Go语言 数据操作 ElasticSearch

ElasticSearch 介绍

Elasticsearch（ES）是一个基于Lucene构建的开源、**分布式**、**RESTful**接口的全文搜索引擎。Elasticsearch还是一个分布式文档数据库，其中每个字段均可被索引，而且每个字段的数据均可被搜索，ES能够横向扩展至数以百计的服务器存储以及处理PB级的数据。可以在极短的时间内存储、搜索和分析大量的数据。通常作为具有复杂搜索场景情况下的核心发动机。

Elasticsearch 能做什么

- 当你经营一家网上商店，你可以让你的客户搜索你卖的商品。在这种情况下，你可以使用ElasticSearch来存储你的整个产品目录和库存信息，为客户提供精准搜索，可以为客户推荐相关商品。
- 当你想收集日志或者交易数据的时候，需要分析和挖掘这些数据，寻找趋势，进行统计，总结，或发现异常。在这种情况下，你可以使用Logstash或者其他工具来进行收集数据，当这引起数据存储到ElasticsSearch中。你可以搜索和汇总这些数据，找到任何你感兴趣的信息。
- 对于程序员来说，比较有名的案例是GitHub，GitHub的搜索是基于ElasticSearch构建的，在github.com`/search`页面，你可以搜索项目、用户、issue、pull request，还有代码。共有`40~50`个索引库，分别用于索引网站需要跟踪的各种数据。虽然只索引项目的主分支（master），但这个数据量依然巨大，包括20亿个索引文档，30TB的索引文件。

Elasticsearch 基本概念

Near Realtime(NRT) 几乎实时

Elasticsearch是一个几乎实时的搜索平台。意思是，从索引一个文档到这个文档可被搜索只需要一点点的延迟，这个时间一般为毫秒级。

Cluster 集群

群集是一个或多个节点（服务器）的集合， 这些节点共同保存整个数据，并在所有节点上提供联合索引和搜索功能。一个集群由一个唯一集群ID确定，并指定一个集群名（默认为“elasticsearch”）。该集群名非常重要，因为节点可以通过这个集群名加入群集，一个节点只能是群集的一部分。

确保在不同的环境中不要使用相同的群集名称，否则可能会导致连接错误的群集节点。例如，你可以使用logging-dev、logging-stage、logging-prod分别为开发、阶段产品、生产集群做记录。

Node节点

节点是单个服务器实例，它是群集的一部分，可以存储数据，并参与群集的索引和搜索功能。就像一个集群，节点的名称默认为一个随机的通用唯一标识符（UUID），确定在启动时分配给该节点。如果不希望默认，可以定义任何节点名。这个名字对管理很重要，目的是要确定你的网络服务器对应于你的ElasticSearch群集节点。

我们可以通过群集名配置节点以连接特定的群集。默认情况下，每个节点设置加入名为“elasticSearch”的集群。这意味着如果你启动多个节点在网络上，假设他们能发现彼此都会自动形成和加入一个名为“elasticsearch”的集群。

在单个群集中，你可以拥有尽可能多的节点。此外，如果“elasticsearch”在同一个网络中，没有其他节点正在运行，从单个节点的默认情况下会形成一个新的单节点名为”elasticsearch”的集群。

Index索引

索引是具有相似特性的文档集合。例如，可以为客户数据提供索引，为产品目录建立另一个索引，以及为订单数据建立另一个索引。索引由名称（必须全部为小写）标识，该名称用于在对其中的文档执行索引、搜索、更新和删除操作时引用索引。在单个群集中，你可以定义尽可能多的索引。

Type类型

在索引中，可以定义一个或多个类型。类型是索引的逻辑类别/分区，其语义完全取决于你。一般来说，类型定义为具有公共字段集的文档。例如，假设你运行一个博客平台，并将所有数据存储在一个索引中。在这个索引中，你可以为用户数据定义一种类型，为博客数据定义另一种类型，以及为注释数据定义另一类型。不过在`Elasticsearch7.0`以后的版本,已经废弃文档类型了。

Document文档

文档是可以被索引的信息的基本单位。例如，你可以为单个客户提供一个文档，单个产品提供另一个文档，以及单个订单提供另一个文档。本文件的表示形式为JSON（JavaScript Object Notation）格式，这是一种非常普遍的互联网数据交换格式。

在索引/类型中，你可以存储尽可能多的文档。请注意，尽管文档物理驻留在索引中，文档实际上必须索引或分配到索引中的类型。

Field字段

文档由多个`json`字段，这个字段跟mysql中的表的字段是类似的。ES中的字段也是有类型的，常用字段类型有：

- 数值类型(long、integer、short、byte、double、float)
- Date 日期类型
- boolean布尔类型
- Text 支持全文搜索
- Keyword 不支持全文搜索，例如：phone这种数据，用一个整体进行匹配就ok了，也不要进行分词处理
- Geo 这里主要用于地理信息检索、多边形区域的表达。

mapping映射

Elasticsearch的`mapping`类似于mysql中的表结构体定义，每个索引都有一个映射的规则，我们可以通过定义索引的映射规则，提前定义好文档的`json`结构和字段类型，如果没有定义索引的映射规则，ElasticSearch会在写入数据的时候，根据我们写入的数据字段推测出对应的字段类型，相当于自动定义索引的映射规则。

注意：ES的自动映射是很方便的，但是实际业务中，对于关键字段类型，我们都是通常预先定义好，这样可以避免ES自动生成的字段类型不是你想要的类型。

Shards & Replicas分片与副本

索引可以存储大量的数据，这些数据可能超过单个节点的硬件限制。例如，十亿个文件占用磁盘空间1TB的单指标可能不适合对单个节点的磁盘或可能太慢服务仅从单个节点的搜索请求。

为了解决这一问题，Elasticsearch提供细分你的指标分成多个块称为分片的能力。当你创建一个索引，你可以简单地定义你想要的分片数量。每个分片本身是一个全功能的、独立的"指数"，可以托管在集群中的任何节点。

Shards分片的重要性主要体现在以下两个特征：

1.副本为分片或节点失败提供了高可用性。为此，需要注意的是，一个副本的分片不会分配在同一个节点作为原始的或主分片，副本是从主分片那里复制过来的。

2.副本允许用户扩展你的搜索量或吞吐量，因为搜索可以在所有副本上并行执行。

ES基本概念与关系型数据库的比较

ES概念	关系型数据库
Index（索引）支持全文检索	Database（数据库）
Type（类型）	Table（表）
Document（文档），不同文档可以有不同的字段集合	Row（数据行）
Field（字段）	Column（数据列）
Mapping（映射）	Schema（模式）

ES API

以下示例使用curl演示

查看健康状态

```
curl -X GET 127.0.0.1:9200/_cat/health?v
```

输出

```
epoch      timestamp cluster       status node.total node.data shards pri relo
init unassign pending_tasks max_task_wait_time active_shards_percent
1670317419 09:03:39  elasticsearch yellow          1         1     10  10    0
0        1             0                  -                 90.9%
```

查询当前es集群中所有的indices

```
curl -X GET 127.0.0.1:9200/_cat/indices?v
```

输出

```
health status index                           uuid                   pri rep
docs.count docs.deleted store.size pri.store.size
green  open   .geoip_databases                drQVgdAWRhqCpUWJ1p3kUw   1   0
    41           38     39.2mb         39.2mb
green  open   .apm-custom-link                RvtlG-oNRyeYZOfmsZuZGw   1   0
     0            0       226b           226b
green  open   .kibana_task_manager_7.17.7_001 1bOwmEnOR86aTNc6vM8ppA   1   0
    17           25    120.5kb        120.5kb
green  open   .apm-agent-configuration        UpbwcPFZTA6L7ZrgeqB2YA   1   0
     0            0       226b           226b
green  open   .kibana_7.17.7_001              MIer38HEQ2mxeHsfL2OGFw   1   0
   706           33      2.4mb          2.4mb
yellow open   es_db                           gqz-JBs8TKi1ZF-z7-ftKQ   1   1
     7            0     12.6kb         12.6kb
green  open   .tasks                          QRRZjPHgRDWeCM8ENVbjOQ   1   0
     4            0     21.3kb         21.3kb
```

创建索引

```
curl -X PUT 127.0.0.1:9200/www
```

输出

```
{"acknowledged":true,"shards_acknowledged":true,"index":"www"}
```

删除索引

```
curl -X DELETE 127.0.0.1:9200/www
```

输出:

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

插入记录

```
curl -H "ContentType:application/json" -X POST 127.0.0.1:9200/user/person -d '
    {
        "name": "LMH",
        "age": 18,
        "married": true
    }'
```

输出:

```
{
    "_index": "user",
    "_type": "person",
    "_id": "MLcwUWwBvEa8j5UrLZj4",
    "_version": 1,
    "result": "created",
```

```
    "_shards": {
        "total": 2,
        "successful": 1,
        "failed": 0
    },
    "_seq_no": 3,
    "_primary_term": 1
}
```

也可以使用PUT方法，但是需要传入id

```
curl -H "ContentType:application/json" -X PUT 127.0.0.1:9200/user/person/4 -d '
    {
        "name": "LMH",
        "age": 18,
        "married": false
    }'
```

检索

Elasticsearch的检索语法比较特别，使用GET方法携带JSON格式的查询条件。

全检索:

```
curl -X GET 127.0.0.1:9200/user/person/_search
```

按条件检索:

```
curl -H "ContentType:application/json" -X PUT 127.0.0.1:9200/user/person/4 -d '
    {
        "query":{
            "match": {"name": "LMH"}
        }
    }'
```

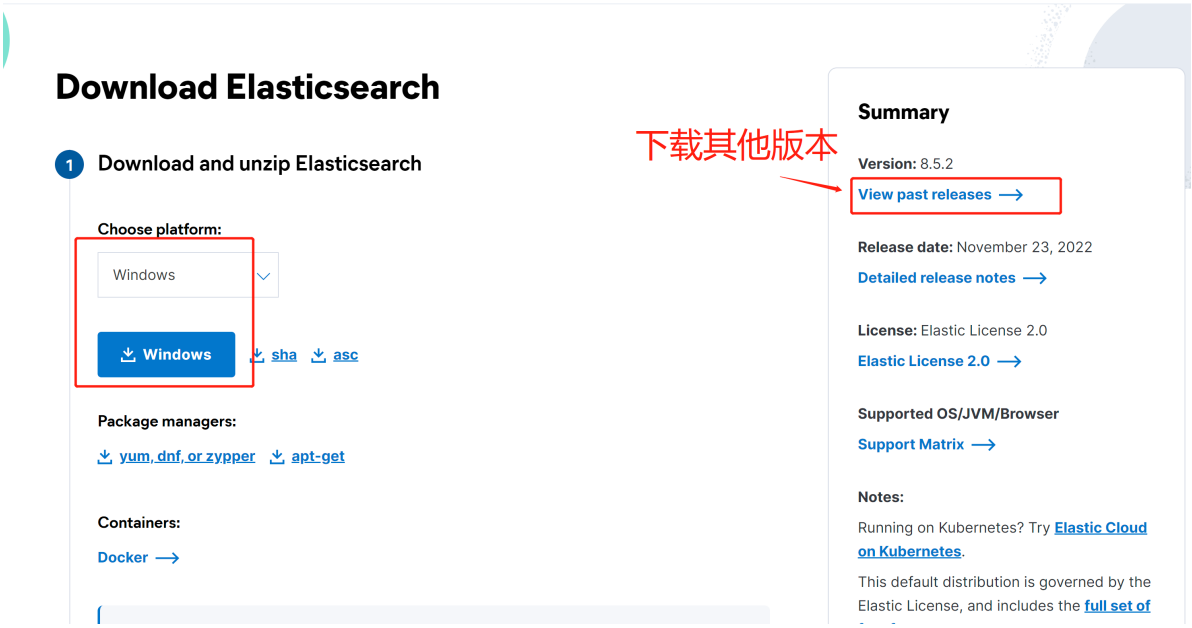
ElasticSearch默认一次最多返回10条结果，可以像下面的示例通过size字段来设置返回结果的数目。

```
curl -H "ContentType:application/json" -X PUT 127.0.0.1:9200/user/person/4 -d '
    {
        "query":{
            "match": {"name": "LMH"},
            "size": 2
        }
    }'
```

Elasticsearch 安装

官方网站下载链接：https://www.elastic.co/cn/downloads/elasticsearch

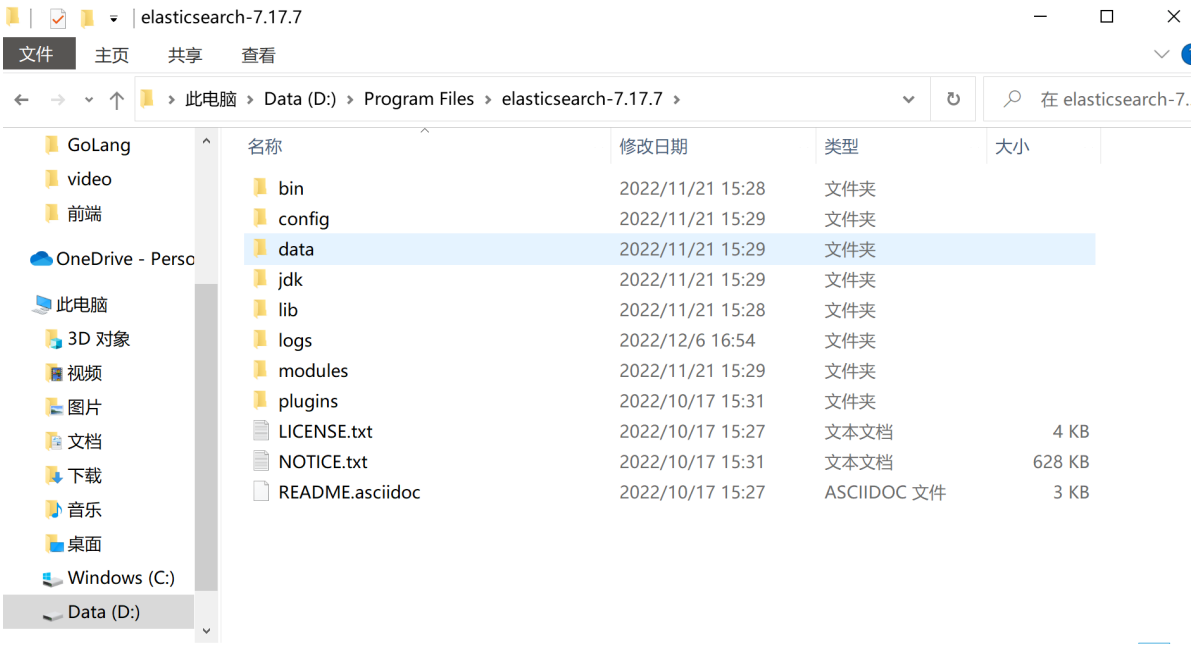
下载



请根据自己的需求下载对应的版本。

安装

将上一步下载的压缩包解压，下图以Windows为例



启动

执行bin\elasticsearch.bat启动，默认在本机的9200端口启动服务。

使用浏览器访问elasticsearch服务，可以看到类似下面的信息。

Kibana 介绍

官网链接：https://www.elastic.co/cn/products/kibana

Kibana是一个开源的分析和可视化平台，设计用于和Elasticsearch一起工作。

你可以使用Kibana来搜索、查看、并和存储在Elasticsearch索引中的数据进行交互。

你可以轻松地执行高级数据分析，并且以各种图标、表格和地图的形式可视化数据。

Kibana使得理解大量数据变得很容易。它简单的、基于浏览器的界面使你能够快速创建和共享动态仪表板，实时显示Elasticsearch查询的变化。

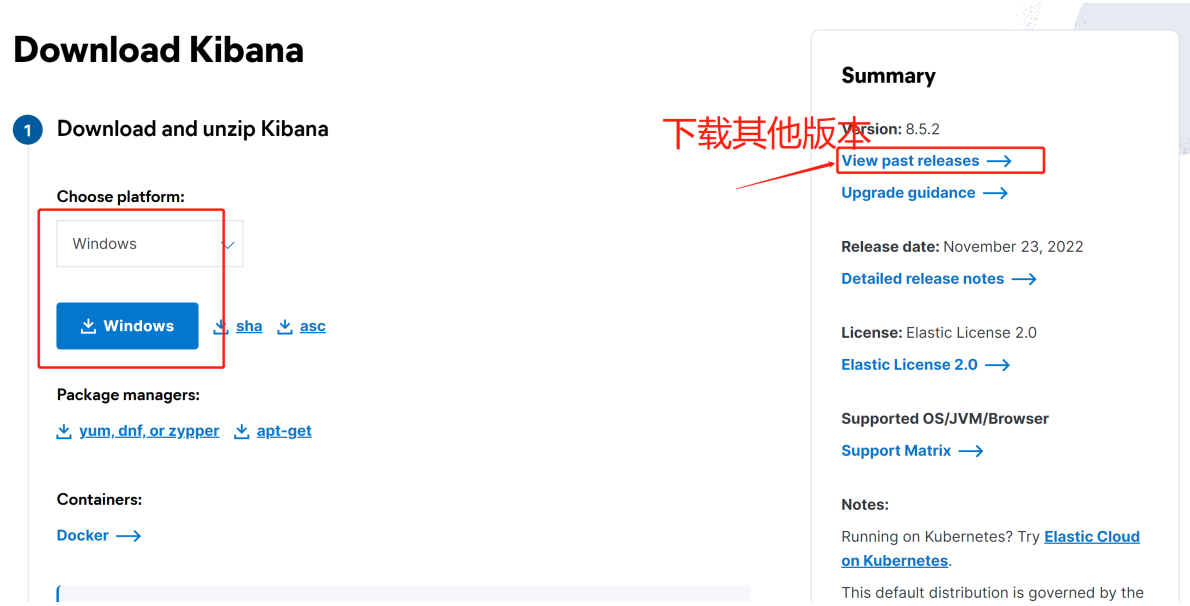
下载

官方下载链接：https://www.elastic.co/cn/downloads/kibana

请根据需求下载对应的版本。

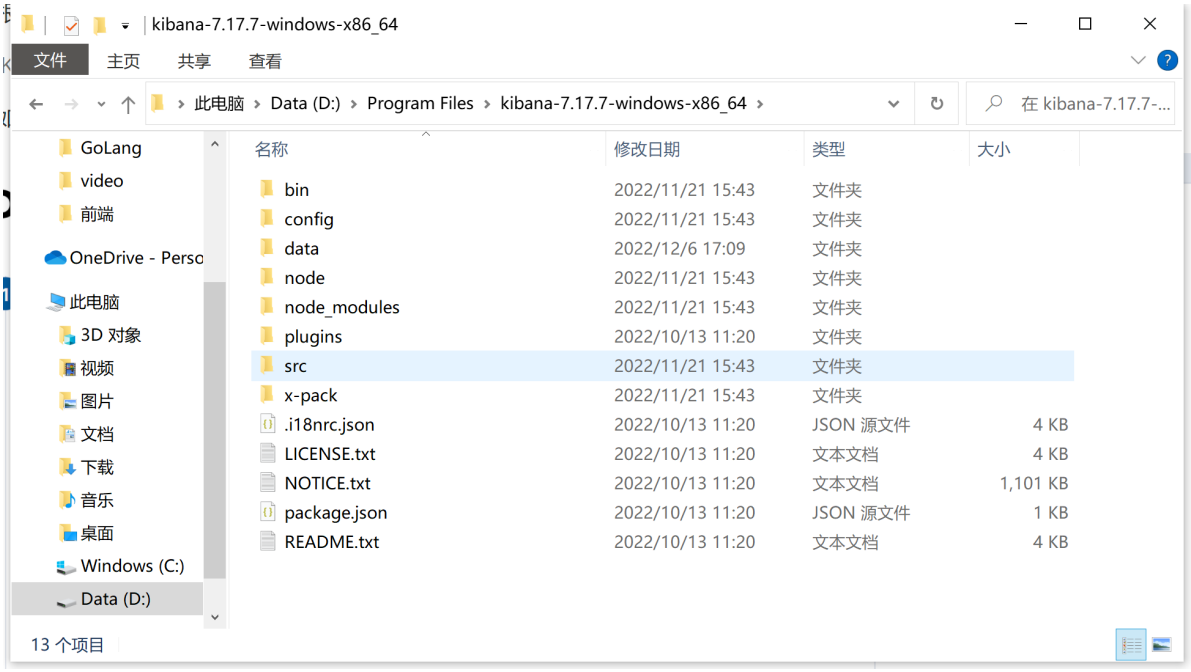
> Kibana与Elasticsearch的版本要相互对应，否则可能不兼容！！！

例如：Elasticsearch是7.17.7的版本，那么你的Kibana也要下载7.17.7的版本。



安装

将上一步下载得到的文件解压。



修改config目录下的配置文件kibana.yml（如你是本机没有发生改变可以省略这一步）

将配置文件中 elasticsearch.hosts设置为你elasticseatch的地址，例如：

```
# The Kibana server's name.  This is used for display purposes.
#server.name: "your-hostname"

# The URLs of the Elasticsearch instances to use for all your queries.
#elasticsearch.hosts: ["http://localhost:9200"]

# Kibana uses an index in Elasticsearch to store saved searches, visualizations and
# dashboards. Kibana creates a new index if the index doesn't already exist.
#kibana.index: ".kibana"
```

我这里没有修改。

然后翻到最后修改一下语言，配置成简体中文。

```
# Set the interval in milliseconds to sample system and process performance
# metrics. Minimum is 100ms. Defaults to 5000.
#ops.interval: 5000

# Specifies locale to be used for all localizable strings, dates and number formats.
# Supported languages are the following: English - en , by default , Chinese - zh-CN .
i18n.locale: "zh-CN"
```

启动

执行bin\kibana.bat启动

使用浏览器访问本机的5601端口即可看到类似下面的界面：

Go操作 ElasticSearch

Go操作Elasticsearch主要有以下两个sdk

- github.com/olivere/elastic
- github.com/elastic/go-elasticsearch

go-elastic

使用第三方库https://github.com/elastic/go-elasticsearch来链接ES并进行操作。

注意下载与你的ES相同版本的client，例如我这里使用的ES是7.17.7的版本，那么我们下载的client也要与之对应为github.com/olivere/elastic/v7。

语言客户端是向前兼容的，这代表这客户端可以和更高或是相等版本的Elasticsearch进行通讯。简单来说就是，允许使用7.X的Elasticsearch的语言客户端何以访问8.X的Elasticsearch。

```
github.com/elastic/go-elasticsearch/v7
```

链接客户端

一个默认配置的客户端，请求并接受响应：

```
es, _ := elasticsearch.NewDefaultClient()
res, _ := es.Info()
defer res.Body.Close()
```

当然客户端也可以根据自己的需要自行配置，如下：

```
// 自定义配置
cfg := elasticsearch.Config{
   // 有多个节点时需要配置
   Addresses: []string{
      "http://localhost:9200",
   },
   // 配置HTTP传输对象
   Transport: &http.Transport{
      //MaxIdleConnsPerHost 如果非零，控制每个主机保持的最大空闲(keep-alive)连接。如果为零，则使用默认配置2。
      MaxIdleConnsPerHost:   10,
      //ResponseHeaderTimeout 如果非零，则指定在写完请求(包括请求体，如果有)后等待服务器响应头的时间。
      ResponseHeaderTimeout: time.Second,
      //DialContext 指定拨号功能，用于创建不加密的TCP连接。如果DialContext为nil(下面已弃用的Dial也为nil)，那么传输拨号使用包网络。
      DialContext:           (&net.Dialer{Timeout: time.Second}).DialContext,
      // TLSClientConfig指定TLS.client使用的TLS配置。
      //如果为空，则使用默认配置。
      //如果非nil，默认情况下可能不启用HTTP/2支持。
      TLSClientConfig: &tls.Config{
         MaxVersion:         tls.VersionTLS11,
         //InsecureSkipVerify 控制客户端是否验证服务器的证书链和主机名。
         InsecureSkipVerify: true,
      },
   },
}
es, _ := elasticsearch.NewClient(cfg)
res, _ := es.Info()
defer res.Body.Close()
log.Println(res)
```

我们进行简单的封装

```
import (
    "context"
    "log"
    "strings"

    "github.com/elastic/go-elasticsearch/v7"
    "github.com/elastic/go-elasticsearch/v7/esapi"
)

var es *elasticsearch.Client
```

```
func initEs() {
    var err error
    es, err = elasticsearch.NewDefaultClient()
    if err != nil {
        panic(err)
    }
}
```

新增文档

使用 index api对文档进行增添或是修改操作。**如果id不存在为创建文档，如果文档存在则进行修改。**

使用esapi进行请求的包装，然后使用Do()方法执行请求

```
// 新增文档
func addReEs() {
    initEs()
    req := esapi.IndexRequest{
        Index:      "test",                                  // Index name
        Body:       strings.NewReader(`{"title" : "Test"}`), // Document body
        DocumentID: "1",                                     // Document ID
        Refresh:    "true",                                  // Refresh
    }
    res, err := req.Do(context.Background(), es)
    if err != nil {
        log.Fatalf("Error getting response: %s", err)
    }
    defer res.Body.Close()

    log.Println(res)
}
```

不覆盖的创建文档

如果不想因为在创建文档填写错了id而对不想进行操作的文档进行了修改，那么可以使用CreateRequest包装请求。

```
// 新增不覆盖文档
func addEs() {
    initEs()
    req := esapi.CreateRequest{
        Index: "test",
        // DocumentType: "user",
        DocumentID: "3",
        Body:       strings.NewReader(`{"name": "张三","age": 25, "about": "我会后空翻"}`),
    }
    res, err := req.Do(context.Background(), es)
    if err != nil {
        log.Println("我错了", err)
    }
    log.Println(res)
}
```

查询单个文档

使用GetRequest包装请求。

```
// 查询单个文档
func findEs() {
    initEs()
    req := esapi.GetRequest{
        Index: "test",
        // DocumentType: "user",
        DocumentID: "1",
    }
    res, err := req.Do(context.Background(), es)
    if err != nil {
        log.Fatalf("ERROR: %s", err)
    }
    defer res.Body.Close()

    log.Println(res)
}
```

查询多个文档

使用MgetRequest包装请求。

```
// 查询多个文档
func findManyEs() {
    initEs()
    request := esapi.MgetRequest{
        Index: "test",
        // DocumentType: "user",
        Body: strings.NewReader(`{
    "docs": [
      {
        "_id": "1"
      },
      {
        "_id": "2"
      }
    ]
  }`),
    }
    res, err := request.Do(context.Background(), es)
    if err != nil {
        log.Println("出错了，错误是", err)
    }
    log.Println(res)
}
```

修改文档

在上面我们已经进行了创建或者修改的操作，但是使用 index api进行的修改操作需要提供所有的字段，不然会返回 400。但我们大多数时候只是进行单个字段或多个字段的修改，并不会修改整个文档，这时候我们可以使用UpdateRequest包装请求。

```
// 修改文档
func editEs() {
    initEs()
    req := esapi.UpdateRequest{
        Index: "test",
        // DocumentType: "user",
        DocumentID: "2",
        Body: strings.NewReader(`
        {
           "doc": {
           "name": "李四"
           }
        }`),
    }
    res, err := req.Do(context.Background(), es)
    if err != nil {
        log.Println("出错了，错误是", err)
    }
    log.Println(res)
}
```

删除文档

使用DeleteRequest包装请求。

```
// 删除文档
func delEs() {
    initEs()
    // 使用index请求
    req := esapi.DeleteRequest{
        Index: "test",
        // DocumentType: "_doc",
        DocumentID: "1",
    }
    res, err := req.Do(context.Background(), es)
    if err != nil {
        log.Fatalf("ERROR: %s", err)
    }
    defer res.Body.Close()

    log.Println(res)
}
```

批量操作

使用BulkRequest包装请求。

```
// 批量操作
func batchEs() {
    initEs()
    // 使用index请求
    req := esapi.BulkRequest{
        // 在body中写入bulk请求
        Body: strings.NewReader(`{ "index" : { "_index" : "test", "_id" : "1" }
}
{ "title" : "Test2" }
{ "delete" : { "_index" : "test", "_id" : "2" } }
{ "create" : { "_index" : "test", "_id" : "3" } }
{ "field1" : "value3" }
{ "update" : { "_id" : "1", "_index" : "test" } }
{ "doc" : { "field2" : "value2" } }
`),
    }
    res, err := req.Do(context.Background(), es)
    if err != nil {
        log.Fatalf("ERROR: %s", err)
    }
    defer res.Body.Close()

    log.Println(res)
}
```

> 注意：格式一定要按照bulk api的格式来写，不然会400，最后别忘了回车

搜索

使用SearchRequest对请求进行包装。

```
// 搜索
func searchEs() {
    initEs()
    req := esapi.SearchRequest{
        Index: []string{"test"},
        // DocumentType: []string{"user"},
        Body: strings.NewReader(`{
    "query": {
      "match": {
        "about": "后空翻"
      }
    }
  }`),
    }
    response, err := req.Do(context.Background(), es)
    if err != nil {
        log.Println("我错了", err)
    }
    log.Println(response)
}
```

olivere

使用第三方库https://github.com/olivere/elastic 来连接ES并进行操作。

注意下载与你的ES相同版本的client，例如我这里使用的ES是7.17.7的版本，那么我们下载的client也要与之对应为github.com/olivere/elastic/v7。

```
go get -u github.com/olivere/elastic/v7
```

链接客户端

```
package main

import (
    "fmt"

    "github.com/olivere/elastic"
)

func main() {
    // 链接客户端
    client, err := elastic.NewClient(elastic.SetURL("http://127.0.0.1:9200"))
    if err != nil {
        fmt.Printf("链接失败! 错误是: %v\n", err)
    } else {
        fmt.Println("链接成功")
    }
}
```

我们来进行简单的封装

```
package main

import (
    "context"
    "fmt"

    "github.com/olivere/elastic"
)

var client *elastic.Client
var host = "http://127.0.0.1:9200/"

// 初始化
func init() {
    var err error
    // 链接客户端
    client, err = elastic.NewClient(elastic.SetURL(host))
    if err != nil {
        fmt.Printf("连接失败，错误: %v\n", err)
        panic(err)
    }
```

```
    // 测试链接
    info, code, err := client.Ping(host).Do(context.Background())
    if err != nil {
        panic(err)
    }
    fmt.Printf("Elasticsearch链接成功，返回码code %d。版本号 %s\n", code,
info.Version.Number)
}
```

创建文档

```
// 定义用户结构体
type User struct {
    Name      string   `json:"name"`
    Age       int      `json:"age"`
    About     string   `json:"about"`
    Interests []string `json:"interests"`
}

// 创建文档
func create() {
    //使用结构体创建
    e1 := User{"万叶", 23, "我会落叶飘零", []string{"吹笛子"}}
    put1, err := client.Index().
        Index("ys").
        Type("user").
        Id("1").
        BodyJson(e1).
        Do(context.Background())
    if err != nil {
        panic(err)
    }
    fmt.Printf("文档ID：%s ，索引index： %s, 类型是： %s\n", put1.Id, put1.Index,
put1.Type)

    //使用字符串
    e2 := `{"name":"刻晴","age":25,"about":"我喜欢工作","interests":["看书","工
作"]}`
    put2, err := client.Index().
        Index("ys").
        Type("user").
        Id("2").
        BodyJson(e2).
        Do(context.Background())
    if err != nil {
        panic(err)
    }
    fmt.Printf("文档ID：%s ，索引index： %s, 类型是： %s\n", put2.Id, put2.Index,
put2.Type)

    e3 := `{"name":"雷电将军","age":35,"about":"我不会做饭","interests":["发呆"]}`
    put3, err := client.Index().
        Index("ys").
        Type("user").
        Id("3").
```

```
        BodyJson(e3).
        Do(context.Background())
    if err != nil {
        panic(err)
    }
    fmt.Printf("文档ID：%s ，索引index： %s, 类型是： %s\n", put3.Id, put3.Index,
put3.Type)
}
```

删除文档

```
// 删除文档
func delete() {
    res, err := client.Delete().Index("ys").
        Type("user").
        Id("1").
        Do(context.Background())
    if err != nil {
        println(err.Error())
        return
    }
    fmt.Printf("删除结果： %s\n", res.Result)
}
```

修改文档

```
// 修改文档
func update() {
    res, err := client.Update().
        Index("ys").
        Type("user").
        Id("3").
        Doc(map[string]interface{}{"age": 88}).
        Do(context.Background())
    if err != nil {
        println(err.Error())
    }
    fmt.Printf("修改结果 %s\n", res.Result)
}
```

查询文档

```
// 查找
func gets() {
    //通过id查找
    get1, err :=
client.Get().Index("ys").Type("user").Id("2").Do(context.Background())
    if err != nil {
        panic(err)
    }
    if get1.Found {
        var resultType User
        json.Unmarshal(*get1.Source, &resultType)
        fmt.Printf("文档ID：%s， 索引：%s, 类型：%s, 详细内容：%v\n", get1.Id,
get1.Index, get1.Type, resultType)
    }
}
```

搜索

```
// 搜索
func query() {
    var res *elastic.SearchResult
    var err error

    //取所有
    res, err = client.Search("ys").Type("user").Do(context.Background())
    printUser(res, err)

    //字段相等
    q := elastic.NewQueryStringQuery("name:雷电将军")
    res, err =
client.Search("ys").Type("user").Query(q).Do(context.Background())
    if err != nil {
        println(err.Error())
    }
    printUser(res, err)

    //条件查询
    //年龄大于25岁的
    boolQ := elastic.NewBoolQuery()
    boolQ.Must(elastic.NewMatchQuery("name", "刻晴"))
    boolQ.Filter(elastic.NewRangeQuery("age").Gt(25))
    res, err =
client.Search("ys").Type("user").Query(q).Do(context.Background())
    printUser(res, err)

    //短语搜索 搜索about字段中有 做饭
    matchPhraseQuery := elastic.NewMatchPhraseQuery("about", "做饭")
    res, err =
client.Search("ys").Type("user").Query(matchPhraseQuery).Do(context.Background()
)
    printUser(res, err)

    //分析 interests
    aggs := elastic.NewTermsAggregation().Field("interests.keyword")
```

```
    res, err = client.Search("ys").Type("user").Aggregation("all_interests",
aggs).Do(context.Background())
    printUser(res, err)
}
```

简单分页

```
// 简单分页
func list(size, page int) {
    if size < 0 || page < 1 {
        fmt.Printf("param error")
        return
    }
    res, err := client.Search("ys").
        Type("user").
        Size(size).
        From((page - 1) * size).
        Do(context.Background())
    printUser(res, err)
}
```

打印查询

```
// 打印查询到的User
func printUser(res *elastic.SearchResult, err error) {
    if err != nil {
        print(err.Error())
        return
    }
    var typ User
    for _, item := range res.Each(reflect.TypeOf(typ)) { //从搜索结果中取数据的方法
        t := item.(User)
        fmt.Printf("%#v\n", t)
    }
}
```