目錄

Elasticsearch Java Rest API 手册

本手册由 全科 翻译，并且整理成电子书，支持PDF,ePub,Mobi格式，方便大家下载阅读。

阅读地址：https://es.quanke.name

下载地址：https://www.gitbook.com/book/quanke/elasticsearch-java-rest

github地址：https://github.com/quanke/elasticsearch-java-rest

gitee 地址：https://gitee.com/quanke/elasticsearch-java-rest

编辑：http://quanke.name

编辑整理辛苦，还望大神们点一下star ，抚平我虚荣的心

> 不只是官方文档的翻译，还包含使用实例，包含我们使用踩过的坑

更多请关注我的微信公众号：

概览

Java REST 客户端分为两种:

- Java 低级 REST

Elasticsearch 官方低级客户端： 通过 `http` 协议 与Elasticsearch服务进行通信。请求编码和响应解码保留给用户实现。与所有 Elasticsearch 版本兼容。

- Java 高级 REST

Elasticsearch 官方高级客户端： 基于低级客户端，提供特定的方法的API，并处理请求编码和响应解码。

Java 低级 REST 客户端

低级客户端的功能：

- 最小依赖
- 负载均衡
- 故障转移
- 故障连接策略 (是否重新连接故障节点取决于连续失败多少次；失败次数越多，在再次尝试同一个节点之前，客户端等待的时间越长)
- 持久化连接
- 跟踪记录请求和响应
- 自动发现集群节点

起步

本节介绍了如何在应用程序中使用低级REST客户端。

Javadoc

https://artifacts.elastic.co/javadoc/org/elasticsearch/client/elasticsearch-rest-client/5.6.0/index.html

Maven Repository

托管在 Maven Central

Java 版本最低要求 1.7

> `low-level` REST 客户端与 `elasticsearch` 的发布周期相同。可以使用版本替换，但必须是 5.0.0-alpha4 之后的版本。客户端版本与 `Elasticsearch` 服务版本之间没有关联。 `low-level` REST 客户端兼容所有 Elasticsearch 版本。

Maven 配置

使用 Maven 作依赖管理,将下列内容添加到你的 pom.xml 文件里：

```
<dependency>
    <groupId>org.elasticsearch.client</groupId>
    <artifactId>elasticsearch-rest-client</artifactId>
    <version>5.6.4</version>
</dependency>
```

Gradle 配置

使用 gradle 作依赖管理，将下列内容添加到你的 build.gradle 文件里：

```
dependencies {
    compile 'org.elasticsearch.client:elasticsearch-rest-client:
5.6.4'
}
```

依赖

低级 `Java REST` 客户端内部使用 `Apache Http Async Client` 发送 `http` 请求。它依赖以下工具，即 `Apache Http Async Client` 依赖：

- org.apache.httpcomponents:httpasyncclient
- org.apache.httpcomponents:httpcore-nio
- org.apache.httpcomponents:httpclient
- org.apache.httpcomponents:httpcore
- commons-codec:commons-codec
- commons-logging:commons-logging

Shading

为了避免版本冲突，可以在单个JAR文件（有时称为“uber JAR”或“fat JAR”）中将客户端中的依赖项隐藏并打包在客户端中。 `Shading JAR` 可以通过 `Gradle` 和 `Maven` 的第三方插件完成。

请注意，Shading JAR也有影响。 例如，Shading Commons Logging层意味着第三方日志记录后端也需要被shaded 。

Maven 配置

你可以这样配置 Maven Shade 插件 。将下列内容添加到你的 pom.xml 文件里：

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-shade-plugin</artifactId>
            <version>3.1.0</version>
            <executions>
                <execution>
                    <phase>package</phase>
                    <goals><goal>shade</goal></goals>
                    <configuration>
                        <relocations>
                            <relocation>
                                <pattern>org.apache.http</patter
n>
                                <shadedPattern>hidden.org.apache
.http</shadedPattern>
                            </relocation>
                            <relocation>
                                <pattern>org.apache.logging</pat
tern>
                                <shadedPattern>hidden.org.apache
.logging</shadedPattern>
                            </relocation>
```

```
                                <relocation>
                                    <pattern>org.apache.commons.code
c</pattern>
                                    <shadedPattern>hidden.org.apache
.commons.codec</shadedPattern>
                                </relocation>
                                <relocation>
                                    <pattern>org.apache.commons.logg
ing</pattern>
                                    <shadedPattern>hidden.org.apache
.commons.logging</shadedPattern>
                                </relocation>
                            </relocations>
                        </configuration>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
```

Gradle 配置

你可以这样配置 Gradle ShadowJar 插件. 添加以下内容到你的 build.gradle 文件中：

```
shadowJar {
    relocate 'org.apache.http', 'hidden.org.apache.http'
    relocate 'org.apache.logging', 'hidden.org.apache.logging'
    relocate 'org.apache.commons.codec', 'hidden.org.apache.comm
ons.codec'
    relocate 'org.apache.commons.logging', 'hidden.org.apache.co
mmons.logging'
}
```

初始化

`RestClient` 实例可以通过相应的 `RestClientBuilder` 类来构建,通过静态方法 `RestClient#builder(HttpHost...)` 创建。唯一必需的参数是服务的host和端口（默认9200，切记不要使用9300），以 `HttpHost` 实例的方式提供给建造者，如下所示：

```
RestClient restClient = RestClient.builder(
        new HttpHost("localhost", 9200, "http"),
        new HttpHost("localhost", 9201, "http")).build();
```

`RestClient` 类是线程安全的，理想状态下它与使用它的应用程序具有相同的生命周期。当不再使用时强烈建议关闭它：

```
restClient.close();
```

`RestClientBuilder` 在构建 `RestClient` 实例时可以设置以下的可选配置参数：

```
RestClientBuilder builder = RestClient.builder(new HttpHost("loc
alhost", 9200, "http"));
Header[] defaultHeaders = new Header[]{
        new BasicHeader("header", "value")
    };
builder.setDefaultHeaders(defaultHeaders); // 设置默认头文件，避免每
个请求都必须指定。
```

```
RestClientBuilder builder = RestClient.builder(new HttpHost("loc
alhost", 9200, "http"));

builder.setMaxRetryTimeoutMillis(10000);//设置在同一请求进行多次尝试
时应该遵守的超时时间。默认值为30秒，与默认`socket`超时相同。 如果自定义设置
了`socket`超时，则应该相应地调整最大重试超时。
```

```
RestClientBuilder builder = RestClient.builder(new HttpHost("loc
alhost", 9200, "http"));
builder.setFailureListener(new RestClient.FailureListener() {
    @Override
    public void onFailure(HttpHost host) {
        //设置每次节点发生故障时收到通知的侦听器。内部嗅探到故障时被启用。
    }
});
```

```
RestClientBuilder builder = RestClient.builder(new HttpHost("loc
alhost", 9200, "http"));
builder.setRequestConfigCallback(new RestClientBuilder.RequestCo
nfigCallback() {
        @Override
        public RequestConfig.Builder customizeRequestConfig(Requ
estConfig.Builder requestConfigBuilder) {
            return requestConfigBuilder.setSocketTimeout(10000);
 // 设置修改默认请求配置的回调（例如：请求超时，认证，或者其他 [org.apache
.http.client.config.RequestConfig.Builder](https://hc.apache.org
/httpcomponents-client-ga/httpclient/apidocs/org/apache/http/cli
ent/config/RequestConfig.Builder.html) 设置）。
        }
    });
```

```
RestClientBuilder builder = RestClient.builder(new HttpHost("loc
alhost", 9200, "http"));
builder.setHttpClientConfigCallback(new RestClientBuilder.HttpCl
ientConfigCallback() {
    @Override
    public HttpAsyncClientBuilder customizeHttpClient(HttpAsyncC
lientBuilder httpClientBuilder) {
        return httpClientBuilder.setProxy(new HttpHost("proxy",
9000, "http")); //设置修改 http 客户端配置的回调（例如：ssl 加密通讯，
或其他任何 [org.apache.http.impl.nio.client.HttpAsyncClientBuilder
](http://hc.apache.org/httpcomponents-asyncclient-dev/httpasyncc
lient/apidocs/org/apache/http/impl/nio/client/HttpAsyncClientBui
lder.html) 设置）
    }
});
```

发送请求

一旦创建了 `RestClient` ，就可以通过调用其中一个 `performRequest` 或 `performRequestAsync` 方法来发送请求。 `performRequest` 方法是同步的，并直接返回 `Response` ，这意味着客户端将阻塞并等待返回的响应。 `performRequestAsync` 返回 `void` ，并接受一个额外的 `ResponseListener` 作为参数，这意味着它们是异步执行的。 提供的监听器将在请求完成或失败时通知。

```
Response response = restClient.performRequest("GET", "/"); //最简单的发送一个请求
```

```
Map<String, String> params = Collections.singletonMap("pretty", "true");
Response response = restClient.performRequest("GET", "/", params);  //发送一个带参数的请求
```

```
Map<String, String> params = Collections.emptyMap();
String jsonString = "{" +
            "\"user\":\"kimchy\"," +
            "\"postDate\":\"2013-01-30\"," +
            "\"message\":\"trying out Elasticsearch\"" +
        "}";
HttpEntity entity = new NStringEntity(jsonString, ContentType.APPLICATION_JSON);// org.apache.http.HttpEntity  为了让Elasticsearch 能够解析,需要设置ContentType。
Response response = restClient.performRequest("PUT", "/posts/doc/1", params, entity);
```

```
Map<String, String> params = Collections.emptyMap();
HttpAsyncResponseConsumerFactory.HeapBufferedResponseConsumerFac
tory consumerFactory =
        new HttpAsyncResponseConsumerFactory.HeapBufferedRespons
eConsumerFactory(30 * 1024 * 1024); //[ org.apache.http.nio.prot
ocol.HttpAsyncResponseConsumer](http://hc.apache.org/httpcompone
nts-core-ga/httpcore-nio/apidocs/org/apache/http/nio/protocol/Ht
tpAsyncResponseConsumer.html) ,
Response response = restClient.performRequest("GET", "/posts/_se
arch", params, null, consumerFactory);
```

```
ResponseListener responseListener = new ResponseListener() {
    @Override
    public void onSuccess(Response response) {
        // 请求成功回调
    }

    @Override
    public void onFailure(Exception exception) {
        //请求失败时回调
    }
};
restClient.performRequestAsync("GET", "/", responseListener); //
发送异步请求
```

```
Map<String, String> params = Collections.singletonMap("pretty",
"true");
restClient.performRequestAsync("GET", "/", params, responseListe
ner); // 发送带参数的异步请求
```

```
String jsonString = "{" +
        "\"user\":\"kimchy\"," +
        "\"postDate\":\"2013-01-30\"," +
        "\"message\":\"trying out Elasticsearch\"" +
        "}";
HttpEntity entity = new NStringEntity(jsonString, ContentType.AP
PLICATION_JSON);
restClient.performRequestAsync("PUT", "/posts/doc/1", params, en
tity, responseListener);
```

```
HttpAsyncResponseConsumerFactory.HeapBufferedResponseConsumerFac
tory consumerFactory =
        new HttpAsyncResponseConsumerFactory.HeapBufferedRespons
eConsumerFactory(30 * 1024 * 1024);
restClient.performRequestAsync("GET", "/posts/_search", params,
null, consumerFactory, responseListener);
```

以下是如何发送异步请求的基本示例：

```
final CountDownLatch latch = new CountDownLatch(documents.length
);
for (int i = 0; i < documents.length; i++) {
    restClient.performRequestAsync(
            "PUT",
            "/posts/doc/" + i,
            Collections.<String, String>emptyMap(),
            //let's assume that the documents are stored in an H
ttpEntity array
            documents[i],
            new ResponseListener() {
                @Override
                public void onSuccess(Response response) {

                    latch.countDown();//处理返回响应
                }

                @Override
                public void onFailure(Exception exception) {

                    latch.countDown();//处理失败响应，exception 里
带错误码
                }
            }
    );
}
latch.await();
```

上面列出的每一种方法都支持通过 `Header varargs` 参数和请求一起 `headers` ，如下例所示：

```
Response response = restClient.performRequest("GET", "/", new Ba
sicHeader("header", "value"));
```

```
Header[] headers = {
        new BasicHeader("header1", "value1"),
        new BasicHeader("header2", "value2")
};
restClient.performRequestAsync("GET", "/", responseListener, hea
ders);
```

读取响应

返回 `Response` 对象（ `performRequest` ,方法返回， `ResponseListener#onSuccess(Response)` 接收）

```
Response response = restClient.performRequest("GET", "/");
RequestLine requestLine = response.getRequestLine(); //请求信息
HttpHost host = response.getHost(); //返回response host信息
int statusCode = response.getStatusLine().getStatusCode(); //返回状态行，获取状态码
Header[] headers = response.getHeaders(); //response headers ，也可以通过名字获取 `getHeader(String)`
String responseBody = EntityUtils.toString(response.getEntity()); //response  org.apache.http.HttpEntity 对象
```

请求时可能抛出一下异常（或者 `ResponseListener#onFailure(Exception)` 参数接收错误信息）

- `IOException`

通信问题（例如: `SocketTimeoutException` ）

- `ResponseException`

返回了一个 response，但是它的状态码显示了一个错误（不是2xx）。 `ResponseException` 说明连接是通的。

> 对于返回404状态码的请求，不会抛出 `ResponseException` ，因为这是一个预期的响应，只是表示找不到该资源。 除非 `ignore` 参数包含404，否则所有其他HTTP方法（例如GET）都会为404响应抛出 `ResponseException` 。 `ignore` 是一个特殊的客户端参数，不会发送到 `Elasticsearch` ，并且包含以逗号分隔的错误状态码列表。 它允许控制某些错误状态代码是否应该被视为预期的响应，而不是一个例外。 这对 `get api` 来说很有用，因为它可以在缺少文档时返回404，在这种情况下，响应主体不会包含错误，而是通常的 `get api` 响应，而不是文档，因为它没有找到。

注意: 低级别的客户端不公开任何 helper jsonmarshalling和un-marshalling。 用户可以自由使用他们喜欢的库。

底层的Apache异步Http客户端附带不同的 `org.apache.http.HttpEntity` 实现，可以使用不同的格式（流，字节数组，字符串等）提供请求主体。 至于读取响应主体， `HttpEntity#getContent` 方法很方便，它返回来自先前缓冲的响应主体 `InputStream` 。 可以提供一个自定义的 `org.apache.http.nio.protocol.HttpAsyncResponseConsumer` 来控制如何读取和缓冲字节，作为替代。

日志记录

Java REST 客户端使用和 `Apache Async Http` 客户端使用的相同日志记录库： `Apache Commons Logging` ，它支持许多流行的日志记录实现。 用于启用日志记录的java包是客户端本身的 `org.elasticsearch.client` ，嗅探器是 `org.elasticsearch.client.sniffer` 。

请求 `tracer` 日志记录可以开启以curl格式记录。 这在调试时非常方便，例如在需要手动执行请求的情况下，检查是否仍然产生相同的响应。 请注意，这种类型的日志记录非常消耗资源，不应一直在生产环境中启用，而只是在需要时暂时使用。

通用配置

正如初始化中所解释的那样，`RestClientBuilder` 支持同时提供 `RequestConfigCallback` 和 `HttpClientConfigCallback` ，它们允许任何定制的 `Apache Async Http Client` 。这些回调可以修改客户端的某些特定行为，而不会覆盖 `RestClient` 初始化的其他任何默认配置。本节介绍了一些需要对低级Java REST客户端进行额外配置的常见方案。

超时

配置请求超时可以通过构建器构建 `RestClient` 时提供 `RequestConfigCallback` 实例来完成。该接口有一个方法接收 `org.apache.http.client.config.RequestConfig.Builder` 的一个实例作为参数，并具有相同的返回类型。请求配置生成器可以修改，然后返回。在下面的例子中，我们增加了连接超时（默认为1秒）和socket超时（默认为30秒）。也调整最大重试超时时间（默认为30秒）。

```
RestClientBuilder builder = RestClient.builder(new HttpHost("loc
alhost", 9200))
        .setRequestConfigCallback(new RestClientBuilder.RequestC
onfigCallback() {
            @Override
            public RequestConfig.Builder customizeRequestConfig(
RequestConfig.Builder requestConfigBuilder) {
                return requestConfigBuilder.setConnectTimeout(50
00)
                        .setSocketTimeout(60000);
            }
        })
        .setMaxRetryTimeoutMillis(60000);
```

线程数

`Apache Http Async Client` 默认启动一个调度线程，连接管理器使用多个 `worker` 线程,线程的数量和CPU核数量相同（等于 `Runtime.getRuntime().availableProcessors()` `Runtime.getRuntime().availableProcessors()` 返回的数量），线程数可以修改如下：

```
RestClientBuilder builder = RestClient.builder(new HttpHost("loc
alhost", 9200))
        .setHttpClientConfigCallback(new RestClientBuilder.HttpC
lientConfigCallback() {
            @Override
            public HttpAsyncClientBuilder customizeHttpClient(Ht
tpAsyncClientBuilder httpClientBuilder) {
                return httpClientBuilder.setDefaultIOReactorConf
ig(
                        IOReactorConfig.custom().setIoThreadCoun
t(1).build());
            }
        });
```

基本认证

构建 `RestClient` 时配置 `HttpClientConfigCallback` 来配置基本认证。 该接口有一个方法接
收 `org.apache.http.impl.nio.client.HttpAsyncClientBuilder` 的一个实例作为参数，并具有相同的返回类型。 `httpClientBuilder` 被修改，然后返回。 在以下示例中，设置了基本身份验证。

```
final CredentialsProvider credentialsProvider = new BasicCredent
ialsProvider();
credentialsProvider.setCredentials(AuthScope.ANY,
        new UsernamePasswordCredentials("user", "password"));

RestClientBuilder builder = RestClient.builder(new HttpHost("loc
alhost", 9200))
        .setHttpClientConfigCallback(new RestClientBuilder.HttpC
lientConfigCallback() {
            @Override
            public HttpAsyncClientBuilder customizeHttpClient(Ht
tpAsyncClientBuilder httpClientBuilder) {
                return httpClientBuilder.setDefaultCredentialsPr
ovider(credentialsProvider);
            }
        });
```

Preemptive 身份验证可以禁用，这意味着每个发送出去的请求没有授权头，当收到 `HTTP 401` 响应时，将重新发送与基本身份验证头完全相同的请求。 可以通过 `HttpAsyncClientBuilder` 来禁用：

```
final CredentialsProvider credentialsProvider = new BasicCredent
ialsProvider();
credentialsProvider.setCredentials(AuthScope.ANY,
        new UsernamePasswordCredentials("user", "password"));

RestClientBuilder builder = RestClient.builder(new HttpHost("loc
alhost", 9200))
        .setHttpClientConfigCallback(new RestClientBuilder.HttpC
lientConfigCallback() {
            @Override
            public HttpAsyncClientBuilder customizeHttpClient(Ht
tpAsyncClientBuilder httpClientBuilder) {
                httpClientBuilder.disableAuthCaching(); //禁用 pr
eemptive 身份验证
                return httpClientBuilder.setDefaultCredentialsPr
ovider(credentialsProvider);
            }
        });
```

加密传输

可以通过 `HttpClientConfigCallback` 配置加密传输。 参数 `org.apache.http.impl.nio.client.HttpAsyncClientBuilder` 公开了多个方法来配置加密传输：`setSSLContext` ， `setSSLSessionStrategy` 和 `setConnectionManager` ，以下是一个例子：

```
KeyStore truststore = KeyStore.getInstance("jks");
try (InputStream is = Files.newInputStream(keyStorePath)) {
    truststore.load(is, keyStorePass.toCharArray());
}
SSLContextBuilder sslBuilder = SSLContexts.custom().loadTrustMat
erial(truststore, null);
final SSLContext sslContext = sslBuilder.build();
RestClientBuilder builder = RestClient.builder(new HttpHost("loc
alhost", 9200, "https"))
        .setHttpClientConfigCallback(new RestClientBuilder.HttpC
lientConfigCallback() {
            @Override
            public HttpAsyncClientBuilder customizeHttpClient(Ht
tpAsyncClientBuilder httpClientBuilder) {
                return httpClientBuilder.setSSLContext(sslContex
t);
            }
        });
```

如果没有提供明确的配置，则使用系统默认配置。

其他

对于其他配置，查阅 `Apache HttpAsyncClient` 文档：https：//hc.apache.org/httpcomponents-asyncclient-4.1.x/。

嗅探器

这是个小型库，可以允许从一个正在运行的 `Elasticsearch` 集群上自动发现节点并将节点列表更新到已经存在的 `RestClient` 实例上。 它默认使用 `Nodes Info api` 检索属于集群的节点，并使用 `jackson` 解析获取的 `json` 响应。

与 `Elasticsearch 2.x` 及以上兼容。

Javadoc

REST 客户端嗅探器的 `javadoc`
https://artifacts.elastic.co/javadoc/org/elasticsearch/client/elasticsearch-rest-client-sniffer/5.6.0/index.html

Maven 仓库

REST 客户端嗅探器与 elasticsearch 的发行周期相同。可以使用期望的嗅探器版本替换，但必须是 5.0.0-alpha4 之后的版本。嗅探器版本与其通信的 Elasticsearch 版本之间没有关联。嗅探器支持从 elasticsearch 2.x 及以上的版本上获取节点列表。

Maven 配置

若使用 Maven 作依赖管理，你可以这样配置依赖。将下列内容添加到你的 pom.xml 文件里：

```
<dependency>
    <groupId>org.elasticsearch.client</groupId>
    <artifactId>elasticsearch-rest-client-sniffer</artifactId>
    <version>5.6.4</version>
</dependency>
```

Gradle 配置

若使用 gradle 作依赖管理，你可以这样配置依赖。将下列内容添加到你的 build.gradle 文件里：

```
dependencies {
    compile 'org.elasticsearch.client:elasticsearch-rest-client-
sniffer:5.6.0'
}
```

用法

一旦创建了 `RestClient` 实例，如 初始化 中所示，可以将嗅探器与之相关联。 `Sniffer` 将使用关联的 `RestClient` 定期（默认为每5分钟）从集群中获取当前可用的节点列表，并通过调用 `RestClient#setHosts` 来更新它们。

```
RestClient restClient = RestClient.builder(
        new HttpHost("localhost", 9200, "http"))
        .build();
Sniffer sniffer = Sniffer.builder(restClient).build();
```

关闭 `Sniffer` 非常重要，如此嗅探器后台线程才能正取关闭并释放他持有的资源。 `Sniffer` 对象应该与 `RestClient` 具有相同的生命周期，并在客户端之前关闭：

```
sniffer.close();
restClient.close();
```

`Sniffer` 默认每5分钟更新一次节点列表。这个周期可以如下方式通过提供一个参数（毫秒数）自定义设置：

```
RestClient restClient = RestClient.builder(
        new HttpHost("localhost", 9200, "http"))
        .build();
Sniffer sniffer = Sniffer.builder(restClient)
        .setSniffIntervalMillis(60000).build();
```

也可以在发生故障是启用嗅探，这意味着每次故障后将直接获取并更新节点列表，而不是等到下一次正常的更新周期。此种情况时， SniffOnFailureListener 需要首先被创建，并将实例在 RestClient 创建时提供给它。 同样的，在之后创建 Sniffer 时，他需要被关联到同一个 SniffOnFailureListener 实例上，这个实例将在每个故障发生后被通知到，然后调用 Sniffer 去执行额外的嗅探行为。

```
SniffOnFailureListener sniffOnFailureListener = new SniffOnFailu
reListener();
RestClient restClient = RestClient.builder(new HttpHost("localho
st", 9200))
        .setFailureListener(sniffOnFailureListener) //为 RestCli
ent 实例设置故障监听器
        .build();
Sniffer sniffer = Sniffer.builder(restClient
        .setSniffAfterFailureDelayMillis(30000) /* 故障后嗅探，不仅
意味着每次故障后会更新节点，也会添加普通计划外的嗅探行为，默认情况是故障之后1
分钟后，假设节点将恢复正常，那么我们希望尽可能快的获知。如上所述，周期可以通
过 `setSniffAfterFailureDelayMillis` 方法在创建 Sniffer 实例时进行自
定义设置。 需要注意的是，当没有启用故障监听时，这最后一个配置参数不会生效 */
        .build();
sniffOnFailureListener.setSniffer(sniffer); // 将 嗅探器关联到嗅探
故障监听器上
```

`Elasticsearch Nodes Info api` 不会返回连接节点使用的协议，而只有他们的 `host:port` 键值对，因此默认使用 `http` 。如果需要使用 `https` ，必须手动创建和提供 `ElasticsearchHostsSniffer` 实例，如下所示：

```
RestClient restClient = RestClient.builder(
        new HttpHost("localhost", 9200, "http"))
        .build();
HostsSniffer hostsSniffer = new ElasticsearchHostsSniffer(
        restClient,
        ElasticsearchHostsSniffer.DEFAULT_SNIFF_REQUEST_TIMEOUT,
        ElasticsearchHostsSniffer.Scheme.HTTPS);
Sniffer sniffer = Sniffer.builder(restClient)
        .setHostsSniffer(hostsSniffer).build();
```

使用同样的方式，可以自定义设置 sniffRequestTimeout参数，该参数默认值为 1 秒。这是一个调用 Nodes Info api 时作为 querystring 参数的超时参数，这样当服务端超市时，仍然会返回一个有效响应，虽然它可能仅包含属于集群的一部分节点，其他节点会在随后响应。

```
RestClient restClient = RestClient.builder(
        new HttpHost("localhost", 9200, "http"))
        .build();
HostsSniffer hostsSniffer = new ElasticsearchHostsSniffer(
        restClient,
        TimeUnit.SECONDS.toMillis(5),
        ElasticsearchHostsSniffer.Scheme.HTTP);
Sniffer sniffer = Sniffer.builder(restClient)
        .setHostsSniffer(hostsSniffer).build();
```

同样的，一个自定义的 HostsSniffer 实现可以提供一个高级用法功能，比如可以从 Elasticsearch 之外的来源获取主机：

```
RestClient restClient = RestClient.builder(
        new HttpHost("localhost", 9200, "http"))
        .build();
HostsSniffer hostsSniffer = new HostsSniffer() {
    @Override
    public List<HttpHost> sniffHosts() throws IOException {
        return null; // 从外部源获取主机
    }
};
Sniffer sniffer = Sniffer.builder(restClient)
        .setHostsSniffer(hostsSniffer).build();
```

Java 高级 REST 客户端

Java高级REST客户端可以在Java Low Level REST客户机之上工作。 其主要目标是公开特定方法的API，接受请求对象作为参数并返回响应对象，以便客户端自己处理请求编组和响应解组。

每个API可以同步或异步地调用。 同步方法返回一个响应对象，而名称以 `async` 后缀结尾的异步方法需要收到响应或错误后才会通知（在低级别客户端管理的线程池上）的侦听器参数。

Java高级REST客户端依赖于 `Elasticsearch` 核心项目。 它接受与 `TransportClient` 相同的请求参数，并返回相同的响应对象。

起步

本节介绍了如何在应用程序中使用高级REST客户端。

兼容性

Java高级REST客户端需要Java 1.8，并依赖于Elasticsearch核心项目。 客户端版本要与客户端开发的Elasticsearch版本相同。 它接受与 `TransportClient` 相同的请求参数，并返回相同的响应对象。 如果需要将应用程序从TransportClient迁移到新的REST客户端，请参阅 迁移指南 。

高级客户端保证能够与运行在相同主版本和大于或等于次要版本的任何Elasticsearch节点进行通信。它不需要与它进行通信的弹性搜索节点相同的次要版本，因为它是向前兼容的，意味着它支持与之前开发的弹性搜索的更新版本进行通信。

5.6 客户端可以与任何 5.6.x Elasticsearch 节点进行通信。以前的 5.x 小版本，如 5.5.x，5.4.x 等不（完全）支持。

6.0 客户端能够与任何 6.x Elasticsearch 节点进行通信，而 6.1 客户端确实能够与 6.1,6.2 和以后的 6.x 版本进行通信，但与以前的 Elasticsearch 节点版本通信时可能会出现不兼容问题例如 6.1 到 6.0 之间，例如 6.1 客户端支持而 6.0 节点不知道的某些API的新请求主体字段。

建议在将Elasticsearch集群升级到新的主要版本时升级高级客户端，因为REST API突破性更改可能会导致意外的结果，具体取决于请求所击中的节点，新添加的API只能由较新版本的客户端。一旦群集中的所有节点都升级到新的主版本，则客户端应当更新。

Javadoc

https://artifacts.elastic.co/javadoc/org/elasticsearch/client/elasticsearch-rest-high-level-client/5.6.0/index.html

Maven 仓库

高级 Java REST 客户端被托管在 Maven 中央仓库里。所需的最低Java版本为 1.8。

高级 REST 客户端与 elasticsearch 的发行周期相同。可以使用期望的版本进行替换。

Maven 配置

若使用 Maven 作依赖管理，你可以这样配置依赖。将下列内容添加到你的 pom.xml 文件里：

```
<dependency>
    <groupId>org.elasticsearch.client</groupId>
    <artifactId>elasticsearch-rest-high-level-client</artifactId
>
    <version>5.6.0</version>
</dependency>
```

Gradle 配置

若使用 gradle 作依赖管理，你可以这样配置依赖。将下列内容添加到你的 build.gradle 文件里：

```
dependencies {
    compile 'org.elasticsearch.client:elasticsearch-rest-high-le
vel-client:5.6.0'
}
```

依赖项

高级 Java REST Client 依赖以下包：

- org.elasticsearch.client:elasticsearch-rest-client
- org.elasticsearch:elasticsearch

初始化

RestHighLevelClient 实例的构建需要一个 REST 低级客户端就像下面这样：

```
RestHighLevelClient client =
    new RestHighLevelClient(lowLevelRestClient); //lowLevelRestC
lient： 我们之前创建的 [REST低级客户端](https://www.elastic.co/guide
/en/elasticsearch/client/java-rest/current/java-rest-low-usage-i
nitialization.html) 实例
```

在本文档中关于Java高级客户端的的其余部分里，`RestHighLevelClient` 实例将以 `client` 被引用。

支持的 API

Java 高级 REST 客户端支持下列 API ：

单一文档 API

- Index API
- Get API
- Delete API
- Update API

多文档 API

- Bulk API

搜索 API

- Search API
- Search Scroll API
- Clear Scroll API

各种 API

- Info API

Index API

Index 请求

IndexRequest 要求下列参数：

```
IndexRequest request = new IndexRequest(
        "posts",  //Index
        "doc",  //Type
        "1");   //Document id
String jsonString = "{" +
        "\"user\":\"kimchy\"," +
        "\"postDate\":\"2013-01-30\"," +
        "\"message\":\"trying out Elasticsearch\"" +
        "}";
request.source(jsonString, XContentType.JSON); /以字符串提供的 Doc
ument source
```

文档来源

文件来源可以以不同的方式提供：

```
Map<String, Object> jsonMap = new HashMap<>();
jsonMap.put("user", "kimchy");
jsonMap.put("postDate", new Date());
jsonMap.put("message", "trying out Elasticsearch");
IndexRequest indexRequest = new IndexRequest("posts", "doc", "1"
).source(jsonMap); //Map 作为文档源，它可以自动转换为 JSON 格式。
```

```
XContentBuilder builder = XContentFactory.jsonBuilder();
builder.startObject();
{
    builder.field("user", "kimchy");
    builder.field("postDate", new Date());
    builder.field("message", "trying out Elasticsearch");
}
builder.endObject();
IndexRequest indexRequest = new IndexRequest("posts", "doc", "1"
).source(builder); //XContentBuilder 对象作为文档源，由 Elasticsear
ch 内置的帮助器生成 JSON 内容
```

```
IndexRequest indexRequest = new IndexRequest("posts", "doc", "1"
)
        .source("user", "kimchy",
                "postDate", new Date(),
                "message", "trying out Elasticsearch"); //以键值
对对象作为文档来源，它自动转换为 JSON 格式
```

可选参数

下列参数可选：

```
request.routing("routing"); //Routing 值
```

```
request.parent("parent"); //Parent 值
```

```
request.timeout(TimeValue.timeValueSeconds(1)); //`TimeValue`类型
的等待主分片可用的超时时间
request.timeout("1s"); //`String` 类型的等待主分片可用的超时时间
```

```
request.setRefreshPolicy(WriteRequest.RefreshPolicy.WAIT_UNTIL);
 //以 WriteRequest.RefreshPolicy 实例的刷新策略参数
request.setRefreshPolicy("wait_for"); // 字符串刷新策略参数
```

```
request.version(2); //版本
```

```
request.versionType(VersionType.EXTERNAL); //版本类型
```

```
request.opType(DocWriteRequest.OpType.CREATE); //提供一个 DocWrit
eRequest.OpType 值作为操作类型
request.opType("create"); //字符串类型的操作类型参数: 可以是 create
或 update (默认值)
```

```
request.setPipeline("pipeline"); //在索引文档之前要执行的摄取管道的名
称
```

同步执行

```
IndexResponse indexResponse = client.index(request);
```

异步执行

```
client.indexAsync(request, new ActionListener<IndexResponse>() {
    @Override
    public void onResponse(IndexResponse indexResponse) {
        //当操作成功完成的时候被调用。响应对象以参数的形式传入。
    }
    @Override
    public void onFailure(Exception e) {
        //故障时被调用。异常对象以参数的形式传入
    }
});
```

Index 响应

返回的“IndexResponse”可以检索有关执行操作的信息，如下所示：

```
String index = indexResponse.getIndex();
String type = indexResponse.getType();
String id = indexResponse.getId();
long version = indexResponse.getVersion();
if (indexResponse.getResult() == DocWriteResponse.Result.CREATED
) {
    //处理（如果需要）首次创建文档的情况
} else if (indexResponse.getResult() == DocWriteResponse.Result.
UPDATED) {
    //处理（如果需要）文档已经存在时被覆盖的情况
}
ReplicationResponse.ShardInfo shardInfo = indexResponse.getShard
Info();
if (shardInfo.getTotal() != shardInfo.getSuccessful()) {
    //处理成功碎片数少于总碎片的情况
}
if (shardInfo.getFailed() > 0) {
    for (ReplicationResponse.ShardInfo.Failure failure : shardIn
fo.getFailures()) {
        String reason = failure.reason();//处理潜在的故障
    }
}
```

如果存在版本冲突，将抛出 ElasticsearchException :

```
IndexRequest request = new IndexRequest("posts", "doc", "1")
        .source("field", "value")
        .version(1);
try {
    IndexResponse response = client.index(request);
} catch(ElasticsearchException e) {
    if (e.status() == RestStatus.CONFLICT) {
        //表示是由于返回了版本冲突错误引发的异常
    }
}
```

发生同样的情况发生在opType设置为create但是已经存在具有相同索引，类型和id的文档时：

```
IndexRequest request = new IndexRequest("posts", "doc", "1")
        .source("field", "value")
        .opType(DocWriteRequest.OpType.CREATE);
try {
    IndexResponse response = client.index(request);
} catch(ElasticsearchException e) {
    if (e.status() == RestStatus.CONFLICT) {
        //表示由于返回了版本冲突错误引发的异常
    }
}
```

Get API

获取请求对象

GetRequest 需要以下参数：

```
GetRequest getRequest = new GetRequest(
        "posts", // 索引
        "doc",  // 类别
        "1");   // 文档id
```

可选参数

提供以下可选参数：

```
request.fetchSourceContext(new FetchSourceContext(false)); //禁用检索源，默认为启用
```

String[] includes = new String[]{"message", "*Date"}; String[] excludes = Strings.EMPTY_ARRAY; FetchSourceContext fetchSourceContext = new FetchSourceContext(true, includes, excludes); request.fetchSourceContext(fetchSourceContext); //设置源包含的特定域

```

```

String[] includes = Strings.EMPTY_ARRAY; String[] excludes = new String[] {"message"}; FetchSourceContext fetchSourceContext = new FetchSourceContext(true, includes, excludes); request.fetchSourceContext(fetchSourceContext); Configure source exclusion for specific fields

```
```

request.storedFields("message"); //Configure retrieval for specific stored fields (requires fields to be stored separately in the mappings) GetResponse getResponse = client.get(request); String message = (String) getResponse.getField("message").getValue(); //Retrieve the message stored field (requires the field to be stored separately in the mappings)

```
```
request.routing("routing"); //Routing value
request.parent("parent"); //Parent value
request.preference("preference"); //Preference value
request.realtime(false); //Set realtime flag to false (true by d
efault)
request.refresh(true); //Perform a refresh before retrieving the
 document (false by default)
request.version(2); //Version
request.versionType(VersionType.EXTERNAL); //Version type
```

## 同步执行

```
GetResponse getResponse = client.get(getRequest);
```

异步执行
```

```
client.getAsync(request, new ActionListener<GetResponse>() {
    @Override
    public void onResponse(GetResponse getResponse) {
//Called when the execution is successfully completed. The respo
nse is provided as an argument.
    }
    @Override
    public void onFailure(Exception e) {
//Called in case of failure. The raised exception is provided as
 an argument.
    }
});
```

获取响应

The returned GetResponse allows to retrieve the requested document along with its metadata and eventually stored fields.

```
String index = getResponse.getIndex();
String type = getResponse.getType();
String id = getResponse.getId();
if (getResponse.isExists()) {
    long version = getResponse.getVersion();
    String sourceAsString = getResponse.getSourceAsString(); //R
etrieve the document as a String
    Map<String, Object> sourceAsMap = getResponse.getSourceAsMap
(); //Retrieve the document as a Map<String, Object>
    byte[] sourceAsBytes = getResponse.getSourceAsBytes();  //Re
trieve the document as a byte[]
} else {
//Handle the scenario where the document was not found. Note tha
t although the returned response has 404 status code, a valid Ge
tResponse is returned rather than an exception thrown. Such resp
onse does not hold any source document and its isExists method r
eturns false.
}
```

当针对不存在的索引执行获取请求时，响应有404状态码，抛出一个 ElasticsearchException 异常，需要如下处理：

```
GetRequest request = new GetRequest("does_not_exist", "doc", "1"
);
try {
    GetResponse getResponse = client.get(request);
} catch (ElasticsearchException e) {
    if (e.status() == RestStatus.NOT_FOUND) {
        // 处理因为索引不存在而抛出的异常，
    }
}
```

如果请求了特定文档版本，但现有文档具有不同的版本号，则会引发版本冲突：

```
try {
    GetRequest request = new GetRequest("posts", "doc", "1").ver
sion(2);
    GetResponse getResponse = client.get(request);
} catch (ElasticsearchException exception) {
    if (exception.status() == RestStatus.CONFLICT) {
        //表示返回了版本冲突错误引发异常
    }
}
```

Delete API

删除请求对象

DeleteRequest 需要下列参数：

```
DeleteRequest request = new DeleteRequest(
        "posts",    // index
        "doc",     //Type
        "1");  // Document id
```

可选参数

提供下列可选参数：

```
request.routing("routing");  // 路由值
request.parent("parent");  //Parent 值
request.timeout(TimeValue.timeValueMinutes(2)); // TimeValue 类型的等待主分片可用的超时时间
request.timeout("2m");  // 字符串类型的等待主分片可用的超时时间
request.setRefreshPolicy(WriteRequest.RefreshPolicy.WAIT_UNTIL);
  // Refresh policy as a WriteRequest.RefreshPolicy instance
request.setRefreshPolicy("wait_for");  // Refresh policy as a String
request.version(2);   // Version
request.versionType(VersionType.EXTERNAL);  // Version type
```

同步执行

```
DeleteResponse deleteResponse = client.delete(request);
```

异步执行

```
client.deleteAsync(request, new ActionListener<DeleteResponse>()
 {
    @Override
    public void onResponse(DeleteResponse deleteResponse) {
        //Called when the execution is successfully completed. T
he response is provided as an argument
    }
    @Override
    public void onFailure(Exception e) {
        //Called in case of failure. The raised exception is pro
vided as an argument
    }
});
```

删除操作的响应

返回的 DeleteResponse 对象允许通过执行以下操作获取相关信息：

```
String index = deleteResponse.getIndex();
String type = deleteResponse.getType();
String id = deleteResponse.getId();
long version = deleteResponse.getVersion();
ReplicationResponse.ShardInfo shardInfo = deleteResponse.getShar
dInfo();
if (shardInfo.getTotal() != shardInfo.getSuccessful()) {
// Handle the situation where number of successful shards is les
s than total shards
}
if (shardInfo.getFailed() > 0) {
    for (ReplicationResponse.ShardInfo.Failure failure : shardIn
fo.getFailures()) {
        String reason = failure.reason();  // Handle the potenti
al failures
    }
}
```

也可以检查文档是否被发现：

```
DeleteRequest request = new DeleteRequest("posts", "doc", "does_
not_exist");
DeleteResponse deleteResponse = client.delete(request);
if (deleteResponse.getResult() == DocWriteResponse.Result.NOT_FO
UND) {
    //如果被删除的文档没有被找到，做某些操作
}
```

如果有版本冲突，将抛出 ElasticsearchException 异常信息：

```
try {
    DeleteRequest request = new DeleteRequest("posts", "doc", "1
").version(2);
    DeleteResponse deleteResponse = client.delete(request);
} catch (ElasticsearchException exception) {
    if (exception.status() == RestStatus.CONFLICT) {
        //由于版本冲突错误导致的异常
    }
}
```

Update API

更新请求

UpdateRequest 要求下列参数：

```
UpdateRequest request = new UpdateRequest(
        "posts", // Index
        "doc",  // Type
        "1"); // Document id
```

Update API允许通过使用脚本或传递部分文档来更新现有文档。

使用脚本更新

该脚本可以是一个内联脚本：

```
Map<String, Object> parameters = singletonMap("count", 4); // 脚本参数以一个 Map 对象提供。
Script inline = new Script(ScriptType.INLINE, "painless", "ctx._source.field += params.count", parameters);  // 使用 painless 语言创建内联脚本，并传入参数值。
request.script(inline);  //  将脚本传递给更新请求对象
```

或者是一个存储脚本：

```
// 引用一个名为 increment-field 的painless 的存储脚本
Script stored =
        new Script(ScriptType.STORED, "painless", "increment-field", parameters);
request.script(stored); // 给请求设置脚本
```

使用局部文档更新

使用局部文档更新时，局部文档将与现有文档合并。

局部文档可以以不同的方式提供：

```
UpdateRequest request = new UpdateRequest("posts", "doc", "1");
String jsonString = "{" +
        "\"updated\":\"2017-01-01\"," +
        "\"reason\":\"daily update\"" +
        "}";
request.doc(jsonString, XContentType.JSON); // 以一个 JSON 格式的字符串提供的局部文档源
```

```
Map<String, Object> jsonMap = new HashMap<>();
jsonMap.put("updated", new Date());
jsonMap.put("reason", "daily update");
UpdateRequest request = new UpdateRequest("posts", "doc", "1")
        .doc(jsonMap); // 以一个可自动转换为 JSON 格式的 Map 提供局部文档源
```

```
XContentBuilder builder = XContentFactory.jsonBuilder();
builder.startObject();
{
    builder.field("updated", new Date());
    builder.field("reason", "daily update");
}
builder.endObject();
UpdateRequest request = new UpdateRequest("posts", "doc", "1")
        .doc(builder); // 以 XContentBuilder 对象提供局部文档源， Elasticsearch 构建辅助器将生成 JSON 格式。
```

```
UpdateRequest request = new UpdateRequest("posts", "doc", "1")
        .doc("updated", new Date(),
             "reason", "daily update"); // 以 Object 键值对提供局部文档源，它将被转换为 JSON 格式。
```

更新或插入

如果文档不存在，可以使用upsert方法定义一些将作为新文档插入的内容：

```
String jsonString = "{\"created\":\"2017-01-01\"}";
request.upsert(jsonString, XContentType.JSON);  // 以字符串提供更新插入的文档源
与局部文档更新类似，可以使用接受 String， Map ， XContentBuilder 或 Object 键值对的方式使用upsert 方法更新或插入文档的内容。
```

可选参数

提供下列可选参数：

```
request.routing("routing");  // 路由值
request.parent("parent");  //Parent 值
request.timeout(TimeValue.timeValueMinutes(2)); // TimeValue 类型的等待主分片可用的超时时间
request.timeout("2m");  // 字符串类型的等待主分片可用的超时时间
request.setRefreshPolicy(WriteRequest.RefreshPolicy.WAIT_UNTIL);
  // Refresh policy as a WriteRequest.RefreshPolicy instance
request.setRefreshPolicy("wait_for");  // Refresh policy as a String
request.retryOnConflict(3);  //    How many times to retry the update operation if the document to update has been changed by another operation between the get and indexing phases of the update operation
request.fetchSource(true); //Enable source retrieval, disabled by default
request.version(2); // 版本号
request.detectNoop(false);  // Disable the noop detection
request.scriptedUpsert(true); // Indicate that the script must run regardless of whether the document exists or not, ie the script takes care of creating the document if it does not already exist.
request.docAsUpsert(true); // Indicate that the partial document must be used as the upsert document if it does not exist yet.
request.waitForActiveShards(2);  //Sets the number of shard copies that must be active before proceeding with the update operation.
request.waitForActiveShards(ActiveShardCount.ALL); //Number of shard copies provided as a ActiveShardCount: can be ActiveShardCount.ALL, ActiveShardCount.ONE or ActiveShardCount.DEFAULT (default)
```

```
String[] includes = new String[]{"updated", "r*"};
String[] excludes = Strings.EMPTY_ARRAY;
request.fetchSource(new FetchSourceContext(true, includes, excludes)); // Configure source inclusion for specific fields
```

```
String[] includes = Strings.EMPTY_ARRAY;
String[] excludes = new String[]{"updated"};
request.fetchSource(new FetchSourceContext(true, includes, exclu
des)); //Configure source exclusion for specific fields
```

同步执行

```
UpdateResponse updateResponse = client.update(request);
```

异步执行

```
client.updateAsync(request, new ActionListener<UpdateResponse>()
 {
    @Override
    public void onResponse(UpdateResponse updateResponse) {
        //    Called when the execution is successfully complete
d. The response is provided as an argument.
    }
    @Override
    public void onFailure(Exception e) {
        // Called in case of failure. The raised exception is pr
ovided as an argument.
    }
});
```

更新响应

返回的UpdateResponse允许获取执行操作的相关信息，如下所示：

```
String index = updateResponse.getIndex();
String type = updateResponse.getType();
String id = updateResponse.getId();
long version = updateResponse.getVersion();
if (updateResponse.getResult() == DocWriteResponse.Result.CREATE
D) {
//Handle the case where the document was created for the first t
ime (upsert)
} else if (updateResponse.getResult() == DocWriteResponse.Result
.UPDATED) {
//    Handle the case where the document was updated
} else if (updateResponse.getResult() == DocWriteResponse.Result
.DELETED) {
//Handle the case where the document was deleted
} else if (updateResponse.getResult() == DocWriteResponse.Result
.NOOP) {
//Handle the case where the document was not impacted by the upd
ate, ie no operation (noop) was executed on the document
}
```

当通过 fetchSource 方法在UpdateRequest 里设置了启用接收源，相应对象将包含被更新的文档源：

```
GetResult result = updateResponse.getGetResult(); //Retrieve the
 updated document as a GetResult
if (result.isExists()) {
    String sourceAsString = result.sourceAsString(); //Retrieve
the source of the updated document as a String
    Map<String, Object> sourceAsMap = result.sourceAsMap(); //Re
trieve the source of the updated document as a Map<String, Objec
t>
    byte[] sourceAsBytes = result.source(); //Retrieve the sourc
e of the updated document as a byte[]
} else {
    //Handle the scenario where the source of the document is no
t present in the response (this is the case by default)
}
```

这也可以用了检查分片故障：

```
ReplicationResponse.ShardInfo shardInfo = updateResponse.getShar
dInfo();
if (shardInfo.getTotal() != shardInfo.getSuccessful()) {
    //Handle the situation where number of successful shards is
less than total shards
}
if (shardInfo.getFailed() > 0) {
    for (ReplicationResponse.ShardInfo.Failure failure : shardIn
fo.getFailures()) {
        String reason = failure.reason(); //Handle the potential
 failures
    }
}
```

当对一个不存在的文档执行 UpdateRequest 时，响应将包含 404 状态码，并抛出一个需要如下所示处理的 ElasticsearchException 异常：

```
UpdateRequest request = new UpdateRequest("posts", "type", "does
_not_exist").doc("field", "value");
try {
    UpdateResponse updateResponse = client.update(request);
} catch (ElasticsearchException e) {
    if (e.status() == RestStatus.NOT_FOUND) {
        // 处理由于文档不存在导致的异常。
    }
}
```

如果有文档版本冲突，也会抛出 ElasticsearchException:

```
UpdateRequest request = new UpdateRequest("posts", "doc", "1")
        .doc("field", "value")
        .version(1);
try {
    UpdateResponse updateResponse = client.update(request);
} catch(ElasticsearchException e) {
    if (e.status() == RestStatus.CONFLICT) {
        // 表明异常是由于返回来了版本冲突错误导致的
    }
}
```

Bulk API

注意： Java高级REST客户端提供批量处理器来协助大量请求

Bulk 请求

BulkRequest 可以被用在使用单个请求执行多个 索引，更新 和/或 删除 操作的情况下。

它要求至少要一个操作被添加到 Bulk 请求上：

```
BulkRequest request = new BulkRequest();  // 创建 BulkRequest
request.add(new IndexRequest("posts", "doc", "1")  // 添加第一个 I
ndexRequest 到 Bulk 请求上， 参看 [Index API](https://www.elastic.
co/guide/en/elasticsearch/client/java-rest/current/java-rest-hig
h-document-index.html) 获取更多关于如何构建 IndexRequest 的信息.
        .source(XContentType.JSON,"field", "foo"));
request.add(new IndexRequest("posts", "doc", "2")   // 添加第二个
IndexRequest
        .source(XContentType.JSON,"field", "bar"));
request.add(new IndexRequest("posts", "doc", "3")  // 添加第三个 I
ndexRequest
        .source(XContentType.JSON,"field", "baz"));
```

> 警告: Bulk API仅支持以JSON或SMILE编码的文档。 以任何其他格式提供文件将导致错误。

不同的操作类型也可以添加到同一个BulkRequest中：

```
BulkRequest request = new BulkRequest();
request.add(new DeleteRequest("posts", "doc", "3")); //
Adds a DeleteRequest to the BulkRequest. See Delete API for more
 information on how to build DeleteRequest.
request.add(new UpdateRequest("posts", "doc", "2")
        .doc(XContentType.JSON,"other", "test")); //
Adds an UpdateRequest to the BulkRequest. See Update API for mor
e information on how to build UpdateRequest.
request.add(new IndexRequest("posts", "doc", "4")   //Adds an In
dexRequest using the SMILE format
        .source(XContentType.JSON,"field", "baz"));
```

可选参数

提供下列可选参数：

```
request.timeout(TimeValue.timeValueMinutes(2));
request.timeout("2m");
Timeout to wait for the bulk request to be performed as a TimeVa
lue
Timeout to wait for the bulk request to be performed as a String
request.setRefreshPolicy(WriteRequest.RefreshPolicy.WAIT_UNTIL);
request.setRefreshPolicy("wait_for");

Refresh policy as a WriteRequest.RefreshPolicy instance
Refresh policy as a String
request.waitForActiveShards(2);
request.waitForActiveShards(ActiveShardCount.ALL);
Sets the number of shard copies that must be active before proce
eding with the index/update/delete operations.
Number of shard copies provided as a ActiveShardCount: can be Ac
tiveShardCount.ALL, ActiveShardCount.ONE or ActiveShardCount.DEF
AULT (default)
```

同步执行

```
BulkResponse bulkResponse = client.bulk(request);
```

异步执行

```
client.bulkAsync(request, new ActionListener<BulkResponse>() {
    @Override
    public void onResponse(BulkResponse bulkResponse) {
        //Called when the execution is successfully completed. T
he response is provided as an argument and contains a list of in
dividual results for each operation that was executed. Note that
 one or more operations might have failed while the others have
been successfully executed.
    }
    @Override
    public void onFailure(Exception e) {
        //Called when the whole BulkRequest fails. In this case
the raised exception is provided as an argument and no operation
 has been executed.
    }
});
```

Bulk 响应

返回的BulkResponse包含有关执行操作的信息，并允许对每个结果进行迭代，如下所示：

```
for (BulkItemResponse bulkItemResponse : bulkResponse) { //迭代所有操作的结果
    DocWriteResponse itemResponse = bulkItemResponse.getResponse(); //Retrieve the response of the operation (successful or not), can be IndexResponse, UpdateResponse or DeleteResponse which can all be seen as DocWriteResponse instances
    if (bulkItemResponse.getOpType() == DocWriteRequest.OpType.INDEX
            || bulkItemResponse.getOpType() == DocWriteRequest.OpType.CREATE) {
                //    Handle the response of an index operation
        IndexResponse indexResponse = (IndexResponse) itemResponse;
    } else if (bulkItemResponse.getOpType() == DocWriteRequest.OpType.UPDATE) {
        //Handle the response of a update operation
        UpdateResponse updateResponse = (UpdateResponse) itemResponse;
    } else if (bulkItemResponse.getOpType() == DocWriteRequest.OpType.DELETE) {
        //    Handle the response of a delete operation
        DeleteResponse deleteResponse = (DeleteResponse) itemResponse;
    }
}
```

批量响应提供了一种快速检查一个或多个操作是否失败的方法：

```
if (bulkResponse.hasFailures()) { // 只要有一个操作失败了，这个方法就返回 true
}
```

在这种情况下，有必要迭代所有运算结果，以检查操作是否失败，如果是，则检索相应的故障：

```
for (BulkItemResponse bulkItemResponse : bulkResponse) {
    if (bulkItemResponse.isFailed()) { //Indicate if a given ope
ration failed
        BulkItemResponse.Failure failure = bulkItemResponse.getF
ailure(); //Retrieve the failure of the failed operation
    }
}
```

Bulk 处理器

BulkProcessor通过提供允许索引/更新/删除操作在添加到处理器时透明执行的实用程序类来简化Bulk API的使用。

为了执行请求，BulkProcessor需要3个组件：

- RestHighLevelClient

这个客户端用来执行 BulkRequest 并接收 BulkResponse 。

- BulkProcessor.Listener

这个监听器会在每个 BulkRequest 执行之前和之后被调用，或者当 BulkRequest 失败时调用。

- ThreadPool

BulkRequest执行是使用这个池的线程完成的，允许BulkProcessor以非阻塞的方式工作，并允许在批量请求执行的同时接受新的索引/更新/删除请求。

然后 BulkProcessor.Builder 类可以被用来构建新的 BulkProcessor ：

```
ThreadPool threadPool = new ThreadPool(settings); // 使用已有的 Se
ttings 对象创建 ThreadPool。
BulkProcessor.Listener listener = new BulkProcessor.Listener() {
 // 创建 BulkProcessor.Listener
    @Override
    public void beforeBulk(long executionId, BulkRequest request
) {
        //This method is called before each execution of a BulkR
equest
    }
    @Override
    public void afterBulk(long executionId, BulkRequest request,
 BulkResponse response) {
        //    This method is called after each execution of a Bu
lkRequest
    }
    @Override
    public void afterBulk(long executionId, BulkRequest request,
 Throwable failure) {
        //This method is called when a BulkRequest failed
    }
};
BulkProcessor bulkProcessor = new BulkProcessor.Builder(client::
bulkAsync, listener, threadPool)
        .build(); // 通过调用 BulkProcessor.Builder 的 build() 方
法创建 BulkProcessor。 RestHighLevelClient.bulkAsync() 将被用来执行
 BulkRequest。
```

BulkProcessor.Builder 提供了方法来配置 BulkProcessor 应该如何处理请求的执行：

```
BulkProcessor.Builder builder = new BulkProcessor.Builder(client
::bulkAsync, listener, threadPool);
builder.setBulkActions(500);  //Set when to flush a new bulk req
uest based on the number of actions currently added (defaults to
 1000, use -1 to disable it)
builder.setBulkSize(new ByteSizeValue(1L, ByteSizeUnit.MB));  //
    Set when to flush a new bulk request based on the size of ac
tions currently added (defaults to 5Mb, use -1 to disable it)
builder.setConcurrentRequests(0); //Set the number of concurrent
 requests allowed to be executed (default to 1, use 0 to only al
low the execution of a single request)
builder.setFlushInterval(TimeValue.timeValueSeconds(10L)); //
 Set a flush interval flushing any BulkRequest pending if the in
terval passes (defaults to not set)
builder.setBackoffPolicy(BackoffPolicy.constantBackoff(TimeValue
.timeValueSeconds(1L), 3)); //Set a constant back off policy tha
t initially waits for 1 second and retries up to 3 times. See Ba
ckoffPolicy.noBackoff(), BackoffPolicy.constantBackoff() and Bac
koffPolicy.exponentialBackoff() for more options.
```

一旦创建了BulkProcessor，可以向其添加请求：

```
IndexRequest one = new IndexRequest("posts", "doc", "1").
        source(XContentType.JSON, "title", "In which order are m
y Elasticsearch queries executed?");
IndexRequest two = new IndexRequest("posts", "doc", "2")
        .source(XContentType.JSON, "title", "Current status and
upcoming changes in Elasticsearch");
IndexRequest three = new IndexRequest("posts", "doc", "3")
        .source(XContentType.JSON, "title", "The Future of Feder
ated Search in Elasticsearch");
bulkProcessor.add(one);
bulkProcessor.add(two);
bulkProcessor.add(three);
```

这些请求将由 BulkProcessor 执行，它负责为每个批量请求调用 BulkProcessor.Listener 。 监听器提供方法接收 BulkResponse 和 BulkResponse ：

```
BulkProcessor.Listener listener = new BulkProcessor.Listener() {
    @Override
    public void beforeBulk(long executionId, BulkRequest request
) {
        int numberOfActions = request.numberOfActions(); //Calle
d before each execution of a BulkRequest, this method allows to
know the number of operations that are going to be executed with
in the BulkRequest
        logger.debug("Executing bulk [{}] with {} requests", exe
cutionId, numberOfActions);
    }
    @Override
    public void afterBulk(long executionId, BulkRequest request,
 BulkResponse response) {
        if (response.hasFailures()) {
            //在每个 BulkRequest 执行之后调用，此方法允许获知 BulkResp
onse 是否包含错误
            logger.warn("Bulk [{}] executed with failures", exec
utionId);
        } else {
            logger.debug("Bulk [{}] completed in {} milliseconds
", executionId, response.getTook().getMillis());
        }
    }
    @Override
    public void afterBulk(long executionId, BulkRequest request,
 Throwable failure) {
        logger.error("Failed to execute bulk", failure); //如果 B
ulkRequest 执行失败则调用，此方法可获知失败情况。
    }
};
```

一旦将所有请求都添加到BulkProcessor，其实例需要使用两种可用的关闭方法之一关闭。

一旦所有请求都被添加到了 BulkProcessor, 它的实例就需要使用两种可用的关闭方法之一进行关闭。

awaitClose() 可以被用来等待到所有请求都被处理，或者到指定的等待时间：

```
boolean terminated = bulkProcessor.awaitClose(30L, TimeUnit.SECO
NDS);
```

如果所有批量请求完成，则该方法返回 true ，如果在完成所有批量请求之前等待时间过长，则返回 false 。

close() 方法可以被用来立即关闭 BulkProcessor ：

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

在关闭处理器之前，两个方法都会刷新已经被天教导处理器的请求，并禁止添加任何新的请求。

Search API

搜索请求

SearchRequest用于与搜索文档，聚合，建议有关的任何操作，并且还提供了在生成的文档上请求突出显示的方法。 在最基本的形式中，我们可以向请求添加一个查询：

```
SearchRequest searchRequest = new SearchRequest(); //穿件SeachRequest，Without arguments this runs against all indices.
SearchSourceBuilder searchSourceBuilder = new SearchSourceBuilder();// 大多数的搜索参数被添加到 SearchSourceBuilder 。它为每个进入请求体的每个东西都提供 setter 方法。
searchSourceBuilder.query(QueryBuilders.matchAllQuery());  // 添加一个 match_all 查询到 searchSourceBuilder 。
```

可选参数

我们先看一下SearchRequest的一些可选参数：

```
SearchRequest searchRequest = new SearchRequest("posts"); // 限制请求到某个索引上
searchRequest.types("doc");  // 限制请求的类别
```

还有一些其他有趣的可选参数：

```
searchRequest.routing("routing");  // 设置路由参数、
searchRequest.indicesOptions(IndicesOptions.lenientExpandOpen()); // 设置IndicesOptions控制如何解析不可用索引以及扩展通配符表达式
searchRequest.preference("_local"); //使用首选参数，例如，执行搜索优先选择本地分片。 默认值是随机化分片。
```

使用 SearchSourceBuilder

可以在SearchSourceBuilder上设置大多数控制搜索行为的选项，其中包含或多或少相当于 Rest API 的搜索请求正文中的选项。

以下是一些常见选项的示例：

```
SearchSourceBuilder sourceBuilder = new SearchSourceBuilder(); //使用默认选项创建 SearchSourceBuilder 。
sourceBuilder.query(QueryBuilders.termQuery("user", "kimchy")); //设置查询对象。可以使任何类型的 QueryBuilder
sourceBuilder.from(0); //设置from选项，确定要开始搜索的结果索引。 默认为0。
sourceBuilder.size(5); //设置大小选项，确定要返回的搜索匹配数。 默认为10。
sourceBuilder.timeout(new TimeValue(60, TimeUnit.SECONDS)); //设置一个可选的超时时间，用于控制搜索允许的时间。
```

构建查询

搜索查询可以使用 QueryBuilder 对象创建。 对于Elasticsearch的 Query DSL 支持的每个搜索查询类型，都存在QueryBuilder。 QueryBuilder 可以使用它的构造器创建：

```
//创建一个字段“user”与文本“kimchy”相匹配的的全文匹配查询。
MatchQueryBuilder matchQueryBuilder = new MatchQueryBuilder("user", "kimchy");
```

创建后，QueryBuilder对象提供了配置搜索查询选项的方法：

```
matchQueryBuilder.fuzziness(Fuzziness.AUTO); //在匹配查询上启用模糊匹配
matchQueryBuilder.prefixLength(3);  //在匹配查询上设置前缀长度
matchQueryBuilder.maxExpansions(10); //设置最大扩展选项以控制查询的模糊过程
```

QueryBuilder 对象也可以使用 QueryBuilders 工具类创建。这个类提供了使用链式编程风格的辅助方法来创建 QueryBuilder 对象：

```
QueryBuilder matchQueryBuilder = QueryBuilders.matchQuery("user"
, "kimchy")
                                                .fuzziness(Fuzzi
ness.AUTO)
                                                .prefixLength(3)
                                                .maxExpansions(1
0);
```

无论用于创建它的方法如何， QueryBuilder 对象必须添加到 SearchSourceBuilder 中，如下所示：

```
searchSourceBuilder.query(matchQueryBuilder);
```

“构建查询”页面列出了所有可用的搜索查询及其对应的QueryBuilder对象和QueryBuilders辅助方法。

指定排序

SearchSourceBuilder允许添加一个或多个SortBuilder实例。 有四个特殊的实现（Field-，Score-，GeoDistance-和ScriptSortBuilder）。

```
sourceBuilder.sort(new ScoreSortBuilder().order(SortOrder.DESC))
;  // 按_score降序排序（默认值）
sourceBuilder.sort(new FieldSortBuilder("_uid").order(SortOrder.
ASC));  //也按_id字段排序升序
```

源过滤

默认情况下，搜索请求返回文档的内容，_source但像 Rest API 中的内容一样，您可以覆盖此行为。例如，您可以完全关闭 _source 检索：

```
sourceBuilder.fetchSource(false);
```

该方法还接受一个或多个通配符模式的数组，以便以更精细的方式控制哪些字段包含或排除：

```
String[] includeFields = new String[] {"title", "user", "innerOb
ject.*"};
String[] excludeFields = new String[] {"_type"};
sourceBuilder.fetchSource(includeFields, excludeFields);
```

请求高亮

突出显示搜索结果可以通过设置来实现HighlightBuilder的 SearchSourceBuilder。可以通过向HighlightBuilder.Fielda 添加一个或多个实例来为每个字段定义不同的突出显示行为HighlightBuilder。

```
SearchSourceBuilder searchSourceBuilder = new SearchSourceBuilde
r();
HighlightBuilder highlightBuilder = new HighlightBuilder();//Cre
ates a new HighlightBuilder
HighlightBuilder.Field highlightTitle =
        new HighlightBuilder.Field("title");//Create a field hig
hlighter for the title field
highlightTitle.highlighterType("unified");  //Set the field high
lighter type
highlightBuilder.field(highlightTitle);  //Add the field highlig
hter to the highlight builder
HighlightBuilder.Field highlightUser = new HighlightBuilder.Fiel
d("user");
highlightBuilder.field(highlightUser);
searchSourceBuilder.highlighter(highlightBuilder);
```

有很多选项，这在Rest API文档中有详细的介绍。Rest API参数（例如，pre_tags）通常由具有相似名称的 setter（例如#preTags（String ...））更改。

随后可以从 SearchResponse 中检索突出显示的文本片段。

请求聚合

可以通过首先创建适当的集合AggregationBuilder然后在其上设置集合来将搜索添加到搜索结果中 SearchSourceBuilder。在下面的示例中，我们terms在公司名称上创建一个聚合，其中包含公司员工平均年龄的子聚合：

```
SearchSourceBuilder searchSourceBuilder = new SearchSourceBuilde
r();
TermsAggregationBuilder aggregation = AggregationBuilders.terms(
"by_company")
        .field("company.keyword");
aggregation.subAggregation(AggregationBuilders.avg("average_age"
)
        .field("age"));
searchSourceBuilder.aggregation(aggregation);
```

“构建聚合”页面提供了所有可用聚合以及其相应AggregationBuilder对象和AggregationBuilders帮助方法的列表。

后面我们将看到如何访问聚合的 SearchResponse。

请求建议

要向搜索请求添加建议，请使用SuggestionBuilder从SuggestBuilders工厂类轻松访问的其中一个实现。建议构建者需要添加到顶层SuggestBuilder，本身可以设置在顶层SearchSourceBuilder。

```
SearchSourceBuilder searchSourceBuilder = new SearchSourceBuilde
r();
SuggestionBuilder termSuggestionBuilder =
    SuggestBuilders.termSuggestion("user").text("kmichy"); //
Creates a new TermSuggestionBuilder for the user field and the t
ext kmichy
SuggestBuilder suggestBuilder = new SuggestBuilder();
suggestBuilder.addSuggestion("suggest_user", termSuggestionBuild
er); //Adds the suggestion builder and names it suggest_user
searchSourceBuilder.suggest(suggestBuilder);
```

后面我们将看到如何从 SearchResponse 获取建议。

自定义查询和聚合

该配置文件API可用于简档查询和聚集的执行针对特定搜索请求。为了使用它，配置文件标志必须设置为true SearchSourceBuilder：

```
SearchSourceBuilder searchSourceBuilder = new SearchSourceBuilde
r();
searchSourceBuilder.profile(true);
```

一旦SearchRequest执行，相应的SearchResponse将 包含分析结果。

同步执行

SearchRequest以下列方式执行时，客户端SearchResponse在继续执行代码之前等待返回：

```
SearchResponse searchResponse = client.search(searchRequest);
```

异步执行

```
client.searchAsync(searchRequest, new ActionListener<SearchRespo
nse>() {
    @Override
    public void onResponse(SearchResponse searchResponse) {
        //执行成功完成时调用。
    }
    @Override
    public void onFailure(Exception e) {
        //当整个SearchRequest失败时调用。
    }
});
```

搜索响应

通过执行搜索返回的 SearchResponse 提供了关于搜索执行本身以及对返回的文档的访问的详细信息。 首先，有关于请求执行本身的有用信息，如HTTP状态代码，执行时间或请求提前终止或超时：

```
RestStatus status = searchResponse.status();
TimeValue took = searchResponse.getTook();
Boolean terminatedEarly = searchResponse.isTerminatedEarly();
boolean timedOut = searchResponse.isTimedOut();
```

其次，响应还通过提供关于搜索影响的分片总数以及成功与不成功的分片的统计信息，提供关于分片级别执行的信息。 可能的故障也可以通过遍历ShardSearchFailures上的数组进行迭代来处理，如下例所示：

```
int totalShards = searchResponse.getTotalShards();
int successfulShards = searchResponse.getSuccessfulShards();
int failedShards = searchResponse.getFailedShards();
for (ShardSearchFailure failure : searchResponse.getShardFailure
s()) {
    // 故障应该在这里被处理
}
```

检索 SearchHits

要访问返回的文档，我们需要首先得到响应中包含的 SearchHits ：

```
SearchHits hits = searchResponse.getHits();
```

将SearchHits提供命中的所有全局信息，比如命中总数或最大分数：

```
long totalHits = hits.getTotalHits();
float maxScore = hits.getMaxScore();
```

嵌套在 SearchHits 的各个搜索结果可以被迭代访问：

```
SearchHit[] searchHits = hits.getHits();
for (SearchHit hit : searchHits) {
    // do something with the SearchHit
}
```

SearchHit可以访问基本信息，如索引，类型，文档ID 以及每个搜索匹配的分数：

```
String index = hit.getIndex();
String type = hit.getType();
String id = hit.getId();
float score = hit.getScore();
```

此外，它可以让您将文档源作为简单的JSON-String或键/值对的映射。 在该映射中，字段名为键，含字段值为值。 多值字段作为对象的列表返回，嵌套对象作为另一个键/值映射。 这些情况需要相应地执行：

```
String sourceAsString = hit.getSourceAsString();
Map<String, Object> sourceAsMap = hit.getSourceAsMap();
String documentTitle = (String) sourceAsMap.get("title");
List<Object> users = (List<Object>) sourceAsMap.get("user");
Map<String, Object> innerObject = (Map<String, Object>) sourceAs
Map.get("innerObject");
Retrieving Highlighting
```

如果请求，可以从SearchHit结果中的每一个中检索突出显示的文本片段。命中对象提供对HighlightField实例的字段名称映射的访问，每个实例都包含一个或多个突出显示的文本片段：

```
SearchHits hits = searchResponse.getHits();
for (SearchHit hit : hits.getHits()) {
    Map<String, HighlightField> highlightFields = hit.getHighlig
htFields();
    HighlightField highlight = highlightFields.get("title"); //
获取该title领域 的突出显示
    Text[] fragments = highlight.fragments();  //获取包含突出显示的
字段内容的一个或多个片段
    String fragmentString = fragments[0].string();
}
```

检索聚合

可以SearchResponse通过首先获取聚合树的根，Aggregations对象，然后通过名称获取聚合来检索聚合。

```
Aggregations aggregations = searchResponse.getAggregations();
Terms byCompanyAggregation = aggregations.get("by_company"); //G
et the by_company terms aggregation
Bucket elasticBucket = byCompanyAggregation.getBucketByKey("Elas
tic"); //
Avg averageAge = elasticBucket.getAggregations().get("average_ag
e"); //Get the average_age sub-aggregation from that bucket
double avg = averageAge.getValue();
```

请注意，如果按名称访问聚合，则需要根据您请求的聚合类型指定聚合接口，否则将抛出 ClassCastException ：

```
//This will throw an exception because "by_company" is a terms a
ggregation but we try to retrieve it as a range aggregation
Range range = aggregations.get("by_company");
```

也可以以聚合名称键入的映射来访问所有聚合。在这种情况下，转换为适当聚合接口需要明确发生：

```
Map<String, Aggregation> aggregationMap = aggregations.getAsMap(
);
Terms companyAggregation = (Terms) aggregationMap.get("by_compan
y");
```

还有getter 方法将所有顶级聚合作为列表返回：

```
List<Aggregation> aggregationList = aggregations.asList();
```

最后但并非最不重要的是，您可以迭代所有聚合，然后根据其类型决定如何进一步处理它们：

```
for (Aggregation agg : aggregations) {
    String type = agg.getType();
    if (type.equals(TermsAggregationBuilder.NAME)) {
        Bucket elasticBucket = ((Terms) agg).getBucketByKey("Ela
stic");
        long numberOfDocs = elasticBucket.getDocCount();
    }
}
```

检索建议

要从SearchResponse返回建议，请使用Suggest对象作为入口点，然后检索嵌套的建议对象：

```
Suggest suggest = searchResponse.getSuggest();// Use the Suggest
 class to access suggestions
TermSuggestion termSuggestion = suggest.getSuggestion("suggest_u
ser");// Suggestions can be retrieved by name. You need to assig
n them to the correct type of Suggestion class (here TermSuggest
ion), otherwise a ClassCastException is thrown
for (TermSuggestion.Entry entry : termSuggestion.getEntries()) {
// Iterate over the suggestion entries
    for (TermSuggestion.Entry.Option option : entry) {// Iterate
 over the options in one entry
        String suggestText = option.getText().string();
    }
}
```

检索分析结果

从SearchResponse使用该getProfileResults()方法检索分析结果。此方法返回Map包含执行中ProfileShardResult涉及的每个分片的对象 SearchRequest。ProfileShardResult存储在Map使用唯一地标识分析结果对应的分片的密钥。

这是一个示例代码，显示如何遍历每个分片的所有分析结果：

```
Map<String, ProfileShardResult> profilingResults = searchRespons
e.getProfileResults(); //Retrieve the Map of ProfileShardResult
from the SearchResponse
for (Map.Entry<String, ProfileShardResult> profilingResult : pro
filingResults.entrySet()) {  //Profiling results can be retrieve
d by shard's key if the key is known, otherwise it might be simp
ler to iterate over all the profiling results
    String key = profilingResult.getKey();//Retrieve the key tha
t identifies which shard the ProfileShardResult belongs to
    ProfileShardResult profileShardResult = profilingResult.getV
alue();//Retrieve the ProfileShardResult for the given shard
}
```

ProfileShardResult对象本身包含一个或多个查询配置文件结果，一个针对基本的Lucene索引执行的每个查询：

```
List<QueryProfileShardResult> queryProfileShardResults = profile
ShardResult.getQueryProfileResults(); // 检索 QueryProfileShardRe
sult 列表
for (QueryProfileShardResult queryProfileResult : queryProfileSh
ardResults) {
//迭代每个 QueryProfileShardResult
}
```

每个都QueryProfileShardResult可以访问详细的查询树执行，作为ProfileResult对象列表返回 ：

```
for (ProfileResult profileResult : queryProfileResult.getQueryRe
sults()) {//迭代配置文件结果
    String queryName = profileResult.getQueryName(); //检索Lucene
查询的名称
    long queryTimeInMillis = profileResult.getTime(); //检索在执行
Lucene查询时花费的时间
    List<ProfileResult> profiledChildren = profileResult.getProf
iledChildren();//
检索子查询的配置文件结果（如果有）
}
```

Rest API文档包含有关查询分析信息描述的Profiling Queries的更多信息。

该 QueryProfileShardResult 还可以访问了Lucene的收藏家的分析信息：

```
CollectorResult collectorResult = queryProfileResult.getCollecto
rResult();  //检索Lucene收集器的分析结果
String collectorName = collectorResult.getName();  //检索Lucene
的名字
Long collectorTimeInMillis = collectorResult.getTime();//以毫秒计
算的时间用于执行Lucene集合
List<CollectorResult> profiledChildren = collectorResult.getProf
iledChildren();//
```

检索子集合的个人资料结果（如果有） Rest API文档包含有关Lucene收集器的分析信息的更多信息 。

以与查询树执行非常相似的方式，QueryProfileShardResult对象可以访问详细的聚合树执行：

```
AggregationProfileShardResult aggsProfileResults = profileShardR
esult.getAggregationProfileResults(); // 检索 AggregationProfileS
hardResult
for (ProfileResult profileResult : aggsProfileResults.getProfile
Results()) { //迭代聚合配置文件结果
    String aggName = profileResult.getQueryName(); //Retrieve th
e type of the aggregation (corresponds to Java class used to exe
cute the aggregation)
    long aggTimeInMillis = profileResult.getTime();//Retrieve th
e time in millis spent executing the Lucene collector
    List<ProfileResult> profiledChildren = profileResult.getProf
iledChildren(); //Retrieve the profile results for the sub-aggre
gations (if any)
}
```

Rest API文档包含更多关关于 Profiling Aggregations 的信息

Search Scroll API

Scroll API可用于从搜索请求中检索大数量的结果。

为了使用滚动，需要按照给定的顺序执行以下步骤。

初始化搜索滚动上下文

包含一个 scroll 参数的初始化搜索请求必须通过执行 Search API 初始化滚动回话。 在处理此SearchRequest时，Elasticsearch将检测滚动参数的存在，并使搜索上下文保持相应的时间间隔。

```
SearchRequest searchRequest = new SearchRequest("posts");
SearchSourceBuilder searchSourceBuilder = new SearchSourceBuilde
r();
searchSourceBuilder.query(matchQuery("title", "Elasticsearch"));
searchSourceBuilder.size(size); //Create the SearchRequest and i
ts corresponding SearchSourceBuilder. Also optionally set the si
ze to control how many results to retrieve at a time.
searchRequest.source(searchSourceBuilder);
searchRequest.scroll(TimeValue.timeValueMinutes(1L)); // Set the
 scroll interval
SearchResponse searchResponse = client.search(searchRequest);
String scrollId = searchResponse.getScrollId(); // Read the retu
rned scroll id, which points to the search context that’s being
kept alive and will be needed in the following search scroll cal
l
SearchHits hits = searchResponse.getHits();  // Retrieve the fir
st batch of search hits
```

检索所有相关文档

其次， 接收到的滚动标识符必须被设置到下一个新的滚动间隔的SearchScrollRequest， 并通过 searchScroll 方法发送。Elasticsearch会使用新的滚动标识符返回另一批结果。 然后可以在随后的SearchScrollRequest中使用此新

的滚动标识符来检索下一批结果，等等。应该循环重复此过程，直到不再返回结果，这意味着滚动已经用尽，并且已经检索到所有匹配的文档。

```
SearchScrollRequest scrollRequest = new SearchScrollRequest(scro
llId);  //Create the SearchScrollRequest by setting the required
 scroll id and the scroll interval
scrollRequest.scroll(TimeValue.timeValueSeconds(30));
SearchResponse searchScrollResponse = client.searchScroll(scroll
Request);
scrollId = searchScrollResponse.getScrollId();   //    Read the
new scroll id, which points to the search context that’s being k
ept alive and will be needed in the following search scroll call
hits = searchScrollResponse.getHits(); //Retrieve another batch
of search hits <4>
assertEquals(3, hits.getTotalHits());
assertEquals(1, hits.getHits().length);
assertNotNull(scrollId);
```

清除滚动上下文

最后，可以使用Clear Scroll API删除最后一个滚动标识符，以释放搜索上下文。 当滚动到期时，会自动发生，但最佳实践是当滚动会话结束后尽快释放资源。

可选参数

构建 SearchScrollRequest 是可以选择使用以下参数：

```
scrollRequest.scroll(TimeValue.timeValueSeconds(60L));  //     S
croll interval as a TimeValue
scrollRequest.scroll("60s"); // Scroll interval as a String
```

同步执行

```
SearchResponse searchResponse = client.searchScroll(scrollReques
t);
```

异步执行

```
client.searchScrollAsync(scrollRequest, new ActionListener<Searc
hResponse>() {
    @Override
    public void onResponse(SearchResponse searchResponse) {
        // 当执行成功的时候调用，响应对象以参数的形式传入
    }
    @Override
    public void onFailure(Exception e) {
        // 失败时调用，异常以参数形式传入
    }
});
```

响应

与 Search API 一样，滚动搜索 API 也返回一个 SearchResponse 对象

完整示例

下面是一个滚动搜索的完整示例：

```
final Scroll scroll = new Scroll(TimeValue.timeValueMinutes(1L))
;
SearchRequest searchRequest = new SearchRequest("posts");
searchRequest.scroll(scroll);
SearchSourceBuilder searchSourceBuilder = new SearchSourceBuilde
r();
searchSourceBuilder.query(matchQuery("title", "Elasticsearch"));
searchRequest.source(searchSourceBuilder);
SearchResponse searchResponse = client.search(searchRequest); //
 通过发送初始化 SearchRequest 来初始化搜索上下文
String scrollId = searchResponse.getScrollId();
SearchHit[] searchHits = searchResponse.getHits().getHits();
while (searchHits != null && searchHits.length > 0) { //在一个循环
中通过调用 Search Scroll api 检索所有搜索命中结果，知道没有文档返回为止。
    //创建一个新的SearchScrollRequest，持有最近一次返回的滚动标识符和滚
动间隔
    SearchScrollRequest scrollRequest = new SearchScrollRequest(
scrollId);
    scrollRequest.scroll(scroll);
    searchResponse = client.searchScroll(scrollRequest);
    scrollId = searchResponse.getScrollId();
    searchHits = searchResponse.getHits().getHits();
//处理返回的搜索结果
}
ClearScrollRequest clearScrollRequest = new ClearScrollRequest()
; //一旦滚动完成，清除滚动上下文
clearScrollRequest.addScrollId(scrollId);
ClearScrollResponse clearScrollResponse = client.clearScroll(cle
arScrollRequest);
boolean succeeded = clearScrollResponse.isSucceeded();
```

Clear Scroll API

使用 Search Scroll API 的搜索上下文在超过滚动时间时，会自动清除。但是，建议当不再需要使用 Clear Scroll API 后尽可能快的释放搜索上下文。

Clear Scroll 请求

ClearScrollRequest 可以像如下方式创建:

```
ClearScrollRequest request = new ClearScrollRequest();  // 创建一个新的 ClearScrollRequest
request.addScrollId(scrollId); // 添加一个滚动id到要清除的滚动标志列表里
https://www.elastic.co/guide/en/elasticsearch/client/java-rest/current/java-rest-high-clear-scroll.html
```

Info API

执行

集群信息可以通过 info() 方法被获取到：

```
MainResponse response = client.info();
```

响应

返回的 MainResponse 提供了有关集群的各种信息：

```
ClusterName clusterName = response.getClusterName(); // 获取包含集群名称信息的 ClusterName 对象
String clusterUuid = response.getClusterUuid(); // 获取集群的唯一标识符
String nodeName = response.getNodeName(); // 获取执行请求的节点的名称
Version version = response.getVersion(); // 获取已执行请求的节点版本
Build build = response.getBuild(); // 获取已执行请求的节点的构建信息
```

使用 Java 建造者

Java高级REST客户端依赖于 Elasticsearch 核心项目提供的不同类型的 Java Builders 对象，包括：

Query Builders The query builders are used to create the query to execute within a search request. There is a query builder for every type of query supported by the Query DSL. Each query builder implements the QueryBuilder interface and allows to set the specific options for a given type of query. Once created, the QueryBuilder object can be set as the query parameter of SearchSourceBuilder. The Search Request page shows an example of how to build a full search request using SearchSourceBuilder and QueryBuilder objects. The Building Search Queries page gives a list of all available search queries with their corresponding QueryBuilder objects and QueryBuilders helper methods.

Aggregation Builders Similarly to query builders, the aggregation builders are used to create the aggregations to compute during a search request execution. There is an aggregation builder for every type of aggregation (or pipeline aggregation) supported by Elasticsearch. All builders extend the AggregationBuilder class (or PipelineAggregationBuilderclass). Once created, AggregationBuilder objects can be set as the aggregation parameter of SearchSourceBuilder. There is a example of how AggregationBuilder objects are used with SearchSourceBuilder objects to define the aggregations to compute with a search query in Search Request page. The Building Aggregations page gives a list of all available aggregations with their corresponding AggregationBuilder objects and AggregationBuilders helper methods.

构建查询

此页面列出了在 QueryBuilders 实用程序类中所有可用的搜索查询及其对应的 QueryBuilder类名和帮助方法名称。
https://www.elastic.co/guide/en/elasticsearch/client/java-rest/current/java-rest-high-query-builders.html

构建聚合

https://www.elastic.co/guide/en/elasticsearch/client/java-rest/current/java-rest-high-aggregation-builders.html

迁移指南

本节介绍如何将现有代码从 TransportClient 迁移到使用 Elasticsearch 5.6.0 版本发布的新的 Java 高级 REST 客户端。

使用一个新的Java客户端的动机

自从第一次提交以来，现有的TransportClient已经成为Elasticsearch的一部分。 它是一个特殊的客户端，因为它使用传输协议与Elasticsearch进行通信，如果客户端的Elasticsearch实例与Elasticsearch实例的版本不同，则会导致兼容性问题。

我们在2016年发布了一个低级别的REST客户端，它基于众所周知的Apache HTTP客户端，并且允许在任何版本中使用HTTP与Elasticsearch集群进行通信。 最重要的是，我们发布了基于低级客户端的高级REST客户端，但是负责请求编组和响应解组。

如果您有兴趣了解更多关于这些变化的信息，我们写了一篇关于官方Elasticsearch Java客户端状态的博客文章。

条件

Java 1.8

怎样合并

调整现有代码以使用RestHighLevelClient而不是TransportClient需要以下步骤：

- 更新依赖关系
- 更新客户端初始化
- 更新应用程序代码

更新依赖

使用TransportClient的Java应用程序依赖于org.elasticsearch.client：transport artifact。 这个依赖关系必须被高级客户端的新依赖所取代。

“入门”页面显示了Maven和Gradle的典型配置，并介绍了高级客户端提供的依赖关系。