**Elasticsearch文档**

|  |  |  |  |
| --- | --- | --- | --- |
| 版本 | 发布时间 | 修订章节 | 作者 |
| 1.0 | 2016.11.30 | 第一版 | 刘吴强 |
|  |  |  |  |
|  |  |  |  |

* 手册目的

ElasticSearch学习资料

* 声明

无

* 名词定义和缩略语说明

表 2 名词定义及缩略语说明

| 序号 | 缩写 | 说明 |
| --- | --- | --- |
| 1 | ES | Elasticsearch，一种设计用于云计算的分布式全文索引解决方案。 |
|  |  |  |
|  |  |  |

目录

[ 参考资料 2](#_Toc468269579)

[ 手册目的 2](#_Toc468269580)

[ 声明 2](#_Toc468269581)

[ 名词定义和缩略语说明 2](#_Toc468269582)

[1、总述 4](#_Toc468269583)

[1.1、简介 4](#_Toc468269584)

[1.2、国外使用案例 4](#_Toc468269585)

[1.3、Scaling Lucene 5](#_Toc468269586)

[1.1.1. Building Blocks 6](#_Toc468269587)

[1.1.2. Partitioning 6](#_Toc468269588)

[1.1.3. Replication 8](#_Toc468269589)

[1.1.4. Transaction Log 9](#_Toc468269590)

[2、服务器搭建 10](#_Toc468269591)

[3、Moduls 10](#_Toc468269592)

[3.1、Cluster 10](#_Toc468269593)

[3.2、Shards 10](#_Toc468269594)

[3.3、Replicas 10](#_Toc468269595)

[3.4、Recovery 11](#_Toc468269596)

[3.5、River 11](#_Toc468269597)

[3.6、Gateway 11](#_Toc468269598)

[3.7、discovery.zen 11](#_Toc468269599)

[3.8、Transport 11](#_Toc468269600)

[4、Java API 11](#_Toc468269601)

[4.1与集群交互 11](#_Toc468269602)

[4.1.1、Node方式 12](#_Toc468269603)

[4.1.2、TransportClient方式 12](#_Toc468269604)

# 1、总述

## 1.1、简介

ElasticSearch是一个基于Lucene的搜索服务器。它提供了一个分布式多用户能力的全文搜索引擎，基于RESTful web接口。Elasticsearch是用Java开发的，并作为Apache许可条款下的开放源码发布，是当前流行的企业级搜索引擎。设计用于云计算中，能够达到实时搜索，稳定，可靠，快速，安装使用方便。

我们建立一个网站或应用程序，并要添加搜索功能，令我们受打击的是：搜索工作是很难的。我们希望我们的搜索解决方案要快，我们希望有一个零配置和一个完全免费的搜索模式，我们希望能够简单地使用JSON通过HTTP的索引数据，我们希望我们的搜索服务器始终可用，我们希望能够一台开始并扩展到数百，我们要实时搜索，我们要简单的多租户，我们希望建立一个云的解决方案。Elasticsearch旨在解决所有这些问题和更多的问题。

## 1.2、国外使用案例

**Github**

“Github使用Elasticsearch搜索20TB的数据，包括13亿的文件和1300亿行的代码”

这个不用介绍了吧，码农们都懂的，Github在2013年1月升级了他们的代码搜索，由solr转为elasticsearch，目前集群规模为26个索引存储节点和8个客户端节点（负责处理搜索请求），详情请看官方博客https://github.com/blog/1381-a-whole-new-code-search

**Foursquare**

”实时搜索5千万地点信息？Foursquare每天都用Elasticsearch做这样的事“

Foursquare是一家基于用户地理位置信息的手机服务网站，并鼓励手机用户同他人分享自己当前所在地理位置等信息。与其他老式网站不同，Foursquare用户界面主要针对手机而设计，以方便手机用户使用。

**SoundCloud**

“SoundCloud使用Elasticsearch来为1.8亿用户提供即时精准的音乐搜索服务”

SoundCloud是一家德国网站，提供音乐分享社区服务，成长很快，Alexa世界排名已达第236位。你可以在线录制或上传任何声音到SoundCloud与大家分享，可在线上传也可以通过软件客户端来上传音乐文件，没有文件大小限制，但免费版限制上传音频总长不可超过2个小时播放时长，每首歌曲限最多100次下载。SoundCloud允许音乐通过Flash播放器方式嵌入到网页中。

**Fog Creek**

“Elasticsearch使Fog Creek可以在400亿行代码中进行一个月3千万次的查询“

**StumbleUpon**

”Elasticsearch是StumbleUpon的关键部件，它每天为社区提供百万次的推荐服务“

StumbleUpon是个能发现你喜欢的网页的网站，进去时先注册，注册完就选择你感兴趣的东西，它会自动帮你推荐一些网页，如果你喜欢这个网页就点喜欢按钮，按 stumble按钮就会推荐下一个网页。

目前其数据量达到 25亿，基本数据存储在HBase中，并用 elasticsearch建立索引，elasticsearch 在其中除了用在搜索功能还有在推荐和统计功能。之前他们是使用solr作为搜索，由于solr满足不了他们的业务增长需要而替换为 elasticsearch。

**Mozilla**

Mozilla公司以火狐著名，它目前使用 WarOnOrange 这个项目来进行单元或功能测试，测试的结果以 json的方式索引到elasticsearch中，开发人员可以非常方便的查找 bug。

Socorro是Mozilla 公司的程序崩溃报告系统，一有错误信息就插入到 Hbase和Postgres 中，然后从 Hbase中读取数据索引到elasticsearch中，方便查找。

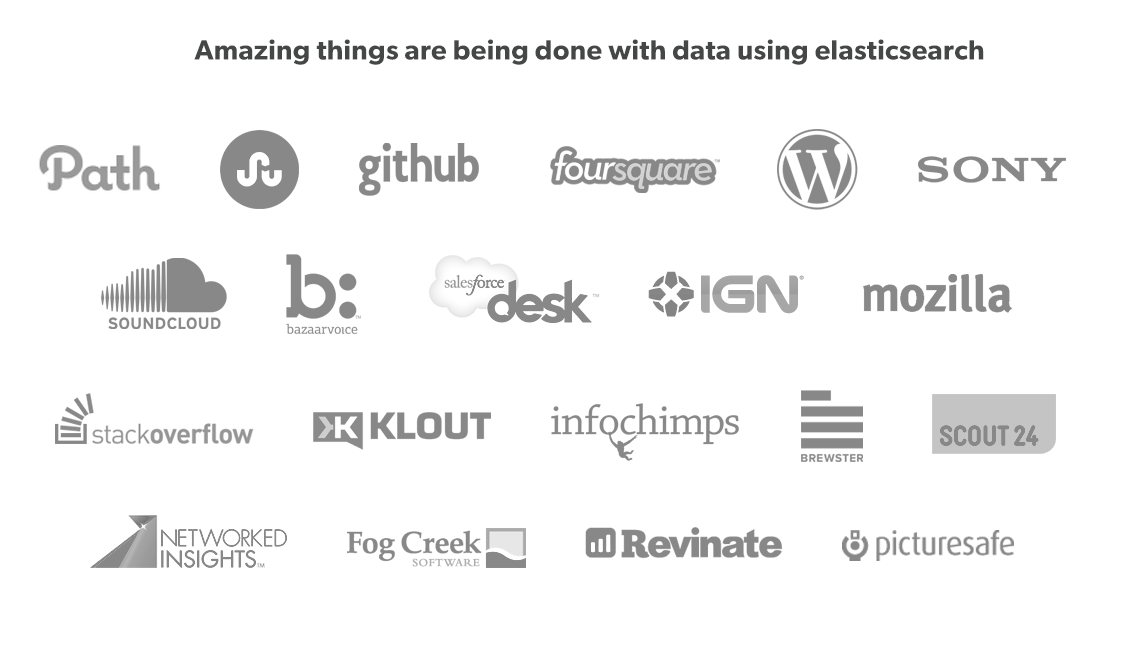
**Sony**

Sony公司使用elasticsearch 作为信息搜索引擎

**Infochimps**

“在 Infochimps，我们已经索引了25亿文档，总共占用 4TB的空间”。

Infochimps是一家位于德克萨斯州奥斯丁的创业公司，为大数据平台提供商。它主要提供基于hadoop的大数据处理方案。



## 1.3、Scaling Lucene

怎样在Lucene之上构建一个分布式、高度伸缩、接近实时的搜索引擎呢?

让我们回顾一下在搜索引擎（基于lucene）伸缩性这条路上都做了那些尝试，并且elasticsearch是如何尝试并去解决这些挑战的。

首先我们了解下最基础的理论知识 **building blocks** （这些理论基础是构建分布式近实时搜索引擎的基础）。 接着我们研究一下到底哪种才是最佳的分区策略 **partitioning** （将lucene索引文档分割到多个分布式的分片中去）。 然后我们同样需要决定使用哪种分区复制方式 **replication** （复制能够保证系统的高可用以及提高搜索的吞吐）。 最后，我们再看一下事务日志 **transaction log** （事务日志在elasticsearch里面是一个保证数据一致性的非常酷的功能）。

### Building Blocks

当我们要构建一个分布式接近实时的搜索引擎，并且要让lucene可伸缩可扩展，必须首先知道lucene的关键概念以及它们与我们要达成目标的一些局限性.

* **Directory**

Lucene Directory 是一个抽象的文件系统的接口，用来允许你读写文件，不管lucene的索引是存放在内存中还是在物理磁盘上，它都是通过lucene的Directory抽象层来访问和维护的。

* **IndexWriter**

IndexWriter 用来添加、删除和更新lucene里面的索引文档。这些操作是在内存中完成以保证更好的性能，但是如果要保证这些操作的持久化，这些操作是需要flush到磁盘的。并且，flush操作或者是显式的commit提交开销都是比较大的，因为这些操作通常需要处理很多的文件，而处理这些文件又涉及到大量的磁盘io。  
此外, 每次只能有一个IndexWriter对象来对一个索引目录进行索引操作，并且创建这个对象的开销很大，所以必须尽可能去重用这个对象.

* **Index Segments**

Lucene 索引被分解为很多段（segments）。每个索引段实际上是一个功能完整的lucene索引，一旦一个索引段创建完成，它将是不可变的，并且不能删除段里面的索引文档。commit提交操作用来往索引里面添加一个新段。lucene内部会来对这些段进行合并，所以我们必须要有策略来控制这些合并(MergePolisy, MergeScheuler, … etc)。Because segments need to be kept at bay they are being merged continuously by internal Lucene processes (MergePolisy, MergeScheuler, … etc)。

因为段是不可变的，所以用来做缓存（caching）是一个很好的选择，你可以加载所有的term词条并且创建一个跳跃列表（ skip lists ) ，或者用来构造FieldCache，如果段没有变化，你就不需要重新加载。

* **IndexReader**

IndexReader 用来执行搜索索引。这个对象通过IndexWriter来提供，并且创建代价也是比较高。一旦IndexReader打开之后，它就不能够发现打开之后的索引变化，如果要知道这些由IndexWriter产生的索引变化，除非刷新IndexReader对象（当然前提需要flush操作）。搜索操作在内部其实是按段来进行的（每次一个段）.

* **Near Real-Time Search**

获取一个新的IndexReader开销很大，所以也是我们不能每次一有索引操作就真的去获取一个新的IndexReader，你可以隔一段时间去刷新一下，比如每隔一秒钟等等，这也是我们在这里称之为接近实时( near real-time )的原因.

### Partitioning

可能用来伸缩Lucene的途径（Possible approach to Scale Lucene）：

* **Distributed Directory**

其中一个途径用来伸缩Lucene就是使用分布式文件系统，大文件会被拆分成chunks块并且会保存到分布式存储系统（比如 Coherence, Terracota, GigaSpaces or Infinispan等等)。这样IndexWriter和IndexReader都是工作在一个自定义的Directory分布式实现上，每个操作后面其实是分布了很多个节点，每个节点上面存储了索引文件的一部分.

**但是这种方案有一些问题**：

首先，这种方案会产生密集的网络流量。尽管可以用一些高级的方法如本地缓存等，但仍然会产生大量的网络请求，因为最主要的原因是因为这种将文件拆分为块的想法与lucene索引文件构建方式和使用方式实在相隔太远，结论就是使用这种方式来做大规模索引和搜索是不切实际的。（ps：所以solandra这种玩意还是不要去考虑了）。

其次，大的索引必然会使IndexReader变的无法分布式。IndexReader是一个很重的对象，并且term词条越多，其消耗的内存也会越多。

最后，索引操作也会变的非常困难，因为只有一个单一的IndexWriter能够写索引。所以，我们把目光投向另一种方式。

* **Partitioning**

有2种通过将数据分区方式来scale搜索引擎: **基于文档（Document based partitioning）** and **基于词条（Term based partitioning）**. Elasticsearch 使用的基于文档的分区方式。

**基于文档的分区（Document Based Partitioning）**

每一个文档只存一个分区，每个分区持有整个文档集的一个子集，分区是一个功能完整的索引。

**优点：**

每个分区都可以独立的处理查询。

可以非常简单的添加以文档为单位的索引信息。

网络开销很小，每个节点可以分别执行搜索，执行完了之后只需用返回文档的ID和评分信息就可以了，然后在其中一个我们执行分布式搜索的节点上执行合并就可以了。

**缺点：**

查询如果需要在所有的分区上执行，那么它将执行 O(K\*N) 次磁盘操作（K是词条（Term，或者理解为Field）的数量，N是分区的数量）。

在实用性的角度来看基于文档的分区方式已经被证明是一个构建大型的分布式信息检索系统的一种行之有效的方法, 关于这方面的详细内容，可以看 这里 talk by Jeffrey Dean (Google)。

**基于词条的分区（Term Based Partitioning）**

每个分区拥有一部分词条，词条里面包含了整个index的文档数据。

一些基于词条分区的系统，如Riak Search (built on top of Riak key-value store engine) 或是 Lucandra/Solandra (on top of Cassandra). 尽管这些系统不是完全一样，但是它们都面临一个相似的挑战，当然也得益于相同的设计理念。

**优点：**

一般来说，你只需要在很少的部分分区上执行查询就行了，比如，我们有5个term词条的查询，我们将至多命中5个分区，如果这5个term词条都保存同一个分区中，那么我们只需用访问一个分区即可，而不管我们是不是实际上有50个分区。

另外一个优势就是对应K个Term词条的查询，你只需用执行 O(K) 次磁盘查找（假设我们使用的优化过的实现）。

**缺点：**

最主要的问题是Lucene Segment概念里面固有的很多结构都将失去。

The main problem is that whole notion of Lucene Segment which is inherent to a lot of constructs in Lucene is lost.

对于那些复杂的查询，网络开销将会变得非常高，并且可能使得系统可用性大大降低，尤其是那些会expand出大量的term词条的查询，如fuzzy或者prefix查询。

另外一个问题就是获取每个文档的信息将会变得非常困难，举例来说，如果你想获取文档的一部分数据来做进一步的控制，比如（google的PageRank算法），获取每个文档的这些数据都会变得非常困难，因为这种分区的方式使得文档的数据被分散到了不同的地方，所以实现faceting、评分、自定义评分等等都将变得难以实现。

### Replication

分布式系统的另外一方面就是复制(replication)了。通过复制我们可以得到2个主要的好处：

High Availability (HA高可用性)。如果一个节点挂了，另外一个节点能从它趴下的地方应头顶上，如果一个节点上面持有索引分片，而另一个节点持有该分片的副本，那么我们的数据就有了一个备份。

拥有数据多个副本的另一个好处就是 scalability (可伸缩性)。我们没有理由不通过增加副本来提高搜索能力，而我们只需要简单的增加几个副本或从节点（slave nodes）就能提升我们搜索的吞吐，何乐而不为呢。

一般有两种方式来实现复制: **Push Replication（推模式）** 和 **Pull Replication（拉模式）**。 Elasticsearch 使用的是Push Replication(推模式)。

* **Push Replication**

工作起来非常简单， 当你往 [master] 主分片上面索引一个文档，该分片会复制该文档(document)到剩下的所有 [replica] 副本分片中，这些分片也会索引这个文档。

**缺点：**

同一个文档重复索引多次，相比拉模式而言，要传输相对较少的数据(众所周知，Lucene索引速度非常快)。

You index the same document several times， but we transfer much less data compared to Pull replication (and Lucene is known to index very fast)。

这就需要在并发索引的时候进行一些微妙的控制，比如对同一个文档进行非常频繁的索引，在主分片和副本分片上面执行索引操作的时候，你必须保证每次更新是按照正确的顺序，或者通过版本（versioning）来拒绝旧版本的操作，而拉模式就没有这个问题。

**优点：**

一旦文档索引完毕，那么该文档在所有的分片及副本上都是立即可用的。 索引操作会等待直到确认所有的副本也执行了同样的索引操作(注意: 如果需要，你也可以使用异步复制)。 这意味着索引的实时性。 然后你只需要 refresh 一下 IndexReader 就能搜索到新的数据了。

这样的架构能让你非常方便的在节点之间进行切换，假如包含主分片（primary shard）的节点挂了，我们能够很快的进行切换，因为其它的分片和主分片都是一模一样的。

* **Pull Replication**

拉模式是一种主从方式（master – slave）(Solr 用的就是这种)。 当一个文档在master上面进行索引，并且数据通过commit操作产生了新的段文件（segment），这个时候，从节点（slave）把这些段文件（segments）拉到自己的机器然后再执行相应的刷新操作，并保证lucene能够使用这些新的数据。

**缺点：**

需要在master上面执行commit操作来产生这些段文件（segment），这样slave才能够执行pull操作。 不知道你还记不记得前面说过，lucene的commit的开销是非常大的，如果可能，commit次数越少越好。

数据的传输会有不必要的冗余。 在分布式系统里面，网络通常来说是非常宝贵的资源(如果你跑在EC2上面，那将更加宝贵，$$$) 并且最终要移动的数据会越来越多，举例来说，如果你有2个段文件，里面包含了文档，文档里面的字段都是存储的（stored fields），并且Lucene决定要合并这2个段文件，那么你也必须要传输这部分数据（合并之后的段文件），因为这是一个新的段文件，但是实际上你传输的是一份相同的数据。

这将造成一个这样的局面，所有的slaves确实是在master后面。 也可能是确实没有理由每次都进行commit或者花大量时间来传输一个大的段文件。但是至少意味着你的slave会丢失 high availability，并且不可能当成是一个实时的slave（a real time high available slave）。 实时搜索不可能存在，并且（使用拉模式）也不可能有这种1秒的刷新率，然后lucene就能实时搜索。

### Transaction Log

正如前面提到过的，索引提交（commit）的开销实在太大，但是我们又必须通过提交操作来保证数据被可靠的持久化，如果拥有数据的节点突然崩溃的话，那么最后一次提交操作之后产生的数据操作将会丢失。

* **数据可靠性（Data Persistency）**

ElasticSearch通过使用 transaction log (或预写日志(write ahead log)) 来解决这个问题，通过日志记录发生在索引上的各种操作，来保证就算没有调用commit操作也能保证数据的持久化。并且能够很自然的支持推送复制（push replication），因为我们能够让每个不同的shard都拥有 transaction log ，就算某些节点崩溃了，如果有必要，可以很轻松对日志操作进行重放（replay）。

Transaction log 周期性的将数据刷新(flushed)到磁盘，你可以通过 参数 来进行控制。 简单来说就是保存两次提交之间的连续数据操作的记录。

尽管你只运行了一个elasticsearch的服务节点（可能暂时不需要分布式），trasncation log也能够使你的es即使被强制结束进程（ “kill -9” ）也不会丢失任何数据。

当然，还不止这些！Transaction log还有一个重要的功能就是可以保证当你生成快照（ shared gateway snapshot ）、分片恢复（ peer shard recovery ）或是分片热迁移（shard “Hot” relocation）的时候，索引数据不会丢失。

* **Shared Gateway Snapshot**

使用共享gateway时，会周期性的生成数据改变(changes)的快照 ( snapshots ) ，并存储到共享存储中（shared storage)，并且transaction log也是持久化数据的一部分。

* **Peer Shard Reovery**

当分片从一个节点迁移到另一个节点或者需要分配更多的分片(比如你 增加 了副本数) 的时候，数据会从某一个节点上取来进行恢复，而不是从gateway。

迁移数据时，首先我们保证不会删除Lucene的段文件（segment files)，然后禁用flushing操作，这个时候保证不调用commit操作，然后开始迁移这些段文件，这个时候产生的索引改变，我们存放到transaction log中，一旦这个步骤结束（ie：索引索引文件拷贝完毕），我们开始对transaction log里面的日志在replica分片上进行重放操作（replay），完毕之后，我们就可以进行切换了，数据迁移成功！

迁移操作进行时，你仍然可以进行索引，仍然可以进行搜索，只有索引切换的时候会有一段很短的时间阻塞（blocking），但是直到切换前，迁移对你来说是完全透明的。

## 1.4、Elastic中**索引**、**类型**、**文档**以及**字段**的关系

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）。

# 2、服务器搭建

<Elasticsearch集群搭建.docx>

# 3、Moduls

## 3.1、集群（Cluster）

代表一个集群，集群中有多个节点，其中有一个为主节点，这个主节点是可以通过选举产生的，主从节点是对于集群内部来说的。es的一个概念就是去中心化，字面上理解就是无中心节点，这是对于集群外部来说的，因为从外部来看es集群，在逻辑上是个整体，你与任何一个节点的通信和与整个es集群通信是等价的。

## 3.2、分片（Shards）

代表索引分片，es可以把一个完整的索引分成多个分片，这样的好处是可以把一个大的索引拆分成多个，分布到不同的节点上。构成分布式搜索。分片的数量只能在索引创建前指定，并且索引创建后不能更改。

## 3.3、副本（Replicas）

代表索引副本，es可以设置多个索引的副本，副本的作用一是提高系统的容错性，当个某个节点某个分片损坏或丢失时可以从副本中恢复。二是提高es的查询效率，es会自动对搜索请求进行负载均衡。

## 3.4、Recovery

代表数据恢复或叫数据重新分布，es在有节点加入或退出时会根据机器的负载对索引分片进行重新分配，挂掉的节点重新启动时也会进行数据恢复。

## 3.5、River

代表es的一个数据源，也是其它存储方式（如：数据库）同步数据到es的一个方法。它是以插件方式存在的一个es服务，通过读取river中的数据并把它索引到es中，官方的river有couchDB的，RabbitMQ的，Twitter的，Wikipedia的，river这个功能将会在后面的文件中重点说到。

## 3.6、Gateway

代表es索引的持久化存储方式，es默认是先把索引存放到内存中，当内存满了时再持久化到硬盘。当这个es集群关闭再重新启动时就会从gateway中读取索引数据。es支持多种类型的gateway，有本地文件系统（默认），分布式文件系统，Hadoop的HDFS和amazon的s3云存储服务。

## 3.7、discovery.zen

代表es的自动发现节点机制，es是一个基于p2p的系统，它先通过广播寻找存在的节点，再通过多播协议来进行节点之间的通信，同时也支持点对点的交互。

## 3.8、Transport

代表es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）。

# 4、Java API

## 4.1与集群交互

可以通过两种方式来连接到elasticsearch（简称es）集群，第一种是通过在你的程序中创建一个嵌入es节点（Node），使之成为es集群的一部分，然后通过这个节点来与es集群通信。第二种方式是用TransportClient这个接口和es集群通信。

两种方式的比较：

<https://www.elastic.co/guide/en/elasticsearch/guide/current/_transport_client_versus_node_client.html>

### 4.1.1、Node方式

创建嵌入节点的方式如下：

|  |
| --- |
| import static org.elasticsearch.node.NodeBuilder.\*;  //启动节点  Node node = nodeBuilder().node();  Client client = node.client();  //关闭节点  node.close(); |

当你启动一个节点，它会自动加入同网段的es集群，一个前提就是es的集群名（cluster.name）这个参数要设置一致。

默认的话启动一个节点，es集群会自动给它分配一些索引的分片，如果你想这个节点仅仅作为一个客户端而不去保存数据，你就可以设置把node.data设置成false或 node.client设置成true。下面是例子：

|  |
| --- |
| Node node = nodeBuilder().clusterName(clusterName).client(true).node(); |

还有一种情况是你并不想把节点加入集群，只想用它进行单元测试时，就要启动一个“本地”的es，这里“本地”指的是在jvm的级别下运行，即两个不同的es节点运行在同一个JVM中时会组成一个集群。它需要把节点的local参数设置成true，下面是例子：

|  |
| --- |
| Node node = nodeBuilder().local(true).node(); |

### 4.1.2、TransportClient方式

通过TransportClient这个接口，我们可以不启动节点就可以和es集群进行通信，它需要指定es集群中其中一台或多台机的ip地址和端口，例子如下：

|  |
| --- |
| Client client = new TransportClient()  .addTransportAddress(new InetSocketTransportAddress("host1", 9300))  .addTransportAddress(new InetSocketTransportAddress("host2", 9300));  client.close(); |

如果你需要更改集群名（默认是elasticsearch），需要如下设置：

|  |
| --- |
| Settings settings = ImmutableSettings.settingsBuilder()  .put("cluster.name", "myClusterName").build();  Client client = new TransportClient(settings); |

你可以设置client.transport.sniff为true来使客户端去嗅探整个集群的状态，把集群中其它机器的ip地址加到客户端中，这样做的好处是一般你不用手动设置集群里所有集群的ip到连接客户端，它会自动帮你添加，并且自动发现新加入集群的机器。代码实例如下：

|  |
| --- |
| Settings settings = ImmutableSettings.settingsBuilder()  .put("client.transport.sniff", true).build();  TransportClient client = new TransportClient(settings); |

### 4.1.3、put Mapping定义索引字段属性

Mapping,就是对索引库中索引的字段名及其数据类型进行定义，类似于关系数据库中表建立时要定义字段名及其数据类型那样，不过es的mapping比数据库灵活很多，它可以动态添加字段。一般不需要要指定mapping都可以，因为es会自动根据数据格式定义它的类型，如果你需要对某些字段添加特殊属性（如：定义使用其它分词器、是否分词、是否存储等），就必须手动添加mapping。有两种添加mapping的方法，一种是定义在配置文件中，一种是运行时手动提交mapping，两种选一种就行了。

先介绍在配置文件中定义mapping，你可以把[mapping名].json文件放到config/mappings/[索引名]目录下，这个目录要自己创建，一个mapping和一个索引对应，你也可以定义一个默认的mapping，把自己定义的default-mapping.json放到config目录下就行。json格式如下：

{

"mappings":{

"properties":{

"title":{

"type":"string",

"store":"yes"

},

"description":{

"type":"string",

"index":"not\_analyzed"

},

"price":{

"type":"double"

},

"onSale":{

"type":"boolean"

},

"type":{

"type":"integer"

},

"createDate":{

"type":"date"

}

}

}

}

接下来介绍通过请求添加mapping，下面为一个添加productIndex索引库的mapping的json格式请求。其中productIndex为索引类型，properties下面的为索引里面的字段，type为数据类型，store为是否存储，"index":"not\_analyzed"为不对该字段进行分词。

{

"productIndex":{

"properties":{

"title":{

"type":"string",

"store":"yes"

},

"description":{

"type":"string",

"index":"not\_analyzed"

},

"price":{

"type":"double"

},

"onSale":{

"type":"boolean"

},

"type":{

"type":"integer"

},

"createDate":{

"type":"date"

}

}

}

}

用java api调用的代码如下：

先创建空索引库

client.admin().indices().prepareCreate("productIndex").execute().actionGet();

put mapping：

|  |
| --- |
| //指定索引的类型映射  String type = "product";  XContentBuilder mapping = jsonBuilder()  .startObject()  .startObject(type)  .startObject("properties")  .startObject("title").field("type", "string").field("store", "yes").field("analyzer", "ik").field("search\_analyzer", "ik\_smart").endObject()  .startObject("description").field("type", "string").field("analyzer", "ik").field("search\_analyzer", "ik\_smart").endObject()  .startObject("price").field("type", "double").endObject()  .startObject("onSale").field("type", "boolean").endObject()  .startObject("type").field("type", "integer").endObject()  .startObject("updateDate").field("type", "date").field("format", "yyyy-MM-dd HH:mm:ss").endObject()  .endObject()  .endObject()  .endObject();  PutMappingRequest mappingRequest = Requests.putMappingRequest(INDEX)  .type(type).source(mapping);  client.admin().indices().putMapping(mappingRequest).actionGet(); |

### 4.1.4、索引数据

|  |
| --- |
| //es索引数据非常方便，只需构建个json格式的数据提交到es就行  XContentBuilder doc = jsonBuilder()  .startObject()  .field("title", "this is a title!")  .field("description", "descript what?")  .field("price", 100)  .field("onSale", true)  .field("type", 1)  .field("createDate", new Date())  .endObject();  IndexResponse response = client.prepareIndex(INDEX, "product")  .setId("1") //需要指定id，否则会生成一个随机序列号  .setSource(doc).execute().actionGet();  System.out.println(response.toString()); |

其中productIndex为索引库名，一个es集群中可以有多个索引库。productType为索引类型，是用来区分同索引库下不同类型的数据的，一个索引库下可以有多个索引类型。

### 4.1.5、删除索引数据

通过id删除：

|  |
| --- |
| String id = "1001";  DeleteResponse response = ESClient.getClient()  .prepareDelete(INDEX, Info.class.getSimpleName(), id).get(); |

### 4.1.6、搜索

elasticsearch的查询是通过执行json格式的查询条件，在java api中就是构造QueryBuilder对象，elasticsearch完全支持queryDSL风格的查询方式，QueryBuilder的构建类是QueryBuilders，filter的构建类是FilterBuilders。下面是构造QueryBuilder的例子：

|  |
| --- |
| import static org.elasticsearch.index.query.FilterBuilders.\*;  import static org.elasticsearch.index.query.QueryBuilders.\*;  QueryBuilder qb1 = termQuery("name", "kimchy");  QueryBuilder qb2 = boolQuery()  .must(termQuery("content", "test1"))  .must(termQuery("content", "test4"))  .mustNot(termQuery("content", "test2"))  .should(termQuery("content", "test3"));  QueryBuilder qb3 = filteredQuery(  termQuery("name.first", "shay"),  rangeFilter("age")  .from(23)  .to(54)  .includeLower(true)  .includeUpper(false)  ); |

其中qb1构造了一个TermQuery，对name这个字段进行项搜索，项是最小的索引片段，这个查询对应lucene本身的TermQuery。 qb2构造了一个组合查询（BoolQuery），其对应lucene本身的BooleanQuery，可以通过must、should、mustNot方法对QueryBuilder进行组合，形成多条件查询。 qb3构造了一个过滤查询，就是在TermQuery的基础上添加一个过滤条件RangeFilter，这个范围过滤器将限制查询age字段大于等于23，小于等于54的结果。除了这三个，elasticsearch还支持很多种类的查询方式，迟点写个介绍。

构造好了Query就要传到elasticsearch里面进行查询，下面是例子：

SearchResponse response = client.prepareSearch("test")

.setQuery(query)

.setFrom(0).setSize(60).setExplain(true)

.execute()

.actionGet();

这句的意思是，查询test索引，查询条件为query，从第0条记录开始，最多返回60条记录。返回结果为SearchResponse，下面解析SearchResponse：

SearchHits hits = searchResponse.hits();

for (int i = 0; i < 60; i++) {

System.out.println(hits.getAt(i).getSource().get("field"));

}

获得SearchResponse中的SearchHits，然后hits.getAt(i).getSource().get("field")获得field字段的值。

### 4.1.7、批量添加索引

elasticsearch支持批量添加或删除索引文档，java api里面就是通过构造BulkRequestBuilder，然后把批量的index/delete请求添加到BulkRequestBuilder里面，执行BulkRequestBuilder。下面是个例子：

|  |
| --- |
| import static org.elasticsearch.common.xcontent.XContentFactory.\*;    BulkRequestBuilder bulkRequest = client.prepareBulk();    bulkRequest.add(client.prepareIndex("twitter", "tweet", "1")  .setSource(jsonBuilder()  .startObject()  .field("user", "kimchy")  .field("postDate", new Date())  .field("message", "trying out Elastic Search")  .endObject()  )  );    bulkRequest.add(client.prepareIndex("twitter", "tweet", "2")  .setSource(jsonBuilder()  .startObject()  .field("user", "kimchy")  .field("postDate", new Date())  .field("message", "another post")  .endObject()  )  );    BulkResponse bulkResponse = bulkRequest.execute().actionGet();  if (bulkResponse.hasFailures()) {  //处理错误  } |

### 4.1.8、更新数据

|  |
| --- |
| UpdateRequest updateRequest = new UpdateRequest();  updateRequest.index(INDEX);  updateRequest.type(Info.class.getSimpleName());  updateRequest.id("1002");  updateRequest.doc(jsonBuilder()  .startObject()  .field("name3", "测试2")  .endObject());  ESClient.getClient().update(updateRequest).get(); |

注：id不存在是将抛出异常，不会自动新建该条数据，域(field)不存在时会新增一个新域

# 高级配置

## 分片分布规则设置

分片分布是把索引分片分布到节点的过程。这个操作会在初次启动集群，副本分配，负载均衡，或增加删除节点时进行。

下面是一些与分片分布相关的设置：

cluster.routing.allocation.allow\_rebalance

设置根据集群中机器的状态来重新分配分片，可以设置为always, indices\_primaries\_active和indices\_all\_active，默认是设置成indices\_all\_active来减少集群初始启动时机器之间的交互。

cluster.routing.allocation.cluster\_concurrent\_rebalance

设置在集群中最大允许同时进行分片分布的个数，默认为2，也就是说整个集群最多有两个分片在

进行重新分布。

cluster.routing.allocation.node\_initial\_primaries\_recoveries

设置指定初始每个节点。由于多数情况下是使用local的gateway，这应该会更快，

cluster.routing.allocation.node\_concurrent\_recoveries

设置在节点中最大允许同时进行分片分布的个数，默认为2

cluster.routing.allocation.disable\_allocation

使主要分片或副本的分布失效。要知道，如果主分片不存在（那个节点挂了）那么其副本仍然会被提升为主分片，这个设置只有在动态地使用集群更新设置api调用时才生效。

cluster.routing.allocation.disable\_replica\_allocation

使副本分布失效。和上一个设置一样，只有动态地使用集群更新设置api调用时才生效。

indices.recovery.concurrent\_streams

当从一个点（peer）恢复分片时当前节点最多允许的文件读取流的个数，默认为5

自定义分片分布规则

可以通过设置分片的分布规则来人为地影响分片的分布，下面是个例子：

假设我们有几个机架。当我们启动一个节点，我们可以设置一个叫rack\_id（其它名字也可以）的属性，例如下面设置：

node.rack\_id: rack\_one

上面这个例子设置了一个属性叫rack\_id，它的值为rack\_one。现在，我们要设置rack\_id作为分片分布规则的一个属性（在所有节点都要设置）。

cluster.routing.allocation.awareness.attributes: rack\_id

上面设置意味着rack\_id会用来作为分片分布的依据。例如：我们启动两个node.rack\_id设置rack\_one的节点，然后建立一个5个分片，一个副本的索引。这个索引就会完全分布在这两个节点上。如果再启动另外两个节点，node.rack\_id设置成rack\_two，分片会重新分布，但是一个分片和它的副本不会分配到同样rack\_id值的节点上。可以为分片分布规则设置多个属性，例如：

cluster.routing.allocation.awareness.attributes: rack\_id,zone

注意：当设置了分片分布属性时，如果集群中的节点没有设置其中任何一个属性，那么分片就不会分布到这个节点中。

强制分布规则

更多的时候，我们不想更多的副本被分布到相同分布规则属性值的一群节点上，那么，我们可以强制分片规则为一个指定的值。

例如，我们有一个分片规则属性叫zone，并且我们知道有两个zone，zone1和zone2.下面是设置：

cluster.routing.allocation.awareness.force.zone.values: zone1,zone2

cluster.routing.allocation.awareness.attributes: zone

现在我们启动两个node.zone设置成zone1的节点，然后创建一个5个分片，一个副本的索引。索引建立完成后只有5个分片（没有副本），只有当我们启动node.zone设置成zone2的节点时，副本才会分配到那节点上。

分片分布过滤

允许通过include/exclude过滤器来控制分片的分布。这些过滤器可以设置在索引级别上或集群级别上。下面是个索引级别上的例子:

假如我们有四个节点，每个节点都有一个叫tag（可以是任何名字）的属性。每个节点都指定一个tag的值。如：节点一设置成node.tag: value1，节点二设置成node.tag: value2，如此类推。我们可以创建一个索引然后只把它分布到tag值为value1和value2的节点中，可以通过设置

index.routing.allocation.include.tag 为value1,value2达到这样的效果，如：

curl -XPUT localhost:9200/test/\_settings -d '{

"index.routing.allocation.include.tag" : "value1,value2"

}'

与此相反，通过设置index.routing.allocation.exclude.tag为value3，我们也可以创建一个索引让其分布在除了tag设置为value3的所有节点中，如：

curl -XPUT localhost:9200/test/\_settings -d '{

"index.routing.allocation.exclude.tag" : "value3"

}'

include或exclude过滤器的值都会使用通配符来匹配，如value\*。一个特别的属性名是\_ip，它可以用来匹配节点的ip地址。

显然，一个节点可能拥有多个属性值，所有属性的名字和值都在配置文件中配置。如，下面是多个节点的配置：

node.group1: group1\_value1

node.group2: group2\_value4

同样的方法，include和exclude也可以设置多个值，如：

curl -XPUT localhost:9200/test/\_settings -d '{

"index.routing.allocation.include.group1" : "xxx"

"index.routing.allocation.include.group2" : "yyy",

"index.routing.allocation.exclude.group3" : "zzz",

}'

上面的设置可以通过索引更新的api实时更新到索引上，允许实时移动索引分片。

集群范围的过滤器也可以定义，可以通过集群更新api实时更新到集群上。这些设置可以用来做让一些节点退出集群的操作。下面是通过ip地址去掉一个节点的操作：

curl -XPUT localhost:9200/\_cluster/settings -d '{

"transient" : {

"cluster.routing.allocation.exclude.\_ip" : "10.0.0.1"

}

}'

## 线程池设置

一个Elasticsearch节点会有多个线程池，但重要的是下面四个：

索引（index）：主要是索引数据和删除数据操作（默认是cached类型）

搜索（search）：主要是获取，统计和搜索操作（默认是cached类型）

批量操作（bulk）：主要是对索引的批量操作（默认是cached类型）

更新（refresh）：主要是更新操作（默认是cached类型）

可以通过给设置一个参数来改变线程池的类型（type），例如，把索引的线程池改成blocking类型：

min: 1

size: 30

wait\_time: 30s

下面是三种可以设置的线程池的类型

**cache**

cache线程池是一个无限大小的线程池，如果有很请求的话都会创建很多线程，下面是个例子：

threadpool:

index:

type: cached

**fixed**

fixed线程池保持固定个数的线程来处理请求队列。

size参数设置线程的个数，默认设置是cpu核心数的5倍

queue\_size可以控制待处理请求队列的大小。默认是设置为-1，意味着无限制。当一个请求到来但队列满了的时候，reject\_policy参数可以控制它的行为。默认是abort，会使那个请求失败。设置成caller会使该请求在io线程中执行。

threadpool:

index:

type: fixed

size: 30

queue: 1000

reject\_policy: caller

**blocking**

blocking线程池允许设置一个最小值（min，默认为1）和线程池大小（size，默认为cpu核心数的5倍）。它也有一个等待队列，队列的大小（queue\_size ）默认是1000，当这队列满了的时候。它会根据定好的等待时间（wait\_time，默认是60秒）来调用io线程，如果没有执行就会报错。

threadpool:

index:

type: blocking

min: 1

size: 30

wait\_time: 30s

## 虚拟机配置

引言：

今天，事情终于发生了。Java6（Mustang），是2006年早些时候出来的，至今仍然应用在众多生产环境中，现在终于走到了尽头。已经没有什么理由阻止迁移到Java7(Dolphin)上了。

这也促使我想写一篇关于在ElasticSearch上配置Java6和7的细微差异的博文。

Elasticsearch对Java虚拟机进行了预先的配置。通常情况下，因为这些配置的选择还是很谨慎的，所以你不需要太关心，并且你能立刻使用ElasticSearch。

但是，当你监视ElasticSearch节点内存时，你可能尝试修改一些配置。这些修改是否会改善你的处境？

这篇博文尝试揭开Elasticsearch配置的神秘面纱，并且讨论最常见的调整。最终，会给出一些推荐的配置调整。

Elasticsearch JVM 配置概览：

这些是Elasticsearch 0.19.11版本的默认配置。

|  |  |  |
| --- | --- | --- |
| JVM参数 | Elasticsearch默认值 | Environment变量 |
| -Xms | 256m | ES\_MIN\_MEM |
| -Xmx | 1g | ES\_MAX\_MEM |
| -Xms and -Xmx |  | ES\_HEAP\_SIZE |
| -Xmn |  | ES\_HEAP\_NEWSIZE |
| -XX:MaxDirectMemorySize |  | ES\_DIRECT\_SIZE |
| -Xss | 256k |  |
| -XX:UseParNewGC | + |  |
| -XX:UseConcMarkSweepGC | + |  |
| -XX:CMSInitiatingOccupancyFraction | 75 |  |
| -XX:UseCMSInitiatingOccupancyOnly | + |  |
| -XX:UseCondCardMark | (commented out) |  |

首先你注意到的是，Elasticsearch预留了256M到1GB的堆内存。

这个设置适用于开发和演示环境。开发人员只需要简单的解压发行包，再执行./bin/elasticsearch -f就完成了Elasticsearch的安装。当然这点对于开发来说非常棒，并且在很多场景下都能工作，但是当你需要更多内存来降低 Elasticsearch负载的时候就不行了，你需要比2GB RAM更多的可用内存。

ES\_MIN\_MEM/ES\_MAX\_MEM是控制堆大小的配置。新的ES\_HEAP\_SIZE变量是一个更为便利的选择，因为将堆的初始大小和最大值设为相同。也推荐在分配堆内存时尽可能不要用内存的碎片。内存碎片对于性能优化来说非常不利。

ES\_HEAP\_NEWSIZE是可选参数，它控制堆的子集大小，也就是新生代的大小。

ES\_DIRECT\_SIZE控制本机直接内存大小，即JVM管理NIO框架中使用的数据区域大小。本机直接内存可以被映射到虚拟地址空间上，这样在64位的机器上更高效，因为可以规避文件系统缓冲。Elasticsearch对本机直接内存没有限制(可能导致OOM)。

由于历史原因Java虚拟机有多个垃圾收集器。可以通过以下的JVM参数组合启用：

|  |  |
| --- | --- |
| JVM parameter | Garbage collector |
| -XX:+UseSerialGC | serial collector |
| -XX:+UseParallelGC | parallel collector |
| -XX:+UseParallelOldGC | Parallel compacting collector |
| -XX:+UseConcMarkSweepGC | Concurrent-Mark-Sweep (CMS) collector |
| -XX:+UseG1GC | Garbage-First collector (G1) |

UseParNewGC和UseConcMarkSweepGC组合启用垃圾收集器的并发多线程模式。UseConcMarkSweepGC自动选择UseParNewGC模式并禁用串行收集器（Serial collector）。在Java6中这是默认行为。

CMSInitiatingOccupancyFraction提炼了一种CMS（Concurrent-Mark-Sweep）垃圾收集设置；它将旧生代触发垃圾收集的阀值设为75.旧生代的大小是堆大小减去新生代大小。这告诉JVM当堆内容达到75%时启用垃圾收集。这是个估计的值，因为越小的堆可能需要越早启动GC。

UseCondCardMark将在垃圾收集器的card table使用时，在marking之前进行额外的判断，避免冗余的store操作。UseCondCardMark不影响Garbage-First收 集器。强烈推荐在高并发场景下配置这个参数（规避card table marking技术在高并发场景下的降低吞吐量的负面作用）。在ElasticSearch中，这个参数是被注释掉的。

有些配置可以参考诸如Apache Cassandra项目，他们在JVM上有类似的需求。

总而言之，ElastciSearch配置上推荐：

1. 不采用自动的堆内存配置，将堆大小默认最大值设为1GB

2.调整触发垃圾收集的阀值，比如将gc设为75%堆大小的时候触发，这样不会影响性能。

3.禁用Java7默认的G1收集器，前提是你的ElasticSearch跑在Java7u4以上的版本上。

JVM进程的内存结果

JVM内存由几部分组成：

Java代码本身：包括内部代码、数据、接口，调试和监控代理或者字节码指令

非堆内存：用于加载类

栈内存：用于为每个线程存储本地变量和操作数

堆内存：用于存放对象引用和对象本身

直接缓冲区：用于缓冲I/O数据

堆内存的大小设置非常重要，因为Java的运行依赖于合理的堆大小，并且JVM需要从操作系统那获取有限的堆内存，用于支撑整个JVM生命周期。

如果堆太小，垃圾回收就会频繁发生，发生OOM的几率会很大。

如果堆太大，垃圾回收会延迟，但是一旦回收，就需要处理大量的存活堆数据。并且，操作系统的压力也会变大，因为JVM进程需要更大的堆，产生换页的可能性就会提高。

注意，使用CMS垃圾收集器，Java不会把内存还给操作系统，因此配置合理的堆初始值和最大值就非常重要。

非堆内存由Java应用自动分配。没有什么参数控制这里的大小，这是由Java应用程序代码自己决定的。

栈内存在每个线程中分配，在Elasticsearch中，每个线程大小必须由128K增加到256K，因为Java7比Java6需要更大的栈内存 ，这是由于Java7支持新的编程语言特征来利用栈空间。比如，引入了continuations模型，编程语言的一个著名概念。Continuations模型对于

协同程序、绿色线程（green thread）、纤程（fiber）非常有用 。当实现非阻塞I/O时，一个大的优势是，代码可以根据线程实际使用情况编写，但是运行时仍然在后台采用非 阻塞I/O。Elasticsearch使用了多个线程池，因为Netty I/O框架和Guava是Elasticsearch的基础组件，因此在用Java7时，可以考虑进一步挖掘优化线程的特性。

发挥增加栈空间大小的优势还是有挑战的，因为不同的操作系统、不同的CPU架构，甚至在不同的JVM版本之间，栈空间的消耗不是容易比较的。取决于CPU 架构和操作系统，JVM的栈空间大小是内建的。他们是否在所有场景下都适合？例如Sloaris Sparc 64位的JVM Xss默认为512K，因为有更大地址指针，Sloaris X86为320K。Linux降为256K。Windows 32位Java6默认320K，Windows 64位则为1024K。

大堆的挑战

今天，几GB的内存是很常见的。但是在不久以前，系统管理员还在为多几G的内存需求泪流满面。

Java垃圾收集器是随着2006年的Java6的出现而显著改进的。从那以后，可以并发执行多任务，并且减少了GC停顿几率： stop - the - world阶段。CMS算法是革命性的，多任务，并发， 不需要移动的GC。但是不幸的是，对于堆的存活数据量来说，它是不可扩展的。Prateek Khanna 和 Aaron Morton给出了CMS垃圾收集器能够处理的堆规模的数字。

避免Stop-the-world阶段

我们已经学习了Elasticsearch如何配置CMS垃圾收集器。但这并不能组织长时间的GC停顿，它只是降低了发生的几率。CMS是一个低停顿几率 的收集器，但是仍然有一些边界情况。当堆上有MB级别的大数组，或者其他一些特殊的场景，CMS可能比预期要花费更多的时间。

MB级别数组的创建在Lucene segment-based索引合并时是很常见的。如果你希望降低CMS的额外负载，就需要调整Lucene合并阶段的段数量，使用参数index.merge.policy.segments\_per\_tier

减少换页

大堆的风险在于内存压力上。注意，如果Java JVM在处理大堆时，这部分内存对于系统其它部分来说是不可用的。如果内存吃紧，操作系统会进行换页，并且，在紧急情况下，当所有其他方式回收内存都失败 时，会强制杀掉进程。如果换页发生，整个系统的性能会下降，自然GC的性能也跟着下降。所以，不要给堆分配太多的内存。

垃圾收集器的选择

从Java JDK 7u4开始，Garbage-First（G1）收集器是Java7默认的垃圾收集器。它适用于多核的机器以及大内存。它一方面降低了停顿时间，另一方面 增加了停顿的次数。整个堆的操作，例如全局标记，是在应用线程中并发执行的。这会防止随着堆或存活数据大小的变化，中断时间也成比例的变化。

G1收集器目标是获取更高的吞吐量，而不是速度。在以下情况下，它能运行的很好：

1. 存活数据占用了超过50%的Java堆

2. 对象分配比例或者promotion会有明显的变化

3. 不希望gc或者compaction停顿时间长（超过0.5至1s）

注意，如果使用G1垃圾收集器，堆不再使用的内存可能会被归还给操作系统

G1垃圾收集器的不足是CPU使用率越高，应用性能越差。因此，如果在内存足够和CPU能力一般的情况下，CMS可能更胜一筹。

对于Elasticsearch来说，G1意味着没有长时间的stop-the-world阶段，以及更灵活的内存管理，因为buffer memory和系统I/O缓存能更充分的利用机器内存资源。代价就是小成本的最大化性能，因为G1利用了更多CPU资源。

性能调优策略

你读这篇博文因为你希望在性能调优上得到一些启示：

1. 清楚了解你的性能目标。你希望最大化速度，还是最大化吞吐量？

2. 记录任何事情（log everything），收集统计数据，阅读日志、分析事件来诊断配置

3. 选择你调整的目标（最大化性能还是最大化吞吐量）

4. 计划你的调整

5. 应用你的新配置

6. 监控新配置后的系统

7. 如果新配置没有改善你的处境，重复上面的一系列动作，反复尝试

Elasticsearch垃圾收集日志格式

Elasticsearch长时间GC下warns级别的日志如下所示：

[2012-11-26 18:13:53,166][WARN ][monitor.jvm              ] [Ectokid] [gc][ParNew][1135087][11248] duration [2.6m], collections [1]/[2.7m], total [2.6m]/[6.8m], memory [2.4gb]->[2.3gb]/[3.8gb], all\_pools {[Code Cache] [13.7mb]->[13.7mb]/[48mb]}{[Par Eden Space] [109.6mb]->[15.4mb]/[1gb]}{[Par Survivor Space] [136.5mb]->[0b]/[136.5mb]}{[CMS Old Gen] [2.1gb]->[2.3gb]/[2.6gb]}{[CMS Perm Gen] [35.1mb]->[34.9mb]/[82mb]}

JvmMonitorService类中有相关的使用方式：

|  |  |
| --- | --- |
| Logfile | Explanation |
| gc | 运行中的gc |
| ParNew | new parallel garbage collector |
| duration 2.6m | gc时间为2.6分钟 |
| collections [1]/[2.7m] | 在跑一个收集，共花2.7分钟 |
| memory [2.4gb]->[2.3gb]/[3.8gb] | 内存消耗, 开始是2.4gb, 现在是2.3gb, 共有3.8gb内存 |
| Code Cache [13.7mb]->[13.7mb]/[48mb] | code cache占用内存 |
| Par Eden Space [109.6mb]->[15.4mb]/[1gb] | Par Eden Space占用内存 |
| Par Survivor Space [136.5mb]->[0b]/[136.5mb] | Par Survivor Space占用内存 |
| CMS Old Gen [2.1gb]->[2.3gb]/[2.6gb] | CMS Old Gen占用内存 |
| CMS Perm Gen [35.1mb]->[34.9mb]/[82mb] | CMS Perm Gen占用内存 |

JvmMonitorSer

一些建议

1. 不要在Java 6u22之前的发布版本中跑Elasticsearch。有内存方面的bug。那些超过两三年的bug和缺陷会妨碍Elasticsearch的正常运行。与旧的OpenJDK 6相比，更推荐Sun/Oracle的版本，因为后者修复了很多bug。

2. 放弃Java6，转到Java7。Oracle宣称Java6更新到2013年2月结束。考虑到Elasticsearch还是一个相对新的软件，应该使用更新的技术来提升性能。尽量从JVM中挤压性能。检查操作系统的版本。在最新版本的操作系统中运行，有助于你的Java运行环境达到最佳性能。

3. 定期更新Java运行环境。平均一个季度一次。告诉sa你需要及时更新Java版本，以获取Java性能的提升。

4. 从小到大。先在Elasticsearch单节点上进行开发。但是不要忘了Elasticsearch分布式的强大功能。单节点不能模拟生产环境的特征，至少需要3个节点进行开发测试。

5. 在调整JVM之前先做一下性能测试。对你的系统建立性能基线。调整测试时候的节点数量。如果索引时候负载很高，你可能需要降低Elasticsearch索引时候占用的堆大小，通过index.merge.policy.segments\_per\_tierparameter参数调整段的合并。

6. 调整前清楚你的性能目标，然后决定是调整速度还是吞吐量。

7. 启用日志以便更好的进行诊断。在优化系统前进行小心的评估。

8. 如果使用CMS垃圾收集器，你可能需要加上合理的 -XX:CMSWaitDuration 参数。

9. 如果你的堆超过6-8GB，超过了CMS垃圾收集器设计容量，你会遇到长时间的stop-the-world阶段，你有几个方案：调整CMSInitiatingOccupancyFraction参数降低长时间GC的几率减少最大堆的大小；启用G1垃圾收集器。

10. 学习垃圾收集调优艺术。如果你想精通的话，列出可用的JVM选项，在java命令中加入java -XX:+UnlockDiagnosticVMOptions -XX:+PrintFlagsFinal -version，然后调优。

参考资料

* 1. Elasticsearch官网：<http://www.elasticsearch.cn/guide/>
  2. <http://blog.csdn.net/u010994304/article/details/50423883>
  3. <http://blog.csdn.net/geloin/article/details/8444658>