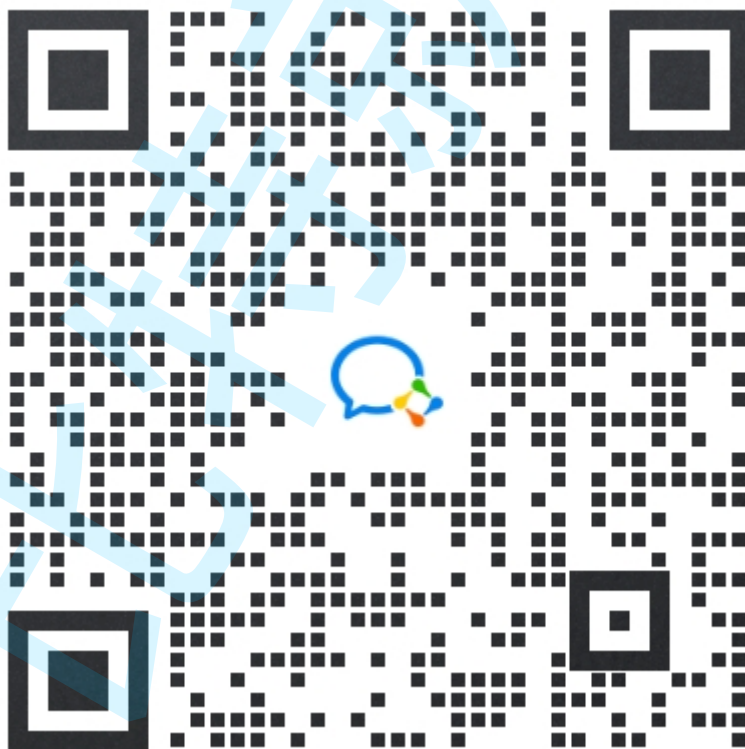


# ES C++客户端安装及使用

## 版权说明

本“比特就业课”项目（以下简称“本项目”）的所有内容，包括但不限于文字、图片、音频、视频、软件、程序、数据库、设计、布局、界面等，均由本项目的开发者或授权方拥有版权。我们鼓励个人学习者使用本项目进行学习和研究。在遵守相关法律法规的前提下，个人学习者可以下载、浏览、学习本项目的内容，并为了个人学习、研究或教学目的而使用其中的材料。但请注意，未经我们明确授权，个人学习者不得将本项目的内容用于任何商业目的，包括但不限于销售、转让、许可或以其他方式从中获利。此外，个人学习者也不得擅自修改、复制、传播、展示、表演或制作本项目内容的衍生作品。任何未经授权的使用均属侵权行为，我们将依法追究法律责任。如果您希望以其他方式使用本项目的内容，包括但不限于引用、转载、摘录、改编等，请事先与我们联系，获取书面授权。感谢您对“比特就业课”项目的关注与支持，我们将持续努力，为您提供更好的学习体验。特此说明。比特就业课版权所有方

对比特项目感兴趣，可以联系这个微信。



## 代码 & 板书链接

<https://gitee.com/bitedu-tech/cpp-chatsystem>

### 1. ES 介绍

Elasticsearch, 简称 ES, 它是个开源分布式搜索引擎, 它的特点有: 分布式, 零配置, 自动发现, 索引自动分片, 索引副本机制, restful 风格接口, 多数据源, 自动搜索负载等。它可以近乎实时的存储、检索数据; 本身扩展性很好, 可以扩展到上百台服务器, 处理 PB 级别的数据。es 也使用 Java 开发并使用 Lucene 作为其核心来实现所有索引和搜索的功能, 但是它的目的是通过简单的 RESTful API 来隐藏 Lucene 的复杂性, 从而让全文搜索变得简单。

Elasticsearch 是面向文档(document oriented)的, 这意味着它可以存储整个对象或文档(document)。然而它不仅仅是存储, 还会索引(index)每个文档的内容使之可以被搜索。在 Elasticsearch 中, 你可以对文档 (而非成行成列的数据) 进行索引、搜索、排序、过滤。

### 2. ES 安装

Shell

```
# 添加仓库秘钥
wget -qO - https://artifacts.elastic.co/GPG-KEY-elasticsearch |
sudo apt-key add -
# 上边的添加方式会导致一个 apt-key 的警告, 如果不想报警告使用下边这个
curl -s https://artifacts.elastic.co/GPG-KEY-elasticsearch | sudo
gpg --no-default-keyring --keyring gnupg-
ring:/etc/apt/trusted.gpg.d/icsearch.gpg --import

# 添加镜像源仓库
echo "deb https://artifacts.elastic.co/packages/7.x/apt stable
main" | sudo tee /etc/apt/sources.list.d/elasticsearch.list
# 更新软件包列表
sudo apt update
# 安装 es
sudo apt-get install elasticsearch=7.17.21
# 启动 es
sudo systemctl start elasticsearch
# 安装 ik 分词器插件
sudo /usr/share/elasticsearch/bin/elasticsearch-plugin install
https://get.infini.cloud/elasticsearch/analysis-ik/7.17.21
```

若 apt update 更新源报错:

SQL

```
dev@dev-host:~/workspace$ apt-key list
Warning: apt-key is deprecated. Manage keyring files in
trusted.gpg.d instead (see apt-key(8)).
/etc/apt/trusted.gpg      ubuntu 希望将 apt-key 放到
/etc/apt/trusted.gpg.d/下而不是这个文件中
-----
pub   rsa2048 2013-09-16 [SC]
      4609 5ACC 8548 582C 1A26  99A9 D27D 666C D88E 42B4  注意最后
      这 8 个字符
uid           [ unknown] Elasticsearch (Elasticsearch Signing Key)
<dev_ops@elasticsearch.org>
sub   rsa2048 2013-09-16 [E]
dev@dev-host:~$ sudo apt-key export D88E42B4 | sudo gpg --dearmor
-o /etc/apt/trusted.gpg.d/elasticsearch.gpg
```

完成后, 查看/etc/apt/trusted.gpg.d/, 应该已经将 apt-key 单独保存到目录下了

启动 es 的时候报错:

```
z@hcss-ecs-2618:~$ sudo systemctl start elasticsearch
[sudo] password for z:
Job for elasticsearch.service failed.
See "systemctl status elasticsearch.service" and "journalctl -xeu elasticsearch.service" for details.
```

解决办法:

Shell

# 调整 ES 虚拟内存, 虚拟内存默认最大映射数为 65530, 无法满足 ES 系统要求, 需要调整为 262144 以上

```
sysctl -w vm.max_map_count=262144
```

# 增加虚拟机内存配置

```
vim /etc/elasticsearch/jvm.options
```

# 新增如下内容

```
-Xms512m
```

```
-Xmx512m
```

```
## -Xms4g
## -Xmx4g
##
## See https://www.elastic.co/guide/en/elasticsearch/reference/7.17/heap-size.html
## for more information
##
#####

-Xms512m
-Xmx512m
```

- 启动 es

Shell

```
sudo systemctl start elasticsearch
```

- 查看 es 服务的状态

Shell

```
sudo systemctl status elasticsearch.service
```

```
z@hcss-ecs-2618:~$ sudo systemctl status elasticsearch.service
● elasticsearch.service - Elasticsearch
   Loaded: loaded (/lib/systemd/system/elasticsearch.service; disabled; vendor preset: enabled)
   Active: active (running) since Thu 2024-03-28 17:29:49 CST; 37s ago
     Docs: https://www.elastic.co
   Main PID: 46418 (java)
    Tasks: 73 (limit: 2006)
   Memory: 793.0M
      CPU: 1min 20.865s
   CGroup: /system.slice/elasticsearch.service
           └─46418 /usr/share/elasticsearch/jdk/bin/java -Xshare:auto -Des.networkaddress.cache.ttl=6
             46568 /usr/share/elasticsearch/modules/x-pack-ml/platform/linux-x86_64/bin/controller
```

- 验证 es 是否安装成功

Shell

```
curl -X GET "http://localhost:9200/"
```

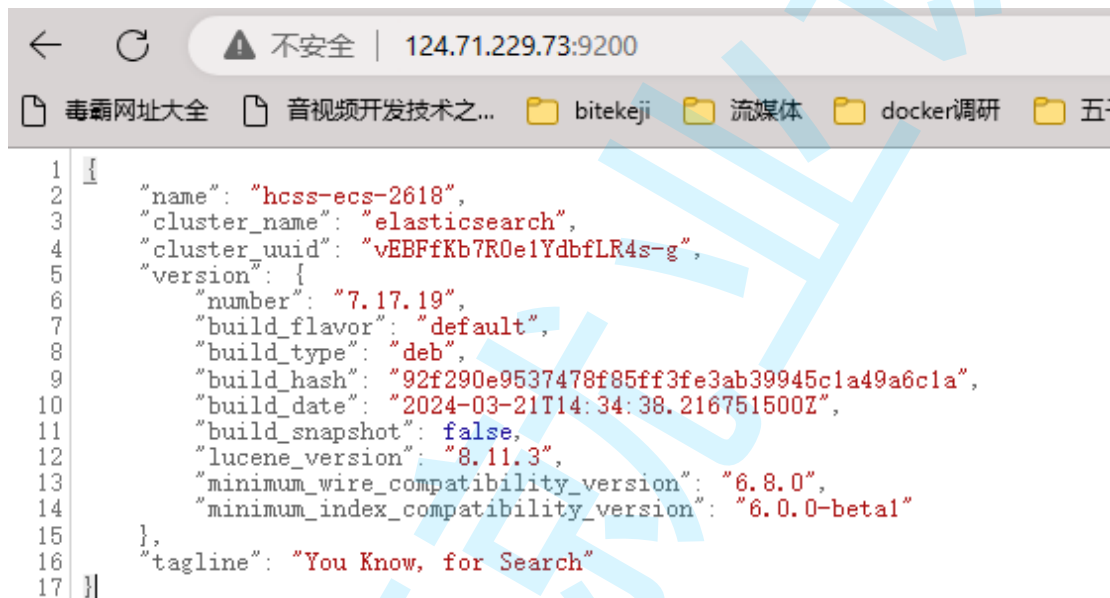
```
z@hcss-ecs-2618:~$ curl -X GET "http://localhost:9200/"
{
  "name" : "hcss-ecs-2618",
  "cluster_name" : "elasticsearch",
  "cluster_uuid" : "vEBFfKb7R0e1YdbfLR4s-g",
  "version" : {
    "number" : "7.17.19",
    "build_flavor" : "default",
    "build_type" : "deb",
    "build_hash" : "92f290e9537478f85ff3fe3ab39945c1a49a6c1a",
    "build_date" : "2024-03-21T14:34:38.216751500Z",
    "build_snapshot" : false,
    "lucene_version" : "8.11.3",
    "minimum_wire_compatibility_version" : "6.8.0",
    "minimum_index_compatibility_version" : "6.0.0-beta1"
  },
  "tagline" : "You Know, for Search"
}
```

- 设置外网访问：如果新配置完成的话，默认只能在本机进行访问。

```
Shell
vim /etc/elasticsearch/elasticsearch.yml

# 新增配置
network.host: 0.0.0.0
http.port: 9200
cluster.initial_master_nodes: ["node-1"]
```

浏览器访问 <http://124.71.229.73:9200/>



## 2.1 安装 kibana

```
Shell

安装 Kibana:
使用 apt 命令安装 Kibana。
sudo apt install kibana

配置 Kibana (可选):
根据需要配置 Kibana。配置文件通常位于 /etc/kibana/kibana.yml。可能需要
设置如服务器地址、端口、Elasticsearch URL 等。
sudo vim /etc/kibana/kibana.yml

例如，你可能需要设置 Elasticsearch 服务的 URL: 大概 32 行左右
elasticsearch.host: "http://localhost:9200"

启动 Kibana 服务:
安装完成后，启动 Kibana 服务。
```

```
sudo systemctl start kibana
```

设置开机自启（可选）：

如果你希望 Kibana 在系统启动时自动启动，可以使用以下命令来启用自启动。

```
sudo systemctl enable kibana
```

验证安装：

使用以下命令检查 Kibana 服务的状态。

```
sudo systemctl status kibana
```

访问 Kibana：

在浏览器中访问 Kibana，通常是 `http://<your-ip>:5601`

### 3. ES 核心概念

#### 索引 (Index)

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

#### 类型 (Type)

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

#### 字段 (Field)

字段相当于是数据表的字段，对文档数据根据不同属性进行的分类标识。

分类	类型	备注
字符串	text, keyword	text 会被分词生成索引; keyword 不会被分词生成索引，只能精确值搜索
整形	integer, long, short, byte	

浮点	double, float	
逻辑	boolean	true 或 false
日期	date, date_nanos	“2018-01-13” 或 “2018-01-13 12:10:30” 或者时间戳，即 1970 到现在的秒数/毫秒数
二进制	binary	二进制通常只存储，不索引
范围	range	

## 映射 (mapping)

映射是在处理数据的方式和规则方面做一些限制，如某个字段的数据类型、默认值、分析器、是否被索引等等，这些都是映射里面可以设置的，其它就是处理 es 里面数据的一些使用规则设置也叫做映射，按着最优规则处理数据对性能提高很大，因此才需要建立映射，并且需要思考如何建立映射才能对性能更好。

名称	数值	备注
enabled	true(默认)   false	是否仅作存储，不做搜索和分析
index	true(默认)   false	是否构建倒排索引（决定了是否分词，是否被索引）
index_option		
dynamic	true（缺省）  false	控制 mapping 的自动更新
doc_value	true(默认)   false	是否开启 doc_value，用户聚合和排序分析，分词字段不能使用
fielddata	fielddata: {"format": "disabled"}	<p>是否为 text 类型启动 fielddata，实现排序和聚合分析</p> <p>针对分词字段，参与排序或聚合时能提高性能，</p> <p>不分词字段统一建议使用 doc_value</p>

store	true   false(默认)	是否单独设置此字段的是否存储而从 <code>_source</code> 字段中分离， 只能搜索，不能获取值
coerce	true(默认)   false	是否开启自动数据类型转换功能， 比如：字符串转数字，浮点转整型
analyzer	"analyzer": "ik"	指定分词器，默认分词器为 <code>standard analyzer</code>
boost	"boost": 1.23	字段级别的分数加权，默认值是 1.0
fields	<pre>"fields": {   "raw": {     "type":     "text",     "index":     "not_analyzed"   } }</pre>	对一个字段提供多种索引模式， 同一个字段的值，一个分词，一个不分词
data_detection	true(默认)   false	是否自动识别日期类型

## 文档 (document)

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

Elasticsearch 与传统关系型数据库相比如下：

DB	Database	Table	Row	Column
ES	Index	Type	Document	Field

点击图片可查看完整电子表格

## 4. Kibana 访问 es 进行测试

通过网页访问 kibana:

创建索引库,

```
C++
POST /user/_doc
{
  "settings" : {
    "analysis" : {
      "analyzer" : {
        "ik" : {
          "tokenizer" : "ik_max_word"
        }
      }
    }
  },
  "mappings" : {
    "dynamic" : true,
    "properties" : {
      "nickname" : {
        "type" : "text",
        "analyzer" : "ik_max_word"
      },
      "user_id" : {
        "type" : "keyword",
        "analyzer" : "standard"
      },
      "phone" : {
        "type" : "keyword",
        "analyzer" : "standard"
      },
      "description" : {
        "type" : "text",
        "enabled" : false
      },
      "avatar_id" : {
        "type" : "keyword",
        "enabled" : false
      }
    }
  }
}
```

```
}
```

新增数据:

C++

POST /user/\_doc/\_bulk

```
{"index":{"_id":"1"}}
```

```
{"user_id" : "USER4b862aaa-2df8654a-7eb4bb65-e3507f66", "nickname" : "昵称 1", "phone" : "手机号 1", "description" : "签名 1", "avatar_id" : "头像 1"}
```

```
{"index":{"_id":"2"}}
```

```
{"user_id" : "USER14eeea5-442771b9-0262e455-e4663d1d", "nickname" : "昵称 2", "phone" : "手机号 2", "description" : "签名 2", "avatar_id" : "头像 2"}
```

```
{"index":{"_id":"3"}}
```

```
{"user_id" : "USER484a6734-03a124f0-996c169d-d05c1869", "nickname" : "昵称 3", "phone" : "手机号 3", "description" : "签名 3", "avatar_id" : "头像 3"}
```

```
{"index":{"_id":"4"}}
```

```
{"user_id" : "USER186ade83-4460d4a6-8c08068f-83127b5d", "nickname" : "昵称 4", "phone" : "手机号 4", "description" : "签名 4", "avatar_id" : "头像 4"}
```

```
{"index":{"_id":"5"}}
```

```
{"user_id" : "USER6f19d074-c33891cf-23bf5a83-57189a19", "nickname" : "昵称 5", "phone" : "手机号 5", "description" : "签名 5", "avatar_id" : "头像 5"}
```

```
{"index":{"_id":"6"}}
```

```
{"user_id" : "USER97605c64-9833ebb7-d0455353-35a59195", "nickname" : "昵称 6", "phone" : "手机号 6", "description" : "签名 6", "avatar_id" : "头像 6"}
```

查看并搜索数据

C++

GET /user/\_doc/\_search?pretty

```
{
```

```
  "query" : {
```

```
    "bool" : {
```

```
      "must_not" : [
```

```
        {
```

```
          "terms" : {
```

```
            "user_id.keyword" : [
```

```
              "USER4b862aaa-2df8654a-7eb4bb65-
```

```

e3507f66",
                                "USER14eeeee5-442771b9-0262e455-
e4663d1d",
                                "USER484a6734-03a124f0-996c169d-
d05c1869"
                                ]
                            }
                        },
                    ],
                    "should" : [
                        {
                            "match" : {
                                "user_id" : "昵称"
                            }
                        },
                        {
                            "match" : {
                                "nickname" : "昵称"
                            }
                        },
                        {
                            "match" : {
                                "phone" : "昵称"
                            }
                        }
                    ]
                }
            }
        }
    }
}

```

删除索引:

```

C++
DELETE /user

```

## 5. ES 客户端的安装

代码: <https://github.com/seznam/elasticlient>

官网: <https://seznam.github.io/elasticlient/index.html>

ES C++的客户端选择并不多, 我们这里使用 elasticlient 库, 下面进行安装。

```

Shell

```

```
# 克隆代码
git clone https://github.com/seznam/elasticlient
# 切换目录
cd elasticlient
# 更新子模块
git submodule update --init --recursive
# 编译代码
mkdir build
cd build
cmake ..
make
# 安装
make install
```

cmake 生成 makefile 的过程会遇到一个问题:

```
-- Configuring done
CMake Error: The following variables are used in this project, but they are set to NOTFOUND.
Please set them or make sure they are set and tested correctly in the CMake files:
MHD_INCLUDE_DIR (ADVANCED)
  used as include directory in directory /home/zsc/elasticlient/external/httpmockserver/src
  used as include directory in directory /home/zsc/elasticlient/external/httpmockserver/src
  used as include directory in directory /home/zsc/elasticlient/external/httpmockserver/src
  used as include directory in directory /home/zsc/elasticlient/external/httpmockserver/src
  used as include directory in directory /home/zsc/elasticlient/external/httpmockserver/src
  used as include directory in directory /home/zsc/elasticlient/external/httpmockserver/src
```

解决: 需要安装 MicroHTTPD 库

```
Shell
sudo apt-get install libmicrohttpd-dev
```

make 的时候编译出错: 这是子模块 googletest 没有编译安装

```
C++
collect2: error: ld returned 1 exit status
make[2]: *** [external/httpmockserver/test/CMakeFiles/test-server.dir/build.make:105: bin/test-server] Error 1
make[1]: *** [CMakeFiles/Makefile2:675: external/httpmockserver/test/CMakeFiles/test-server.dir/all] Error 2
make: *** [Makefile:146: all] Error 2
```

解决: 手动安装子模块

```
C++
cd ../external/googletest/
mkdir cmake && cd cmake/
cmake -DCMAKE_INSTALL_PREFIX=/usr ..
```

```
make && sudo make install
```

安装好重新 cmake 即可。

```
-- Installing: /usr/local/include/json/allocator.h
-- Installing: /usr/local/include/json/assertions.h
-- Installing: /usr/local/include/json/autolink.h
-- Installing: /usr/local/include/json/config.h
-- Installing: /usr/local/include/json/features.h
-- Installing: /usr/local/include/json/forwards.h
-- Installing: /usr/local/include/json/json.h
-- Installing: /usr/local/include/json/reader.h
-- Installing: /usr/local/include/json/value.h
-- Installing: /usr/local/include/json/version.h
-- Installing: /usr/local/include/json/writer.h
-- Installing: /usr/local/lib/libelasticlient.so.2.1.0
-- Installing: /usr/local/lib/libelasticlient.so.2
-- Set runtime path of "/usr/local/lib/libelasticlient.so.2.1.0" to ""
-- Installing: /usr/local/lib/libelasticlient.so
```

Shell

# 运行测试用例

```
make test
```

```
root@hcss-ecs-2618:/home/zsc/elasticlient/build# make test
Running tests...
Test project /home/zsc/elasticlient/build
  Start 1: tests-elasticlient
1/1 Test #1: tests-elasticlient ..... Passed    0.09 sec

100% tests passed, 0 tests failed out of 1

Total Test time (real) =  0.10 sec
```

至此， elasticlient 安装成功。

## 6. ES 客户端接口介绍

C++

```
/**
 * Perform search on nodes until it is successful. Throws
 * exception if all nodes
 * * has failed to respond.
 * * \param indexName specification of an Elasticsearch index.
 * * \param docType specification of an Elasticsearch document type.
 * * \param body Elasticsearch request body.
 * * \param routing Elasticsearch routing. If empty, no routing has
 * been used.
 *
 */
```

```

    * \return cpr::Response if any of node responds to request.
    * \throws ConnectionException if all hosts in cluster failed to
respond.
    */
cpr::Response search(const std::string &indexName,
                    const std::string &docType,
                    const std::string &body,
                    const std::string &routing = std::string());

/**
 * Get document with specified id from cluster. Throws exception
if all nodes
 * has failed to respond.
 * \param indexName specification of an Elasticsearch index.
 * \param docType specification of an Elasticsearch document type.
 * \param id Id of document which should be retrieved.
 * \param routing Elasticsearch routing. If empty, no routing has
been used.
 *
 * \return cpr::Response if any of node responds to request.
 * \throws ConnectionException if all hosts in cluster failed to
respond.
 */
cpr::Response get(const std::string &indexName,
                 const std::string &docType,
                 const std::string &id = std::string(),
                 const std::string &routing = std::string());

/**
 * Index new document to cluster. Throws exception if all nodes
has failed to respond.
 * \param indexName specification of an Elasticsearch index.
 * \param docType specification of an Elasticsearch document type.
 * \param body Elasticsearch request body.
 * \param id Id of document which should be indexed. If empty, id
will be generated
 * automatically by Elasticsearch cluster.
 * \param routing Elasticsearch routing. If empty, no routing has
been used.
 *
 * \return cpr::Response if any of node responds to request.
 * \throws ConnectionException if all hosts in cluster failed to
respond.
 */

```

```

cpr::Response index(const std::string &indexName,
                    const std::string &docType,
                    const std::string &id,
                    const std::string &body,
                    const std::string &routing = std::string());

/**
 * Delete document with specified id from cluster. Throws
exception if all nodes
 * has failed to respond.
 * \param indexName specification of an Elasticsearch index.
 * \param docType specification of an Elasticsearch document type.
 * \param id Id of document which should be deleted.
 * \param routing Elasticsearch routing. If empty, no routing has
been used.
 *
 * \return cpr::Response if any of node responds to request.
 * \throws ConnectionException if all hosts in cluster failed to
respond.
 */
cpr::Response remove(const std::string &indexName,
                     const std::string &docType,
                     const std::string &id,
                     const std::string &routing = std::string());

```

## 7. 入门案例

针对上边通过 kibana 添加的数据通过客户端 api 进行一次数据获取。

```

C++
#include <string>
#include <vector>
#include <iostream>
#include <cpr/response.h>
#include <elasticclient/client.h>

int main() {
    try {
        elasticclient::Client
client({"http://192.168.65.138:9200/"});

        // 检索文档
        std::string search_body = R"({"query": { "match_all":
{} } })";

```

```

    cpr::Response retrievedDocument = client.search(
        "user", "_doc", search_body);
    std::cout << retrievedDocument.status_code << std::endl;
    std::cout << retrievedDocument.text << std::endl;
} catch(std::exception &e) {
    std::cout << e.what() << std::endl;
}

return 0;
}

```

编译链接:

```

C++
main : main.cc
g++ -std=c++17 $^ -o $@ -lelasticclient -lcpr -ljsoncpp

```

libelasticclient.so 默认安装到/usr/local/lib 目录下, 新增动态库路径:

```

Shell
export LD_LIBRARY_PATH=/usr/local/lib:$LD_LIBRARY_PATH

```

运行结果如下所示:

```

root@hcss-ecs-2618:/home/zsc/elasticclient/build/bin# ./hello-world
201
application/json; charset=UTF-8
{"_index":"testindex","_type":"docType","_id":"docId","_version":1,"result":"created","_shards":{"total":2,"successful":1,"failed":0},"_seq_no":0,"_primary_term":1}
200
application/json; charset=UTF-8
{"_index":"testindex","_type":"docType","_id":"docId","_version":1,"_seq_no":0,"_primary_term":1,"found":true,"_source":{"message": "Hello world!"}}
200
application/json; charset=UTF-8
{"_index":"testindex","_type":"docType","_id":"docId","_version":2,"result":"deleted","_shards":{"total":2,"successful":1,"failed":0},"_seq_no":1,"_primary_term":1}

```

<https://github.com/ATinyAnt/ESClient/blob/master/request.h>

## 8. ES 客户端 API 二次封装思想:

封装客户端 api 主要是因为, 客户端只提供了基础的数据存储获取调用功能, 无法根据我们的思想完成索引的构建, 以及查询正文的构建, 需要使用者自己组织好 json 进行序列化后才能作为正文进行接口的调用。

而封装的目的就是简化用户的操作, 将索引的 json 正文构造, 以及查询搜索的正文构造操作给封装起来, 使用者调用接口添加字段就行, 不用关心具体的 json 数据格式。

封装内容:

- 索引构造过程的封装
  - 索引正文构造过程, 大部分正文都是固定的, 唯一不同的地方是各个字段不同的名称以及是否只存储不索引这些选项, 因此重点关注以下几个点即可:

- 字段类型: `type : text / keyword` (目前只用到这两个类型)
- 是否索引: `enable : true/false`
- 索引的话分词器类型: `analyzer : ik_max_word / standard`
- 新增文档构造过程的封装
  - 新增文档其实在常规下都是单条新增, 并非批量新增, 因此直接添加字段和值就行
- 文档搜索构造过程的封装
  - 搜索正文构造过程, 我们默认使用条件搜索, 我们主要关注的两个点:
    - 应该遵循的条件是什么: `should` 中有什么
    - 条件的匹配方式是什么: `match` 还是 `term/terms`, 还是 `wildcard`
    - 过滤的条件字段是什么: `must_not` 中有什么
    - 过滤的条件字段匹配方式是什么: `match` 还是 `wildcard`, 还是 `term/terms`

整个封装的过程其实就是对 `Json::Value` 对象的一个组织的过程, 并无太大的难点。