Elasticsearch文档

1. Elasticsearch是什么

Elasticsearch是一个实时分布式搜索和分析引擎，它用于全文搜索、结构化搜索、分析以及将这三者混合使用。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。

但是，Lucene 只是一个库。想要使用它，你必须使用 Java 来作为开发语言并将其直接集成到你的应用中，更糟糕的是，Lucene 非常复杂，你需要深入了解检索的相关知识来理解它是如何工作的。

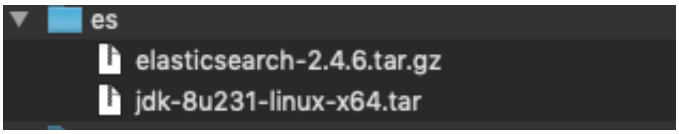
不过，Elasticsearch 不仅仅是 Lucene 和全文搜索，我们还能这样去描述它：

- 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
- 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

而且，所有的这些功能被集成到一个服务里面，你的应用可以通过简单的 `RESTful API`、各种语言的客户端甚至命令行与之交互。

2. Elasticsearch 的安装及启动

安装 Elasticsearch 之前，需要先安装一个较新的版本的 Java，最好的选择是，你可以从 https://www.oracle.com/technetwork/java/javase/downloads/index.html 获得官方提供的最新版本的 Java。



解压安装包jdk-8u231-linux-x64.tar

tar -zxvf jdk-8u231-linux-x64.tar

将解压后的文件夹移到/usr/lib目录下

切换到 /usr/lib目录下

```
cd  /usr/lib
```

并新建jdk目录

```
sudo mkdir jdk
```

将解压的jdk文件复制到新建的/usr/lib/jdk目录下来

```
sudo mv jdk1.8.0_231 /usr/lib/jdk
```

配置java环境变量

这里是将环境变量配置在 ~/.bashrc，即为所有用户配置 JDK 环境。

使用命令打开~/.bashrc文件

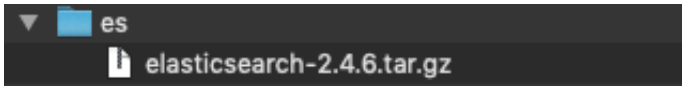
```
sudo vi ~/.bashrc
```

在末尾添加以下几行文字：

```
# set java env
export JAVA_HOME=/usr/lib/jdk/jdk1.8.0_231
export PATH=$JAVA_HOME/bin:$PATH
```

执行命令使修改立即生效

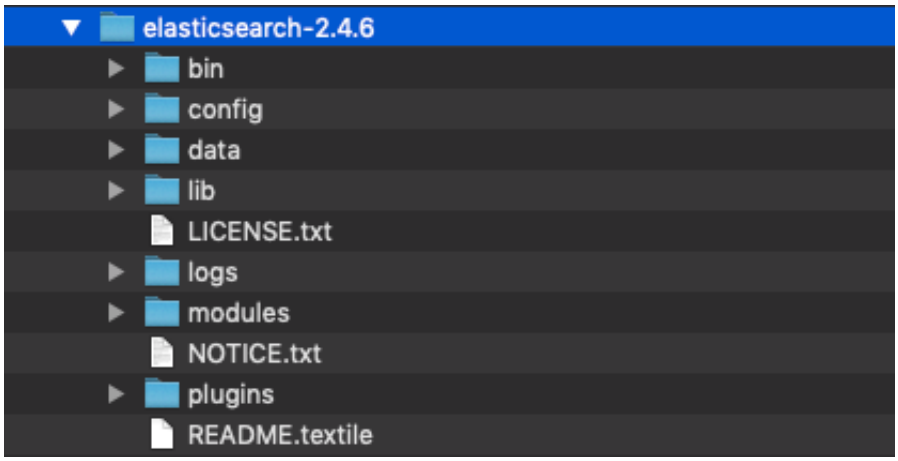
```
source ~/.bashrc
```



我们所使用的版本是 2.4.6，该版本较为稳定。我们首先安装 es：

```
tar -zxvf elasticsearch-2.4.6.tar.gz
```

解压后的目录文档如图所示：



打开 elasticsearc-2.4.6/config/elasticsearch.yml 文件，并定位到 54 行：

```
# Set the bind address to a specific IP (IPv4 or IPv6):
#
network.host: 127.0.0.1
#
# Set a custom port for HTTP:
#
# http.port: 9200
```

如果需要其他 ip 访问服务，则将 127.0.0.1 修改为 0.0.0.0

出于系统安全考虑的设置，elasticsearch不允许root用户启动，所以在启动之前需要创建一个非root用户：

创建一个新用户，用于启动elasticsearch(如有tarena用户，切换到tarena后直接启动es即可，无需以下操作)

1）创建新用户es

```
useradd es
```

2)设置权限(如不设置后期es服务会因为无法写入log日志而宕机)

```
chown -R root:es elasticsearch路径
```

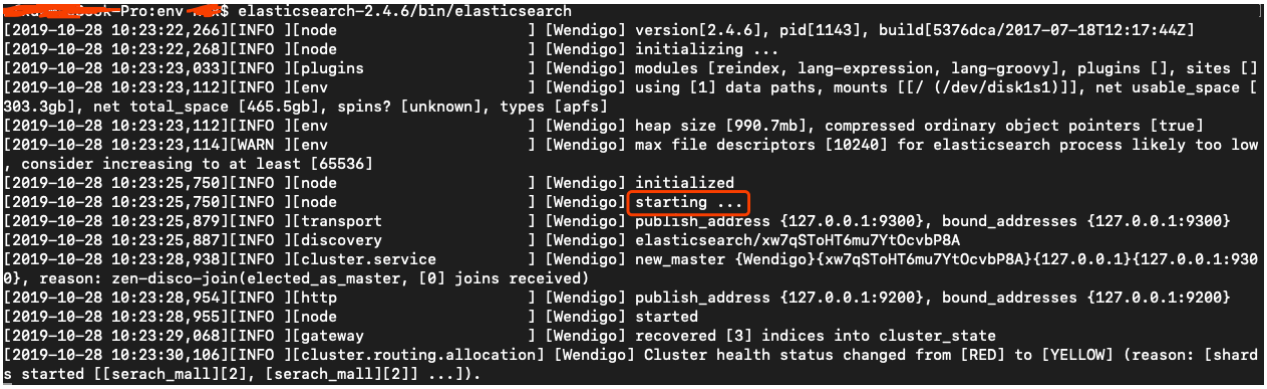
3)切换至用户es

```
su es
```

4)修改文件夹权限

```
chmod 770 elasticsearch路径
```

配置完毕后，运行 es，执行命令 elasticsearch-2.4.6/bin/elasticsearch



会出现 starting ... 的字样则，在浏览器出入 127.0.0.1:9200，会出现如下 json 样式：

```
{
  "name" : "Wendigo",
  "cluster_name" : "elasticsearch",
  "cluster_uuid" : "LmvJFxspQ_-yto4uP1-VFA",
  "version" : {
    "number" : "2.4.6",
    "build_hash" : "5376dca9f70f3abef96a77f4bb22720ace8240fd",
    "build_timestamp" : "2017-07-18T12:17:44Z",
    "build_snapshot" : false,
    "lucene_version" : "5.5.4"
  },
  "tagline" : "You Know, for Search"
}
```

至此，es 服务启动完毕。

3. Elasticsearch原理

- 反向索引又叫倒排索引(关于倒排索引的原理，请见 ***倒排索引.pdf***)，是根据文章内容中的关键字建立索引。
- 搜索引擎原理就是建立倒排索引。
- Elasticsearch 在 Lucene 的基础上进行封装，实现了分布式搜索引擎。

- Elasticsearch 中的索引、类型和文档的概念比较重要，类似于 MySQL 中的数据库、表和行。

4. Elasticsearch核心的http api

查看相关APIs

- **cat_indices**

```
说明：
indices负责提供索引的相关信息，包括组成一个索引（index）的shard、document的数量，删除的
doc数量，主存大小和所有索引的总存储大小。

命令：
GET /_cat/indices?v

注意：/_cat/indices?v 中的 v 是显示表头，即health, status, index......

返回值：
health status index      pri rep docs.count docs.deleted store.size
pri.store.size
yellow open   new_index    5   1          3            0      7.3kb
 7.3kb
yellow open   spu          5   1         18            0     66.7kb
66.7kb
yellow open   sku          5   1          0            0       795b
795b
yellow open   dadashp      5   1          4            0     16.5kb
16.5kb
yellow open   test_index   5   1         12            1     31.8kb
31.8kb
```

- **cat aliases(集群搭建，如果没有集群则暂无返回值)**

```
说明：
aliases 负责展示当前es集群配置别名包括filter和routing信息。

命令：
GET /_cat/aliases?v
GET /_cat/aliases/alias1,alias2

返回：
alias  index filter routing.index routing.search
alias1 test1 -      -             -
alias2 test1 *      -             -
alias3 test1 -      1             1
alias4 test1 -      2             1,2
```

- **cat allocation**

```
说明：
allocation负责展示es的每个数据节点分配的索引分片以及使用的磁盘空间。

命令：
GET /_cat/allocation?v

返回值：
shards disk.indices disk.used disk.avail disk.total disk.percent host      ip
       node
     5          260b    47.3gb     43.4gb    100.7gb           46 127.0.0.1
127.0.0.1 CSUXak2
```

- **cat master**

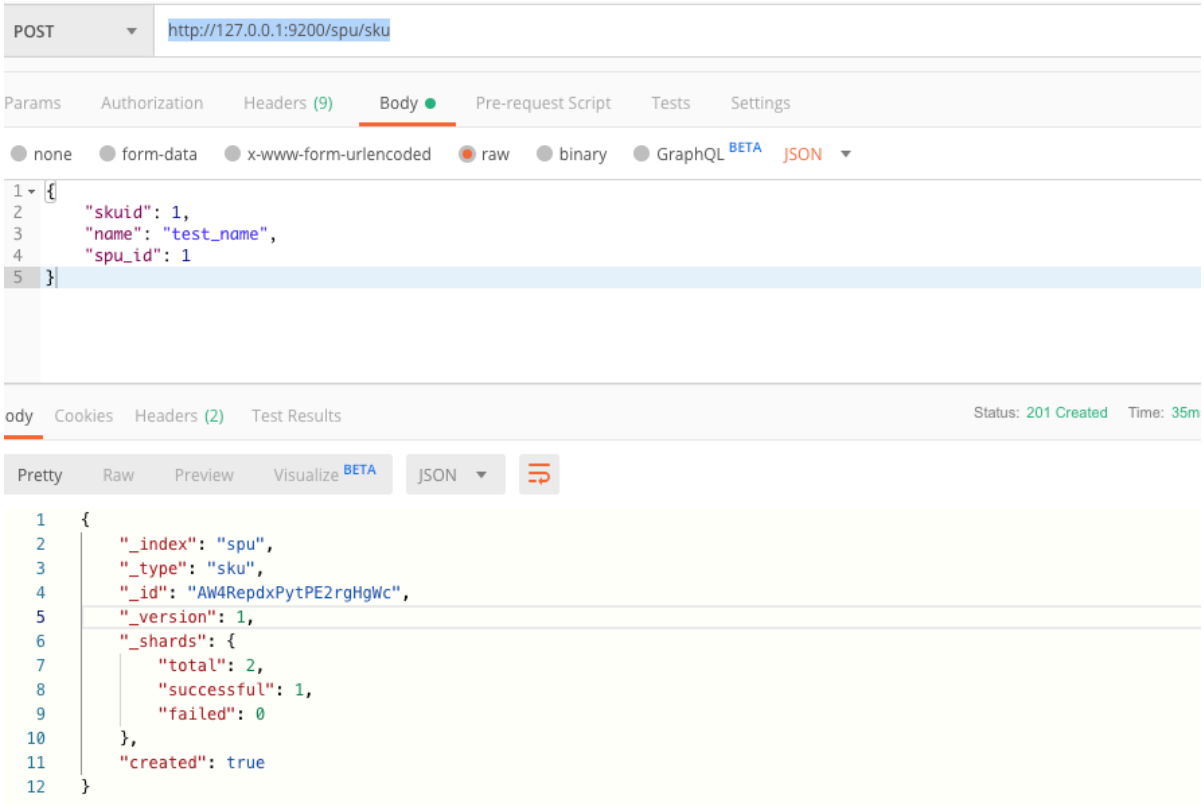
```
说明：
master负责展示es集群的master节点信息包括节点id、节点名、ip地址等。

命令：
GET /_cat/master?v

返回值：
id                     host      ip        node
YzWoH_2BT-6UjVGDyPdqYg 127.0.0.1 127.0.0.1 YzWoH_2
```

其他 Elasticsearch 的 HTTP API 请见：https://www.elastic.co/guide/en/elasticsearch/reference/current/rest-apis.html

新增文档

- **POST方式：http://127.0.0.1:9200/spu/sku，POST方式会自动生成id**
- **注意：如果想新创建索引index，则url改为 http://127.0.0.1:9200/new_index，如果想创建类型type，则url改为 http://127.0.0.1:9200/spu/new_type**

  测试结果



这里介绍下 "_shards"，翻译过来叫分片，es 中所有数据**均衡**的存储在集群中**各个节点**的**分片**中，会影响ES的性能、安全和稳定性，简单来讲就是在 ES 中所有数据的文件块，也是数据的最小单元块，整个 ES 集群的核心就是对所有分片的分布、索引、负载、路由等达到惊人的速度。

我们看到的是 total: 2，successful: 1，faild: 0，总共为2，但是失败的和成功的加一起是1，另一个1哪去了，这是为什么？下面引入两个概念：

`number_of_shards`

每个索引的主分片数，默认值是 `5` 。这个配置在索引创建后不能修改。

`number_of_replicas`

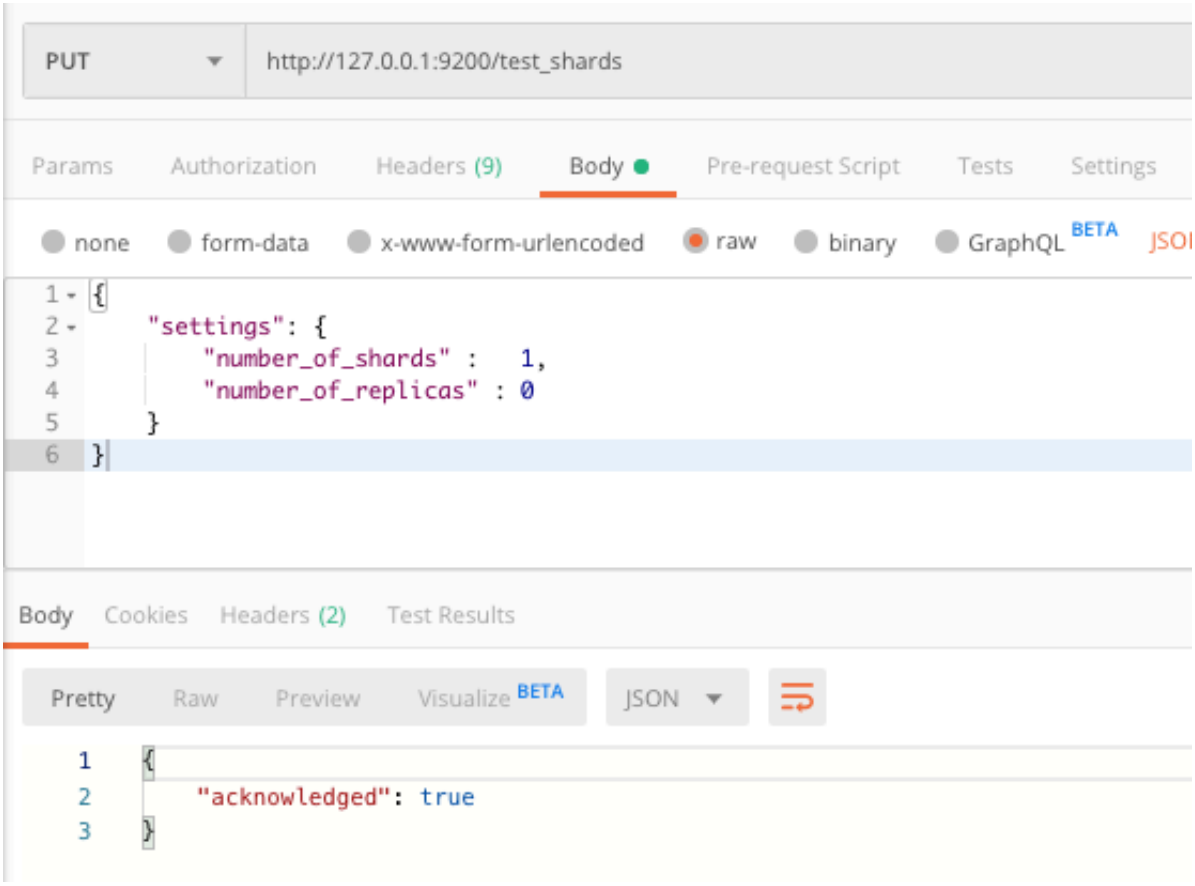
每个主分片的副本数，默认值是 `1` 。对于活动的索引库，这个配置可以随时修改。

查看 index 的切片：**GET方式**：http://127.0.0.1:9200/spu/sku

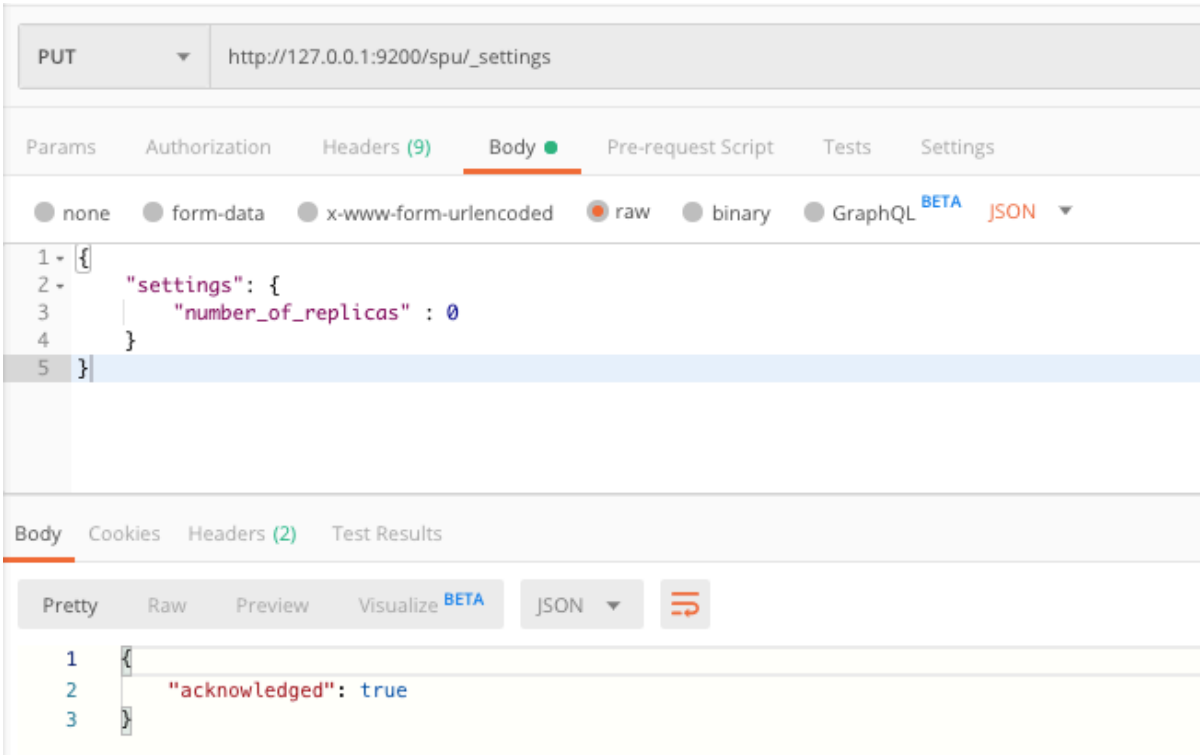
所以上面问题的原因就是我们没有用到副本，所以只成功 successful 的值为1。

如何设置切片的数量?

新创建 index 即可。**PUT方式**：http://127.0.0.1:9200/.....

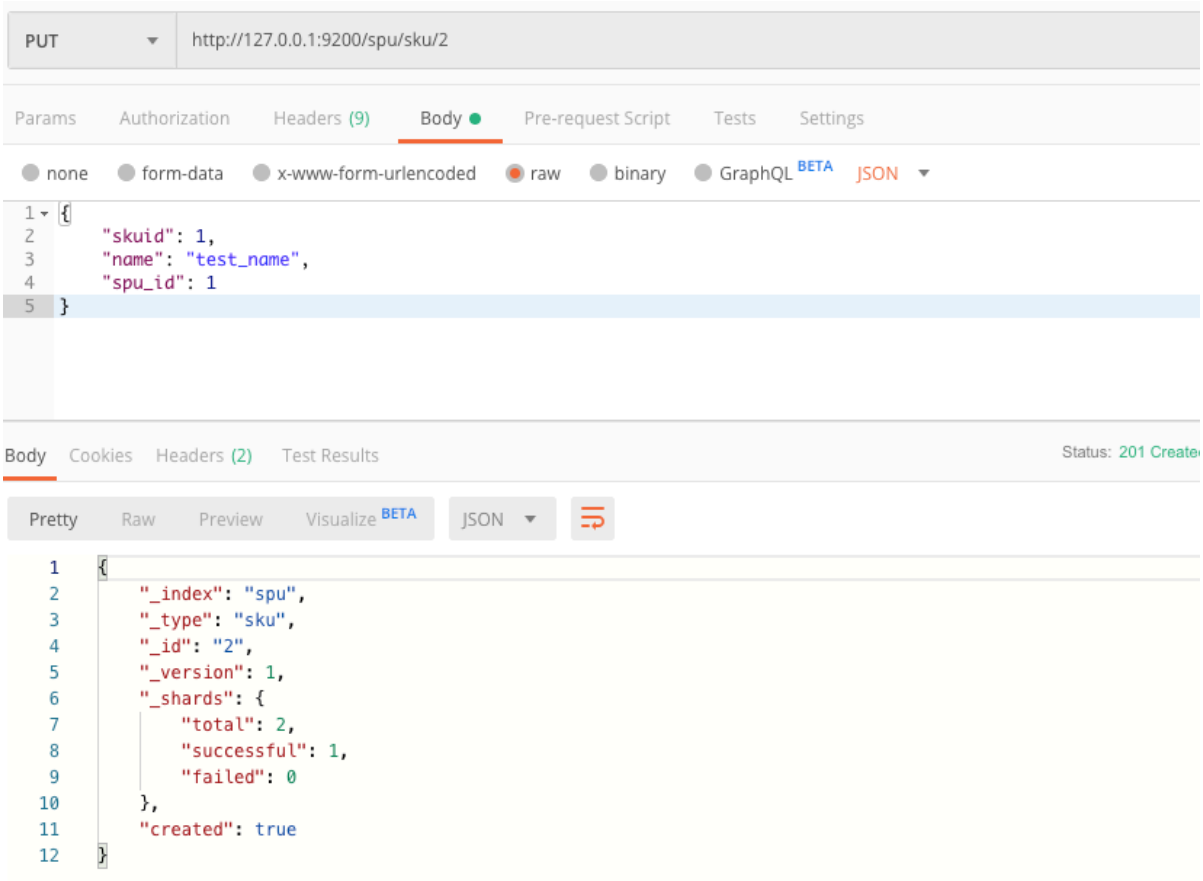


更新已存在 index 的切片副本：**PUT方式**：http://127.0.0.1:9200/spu/_settings



- **PUT方式：http://127.0.0.1:9200/spu/sku/2，PUT方式需要指定id，存在则更新，不存在则新增**

测试结果

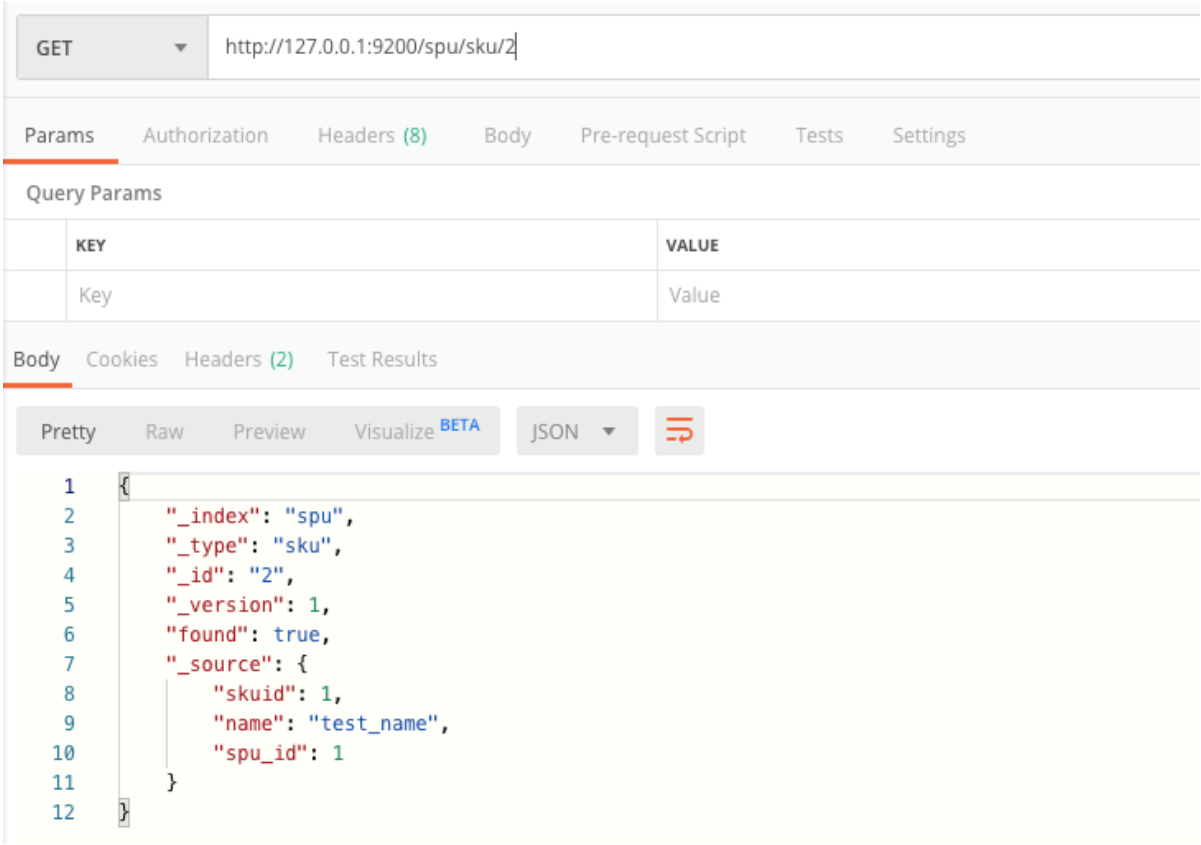


根据主键查询

- **GET方式 http://127.0.0.1:9200/spu/sku/2**
- **返回结果中 source 表示返回的 Doucment(文档)类容，其他几个是Elasticsearch文档结构字段。如果只需要source内容不需要其他结构字段，还可以在请求url上加上属性"_source"，将只返回source部分容，请求：**

```
http://127.0.0.1:9200/spu/sku/2/_source
```

测试结果

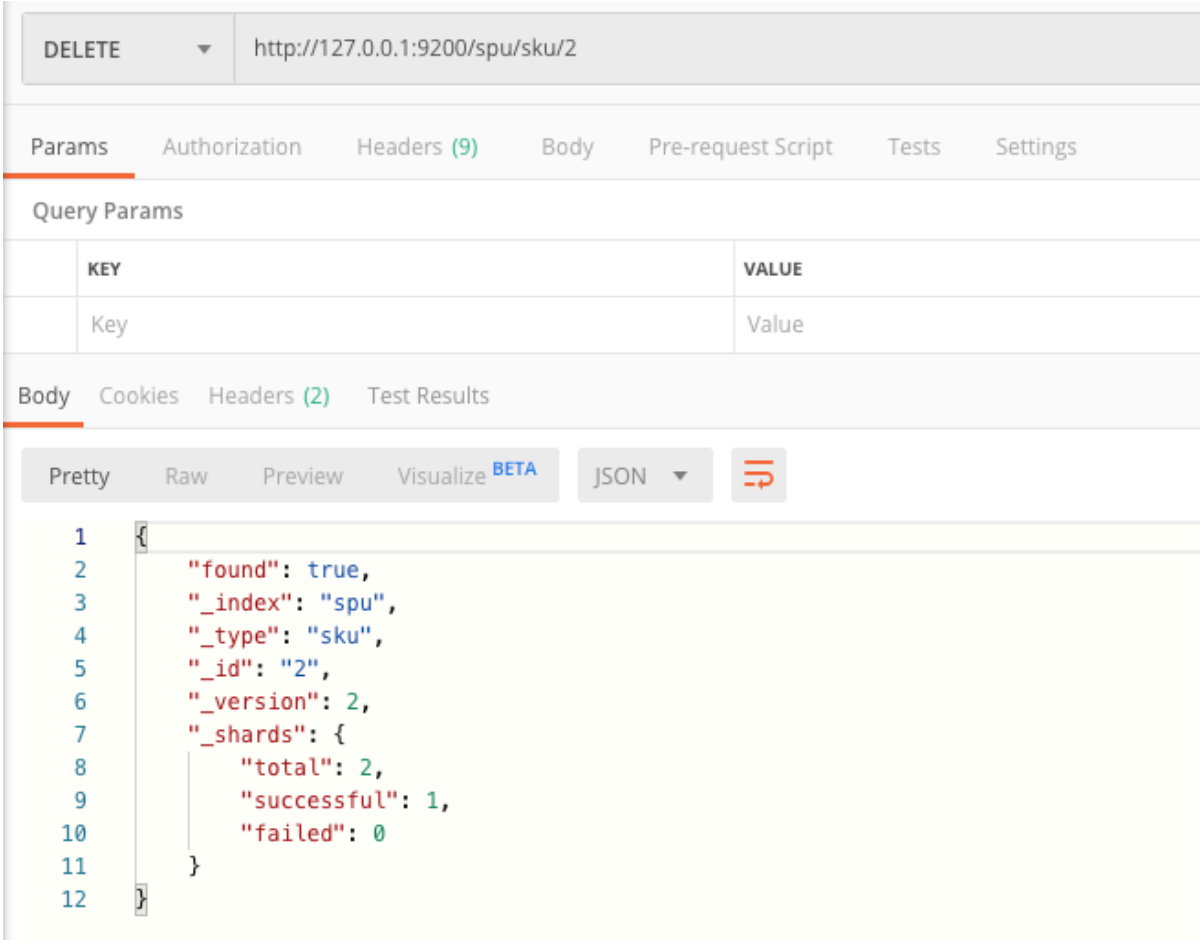


如果查询的id不存在，则搜索失败，失败结果：

根据主键删除

- DELETE方式 http://127.0.0.1:9200/spu/sku/2
- 注意：若想删除索引或者类型，只需修改 url 即可。

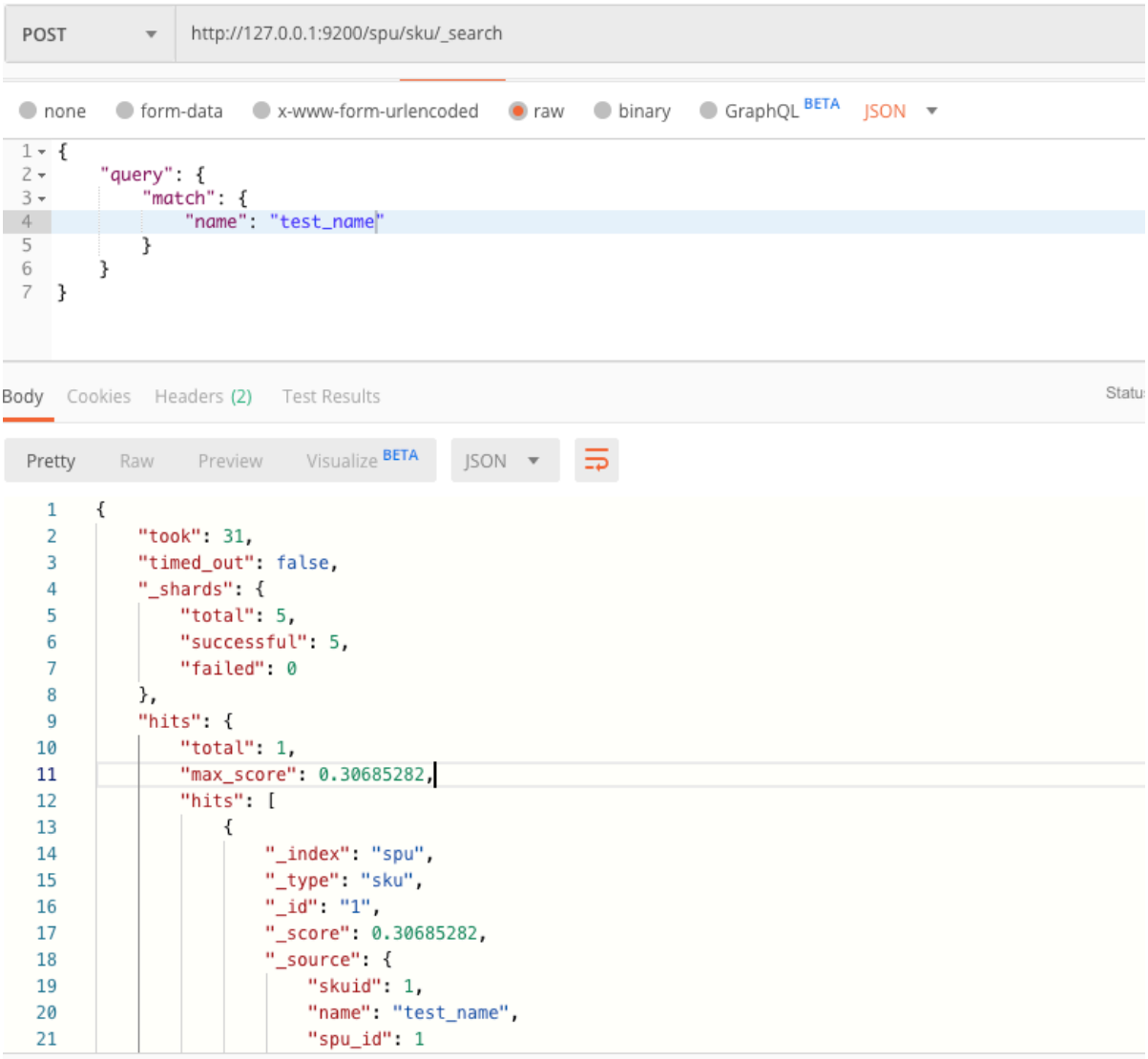
  测试结果



搜索文档

- **POST方式：使用match表达式进行全文搜索**
- **返回结果中_score是搜索引擎的概念，表示相关度，分数越高表示此文档与搜索条件关键字的匹配程度越高。**

  测试结果



5. 在Python中操作Elasticsearch

安装 ElasticSearch 模块：

```
pip3 install elasticsearch==2.4.1
pip3 install elasticstack==0.4.1
pip3 install django-haystack
```

这三个库的用处：

```
Django 不用提了。
Haystack 是将 Django 项目模型映射到搜索索引并搜索站点的最快方法。
ElasticSearch 是基于 Lucene 的搜索引擎和带有 JSON 接口的分布式数据存储。
elasticstack 是针对基于 Haystack 的项目的一组 ElasticSearch 特定帮助器。
```

基本操作：

添加数据

```
from elasticsearch import Elasticsearch

# 默认host为localhost,port为9200.但也可以指定host与port
# 在__init__方法中可以修改以下数据，根据情况自定义：
# def __init__(self, hosts=None, transport_class=Transport, **kwargs):
es = Elasticsearch()

# 1. 添加数据
# 添加或更新数据,index，doc_type名称可以自定义，id可以根据需求赋值,body为内容
es.index(index="test_index", doc_type="test_type", id=1, body=
{"name":"python","addr":"海淀"})
# 继续添加内容
es.index(index="test_index", doc_type="test_type", id=1, body=
{"name":"python","addr":"西二旗"})
```

查询数据

注意：es添加，修改或删除记录需要一点点时间

ES 的读取分为 get 和 search 两种操作，这两种读取操作有较大的差异，GET 必须指定三元组：index, type, id。也就是说，根据文档 id 从正排索引中获取内容。而 search 不指定 id，根据关键词从倒排索引中获取内容。

```
# 2. 查询数据
# 获取索引为test_index, 文档类型为test_type, result为一个字典类型
result = es.search(index="test_index",doc_type="test_type")
for item in result["hits"]["hits"]:
    print(item["_source"])

# 搜索id=1的文档
result = es.get(index="test_index",doc_type="test_type",id=1)
print(result)
```

删除数据

```
# 3. 删除数据
# 删除id=1的数据
result = es.delete(index="test_index",doc_type="test_type",id=1)
```

高阶查询：

1. match:匹配name包含python关键字的数据

```
# match:匹配name包含python关键字的数据
body = {
  "query":{
    "match":{
      "name":"python"
    }
  }
}
# 查询name包含python关键字的数据
es.search(index="test_index",doc_type="test_type",body=body)
# multi_match:在name和addr里匹配包含海淀关键字的数据
body = {
  "query":{
    "multi_match":{
      "query":"海淀",
      "fields":["name","addr"]
    }
  }
}
# 查询name和addr包含"海淀"关键字的数据
es.search(index="test_index",doc_type="test_type",body=body)
```

2. 搜索出id为1或2的所有数据

```
body = {
  "query":{
    "ids":{
      "type":"test_type",
      "values":[
        "1","2"
      ]
    }
  }
}
# 搜索出id为1或2的所有数据
es.search(index="test_index",doc_type="test_type",body=body)
```

3. 从第2条数据开始，获取4条数据（切片式查询）

```
body = {
    "query":{
        "match_all":{}
    },
    "from":2, # 从第二条数据开始
    "size":4 # 获取4条数据
}
# 从第2条数据开始，获取4条数据
es.search(index="test_index",doc_type="test_type",body=body)
```

4. 查询前缀为"p"的所有数据

```
body = {
  "query":{
    "prefix":{
      "name":"p"
    }
  }
}
# 查询前缀为"p"的所有数据
es.search(index="test_index",doc_type="test_type",body=body)
```

5. 获取name="python"并且addr="海淀"的数据

```
body = {
  "query":{
    "bool":{
      "must":[
        {
          "term":{
            "name":"python"
          }
        },
        {
          "term":{
            "addr":"海淀"
          }
        }
      ]
    }
  }
}
# 获取name="python"addr=海淀的所有数据
es.search(index="test_index", doc_type="test_type", body=body)
```

这里查不到数据的原因：

我们使用的是 es 的默认分词器，默认分词器会将中文拆分成一个个的单个汉字，搜索“海淀”，会被分析为**“海”**和**“淀”**，所以**“海”**和**“淀”**就是倒排索引，从而进行搜索。而对于搜索，我们常用的是 match 搜索，类似于数据库的**模糊查询**，而 term 搜索为**精确查询**。match 的底层使用的是 term 查询，将查询结果集根据 **_score** 进行排序，筛选出评分高的一项当作最终查询文档。

使用的时候会出现以下情况：

我们使用 match 搜索来搜索“海淀”，得到的结果显而易见是“海淀”，搜索“海”，得到的可能不仅是“海淀”，另外几个包含“海”的地名也会列出来，例如“后海”。但是当我们使用 term 搜索“海淀”时，我们得到的是空的结果，这也是显而易见的，因为倒排索引只有**“海”**和**“淀”**，使用“海淀”关键字去进行 term 精确查询是查找不到的。

默认的分词器会讲中文分成单个字，不会有任何联字，也就是说，两个字的“海淀”产生的倒排索引是“海“、”淀“，使用term查询时，必须是以上两个单字中的某一个才行。但是对于非文本类型，如 bool 和数字类型，或者大于小于等比较操作，使用term会非常的精准。

6. 查询name以on为后缀的所有数据

```
body = {
  "query":{
    "wildcard":{ # wildcard不能变
      "name":"*on"
    }
  }
}
# 查询name以on为后缀的所有数据
es.search(index="test_index", doc_type="test_type", body=body)
```

7. 执行查询并获取该查询的匹配数

```
# 获取数据量
es.count(index="test_index",doc_type="test_type")
```

8. 搜索所有数据，并获取age最小的值以及最大的值

注意：需要重新创建索引，以下案例涉及到数字类型的聚合操作，否则会报异常 Expected numeric type on field [age], but got [string]

原因：因为数据一旦写入，索引的类型就会固定，test_index 索引第一个传入的文档类型是 string 类型，所以该索引就会关闭 Numeric detection (数字探测)

```
es.index(index="new_index", doc_type="test_type", id=1, body=
{"name":"laowang1","age":18})
```

```
es.index(index="new_index", doc_type="test_type", id=2, body=
{"name":"laowang2","age":28})
es.index(index="new_index", doc_type="test_type", id=3, body=
{"name":"laowang3","age":38})

body = {
  "query":{
    "match_all":{}
  },
  "aggs":{              # 聚合查询，aggs不能变
    "min_age":{           # 最小值的key，min_age可以自定义，如min_age_value
      "min":{           # 最小，字段名不可自定义
        "field":"age"     # 查询"age"的最小值
      }
    }
  }
}
# 搜索所有数据，并获取age最小的值
es.search(index="new_index",doc_type="test_type",body=body)

body = {
  "query":{
    "match_all":{}
  },
  "aggs":{              # 聚合查询
    "max_age":{           # 最大值的key，字段名可自定义
      "max":{           # 最大，字段名不可自定义
        "field":"age"     # 查询"age"的最大值
      }
    }
  }
}
# 搜索所有数据，并获取age最大的值
es.search(index="new_index",doc_type="test_type",body=body)
```

9. 搜索所有数据，并获取所有age的和(沿用案例8的索引new_index)

```
body = {
  "query":{
    "match_all":{}
  },
  "aggs":{              # 聚合查询
    "sum_age":{           # 和的key，字段名可自定义
      "sum":{           # 和，字段名不可自定义
        "field":"age"     # 获取所有age的和
      }
    }
```

```
    }
}
# 搜索所有数据，并获取所有age的和
es.search(index="new_index",doc_type="test_type",body=body)
```

更多的搜索用法：https://elasticsearch-py.readthedocs.io/en/master/api.html

6. Elasticsearch与Django的结合

Elasticsearch与Django的结合需要用到haystack，haystack是一款同时支持whoosh，solr，Xapian，Elasticsearch四种全文检索引擎的第三方app，我们使用的是Elasticsearch搜索引擎。

首先需要安装第三方库：

```
pip3 install django-haystack
pip3 install elasticsearch==2.4.1
pip3 install elasticstack==0.4.1
```

这三个库都是做什么的?

```
Django 不用提了。
ElasticSearch 是基于 Lucene 的搜索引擎和带有 JSON 接口的分布式数据存储。
Haystack 是将 Django 项目模型映射到搜索索引并搜索站点的最快方法。
elasticstack 是针对基于 Haystack 的项目的一组 ElasticSearch 特定帮助器。
```

1. 修改settings.py

在settings.py中添加HAYSTACK的配置信息，ENGINE为使用elasticstack的引擎

```
# Haystack
HAYSTACK_CONNECTIONS = {
    'default': {
        'ENGINE': 'elasticstack.backends.ConfigurableElasticSearchEngine',
        'URL': 'http://127.0.0.1:9200/',
        'INDEX_NAME': 'djangotest',
    },
}
# 当添加、修改、删除数据时，自动生成倒排索引
HAYSTACK_SIGNAL_PROCESSOR = 'haystack.signals.RealtimeSignalProcessor'
# 搜索的每页大小（根据需求自行修改或添加）
HAYSTACK_SEARCH_RESULTS_PER_PAGE = 9
```

2. 注册haystack的app

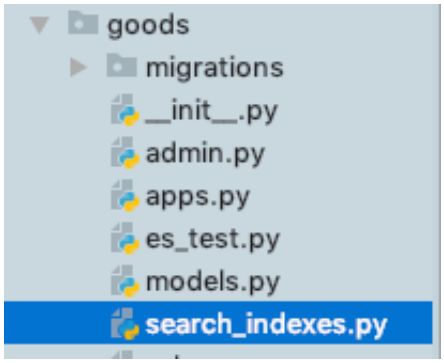
```
INSTALLED_APPS = [
    ...
    'haystack',
]
```

3. 在goods应用目录下，添加一个索引，编辑 goods/search_indexes.py

这里我们先介绍一个概念，SearchIndex 对象是 Haystack 确定应在搜索索引中放置哪些数据并处理数据流的方式。可以将它们视为与 Django Models 或 Forms 相似的对象，因为它们是基于字段处理/存储数据的。

要构建 SearchIndex，只需让我们的索引类继承 index.SearchIndex 和 index.Indexable，定义要存储数据的字段以及重写 get_model 和 index_queryset 方法。

由于达达商城的搜索功能是基于 SKU 进行检索的，所以我们需要在 goods 的 app 目录下创建 search_indexes.py 文件，search_indexes.py 文件名符合规范且最好不要修改。



如果想根据 某个 app 进行检索，则需要在该app目录下创建 search_indexes.py 文件。

接下来我们了解一下它的哪些字段创建索引，怎么指定。

每个 SearchIndex 都必须有且仅有一个 document = True 的字段。 它向 Haystack 和搜索引擎指示哪个字段是要进行搜索的主要字段。

注意：如果使用一个字段设置了document=True，则一般约定此字段名为text，这是在SearchIndex类里面一贯的命名，以防止后台混乱，当然名字你也可以随便改，不过不建议改（源码部分在后面展示）。

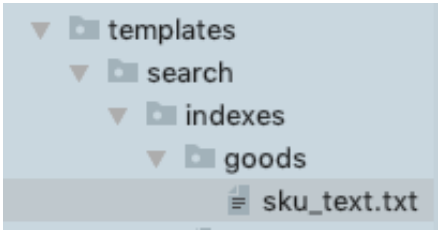
```
from haystack import indexes
# 1. 改成你自己的model
from .models import SKU

# 2. 类名为模型类的名称+Index，比如模型类为Type,则这里类名为TypeIndex
class SKUIndex(indexes.SearchIndex, indexes.Indexable):
    # 指明哪些字段产生索引，产生索引的字段，会作为前端检索查询的关键字；
    # document是指明text是使用的文档格式，产生字段的内容在文档中进行描述；
    # use_template是指明在模板中被声明需要产生索引；
    text = indexes.CharField(document=True, use_template=True)

    # 3. 返回的是你自己的model
    def get_model(self):
        """返回建立索引的模型类"""
```

```
        return SKU

    # 4. 修改return 可以修改返回查询集的内容，比如返回时，有什么条件限制
    def index_queryset(self, using=None):
        """返回要建立索引的数据查询集"""
        return self.get_model().objects.filter(is_launched=True)
```

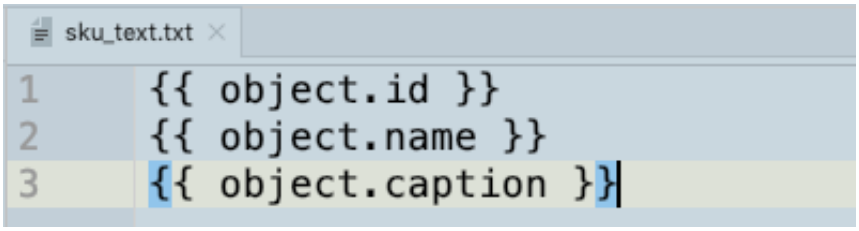
并且，haystack提供了 use_template = True 在 text 字段，这样就允许我们使用数据模板去建立搜索引擎索引的文件，说得通俗点就是索引里面需要存放一些什么东西，例如 SKU 的 name 字段，这样我们可以通过 name 内容来检索 SKU 数据了，举个例子，假如你搜索手提包，那么就可以检索出 name 中含有手提包的 SKU 了。

4. 创建数据模版

此外，我们在文本字段上提供 use_template = True。 这使得我们可以使用数据模板来构建搜索索引的文档。 您需要在模板目录中创建一个名为 search / indexes / myapp / model 类名(大小写随意，如果不是类名则 rebuild_index不会成功)_text.txt的新模板：



sku_text.txt的内容如下：



在这里我们可以通过 SKU 的id、name、caption 字段来进行搜索。

注意：如果在 search_indexes.py 文件中的 text 字段的 use_template 的值设置为 false，则不能用模版进行检索。

5. 修改 views.py

```
class GoodsSearchView(View):
    def post(self,request):
        """
        首页查询功能
        :param request:
        :return:
        """
        # 127.0.0.1:8000/v1/goods/search
        from dadashop.settings import HAYSTACK_SEARCH_RESULTS_PER_PAGE
        query = ''
```

```
        page_size = HAYSTACK_SEARCH_RESULTS_PER_PAGE
        results = EmptySearchQuerySet()
        # 为什么用'q'在下文介绍
        if request.POST.get('q'):
            # searchqueryset参数在下文介绍
            form = ModelSearchForm(request.POST, searchqueryset=None,
load_all=True)
            if form.is_valid():
                query = form.cleaned_data['q']
                results = form.search()

        paginator = Paginator(results, page_size)
        try:
            page = paginator.page(int(request.POST.get('page', 1)))
        except:
            result = {'code': 40200, 'data': '页数有误，小于0或者大于总页数'}
            return JsonResponse(result)

        # 记录查询信息
        context = {
            'form': form,
            'page': page,
            'paginator': paginator,
            'query': query,
        }

        sku_list = []
        # print(len(page.object_list))
        for result in page.object_list:
            sku = {
                'skuid': result.object.id,
                'name': result.object.name,
                'price': result.object.price,
            }
            sku_image_count =
SKUImage.objects.filter(sku_id=result.object.id).count()
            # 获取图片
            sku_image = str(result.object.default_image_url)
            sku['image'] = sku_image
            sku_list.append(sku)
        result = {"code": 200, "data": sku_list, 'paginator': {'pagesize':
page_size, 'total': len(results)}, 'base_url': PIC_URL}
        return JsonResponse(result)
```

注意：

1. 这里有一个注意的点就是为什么要使用 request.POST.get('q') ， 参数为什么是 q ， 可以是其他的参数么?

我们使用的 form 类型是 ModelSearchForm，这个类的父类是 SearchForm，而 results = form.search() 调用的search 方法恰好是父类的 search 方法，跟进 ModelSearchForm 源码查看：

```
# haystack/forms.py
class ModelSearchForm(SearchForm):
    def __init__(self, *args, **kwargs):
        super(ModelSearchForm, self).__init__(*args, **kwargs)
        self.fields['models'] =
forms.MultipleChoiceField(choices=model_choices(), required=False,
label=_('Search In'), widget=forms.CheckboxSelectMultiple)

    def get_models(self):
        """Return a list of the selected models."""
        search_models = []

        if self.is_valid():
            for model in self.cleaned_data['models']:
                search_models.append(haystack_get_model(*model.split('.')))

        return search_models

    def search(self):
        # 调用父类的 search 方法，跟进父类(SearchForm)看源码
        sqs = super(ModelSearchForm, self).search()
        return sqs.models(*self.get_models())
```

跟进查看 SearchForm 类的search方法是如何实现的：

```
  # haystack/forms.py
  def search(self):
        if not self.is_valid():
            return self.no_query_found()

        if not self.cleaned_data.get('q'):
            return self.no_query_found()
        # 直接从前端获取参数为 “q” 的值，所以不能获取其他参数的值，必须根据 “q”获取
        sqs = self.searchqueryset.auto_query(self.cleaned_data['q'])

        if self.load_all:
            sqs = sqs.load_all()

        return sqs
```

2. searchqueryset 参数?

```
form = ModelSearchForm(request.POST, searchqueryset=None, load_all=True)
```

首先进入 ModelSearchForm 源码：

```
# haystack/forms.py
class ModelSearchForm(SearchForm):
    def __init__(self, *args, **kwargs):
        super(ModelSearchForm, self).__init__(*args, **kwargs)
        self.fields['models'] =
forms.MultipleChoiceField(choices=model_choices(),  required=False,
label=_('Search In'), widget=forms.CheckboxSelectMultiple)
```

会发现在初始化 ModelSearchForm 时调用了父类的初始化方法，继续跟进：

```
# haystack/forms.py
class SearchForm(forms.Form):
    q = forms.CharField(required=False, label=_('Search'),
                        widget=forms.TextInput(attrs={'type': 'search'}))

    def __init__(self, *args, **kwargs):
        self.searchqueryset = kwargs.pop('searchqueryset', None)
        self.load_all = kwargs.pop('load_all', False)

        if self.searchqueryset is None:
            self.searchqueryset = SearchQuerySet()

        super(SearchForm, self).__init__(*args, **kwargs)
```

因为传递的 searchqueryset 的值为 None，所以会执行 self.searchqueryset = SearchQuerySet()，点进 SearchQuerySet() 继续观察：

```
# query.py
class SearchQuerySet(object):
    """
    Provides a way to specify search parameters and lazily load results.

    Supports chaining (a la QuerySet) to narrow the search.
    """
    def __init__(self, using=None, query=None):
        # ``_using`` should only ever be a value other than ``None`` if it's
        # been forced with the ``.using`` method.
        self._using = using
        self.query = None
        self._determine_backend()
......
......
```

根据关于参数 'q'的分析得知，最后调用的查询方法是 self.searchqueryset.auto_query(self.cleaned_data['q'])，所以在 SearchQuerySet类中跟进 auto_query 方法：

```
# query.py
def auto_query(self, query_string, fieldname='content'):
      """
      Performs a best guess constructing the search query.

      This method is somewhat naive but works well enough for the simple,
      common cases.
      """
      kwargs = {
          fieldname: AutoQuery(query_string)
      }
      return self.filter(**kwargs)
```

方法中的 query_string 就是 'q' 的值，即项目中首页搜索框的中的值，将其封装成一个 fieldname 进行查询，调用 self.filter(**kwargs) 进行过滤查询。

3. 为什么 text 字段是规范?

```
# indexes.py
class BasicSearchIndex(SearchIndex):
    text = CharField(document=True, use_template=True)
```

6. 生成索引

在生成索引前同步一下数据库：

```
python3 manage.py makemigrations
python3 manage.py migrate
```

需要手动生成一次索引，之后会跟据 settings.py 中的配置自动生成倒排索引：

注意：在执行前，要确保之前所有步骤已经完成，否则会失败

```
python3 manage.py rebuild_index
```

7. 启动项目并测试

```
python3 manage.py runserver 0.0.0.0:8000
```