placeholder7

管理、　控和部署

本　大部分介　了使用　　Elasticsearch　　作　后端　建　用程序。本章　　微不同。在　里，　将学　到如何管理　Elasticsearch　自身。Elasticsearch　是一个　　的　件，有　多可移　　件，大量的　API　　用来　助管理　的 Elasticsearch 部署。

在　个章　，我　涵　三个主　：

- 根据　控　的集群重要数据的　　，去了解　些行　是正常的，　些　　引起警告，并解　Elasticsearch 提供的各　　信息。
- 部署　的集群到生　　境，包括最佳　践和　　（或不　　！）修改的重要配置。
- 部署后的　　，如 Rolling Restart 或　　的集群

控

Elasticsearch　　常以多　点集群的方式部署。有多　　API　　可以管理和　控集群本身，而不用和集群里存　的数据打交道。

和　Elasticsearch　里　大多数功能一　，我　有一个　体的　目　，即任　　通　　API　行，而不是通　修改静　的配置文件。　一点在　的集群　容　尤　重要。即便通　配置管理系　（比如 Puppet，Chef　或者　Ansible），一个　的　HTTP　API　　用，也比往上百台物理上推送新配置文件　　多了。

因此，本章将介　各　可以　　整、　和　配集群的　　API。同　，　会介　一系列提供集群自身　　数据的 API，　可以用　些接口来　控集群健康状　和性能。

Marvel　控

Marvel　　可以很　的通　　Kibana　　控　Elasticsearch。　可以　　看　的集群健康状　和性能，也可以分析　去的集群、索引和　点指　。

　然　可以通　本章介　的　API　　看大量的指　数据，但是它　展示的都是当前　　点的即　情况。了解　个瞬　的内存占用比当然很有用，但是了解内存占用比　随　　的　　更加有用。Marvel　会　并聚合　些数据，　可以通　可　化效果看到自己集群随　　的　化，　　可以很容易的　　展的　　。

随着　集群　模的　展，　　API　　的　出内容会　　得　人完全没法看。当　有一大把　点，比如　一百个，再　　个　出的　　JSON　　就非常乏味了。而　　Marvel　　可以　　交互式的探索　些数据，更容易于集中　注特定　点或者索引上　生了什　。

Marvel　　使用公　的　　API，和　自己能　到的一　— 它没有暴露任何　通　　API　　不到的　　信息。但是，Marvel　大的　化了　些　　信息的采集和可　化工作。

Marvel　可以免　使用（包括生　　境上！），所以　　在就　始用起来　！安装介　，参　Marvel　入　。

集群健康

一个 Elasticsearch 集群至少包括一个 点和一个索引。或者它可能有一百个数据 点、三个 独的主 点，以及一小打客 端 点—— 些共同操作一千个索引（以及上万个分片）。

不管集群 展到多大 模， 都会想要一个快速 取集群状 的途径。Cluster Health API 充当的就是 个角色。 可以把它想象成是在一万英尺的高度 瞰集群。它可以告 安心 一切都好，或者警告 集群某个地方有 。

我 行一下 cluster-health API 然后看看 体是什 子的：

```
GET _cluster/health
```

和 Elasticsearch 里其他 API 一 ，cluster-health 会返回一个 JSON 。 自 化和告警系 来 ，非常便于解析。 中包含了和 集群有 的一些 信息：

```
{
   "cluster_name": "elasticsearch_zach",
   "status": "green",
   "timed_out": false,
   "number_of_nodes": 1,
   "number_of_data_nodes": 1,
   "active_primary_shards": 10,
   "active_shards": 10,
   "relocating_shards": 0,
   "initializing_shards": 0,
   "unassigned_shards": 0
}
```

信息中最重要的一 就是 status 字段。状 可能是下列三个 之一：

green
所有的主分片和副本分片都已分配。 的集群是 100% 可用的。

yellow
所有的主分片已 分片了，但至少 有一个副本是 失的。不会有数据 失，所以搜索 果依然是完整的 。不 ， 的高可用性在某 程度上被弱化。如果 更多的 分片消失， 就会 数据了。把 yellow 想象成一个需要及 的警告。

red
至少一个主分片（以及它的全部副本）都在 失中。 意味着 在 少数据：搜索只能返回部分数据，而 分配到 个分片上的写入 求会返回一个 常。

green/yellow/red 状 是一个概 的集群并了解眼下正在 生什 的好 法。剩下来的指 列出来集群的状 概要：

- number_of_nodes 和 number_of_data_nodes 个命名完全是自描述的。
- active_primary_shards 指出 集群中的主分片数量。 是涵 了所有索引的 。

- active_shards 是涵　了所有索引的_所有_分片的　　，即包括副本分片。
- relocating_shards　示当前正在从一个　点　往其他　点的分片的数量。通常来　　是　0，不　在 Elasticsearch　　集群不太均衡　，　　会上　。比如　：添加了一个新　点，或者下　了一个　点。
- initializing_shards　是　　建的分片的个数。比如，当　　建第一个索引，分片都会短　的　于 initializing　状　。　通常会是一个　　事件，分片不　　期停留在　initializing　状　。　可能在　点　重　的　候看到　initializing　分片：当分片从磁　上加　后，它　会从　initializing 状　　始。
- unassigned_shards　　是已　在集群状　中存在的分片，但是　　在集群里又不着。通常未分配分片的来源是未分配的副本。比如，一个有　5　分片和　1　副本的索引，在　点集群上，就会有　　5　　个未分配副本分片。如果　的集群是　　red　　状　，也会期保有未分配分片（因　　少主分片）。

更深点：　到　　索引

想象一下某天　到　　了，而　　的集群健康状　看起来像是　　：

```
{
   "cluster_name": "elasticsearch_zach",
   "status": "red",
   "timed_out": false,
   "number_of_nodes": 8,
   "number_of_data_nodes": 8,
   "active_primary_shards": 90,
   "active_shards": 180,
   "relocating_shards": 0,
   "initializing_shards": 0,
   "unassigned_shards": 20
}
```

好了，从　个健康状　里我　能推断出什　来？　，我　集群是　red　，意味着我　　数据（主分片　+ 副本分片）了。我　知道我　集群原先有　10　个　点，但是在　个健康状　里列出来的只有　8　个数据　点。有　个数据　点不　了。我　看到有 20 个未分配分片。

　就是我　能收集到的全部信息。那些　失分片的情况依然是个　。我　是　了 20 个索引，　个索引里少 1 个主分片？　是　1 个索引里的 20 个主分片？　是 10 个索引里的各 1 主 1 副本分片？具体是　个索引？

要回答　个　　，我　需要使用 level 参数　cluster-health 答出更多一点的信息：

```
GET _cluster/health?level=indices
```

　个参数会　cluster-health API 在我　的集群信息里添加一个索引清　，以及有　　个索引的　　（状　、分片数、未分配分片数等等）：

```
{
   "cluster_name": "elasticsearch_zach",
   "status": "red",
   "timed_out": false,
   "number_of_nodes": 8,
   "number_of_data_nodes": 8,
   "active_primary_shards": 90,
   "active_shards": 180,
   "relocating_shards": 0,
   "initializing_shards": 0,
   "unassigned_shards": 20
   "indices": {
      "v1": {
         "status": "green",
         "number_of_shards": 10,
         "number_of_replicas": 1,
         "active_primary_shards": 10,
         "active_shards": 20,
         "relocating_shards": 0,
         "initializing_shards": 0,
         "unassigned_shards": 0
      },
      "v2": {
         "status": "red", ①
         "number_of_shards": 10,
         "number_of_replicas": 1,
         "active_primary_shards": 0,
         "active_shards": 0,
         "relocating_shards": 0,
         "initializing_shards": 0,
         "unassigned_shards": 20 ②
      },
      "v3": {
         "status": "green",
         "number_of_shards": 10,
         "number_of_replicas": 1,
         "active_primary_shards": 10,
         "active_shards": 20,
         "relocating_shards": 0,
         "initializing_shards": 0,
         "unassigned_shards": 0
      },
      ....
   }
}
```

① 我　可以看到 v2 索引就是　集群　red 的那个索引。

② 由此明　了，20 个　失分片全部来自　个索引。

一旦我　　要索引的　出，　个索引有　　立　就很清楚了：v2　　索引。我　　可以看到　个索引曾　有

10 个主分片和一个副本，而 在 20 个分片全不 了。可以推 ， 20 个索引就是位于从我集群里不 了的那 个 点上。

level 参数 可以接受其他更多 ：

```
GET _cluster/health?level=shards
```

shards 会提供一个 得多的 出，列出 个索引里 个分片的状 和位置。 个 出有 候很有用，但是由于太 会比 用。如果 知道 个索引有 了，本章 的其他 API 得更加有用一点。

阻塞等待状 化

当 建 元和集成 ，或者 和 Elasticsearch 相 的自 化脚本 ，cluster-health API 有一个小技巧非常有用。 可以指定一个 wait_for_status 参数，它只有在状 之后才会返回。比如：

```
GET _cluster/health?wait_for_status=green
```

个 用会 阻塞 （不 的程序返回控制 ）住直到 cluster-health 成 green ，也就是所有主分片和副本分片都分配下去了。 自 化脚本和 非常重要。

如果 建一个索引，Elasticsearch 必 在集群状 中向所有 点广播 个 更。那些 点必 初始化 些新分片，然后 主 点 些分片已 <code>Started</code> 。 个 程很快，但是因 延 ，可能要花 10–20ms。

如果 有个自 化脚本是 (a) 建一个索引然后 (b) 立刻写入一个文 ， 个操作会失 。因 索引没完全初始化完成。在 (a) 和 (b) 之 的 可能不到 1ms —— 延 来 可不 。

比起使用 sleep 命令，直接 的脚本或者 使用 wait_for_status 参数 用 cluster-health 更好。当索引完全 建好，cluster-health 就会 成 green ，然后 个 用就会把控制 交 的脚本，然后 就可以 始写入了。

有效的 是： green 、 yellow 和 red 。 个 回会在 到 要求（或者『更高』）的状 返回。比如，如果 要求的是 yellow ，状 成 yellow 或者 green 都会打 用。

控 个 点

集群健康 就像是光 的一端—— 集群的所有信息 行高度概述。而 点 API 是在一端。它提供一个 人眼花 乱的 数据的数 ，包含集群的 一个 点 。

点 提供的 如此之多，在完全熟悉它之前， 可能都 不清楚 些指 是最 得 注的。我将会高亮那些最重要的 控指 （但是我 鼓励 接口提供的所有指 ——或者用 Marvel —— 因 永 不知道何 需要某个或者 一个 ）。

点 API 可以通 如下命令 行：

```
GET _nodes/stats
```

在 出内容的 ，我 可以看到集群名称和我 的第一个 点：

```
{
   "cluster_name": "elasticsearch_zach",
   "nodes": {
      "UNr6ZMf5Qk-YCPA_L18BOQ": {
         "timestamp": 1408474151742,
         "name": "Zach",
         "transport_address": "inet[zacharys-air/192.168.1.131:9300]",
         "host": "zacharys-air",
         "ip": [
            "inet[zacharys-air/192.168.1.131:9300]",
            "NONE"
         ],
...
```

点是排列在一个哈希里，以 点的 UUID 作 名。 示了 点 属性的一些信息（比如 地址和主机名）。 些 如 点未加入集群 自 很有用。通常 会 是端口用 了，或者 点 定在 的 IP 地址/ 接口上了。

索引部分

索引(indices) 部分列出了 个 点上所有索引的聚合 的 ：

```
    "indices": {
        "docs": {
           "count": 6163666,
           "deleted": 0
        },
        "store": {
           "size_in_bytes": 2301398179,
           "throttle_time_in_millis": 122850
        },
```

返回的 被 入以下部分：

- docs 展示 点内存有多少文 ，包括 没有从段里清除的已 除文 数量。
- store 部分 示 点耗用了多少物理存 。 个指 包括主分片和副本分片在内。如果限流 很大，那可能表明 的磁 限流 置得 低（在段和合并里 ）。

```
        "indexing": {
           "index_total": 803441,
           "index_time_in_millis": 367654,
           "index_current": 99,
           "delete_total": 0,
           "delete_time_in_millis": 0,
           "delete_current": 0
        },
        "get": {
           "total": 6,
           "time_in_millis": 2,
           "exists_total": 5,
           "exists_time_in_millis": 2,
           "missing_total": 1,
           "missing_time_in_millis": 0,
           "current": 0
        },
        "search": {
           "open_contexts": 0,
           "query_total": 123,
           "query_time_in_millis": 531,
           "query_current": 0,
           "fetch_total": 3,
           "fetch_time_in_millis": 55,
           "fetch_current": 0
        },
        "merges": {
           "current": 0,
           "current_docs": 0,
           "current_size_in_bytes": 0,
           "total": 1128,
           "total_time_in_millis": 21338523,
           "total_docs": 7241313,
           "total_size_in_bytes": 5724869463
        },
```

- indexing 示已 索引了多少文 。 个 是一个累加 数器。在文 被 除的 候，数不会下降。 要注意的是，在 生内部 索引 操作的 候， 个 也会 加，比如 文 更新。

  列出了索引操作耗 的 ，正在索引的文 数量，以及 除操作的 似 。

- get 示通 ID 取文 的接口相 的 。包括 个文 的 GET 和 HEAD 求。

- search 描述在活 中的搜索（ open_contexts ）数量、 的 数量、以及自 点 以来在 上消耗的 。用 query_time_in_millis / query_total 算的比 ，可以用来粗略的 的 有多高效。比 越大， 个 花 的 越多， 要考 了。

  fetch 展示了 理的后一半流程（query-then-fetch 里的 *fetch* ）。如果 fetch 耗 比 query 多， 明磁 慢，或者 取了太多文 ，或者可能搜索 求 置了太大的分 （比如， size: 10000 ）。

- `merges` 包括了 Lucene 段合并相 的信息。它会告 目前在 行几个合并，合并 及的文数量，正在合并的段的 大小，以及在合并操作上消耗的 。

  在 的集群写入 力很大 ，合并 非常重要。合并要消耗大量的磁 I/O 和 CPU 源。如果 的索引有大量的写入，同 又 大量的合并数，一定要去 索引性能技巧。

  注意：文 更新和 除也会 致大量的合并数，因 它 会 生最 需要被合并的段 碎片 。

```
      "filter_cache": {
         "memory_size_in_bytes": 48,
         "evictions": 0
      },
      "fielddata": {
         "memory_size_in_bytes": 0,
         "evictions": 0
      },
      "segments": {
         "count": 319,
         "memory_in_bytes": 65812120
      },
      ...
```

- `filter_cache` 展示了已 存的 器位集合所用的内存数量，以及 器被 逐出内存的次数。多的 逐数 可能 明 需要加大 器 存的大小，或者 的 器不太 合 存（比如它 因 高基数而在大量 生，就像是 存一个 `now` 表 式）。

  不 ， 逐数是一个很 定的指 。 器是在 个段的基 上 存的，而从一个小的段里 逐器，代 比从一个大的段里要廉 的多。有可能 有很大的 逐数，但是它 都 生在小段上，也就意味着 些 性能只有很小的影 。

  把 逐数指 作 一个粗略的参考。如果 看到数字很大， 一下 的 器， 保他 都是正常存的。不断 逐着的 器， 怕都 生在很小的段上，效果也比正 存住了的 器差很多。

- `field_data` 示 fielddata 使用的内存，用以聚合、排序等等。 里也有一个 逐 数。和 `filter_cache` 不同的是， 里的 逐 数是很有用的： 个数 或者至少是接近于 0。因 fielddata 不是 存，任何 逐都消耗巨大， 避免掉。如果 在 里看到 逐数， 需要重新 估 的内存情况，fielddata 限制， 求 句，或者 三者。

- <code>segments</code> 会展示 个 点目前正在服 中的 Lucene 段的数量。 是一个重要的数字。大多数索引会有大概 50–150 个段， 怕它 存有 TB 的数十条文 。段数量 大表明合并出 了 （比如，合并速度跟不上段的 建）。注意 个 是 点上所有索引的 聚 数。 住 点。

  `memory` 展示了 Lucene 段自己用掉的内存大小。 里包括底 数据 ，比如倒排表，字典，和布隆 器等。太大的段数量会 加 些数据 来的 ， 个内存使用量就是一个方便用来衡量 的度量 。

操作系　和　程部分

`OS` 和 `Process` 部分基本是自描述的，不会在　　中展　　解。它　列出来基　的　源　　，比如 CPU 和　　。`OS` 部分描述了整个操作系　，而 `Process` 部分只　示 Elasticsearch 的 JVM　　程使用的　源情况。

　些都是非常有用的指　，不　通常在　的　控技　　里已　都　量好了。　　包括下面　些：

- CPU
-
- 内存使用率
- Swap 使用率
- 打　的文件描述符

JVM 部分

`jvm` 部分包括了　行 Elasticsearch 的 JVM　　程一些很　　的信息。最重要的，它包括了　　回收的　，　　的 Elasticsearch 集群的　定性有着重大影　。

回收入

在我 描述 之前，先上一 速成 程 解 回收以及它 Elasticsearch 的影 是非常有用的。如果 JVM 的 回收很熟悉， 跳 段。

Java 是一 回收 言，也就是 程序 不用手 管理内存分配和回收。程序 只管写代 ，然后 Java 虚 机（JVM）按需分配内存，然后在 后不再需要的 候清理 部分内存。

当内存分配 一个 JVM 程，它是分配到一个大 里， 个 叫做 堆 。JVM 把堆分成 ，用 代来表示：

新生代（或者伊 ）
: 新 例化的 象分配的空 。新生代空 通常都非常小，一般在 100 MB–500 MB。新生代也包含 个 幸存 空 。

老生代
: 老的 象存 的空 。 些 象 将 期留存并持 上很 一段 。老生代通常比新生代大很多。Elasticsearch 点可以 老生代用到 30 GB。

当一个 象 例化的 候，它被放在新生代里。当新生代空 了，就会 生一次新生代 回收（GC）。依然是"存活"状 的 象就被 移到一个幸存区内，而"死掉"的 象被移除。如果一个 象在多次新生代 GC 中都幸存了，它就会被" 身"置于老生代了。

似的 程在老生代里同 生：空 的 候， 生一次 回收，死掉的 象被移除。

不 ，天下没有免 的午餐。新生代、老生代的 回收都有一个 段会“停止 ”。在 段 里 ，JVM 字面意 上的停止了程序 行，以便跟踪 象 ，收集死亡 象。在 个 停止 段，一切都不会 生。 求不被服 ，ping 不被回 ，分片不被分配。整个世界都真的停止了。

于新生代， 不是什 大 ；那 小的空 意味着 GC 会很快 行完。但是老生代大很多，而里面一个慢 GC 可能就意味着 1 秒乃至 15 秒的 停—— 于服 器 件来 是不可接受的。

JVM 的 回收采用了 非常 精密的算法，在 少 停方面做得很棒。而且 Elasticsearch 非常努力的 成 回收友好 的程序，比如内部智能的重用 象，重用 冲，以及 用 [docvalues] 功能。但最 ，GC 的 率和 依然是 需要去 察的指 。因 它是集群不 定的 号嫌疑人。

一个 常 生 GC 的集群就会因 内存不足而 于高 力下。 些 GC 会 致 点短 内从集群里掉 。 不 定会 致分片 繁重定位，因 Elasticsearch 会 保持集群均衡，保 有足 的副本在 。 接着就 致 流量和磁 I/O 的 加。而所有 些都是在 的集群努力服 于正常的索引和 的同 生的。

而言之， GC 是不好的，需要尽可能的 少。

因 回收 Elasticsearch 是如此重要， 非常熟悉 `node-stats` API 里的 部分内容：

```
      "jvm": {
         "timestamp": 1408556438203,
         "uptime_in_millis": 14457,
         "mem": {
            "heap_used_in_bytes": 457252160,
            "heap_used_percent": 44,
            "heap_committed_in_bytes": 1038876672,
            "heap_max_in_bytes": 1038876672,
            "non_heap_used_in_bytes": 38680680,
            "non_heap_committed_in_bytes": 38993920,
```

- jvm 部分首先列出一些和 heap 内存使用有 的常 。 可以看到有多少 heap 被使用了，多少被指派了（当前被分配 程的），以及 heap 被允 分配的最大 。理想情况下，heap_committed_in_bytes 等于 heap_max_in_bytes 。如果指派的大小更小，JVM 最 会被迫 整 heap 大小—— 是一个非常昂 的操作。如果 的数字不相等， 堆内存: 大小和交 学 如何正 的配置它。

  heap_used_percent 指 是 得 注的一个数字。Elasticsearch 被配置 当 heap 到 75% 的 候 始 GC。如果 的 点一直 >= 75%， 的 点正 于 内存 力 状 。 是个危 信号，不 的未来可能就有慢 GC 要出 了。

  如果 heap 使用率一直 >=85%， 就麻 了。Heap 在 90–95% 之 ， 面 可怕的性能 ，此 最好的情况是 10–30s 的 GC，最差的情况就是内存溢出（OOM） 常。

```
      "pools": {
         "young": {
            "used_in_bytes": 138467752,
            "max_in_bytes": 279183360,
            "peak_used_in_bytes": 279183360,
            "peak_max_in_bytes": 279183360
         },
         "survivor": {
            "used_in_bytes": 34865152,
            "max_in_bytes": 34865152,
            "peak_used_in_bytes": 34865152,
            "peak_max_in_bytes": 34865152
         },
         "old": {
            "used_in_bytes": 283919256,
            "max_in_bytes": 724828160,
            "peak_used_in_bytes": 283919256,
            "peak_max_in_bytes": 724828160
         }
      }
   },
```

- 新生代(young) 、 幸存区(survivor) 和 老生代(old) 部分分 展示 GC 中 一个代的内存使用情况。

些 很方便 察其相 大小，但是在 的 候，通常并不 重要。

```
"gc": {
   "collectors": {
      "young": {
         "collection_count": 13,
         "collection_time_in_millis": 923
      },
      "old": {
         "collection_count": 0,
         "collection_time_in_millis": 0
      }
   }
}
```

- gc 部分 示新生代和老生代的 回收次数和累 。大多数 候 可以忽略掉新生代的次数：个数字通常都很大。 是正常的。

  与之相反，老生代的次数 很小，而且 collection_time_in_millis 也 很小。 些是累 ，所以很 出一个 表示 要 始操心了（比如，一个 了一整年的 点，即使很健康，也会有一个比 大的 数）。 就是像 Marvel 工具很有用的一个原因。GC 数的 是个重要的考 因素。

  GC 花 的 也很重要。比如，在索引文 ，一系列 生成了。 是很常 的情况，刻都会 致 GC。 些 GC 大多数 候都很快， 点影 很小：新生代一般就花一 秒，老生代花一百多 秒。 些跟 10 秒 的 GC 是很不一 的。

  我 的最佳建 是定期收集 GC 数和 （或者使用 Marvel）然后 察 GC 率。 也可以 慢 GC 日志 ，在 日志 小 已 。

程池部分

Elasticsearch 在内部 了 程池。 些 程池相互 作完成任 ，有必要的 相互 会 任 。通常来 ， 不需要配置或者 程池，不 看它 的 有 候 是有用的，可以洞察 的集群表 如何。

有一系列的 程池，但以相同的格式 出：

```
  "index": {
     "threads": 1,
     "queue": 0,
     "active": 0,
     "rejected": 0,
     "largest": 1,
     "completed": 1
  }
```

个 程池会列出已配置的 程数量（ threads ），当前在 理任 的 程数量（ active ），以及在

列中等待 理的任 元数量（ queue ）。

如果 列中任 元数 到了 限，新的任 元会 始被拒 ， 会在 rejected 上看到它反映出来。 通常是 的集群在某些 源上 到瓶 的信号。因 列 意味着 的 点或集群在用最高速度 行，但依然跟不上工作的蜂 而入。

批量操作的被拒 数

如果 到了 列被拒，一般来 都是批量索引 求 致的。通 并 入程序 送大量批量 求非常 。越多越好 ， 不？

事 上， 个集群都有它能 理的 求上限。一旦 个 被超 ， 列会很快塞 ，然后新的批量 求就被拒 了。

是一件 好事情 。 列的拒 在回 方面是有用的。它 知道 的集群已 在最大容量了。比把数据塞 内存 列要来得好。 加 列大小并不能 加性能，它只是 藏了 。当 的集群只能 秒 理 10000 个文 的 候，无 列是 100 是 10000000 都没 系—— 的集群是只能 秒 理 10000 个文 。

列只是 藏了性能 ，而且 来的是真 的数据 失的 。在 列里的数据都是 没 理的，如果 点挂掉， 些 求都会永久的 失。此外， 列 要消耗大量内存， 也是不理想的。

在 的 用中， 雅的 理来自 列的回 ，才是更好的 。当 收到拒 的 候，采取如下几 ：

1. 停 入 程 3–5 秒。
2. 从批量操作的 里提取出来被拒 的操作。因 可能很多操作 是成功的。 会告 些成功， 些被拒 了。
3. 送一个新的批量 求，只包含 些被拒 的操作。
4. 如果依然 到拒 ，再次从 1 始。

通 个流程， 的代 可以很自然的 集群的 ，做到自 回 。

拒 不是 ：它 只是意味着 要 后重 。

里的一系列的 程池，大多数 可以忽略，但是有一小部分 是 得 注的：

`indexing`
普通的索引 求的 程池

`bulk`
批量 求，和 条的索引 求不同的 程池

`get`
Get-by-ID 操作

`search`
所有的搜索和 求

`merging`
    用于管理 Lucene 合并的 程池

文件系 和   部分

  向下   `node-stats`   API， 会看到一串和 的文件系 相 的   ：可用空 ，数据目路径，磁 I/O   ，等等。如果 没有 控磁 可用空 的 ，可以从 里 取 些   。磁 I/O   也很方便，不 通常那些更   的命令行工具（比如 `iostat` ）会更有用些。

然，Elasticsearch 在磁 空   的 候很   行——所以   保不会   。

有 个跟   相 的部分：

```
        "transport": {
            "server_open": 13,
            "rx_count": 11696,
            "rx_size_in_bytes": 1525774,
            "tx_count": 10282,
            "tx_size_in_bytes": 1440101928
         },
         "http": {
            "current_open": 4,
            "total_opened": 23
         },
```

- `transport`   示和   地址   相 的一些基   。包括 点 的通信（通常是   9300 端口）以及任意   客 端或者 点客 端的 接。如果看到 里有很多 接数不要担心；Elasticsearch 在 点之   了大量的 接。
- `http` 示 HTTP 端口（通常是 9200）的   。如果 看到 `total_opened` 数很大而且 在一直上， 是一个明 信号， 明 的 HTTP 客 端里有没 用 keep-alive   接的。持 的 keep-alive 接 性能很重要，因   接、断 套接字是很昂 的（而且浪 文件描述符）。   的客 端都配置正 。

断路器

于，我 到了最后一段：跟 fielddata 断路器（在 [circuit-breaker] 介   ）相 的   ：

```
         "fielddata_breaker": {
            "maximum_size_in_bytes": 623326003,
            "maximum_size": "594.4mb",
            "estimated_size_in_bytes": 0,
            "estimated_size": "0b",
            "overhead": 1.03,
            "tripped": 0
         }
```

  里 可以看到断路器的最大 （比如，一个 求申 更多的内存 会触 断路器）。 个部分 会   知道断路器被触 了多少次，以及当前配置的 接   。 接   用来   估，因 有些 求比其他 求更

估。

主要需要 注的是 tripped 指 。如果 个数字很大或者持 上 ， 是一个信号， 明 的 求需要 化，或者 需要添加更多内存（ 机上添加，或者通 添加新 点的方式）。

集群

集群 API 提供了和 点 相似的 出。 但有一个重要的区 ： 点 示的是 个点上的 ，而 集群 展示的是 于 个指 ，所有 点的 和 。

里面提供一些很 得一看的 。比如 可以看到，整个集群用了 50% 的堆内存，或者 器存的 逐情况不 重。 个接口主要用途是提供一个比 集群健康 更 、但又没有 点 那 的快速概 。 于非常大的集群来 也很有用，因 那 候 点 的 出已 非常 于 了。

个 API 可以像下面 用：

```
GET _cluster/stats
```

索引

到目前 止，我 看到的都是以 点 中心 的 ： 点有多少内存？用了多少 CPU ？正在服 多少个搜索？

有 候从 索引 中心 的角度看 也很有用： 个索引 收到了多少个搜索 求？那个索引 取文 耗了多少 ？

要做到 点， 感 趣的索引（或者多个索引）然后 行一个索引 的 API：

```
GET my_index/_stats ①

GET my_index,another_index/_stats ②

GET _all/_stats ③
```

① my_index 索引。

② 使用逗号分隔索引名可以 求多个索引 。

③ 使用特定的 _all 可以 求全部索引的

返回的 信息和 点 的 出很相似：search 、 fetch 、 get 、 index 、 bulk 、 segment counts 等等。

索引 中心的 在有些 候很有用，比如 或 集群中的 索引，或者 出某些索引比其他索引更快或者更慢的原因。

践中， 点 中心的 是 得更有用些。瓶 往往是 整个 点而言，而不是 于 个索引。因 索引一般是分布在多个 点上的， 致以索引 中心的 通常不是很有用，因 它 是从不同 境的物理机器上 聚的数据。

索引　中心的　　作　一个有用的工具可以保留在　的技能表里，但是通常它不会是第一个用的上的工具。

等待中的任

有一些任　只能由主　点去　理，比如　建一个新的索引或者在集群中移　分片。由于一个集群中只能有一个主　点，所以只有　一　点可以　理集群　　的元数据　　。在　　99.9999%　　的　　里，　不会有什　　。元数据　　的　列基本上保持　零。

在一些　　的集群里，元数据　　的次数比主　点能　理的　快。　会　致等待中的操作会累　成　列。

等待中的任　API 会　　展示　列中（如果有的　）等待的集群　　的元数据　更操作：

```
GET _cluster/pending_tasks
```

通常，　　都是像　　的：

```
{
   "tasks": []
}
```

意味着没有等待中的任　。如果　有一个　　的集群在主　点出　瓶　了，等待中的任　列表可能会像　：

```
{
   "tasks": [
      {
         "insert_order": 101,
         "priority": "URGENT",
         "source": "create-index [foo_9], cause [api]",
         "time_in_queue_millis": 86,
         "time_in_queue": "86ms"
      },
      {
         "insert_order": 46,
         "priority": "HIGH",
         "source": "shard-started ([foo_2][1], node[tMTocMvQQgGCkj7QDHl3OA], [P],
         s[INITIALIZING]), reason [after recovery from gateway]",
         "time_in_queue_millis": 842,
         "time_in_queue": "842ms"
      },
      {
         "insert_order": 45,
         "priority": "HIGH",
         "source": "shard-started ([foo_2][0], node[tMTocMvQQgGCkj7QDHl3OA], [P],
         s[INITIALIZING]), reason [after recovery from gateway]",
         "time_in_queue_millis": 858,
         "time_in_queue": "858ms"
      }
  ]
}
```

可以看到任 都被指派了 先 （ 比如 URGENT 要比 HIGH 更早的 理 ），任 入的次序、操作 入 列多久，以及打算 理什 。在上面的列表中，有一个 建索引(create-index) 和 个 分片(shard-started) 的操作在等待。

什 候 担心等待中的任 ？

就像曾 提到 的，主 点很少会成 集群的瓶 。唯一可能成 瓶 的是集群状 非常大 而且更新 繁。

例如，如果 允 客 按照他 的意 建任意的 字段，而且 个客 天都有一个独立索引，那 的集群状 会 得非常大。集群状 包括 （ 但不限于 ） 所有索引及其 型，以及 个索引的全部字段。

所以如果 有 100000 客 ，然后 个客 平均有 1000 个字段，而且数据有 90 天的保留期 — 就有九十 个字段需要保存在集群状 中。不管它何 生 更，所有的 点都需要被通知。

主 点必 理 些 ， 需要不小的 CPU ，加上推送更新的集群状 到所有 点的 。

就是那些可以看到集群状 操作 列上 的集群。没有 的 法可以解决 个 ，不 有三个 ：

- 使用一个更 大的主 点。不幸的是， 垂直 展只是延 必然 果出 而已。
- 通 某些方式限定文 的 性 来限制集群状 的大小。
- 到 某个 后 建 外一个集群。

cat API

如果 常在命令行 境下工作，`cat` API 会非常有用。用 Linux 的 `cat` 命令命名， 些 API 也就 成像 *nix 命令行工具一 工作了。

他 提供的 和前面已 的 API （ 健康、 点 等等 ） 是一 的。但是 出以表格的形式提供，而不是 JSON。 于系 管理 来 是 非常 方便的， 想一遍集群或者 出内存使用偏高的 点而已。

通 `GET` 求 送 `cat` 命名可以列出所有可用的 API：

```
GET /_cat

=^.^=
/_cat/allocation
/_cat/shards
/_cat/shards/{index}
/_cat/master
/_cat/nodes
/_cat/indices
/_cat/indices/{index}
/_cat/segments
/_cat/segments/{index}
/_cat/count
/_cat/count/{index}
/_cat/recovery
/_cat/recovery/{index}
/_cat/health
/_cat/pending_tasks
/_cat/aliases
/_cat/aliases/{alias}
/_cat/thread_pool
/_cat/plugins
/_cat/fielddata
/_cat/fielddata/{fields}
```

多 API 看起来很熟悉了 (是的，　上　有一只猫:))。　我　看看 cat 的健康　　API：

```
GET /_cat/health

1408723713 12:08:33 elasticsearch_zach yellow 1 1 114 114 0 0 114
```

首先　会注意到的是　　是表格　式的　文本，而不是　　JSON。其次　会注意到各列　　是没有表　的。　都是模　*nix 工具　　的，因　它假　一旦　　出熟悉了，　就再也不想看　表　了。

要　用表　，添加 ?v 参数即可：

```
GET /_cat/health?v

epoch   time    cluster status node.total node.data shards pri relo init
1408[..] 12[..] el[..]  1         1         114 114    0    0     114
unassign
```

，好多了。我　　在看到　　、集群名称、状　、集群中　点的数量等等—所有信息和 集群健康 API 返回的都一　。

我　再看看 cat API 里面的　点　　：

```
GET /_cat/nodes?v

host         ip            heap.percent ram.percent load node.role master name
zacharys-air 192.168.1.131           45          72 1.85 d         *      Zach
```

我 看到集群里 点的一些 ，不 和完整的 点 出相比而言是非常基 的。 可以包含更多的指 ，但是比起 文 ， 我 直接 cat API 有 些可用的 。

可以 任意 API 添加 ?help 参数来做到 点：

```
GET /_cat/nodes?help

id                | id,nodeId               | unique node id
pid               | p                       | process id
host              | h                       | host name
ip                | i                       | ip address
port              | po                      | bound transport port
version           | v                       | es version
build             | b                       | es build hash
jdk               | j                       | jdk version
disk.avail        | d,disk,diskAvail        | available disk space
heap.percent      | hp,heapPercent          | used heap ratio
heap.max          | hm,heapMax              | max configured heap
ram.percent       | rp,ramPercent           | used machine memory ratio
ram.max           | rm,ramMax               | total machine memory
load              | l                       | most recent load avg
uptime            | u                       | node uptime
node.role         | r,role,dc,nodeRole      | d:data node, c:client node
master            | m                       | m:master-eligible, *:current master
...
...
```

(注意 个 出 了 面 而被截断了)。

第一列 示完整的名称，第二列 示 写，第三列提供了 于 个参数的 介。 在我 知道了一些列名了，我 可以用 ?h 参数来明 指定 示 些指 ：

```
GET /_cat/nodes?v&h=ip,port,heapPercent,heapMax

ip            port heapPercent heapMax
192.168.1.131 9300          53 990.7mb
```

因 cat API 像 *nix 工具一 工作， 可以使用管道命令将 果 其他工具，比如 sort 、 grep 或者 awk 。例如，通 以下方式可以 到集群中最大的索引：

```
% curl 'localhost:9200/_cat/indices?bytes=b' | sort -rnk8

yellow test_names         5 1 3476004 0 376324705 376324705
yellow .marvel-2014.08.19 1 1  263878 0 160777194 160777194
yellow .marvel-2014.08.15 1 1  234482 0 143020770 143020770
yellow .marvel-2014.08.09 1 1  222532 0 138177271 138177271
yellow .marvel-2014.08.18 1 1  225921 0 138116185 138116185
yellow .marvel-2014.07.26 1 1  173423 0 132031505 132031505
yellow .marvel-2014.08.21 1 1  219857 0 128414798 128414798
yellow .marvel-2014.07.27 1 1   75202 0  56320862  56320862
yellow wavelet            5 1    5979 0  54815185  54815185
yellow .marvel-2014.07.28 1 1   57483 0  43006141  43006141
yellow .marvel-2014.07.21 1 1   31134 0  27558507  27558507
yellow .marvel-2014.08.01 1 1   41100 0  27000476  27000476
yellow kibana-int         5 1       2 0     17791     17791
yellow t                  5 1       7 0     15280     15280
yellow website            5 1      12 0     12631     12631
yellow agg_analysis       5 1       5 0      5804      5804
yellow v2                 5 1       2 0      5410      5410
yellow v1                 5 1       2 0      5367      5367
yellow bank               1 1      16 0      4303      4303
yellow v                  5 1       1 0      2954      2954
yellow p                  5 1       2 0      2939      2939
yellow b0001_072320141238 5 1       1 0      2923      2923
yellow ipaddr             5 1       1 0      2917      2917
yellow v2a                5 1       1 0      2895      2895
yellow movies             5 1       1 0      2738      2738
yellow cars               5 1       0 0      1249      1249
yellow wavelet2           5 1       0 0       615       615
```

通 添加 `?bytes=b` ，我 了人 可 的数字格式化， 制它 以字 数 出。随后通 管道命令将 出 `sort` 索引按大小（第八列）排序

不幸的是， 会注意到 Marval 索引也出 在 果中，但是我 目前并不真正在意 些索引。 我 把果 `grep` 命令来移除提到 Marval 的数据：

```
% curl 'localhost:9200/_cat/indices?bytes=b' | sort -rnk8 | grep -v marvel

yellow test_names         5 1 3476004 0 376324705 376324705
yellow wavelet            5 1    5979 0  54815185  54815185
yellow kibana-int         5 1       2 0     17791     17791
yellow t                  5 1       7 0     15280     15280
yellow website            5 1      12 0     12631     12631
yellow agg_analysis       5 1       5 0      5804      5804
yellow v2                 5 1       2 0      5410      5410
yellow v1                 5 1       2 0      5367      5367
yellow bank               1 1      16 0      4303      4303
yellow v                  5 1       1 0      2954      2954
yellow p                  5 1       2 0      2939      2939
yellow b0001_072320141238 5 1       1 0      2923      2923
yellow ipaddr             5 1       1 0      2917      2917
yellow v2a                5 1       1 0      2895      2895
yellow movies             5 1       1 0      2738      2738
yellow cars               5 1       0 0      1249      1249
yellow wavelet2           5 1       0 0       615       615
```

！在 grep （ 通 -v 来 掉不需要匹配的数据 ） 之后，我 得到了一个没有 Marval 混的索引排序列表了。

只是命令行上 cat 的 活性的一个 示例。一旦 了使用 cat ， 会 它和其他所有 *nix 工具一 并且 始 狂的使用管道、排序和 。如果 是一个系 管理 并且永 都是 SSH 登 到 上，那 当然要花些 来熟悉 cat API 了。

部署

如果 按照 中 做到了 一 ，希望 已 学到了一 件 于 Elasticsearch 的事情并且准 把 的集群部署到生 境。 一章不是在生 中 行集群的 尽指南，但是它涵 了集群上 之前需要考的 事 。

主要包括三个方面：

- 后勤方面的考 ，如硬件和部署策略的建
- 更 合于生 境的配置更改
- 部署后的考 ，例如安全，最大限度的索引性能和

硬件

按照正常的流程， 可能已 在自己的 本 或集群上使用了 Elasticsearch。 但是当要部署 Elasticsearch 到生 境 ，有一些建 是 需要考 的。 里没有什 必 要遵守的准 ，Elasticsearch 被用于在 多的机器上 理各 任 。基于我 在生 境使用 Elasticsearch 集群的 ， 些建 可以 提供一个好的起点。

内存

如果有一　　源是最先被耗尽的，它可能是内存。排序和聚合都很耗内存，所以有足　的堆空　来　付它　是很重要的。即使堆空　是比　小的　候，　　也能　操作系　文件　存提供　外的内存。因　　Lucene 使用的　多数据　　是基于磁　的格式，Elasticsearch 利用操作系　　存能　生很大效果。

64 GB 内存的机器是非常理想的，但是32 GB 和16 GB 机器也是很常　的。少于8 GB 会　得其反（　最　需要很多很多的小机器），大于64 GB 的机器也会有　　，我　将在 堆内存:大小和交　中　　。

CPUs

大多数　Elasticsearch　部署往往　　CPU　要求不高。因此，相　其它　源，具体配置多少个（CPU）不是那　　。　　具有多个内核的　代　理器，常　的集群使用　到八个核的机器。

如果　要在更快的　　CPUs　　和更多的核心之　　，　　更多的核心更好。多个内核提供的　外并　　微快一点点的　　率。

硬

硬　　所有的集群都很重要，　大量写入的集群更是加倍重要（例如那些存　日志数据的）。硬　是服　器上最慢的子系　，　意味着那些写入量很大的集群很容易　硬　　和，使得它成　集群的瓶　。

如果　　担得起　SSD，它将　　超出任何旋　介　（注：机械硬　，磁　等）。　基于　SSD　的　点，　　和索引性能都有提升。如果　　担得起，SSD 是一个好的　　。

的 I/O　度程序

如果　正在使用　SSDs，　保　的系　　I/O　　度程序是配置正　的。　当　向硬　写数据，I/O　度程序决定何　把数据　　送到硬　。　　大多数　　*nix　　行版下的　度程序都叫做 `cfq`（完全公平　列）。

　度程序分配　　片　　到　个　程。并且　化　些到硬　的　多　列的　　。但它是　旋　介　化的：机械硬　的固有特性意味着它写入数据到基于物理布局的硬　会更高效。

　　SSD　来　是低效的，尽管　里没有　及到机械硬　。但是，`deadline`　或者　`noop` 被使用。`deadline`　度程序基于写入等待　　行　化，`noop` 只是一个　　的 FIFO　列。

　个　　的更改可以　来　著的影　。　　是使用正　的　度程序，我　看到了500倍的写入能力提升。

如果　使用旋　介　，　　取尽可能快的硬　（高性能服　器硬　，15k RPM　　器）。

使用 RAID 0 是提高硬　速度的有效途径，　机械硬　和 SSD 来　都是如此。没有必要使用　像或其它 RAID　体，因　高可用已　通　replicas 内建于 Elasticsearch 之中。

最后，避免使用　　附加存　（NAS）。人　常声称他　的　　NAS　　解决方案比本地　　器更快更可　。除却　些声称，　我　从没看到　NAS　能配得上它的大肆宣　。NAS　常常很慢，　露出更大的延　和更　的平均延　方差，而且它是　点故障的。

快速可 的 然 分布式系 的性能是很重要的。 低延 能 助 保 点 能容易的通 ，大 能 助分片移 和恢 。 代数据中心 （1 GbE, 10 GbE） 大多数集群都是足 的。

即使数据中心 近在咫尺，也要避免集群跨越多个数据中心。 要避免集群跨越大的地理距 。

Elasticsearch 假定所有 点都是平等的—并不会因 有一半的 点在150ms 外的 一数据中心而有所不同。更大的延 会加重分布式系 中的 而且使得 和排 更困 。

和 NAS 的争 似， 个人都声称他 的数据中心 的 路都是健壮和低延 的。 是真的—直到它不是 （ 失 究是会 生的， 可以相信它）。 从我 的 来看，理跨数据中心集群的麻 事是根本不 得的。

取真正的高配机器在今天是可能的：成百 GB 的 RAM 和几十个 CPU 核心。 反之，在云平台上串起成千的小虚 机也是可能的，例如 EC2。 方式是最好的?

通常， 中配或者高配机器更好。避免使用低配机器， 因 不会希望去管理 有上千个 点的集群，而且在 些低配机器上 行 Elasticsearch 的 也是 著的。

与此同 ，避免使用真正的高配机器。它 通常会 致 源使用不均衡（例如，所有的内存都被使用，但 CPU 却没有）而且在 机上 行多个 点 ，会 加 度。

Java 虚 机

始 行最新版本的 Java 虚 机（ JVM ）， 除非 Elasticsearch 站上 有 明。Elasticsearch， 特 是 Lucene，是一个高要求的 件。Lucene 的 元 和集成 常暴露出 JVM 本身的 bug。 些 bug 的 从 微的麻 到 重段 ，所以，最好尽可能的使用最新版本的 JVM。

Java 8 烈 先 于 Java 7。不再支持 Java 6。Oracle 或者 OpenJDK 是可以接受的，它 在性能和 定性也差不多。

如果 的 用程序是用 Java 写并正在使用 客 端（注：Transport Client，下同）或 点客 端（注：Node Client，下同）， 保 行 用程序的 JVM 和服 器的 JVM 是完全一 的。 在 Elasticsearch 的几个地方，使用 Java 的本地序列化（ IP 地址、 常等等）。不幸的是，Oracle 的 JVM 在几个小版本之 有修改序列化格式，从而 致奇怪的 。 情况很少 ，但最佳 践是客 端和服 器使用相同版本 JVM。

不要 整 JVM 置

JVM 暴露出几十个（甚至数百）的 置、参数和配置。 它 允 行微 JVM 几乎是 一个方面。 当遇到一个旋 ，要打 它是人的本性。我 求 制 个本性，而 不要 去 整 JVM 参数。Elasticsearch 是 的 件，并且我 根据多年的 使用情况 整了当前 JVM 置。它很容易 始 旋 ，并 生 以衡量的、未知的影 ，并最 使集群 入一个 慢的、不 定的混乱的效果。当 集群 ，第一 往往是去除所有的自定 配置。多数情况下， 此就可以恢 定和性能。

Transport Client 与 Node Client

如果 使用的是 Java， 可能想知道何 使用 客 端（注：Transport Client，下同）与 点客 端（注：Node Client，下同）。 在 的 所述， 客 端作 一个集群和 用程序之 的通信 。它知道 API 并能自 在 点之 ， 嗅探集群等等。但它是集群 外部的 ，和 REST 客端 似。

一方面， 点客 端， 上是一个集群中的 点（但不保存数据，不能成 主 点）。因 它是一个 点，它知道整个集群状 （所有 点 留，分片分布在 些 点，等等）。 意味着它可以 行 APIs 但少了一个 点。

里有 个客 端案例的使用情况：

- 如果要将 用程序和 Elasticsearch 集群 行解 ， 客 端是一个理想的 。例如，如果 的 用程序需要快速的 建和 到集群的 接， 客 端比 点客 端” ”，因 它不是一个集群的一部分。

  似地，如果 需要 建成千上万的 接， 不想有成千上万 点加入集群。 客 端（ TC ）将是一个更好的 。

- 一方面，如果 只需要有少数的、 期持久的 象 接到集群，客 端 点可以更高效，因 它知道集群的布局。但是它会使 的 用程序和集群 合在一起，所以从防火 的角度，它可能会 成 。

配置管理

如果 已 使用配置管理（ Puppet，Chef，Ansible）， 可以跳 此提示。

如果 没有使用配置管理工具，那 注意了！通 `parallel-ssh` 管理少量服 器 在可能正常工作，但伴随着集群的 它将成 一 梦。 在不犯 的情况下手 30 个配置文件几乎是不可能的。

配置管理工具通 自 化更改配置的 程保持集群的一致性。 可能需要一点 来建立和学 ，但它本身 ，随着 的推移会有 厚的回 。

重要配置的修改

Elasticsearch 已 有了 很好 的 ，特 是 及到性能相 的配置或者 。 如果 有疑 ，最好就不要 它。我 已 目 了数十个因 的 置而 致 的集群， 因 它的管理者 改 一个配置或者 就可以 来 100 倍的提升。

NOTE	整 文章，所有的配置 都同等重要，和描述 序无 ， 所有的配置 ，并用到 的集群中。

其它数据 可能需要 ，但 得来 ，Elasticsearch 不需要。 如果 遇到了性能 ，解决方法通常是更好的数据布局或者更多的 点。 在 Elasticsearch 中很少有“神奇的配置 ”，如果存在，我 也已 化了！

外，有些 上的 配置在生 境中是 整的。 些 整可能会 的工作更加 松，又或者因 没 法 定一个 （它取决于 的集群布局）。

指定名字

Elasticsearch 的集群名字叫 elasticsearch 。 最好 的生 境的集群改个名字，改名字的目的很 ， 就是防止某人的 本 加入了集群 意外。 修改成 elasticsearch_production 会很省心。

可以在 的 elasticsearch.yml 文件中修改：

```
cluster.name: elasticsearch_production
```

同 ，最好也修改 的 点名字。就像 在可能 的那 ， Elasticsearch 会在 的 点 的 候随机 它指定一个名字。 可能会 得 很有趣，但是当凌晨 3 点 的 候， 在 回 台物理机是 Tagak the Leopard Lord 的 候， 就不 得有趣了。

更重要的是， 些名字是在 的 候 生的， 次 点， 它都会得到一个新的名字。 会使日志 得很混乱，因 所有 点的名称都是不断 化的。

可能会 得 ，我 建 个 点 置一个有意 的、清楚的、描述性的名字，同 可以在 elasticsearch.yml 中配置：

```
node.name: elasticsearch_005_data
```

路径

情况下，Elasticsearch 会把 件、日志以及 最重要的数据放在安装目 下。 会 来不幸的事故，如果 重新安装 Elasticsearch 的 候不小心把安装目 覆 了。如果 不小心， 就可能把的全部数据 掉了。

不要笑， 情况，我 很多次了。

最好的 就是把 的数据目 配置到安装目 以外的地方，同 也可以 移 的 件和日志目 。

可以更改如下：

```
path.data: /path/to/data1,/path/to/data2 ①

# Path to log files:
path.logs: /path/to/logs

# Path to where plugins are installed:
path.plugins: /path/to/plugins
```

① 注意： 可以通 逗号分隔指定多个目 。

数据可以保存到多个不同的目 ， 如果将 个目 分 挂 不同的硬 ， 可是一个 且高效 一个磁 列（ RAID 0 ）的 法。Elasticsearch 会自 把条 化（注：RAID 0 又称 Stripe（条化），在磁 列中,数据是以条 的方式 穿在磁 列所有硬 中的） 数据分隔到不同的目 ，以便提高性能。

WARNING

多个数据路径的安全性和性能

如同任何磁　　列（　RAID　0　）的配置，只有　一的数据拷　保存到硬器。如果　失去了一个硬　　器，　　肯定　　会失去　　算机上的一部分数据。　气好的　　的副本在集群的其他地方，可以用来恢　数据和最近的　　。

Elasticsearch　　将全部的条　化分片放到　个　　器来保　最小程度的数据失。　意味着 `分片　0` 将完全被放置在　个　　器上。 Elasticsearch 没有一个条　化的分片跨越在多个　　器，因　一个　　器的　失会破坏整个分片。

　　性能　生的影　是：如果　添加多个　　器来提高一个　独索引的性能，可能助不大，因　　大多数　点只有一个分片和　　一个　　的器。多个数据路径只是　助如果　有　多索引／分片在　个　点上。

多个数据路径是一个非常方便的功能，但到　来，Elasticsearch 并不是　磁　　列（software　RAID　）的　件。如果　需要更高　的、　健的、　活的配置，　我　建　　使用　磁　　列（software RAID）的　件，而不是多个数据路径的功能。

最小主　点数

`minimum_master_nodes`　定　　的集群的　定　　其　重要。　当　的集群中有　个　masters（注：主点）的　候，　个配置有助于防止　裂，一　　个主　点同　存在于一个集群的　象。

如果　的集群　生了　裂，那　　的集群就会　在　失数据的危　中，因　主　点被　　是　个集群的最高　治者，它决定了什　　候新的索引可以　建，分片是如何移　的等等。如果　有　　个　masters　　点，　的数据的完整性将得不到保　，因　　有　个　点　　他　有集群的控制　。

　个配置就是告　Elasticsearch 当没有足　master 候　　点的　候，就不要　行 master　点　　，等 master 候　　点足　了才　行　　。

此　置　　始　被配置　　master　候　　点的法定个数（大多数个）。法定个数就是　`（　master　候　点个数　/　2）+ 1`。　里有几个例子：

- 如果　有 10 个　点（能保存数据，同　能成　master），法定数就是 `6`。
- 如果　有 3 个候　master　点，和 100 个 data　点，法定数就是 `2`，　只要数数那些可以做 master 的　点数就可以了。
- 如果　有　个　点，　遇到　　了。法定数当然是　　`2`　　，但是　意味着如果有一个　点挂掉，　整个集群就不可用了。　　置成　`1`　可以保　集群的功能，但是就无法保　集群　裂了，像　的情况，　最好至少保　有 3 个　点。

　可以在　的 `elasticsearch.yml` 文件中　　配置：

```
discovery.zen.minimum_master_nodes: 2
```

但是由于　ELasticsearch　是　　的，　可以很容易的添加和　除　点，　但是　会改　　个法定个数。　不得不修改　一个索引　点的配置并且重　　的整个集群只是　了　配置生效，　将是非常痛苦的一件事情。

基于　个原因，　`minimum_master_nodes`　（　有一些其它配置）允　通　　API　　用的方式　　行配置。

当 的集群在 行的 候， 可以 修改配置：

```
PUT /_cluster/settings
{
    "persistent" : {
        "discovery.zen.minimum_master_nodes" : 2
    }
}
```

将成 一个永久的配置，并且无 配置 里配置的如何， 个将 先生效。当 添加和 除 master 点的 候， 需要更改 个配置。

集群恢 方面的配置

当 集群重 ，几个配置 影 的分片恢 的表 。首先，我 需要明白如果什 也没配置将会 生什 。

想象一下假 有 10 个 点， 个 点只保存一个分片， 个分片是一个主分片或者是一个副本分片，或者 有一个有 5 个主分片／1 个副本分片的索引。有 需要 整个集群做 （比如， 了安装一个新的 程序）， 当 重 的集群，恰巧出 了 5 个 点已 ， 有 5 个 没 的 景。

假 其它 5 个 点出 ，或者他 根本没有收到立即重 的命令。不管什 原因， 有 5 个 点在 上， 五个 点会相互通信， 出一个 master，从而形成一个集群。 他 注意到数据不再均 分布，因 有 5 个 点在集群中 失了，所以他 之 会立即 分片 制。

最后， 的其它 5 个 点打 加入了集群。 些 点会 它 的数据正在被 制到其他 点，所以他 除本地数据（因 数据要 是多余的，要 是 的）。 然后整个集群重新 行平衡，因 集群的大小已 从 5 成了 10。

在整个 程中， 的 点会消耗磁 和 ，来回移 数据，因 没有更好的 法。 于有 TB 数据的大集群， 无用的数据 需要 很 。如果等待所有的 点重 好了，整个集群再上 ，所有的本地的数据都不需要移 。

在我 知道 的所在了，我 可以修改一些 置来 解它。 首先我 要 ELasticsearch 一个 格的限制：

```
gateway.recover_after_nodes: 8
```

将阻止 Elasticsearch 在存在至少 8 个 点（数据 点或者 master 点）之前 行数据恢 。 个 的 定取决于个人喜好：整个集群提供服 之前 希望有多少个 点在 ？ 情况下，我 置 8，意味着至少要有 8 个 点， 集群才可用。

在我 要告 Elasticsearch 集群中 有多少个 点，以及我 意 些 点等待多 ：

```
gateway.expected_nodes: 10
gateway.recover_after_time: 5m
```

意味着 Elasticsearch 会采取如下操作：

- 等待集群至少存在 8 个 点
- 等待 5 分 ，或者10 个 点上 后，才 行数据恢 ， 取决于 个条件先 到。

三个 置可以在集群重 的 候避免 多的分片交 。 可能会 数据恢 从数个小 短 几秒 。

注意： 些配置只能 置在 `config/elasticsearch.yml` 文件中或者是在命令行里（它 不能更新）它 只在整个集群重 的 候有 性作用。

最好使用 播代替 播

Elasticsearch 被配置 使用 播 ，以防止 点无意中加入集群。只有在同一台机器上 行的 点才会自 成集群。

然 播 然 作 件提供， 但它 永 不被使用在生 境了，否在 得到的 果就是一个 点意外的加入到了 的生 境， 是因 他 收到了一个 的 播信号。 于 播 本身 并没有 ， 播会 致一些愚蠢的 ，并且 致集群 的脆弱（比如，一个 工程 正在 鼓 ，而没有告 ， 会 所有的 点突然 不了 方了）。

使用 播， 可以 Elasticsearch 提供一些它 去 接的 点列表。 当一个 点 系到播列表中的成 ，它就会得到整个集群所有 点的状 ，然后它会 系 master 点，并加入集群。

意味着 的 播列表不需要包含 的集群中的所有 点， 它只是需要足 的 点，当一个新 点系上其中一个并且 上 就可以了。如果 使用 master 候 点作 播列表，只要列出三个就可以了。 个配置在 `elasticsearch.yml` 文件中：

```
discovery.zen.ping.unicast.hosts: ["host1", "host2:port"]
```

于 Elasticsearch 点 的 信息， 参 Zen Discovery Elasticsearch 文献。

不要触 些配置！

在 Elasticsearch 中有一些 点，人 可能不可避免的会 到。 我 理解的，所有的 整就是 了 化，但是 些 整， 真的不需要理会它。因 它 常会被乱用，从而造成系 的不 定或者糟 的性能，甚至 者都有可能。

回收器

里已 要介 了 回收入 ，JVM 使用一个 回收器来 放不再使用的内存。 篇内容的是上一篇的一个延 ，但是因 重要，所以 得 独拿出来作 一 。

不要更改 的 回收器！

Elasticsearch 的 回收器（ GC ）是 CMS。 个 回收器可以和 用并行 理，以便它可以最小化停 。然而，它有 个 stop-the-world 段， 理大内存也有点吃力。

尽管有 些 点，它 是目前 于像 Elasticsearch 低延 需求 件的最佳 回收器。官方建 使用

CMS。

在有一款新的 回收器，叫 G1 回收器（ G1GC ）。 款新的 GC 被 ，旨在比 CMS 更小的停 ，以及 大内存的 理能力。 它的原理是把内存分成 多区域，并且 些区域最有可能需要回收内存。通 先收集 些区域（ *garbage first* ）， 生更小的 停 ，从而能 更大的内存。

听起来很棒！ 憾的是，G1GC 是太新了， 常 新的 bugs。 些 通常是段（ segfault ） 型，便造成硬 的崩 。 Lucene 的 套件 回收算法要求 格，看起来 些 陷 G1GC 并没有很好地解决。

我 很希望在将来某一天推 使用 G1GC，但是 于 在，它 不能足 定的 足 Elasticsearch 和 Lucene 的要求。

程池

多人 喜 整 程池。 无 什 原因，人 都 加 程数无法抵抗。索引太多了？ 加 程！搜索太多了？ 加 程！ 点空 率低于 95%？ 加 程！

Elasticsearch 的 程 置已 是很合理的了。 于所有的 程池（除了 搜索 ）， 程个数是根据 CPU 核心数 置的。 如果 有 8 个核， 可以同 行的只有 8 个 程，只分配 8 个 程 任何特定的 程池是有道理的。

搜索 程池 置的大一点，配置 `int（（ 核心数 * 3 ）/ 2 ）+ 1`。

可能会 某些 程可能会阻塞（如磁 上的 I/O 操作），所以 才想加大 程的。 于 Elasticsearch 来 并不是一个 ：因 大多数 I/O 的操作是由 Lucene 程管理的，而不是 Elasticsearch。

此外， 程池通 彼此之 的工作配合。 不必再因 它正在等待磁 写操作而担心 程阻塞，因 程早已把 个工作交 外的 程池，并且 行了 。

最后， 的 理器的 算能力是有限的， 有更多的 程会 致 的 理器 繁切 程上下文。 一个 理器同 只能 行一个 程。所以当它需要切 到其它不同的 程的 候，它会存 当前的状 （寄存器等等），然后加 外一个 程。 如果幸 的 ， 个切 生在同一个核心，如果不幸的 ， 个切 可能 生在不同的核心， 就需要在内核 上 行 。

个上下文的切 ，会 CPU 周期 来管理 度的 ；在 代的 CPUs 上， 估 高 30 µs。也就是 程会被堵塞超 30 µs，如果 个 用于 程的 行， 有可能早就 束了。

人 常稀里糊 的 置 程池的 。8 个核的 CPU，我 遇到 有人配了 60、100 甚至 1000 个 程。 些 置只会 CPU 工作效率更低。

所以，下次 不要 整 程池的 程数。如果 真 想 整 ， 一定要 注 的 CPU 核心数，最多 置成核心数的 倍，再多了都是浪 。

堆内存:大小和交

Elasticsearch 安装后 置的堆内存是 1 GB。 于任何一个 部署来 ， 个 置都太小了。如果 正在使用 些 堆内存配置， 的集群可能会出 。

里有 方式修改 Elasticsearch 的堆内存。最 的一个方法就是指定 `ES_HEAP_SIZE` 境 量。服

程在　　候会　取　个　量，并相　的　置堆的大小。比如，　可以用下面的命令　置它：

```
export ES_HEAP_SIZE=10g
```

此外，　也可以通　命令行参数的形式，在程序　　的　候把内存大小　　它，如果　　得　　更　　的　　：

```
./bin/elasticsearch -Xmx10g -Xms10g ①
```

① 保堆内存最小　（ Xms ）与最大　（ Xmx ）的大小是相同的，防止程序在　行　改　堆内存大小，是一个很耗系　　源的　程。

通常来　，　置 ES_HEAP_SIZE　境　量，比直接写 -Xmx -Xms 更好一点。

把　的内存的（少于）一半　Lucene

一个常　的　　是　Elasticsearch 分配的内存 太 大了。假　　有一个 64 GB 内存的机器，　天　，我要把 64 GB 内存全都　Elasticsearch。因　越多越好　！

当然，内存　于　　Elasticsearch　　来　　是重要的，它可以被　多内存数据　　使用来提供更快的操作。但是　到　里，　　有　外一个内存消耗大　　非堆内存　（off-heap）：Lucene。

Lucene　　被　　可以利用操作系　底　机制来　存内存数据　　。　Lucene　　的段是分　存　到　个文件中的。因　段是不可　的，　些文件也都不会　化，　是　　存友好的，同　操作系　也会把　些段文件　存起来，以便更快的　　。

Lucene　　的性能取决于和操作系　的相互作用。如果　把所有的内存都分配　　Elasticsearch 的堆内存，那将不会有剩余的内存交　Lucene。　将　重地影　全文　索的性能。

　准的建　是把 50% 的可用内存作　Elasticsearch 的堆内存，保留剩下的 50%。当然它也不会被浪，Lucene 会很　意利用起余下的内存。

如果　不需要　分　字符串做聚合　算（例如，不需要　　fielddata　　）可以考降低堆内存。堆内存越小，Elasticsearch（更快的 GC）和 Lucene（更多的内存用于　存）的性能越好。

不要超　32 GB！

　里有　外一个原因不分配大内存　　Elasticsearch。事　上，　JVM　在内存小于　32　GB　的　候会采用一个内存　象指　　技　。

在　Java　中，所有的　象都分配在堆上，并通　一个指　行引用。　普通　象指　（OOP）指向　些　象，通常　CPU 字　的大小：32 位或 64 位，取决于　的　理器。指　引用的就是　个 OOP　的字　位置。

　于 32 位的系　，意味着堆内存大小最大　4 GB。　于 64 位的系　，　可以使用更大的内存，但是 64 位的指　意味着更大的浪　，因　　的指　本身大了。更糟　的是，　更大的指　在主内存和各　　存（例如 LLC，L1 等）之　移　数据的　候，会占用更多的　　。

Java 使用一个叫作 内存指 （compressed oops）的技 来解决 个 。 它的指 不再表示 象在内存中的精 位置，而是表示 偏移量 。 意味着 32 位的指 可以引用 40 个 象， 而不是 40 个字 。最 ， 也就是 堆内存 到 32 GB 的物理内存，也可以用 32 位的指 表示。

一旦 越 那个神奇的 ~32 GB 的 界，指 就会切回普通 象的指 。 个 象的指 都 了，就会使用更多的 CPU 内存 ，也就是 上失去了更多的内存。事 上，当内存到 40–50 GB 的 候，有效内存才相当于使用内存 象指 技 候的 32 GB 内存。

段描述的意思就是 ：即便 有足 的内存，也尽量不要 超 32 GB。因 它浪 了内存，降低了 CPU 的性能， 要 GC 大内存。

到底需要低于 32 GB多少，来 置我的 JVM？

憾的是， 需要看情况。 切的 分要根据 JVMs 和操作系 而定。 如果 想保 其安全可 ， 置堆内存 31 GB 是一个安全的 。 外， 可以在 的 JVM 置里添加 -XX:+PrintFlagsFinal 用来 JVM 的 界 ， 并且 UseCompressedOops 的 是否 true。 于 自己使用的 JVM 和操作系 ， 将 到最合 的堆内存 界 。

例如，我 在一台安装 Java 1.7 的 MacOSX 上 ，可以看到指 在被禁用之前，最大堆内存大 是在 32600 mb（~31.83 gb）：

```
$ JAVA_HOME=`/usr/libexec/java_home -v 1.7` java -Xmx32600m -XX:+PrintFlagsFinal 2>
/dev/null | grep UseCompressedOops
     bool UseCompressedOops    := true
$ JAVA_HOME=`/usr/libexec/java_home -v 1.7` java -Xmx32766m -XX:+PrintFlagsFinal 2>
/dev/null | grep UseCompressedOops
     bool UseCompressedOops    = false
```

相比之下，同一台机器安装 Java 1.8，可以看到指 在被禁用之前，最大堆内存大 是在 32766 mb（~31.99 gb）：

```
$ JAVA_HOME=`/usr/libexec/java_home -v 1.8` java -Xmx32766m -XX:+PrintFlagsFinal 2>
/dev/null | grep UseCompressedOops
     bool UseCompressedOops    := true
$ JAVA_HOME=`/usr/libexec/java_home -v 1.8` java -Xmx32767m -XX:+PrintFlagsFinal 2>
/dev/null | grep UseCompressedOops
     bool UseCompressedOops    = false
```

个例子告 我 ，影 内存指 使用的 界 ， 是会根据 JVM 的不同而 化的。 所以从其他地方 取的例子，需要 慎使用，要 操作系 配置和 JVM。

如果使用的是 Elasticsearch v2.2.0， 日志其 会告 JVM 是否正在使用内存指 。 会看到像 的日志消息：

```
[2015-12-16 13:53:33,417][INFO ][env] [Illyana Rasputin] heap size [989.8mb],
compressed ordinary object pointers [true]
```

表明内存指　　正在被使用。如果没有，日志消息会　示 `[false]`。

我有一个 1 TB 内存的机器！

　个　32　GB　的分割　是很重要的。那如果　的机器有很大的内存　　？　一台有着 512–768 GB内存的服　器愈　常　。

首先，我　建　避免使用　的高配机器（参考 硬件）。

但是如果　已　有了　的机器，　有三个可　：

- 主要做全文　索　？考　Elasticsearch 4 - 32 GB 的内存，　Lucene 通　操作系　文件存来利用余下的内存。那些内存都会用来　存 segments，　来　速的全文　索。
- 需要更多的排序和聚合？而且大部分的聚合　算是在数字、日期、地理点和　非分字符串上？　很幸　，　的聚合　算将在内存友好的 doc values 上完成！　Elasticsearch 4 到 32 GB 的内存，其余部分　操作系　存内存中的 doc values。
- 在　分　字符串做大量的排序和聚合（例如，　或者　SigTerms，等等）不幸的是，意味着　需要　fielddata，意味着　需要堆空　。考　在　个机器上　行　个或多个点，而不是　有大量 RAM 的一个　点。　然要　持 50% 原　。

假　有个机器有　128　GB　的内存，　可以　建　个　点，　个　点内存分配不超　32　GB。也就是　不超　64 GB 内存　ES 的堆内存，剩下的超　64 GB 的内存　Lucene。

如果　一　，　需要配置　`cluster.routing.allocation.same_shard.host:　true`　。　会防止同一个分片（shard）的主副本存在同一个物理机上（因　如果存在一个机器上，副本的高可用性就没有了）。

Swapping 是性能的　墓

　是　而易　的，但是　是有必要　的更清楚一点：内存交　到磁　服　器性能来　是　致命的。想想看：一个内存操作必　能　被快速　行。

如果内存交　到磁　上，一个　100　微秒的操作可能　成　10　秒。　再想想那　多　10　微秒的操作　延累加起来。不　看出 swapping　于性能是多　可怕。

最好的　法就是在　的操作系　中完全禁用 swap。　可以　禁用：

```
sudo swapoff -a
```

如果需要永久禁用，　可能需要修改 `/etc/fstab` 文件，　要参考　的操作系　相　文　。

如果　并不打算完全禁用　swap，也可以　降低　`swappiness`　的　。　个　决定操作系　交　内存的　率。　可以　防正常情况下　生交　，但　允　操作系　在　急情况下　生交　。

　于大部分Linux操作系　，可以在 `sysctl` 中　配置：

```
vm.swappiness = 1 ①
```

① swappiness 置 1 比 置 0 要好，因 在一些内核版本 swappiness 置 0 会触 系 OOM-killer（注：Linux 内核的 Out of Memory（OOM）killer 机制）。

最后，如果上面的方法都不合 ， 需要打 配置文件中的 mlockall 。 它的作用就是允 JVM 住内存，禁止操作系 交 出去。在 的 elasticsearch.yml 文件中， 置如下：

```
bootstrap.mlockall: true
```

文件描述符和 MMap

Lucene 使用了 大量的 文件。 同 ，Elasticsearch 在 点和 HTTP 客 端之 行通信也使用了大量的套接字（注：sockets）。 所有 一切都需要足 的文件描述符。

可悲的是， 多 代的 Linux 行版本， 个 程 允 一个微不足道的 1024 文件描述符。 一个小的 Elasticsearch 点来 在是太 低 了，更不用 一个 理数以百 索引的 点。

加 的文件描述符， 置一个很大的 ，如 64,000。 个 程困 得 人 火，它高度依 于 的特定操作系 和分布。 参考 操作系 文 来 定如何最好地修改允 的文件描述符数量。

一旦 已 改 了它， Elasticsearch，以 保它的真的起作用并且有足 的文件描述符：

```
GET /_nodes/process

{
   "cluster_name": "elasticsearch__zach",
   "nodes": {
      "TGn9iO2_QQKb0kavcLbnDw": {
         "name": "Zach",
         "transport_address": "inet[/192.168.1.131:9300]",
         "host": "zacharys-air",
         "ip": "192.168.1.131",
         "version": "2.0.0-SNAPSHOT",
         "build": "612f461",
         "http_address": "inet[/192.168.1.131:9200]",
         "process": {
            "refresh_interval_in_millis": 1000,
            "id": 19808,
            "max_file_descriptors": 64000, ①
            "mlockall": true
         }
      }
   }
}
```

① max_file_descriptors 字段 示 Elasticsearch 程可以 的可用文件描述符数量。

Elasticsearch 各 文件混合使用了 NioFs（ 注：非阻塞文件系 ）和 MMapFs （注：内存映射文件系 ）。 保 配置的最大映射数量，以便有足 的虚 内存可用于 mmapped 文件。 可以 置：

```
sysctl -w vm.max_map_count=262144
```

或者 可以在 /etc/sysctl.conf 通 修改 vm.max_map_count 永久 置它。

在生 之前，重温 个列表

在 入生 之前， 可能 了本 。本章中 及的 非常好，一般是可以知道的，但 是，正 部署到生 境之前需要重温 个列表。

一些 会 地阻止 (如：可用的文件描述符太少)。因 他 很快 出来， 些都是容易 的。其他的一些 ，如 裂和内存 置，只有在糟 的事情 生之后才可 。在 一点上，解决 法往往是凌乱和繁 的。

在 生 之前 ，通 当配置集群来主 阻止 些情况 生，是更好的 。所以如果 想要从整本 的一个部分折角（或保存 ），本章将是一个很好的 。在部署到生 境的前一周， 地里 出的列表，并 所有的建 。

部署后

一旦将集群部署到生 境后，就需要有一些工具及最佳 践 来保 集群 行在最佳状 。本章将探 配置、日志 、索引性能 化以及集群 。

更 置

Elasticsearch 里很多 置都是 的，可以通 API 修改。需要 制重 点（或者集群）的配置修改都要 力避免。而且 然通 静 配置 也可以完成 些 更，我 建 是用 API 来 。

集群更新 API 有 工作模式：

（*Transient*）
: 些 更在集群重 之前一直会生效。一旦整个集群重 ， 些配置就被清除。

永久（*Persistent*）
: 些 更会永久存在直到被 式修改。即使全集群重 它 也会存活下来并覆 掉静 配置文件里的 。

或永久配置需要在 JSON 体里分 指定：

```
PUT /_cluster/settings
{
    "persistent" : {
        "discovery.zen.minimum_master_nodes" : 2 ①
    },
    "transient" : {
        "indices.store.throttle.max_bytes_per_sec" : "50mb" ②
    }
}
```

① 个永久 置会在全集群重 存活下来。

② 个 置会在第一次全集群重 后被移除。

可以 更新的 置的完整清 , {ref}/cluster-update-settings.html[online reference docs]。

日志

Elasticsearch 会 出很多日志，都放在 ES_HOME/logs 目 下。 的日志 等 是 INFO 。它提供了 度的信息，但是又 好了不至于 的日志太 大。

当 的 候，特 是 点 相 的 （因 个 常依 于各式 于繁 的 配置），提高日志 等 到 DEBUG 是很有 助的。

可以 修改 logging.yml 文件然后重 的 点——但是 做即繁 会 致不必要的宕机 。作替代， 可以通 cluster-settings API 更新日志 ，就像我 前面 学 的那 。

要 个更新， 感 趣的日志器，然后在前面 上 logger. 。 根日志器 可以用 logger._root 来表示。

我 高 点 的日志 ：

```
PUT /_cluster/settings
{
    "transient" : {
        "logger.discovery" : "DEBUG"
    }
}
```

置失效，Elasticsearch 将 始 出 discovery 模 的 DEBUG 的日志。

TIP 避免使用 TRACE 。 个 非常的 , 到日志反而不再有用了。

慢日志

有 一个日志叫 慢日志 。 个日志的目的是捕 那些超 指定 的 和索引 求。个日志用来追踪由用 生的很慢的 求很有用。

情况，慢日志是不 的。要 它，需要定 具体 作（query，fetch 是 index），

期望的事件　　等　（WARN、DEBUG 等），以及　　。

　是一个索引　　的　置，也就是　可以独立　用　　个索引：

```
PUT /my_index/_settings
{
    "index.search.slowlog.threshold.query.warn" : "10s", ①
    "index.search.slowlog.threshold.fetch.debug": "500ms", ②
    "index.indexing.slowlog.threshold.index.info": "5s" ③
}
```

①　　慢于 10 秒　出一个 WARN 日志。

②　取慢于 500　秒　出一个 DEBUG 日志。

③ 索引慢于 5 秒　出一个 INFO 日志。

　也可以在　　elasticsearch.yml　　文件里定　些　。没有　　置的索引会自　承在静配置文件里配置的参数。

一旦　　置　了，　可以和其他日志器一　切　日志　　等　：

```
PUT /_cluster/settings
{
    "transient" : {
        "logger.index.search.slowlog" : "DEBUG", ①
        "logger.index.indexing.slowlog" : "WARN" ②
    }
}
```

①　置搜索慢日志　DEBUG　　。

②　置索引慢日志　WARN　　。

索引性能技巧

如果　是在一个索引　　很重的　境，比如索引的是基　　施日志，　可能　意　牲一些搜索性能　取更快的索引速率。在　些　景里，搜索常常是很少　的操作，而且一般是由　公司内部的人　起的。他　也　意　一个搜索等上几秒　，而不像普通消　者，要求一个搜索必　　秒　返回。

基于　　特殊的　景，我　可以有几　　衡　法来提高　的索引性能。

> **些技巧　　用于 Elasticsearch 1.3 及以后的版本**
>
> 本　是　最新几个版本的 Elasticsearch 写的，　然大多数内容在更老的版本也也有效。
>
> 不　，本　提及的技巧，　只　　1.3　及以后版本。　版本后有不少性能提升和故障修　是直接影　到索引的。事　上，有些建　在老版本上反而会因　故障或性能　陷而 降低 性能。

科学的　　性能

性能　　永　是　　的，所以在　的方法里已　要尽可能的科学。随机　弄旋　以及写入　　可不是做性能　　的好　法。如果有太多　可能，我　就无法判断到底　一　有最好的 效果 。合理的　　方法如下：

1. 在　个　点上，　　个分片，无副本的　景　　性能。
2. 在 100%　　配置的情况下　　性能　果，　　就有了一个　比基　。
3. 　保性能　　行足　　的　　（30　　分　以上）　　可以　估　期性能，而不是短期的峰　或延　。一些事件（比如段合并，GC）不会立刻　生，所以性能概况会随着　　而改　的。
4. 　始在基　上逐一修改　　。　格　　它　，如果性能提升可以接受，保留　个配置　，　始下一　。

使用批量　求并　整其大小

　而易　的，　化性能　　使用批量　求。批量的大小　取决于　的数据、分析和集群配置，不　　次批量数据　5–15　MB　大是个不　的起始点。注意　里　的是物理字　数大小。文　　数　批量大小来　不是一个好指　。比如　，如果　　次批量索引 1000 个文　，　住下面的事　：

- 1000 个 1 KB 大小的文　加起来是 1 MB 大。
- 1000 个 100 KB 大小的文　加起来是 100 MB 大。

　可是完完全全不一　的批量大小了。批量　求需要在　　点上加　　内存，所以批量　求的物理大小比文　　数重要得多。

从 5–15 MB　始　　批量　求大小，　慢　加　个数字，直到　看不到性能提升　止。然后　始　加　的批量写入的并　度（多　程等等　法）。

用 Marvel 以及　如 `iostat`　、　`top`　和　`ps`　等工具　控　的　点，　察　源什　　候　到瓶　。如果　始收到　　`EsRejectedExecutionException`　　，　的集群没　法再　　了：至少有一　　源到瓶了。或者　少并　数，或者提供更多的受限　源（比如从机械磁　　成 SSD），或者添加更多　点。

NOTE	写数据的　候，要　保批量　求是　　往　的全部数据　点的。不要把所有　求都　个　点，因　　个　点会需要在　理的　候把所有批量　求都存在内存里。

存

磁　在　代服　器上通常都是瓶　。Elasticsearch　　重度使用磁　，　的磁　能　理的　吐量越大，　的　点就越　定。　里有一些　化磁　I/O 的技巧：

- 使用 SSD。就像其他地方提　的，他　比机械磁　　秀多了。
- 使用　　RAID　　0。条　化　　RAID　　会提高磁　　I/O，代　　然就是当一　硬　故障整个就故障了。不要使用　像或者奇偶校　RAID 因　副本已　提供了　个功能。
- 　外，使用多　硬　，并允　　Elasticsearch　通　多个　`path.data`　目　配置把数据条　化分配到它上面。
- 不要使用　程挂　的存　，比如 NFS 或者 SMB/CIFS。　个引入的延　　性能来　完全是背道而　的。
- 如果　用的是 EC2，当心 EBS。即便是基于 SSD 的 EBS，通常也比本地　例的存　要慢。

段和合并

段合并的 算量 大，而且 要吃掉大量磁 I/O。合并在后台定期操作，因 他 可能要很 才能完成，尤其是比 大的段。 个通常来 都没 ，因 大 模段合并的概率是很小的。

不 有 候合并会 累写入速率。如果 个真的 生了，Elasticsearch 会自 限制索引 求到 个 程里。 个可以防止出 段爆炸 ，即数以百 的段在被合并之前就生成出来。如果 Elasticsearch 合并 累索引了，它会会 一个声明有 `now throttling indexing` 的 `INFO` 信息。

Elasticsearch 置在 比 保守：不希望搜索性能被后台合并影 。不 有 候（尤其是 SSD，或者日志 景）限流 太低了。

是 20 MB/s， 机械磁 是个不 的 置。如果 用的是 SSD，可以考 提高到 100–200 MB/s。 的系 个 合 ：

```
PUT /_cluster/settings
{
    "persistent" : {
        "indices.store.throttle.max_bytes_per_sec" : "100mb"
    }
}
```

如果 在做批量 入，完全不在意搜索， 可以 底 掉合并限流。 的索引速度 到 磁 允 的 限：

```
PUT /_cluster/settings
{
    "transient" : {
        "indices.store.throttle.type" : "none" ①
    }
}
```

① 置限流 型 `none` 底 合并限流。等 完成了 入， 得改回 `merge` 重新打 限流。

如果 使用的是机械磁 而非 SSD， 需要添加下面 个配置到 的 `elasticsearch.yml` 里：

```
index.merge.scheduler.max_thread_count: 1
```

机械磁 在并 I/O 支持方面比 差，所以我 需要降低 个索引并 磁 的 程数。 个 置允 `max_thread_count + 2` 个 程同 行磁 操作，也就是 置 `1` 允 三个 程。

于 SSD， 可以忽略 个 置， 是 `Math.min(3, Runtime.getRuntime().availableProcessors() / 2)`， SSD 来 行的很好。

最后， 可以 加 `index.translog.flush_threshold_size` 置，从 的 512 MB 到更大一些的 ，比如 1 GB。 可以在一次清空触 的 候在事 日志里 累出更大的段。而通 建更大的段，清空的 率 低，大段合并的 率也 低。 一切合起来 致更少的磁 I/O 和更好的索引速率。当然， 会需要 量 的 heap 内存用以 累更大的 冲空 ， 整 个 置的 候 住 点。

其他

最后，　有一些其他　得考　的　西需要　住：

- 如果　的搜索　果不需要近　　的准　度，考　把　个索引的　　`index.refresh_interval`改到　　`30s`。如果　是在做大批量　入，　入期　　可以通　　置　个　　`-1`　　掉刷新。　忘　在完工的候重新　　它。
- 如果　在做大批量　入，考　通　　置　　`index.number_of_replicas: 0`　副本。文　在　制的　候，整个文　内容都被　往副本　点，然后逐字的把索引　程重　一遍。　意味着　个副本也会　行分析、索引以及可能的合并　程。

  相反，如果　的索引是零副本，然后在写入完成后再　　副本，恢　　程本　上只是一个字　到字　的　　。相比重　索引　程，　个算是相当高效的了。

- 如果　没有　　个文　自　　ID，使用　Elasticsearch　的自　　ID　功能。　个　避免版本　　做了化，因　自　生成的 ID 是唯一的。
- 如果　在使用自己的 ID，　　使用一　Lucene 友好的 ID。包括零填充序列 ID、UUID-1 和　秒；　些 ID　都是有一致的，　　良好的序列模式。相反的，像　UUID-4　　的　ID，本　上是随机的，　比很低，会明　　慢 Lucene。

推　分片分配

正如我　在　[_scale_horizontally]　　，　Elasticsearch　将自　在可用　点　　行分片均衡，包括新　点的加入和　有　点的　　。

理　上来　，　个是理想的行　，我　想要提　副本分片来尽快恢　　失的主分片。　　我　同　也希望保　源在整个集群的均衡，用以避免　点。

然而，在　践中，立即的再均衡所造成的　　会比其解决的更多。　例来　，考　到以下情形：

1. Node（　点）19 在　　中失　了（某个家　　到了　源　）
2. Master　立即注意到了　个　点的　　，它决定在集群内提　其他　有　Node　19　上面的主分片　　的副本分片　主分片
3. 在副本被提　　主分片以后，master　　点　始　行恢　操作来重建　失的副本。集群中的　点之　互相拷　分片数据，　　力　　，集群状　　。
4. 由于目前集群　于非平衡状　，　个　程　有可能会触　小　模的分片移　。其他不相　的分片将在　点　　移来　到一个最佳的平衡状

与此同　，那个　到　源　的倒　管理　，把服　器　好　源　　行了重　，　在　点　　Node　　19 又重新加入到了集群。不幸的是，　个　点被告知当前的数据已　没有用了，　　数据已　在其他　点上重新分配了。所以 Node 19 把本地的数据　行　除，然后重新　始恢　集群的其他分片（然后　又致了一个新的再平衡）

如果　一切听起来是不必要的且　　大，那就　了。是的，不　前提是　知道　个　点会很快回来。如果　点 Node 19 真的　了，上面的流程　　正是我　想要　生的。

　了解决　　瞬　中断的　　，Elasticsearch　　可以推　分片的分配。　可以的集群在重新分配之前有　　去　　个　点是否会再次重新加入。

修改　　延

　　情况，集群会等待一分　来　看　点是否会重新加入，如果　个　点在此期　重新加入，重新加入的点会保持其　有的分片数据，不会触　新的分片分配。

通　修改参数 `delayed_timeout`，　　等待　　可以全局　置也可以在索引　　行修改：

```
PUT /_all/_settings ①
{
  "settings": {
    "index.unassigned.node_left.delayed_timeout": "5m" ②
  }
}
```

① 通　使用 `_all` 索引名，我　可以　集群里面的所有的索引使用　个参数

②　　被修改成了 5 分

　个配置是　　的，可以在　行　　行修改。如果　希望分片立即分配而不想等待，　可以　置参数：`delayed_timeout: 0`.

NOTE	延　分配不会阻止副本被提　　主分片。集群　是会　行必要的提　来　集群回到　yellow 状　。　失副本的重建是唯一被延　的　程。

自　取消分片　移

如果　点在超　之后再回来，且集群　没有完成分片的移　，会　生什　事情　？在　　情形下，Elasticsearch　　会　　机器磁　上的分片数据和当前集群中的活　主分片的数据是不是一　—如果　者匹配，　　明没有　来新的文　，包括　除和修改—那　　master　将会取消正在　行的再平衡并恢　机器磁　上的数据。

之所以　　做是因　本地磁　的恢　永　要比　　要快，并且我　保　了他　的分片数据是一　的，　个　程可以　是双　。

如果分片已　　生了分　（比如：　点　　之后又索引了新的文　），那　恢　　程会　　按照正常流程行。重新加入的　点会　除本地的、　　的数据，然后重新　取一　新的。

重

　有一天　会需要做一次集群的　　重　——保持集群在　和可操作，但是逐一把　点下　。

常　的原因：Elasticsearch　　版本升　，或者服　器自身的一些　　操作（比如操作系　升　或者硬件相　）。不管　　情况，都要有一　特　的方法来完成一次　　重　。

正常情况下，Elasticsearch　　希望　的数据被完全的　制和均衡的分布。如果　手　　了一个　点，集群会立刻　　点的　失并　始再平衡。如果　点的　　是短期工作的　，　一点就很　人了，因　大型分片的再平衡需要花　相当的　　（想想　　制　　1TB　　的数据——即便在高速　上也是不一般的事情了）。

我 需要的是，告 Elasticsearch 推 再平衡，因 外部因子影 下的集群状 ，我 自己更了解。操作流程如下：

1. 可能的 ，停止索引新的数据。 然不是 次都能真的做到，但是 一 可以 助提高恢 速度。
2. 禁止分片分配。 一 阻止 Elasticsearch 再平衡 失的分片，直到 告 它可以 行了。如果 知道 口会很短， 个主意棒 了。 可以像下面 禁止分配：

```
PUT /_cluster/settings
{
    "transient" : {
        "cluster.routing.allocation.enable" : "none"
    }
}
```

3. 个 点。
4. 行 /升 。
5. 重 点，然后 它加入到集群了。
6. 用如下命令重 分片分配：

```
PUT /_cluster/settings
{
    "transient" : {
        "cluster.routing.allocation.enable" : "all"
    }
}
```

分片再平衡会花一些 。一直等到集群 成 色 状 后再 。

7. 重 第 2 到 6 操作剩余 点。
8. 到 可以安全的恢 索引了（如果 之前停止了的 ），不 等待集群完全均衡后再恢 索引，也会有助于提高 理速度。

的集群

使用无 个存 数据的 件，定期 的数据都是很重要的。Elasticsearch 副本提供了高可性；它 可以容忍零星的 点 失而不会中断服 。

但是，副本并不提供 性故障的保 。 情况， 需要的是 集群真正的 ——在某些 西出 的 候有一个完整的拷 。

要 的集群， 可以使用 snapshot API。 个会拿到 集群里当前的状和数据然后保存到一个共享 里。 个 程是"智能"的。 的第一个快照会是一个数据的完整拷 ，但是所有后 的快照会保留的是已存快照和新数据之 的差 。随着 不 的 数据 行快照， 也在量的添加和 除。 意味着后 会相当快速，因 它 只 很小的数据量。

要使用 个功能， 必 首先 建一个保存数据的 。有多个 型可以供 ：

- 共享文件系 ，比如 NAS
- Amazon S3
- HDFS (Hadoop 分布式文件系)
- Azure Cloud

建

我部署一个共享文件系 ：

```
PUT _snapshot/my_backup ①
{
    "type": "fs", ②
    "settings": {
        "location": "/mount/backups/my_backup" ③
    }
}
```

① 我 的 取一个名字，在本例它叫 my_backup 。

② 我 指定 的 型 是一个共享文件系 。

③ 最后，我 提供一个已挂 的 作 目的地址。

注意：共享文件系 路径必 保集群所有 点都可以 到。

会在挂 点 建 和所需的元数据。 有一些其他的配置 可能想要配置的， 些取决于 的 点、的性能状况和 位置：

max_snapshot_bytes_per_sec
当快照数据 入 ， 个参数控制 个 程的限流情况。 是 秒 20mb 。

max_restore_bytes_per_sec
当从 恢 数据 ， 个参数控制什 候恢 程会被限流以保障 的 不会被占 。 是 秒 20mb。

假 我 有一个非常快的 ，而且 外的流量也很 OK，那我 可以 加 些 ：

```
POST _snapshot/my_backup/ ①
{
    "type": "fs",
    "settings": {
        "location": "/mount/backups/my_backup",
        "max_snapshot_bytes_per_sec" : "50mb", ②
        "max_restore_bytes_per_sec" : "50mb"
    }
}
```

① 注意我 用的是 POST 而不是 PUT 。 会更新已有 的 置。

② 然后添加我 的新 置。

快照所有打　的索引

一个　　可以包含多个快照。　个快照跟一系列索引相　（比如所有索引，一部分索引，或者　个索引）。当　建快照的　候，　指定　感　趣的索引然后　快照取一个唯一的名字。

我　从最基　的快照命令　始：

```
PUT _snapshot/my_backup/snapshot_1
```

个会　　所有打　的索引到　　my_backup　　下一个命名　　snapshot_1　　的快照里。　个用会立刻返回，然后快照会在后台　行。

TIP

通常　会希望　的快照作　后台　程　行，不　有　候　会希望在　的脚本中一直等待到完成。　可以通　添加一个 wait_for_completion　　：

```
PUT _snapshot/my_backup/snapshot_1?wait_for_completion=true
```

个会阻塞　用直到快照完成。注意大型快照会花很　　才返回。

快照指定索引

　行　是　　所有打　的索引。不　如果　在用　Marvel，　不是真的想要把所有　断相　的　.marvel 索引也　　起来。可能　就　根没那　大空　　所有数据。

　情况下，　可以在快照　的集群的　候指定　　些索引：

```
PUT _snapshot/my_backup/snapshot_2
{
    "indices": "index_1,index_2"
}
```

个快照命令　在只会　　index1 和 index2 了。

列出快照相　的信息

一旦　　始在　的　　里　　起快照了，　可能就慢慢忘　里面各自的　　了——特　是快照按照　　分命名的　候（比如，backup_2014_10_28 ）。

要　得　个快照的信息，直接　　和快照名　起一个 GET　求：

```
GET _snapshot/my_backup/snapshot_2
```

个会返回一个小　　，包括快照相　的各　信息：

```
{
   "snapshots": [
      {
         "snapshot": "snapshot_1",
         "indices": [
            ".marvel_2014_28_10",
            "index1",
            "index2"
         ],
         "state": "SUCCESS",
         "start_time": "2014-09-02T13:01:43.115Z",
         "start_time_in_millis": 1409662903115,
         "end_time": "2014-09-02T13:01:43.439Z",
         "end_time_in_millis": 1409662903439,
         "duration_in_millis": 324,
         "failures": [],
         "shards": {
            "total": 10,
            "failed": 0,
            "successful": 10
         }
      }
   ]
}
```

要 取一个 中所有快照的完整列表，使用 _all 占位符替 掉具体的快照名称：

```
GET _snapshot/my_backup/_all
```

除快照

最后，我 需要一个命令来 除所有不再有用的旧快照。 只要 /快照名称 一个 的 DELETE HTTP 用：

```
DELETE _snapshot/my_backup/snapshot_2
```

用 API 除快照很重要，而不能用其他机制（比如手 除，或者用 S3 上的自 清除工具）。因 快照是 量的，有可能很多快照依 于 去的段。delete API 知道 些数据 在被更多近期快照使用，然后会只 除不再被使用的段。

但是，如果 做了一次人工文件 除， 将会面 重 坏的 ，因 在 除的是可能 在使用中的数据。

控快照 度

wait_for_completion 提供了一个 控的基 形式，但 怕只是 一个中等 模的集群做快照恢 的 候，它都真的不 用。

外 个 API 会 有 快照状 更 的信息。首先 可以 快照 ID 行一个 GET，就像我 之前取一个特定快照的信息 做的那 ：

```
GET _snapshot/my_backup/snapshot_3
```

如果 用 个命令的 候快照 在 行中， 会看到它什 候 始， 行了多久等等信息。不 要注意， 个 API 用的是快照机制相同的 程池。如果 在快照非常大的分片，状 更新的 隔会很大，因 API 在 争相同的 程池 源。

更好的方案是 取 _status API 数据：

```
GET _snapshot/my_backup/snapshot_3/_status
```

_status API 立刻返回，然后 出 的多的 出：

```
{
   "snapshots": [
      {
         "snapshot": "snapshot_3",
         "repository": "my_backup",
         "state": "IN_PROGRESS", ①
         "shards_stats": {
            "initializing": 0,
            "started": 1, ②
            "finalizing": 0,
            "done": 4,
            "failed": 0,
            "total": 5
         },
         "stats": {
            "number_of_files": 5,
            "processed_files": 5,
            "total_size_in_bytes": 1792,
            "processed_size_in_bytes": 1792,
            "start_time_in_millis": 1409663054859,
            "time_in_millis": 64
         },
         "indices": {
            "index_3": {
               "shards_stats": {
                  "initializing": 0,
                  "started": 0,
                  "finalizing": 0,
                  "done": 5,
                  "failed": 0,
                  "total": 5
               },
               "stats": {
```

```
                  "number_of_files": 5,
                  "processed_files": 5,
                  "total_size_in_bytes": 1792,
                  "processed_size_in_bytes": 1792,
                  "start_time_in_millis": 1409663054859,
                  "time_in_millis": 64
               },
               "shards": {
                  "0": {
                     "stage": "DONE",
                     "stats": {
                        "number_of_files": 1,
                        "processed_files": 1,
                        "total_size_in_bytes": 514,
                        "processed_size_in_bytes": 514,
                        "start_time_in_millis": 1409663054862,
                        "time_in_millis": 22
                     }
                  },
                  ...
```

① 一个正在 行的快照会 示 IN_PROGRESS 作 状 。

② 个特定快照有一个分片 在 （ 外四个已 完成）。

包括快照的 体状况，但也包括下 到 个索引和 个分片的 。 个 展示了有 快照 展的非常 的 。分片可以在不同的完成状 ：

INITIALIZING
分片在 集群状 看看自己是否可以被快照。 个一般是非常快的。

STARTED
数据正在被 到 。

FINALIZING
数据 完成；分片 在在 送快照元数据。

DONE
快照完成！

FAILED
快照 理的 候 到了 ， 个分片/索引/快照不可能完成了。 的日志 取更多信息。

取消一个快照

最后， 可能想取消一个快照或恢 。因 它 是 期 行的 程， 行操作的 候一个 或者 就会花很 来解决——而且同 会耗尽有 的 源。

要取消一个快照，在他 行中的 候 的 除快照就可以：

```
DELETE _snapshot/my_backup/snapshot_3
```

个会中断快照 程。然后 除 里 行到一半的快照。

从快照恢

一旦 了数据，恢 它就 了：只要在 希望恢 回集群的快照 ID后面加上 _restore 即可：

```
POST _snapshot/my_backup/snapshot_1/_restore
```

行 是把 个快照里存有的所有索引都恢 。如果 snapshot_1 包括五个索引， 五个都会被恢到我 集群里。和 snapshot API 一 ，我 也可以 希望恢 具体 个索引。

有附加的 用来重命名索引。 个 允 通 模式匹配索引名称，然后通 恢 程提供一个新名称。如果 想在不替 有数据的前提下，恢 老数据来 内容，或者做其他 理， 个 很有用。我 从快照里恢 个索引并提供一个替 的名称：

```
POST /_snapshot/my_backup/snapshot_1/_restore
{
    "indices": "index_1", ①
    "rename_pattern": "index_(.+)", ②
    "rename_replacement": "restored_index_$1" ③
}
```

① 只恢 index_1 索引，忽略快照中存在的其余索引。

② 所提供的模式能匹配上的正在恢 的索引。

③ 然后把它 重命名成替代的模式。

个会恢 index_1 到 及群里，但是重命名成了 restored_index_1 。

TIP

和快照 似， restore 命令也会立刻返回，恢 程会在后台 行。如果 更希望 的 HTTP 用阻塞直到恢 完成，添加 wait_for_completion ：

```
POST _snapshot/my_backup/snapshot_1/_restore?wait_for_completion=true
```

控恢 操作

从 恢 数据借 了 Elasticsearch 里已有的 行恢 机制。在内部 上，从 恢 分片和从一个 点恢 是等 的。

如果 想 控恢 的 度， 可以使用 recovery API。 是一个通用目的的 API，用来展示 集群中移着的分片状 。

个 API 可以 在恢 的指定索引 独 用：

```
GET restored_index_3/_recovery
```

或者 看 集群里所有索引，可能包括跟 的恢 程无 的其他分片移 ：

```
GET /_recovery/
```

出会跟 个 似（注意，根据 集群的活 度， 出可能会 得非常 ！）：

```
{
  "restored_index_3" : {
    "shards" : [ {
      "id" : 0,
      "type" : "snapshot", ①
      "stage" : "index",
      "primary" : true,
      "start_time" : "2014-02-24T12:15:59.716",
      "stop_time" : 0,
      "total_time_in_millis" : 175576,
      "source" : { ②
        "repository" : "my_backup",
        "snapshot" : "snapshot_3",
        "index" : "restored_index_3"
      },
      "target" : {
        "id" : "ryqJ5lO5S4-lSFbGntkEkg",
        "hostname" : "my.fqdn",
        "ip" : "10.0.1.7",
        "name" : "my_es_node"
      },
      "index" : {
        "files" : {
          "total" : 73,
          "reused" : 0,
          "recovered" : 69,
          "percent" : "94.5%" ③
        },
        "bytes" : {
          "total" : 79063092,
          "reused" : 0,
          "recovered" : 68891939,
          "percent" : "87.1%"
        },
        "total_time_in_millis" : 0
      },
      "translog" : {
        "recovered" : 0,
        "total_time_in_millis" : 0
      },
      "start" : {
        "check_index_time" : 0,
        "total_time_in_millis" : 0
      }
    } ]
  }
}
```

① type 字段告　　恢　的本　；　个分片是在从一个快照恢　。

② source 哈希描述了作　恢　来源的特定快照和　　。

③ percent 字段　　恢　的状　有个概念。　个特定分片目前已　恢　了 94% 的文件；它就快完成了。

　出会列出所有目前正在　　恢　的索引，然后列出　些索引里的所有分片。　个分片里会有　　/停止、持　　、恢　百分比、　　字　数等　　。

取消一个恢

要取消一个恢　，　需要　除正在恢　的索引。因　恢　　程其　就是分片恢　，　送一个　除索引　API 修改集群状　，就可以停止恢　　程。比如：

```
DELETE /restored_index_3
```

如果 restored_index_3 正在恢　中，　个　除命令会停止恢　，同　　除所有已　恢　到集群里的数据。

集群是活着的、呼吸着的生命

一旦　的集群投入生　，　会　　他就　始了他自己的一生。Elasticsearch　　努力工作来保　集群自　自足而且 真就在工作 。不　一个集群也　要有日常照料和投　，比如日常　　和升　。

Elasticsearch　　以非常快的速度　布新版本，　行　　修　和性能　　。保持　的集群采用最新版　是一个好主意。　似的，Lucene　　持　在　　JVM　　自身的新的和令人　　的　　，　意味着　需要尽量保持　的 JVM 是最新的。

　意味着最好是　有一个　准化的、日常的方案来操作　集群的　　重　和升　。升　　是一个日常程序，而不是一个需要好多个小　的精　　下的年度『惨　』。

　似的，　有一个　　是很重要的。　　的集群做　繁的快照——而且通　　行真　恢　的方式定期　　些快照！有些　　做日常　　却从不　　他　的恢　机制，　　直太常　了。通常　会在第一次演　真　恢　的　候　　明　的　陷（比如用　不知道　　挂　　个磁　）。比起在凌晨　　3　　点真的生危机的　候，在日常　　中暴露出　些　　是更好的。