尚硅谷 Elasticsearch6.0

版本：V2.0

一、ElasticSearch 简介

1.1、Elasticsearch

Elasticsearch 是一个基于 Apache Lucene(TM)的开源搜索引擎。无论在开源还是专有领域，Lucene 可以被认为是迄今为止最先进、性能最好的、功能最全的搜索引擎库。

特点：

- 分布式的实时文件存储，每个字段都被索引并可被搜索
- 分布式的实时分析搜索引擎--做不规则查询
- 可以扩展到上百台服务器，处理 PB 级结构化或非结构化数据

Elasticsearch 也使用 Java 开发并使用 Lucene 作为其核心来实现所有索引和搜索的功能，但是它的目的是通过简单的 RESTful API 来隐藏 Lucene 的复杂性，从而让全文搜索变得简单。

ES 能做什么？

全文检索（全部字段）、模糊查询（搜索）、数据分析（提供分析语法，例如聚合）

1.2、Elasticsearch 使用案例

（1）2013 年初，GitHub 抛弃了 Solr，采取 ElasticSearch 来做 PB 级的搜索。 “GitHub 使用 ElasticSearch 搜索 20TB 的数据，包括 13 亿文件和 1300 亿行代码”

（2）维基百科：启动以 elasticsearch 为基础的核心搜索架构 SoundCloud：“SoundCloud 使用 ElasticSearch 为 1.8 亿用户提供即时而精准的音乐搜索服务”

（3）百度：百度目前广泛使用 ElasticSearch 作为文本数据分析，采集百度所有服务器上的各类指标数据及用户自定义数据，通过对各种数据进行多维分析展示，辅助定位分析实例异常或业务层面异常。目前覆盖百度内部 20 多个业务线（包括 casio、云分析、网盟、预测、文库、直达号、钱包、风控等），单集群最大 100 台机器，200 个 ES 节点，每天导入 30TB+数据

（4）新浪使用 ES 分析处理 32 亿条实时日志

（5）阿里使用 ES 构建挖财自己的日志采集和分析体系

1.3、同类产品

Solr、ElasticSearch、Hermes（腾讯）（实时检索分析）

- Solr、ES

1. 源自搜索引擎，侧重搜索与全文检索。
2. 数据规模从几百万到千万不等，数据量过亿的集群特别少。

有可能存在个别系统数据量过亿，但这并不是普遍现象（就像 Oracle 的表里的数据规模有可能超过 Hive 里一样，但需要小型机）。

- Hermes

1. 一个基于大索引技术的海量数据实时检索分析平台。侧重数据分析。
2. 数据规模从几亿到万亿不等。最小的表也是千万级别。

在 腾讯 17 台 TS5 机器，就可以处理每天 450 亿的数据(每条数据 1kb 左右)，数据可以保存一个月之久。

- Solr、ES 区别

全文检索、搜索、分析。基于 lucene

1. Solr 利用 Zookeeper 进行分布式管理，而 Elasticsearch 自身带有分布式协调管理功能;
2. Solr 支持更多格式的数据，而 Elasticsearch 仅支持 json 文件格式;
3. Solr 官方提供的功能更多，而 Elasticsearch 本身更注重于核心功能，高级功能多有第三方插件提供;
4. Solr 在传统的搜索应用中表现好于 Elasticsearch，但在处理实时搜索应用时效率明显低于 Elasticsearch-----附近的人

Lucene 是一个开放源代码的全文检索引擎工具包，但它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

搜索引擎产品简介

二、ElasticSearch

2.1、准备工作

安装 Centos7、建议内存 2G 以上、安装 java1.8 环境

2.2、基本配置

- 设置 IP 地址

vi /etc/sysconfig/network-scripts/ifcfg-ens33

```
TYPE="Ethernet"
BOOTPROTO="static"
DEFROUTE="yes"
PEERDNS="yes"
PEERROUTES="yes"
IPV4_FAILURE_FATAL="no"
IPV6INIT="yes"
IPV6_AUTOCONF="yes"
IPV6_DEFROUTE="yes"
IPV6_PEERDNS="yes"
IPV6_PEERROUTES="yes"
IPV6_FAILURE_FATAL="no"
NAME="eno16777728"
UUID="3fcc8bea-f99d-427d-ae73-ce92f501a8b8"
DEVICE="eno16777728"
ONBOOT="yes"
IPADDR=192.168.127.128
NETMASK=255.255.255.0
GATEWAY=192.168.127.2
```

网络重置
service network restart

- 添加用户

[root@localhost ~]# adduser elk

[root@localhost ~]# passwd elk

以下授权步骤可省略

[root@localhost ~]# whereis sudoers

[root@localhost ~]# ls -l /etc/sudoers

[root@localhost ~]# vi /etc/sudoers

```
## Allow root to run any commands anywher
root        ALL=(ALL)              ALL
linuxidc    ALL=(ALL)              ALL    #这个是新增的用户
```

[root@localhost ~]# chmod -v u-w /etc/sudoers

[root@localhost ~]# su elk

2.3、Java 环境安装

- 解压安装包

[root@localhost jdk1.8]# tar -zxvf jdk-8u171-linux-x64.tar.gz

- 设置 Java 环境变量

[root@localhost jdk1.8.0_171]# vi /etc/profile

在文件最后添加

```
export JAVA_HOME=/home/elk1/jdk1.8/jdk1.8.0_171
export JRE_HOME=$JAVA_HOME/jre
export CLASSPATH=.:$JAVA_HOME/LIB:$JRE_HOME/LIB:$CLASSPATH
export PATH=$JAVA_HOME/bin:$JRE_HOME/bin:$PATH
```

[root@localhost jdk1.8.0_171]# source /etc/profile
[root@localhost jdk1.8.0_171]# java -version
java version "1.8.0_171"
Java(TM) SE Runtime Environment (build 1.8.0_171-b11)
Java HotSpot(TM) 64-Bit Server VM (build 25.171-b11, mixed mode)

2.4、ElasticSerach 单机安装

192.168.14.10 root elk
/home/elk/soft

[root@localhost elasticserach]# tar -zxvf elasticsearch-6.3.1.tar.gz

[root@localhost elasticserach]# cd elasticsearch-6.3.1/bin

[root@localhost bin]# ./elasticsearch

```
[root@localhost bin]# ./elasticsearch
[2018-07-13T15:22:41,083][WARN ][o.e.b.ElasticsearchUncaughtExceptionHandler] [] uncaught exception in thread [main]
org.elasticsearch.bootstrap.StartupException: java.lang.RuntimeException: can not run elasticsearch as root
        at org.elasticsearch.bootstrap.Elasticsearch.init(Elasticsearch.java:140) ~[elasticsearch-6.3.1.jar:6.3.1]
        at org.elasticsearch.bootstrap.Elasticsearch.execute(Elasticsearch.java:127) ~[elasticsearch-6.3.1.jar:6.3.1]
        at org.elasticsearch.cli.EnvironmentAwareCommand.execute(EnvironmentAwareCommand.java:86) ~[elasticsearch-6.3.1.jar:6.3.1]
        at org.elasticsearch.cli.Command.mainWithoutErrorHandling(Command.java:124) ~[elasticsearch-cli-6.3.1.jar:6.3.1]
        at org.elasticsearch.cli.Command.main(Command.java:90) ~[elasticsearch-cli-6.3.1.jar:6.3.1]
        at org.elasticsearch.bootstrap.Elasticsearch.main(Elasticsearch.java:93) ~[elasticsearch-6.3.1.jar:6.3.1]
        at org.elasticsearch.bootstrap.Elasticsearch.main(Elasticsearch.java:86) ~[elasticsearch-6.3.1.jar:6.3.1]
Caused by: java.lang.RuntimeException: can not run elasticsearch as root
        at org.elasticsearch.bootstrap.Bootstrap.initializeNatives(Bootstrap.java:104) ~[elasticsearch-6.3.1.jar:6.3.1]
        at org.elasticsearch.bootstrap.Bootstrap.setup(Bootstrap.java:171) ~[elasticsearch-6.3.1.jar:6.3.1]
        at org.elasticsearch.bootstrap.Bootstrap.init(Bootstrap.java:326) ~[elasticsearch-6.3.1.jar:6.3.1]
        at org.elasticsearch.bootstrap.Elasticsearch.init(Elasticsearch.java:136) ~[elasticsearch-6.3.1.jar:6.3.1]
        ... 6 more
```

[root@localhost bin]# su elk1
[elk1@localhost bin]$ ./elasticsearch

```
[elk1@localhost bin]$ ./elasticsearch
Exception in thread "main" java.nio.file.AccessDeniedException: /home/elk1/elasticserach/elasticsearch-6.3.1/config/jvm.options
        at sun.nio.fs.UnixException.translateToIOException(UnixException.java:84)
        at sun.nio.fs.UnixException.rethrowAsIOException(UnixException.java:102)
        at sun.nio.fs.UnixException.rethrowAsIOException(UnixException.java:107)
        at sun.nio.fs.UnixFileSystemProvider.newByteChannel(UnixFileSystemProvider.java:214)
        at java.nio.file.Files.newByteChannel(Files.java:361)
        at java.nio.file.Files.newByteChannel(Files.java:407)
        at java.nio.file.spi.FileSystemProvider.newInputStream(FileSystemProvider.java:384)
        at java.nio.file.Files.newInputStream(Files.java:152)
        at org.elasticsearch.tools.launchers.JvmOptionsParser.main(JvmOptionsParser.java:58)
```

[root@localhost bin]# chown -R elk1:elk1 /home/elk1/elasticsearch

[elk1@localhost bin]$ ./elasticsearch

[elk1@localhost config]$ vi jvm.options

```
## See https://www.elastic.co/guide/en/elasticsearch/reference/current/heap-size.html
## for more information
##
################################################################

# Xms represents the initial size of total heap space
# Xmx represents the maximum size of total heap space

-Xms2g
-Xmx2g
```

[elk1@localhost bin]$ ./elasticsearch

```
[2018-07-13T16:05:00,979][INFO ][o.e.p.PluginsService     ] [_uHU_cC] loaded module [tribe]
[2018-07-13T16:05:00,979][INFO ][o.e.p.PluginsService     ] [_uHU_cC] loaded module [x-pack-core]
[2018-07-13T16:05:00,979][INFO ][o.e.p.PluginsService     ] [_uHU_cC] loaded module [x-pack-deprecation]
[2018-07-13T16:05:00,979][INFO ][o.e.p.PluginsService     ] [_uHU_cC] loaded module [x-pack-graph]
[2018-07-13T16:05:00,979][INFO ][o.e.p.PluginsService     ] [_uHU_cC] loaded module [x-pack-logstash]
[2018-07-13T16:05:00,980][INFO ][o.e.p.PluginsService     ] [_uHU_cC] loaded module [x-pack-ml]
[2018-07-13T16:05:00,980][INFO ][o.e.p.PluginsService     ] [_uHU_cC] loaded module [x-pack-monitoring]
[2018-07-13T16:05:00,980][INFO ][o.e.p.PluginsService     ] [_uHU_cC] loaded module [x-pack-rollup]
[2018-07-13T16:05:00,980][INFO ][o.e.p.PluginsService     ] [_uHU_cC] loaded module [x-pack-security]
[2018-07-13T16:05:00,980][INFO ][o.e.p.PluginsService     ] [_uHU_cC] loaded module [x-pack-sql]
[2018-07-13T16:05:00,981][INFO ][o.e.p.PluginsService     ] [_uHU_cC] loaded module [x-pack-upgrade]
[2018-07-13T16:05:00,981][INFO ][o.e.p.PluginsService     ] [_uHU_cC] loaded module [x-pack-watcher]
[2018-07-13T16:05:00,981][INFO ][o.e.p.PluginsService     ] [_uHU_cC] no plugins loaded
[2018-07-13T16:05:13,853][INFO ][o.e.x.s.a.s.FileRolesStore] [_uHU_cC] parsed [0] roles from file [/home/elk1/elasticserach/elasticsearch-6.3.1/config/roles.yml]
[2018-07-13T16:05:15,570][INFO ][o.e.x.m.j.p.l.CppLogMessageHandler] [controller/10016] [Main.cc@109] controller (64 bit): Version 6.3.1 (Build 4d0b8f0a0ef401) Co
[2018-07-13T16:05:17,231][DEBUG][o.e.a.ActionModule       ] Using REST wrapper from plugin org.elasticsearch.xpack.security.Security
[2018-07-13T16:05:17,756][INFO ][o.e.d.DiscoveryModule    ] [_uHU_cC] using discovery type [zen]
[2018-07-13T16:05:19,456][INFO ][o.e.n.Node               ] [_uHU_cC] initialized
[2018-07-13T16:05:19,456][INFO ][o.e.n.Node               ] [_uHU_cC] starting ...
[2018-07-13T16:05:20,082][INFO ][o.e.t.TransportService   ] [_uHU_cC] publish_address {127.0.0.1:9300}, bound_addresses {[::1]:9300}, {127.0.0.1:9300}
[2018-07-13T16:05:20,234][WARN ][o.e.b.BootstrapChecks    ] [_uHU_cC] max file descriptors [4096] for elasticsearch process is too low, increase to at least [6553
[2018-07-13T16:05:20,234][WARN ][o.e.b.BootstrapChecks    ] [_uHU_cC] max number of threads [3818] for user [elk1] is too low, increase to at least [4096]
[2018-07-13T16:05:20,234][WARN ][o.e.b.BootstrapChecks    ] [_uHU_cC] max virtual memory areas vm.max_map_count [65530] is too low, increase to at least [262144]
```

[root@localhost jdk1.8.0_171]# curl 127.0.0.1:9200

```
[root@localhost jdk1.8.0_171]# curl 127.0.0.1:9200
{
  "name" : "_uHU_cC",
  "cluster_name" : "elasticsearch",
  "cluster_uuid" : "mqFXQFsuSrKQpYtW8wWJYw",
  "version" : {
    "number" : "6.3.1",
    "build_flavor" : "default",
    "build_type" : "tar",
    "build_hash" : "eb782d0",
    "build_date" : "2018-06-29T21:59:26.107521Z",
    "build_snapshot" : false,
    "lucene_version" : "7.3.1",
    "minimum_wire_compatibility_version" : "5.6.0",
    "minimum_index_compatibility_version" : "5.0.0"
  },
  "tagline" : "You Know, for Search"
}
```

#后台启动

[elk1@localhost bin]$ ./elasticsearch -d

#关闭程序

[elk1@localhost bin]$ ps -ef|grep elastic

```
[elk1@localhost bin]$ ps -ef|grep elastic
elk1      10097      1  8 16:07 pts/0    00:00:34 /home/elk1/jdk1.8/jdk1.8.0_171/bin/java -Xms2g -Xmx2g -XX:+UseConcMarkSwee
aysPreTouch -Xss1m -Djava.awt.headless=true -Dfile.encoding=UTF-8 -Djna.nosys=true -XX:-OmitStackTraceInFastThrow -Dio.netty
rThread=0 -Dlog4j.shutdownHookEnabled=false -Dlog4j2.disable.jmx=true -Djava.io.tmpdir=/tmp/elasticsearch.FJ7pocrL -XX:+Heap
+PrintGCDetails -XX:+PrintGCDateStamps -XX:+PrintTenuringDistribution -XX:+PrintGCApplicationStoppedTime -Xloggc:logs/gc.log
home=/home/elk1/elasticserach/elasticsearch-6.3.1 -Des.path.conf=/home/elk1/elasticserach/elasticsearch-6.3.1/config -Des.di
sticsearch-6.3.1/lib/* org.elasticsearch.bootstrap.Elasticsearch -d
elk1      10348   2340  0 16:14 pts/0    00:00:00 grep --color=auto elastic
```

[elk1@localhost bin]$ kill 10097

#设置浏览器访问

[root@localhost bin]systemctl stop firewalld

[root@localhost bin]vi config/elasticsearch.yml

```
# Elasticsearch performs poorly when the system is swapping the memory.
#
# ---------------------------------- Network -----------------------------------
#
# Set the bind address to a specific IP (IPv4 or IPv6):
#
network.host: 192.168.14.13
#
# Set a custom port for HTTP:
#
#http.port: 9200
```

安装问题：

```
ERROR: [3] bootstrap checks failed
[1]: max file descriptors [4096] for elasticsearch process is too low, increase to at least [65536]
[2]: max number of threads [3818] for user [elk1] is too low, increase to at least [4096]
[3]: max virtual memory areas vm.max_map_count [65530] is too low, increase to at least [262144]
[2018-07-13T16:24:42,964][INFO ][o.e.n.Node               ] [_uHU_cC] stopping ...
[2018-07-13T16:24:43,183][INFO ][o.e.n.Node               ] [_uHU_cC] stopped
[2018-07-13T16:24:43,183][INFO ][o.e.n.Node               ] [_uHU_cC] closing ...
[2018-07-13T16:24:43,228][INFO ][o.e.n.Node               ] [_uHU_cC] closed
[2018-07-13T16:24:43,252][INFO ][o.e.x.m.j.p.NativeController] Native controller process has stopped -
```

[1] [2]解决方案
[root@localhost bin]# vi /etc/security/limits.conf

```
#@student        hard    nproc           20
#@faculty        soft    nproc           20
#@faculty        hard    nproc           50
#ftp             hard    nproc           0
#@student        -       maxlogins       4

* hard nofile 65536
* soft nofile 131072
* hard nproc 4096
* soft nproc 2048
End of file
```

*代表所有用户

nofile - 打开文件的最大数目
noproc - 进程的最大数目
soft 指的是当前系统生效的设置值
hard 表明系统中所能设定的最大值

* hard nofile 655360
* soft nofile 131072
* hard nproc 4096
* soft nproc 2048

[3] 解决方案
[root@localhost bin]# vi /etc/sysctl.conf

[root@localhost bin]# sysctl -p

vm.max_map_count=655360
fs.file-max=655360

```
# sysctl settings are defined through files in
# /usr/lib/sysctl.d/, /run/sysctl.d/, and /etc/sysctl.d/.
#
# Vendors settings live in /usr/lib/sysctl.d/.
# To override a whole file, create a new file with the same in
# /etc/sysctl.d/ and put new settings there. To override
# only specific settings, add a file with a lexically later
# name in /etc/sysctl.d/ and put new settings there.
#
# For more information, see sysctl.conf(5) and sysctl.d(5).
vm.max_map_count=655360
fs.file-max=655360
```

vm.max_map_count=655360，因此缺省配置下，单个 jvm 能开启的最大线程数为其一半
file-max 是设置 系统所有进程一共可以打开的文件数量

测试

Liunx 执行： curl 'http://localhost:9200/?pretty'

浏览器访问：http://localhost:9200/?pretty

状态查看命令

语法：ip:post/_cat/[args](?v|?format=json&pretty)

（?v 表示显示字段说明,?format=json&pretty 表示显示成 json 格式）

1、查看所有索引

GET _cat/indices?v

2、查看 es 集群状态

GET _cat/health?v

2.5、Elasticsearch 的交互方式

1、基于 HTTP 协议，以 JSON 为数据交互格式的 RESTful API

GET POST PUT DELETE HEAD

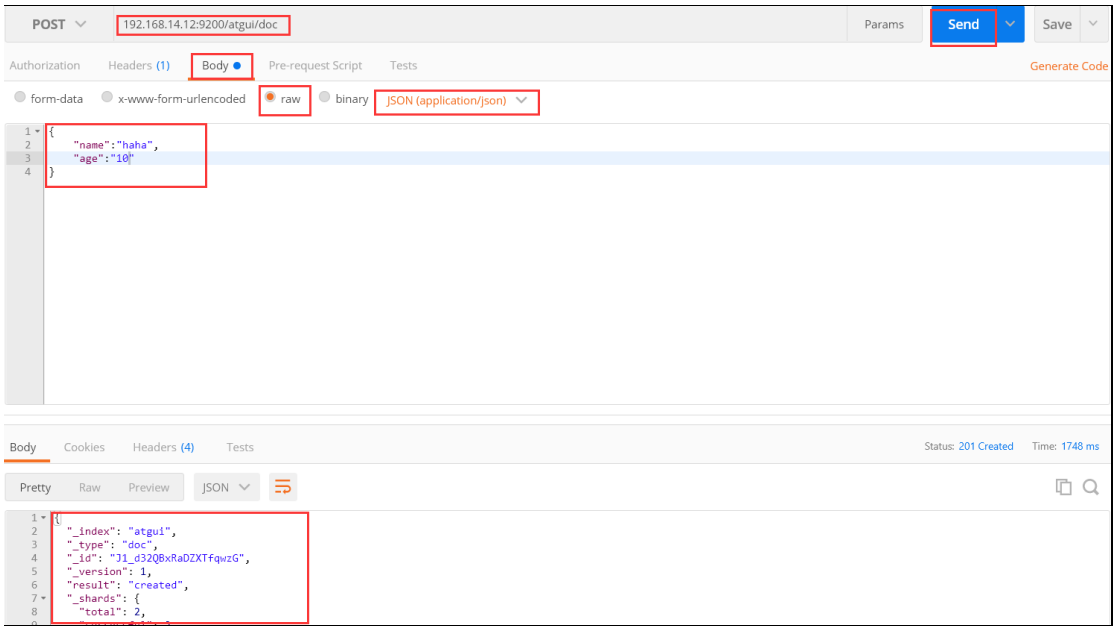
2、Elasticsearch 官方提供了多种程序语言的客户端—java，Javascript，.NET，PHP，Perl，Python，以及 Ruby——还有很多由社区提供的客户端和插件

https://www.elastic.co/guide/en/elasticsearch/client/java-rest/6.3/java-rest-high-getting-started-maven.html

2.6、Elasticsearch 操作工具

- REST 访问 ES 方式（需要 Http Method、URI）

1. 浏览器（postman）



2. Linux 命令行

请求：

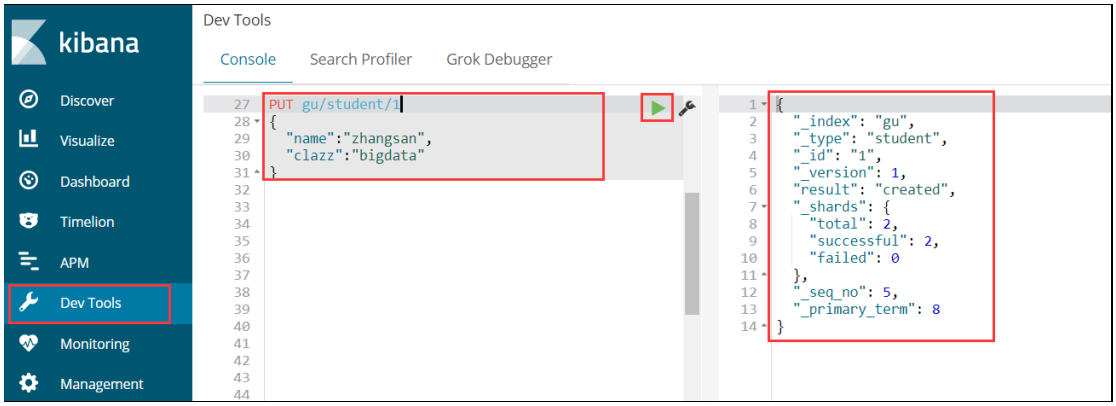
[root@localcurl -XPOST 'http://192.168.14.12:9200/atguig/doc' -i -H "Content-Type:application/json" -d '{"name":"haha","age":"10"}'

响应：

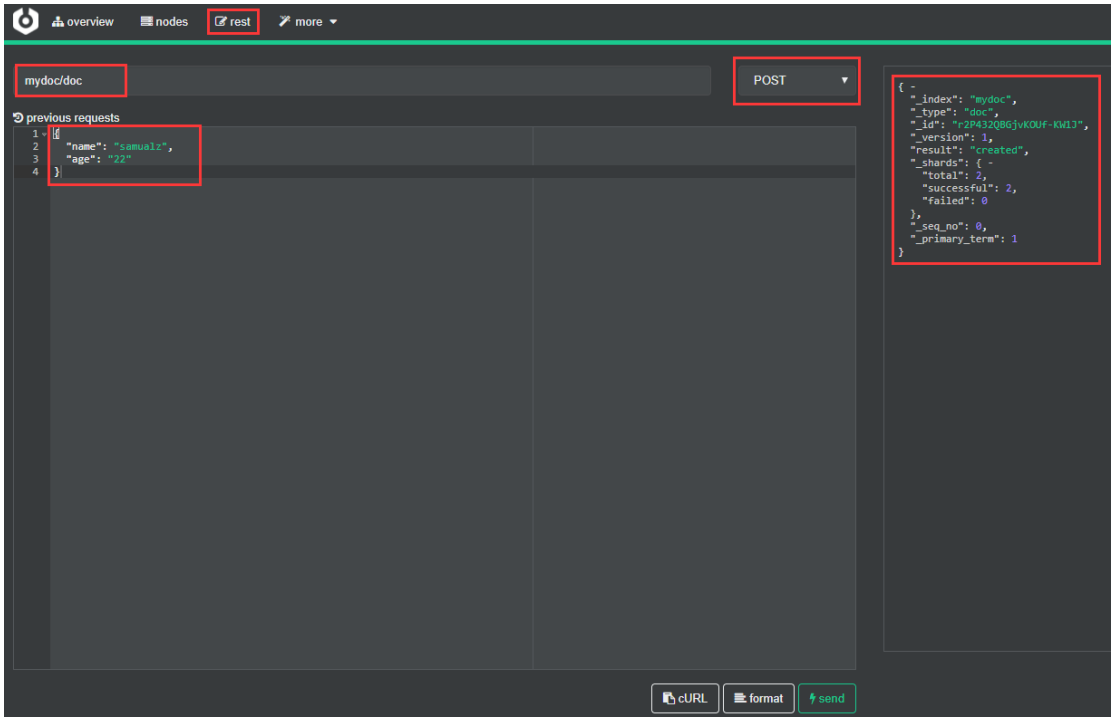
HTTP/1.1 201 Created
Location: /atguig/doc/KF_t32QBxRaDZXTftAxg
content-type: application/json; charset=UTF-8
content-length: 172

{"_index":"atguig","_type":"doc","_id":"KF_t32QBxRaDZXTftAxg","_version":1,"result": "created","_shards":{"total":2,"successful":1,"failed":0},"_seq_no":0,"_primary_term": 1}

3. Kibana 的 Dev Tools



4. Cerebro 插件



2.7、Elasticsearch 数据存储方式(基本概念)

2.7.1、Elasticsearch 存储方式

（1）面向文档

Elasticsearch 是面向文档(document oriented)的，这意味着它可以存储整个对象或文档(document)。然而它不仅仅是存储，还会索引(index)每个文档的内容使之可以被搜索。在 Elasticsearch 中，你可以对文档（而非成行成列的数据）进行索引、搜索、排序、过滤。这种理解数据的方式与以往完全不同，这也是 Elasticsearch 能够执行复杂的全文搜

索的原因之一。

（2）JSON

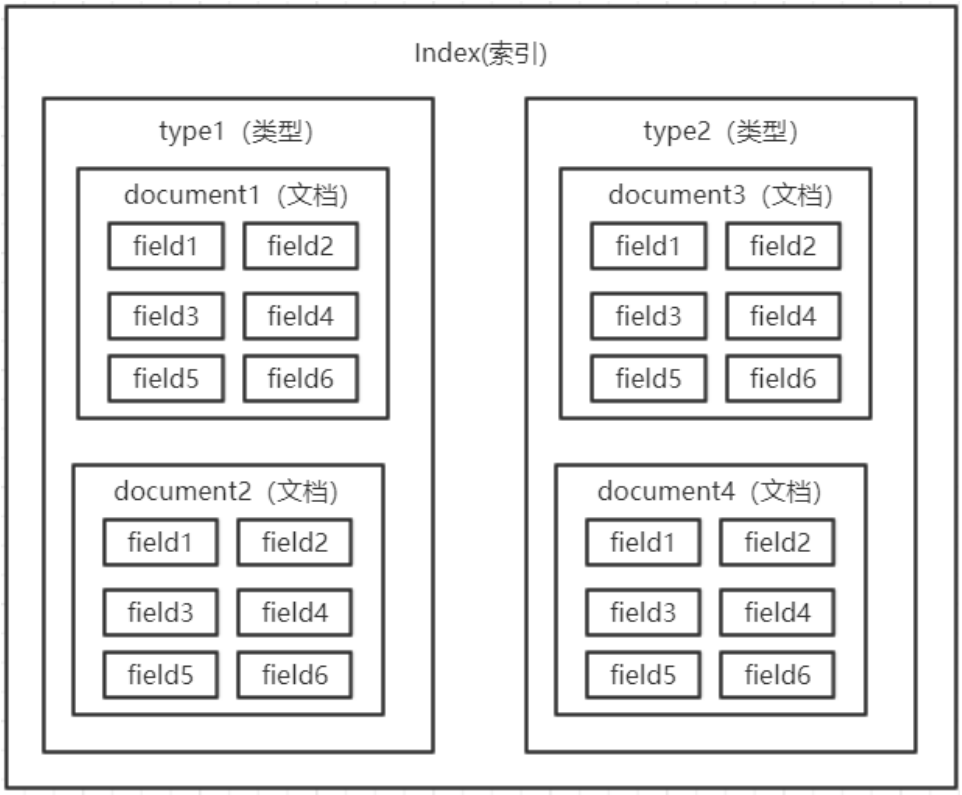
ELasticsearch 使用 Javascript 对象符号(JavaScript Object Notation)，也就是 JSON，作为文档序列化格式。JSON 现在已经被大多语言所支持，而且已经成为 NoSQL 领域的标准格式。它简洁、简单且容易阅读。

以下使用 JSON 文档来表示一个用户对象：

```
{
    "email":        "john@smith.com",
    "first_name": "John",
    "last_name":    "Smith",
    "info": {
        "bio":          "Eco-warrior and defender of the weak",
        "age":          25,
        "interests": [ "dolphins", "whales" ]
    },
    "join_date": "2014/05/01"
}
```

尽管原始的 user 对象很复杂，但它的结构和对象的含义已经被完整的体现在 JSON 中了，在 Elasticsearch 中将对象转化为 JSON 并做索引要比在表结构中做相同的事情简单的多。

2.7.2、Elasticsearch 存储结构



Mysql 数据与 ES 数据转化

（1）元数据

创建文档语句
PUT atguigu/doc
{
“name”:”zhangsan”,
“age”:10
}

_index:文档所在索引名称
_type:文档所在类型名称
_id:文档唯一 id
_uid:组合 id，由_type 和_id 组成（6.x 后，_type 不再起作用，同_id）
_source:文档的原始 Json 数据，包括每个字段的内容
_all:将所有字段内容整合起来，默认禁用（用于对所有字段内容检索）

（2）名词解释

- 索引 index

  一个索引就是一个拥有几分相似特征的文档的集合。比如说，你可以有一个客户数据的索引，另一个产品目录的索引，还有一个订单数据的索引。一个索引由一个名字来标识（必须全部是小写字母的），并且当我们要对对应于这个索引中的文档进行索引、搜索、更新和删除的时候，都要使用到这个名字。在一个集群中，可以定义任意多的索引。

- 类型 type

  Es6 之后，一个 index 中只能有一个 type

  在一个索引中，你可以定义一种或多种类型。一个类型是你的索引的一个逻辑上的分类/分区，其语义完全由你来定。通常，会为具有一组共同字段的文档定义一个类型。比如说，我们假设你运营一个博客平台并且将你所有的数据存储到一个索引中。在这个索引中，你可以为用户数据定义一个类型，为博客数据定义另一个类型，当然，也可以为评论数据定义另一个类型。

- 字段 Field

  相当于是数据表的字段，对文档数据根据不同属性进行的分类标识

- document

  一个文档是一个可被索引的基础信息单元。比如，你可以拥有某一个客户的文档，某一个产品的一个文档，当然，也可以拥有某个订单的一个文档。文档以 JSON（Javascript Object Notation）格式来表示，而 JSON 是一个到处存在的互联网数据交互格式。在一个 index/type 里面，你可以存储任意多的文档。注意，尽管一个文档，物理上存在于一个索引之中，文档必须被索引/赋予一个索引的 type。

cluster	整个 elasticsearch 默认就是集群状态，整个集群是一份完整、互备的数据。
node	集群中的一个节点，一般只一个进程就是一个 node
shard	分片，即使是一个节点中的数据也会通过 hash 算法，分成多个片存放，默认是 5 片。
index	相当于 rdbms 的 database, 对于用户来说是一个逻辑数据库，虽然物理上会被分多个 shard 存放，也可能存放在多个 node 中。
type	类似于 rdbms 的 table，但是与其说像 table，其实更像面向对象中的 class , 同一 Json 的格式的数据集合。
document	类似于 rdbms 的 row、面向对象里的 object
field	相当于字段、属性

2.8、Elasticsearch 检索

2.8.1、检索文档

Mysql : select * from user where id = 1

ES : GET /atguigu/doc/1

响应：

```
{
   "_index" :     "megacorp",
   "_type" :      "employee",
   "_id" :        "1",
   "_version" : 1,
   "found" :      true,
   "_source" :    {
         "first_name" :   "John",
         "last_name" :    "Smith",
         "age" :           25,
         "about" :         "I love to go rock climbing",
         "interests":   [ "sports", "music" ]
   }
}
```

我们通过 HTTP 方法 GET 来检索文档，同样的，我们可以使用 DELETE 方法删除文档，

使用 HEAD 方法检查某文档是否存在。如果想更新已存在的文档，我们只需再 PUT 一次。

2.8.2、简单检索

Mysql : select * from user

ES : GET /megacorp/employee/_search

响应内容不仅会告诉我们哪些文档被匹配到，而且这些文档内容完整的被包含在其中—我们在给用户展示搜索结果时需要用到的所有信息都有了。

2.8.3、全文检索

ES : GET /megacorp/employee/_search?q=haha

查询出所有文档字段值为 haha 的文档

2.8.4、搜索（模糊查询）

ES : GET /megacorp/employee/_search?q=hello

查询出所有文档字段值分词后包含 hello 的文档

2.8.5、聚合

```
PUT atguigu/_mapping/doc
{
  "properties": {
    "interests": {
      "type": "text",
      "fielddata": true
      }
  }
}
```

Group by

Elasticsearch 有一个功能叫做聚合(aggregations)，它允许你在数据上生成复杂的分析统计。它很像 SQL 中的 GROUP BY 但是功能更强大。

举个例子，让我们找到所有职员中最大的共同点（兴趣爱好）是什么：

```
GET /atguigu/doc/_search
```

```
{
  "aggs": {
    "all_interests": {
      "terms": { "field": "interests" }
    }
  }
}
```

暂时先忽略语法只看查询结果：

```
{
   ...
   "hits": { ... },
   "aggregations": {
      "all_interests": {
         "buckets": [
            {
               "key":          "music",
               "doc_count": 2
            },
            {
               "key":          "forestry",
               "doc_count": 1
            },
            {
               "key":          "sports",
               "doc_count": 1
            }
         ]
      }
   }
}
```

我们可以看到两个职员对音乐有兴趣，一个喜欢林学，一个喜欢运动。这些数据并没有被预先计算好，它们是实时的从匹配查询语句的文档中动态计算生成的。如果我们想知道所有姓"Smith"的人最大的共同点（兴趣爱好），我们只需要增加合适的语句既可：

```
GET /atguigu/doc/_search
{
  "query": {
    "match": {
      "last_name": "smith"
    }
  },
  "aggs": {
    "all_interests": {
      "terms": {
```

```
            "field": "interests"
        }
    }
  }
}
```

all_interests 聚合已经变成只包含和查询语句相匹配的文档了：

```
  ...
  "all_interests": {
      "buckets": [
          {
              "key": "music",
              "doc_count": 2
          },
          {
              "key": "sports",
              "doc_count": 1
          }
      ]
  }
```

```
PUT atguigu/_mapping/doc/
{
  "properties": {
    "interests": {
      "type":         "text",
      "fielddata": true
    }
  }
}
```

2.9、Elasticsearch 搜索原理

https://www.cs.usfca.edu/~galles/visualization/Algorithms.html

2.9.1、正排索引和倒排索引

- 正排索引
  记录文档 Id 到文档内容、单词的关联关系
  尚硅谷：1，3

docid	content
1	尚硅谷是最好的培训机构
2	php 是世界上最好的语言
3	机构尚硅谷是如何诞生的

- 倒排索引
  记录单词到文档 id 的关联关系，包含：
  单词词典（Term DicTionary）：记录所有文档的单词，一般比较大
  倒排索引（Posting List）：记录单词倒排列表的关联信息

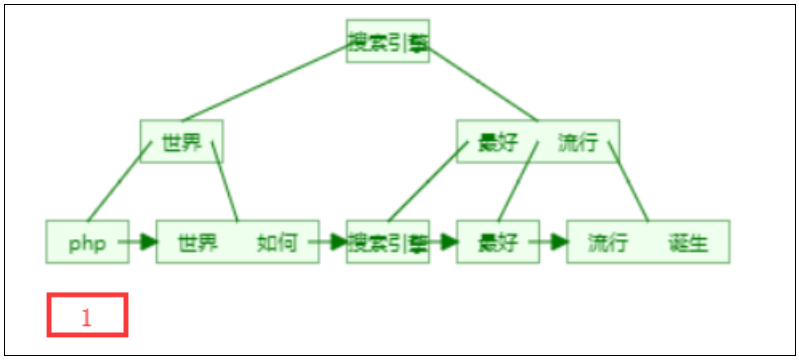
  例如：尚硅谷
  1、Term Dictionary
  尚硅谷

  2、Posting List

DocId	TF	Position	Offset
1	1	0	<0,2>
3	1	0	<0,2>

DocId：文档 id，文档的原始信息
TF：单词频率，记录该词再文档中出现的次数，用于后续相关性算分
Position：位置，记录 Field 分词后，单词所在的位置，从 0 开始
Offset：偏移量，记录单词在文档中开始和结束位置，用于高亮显示等

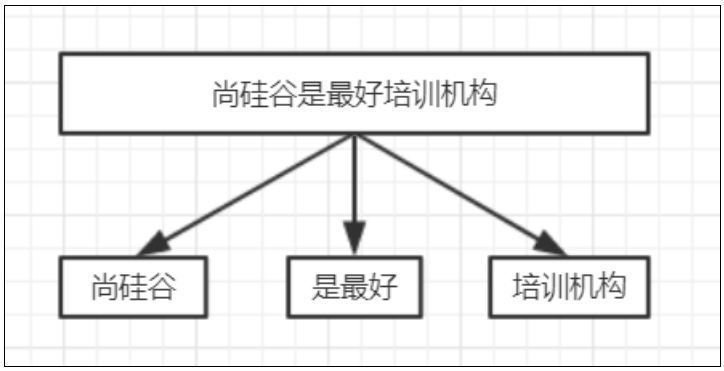
3、内存结构
https://www.cs.usfca.edu/~galles/visualization/Algorithms.html
B+Tree



每个文档字段都有自己的倒排索引

2.9.2、分词

分词是指将文本转换成一系列单词（term or token）的过程，也可以叫做文本分析，在 es 里面称为 Analysis



- 分词机制

Character Filter	对原始文本进行处理	例：去除 html 标签、特殊字符等
Tokenizer	将原始文本进行分词	例：培训机构-->培训，机构
Token Filters	分词后的关键字进行加工	例：转小写、删除语气词、近义词和同义词等

- 分词 API

1、直接指定测试（指定分词器）

Request:

```
POST _analyze
{
  "analyzer": "standard",
  "text":"hello 1111"
}
```

Response:

```
{
  "tokens": [
    {
      "token": "hello",                #分词
      "start_offset": 0,               #开始偏移
      "end_offset": 5,                 #结束偏移
      "type": "<ALPHANUM>",            #单词类型
      "position": 0                    #位置
    },
    {
      "token": "world",
      "start_offset": 6,
      "end_offset": 11,
      "type": "<NUM>",
      "position": 1
    }
```

```
        ]
    }
```

2、针对索引的字段进行分词测试（利用该字段的分词器）

Request：

```
POST atguigu/_analyze
{
  "field": "name",
  "text":"hello world"
}
```

Response:

```
{
  "tokens": [
    {
      "token": "hello",
      "start_offset": 0,
      "end_offset": 5,
      "type": "<ALPHANUM>",
      "position": 0
    },
    {
      "token": "world",
      "start_offset": 6,
      "end_offset": 11,
      "type": "<ALPHANUM>",
      "position": 1
    }
  ]
}
```

3、自定义分词器

Request:

```
POST _analyze
{
  "tokenizer": "standard",
  "filter": ["lowercase"],
  "text":"Hello WORLD"
}
```

Response:

```
{
```

```
  "tokens": [
    {
      "token": "hello",
      "start_offset": 0,
      "end_offset": 5,
      "type": "<ALPHANUM>",
      "position": 0
    },
    {
      "token": "world",
      "start_offset": 6,
      "end_offset": 11,
      "type": "<ALPHANUM>",
      "position": 1
    }
  ]
}
```

- Elasticsearch 自带的分词器

分词器（Analyzer）	特点
Standard（es 默认）	支持多语言，按词切分并做小写处理
Simple	按照非字母切分，小写处理
Whitespace	按照空格来切分
Stop	去除语气助词，如 the、an、的、这等
Keyword	不分词
Pattern	正则分词，默认\w+,即非字词符号做分割符
Language	常见语言的分词器（30+）

- 中文分词

分词器名称	介绍	特点	地址
IK	实现中英文单词切分	自定义词库	https://github.com/medcl/elasticsearch-analysis-ik
Jieba	python 流行分词系统，支持分词和词性标注	支持繁体、自定义、并行分词	http://github.com/sing1ee/elasticsearch-jieba-plugin
Hanlp	由一系列模型于算法组成的 java 工具包	普及自然语言处理在生产环境中的应用	https://github.com/hankcs/HanLP
THULAC	清华大学中文词法分析工具包	具有中文分词和词性标注功能	https://github.com/microbun/elasticsearch-thulac-plugin

- Character Filters

  在进行 Tokenizer 之前对原始文本进行处理，如增加、删除或替换字符等

HTML Strip	去除 html 标签和转换 html 实体
Mapping	字符串替换操作
Pattern Replace	正则匹配替换

注意：进行处理后，会影响后续 tokenizer 解析的 position 和 offset

Request：

```
POST _analyze
{
  "tokenizer": "keyword",
  "char_filter": ["html_strip"],
  "text":"<div><h1>B<sup>+</sup>Trees</h1></div>"
}
```

Response:

```
{
  "tokens": [
    {
      "token": """


B+Trees


""",
      "start_offset": 0,
      "end_offset": 38,
      "type": "word",
      "position": 0
    }
  ]
}
```

- Token Filter

  对输出的单词（term）进行增加、删除、修改等操作

Lowercase	将所有 term 转换为小写
stop	删除 stop words
NGram	和 Edge NGram 连词分割
Synonym	添加近义词的 term

Request:

```
POST _analyze
```

```
{
  "tokenizer": "standard",
  "text":"a Hello World",
  "filter": [
        "stop",
        "lowercase",
        {
          "type":"ngram",
          "min_gram":3,
          "max_gram":4

        }
      ]
}
```

Response:

```
{
  "tokens": [
    {
      "token": "hel",
      "start_offset": 2,
      "end_offset": 7,
      "type": "<ALPHANUM>",
      "position": 1
    },
    {
      "token": "hell",
      "start_offset": 2,
      "end_offset": 7,
      "type": "<ALPHANUM>",
      "position": 1
    },
    {
      "token": "ell",
      "start_offset": 2,
      "end_offset": 7,
      "type": "<ALPHANUM>",
      "position": 1
    },
    {
      "token": "ello",
      "start_offset": 2,
      "end_offset": 7,
```

```
      "type": "<ALPHANUM>",
      "position": 1
    },
    {
      "token": "llo",
      "start_offset": 2,
      "end_offset": 7,
      "type": "<ALPHANUM>",
      "position": 1
    },
    {
      "token": "wor",
      "start_offset": 8,
      "end_offset": 13,
      "type": "<ALPHANUM>",
      "position": 2
    },
    {
      "token": "worl",
      "start_offset": 8,
      "end_offset": 13,
      "type": "<ALPHANUM>",
      "position": 2
    },
    {
      "token": "orl",
      "start_offset": 8,
      "end_offset": 13,
      "type": "<ALPHANUM>",
      "position": 2
    },
    {
      "token": "orld",
      "start_offset": 8,
      "end_offset": 13,
      "type": "<ALPHANUM>",
      "position": 2
    },
    {
      "token": "rld",
      "start_offset": 8,
      "end_offset": 13,
      "type": "<ALPHANUM>",
```

```
      "position": 2
    }
  ]
}
```

- 自定义分词 api

Request:

```
PUT my_analyzer
{
  "settings": {
    "analysis": {
      "analyzer": {
        "my":{
          "tokenizer":"punctuation",
          "type":"custom",
          "char_filter":["emoticons"],
          "filter":["lowercase","english_stop"]
        }
      },
      "tokenizer": {
        "punctuation":{
          "type":"pattern",
          "pattern":"[.,!?]"
        }
      },
      "char_filter": {
        "emoticons":{
          "type":"mapping",
          "mappings":[
              ":)=>_happy_",
              ":(=>_sad_"
            ]
        }
      },
      "filter": {
        "english_stop":{
          "type":"stop",
          "stopwords":"_english_"
        }
      }
    }
  }
```

```
}
```

测试：

```
POST my_analyzer/_analyze
{
  "analyzer": "my",
  "text":"l'm a :) person,and you?"
}
```

```
{
  "tokens": [
    {
      "token": "l'm a _happy_ person",
      "start_offset": 0,
      "end_offset": 15,
      "type": "word",
      "position": 0
    },
    {
      "token": "and you",
      "start_offset": 16,
      "end_offset": 23,
      "type": "word",
      "position": 1
    }
  ]
}
```

- 分词使用场景

1、索引时分词：创建或更新文档时，会对相应得文档进行分词(指定字段分词)

```
PUT my_test
{
“mappings”:{
“doc”:{
“properties”:{
“title”:{
“type”:”text”,
“analyzer”:”ik_smart”
}
}
}
}
}
```

2、查询时分词：查询时会对查询语句进行分词

```
POST my_test/_search
{
“query”:{
“match”:{
“message”:{
“query”:”hello”,
“analyzer”:”standard”
}
}
}
}
```

```
PUT my_test
{
“mappings”:{
“doc”:{
“properties”:{
“title”:{
“type”:”text”,
“analyzer”:”whitespace”,
“search_analyzer”:”standard”                    #查询指定分词器
}
}
}
}
}
```

一般不需要特别指定查询时分词器，直接使用索引时分词器即可，否则会出现无法匹配得情况，如果不需要分词将字段 type 设置成 keyword，可以节省空间

2.9.3、IK 分词器

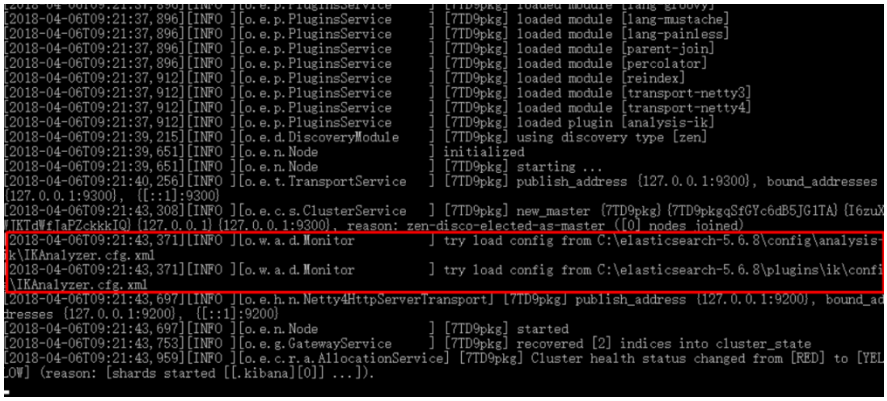
- IK 分词器的安装

  1）下载地址：https://github.com/medcl/elasticsearch-analysis-ik/releases

  下载与安装的 ES 相对应的版本

  2）解压，将解压后的 elasticsearch 文件夹拷贝到 elasticsearch-5.6.8\plugins 下，并重命名文件夹为 analysis-ik

  3）重新启动 ElasticSearch，即可加载 IK 分词器



- IK 分词器测试

IK 提供了两个分词算法 ik_smart 和 ik_max_word，其中 ik_smart 为最少切分，ik_max_word 为最细粒度划分

1）最小切分：

在浏览器地址栏输入地址

http://127.0.0.1:9200/_analyze?analyzer=ik_smart&pretty=true&text=我是程序员

输出的结果为：

```
{
"tokens" : [
{
"token" : "我",
"start_offset" : 0,
"end_offset" : 1,
"type" : "CN_CHAR",
"position" : 0
},
{
"token" : "是",
"start_offset" : 1,
"end_offset" : 2,
"type" : "CN_CHAR",
"position" : 1
},
{
"token" : "程序员",
"start_offset" : 2,
"end_offset" : 5,
"type" : "CN_WORD",
"position" : 2
}
]
```

```
}
```

2）最细切分：在浏览器地址栏输入地址

http://127.0.0.1:9200/_analyze?analyzer=ik_max_word&pretty=true&text=我是程序员

输出的结果为：

```
{
"tokens" : [
{
"token" : "我",
"start_offset" : 0,
"end_offset" : 1,
"type" : "CN_CHAR",
"position" : 0
},
{
"token" : "是",
"start_offset" : 1,
"end_offset" : 2,
"type" : "CN_CHAR",
"position" : 1
},
{
"token" : "程序员",
"start_offset" : 2,
"end_offset" : 5,
"type" : "CN_WORD",
"position" : 2
},
{
"token" : "程序",
"start_offset" : 2,
"end_offset" : 4,
"type" : "CN_WORD",
"position" : 3
},
{
"token" : "员",
"start_offset" : 4,
"end_offset" : 5,
"type" : "CN_CHAR",
"position" : 4
}
```

```
]
}
```

2.10、Mapping

- 作用：
  定义数据库中的表的结构的定义，通过 mapping 来控制索引存储数据的设置
  a. 定义 Index 下的字段名（Field Name）
  b. 定义字段的类型，比如数值型、字符串型、布尔型等
  c. 定义倒排索引相关的配置，比如 documentId、记录 position、打分等
- 获取索引 mapping
  不进行配置时，自动创建的 mapping
  请求：
  GET /atguigu/_mapping
  响应：

```
{
  "atguigu": {                                    #索引名称
    "mappings": {                                 #mapping 设置
      "student": {                                #type 名称
        "properties": {                           #字段属性
          "clazz": {
            "type": "text",                       #字段类型，字符串默认类型
            "fields": {                                 #子字段属性设置
              "keyword": {                        #分词类型（不分词）
                "type": "keyword",
                "ignore_above": 256
              }
            }
          },
          "description": {
            "type": "text",
            "fields": {
              "keyword": {
                "type": "keyword",
                "ignore_above": 256
              }
            }
          },
          "name": {
            "type": "text",
            "fields": {
              "keyword": {
```

```
                "type": "keyword",
                "ignore_above": 256
            }
        }
    }
  }
 }
}
```

- 自定义 mapping

请求：

```
PUT my_index                                    #索引名称
{
  "mappings":{
    "doc":{                                     #类型名称
      "dynamic":false,
      "properties":{
        "title":{
          "type":"text"                         #字段类型
        },
        "name":{
          "type":"keyword"
        },
        "age":{
          "type":"integer"
        }
      }
    }
  }
}
```

响应：

```
{
  "acknowledged": true,
  "shards_acknowledged": true,
  "index": "my_index"
}
```

- Dynamic Mapping

es 依靠 json 文档字段类型来实现自动识别字段类型，支持的类型

JSON 类型	es 类型

null	忽略
Boolean	boolean
浮点类型	float
整数	long
Object{}	object
Array[]	由第一个非 null 值的类型决定
String””	匹配为日期则设为 data 类型（默认开启） 匹配为数字的话设为 float 或 long 类型（默认关闭） 设为 text 类型，并附带 keyword 的子字段

- 注意：
  mapping 中的字段类型一旦设定后，禁止修改
  原因：Lucene 实现的倒排索引生成后不允许修改(提高效率)
  如果要修改字段的类型，需要从新建立索引，然后做 reindex 操作
- dynamic 设置
  a. true：允许自动新增字段（默认的配置）
  b. False：不允许自动新增字段，但是文档可以正常写入，无法对字段进行查询操作
  c. strict：文档不能写入（如果写入会报错）

  可以设置在 type 下，也可以设置在字段中（object 类型的字段中）
  例如：

```
put my_index
{
“mappings”:{
“doc”:{
“dynamic”:false,
“properties”:{
“user”:{
“properties”:{
“name”:{
“type”:”text”
},”social_networks”:{
“dynamic”:true,
“properties”:{}
}
}
}
}
}
}
}
```

- copy_to
  将该字段的值复制到目标字段，实现_all 的作用
  不会出现在_source 中，只用来搜索

```
put my_index
{
“mappings”:{
“doc”:{
“properties”:{
“frist_name”:{
“type”:”text”,
“cope_to”:”full_name”
},”last_name”:{
“type”:”text”,
“cope_to”:“full_name”
},”full_name”:{
“type”:”text”
}
}
}
}
}
```

```
put my_index/doc
{
“frist_name”:”John”,
“last_name”:”Smith”
}
```

```
GET my_index/doc
{
“query”:{
“match”:{
“full_name”:”John Smith”,
“operator”:”and”
}
}
}
```

- Index 属性
  Index 属性，控制当前字段是否索引，默认为 true，即记录索引，false 不记录，即不可以搜索，比如：手机号、身份证号等敏感信息，不希望被检索

例如：

1、创建 mapping

```
PUT my_index
{
  "mappings": {
    "doc":{
      "properties": {
        "cookie":{
          "type":"text",
          "index": false
        }
      }
    }
  }
}
```

1、创建文档

```
PUT my_index/doc/1
{
  "cookie":"123",
  "name":"home"
}
```

2、查询

```
GET my_index/_search
{
 "query": {
   "match": {
     "cookie":"123"
   }
 }
}
#报错
GET my_index/_search
{
 "query": {
   "match": {
     "name":"home"
   }
 }
}
#有结果
```

- Index_options 用于控制倒排索引记录的内容，有如下 4 中配置

docs：只记录 docid
freqs：记录 docid 和 term frequencies（词频）
position：记录 docid、term frequencies、term position
Offsets：记录 docid、term frequencies、term position、character offsets

text 类型默认配置为 position，其默认认为 docs
记录的内容越多，占用的空间越大

2.11、数据类型

实际上每个 type 中的字段是什么数据类型，由 mapping 定义。
但是如果没有设定 mapping 系统会自动，根据一条数据的格式来推断出应该的数据格式。

- true/false → boolean
- 1020 → long
- 20.1 → double
- “2018-02-01” → date
- “hello world” → text +keyword

默认只有 text 会进行分词，keyword 是不会分词的字符串。
mapping 除了自动定义，还可以手动定义，但是只能对新加的、没有数据的字段进行定义。一旦有了数据就无法再做修改了。
注意：虽然每个 Field 的数据放在不同的 type 下,但是同一个名字的 Field 在一个 index 下只能有一种 mapping 定义。

- 核心数据类型
  字符串型：text、keyword
  数值型：long、integer、short、byte、double、float、half_float、scaled_float
  日期类型：date
  布尔类型：boolean
  二进制类型：binary
  范围类型：integer_range、float_range、long_range、double_range、date_range
- 复杂数据类型
  数组类型：array
  对象类型：object
  嵌套类型：nested object
- 地理位置数据类型
  geo_point(点)、geo_shape(形状)
- 专用类型
  记录 IP 地址 ip

实现自动补全 completion

记录分词数：token_count

记录字符串 hash 值母乳 murmur3

- 多字段特性 multi-fields

允许对同一个字段采用不同的配置，比如分词，例如对人名实现拼音搜索，只需要在人名中新增一个子字段为 pinyin 即可

1、创建 mapping

```
PUT my_index1
{
  "mappings": {
    "doc":{
      "properties":{
        "username":{
          "type": "text",
          "fields": {
            "pinyin":{
              "type": "text"
            }
          }
        }
      }
    }
}
```

2、创建文档

```
PUT my_index1/doc/1
{
  "username":"haha heihei"
}
```

3、查询

```
GET my_index1/_search
{
  "query": {
    "match": {
      "username.pinyin": "haha"
    }
  }
}
```

- Dynamic Mapping

es 可以自动识别文档字段类型，从而降低用户使用成本

```
PUT /test_index/doc/1
{
  "username":"alfred",
  "age":1
}
```

```
{
  "test_index": {
    "mappings": {
      "doc": {
        "properties": {
          "age": {
            "type": "long"
          },
          "username": {
            "type": "text",
            "fields": {
              "keyword": {
                "type": "keyword",
                "ignore_above": 256
              }
            }
          }
        }
      }
    }
  }
}
```

age 自动识别为 long 类型，username 识别为 text 类型

```
PUT test_index/doc/1
{
  "username":"samualz",
  "age":14,
  "birth":"1991-12-15",
  "year":18,
  "tags":["boy","fashion"],
  "money":"100.1"
}
```

```
{
  "test_index": {
    "mappings": {
```

```
        "doc": {
          "properties": {
            "age": {
              "type": "long"
            },
            "birth": {
              "type": "date"
            },
            "money": {
              "type": "text",
              "fields": {
                "keyword": {
                  "type": "keyword",
                  "ignore_above": 256
                }
              }
            },
            "tags": {
              "type": "text",
              "fields": {
                "keyword": {
                  "type": "keyword",
                  "ignore_above": 256
                }
              }
            },
            "username": {
              "type": "text",
              "fields": {
                "keyword": {
                  "type": "keyword",
                  "ignore_above": 256
                }
              }
            },
            "year": {
              "type": "long"
            }
          }
        }
      }
    }
```

日期的自动识别可以自行配置日期格式，以满足各种需求

1、自定义日期识别格式

```
PUT my_index
{
  "mappings":{
    "doc":{
      "dynamic_date_formats": ["yyyy-MM-dd","yyyy/MM/dd"]
    }
  }
}
```

2、关闭日期自动识别

```
PUT my_index
{
  "mappings": {
    "doc": {
      "date_detection": false
    }
  }
}
```

字符串是数字时，默认不会自动识别为整形，因为字符串中出现数字时完全合理的

Numeric_datection 可以开启字符串中数字的自动识别

```
PUT my_index
{
  "mappings":{
    "doc":{
      "numeric_datection": true
    }
  }
}
```

2.12、文档操作

CRUD

- 创建文档

1、索引一个文档

文档通过 index API 被索引——使数据可以被存储和搜索。但是首先我们需要决定文档所在。正如我们讨论的，文档通过其_index、_type、_id 唯一确定。们可以自己提供一个_id，或者也使用 index API 为我们生成一个。

PUT {index}/{type}/{id}
{
“”:””
}

2、使用自己的 ID

如果你的文档有自然的标识符（例如 user_account 字段或者其他值表示文档），你就可以提供自己的_id，使用这种形式的 index API：

PUT /{index}/{type}/{id}
{
"field": "value",
...
}

例如我们的索引叫做“website”，类型叫做“blog”，我们选择的 ID 是“123”，那么这个索引请求就像这样：

PUT /website/blog/123
{
"title": "My first blog entry",
"text": "Just trying this out...",
"date": "2014/01/01"
}

Elasticsearch 的响应：

{
"_index": "website",
"_type": "blog",
"_id": "123",
"_version": 1,
"created": true
}

响应指出请求的索引已经被成功创建，这个索引中包含_index、_type 和_id 元数据，以及一个新元素：_version。

Elasticsearch 中每个文档都有版本号，每当文档变化（包括删除）都会使_version 增加。_version 确保你程序的一部分不会覆盖掉另一部分所做的更改。

3、自增 ID

如果我们的数据没有自然 ID，我们可以让 Elasticsearch 自动为我们生成。请求结构发生了变化：PUT 方法——“在这个 URL 中存储文档”变成了 POST 方法——"在这个类型下存储文档"。（译者注：原来是把文档存储到某个 ID 对应的空间，现在是把这个文档添加到某个_type 下）。

URL 现在只包含_index 和_type 两个字段：

POST /website/blog/

```
{
  "title": "My second blog entry",
  "text":  "Still trying this out...",
  "date":  "2014/01/01"
}
```

响应内容与刚才类似，只有_id 字段变成了自动生成的值：

```
{
    "_index":      "website",
    "_type":       "blog",
    "_id":         "wM0OSFhDQXGZAWDf0-drSA",
    "_version":    1,
    "created":     true
}
```

自动生成的 ID 有 22 个字符长，URL-safe, Base64-encoded string universally unique identifiers, 或者叫 UUIDs。

- 获取文档

1、检索文档

想要从 Elasticsearch 中获取文档，我们使用同样的_index、_type、_id，但是 HTTP 方法改为 GET：

```
GET /website/blog/123?pretty
```

响应包含了现在熟悉的元数据节点，增加了_source 字段，它包含了在创建索引时我们发送给 Elasticsearch 的原始文档。

```
{
  "_index" :     "website",
  "_type" :      "blog",
  "_id" :        "123",
  "_version" : 1,
  "found" :      true,
  "_source" :    {
      "title": "My first blog entry",
      "text":  "Just trying this out...",
      "date":  "2014/01/01"
  }
}
```

2、pretty

在任意的查询字符串中增加 pretty 参数，类似于上面的例子。会让 Elasticsearch 美化输出(pretty-print)JSON 响应以便更加容易阅读。_source 字段不会被美化，它的样子与我们输入的一致。

GET 请求返回的响应内容包括{"found": true}。这意味着文档已经找到。如果我们请求一个不存在的文档，依旧会得到一个 JSON，不过 found 值变成了 false。

此外，HTTP 响应状态码也会变成'404 Not Found'代替'200 OK'。我们可以在 curl 后加-i 参数得到响应头：

curl -i -XGET http://localhost:9200/website/blog/124?pretty

现在响应类似于这样：

```
HTTP/1.1 404 Not Found
Content-Type: application/json; charset=UTF-8
Content-Length: 83

{
   "_index" : "website",
   "_type" :   "blog",
   "_id" :      "124",
   "found" :   false
}
```

3、检索文档的一部分

通常，GET 请求将返回文档的全部，存储在_source 参数中。但是可能你感兴趣的字段只是 title。请求个别字段可以使用_source 参数。多个字段可以使用逗号分隔：

GET /website/blog/123?_source=title,text

_source 字段现在只包含我们请求的字段，而且过滤了 date 字段：

```
{
   "_index" :     "website",
   "_type" :      "blog",
   "_id" :        "123",
   "_version" : 1,
   "exists" :     true,
   "_source" : {
         "title": "My first blog entry" ,
         "text":   "Just trying this out..."
   }
}
```

或者你只想得到_source 字段而不要其他的元数据，你可以这样请求：

GET /website/blog/123/_source

它仅仅返回:

```
{
```

```
    "title": "My first blog entry",
    "text":  "Just trying this out...",
    "date":  "2014/01/01"
}
```

- 更新

```
POST /website/blog/123
{
  "title": "My first blog entry",
  "text":  "I am starting to get the hang of this...",
  "date":  "2014/01/02"
}
```

在响应中，我们可以看到 Elasticsearch 把_version 增加了。

```
{
  "_index" :    "website",
  "_type" :     "blog",
  "_id" :       "123",
  "_version" : 2,
  "created":    false <1>
}
```

- 删除文档

删除文档的语法模式与之前基本一致，只不过要使用 DELETE 方法：

DELETE /website/blog/123

- 局部更新

```
POST /website/blog/1/_update
{
   "doc" : {
      "tags" : [ "testing" ],
      "views": 0
   }
}
```

如果请求成功，我们将看到类似 index 请求的响应结果：

```
{
   "_index" :    "website",
   "_id" :       "1",
   "_type" :     "blog",
   "_version" : 3
}
```

检索文档文档显示被更新的_source 字段：

```
{
    "_index":       "website",
    "_type":        "blog",
    "_id":          "1",
    "_version":    3,
    "found":        true,
    "_source": {
        "title":    "My first blog entry",
        "text":     "Starting to get the hang of this...",
        "tags": [ "testing" ], <1>
        "views":    0 <1>
    }
}
```

- 批量插入

每个 json 之间不能有换行\n

```
POST test_search_index/doc/_bulk
{
  "index":{
    "_id":1
  }
}
{
  "username":"alfred way",
  "job":"java engineer",
  "age":18,
  "birth":"1991-12-15",
  "isMarried":false
}
{
  "index":{
    "_id":2
  }
}
{
  "username":"alfred",
  "job":"java senior engineer and java specialist",
  "age":28,
  "birth":"1980-05-07",
  "isMarried":true
}
```

```
{
  "index":{
    "_id":3
  }
}
{
  "username":"lee",
  "job":"java and ruby engineer",
  "age":22,
  "birth":"1985-08-07",
  "isMarried":false
}
{
  "index":{
    "_id":4
  }
}
{
  "username":"lee junior way",
  "job":"ruby engineer",
  "age":23,
  "birth":"1986-08-07",
  "isMarried":false
}
```

- 检索多个文档

像 Elasticsearch 一样，检索多个文档依旧非常快。合并多个请求可以避免每个请求单独的网络开销。如果你需要从 Elasticsearch 中检索多个文档，相对于一个一个的检索，更快的方式是在一个请求中使用 multi-get 或者 mget API。

mget API 参数是一个 docs 数组，数组的每个节点定义一个文档的_index、_type、_id 元数据。如果你只想检索一个或几个确定的字段，也可以定义一个_source 参数：

```
POST /_mget
{
    "docs" : [
        {
            "_index" : "website",
            "_type" :   "blog",
            "_id" :      2
        },
        {
            "_index" : "website",
            "_type" :   "pageviews",
            "_id" :      1,
```

```
            "_source": "views"
        }
    ]
}
```

响应体也包含一个 docs 数组，每个文档还包含一个响应，它们按照请求定义的顺序排列。每个这样的响应与单独使用 get request 响应体相同：

```
{
    "docs" : [
        {
            "_index" :      "website",
            "_id" :         "2",
            "_type" :       "blog",
            "found" :       true,
            "_source" : {
                "text" :    "This is a piece of cake...",
                "title" : "My first external blog entry"
            },
            "_version" : 10
        },
        {
            "_index" :      "website",
            "_id" :         "1",
            "_type" :       "pageviews",
            "found" :       true,
            "_version" : 2,
            "_source" : {
                "views" : 2
            }
        }
    ]
}
```

如果你想检索的文档在同一个_index 中（甚至在同一个_type 中），你就可以在 URL 中定义一个默认的/_index 或者/_index/_type。

你可以通过简单的 ids 数组来代替完整的 docs 数组：

```
POST /website/blog/_mget
{
    "ids" : [ "2", "1" ]
}
```

注意到我们请求的第二个文档并不存在。我们定义了类型为 blog，但是 ID 为 1 的文档类型为 pageviews。这个不存在的文档会在响应体中被告知。

```
{
```

```
  "docs" : [
    {
      "_index" :    "website",
      "_type" :     "blog",
      "_id" :       "2",
      "_version" : 10,
      "found" :     true,
      "_source" : {
        "title":    "My first external blog entry",
        "text":     "This is a piece of cake..."
      }
    },
    {
      "_index" :    "website",
      "_type" :     "blog",
      "_id" :       "1",
      "found" :     false   <1>
    }
  ]
}
```

2.13、Search API(URI)

```
GET /_search                              #查询所有索引文档
GET /my_index/_search                     #查询指定索引文档
GET /my_index1,my_index2/_search          #多索引查询
GET /my_*/_search
2019-03-xxx
2019-04-vvv
2019-05-xxx
```

- URI 查询方式（查询有限制，很多配置不能实现）

```
GET /my_index/_search?q=user:alfred       #指定字段查询

GET /my_index/_search?q=keyword&df=user&sort=age:asc&from=4&size=10&timeout=1s
```

```
q：指定查询的语句，例如 q=aa 或 q=user:aa
df:q 中不指定字段默认查询的字段，如果不指定，es 会查询所有字段
Sort：排序，asc 升序，desc 降序
timeout：指定超时时间，默认不超时
from，size：用于分页
```

- term 与 phrase
  term 相当于单词查询，phrase 相当于词语查询
  term：Alfred way 等效于 alfred or way
  phrase：”Alfred way” 词语查询，要求先后顺序
- 泛查询
  Alfred 等效于在所有字段去匹配该 term(不指定字段查询)
- 指定字段
  name:alfred
- Group 分组设定（），使用括号指定匹配的规则
  （quick OR brown）AND fox：通过括号指定匹配的优先级
  status:(active OR pending) title:(full text search)：把关键词当成一个整体

- 查询案例及详解

1、批量创建文档

```
POST test_search_index/doc/_bulk
{
  "index":{
    "_id":1
  }
}
{
  "username":"alfred way",
  "job":"java engineer",
  "age":18,
  "birth":"1991-12-15",
  "isMarried":false
}
{
  "index":{
    "_id":2
  }
}
{
  "username":"alfred",
  "job":"java senior engineer and java specialist",
  "age":28,
  "birth":"1980-05-07",
  "isMarried":true
}
{
  "index":{
    "_id":3
  }
}
```

```
{
  "username":"lee",
  "job":"java and ruby engineer",
  "age":22,
  "birth":"1985-08-07",
  "isMarried":false
}
{
  "index":{
    "_id":4
  }
}
{
  "username":"lee junior way",
  "job":"ruby engineer",
  "age":23,
  "birth":"1986-08-07",
  "isMarried":false
}
```

2、泛查询

```
GET test_search_index/_search?q=alfred
```

3、查询语句执行计划查看

```
GET test_search_index/_search?q=alfred
{
  "profile":true
}
```

4、term 查询

```
GET test_search_index/_search?q=username:alfred way          #alfred OR way
```

5、phrase 查询

```
GET test_search_index/_search?q=username:"alfred way"
```

6、group 查询

```
GET test_search_index/_search?q=username:(alfred OR way)
```

7、布尔操作符

（1）AND(&&),OR(||),NOT(!)

例如：name:(tom NOT lee)

#表示 name 字段中可以包含 tom 但一定不包含 lee

（2）+、-分别对应 must 和 must_not

例如：name:(tom +lee -alfred)

#表示 name 字段中，一定包含 lee，一定不包含 alfred，可以包含 tom

注意：+在 url 中会被解析成空格，要使用 encode 后的结果才可以，为%2B

```
GET test_search_index/_search?q=username:(alfred %2Bway)
```

- 范围查询，支持数值和日期

  1、区间：闭区间：[]，开区间:{}

  age:[1 TO 10] #1<=age<=10

  age:[1 TO 10} #1<=age<10

  age:[1 TO] #1<=age

  age:[* TO 10] #age<=10

  2、算术符号写法

  age:>=1

  age:(>=1&&<=10)或者 age:(+>=1 +<=10)

- 通配符查询

  ?:1 个字符

  *:0 或多个字符

  例如：name:t?m

  name:tom*

  name:t*m

  注意：通配符匹配执行效率低，且占用较多内存，不建议使用，如无特殊要求，不要讲?/*放在最前面

- 正则表达式

  name:/[mb]oat/

- 模糊匹配 fuzzy query

  name:roam~1 [0,1,2]

  匹配与 roam 差 1 个 character 的词，比如 foam、roams 等

- 近似度查询 proximity search

  “fox quick”~5

  以 term 为单位进行差异比较，比如”quick fox” “quick brown fox”

2.14、Search API(Request Body Search)

1 查看 es 中有哪些索引

```
GET /_cat/indices?v
```

es 中会默认存在一个名为.kibana 的索引

表头的含义

health	green(集群完整) yellow(单点正常、集群不完整) red(单点不正常)
status	是否能使用
index	索引名
uuid	索引统一编号
pri	主节点几个
rep	从节点几个
docs.count	文档数
docs.deleted	文档被删了多少
store.size	整体占空间大小
pri.store.size	主节点占

cluster	整个 elasticsearch 默认就是集群状态，整个集群是一份完整、互备的数据。
node	集群中的一个节点，一般只一个进程就是一个 node
shard	分片，即使是一个节点中的数据也会通过 hash 算法，分成多个片存放，默认是 5 片。
index	相当于 rdbms 的 database, 对于用户来说是一个逻辑数据库，虽然物理上会被分多个 shard 存放，也可能存放在多个 node 中。
type	类似于 rdbms 的 table，但是与其说像 table，其实更像面向对象中的 class , 同一 Json 的格式的数据集合。
Document(json)	类似于 rdbms 的 row、面向对象里的 object
field	相当于字段、属性

2 增加一个索引(库)

```
PUT /movie_index
```

3 删除一个索引

ES 是不删除也不修改任何数据

```
DELETE /movie_index
```

4 新增文档

1、 格式 PUT /index/type/id

```
PUT /movie_index/movie/1
{ "id":1,
  "name":"operation red sea",
  "doubanScore":8.5,
  "actorList":[
    {"id":1,"name":"zhang yi"},
    {"id":2,"name":"hai qing"},
    {"id":3,"name":"zhang han yu"}
  ]
}
PUT /movie_index/movie/2
{
  "id":2,
  "name":"operation meigong river",
  "doubanScore":8.0,
  "actorList":[
    {"id":3,"name":"zhang han yu"}
    ]
}

PUT /movie_index/movie/3
{
  "id":3,
  "name":"incident red sea",
  "doubanScore":5.0,
  "actorList":[
    {"id":4,"name":"zhang chen"}
  ]
}
```

如果之前没建过 index 或者 type，es 会自动创建。

5 直接用 id 查找

```
GET movie_index/movie/1
```

6 修改—整体替换

和新增没有区别

```
PUT /movie_index/movie/3
{
    "id":"3",
    "name":"incident red sea",
    "doubanScore":"5.0",
    "actorList":[
     {"id":"1","name":"zhang chen"}
    ]
}
```

7 修改—某个字段

```
POST movie_index/movie/3/_update
{
   "doc": {
      "doubanScore":"7.0"
   }
}
```

8 删除一个 document

```
DELETE movie_index/movie/3
```

9 搜索 type 全部数据

```
GET movie_index/movie/_search
```

结果

```
{
   "took": 2,      //耗费时间 毫秒
   "timed_out": false, //是否超时
   "_shards": {
      "total": 5,     //发送给全部 5 个分片
      "successful": 5,
      "skipped": 0,
      "failed": 0
   },
   "hits": {
      "total": 3,    //命中 3 条数据
      "max_score": 1,     //最大评分
```

```
    "hits": [    // 结果
      {
        "_index": "movie_index",
        "_type": "movie",
        "_id": 2,
        "_score": 1,
        "_source": {
          "id": "2",
          "name": "operation meigong river",
          "doubanScore": 8.0,
          "actorList": [
            {
              "id": "1",
              "name": "zhang han yu"
            }
          ]
        }
          ◦ ◦ ◦ ◦ ◦ ◦ ◦ ◦
          ◦ ◦ ◦ ◦ ◦ ◦ ◦ ◦
      }
```

10 按条件查询(全部)

```
GET movie_index/movie/_search
{
  "query":{
    "match_all": {}
  }
}
```

11 按分词查询

```
GET movie_index/movie/_search
{
  "query":{
    "match": {"name":"red"}
  }
}
```

注意结果的评分

12 按分词子属性查询

```
GET movie_index/movie/_search
{
  "query":{
    "match": {"actorList.name":"zhang"}
  }
```

```
}
```

13 match phrase

```
GET movie_index/movie/_search
{
      "query":{
        "match_phrase": {"name":"operation red"}
      }
}
```

按短语查询，不再利用分词技术，直接用短语在原始数据中匹配

14 fuzzy 查询

```
GET movie_index/movie/_search
{
      "query":{
        "fuzzy": {"name":"rad"}
      }
}
```

校正匹配分词，当一个单词都无法准确匹配，es 通过一种算法对非常接近的单词也给与一定的评分，能够查询出来，但是消耗更多的性能。

15 过滤--查询后过滤

```
GET movie_index/movie/_search
{
      "query":{
        "match": {"name":"red"}
      },
      "post_filter":{
        "term": {
          "actorList.id": 3
        }
      }
}
```

16 过滤--查询前过滤（推荐）

```
GET movie_index/movie/_search
{
```

```
      "query":{
            "bool":{
              "filter":[ {"term": {    "actorList.id": "1"    }},
                                {"term": {    "actorList.id": "3"    }}
                ],
                "must":{"match":{"name":"red"}}
             }
      }
}
```

17 过滤--按范围过滤

```
GET movie_index/movie/_search
{
    "query": {
      "bool": {
        "filter": {
          "range": {
              "doubanScore": {"gte": 8}
          }
        }
      }
    }
}
```

关于范围操作符：

gt	大于
lt	小于
gte	大于等于
lte	小于等于

18 排序

```
GET movie_index/movie/_search
{
   "query":{
      "match": {"name":"red sea"}
   }
   , "sort": [
      {
         "doubanScore": {
            "order": "desc"
         }
      }
   ]
}
```

19 分页查询

```
GET movie_index/movie/_search
{
   "query": { "match_all": {} },
   "from": 1,
   "size": 1
}
```

20 指定查询的字段

```
GET movie_index/movie/_search
{
   "query": { "match_all": {} },
   "_source": ["name", "doubanScore"]
}
```

21 高亮

```
GET movie_index/movie/_search
{
      "query":{
         "match": {"name":"red sea"}
      },
      "highlight": {
         "fields": {"name":{} }
      }

}
```

22 java 客户端

```
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-elasticsearch</artifactId>
        </dependency>

        <!-- https://mvnrepository.com/artifact/io.searchbox/jest -->
        <dependency>
            <groupId>io.searchbox</groupId>
            <artifactId>jest</artifactId>
```

```
            <version>5.3.3</version>
        </dependency>


        <!-- https://mvnrepository.com/artifact/net.java.dev.jna/jna -->
        <dependency>
            <groupId>net.java.dev.jna</groupId>
            <artifactId>jna</artifactId>
            <version>4.5.1</version>
        </dependency>
```

23 复杂查询

先过滤，后查询

```
"query":{
        "bool":{
        "filter":[ {"term": {   "actorList.id": "1"   }},{"term": {   "actorList.id": "3"   }}],
        "must":[{"match":{"name":"red"}}]
        }
}
"query": {
         "bool": {
         "filter": [{"terms":{ "actorList.id": [1,3]}}] ,
         "must": [{"match": {"name": "red"}}]
         }
}
```

创建 mapping

mappings movie properties

```
PUT gmall
{
  "mappings": {
    "SkuInfo":{
      "properties": {
        "id":{
          "type": "keyword"
          , "index": false
        },
        "price":{
          "type": "double"
        },
         "skuName":{
          "type": "text",
          "analyzer": "ik_max_word"
```

```
            },
            "skuDesc":{
               "type": "text",
               "analyzer": "ik_smart"
            },
            "catalog3Id":{
               "type": "keyword"
            },
            "skuDefaultImg":{
               "type": "keyword",
               "index": false
            },
            "skuAttrValueList":{
               "properties": {
                  "valueId":{
                     "type":"keyword"
                  }
               }
            }
         }
      }
   }
}
```

查询

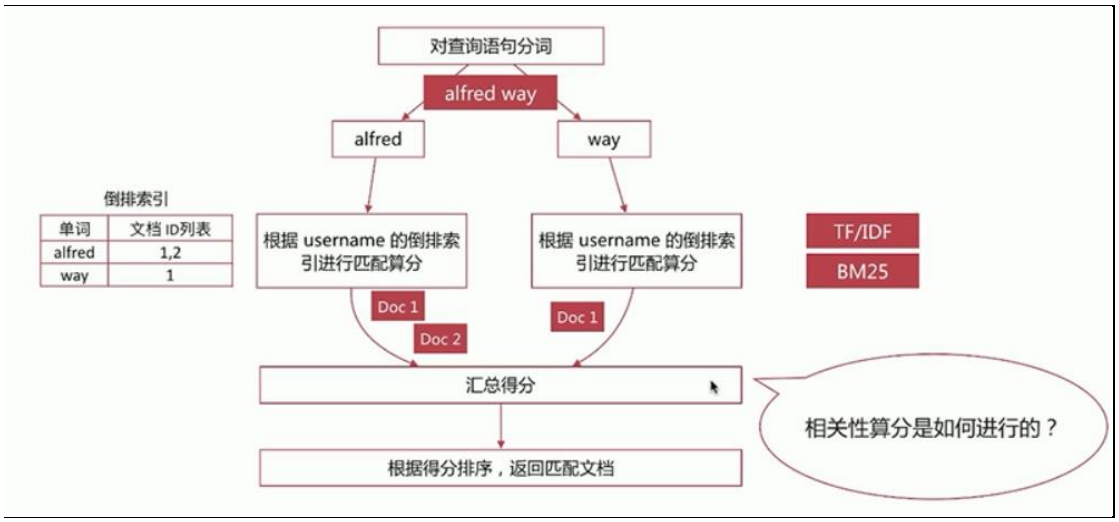
```
GET gmall/SkuInfo/_search
{
"query": {
"bool": {
        "filter": [{"terms":{ "skuAttrValueList.valueId":
["46","45"]}},{"term":{"catalog3Id":"61"}}],
        "must": { "match": { "skuName": "小米" } }
         }
},
"highlight": {
     "fields": {"skuName":{}}
},
"sort":{
   "hotScore":{"order":"desc"}},
   "aggs": { "groupby_attr": {"terms": {"field": "skuAttrValueList.valueId" }}
}

}
```

```
Index index = new Index.Builder(null).index("").type("").id(null).build();
```

如果聚合查询出现以下报错，则通过下面命令解决

Err: fielddata is disabled on text fields by default. set fielddata=true on [aaaa]

2.15、相关性算分

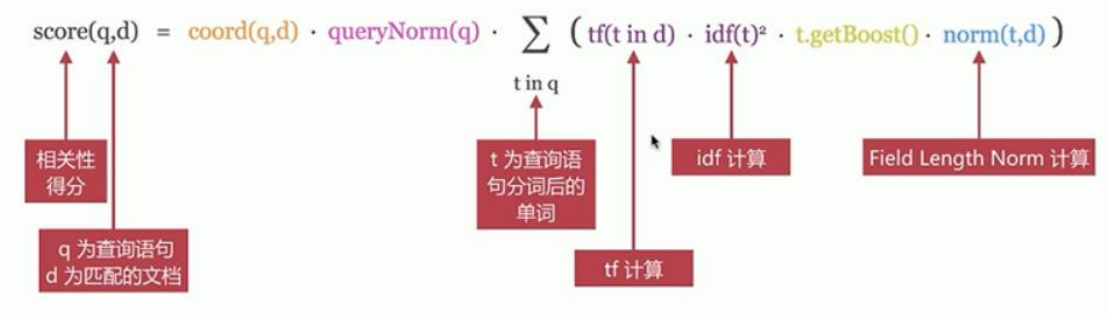


相关性算分：指文档与查询语句间的相关度，通过倒排索引可以获取与查询语句相匹配的文档列表

如何将最符合用户查询需求的文档放到前列呢？
本质问题是一个排序的问题，排序的依据是相关性算分，确定倒排索引哪个文档排在前面

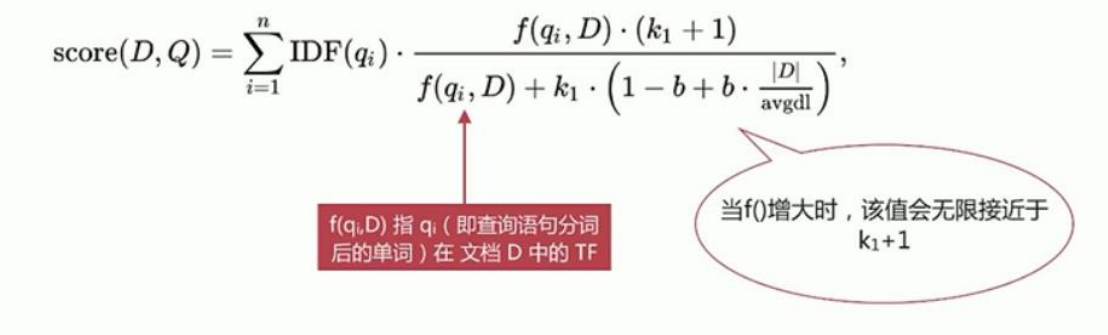
影响相关度算分的参数：
1、TF(Term Frequency)：词频，即单词在文档中出现的次数，词频越高，相关度越高
2、Document Frequency(DF)：文档词频，即单词出现的文档数
3、IDF(Inverse Document Frequency)：逆向文档词频，与文档词频相反，即 1/DF。即单词出现的文档数越少，相关度越高（如果一个单词在文档集出现越少，算为越重要单词）
4、Field-length Norm：文档越短，相关度越高

- TF/IDE 模型



- BM25 模型（5.X 之后的默认模型）

对之前算分进行优化



BM25 相比 TF/IDF 的一大优化是降低了 tf 在过大时的权重，避免词频对查询影响过大

查看算分

```
GET test_search_index/_search
{
```

```
  "explain": true,
  "query":{
    "match": {
      "username": {
        "query":"alfred way",
        "operator":"and"
      }
    }
  }
}
```

三、Elasticsearch 集群

3.1、ElasticSerach 集群安装

- 修改配置文件 elasticserach.yml
  vim /elasticsearch.yml

```
cluster.name: aubin-cluster        #必须相同
# 集群名称（不能重复）
node.name: els1（必须不同）
# 节点名称，仅仅是描述名称，用于在日志中区分（自定义）
#指定了该节点可能成为 master 节点，还可以是数据节点
node.master: true
node.data: true
path.data: /opt/data
# 数据的默认存放路径（自定义）
path.logs: /opt/logs
# 日志的默认存放路径
network.host: 192.168.0.1
# 当前节点的 IP 地址
http.port: 9200
# 对外提供服务的端口
transport.tcp.port: 9300
#9300 为集群服务的端口
discovery.zen.ping.unicast.hosts: ["172.18.68.11", "172.18.68.12","172.18.68.13"]
# 集群个节点 IP 地址，也可以使用域名，需要各节点能够解析
discovery.zen.minimum_master_nodes: 2
# 为了避免脑裂，集群节点数最少为 半数+1
```

注意：清空 data 和 logs 数据

192.168.14.12:9200/_cat/nodes?v

3.2、安装 head 插件

- 下载 head 插件

  `wget https://github.com/mobz/elasticsearch-head/archive/elasticsearch-head-master.zip`

  也可以用 git 下载，前提 yum install git

  unzip elasticsearch-head-master.zip
- 安装 node.js

  `wget https://npm.taobao.org/mirrors/node/latest-v4.x/node-v4.4.7-linux-x64.tar.gz`

  tar -zxvf node-v9.9.0-linux-x64.tar.gz
- 添加 node.js 到环境变量

```
export JAVA_HOME=/home/elk1/jdk1.8/jdk1.8.0_171
export JRE_HOME=$JAVA_HOME/jre
export CLASSPATH=.:$JAVA_HOME/LIB:$JRE_HOME/LIB:$CLASSPATH
export NODE_HOME=/home/elk1/elasticserach/head/node-v9.9.0-linux-x64
export PATH=$JAVA_HOME/bin:$JRE_HOME/bin:$NODE_HOME/bin:$PATH
```

  source /etc/profile
- 测试

  node -v

  npm -v
- 安装 grunt（grunt 是一个很方便的构建工具，可以进行打包压缩、测试、执行等等的工作）

  进入到 elasticsearch-head-master

  npm install -g grunt-cli

  npm install

  (npm install -g cnpm --registry=https://registry.npm.taobao.org)

- 修改 Elasticsearch 配置文件

  编辑 elasticsearch-6.3.1/config/elasticsearch.yml,加入以下内容：

  http.cors.enabled: true

  http.cors.allow-origin: "*"

- 修改 Gruntfile.js（注意’，’）

打开 elasticsearch-head-master/Gruntfile.js，找到下面 connect 属性，新增 hostname:’*’:

connect: {

  server: {

```
            options: {
                hostname: '*',
                port: 9100,
                base: '.',
                keepalive: true
            }
        }
    }
```

- 启动 elasticsearch-head
  进入 elasticsearch-head 目录，执行命令：grunt server
- 后台启动 elasticsearch-head
  nohup grunt server &exit
- 关闭 head 插件
  ps -aux|grep head
  kill 进程号

3.3、集群简介

一个节点(node)就是一个 Elasticsearch 实例，而一个集群(cluster)由一个或多个节点组成，它们具有相同的 cluster.name，它们协同工作，分享数据和负载。

当加入新的节点或者删除一个节点时，集群就会感知到并平衡数据（同步）。

3.3.1、集群节点

1、集群中一个节点会被选举为主节点(master)

2、主节点临时管理集群级别的一些变更，例如新建或删除索引、增加或移除节点等。

3、主节点不参与文档级别的变更或搜索，这意味着在流量增长的时候，该主节点不会成为集群的瓶颈。

4、任何节点都可以成为主节点。

5、用户，我们能够与集群中的任何节点通信，包括主节点。

6、每一个节点都知道文档存在于哪个节点上，它们可以转发请求到相应的节点上。

7、我们访问的节点负责收集各节点返回的数据，最后一起返回给客户端。这一切都由 Elasticsearch 处理。

3.3.2、集群健康

在 Elasticsearch 集群中可以监控统计很多信息，但是只有一个是最重要的：集群健康(cluster health)。集群健康有三种状态：green、yellow 或 red。

在一个没有索引的空集群中运行如上查询，将返回这些信息：

```
GET /_cluster/health
{
    "cluster_name":                 "elasticsearch",
    "status":                       "green",
    "timed_out":                    false,
    "number_of_nodes":              1,
    "number_of_data_nodes":         1,
    "active_primary_shards": 0,
    "active_shards":                0,
    "relocating_shards":            0,
    "initializing_shards":          0,
    "unassigned_shards":            0
}
```

status 字段提供一个综合的指标来表示集群的的服务状况。三种颜色各自的含义：

颜色	意义
green	所有主要分片和复制分片都可用
yellow	所有主要分片可用，但不是所有复制分片都可用
red	不是所有的主要分片都可用

3.3.3、集群分片

索引只是一个用来指向一个或多个分片(shards)的“逻辑命名空间(logical namespace)”.

分片(shard)是一个最小级别“工作单元(worker unit)”,它只是保存了索引中所有数据的一部分，是一个 Lucene 实例，并且它本身就是一个完整的搜索引擎。我们的文档存储在分片中，并且在分片中被索引，但是我们的应用程序不会直接与它们通信，取而代之的是，直接与索引通信。

分片是 Elasticsearch 在集群中分发数据的关键。把分片想象成数据的容器。文档存储在分片中，然后分片分配到你集群中的节点上。当你的集群扩容或缩小，Elasticsearch 将会自动在你的节点间迁移分片，以使集群保持平衡。

1、主分片

索引中的每个文档属于一个单独的主分片，所以主分片的数量决定了索引最多能存储多少数据。

理论上主分片能存储的数据大小是没有限制的，限制取决于你实际的使用情况。分片的最大容量完全取决于你的使用状况：硬件存储的大小、文档的大小和复杂度、如何索引和查询你的文档，以及你期望的响应时间。

2、副分片

复制分片只是主分片的一个副本，它可以防止硬件故障导致的数据丢失，同时可以

提供读请求，比如搜索或者从别的 shard 取回文档。

当索引创建完成的时候，主分片的数量就固定了，但是复制分片的数量可以随时调整。

创建分片：

```
PUT /blogs
{
    "settings" : {
        "number_of_shards" : 3,
        "number_of_replicas" : 1
    }
}
```

增加副分片：

```
PUT /blogs/_settings
{
    "number_of_replicas" : 2
}
```

主分片设置后不能进行修改，只能修改副本分片

集群的健康状态 yellow 表示所有的主分片(primary shards)启动并且正常运行了——集群已经可以正常处理任何请求——但是复制分片(replica shards)还没有全部可用。事实上所有的三个复制分片现在都是 unassigned 状态——它们还未被分配给节点。在同一个节点上保存相同的数据副本是没有必要的，如果这个节点故障了，那所有的数据副本也会丢失。

3.3.4、故障转移

在单一节点上运行意味着有单点故障的风险——没有数据备份。幸运的是，要防止单点故障，我们唯一需要做的就是启动另一个节点。

第二个节点已经加入集群，三个复制分片(replica shards)也已经被分配了——分别对应三个主分片，这意味着在丢失任意一个节点的情况下依旧可以保证数据的完整性。

文档的索引将首先被存储在主分片中，然后并发复制到对应的复制节点上。这可以确保我们的数据在主节点和复制节点上都可以被检索。

3.4、集群操作原理

3.4.1、路由

当你索引一个文档，它被存储在单独一个主分片上。Elasticsearch 是如何知道文档属于

哪个分片的呢？当你创建一个新文档，它是如何知道是应该存储在分片 1 还是分片 2 上的呢？

进程不能是随机的，因为我们将来要检索文档。

算法决定：

shard = hash(routing) % number_of_primary_shards

routing 值是一个任意字符串，它默认是_id 但也可以自定义。

为什么主分片的数量只能在创建索引时定义且不能修改？

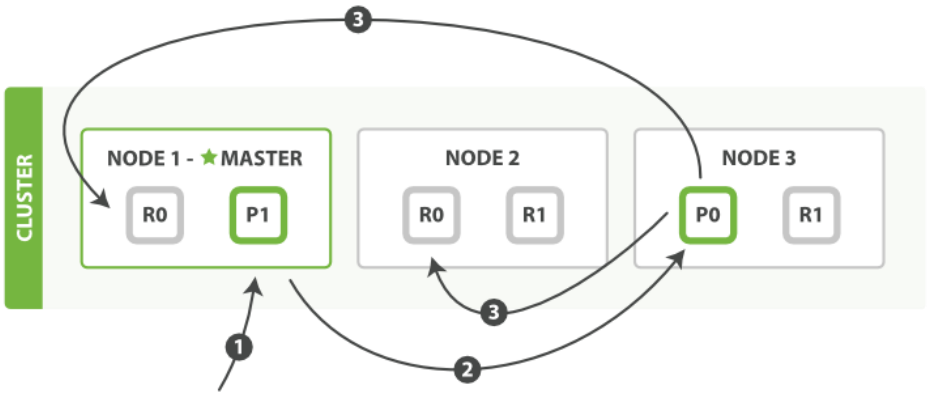
如果主分片的数量在未来改变了，所有先前的路由值就失效了，文档也就永远找不到了。

所有的文档 API（get、index、delete、bulk、update、mget）都接收一个 routing 参数，它用来自定义文档到分片的映射。自定义路由值可以确保所有相关文档——例如属于同一个人的文档——被保存在同一分片上。

3.4.2、操作数据节点工作流程

每个节点都有能力处理任意请求。每个节点都知道任意文档所在的节点，所以也可以将请求转发到需要的节点。

新建、索引和删除请求都是写(write)操作，它们必须在主分片上成功完成才能复制到相关的复制分片上。



1. 客户端给 Node 1 发送新建、索引或删除请求。
2. 节点使用文档的_id 确定文档属于分片 0。它转发请求到 Node 3，分片 0 位于这个节点上。
3. Node 3 在主分片上执行请求，如果成功，它转发请求到相应的位于 Node 1 和 Node 2 的复制节点上。当所有的复制节点报告成功，Node 3 报告成功到请求的节点，请求的节点再报告给客户端。

replication

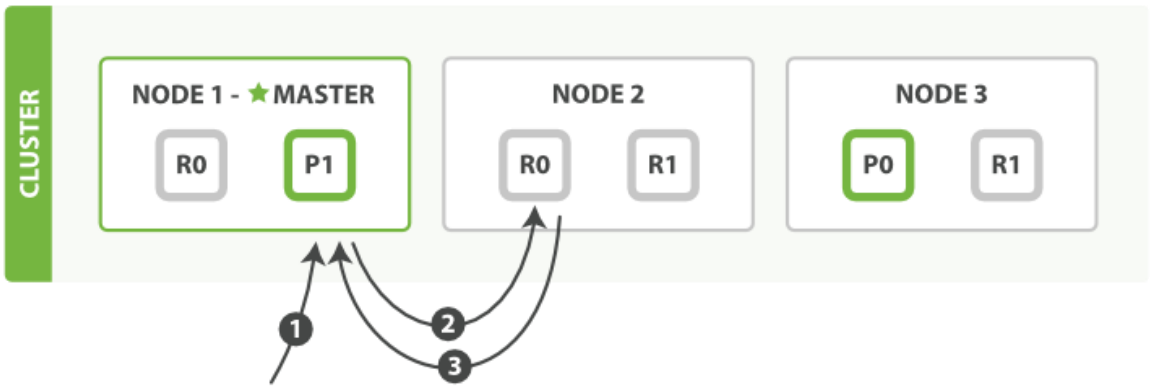
复制默认的值是 sync。这将导致主分片得到复制分片的成功响应后才返回。

如果你设置 replication 为 async，请求在主分片上被执行后就会返回给客户端。它依旧会转发请求给复制节点，但你将不知道复制节点成功与否。

上面的这个选项不建议使用。默认的 sync 复制允许 Elasticsearch 强制反馈传输。async 复制可能会因为在不等待其它分片就绪的情况下发送过多的请求而使 Elasticsearch 过载。

3.4.3、检索流程

文档能够从主分片或任意一个复制分片被检索。



1. 客户端给 Node 1 发送 get 请求。
2. 节点使用文档的_id 确定文档属于分片 0。分片 0 对应的复制分片在三个节点上都有。此时，它转发请求到 Node 2。
3. Node 2 返回文档(document)给 Node 1 然后返回给客户端。

对于读请求，为了平衡负载，请求节点会为每个请求选择不同的分片——它会循环所有分片副本。

可能的情况是，一个被索引的文档已经存在于主分片上却还没来得及同步到复制分片上。这时复制分片会报告文档未找到，主分片会成功返回文档。一旦索引请求成功返回给用户，文档则在主分片和复制分片都是可用的。

四 安装 kibana

拷贝 kibana-5.6.4-linux-x86_64.tar 到/opt 下

解压缩

进入 kibana 主目录的 config 目录下

```
drwxr-xr-x.   2 atguigu atguigu     76 3月   16 23:08 bin
drwxrwxr-x.   2 atguigu atguigu     24 3月   16 23:07 config
drwxrwxr-x.   2 atguigu atguigu     18 3月   16 22:57 data
-rw-rw-r--.   1 atguigu atguigu    562 11月   1 03:04 LICENSE.txt
drwxrwxr-x.   6 atguigu atguigu    108 11月   1 03:04 node
drwxrwxr-x. 618 atguigu atguigu  20480 11月   1 03:04 node_modules
-rw-rw-r--.   1 atguigu atguigu 798420 11月   1 03:04 NOTICE.txt
drwxrwxr-x.   3 atguigu atguigu     45 11月   1 03:04 optimize
-rw-rw-r--.   1 atguigu atguigu    721 11月   1 03:04 package.json
drwxrwxr-x.   2 atguigu atguigu      6 11月   1 03:04 plugins
-rw-rw-r--.   1 atguigu atguigu   4899 11月   1 03:04 README.txt
drwxr-xr-x.  13 atguigu atguigu    165 11月   1 03:04 src
drwxrwxr-x.   5 atguigu atguigu     52 11月   1 03:04 ui_framework
drwxr-xr-x.   2 atguigu atguigu   4096 11月   1 03:04 webpackShims
[root@jack kibana-5.6.4-linux-x86_64]# cd config
[root@jack config]#
```

vim kibana.yml

```
# Kibana is served by a back end server. This setting specifies th
#server.port: 5601

# Specifies the address to which the Kibana server will bind. IP a
# The default is 'localhost', which usually means remote machines
# To allow connections from remote users, set this parameter to a
server.host: "0.0.0.0"

# Enables you to specify a path to mount Kibana at if you are runn
# the URLs generated by Kibana, your proxy is expected to remove t
# to Kibana. This setting cannot end in a slash.
#server.basePath: ""

# The maximum payload size in bytes for incoming server requests.
#server.maxPayloadBytes: 1048576

# The Kibana server's name.  This is used for display purposes.
#server.name: "your-hostname"

# The URL of the Elasticsearch instance to use for all your querie
elasticsearch.url: "http://192.168.67.163:9200"

# When this setting's value is true Kibana uses the hostname speci
```

启动

在 kibana 主目录 bin 目录下执行

nohup ./kibana &

然后 ctrl+c 退出

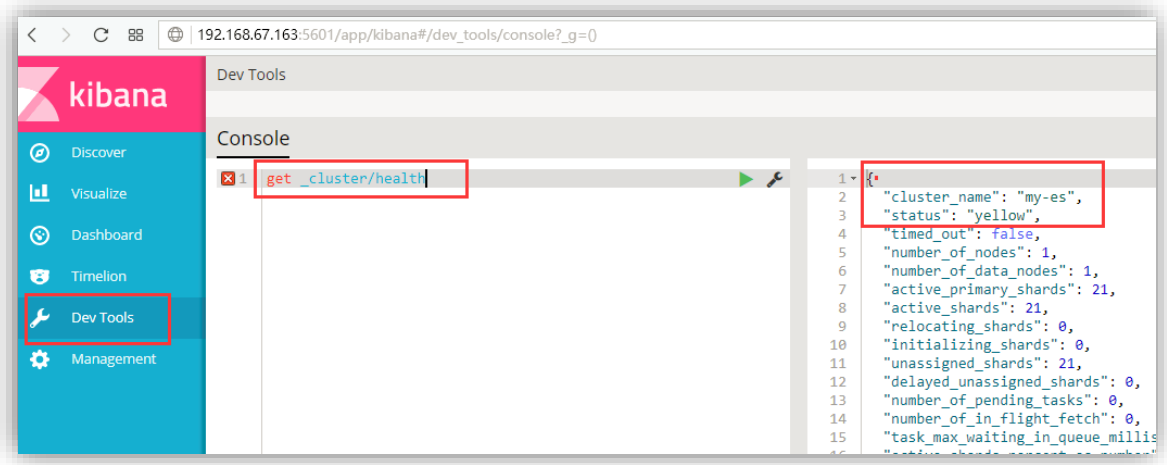
执行 ps -ef

```
root       1757   1285  4 23:38 pts/0    00:00:10 ./../node/bin/node --no-warnings ./../src/cli
```

如上图,1757 号进程就是 kibana 的进程

用浏览器打开

http://192.168.xx.xx:5601/



点击左边菜单 DevTools

在 Console 中

执行 get _cluster/health

右边的结果中，status 为 yellow 或者 green。

表示 es 启动正常，并且与 kibana 连接正常。