目录

Elasticsearch .net client NEST 使用总结

Elasticsearch.Net 与 NEST 是 Elasticsearch 为 C#提供的一套客户端驱动，方便 C#调用 Elasticsearch 服务接口。Elasticsearch.Net 是对 Elasticsearch 服务接口较基层的的实现，NEST 是在前者基础之上进行的封装，方法更简洁使用更方便。本文是针对 NEST 2.X 的使用的总结。

。

引用

引用 dll

```
NEST.dll
Elasticsearch.Net.dll
Newtonsoft.Json.dll
```

概念

存储结构：

在 Elasticsearch 中，文档(Document)归属于一种类型(type),而这些类型存在于索引(index)中.

类比传统关系型数据库：

```
Relational DB -> Databases -> Tables -> Rows -> Columns
Elasticsearch -> Indices   -> Types  -> Documents -> Fields
```

客户端语法

链式 lambda 表达式（ [powerful query DSL](#)）语法

```
s => s.Query(q => q
    .Term(p => p.Name, "elasticsearch")
)
```

对象初始化语法

```
var searchRequest = new SearchRequest<VendorPriceInfo>
  {
     Query = new TermQuery
     {
        Field = "name",
        Value = "elasticsearch"
     }
  };
```

Connection 链接

```
//单 node
Var node = new Uri("……");
var settings = new ConnectionSettings(node);

//多 uris
Var uris = new Uri [] {
        new Uri("……"),
        new Uri("……")
};
var pool = new StaticConnectionPool(uris);

//多 node
Var nodes = new Node [] {
        new Node (new Uri("……")),
        new Node (new Uri("……"))
};
var pool = new StaticConnectionPool(nodes);

var settings = new ConnectionSettings(pool);

var client = new ElasticClient(settings);
```

连接池类型

```
//单节点
IConnectionPool pool = new SingleNodeConnectionPool(urls.FirstOrDefault());


//请求时随机发送请求到各个正常的节点，不请求异常节点,异常节点恢复后会重新被请求
IConnectionPool pool = new StaticConnectionPool(urls);


IConnectionPool pool = new SniffingConnectionPool(urls);
//false.创建客户端时，随机选择一个节点作为客户端的请求对象，该节点异常后不会切换其它节点
//true，请求时随机请求各个正常节点，不请求异常节点,但异常节点恢复后不会重新被请求
pool.SniffedOnStartup = true;


//创建客户端时，选择第一个节点作为请求主节点，该节点异常后会切换其它节点，待主节点恢复后会自动切换回来
IConnectionPool pool = new StickyConnectionPool(urls);
```

索引选择

方式 1

```
//链接设置时设置默认使用索引
var settings = new ConnectionSettings().DefaultIndex("defaultindex");
```

方式 2

```
//链接设置时设置索引与类型对应关系，操作时根据类型使用对应的索引
var settings = new ConnectionSettings().MapDefaultTypeIndices(m => m.Add(typeof(Project), "projec
ts")  );
```

方式 3

```
//执行操作时指定索引
client.Search<VendorPriceInfo>(s => s.Index("test-index"));

client.Index(data,o=>o.Index("test-index"));
```

优先级：方式 3 > 方式 2 > 方式 1

索引（Index）

数据唯一 Id

1） 默认以“Id”字段值作为索引唯一 Id 值,无“Id”属性，Es 自动生成唯一 Id 值，添加数据时统一类型数据唯一 ID 已存在相等值，将只做更新处理。

注：自动生成的 ID 有 22 个字符长，URL-safe, Base64-encoded string universally unique identifiers，或者叫 UUIDs。

2） 标记唯一 Id 值

```
[ElasticsearchType(IdProperty = "priceID")]
 public class VendorPriceInfo
 {
     public Int64 priceID { get; set; }
     public int oldID { get; set; }
     public int source { get; set; }
}
```

3） 索引时指定

```
client.Index(data, o => o.Id(data.vendorName));
```

优先级： 3) > 2) > 1)

类型（Type）

1） 默认类型为索引数据的类名（自动转换为全小写）

2） 标记类型

```
[ElasticsearchType(Name = "datatype")]
 public class VendorPriceInfo
 {
     public Int64 priceID { get; set; }
     public int oldID { get; set; }
     public int source { get; set; }
 }
```

3） 索引时指定

```
client.Index(data, o => o.Type(new TypeName() { Name = "datatype", Type = typeof(VendorPriceInfo)
 }));
或
client.Index(data, o => o.Type<MyClass>());//使用 2）标记的类型
```

优先级：3）> 2) > 1)

创建

```
client.CreateIndex("test2");
//基本配置
IIndexState indexState=new IndexState()
{
      Settings = new IndexSettings()
      {
          NumberOfReplicas = 1,//副本数
          NumberOfShards = 5//分片数
      }
};
client.CreateIndex("test2", p => p.InitializeUsing(indexState));

//创建并 Mapping
client.CreateIndex("test-index3", p => p.InitializeUsing(indexState).Mappings(m => m.Map<VendorPr
iceInfo>(mp => mp.AutoMap())));
```

注：索引名称必须小写

判断

```
client.IndexExists("test2");
```

删除

```
client.DeleteIndex("test2");
```

映射

概念

每个类型拥有自己的**映射(mapping)**或者**模式定义(schema definition)**。一个映射定义了字段类型、每个字段的数据类型、以及字段被 Elasticsearch 处理的方式。

获取映射

```
var resule = client.GetMapping<VendorPriceInfo>();
```

特性

```
    /// <summary>
    /// VendorPrice 实体
    /// </summary>
    [ElasticsearchType(IdProperty = "priceID", Name = "VendorPriceInfo")]
    public class VendorPriceInfo
    {
        [Number(NumberType.Long)]
        public Int64 priceID { get; set; }
        [Date(Format = "mmddyyyy")]
        public DateTime modifyTime { get; set; }
        /// <summary>
        /// 如果 string 类型的字段不需要被分析器拆分，要作为一个正体进行查询，需标记此声明，否则索引的值将被分析器拆分
        /// </summary>
        [String(Index = FieldIndexOption.NotAnalyzed)]
        public string pvc_Name { get; set; }
        /// <summary>
        /// 设置索引时字段的名称
        /// </summary>
        [String(Name = "PvcDesc")]
        public string pvc_Desc { get; set; }
        /// <summary>
        /// 如需使用坐标点类型需添加坐标点特性，在 maping 时会自动映射类型
        /// </summary>
```

```
        [GeoPoint(Name = "ZuoBiao",LatLon = true)]
        public GeoLocation Location { get; set; }
    }
```

映射

```
            //根据对象类型自动映射
            var result= client.Map<VendorPriceInfo>(m => m.AutoMap());
            //手动指定
            var result1 = client.Map<VendorPriceInfo>(m => m.Properties(p => p
            .GeoPoint(gp => gp.Name(n => n.Location)// 坐标点类型
                .Fielddata(fd => fd
                    .Format(GeoPointFielddataFormat.Compressed)//格式 array doc_values compressed dis
abled
                    .Precision(new Distance(2, DistanceUnit.Meters)) //精确度
                    ))
            .String(s => s.Name(n => n.b_id))//string 类型
            ));
            //在原有字段下新增字段（用于存储不同格式的数据，查询使用该字段方法查看【基本搜索】）
            //eg:在 vendorName 下添加无需分析器分析的值  temp
            var result2 = client.Map<VendorPriceInfo>(
                m => m
                    .Properties(p => p.String(s => s.Name(n => n.vendorName).Fields(fd => fd.String(s
s => ss.Name("temp").Index(FieldIndexOption.NotAnalyzed))))));
```

注：映射时已存在的字段将无法重新映射，只有新加的字段能映射成功。所以最好在首次创建索引后先进性映射再索引数据。

数据

数据操作

说明

- 添加数据时，如果文档的唯一 **id** 在索引里已存在，那么会替换掉原数据；
- 添加数据时，如果索引不存在，服务会自动创建索引；
- 如果服务自动创建索引，并索引了数据，那么索引的映射关系就是服务器自动设置的；
- 通常正确的使用方法是在紧接着创建索引操作之后进行映射关系的操作，以保证索引数据的映射是正确的。然后才是索引数据；
- 文档在 **Elasticsearch** 中是不可变的,执行 **Update** 事实上 **Elasticsearch** 的处理过程如下：

1. 从旧文档中检索 **JSON**
2. 修改 **JSON** 值
3. 删除旧文档
4. 索引新文档

所以我们也可以使用 **Index** 来更新已存在文档，只需对应文档的唯一 **id**。

添加索引数据

添加单条数据

```
var data = new VendorPriceInfo() { vendorName = "测试"};

client.Index(data);
```

添加多条数据

```
var datas = new List<VendorPriceInfo> {

              new VendorPriceInfo(){priceID = 1,vendorName = "test1"},

              new VendorPriceInfo(){priceID = 2,vendorName = "test2"}};

client.IndexMany(datas);
```

删除数据

单条数据

```
DocumentPath<VendorPriceInfo> deletePath=new DocumentPath<VendorPriceInfo>(7);

client.Delete(deletePath);

或

IDeleteRequest request = new DeleteRequest("test3", "vendorpriceinfo", 0);

client.Delete(request);
```

注：删除时根据唯一id 删除

集合数据

```
           Indices indices = "test-1";

          Types types = "vendorpriceinfo";

          //批量删除 需要 es 安装 delete-by-query 插件

          var result = client.DeleteByQuery<VendorPriceInfo>(indices, types,

              dq =>

                  dq.Query(

                      q =>

                          q.TermRange(tr => tr.Field(fd => fd.priceID).GreaterThanOrEquals("5").Less
ThanOrEquals("10")))

                          );
```

更新数据

更新所有字段

```
DocumentPath<VendorPriceInfo> deletePath=new DocumentPath<VendorPriceInfo>(2);
```

```
Var response=client.Update(deletePath,(p)=>p.Doc(new VendorPriceInfo(){vendorName = "test2updat
e..."}));
//或
IUpdateRequest<VendorPriceInfo, VendorPriceInfo> request = new UpdateRequest<VendorPriceInfo, Ven
dorPriceInfo>(deletePath)
          {
              Doc = new VendorPriceInfo()
              {
                  priceID = 888,
                  vendorName = "test4update........"
              }
          };
          var response = client.Update<VendorPriceInfo, VendorPriceInfo>(request);
```

更新部分字段

```
IUpdateRequest<VendorPriceInfo, VendorPriceInfoP> request = new UpdateRequest<VendorPriceInfo, Ve
ndorPriceInfoP>(deletePath)
          {
              Doc = new VendorPriceInfoP()
              {
                  priceID = 888,
                  vendorName = "test4update........"
              }

          };
          var response = client.Update(request);
```

更新部分字段

```
IUpdateRequest<VendorPriceInfo, object> request = new UpdateRequest<VendorPriceInfo, object>(dele
tePath)
          {
              Doc = new
              {
                  priceID = 888,
                  vendorName = " test4update........"
```

```
            }
        };
        var response = client.Update(request);
//或
client.Update<VendorPriceInfo, object>(deletePath, upt => upt.Doc(new { vendorName = "ptptptptp"
 }));
```

注：更新时根据唯一id 更新

获取数据

```
var response = client.Get(new DocumentPath<VendorPriceInfo>(0));
//或
var response =
 client.Get(new DocumentPath<VendorPriceInfo>(0),pd=>pd.Index("test4").Type("v2"));
//多个
var response = client.MultiGet(m => m.GetMany<VendorPriceInfo>(new List<long> { 1, 2, 3, 4 }));
```

注：获取时根据唯一id 获取

搜索

说明

- **搜索分页时随着分页深入，资源花费是成倍增长的。限制 from+size<=10000** 分页说明资料　资料
- **需要扫描所有数据时，使用滚屏扫描。** 扫描与滚屏资料
- **搜索中提供了查询（Query）和过滤（Filter）：**
    - **query 是要计算相关性评分的，filter 不要；**
    - **query 结果不缓存，filter 缓存。**
    - **全文搜索、评分排序，使用 query；**
    - **是非过滤，精确匹配，使用 filter。**

基本搜索

```
var result = client.Search<VendorPriceInfo>(
            s => s
                .Explain() //参数可以提供查询的更多详情。
                .FielddataFields(fs => fs //对指定字段进行分析
                    .Field(p => p.vendorFullName)
                    .Field(p => p.cbName)
                )
                .From(0) //跳过的数据个数
```

```
                .Size(50) //返回数据个数
                .Query(q =>
                    q.Term(p => p.vendorID, 100) // 主要用于精确匹配哪些值，比如数字，日期，布尔值或 not_analyzed 的字符串(未经分析的文本数据类型)：
                    &&
                    q.Term(p => p.vendorName.Suffix("temp"), "姓名") //用于自定义属性的查询（定义方法查看 MappingDemo）
                    &&
                    q.Bool( //bool 查询
                        b => b
                            .Must(mt => mt //所有分句必须全部匹配，与 AND 相同
                                .TermRange(p => p.Field(f => f.priceID).GreaterThan("0").LessThan("1"))) //指定范围查找
                            .Should(sd => sd //至少有一个分句匹配，与 OR 相同
                                .Term(p => p.priceID, 32915),
                                sd => sd.Terms(t => t.Field(fd => fd.priceID).Terms(new[] {10, 20, 30})),//多值
                                //||
                                //sd.Term(p => p.priceID, 1001)
                                //||
                                //sd.Term(p => p.priceID, 1005)
                                sd => sd.TermRange(tr => tr.GreaterThan("10").LessThan("12").Field(f => f.vendorPrice))
                            )
                            .MustNot(mn => mn//所有分句都必须不匹配，与 NOT 相同
                                .Term(p => p.priceID, 1001)
                                ,
                                mn => mn.Bool(
                                    bb=>bb.Must(mt=>mt
                                        .Match(mc=>mc.Field(fd=>fd.carName).Query("至尊"))
                                    ))
                            )
                        )
                )//查询条件
            .Sort(st => st.Ascending(asc => asc.vendorPrice))//排序
            .Source(sc => sc.Include(ic => ic
                .Fields(
                    fd => fd.vendorName,
```

```
                    fd => fd.vendorID,
                    fd => fd.priceID,
                    fd => fd.vendorPrice))) //返回特定的字段
            );
        //TResult
        var result1 = client.Search<VendorPriceInfo, VendorPriceInfoP>(s => s.Query(
            q => q.MatchAll()
            )
            .Size(15)
            );
```

或

```
var result = client.Search<VendorPriceInfo>(new SearchRequest()
      {
        Sort =new List<ISort>
              {
                new SortField { Field = "vendorPrice", Order = SortOrder.Ascending }
              },
        Size = 10,
        From = 0,
        Query = new TermQuery()
        {
            Field = "priceID",
            Value = 6
        }
        ||
        new TermQuery(
        {
            Field = "priceID",
            Value = 8
        }
      });
```

分页

```
        //分页最大限制（from+size<=10000）
        int pageSize = 10;
```

```
int pageIndex = 1;
var result = client.Search<VendorPriceInfo>(s => s.Query(q => q
    .MatchAll())
    .Size(pageSize)
    .From((pageIndex - 1) * pageSize)
    .Sort(st => st.Descending(d => d.priceID)));
```

扫描和滚屏

```
string scrollid = "";
var result = client.Search<VendorPriceInfo>(s => s.Query(q => q.MatchAll())
    .Size(100)
    .SearchType(SearchType.Scan)
    .Scroll("1m"));//scrollid过期时间
//得到滚动扫描的id
scrollid = result.ScrollId;
//执行滚动扫描得到数据 返回数据量是 result.Shards.Successful*size(查询成功的分片数*size)
result = client.Scroll<VendorPriceInfo>("1m", scrollid);
//得到新的id
scrollid = result.ScrollId;
```

查询条件相关性加权

eg:通过城市与省份查询，设置城市的相关性高于省份的，结果的等分匹配城市的将高于匹配省份的。

```
// 在原分值基础上 设置不同匹配的加成值 具体算法为lucene内部算法
var result = client.Search<VendorPriceInfo>(s => s
            .Query(q =>
                q.Term(t => t
                          .Field(f => f.cityID).Value(2108).Boost(4))
                           ||
                q.Term(t => t
                          .Field(f => f.pvcId).Value(2103).Boost(1))
                )
            .Size(3000)
            .Sort(st => st.Descending(SortSpecialField.Score))
```

```
            );
```

得分控制

```
//使用 functionscore 计算得分
var result1 = client.Search<VendorPriceInfo>(s => s
             .Query(q=>q.FunctionScore(f=>f
                       //查询区
                    .Query(qq => qq.Term(t => t
                                  .Field(fd => fd.cityID).Value(2108))
                                  ||
                               qq.Term(t => t
                                .Field(fd => fd.pvcId).Value(2103))
                    )
                    .Boost(1.0) //functionscore 对分值影响
                    .BoostMode(FunctionBoostMode.Replace)//计算 boost 模式 ; Replace 为替换
                    .ScoreMode(FunctionScoreMode.Sum) //计算 score 模式: Sum 为累加
                     //逻辑区
                    .Functions(fun=>fun
                       .Weight(w => w.Weight(2).Filter(ft => ft
                          .Term(t => t
                                  .Field(fd => fd.cityID).Value(2108))))//匹配 cityid +2
                       .Weight(w => w.Weight(1).Filter(ft => ft
                          .Term(t => t
                                  .Field(fd => fd.pvcId).Value(2103))))//匹配 pvcid +1
                    )
                 )
                )
             .Size(3000)
             .Sort(st => st.Descending(SortSpecialField.Score).Descending(dsc=>dsc.priceID))
             );
            //结果中 cityid=2108, 得分=2;  pvcid=2103 , 得分=1 ,两者都满足的, 得分=3
```

查询字符串-简单的检索

```
          var result = client.Search<VendorPriceInfo>(s => s
              .Query(q => q.QueryString(m => m.Fields(fd=>fd.Field(fdd=>fdd.carName).Field(fdd=>fd
d.carGearBox))
```

```
                        .Query("手自一体")
                        )
                )
            .From(0)
            .Size(15)
            );
```

全文检索-关键词搜索

```
var result=client.Search<VendorPriceInfo>(s=>s
            .Query(q=>q
                .Match(m=>m.Field(f=>f.carName)
                        .Query("尊贵型")
                        )
            )
            .From(0)
            .Size(15)
            );
        //多字段匹配
        var result1 = client.Search<VendorPriceInfo>(s => s
            .Query(q => q
                .MultiMatch(m => m.Fields(fd=>fd.Fields(f=>f.carName,f=>f.carGearBox))
                        .Query("尊贵型")
                        )
            )
            .From(0)
            .Size(15)
            );
```

全文搜索-短语搜索

```
var result = client.Search<VendorPriceInfo>(s => s
            .Query(q => q.MatchPhrase(m => m.Field(f => f.carName)
                        .Query("尊贵型")
                        )
            )
            .From(0)
```

```
            .Size(15)
            );
```

坐标点搜索-根据坐标点及距离搜索

eg:搜索指定地点附近所有经销商

```
        const double lat = 39.8694890000;
        const double lon = 116.4206470000;
        const double distance = 2000.0;
        //1 bool 查询中作为 must 一个条件
        var result = client.Search<VendorPriceInfo>(s => s
            .Query(q => q
            .Bool(b => b.Must(m => m
                .GeoDistance(gd => gd
                    .Location(lat, lon)
                    .Distance(distance, DistanceUnit.Meters)
                    .Field(fd => fd.Location)
                    ))
                )
            )
            .From(0)
            .Size(15)
            );
         //2 与 bool 查询同级条件
        var location = new GeoLocation(lat, lon);
        var distancei = new Distance(distance, DistanceUnit.Meters);
        var result1 = client.Search<VendorPriceInfo>(s => s
            .Query(q => q
                    .Bool(b => b.Must(m => m
                        .Exists(e => e.Field(fd => fd.Location))
                        )
                    )
                    &&
                    q.GeoDistance(gd => gd
                    .Location(location)
                    .Distance(distancei)
                    .Field(fd => fd.Location)
```

```
                )
            )
            .From(0)
            .Size(15)
            );
        //3 作为 bool 查询 中 must 的一个筛选项-筛选项只筛选结果不会影响分值
        var result2 = client.Search<VendorPriceInfo>(s => s
            .Query(q => q
                .Bool(b=>b
                    .Must(m=>m.MatchAll())
                    .Filter(f=>f
                        .GeoDistance(g => g
                            .Name("named_query")
                            .Field(p => p.Location)
                            .DistanceType(GeoDistanceType.Arc)
                            .Location(lat,lon)
                            .Distance("2000.0m")
                            )
                        )
                    )
                )
            .From(0)
            .Size(15)
            );
```

聚合

聚合-基本

```
var result = client.Search<VendorPriceInfo>(s => s
            .From(0)
            .Size(15)
            .Aggregations(ag=>ag
                    .ValueCount("Count", vc => vc.Field(fd => fd.vendorPrice))//总数
                    .Sum("vendorPrice_Sum", su => su.Field(fd => fd.vendorPrice))//求和
                    .Max("vendorPrice_Max", m => m.Field(fd => fd.vendorPrice))//最大值
                    .Min("vendorPrice_Min", m => m.Field(fd => fd.vendorPrice))//最小值
                    .Average("vendorPrice_Avg", avg => avg.Field(fd => fd.vendorPrice))//平均值
```

```
                .Terms("vendorID_group", t => t.Field(fd => fd.vendorID).Size(100))//分组
            )
        );
```

聚合-分组

eg:统计每个经销商下的平均报价

```
        //每个经销商 的平均报价
        var result = client.Search<VendorPriceInfo>(s => s
            .Size(0)
            .Aggregations(ag => ag
                .Terms("vendorID_group", t => t
                    .Field(fd => fd.vendorID)
                    .Size(100)
                    .Aggregations(agg => agg.Average("vendorID_Price_Avg", av => av.Field(fd =>
fd.vendorPrice)))
                )//分组
                .Cardinality("vendorID_group_count", dy => dy.Field(fd => fd.vendorID))//分组
数量
                .ValueCount("Count", c => c.Field(fd => fd.vendorID))//总记录数
            )
        );
```

聚合-分组-聚合..

eg:统计每个经销商对每个品牌的平均报价、最高报价、最低报价等

```
        //每个经销商下 每个品牌 的平均报价
        var result = client.Search<VendorPriceInfo>(s => s
            .Size(0)
            .Aggregations(ag => ag
                .Terms("vendorID_group", //vendorID 分组
                    t => t.Field(fd => fd.vendorID)
                    .Size(100)
                    .Aggregations(agg => agg
                                .Terms("vendorID_cbID_group", //cbID 分组
                                    tt => tt.Field(fd => fd.cbID)
                                    .Size(50)
```

```
                                    .Aggregations(aggg => aggg
                                        .Average("vendorID_cbID_Price_Avg", av => av.Field(fd =>
fd.vendorPrice))//Price avg
                                        .Max("vendorID_cbID_Price_Max", m => m.Field(fd => fd.ven
dorPrice))//Price max
                                        .Min("vendorID_cbID_Price_Min", m => m.Field(fd => fd.ven
dorPrice))//Price min
                                        .ValueCount("vendorID_cbID_Count", m => m.Field(fd => fd.
cbID))//该经销商对该品牌 报价数 count
                                        )
                                    )
                                .Cardinality("vendorID_cbID_group_count", dy => dy.Field(fd => f
d.cbID))//分组数量
                                .ValueCount("vendorID_Count", c => c.Field(fd => fd.vendorID))//
该经销商的报价数
                        )
                    )
                    .Cardinality("vendorID_group_count",dy=>dy.Field(fd=>fd.vendorID))//分组数量
                    .ValueCount("Count",c=>c.Field(fd=>fd.priceID))//总记录数
            ) //分组
            );
```

附

距离单位：

- `mm` （毫米）
- `cm` （厘米）
- `m` （米）
- `km` （千米）
- `in` （英寸）
- `ft` （英尺）
- `yd` （码）
- `mi` （英里）
- `nmi` or `NM` （海里）

日期单位：

- y (year)
- M (month)
- w (week)
- d (day)
- h (hour)
- m (minute)
- s (second)
- ms (milliseconds)

官网文档

https://www.elastic.co/guide/en/elasticsearch/client/net-api/2.x/introduction.html