第 6 章 Chapter 6

故障处理

上一章中，我们对 ElasticSearch 的各种管理属性进行了深入探讨。首先了解了如何选择合适的 Lucene 目录实现，以及哪种实现最适合当前的情况。然后讨论了 ElasticSearch 发现模块的工作原理，了解了多播发现和单播发现，以及如何使用 Amazon EC2 的发现模块。此外，我们了解了网关模块并学习如何配置它的恢复行为。我们使用了一些额外的 ElasticSearch API，并学习如何查找当前索引对应的索引段，以及如何将这些索引信息进行可视化展示。最后，了解了 ElasticSearch 缓存的种类、用途和配置，以及如何通过 ElasticSearch API 来清除缓存。读完这一章，你将掌握以下内容：

- 垃圾回收器是什么，它如何工作，如何定位垃圾回收器产生的问题。
- 如何控制 ElasticSearch 的 I/O 操作数量。
- 预热器加快搜索速度的原理及其示例。
- 什么是热点线程以及如何获取热点线程的列表。
- 在诊断集群和节点故障时应使用哪个 ElasticSearch API。

6.1 了解垃圾回收器

ElasticSearch 是一个 Java 应用程序，因此它需要在 Java 虚拟机（JVM）中运行。每个 Java 应用程序都会编译成可被 JVM 执行的字节码。一般来说，你可以把 JVM 的用途理解成用来执行其他程序并控制其行为。然而，在这里我们不关心这些，除非需要给 ElasticSearch 开发插件，插件开发的相关技术我们将在第 9 章中讨论。这里我们需要关注的是垃圾回收器，即 JVM 中负责内存管理的部分。当取消引用对象时，由垃圾回收器负责从内存中清除它；当内存资源紧张时，垃圾回收器开始工作。在本节中，我们将学习如何

配置垃圾回收器，如何避免内存交换，如何对垃圾回收行为记录日志、诊断故障，并使用一些 Java 工具，而这些工具能向我们展示垃圾回收相关的工作过程。

你可以从互联网上的许多地方了解更多关于 JVM 架构的知识，例如，你可以访问维基百科：http://en.wikipedia.org/wiki/Java_virtual_machine。

6.1.1 Java 内存

我们用 Xms 和 Xmx 参数（或者使用 ElasticSearch 的 ES_MIN_MEM 和 ES_MAX_MEM 属性）指定内存大小，其实质是指定了 JVM 的堆空间大小。堆空间本质上是内存中划拨给 JVM 的一块保留区域，该区域可以被 Java 程序使用。在这里，Java 程序就是指 ElasticSearch。Java 进程不会占用太多堆空间，最多只会使用 Xmx 参数（或 ES_MAX_MEM 属性）所指定的那么多堆内存。在 Java 程序中，当一个新对象在创建时，它就会被放入堆中。而当不再使用它时，垃圾回收器会尝试把它从堆中去除，释放它占用的空间，便于以后 JVM 重用。你可以想象，如果我们的应用程序没有足够的堆空间来创建、存储新对象，噩梦就发生了：JVM 会抛出 OutOfMemory 异常，这通常意味着内存出了状况，要么没有足够的内存，要么存在内存泄漏，或没有释放掉不再使用的对象。

JVM 的内存空间分为以下区域：

- Eden 区（Eden space）：JVM 初次分配的大部分对象都在该区域内。
- Survivor 区（Survivor space）：这块区域存储的对象是对 Eden 区进行垃圾回收后仍然存活的对象。Survivor 区分为两部分：Survivor 0 区和 Survivor 1 区。
- 年老代（Tenured generation）：这块区域存储的是那些在 Surviror 区存活较长时间的对象。
- 持久代（Permanent generation）：这是一块非堆空间，用来存储所有 JVM 自身的数据，如 Java 类、对象方法等。
- 代码缓存区（Code cache）：这是 HotSpot JVM 中存在的一块非堆空间，用来编译、存放本地原生代码。

上述分类方法可以进一步简化：Eden 区和 Survivor 区可以合称为年轻代（Young generation）。

Java 对象的生命周期和垃圾回收

为了考察垃圾回收器的工作过程，我们来看一个简单 Java 对象的生命周期。

Java 程序将新创建的对象放置在年轻代的 Eden 区。如果年轻代执行下一次垃圾回收时，这个对象仍然存活（一般来说，它不是一次性对象，Java 程序仍然需要用到它），那么它会被挪到 Survivor 区（先进入 Survivor 0 区，如果经过年轻代的下一轮垃圾回收仍然存活，则它会被挪到 Survivor 1 区）。

当对象在 Survivor 1 区存活一段时间后，会被挪到年老代，成为年老代对象的一员，

且从此以后，年轻代的垃圾回收不再对它起作用，因而该对象会一直在年老代中生存下去直到 Java 程序不再使用它。在这种情况下，如果它不再使用，那么在下次做全局垃圾回收时，会把它从堆空间中移除，并回收空间给新对象使用。

基于上面的介绍，我们可以断定（实质上也是如此）：截至目前 Java 还在使用分代的垃圾回收机制；对象经历垃圾回收次数越多，越容易往年老代迁移。因此我们可以说，存在两种垃圾回收器在并行运行：年轻代垃圾回收器（也称为次垃圾回收器）和年老代垃圾回收器（也称为主垃圾回收器）。

6.1.2 处理垃圾回收问题

处理垃圾回收问题时，首先要确定问题的源头。这不是简单直接的工作，通常需要取得系统管理员或集群负责人的帮助。在本节中，我们会提供两种检查和标识垃圾回收器问题的方法：第一种是开启 ElasticSearch 中的垃圾回收器日志，第二种是使用在大部分 Java 发行版中都携有的 jstat 命令。

打开垃圾回收日志

ElasticSearch 允许我们观察那些执行周期较长的垃圾回收器。在默认的 elasticsearch.yml 配置文件里，你会发现如下设置，它们默认被注释掉了：

```
monitor.jvm.gc.ParNew.warn: 1000ms
monitor.jvm.gc.ParNew.info: 700ms
monitor.jvm.gc.ParNew.debug: 400ms
monitor.jvm.gc.ConcurrentMarkSweep.warn: 10s
monitor.jvm.gc.ConcurrentMarkSweep.info: 5s
monitor.jvm.gc.ConcurrentMarkSweep.debug: 2s
```

如你所见，配置文件中指定了 3 种日志级别，并分别指定了阈值。以 info 级别的日志配置为例：如果年轻代垃圾回收过程耗时达到或超过 700 毫秒，ElasticSearch 就会往日志文件里写入日志；同样，如果年老代垃圾回收过程耗时达到或超过 5 秒，ElasticSearch 也会记录日志。

你可以在日志文件中看到类似下面的信息：

```
[EsTestNode] [gc][ConcurrentMarkSweep][964][1] duration [14.8s],
  collections [1]/[15.8s], total [14.8s]/[14.8s], memory [8.6gb]-
  >[3.4gb]/[11.9gb], all_pools {[Code Cache] [8.3mb]-
  >[8.3mb]/[48mb]}{[Par Eden Space] [13.3mb]
  >[3.2mb]/[266.2mb]}{[Par Survivor Space] [29.5mb]
  >[0b]/[33.2mb]}{[CMS Old Gen] [8.5gb]->[3.4gb]/[11.6gb]}{[CMS
  Perm Gen] [44.3mb]->[44.2mb]/[82mb]}
```

正如你所看到的那样，上面这行日志是关于 ConcurrentMarkSweep 垃圾回收器的，也就是说，它是关于年老代垃圾回收的。从中可以看出：总的回收时间是 14.8 秒；在回收前，总共 11.9GB 空间中有 8.6GB 被占用，而在回收后，被占用空间减少到 4.3GB；接下来是更

详细的统计信息，用来确定堆内存中的各个部分如代码缓存区、Eden 区、Survivor 区、年老代、持久代等的回收处理情况。

开启垃圾回收器日志并设置好特定阈值后，如果系统没有按计划工作，我们就可以通过日志文件及时发现。而如果你需要了解更多出错信息，可以用 Java 提供的工具：jstat 命令。

使用 jstat

使用 jstat 检查垃圾回收器的工作状况，只需输入如下简单命令：

```
jstat -gcutil 123456 2000 1000
```

其中，-gcutil 开关用来告知 jstat 监控垃圾回收器的工作状态；123456 用于标识 ElasticSearch 所在的 JVM；2000 表示抽样间隔，单位是毫秒；1000 表示的抽样数。因此在本例中，执行这个命令至少会耗时 33 分钟（2000 * 1000 / 1000 / 60）。

大多数情况下，JVM 的标识符与进程号相似，甚至一样，但是也有例外。为了确定正在运行的 Java 进程及其对应虚拟机的标识符，我们只需要执行 jps 命令，大部分 JDK 发行版中都包含这个命令。一个简单的 jps 命令如下所示：

```
jps
```

它的执行结果如下：

```
16232 Jps
11684 ElasticSearch
```

从结果中可以看到，每一行代表一个 JVM 标识符和 Java 进程名称。如果你想要了解更多 jps 命令的信息，请访问 Java 文档：http://docs.oracle.com/javase/7/docs/ technotes/tools/share/jps.html。

请确保执行 jstat 命令的账号与 ElasticSearch 进程所使用的账号相同，如果做不到这一点，请在执行时加上管理员权限（如在 Linux 系统中使用 sudo 命令）。必须确保 jstat 命令拥有访问 ElasticSearch 进程的权限，否则它无法连接到 ElasticSearch 进程以获取信息。

接下来，我们看一个关于 jstat 命令输出的示例：

```
 S0     S1     E      O      P     YGC     YGCT    FGC    FGCT     GCT
12.44   0.00  27.20   9.49  96.70    78    0.176     5    0.495    0.672
12.44   0.00  62.16   9.49  96.70    78    0.176     5    0.495    0.672
12.44   0.00  83.97   9.49  96.70    78    0.176     5    0.495    0.672
 0.00   7.74   0.00   9.51  96.70    79    0.177     5    0.495    0.673
 0.00   7.74  23.37   9.51  96.70    79    0.177     5    0.495    0.673
 0.00   7.74  43.82   9.51  96.70    79    0.177     5    0.495    0.673
 0.00   7.74  58.11   9.51  96.71    79    0.177     5    0.495    0.673
```

上面这条输出来自 Java 文档，它完美地展示了 jstat 命令的方方面面，我们以它为例来

详细讨论。首先阐述一下每一列的含义：

- S0：表示 survivor 0 区的使用率，用百分比表示。
- S1：表示 survivor 1 区的使用率，用百分比表示。
- E：表示 Eden 区的使用率，用百分比表示。
- O：表示年老代的使用率，用百分比表示。
- YGC：表示年轻代垃圾回收事件的次数。
- YGCT：表示年轻代垃圾回收耗时，单位为秒。
- FGC：表示年老代垃圾回收事件的次数。
- FGCT：表示年老代垃圾回收耗时，单位为秒。
- GCT：表示垃圾回收总耗时。

现在回到刚才的示例。如你所见，在第三和第四次抽样之间 JVM 对年轻代执行了一次垃圾回收，此次操作耗时 0.001 秒（第四次抽样的 YGCT 是 0.177，第三次抽样的 YGCT 是 0.176，两次 YGCT 之差为 0.001）。同时我们可以看到，Eden 区使用率从 83.7% 变为 0%，而年老代使用率从 9.49% 上升为 9.51%，这是因为此次操作中一些对象从 Eden 区迁移到了年老代。这个示例同时也展示了如何对 jstat 的输出进行分析。当然，分析工作比较费时，也需要你了解垃圾回收器是如何工作以及明白堆中都存放了什么。尽管如此，这可能是 ElasticSearch 故障时你能用的唯一方法。如果你遇到以下情况，ElasticSearch 运行不正常，或者 S0、S1、E 列的值达到 100%，并且垃圾回收工作对这些堆空间不起作用，那么原因可能是：年轻代太小了，你需要把它调大一些（当然，前提是拥有足够的物理内存）；内存出问题了，比如说因为一些资源没有释放占用的内存而导致内存泄露。还有一种情况，如果年老代使用率达到 100% 且垃圾回收器多次尝试仍无法释放它的空间，这大概意味着没有足够的堆空间让 ElasticSearch 节点正常运作了。此时，如果你不想改变索引结构，就只能通过增大运行 ElasticSearch 的 JVM 的堆空间来解决了（更多 JVM 参数信息请参考：http://www.oracle.com/technetwork/java/javase/ tech/vmoptions-jsp-140102.html）。

生成 Memory Dump

JVM 还拥有把堆空间转储到文件的能力。Java 允许我们获取特定时间点的一个内存快照，并通过分析快照的存储内容，从而发现问题。你可以使用 jmap 命令（http:// docs.oracle.com/javase/7/docs/technotes/tools/share/jmap.html）去转储 Java 进程的内存。Jmap 命令示例如下：

```
jmap -dump:file=heap.dump 123456
```

123456 表示需要转储的 Java 进程号。-dump:file=heap.dump 指定转储的目标文件为 heap.dump。我们可以通过一些特殊的软件进一步分析转储文件，如 jhat（http://docs.oracle.com/javase/7/docs/ technotes/tools/share/jhat.html）。关于这些软件的使用超出了本书的讨论范围，本书不做具体讲解。

垃圾回收器的更多工作信息

对垃圾回收进行调优并不容易。ElasticSearch 发布版提供的默认选项足够应付绝大部分情况。你只需要做一件事：调整节点内存大小。垃圾回收调优的相关话题不在本书中讨论。它涉及的话题非常广泛，甚至被某些开发者称为黑魔法。然而，如果你想了解更多关于垃圾回收器的信息，如它有哪些选项以及这些选项如何作用到你的程序，建议阅读这篇伟大的文章：http://www.oracle.com/ technetwork/java/javase/gc-tuning-6-140523.html。尽管该文章是关于 Java 6 的，但其中提及的绝大部分选项对 Java 7 应用同样适用。

谨记一件事：尽可能进行多次轻量级垃圾回收操作，而不是一次性、耗时很长的垃圾回收操作。因为保持 ElasticSearch 程序性能的稳定性是第一位的，对 ElasticSearch 来说，垃圾回收应该是透明的。一次重大的垃圾回收操作会停止全局的垃圾回收事件，在短时间内冻结 ElasticSearch 的工作，从而使查询变慢且短期无法执行索引操作。

调整 ElasticSearch 中垃圾回收器的工作方式

我们已经知道了垃圾回收器如何工作，也明白了如何诊断垃圾回收过程中的故障，接下来可以探讨一下如何通过修改 ElasticSearch 启动参数来调整垃圾回收器的工作方式。参数的修改方式视 ElasticSearch 的运行方式而定，目前常见的有两种：一种是使用 ElasticSearch 发行包自带的标准启动脚本，另一种是使用服务包装器（service wrapper）。

使用标准启动脚本

为了添加额外的 JVM 参数，我们需要把它们加入 JAVA_OPTS 环境变量中。例如，要给 Linux 环境下的 ElasticSearch 添加 -XX:+UseParNewGC -XX:+UseConcMarkSweepGC 启动参数，需要执行如下命令：

```
export JAVA_OPTS="-XX:+UseParNewGC -XX:+UseConcMarkSweepGC"
```

执行如下命令以确定参数是否添加成功：

```
echo $JAVA_OPTS
```

在本例中，应该会有如下输出：

```
-XX:+UseParNewGC -XX:+UseConcMarkSweepGC
```

使用服务包装器

可以使用 Java 服务包装器（https://github.com/ElasticSearch/ElasticSearch-servicewrapper）把 ElasticSearch 包装成服务。这种给 ElasticSearch 添加启动参数的方法与使用标准启动脚本的方法不同。

我们所需要做的是修改 elasticsearch.conf 文件，该文件一般位于 /opt/ElasticSearch/bin/service/ 目录下（如果 ElasticSearch 安装目录是 /opt/ElasticSearch）。文件中包含了以下属性：

```
set.default.ES_HEAP_SIZE=1024

wrapper.java.additional.1=-Delasticsearch-service
wrapper.java.additional.2=-Des.path.home=%ES_HOME%
wrapper.java.additional.3=-Xss256k
wrapper.java.additional.4=-XX:+UseParNewGC
wrapper.java.additional.5=-XX:+UseConcMarkSweepGC
wrapper.java.additional.6=-XX:CMSInitiatingOccupancyFraction=75
wrapper.java.additional.7=-XX:+UseCMSInitiatingOccupancyOnly
wrapper.java.additional.8=-XX:+HeapDumpOnOutOfMemoryError
wrapper.java.additional.9=-Djava.awt.headless=true
```

第一行负责设置 ElasticSearch 的堆空间大小，之后的其他行是附加 JVM 参数。如果需要添加其他参数，只需新增一行 wrapper.java.additional，后面紧跟下一个可用数字，例如：

```
wrapper.java.additional.10=-server
```

垃圾回收调优并不是一件一劳永逸的事情。它需要反复试验，因为调优结果严重依赖于你的数据、查询和各种组合环境。在 ElasticSearch 出故障的时候要勇于改变和调优，在做出改变后也要继续观察、监控 ElasticSearch 的运行状况。

6.1.3 在类 UNIX 系统中避免内存交换

弄明白如何禁用内存交换是至关重要的，即使这跟垃圾回收和堆空间使用没有直接关系。内存交换是一个把内存中的页（page）写入外存磁盘（Linux 系统中指 swap 分区）的过程，且发生在物理内存不足时，或者操作系统由于某些原因需要把部分内存数据写入磁盘时。如果有进程请求访问被换出内存的页，那么操作系统就会把它们从交换分区中读出来，放入内存并允许进程访问。显然这个过程是需要消耗时间和资源的。

使用 ElasticSearch 时，我们要确保它的内存空间不会被换出。可以想象一下，如果让 ElasticSearch 的部分内存交换到磁盘中，紧接着读取这块被换出的数据，就会对查询和索引的性能造成负面影响。因此 ElasticSearch 允许我们关闭针对它的内存交换，具体通过设置 elasticsearch.yml 的 bootstrap.mlockall 为 true 来实现。

除了前面的设置，我们还需要确保 JVM 的堆大小固定。要做到这一点，我们需要设置 Xmx 和 Xms 参数为相同值（或者设置 ES_MAX_MEM 和 ES_MIN_MEM 为相同值）。仍需谨记，要有足够的物理内存来支持以上设置。

此时运行 ElasticSearch，可以看到如下日志：

```
[2013-06-11 19:19:00,858][WARN ][common.jna                ]
  Unknown mlockall error 0
```

这个错误意味着内存锁定未起作用，因而我们还需修改两个系统文件（需要系统管理员权限）。在做修改之前，假定运行 ElasticSearch 服务的用户为 elasticsearch。

首先修改 /etc/security/limits.conf 文件，添加如下两行记录：

```
elasticsearch - nofile 64000
elasticsearch - memlock unlimited
```

然后修改 /etc/pam.d/common-session 文件，添加一行：

```
session required pam_limits.so
```

重新用 elasticsearch 用户登录后，再次运行 ElasticSearch，就不会看到 mlockall error 的日志了。

6.2 关于 I/O 调节

在 5.1 节中，我们讨论了存储类型，它们能配置适合特定需求的存储模块。然而，我们并没有深入讨论存储模块的方方面面，甚至都没有提及 I/O 调节，因而这就是接下来的任务。

6.2.1 控制 IO 节流

在 3.6 节中我们了解到，Apache Lucene 把索引数据保存在可一次写入多次读取的不可变索引段中。索引合并的过程是异步的，从 Lucene 的角度看是不会干扰索引和查询过程的。然而，这很可能会出现问题，因为合并操作非常消耗 I/O，需要先读取旧索引段，然后合并写入新索引段中。如果在此同时进行查询和索引，那么 I/O 子系统的负荷会非常大，这个问题在那些 I/O 速度较慢的系统中表现得尤为突出。这就是 I/O 节流的切入点。我们可以控制 ElasticSearch 使用的 I/O 量。

6.2.2 配置

在节点级和索引级都可以配置 I/O 节流。这意味着你可以分别配置节点和索引的资源使用量。

节流类型

在节点级配置节流，可以使用 indices.store.throttle.type 属性。它支持 none、merge、all 这三个属性值：none 为默认值，表示不作任何限制；merge 表示在节点上进行索引合并时限制 I/O 使用量；All 表示对所有基于存储模块的操作都做 I/O 限制。

在索引级配置节流，可以使用 index.store.throttle.type 属性。它除了支持 indices. store. throttle.type 的所有属性值，还支持一个默认的 node 属性值，即表示使用节点级配置取代索引级配置。

每秒最大吞吐量

在上面两种配置中，无论使用索引级还是节点级的节流配置，都可以设置 I/O 可使用的每秒最大字节数，如设置为 10MB、500MB 或任意我们需要的值。如果是索引级的配置，

可以使用 index.store.throttle.max_bytes_per_sec 属性，而如果是节点级的配置，则可以使用 indices.store.throttle. max_bytes_per_sec 属性。

以上配置都可以通过 elasticsearch.yml 文件配置，也可以动态更新：使用集群更新设置接口来更新节点级配置，使用索引更新设置接口来更新索引级配置。

节点的默认节流配置

在节点级，I/O 节流从 ElasticSearch 0.90.1 版本起就默认开启，其中 indices.store.throttle.type 的属性值为 merge，indices.store.throttle.max_bytes_per_sec 的属性值为 20MB。

配置示例

假定有一个 4 节点的集群，我们需要给整个集群配置 I/O 节流，并希望单节点的索引合并操作每秒最多处理 50MB 的数据。因为我们知道在这个限制下不会影响查询性能，而这正是我们的目的。为此，我们需要执行如下命令：

```
curl -XPUT 'localhost:9200/_cluster/settings' -d '{
  "persistent" : {
    "indices.store.throttle.type" : "merge",
    "indices.store.throttle.max_bytes_per_sec" : "50mb"
  }
}'
```

此外，我们还有一个名为 payments 的索引。这个索引极少使用，而我们把它放在整个集群中最小的那台机器上。该索引是单分片的且没有副本。如果要限制它的索引合并操作，只允许它每秒最多处理 10MB 的数据，那么在上一个命令的基础上，我们还需要执行如下命令：

```
curl -XPUT 'localhost:9200/payments/_settings' -d '{
  "index.store.throttle.type" : "merge",
  "index.store.throttle.max_bytes_per_sec" : "10mb"
}'
```

然而，这个请求命令目前并未生效，因为 ElasticSearch 还没有刷新 payments 索引的 I/O 节流设置。要实现这一点，需要先关闭再重新打开 payments 索引，从而强制刷新。使用如下命令：

```
curl -XPOST 'localhost:9200/payments/_close'
curl -XPOST 'localhost:9200/payments/_open'
```

然后，可以通过执行如下命令检查索引设置：

```
curl -XGET 'localhost:9200/payments/_settings?pretty'
```

并得到如下 JSON 响应：

```
{
  "payments" : {
    "settings" : {
      "index.number_of_shards" : "5",
      "index.number_of_replicas" : "1",
      "index.version.created" : "900099",
      "index.store.throttle.type" : "merge",
      "index.store.throttle.max_bytes_per_sec" : "10mb"
    }
  }
}
```

如你所见，通过更新索引设置、关闭再重新打开索引，最终使配置变更生效。

6.3 用预热器提升查询速度

如果在工作中用到 ElasticSearch，那你很可能听说或使用过预热器 API。这个实用接口允许我们预添加一些常用查询来预热索引段。预热器的功能恰恰如此（一个或多个注册到 ElasticSearch 的查询，用来预先准备好要查询的索引）。本节我们将回顾预热器的添加、管理和使用场景。

6.3.1 为什么使用预热器

也许你会疑虑，预热器是否真的那么有用。答案取决于索引数据和查询，但一般来说，它们是有用的。我们之前提到过（如 5.3 节中关于缓存的讨论），ElasticSearch 需要提前加载一些数据到缓存中，目的是使用一些如父 - 子关系、切面计算和基于字段的排序等特定功能。预加载过程需要花费一些时间和资源，在某些时候会使查询变慢。更要命的是，如果索引频繁更新，缓存就需要频繁刷新，而查询性能也就更糟了。

这也是 ElasticSearch 0.20 版本引入预热器 API 的原因。预热器是一些标准查询，这些查询在 ElasticSearch 尚未对外提供查询服务时，先在冷的（尚未使用的）索引段上执行。查询操作不仅会在 ElasticSearch 启动时执行，在新索引段提交后也会执行。因此，使用一些合适的查询对系统进行预热后，我们会把所有需要的数据加载到缓存中，并且预热了操作系统的 I/O 缓存（通过加载冷的索引段）。做完这些之后，当索引段最终接受查询请求时，我们能确保得到最优的查询性能，且此时所需的数据都已经就位。

在本节最后，我们会展示一个小例子，用来描述预热器是如何提升查询性能的。请注意使用预热器前后的性能区别。

6.3.2 操作预热器

ElasticSearch 允许我们创建、检索和删除预热器。每个预热器都关联了一个索引，或者索引和类型的组合。我们在如下场合引入预热器：创建索引的请求中携带预热器；模板

中包含预热器；使用 PUT 预热器 API 创建预热器。当然也可以完全禁用所有预热器而不是先删除它们，所以在不需要它们时，可以简单地禁用它们。

使用 PUT Warmer API

添加预热器最简单的方法是使用 PUT Warmer API。为此我们需要给 _warmer REST 端点发送一个带查询的 HTTP PUT 请求。例如，要给 mastering 索引的 doc 类型添加一个简单的 match_all 查询并附带一些基于词项的切面计算，可以使用如下命令：

```
curl -XPUT 'localhost:9200/mastering/doc/_warmer/testWarmer' -d '
  "query" : {
    "match_all" : {}
  },
  "facets" : {
    "nameFacet" : {
      "terms" : {
        "field" : "name"
      }
    }
  }
}'
```

可以看出，每个预热器都有唯一的名称（如本例中的 testWarmer）。我们可以使用这个名称来检索和删除它。

如果你想添加一个针对整个 mastering 索引，除此之外和上面一模一样的预热器，可以使用如下命令：

```
curl -XPUT 'localhost:9200/mastering/_warmer/testWarmer' -d '{
  ...
}'
```

在创建索引时添加预热器

除了可以使用上面的 PUT Warmer API，我们还可以在创建索引时添加预热器。为此，我们需要在请求体中和 mappings 配置节同一层级的地方添加一个 warmers 配置节。例如，要给 mastering 索引和 doc 类型添加跟上一小节中相同的预热器，可以执行如下命令：

```
curl -XPUT 'localhost:9200/mastering' -d '{
  "warmers" : {
    "testWarmer" : {
      "types" : ["doc"],
      "source" : {
        "query" : {
          "match_all" : {}
        },
        "facets" : {
          "nameFacet" : {
            "terms" : {
```

```
            "field" : "name"
          }
        }
      }
    }
  },
  "mappings" : {
    "doc" : {
      "properties" : {
        "name": { "type": "string", "store": "yes", "index":
          "analyzed" }
      }
    }
  }
}'
```

如你所见，在 mappings 配置节之前，添加了一个 warmers 配置节。这个 warmers 节点用来给将要创建的 mastering 索引注入预热器。每个预热器用名称（name）来唯一标识（如本例中的 testWarmer），并且有两个属性：types 和 source。Types 属性表示预热器所作用的索引类型集合，如果该属性值为空，则表示预热器将作用于当前索引下的所有类型。Source 属性用来指定查询语句。此外，在一次索引创建请求中可以一次性指定多个预热器。

在模板中添加预热器

ElasticSearch 同样支持在模板中添加预热器，做法和创建索引时类似。例如，使用如下命令即可为一个简单模板添加预热器：

```
curl -XPUT 'localhost:9200/_template/templateone' -d '{
  "warmers" : {
    "testWarmer" : {
      "types" : ["doc"],
      "source" : {
        "query" : {
          "match_all" : {}
        },
        "facets" : {
          "nameFacet" : {
            "terms" : {
              "field" : "name"
            }
          }
        }
      }
    }
  },
  "template" : "test*"
}'
```

检索预热器

所有预热器都支持检索功能。你只需向 _warmer REST 终端发送一条 HTTP GET 请求

即可。例如，可以通过名称来检索某个预热器：

```
curl -XGET 'localhost:9200/mastering/_warmer/warmerOne'
```

也可以使用通配符检索名字带有特定前缀的预热器。例如，检索出所有名称以 w 开头的预热器：

```
curl -XGET 'localhost:9200/mastering/_warmer/w*'
```

还可以获取某个索引下的所有预热器：

```
curl -XGET 'localhost:9200/mastering/_warmer/'
```

当然，还可以在上面这几条命令中引入索引类型，从而找出指定索引类型的预热器。

删除预热器

和检索接口类似，ElasticSearch 也支持通过 REST 终端来删除预热器，只需向 _warmer 终端发送一条 HTTP DELETE 请求即可。例如，要从 mastering 索引中删除名字为 warmerOne 的预热器，可以使用如下命令：

```
curl -XDELETE 'localhost:9200/mastering/ warmer/warmerOne'
```

也可以删除名字带特定前缀所有预热器。例如，要从 mastering 索引中删除所有名字以 w 开头的预热器，可以使用如下命令：

```
curl -XDELETE 'localhost:9200/mastering/_warmer/w*'
```

同样也可以删除执行索引下的所有预热器。为此我们需要向 _warmer REST 终端发送不带预热器名称的 HTTP DELETE 请求。例如，使用如下命令可删除 mastering 索引的所有预热器：

```
curl -XDELETE 'localhost:9200/mastering/_warmer/'
```

当然，和检索接口类似，在上面这几条命令中还可以指定索引类型，用来删除指定索引类型的预热器。

禁用预热器

预热器如果暂不使用，又不想删除，则可以禁用它们，只需将 index.warmer.enabled 属性设置为 false 即可。这个属性可以在 elasticsearch.yml 文件中设置，也可以通过更新设置 API 设置，例如：

```
curl -XPUT 'localhost:9200/mastering/_settings' -d '{
  "index.warmer.enabled": false
}'
```

当你想再次使用预热器时，也只需将 index.warmer.enabled 属性更改为 true 即可。

6.3.3 测试预热器

为了测试预热器，我们需要测试一下。首先使用如下命令创建一个简单索引：

```
curl -XPUT localhost:9200/docs -d '{
  "mappings" : {
    "doc" : {
      "properties" : {
        "name": { "type": "string", "store": "yes", "index":
          "analyzed" }
      }
    }
  }
}'
```

接着创建一个名为 child 的索引类型，作为 doc 索引类型的子类型。其中，child 类型文档是 doc 类型文档的子文档。具体实现可以使用如下命令：

```
curl -XPUT 'localhost:9200/docs/child/_mapping' -d '{
  "child" : {
    "_parent":{
      "type" : "doc"
  },
  "properties" : {
    "name": { "type": "string", "store": "yes", "index":
      "analyzed" }
    }
  }
}'
```

然后，我们给 doc 类型索引了一个文档，并给 child 类型索引了 80 000 个文档。所有这些 child 类型的文档在索引时都使用 parent 参数指向那个唯一的 doc 类型文档。

查询时不适用预热器

完成上面的索引过程后，重启 ElasticSearch。然后，执行如下查询：

```
{
  "query" : {
    "has_child" : {
      "type" : "child",
      "query" : {
        "term" : {
          "name" : "document"
        }
      }
    }
  }
}
```

如你所见，这个简单查询会返回一个父文档，且它的子文档中至少有一个包含有指定的词项。ElasticSearch 给出的响应如下：

```
{
  "took" : 479,
  "timed_out" : false,
  "_shards" : {
    "total" : 1,
    "successful" : 1,
    "failed" : 0
  },
  "hits" : {
    "total" : 1,
    "max_score" : 1.0,
    "hits" : [ {
      "_index" : "docs",
      "_type" : "doc",
      "_id" : "1",
      "_score" : 1.0, "_source" : {"name":"Test 1234"}
    } ]
  }
}
```

执行时间为 479 毫秒。看起来很长吧？而如果再次执行这个查询，执行时间就会缩短。

查询时使用预热器

为了提升初始查询的性能，我们需要引入一个简单的预热器。这个预热器不仅能预热 I/O 缓存，还能迫使 ElasticSearch 把父文档的 id 列表加载到内存中，以支持更快的父子查询。正如我们所知，该引入操作会在 ElasticSearch 首次处理这种带有父子关系的查询时执行。基于以上信息，我们可以使用下面的命令来添加预热器：

```
curl -XPUT 'localhost:9200/docs/_warmer/sampleWarmer' -d '{
  "query" : {
    "has_child" : {
      "type" : "child",
      "query" : {
        "match_all" : {}
      }
    }
  }
}'
```

接下来，重启 ElasticSearch 并执行与上一小节（6.3.3 节）中相同的查询，会得到类似如下结果：

```
{
  "took" : 38,
  "timed_out" : false,
  "_shards" : {
    "total" : 1,
    "successful" : 1,
    "failed" : 0
```

```
  },
  "hits" : {
    "total" : 1,
    "max_score" : 1.0,
    "hits" : [ {
      "_index" : "docs",
      "_type" : "doc",
      "_id" : "1",
      "_score" : 1.0, "_source" : {"name":"Test 1234"}
    } ]
  }
}
```

我们可以看出，在 ElasticSearch 重启后，预热器的查询性能确实得到了改善。未使用预热器的查询耗时接近半秒钟，而使用了预热器的查询耗时不到 40 毫秒。

当然，性能的优化不仅归功于 ElasticSearch 把父文档 id 列表加载到内存中，还因为操作系统对相关索引段做了缓存。不管怎么说，性能提升是显著的。因此，对那些可以使用预热器提速的查询（如重量级过滤、父子文档关系、切面计算等）使用预热器确实是一个好办法。

6.4 热点线程

当集群变慢且占用较多 CPU 资源时，你有必要采取一些处理措施来恢复它。然而，热点线程 API 就提供了必要的信息，帮助你找到问题根源。热点线程指 CPU 占用高且执行时间较长的 Java 线程，而热点线程 API 可以返回如下信息：从 CPU 视角看到的 ElasticSearch 中执行最频繁的代码段，以及 ElasticSearch 卡在什么地方。

热点线程 API 可以检查所有 ElasticSearch 节点、部分节点或某个特殊节点，只需使用 /_nodes/hot_threads 或 /_nodes/{node or nodes}/hot_threads 端点即可。例如，使用下面的命令来检查所有节点上的热点线程：

```
curl localhost:9200/_nodes/hot_threads
```

热点线程 API 支持如下参数：

- threads（默认值为 3）：经分析后输出的线程数。ElasticSearch 会根据 type 参数指定的信息挑选出最“热”的 threads 个线程。
- interval（默认值 500ms）：ElasticSearch 需要分两次检查线程，目的是计算特定线程与 type 参数对应操作的耗时百分比。两次检查的间隔时间由 interval 参数设置。
- type（默认值为 cpu）：本参数确定了要检查的线程状态的类型，具体支持如下状态类型：指定线程的 CPU 耗时（cpu）、BLOCK 状态耗时（block）、WAITING 状态耗时（wait）。如果你想了解更多线程状态信息，请参见：http://docs.oracle.com/javase/6/docs/api/java/lang/Thread.State.html。
- snapshots（默认值 10）：堆栈轨迹快照的数量。其中，堆栈轨迹指特定时间点的嵌

套函数调用。

接下来演示关于热点线程 API 及上述参数使用的示例。例如，执行如下命令可以让 ElasticSearch 输出处在 WAITING 状态的线程的信息，检查间隔为 1 秒：

```
curl 'localhost:9200/_nodes/hot_threads?type=wait&interval=1s'
```

6.4.1 澄清热点线程 API 的用法误区

热点线程 API 的输出与众不同。其他 API 的输出都是 JSON 格式，而热点线程 API 的输出是格式化的、可分为几个部分的文本。在此需要先阐述一下热点线程 API 背后的逻辑。ElasticSearch 首先扫描所有正在运行的线程并从它们那里收集多种信息，包括每个线程占用 CPU 的时间、特定线程被阻塞或处于等待状态的次数、阻塞或等待时长等。然后，等待一定的时长（由 interval 参数指定），之后再次扫描并收集同样的信息。接下来把两次收集得到的信息按线程执行时长做降序排序，当然，这里的执行时间是指由 type 参数指定的特定操作的执行时间。之后，ElasticSearch 分析前 N 个线程（N 为 threads 参数的取值）。实际上 ElasticSearch 的做法是，每隔几毫秒就截取一些前 N 个线程的堆栈轨迹快照（快照数由 snapshots 参数指定）。最后，ElasticSearch 把这些堆栈轨迹分组并可视化呈现线程状态的变化。

6.4.2 热点线程 API 的响应信息

接下来我们浏览一下热点线程 API 响应信息的各个部分。例如，下面这个截图截取自一个刚启动的 ElasticSearch 的热点线程 API 响应：

```
> curl -XGET 'localhost:9200/_nodes/hot_threads'
::: [Ozone][_EDivmEeQZGlhbCrKB9ljQ][inet[/192.168.0.100:9300]]

    0,1% (308micros out of 500ms) cpu usage by thread 'elasticsearch[Ozone][transport_client_timer][T#1]{Hashed wheel timer
 #1}'
     10/10 snapshots sharing following 5 elements
       java.lang.Thread.sleep(Native Method)
       org.elasticsearch.common.netty.util.HashedWheelTimer$Worker.waitForNextTick(HashedWheelTimer.java:467)
       org.elasticsearch.common.netty.util.HashedWheelTimer$Worker.run(HashedWheelTimer.java:376)
       org.elasticsearch.common.netty.util.ThreadRenamingRunnable.run(ThreadRenamingRunnable.java:108)
       java.lang.Thread.run(Thread.java:722)

    0,0% (228micros out of 500ms) cpu usage by thread 'elasticsearch[Ozone][[timer]]'
     10/10 snapshots sharing following 2 elements
       java.lang.Thread.sleep(Native Method)
       org.elasticsearch.threadpool.ThreadPool$EstimatedTimeThread.run(ThreadPool.java:607)

    0,0% (103micros out of 500ms) cpu usage by thread 'elasticsearch[Ozone][http_server_worker][T#4]{New I/O worker #31}'
     10/10 snapshots sharing following 15 elements
       sun.nio.ch.KQueueArrayWrapper.kevent0(Native Method)
       sun.nio.ch.KQueueArrayWrapper.poll(KQueueArrayWrapper.java:159)
       sun.nio.ch.KQueueSelectorImpl.doSelect(KQueueSelectorImpl.java:103)
       sun.nio.ch.SelectorImpl.lockAndDoSelect(SelectorImpl.java:87)
       sun.nio.ch.SelectorImpl.select(SelectorImpl.java:98)
       org.elasticsearch.common.netty.channel.socket.nio.SelectorUtil.select(SelectorUtil.java:64)
       org.elasticsearch.common.netty.channel.socket.nio.AbstractNioSelector.select(AbstractNioSelector.java:409)
       org.elasticsearch.common.netty.channel.socket.nio.AbstractNioSelector.run(AbstractNioSelector.java:206)
       org.elasticsearch.common.netty.channel.socket.nio.AbstractNioWorker.run(AbstractNioWorker.java:88)
       org.elasticsearch.common.netty.channel.socket.nio.NioWorker.run(NioWorker.java:178)
       org.elasticsearch.common.netty.util.ThreadRenamingRunnable.run(ThreadRenamingRunnable.java:108)
       org.elasticsearch.common.netty.util.internal.DeadLockProofWorker$1.run(DeadLockProofWorker.java:42)
       java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1110)
       java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:603)
       java.lang.Thread.run(Thread.java:722)

>
```

现在来探讨一个更有用的示例。响应的第一部分是一则通用响应头信息：

```
::: [Cecilia][0d7etUE3TQuVi9kGRsOpuw][inet[/192.168.0.100:9300]]
```

从头信息中我们可以定位到具体的 ElasticSearch 节点，这在热点线程 API 向多个节点发出调用请求时很有用。

下一行可分为几部分，开头类似下面的信息：

```
22.5% (112.3ms out of 500ms) cpu usage by thread
  'elasticsearch[Cecilia][search][T#993]'
```

在本例中我们看到，名为 search 的线程在分析抽样时占用了 22.5% 的 CPU 时间。cpu usage 文本揭示出当前的 type 参数是 cpu（该文本也可能是 block usage 和 wait usage，分别代表被阻塞的线程和处于等待状态的线程）。线程名称非常重要，因为就是根据名称来推测 ElasticSearch 的哪部分功能出问题了。比如在本例中线程名为 search，因此我们认为这个线程和查询有关。此外，线程名称的取值还包括 recovery_stream（关于恢复模块）、cache（关于缓存）、merge（关于段合并线程）、index（关于数据索引线程）等。

响应的下一部分以如下信息开头：

```
10/10 snapshots sharing following 5 elements
```

紧接着上面的信息之后出现的是堆栈轨迹快照。其中，10/10 表示采集了 10 个一模一样的堆栈轨迹快照。一般这意味着当前线程在检查期间都在忙于执行同一段 ElasticSearch 代码。

6.5 现实场景

本节在内容编排上和上一节有所不同。在这里我们不想阐述 ElasticSearch 的各种功能，而是想带你展开一段简短的旅程，以便向你展示如何使用不同的 API 来查看 ElasticSearch 集群正在做什么。注意，这是我们曾经遇到的真实场景。

6.5.1 越来越差的性能

我们把 ElasticSearch 成功部署到生产环境，起初一切正常。然而过了一段时间，系统管理员进来向我们示警说，监控系统监测到一些性能下降，查询延迟提高了 30%。也许我们不必为此担心（而事实上必须担心），不过从长远来看，确实有必要做点什么了。

首先让我们连上 ElasticSearch 集群，收集一些统计信息。执行下面命令：

```
curl 'localhost:9200/_stats?pretty'
```

关于这条命令的详细描述不在本书论述范围内。如果你想要了解更多，请阅读我们的上一本书《 ElasticSearch Server 》，或者访问：http://www.ElasticSearch.org/guide/reference/api/admin-indices-stats/。

命令返回了大量信息，其中包括所有索引的信息，以及根据集群中每个索引分组的信息。我们的系统管理员有一个习惯，即他喜欢每小时执行一次这条命令并把执行结果保存下来，然而这样我们就可以对比前一次的执行结果并识别出哪些地方发生了变化（保存历史结果是值得的，不是吗？），进而判断发生了什么。本例中，索引拥有的文档数几乎没有增减，而且索引写入磁盘后都很少更改。因此我们可以判定，性能下降和索引过程无关。索引统计信息也看不出任何有趣的东西。然而，get 和 search 操作的统计数据却让人难以捉摸：

```
"get" : {
  "total" : 408709,
  "time" : "3.6h",
  "time_in_millis" : 13048877,
  "exists_total" : 359320,
  "exists_time" : "2.9h",
  "exists_time_in_millis" : 10504331,
  "missing_total" : 49389,
  "missing_time" : "42.4m",
  "missing_time_in_millis" : 2544546,
  "current" : 0
},
"search" : {
  "query_total" : 136704,
  "query_time" : "15.1h",
  "query_time_in_millis" : 54427259,
  "query_current" : 0,
  "fetch_total" : 84127,
  "fetch_time" : "10.8m",
  "fetch_time_in_millis" : 648088,
  "fetch_current" : 0
}
```

无论你是否相信，我们发现相比上次统计结果，从 ElasticSearch 读取的数据量上升了。让我们做点数学计算：在 3.6 小时内 ElasticSearch 处理了 408 709 个 get 请求，每个请求大约耗时 32 毫秒；而对于 search 请求，ElasticSearch 共花费了 15.1 小时，处理了 136 704 个请求，每个请求约耗时 400 毫秒。这些数据本身并不能说明压力分布情况，而且单靠平均响应时间也不能说明任何性能问题，除非拿它和历史记录做对比。请求数量的增长在此显得尤为重要（和历史同期相比 ElasticSearch 处理了更多的 search 和 get 请求）。我们期望达到的目标是：保持平均响应时间和请求量增长之前相同。要实现这一目标最简单的方法也许就是把请求分流到更多 ElasticSearch 节点上，这意味着我们要添加新的节点，再把部分索引分片放置到新节点上。这里我们需要做的是增加副本的数量，而这一点 ElasticSearch 可以轻易实现。例如，对于 message 索引，可以执行如下命令（假定在本命令执行前副本数不超过 2 个）：

```
curl -XPUT localhost:9200/messages/_settings -d '{
  "index.number_of_replicas" : 3
}'
```

然后，还需要执行如下命令来查看集群状态：

```
curl localhost:9200/_cluster/state?pretty
```

此前集群处于平衡状态，而执行完上面两条命令后我们发现，集群中出现了未分配分片（unsigned shard）：

```
{
  "state" : "UNASSIGNED",
  "primary" : false,
  "node" : null,
  "relocating_node" : null,
  "shard" : 1,
  "index" : "messages"
}
```

假如使用了其他报告插件如 paramedic 或 head，你可以更清晰地看到，索引拥有未分配的分片。这些分片将在新节点加入集群后被放置到新节点上。如果现在启动一个新节点，那么过一段时间 ElasticSearch 就会把所有未分配的分片（或部分未分配的分片，依赖于具体设置）分配到这个新节点上，而这个新节点也将自动接手一部分请求处理工作。此时唯一要做的是，过一段时间去向系统管理员要新的统计数据，然后检查系统负载是否下降了（或者不找系统管理员，而是自行通过一些自动监控系统来收集数据）。

6.5.2 混杂的环境和负载不平衡

接下来要分享的例子是关于索引期间性能下降问题的。但这里我们遇到了一个难题：业务需求强调新消息的快速发布。因此，必须提高索引操作速度并减少或消除因此带来的性能影响。我们首先想到的方法是用性能更强的机器来支撑所有索引操作。

通过上一个例子我们知道，可以使用索引状态管理 API 来收集部分流量相关信息。这里再次使用这一招：

```
curl 'localhost:9200/_stats?pretty'
```

响应消息中最有趣的部分（或者说我们最关注的）是下面这个片段：

```
"indexing" : {
  "index_total" : 1475514,
  "index_time" : "1.2h",
  "index_time_in_millis" : 4400308,
  "index_current" : 167,
  "delete_total" : 2608506,
  "delete_time" : "1.3h",
  "delete_time_in_millis" : 4997302,
  "delete_current" : 0
}
```

这个结果已经很不错了，然而我们并不满足于此。正如之前所说，索引操作可以迁

移到性能更强的机器上。这不是问题，因为我们有几台不同硬件配置的机器，而且可以在它们之间迁移索引数据（请完整阅读第4章以了解更多关于分片分配信息）。最初我们的ElasticSearch集群由相同配置的节点组成，每个节点承担相等的职责。当然，集群中会有一个主节点（master node），但这个角色是公平选举出来的，每个节点都可能被选中。然而现在面临的问题是：怎样才能支持特殊节点。在5.2.2节中，我们介绍了node.data和node.master的相关设置，现在让我们回顾一下这些知识，然后再判断应该给ElasticSearch节点安排哪个角色：

- node.data = true 且 node.master = true：这是标准配置。节点可以存储索引数据、处理查询请求、可以被选为主节点。
- node.data = true 且 node.master = false：这种情况下，节点永远不会被选为主节点，但仍可以存储索引数据、处理查询请求。
- node.data = false 且 node.master = true：这种情况下，节点可以被选为主节点，也可以处理查询请求，但是不存储索引数据。
- node.data = false 且 node.master = false：这种节点永远不会被选为主节点，也不会存储索引数据，但仍可以处理查询请求。

在此澄清一下。如你所见，基于ElasticSearch的分布式特性，节点即使在不存储数据的情况下也能处理查询请求。当它接收到查询请求时，会把请求转发给存储了必要索引分片的节点，并由那些节点去执行查询请求（你可以在4.5节中了解到关于ElasticSearch如何选择和控制分片的知识）。那些由持有数据的节点所返回的结果，是先经过ElasticSearch某节点的合并和处理，才发送给请求客户端的。在更复杂的查询中（如切面计算），合并、加工数据的过程是非常消耗资源的。我们大量使用了切面计算，因此我们决定将其中一部分节点分离出来，命名为聚合器节点，即该节点不持有数据，不做主节点，只负责处理查询请求。多亏了这类节点，我们可以把请求只发送给它们而不会给数据节点带来聚合操作的压力。总之，这意味着给予了数据节点处理更多索引请求的能力。

6.5.3 我的服务器出故障了

还有一个小问题。在某些情况下，集群中的某个ElasticSearch节点的负载会变得很高，且无法轻易发现问题原因。如果你是一名咨询顾问，那么你可能是最终的求助对象。而且这一次，解决问题的时间会比较长，你需要一直等待问题的再次出现（因为没有监控软件，而且负载高的节点也被重启了），然后执行如下命令：

```
curl localhost:9200/_nodes/hot_threads
```

查看结果你会发现，负责处理查询的线程数量极多，而只有少量的索引线程和更少的缓存线程。而且有一个用来合并索引段的线程，值得我们做进一步检查。片刻之后，通过使用iostat系统命令（我们使用linux系统）我们发现I/O读写非常频繁，因此导致查询请求大量堆积，响应时间增加，最终致使节点不能响应。“让我们买SSD硬盘吧”，这是我们

能想到的一个好方法。但在这里，我们决定采用另一个方法，即限制索引段合并线程的 I/O 操作。更多关于如何限制 I/O 操作的信息请参见 6.2 节。我们使用下面的命令来调节 I/O：

```
curl -XPUT 'localhost:9200/_cluster/settings' -d '{
  "persistent" : {
    "indices.store.throttle.type" : "merge",
    "indices.store.throttle.max_bytes_per_sec" : "20mb"
  }
}'
```

这个命令最终解决了问题。我们反复测试合适的节流阈值，并将取值最终确定为 20mb。这个值对本例来说足够了，当索引段合并时不会干扰节点的正常工作。

6.6 小结

本章我们了解了垃圾回收器的概念，以及它是如何工作并在发生问题时做出诊断的。我们还掌握了如何控制存储模块的 I/O 操作数，见识了预热器给查询带来的速度提升。接着我们认识了热点线程，并学会了通过 ElasticSearch 的 API 去获取热点线程的信息，以及如何解释热点线程 API 的响应输出。最后我们尝试利用 ElasticSearch 的 API 来获取统计信息，并进一步利用这些信息诊断了 ElasticSearch 的各种故障。

下一章我们主要聚焦于提升用户体验，包括使用新引入的搜索提示 API 去纠正用户的拼写错误，用切面计算帮助用户快速找到所需信息，以及使用 ElasticSearch 支持的不同查询类型来改进查询相关性。