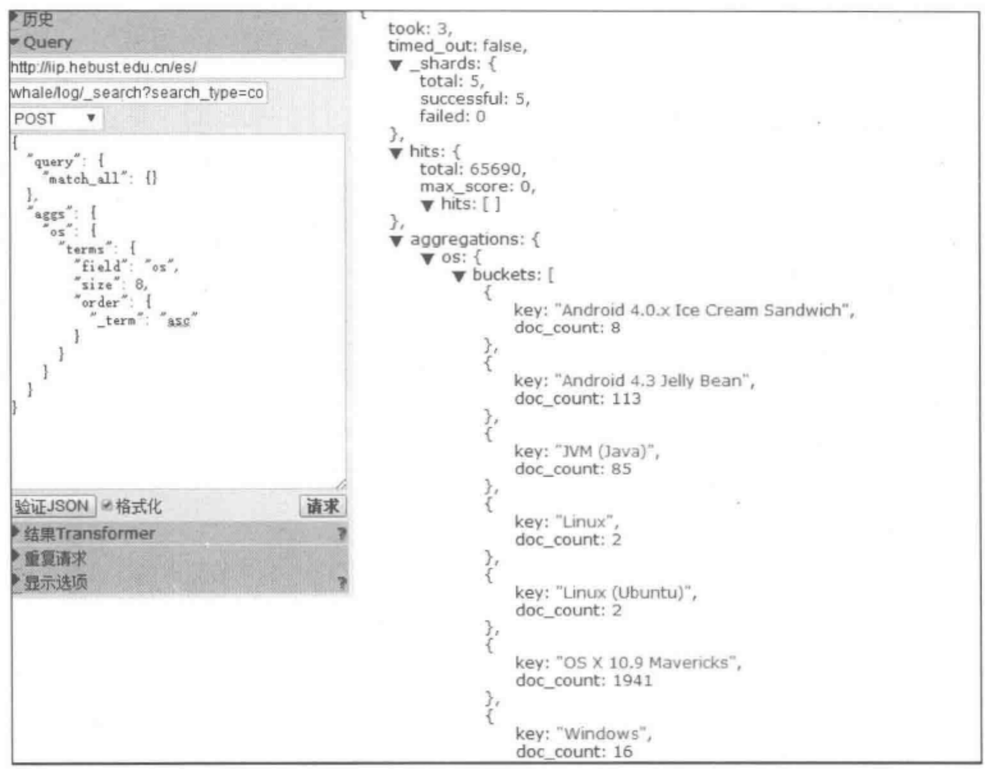


图 4.23 对客户端使用的操作系统情况进行统计（已排序）

另外，可以在 aggs 中嵌套子 aggs 来在 bucket 进行进一步的嵌套统计（即 bucket 嵌套）。代码段 4.26 演示了在类型文件 log 中针对不同的操作系统（os 字段），分析 HTTP 返回的数据大小（size 字段），并按照内层统计结果（均值）的倒序进行排列的方法。实际结果如图 4.24 所示。

```
//代码段 4.26: bucket 嵌套
{
  "query": {
    "match_all": {}
  },
  "aggs": {                                  //外层 aggs
    "The first layer: os": {                 //名称
      "terms": {                             //terms aggs
```

```
        "field": "os",        //统计字段
        "size": 5,            //返回结果集大小
        "order": {
          "avg_size": "desc" //对内层统计结果 aggs 的排序策略
        }
      },
      "aggs": {               //内层 aggs
        "avg_size": {         //名称
          "avg": {            //测度,是统计均值,而不是计数
            "field": "size"   //这里的 size 是统计字段,不是指返回结果集大小(注:重名)
          }
        }
      }
    }
  }
}
```

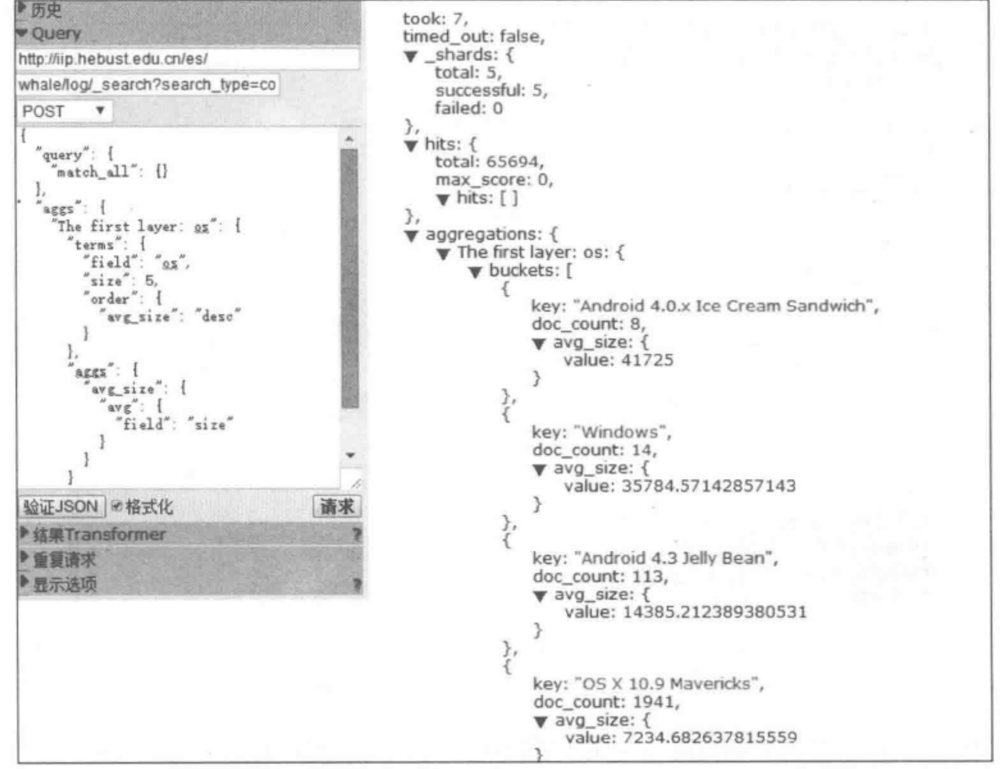


图 4.24 Bucket 的嵌套统计

另外,也可以在 terms aggs 中添加 min_doc_count 参数来控制最小返回的词项计数(相当于 Top-N 设置)。代码段 4.27 展示了对返回的使用操作系统情况的统计结果。由于这

里要求显示使用最少的操作系统数量不得低于 1000,因此针对类型文件 log 的实际统计结果见图 4.25。

```
//代码段 4.27: 用 min_doc_count 参数指定返回最小词项计数
{
  "query": {
    "match_all": {}
  },
  "aggs": {
    "tags": {                                  //名称
      "terms": {                               //terms aggs 统计
        "field": "os",                         //统计字段
        "min_doc_count": 1000                  //阈值
      }
    }
  }
}
```

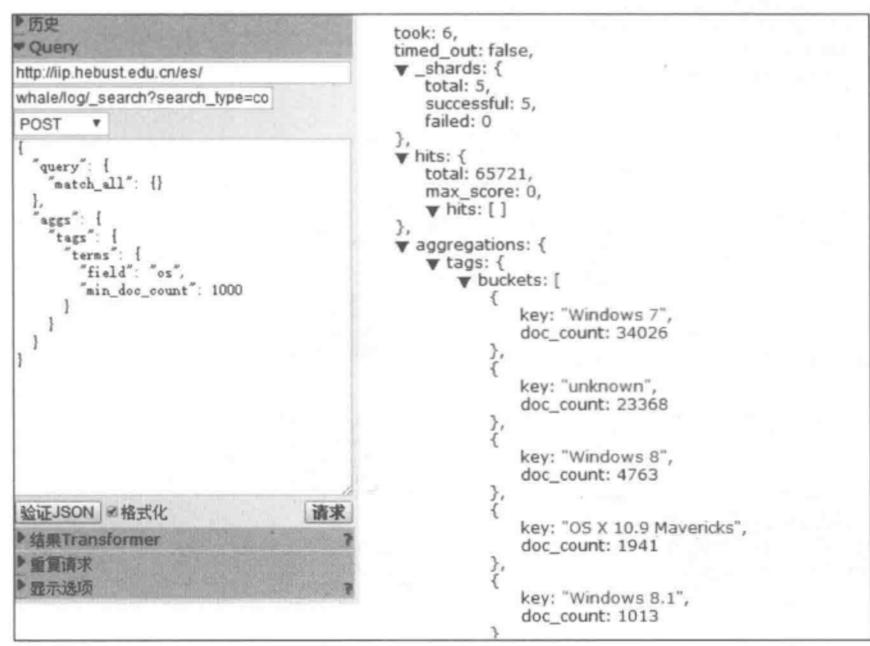


图 4.25 min_doc_count 参数的使用

代码段 4.28 演示了如何进行过滤和脚本的用法。这段代码的含义是先把所有返回的 terms(由_value 表示)的前面都加上指定的字符“OS: ”(由代码段中的“"script": "'OS: ' +_value"”来实现),然后再选择过滤(即,只要以“OS: Windows ”开头的项,而不要以“OS: OS X. ”开头的项)。针对类型文件 log 中的实际运行效果见图 4.26。从图中可以看出,所有的 key 字段的值的前面都冠以“OS: ”字符。

```
//代码段 4.28: 脚本的应用
{
  "query": {
    "match_all": {}
  },
  "aggs": {                               //aggregations
    "tags": {                             //名称
      "terms": {                          //执行 terms aggs
        "field": "os",                    //统计字段
        "script": "'OS: '+_value",        //脚本
        "min_doc_count": 1000,            //过滤条件
        "include": "OS: Windows.* ",      //包含
        "exclude": "OS: OS X.* "          //不含
      }
    }
  }
}
```

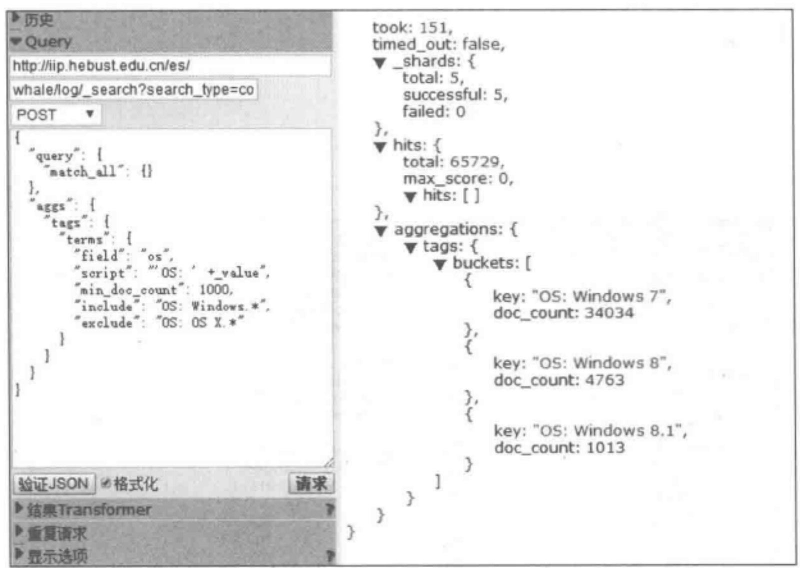

图 4.26　脚本与过滤机制的应用

## 4.3.5　Range Aggregations

和 range facets 的情形类似，range aggregations 用于范围统计。代码段 4.29 演示了 range aggregations 的使用方法，其目的是对类型文件 log 中的 size 字段进行统计，并按照 HTTP 返回的 size 信息，按小于 50、50～100、大于 100 分别进行范围统计。实际运行情况如图 4.27 所示。

```
//代码段 4.29: 范围统计
{
  "query": {
    "match_all": {}
  },
  "aggs": {
    "size_ranges": {                    //名称
      "range": {                        //range aggregations
        "field": "size",                //统计字段
        "ranges": [                     //范围
          {
            "to": 50                    //小于 50
          },
          {
            "from": 50,                 //50~1000
            "to": 100
          },
          {
            "from": 100                 //大于 100
          }
        ]
      }
    }
  }
}
```

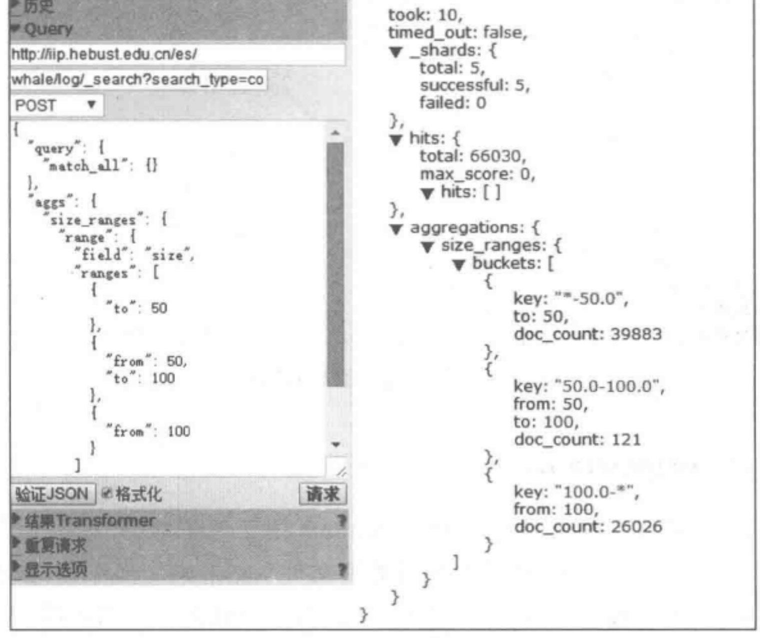


图 4.27 范围统计

可以使用 keyed 参数并将这个参数置为 true，这样可以将返回值中的 key 作为这个 JSON 对象的名称。也可以使用 key 参数来自定义 key 的名称，还可以在 range aggregations 中添加嵌套的子 aggregations，如代码段 4.30 所示，它演示了针对类型文件 log，分析字段 size 在三个不同区段内的统计值。同样，range aggregations 也支持 script 和 script 参数。

```
//代码段 4.30：桶嵌套，分别统计
{
  "query": {
    "match_all": {}
  },
  "aggs": {
    "size_ranges": {      //外层名称 size_ranges
      "range": {          //外层统计
        "field": "size",  //统计字段
        "keyed": true,
        "ranges": [  //内层统计，按照不同区段(small、medium、large)在数组中分别统计
          {
            "key": "small",
            "to": 50
          },
          {
            "key": "medium",
            "from": 50,
            "to": 100
          },
          {
            "key": "large",
            "from": 100
          }
        ]
      },
      "aggs": {           //内存统计
        "size_stats": {   //内层统计的名称：size_stats
          "stats": {      // stats aggregation 是一个多值统计，见 4.3.3 节中的叙述
            "field": "size"   //统计字段
          }
        }
      }
    }
  }
}
```

```
历史
Query
http://iip.hebust.edu.cn/es/
whale/log/_search?search_type=co
POST
          "key": "medium",
          "from": 50,
          "to": 100
        },
        {
          "key": "large",
          "from": 100
        }
      ]
    },
    "aggs": {
      "size_stats": {
        "stats": {
          "field": "size"
        }
      }
    }
  }
}
}
验证JSON 格式化 请求
结果Transformer
重复请求
显示选项
```


```
{
  took: 8,
  timed_out: false,
  _shards: {
    total: 5,
    successful: 5,
    failed: 0
  },
  hits: {
    total: 66073,
    max_score: 0,
    hits: [ ]
  },
  aggregations: {
    size_ranges: {
      buckets: {
        small: {
          to: 50,
          doc_count: 39883,
          size_stats: {
            count: 39883,
            min: 0,
            max: 49,
            avg: 0.2777875285209237,
            sum: 11079
          }
        },
        medium: {
          from: 50,
          to: 100,
          doc_count: 121,
          size_stats: {
            count: 121,
            min: 55,
            max: 99,
            avg: 69.64462809917356,
            sum: 8427
          }
        },
        large: {
          from: 100,
          doc_count: 26069,
          size_stats: {
```


图 4.28 嵌套的范围统计

### 4.3.6 Date_range Aggregations

上面提到的 range aggregations 是对于普通的数值类型字段的聚合统计，而 date_range aggregations 是专门对于时间类型的字段进行区段统计的。代码段 4.31 演示了其基本使用方法，统计在两个时间区段中指定字段的情况。返回值中的字段@timestamp(时间戳)值是一串用数字表示的时间，并不直观，可以对其进行格式化，使其变得可读。下面的例子演示了通过设置 format 参数将时间变得可读。

```
//代码段 4.31：针对日期型字段的区段统计
{
  "query": {
    "match_all": {}
  },
  "aggs": {
    "name of this aggs": {          //统计结果名称
      "date_range": {              //统计函数，为 date_range aggregations
        "field": "@ timestamp",    //用于统计的字段
        "format": "yyyy/MM/dd",    //转换其显示时间格式为：年/月/日
```

```
        "ranges": [             //定义统计范围
          {
            "to": "now-10d"     //统计范围 1: 从最远点到当前日期的前 10 天
          },
          {
            "from": "now-10d" //统计范围 2: 从当前日期的前 10 天到现在(即当前日期)
          }
        ]
      }
    }
  }
```

图 4.29 给出实际效果,这里以 to_as_string 显示时间戳对应的"年/月/日"格式,以 doc_count 显示统计结果。

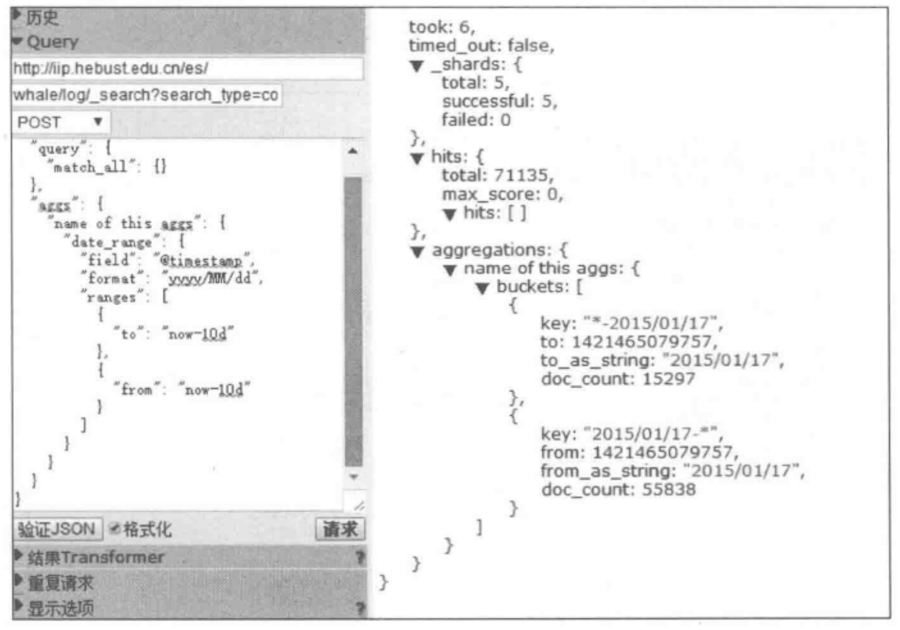


图 4.29 针对日期数据的范围统计

## 4.3.7 Histogram Aggregations

和 histogram facets 类似,histogram aggregations 是一种可以根据其返回值(针对数值型或日期型的字段)生成将来可用于柱状图的聚合数据。代码段 4.32 是计算 size 字段每间隔 5000 的统计分布情况,针对类型文件 log 的返回结果如图 4.30 所示。通过上面的聚合分析,可以得到 HTTP 请求的 size 在各个区段的分布数量。在这个基础上,可以利用一些可视化软件对上述数据生成一张图表。

```
//代码段 4.32：统计
{
  "query": {
    "match_all": {}
  },
  "aggs": {
    "name of this example": {       //此统计结果名称
      "histogram": {                //函数 histogram aggregations
        "field": "size",            //统计字段
        "interval": 50000,          //间隔
        "order": {                  //输出结果的排序策略
          "_key": "desc"  //以统计 Key 值(size 值)降序排序。可改为_count 按计数值排序
        }
      }
    }
  }
}
```

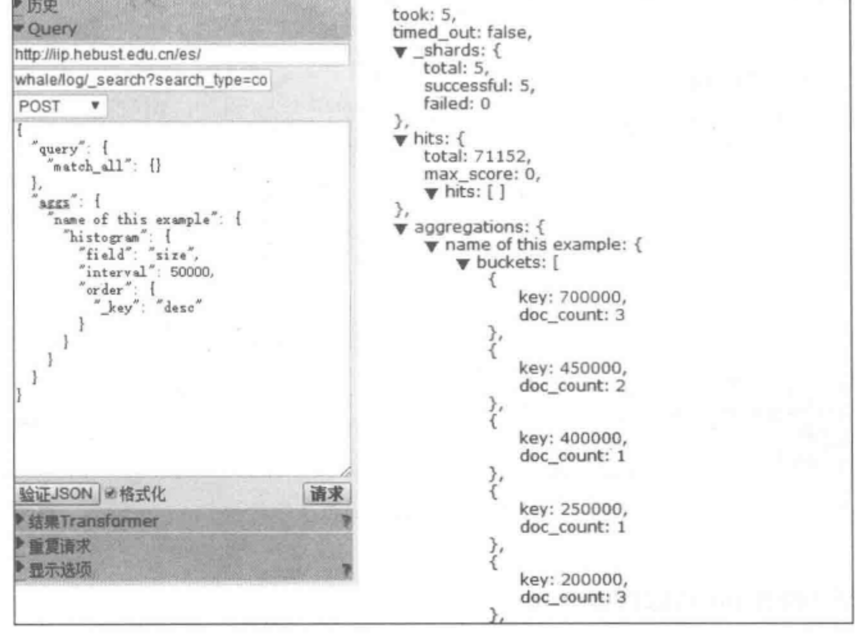

图 4.30 histogram aggregations

也可以在 histogram aggregations 中添加子 aggs 来实现在现有的 aggs 中再次嵌套进行统计。代码段 4.33 演示了对每一个 agg 中再次进行 stats aggregation 并按照子 aggs 中的最小值字段(即代码中的 size_stats.min)进行降序排序的实现方法，针对类型文件 log 的实际运行效果如图 4.31 所示。

```
//代码段 4.33: histogram aggregations 嵌套统计
{
  "query": {
    "match_all": {}
  },
  "aggs": {
    "first bucket layer": {          //外层 aggs 名称
      "histogram": {                 //统计函数
        "field": "size",             //统计字段
        "interval": 50000,           //间隔
        "order": {                   //排序准则
          "size_stats.min": "desc"
                      //按照子 aggs 测度中的最小值(即 size_stats.min)降序排序
        }
      },
      "aggs": {                      //子 aggs
        "size_stats": {              //子 aggs 名称
          "stats": {}    //stats aggregation 是一个多值统计,返回值包括最值、均值等
        }
      }
    }
  }
}
```

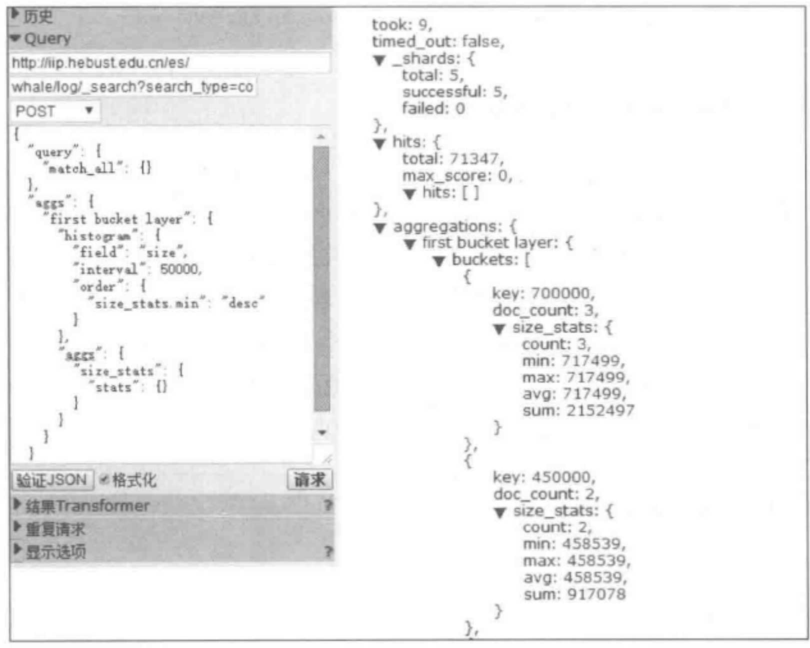

图 4.31 histogram aggregations 的嵌套

在上述例子中可以为中间结果取名，只需使用 keyed 参数并将其设置为 true，这样，前端程序可以通过这个名字来取得相应的统计结果并进行展示，如图 4.32 所示（代码实现参见图 4.32 的左侧）。请注意图 4.31 和图 4.32 的区别：图 4.31 的 buckets 结果是以数组方式给出的，而图 4.32 的 buckets 则是以键值对的形式给出的。

图 4.32 对中间结果取名的 histogram aggregations

### 4.3.8 Date_histogram Aggregations

date_histogram aggregations 是一个增强型的专门针对日期型字段统计的 histogram aggregation，它允许使用 year、month、week、day、hour、minute 等常量作为 interval 属性的取值。在代码段 4.34 中，在 field 中填写一个日期类型的字段名称，在 interval 中写一个步长值。返回值中的 key 是一串数字表示的时间，不直观。可以对其进行格式化，使其变得可读。下面的代码段 4.34 演示了通过设置 format 参数将时间变得可读的方法。相应地，图 4.33 中的 key_as_string 就是@timestamp 字段值转换为指定日期格式后的形式，其下的 doc_count 是返回满足条件的数据集大小（即在这个时间段有多少条 log 日志）。

```
//代码段 4.34: date_histogram aggregations
{
  "query": {
    "match_all": {}
  },
  "aggs": {
```

```
    "timestamp_aggs": {                     //统计名称
      "date_histogram": {                   //统计函数 date_histogram aggregations
        "field": "@ timestamp",             //统计字段
        "interval": "1d",                   //时间间隔,1d 为 1 天
"format": "yyyy-MM-dd"                      //显示格式
      }
    }
  }
}
```

```
▶历史
▼Query
http://iip.hebust.edu.cn/es/
whale/log/_search?search_type=co
POST
{
  "query": {
    "match_all": {}
  },
  "aggs": {
    "timestamp_aggs": {
      "date_histogram": {
        "field": "@timestamp",
        "interval": "1d",
        "format": "yyyy-MM-dd"
      }
    }
  }
}
```

```
{
  took: 9,
  timed_out: false,
  ▼ _shards: {
    total: 5,
    successful: 5,
    failed: 0
  },
  ▼ hits: {
    total: 71600,
    max_score: 0,
    ▼ hits: [ ]
  },
  ▼ aggregations: {
    ▼ timestamp_aggs: {
      ▼ buckets: [
        {
          key_as_string: "2015-01-12",
          key: 1421020800000,
          doc_count: 289
        },
        {
          key_as_string: "2015-01-13",
          key: 1421107200000,
          doc_count: 2675
```

图 4.33 基于 date_histogram 的统计

也可以在其中添加子 aggregations(即嵌套桶)来实现更丰富的统计。代码段 4.35 演示了以一天为步长单位,统计在每一天各个状态码的出现次数的实现方法。实际运行效果如图 4.34 所示。

```
//代码段 4.35: 基于嵌套桶机制实现的 date_histogram aggs
{
  "query": {
    "match_all": {}
  },
  "aggs": {                                 //外层 aggs
    "timestamp_aggs": {                     //外层 aggs 名称
      "date_histogram": {                   //统计函数 date_histogram aggregations
        "field": "@timestamp",              //统计字段
        "interval": "1d",                   //步长,1 天
        "format": "yyyy-MM-dd"              //针对指定字段的输出格式
      },
```

```
        "aggs": {                               //内层 aggs
          "statusCode": {                       //内层 aggs 名称
            "terms": {                          //统计函数 terms aggregations
              "field": "statusCode",            //统计字段
              "order": {                        //排序
                "_term": "asc"
              }
            }
          }
        }
      }
    }
  }
```

图 4.34 基于 date_histogram aggregations 的嵌套统计

### 4.3.9 Filter Aggregations

类似于 SQL 语句中 where 子句的作用,filter aggregation 可以为当前文档集合定义一个过滤条件来聚焦现有的数据集。凡满足定义的过滤条件(filter)的文档(document)都会被置入这个桶(bucket)中。代码段 4.36 展示了对类型文件 log 所有 size 字段大于 1000 的文档进行平均值统计的方法(均值统计是在嵌套的 aggregations 中实现的),相当于满足指定条件后再进行的统计。针对类型文件 log 的实际效果如图 4.35 所示。

```
//代码段 4.36: 过滤统计
{
  "query": {
    "match_all": {}
  },
  "aggs": {                               //外层 aggs
    "size_aggs": {                        //外层 aggs 的名称
      "filter": {                         //定义过滤条件
        "range": {                        //范围统计,执行 range aggregations
          "size": {                       //统计字段
            "gt": 1000                    //这里的"gt"是大于的意思
          }
        }
      },
      "aggs": {                           //子 aggs
        "avg_size": {                     //内层 aggs 的名称
          "avg": {                        //测度
            "field": "size"               //统计字段
          }
        }
      }
    }
  }
}
```

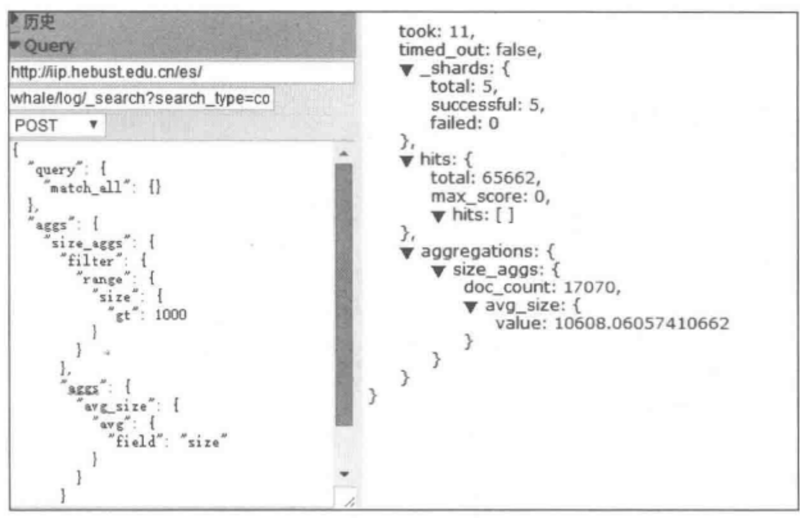


图 4.35 嵌套的 filter aggregations 统计

和 facets 类似,在 aggregations 中也是可以使用正则表达式的。代码段 4.37 给出了更复杂的 filter 使用(在 terms aggregations 中添加 min_doc_count 参数来控制最小返回的词项计数,相当于 Top-N 设置)。例子中添加了对于正则表达式的模式设定,可用的正则编译模式有 CANON _ EQ、CASE _ INSENSITIVE、COMMENTS、DOTALL、LITERAL、

MULTILINE、UNICODE_CASE、UNICODE_CHARACTER_CLASS 和 UNIX_LINES 等。多个模式之间可以使用"|"来进行分隔。针对类型文件 log 的运行效果参见图 4.36。

```
//代码段 4.37: filter aggregations 及正则表达式的应用
{
  "query": {
    "match_all": {}
  },
  "aggs": {
    "tags": {                          //aggs 的名称
      "terms": {                       //term aggregations
        "field": "os",                 //统计字段
        "script": "'OS: '+_value",     //脚本,作用是在 fields 字段值前添加指定的
                                       //字符串"OS: "
        "min_doc_count": 1000,         //控制最小返回词项计数
        "include": {                   //指定要包括的内容
          "pattern": "OS: Windows.*",
          "flags": "CANON_EQ|CASE_INSENSITIVE"
        },
        "exclude": {                   //指定排除项内容
          "pattern": "OS: OS X.*",
          "flags": "CANON_EQ|CASE_INSENSITIVE"
        }
      }
    }
  }
}
```

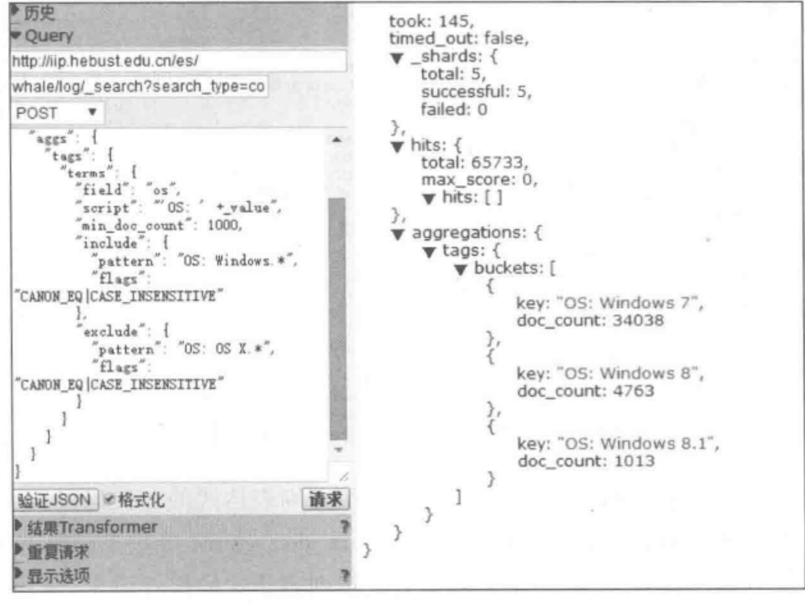


图 4.36 过滤与正则表达式的应用

### 4.3.10 Missing Aggregations

missing aggregation 用来统计缺少指定字段的文档个数，也称为缺值统计。代码段4.38统计类型文件 log 中缺少 ip 字段的记录。从图 4.37 返回的结果集可以看到，图中的 doc_count 返回了满足条件的数据集大小，也就是说，示例数据集中没有缺少 ip 字段的数据子集。

```
//代码段 4.38: missing aggregations
{
  "query": {
    "match_all": {}
  },
  "aggs": {                              //aggs
    "products_without_ip": {             //统计名称
      "missing": {                       //执行 missing aggregations
        "field": "ip"                    //待统计的字段
      }
    }
  }
}
```

```
▶历史
▼Query
http://iip.hebust.edu.cn/es/
whale/log/_search?search_type=count    POST
{
  "query": {
    "match_all": {}
  },
  "aggs": {
    "products_without_ip": {
      "missing": {
        "field": "ip"
      }
    }
  }
}
```

```
{
  took: 4,
  timed_out: false,
  ▼ _shards: {
      total: 5,
      successful: 5,
      failed: 0
  },
  ▼ hits: {
      total: 121218,
      max_score: 0,
      ▼ hits: [ ]
  },
  ▼ aggregations: {
      ▼ products_without_ip: {
          doc_count: 0
      }
  }
}
```

图 4.37 缺值统计

## 4.4 搜索提示

在很多大型搜索网站（如电商和搜索引擎）中都提供搜索提示功能，这将提升用户的搜索体验。Suggester plugin 是 Elasticsearch 的一个插件，可用来提供搜索提示。如果使用的是 Elasticsearch V1.0，可以在 Elasticsearch 的目录下执行如下命令就可以安装 Suggester plugin 插件：

```
bin/plugin -install de.spinscale/elasticsearch-plugin-suggest/1.0.1-2.0.0
```

下面以索引文件 page 下类型文件 pages 为例，说明 suggester 的用法。

首先构造一个 HTTP 请求,需要注意的是,suggest 前面是两个下划线,通过 field 参数来设置搜索候选词的来源字段,term 就是在搜索框内输入的字符,size 参数设置了需要返回的词项集合的大小。针对此例的返回结果如图 4.38。在返回结果中有一个 suggestions 列表,里面显示的就是跟输入词项有关的推荐词。之后,可以在前端页面上监听输入框的输入变化,并适时将这些搜索建议词通过 Ajax 的形式提交至前端页面。

```
curl -X POST 'localhost:9200/page/pages/__suggest?pretty=1' -d '{ "field":
"content", "term": "中", "size": "10"  }'
```

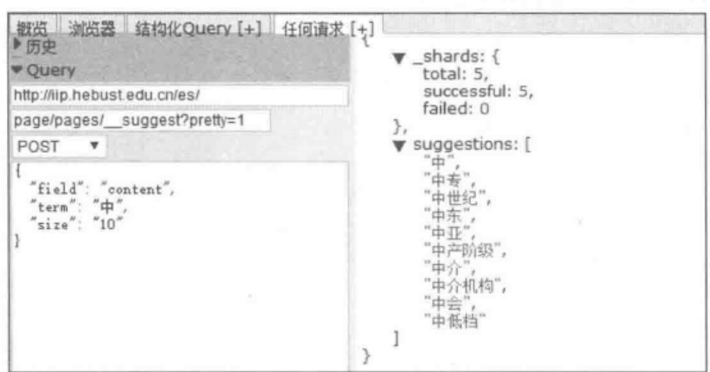


图 4.38 推荐结果示例

## 4.5 扩展知识与阅读

本章对信息统计进行了阐述,其中多处提到正则表达式的用法。文献[余晟,2012]总结出一套使用正则表达式解决问题的办法,并通过具体的例子说明其实际应用,文中提到的各种统计是可以应用在实际的信息检索系统中的。文献[罗刚,2014] 总结搜索引擎相关理论与实际解决方案,包括搜索提示等的内容,并给出了 Java 实现。在完成搜索提示时,很多时候可能会用到 Ajax 技术。文献[李刚,2014] 介绍了 jQuery 1.8、Ext JS 4.1、Prototype 1.7.1、DWR 这几个常用 Ajax 框架的用法,针对每个框架提供了实用方法。

## 4.6 本章小结

本章是对基于 RESTful 的 Elasticsearch 信息检索方法的进一步深化,是在搜索结果的基础上完成的统计和分析。较早的 Elasticsearch 版本是用 facets 完成这种统计分析工作的。随着 Elasticsearch 功能的逐步提高和更高版本的推出,facets 即将退出历史舞台,转而由 aggregations 来完成这种复杂的统计和分析工作。基于 buckets 机制的 aggregations 统计分析功能更趋完善。在进行统计分析时,不仅可以使用各种脚本命令,也能将正则表达式等嵌入到表达式中。基于 facets 和 aggregations 的统计结果又可以由可视化工具加工处理成可视化的结果提供给用户,可进一步提升用户的搜索体验。

Chapter 5

# Elasticsearch 部分功能的 Java 客户端实现

> “Elasticsearch is built using Java, and requires at least Java 7 in order to run. Only Oracle′s Java and the OpenJDK are supported. The same JVM version should be used on all Elasticsearch nodes and clients. We recommend installing the Java 8 update 20 or later, or Java 7 update 55 or later. Previous versions of Java 7 are known to have bugs that can cause index corruption and data loss. The version of Java to use can be configured by setting the JAVA_HOME environment variable.”——http://www.elasticsearch.org/guide/en/elasticsearch/reference/current/setup.html

前述章节中已经对 Elasticsearch 的索引、检索、统计等功能进行了说明。一般来说，对 Elasticsearch 中的信息进行检索等处理的大致步骤是：基于 Analyzer 分析结果构建 Query、给后台的 Elasticsearch 集群发送 Query、完成检索并返回结果集合。其实，Elasticsearch 不仅可以通过 RESTful 等方式进行操作，通过各种语言的客户端也可以做几乎所有的操作。本章以应用较为广泛的 Java 客户端为例，介绍 Elasticsearch 相关功能的 Java 实现。在完成客户端编程时，一般首先要做的工作是将 Elasticsearch 节点实例化，之后构建索引文件，基于用户给定的查询项获取文档集合（当然也可以完成对文档的增、删、改等工作），并根据实际需求完成基于 facets 或 aggregations 的数据分析工作，在完成进一步处理（如分页、高亮、过滤等）后，将处理结果返回到前端。

## 5.1 Elasticsearch 节点实例化

在进行 Elasticsearch 的客户端编程时，首先要对 Elasticsearch 的节点进行实例化。在这之前，需要在 Java 工程中添加相关的 Elasticsearch 依赖（可以通过 Maven 添加）。

### 5.1.1 通过 Maven 添加对 Elasticsearch 依赖

Maven 是一个项目管理工具，是基于项目对象模型的、通过描述信息来管理项目的构

建和文档的软件项目管理工具。Maven 包含一个项目对象模型、一组标准集合、一个项目生命周期、一个依赖管理系统等，以及用来运行定义在生命周期阶段中插件目标的逻辑。Maven 可以应用一些来自一组共享的(或者自定义的)逻辑插件。使用 Maven 时，可以用一个明确定义的项目对象模型来描述项目。Maven 的定义包括发布项目信息的方式以及一种在多个项目中共享 JAR 的方式[Maven,2014a]。

> Tips：和单纯的软件构建工具不同，Maven 是软件项目管理和理解工具。Maven 除了具备 Ant(Ant 是一个将软件编译、测试、部署等步骤联系在一起的一个工具)的功能外，还使用项目对象模型来对软件项目进行管理。它内置了更多的隐式规则，使得构建文件更加简单，内置依赖管理和 Repository 来实现对依赖的管理和统一存储，内置软件构建的生命周期。

POM 是 Maven 对一个单一项目的描述，它实现了一种以模型来进行描述的构建方式。图 5.1 显示的是在 IDEA 开发环境中基于 Maven 构建的工程中 pom.xml 的位置。可以在 Java 工程中的 pom.xml 文件中添加和 Elasticsearch 相关的语句，以便将 Elasticsearch 的相关 JAR 包文件引入到相应的工程中来，注意在 pom.xml 中要把 Elasticsearch 的{es.version}替换为当前的 Elasticsearch 版本号。代码段 5.1 是一个 pom.xml 文件中的内容。

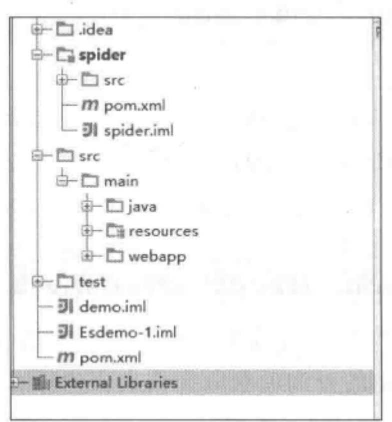


图 5.1 pom.xml

```
//代码段 5.1: pom.xml
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.
w3.org/2001/XMLSchema-instance"
   xsi: schemaLocation =" http://maven. apache. org/POM/4. 0. 0 http://maven.
apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.gs</groupId>
  <artifactId>demo</artifactId>
```

```
    <packaging>pom</packaging>
    <version>1.0-SNAPSHOT</version>
      <modules>
          <module>spider</module>
      </modules>
      <name>demo Maven Webapp</name>
    <url>http://maven.apache.org</url>
      <dependencies>
      <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>3.8.1</version>
        <scope>test</scope>
      </dependency>
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-api</artifactId>
            <version>1.7.6</version>
        </dependency>
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-log4j12</artifactId>
            <version>1.7.6</version>
        </dependency>
              <dependency>
            <groupId>org.elasticsearch</groupId>
            <artifactId>elasticsearch</artifactId>
            <version>1.4.0</version>         //此处的版本号必须与实际情况一致,否则
                                             //将会出现 No node available 的错误
        </dependency>
          </dependencies>
    <build>
      <finalName>demo</finalName>
    </build>
  </project>
```

## 5.1.2 初始化 Elasticsearch Client

接下来需要初始化 Elasticsearch Client。一般来说,初始化方法有如下两种方式。

方式 1：基于 Node Client 的方式,即 Node Client 本身作为 Elasticsearch 集群的一个节点加入该集群,但这种方法升级维护比较麻烦。代码段 5.2 演示了如何基于 Node Client 方式初始化 Elasticsearch Client。

```
//代码段 5.2: 基于 Node Client 方式初始化
//import 语句,略
public void test2() {                          //测试函数
    Node node=nodeBuilder().node();            //启动 Node Client
    Client client=node.client();
    node.close();                              //关闭 Node Cient
    System.out.println("NodeBuilder 方式");
}
```

方式2：基于 Transport Client 的方式。这是一种更轻量级的方法，它通过 socket 与 Elasticsearch 集群相连，是基于 netty 线程池的方式。在 InetSocketTransportAddress 的构造方法内需填写一个已经启动的 Elasticsearch 节点的 Host 及其端口号(默认端口号是9300)。可通过链式调用 addTransportAddress 方法的方式添加很多类似节点。代码段 5.3 给出了实现方法。

```
//代码段 5.3: 基于 Transport Client 的方式初始化 Client
//import 语句,略
//启动一个 Client
Client client=new TransportClient()
    .addTransportAddress(new InetSocketTransportAddress("hostA", 9300))
    .addTransportAddress(new InetSocketTransportAddress("hostB", 9300));
client.close();//关闭 Client
```

如果有很多集群，还可通过使用指定集群名称的方法来构建 Transport Client，代码段 5.4 给出了实现方法。

```
//代码段 5.4: 通过指定集群名称的方法来构建 Transport Client
//import 语句,略
Settings settings=ImmutableSettings
    .settingsBuilder()
    .put("cluster.name", "myClusterName")
    .build();
Client client=    new TransportClient(settings);
//添加 Transport 地址,然后可以用 Client 做其他事情
```

还可通过 client. transport. sniff 方法开启嗅探模式来自动加入集群，如代码段 5.5 所示。

```
//代码段 5.5: 开启嗅探模式
//import 语句,略
Settings settings=ImmutableSettings.settingsBuilder()
    .put("client.transport.sniff", true)
    .build();
```

```
//启动一个 Client
Client client=new TransportClient()
    .addTransportAddress(new InetSocketTransportAddress("hostA", 9300))
    .addTransportAddress(new InetSocketTransportAddress("hostB", 9300));
client.close();          //关闭 Client
```

：client.close()：这个方法用于关闭客户端。

## 5.2　索引数据

本节介绍创建索引的方法。先从准备 JSON 数据开始。

### 5.2.1　准备 JSON 数据

JSON 数据的产生有多种方法。

方法 1：产生符合 JSON 规范的字符串，并将其存于 String 或 byte[]类型的变量中。代码段 5.6 就是一个通过手工方法构建 JSON 字符串的示例。

```
//代码段 5.6：通过手工方法构建 json 字符串
String json="{"+  "\"name\":\"Tom\","+  "\"time\":\"2015-01-30\","+  "\"
content\":\"happy\""+  "}";
```

方法 2：使用第三方的 JSON 工具库(如 gson、jackson 等)来序列化 java bean。

：gson 是 Google 公司发布的一个开放源代码的 Java 库，主要用途为序列化 Java 对象为 JSON 字符串，或反序列化 JSON 字符串成 Java 对象。

在 gson 实现过程中，首先在 pom.xml 中添加对 gson 的依赖，见代码段 5.7。

```
//代码段 5.7：在 POM 中添加对 gson 的依赖
<dependency>
    <groupId>com.google.code.gson</groupId>
    <artifactId>gson</artifactId>
    <version>2.3</version>
</dependency>
```

之后，对需要处理的数据完成 java bean 序列化，见代码段 5.8 的 new Gson().toJson()语句。

```
//代码段 5.8：序列化
//import 语句，略
Map<String, Object>map=Maps.newHashMap();   //构建 map 对象
map.put("name","hebust");
map.put("age",23);
map.put("content","hello world");
map.put("haa",new String[]{"big data","mining","information retrieval"});
String s=new Gson().toJson(map);             //将 map 中的数据完成序列化
System.out.println(s);
```

方法 3：使用 Elasticsearch 自带的工具 XContentFactory. jsonBuilder()。代码段 5.9 是使用这个 JSON 工具类进行的操作。实际运行效果如图 5.2 的下方所示。所有格式的数据都会被转换成 byte[]格式的。因此，如果待处理的数据就是这样的格式，就可以直接使用它而不用再转换。

```
//代码段 5.9：基于 XContentFactory.jsonBuilder()的方法
//import 语句，略
XContentBuilder builder=jsonBuilder().startObject()
                                          //基于 jsonBuilder()构建数据
    .field("name", "Tom")
    .field("time", new Date())
    .field("content", "happy")
    .endObject();
System.out.println(builder.string());         //显示生成的 JSON 字符串
```

```
            .field("time", new Date())
            .field("content", "happy")
            .endObject();
        System.out.println(builder.string());
    }
Done: 1 of 1 (in 3s)
"C:\Program ...
20:30:58[INFO org.elasticsearch.plugins] [Forgotten One] loaded [], sites []
{"name":"Tom","time":"2015-02-15T12:31:00.772Z","content":"happy"}
```

图 5.2 JSON 数据准备

有关 jsonBuilder()的使用，在 5.3.3 节更新索引文档部分亦有体现。

## 5.2.2 索引 JSON 数据

当对生成的 JSON 格式的文档进行索引时，可采用在 IndexResponse 里包含 Elasticsearch 索引信息的方法。

Tips：常用的准备索引 JSON 数据的 API 有：

- prepareIndex()。
- prepareIndex(String index, String type)。
- prepareIndex(String index, String type, String id)。

代码段 5.10 给出索引 JSON 数据的实现方法，其中变量 s 是在代码段 5.8 中提到的通过 new Gson().toJson()语句生成的 JSON 数据，myweibo1 是在 Elasticsearch 中建立好的索引文件，my1 是其中的类型文件(注：有关代码的完整部分参见图 5.3 的上半部分)。

```
//代码段 5.10：索引 json 数据
//import 语句，略
Client client=new TransportClient()
    .addTransportAddress(new InetSocketTransportAddress("localhost",
9300));    //初始化 Elasticsearch Client
IndexResponse response=client.prepareIndex("myweibo1", "my1")
    .setSource(s)
    .execute()
    .actionGet();
//下面的语句是在控制台输出得到的索引信息
String _index=response.getIndex();        //得到 index 名称
String _type=response.getType();          //得到 Type 名称
String _id=response.getId();              //得到 Document ID
long _version=response.getVersion();      //版本号。如是首次索引该文档，则此值为 1
System.out.print(_index+"\t"+_type+"\t"+_id+"\t"+_version+"\n");
```

针对代码段 5.10 的运行效果如图 5.3 下半部分所示；相应地，Elasticsearch 产生的索引数据如图 5.4 所示。

```
@Test
public void test2() {
    Map<String, Object> map = Maps.newHashMap();
    map.put("name", "hebust");
    map.put("age", 23);
    map.put("content", "hello world");
    map.put("haa", new String[]{"big data", "mining", "information retrieval"});
    String s = new Gson().toJson(map);
    System.out.println(s);
    IndexResponse response = client.prepareIndex("myweibo1", "my1").setSource(s).execute().actionGet();

    String _index = response.getIndex(); //得到index 名称
    String _type = response.getType(); //得到Type 名称
    String _id = response.getId();//得到Document ID
    long _version = response.getVersion();// 版本号，如是首次索引该文档，则此值为1

    System.out.print(_index + "\t" + _type + "\t" + _id + "\t" + _version + "\n");

Done: 1 of 1 (in 1s)
"C:\Program ...
11:49:37[INFO org.elasticsearch.plugins] [Freak] loaded [], sites []
{"content":"hello world","age":23,"name":"hebust","haa":["big data","mining","information retrieval"]}
myweibo1    my1 AUvoDZPKXW72dUDWQAo6    1
```

图 5.3 程序运行效果

```
{
    "_index": "myweibo1",
    "_type": "my1",
    "_id": "AUvoDZPKXW72dUDWQAo6",
    "_version": ,
    "_score": ,
      "_source": {
       "content": "hello world",
       "age": ,
       "name": "hebust",
         "haa": [
          "big data"
          ,
          "mining"
          ,
          "information retrieval"
        ]
    }
}
```

图 5.4　产生的索引数据

# 5.3　对索引文档的操作

## 5.3.1　获取索引文档

本书前述章节已经介绍了如何在 Elasticsearch 中获取待处理的文档信息。其实，在 Get API 的帮助下，也可以在 Java 客户端获取文档信息。代码段 5.11 给出了方法，请注意其中的 prepareGet(String index, String type, String id)方法，其中的三个参数分别是 Elasticsearch 中的某个索引名、其中的某个类型名、文档的 ID 号(为便于说明，代码中直接给出了某个文档的 ID 号)。

```
//代码段 5.11：获取文档信息
//import 语句，略
client=new TransportClient().addTransportAddress(new InetSocketTransport-
Address("localhost", 9300));
GetResponse response=client.prepareGet("test", "news", "C3D5nQgLTI-jvx8-
CAE6A")
      .execute()
      .actionGet();
System.out.println(response.getSource());                    //显示针对该条的数据细节
```

图 5.5 给出了实际运行效果，其中图左侧是 Elasticsearch 中某个文档的内容，图右侧下部是程序运行结果。

```
Result Source
{
  _index: "test",
  _type: "news",
  _id: "C3D5nQgLTI--jvX8-CAE6A",
  _version: ,
  _score: ,
    _source: {
     content: "中国总理李克强访问德国,并会见了德国总理默克尔",
     title: "中国总理出访",
     author: "高凯",
     publish_date: "2014/10/13",
     category: "国际新闻"
  }
}
```

```
@Test
public void test3() {//测试 GetResponse
    client = new TransportClient().addTransportAddress(new InetSocketTrans
    GetResponse response = client.prepareGet("test", "news", "C3D5nQgLTI--
    System.out.println(response.getSource());
}

1 of 1 (in 1s)
"C:\Program ...
07:16:49[INFO org.elasticsearch.plugins] [Cordelia Frost] loaded [], sites []
07:16:50[INFO org.elasticsearch.plugins] [Devastator] loaded [], sites []
{content=中国总理李克强访问德国,并会见了德国总理默克尔, author=高凯, category=国际新闻, title=中国总理出访
```

图 5.5　获取索引文档数据

：常用的获取索引文档数据的 API 有：

- get(GetRequest request)、get(GetRequest request, ActionListener<GetResponse> listener)——根据 GetRequest 提供的 index、type 和 id 获取文档。
- prepareGet()、prepareGet(String index, String type, String id)——准备执行获取。
- multiGet (MultiGetRequest request)、multiGet (MultiGetRequest request, ActionListener<MultiGetResponse>listener)——批量获取文档。
- prepareMultiGet()——准备批量获取。

### 5.3.2 删除索引文档

和获取文档信息类似，也可以在 Java 客户端删除索引信息，可以通过 prepareDelete() 方法实现。实现方法如代码段 5.12 所示（注：prepareDelete 方法的第三个参数是拟删除文档的 ID 号，其前面两个参数则分别是索引文件名称、类型文件名称）。

```
//代码段 5.12: 删除文档信息
//import 语句,略
client=new TransportClient().addTransportAddress(new InetSocketTransport-
Address("localhost", 9300));
DeleteResponse response = client. prepareDelete ( " myweibo1 ",  " my1 ",  " JN -
cWm6OSnKxH-dqbgqvgw")
     .execute()
     .actionGet();
boolean isFound=response.isFound();
System.out.println(isFound);          //返回索引是否存在
Map headers=response.getHeaders();
System.out.println(headers);          //返回响应头
System.out.println("指定 ID 号的记录已经被删除");
```

：常用的删除索引文档数据的 API 有：

- prepareDelete()、prepareDelete(String index, String type, String id)——准备删除。
- delete(DeleteRequest request)、delete(DeleteRequest request, ActionListener <DeleteResponse> listener)——根据 DeleteRequest 提供的 index、type 和 id 从索引中删除索引数据。
- deleteByQuery (DeleteByQueryRequest request)、deleteByQuery (DeleteByQuery-Request request, ActionListener<DeleteByQueryResponse> listener)——基于查询删除索引数据。

在客户端，可以基于 Elasticsearch 提供的 DeleteByQuery APIs 完成指定查询条件（由 setQuery()方法来实现）的删除操作。代码段 5.13 给出基于 prepareDeleteByQuery()的删除索引数据的方法。在这里要首先基于 setQuery()完成对拟删除信息的查找，找到后再删除。

```
//代码段 5.13：基于 prepareDeleteByQuery()删除数据的操作
//import 语句，略
DeleteByQueryResponse response=client.prepareDeleteByQuery("test")
                                                    //这里的 test 是索引文件名
    .setQuery(termQuery("author","大数据"))
    .execute()
    .actionGet();
System.out.println(response.status());
```

### 5.3.3 更新索引文档

和获取文档信息的方法类似类似，也可以在 Java 客户端更新索引文档。代码段 5.14 给出了方法，通过 client 对象的 update()方法实现。注意这里对指定 id 号的多个字段信息进行更新的方法（代码段中对 author 和 publish_date 两个字段进行了数据更新操作）。

```
//代码段 5.14：更新文档，注意 jsonBuilder()的用法
//import 语句，略
UpdateRequest updateRequest=new UpdateRequest();
updateRequest.index("test");              //索引名
updateRequest.type("news");               //type 名
updateRequest.id("C3D5nQgLTI--jvX8-CAE6A");
      //到此为止，id 号已定位到待更新的记录上。下面是对该记录中的部分字段值进行更新
updateRequest.doc(jsonBuilder().startObject()
.field("author", "本报记者")                //修改 author 字段中的内容
.field("publish_date", "2015/3/5")         //修改 publish_date 字段中的内容
.endObject());
client.update(updateRequest).get();        //通过 client 对象的 update()方法实现
```

Tips：常用的更新索引文档数据的 API 有：

- update(UpdateRequest request)——基于 script 更新文档并返回更新后的结果。
- update ( UpdateRequest request, ActionListener < UpdateResponse > listener)——更新但不返回结果，而是交由监听器进行处理。
- prepareUpdate()、prepareUpdate(String index, String type, String id)——准备更新。

### 5.3.4 批量操作索引文件

可通过 Bulk APIs 来对指定的文档进行批量操作，示例方法如代码段 5.15 所示，这里通过 Bulk 操作同时进行了两个增加数据到索引的操作（prepareIndex）和一个更新操作（prepareUpdate）。

```
//代码段 5.15：对文档的批量操作
//import 语句,略
BulkRequestBuilder bulkRequest=client.prepareBulk();
bulkRequest.add(client.prepareIndex("test", "news", "111")    //第一段操作
    .setSource(jsonBuilder()
    .startObject()
        .field("publish_date", "2015/6/1")
        .field("title","国际儿童节")
        .field("content", "儿童的节日")
        .endObject()));
bulkRequest.add(client.prepareIndex("test", "news", "222")    //第二段操作
    .setSource(jsonBuilder()
    .startObject()
        .field("publish_date", "2015/5/1")
        .field("title", "国际劳动节")
        .field("content", "今天是劳动节")
        .endObject()));
bulkRequest.add(client.prepareUpdate ("test", "news", "C3D5nQgLTI--jvX8-
CAE6A")                                                        //第三段操作
    .setDoc(jsonBuilder()
        .startObject()
        .field("author", "大数据")
        .endObject()));
BulkResponse bulkResponse=bulkRequest.execute().actionGet();
if (bulkResponse.hasFailures()) {
    //可在这里对于失败请求进行处理,略
    }
```

Tips：常用的批量操作索引文档的 API 有：

- bulk(BulkRequest request)、bulk(BulkRequest request, ActionListener<BulkResponse>listener)——批量操作。
- prepareBulk()——准备执行批量操作。

### 5.3.5 简单的统计操作

在客户端可以基于 Elasticsearch 提供的 API 完成多种统计操作。这里给出计数统计

count 的操作方法，见代码段 5.16 中的 response.getCount()。通过使用 count API，可以返回被某查询（由 setQuery()方法来实现）命中的索引文档的数量。

```
//代码段 5.16: 统计操作
//import 语句，略
//下面语句中的 test 是 index 名称，author 是其中的一个字段名称
CountResponse response=client.prepareCount("test")
    .setQuery(QueryBuilders.termQuery("author","大数据"))
    .execute()
    .actionGet();
System.out.println(response.getCount());    //返回被查询命中的索引文档的数量
```

Tips：常用的计数统计 API 有：

- count(CountRequest request)、count(CountRequest request, ActionListener <CountResponse> listener)——获取适应特定 query 的文档数量。
- prepareCount(String... indices)——准备获取适应特定 query 的文档数量。

## 5.4 信息检索

信息检索是 Elasticsearch 的主要功能。同样，在 Java 客户端，Elasticsearch 也提供多种方式供前端实现信息检索功能。

### 5.4.1 概述

Elasticsearch 中提供了强大的全文检索功能。借助相应的 API，可以构造一个 Elasticsearch 检索请求。一般地，首先要在项目中引入相应的类文件，如：

```
import org.elasticsearch.action.search.SearchResponse;
import org.elasticsearch.action.search.SearchType;
import org.elasticsearch.index.query.FilterBuilders.*;
import org.elasticsearch.index.query.QueryBuilders.*;
```

之后，在 Elasticsearch client 实例化对象的 prepareSearch()方法中，设置索引文件和类型文件的名称。在 setTypes()方法里可设置需要搜索的类型文件，在 setQuery()方法里可设置查询，而这个 query 可使用 Elasticsearch 自带的 QueryBuilders()方法来构建查询。当然，也可在 setPostFilter()方法中设置在搜索前需要执行的过滤操作（可使用 FilterBuilders()方法构建过滤器），如代码段 5.17 所示。其中，client.prepareSearch 用来创建一个 SearchRequestBuilder(client.prepareSearch 方法的参数是一个或多个索引文件名称)，这里的 setSearchType()中的参数是一个枚举类型的类，其值如表 5.1 所示[CSDN, 2015]。

```
//代码段 5.17：检索操作
//import 语句，略
SearchResponse response=client.prepareSearch("index1", "index2").setTypes
("pageA", "pageB")
    .setSearchType(SearchType.DFS_QUERY_THEN_FETCH)
    .setQuery(QueryBuilders.termQuery("multi", "test"))
    .setPostFilter(FilterBuilders.rangeFilter("size").from(1200).to(1800))
    .setFrom(0)
    .setSize(90)
    .setExplain(true)
    .execute()
    .actionGet();
```

表 5.1 setSearchType()取值及其含义

| 枚举元素 | 含义 |
|---|---|
| QUERY_THEN_FETCH | 先向所有的 shards 发出请求，各分片只返回排序和排名相关的信息(不包括 document)，然后按照各 shards 返回的分数进行排序并取前 size 个文档，之后再去相关的 shard 取 document。这对于有许多 shards 的索引来说是很便利的，返回结果不会有重复的 |
| QUERY_AND_FETCH | 最快的处理方式，它向索引的所有分片 shards 都发出查询请求，每个 shard 分别返回一定数量的结果 |
| DFS_QUERY_THEN_FETCH | 与 QUERY_THEN_FETCH 相似 |
| DFS_QUERY_AND_FETCH | 与 QUERY_AND_FETCH 相似 |
| SCAN | 当进行了没有进行任何排序的检索时，执行浏览操作 |
| COUNT | 计算结果的数量 |

Tips 常用的检索 API 有：

- search(SearchRequest request)、search(SearchRequest request, ActionListener <SearchResponse> listener)——根据 SearchRequest 提供的 index、id 和 type 执行搜索。
- prepareSearch(String... indices)——准备执行搜索。

## 5.4.2 MultiSearch

MultiSearch 是 Elasticsearch 提供的针对多个查询请求进行一次查询的接口。该接口虽能同时执行多个不同的查询，但无法对最终结果自动分页，而且有可能多个 SearchRequest 查询出来的结果中存在重复的内容，但 MultiSearch 并不负责去重。下面的示例代码段 5.18 展示了如何使用 Java 客户端进行 MultiSearch 操作。在代码中，首先在

prepareSearch()中构造两个查询(设定查询项及返回结果集大小),之后分别将它们添加到prepareMultiSearch()中并执行多重查询。余下代码段中的外层 for 循环的作用是得到结果集(可以从 MultiSearchResponse. getResponses()方法中来获取单个的 response),内层 for 循环则分别显示它们。

```
//代码段 5.18: MultiSearch
//import 语句,略
SearchRequestBuilder srb1=client.prepareSearch()
    .setQuery(QueryBuilders.queryString("content:中国")) //设定查询项
    .setSize(3);                                          //设定返回结果集大小
SearchRequestBuilder srb2=client.prepareSearch()
    .setQuery(QueryBuilders.termQuery("content", "美国"))//设定查询项
    .setSize(4);                                          //设定返回结果集大小
MultiSearchResponse sr=client.prepareMultiSearch()
    .add(srb1)                                            //添加第一个结果
    .add(srb2)                                            //添加第二个结果
    .execute()
    .actionGet();
for (MultiSearchResponse.Item item: sr.getResponses()) {  //外层循环得到结果集
    SearchResponse response=item.getResponse();
    for (SearchHit hit: response.getHits().getHits()) {   //内层循环完成显示
        System.out.println(hit.getSourceAsString());
    }
  System.out.println("==========================="); //分隔符,非必须
}
```

Tips: 常用的执行多个搜索请求的 multiSearch API 有:

- multiSearch(MultiSearchRequest request)——根据 MultiSearchRequest 提供的信息执行多重搜索。
- multiSearch(MultiSearchRequest request, ActionListener<MultiSearchResponse> listener)。
- prepareMultiSearch()——准备执行多重搜索。

### 5.4.3 Query DSL 概述

本章前面提到的 Search API 允许执行一个信息检索,返回一个与查询匹配的结果集(Hits)。它可以在一个(或多个)索引文件及其类型文件上执行,此时可能会用到 Query API 或 Filter API。Query DSL 用来确定哪些文档匹配了指定的 Filter,同样对文档进行排序,并通过它们的相似度来做排序的依据。Query DSL 包括 Query 和 Filter。常见的 Query

有 matchQuery、matchAllQuery、multiMatchQuery、boolQuery、termQuery、wildcardQuery、queryString、moreLikeThis 等多种形式（鉴于其功能上的紧密关联性，这里将 queryString、moreLikeThis 也归在此类中），而 Query 还能关联特定的 Filter 表达式。Query 和 Filter 可以被用于各种不同的 API 接口。基于 Filter 的操作比单纯的 Query 更快，因为它不进行文档评分并且会自动缓存结果，而且其他的 Query 可以重用这些 Filter，因此在很多时候使用 Filter 的效率较高。多个 Filter 间还可以基于 and、not 和 or 的逻辑关系进行组合。过滤器允许设置 _cache 元素来显式控制缓存与否，且允许设置一个_cache_key 用来当作缓存的主键，这个在过滤大集合的情况下有用。Elasticsearch 有多种使用 Filter 的方式，可以控制将 Filter 应用在 Query 和 Facet 上。本节介绍的 Filter 方法是应用在 Query 上的。

在 Java 客户端使用的 Query API 或 Filter API 的方式和 RESTful 是相似的。例如，编程者可以用 QueryBuilders 来构建 Query，然后使用 Search API 来进行搜索。当然也可以使用 Filter。本节首先介绍各种基于 Query 的客户端实现方法，之后介绍常用 Filter 的实现方法。

### 5.4.4　MatchQuery

matchQuery(String name, Object text)能够使用某一字段的值对文档进行检索。代码段 5.19 给出示例代码，其功能是在索引文件名和类型文件名均为 baike 的文件中，在 content 字段搜索包含有“中国”字符的数据集。其他的相关方法如 setSearchType()、setFrom()、setSize()、setExplain()等的解释不再赘述。可以把 matchQuery()放到 QueryBuilders 方法中并将结果作为 setQuery 的参数，代码段 5.19 是相关代码。

```
//代码段 5.19: matchQuery()
//import 语句,略
SearchResponse response=client.prepareSearch("baike")
  .setTypes("baike")
  .setSearchType(SearchType.DFS_QUERY_THEN_FETCH)  //参数含义如表 5.1 所示
  .setQuery(QueryBuilders.matchQuery("content","中国")) //在哪些字段上搜索什么内容
  .setFrom(0)
  .setSize(10)
  .setExplain(true)
  .execute()
  .actionGet();
for (SearchHit hit: response.getHits().getHits()) { //遍历检索结果
  System.out.println(hit.getSourceAsString());
}
```

### 5.4.5　MatchAllQuery

matchAllQuery 用于匹配文档中的所有字段。当需要匹配所有项，或者查询某些索引、某些类型文件下的记录总量时，可以使用 matchAllQuery()方法。它相当于关系数据库中

的“Select * from”语句，其后的 setFrom()和 setSize()用于控制返回的结果集。和 matchQuery()类似，可以把 matchAllQuery()放到 QueryBuilders 方法中并将结果作为 setQuery 的参数，实现方法见代码段 5.20。

```
//代码段 5.20: matchAllQuery()应用
//import 语句,略
SearchResponse response=client.prepareSearch("baike")
    .setTypes("baike")
    .setSearchType(SearchType.DFS_QUERY_THEN_FETCH)     //参数含义如表 5.1 所示
    .setQuery(QueryBuilders.matchAllQuery())
    .setFrom(0)
    .setSize(10)
    .setExplain(true)
    .execute()
    .actionGet();
for (SearchHit hit: response.getHits().getHits()) {      //遍历检索结果
  System.out.println(hit.getSourceAsString());
}
```

Tips：matchQuery 能够使用某一字段的值对文档进行检索；matchAllQuery 用于匹配文档中的所有字段。

## 5.4.6 MultiMatchQuery

相对于前面的 matchQuery，这里提到的 multiMatchQuery("查询字符串", "field1", "field2",...)针对的是对多个字段的搜索。也就是说，当 multiMatchQuery 中字段列表参数只有一个时，其作用与 matchQuery 相当；而当字段列表有 field1、field2 等多个参数时，则要么是 field1、要么是 field2 等多个参数中包含指定的文本，则都算检索到结果。限于篇幅，此处不再给出代码段，在图 5.6 上部给出相关代码(在 baike 文件中的 content 和 taglist 字段中完成检索)，下部是实际执行结果。

## 5.4.7 BoolQuery

这是一个由其他类型查询组合而成的文档匹配类型的查询，可由一个或者多个查询语句构成，每种语句都有它们的匹配条件，可能的匹配条件如下：

- boolQuery().must (termQuery("字段", "内容"))——待匹配的文档指定字段必须满足查询内容。
- boolQuery().should (termQuery("字段", "内容"))——待匹配的文档可以满足该查询内容。
- boolQuery().must_not(termQuery("字段", "内容"))——待匹配的文档指定字段

```
@Test
public void test13(){
    SearchResponse response = client.prepareSearch("baike").setTypes("baike")
            .setSearchType(SearchType.DFS_QUERY_THEN_FETCH)
            .setQuery(QueryBuilders.multiMatchQuery("中国","content","taglist")) //
            .setFrom(0).setSize(10).setExplain(true)
            .execute().actionGet();
    for (SearchHit hit : response.getHits().getHits()) {
        System.out.println(hit.getSourceAsString());
    }
}

}
```

```
of 1 (in 2s)
C:\Program ...
5:28:27[INFO org.elasticsearch.plugins] [Scott Washington] loaded [], sites []
{"title":"成龙","url":"http://baike.baidu.com/view/3539.htm","content":"成龙，1954年4月7日生于香港中西区，祖籍安徽芜湖，[1]
{"title":"华裔","url":"http://baike.baidu.com/view/71292.htm","content":"读音:  huá yì<br>古指我国中原地区。华裔对举，始见
{"title":"六方会谈","url":"http://baike.baidu.com/view/592990.htm","content":"六方会谈参与国六方会谈的目的是解决朝鲜核危机。
{"title":"亚马逊（网络电子商务公司）","url":"http://baike.baidu.com/view/27780.htm","content":"亚马逊公司是在1995年7月16日由
{"title":"安倍晋三","url":"http://baike.baidu.com/view/90906.htm","content":"安倍晋三（shinzou abe），1 安倍晋三的妻子安倍昭
{"title":"朝鲜半岛","url":"http://baike.baidu.com/view/119146.htm","content":"朝鲜和韩国运动员共同举着朝鲜半岛旗<br>朝鲜半岛
{"title":"中央电视台（中国中央电视台）","url":"http://baike.baidu.com/view/17724.htm","content":"中央广播事业局派罗东和孟启
{"title":"视频网站","url":"http://baike.baidu.com/view/1557113.htm","content":"(1)优酷网<br>优酷网是中国领先的视频分享网站，
{"title":"史蒂芬·哈珀","url":"http://baike.baidu.com/subview/54108/11107133.htm","content":"史蒂芬·哈珀 (Stephen Harper)
```

图 5.6 multiMatchQuery 结果输出

必须不满足该查询内容，但不能只用一个 must_not 语句搜索文档。

上述匹配条件可以组合起来并作为 QueryBuilders 的具体方法。代码段 5.21 给出一个形式化的例子。

```
//代码段 5.21：有关 boolQuery 的形式化表示
QueryBuilder qb=QueryBuilders.boolQuery()
    .must(termQuery("字段 1", "内容 1"))
    .must(termQuery("字段 2", "内容 2"))
    .mustNot(termQuery("字段 3", "内容 3"))
    .should(termQuery("content", "content4"));
```

代码段 5.22 给出实际使用方法，这里是在类型文件 baike 中，在其 content 字段中查询包含有“中国”且不含“日本”且在 taglist 中可以包含“历史”字符的结果集。

```
//代码段 5.22：boolQuery()应用
//import 语句，略
SearchResponse response=client.prepareSearch("baike")
    .setTypes("baike")
    .setSearchType(SearchType.DFS_QUERY_THEN_FETCH)
    .setQuery(QueryBuilders.boolQuery()
      .must(termQuery("content", "中国"))
      .mustNot(termQuery("content", "日本"))
      .should(termQuery("taglist", "历史")))
```

```
        .setFrom(0)
        .setSize(10)
        .setExplain(true)
        .execute()
        .actionGet();
    for (SearchHit hit: response.getHits().getHits()) {
        System.out.println(hit.getSourceAsString());
    }
```

### 5.4.8 TermQuery

termQuery(查询字段，查询词)的作用是在指定的字段中检索指定的查询词，如QueryBuilder qb=QueryBuilders.termQuery("content","中国")是在content字段中查询包括"中国"字符串的结果集。针对baike类型文件的相关实现见代码段5.23。

```
//代码段 5.23: termQuery()应用
//import 语句,略
SearchResponse response=client.prepareSearch("baike")
        .setTypes("baike")
        .setSearchType(SearchType.DFS_QUERY_THEN_FETCH)
        .setQuery(QueryBuilders.termQuery("content", "中国"))
        .setFrom(0)
        .setSize(10)
        .setExplain(true)
        .execute()
        .actionGet();
for (SearchHit hit: response.getHits().getHits()) {
        System.out.println(hit.getSourceAsString());
}
```

### 5.4.9 WildcardQuery

wildcardQuery(查询字段，含有通配符的查询词)的作用是在指定的字段中检索含有通配符的查询词，如 QueryBuilder qb = QueryBuilders.wildcardQuery("content","?国")。有关通配符检索的内容参见本书3.3.3节，此处不再赘述。在baike类型文件中的相关实现见代码段5.24。

```
//代码段 5.24: wildcardString()应用
//import 语句,略
SearchResponse response=client.prepareSearch("baike")
    .setTypes("baike")
    .setSearchType(SearchType.DFS_QUERY_THEN_FETCH)
```

```
    .setQuery(QueryBuilders.wildcardQuery("content", "?国"))
    .setFrom(0)
    .setSize(10)
    .setExplain(true)
    .execute()
    .actionGet();
for(SearchHit hit: response.getHits().getHits()) {
    System.out.println(hit.getSourceAsString());
}
```

### 5.4.10　QueryString

queryString 查询支持 Lucene 所有的查询语法。对给定的内容，它会使用查询解析器来构造实际的查询，如 QueryBuilder qb＝QueryBuilders.queryString("＋中国 -日本")，其含义是搜索包含“中国”且不含“日本”字符串的结果集。相关实现见代码段 5.25。

```
//代码段 5.25: queryString()应用
//import 语句,略
SearchResponse response=client.prepareSearch("baike")
    .setTypes("baike")
    .setSearchType(SearchType.DFS_QUERY_THEN_FETCH)
    .setQuery(QueryBuilders.queryString("+中国 -日本"))      //注意减号前有空格
    .setFrom(0)
    .setSize(10)
    .setExplain(true)
    .execute()
    .actionGet();
for (SearchHit hit: response.getHits().getHits()) {
    System.out.println(hit.getSourceAsString());
}
```

### 5.4.11　MoreLikeThis

可以通过 moreLikeThis()查询到与所提供的文本相似的文档，实现方法见代码段 5.26。首先用 MoreLikeThisRequestBuilder(client, "indexName", "indexType", "id")将 moreLikeThis 查询指向一个特定类型文件下的特定 ID 号的记录，其中最后一个参数可以是提供的用来比较是否相似的样品文档的 ID 号。其后的 moreLikeThis()方法是查询与这个 ID 号对应的文档相似的文档。

```
//代码段 5.26: moreLikeThis ()应用
//import 语句,略
MoreLikeThisRequestBuilder mlt=new MoreLikeThisRequestBuilder(client,
"baike", "baike", "XXXXXX");    //XXXXXX 是待匹配的文档的 id,这里只是形式化的
```

```
                                        //表述,不是真实 ID 号
mlt.setField("content");                //在哪个字段上分析
SearchResponse response=client.moreLikeThis(mlt.request())
    .actionGet();                       //找到与参数 mlt 携带的文档类似的结果集
for (SearchHit hit: response.getHits().getHits()) {
    System.out.println(hit.getSourceAsString());
}
```

还有一种方法是通过构造 Query 进行查询,如图 5.7 所示(限于篇幅,代码段参见图 5.7 的上部,这里不再单独列出实现代码)。其中,setQuery(QueryBuilders. moreLikeThisQuery("content"))是拟在指定的 content 字段中进行相似检索,而其中的子方法 likeText("指定的文本内容")是指检索和指定的文本相似的结果集;方法 minTermFreq()是一篇文档中一个词语至少出现的次数(小于这个值的将被忽略,默认是 2);方法 boost()是设置查询权重(默认是 1)。除此之外,还有一些调节参数,其含义可参阅相关手册,此处不再赘述。

```
@Test
public void test16() {   //moreLikeThis
    SearchResponse response = client.prepareSearch("baike").setTypes("baike")
            .setSearchType(SearchType.DFS_QUERY_THEN_FETCH)
            .setQuery(QueryBuilders.moreLikeThisQuery("content")
                    .likeText("联合国"这一名称是由美国总统富兰克林·D·罗斯福设想出来的，该
            .boost(1.0f)
            .minTermFreq(3))
            .setFrom(0).setSize(10).setExplain(true)
            .execute().actionGet();
    for (SearchHit hit : response.getHits().getHits()) {
        System.out.println(hit.getSourceAsString());
    }
```

```
of 1 (in 2s)
{"title":"金正恩（朝鲜最高领导人）","url":"http://baike.baidu.com/view/108598.htm","content":"留学瑞士<br>金正恩出生于198
{"title":"成龙","url":"http://baike.baidu.com/view/3539.htm","content":"成龙，1954年4月7日生于香港中西区，祖籍安徽芜湖，
{"title":"首尔大学","url":"http://baike.baidu.com/view/142165.htm","content":"首尔大学又称国立首尔大学或首尔国立大学，是
{"title":"江南style","url":"http://baike.baidu.com/view/8988958.htm","content":"“Gangnam Style” 《江南style》专辑封面是
{"title":"朴载相","url":"http://baike.baidu.com/subview/985134/5110102.htm","content":"朴载相（Psy），1977年12月31日出生
```

图 5.7 moreLikeThis 方法使用

> **Tips**：moreLikeThis()方法根据 index、type、id 找到一个文档,再找到跟这个文档相似的其他文档。常用的 moreLikeThis API 有:
>
> - moreLikeThis(MoreLikeThisRequest request)。
> - moreLikeThis(MoreLikeThisRequest request, ActionListener<SearchResponse> listener)——根据 SearchResponse 提供的参数执行相似搜索。
> - prepareMoreLikeThis(String index, String type, String id)——准备执行相似搜索。

### 5.4.12 Filter 概述

有时，我们需要选择索引文件中的一部分数据显示。类似于操纵关系数据库数据的 select 语句中的 where 子句，filter 子句用于对检索结果的选择。和 RESTful DSL 类似，只需在 query 中添加对应的 filter 字段就可以在任何搜索中使用 filter。Filter 只处理文档是否匹配与否，但不涉及文档评分操作。

在 Java 客户端编程中，一个比较简单的做法是把 filter 子句——如 andFilter()、termFilter()等——放在 FilterBuilders()方法中来构建 filter，并将 FilterBuilders 作为 setPostFilter()的参数。为使用 FilterBuilders，应在工程中引入如下的包文件：

```
import org.elasticsearch.index.query.FilterBuilders.*;
```

### 5.4.13 TermFilter

filterBuilders. termFilter(String name, String value) 是用于在指定的字段(由 termFilter 的第一个参数表示)中设定要获取得到的关键词(由 termFilter 的第二个参数表示)。代码段 5.27 的作用是从类型文件 baike 中获取在 content 字段含有“中国”二字的结果集并显示在控制台上。注意，这里是将 termFilter 作为 FilterBuilders 的一部分并被包裹在 setPostFilter()中，而 setPostFilter()又是 prepareSearch()的一个参数。

```
//代码段 5.27: termFilter 的用法
//import 语句，略
SearchResponse response=client.prepareSearch("baike")
  .setTypes("baike")
  .setSearchType(SearchType.DFS_QUERY_THEN_FETCH)
  .setQuery(QueryBuilders.matchAllQuery())                    //此处是 query 部分
  .setPostFilter(FilterBuilders.termFilter("content","中国"))       //filter
  .setFrom(0)
  .setSize(10)
  .setExplain(true)
  .execute()
  .actionGet();
for(SearchHit hit: response.getHits().getHits()) {
  System.out.println(hit.getSourceAsString());
  }
```

### 5.4.14 ExistsFilter

由于 Elasticsearch 可以处理非结构化的大数据，因此在其类型文件中可能存在这种情况，即不同的文档有不同结构，这一点是和关系型数据库完全不一样的。可以使用 existsFilter()来选择出含有指定字段的结果集列表，其参数就是拟显示出的指定字段，如代

码段 5.28 所示，其作用是在类型文件 baike 中查找含有指定字段的结果集。

```
//代码段 5.28: existsFilter 的用法
//import 语句,略
SearchResponse response=client.prepareSearch("baike")
     .setTypes("baike")
     .setSearchType(SearchType.DFS_QUERY_THEN_FETCH)
     .setQuery(QueryBuilders.matchAllQuery())             //query 部分
     .setPostFilter(FilterBuilders.existsFilter("taglist"))//指定字段为 taglist
     .setFrom(0)
     .setSize(10)
     .setExplain(true)
     .execute()
     .actionGet();
for(SearchHit hit: response.getHits().getHits()) {
     System.out.println(hit.getSourceAsString());
  }
```

### 5.4.15 MatchAllFilter

顾名思义，matchAllFilter 可匹配所有内容，可将其放到 FilterBuilders 的方法中并将结果作为 prepareSearch 方法的 setPostFilter 参数，相关实现见代码段 5.29。

```
//代码段 5.29: matchAllFilter 的用法
//import 语句,略
SearchResponse response=client.prepareSearch("baike")
     .setTypes("baike")
     .setSearchType(SearchType.DFS_QUERY_THEN_FETCH)
     .setQuery(QueryBuilders.matchAllQuery())          //query 部分
     .setPostFilter(FilterBuilders.matchAllFilter())   //filter 部分
     .setFrom(0).
     setSize(10).
     setExplain(true)
     .execute()
     .actionGet();
for(SearchHit hit: response.getHits().getHits()) {
     System.out.println(hit.getSourceAsString());
}
```

### 5.4.16 QueryFilter

前面已经介绍了 queryString 的用法，可以把 queryString 子句包裹到 queryFilter 中并将结果作为 setPostFilter()的参数，这样可以使一个 queryString 以 filter 的面目出现，这样

便于 cache 操作，提升处理效率，但可能同时会丢掉诸如高亮、分值排序等 filter 所不支持的部分功能。相应实现方法如代码段 5.30 所示，其所完成的功能和 queryString 一样。

```
//代码段 5.30: queryFilter 的用法
//import 语句,略
SearchResponse response=client.prepareSearch("baike")
    .setTypes("baike")
    .setSearchType(SearchType.DFS_QUERY_THEN_FETCH)
    .setQuery(QueryBuilders.matchAllQuery())
    .setPostFilter(FilterBuilders.queryFilter(QueryBuilders.queryString
("+中国 -美国")) )
    .setFrom(0)
    .setSize(10)
    .setExplain(true)
    .execute()
    .actionGet();
for(SearchHit hit: response.getHits().getHits()) {
    System.out.println(hit.getSourceAsString());
  }
```

### 5.4.17 RangeFilter

rangeFilter("字段",范围)用于对指定范围内容的选择过滤，可以用 from、to 方法设定范围的起点和终点；includeLower(true or false)、includeUpper(true or false)可以设定是否允许大小写字符；gte、gt、lte、lt 分别表示大于等于、大于、小于等于、小于等关系表达式，如下的代码表示选择范围是年龄段在 10～30 之间的记录。

```
FilterBuilders.rangeFilter("age").gte(10) .lt(30);
```

RangeFilter 可放到 FilterBuilders 的方法中，并作为 prepareSearch 的 setPostFilter 参数，相关实现见代码段 5.31。

```
//代码段 5.31: rangeFilter 的用法
//import 语句,略
SearchResponse response=client.prepareSearch("baike")
  .setTypes("baike")
  .setSearchType(SearchType.DFS_QUERY_THEN_FETCH)
  .setQuery(QueryBuilders.matchAllQuery())
  .setPostFilter(FilterBuilders.rangeFilter("lastModifyTime")
    .from("2015-1-1")
    .to("2015-1-31")
    .includeLower(true)
    .includeUpper(false)
```

```
        )
        .setFrom(0)
        .setSize(10)
        .setExplain(true)
        .execute()
        .actionGet();
    for(SearchHit hit: response.getHits().getHits()) {
        System.out.println(hit.getSourceAsString());
    }
```

### 5.4.18 TypeFilter

typeFilter 返回指定类型的所有文档，当查询被定向到多个索引或者一个有大量不同数据类型的索引上时，该过滤器是有用的。例如，用来限制类型文件名为 baike 的过滤器示例如代码段 5.32 所示。

```
//代码段 5.32: typeFilter 的用法
//import 语句,略
SearchResponse response=client.prepareSearch("baike")
    .setSearchType(SearchType.DFS_QUERY_THEN_FETCH)
    .setPostFilter(FilterBuilders.typeFilter("baike"))        //设定 type 名称
    .setFrom(0)
    .setSize(10)
    .setExplain(true)
    .execute()
    .actionGet();
for(SearchHit hit: response.getHits().getHits()) {
    System.out.println(hit.getSourceAsString());
}
```

### 5.4.19 过滤器间的组合：BoolFilter、NotFilter、OrFilter、AndFilter

boolFilter、notFilter、orFilter、andFilter 可以将多个过滤器结果进行组合，但它们之间还是有所区别的。其中，boolFilter 会使用到前面提到过的 Bitset 数据结构，可以重复利用缓存资源；但 notFilter、orFilter、andFilter 是依次逐个处理文档并检查是否满足查询条件，这个过程不用 Bitset，因此也就不能重复利用缓存资源了。如果查询结果集结果很大，那么使用 notFilter、orFilter、andFilter 的性能是低下的，但是有些 filter 必须逐个文档依次处理(如 Geo * filters、Scripts、Numeric_range 等)。另外，如果有多个 filter 条件(即一个 and、or、not 里面包含多个 filter 过滤条件)，一般来说，限制条件较苛刻的可放在前面执行，这样后面的 filter 需要处理的文档数就会小，可提高处理速度。有关 boolFilter 的示例代码如代码段 5.33 所示。

```
//代码段 5.33: boolFilter 的用法
FilterBuilders.boolFilter()
  .must(FilterBuilders.termFilter("tag", "ha"))          //tag 是字段名
  .mustNot(FilterBuilders.rangeFilter("size").from("100").to("2000"))
  .should(FilterBuilders.termFilter("tag", "tagA"))      //tag 是字段名
  .should(FilterBuilders.termFilter("tag", "tagB"));     //tag 是字段名
```

有关 notFilter 的示例代码如代码段 5.34 所示。

```
//代码段 5.34: notFilter 的用法,size 值在 100~500 之间
FilterBuilders.notFilter(
  FilterBuilders.rangeFilter("size")
    .from("100")
    .to("500"));
```

有关 orFilter 的示例代码如代码段 5.35 所示。

```
//代码段 5.35: orFilter 的用法
FilterBuilders.orFilter(
    FilterBuilders.termFilter("content", "中国"),   //第一个条件
    FilterBuilders.termFilter("tag", "tagA")       //第二个条件,tag 是字段名
  );
```

有关 andFilter 的示例代码如代码段 5.36 所示。

```
//代码段 5.36: andFilter 的用法,同时满足下面的两个条件
FilterBuilders.andFilter(
  FilterBuilders.rangeFilter("time")              //第一个条件
    .from("2014-03-01")
    .to("2014-06-01"),
FilterBuilders.prefixFilter("content", "ha")      //第二个条件
);
```

## 5.5 统计分析

### 5.5.1 Facets

在 Elasticsearch 较早版本中完成统计分析功能是 facets。facets 也有多种类型,前面亦有叙述,这里只以其中的 termsFacet、DateHistogramFacet 等为例,介绍其在 Java 客户端的实现。代码段 5.37 给出基于 prepareSearch()的统计实现,这里的 Facet_name1、Facet_name2 分别是 termsFacets、DateHistogramFacet 的名称,在 Facet_name2 中设置了时间步

长。之后,通过 getFacets()方法获取前面得到的统计结果并完成进一步处理。

```
//代码段 5.37: termsFacet 的用法
SearchResponse sr=node.client().prepareSearch()
  .setQuery(QueryBuilders.matchAllQuery())      //构建查询
  .addFacet(FacetBuilders.termsFacet("Facet_name1").field("field_name_1"))
  .addFacet(FacetBuilders.dateHistogramFacet("Facet_name2").field("field_
  name_2").interval("步长"))
  .execute()
  .actionGet();
//使用下面的方式来获取相应的统计数据
TermsFacet f1=(TermsFacet) sr
  .getFacets()
  .facetsAsMap()
  .get("Facet_name1");
DateHistogramFacet f2=(DateHistogramFacet) sr
  .getFacets()
  .facetsAsMap()
  .get("Facet_name2");
//后续的进一步处理,略
```

代码段 5.38 给出了一个比较完整的示例,其作用是分别统计在类型文件 baike 中的 content 字段中出现的词频,并统计在 lastModifyTime 字段中的每一天的数据分布情况。

```
//代码段 5.38: termsFacet 和 DateHistogramFacet 的用法
//import 语句,略
SearchResponse sr=client.prepareSearch("baike")
    .setTypes("baike")
    .setQuery(QueryBuilders.matchAllQuery())      //构建查询
    .addFacet(FacetBuilders.termsFacet("facet1")
                                                //在查询中构建 terms facets 统计
        .field("content"))                        //统计在 content 中的词频
    .addFacet(FacetBuilders.dateHistogramFacet("dateFacet")
                                              //在查询中构建 date histogram 统计
        .field("lastModifyTime")                  //统计字段
        .interval("day"))                         //时间间隔步长
    .execute()                                    //执行搜索
    .actionGet();
TermsFacet f1=(TermsFacet) sr                     //取出 terms facets 统计结果
    .getFacets()
    .facetsAsMap()
    .get("facet1");
for(TermsFacet.Entry entry: f1.getEntries()){     //显示 terms facets 统计结果
    System.out.println(entry.getTerm()+"==="+entry.getCount());
}
```

```
System.out.println("--------------------------------------------");
//下面获取对 lastModifyTime 的统计
DateHistogramFacet f2=(DateHistogramFacet) sr   //取出 date histogram 统计结果
    .getFacets()
    .facetsAsMap()
    .get("dateFacet");
for(DateHistogramFacet.Entry entry: f2.getEntries()){
                                          //显示 date histogram 统计结果
    System.out.println(new Date(entry.getTime())+"==="+entry.getCount());
}
```

### 5.5.2 Aggregations

为了兼顾老版本，Elasticsearch 1.0 以上的版本仍然支持 facets，但是在 Elasticsearch 1.0 以后的版本中，其统计分析功能已经转移到由 aggregations 实现。虽然它们的作用类似，但建议在高版本的 Elasticsearch 中使用 aggregations 完成统计。限于篇幅，这里也不对各种 aggregations 进行详述（详见本书 4.3 节）。代码段 5.39 给出示例，对类型文件 baike 中的 content 字段中含有“中国”字符串的结果集进行统计。其中，在父统计中，完成对 lastModifyTime 字段的以“天”为步长的统计；在子统计中（在每一天的结果集合中），完成对 taglist 字段的统计，如图 5.8 所示，图中给出针对 baike 的每一天中的 taglist 分布统计结果。

```
//代码段 5.39: Aggregations 的用法
//import 语句，略
SearchResponse response=client.prepareSearch("baike")
    .setTypes("baike")
    .setSearchType(SearchType.DFS_QUERY_THEN_FETCH)
    .setQuery(QueryBuilders.termQuery("content", "中国"))  //构建查询
    .addAggregation(AggregationBuilders.dateHistogram("dateAgg")
                                                   //父 Agg 统计
      .field("lastModifyTime")                     //统计字段
      .interval(DateHistogram.Interval.DAY)        //步长
      .subAggregation(AggregationBuilders.terms("tagsAgg")//子统计
          .field("taglist")))                      //子 Agg 统计字段
    .setFrom(0)
    .setSize(90)
    .setExplain(true)
    .execute()
    .actionGet();
DateHistogram agg=response.getAggregations().get("dateAgg");  //得到父统计结果
for(DateHistogram.Bucket bucket: agg.getBuckets()) {          //显示父统计结果
    System.out.println(bucket.getKeyAsDate()+"==="+bucket.getDocCount());
```

```
        Terms termAgg=bucket.getAggregations().get("tagsAgg"); //得到子统计结果
        for (Terms.Bucket bucket1: termAgg.getBuckets()) {      //显示子统计结果
            System.out.println("\t\t\t"+bucket1.getKey()+"==="+bucket1.
            getDocCount());
        }
    }
```

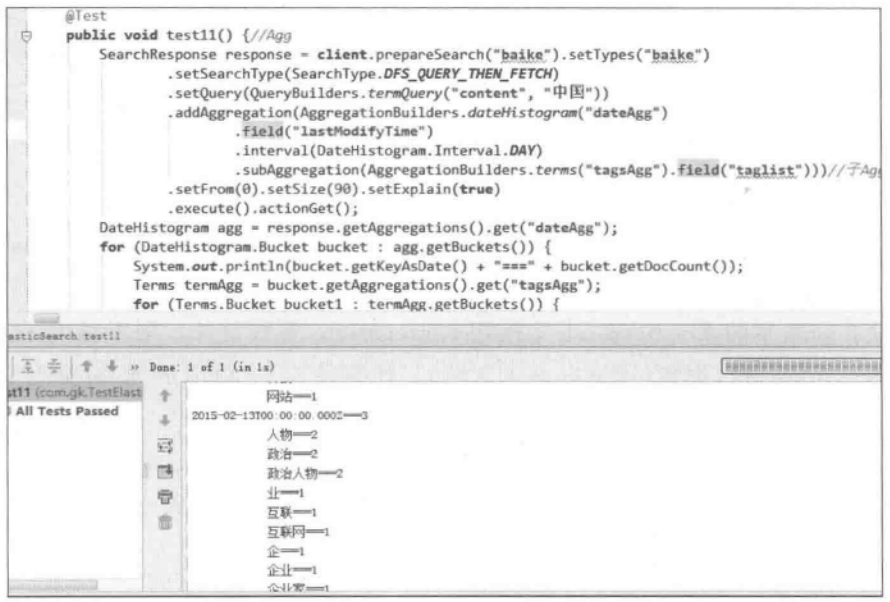


图 5.8　Aggregations 的使用

## 5.6 对检索结果的进一步处理

### 5.6.1 控制每页的显示数量及显示排序依据

在上述的诸多例子中多次出现 setFrom()、setSize()、setExplain()子句。这是在搜索结果(例如 prepareSearch 中)，通过 setFrom(起始检索结果)和 setSize(每页显示结果集)等参数来控制显示数量的一种方案，从而实现对检索结果的分页。同时，也可以通过使用 setExplain(true)参数来设置是不是需要对文档的打分情况进行解释和说明。示例程序如图 5.9 所示。在图 5.9 下方的控制台可看出，检索结果正常，给出了文档的排序依据，而这恰好是 setExplain(true)参数的作用。

Tips：为指定的请求计算排序分数的常用 API 有：

- explain(ExplainRequest request)。
- explain(ExplainRequest request,ActionListener<ExplainResponse>listener)。
- prepareExplain(String index,String type,String id)。

```
    @Test
    public void test9 () throws IOException {//检索操作
        SearchResponse response = client.prepareSearch("test")
                .setTypes("news")
                .setSearchType(SearchType.DFS_QUERY_THEN_FETCH)
                .setQuery(QueryBuilders.termQuery("content", "节日"))
                .setFrom(0).setSize(90).setExplain(true)
                .execute().actionGet();
        System.out.println(response.status());
        System.out.println(response.getHits().getAt(0).getExplanation());
    }
```

```
ne: 1 of 1 (in 1s)
OK
1.1976817 = weight(content:节日 in 0) [PerFieldSimilarity], result of:
  1.1976817 = fieldWeight in 0, product of:
    1.0 = tf(freq=1.0), with freq of:
      1.0 = termFreq=1.0
    1.9162908 = idf(docFreq=1, maxDocs=5)
    0.625 = fieldNorm(doc=0)
```

图 5.9 检索数据并返回排序依据

## 5.6.2 基于 Scroll 方法的检索结果及其分页

既然能控制检索结果的显示数量，就能对检索结果进行翻页处理。Scroll API 不用于实时请求，而是处理大量类似数据，可以用一个语句检索大量甚至所有的结果，这与传统的使用游标查询数据库的方法类似。代码段 5.40 演示了基于 scroll API 的检索结果及其分页功能的实现，此例是对索引中类型文件名为 baike 的索引数据进行检索，例子中提交的检索词为“中国”。

```
//代码段 5.40: 基于 scroll APIs 的检索结果及其分页
//import 语句,略
SearchRequestBuilder searchRequestBuilder=client.prepareSearch("baike")
    .setScroll(TimeValue.timeValueMinutes(1))
    .setSearchType(SearchType.SCAN)
    .setTypes("baike")
    .setQuery(QueryBuilders.termQuery("content", "中国"))
    .setFrom(0)
    .setSize(3)
    .setExplain(true);
SearchResponse response=searchRequestBuilder
    .execute()
    .actionGet();
String sid=response.getScrollId();
while (true) {
    response=client.prepareSearchScroll(sid)
        .setScroll(TimeValue.timeValueMinutes(1))
```

```
            .execute()
            .actionGet();
    for(SearchHit hit: response.getHits().getHits()) {
        System.out.println(hit.getSourceAsString());
    }
    if(response.getHits().getHits().length==0) break;
    System.out.println("=============================");
    }
```

在代码段 5.40 中，首先在 client. prepareSearch()方法中指定 index 和 type 名称，同时设置 setSearchType(此例设置为 SearchType. SCAN，其含义见表 5.1 中的说明)；在其 setQuery ()方法中设置检索字段(content)和检索词(“中国”)；setFrom()和 setSize()等控制显示数量。setScroll()方法中的参数是过期时间，程序中设置的是 1 分钟，1 分钟之后该上下文就不存在了，需重新构建 context。就是说，在 setScroll 中设置每一个 scroll context 的存活时间。

第二，通过 searchRequestBuilder. execute(). actionGet()方法，到每一个数据分片中去检索数据。此时如不使用 scroll 方法，则每一次翻页操作都需要处理相应数据的上下文 context。为提高检索效率，从数据重用的角度出发，可以将相关数据的 context 放到内存中，代码中使用 response. getScrollId()来获取此次检索结果的 context，此时需要比较过期时间(即此次检索结果的生命周期)。此时，response = client. prepareSearchScroll(sid). setScroll(TimeValue. timeValueMinutes(1)). execute(). actionGet()得到的仍只是元结果，尚无具体的检索结果集。

第三，因为返回值中只包含 scrollID，所以需要拿这个 scrollID 进行二次搜索时才会返回结果。具体实现是在 for 循环中，基于 response. getHits(). getHits()方法遍历检索结果集并通过在循环体中使用 getSourceAsString()将每一条检索结果集显示出来。代码中的 response. getHits(). getHits(). length==0 表示检索结果集的末尾。

代码段 5.40 的执行结果如图 5.10 所示(所处理的数据是从“百度百科”中采集到的内容，有关此应用的完整说明及实现方法，详见本书第 9 章)。值得注意的是，代码段 5.40 设置的 setSize(3)并不是实际返回数量 3，而是每一个分片返回的数量，每页实际返回的检索

图 5.10　测试环境的分片数(左)与检索结果集合(右)

集合数量是 setSize 的值再乘以当前分布式环境的分片数量。由于测试环境当前的分片数是 5(如图 5.10 左图所示,可见当前的索引文件 baike 存在于两个 node 上,总的 shard 数量是 5 个),因此每一页的检索结果集合是 3(注：setSize 参数)再乘以 5(注：shard 数量),如图 5.10 右侧所示。

Tips：基于 scroll 方法的迭代搜索结果的常用 API 有：

- searchScroll(SearchScrollRequest request)。
- searchScroll(SearchScrollRequest request, ActionListener<SearchResponse> listener)。
- prepareSearchScroll(String scrollId)。

### 5.6.3 高亮显示检索词

SearchResponse 对象中的 addHighlightedField()方法可以定制在哪个域值的检索结果的关键字上增加高亮显示。setHighlighterPreTags 和 setHighlighterPostTags 分别是添加 HTML 标记的前后缀,如代码段 5.41 所示。从如图 5.11 所示的执行结果中可以看到,定义的高亮标签已经追加到指定的域上了。此处的高亮词是"中国"(见代码段 5.41 中的 termQuery()部分),高亮颜色是红色(见代码段 5.41 中的 setHighlighterPreTags()部分)。

```
//代码段 5.41：高亮显示检索词
//import 语句,略
SearchResponse response=client.prepareSearch("baike")
    .setTypes("baike")
    .setSearchType(SearchType.DFS_QUERY_THEN_FETCH)
    .setQuery(QueryBuilders.termQuery("content", "中国"))
                                            //基于 term Query 构造检索
    .addHighlightedField("content")         //确定高亮字段
    .addHighlightedField("title")           //确定高亮字段
    .setHighlighterPreTags("<span style=\"color:red\">")
                                            //确定高亮结果前要增加的 HTML 标记
    .setHighlighterPostTags("</span>")      //确定高亮结果后要增加的 HTML 标记
    .setFrom(0)
    .setSize(90)
    .setExplain(true)
    .execute()
    .actionGet();
for(SearchHit searchHit: response.getHits()) {
    Map<String, HighlightField>result=searchHit.highlightFields();
```

```
        HighlightField highlightedSummary=result.get("content");
                                                        //从设定的高亮域中取得指定域
        HighlightField highlightedTitle=result.get("title");
                                                        //从设定的高亮域中取得指定域
        String summary="";
        String title="";
        if(highlightedSummary !=null && highlightedTitle !=null) {
            Text[] summaryTexts=highlightedSummary.fragments();
                                                        //取得定义的 content 高亮域
            for(Text text: summaryTexts) {
                summary+=text;
            }
            Text[] titleTexts=highlightedTitle.fragments();
                                                        //取得定义的 title 高亮域
            for (Text text: titleTexts) {
                title+=text;
            }
          }
        System.out.println(summary);
        System.out.println(title);
    }
```

```
(中央电视台前身)试验播出，<span style="color:red">中国</span>自己的电视播出信号第一次出现在北京，<span style="color:red">中国</span>
<span style="color:red">中国</span>中央电视台)
```

图 5.11　高亮输出

## 5.7　扩展知识与阅读

众所周知，中文词法分析是中文信息处理的基础与关键。在 Java 客户端程序设计全文检索程序时，往往需要指定词法分析器，限于篇幅，本章未对 Analyze 部分进行讲述。文献[高凯，2010]对低版本 Lucene 的 Analyze 部分进行了说明；文献[张华平，2014] 对汉语词法分析(包括网络语言分析、新特征语言抽取、自动分类、自动聚类、自动摘要、关键词抽取)等进行了详细的分析，并给出了算法实现。文献[罗刚，2014]也对相关技术实现进行了说明。而有关 Maven 的背景资料，可以参阅文献[许晓斌，2011]。

## 5.8　本章小结

本章给出在 Java 客户端实现 Elasticsearch 提供的部分功能，涵盖简单的信息检索、统计等方面。在 Java 等客户端可以基于 Elasticsearch 提供的功能，实现对应的信息处理等操

作。总的来说，Elasticsearch 部分功能的 Java 客户端实现一般需要如下步骤：首先，通过 maven 引入依赖 jar 包，在 pom.xml 中加入对 Elasticsearch 的依赖；第二，实例化 Elasticsearch client；第三，进行查询，查询形式有多种，可以通过 QueryBuilders 来构建和组合它们。和 Query 类似，可以使用 filter 进行过滤（使用 FilterBuilders 来构建），通过 .setPostFilter(filter) 指定给实例化的搜索对象 search。最后，如果需要，可以设定分页查询，一种方式是指定 from/size；另外一种是使用 scroll/size，可以用 scrollId 继续往下查询，但是这种分页机制必须在设定的缓存时间内完成。

# Elasticsearch 配置与集群管理

> "Elasticsearch comes with reasonable defaults for most settings, so you can try it out without bothering with configuration. Most of the time, these defaults are just fine for running a production cluster." ——— elasticsearch.yml

前面已经对 Elasticsearch 的索引、检索、统计分析等进行了介绍。本章将对 Elasticsearch 的配置、集群管理等进行说明，并对提高索引和查询效率的策略进行了简述。通过对本章的学习，能达到更好地配置和使用 Elasticsearch 的目的。

## 6.1 Elasticsearch 部分基本配置及其说明

Elasticsearch 的大多数配置信息位于"%ES_HOME%/config/elasticsearch.yml"文件中，所有配置都可使用环境变量。另一个是日志配置文件"%ES_HOME%/config/logging.yml"，它对日志进行配置，其设置按普通 Log4j 配置文件进行设置即可。

Elasticsearch.yml 负责设置服务器的默认状态。参考文献[Open，2014a][子猴博客，2014]，本节给出针对 elasticsearch.yml 的部分配置设置信息，包括：

(1) 集群名称 cluster.name——例如"cluster.name: elasticsearch"。设置好以后，会自动发现在同一网段下的节点，如果在同一网段下有多个集群，可用这个属性来区分不同的集群。

(2) 节点名称 node.name——Elasticsearch 启动时会自动创建节点名称，但也可在 node.name 中配置，例如"node.name: "Franz Kafka""。指定节点名称有助于利用 API 访问具体的节点。虽然默认的集群启动时会自己给每个节点初始化一个名称，但建议这里还是由自己设置比较好。

(3) 节点是否为 master 主节点——每个节点都可被配置成为主节点，默认值为 true，如"node.master: true"。在 node.master: true 中进行设置，目的是指定该节点是否有资格被选举成为 node，默认集群中的第一台机器为 master，如果这台机挂了就会重新选举 master。

(4) 设置节点是否存储数据——默认值为 true，在 node. data：true 进行设置。如“node. data：true”。如果希望节点只是一个 master 但不存储数据，则应当设置为代码段 6.1 所示的属性（注：配置中的 # 标记后的文字是注释说明）。

```
#代码段 6.1: 设置节点是 master 但不存储数据
node.master: true
node.data: false
```

如果希望节点只存储数据但不是一个 master，则应当设置为如代码段 6.2 所示的属性。

```
#代码段 6.2: 设置节点不作为 master 但存储数据
node.master: false
node.data: true
```

如果既不希望该节点为一个 master 也不想它存储数据（此时会成为一个 search balance），则应该设置为如代码段 6.3 所示的属性。

```
#代码段 6.3: 设置节点既不是 master 也不存储数据
node.master: false
node.data: false
```

(5) 设置机架编号——可在 node. rack 中进行设置。

(6) 可在 node. max_local_storage_nodes 中设置一台机器能运行的节点数目。

(7) 设置索引的碎片数量——默认值为 5，可在配置文件的 index. number_of_shards 中进行设置。

(8) 设置索引副本数量——默认值为 1，可在配置文件的 index. number_of_replicas 中进行设置。

(9) 设置配置文件的存储路径——path. conf：/path/to/conf，默认是 Elasticsearch 根目录下的 config 文件夹。

(10) 设置分配给当前节点的索引数据所在的位置——可在配置文件的 path. data：/path/to/data 中进行设置，默认是 Elasticsearch 根目录下的 data 文件夹，可以选择包含一个以上的位置，用逗号隔开，这样使得数据在文件级别可跨越位置，在创建时就有更多的自由路径可供选择。

(11) 设置临时文件位置——可在 path. work：/path/to/work 中进行设置，默认是 Elasticsearch 根目录下的 work 文件夹。

(12) 设置日志文件所在位置——可在 path. logs：/path/to/logs 中进行设置，默认是 Elasticsearch 根目录下的 logs 文件夹。

(13) 设置插件安装的位置——可在 path. plugins：/path/to/plugins 中进行设置，默认是 Elasticsearch 根目录下的 plugins 文件夹。

(14) 设置绑定的 IP 地址——可以是 IPV4 或 IPV6 的，默认为 0.0.0.0。默认情况下 Elasticsearch 使用 0.0.0.0 地址，并为 HTTP 传输开启 9200-9300 端口，为节点到节点的通信开启 9300-9400 端口，也可自行设置 IP 地址，可在配置文件的 network.bind_host 和 network.publish_host 中进行设置。

(15) 设置节点与其他节点交互的 TCP 端口——默认是 9300，可在配置文件的 transport.tcp.port 中进行设置。

(16) 设置是否压缩 TCP 传输时的数据——默认为 False，可在配置文件的 transport.tcp.compress 中进行设置。

(17) 设置为 HTTP 传输监听定制的端口——默认是 9200，可在配置文件的 http.port 中进行设置。

(18) 设置是否使用 HTTP 协议对外提供服务——默认为 true，可在配置文件的 http.enabled 中进行设置。

(19) 设置内容的最大长度——默认是 100mb，可在配置文件的 http.max_content_length 中进行设置。

(20) 设置集群中主 Master 节点的数量——当多于三个节点时，该值可在 2～4 之间，可在配置文件的 discovery.zen.minimum_master_nodes 中进行设置。

(21) 设置 ping 其他节点时的超时时间——可在配置文件的 discovery.zen.ping.timeout 中进行设置。

(22) 设置 gateway 的类型——默认为 local(即为本地文件系统)，可设置为本地文件系统，分布式文件系统，Hadoop 的 HDFS，和 Amazon 的 S3 服务器，如：gateway.type：local。

> Tips：Gateway 是一种存储集群中各节点元数据的状态方式，这里的元数据主要用来记录所有的索引在创建时各自的设置和明确的类型映射。每次当元数据改变，比如一个索引被加入或被删除，这些变化都会通过 gateway 存储起来。当集群启动时，这些状态将会从 gateway 中读取并应用。

(23) 设置集群中 $N$ 个节点启动时进行数据恢复——默认为 1，可在配置文件的 gateway.recover_after_nodes 中进行设置。

(24) 设置初始化数据恢复进程的超时时间——默认是 5 分钟，可在配置文件的 gateway.recover_after_time 中进行设置。

(25) 设置这个集群中节点的数量——默认为 2，一旦这 $N$ 个节点启动，就会立即进行数据恢复，可在 gateway.expected_nodes 中进行设置。

(26) 初始化数据恢复时并发恢复线程的个数——默认为 4，可在配置文件的 cluster.routing.allocation.node_initial_primaries_recoveries 中进行设置。

(27) 设置添加删除节点或负载均衡时并发恢复线程的个数——默认为 4，可在配置文件的 cluster.routing.allocation.node_concurrent_recoveries 中进行设置。

(28) 设置数据恢复时限制的带宽——如 100mb，默认为0(即无限制)，可在配置文件的 indices. recovery. max_size_per_sec 中进行设置。

(29) 设置参数来限制从其他分片恢复数据时最大同时打开并发流的个数——默认为 5，可在配置文件的 indices. recovery. concurrent_streams 中进行设置。

(30) 设置参数来保证集群中的节点可以知道其他 *N* 个有 master 资格的节点——默认为 1，对于大的集群来说，可以设置较大值(2～4)，可在配置文件的 discovery. zen. minimum _master_nodes 中进行设置。

(31) 设置集群中自动发现其他节点时 ping 连接超时时间——默认为 3 秒，对于比较差的网络环境可以设置为较大的值来防止自动发现时出错，可在配置文件的 discovery. zen. ping. timeout：中进行设置。

(32) 设置是否打开多播发现节点——默认是 true，可在配置文件的 discovery. zen. ping. multicast. enabled 中进行设置。

(33) 设置集群中 master 节点的初始列表——可通过这些节点来自动发现新加入集群节点：discovery. zen. ping. unicast. hosts：["host1", "host2：port", "host3[portX-portY]"]

(34) 在配置文件中也可以设置采用的中文分词器——如代码段 1.1 所示(注：来源于 elasticsearch. yml 配置文件)。此处不再赘述。

：yml 是 Elasticsearch 主要的配置文件，几乎所有的配置都在 yml 文件里完成。

## 6.2 提高索引和查询效率的策略

Elasticsearch 的索引是基于倒排索引机制完成的。从索引优化的角度出发，在建立索引时，要考虑到影响索引速度的因素有：

- Shard 数量。
- 节点数量。
- 索引操作(如合并、优化、索引、写操作等)。
- 磁盘 IO 次数及速度。

从提高效率的角度出发，可以从如下几点来考虑提高索引工作效率：

- Client 端减少频繁的连接并提高效率(如在可能的前提下使用 TCP 长连接、采用多线程机制、建立连接池等)。
- 尽量减少索引大小(索引前预处理、过滤等)。
- 合理规划映像。
- 合理使用分词。

从查询优化的角度来说，可以从如下几点入手来提高查询效率：

- 使用 filter。
- 合理规划 index 和 shard。

除了上述这些策略外，路由选择也是经常要用到的。由于 Elasticsearch 的信息往往分布在不同的 shards 中，因此在搜索时，大多数情况下需要遍历所有 shards 分片以便能检索到相关信息。其实，在某些情况下，如果能指定特定的 shards，即显式地指定路由，有时是能够提高检索效率的。这就需要在建立索引 mapping 时，通过参数_routing 参数来指定路由信息。对同样的 userid 取值会得到同样的哈希值，因此特定用户的所有文档会被放置在一个分片上。在检索时，利用哈希值就只需一个分片而非遍历所有分片。

提供路由最简单的方法是在建立 mapping 时使用_routing 参数。代码段 6.4 给出对 example1 建立索引映像时指定_routing 的方法。

```
#代码段 6.4: _routing 参数
"mappings": {
    "example1": {
        "_routing": {                              //设置路由参数
        "required": true,
        "path": userId
        },
        "properties": {
            "user": {  "type": "string",
                       "index": "not_analyzed"
                       "userId": {"type": "long", "store": "yes"}
                    }
        }
    }
}
```

也可基于上述方法在 HTTP API 中直接使用 routing 参数(不必在 mapping 中添加_routing 字段)。具体示例如下：

```
curl -XGET 'localhost:9200/indexname/typename/_search?routing=10&q=userId:
10'
```

## 6.3 监控集群状态

通过 HTTP 接口方式监控集群状态，例如 http://localhost:9200/_cluster/health?pretty，可以观察到当前集群系统的健康状况。图 6.1 和图 6.2 分别是针对两个不同的 Elasticsearch 集群系统健康状况。图中列出的一些指标如下(其中的英文含义非常明确，这里不再解释)。其实，对于一个 Elasticsearch 集群系统来说，其 status 的取值可以有如下几种：

- green——当 Elasticsearch 能够根据配置分配所有分片和副本时，如图 6.1 所示。
- yellow——主分片已经分配完毕，已经做好可以处理请求的准备，但是某些副本尚

未完成分配,例如当只有一个节点却同时有多个副本时,因为尚无其他节点来放置这些副本,因此其状态值可能就是 yellow 的。如图 6.2 所示。


```
{
  "cluster_name" : "elasticsearch",
  "status" : "green",
  "timed_out" : false,
  "number_of_nodes" : 4,
  "number_of_data_nodes" : 3,
  "active_primary_shards" : 44,
  "active_shards" : 88,
  "relocating_shards" : 0,
  "initializing_shards" : 0,
  "unassigned_shards" : 0
}
```


图 6.1 健康的 green 集群状态


```
{
  "cluster_name" : "elasticsearch",
  "status" : "yellow",
  "timed_out" : false,
  "number_of_nodes" : 4,
  "number_of_data_nodes" : 3,
  "active_primary_shards" : 44,
  "active_shards" : 87,
  "relocating_shards" : 0,
  "initializing_shards" : 1,
  "unassigned_shards" : 0
}
```


图 6.2 亚健康的 yellow 集群状态

- red——集群目前尚未准备就绪,可能至少一个主分片没有准备好。

通过在 Elasticsearch 集群 IP 地址后加上/_cluster/health?pretty,再加上 &level=indices(例如,http://localhost:9200/_cluster/health?pretty&level=indices,如图 6.3 所示)或者 &level=indices 参数(例如,http://localhost:9200/_cluster/health?pretty&level=shards,如图 6.4 所示),可以返回更加详细的集群状态信息。图 6.4 不仅有索引 indices 的更加详细的信息,还有其分片 shards 的状态信息。


```
{
  "cluster_name" : "elasticsearch",
  "status" : "green",
  "timed_out" : false,
  "number_of_nodes" : 4,
  "number_of_data_nodes" : 3,
  "active_primary_shards" : 44,
  "active_shards" : 88,
  "relocating_shards" : 0,
  "initializing_shards" : 0,
  "unassigned_shards" : 0,
  "indices" : {
    ".marvel-2015.02.24" : {
      "status" : "green",
      "number_of_shards" : 1,
      "number_of_replicas" : 1,
      "active_primary_shards" : 1,
      "active_shards" : 2,
      "relocating_shards" : 0,
      "initializing_shards" : 0,
      "unassigned_shards" : 0
    },
    ".marvel-2015.02.15" : {
      "status" : "green",
      "number_of_shards" : 1,
      "number_of_replicas" : 1,
      "active_primary_shards" : 1,
      "active_shards" : 2,
      "relocating_shards" : 0,
      "initializing_shards" : 0,
      "unassigned_shards" : 0
    },
    ".marvel-2015.02.25" : {
      "status" : "green",
      "number_of_shards" : 1,
      "number_of_replicas" : 1,
      "active_primary_shards" : 1,
      "active_shards" : 2,
      "relocating_shards" : 0,
      "initializing_shards" : 0,
      "unassigned_shards" : 0
    },
    ".marvel-2015.02.16" : {
      "status" : "green",
```


图 6.3 增加参数 level=indices 返回的集群信息


```
{
  "cluster_name" : "elasticsearch",
  "status" : "green",
  "timed_out" : false,
  "number_of_nodes" : 4,
  "number_of_data_nodes" : 3,
  "active_primary_shards" : 44,
  "active_shards" : 88,
  "relocating_shards" : 0,
  "initializing_shards" : 0,
  "unassigned_shards" : 0,
  "indices" : {
    ".marvel-2015.02.24" : {
      "status" : "green",
      "number_of_shards" : 1,
      "number_of_replicas" : 1,
      "active_primary_shards" : 1,
      "active_shards" : 2,
      "relocating_shards" : 0,
      "initializing_shards" : 0,
      "unassigned_shards" : 0,
      "shards" : {
        "0" : {
          "status" : "green",
          "primary_active" : true,
          "active_shards" : 2,
          "relocating_shards" : 0,
          "initializing_shards" : 0,
          "unassigned_shards" : 0
        }
      }
    },
    ".marvel-2015.02.15" : {
      "status" : "green",
      "number_of_shards" : 1,
      "number_of_replicas" : 1,
      "active_primary_shards" : 1,
      "active_shards" : 2,
      "relocating_shards" : 0,
      "initializing_shards" : 0,
      "unassigned_shards" : 0,
      "shards" : {
        "0" : {
          "status" : "green",
```


图 6.4 增加参数 level=shards 返回的集群信息

类似地，也可以监控节点状态，它可以告诉我们集群在工作过程中发生了什么。和监控集群状态类似，只需在 URL 后给出相应节点的信息(例如，如果监控 Presence 节点，只需在 URL 后的/_nodes 参数后添加节点名称及拟查询的统计信息即可)，如 http://localhost:9200/_nodes/Presence/stats? os&jvm&pretty，如图 6.5 所示。可以直接指明的可用标记信息有：

- indices——得到与索引状态 API 返回信息和 cache 使用状况相关的信息。
- os——获得服务器的正常运行时间、负载等信息。
- process——获得进程的内存与 CPU 等的使用信息。
- JVM——获得 Java 虚拟机的内存和垃圾回收的信息。
- network——获得 TCP 层的信息。
- transport——获得关于传输模块发送和接收数据的信息。
- http——获得 HTTP 连接的信息。
- FS——获得关于可用磁盘空间和 I/O 操作的统计信息。
- thread_pool——获得分配给不同操作的线程状态信息。
- all——获得上面提到的所有信息。

localhost:9200/_nodes/Presence/stats?os&jvm&pretty


```
{
  "cluster_name" : "gsElasticSearch",
  "nodes" : {
    "2DdZaruET4C4ROzOAaaalg" : {
      "timestamp" : 1425336097348,
      "name" : "Presence",
      "transport_address" : "inet[hebust/10.81.10.105:9300]",
      "host" : "hebust",
      "ip" : [ "inet[hebust/10.81.10.105:9300]", "NONE" ],
    · "indices" : {
        "docs" : {
          "count" : 152377,
          "deleted" : 3
        },
        "store" : {
          "size_in_bytes" : 108275900,
          "throttle_time_in_millis" : 248
        },
        "indexing" : {
          "index_total" : 438,
          "index_time_in_millis" : 448,
          "index_current" : 0,
          "delete_total" : 0,
          "delete_time_in_millis" : 0,
          "delete_current" : 0,
          "noop_update_total" : 0,
          "is_throttled" : false,
          "throttle_time_in_millis" : 0
        },
        "get" : {
          "total" : 0,
          "time_in_millis" : 0,
          "exists_total" : 0,
          "exists_time_in_millis" : 0,
          "missing_total" : 0,
          "missing_time_in_millis" : 0,
          "current" : 0
        },
        "search" : {
          "open_contexts" : 0,
          "query_total" : 14,
          "query_time_in_millis" : 161,
          "query_current" : 0,
          "fetch_total" : 1,
          "fetch_time_in_millis" : 14,
          "fetch_current" : 0
        },
```


图 6.5　节点状态信息

## 6.4 控制索引分片与副本分配

集群中的索引可能由多个分片构成,且每个分片可能拥有多个副本。通过将一个单独的索引分成多个分片,可以处理由于文件太大而不能在一个单一机器上运行的大型索引。由于每个分片可以有多个副本,通过将副本分配到多个服务器上可以处理更高的查询负载。为了进行分片和副本分配操作,Elasticsearch 需要确定将这些分片和副本放在集群的什么地方,即需要确定每个分片和副本分配到哪一个服务器/节点之上[Rafa, 2015]。

可以对一个索引指定每个节点上的最大分片。例如,如果希望 baike 索引在每个节点有两个分片,可以运行代码段 6.5 所示的命令。

```
#代码段 6.5: 指定每个节点上的最大分片
curl -XPUT 'localhost:9200/baike/_settings' -d '{
   "index.routing.allocation.total_shards_per_node": 2
}'
```

这个属性可以放置在 elasticsearch.yml 文件,或使用上面的命令在活动索引上更新[Rafa,2015]。也可以手动移动分片和副本。可以使用 Elasticsearch 提供的_cluster/reroute REST 端点进行控制。

假设有两个节点 node1 和 node2,Elasticsearch 在第一个节点 node1 上分配了 baike 索引的两个分片。假设希望将第二个分片移动到第二个节点 node2 上,可以采用如下方式,见代码段 6.6。

```
#代码段 6.6: 移动分片
curl -XPOST 'localhost:9200/_cluster/reroute' -d ' {
  "commands": [ {
    "move": {                     //移动分片
    "index": "baike",             //待移动的索引文件
    "shard": 1,                   //指定希望移动的分片个数
    "from_node": "node1",         //源
    "to_node": "node2"            //目的地
  }
  } ]
  }'
```

如果希望取消一个正在进行的分配过程,可以通过运行 cancel 命令来指定希望取消分配的索引、节点以及分片,如代码段 6.7 所示。

```
#代码段 6.7: 取消分片
curl -XPOST 'localhost:9200/_cluster/reroute' -d ' {
  "commands": [ {
```

```
    "cancel": {                 //取消操作
    "index": "baike",           //索引文件
    "shard": 1,                 //指定拟取消在 node1 上的 baike 索引的第 1 个分片的分配
    "node": "node1"             //指定取消在哪个节点的操作
  }
  } ]
  }'
```

另外，还可以将一个未分配的分片分配到一个指定的节点上。假定 baike 索引上有一个编号为 1 的分片尚未分配，现希望将其分配到 node2 上，可使用代码段 6.8 的方式来实现。

```
#代码段 6.8: 分配分片
curl -XPOST 'localhost:9200/_cluster/reroute' -d ' {
  "commands": [ {
    "allocate": {               //分配分片
    "index": "baike",           //索引文件
    "shard": 1,                 //指定拟分配分片的编号
    "node": "node2"             //指定将分片分配到哪个节点
  }
  } ]
  }'
```

## 6.5 扩展知识与阅读

本章对 Elasticsearch 的配置、集群管理等进行说明，并对提高索引和查询效率的策略进行了简述。文献[Rafa, 2015]给出了路由选择方法、索引别名及其用途方面的叙述，给出了监控集群状态与健康状况的 API 的使用方法，并对常用的集群诊断工具进行了简介。同时，对一些问题的处理(如为什么靠后页面中的结果会比较慢、控制集群再平衡、验证查询等)也给出了建议。

## 6.6 本章小结

本章首先从配置文件方面介绍了 Elasticsearch 的基本配置。虽说默认的配置文件已经能够应付大多数情况，但了解有关 yml 配置文件的来龙去脉，还是很有必要的。除此之外，本章介绍了提高索引和查询效率的策略，介绍了_routing 参数的使用，这在有些情况下是有用的。通过监控集群状态，可以使管理员了解集群的整体运行情况，这对及时发现可能存在的问题是很有必要的。而手动控制索引分片与副本分配，能更有效地调节集群状态，在很多情况下可能也是必需的。

Chapter 7 第7章

# 基于 Logstash 的日志处理

> "Logstash is a tool for managing events and logs. You can use it to collect logs, parse them, and store them for later use. Speaking of searching, Logstash comes with a web interface for searching and drilling into all of your logs. It is now a part of the Elasticsearch family. This allows us to build better software much faster as well as offering production support."________ http://logstash.net/

随着大数据、大型综合网站以及 Web 2.0 技术的普及，越来越多的软件开发者需要处理海量的日志信息，通常采用的解决办法就是在中央系统上由专门的日志服务器收集日志，这个做法可以帮助用户通过 SSH 在多台服务器之间浏览查找时提供方便，诸如 SPLUNK 这些商用工具就提供了这种日志服务。但是传统的日志系统接入周期较长，而且常常不是任何字段都可以搜索得到的，且处理速度比较慢，当搜索结果太多时处理效果不佳，而且统计数据都是预先聚合好的，无法按需实时计算，如果新增加一个维度时，往往也要进行二次开发。当日志信息越来越大的时候，从快速增长的日志数据流中提取出所需的信息，并将其与其他相关联的事件进行关联，可能将变得越加困难。

作为 ELK(Elasticsearch＋Logstash＋Kibana)家族的重要一员，Logstash 提供了一个很好的解决方案，而且能与上游的 Elasticsearch 和下游的 Kibana 有效衔接。Logstash 由 JRuby 编写，是一个基于消息的简单架构，运行在 Java 虚拟机上。不同于分离的代理端(agent)或主机端(server)，Logstash 可配置单一的代理端并与其他开源软件结合，以实现不同的功能。

## 7.1 概述

其实，Logstash 本身并不产生日志，它只是一个内置有分析和转换工具的日志管理工具，是一个接收、处理、转发日志的"管道"。利用它，可以帮助编程人员将日志等信息从分布式服务器移动到另一个地方。它不但可以接受多种格式的日志输入，还可以解析日志，也可以将多种格式的日志输出到不同的目的地。图 7.1 引用了在官方网站上介绍 Logstash 的示意图。

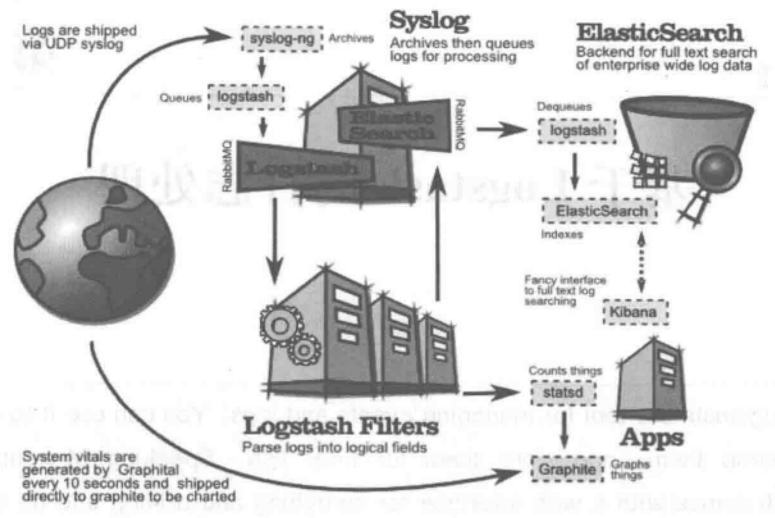


图 7.1 Logstash 示意图

在使用 Logstash 前，首先要确保安装好 Java JDK。之后，需要从 Logstash 官网(http://www.logstash.net/)下载安装包并解压。一般需要创建一个定义 input、output 等内容的配置文件，可以在文件内写入数据来源和目的地、是否对日志数据进行某种格式转换等，并在启动 Logstash 时指定拟采用配置文件的名称(如本章后面以 logstash\bin\conf.conf 作为配置文件)。

Tips：本书采用的 Logstash 版本是目前的最新版 1.4.2。本章给出的 Java 测试程序的 IDE 是 IEDA，使用者也可根据情况选用其他的 IDE，如 Eclipse 等。

代码段 7.1 给出 Logstash 的配置文件信息，其设置的数据输入为从键盘输入的信息，输出的信息也在屏幕上显示(按照 Logstash 配置文件的规定，代码段 7.1 中以 # 来进行注释说明)。

```
#代码段 7.1：在 logstash 的配置文件中设置数据来源和输出目的地
input {                      #数据来源地
    stdin{}
}
output {                     #输出输出地
    stdout{}
}
```

Tips：stdin 是标准输入，stdout 是标准输出。

可以使用下面的命令来启动 bin 目录下的配置文件。这里的启动参数 agent 的作用是指定 Logstash 作为一个基础代理，-f 参数的作用是要指定一个 Logstash 配置文件(此例为

conf.conf)。

```
./logstash agent -f conf.conf
```

之后,系统会等待用户输入,用户可以通过键盘来输入一些字符(如代码段 7.1 所示),输入完毕之后,屏幕上会出现如类似图 7.2 所示的内容(其中的“hello world”是用户输入的内容)。从返回的内容中可以看到,其输出结果中包含了时间戳、hostname(注:测试环境中的 hostname 是本机名称 hebust)和使用者输入的内容。如果能看到类似的结果,说明 Logstash 已经可以正常运行了。当然,如果想退出,可使用 CTRL-C 来退出 Logstash。

```
D:\Logstash-1.4.2\bin>logstash agent -f conf.conf
hello world
2014-11-15T02:01:16.963+0000 hebust hello world
```

图 7.2 Logstash 运行后界面

在 Logstash 中,inputs、codecs、filters、outputs 构成了其核心配置。其中,inputs 部分负责将某种格式的日志数据传输到 Logstash 中;codecs 用于对数据流进行某种格式化处理,如 JSON、msgpack、plain(text)等,它可作为 input、output 中的一部分,帮助分隔或格式化数据;filters 用于过滤信息;outputs 是 Logstash 的输出部分。在 Logstash 的官方网站中给出其所支持的部分 inputs、codecs、filters、outputs 的内容列表,如图 7.3 所示。限于篇幅,本章仅挑选部分常用内容进行介绍。

| inputs | codecs | filters | outputs |
|---|---|---|---|
| collectd | cloudtrail | advisor | boundary |
| drupal_dblog | collectd | alter | circonus |
| elasticsearch | compress_spooler | anonymize | cloudwatch |
| eventlog | dots | checksum | csv |
| exec | edn | cidr | datadog |
| file | edn_lines | cipher | datadog_metrics |
| ganglia | fluent | clone | elasticsearch |
| gelf | graphite | collate | elasticsearch_http |
| gemfire | json | csv | elasticsearch_river |
| generator | json_lines | date | email |
| graphite | json_spooler | dns | exec |
| heroku | line | drop | file |
| imap | msgpack | elapsed | ganglia |
| invalid_input | multiline | elasticsearch | gelf |
| irc | netflow | environment | gemfire |
| jmx | noop | extractnumbers | google_bigquery |
| log4j | oldlogstashjson | fingerprint | google_cloud_storage |
| lumberjack | plain | gelfify | graphite |
| pipe | rubydebug | geoip | graphtastic |
| puppet_facter | spool | grep | hipchat |
| rabbitmq | | grok | http |
| rackspace | | grokdiscovery | irc |
| redis | | i18n | jira |
| relp | | json | juggernaut |

图 7.3 Logstash 所支持的部分 inputs、codecs、filters、outputs 列表

## 7.2 Input: 处理输入的日志数据

可以把 Logstash 近似看成是一个对日志信息进行处理的"黑箱"或"管道"。由于日志数据的来源可以是多种多样的，因此可以在 Logstash 中的 input 部分设置对不同来源的数据进行处理的方法。

> Tips: Logstash 可以多个不同的数据源作为日志输入端，如 TCP/UDP、文件、系统日志、Microsoft Windows EventLogs、Redis 存储系统以及其他的数据源等。通过使用过滤机制，可以允许编程人员修改、操纵这些数据和事件。

### 7.2.1 处理基于 File 方式输入的日志信息

修改代码段 7.1，通过 file 方式，可以从指定的文件中读取数据并输入到 Logstash。基于这一特性，可以监控某些程序的日志文件(要求这些日志文件的格式是以行来组织的)。例如，可以修改代码段 7.1 为代码段 7.2 所示内容(注：涉及 Logstash 和 Kibana 相关配置文件中的代码采用#标注的方式，#后的文字是说明)。

```
#代码段 7.2: logstash 的形式化配置文件，设计基于 file 方式读取指定文件
input {
  file {
    codec=>...                      #可选项，默认是 plain，可通过这个参数设置编码方式
    discover_interval=>...          #可选项，指 logstash 隔多久去检查被监听的路径下是
                                    #否有新文件，默认 15 秒
    exclude=>...                    #可选项，不想被监听的文件可在这里指定
    sincedb_path=>...               #可选项，如果不想用默认的 $ HOME/.sincedb，可在这
                                    #里配置定义文件到其他位置
    sincedb_write_interval=>...     #可选项，指 logstash 每隔多久写一次 sincedb 文件，
                                    #默认 15 秒
    path=>...                       #必选项，指定需处理的文件路径
    stat_interval=>...              #可选项，指 logstash 隔多久检查一次被监听文件状态
                                    #是否有更新，默认 1 秒
    start_position=>...             #可选项，指 logstash 从什么位置开始读取文件数据，
                                    #默认是结束位置，如要导入原有数据，将其设定改成
                                    #beginning，logstash 就从头开始读取
    tags=>...                       #可选项
    type=>...                       #可选项
  }
}
```

依照这种方式，这里给出一个配置文件实例，见图 7.4。在这里指定了待处理的日志文件的路径，其中的参数 start_position 设为 beginning，是指从该文件的头开始处理。

```
input {
    file {
      path => "E:\Tools\elasticsearch-1.4.2\logs\gsElasticSearch.log"
      start_position => "beginning"
      sincedb_path => "E:\Tools\elasticsearch-1.4.2\logs"
    }
}

output {
    stdout{}
}
```

图 7.4 conf.conf 配置文件

### 7.2.2 处理基于 Generator 产生的日志信息

Generator 可产生指定的或随机的数据流，代码段 7.3 给出在 input 段中使用 generator 的方法，数据输出则是用 stdout()方法在屏幕上输出，实际效果如图 7.5 所示（图 7.5 的左侧是配置文件，请注意这里的 lines 和 count 的用法；图 7.5 的右侧是实际执行效果）。

```
#代码段 7.3: logstash 的形式化配置文件,设计基于 generator 的日志输入
input {
  generator {
    codec=>...        #可选项, 默认 plain
    count=>...        #可选项, 默认 0,可设置发送多少次,默认不限制
    lines=>...        #可选项,可以数组格式顺序发送
    message=>...      #可选项,可写成指定字符串
    tags=>...         #可选项,标记日志,可用于后续处理工作
    threads=>...      #可选项, 默认 1,可以设置用来发送日志信息的线程数
    type=>...         #可选项,默认是 string 类型
  }
}
```

```
conf.conf
1  input {
2   generator {
3     codec => "plain"
4     count => 5
5     lines => ["lizi-1","lizi-2"]
6     tags => ["t1","t2"]
7     threads => 2
8     }
9   }
10  output {
11      stdout{}
12  }
13
14
```

```
2015-02-11T02:57:11.234+0000 hebust lizi-2

D:\Logstash-1.4.2\bin>logstash agent -f conf.conf
2015-02-11T02:58:02.635+0000 hebust lizi-1
2015-02-11T02:58:02.635+0000 hebust lizi-1
2015-02-11T02:58:02.637+0000 hebust lizi-2
2015-02-11T02:58:02.637+0000 hebust lizi-2
2015-02-11T02:58:02.638+0000 hebust lizi-1
2015-02-11T02:58:02.638+0000 hebust lizi-1
2015-02-11T02:58:02.638+0000 hebust lizi-2
2015-02-11T02:58:02.638+0000 hebust lizi-2
2015-02-11T02:58:02.639+0000 hebust lizi-1
2015-02-11T02:58:02.639+0000 hebust lizi-1
2015-02-11T02:58:02.640+0000 hebust lizi-2
2015-02-11T02:58:02.640+0000 hebust lizi-2
2015-02-11T02:58:02.641+0000 hebust lizi-1
2015-02-11T02:58:02.641+0000 hebust lizi-1
2015-02-11T02:58:02.642+0000 hebust lizi-2
2015-02-11T02:58:02.642+0000 hebust lizi-2
2015-02-11T02:58:02.643+0000 hebust lizi-1
2015-02-11T02:58:02.643+0000 hebust lizi-1
2015-02-11T02:58:02.643+0000 hebust lizi-2
2015-02-11T02:58:02.644+0000 hebust lizi-2

D:\Logstash-1.4.2\bin>
```

图 7.5 基于 generator 的日志输入及实际输出

### 7.2.3 处理基于 Log4j 的日志信息

通过在 Logstash 的配置文件的 input 部分配置有关 log4j 的部分，可将 log4j 的日志信息通过 TCP Socket，从 Socket Appender 输出到 Logstash 里。代码段 7.4 给出了在 Logstash 端处理基于 log4j 的日志信息的配置文件情况。

```
#代码段 7.4: logstash 的形式化配置文件，设计基于 log4j 的日志输入
input {
  log4j {
    codec=>...              #可选项，默认 plain
    data_timeout=>...       #可选项，默认 5
    host=>...               #可选项，默认 0.0.0.0
    mode=>...               #string 类型的可选项，["server", "client"]，默认"server"
    port=>...               #可选项，默认 4560
    tags=>...               #可选项，标记日志，可用于后续处理工作
    type=>...               #可选项，默认 string 类型
  }
}
```

代码段 7.5 给出了处理基于 log4j 日志数据的方法。首先，要在 Java 程序中添加 log4j 的 JAR 包并进行设置，如图 7.6 所示，以便将在 log4j 中输出的信息作为 Logstash 的日志数据输入流。其次，设计有关处理日志信息的 Java 应用程序，实现代码如代码段 7.5 所示。

```
//代码段 7.5: 在 Java 应用程序中设计处理日志数据的方法
package com.gk;
import org.apache.log4j.Logger;              //添加相应的包
//添加其他的包文件，略
public class HelloLiZi {
    private static Logger LOG=Logger.getLogger(HelloLiZi.class);
    public static void main(String[] args) {
    for(int i=0;i<=100;i+=3)
        LOG.info("lizi---"+i);
    }
}
```

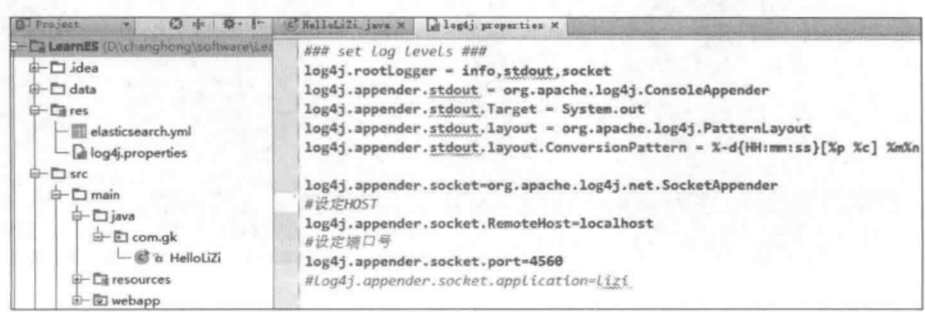


图 7.6 设置 log4j 的相关属性

> Tips：套接字输出(SocketAppender)用于将日志数据通过 TCP 协议发送给远程服务器，SocketAppender 会与远程服务器建立 Socket 连接并将信息封装为 LoggingEvent 对象，串行化后发送给对方，输出类为 org.apache.log4j.net.SocketAppender。

运行程序后，就会将此程序的输出作为 Logstash 的输入数据，如图 7.7 右侧图所示。

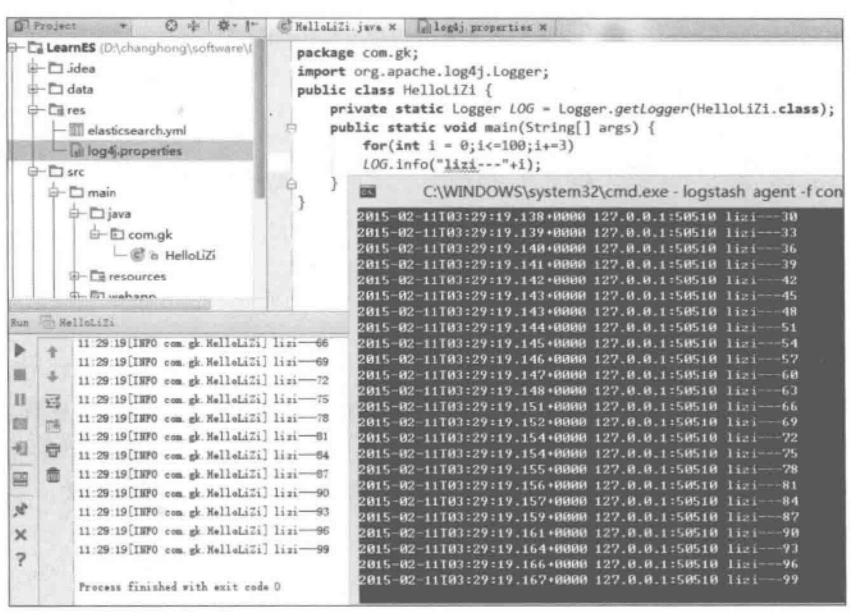


图 7.7 将基于 log4j 产生的输出数据作为 logstash 的输入信息

## 7.2.4 处理基于 Redis 的日志信息

Logstash 也可从 Redis 实例中获取日志。通过在 Logstash 的配置文件中使用 Redis 部分，可以将 Redis 中的日志信息输出到 Logstash 里。

> Tips：Redis 是一个 key-value 存储系统，支持存储的 value 类型包括 String、List、Set、Hash 等。为了保证效率，数据都是缓存在内存中。

代码段 7.6 给出了在 Logstash 端的配置文件情况。应注意如下几点：

- data_type——在配置文件中通过设定它来指定 Logstash 即将处理的 Redis 数据源是 List 形式的还是 Channel 或 Patten_channel 形式的。
- key——由于 Redis 是一个基于 key-value 的存储系统，因此在进行数据处理时，要设定数据的 key。如果 data_type 选择的是 List 形式，这个 key 就是这个 List 的名字；如果 data_type 选择的是 Channel 形式，这个 key 就是这个 Channel 的名字。在这里，Redis 相当于消息的发布方，而 Logstash 则是订阅消息的一方。

```
#代码段 7.6: logstash 形式化配置文件,配置 Redis 输入数据流
input {
  redis {
    add_field=>...        #hash,可选项,默认 {}
    batch_count=>...      #number,可选项,默认 1
    codec=>...            #可选项,默认 plain
    data_type=>...        #string, 可选项,["list", "channel", "pattern_channel"]
                          #其中之一
    db=>...               #number,可选项,默认 0
    host=>...             #可选项,默认 0.0.0.0
    key=>...              #string,可选项
    password=>...         #可选项
    port=>...             #可选项,默认 6379
    tags=>...             #可选项,标记日志,可用于后续处理工作
    threads=>...          #number(可选), 默认 1
    timeout=>...          #number(可选), 默认 5
    type=>...             #可选项,默认 string 类型
    }
}
```

下面给出基于 Redis 的日志数据应用实例。首先安装配置好 Redis(具体步骤不再赘述),启动 Redis 服务器端,在 Redis 配置文件(如 redis. conf)中设置服务器端密码、端口号等(默认端口号为 6379)。启动服务端后,可使用客户端 redis-cli. exe 测试。当确保 Redis 正常启动以后,可在 Redis 端设置日志数据流。

### 1. 测试基于 List 的日志数据流

首先测试基于 List 的日志数据流。图 7.8 给出 Logstash 的配置文件,图左侧的数值表示行号,其中第 6 行指明输入的数据是来源自 Redis 的基于 list 的数据;第 9 行指明 key 值是 msg;第 10 行指明此 Redis 服务器端密码是 123456;其他的采用默认值即可。

```
1  input {
2   redis {
3      # add_field => ... # hash, 可选项, 默认 {}
4      batch_count => 2  # number, 可选项, 默认1
5      #codec => ... # 可选项, 默认是"plain"
6      data_type => "list" # 可选字符串, ["list", "channel", "pattern_channel"]
7      #db => ... # number, 可选项, 默认 0
8      host => "127.0.0.1" #可选项, 默认0.0.0.0
9      key => "msg" # string, 可选项
10     password => "123456" #可选项
11     #port => ... # 可选项, 默认6379
12     #tags => ... # 可选项, 标记日志, 可用于后续处理工作
13     #threads => ... # number (可选), 默认: 1
14     #timeout => ... # number (可选), 默认: 5
15     #type => ... # 可选项, 默认是string类型
16  }
17
18 }
19 output {
20     stdout{}
21 }
22
```

图 7.8 Logstash 的配置

在尚未启动 Logstash 前，可先测试一下 Redis 的运行情况。在图 7.9 左侧图中的第一条命令是启动 Redis 客户端，之后通过 rpush 命令向名为 msg 的 List 中插入了三个 value，分别是 hello、world、logstash，从图 7.9 的左图可以看出，它们分别顺序地插入到了名称为 msg 的 List 中。

> Tips：rpush key value [value ...]是 Redis 的命令，作用是将一个或多个值 value 插入到列表 key 的表尾(最右边)。

之后，启动 Logstash，则在图 7.9 的右侧可以看到 Redis 中的数据(即 msg 中的 hello、world、logstash 字符串)已经通过 Logstash 显示在屏幕上了，此时 msg 这个 List 中的数据流被清空。之后，通过 rpush 命令输入新的字符串 Redis(注：此时 msg 这个 List 中的数据流数量是 1 条)。由于此时 Logstash 已启动，因此这个输入的数据(即 Redis 字符串)会经 Logstash 显示在屏幕上(如图 7.9 右图所示)。此时，Redis 端的名为 msg 的这个 List 中的数据流被清空(可使用“lrange msg 0 -1”命令查看这个 msg 中的 value 情况，如图 7.9 左图倒数第三行所示，结果显示这个 list 为空，如图 7.9 左图倒数第二行所示)，因为数据流已经由 Logstash 这个管道流向了屏幕端(如图 7.9 右图倒数第一行所示)。

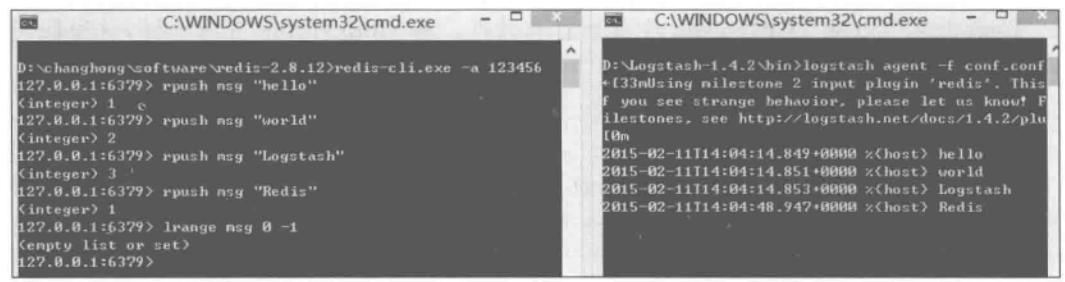


图 7.9　Redis 端的数据情况(左图)及基于 List 方式的 logstash 处理后的结果(右图)

**2. 测试基于 Channel 的日志数据流**

基于消息订阅机制，在 Redis 中也可使用 publish channel message 命令，将信息 message 发送到指定的频道 channel。修改图 7.8 的第 6 行代码，改为输入数据来源自 Redis 的基于 channel 的数据，其他的采用默认值或者类似于图 7.8 中的设置即可。

> Tips：Redis 通过 publish 命令向给定的频道发布信息，返回值为接收到信息的订阅者数量。

之后，启动 Logstash。在 Redis 的客户端通过 publish 命令向名为 msg 的这个 channel 发布信息，如图 7.10 左图所示。此例中，第一次发布的消息是字符串 China，下方显示的数字 1 表示有一个订阅者(注：这个订阅者就是 Logstash，如图 7.10 右侧所示，这个输入到

channel 的信息已经由 Logstash 订阅并显示到屏幕)。同理,可依次在 Redis 端发布"Japan"、"USA"等字符串(即消息,如图 7.10 左图第 3 行和第 5 行所示),它们分别经由 Logstash 这个管道显示在屏幕上,如图 7.10 右图倒数第 2 行和倒数第 1 行所示。

```
C:\WINDOWS\system32\cm...
127.0.0.1:6379> publish msg "China"
(integer) 1
127.0.0.1:6379> publish msg "Japan"
(integer) 1
127.0.0.1:6379> publish msg "USA"
(integer) 1
127.0.0.1:6379>
```

```
C:\WINDOWS\system32\cmd.exe
D:\Logstash-1.4.2\bin>logstash agent -f conf.co
←[33mUsing milestone 2 input plugin 'redis'. Th
f you see strange behavior, please let us know!
ilestones, see http://logstash.net/docs/1.4.2/p
[0m
2015-02-11T14:33:10.010+0000 %{host} China
2015-02-11T14:33:24.691+0000 %{host} Japan
2015-02-11T14:34:13.678+0000 %{host} USA
```

图 7.10 Redis 端的数据情况(左图)及基于 channel 方式的 logstash 处理后的结果(右图)

### 7.2.5 处理基于 Stdin 方式输入的信息

这里处理基于 stdin 输入方式来得到的日志信息。Logstash 可以获取标准输入流,这也是本章入门介绍中使用的方法,如代码段 7.1 所示,此处不再赘述。

### 7.2.6 处理基于 TCP 传输的日志数据

Logstash 可以从 TCP Socket 中获取日志数据。像 stdin 和 file 方法一样,这里假定每一行是一个日志。

> Tips: TCP 是因特网中的传输层协议,使用三次握手协议建立连接。当主动方发出 SYN 连接请求后,等待对方回答 SYN+ACK,并最终对对方的 SYN 执行 ACK 确认。

代码段 7.7 给出了基于 TCP 传输到输入端的 Logstash 的配置文件。

```
#代码段 7.7: logstash 形式化配置文件,处理基于 TCP 传输的日志数据
input {
  tcp {
    add_field=>...                  #hash,可选项,默认{}
    codec=>...                      #可选项,默认 plain
    data_timeout=>...               #number,可选项,默认-1
    host=>...                       #可选项, 默认 0.0.0.0
    mode=>...                       #须是["server", "client"]其中之一,可选项,
                                    #默认是 server
    port=>...                       #端口号,需和通信的另一端匹配
    ssl_cacert=>...                 #一个可用的文件系统路径,可选项
    ssl_cert=>...                   #一个可用的文件系统路径,可选项
    ssl_enable=>...                 #boolean,可选项,默认 false
    ssl_key=>...                    #一个可用的文件系统路径,可选项
    ssl_key_passphrase=>...         #密码,可选项,默认 nil
```

```
    ssl_verify=>... #boolean,可选项,默认 false
    tags=>... #array,可选项
    type=>... #string,可选项
  }
}
```

**1. 将 Logstash 的 Input 部分作为 Client 模式**

首先,设计 Java 应用程序并使之充当 Server 端(如代码段 7.8 所示),它会发送信息。作为 Client 端的 Logstash 则基于 TCP 协议,接收来自这个 Server 端传来的信息并显示之。

```
//代码段 7.8: Java 应用程序,Server 端
package com.gk;
import java.io.*;
import java.net.ServerSocket;
import java.net.Socket;
public class TcpServer {
    public static void main(String[] args) throws IOException {
        ServerSocket serverSocket=new ServerSocket(5656);  //设定端口号
        for(int i=0;i<=3;i++){                          //循环 4 次,结果如图 7.12 下部所示
            Socket socket=serverSocket.accept();                //阻塞等待消息
            outputStream outputStream=socket.getoutputStream();
            outputStream.write(("Welcome,logstash"+i).getBytes());
            outputStream.close();
            socket.close();
        }
        serverSocket.close();
    }
}
```

其次,设计 Logstash 端的配置如图 7.11 所示,注意这里的第 4 行 mode 是设定 Logstash 作为 Client;第 5 行指定的端口号要和代码段 7.8 中 ServerSocket serverSocket=new ServerSocket(5656)语句设置的端口号匹配。

```
input {
 tcp {
    host => "localhost" # string (可选), 默认: "0.0.0.0"
    mode => "client"  #指定Losstash是client模式
    port => 5656 # 要和Server端指定的端口号匹配
 }
}
output {
    stdout{}
}
```

图 7.11 Logstash 配置

最后,依次启动 Java 应用程序(即启动 Server 端,如图 7.12 上图所示)、Logstash(即开启 Client 端,如图 7.12 下图所示)。按照代码段 7.8 中的设计思路,Server 端发送指定的字

符串，可看到由 Server 端发出的信息已经经过 Logstash 这个管道流向了屏幕（即 Client 端，如图 7.12 下图所示）。

```
package com.gk;
import java.io.*;
import java.net.ServerSocket;
import java.net.Socket;
public class TcpServer {
    public static void main(String[] args) throws IOException, Interr
        ServerSocket serverSocket=new ServerSocket(5656);  //设定端口
        for(int i = 0;i<=3;i++){
            Socket socket=serverSocket.accept();//阻塞等待消息
            OutputStream outputStream=socket.getOutputStream();
            outputStream.write(("Welcome,Logstash"+i).getBytes() );
            outputStream.close();
            socket.close();
        }
        Thread.sleep(999999);
        serverSocket.close();
    }
}
```

```
D:\Logstash-1.4.2\bin>logstash agent -f conf.conf
←[33mUsing milestone 2 input plugin 'tcp'. This plugin should be s
you see strange behavior, please let us know! For more information
estones, see http://logstash.net/docs/1.4.2/plugin-milestones {:le
m
2015-02-12T03:19:57.229+0000 127.0.0.1:5656 Welcome,Logstash0
2015-02-12T03:19:57.235+0000 127.0.0.1:5656 Welcome,Logstash1
2015-02-12T03:19:57.239+0000 127.0.0.1:5656 Welcome,Logstash2
2015-02-12T03:19:57.247+0000 127.0.0.1:5656 Welcome,Logstash3
```

图 7.12 Server 端设置（上）及 Client 端输出（下）

**2. 将 Logstash 的 Input 部分作为 Server 模式**

首先，设计 Java 应用程序并使之作为 Client 端（如代码段 7.9 所示。），将 Logstash 的 input 部分作为 Server 模式。

```
//代码段 7.9: Java Client 端设计
package com.gk;
import java.io.*;
import java.net.InetAddress;
import java.net.Socket;
public class TcpClient {
    public static void main(String[] args) {
        DataoutputStream dos=null;
        BufferedReader brNet=null;
        BufferedReader brKey=null;
        Socket s=null;
        try {
            //建立 Socket
            s=new Socket(InetAddress.getByName("localhost"), 5656);
            inputStream ips=s.getinputStream();
            outputStream ops=s.getoutputStream();
```

```
            brKey=new BufferedReader(new inputStreamReader(System.in));
            dos=new DataoutputStream(ops);
            brNet=new BufferedReader(new inputStreamReader(ips));
            while (true) {
              String strWord=brKey.readLine();
              dos.writeBytes(strWord+System.getProperty("line.separator"));
                if (strWord.equalsIgnoreCase("quit"))
                    break;
                else
                    System.out.println(strWord);
            }
        }
//后略
```

其次，设计 Logstash 端的配置信息，如图 7.13 所示，注意这里的第 4 行是设定 Logstash 作为 Server；第 5 行指定的端口号要和代码段 7.9 中的 new Socket(InetAddress.getByName("localhost"), 5656)设置相匹配。

```
1  input {
2   tcp {
3      host => "localhost" # string (可选), 默认: "0.0.0.0"
4      mode => "server"  #指定Losstash是server模式
5      port => 5656 # 要和client端指定的端口号匹配
6  }
7  }
8  output {
9      stdout{}
10 }
```

图 7.13 Logstash 配置

最后，依次启动 Logstash(开启 Server 端)、启动作为 Client 端的 Java 应用程序。在 Client 端输入一些字符(如图 7.14 左侧所示)，它们会经由 Logstash 这个管道输送到屏幕上显示，如图 7.14 右侧所示。可见，在图左侧输入的字符会经 Logstash 在屏幕上显示。

```
Run: TcpClient
"C:\Program ...
hello
hello
gaoshen
gaoshen
```

```
D:\Logstash-1.4.2\bin>logstash agent -f conf.conf
←[33mUsing milestone 2 input plugin 'tcp'. This plugin should b
you see strange behavior, please let us know! For more informat
estones, see http://logstash.net/docs/1.4.2/plugin-milestones <
m
2015-02-12T06:52:45.451+0000 127.0.0.1:50302 hello
2015-02-12T06:52:53.848+0000 127.0.0.1:50302 gaoshen
```

图 7.14 运行结果

### 7.2.7 处理基于 UDP 传输的日志数据

和基于 TCP 的信息传输过程类似，Logstash 也可从 UDP Socket 中获取数据。代码段 7.10给出了在 Logstash 配置文件中配置基于 UDP 传输数据流的方法。图 7.15 给出实际 Logstash 配置文件。

Tips：UDP 是用户数据报协议，它是一个简单的面向数据报的运输层协议。由于 UDP 在传输数据报前不用在客户和服务器之间建立一个连接且没有超时重发等机制，因此传输速度很快。

```
#代码段 7.10: 基于 UDP 的 logstash 形式化配置文件
input {
  udp {
    add_field=>...         #hash(可选), 默认{}
    buffer_size=>...       #number(可选), 默认 8192
    codec=>...             #codec(可选), 默认 plain
    host=>...              #string(可选), 默认 0.0.0.0
    port=>...              #number(必选)
    queue_size=>...        #number(可选), 默认 2000
    tags=>...              #array(可选)
    type=>...              #string(可选)
    workers=>...           #number(可选), 默认 2
  }
}
```

```
input {
 udp {
    host => "localhost" # string (可选), 默认: "0.0.0.0"
    port => 5656 # 要和client端指定的端口号匹配
 }
}
output {
    stdout{}
}
```

图 7.15　基于 UDP 的 logstash 配置文件

为验证运行效果，需要设计一个 Java 应用程序(如代码段 7.11 所示。注意 Java 应用程序中的端口号应和 Logstash 配置文件中的端口号一致)。之后，依次启动 Logstash 和 Java 应用程序 UDPClient(见代码段 7.11)，实际运行效果如图 7.16 所示。从图中可以看出，输入的信息(见图 7.16 的下部)以 UDP 的方式经过 Logstash 这个管道显示在屏幕上(见图 7.16 的上部)。

```
#代码段 7.11: Java 应用程序
package com.gk;
import java.io.*;
import java.net.*;
class UDPClient {
    public static void main(String[] args) throws IOException {
        DatagramSocket client=new DatagramSocket();
```

```
        InetAddress addr=InetAddress.getByName("localhost");
        int port=5656;
        while (true) {
             byte [] send=new BufferedReader(new inputStreamReader(System.
             in)).readLine().getBytes();
            DatagramPacket sendPacket
                   =new DatagramPacket(send, send.length, addr, port);
            client.send(sendPacket);
        }
    }
}
```

```
C:\WINDOWS\system32\cmd.exe
D:\Logstash-1.4.2\bin>logstash agent -f conf.conf
←[33mUsing milestone 2 input plugin 'udp'. This plugin should be stable,
you see strange behavior, please let us know! For more information on pl
estones, see http://logstash.net/docs/1.4.2/plugin-milestones {:level=>:
m
2015-02-12T09:21:20.468+0000 127.0.0.1 hello
2015-02-12T09:22:17.215+0000 127.0.0.1 world
2015-02-12T09:22:23.024+0000 127.0.0.1 this is an example

Run  UDPClient
"C:\Program ...
hello
world
this is an example
```

图 7.16　基于 UDP 的信息输入处理

## 7.3 Codecs: 格式化日志数据

在 Logstash 中,codecs 部分可用于格式化日志数据,本节主要介绍面向 JSON、plain 等的用法。

### 7.3.1 JSON 格式

可利用这个 codecs 机制来解析 JSON 格式的日志数据(如 JSON 流式数据是以"\n"来分隔的,可参考 json_lines 方式)。代码段 7.12 所示是 codecs 可采用的字符集形式。

```
#代码段 7.12: logstach 形式化配置文件,使用 codecs
input {
file {
  codec=>json {
    charset=>... #string, 必须是["ASCII-8BIT", "Big5", "Big5-HKSCS", "Big5-
UAO", "CP949", "Emacs-Mule", "EUC-JP", "EUC-KR", "EUC-TW", "GB18030", "GBK",
"ISO-8859-1", "ISO-8859-2", "ISO-8859-3", "ISO-8859-4", "ISO-8859-5", "ISO-
```

```
8859-6", "ISO-8859-7", "ISO-8859-8", "ISO-8859-9", "ISO-8859-10", "ISO-8859
-11", "ISO-8859-13", "ISO-8859-14", "ISO-8859-15", "ISO-8859-16", "KOI8-
R","KOI8-U", "Shift_JIS", "US-ASCII", "UTF-8", "UTF-16BE", "UTF-16LE", "UTF-
32BE", "UTF-32LE", "Windows-1251", "GB2312", "IBM437", "IBM737", "IBM775",
"CP850", "IBM852", "CP852", "IBM855", "CP855", "IBM857", "IBM860", "IBM861",
"IBM862", "IBM863", "IBM864", "IBM865", "IBM866", "IBM869", "Windows-1258",
"GB1988", " macCentEuro ", " macCroatian ", " macCyrillic ", " macGreek ",
"macIceland", " macRoman ", " macRomania ", " macThai ", " macTurkish ",
"macUkraine", "CP950", "CP951", "stateless-ISO-2022-JP", "eucJP-ms",
"CP51932", "GB12345", "ISO-2022-JP", "ISO-2022-JP-2", "CP50220", "CP50221",
"Windows-1252", "Windows-1250", "Windows-1256", "Windows-1253", "Windows-
1255", "Windows-1254", "TIS-620", "Windows-874", "Windows-1257", "Windows-
31J", "MacJapanese", "UTF-7", "UTF8-MAC", "UTF-16", "UTF-32", "UTF8-DoCoMo",
"SJIS-DoCoMo", "UTF8-KDDI", "SJIS-KDDI", "ISO-2022-JP-KDDI", "stateless-
ISO-2022-JP-KDDI", "UTF8-SoftBank", "SJIS-SoftBank", "BINARY", "CP437",
"CP737", "CP775", "IBM850", "CP857", "CP860", "CP861", "CP862", "CP863",
"CP864", "CP865", "CP866", "CP869", "CP1258", "Big5-HKSCS:2008", "eucJP", "euc
-jp-ms", "eucKR", "eucTW", "EUC-CN", "eucCN", "CP936", "ISO2022-JP", "ISO2022
-JP2", "ISO8859-1", "CP1252", "ISO8859-2", "CP1250", "ISO8859-3", "ISO8859-
4", "ISO8859-5", "ISO8859-6", "CP1256", "ISO8859-7", "CP1253", "ISO8859-8",
"CP1255", "ISO8859-9", "CP1254", "ISO8859-10", "ISO8859-11", "CP874",
"ISO8859-13", "CP1257", "ISO8859-14", "ISO8859-15", "ISO8859-16", "CP878",
"CP932", "csWindows31J", "SJIS", "PCK", "MacJapan", "ASCII", "ANSI_X3.4-
1968", "646", "CP65000", "CP65001", "UTF-8-MAC", "UTF-8-HFS", "UCS-2BE", "UCS
-4BE", "UCS-4LE", "CP1251", "external", "locale"]其中之一（可选），默认："UTF-8"
        }
    }
}
```

下面给出 codecs 和 output 一起使用的例子。图 7.17 的 Logstash 配置文件给出在 output 中指定的数据格式为 JSON。之后，运行 Logstash，则用户输入的字符串就会以 JSON 格式显示出来，如图 7.18 所示。图中所示为分别输入了"helloworld"字符串和"this is an example"字符串后的情况，从图中反馈的信息可以看出，输入的字符串都被解析为 JSON 格式的数据了。

```
input {
 stdin {

 }
}
output {
    stdout{codec => json}
}
```

图 7.17 Logstash 配置文件

```
D:\Logstash-1.4.2\bin>logstash agent -f conf.conf
helloworld
{"message":"helloworld\r","@version":"1","@timestamp":"2015-02-12T09:56:37.144Z"
,"host":"hebust"}this is an example
{"message":"this is an example\r","@version":"1","@timestamp":"2015-02-12T09:56:
48.360Z","host":"hebust"}
```

图 7.18 实际运行效果

如果采用 json_lines，则可解析按行分隔的 JSON 格式的日志数据。代码段 7.13 是 Logstash 配置文件，它给出了在 UDP 输入模式下的 json_lines 的用法。

```
#代码段 7.13: 在 UDP 输入模式中采用 codecs 的配置文件
input {
  udp {
    port=>1234
    codec=>json_lines {
      charset=>... #string (可选), 默认: "UTF-8"
    }
  }
}
```

## 7.3.2 Rubydebug 格式

这个解析器会使用 Ruby Awesome Print 库来解析日志格式。图 7.19 给出 Logstash 的配置信息，注意这里设置的输入信息是 JSON 格式的（因此从键盘输入信息时应采用 JSON 格式）；而数据输出格式则采用 rubydebug 方式。图 7.20 是实际运行效果，注意图中第 2 行是输入的 JSON 格式的数据，而从第 3 行开始可以看到输出了带有相应格式的输出信息。

```
1 input {
2  stdin {codec => json
3
4 }
5 }
6 output {
7     stdout{codec => rubydebug}
8 }
9
```

图 7.19 Logstash 配置

```
D:\Logstash-1.4.2\bin>logstash agent -f conf.conf
{"name":"gaokai","age": 17}
{
        "name" => "gaokai",
         "age" => 17,
    "@version" => "1",
  "@timestamp" => "2015-02-12T10:02:45.708Z",
        "host" => "hebust"
}
```

图 7.20 实际效果

### 7.3.3 Plain 格式

这是一个空的解析器，解析格式可以由使用者自行指定。代码段 7.14 给出当 Logstash 输入源是文件的情况下，通过 codec 进行格式转换的方法。图 7.21 给出采用代码段 7.14 中的配置文件运行 Logstash 的实际效果。从图中可见，输出字符串的部分不带任何格式。

```
#代码段 7.14: logstash 配置文件，输出采用 plain
input {
  stdin{ }
}
output {
    stdout{ codec=>plain }
}
```

```
D:\Logstash-1.4.2\bin>logstash agent -f conf1.conf
hello
2015-02-12T14:10:35.976+0000 hebust hello
```

图 7.21　基于 plain 的实际效果

## 7.4 基于 Filter 的日志处理与转换

在基于 Logstash 的日志处理中，往往要处理多种不同的日志信息，如 Apache 服务器日志、Postfix 日志、Java 应用程序或和某种特定应用相关的日志信息等[Turnbull,2015]。

> Tips：Postfix 是 Wietse Venema 在 IBM 的 GPL 协议之下开发的一种邮件传输代理软件。

截至目前，已经介绍了如何让 Logstash 直接处理各种来源的日志数据，但尚未利用、抽取、过滤这些数据中可能蕴含的一些元数据信息。例如，对于如下的 Apache 服务器日志信息来说，就包含了诸如 IP 地址、时间戳、HTTP 方法、HTTP 响应模式等诸多信息。

```
186.4.131.228--[20/Dec/2012:20:34:08 -0500] "GET /2012/12/new-product/ HTTP/
1.0"200 10902
"http://www.example.com/20012/12/new-product/"
"Mozilla/5.0 (Windows; U; Windows NT 5.1; pl; rv:1.9.1.3) Gecko/20090824
Firefox/3.5.3"
```

对上述日志来说，如果不做任何解析处理，而是一股脑地将这些信息统统塞给

Logstash,那么 Logstash 也会无所适从,因为它不清楚其中的各个部分所代表的含义。其实,Logstash 提供的 filter 机制可以方便地按照用户要求解析日志信息。如果需要抽取、分析、计算这些可能的元信息,可以利用 Logstash 中的 filter 机制完成相应的功能。限于篇幅,本章只对 filter 中的 json filter、grok filter、kv filter 等进行简介。

### 7.4.1 JSON Filter

如果日志数据源是 json 格式的,则利用 filter,可将其扩展成一个编程人员所需要的数据结构。代码段 7.15 给出在 filter 中使用 json 格式的方法。

```
#代码段 7.15: logstash 形式化配置文件,解析并处理 json 格式的数据为指定的数据结构
filter {
  json {
    add_field=>...          #hash(可选项), 默认{}
    add_tag=>...            #array(可选项), 默认[ ]
    remove_field=>...       #array(可选项), 默认[ ]
    remove_tag=>...         #array(可选项), 默认[ ]
    source=>...             #string(必选项)
    target=>...             #string(可选项)
  }
}
```

这里对代码段 7.15 中的部分内容解释如下:

- add_field——值类型为 hash,默认为空。如设置了这个部分,会增加指定的字段到这个事件,例如,“add_field=> [ "foo_%{somefield}", "Hello world, from %{host}" ]”,也就是说,如果这个事件有一个字段 somefiled 且值是 hello,那么会增加一个字段 foo_hello,字段值则用%{host}代替。
- add_tag——值类型为数组,默认为空。若执行成功,会增加一个 tags,例如,“add_tag=> [ "foo_%{somefield}" ]”。
- remove_field——值类型为数组,默认是空,若执行成功则删除一个 field,如“remove_tag=> [ "foo_%{somefield}" ]”。
- source——值类型为字符串,默认没有设置。

代码段 7.16 给出 Logstash 的配置信息。在 input、output 部分中间的 filter 部分嵌入了对 json 数据的解析方法,这里的 filter 里面的 source=>是指要让这个 json filter 去解析哪个字段里面的 json 数据,例如,source=>"message"是让 json filter 去解析 message 这个字段里面的 json 数据,换句话说,就是欲处理的 json 数据放在哪个字段里面。图 7.22 给出了实际的运行结果。当用户从键盘输入 json 格式的数据后,Logstash 的 filter 会对输入信息进行加工处理,如图 7.22 所示,在此例中,用户输入的字符串是{"name": "lily", "age": 13},经处理后返回的结果显示如图 7.22 所示。

```
#代码段 7.16: 基于 json filter 的 logstash 配置文件
input {
  stdin{ }
}
filter {
    json {
      source=>"message"
  }
}
output {
  stdout { codec=>rubydebug }
}
```

```
{"name":"lily","age":13}
{
       "message" => "{\"name\":\"lily\",\"age\":13}",
      "@version" => "1",
    "@timestamp" => "2015-02-12T13:38:03.993Z",
          "host" => "localhost",
          "name" => "lily",
           "age" => 13
}
```

图 7.22　对 json 格式数据的 filter 处理结果

## 7.4.2　Grok Filter

Grok 是一个数据结构化工具。利用它,只需要通过简单定义,就可将文本格式的字符串转换成为具体的结构化的数据。其实,Logstash 默认带有上百个 grok 变量,可以直接使用或者稍微改写成需要的新的 grok。Grok filter 适合对 syslog、apache log 等可读日志进行解析。代码段 7.17 给出基于 grok filter 的 Logstash 配置信息,其中的 add_field 等的含义同上,此处不再赘述。

```
#代码段 7.17: 基于 grok filter 的 logstash 形式化配置文件
filter {
  grok {
    add_field=>...                #hash(可选项),默认{}
    add_tag=>...                  #array(可选项),默认[]
    break_on_match=>...           #boolean(可选项),默认 true
    drop_if_match=>...            #boolean(可选项),默认 false
    keep_empty_captures=>...      #boolean(可选项),默认 false
    match=>...                    #hash(可选项),默认{}
    named_captures_only=>...      #boolean(可选项),默认 true
    overwrite=>...                #array(可选项),默认[]
```

```
    patterns_dir=>...              #array(可选项),默认[ ]
    remove_field=>...              #array(可选项),默认[ ]
    remove_tag=>...                #array(可选项),默认[ ]
    tag_on_failure=>...            #array(可选项),默认["_grokparsefailure"]
  }
}
```

作为实例,代码段 7.18 给出一个基于 grok filter 的 Logstash 配置实例,对于如下代码:

```
match=>[ "message", "%{IP:client} %{WORD:method} %{URIPATHPARAM:request} %{NUMBER:bytes} %{NUMBER:duration}" ]
```

其中,第一个参数是指定要匹配哪个字段(此例是指明要匹配 message 字段);第二个参数是要匹配一组以空格(由%表示)分开的字符串并将它们分别指定一个名字,此例包括如下几项:

- 对数据类型 IP 指定其变量名为 client。
- 对数据类型 WORD 指定其变量名为 method。
- 对 URI 路径参数 URIPATHPARAM 指定其变量名为 request。
- 对数据类型 NUMBER 指定其变量名为 bytes。
- 对数据类型 NUMBER 指定其变量名为 duration。

在此基础上,Logstash 可将用户输入的内容进行解析。

```
#代码段 7.18: 基于 grok filter 的 logstash 配置文件
input {
  stdin{ }
}
filter {
    grok {
      match=>[ "message", "%{IP:client} %{WORD:method} %{URIPATHPARAM:
      request} %{NUMBER:bytes} %{NUMBER:duration}" ]
    }
  }
output {
  stdout { codec=>rubydebug }
  }
```

例如,当处理如下日志信息"55.3.244.1 GET /index.html 15824 0.043"时,按照代码段 7.18 中的 Logstash 配置文件 conf.conf 进行解析,会得到如图 7.23 所示的结果(注:图中第一行是用命令方式启动 Logstash,第二行是待处理的字符串,后面大括号中的内容是解析后的结果)。

```
localhost:bin gaoshen$ ./logstash -f conf.conf
55.3.244.1 GET /index.html 15824 0.043
{
       "message" => "55.3.244.1 GET /index.html 15824 0.043",
      "@version" => "1",
    "@timestamp" => "2015-02-12T13:31:40.563Z",
          "host" => "localhost",
        "client" => "55.3.244.1",
        "method" => "GET",
       "request" => "/index.html",
         "bytes" => "15824",
      "duration" => "0.043"
}
```

图 7.23　基于 grok filter 的处理结果

## 7.4.3　Kv Filter

kv filter 用于对诸如 key-value 这种键值对数据进行解析。代码段 7.19 给出基于 kv filter 的 Logstash 配置，field_split 这个参数是用来设置分隔符的，代码"kv {field_split=> "&?"}"的含义是说明字符串分隔的标记是 & 或者?。例如，从图 7.24 中用户输入的日志字符串"http://www.baidu.com/s? wd=%E4%B8%AD%E5%9B%BD&rsv_spt=1&issp=1&f=8&rsv_bp=0&rsv_idx=2&ie=utf-8&tn=SE_hldp01550_7zn76813&rsv_enter=1&rsv_sug3=2&rsv_sug1=1&rsv_sug2=0&inputT=2950&rsv_sug4=2951&rsv_sug=2"可清楚地看到，经过 Logstash 的处理后，已经得到以"&"或"?"分隔的子字串。对应地，图 7.24 给出了基于该配置信息的 Logstash 处理结果。

```
#代码段 7.19: 基于 kv filter 的 logstash 配置文件
input { stdin{ }  }
filter {  kv {   field_split=>"&?"  }     }
output {  stdout { codec=>rubydebug } }
```

```
https://www.baidu.com/s?wd=%E4%B8%AD%E5%9B%BD&rsv_spt=1&issp=1&f=8&rsv_bp=0&rsv_idx=2&ie=utf
&inputT=2580&rsv_sug4=3239
{
       "message" => [illegible]
rsv_sug1=3&rsv_sug2=0&inputT=2580&rsv_sug4=3239",
      "@version" => "1",
    "@timestamp" => [illegible],
          "host" => "localhost",
            "wd" => "%E4%B8%AD%E5%9B%BD",
       "rsv_spt" => "1",
          "issp" => "1",
             "f" => "8",
        "rsv_bp" => "0",
       "rsv_idx" => "2",
            "ie" => "utf-8",
            "tn" => "baiduhome_pg",
     "rsv_enter" => "1",
      "rsv_sug3" => "4",
      "rsv_sug1" => [illegible],
      "rsv_sug2" => "0",
        "inputT" => "2580",
      "rsv_sug4" => "3239"
}
```

图 7.24　基于 kv filter 的处理结果

从图 7.24 中还可以看出，解析出的查询字符串"wd"是在输入的 URL 字串中表现的模式(在图中显示为“%E4%B8%AD%E5%9B%BD”)，而并非用户在键盘输入的可以识别的中文字符串(注：这里应该是用户输入的“中国”二字)。

为解决上述问题，只需要在配置文件中添加 urldecode 部分即可，详见代码段 7.20 中的配置文件，基于该配置代码的 Logstash 处理结果如图 7.25 所示。比较图 7.24 和图 7.25 可清楚地看到，图 7.25 已经顺利解析出 wd 内容了(显示为“中国”二字)。

```
#代码段 7.20: 基于 kv filter 和 urldecode 的 logstash 配置文件
input {
  stdin{ }
}
filter {
  kv {   field_split=>"&?"  }          #说明字符串分隔的标记是 & 或者?
  urldecode {field=>wd }               #说明解码字段是 wd
  }
output {  stdout { codec=>rubydebug } }
```

```
{
       "message" => "[illegible]",
      "@version" => "1",
    "@timestamp" => "[illegible]",
          "host" => "localhost",
            "wd" => "中国",
       "rsv_spt" => "1",
          "issp" => "1",
             "f" => "8",
        "rsv_bp" => "0",
       "rsv_idx" => "2",
            "ie" => "utf-8",
            "tn" => "baiduhome_pg",
     "rsv_enter" => "1",
      "rsv_sug3" => "4",
      "rsv_sug1" => "3",
      "rsv_sug2" => "0",
        "inputT" => "2580",
      "rsv_sug4" => "3239"
}
```

图 7.25 基于 kv filter 和 urldecode 的处理结果

> Tips: urldecode 是对字符串进行解码，其返回值是已解码的字符串。其编码规则一般是这样的：数字和字母不变，空格变为“+”号，其他字符被编码，比如“中国”二字的编码形式为“%E4%B8%AD%E5%9B%BD”。

## 7.5 Output: 处理输出的日志数据

截至目前，已经对 Logstash 中的 inputs、codecs、filters 等进行了简述。在 Logstash 配置文件的 output 部分，可以使用 stdout 来把经由 Logstash 处理的日志传送到显示器上输

出。当然，在设置输出时，也可以在 stdout 中同时设置 codec，如代码段 7.20 所示。其实，当 Logstash 处理完日志数据后，不仅可以在显示器上显示，也可需要将它们输出到 Elasticsearch、Email、Redis、文件等处。本节介绍输出日志数据的方法。

### 7.5.1 将处理后的日志输出到 Elasticsearch 中

通过设置 Logstash 的配置文件中的 elasticsearch_http 部分（见代码段 7.21），可使用 Elasticsearch 作为处理日志的接收端。这种方式可以将收集到的日志通过 HTTP 接口存放到 Elasticsearch 中。代码段 7.21 给出 Logstash 配置文件中的 output 输出部分的设置。

```
#代码段 7.21: logstash 形式化配置文件中的 output 输出部分
output {
  elasticsearch_http {
    codec=>...                    #codec(可选), 默认 plain
    document_id=>...              #string(可选), 默认 nil
    flush_size=>...               #number(可选), 默认 100
    host=>...                     #string(必选) Elasticsearch 服务器的 host
    idle_flush_time=>...          #number(可选), 默认: 1
    index=>...                    #string(可选), 默认 "logstash-%{+YYYY.MM.dd}"
    index_type=>...               #string(可选)
    manage_template=>...          #boolean(可选), 默认 true
    password=>...                 #password(可选), 默认 nil
    port=>...                     #number(可选), 默认 9200
    replication=>...              #string(可选), ["async", "sync"] 默认 sync
    template=>...                 #a valid filesystem path(可选)
    template_name=>...            #string(可选), 默认 logstash
    template_overwrite=>...       #boolean(可选), 默认 false
    user=>...                     #string(可选), 默认 nil
    workers=>...                  #number(可选), 默认 1
  }
}
```

代码段 7.22 给出一个完整的 Logstash 配置文件实例，实际运行效果如图 7.26 所示。

```
#代码段 7.22: logstash 配置文件，用 Elasticsearch 作为日志接收端
input {
    stdin{}
}
output {
    elasticsearch_http {          #通过 HTTP 的方式将数据传输到 Elasticsearch 中
    host=>localhost
    }
    stdout{}
}
```

| Searched 5 of 5 shards. 2 hits. 0.002 seconds | | | | | | | |
|---|---|---|---|---|---|---|---|
| _index | _type | _id | _score ▼ | message | @version | @timestamp | host |
| logstash-2015.02.02 | logs | AUtHqegaadm4TrPXkSXL | 1 | lizi | 1 | 2015-02-02T00:21:31.935Z | hebust |
| logstash-2015.02.02 | logs | AUtHupm4adm4TrPXkSqW | 1 | goodluck | 1 | 2015-02-02T00:39:46.351Z | hebust |

图 7.26 基于 elasticsearch_http 的输出结果

之后，依次启动 Elasticsearch、启动 Logstash，可在控制台输入部分字符。根据代码段 7.22 中的设置，输入的信息既可显示在屏幕上，同时也存入了 Elasticsearch 中，如图 7.26 所示是将数据发往 Elasticsearch 索引文件后的结果。

## 7.5.2 将处理后的日志输出至文件中

Logstash 也可以将收集到的日志(经处理后)输出到一个指定的文件中(可以用域中的一些值作为文件名或者路径名称)。在代码段 7.23 的 Logstash 配置文件中指定用文件作为日志接收端。

```
#代码段 7.23: logstash 形式化配置文件，用 file 作为日志接收端
output {
  file {
    codec=>...                    #codec(可选), 默认 plain
    flush_interval=>...           #number(可选), 默认 2
    gzip=>...                     #boolean(可选), 默认 false
    max_size=>...                 #string(可选)
    message_format=>...           #string(可选)
    path=>...                     #string(必选)待存放文件的路径
    workers=>...                  #number(可选), 默认 1
  }
}
```

作为实例，代码段 7.24 给出一个完整的 Logstash 配置文件。其中，field_split 这个参数是用来设置分隔符的(注：由于这里给出的日志字符串是用户输入的 URL，因此这里是指定 & 或？作为分隔符)；urldecode 是对字符串进行 URL 解码，其含义不再赘述。给出的 URL 字符串经过 Logstash 处理后存入到指定的文件中，如图 7.27 所示。

```
#代码段 7.24: 用指定路径的文件来存放处理以后的日志信息
input {
  stdin { }
}
filter {
  kv {
    field_split=>"&?"
  }
  urldecode{
```

```
      field=>wd
    }
  }
  output {
    file {
      path=>"/log.txt"
      codec=>rubydebug
    }
    stdout {
      codec=>plain
    }
  }
```

log.txt - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)


```
{"message":"http://www.baidu.com/s?wd=%E4%B8%AD%E5%9B
%BD&rsv_spt=1&issp=1&f=8&rsv_bp=0&rsv_idx=2&ie=utf-
8&tn=SE_hldp01550_7zn76813&rsv_enter=1&rsv_sug3=2&rsv_sug1=1&rsv_sug2=0&i
nputT=2950&rsv_sug4=2951&rsv_sug=2\r","@version":"1","@timestamp":"2015-
02-14T00:50:34.356Z","host":"hebust","wd":"中
国","rsv_spt":"1","issp":"1","f":"8","rsv_bp":"0","rsv_idx":"2","ie":"utf
-
8","tn":"SE_hldp01550_7zn76813","rsv_enter":"1","rsv_sug3":"2","rsv_sug1"
:"1","rsv_sug2":"0","inputT":"2950","rsv_sug4":"2951","rsv_sug":"2\r"}
```


图 7.27 处理后的日志信息存入文件中

### 7.5.3 将处理后的部分日志输出到 csv 格式的文件中

通过对启动配置文件的设置,也可以将收集到的日志以 csv 的格式输出到指定的文件中。代码段 7.25 给出了 Logstash 配置文件中有关 csv 输出的部分。

```
#代码段 7.25: logstash 形式化配置文件,用 csv 作为日志接收端
output {
  csv {
    codec=>...                  #codec(可选), 默认 plain
    csv_options=>...            #hash(可选), 默认 {}
    fields=>...                 #array(必选)指定 csv 中各个域的名称
    flush_interval=>...         #number(可选), 默认 2
    gzip=>...                   #boolean(可选), 默认 false
    max_size=>...               #string(可选)
    message_format=>...         #string(可选)
    path=>...                   #string(必选)指定输出的 csv 文件路径
    workers=>...                #number(可选), 默认 1
  }
}
```

代码段 7.26 给出了一个完整的 Logstash 配置文件，可以看出，在对输入的日志字符串进行 kv filter、urldecode 处理后，将解析的部分字段结果(如@timestamp、host、wd 等部分)存入指定的文件中，结果如图 7.28 所示。

```
#代码段 7.26: logstash 配置文件,用 csv 作为日志接收端
input {
  stdin { }
}
filter {
  kv {
    field_split=>"&?"
  }
  urldecode{
    field=>wd
  }
}
output {
  csv{
    path=>"/log.txt"
    fields=>["@ timestamp","host","wd"]
  }
  stdout {
    codec=>plain
  }
}
```

log.txt - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
2015-02-14 01:14:02 UTC, hebust, 中国

图 7.28 处理后的部分日志信息存入文件中

### 7.5.4 将处理后的日志输出到 redis 中

和输入 redis 中的日志信息类似，Logstash 也可以从 redis 实例中获取日志信息。通过在 Logstash 的配置文件的 output 部分使用 redis，可以将处理之后的日志信息输出到 redis 中。代码段 7.27 给出 Logstash 配置文件中有关 output 中 redis 部分的实现方法。

```
#代码段 7.27: logstash 形式化配置文件,有关 output 中 redis 的部分
output {
  redis {
    batch=>...                      #boolean(可选), 默认 false
    batch_events=>...               #number(可选), 默认 50
```

```
    batch_timeout=>...               #number(可选), 默认 5
    codec=>...                       #codec(可选), 默认 plain
    congestion_interval=>...         #number(可选), 默认 1
    congestion_threshold=>...        #number(可选), 默认 0
    data_type=>...                   #string, ["list", "channel"]其中之一(可选)
    db=>...                          #number(可选), 默认 0
    host=>...                        #array(可选), 默认 ["127.0.0.1"]
    key=>...                         #string(可选)
    password=>...                    #password(可选)
    port=>...                        #number(可选), 默认 6379
    reconnect_interval=>...          #number(可选), 默认 1
    shuffle_hosts=>...               #boolean(可选), 默认 true
    timeout=>...                     #number(可选), 默认 5
    workers=>...                     #number(可选), 默认 1
  }
}
```

代码段 7.28 给出了 Logstash 配置文件中 data_type 的基于 list 的实现方法。

```
#代码段 7.28: logstash 配置文件
input {
  stdin { }
}
output {
  redis {
    data_type=>"list"              #可选参数有 list 和 channel 等
    host=>"127.0.0.1"
    key=>"example1"                #给定的列表 key 的名称
    password=>123456               #对应于 redis 的密码
  }
}
```

相应的步骤是：依次启动 Redis 的 server 端和 client 端；基于代码段 7.28 的配置文件启动 Logstash；在 Logstash 控制台输入测试字符串；切换到 Redis 的 client 端，输入 lrange example1 0 -1(注意，这里的示例字符串"example1"要和代码段 7.28 中的 key 值匹配)。

> Tips: Redis 中的语句 lrange key start stop 的作用是返回列表 key 中在指定区间内的元素，区间以偏移量 start 和 stop 指定，参数 start 和 stop 都是从 0 开始，0 表示列表的第 1 个元素，1 表示列表的第 2 个元素，-1 表示列表的最后元素，-2 表示列表的倒数第二个元素，以此类推。

代码段 7.28 给出 Logstash 配置文件中基于 List 的完整例子。和在 input 中的情况类

似，如将这里的 data_type=> "list"改换成 data_type=> "channel"，则输出信息会以 channel 的方式注入到 Redis 中，图 7.29 给出基于 channel 的 Redis 输出结果。请注意在测试时，要用 subscribe 命令，且其后的 channel 名称应和在 Logstash 中定义的相匹配（例子中使用的 channel 示例名称为 example2，如图 7.29 左图所示）。

> Tips：Redis 中的语句 subscribe channel [channel ...]是用于订阅给定的一个或多个频道的信息。例如输入 subscribe msg chat_room，可能会返回如下信息：
>
> 1) "subscribe"　　#返回值的类型，表示订阅成功
>
> 2) "msg"　　#订阅的频道名字
>
> 3) (integer) 1　　#目前已订阅的频道数量

```
D:\changhong\software\redis-2.8.12>redis-cli.exe -a 123456
127.0.0.1:6379> subscribe example2
Reading messages... (press Ctrl-C to quit)
1) "subscribe"
2) "example2"
3) (integer) 1
1) "message"
2) "example2"
3) "{\"message\":\"hello redis\\r\",\"@version\":\"1\",\"@timestamp\":
14T03:01:30.853Z\",\"host\":\"hebust\"}"
1) "message"
2) "example2"
3) "{\"message\":\"ergeg\\r\",\"@version\":\"1\",\"@timestamp\":\"201
01:56.055Z\",\"host\":\"hebust\"}"
```

```
D:\Logstash-1.4.2\bin>log
←[33mUsing milestone 2 ou
if you see strange behavi
milestones, see http://lo
←[0m
hello redis
ergeg
```

图 7.29　日志以 channel 形式输出到 Redis 中（左图为 Redis 输入端，右图为 Logstash 输出端）

### 7.5.5　将处理后的部分日志通过 UDP 协议输出

在 Logstash 配置文件的 output 部分可以使用 UDP 的方式，将日志通过网络发送到另外一台主机上。代码段 7.29 给出了部分基于 UDP 方式输出的 Logstash 配置文件。

```
#代码段 7.29: logstash 形式化配置文件
output {
  udp {
    codec=>...            #codec(可选), 默认 plain
    host=>...             #string(必选)
    port=>...             #number(必选)
    workers=>...          #number(可选), 默认 1
  }
}
```

代码段 7.30 给出了完整的 Logstash 配置文件。

```
#代码段 7.30: logstash 配置文件
input {
  stdin { }
```

```
}
output {
  udp {
    host=>"127.0.0.1"
    port=>5656
  }
}
```

为测试效果,代码段 7.31 给出了 UDPClient(即传输数据的接收端)的 Java 实现。

```
#代码段 7.31: UDP client
package com.gk;
import java.io.*;
import java.net.*;
class UDPClient {
    public static void main(String[] args) throws IOException {
        InetAddress addr=InetAddress.getByName("localhost");
        int port=5656;          //这个端口号要和代码段 7.30 配置文件中的端口号设置一样
        DatagramSocket client=new DatagramSocket(port,addr);
        while(true) {
            byte[] recvBuf=new byte[100];
            DatagramPacket recvPacket=new DatagramPacket(recvBuf , recvBuf.
            length);
            client.receive(recvPacket);
            String recvStr=new String(recvPacket.getData() , 0 ,recvPacket.
            getLength());
            System.out.println("收到:"+recvStr);
        }
    }
}
```

实际运行效果如图 7.30 所示。在图 7.30 中的上半部分中是输入的信息,经过 Logstash 这个基于 UDP 的输出“管道”后,如图 7.30 的下半部分所示,内容含输入的字符串、版本号、时间戳、host 名称等。

```
D:\Logstash-1.4.2\bin>logstash agent -f conf.conf
←[33mUsing milestone 1 output plugin 'udp'. This plugin should work, but would b
enefit from use by folks like you. Please let us know if you find bugs or have s
uggestions on how to improve this plugin.  For more information on plugin milest
ones, see http://logstash.net/docs/1.4.2/plugin-milestones {:level=>:warn}←[0m
hello udp
```

```
Client
"C:\Program ...
收到:{"message":"hello udp\r","@version":"1","@timestamp":"2015-02-14T03:32:24.884Z","host":"hebust"}
```

图 7.30　日志经 udp 方式输出

### 7.5.6 将处理后的部分日志通过 TCP 协议输出

Logstash 可以将处理后的日志从 TCP Socket 中输出。代码段 7.32 给出了 Logstash 的 output 中有关 TCP 的设置。

```
#代码段 7.32: logstash 形式化配置文件,有关 output 设置
output {
  tcp {
    codec=>...                          #codec(可选), 默认 plain
    host=>...                           #string(必选)
    mode=>...                           #string, ["server", "client"]其中之一(可选),
                                        #默认 client
    port=>...                           #number(必选)
    reconnect_interval=>...             #number(可选), 默认 10
    workers=>...                        #number(可选), 默认 1
  }
}
```

将处理后的部分日志通过 TCP 协议输出,又分为两种情况。

**1. 将 Logstash 的 Output 作为 Client 端**

首先,设计 Java 应用程序,并使之作为 Server 端;作为 Client 端的 Logstash 则基于 TCP 协议发送处理之后的日志信息到 Server 端。TCP Server 端的代码如代码段 7.33 所示。

```
#代码段 7.33: TCPServer 端的设计
package com.gk;
import java.io.*;
import java.net.ServerSocket;
import java.net.Socket;
public class TCPServer_output {
    public static void main(String[] args) throws IOException {
        ServerSocket serverSocket=new ServerSocket(5656);
        Socket socket=serverSocket.accept();
        InputStream inputStream=socket.getInputStream();
        while(true) {
            byte[] buf=new byte[1024];
            int len=inputStream.read(buf);
            System.out.println(new String(buf, 0, len));
        }
    }
}
```

其次,设计 Logstash 端的配置文件如代码段 7.34 所示,注意,这里是设定 Logstash 作

为 client；端口号要和代码段 7.33 中的 ServerSocket serverSocket＝new ServerSocket(5656)设置匹配。

```
#代码段 7.34: logstash 配置文件
input {
  stdin{}
}
output {
  stdout { codec=>rubydebug }
  tcp{
    host=>"localhost"
    port=>5656
    mode=>"client"
  }
}
```

最后，依次启动 Java 应用程序（启动 Server 端）、Logstash（开启 Client 端）。按照代码段 7.33 中的设计思路，Server 端接收在 Logstash 端得到的日志信息。在图 7.31 上部的控制台是 Logstash 端(client 端)，这里用输入的字符“123456”模拟为传输到 Logstash 的日志信息，它们会被传递到 server 端显示（见图 7.31 的下半部分）。

```
123456
{
       "message" => "123456\r",
      "@version" => "1",
    "@timestamp" => "2015-02-14T07:08:26.842Z",
          "host" => "hebust"
}
```

```
"C:\Program ...
{"message":"123456\r","@version":"1","@timestamp":"2015-02-14T07:08:26.842Z","host":"hebust"}
```

图 7.31　日志经 TCP 方式输出（logstash 作为 client 端）

**2. 将 Logstash 的 Output 作为 Server 端**

为验证 Logstash 的 output 作为 server 端的运行效果，这里设计 Java 应用程序并使之作为 Client 端，如代码段 7.35 所示。

```
#代码段 7.35: TCPClient 端的设计
package com.gk;
import java.io.*;
import java.net.InetAddress;
import java.net.Socket;
public class TCPClient_output {
    public static void main(String[] args) {
        DataOutputStream dos=null;
        BufferedReader brNet=null;
```

```
        BufferedReader brKey=null;
        Socket s=null;
        try {
            //建立 Socket
            s=new Socket(InetAddress.getByName("localhost"), 5656);
            InputStream ips=s.getInputStream();
            while(true) {
                byte[] buf=new byte[1024];
                int len=ips.read(buf);
                System.out.println(new String(buf, 0, len));
            }
//后略
```

Logstash 端 output 的配置如代码段 7.36 所示，注意这里设定 Logstash 作为 server；port 端口号要和代码段 7.35 中的设置 Socket(InetAddress. getByName("localhost"), 5656)匹配。

```
#代码段 7.36: logstash 的 output 端的配置文件
input {
  stdin{}
}
output {
  stdout { codec=>rubydebug }
  tcp{
    host=>"localhost"
    port=>5656
    mode=>"server"
  }
}
```

依次启动 Logstash(开启 Server 端)、启动 client 端的 Java 应用程序。在控制台输入一些字符，用它们来模拟得到的日志信息，它们会经由 Logstash 这个管道以 TCP 协议方式输送并显示，如图 7.32 所示。在图 7.32 上半部分的控制台 Logstash 端输入的字符“hello”(Server 端)，会传递到图 7.32 下半部分显示(Client 端)。

```
hello
{
       "message" => "hello\r",
      "@version" => "1",
    "@timestamp" => "2015-02-14T07:02:33.398Z",
          "host" => "hebust"
}
Client_output
"C:\Program ...
{"message":"hello\r","@version":"1","@timestamp":"2015-02-14T07:02:33.398Z","host":"hebust"}
```

图 7.32 日志经 TCP 方式输出(logstash 作为 Server 端)

### 7.5.7 将收集到的日志信息传输到自定义的 HTTP 接口中

可以使用 HTTP 接口的方式，将收集到的日志信息传送到自定义的 HTTP 接口中。代码段 7.37 给出的 Logstash 配置文件中设定了如何通过 HTTP 接口接收由 Logstash 传送来的日志信息。

```
#代码段 7.37: logstash 形式化配置文件,用 http 接口作为日志接收端
output {
  http {
    codec=>...                  #codec(可选), 默认 plain
    content_type=>...           #string(可选)
    format=>...                 #string, ["json", "form", "message"]其中之一
                                #(可选), 默认 json
    headers=>...                #hash(可选)
    http_method=>...            #string, ["put", "post"]其中之一(必选)
    mapping=>...                #hash(可选)
    message=>...                #string(可选)
    url=>...                    #string(必选)
    verify_ssl=>...             #boolean(可选), 默认: true
    workers=>...                #number(可选), 默认: 1
  }
}
```

## 7.6 扩展知识与阅读

redis 是一个高性能的 key-value 数据库，它的出现，在很大程度上补偿了 memcached 这类 key/value 存储的不足，在部分场合可以对关系数据库起到很好的补充作用。有关 redis 的背景知识及操作，可参阅文献[李子骅,2013] [黄健宏,2014]。通过使用 log4j，可以控制日志信息输送的目的地，也可以控制每一条日志的输出格式、定义每一条日志信息的级别等。在得到日志流以后，可以进行更进一步的分析。文献[王继民,2014]给出了互联网用户查询日志挖掘及其应用研究领域的主要技术、方法与实证研究成果。文献[李志义,2015]采用了众多流行的数据挖掘算法，如利用 K-Means 算法进行信息聚类和网页自动抽取，利用贝叶斯分类器实现信息过滤与分类，将智能 Web 算法与网站优化有机地结合起来。

## 7.7 本章小结

Logstash 架构专为收集、分析和存储日志所设计。它自身的组件架构支持通过代理对不同服务器的日志流进行管理，并最终传送至存储系统中。本章对 Logstash 的主要功

能——日志输入输出、过滤处理等进行了说明，并给出了具体的测试与应用实例。由于在Logstash中所有的工具都是可安装、可配置以及可管理的，因此，也便于和其他应用对接。将Elasticsearch、Logstash、Kibana结合起来使用，能有效应对大数据搜索与挖掘及可视化方面的应用需求，具有广阔的应用与开发前景。

# 第8章 Chapter 8

# 基于 Kibana 的数据分析可视化

> "By combining the massively popular Elasticsearch, Logstash and Kibana, Elasticsearch has created an end-to-end stack that delivers actionable insights in real-time from almost any type of structured and unstructured data source. Built and supported by the engineers behind each of these open source products, the Elasticsearch ELK stack makes searching and analyzing data easier than ever before." ________ http://www.elasticsearch.org/webinars/introduction-elk-stack/

在对大数据进行分布式索引、检索，对用户日志进行存储和处理后，亟需一个信息可视化工具来对挖掘结果进行展示。信息可视化技术是利用计算机实现对抽象数据的可视表示，来增强人们对抽象信息的感知。它不仅在揭示信息资源的广度与深度上有很大优势，而且能够将隐藏在信息资源内部的、复杂的、抽象的信息以直观的方式呈现给用户。它可以利用人们对可视模式快速识别的能力，将数据信息转化为视觉形式，形象化地揭示数据深层次的联系、趋势等，以此来帮助人们发现隐含在信息中的知识和联系，这有助于人们做出科学决策。近年来，信息可视化的研究已经引起了科研人员的密切关注，欧美一些国家启动了相应的科研计划，在理论模型和有用技术方面已取得了较大进展，并在 IEEE 系列会议、信息可视化和人机交互相关会议（如 SIGGRAPH、ACM PVG、ACM SIGCHI 等）发表了一些重要的研究成果。例如，河流模型是将得到的海量信息资源集合，按照时间维构造一个类似河流的可视化显示方式；关联分析模型则是按照新闻报道的要素，通过提取并分析时间、地点、分类等属性，构造关联分析模型，用于发现它们之间可能存在的关联关系。为了建立数据的可视映射，有的文献提出了可视化参考模型用来描述原始数据、数据表格、可视化架构和视图之间的转换关系，以及用户根据任务通过人机界面进行数据变换、可视化映射、视图变换等操作。但是对于一般用户而言，很多可视化模型或工具的使用门槛较高。

Kibana 是一个开源的使用 Elasticsearch、Logstash 分析结果的基于 Web 的可视化工具，可以帮助汇总、分析和搜索重要数据日志并提供友好的界面，例如通过 histogram 面板，配合不同条件的多个查询，可以对一个事件给出从多个不同维度看的组合统计结果并给出不同的时间序列走势；通过 Kibana 的交互式界面可以很快将异常时间或者事件范围缩小到秒级别或者个位数。Kibana 支持强大的 Lucene query string 语法，能用上 Elasticsearch

的过滤、统计功能。由于 Kibana 是用 HTML 和 Javascript 构建的，所以它不需要任何后台组件，对于 Elasticsearch 用户而言，它极易上手。本章以前述章节中提到的 Logstash 日志中的数据进行可视化为目的，介绍 Kibana V3 在可视化方面的实际使用方法。有了 Kibana V3 的基础，并作为对新知识的补充，在本章最后简介目前的最新版 Kibana V4 的使用。

## 8.1 安装 Kibana

Kibana 的安装非常简单。用户可以登录 Kibana 的官方站点（如 http://www.elasticsearch.org/overview/kibana/installation/）进行下载，如图 8.1 所示。将其解压并复制到相应的 Web 容器所在的相应的文件夹下，启动相应的 Web 容器后，在浏览器的地址栏中输入相应的 URL，即可开启 Kibana 之旅，如图 8.2 所示。本章首先以 Kibana V3.1.2 版本为例进行讲解。

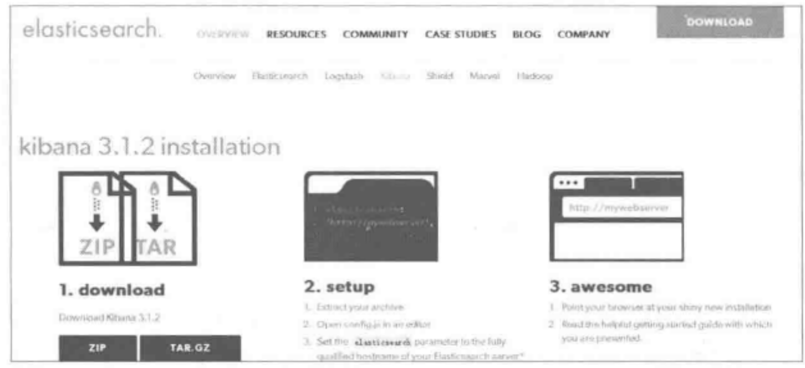


图 8.1 下载 Kibana

> Tips：在使用 Kibana 之前，首先要在计算机上安装好 Web 服务器（如 Apache、Nginx，当然也可以使用 Tomcat）；确保安装配置好 Elasticsearch 并启动之。

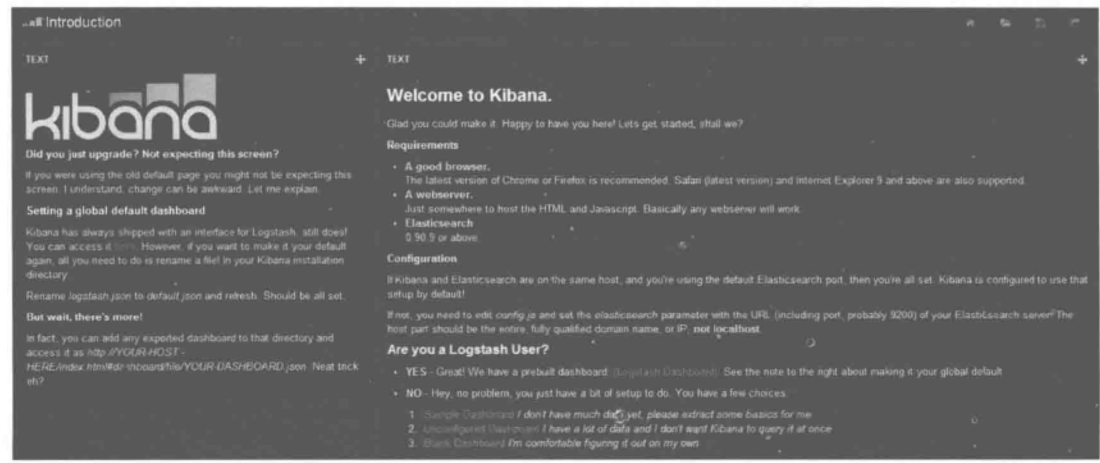


图 8.2 Kibana 主面板

Tips：在 Elasticsearch 发布 1.4 版后，使用 Kibana3 访问 Elasticsearch 1.4 集群会显示错误，这是因为 Elasticsearch 1.4 增强了权限管理。这需要在 Elasticsearch 配置文件 elasticsearch.yml 中添加下列配置并重启服务后才能正常访问[饶琛琳,2015]：

```
http.cors.enabled: true
http.cors.allow-origin: "*"
```

## 8.2 Kibana 概述

Kibana 是由类似栅格的仪表盘(Dashboard)构成的，而仪表盘是由行(Row)和面板(Panel)组成的。一般地，仪表盘由多个行组成，也就是说，在仪表盘面板上可添加多个行，而这些行高都是可以设置的。在每个行上可以添加多个面板。存在于某个行中的面板的高度是由所在行的高度决定的，但其宽度是可以设置的。

Tips：行是组成仪表盘的基本单位，一个仪表盘由多个行组成，在每一行上可以放置多个面板。

单击图 8.2 右下角的 Sample Dashboard 或 Blank Dashboard 按钮，会分别打开一个有简单布局的仪表盘以及一个空白的仪表盘。用户可以在仪表盘顶部的 Query 框中书写，Kibana 允许使用者采用 Lucene query string 语法来检索 Elasticsearch 中的数据。另外，在进行可视化时要指定待处理的索引信息源。点击仪表盘右上角的小齿轮标记的配置按钮，可以设置拟处理的索引信息源。默认情况下，Kibana 指向的是基于 Elasticsearch 的名为 _all 的索引，可将其理解为全部索引，如图 8.3 所示。当然，使用者也可以在这里指定特定的索引名，如本章多处使用了 whale 索引。

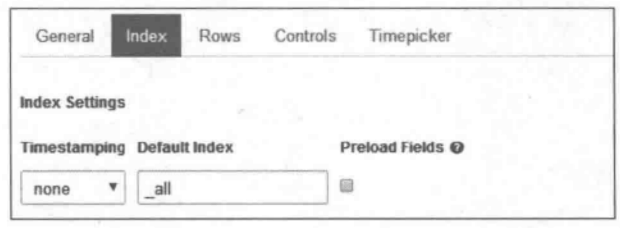


图 8.3　配置索引数据源

一般地，新的空白仪表板(Blank Dashboard)只有展开的 Query 和 Filtering，如图 8.4 所示。在顶栏上有针对时间的 Time filter 过滤选择器，如图 8.4 所示。单击右下角的添加行(ADD A ROW)按钮 ADD A ROW ，如图 8.4 所示，可以在仪表盘上添加一个新行。此时，可以给新建立的行取名，然后单击 Create Row 按钮，会看到创建的新行出现在左侧的行列表

图 8.4 可以在空白的仪表盘上添加新行

里。此时，仪表盘上多了一些新元素，即左侧多出来的三个小的分别为蓝色、橘黄色、绿色的长方形，它们的作用分别是折叠行（上部的蓝色三角图标）、配置行（中间的橘黄色齿轮图标）、添加新的面板（下方的绿色加号图标），如图 8.5 所示。

图 8.5 行操作按钮

> Tips：单击折叠行（蓝色三角图标）按钮后该行将被折叠。被折叠行里的面板不会刷新数据，因此也就不要求 Elasticsearch 资源。折叠行可用于那些不需要经常看的数据。

## 8.2.1 在仪表盘上添加新行

单击图 8.4 右下角的 ADD A ROW 灰色按钮，可以弹出如图 8.6 所示的添加新行对话框，在这里可以设置拟添加的行名称、行高（默认是 150px，可根据需要修改），如图 8.6 所示。

图 8.6 设置新行名称和高度

## 8.2.2 在行中添加新面板

在某个行中可以方便地添加新的面板。点击某个行左侧的添加新面板按钮（绿色加号图标，如图 8.5 下方按钮所示）可以打开增添并设置行的对话框，如图 8.7 所示。在 Select Panel Type 中可以选择该面板的功能类型。例如，选择 terms 面板可以用上

Elasticsearch 的 terms facet 功能来可视化一个字段内最经常出现的几个值的统计结果。

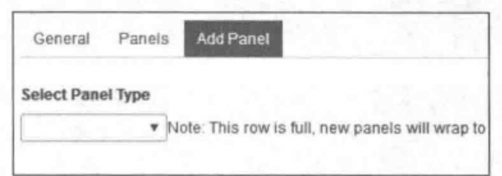


图 8.7 添加新的面板

Tips：在 Kibana 中，可供选择的面板类型（注：在如图 8.7 所示的对话框的 Select Panel Type 中选择）有 better map、column、histogram、hits、map、pie、sparklines、table、terms、text、trends 等类型。

选择面板类型以后，可以在对话框中设置相应的类型参数。选择不同的面板类型，弹出的可选项也是不同的。图 8.8 是基于 terms 的面板（可在图 8.8 左上角的 Select Panel Type 中选择）。部分可选项参数及其含义如下：

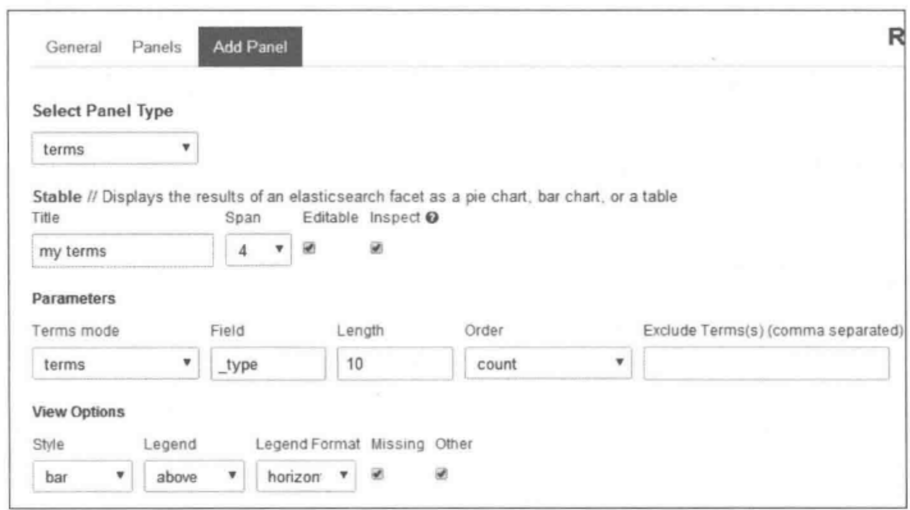


图 8.8 设置基于 terms 面板的参数

- Title——面板名称。
- Span——面板宽度。Kibana 的仪表板等分成 12 个 span，因此面板最宽就是 12 个 span。此处默认值是 4。
- Editable——是否在未来的面板上显示编辑按钮。
- Inspect——面板是否允许用户查看所用的请求内容。

另外，如果一个行中包括诸多不同的面板，在图 8.8 中单击顶端的 Panels 标签，可以对这些面板进行设置，如可以在这里设置每个面板的宽度、删除某个面板、移动某个面板的位置、是否隐藏某个面板等，如图 8.9 所示。显然，图中在这个行中含有两个基于 terms 的面

板，其默认的 Title 分别是 1-1 和 1-2。

General | Panels | Add Panel    **Row Settings**

Panels

| Title | Type | Span (10/12) | Delete | Move | Hide |
|---|---|---|---|---|---|
| 1-1 | terms | 4 | × | ↓ | ☐ |
| 1-2 | terms | 6 | × | ↑ | ☐ |

Add Panel | Save | Cancel

图 8.9 对行中诸多面板的属性设置

另外，面板也可以在当前行和其他行之间移动。按住某个面板右上角的十字架形状小图标，如图 8.10 中的按钮所示，然后拖动到目的行中即可。当然，利用图 8.10 提供的按钮，还可以设置、复制、配置、删除相应的面板。

当对仪表盘中的行和列都构造完毕以后，可以将设计的仪表盘持久化到 Elasticsearch 里，每个持久化到 Elasticsearch 里的仪表板都有一个对应的 URL。保存时，可以在对话框中输入拟保存的名字，如图 8.11 所示。之后，单击界面上方的加载 Load 按钮，可以将指定名称的仪表盘载入。

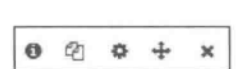

图 8.10 面板调整快捷方式

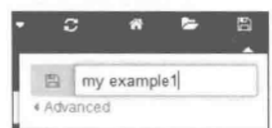


图 8.11 按指定名称保存仪表盘

可以对仪表盘附加一个指定的静态 URL，如图 8.12 所示。这样，便于在应用程序中调用这个仪表盘。当然，已保存的仪表板可以通过浏览器地址栏里的 URL 分享出去。

图 8.12 为仪表盘自动指定 URL

## 8.2.3 设置 Query 和 Filtering

Kibana 允许使用者采用 Lucene query string 语法搜索 Elasticsearch 中的数据。拟提交的请求可以在页面顶部的请求输入框中书写。

在图 8.4 的上方可以见到蓝色的 QUERY 按钮 QUERY ▸ 和绿色的 FILTERING 按钮 FILTERING ▸。其中，蓝色的 QUERY 按钮 QUERY ▸ 是设置查询条件，在这里可以选择查询条

件，可以是基于 Lucene 表达式的查询，还是基于正则表达式的查询，抑或是对相应字段的 TopN 统计，如图 8.13 所示（此时如果要删除这个 QUERY，单击其左上角的 X 图标[⊙˚]即可；另外，Kibana 会自动给你的请求分配一个可用的颜色，使用者也可以手动设置颜色），之后会在 QUERY 栏下方出现设定的 QUERY，在这里可以单击 QUERY 输入框右侧的＋号[⊙˚ Q+]，即可添加一个新的 QUERY 框，如图 8.15 所示。

如果选择采用基于 Lucene 的表达式，则可以在图 8.14 中相应的框中输入基于 Lucene 的表达式[lucene ▾]；如果选择 topN 方式的表达式[topN]，例如对 statusCode 字段统计 TOP-9，可以如图 8.14 所示进行设置。

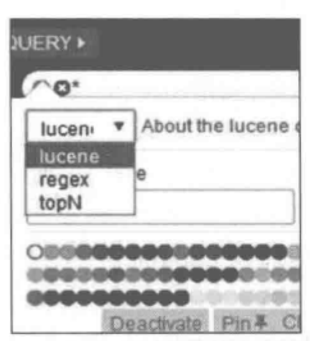


图 8.13　设定 QUERY

图 8.14　在基于 topN 的 QUERY 中设置具体的查询条件

> Tips：Kibana 中给出部分基于 Lucene 的查询示例如下：
> - 字段 login 为真或假的，如“login:true”或“login: false”，如图 8.15 所示。
> - 字段 title 包含特定字符的，如需找出在 title 字段中包含“quick”或者“brown”字符串，可表示为 title: (quick brown)。
> - 可以使用通配符，其中？代表任意一个字符，* 代表零到任意多个任意字符，如图 8.15 所示。
> - 指定字段的范围，如 statusCode: [500 TO 599]或 statusCode[200 TO * ]等，如图 8.15 所示。
> - 匹配全部：*。

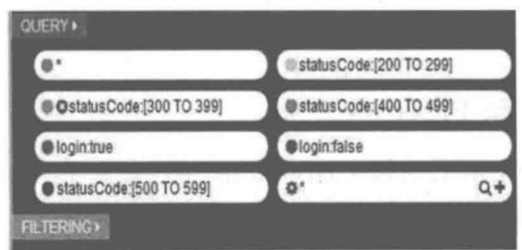


图 8.15　已经完成设定的 QUERY，每一个对应一条统计线

正是由于在 QUERY 中进行了设置，因此在后续的面板操作中，如果选择基于 select 的查询，才能在其后面显示出对应的 QUERY，如图 8.16 所示。比较图 8.15 和图 8.16，可以很清楚地看到，图 8.16 中显示的查询条件及其对应的颜色，都是在图 8.15 中事先设置好的，也就是说，二者是一一对应的。

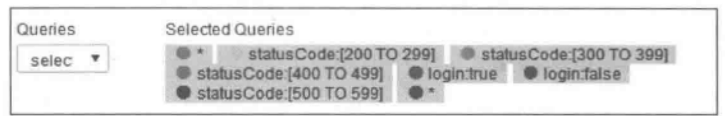


图 8.16 面板操作过程中的 Queriy 操作列表

针对绿色的 FILTERING 按钮 FILTERING ▸，在使用者添加了不同类型的面板后，如果在对应的可视化图表上用鼠标划出一个矩形框，则会在所有的可视化面板中展示相应的这个选择区间中的统计数据，而这个选择区间就是在 Filtering 中显示的内容，如图 8.17 所示。通过 Kibana 提供的这种交互式界面，可以很快将异常时间或者事件的范围缩小到很短的时间范围内。

图 8.17 过滤操作集合

## 8.3 常用面板类型

### 8.3.1 Histogram

在图 8.7 的 Select Panel Type 下拉列表框中选择 histogram，可以建立一个基于 histogram 的面板。Histogram 面板可显示序列图，包括几种不同的模式，可以显示计数、平均数、最大值、最小值、数值字段和等多种测度，如图 8.18 所示。由于这些属性大都能够通过对话框中的英文标记进行解读，因此这里不再一一赘述，只对部分属性说明如下（本章后续同）：

- Chart Value——Y 轴的测度（如 count、mean、max 等）。
- Value Field——测度要求的字段类型。除 count 测度外，其他测度要求定义 Value Field 参数。
- Time Field——X 轴字段。一般情况下应该是在 Elasticsearch 中定义为时间类型的字段。
- Scale——缩放因子。

- Y Format——Y 轴数值格式，可选值包括 none、bytes、short 等。
- xAxis、yAxis——是否显示 X 轴和 Y 轴。

Select Panel Type
histogram Note: This row is full, new panels will wrap to a new line. You should add another row.
Stable // A bucketed time series chart of the current query or queries. Uses the Elasticsearch date_histogram facet. If using time stamped indic
Elasticsearch cluster
Title Span Editable Inspect
4
Values Transform Series Time Options
Chart value Value Field Scale Seconds Derivative Time Field Time correction Auto-interval Resolution
mean Start typing 1 @timestamp brows 100
Style
Chart Options Multiple Series
Bars Lines Points Selectable xAxis yAxis Y Format Stack Percent Stacked Values
none cumu
Header Legend Grid
Zoom View Legend Query Counts Min / Auto Max / Auto
0
Queries
Charted
Queries
all

图 8.18 histogram 面板

下面通过一个简单示例来说明如何定义 histogram 类型面板。假定已经在日志文件中对用户的每一个 HTTP 请求进行了记录，并由网站流量 Page View(PV)来表征它们。现在需要统计 PV 值，并在 histogram 类型面板显示。要求横轴显示时间戳(相应字段为@timestamp)，纵轴是 PV 计数值(count 测度)，并且要在这一张图中显示两个不同的 PV 值：一个是总体 PV 值，一个是状态码(相应字段为 statusCode)在[200,299]区间的分布情况，具体操作如下：

首先增添一个面板 Panel 并选用 histogram 类型，可以设定其标题(此例为 Page View Status，如图 8.19 所示)；纵轴测度 Chart Value 选用计数值 count，用于统计基于 PV 的计数；横轴 Time Field 选用@timestamp 字段，用于显示相应的时间；在查询条件 Queries 中选择 selected 按钮 selec ▾ ，并选择在右侧出现的统计所有内容(以 * 显示)以及范围在[200,299]的内容，如图 8.19 的 * ● statusCode:[200 TO 299] 所示，以便能在这一张图中显示两个不同的 PV 值。最后效果如图 8.20 所示。

> Tips：请注意在图 8.19 最下方的 Selected Queries 下方的状态，带有粗线框的是选中的状态，分别对应查询所有内容(以 * 显示)、查询 PV 范围在[200,299]的内容。

图 8.19 配置基于 histogram 的面板属性

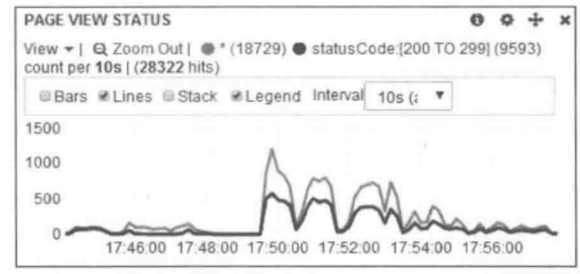


图 8.20 在一个基于 histogram 的面板中显示两种不同的统计结果

## 8.3.2 Table

在图 8.7 的 Select Panel Type 下拉列表框中选择 table，可以建立一个基于 table 的面板。基于 table 的面板是一个表格面板，在这里有一个可排序的分页文档，可以定义需要排列哪些字段，如图 8.21 所示。下面对 table 面板中的部分可选项进行简述。

- Trim Factor——裁剪因子，参考表格中的列数来决定裁剪字段长度，如设置裁剪因子为 100，表格中有 5 列，那么每列数据就会被裁剪为 20 个字符。
- Local Time——设为 true 时，可调整 Time Field 的数据遵循浏览器的本地时区。
- Queries——请求对象，描述本面板使用的请求，选项有 all、pinned、unpinned、selected 等。all 是 Kibana 通过 Elasticsearch 的_mapping API 直接获取的所有索引内存在的字段。

图 8.21 table 面板及其属性设置

- Columns——字段选择，在 Columns 中可以输入拟在面板中显示的字段，选择某个字段后，单击 Columns 右侧的加号按钮 可以将相应的字段加到此表格中，如图 8.22 所示。当添加部分字段后，会在 Columns 下方以灰色底纹显示出已经加入到 table 面板中的部分字段，如 和 按钮所示。

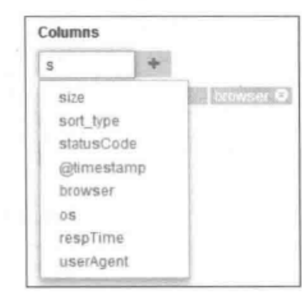


图 8.22 字段选择

基于上述方法可以建立一个基于 table 的面板，在这里可以单击左侧的字段列表，复选拟在表格中显示的内容，如图 8.23 所示。

| size | sort_type | browser | os | location | query | reqTime | referfer |
|---|---|---|---|---|---|---|---|
| 16958 | | Chrome | Windows 8.1 | 河北省石家庄市 | | 0.003 | - |
| 284 | | Chrome | Windows 8.1 | 河北省石家庄市 | | 0 | - |
| 16958 | | Sogou Explorer | Windows 8.1 | 北京市 | | 0 | - |
| 0 | | Chrome | Windows 8.1 | 河北省石家庄市 | | 0.039 | - |
| 2082 | | Chrome | Windows 8.1 | 河北省石家庄市 | | 0.003 | http://ip.heb |
| 2158 | | Chrome | Windows 8.1 | 河北省石家庄市 | | 0.593 | - |
| 0 | | Chrome | Windows 8.1 | 河北省石家庄市 | | 0.025 | - |
| 50607 | | IE | Windows 8.1 | 河北省石家庄市 | | 0 | http://ip.heb |

图 8.23 建立基于 table 的面板

## 8.3.3 Map 和 Bettermap

在图 8.7 的 Select Panel Type 下拉列表框中选择 map 和 bettermap，可以建立基于 map 和 bettermap 的面板。map 和 bettermap 都是基于地图的面板。和上述面板类似，这里可以给这些地图设定标题、显示字段（Field 字段），而即将显示的地图类型可以是世界地图、欧洲地图、美国地图等，如图 8.24 所示。Bettermap 面板是为了解决 map 面板地图种类

太少且不方便大批量添加各国地图文件的问题开发的，它采用了 leaflet 库，其 L. tileLayer 加载的 OpenStreetMap 地图文件都是在使用的时候单独请求下载的[饶琛琳,2015]。

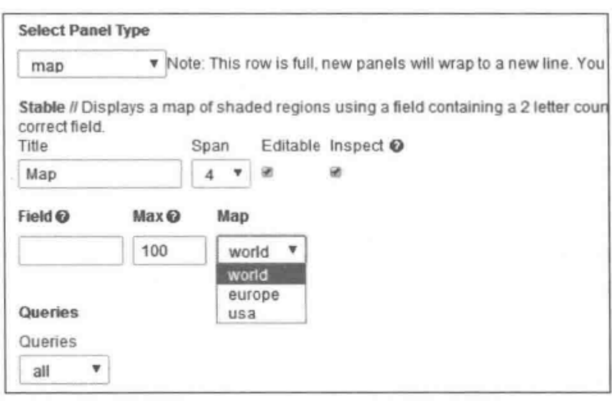


图 8.24 选择地图种类

## 8.3.4 Terms

在图 8.7 的 Select Panel Type 下拉列表框中选择 terms 可以建立基于 terms 的面板。这是基于 Elasticsearch 的 terms facets 接口的数据展现，可以实现条带图、饼图等，如图 8.25 所示。下面对 terms 面板中的部分可选项进行简述。

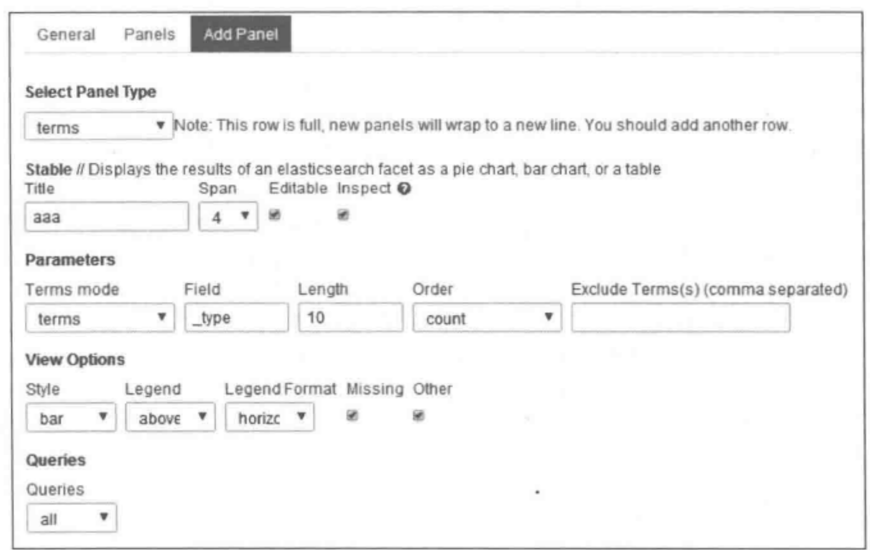


图 8.25 terms 面板及其属性设置

- Terms Mode——如选择 terms（即普通计数，类比 SQL 中的 group by），则排序 Order 的可选项有 count、term、reverse_count、reverse_term 等；如这里选择 term_stats（如图 8.26 所示，可在 terms 的基础上获取另一个数值类型字段的统计值作为显示内容，可选的统计值有 count、total_count、min、max、total、mean 等），则排序

Order 可选项有 count、reverse_count、total、reverse_total、min、reverse_min、max、reverse_max、mean、reverse_mean 等。

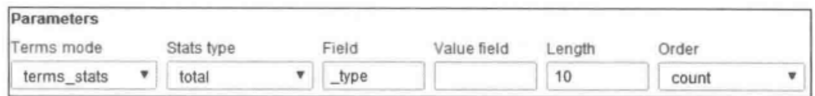


图 8.26 terms mode 选择

- Field——用于 term facet 中统计的字段名称。
- Exclude Term(s)——要从结果数据中排除掉的 terms。
- Missing——不选择该复选项时,可以不显示数据集内有多少结果没有使用者所指定的字段。
- Other——不选择该复选项时,可以不显示聚合结果在使用者的 size 属性设定范围以外的总计数值。
- Style 模式——如选择 pie,则会出现一些相关选项,如图 8.27 所示。其中的 Donut 选项是指甜圈样式;Tilt 选项是指倾斜饼变成椭圆形;Lables 选项是指在饼图分片上绘制标签。如果选择条带 bar 或者饼图 pie 模式,会有其他不同但类似的选项,这里不再详述。

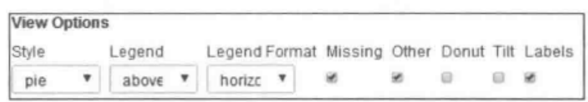


图 8.27 style 模式及其相应选项

## 8.3.5 Text

在图 8.7 的 Select Panel Type 下拉列表框中选择 text 可以建立基于文本的面板。Text 文本面板用来显示静态文本内容,支持 markdown、简单的 html 和纯文本格式,如图 8.28 所示。下面对 text 面板中的部分可选项进行简述。

图 8.28 text 模式及其相应选项

Tips：Markdown 是一种易读易写的标记语言。Markdown 的语法由一些符号所组成，如在文字两旁加上星号，看起来就像 * 强调 *。Markdown 的区块引用看起来就真的像是引用一段文字。

- Mode——选择此 text 面板所支持的文字格式，支持 html、markdown、text。
- Content——文字面板中即将显示的内容，用 mode 参数指定的标记语言书写。使用时，只需在面板内容框中输入符合语法的 Html 或 Markdown 文本或纯文本即可。图 8.29 是用网站 http://mahua.jser.me/中的示例 Markdown 格式文字完成的一个 text 面板。

图 8.29 text 面板示例

### 8.3.6 Sparklines

在图 8.7 的 Select Panel Type 下拉列表框中选择 sparklines 可以建立基于微型时间的面板。Sparklines 的可选值有 count、mean、max、min、total 等。除 count 外，其他都需要定义 Value Field 字段，如图 8.30 所示。下面对 sparklines 面板中的部分可选项进行简述。基于 sparklines 的面板示例如图 8.31 所示。

- Time Field——X 轴字段(须是 Elasticsearch 中的时间类型字段)。
- Value Field——如果 Chart value 设置为 mean、max、min、total，则 Y 轴字段须是数值类型字段。

图 8.30 sparklines 面板属性设置

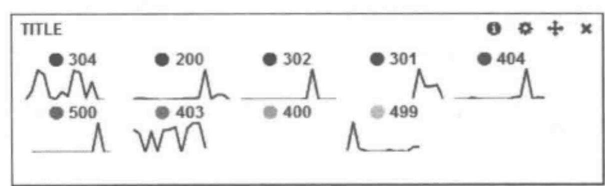


图 8.31 基于 sparklines 的面板

## 8.3.7 Trends

在图 8.32 的 Select Panel Type 下拉列表框中选择 trends，可以建立基于 trends 的面板。它以类似证券报价器风格展示请求随着时间移动的情况。如当前时间是 1:10p. m.，时间选择器设置的是“Last 10m”，而本面板的“Time Ago”参数设置的是“1h”，那么面板会显示的是请求结果从 12:00a. m. ～12:10p. m. 以来相关数据变化了多少。下面对 trends 面板中的部分可选项进行简述，如图 8.33 所示。

General Panels Add Panel

Select Panel Type

trends Note: This row is full, new panels will wrap to a new line. You should add another row.

Beta // A stock-ticker style representation of how queries are moving over time. For example, if the time is 1:10pm, your time picker was set to "Last 10m", and the "Time Ago" paramet query results have changed since 12:00-12:10pm

Title Span Editable Inspect

4

Settings

Use Elasticsearch date math format here (eg 1m, 5m, 1d, 2w, 1y)

Time Ago 1d

Font Size 14pt

List Format vertic

Queries

Queries all

图 8.32 trends 面板属性设置

- Time Ago——描述需要对比请求的时期的时间数值型字符串。
- List Format——选择 horizontal 或 vertical。

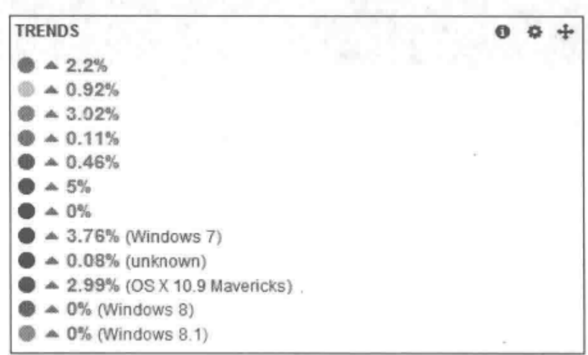


图 8.33 trends 面板示例

## 8.4 网站性能监控可视化应用的设计与实现

作为应用示例，基于本书前面介绍的基于 Logstash 的网站操作日志，本节介绍基于 Kibana 的网站性能监控可视化应用的设计与实现。

### 8.4.1 概述

实验数据集来源自 Elasticsearch 中索引文件名为 whale 类型文件为 log 的数据文件，该索引文件主要字段及其存储的实际数据示例如图 8.34 所示，这里的各个字段的含义如其名称的英文所示，此处不再赘述。拟用 Kibana 实现的网站性能监控由多个行组成，如图 8.35 所示。

```
event_type: "RESOURCE",
@timestamp: "2015-01-12T13:05:26.000Z",
login: true,
uid: "9",
userAgent: "Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/39.0.2171.95 Safari/537.36",
browser: "Chrome",
device: "Personal computer",
os: "Windows 7",
size: 0,
http_method: "GET",
ip: "121.193.210.107",
location: "河北省石家庄市",
reqTime: 0,
respTime: 0,
statusCode: 304,
referfer: "http://iip.hebust.edu.cn/recommender/offline/report/review/list?
page=0",
referferDomain: "iip.hebust.edu.cn",
uri: "/resources/App_Themes/Tehemes/Css/top.css"
```

图 8.34 日志文件示例

图 8.35 仪表盘中的各个行

限于篇幅，这里不对所有行中的面板操作进行详述。本节以图 8.35 中的 PV 行、响应/请求时间行、流量行、状态码行、UA 行为例，简要说明各行中的面板操作与设置。

## 8.4.2 Page View

Page View（综合浏览量，简写为 PV）是指网站各网页被浏览的总次数，它也是反映一个网站受欢迎程度的重要指标之一。一般地，一个访客有可能创造十几个甚至更多的 PV。监控 PV 指标是判断网站访问流量最常用的方式。本节采用基于 histogram 的面板显示 PV。该面板的基本属性如图 8.36 所示。假设希望统计在特定的时间下，全部命中文档个数以及当时返回的错误码在[500,599]之间的文档个数。为此，在图 8.37 中设定横轴是以计数形式显示的@timeStamp 字段，图 8.38 中设置图表纵轴的测度，其中之一是全部命中的文档个数（对应图 8.38 中的 ● * 标记，其最终统计结果对应图 8.39 中有凸起的浅色曲线），另一个是对应返回的错误码在[500，599]之间的文档个数（对应图 8.38 中的 ● statusCode:[500 T 标记，其最终统计结果对应图 8.39 中较平缓的深色曲线）。

图 8.36 PV 行名称

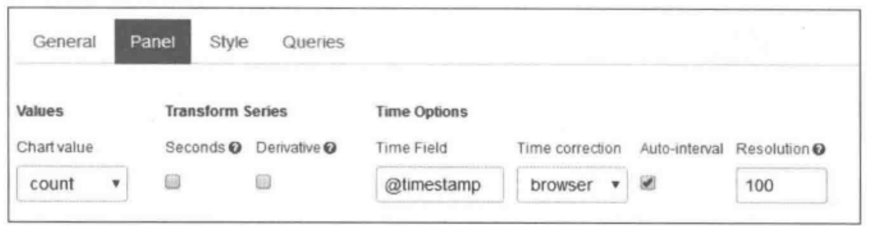


图 8.37 图表横轴及其测度

图 8.38 图表纵轴及其测度

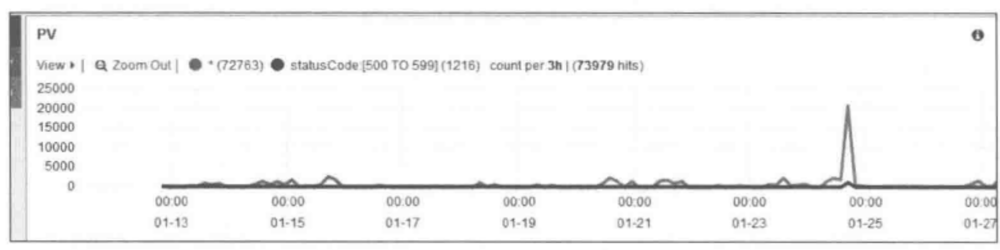


图 8.39 PV 统计的实际结果

## 8.4.3 响应/请求时间

网站响应时间与网站请求时间的配置相似，这里仅以网站响应时间的实现为例进行说明。该面板的基本属性及其设置与上述 PV 操作类似，如图 8.40 所示。图表横轴及其测度如图 8.41 所示。

General Panel Style Queries

Stable // A bucketed time series chart of the current query lighest possible load to your Elasticsearch cluster

Title: 响应时间 Span: 6 Editable Inspect

图 8.40 响应时间的行名称

图 8.41 图表横轴及其测度

图表纵轴及其测度如图 8.42 所示，注意这里是显示全部命中文档个数(对应图 8.42 中的[●*]标记)，这也是唯一选中的内容，因此在最终结果的图 8.43 中只有一条曲线。图 8.43 左侧是网站响应时间统计分析情况，图 8.43 右侧是网站请求时间统计分析情况。

图 8.42 图表纵轴及其测度

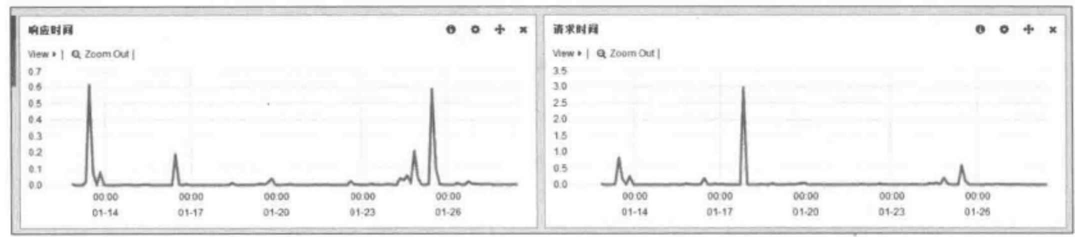


图 8.43 响应/请求时间

## 8.4.4 流量走势与统计

在接下来的行中，进行网络流量走势的可视化统计。这一行放置两个面板(这两个面板的设置参数参见图 8.44)，分别是流量走势(基于 histogram 实现)与流量统计(基于 status 实现)。

对流量走势面板中的横轴坐标设置如图 8.45 所示。在 Panel 面板中设置采用@timestamp 字段作为横轴，采用 size 字段的 total 测度值作为纵轴坐标，而对该面板的整体效果设置如图 8.46 所示。

图表纵轴及其测度如图 8.47 所示，这里是显示全部命中文档个数(对应图 8.47 中的[●*]标记)，这也是唯一选中的内容，因此在最终结果图的左侧只有一条曲线。

图 8.44　两个面板的属性设置

图 8.45　横、纵坐标设置

图 8.46　面板其他属性设置

图 8.47　纵轴测度设置

流量统计(基于 status 实现)的设置与流量走势(基于 histogram 实现)类似。部分属性设置如图 8.48 和图 8.49 所示,此处不再赘述。它们的总体效果如图 8.50 所示。

General Panel Queries **Stats Settings**

**Beta** // A statistical panel for displaying aggregations using the Elastic Search statistical facet query.

Title: 流量统计 Span: 6 Editable: ✔ Inspect: ✔

图 8.48 一般设置

General Panel Queries **Stats Settings**

**Details**

Function: total Field: size Unit:

**Formating**

Format: bytes Font Size: 72pt Display Breakdowns: yes Label column name: Query

Value column name: Value

图 8.49 面板风格设置

图 8.50 流量走势与统计的总体效果

## 8.4.5 状态码监控

在图 8.35 的状态码行设置状态码信息,这里包括 3 个面板,分别是基于 histogram 的状态码监测、基于 terms 的状态码、基于 trends 的状态码走势。对该行的总体设置,如对包含的各个面板名称、类型、宽度等的设置方法,参见图 8.51。具体细节不再赘述。

对基于 histogram 的状态码监测面板的设置参见图 8.52、图 8.53、图 8.54。它们分别统计网站返回的状态码(字段 statusCode)在[300,399]、[400,499]、[500,599]范围内的分布情况。

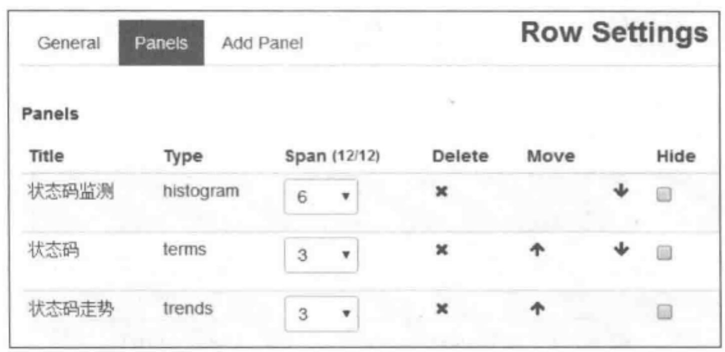


图 8.51 行总体设置

General Panel Style Queries Histogra
Values
Chart value
count
Transform Series
Seconds
Derivative
Time Options
Time Field
@timestamp
Time correction
browser
Auto-interval
Resolution
100

图 8.52 状态码监测面板横纵坐标设置

General Panel Style Queries Histogram Set
Chart Options
Bars Lines Points Selectable xAxis yAxis Y Format
none
Multiple Series
Stack Percent Stacked Values
cumulati
Header
Zoom View
Legend
Legend Query Counts
Grid
Min / Auto
0
Max / Auto

图 8.53 状态码监测面板 Style 设置

对基于 terms 的状态码面板的设置，参见图 8.55 和图 8.56。它们分别统计网站返回的所有状态码(统计字段是 statusCode，如图 8.55 所示)的分布情况。

对基于 trends 的状态码走势面板的设置，参见图 8.57 和图 8.58。它们统计和一天之前的情况对比(见图 8.57 中的 Time Ago 字段 1d 设置)，并分别统计网站返回的状态

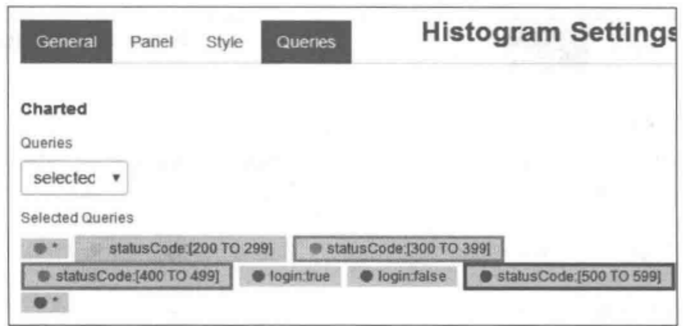


图 8.54 状态码监测面板拟显示的 3 种统计内容

General Panel Queries Term

Parameters

Terms mode: terms　Field: statusCode　Length: 7

Order: count　Exclude Terms(s) (comma separated)

View Options

Style: table　Font Size: 10p　Missing　Other

图 8.55 状态码面板统计字段设置

General Panel Queries Tern

Queries: selected

Selected Queries

* statusCode:[200 TO 299] statusCode:[300 TO 399] statusCode:[400 TO 499] login:true login:false status *

图 8.56 状态码面板拟显示的一种统计内容

General Panel Queries Trends Se

Settings

Use Elasticsearch date math format here (eg 1m, 5m, 1d, 2w, 1y)

Time Ago: 1d

Font Size: 36pt

List Format: vertical

图 8.57 状态码走势面板设置

码(字段 statusCode)在[200,299]、[300,399]、[400,499]、[500,599]范围内的分布情况。这些统计范围分别对应 statusCode:[200 TO 299]、● statusCode:[300 TO 399]、● statusCode:[400 TO 499] 和 ● statusCode:[500 TO 599]。状态码监控的总体效果如图 8.59 所示。

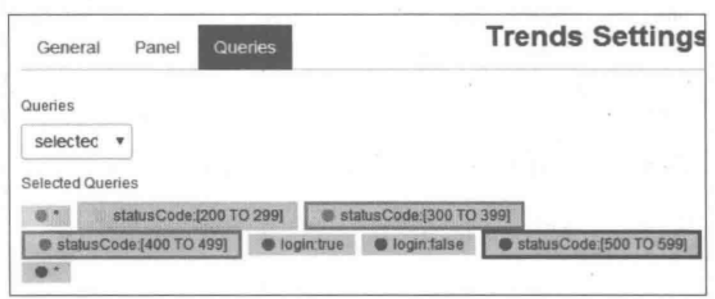


图 8.58 状态码走势面板拟显示的几种统计内容

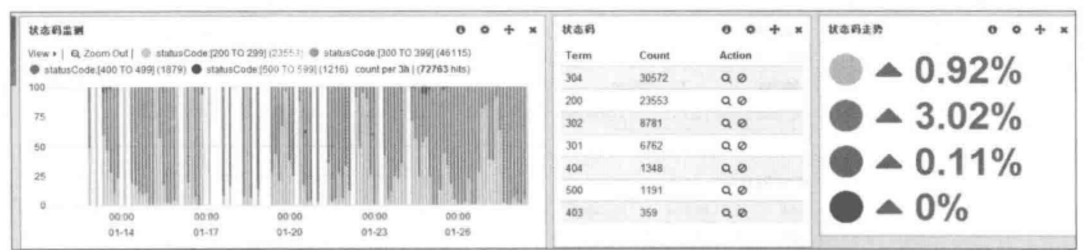


图 8.59 状态码监控的总体效果

## 8.4.6 UA 行

UA 行(User Agent)设置的功能是对用户上网使用的设备、使用的操作系统、浏览器的分布情况进行可视化显示,最终效果如图 8.60 所示,其中前三个面板是基于 pie 实现的,它们分别对 device(设备)、os(操作系统)、browser(浏览器)字段进行统计,最后一个面板是基于 bar 实现的(对 OS 字段进行统计)。行中的三个面板的查询 Queries 均设置均为 all,各个面板 Panel 的设置如图 8.61 至图 8.65 所示。

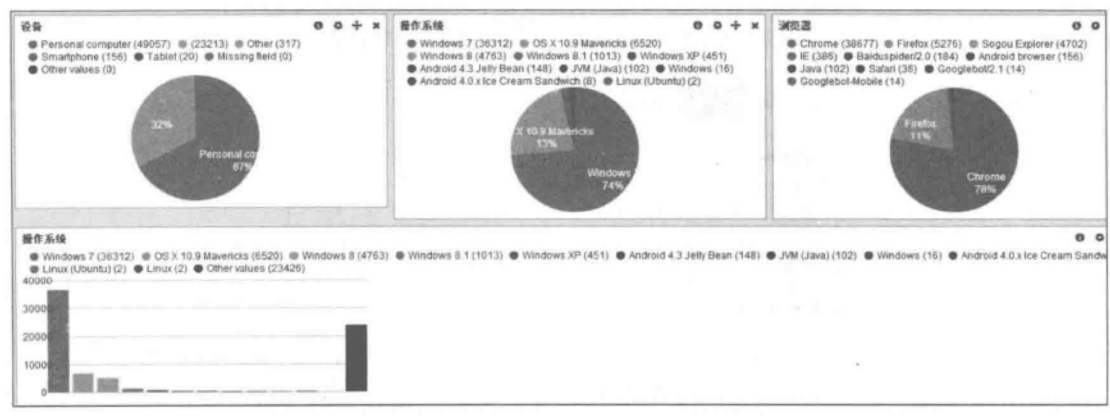


图 8.60 总体效果

General Panels Add Panel **Row Settings**

**Panels**

| Title | Type | Span (24/12) | Delete | Move | | Hide |
|---|---|---|---|---|---|---|
| 设备 | terms | 4 | ✖ | | ↓ | ☐ |
| 操作系统 | terms | 4 | ✖ | ↑ | ↓ | ☐ |
| 浏览器 | terms | 4 | ✖ | ↑ | ↓ | ☐ |
| 操作系统 | terms | 12 | ✖ | ↑ | | ☐ |

图 8.61 行总体设置

General Panel Queries

**Parameters**

Terms mode: terms　Field: device　Length: 10　Order: count

Exclude Terms(s) (comma separated)

**View Options**

Style: pie　Legend: above　Legend Format: horizc　Missing　Other　Donut　Tilt　Labels

图 8.62 基于 pie 实现的"设备"面板设置

General Panel Queries

**Parameters**

Terms mode: terms　Field: os　Length: 10　Order: count

Exclude Terms(s) (comma separated): unknown

**View Options**

Style: pie　Legend: above　Legend Format: horizc　Missing　Other　Donut　Tilt　Labels

图 8.63 基于 pie 实现的"操作系统"面板设置

General Panel Queries

**Parameters**

Terms mode: terms　Field: browser　Length: 10　Order: count

Exclude Terms(s) (comma separated): unknown

**View Options**

Style: pie　Legend: above　Legend Format: horizc　Missing　Other　Donut　Tilt　Labels

图 8.64 基于 pie 实现的"浏览器"面板设置

图 8.65 基于 bar 实现的“操作系统”面板设置

## 8.5 Kibana V4 简介

Kibana 目前（注：截至本书出版时）的最新版本是 V4（和 Elasticsearch 类似，从 Kibana V4 版本废弃 facets，改用 aggregations）。我们将在本节对 Kibana V4 进行简介。

首先，从 Elasticsearch 官网上下载 Kibana 最新版本 V4。之后解压，在 config 文件夹中的配置文件中，可以配置和 Kibana 连接的 Elasticsearch 信息（默认连接本机的 9200 端口，而 Kibana 运行在 5601 端口）。之后，转到 bin 文件夹下运行 Kibana。在浏览器中输入 http://localhost:5601/打开 Kibana，如图 8.66 所示。

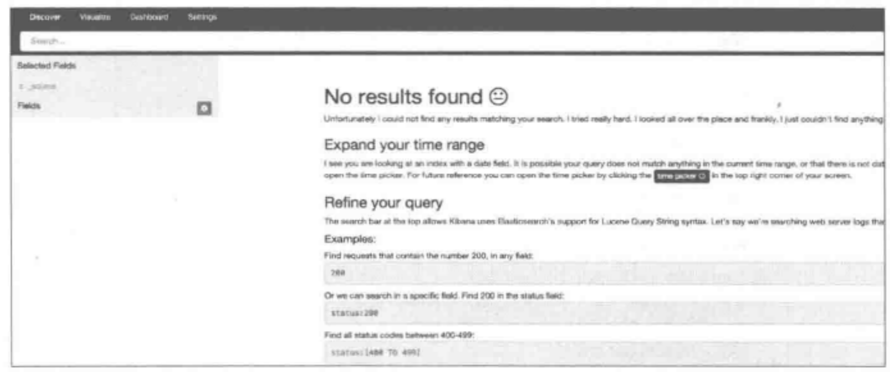


图 8.66 Kibana 启动页面

接下来，需要配置用于数据分析的索引文件 index。单击图 8.66 左上角的 Fields 按钮，弹出如图 8.67 所示的页面。

### 8.5.1 新建视图

在 Kibana 的首页选择新建，会弹出如图 8.68 所示的页面。可以选择以什么方式创建一个视图（From a new search、From a saved search），之后选择视图类型（如选择 Area chart），如图 8.69 所示。

在图 8.70 上方的输入框用于输入查询内容；右上角的时钟按钮用于选择时间区域。

Configure an index pattern

In order to use Kibana you must configure at least one index pattern. Index patterns are used to identify the Elasticsearch index to run search and analytics

☑ Index contains time-based events

☐ Use event times to create index names

Index name or pattern

Patterns allow you to define dynamic index names using * as a wildcard. Example: logstash-*

whale

Time-field name refresh fields

@timestamp

Create

图 8.67 配置索引模式

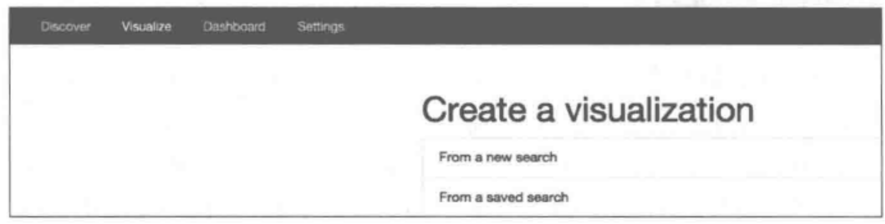


图 8.68 新建视图

图 8.69 选择视图类型

Discover Visualize Dashboard Settings

*

metrics

Y-Axis Count of documents

+ Add Aggregation

buckets

+ Add Aggregation

view options ▸

图 8.70 输入查询内容

在图 8.70 中可以配置 Y 轴，如图 8.71 所示。Y 轴可选的测度有计数、平均值、求和、最小值、最大值等，这里选择计数值，选择 buckets 测度，如图 8.72 所示，单击 Add Aggregation 选项。选择 X 轴，选择 date histogram 类型和相应的 Field 及 Interval 测度，如图 8.73 所示。之后，请求数随时间变化图就完成了。

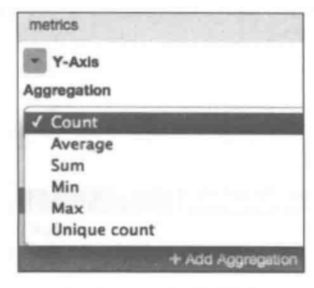


图 8.71 配置 Y 轴

在这个图中可以再添加一些其他内容。如图 8.74 所示，单击 buckets 下面的 Add Sub Aggregation 选项可添加子 Aggregation，选择 Split Area，按照图 8.75 进行配置。在保存图表后，通过 Elasticsearch 的 Head 插件可以发现在 Elasticsearch 中多了一个名为“.Kibana”的索引，这个索引中就保存着各个图表的配置信息。相似地，还可以通过 Kibana 创建其他类型的图表，此处不再赘述。

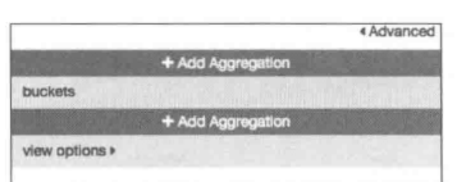


图 8.72 选择 Buckets 测度

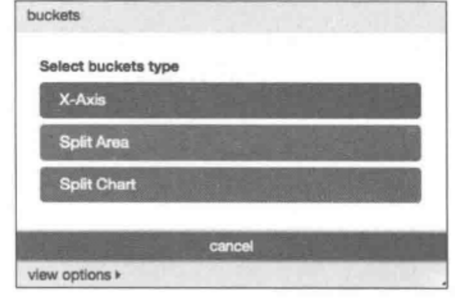


图 8.73 选择 buckets 类型

图 8.74 配置 X 轴

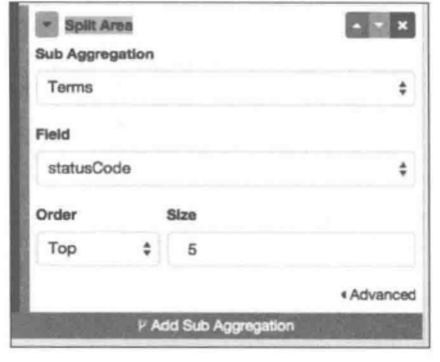


图 8.75 设置 Split Area

### 8.5.2 建立 Dashboard

单击页面最上方的 Dashboard，就能创建一个空的 Dashboard。单击加号，在这个 Dashboard 中可以添加几个我们刚刚建立好的图表。也可以通过单击叉号删掉一个图表，或者单击修改图表，在每个图的右下角可以调整图表的大小，如图 8.76 所示。

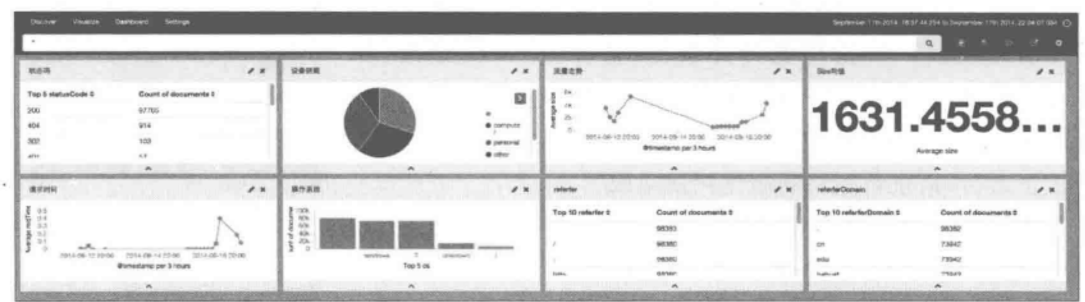

图 8.76　Dashboard 中存放不同的图表

### 8.5.3　配置

在 Kibana 页面的上方有一个 Settings 选项 Settings，如图 8.68 所示，单击可以配置 Kibana。在其中的 objects 子菜单中，可以配置我们刚刚创建的图表和面板，而在 Advance 标签中，还可以进行更为细致的设置，如图 8.77 所示。

Caution: You can break stuff here

Be careful in here, these settings are for very advanced users only. Tweaks you make here can break large portions of Kibana. Some of these settings may be undocumented, unsupported or experimental. Blanking a field will cause Kibana to use its internal defaults which may be unacceptable given other configuration directives.

Filter

| Name | Value | Actions |
| --- | --- | --- |
| **dateFormat**<br>When displaying a pretty formatted date, use this format | MMMM Do YYYY, HH:mm:ss.SSS | |
| **defaultIndex** (Default: null)<br>The index to access if no index is set | logstash-* | |
| **metaFields**<br>Fields that exist outside of _source to merge into our document when displaying it | _source, _id, _type, _index | |
| **discover:sampleSize**<br>The number of rows to show in the table | 500 | |
| **fields:popularLimit**<br>The top N most popular fields to show | 10 | |
| **histogram:barTarget**<br>Attempt to generate around this many bar when using "auto" interval in date histograms | 50 | |
| **histogram:maxBars**<br>Never show more than this many bar in date histograms, scale values if needed | 100 | |
| **csv:separator**<br>Separate exported values with this string | , | |
| **csv:quoteValues**<br>Should values be quoted in csv exports? | true | |
| **history:limit** | 10 | |

图 8.77　设置 Dashboard

## 8.6　扩展知识与阅读

基于 Kibana 的信息可视化能够完成对信息的显示处理，相对来说比较简单和易于上手，这也是 Kibana 广为流行的原因之一。从学术界来说，为了建立数据的可视映射，需要建立可视化参考模型，用来描述原始数据、数据表格、可视化架构和视图之间的转换关系，以及用户根据任务通过人机界面进行数据变换、可视化映射、视图变换等操作。相关资料可以参阅文献[周宁，2005]。

## 8.7 本章小结

Kibana 是一个为 Logstash 和 Elasticsearch 提供的日志可视化分析的 Web 工具，可使用它对日志进行高效的可视化、分析等各种操作。在已经具备的基于 Logstash 日志的数据基础上，本章首先对 Kibana 的仪表盘及面板的一般配置进行了概述，并以 Logstash 日志为例，介绍基于 Kibana 的可视化实现。另外，通过 Kibana 提供的交互式界面，可以很快将异常时间或者事件范围缩小到很短的时间范围内。

Chapter 9

# 网络信息检索与分析实践

> "By combining the massively popular Elasticsearch, Logstash and Kibana, Elasticsearch Inc. has created an end-to-end stack that delivers actionable insights in real-time from almost any type of structured and unstructured data source. Built and supported by the engineers behind each of these open source products, the Elasticsearch ELK stack makes searching and analyzing data easier than ever before" ________ http://www.elasticsearch.org/webinars/introduction-elk-stack/

随着大数据、大型综合网站以及 Web 2.0 技术的普及,越来越多的软件开发者需处理海量异构信息的索引、检索、日志挖掘、可视化等和信息检索与大数据挖掘相关的业务。本章给出了一个融合了 Elasticsearch、Logstash、Kibana 的针对静态网页内容的信息检索与分析工程实例。通过对本章的实例的介绍,可以引导读者从工程角度看 ELK 架构(Elasticsearch+Logstash+Kibana)的实际应用。

本工程涉及对静态网页信息的采集以及对 Web 检索端的设计。这里给出的信息采集是使用 WebMagic 实现了对静态网页信息的采集,并存入 Elasticsearch 的索引(名为 baike)中。由于 Java Web 及 SSH 框架的复杂性,对初学者来说有一定的难度且其内容已超出本书涉及范围,因此示例的 Web 端检索设计是基于 Python 实现的。本章首先给出对静态网页信息采集的方法,之后完成基于 Elasticsearch 的搜索设计。应用 Logstash 处理系统 Nginx 日志,用 Kibana 展示和本搜索程序相关的部分日志内容。

## 9.1 信息采集

信息采集架构如图 9.1 所示,主要包括两部分:

- 采集器——其中,App 负责采集静态网页的网页内容,并存入 Elasticsearch 中;ESClientHelper 负责 Elasticsearch Client 实例化工作等。
- 相关资源配置——包括对 elasticsearch.yml、log4j.properties 等文件的配置。其中,elasticsearch.yml 是从 Elasticsearch 文件夹下的 Elasticsearch.yml 文件复制来

的，这里有连接 Elasticsearch 的基本配置信息等；log4j. properties 的作用是设置日志输出配置等。

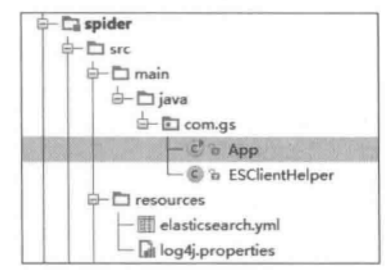


图 9.1 信息采集架构

对静态网页信息的采集方法详见代码段 9.1。webmagic 是一个简单灵活的爬虫框架。基于 webmagic，这里实现了 PageProcessor 接口，这个接口是定义网页处理逻辑的。

首先，在 main 函数中创建了一个布隆过滤器，接下来创建了一个爬虫并设定了它的起始链接、调度器、去重器、输出方式、线程等信息。这里的 process 方法中定义了如何抽取网页中的信息、如何抽取网页中的链接、如何将网页实体序列化为一个 JSON 字符串并存入 Elasticsearch 的逻辑和方法。这里的 excute()方法是用于执行 Elasticsearch 的索引请求的。下面的 MyPipeline 定义了一个空的输出。其他代码的解释和相关的 API 文档已经超出本书涉及的范畴，有关 webmagic 的情况可参考 http://webmagic. io/docs/zh/。

> Tips：布隆过滤器实际上是一个很长的二进制向量和一系列随机映射函数，可以用于检索一个元素是否在一个集合中，可用于对网页的去重处理。

```
//代码段 9.1：信息采集
package com.gs;
import org.apache.commons.lang3.StringUtils;
import org.elasticsearch.action.index.IndexRequestBuilder;
import org.elasticsearch.action.search.SearchRequestBuilder;
import org.elasticsearch.action.search.SearchResponse;
import org.elasticsearch.client.Client;
import org.elasticsearch.common.xcontent.XContentBuilder;
import us.codecraft.webmagic.*;
import us.codecraft.webmagic.pipeline.Pipeline;
import us.codecraft.webmagic.processor.PageProcessor;
import us.codecraft.webmagic.scheduler.QueueScheduler;
import us.codecraft.webmagic.scheduler.component.BloomFilterDuplicateRemover;
import java.io.IOException;
import java.text.DateFormat;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.List;
import static org.elasticsearch.common.xcontent.XContentFactory.jsonBuilder;
public class App implements PageProcessor {
  private Site site=Site.me().setRetryTimes(3).setSleepTime(1000)
  .setUseGzip(true);
```

```
  private static DateFormat dateFormat=new SimpleDateFormat("yyyy-MM-dd");
  private static Client client=ESClientHelper.getClient();
  private static final int SIZE_PER_PAGE=1000;
  public static void main(String[] args) {
  BloomFilterDuplicateRemover bloomFilterDuplicateRemover=new
  BloomFilterDuplicateRemover(9999);
  Spider.create(new App()).addUrl("http://baike.baidu.com/view/908354.htm")
   .addUrl("http://baike.baidu.com/view/3385550.htm").addUrl("http://baike.
    baidu.com/subview/28283/5418753.htm")
   .setScheduler(new QueueScheduler()
   .setDuplicateRemover(bloomFilterDuplicateRemover))
.addPipeline(new MyPipeline()).thread(5).start();
  System.out.println("爬虫退出");
}
@ Override
public void process(Page page) {
  XContentBuilder xContentBuilder=null;
  page.addTargetRequests(page.getHtml().links().regex("(http://baike\\.
baidu\\.com/view/\\d+\\.htm)").all());
  page.addTargetRequests(page.getHtml().links().regex("(http://baike\\.
baidu\\.com/subview/\\d+/\\d+\\.htm)").all());
  page.putField("title", page.getHtml().xpath("//span[@ class=
'lemmaTitleH1']//allText()").toString());
  if(page.getResultItems().get("title")==null) {
     page.setSkip(true);
     return;
  }
try {
    xContentBuilder=jsonBuilder().startObject();
    xContentBuilder=xContentBuilder.field("title", page.getResultItems()
    .get("title"));
    xContentBuilder=xContentBuilder.field("url", page.getUrl().get());
    page.putField("content", StringUtils.join(page.getHtml().xpath("//div
    [@ id='lemmaContent-0']//div[@ class='para']/allText()").all(),
    "<br>"));
    xContentBuilder=xContentBuilder.field("content", page.getResultItems
    ().get("content"));
    String lastModifyTime=page.getHtml().xpath("//span[@ id=
    'lastModifyTime']/text()").toString();
   try {
    Date date=dateFormat.parse(lastModifyTime);
    page.putField("lastModifyTime", date);
    xContentBuilder=xContentBuilder.field("lastModifyTime",
    page.getResultItems().get("lastModifyTime"));
```

```
    } catch(ParseException e) {
        if(lastModifyTime.equals("今天")) {
         page.putField("lastModifyTime", new Date());
         xContentBuilder=xContentBuilder.field("lastModifyTime",
         page.getResultItems().get("lastModifyTime"));
        } else {
           System.out.println("无法识别的编辑日期:"+lastModifyTime);
        }
    }
   List<String>tagList=page.getHtml().xpath("//sapn[@ class='taglist']
   /text()").all();
   if(tagList.size()>0) {
     page.putField("taglist", tagList);
     xContentBuilder=xContentBuilder.field("taglist",
     page.getResultItems().get("taglist"));
    }
 } catch(IOException e) {
     e.printStackTrace();
    }
 try {
   String source=xContentBuilder.endObject().string();
   IndexRequestBuilder indexRequestBuilder=client.prepareIndex("baike",
   "baike").setSource(source);
   indexRequestBuilder.execute().actionGet();
   System.out.println(page.getResultItems().get("title")+" Index Finish. At
   "+new Date());
   } catch(IOException e) {
   e.printStackTrace();
   }
 }
 private static SearchResponse excute(SearchRequestBuilder searchRequestBuilder,
 int page) {
   SearchResponse response=searchRequestBuilder.addField("url")
 .setFrom(page * SIZE_PER_PAGE).setSize(SIZE_PER_PAGE).execute().actionGet();
 return response;
 }
 @ Override
 public Site getSite() {
   return site;
 }
 static class MyPipeline implements Pipeline{
 @ Override
 public void process(ResultItems resultItems, Task task) {
   }
 }
 }
```

代码段 9.2 给出了 ESClientHelper 的实现方法，在这里主要完成的是 Elasticsearch 的 Client 实例化工作。

```
//代码段 9.2: ESClientHelper
import org.elasticsearch.client.Client;
import org.elasticsearch.client.transport.TransportClient;
import org.elasticsearch.common.transport.InetSocketTransportAddress;
public class ESClientHelper {
    private static Client client=new TransportClient()
                .addTransportAddress (new  InetSocketTransportAddress ( "
localhost", 9300));
    protected static Client getClient() {
        return client;
    }
}
```

## 9.2 基于 Python 的信息检索及 Web 端设计

本节介绍基于 Python 的信息检索及 Web 端设计。为保证系统顺利运行，建议在 Linux 环境下操作。

### 9.2.1 安装 Python 及 Django

(1) 执行下述命令完成 python 安装。

```
sudo apt-get install python
```

(2) 安装 python 开发环境，方便今后编译其他扩展库。

```
sudo apt-get install python-dev
```

(3) 安装 pip(pip 是 python 的一个安装和管理扩展库的工具)。

```
sudo apt-get install python-pip
```

(4) 安装 django。可通过 pip 安装 django，如图 9.2 所示。

```
gsh@spark-master:~$ sudo pip install Django==1.7.4
Downloading/unpacking Django==1.7.4
  Downloading Django-1.7.4-py2.py3-none-any.whl (7.4MB): 7.4MB downloaded
Installing collected packages: Django
Successfully installed Django
Cleaning up...
```

图 9.2 安装 django

接下来需要测试 django 是否安装成功且工作状态良好。在 Shell 中切换到另一个目录，输入 python 打开 python 的交互解释器。如果安装成功，应该可以导入 django 模块了，如图 9.3 所示。

```
gsh@spark-master:~$ python
Python 2.7.6 (default, Mar 22 2014, 22:59:56)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import django
>>> django.VERSION
(1, 7, 4, 'final', 0)
```

图 9.3 测试 django

Tips: django 项目是一个 python Web 开源框架，其主要功能有：

- 用于创建模型的对象关系映射。
- 为最终用户设计的完美管理界面。

### 9.2.2 安装 Elasticsearch 的 Python 插件

安装 Elasticsearch 的 python 插件，安装步骤及返回结果等如图 9.4 所示。

```
gsh@spark-master:~$ sudo pip install elasticsearch
[sudo] password for gsh:
Downloading/unpacking elasticsearch
  Downloading elasticsearch-1.4.0-py2.py3-none-any.whl (56kB): 56kB downloaded
Downloading/unpacking urllib3>=1.8,<2.0 (from elasticsearch)
  Downloading urllib3-1.10.1-py2-none-any.whl (76kB): 76kB downloaded
Installing collected packages: elasticsearch, urllib3
  Found existing installation: urllib3 1.7.1
    Uninstalling urllib3:
      Successfully uninstalled urllib3
Successfully installed elasticsearch urllib3
Cleaning up...
gsh@spark-master:~$ _
```

图 9.4 安装 Elasticsearch 插件

创建一个目录，并在其中初始化 django 项目，可按如下命令创建名为 demo 的项目。

```
django-admin.py startproject demo
```

接下来创建一个名为 search 的应用(执行命令 python manage.py startapp search)，目录结构如图 9.5 所示。

在项目的 settings.py 文件中的 INSTALLED_APPS 中，加之创建的 search 应用，其内容如图 9.6 所示。

```
.
├── demo
│   ├── __init__.py
│   ├── __init__.pyc
│   ├── settings.py
│   ├── settings.pyc
│   ├── urls.py
│   └── wsgi.py
├── manage.py
└── search
    ├── admin.py
    ├── __init__.py
    ├── migrations
    │   └── __init__.py
    ├── models.py
    ├── tests.py
    └── views.py

3 directories, 13 files
```

图 9.5 Demo 项目目录结构

```
INSTALLED_APPS = (
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'search',
)
```

图 9.6 INSTALLED_APPS

在项目 settings.py 文件中的 MIDDLEWARE_CLASSES 部分加入需要的中间件，如图 9.7 所示。

```
MIDDLEWARE_CLASSES = (
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.auth.middleware.SessionAuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
    'django.middleware.cache.FetchFromCacheMiddleware',
    'django.middleware.cache.UpdateCacheMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.http.ConditionalGetMiddleware',
    'django.middleware.gzip.GZipMiddleware',
)
```

图 9.7 MIDDLEWARE_CLASSES

再在项目 settings.py 文件中加入 cache 配置，选择基于内存的缓存，如图 9.8 所示。

```
CACHE_BACKEND = 'locmem:///'
```

图 9.8 cache 配置

> Tips：在 django 中与工程全局相关的设置都需要在配置文件 settings.py 中添加。为使 django 识别开发者添加的应用模块，在 settings.py 文件的 INSTALLED_APPS 部分中需要定义 django 工程加载的应用列表。为了激活中间件组件，要把它们添加到 MIDDLEWARE_CLASSES 列表中。

### 9.2.3 Web 页面设计

在 settings.py 文件中配置关于页面模板的路径，代码如下所示。

```
TEMPLATE_DIRS=(os.path.join(os.path.dirname(__file__), '../templates'), )
```

接下来创建两个 View（一个是首页 index.html，一个是结果页 result.html）。打开 views.py，并在 view.py 中插入如代码段 9.3 所示的内容。

```
//代码段 9.3: 在 view.py 中完成对上述两个 View 的配置
from django.shortcuts import render_to_response
from elasticsearch import Elasticsearch
def home(request):
    return render_to_response('views/index.html')
def search(request):
    query=request.GET.get('query')
    es=Elasticsearch()
    res=es.search(
        index='baike',
        doc_type='baike',
        body={
```

```
            'query': {
                'query_string': {
                    'default_field': 'content',
                    'query': query
                }
            },
            'highlight': {
                'fields': {
                    'content': {}
                }
            }
        }
    )
    result=[]
    for source in res['hits']['hits']:
        result.append(source['_source']['content'])
    return render_to_response('views/result.html',{'query': query,
        'took': res['took'], 'total': res['hits']['total'], 'result': result})
```

接下来建立文件夹 templates，在其中创建 index. html，内容如代码段 9.4 所示。

```
//代码段 9.4: index.tml
<!DOCTYPE html>
<html>
<head lang="en">
    <meta charset="UTF-8">
    <title>Hello</title>
</head>
<body>
<form action="/search/">
    <label for="query">Query</label>
    <input type="text" name="query" id="query">
    <input type="submit">
</form>
</body>
</html>
```

在文件夹 templates 中创建 result. html，内容如代码段 9.5 所示。

```
//代码段 9.5: result.html
<!DOCTYPE html>
<html>
<head lang="en">
    <meta charset="UTF-8">
```

```
    <title>Result</title>
</head>
<body>
<form action="/search/">
    <label for="query">Query</label>
    <input type="text" name="query" id="query" value="{{ query }}">
    <input type="submit">
</form>
<hr>
Total:{{ total }}
<ul>
{%for item in result %}
    <li>{{ item }}</li>
    <hr>
{%endfor %}
</ul>
</body>
</html>
```

之后，在 urls. py 中建立两个 URL，目的是将上述两个视图和相应的 URL 对应起来，如图 9.9 所示。

```
from django.conf.urls import patterns, include, url
from django.contrib import admin
from search.views import *
urlpatterns = patterns('',
    url(r'^admin/', include(admin.site.urls)),
    url('^$', home),
    url('^search/$', search),
)
```

图 9.9 建立 URL 和视图间的对应关系

启动 django 内置的 Web 服务器，如图 9.10 所示。从图 9.10 中可看到在图 9.7 和图 9.8 中设置的缓存策略产生效果了。

```
gsh@spark-master:~/demo/demo$ python manage.py runserver 0.0.0.0:5656
Performing system checks...

System check identified no issues (0 silenced).

You have unapplied migrations; your app may not work properly until they are applied.
Run 'python manage.py migrate' to apply them.

February 10, 2015 - 15:15:19
Django version 1.7.4, using settings 'demo.settings'
Starting development server at http://0.0.0.0:5656/
Quit the server with CONTROL-C.
[10/Feb/2015 15:15:32] "GET /search/?query=%E4%B8%AD%E5%9B%BD HTTP/1.1" 200 175041
[10/Feb/2015 15:16:01] "GET /search/?query=%E4%B8%AD%E5%9B%BD HTTP/1.1" 304 0
```

图 9.10 启动 django 内置的 Web 服务器

之后，就能在浏览器中输入 http://localhost:5656 看到首页了，如图 9.11 和图 9.12 所示。

Hello

Query

Enter Query

Submit

图 9.11 index.html 页面

Query:中国 Total:24

**成龙**

成龙，1954年4月7日生于香港中西区，祖籍安徽芜湖，[1] 国家一级演员，大中华区影坛和国际功夫影星。[2]
1960年进入中国戏剧学校学习戏曲，以武师身份进身电影圈。1970年自戏校毕业，因为自幼在影片中跑过龙套，所以
片”概念的电影《红番区》在美国公映后大受欢迎，使其成功打入好莱坞。[4] 1997年香港回归，江泽民在宴会上敬
选为代表“中国大人物”之一，仅次于毛泽东、邓小平，高居第3名。[6] 2012年8月24日 《纽约时报》评选出史上2
龙再次当选为香港演艺人协会会长。[8]
成龙与周润发、周星驰、并称“双周一成”。[9] 他擅长功夫片，电影的谐趣风格及袁和平设计的活泼灵巧兼具杂耍
亿元。[10]
儿童时代
童年时的成龙[11]1954年成龙在香港出生，祖籍安徽省芜湖市鸠江区沈巷镇房桥村，成龙的父母亲最初在法国领事馆
法国人、美国人，所以经常跟外国小孩子打。这种情况一直到他小学一年级，但由于常打架、闹事，所以无法升级。
的明星，成龙崇拜他们，一心想上山学艺。[12]
1961年，成龙的父亲带着成龙来到尖沙咀的美丽都大厦，拜访京剧武生于占元师傅，他正是成龙崇拜的武侠女星于素
成龙便成为这儿的一员，与洪金宝（元龙）、元奎、元华、元彬、元德、元彪成为七小福。于占元师傅的教育方式基
了，最初的那段日子，成龙常常在晚上暗自哭泣。[13]
练武时期

图 9.12 result.html 页面

# 9.3 基于 Logstash 的日志处理

本节介绍基于 Logstash 的日志处理，这里使用 Logstash 来监听 Nginx 日志文件。同样，为了系统稳定运行，建议在 Linux 环境下操作。

## 9.3.1 安装和配置 Nginx

安装 Nginx，如图 9.13 所示。

```
gsh@spark-master:~/tenginx-new/sbin$ sudo apt-get install nginx
正在读取软件包列表... 完成
正在分析软件包的依赖关系树
正在读取状态信息... 完成
将会安装下列额外的软件包:
  fontconfig-config fonts-dejavu-core libfontconfig1 libgd3 libjbig0
  libjpeg-turbo8 libjpeg8 libtiff5 libvpx1 libxpm4 nginx-common nginx-core
建议安装的软件包:
  libgd-tools fcgiwrap nginx-doc
下列【新】软件包将被安装:
  fontconfig-config fonts-dejavu-core libfontconfig1 libgd3 libjbig0
  libjpeg-turbo8 libjpeg8 libtiff5 libvpx1 libxpm4 nginx nginx-common
  nginx-core
升级了 0 个软件包，新安装了 13 个软件包，要卸载 0 个软件包，有 92 个软件包未被升级。
需要下载 2,558 kB 的软件包。
解压缩后会消耗掉 8,558 kB 的额外空间。
您希望继续执行吗? [Y/n]
```

图 9.13 安装 Nginx

Tips：Nginx 是一款轻量级的 Web 服务器/反向代理服务器及电子邮件(IMAP/POP3)代理服务器。

之后，在/etc/nginx/sites-available/default 中配置相关参数，参见代码段 9.6，其目的是将请求转发至 django 服务器中。

```
//代码段 9.6: 转发请求
location / {
            proxy_pass  http://localhost:5656
}
```

下面可以测试一下日志。启动 django 的 server，然后再启动 Nginx，使用 Nginx 的 start 命令，之后可在/var/log/nginx 目录下看到两个日志文件(access.log 和 error.log)。下面将使用 Logstash 来监听这个日志文件。

### 9.3.2 设计面向日志文件的 Pattern

创建一个模式(pattern)来匹配 Nginx 的日志。在 Logstash 的 pattern 目录新建一个文件并输入下面的内容，以创建针对 Nginx 的 access.log 的模式，见代码段 9.7。

```
//代码段 9.7: 用于匹配 Nginx 日志的 pattern
NGUSERNAME [a-zA-Z\.\@\-\+_%]+
NGUSER %{NGUSERNAME}
NGINXACCESS %{IPORHOST:clientip} %{NGUSER:ident} %{NGUSER:auth} [%{HTTPDATE:
timestamp}] "% { WORD: verb } % { URIPATHPARAM: request } HTTP/% { NUMBER:
httpversion}" % { NUMBER: response } (?:% { NUMBER: bytes } | -) (?:" (?:% { URI:
referrer}|-)"|%{QS:referrer}) %{QS:agent}
```

### 9.3.3 在 Logstash 中进行相关配置

在 Logstash 的配置文件(注：这里是在 Logstash 的 bin 文件夹下的 conf.conf 配置文件中)中进行相关的配置，见代码段 9.8。

```
//代码段 9.8: Logstash 的配置文件,注意 host=>"10.81.10.106"要进行有针对性的修改,
如 localhost
input {
    file {
      path=>["/var/log/nginx/access.log"]
    }
}
filter {
```

```
  grok {
      match=>{ "message"=>"%{NGINXACCESS}" }
  }
  kv {
      source=>"request"
      field_split=>"&?"
  }
  urldecode{
      field=>"query"
  }
}
output {
  stdout { codec=>rubydebug }
  elasticsearch {
      host=>"10.81.10.106"
    protocol=>"http"
    }
}
```

启动 Elasticsearch。使用命令./logstash -f conf.conf 启动 Logstash，然后就可以利用 Elasticsearch 的 Head 工具看到 Nginx 的日志信息已经进入到 Elasticsearch 的索引中了，如图 9.14 所示。


```
{
    "_index": "logstash-2015.02.21",
    "_type": "logs",
    "_id": "AUur_kHLEqP_DiGNx2ZT",
    "_version": ,
    "_score": ,
      "_source": {
        "message": "10.81.10.106 - - [21/Feb/2015:19:55:41 +0800] "GET /search/?
        query=%E4%B8%AD%E5%9B%BD HTTP/1.1" 200 175039 "-" "Mozilla/5.0
        (Macintosh; Intel Mac OS X 10_9_5) AppleWebKit/537.36 (KHTML, like Gecko)
        Chrome/40.0.2214.111 Safari/537.36"",
        "@version": "1",
        "@timestamp": "2015-02-21T11:55:41.928Z",
        "host": "spark-master",
        "path": "/var/log/nginx/access.log",
        "clientip": "10.81.10.106",
        "ident": "-",
        "auth": "-",
        "timestamp": "21/Feb/2015:19:55:41 +0800",
        "verb": "GET",
        "request": "/search/?query=%E4%B8%AD%E5%9B%BD",
        "httpversion": "1.1",
        "response": "200",
        "bytes": "175039",
        "agent": ""Mozilla/5.0 (Macintosh; Intel Mac OS X 10_9_5) AppleWebKit/537.36
        (KHTML, like Gecko) Chrome/40.0.2214.111 Safari/537.36"",
        "query": "中国"
    }
}
```


图 9.14　Nginx 日志信息已被索引

## 9.4 基于 Kibana 的日志分析结果可视化设计与实现

开启 Kibana4，如图 9.15 所示。

```
Last login: Sat Feb 21 19:08:32 on ttys001
localhost:~ gaoshen$ /Users/gaoshen/Documents/Tools/kibana-4.0.0-beta3/bin/kiban
a ; exit;
The Kibana Backend is starting up... be patient
{"@timestamp":"2015-02-21T20:11:13+08:00","level":"INFO","name":"Kibana","messag
e":"Kibana server started on tcp://0.0.0.0:5601 in production mode."}
```

图 9.15 启动 Kibana

添加 Logstash 用于存放日志的 index 并将其设置为主 index，如图 9.16 所示，单击 Add New 按钮，出现如图 9.17 所示的页面。

图 9.16 添加索引

Configure an index pattern

In order to use Kibana you must configure at least one index pattern. Index patterns are used to identify the Elasticsearch index to run search and analytics against. They are also used to configure fields.

Index contains time-based events
Use event times to create index names
Index name or pattern
Patterns allow you to define dynamic index names using * as a wildcard. Example: logstash-*
logstash-*
Time-field name refresh fields
@timestamp
Create

图 9.17 设置用于可视化的 index

### 9.4.1 图表 1：状态码走势分析

选择创建相应类型的图表，具体步骤如图 9.18 至图 9.23 所示，以此来分析网页状态码走势。

Discover Visualize Dashboard Settings
Create a visualization Step 1
From a new search
From a saved search
From an existing visualization

图 9.18 创建可视化图表

I want to create a　Step 2

Area chart
Data table
Line chart
Metric
Pie chart
Tile map
Vertical bar chart

图 9.19　选择图表类型

metrics
Y-Axis
Aggregation
Count
Advanced
+ Add Aggregation
buckets
Select buckets type
X-Axis
Split Lines
Split Chart
cancel
view options

图 9.20　设定图表的 X 和 Y 轴数据来源

图 9.21　Aggregation：Data histogram

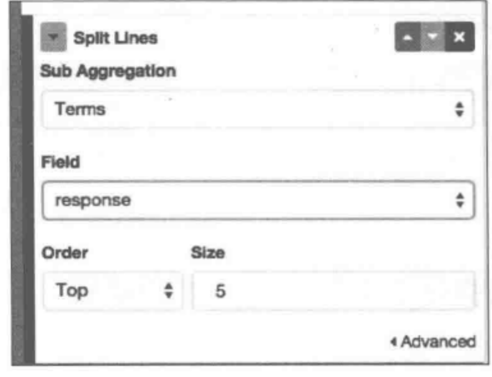


图 9.22　添加子 Aggregation 统计状态码

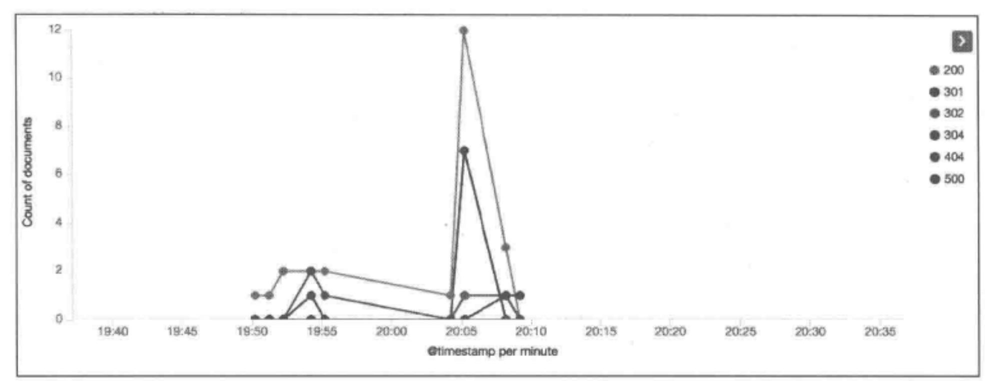


图 9.23 状态码走势分析

## 9.4.2 图表 2：查询词分析

下面给出基于 Kibana 的分析用户给出的查询词的方法，具体步骤参见图 9.24 至图 9.26。

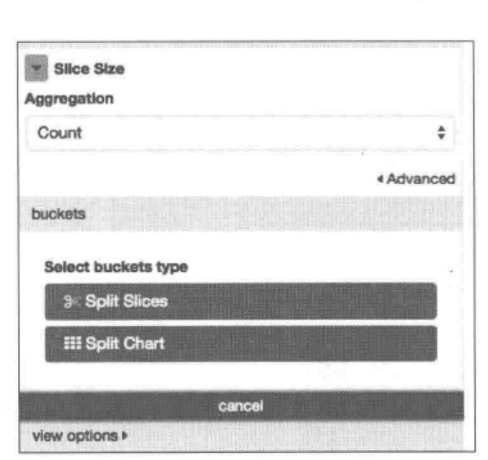


图 9.24 选择 Aggregation：Count

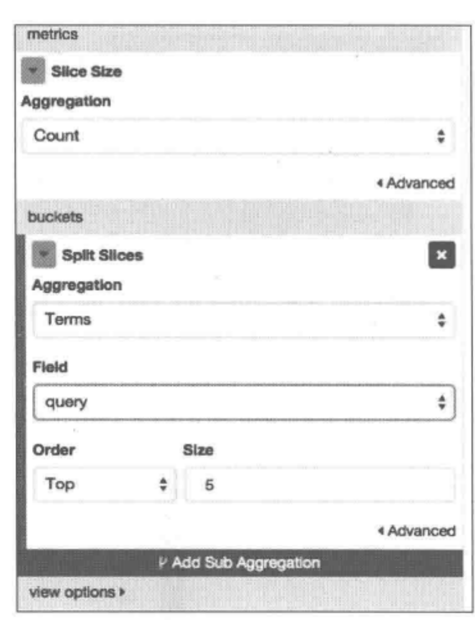


图 9.25 选择统计的字段及其排序、返回结果集的大小等

## 9.4.3 图表 3：分析各状态码随时间的变迁情况

下面给出创建柱状图并分析每个时间段内的各个网站状态码情况的方法，具体步骤参见图 9.27 和图 9.28。

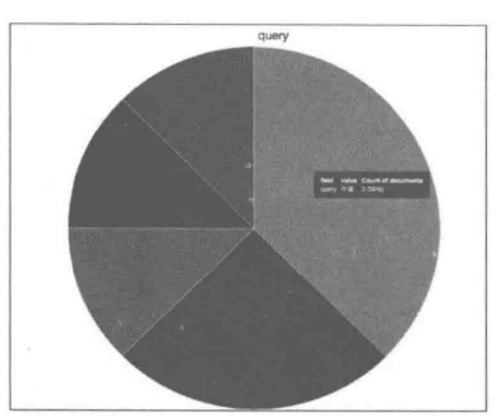


图 9.26　保存的统计结果

图 9.27　设定图表的 X 轴、Y 轴的统计字段、时间间隔、排序、返回结果集的大小等信息

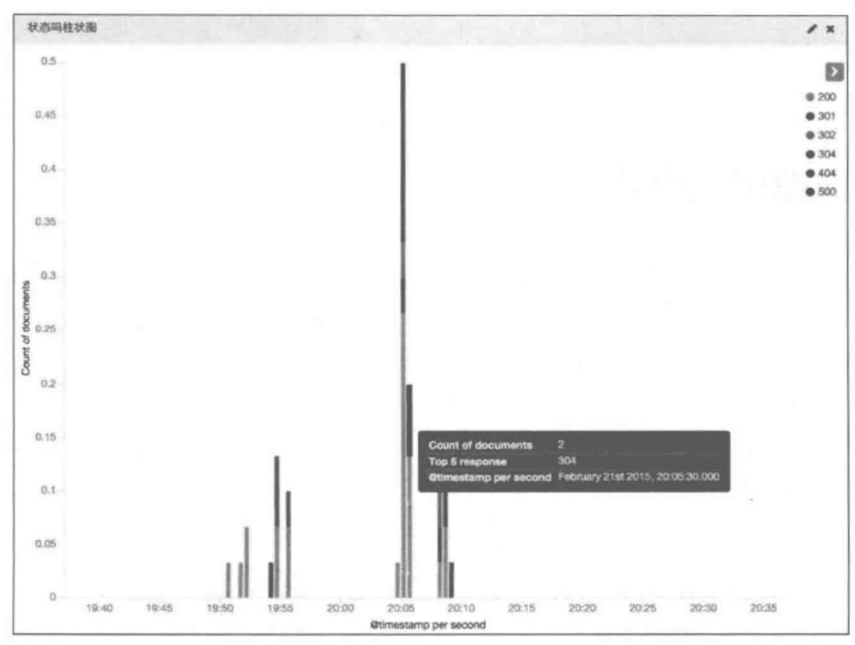


图 9.28　统计结果

### 9.4.4 集成上述图表

下面给出创建面板来放置上述图表结果的方法。在图 9.29 中，单击加号添加图表。将上述建立好的各个图表添加到 Dashboard 中，如图 9.30 所示。

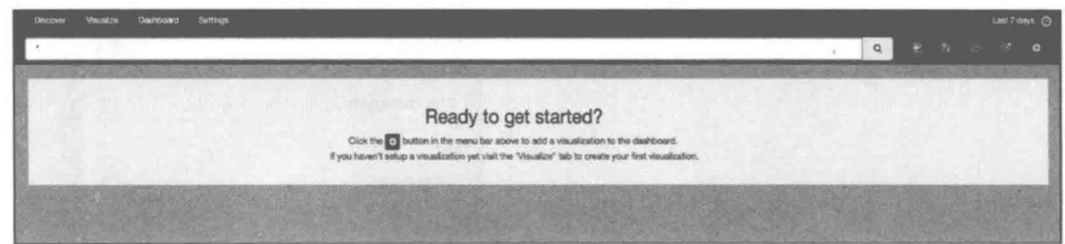


图 9.29 Dashboard 首页

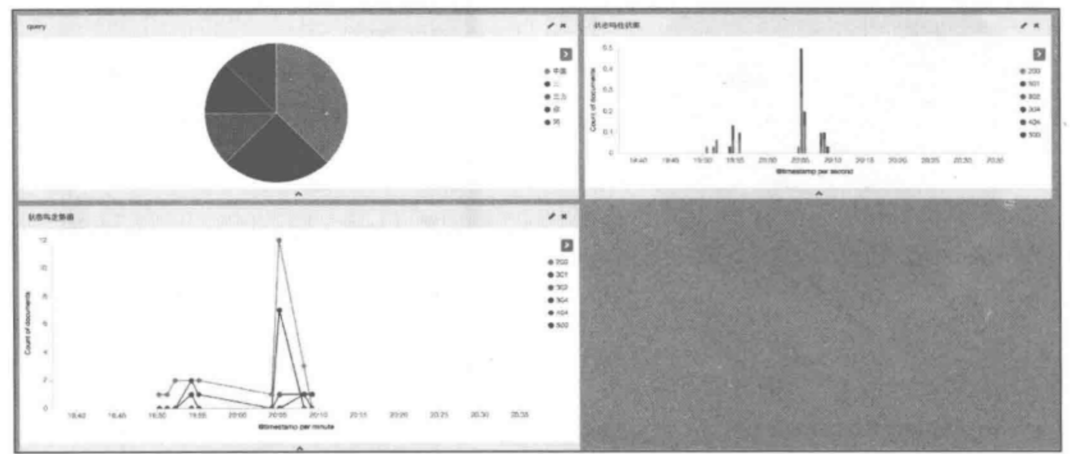


图 9.30 Dashboard

## 9.5 扩展知识与阅读

本章给出在一个大型网站中可能用到的网络信息搜索系统、数据分布与存储、基于 Logstash 和 Kibana 的日志及监控系统的实现。其实，要用 Java 构建一个大流量且有着复杂处理流程的网站，中间件技术也是必须要用到的，没有中间件就无法开发一个高质量的网站。一般来说，中间件包括远程过程调用和对象访问中间件、消息中间件、数据访问中间件。文献[曾宪杰，2014]给出了有关大型网站系统与 Java 中间件实践相关技术。而有关中间件的理论背景，可参阅[Andrew，2007]。除此之外，软件负载中心与集中配置管理也是需要考虑的问题。另外，文献[Hetland，2014]给出了有关 Python 的相关背景知识与用法。

## 9.6 本章小结

作为全书总结，本书给出一个基于 ELK(Elasticsearch+Logstash+Kibana)的网络信息检索、日志分析及可视化的工程实例。其中，信息采集是基于 webmagic 进行的，在采集了相关信息后，放到 Elasticsearch 的索引中，而 Web 检索页面是基于 Python 完成的(当然 Web 检索页面也可以基于 Java Web 完成)；对日志的处理采用了 Logstash，可视化则是基于 Kibana4 实现的。通过本章的学习，目的是对本书前述各章内容的实际应用有一个总体认识，也便于读者尽快上手，开发出一款属于自己的大数据搜索、日志挖掘与可视化系统。

# 参考文献


1. [51CTO,2015] Logstash：日志分析整合利器,http://os. 51cto. com/art/201309/409787. htm.
2. [Andrew,2007] Andrew S. Tanenbaum,Maarten van Steen. 分布式系统原理与范型. 北京：清华大学出版社,2008.
3. [CSDN,2014] CSDN,一些国外优秀的 elasticsearch 使用案例,http://blog. csdn. net/july_2/article/details/24779533.
4. [CSDN,2015] CSDN,分布式搜索 Elasticsearch,http://blog. csdn. net/geloin/article/details/8908238.
5. [Db-engines,2015] DB-Engines Ranking,http://db-engines. com/en/ranking.
6. [Fu,2015] Fu Kailong,ElasticSearch 权威指南,http://fuxiaopang. gitbooks. io/learnelasticsearch/,2015.
7. [Maven,2014a] Maven,百度百科,http://baike. baidu. com/view/336103. htm? fr=aladdin.
8. [Maven,2014b] 项目构建工具 Maven,开源中国社区,http://www. oschina. net/p/maven.
9. [Michael,2011] Michael McCandless,Erik Hatcher,Otis Gospodnetic. Lucene in Action 实战(第 2 版). 牛长流,等译. 北京：人民邮电出版社,2011.
10. [Open,2014a] Open,分布式搜索 elasticsearch 配置文件详解,http://www. open-open. com/lib/view/open1397003561934. html.
11. [Open, 2014b] Elasticsearch 学习入门, http://www. open-open. com/doc/view/b6a3a83c2a494acea97c7e6045994c4e.
12. [Rafa,2015] Rafal K. ,Marek R. ElasticSearch. 可扩展的开源弹性搜索解决方案. 北京：电子工业出版社,2015.
13. [Ricardo,2012] Ricardo Baeza-Yates,Berthier Ribeiro-Neto. 现代信息检索. 黄萱菁,等译. 北京：机械工业出版社,2012.
14. [Steven,2013] Steven John Metsker,William C. Wake. Java 设计模式. 张逸,等译. 北京：电子工业出版社,2013.
15. [Turnbull,2015] James Turnbull. *The Logstash Book：Log Management Made Easy*,http:www. logstas hbook. com/,2015.
16. [百度,2014] 百度,ElasticSearch,http://baike. baidu. com/view/8005387. htm?fr=aladdin.
17. [高凯,2010]高凯,郭立炜,许云峰. 网络信息检索技术与搜索引擎系统开发. 北京：科学出版社,2010.
18. [高凯,2014] 高凯,仇晶,张晓明,王伟,张华平. 信息检索与智能处理. 北京：国防工业出版社,2014.
19. [韩陆,2014] 韩陆. Java RESTful Web Service 实战. 北京：机械工业出版社,2014.
20. [黄健宏,2014] 黄健宏. Redis 设计与实现. 北京：机械工业出版社,2014.
21. [Hetland,2014] Hetland,等. Python 基础教程. 北京：人民邮电出版社,2014.
22. [李志义,2015] 李志义. 网站信息组织优化：基于网络日志的用户行为分析. 北京：电子工业出版社,2015.
23. [李子骅,2013] 李子骅. Redis 入门指南. 北京：人民邮电出版社,2013.
24. [李刚,2014] 李刚. 疯狂 Ajax 讲义. 北京：电子工业出版社,2014.
25. [罗刚,2014]罗刚. 解密搜索引擎技术实战. 北京：电子工业出版社,2014.

26. [饶琛琳,2015]饶琛琳. Kibana 中文指南,http://kibana. logstash. es/,2015.

27. [王继民,2014]王继民. Web 用户查询日志挖掘与应用. 北京：知识产权出版社,2014.

28. [吴军,2013] 吴军. 数学之美. 北京：人民邮电出版社,2013.

29. [吴晓刚,2015]吴晓刚. 10TB 级日志的秒级搜索,携程网,2015.

30. [许晓斌,2011] 许晓斌. Maven 实战. 北京：机械工业出版社,2011.

31. [杨传辉,2014] 杨传辉. 大规模分布式存储系统原理解析与架构实战. 北京：机械工业出版社,2014.

32. [余晟,2012] 余晟. 正则指引. 北京：电子工业出版社,2012.

33. [云端分布式搜索技术,2013] [云端分布式搜索技术,elasticsearch 插件大全,http://www. searchtech. pro/elasticsearch-plugins,2013.

34. [曾宪杰,2014] 曾宪杰. 大型网站系统与 Java 中间件实践. 北京：电子工业出版社,2014.

35. [张华平,2014] 张华平,高凯,黄河燕,赵燕平. 大数据搜索与挖掘. 北京：科学出版社,2014.

36. [周宁,2005] 周宁,张玉峰,张李义. 信息可视化与知识检索. 北京：科学出版社,2005.

37. [子猴博客,2014] 子猴博客,elasticsearch. yml 配置说明,http://www. zihou. me/html/2014/01/17/9061. html.