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[segments-and-merging]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 [indexing-performance]。

  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， [heap-sizing] 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 [logging] 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。