Elasticsearch指南

一、概述

官网地址： https://www.elastic.co/cn/products/elasticsearch

Elasticsearch是什么

ElasticSearch是一个基于Lucene的搜索服务器。它提供了一个分布式多用户能力的全文搜索引擎，基于RESTful web接口。Elasticsearch是用Java开发的，并作为Apache许可条款下的开放源码发布，是当前流行的企业级搜索引擎。设计用于**云计算**中，能够达到实时搜索，稳定，可靠，快速，安装使用方便。

Elasticsearch不仅仅是Lucene和全文搜索引擎，它还提供：

- 分布式的搜索引擎和数据分析引擎
- 全文检索
- 对海量数据进行近实时的处理

ElasticSearch的应用场景

- **全文检索**：主要和 Solr 竞争，属于后起之秀。
- NoSQL JSON文档数据库：**主要抢占 Mongo 的市**场，它在读写性能上优于 Mongo ，同时**也支持地理位置查询**，还方便**地理位置和文本混合查询**。
- 监控：统计、日志类时间序的数据存储和分析、可视化，这方面是引领者。
- 国外：Wikipedia（维基百科）使用ES提供全文搜索并高亮关键字、StackOverflow（IT问答网站）结合全文搜索与地理位置查询、Github使用Elasticsearch检索1300亿行的代码。
- 国内：百度（在云分析、网盟、预测、文库、钱包、风控等业务上都应用了ES，单集群每天导入30TB+数据，总共每天60TB+）、新浪 、阿里巴巴、腾讯等公司均有对ES的使用。
- 使用比较广泛的平台**ELK(ElasticSearch, Logstash, Kibana)**。

二、基本概念

RESTful介绍

参考资料：

1. http://www.ruanyifeng.com/blog/2011/09/restful.html
2. http://www.ruanyifeng.com/blog/2018/10/restful-api-best-practices.html

`REST`：表现层状态转化(Representational State Transfer)，如果一个架构符合REST原则，就称它为`RESTful`架构风格。

- **资源**： 所谓"资源"，就是网络上的一个实体，或者说是网络上的一个具体信息

- **表现层**：我们把"资源"具体呈现出来的形式，叫做它的"表现层"（Representation）。
- **状态转化（State Transfer）**：如果客户端想要操作服务器，必须通过某种手段，让服务器端发生"状态转化"（State Transfer）。而这种转化是建立在表现层之上的，所以就是"表现层状态转化"。就是HTTP协议里面，四个表示操作方式的动词：GET、POST、PUT、DELETE。它们分别对应四种基本操作：**GET用来获取资源，POST用来新建资源（也可以用于更新资源），PUT用来更新资源，DELETE用来删除资源。**

Elasticsearch中涉及到的重要概念

接近实时（NRT）

Elasticsearch是一个接近实时的搜索平台。这意味着，从索引一个文档直到这个文档能够被搜索到有一个轻微的延迟（通常是1秒）

集群（cluster）

一个集群就是由一个或多个节点组织在一起，它们共同持有你整个的数据，并一起提供索引和搜索功能。一个集群由一个唯一的名字标识，这个名字默认就是“elasticsearch”。这个名字是重要的，因为一个节点只能通过指定某个集群的名字，来加入这个集群。在产品环境中显式地设定这个名字是一个好习惯，但是使用默认值来进行测试/开发也是不错的。

节点（node）

一个节点是你集群中的一个服务器，作为集群的一部分，它存储你的数据，参与集群的索引和搜索功能。和集群类似，一个节点也是由一个名字来标识的，默认情况下，这个名字是一个随机的漫威漫画角色的名字，这个名字会在启动的时候赋予节点。这个名字对于管理工作来说挺重要的，因为在这个管理过程中，你会去确定网络中的哪些服务器对应于Elasticsearch集群中的哪些节点。

一个节点可以通过配置集群名称的方式来加入一个指定的集群。默认情况下，每个节点都会被安排加入到一个叫做“elasticsearch”的集群中，这意味着，如果你在你的网络中启动了若干个节点，并假定它们能够相互发现彼此，它们将会自动地形成并加入到一个叫做“elasticsearch”的集群中。

在一个集群里，只要你想，可以拥有任意多个节点。而且，如果当前你的网络中没有运行任何Elasticsearch节点，这时启动一个节点，会默认创建并加入一个叫做“elasticsearch”的集群。

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。索引类似于关系型数据库中Database的概念。在一个集群中，如果你想，可以定义任意多的索引。

类型（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。类型类似于关系型数据库中Table的概念。

文档（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，只要你想，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。文档类似于关系型数据库中Record的概念。实际上一个文档除了用户定义的数据外，还包括 *index*、 type和_id字段

分片和复制（shards & replicas）

一个索引可以存储超出单个结点硬件限制的大量数据。比如，一个具有10亿文档的索引占据1TB的磁盘空间，而任一节点都没有这样大的磁盘空间；或者单个节点处理搜索请求，响应太慢。

为了解决这个问题，Elasticsearch提供了将索引划分成多份的能力，这些份就叫做分片。当你创建一个索引的时候，你可以指定你想要的分片的数量。每个分片本身也是一个功能完善并且独立的“索引”，这个“索引”可以被放置到集群中的任何节点上。 分片之所以重要，主要有两方面的原因：

- 允许你水平分割/扩展你的内容容量
- 允许你在分片（潜在地，位于多个节点上）之上进行分布式的、并行的操作，进而提高性能/吞吐量 至于一个分片怎样分布，它的文档怎样聚合回搜索请求，是完全由Elasticsearch管理的，对于作为用户的你来说，这些都是透明的。

在一个网络/云的环境里，失败随时都可能发生，在某个分片/节点不知怎么的就处于离线状态，或者由于任何原因消失了。这种情况下，有一个故障转移机制是非常有用并且是强烈推荐的。为此目的，Elasticsearch允许你创建分片的一份或多份拷贝，这些拷贝叫做复制分片，或者直接叫复制。复制之所以重要，主要有两方面的原因：

- 在分片/节点失败的情况下，提供了高可用性。因为这个原因，注意到复制分片从不与原/主要（original/primary）分片置于同一节点上是非常重要的。
- 扩展你的搜索量/吞吐量，因为搜索可以在所有的复制上并行运行

总之，每个索引可以被分成多个分片。一个索引也可以被复制0次（意思是没有复制）或多次。一旦复制了，每个索引就有了主分片（作为复制源的原来的分片）和复制分片（主分片的拷贝）之别。分片和复制的数量可以在索引创建的时候指定。在索引创建之后，你可以在任何时候动态地改变复制数量，但是不能改变分片的数量。

默认情况下，Elasticsearch中的每个索引被分片5个主分片和1个复制，这意味着，如果你的集群中至少有两个节点，你的索引将会有5个主分片和另外5个复制分片（1个完全拷贝），这样的话每个索引总共就有10个分片。一个索引的多个分片可以存放在集群中的一台主机上，也可以存放在多台主机上，这取决于你的集群机器数量。主分片和复制分片的具体位置是由ES内在的策略所决定的。

映射（Mapping）

Mapping是ES中的一个很重要的内容，它类似于传统关系型数据中table的schema，用于定义一个索引（index）的某个类型（type）的数据的结构。

在ES中，我们无需手动创建type（相当于table）和mapping(相关与schema)。在默认配置下，ES可以根据插入的数据自动地创建type及其mapping。

mapping中主要包括字段名、字段数据类型和字段索引类型

三、Elasticsearch环境搭建

准备工作

- CentOS（ *版本需大于7 如：CentOS-7-x86_64-Minimal-1804.iso* ）
- Java(*版本需大于1.8 如：jdk-8u181-linux-x64.rpm*)
- ES安装包（ *如： elasticsearch-6.4.0.tar.gz* ）

环境搭建

安装Java

```
[root@localhost ~]# rpm -ivh jdk-8u181-linux-x64.rpm
warning: jdk-8u181-linux-x64.rpm: Header V3 RSA/SHA256 Signature, key ID ec551f03: NOKEY
Preparing...                          ################################# [100%]
Updating / installing...
    1:jdk1.8-2000:1.8.0_181-fcs       ################################# [100%]
Unpacking JAR files...
    tools.jar...
    plugin.jar...
    javaws.jar...
    deploy.jar...
    rt.jar...
    jsse.jar...
    charsets.jar...
    localedata.jar...

[root@localhost latest]# vi /etc/profile
export JAVA_HOME=/usr/java/latest
export CLASSPATH=.
export PATH=$PATH:$JAVA_HOME/bin

[root@localhost latest]# source /etc/profile
```

安装ES

```
[root@localhost ~]# tar -zxvf elasticsearch-6.4.0.tar.gz -C /usr
```

启动

```
[root@localhost elasticsearch-6.4.0]# bin/elasticsearch
```

1. 出现异常： `Caused by: java.lang.RuntimeException: can not run elasticsearch as root`

   原因：ES不允许通过ROOT用户启动

   解决方案：

   ```
   [root@localhost elasticsearch-6.4.0]# groupadd es
   [root@localhost elasticsearch-6.4.0]# useradd -g es es
   [root@localhost elasticsearch-6.4.0]# chown -R es:es /usr/elasticsearch-6.4.0/
   [root@localhost elasticsearch-6.4.0]# ll
   total 436
   drwxr-xr-x.  3 es es   4096 Dec 18 01:15 bin
   drwxr-xr-x.  2 es es    178 Dec 18 01:16 config
   drwxr-xr-x.  3 es es   4096 Aug 17 19:23 lib
   ```

```
-rw-r--r--.  1 es es  13675 Aug 17 19:11 LICENSE.txt
drwxr-xr-x.  2 es es    218 Dec 18 01:16 logs
drwxr-xr-x. 27 es es   4096 Aug 17 19:23 modules
-rw-r--r--.  1 es es 401465 Aug 17 19:22 NOTICE.txt
drwxr-xr-x.  2 es es      6 Aug 17 19:22 plugins
-rw-r--r--.  1 es es   8511 Aug 17 19:11 README.textile
[root@localhost elasticsearch-6.4.0]# su es
[es@localhost elasticsearch-6.4.0]$ bin/elasticsearch
```

测试

```
[root@localhost ~]# curl -X GET localhost:9200
{
  "name" : "x0vIhEF",
  "cluster_name" : "elasticsearch",
  "cluster_uuid" : "Cpga8etSTu6LmcYm35QqkA",
  "version" : {
    "number" : "6.4.0",
    "build_flavor" : "default",
    "build_type" : "tar",
    "build_hash" : "595516e",
    "build_date" : "2018-08-17T23:18:47.308994Z",
    "build_snapshot" : false,
    "lucene_version" : "7.4.0",
    "minimum_wire_compatibility_version" : "5.6.0",
    "minimum_index_compatibility_version" : "5.0.0"
  },
  "tagline" : "You Know, for Search"
}
```

配置远程访问

```
[root@localhost elasticsearch-6.4.0]# vim config/elasticsearch.yml
 51 # ---------------------------------- Network -----------------------------------
 52 #
 53 # Set the bind address to a specific IP (IPv4 or IPv6):
 54 #
 55 network.host: 192.168.23.141
 56 #
 57 # Set a custom port for HTTP:
 58 #
 59 #http.port: 9200
 60 #
 61 # For more information, consult the network module documentation.
 62 #
 63 # --------------------------------- Discovery ----------------------------------
```

1. 重新启动ES服务，出现异常

```
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 [3802] for user [es] 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]
```

解决方案（切换到root用户）：

```
[root@localhost elasticsearch-6.4.0]# vim /etc/security/limits.conf
# 添加以下内容
*   soft    nofile  65536
*   hard    nofile  131072
*   soft    nproc   2048
*   hard    nproc   4096

[root@localhost elasticsearch-6.4.0]# vim /etc/sysctl.conf
# 添加以下内容
vm.max_map_count=655360
[root@localhost elasticsearch-6.4.0]# sysctl -p
vm.max_map_count = 655360
[root@localhost elasticsearch-6.4.0]# reboot

[root@localhost ~]# systemctl stop firewalld
[root@localhost ~]# systemctl disable firewalld
Removed symlink /etc/systemd/system/multi-user.target.wants/firewalld.service.
Removed symlink /etc/systemd/system/dbus-org.fedoraproject.FirewallD1.service.
```

测试访问


```
{
  "name" : "x0vIhEF",
  "cluster_name" : "elasticsearch",
  "cluster_uuid" : "Cpga8etSTu6LmcYm35QqkA",
  "version" : {
    "number" : "6.4.0",
    "build_flavor" : "default",
    "build_type" : "tar",
    "build_hash" : "595516e",
    "build_date" : "2018-08-17T23:18:47.308994Z",
    "build_snapshot" : false,
    "lucene_version" : "7.4.0",
    "minimum_wire_compatibility_version" : "5.6.0",
    "minimum_index_compatibility_version" : "5.0.0"
  },
  "tagline" : "You Know, for Search"
}
```


安装Kibana

概述

Kibana是一个针对Elasticsearch的开源数据分析及可视化平台，用来搜索、查看交互存储在Elasticsearch索引中的数据。使用Kibana，可以通过各种图表进行高级数据分析及展示。

Kibana让海量数据更容易理解。它操作简单，基于浏览器的用户界面可以快速创建仪表板（dashboard）实时显示Elasticsearch查询动态。

设置Kibana非常简单。无需编码或者额外的基础架构，几分钟内就可以完成Kibana安装并启动Elasticsearch索引监测。

安装配置

```
[root@localhost ~]# tar -zxvf kibana-6.4.0-linux-x86_64.tar.gz -C /usr
[root@localhost ~]# cd /usr/kibana-6.4.0-linux-x86_64/
[root@localhost kibana-6.4.0-linux-x86_64]# vim config/kibana.yml

# To allow connections from remote users, set this parameter to a non-loopback address.
server.host: "192.168.23.141"

# The URL of the Elasticsearch instance to use for all your queries.
elasticsearch.url: "http://192.168.23.141:9200"

[root@localhost kibana-6.4.0-linux-x86_64]# bin/kibana
```

启动测试

http://192.168.23.141:5601

四、使用Kibana实现基本的增删改查

查看集群

查看集群健康信息

GET /_cat/health?v

```
epoch      timestamp cluster       status node.total node.data shards pri relo init unassign pending_tasks max_task_wait_time active_shards_percent
1545270315 20:45:15  elasticsearch yellow          1         1      5   5    0    0        5             0                  -                 50.0%
```

集群状态（status）

- Green（正常）
- Yellow（正常，但是一些副本还没有分配）
- Red（非正常）

可以使用 GET /_cat/health?help 查看每个操作返回结果字段的意义

查看集群中节点信息

GET /_cat/nodes?v

```
ip             heap.percent ram.percent cpu load_1m load_5m load_15m node.role master name
192.168.23.141            9          91   7    0.10    0.08     0.13 mdi       *      x0vIhEF
```

查看集群中的索引信息

GET /_cat/indices?v

```
health status index  uuid                   pri rep docs.count docs.deleted store.size
yellow open   baizhi BYzhTHMzQIKEiyaKXklueQ   5   1          0            0      1.2kb
```

简化写法

GET /_cat/indices?v&h=health,status,index

```
health status index
yellow open   baizhi
```

索引操作

创建索引

PUT /baizhi

```
#! Deprecation: the default number of shards will change from [5] to [1] in 7.0.0; if you wish to continue using the default of [5] shards, you must manage this on the create index request or with an index template
{
  "acknowledged": true, # 创建成功返回true
  "shards_acknowledged": true,
  "index": "baizhi"
}
```

上面的操作使用默认的配置信息创建一个索引

删除索引

DELETE /baizhi

```
{
  "acknowledged": true
}
```

创建类型Mapping

```
PUT /baizhi # 创建index(baizhi)并添加类型mapping(_doc)
{
  "mappings": {
    "_doc": {
      "properties": {
        "title":    { "type": "text"  },
        "name":     { "type": "text"  },
        "age":      { "type": "integer" },
        "created":  {
          "type":   "date",
          "format": "strict_date_optional_time||epoch_millis"
        }
      }
    }
  }
}
```

或

```
POST /baizhi/user # 创建index(baizhi)后，在指定index中添加类型mapping(user)
{
  "user": {
      "properties": {
        "id":    { "type": "text"  },
        "name":      { "type": "text"  },
        "age":       { "type": "integer" },
        "created":  {
          "type":   "date",
          "format": "strict_date_optional_time||epoch_millis"
        }
      }
    }
}
```

> Mapping Type：
>
> 1. 简单类型： `text` , `keyword` , `date` , `long` , `double` , `boolean` or `ip`
> 2. 其它类型： `object` , `geo_point` , `geo_shape` 等

查看类型mapping

```
GET /baizhi/_mapping/_doc # 语法：GET /索引名/_mapping/类型名

-------------------------------------------
{
  "baizhi": {
    "mappings": {
      "_doc": {
        "properties": {
          "age": {
            "type": "integer"
          },
          "created": {
            "type": "date"
          },
          "name": {
            "type": "text"
          },
          "title": {
            "type": "text"
          }
        }
      }
    }
  }
}
```

> **注意**：mapping types将会在ES 7.0版本中移除。原因可参考： **https://www.elastic.co/guide/en/elasticsearch/reference/current/removal-of-types.html** #_why_are_mapping_types_being_removed

文档操作

新增单个文档

```
PUT /baizhi/_doc/1 # put /索引名/类型名/id
{                  # request body
  "name":"zs",
  "title":"张三",
  "age":18,
  "created":"2018-12-25"
}
```

或

```
POST /baizhi/_doc
{
  "name":"ls",
  "title":"李四",
  "age":28,
  "created":"2018-12-26"
}
-------------------------------------------------------------
{
  "_index": "baizhi",
  "_type": "_doc",
  "_id": "KbOj6GcBVEuCC3JSh18Y", # ES自动生成的文档的id
  "_version": 1,
  "result": "created",
  "_shards": {
    "total": 2,
    "successful": 1,
    "failed": 0
  },
  "_seq_no": 0,
  "_primary_term": 1
}
```

查询单个文档

```
GET /baizhi/_doc/1  # 语法： GET /索引名/类型名/id
-------------------------------------------------------------------
{
  "_index": "baizhi",
  "_type": "_doc",
  "_id": "1",
  "_version": 1,
  "found": true,
  "_source": {
    "name": "zs",
    "title": "张三",
    "age": 18,
    "created": "2018-12-25"
  }
```

```
}

GET /baizhi/_doc/KbOj6GcBVEuCC3JSh18Y
--------------------------------------------------------------------
{
  "_index": "baizhi",
  "_type": "_doc",
  "_id": "KbOj6GcBVEuCC3JSh18Y",
  "_version": 1,
  "found": true,
  "_source": {
    "name": "ls",
    "title": "李四",
    "age": 28,
    "created": "2018-12-26"
  }
}
```

修改单个文档

```
PUT /baizhi/_doc/KbOj6GcBVEuCC3JSh18Y  # 语法： PUT /索引名/类型名/id
{
  "name":"lxs",
  "title":"李小四"
}
--------------------------------------------------------------------
{
  "_index": "baizhi",
  "_type": "_doc",
  "_id": "KbOj6GcBVEuCC3JSh18Y",
  "_version": 2,
  "found": true,
  "_source": {
    "name": "lxs",
    "title": "李小四"
  }
}
```

删除单个文档

```
DELETE /baizhi/_doc/1   # 语法： DELETE /索引名/类型名/id

--------------------------------------------------------------------
{
  "_index": "baizhi",
  "_type": "_doc",
  "_id": "1",
  "_version": 2,
  "result": "deleted",
  "_shards": {
    "total": 2,
    "successful": 1,
    "failed": 0
```

```
  },
  "_seq_no": 1,
  "_primary_term": 1
}
```

批处理操作

除了能够索引、更新和删除单个文档外，Elasticsearch还提供了使用 `_bulk API` 批量执行上述任何操作的能力。这个功能非常重要，因为它提供了一种非常有效的机制，可以以尽可能少的网络往返尽可能快地执行多个操作

```
POST /baizhi/_doc/_bulk # 批量插入多个document
{"index":{}}
{"name":"ww","title":"王五","age":18,"created":"2018-12-27"}
{"index":{}}
{"name":"zl","title":"赵六","age":25,"created":"2018-12-27"}
---------------------------------------------------------------------
{
  "took": 65,
  "errors": false, # 批量插入成功
  "items": [
    {
      "index": {
        "_index": "baizhi",
        "_type": "_doc",
        "_id": "KrOP6WcBVEuCC3JS8V9K",
        "_version": 1,
        "result": "created",
        "_shards": {
          "total": 2,
          "successful": 1,
          "failed": 0
        },
        "_seq_no": 0,
        "_primary_term": 1,
        "status": 201
      }
    },
    {
      "index": {
        "_index": "baizhi",
        "_type": "_doc",
        "_id": "K7OP6WcBVEuCC3JS8V9K",
        "_version": 1,
        "result": "created",
        "_shards": {
          "total": 2,
          "successful": 1,
          "failed": 0
        },
        "_seq_no": 0,
        "_primary_term": 1,
        "status": 201
      }
```

```
    }
  ]
}
```

```
POST /baizhi/_doc/_bulk  # 批量操作（包含修改和删除）
{"update":{"_id":"KrOP6WcBVEuCC3JS8V9K"}}  # 修改
{"doc":{"title":"王小五"}}
{"delete":{"_id":"K7OP6WcBVEuCC3JS8V9K"}}  # 删除
```

五、深入搜索

搜索方式

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 DSL(Request Body) 进行搜索

> DSL：Domain Specified Language，**特定领域语言**
>
> 使用请求体可以让你的JSON数据以一种更加可读和更加富有展现力的方式发送。

导入测试数据

```
# 批量插入测试数据
POST /zpark/user/_bulk
{"index":{"_id":1}}
{"name":"zs","realname":"张三","age":18,"birthday":"2018-12-27","salary":1000.0,"address":"北京市昌平区沙阳路55号"}
{"index":{"_id":2}}
{"name":"ls","realname":"李四","age":20,"birthday":"2017-10-20","salary":5000.0,"address":"北京市朝阳区三里屯街道21号"}
{"index":{"_id":3}}
{"name":"ww","realname":"王五","age":25,"birthday":"2016-03-15","salary":4300.0,"address":"北京市海淀区中关村大街新中关商城2楼511室"}
{"index":{"_id":4}}
{"name":"zl","realname":"赵六","age":20,"birthday":"2003-04-19","salary":12300.0,"address":"北京市海淀区中关村软件园9号楼211室"}
{"index":{"_id":5}}
{"name":"tq","realname":"田七","age":35,"birthday":"2001-08-11","salary":1403.0,"address":"北京市海淀区西二旗地铁辉煌国际大厦负一楼"}
```

查询(Query)

1. 查看所有并按照年龄降序排列

> 查询所有并排序

- URL实现

```
GET /zpark/user/_search?q=*&sort=age:desc&pretty
```

- DSL实现

```
GET /zpark/user/_search
{
  "query":{
    "match_all":{}  # 查询所有
  },
  "sort":{
      "age":"desc"  # 按年龄倒序排列
    }
}
```

2. 查询第2页的用户（每页显示2条）

> 分页查询

- URL实现

```
GET /zpark/user/_search?q=*&sort=_id:asc&from=2&size=2
```

- DSL实现

```
GET /zpark/user/_search
{
  "query":{
    "match_all":{}  # 查询所有
  },
  "sort":{
      "_id":"asc"   # 按年龄倒序排列
   },
  "from":2,         # 从（nowPage-1）*pageSize检索
  "size":2          # 查 pageSize条
}
```

3. 查询 address 在海淀区的所有用户，并 高亮

> 基于全文检索的查询（分析检索关键词 匹配索引库 返回结果）

- DSL实现

```
GET /zpark/user/_search
{
  "query": {
    "match": {
      "address":"海淀区"
    }
  },
  "highlight": {
    "fields": {     # 需要高亮的字段列表
      "address": {}
    }
  }
}
```

4. 查询 name 是 zs 关键字的用户

> 基于Term词元查询

- URL实现

```
GET /zpark/user/_search?q=name:zs
```

- DSL实现

```
GET /zpark/user/_search
{
  "query":{
    "term": {
      "name": {
        "value": "zs"
      }
    }
  }
}
```

5. 查询年龄在 20~30 岁之间的用户

> 基于范围查询

- DSL实现

```
GET /zpark/user/_search
{
  "query": {
    "range": {
      "age": {
        "gte": 20,
        "lte": 30
      }
    }
  }
}
```

6. 查询真实姓名以 张 开头的用户

> 基于前缀（prefix）查询

- DSL实现

```
GET /zpark/user/_search
{
  "query": {
    "prefix": {
      "realname": {
        "value": "李"
      }
    }
  }
}
```

7. 查询名字已 s 结尾的用户

> 基于通配符（wildcard）的查询
>
> - ? 匹配一个字符
> - * 匹配0~n个字符

- DSL实现

```
GET /zpark/user/_search
{
  "query": {
    "wildcard": {
      "name": {
        "value": "*s"
      }
    }
  }
}
```

8. 查询 id 为1，2，3的用户

> 基于Ids的查询

- DSL实现

```
GET /zpark/user/_search
{
  "query": {
    "ids": {
      "values": [1,2,3]
    }
  }
}
```

9. 模糊查询 realname 中包含 张 关键字的用户

> 基于Fuzzy的查询

- DSL实现

```
GET /zpark/user/_search
{
  "query": {
    "fuzzy": {
      "realname": {"value": "张"}
    }
  }
}
```

10. 查询 age 在15-30岁之间并且 name 必须通配z*

> 基于Boolean的查询（多条件查询）
>
> - must ：查询结果必须符合该查询条件（列表）。
> - should ：类似于or的查询条件。
> - must_not ：查询结果必须不符合查询条件（列表）。

- DSL实现

```
GET /zpark/user/_search
{
  "query": {
    "bool": {
      "must": [     #年龄在15~30岁之间并且必须名字通配z*
        {
          "range": {
            "age": {
              "gte": 15,
              "lte": 30
            }
          }
        },
        {
          "wildcard": {
            "name": {
              "value": "z*"
            }
          }
        }
      ],
      "must_not": [   # 正则查询 name必须不能以s结尾
        {
          "regexp": {
            "name": ".*s"
          }
        }
```

```
        ]
      }
    }
}
```

过滤器（Filter）

其实准确来说，ES中的查询操作分为2种：**查询（query）和过滤（filter）。查询即是之前提到的query查询，它（查询）默认会计算每个返回文档的得分，然后根据得分排序。而过滤（filter）只会筛选出符合的文档，并不计算得分，且它可以缓存文档。**所以，单从性能考虑，过滤比查询更快。

换句话说，过滤适合在大范围筛选数据，而查询则适合精确匹配数据。一般应用时，**应先使用过滤操作过滤数据，然后使用查询匹配数据。**

过滤器使用

```
GET /zpark/user/_search
{
   "query":{
      "bool": {
        "must": [
          {"match_all": {}}
        ],
        "filter": {     # 过滤年龄大于等于25岁的用户
          "range": {
            "age": {
              "gte": 25
            }
          }
        }
      }
   }
}
```

> 注意： 过滤查询运行时先执行过滤语句，后执行普通查询

过滤器的类型

1. `term` 、 `terms` Filter

term、terms的含义与查询时一致。term用于精确匹配、terms用于多词条匹配

```
GET /zpark/user/_search
{
   "query":{
      "bool": {
        "must": [
          {"match_all": {}}
        ],
        "filter": {
          "terms": {
```

```
            "name": [
              "zs",
              "ls"
            ]
          }
        }
      }
    }
}
```

2. `ranage` filter

3. `exists` filter

`exists` 过滤指定字段没有值的文档

```
GET /zpark/user/_search
{
  "query": {
    "bool": {
      "must": [
        {
          "match_all": {}
        }
      ],
      "filter": {    # 排除salary为null的结果
        "exists": {
          "field": "salary"
        }
      }
    }
  },
  "sort": [
    {
      "_id": {
        "order": "asc"
      }
    }
  ]
}
```

或(相反操作)

```
GET /zpark/user/_search
{
  "query": {
    "bool": {
      "must_not": [
        {
          "exists": {
            "field": "salary"
          }
        }
      ]
```

```
    }
  }
}
```

4. ids filter

需要过滤出若干指定_id的文档，可使用标识符过滤器(ids)

```
GET /zpark/user/_search
{
  "query": {
    "bool": {
      "must": [
        {
          "match": {
            "address": "昌平区"
          }
        }
      ],
      "filter": {
        "ids": {   # id 过滤器
          "values": [
            1,
            2,
            3
          ]
        }
      }
    }
  }
}
```

5. 其余使用方式可查阅官网

> Note：
>
> Query和Filter更详细的对比可参考： https://blog.csdn.net/laoyang360/article/details/80468757

聚合（Aggregations）

https://www.elastic.co/guide/en/elasticsearch/reference/6.x/search-aggregations.html

聚合提供了功能可以分组并统计你的数据。理解聚合最简单的方式就是可以把它粗略的看做SQL的GROUP BY操作和SQL的聚合函数。

ES中常用的聚合：

- **metric**（度量）聚合：度量类型聚合主要针对的number类型的数据，需要ES做比较多的计算工作
- **bucketing**（桶）聚合：划分不同的“桶”，将数据分配到不同的“桶”里。非常类似sql中的group语句的含义

ES中的聚合API如下：

```
"aggregations" : {                  // 表示聚合操作，可以使用aggs替代
    "<aggregation_name>" : {        // 聚合名，可以是任意的字符串。用做响应的key，便于快速取得正确的响应数据。
        "<aggregation_type>" : {    // 聚合类别，就是各种类型的聚合，如min等
            <aggregation_body>      // 聚合体，不同的聚合有不同的body
        }
        [,"aggregations" : { [<sub_aggregation>]+ } ]? // 嵌套的子聚合，可以有0或多个
    }
    [,"<aggregation_name_2>" : { ... } ]* // 另外的聚合，可以有0或多个
}
```

度量（metric）聚合

1. Avg Aggregation

平均值查询，作用于number类型字段上。如：查询用户的平均年龄

```
POST /zpark/user/_search
{
  "aggs": {
    "age_avg": {
      "avg": {"field": "age"}
    }
  }
}
-----------------------------------------------------------------------
{
    ......
    "aggregations": {
    "age_avg": {
      "value": 23.6
    }
  }
}
```

也可以先过滤，再进行统计，如：

```
POST /zpark/user/_search
{ "query": {
    "ids": {
      "values":[1,2,3]
    }
  },
  "aggs": {
    "age_avg": {
      "avg": {"field": "age"}
    }
  }
}
```

2. Max Aggregation

最大值查询。如：查询员工的最高工资

```
POST /zpark/user/_search
{
  "aggs": {
    "max_salary": {
      "max": {
        "field": "salary"
      }
    }
  }
}
```

3. Min Aggregation

4. Sum Aggregation

5. Stats Aggregation

统计查询，一次性统计出某个字段上的常用统计值

```
POST /zpark/user/_search
{
  "aggs": {
    "max_salary": {
      "stats": {
        "field": "salary"
      }
    }
  }
}
-------------------------------------------
{
    ....
    "aggregations": {
    "max_salary": {
      "count": 4,
      "min": 1000,
      "max": 12300,
      "avg": 5650,
      "sum": 22600
    }
  }
}
```

桶（bucketing）聚合

1. Range Aggregation

自定义区间范围的聚合，我们可以自己手动地划分区间，ES会根据划分出来的区间将数据分配不同的区间上去。

如： 统计0-20岁，20-35岁，35~60岁用户人数

```
POST /zpark/user/_search
{
  "aggs": {
    "age_ranges": {
```

```
      "range": {
        "field": "age",
        "ranges": [
          {
            "from": 0,
            "to": 20
          },
          {
            "from": 20,
            "to": 35
          },
          {
            "from": 35,
            "to": 60
          }
        ]
      }
    }
  }
}
-------------------------------------------------------------
{
    ......
    "aggregations": {
    "age_ranges": {
      "buckets": [
        {
          "key": "0.0-20.0",
          "from": 0,
          "to": 20,
          "doc_count": 1    # 区间范围的文档数量
        },
        {
          "key": "20.0-35.0",
          "from": 20,
          "to": 35,
          "doc_count": 3
        },
        {
          "key": "35.0-60.0",
          "from": 35,
          "to": 60,
          "doc_count": 1
        }
      ]
    }
  }
}
```

2. Terms Aggregation

自定义分组依据Term，对分组后的数据进行统计

如：根据年龄分组，统计相同年龄的用户

```
POST /zpark/user/_search
{
  "aggs": {
    "age_counts":{
      "terms": {
        "field": "age",
        "size": 2  // 保留2个统计结果
      }
    }
  }
}
------------------------------------------------------------
{
    ......
    "aggregations": {
    "age_counts": {
      "doc_count_error_upper_bound": 0,
      "sum_other_doc_count": 2,
      "buckets": [
        {
          "key": 20,
          "doc_count": 2
        },
        {
          "key": 18,
          "doc_count": 1
        }
      ]
    }
  }
}
```

3. Date Range Aggregation

时间区间聚合专门针对date类型的字段，它与Range Aggregation的主要区别是其可以使用时间运算表达式。

- now+10y：表示从现在开始的第10年。
- now+10M：表示从现在开始的第10个月。
- 1990-01-10||+20y：表示从1990-01-01开始后的第20年，即2010-01-01。
- now/y：表示在年位上做舍入运算。

如： 统计生日在2018年、2017年、2016年的用户

```
POST /zpark/user/_search
{
  "aggs": {
    "date_counts": {
      "date_range": {
        "field": "birthday",
        "format": "yyyy-MM-dd",
        "ranges": [
          {
            "from": "now/y",   # 当前年的1月1日
            "to": "now"        # 当前时间
```

```
          },
          {
            "from": "now/y-1y",   # 当前年上一年的1月1日
            "to":"now/y"          # 当前年的1月1日
          },
          {
            "from": "now/y-2y",
            "to":"now/y-1y"
          }
        ]
      }
    }
  }
}
--------------------------------------------------------------------------------
{
    .....
  "aggregations": {
    "date_counts": {
      "buckets": [
        {
          "key": "2016-01-01-2017-01-01",
          "from": 1451606400000,
          "from_as_string": "2016-01-01",
          "to": 1483228800000,
          "to_as_string": "2017-01-01",
          "doc_count": 1
        },
        {
          "key": "2017-01-01-2018-01-01",
          "from": 1483228800000,
          "from_as_string": "2017-01-01",
          "to": 1514764800000,
          "to_as_string": "2018-01-01",
          "doc_count": 1
        },
        {
          "key": "2018-01-01-2018-12-26",
          "from": 1514764800000,
          "from_as_string": "2018-01-01",
          "to": 1545847233691,
          "to_as_string": "2018-12-26",
          "doc_count": 0
        }
      ]
    }
  }
}
```

4. Histogram Aggregation

直方图聚合，它将某个number类型字段等分成n份，统计落在每一个区间内的记录数。它与前面介绍的Range聚合非常像，只不过Range可以任意划分区间，而Histogram做等间距划分。既然是等间距划分，那么参数里面必然有距离参数，就是interval参数。

如：根据年龄间隔（5岁）统计

```
POST /zpark/user/_search
{
  "aggs": {
    "histogram_age": {
      "histogram": {
        "field": "age",
        "interval": 5
      }
    }
  }
}
--------------------------------------------------------------------------------
{
    ......
    "aggregations": {
    "histogram_age": {
      "buckets": [
        {
          "key": 15,
          "doc_count": 1
        },
        {
          "key": 20,
          "doc_count": 2
        },
        {
          "key": 25,
          "doc_count": 1
        },
        {
          "key": 30,
          "doc_count": 0
        },
        {
          "key": 35,
          "doc_count": 1
        }
      ]
    }
  }
}
```

5. Date Histogram Aggregation

日期直方图聚合，专门对时间类型的字段做直方图聚合。这种需求是比较常用见得的，我们在统计时，通常就会按照固定的时间断（1个月或1年等）来做统计。

如：按年统计用户

```
POST /zpark/user/_search
{
  "aggs": {
    "date_histogram": {
      "date_histogram": {
        "field": "birthday",
        "interval": "year",
        "format": "yyyy-MM-dd"
      }
    }
  }
}
--------------------------------------------------------------------------------
{
    ......
    "aggregations": {
    "date_histogram": {
      "buckets": [
        {
          "key_as_string": "2001-01-01",
          "key": 978307200000,
          "doc_count": 1
        },
        {
          "key_as_string": "2002-01-01",
          "key": 1009843200000,
          "doc_count": 0
        },
        {
          "key_as_string": "2003-01-01",
          "key": 1041379200000,
          "doc_count": 1
        },
        {
          "key_as_string": "2004-01-01",
          "key": 1072915200000,
          "doc_count": 0
        },
        {
          "key_as_string": "2005-01-01",
          "key": 1104537600000,
          "doc_count": 0
        },
        {
          "key_as_string": "2006-01-01",
          "key": 1136073600000,
          "doc_count": 0
        },
        {
          "key_as_string": "2007-01-01",
          "key": 1167609600000,
          "doc_count": 0
        },
```

```
        {
          "key_as_string": "2008-01-01",
          "key": 1199145600000,
          "doc_count": 0
        },
        {
          "key_as_string": "2009-01-01",
          "key": 1230768000000,
          "doc_count": 0
        },
        {
          "key_as_string": "2010-01-01",
          "key": 1262304000000,
          "doc_count": 0
        },
        {
          "key_as_string": "2011-01-01",
          "key": 1293840000000,
          "doc_count": 0
        },
        {
          "key_as_string": "2012-01-01",
          "key": 1325376000000,
          "doc_count": 0
        },
        {
          "key_as_string": "2013-01-01",
          "key": 1356998400000,
          "doc_count": 0
        },
        {
          "key_as_string": "2014-01-01",
          "key": 1388534400000,
          "doc_count": 0
        },
        {
          "key_as_string": "2015-01-01",
          "key": 1420070400000,
          "doc_count": 0
        },
        {
          "key_as_string": "2016-01-01",
          "key": 1451606400000,
          "doc_count": 1
        },
        {
          "key_as_string": "2017-01-01",
          "key": 1483228800000,
          "doc_count": 1
        },
        {
          "key_as_string": "2018-01-01",
          "key": 1514764800000,
```

```
          "doc_count": 1
        }
      ]
    }
  }
}
```

嵌套使用

聚合操作是可以嵌套使用的。通过嵌套，可以使得metric类型的聚合操作作用在每一bucket上。我们可以使用ES的嵌套聚合操作来完成稍微复杂一点的统计功能。

如：统计每年中用户的最高工资

```
POST /zpark/user/_search
{
  "aggs": {
    "date_histogram": {          # bucket聚合  按照年分区
      "date_histogram": {
        "field": "birthday",
        "interval": "year",
        "format": "yyyy-MM-dd"
      },
      "aggs": {
        "salary_max": {
          "max": {               # metric聚合  求最大工资
            "field": "salary"
          }
        }
      }
    }
  }
}
-------------------------------------------------------------------------------------
{
    ......
    "aggregations": {
    "date_histogram": {
      "buckets": [
        ......
        {
          "key_as_string": "2017-01-01",
          "key": 1483228800000,
          "doc_count": 1,
          "salary_max": {
            "value": 5000
          }
        },
        {
          "key_as_string": "2018-01-01",
          "key": 1514764800000,
          "doc_count": 1,
```

```
          "salary_max": {
            "value": 1000
          }
        }
      ]
    }
  }
}
```

六、JAVA API

https://spring.io/projects/spring-data-elasticsearch#overview

查询方式

- **Restful API**

  基于http协议，使用JSON为数据交换格式，通过9200端口的与Elasticsearch进行通信

- **JAVA API**（*Spring Data ElasticSearch*）

  Spring Data ElasticSearch封装了与ES交互的实现细节，可以使系统开发者以Spring Data Repository 风格实现与ES的数据交互。Elasticsearch为Java用户提供了两种内置客户端：

  节点客户端(node client)： 节点客户端以无数据节点(none data node)身份加入集群，换言之，它自己不存储任何数据，但是它知道数据在集群中的具体位置，并且能够直接转发请求到对应的节点上。

  传输客户端(Transport client)： 这个更轻量的传输客户端能够发送请求到远程集群。它自己不加入集群，只是简单转发请求给集群中的节点。两个Java客户端都通过9300端口与集群交互，使用Elasticsearch传输协议(Elasticsearch Transport Protocol)。集群中的节点之间也通过9300端口进行通信。如果此端口未开放，你的节点将不能组成集群。

Spring Data ElasticSearch实践

Maven依赖

```
<dependencies>
    <dependency>
        <groupId>org.springframework.data</groupId>
        <artifactId>spring-data-elasticsearch</artifactId>
        <version>3.1.3.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-test</artifactId>
        <version>5.1.3.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.12</version>
    </dependency>
```

```
</dependencies>
```

准备配置文件

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:elasticesearch="http://www.springframework.org/schema/data/elasticsearch"
       xmlns:elasticsearch="http://www.springframework.org/schema/data/elasticsearch"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/data/elasticsearch
http://www.springframework.org/schema/data/elasticsearch/spring-elasticsearch.xsd">

    <!-- Spring data 自动扫描es repository接口，生成实现类 -->
    <elasticsearch:repositories base-package="com.baizhi.es.dao"></elasticsearch:repositories>

    <!-- ip:port换成具体的ip和端口，多个以逗号分隔 -->
    <elasticesearch:transport-client id="client" cluster-name="elasticsearch"
                                     cluster-nodes="192.168.23.141:9300">
</elasticesearch:transport-client>

    <!-- es操作对象-->
    <bean id="elasticsearchTemplate"
class="org.springframework.data.elasticsearch.core.ElasticsearchTemplate">
        <constructor-arg name="client" ref="client"></constructor-arg>
    </bean>

    <bean id="customUserRepository" class="com.baizhi.es.dao.CustomUserRepositoryImpl">

    </bean>
</beans>
```

准备映射实体类

```
package com.baizhi.entity;

import org.springframework.data.annotation.Id;
import org.springframework.data.annotation.PersistenceConstructor;
import org.springframework.data.elasticsearch.annotations.Document;

import java.util.Date;

/**
 * @author gaozhy
 * @date 2018/12/28.17:05
 */
// 文档注解 用于描述索引及其相关信息
@Document(indexName = "zpark",type = "user")
public class User {

    // 主键
    @Id
```

```
    private String id;

    private String name;

    private String realname;

    private Integer age;

    private Double salary;

    private Date birthday;

    private String address;

    public User() {
    }

    // 从es中恢复数据时使用的构造方法
    @PersistenceConstructor
    public User(String id, String name, String realname, Integer age, Double salary, Date
birthday, String address) {
        this.id = id;
        this.name = name;
        this.realname = realname;
        this.age = age;
        this.salary = salary;
        this.birthday = birthday;
        this.address = address;
    }
    // 省略get/set toString方法 ......
}
```

spring data repository

spring data elsaticsearch提供了三种构建查询模块的方式：

- 基本的增删改查：继承spring data提供的接口就默认提供
- 接口中声明方法：无需实现类，spring data根据方法名，自动生成实现类，方法名必须符合一定的规则（这里还扩展出一种忽略方法名，根据注解的方式查询）
- 自定义repository：在实现类中注入elasticsearchTemplate，实现上面两种方式不易实现的查询（例如：聚合、分组、深度翻页等）

> 上面的第一点和第二点只需要声明接口，无需实现类，spring data会扫描并生成实现类

基础的repository接口：提供基本的增删改查和根据方法名的查询

```
package com.baizhi.es.dao;

import com.baizhi.entity.User;
import org.springframework.data.elasticsearch.annotations.Query;
import org.springframework.data.elasticsearch.repository.ElasticsearchRepository;
```

```
import java.util.List;

/**
 * 基础操作的es repository接口（定义的有通用的增删改查方法）
 *
 * @author gaozhy
 * @date 2018/12/29.9:26
 */
public interface UserRepository extends ElasticsearchRepository<User,String> {

    /**
     * 根据年龄区间查询数据 并根据年龄降序排列
     */
    public List<User> findByAgeBetweenOrderByAgeDesc(int start,int end);

    /**
     * 查询真实姓名已“王”开头的数据
     */
    public List<User> findByRealnameStartingWith(String startStr);

    /**
     * 通过Query注解自定义查询表达式
     */
    @Query("{\"bool\" : {\"must\" : {\"fuzzy\" : {\"name\" : \"?0\"}}}}")
    public List<User> findByNameLike(String name);
}
```

测试代码如下：

```
package com.baizhi.es.test;

import com.baizhi.entity.User;
import com.baizhi.es.dao.UserRepository;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.data.domain.Sort;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

import java.util.Date;
import java.util.List;
import java.util.Optional;

/**
 * @author gaozhy
 * @date 2018/12/29.9:30
 */
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:applicationContext-es.xml")
public class UserRepositoryTest {
```

```
    @Autowired
    private UserRepository userRepository;

    /**
     * 查所有
     */
    @Test
    public void testQueryAll(){
        Iterable<User> users = userRepository.findAll();
        for (User user : users) {
            System.out.println(user);
        }
    }

    /**
     * 查询所有 并根据年龄倒序排列
     */
    @Test
    public void testQueryBySort(){
        Iterable<User> users = userRepository.findAll(Sort.by(Sort.Direction.DESC, "age"));
        for (User user : users) {
            System.out.println(user);
        }
    }

    /**
     * 根据id查询
     */
    @Test
    public void testQueryById(){
        Optional<User> user = userRepository.findById("1");
        System.out.println(user.get());
    }

    /**
     * 新增或者修改数据
     */
    @Test
    public void testAdd(){
        User user = userRepository.save(new User("6", "wb", "王八", 26, 10000D, new Date(), "河南省郑州市二七区德化街南路33号"));
        System.out.println(user);
    }

    //==========================================================================

    /**
     * 接口中声明方法查询：
     *     根据年龄区间查询数据 并根据年龄降序排列
     */
    @Test
    public void testQueryByRange(){
```

```
        List<User> users = userRepository.findByAgeBetweenOrderByAgeDesc(20, 28);
        users.forEach(user -> System.out.println(user));
    }

    /**
     * 接口中声明方法查询：
     *    查询真实姓名已“王”开头的数据
     *
     *    响应结果：
     *    User{id='6', name='wb', realname='王八', age=26, salary=10000.0, birthday=Sat Dec 29
14:38:39 CST 2018, address='河南省郑州市二七区德化街南路33号'}
          User{id='3', name='ww', realname='王五', age=25, salary=4300.0, birthday=Tue Mar 15
08:00:00 CST 2016, address='北京市海淀区中关村大街新中关商城2楼511室'}
     */
    @Test
    public void testQueryByPrefix(){
        List<User> users = userRepository.findByRealnameStartingWith("王");
        users.forEach(user -> System.out.println(user));
    }

    //=========================================================================
    /**
     * 通过Query注解自定义查询表达式
     */
    @Test
    public void testQueryByNameLike(){
        List<User> users = userRepository.findByNameLike("zs");
        users.forEach(user -> System.out.println(user));
    }
}
```

自定义Repository接口：使用 elasticsearchTemplate 实现复杂查询

自定义 CustomUserRepository 接口

```
package com.baizhi.es.dao;

import com.baizhi.entity.User;

import java.util.List;
import java.util.Map;

/**
 * @author gaozhy
 * @date 2019/1/1.23:10
 */
public interface CustomUserRepository {

    public List<User> findByPageable(int nowPage,int pageSize);

    public List<User> findByFieldDesc(String field);
```

```
    public List<User> findByRealNameLikeAndHighLight(String realName);

    public List<User> findByNameWithTermFilter(String ...terms);

    public List<User> findByAgeWithRangeFilter(int start,int end);

    public Map findByNameStartingWithAndAggregations(String prefixName);

    /**
     * 嵌套查询：
     *
     * 先按年龄直方图（桶聚合）统计
     * 然后再统计区间内员工的最高工资（度量聚合）
     */
    public Map aggregationsWithHistogramAndMax();

    /**
     * 日期直方图（桶聚合）
     */
    public Map aggregationsWithDateHistogram();

}
```

自定义 CustomUserRepositoryImpl 实现类

```
package com.baizhi.es.dao;

import com.baizhi.entity.User;
import org.elasticsearch.action.search.SearchResponse;
import org.elasticsearch.search.SearchHit;
import org.elasticsearch.search.SearchHits;
import org.elasticsearch.search.aggregations.Aggregation;
import org.elasticsearch.search.aggregations.AggregationBuilders;
import org.elasticsearch.search.aggregations.Aggregations;
import org.elasticsearch.search.aggregations.bucket.histogram.DateHistogramInterval;
import org.elasticsearch.search.fetch.subphase.highlight.HighlightBuilder;
import org.elasticsearch.search.sort.SortBuilders;
import org.elasticsearch.search.sort.SortOrder;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.domain.PageRequest;
import org.springframework.data.domain.Pageable;
import org.springframework.data.elasticsearch.core.ElasticsearchTemplate;
import org.springframework.data.elasticsearch.core.ResultsExtractor;
import org.springframework.data.elasticsearch.core.SearchResultMapper;
import org.springframework.data.elasticsearch.core.aggregation.AggregatedPage;
import org.springframework.data.elasticsearch.core.aggregation.impl.AggregatedPageImpl;
import org.springframework.data.elasticsearch.core.query.NativeSearchQueryBuilder;
import org.springframework.data.elasticsearch.core.query.SearchQuery;

import java.util.ArrayList;
import java.util.Date;
import java.util.List;
```

```java
import java.util.Map;

import static org.elasticsearch.index.query.QueryBuilders.*;

/**
 * @author gaozhy
 * @date 2019/1/1.23:11
 */
public class CustomUserRepositoryImpl implements CustomUserRepository {

    @Autowired
    private ElasticsearchTemplate template;

    /**
     * ====================================
     * {
     * "query": {
     * "match_all": {}
     * },
     * "from":1,      //从第几条开始    (从0开始)
     * "size":1       //大小
     * }
     * ====================================
     *
     * @param nowPage
     * @param pageSize
     * @return
     */
    @Override
    public List<User> findByPageable(int nowPage, int pageSize) {
        SearchQuery query = new NativeSearchQueryBuilder()
                .withQuery(matchAllQuery())
                .withPageable(new PageRequest((nowPage - 1) * pageSize, pageSize))
                .build();

        return template.queryForList(query, User.class);
    }

    /**
     * @param field
     * @return
     */
    @Override
    public List<User> findByFieldDesc(String field) {
        SearchQuery query = new NativeSearchQueryBuilder()
                .withQuery(matchAllQuery())
                .withSort(SortBuilders.fieldSort(field).order(SortOrder.DESC))
                .build();
        return template.queryForList(query, User.class);
    }

    /**
     * 高亮
```

```
     *
     * @param realName
     * @return
     */
    @Override
    public List<User> findByRealNameLikeAndHighLight(String realName) {
        SearchQuery query = new NativeSearchQueryBuilder()
                .withQuery(matchQuery("realname", realName)
                )
                .withHighlightFields(new HighlightBuilder.Field("realname"))
                .build();
        AggregatedPage<User> users = template.queryForPage(query, User.class, new
SearchResultMapper() {
            @Override
            public <T> AggregatedPage<T> mapResults(SearchResponse searchResponse, Class<T>
aClass, Pageable pageable) {
                ArrayList<User> users = new ArrayList<>();
                SearchHits searchHits = searchResponse.getHits();
                for (SearchHit searchHit : searchHits) {
                    if (searchHits.getHits().length <= 0) {
                        return null;
                    }
                    User user = new User();
                    user.setId(searchHit.getId());
                    // searchHit.getSourceAsMap().forEach((k, v) -> System.out.println(k + " "
+ v));
                    user.setName(searchHit.getSourceAsMap().get("name").toString());
                    user.setAddress(searchHit.getSourceAsMap().get("address").toString());

 user.setAge(Integer.parseInt(searchHit.getSourceAsMap().get("age").toString()));
                    user.setBirthday(new
Date(Long.parseLong(searchHit.getSourceAsMap().get("birthday").toString())));

 user.setSalary(Double.parseDouble(searchHit.getSourceAsMap().get("salary").toString()));
                    String realname =
searchHit.getHighlightFields().get("realname").fragments()[0].toString();
                    user.setRealname(realname);

                    users.add(user);
                }
                return new AggregatedPageImpl<T>((List<T>) users);
            }
        });
        return users.getContent();
    }

    @Override
    public List<User> findByNameWithTermFilter(String... terms) {
        SearchQuery query = new NativeSearchQueryBuilder()
                .withQuery(matchAllQuery())
                .withFilter(termsQuery("name",terms))
                .build();
        System.out.println(query.getFilter());
```

```
        return template.queryForList(query,User.class);
    }

    @Override
    public List<User> findByAgeWithRangeFilter(int start, int end) {
        SearchQuery query = new NativeSearchQueryBuilder()
                .withFilter(rangeQuery("age").gte(start).lte(end))
                .build();
        System.out.println(query.getQuery());
        System.out.println(query.getFilter());
        return template.queryForList(query,User.class);
    }

    @Override
    public Map<String, Aggregation> findByNameStartingWithAndAggregations(String prefixName) {
        SearchQuery query = new NativeSearchQueryBuilder()
                .withQuery(prefixQuery("name",prefixName))
                // result为度量聚合结果的别名
                .addAggregation(AggregationBuilders.avg("result").field("age"))
                .build();
        Aggregations aggregations = template.query(query, new ResultsExtractor<Aggregations>()
{
            @Override
            public Aggregations extract(SearchResponse searchResponse) {
                Aggregations aggregations = searchResponse.getAggregations();
                return aggregations;
            }
        });
        Map<String, Aggregation> map = aggregations.getAsMap();
        return map;
    }

    @Override
    public Map aggregationsWithHistogramAndMax() {
        SearchQuery query = new NativeSearchQueryBuilder()

.addAggregation(AggregationBuilders.histogram("result").field("age").interval(5)
                        .subAggregation(AggregationBuilders.max("max_salary").field("salary")))
                .build();
        Aggregations aggregations = template.query(query, new ResultsExtractor<Aggregations>()
{

            @Override
            public Aggregations extract(SearchResponse searchResponse) {
                return searchResponse.getAggregations();
            }
        });
        return aggregations.getAsMap();
    }

    @Override
    public Map aggregationsWithDateHistogram() {
        SearchQuery query = new NativeSearchQueryBuilder()
```

```
.addAggregation(AggregationBuilders.dateHistogram("result").field("birthday").format("yyyy-MM-dd").dateHistogramInterval(DateHistogramInterval.YEAR))
                .build();
        Aggregations aggregations = template.query(query, new ResultsExtractor<Aggregations>() {

            @Override
            public Aggregations extract(SearchResponse searchResponse) {
                return searchResponse.getAggregations();
            }
        });
        return aggregations.getAsMap();
    }
}
```

自定义 `CustomUserRepositoryTest` 测试类

```
package com.baizhi.es.test;

import com.baizhi.entity.User;
import com.baizhi.es.dao.CustomUserRepository;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

import java.util.List;
import java.util.Map;

/**
 * @author gaozhy
 * @date 2019/1/1.23:26
 */
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:applicationContext-es.xml")
public class CustomUserRepositoryTest {

    @Autowired
    private CustomUserRepository repository;

    @Test
    public void testQueryByPage(){
        List<User> users = repository.findByPageable(0, 2);
        users.forEach(user -> {
            System.out.println(user);
        });
    }
    @Test
    public void testQueryBySort(){
        List<User> users = repository.findByFieldDesc("_id");
        users.forEach(user -> {
```

```
            System.out.println(user);
        });
    }
    @Test
    public void testQueryByHighLight(){
        List<User> users = repository.findByRealNameLikeAndHighLight("王八");
        users.forEach(user -> {
            System.out.println(user);
        });
    }
    @Test
    public void testQueryByNameWithTermFilter(){
        List<User> users = repository.findByNameWithTermFilter("zs","ls");
        users.forEach(user -> {
            System.out.println(user);
        });
    }

    @Test
    public void testQueryByAgeWithRangeFilter(){
        List<User> users = repository.findByAgeWithRangeFilter(21,30);
        users.forEach(user -> {
            System.out.println(user);
        });
    }

    @Test
    public void testQueryByNameStartingWithAndAggregations(){
        Map map = repository.findByNameStartingWithAndAggregations("z");
        System.out.println(map.get("result"));
    }

    @Test
    public void testAggregationsWithHistogramAndMax(){
        Map map = repository.aggregationsWithHistogramAndMax();
        System.out.println(map.get("result"));
    }

    @Test
    public void testAggregationsWithDateHistogram(){
        Map map = repository.aggregationsWithDateHistogram();
        System.out.println(map.get("result"));
    }
}
```

七、集成中文分词器IK

参考资料： https://github.com/medcl/elasticsearch-analysis-ik

安装

```
[es@localhost root]$ cd /usr/elasticsearch-6.4.0/
[es@localhost elasticsearch-6.4.0]$ ./bin/elasticsearch-plugin install
https://github.com/medcl/elasticsearch-analysis-ik/releases/download/v6.4.0/elasticsearch-
analysis-ik-6.4.0.zip
-> Downloading https://github.com/medcl/elasticsearch-analysis-
ik/releases/download/v6.4.0/elasticsearch-analysis-ik-6.4.0.zip
[=================================================] 100%
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@     WARNING: plugin requires additional permissions     @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
* java.net.SocketPermission * connect,resolve
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
-> Installed analysis-ik
[es@localhost elasticsearch-6.4.0]$ ll plugins/
total 0
drwxr-xr-x. 2 es es 229 Jan  3 10:04 analysis-ik

# 重新启动es的服务
[root@localhost ~]# jps
2673 Elasticsearch
46151 Jps
40921 PluginCli
[root@localhost ~]# kill -9 2673
[root@localhost ~]# cd /usr/elasticsearch-6.4.0/
[root@localhost elasticsearch-6.4.0]# su es
[es@localhost elasticsearch-6.4.0]$ bin/elasticsearch
```

测试

创建测试索引

```
PUT /news
```

创建类型映射

```
POST /news/international/_mapping
{
  "properties": {
    "content":{
      "type": "text",
      "analyzer": "ik_max_word",  # 会将文本做最细粒度的拆分
      "search_analyzer": "ik_max_word"
    }
  }
}
```

插入测试数据

```
POST /news/international/_bulk
{"index":{"_id":1}}
{"content":"美国留给伊拉克的是个烂摊子吗"}
{"index":{"_id":2}}
{"content":"公安部：各地校车将享最高路权"}
{"index":{"_id":3}}
{"content":"中韩渔警冲突调查：韩警平均每天扣1艘中国渔船"}
{"index":{"_id":4}}
{"content":"中国驻洛杉矶领事馆遭亚裔男子枪击 嫌犯已自首"}
```

根据关键词高亮查询

```
GET /news/international/_search
{
  "query": {
    "match": {
      "content": "中国"
    }
  },
  "highlight": {
    "fields": {"content": {}}
  }
}

---------------------------------------------------------------------
{
  "took": 177,
  "timed_out": false,
  "_shards": {
    "total": 5,
    "successful": 5,
    "skipped": 0,
    "failed": 0
  },
  "hits": {
    "total": 2,
    "max_score": 0.6489038,
    "hits": [
      {
        "_index": "news",
        "_type": "international",
        "_id": "4",
        "_score": 0.6489038,
        "_source": {
          "content": "中国驻洛杉矶领事馆遭亚裔男子枪击 嫌犯已自首"
        },
        "highlight": {
          "content": [
            "<em>中国</em>驻洛杉矶领事馆遭亚裔男子枪击 嫌犯已自首"
          ]
        }
      },
      {
```

```
        "_index": "news",
        "_type": "international",
        "_id": "3",
        "_score": 0.2876821,
        "_source": {
          "content": "中韩渔警冲突调查：韩警平均每天扣1艘中国渔船"
        },
        "highlight": {
          "content": [
            "中韩渔警冲突调查：韩警平均每天扣1艘<em>中国</em>渔船"
          ]
        }
      }
    ]
  }
}
```

IK词典配置

```
[root@localhost analysis-ik]# vim /usr/elasticsearch-6.4.0/config/analysis-
ik/IKAnalyzer.cfg.xml

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
        <comment>IK Analyzer 扩展配置</comment>
        <!--用户可以在这里配置自己的扩展字典 -->
        <entry key="ext_dict">mydict.dic;</entry>
         <!--用户可以在这里配置自己的扩展停止词字典-->
        <entry key="ext_stopwords"></entry>
        <!--用户可以在这里配置远程扩展字典 -->
        <!--<entry key="remote_ext_dict">location</entry>-->
        <!--用户可以在这里配置远程扩展停止词字典-->
        <!--<entry key="remote_ext_stopwords">http://xxx.com/xxx.dic</entry>-->
</properties>
```

自定义扩展词测试

```
POST /_analyze
{
  "analyzer": "ik_max_word",
  "text": ["抖音视频真的好火啊"]
}
# 未添加扩展词 效果如下
--------------------------------------------------
{
  "tokens": [
    {
      "token": "抖",
      "start_offset": 0,
      "end_offset": 1,
```

```
      "type": "CN_CHAR",
      "position": 0
    },
    {
      "token": "音视频",
      "start_offset": 1,
      "end_offset": 4,
      "type": "CN_WORD",
      "position": 1
    },
    ......
  ]
}

# 添加扩展词 效果如下
---------------------------------------------------
{
  "tokens": [
    {
      "token": "抖音",
      "start_offset": 0,
      "end_offset": 2,
      "type": "CN_WORD",
      "position": 0
    },
    ......
    {
      "token": "腰子姐",
      "start_offset": 17,
      "end_offset": 20,
      "type": "CN_WORD",
      "position": 8
    }
  ]
}
```