<https://my.oschina.net/galenz/blog/422189>

在Elasticsearch中，文档归属于一种类型(type),而这些类型存在于索引(index)中，我们可以 画一些简单的对比图来类比传统关系型数据库：

Relational DB -> Databases -> Tables -> Rows -> Columns Elasticsearch -> Indices -> Types -> Documents -> Fields

Elasticsearch集群可以包含多个索引(indices)（数据库），每一个索引可以包含多个类型 (types)（表），每一个类型包含多个文档(documents)（行），然后每个文档包含多个字段 (Fields)（列）。

（索引（名词） 如上文所述，一个索引(index)就像是传统关系数据库中的数据库，它 是相关文档存储的地方，index的复数是indices 或indexes。 索引（动词） 「索引一个文档」表示把一个文档存储到索引（名词）里，以便它可 以被检索或者查询。这很像SQL中的 INSERT 关键字，差别是，如果文档已经存在， 新的文档将覆盖旧的文档。 倒排索引 传统数据库为特定列增加一个索引，例如B-Tree索引来加速检索。 Elasticsearch和Lucene使用一种叫做倒排索引(inverted index)的数据结构来达到相 同目的）

**创建实例：**

**为了创建员工目录，我们将进行如下操作：**

**为每个员工的文档(document)建立索引，每个文档包含了相应员工的所有信息。 每个文档的类型为 employee 。 employee 类型归属于索引 megacorp 。 megacorp 索引存储在Elasticsearch集群中。**

**能通过一个命令执行完成的操 作：**

**PUT /megacorp/employee/1 { "first\_name" : "John", "last\_name" : "Smith", "age" : 25, "about" : "I love to go rock climbing", "interests": [ "sports", "music" ] }**

**我们看到path: /megacorp/employee/1 包含三部分信息：**

**名字 说明 megacorp 索引名 employee 类型名 1 这个员工的ID**

**Elasticsearch权威指南（中文版）**

**27索 引**

**请求实体（JSON文档），包含了这个员工的所有信息。他的名字叫“John Smith”，25岁，喜 欢攀岩。**

**它不需要你做额外的管理操作，比如创建索引或者定义每个字段的数据类型。我 们能够直接索引文档，Elasticsearch已经内置所有的缺省设置，所有管理操作都是透明的。**

**检索文档：**

**检索单个员工的信息**

**这对于Elasticsearch来说非常简单。我们只要执行HTTP GET请求并指出文档的“地址”——索 引、类型和ID既可。根据这三部分信息，我们就可以返回原始JSON文档：**

**GET /megacorp/employee/1**

**响应的内容中包含一些文档的元信息，John Smith的原始JSON文档包含在 \_source 字段中。**

**{ "\_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" ] } }**

**我们通过HTTP方法 GET 来检索文档，同样的，我们可以使用 DELETE 方法删除文档，使 用 HEAD 方法检查某文档是否存在。如果想更新已存在的文档，我们只需再 PUT 一次**

**搜索：**

**最简单的搜索全部员工的请求：GET /megacorp/employee/\_search**

**可以看到我们依然使用 megacorp 索引和 employee 类型，但是我们在结尾使用关键 字 \_search 来取代原来的文档ID。响应内容的 hits 数组中包含了我们所有的三个文档。默认 情况下搜索会返回前10个结果。**

**搜索姓氏中包含“Smith”的员工。要做到这一点，我们将在命令行中使用轻量 级的搜索方法。这种方法常被称作查询字符串(query string)搜索，因为我们像传递URL参数 一样去传递查询语句：**

**GET /megacorp/employee/\_search?q=last\_name:Smith**

**我们在请求中依旧使用 \_search 关键字，然后将查询语句传递给参数 q= 。这样就可以得到所 有姓氏为Smith的结果：**

### DSL语句查询

查询字符串搜索便于通过命令行完成特定(ad hoc)的搜索，但是它也有局限性（参阅简单搜索 章节）。Elasticsearch提供丰富且灵活的查询语言叫做DSL查询(Query DSL),它允许你构建 更加复杂、强大的查询

DSL(Domain Specific Language特定领域语言)以JSON请求体的形式出现。我们可以这样 表示之前关于“Smith”的查询:

GET /megacorp/employee/\_search { "query" : { "match" : { "last\_name" : "Smith" } } }

这会返回与之前查询相同的结果。你可以看到有些东西改变了，我们不再使用查询字符串 (query string)做为参数，而是使用请求体代替。这个请求体使用JSON表示，其中使用 了 match 语句（查询类型之一，具体我们以后会学到）。

#### 更复杂的查询

**依旧想要找到姓氏为“Smith”的员工，但是我们只想得 到年龄大于30岁的员工。我们的语句将添加过滤器(filter),它使得我们高效率的执行一个结构 化搜索：**

**GET /megacorp/employee/\_search { "query" : { "filtered" : { "filter" : { "range" : { "age" : { "gt" : 30 } <1> } }, "query" : { "match" : { "last\_name" : "smith" <2> } } } } }**

**<1> 这部分查询属于区间过滤器(range filter),它用于查找所有年龄大于30岁的数据—— gt 为"greater than"的缩写。**

**<2> 这部分查询与之前的 match 语句(query)一致。**

**只要知道我们添加了一个过滤器 (filter)用于执行区间搜索，然后重复利用了之前的 match 语句。现在我们的搜索结果只显示 了一个32岁且名字是“Jane Smith”的员工：**

**{ ... "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" ] } } ] } }**

### 全文搜索

到目前为止搜索都很简单：搜索特定的名字，通过年龄筛选。让我们尝试一种更高级的搜 索，全文搜索——一种传统数据库很难实现的功能

我们将会搜索所有喜欢“rock climbing”的员工：

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

你可以看到我们使用了之前的 match 查询，从 about 字段中搜索"rock climbing"，我们得到 了两个匹配文档：

{ ... "hits": { "total": 2, "max\_score": 0.16273327, "hits": [ { ... "\_score": 0.16273327, <1> "\_source": { "first\_name": "John", "last\_name": "Smith", "age": 25, "about": "I love to go rock climbing", "interests": [ "sports", "music" ] } }, { ... "\_score": 0.016878016, <2> "\_source": { "first\_name": "Jane", "last\_name": "Smith", "age": 32, "about": "I like to collect rock albums", "interests": [ "music" ] } } ] } }

<1><2> 结果相关性评分。

默认情况下，Elasticsearch根据结果相关性评分来对结果集进行排序，所谓的「结果相关性 评分」就是文档与查询条件的匹配程度。很显然，排名第一的 John Smith 的 about 字段明确 的写到“rock climbing”。 但是为什么 Jane Smith 也会出现在结果里呢？原因是“rock”在她的 abuot 字段中被提及了。 因为只有“rock”被提及而“climbing”没有，所以她的 \_score 要低于John。

这个例子很好的解释了Elasticsearch如何在各种文本字段中进行全文搜索，并且返回相关性 最大的结果集。相关性(relevance)的概念在Elasticsearch中非常重要，而这个概念在传统关 系型数据库中是不可想象的，因为传统数据库对记录的查询只有匹配或者不匹配。

#### 短语搜索

目前我们可以在字段中搜索单独的一个词，这挺好的，但是有时候你想要确切的匹配若干个 单词或者短语(phrases)。例如我们想要查询同时包含"rock"和"climbing"（并且是相邻的）的 员工记录。

要做到这个，我们只要将 match 查询变更为 match\_phrase 查询即可:

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

毫无疑问，该查询返回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" ] } } ] } }

#### 高亮我们的搜索

很多应用喜欢从每个搜索结果中高亮(highlight)匹配到的关键字，这样用户可以知道为什么这 些文档和查询相匹配。在Elasticsearch中高亮片段是非常容易的。

让我们在之前的语句上增加 highlight 参数：

Elasticsearch权威指南（中文版）

36搜 索

GET /megacorp/employee/\_search { "query" : { "match\_phrase" : { "about" : "rock climbing" } }, "highlight": { "fields" : { "about" : {} } } }

当我们运行这个语句时，会命中与之前相同的结果，但是在返回结果中会有一个新的部分叫 做 highlight ，这里包含了来自 about 字段中的文本，并且用 <em></em> 来标识匹配到的单 词。

{ ... "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>" <1> ] } } ] } }

<1> 原有文本中高亮的片段

## 分析

最后，我们还有一个需求需要完成：允许管理者在职员目录中进行一些分析。 Elasticsearch 有一个功能叫做聚合(aggregations)，它允许你在数据上生成复杂的分析统计。它很像SQL 中的 GROUP BY 但是功能更强大。

举个例子，让我们找到所有职员中最大的共同点（兴趣爱好）是什么：

GET /megacorp/employee/\_search { "aggs": { "all\_interests": { "terms": { "field": "interests" } } } }

暂时先忽略语法只看查询结果：

{ ... "hits": { ... }, "aggregations": { "all\_interests": { "buckets": [ { "key": "music", "doc\_count": 2 }, { "key": "forestry", "doc\_count": 1 }, { "key": "sports", "doc\_count": 1 } ] } } }

我们可以看到两个职员对音乐有兴趣，一个喜欢林学，一个喜欢运动。这些数据并没有被预 先计算好，它们是实时的从匹配查询语句的文档中动态计算生成的。如果我们想知道所有 姓"Smith"的人最大的共同点（兴趣爱好），我们只需要增加合适的语句既可：

Elasticsearch权威指南（中文版）

38聚 合

GET /megacorp/employee/\_search { "query": { "match": { "last\_name": "smith" } }, "aggs": { "all\_interests": { "terms": { "field": "interests" } } } }

all\_interests 聚合已经变成只包含和查询语句相匹配的文档了：

... "all\_interests": { "buckets": [ { "key": "music", "doc\_count": 2 }, { "key": "sports", "doc\_count": 1 } ] }

聚合也允许分级汇总。例如，让我们统计每种兴趣下职员的平均年龄：

GET /megacorp/employee/\_search { "aggs" : { "all\_interests" : { "terms" : { "field" : "interests" }, "aggs" : { "avg\_age" : { "avg" : { "field" : "age" } } } } } }

Elasticsearch权威指南（中文版）

39聚 合

虽然这次返回的聚合结果有些复杂，但任然很容易理解：

... "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 } } ] }

该聚合结果比之前的聚合结果要更加丰富。我们依然得到了兴趣以及数量（指具有该兴趣的 员工人数）的列表，但是现在每个兴趣额外拥有 avg\_age 字段来显示具有该兴趣员工的平均 年龄。

即使你还不理解语法，但你也可以大概感觉到通过这个特性可以完成相当复杂的聚合工作， 你可以处理任何类型的数据

**这只是一些皮毛，为了保持 简短，还有很多的特性未提及——像推荐、定位、渗透、模糊以及部分匹配等。但这也突出 了构建高级搜索功能是多么的容易。无需配置，只需要添加数据然后开始搜索**

## 分布式的特性

Elasticsearch可以扩展到上百（甚至上千）的服务器来处理PB级的数据，Elasticsearch为分布式而生，而且它的设计隐藏了分布式本身的复杂性。

Elasticsearch在分布式概念上做了很大程度上的透明化，在教程中你不需要知道任何关于分 布式系统、分片、集群发现或者其他大量的分布式概念。所有的教程你既可以运行在你的笔 记本上，也可以运行在拥有100个节点的集群上，其工作方式是一样的。

Elasticsearch致力于隐藏分布式系统的复杂性。以下这些操作都是在底层自动完成的：

将你的文档分区到不同的容器或者分片(shards)中，它们可以存在于一个或多个节点 中。 将分片均匀的分配到各个节点，对索引和搜索做负载均衡。 冗余每一个分片，防止硬件故障造成的数据丢失。 将集群中任意一个节点上的请求路由到相应数据所在的节点。 无论是增加节点，还是移除节点，分片都可以做到无缝的扩展和迁移

不懂这些内部机制也可以使用Elasticsearch的。但是这些能够帮助 你更深入和完整的了解Elasticsearch。你可以略读它们，然后在你需要更深入的理解时再回 头翻阅。

## 集群内部工作方式

Elasticsearch用于构建高可用和可扩展的系统。扩展的方式可以是购买更好的服务器(纵向扩 展(vertical scale or scaling up))或者购买更多的服务器（横向扩展(horizontal scale or scaling out)）。

Elasticsearch虽然能从更强大的硬件中获得更好的性能，但是纵向扩展有它的局限性。真正 的扩展应该是横向的，它通过增加节点来均摊负载和增加可靠性。

对于大多数数据库而言，横向扩展意味着你的程序将做非常大的改动才能利用这些新添加的 设备。对比来说，Elasticsearch天生就是分布式的：它知道如何管理节点来提供高扩展和高 可用。这意味着你的程序不需要关心这些。

### 空集群

如果我们启动一个单独的节点，它还没有数据和索引，这个集群看起来就像图1。



一个节点(node)就是一个Elasticsearch实例，而一个集群(cluster)由一个或多个节点组成， 它们具有相同的 cluster.name ，它们协同工作，分享数据和负载。当加入新的节点或者删除 一个节点时，集群就会感知到并平衡数据。

集群中一个节点会被选举为主节点(master),它将临时管理集群级别的一些变更，例如新建或 删除索引、增加或移除节点等。主节点不参与文档级别的变更或搜索，这意味着在流量增长 的时候，该主节点不会成为集群的瓶颈。任何节点都可以成为主节点。我们例子中的集群只 有一个节点，所以它会充当主节点的角色。

做为用户，我们能够与集群中的任何节点通信，包括主节点。每一个节点都知道文档存在于 哪个节点上，它们可以转发请求到相应的节点上。我们访问的节点负责收集各节点返回的数 据，最后一起返回给客户端。这一切都由Elasticsearch处理。

### 集群健康

在Elasticsearch集群中可以监控统计很多信息，但是只有一个是最重要的：集群健康(cluster health)。集群健康有三种状态： green 、 yellow 或 red 。

GET /\_cluster/health

在一个没有索引的空集群中运行如上查询，将返回这些信息：

{ "cluster\_name": "elasticsearch", "status": "green", <1> "timed\_out": false, "number\_of\_nodes": 1, "number\_of\_data\_nodes": 1, "active\_primary\_shards": 0, "active\_shards": 0, "relocating\_shards": 0, "initializing\_shards": 0, "unassigned\_shards": 0 }

<1> status 是我们最感兴趣的字段

status 字段提供一个综合的指标来表示集群的的服务状况。三种颜色各自的含义：

颜色 意义

green 所有主要分片和复制分片都可用

yellow 所有主要分片可用，但不是所有复制分片都可用

red 不是所有的主要分片都可用

### 添加索引

为了将数据添加到Elasticsearch，我们需要索引(index)——一个存储关联数据的地方。实际 上，索引只是一个用来指向一个或多个分片(shards)的“逻辑命名空间(logical namespace)”. 一个分片(shard)是一个最小级别“工作单元(worker unit)”,它只是保存了索引中所有数据的一 部分。在接下来的《深入分片》一章，我们将详细说明分片的工作原理，但是现在我们只要 知道分片就是一个Lucene实例，并且它本身就是一个完整的搜索引擎。我们的文档存储在分 片中，并且在分片中被索引，但是我们的应用程序不会直接与它们通信，取而代之的是，直 接与索引通信。

分片是Elasticsearch在集群中分发数据的关键。把分片想象成数据的容器。文档存储在分片 中，然后分片分配到你集群中的节点上。当你的集群扩容或缩小，Elasticsearch将会自动在 你的节点间迁移分片，以使集群保持平衡。

分片可以是主分片(primary shard)或者是复制分片(replica shard)。你索引中的每个文档属 于一个单独的主分片，所以主分片的数量决定了索引最多能存储多少数据。

理论上主分片能存储的数据大小是没有限制的，限制取决于你实际的使用情况。分片的 最大容量完全取决于你的使用状况：硬件存储的大小、文档的大小和复杂度、如何索引 和查询你的文档，以及你期望的响应时间。

复制分片只是主分片的一个副本，它可以防止硬件故障导致的数据丢失，同时可以提供读请 求，比如搜索或者从别的shard取回文档。

当索引创建完成的时候，主分片的数量就固定了，但是复制分片的数量可以随时调整。

让我们在集群中唯一一个空节点上创建一个叫做 blogs 的索引。默认情况下，一个索引被分 配5个主分片，但是为了演示的目的，我们只分配3个主分片和一个复制分片（每个主分片都 有一个复制分片）：

PUT /blogs { "settings" : { "number\_of\_shards" : 3, "number\_of\_replicas" : 1 } }

Elasticsearch权威指南（中文版）

47添 加索引

附带索引的单一节点集群：

我们的集群现在看起来就像上图——三个主分片都被分配到 Node 1 。如果我们现在检查集群 健康(cluster-health)，我们将见到以下信息：

{ "cluster\_name": "elasticsearch", "status": "yellow", <1> "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 <2> }

<1> 集群的状态现在是 yellow <2> 我们的三个复制分片还没有被分配到节点上

集群的健康状态 yellow 表示所有的主分片(primary shards)启动并且正常运行了——集群已 经可以正常处理任何请求——但是复制分片(replica shards)还没有全部可用。事实上所有的 三个复制分片现在都是 unassigned 状态——它们还未被分配给节点。在同一个节点上保存相 同的数据副本是没有必要的，如果这个节点故障了，那所有的数据副本也会丢失。

现在我们的集群已经功能完备，但是依旧存在因硬件故障而导致数据丢失的风险。

### 增加故障转移

在单一节点上运行意味着有单点故障的风险——没有数据备份。幸运的是，要防止单点故 障，我们唯一需要做的就是启动另一个节点。

启动第二个节点

为了测试在增加第二个节点后发生了什么，你可以使用与第一个节点相同的方式启动第 二个节点（《运行Elasticsearch》一章），而且命令行在同一个目录——一个节点可以 启动多个Elasticsearch实例。

只要第二个节点与第一个节点有相同的 cluster.name （请 看 ./config/elasticsearch.yml 文件），它就能自动发现并加入第一个节点所在的集群。 如果没有，检查日志找出哪里出了问题。这可能是网络广播被禁用，或者防火墙阻止了 节点通信。

如果我们启动了第二个节点，这个集群看起来就像下图。

双节点集群——所有的主分片和复制分片都已分配:

第二个节点已经加入集群，三个复制分片(replica shards)也已经被分配了——分别对应三个 主分片，这意味着在丢失任意一个节点的情况下依旧可以保证数据的完整性。

文档的索引将首先被存储在主分片中，然后并发复制到对应的复制节点上。这可以确保我们 的数据在主节点和复制节点上都可以被检索。

cluster-health 现在的状态是 green ，这意味着所有的6个分片（三个主分片和三个复制分 片）都已可用：

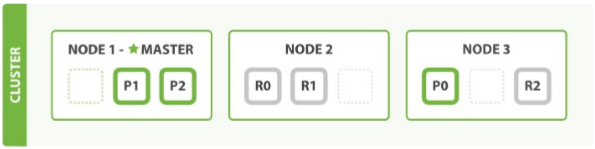
{ "cluster\_name": "elasticsearch", "status": "green", <1> "timed\_out": false, "number\_of\_nodes": 2, "number\_of\_data\_nodes": 2, "active\_primary\_shards": 3, "active\_shards": 6, "relocating\_shards": 0, "initializing\_shards": 0, "unassigned\_shards": 0 }

<1> 集群的状态是 green .

我们的集群不仅是功能完备的，而且是高可用的

### 横向扩展

随着应用需求的增长，我们该如何扩展？如果我们启动第三个节点，我们的集群会重新组织 自己，就像图4： 图4：包含3个节点的集群——分片已经被重新分配以平衡负载：



Node3 包含了分别来自 Node 1 和 Node 2 的一个分片，这样每个节点就有两个分片，和之前 相比少了一个，这意味着每个节点上的分片将获得更多的硬件资源（CPU、RAM、I/O）。

分片本身就是一个完整的搜索引擎，它可以使用单一节点的所有资源。我们拥有6个分片（3 个主分片和三个复制分片），最多可以扩展到6个节点，每个节点上有一个分片，每个分片可 以100%使用这个节点的资源。

### 继续扩展

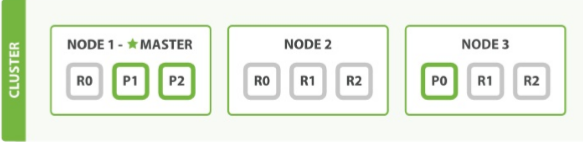
如果我们要扩展到6个以上的节点，要怎么做？

主分片的数量在创建索引时已经确定。实际上，这个数量定义了能存储到索引里数据的最大 数量（实际的数量取决于你的数据、硬件和应用场景）。然而，主分片或者复制分片都可以 处理读请求——搜索或文档检索，所以数据的冗余越多，我们能处理的搜索吞吐量就越大。

复制分片的数量可以在运行中的集群中动态地变更，这允许我们可以根据需求扩大或者缩小 规模。让我们把复制分片的数量从原来的 1 增加到 2 ：

PUT /blogs/\_settings { "number\_of\_replicas" : 2 }

图5：增加 number\_of\_replicas 到2：



从图中可以看出， blogs 索引现在有9个分片：3个主分片和6个复制分片。这意味着我们能够 扩展到9个节点，再次变成每个节点一个分片。这样使我们的搜索性能相比原始的三节点集群 增加三倍。

当然，在同样数量的节点上增加更多的复制分片并不能提高性能，因为这样做的话平均 每个分片的所占有的硬件资源就减少了（译者注：大部分请求都聚集到了分片少的节 点，导致一个节点吞吐量太大，反而降低性能），你需要增加硬件来提高吞吐量。

不过这些额外的复制节点使我们有更多的冗余：通过以上对节点的设置，我们能够承受 两个节点故障而不丢失数据。

### 应对故障

我们已经说过Elasticsearch可以应对节点失效，所以让我们继续尝试。如果我们杀掉第一个 节点的进程（以下简称杀掉节点），我们的集群看起来就像这样：

图5：杀掉第一个节点后的集群



我们杀掉的节点是一个主节点。一个集群必须要有一个主节点才能使其功能正常，所以集群 做的第一件事就是各节点选举了一个新的主节点： Node 2 。

主分片 1 和 2 在我们杀掉 Node 1 时已经丢失，我们的索引在丢失主分片时不能正常工作。 如果此时我们检查集群健康，我们将看到状态 red ：不是所有主分片都可用！

幸运的是丢失的两个主分片的完整拷贝存在于其他节点上，所以新主节点做的第一件事是把 这些在 Node 2 和 Node 3 上的复制分片升级为主分片，这时集群健康回到 yellow 状态。这个 提升是瞬间完成的，就好像按了一下开关。

为什么集群健康状态是 yellow 而不是 green ？我们有三个主分片，但是我们指定了每个主分 片对应两个复制分片，当前却只有一个复制分片被分配，这就是集群状态无法达到 green 的 原因，不过不用太担心这个：当我们杀掉 Node 2 ，我们的程序依然可以在没有丢失数据的情 况下继续运行，因为 Node 3 还有每个分片的拷贝。

如果我们重启 Node 1 ，集群将能够重新分配丢失的复制分片，集群状况与上一节的 图5：增 加number\_of\_replicas到2 类似。如果 Node 1 依旧有旧分片的拷贝，它将会尝试再利用它 们，它只会从主分片上复制在故障期间有数据变更的那一部分。

现在你应该对分片如何使Elasticsearch可以水平扩展并保证数据安全有了一个清晰的认识。

### 数据吞吐（json）

无论程序怎么写，意图是一样的：组织数据为我们的目标所服务。但数据并不只是由随机比 特和字节组成，我们在数据节点间建立关联来表示现实世界中的实体或者“某些东西”。属于同 一个人的名字和Email地址会有更多的意义。

在现实世界中，并不是所有相同类型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，另一个人可能只有一个手机号码，有些人可能两者都有。一个人可能有三个Email地 址，其他人可能没有。西班牙人可能有两个姓氏，但是英国人（英语系国家的人）可能只有 一个。

面向对象编程语言流行的原因之一，是我们可以用对象来表示和处理现实生活中那些有着潜 在关系和复杂结构的实体。到目前为止，这种方式还不错。

但当我们想存储这些实体时问题便来了。传统上，我们以行和列的形式把数据存储在关系型 数据库中，相当于使用电子表格。这种固定的存储方式导致对象的灵活性不复存在了。

但是如何能以对象的形式存储对象呢？相对于围绕表格去为我们的程序去建模，我们可以专 注于使用数据，把对象本来的灵活性找回来。

对象(object)是一种语言相关，记录在内存中的的数据结构。为了在网络间发送，或者存储 它，我们需要一些标准的格式来表示它。JSON (JavaScript Object Notation)是一种可读的以 文本来表示对象的方式。它已经成为NoSQL世界中数据交换的一种事实标准。当对象被序列 化为JSON，它就成为JSON文档(JSON document)了。 Elasticsearch是一个分布式的文档(document)存储引擎。它可以实时存储并检索复杂数据结 构——序列化的JSON文档。换言说，一旦文档被存储在Elasticsearch中，它就可以在集群的 任一节点上被检索。

当然，我们不仅需要存储数据，还要快速的批量查询。虽然已经有很多NoSQL的解决方案允 许我们以文档的形式存储对象，但它们依旧需要考虑如何查询这些数据，以及哪些字段需要 被索引以便检索时更加快速。

在Elasticsearch中，每一个字段的数据都是默认被索引的。也就是说，每个字段专门有一个 反向索引用于快速检索。而且，与其它数据库不同，它可以在同一个查询中利用所有的这些 反向索引，以惊人的速度返回结果。

在这一章我们将探讨如何使用API来创建、检索、更新和删除文档。目前，我们并不关心数据 如何在文档中以及如何查询他们。所有我们关心的是文档如何安全在Elasticsearch中存储， 以及如何让它们返回

#### 什么是文档？

程序中大多的实体或对象能够被序列化为包含键值对的JSON对象，键(key)是字段(field)或属 性(property)的名字，值(value)可以是字符串、数字、布尔类型、另一个对象、值数组或者 其他特殊类型，比如表示日期的字符串或者表示地理位置的对象。

{ "name": "John Smith", "age": 42, "confirmed": true, "join\_date": "2014-06-01", "home": { "lat": 51.5, "lon": 0.1 }, "accounts": [ { "type": "facebook", "id": "johnsmith" }, { "type": "twitter", "id": "johnsmith" } ] }

通常，我们可以认为对象(object)和文档(document)是等价相通的。不过，他们还是有所差 别：对象(Object)是一个JSON结构体——类似于哈希、hashmap、字典或者关联数组；对象 (Object)中还可能包含其他对象(Object)。 在Elasticsearch中，文档(document)这个术语有着 特殊含义。它特指最顶层结构或者根对象(root object)序列化成的JSON数据（以唯一ID标识 并存储于Elasticsearch中）。

#### 文档元数据

索引(index)类似于关系型数据库里的“数据库”——它是我们存储和索引关联数据的地方。

提示：

事实上，我们的数据被存储和索引在分片(shards)中，索引只是一个把一个或多个分片 分组在一起的逻辑空间。然而，这只是一些内部细节——我们的程序完全不用关心分 片。对于我们的程序而言，文档存储在索引(index)中。剩下的细节由Elasticsearch关心 既可。

我们将会在《索引管理》章节中探讨如何创建并管理索引，但现在，我们将让Elasticsearch 为我们创建索引。我们唯一需要做的仅仅是选择一个索引名。这个名字必须是全部小写，不 能以下划线开头，不能包含逗号。让我们使用 website 做为索引名。

\_type

在应用中，我们使用对象表示一些“事物”，例如一个用户、一篇博客、一个评论，或者一封邮 件。每个对象都属于一个类(class)，这个类定义了属性或与对象关联的数据。 user 类的对象 可能包含姓名、性别、年龄和Email地址。

在关系型数据库中，我们经常将相同类的对象存储在一个表里，因为它们有着相同的结构。 同理，在Elasticsearch中，我们使用相同类型(type)的文档表示相同的“事物”，因为他们的数 据结构也是相同的。

每个类型(type)都有自己的映射(mapping)或者结构定义，就像传统数据库表中的列一样。所 有类型下的文档被存储在同一个索引下，但是类型的映射(mapping)会告诉Elasticsearch不同 的文档如何被索引。 我们将会在《映射》章节探讨如何定义和管理映射，但是现在我们将依 赖Elasticsearch去自动处理数据结构。

\_type 的名字可以是大写或小写，不能包含下划线或逗号。我们将使用 blog 做为类型名。

\_id

id仅仅是一个字符串，它与 \_index 和 \_type 组合时，就可以在Elasticsearch中唯一标识一个 文档。当创建一个文档，你可以自定义 \_id ，也可以让Elasticsearch帮你自动生成。

Elasticsearch权威指南（中文版）

56文 档

其它元数据

还有一些其它的元数据，我们将在《映射》章节探讨。使用上面提到的元素，我们已经可以 在Elasticsearch中存储文档并通过ID检索——换言说，把Elasticsearch做为文档存储器使用 了

#### 索引一个文档

文档通过 index API被索引——使数据可以被存储和搜索。但是首先我们需要决定文档所 在。正如我们讨论的，文档通过其 \_index 、 \_type 、 \_id 唯一确定。们可以自己提供一 个 \_id ，或者也使用 index API 为我们生成一个。

##### 使用自己的ID

如果你的文档有自然的标识符（例如 user\_account 字段或者其他值表示文档），你就可以提 供自己的 \_id ，使用这种形式的 index API：

PUT /{index}/{type}/{id} { "field": "value", ... }

例如我们的索引叫做 “website” ，类型叫做 “blog” ，我们选择的ID是 “123” ，那么这个索引 请求就像这样：

PUT /website/blog/123 { "title": "My first blog entry", "text": "Just trying this out...", "date": "2014/01/01" }

Elasticsearch的响应：

{ "\_index": "website", "\_type": "blog", "\_id": "123", "\_version": 1, "created": true }

响应指出请求的索引已经被成功创建，这个索引中包含 \_index 、 \_type 和 \_id 元数据，以 及一个新元素： \_version 。

Elasticsearch中每个文档都有版本号，每当文档变化（包括删除）都会使 \_version 增加。在 《版本控制》章节中我们将探讨如何使用 \_version 号确保你程序的一部分不会覆盖掉另一部 分所做的更改。

##### 自增ID

如果我们的数据没有自然ID，我们可以让Elasticsearch自动为我们生成。请求结构发生了变 化： PUT 方法—— “在这个URL中存储文档” 变成了 POST 方法—— "在这个类型下存储文档" 。（译者 注：原来是把文档存储到某个ID对应的空间，现在是把这个文档添加到某个 \_type 下）。

URL现在只包含 \_index 和 \_type 两个字段：

POST /website/blog/ { "title": "My second blog entry", "text": "Still trying this out...", "date": "2014/01/01" }

响应内容与刚才类似，只有 \_id 字段变成了自动生成的值：

{ "\_index": "website", "\_type": "blog", "\_id": "wM0OSFhDQXGZAWDf0-drSA", "\_version": 1, "created": true }

自动生成的ID有22个字符长，URL-safe, Base64-encoded string universally unique identifiers, 或者叫 UUIDs。

### 检索文档