目录

Elasticsearch 权威指南

1. GitBook

Download PDF: Elasticsearch:权威指南

2. 声明

本书整理自官方文档 《*Elasticsearch: 权威指南*》

该文件修订时间： 2021-03-07

1. 序言

我仍然清晰地记得那个日子，我发布了这个开源项目第一个版本并在 IRC 聊天室创建一个频道， 在那个最紧张的时刻，独自一人，急切地希望和盼望着第一个用户的到来。

第一个跳进 IRC 频道的用户就是 Clint（克林顿），当时我欣喜若狂。好吧…直到我发现 Clint 实际上是 Perl 用户啦， 而且还是跟死亡讣告网站打交道。 我记得（当时）问自己为什么他不是来自于更“主流”的社区，像 Ruby 或 Python，亦或是一个稍微好点的使用案例。

后来发生的一切都证明，我真是大错特错！Clint 最终对 Elasticsearch 的成功起到了重要作用。 他是第一个将 Elasticsearch 投入生产环境的人（还是 0.4 的版本！），初期与 Clint 的交流和沟通对于将 Elasticsearch 塑造成今天的样子非常关键。 对于什么是简单，Clint 有独特的见解并且他很少出错，这对 Elasticsearch 从管理、API 设计到日常使用等各个方面的易用性产生了深远的影响。 所以公司成立不久，我们想也没想立即就联系 Clint ，询问他是否愿意加入我们。

公司成立后，我们做的第一件事就是提供公开培训。很难表达我们当时有多么紧张和担心是否真的有人会报名。

但我们错了。

培训到现在依然很成功，很多主要城市都还有大量的人等待参加。参加培训的成员之中，有一个叫 Zach 年轻小伙吸引了我们注意。 我们知道他写过很多关于 Elasticsearch 的博客（并暗自嫉妒他能够用非常简洁的方式来阐述复杂概念的能力），他还编写了一个 PHP 的客户端。 然后我们发现 Zach 他还是自掏腰包来参加我们的培训！你真的不能要求更多，于是我们找到 Zach，问他是否愿意加入我们的公司。

Clint 和 Zach 是 Elasticsearch 能否成功的关键。他们是完美的解说家，从简单的上层应用到复杂的（Apache Lucene）底层逻辑。 在 Elastic 这里我们非常珍惜这种独特技能。 Clint 还负责 Elasticsearch Perl 客户端，而 Zach 则负责 PHP，都是精彩的代码。

最后，两位在 Elasticsearch 项目每天的日常事务中也扮演着重要的角色。Elasticsearch 如此受欢迎的主要原因之一， 就是它拥有与用户沟通产生共鸣的能力，Clint 和 Zach 都是这个集体的一份子，这让一切成为可能。

Shay Banon

1. 引言

这个世界已然被数据淹没。多年来，我们系统间流转和产生的大量数据已让我们不知所措。 现有的技术都集中在如何解决数据仓库存储以及如何结构化这些数据。 这些看上去都挺美好，直到你实际需要基于这些数据实时做决策分析的时候才发现根本不是那么一回事。

Elasticsearch 是一个分布式、可扩展、实时的搜索与数据分析引擎。 它能从项目一开始就赋予你的数据以搜索、分析和探索的能力， 这是通常没有预料到的。它存在还因为原始数据如果只是躺在磁盘里面根本就毫无用处。

无论你是需要全文搜索，还是结构化数据的实时统计，或者两者结合，这本指南都能帮助你了解其中最基本的概念， 从最基本的操作开始学习 Elasticsearch。之后，我们还会逐渐开始探索更加高级的搜索技术，不断提升搜索体验来满足你的需求。

Elasticsearch 不仅仅只是全文搜索，我们还将介绍结构化搜索、数据分析、复杂的人类语言处理、地理位置和对象间关联关系等。 我们还将探讨为了充分利用 Elasticsearch 的水平伸缩性，应当如何建立数据模型，以及在生产环境中如何配置和监控你的集群。

1. 谁应该读这这本书

这本书是写给任何想要把他们的数据拿来干活做点事情的人。不管你是从头构建一个新项目，还是为了给已有的系统改造换血， Elasticsearch 都能够帮助你解决现有问题和开发新的功能，有些可能是你之前没有想到的功能。

这本书既适合初学者也适合有经验的用户。我们希望你有一定的编程基础，虽然不是必须的，但有用过 SQL 和关系数据库会更佳。 我们会从原理解释和基本概念出发，帮助新手在复杂的搜索世界里打下稳固的知识基础。

具有搜索背景的读者也会受益于这本书。有经验的用户将懂得其所熟悉搜索的概念在 Elasticsearch 是如何对应和具体实现的。 即使是高级用户，前面几个章节所包含的信息也是非常有用的。

最后，也许你是一名 DevOps，其他部门一直尽可能快的往 Elasticsearch 里面灌数据，而你是那个负责防止 Elasticsearch 服务器起火的消防员。 只要用户在规则内行事，Elasticsearch 集群扩容相当轻松。不过你需要知道如何在进入生产环境前搭建一个稳定的集群， 还能要在凌晨三点钟能识别出警告信号，以防止灾难发生。 前面几章你可能不太感兴趣，但这本书的最后一部分是非常重要的，包含所有你需要知道的用以避免系统崩溃的知识。

1. 为什么要我们写这本书

我们写这本书，因为 Elasticsearch 需要更好的阐述。 现有的参考文档是优秀的 — 前提是你知道你在寻找什么。 它假定你已经熟悉信息检索、分布式系统原理、Query DSL 和许多其他相关的概念。

这本书没有这样的假设。它的目的是写一本即便是什么都不懂的初学者（不管是对于搜索还是对于分布式系统）也能拿起它简单看完几章， 就能开始搭建一个原型。

我们采取一种基于问题求解的方式：这是一个问题，我该怎么解决？ 如何对候选方案进行权衡取舍？我们从基础知识开始， 循序渐进，每一章都建立在前一章之上，同时提供必要的实用案例和理论解释。

现有的参考文档解决了 如何 使用这些功能，我们希望这本书解决的是 **为什么** 和 **什么时候** 使用这些功能。

1. 如何读这本书

Elasticsearch 做了很多努力和尝试来让复杂的事情变得简单， 很大程度上来说 Elasticsearch 的成功来源于此。 换句话说，搜索以及分布式系统是非常复杂的， 不过为了充分利用 Elasticsearch，迟早你也需要掌握它们。

恩，是有点复杂，但不是魔法。我们倾向于认为复杂系统如同神奇的黑盒子，能响应外部的咒语，但是通常里面的工作逻辑很简单。 理解了这些逻辑过程你就能驱散魔法， 理解内在能够让你更加明确和清晰，而不是寄托于黑盒子做你想要做的。

这本权威指南不仅会帮助你学习 Elasticsearch，而且希望能够带你接触一些更深入、更有趣的话题，如 集群内的原理 、 分布式文档存储 、 执行分布式检索 和 分片内部原理 ，这些虽然不是必要的阅读却能让你深入理解其内在机制。

本书的第一部分应该按章节顺序阅读，因为每一章建立在上一章的基础上（尽管你也可以浏览刚才提到的章节）。 后续各章节如 近似匹配 和 部分匹配 相对独立，你可以按需选择性参阅。

1. 本书导航

这本书分为七个部分：

- 章节 你知道的, 为了搜索… 到 分片内部原理 主要是介绍 Elasticsearch。介绍了 Elasticsearch 的数据输入输出以及 Elasticsearch 如何处理你的文档数据。如何进行基本的搜索操作和管理你的索引。 本章结束你将学会如何将 Elasticsearch 集成到你的应用程序中。 章节：集群内的原理、分布式文档存储、 执行分布式检索 和 分片内部原理 为附加章节，目的是让你了解分布式处理的过程，不是必读的。
- 章节 结构化搜索 到 控制相关度 带你深入了解搜索，如何借助一些更高级的特性，如邻近词（word proximity）和 部分匹配（partial matching）来索引和查询你的数据。你将了解相关度评分是如何工作的以及如何控制它来确保第一页总是返回最佳的搜索结果。
- 章节 开始处理各种语言 到 拼写错误 解决如何有效使用分析器和查询来处理人类语言的棘手问题。 我们会从一次简单的语言分析下手，然后逐步深入，如字母表和排序，还会涉及到词干提取、停用词、同义词和模糊匹配。
- 章节 高阶概念 到 Doc Values and Fielddata 讨论聚合（aggregations）和分析，对你的数据进行摘要化和分组来呈现总体趋势。
- 章节 地理坐标点 到 地理形状 介绍 Elasticsearch 支持的两种地理位置检索方式：经纬坐标点和复杂的地理形状（geo-shapes）。
- 章节 关联关系处理 到 扩容设计 谈到了为了高效使用 Elasticsearch，应当如何为你的数据建立模型。 在搜索引擎里表达实体间的关系可能不是那么容易，因为它不是用来设计做这个的。这些章节还会阐述如何设计索引来匹配你系统中的数据流。
- 最后，章节 监控 到 部署后 将讨论生产环境上线的重要配置、监控点以及如何诊断以避免出现问题。

1. 本书协议约定

以下是本书中使用的印刷规范：

斜体
　　表示重点、新的术语或概念。
等宽字体
　　用于程序列表以及在段落中引用变量或程序元素如：**函数名称**、**数据库**、**数据类型**、**环境变量**、**语句**和**关键字**。

1. 基础入门

Elasticsearch 是一个实时的分布式搜索分析引擎，它能让你以前所未有的速度和规模，去探索你的数据。 它被用作全文检索、结构化搜索、分析以及这三个功能的组合：

- Wikipedia 使用 Elasticsearch 提供带有高亮片段的全文搜索，还有 search-as-you-type 和 did-you-mean 的建议。
- 卫报 使用 Elasticsearch 将网络社交数据结合到访客日志中，为它的编辑们提供公众对于新文章的实时反馈。
- Stack Overflow 将地理位置查询融入全文检索中去，并且使用 more-like-this 接口去查找相关的问题和回答。
- GitHub 使用 Elasticsearch 对1300亿行代码进行查询。

Elasticsearch 不仅仅为巨头公司服务。它也帮助了很多初创公司，比如 Datadog 和 Klout，Elasticsearch 帮助他们将想法用原型实现，并转化为可扩展的解决方案。Elasticsearch 能运行在你的笔记本电脑上， 或者扩展到数百台服务器上来处理PB级数据。

Elasticsearch 中没有一个单独的组件是全新的或者是革命性的。全文搜索很久之前就已经可以做到了， 就像很早之前出现的分析系统和分布式数据库。 革命性的成果在于将这些单独的， 有用的组件融合到一个单一的、一致的、实时的应用中。对于初学者而言它的门槛相对较低， 而当你的技能提升或需求增加时，它也始终能满足你的需求。

如果你现在打开这本书，是因为你拥有数据。除非你准备使用它 做些什么 ，否则拥有这些数据将没有意义。

不幸的是，大部分数据库在从你的数据中提取可用知识时出乎意料的低效。 当然，你可以通过时间戳或精确值进行过滤， 但是它们能够全文检索、处理同义词、通过相关性给文档评分么？ 它们能从同样的数据中生成分析与聚合数据吗？ 最重要的是，它们能实时地做到上述操作，而不经过大型批处理的任务么？

这就是 Elasticsearch 脱颖而出的地方：Elasticsearch 鼓励你去探索与利用数据，而不是因为查询数据太困难， 就让它们烂在数据仓库里面。

Elasticsearch 将成为你最好的朋友。

1. 基础知识

Elasticsearch 是一个开源的搜索引擎，建立在一个全文搜索引擎库 Apache Lucene™ 基础之上。 Lucene 可以说是当下最先进、高性能、全功能的搜索引擎库—无论是开源还是私有。

但是 Lucene 仅仅只是一个库。为了充分发挥其功能，你需要使用 Java 并将 Lucene 直接集成到应用程序中。 更糟糕的是，您可能需要获得信息检索学位才能了解其工作原理。Lucene 非常 复杂。

Elasticsearch 也是使用 Java 编写的，它的内部使用 Lucene 做索引与搜索，但是它的目的是使全文检索变得简单， 通过隐藏 Lucene 的复杂性，取而代之的提供一套简单一致的 RESTful API。

然而，Elasticsearch 不仅仅是 Lucene，并且也不仅仅只是一个全文搜索引擎。 它可以被下面这样准确的形容：

- 一个分布式的实时文档存储，每个字段 可以被索引与搜索
- 一个分布式实时分析搜索引擎
- 能胜任上百个服务节点的扩展，并支持 PB 级别的结构化或者非结构化数据

Elasticsearch 将所有的功能打包成一个单独的服务，这样你可以通过程序与它提供的简单的 RESTful API 进行通信， 可以使用自己喜欢的编程语言充当 web 客户端，甚至可以使用命令行（去充当这个客户端）。

就 Elasticsearch 而言，起步很简单。对于初学者来说，它预设了一些适当的默认值，并隐藏了复杂的搜索理论知识。 它 开箱即用 。只需最少的理解，你很快就能具有生产力。

随着你知识的积累，你可以利用 Elasticsearch 更多的高级特性，它的整个引擎是可配置并且灵活的。 从众多高级特性中，挑选恰当去修饰的 Elasticsearch，使它能解决你本地遇到的问题。

你可以免费下载，使用，修改 Elasticsearch。它在 Apache 2 license 协议下发布的， 这是众多灵活的开源协议之一。 Elasticsearch 的源码被托管在 Github 上 github.com/elastic/elasticsearch。如果你想加入我们这个令人惊奇的 contributors 社区， 看这里 Contributing to Elasticsearch。

如果你对 Elasticsearch 有任何相关的问题，包括特定的特性(specific features)、语言客户端(language clients)、 插件(plugins)，可以在这里 discuss.elastic.co 加入讨论。

> 回忆时光
>
> > 许多年前，一个刚结婚的名叫 Shay Banon 的失业开发者，跟着他的妻子去了伦敦，他的妻子在那里学习厨师。 在寻找一个赚钱的工作的时候，为了给他的妻子做一个食谱搜索引擎，他开始使用 Lucene 的一个早期版本。
> >
> > 直接使用 Lucene 是很难的，因此 Shay 开始做一个抽象层，Java 开发者使用它可以很简单的给他们的程序添加搜索功能。 他发布了他的第一个开源项目 Compass。
> >
> > 后来 Shay 获得了一份工作，主要是高性能，分布式环境下的内存数据网格。这个对于高性能，实时，分布式搜索引擎的需求尤为突出， 他决定重写 Compass，把它变为一个独立的服务并取名 Elasticsearch。
> >
> > 第一个公开版本在2010年2月发布，从此以后，Elasticsearch 已经成为了 Github 上最活跃的项目之一，他拥有超过300名 contributors(目前736名 contributors)。 一家公司已经开始围绕 Elasticsearch 提供商业服务，并开发新的特性，但是，Elasticsearch 将永远开源并对所有人可用。
> >
> > 据说，Shay 的妻子还在等着她的食谱搜索引擎…

1. 和Elasticsearch交互

和 Elasticsearch 的交互方式取决于你是否使用 Java。

1.1. Java Api

如果你正在使用 Java，在代码中你可以使用 Elasticsearch 内置的两个客户端：

节点客户端（Node client）

节点客户端作为一个非数据节点加入到本地集群中。换句话说，它本身不保存任何数据，但是它知道数据在集群中的哪个节点中，并且可以把请求转发到正确的节点。

传输客户端（Transport client）

轻量级的传输客户端可以将请求发送到远程集群。它本身不加入集群，但是它可以将请求转发到集群中的一个节点上。

两个 Java 客户端都是通过 9300 端口并使用 Elasticsearch 的原生 传输 协议和集群交互。集群中的节点通过端口 9300 彼此通信。如果这个端口没有打开，节点将无法形成一个集群。

> 建议
>
> > Java 客户端作为节点必须和 Elasticsearch 有相同的 主要 版本；否则，它们之间将无法互相理解。

更多的 Java 客户端信息可以在 Elasticsearch Clients 中找到。

1.2. Restful API with Json over HTTP

所有其他语言可以使用 RESTful API 通过端口 9200 和 Elasticsearch 进行通信，你可以用你最喜爱的 web 客户端访问 Elasticsearch 。 事实上，正如你所看到的，你甚至可以使用 curl 命令来和 Elasticsearch 交互。

> 注意
>
> > Elasticsearch 为以下语言提供了官方客户端--Groovy、JavaScript、.NET、 PHP、 Perl、 Python 和 Ruby—还有很多社区提供的客户端和插件， 所有这些都可以在 Elasticsearch Clients 中找到。

一个 Elasticsearch 请求和任何 HTTP 请求一样由若干相同的部件组成：

```
curl -X<VERB> '<PROTOCOL>://<HOST>:<PORT>/<PATH>?<QUERY_STRING>' -H 'Content-T
```

被 < > 标记的部件：

变量	备注
VERB	适当的 HTTP 方法 或 谓词 : GET、 POST、 PUT、 HEAD 或者 DELETE。
PROTOCOL	http 或者 https（如果你在 Elasticsearch 前面有一个 https 代理）
HOST	Elasticsearch 集群中任意节点的主机名，或者用 localhost 代表本地机器上的节点。
PORT	运行 Elasticsearch HTTP 服务的端口号，默认是 9200 。
PATH	API 的终端路径（例如 _count 将返回集群中文档数量）。Path 可能包含多个组件，例如：_cluster/stats 和 _nodes/stats/jvm 。
QUERY_STRING	任意可选的查询字符串参数 (例如 ?pretty 将格式化地输出 JSON 返回值，使其更容易阅读)
BODY	一个 JSON 格式的请求体 (如果请求需要的话)

例如，计算集群中文档的数量，我们可以用这个:

```
curl -XGET 'http://localhost:9200/_count?pretty' -d '
{
    "query": {
        "match_all": {}
    }
}
'
```

Elasticsearch 返回一个 HTTP 状态码（例如：200 OK）和（除 `HEAD` 请求）一个 JSON 格式的返回值。 前面的 curl 请求将返回一个像下面一样的 JSON 体：

```
{
    "count" : 0,
    "_shards" : {
        "total" : 5,
        "successful" : 5,
        "failed" : 0
    }
}
```

在返回结果中没有看到 HTTP 头信息是因为我们没有要求 `curl` 显示它们。想要看到头信息，需要结合 -i 参数来使用 curl 命令：

```
curl -i -XGET 'localhost:9200/'
```

在书中剩余的部分，我们将用缩写格式来展示这些 curl 示例， 所谓的缩写格式就是省略请求中所有相同的部分，例如主机名、端口号以及 curl 命令本身。而不是像下面显示的那样用一个完整的请求：

```
curl -XGET 'localhost:9200/_count?pretty' -d '
{
    "query": {
        "match_all": {}
    }
}'
```

我们将用缩写格式显示：

```
GET /_count
{
    "query": {
        "match_all": {}
    }
}
```

事实上， kibana DevTool 控制台 也使用这样相同的格式。

1. 面向文档

在应用程序中对象很少只是一个简单的键和值的列表。通常，它们拥有更复杂的数据结构，可能包括日期、地理信息、其他对象或者数组等。

也许有一天你想把这些对象存储在数据库中。使用关系型数据库的行和列存储，这相当于是把一个表现力丰富的对象塞到一个非常大的电子表格中： 为了适应表结构，你必须设法将这个对象扁平化—通常一个字段对应一列—而且每次查询时又需要将其重新构造为对象。

Elasticsearch 是 面向文档 的，意味着它存储整个对象或 文档。Elasticsearch 不仅存储文档，而且 索引 每个文档的内容， 使之可以被检索。在 Elasticsearch 中，我们对文档进行索引、检索、排序和过滤—而不是对行列数据。 这是一种完全不同的思考数据的方式，也是 Elasticsearch 能支持复杂全文检索的原因。

2. JSON

Elasticsearch 使用 JavaScript Object Notation（或者 JSON）作为文档的序列化格式。 JSON 序列化为大多数编程语言所支持，并且已经成为 NoSQL 领域的标准格式。 它简单、简洁、易于阅读。

下面这个 JSON 文档代表了一个 user 对象：

```
{
    "email":      "john@smith.com",
    "first_name": "John",
    "last_name":  "Smith",
    "info": {
        "bio":         "Eco-warrior and defender of the weak",
        "age":         25,
        "interests": [ "dolphins", "whales" ]
    },
    "join_date": "2014/05/01"
}
```

虽然原始的 user 对象很复杂，但这个对象的结构和含义在 JSON 版本中都得到了体现和保留。 在 Elasticsearch 中将对象转化为 JSON 后构建索引要比在一个扁平的表结构中要简单的多。

> 注意
>
> > 几乎所有的语言都有可以将任意的数据结构或对象转化成 JSON 格式的模块，只是细节各不相同。 具体请查看 serialization 或者 marshalling 这两个处理 JSON 的模块。官方 Elasticsearch 客户端 自动为您提供 JSON 转化。

适应新环境　　为了让大家对 Elasticsearch 能实现什么及其上手难易程度有一个基本印象，让我们从一个简单的教程开始并介绍索引、搜索及聚合等基础概念。

我们将一并介绍一些新的技术术语，即使无法立即全部理解它们也无妨，因为在本书后续内容中，我们将继续深入介绍这里提到的所有概念。

接下来尽情享受 Elasticsearch 探索之旅。

1. 创建一个雇员目录

我们受雇于 Megacorp 公司，作为 HR 部门新的 “热爱无人机” ("We love our drones!") 激励项目的一部分， 我们的任务是为此创建一个员工目录。该目录应当能培养员工认同感及支持实时、高效、动态协作，因此有一些业务需求：

- 支持包含多值标签、数值、以及全文本的数据
- 检索任一员工的完整信息
- 允许结构化搜索，比如查询 30 岁以上的员工
- 允许简单的全文搜索以及较复杂的短语搜索
- 支持在匹配文档内容中高亮显示搜索片段
- 支持基于数据创建和管理分析仪表盘

该文件修订时间： 2021-03-06

1. 索引员工文档

第一个业务需求是存储员工数据。 这将会以 员工文档 的形式存储：一个文档代表一个员工。 存储数据到 Elasticsearch 的行为叫做 索引 ，但在索引一个文档之前，需要确定将文档存储在哪里。

一个 Elasticsearch 集群可以 包含多个 索引 ，相应的每个索引可以包含多个类型 (在6版本之后一个索引表只允许包含一个类型)。 一个类型存储着多个 文档 ，每个文档又有 多个 属性 。

> Index Versus Index Versus Index
>
> > 你也许已经注意到 索引 这个词在 Elasticsearch 语境中有多种含义， 这里有必要做一些说明：
> > **索引（名词）**：
> > 如前所述，一个 索引 类似于传统关系数据库中的一个 数据库 ，是一个存储关系型文档的地方。 索引 (index) 的复数词为 indices 或 indexes 。
> >
> > **索引（动词）**：
> > 索引一个文档 就是存储一个文档到一个 索引 （名词）中以便被检索和查询。这非常类似于 SQL 语句中的 INSERT 关键词，除了文档已存在时，新文档会替换旧文档情况之外。
> >
> > **倒排索引**：
> > 关系型数据库通过增加一个 索引 比如一个 B树（B-tree）索引 到指定的列上，以便提升数据检索速度。Elasticsearch 和 Lucene 使用了一个叫做 倒排索引 的结构来达到相同的目的。
> >
> > 默认的，一个文档中的每一个属性都是 被索引 的（有一个倒排索引）和可搜索的。 一个没有倒排索引的属性是不能被搜索到的。我们将在 倒排索引 讨论倒排索引的更多细节。

对于员工目录，我们将做如下操作：

- 每个员工索引一个文档，文档包含该员工的所有信息。
- 每个文档都将是 *employee* 类型 。
- 该类型位于 索引 *megacorp* 内。
- 该索引保存在我们的 Elasticsearch 集群中。

实践中这非常简单（尽管看起来有很多步骤），我们可以通过一条命令完成所有这些动作：

```
PUT /megacorp/employee/1
{
    "first_name" : "John",
    "last_name" :  "Smith",
    "age" :        25,
    "about" :      "I love to go rock climbing",
    "interests": [ "sports", "music" ]
}
```

注意，路径 /megacorp/employee/1 包含了三部分的信息：

megacorp
索引名称
employee
类型名称
1
特定雇员的ID

请求体 —— JSON 文档 —— 包含了这位员工的所有详细信息，他的名字叫 John Smith ，今年 25 岁，喜欢攀岩。

很简单！无需进行执行管理任务，如创建一个索引或指定每个属性的数据类型之类的，可以直接只索引一个文档。Elasticsearch 默认地完成其他一切，因此所有必需的管理任务都在后台使用默认设置完成。

进行下一步前，让我们增加更多的员工信息到目录中：

```
PUT /megacorp/employee/2
{
    "first_name" :  "Jane",
    "last_name" :   "Smith",
    "age" :         32,
    "about" :       "I like to collect rock albums",
    "interests":  [ "music" ]
}

PUT /megacorp/employee/3
{
    "first_name" :  "Douglas",
    "last_name" :   "Fir",
    "age" :         35,
    "about":        "I like to build cabinets",
    "interests":  [ "forestry" ]
}
```

1. 检索文档

目前我们已经在 Elasticsearch 中存储了一些数据， 接下来就能专注于实现应用的业务需求了。第一个需求是可以检索到单个雇员的数据。

这在 Elasticsearch 中很简单。简单地执行 一个 HTTP GET 请求并指定文档的地址——索引库、类型和ID。 使用这三个信息可以返回原始的 JSON 文档：

```
GET /megacorp/employee/1
```

返回结果包含了文档的一些元数据，以及 _source 属性，内容是 John Smith 雇员的原始 JSON 文档：

```
{
  "_index" :   "megacorp",
  "_type" :    "employee",
  "_id" :      "1",
  "_version" : 1,
  "found" :    true,
  "_source" :  {
      "first_name" :  "John",
      "last_name" :   "Smith",
      "age" :         25,
      "about" :       "I love to go rock climbing",
      "interests":  [ "sports", "music" ]
  }
}
```

> 建议
>
> > 将 HTTP 命令由 PUT 改为 GET 可以用来检索文档，同样的，可以使用 DELETE 命令来删除文档， 以及使用 HEAD 指令来检查文档是否存在。如果想更新已存在的文档，只需再次 PUT 。

1. 轻量检索

一个 GET 是相当简单的，可以直接得到指定的文档。 现在尝试点儿稍微高级的功能，比如一个简单的搜索！

第一个尝试的几乎是最简单的搜索了。我们使用下列请求来搜索所有雇员：

```
GET /megacorp/employee/_search
```

可以看到，我们仍然使用索引库 megacorp 以及类型 employee，但与指定一个文档 ID 不同，这次使用 _search 。 返回结果包括了所有三个文档，放在数组 hits 中。一个搜索默认返回十条结果。

```
{
   "took":      6,
   "timed_out": false,
   "_shards": { ... },
   "hits": {
      "total":      3,
      "max_score":  1,
      "hits": [
         {
            "_index":         "megacorp",
            "_type":          "employee",
            "_id":            "3",
            "_score":         1,
            "_source": {
               "first_name":  "Douglas",
               "last_name":   "Fir",
               "age":         35,
               "about":       "I like to build cabinets",
               "interests": [ "forestry" ]
            }
         },
         {
            "_index":         "megacorp",
            "_type":          "employee",
            "_id":            "1",
            "_score":         1,
            "_source": {
               "first_name":  "John",
               "last_name":   "Smith",
               "age":         25,
               "about":       "I love to go rock climbing",
               "interests": [ "sports", "music" ]
            }
         },
         {
            "_index":         "megacorp",
            "_type":          "employee",
            "_id":            "2",
            "_score":         1,
            "_source": {
               "first_name":  "Jane",
               "last_name":   "Smith",
               "age":         32,
               "about":       "I like to collect rock albums",
               "interests": [ "music" ]
            }
         }
      ]
   }
}
```

注意：返回结果不仅告知匹配了哪些文档，还包含了整个文档本身：显示搜索结果给最终用户所需的全部信息。

接下来，尝试下搜索姓氏为 `Smith` 的雇员。为此，我们将使用一个 高亮 搜索，很容易通过命令行完成。 这个方法一般涉及到一个 查询字符串 （query-string） 搜索，因为我们通过一个URL参数来传递查询信息给搜索接口：

```
GET /megacorp/employee/_search?q=last_name:Smith
```

我们仍然在请求路径中使用 _search 端点，并将查询本身赋值给参数 q= 。返回结果给出了所有的 Smith：

```
{
   ...
   "hits": {
      "total":      2,
      "max_score":  0.30685282,
      "hits": [
         {
            ...
            "_source": {
               "first_name":  "John",
               "last_name":   "Smith",
               "age":         25,
               "about":       "I love to go rock climbing",
               "interests": [ "sports", "music" ]
            }
         },
         {
            ...
            "_source": {
               "first_name":  "Jane",
               "last_name":   "Smith",
               "age":         32,
               "about":       "I like to collect rock albums",
               "interests": [ "music" ]
            }
         }
      ]
   }
}
```

1. 使用查询表达式搜索

Query-string 搜索通过命令非常方便地进行临时性的即席搜索，但它有自身的局限性（参见 轻量搜索）。 Elasticsearch 提供一个丰富灵活的查询语言叫做 查询表达式， 它支持构建更加复杂和健壮的查询。

领域特定语言 （DSL）， 使用 JSON 构造了一个请求。我们可以像这样重写之前的查询所有名为 Smith 的搜索：

```
GET /megacorp/employee/_search
{
    "query" : {
        "match" : {
            "last_name" : "Smith"
        }
    }
}
```

返回结果与之前的查询一样，但还是可以看到有一些变化。其中之一是，不再使用 query-string 参数， 而是一个请求体替代。这个请求使用 JSON 构造，并使用了一个 match 查询（属于查询类型之一，后面将继续介绍）。

1. 更复杂的搜索

现在尝试下更复杂的搜索。 同样搜索姓氏为 Smith 的员工，但这次我们只需要年龄大于 30 的。 查询需要稍作调整，使用过滤器 filter ， 它支持高效地执行一个结构化查询。

```
GET /megacorp/employee/_search
{
    "query" : {
        "bool": {
            "must": {
                "match" : {
                    "last_name" : "smith"  (a)
                }
            },
            "filter": {
                "range" : {
                    "age" : { "gt" : 30 }  (b)
                }
            }
        }
    }
}
```

(a) 这部分与我们之前使用的 match 查询 一样。

(b) 这部分是一个 range 过滤器， 它能找到年龄大于 30 的文档，其中 gt 表示大于 (great than)。

目前无需太多担心语法问题，后续会更详细地介绍。只需明确我们添加了一个过滤器 用于执行一个范围查询， 并复用之前的 match 查询。现在结果只返回了一名员工，叫 Jane Smith，32 岁。

```
{
   ...
   "hits": {
      "total":      1,
      "max_score":  0.30685282,
      "hits": [
         {
            ...
            "_source": {
               "first_name":  "Jane",
               "last_name":   "Smith",
               "age":         32,
               "about":       "I like to collect rock albums",
               "interests": [ "music" ]
            }
         }
      ]
   }
}
```

1.1. 全文搜索

截止目前的搜索相对都很简单：单个姓名，通过年龄过滤。现在尝试下稍微高级点儿的全文搜索——一项 传统数据库确实很难搞定的任务。

搜索下所有喜欢攀岩（rock climbing）的员工：

```
GET /megacorp/employee/_search
{
    "query" : {
        "match" : {
            "about" : "rock climbing"
        }
    }
}
```

显然我们依旧使用之前的 match 查询在 `about` 属性上搜索 “rock climbing” 。得到两个匹配的文档：

```
{
   ...
   "hits": {
      "total":      2,
      "max_score":  0.16273327,
      "hits": [
         {
            ...
            "_score":         0.16273327, (a)
            "_source": {
               "first_name":  "John",
               "last_name":   "Smith",
               "age":         25,
               "about":       "I love to go rock climbing",
               "interests": [ "sports", "music" ]
            }
         },
         {
            ...
            "_score":         0.016878016, (a)
            "_source": {
               "first_name":  "Jane",
               "last_name":   "Smith",
               "age":         32,
               "about":       "I like to collect rock albums",
               "interests": [ "music" ]
            }
         }
      ]
   }
}
```

(a) 相关性得分

Elasticsearch 默认按照相关性得分排序，即每个文档跟查询的匹配程度。 第一个最高得分的结果很明显：John Smith 的 about 属性清楚地写着 “rock climbing” 。

但为什么 Jane Smith 也作为结果返回了呢？原因是她的 about 属性里提到了 “rock” 。 因为只有 “rock” 而没有 “climbing” ，所以她的相关性得分低于 John 的。

这是一个很好的案例，阐明了 Elasticsearch 如何 在 全文属性上搜索并返回相关性最强的结果。 Elasticsearch中的 相关性 概念非常重要，也是完全区别于传统关系型数据库的一个概念，数据库中的一条记录要么匹配要么不匹配。

1. 短语搜索

找出一个属性中的独立单词是没有问题的，但有时候想要精确匹配一系列单词或者*短语* 。 比如， 我们想执行这样一个查询，仅匹配同时包含 “rock” 和 “climbing” ，并且 二者以短语 “rock climbing” 的形式紧挨着的雇员记录。

为此对 match 查询稍作调整，使用一个叫做 match_phrase 的查询：

```
GET /megacorp/employee/_search
{
    "query" : {
        "match_phrase" : {
            "about" : "rock climbing"
        }
    }
}
```

毫无悬念，返回结果仅有 John Smith 的文档。

```
{
   ...
   "hits": {
      "total":      1,
      "max_score":  0.23013961,
      "hits": [
         {
            ...
            "_score":         0.23013961,
            "_source": {
               "first_name":  "John",
               "last_name":   "Smith",
               "age":         25,
               "about":       "I love to go rock climbing",
               "interests": [ "sports", "music" ]
            }
         }
      ]
   }
}
```

1. 高亮搜索

许多应用都倾向于在每个搜索结果中 高亮 部分文本片段，以便让用户知道为何该文档符合查询条件。在 Elasticsearch 中检索出高亮片段也很容易。

再次执行前面的查询，并增加一个新的 highlight 参数：

```
GET /megacorp/employee/_search
{
    "query" : {
        "match_phrase" : {
            "about" : "rock climbing"
        }
    },
    "highlight": {
        "fields" : {
            "about" : {}
        }
    }
}
```

当执行该查询时，返回结果与之前一样，与此同时结果中还多了一个叫做 highlight 的部分。这个部分包含了 about 属性匹配的文本片段，并以 HTML 标签封装：

```
{
   ...
   "hits": {
      "total":      1,
      "max_score":  0.23013961,
      "hits": [
         {
            ...
            "_score":         0.23013961,
            "_source": {
               "first_name":  "John",
               "last_name":   "Smith",
               "age":         25,
               "about":       "I love to go rock climbing",
               "interests": [ "sports", "music" ]
            },
            "highlight": {
               "about": [
                  "I love to go <em>rock</em> <em>climbing</em>" (a)
               ]
            }
         }
      ]
   }
}
```

(a) 原始文本中的高亮片段

关于高亮搜索片段，可以在 highlighting reference documentation 了解更多信息。

1. 分析

终于到了最后一个业务需求：支持管理者对员工目录做分析。 Elasticsearch 有一个功能叫聚合（aggregations），允许我们基于数据生成一些精细的分析结果。聚合与 SQL 中的 GROUP BY 类似但更强大。

举个例子，挖掘出员工中最受欢迎的兴趣爱好：

```
GET /megacorp/employee/_search
{
  "aggs": {
    "all_interests": {
      "terms": { "field": "interests" }
    }
  }
}
```

暂时忽略掉语法，直接看看结果：

```
{
   ...
   "hits": { ... },
   "aggregations": {
      "all_interests": {
         "buckets": [
            {
               "key":       "music",
               "doc_count": 2
            },
            {
               "key":       "forestry",
               "doc_count": 1
            },
            {
               "key":       "sports",
               "doc_count": 1
            }
         ]
      }
   }
}
```

可以看到，两位员工对音乐感兴趣，一位对林业感兴趣，一位对运动感兴趣。这些聚合的结果数据并非预先统计， 而是根据匹配当前查询的文档即时生成的。如果想知道叫 Smith 的员工中最受欢迎的兴趣爱好，可以直接构造一个组合查询：

```
GET /megacorp/employee/_search
{
  "query": {
    "match": {
      "last_name": "smith"
    }
  },
  "aggs": {
    "all_interests": {
      "terms": {
        "field": "interests"
      }
    }
  }
}
```

all_interests 聚合已经变为只包含匹配查询的文档：

```
...
  "all_interests": {
     "buckets": [
        {
           "key": "music",
           "doc_count": 2
        },
        {
           "key": "sports",
           "doc_count": 1
        }
     ]
  }
```

聚合还支持分级汇总 。比如，查询特定兴趣爱好员工的平均年龄：

```
GET /megacorp/employee/_search
{
    "aggs" : {
        "all_interests" : {
            "terms" : { "field" : "interests" },
            "aggs" : {
                "avg_age" : {
                    "avg" : { "field" : "age" }
                }
            }
        }
    }
}
```

得到的聚合结果有点儿复杂，但理解起来还是很简单的：

```
...
 "all_interests": {
    "buckets": [
       {
          "key": "music",
          "doc_count": 2,
          "avg_age": {
             "value": 28.5
          }
       },
       {
          "key": "forestry",
          "doc_count": 1,
          "avg_age": {
             "value": 35
          }
       },
       {
          "key": "sports",
          "doc_count": 1,
          "avg_age": {
             "value": 25
          }
       }
    ]
 }
```

输出基本是第一次聚合的加强版。依然有一个兴趣及数量的列表，只不过每个兴趣都有了一个附加的 avg_age 属性， 代表有这个兴趣爱好的所有员工的平均年龄。

即使现在不太理解这些语法也没有关系，依然很容易了解到复杂聚合及分组通过 Elasticsearch 特性实现得很完美， 能够提取的数据类型也没有任何限制。

1. 教程结语

欣喜的是，这是一个关于 Elasticsearch 基础描述的教程，且仅仅是浅尝辄止， 更多诸如 *suggestions*、*geolocation*、*percolation*、*fuzzy* 与 *partial matching* 等特性均被省略， 以便保持教程的简洁。但它确实突显了开始构建高级搜索功能多么容易。不需要配置——只需要添加数据并开始搜索！

很可能语法会让你在某些地方有所困惑，并且对各个方面如何微调也有一些问题。 没关系！本书后续内容将针对每个问题详细解释，让你全方位地理解 Elasticsearch 的工作原理。

1. 分布式特性

在本章开头，我们提到过 Elasticsearch 可以横向扩展至数百（甚至数千）的服务器节点， 同时可以处理PB级数据。我们的教程给出了一些使用 Elasticsearch 的示例，但并不涉及任何内部机制。 Elasticsearch 天生就是分布式的，并且在设计时屏蔽了分布式的复杂性。

Elasticsearch 在分布式方面几乎是透明的。 教程中并不要求了解分布式系统、分片、集群发现或其他的各种分布式概念。 可以使用笔记本上的单节点轻松地运行教程里的程序，但如果你想要在 100 个节点的集群上运行程序，一切依然顺畅。

Elasticsearch 尽可能地屏蔽了分布式系统的复杂性。这里列举了一些在后台自动执行的操作：

- 分配文档到不同的容器 或 分片 中，文档可以储存在一个或多个节点中
- 按集群节点来均衡分配这些分片，从而对索引和搜索过程进行负载均衡
- 复制每个分片以支持数据冗余，从而防止硬件故障导致的数据丢失
- 将集群中任一节点的请求路由到存有相关数据的节点
- 集群扩容时无缝整合新节点，重新分配分片以便从离群节点恢复

当阅读本书时，将会遇到有关 Elasticsearch 分布式特性的补充章节。 这些章节将介绍有关集群扩容、故障转移(集群内的原理) 、应对文档存储(分布式文档存储) 、执行分布式搜索(执行分布式检索) ， 以及分区（shard）及其工作原理(分片内部原理) 。

这些章节并非必读，完全可以无需了解内部机制就使用 Elasticsearch， 但是它们将从另一个角度帮助你了解更完整的 Elasticsearch 知识。可以根据需要跳过它们， 或者想更完整地理解时再回头阅读也无妨。

1. 集群内的原理

> 补充章节
>
> > 如前文所述，这是补充章节中第一篇介绍 Elasticsearch 在分布式环境中的运行原理。 在这个章节中， 我们将会介绍 cluster 、 node 、 shard 等常用术语，Elastisearch 的扩容机制， 以及如何处理硬件故障的内容。
> >
> > 虽然这个章节不是必读的—您完全可以在不关注分片、副本和失效切换等内容的情况下长期使用Elasticsearch-- 但是这将帮助你了解 Elasticsearch 的内部工作过程。您可以先快速阅览该章节，将来有需要时再次查看。

ElasticSearch 的主旨是随时可用和按需扩容。 而扩容可以通过购买性能更强大（ 垂直扩容 ，或 纵向扩容 ） 或者数量更多的服务器（ 水平扩容 ，或 横向扩容 ）来实现。

虽然 Elasticsearch 可以获益于更强大的硬件设备，但是垂直扩容是有极限的。 真正的扩容能力是来自于水平扩容—为集群添加更多的节点，并且将负载压力和稳定性分散到这些节点中。

对于大多数的数据库而言，通常需要对应用程序进行非常大的改动，才能利用上横向扩容的新增资源。 与之相反的是，ElastiSearch天生就是 分布式的 ，它知道如何通过管理多节点来提高扩容性和可用性。 这也意味着你的应用无需关注这个问题。

本章将讲述如何按需配置集群、节点和分片，并在硬件故障时确保数据安全。

1. 空集群

如果我们启动了一个单独的节点，里面不包含任何的数据和索引，那我们的集群看起来就是一个 Figure 1, “包含空内容节点的集群”。

> Figure 1, “包含空内容节点的集群”

一个运行中的 Elasticsearch 实例称为一个节点，而集群是由一个或者多个拥有相同 cluster.name 配置的节点组成， 它们共同承担数据和负载的压力。 当有节点加入集群中或者从集群中移除节点时，集群将会重新平均分布所有的数据。

当一个节点被选举成为 主 节点时， 它将负责管理集群范围内的所有变更，例如增加、删除索引，或者增加、删除节点等。 而主节点并不需要涉及到文档级别的变更和搜索等操作，所以当集群只拥有一个主节点的情况下，即使流量的增加它也不会成为瓶颈。 任何节点都可以成为主节点。我们的示例集群就只有一个节点，所以它同时也成为了主节点。

作为用户，我们可以将请求发送到 集群中的任何节点 ，包括主节点。 每个节点都知道任意文档所处的位置， 并且能够将我们的请求直接转发到存储我们所需文档的节点。 无论我们将请求发送到哪个节点， 它都能负责从各个包含我们所需文档的节点收集回数据，并将最终结果返回給客户端。 Elasticsearch 对这一切的管理都是透明的。

1. 集群健康

Elasticsearch 的集群监控信息中包含了许多的统计数据，其中最为重要的一项就是 集群健康， 它在 status 字段中展示为 green 、 yellow 或者 red 。

```
GET /_cluster/health
```

在一个不包含任何索引的空集群中，它将会有一个类似于如下所示的返回内容：

```
{
  "cluster_name": "elasticsearch",
  "status": "yellow", (a)
  "timed_out": false,
  "number_of_nodes": 10,
  "number_of_data_nodes": 9,
  "active_primary_shards": 6921,
  "active_shards": 13661,
  "relocating_shards": 0,
  "initializing_shards": 0,
  "unassigned_shards": 181,
  "delayed_unassigned_shards": 0,
  "number_of_pending_tasks": 0,
  "number_of_in_flight_fetch": 0,
  "task_max_waiting_in_queue_millis": 0,
  "active_shards_percent_as_number": 98.6923854934258
}
```

(a)status 字段是我们最关心的。

status 字段指示着当前集群在总体上是否工作正常。它的三种颜色含义如下：

green

所有的主分片和副本分片都正常运行。

yellow

所有的主分片都正常运行，但不是所有的副本分片都正常运行。

red

有主分片没能正常运行。

在本章节剩余的部分，我们将解释什么是 *主* 分片和 *副本* 分片，以及上面提到的这些颜色的实际意义。

1. 添加索引

我们往 Elasticsearch 添加数据时需要用到 索引 —— 保存相关数据的地方。索引实际上是指向一个或者多个物理 分片 的 逻辑命名空间 。

一个 分片 是一个底层的 工作单元 ，它仅保存了全部数据中的一部分。 在分片内部机制中， 我们将详细介绍分片是如何工作的，而现在我们只需知道一个分片是一个 Lucene 的实例，以及它本身就是一个完整的搜索引擎。 我们的文档被存储和索引到分片内，但是应用程序是直接与索引而不是与分片进行交互。

Elasticsearch 是利用分片将数据分发到集群内各处的。分片是数据的容器，文档保存在分片内， 分片又被分配到集群内的各个节点里。 当你的集群规模扩大或者缩小时， Elasticsearch 会自动的在各节点中迁移分片， 使得数据仍然均匀分布在集群里。

一个分片可以是 主 分片或者 副本 分片。 索引内任意一个文档都归属于一个主分片， 所以主分片的数目决定着索引能够保存的最大数据量。

> 注意
>
> > 技术上来说，一个主分片最大能够存储 Integer.MAX_VALUE - 128 个文档， 但是实际最大值还需要参考你的使用场景：包括你使用的硬件， 文档的大小和复杂程度， 索引和查询文档的方式以及你期望的响应时长。

一个副本分片只是一个主分片的拷贝。副本分片作为硬件故障时保护数据不丢失的冗余备份， 并为搜索和返回文档等读操作提供服务。

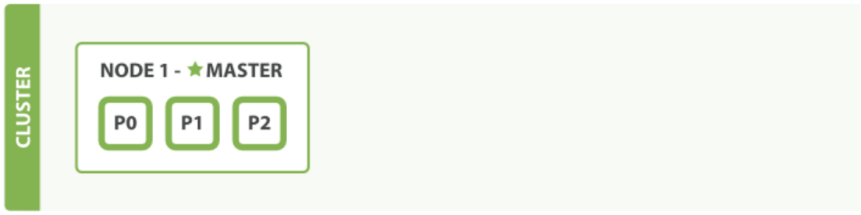
在索引建立的时候就已经确定了主分片数，但是副本分片数可以随时修改。

让我们在包含一个空节点的集群内创建名为 blogs 的索引。 索引在默认情况下会被分配5个主分片， 但是为了演示目的，我们将分配3个主分片和一份副本（每个主分片拥有一个副本分片）：

```
PUT /blogs
{
   "settings" : {
      "number_of_shards" : 3,
      "number_of_replicas" : 1
   }
}
```

我们的集群现在是Figure 2, “拥有一个索引的单节点集群”。所有3个主分片都被分配在 Node 1 。

> Figure 2. 拥有一个索引的单节点集群



如果我们现在查看集群健康，我们将看到如下内容：

```
{
  "cluster_name": "elasticsearch",
  "status": "yellow", (a)
  "timed_out": false,
  "number_of_nodes": 1,
  "number_of_data_nodes": 1,
  "active_primary_shards": 3,
  "active_shards": 3,
  "relocating_shards": 0,
  "initializing_shards": 0,
  "unassigned_shards": 3,  (b)
  "delayed_unassigned_shards": 0,
  "number_of_pending_tasks": 0,
  "number_of_in_flight_fetch": 0,
  "task_max_waiting_in_queue_millis": 0,
  "active_shards_percent_as_number": 50
}
```

(a)集群 status 值为 yellow 。
(b)没有被分配到任何节点的副本数。

集群的健康状况为 yellow 则表示全部 主 分片都正常运行（集群可以正常服务所有请求）， 但是 副本 分片没有全部处在正常状态。 实际上，所有3个副本分片都是 unassigned —— 它们都没有被分配到任何节点。 在同一个节点上既保存原始数据又保存副本是没有意义的，因为一旦失去了那个节点，我们也将丢失该节点上的所有副本数据。

当前我们的集群是正常运行的，但是在硬件故障时有丢失数据的风险。

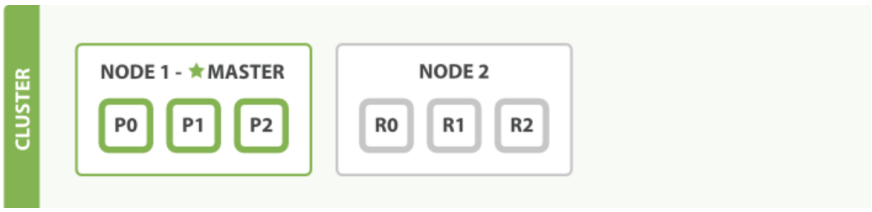
1. 添加故障转移

当集群中只有一个节点在运行时，意味着会有一个单点故障问题——没有冗余。 幸运的是，我们只需再启动一个节点即可防止数据丢失。

> 启动第二个节点
>
> > 为了测试第二个节点启动后的情况，你可以在同一个目录内，完全依照启动第一个节点的方式来启动一个新节点 （参考安装并运行Elasticsearch）。多个节点可以共享同一个目录。
> >
> > 当你在同一台机器上启动了第二个节点时，只要它和第一个节点有同样的 cluster.name 配置， 它就会自动发现集群并加入到其中。 但是在不同机器上启动节点的时候，为了加入到同一集群， 你需要配置一个可连接到的单播主机列表。 详细信息请查看最好使用单播代替组播

如果启动了第二个节点，我们的集群将会如Figure 3, “拥有两个节点的集群——所有主分片和副本分片都已被分配”所示。

> Figure 3. 拥有两个节点的集群——所有主分片和副本分片都已被分配



当第二个节点加入到集群后，3个 副本分片 将会分配到这个节点上——每个主分片对应一个副本分片。 这意味着当集群内任何一个节点出现问题时，我们的数据都完好无损。

所有新近被索引的文档都将会保存在主分片上，然后被并行的复制到对应的副本分片上。 这就保证了我们既可以从主分片又可以从副本分片上获得文档。

cluster-health 现在展示的状态为 *green* ， 这表示所有6个分片（包括3个主分片和3个副本分片）都在正常运行。

```
{
  "cluster_name": "elasticsearch",
  "status": "green", (a)
  "timed_out": false,
  "number_of_nodes": 2,
  "number_of_data_nodes": 2,
  "active_primary_shards": 3,
  "active_shards": 6,
  "relocating_shards": 0,
  "initializing_shards": 0,
  "unassigned_shards": 0,
  "delayed_unassigned_shards": 0,
  "number_of_pending_tasks": 0,
  "number_of_in_flight_fetch": 0,
  "task_max_waiting_in_queue_millis": 0,
  "active_shards_percent_as_number": 100
}
```

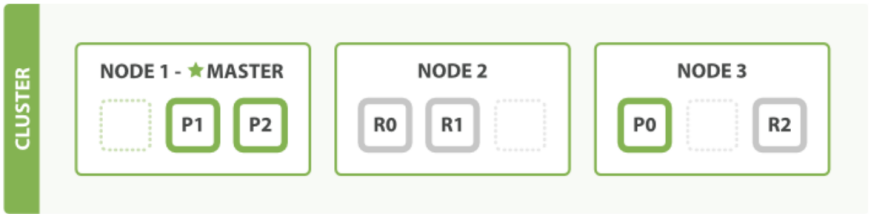
(a)集群 status 值为 green 。

我们的集群现在不仅仅是正常运行的，并且还处于 始终可用 的状态。

1. 水平扩容

怎样为我们的正在增长中的应用程序按需扩容呢？ 当启动了第三个节点，我们的集群将会看起来 如Figure 4, “拥有三个节点的集群——为了分散负载而对分片进行重新分配”所示。

> Figure 4. 拥有三个节点的集群——为了分散负载而对分片进行重新分配



Node 1 和 *Node 2* 上各有一个分片被迁移到了新的 Node 3 节点，现在每个节点上都拥有2个分片，而不是之前的3个。 这表示每个节点的硬件资源（CPU, RAM, I/O）将被更少的分片所共享，每个分片的性能将会得到提升。

分片是一个功能完整的搜索引擎，它拥有使用一个节点上的所有资源的能力。我们这个拥有6个分片（3个主分片和3个副本分片）的索引可以最大扩容到6个节点， 每个节点上存在一个分片，并且每个分片拥有所在节点的全部资源。

2. 更多的扩容

但是如果我们想要扩容超过6个节点怎么办呢?

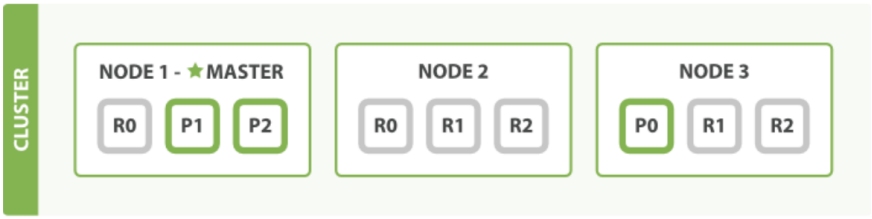
主分片的数目在索引创建时就已经确定了下来。实际上，这个数目定义了这个索引能够 存储 的最大数据量。 （实际大小取决于你的数据、硬件和使用场景。）但是，读操作——搜索和返回数据——可以同时被主分片 或 副本分片所处理，所以当你拥有越多的副本分片时，也将拥有越高的吞吐量。

在运行中的集群上是可以动态调整副本分片数目的，我们可以按需伸缩集群。让我们把副本数从默认的 1 增加到 2：

```
PUT /blogs/_settings
{
   "index.number_of_replicas" : 2
}
```

如Figure 5, “将参数 number_of_replicas 调大到 2”所示， blogs 索引现在拥有9个分片：3个主分片和6个副本分片。 这意味着我们可以将集群扩容到9个节点，每个节点上一个分片。相比原来3个节点时，集群搜索性能可以提升 3 倍。

> Figure 5. 将参数 number_of_replicas 调大到 2

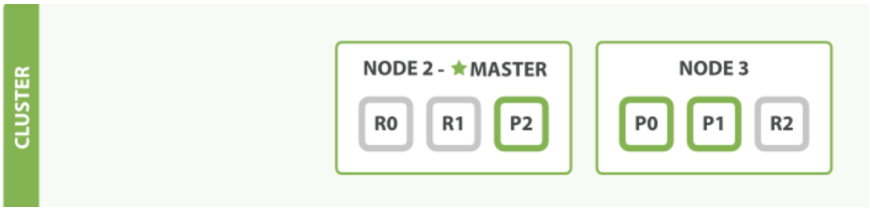


> 注意
>
> > 当然，如果只是在相同节点数目的集群上增加更多的副本分片并不能提高性能，因为每个分片从节点上获得的资源会变少。 你需要增加更多的硬件资源来提升吞吐量。 但是更多的副本分片数提高了数据冗余量：按照上面的节点配置，我们可以在失去2个节点的情况下不丢失任何数据。

1. 应对故障

我们之前说过 Elasticsearch 可以应对节点故障，接下来让我们尝试下这个功能。 如果我们关闭第一个节点，这时集群的状态为Figure 6, “关闭了一个节点后的集群”

> Figure 6. 关闭了一个节点后的集群



我们关闭的节点是一个主节点。而集群必须拥有一个主节点来保证正常工作，所以发生的第一件事情就是选举一个新的主节点： Node 2 。

在我们关闭 Node 1 的同时也失去了主分片 1 和 2 ，并且在缺失主分片的时候索引也不能正常工作。 如果此时来检查集群的状况，我们看到的状态将会为 red ：不是所有主分片都在正常工作。

幸运的是，在其它节点上存在着这两个主分片的完整副本， 所以新的主节点立即将这些分片在 Node 2 和 Node 3 上对应的副本分片提升为主分片， 此时集群的状态将会为 yellow 。 这个提升主分片的过程是瞬间发生的，如同按下一个开关一般。

为什么我们集群状态是 yellow 而不是 green 呢？ 虽然我们拥有所有的三个主分片， 但是同时设置了每个主分片需要对应2份副本分片，而此时只存在一份副本分片。 所以集群不能为 green 的状态，不过我们不必过于担心：如果我们同样关闭了 Node 2 ， 我们的程序 依然 可以保持在不丢任何数据的情况下运行，因为 Node 3 为每一个分片都保留着一份副本。

如果我们重新启动 Node 1 ，集群可以将缺失的副本分片再次进行分配， 那么集群的状态也将如Figure 5, “将参数 number_of_replicas 调大到 2”所示。 如果 Node 1 依然拥有着之前的分片，它将尝试去重用它们，同时仅从主分片复制发生了修改的数据文件。

到目前为止，你应该对分片如何使得 Elasticsearch 进行水平扩容以及数据保障等知识有了一定了解。 接下来我们将讲述关于分片生命周期的更多细节。

1. 数据输入和输出

无论我们写什么样的程序，目的都是一样的：以某种方式组织数据服务我们的目的。 但是数据不仅仅由随机位和字节组成。 我们建立数据元素之间的关系以便于表示实体，或者现实世界中存在的 事物 。 如果我们知道一个名字和电子邮件地址属于同一个人，那么它们将会更有意义。

尽管在现实世界中，不是所有的类型相同的实体看起来都是一样的。 一个人可能有一个家庭电话号码，而另一个人只有一个手机号码，再一个人可能两者兼有。一个人可能有三个电子邮件地址，而另一个人却一个都没有。一位西班牙人可能有两个姓，而讲英语的人可能只有一个姓。

面向对象编程语言如此流行的原因之一是对象帮我们表示和处理现实世界具有潜在的复杂的数据结构的实体， 到目前为止，一切都很完美！

但是当我们需要存储这些实体时问题来了，传统上，我们以行和列的形式存储数据到关系型数据库中， 相当于使用电子表格。 正因为我们使用了这种不灵活的存储媒介导致所有我们使用对象的灵活性都丢失了。

但是否我们可以将我们的对象按对象的方式来存储？这样我们就能更加专注于使用 数据， 而不是在电子表格的局限性下对我们的应用建模。 我们可以重新利用对象的灵活性。

一个 对象 是基于特定语言的内存的数据结构。为了通过网络发送或者存储它， 我们需要将它表示成某种标准的格式。 JSON 是一种以人可读的文本表示对象的方法。 它已经变成 NoSQL 世界交换数据的事实标准。当一个对象被序列化成为 JSON，它被称为一个 JSON 文档 。

Elastcisearch 是分布式的 文档 存储。它能存储和检索复杂的数据结构—序列化成为JSON文档—以 实时 的方式。 换句话说，一旦一个文档被存储在 Elasticsearch 中，它就是可以被集群中的任意节点检索到。

当然，我们不仅要存储数据，我们一定还需要查询它，成批且快速的查询它们。 尽管现存的 NoSQL 解决方案允许我们以文档的形式存储对象，但是他们仍旧需要我们思考如何查询我们的数据， 以及确定哪些字段需要被索引以加快数据检索。

在 Elasticsearch 中， 每个字段的所有数据 都是 默认被索引的 。 即每个字段都有为了快速检索设置的专用倒排索引。而且，不像其他多数的数据库， 它能在同一个查询中 使用所有这些倒排索引，并以惊人的速度返回结果。

在本章中，我们展示了用来创建，检索，更新和删除文档的 API。就目前而言， 我们不关心文档中的数据或者怎样查询它们。 所有我们关心的就是在 Elasticsearch 中怎样安全的存储文档，以及如何将文档再次返回。