第三阶段：Go中间件使用

一、学习路线

1.1 课程内容

- Redis的认识与使用
- Elasticsearch的认识与使用
- Kafka的认识与使用

1.2 学习路线

- 认识
- 常用场景
- 基础操作
- 案例Demo

1.3 目标

- 掌握Redis、Elasticsearch、Kafka的基本操作与使用
- 掌握学习新知识点的技巧，通过自学方式掌握其他三方库的用法

二、Redis使用

2.1介绍

Redis（Remote Dictionary Server）是一个开源的内存数据库，它提供了快速、可扩展和灵活的数据存储和访问解决方案。Redis以键值对的形式存储数据，并支持多种数据结构，包括字符串、哈希、列表、集合、有序集合等。它的主要特点包括：

- 快速高效：Redis完全存储在内存中，因此具有快速的读写性能。它使用高效的数据结构和算法，可以在毫秒级别内处理大量的请求。
- 数据持久化：Redis支持将数据持久化到磁盘，可以将内存中的数据定期写入磁盘，以防止数据丢失。
- 高可用性：Redis提供了主从复制和哨兵机制，可以实现数据的高可用性和故障恢复。
- 数据结构丰富：Redis支持多种数据结构，如字符串、哈希、列表、集合和有序集合，使得它非常适合于各种应用场景。

2.2 常用场景

- 缓存：Redis可以用作高速缓存存储，将常用的数据存储在内存中，以提高读取性能。它可以有效减轻数据库的负载，加快网站或应用程序的响应速度。
- 会话存储：将用户会话数据存储在Redis中，以实现分布式和可扩展的会话管理。这样可以实现无状态的应用服务器，提高应用程序的扩展性和性能。
- 计数器和排行榜：使用Redis的原子操作和有序集合，可以实现计数器和排行榜功能。例如，统计网站的访问次数或实时更新热门文章的排行。
- 分布式锁：利用Redis的原子性操作和过期时间设置，可以实现分布式锁的功能，用于协调多个应用程序之间的资源访问。

总之，Redis是一个功能强大的内存数据库，适用于各种场景，包括缓存、会话存储、计数器和排行榜、消息队列等。它的快速性能、灵活的数据结构和丰富的功能使其成为许多应用程序中常用的数据存储解决方案。

2.3 Go三方库

- `github.com/go-redis/redis`：是一个功能丰富且易于使用的Redis客户端库，提供了各种操作Redis的方法，包括数据读写、事务、发布/订阅等。
- GitHub链接：`https://github.com/go-redis/redis`

2.4 Redis使用

1. 安装依赖包

```
go get github.com/go-redis/redis
```

2. 初始化Client

```
client := redis.NewClient(&redis.Options{
    Addr:         "0.0.0.0:6379",
    Password:     "",
    DB:           0,
    WriteTimeout: 600 * time.Millisecond, //写超时时间
    ReadTimeout:  300 * time.Millisecond, //读超时时间
    DialTimeout:  3 * time.Minute,        //连接超时时间
    PoolSize:     10,                     //最大连接数
    MinIdleConns: 3,                      //最小空闲连接数
    IdleTimeout:  1 * time.Minute,        //空闲连接超时时间
})
```

3. 初始化参数详解

```
type Options struct {
    // 网络类型 tcp 或者 unix.
    // 默认是 tcp.
    Network string
    // redis地址，格式 host:port
    Addr string

    // 新建一个redis连接的时候，会回调这个函数
    OnConnect func(*Conn) error

    // redis密码，redis server没有设置可以为空。
    Password string
```

```
    // redis数据库，序号从0开始，默认是0，可以不用设置
    DB int

    // redis操作失败最大重试次数，默认不重试。
    MaxRetries int

    // 最小重试时间间隔.
    // 默认是 8ms ; -1 表示关闭.
    MinRetryBackoff time.Duration

    // 最大重试时间间隔
    // 默认是 512ms; -1 表示关闭.
    MaxRetryBackoff time.Duration

    // redis连接超时时间.
    // 默认是 5 秒.
    DialTimeout time.Duration

    // socket读取超时时间
    // 默认 3 秒.
    ReadTimeout time.Duration

    // socket写超时时间
    WriteTimeout time.Duration

    // redis连接池的最大连接数.
    // 默认连接池大小等于 cpu个数 * 10
    PoolSize int

    // redis连接池最小空闲连接数.
    MinIdleConns int
    // redis连接最大的存活时间，默认不会关闭过时的连接.
    MaxConnAge time.Duration

    // 当你从redis连接池获取一个连接之后，连接池最多等待这个拿出去的连接多长时
间。
    // 默认是等待 ReadTimeout + 1 秒.
    PoolTimeout time.Duration
    // redis连接池多久会关闭一个空闲连接.
```

```
    // 默认是 5 分钟. -1 则表示关闭这个配置项
    IdleTimeout time.Duration
    // 多长时间检测一下，空闲连接
    // 默认是 1 分钟. -1 表示关闭空闲连接检测
    IdleCheckFrequency time.Duration

    // 只读设置，如果设置为true， redis只能查询缓存不能更新。
    readOnly bool
}
```

4. Redis 基本键值用法

- 函数解析

函数	功能
Set	设置一个key的值
Get	查询key的值
GetSet	设置一个key的值，并返回这个key的旧值
SetNX	如果key不存在，则设置这个key的值
MGet	批量查询key的值
MSet	批量设置key的值
Incr,IncrBy,IncrByFloat	针对一个key的数值进行递增操作
Decr,DecrBy	针对一个key的数值进行递减操作
Del	删除key操作，可以批量删除
Expire	设置key的过期时间

- 代码示例

```
// 第三个参数代表key的过期时间，0代表不会过期。
err := client.Set("key", "value", 0).Err()
if err != nil {
    panic(err)
}
```

```
// Result函数返回两个值，第一个是key的值，第二个是错误信息
val, err := client.Get("key").Result()
// 判断查询是否出错
if err != nil {
    panic(err)
}
fmt.Println("key", val)

// 设置一个key的值，并返回这个key的旧值
oldVal, err := client.GetSet("key", "new value").Result()
if err != nil {
    panic(err)
}
fmt.Println("key", oldVal)

// 如果key不存在，则设置这个key的值
err := client.SetNX("key", "value", 0).Err()
if err != nil {
    panic(err)
}

// MGet函数可以传入任意个key，一次性返回多个值。
vals, err := client.MGet("key1", "key2", "key3").Result()
if err != nil {
    panic(err)
}
fmt.Println(vals)

//批量设置Key,不能设置过期时间
err := client.MSet("key1", "value1", "key2", "value2", "key3",
"value3").Err()
if err != nil {
  panic(err)
}

// Incr函数每次加一,value必须为整数
val, err := client.Incr("key").Result()
if err != nil {
    panic(err)
```

```
}
fmt.Println("最新值", val)

// IncrBy函数，可以指定每次递增多少
val, err := client.IncrBy("key", 2).Result()
if err != nil {
    panic(err)
}
fmt.Println("最新值", val)

// IncrByFloat函数，可以指定每次递增多少，跟IncrBy的区别是累加的是浮点数
val, err := client.IncrByFloat("key", 2).Result()
if err != nil {
    panic(err)
}
fmt.Println("最新值", val)

// Decr函数每次减一
val, err := client.Decr("key").Result()
if err != nil {
    panic(err)
}
fmt.Println("最新值", val)

// DecrBy函数，可以指定每次递减多少
val, err := client.DecrBy("key", 2).Result()
if err != nil {
    panic(err)
}
fmt.Println("最新值", val)

// 删除key
client.Del("key")

// 删除多个key，Del函数支持删除多个key
err := client.Del("key1", "key2", "key3").Err()
if err != nil {
    panic(err)
}
```

```
//设置key的过期时间
err := client.Expire("key", 3).Err()
if err != nil {
    panic(err)
}
```

5. Redis hash用法

在Redis中，Hash（哈希）是一种数据类型，它类似于一个键值对的集合。在Hash中，每个键都与一个值相关联，这些键值对被存储在一个哈希表中。Redis的Hash提供了高效的存储和访问方式，适用于存储和操作具有结构化数据的场景。

redis hash操作主要有2-3个元素组成：

- key - redis key 唯一标识
- field - hash数据的字段名
- value - 值，有些操作不需要值

- 函数解析

函数	功能
HSet	根据key和field字段设置，field字段的值
HGet	根据key和field字段，查询field字段的值
HGetAll	根据key查询所有字段和值
HIncrBy	根据key和field字段，累加数值
HKeys	根据key返回所有字段名
HLen	根据key，查询hash的字段数量
HMGet	根据key和多个字段名，批量查询多个hash字段值
HMSet	根据key和多个字段名和字段值，批量设置hash字段值
HSetNX	如果field字段不存在，则设置hash字段值
HDel	根据key和字段名，删除hash字段，支持批量删除hash字段
HExists	检测hash字段名是否存在

- 代码示例

```
// 根据key和field字段设置，field字段的值
// user_1 是hash key，username 是字段名，tizi365是字段值
err := client.HSet("user_1", "username", "tizi365").Err()
if err != nil {
    panic(err)
}

// 根据key和field字段，查询field字段的值
username, err := client.HGet("user_1", "username").Result()
if err != nil {
    panic(err)
}
fmt.Println(username)

// 根据key查询所有字段和值
data, err := client.HGetAll("user_1").Result()
if err != nil {
    panic(err)
}
// data是一个map类型，这里使用使用循环迭代输出
```

```go
for field, val := range data {
    fmt.Println(field,val)
}

// 根据key和field字段，累加字段的数值
// 累加count字段的值，一次性累加2， user_1为hash key
count, err := client.HIncrBy("user_1", "count", 2).Result()
if err != nil {
    panic(err)
}
fmt.Println(count)

// 根据key返回所有field字段名
// keys是一个string数组
keys, err := client.HKeys("user_1").Result()
if err != nil {
    panic(err)
}
fmt.Println(keys)

// 根据key，查询field字段数量
size, err := client.HLen("user_1").Result()
if err != nil {
    panic(err)
}
fmt.Println(size)

// 根据key和多个字段名，批量查询多个hash字段值
vals, err := client.HMGet("user_1","username", "count").Result()
if err != nil {
    panic(err)
}
// vals是一个数组
fmt.Println(vals)

// 如果field字段不存在，则设置hash字段值
err := client.HSetNX("user_1", "id", 100).Err()
if err != nil {
    panic(err)
```

```
}

// 删除一个字段id
client.HDel("user_1", "id")
// 删除多个字段
client.HDel("user_1", "id", "username")

// 检测hash字段名是否存在
val, err := client.HExists("user_1", "count").Result()
  if err != nil {
    panic(err)
  }
fmt.Println(val)
```

6. Redis 列表(list)用法

Redis列表是简单的字符串列表，列表是有序的，列表中的元素可以重复。可以添加一个元素到列表的头部（左边）或者尾部（右边）

- 函数解析

函数	功能
LPush	从列表左边插入数据
LPushX	跟LPush的区别是，仅当列表存在的时候才插入数据
RPop	从列表的右边删除第一个数据，并返回删除的数据
RPush	从列表右边插入数据
RPushX	跟RPush的区别是，仅当列表存在的时候才插入数据
LPop	从列表左边删除第一个数据，并返回删除的数据
LLen	返回列表的大小
LRange	返回列表的一个范围内的数据，也可以返回全部数据
LRem	删除列表中的数据
LIndex	根据索引坐标，查询列表中的数据
LInsert	在指定位置插入数据

- 代码示例

```
// 从列表左边插入数据
// 插入一个数据
client.LPush("key01", "data1")
// LPush支持一次插入任意个数据
err := client.LPush("key01", 1,2,3,4,5).Err()
if err != nil {
    panic(err)
}

// 从列表的右边删除第一个数据，并返回删除的数据
val, err := client.RPop("key01").Result()
if err != nil {
    panic(err)
}
fmt.Println(val)

// 从列表右边插入数据
// 插入一个数据
client.RPush("key01", "data1")
// 支持一次插入任意个数据
```

```go
err := client.RPush("key01", 1,2,3,4,5).Err()
if err != nil {
    panic(err)
}

// 从列表左边删除第一个数据，并返回删除的数据
val, err := client.LPop("key01").Result()
if err != nil {
    panic(err)
}
fmt.Println(val)

// 返回列表的大小
val, err := client.LLen("key01").Result()
if err != nil {
    panic(err)
}
fmt.Println(val)

// 返回列表的一个范围内的数据，也可以返回全部数据
vals, err := client.LRange("key01",0,-1).Result()
if err != nil {
    panic(err)
}
fmt.Println(vals)

// 删除列表中的数据
// 从列表左边开始，删除100， 如果出现重复元素，仅删除1次，也就是删除第一个
dels, err := client.LRem("key01", 1, 5).Result()
  if err != nil {
    panic(err)
  }
  fmt.Println(dels)

// 根据索引坐标，查询列表中的数据
val, err := client.LIndex("key01",5).Result()
if err != nil {
    panic(err)
}
```

```
fmt.Println(val)

// 在指定位置插入数据
// 在列表中 data1 元素的前面插入 欢迎你
client.LInsert("key01","before", "data1", "欢迎你")
// 在列表中 data1 元素的后面插入 2023
client.LInsert("key01","after", "data1", "2023")
```

7. Redis 集合(set)用法

redis的set类型（集合）是string类型数值的无序集合，并且集合元素唯一。

- 函数解析

函数	功能
SAdd	添加集合元素
SCard	获取集合元素个数
SIsMember	判断元素是否在集合中
SMembers	获取集合中所有的元素
SRem	删除集合元素
SPop,SPopN	随机返回集合中的元素，并且删除返回的元素

- 代码示例

```
// 添加集合元素
err := client.SAdd("key02",100).Err()
if err != nil {
    panic(err)
}
client.SAdd("key02",100, 200, 300)
```

```
// 获取集合元素个数
size, err := client.SCard("key02").Result()
if err != nil {
    panic(err)
}
fmt.Println(size)

// 判断元素是否在集合中
ok, _ := client.SIsMember("key02", 100).Result()
if ok {
    fmt.Println("集合包含指定元素")
}

// 获取集合中所有的元素
res, _ := client.SMembers("key02").Result()
// 返回的es是string数组
fmt.Println(res)

// 删除集合元素
client.SRem("key02", "100", "200")

// 随机返回集合中的元素，并且删除返回的元素
// 随机返回集合中的一个元素，并且删除这个元素
val, _ := client.SPop("key02").Result()
fmt.Println(val)
// 随机返回集合中的5个元素，并且删除这些元素
vals, _ := client.SPopN("key02", 5).Result()
fmt.Println(vals)
```

8. Redis 有序集合(sorted set)用法

Redis 有序集合（sorted set）和集合一样也是string类型元素的集合,且不允许重复的成员，不同的是每个元素都会关联一个double类型的分数，这个分数主要用于集合元素排序，分数越高排序优先级则越高。

- 函数解析

函数	功能
ZAdd	添加一个或者多个元素到集合，如果元素已经存在则更新分数
ZCard	返回集合元素个数
ZCount	统计某个分数范围内的元素个数
ZIncrBy	增加元素的分数
ZRange,ZRevRange	返回集合中某个索引范围的元素，根据分数从小到大排序
ZRangeByScore,ZRevRangeByScore	根据分数范围返回集合元素，元素根据分数从小到大排序，支持分页
ZRem	删除集合元素
ZRemRangeByRank	根据索引范围删除元素
ZRemRangeByScore	根据分数范围删除元素
ZScore	查询元素对应的分数
ZRank, ZRevRank	查询元素的排名

- 代码示例

```
// 添加一个或者多个元素到集合，如果元素已经存在则更新分数
err := client.ZAdd("key03", redis.Z{2.5,"tizi"}).Err()
if err != nil {
    panic(err)
}

// 返回集合元素个数
size, err := client.ZCard("key03").Result()
if err != nil {
    panic(err)
}
fmt.Println(size)
```

```go
// 统计某个分数范围内的元素个数
// 返回： 1<=分数<=5 的元素个数, 注意："1", "5"两个参数是字符串
size, err := client.ZCount("key03", "1","5").Result()
if err != nil {
    panic(err)
}
fmt.Println(size)

// 增加元素的分数
// 给元素tizi，加上2分
client.ZIncrBy("key03", 2, "tizi")

// 返回集合中某个索引范围的元素，根据分数从小到大排序
// 返回从0到-1位置的集合元素， 元素按分数从小到大排序
// 0到-1代表则返回全部数据
vals, err := client.ZRange("key03", 0,-1).Result()
if err != nil {
    panic(err)
}
for _, val := range vals {
    fmt.Println(val)
}

// 根据分数范围返回集合元素，元素根据分数从小到大排序，支持分页。
// 初始化查询条件， Offset和Count用于分页
op := redis.ZRangeBy{
    Min:"2", // 最小分数
    Max:"10", // 最大分数
    Offset:0, // 类似sql的limit，表示开始偏移量
    Count:5, // 一次返回多少数据
}
vals, err := client.ZRangeByScore("key03", op).Result()
if err != nil {
    panic(err)
}
for _, val := range vals {
    fmt.Println(val)
}
```

```
// 删除集合元素
client.ZRem("key03", "tizi", "xiaoli")

// 根据索引范围删除元素
// 集合元素按分数排序，从最低分到高分，删除第0个元素到第5个元素。
// 这里相当于删除最低分的几个元素
client.ZRemRangeByRank("key03", 0, 5)

// 根据分数范围删除元素
// 删除范围： 2<=分数<=5 的元素
client.ZRemRangeByScore("key03", "2", "5")

// 查询集合元素tizi的分数
score, _ := client.ZScore("key03", "tizi").Result()
fmt.Println(score)

// 根据元素名，查询集合元素在集合中的排名，从0开始算，集合元素按分数从小到大排序
rk, _ := client.ZRank("key03", "tizi").Result()
fmt.Println(rk)
```

2.5 Redis事务

redis事务可以一次执行多个命令， 并且带有以下两个重要的保证：

- 事务是一个单独的隔离操作：事务中的所有命令都会序列化、按顺序地执行。事务在执行的过程中，不会被其他客户端发送来的命令请求所打断。
- 事务是一个原子操作：事务中的命令要么全部被执行，要么全部都不执行。

go redis事务常用函数：

- TxPipeline - 以Pipeline的方式操作事务
- Watch - redis乐观锁支持

1. TxPipeline

以Pipeline的方式操作事务

```
// Redis的事务和MySQL或者关系型数据库的事务有些差别，Redis的事务只是能够确保需要执行的多个命令能够单线程 // 一起执行，但是执行多个命令中间出现错误时，不会回滚。需要业务开发人员考虑如何处理失败场景的处理。
// 开启事务
  pipeline := client.TxPipeline()

  // 执行多个 Redis 命令
  pipeline.Get("key04")
  pipeline.Set("key05", "value05", 0)

  // 提交事务
  _, err := pipeline.Exec()
  if err != nil {
    // 发生错误，放弃事务
    fmt.Println("err:", err)
  } else {
    // 事务执行成功
    fmt.Println("Transaction successful")
  }
  val, _ := client.Get("key05").Result()
  fmt.Println(val)
```

2. watch

redis乐观锁支持，可以通过watch监听一些Key, 如果这些key的值没有被其他人改变的话，才可以提交事务。

```
// 定义一个回调函数，用于处理事务逻辑
  fn := func(tx *redis.Tx) error {
    // 先查询下当前watch监听的key的值
    v, err := tx.Get("key07").Result()
    if err != nil && err != redis.Nil {
      return err
```

```
    }

    // 这里可以处理业务
    fmt.Println(v)

    // 如果key的值没有改变的话，Pipelined函数才会调用成功
    _, err = tx.Pipelined(func(pipe redis.Pipeliner) error {
      // 在这里给key设置最新值
      pipe.Set("key07", "new value", 0)
      pipe.Get("key100")
      return nil
    })
    return err
  }

  // 使用Watch监听一些Key，同时绑定一个回调函数fn，监听Key后的逻辑写在fn这个回调函数里面
  // 如果想监听多个key，可以这么写：client.Watch(fn, "key1", "key2", "key3")
  err := client.Watch(fn, "key07")
  if err != nil {
    fmt.Println("err", err)
  }
```

2.5 案例：断点续查

断点续查是指在查询大量数据时，可以设置一个断点，记录已经获取的数据位置，以便下次继续查询。以下是一种基于 Redis 实现断点续查的一般方法：

- 定义断点键（Checkpoint Key）：您可以在 Redis 中设置一个键，用于存储断点位置。这个键可以是一个唯一的标识，用来跟踪断点的位置。
- 设置初始断点位置：在执行查询或处理操作之前，将初始断点位置存储在 Redis 中的断点键中。初始断点位置通常为 0 或某个合适的起始位置。
- 分页查询或处理：按照分页的方式执行查询或处理操作，每次处理一个固定大小的数据块。

- 更新断点位置：在处理完每个数据块后，将当前断点位置存储回 Redis 的断点键中，以便下次继续从断点位置开始。
- 循环处理：重复执行分页查询或处理的过程，直到处理完所有数据或达到终止条件。

```
package main

import (
  "context"
  "fmt"
  "github.com/go-redis/redis"
  "time"
)

func main() {
  client := redis.NewClient(&redis.Options{
    Addr:         "192.168.0.14:6379",
    Password:     "Hangzhou@123",
    DB:           0,
    WriteTimeout: 600 * time.Millisecond, //写超时时间
    ReadTimeout:  300 * time.Millisecond, //读超时时间
    DialTimeout:  3 * time.Minute,        //连接超时时间
    PoolSize:     10,                     //最大连接数
    MinIdleConns: 3,                      //最小空闲连接数
    IdleTimeout:  1 * time.Minute,        //空闲连接超时时间
  })

  //生成数据
  client.LPush("key10", "a", "b", "c", "d", "e", "f", "g").Err()
  //vals, _ := client.LRange("key10",0,-1).Result()
  //fmt.Println(vals)

  ctx := context.Background()
  checkpointKey := "my_checkpoint"
  pageSize := 2
  stopProcessing := false
  client.Set(checkpointKey, 1, 0)

  // 获取上次查询的断点位置（页数）
```

```
    page, err := client.Get(checkpointKey).Int()
    if err != nil && err != redis.Nil {
        fmt.Println("Error retrieving checkpoint:", err)
        return
    }

    // 处理数据直到终止条件满足
    for !stopProcessing {
        // 查询数据，从断点位置开始
        data, err := queryDataFromRedis(ctx, client, page, pageSize)
        if err != nil {
            fmt.Println("Error querying data:", err)
            return
        }

        // 处理数据，可以根据实际需求进行操作
        fmt.Println(data)

        // 更新断点位置（页数）
        page++
        err = client.Set(checkpointKey, page, 0).Err()
        if err != nil {
            fmt.Println("Error updating checkpoint:", err)
            return
        }

        // 检查是否达到终止条件
        if len(data) < pageSize {
            stopProcessing = true
        }
    }

    fmt.Println("Data processing completed.")
}

// 从 Redis 中查询数据并返回指定页数的数据块
func queryDataFromRedis(ctx context.Context, client *redis.Client,
page, pageSize int) ([]string, error) {
    // 实际查询数据的逻辑，根据需要自行实现
```

```
  // 假设这里的数据是 List，每页存储 pageSize 条数据
  start := (page - 1) * pageSize
  end := page * pageSize - 1

  results, err := client.LRange("key10", int64(start),
int64(end)).Result()
  if err != nil {
    return nil, err
  }

  return results, nil
}
```

三、Elasticsearch使用

3.1 介绍

Elasticsearch 是一个开源的分布式搜索和分析引擎，构建在 Apache Lucene 基础之上。它提供了强大的全文搜索、结构化查询、分析能力和实时数据分析功能，被广泛应用于各种领域，包括企业搜索、日志分析、产品搜索、监控数据分析等。

以下是 Elasticsearch 的一些主要特点和功能：

- 分布式和高可用性：Elasticsearch 是一个分布式的搜索引擎，它可以在多个节点上存储和处理数据，具备高可用性和容错性。数据可以分片和复制到多个节点，从而实现数据的水平扩展和故障恢复。
- 实时搜索和分析：Elasticsearch 提供了实时搜索和分析功能，可以在大规模数据集上快速进行搜索、聚合和分析。它支持全文搜索、近实时的数据索引和搜索，以及复杂的查询和聚合操作。
- 多样化的查询和聚合：Elasticsearch 提供了丰富的查询语言和灵活的聚合功能，使用户可以进行复杂的数据检索和分析。您可以使用诸如匹配查询、范围查询、过

滤器、聚合等查询和分析操作来探索数据集。

- 多种数据类型支持：Elasticsearch 支持多种数据类型，包括文本、数值、日期、地理位置等。它具有强大的全文搜索功能，可以处理各种语言和文本分析需求。
- 高性能和可扩展性：Elasticsearch 基于 Apache Lucene 引擎，具有高性能的搜索和索引功能。它可以水平扩展，通过增加节点来处理大规模数据和高并发请求。
- 集成和生态系统：Elasticsearch 提供了丰富的 API 和工具，方便与其他系统进行集成。它与 Logstash、Kibana 和 Beats 等工具集成，形成了 ELK Stack（现在称为 Elastic Stack），用于日志分析和实时数据处理。
- 安全性和权限控制：Elasticsearch 提供了访问控制、身份验证和权限管理等安全功能，以保护数据的机密性和完整性。

3.2 常用场景

- 搜索引擎：Elasticsearch 提供了高效的全文搜索和实时搜索功能，使其成为构建搜索引擎和相关应用的理想选择。它可以轻松处理大规模的文本数据，并提供强大的搜索、过滤、排序和高亮显示等功能。
- 日志和事件分析：Elasticsearch 被广泛用于实时日志和事件数据分析。它可以接收、索引和存储大量的日志数据，并提供快速的搜索和聚合能力，使您能够以实时方式分析日志数据并提取有用的见解。
- 监控和指标分析：Elasticsearch 可以用作实时监控和指标数据的存储和分析引擎。它可以接收和处理大量的指标数据，并提供灵活的聚合和可视化功能，以帮助监控系统性能、应用程序指标和基础设施指标。
- 商业智能和数据分析：Elasticsearch 与 Kibana 结合使用，可以构建强大的商业智能和数据分析平台。它可以处理和分析结构化和非结构化数据，提供高级的数据聚合、过滤和可视化功能，使用户能够发现数据中的模式和洞察，并进行深入的数据分析。
- 实时推荐系统：Elasticsearch 的实时性能和搜索功能使其成为实时推荐系统的理想选择。它可以接收实时事件数据，并通过实时搜索和聚合来生成个性化的推荐结果，提供更好的用户体验和个性化的推荐服务。
- 地理空间分析：Elasticsearch 具有强大的地理空间搜索和分析功能，可以处理和分析与地理位置相关的数据。它支持地理坐标索引、距离计算、地理范围查询等功能，使其适用于地理空间数据分析和地理信息系统（GIS）应用。

3.3 Go三方库

- `github.com/olivere/elastic/v7`：是一个用于与 Elasticsearch 进行交互的 Go 客户端库。它提供了方便的 API，使您能够索引、检索、更新和删除 Elasticsearch 中的文档，以及执行聚合和搜索操作。
- Github链接：`https://github.com/olivere/elastic`

3.4 Elasitcsearch使用

1. 安装依赖包

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

2. 初始化Client

```
package main

import (
  "fmt"
  "github.com/olivere/elastic/v7"
  "log"
)

func main() {
  // 创建 Elasticsearch 客户端
  client, err := elastic.NewClient(
    elastic.SetSniff(false),
    elastic.SetURL("http://192.168.0.14:9200"),
  )
  if err != nil {
    log.Fatalf("Error creating the client: %s", err)
  }
  fmt.Println(client)
  // 使用 Elasticsearch 客户端进行操作...
}
```

3. 初始化参数详解

- `elastic.SetURL(url)`用来设置`ES`服务地址，如果是本地，就是`127.0.0.1:9200`。支持多个地址，用逗号分隔即可。
- `elastic.SetBasicAuth("user", "secret")`这个是基于http base auth 验证机制的账号密码。
- `elastic.SetGzip(true)`启动`gzip`压缩
- `elastic.SetHealthcheckInterval(10*time.Second)`用来设置监控检查时间间隔
- `elastic.SetMaxRetries(5)`设置请求失败最大重试次数，v7版本以后已被弃用
- `elastic.SetSniff(false)`允许指定弹性是否应该定期检查集群（默认为true）
- `elastic.SetErrorLog(log.New(os.Stderr, "ELASTIC ", log.LstdFlags))`设置错误日志输出
- `elastic.SetInfoLog(log.New(os.Stdout, "", log.LstdFlags))`设置info日志输出

4. 索引的CRUD

```
package main

import (
  "context"
  "fmt"
  "github.com/olivere/elastic/v7"
  "log"
)

var (
  //索引名
  indexName = "your_index"
  //索引映射关系
  mapping = `{
    "mappings":{
      "dynamic": "strict",
      "properties":{
        "id":          { "type": "long" },
        "username":    { "type": "keyword" },
```

```
        "nickname":     { "type": "text" },
        "phone":      { "type": "keyword" },
        "age":        { "type": "integer" },
        "ancestral":    { "type": "text" },
        "identity":         { "type": "text" },
        "update_time":    { "type": "long" },
        "create_time":    { "type": "long" }
      }
    },
    "settings" : {
      "index" : {
        "number_of_shards" : "1",
        "number_of_replicas" : "1"
      }
    }
  }`
)

var ESServerURL = []string{"http://192.168.0.14:9200"}

func main() {
  // 创建 Elasticsearch 客户端
  //初始化es连接
  client, err := elastic.NewClient(
    elastic.SetSniff(false),
    elastic.SetURL("http://192.168.0.14:9200"))
  if err != nil {
    log.Fatalf("Error creating the client: %s", err)
  }
  fmt.Println(client)

  //创建索引
  _, err =
client.CreateIndex(indexName).BodyString(mapping).Do(context.Backgrou
nd())
  if err != nil {
    fmt.Println("Failed to create index:", err)
    return
  }
```

```
  //删除索引
  _, err = client.DeleteIndex(indexName).Do(context.Background())
  if err != nil {
    // 处理删除索引错误
    fmt.Println("Failed to delete index:", err)
    return
  }

  //新增索引字段，对于已存在的字段无法修改
  newMapping := `
    {
      "properties": {
        "age2": {
          "type": "keyword",
          "index": false
        },
        "age3": {
          "type": "text",
          "analyzer": "standard"
        }
      }
    }`
  _, err =
client.PutMapping().Index(indexName).BodyString(newMapping).Do(contex
t.Background())
  if err != nil {
    fmt.Println("Failed to update mapping:", err)
    return
  }

  //迁移索引，迁移前必须保证targetIndex存在
  sourceIndex := "my_index"
  targetIndex := "your_index"

  // 创建 Reindex 请求体
  reindexService :=
client.Reindex().SourceIndex(sourceIndex).DestinationIndex(targetInde
x)
```

```
  // 执行 Reindex 操作
  response, err := reindexService.Do(context.Background())
  if err != nil {
    fmt.Println(err)
  }
  fmt.Println(response)

  //查询索引
  searchResult, err := client.Search().
    Index(indexName).
    Query(elastic.NewMatchAllQuery()).
    Do(context.Background())
  if err != nil {
    // 处理查询错误
    fmt.Println("Failed to execute search query:", err)
    return
  }
  fmt.Println(searchResult)
  //查看搜索到的文件数量
  if searchResult.Hits.TotalHits.Value > 0 {
    fmt.Printf("Found %d documents in index %s\n",
searchResult.Hits.TotalHits.Value, indexName)
    for _, hit := range searchResult.Hits.Hits {
      //hit.Score 表示文档的相关性分数，用于衡量文档与搜索查询的匹配程度
      //hit.Source 是文档的原始源数据，它包含了文档的所有字段和对应的值
      fmt.Printf("Document ID: %s, Score: %f, Source: %s\n", hit.Id,
*hit.Score, hit.Source)
    }
  } else {
    fmt.Println("No documents found in index:", indexName)
  }

}
```

5. 文档的CRUD

```
package main

import (
  "context"
  "fmt"
  "github.com/olivere/elastic/v7"
  "log"
)
var (
  //索引名
  indexName = "my_index"
)

type MyDocument struct {
  ID         int64  `json:"id"`
  Username   string `json:"username"`
  Nickname   string `json:"nickname"`
  Phone      string `json:"phone"`
  Age        int64  `json:"age"`
  Ancestral  string `json:"ancestral"`
  Identity   string `json:"identity"`
}
func main() {
  // 创建 Elasticsearch 客户端
  client, err := elastic.NewClient(
    elastic.SetSniff(false),
    elastic.SetURL("http://192.168.0.14:9200"),
  )
  if err != nil {
    log.Fatalf("Error creating the client: %s", err)
  }
  fmt.Println(client)

  //定义文档内容
  doc := &MyDocument{
    ID:         1,
```

```
    Username:   "john",
    Nickname:   "John Doe",
    Phone:      "123456789",
    Age:        30,
    Ancestral:  "English",
    Identity:   "Employee",
  }
  //创建文档
  resp, err := client.Index().
    Index(indexName).
    OpType("create").
    BodyJson(doc).
    Do(context.Background())
  if err != nil {
    fmt.Println("Failed to create document:", err)
    return
  }
  fmt.Println(resp.Id)

  //批量创建文档
  docs := []*MyDocument{
    {
      ID:         2,
      Username:   "alice",
      Nickname:   "Alice Johnson",
      Phone:      "555555555",
      Age:        35,
      Ancestral:  "French",
      Identity:   "Manager",
    },
    {
      ID:         3,
      Username:   "jane",
      Nickname:   "Jane Smith",
      Phone:      "987654321",
      Age:        28,
      Ancestral:  "American",
      Identity:   "Employee",
    },
```

```
  }
  //创建bulkservice
  bulkRequest := client.Bulk().Index(indexName)
  //添加文档
  for _, doc := range docs {
    bulkRequest.Add(elastic.NewBulkIndexRequest().Doc(doc))
  }
  //发起批量请求
  bulkResponse, err := bulkRequest.Do(context.Background())
  if err != nil {
    fmt.Println("Failed to create documents:", err)
    return
  }

  if len(bulkResponse.Failed()) > 0 {
    for _, item := range bulkResponse.Failed() {
      fmt.Printf("Failed to create document: %s\n",
item.Error.Reason)
    }
  }

  //根据ID查询
  id := "X5BehogBWQOXvTp6mdWH"
  result, _ :=
client.Get().Index(indexName).Id(id).Do(context.Background())
  fmt.Println(string(result.Source))

  //批量ID查询
  ids := []string{"XpBehogBWQOXvTp6mdVl", "X5BehogBWQOXvTp6mdWH"}
  mgetSvc := client.MultiGet()
  for _, id := range ids {
    mgetSvc.Add(elastic.NewMultiGetItem().
      Index(indexName).
      Id(id))
  }
  result2, _ := mgetSvc.Do(context.Background())
  fmt.Println(result2.Docs)

  // 创建 terms 查询条件，用于多值精确查询
```

```
    termsQuery := elastic.NewTermsQuery("username", "john", "alice")
    result3, _ := client.Search().
        Index(indexName).           // 设置索引名
        Query(termsQuery).          // 设置查询条件
        From(0).                    // 设置分页参数 - 起始偏移量，从第 0 行记录
开始
        Size(10).                   // 设置分页参数 - 每页大小
        Do(context.Background())    // 执行请求
    fmt.Println(len(result3.Hits.Hits))

    rangeQuery := elastic.NewRangeQuery("age").Gte(18).Lte(35)
    result4, _ := client.Search().
        Index(indexName).           // 设置索引名
        Query(rangeQuery).          // 设置查询条件
        From(0).                    // 设置分页参数 - 起始偏移量，从第 0 行记录
开始
        Size(10).                   // 设置分页参数 - 每页大小
        Do(context.Background())    // 执行请求
    fmt.Println(len(result4.Hits.Hits))

    //bool组合查询
    boolQuery := elastic.NewBoolQuery()

    //创建查询条件
    termQuery := elastic.NewTermQuery("username", "alice")
    rangeQuery = elastic.NewRangeQuery("age").Gte(18).Lte(35)

    //设置 bool 查询的 must 条件，组合了两个子查询
    //搜索用户名为 bob 且年龄在 18～35 岁的用户
    boolQuery.Must(termQuery, rangeQuery)

    result5, _ := client.Search().
        Index(indexName). // 设置索引名
        Query(boolQuery).           // 设置查询条件
        From(0).                    // 设置分页参数 - 起始偏移量，从第 0 行记录
开始
        Size(10).                   // 设置分页参数 - 每页大小
        Do(context.Background())    // 执行请求
    if result5.Hits.TotalHits.Value > 0 {
```

```
    fmt.Printf("Found %d documents in index %s\n",
result5.Hits.TotalHits.Value, indexName)
    for _, hit := range result5.Hits.Hits {
      //hit.Score 表示文档的相关性分数，用于衡量文档与搜索查询的匹配程度
      //hit.Source 是文档的原始源数据，它包含了文档的所有字段和对应的值
      fmt.Printf("Document ID: %s, Score: %f, Source: %s\n", hit.Id,
*hit.Score, hit.Source)
    }
  } else {
    fmt.Println("No documents found in index:", indexName)
  }

  //根据ID删除
  //Refresh("true")立即刷新索引
  result6, _ :=
client.Delete().Index(indexName).Id(id).Refresh("true").Do(context.Ba
ckground())
  fmt.Println(result6)

  bulkService := client.Bulk().Index(indexName).Refresh("true")
  for i := range ids {
    req := elastic.NewBulkDeleteRequest().Id(ids[i])
    bulkService.Add(req)
  }
  result7, _ := bulkService.Do(context.Background())
  fmt.Println(result7)

  //修改文档
  doc = &MyDocument{
    ID:       1,
    Username:   "john",
    Nickname:   "John Doe",
    Phone:      "123456789",
    Age:        30,
    Ancestral:  "English",
    Identity:   "Employee",
  }
  result9, _ := client.
    Update().
```

```
    Index(indexName).
    Id(id).
    Doc(doc).
    Refresh("true").
    Do(context.Background())
  fmt.Println(result9)

  //根据条件修改
  //index 索引; query 条件; script 脚本指定待更新的字段与值
  query2 := elastic.NewBoolQuery()
  query2.Filter(elastic.NewTermQuery("username", "alice"))
  query2.Filter(elastic.NewRangeQuery("age").Lte(36))
  script :=
elastic.NewScriptInline("ctx._source.nickname=params.nickname;ctx._so
urce.ancestral=params.ancestral").Params(
    map[string]interface{}{
      "nickname":  "cat",
      "ancestral": "Shanghai",
    })
  result10, _ := client.
    UpdateByQuery(indexName).
    Query(query2).
    Script(script).
    Refresh("true").
    Do(context.Background())
  fmt.Println(result10)
}
```

3.5 案例：日志分析

- `access.log`

```
```

```
192.168.1.1 - - [12/Nov/2022:10:00:00 +0000] "GET /path/to/resource
HTTP/1.1" 200 1000 "-" "Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/96.0.4664.93
Safari/537.36"
192.168.1.2 - - [12/Nov/2022:10:01:23 +0000] "POST /api/endpoint
HTTP/1.1" 403 0 "-" "Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/96.0.4664.93
Safari/537.36"
192.168.1.3 - - [12/Nov/2022:10:02:45 +0000] "GET
/path/to/another/resource HTTP/1.1" 404 0 "-" "Mozilla/5.0 (Windows
NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/96.0.4664.93 Safari/537.36"
192.168.1.1 - - [12/Nov/2022:10:00:00 +0000] "GET /path/to/resource
HTTP/1.1" 200 1000 "-" "Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/96.0.4664.93
Safari/537.36"
192.168.1.2 - - [12/Nov/2022:10:01:23 +0000] "POST /api/endpoint
HTTP/1.1" 403 0 "-" "Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/96.0.4664.93
Safari/537.36"
192.168.1.3 - - [12/Nov/2022:10:02:45 +0000] "GET
/path/to/another/resource HTTP/1.1" 404 0 "-" "Mozilla/5.0 (Windows
NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/96.0.4664.93 Safari/537.36"
192.168.1.1 - - [12/Nov/2022:10:00:00 +0000] "GET /path/to/resource
HTTP/1.1" 200 1000 "-" "Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/96.0.4664.93
Safari/537.36"
192.168.1.2 - - [12/Nov/2022:10:01:23 +0000] "POST /api/endpoint
HTTP/1.1" 403 0 "-" "Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/96.0.4664.93
Safari/537.36"
192.168.1.3 - - [12/Nov/2022:10:02:45 +0000] "GET
/path/to/another/resource HTTP/1.1" 404 0 "-" "Mozilla/5.0 (Windows
NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/96.0.4664.93 Safari/537.36"
```

```
192.168.1.1 - - [12/Nov/2022:10:00:00 +0000] "GET /path/to/resource
HTTP/1.1" 200 1000 "-" "Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/96.0.4664.93
Safari/537.36"
192.168.1.2 - - [12/Nov/2022:10:01:23 +0000] "POST /api/endpoint
HTTP/1.1" 403 0 "-" "Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/96.0.4664.93
Safari/537.36"
192.168.1.3 - - [12/Nov/2022:10:02:45 +0000] "GET
/path/to/another/resource HTTP/1.1" 404 0 "-" "Mozilla/5.0 (Windows
NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/96.0.4664.93 Safari/537.36"
192.168.1.1 - - [12/Nov/2022:10:00:00 +0000] "GET /path/to/resource
HTTP/1.1" 200 1000 "-" "Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/96.0.4664.93
Safari/537.36"
192.168.1.2 - - [12/Nov/2022:10:01:23 +0000] "POST /api/endpoint
HTTP/1.1" 403 0 "-" "Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/96.0.4664.93
Safari/537.36"
192.168.1.3 - - [12/Nov/2022:10:02:45 +0000] "GET
/path/to/another/resource HTTP/1.1" 404 0 "-" "Mozilla/5.0 (Windows
NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/96.0.4664.93 Safari/537.36"
```

- 日志写入

```
package main

import (
  "bufio"
  "context"
  "fmt"
  "github.com/olivere/elastic/v7"
  "log"
  "os"
  "strconv"
```

```
  "strings"
  "time"
)

type AccessLog struct {
  Timestamp    time.Time `json:"timestamp"`
  IP           string    `json:"ip"`
  Method       string    `json:"method"`
  URL          string    `json:"url"`
  StatusCode   int       `json:"status_code"`
  ResponseTime int       `json:"response_time"`
  UserAgent    string    `json:"user_agent"`
}


func main() {
  // 创建 Elasticsearch 客户端
  client, err := elastic.NewClient(
    elastic.SetSniff(false),
    elastic.SetURL("http://192.168.0.14:9200"),
  )
  if err != nil {
    log.Fatalf("Error creating the client: %s", err)
  }
  fmt.Println(client)

  // 打开 Access Log 文件
  file, err := os.Open("./demo/access.log")
  if err != nil {
    log.Fatalf("Error opening the file: %s", err)
  }
  defer file.Close()

  // 逐行读取 Access Log 文件并解析
  scanner := bufio.NewScanner(file)
  for scanner.Scan() {
    line := scanner.Text()
    accessLog := parseAccessLog(line)
```

```
    if accessLog != nil {
      indexAccessLog(client, accessLog)
    }
  }

  if err := scanner.Err(); err != nil {
    log.Fatalf("Error reading the file: %s", err)
  }

  log.Println("Access logs indexing completed")
}

// 解析单行 Access Log
func parseAccessLog(line string) *AccessLog {
  // 解析 Access Log 行并返回 AccessLog 结构体
  // 请根据实际的日志格式进行解析

  parts := strings.Split(line, " ")
  if len(parts) < 7 {
    return nil
  }

  timestamp, err := time.Parse("02/Jan/2006:15:04:05", parts[3][1:])
  if err != nil {
    log.Printf("Error parsing timestamp: %s", err)
    return nil
  }

  return &AccessLog{
    Timestamp:    timestamp,
    IP:           parts[0],
    Method:       parts[5][1:],
    URL:          parts[6],
    StatusCode:   parseStatusCode(parts[8]),
    ResponseTime: parseResponseTime(parts[9]),
    UserAgent:    strings.Join(parts[11:], " "),
  }
}
```

```
// 解析状态码
func parseStatusCode(code string) int {
  statusCode, _ := strconv.Atoi(code)
  return statusCode
}

// 解析响应时间
func parseResponseTime(timeStr string) int {
  responseTime, _ := strconv.Atoi(timeStr)
  return responseTime
}

// 将日志事件索引到 Elasticsearch
func indexAccessLog(client *elastic.Client, accessLog *AccessLog) {
  _, err := client.Index().
    Index("nginx-logs").
    BodyJson(accessLog).
    Refresh("true").
    Do(context.Background())
  if err != nil {
    log.Printf("Error indexing the access log: %s", err)
  }
}
```

- 分析各IP数量

```
package main

import (
  "context"
  "fmt"
  "log"

  elastic "github.com/olivere/elastic/v7"
)

func main() {
```

```
    // 创建 Elasticsearch 客户端
    client, err := elastic.NewClient(
        elastic.SetSniff(false),
        elastic.SetURL("http://192.168.0.14:9200"))
    if err != nil {
        log.Fatalf("Error creating the client: %s", err)
    }

    // 构建聚合查询
    aggs :=
elastic.NewTermsAggregation().Field("ip.keyword").Size(10000)

    searchResult, err := client.Search().
        Index("nginx-logs").
        Size(0).
        Aggregation("ips", aggs).
        Do(context.Background())
    if err != nil {
        log.Fatalf("Error executing the search: %s", err)
    }

    // 从聚合结果中获取 IP 数量
    ipsAgg, found := searchResult.Aggregations.Terms("ips")
    if found {
        fmt.Printf("Total IPs: %d\n", len(ipsAgg.Buckets))

        // 遍历每个 IP 桶，并输出 IP 和对应的文档数量
        for _, bucket := range ipsAgg.Buckets {
            fmt.Printf("IP: %s, Count: %d\n", bucket.Key, bucket.DocCount)
        }
    } else {
        fmt.Println("No IP aggregation found")
    }
}
```

四、Kafka使用

4.1 介绍

Kafka 是一种分布式流处理平台，最初由 LinkedIn 公司开发，现在由 Apache 软件基金会维护。它被设计用于高吞吐量、低延迟的数据传输，并具有可扩展性和容错性。

Kafka 的核心概念是消息传递系统，它通过将数据以消息的形式进行发布和订阅来实现数据流的处理。以下是 Kafka 的一些关键特点和概念：

- 消息发布和订阅：Kafka 使用发布-订阅模型，消息发布者将消息发送到一个或多个主题（Topics），而消息订阅者可以从这些主题中获取消息。
- 分布式架构：Kafka 是一个分布式系统，可以在多个节点上部署，实现数据的分区和复制，以实现高可用性和可扩展性。
- 消息持久化：Kafka 使用日志（Log）的方式持久化消息，每个主题的消息以追加的方式写入分区中，并且消息会在一段时间后根据配置进行保留。
- 分区和复制：Kafka 将每个主题分成一个或多个分区（Partitions），每个分区可以在多个节点上进行复制，以实现负载均衡和容错性。
- 高吞吐量：Kafka 具有非常高的吞吐量和低延迟，能够处理大量的消息流，并保持较低的延迟。
- 实时数据处理：Kafka 可以与流处理框架（如 Apache Spark、Apache Flink）集成，支持实时的数据处理和流式分析。

4.2 常用场景

- 数据流集成和数据管道：Kafka 可以作为数据流的中间件，用于连接不同的数据系统和应用程序。它可以集成多个数据源和数据目的地，实现可靠的数据传输和流水线处理。
- 日志收集和集中式日志管理：Kafka 可以用作日志收集系统，收集和存储分布在多个应用程序和服务器上的日志数据。它提供高吞吐量和持久化存储，支持实时的日志数据处理和分析。
- 事件驱动架构：Kafka 的发布-订阅模型使其成为构建事件驱动架构的理想选择。应用程序可以将事件发布到 Kafka 主题，然后其他应用程序可以订阅并处理这些事件，实现松耦合、可扩展和实时的事件驱动架构。
- 流式处理和实时分析：Kafka 可以与流处理框架（如 Apache Spark、Apache Flink）集成，支持实时的流式数据处理和复杂的实时分析。通过将数据流导入 Kafka 主题，应用程序可以实时处理和分析数据，实现实时的洞察和决策。

- 异步消息队列：Kafka 可以用作高吞吐量的异步消息队列，实现应用程序之间的解耦和异步通信。应用程序可以将消息发送到 Kafka 主题，并异步处理这些消息，提高系统的可伸缩性和弹性。
- 数据备份和容错性：Kafka 支持数据的分区和复制，可以在多个节点上复制和存储数据，以实现数据的冗余备份和容错性。

4.3 Go三方库

- `github.com/Shopify/sarama`：是一个 Go 语言编写的 Apache Kafka 客户端库，用于与 Kafka 集群进行交互。它提供了丰富的功能和易于使用的 API，使开发人员能够方便地使用 Go 语言来生产和消费 Kafka 消息。
- Github链接：`https://github.com/Shopify/sarama`

4.4 Kafka使用

1. 安装依赖包

```
go get github.com/Shopify/sarama
```

2. 初始化Client

- client

```
package main

import (
  "fmt"
  "github.com/Shopify/sarama"
)

func main() {
  // 创建配置
  config := sarama.NewConfig()
  config.Producer.Return.Successes = true
```

```
    // 初始化客户端
    client, err := sarama.NewClient([]string{"localhost:9092"}, config)
    if err != nil {
        fmt.Println("Failed to create client: ", err)
        return
    }
    defer client.Close()

    // 进行其他操作，如创建生产者或消费者

    fmt.Println("Client initialized successfully!")
}
```

- 生产端

```
producer, err := sarama.NewSyncProducerFromClient(client)
if err != nil {
    // 处理生产者创建错误
    return
}
defer producer.Close()
```

- 消费端

```
consumer, err := sarama.NewConsumerFromClient(client)
if err != nil {
    // 处理消费者创建错误
    return
}
defer consumer.Close()
```

3. 初始化参数详解

```
Version：指定使用的Kafka协议版本。
ClientID：指定客户端ID，用于在Kafka服务器端进行日志记录和跟踪。
Net：指定用于与Kafka服务器通信的网络类型。
Metadata.Refresh：控制是否从Kafka集群获取和更新元数据信息。
Producer.Return.Successes：在成功发送消息后返回成功的消息元数据。
Producer.RequiredAcks：指定生产者需要等待的确认数。
Consumer.Group.Rebalance.Strategy：指定消费者组的再平衡策略。
Consumer.Offsets.Initial：指定消费者从哪个偏移量开始消费。
Consumer.MaxWaitTime：设置消费者在等待消息时的最大等待时间。
Consumer.Fetch.Default：指定消费者默认一次从Kafka服务器获取的消息字节数。
Admin.Timeout：设置管理员操作的超时时间。
```

4. Topic 的增删查

```go
package main

import (
  "fmt"
  "github.com/Shopify/sarama"
)

func main() {
  // 创建配置
  config := sarama.NewConfig()
  config.Producer.Return.Successes = true

  // 初始化客户端
  client, err := sarama.NewClient([]string{"192.168.0.14:9092"},
config)
  if err != nil {
    fmt.Println("Failed to create client: ", err)
    return
  }

  fmt.Println("Client initialized successfully!")

  //创建Topic，首先生成admin client
```

```
admin, err := sarama.NewClusterAdminFromClient(client)
if err != nil {
  // 处理创建管理员错误
  return
}
defer admin.Close()
//设置分区以及副本
//topicDetail := &sarama.TopicDetail{
//  NumPartitions:     2,
//  ReplicationFactor: 1,
//}
//创建Topic
//err = admin.CreateTopic("my-topic", topicDetail, false)
//if err != nil {
//  fmt.Println("Failed to create topic: ", err)
//  return
//}

//查看所有Topic
topics, err := admin.ListTopics()
if err != nil {
  fmt.Println("Failed to list topic: ", err)
  return
}
fmt.Println(topics)

//删除Topic
//err = admin.DeleteTopic("my-topic")
//if err != nil {
//  fmt.Println("Failed to delete topic: ", err)
//  return
//}

//查看Topic详情
partitions, err := admin.DescribeTopics([]string{"my-topic"})
if err != nil {
  fmt.Println("Failed to describe topic: ", err)
  return
}
```

```
  for _, partition := range partitions {
    fmt.Println(partition)
  }

  //查找指定主题和分区的最早和最新偏移量
  earliestOffset, err := client.GetOffset("my-topic", 0,
sarama.OffsetOldest)
  if err != nil {
    fmt.Println("Failed to get earliest offset: ", err)
    return
  }

  latestOffset, err := client.GetOffset("my-topic", 0,
sarama.OffsetNewest)
  if err != nil {
    fmt.Println("Failed to get latest offset: ", err)
    return
  }
  fmt.Println(earliestOffset, latestOffset)

  //查找指定主题分区的消费位移
  consumer, _ := sarama.NewConsumerFromClient(client)

  defer consumer.Close()
  //获取分区的消费者
  partitionConsumer, _ := consumer.ConsumePartition("my-topic", 0,
sarama.OffsetNewest)

  defer partitionConsumer.Close()
  //获取消费者的最新消费位移
  partitionOffset := partitionConsumer.HighWaterMarkOffset()
  fmt.Printf("Partition 0 Offset: %d\n", partitionOffset)
}
```

5. 生产消息

```
package main

import (
  "fmt"
  "github.com/Shopify/sarama"
  "log"
  "time"
)

func main() {
  // 创建配置
  config := sarama.NewConfig()
  config.Producer.Return.Errors = true
  config.Producer.Return.Successes = true

  // 同步生产消息
  producer, err :=
sarama.NewSyncProducer([]string{"192.168.0.14:9092"}, config)
  if err != nil {
    log.Fatalln("Failed to create sync producer: ", err)
  }
  defer producer.Close()

  // 指定要发送的主题和消息内容
  topic := "my-topic"
  message := "Hello, Kafka!"

  // 创建消息对象
  msg := &sarama.ProducerMessage{
    Topic: topic,
    Value: sarama.StringEncoder(message),
  }

  // 发送消息
  partition, offset, err := producer.SendMessage(msg)
  if err != nil {
```

```
    log.Println("Failed to send message: ", err)
  } else {
    log.Printf("Message sent successfully! Partition=%d,
Offset=%d\n", partition, offset)
  }

  // 创建异步生产者
  producer, err :=
sarama.NewAsyncProducer([]string{"192.168.0.14:9092"}, config)
  if err != nil {
    log.Fatalln("Failed to create async producer: ", err)
  }
  defer producer.Close()

  // 指定要发送的主题和消息内容
  topic := "my-topic"
  message := "Hello, Kafka!"

  // 创建消息对象
  msg := &sarama.ProducerMessage{
    Topic: topic,
    Value: sarama.StringEncoder(message),
  }

  // 设置消息的键（Key）
  msg.Key = sarama.StringEncoder("my-key")

  // 设置消息的分区（Partition）
  msg.Partition = 0

  // 设置消息的时间戳（Timestamp）
  msg.Timestamp = time.Now()

  // 设置消息的头部（Headers）
  msg.Headers = []sarama.RecordHeader{
    {Key: []byte("header-key"), Value: []byte("header-value")},
  }

  // 发送消息
```

```
  producer.Input() <- msg

  // 处理发送结果
  select {
  case success := <-producer.Successes():
    fmt.Printf("Message sent successfully! Partition=%d,
Offset=%d\n", success.Partition, success.Offset)
  case err := <-producer.Errors():
    log.Println("Failed to send message: ", err.Err)
  case <-time.After(5 * time.Second):
    log.Println("Timeout reached. Failed to send message.")
  }
}
```

6. 消费消息

- 单分区消费者

```
package main

import (
  "fmt"
  "github.com/Shopify/sarama"
  "log"
  "os"
  "os/signal"
)

func main() {
  // 创建配置
  config := sarama.NewConfig()
  config.Producer.Return.Errors = true
  config.Producer.Return.Successes = true

  // 同步生产消息
  consumer, err := sarama.NewConsumer([]string{"192.168.0.14:9092"},
config)
```

```
  if err != nil {
    fmt.Println("Failed to create consumer: ", err)
    return
  }
  defer consumer.Close()

  // 订阅主题
  topic := "my-topic"
  partition := int32(0) // 消费的分区
  offset := int64(0)    // 消费的起始偏移量

  // 根据指定分区和偏移量创建分区消费者
  partitionConsumer, err := consumer.ConsumePartition(topic,
partition, offset)
  if err != nil {
    log.Fatalln("Failed to create partition consumer: ", err)
  }
  defer partitionConsumer.Close()

  // 处理收到的消息
  go func() {
    for message := range partitionConsumer.Messages() {
      fmt.Printf("Received message: Topic=%s, Partition=%d,
Offset=%d, Key=%s, Value=%s\n",
        message.Topic, message.Partition, message.Offset,
string(message.Key), string(message.Value))
    }
  }()

  // 处理消费错误
  go func() {
    for err := range partitionConsumer.Errors() {
      fmt.Println("Error:", err.Err)
    }
  }()

  // 等待程序终止信号
  sigchan := make(chan os.Signal, 1)
  signal.Notify(sigchan, os.Interrupt)
```

```
  <-sigchan
}
```

- 多分区消费者

```
package main

import (
  "fmt"
  "github.com/Shopify/sarama"
  "os"
  "os/signal"
  "sync"
  "syscall"
)

func main() {
  // 创建配置
  config := sarama.NewConfig()
  config.Producer.Return.Errors = true
  config.Producer.Return.Successes = true

  // 同步生产消息
  consumer, err := sarama.NewConsumer([]string{"192.168.0.14:9092"},
config)
  if err != nil {
    fmt.Println("Failed to create consumer: ", err)
    return
  }
  defer consumer.Close()

  // 订阅主题
  topic := "my-topic"

  // 消费多个分区
  partitions := []int32{0, 1, 2} // 消费的分区列表

  // 使用 WaitGroup 等待消费完成
```

```
  wg := &sync.WaitGroup{}
  wg.Add(len(partitions))

  // 创建分区消费者并消费消息
  for _, partition := range partitions {
    go func(p int32) {
      defer wg.Done()
      // 根据指定分区创建分区消费者
      partitionConsumer, err := consumer.ConsumePartition(topic, p,
int64(0))
      if err != nil {
        fmt.Printf("Failed to create partition consumer for partition
%d: %s\n", p, err.Error())
        return
      }
      defer partitionConsumer.Close()

      // 消费消息
      for message := range partitionConsumer.Messages() {
        // 处理收到的消息
        fmt.Printf("Message received: Topic=%s, Partition=%d,
Offset=%d, Key=%s, Value=%s\n",
          message.Topic, message.Partition, message.Offset,
string(message.Key), string(message.Value))
      }
    }(partition)
  }

  // 等待程序终止信号
  sigchan := make(chan os.Signal, 1)
  signal.Notify(sigchan, syscall.SIGINT, syscall.SIGTERM)
  <-sigchan

  // 等待消费完成
  wg.Wait()
}
```

- 消费组

```go
package main

import (
  "context"
  "fmt"
  "log"
  "os"
  "os/signal"

  "github.com/Shopify/sarama"
)

func main() {
  // 创建配置
  config := sarama.NewConfig()
  config.Consumer.Offsets.Initial = sarama.OffsetNewest

  // 创建消费者组
  consumerGroup, err :=
sarama.NewConsumerGroup([]string{"192.168.0.11:9092"}, "my-group",
config)
  if err != nil {
    log.Fatal("Failed to create consumer group: ", err)
  }
  defer consumerGroup.Close()

  // 指定要订阅的主题
  topics := []string{"my-topic"}

  // 创建一个消费者组处理器
  handler := ConsumerGroupHandler{}

  // 启动消费者组
  ctx := context.Background()
  go func() {
    for {
      err := consumerGroup.Consume(ctx, topics, handler)
      if err != nil {
        log.Fatal("Error from consumer group: ", err)
```

```
        }
    }
  }()

  // 等待程序终止信号
  sigchan := make(chan os.Signal, 1)
  signal.Notify(sigchan, os.Interrupt)
  <-sigchan
}

// 自定义消费者组处理器
type ConsumerGroupHandler struct{}

// 实现 sarama.ConsumerGroupHandler 接口的方法
// Setup 方法在消费者组启动时调用，用于进行一些初始化操作
func (h ConsumerGroupHandler) Setup(sarama.ConsumerGroupSession)
error {
  fmt.Println("Consumer group setup")
  return nil
}
// Cleanup 方法在消费者组结束时调用，用于进行清理操作
func (h ConsumerGroupHandler) Cleanup(sarama.ConsumerGroupSession)
error {
  fmt.Println("Consumer group cleanup")
  return nil
}

func (h ConsumerGroupHandler) ConsumeClaim(session
sarama.ConsumerGroupSession, claim sarama.ConsumerGroupClaim) error {
  for message := range claim.Messages() {
    fmt.Printf("Received message: Topic=%s, Partition=%d, Offset=%d,
Key=%s, Value=%s\n",message.Topic, message.Partition, message.Offset,
string(message.Key), string(message.Value))
    session.MarkMessage(message, "") // 提交消费位移
  }
  return nil
}
```