elasticsearch-java-api-cn Documentation

Release 6.2

jackiehff

Feb 12, 2018

Contents

CHAPTER 1

前言

本节描述了 Elasticsearch 提供的 Java API。Elasticsearch 的所有操作都是使用 客户端 对象来执行的。所有操作本质上都是完全异步的 (要么接受一个监听器, 要么返回一个 Future 对象)。

另外, 客户端上的操作可以累积起来并以 批量 的方式执行。

注意，所有的 API 都是通过 Java API 暴露的(实际上, Elasticsearch 内部是使用 Java API 来执行操作的)。

Warning: 我们计划在 Elasticsearch 7.0 版本中关闭 TransportClient 并且在 8.0 版本中完全删除它。你应该使用 Java 高级 REST 客户端作为替代, 它可以执行 HTTP 请求, 而不是序列化 Java 请求。迁移指南 中描述了迁移所需的所有步骤。

Java 高级 REST 客户端目前已经支持常用的 API, 但是仍然有很多需要添加进来。你可以对 Java 高级 REST 客户端完整性 这个问题添加评论来告诉我们你的应用程序需要哪些缺少的API，从而帮助我们确定优先级。

任何缺少的 API 现在都可以通过使用具有 JSON 请求和响应体的 低级 Java REST 客户端 来实现。

CHAPTER 2

Java 文档

传输客户端的 Java 文档参见 https://artifacts.elastic.co/javadoc/org/elasticsearch/client/transport/6.2.1/index.html 。

CHAPTER 3

Maven仓库

Elasticsearch 托管在 Maven中央仓库 中。

例如, 你可以在 *pom.xml* 文件中声明最新版本的依赖:

```
<dependency>
    <groupId>org.elasticsearch.client</groupId>
    <artifactId>transport</artifactId>
    <version>6.2.1</version>
</dependency>
```

3.1 Lucene快照仓库

任何主版本的第一个发布版本(比如beta)都可能构建于一个 Lucene 快照版本之上。在这种情况下，你无法解析客户端的 Lucene 依赖。

比如, 你想使用 6.0.0-beta1 这个版本, 它依赖于 Lucene 7.0.0-snapshot-00142c9 版本, 你就必须定义下面这个仓库。

如果使用 Maven，定义如下:

```
<repository>
    <id>elastic-lucene-snapshots</id>
    <name>Elastic Lucene Snapshots</name>
    <url>http://s3.amazonaws.com/download.elasticsearch.org/lucenesnapshots/00142c9</
↪url>
    <releases><enabled>true</enabled></releases>
    <snapshots><enabled>false</enabled></snapshots>
</repository>
```

如果使用 Gradle， 定义如下:

```
maven {
    url 'http://s3.amazonaws.com/download.elasticsearch.org/lucenesnapshots/00142c9'
}
```

3.2 Log4j 2 日志记录器

你还需要引入 Log4j 2 依赖:

```
<dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-core</artifactId>
    <version>2.9.1</version>
</dependency>
```

并且还需要在类路径中提供一个 Log4j 2 配置文件。例如, 你可以在 *src/main/resources* 这个工程目录下添加一个 *log4j2.properties* 文件, 内容如下:

```
appender.console.type = Console
appender.console.name = console
appender.console.layout.type = PatternLayout

rootLogger.level = info
rootLogger.appenderRef.console.ref = console
```

3.3 使用其它日志记录器

如果你想要使用 Log4j 2 之外的日志记录器, 可以使用 SLF4J 桥接器来实现:

```
<dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-to-slf4j</artifactId>
    <version>2.9.1</version>
</dependency>

<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-api</artifactId>
    <version>1.7.24</version>
</dependency>
```

SLF4J用户手册 这篇文章中列举了所有可使用的日志实现。选择你最喜欢的日志记录器并将其作为 Maven 依赖添加进来。举个例子, 我们将使用 *slf4j-simple* 这个日志记录器:

```
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-simple</artifactId>
    <version>1.7.21</version>
</dependency>
```

CHAPTER 4

解决JAR包依赖冲突

如果你想在 Java 应用程序中使用 Elasticsearch, 你可能需要处理像 Guava 和 Joda 这样的第三方依赖之间的版本冲突。例如 Elasticsearch 可能使用的是 Joda 2.8, 而你的代码中使用的是 Joda 2.1。

你有两个选择:

- 最简单的解决方案就是升级。较新的模块版本通常会解决一些老版本的 bug, 如果你的版本落后的比较远, 那么以后升级就会更加困难。当然, 有可能你使用的一个第三方依赖, 它反过来依赖于一个已过时版本的包, 这样也会阻止你的升级。
- 第二个解决方案就是重新安置冲突的依赖, 并且使用自己的应用程序或者是 Elasticsearch 以及 Elasticsearch 客户端所依赖的任何插件来隐藏它们。

"隐藏或不隐藏"这篇博客文章 描述了这样做的所有步骤。

CHAPTER 5

将依赖嵌入jar包

如果你想创建一个包含你应用程序代码和所有依赖的 jar 包, 请不要使用 *maven-assembly-plugin* 插件, 因为它无法处理 Lucene jar 所要求的 *META-INF/services* 结构。

相反, 你可以使用 *maven-shade-plugin* 插件并使用如下配置:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-shade-plugin</artifactId>
    <version>2.4.1</version>
    <executions>
        <execution>
            <phase>package</phase>
            <goals><goal>shade</goal></goals>
            <configuration>
                <transformers>
                    <transformer implementation="org.apache.maven.plugins.shade.
↪resource.ServicesResourceTransformer"/>
                </transformers>
            </configuration>
        </execution>
    </executions>
</plugin>
```

注意: 如果你想在运行 *java -jar yourjar.jar* 命令时自动调用一个 *main* 类, 你只需要将其添加到 *transformers* 元素中:

```
<transformer implementation="org.apache.maven.plugins.shade.resource.
↪ManifestResourceTransformer">
    <mainClass>org.elasticsearch.demo.Generate</mainClass>
</transformer>
```

CHAPTER 6

客户端

你可以以多种方式使用 *Java* 客户端:

- 在现有集群上执行标准的 索引 *API*, 获取 *API*, 删除 *API* 和 搜索*API* 操作
- 在运行着的集群上执行管理任务

获取一个 Elasticsearch 客户端很简单, 最常见的方式就是创建一个可以连接到集群的 传输客户端 对象。

Important: 客户端和集群中的节点必须使用相同的主版本 (例如, `2.x` 或 `5.x`)。客户端可以连接到使用不同次要版本的集群 (例如 `2.3.x`), 但是这样的话可能就没办法支持新的功能。理想情况下, 客户端和集群应该使用完全相同的版本。

Warning: 我们计划在 Elasticsearch 7.0 版本中关闭 TransportClient 并且在 8.0 版本中完全删除它。你应该使用 Java 高级 REST 客户端作为替代, 它可以执行 HTTP 请求, 而不是序列化 Java 请求。迁移指南 中描述了迁移所需的所有步骤。

Java 高级 REST 客户端目前已经支持常用的 API, 但是仍然有很多需要添加进来。你可以对 Java 高级 REST 客户端完整性 这个问题添加评论来告诉我们你的应用程序需要哪些缺少的API，从而帮助我们确定优先级。

任何缺少的 API 现在都可以通过使用具有 JSON 请求和响应体的 低级 Java REST 客户端 来实现。

6.1 传输客户端

`TransportClient` 使用传输模块远程连接到 Elasticsearch 集群。它不加入集群, 而只是简单地获取一个或多个初始传输地址并且针对每个动作以轮询的方式与传输地址进行通信(尽管大多数动作可能是"两跳"操作)。

```
// on startup
```

```
TransportClient client = new PreBuiltTransportClient(Settings.EMPTY)
    .addTransportAddress(new TransportAddress(InetAddress.getByName("host1"), 9300))
    .addTransportAddress(new TransportAddress(InetAddress.getByName("host2"), 9300));

// on shutdown

client.close();
```

注意: 如果你使用的集群名称不是 “elasticsearch”, 那么你需要设置一个不同的集群名称:

```
Settings settings = Settings.builder()
    .put("cluster.name", "myClusterName").build();
TransportClient client = new PreBuiltTransportClient(settings);
//Add transport addresses and do something with the client...
```

传输客户端附带了一个集群嗅探特性, 使它能够动态地添加新的主机并删除旧的主机。当启用嗅探时, 传输客户端将连接到其内部节点列表中的节点, 这个节点列表是通过调用 `addTransportAddress` 方法来构建的。之后, 客户端将会在这些节点上调用内部的集群状态 API 来发现可用的数据节点。客户端的内部节点列表只会被那些数据节点替换。默认情况下, 该列表每五秒钟刷新一次。请注意, 嗅探器连接到的 IP 地址是在那些节点的 Elasticsearch 配置中被声明为 ‘publish’ 的地址。

请记住, 传输客户端的内部节点列表可能不包括它连接到的原始节点, 如果那个节点不是一个数据节点的话。例如，如果你最开始连接到的是一个主节点, 在经过嗅探之后, 后续的请求没有去到该主节点, 而是到达任意的数据节点。传输客户端排除非数据节点的原因是为了避免将搜索流量发送到仅作为主节点的节点。

如果要启用嗅探, 需要将 `client.transport.sniff` 的值设置为 `true`:

```
Settings settings = Settings.settingsBuilder()
    .put("client.transport.sniff", true).build();
TransportClient client = new PreBuiltTransportClient(settings);
```

其它传输客户端级别的设置包括:

参数	描述
`client.transport.ignore_cluster_name`	设置为 `true` 可以忽略已连接节点的集群名称校验。(从 0.19.4 版本开始)
`client.transport.ping_timeout`	节点 ping 响应的等待时长, 默认值是 `5s` 。
`client.transport.nodes_sampler_interval`	对列出和连接的节点进行抽样或ping操作的频率, 默认值是 `5s`。

6.2 客户端连接到协调节点

你可以在本地启动一个 协调节点 , 然后在你的应用程序中简单地创建一个 TransportClient 对象并连接到这个协调节点。

这样, 协调节点能够加载任何你所需要的插件(例如发现插件)。

CHAPTER 7

文档API

本节描述了以下CRUD API:

单文档API

- 索引 *API*
- 获取 *API*
- 删除 *API*
- 根据查询删除*API*
- 更新*API*

多文档API

- *Multi Get API*
- *Bulk API*

Note: 所有 CRUD API 都是单索引 API。`index` 参数接受单个索引名称或者指向单个索引的一个别名 `alias`。

7.1 索引 API

索引 API 允许用户将一个类型化的 JSON 文档索引到一个特定的索引中, 并且使它可以被搜索到。

7.1.1 生成 JSON 文档

生成一个 JSON 文档有以下几种不同的方式:

- 手动(即自己动手)使用本地 *byte[]* 或者作为 *String*

- 使用 *Map*, 它会自动转换成与其等价的 JSON
- 使用 Jackson 这样的第三方类库来序列化你的 Java Bean
- 使用内置的帮助类 XContentFactory.jsonBuilder()

在 Elasticsearch 内部, 每种类型都被转换成 *byte[]*(因此字符串也会被转换成一个 *byte[]*)。因此, 可以直接使用已经是这种形式的对象。*jsonBuilder* 是高度优化过的 JSON 生成器, 它可以直接构造一个 *byte[]*。

自己动手

这里没有什么真正很难的地方, 但是需要注意的是, 你需要根据 日期格式 来编码日期。

```
String json = "{" +
        "\"user\":\"kimchy\"," +
        "\"postDate\":\"2013-01-30\"," +
        "\"message\":\"trying out Elasticsearch\"" +
    "}";
```

使用 Map

Map 是一个键值对集合, 它表示一个 JSON 结构:

```
Map<String, Object> json = new HashMap<String, Object>();
json.put("user","kimchy");
json.put("postDate",new Date());
json.put("message","trying out Elasticsearch");
```

序列化你的Bean

可以使用 Jackson 将你的 Java Bean 序列化成 JSON 格式。你需要在项目中添加 Jackson Databind 依赖, 接着你就可以使用 *ObjectMapper* 来序列化你的 Java Bean:

```
import com.fasterxml.jackson.databind.*;

// instance a json mapper
ObjectMapper mapper = new ObjectMapper(); // create once, reuse

// generate json
byte[] json = mapper.writeValueAsBytes(yourbeaninstance);
```

使用 Elasticsearch 帮助类

Elasticsearch 提供了内置的帮助类来生成 JSON 内容。

```
import static org.elasticsearch.common.xcontent.XContentFactory.*;

XContentBuilder builder = jsonBuilder()
    .startObject()
        .field("user", "kimchy")
        .field("postDate", new Date())
        .field("message", "trying out Elasticsearch")
    .endObject()
```

请注意, 你还可以使用 *startArray(String)* 和 *endArray()* 方法来添加数组。顺便说一下, *field* 方法可以接受许多对象类型, 你可以直接传递数字, 日期, 甚至是其它 XContentBuilder 对象。

如果你要查看生成的 JSON 内容, 你可以使用 *string()* 方法.

```
String json = builder.string();
```

7.1.2 索引文档

下面的例子将一个 JSON 文档索引到一个名为 twitter 的索引中, 文档对应的类型为 tweet, 文档 ID 为 1:

```
import static org.elasticsearch.common.xcontent.XContentFactory.*;

IndexResponse response = client.prepareIndex("twitter", "tweet", "1")
        .setSource(jsonBuilder()
                    .startObject()
                        .field("user", "kimchy")
                        .field("postDate", new Date())
                        .field("message", "trying out Elasticsearch")
                    .endObject()
                  )
        .get();
```

请注意, 你也可以将文档作为 JSON 字符串进行索引并且不需要指定文档 ID:

```
String json = "{" +
        "\"user\":\"kimchy\"," +
        "\"postDate\":\"2013-01-30\"," +
        "\"message\":\"trying out Elasticsearch\"" +
    "}";

IndexResponse response = client.prepareIndex("twitter", "tweet")
        .setSource(json)
        .get();
```

IndexResponse 对象返回给你所有信息:

```
// Index name
String _index = response.getIndex();
// Type name
String _type = response.getType();
// Document ID (generated or not)
String _id = response.getId();
// Version (if it's the first time you index this document, you will get: 1)
long _version = response.getVersion();
// status has stored current instance statement.
RestStatus status = response.status();
```

有关索引操作的更多信息, 请查看 REST 索引 文档。

7.2 获取 API

获取 API 允许用户根据文档 ID 获取索引中一个类型化的 JSON 文档。下面的例子从 twitter 索引中获取一个 JSON 文档, 文档对应的类型是 tweet, 文档 ID 为 1:

```
GetResponse response = client.prepareGet("twitter", "tweet", "1").get();
```

更多有关获取操作的信息, 请查看 REST 获取 文档。

7.3 删除 API

删除 API 允许用户根据文档 ID 删除特定索引中一个类型化的 JSON 文档。下面的示例从索引 twitter 中删除类型 tweet、文档 ID 为 1 的 JSON 文档:

```
DeleteResponse response = client.prepareDelete("twitter", "tweet", "1").get();
```

更多关于删除操作的信息, 请查看 删除 API 文档。

7.4 根据查询删除API

根据查询删除 API 允许用户删除根据查询结果得到文档集合:

```
BulkIndexByScrollResponse response =
    DeleteByQueryAction.INSTANCE.newRequestBuilder(client)
        .filter(QueryBuilders.matchQuery("gender", "male")) <1>
        .source("persons")                                  <2>
        .get();                                             <3>

long deleted = response.getDeleted();                       <4>
```

<1> 查询 <2> 索引 <3> 执行操作 <4> 删除的文档数量

因为它是一个耗时比较长的操作, 所以如果你想要异步执行, 你可以调用 *execute* 方法来替代 *get* 方法并提供一个像下面这样的监听器:

```
DeleteByQueryAction.INSTANCE.newRequestBuilder(client)
    .filter(QueryBuilders.matchQuery("gender", "male"))                 <1>
    .source("persons")                                                  <2>
    .execute(new ActionListener<BulkIndexByScrollResponse>() {          <3>
        @Override
        public void onResponse(BulkIndexByScrollResponse response) {
            long deleted = response.getDeleted();                       <4>
        }
        @Override
        public void onFailure(Exception e) {
            // Handle the exception
        }
    });
```

<1> 查询 <2> 索引 <3> 监听器 <4> 删除的文档数量

7.5 更新API

你可以创建一个 *UpdateRequest* 对象并将其发送给客户端:

```
UpdateRequest updateRequest = new UpdateRequest();
updateRequest.index("index");
updateRequest.type("type");
updateRequest.id("1");
updateRequest.doc(jsonBuilder()
        .startObject()
            .field("gender", "male")
        .endObject());
client.update(updateRequest).get();
```

或者你也可以使用 *prepareUpdate()* 方法:

```
client.prepareUpdate("ttl", "doc", "1")
        .setScript(new Script("ctx._source.gender = \"male\"" <1> , ScriptType.INLINE,
↪ null, null))
        .get();

client.prepareUpdate("ttl", "doc", "1")
        .setDoc(jsonBuilder()                <2>
            .startObject()
                .field("gender", "male")
            .endObject())
        .get();
```

<1> 你的脚本. 它也可以是一个本地存储的脚本名称。在这种情况下, 你需要使用 *ScriptType.FILE*。 <2> 将要合并到现有文档的一个文档。

请注意, 你不能同时提供 *script* 和 *doc*.

7.5.1 通过脚本更新

更新 API 允许根据用户所提供的脚本来更新文档:

```
UpdateRequest updateRequest = new UpdateRequest("ttl", "doc", "1")
        .script(new Script("ctx._source.gender = \"male\""));
client.update(updateRequest).get();
```

7.5.2 通过合并文档进行更新

更新 API 还支持传递部分文档, 它将合并到现有的文档中(简单递归合并, 对象的内部合并, 替换核心的键值对和数组)。例如:

```
UpdateRequest updateRequest = new UpdateRequest("index", "type", "1")
        .doc(jsonBuilder()
            .startObject()
                .field("gender", "male")
            .endObject());
client.update(updateRequest).get();
```

7.5.3 更新插入

Elasticsearch 还支持更新插入。如果文档不存在, *upsert* 元素将用于索引新鲜文档:

```
IndexRequest indexRequest = new IndexRequest("index", "type", "1")
        .source(jsonBuilder()
            .startObject()
                .field("name", "Joe Smith")
                .field("gender", "male")
            .endObject());
UpdateRequest updateRequest = new UpdateRequest("index", "type", "1")
        .doc(jsonBuilder()
            .startObject()
                .field("gender", "male")
            .endObject())
        .upsert(indexRequest);              <1>
client.update(updateRequest).get();
```

<1> 如果文档不存在, *indexRequest* 中的文档将会被添加进来

如果文档 *index/type/1* 已经存在, 那么在该操作之后我们将会得到一个像下面这样的文档:

```
{
    "name"   : "Joe Dalton",
    "gender": "male"        <1>
}
```

<1> 该字段由更新请求添加

如果它不存在, 那么我们将会得到一个新的文档:

```
{
    "name" : "Joe Smith",
    "gender": "male"
}
```

7.6 Multi Get API

Multi Get API 允许用户根据文档的 *index*, *type* 以及 *id* 来获取文档列表:

```
MultiGetResponse multiGetItemResponses = client.prepareMultiGet()
    .add("twitter", "tweet", "1")           <1>
    .add("twitter", "tweet", "2", "3", "4") <2>
    .add("another", "type", "foo")          <3>
    .get();

for (MultiGetItemResponse itemResponse : multiGetItemResponses) { <4>
    GetResponse response = itemResponse.getResponse();
    if (response.isExists()) {                      <5>
        String json = response.getSourceAsString(); <6>
    }
}
```

<1> 根据单个文档 ID 进行查询 <2> 对同样的索引/类型根据文档 ID 列表进行查询 <3> 还可以从另一个索引上查询 <4> 遍历结果集 <5> 检查文档是否存在 <6> 访问 *_source* 字段

更多有关 Multi Get 操作的信息, 请查看 REST multi get 文档.

7.7 Bulk API

Bulk API 允许用户在单个请求中索引和删除多个文档。下面是一个使用示例:

```
import static org.elasticsearch.common.xcontent.XContentFactory.*;

BulkRequestBuilder bulkRequest = client.prepareBulk();

// either use client#prepare, or use Requests# to directly build index/delete requests
bulkRequest.add(client.prepareIndex("twitter", "tweet", "1")
        .setSource(jsonBuilder()
                    .startObject()
                        .field("user", "kimchy")
                        .field("postDate", new Date())
                        .field("message", "trying out Elasticsearch")
                    .endObject()
                  )
        );

bulkRequest.add(client.prepareIndex("twitter", "tweet", "2")
        .setSource(jsonBuilder()
                    .startObject()
                        .field("user", "kimchy")
                        .field("postDate", new Date())
                        .field("message", "another post")
                    .endObject()
                  )
        );

BulkResponse bulkResponse = bulkRequest.get();
if (bulkResponse.hasFailures()) {
    // process failures by iterating through each bulk response item
}
```

7.8 使用批处理器

BulkProcessor 类提供了一个简单的接口, 它可以根据请求数量或在指定的时间段后自动地刷新批量操作。

要使用它的话, 首先要创建一个 *BulkProcessor* 实例:

```
import org.elasticsearch.action.bulk.BackoffPolicy;
import org.elasticsearch.action.bulk.BulkProcessor;
import org.elasticsearch.common.unit.ByteSizeUnit;
import org.elasticsearch.common.unit.ByteSizeValue;
import org.elasticsearch.common.unit.TimeValue;

BulkProcessor bulkProcessor = BulkProcessor.builder(
        client,  <1>
        new BulkProcessor.Listener() {
            @Override
            public void beforeBulk(long executionId,
                                   BulkRequest request) { ... } <2>

            @Override
            public void afterBulk(long executionId,
```

```
                                BulkRequest request,
                                BulkResponse response) { ... } <3>

            @Override
            public void afterBulk(long executionId,
                                BulkRequest request,
                                Throwable failure) { ... } <4>
        })
        .setBulkActions(10000) <5>
        .setBulkSize(new ByteSizeValue(5, ByteSizeUnit.MB)) <6>
        .setFlushInterval(TimeValue.timeValueSeconds(5)) <7>
        .setConcurrentRequests(1) <8>
        .setBackoffPolicy(
            BackoffPolicy.exponentialBackoff(TimeValue.timeValueMillis(100), 3)) <9>
        .build();
```

<1> 添加 Elasticsearch 客户端 <2> 该方法在批量操作执行前调用。例如, 你可以使用 *request.numberOfActions()* 方法查看 numberOfActions <3> 该方法在批量操作执行后调用。例如, 你可以使用 *response.hasFailures()* 方法检查是否有失败的请求 <4> 该方法在批量失败并抛出 *Throwable* 时调用 <5> 每 10 000 个请求执行一次批量操作 <6> 每 5mb 刷新一次批量操作 <7> 不管请求数量多少, 每 5s 刷新一次批量操作 <8> 设置并发请求数量。值为 0 的话意味着一次只允许执行一个请求。值为1的话意味着在累积新的批量请求时值允许执行1个并发请求。 <9> 设置自定义退避策略, 该策略最初将等待100毫秒, 按指数增加并且最多重试三次。当一个或多个批量项目请求因为 *EsRejectedExecutionException* 异常而失败, 这通常意味着没有足够的计算资源来处理这个请求, 一般会尝试重试。要禁用退避, 可以传递 *BackoffPolicy.noBackoff()*。

默认情况下, *BulkProcessor* 会做以下事情:

- 设置 bulkActions 的值为 *1000*
- 设置 bulkSize 的值为 *5mb*
- 不设置 flushInterval
- 设置 concurrentRequests 的值为 1, 意味着异步执行刷新操作。
- 设置 backoffPolicy 的值为重试 8 次以及 50ms 启动延迟的一个指数退避, 总的等待时间大概是 5.1s。

7.8.1 添加请求

接着你可以简单地将请求添加到 *BulkProcessor*:

```
bulkProcessor.add(new IndexRequest("twitter", "tweet", "1").source(/* your doc here */
↪));
bulkProcessor.add(new DeleteRequest("twitter", "tweet", "2"));
```

7.8.2 关闭批处理器

当所有的文档都被加载到 *BulkProcessor* 后, 可以使用 *awaitClose* 或 *close* 方法来关闭它:

```
bulkProcessor.awaitClose(10, TimeUnit.MINUTES);
```

或

```
bulkProcessor.close();
```

如果他们是通过设置 *flushInterval* 预先安排的, 那么两种方法都会清除所有剩余的文档并禁用所有其它事先安排的刷新。如果启用了并发请求, 那么 *awaitClose* 方法将在指定的超时时间内等待所有的批量请求执行完成后返回 *true*, 如果在指定的等待时间之后所有批量请求还未完成则返回 *false*。*close* 方法会立即退出而不会等待任何还未完成的批量请求。

7.8.3 在测试中使用批处理器

如果你正在使用 Elasticsearch 运行测试并且使用 *BulkProcessor* 来填充你的数据集, 那么你最好将并发请求的数量设置为 *0*, 这样的话批量刷新操作将会以同步的方式执行:

```
BulkProcessor bulkProcessor = BulkProcessor.builder(client, new BulkProcessor.
↪Listener() { /* Listener methods */ })
        .setBulkActions(10000)
        .setConcurrentRequests(0)
        .build();

// Add your requests
bulkProcessor.add(/* Your requests */);

// Flush any remaining requests
bulkProcessor.flush();

// Or close the bulkProcessor if you don't need it anymore
bulkProcessor.close();

// Refresh your indices
client.admin().indices().prepareRefresh().get();

// Now you can start searching!
client.prepareSearch().get();
```

CHAPTER 8

搜索API

搜索 API 允许用户执行搜索查询并返回匹配查询的搜索命中结果。它可以跨一个或多个索引以及跨一个或多个类型执行。可以使用 查询*DSL* 提供的查询功能。搜索请求的正文是使用 *SearchSourceBuilder* 对象来构建的。下面是一个代码示例:

```
import org.elasticsearch.action.search.SearchResponse;
import org.elasticsearch.action.search.SearchType;
import org.elasticsearch.index.query.QueryBuilders.*;
```

```
SearchResponse response = client.prepareSearch("index1", "index2")
        .setTypes("type1", "type2")
        .setSearchType(SearchType.DFS_QUERY_THEN_FETCH)
        .setQuery(QueryBuilders.termQuery("multi", "test"))                 // Query
        .setPostFilter(QueryBuilders.rangeQuery("age").from(12).to(18))     // Filter
        .setFrom(0).setSize(60).setExplain(true)
        .get();
```

请注意, 所有的参数都是可选的。下面是可以写出来的最短的搜索调用代码:

```
// MatchAll on the whole cluster with all default options
SearchResponse response = client.prepareSearch().get();
```

Note: 尽管 Java API 定义了额外的搜索类型 QUERY_AND_FETCH 和 DFS_QUERY_AND_FETCH, 这些模式也都经过了内部优化, API的使用者不需要显示地指定搜索类型.

有关搜索操作的更多信息, 请查看 REST 搜索 文档。

8.1 在 Java 中使用滚动

首先请阅读 滚动文档 !

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

QueryBuilder qb = termQuery("multi", "test");

SearchResponse scrollResp = client.prepareSearch(test)
        .addSort(FieldSortBuilder.DOC_FIELD_NAME, SortOrder.ASC)
        .setScroll(new TimeValue(60000))
        .setQuery(qb)
        .setSize(100).get(); //max of 100 hits will be returned for each scroll
//Scroll until no hits are returned
do {
    for (SearchHit hit : scrollResp.getHits().getHits()) {
        //Handle the hit...
    }

    scrollResp = client.prepareSearchScroll(scrollResp.getScrollId()).setScroll(new␣
↪TimeValue(60000)).execute().actionGet();
} while(scrollResp.getHits().getHits().length != 0); // Zero hits mark the end of the␣
↪scroll and the while loop.
```

8.2 MultiSearch API

参见 Multi Search API 文档。

```
SearchRequestBuilder srb1 = client
    .prepareSearch().setQuery(QueryBuilders.queryStringQuery("elasticsearch")).
↪setSize(1);
SearchRequestBuilder srb2 = client
    .prepareSearch().setQuery(QueryBuilders.matchQuery("name", "kimchy")).setSize(1);

MultiSearchResponse sr = client.prepareMultiSearch()
        .add(srb1)
        .add(srb2)
        .get();

// You will get all individual responses from MultiSearchResponse#getResponses()
long nbHits = 0;
for (MultiSearchResponse.Item item : sr.getResponses()) {
    SearchResponse response = item.getResponse();
    nbHits += response.getHits().getTotalHits();
}
```

8.3 使用聚合

下面的代码展示了如何在搜索中添加两个聚合操作:

```
SearchResponse sr = client.prepareSearch()
    .setQuery(QueryBuilders.matchAllQuery())
    .addAggregation(
            AggregationBuilders.terms("agg1").field("field")
    )
    .addAggregation(
```

```
            AggregationBuilders.dateHistogram("agg2")
                    .field("birth")
                    .dateHistogramInterval(DateHistogramInterval.YEAR)
    )
    .get();

// Get your facet results
Terms agg1 = sr.getAggregations().get("agg1");
DateHistogram agg2 = sr.getAggregations().get("agg2");
```

详情参见 <<java-aggs,Java 聚合 API>> 文档。

8.4 Terminate After

当到达每个分片要收集文档的最大数量时, 查询执行操作将提前终止。如果设置了这个数量, 你可以在 *SearchResponse* 对象上调用 *isTerminatedEarly()* 方法来检查操作是否提前终止:

```
SearchResponse sr = client.prepareSearch(INDEX)
    .setTerminateAfter(1000)    <1>
    .get();

if (sr.isTerminatedEarly()) {
    // We finished early
}
```

<1> 到达1000个文档后结束

8.5 搜索模板

参见 搜索模板 文档。

将你的模板参数定义成一个 *Map<String,Object>*:

```
Map<String, Object> template_params = new HashMap<>();
template_params.put("param_gender", "male");
```

你可以使用存储在 *config/scripts* 中的搜索模板。例如, 如果你有一个名为 *config/scripts/template_gender.mustache* 的文件, 内容如下:

```
{
    "template" : {
        "query" : {
            "match" : {
                "gender" : "{{param_gender}}"
            }
        }
    }
}
```

创建你的搜索模板请求:

```
SearchResponse sr = new SearchTemplateRequestBuilder(client)
    .setScript("template_gender")                <1>
```

```
    .setScriptType(ScriptService.ScriptType.FILE) <2>
    .setScriptParams(template_params)             <3>
    .setRequest(new SearchRequest())              <4>
    .get()                                        <5>
    .getResponse();                               <6>
```

<1> 模板名称 <2> 存储于磁盘上 *gender_template.mustache* 文件中的模板 <3> 参数 <4> 设置执行上下文(即在这里定义索引名称) <5> 执行并获取模板响应 <6> 从模板响应中获取搜索本身的响应

你还可以将模板存储在集群状态中:

```
client.admin().cluster().preparePutStoredScript()
    .setScriptLang("mustache")
    .setId("template_gender")
    .setSource(new BytesArray(
        "{\n" +
        "    \"template\" : {\n" +
        "        \"query\" : {\n" +
        "            \"match\" : {\n" +
        "                \"gender\" : \"{{param_gender}}\"\n" +
        "            }\n" +
        "        }\n" +
        "    }\n" +
        "}")).get();
```

要执行一个已存储的模板, 可以使用 *ScriptType.STORED*:

```
SearchResponse sr = new SearchTemplateRequestBuilder(client)
        .setScript("template_gender")                       <1>
        .setScriptType(ScriptType.STORED)     <2>
        .setScriptParams(template_params)                   <3>
        .setRequest(new SearchRequest())                    <4>
        .get()                                              <5>
        .getResponse();                                     <6>
```

<1> 模板名称 <2> 存储在集群状态中的模板 <3> 参数 <4> 设置执行上下文(即在这里定义索引名称) <5> 执行并获取模板响应 <6> 从模板响应中获取搜索本身的响应

你还可以执行内联模板:

```
sr = new SearchTemplateRequestBuilder(client)
        .setScript("{\n" +                                  <1>
                "        \"query\" : {\n" +
                "            \"match\" : {\n" +
                "                \"gender\" : \"{{param_gender}}\"\n" +
                "            }\n" +
                "        }\n" +
                "}")
        .setScriptType(ScriptType.INLINE)    <2>
        .setScriptParams(template_params)                   <3>
        .setRequest(new SearchRequest())                    <4>
        .get()                                              <5>
        .getResponse();                                     <6>
```

<1> 模板名称 <2> 内联传递的模板 <3> 参数 <4> 设置执行上下文(即在这里定义索引名称) <5> 执行并获取模板响应 <6> 从模板响应中获取搜索本身的响应

CHAPTER 9

聚合

Elasticsearch 提供了一套完整的 Java API 来使用聚合. 参见 聚合指南 。

使用聚合构造器工厂(`AggregationBuilders`)并且将查询时你想计算的每个聚合操作都添加到你的搜索请求中:

```
SearchResponse sr = node.client().prepareSearch()
    .setQuery( /* your query */ )
    .addAggregation( /* add an aggregation */ )
    .execute().actionGet();
```

请注意, 你可以添加多个聚合. 详情参见 Java搜索API 。

要构建一个聚合请求, 可以使用 `AggregationBuilders` 帮助类。只需要在你的类中引入它:

```
import org.elasticsearch.search.aggregations.AggregationBuilders;
```

9.1 结构化聚合

正如 聚合指南 中的解释, 你可以在聚合内部定义一个子聚合。

聚合可以是度量聚合或桶聚合.

例如, 这里由一个由以下三种聚合组成的一个3级聚合:

- 词项聚合(bucket)
- 日期直方图聚合(bucket)
- 平均聚合(metric)

```
SearchResponse sr = node.client().prepareSearch()
.addAggregation(
    AggregationBuilders.terms("by_country").field("country")
    .subAggregation(AggregationBuilders.dateHistogram("by_year")
        .field("dateOfBirth")
```

```
        .dateHistogramInterval(DateHistogramInterval.YEAR)
        .subAggregation(AggregationBuilders.avg("avg_children").field("children"))
    )
)
.execute().actionGet();
```

9.2 Metrics聚合

9.2.1 最小值聚合

下面展示了如何使用 Java API 进行 最小值聚合 。

创建聚合请求

下面是一个创建聚合请求的代码示例:

```
MinAggregationBuilder aggregation =
        AggregationBuilders
                .min("agg")
                .field("height");
```

使用聚合响应

引入聚合定义类:

```
import org.elasticsearch.search.aggregations.metrics.min.Min;
```

```
// sr is here your SearchResponse object
Min agg = sr.getAggregations().get("agg");
double value = agg.getValue();
```

9.2.2 最大值聚合

下面展示了如何使用 Java API 进行 最大值聚合 。

创建聚合请求

下面是一个创建聚合请求的代码示例:

```
MaxAggregationBuilder aggregation =
        AggregationBuilders
                .max("agg")
                .field("height");
```

使用聚合响应

引入聚合定义类:

```
import org.elasticsearch.search.aggregations.metrics.max.Max;
```

```
// sr is here your SearchResponse object
Max agg = sr.getAggregations().get("agg");
double value = agg.getValue();
```

9.2.3 求和聚合

下面展示了如何使用 Java API 进行 求和聚合 。

创建聚合请求

下面是一个创建聚合请求的代码示例:

```
SumAggregationBuilder aggregation =
        AggregationBuilders
                .sum("agg")
                .field("height");
```

使用聚合响应

引入聚合定义类:

```
import org.elasticsearch.search.aggregations.metrics.sum.Sum;
```

```
// sr is here your SearchResponse object
Sum agg = sr.getAggregations().get("agg");
double value = agg.getValue();
```

9.2.4 平均值聚合

下面展示了如何使用 Java API 进行 平均值聚合 。

创建聚合请求

下面是一个创建聚合请求的代码示例:

```
AvgAggregationBuilder aggregation =
        AggregationBuilders
                .avg("agg")
                .field("height");
```

使用聚合响应

引入聚合定义类:

```
import org.elasticsearch.search.aggregations.metrics.avg.Avg;
```

```
// sr is here your SearchResponse object
Avg agg = sr.getAggregations().get("agg");
double value = agg.getValue();
```

9.2.5 统计聚合

下面展示了如何使用 Java API 进行 统计聚合 。

创建聚合请求

下面是一个创建聚合请求的代码示例:

```
StatsAggregationBuilder aggregation =
        AggregationBuilders
                .stats("agg")
                .field("height");
```

使用聚合响应

引入聚合定义类:

```
import org.elasticsearch.search.aggregations.metrics.stats.Stats;
```

```
// sr is here your SearchResponse object
Stats agg = sr.getAggregations().get("agg");
double min = agg.getMin();
double max = agg.getMax();
double avg = agg.getAvg();
double sum = agg.getSum();
long count = agg.getCount();
```

9.2.6 扩展统计聚合

下面展示了如何使用 Java API 进行 扩展统计聚合 。

创建聚合请求

下面是一个创建聚合请求的代码示例:

```
ExtendedStatsAggregationBuilder aggregation =
        AggregationBuilders
                .extendedStats("agg")
                .field("height");
```

使用聚合响应

引入聚合定义类:

```
import org.elasticsearch.search.aggregations.metrics.stats.extended.ExtendedStats;
```

```
// sr is here your SearchResponse object
ExtendedStats agg = sr.getAggregations().get("agg");
double min = agg.getMin();
double max = agg.getMax();
double avg = agg.getAvg();
double sum = agg.getSum();
long count = agg.getCount();
double stdDeviation = agg.getStdDeviation();
double sumOfSquares = agg.getSumOfSquares();
double variance = agg.getVariance();
```

9.2.7 值计数聚合

下面展示了如何使用 Java API 进行 值计数聚合 。

创建聚合请求

下面是一个创建聚合请求的代码示例:

```
ValueCountAggregationBuilder aggregation =
        AggregationBuilders
                .count("agg")
                .field("height");
```

使用聚合响应

引入聚合定义类:

```
import org.elasticsearch.search.aggregations.metrics.valuecount.ValueCount;
```

```
// sr is here your SearchResponse object
ValueCount agg = sr.getAggregations().get("agg");
long value = agg.getValue();
```

9.2.8 百分比聚合

下面展示了如何使用 Java API 进行 百分比聚合 。

创建聚合请求

下面是一个创建聚合请求的代码示例:

```
PercentilesAggregationBuilder aggregation =
        AggregationBuilders
                .percentiles("agg")
                .field("height");
```

你可以提供自己的百分比替代默认值:

```
PercentilesAggregationBuilder aggregation =
        AggregationBuilders
                .percentiles("agg")
                .field("height")
                .percentiles(1.0, 5.0, 10.0, 20.0, 30.0, 75.0, 95.0, 99.0);
```

使用聚合响应

引入聚合定义类:

```
import org.elasticsearch.search.aggregations.metrics.percentiles.Percentile;
import org.elasticsearch.search.aggregations.metrics.percentiles.Percentiles;
```

```
// sr is here your SearchResponse object
Percentiles agg = sr.getAggregations().get("agg");
// For each entry
for (Percentile entry : agg) {
    double percent = entry.getPercent();    // Percent
    double value = entry.getValue();        // Value

    logger.info("percent [{}], value [{}]", percent, value);
}
```

第一个示例的基本输出如下:

```
percent [1.0], value [0.814338896154595]
percent [5.0], value [0.8761912455821302]
percent [25.0], value [1.173346540141847]
percent [50.0], value [1.5432023318692198]
percent [75.0], value [1.923915462033674]
percent [95.0], value [2.2273644908535335]
percent [99.0], value [2.284989339108279]
```

9.2.9 百分比范围聚合

下面展示了如何使用 Java API 进行 百分比范围聚合 。

创建聚合请求

下面是一个创建聚合请求的代码示例:

```
PercentileRanksAggregationBuilder aggregation =
        AggregationBuilders
                .percentileRanks("agg")
                .field("height")
                .values(1.24, 1.91, 2.22);
```

使用聚合响应

引入聚合定义类:

```
import org.elasticsearch.search.aggregations.metrics.percentiles.Percentile;
import org.elasticsearch.search.aggregations.metrics.percentiles.PercentileRanks;
```

```
// sr is here your SearchResponse object
PercentileRanks agg = sr.getAggregations().get("agg");
// For each entry
for (Percentile entry : agg) {
    double percent = entry.getPercent();    // Percent
    double value = entry.getValue();        // Value

    logger.info("percent [{}], value [{}]", percent, value);
}
```

该示例会产生以下基本输出:

```
percent [29.664353095090945], value [1.24]
percent [73.9335313461868], value [1.91]
percent [94.40095147327283], value [2.22]
```

9.2.10 基数聚合

下面展示了如何使用 Java API 进行 基数聚合 。

创建聚合请求

下面是一个创建聚合请求的代码示例:

```
CardinalityAggregationBuilder aggregation =
        AggregationBuilders
                .cardinality("agg")
                .field("tags");
```

使用聚合响应

引入聚合定义类:

```
import org.elasticsearch.search.aggregations.metrics.cardinality.Cardinality;
```

```
// sr is here your SearchResponse object
Cardinality agg = sr.getAggregations().get("agg");
long value = agg.getValue();
```

9.2.11 地理边界聚合

下面展示了如何使用 Java API 进行 地理边界聚合 。

创建聚合请求

下面是一个创建聚合请求的代码示例:

```
GeoBoundsBuilder aggregation =
        GeoBoundsAggregationBuilder
                .geoBounds("agg")
                .field("address.location")
                .wrapLongitude(true);
```

使用聚合响应

引入聚合定义类:

```
import org.elasticsearch.search.aggregations.metrics.geobounds.GeoBounds;
```

```
// sr is here your SearchResponse object
GeoBounds agg = sr.getAggregations().get("agg");
GeoPoint bottomRight = agg.bottomRight();
GeoPoint topLeft = agg.topLeft();
logger.info("bottomRight {}, topLeft {}", bottomRight, topLeft);
```

该示例会产生以下基本输出:

```
bottomRight [40.70500764381921, 13.952946866893775], topLeft [53.49603022435221, -4.
↪190029308156676]
```

9.2.12 Top Hits聚合

下面展示了如何使用 Java API 进行 Top Hits聚合 。

创建聚合请求

下面是一个创建聚合请求的代码示例:

```
AggregationBuilder aggregation =
    AggregationBuilders
        .terms("agg").field("gender")
        .subAggregation(
            AggregationBuilders.topHits("top")
        );
```

你可以使用大部分的选项用于标准搜索, 比如 *from*, *size*, *sort*, *highlight*, *explain* 等等。

```
AggregationBuilder aggregation =
    AggregationBuilders
        .terms("agg").field("gender")
        .subAggregation(
            AggregationBuilders.topHits("top")
                .explain(true)
                .size(1)
                .from(10)
        );
```

使用聚合响应

引入聚合定义类:

```
import org.elasticsearch.search.aggregations.bucket.terms.Terms;
import org.elasticsearch.search.aggregations.metrics.tophits.TopHits;
```

```
// sr is here your SearchResponse object
Terms agg = sr.getAggregations().get("agg");

// For each entry
for (Terms.Bucket entry : agg.getBuckets()) {
    String key = entry.getKey();                    // bucket key
    long docCount = entry.getDocCount();            // Doc count
    logger.info("key [{}], doc_count [{}]", key, docCount);

    // We ask for top_hits for each bucket
    TopHits topHits = entry.getAggregations().get("top");
    for (SearchHit hit : topHits.getHits().getHits()) {
        logger.info(" -> id [{}], _source [{}]", hit.getId(), hit.
↪getSourceAsString());
    }
}
```

第一个示例会产生以下基本输出:

```
key [male], doc_count [5107]
 -> id [AUnzSZze9k7PKXtq04x2], _source [{"gender":"male",...}]
 -> id [AUnzSZzj9k7PKXtq04x4], _source [{"gender":"male",...}]
 -> id [AUnzSZzl9k7PKXtq04x5], _source [{"gender":"male",...}]
key [female], doc_count [4893]
 -> id [AUnzSZzM9k7PKXtq04xy], _source [{"gender":"female",...}]
 -> id [AUnzSZzp9k7PKXtq04x8], _source [{"gender":"female",...}]
 -> id [AUnzSZ0W9k7PKXtq04yS], _source [{"gender":"female",...}]
```

9.2.13 脚本度量聚合

下面展示了如何使用 Java API 进行 脚本度量聚合 。

如果你想要在一个嵌入式的数据节点上运行 Groovy 脚本, 不要忘了将 Groovy 添加到你的类路径中(例如用于单元测试)。 例如, 你可以将下面的依赖添加到 Maven 的 *pom.xml* 文件中:

```
<dependency>
    <groupId>org.codehaus.groovy</groupId>
    <artifactId>groovy-all</artifactId>
    <version>2.3.2</version>
    <classifier>indy</classifier>
</dependency>
```

创建聚合请求

下面是一个创建聚合请求的代码示例:

```
ScriptedMetricAggregationBuilder aggregation = AggregationBuilders
    .scriptedMetric("agg")
    .initScript(new Script("params._agg.heights = []"))
    .mapScript(new Script("params._agg.heights.add(doc.gender.value == 'male' ? doc.
↪height.value : -1.0 * doc.height.value)"));
```

你也可以指定一个 *combine* 脚本, 它将在每个分片上执行:

```
ScriptedMetricAggregationBuilder aggregation = AggregationBuilders
    .scriptedMetric("agg")
    .initScript(new Script("params._agg.heights = []"))
    .mapScript(new Script("params._agg.heights.add(doc.gender.value == 'male' ? doc.
↪height.value : -1.0 * doc.height.value)"))
    .combineScript(new Script("double heights_sum = 0.0; for (t in params._agg.
↪heights) { heights_sum += t } return heights_sum"));
```

你也可以指定一个 *reduce* 脚本, 它将在节点上执行并得到下面这个请求:

```
ScriptedMetricAggregationBuilder aggregation = AggregationBuilders
    .scriptedMetric("agg")
    .initScript(new Script("params._agg.heights = []"))
    .mapScript(new Script("params._agg.heights.add(doc.gender.value == 'male' ? doc.
↪height.value : -1.0 * doc.height.value)"))
    .combineScript(new Script("double heights_sum = 0.0; for (t in params._agg.
↪heights) { heights_sum += t } return heights_sum"))
    .reduceScript(new Script("double heights_sum = 0.0; for (a in params._aggs) {␣
↪heights_sum += a } return heights_sum"));
```

使用聚合响应

引入聚合定义类:

```
import org.elasticsearch.search.aggregations.bucket.terms.Terms;
import org.elasticsearch.search.aggregations.metrics.tophits.TopHits;
```

```
// sr is here your SearchResponse object
ScriptedMetric agg = sr.getAggregations().get("agg");
Object scriptedResult = agg.aggregation();
logger.info("scriptedResult [{}]", scriptedResult);
```

请注意, 返回结果取决于你构建的脚本.对于第一个示例, 基本输出如下:

```
scriptedResult object [ArrayList]
scriptedResult [ {
"heights" : [ 1.122218480146643, -1.8148918111233887, -1.7626731575142909, ... ]
}, {
"heights" : [ -0.8046067304119863, -2.0785486707864553, -1.9183567430207953, ... ]
}, {
"heights" : [ 2.092635728868694, 1.5697545960886536, 1.8826954461968808, ... ]
}, {
"heights" : [ -2.1863201099468403, 1.6328549117346856, -1.7078288405893842, ... ]
}, {
"heights" : [ 1.6043904836424177, -2.0736538674414025, 0.9898266674373053, ... ]
} ]
```

第二个示例输出如下:

```
scriptedResult object [ArrayList]
scriptedResult [-41.279615707402876,
                -60.88007362339038,
                38.823270659734256,
                14.840192739445632,
                11.300902755741326]
```

最后一个示例输出如下:

```
scriptedResult object [Double]
scriptedResult [2.171917696507009]
```

9.3 Bucket聚合

9.3.1 全局聚合

下面展示了如何使用 Java API 进行 全局聚合 。

创建聚合请求

下面是一个创建聚合请求的代码示例:

```
AggregationBuilders
    .global("agg")
    .subAggregation(AggregationBuilders.terms("genders").field("gender"));
```

使用聚合响应

引入聚合定义类:

```
import org.elasticsearch.search.aggregations.bucket.global.Global;
```

```
// sr is here your SearchResponse object
Global agg = sr.getAggregations().get("agg");
agg.getDocCount(); // Doc count
```

9.3.2 过滤器聚合

下面展示了如何使用 Java API 进行 过滤器聚合 。

创建聚合请求

下面是一个创建聚合请求的代码示例:

```
AggregationBuilders
    .filter("agg", QueryBuilders.termQuery("gender", "male"));
```

使用聚合响应

引入聚合定义类:

```
import org.elasticsearch.search.aggregations.bucket.filter.Filter;
```

```
// sr is here your SearchResponse object
Filter agg = sr.getAggregations().get("agg");
agg.getDocCount(); // Doc count
```

9.3.3 多过滤器聚合

下面展示了如何使用 Java API 进行 多过滤器聚合 。

创建聚合请求

下面是一个创建聚合请求的代码示例:

```
AggregationBuilder aggregation =
    AggregationBuilders
        .filters("agg",
            new FiltersAggregator.KeyedFilter("men", QueryBuilders.termQuery("gender",
↪ "male")),
            new FiltersAggregator.KeyedFilter("women", QueryBuilders.termQuery("gender
↪", "female")));
```

使用聚合响应

引入聚合定义类:

```
import org.elasticsearch.search.aggregations.bucket.filters.Filters;
```

```
// sr is here your SearchResponse object
Filters agg = sr.getAggregations().get("agg");

// For each entry
for (Filters.Bucket entry : agg.getBuckets()) {
    String key = entry.getKeyAsString();            // bucket key
    long docCount = entry.getDocCount();            // Doc count
    logger.info("key [{}], doc_count [{}]", key, docCount);
}
```

该示例会产生以下基本输出:

```
key [men], doc_count [4982]
key [women], doc_count [5018]
```

9.3.4 Missing Aggregation

下面展示了如何使用 Java API 进行 Missing Aggregation 。

创建聚合请求

下面是一个创建聚合请求的代码示例:

```
AggregationBuilders.missing("agg").field("gender");
```

使用聚合响应

引入聚合定义类:

```
import org.elasticsearch.search.aggregations.bucket.missing.Missing;
```

```
// sr is here your SearchResponse object
Missing agg = sr.getAggregations().get("agg");
agg.getDocCount(); // Doc count
```

9.3.5 嵌套聚合

下面展示了如何使用 Java API 进行 嵌套聚合 。

创建聚合请求

下面是一个创建聚合请求的代码示例:

```
AggregationBuilders
    .nested("agg", "resellers");
```

使用聚合响应

引入聚合定义类:

```
import org.elasticsearch.search.aggregations.bucket.nested.Nested;
```

```
// sr is here your SearchResponse object
Nested agg = sr.getAggregations().get("agg");
agg.getDocCount(); // Doc count
```

9.3.6 反向嵌套聚合

下面展示了如何使用 Java API 进行 反向嵌套聚合 。

创建聚合请求

下面是一个创建聚合请求的代码示例:

```
AggregationBuilder aggregation =
    AggregationBuilders
        .nested("agg", "resellers")
        .subAggregation(
                AggregationBuilders
                        .terms("name").field("resellers.name")
                        .subAggregation(
                                AggregationBuilders
                                        .reverseNested("reseller_to_product")
                        )
        );
```

使用聚合响应

引入聚合定义类:

```
import org.elasticsearch.search.aggregations.bucket.nested.Nested;
import org.elasticsearch.search.aggregations.bucket.nested.ReverseNested;
import org.elasticsearch.search.aggregations.bucket.terms.Terms;
```

```
// sr is here your SearchResponse object
Nested agg = sr.getAggregations().get("agg");
Terms name = agg.getAggregations().get("name");
for (Terms.Bucket bucket : name.getBuckets()) {
    ReverseNested resellerToProduct = bucket.getAggregations().get("reseller_to_
↪product");
    resellerToProduct.getDocCount(); // Doc count
}
```

9.3.7 Children聚合

下面展示了如何使用 Java API 进行 Children 聚合 。

创建聚合请求

下面是一个创建聚合请求的代码示例:

```
AggregationBuilder aggregation =
    AggregationBuilders
        .children("agg", "reseller"); <1>
```

1. *“agg”* 是聚合的名称, *“reseller”* 是子类型

使用聚合响应

引入聚合定义类:

```
import org.elasticsearch.search.aggregations.bucket.children.Children;
```

```
// sr is here your SearchResponse object
Children agg = sr.getAggregations().get("agg");
agg.getDocCount(); // Doc count
```

9.3.8 词条聚合

下面展示了如何使用 Java API 进行 词条聚合 。

创建聚合请求

下面是一个创建聚合请求的代码示例:

```
AggregationBuilders
    .terms("genders")
    .field("gender");
```

使用聚合响应

引入聚合定义类:

```
import org.elasticsearch.search.aggregations.bucket.terms.Terms;
```

```
// sr is here your SearchResponse object
Terms genders = sr.getAggregations().get("genders");

// For each entry
for (Terms.Bucket entry : genders.getBuckets()) {
    entry.getKey();      // Term
    entry.getDocCount(); // Doc count
}
```

9.3.9 排序

按照桶的 *doc_count* 的升序排序:

```
AggregationBuilders
    .terms("genders")
    .field("gender")
    .order(Terms.Order.count(true))
```

按照桶中词条的字母的升序排序:

```
AggregationBuilders
    .terms("genders")
    .field("gender")
    .order(Terms.Order.term(true))
```

按照单值的度量子聚合进行桶排序 (按照聚合名称进行标识):

```
AggregationBuilders
    .terms("genders")
    .field("gender")
    .order(Terms.Order.aggregation("avg_height", false))
    .subAggregation(
        AggregationBuilders.avg("avg_height").field("height")
    )
```

9.3.10 重要词条聚合

下面展示了如何使用 Java API 进行 重要词条聚合 。

创建聚合请求

下面是一个创建聚合请求的代码示例:

```
AggregationBuilder aggregation =
        AggregationBuilders
                .significantTerms("significant_countries")
                .field("address.country");

// Let say you search for men only
SearchResponse sr = client.prepareSearch()
        .setQuery(QueryBuilders.termQuery("gender", "male"))
        .addAggregation(aggregation)
        .get();
```

使用聚合响应

引入聚合定义类:

```
import org.elasticsearch.search.aggregations.bucket.significant.SignificantTerms;
```

```
// sr is here your SearchResponse object
SignificantTerms agg = sr.getAggregations().get("significant_countries");

// For each entry
for (SignificantTerms.Bucket entry : agg.getBuckets()) {
    entry.getKey();      // Term
    entry.getDocCount(); // Doc count
}
```

9.3.11 范围聚合

下面展示了如何使用 Java API 进行 范围聚合 。

创建聚合请求

下面是一个创建聚合请求的代码示例:

```
AggregationBuilder aggregation =
        AggregationBuilders
                .range("agg")
                .field("height")
                .addUnboundedTo(1.0f)               // from -infinity to 1.0␣
↪(excluded)
                .addRange(1.0f, 1.5f)               // from 1.0 to 1.5 (excluded)
                .addUnboundedFrom(1.5f);            // from 1.5 to +infinity
```

使用聚合响应

引入聚合定义类:

```
import org.elasticsearch.search.aggregations.bucket.range.Range;
```

```
// sr is here your SearchResponse object
Range agg = sr.getAggregations().get("agg");

// For each entry
for (Range.Bucket entry : agg.getBuckets()) {
    String key = entry.getKeyAsString();             // Range as key
    Number from = (Number) entry.getFrom();          // Bucket from
    Number to = (Number) entry.getTo();              // Bucket to
    long docCount = entry.getDocCount();    // Doc count

    logger.info("key [{}], from [{}], to [{}], doc_count [{}]", key, from, to,␣
↪docCount);
}
```

第一个示例的基本输出如下:

```
key [*-1.0], from [-Infinity], to [1.0], doc_count [9]
key [1.0-1.5], from [1.0], to [1.5], doc_count [21]
key [1.5-*], from [1.5], to [Infinity], doc_count [20]
```

9.3.12 日期范围聚合

下面展示了如何使用 Java API 进行 日期范围聚合 。

创建聚合请求

下面是一个创建聚合请求的代码示例:

```
AggregationBuilder aggregation =
        AggregationBuilders
                .dateRange("agg")
                .field("dateOfBirth")
                .format("yyyy")
                .addUnboundedTo("1950")    // from -infinity to 1950 (excluded)
                .addRange("1950", "1960")  // from 1950 to 1960 (excluded)
                .addUnboundedFrom("1960"); // from 1960 to +infinity
```

使用聚合响应

引入聚合定义类:

```
import org.elasticsearch.search.aggregations.bucket.range.Range;
```

```
// sr is here your SearchResponse object
Range agg = sr.getAggregations().get("agg");

// For each entry
for (Range.Bucket entry : agg.getBuckets()) {
    String key = entry.getKeyAsString();                // Date range as key
    DateTime fromAsDate = (DateTime) entry.getFrom();   // Date bucket from as a Date
    DateTime toAsDate = (DateTime) entry.getTo();       // Date bucket to as a Date
    long docCount = entry.getDocCount();                // Doc count

    logger.info("key [{}], from [{}], to [{}], doc_count [{}]", key, fromAsDate,␣
↪toAsDate, docCount);
}
```

该示例会产生以下基本输出:

```
key [*-1950], from [null], to [1950-01-01T00:00:00.000Z], doc_count [8]
key [1950-1960], from [1950-01-01T00:00:00.000Z], to [1960-01-01T00:00:00.000Z], doc_
↪count [5]
key [1960-*], from [1960-01-01T00:00:00.000Z], to [null], doc_count [37]
```

9.3.13 IP范围聚合

下面展示了如何使用 Java API 进行 IP范围聚合 。

创建聚合请求

下面是一个创建聚合请求的代码示例:

```
AggregatorBuilder<?> aggregation =
        AggregationBuilders
                .ipRange("agg")
                .field("ip")
                .addUnboundedTo("192.168.1.0")             // from -infinity to 192.
↪168.1.0 (excluded)
                .addRange("192.168.1.0", "192.168.2.0")    // from 192.168.1.0 to 192.
↪168.2.0 (excluded)
                .addUnboundedFrom("192.168.2.0");          // from 192.168.2.0 to␣
↪+infinity
```

请注意, 你还可以将IP掩码用作范围:

```
AggregatorBuilder<?> aggregation =
        AggregationBuilders
                .ipRange("agg")
                .field("ip")
                .addMaskRange("192.168.0.0/32")
                .addMaskRange("192.168.0.0/24")
                .addMaskRange("192.168.0.0/16");
```

使用聚合响应

引入聚合定义类:

```
import org.elasticsearch.search.aggregations.bucket.range.Range;
```

```
// sr is here your SearchResponse object
Range agg = sr.getAggregations().get("agg");

// For each entry
for (Range.Bucket entry : agg.getBuckets()) {
    String key = entry.getKeyAsString();            // Ip range as key
    String fromAsString = entry.getFromAsString();  // Ip bucket from as a String
    String toAsString = entry.getToAsString();      // Ip bucket to as a String
    long docCount = entry.getDocCount();            // Doc count

    logger.info("key [{}], from [{}], to [{}], doc_count [{}]", key, fromAsString,␣
↪toAsString, docCount);
}
```

第一个示例的基本输出如下:

```
key [*-192.168.1.0], from [null], to [192.168.1.0], doc_count [13]
key [192.168.1.0-192.168.2.0], from [192.168.1.0], to [192.168.2.0], doc_count [14]
key [192.168.2.0-*], from [192.168.2.0], to [null], doc_count [23]
```

而对于第二个示例 (使用 Ip 掩码) 的基本输出:

```
key [192.168.0.0/32], from [192.168.0.0], to [192.168.0.1], doc_count [0]
key [192.168.0.0/24], from [192.168.0.0], to [192.168.1.0], doc_count [13]
key [192.168.0.0/16], from [192.168.0.0], to [192.169.0.0], doc_count [50]
```

9.3.14 直方图聚合

下面展示了如何使用 Java API 进行 直方图聚合 。

创建聚合请求

下面是一个创建聚合请求的代码示例:

```
AggregationBuilder aggregation =
        AggregationBuilders
                .histogram("agg")
                .field("height")
                .interval(1);
```

使用聚合响应

引入聚合定义类:

```
import org.elasticsearch.search.aggregations.bucket.histogram.Histogram;
```

```
// sr is here your SearchResponse object
Histogram agg = sr.getAggregations().get("agg");

// For each entry
for (Histogram.Bucket entry : agg.getBuckets()) {
    Number key = (Number) entry.getKey();    // Key
    long docCount = entry.getDocCount();     // Doc count

    logger.info("key [{}], doc_count [{}]", key, docCount);
}
```

9.3.15 日期直方图聚合

下面展示了如何使用 Java API 进行 日期直方图聚合 。

创建聚合请求

下面是一个创建聚合请求的代码示例:

```
AggregationBuilder aggregation =
        AggregationBuilders
                .dateHistogram("agg")
                .field("dateOfBirth")
                .dateHistogramInterval(DateHistogramInterval.YEAR);
```

或者你想要设置10天的一个间隔:

```
AggregationBuilder aggregation =
        AggregationBuilders
                .dateHistogram("agg")
                .field("dateOfBirth")
                .dateHistogramInterval(DateHistogramInterval.days(10));
```

使用聚合响应

引入聚合定义类:

```
import org.elasticsearch.search.aggregations.bucket.histogram.Histogram;
```

```
// sr is here your SearchResponse object
Histogram agg = sr.getAggregations().get("agg");

// For each entry
for (Histogram.Bucket entry : agg.getBuckets()) {
    DateTime key = (DateTime) entry.getKey();    // Key
    String keyAsString = entry.getKeyAsString(); // Key as String
    long docCount = entry.getDocCount();         // Doc count

    logger.info("key [{}], date [{}], doc_count [{}]", keyAsString, key.getYear(),␣
↪docCount);
}
```

该示例会产生第一个示例的基本输出:

```
key [1942-01-01T00:00:00.000Z], date [1942], doc_count [1]
key [1945-01-01T00:00:00.000Z], date [1945], doc_count [1]
key [1946-01-01T00:00:00.000Z], date [1946], doc_count [1]
...
key [2005-01-01T00:00:00.000Z], date [2005], doc_count [1]
key [2007-01-01T00:00:00.000Z], date [2007], doc_count [2]
key [2008-01-01T00:00:00.000Z], date [2008], doc_count [3]
```

9.3.16 地理距离聚合

下面展示了如何使用 Java API 进行 地理距离聚合 。

创建聚合请求

下面是一个创建聚合请求的代码示例:

```
AggregationBuilder aggregation =
        AggregationBuilders
                .geoDistance("agg", new GeoPoint(48.84237171118314,2.33320027692004))
                .field("address.location")
                .unit(DistanceUnit.KILOMETERS)
                .addUnboundedTo(3.0)
                .addRange(3.0, 10.0)
                .addRange(10.0, 500.0);
```

使用聚合响应

引入聚合定义类:

```
import org.elasticsearch.search.aggregations.bucket.range.Range;
```

```
// sr is here your SearchResponse object
Range agg = sr.getAggregations().get("agg");

// For each entry
for (Range.Bucket entry : agg.getBuckets()) {
    String key = entry.getKeyAsString();    // key as String
    Number from = (Number) entry.getFrom(); // bucket from value
    Number to = (Number) entry.getTo();     // bucket to value
    long docCount = entry.getDocCount();    // Doc count

    logger.info("key [{}], from [{}], to [{}], doc_count [{}]", key, from, to,␣
↪docCount);
}
```

该示例会产生以下基本输出:

```
key [*-3.0], from [0.0], to [3.0], doc_count [161]
key [3.0-10.0], from [3.0], to [10.0], doc_count [460]
key [10.0-500.0], from [10.0], to [500.0], doc_count [4925]
```

9.3.17 地理哈希网格聚合

下面展示了如何使用 Java API 进行 地理哈希网格聚合 。

创建聚合请求

下面是一个创建聚合请求的代码示例:

```
AggregationBuilder aggregation =
        AggregationBuilders
                .geohashGrid("agg")
                .field("address.location")
                .precision(4);
```

使用聚合响应

引入聚合定义类:

```
import org.elasticsearch.search.aggregations.bucket.geogrid.GeoHashGrid;
```

```
// sr is here your SearchResponse object
GeoHashGrid agg = sr.getAggregations().get("agg");

// For each entry
for (GeoHashGrid.Bucket entry : agg.getBuckets()) {
    String keyAsString = entry.getKeyAsString(); // key as String
    GeoPoint key = (GeoPoint) entry.getKey();    // key as geo point
    long docCount = entry.getDocCount();         // Doc count

    logger.info("key [{}], point {}, doc_count [{}]", keyAsString, key, docCount);
}
```

该示例会产生以下基本输出:

```
key [gbqu], point [47.197265625, -1.58203125], doc_count [1282]
key [gbvn], point [50.361328125, -4.04296875], doc_count [1248]
key [u1j0], point [50.712890625, 7.20703125], doc_count [1156]
key [u0j2], point [45.087890625, 7.55859375], doc_count [1138]
...
```

CHAPTER 10

查询DSL

Elasticsearch 以类似REST‘查询DSL <https://www.elastic.co/guide/en/elasticsearch/reference/6.2/query-dsl.html>‘_ 的方式提供了一套完整的Java查询DSL。查询构造器工厂是 *QueryBuilders*。一旦查询准备好之后, 你可以使用 搜索*API*。

要使用 *QueryBuilders*, 只需要在你的类中引入:

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

请注意, 你可以在 *QueryBuilder* 对象上使用 *toString()* 方法轻松地打印(即 debug)出生成的JSON查询。

接着 *QueryBuilder* 就可以用于任何接受像 *count* 和 *search* 查询的API。

10.1 Match All Query

参见 Match All Query 。

```
matchAllQuery();
```

10.2 全文查询

高级的全文查询通常用于在像电子邮件正文这样的全文本字段上运行全文查询。他们了解如何分析被查询的字段, 并在执行之前将每个字段的 *analyzer*‘（或 ‘*search_analyzer*）应用于查询字符串。

该分组中的查询有:

match 查询

执行全文查询的标准查询, 包括模糊匹配以及短语或邻近查询。

multi_match 查询

match 查询的多字段版本。

common_terms 查询

一个更专门的查询, 它偏好不常见的单词。

query_string 查询

支持紧凑的 Lucene 查询字符串语法, 允许你在单个查询字符串中指定 AND|OR|NOT 条件和多字段搜索。仅限专家用户。

simple_query_string

一个更加简单, 更加健壮版本的 *query_string* 语法, 适合直接暴露给用户。

10.2.1 匹配查询

参见 匹配查询 。

```
matchQuery(
        "name",                     <1>
        "kimchy elasticsearch");    <2>
```

<1> 字段 <2> 文本

10.2.2 Multi Match Query

参见 Multi Match Query 。

```
multiMatchQuery(
        "kimchy elasticsearch",           <1>
        "user", "message");               <2>
```

<1> 文本 <2> 字段

10.2.3 通用词条查询

参见 通用词条查询 。

```
commonTermsQuery("name",          <1>
                 "kimchy");       <2>
```

<1> 字段 <2> 值

10.2.4 查询字符串查询

参见 查询字符串查询 。

```
queryStringQuery("+kimchy -elasticsearch");
```

10.2.5 简单查询字符串查询

参见 简单查询字符串查询 。

```
simpleQueryStringQuery("+kimchy -elasticsearch");
```

10.3 词条级别的查询

全文查询 在执行查询前会对查询字符串进行分词, 而 term-level 查询操作于存储在倒排索引中的每个词条上。

这些查询通常用于数字、日期以及枚举之类的结构化的数据, 而不是全文本字段。Alternatively, they allow you to craft low-level queries, foregoing the analysis process.

该分组中的查询有:

term 查询

查询指定字段中精确包含指定词条的文档

terms 查询

Find documents which contain any of the exact terms specified in the field specified.

range 查询

查询指定字段中包含指定范围值(日期、数字或字符串)的文档。

exists 查询

查询指定字段中包含任意非空值的文档。

prefix 查询

Find documents where the field specified contains terms which being with the exact prefix specified.

wildcard 查询

Find documents where the field specified contains terms which match the pattern specified, where the pattern supports single character wildcards (*?*) and multi-character wildcards (*)

regexp 查询

查询指定字段中包含匹配指定正则表达式的词条的文档。

fuzzy 查询

Find documents where the field specified contains terms which are fuzzily similar to the specified term. Fuzziness is measured as a 莱文斯坦编辑距离 of 1 or 2.

type 查询

查询指定类型的文档。

ids 查询

查询指定类型和文档ID列表的文档。

10.3.1 词条查询

参见 词条查询 。

```
termQuery(
        "name",       <1>
        "kimchy");    <2>
```

<1> 字段 <2> 文本

10.3.2 多词条查询

参见 多词条查询 。

```
termsQuery("tags",                          <1>
        "blue", "pill");                    <2>
```

<1> 字段 <2> 值

10.3.3 范围查询

参见 范围查询 。

```
rangeQuery("price")                         <1>
    .from(5)                                <2>
    .to(10)                                 <3>
    .includeLower(true)                     <4>
    .includeUpper(false);                   <5>
```

<1> 字段 <2> from <3> to <4> 包含较低值意味着为 *false* 时 *from* 是 *gt* 而为 *true'* 时 *'from* 是 *gte*。 <5> 包含较高值意味着为 *false* 时 *to* 是 *lt* 而为 *false* 时 *to* 是 *lte*。

```
// A simplified form using gte, gt, lt or lte
rangeQuery("age")                           <1>
    .gte("10")                              <2>
    .lt("20");                              <3>
```

<1> 字段 <2> 设置 *from* 值为 10, *includeLower* 值为 *true* <3> 设置 *to* 值为 20, *includeUpper* 值为 *false*

10.3.4 存在查询

参见 存在查询 。

```
existsQuery("name");        <1>
```

<1> 字段

10.3.5 前缀查询

参见 前缀查询 。

```
prefixQuery(
        "brand",        <1>
        "heine");       <2>
```

<1> 字段 <2> 前缀

10.3.6 通配符查询

参见 通配符查询 。

```
wildcardQuery(
        "user",           <1>
        "k?mch*");        <2>
```

<1> 字段 <2> 通配符表达式

10.3.7 正则表达式查询

参见 正则表达式查询 。

```
regexpQuery(
        "name.first",         <1>
        "s.*y");              <2>
```

<1> 字段 <2> 正则表达式

10.3.8 模糊查询

参见 模糊查询 。

```
fuzzyQuery(
        "name",        <1>
        "kimzhy");     <2>
```

<1> 字段 <2> 文本

10.3.9 类型查询

参见 类型查询 。

```
typeQuery("my_type");    <1>
```

<1> 类型名称

10.3.10 文档ID查询

参见 文档ID查询 。

```
idsQuery("my_type", "type2")
        .addIds("1", "4", "100");

idsQuery()                                  <1>
        .addIds("1", "4", "100");
```

<1> 类型是可选的

10.4 复合查询

Compound queries wrap other compound or leaf queries, either to combine their results and scores, to change their behaviour, or to switch from query to filter context.

该分组中的查询有:

constant_score 查询

A query which wraps another query, but executes it in filter context. All matching documents are given the same "constant" *score*.

bool 查询

The default query for combining multiple leaf or compound query clauses, as *must*, *should*, *must_not*, or *filter* clauses. The *must* and *should* clauses have their scores combined – the more matching clauses, the better – while the *must_not* and *filter* clauses are executed in filter context.

dis_max 查询

A query which accepts multiple queries, and returns any documents which match any of the query clauses. While the *bool* query combines the scores from all matching queries, the *dis_max* query uses the score of the single best-matching query clause.

function_score 查询

Modify the scores returned by the main query with functions to take into account factors like popularity, recency, distance, or custom algorithms implemented with scripting.

boosting 查询

Return documents which match a *positive* query, but reduce the score of documents which also match a *negative* query.

indices 查询

对指定的索引执行一个查询，为其他索引执行另一个查询。

10.4.1 Constant Score Query

参见 Constant Score Query 。

```
constantScoreQuery(
        termQuery("name","kimchy"))      <1>
    .boost(2.0f);                        <2>
```

<1> 查询 <2> 查询分数

10.4.2 布尔查询

参见 布尔查询 。

```
boolQuery()
        .must(termQuery("content", "test1"))     <1>
        .must(termQuery("content", "test4"))     <2>
        .mustNot(termQuery("content", "test2")) <3>
        .should(termQuery("content", "test3"))  <4>
        .filter(termQuery("content", "test5")); <5>
```

<1> must 查询 <2> <3> must not 查询 <4> should 查询 <5> 必须出现在匹配文档中但不对评分有贡献的查询。

10.4.3 Dis Max Query

参见 Dis Max Query 。

```
disMaxQuery()
        .add(termQuery("name", "kimchy"))               <1>
        .add(termQuery("name", "elasticsearch"))        <2>
        .boost(1.2f)                                    <3>
        .tieBreaker(0.7f);                              <4>
```

<1> 添加查询 <2> 添加查询 <3> boost factor <4> tie breaker

10.4.4 Function Score Query

参见 Function Score Query 。

要使用 *ScoreFunctionBuilders*, 只需要在你的类中引入它们:

```
import static org.elasticsearch.index.query.functionscore.ScoreFunctionBuilders.*;
```

```
FilterFunctionBuilder[] functions = {
        new FunctionScoreQueryBuilder.FilterFunctionBuilder(
                matchQuery("name", "kimchy"),                   <1>
                randomFunction()),                              <2>
        new FunctionScoreQueryBuilder.FilterFunctionBuilder(
                exponentialDecayFunction("age", 0L, 1L))        <3>
};
functionScoreQuery(functions);
```

<1> 基于查询添加第一个函数 <2> 基于给定的种子随机化评分 <3> 基于 age 字段添加另一个函数

10.4.5 Boosting Query

参见 Boosting Query 。

```
boostingQuery(
            termQuery("name","kimchy"),          <1>
            termQuery("name","dadoonet"))        <2>
        .negativeBoost(0.2f);                    <3>
```

<1> 提升文档的查询 <2> 降级文档的查询 <3> negative boost

10.5 连接查询

在像 Elasticsearch 这样的分布式系统中执行完全 SQL 风格的连接查询, 代价是非常昂贵的。相反, Elasticsearch 提供了两种形式的连接, 它们主要设计用于水平扩展。

嵌套查询

> 文档可能包含 *nested* 类型的字段。这些字段用于索引对象数组, 其中每个对象可以作为一个独立的文本进行查询(使用嵌套查询)。

has_child 和 *has_parent*

单个索引中的两种类型的文档之间可以存在父子关系。因为子文档匹配特定的查询, *has_child* 查询会返回父文档, 而因为父文档匹配特定的查询, *has_parent* 查询会返回子文档。

10.5.1 嵌套查询

参见 嵌套查询 。

```
nestedQuery(
        "obj1",                                         <1>
        boolQuery()                                     <2>
                .must(matchQuery("obj1.name", "blue"))
                .must(rangeQuery("obj1.count").gt(5)),
        ScoreMode.Avg);                                 <3>
```

<1> 嵌套文档路径 <2> 你的查询. 查询中引用的任何字段都必须使用完整的路径(全限定的). <3> 评分模式可以是 *ScoreMode.Max*, *ScoreMode.Min*, *ScoreMode.Total*, *ScoreMode.Avg* 或 *ScoreMode.None*

10.5.2 Has Child查询

参见 Has Child Query 。

当使用 has_child 查询时, 使用 PreBuiltTransportClient 而不是常规客户端是很重要的:

```
Settings settings = Settings.builder().put("cluster.name", "elasticsearch").build();
TransportClient client = new PreBuiltTransportClient(settings);
client.addTransportAddress(new InetSocketTransportAddress(new␣
↪InetSocketAddress(InetAddresses.forString("127.0.0.1"), 9300)));
```

否则的话 parent-join 模块不会被加载并且传输客户端无法使用 has_child 查询。

```
JoinQueryBuilders.hasChildQuery(
        "blog_tag",                     <1>
        termQuery("tag","something"),   <2>
        ScoreMode.None);                <3>
```

<1> 要查询的子类型 <2> 查询 <3> 评分模式可以是 *ScoreMode.Avg*, *ScoreMode.Max*, *ScoreMode.Min*, *ScoreMode.None* 或 *ScoreMode.Total*

10.5.3 Has Parent Query

参见 Has Parent Query 。

当使用 has_parent 查询时, 使用 PreBuiltTransportClient 而不是常规客户端是很重要的:

```
Settings settings = Settings.builder().put("cluster.name", "elasticsearch").build();
TransportClient client = new PreBuiltTransportClient(settings);
client.addTransportAddress(new InetSocketTransportAddress(new␣
↪InetSocketAddress(InetAddresses.forString("127.0.0.1"), 9300)));
```

否则的话 parent-join 模块不会被加载并且传输客户端无法使用 has_parent 查询。

```
JoinQueryBuilders.hasParentQuery(
    "blog",                     <1>
    termQuery("tag","something"),   <2>
    false);                     <3>
```

<1> 要查询的父类型 <2> 查询 <3> whether the score from the parent hit should propogate to the child hit

10.6 地理查询

Elasticsearch 支持两种类型的地理数据:支持 lat/lon 对的 *geo_point* 字段和支持点、线、circles、多边形, multi-polygons等的 *geo_shape* 字段。

该分组中的查询有:

geo_shape 查询

Find document with geo-shapes which either intersect, are contained by, or do not intersect with the specified geo-shape。

geo_bounding_box 查询

Finds documents with geo-points that fall into the specified rectangle。

geo_distance 查询

Finds document with geo-points within the specified distance of a central point。

geo_polygon 查询

Find documents with geo-points within the specified polygon。

10.6.1 地理形状查询

参见 地理形状查询 。

注意: *geo_shape* 类型使用了 *Spatial4J* 和 *JTS*, 它们都是可选的依赖。因此为了使用这种类型, 你必须要将 *Spatial4J* 和 *JTS* 依赖添加到你的类路径中:

```
<dependency>
    <groupId>org.locationtech.spatial4j</groupId>
    <artifactId>spatial4j</artifactId>
    <version>0.6</version>                          <1>
</dependency>

<dependency>
    <groupId>com.vividsolutions</groupId>
    <artifactId>jts</artifactId>
    <version>1.13</version>                           <2>
    <exclusions>
        <exclusion>
            <groupId>xerces</groupId>
            <artifactId>xercesImpl</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

<1> 在 Maven 中央仓库 中检查更新 <2> 在 Maven 中央仓库 中检查更新

```
// Import ShapeRelation and ShapeBuilder
import org.elasticsearch.common.geo.ShapeRelation;
import org.elasticsearch.common.geo.builders.ShapeBuilder;
```

```
GeoShapeQueryBuilder qb = geoShapeQuery(
        "pin.location",                                 <1>
        ShapeBuilders.newMultiPoint(                    <2>
                new CoordinatesBuilder()
            .coordinate(0, 0)
            .coordinate(0, 10)
            .coordinate(10, 10)
            .coordinate(10, 0)
            .coordinate(0, 0)
            .build()));
qb.relation(ShapeRelation.WITHIN);                      <3>
```

<1> 字段 <2> 形状 <3> relation 可以是 *ShapeRelation.CONTAINS*, *ShapeRelation.WITHIN*, *ShapeRelation.INTERSECTS* 或 *ShapeRelation.DISJOINT*

```
// Using pre-indexed shapes
GeoShapeQueryBuilder qb = geoShapeQuery(
            "pin.location",                             <1>
            "DEU",                                      <2>
            "countries");                               <3>
qb.relation(ShapeRelation.WITHIN)                       <4>
    .indexedShapeIndex("shapes")                        <5>
    .indexedShapePath("location");                      <6>
```

<1> 字段 <2> The ID of the document that containing the pre-indexed shape. <3> Index type where the pre-indexed shape is. <4> relation <5> Name of the index where the pre-indexed shape is. Defaults to 'shapes'. <6> The field specified as path containing the pre-indexed shape. Defaults to 'shape'.

10.6.2 Geo Bounding Box Query

参见 Geo Bounding Box Query 。

```
geoBoundingBoxQuery("pin.location")                     <1>
    .setCorners(40.73, -74.1,                           <2>
                40.717, -73.99);                        <3>
```

<1> 字段 <2> 边界框顶部左边点 <3> 边界框底部右边点

10.6.3 地理距离查询

参见 地理距离查询 。

```
geoDistanceQuery("pin.location")  <1>
    .point(40, -70)                                     <2>
    .distance(200, DistanceUnit.KILOMETERS);            <3>
```

<1> 字段 <2> 中心点 <3> 到中心点的距离

10.6.4 地理多变形查询

参见 地理多变形查询 。

```
List<GeoPoint> points = new ArrayList<>();              <1>
points.add(new GeoPoint(40, -70));
points.add(new GeoPoint(30, -80));
points.add(new GeoPoint(20, -90));

geoPolygonQuery("pin.location", points);                <2>
```

<1> 添加文档应落入的多边形的点 <2> 使用字段和点初始化查询

10.7 Specialized queries

This group contains queries which do not fit into the other groups:

more_like_this 查询

This query finds documents which are similar to the specified text, document, or collection of documents.

script 查询

This query allows a script to act as a filter. Also see the <<java-query-dsl-function-score-query,'function_score' query>>.

percolate 查询

This query finds percolator queries based on documents.

10.7.1 More Like This Query (mlt)

参见 More Like This Query 。

```
String[] fields = {"name.first", "name.last"};                  <1>
String[] texts = {"text like this one"};                        <2>
Item[] items = null;

moreLikeThisQuery(fields, texts, items)
    .minTermFreq(1)                                             <3>
    .maxQueryTerms(12);                                         <4>
```

<1> 字段 <2> 文本 <3> 忽略阈值 <4> 生成的查询中词条的最大数量

10.7.2 脚本查询

参见 脚本查询 。

```
scriptQuery(
        new Script("doc['num1'].value > 1")      <1>
);
```

<1> 内联脚本

如果你在每个数据节点上都存储了一个名为 *myscript.painless* 的脚本, 脚本内容如下:

```
doc['num1'].value > params.param1
```

那么你可以像下面这样使用它:

```
Map<String, Object> parameters = new HashMap<>();
parameters.put("param1", 5);
scriptQuery(new Script(
        ScriptType.STORED,                                  <1>
        null,                                               <2>
        "myscript",                                         <3>
        singletonMap("param1", 5)));                        <4>
```

<1> 脚本类型: *ScriptType.FILE*, *ScriptType.INLINE* 或 *ScriptType.INDEXED* <2> 脚本引擎 <3> 脚本名称 <4> 参数是 *<String, Object>* 类型的 *Map*

10.7.3 Percolate Query

参见 Percolate Query 。

```
Settings settings = Settings.builder().put("cluster.name", "elasticsearch").build();
TransportClient client = new PreBuiltTransportClient(settings);
client.addTransportAddress(new InetSocketTransportAddress(new␣
↪InetSocketAddress(InetAddresses.forString("127.0.0.1"), 9300)));
```

Before the *percolate* query can be used an *percolator* mapping should be added and a document containing a percolator query should be indexed:

```
// create an index with a percolator field with the name 'query':
client.admin().indices().prepareCreate("myIndexName")
                        .addMapping("query", "query", "type=percolator")
                        .addMapping("docs", "content", "type=text")
                        .get();

//This is the query we're registering in the percolator
QueryBuilder qb = termQuery("content", "amazing");

//Index the query = register it in the percolator
client.prepareIndex("myIndexName", "query", "myDesignatedQueryName")
    .setSource(jsonBuilder()
        .startObject()
            .field("query", qb) // Register the query
        .endObject())
    .setRefreshPolicy(RefreshPolicy.IMMEDIATE) // Needed when the query shall be␣
↪available immediately
    .get();
```

This indexes the above term query under the name *myDesignatedQueryName*.

In order to check a document against the registered queries, 使用下面的代码:

```
//Build a document to check against the percolator
XContentBuilder docBuilder = XContentFactory.jsonBuilder().startObject();
docBuilder.field("content", "This is amazing!");
docBuilder.endObject(); //End of the JSON root object

PercolateQueryBuilder percolateQuery = new PercolateQueryBuilder("query", "docs",␣
↪docBuilder.bytes());

// Percolate, by executing the percolator query in the query dsl:
SearchResponse response = client().prepareSearch("myIndexName")
        .setQuery(percolateQuery))
```

```
        .get();
//Iterate over the results
for(SearchHit hit : response.getHits()) {
    // Percolator queries as hit
}
```

10.8 Span queries

Span queries are low-level positional queries which provide expert control over the order and proximity of the specified terms. These are typically used to implement very specific queries on legal documents or patents.

Span queries cannot be mixed with non-span queries (with the exception of the *span_multi* query).

该分组中的查询有:

span_term 查询

The equivalent of the <<java-query-dsl-term-query,'term' query>> but for use with other span queries.

span_multi 查询

Wraps a <<java-query-dsl-term-query,'term'>>, <<java-query-dsl-range-query,'range'>>,

<<java-query-dsl-prefix-query,'prefix'>>, <<java-query-dsl-wildcard-query,'wildcard'>>, <<java-query-dsl-regexp-query,'regexp'>>, or <<java-query-dsl-fuzzy-query,'fuzzy'>> query.

span_first 查询

Accepts another span query whose matches must appear within the first N positions of the field.

span_near 查询

Accepts multiple span queries whose matches must be within the specified distance of each other, and possibly in the same order.

span_or 查询

Combines multiple span queries – returns documents which match any of the specified queries.

span_not 查询

Wraps another span query, and excludes any documents which match that query.

span_containing 查询

Accepts a list of span queries, but only returns those spans which also match a second span query.

span_within 查询

The result from a single span query is returned as long is its span falls within the spans returned by a list of other span queries.

10.8.1 Span Term Query

参见 Span Term Query 。

```
spanTermQuery(
        "user",                                 <1>
        "kimchy");                              <2>
```

<1> field <2> value

10.8.2 Span Multi Term Query

参见 Span Multi Term Query 。

```
spanMultiTermQueryBuilder(
        prefixQuery("user", "ki"));                    <1>
```

<1> 可以是任何继承了 *MultiTermQueryBuilder* 类的生成器。例如: *FuzzyQueryBuilder*, *PrefixQueryBuilder*, *RangeQueryBuilder*, *RegexpQueryBuilder* 或者 *WildcardQueryBuilder*.

10.8.3 Span First Query

参见 Span First Query 。

```
spanFirstQuery(
        spanTermQuery("user", "kimchy"),             <1>
        3                                            <2>
);
```

<1> 查询 <2> 最大结束位置

10.8.4 Span Near Query

参见 Span Near Query 。

```
spanNearQuery(
        spanTermQuery("field","value1"),                 <1>
        12)                                              <2>
            .addClause(spanTermQuery("field","value2")) <3>
            .addClause(spanTermQuery("field","value3")) <4>
            .inOrder(false);                             <5>
```

<1> span term queries <3> <4> <2> slop factor: the maximum number of intervening unmatched positions <5> whether matches are required to be in-order

10.8.5 Span Or Query

参见 Span Or Query 。

```
spanOrQuery(spanTermQuery("field","value1"))        <1>        <1>
    .addClause(spanTermQuery("field","value2"))     <2>
    .addClause(spanTermQuery("field","value3"));    <3>
```

<1> span term queries <2> <3>

10.8.6 Span Not Query

参见 Span Not Query 。

```
spanNotQuery(
        spanTermQuery("field","value1"),    <1>
        spanTermQuery("field","value2"));   <2>
```

<1> 匹配结果被过滤的span query <2> span query whose matches must not overlap those returned

10.8.7 Span Containing Query

参见 Span Containing Query 。

```
spanContainingQuery(
        spanNearQuery(spanTermQuery("field1","bar"), 5) <1>
            .addClause(spanTermQuery("field1","baz"))
            .inOrder(true),
        spanTermQuery("field1","foo"));                 <2>
```

<1> *big* 部分 <2> *little* 部分

10.8.8 Span Within Query

参见 Span Within Query 。

```
spanWithinQuery(
        spanNearQuery(spanTermQuery("field1", "bar"), 5)  <1>
            .addClause(spanTermQuery("field1", "baz"))
            .inOrder(true),
        spanTermQuery("field1", "foo"));                  <2>
```

<1> *big* part <2> *little* part

CHAPTER 11

Java管理API

Elasticsearch 提供了一套完整的 Java API 来处理管理任务.

要想访问他们, 你需要在客户端对象上调用 *admin()* 方法来获取一个 *AdminClient* 对象:

```
AdminClient adminClient = client.admin();
```

Note: 在本指南的剩余部分, 我们将使用 *client.admin()*。

11.1 索引管理

要通过 Java API 访问索引, 你需要调用 *AdminClient* 对象的 *indices()* 方法:

```
IndicesAdminClient indicesAdminClient = client.admin().indices();
```

Note: 在本指南的剩余部分, 我们将使用 *client.admin().indices()*。

11.1.1 创建索引

使用 IndicesAdminClient 对象, 你可以创建一个使用所有默认设置并且没有映射的索引:

```
client.admin().indices().prepareCreate("twitter").get();
```

索引设置

创建的每个索引都可以有与其相关的特定设置.

```
client.admin().indices().prepareCreate("twitter")
        .setSettings(Settings.builder()             <1>
                .put("index.number_of_shards", 3)
                .put("index.number_of_replicas", 2)
        )
        .get();                                     <2>
```

<1> 该索引的设置 <2> 执行操作并等待响应结果

11.1.2 Put Mapping

PUT 映射 API 允许你在创建索引时添加一个新的类型:

```
client.admin().indices().prepareCreate("twitter")    <1>
    .addMapping("tweet", "{\n" +                     <2>
            "    \"tweet\": {\n" +
            "      \"properties\": {\n" +
            "        \"message\": {\n" +
            "          \"type\": \"text\"\n" +
            "        }\n" +
            "      }\n" +
            "    }\n" +
            "  }")
    .get();
```

<1> 创建一个名为 *twitter* 的索引 <2> 它同时添加了一个名为 *tweet* 的映射类型。

PUT 映射 API 还允许用户给现有的索引添加一个新的类型:

```
client.admin().indices().preparePutMapping("twitter")     <1>
        .setType("user")                                  <2>
        .setSource("{\n" +                                <3>
                "  \"properties\": {\n" +
                "    \"name\": {\n" +
                "      \"type\": \"text\"\n" +
                "    }\n" +
                "  }\n" +
                "}")
        .get();

// You can also provide the type in the source document
client.admin().indices().preparePutMapping("twitter")
        .setType("user")
        .setSource("{\n" +
                "    \"user\":{\n" +                        <4>
                "        \"properties\": {\n" +
                "            \"name\": {\n" +
                "                \"type\": \"text\"\n" +
                "            }\n" +
                "        }\n" +
                "    }\n" +
                "}")
        .get();
```

<1> 在现有索引上添加一个映射 *twitter* <2> 添加一个 *user* 映射类型. <3> *user* 有一个预定义的类型 <4> 类型也可以在source中提供

你可以使用同样的 API 来更新一个现有的映射:

```
client.admin().indices().preparePutMapping("twitter")    <1>
        .setType("user")                                 <2>
        .setSource("{\n" +                                <3>
                "  \"properties\": {\n" +
                "    \"user_name\": {\n" +
                "      \"type\": \"text\"\n" +
                "    }\n" +
                "  }\n" +
                "}")
        .get();
```

<1> 在现有索引上添加一个映射 *twitter* <2> 更新 *user* 映射类型. <3> *user* 现在有一个新的字段 *user_name*

11.1.3 刷新

刷新 API 允许用户显示地刷新一个或多个索引:

```
client.admin().indices().prepareRefresh().get(); <1>
client.admin().indices()
        .prepareRefresh("twitter")                <2>
        .get();
client.admin().indices()
        .prepareRefresh("twitter", "company")    <3>
        .get();
```

<1> 刷新所有索引 <2> 刷新一个索引 <3> 刷新多个索引

11.1.4 获取设置

获取设置 API API 允许用户检索一个或多个索引的设置:

```
GetSettingsResponse response = client.admin().indices()
        .prepareGetSettings("company", "employee").get();
↪<1>
for (ObjectObjectCursor<String, Settings> cursor : response.getIndexToSettings()) {
↪<2>
    String index = cursor.key;
↪<3>
    Settings settings = cursor.value;
↪<4>
    Integer shards = settings.getAsInt("index.number_of_shards", null);
↪<5>
    Integer replicas = settings.getAsInt("index.number_of_replicas", null);
↪<6>
}
```

<1> 获取索引 *company* 和 *employee* 的设置 <2> 遍历结果集 <3> 索引名称 <4> 指定索引的设置 <5> 索引的分片数量 <6> 索引的副本数量

11.1.5 更新索引设置

通过调用以下代码你可以更改索引设置:

```
client.admin().indices().prepareUpdateSettings("twitter")    <1>
        .setSettings(Settings.builder()                      <2>
                .put("index.number_of_replicas", 0)
        )
        .get();
```

<1> 待更新的索引 <2> 设置

11.2 集群管理

要访问集群 Java API, 你需要在 *AdminClient* 对象上调用 *cluster()* 方法:

```
ClusterAdminClient clusterAdminClient = client.admin().cluster();
```

Note: 在本指南的剩余部分, 我们将使用 *client.admin().cluster()*.

11.2.1 集群健康

健康

集群健康 API 允许用户获取有关集群健康的一个非常简单的状态并且也可以给你一些有关每个索引的集群状态的技术信息:

```
ClusterHealthResponse healths = client.admin().cluster().prepareHealth().get(); <1>
String clusterName = healths.getClusterName();               <2>
int numberOfDataNodes = healths.getNumberOfDataNodes();      <3>
int numberOfNodes = healths.getNumberOfNodes();              <4>

for (ClusterIndexHealth health : healths.getIndices().values()) { <5>
    String index = health.getIndex();                        <6>
    int numberOfShards = health.getNumberOfShards();         <7>
    int numberOfReplicas = health.getNumberOfReplicas();     <8>
    ClusterHealthStatus status = health.getStatus();         <9>
}
```

<1> 获取所有索引信息 <2> 获取集群名称 <3> 获取数据节点总数 <4> 获取节点总数 <5> 遍历所有索引 <6> 索引名称 <7> 分片数量 <8> 副本数量 <9> 索引状态

等待特定状态

你可以使用集群健康 API 来等待整个集群或指定的索引达到一个特定的状态:

```
client.admin().cluster().prepareHealth()                     <1>
        .setWaitForYellowStatus()                            <2>
        .get();
client.admin().cluster().prepareHealth("company")            <3>
        .setWaitForGreenStatus()                             <4>
        .get();

client.admin().cluster().prepareHealth("employee")           <5>
```

```
        .setWaitForGreenStatus()                        <6>
        .setTimeout(TimeValue.timeValueSeconds(2))   <7>
        .get();
```

<1> 准备一个健康请求对象 <2> 等待集群状态变成 yellow <3> 为 *company* 索引准备健康请求对象 <4> 等待索引状态变成 green <5> 为 *employee* 索引准备健康请求对象 <6> 等待索引状态变成 green <7> 最多等待 2s

如果索引没有达到预期的状态值并且你想在这种情况下失败, 你需要显示地中断结果:

```
ClusterHealthResponse response = client.admin().cluster().prepareHealth("company")
        .setWaitForGreenStatus()    <1>
        .get();

ClusterHealthStatus status = response.getIndices().get("company").getStatus();
if (!status.equals(ClusterHealthStatus.GREEN)) {
    throw new RuntimeException("Index is in " + status + " state"); <2>
}
```

<1> 等待索引状态变成 green <2> 如果不是 *GREEN* 抛出异常

11.2.2 存储脚本 API

存储脚本 API 允许用户和存储在 Elasticsearch 索引中的脚本和模板进行交互。它可以用来创建, 更新, 查询以及删除存储的脚本和模板。

```
PutStoredScriptResponse response = client.admin().cluster().preparePutStoredScript()
                .setId("script1")
                .setContent(new BytesArray("{\"script\": {\"lang\": \"painless\", \
↪"source\": \"_score * doc['my_numeric_field'].value\"} }"), XContentType.JSON)
                .get();

GetStoredScriptResponse response = client().admin().cluster().prepareGetStoredScript()
                .setId("script1")
                .get();

DeleteStoredScriptResponse response = client().admin().cluster().
↪prepareDeleteStoredScript()
                .setId("script1")
                .get();
```

要存储模板, 可以简单地在 scriptLang 上使用 “mustache”。

脚本语言

该 API 允许用户设置与之交互的索引脚本的语言. 如果没有设置则将使用默认的脚本语言。

CHAPTER 12

Indices and tables

- genindex
- modindex
- search