## 日期筛选

 日期筛选可以让搜索限定在指定的日期内，而不是搜索全部内容，通过时间限定，可以从集群中减少搜索的内容，提高搜索效率和减少资源占用。例如只搜索最近两天的错误日志。

备注：几乎所有的API都支持日期筛选。

  日期筛选的语法为：

<static\_name{date\_math\_expr{date\_format|time\_zone}}>

语法解释：

static\_name ：索引的名称；

date\_math\_expr：动态日期计算表达式；

date\_format ：日期格式；

time\_zone：时区，默认为UTC。

例如：

curl -XGET 'localhost:9200/<logstash-{now%2Fd-2d}>/\_search' {  
  "query" : {  
    ...  
  }  
}

备注：由于url编码的问题/被替换成了%2F。

假设当前时间为2013.7.17日中午，下面列举几个例子：

<secilog-{now/d}>:secilog-20130.7.17

<secilog-{now/M}>secilog-2013.07.01

<secilog-{now/M{YYYY.MM}}>secilog-2013.07

<secilog-{now/M-1M{YYYY.MM}}>secilog-2013.06

<secilog-{now/d{YYYY.MM.dd|+12:00}}secilog-2013.7.18

备注：如果索引名称有{}，可以通过添加\来代替，如：

<elastic\\{ON\\}-{now/M}>被转换为 elastic{ON}-2013.07.01

时间搜索也可以通过逗号，来选择多个时间，例如，选择最近三天的数据：

curl -XGET 'localhost:9200/<secilog-{now%2Fd-2d}>,<secilog-{now%2Fd-1d}>,<secilog-{now%2Fd}>/\_search' {  
  "query" : {  
    ...  
  }  
}

## 通用参数

pretty参数，当你在任何请求中添加了参数?pretty=true时，请求的返回值是经过格式化后的JSON数据，这样阅读起来更加的方便。系统还提供了另一只格式的格式化，?format=yaml，YAML格式，这将导致返回的结果具有可读的YAML格式。

 human参数，对于统计数据，系统支持计算机数据，同时也支持比较日和人类阅读的数据，比如：计算机数据"exists\_time\_in\_millis": 3600000 or "size\_in\_bytes": 1024，更适合人类阅读的数据："exists\_time": "1h" or "size": "1kb"。当?human=false 的时候，只输出计算机数据，当?human=ture的时候输出更适合人类阅读的数据，但这会消耗更多的资源，默认是false。

  日期表达式，大多数参数接受格式化日期表达式 ， 如范围查询范围查询gt（大于）和lt（小于），或在日期聚合中用from to来表达时间范围。表达式设定的日期为now或者日期字符串加||。

• +1h - 增加一小时

• -1d - 减少一个小时

•/d - 上一个小时

所支持的时间单位为：y（年）、M（月）、w（周）、d（日）、h（小时）、m（分钟）、s（秒）。

例如：

now+1h ：当前时间加一小时，以毫秒为单位。

now+1h+1m ：当前时间加一小时和一分钟，以毫秒为单位。

now+1h/d ：当前时间加一小时，四舍五入到最近的一天。

2015-01-01||+1M/d ：2015-01-01加一个月，向下舍入到最接近的一天。

  响应过滤(filter\_path)。所有的返回值可以通过filter\_path来减少返回值的内容，多个值可以用逗号分开。例如：

curl -XGET 'localhost:9200/\_search?pretty&filter\_path=took,hits.hits.\_id,hits.hits.\_score'{  
  "took" : 3,  
  "hits" : {  
    "hits" : [  
      {  
        "\_id" : "3640",  
        "\_score" : 1.0  
      },  
      {  
        "\_id" : "3642",  
        "\_score" : 1.0  
      }  
    ]  
  }  
}

它也支持通配符\*匹配任何部分字段的名称，例如：

curl -XGET 'localhost:9200/\_nodes/stats?filter\_path=nodes.\*.ho\*'{  
  "nodes" : {  
    "lvJHed8uQQu4brS-SXKsNA" : {  
      "host" : "portable"  
    }  
  }  
}

我们可以用两个通配符\*\*来匹配不确定名称的字段，例如我们可以返回Lucene版本的段信息：

curl 'localhost:9200/\_segments?pretty&filter\_path=indices.\*\*.version'{  
  "indices" : {  
    "movies" : {  
      "shards" : {  
        "0" : [ {  
          "segments" : {  
            "\_0" : {  
              "version" : "5.2.0"  
            }  
          }  
        } ],  
        "2" : [ {  
          "segments" : {  
            "\_0" : {  
              "version" : "5.2.0"  
            }  
          }  
        } ]  
      }  
    },  
    "books" : {  
      "shards" : {  
        "0" : [ {  
          "segments" : {  
            "\_0" : {  
              "version" : "5.2.0"  
            }  
          }  
        } ]  
      }  
    }  
  }  
}

注意，有时直接返回Elasticsearch的某个字段的原始值，如\_source字段。如果你想过滤\_source字段，可以结合\_source字段和filter\_path参数，例如：

curl -XGET 'localhost:9200/\_search?pretty&filter\_path=hits.hits.\_source&\_source=title'{  
  "hits" : {  
    "hits" : [ {  
      "\_source":{"title":"Book #2"}  
    }, {  
      "\_source":{"title":"Book #1"}  
    }, {  
      "\_source":{"title":"Book #3"}  
    } ]  
  }  
}

紧凑参数flat\_settings，flat\_settings为true时候返回的内容更加的紧凑，false是返回的值更加的容易阅读。例如为true的时候：

{  
  "persistent" : { },  
  "transient" : {  
    "discovery.zen.minimum\_master\_nodes" : "1"  
  }  
}

为false的时候，默认的情况下为false：

{  
  "persistent" : { },  
  "transient" : {  
    "discovery" : {  
      "zen" : {  
        "minimum\_master\_nodes" : "1"  
      }  
    }  
  }  
}

#### 基于URL的访问控制

  当多用户通过URL访问Elasticsearch索引的时候，为了防止用户误删除等操作，可以通过基于URL的访问控制来限制用户对某个具体索引的访问。可以在配置文件中添加参数：rest.action.multi.allow\_explicit\_index: false。这个参数默认为true。当为false的时候。在请求参数中指定具体索引的请求将会被拒绝。

## 分片选择(routing)

    可以在查询的时候指定路由选择(routing),当路由不存在的时候，返回为空值，此实例是在事先做了路由的操作，例如：

请求：GET http://localhost:9200/secilog/log/1?routing=secisland&pretty

返回的内容：

{

  "\_index" : "secilog",

  "\_type" : "log",

  "\_id" : "1",

  "\_version" : 1,

  "\_routing" : "secisland",

  "found" : true,

  "\_source" : {

    "collect\_type" : "syslog",

    "collect\_date" : "2016-01-11T09:32:12",

    "message" : "Failed password for root from 192.168.21.2 port 50790 ssh2"

  }

}

通过参数控制，查询的时候可以指定查询是在主节点上查询还是在副本节点上查询。

\_primary：在主节点进行查询；

\_local：尽可能在本地节点上进行查询；

刷新参数：

    refresh参数可以被设置为true，使之在搜索操作前刷新相关的分片保证可以及时查询到。但这个参数会消耗系统的资源，除非有必要，正常情况下不需要设置。

## 更新删除文档

*摘要: Elasticsearch的更新文档API准许通过脚本操作来更新文档。更新操作从索引中获取文档，执行脚本，然后获得返回结果。它使用版本号来控制文档获取或者重建索引。在最新版本的Elasticsearch中,基于安全考虑(如果用不到,请保持禁用),默认禁用了动态脚本功能。*

    Elasticsearch的更新文档API准许通过脚本操作来更新文档。更新操作从索引中获取文档，执行脚本，然后获得返回结果。它使用版本号来控制文档获取或者重建索引。

 备注：在Elasticsearch中的更新操作是完全从新索引文件。

    我们新建一个文档：

请求：PUT http://localhost:9200/test/type1/1?pretty

参数：

{

    "counter" : 1,

    "tags" : ["red"]

}

#### 脚本开启功能

    在最新版本的Elasticsearch中,基于安全考虑(如果用不到,请保持禁用),默认禁用了动态脚本功能.如果被禁用，在使用脚本的时候则报以下的错误：

scripts of type [inline], operation [update] and lang [groovy] are disabled

    可以用以下方式完全开启动态脚本功能，在config/elasticsearch.yml文件,在最后添加以下代码：

script.inline: on

script.indexed: on

script.file: on

配置后，重启Elasticsearch。

下面我们用脚本来更新此文档。

请求：POST http://localhost:9200/test/type1/1/\_update?pretty

参数：

{

    "script" : {

        "inline": "ctx.\_source.counter += count",

        "params" : {

            "count" : 4

        }

    }

}

执行完后，我们在查询一下文档内容，可以发现counter的值为5：

{

  "\_index" : "test",

  "\_type" : "type1",

  "\_id" : "1",

  "\_version" : 5,

  "found" : true,

  "\_source" : {

    "counter" : 5,

    "tags" : [ "red" ]

  }

}

在看下面的更新操作：

请求：POST http://localhost:9200/test/type1/1/\_update?pretty

参数：

{

    "script" : {

        "inline": "ctx.\_source.tags += tag",

        "params" : {

            "tag" : "blue"

        }

    }

}

返回的内容为，表示更新成功，我们看一下\_version为6，比刚才的值增加了1：

{

  "\_index" : "test",

  "\_type" : "type1",

  "\_id" : "1",

  "\_version" : 6,

  "\_shards" : {

    "total" : 2,

    "successful" : 1,

    "failed" : 0

  }

}

然后我们在查询一下文档内容：

{

  "\_index" : "test",

  "\_type" : "type1",

  "\_id" : "1",

  "\_version" : 6,

  "found" : true,

  "\_source" : {

    "counter" : 5,

    "tags" : [ "red", "blue" ]

  }

}

在脚本中除了\_source外其他内置参数也可以使用，例如\_index, \_type, \_id, \_version, \_routing, \_parent, \_timestamp, \_ttl。

下面我们通过脚本增加一列。

请求：POST http://localhost:9200/test/type1/1/\_update?pretty

参数：

{

    "script" : "ctx.\_source.name\_of\_new\_field = \"value\_of\_new\_field\""

}

然后查询此文档：

{

  "\_index" : "test",

  "\_type" : "type1",

  "\_id" : "1",

  "\_version" : 7,

  "found" : true,

  "\_source" : {

    "counter" : 5,

    "tags" : [ "red", "blue" ],

    "name\_of\_new\_field" : "value\_of\_new\_field"

  }

}

从中可以看出，文档中又增加了一列。

删除一列，请求和刚才的一样，参数变为：

{

    "script" : "ctx.\_source.remove(\"name\_of\_new\_field\")"

}

甚至可以通过表达式来判断做某些事情，例如：下面的示例将删除的文件如果标签字段包含蓝色，否则什么也不做（空）：

请求参数：

{

    "script" : {

        "inline": "ctx.\_source.tags.contains(tag) ? ctx.op = \"delete\" : ctx.op = \"none\"",

        "params" : {

            "tag" : "blue"

        }

    }

}

部分文档更新：

    该更新接口还支持更新部分文档，将文档合并到现有文档中（简单的递归合并、对象的内部合并、替换核心的“键/值”和数组）。例如：

{

    "doc" : {

        "name" : "new\_name"

    }

}

更新后，可以发现文档中多了一列name。

{

  "\_index" : "test",

  "\_type" : "type1",

  "\_id" : "1",

  "\_version" : 23,

  "found" : true,

  "\_source" : {

    "counter" : 5,

    "tags" : [ "red", "blue" ],

    "name" : "new\_name"

  }

}

    当文档指定的值与现有的\_source合并。当新的文档和老的文档不一致的时候，文档将会被从新建立索引。当新旧文档一样的时候，则不进行从建索引的操作。可以通过设置detect\_noop为false，让任何情况下都从新建立索引，例如下面的更新操作：

{

    "doc" : {

        "name" : "new\_name"

    },

    "detect\_noop": false

}

#### 删除文档

    删除文档相对比较简单：

请求：DELETE http://localhost:9200/test/type1/1

返回的内容为：

{

    "found": true,

    "\_index": "test",

    "\_type": "type1",

    "\_id": "1",

    "\_version": 24,

    "\_shards": {

        "total": 2,

        "successful": 1,

        "failed": 0

    }}

则表示删除了此文档。

## 新增文档

*在Elasticsearch中新增文档是一项基本的操作，创建文档类似数据库中的增加操作，但又和数据库中的概念不一样，主要的区别在Elasticsearch中新增文档和修改文档可以是同一个语法。Elasticsearch中对每个文档还有版本号的概念等等。*

#### 新增文档

    新增文档是把一条新的文档增加到索引中，使之能够进行搜索，文档的格式是JSON格式。 注意：在Elasticsearch中如果文档的id存在，则更新此文档。例如我们在索引secilog中下面增加一条文档：

请求：PUT http://localhost:9200/secilog/log/1

参数：

{

    "collect\_type" : "syslog",

    "collect\_date" : "2016-01-11T09:32:12",

    "message" : "Failed password for root from 192.168.21.1 port 50790 ssh2"

}

返回结果：

{

    "\_index": "secilog",

    "\_type": "log",

    "\_id": "1",

    "\_version": 1,

    "\_shards": {

        "total": 2,

        "successful": 1,

        "failed": 0

    },

    "created": true

}

返回结果中的\_shards说明，\_shards提供了索引创建的过程信息:

total:文档被创建的时候，在多少个分片中进行了操作，包括主分片和副本分片；

successful：成功建立索引分片的数量，当创建成功后，成功创建索引分片的数量最少是1；

failed:失败建立索引分片的数量。

#### 自动创建索引

    当创建文档的时候，如果索引不存在，则会自动创建该索引。自动创建的索引会自动映射每个字段的类型。自动创建字段类型是非常灵活的，新的字段类型将会自动匹配字段对象的类型。比如字符串类型，日期类型。自动创建索引可以通过配置文件设置action.auto\_create\_index为false在所有节点的配置文件中禁用。自动映射的字段类型可以通过配置文件设置index.mapper.dynamic为false禁用。自动创建索引可以通过模板设置索引名称，例如：可以设置 action.auto\_create\_index为 +aaa\*,-bbb\*,+ccc\*,-\* (+表示准许，-表示禁止)。

#### 版本号

    每个文档都有一个版本号，版本号的具体值放在在创建索引的返回值中("\_version":)。通过版本号参数可以达到乐观并发控制的效果。当在操作文档的过程中指定版本号，如果和版本号不一致的时候操作会被拒绝。版本号常用在对事务的处理中。例如，更新刚才创建的文档：

请求：PUT http://localhost:9200/secilog/log/1?version=2&pretty

参数：

{

    "message" : "elasticsearch now has versioning support!"

}

返回内容为更新失败：

{

  "error" : {

    "root\_cause" : [ {

      "type" : "version\_conflict\_engine\_exception",

      "reason" : "[log][1]: version conflict, current [-1], provided [2]",

      "index" : "secilog",

      "shard" : "2"

    } ],

    "type" : "version\_conflict\_engine\_exception",

    "reason" : "[log][1]: version conflict, current [-1], provided [2]",

    "index" : "secilog",

    "shard" : "2"

  },

  "status" : 409

}

注意：版本号是实时更新的，不会存在缓存现象。当操作的时候不指定版本号，则系统不会对版本号是否一致进行检查。

    默认情况下对文档的操作版本号从1开始递增，包括修改文档和删除文档。当然版本号还可以从外部获取，比如从数据库中获取，要启用此功能，version\_type应设置为external，这个值必须是一个大于0小于 9.2e+18的数字。当使用外部版本号来代替自动生成的版本号时，在操作文档的时候，系统通过对比参数中的版本号是否大于文档中的版本号来做判断，当参数中的版本号大于系统中的版本号，则执行此操作，并更新版本号。反之则拒绝操作(包括小于或者等于)。

    版本号同时产生了一个比较实用的功能，只要版本号从源数据库中使用，在异步索引操作的时候就不需要对源数据库的变化执行严格排序。任何操作都只会对最新的版本号起作用，不管这个版本号是内部的还是从外部获取的。

本文由赛克蓝德(secisland)原创，转载请标明作者和出处。

#### 操作类型

    系统同时支持通过op\_type=create参数强制命令只执行创建操作，只有系统中不存在此文档的时候才会创建成功。如果不指定此操作类型，如果存在此文档，则会更新此文档。例如再次创建文档：

请求：PUT http://localhost:9200/secilog/log/1?op\_type=create&pretty

参数：

{

    "collect\_type" : "syslog",

    "collect\_date" : "2016-01-11T09:32:12",

    "message" : "Failed password for root from 192.168.21.2 port 50790 ssh2"

}

返回值，表示创建失败：

{

  "error" : {

    "root\_cause" : [ {

      "type" : "document\_already\_exists\_exception",

      "reason" : "[log][1]: document already exists",

      "index" : "secilog",

      "shard" : "3"

    } ],

    "type" : "document\_already\_exists\_exception",

    "reason" : "[log][1]: document already exists",

    "index" : "secilog",

    "shard" : "3"

  },

  "status" : 409

}

当不指定op\_type=create时候，则更新此文档。

创建操作的另一个写法为：http://localhost:9200/secilog/log/1/\_create?pretty

#### 自动创建ID

    当创建文档的时候，如果不指定id，系统会自动创建id。自动生成的id是一个不会重复的随机数。例如：

请求：POST http://localhost:9200/secilog/log/?op\_type=create&pretty

参数：

{

    "collect\_type" : "syslog",

    "collect\_date" : "2016-01-11T09:32:12",

    "message" : "Failed password for root from 192.168.21.2 port 50790 ssh2"

}

返回值：

{

  "\_index" : "secilog",

  "\_type" : "log",

  "\_id" : "AVLOV99W6rG7Qqt6i\_gk",

  "\_version" : 1,

  "\_shards" : {

    "total" : 2,

    "successful" : 1,

    "failed" : 0

  },

  "created" : true

}

从中可以看出\_id自动生成了一个随机数。

#### 分片选择

    默认情况下，分片的选择是通过ID的散列值进行控制，分片的选择。这个只可以通过router 参数进行手动的控制。可以在每个操作的基础上直接通过哈希函数的值来指定分片的选择。例如：

请求：POST http://localhost:9200/secilog/log/?routing=secisland&pretty

在上面的例子中，分片的选择是通过指定routing=secisland参数的哈希值来确定的。

#### 其他说明

分布式：索引操作主要是针对主节点的分片进行，当主节点完成索引操作后，如果有副本节点，则分发到副本中。

一致性：为了防止当网络出现问题的时候写入不一致，系统只有在有效节点的数量大于一定数量的时候生效(总结点数/2+1)，该值可以通过action.write\_consistency参数进行修改。

刷新：更新的时候可以指定refresh 参数为true来立即刷新所有的副本，当refresh设置为true的时候，系统做了充分的优化，不会对系统产生任何影响，需要注意的是查询操作refresh参数没有任何的意义。

空操作：当文档内容没有任何改变的时候，更新文档操作也会生效，具体体现在版本号会发生变化。如果不希望此情况发生，在更新的时候指定 detect\_noop为true。这个参数在创建索引的时候无效。

超时：默认情况下系统的超时时间是一分钟。可以通过设置timeout来修改超时的时间，例如timeout=5m，表示超时的时间是5分钟。

## 多文档操作

*在Elasticsearch对文档的操作中，之前介绍的都是对单个文档进行操作，其实Elasticsearch同时可以对多个文档同时操作,多文档查询可以在同一个接口中查询多个文档，可以分别指定index，type，id来进行多个文档的查询。*

    在Elasticsearch对文档的操作中，之前介绍的都是对单个文档进行操作，其实Elasticsearch同时可以对多个文档同时操作。下面介绍多文档查询。

#### 多文档查询

    多文档查询可以在同一个接口中查询多个文档，可以分别指定index，type，id来进行多个文档的查询，响应包括所有查询到文档数组，每个元素在结构上类似于单个文档查询，例如：

请求：POST  http://localhost:9200/\_mget?pretty

参数：

{

    "docs" : [

        {

            "\_index" : "secilog",

            "\_type" : "log",

            "\_id" : "1"

        },

        {

            "\_index" : "secilog",

            "\_type" : "log",

            "\_id" : "2"

        }

    ]

}

返回结果：

{

  "docs" : [ {

    "\_index" : "secilog",

    "\_type" : "log",

    "\_id" : "1",

    "\_version" : 3,

    "found" : true,

    "\_source" : {

      "collect\_type" : "syslog",

      "collect\_date" : "2016-01-11T09:32:12",

      "message" : "Failed password for root from 192.168.21.2 port 50790 ssh2"

    }

  }, {

    "\_index" : "secilog",

    "\_type" : "log",

    "\_id" : "2",

    "\_version" : 1,

    "found" : true,

    "\_source" : {

      "collect\_type" : "syslog",

      "collect\_date" : "2016-01-12T09:32:12",

      "message" : "secisland mget test!"

    }

  } ]

}

从中可以看出，一次查询了两个文档。

    在查询的时候，index，type可以在url中直接填写。例如下面两个请求和之前的是等价的。

请求：POST http://localhost:9200/secilog/\_mget?pretty

参数：

{

    "docs" : [

        {

            "\_type" : "log",

            "\_id" : "1"

        },

        {

            "\_type" : "log",

            "\_id" : "2"

        }

    ]

}

请求：POST http://localhost:9200/secilog/log/\_mget?pretty

参数：

{

    "docs" : [

        {

            "\_id" : "1"

        },

        {

            "\_id" : "2"

        }

    ]

}

对于上一种，可以用更加简化的方式查询：

请求：POST http://localhost:9200/secilog/log/\_mget?pretty

参数：

{

    "ids" : ["1","2" ]

}

从上面的例子可以看出，Elasticsearch的多文档查询还是很灵活的。

type参数说明

    在多文档查询中，\_type允许为空，当他设置为空或者\_all的时候，系统会匹配第一个查询到的结果。如果不设置\_type，当有许多文件有相同的\_id的时候，系统最终得到的只有第一个匹配的文档。例如：

请求：POST http://localhost:9200/secilog/\_mget?pretty

参数：

{

    "ids" : ["1","2" ]

}

上面的查询当有多个type中都有1，2id的时候，系统只会返回第一个找到的文档。如果想要多个，就需要把type在请求参数中指出来。

    默认情况下，\_source字段将在每个文件中返回（如果存储）。类似单个文档的查询，可以在url中指定\_source, \_source\_include 或者\_source\_exclude来对查询的结果进行过滤。例如：

请求：POST http://localhost:9200/secilog/log/\_mget?pretty

参数：

{

    "docs" : [

        {

            "\_id" : "1",

            "\_source" : false

        },

        {

            "\_id" : "2",

            "\_source" : ["collect\_type", "collect\_date"]

        }

    ]

}

返回结果：

{

  "docs" : [ {

    "\_index" : "secilog",

    "\_type" : "log",

    "\_id" : "1",

    "\_version" : 3,

    "found" : true

  }, {

    "\_index" : "secilog",

    "\_type" : "log",

    "\_id" : "2",

    "\_version" : 1,

    "found" : true,

    "\_source" : {

      "collect\_date" : "2016-01-12T09:32:12",

      "collect\_type" : "syslog"

    }

  } ]

}

类似单个文档查询，在请求的url中或者参数的docs中可以指定field，routing参数。

#### 块操作

    快操作可以在一个接口中，处理个文档的内容，包括创建文档，删除文档，和修改文档。用块操作方式操作多个文档可以提高系统的效率。例如：

请求：POST http://localhost:9200/\_bulk?pretty

参数：

{ "index" : { "\_index" : "test", "\_type" : "type1", "\_id" : "10" } }

{ "field1" : "value1" }

{ "index" : { "\_index" : "test", "\_type" : "type1", "\_id" : "13" } }

{ "field1" : "value3" }

{ "delete" : { "\_index" : "test", "\_type" : "type1", "\_id" : "12" } }

}

返回结果：

{

  "took" : 1,

  "errors" : false,

  "items" : [ {

    "index" : {

      "\_index" : "test",

      "\_type" : "type1",

      "\_id" : "10",

      "\_version" : 6,

      "\_shards" : {

        "total" : 2,

        "successful" : 1,

        "failed" : 0

      },

      "status" : 200

    }

  }, {

    "index" : {

      "\_index" : "test",

      "\_type" : "type1",

      "\_id" : "13",

      "\_version" : 1,

      "\_shards" : {

        "total" : 2,

        "successful" : 1,

        "failed" : 0

      },

      "status" : 201

    }

  }, {

    "delete" : {

      "\_index" : "test",

      "\_type" : "type1",

      "\_id" : "12",

      "\_version" : 2,

      "\_shards" : {

        "total" : 2,

        "successful" : 1,

        "failed" : 0

      },

      "status" : 404,

      "found" : false

    }

  } ]

}

    和批量查询类似， /\_bulk, /{index}/\_bulk, and {index}/{type}/\_bulk这三种方式都可以执行，只需要在请求的参数中做出相应的对应

## 索引词频率（作用？）

<https://my.oschina.net/secisland/blog/614627>

## 搜索详解

*Elasticsearch搜索有两种方式，一种是通过url参数进行搜索，另一种是通过POST请求参数进行搜索。每种搜索都有很多的参数可以选择，下面详细介绍相关的参数。*

    之前的文章中简单介绍过搜索，在这里对搜索进行详细的介绍。在之前的介绍中可以知道，搜索有两种方式，一种是通过url参数进行搜索，另一种是通过POST请求参数进行搜索。

#### url参数搜索

请求：GET http://localhost:9200/secilog/log/\_search?参数，多个参数用&分来。

参数详解：

|  |  |
| --- | --- |
| 参数 | 解释 |
| q | 查询字符串，例如：q=syslog |
| df | 当查询中没有定义前缀的时候默认使用的字段。 |
| analyzer | 当分析查询字符串的时候使用的分词器 |
| lowercase\_expanded\_terms | 搜索的时候忽略大小写标志，默认为true |
| analyze\_wildcard | 通配符或者前缀查询是否被分析，默认false |
| default\_operator | 默认多个条件的关系，AND或者OR，默认OR |
| lenient | 如果设置为true，字段类型转换失败的时候将被忽略，默认为false |
| explain | 在每个返回结果中，将包含评分机制的解释 |
| \_source | 是否包含元数据，同时支持\_source\_include 和\_source\_exclude |
| fields | 只放回索引中指定的列，多个列中间用逗号分开 |
| sort | 排序，例如fieldName:asc或者fieldName:desc |
| track\_scores | 评分轨迹，当排序的时候，设置为true的时候返回评分的信息 |
| timeout | 超时的时间设置 |
| terminate\_after | 在每个分片中查询的最大条数，如果设置返回结果中会有一个terminated\_early字段 |
| from | 开始的记录数 |
| size | 搜索结果中的条数 |
| search\_type | 搜索的类型，可以是dfs\_query\_then\_fetch，query\_then\_fetch，默认query\_then\_fetch |

**POST请求参数搜索**

请求：POST http://localhost:9200/secilog/log/\_search

参数在请求头中。

参数是JSON格式的查询领域语法（query dsl）。例如：

{

    "query" : {

        "term" : { "type" : "syslog" }

    }

}

返回的结果：

{

  "took" : 2,

  "timed\_out" : false,

  "\_shards" : {

    "total" : 1,

    "successful" : 1,

    "failed" : 0

  },

  "hits" : {

    "total" : 1,

    "max\_score" : 1.0,

    "hits" : [ {

      "\_index" : "secilog",

      "\_type" : "log",

      "\_id" : "1",

      "\_score" : 1.0,

      "\_source" : {

        "type" : "syslog",

        "message" : "secilog test test test "

      }

    } ]

  }

}

    如果需要搜索分页，可以通过from size组合来进行。from表示从第几行开始，size表示查询多少条文档。from默认为0，size默认为10，例如：

{

    "from" : 0, "size" : 10,

    "query" : {

        "term" : { "type" : "syslog" }

    }

}

注意：size的大小不能超过index.max\_result\_window这个参数的设置，默认为10,000。

#### 排序sort

    可以通过一个或者多个字段进行排序。例如：

{

"sort" : [

         {"type" :{"order": "asc"}},

         {"message" :{"order": "asc"}},

    ]

}

返回的结果为：

{

  "took" : 2,

  "timed\_out" : false,

  "\_shards" : {

    "total" : 1,

    "successful" : 1,

    "failed" : 0

  },

  "hits" : {

    "total" : 2,

    "max\_score" : null,

    "hits" : [ {

      "\_index" : "secilog",

      "\_type" : "log",

      "\_id" : "2",

      "\_score" : null,

      "\_source" : {

        "type" : "file",

        "message" : "Another secilog test "

      },

      "sort" : [ "file", "test" ]

    }, {

      "\_index" : "secilog",

      "\_type" : "log",

      "\_id" : "1",

      "\_score" : null,

      "\_source" : {

        "type" : "syslog",

        "message" : "secilog test test test "

      },

      "sort" : [ "syslog", "test" ]

    } ]

  }

}

    当一个字段的内容有多个值的时候，系统支持一些计算进行排序，包括min，max，sum，avg，median(中间值)，例如下面的请求表示order有多个值，取平均值排序：

{

   "query" : {

    ...

   },

   "sort" : [

      {"price" : {"order" : "asc", "mode" : "avg"}}

   ]}

#### 数据列过滤

    数据列过滤允许在查询的时候不显示原始数据或者显示部分原始字段。例如不显示原始文档：

{

    "\_source": false,

    "query" : {

        "term" : { "type" : "syslog" }

    }}

例如显示部分文档列：

{

    "\_source": "obj.\*",

    "query" : {

        "term" : { "type" : "syslog" }

    }

}

{

    "\_source": [ "obj1.\*", "obj2.\*" ],

    "query" : {

        "term" : { "type" : "syslog" }

    }

}

例如可以包含或者排除某些列：

{

    "\_source": {

        "include": [ "obj1.\*", "obj2.\*" ],

        "exclude": [ "\*.description" ]

    },

    "query" : {

         "term" : { "type" : "syslog" }

    }

}

#### 脚本支持

同样对搜索也支持脚本，例如：

请求：POST http://localhost:9200/secilog/log/\_search?pretty

参数：

{

    "query": {

        "term": {

            "type": "syslog"

        }

    },

    "script\_fields": {

        "test1": {

            "script": "doc['type'].value \* 2"

        }

    }

}

得到返回结果：

{

  "took" : 16,

  "timed\_out" : false,

  "\_shards" : {

    "total" : 1,

    "successful" : 1,

    "failed" : 0

  },

  "hits" : {

    "total" : 1,

    "max\_score" : 1.0,

    "hits" : [ {

      "\_index" : "secilog",

      "\_type" : "log",

      "\_id" : "1",

      "\_score" : 1.0,

      "fields" : {

        "test1" : [ "syslogsyslog" ]

      }

    } ]

  }

}

## 评分

*短语 查询比简单的match查询在需要消耗更多的资源，因此可以用从新评分的方式进行减少资源的消耗。*

    在Elasticsearch中，搜索单个单词是比较快的，当搜索短语的时候，效率会比较低。所以Elasticsearch提供了从新评分的方法来提高效率。他的原理主要是当在整个索引中搜索短语消耗的资源会比较多，但大多数时候，人们只关注最近发生一部分文档，所以可以先在最近的一段文档中先对短语进行从新评分，然后再查询，这个时候看起来效率会增加很多。

    在返回搜索请求节点处理排序结果之前在每个分片上执行从新评分。通常情况下当我们用从新评分API来重新评分的过程只执行一次，在未来有可能会被调整。当search\_type为 scan或者count的时候将不被执行。在用分页查询的时候，每一页查询的时候window\_size尽量不要修改，如果修改了可能会引起排序的混乱，导致不可预知的结果。查询评分时在query或者post\_filter返回的top-k结果，在每个分片返回文档的数量有window\_size控制，默认返回from到size的个数。

    默认情况下对每个文档最终的得分(\_score) 是原始的得分和从新评分后的得分进行线性组合后的结果。原始评分和从新评分的比例关系分布由query\_weight 和rescore\_query\_weigh控制，默认都是1。例如从新评分查询：

请求：POST localhost:9200/\_search

参数：

{

   "query" : {

      "match" : {

         "field1" : {

            "operator" : "or",

            "query" : "the quick brown",

            "type" : "boolean"

         }

      }

   },

   "rescore" : {

      "window\_size" : 50,

      "query" : {

         "rescore\_query" : {

            "match" : {

               "field1" : {

                  "query" : "the quick brown",

                  "type" : "phrase",

                  "slop" : 2

               }

            }

         },

         "query\_weight" : 0.7,

         "rescore\_query\_weight" : 1.2

      }

   }

}

分数相结合的方式可以与score\_mode控制：score\_mode的参数有：

total：将原始分数和从新评分的分值相加，默认方式。

multiply：将原始分数和从新评分的分值相乘，对于用函数方式从新评分的时候比较有用。

avg：将原始分数和从新评分的分值进行平均。

max：在原始分数和从新评分的分值中取最大的值。

min：在原始分数和从新评分的分值中取最小的值。

系统也支持顺序执行多个从新评分查询,例如：

{

   "query" : {

      "match" : {

         "field1" : {

            "operator" : "or",

            "query" : "the quick brown",

            "type" : "boolean"

         }

      }

   },

   "rescore" : [ {

      "window\_size" : 100,

      "query" : {

         "rescore\_query" : {

            "match" : {

               "field1" : {

                  "query" : "the quick brown",

                  "type" : "phrase",

                  "slop" : 2

               }

            }

         },

         "query\_weight" : 0.7,

         "rescore\_query\_weight" : 1.2

      }

   }, {

      "window\_size" : 10,

      "query" : {

         "score\_mode": "multiply",

         "rescore\_query" : {

            "function\_score" : {

               "script\_score": {

                  "script": "log10(doc['numeric'].value + 2)"

               }

            }

         }

      }

   } ]

}

    先是第一个得到评分的结果，然后第二个评分在第一个评分的结果在此评分，第二次评分参考了第一次评分的排序结果，所以它可以用在第一从新评分结果比较多的情况下利用第二次评分得到一个较小结果文档。

## 高亮显示

*Elasticsearch中的高亮显示是来源于lucene的功能，他允许在一个或者多个字段上突出显示搜索内容， lucene支持三种高亮显示方式highlighter, fast-vector-highlighter， postings-highlighter.*

    Elasticsearch中的高亮显示是来源于lucene的功能，他允许在一个或者多个字段上突出显示搜索内容， lucene支持三种高亮显示方式highlighter, fast-vector-highlighter， postings-highlighter，第一种是默认的标准类型。下面先看一个实例，在搜索前，先增加一条文档。

请求：PUT http://localhost:9200/secilog/log/10?pretty

参数：

{

"type":"file",

"message":"secilog is a log real-time analyse software,it's full text search is based on Elasticsearch "

}

文档创建好后，我们在进行高亮搜索：

请求：POST http://localhost:9200/secilog/log/\_search?pretty

参数：

{

    "query": {

        "term": {

            "message": "analyse"

        }

    },

    "highlight": {

        "fields": {

            "message": { }

        }

    }

}

返回结果如下：

{

  "took" : 1,

  "timed\_out" : false,

  "\_shards" : {

    "total" : 1,

    "successful" : 1,

    "failed" : 0

  },

  "hits" : {

    "total" : 1,

    "max\_score" : 0.4232868,

    "hits" : [ {

      "\_index" : "secilog",

      "\_type" : "log",

      "\_id" : "10",

      "\_score" : 0.4232868,

      "\_source" : {

        "type" : "file",

        "message" : "secilog is a log real-time analyse software,it's full text search is based on Elasticsearch "

      },

      "highlight" : {

        "message" : [ "secilog is a log real-time <em>analyse</em> software,it's full text search is based on Elasticsearch " ]

      }

    } ]

  }

}

    从结果中可以看出，有高亮显示的内容，<em>analyse</em>。为了执行高亮显示，该字段必须有实际的内容。并且这个字段必须进程存储，就是在字段映射中store的值必须为ture，不能只在内存中。否则系统会自动加载\_source字段并匹配相关的列。字段名称支持通配符符号，例如可以用"message\*": { }参数匹配所有message开头的字段。

#### fast-vector-highlighter

    前面的高亮显示是普通的高亮显示，lucene还支持fast-vector-highlighter高亮显示，fast-vector-highlighter高亮显示具有如下特点：

* 快，特别是内容别大的字段，比如大于1M。
* 可定制的boundary\_chars，boundary\_max\_scan，和fragment\_offset。
* 需要设置term\_vector的值为with\_positions\_offsets，增加索引的大小。
* 可以将多个字段的匹配组合成一个结果。
* 可以分配不同的权重匹配在不同的位置上，

    Elasticsearch中需要在建立索引的时候映射字段类型，才可以实现postings-highlighter高亮显示，例如对content字段采用fast-vector高亮类型：

{

    "type\_name" : {

        "content" : {"type":"string","term\_vector" : "with\_positions\_offsets"}

    }

}

#### postings-highlighter

    lucene还支持postings-highlighter高亮显示，postings-highlighter高亮显示具有如下特点：

* 快，因为它不需要重新分析文档：尤其是对大文件对性能的提高更为明显。
* 占用更少的磁盘空间。
* 把高亮显示和句子分开，这个更有利于人类的阅读。
* 使用BM25算法，使搜索的时候像是整篇文档。

    Elasticsearch中需要在建立索引的时候映射字段类型，才可以实现postings-highlighter高亮显示，例如对content字段采用postings高亮类型：

{

    "type\_name" : {

        "content" : {"type":"string","index\_options" : "offsets"}

    }

}

备注：高亮查询不支持复杂的查询，比如查询类型设置为match\_phrase\_prefix的查询。

    对于后两种特殊的类型，都会增加索引的大小，但对于高亮显示查询的执行时间会减少。

    使用type字段可以强制使用特定的高亮类型，当设置了term\_vectors高亮类型的时候，想用普通高亮显示的时候非常有用。这个只有三中类型，plain, postings ， fvh分别对应高亮显示的三种类型，例如：

{

    "query" : {...},

    "highlight" : {

        "fields" : {

            "content" : {"type" : "plain"}

        }

    }}

#### 默认高亮显示html标记

    默认情况下，高亮显示的文本在<em>和</em>中。这可以通过设置pre\_tags和post\_tags进行修改，例如：

{

    "query" : {...},

    "highlight" : {

        "pre\_tags" : ["<b>"],

        "post\_tags" : ["</b>"],

        "fields" : {

            "\_all" : {}

        }

    }}

    使用快速矢量记号可以有多个标签，按照“重要性”进行排序，例如：

{

    "query" : {...},

    "highlight" : {

        "pre\_tags" : ["<tag1>", "<tag2>"],

        "post\_tags" : ["</tag1>", "</tag2>"],

        "fields" : {

            "\_all" : {}

        }

    }

}

    系统对于这种情况有默认的多个pre\_tags，需要设置tags\_schema为styled，默认post\_tags为</em>，默认多个pre\_tags标签为：

<em class="hlt1">, <em class="hlt2">, <em class="hlt3">,<em class="hlt4">, <em class="hlt5">, <em class="hlt6">,<em class="hlt7">, <em class="hlt8">, <em class="hlt9">,<em class="hlt10">

   当我们需要设置默认多个标签的时候的例子如下：

{

    "query" : {...},

    "highlight" : {

        "tags\_schema" : "styled",

        "fields" : {

            "content" : {}

        }

    }

}

    每个字段都可以设置高亮显示的字符片fragment\_size段大小（默认为100），以及返回的最大片段数number\_of\_fragments（默认为5），如果number\_of\_fragments值设置为0则片段产生，当order设置为score时候可以按照评分进行排序。例如：

{

    "query" : {...},

    "highlight" : {

        "order" : "score",

        "fields" : {

            "content" : {"fragment\_size" : 150, "number\_of\_fragments" : 3}

        }

    }

}

## 滚动查询请求

<https://my.oschina.net/secisland/blog/615338>

## 隐藏内容查询

*在Elasticsearch中有嵌套结构和父子结构，在这两种结构中允许通过不同的范围查询搜索到不同的文档。在父/子情况下，可以通过查询子文档的内容返回父文档或者通过父文档查询返回子文档；在嵌套文档情况下，可以通过嵌套内部的对象查询得到嵌套文档。*

    在Elasticsearch中有嵌套结构和父子结构，在这两种结构中允许通过不同的范围查询搜索到不同的文档。在父/子情况下，可以通过查询子文档的内容返回父文档或者通过父文档查询返回子文档；在嵌套文档情况下，可以通过嵌套内部的对象查询得到嵌套文档。

    在这两种情况下，返回的实际匹配的内容是隐藏的。但在有些情况下，了解实际匹配的内容是非常有用的。在这种情况下就要用到 inner hits查询。

请求：PUT http://127.0.0.1:9200/secilog

{

  "mappings": {

    "weblog": {

      "properties": {

        "message": {

          "type": "string",

          "index": "analyzed"

        },

        "apache": {

          "type": "nested",

          "properties": {

            "method": {

              "type": "string",

              "index": "not\_analyzed"

            },

            "status": {

              "type": "string",

              "index": "not\_analyzed"

            }

          }

        }

      }

    }

  }

}

然后我们插入一条记录：

请求：POST  http://127.0.0.1:9200/secilog/weblog/1

参数：

{

    "message": "58.240.83.67 - - [16/Feb/2016:12:58:47 +0800] \"POST /copyright/check HTTP/1.1\" 200 2",

    "apache": [

        {

            "method": "POST",

            "status": "200"

        }

    ]

}

我们通过嵌套查询来查询里面的文档：

请求：POST  http://127.0.0.1:9200/secilog/weblog/\_search/

参数

{

  "query": {

    "nested": {

      "path": "apache",

      "query": {

        "bool": {

          "must": [

            {

              "term": {

                "apache.method": "POST"

              }

            }

          ]

        }

      }

    }

  }

}

返回结果：

{

    "took": 11,

    "timed\_out": false,

    "\_shards": {

        "total": 5,

        "successful": 5,

        "failed": 0

    },

    "hits": {

        "total": 1,

        "max\_score": 1,

        "hits": [

            {

                "\_index": "secilog",

                "\_type": "weblog",

                "\_id": "1",

                "\_score": 1,

                "\_source": {

                    "message": "58.240.83.67 - - [16/Feb/2016:12:58:47 +0800] \"POST /copyright/check HTTP/1.1\" 200 2",

                    "apache": [

                        {

                            "method": "POST",

                            "status": "200"

                        }

                    ]

                }

            }

        ]

    }}

当我们加了参数inner\_hits请求后：

{

  "query": {

    "nested": {

      "path": "apache",

      "query": {

        "bool": {

          "must": [

            {

              "term": {

                "apache.method": "POST"

              }

            }

          ]

        }

      },

      "inner\_hits": {

        "from": 0,

        "size": 10

      }

    }

  }

}

返回结果如下:

{

    "took": 13,

    "timed\_out": false,

    "\_shards": {

        "total": 5,

        "successful": 5,

        "failed": 0

    },

    "hits": {

        "total": 1,

        "max\_score": 1,

        "hits": [

            {

                "\_index": "secilog",

                "\_type": "weblog",

                "\_id": "1",

                "\_score": 1,

                "\_source": {

                    "message": "58.240.83.67 - - [16/Feb/2016:12:58:47 +0800] \"POST /copyright/check HTTP/1.1\" 200 2",

                    "apache": [

                        {

                            "method": "POST",

                            "status": "200"

                        }

                    ]

                },

                "inner\_hits": {

                    "apache": {

                        "hits": {

                            "total": 1,

                            "max\_score": 1,

                            "hits": [

                                {

                                    "\_index": "secilog",

                                    "\_type": "weblog",

                                    "\_id": "1",

                                    "\_nested": {

                                        "field": "apache",

                                        "offset": 0

                                    },

                                    "\_score": 1,

                                    "\_source": {

                                        "method": "POST",

                                        "status": "200"

                                    }

                                }

                            ]

                        }

                    }

                }

            }

        ]

    }

}

    从返回结果中可以看出，之前隐藏的信息都显示出来了。\_nested关键字是在上面的例子中是至关重要的，因为它定义了内部嵌套的对象查询从哪里获取。

inner\_hits支持的参数：

from：搜索命中的开始位置；

size：搜索的文档的条数；

sort：排序字段；

name：在响应中用于特殊内部命中定义的名称，在一个搜索中定义了多个内部隐藏查询(inner hits)的时候比较有用.

## 搜索相关函数

*Elasticsearch提供了很多使用的函数来更加方便的满足不用的场景需求。*

    Elasticsearch提供了很多使用的函数来更加方便的满足不用的场景需求。

    preference，搜索分片副本执行偏好设置。默认情况下，使用随机副本方法搜索副本的分片。可以通过preference参数设置搜索分片的范围。例如：

请求：POST http://127.0.0.1:9200/secilog/\_search?\_primary

参数：

{

  "query": {

    "match": {

      "message": "POST "

    }

  }

}

    这个值也可以是一个自定义的值，它将被用来保证同样的分片将使用相同的自定义值，这个值可以是网络会话标识，或用户名称。例如：请求：POST http://127.0.0.1:9200/secilog/\_search?preference=xyzabc123。

    版本号version：通过"version": true可以对每个返回的之中有版本号。

{

    "version": true,

    "query" : {

        "term" : { "message": "POST" }

    }

}

    索引加速Index Boost ：当搜索一个以上的索引的时候，可以对每个索引配置不同的索引加速级别。当有多个索引存储类似的文档的时候，一个索引的命中级别高于另一个索引的时候非常有用，例如：

{

    "indices\_boost" : {

        "index1" : 1.4,

        "index2" : 1.3

    }

}

    最小分值min\_score：可以指定搜索时候的最小评分的分值。例如：

{

    "min\_score": 0.5,

    "query" : {

        "term" : { "message": "POST" }

    }

}

    分值解释explain：可以使每个命中的查询解释它的得分是如何计算出来的。

{

  "explain": true,

  "query": {

    "term": {

      "message": "copyright"

    }

  }

}

    分片情况查询，例如：

请求：GET http://127.0.0.1:9200/secilog/\_search\_shards/

    总数查询：例如：

请求：POST http://127.0.0.1:9200/secilog/\_count/

参数：

{

  "explain": true,

  "query": {

    "term": {

      "message": "copyright"

    }

  }

}

返回结果：

{

    "count": 1,

    "\_shards": {

        "total": 5,

        "successful": 5,

        "failed": 0

    }

}

    是否存在查询：

请求：POST http://127.0.0.1:9200/customer/\_search/

参数：

{

  "query": {

    "term": {

      "state": "close"

    }

  },

  "size": 0,

  "terminate\_after": 1

}

返回的结果：

{

    "took": 2,

    "timed\_out": false,

    "terminated\_early": true,

    "\_shards": {

        "total": 5,

        "successful": 5,

        "failed": 0

    },

    "hits": {

        "total": 2,

        "max\_score": 0,

        "hits": [ ]

    }

}

    验证接口：验证接口允许用户验证一个查询的语法是否正确，这样可以不用实际执行，可以提高效率，防止误操作。例如：

请求：POST http://127.0.0.1:9200/customer/\_validate/query/

参数：

{

  "query": {

    "term": {

      "state": "close"

    }

  }

}

返回结果：

{

    "valid": true,

    "\_shards": {

        "total": 1,

        "successful": 1,

        "failed": 0

    }

}

    列字段状态查询：

请求：GET http://127.0.0.1:9200/customer/\_field\_stats/

结果：

{

    "\_shards": {

        "total": 5,

        "successful": 5,

        "failed": 0

    },

    "indices": {

        "\_all": {

            "fields": {

                "state": {

                    "max\_doc": 5,

                    "doc\_count": 5,

                    "density": 100,

                    "sum\_doc\_freq": 5,

                    "sum\_total\_term\_freq": 5,

                    "min\_value": "close",

                    "max\_value": "open"

                }

            }

        }

    }

}

## 集群相关接口

*介绍了集群相关的内容，包括集群任务，集群分片操作，集群配置更新，节点统计，节点信息，活跃线程查看*

#### 集群健康检查

    群集健康检查API可以得到集群健康的简单情况。

请求：GET http://127.0.0.1:9200/\_cluster/health?pretty=true

返回:

{

  "cluster\_name" : "elasticsearch",

  "status" : "yellow",

  "timed\_out" : false,

  "number\_of\_nodes" : 1,

  "number\_of\_data\_nodes" : 1,

  "active\_primary\_shards" : 16,

  "active\_shards" : 16,

  "relocating\_shards" : 0,

  "initializing\_shards" : 0,

  "unassigned\_shards" : 15,

  "delayed\_unassigned\_shards" : 0,

  "number\_of\_pending\_tasks" : 0,

  "number\_of\_in\_flight\_fetch" : 0,

  "task\_max\_waiting\_in\_queue\_millis" : 0,

  "active\_shards\_percent\_as\_number" : 51.61290322580645

}

如果请求的后面加上索引的名称，则可以得到这个索引的健康检查情况。例如：

请求：GET http://127.0.0.1:9200/\_cluster/health/secisland?pretty=true

status字段提供的值反应了集群整体的健康程度，它的状态是由系统中最差的分片决定的。它的值的意义如下：

green：所有的主分片(Primary Shard)和副本分片(Replica Shard)都处于活动状态

yellow：所有的主分片都处于活动状态，但是并不是所有的副本分片都处于活跃状态

red：不是所有的主分片都处于活动状态。

#### 群集状态的接口

    可以得到整个群集的综合状态信息。

请求：GET 'http://127.0.0.1:9200/\_cluster/state

默认情况下，群集状态请求的是主节点的状态。

其他的几个例子：

$ curl -XGET 'http://127.0.0.1:9200/\_cluster/state/metadata,routing\_table/foo,bar'

$ curl -XGET 'http://127.0.0.1:9200/\_cluster/state/\_all/foo,bar'

$ curl -XGET 'http://127.0.0.1:9200/\_cluster/state/blocks'

#### 群集统计

    群集统计API可以从一个群集的角度来统计数据。

请求：GET http://127.0.0.1:9200/\_cluster/stats/

#### 集群任务

    任务API返回一个集群尚未执行列表，这通常为空，因为系统正常情况下处理是很快的。

请求：http://127.0.0.1:9200/\_cluster/pending\_tasks

#### 集群分片操作

   在Elasticsearch中可以通过集群路由API来对集群中的分片进行操作，例如可以在集群中把一个分片从一个节点迁移到另一个节点，未分配的分片可以被分配到一个特定节点上等。不过要想完全手动，必须先把cluster.routing.allocation.disable\_allocation参数设置为true，禁止es进行自动索引分片分配，否则你从一节点把分片移到另外一个节点，那么另外一个节点的一个分片又会移到那个节点。 语法如下：

请求：POST 127.0.0.1:9200/\_cluster/reroute

参数：

{

    "commands" : [ {

        "move" :

            {

              "index" : "test", "shard" : 0,

              "from\_node" : "node1", "to\_node" : "node2"

            }

        },

        {

          "allocate" : {

              "index" : "test", "shard" : 1, "node" : "node3"

          }

        }

    ]

}

一共有三种操作，分别为：移动（move），取消（cancel）和分配（allocate）。下面分别介绍这三种情况：

**移动（move）**

把分片从一节点移动到另一个节点。可以指定索引名和分片号。

**取消（cancel）**

取消分配一个分片。可以指定索引名和分片号。node参数可以指定在那个节点取消正在分配的分片。allow\_primary参数支持取消分配主分片。

**分配（allocate）**

分配一个未分配的分片到指定节点。可以指定索引名和分片号。node参数指定分配到那个节点。allow\_primary参数可以强制分配主分片，不过这样可能导致数据丢失。allow\_primary参数将使一个新的空主分片被分配。如果一个节点有一份原始的分片(包括数据)，后面如果重新加入集群后，数据将被删除。以前分片的副本将会被新的分片的副本替换。

本文由赛克 蓝德(secisland)原创，转载请标明作者和出处。

#### 集群配置更新

    系统允许对群集的配置进行更新。配置更新有两种状态，一是持久的，二是临时的，例如：

请求：PUT 127.0.0.1:9200/\_cluster/settings

参数：

{

    "persistent" : {

        "discovery.zen.minimum\_master\_nodes" : 2

    }

}

上面的例子是持久更新配置，如果把参数persistent换成了transient，就是临时更新。

#### 节点统计

    群集节点统计API可以对一个或多个（或全部）的群集节点进行统计。例如：

请求：GET　http://127.0.0.1:9200/\_nodes/stats/

请求：GET　http://127.0.0.1:9200/\_nodes/Artemis/stats/

Artemis是节点名称，多个节点可以用逗号分开。这个命令把所有的统计都返回了，可以在最后面加上指标，只返回你关注指标的统计，多个指标可以用都好分开，例如：

请求：GEThttp://127.0.0.1:9200/\_nodes/Artemis/stats/os,fs

上面的命令只返回节点中操作系统的信息，统计内容解释：

|  |  |
| --- | --- |
| 指标 | 解释 |
| indices | 索引相关的数据统计，包括size, document count, indexing and deletion times, search times, field cache size, merges and flushes。 |
| fs | 文件系统信息，包括路径，磁盘空间，读/写数据等 |
| http | HTTP连接信息 |
| jvm | JVM虚拟机的统计，包括内存池的数据，收集垃圾，缓冲池，一些加载/卸载类等 |
| os | 操作系统统计，平均负载、内存、交换分区等 |
| process | 进程统计，包括内存消耗，处理器使用，打开文件描述符等 |
| thread\_pool | 线程池的统计，包括当前大小、队列和拒绝的任务等 |
| transport | 网络数据统计，集群通信中发送和接收字节的统计 |
| breaker | 数据处理的统计 |

#### 节点信息

    群集节点信息接口允许检索一个或多个群集节点信息（或全部）。节点信息和节点统计信息的语法基本一样，只是没有stats字段，从指标上来看也有一定的区别。例如：

请求：GEThttp://127.0.0.1:9200/\_nodes

从指标上来看有settings, os, process, jvm, thread\_pool, transport, http和plugins。

和节点统计类似，也可以在最后面加上指标信息，对关注的指标单独查看。

#### 活跃线程查看

    系统支持在索引的每个节点上查看活跃线程的信息。

请求：GET http://127.0.0.1:9200/\_nodes/hot\_threads

或者请求具体某个节点：GET \_nodes/{nodesIds}/hot\_threads

返回的内容：

::: {Artemis}{jevOQqVQT*\_a\_*pAGqKA0p7w}{127.0.0.1}{127.0.0.1:9300}

   Hot threads at 2016-02-18T02:31:33.163Z, interval=500ms, busiestThreads=3, ignoreIdleThreads=true:

    6.3% (31.2ms out of 500ms) cpu usage by thread 'elasticsearch[Artemis][transport\_client\_timer][T#1]{Hashed wheel timer #1}'

     10/10 snapshots sharing following 5 elements

       java.lang.Thread.sleep(Native Method)

       org.jboss.netty.util.HashedWheelTimer$Worker.waitForNextTick(HashedWheelTimer.java:445)

       org.jboss.netty.util.HashedWheelTimer$Worker.run(HashedWheelTimer.java:364)

       org.jboss.netty.util.ThreadRenamingRunnable.run(ThreadRenamingRunnable.java:108)

       java.lang.Thread.run(Thread.java:745)

## 分词篇（中文分词！！）

*摘要: 在Elasticsearch中，内置了很多分词器（analyzers），但默认的分词器对中文的支持都不是太好。所以需要单独安装插件来支持，比较常用的是中科院 ICTCLAS的smartcn和IKAnanlyzer效果还是不错的，但是smartcn不支持自定义词库，所以优先用IKAnanlyzer。*

    在Elasticsearch中，内置了很多分词器（analyzers），但默认的分词器对中文的支持都不是太好。所以需要单独安装插件来支持，比较常用的是中科院 ICTCLAS的smartcn和IKAnanlyzer效果还是不错的，但是目前IKAnanlyzer还不支持最新的Elasticsearch2.2.0版本，但是smartcn中文分词器默认官方支持，它提供了一个中文或混合中文英文文本的分析器。支持最新的2.2.0版本版本。但是smartcn不支持自定义词库，作为测试可先用一下。后面的部分介绍如何支持最新的版本。

#### smartcn

安装分词：plugin install analysis-smartcn

卸载：plugin remove analysis-smartcn

测试：

请求：POST http://127.0.0.1:9200/\_analyze/

{

  "analyzer": "smartcn",

  "text": "联想是全球最大的笔记本厂商"

}

返回结果：

{

    "tokens": [

        {

            "token": "联想",

            "start\_offset": 0,

            "end\_offset": 2,

            "type": "word",

            "position": 0

        },

        {

            "token": "是",

            "start\_offset": 2,

            "end\_offset": 3,

            "type": "word",

            "position": 1

        },

        {

            "token": "全球",

            "start\_offset": 3,

            "end\_offset": 5,

            "type": "word",

            "position": 2

        },

        {

            "token": "最",

            "start\_offset": 5,

            "end\_offset": 6,

            "type": "word",

            "position": 3

        },

        {

            "token": "大",

            "start\_offset": 6,

            "end\_offset": 7,

            "type": "word",

            "position": 4

        },

        {

            "token": "的",

            "start\_offset": 7,

            "end\_offset": 8,

            "type": "word",

            "position": 5

        },

        {

            "token": "笔记本",

            "start\_offset": 8,

            "end\_offset": 11,

            "type": "word",

            "position": 6

        },

        {

            "token": "厂商",

            "start\_offset": 11,

            "end\_offset": 13,

            "type": "word",

            "position": 7

        }

    ]

}

作为对比，我们看一下标准的分词的结果，在请求中巴smartcn，换成standard

然后看返回结果：

{

    "tokens": [

        {

            "token": "联",

            "start\_offset": 0,

            "end\_offset": 1,

            "type": "<IDEOGRAPHIC>",

            "position": 0

        },

        {

            "token": "想",

            "start\_offset": 1,

            "end\_offset": 2,

            "type": "<IDEOGRAPHIC>",

            "position": 1

        },

        {

            "token": "是",

            "start\_offset": 2,

            "end\_offset": 3,

            "type": "<IDEOGRAPHIC>",

            "position": 2

        },

        {

            "token": "全",

            "start\_offset": 3,

            "end\_offset": 4,

            "type": "<IDEOGRAPHIC>",

            "position": 3

        },

        {

            "token": "球",

            "start\_offset": 4,

            "end\_offset": 5,

            "type": "<IDEOGRAPHIC>",

            "position": 4

        },

        {

            "token": "最",

            "start\_offset": 5,

            "end\_offset": 6,

            "type": "<IDEOGRAPHIC>",

            "position": 5

        },

        {

            "token": "大",

            "start\_offset": 6,

            "end\_offset": 7,

            "type": "<IDEOGRAPHIC>",

            "position": 6

        },

        {

            "token": "的",

            "start\_offset": 7,

            "end\_offset": 8,

            "type": "<IDEOGRAPHIC>",

            "position": 7

        },

        {

            "token": "笔",

            "start\_offset": 8,

            "end\_offset": 9,

            "type": "<IDEOGRAPHIC>",

            "position": 8

        },

        {

            "token": "记",

            "start\_offset": 9,

            "end\_offset": 10,

            "type": "<IDEOGRAPHIC>",

            "position": 9

        },

        {

            "token": "本",

            "start\_offset": 10,

            "end\_offset": 11,

            "type": "<IDEOGRAPHIC>",

            "position": 10

        },

        {

            "token": "厂",

            "start\_offset": 11,

            "end\_offset": 12,

            "type": "<IDEOGRAPHIC>",

            "position": 11

        },

        {

            "token": "商",

            "start\_offset": 12,

            "end\_offset": 13,

            "type": "<IDEOGRAPHIC>",

            "position": 12

        }

    ]

}

    从中可以看出，基本上不能使用，就是一个汉字变成了一个词了。

本文由赛克 蓝德(secisland)原创，转载请标明作者和出处。

#### IKAnanlyzer支持2.2.0版本

    目前github上最新的版本只支持Elasticsearch2.1.1,路径为https://github.com/medcl/elasticsearch-analysis-ik。但现在最新的Elasticsearch已经到2.2.0了所以要经过处理一下才能支持。

1、下载源码，下载完后解压到任意目录，然后修改elasticsearch-analysis-ik-master目录下的pom.xml文件。找到<elasticsearch.version>行，然后把后面的版本号修改成2.2.0。

2、编译代码mvn package。

3、编译完成后会在target\releases生成elasticsearch-analysis-ik-1.7.0.zip文件。

4、解压文件到Elasticsearch/plugins目录下。

5、修改配置文件增加一行：index.analysis.analyzer.ik.type : "ik"

6、重启Elasticsearch。

测试：和上面的请求一样，只是把分词替换成ik

返回的结果：

{

    "tokens": [

        {

            "token": "联想",

            "start\_offset": 0,

            "end\_offset": 2,

            "type": "CN\_WORD",

            "position": 0

        },

        {

            "token": "全球",

            "start\_offset": 3,

            "end\_offset": 5,

            "type": "CN\_WORD",

            "position": 1

        },

        {

            "token": "最大",

            "start\_offset": 5,

            "end\_offset": 7,

            "type": "CN\_WORD",

            "position": 2

        },

        {

            "token": "笔记本",

            "start\_offset": 8,

            "end\_offset": 11,

            "type": "CN\_WORD",

            "position": 3

        },

        {

            "token": "笔记",

            "start\_offset": 8,

            "end\_offset": 10,

            "type": "CN\_WORD",

            "position": 4

        },

        {

            "token": "笔",

            "start\_offset": 8,

            "end\_offset": 9,

            "type": "CN\_WORD",

            "position": 5

        },

        {

            "token": "记",

            "start\_offset": 9,

            "end\_offset": 10,

            "type": "CN\_CHAR",

            "position": 6

        },

        {

            "token": "本厂",

            "start\_offset": 10,

            "end\_offset": 12,

            "type": "CN\_WORD",

            "position": 7

        },

        {

            "token": "厂商",

            "start\_offset": 11,

            "end\_offset": 13,

            "type": "CN\_WORD",

            "position": 8

        }

    ]

}

从中可以看出，两个分词器分词的结果还是有区别的。

扩展词库，在config\ik\custom下在mydict.dic中增加需要的词组，然后重启Elasticsearch，需要注意的是文件编码是UTF-8 无BOM格式编码。

比如增加了赛克蓝德单词。然后再次查询：

请求：POST http://127.0.0.1:9200/\_analyze/

参数：

{

  "analyzer": "ik",

  "text": "赛克蓝德是一家数据安全公司"

}

返回结果：

{

    "tokens": [

        {

            "token": "赛克蓝德",

            "start\_offset": 0,

            "end\_offset": 4,

            "type": "CN\_WORD",

            "position": 0

        },

        {

            "token": "克",

            "start\_offset": 1,

            "end\_offset": 2,

            "type": "CN\_WORD",

            "position": 1

        },

        {

            "token": "蓝",

            "start\_offset": 2,

            "end\_offset": 3,

            "type": "CN\_WORD",

            "position": 2

        },

        {

            "token": "德",

            "start\_offset": 3,

            "end\_offset": 4,

            "type": "CN\_CHAR",

            "position": 3

        },

        {

            "token": "一家",

            "start\_offset": 5,

            "end\_offset": 7,

            "type": "CN\_WORD",

            "position": 4

        },

        {

            "token": "一",

            "start\_offset": 5,

            "end\_offset": 6,

            "type": "TYPE\_CNUM",

            "position": 5

        },

        {

            "token": "家",

            "start\_offset": 6,

            "end\_offset": 7,

            "type": "COUNT",

            "position": 6

        },

        {

            "token": "数据",

            "start\_offset": 7,

            "end\_offset": 9,

            "type": "CN\_WORD",

            "position": 7

        },

        {

            "token": "安全",

            "start\_offset": 9,

            "end\_offset": 11,

            "type": "CN\_WORD",

            "position": 8

        },

        {

            "token": "公司",

            "start\_offset": 11,

            "end\_offset": 13,

            "type": "CN\_WORD",

            "position": 9

        }

    ]

}

## 分词篇（分析模块！！）

*在Elasticsearch中，索引分析模块是可以通过注册分词器(Analyzer)来进行配置。分词器的作用是当一个文档被索引的时候，分词器从文档中提取出若干词元(Token)来支持索引的存储和搜索。分词器是由一个分解器(Tokenizer)和零个或多个词元过滤器(token filters)组成。词元过滤器的作用是对分词器提取出来的Token(词元)被进一步处理。*

    在Elasticsearch中，索引分析模块是可以通过注册分词器(Analyzer)来进行配置。分词器的作用是当一个文档被索引的时候，分词器从文档中提取出若干词元(Token)来支持索引的存储和搜索。

    分词器是由一个分解器(Tokenizer)和零个或多个词元过滤器(token filters)组成。分解器处理前可能要做一些预处理， 比如去掉里面的HTML标记， 这些处理的算法被称为字符过滤器(Character Filter)，一个分解器会有一个或多个字符过滤器。分解器是用来分解字符串成一系列术语或词元(Token)。一个简单的分解器是把一个句子当遇到空格或标点符号是进行分解成一个个的索引词。Elasticsearch内置了分解器，同时也支持自定义分解器。

    词元过滤器的作用是对分词器提取出来的Token(词元)被进一步处理， 比如转成小写,增加同义词等。被处理后的结果被称为Term(索引词)， 文档中包含了几个这样的Term被称为Frequency(词频)。 引擎会建立Term和原文档的Inverted Index(倒排索引)， 这样就能根据Term很快到找到源文档了。

    ES内置了很多Analyzer, 还有很多第三方的Analyzer插件， 比如一些处理中文的Analyzer(中文分词)。

    一个索引分析模块由一个分词器(Analyzer)，分解器(Tokenizer)和词元过滤器(token filters)组成。如果没有明确的定义分析模块，系统会用内置的分词器，分解器，和过滤器。下面是一个分析器的配置：

index :

    analysis :

        analyzer :

            standard :

                type : standard

                stopwords : [stop1, stop2]

            myAnalyzer1 :

                type : standard

                stopwords : [stop1, stop2, stop3]

                max\_token\_length : 500

            # configure a custom analyzer which is

            # exactly like the default standard analyzer

            myAnalyzer2 :

                tokenizer : standard

                filter : [standard, lowercase, stop]

        tokenizer :

            myTokenizer1 :

                type : standard

                max\_token\_length : 900

            myTokenizer2 :

                type : keyword

                buffer\_size : 512

        filter :

            myTokenFilter1 :

                type : stop

                stopwords : [stop1, stop2, stop3, stop4]

            myTokenFilter2 :

                type : length

                min : 0

                max : 2000

   当没有明确的定义分词器模块，系统会用内置的分词器，系统有一个配置选项来定义哪个分词器将在默认情况下使用。默认的逻辑名称下有一个的分词器，将用于索引和搜索。配置参数为default\_search。例如：

index :

  analysis :

    analyzer :

      default :

        tokenizer : keyword

    分词器可以通过别名来查找相关的分词器来分词。例如，以下将允许标准分词器也可参照alias1和alias2值。

index :

  analysis :

    analyzer :

      standard :

        alias: [alias1, alias2]

        type : standard

        stopwords : [test1, test2, test3]

    系统默认的分词器有： Standard Analyzer，Simple Analyzer，Whitespace Analyzer，Stop Analyzer，Keyword Analyzer，Pattern Analyzer，Language Analyzers，Snowball Analyzer，Custom Analyzer。

    系统默认的分解器有：Standard Tokenizer，Edge NGram Tokenizer，Keyword Tokenizer，Letter Tokenizer，Lowercase Tokenizer，NGram Tokenizer，Whitespace Tokenizer，Pattern Tokenizer，UAX Email URL Tokenizer，Path Hierarchy Tokenizer，Classic Tokenizer，Thai Tokenizer。

    系统默认的词元过滤器有：Standard Token Filter，ASCII Folding Token Filter，Length Token Filter，Lowercase Token Filter，Uppercase Token Filter，NGram Token Filter，Edge NGram Token Filter，Porter Stem Token Filter，Shingle Token Filter，Stop Token Filter，Word Delimiter Token Filter，Stemmer Token Filter，Stemmer Override Token Filter，Keyword Marker Token Filter，Keyword Repeat Token Filter，KStem Token Filter，Snowball Token Filter，Phonetic Token Filter，Synonym Token Filter，Compound Word Token Filter，Reverse Token Filter，Elision Token Filter，Truncate Token Filter，Unique Token Filter，Pattern Capture Token Filter，Pattern Replace Token Filter，Trim Token Filter，Limit Token Count Token Filter，Hunspell Token Filter，Common Grams Token Filter，Normalization Token Filter，CJK Width Token Filter，CJK Bigram Token Filter，Delimited Payload Token Filter，Keep Words Token Filter，Keep Types Token Filter，Classic Token Filter，Apostrophe Token Filter，Decimal Digit Token Filter。

    系统默认带的字符过滤器有：Mapping Char Filter，HTML Strip Char Filter，Pattern Replace Char Filter。

分析模块的使用可以通过索引分析来进行体现。比如：

请求：POST http://127.0.0.1:9200/\_analyze/

参数：

{

  "tokenizer": "standard",

  "filters": [

    "lowercase",

    "stop"

  ],

  "text": "SecIsland is a data company."

}

注意"analyzer","tokenizer","filters"三个是关键字。analyzer在内容上包括tokenizer和filters，所以可以只输入analyzer，或者输tokenizer和filters。

得到的结果是：

{

    "tokens": [

        {

            "token": "secisland",

            "start\_offset": 0,

            "end\_offset": 9,

            "type": "<ALPHANUM>",

            "position": 0

        },

        {

            "token": "data",

            "start\_offset": 15,

            "end\_offset": 19,

            "type": "<ALPHANUM>",

            "position": 3

        },

        {

            "token": "company",

            "start\_offset": 20,

            "end\_offset": 27,

            "type": "<ALPHANUM>",

            "position": 4

        }

    ]

}

## 精简API（将返回内容变得易读）

**这里是在地址栏输入地址**

*在Elasticsearch提供了一套精简API来查看系统的状态，官方的文档叫cat API。主要的目的是由于Elasticsearch默认提供的接口返回都是JSON格式的，这种格式不利于人类的阅读，所以搞出来一套cat API来简化。*

    在Elasticsearch提供了一套精简API来查看系统的状态，官方的文档叫cat API。主要的目的是由于Elasticsearch默认提供的接口返回都是JSON格式的，这种格式不利于人类的阅读，所以搞出来一套cat API来简化。

    每个命令都是以/\_cat开头，可以接收v参数得到详细输出；可以通过help参数得到返回的每列的帮助信息。可以通h=参数名返回部分内容，多个参数可以用逗号分开，甚至可以支持通配符。例如：

请求：GET http://127.0.0.1:9200/\_cat/master?v

返回：

id                     host      ip        node

jevOQqVQT\_a\_pAGqKA0p7w 127.0.0.1 127.0.0.1 Artemis

查询帮助实例：

请求：GET http://127.0.0.1:9200/\_cat/master?help

id   |   | node id

host | h | host name

ip   |   | ip address

node | n | node name

|  |  |
| --- | --- |
| 指标 | 说明 |
| aliases | 别名显示有关的别名的一些信息，包括过滤和路由等信息，可以在最后面加上具体别名的名称单独查询某个具体的别名。 |
| allocation | 在分配给每个数据节点上提供多少分片快照和他们使用的磁盘空间有多大。 |
| count | 提供节点的文档总数，如果后面加上索引名称，可以的得到具体索引的文档数量。 |
| fielddata | 在集群中的每个数据节点上正在使用的堆内存的大小。 |
| health | 提供集群的健康情况。 |
| indices | 提供节点下的索引信息，可以得到索引的分片，文档，存储大小等信息。 |
| master | 显示master节点的概要信息，主要是id，ip和节点名称。 |
| nodeattrs | 显示自定义节点属性。 |
| nodes | 显示节点相关的信息。 |
| pending tasks | 集群任务接口，和集群接口类似，参见[集群任务](http://my.oschina.net/secisland/blog/616823#OSC_h4_4)文档。 |
| plugins | 提供激情群中插件安装的信息。 |
| recovery | 正在进行和以前完成的索引分片的回收率。 |
| repositories | 显示在群集中注册的快照库。 |
| thread pool | 显示每个群集节点的线程池统计。 |
| shards | 索引分片的使用情况。分片有很多的指标，可以用help查看。 |
| segments | 每个索引分片中低水平段的信息 |
| snapshots | 显示仓库的快照信息。 |

本文由赛克 蓝德(secisland)原创，转载请标明作者和出处。

#### indices

indices有非常多的参数可以选择，具体用help查看。例如：

请求：GET http://127.0.0.1:9200/\_cat/indices?pri&v&h=health,index,prirep,docs.count,mt

返回的结果：

health index     docs.count mt pri.mt

yellow customer           5  0      0

green  .scripts           2  0      0

green  secilog            2  0      0

yellow secisland          4  0      0

#### nodes

nodes有非常多的参数可以选择，具体用help查看。例如：

请求：GET http://127.0.0.1:9200/\_cat/nodes?v&h=id,ip,port,v,m,jdk

返回的结果：

id   ip        port v     m jdk

qKTT 127.0.0.1 9300 2.2.0 \* 1.7.0\_79

#### 回复命令recovery

    在一个集群中，一个索引的分片可能随时会从一个节点迁移到另一个节点，比如在恢复快照的过程中，新节点启动的过程中等等。例如下面的示例表示没有任何分片的数据在传输：

请求：GET http://127.0.0.1:9200/\_cat/recovery/secisland?v

返回的结果：

index     shard time type  stage source\_host target\_host repository snapshot files files\_percent bytes bytes\_percent total\_files total\_bytes translog translog\_percent total\_translog

secisland 0     115  store done  127.0.0.1   127.0.0.1   n/a        n/a      0     100.0%        0     100.0%        1           159         0        -1.0%            -1

secisland 1     15   store done  127.0.0.1   127.0.0.1   n/a        n/a      0     100.0%        0     100.0%        1           159         0        -1.0%            -1

secisland 2     55   store done  127.0.0.1   127.0.0.1   n/a        n/a      0     100.0%        0     100.0%        7           6784        0        -1.0%            -1

secisland 3     51   store done  127.0.0.1   127.0.0.1   n/a        n/a      0     100.0%        0     100.0%        4           3468        0        -1.0%            -1

secisland 4     182  store done  127.0.0.1   127.0.0.1   n/a        n/a      0     100.0%        0     100.0%        4           3468        0        -1.0%            -1

通过增加副本节点的数量可以让分片的数据进行传输。

例如：

index shard time type    stage source target files percent bytes     percent

wiki  0     1252 store done  hostA  hostA  4     100.0%  23638870 100.0%

wiki  0     1672 replica index hostA  hostB  4     75.0%   23638870 48.8%

wiki  1     1698 replica index hostA  hostB  4     75.0%   23348540 49.4%

wiki  1     4812 store done  hostA  hostA  33    100.0%  24501912 100.0%

wiki  2     1689 replica index hostA  hostB  4     75.0%   28681851 40.2%

wiki  2     5317 store done  hostA  hostA  36    100.0%  30267222 100.0%

#### 线程池

线程池有非常多的参数可以选择，具体用help查看。例如：

请求：GET http://127.0.0.1:9200/\_cat/thread\_pool?v&h=id,host,suggest.active,suggest.rejected,suggest.completed

返回的结果：

id   host      suggest.active suggest.rejected suggest.completed

qKTT 127.0.0.1              0                0                 0

## lucene倒排索引

*倒排索引源于实际应用中需要根据属性的值来查找记录。这种索引表中的每一项都包括一个属性值和具有该属性值的各记录的地址。由于不是由记录来确定属性值，而是由属性值来确定记录的位置，因而称为倒排索引(inverted index)。带有倒排索引的文件我们称为倒排索引文件，简称倒排文件(inverted file)。*

1.简介

倒排索引源于实际应用中需要**根据属性的值**来查找记录。这种索引表中的每一项都包括**一个属性值和具有该属性值的各记录的地址**。由于不是由记录来确定属性值，而是由属性值来确定记录的位置，因而称为倒排索引(inverted index)。带有倒排索引的文件我们称为倒排索引文件，简称倒排文件(inverted file)。

倒排文件（倒排索引），索引对象是文档或者文档集合中的**单词**等，用来**存储这些单词在一个文档或者一组文档中的存储位置**，是对文档或者文档集合的一种最常用的索引机制。

**搜索引擎的关键步骤就是建立倒排索引**，倒排索引一般表示为一个关键词，然后是它的频度（出现的次数），位置（出现在哪一篇文章或网页中，及有关的日期，作者等信息），它相当于为互联网上几千亿页网页做了一个索引，好比一本书的目录、标签一般。读者想看哪一个主题相关的章节，直接根据目录即可找到相关的页面。不必再从书的第一页到最后一页，一页一页的查找。

2.Lucene倒排索引原理

Lucerne是一个开放源代码的高性能的java**全文检索引擎工具包**，**不是一个完整的全文检索引擎**，而是一个**全文检索引擎的架构**，提供了**完整的查询引擎和索引引擎**，**部分文本分析引擎**。目的是为软件开发人员提供一个简单易用的工具包，以方便**在目标系统中实现全文检索的功能，或者以此为基础建立起完整的全文检索引擎。**

Lucerne使用的是倒排文件索引结构。该结构及相应的生成算法如下：

设有两篇文章1和2：

文章1的内容为：Tom lives in Guangzhou,I live in Guangzhou too. 　　 文章2的内容为：He once lived in Shanghai.

<1>取得关键词

由于lucene是基于关键词索引和查询的，首先我们要取得这两篇文章的关键词，通常我们需要如下处理措施：

 a.我们现在有的是文章内容，即一个字符串，我们先要找出字符串中的所有单词，即**分词**。英文单词由于用空格分隔，比较好处理。中文单词间是连在一起的需要特殊的分词处理。

b.文章中的”in”, “once” “too”等词没有什么实际意义，中文中的“的”“是”等字通常也无具体含义，这些**不代表概念的词可以过滤掉**

c.用户通常希望查“He”时能把含“he”，“HE”的文章也找出来，所以**所有单词需要统一大小写**。

d.用户通常希望查“live”时能把含“lives”，“lived”的文章也找出来，所以需要把“lives”，“lived”**还原**成“live”

e.文章中的**标点符号**通常不表示某种概念，也可以过滤掉

在lucene中以上措施由Analyzer类完成。 经过上面处理后，

文章1的所有关键词为：[tom] [live] [guangzhou] [i] [live] [guangzhou] 　 　 文章2的所有关键词为：[he] [live] [shanghai]

<2>建立倒排索引

 有了关键词后，我们就可以建立倒排索引了。上面的对应关系是：“文章号”对“文章中所有关键词”。倒排索引把这个关系倒过来，变成: **“关键词”对“拥有该关键词的所有文章号”。**

文章1，2经过倒排后变成

关键词          文章号

guangzhou        1

he               2

i                1

live             1,2

shanghai         2

tom              1

通常仅知道关键词在哪些文章中出现还不够，我们还需要知道**关键词在文章中出现次数和出现的位置**，通常有两种位置：

a.**字符位置**，即记录该词是文章中第几个字符（优点是关键词亮显时定位快）；

b.**关键词位置**，即记录该词是文章中第几个关键词（优点是节约索引空间、词组（phase）查询快），lucene中记录的就是这种位置。

加上“出现频率”和“出现位置”信息后，我们的索引结构变为：

关键词            文章号[出现频率]              出现位置

guangzhou           1[2]                      3，6

he                  2[1]                      1

i                   1[1]                      4

live                1[2]                      2，5,

                    2[1]                      2

shanghai            2[1]                      3

tom                 1[1]                      1

以live 这行为例我们说明一下该结构：live在文章1中出现了2次，文章2中出现了一次，它的出现位置为“2,5,2”这表示什么呢？我们需要结合文章号和出现频率来分析，文章1中出现了2次，那么“2,5”就表示live在文章1中出现的两个位置，文章2中出现了一次，剩下的“2”就表示live是文章2中第 2个关键字。

 以上就是lucene索引结构中最核心的部分。我们注意到**关键字是按字符顺序排列的（lucene没有使用B树结构），因此lucene可以用二元搜索算法快速定位关键词**。

<3>实现

实现时，lucene将上面三列分别作为词典文件（Term Dictionary）、频率文件(frequencies)、位置文件 (positions)保存。其中词典文件不仅保存有每个关键词，还保留了指向频率文件和位置文件的指针，通过指针可以找到该关键字的频率信息和位置信息。

Lucene中使用了field的概念，用于表达信息所在位置（如标题中，文章中，url中），在建索引中，该field信息也记录在词典文件中，每个关键词都有一个field信息(因为每个关键字一定属于一个或多个field)。

<4>压缩算法

为了减小索引文件的大小，Lucene对索引还使用了压缩技术。

首先，对词典文件中的关键词进行了压缩，关键词压缩为<前缀长度，后缀>，例如：当前词为“阿拉伯语”，上一个词为“阿拉伯”，那么“阿拉伯语”压缩为<3，语>。

其次大量用到的是对数字的压缩，数字只保存与上一个值的差值（这样可以减小数字的长度，进而减少保存该数字需要的字节数）。例如当前文章号是16389（不压缩要用3个字节保存），上一文章号是16382，压缩后保存7（只用一个字节）。

<5>应用原因

下面我们可以通过对该索引的查询来解释一下为什么要建立索引。

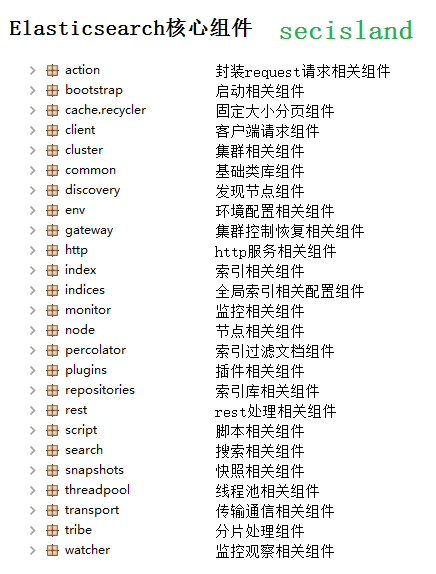
假设要查询单词 “live”，lucene先对词典二元查找、找到该词，通过指向频率文件的指针读出所有文章号，然后返回结果。词典通常非常小，因而，整个过程的时间是毫秒级的。

而用普通的顺序匹配算法，不建索引，而是对所有文章的内容进行字符串匹配，这个过程将会相当缓慢，当文章数目很大时，时间往往是无法忍受的。

## 主要模块和配置介绍

*在Elasticsearch的配置中，主要有两种方式配置，一种是静态配置，另一种是动态配置。静态配置的参数只能在配置文件中事先写好，动态配置的参数可以通过\_cluster/settings进行设置。*

    Elasticsearch中有很多模块，在官方没有找到架构图，所以从代码中整理了一下Elasticsearch的相关组件说明，建下图：



    相关模块配置。

    在Elasticsearch的配置中，主要有两种方式配置，一种是静态配置，另一种是动态配置。静态配置的参数只能在配置文件中事先写好，动态配置的参数可以通过\_cluster/settings进行设置。主要包括的组件配置有：

集群Cluster 平衡配置：

    代表一个集群，集群中有多个节点，其中有一个master节点，master通过选举自动产生，集群控制模块主要分析节点如何进行分片管理。集群平衡在Elasticsearch中有多种策略方式，集群水平分配平衡（Cluster Level Shard Allocation），基于磁盘大小的平衡(Disk-based Shard Allocation)，集群区域平衡（Shard Allocation Awareness），分片过滤配置（Shard Allocation Filtering）。

发现Discovery

    Elasticsearch自动发现节点的机制，ES是一个基于P2P协议的系统，它先通过广播寻找存在的节点，再通过多播协议来进行节点之间的通信，同时也支持点对点的交互。主节点的主要作用是维持全局集群状态和控制节点行为，比如节点加入或离开时集群通过再分配的分片使之达到新的平衡。

Local Gateway

    本地网关是当所有集群从新启动时存储集群状态和分片数据的模块。

HTTP

    设置和控制HTTP REST接口的模块。

Indices

    全局索引相关设置模块。包括内存控制，数据缓存，节点查询缓存，索引缓冲区，碎片请求缓存，恢复，过期文件删除配置。

Network Settings

    网络设置模块。

Node client

    一个java客户端模块，不能保存数据或作为主节点。

Plugins

    插件模块

Scripting

    脚本模块，常用的脚本有Groovy, Python,和Javascript。脚本默认情况下常规使用Groovy，

Snapshot And Restore

    快照备份恢复模块。

Thread Pool

    线程池模块

Transport

    数据传输模块，

Tribe node

    Tribe节点可以加入一个或多个集群并同时连接他们，它的主要特点是可以同时访问多个集群，通过检查每个集群的状态可以形成一个大的系群群。

    一个主节点的主要作用是哪些分片被分配到一个节点。并通过移动节点的分片达到集群的平衡。

index

    索引模块是控制每个索引指标的模块。索引模块包括分词，分片控制和分配，字段映射，索引相似性配置，慢查询记录，文件系统配置和控制事务和刷新模块。

    相似性定义了如何对文件进行评分机制，相似性对应的维度是列，这意味着可以对每一列进行不同相似性的配置。ElasticSearch 允许用户不作任何配置使用default和BM25相似度模型，因为他们是预先在系统中配置好的。如果是DFR和IB模型，我们需要配置才能使用他们。

    索引分片分片配置主要：有分片配置过滤(Shard allocation filtering)，延迟分配(Delayed allocation)和分片总数配置(Total shards per node)。

## 集群配置

*在Elasticsearch中，集群是非常重要的内容，系统支持非常多的集群设置，虽然大多数默认设置也可以很好的工作，但了解更多的集群设置对系统优化有很大的帮助。*

#### 集群分片配置

     在Elasticsearch集群分片有几种方式，集群水平分片分配(Cluster Level Shard Allocation)。集群分片会经常发生变化，然后从新分配，比如当初始恢复，复制，节点加入或者取消的时候等。包括分片分配设置，分片平衡设置，启发式分片平衡配置。

   分片分配设置：下面的动态设置可以用来控制分片的分配和回收。

cluster.routing.allocation.enable：禁用或启用哪种类型的分片，可选的参数有：

|  |  |
| --- | --- |
| all | 允许所有的分片被从新分配 |
| primaries | 只允许主结点分片被从新分配 |
| new\_primaries | 只允许新的主结点索引的分片被从新分配 |
| none | 不对任何分片进行从新分配 |

cluster.routing.allocation.node\_concurrent\_recoveries：允许在一个节点上同时并发多少个分片分配，默认为2。

cluster.routing.allocation.node\_initial\_primaries\_recoveries：当副本分片加入集群的时候，在一个节点上同行发生分片分配的数量，默认是4个。

cluster.routing.allocation.same\_shard.host：在一个主机上的当有多个相同的集群名称的分片分配时，是否进行检查，检查主机名和主机ip地址。默认为false，此设置仅适用于在同一台机器上启动多个节点在时配置。

indices.recovery.concurrent\_streams：从一个节点恢复的时候，同时打开的网络流量的数量，默认为3。

indices.recovery.concurrent\_small\_file\_streams：从同伴的分片恢复时打开每个节点的小文件(小于5M)流的数目，默认为2。

分片平衡设置

    下面的动态设置可以用来控制整个集群的碎片再平衡，配置有：

cluster.routing.rebalance.enable：启用或禁用特定种类的分片重新平衡，可选的参数有：

|  |  |
| --- | --- |
| all | 允许所有的分片进行分片平衡，默认配置。 |
| primaries | 只允许主分片进行平衡。 |
| replicas | 只允许从分片进行平衡。 |
| none | 不允许汾河分片进行平衡。 |

cluster.routing.allocation.allow\_rebalance：当分片的再平衡时允许的操作，可选的参数有：

|  |  |
| --- | --- |
| always | 总是让再平衡。 |
| indices\_primaries\_active | 只有主节点索引允许在平衡。 |
| indices\_all\_active | 所有的分片允许在平衡，默认参数。 |

cluster.routing.allocation.cluster\_concurrent\_rebalance：重新平衡时允许多少个并发的分片同时操作，默认为2。

启发式分片平衡

    以下设置来确定在何处放置每个碎片的数据。

cluster.routing.allocation.balance.shard：在节点上分配每个分片的权重，默认是0.45。

cluster.routing.allocation.balance.index：在特定节点上，每个索引分配的分片的数量，默认0.55。

cluster.routing.allocation.balance.threshold：操作的最小最优化的值。默认为1。

#### 基于磁盘的分片分配配置

    Elasticsearch可以根据磁盘的大小来决定是否从新进行分片的分配。

cluster.routing.allocation.disk.threshold\_enabled：是否启用磁盘分配决策，默认为true。

cluster.routing.allocation.disk.watermark.low：允许分配时的磁盘空间最小值，可以是比例或者绝对值，比如85%或者1G。当磁盘占用超过设定的值之后，系统将不会对此节点进行分配操作。

cluster.routing.allocation.disk.watermark.high：允许保存分片节点磁盘空间的最大值，当超过这个值后，系统会把分片迁移到别的节点。默认90%。也可以设置一个具体的大小值，当空间小于这个值的时候，系统会自动迁移到别的节点。

cluster.info.update.interval：检查集群中的每个节点的磁盘使用情况的时间间隔，默认30秒。

cluster.routing.allocation.disk.include\_relocations：当计算节点的磁盘使用时需要考虑当前被分片的情况。

比如下面的一个配置实例：

PUT /\_cluster/settings{

  "transient": {

    "cluster.routing.allocation.disk.watermark.low": "80%",

    "cluster.routing.allocation.disk.watermark.high": "5gb",

    "cluster.info.update.interval": "1m"

  }

}

    实例的含义是，每分钟检查一下磁盘空间，当磁盘空间小于80%的时候参与分片分配，当空间不足5G的时候，迁移节点的分片到别的节点。

    需要注意的是，在2.0.0版本前，当系统有多个数据盘的时候，系统考虑的是总大小，在2.0.0之后，系统考虑的是每个磁盘的使用情况。

本文由赛克 蓝德(secisland)原创，转载请标明作者和出处。

#### 区域分片分配(Shard Allocation Awareness)

    实际部署的时候，很多时候是部署在虚拟机中，共享同一个物理节点，或者部署的时候在同一个机架或者同一个网络区域中。当这些情况下遇到故障的时候，很多节点会同时发生故障，导致系统出现问题。如果在配置Elasticsearch的时候事先能够注意到吧不同的节点分布在不同的物理机器，不同的机架或者不同的网络区域，这样当一个节点出现问题的时候，会使风险降到最低。

    区域分片分配(Shard Allocation Awareness)设置允许配置Elasticsearch关于硬件的信息，例如，我们在启动的时候在启动程序后面加上--node.rack\_id，后面跟上一个指定的名称，这个配置也可以放在配置文件中：

elasticsearch --node.rack\_id rack\_one

    同时我们需要设置cluster.routing.allocation.awareness.attributes: rack\_id，可以在配置文件中设置，或者通过cluster-update-settings API接口设置。

    假设有两个节点node.rack\_id的名称为rack\_one，我们创建一个索引有5个主要的分片和一个副本分片。所有的主分片和副本分片被分配在两个节点上。然后我们新加两个节点，节点node.rack\_id的名称为rack\_two,这样配置后，系统会自定分配节分片到新的节点上，确保没有两个相同的分片在同一个区域中。

    当搜索的时候，系统会智能的处理，只在一个区域中搜索， 这样会比在不同的区域中搜索更快。

    当使用区域分片分配属性，碎片不会分配给没有设置这些属性的值的节点。在相同属性的上主从分片分配节点的数量是由属性值的数量决定的。当一组节点的数量是不平衡的，可能有许多的副本，复制分片可能会停止。

  强制分配属性，解决了不允许相同的分片副本被分配到同一区域的问题。假设当我们有两个区域的时候，每个区域的大小只够分配一半的分片，如果一个区域不可用，全部分片都在一个区域会导致空间不够，引起系统异常，这个时候，强制分配属性就有意义了。配置：

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

cluster.routing.allocation.awareness.attributes: zone

这个时候，如果我们启动了zone1上的两个节点，并创建有5个分片一个副本的索引，这个时候在zone1上只会启动主分片，只到zone2的节点启动后，才会启动副本分片。

#### 分片配置过滤

    分片配置过滤可以允许或禁止索引的分片分配到特定节点。这个配置主要的作用是当想停止一个节点到集群中的比较有用。例如我们想停止10.0.0.1这个ip上的所有节点：

PUT /\_cluster/settings{

  "transient" : {

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

  }

}

    这样正常情况下，10.0.0.1节点上的分片会被迁移到其他节点。动态配置的属性如下：

cluster.routing.allocation.include.{attribute}：将索引分配给一个节点，其{attribute}至少有一个逗号分隔的值。

cluster.routing.allocation.require.{attribute}：将索引分配给一个节点，该节点的{attribute}具有所有的逗号分隔值。

cluster.routing.allocation.exclude.{attribute}：将索引分配给一个节点，其{attribute}没有一个逗号分隔的值。

attributes 可以包含的值有：

\_name：通过节点名称匹配节点

\_ip：通过IP地址匹配节点

\_host：通过机器名称匹配节点

所有的属性值可以用通配符，如：

PUT \_cluster/settings{

  "transient": {

    "cluster.routing.allocation.include.\_ip": "192.168.2.\*"

  }

}

#### 其他集群设置

只读设置：可以设置cluster.blocks.read\_only使整个群集为只读。

日志记录：设置日志记录的级别，例如增加的indices.recovery模块的日志记录级别的调试：

PUT /\_cluster/settings{

  "transient": {

    "logger.indices.recovery": "DEBUG"

  }

}

## 索引配置

*摘要: 在Elasticsearch中，索引是非常重要的内容，系统支持非常多的索引设置，虽然大多数默认设置也可以很好的工作，但了解更多的索引设置对系统优化有很大的帮助。*

#### 内存控制器

    在Elasticsearch中有很多控制器可以防止内存溢出，每个控制器可以指定内存使用的最大值，除此之外，还有一个总的控制器在确定整个系统使用的最大内存值。这些配置都可以动态更新。总的内存控制有以下参数：

indices.breaker.total.limit：总的内存使用大小，默认为JVM堆内存大小的70%。

列数据内存大小：

列数据内存大小是指，在Elasticsearch系统中，系统会估计有多少数据被加载到内存中，如果估计超过这个阀值，它可以通过一个异常来防止该字段的数据加载。

indices.breaker.fielddata.limit：列数据内存大小的限制，默认为JVM堆内存大小的60%。

indices.breaker.fielddata.overhead：所有列估计的内存大小的乘积，默认是1.03.

请求控制器，防止Elasticsearch每个请求的数据结构()超过一定的值。

indices.breaker.request.limit：请求控制器的大小，默认为JVM堆内存大小的40%。

indices.breaker.request.overhead：所有请求的乘积，默认为1。

#### 数据缓存

    现场数据缓存主要用于当排序或聚合操作的时候。它将所有的字段值加载到内存中以便提供快速访问文档中的这些值。

indices.fielddata.cache.size：数据缓存的最大值，可以是一个节点的堆内存大小的比例，例如30%，也可以是一个绝对数字，比如12GB。默认是无限制，可以最大的利用内存。这个配置是静态的配置，必须在集群中的每个数据节点上启动前配置好。可以通过http://localhost:9200/\_nodes/stats请求来监控节点的使用情况。

#### 节点查询缓存

    查询缓存是负责缓存查询的结果。每个节点都有一个查询缓存，这个缓存为这个节点下的所有分片服务。这个缓存采用最近最少使用算法;  当缓存满的是，把最少使用的数据优先删掉。查询缓存只有使用过滤的时候才会起作用。

indices.queries.cache.size：可以是一个节点的堆内存大小的比例，例如5%，也可以是一个绝对数字，比如 512mb。默认为JVM堆内存大小的10%。

#### 索引缓冲区

    索引缓冲区用于存储新的索引文档。当缓冲区满后，缓冲区中的文件被写入磁盘上的一个段，它会在节点的所有分片上分离。它的设置是静态的，并且必须在群集中的每个数据节点上配置。

indices.memory.index\_buffer\_size：一个节点索引缓冲区的大小，可以是一个节点的堆内存大小的比例获知是一个绝对数字。默认为JVM堆内存大小的10%。

indices.memory.min\_index\_buffer\_size：可以使用此设置指定最小的索引缓冲区大小。默认为48MB。

indices.memory.max\_index\_buffer\_size：可以使用此设置指定最大的索引缓冲区大小。默认为无限制。

indices.memory.min\_shard\_index\_buffer\_size：设置分配给每个分片索引缓冲区的内存最小值，默认4MB。

本文由赛克 蓝德(secisland)原创，转载请标明作者和出处。

#### 分片请求缓存

    当一个搜索请求是对一个索引或者多个索引的时候，每一个分片都是进行它自己内容的搜索然后把结果返回到协调节点，然后把这些结果合并到一起统一对外提供。分片缓存模块缓存了这个分片的搜索结果。这使得搜索频率高的请求会立即返回。

注意：请求缓存只缓存查询条件 size=0的搜索，缓存的内容有hits.total, aggregations, suggestions，不缓存原始的hits。通过now查询的结果将不缓存。

缓存失效：只有在分片的数据实际上发生了变化的时候刷新分片缓存才会失效。刷新的时间间隔越长，缓存的数据越多，当缓存不够的时候，最少使用的数据将被删除。缓存过期可以手工设置，例如：

localhost:9200/kimchy,elasticsearch/\_cache/clear?request\_cache=true

默认情况下缓存未启用，但在创建新的索引时可启用，例如：

PUT localhost:9200/my\_index

{

  "settings": {

    "index.requests.cache.enable": true

  }

}

当然也可以通过动态参数配置来进行设置：

PUT localhost:9200/my\_index/\_settings -d'

{ "index.requests.cache.enable": true }

每请求启用缓存，查询字符串参数request\_cache可用于启用或禁用每个请求的缓存。例如：

localhost:9200/my\_index/\_search?request\_cache=true

{

  "size": 0,

  "aggs": {

    "popular\_colors": {

      "terms": {

        "field": "colors"

      }

    }

  }

}

注意：如果你的查询使用了一个脚本，其结果是不确定的（例如，它使用一个随机函数或引用当前时间）应该设置request\_cache为false禁用请求缓存。

缓存key，数据的缓存是整个JSON，这意味着如果JSON发生了变化 ，例如如果输出的顺序顺序不同，缓存的内容江将会不同。不过大多数JSON库对JSON键的顺序是固定的。

分片请求缓存是在节点级别进行管理的，并有一个默认的值是JVM堆内存大小的1%，可以通过配置文件进行修改。

例如：indices.requests.cache.size: 2%

可以通过localhost:9200/\_stats/request\_cache?pretty&human或者'localhost:9200/\_nodes/stats/indices/request\_cache?pretty&human来缓存监控，缓存的大小（以字节为单位）。

#### 索引恢复

indices.recovery.concurrent\_streams：默认为3。

indices.recovery.concurrent\_small\_file\_streams：默认为2。

indices.recovery.file\_chunk\_size：默认为512KB。

indices.recovery.translog\_ops：默认为1000。

indices.recovery.translog\_size：默认为512KB。

indices.recovery.compress：默认为true。

indices.recovery.max\_bytes\_per\_sec：默认为40MB。

#### TTL区间

    文档有个ttl值可以设置当过期的时候是否需要删除，

indices.ttl.interval：删除程序的运行时间。默认为60。

indices.ttl.bulk\_size：删除处理与批量请求的数量，默认为10000。

## 节点类型详解

*摘要: 在Elasticsearch中节点可以分为主(master)节点，数据(data)节点，客户端节点和部落节点，每种类型的节点有不同的使用方法，对于一个大的集群中，合理的配置这些属性，对集群的健壮性和性能有很大的帮助。*

    当我们启动Elasticsearch的实例，就会启动至少一个节点。相同集群名的多个节点的连接就组成了一个集群，在默认情况下，集群中的每个节点都可以处理http请求，和集群节点的数据传输，集群中所有的节点都知道集群中其他所有的节点，可以将客户端请求转发到适当的节点。节点有以下类型：

主(master)节点：在一个节点上当node.master设置为True（默认）的时候，它有资格被选作为主节点，控制整个集群。

数据(data)节点：在一个节点上node.data设置为True（默认）的时候。该节点保存数据和执行数据相关的操作，如增删改查，搜索，和聚合。

客户端节点：当一个节点的node.master和node.data都设置为false的时候，它既不能保持数据也不能成为主节点，该节点可以作为客户端节点，可以响应用户的情况，并把相关操作发送到其他节点。

部落节点： 当一个节点配置tribe.\*的时候，它是一个特殊的客户端，它可以连接多个集群，在所有连接的集群上执行搜索和其他操作。

默认情况下，节点配置是一个主节点和一个数据节点。这是非常方便的小集群，但随着集群的发展，分离主节点和数据节点将变得非常重要。

节点协调，搜索请求或批量增加索引请求等可能涉及在不同的数据节点上操作。在这些请求会分成两个阶段，一是接收客户端的请求 ，二是协调节点执行相关操作。当数据分散在不同的节点上的时候，协调节点将请求转发到数据节点，每个数据节点在本地执行请求并把结果传输给协调节点，然后协调节点收集各个数据节点的结果转换成一个单一的请求结果返回。所以需要客户端有足够的内存和cpu来处理各个节点的返回结果。

#### 主资格节点说明

    主资格节点的主要职责是和集群操作相关的内容，如创建或删除索引，跟踪哪些节点是群集的一部分，并决定哪些分片分配给相关的节点。稳定的主节点对集群的健康是非常重要的。默认情况下任何一个集群中的节点都有可能被选为主节点。索引数据和搜索查询等操作会占用大量的cpu，内存，io资源，为了确保一个集群的稳定，分离主节点和数据节点是一个比较好的选择。

    虽然主节点也可以协调节点，路由搜索和从客户端新增数据到数据节点，但最好不要使用这些专用的主节点。一个重要的原则是，尽可能做尽量少的工作。创建一个独立的主节点的配置为：

node.master: true

node.data: false

    为了防止数据丢失，配置discovery.zen.minimum\_master\_nodes设置是至关重要的（默认为1），每个主节点应该知道形成一个集群的最小数量的主资格节点的数量。解析一下：

    假设我们有一个集群。有3个主资格节点，当网络发生故障的时候，有可能其中一个节点不能和其他节点进行通信了。这个时候，当discovery.zen.minimum\_master\_nodes设置为1的时候，就会分成两个小的独立集群，当网络好的时候，就会出现数据错误或者丢失数据的情况。当discovery.zen.minimum\_master\_nodes设置为2的时候，一个网络中有两个主资格节点，可以继续工作，另一部分，由于只有一个主资格节点，则不会形成一个独立的集群，这个时候当网络回复的时候，节点又会从新加入集群。设置这个值的原则是：

（master\_eligible\_nodes / 2）+ 1。

这个参数也可以动态设置：

PUT \_cluster/settings

{

  "transient": {

    "discovery.zen.minimum\_master\_nodes": 2

  }

}

本文由赛克 蓝德(secisland)原创，转载请标明作者和出处。

#### 数据节点

    数据节点主要是存储索引数据的节点，主要对文档进行增删改查操作，聚合操作等。数据节点对cpu，内存，io要求较高，在优化的时候需要监控数据节点的状态，当资源不够的时候，需要在集群中添加新的节点。数据节点的配置如下：

node.master: false

node.data: true

#### 客户端节点

    当主节点和数据节点配置都设置为false的时候，该节点只能处理路由请求，处理搜索，分发索引操作等，从本质上来说该客户节点表现为智能负载平衡器。独立的客户端节点在一个比较大的集群中是非常有用的，他协调主节点和数据节点，客户端节点加入集群可以得到集群的状态，根据集群的状态可以直接路由请求。

警告：添加太多的客户端节点对集群是一种负担，因为主节点必须等待每一个节点集群状态的更新确认！客户节点的作用不应被夸大 ， 数据节点也可以起到类似的作用。配置如下：

node.master: false

node.data: false

数据节点路径设置：

    每一个主节点和数据节点都需要知道分片，索引，元数据的物理存储位置，path.data默认位为 $ES\_HOME/data，可以通过配置文件 elasticsearch.yml进行修改，例如：

path.data:  /var/elasticsearch/data

这个设置也可以在命令行上执行，例如：

elasticsearch --path.data /var/elasticsearch/data

这个路径最好进行单独配置，这样Elasticsearch的目录和数据的目录就会分开。当删除了Elasticsearch主目录的时候，不会影响到数据。通过rpm安装默认是分来的。

    数据目录可以被多个节点共享，甚至可以属于不同的集群，为了防止多个节点共享相同的数据路径，可以在配置文件elasticsearch.yml中添加：node.max\_local\_storage\_nodes: 1

注意：在相同的数据目录不要运行不同类型的节点（例如：master, data, client）这会导致意外的数据丢失。

#### 部落节点

    部落节点可以跨越多个集群，它可以接收每个集群的状态，然后合并成一个全局集群的状态，它可以读写所有节点上的数据，部落节点在elasticsearch.yml中的配置如下：

tribe:

    t1:

        cluster.name:   cluster\_one

    t2:

        cluster.name:   cluster\_two

T1和T2是任意的名字代表连接到每个集群。上面的示例配置两集群连接，名称分别是T1和T2。默认情况下部落节点通过广播可以做为客户端连接每一个集群。大多数情况下，部落节点可以像单节点一样对集群进行操作。

注意：以下操作将和单节点操作不同，

如果两个集群的名称相同，部落节点只会连接其中一个。

由于没有主节点，当设置local为true的是，主节点的读操作会被自动的执行，例如：集群统计，集群健康度。

主节点级别的写操作将被拒绝，这些应该是在一个集群进行。

部落节点可以通过块(block)设置所有的写操作和所有的元数据操作，例如：

tribe:

    blocks:

        write:    true

        metadata: true

部落节点可以也可以在选中的索引块中进行配置，例如：

tribe:

    blocks:

        write.indices:    hk\*,ldn\*

        metadata.indices: hk\*,ldn\*

    当多个集群有相同的索引名的时候，默认情况下，部落的节点将选择其中一个。这可以通过tribe.on\_conflict setting进行配置，可以设置排除那些索引或者指定固定的部落名称。

## 节点发现

*摘要: 在Elasticsearch中，节点之间可以相互发现，并把相同集群名称的节点统一成一个集群，那节点是如何发现的呢，这次就谈一下节点发现的一些内部细节。在Elasticsearch内部，zen发现机制是默认的发现模块。它提供了单播发现方式，能够很容易的扩展至云环境。*

    在Elasticsearch中，节点之间可以相互发现，并把相同集群名称的节点统一成一个集群，那节点是如何发现的呢，这次就谈一下节点发现的一些内部细节。在Elasticsearch内部，zen发现机制是默认的发现模块。它提供了单播发现方式，能够很容易的扩展至云环境。发现模块和其他模块集成，例如所有节点的通信是通过传输模块。发现模块分成两个模块。

ping模块，通过ping模块可以寻找其他节点。

单播模块，需要提供一个主机列表作为路由列表。以discovery.zen.ping.unicast作为前缀。

配置         描述

hosts        是一个数组的配置，或以逗号分隔的字符串。格式为host:port，或者host[port1-port2]，ip6的主机必须放                    在括号内，例如127.0.0.1, [::1]。

单播发现依赖传输模块实现。注意port默认应该是9300。

主节点选举

    在集群中，会自动通过ping来来进行相互选举或者加入主节点。discovery.zen.ping\_timeout(默认3秒)配置允许对选举的时间进行调整，用来处理缓慢或拥挤的网络。当一个节点请求加入主节点，它会发送请求到信息到主节点，请求的超时时间配置为discovery.zen.join\_timeout，这个时间比较长，是discovery.zen.ping\_timeout时间的20倍。

    当主节点发生问题的时候，现有的节点又会通过ping来从新选举一个新的主节点。当discovery.zen.master\_election.filter\_client设置为true的时候，在选举主节点的时候从客户端节点(node.client为true,或者node.data和node.master同时为false)的ping操作将被忽略，默认为true。当discovery.zen.master\_election.filter\_data为true。在选举主节点的时候从数据节点(node.data为ture，node.master同时为false)的ping操作将被忽略，默认为false。主节点配置为true的节点一直都有选举的资格。

当节点node.master设置为false后者node.client设置为true的时候，他们将自动排除成为主节点的可能性。

discovery.zen.minimum\_master\_nodes设置需要加入一个新当选的主节点的最小节点数目，或者接受它的作为主节点的最小节点数。如果不满足这一要求，主节点会下台，一个新的选举将重新开始。

故障检测

    有两种方式进行故障检测，第一个是由主节点到所有其他节点的验证，证明他们还活着。另一中，是每个节点ping主节点验证，当主节点有故障的时候会启动选举过程。控制故障检测过程使用discovery.zen.fd前缀设置：

ping\_interval：ping检查的频率。默认值为1秒。

ping\_timeout：ping的超时时间，默认为30秒。

ping\_retries：多少次ping失败或者超时的节点被认为是失败的。默认3次。

    主节点是一个集群中唯一的节点，可以改变集群的状态。主节点处理一个群集状态的更新，适用于所需的更改，并将更新的群集状态发布到集群中的所有其他节点上。每个节点接收发布消息，更新它自己的群集状态，并对主节点进行应答，该主节点等待所有节点响应，然后在处理下一个更新的队列前进行超时处理。discovery.zen.publish\_timeout默认为30秒，可以通过动态配置进行设置。

    一个健康的集群必须有一个主节点，并且有多个满足主节点选举条件的节点，这个数字由discovery.zen.minimum\_master\_nodes设置。discovery.zen.no\_master\_block设置没有有效的主节点时应拒绝的操作。它有两个选择。

all：这个节点上的所有行为被拒绝，包括读写，同时拒绝集群状态的读写操作，比如得到索引设置。

write：基于最后一次的群集配置，写操作被拒绝，允许读操作。这可能会导致部分读取的数据是过期的，因为这个节点已经从群集中分离，这个是默认设置。

 discovery.zen.no\_master\_block 设置不适用于节点的API例如cluster stats, node info和 node stats apis。

## 快照与恢复（？）

*摘要: 在Elasticsearch中的可以创建快照，它创建单个索引或整个群集到一个远程的存储库中。在以前的版本中，系统只支持共享存储的快照创建，最新的版本可以通过插件支持更多的方式。在执行快照操作前，需要在Elasticsearch中进行注册快照仓库。*

    在Elasticsearch中的可以创建快照，它创建单个索引或整个群集到一个远程的存储库中。在以前的版本中，系统只支持共享存储的快照创建，最新的版本可以通过插件支持更多的方式。在执行快照操作前，需要在Elasticsearch中进行注册快照仓库。注册仓库的设置需要执行仓库类型，例如：

请求：PUT /\_snapshot/my\_backup

参数：

{

  "type": "fs",

  "settings": {

        ... repository specific settings ...

  }

}

    当注册后，可以通过查询来得到之前注册的信息。当然系统也支持用逗号分开来查询多个注册，支持通配符查询多个注册，甚至可以使用\_all查询所有注册。

请求：GET /\_snapshot/my\_backup

返回值：

{

  "my\_backup": {

    "type": "fs",

    "settings": {

      "compress": "true",

      "location": "/mount/backups/my\_backup"

    }

  }

}

    共享文件系统存储("type": "fs")使用共享文件系统来存储快照。为了注册共享的文件系统存储库，必须将同一个共享文件系统安装到所有主节点和数据节点上相同的位置上。这个位置（或它的父目录）必须在所有主节点数据节点上设置path.repo参数。假设共享文件系统安装在/mount/backups/my\_backup目录下，在elasticsearch.yml文件中的设置应增加：

path.repo: ["/mount/backups", "/mount/longterm\_backups"]

备注：path.repo设置支持微软Windows UNC路径，只要服务器名和共享名称进行正确的设置，例如：

path.repo: ["\\\\MY\_SERVER\\Snapshots"]

    只有当所有的节点都重新启动后，使用以下面的命令可以为名称为my\_backup的共享文件系统库进行快照和恢复，例如：

请求：PUT http://localhost:9200/\_snapshot/my\_backup

参数：

{

    "type": "fs",

    "settings": {

        "location": "/mount/backups/my\_backup",

        "compress": true

    }

}

    系统支持相对路径的，如果存储位置指定为相对路径，则不需要写具体的全路径，例如：

请求：PUT http://localhost:9200/\_snapshot/my\_backup

参数：

{

    "type": "fs",

    "settings": {

        "location": "my\_backup",

        "compress": true

    }

}

系统支持一下参数：

location：快照位置，必须要有。

compress：打开快照文件的压缩。压缩仅适用于元数据文件，数据文件不进行压缩。默认为true。

chunk\_size：如果需要可以把大的文件分解成不同的快照。块的大小可以指定字节例如1G，10m，5K。默认值为null表示无限的块大小。

max\_restore\_bytes\_per\_sec：使每个节点的恢复的速度。默认为40MB每秒。

max\_snapshot\_bytes\_per\_sec：每个节点生成的快照的速度。默认为40MB每秒。

readonly：使存储库只读。默认为false。

只读仓库

    URL仓库("type": "url")被用在共享文件系统库中创建只读方式数据仓库。在url参数中指定指向共享的文件系统存储库的根节点，支持以下设置：

url：强制性的快照位置。

     url参数支持以下协议："http", "https", "ftp", "file" ， "jar"，在http:, https:, and ftp: 这些协议中，可以支持白名单，白名单的设置在 repositories.url.allowed\_urls 参数中，例如：

repositories.url.allowed\_urls: ["http://www.example.org/root/\*", "https://\*.mydomain.com/\*?\*#\*"]

存储库可以通过插件支持更多的方式，比如：

AWS云插件的S3存储库，Hadoop的HDFS插件环境，Azure云存储库插件。

    当一个存储库被注册时，它立即被验证在所有主节点和数据节点上，以确保它在群集中的所有节点上都可以使用。验证参数可用于在注册或更新存储库时显式禁用，例如：

请求：PUT /\_snapshot/s3\_repository?verify=false

参数：

{

  "type": "s3",

  "settings": {

    "bucket": "my\_s3\_bucket",

    "region": "eu-west-1"

  }

}

验证过程也可以通过运行以下命令来手动执行：

POST /\_snapshot/s3\_repository/\_verify

它返回一个节点列表，如果验证过程失败，则会成功验证或错误消息。

本文由赛克 蓝德(secisland)原创，转载请标明作者和出处。

#### 快照

    一个存储库可以包含同一个群集的多个快照。快照在集群内的名称是唯一的。在一个库中创建名称为snapshot\_1快照可以执行下面的命令：

PUT /\_snapshot/my\_backup/snapshot\_1?wait\_for\_completion=true

    wait\_for\_completion参数指定快照初始化后立即返回（默认）还是等待快照完成后返回。在快照初始化时，所有以前的快照信息加载到内存，这意味着创建一个大的库快照可能需要几秒钟（甚至几分钟）。所以有时候wait\_for\_completion参数设置为false也是需要等待一会的。

    默认情况下所有打开的快照和在集群中开始索引时创建，此行为可以通过指定快照请求中的索引列表来改变，例如：

请求：PUT /\_snapshot/my\_backup/snapshot\_1

参数：

{

  "indices": "index\_1,index\_2",

  "ignore\_unavailable": "true",

  "include\_global\_state": false

}

    可以使用支持多索引语法的索引参数来指定快照的索引列表。快照请求也支持ignore\_unavailable选项，将它设置为true时可以在快照创建过程中把不存在的索引忽略掉。默认情况下，当ignore\_unavailable选项不设置或者设置为false的时候，如果不存在索引，索引的快照请求将失败。通过设置include\_global\_state为false可以防止集群全局状态被被存储为快照的一部分。默认情况下，如果有一个或多个索引没有可用的主分片整个快照会失败，这种行为可以通过设置部分来改变。

    索引快照处理是渐进的。在索引快照的过程中Elasticsearch会分析索引文件的列表是否已存储在存储库中，同时复制上次创建的快照或更改的文件。这允许在一个紧凑形式的存储库中保存多个快照，快照的过程是非阻塞的方式执行。当对索引正在执行快照时所有的索引和搜索操作可以继续执行。然而，快照执行的数据是在快照创建时的时间点确定的，所以在快照过程开始后添加到索引中的记录将不会再快照中出现。在1.2.0之前版本之前，如果群集有迁移或者在索引中初始化分片时会失败，在1.2.0之后，快照的操作会等待这些完成后再操作。一个快照除了创建集群的副本外，还可以存储全局群集元数据，其中包括持久性群集设置和模板。临时设置和注册的快照库不会存储为快照的一部分。

    在任何时间只有一个快照过程在集群中被执行。而一个特定的快照被创建分片时，这分片不能移动到另一个节点，它可能干扰再平衡过程和过滤操作。一次快照完成Elasticsearch只能移动一块到另一个节点（根据当前配置过滤设置和调整算法）。一旦一个快照完成，关于这个快照的信息，可以使用以下命令来获得：

GET /\_snapshot/my\_backup/snapshot\_1

类似的仓库，对多个快照信息可以通过一次进行查询，以及支持通配符，或者用\_all查询所有：

GET /\_snapshot/my\_backup/snapshot\_\*,some\_other\_snapshot

    如果一些快照不可用，该命令将失败。布尔参数ignore\_unvailable可用于返回所有目前可用的快照。可以使用以下命令查询当前正在运行的快照：

GET localhost:9200/\_snapshot/my\_backup/\_current

快照可以从使用下列命令的存储库中删除：

DELETE /\_snapshot/my\_backup/snapshot\_1

    当快照是从库中删除时，Elasticsearch将删除与快照关联的和其他快照也不使用的所有文件，如果执行快照创建的过程中同时删除快照，则快照创建的过程会停止，同时删除相关的文件。因此，删除快照操作可以用来取消被错误启动的长时间运行快照操作。使用下面的命令可以删除存储库：

DELETE /\_snapshot/my\_backup

    当一个库被删除，Elasticsearch仅删除存储库的位置的引用，快照本身并没有被删除。

#### 恢复

使用以下命令可恢复快照：

POST /\_snapshot/my\_backup/snapshot\_1/\_restore

    默认情况下，快照中的所有索引以及群集状态将被恢复，在恢复include\_global\_state选项时，可以指定具体的索引和集群状态被恢复。索引的列表支持多索引语法。rename\_pattern和rename\_replacement选项也可用于重命名索引，可以使用正则表达式，该表达式支持引用原始文本。设置include\_aliases为假可以防止相关索引的别名被一起恢复，例如：

请求：POST /\_snapshot/my\_backup/snapshot\_1/\_restore

参数：

{

  "indices": "index\_1,index\_2",

  "ignore\_unavailable": "true",

  "include\_global\_state": false,

  "rename\_pattern": "index\_(.+)",

  "rename\_replacement": "restored\_index\_$1"

}

    恢复操作可以在一个功能集群上执行。在快照索中有相同数量分片的索引如果是关闭的，则只能恢复。如果索引是关闭的则恢复操作会自动打开索引，如果索引在集群中不存在，恢复操作将会创建新的索引，如果恢复群集状态，则在该群集中不存在的模板将被添加，已恢复的模板会替换为具有相同名称的现有模板。恢复中的设置被添加到现有的设置中。

    部分恢复，默认情况下，如果一个或更多的索引如果没有可用的快照碎片，整个恢复操作将失败。如果一些碎片没有快照，通过设置partial为true，仍可以恢复部分索引。请注意，在这种情况下只有有分片的索引会被正确恢复，没有的索引会建立空的索引。

    在恢复过程中改变索引设置，在恢复过程中大多数索引的设置可以被覆盖。例如，下面的命令将不创建任何副本而切换回默认刷新间隔的恢复索引index\_1：

请求：POST /\_snapshot/my\_backup/snapshot\_1/\_restore

参数：

{

  "indices": "index\_1",

  "index\_settings": {

    "index.number\_of\_replicas": 0

  },

  "ignore\_index\_settings": [

    "index.refresh\_interval"

  ]}

请注意，一些设置，如index.number\_of\_shards在恢复操作期间是无法更改的。

    恢复到不同的集群，快照中存储的信息不依赖于特定的群集或群集名称。因此，它有可能恢复从一个群集到另一个群集的快照。仅要求注册存储库中包含快照，并启动恢复过程，新的群集不具有相同的大小或拓扑结构。如果一个集群有的空间比较小，需要考虑的事情有：首先，要确保新的集群有足够的容量来存储快照中的所有索引。在还原的过程中，可以改变索引设置，以减少复制的次数，这可以帮助将快照恢复到较小的群集，也可以选择使用索引参数。在版本1.5.0之前，Elasticsearch没有检查恢复持续的设置使得有可能恢复出错，参数discovery.zen.minimum\_master\_nodes 禁用一个小集群直到添加所需的主节点数。从版本1.5.0此的设置将被忽略。如果在原始集群的索引被分配到特定的节点，使用过滤分片配置，同样的规则将在新的集群执行。因此，如果新的群集不包含具有已有属性的节点，则该恢复的索引可以被分配，除非这些索引分配设置在恢复操作期间改变，否则不会恢复成功。

#### 快照状态

快照statusedit

使用以下命令可获得当前运行快照的详细状态信息列表：

GET /\_snapshot/\_status

下面的命令将返回有关当前正在运行的快照的信息。通过指定一个存储库的名称，可以将结果限制为特定的存储库：

GET /\_snapshot/my\_backup/\_status

如果指定了存储库名称和快照标识，该命令将返回给定快照的详细状态信息：

GET /\_snapshot/my\_backup/snapshot\_1/\_status

同样支持多个id的查询，例如：

GET /\_snapshot/my\_backup/snapshot\_1,snapshot\_2/\_status

监控快照恢复过程

    当快照的进度和恢复正在运行有几种方法来监视它们过程。wait\_for\_completion参数会阻止客户端直到操作完成。这是最简单的方法，可以用来获得有关操作完成的通知。

快照操作还可以定期对快照信息进行监视，例如：

GET /\_snapshot/my\_backup/snapshot\_1

    请注意，快照信息的操作使用相同的资源和线程池作为快照操作。所以，当大的分片被快照会导致快照操作，在返回结果之前在等待可用资源，所以这种会占用很多的时间。为了获得更直接的快照状态的完整信息，可以使用快照状态命令来查询，例如：

GET /\_snapshot/my\_backup/snapshot\_1/\_status

    快照信息方法在处理过程中返回快照的基本信息，快照状态返回在快照中每个分片的当前状态。

    在恢复操作时，Elasticsearch的集群通常会进入红色状态。这是因为还原操作开始恢复的索引主要碎片。在这个操作时，主分片变得不可用，这体现在红色状态。一旦碎片恢复完成Elasticsearch是转换为标准的复制过程，创建所需数量的副本这时集群切换到黄色状态。一旦所有所需的复制品被创建，群集切换到绿色状态。群集健康操作仅为恢复过程的高层次状态。通过使用索引恢复和精简api可以获得更详细的恢复过程中的状态。

停止当前正在运行的快照和恢复操作

    快照和恢复操作允许一次只运行一个快照或一次恢复。如果一个当前运行的快照被错误地执行，或者需要很长的时间，可以使用快照删除操作来终止它。快照删除时是会进行检查，如果删除快照正在运行，则停止此次删除。

    还原操作使用标准的分片恢复机制。因此，可以通过删除索引来取消恢复的操作。请注意，在这种情况下所有已删除索引的数据将被从集群中移除。

    快照和恢复操作对集群块的影响，快照和恢复操作都受集群和索引块的影响。例如，注册和注销库需要通过编写全局元数据来进行访问。快照操作需要所有的索引和元数据以及全局元数据都是可读的。恢复操作需要全局元数据是可写的，但是索引水平块将被忽略，因为在恢复的过程中索引是重建的。请注意，存储库内容不是群集的一部分，因此群集块不影响内部存储库操作，例如已注册的存储库中的列表或删除快照。

## 插件

### 插件安装

*摘要: Elasticsearch通过插件来增强其个性化功能需求。他们可以添加自定义映射类型，自定义分析器，本地脚本，自定义发现等等。在Elasticsearch中，有三种类型的插件：java插件，站点插件和混合插件。*

    Elasticsearch通过插件来增强其个性化功能需求。他们可以添加自定义映射类型，自定义分析器，本地脚本，自定义发现等等。在Elasticsearch中，有三种类型的插件：

java插件：这些插件只包含jar文件，并且必须安装在集群中的每个节点上。安装后，每个节点必须重新启动后才生效。

站点插件：这些插件包含了静态的Web内容，如JavaScript，HTML和CSS文件，站点插件只需要在一个节点上安装，并且不需要重新启动就可以使用。站点插件通过url进行访问，例如：http://localhost:9200/\_plugin/[plugin name]。

混合插件：混合插件是即包含jar文件又站点插件。

#### 插件管理

    插件是通过脚本来进行管理的，可以安装插件，查询插件和删除插件。正常情况下插件位于$ES\_HOME/bin下，通过rpm等安装的位置可能会不同。可以通命令获取插件帮助：在目录下执行，plugin -h

plugin -h

NAME

    plugin - Manages plugins

SYNOPSIS

    plugin <command>

DESCRIPTION

    Manage plugins

COMMANDS

    install    Install a plugin

    remove     Remove a plugin

    list       List installed plugins

NOTES

    [\*] For usage help on specific commands please type "plugin <command> -h"

注意：linux需要用管理员来运行，比如sudo plugin -h

插件安装：

    每种插件都会有插件安装文档，但大多数情况下，插件安装有以下几种方式：

核心插件：

sudo bin/plugin install [plugin\_name]

例如安装Lucenea icu插件。sudo bin/plugin install analysis-icu，这样就会按照合适的版本到Elasticsearch中。

非核心插件：非核心插件可以是官方提供的，也可以是社区提供的，可以从官方，Maven或者GitHub中下载安装。

sudo bin/plugin install [org]/[user|component]/[version]

例如，安装github上的插件，plugin install lmenezes/elasticsearch-kopf

插件会尝试先到官方去下载，如果没有找到会到maven.com中去下载，如果在没有找到回到github中去下载。脚本还是非常智能的。

当我们从直接maven中央库中安装时可以直接使用下面的方式，最后的版本号是必须要写的。

plugin install org.elasticsearch.plugin/mapper-attachments/3.0.0

从自定义网址或者本地安装：plugin install [url] ，例如，在本地文件系统中安装一个插件，可以运行：plugin install file:///path/to/plugin.zip。

    插件脚本会拒绝从一个不受信任的证书的https网站安装。使用HTTPS证书自签名，需要添加的CA证书到本地信任库，可以通过java脚本添加：plugin -Djavax.net.ssl.trustStore=/path/to/trustStore.jks install https://...

查询插件：plugin list

删除插件：plugin remove [pluginname]，jar插件删除后要重启。

Silent/Verbose参数：

当使用--verbose参数输出更多的调试信息， 当使用--silent参数时可以关闭所有输出。该脚本可以返回下面的代码：

0：一切都好

64：未知命令或不正确的选项参数

74：输入输出错误

70：任何其他错误

自定义配置目录：如果elasticsearch.yml配置文件在一个自定义的位置，在使用插件脚本时需要指定配置文件的路径，例如：plugin -Des.path.conf=/path/to/custom/config/dir install <plugin name>，您可以设置conf\_dir环境变量来指定自定义配置文件的路径。

超时设置：可以指定超时时间，例如等待30秒，plugin install mobz/elasticsearch-head --timeout 30s。当设置为0时一直等待。

代理设置：可以通过代理安装插件，Java的设置是proxyHost和proxyPort。例如：在Unix中

plugin install mobz/elasticsearch-head -DproxyHost=host\_name -DproxyPort=port\_number

在Windows中，需要添加参数到java\_opts环境变量，例如：

set JAVA\_OPTS="-DproxyHost=host\_name -DproxyPort=port\_number"

plugin install mobz/elasticsearch-head

自定义插件目录：插件目录可以在配置文件中elasticsearch.yml进行修改，参数：path.plugins: /path/to/custom/plugins/dir。

强制插件：可以在配置文件中elasticsearch.yml中添加plugin.mandatory来设置必须要安装的插件，例如：plugin.mandatory: mapper-attachments,lang-python。注意：出于安全的原因，如果缺少一个强制插件，该节点将不会启动成功。

### 插件清单

*摘要: 在Elasticsearch中非常多的插件，这些插件主要分成以下几类：API插件，报警插件，分析插件，发现插件，管理和站点插件，映射器插件，脚本插件，安全插件，快照/恢复插件，传输插件。有很多新版本都不支持，本文介绍的插件基本都可以在新版本上使用。*

    在Elasticsearch中非常多的插件，这些插件主要分成以下几类：API插件，报警插件，分析插件，发现插件，管理和站点插件，映射器插件，脚本插件，安全插件，快照/恢复插件，传输插件。

#### API插件

    API插件主要对Elasticsearch添加新的API特性或者功能，通常用于搜索或者映射。

核心插件：delete-by-query

该插件可以通过查询来删除文档，在内部，它使用Scroll和BulkAPI来删除文件。注意不要用此方法来删除大量的文档，因为它是一条一条的删除文档，这会导致消耗的时间比较长，比较好的方式是创建一个新的文档，把需要的文档copy过去。

使用方法：

DELETE /twitter/tweet/\_query?q=user:kimchy

或者：

DELETE /twitter/tweet/\_query

参数：

{

  "query": {

    "term": {

      "user": "kimchy"

    }

  }

}

社区插件：

[carrot2 Plugin](https://github.com/carrot2/elasticsearch-carrot2)::基于carrot2或者Lingo3G算法的聚合插件。

[SQL language Plugin:](https://github.com/kzwang/elasticsearch-image)通过Sql语法来搜索Elasticsearch

[WebSocket Change Feed Plugin](https://github.com/jurgc11/es-change-feed-plugin) ：通过WebSocket连接到Elasticsearch节点和接收数据

#### 报警插件

当Elasticsearch的索引等指标超出阈值时报警插件会触发报警。

Watcher插件是Elasticsearch官方支持的报警插件，但是这个插件是付费的，初次安装有30天的试用许，过期后可以付费购买。

#### 分词插件

核心插件：

analysis-icu：ICU分析插件集成Lucene ICU模块到Elasticsearch，它支持Unicode的ICU库，包括更好的分析亚洲语言。

analysis-phonetic：语音分析插件。

analysis-smartcn：中科院的中文分词插件，不支持扩展，不建议使用。

analysis-stempel：支持Lucene的Stempel analysis 模块的插件。

社区插件：

[IK Analysis Plugin](https://github.com/medcl/elasticsearch-analysis-ik) ：比较好的中文插件。[详见](http://my.oschina.net/secisland/blog/617822)介绍。

[Mmseg Analysis Plugin](https://github.com/medcl/elasticsearch-analysis-mmseg) ：集成Lucene mmseg4j-analyzer的插件，支持中文。

[Pinyin Analysis Plugin](https://github.com/medcl/elasticsearch-analysis-pinyin) ：集成Pinyin4j插件，pinyin4j是支持汉字和拼音系统之间转换的最受欢迎流行的java库。可自定义拼音输出格式。上面三个插件都是处于一个人之手，佩服。

[Network Addresses Analysis Plugin](https://github.com/ofir123/elasticsearch-network-analysis)：网络MAC地址分析插件。

**发现插件**

发现节点插件是替换Elasticsearch自身发现功能的插件，[详见](http://my.oschina.net/secisland/blog/619686)介绍。

核心插件主要包括：

* [AWS Cloud](https://www.elastic.co/guide/en/elasticsearch/plugins/current/cloud-aws.html)：亚马逊云。
* [Azure Cloud](https://www.elastic.co/guide/en/elasticsearch/plugins/current/cloud-azure.html)：微软云。
* [GCE Cloud](https://www.elastic.co/guide/en/elasticsearch/plugins/current/cloud-gce.html)：google云。
* [Multicast](https://www.elastic.co/guide/en/elasticsearch/plugins/current/discovery-multicast.html)：组播插件发送多播消息，发现在集群中的其他节点。

社区插件

[eskka Discovery Plugin](https://github.com/shikhar/eskka)：支持Gossip协议的Akka Cluster集群节点。

[Kubernetes Discovery Plugin](https://github.com/fabric8io/elasticsearch-cloud-kubernetes)：通过Kubernetes API发现节点。

**管理和站点插件**

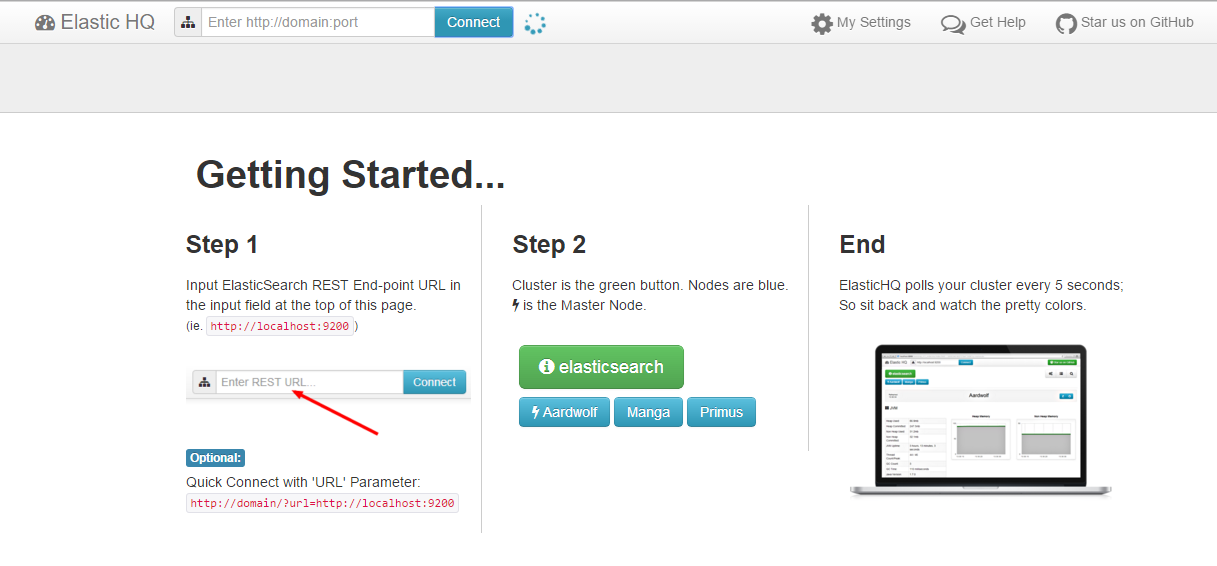
核心插件：Marvel 官方支持的管理插件是收费插件。

社区插件：

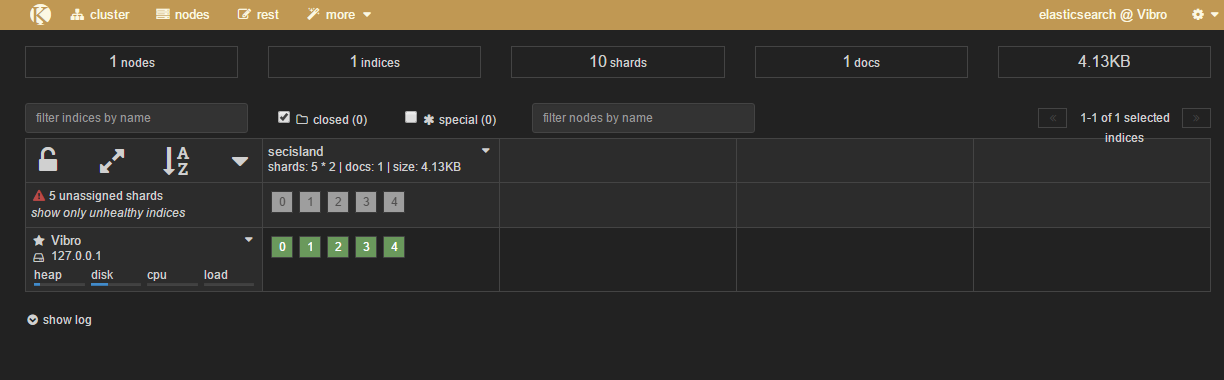
[Elasticsearch Head Plugin](https://github.com/mobz/elasticsearch-head) ：mobz/elasticsearch-head插件，推荐使用。



[Elasticsearch HQ](https://github.com/royrusso/elasticsearch-HQ) ：royrusso/elasticsearch-HQ插件。这个插件在2.2.0版本链接不成功，界面看起来不错。



[Kopf Plugin](https://github.com/lmenezes/elasticsearch-kopf) ：lmenezes/elasticsearch-kopf插件，推荐使用，界面看起来也很不错。



**映射器插件**

* [Mapper Attachments Plugin](https://www.elastic.co/guide/en/elasticsearch/plugins/current/mapper-attachments.html)：集成了 Apache Tika提供了一个新的字段类型，它可以直接索引pdf或者word文档。
* [Mapper Size Plugin](https://www.elastic.co/guide/en/elasticsearch/plugins/current/mapper-size.html)：提供了 \_size原数据，\_source字段可以查询指定字节大小。
* [Mapper Murmur3 Plugin](https://www.elastic.co/guide/en/elasticsearch/plugins/current/mapper-murmur3.html)：插件提供了在存储和索引的时候计算字段值的哈希索值。这在大字段的聚合的时候比较有用。

#### 脚本插件

* [JavaScript Language](https://www.elastic.co/guide/en/elasticsearch/plugins/current/lang-javascript.html)：JavaScript脚本支持。
* [Python Language](https://www.elastic.co/guide/en/elasticsearch/plugins/current/lang-python.html)：Python脚本支持。

#### 安全插件

核心插件：Shield，这个是官方提供的收费插件。

社区插件：

[Kerberos/SPNEGO Realm](https://github.com/codecentric/elasticsearch-shield-kerberos-realm): 通过Kerberos/SPNEGO认证的插件。

[Readonly REST](https://github.com/sscarduzio/elasticsearch-readonlyrest-plugin): 防止修改和删除数据的插件。

#### 快照/恢复插件

* [AWS Cloud](https://www.elastic.co/guide/en/elasticsearch/plugins/current/cloud-aws.html)：亚马逊备份恢复插件。
* [Azure Cloud](https://www.elastic.co/guide/en/elasticsearch/plugins/current/cloud-azure.html)：微软云备份恢复插件。
* [Hadoop HDFS Repository](https://github.com/elastic/elasticsearch-hadoop/tree/master/repository-hdfs)：Hadoop备份回复插件。

**传输插件**

核心插件：Servlet transport，servlet的REST接口的插件。

社区插件：

[ØMQ transport plugin](https://github.com/tlrx/transport-zeromq)：transport-zeromq，MQ传输插件。

## 索引管理

*摘要: 在Elasticsearch中的索引管理主要包括创建索引，删除索引，获取索引，判断索引是否存在，打开关闭索引接口。*

    在Elasticsearch中的索引管理主要包括创建索引，删除索引，获取索引，判断索引是否存在，打开关闭索引接口。

**创建索引**

   创建索引接口允许系统创建实例化索引，Elasticsearch支持创建多个索引，包括对多个索引的操作。

创建索引

curl -XPUT 'http://localhost:9200/twitter/'

curl -XPUT 'http://localhost:9200/twitter/' -d '

 index :

   number\_of\_shards : 3

   number\_of\_replicas : 2'

number\_of\_shards 索引设置主分片的数量，默认是5，占用5个分片。

number\_of\_replicas 索引副本分片的数量，默认是1个。

上面第二个curl示例展示了创建一个‘twitter’的索引时使用YAML语法进行自定义设置。

在这个例子中，创建了一个拥有3个主分片的索引，每个分片有2个副本分片。

上面示例对应的JSON格式如下：

curl -XPUT 'http://localhost:9200/twitter/' -d '{

 "settings" : {

   "index" : {

     "number\_of\_shards" : 3,

     "number\_of\_replicas" : 2

   }

 }

}'

或者简写成：

curl -XPUT 'http://localhost:9200/twitter/' -d '{

 "settings" : {

   "number\_of\_shards" : 3,

   "number\_of\_replicas" : 2

 }

}'

提示：没必要写全settings中的index选项。

映射：创建索引允许你设置一个或多个映射。

curl -XPOST localhost:9200/test -d '{

  "settings" : {

   "number\_of\_shards" : 1

  },

  "mappings" : {

   "type1" : {

     "properties" : {

       "field1" : { "type" : "string", "index" : "not\_analyzed" }

     }

   }

 }

}'

**删除索引**

删除索引示例：

curl -XDELETE 'http://localhost:9200/twitter/'

上面的示例删除了名为twitter的索引。删除索引需要指定索引名称，别名或者通配符。

删除索引可以使用逗号分隔符，或者使用\_all或\*号删除全部索引。

注意：\_all 或 \* 删除全部索引时要谨慎操作。

为了防止误删除，可以设置config/elasticsearch.yml属性action.destructive\_requires\_name为true，禁止使用通配符或\_all删除索引，必须使用名称或别名才能删除该索引。

**获取索引**

获取索引接口允许从一个或多个索引中获取信息。

curl -XGET 'http://localhost:9200/twitter/'

上面的示例获取名为twitter的索引。获取索引需要指定索引名称，别名或者通配符。

获取索引可以使用通配符获取多个索引，或者使用\_all或\*号获取全部索引。

返回结果过滤：

可以自定义返回结果的属性。

curl -XGET 'http://localhost:9200/twitter/\_settings,\_mappings'

上面示例只返回twitter索引的settings和mappings属性。

可配置的属性包括\_settings, \_mappings, \_warmers and \_aliases。

**判断索引是否存在**

    如果只想检查索引存在的状态，而不需要获取内容，使用HEAD代替GET，返回内容不包括索引内容。

用来检查索引是否存在。如下：

curl -XHEAD -i 'http://localhost:9200/twitter'

HTTP状态码标识索引是否存在。索引存在返回200，不存在返回404。

**打开关闭索引接口**

    打开/关闭索引接口允许关闭一个打开的索引或者打开一个已经关闭的索引。关闭的索引只能显示索引元数据信息，不能够进行读写操作。

打开/关闭索引的方式是/{索引名}/\_close 或者/{索引名}/\_open，完整示例如下：

curl -XPOST 'localhost:9200/my\_index/\_close'

curl -XPOST 'localhost:9200/my\_index/\_open'

可以同时打开或关闭多个索引。如果指向不存在的索引会抛出错误。可以使用配置ignore\_unavailable=true，不显示异常。

使用\_all打开或关闭全部索引，或者使用通配符表示全部（比如 \*）

设置config/elasticsearch.yml属性action.destructive\_requires\_name为true，禁止使用通配符或者\_all标识索引。

因为关闭的索引会继续占用磁盘空间而不能使用，所以关闭索引接口可能造成磁盘空间的浪费。

禁止使用关闭索引功能，可以设置settingscluster.indices.close.enable为false，默认是true。

## 查询更新接口（2.3新增）

此接口在索引中更新每一个文档，而文档的内容并没有改变的情况下使用，这在增加新的属性或者修改映射的时候非常有用。例如：

请求：POST /twitter/\_update\_by\_query?conflicts=proceed

返回结果为：

{

  "took" : 639,

  "updated": 1235,

  "batches": 13,

  "version\_conflicts": 2,

  "failures" : [ ]

}

\_update\_by\_query当系统开始或者正在用内部版本号进行索引的时候获得一个索引的快照，当快照被处理时或者当索引请求被处理时，如果文档发生了变化，系统将会产生版本冲突。当匹配版本的文档更新后，文档的版本号会递增。当更新和查询失败时，\_update\_by\_query将会中止并在返回值中返回失败的原因。已执行的更新将会保持。也就是说，这个过程是不回滚的，只有中止。当第一条失败时，会导致程序中止，这意味着所有的更新失败，这时候会返回大量的失败元素。

    如果你想当版本冲突时只进行简单的计数，而不中止，可以在url中设置 conflicts=proceed或者在请求内容中设置"conflicts": "proceed"，在api中，可以只针对索引中的一个类型进行操作，例如：

请求：POST /twitter/tweet/\_update\_by\_query?conflicts=proceed

同样可以采用DSL语法进行查询，例如：

请求：POST /twitter/\_update\_by\_query?conflicts=proceed

{

  "query": {

    "term": {

      "user": "kimchy"

    }

  }

}

这里的搜索和其他搜索的语法是一致的。

    之前介绍的都是没有改变文档内容的，其实\_update\_by\_query是可以在支持脚本对文档内容的更新。例如：

请求：POST /twitter/\_update\_by\_query

{

  "script": {

    "inline": "ctx.\_source.likes++"

  },

  "query": {

    "term": {

      "user": "kimchy"

    }

  }

}

这个接口可以在多个索引和多个类型的中同时操作，例如：

请求：POST /twitter,blog/tweet,post/\_update\_by\_query

同时，也可以指定路由查询，例如：

请求：POST /twitter/\_update\_by\_query?routing=1

默认情况下，\_update\_by\_query采用滚动100批次。你可以在URL参数用scroll\_size来改变批次的大小，例如：

请求：POST /twitter/\_update\_by\_query?scroll\_size=1000

    除了标注的参数外，\_update\_by\_query还支持refresh, wait\_for\_completion, consistency, timeout.

    刷新(refresh)操作只是当请求完成时在索引更新时所有分片，这不同于更新索引时的刷新，索引的时候当收到新的数据的时候，只针对当前数据分片进行刷新。如果请求中包含wait\_for\_completion=false，Elasticsearch将进行执行前检查，启动请求，然后返回任务，用这个任务可以取消操作或获得任务的状态。一旦请求完成任务就结束了，唯一有记录的地方是在elasticsearch日志文件中有任务的执行结果，这个问题将在以后的版本修复。一致性(consistency)控制每次请求时有多少分片的拷贝被响应，超时控制每次请求等待的时间。在Bulk API中可以精确的知道他们是如何工作的。超时控制多久每批等待成为目标碎片。

响应介绍，每次响应的内容大概是：

{

  "took" : 639,

  "updated": 0,

  "batches": 1,

  "version\_conflicts": 2,

  "failures" : [ ]

}

took：从开始到结束的整个操作的毫秒数。

updated：已成功更新的文档数。

batches：滚动响应的数量。

version\_conflicts：通过查询命中更新的版本冲突的数量。

failures：所有索引失败的数组。如果这是非空的，则请求中止。

当查询更新操作发生后，可以使用任务接口来获取它们的状态，例如：

请求：POST /\_tasks/?pretty&detailed=true&action=\*byquery

响应：

{

  "nodes" : {

    "r1A2WoRbTwKZ516z6NEs5A" : {

      "name" : "Tyrannus",

      "transport\_address" : "127.0.0.1:9300",

      "host" : "127.0.0.1",

      "ip" : "127.0.0.1:9300",

      "attributes" : {

        "testattr" : "test",

        "portsfile" : "true"

      },

      "tasks" : {

        "r1A2WoRbTwKZ516z6NEs5A:36619" : {

          "node" : "r1A2WoRbTwKZ516z6NEs5A",

          "id" : 36619,

          "type" : "transport",

          "action" : "indices:data/write/update/byquery",

          "status" : {

            "total" : 6154,

            "updated" : 3500,

            "created" : 0,

            "deleted" : 0,

            "batches" : 36,

            "version\_conflicts" : 0,

            "noops" : 0

          },

          "description" : ""

        }

      }

    }

  }

}

    该对象包含实际状态。里面有个重要的参数是total，预计重建执行总的操作数。你可以通过添加更新、创建和删除字段来估计这个进展。当它们的总和等于总的列时，该请求将完成。

    当创建了一个没有动态映射的索引，如果有新的数据内容，系统会添加新的映射值来匹配数据中的多个字段，例如：

请求：PUT test，数据结构只有test

{

  "mappings": {

    "test": {

      "dynamic": false,

      "properties": {

        "text": {"type": "string"}

      }

    }

  }

}

插入数据：

请求：POST test/test?refresh

{

  "text": "words words",

  "flag": "bar"

}

然后得到数据结构：

请求：PUT test/\_mapping/test

{

  "properties": {

    "text": {"type": "string"},

    "flag": {"type": "string", "analyzer": "keyword"}

  }

}

这个时候会自动添加一个flag映射。

但这个时候查询数据，不会找到任何值：

请求：POST test/\_search?filter\_path=hits.total

参数：

{

  "query": {

    "match": {

      "flag": "bar"

    }

  }

}

返回：

{

  "hits" : {

    "total" : 0

  }

}

这个时候我们可以用更新的查询请求来获得数据，例如：

请求：POST test/\_update\_by\_query?refresh&conflicts=proceed

POST test/\_search?filter\_path=hits.total

参数：

{

  "query": {

    "match": {

      "flag": "foo"

    }

  }

}

返回：

{

  "hits" : {

    "total" : 1

  }

}

## 索引映射管理（！！）

*摘要: Elasticsearch准许对已经建好的索引映射进行管理。包括增加映射，修改映射，获取映射，获取字段映射，和判断类型是否存在。*

#### 增加映射

API允许你向索引(index)添加文档类型(type)，或者向文档类型(type)中添加字段(field)。

PUT secisland

{

  "mappings": {

    "log": {

      "properties": {

        "message": {

          "type": "string"

        }

      }

    }

  }

}

添加索引名为secisland，文档类型为log，其中包含字段message，字段类型是字符串。

PUT secisland/\_mapping/user

{

  "properties": {

    "name": {

      "type": "string"

    }

  }

}

向已经存在的索引secisland添加文档类型为user，包含字段name，字段类型是字符串。

PUT secisland/\_mapping/log

{

  "properties": {

    "user\_name": {

      "type": "string"

    }

  }

}

已经存在的索引secisland，文档类型为log，添加新的字段user\_name，字段类型是字符串。

多个索引设置映射,可以一次向多个索引添加文档类型

PUT /{index}/\_mapping/{type}

{ body }

{index}可以有多种方式,逗号分隔，比如test1,test2,test3

\_all,表示所有索引，通配符，\*表示所有，test\*表示以test开头

{type}需要添加或更新的文档类型

{body}需要添加的字段或字段类型

#### 更新字段映射

在一般情况下，对现有字段的映射不会更新。对这个规则有一些例外。例如：

新的属性被添加到对象数据类型的字段。

新的多域字段被添加到现有的字段。

doc\_values可以被禁用。

增加了ignore\_above参数。

例如：

请求：PUT my\_index

参数:

 {

  "mappings": {

    "user": {

      "properties": {

        "name": {

          "properties": {

            "first": {

              "type": "string"

            }

          }

        },

        "user\_id": {

          "type": "string",

          "index": "not\_analyzed"

        }

      }

    }

  }

 }

user的第一个name属性是对象数据类型(Object datatype)字段，对上个索引进行修改：

请求：PUT my\_index/\_mapping/user

参数：

{

  "properties": {

    "name": {

      "properties": {

        "last": {

          "type": "string"

        }

      }

    },

    "user\_id": {

      "type": "string",

      "index": "not\_analyzed",

      "ignore\_above": 100

    }

  }

}

修改映射，对第一个对象数据类型增加了一个熟悉是last。修改了user\_id, 通过设置ignore\_above使默认的更新为0。

#### 不同类型之间的冲突

    在同一个索引的不同类型(type)中，相同名称的字段中必须有相同的映射，因为他们内部是在同一个领域内，如果试图在这种情况下更新映射参数，系统将会抛出异常。除非在更新的时候指定 update\_all\_types参数。在这种情况下它将更新在相同的指标参数在所有同名的字段。

例如：

请求：PUT my\_index

参数：

{

  "mappings": {

    "type\_one": {

      "properties": {

        "text": {

          "type": "string",

          "analyzer": "standard"

        }

      }

    },

    "type\_two": {

      "properties": {

        "text": {

          "type": "string",

          "analyzer": "standard"

        }

      }

    }

  }

}

修改映射

请求：PUT my\_index/\_mapping/type\_one

参数：

{

  "properties": {

    "text": {

      "type": "string",

      "analyzer": "standard",

      "search\_analyzer": "whitespace"

    }

  }

}

这个时候会抛出异常，然后增加参数，update\_all\_types，这个时候会同时更新两个类型。

请求：PUT my\_index/\_mapping/type\_one?update\_all\_types

 {

  "properties": {

    "text": {

      "type": "string",

      "analyzer": "standard",

      "search\_analyzer": "whitespace"

    }

  }

}

#### 获取映射

获取文档映射接口允许通过索引或者索引和类型来检索。

curl -XGET 'http://localhost:9200/twitter/\_mapping/tweet'

系统同时支持获取多个索引和类型的语法：

获取文档映射接口一次可以获取多个索引或文档映射类型。该接口通常是如下格式：

host:port/{index}/\_mapping/{type}，{index}和{type}可以接受逗号(,)分隔符，也可以使用\_all来表示全部索引。如下所示：

curl -XGET 'http://localhost:9200/\_mapping/twitter,kimchy'

curl -XGET 'http://localhost:9200/\_all/\_mapping/tweet,book'

第一个省略\_all，第二个使用\_all都是表示全部索引。也就是说，下面两个是等价的：

curl -XGET 'http://localhost:9200/\_all/\_mapping'

curl -XGET 'http://localhost:9200/\_mapping'

#### 获取字段映射

获取文档字段接口允许你检索一个或多个字段。这个用来检索想要检索的字段，而不是某个索引或者文档类型的全部内容。

这段请求只返回字段为text的内容：

curl -XGET 'http://localhost:9200/twitter/\_mapping/tweet/field/text'

响应结果如下（假定text为String类型）

{

   "twitter": {

      "tweet": {

         "text": {

            "full\_name": "text",

            "mapping": {

               "text": { "type": "string" }

            }

         }

      }

   }

}

获取多索引和类型的字段映射。

获取文档字段映射接口一次可以获取多个索引或文档映射类型。该接口通常是如下格式：host:port/{index}/{type}/\_mapping/field/{field}

{index}，{type}，{field}可以使用逗号(,)分隔，也可以使用\*作为通配符{type}，{field}可以使用逗号(,)分隔。

其中{index}可以使用\_all表示全部索引，示例如下：

curl -XGET 'http://localhost:9200/twitter,kimchy/\_mapping/field/message'

curl -XGET 'http://localhost:9200/\_all/\_mapping/tweet,book/field/message,user.id'

curl -XGET 'http://localhost:9200/\_all/\_mapping/tw\*/field/\*.id'

指定字段

获取文档字段接口，可以使用逗号(,)分隔符或者通配符(\*)。

如下文档示例，如果只使用字段名id会产生歧义。

{

     "article": {

         "properties": {

             "id": { "type": "string" },

             "title":  { "type": "string"},

             "abstract": { "type": "string"},

             "author": {

                 "properties": {

                     "id": { "type": "string" },

                     "name": { "type": "string" }

                 }

             }

         }

     }

 }

如果想要表示author中的id，name，使用author.id，author.name。请求如下：

curl -XGET "http://localhost:9200/publications/\_mapping/article/field/

author.id,abstract,author.name"

返回结果如下：

{

   "publications": {

      "article": {

         "abstract": {

            "full\_name": "abstract",

            "mapping": {

               "abstract": { "type": "string" }

            }

         },

         "author.id": {

            "full\_name": "author.id",

            "mapping": {

               "id": { "type": "string" }

            }

         },

         "author.name": {

            "full\_name": "author.name",

            "mapping": {

               "name": { "type": "string" }

            }

         }

      }

   }

}

#### 判断类型是否存在

检查索引或文档类型是否存在

curl -XHEAD -i 'http://localhost:9200/twitter/tweet'

存在返回200，不存在返回404。

## 索引别名

*摘要: 在Elasticsearch所有的API中，对应的是一个或者多个索引。Elasticsearch可以对一个或者多个索引指定别名，通过别名可以查询到一个或者多个索引的内容，在内部，Elasticsearch会自动把别名映射到响应的索引上。可以对别名编写过滤器或者路由，在系统中别名不能重复，也不能和索引名重复。其实在Elasticsearch的别名机制有点像数据库中的视图。*

#### 别名管理

    在Elasticsearch所有的API中，对应的是一个或者多个索引。Elasticsearch可以对一个或者多个索引指定别名，通过别名可以查询到一个或者多个索引的内容，在内部，Elasticsearch会自动把别名映射到响应的索引上。可以对别名编写过滤器或者路由，在系统中别名不能重复，也不能和索引名重复。其实在Elasticsearch的别名机制有点像数据库中的视图。例如：把索引test1增加一个别名alias1。

请求：POST 'http://localhost:9200/\_aliases

参数：

{

    "actions" : [

        { "add" : { "index" : "test1", "alias" : "alias1" } }

    ]

}

删除别名：

请求是一样的，参数不一样。

{

    "actions" : [

        { "remove" : { "index" : "test1", "alias" : "alias1" } }

    ]

}

注意：别名没有修改的语法，当需要修改别名的时候，可以先删除别名，然后再增加别名，例如：

{

    "actions" : [

        { "remove" : { "index" : "test1", "alias" : "alias1" } },

        { "add" : { "index" : "test1", "alias" : "alias2" } }

    ]

}

一个别名关联多个索引：

{

    "actions" : [

        { "add" : { "index" : "test1", "alias" : "alias1" } },

        { "add" : { "index" : "test2", "alias" : "alias1" } }

    ]

}

或者用下面的语法：

{

    "actions" : [

        { "add" : { "indices" : ["test1", "test2"], "alias" : "alias1" } }

    ]

}

或者使用通配符：

{

    "actions" : [

        { "add" : { "index" : "test\*", "alias" : "all\_test\_indices" } }

    ]

}

注意：通配符指定的索引只是在当前生效，后面添加的索引不会被自动添加到别名上。

对一个别名做索引的时候，如果别名关联多个索引的时候会报错。

#### 过滤索引别名

    通过过滤索引来指定别名提供了对索引查看的不同视图，该过滤器可以使用查询DSL定义适用于所有的搜索，计数，删除查询，这样的操作与此别名。

要创建一个过滤的别名，首先我们需要确保映射中已经存在的字段：

创建一个索引，请求：PUT http://localhost:9200/test1

参数

{

  "mappings": {

    "type1": {

      "properties": {

        "user" : {

          "type": "string",

          "index": "not\_analyzed"

        }

      }

    }

  }

}

创建过滤别名，请求：POST  http://localhost:9200/\_aliases

参数：

{

    "actions" : [

        {

            "add" : {

                 "index" : "test1",

                 "alias" : "alias2",

                 "filter" : { "term" : { "user" : "kimchy" } }

            }

        }

    ]

}

通过别名也可以和路由关联，此功能可以和过滤别名命令一起使用，以避免不必要的碎片操作。例如:

请求：POST 'http://localhost:9200/\_aliases

参数：

{

    "actions" : [

        {

            "add" : {

                 "index" : "test",

                 "alias" : "alias1",

                 "routing" : "1"

            }

        }

    ]

}

同时可以指定搜索路由或者查询路由，例如参数：

{

    "actions" : [

        {

            "add" : {

                 "index" : "test",

                 "alias" : "alias2",

                 "search\_routing" : "1,2",

                 "index\_routing" : "2"

            }

        }

    ]

}

注意：搜索路由可以指定多个值，索引路由只能指定一个值。如果使用路由别名的操作同时还有路由参数，则结果是别名路由和路由的交集。例如以下命令将使用“2”作为路由值：

GET http://localhost:9200/alias2/\_search?q=user:kimchy&routing=2,3

#### 通过参数添加别名

语法：PUT /{index}/\_alias/{name}

条件解释：

index：参照的索引，可以使用 \* ， \_all ，正则表达式或者逗号分开的多个name1, name2, …

name：别名的名称。

routing：别名对应的路由。

filter：指定别名时候的过滤条件。

例如：PUT localhost:9200/logs\_201305/\_alias/2013

例如一个索引。

PUT 'localhost:9200/users' -d '{

    "mappings" : {

        "user" : {

            "properties" : {

                "user\_id" : {"type" : "integer"}

            }

        }

    }

}

指定别名：

PUT 'localhost:9200/users/\_alias/user\_12' -d '{

    "routing" : "12",

    "filter" : {

        "term" : {

            "user\_id" : 12

        }

    }

}

#### 建索引的时候同时指定别名

例如：

PUT localhost:9200/logs\_20142801 -d '{

    "mappings" : {

        "type" : {

            "properties" : {

                "year" : {"type" : "integer"}

            }

        }

    },

    "aliases" : {

        "current\_day" : {},

        "2014" : {

            "filter" : {

                "term" : {"year" : 2014 }

            }

        }

    }

}

#### 删除别名

语法：DELETE  /{index}/\_alias/{name}

例如：DELETE localhost:9200/users/\_alias/user\_12

#### 查询现有的别名

可以通过索引名或者别名进行查询。参数：

index：索引别名的名称。部分名称支持通配符，用逗号分隔也可以指定多个索引名称，还可以使用索引的别名名称。

alias：在响应中返回的别名名称。 该参数支持通配符和用逗号分隔的指定多个别名。

ignore\_unavailable：如果一个指定的索引名称不存在该怎么办。如果设置为true，那么这些索引将被忽略。

语法：GET /{index}/\_alias/{alias}

例如：GET localhost:9200/users/\_alias/\*

返回值：

{

  "users" : {

    "aliases" : {

      "user\_13" : {

        "filter" : {

          "term" : {

            "user\_id" : 13

          }

        },

        "index\_routing" : "13",

        "search\_routing" : "13"

      },

      "user\_14" : {

        "filter" : {

          "term" : {

            "user\_id" : 14

          }

        },

        "index\_routing" : "14",

        "search\_routing" : "14"

      },

      "user\_12" : {

        "filter" : {

          "term" : {

            "user\_id" : 12

          }

        },

        "index\_routing" : "12",

        "search\_routing" : "12"

      }

    }

  }

}

下面的例子中包括所有别名为2013的。

GET localhost:9200/\_alias/2013

返回：

{

  "logs\_201304" : {

    "aliases" : {

      "2013" : { }

    }

  },

  "logs\_201305" : {

    "aliases" : {

      "2013" : { }

    }

  }

}

GET localhost:9200/\_alias/2013\_01\*

返回：

{

  "logs\_20130101" : {

    "aliases" : {

      "2013\_01" : { }

    }

  }

}

用HEAD也可以检查别名是否存在，语法和GET类似，例如：

curl -XHEAD -i 'localhost:9200/\_alias/2013'

curl -XHEAD -i 'localhost:9200/\_alias/2013\_01\*'

curl -XHEAD -i 'localhost:9200/users/\_alias/\*'

## 索引设置

*摘要: Elasticsearch 同时支持在运行中对索引的设置，同时支持更改分词库。而且索引还支持索引模板操作。*

**更新索引设置**

在REST风格的url设置中设置 /\_settings (所有索引)或者{index}/\_settings，可以设置一个或者多个索引，例如：

请求：PUT localhost:9200/my\_index/\_settings

参数：

{

    "index" : {

        "number\_of\_replicas" : 4

    }

}

更新分词器

创建索引后可以添加新的分析器。添加分析器之前必须关闭索引，添加之后再打开索引。

POST localhost:9200/myindex/\_close

PUT localhost:9200/myindex/\_settings

参数：

{

  "analysis" : {

    "analyzer":{

      "content":{

        "type":"custom",

        "tokenizer":"whitespace"

      }

    }

  }

}

POST localhost:9200/myindex/\_open

如上示例，先关闭myindex索引，然后添加自定义分析器，分析器策略是空格分析器(whitespace)，就是按照空格进行分词。

**获取索引参数**：

GET http://localhost:9200/twitter/\_settings

获取索引配置参数的请求格式如下：

host:port/{index}/\_settings

host：主机名

port：端口号

{index}：接收多种格式，\* | \_all | name1, name2, …

**过滤配置参数结果**

GET 'http://localhost:9200/twitter/\_settings/name=index.number\_\*'

name=index.number\_\*设置将只返回number\_of\_replicas,number\_of\_shards两个参数详情。

#### ****分词器****

分析(analysis)是这样一个过程：

首先，标记化一个文本块为适用于倒排索引单独的词(term)

然后标准化这些词为标准形式，提高它们的“可搜索性”或“查全率”

这个工作是分析器(analyzer)完成的。一个分析器(analyzer)只是一个包装用于将三个功能放到一起：

字符过滤器

首先字符串经过字符过滤器(character filter)，它们的工作是在标记化前处理字符串。字符过滤器能够去除HTML标记，或者转换"&"为"and"。

分词器

下一步，分词器(tokenizer)被标记化成独立的词。一个简单的分词器(tokenizer)可以根据空格或逗号将单词分开（译者注：这个在中文中不适用）。

标记过滤

最后，每个词都通过所有标记过滤(token filters)，它可以修改词（例如将"Quick"转为小写），去掉词（例如停用词像"a"、"and"``"the"等等），或者增加词（例如同义词像"jump"和"leap"）

Elasticsearch提供很多开箱即用的字符过滤器，分词器和标记过滤器。这些可以组合来创建自定义的分析器以应对不同的需求。

测试分析器：

curl -XGET 'localhost:9200/\_analyze' -d '

{

  "analyzer" : "standard",

  "text" : "this is a test"

}'

该结果将返回”this is a test”使用standard分析器后词的解析情况。

在该分析器下，将会分析称this,is,a,test四个词。

自定义分析器

curl -XGET 'localhost:9200/\_analyze' -d '

{

  "tokenizer" : "keyword",

  "token\_filters" : ["lowercase"],

  "char\_filters" : ["html\_strip"],

  "text" : "this is a <b>test</b>"

}'

使用keyword分词器，lowercase分词过滤，字符过滤器是html\_strip，这3部分构成一个分词器。

上面示例返回分词结果是this is a test，其中html\_strip过滤掉了html字符。

也可以指定索引进行分词。url格式如下：

localhost:9200/test/\_analyze

**索引详情**

如果想获取分析器分析的更多细节，设置explain属性为true(默认是false)，将输出分词器的分词详情。请求格式如下：

请求：GET test/\_analyze

参数：

{

  "tokenizer" : "standard",

  "token\_filters" : ["snowball"],

  "text" : "detailed output",

  "explain" : true,

  "attributes" : ["keyword"]

}

返回结果如下：

{

  "detail" : {

    "custom\_analyzer" : true,

    "charfilters" : [ ],

    "tokenizer" : {

      "name" : "standard",

      "tokens" : [ {

        "token" : "detailed",

        "start\_offset" : 0,

        "end\_offset" : 8,

        "type" : "<ALPHANUM>",

        "position" : 0

      }, {

        "token" : "output",

        "start\_offset" : 9,

        "end\_offset" : 15,

        "type" : "<ALPHANUM>",

        "position" : 1

      } ]

    },

    "tokenfilters" : [ {

      "name" : "snowball",

      "tokens" : [ {

        "token" : "detail",

        "start\_offset" : 0,

        "end\_offset" : 8,

        "type" : "<ALPHANUM>",

        "position" : 0,

        "keyword" : false

      }, {

        "token" : "output",

        "start\_offset" : 9,

        "end\_offset" : 15,

        "type" : "<ALPHANUM>",

        "position" : 1,

        "keyword" : false

      } ]

    } ]

  }

}

#### 索引模板

创建索引模板

索引模板就是创建好一个索引参数设置(settings)和映射（mapping）的模板，在创建新索引的时候指定模板名称就可以使用模板定义好的参数设置和映射。例子如下：

请求：PUT localhost:9200/\_template/template\_1

{

    "template" : "te\*",

    "settings" : {

        "number\_of\_shards" : 1

    },

    "mappings" : {

        "type1" : {

            "\_source" : { "enabled" : false }

        }

    }

}

定义好模板可使用te\*来适配，分片数量为1，默认文档类型type1，\_source的enabled为false。

也可以在模板中定义别名等其他属性。

删除索引模板

curl -XDELETE localhost:9200/\_template/template\_1

template\_1为之前创建的索引模板名称。

获取索引模板

curl -XGET localhost:9200/\_template/template\_1

使用通配符或逗号分隔符

curl -XGET localhost:9200/\_template/temp\*

curl -XGET localhost:9200/\_template/template\_1,template\_2

获取所有索引模板

curl -XGET localhost:9200/\_template/

判断索引模板是否存在

curl -XHEAD -i localhost:9200/\_template/template\_1

多个模板匹配

有这样一种情况，template\_1,template\_2两个模板，使用te\*会匹配2个模板，最后合并两个模板的配置，如果配置重复，这时应该设置order属性，order是从0开始的数字，先匹配order数字小的，再匹配数字大的，如果有相同的属性配置，后匹配的会覆盖之前的配置

## DSL Span查询和查询重写参数

*摘要: SpanQuery是按照词在文章中的距离或者查询几个相邻词的查询。打个比方：如“中华人民共和国” 用“中国“做为关键字， 跨度为某个值，如5。跨度代表 中 和国之间的长度。这是比较底层的位置查询，通常用于实现非常具体的法律文件或专利的查询。*

    SpanQuery是按照词在文章中的距离或者查询几个相邻词的查询。打个比方：如“中华人民共和国”    用“中国“做为关键字， 跨度为某个值，如5。跨度代表 中 和国之间的长度。这是比较底层的位置查询，通常用于实现非常具体的法律文件或专利的查询。处了span\_multi查询外，跨度查询和非跨度查询不能在一起混合查询。跨度查询包括以下几种。

span\_term查询：词距查询的基础，结果和TermQuery相似，只不过是增加了查询结果中单词的距离信息。例如：

{

    "span\_term" : { "user" : "kimchy" }

}

{

    "span\_term" : { "user" : { "value" : "kimchy", "boost" : 2.0 } }

}

{

    "span\_term" : { "user" : { "term" : "kimchy", "boost" : 2.0 } }

}

span\_multi查询：可以包含Term，范围，前缀，通配符，正则表达式，或模糊查询的组合查询。例如：

{

    "span\_multi":{

        "match":{

            "prefix" : { "user" :  { "value" : "ki" } }

        }

    }

}

{

    "span\_multi":{

        "match":{

            "prefix" : { "user" :  { "value" : "ki", "boost" : 1.08 } }

        }

    }

}

span\_first查询：在指定距离可以找到第一个单词的查询。例如：

{

    "span\_first" : {

        "match" : {

            "span\_term" : { "user" : "kimchy" }

        },

        "end" : 3

    }

}

span\_near查询：查询的几个语句之间保持者一定的距离。例如：

{

    "span\_near" : {

        "clauses" : [

            { "span\_term" : { "field" : "value1" } },

            { "span\_term" : { "field" : "value2" } },

            { "span\_term" : { "field" : "value3" } }

        ],

        "slop" : 12,

        "in\_order" : false,

        "collect\_payloads" : false

    }

}

clauses表示一个或多个其他的跨越式查询。

slop控制插入的允许最大数目的位置。

span\_or查询：同时查询几个词句查询。

{

    "span\_or" : {

        "clauses" : [

            { "span\_term" : { "field" : "value1" } },

            { "span\_term" : { "field" : "value2" } },

            { "span\_term" : { "field" : "value3" } }

        ]

    }

}

span\_not查询：从一个词距查询结果中，去除一个词距查询。

{

    "span\_not" : {

        "include" : {

            "span\_term" : { "field1" : "hoya" }

        },

        "exclude" : {

            "span\_near" : {

                "clauses" : [

                    { "span\_term" : { "field1" : "la" } },

                    { "span\_term" : { "field1" : "hoya" } }

                ],

                "slop" : 0,

                "in\_order" : true

            }

        }

    }

}

span\_containing查询：返回在另一个范围内查询的匹配结果，从大到小包含。例如：

{

    "span\_containing" : {

        "little" : {

            "span\_term" : { "field1" : "foo" }

        },

        "big" : {

            "span\_near" : {

                "clauses" : [

                    { "span\_term" : { "field1" : "bar" } },

                    { "span\_term" : { "field1" : "baz" } }

                ],

                "slop" : 5,

                "in\_order" : true

            }

        }

    }

}

little和big可为任意跨度类型查询。

span\_within查询：返回在另一个范围内查询的匹配结果，从小到大包含。例如：

{

    "span\_within" : {

        "little" : {

            "span\_term" : { "field1" : "foo" }

        },

        "big" : {

            "span\_near" : {

                "clauses" : [

                    { "span\_term" : { "field1" : "bar" } },

                    { "span\_term" : { "field1" : "baz" } }

                ],

                "slop" : 5,

                "in\_order" : true

            }

        }

    }

}

#### 查询重写机制

    在多条件查询时，如通配符和前缀，最终会有一个查询重写的过程。系统可以通过参数控制他们用何种方式进行重写。

constant\_score：系统默认参数，和constant\_score\_boolean 类似，如果有少量匹配项则显示匹配的文档，否则按顺序访问所有匹配项并标记文档，匹配的文档会分配一个查询到的固定分值。

scoring\_boolean :该重写方法将对应的关键词转换成布尔查询的布尔Should子句，它有可能是CPU密集型的(因为每个关键词都需要计算得分)，而且如果关键词数量太多，则会超出布尔查询的限制，限制条件的上限是1024。与此同时，该类型的查询语句还保存计算的得分。布尔查询的默认数量限制可以通过修改elasticsearch.yml文件中的index.query.bool.max\_clause\_count属性值来修改。但始终要记住的是，产生的布尔查询子句越多，查询的性能越低。

constant\_score\_boolean:该重写方法与上面提到的scoring\_boolean重写方法类似，但是CPU消耗要低很多，因为它不计算得分，每个关键词的得分就是查询的权重，默认是1，也可以通过权重属性来设置其它的值。与scoring\_boolean重写方法类似，该方法也受到布尔查询数量的限制。

top\_terms\_N:该重写方法将对应的关键词转换成布尔查询的布尔Should子句，同时保存计算得分。只是与scoring\_boolean不同点在于，它只保留前N个关键词，来避免触发布尔子句数量的上限。

top\_terms\_boost\_N:该重写方法与top\_terms\_N类似，只是得分的计算只与权重有关，与查询词无关。

top\_terms\_blended\_freqs\_N：该重写方法将对应的关键词转换成布尔查询的布尔Should子句，如果他们有相同的频率则计算所有的查询分数，所使用的频率是所有匹配项的最大频率。此重写方法只使用最高评分条款，所以它不会溢出布尔最大子句计数。N参数控制使用的最高评分条款的大小。

## Es安装成服务

*摘要: Elasticsearch创建了debian安装包和rpm安装包，可以在官网的下载页面中进行下载。安装包需要依赖JAVA，除此就没有任何依赖。*

### linux下作为服务

    Elasticsearch创建了debian安装包和rpm安装包，可以在官网的下载页面中进行下载。安装包需要依赖JAVA，除此就没有任何依赖。

    在debian系统下可以使用 标准的系统工具，init脚本放在 /etc/init.d/elasticsearch下，配置文件默认放在 /etc/default/elasticsearch下。从Debian软件包安装好后默认是不启动服务的。其原因是为了防止实例不小心加入群集。安装好后用dpkg -i命令来确保，当系统启动后启动Elasticsearch需要运行下面的两个命令：

sudo update-rc.d elasticsearch defaults 95 10

sudo /etc/init.d/elasticsearch start

当用户运行Debian8或者Ubuntu14或者更高版本的时候，系统需要用systemd 来代替update-rc.d，在这种情况下，请使用systemd来运行，参见下面的介绍。

Elasticsearch通常的建议是使用Oracle的JDK。可以用下面的命令安装。

sudo add-apt-repository ppa:webupd8team/java

sudo apt-get update

sudo apt-get install oracle-java8-installer

java -version

基于RPM的系统一般使用chkconfig来启用和禁用服务。init脚本位于/etc/init.d/elasticsearch下，配置文件放在/etc/sysconfig/Elasticsearch下。同Debian系统类似，安装好后也不会自动加入自启动服务中。需要手工指定。

sudo /sbin/chkconfig --add elasticsearch

sudo service elasticsearch start

systemd服务启动。

很多linux系统，例如Debian Jessie, Ubuntu 14等，系统不使用chkconfig来注册服务，取而代之的是用systemd来启动和停止服务。命令是 /bin/systemctl 来启动和停止服务。rpm包安装的配置文件在/etc/sysconfig/elasticsearch下，deb包安装的配置文件在 /etc/default/elasticsearch下。安装RPM之后，你必须改变系统配置，然后启动Elasticsearch。

sudo /bin/systemctl daemon-reload

sudo /bin/systemctl enable elasticsearch.service

sudo /bin/systemctl start elasticsearch.service

同时注意改变在/etc/sysconfig/elasticsearch中的MAX\_MAP\_COUNT设置是没有任何效果的。需要改变/usr/lib/sysctl.d/elasticsearch.conf中的配置才起作用。

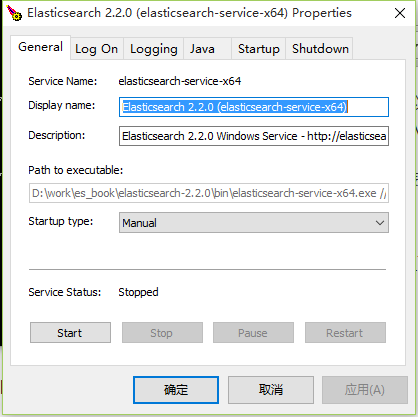
### window下作为服务

Windows用户可以配置Elasticsearch作为服务运行在后台运行，或在没有任何用户交互启动时自动启动。这可以通过bin目录下的service.bat脚本来实现，可以安装，卸载，管理或配置服务命令行为：service.bat install|remove|start|stop|manager [SERVICE\_ID]

SERVICE\_ID是服务id可以不用指定用默认的值，系统可以安装多个服务。manager是启动图形界面的配置。例如运行：service install后显示的内容如下  
Installing service      :  "elasticsearch-service-x64"  
Using JAVA\_HOME (64-bit):  "C:\Program Files\Java\jdk1.7.0\_79"  
The service 'elasticsearch-service-x64' has been installed.

安装好后，有两种方法可以对服务进行设置。

1、图形化界面，可以用命令service manager来启动图形界面。执行后显示界面如下。



2、用命令service start ，service stop来启动停止服务。

还有一个社区支持的可定制MSI安装程序：https://github.com/salyh/elasticsearch-msi-installer也可以安装成服务

## 重建索引（拷贝文件等）

*摘要: 重建索引是2.3.0新增加的接口。重建索引的最基本的功能是拷贝文件从一个索引到另一个索引，在很多时候还是比较有意义的。*

重建索引是2.3.0新增加的接口。这个接口是实验性质的，在未来有可能会改变。

重建索引的最基本的功能是拷贝文件从一个索引到另一个索引，例如：

POST /\_reindex

{

"source": {

"index": "secisland"

},

"dest": {

"index": "new\_secisland"

}

}

返回的内容如下：

{

"took" : 639,

"updated": 112,

"batches": 130,

"version\_conflicts": 0,

"failures" : [ ],

"created": 12344

}

took：从开始到结束的整个操作的毫秒数。

updated：已成功更新的文档数。

created：成功创建的文档数。

batches：拉回来从建索引的数量。

version\_conflicts：重建索引中版本冲突数的数量。

failures：所有索引失败的数组。如果这是非空的，则请求将被中止。

    由于\_reindex是获取源索引的快照，而且目标索引是不同的索引，随意基本上不太可能产生冲突。在接口参数中可以增加dest来进行乐观并发控制。如果version\_type设置为internal会导致Elasticsearch盲目转储文件到目标索引，任何具有相同的类型和ID的文档将被从写。例如：

POST /\_reindex

{

"source": {

"index": "twitter"

},

"dest": {

"index": "new\_twitter",

"version\_type": "internal"

}

}

    如果设置version\_type为external将会导致Elasticsearch保护源索引的版本，如果在目标索引中有一个比源索引旧的版本，则会更新文档。对于源文件中丢失的文档在目标中也会被创建。

POST /\_reindex

{

"source": {

"index": "twitter"

},

"dest": {

"index": "new\_twitter",

"version\_type": "external"

}

}

    设置op\_type为create将导致\_reindex在目标索引中仅创建丢失的文件。所有现有的文件将导致版本冲突。

POST /\_reindex

{

"source": {

"index": "twitter"

},

"dest": {

"index": "new\_twitter",

"op\_type": "create"

}

}

    正常情况下当发生冲突的时候\_reindex过程将被终止，可以在请求体中设置 "conflicts": "proceed" 可以只进行计算。

POST /\_reindex

{

"conflicts": "proceed",

"source": {

"index": "twitter"

},

"dest": {

"index": "new\_twitter",

"op\_type": "create"

}

}

    可以通过向源添加一个类型或者增加一个查询来限制文档的数量，比如只复制用户名为kimchy 的文档。

POST /\_reindex

{

"source": {

"index": "twitter",

"type": "tweet",

"query": {

"term": {

"user": "kimchy"

}

}

},

"dest": {

"index": "new\_twitter"

}

}

    在请求接口中可以列出源索引和类型，可以在一个接口中复制多个源。例如下面的例子将在twitter和blog索引中的tweet和post类型中拷贝数据，这包括twitter索引中的"tweet"和"post"类型，也包括blog索引中的"tweet"和"post"类型。如果需要更具体的文档可以使用查询。当id产生冲突的时候是没有办法处理的，因为执行的顺序是随机的，所以目标索引将无法确定应该保存哪些文档。

POST /\_reindex

{

"source": {

"index": ["twitter", "blog"],

"type": ["tweet", "post"]

},

"dest": {

"index": "all\_together"

}

}

    也可以通过设置大小来限制处理文档的数量。这只会复制一个文件到new\_twitter索引中。

本文由赛克 蓝德(secisland)原创，转载请标明作者和出处。

POST /\_reindex

{

"size": 1,

"source": {

"index": "twitter"

},

"dest": {

"index": "new\_twitter"

}

}

    如果你想要复制特定的文档，可以使用排序。排序会降低效率，但在某些情况下，它是有意义的。如果可能的话，可以选择性的查询来确定复制的大小和排序。下面将从twitter索引中复制10000文档到new\_twitter中。

POST /\_reindex

{

"size": 10000,

"source": {

"index": "twitter",

"sort": { "date": "desc" }

},

"dest": {

"index": "new\_twitter"

}

}

    \_reindex同时支持使用脚本来修改文档。例如：

POST /\_reindex

{

"source": {

"index": "twitter",

},

"dest": {

"index": "new\_twitter",

"version\_type": "external"

}

"script": {

"internal": "if (ctx.\_source.foo == 'bar') {ctx.\_version++; ctx.\_source.remove('foo')}"

}

}

    可以修改的元字段包括\_id、\_type、\_index、\_version、\_routing、\_parent、\_timestamp、\_ttl。所以用脚本复制能力非常强大。

    默认情况下，如果\_reindex可以设置文档路由来进行路由保存除非是通过脚本进行了改变。你可以设置dest 参数来进行路由设置：

keep：设置在匹配的每一个路由上发送请求的路由。这个是默认设置。

discard：为每一个匹配发送的请求设置为空。

=<some text>：设置为每一个匹配的文本请求发送路由上的路由。例如，您可以使用以下要求复制所有文件从源索引为"company"等于"cat"的查询中复制到目标索引，目标索引的路由设置为cat。

POST /\_reindex

{

"source": {

"index": "source"

"query": {

"match": {

"company": "cat"

}

}

}

"dest": {

"index": "dest",

"routing": "=cat"

}

}

    默认情况下\_reindex每次处理的大小为100。可以在源参数中用size参数来进行修改：

POST /\_reindex

{

"source": {

"index": "source",

"size": 1500

},

"dest": {

"index": "dest"

}

}

URL参数：

    \_reindex接口除了接收标准的参数例如pretty，还支持refresh, wait\_for\_completion, consistency, timeout参数。

在url参数中发送refresh会导致请求写的所有索引都将被刷新。这个时候收到新的数据也被索引。

如果请求包含wait\_for\_completion = FALSE，Elasticsearch将进行执行前检查后启动请求，然后返回一个可用于任务API取消或得到任务的状态，现在一旦请求完成任务就不见了，找任务的最终结果的唯一地方是在Elasticsearch日志文件中，这个问题将会在未来的版本中修复。

consistency控制每次写请求必须多少份分片被响应。

timeout控制每个写请求等待可用的分片的时间。

任务查看：

GET /\_tasks/?pretty&detailed=true&actions=\*reindex

返回的结果类似

{

"nodes" : {

"r1A2WoRbTwKZ516z6NEs5A" : {

"name" : "Tyrannus",

"transport\_address" : "127.0.0.1:9300",

"host" : "127.0.0.1",

"ip" : "127.0.0.1:9300",

"attributes" : {

"testattr" : "test",

"portsfile" : "true"

},

"tasks" : {

"r1A2WoRbTwKZ516z6NEs5A:36619" : {

"node" : "r1A2WoRbTwKZ516z6NEs5A",

"id" : 36619,

"type" : "transport",

"action" : "indices:data/write/reindex",

"status" : {

"total" : 6154,

"updated" : 3500,

"created" : 0,

"deleted" : 0,

"batches" : 36,

"version\_conflicts" : 0,

"noops" : 0

},

"description" : ""

}

}

}

}

}

\_reindex可以用来建立索引的时候同时也可以修改列的名称，例如，源索引类型为：

POST test/test/1?refresh&pretty

{

"text": "words words",

"flag": "foo"

}

可以通过\_reindex来修改列的名称，例如：

POST \_reindex?pretty

{

"source": {

"index": "test"

},

"dest": {

"index": "test2"

},

"script": {

"inline": "ctx.\_source.tag = ctx.\_source.remove(\"flag\")"

}

}

然后我们看一下test2的结构，可以看出flag字段名称修改成了tag。

GET test2/test/1?pretty

{

"text": "words words",

"tag": "foo"

}

## 脚本

1、直接在请求体中使用脚本。

2、把脚本存储在索引中，通过引用脚本id来使用。

3、把脚本存储在本地磁盘中，默认的位置为：elasticsearch\config\scripts，通过引用脚本名称进行使用。

*摘要: Elasticsearch的脚本模块可以使用脚本对Elasticsearch的字段进行再次处理。例如，可以用来重新评估查询的自定义得分，可以对索引中的某个字段再次加工处理。脚本默认支持groovy，其他语言可以通过插件来支持。*

Elasticsearch的脚本模块可以使用脚本对Elasticsearch的字段进行再次处理。例如，可以用来重新评估查询的自定义得分，可以对索引中的某个字段再次加工处理。

脚本默认是关闭的，如果这个时候执行脚本会报以下错误：

{

"error": {

"root\_cause": [

{

"type": "remote\_transport\_exception",

"reason": "[Left-Winger][127.0.0.1:9300][indices:data/write/update[s]]"

}],

"type": "illegal\_argument\_exception",

"reason": "failed to execute script",

"caused\_by": {

"type": "script\_exception",

"reason": "scripts of type [indexed], operation [update] and lang [groovy] are disabled"

}

},

"status": 400

}

可以通过配置来启用脚本引擎，配置的位置在elasticsearch.yml文件中添加如下内容:

script.inline: true

script.indexed: true

对这些设置，有三种配置值：

|  |  |
| --- | --- |
| 值 | 描述 |
| false | 完全禁用脚本 |
| true | 启用脚本 |
| sandbox | 脚本仅可以用沙盒语言执行 |

**表格 8.7 细粒度动态脚本设置**

默认配置值为：

script.inline: sandbox

script.indexed: sandbox

script.file: true

也可以在已下操作中执行脚本：

|  |  |
| --- | --- |
| 值 | 描述 |
| aggs | 聚合 |
| search | 搜索接口、过滤接口或建议接口（例如filters、script\_fields）。 |
| update | 升级接口 |
| plugin | 通用plugin类别下，脚本使用的任何插件 |

插件也可以进行自定义操作，利用这种格式：${pluginName}\_${operation}。例如在任何脚本引擎上禁用更新和映射：

script.update: false

script.mapping: false

也支持明确的脚本语言设置，需要script.engine.<engine>前缀，优先权高于其他通用设置：

script.engine.groovy.file.aggs: true

script.engine.groovy.file.mapping: true

script.engine.groovy.file.search: true

script.engine.groovy.file.update: true

script.engine.groovy.file.plugin: true

script.engine.groovy.indexed.aggs: true

script.engine.groovy.indexed.mapping: false

script.engine.groovy.indexed.search: true

script.engine.groovy.indexed.update: false

script.engine.groovy.indexed.plugin: false

script.engine.groovy.inline.aggs: true

script.engine.groovy.inline.mapping: false

script.engine.groovy.inline.search: false

script.engine.groovy.inline.update: false

script.engine.groovy.inline.plugin: false

系统的脚本模块默认使用groovy作为的脚本语言。可以通过设置script.default\_lang可以进行修改。

可以用语言插件来支持不同的语言脚本。所以在使用脚本接口的参数中提供了lang参数定义脚本语言。

|  |  |  |
| --- | --- | --- |
| 语言 | 沙盒 | 必要插件 |
| groovy | 否 | 内置 |
| expression | 是 | 内置 |
| mustache | 是 | 内置 |
| javascript | 否 | elasticsearch-lang-javascript |
| python | 否 | elasticsearch-lang-python |

为了增加安全性，Elasticsearch不允许在请求中指定非沙盒语言。默认的脚本目录位置可以在elasticsearch.yml中设置path.scripts进行修改。放置在这个目录中的脚本会自动选中并且可以被使用。一旦脚本被放在这个目录中，可以通过脚本名进行引用。

### 1.1.1 脚本使用

在Elasticsearch中使用脚本有三种方式：

1、直接在请求体中使用脚本。

2、把脚本存储在索引中，通过引用脚本id来使用。

3、把脚本存储在本地磁盘中，默认的位置为：elasticsearch\config\scripts，通过引用脚本名称进行使用。

下面通过举例来说明使用脚本的三种方法。

首先建一个索引，添加一条数据。

PUT localhost:9200/secisland/secilog/1

{

"eventCount":1,

"eventName" :"linux login event"

}

下面我们用脚本对eventCount做加法操作。

n 用第一种方法直接在请求中执行脚本：

http://127.0.0.1:9200/secisland/secilog/1/\_update

{

    "script" : "ctx.\_source.eventCount+=count",

    "params" : {

        "count" : 4

    }

}

n 下面我们用第二种方法操作，在用第二种方法前，先要把脚本存储在Elasticsearch中：

POST 127.0.0.1:9200/\_scripts/groovy/indexedCalculateCount

{

     "script": "ctx.\_source.eventCount+=count"

}

然后通过脚本Id进行文档操作：

POST http://127.0.0.1:9200/secisland/secilog/1/\_update

{

  "script": {

    "id": "indexedCalculateCount",

    "lang": "groovy",

    "params": {

      "count": 8

    }

  }

}

n 下面我们用第三种方法操作，在第三种方法操作前，先要把脚本存储在文件中，文件名为indexedCalculateCount.groovy,文件中的内容为：ctx.\_source.eventCount+=count。

注意：Elasticsearch对文件读取有个时间，刚建好后，不能生效，做验证的时候可以重启进行生效。

POST http://127.0.0.1:9200/secisland/secilog/1/\_update/

{

  "script": {

    "file": "indexedCalculateCount",

    "lang": "groovy",

    "params": {

      "count": 8

    }

  }

}

### 1.1.2 脚本配置

A. **索引脚本**

Elasticsearch可以在名为\_scripts的内部索引中存储脚本，并且通过id进行引用。脚本请求的格式：/\_scripts/{lang}/{id}

lang表示脚本语言，id表示脚本编号。

1. 保存脚本

POST localhost:9200/\_scripts/groovy/indexedCalculateScore

{

     "script": "log(\_score \* 2) + my\_modifier"

}

2. 使用脚本

POST localhost:9200/\_search

{

  "query": {

    "function\_score": {

      "query": {

        "match": {

          "body": "foo"

        }

      },

      "functions": [

        {

          "script\_score": {

            "script": {

              "id": "indexedCalculateScore",

              "lang" : "groovy",

              "params": {

                "my\_modifier": 8

              }

            }

          }

        }

      ]

    }

  }

}

3. 查看脚本

GET localhost:9200/\_scripts/groovy/indexedCalculateScore

4. 删除脚本

DELETE localhost:9200/\_scripts/groovy/indexedCalculateScore

B. **启用动态脚本**

在应用或代理后面执行Elasticsearch，可以从外界保护Elasticsearch。如果允许用户运行内联脚本或索引脚本，会继承运行Elasticsearch的用户权限。因此动态脚本默认仅支持沙盒语言。

首先，应该使用root用户权限运行Elasticsearch，可以允许脚本在服务器上访问或做任何事情。其次，不应该让用户直接访问Elasticsearch，而是要有一个中间代理应用。如果确实想要用户直接访问Elasticsearch，需要决定是否足够信任用户来运行脚本。

C. **脚本自动重载**

周期性扫描config/scripts目录的修改。新的和修改的脚本会被重载删除的脚本会从预加载脚本缓存中移除。重载频率可以使用resource.reload.interval设置指定，默认值为60s。设置script.auto\_reload\_enabled为false可以完全禁用脚本重载。

本文由赛克 蓝德(secisland)原创，转载请标明作者和出处。

D. **本地（Java）脚本**

有时，groovy和expressions是不够的。在这种情况下，可以执行本地脚本。

执行本地脚本的最好方式是编写并且安装一个插件。

为了注册实际脚本，需要引用NativeScriptFactory来构成脚本。实际脚本需要继承AbstractExecutableScript或AbstractSearchScript。第二种可能是最有用的并且有多个可以继承的子类，比如AbstractLongSearchScript、AbstractDoubleSearchScript和AnstractFloatSearchScript。最后，插件需要通过onModule(ScriptModule)方法注册本地脚本。

public class MyNativeScriptPlugin extends Plugin {

    @Override

    public String name() {

        return "my-native-script";

    }

    @Override

    public String description() {

        return "my native script that does something great";

    }

    public void onModule(ScriptModule scriptModule) {

        scriptModule.registerScript("my\_script", MyNativeScriptFactory.class);

    }

    public static class MyNativeScriptFactory implements NativeScriptFactory {

        @Override

        public ExecutableScript newScript(@Nullable Map<String, Object> params) {

            return new MyNativeScript();

        }

        @Override

        public boolean needsScores() {

            return false;

        }

    }

    public static class MyNativeScript extends AbstractFloatSearchScript {

        @Override

        public float runAsFloat() {

            float a = (float) source().get("a");

            float b = (float) source().get("b");

            return a \* b;

        }

    }

}

可以执行脚本通过指定lang为native，指定inline为脚本名。

POST localhost:9200/\_search

{

  "query": {

    "function\_score": {

      "query": {

        "match": {

          "body": "foo"

        }

      },

      "functions": [

        {

          "script\_score": {

            "script": {

                "inline": "my\_script",

                "lang" : "native"

            }

          }

        }

      ]

    }

  }

}

E. **Lucene表达式脚本**

**警告：Lucene表达式模块正经历重大的发展，展现的功能可能会被修改。**

Lucene的表达式模块提供了一个机制来将Javascript表达式编译成字节码。表达式脚本可以用于script\_score、script\_fields、排序脚本和数字型聚合脚本。

表达式脚本变量可以接受：

l 单值文档字段，例如doc[‘myfield’].value，也可以写作doc[‘myfield’]。

l 传给脚本的参数，例如mymodifier。

l 当前文档的得分，\_score（只有用在script\_score的时候）。

表达式脚本日期类型可以使用这些方法：

l getYear()

l getMonth()

l getDayOfMonth()

l getHourOfDay()

l getMinutes()

l getSeconds()

比如，获取日期字段间年份的不同：

doc[‘date1’].getYear() – doc[‘date0’].getYear()

相对于其他脚本语言，有一些限制：

l 只接受数字型字段

l 保存的字段不可用

l 如果只有一部分文档包含字段值，缺失字段值的文档默认值为0。

F. **得分**

聚合中所有可以使用的脚本，当前文档的得分可以用\_score获得。

G. **文档字段**

大多数脚本围绕指定文档字段数据的使用。doc[‘field\_name’]可以用来访问文档内指定字段数据。注意，只能是简单值字段（不能返回Json对象）并且只有不分词字段或单索引词字段是有意义的。

可以从字段中获取下列数据：

|  |  |
| --- | --- |
| 表达式 | 描述 |
| doc[‘field\_name’].value | 字段的本地值。 |
| doc[‘field\_name’].values | 字段的本地数组值。如果字段没有值，返回一个空数组。 |
| doc[‘field\_name’].empty | 布尔值，表示文档中的字段是否有值。 |
| doc[‘field\_name’].multiValued | 布尔值，表示文集中有多个值的字段。 |
| doc[‘field\_name’].lat | 地理点类型的纬度。 |
| doc[‘field\_name’].lon | 地理点类型的经度。 |
| doc[‘field\_name’].lats | 地理点类型的纬度（复数）。 |
| doc[‘field\_name’].lons | 地理点类型的经度（复数）。 |
| doc[‘field\_name’].distance(lat,lon) | 地理点字段和提供的经纬度之间的平面距离（米）。 |
| doc[‘field\_name’].distanceWithDefault(lat,lon,default) | 地理点字段从提供的经纬度和默认值的平面距离（米）。 |
| doc[‘field\_name’].distanceInMiles(lat,lon) | 地理点字段和提供的经纬度之间的平面距离（英里）。 |
| doc[‘field\_name’].distanceInMilesWithDefault(lat,lon,default) | 地理点字段从提供的经纬度和默认值的平面距离（英里）。 |
| doc[‘field\_name’].distanceInKm(lat,lon) | 地理点字段和提供的经纬度之间的平面距离（千米）。 |
| doc[‘field\_name’].distanceInkmWithDefault(lat,lon,default) | 地理点字段从提供的经纬度和默认值的平面距离（千米）。 |
| doc[‘field\_name’].arcDistance(lat,lon) | 地理点字段和提供的经纬度之间的天穹距离（米）。 |
| doc[‘field\_name’].arcDistanceWithDefault(lat,lon,default) | 地理点字段从提供的经纬度和默认值的天穹距离（米）。 |
| doc[‘field\_name’].arcDistanceInMiles(lat,lon) | 地理点字段和提供的经纬度之间的天穹距离（英里）。 |
| doc[‘field\_name’].arcDistanceInMilesWithDefault(lat,lon,default) | 地理点字段从提供的经纬度和默认值的天穹距离（英里）。 |
| doc[‘field\_name’].arcDistanceInKm(lat,lon) | 地理点字段和提供的经纬度之间的天穹距离（千米）。 |
| doc[‘field\_name’].arcDistanceInKmWithDefault(lat,lon,default) | 地理点字段从提供的经纬度和默认值的天穹距离（千米）。 |
| doc[‘field\_name’].factorDistance(lat,lon) | 地理点字段从提供的经纬度之间的距离因子。 |
| doc[‘field\_name’].factorDistance(lat,lon,default) | 地理点字段从提供的经纬度和默认值的距离因子。 |
| doc[‘field\_name’].geohashDistance(geohash) | 地理点字段从提供的地理散列的天穹距离（米）。 |
| doc[‘field\_name’].geohashDistanceInKm(geohash) | 地理点字段从提供的地理散列的天穹距离（千米）。 |
| doc[‘field\_name’].geohashDistanceInMiles(geohash) | 地理点字段从提供的地理散列的天穹距离（英里）。 |

**表格 8.9 脚本提取数据**

H. **保存的字段**

执行脚本的时候，保存的字段也可以访问。通过使用\_fields[‘my\_field\_name’].value或\_fields[‘my\_field\_name’].values。

I. **在脚本中访问文档的得分**

当使用脚本计算文档得分（例如，通过function\_score查询），可以在groovy脚本中使用\_score变量访问得分。

J. **源字段**

当执行脚本的时候，可以访问源字段。每个文档都会加载源字段、分析并且提供给脚本进行评估。使用方式，形如\_source.obj1.obj2.field3。

K. **Groovy内置方法**

|  |  |
| --- | --- |
| 方法 | 描述 |
| sin(a) | 返回角度的三角正弦 |
| cos(a) | 返回角度的三角余弦 |
| tan(a) | 返回角度的三角正切 |
| asin(a) | 返回反正弦值 |
| acos(a) | 返回反余弦值 |
| atan(a) | 返回反正切值 |
| toRadians(angdeg) | 转换角度度量近似为弧度 |
| toDegrees(angrad) | 转换弧度度量近似为角度 |
| exp(a) | 返回欧拉数e的指数值 |
| log(a) | 返回值的自然对数（基于e） |
| log10(a) | 返回值基于10的对数 |
| sqrt(a) | 返回值的平方根 |
| cbrt(a) | 返回双精度值的立方根 |
| IEEEremainder(f1,f2) | 进行余数运算 |
| ceil(a) | 向下取整 |
| floor(a) | 向上取整 |
| rint(a) | 取整 |
| atan2(y,x) | 转换直角坐标（x,y）到极坐标（r,θ），返回θ值 |
| pow(a,b) | 返回ab |
| round(a) | 取整 |
| random() | 返回随机双精度数 |
| abs(a) | 返回值的绝对值 |
| max(a,b) | 返回大值 |
| min(a,b) | 返回小值 |
| ulp(d) | 返回参数的ulp的大小 |
| signum(d) | 返回自变量的符号函数 |
| sinh(x) | 返回值的双曲正弦值 |
| cosh(x) | 返回值的双曲余弦值 |
| tanh(x) | 返回值的双曲切线 |
| hypot(x,y) | 返回sqrt(x2+y2) |

## 分片错误failed shard recovery解决方法

今天早上Elasticsearch莫名奇怪的报错了。整个控制台不停的打印出

[2016-06-07 08:55:00,976][WARN ][indices.cluster          ] [secilog] [[seci-201  
606][1]] marking and sending shard failed due to [failed recovery]

org.elasticsearch.index.gateway.IndexShardGatewayRecoveryException: [seci-201606  
][1] failed recovery  at org.elasticsearch.index.gateway.IndexShardGatewayService$1.run(IndexS  
hardGatewayService.java:162)

[2016-06-07 08:40:41,967][WARN ][cluster.action.shard     ] [secilog] [seci-201606][1] received shard failed for [seci-201606][1], node[xGIRwM34TdWcKgV95BI74g], [P], s[INITIALIZING], indexUUID [t75Egn6rT2WyBYQk\_2X3IQ], reason [shard failure [failed recovery][IndexShardGatewayRecoveryException[[seci-201606][1] failed recovery]; nested: EngineCreationFailureException[[seci-201606][1] failed to create engine]; nested: CorruptIndexException[codec footer mismatch: actual footer=-464453631 vs expected footer=-1071082520 (resource: MMapIndexInput(path="D:\secisland\secilog\elasticsearch\data\192.168.21.5\nodes\0\indices\seci-201606\1\index\\_6exw.cfs"))]; ]]

通过错误内容大概可以看出是数据的一个分片出了问题。这个问题的解决在网上比较少，特在此介绍一下。

检查：localhost:9200/\_cluster/health

从内容上可以看出有红色的状态，表示出了比较严重的问题。

问题解决：

先停掉Elasticsearch。

转到Elasticsearch的lib目录

在目录下执行java -cp lucene-core-4.10.4.jar -ea:org.apache.lucene... org.apache.lucene.index.CheckIndex D:\secisland\secilog\elasticsearch\data\192.168.21.5\nodes\0\indices\seci-201606\1\index -fix

然后重启看下是否好了。如果还没有好可以再次停掉Elasticsearch。

到D:\secisland\secilog\elasticsearch\data\192.168.21.5\nodes\0\indices\seci-201606\1\translog目录。

删掉里面的\*.recovery文件。

然后重启Elasticsearch。

这样就解决了分片损坏的问题

## Watcher插件

*摘要: Watcher是对Elasticsearch进行告警和通知的插件，可以根据数据的变化采取行动。它的设计原理是在Elasticsearch中执行查询，满足条件的情况下，产生告警。简单地定义查询、设置限定条件、设置预定计划和将要进行的操作后，Watcher会自动完成剩下的操作。*

Watcher是对Elasticsearch进行告警和通知的插件，可以根据数据的变化采取行动。它的设计原理是在Elasticsearch中执行查询，满足条件的情况下，产生告警。简单地定义查询、设置限定条件、设置预定计划和将要进行的操作后，Watcher会自动完成剩下的操作。

### 1.1.1 ****安装****

Watcher是以Elasticsearch插件的形式存在的。所以安装的过程和插件安装基本一致。

安装步骤：

1) 在Elasticsearch中安装Watcher插件。

bin/plugin install license

bin/plugin install watcher

2) 权限设置，在安装watcher的时候会提示额外权限说明如下：

@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

@     WARNING: plugin requires additional permissions     @

@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

\* java.lang.RuntimePermission getClassLoader

\* java.lang.RuntimePermission setContextClassLoader

\* java.lang.RuntimePermission setFactory

See http://docs.oracle.com/javase/8/docs/technotes/guides/security/permissions.html

for descriptions of what these permissions allow and the associated risks.

Continue with installation? [y/N]y

3) 如果在Elasticsearch中设置了禁止自动创建索引的设置，需要在配置文件中添加如下设置：action.auto\_create\_index: .watches,.triggered\_watches,.watcher-history\*

4) 启动Elasticsearch

bin/elasticsearch

5) 验证是否安装成功

GET http://localhost:9200/\_watcher/stats?pretty

返回内容为：

{

  "watcher\_state": "started",

  "watch\_count": 0,

  "execution\_thread\_pool": {

    "queue\_size": 0,

    "max\_size": 0

  }

}

### 1.1.2 ****结构****

一个watch的配置主要包括trigger, input, condition, actions几个部分构成，此外，在执行actions之前，您可以定义transforms过程中处理告警中的内容。

|  |  |
| --- | --- |
| 参数 | 描述 |
| Trigger | 确定什么时候告警进行检查，一个告警中至少有一个Trigger |
| Input | 将数据加载到告警中的内容。如果没有指定输入，则加载一个空的内容。 |
| Condition | 控制是否执行告警操作。如果没有指定条件，条件默认总是执行。 |
| Transform | 您可以定义在监视级别上的转换，也可以定义特定于动作的转换，可选条件。 |
| Actions | 指定当告警条件被满足时会发生的动作。 |

**表格9.1 结构说明**

#### Trigger

每个告警必须有一个触发器，用来定义开始执行的时间点。每当创建一个触发器时，这个触发器被注册到一个合适的触发器引擎中。所有的触发器都是基于时间的类型。

时间触发器用系统时钟来确定当前的时间。所以要确保正确的触发预期时，需要同步所有节点上的时钟。系统提供以下几种类型的触发器类型：

hourly、daily、weekly、monthly、yearly、cron、interval

l hourly小时时刻表：

小时时刻表可以指定每小时执行一个计划，可以设置每小时的几分钟开始执行，如果不这是分钟，默认是每小时的0分开始执行。

例如，以下每小时时间表触发器在每小时的第30分钟执行，12:30，13:30，14:30等等。

{

  ...

  "trigger" : {

    "schedule" : {

      "hourly" : { "minute" : 30 }

    }

  }

  ...

}

在小时时刻表中可以设置多个分钟执行，例如：

{

  ...

  "trigger" : {

    "schedule" : {

      "hourly" : { "minute" : [ 0, 15, 30, 45 ] }

    }

  }

  ...

}

l daily天时刻表

天时刻表指定每一天在某一特定时间触发的一个计划。要配置一个一天的时间表，指定一个单一的时间与属性。如果不指定，默认是0点执行。例如，下面的作息时间触发一次每天下午5:00执行。

{

  ...

  "trigger" : {

    "schedule" : {

      "daily" : { "at" : "17:00" }

    }

  }

  ...

}

在天时时刻表中可以设置多个时间点执行，例如下面的设置在每天的 00:00, 12:00和 17:00执行：

{

  ...

  "trigger" : {

    "schedule" : {

      "daily" : { "at" : [ "midnight", "noon", "17:00" ] }

    }

  }

  ...

}

midnight和noon是保留字，表示00:00和12:00。

l weekly周时刻表

周时刻表指定每一周在某一特定时间触发的一个计划。可以指定一周的名称、缩写或数字（星期日是一周的第一天）：

sunday, monday, tuesday, wednesday, thursday, friday and saturday

sun, mon, tue, wed, thu, fri and sat

1, 2, 3, 4, 5, 6 and 7

例如下面的设置表示每周五下午5点执行：

{

  ...

  "trigger" : {

    "schedule" : {

      "weekly" : { "on" : "friday", "at" : "17:00" }

    }

  }

  ...

}

每周的时刻表也可以设置多个时间点执行，例如：

{

  ...

  "trigger" : {

    "schedule" : {

      "weekly" : [

          { "on" : "tuesday", "at" : "noon" },

          { "on" : "friday", "at" : "17:00" }

      ]

    }

  }

  ...

}

或者：

{

  ...

  "trigger" : {

    "schedule" : {

      "weekly" : {

        "on" : [ "tuesday", "friday" ],

        "at" : [ "noon", "17:00" ]

      }

    }

  }

  ...

}

l monthly月时刻表

月时刻表指定每一月在某一特定时间触发的一个计划。

例如下面的设置表示每月10日中午12点执行：

{

  ...

  "trigger" : {

    "schedule" : {

      "monthly" : { "on" : 10, "at" : "noon" }

    }

  }

  ...

}

每月的时刻表也可以设置多个时间点执行，例如：

{

  ...

  "trigger" : {

    "schedule" : {

      "monthly" : [

          { "on" : 10, "at" : "noon" },

          { "on" : 20, "at" : "17:00" }

      ]

    }

  }

  ...

}

或者：

{

  ...

  "trigger" : {

    "schedule" : {

      "monthly" : {

        "on" : [ 10, 20 ],

        "at" : [ "midnight", "noon" ]

      }

    }

  }

  ...

}

l yearly年时刻表

年时刻表指定每一年在某一特定时间触发的一个计划。可以指定月的名称、缩写或数字：

january, february, march, april, may, june, july, august, september, october, november ,december

jan, feb, mar, apr, may, jun, jul, aug, sep, oct, nov ,dec

1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12

例如下面的设置表示每年1月10日中午12点执行：

{

  ...

  "trigger" : {

    "schedule" : {

      "yearly" : { "in" : "january", "on" : 10, "at" : "noon" }

    }

  }

  ...

}

每年的时刻表也可以设置多个时间点执行，例如：

{

  ...

  "trigger" : {

    "schedule" : {

      "yearly" : [

          { "in" : "january", "on" : 10, "at" : "noon" },

          { "in" : "july", "on" : 20, "at" : "17:00" }

      ]

    }

  }

  ...

}

或者：

{

  ...

  "trigger" : {

    "schedule" : {

      "yearly" : {

        "in: : [ "jan", "dec" ],

        "on" : [ 10, 20 ],

        "at" : [ "midnight", "noon" ]

      }

    }

  }

  ...

}

l cron表达式时刻表

|  |  |  |  |
| --- | --- | --- | --- |
| 域名称 | 是否必须 | 字段返回 | 特殊字符 |
| seconds | yes | 0-59 | , - \* / |
| minutes | yes | 0-59 | , - \* / |
| hours | yes | 0-23 | , - \* / |
| day\_of\_month | yes | 1-31 | , - \* / ? L W |
| month | yes | 1-12 or JAN-DEC | , - \* / |
| day\_of\_week | yes | 1-7 or SUN-SAT | , - \* / ? L # |
| year | no | empty or 1970-2099 | , - \* / |

**表格9.2 cron表达式元素说明**

特殊字符，它们的含义是：

\*：表示匹配该域的任意值，假如在Minutes域使用\*, 即表示每分钟都会触发事件。

?:只能用在day\_of\_month和day\_of\_week两个域。它也匹配域的任意值，但实际不会。因为day\_of\_month和day\_of\_week会相互影响。例如想在每月的20日触发调度，不管20日到底是星期几，则只能使用如下写法： 13 13 15 20 \* ?, 其中最后一位只能用？，而不能使用\*，如果使用\*表示不管星期几都会触发，实际上并不是这样。

-:表示范围，例如在Minutes域使用5-20，表示从5分到20分钟每分钟触发一次

/：表示起始时间开始触发，然后每隔固定时间触发一次，例如在Minutes域使用5/20,则意味着5分钟触发一次，而25，45等分别触发一次.

,:表示列出枚举值值。例如：在Minutes域使用5,20，则意味着在5和20分每分钟触发一次。

L:表示最后，只能出现在day\_of\_month和day\_of\_week域，如果在day\_of\_week域使用5L,意味着在最后的一个星期四触发。

W:表示有效工作日(周一到周五),只能出现在day\_of\_month域，系统将在离指定日期的最近的有效工作日触发事件。例如：在 day\_of\_month使用5W，如果5日是星期六，则将在最近的工作日：星期五，即4日触发。如果5日是星期天，则在6日(周一)触发；如果5日在星期一到星期五中的一天，则就在5日触发。另外一点，W的最近寻找不会跨过月份

LW:这两个字符可以连用，表示在某个月最后一个工作日，即最后一个星期五。

#:用于确定每个月第几个星期几，只能出现在day\_of\_month域。例如在4#2，表示某月的第二个星期三。

例如下面的表达式表示每天中午12点执行：

{

  ...

  "trigger" : {

    "schedule" : {

      "cron" : "0 0 12 \* \* ?"

    }

  }

  ...

}

l interval系统内置表达式：

系统内置表达式可以设置 seconds, minutes, hours, days, weeks:

"Xs" 表示每多少秒执行一次。例如： "30s"表示每30秒执行一次。

"Xm"表示每多少分钟执行一次。例如： "5m"表示每5分钟执行一次。

"Xh"表示每多少小时执行一次。例如： "12h"表示每12小时执行一次。

"Xd"表示每多少天执行一次。例如： "3d"表示每3天执行一次。

"Xw"表示每多少周执行一次。例如： "2w"表示每2周执行一次。

例如下面的例子表示每5分钟执行一次：

{

  ...

  "trigger" : {

    "schedule" : {

      "interval" : "5m"

    }

  }

  ...

}

#### Inputs

输入将静态数据加载到告警执行上下文中作为判断的内容，系统支持四种类型输入源： simple, search, http , chain。

l Simple Input简单输入

简单输入是指将静态的内容加载到告警执行的上下文中，可以定义一个字符串（STR）、数值（NUM），或一个对象（目标）等静态数据作为输入源。例如，下面的告警使用了简单的输入来设置一个提醒电子邮件，在中午的每一天执行：

{

  "trigger" : {

    "schedule" : {

      "daily" : { "at" : "noon" }

    }

  },

  "input" : {

    "simple" : {

      "name" : "John"

    }

  },

  "actions" : {

    "reminder\_email" : {

      "email" : {

        "to" : "to@host.domain",

        "subject" : "Reminder",

        "body" : "Dear {{ctx.payload.name}}, by the time you read these lines, I'll be gone"

      }

    }

  }

}

l Search Input搜索输入

搜索输入只指通过Elasticsearch的搜索的结果作为输入加载到告警执行上下文中作为判断的内容。Conditions, transforms, actions都可以通过结果属性访问搜素中的内容。

例如：

ctx.payload.hits，可以表示搜索的所有内容；

ctx.payload.hits.total，可以表示搜索的结果条数；

ctx.payload.hits.hits.2，可以表示搜索结果的第三条数据。

下面表示搜索所有的日志作为输入的实例：

"input" : {

  "search" : {

    "request" : {

      "indices" : [ "logs" ],

      "types" : [ "event" ],

      "body" : {

        "query" : { "match\_all" : {}}

      }

    }

  }

}

下面表示通过模板搜索的日志作为输入的实例：

{

  "input" : {

    "search" : {

      "request" : {

        "indices" : [ "logs" ],

        "template" : {

          "id" : "my\_template",

          "params" : {

            "value" : 23

          }

        }

      }

    }

  }

  ...

}

l HTTP Input HTTP协议输入

HTTP协议输入指通过HTTP请求Elasticsearch得到的返回结果作为输入加载到告警执行上下文中作为判断的内容。例如：

"input" : {

  "http" : {

    "request" : {

      "host" : "example.com",

      "port" : 9200,

      "path" : "/idx/\_search"

    }

  }

}

或者用过DSL语言进行查询：

"input" : {

  "http" : {

    "request" : {

      "host" : "host.domain",

      "port" : 9200,

      "path" : "/idx/\_search",

      "body" :  "{\"query\" :  {  \"match\" : { \"category\" : \"event\"}}}"

    }

  }

}

或者通过模板来进行查询：

"input" : {

  "http" : {

    "request" : {

      "host" : "host.domain",

      "port" : 9200,

      "path" : "/{{ctx.watch\_id}}/\_search",

      "body" : "{\"query\" : {\"range\": {\"@timestamp\" : {\"from\": \"{{ctx.trigger.triggered\_time}}||-5m\",\"to\": \"{{ctx.trigger.triggered\_time}}\"}}}}"

      }

    }

  }

也可以通过调用Elasticsearch API来进行查询：

"input" : {

  "http" : {

    "request" : {

      "host" : "host.domain",

      "port" : "9200",

      "path" : "/\_cluster/stats",

      "params" : {

        "human" : "true"

      }

    }

  }

}

l Chain Input链式输入

链式输入是指可以一次把多个输入加载到告警执行上下文中作为判断的内容。当基于来自多个源的数据执行操作时，链输入是有用的。还可以使一个输入收集的数据是从另一个源加载数据。例如，以下链输入数据使用由一个简单的输入路径来自HTTP服务器的请求：

{

  "input" : {

    "chain" : {

      "inputs" : [ 

        {

          "first" : {

            "simple" : { "path" : "/\_search" }

          }

        },

        {

          "second" : {

            "http" : {

              "request" : {

                "host" : "localhost",

                "port" : 9200,

                "path" : "{{ctx.payload.first.path}}" ‚

              }

            }

          }

        }

      ]

    }

  }

  ...

}

在一个链中的输入被指定为一个数组，以保证输入的顺序处理。

‚加载由第一个输入设置的路径。本文由赛克 蓝德(secisland)原创，转载请标明作者和出处。

#### Conditions

当一个告警被触发时，Conditions配置的条件是否满足决定了是否执行下一步的动作。系统支持四中类型的条件类型always, never, script, compare。

l Always Condition

表示触发时间到后始终要执行。例如：

"condition" : {

  "always" : {}

}

l Never Condition

表示永远不会被执行，这种情况通常只用于测试。例如：

PUT \_watcher/watch/my-watch

{

  ...

  "condition" : {

    "never" : {}

  }

  ...

}

l Script Condition

表示用一个脚本来表示告警的条件。默认脚本语言Groovy。可以使用任何Elasticsearch支持的脚本语言。以下代码片段将使用内联脚本的情况总是返回true：

"condition" : {

  "script" : "return true"

}

可以使用文件作为脚本数据源，例如：

"condition" : {

  "script" : {

    "file" : "my\_script",

    "lang" : "javascript",

    "params" : {

      "result" : true

    }

  }

}

l Compare Condition

表示一种简单的将执行上下文模型中的值与给定值进行比较的告警条件。例如下面的例子表示查询的结果条数大于等于5的时候满足条件：

{

  ...

  "condition" : {

    "compare" : {

      "ctx.payload.hits.total" : {

        "gte" : 5

      }

  }

  ...

}

条件有以下几种：eq等于,not\_eq不等于,gt大于,gte大于等于,lt小于,lte小于等于。

例如：

{

  ...

  "condition" : {

    "compare" : {

      "ctx.payload.aggregations.status.buckets.error.doc\_count" : {

        "not\_eq" : "{{ctx.payload.aggregations.handled.buckets.true.doc\_count}}"

      }

  }

  ...

}

#### Transforms

转换这项不是必须的，表示在输入到执行动作的过程中进行的数据处理。可以定义在两个地方的变换：

1、在定义告警的顶级结构中，在这种情况下，在执行任何告警操作之前被转换。

2、作为一个特定行动定义的一部分。在这种情况下，执行该操作之前被转换。变换仅适用于特定的动作转换。

例如下面的示例定义了两个转换，一个是全局转换和一个在my\_webhook行为中进行转换。

{

  "trigger" : { ...}

  "input" : { ... },

  "condition" : { ... },

  "transform" : {

    "search" : {

      "body" : { "query" : { "match\_all" : {} } }

    }

  }

  "actions" : {

    "my\_webhook": {

      "transform" : {

        "script" : "return ctx.payload.hits"

      }

      "webhook" : {

        "host" : "host.domain",

        "port" : 8089,

        "path" : "/notify/{{ctx.watch\_id}}"

      }

    }

  ]

  ...

}

转换支持搜索转换，脚本转换和链转换

l Search Transform

在群集上执行搜索的转换，并用返回的搜索结果替换告警执行上下文中的内容，例如：

{

  "transform" : {

    "search" : {

      "request" : {

        "indices" : [ "logstash-\*" ],

        "body" : {

          "size" : 0,

          "query" : {

            "match" : { "priority" : "error"}

          }

        }

      }

    }

  }

}

l Script Transform

在群集上执行脚本的转换，并用返回的脚本执行结果替换告警执行上下文中的内容，例如：

{

  ...

  "transform" : {

    "script" : "return [ time : ctx.trigger.scheduled\_time ]"

  }

  ...

}

l Chain Transform

链式转换是指可以一次把多个转换加载到转换中，利用它可以建立更复杂的转换。例如，可以将搜索转换和脚本转换同时满足，如下面的示例：

"transform" : {

  "chain" : [

    {

      "search" : {

        "indices" : [ "logstash-\*" ],

        "body" : {

          "size" : 0,

          "query" : {

            "match" : { "priority" : "error" }

          }

        }

      }

    },

    {

      "script" : "return [ error\_count : ctx.payload.hits.total ]"

    }

  ]

}

#### Actions

Actions表示当告警条件被满足时会发生的动作。告警的动作可以是邮件，HipChat，Slack等，详见9.1.3章节。例如：

PUT \_watcher/watch/log\_event\_watch

{

  "metadata" : {

    "color" : "red"

  },

  "trigger" : {

    "schedule" : {

      "interval" : "5m"

    }

  },

  "input" : {

    "search" : {

      "request" : {

        "indices" : "log-events",

        "body" : {

          "size" : 0,

          "query" : { "match" : { "status" : "error" } }

        }

      }

    }

  },

  "condition" : {

    "script" : "return ctx.payload.hits.total > 5"

  },

  "actions" : {

    "email\_administrator" : {

      "throttle\_period": "15m", 

      "email" : {

        "to" : "sys.admino@host.domain",

        "subject" : "Encountered {{ctx.payload.hits.total}} errors",

        "body" : "Too many error in the system, see attached data",

        "attachments" : {

          "attached\_data" : {

            "data" : {

              "format" : "json"

            }

          }

        },

        "priority" : "high"

      }

    }

  }

}

表示触发动作的周期在15分钟，以减少告警的数量。

## Marvel插件

*摘要: Marvel是Elastic公司推出的商业监控方案，是用来监控Elasticsearch集群，历史状态的有力工具，便于性能优化以及故障诊断。监控主要分为六个层面，分别是集群层、节点层、索引层、分片层、事件层、Sense。*

Marvel是Elastic公司推出的商业监控方案，是用来监控Elasticsearch集群，历史状态的有力工具，便于性能优化以及故障诊断。监控主要分为六个层面，分别是集群层、节点层、索引层、分片层、事件层、Sense。

### 1.1.1 ****安装****

Marvel是以Elasticsearch插件的形式存在的。在最新版本中Marvel有两个组件，一个是代理，需要安装在Elasticsearch集群中，一个是Kibana的插件，需要安装在Kibana中。

安装步骤：

1) 在Elasticsearch中安装Marvel插件。

bin/plugin install license

bin/plugin install marvel-agent

2) 在Kibana中安装Marvel插件

bin/kibana plugin --install elasticsearch/marvel/latest

3) 启动Elasticsearch和Kibana

bin/elasticsearch

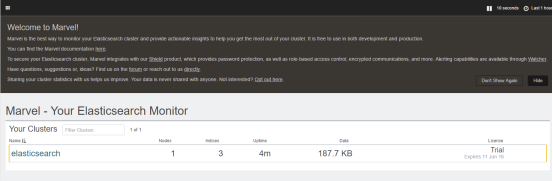
bin/kibana

安装好后在浏览器中输入：http://127.0.0.1:5601/即可看到界面如图9.1所示。



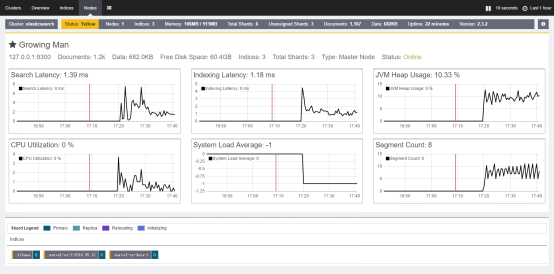
**图9.1 Kibana首页**

然后在界面的菜单中选择Marvel，会看到如图9.2所示的界面：



**图9.2 Marvel首页**

然后点击elasticsearch，看到如图9.3所示的监控界面



**图9-3 Marvel 监控页**

本文由赛克 蓝德(secisland)原创，转载请标明作者和出处。

### 1.1.2 ****配置****

l 监控参数配置

可以通过在每个节点的elasticsearch.yml中配置Marvel参数来控制从Elasticsearch集群中采集数据，可以添加一个自定义索引模板来更改索引监控的设置，从一个集群收集的数据中创建存储。

监控设置的参数在elasticsearch.yml文件中，从marvel.agent开始。

marvel.agent.cluster.state.timeout

设置用于收集群集状态的超时时间。默认为10秒。

marvel.agent.cluster.stats.timeout

设置用于收集群集统计的超时时间。默认为10秒。

marvel.agent.indices

控制哪些索引数据被收集，默认所有索引。指定索引的名称可以以逗号分隔的分开，例如test1，test2，test3。名称中可以包含通配符，例如test\*。你可以明确地包括或排除相关索引。例如，包括所有以test开头，但要排除test3的索引，写法如下：+test\*或者-test3。

marvel.agent.index.stats.timeout

设置用于收集索引统计的超时。默认为10秒。

marvel.agent.indices.stats.timeout

设置用于收集总索引统计的超时时间。默认为10秒。

marvel.agent.exporters

配置代理监测数据。默认情况下，代理安装在本地的群集上，它监视数据，使用HTTP协议把数据发送到一个单独的监控集群上，例如：

marvel.agent.exporters:

  id1:                              # default local exporter

    type: local

  id2:                                     # example of an http exporter

    type: http                             # exporter type, local or http

    host: [ "http://domain:port",... ]     # host(s) to send data to over http or https

    auth:

      username: <string>            # basic auth username

      password: <string>            # basic auth password

    connection:

      timeout: <time\_value>         # http connection timeout (default: 6s)

      read\_timeout: <time\_value>    # http response timeout (default: connection.timeout \* 10)

      keep\_alive: true | false      # use persistent connections (default: true)

    ssl:

      hostname\_verification: true | false  # check host certificate (default: true)

      protocol: <string>                   # security protocol (default: TLSv1.2)

      truststore.path: /path/to/file       # absolute path to the truststore

      truststore.password: <string>        # password for the truststore

      truststore.algorithm: <string>       # format for the truststore (default: SunX509)

    index:

      name:

        time\_format: <string>              # time format suffix for marvel indices (default: "YYYY.MM.dd")

marvel.agent.index.recovery.active\_only

控制是否所有的恢复数据被收集。设置为true收集有效的恢复数据。默认为false。

marvel.agent.index.recovery.timeout

设置用于收集恢复数据的超时时间。默认为10秒。

marvel.agent.interval

控制收集数据样本的频率。默认设置为10秒，设置为-1表示禁用数据收集。

marvel.history.duration

设置监控创建的索引将被自动删除的保留时间。默认为7天。设置为-1来禁用自动删除监控索引。

l 监控索引配置

监控使用一个索引模板来配置用于存储从集群中收集数据的索引。

可以通过如下命令检索默认的模板：

GET /\_template/.marvel-es

默认情况下，模板配置一个分片和一个复制的监控索引。要覆盖默认设置，可以添加自己的模板，例如：

PUT /\_template/custom\_marvel

{

    "template": ".marvel\*",

    "order": 1,

    "settings": {

        "number\_of\_shards": 5,

        "number\_of\_replicas": 2

    }

}

l Kibana配置相关

在Kibana 的配置文件(kibana.yml)中可以设置Marvel监控参数，在大多数情况下，默认值已经可以很好工作。

marvel.max\_bucket\_size

从检索索引节点指标聚合中返回执行聚合检索索引和节点的桶的数量。默认为10000。

marvel.min\_interval\_seconds

时间桶所能表示的最小秒数。默认为10秒。

marvel.node\_resolver

被认为是唯一节点的标志。

marvel.report\_stats

是否将集群统计数据发送到Elastic公司。默认为true。

l Tribe部落节点监控配置

如果你通过一个部落节点连接到一个集群，需要在部落节点以及集群中的节点上安装监控代理来监控集群。如果群集上设置了权限，则还需要在部落节点上安装和配置权限。

排除部落节时需要在elasticsearch.yml配置文件中设置marvel.enabled为false，例如：

node.name: tribe

marvel.enabled: false

tribe:

  t1:

    cluster.name: cluster1

    discovery.zen.ping.unicast.hosts: ["cluster1-node1:9300", "cluster1-node2:9300"]

有了这个配置后，部落节点被包含在监控用户界面中显示的节点数中，但不包含在节点列表中，因为它不将任何数据导出到监视群集中。

下面的示例是在监控所有部落级的数据：

node.name: tribe

marvel.enabled: false

tribe:

  t1:

    cluster.name: cluster1

    discovery.zen.ping.unicast.hosts: ["cluster1-node1:9300", "cluster1-node2:9300"]

    marvel:

      enabled: true

      agent.exporters:

        id1:

          type: http

          host: ["monitoringhost:9200"]

当启用部落节点的数据收集时，它包含在节点计数和节点列表中。值得注意的是，部落节点只支持HTTP的输出，部落节点的数据必须发送到外部监控群集上。

## Shield插件

*摘要: Shield是Elastic公司推出的商业权限管理插件，它可以保护Elasticsearch中的数据，采用加密的通讯密码，基于角色的访问控制，IP过滤和审计等。*

Shield是Elastic公司推出的商业权限管理插件，它可以保护Elasticsearch中的数据，采用加密的通讯密码，基于角色的访问控制，IP过滤和审计等。

#### 安装

Shield是以Elasticsearch插件的形式存在的。安装过程如下：

1) 在Elasticsearch中安装Shield插件。

bin/plugin install license

bin/plugin install shield

2) 启动Elasticsearch。

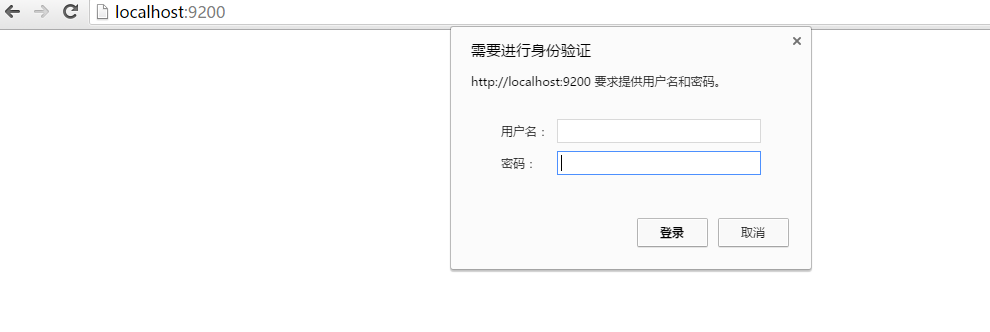
bin/elasticsearch

3) 添加管理员账号

bin/shield/esusers useradd es\_admin -r admin

4) 测试是否生效

在页面访问数据的时候，会显示如图9.4所示的界面：



**图9.4 Shield登录页面**

当输入正确的账户和密码的时候，才可以访问数据。

### 1.1.1 ****工作原理****

Shield是Elasticsearch的一个插件，一旦安装完成，插件将会拦截所有API请求，然后对请求进行认证和授权的校验。该插件同时提供SSL安全协议来传输网络数据，该插件提供了审计日志记录的能力，用来进行验证和审计。

#### 用户认证

Shield定义了一组已知的用户，以便对请求的用户进行身份验证。这些用户的集合被定义为一个抽象的领域。这个抽象的领域可以是数据库，本地文件，LDAP，活动目录或者KPI。

#### 授权

Shield的访问授权的数据模型由以下几个因素构成：

l 资源，包括集群，索引，别名等。

l 权限，对资源的一个或者多个操作。

l 许可，对应安全资源的一个或者多个权限。有2种类型的许可：集群和索引。

l 角色，权限的集合。

l 用户，准许访问资源的对象。

#### 节点认证和信道加密

Shield可以使用SSL / TLS包进行内部通信。当SSL / TLS启用，节点相互验证的证书，建立节点之间的信任，验证防止未经身份验证的节点加入集群，通过验证后，内部通信是经过加密的。

#### IP过滤

Shield提供基于IP的针对节点的访问控制。通过IP控制可以限制其他机器访问Elasticsearch服务器，可以设置白名单或者黑名单进行过滤，可以设置IP或者网段。

#### 审计

安全审计功能提供了访问Elasticsearch特定事件或者活动的记录，包括登录，授权，拒绝访问等。

### 1.1.2 ****用户认证****

#### 用户认证方式

系统支持以下几种方式的用户认证：

Native：一个内置的本地认证系统，默认可用。

File：一种内置的基于文件的认证系统，默认可用。

LDAP：通过外部轻量级目录协议进行身份验证。

AD：通过外部活动目录服务的身份验证。

PKI：通过使用可信的X.509证书的认证。

native, file, LDAP, AD这四种方式是用用户密码进行认证的。系统也支持自定义用户认证方式，可以多种认证方式同时存在，它本质上是一个配置的认证方式的优先列表。列表的顺序决定了将要认证的顺序。在认证过程中，系统将尝试一次验证一个认证方式的请求。一旦一种认证方式认证成功，通过身份认证的用户将与请求相关联的（下一步继续授权阶段）。如果一个认证方式无法对请求进行身份验证，则将认证链中的下一个认证方式。如果链中所有服务器无法验证的要求，认证就失败了，将返回错误（HTTP状态代码401）。

例如下面的代码片段配置认证链包含文件认证和本地服务器认证，以及两个LDAP和Active Directory域服务器认证。

shield.authc:

    realms:

        native:

            type: native

            order: 0

        file:

            type: file

            order: 1

        ldap1:

            type: ldap

            order: 2

            enabled: false

            url: 'url\_to\_ldap1'

            ...

        ldap2:

            type: ldap

            order: 3

            url: 'url\_to\_ldap2'

            ...

        ad1:

            type: active\_directory

            order: 4

            url: 'url\_to\_ad'

#### 匿名用户访问

认证过程可以分为两个阶段：令牌提取和用户认证。在第一阶段（令牌提取阶段），配置的服务器将请求尝试从传入的请求中提取/解析身份验证令牌。找到身份验证的令牌，然后将令牌用于认证。在没有任何身份验证令牌的情况下，传入的请求被认为是匿名的。

默认情况下，匿名请求被拒绝，并返回一个身份验证错误码（状态代码401）。可以通过配置改变这种行为，可以通过在elasticsearch.yml文件中配置下面的内容：

shield.authc:

  anonymous:

    username: anonymous\_user

    roles: role1, role2

    authz\_exception: true

#### 本地认证

本地认证配置如下：

shield:

  authc:

    realms:

      native1:

        type: native

        order: 0

可以通过 Users API来完成本地认证用户的管理。

l 增加本地用户

POST /\_shield/user/ironman

{

  "password" : "j@rV1s",

  "roles" : [ "admin", "other\_role1" ],

  "full\_name" : "Tony Stark",

  "email" : "tony@starkcorp.co",

  "metadata" : {

    "intelligence" : 7

  }

}

l 查询本地用户列表

GET /\_shield/user

返回的值：

{

  "found" : true,

  "users" : [

    {

      "username" : "ironman",

      "roles" : [ "admin", "other\_role1" ],

      "full\_name" : "Tony Stark",

      "email" : "tony@starkcorp.co",

        "metadata" : {

          "intelligence" : 7

        }

      }

    }

  ]

}

l 删除本地用户

DELETE /\_shield/user/ironman

本文由赛克 蓝德(secisland)原创，转载请标明作者和出处。

#### LDAP认证

可以通过配置使用目录访问协议（LDAP）进行用户认证。需要配置LDAP域和指定LDAP的角色在角色映射文件中。为了保护密码，和LDAP服务器之间的通信必须使用SSL / TLS加密。客户节点通过SSL / TLS连接到LDAP服务器需要LDAP服务器的证书或服务器的根CA证书安装在自己的密钥库和信任存储区中。在elasticsearch.yml中添加shield.authc.realms命名空间来进行配置，例如，下面的代码片段显示了在LDAP中配置用户认证：

shield:

  authc:

    realms:

      ldap1:

        type: ldap

        order: 0

        url: "ldaps://ldap.example.com:636"

        bind\_dn: "cn=ldapuser, ou=users, o=services, dc=example, dc=com"

        bind\_password: changeme

        user\_search:

          base\_dn: "dc=example,dc=com"

          attribute: cn

        group\_search:

          base\_dn: "dc=example,dc=com"

        files:

          role\_mapping: "CONFIG\_DIR/shield/role\_mapping.yml"

        unmapped\_groups\_as\_roles: false

配置用户DN模板来进行LDAP认证

如果你的LDAP环境使用一些特定的标准命名条件的用户，可以使用用户DN模板来配置。这种方法的优点是搜索不一定是要找出用户的DN。但多个绑定操作可能需要找到正确的用户DN。例如：

shield:

  authc:

    realms:

      ldap1:

        type: ldap

        order: 0

        url: "ldaps://ldap.example.com:636"

        user\_dn\_templates:

          - "cn={0}, ou=users, o=marketing, dc=example, dc=com"

          - "cn={0}, ou=users, o=engineering, dc=example, dc=com"

        group\_search:

          base\_dn: "dc=example,dc=com"

        files:

          role\_mapping: "/mnt/elasticsearch/group\_to\_role\_mapping.yml"

        unmapped\_groups\_as\_roles: false

配置LDAP组

存储在每个节点的角色映射文件指定了LDAP组。当一个用户使用LDAP认证时，用户所属的组的角色定义了用户的权限。例如，下面的映射配置指定了LDAP管理员组和普通用户组：

monitoring:

  - "cn=admins,dc=example,dc=com"

user:

  - "cn=users,dc=example,dc=com"

  - "cn=admins,dc=example,dc=com"

SSL / TLS加密配置

1、首先要生成证书：

cd CONFIG\_DIR/shield

keytool -importcert -keystore node01.jks -file cacert.pem -alias ldap\_ca

2、在elasticsearch.yml配置文件中配置SSL / TLS加密：

shield.ssl.keystore.path:          /home/es/config/shield/node01.jks

shield.ssl.keystore.password:      myPass

shield.ssl.keystore.key\_password:  myKeyPass

3、配置LDAPS协议指定的URL属性和端口号。例如，url: ldaps://ldap.example.com:636。

4、重启Elasticsearch使elasticsearch.yml生效。

#### AD认证

可以通过配置使用活动目录（LDAP）进行用户认证。需要配置LDAP域和指定LDAP的角色在角色映射文件中。为了保护密码，和LDAP服务器之间的通信必须使用SSL / TLS加密。客户节点通过SSL / TLS连接到LDAP服务器需要LDAP服务器的证书或服务器的根CA证书安装在自己的密钥库和信任存储区中。在elasticsearch.yml中添加shield.authc.realms命名空间来进行配置，例如，下面的代码片段显示了在AD中配置用户认证

shield:

  authc:

    realms:

      active\_directory:

        type: active\_directory

        order: 0

        domain\_name: ad.example.com

        url: ldaps://ad.example.com:636

        unmapped\_groups\_as\_roles: true

AD认证的很多内容和上一节的LDAP基本一样，其他内容请参考上一节。

#### PKI身份认证

您以配置使用公共密钥基础设施（PKI）证书对用户进行身份验证。这要求客户提供X.509证书。可以结合使用PKI认证和用户名/密码认证。例如，可以启用SSL / TLS在传输层上使用一个PKI验证的X.509证书，同时还验证HTTP协议使用的用户名和密码。还可以设置shield.transport.ssl.client.auth允许客户无证书认证等。在elasticsearch.yml中添加shield.authc.realms命名空间来进行配置，例如：

shield:

  authc:

    realms:

      pki1:

        type: pki

为PKI用户分配角色

分配给PKI用户的角色映射文件存储在每个节点。在证书中可以区分不同的用户。例如，下面的映射配置指定John Doe的用户角色：

user:

  - "cn=John Doe,ou=example,o=com"

#### 基于文件的授权

可以管理和认证基于文件的认证，它是系统内容的认证方式，可以通过esusers控制台命令来增加和删除用户，分配用户角色，管理用户密码。在elasticsearch.yml中添加shield.authc.realms命名空间来进行配置，例如：

shield:

  authc:

    realms:

      file1:

        type: file

        order: 0

用户管理命令在ES\_HOME/bin/shield目录下：

l 增加用户

增加用户：esusers useradd <username>

增加用户同时设置密码：esusers useradd <username> -p <secret>

增加用户同时设置角色：esusers useradd <username> -r <comma-separated list of role names>

l 查询用户

esusers list

l 管理密码

esusers passwd <username> -p <password>

l 用户分配角色

用-a增加角色，-r删除角色。

esusers roles <username> -a <commma-separate list of roles> -r <commma-separate list of roles>

l 删除用户

userdel <username>

## 5.0的新特性

*摘要: elasticsearch5.0.0版本变化比较大，再次记录一下主要的变化内容，后续会逐步把重要的变化记录下来。*

10月27日ElasticStack发布了全新的5.0.0版本，ElasticStack是一个解决方案统称包括，Elasticsearch、Kibana、Beats、Logstash几个主要的程序。为了和其他产品版本号保持一致，elasticsearch的版本也从2.4版本直接跳到5.0版本。

升级指南：

5.X版本的ES可以读取2.X版本生成的索引，所以2.X版本要迁移到5.X， 需要重新启动完整的集群。系统不支持1.X版本到5.X版本的升级。对1.X版本的升级需要先对1.X版本的索引进行重建。

系统提供了一个迁移助手插件，可以帮助系统从2.3以上的版本迁移到5.X。  
elasticsearch5.0.0主要新特性

**索引性能**

由于采用了很多新的技术，在5.0.0中的索引性能有了明显的提高。这些新技术包括采用了新的数字类型结构，在并发更新到同一文档中采用了新的技术减少了锁的冲突，在同步事务日志时减少锁定的条件。采用了异步同步日志机制，增加减少了磁盘寻址的时间。当依靠Elasticsearch自动生成文档ID时采用了采用了基于时间的用户实例增加了比较大的效率。支持实时文档内部更改的获取，可以增加更多的可用内存用于索引缓冲区，同时在垃圾收集中花费更少的时间。通常情况，可以提高25%-80%的效率。

**数据转换**(Ingest Node)

在5.0.0中，新增了一个特性数据转换(Ingest Nodes)。他可以不依赖于Logstash实现常用的过滤能力，比如grok, split, convert, date等。它可以用来执行常见的数据转换和处理。可以使用转换节点在实际索引之前对文档进行预处理。在任何转换节点中，处理索引或者块处理之前进行预处理转换。可以在任何节点开启转换功能，或者建立单独的转换节点。在默认情况下，在任何节点都有开启了转换能力，如果要关闭转换能力，需要在配置文件中添加：

node.ingest: false

在索引文档之前进行预处理，它定义了一个指定一系列处理器的管道。每个处理器以某种方式转换文档。例如，你可能有一个管道，包括一个处理器，从文档中删除字段，然后进入另一个处理器，对文档中的字段进行重命名。  
使用一个管道，你只需在一个索引或批量请求后加入管道参数。例如:

PUT my-index/my-type/my-id?pipeline=my\_pipeline\_id

{

"foo": "bar"

}

在使用前，需要先定义管道，例如定义上面的管道：

PUT \_ingest/pipeline/my-pipeline-id

{

"description" : "describe pipeline",

"processors" : [

{

"set" : {

"field": "foo",

"value": "bar"

}

}

]

}

本文由赛克 蓝德(secisland)原创，转载请标明作者和出处。

**新增脚本Painless scripting**

由于外部的脚本引擎太过于强大，什么都能做，用不好或者设置不当就会引起安全风险，基于安全和性能方面的考虑，在5.0中引入了一个新的脚本引擎Painless，顾名思义，简单安全，无痛使用,这个脚本引擎默认是开启的。和Groove的沙盒机制不一样，Painless使用白名单来限制函数与字段的访问，针对es的场景来进行优化，只做es数据的操作，更加轻量级，速度要快好几倍，并且支持Java静态类型，语法保持Groove类似，还支持Java的lambda表达式。

**新的数据结构**

在5.0版本中集成了Lucene 6，Lucene 6对数字类型和地理类型增加了一个新的点数据结构，叫做K-Ds树，它改变了数字类型的索引和搜索。它增加了36%的查询速度，增加了71%索引速度，减少了66%的硬盘空间占用，减少了85%的内存使用。新增加的IP字段支持IP4和IP6。根据Lucene 6新的LatLonPoint结构修改了地理点类型，使地理点查询的性能增加了一倍。同时还增加了一个半精度(half\_float)浮点类型（16位）和大浮点类型(scaled\_float)，他们使用了数据压缩技术来减少磁盘占用。这些新类型可以意味着在许多情况下，特别是在数字型数据中将显着减少的磁盘空间。

引入新的字段类型Text/Keyword 来替换 String

以前的string类型被分成Text和Keyword两种类型，keyword类型的数据只能完全匹配，适合那些不需要分词的数据，对过滤、聚合非常友好，text当然就是全文检索需要分词的字段类型了。将类型分开的好处就是使用起来更加简单清晰，以前需要设置analyzer和index，并且有很多都是自定义的分词器，从名称根本看不出来到底分词没有，用起来很麻烦。另外string类型暂时还在的，6.0会移除。

**搜索和聚合**

Elasticsearch提供了Aggregation缓存，如果你的数据没有变化，ES能够直接返回上次的缓存结果，

但是有一个场景比较特殊，就是 date分组，如：from:now-30d to:now，now是一个变量，每时每刻都在变，所以query条件一直在变，这样缓存也就是没有利用起来。在5.0中做了改进，

**首先，**`now`关键字最终会被重写成具体的值；

**其次，**每个分片会根据自己的数据的范围来重写查询为 `match\_all`或者是`match\_none`查询，所以现在的查询能够被有效的缓存，并且只有个别数据有变化的分片才需要重新计算，大大提升查询速度。

搜索结果分页增加了search\_after特性，它有效地跳过以前返回的结果只返回下一页的数据。移动搜索现在可以被并行执行。默认使用BM25评分算法，替换之前的TF/IDF评分算法。

增加了Percolator type映射类型。完全重写了completion suggester

**用户友好性的改进**

**索引的改进**

Elasticsearch的配置实在太多，在新的5.0版本中，对配置验证更加严格，保证原子性，如果其中一项失败，那个整个更新请求都会失败，不会一半成功一半失败。

**新增了一个Shrink接口**

在5.0之前的版本中索引的分片数是固定的，设置好了之后不能修改，如果发现分片太多或者太少，如果要修改，只能重建索引。在5.0中增加了Shrink接口，它可将分片数进行收缩成它的因数，如之前你是15个分片，你可以收缩成5个或者3个又或者1个，那么我们就可以想象成这样一种场景，在写入压力非常大的收集阶段，设置足够多的索引，充分利用shard的并行写能力，索引写完之后收缩成更少的shard，提高查询性能。

**新增了一个Rollover接口**

对于日志类的数据一般我们按天来对索引进行分割（数据量更大还能进一步拆分），我们以前是在程序里设置一个自动生成索引的模板，例如logstash中的logstash-[YYYY-MM-DD]模板，现在5.0中提供了一个更加简单的方式：Rollover API，例如：

PUT /logs-000001 https://www.elastic.co/guide/en/elasticsearch/reference/5.0/images/icons/callouts/1.png

{

"aliases": {

"logs\_write": {}

}

}

# Add > 1000 documents to logs-000001

POST /logs\_write/\_rollover https://www.elastic.co/guide/en/elasticsearch/reference/5.0/images/icons/callouts/2.png

{

"conditions": {

"max\_age": "7d",

"max\_docs": 1000

}

}

从上面可以看到，首先创建一个logs-0001的索引，它有一个别名是logs\_write,然后我们给这个logs\_write创建了一个rollover规则，即这个索引文档不超过1000个或者最多保存7天的数据，超过会自动切换别名到logs-0002,你也可以设置索引的setting、mapping等参数,剩下的es会自动帮你处理。

**新增：Depreated logging**

大家在用ES的时候，其实有些接口可能以及打上了Depreated标签，即废弃了，在将来的某个版本中就会移除，你当前能用是因为一般废弃的接口都不会立即移除，给足够的时间迁移，但是也是需要知道哪些不能用了，要改应用代码了，所以现在有了Depreated日志，当打开这个日志之后，你调用的接口如果已经是废弃的接口，就会记录下日志，那么接下来的事情你就知道你应该怎么做了。

**新增: Cluster allocation explain接口**

『谁能给我一个shard不能分配的理由』，现在有了，大家如果之前遇到过分片不能正常分配的问题，但是不知道是什么原因，只能尝试手动路由或者重启节点，但是不一定能解决，其实里面有很多原因，现在提供的这个explain接口就是告诉你目前为什么不能正常分配的原因，方便你去解决。

**弹性**

5.0版本增加了很多变化，它使ES的更加的安全。分布式模型的每一部分都已分离，重构，简化，并使之更可靠。群集状态更新要等待集群中的所有节点的确认。当主分片复制失败的时候，主要现在等待master节点的响应。索引的据路径现在使用UUID，而不是索引名称，可以避免命名冲突。

在5.0中增加了启动检查来来确保配置的正确性。同时引入了开发模式和生产模式两种方式，对生产模式的检查会更加的严格。系统增加了新的内存控制器，在聚合请求等过程监控内存的使用，当聚合内存有可能过大的时候则中断查询请求，正常情况下，内存不足的情况基本上不会发生。

对多用户的情况下，系统增加了更多的权限控制来减少普通用户的误操作，例如：请求超时后禁用的文本字段字段的数据加载，限制分片请求的数量，限制映射的字段数量等等。

**java RestClient客户端**

5.0里面提供了第一个Java原生的REST客户端SDK，相比之前的TransportClient，版本依赖简单，集群升级不影响，支持跨Java版本的调用等，新的基于HTTP协议的客户端对Elasticsearch的依赖解耦，没有jar包冲突，提供了集群节点自动发现、日志处理、节点请求失败自动进行请求轮询，充分发挥Elasticsearch的高可用能力，并且性能不相上下。

**迁移助手**

Elasticsearch迁移助手是一个网站的插件，它可以帮助你从Elasticsearch 2.3.x/2.4.x版本到5.x版本的迁移，它配备了三个工具：  
集群的检查：在您的群集、节点和索引上运行一系列检查，并提醒您在升级之前需要纠正的任何已知问题。  
重建索引助手：在V2.0.0版本之前的索引需要重新才可以在5.0中识别。重建索引助手通过一个按钮升级索引。  
过期接口日志：当使用过期接口时，系统会把过期的日志打出来，助手可以在集群中打开或者关闭过期日志打印功能

## 5.0版本搜索和DSL变化

搜索和DSL查询的变化

1、search\_type=count参数移除，这个是在2.0版本中已经不推荐使用的了，取而代之的是用查询，size设为0.

POST /my\_index/\_search

{

"size": 0,

"aggs": {

"my\_terms": {

"terms": {

"field": "foo"

}

}

}

}

2、search\_type=scan参数移除，这个是在2.1版本中已经不推荐使用的了。现在滚动请求用\_doc排序达到同样的效果。

POST /my\_index/\_search?scroll=2m

{

"sort": [

"\_doc"

]

}

3、分片查询限制，在1000个以内，因为分片查询会消耗大量的内存和cpu，如果想要绕开这个限制，需要在配置文件中修改参数：action.search.shard\_count.limit。

3.1、fields参数将被stored\_fields参数替换，它只返回存储的字段，将不在从 \_source字段中获取。

3.2、fielddata\_fields参数将被docvalue\_fields参数替换。

3.3、是否存在接口取消，取而代之的是用查询获得，参数size设为0，terminate\_after设置为1。

4、查询变化

4.1、不支持geo\_point的term查询。

4.2、fuzzy查询在numeric, date，ip字段中失效，取而代之的是范围查询。

4.3、在\_uid和\_id字段中查询range和prefix将不被支持。

4.4、查询不存在的索引将返回失败，而不是返回no hits。

4.5、在fuzzy查询中将不支持min\_similarity参数。

4.6、在query\_string查询中将不支持fuzzy\_min\_sim参数。

4.7、在completion suggester中取消edit\_distance参数。

4.8、在索引查询中将不在支持no\_match\_filter参数。

4.9、在嵌套查询中将不在支持filter字段查询。

4.10、在term查询中将不在支持minimum\_should\_match和disable\_coord，取消了execution参数。

4.11、在\_score查询中取消了top level过滤。

4.12、在span\_near查询中取消了collect\_payloads参数。

4.13、在 nested和has\_child查询中取消了score\_type参数， has\_parent查询中取消了score\_mode，同时取消了 total score mode。

4.14、在has\_child查询中max\_children参数在以前的版本中设置为0表示没有限制，现在表示0.

4.15、如果\_field\_names字段被禁用时， exists查询将返回失败。

4.16、如果cross\_fields，phrase，phrase\_prefix类型中有模糊(fuzziness),则multi\_match查询将失败。

4.17、GeoPolygonQuery,GeoDistanceQuery,GeoBoundingBoxQuery查询中coerce, normalize, ignore\_malformed参数将失效，用validation\_method 参数代替。

4.18、geo\_distance\_range 查询将取消，将用geo\_distance聚合来代替。

4.19、在search接口中top level过滤将取消，用post\_filter来代替。

4.20、多个高亮显示名称将不被支持，唯一支持的是plain, fvh and postings。

4.21、term vectors API不再坚持未映射的字段的映射。DFS参数在term vectors API中已经被完全删除。

4.22、reverse参数在排序中将被取消，在validation\_method方法中取消coerce和ignore\_malformed参数。

4.23、Top level inner hits语法被取消，inner hits只能在存在nested, has\_child，has\_parent查询中一部分。

查询分析器：

在profiling查询的返回值中，query\_type改名为type，lucene改名为description

搜索偏好：

在搜索偏好\_only\_node被删除，可以用\_only\_nodes和指定的node ID来实现。\_only\_nodes替换了\_only\_node的功能，\_only\_node不支持多个节点。preference \_shards参数接收用|分来的第二个参数，例如\_shards:2,3|\_primary。

默认使用BM25评分算法，替换之前的TF/IDF。

fields字段被修改为stored\_fields。

## 5.0版本映射的变化

在elasticsearch5.0.0中映射也有不少的变化。

1、string类型被替换成了text/keyword两种类型。

对于分词字段的字符串字段用text来替换，对于不分词的字符串字段用keyword来替换，为了兼容5.0中还保留了string类型，将在6.0版本中去掉。

字符串映射现在有如下的默认映射：

{

"type": "text",

"fields": {

"keyword": {

"type": "keyword",

"ignore\_above": 256

}

}

}

字符串被映射成了两个字段，一个字段进行分词，另一个子字段不进行分析，可以用于聚合或者排序。

2、数字类型

数字类型现在是一种全新的数据结构，被成为BKD tree。相比以前的方式有更快的对比查询效率和占用更少的磁盘空间。注意，数字类型将不参与索引的评分，如果需要对数字字段进行参与评分，可以同时映射为数字类型和keyword类型。例如：

PUT my\_index

{

"mappings": {

"my\_type": {

"properties": {

"my\_number": {

"type": "long",

"fields": {

"keyword": {

"type": "keyword"

}

}

}

}

}

}

}

3、geo\_point字段

和数字类型类型，Geo point字段类型也用了BKD tree结构。.由于这种结构从根本上进行多维空间数据的支持，下列字段参数将不在支持：Geohash，geohash\_prefix，geohash\_precision，lat\_lon。Geohashes在接口中仍然可以使用，但他们不再是用来索引地理数据点。

4、\_timestamp 和 \_ttl 字段

元字段\_timestamp 和 \_ttl字段将不再被支持，对于\_timestamp可以在文档中添加日期字段来代替或者用ingest pipline，例如：

PUT \_ingest/pipeline/timestamp

{

"description" : "Adds a timestamp field at the current time",

"processors" : [ {

"set" : {

"field": "timestamp",

"value": "{{\_ingest.timestamp}}"

}

} ]

}

PUT newindex/type/1?pipeline=timestamp

{

"example": "data"

}

GET newindex/type/1

对于\_ttl可以用time-based索引或者在一个时间戳字段范围查询(\_delete-by-query)的任务来替换。

POST index/type/\_delete\_by\_query

{

"query": {

"range" : {

"timestamp" : {

"lt" : "2016-05-01"

}

}

}

}

5、索引属性

所有再用的字段类型，除了将要废弃的string，索引属性只有 true/false用来代替not\_analyzed/no，string字段类型还是analyzed/not\_analyzed/no.

6、非索引字段的文档值

在此之前，设置一个字段的属性为index:no将禁用文档的值，现在，doc-values的值对数字和boolean类型的值总是有效，除非doc\_values的值设置为false。

7、默认浮点类型的之用float来代替double

在动态映射浮点类型的字段时，默认映射成float类型，而不是double类型，因为大多数情况下float类型已经够用了，而float类型将显著的减少磁盘空间的占用。

norms现在用boolean来代替对象，norms.enabled被替换成了boolean，norms.loading 参数eager 将不在起作用，现在norms是基于磁盘的.

设置fielddata.format: doc\_values在用于隐式启用字段的文档映射中。现在隐式方式将不在起作用，需要明确的设置doc\_values的值为有效或者无效。

fielddata.filter.regex参数将不在支持，未来版本会取消。

8、字段映射限制

在5.0中对索引中的字段进行了限制，最大1000个字段。

字段的最大深度（嵌套字段）是20。

索引中嵌套字段的最大数量是有限的50。

\_parent字段将不再索引，连接父母与孩子之间的文件不再依赖索引字段，因此从5.0.0起\_parent字段不再索引。为了找到文档引用一个特定的父id，可以使用新的parent\_id来进行查询。搜索返回中的得到响应和hits仍然包括父键下的父标识。

\_source映射不再支持格式选项，现在只是为了兼容性保留，为了会被去掉。

核心类型不再支持对象符号(bject notation)，它被用来提供每个文档的boosts ,例如：

{

"value": "field\_value",

"boost": 42

}

\_all查询的精度

在\_all上的每个字段的长度由以前的4个字节压缩到了一个字节，虽然这将使索引的空间效率更高，这也意味着索引时间的计算将不太准确。

## 5.0中的percolator类型和percolate查询

**percolator类型**

percolator字段类型解析json结构到本地查询并存储到索引中。因此可以用percolate查询来匹配提供的文档。这种情况可以理解正常搜索的反方向，一般情况下我们索引一个文档，然后通过搜索进行查询。percolator是先存储搜索，然后用文档来进行查询是否匹配搜索。

任何含有json对象的列可以被配置成percolator字段，例如下面的配置是映射percolator字段类型，这种类型适用于percolate查询：

{

"properties": {

"query": {

"type": "percolator"

}

}

}

那么下面的JSON代码段可以被索引作为一个本地查询：

{

"query" : {

"match" : {

"field" : "value"

}

}

}

重要：percolator查询必须已经存在于与所使用的percolation索引相关联的映射中，为了确保这些字段的存在，通过创建索引或者设置映射来增加或者更新。percolator类型可以存在于任何索引的任何类型中。一个索引中只能有一个percolator字段类型.

**percolate查询**

percolate查询可以用于将存储在索引中的查询进行匹配。例如创建两个映射的索引：

PUT /my-index

{

"mappings": {

"doctype": {

"properties": {

"message": {

"type": "text"

}

}

},

"queries": {

"properties": {

"query": {

"type": "percolator"

}

}

}

}

}

doctype映射是在percolator查询中被索引到一个临时索引之前用于预处理文件的映射。

queries映射用于索引查询文档。query字段将产生一个JSON对象代表一个实际的Elasticsearch查询。query字段配置为percolator字段类型，此字段类型理解为以这样的一种方式进行查询dsl和存储这些查询，在定义了percolate查询时它可以用于以后的匹配文档。

在percolator中登记一个查询：

PUT /my-index/queries/1?refresh

{

"query" : {

"match" : {

"message" : "bonsai tree"

}

}

}

在登记percolator的查询中匹配文档：

GET /my-index/\_search

{

"query" : {

"percolate" : {

"field" : "query",

"document\_type" : "doctype",

"document" : {

"message" : "A new bonsai tree in the office"

}

}

}

}

响应为：

{

"took": 13,

"timed\_out": false,

"\_shards": {

"total": 5,

"successful": 5,

"failed": 0

},

"hits": {

"total": 1,

"max\_score": 0.5716521,

"hits": [

{ https://static.oschina.net/uploads/img/201611/20230530_nA7M.png

"\_index": "my-index",

"\_type": "queries",

"\_id": "1",

"\_score": 0.5716521,

"\_source": {

"query": {

"match": {

"message": "bonsai tree"

}

}

}

}

]

}

}

在https://static.oschina.net/uploads/img/201611/20230530_nA7M.png处的查询匹配文档。

查询的参数：

field：定义percolator字段类型的字段，必填。

document\_type：映射的稳定字段，必填。

document：需要匹配的原始文档。document文档同时也可以是存储在索引中的文档。在这种情况下，文档参数可以被替换为以下参数：index，type，id，routing，preference，version。

在前面的例子的基础上，插入我们要percolate的文件索引：

PUT /my-index/message/1

{

"message" : "A new bonsai tree in the office"

}

返回值：

{

"\_index": "my-index",

"\_type": "message",

"\_id": "1",

"\_version": 1,

"\_shards": {

"total": 2,

"successful": 1,

"failed": 0

},

"created": true,

"result": "created"

}

Percolating已经存在的文档，可以使用搜索索引，类型，id等方式进行替换document参数进行查询，例如：

GET /my-index/\_search

{

"query" : {

"percolate" : {

"field": "query",

"document\_type" : "doctype",

"index" : "my-index",

"type" : "message",

"id" : "1",

"version" : 1 https://static.oschina.net/uploads/img/201611/20230530_nA7M.png

}

}

}

这个时候document参数被替换成了index，type，id，version。

version是可选的，但在某些情况下是有用的。

这个搜索的放回值和之前的返回值是一样的。

本文由赛克 蓝德(secisland)原创，转载请标明作者和出处。

**percolate查询高亮显示**

percolate查询同时也支持高亮显示，例如保存两查询：查询1

PUT /my-index/queries/1?refresh

{

"query" : {

"match" : {

"message" : "brown fox"

}

}

}

查询2

PUT /my-index/queries/2?refresh

{

"query" : {

"match" : {

"message" : "lazy dog"

}

}

}

高亮查询设置：

GET /my-index/\_search

{

"query" : {

"percolate" : {

"field": "query",

"document\_type" : "doctype",

"document" : {

"message" : "The quick brown fox jumps over the lazy dog"

}

}

},

"highlight": {

"fields": {

"message": {}

}

}

}

返回值如下：

{

"took": 7,

"timed\_out": false,

"\_shards": {

"total": 5,

"successful": 5,

"failed": 0

},

"hits": {

"total": 2,

"max\_score": 0.5446649,

"hits": [

{

"\_index": "my-index",

"\_type": "queries",

"\_id": "2",

"\_score": 0.5446649,

"\_source": {

"query": {

"match": {

"message": "lazy dog"

}

}

},

"highlight": {

"message": [

"The quick brown fox jumps over the <em>lazy</em> <em>dog</em>" https://static.oschina.net/uploads/img/201611/20230530_nA7M.png

]

}

},

{

"\_index": "my-index",

"\_type": "queries",

"\_id": "1",

"\_score": 0.5446649,

"\_source": {

"query": {

"match": {

"message": "brown fox"

}

}

},

"highlight": {

"message": [

"The quick <em>brown</em> <em>fox</em> jumps over the lazy dog" https://static.oschina.net/uploads/img/201611/20230531_RWI4.png

]

}

}

]

}

}

## 5.0中索引和文档接口的变化

**索引接口变化**

当运行索引映射的时候禁止关闭或删除索引操作。

在5.0之前的版本，当索引正在进行映射操作的时候，关闭索引或者删除索引会导致映射失败。在5.0中，如果索引正在进行映射操作，则禁止请求关闭索引或者删除索引操作。但这种行为对部分映射行为和以前的方式没有变化。

由于基于磁盘的规则和默认的文档值发生了几处变化，导致warmers已经没有作用了，因此warmer和warmer API接口被删除。

当更新到5.0时候，系统会直接忽略掉定义在索引中的warmers。

在节点统计和精简节点接口中的OS统计中添加了CPU的利用率。在返回节点统计的接口对象中增加了一个新的对象。这个对象包括cpu的利用率和 load\_average列。在之前的os对象中和cpu对象中移除了load\_average列。现在load\_average会分别输出1分钟，5分钟和15分钟的cpu利用率，如果返回值中没有任何数据，表示这个节点不可用。

精简(cat)节点接口返回的cpu列被移除，取而代之的是load\_1m, load\_5m, load\_15m表示1分钟，五分钟和十五分钟的利用率。如果返回值中没有任何数据，表示这个节点不可用。

org.elasticsearch.monitor.os.OsStats这个类做了修改，移除了getLoadAverage方法。现在在OsStats中增加了内部类Cpu，通过Cpu的getLoadAverage方法获取数据，返回的值不是一个double类型，而是一个对象类型，这个对象包括了一分钟，五分钟和十五分钟你的利用率，Cpu方法增加了getPercent方法来返回当前的利用率。

索引统计中的建议统计被移到搜索统计中。作为搜索统计中的一部分。

为了减少多索引操作的困惑，在创建索引的时候不能以+或者-开头。之前建的索引还可以正常使用。

索引别名将不在支持索引路由(index\_routing)，同时也不支持搜索路由(search\_routing)。因为这两个接口并没有完全测试，同时我们希望直接对索引进行这些操作。

在5.0创建文档接口中用op\_type=create参数将不在支持具体id。

当调用\_flush接口等待的时候wait\_if\_ongoing标志始终返回true，因为在同一时刻有可能有别操作对同一个分片进行刷新。如果返回false的时候，其他操作对同一个分片刷新则会导致正在进行刷新操作被终止而不会返回任何错误。

本文由赛克 蓝德(secisland)原创，转载请标明作者和出处。

**文档接口变化**

对文档的?refresh操作将不在支持truthy和falsy值。

在创建文档接口中返回值中取消了created参数，现在返回的内容是：当创建文档的时候返回"operation": "create"，当修改文档的时候返回"operation": "index"。当用bluk操作的时候返回true。

在删除接口的返回值中取消了found参数，取而代之的是"operation": "deleted"。当返回值为"operation": "noop"的时候表示没有找到文档。当用bluk操作的时候返回true。

在5.0之前的版本\_reindex和\_update\_by\_query only接口在bulk失败的时候仅仅返回retried的值。例如，返回值：

{

...

"retries": 10

...

}

现在当搜索失败的时候也会返回retries值。

{

...

"retries": {

"bulk": 10,

"search": 1

}

...

}

getAPI接口中当最后一次刷新后，如果文档进行了修改，而这个时候下一次刷新还没有执行，这个时候get接口就会产生问题。如果相同的文档频繁的更新将会产生冲突，这种情况下，可以使用realtime=false参数进行请求。

mget API接口中字段fields被重新命名为stored\_fields。

Update和bulk接口中的fields被取消，需要使用 \_source来加载字段

## 5.0参数配置的变化

配置参数在5.0中在生效前将会进行校验。在启动节点的时候，节点级别的设置和默认的节点设置都会被验证；动态设置的集群和索引设置在增加或者修改集群状态之前将被校验。

所有配置都注册在节点或者使用的传出客户端。客户自定义的插件的配置注册在插件加载的地方，注册的方法是SettingsModule类的registerSettings方法。

Index Level Settings

在之前的版本中，指定索引级别的设置在elasticsearch.yaml文件中配置，或者在启动命令行中配置。从5.0以后仅仅选择的设置在节点层生效，其他设置可以设置在具体的索引中。索引模板被在每个索引上的默认值替换。

Node settings

名称设置被取消，被node.name替换，-Dname=some\_node\_name方式制定名称将被取消。

node.add\_id\_to\_custom\_path配置参数修改名称为add\_lock\_id\_to\_custom\_path。

node.name的默认名称是节点id的前7个字母，节点id是随机产生的UUID。

node.mode和node.local两个设置被移除。本地模式通过discovery.type: local和transport.type:local来进行配置。可以通过http.enabled: false来禁止http访问。

Node attribute settings

节点级别的属性可以通过过滤来分配，可以通过node.attr前缀设置其他节点识别或者强制意识。在之前的版本是通过在节点上设置特殊节点的属性。除了这三个node.master, node.data ,node.ingest must属性外，可以通过新的node.attr. namespace命名空间移除。

Node types settings

node.client设置被移除，如果节点设置了node.client节点将无法启动。可以通过分别设置node.master, node.data,node.ingest来设置节点的类型。

Gateway settings

gateway.format设置被取消。默认用format设置smile。

Transport Settings

transport.netty.bind\_host设置被取消，用transport.bind\_host来替换。

Security manager settings

安全管理security.manager.enabled设置被移除，为了授权elasticsearch 用户特殊的许可，需要编辑本地java安全策略。

Network settings

设置network.host的值为\_non\_loopback\_value表示任意选择第一接口不标记为环回。相反，可以指定地址范围(用\_local\_,\_site\_ for制定所有的环回和私有网络地址)；通过显式接口的名称，主机名或地址。

netty.epollBugWorkaround被移除，在高CPU使用率上和早期的JVM版本有个问题，需要通过netty.epollBugWorkaround配置解决，但这个问题实在java7中才有的，现在es5.0要求最低使用java8，所以此设置被移除。

在此之前，全局线程池类型可以动态调整。线程池类型有效地控制线程池的支持队列，修改这些设置需要丰富的经验，修改不好有可能会产生比较大的风险，所以删除这些配置。现在可以为每个线程池单独调整相关的线程池参数比如： keep\_alive, queue\_size等等。

Threadpool settings

建议线程池已经取消了，现在用搜索线程池来替换。

所有线程池前缀的设置从threadpool到thread\_pool。

线程池的最小参数设置从min到core。

线程池的最大参数设置从size到max。

队列的大小对于一个固定的线程池的设置必须queue\_size（所有其他的变种，以前支持的都不再支持）。

现在线程池的设置现在节点级别进行设置的。因此，它不可能通过群集设置来更新线程池。

Analysis settings

分词器的index.analysis.analyzer.default\_index将不在支持，如果想改变索引的分词器，需要修改index.analysis.analyzer.default参数进行替代。

Ping settings

此前，有三种ping超时设置：discovery.zen.initial\_ping\_timeout，discovery.zen.ping.timeout和discovery.zen.ping\_timeout。前两者已被移除，ping超时设置现在只有discovery.zen.ping\_timeout。ping超时的默认值是三秒。

discovery.zen.master\_election.filter\_client和discovery.zen.master\_election.filter\_data被移除，新的参数用discovery.zen.master\_election.ignore\_non\_master\_pings来代替。此设置用来控制在主节点选举时候的ping响应，只有在极端情况下才会使用这个参数，平时一般不用配置。

本文由赛克 蓝德(secisland)原创，转载请标明作者和出处。

Recovery settings

此版本删除了1版本中的恢复参数

index.shard.recovery.translog\_size替换为indices.recovery.translog\_size

index.shard.recovery.translog\_ops 替换为indices.recovery.translog\_ops

index.shard.recovery.file\_chunk\_size替换为indices.recovery.file\_chunk\_size

index.shard.recovery.concurrent\_streams替换为indices.recovery.concurrent\_streams

index.shard.recovery.concurrent\_small\_file\_streams替换为 indices.recovery.concurrent\_small\_file\_streams

indices.recovery.max\_size\_per\_sec替换为indices.recovery.max\_bytes\_per\_sec

如果使用以上任何这些设置，请花时间来检查配置的目的。以上所有的设置都被认为是专家级的设置，如果绝对必要的话才可以使用。如果你已经设置了以上的任何一条在集群上请使用update API和设置相应的取代钥匙。

下列设置已被删除，而不需要更换。

indices.recovery.concurrent\_small\_file\_streams和ndices.recovery.concurrent\_file\_streams回收率现在是单线程的。

Translog settings

index.translog.flush\_threshold\_ops setting将不在被支持，为了控制事物日志的冲洗将使用index.translog.flush\_threshold\_size来代替。

index.translog.fs.type中的translog类型将不再被支持，现在缓冲区是用8K缓冲区。

之前默认的事务日志是在每次index, create, update, delete, bulk 请求后操作，现在在这些操作后是不进行操作了，事实上，它可能是一个性能瓶颈。现在，index.translog.sync\_interval不接受一个小于100ms的值，防止fsyncing同时启用了异步操作。特殊值0不再被支持。

index.translog.interval已被删除。

Request Cache Settings

index.cache.query.enable和indices.cache.query.size配置被移除，取而代之是index.requests.cache.enable和indices.requests.cache.size。

indices.requests.cache.clean\_interval取代了indices.cache.clean\_interval

Field Data Cache Settings

indices.fielddata.cache.clean\_interval设置已被替换为indices.cache.clean\_interval。

Allocation settings

cluster.routing.allocation.concurrent\_recoveries设置已被替换cluster.routing.allocation.node\_concurrent\_recoveries。

Similarity settings

默认的similarity 已被重命名为classic。

Indexing settings

indices.memory.min\_shard\_index\_buffer\_size和indices.memory.max\_shard\_index\_buffer\_size设置已删除。Elasticsearch现在允许所有堆使用量的总和只要低于节点的indices.memory.index\_buffer\_size的配置的量就可以使用分片。（默认为10%的JVM堆）。

移除参数es.max-open-files。设置es.max-open-files 参数为true，可以打印系统进程打开文件的最大个数。这个设置被取消，可以从节点信息的接口获取相同的信息，如果设置得太低，则会在启动时登录一个警告。

移除参数es.netty.gathering。时间证明设置es.netty.gathering这个参数为true和false的时候都没有什么问题。

移除参数es.useLinkedTransferQueue。es.uselinkedtransferqueue可以用来控制群集服务队列的实现，在集群发现过程中处理ping的响应。这是一个未无事实证明的设置，现已删除。

缓存并发级别设置被删除。缓存并发级别设置有两个参数indices.requests.cache.concurrency\_level,indices.fielddata.cache.concurrency\_level因为他们不再适用于用于请求缓存和字段数据缓存的实现。

通过系统属性配置ES进行了调整，将不再支持通过系统属性配置。

通过命令行参数设置将取消。

用过java环境变量设置将取消

通过ES\_JAVA\_OPTS环境变量设置将取消。

现在用-Ename.of.setting来设置配置。

通过双横线设置参数将取消，例如 --name.of.setting value.of.setting，现在用-Ename.of.setting=value.of.setting来替换。

通过.properties文件配置设置将取消。ES的运行配置和日志配置将不在支持.properties文件方式(就是键值对的方式将被取消)

Discovery Settings

当节点有 network.host, network.bind\_host, network.publish\_host, transport.host, transport.bind\_host, transport.publish\_host的时候，必须设置 discovery.zen.minimum\_master\_node，因为系统被认为是在生产环境中。

Realtime get setting

action.get.realtime设置将被取消，

Memory lock settings

bootstrap.mlockall设置被改名为bootstrap.memory\_lock。

Snapshot settings

restoring快照映射参数include\_global\_state的默认值由true改为false,taking快照的默认值还是true。

代表周的时间单位w不再支持。

小数的时间值（例如，0.5s）不再支持。例如，这意味着当设定超时”0.5s”将被拒绝，应该输入“500ms”。

Elasticsearch以前版本默认为允许多个节点共享相同的数据目录(50个以上)，但这种情况有时候会混乱，当同时启动多个节点的时候，第二个节点有可能看到的是一个空的目录，然后认为集群没有任何的数据节点。这种设置在笔记本上做测试是比较好的，但这并不是生产环境中常用的方式。为了更安全的操作，默认node.max\_local\_storage\_nodes被设置为1.

Script settings

索引脚本已被存储的脚本所取代，下面的设置已被替换为：

script.indexed替换为script.stored

script.engine.\*.indexed.aggs 替换为script.engine.\*.stored.aggs

script.engine.\*.indexed.mapping 替换为script.engine.\*.stored.mapping

script.engine.\*.indexed.search 替换为script.engine.\*.stored.search

script.engine.\*.indexed.update 替换为script.engine.\*.stored.update

script.engine.\*.indexed.plugin 替换为script.engine.\*.stored.plugin

(where \* represents the script language, like groovy, mustache, painless etc.)

其中\*代表的脚本语言,例如groovy, mustache, painless等等。

Script mode settings

脚本模式设置，以前的脚本模式设置，例如："script.inline: true", "script.engine.groovy.inline.aggs: false"等等，接受广泛的“truthy”或“falsy”值。现在比较严格，只支持true和false选项。

脚本沙箱设置将被移除，之前script.inline 和script.stored可以设置为sandbox。现在只能设置set script.line: true 或者 script.stored: true。

搜索设置index.query.bool.max\_clause\_count将取消。用indices.query.bool.max\_clause\_count来设置布尔子句的最大数目

## 5.0分配的变化HTTP协议和REST接口的变化

分配的变化，主分片分配

再次之前，当指定数量的分配副本被发现时主分片被分配(这个数量由 index.recovery.initial\_shards参数指定，现在已经废弃了)。在这种情况下，如果主分片的配置只有一个副本，就是集群的定义是一个单一的分片，这意味着索引的任何分片拷贝都有可能成为主分片，甚至是一个很久以前的数据。现在通过分配分片IDs来解决这个问题。

分配给分片拷贝一个唯一的IDs作为标志。这允许集群区分有效的相同的数据和路径的多个副本，在集群启动后，只有分片的副本包含最新的数据时才可以成为主分片。

通过分配IDs来代替以前的版本号来识别分片的拷贝，在索引分片存储API中也做了相应的调整。例如：

请求：GET http://127.0.0.1:9200/secilog/\_shard\_stores

返回值：

{

...

"0": { https://www.elastic.co/guide/en/elasticsearch/reference/current/images/icons/callouts/1.png

"stores": [ https://www.elastic.co/guide/en/elasticsearch/reference/current/images/icons/callouts/2.png

{

"sPa3OgxLSYGvQ4oPs-Tajw": { https://www.elastic.co/guide/en/elasticsearch/reference/current/images/icons/callouts/3.png

"name": "node\_t0",

"transport\_address": "local[1]",

"attributes": {

"mode": "local"

}

},

"allocation\_id": "2iNySv\_OQVePRX-yaRH\_lQ", https://www.elastic.co/guide/en/elasticsearch/reference/current/images/icons/callouts/4.png

"legacy\_version": 42, https://www.elastic.co/guide/en/elasticsearch/reference/current/images/icons/callouts/5.png

"allocation" : "primary" | "replica" | "unused", https://www.elastic.co/guide/en/elasticsearch/reference/current/images/icons/callouts/6.png

"store\_exception": ... https://www.elastic.co/guide/en/elasticsearch/reference/current/images/icons/callouts/7.png

},

...

]

},

...

}

在返回的结果中，新的版本返回字段是allocation\_id，老版本返回的字段是version。这个字段可用于创建Elasticsearch的当前版本或者在集群中有效的版本中所有分片的拷贝。legacy\_version字段可以可以理解为以前版本的version字段。

路由命令在新的版本中有两条命令，allocate\_replica 和allocate\_empty\_primary。我们引入了一个先的命令allocate\_stale\_primary。allocate\_replica命令可以理解为以前设置allow\_primary为false的情况，allocate\_empty\_primary命令可以理解为以前设置allow\_primary为ture的情况。

自定义路由命令Elasticsearch不再支持注册自定义配置命令插件。

index.shared\_filesystem.recover\_on\_any\_node为true的行为进行了调整，在之前当没有分片的副本被发现时，有可能选择任意的节点，但现在为了考虑平衡进行有选择的分配节点。

分片副本的发现的行为也发生了变化，在之前如果没有满足条件的分片副本则不进行分片副本分配，但现在可以分配分配的副本。

HTTP changes

HTTP协议请求压缩始终被支持，在之前的版本中需要设置http.compressed为true才开启压缩功能。

REST API changes

在以前的REST查询中，如果有无法识别字符串参数将被忽略。从用户的角度来看，这样并不合理，但现在的版本如果有不能识别的字符串参数将返回错误。

自定义id的长度如果超过512则会被拒绝。

/\_optimize结尾的请求将被移除，取而代之的是/\_forcemerge

在GET方式的HTTP请求中/\_forcemerge将不在支持，可以换成用POST方式的请求。

创建索引的方式只能用PUT方式，之前创建所以既可以用PUT也可以用POST。现在只支持PUT方式。

判断索引是否存在的接口HEAD {index}/{type} 被替换成了{index}/\_mapping/{type}，为了兼容性5.0版本还可以使用，将在6.0版本中移除。

在 /\_cluster/stats统计返回值中去掉了mem内存部分。

在接口/\_cluster/state的routing table中移除了分片版本号。在集群状态中存储了分片的id，用选择主分片的方式来代替版本信息。

节点角色信息将不再是节点属性的一部分。节点角色在节点统计的返回值中。

禁止不带引号的JSON，此前，JSON文档被允许有不带引号的字段的名称，这种写法不是严谨的JSON格式，如果在之前的ES版本中有不带引号的字段，有些操作可能会报错，因此在jvm.options文件中增加了一个配置-Delasticsearch.json.allow\_unquoted\_field\_names。这个配置将在6.0版本中移除。

过滤接口中的char\_filters参数被命名为char\_filter。token\_filters参数将被移除，用filter代替。

Delete-By-Query插件中的DELETE /\_query请求被移除，用Delete By Query接口代替。

PUT /\_scripts/{lang}/{id}/\_create创建脚本索引被移除，用stored scripts来代替。

PUT /\_search/template/{id}/\_create创建索引模板被移除。用Pre-registered模板来代替。

有些REST接口的结尾可以增加键值对的方式，现在这种方式被移除。

在\_cluster/health以前需要大量的wait\_for\_relocating\_shards参数，现在只需要设置boolean类型的wait\_for\_no\_relocating\_shards参数，如果设置为true，表示请求将等待（直到配置超时）的集群返回之前没有分片的搬迁。默认为false，这意味着操作不会等待

## 5.0 CAT接口和打包接口的变化

CAT API changes

使用Accept报头指定响应的media类型

在精简接口API的返回media类型中，以前的版本是通过Accept头的Content-type字段来确定的，这和HTTP规范中的含义不一致，因此新版本删除了这一特性。

\_cat/nodes接口中host字段被移除。因为这个字段和ip字段的内容是一样的。

在recovery接口中增加了bytes\_recovered和files\_recovered字段，分别表示已恢复的字节数和文件的总数。

total\_files 和total\_bytes字段分别改名成为files\_total和bytes\_total。

translog字段改名为translog\_ops\_recovered，translog\_total改名为translog\_ops，translog\_percent改名为translog\_ops\_percent，这三个字段的简称分别是tor,to,top。

Changes to cat nodes API

在nodes接口中，m代表master，d代码data,i代表ingest节点类型，一个节点可以同时有多个角色，当节点没有任何角色的时候表示这个节点是一个协调角色。当这个节点是集群的主节点的时候，master列会显示\*。

Packaging

打包的变化

Apt/yum安装下载地址发生了变化，从https://packages.elastic.co到 https://artifacts.elastic.co/.

新版本启动时间会变长。在5.0版本中JVM启动参数增加了-XX:+AlwaysPreTouch标志，这个选项将在启动的时候占用JVM堆的所有内存页，这样在GC垃圾回收时间减少内存页体提交的机会。但这将增加ES的启动时间，同时增加ES的初始化内存空间。

JVM options

JAVA虚拟机选择参数配置被放在了一个新的配置文件jvm.options中。

这意味着以前的通过环境变量的设置参数都被取消了，包括ES\_MIN\_MEM, ES\_MAX\_MEM, ES\_HEAP\_SIZE, ES\_HEAP\_NEWSIZE, ES\_DIRECT\_SIZE, ES\_USE\_IPV4, ES\_GC\_OPTS, ES\_GC\_LOG\_FILE, and JAVA\_OPTS.

当通过解压ES包的时候，默认的位置在config/jvm.options中，通过Debain或者RPM包安装后的默认配置路径在/etc/elasticsearch/jvm.options中，也可以通过设置环境变量ES\_JVM\_OPTIONS来改变文件的路径。

用于Windows服务的线程堆栈大小

在之前安装成Windows服务时，安装脚本会配置线程的堆栈大小(这是服务守护进程需要)，但现在由于配置被移到了jvm.options文件中，则安装脚本不在配置线程的堆栈大小。在新版本中，在安装成Windows服务前需要修改jvm.options文件，在32位系统中添加 -Xss320k参数，在64位系统中添加-Xss1m参数。

/bin/bash文件被依赖

之前的版本，用来启动ES和运行插件的命令脚本依赖Bourne-compatible shell。在5.0中，通过RPM和Debian包安装需要依赖bash shell，bash shell的默认路径是/bin/bash。

在之前配置ES环境变量有两种方法，1是通过占位符语法${env.ENV\_VAR\_NAME}，2是没有env前缀的语法 ${ENV\_VAR\_NAME}，现在只保留了第二种写法，第一种写法被移除。

同样，通过JVM系统参数设置的ES配置也被取消。

在之前的版本如果遇到内存溢出或者其他致命错误则ES不会停止工作，但这样会可能会产生比较严重的问题，在5.0后，当遇到这些问题时候，会从新启动ES。

## 5.0中插件的变化，文件系统和路径的变化

**Plugin changes**

插件命令由bin/plugin修改为bin/elasticsearch-plugin。插件的文档结构也做了修改。所有的插件文件必须和elasticsearch评级的目录中。如果你用工具生成，这种结构会自动生成。

插件隔离

隔离(isolation)选项已被删除。每个插件都有它自己的类加载器。

网站的插件删除

网站插件已被删除。网站插件应该在Kibana插件中实现。

多播的插件删除

已删除多播。使用单播发现，或云发现插件。

自定义查询插件的实现

自定义查询插件需要在QueryParser子类中实现fromXContent方法。插件实现自定义评分功能需要在scorefunctionparser子类中实现fromxcontent方法。

Delete-By-Query插件已移除，现在在ES的核心中用Delete By Query接口来实现。当删除时间比较长的时候，可以通过cancel接口取消。

请求：POST secilog/\_delete\_by\_query

参数：

{

"query": { https://www.elastic.co/guide/en/elasticsearch/reference/current/images/icons/callouts/1.png

"match": {

"message": "some message"

}

}

}

返回值类似：

{

"took" : 147,

"timed\_out": false,

"deleted": 119,

"batches": 1,

"version\_conflicts": 0,

"noops": 0,

"retries": {

"bulk": 0,

"search": 0

},

"throttled\_millis": 0,

"requests\_per\_second": -1.0,

"throttled\_until\_millis": 0,

"total": 119,

"failures" : [ ]

}

取消接口

请求：POST \_tasks/task\_id:1/\_cancel

attachments插件被ingest-attachment插件代替

JAVA系统属性设置，在之前的JAVA系统属性设置可以在插件脚本中-D参数直接使用。现在这种情况将不在允许，现在通过ES\_JAVA\_OPTS参数进行设置。

通过path.plugins设置自定义插件路径将取消。

自定义脚本插件将通过ScriptPlugin实现，取消了以前的通过onModule实现。

自定义分词插件通过AnalysisPlugin实现，取消了以前的通过onModule实现。

自定义映射插件通过MapperPlugin 实现，取消了以前的通过onModule实现。

自定义行为插件通过ActionPlugin实现，取消了以前的通过onModule实现。

自定义RestHandler`s插件通过ActionPlugin实现，取消了以前的通过onModule实现。

自定义搜索插件通过SearchPlugin实现，取消了以前的通过onModule实现。

SearchParseElement接口将被移除。自定义请求部分只能通过扩展(ext)元素实现,通过SearchPlugin.SearchExtSpec提供在自定义解析器的附加部分插入自定义的内容，在SearchExtSpec中实现XContent语法。语法解析现在在协调节点。语法解析后的结果和搜索请求的其他部分会通过传输层到数据节点，然后存储在搜索上下文中用于以后检索。

测试自定义插件

ESIntegTestCase# pluginlist已被删除，用Arrays.asList来代替，现在它不需要所有的插件都要用java 1.8来实现。

在2.X版本索引中的元字段\_size在聚合，脚本和排序中不能使用，如果需要使用这些特性，请用5.X版本重建索引。

**文件系统相关的变化**

在2.X版本中，索引文件的子集是通过mmap打开。在5.X中，在64为系统中所有的索引文件都是通过mmap打开，这可能会增加虚拟内存使用量，但影响不大，因为这只增加了地址空间的消耗的使用量，其他实际内存的使用情况和2.X类似。

**磁盘上的数据的路径**

在之前的版本中，数据路径通过path.data进行配置，实际的路径包括集群名称。所以数据路径的全路径为$DATA\_DIR/$CLUSTER\_NAME/nodes/$nodeOrdinal。在5.0中路径中的集群名称将不建议使用，现在存储的全路径为$DATA\_DIR/nodes/$nodeOrdinal。但为了兼容性，在启动后系统会检查集群文件夹和文件夹中的数据是否存在，如果存在系统将会尽可能读取数据。这个特想将在6.0中取消。

如果多个集群实例公用一个数据路径，需要在数据路径下添加集群名称，这样就可以保证不同的集群存储到不同的路径下。

在5.0之前，当node.data: false和node.master: false设置的时候，系统是不写入任何文件到到磁盘。在5.X中，系统需要写入实例化节点ids标志，需要节点来存储这些信息。因此，所有的节点类型都将写入一个小的状态文件到他们的数据文件夹。

## 5.0中聚合和脚本的变化

**聚合的变化**

聚合的变化，数字类型在新版本中进行了重构，采用了不同的数据结构，这种结构有更好的范围查询。然而，这种结构不参与文档评分操作。为了评估在后台的匹配文档的数量数字字段需要返回到运行的查询中，这可能会导致效率的降低。

如果这个数字字段需要进行排序，范围查询，统计聚合的时候，建议用keyword字段来代替数字字段。

IP范围聚合现在，Elasticsearch支持IPv6协议，ip地址在存储的时候用了二进制存储而不是用数字进行存储。因此IP聚合的时候将不在返回数字范围。

地理信息网格聚合的时候，将不再支持size: 0为零，新版本中要求size必须是大于零的数字。

时间值不在支持小数，例如1.5h将不在被支持，应该写成90m。

**脚本的变化**

系统默认的脚本语言从Groovy 到Painless。Painless脚本语言有类似Groovy的语法，被设计为更安全和更快。从Groovy 转到Painless是非常简单的。

Groovy和Painless在语法上有个显著的不同是使用参数上。所有的Painless参数必须有一个前缀，比如：

Groovy:

{

"script\_score": {

"script": {

"lang": "groovy",

"inline": "Math.log(\_score \* 2) + my\_modifier",

"params": {

"my\_modifier": 8

}

}

}

}

Painless (my\_modifer is prefixed with params):

{

"script\_score": {

"script": {

"lang": "painless",

"inline": "Math.log(\_score \* 2) + params.my\_modifier",

"params": {

"my\_modifier": 8

}

}

}

}

 script.default\_lang的设置被移除，因为系统不在支持默认其他脚本语言。如果要使用其他脚本语言，需要在请求的参数中显示的指出来。

本文由赛克 蓝德(secisland)原创，转载请标明作者和出处。

有部分已经存储的过滤查询是没有制定具体的脚本语言，这时候默认的语言由script.legacy.default\_lang设置。

在1.X版本中定义的内联脚本/模板将被移除。基于脚本/模板的索引或者文件将被删除。

脚本和字符串参数将不再被使用，将使用脚本对象的语法来代替，这适用于update api, script sort, script\_score function, script query, scripted\_metric aggregation,script\_heuristic aggregation。

内联脚本将不在被使用。

{

"script\_score": {

"lang": "groovy",

"script": "Math.log(\_score \* 2) + my\_modifier",

"params": {

"my\_modifier": 8

}

}

}

被替换成了下面的写法。

{

"script\_score": {

"script": {

"lang": "groovy",

"inline": "Math.log(\_score \* 2) + my\_modifier",

"params": {

"my\_modifier": 8

}

}

}

}

script和script\_file参数将不在有效，由基于脚本或者模板的文件file来代替。

基于文件的脚本的用法将不再有效：

{

"script\_score": {

"script": "calculate-score",

"params": {

"my\_modifier": 8

}

}

}

现在是:

{

"script\_score": {

"script": {

"lang": "groovy",

"file": "calculate-score",

"params": {

"my\_modifier": 8

}

}

}

}

script\_id参数将不在被使用，由id来代替。

{

"script\_score": {

"script\_id": "indexedCalculateScore",

"params": {

"my\_modifier": 8

}

}

}

现在是:

{

"script\_score": {

"script": {

"id": "indexedCalculateScore",

"lang" : "groovy",

"params": {

"my\_modifier": 8

}

}

}

}

模板查询中的query参数将不在被使用，由inline参数代替。

{

"query": {

"template": {

"query": {"match\_{{template}}": {}},

"params" : {

"template" : "all"

}

}

}

}

现在是：

{

"query": {

"template": {

"inline": {"match\_{{template}}": {}},

"params" : {

"template" : "all"

}

}

}

}

搜索模板中的顶层模板template字段被替换为一致的模板/脚本对象语法。

{

"template" : {

"query": { "match" : { "{{my\_field}}" : "{{my\_value}}" } },

"size" : "{{my\_size}}"

},

"params" : {

"my\_field" : "foo",

"my\_value" : "bar",

"my\_size" : 5

}

}

现在是:

{

"inline" : {

"query": { "match" : { "{{my\_field}}" : "{{my\_value}}" } },

"size" : "{{my\_size}}"

},

"params" : {

"my\_field" : "foo",

"my\_value" : "bar",

"my\_size" : 5

}

}

索引脚本和模板

索引脚本和模板已被存储的脚本所取代，集群state中存储的脚本或者模板来代替.scripts索引。

存储脚本的默认大小不能超过65535个字节。可以通过script.max\_size\_in\_bytes参数来修改这个值。如果脚本非常大，可以考虑用native scripts本地脚本来代替存储脚本。

以前存储在.scripts索引中的脚本将不能在使用，现在ES将在集群state中读取脚本。升级到5.X后，.scripts索引依然存在，如果要继续使用次脚本，需要将存储在.scripts中的脚本迁移到集群state中。两者的语法是一样的，不需要修改。

Python迁移脚本

下面的Python脚本可以用来导入索引脚本到群集state中存储：

from elasticsearch import Elasticsearch,helpers

es = Elasticsearch([

{'host': 'localhost'}

])

for doc in helpers.scan(es, index=".scripts", preserve\_order=True):

es.put\_script(lang=doc['\_type'], id=doc['\_id'], body=doc['\_source'])

该脚本使用了官方Elasticsearch客户端因此你需要确保在环境中安装了客户端。

Perl的迁移脚本

下面的Perl脚本可以用来导入索引脚本到群集state中存储：

use Search::Elasticsearch;

my $es = Search::Elasticsearch->new( nodes => 'localhost:9200');

my $scroll = $es->scroll\_helper( index => '.scripts', sort => '\_doc');

while (my $doc = $scroll->next) {

$e->put\_script(

lang => $doc->{\_type},

id => $doc->{\_id},

body => $doc->{\_source}

);

}

该脚本使用了官方Elasticsearch客户端因此你需要确保在环境中安装了客户端。

验证迁移脚本

当已经通过之前的脚本或以其他方式迁移了，可以用以下请求来验证，例如：

GET \_cluster/state?filter\_path=metadata.stored\_scripts

响应应该包括.scripts索引中的所有脚本。当证明所有的.scripts索引都被移走了，可以删除.scripts索引。

java API索引脚本

所有与索引脚本交互的方法都已被删除。和存储脚本进行交互的java API的方法在clusteradminclient类中。如果一个字符串需要提供bytesarray类应用，则提供脚本的唯一途径是通过bytesreference实现。

脚本引擎现在只能有一种方式注册

在5.0.0之前，脚本引擎可以登记多方式。Javascript脚本语言可以登记为"lang": "js" 或者"lang": "javascript"。脚本引擎现在只能注册一个单一的语言，所有的"lang": "js"将被替换为"lang": "javascript"。

脚本的扩展文件名称只能有一种写法

之前脚本文件的扩展名有多种写法，比如"js" 和"javascript"，现在所有的扩展文件名只能有js这一种写法。

.javascript文件后缀的方法将不在支持。

Update REST接口中删除了string参数。

cript, script\_id 和scripting\_upsert查询参数将被移除。

java传输客户

TemplateQueryBuilder类将被移除。

TransportClient transportClient = TransportClient.builder()

.settings(Settings.builder().put("node.name", "node"))

.addPlugin(MustachePlugin.class)

.build();

transportClient.addTransportAddress(

new InetSocketTransportAddress(new InetSocketAddress(InetAddresses.forString("127.0.0.1"), 9300))

);

在QueryBuilders类中创建TemplateQueryBuilder实例的方法被移除。TemplateQueryBuilder的构造函数现在可以使用。

模板查询接口将不建议使用，在下个版本中将移除。

在GeoPoint脚本中将移除以下方法。

factorDistance

factorDistanceWithDefault

factorDistance02

factorDistance13

arcDistanceInKm

arcDistanceInKmWithDefault

arcDistanceInMiles

arcDistanceInMilesWithDefault

distanceWithDefault

distanceInKm

distanceInKmWithDefault

distanceInMiles

distanceInMilesWithDefault

geohashDistanceInKm

geohashDistanceInMiles

用下面的方法进行代替。

arcDistance, arcDistanceWithDefault, planeDistance, planeDistanceWithDefault, geohashDistance, geohashDistanceWithDefault