Transwarp Scope 9.3 使用手册

星环信息科技（上海）股份有限公司

版本号 T00113x-88-010，2023-06

目录

免责声明

本说明书依据现有信息制作,其内容如有更改,恕不另行通知。星环信息科技（上海）股份有限公司在编写该说明书的时候已尽最大努力保证期内容准确可靠,但星环信息科技（上海）股份有限公司不对本说明书中的遗漏、不准确或印刷错误导致的损失和损害承担责任。具体产品使用请以实际使用为准。

注释：Java® 是Oracle公司在美国和其他国家的商标或注册的商标。Intel® 和Xeon® 是英特尔公司在美国、中国和其他国家的商标或注册的商标。

手册版本信息

版本号：T00113x-88-010

发布日期：2023-06

1. Transwarp Scope 简介

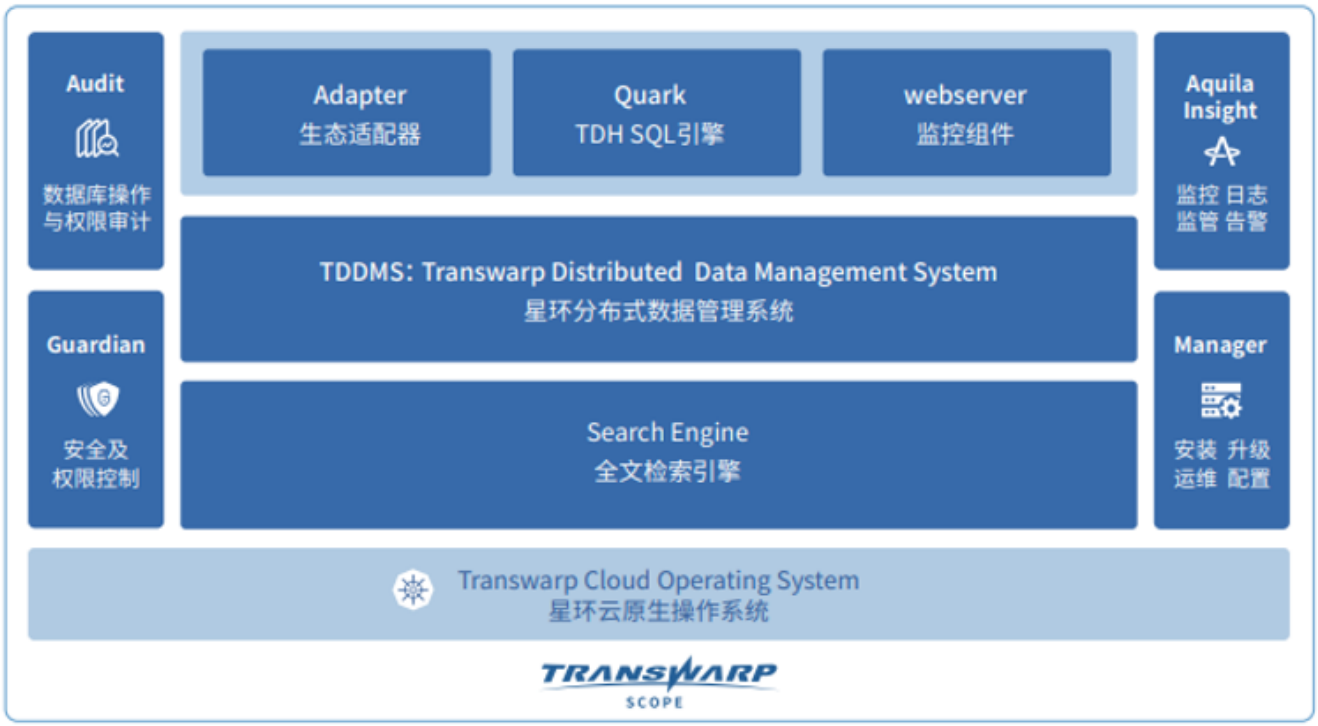
1.1. 产品定义

Transwarp Scope(下面简称Scope)是一个可扩展的分布式搜索和分析引擎。在Transwarp Data Hub中，Transwarp Scope被定义为：

1. 分布式文件存储（Distributed Document Store）；
2. 强大的搜索引擎。常见应用场景有海量数据的存储和搜索、日志分析等。

可以在满足传统Search组件的功能基础上提供更优异、更稳定的性能和更好的运维体验。

1.2. 产品架构



核心组件概述：

1. TDDMS：分布式数据管理系统，自主研发的分布式数据库管理系统，采用shared-nothing架构，通过多副本机制实现数据服务高可用，使用raft协议保证副本之间的数据一致性；TDDMS支持弹性扩缩容、自动故障恢复、权限控制、多租户与冷热数据分层存储等功能。
2. Search engine：自主研发的全文检索引擎，面向半结构化数据设计，支持Schema-less的数据模型，适用于从日志等半结构化数据中高效检索、挖掘关键信息。
3. Adapter：Scope生态适配器，用于承载和处理各类es请求，提供标准的es接口服务。
4. Quark：提供标准sql能力和slipstream接入能力，打通TDH内多类存储的数据流转。
5. Webserver：Scope内置的一个轻量级监控组件，用于监控集群/索引状态，同时提供rest风格的界面化操作平台。

2. Scope 基础

2.1. Transwarp Scope 安装与部署

2.1.1. 安装部署

这里给出整套的集群与组件的安装说明

1. 启动后台manager安装脚本install，在浏览器输入提供的url http://hostname:8179 进行安装
2. 接受基本用户许可协议后进行安装，分别进行给集群命名，ntp服务配置，机架配置，rpm库配置的操作

图 1. 接受许可协议

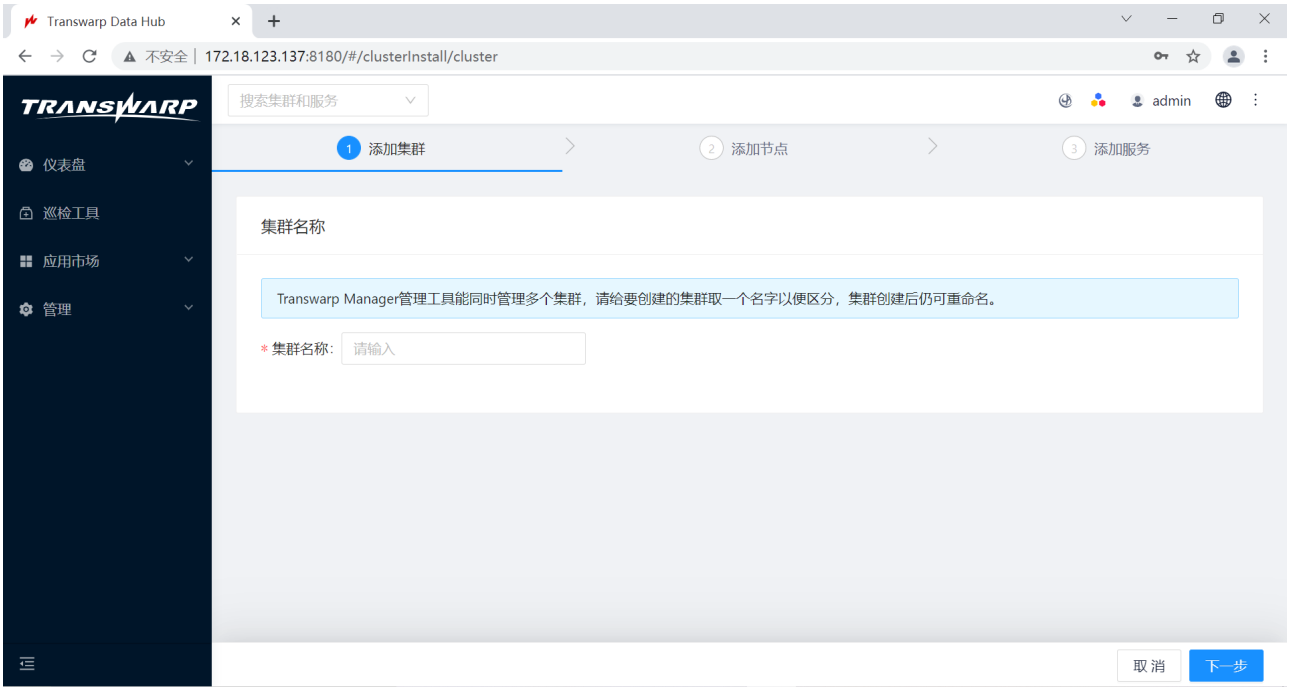


图 2. 集群命名

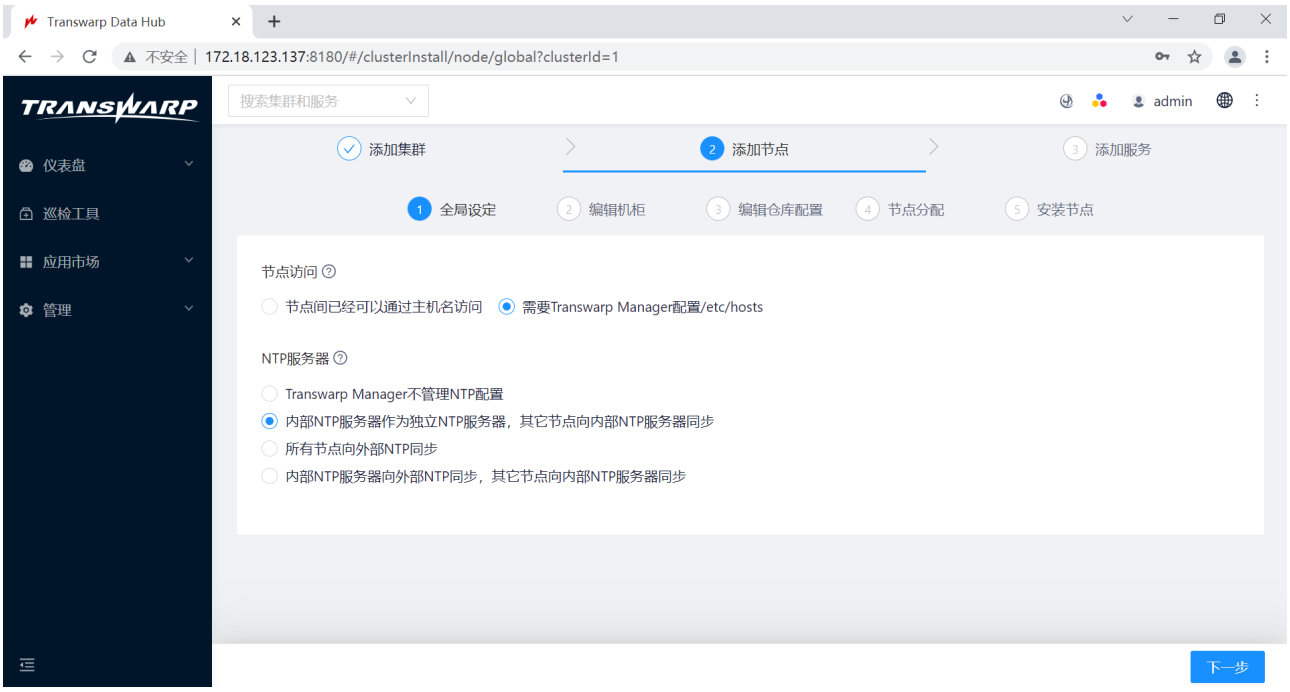


图 3. 集群host和ntp服务配置

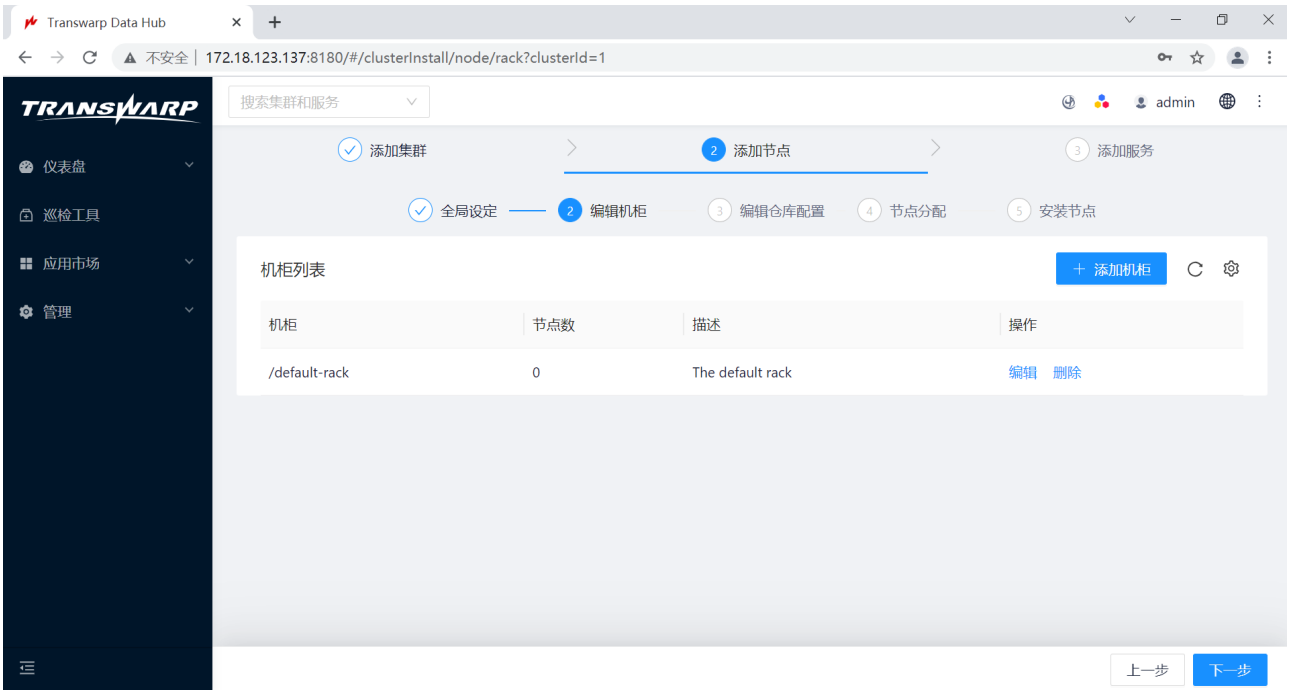


图 4. 机架配置

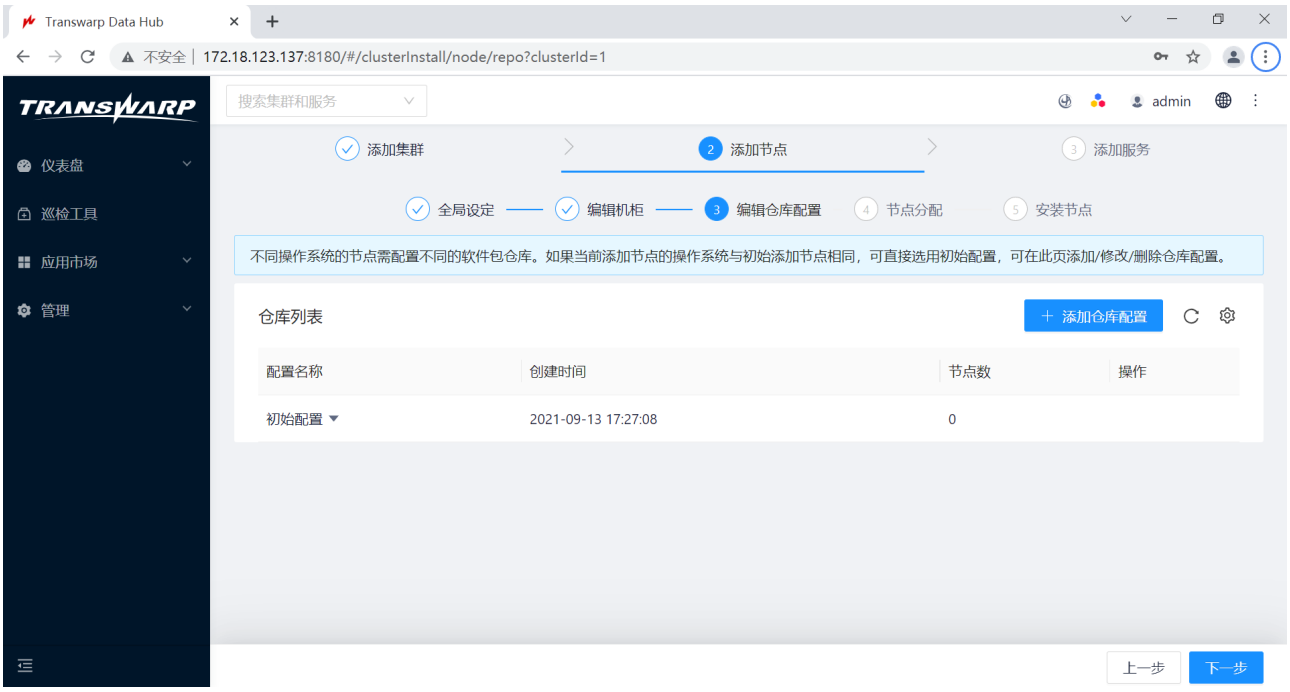


图 5. rpm库配置

3. 节点分配环节，通过添加节点进入开始选择纳入规划的节点，同时可以对不同节点指定机柜和不同的rpm库以满足异构os的需求

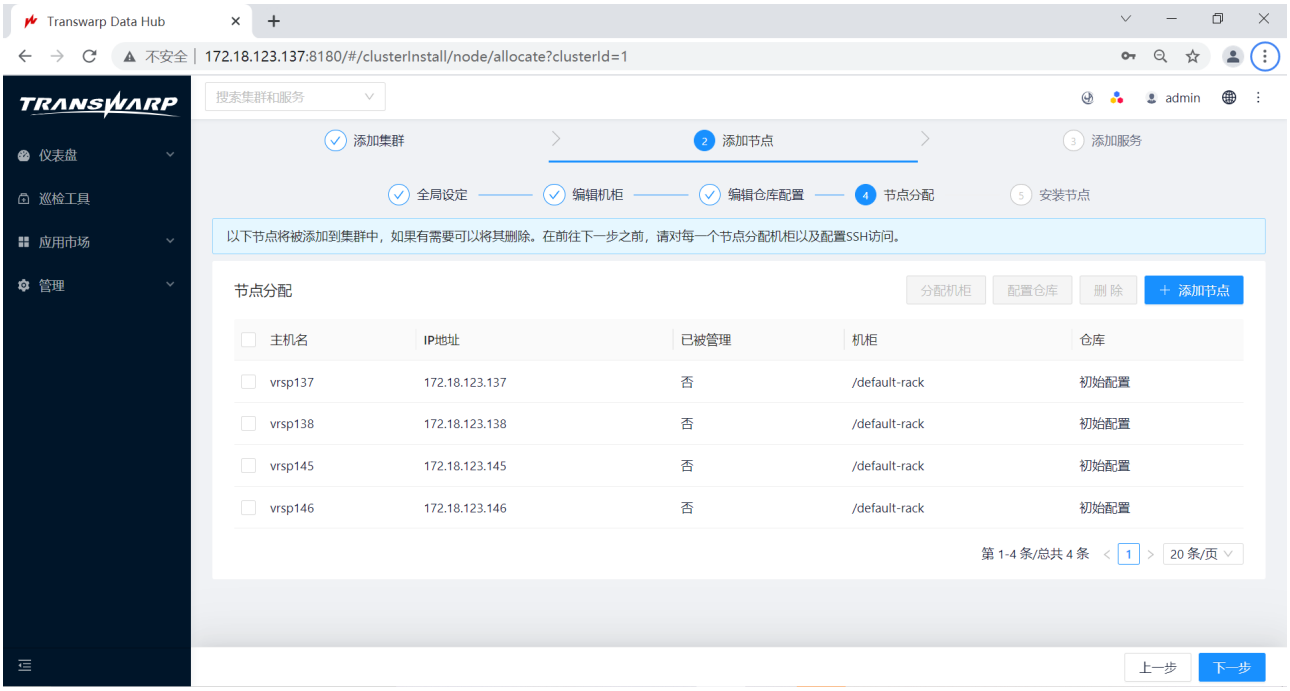


图 6. 节点分配

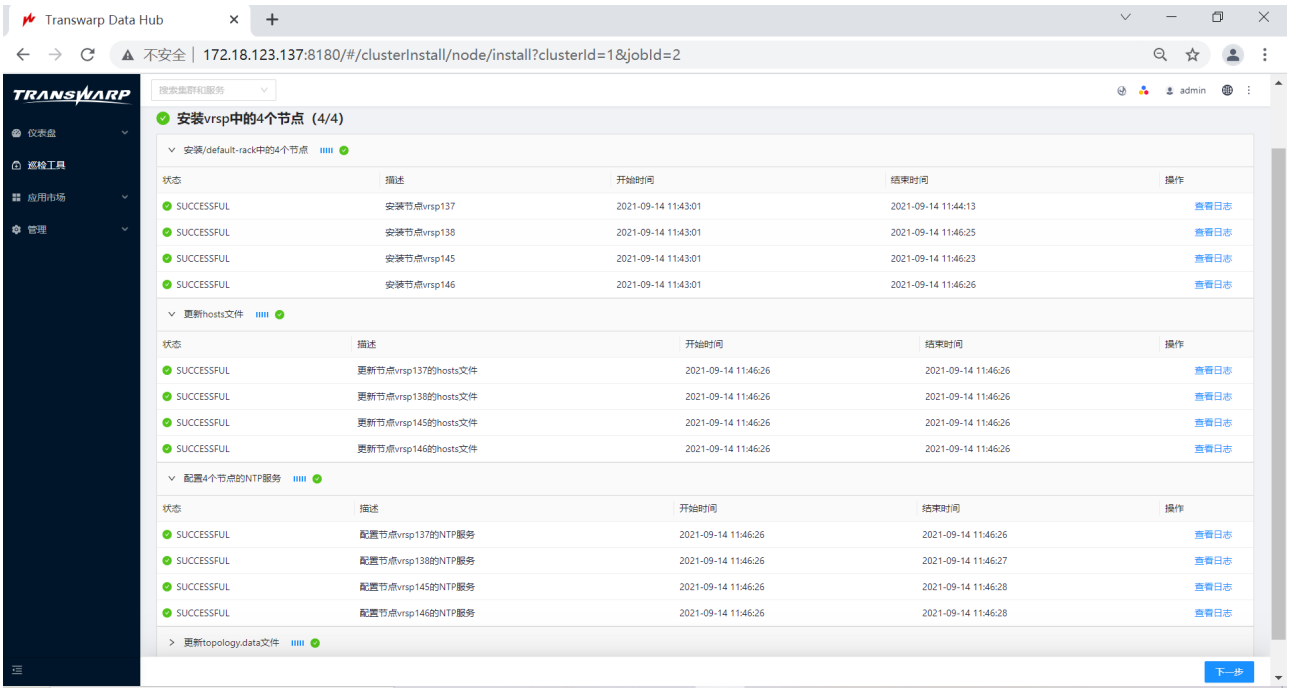


图 7. 节点安装

4. 先把Manager安装完成，在上传产品包

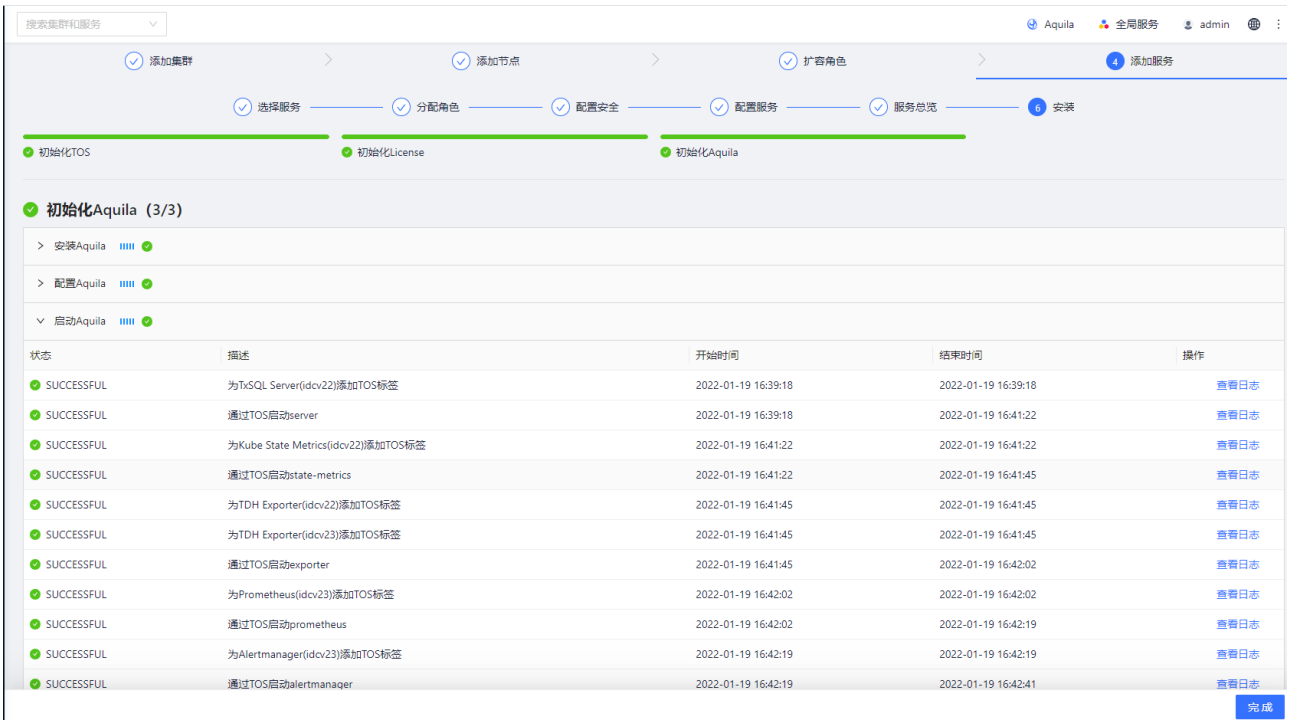


图 8. manager安装完成

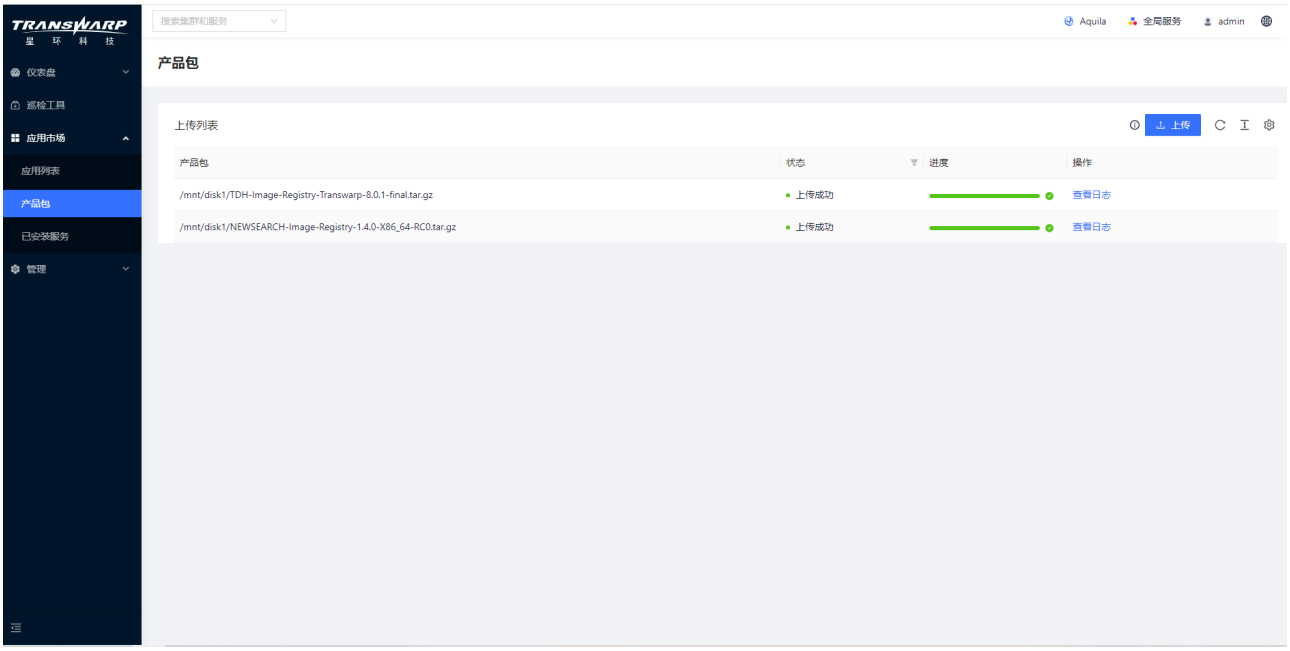


图 9. 安装包上传

5. 安装服务，manager首页通过添加服务选择Scope服务以及依赖的一些组件和对应服务版本

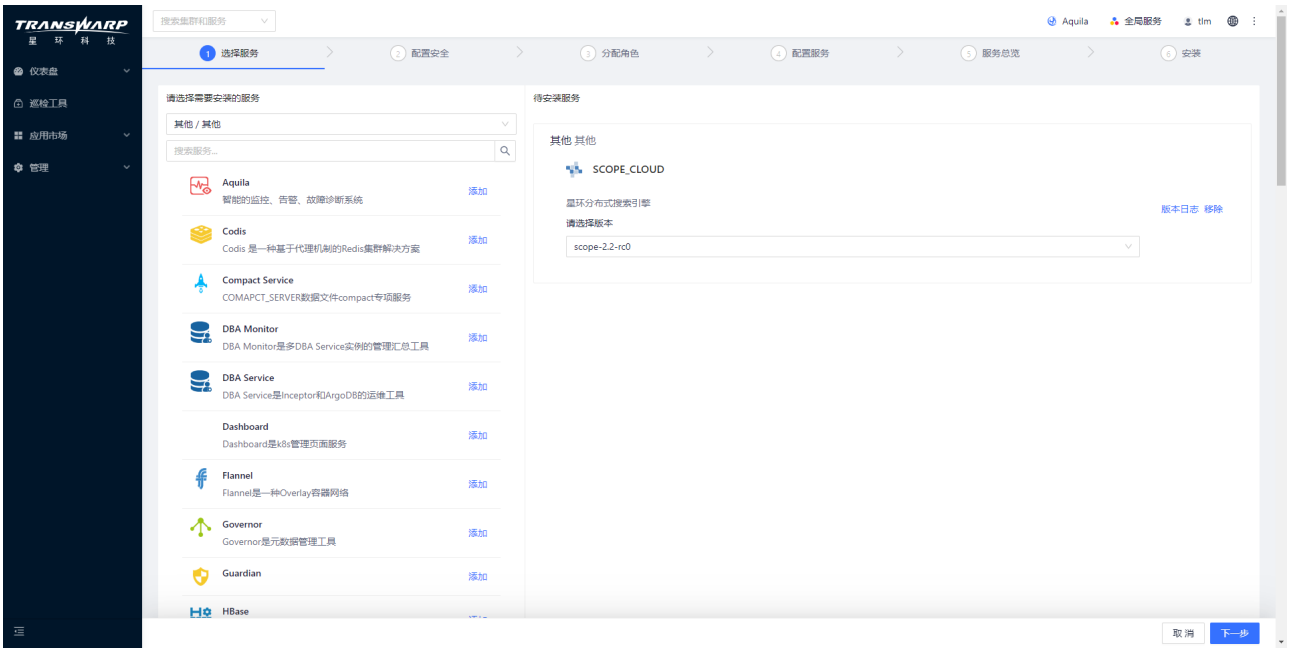


图 10. 选择安装服务和版本

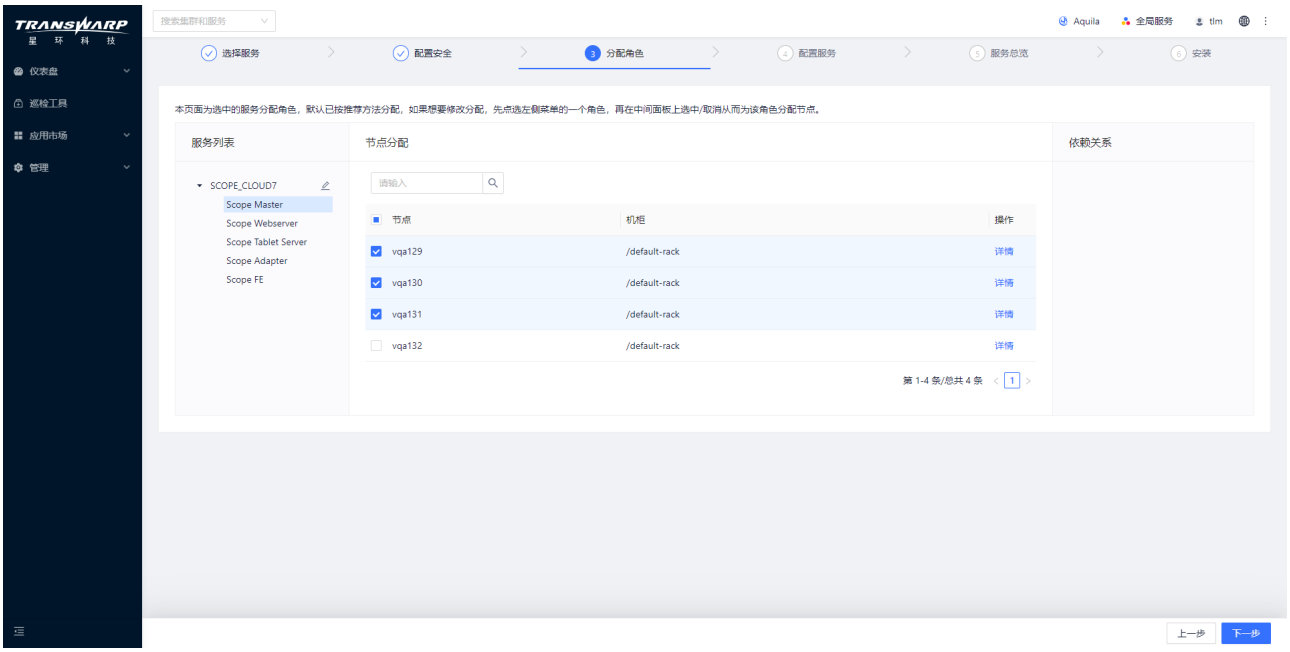


图 11. 根据提示分配组件的各类角色

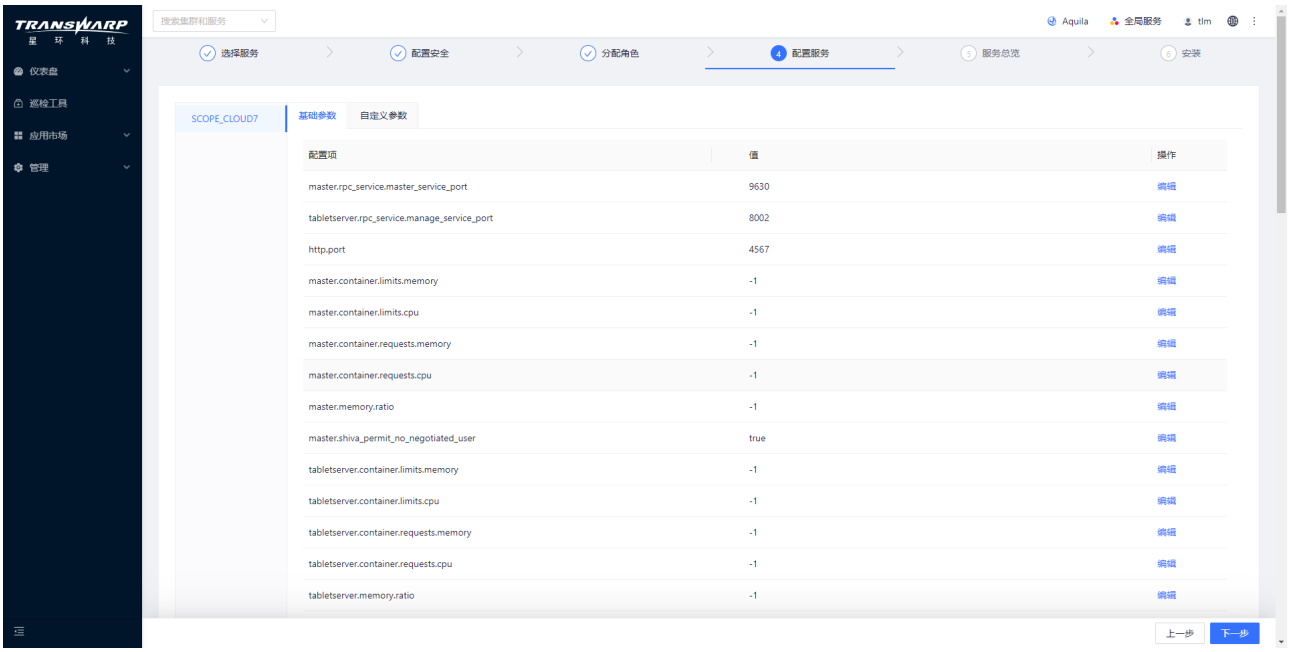


图 12. 预定义组件各类参数

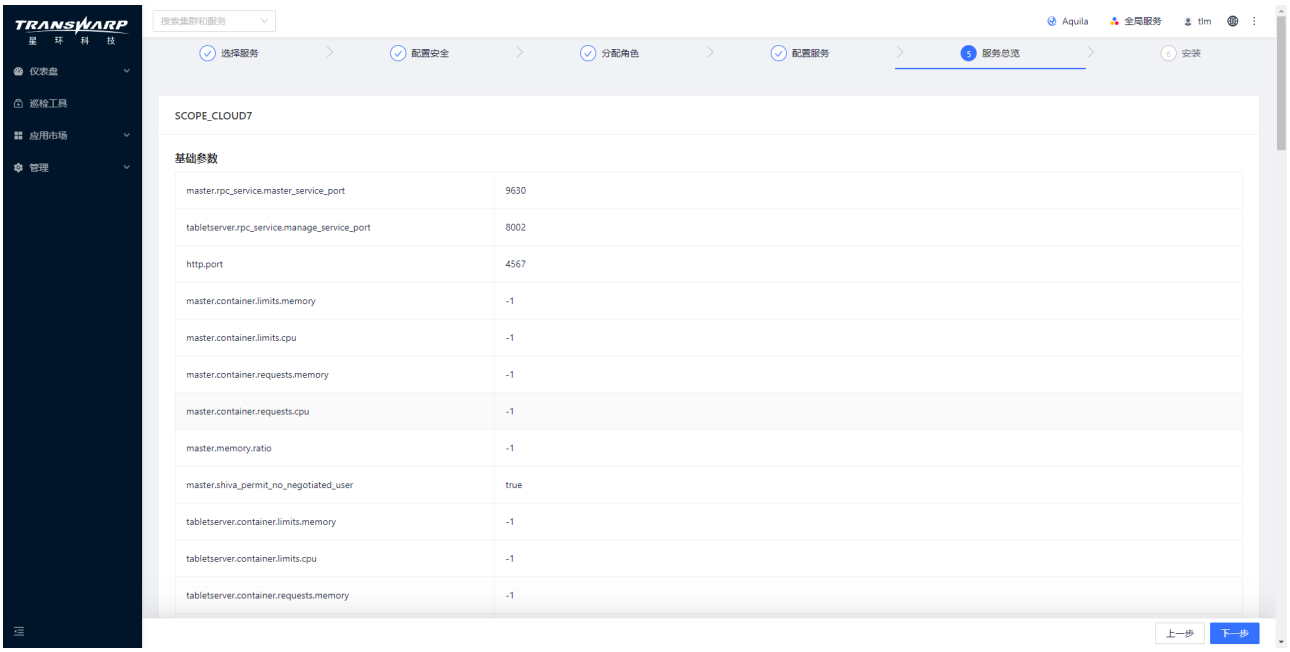


图 13. 服务总览检查

6. 以上步骤完成后，开始进行服务的安装，安装完成后就可以通过manager界面开始正式使用产品

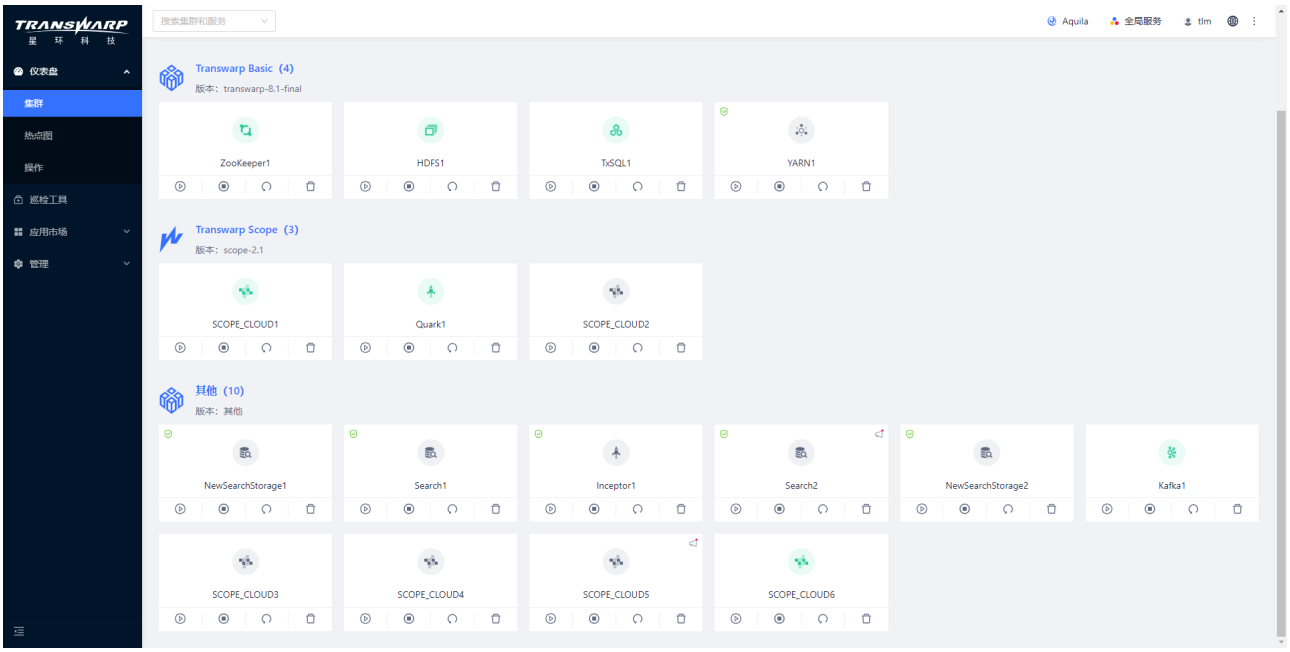


图 14. 服务安装完毕

2.1.2. 参数配置

2.1.2.1. storage 相关

1. master.rpc_service.master_service_port

   master对外提供rpc服务的端口，master使用从该端口开始的连续4个端口。

   前两个端口需要对客户端可见。

   一旦集群部署完毕，该配置不允许修改，否则会导致TDDMS服务异常。

2. tabletserver.rpc_service.manage_service_port

tabletserver对外提供rpc服务的端口，tabletserver使用从该端口开始的连续4个端口。

前两个端口需要对客户端可见。

一旦集群部署完毕，该配置不允许修改，否则会导致TDDMS服务异常。

3. master.master.data_path

   master节点的数据存储目录。

   部署时会默认选择本机第一个挂载点作为数据目录。

   如需修改该配置，需要首先停止该master节点，并将原目录下内容完整拷贝到新的目录下。

   不推荐对该配置进行修改。

4. tabletserver.store.datadirs

   tabletserver节点的数据存储目录。

   部署时会默认选择本机所有容量超过200GB的挂载点作为数据目录。

   如本机不存在容量超过200GB的挂载点，需要部署人员人工指定数据目录。

   如需修改该配置，需要首先停止该tabletserver节点，并将原目录下内容完整拷贝到新的目录下。

   由于一般数据存储量比较大，不推荐对该配置进行修改。

5. store.capacity

   tabletserver节点的单盘存储quota，单位为GB。

   部署时会默认取本机所有容量超过200GB的挂载点最小容量的一半。

   如本机不存在容量超过200GB的挂载点，需要部署人员人工指定该配置。

6. master.log.log_v_level

   master verbose信息输出级别调整

   从0开始配置，值越大表明输出的日志越多

7. master.log.log_min_log_level

   master 日志输出级别调整

   从0-3一共4个级别，值越大表明日志囊括信息越少，分别对应INFO>WARNING>ERROR>FATAL 4类信息

8. topology.topology.rack

   部署tabletserver时定义的每个节点的机架分布

   必须在添加节点和第一次安装集群时设置，后续更改不会生效

9. topology.topology.datacenter

   部署tabletserver时若存在跨中心需求时，可通过该参数定义和区分不同数据中心

   必须在添加节点和第一次安装集群时设置，后续更改不会生效

10. tabletserver.search.refresh.interval

    在数据写入动作触发后，使得数据写入os cache后并可被搜索的操作时间

11. master.scheduler.keep_trash_time_s

    数据删除后在存储中保留的实际按周期，单位为s

12. master.shiva_permit_no_negotiated_user

    安全相关参数

    是否允许匿名用户访问scope，默认为true，即scope对所有用户操作开放权限，以Public用户进行数据操作

13. tabletserver.jvm.use_container_support

    arm版本相关参数

    使得jvm可以正确识别pod内core的个数，确保服务正常使用

    x86版本无需设置该参数，**arm版本请设置为false**

14. tabletserver.jvm.Xms

    tabletserver服务堆内存的初始大小

15. tabletserver.jvm.Xmx

    tabletserver服务堆内存的最大大小

16. master.extra_flags

    用于master角色中添加各类自定义参数

    参数格式为 --key=value,多个参数则换行分隔

17. tabletserver.extra_flags

    用于tabletserver添加各类自定义参数

    参数格式为 --key=value,多个参数则换行分隔

2.1.2.2. esadapter 相关

1. adapter.dc

   配合datacenter参数使用，adapter所连接dc的地址

2. adapter.jvm.use_container_support

   arm版本相关参数

   使得jvm可以正确识别pod内core的个数，确保服务正常使用

   x86版本无需设置该参数，**arm版本请设置为false**

3. adapter.jvm.Xms

   adapter服务堆内存的初始大小

4. adapter.jvm.Xmx

   adapter服务堆内存的最大大小

   一般设置为Xmx=Xms

5. adapter.port

   restful请求端口，默认为9200

6. adapter.extra_flags

   用于adapter添加各类自定义参数

   参数格式为 key: value（注意空格），多个参数则换行分隔

2.1.2.3. webserver 相关

1. webserver.memory

   webserver 服务启动后占用内存的大小

2. http.port

   webserver提供服务的端口。

端口需要对使用者可见。

默认值4567

3. restful.jvm.use_container_support

arm版本相关参数

使得jvm可以正确识别pod内core的个数，确保服务正常使用

x86版本无需设置该参数，**arm版本请设置为false**

2.2. Transwarp Scope 数据对象与架构

2.2.1. Scope数据对象

Scope包括四种对象：Database, Table, Row, Column。

2.2.1.1. Database （数据库）

database(老版本称为namespace)用于对不同业务逻辑以及不同用户管理表(Table)的集合。用户可以在不同database下创建多张同名的表(Table)，互不影响，默认情况下试用default库即可。

2.2.1.2. Table（表）

Scope以Table（表）为单位组织数据。同一个Table下的数据通常有相似的特征。

2.2.1.3. Row（行）

Row（行）是Scope中最基础的数据单元。例如，员工信息Table中某员工的信息可以作为一个Row保存；商品信息Table中某商品的信息也可以作为一个Row保存。Row以结构化数据的格式存储，例如：

```
{
 "name": "Zhang San",
 "age": 26,
 "on_board_date": "2015-10-31",
 "school": "Nanjing University"
}
```

2.2.1.4. Column（列）

Row中的信息存储在Column（列）中。下例的Row中，**name**，**age**，**on_board_date** 和 **school** 为列，**Zhang San**，**26**，**2015-10-31** 和 **Nanjing University** 分别为这些列的值。

```
{
 "name": "Zhang San",
 "age": 26,
 "on_board_date": "2015-10-31",
 "school": "Nanjing University"
}
```

2.2.2. 分片（Tablet）与副本(Replica)

一个Scope Table下可能会有大量的数据，超过硬件的存储能力。Scope中，您可以将Table分成多个 **分片（Tablet）**，将Table分片有两个作用：

- 横向扩展一个Table的容量；
- 提高计算的并行度从而提升性能。

Scope中的分片分为两种：主分片（Leader Tablet）和副本分片（Replica Tablet，或简称Replica）。

- 主分片（Primary Tablet）：

  Table中的每个Row都属于唯一的Leader Tablet，所以Leader Tablet 数量决定了Table的容量。

  Leader Tablet 数量需要在Table创建时指定，创建后不可修改。

- 副本分片（Replica Tablet）：

  Replica Tablet则是Leader Tablet 的拷贝，不仅用于提供数据冗余，也提供数据读取服务（比如检索请求、Row获取请求等）。

  一个Table中每个Leader Tablet的Replica数需要在Table创建时指定，创建后不可修改。

默认设置下，一个Scope Table有10个Leader Tablet，每个Leader Tablet有1个Replica，所以默认情况下一个Scope Table总共有20个分片。

分片的分布和计算的并行化完全由Scope来管理。

Scope会保证一个节点上相同数据只有一份，也就是说Leader Tablet和它自己的Replica永远不会存在一个节点上。

如果create table指的 Replica 数大于或等于Scope集群中节点数量，这个Table中将会有分片无法分配到节点上。

2.2.3. 数据映射关系

由于Scope同样支持开源Es的部分接口和使用方式，那么在使用Scope SQL交互时，不可避免的我们会去比对elasticsearch index和Transwarp Scope中table的映射关系，及Scope SQL在其中发挥的作用，才能正确高效地使用Scope SQL，实现高效的全文检索或者交互分析。

表 1. Scope table与Elasticsearch中Index的映射关系

ELasticsearch Index	Scope SQL
Index(索引)	Table(表)
Document（记录）	Row(行)
Field（字段）	Column(列)
shard	tablet
replica	replica

注意，scope的副本个数和ES的副本个数关系为scope replica number=es replica number+1[相同副本个数时]

2.3. Transwarp Scope 组件角色

Scope有如下几类角色：

1. master

   又称为shiva-master。

负责TDDMS的元信息存储。

master是一个raft group，由多个master节点组成。

部署中推荐配置三或五个master节点。

2. tabletserver

   又称为shiva-tabletserver

   负责TDDMS的数据存储。

   部署中推荐至少存在五个tabletserver节点。

3. webserver

   负责基于TDDMS衍生的各类产品的集群监控，比如Scope。

   部署中至少需要一个webserver节点。

4. esadapter

   负责elasticsearch client与Scope的交互

   提供兼容开源elastisearch 的API接口,目前兼容版本为es 6.7.2

   支持http协议和java

   部署中如果使用到了开源elasticsearch，则推荐部署，节点个数根据需求即可

2.4. localmode &clustermode

Scope具备两种执行模式： Local 和Cluster。编译语句时系统会根据相当前模式生成相应的执行计划

1. Local Mode

   Local模式适用于一些低延时、高并发、参与计算数据量少的场景。Local模式本身要求数据源检索有较快的速度，所以Local模式的优势是可以较快的返回结果，满足低延时需求。Job只在InceptorServer所在机器上另起线程执行，不需要将task分发出去，不需要申请资源注册Executor。

   使用场景：1.高并发业务
   2.各类精确查询或者模糊查询类的检索业务

2. Cluster Mode

   Cluster模式是默认执行模式，适用于批处理业务。它通过InceptorServer将SQL转换成Job分发到各个节点的Executor，并由Task具体处理，各个Task将结果汇总写入结果目录，由Fetch Task显示结果，如果Task执行的是CTAS（CREATE TABLE … AS…）则将结果Move到对应表的数据存储目录下。

   使用场景：涉及分析、统计、计算等业务场景

目前引擎默认为Cluster模式，在适当场景下需要手工设置为Local模式。Local模式支持不启动executor读Scope数据源。

开关转换的控制方式如下所示：

```
SET ngmr.exec.mode = local/cluster;
```

3. Transwarp Scope SQL

3.1. Scope SQL 功能特性

Scope SQL作为Scope的上层sql部分，通过sql引擎层兼容了基础SQL语法，提供了基于sql的读写支持。Scope SQL目前主要功能如下：

- 数据迁移

  使用sql的方式可以比较高效的进行数据迁移，不论是Scope的API还是我们适配的开源API，使用上都有较高的入门门槛，而使用Scope SQL，用户只要有编写SQL的经验，就可以简单的操作和迁移数据到Scope。

- 数据预览

  数据迁移后，提供sql的方式进行简单的数据检索和查询，进行数据的校验和查看。

3.2. Scope SQL 约定符号

为提高本手册Scope SQL语法介绍部分的可读性，本手册将使用一些约定的符号，便于用于正确理解和使用SQL语句。特在此列出：

表 2. Scope SQL语法介绍的约定符号

符号	说明	示例语句	备注
<…>	尖括号，意必选项必须填的变量	create table <tableName> …	tableName 必填，见于建表语法
[…]	方括号，意为可选项，可填可不填的变量	…[WITH SHARD NUMBER <m>]…	WITH SHARD NUMBER <m> 可选，见于建表语法

3.3. Scope SQL 相关配置

下述配置主要针对部分老版本如Scope2.1，Scope2.0或部分其他组件的Quark，**Scope2.3之后的版本** 可以通过界面配置服务依赖自动对下述参数配置无需额外添加

由于Scope和quark为两个相对独立的两个产品与服务，所以两者需要一些配置参数关联起来，这里会介绍下quark与scope的相关配置：

- hive.metastore.pre.event.listeners

  manager界面配置，添加自定义参数即可。确保quark metastore与scope的通讯链接。

  value为org.apache.hadoop.hive.ql.lockmgr.metastoreListener.Shiva2TxnPreEventListener。

- ngmr.shiva2.mastergroup

  manager界面配置，添加自定义参数即可。参数用于quark与scope的master服务的交互。 value为scope集群的master group地址。

- ngmr.shiva2.use.user.login

manager界面配置，添加自定义参数即可。控制是否以用户名密码的方式访问scope2集群。 default value为 true。

- ngmr.shiva2.user.username

  manager界面配置，添加自定义参数即可。在ngmr.shiva2.use.user.login为true的情况下，指定访问scope服务的用户名。

- ngmr.shiva2.user.password

  manager界面配置，添加自定义参数即可。在ngmr.shiva2.use.user.login为true的情况下，配合ngmr.shiva2.user.username参数提供的用户所对应的用户民密码。

- EXTRA_EXECUTOR_OPTS/EXTRA_DRIVER_OPTS/EXTRA_METASTORE_OPTS*

  quark 的附加参数，用于添加自定义参数，这里额外说明 **arm系统下需要添加的特殊参数：**-XX:-UseContainerSupport 该参数用于确保arm系统下jvm可以正确获取processors。

如果集群中Scope期望不使用安全认证即可访问Scope的话，尤其是操作的表既要执行sql也需要执行restful api的情况下，推荐ngmr.shiva2.use.user.login参数设置为false

3.4. DDL

DDL目前包括创建（CREATE）/删除（DROP）/清空（TRUNCATE），下面将一一介绍。

3.4.1. 表创建： CREATE

Scope 建表支持内表/外表两种表类型，这里会详细介绍Scope table的创建流程

3.4.1.1. CREATE TABLE

具体建表的基础语法如下：

建内表语法

```
CREATE TABLE <tableName> ( ①
    <column> <data_type>, ②
    <column> <data_type>,
    ...
)
STORED AS scope  ③
[WITH SHARD NUMBER <m>]   ④
[REPLICATION <n>]   ⑤
[TBLPROPERTIES('key1'='value1',....)]  ⑥
```

① Scope SQL目前不支持中文表名。

② Scope SQL目前不支持中文列名。

③ 简写方式，指定表的存储格式为 scope表 。

④ 可选项。指定Scope中相映射的tablet个数， **默认值为5，建议每个tablet的数据量不超过25G** 。建表后不可改。

⑤ 可选项。指定每个分片的副本数， **推荐使用默认值3（如果是社区开发版请使用1）** 。

⑥ 对表额外定义参数时通过tblproperties指定

建外表语法

```
CREATE EXTERNAL TABLE <tablename>(
<column> <data_type>, ①
<column> <data_type>, ②
[_uid string], ③
...
)
STORED AS scope
[WITH SHARD NUMBER <m>]
[REPLICATION <n>]
TBLPROPERTIES('scope.table.name'='foreign_table_name');
```

① 如果映射的数据类型为decimal，这里的precison和scale需要和外表一致

② 如果映射的数据类型为text，分词信息的填写非必须

③ 该列用于映射scope的唯一标识列，即key，非必需添加。不添加时外表支持insert，select，但不支持upsert和bulkload

为帮助您更好地理解scope table，下面我们将通过一个具体的例子创建一张表来介绍。

例 1. 建Scope SQL表 scope_1

```
create table scope_1(
v1 boolean,
v2 int,
v3 tinyint,
v4 smallint,
v5 bigint,
v6 float,
v7 double,
v8 date,
v9 decimal(10, 4),
v10 string,
v11 string has analyzer 'standard'
)
STORED AS scope with shard number 5 replication 3
TBLPROPERTIES('scope.key.column'='v10','scope.partition.type'='range','scope.partition.column.r
ange'='v8','scope.partition.column.value.v8'='2021-10-01,2022-10-01');
```

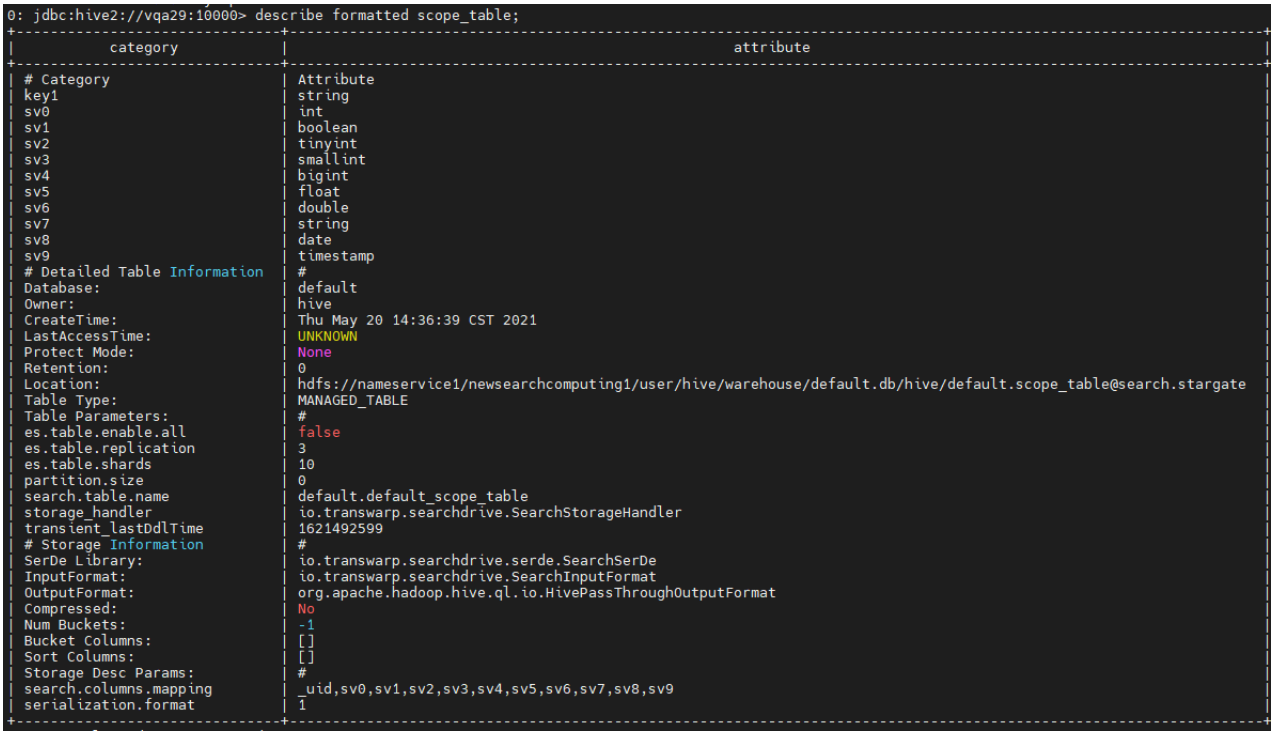
创建了一张名为 scope_1 的表，包含所有的数据类型，存储格式为 scope ，tablet个数为5，副本数（replication）为3，即都采用默认值。有关shard和replication的内容请参见Transwarp Scope 数据对象与架构章节的介绍。

建表后可通过 DESCRIBE FORMATTED 查看表 **scope_table** 的元数据信息。

例 2. 查看 scope_table 元数据信息

```
describe formatted scope_1;
```

结果如下图：



```
0: jdbc:hive2://vqa29:10000> describe formatted scope_table;
+-------------------------------+-----------------------------------------------------------------------------------------------------------------+
|           category            |                                                    attribute                                                    |
+-------------------------------+-----------------------------------------------------------------------------------------------------------------+
| # Category                    | Attribute                                                                                                       |
| key1                          | string                                                                                                          |
| sv0                           | int                                                                                                             |
| sv1                           | boolean                                                                                                         |
| sv2                           | tinyint                                                                                                         |
| sv3                           | smallint                                                                                                        |
| sv4                           | bigint                                                                                                          |
| sv5                           | float                                                                                                           |
| sv6                           | double                                                                                                          |
| sv7                           | string                                                                                                          |
| sv8                           | date                                                                                                            |
| sv9                           | timestamp                                                                                                       |
| # Detailed Table Information  | #                                                                                                               |
| Database:                     | default                                                                                                         |
| Owner:                        | hive                                                                                                            |
| CreateTime:                   | Thu May 20 14:36:39 CST 2021                                                                                    |
| LastAccessTime:               | UNKNOWN                                                                                                         |
| Protect Mode:                 | None                                                                                                            |
| Retention:                    | 0                                                                                                               |
| Location:                     | hdfs://nameservice1/newsearchcomputing1/user/hive/warehouse/default.db/hive/default.scope_table@search.stargate |
| Table Type:                   | MANAGED_TABLE                                                                                                   |
| Table Parameters:             | #                                                                                                               |
| es.table.enable.all           | false                                                                                                           |
| es.table.replication          | 3                                                                                                               |
| es.table.shards               | 10                                                                                                              |
| partition.size                | 0                                                                                                               |
| search.table.name             | default.default_scope_table                                                                                     |
| storage_handler               | io.transwarp.searchdrive.SearchStorageHandler                                                                   |
| transient_lastDdlTime         | 1621492599                                                                                                      |
| # Storage Information         | #                                                                                                               |
| SerDe Library:                | io.transwarp.searchdrive.serde.SearchSerDe                                                                      |
| InputFormat:                  | io.transwarp.searchdrive.SearchInputFormat                                                                      |
| OutputFormat:                 | org.apache.hadoop.hive.ql.io.HivePassThroughOutputFormat                                                        |
| Compressed:                   | No                                                                                                              |
| Num Buckets:                  | -1                                                                                                              |
| Bucket Columns:               | []                                                                                                              |
| Sort Columns:                 | []                                                                                                              |
| Storage Desc Params:          | #                                                                                                               |
| search.columns.mapping        | _uid,sv0,sv1,sv2,sv3,sv4,sv5,sv6,sv7,sv8,sv9                                                                    |
| serialization.format          | 1                                                                                                               |
+-------------------------------+-----------------------------------------------------------------------------------------------------------------+
```

由上图可见， scope_1 表创建成功，且对比建表语法可知一些默认的表元数据信息：

- number_of_shards: 5，tablet数目，可以类比为es中shard个数。
- number_of_replicas: 3，tablet的副本数。

另外，由前文介绍可知，表在创建时，会在Scope中新建一个相映射的table: **<databasename>.<tablename>**，该table可通过webserver管理界面： http://<webserver_ip>:<webserver_port>/page/namespace/default 查看，如下图：

例 3. scope table在底层存储中中相映射的table

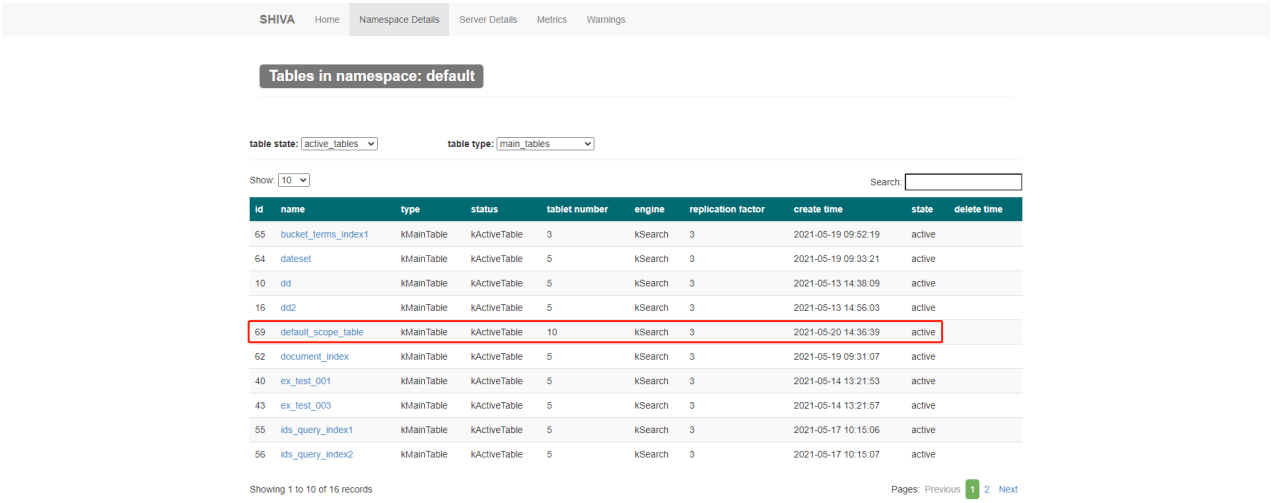


图 15. scope table 映射关系图1

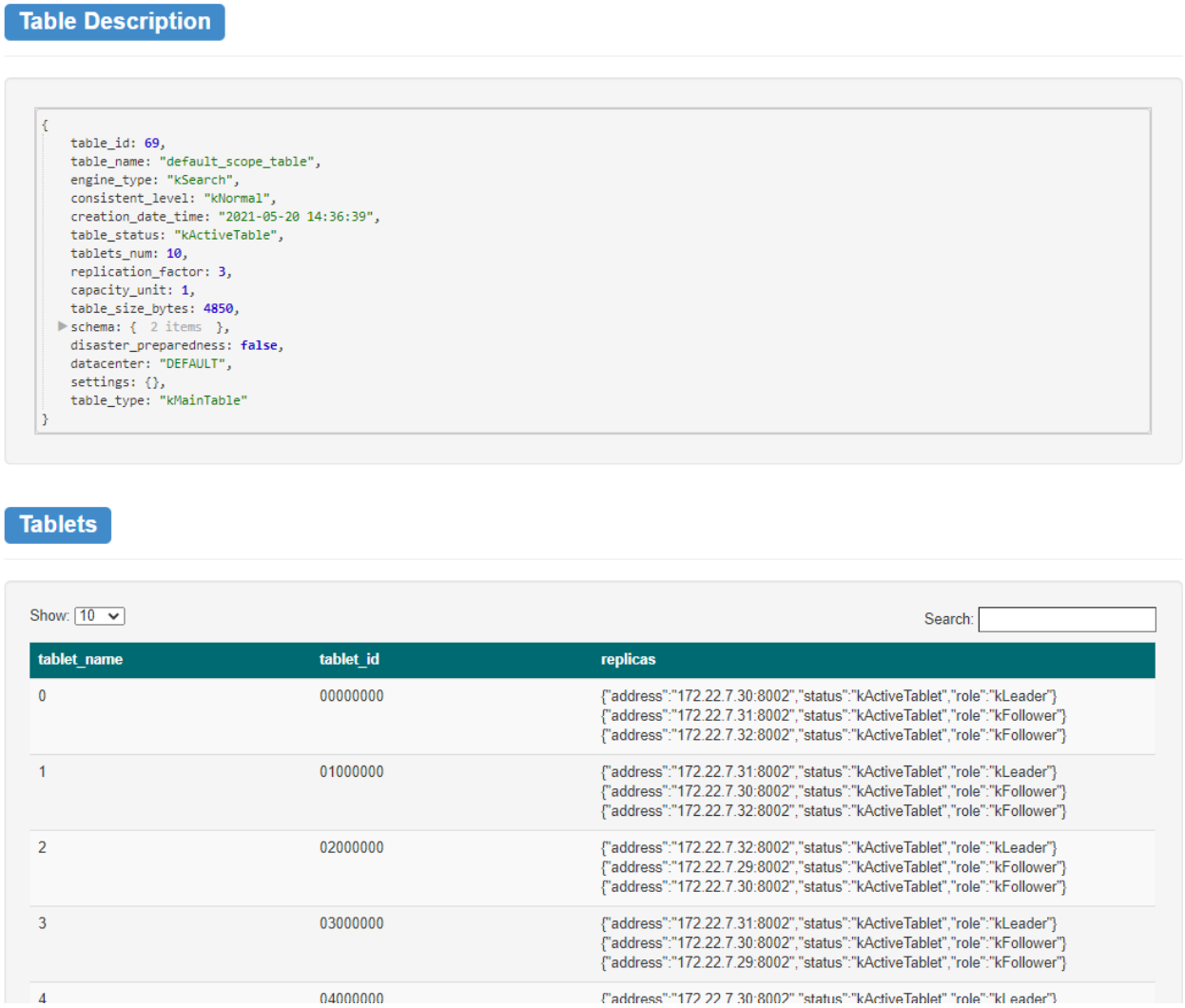


图 16. scope table 映射关系图2

如上图， **scope_1** 的shard数为10，副本数为3。

3.4.1.2. Scope SQL表支持的数据类型

示例中新建的table中包含了Scope SQL所支持的所有数据类型，以及类比elasticsearch的Index中对应字段的实际类型如下表：

Scope SQL支持的数据类型

Scope SQL表支持的数据类型与sql语法中的数据类型对应关系：

Scope SQL支持的数据类型	scope中的实际类型
string	string
int	integer
boolean	boolean
tinyint	byte
smallint	short
bigint	long
float	float
double	double
date	date
decimal	decimal
stirng has analyzer	text

string 和 **string has analyzer** 用于区分字段是否需要指定分词器分词。

3.4.1.3. Scope SQL表支持的表参数

示例中新建的table中包含了Scope SQL支持的tblproperties，以下介绍对应tblproperties的含义

Scope SQL支持的tblproperties

tblproperties	含义
'scope.key.column'	定义scope中的key（需要是string类型，非string类型会报错）对应哪一列，若不指定，默认所有列中的第一列对应scope的key
'scope.partition.type'	若建分区表，需要指定此参数；表示分区类型，可选值：range或者single_value，目前只支持单个值
'scope.partition.column.分区类型'	若建分区表，需要指定此参数；表示分区列，可以填写多个值（目前boolean，float，double，以及precision大于18的decimal类型不可作为分区列的数据类型）；其中，分区类型为'scope.partition.type'指定的分区类型
'scope.partition.column.value.分区列名'	若建分区表，需要指定此参数；表示分区列的取值，可以填写多个值；其中，分区列名为'scope.partition.column.分区类型'中指定的分区列名中的一个
'scope.bulkload.reduce.time'	提升reduce阶段的task个数，可以提升入库速度，生成的task个数为 total task=(shardnumber)**(reduce.time)**(partiton number)

3.4.2. 清空表： TRUNCATE

Scope SQL支持表级别的清空命令，可以一次性清空表中的所有数据，但不删除表的元数据信息。

清空Scope SQL表的语法

```
TRUNCATE TABLE <tableName>;
```

例 4. 清空Scope SQL表

预览数据并进行清除

```
select * from scope_1;
truncate table scope_1;
```

外表不支持 truncate。

3.4.3. 删除表： DROP

Scope SQL删除表的语法如下：

删除Scope SQL表的语法

```
DROP TABLE <tableName>;
```

例 5. 删除表

```
DROP TABLE scope_1;
```

删除操作，既删除Inceptor中的映射关系，也删除Scope中相映射的table。此时，表的数据和元数据都会被删除。 因此，打开webserver的管理页面，不会看到相映射的table。

外表删除不会删除底层映射的table，只会删除映射关系。

3.5. DML

Scope SQL中的DML（Data Manipulation Language）目前只包含插入（INSERT）。

3.5.1. 插入数据：INSERT

Scope SQL支持向scope表中单条插入数据或者批量插入查询结果。

3.5.1.1. 单条插入：INSERT

单条插入数据语法一次只可插入一条记录。具体语法如下：

单条插入的语法

```
INSERT INTO TABLE <tableName> [(<column1>, <column2>, ...)] VALUES (<value1>, <value2>, ...);
```

例 6. 单条插入

单条插入数据：

```
INSERT INTO scope_test(sv0,sv1,sv2,sv3,sv4,sv5,sv6,sv7,sv8,sv9)
VALUES (2,true,3,4,5,6.0,7.33,'s7ad','2010-06-30 08:54:42',1477881007946);
或者
INSERT INTO scope_test select 2,true,3,4,5,6.0,7.33,'s7ad','2010-06-30 08:54:42',1477881007946
from system.dual;
```

3.5.1.2. 批量插入：INSERT

批量插入数据语法允许基于条件插入多条或者一个批次的数据。具体语法如下：

批量插入的语法

```
INSERT INTO TABLE <tableName> [(<column1>, <column2>, ...)] select [*|(<column1>, <column2>, ...)]
from <sourcetable> where CONDITION=...;
```

例 7. 批量插入

批量数据：

```
INSERT INTO scope_test(sv0,sv1,sv2,sv3,sv4,sv5,sv6,sv7,sv8,sv9) select
sv0,sv1,sv2,sv3,sv4,sv5,sv6,sv7,sv8,sv9 from scope_test2;
或者
INSERT INTO scope_test select * from scope_test2;
```

3.6. DQL

Scope SQL中的DQL（Data Query Language）语法和常用的DQL别无二致，满足标准的sql语法规范。

3.6.1. 基础条件查询

首先，提供基础样例数据如下：

```
0: jdbc:hive2://vqa29:10000/default> select * from scope_table;
+-------+------+--------+------+------+------+--------------------+-------+-------+----------------------+--------------------------+
| key1  | sv0  |  sv1   | sv2  | sv3  | sv4  |        sv5         |  sv6  |  sv7  |         sv8          |           sv9            |
+-------+------+--------+------+------+------+--------------------+-------+-------+----------------------+--------------------------+
| k4    | 5    | true   | 3    | 4    | 5    | 6.300000190734863  | 7.11  | ewwd  | 2010-06-30 18:52:37  | 2016-10-31 00:13:27.946  |
| k3    | 4    | true   | 3    | 4    | 5    | 6.199999809265137  | 7.22  | ngfh  | 2010-06-30 06:08:17  | 2010-06-30 01:36:47.946  |
| k2    | 3    | true   | 3    | 4    | 5    | 6.099999904632568  | 7.33  | dsfx  | 2010-06-30 10:07:48  | 2013-08-30 11:06:47.946  |
| k1    | 2    | false  | 3    | 4    | 5    | 6.0                | 7.33  | s7ad  | 2010-06-30 08:54:42  | 2016-10-31 10:30:07.946  |
+-------+------+--------+------+------+------+--------------------+-------+-------+----------------------+--------------------------+
```

查询语法

```
SELECT <column1>[, <column2>, ...] FROM TABLENAME WHERE SQLCONDITION = ...;
```

例 8. 基础条件查询

```
--scope_table基于sv7字段的精确查询
select key1,sv0,sv6 from scope_table where sv7 = "ewwd";

+-------+------+-------+
| key1  | sv0  |  sv6  |
+-------+------+-------+
| k4    | 5    | 7.11  |
+-------+------+-------+
```

3.6.2. 排序查询

查询语法

```
SELECT <column1>[, <column2>, ...] FROM TABLENAME [WHERE SQLCONDITION = ...] order by <column1>
[desc|asc][,<column2>...];
```

例 9. 排序查询

```
--在一定条件下对scope_table查询特定字段并基于sv0字段进行降序排列
select key1,sv0,sv1,sv6 from scope_table where sv1 = true order by sv0 desc;

+-------+------+-------+-------+
| key1  | sv0  |  sv1  |  sv6  |
+-------+------+-------+-------+
| k4    | 5    | true  | 7.11  |
| k3    | 4    | true  | 7.22  |
| k2    | 3    | true  | 7.33  |
| k1    | 2    | true  | 7.33  |
+-------+------+-------+-------+
```

3.6.3. 聚合查询

查询语法

```
SELECT <aggregation function>(column) FROM TABLENAME [WHERE SQLCONDITION = ...] ;
```

例 10. 聚合查询

```
--scope_table在满足sv1字段为true的条件下取得sv0字段最大值,sv6字段最小值以及sv5字段平均值
select max(sv0) max,min(sv6) min,avg(sv5) avg from scope_table where sv1 = true ;

+------+-------+--------------------+
| max  |  min  |        avg         |
+------+-------+--------------------+
| 5    | 7.11  | 6.149999976158142  |
+------+-------+--------------------+
```

常用的聚合条件有以下几类：

3.6.4. 分组查询

查询语法

```
SELECT <column1>[,<column2>,...] FROM TABLENAME [WHERE SQLCONDITION = ...] group by
<column1>[,<column2>,...];
```

例 11. 分组查询

```
--scope_table在满足sv0字段不为5的条件下分组统计出sv6的分布
select sv6,count(sv6) cnt from scope_table where sv0!=5 group by sv6;
+-------+------+
|  sv6  | cnt  |
+-------+------+
| 7.33  | 2    |
| 7.22  | 1    |
+-------+------+
```

3.6.5. 分页查询

查询语法

```
SELECT <column1>[,<column2>,...] FROM TABLENAME [WHERE SQLCONDITION = ...] limit m[,n];
```

例 12. 分页查询

```
--scope_table在满足sv0字段不为5的条件下分组统计出sv6的分布
select key1,sv0,sv6 from scope_table order by key1 asc limit 2,2;
+-------+------+-------+
| key1  | sv0  |  sv6  |
+-------+------+-------+
| k3    | 4    | 7.22  |
| k4    | 5    | 7.11  |
+-------+------+-------+
```

由于Scope的使用场景中一般是针对结果集量不会很大的情况下进行额统计分析，所以对于体量特别大的又有检索需求的表，不论是分页还是聚合分析等查询，都推荐使用条件控制最终的结果集，尤其是分页的 **limit m,n** 其本质是查询 **m+n** 条进行计算而不是查询 **n** 条

3.7. Bulkload

Scope SQL中支持bulkload方式进行批量入库，该方式可以以更高效更快速的方式将数据导入到目标表中。

3.7.1. BULKLOAD INSERT

3.7.1.1. 构建key相关函数： GenerateID

使用bulkload入库前需要构建对应的函数，通过该函数定义插入表的唯一列（Scope有类似于主键key的唯一ID列）。

在当前版本中，已经提供了内置函数scope_id(column)，该函数具备同等效果。

定义生成id函数的创建语法

```
create temporary function <function_name> as "io.transwarp.scope.udf.ScopeGenerateIdUDF";
```

例 13. 构建函数

```
create temporary function id as "io.transwarp.scope.udf.ScopeGenerateIdUDF";
```

创建临时函数id()用于生成scope的id列

3.7.1.2. 批量插入：BULKLOAD INSERT

Bulkload语法类似hyperbase bulkload，需要通过hint和上文提到的函数两者共同进行触发执行。

批量插入的语法

```
 <tableName> [(<column1>, <column2>, ...)] select [*|(<column1>, <column2>, ...)] from <sourcetable>
where CONDITION=...;

INSERT INTO TABLE [(col1,col2,ke3,col4,...)]
SELECT /*+USE_BULKLOAD*/ ①
col1,col2,
idfunction(colm) col3, ②
col4,... ③
from <tablename>
distribute by col3,...; ④
```

① Bulkload hint

② 生成唯一标识的函数所构造的列，该列需和原表的string列对齐，同时必须标注 **别名**

③ bulkload语法可以选择部分列导入目标表中，也支持选择全部列

④ distribute确保数据分布，第一个字段为id列，如果是分区表，后续添加的字段为所有分区列字段

bulkload中，scope的key必须通过key的生成函数进行生成，使用客户自己的key需注意列的唯一性

例 14. 批量插入

批量数据：

```
insert into scope_table select /*+USE_BULKLOAD*/ v1,v2,scope_id(v1) v3,v4 from orc_table
distribute by v3,v4;
或者
insert into scope_table(v1,v2,v3) select /*+USE_BULKLOAD*/ v1,v2,scope_id(v1) v3 from orc_table
distribute by v3;
```

3.8. 分区表

Scope支持分区表的使用。通过分区表可以比较有效的解决一下几类问题：

1. 单表数据量过大带来的tablet过大等问题
2. 降低故障风险，分区表的每个分区都是相对独立的存在

3.8.1. 分区表创建： CREATE

Scope 建表支持内表/外表两种表类型，这里会详细介绍Scope table的创建流程

具体建表的基础语法如下：

建内表语法

```
CREATE TABLE <tableName> (
    <column> <data_type>,
    <column> <data_type>,
    ...
)
STORED AS scope
[WITH SHARD NUMBER <m>]
[REPLICATION <n>]
TBLPROPERTIES('scope.key.column'='v10', ①
'scope.partition.type'='<range|single_value>', ②
'scope.partition.column.<partition_type>'='<col_name>',③
'scope.partition.column.value.<col_name>'='<value1>','<value2>',....)④
;
```

① 定义scope中的key（需要是string类型，非string类型会报错）对应哪一列，若不指定，默认所有列中的第一列对应scope的key

② 表示分区类型，可选值：range或者single_value，表示范围分区和单值分区

③ 表示建表中指定的分区列，可以填写多个值（目前boolean，float，double，以及precision大于18的decimal类型不可作为分区列的数据类型）;其中，<partion_type>类型为’ scope.partition.type’指定的分区类型

④ 表示分区列的取值，可以填写多个值；其中，分区列名为’ scope.partition.column.<partition>’中指定的分区列名中的一个

建外表语法

```
CREATE EXTERNAL TABLE <tableName> (
    _uid string,
    <column> <data_type>,
    <column> <data_type>
    ...
)
STORED AS scope
TBLPROPERTIES('scope.table.name'='<tablename>') ①
;
```

① 在Scope中创建的index/table的表名

为帮助您更好地理解scope 分区表，下面我们将通过两个具体的例子来进行介绍。

例 15. 构建分区内表

```
create table scope_partition_table(
v1 boolean,
v2 int,
v3 tinyint,
v4 smallint,
v5 bigint,
v6 float,
v7 double,
v8 date,
v9 decimal(10, 4),
v10 string,
v11 string has analyzer 'standard'
)
STORED AS scope with shard number 5 replication 3
TBLPROPERTIES('scope.key.column'='v10','scope.partition.type'='range','scope.partition.column.r
ange'='v8','scope.partition.column.value.v8'='2021-10-01,2022-10-01');
```

创建分区表scope_partition_table，v10作为该表的主键，分区表用v8列作为分区列，并设置为范围分区，同时创建两个分区，（-无穷，2021-10-01],(2021-10-01,2022-10-01]。

例 16. 构建分区外表

```
在scope中创建一个index/table
curl -X PUT "localhost:9200/partition-table?pretty" -H 'Content-Type: application/json' -d'
{
   "settings":{
      "index.number_of_replicas":3,
      "index.number_of_shards":5,
      "scope.partition.type":"range",
      "scope.partition.column.range":"happyDate",
      "scope.partition.column.value.happyDate":"2021-10-01,2022-10-01"
   },
  "mappings": {
    "properties": {
      "happyDate": {
        "type": "date",
        "store":"false"
      }
    }
  }
}
'



通过quark构建外表
create external table partition(
_uid string,
happyDate date)
STORED AS scope with shard number 5 replication 3
TBLPROPERTIES('scope.table.name'='partition-table');
```

范围分区表的每个分区是左开右闭的

3.8.2. 分区添加与删除

Scope通过alter tblproperties语法增删分区，具体的语法与使用如下：

增加分区语法

```
ALTER TABLE 表名 SET TBLPROPERTIES('scope.add.partition.column.value.<partiton_col>'='<value>');①
```

① <partition_col>为对应分区列，value为对应分区，可填多个

例 17. 增加分区

```
ALTER TABLE scope_partition_table SET
TBLPROPERTIES('scope.add.partition.column.value.v8'='2023-10-10,2024-10-10');
```

为scope_partition_tablet在原有基础上添加两个分区，(2022-10-01,2023-10-01]，(2023-10-01,2024-10-01]

删除分区语法

```
ALTER TABLE 表名 SET TBLPROPERTIES('scope.drop.partition.column.value.<partiton_col>'='<value>');①
```

① <partition_col>为对应分区列，value为对应分区，可填多个

例 18. 删除分区

```
ALTER TABLE scope_partition_table SET
TBLPROPERTIES('scope.drop.partition.column.value.v8'='2023-10-10,2024-10-10');
```

为scope_partition_tablet在原有基础上删除两个分区，(2022-10-01,2023-10-01]，(2023-10-01,2024-10-01]

分区删除除却删除原分区的所有数据外会使得分区进行合并，可根据现有分区列定义的边界值确定当前的各个分区范围；同时，范围分区中不支持添加的分区小于当前已经存在的分区边界值

3.8.3. 分区表删除

删除分区表语法

```
DROP TABLE <table_name>①
```

① 删除分区表会使得其名下所有分区删除

例 19. 删除分区表

```
DROP TABLE scope_partition_table;
```

3.9. Slipstream on Scope

Scope支持用Slipstream实现准实时的流式入库，一方面解决了kafka这类消息队列的数据的处理和入库，同时也能，降低数据流转的时间周期,做到高吞吐，低延时的数据接入。

由于对于scope和slipstream的使用大家通过两者手册已经有了初步的了解，这里通过一个完整的流程来展示了slipstream入库scope的具体用法

3.9.1. slipstream入库scope完整流程

Scope建表

```
//创建非分区Scope表
create table ss_t1(
    account string,
    btype string,
    SId string,
    amount smallint,
    price decimal(15,5),
    desc string,
    ttime date,
    TId string
)stored as scope
TBLPROPERTIES('scope.key.column'='tid');

//创建分区Scope表
create table ss_t2(
    account string,
    btype string,
    SId string,
    amount smallint,
    price decimal(15,5),
    desc string,
    ttime date,
    TId string
)stored as scope
TBLPROPERTIES('scope.key.column'='tid',
'scope.partition.type'='range',
'scope.partition.column.range'='ttime',
'scope.partition.column.value.ttime'='2020-11-10,2020-11-15,2020-11-20,2020-11-25');
```

创建流

```
//创建一个从尾部开始消费的流
create stream stream_scheckpoint
(TId string,account string,btype string,SId string,amount smallint,price decimal(15,5),desc
string,ttime date)
ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
tblproperties(
    "topic"="topic_t2s1",
    "transwarp.consumer.auto.offset.reset"="latest",
    "kafka.zookeeper"="idc16:2181,idc17:2181,idc18:2181",
    "kafka.broker.list"="idc16:9092");

//创建多topic的流
create stream stream_scheckpoint
(TId string,account string,btype string,SId string,amount smallint,price decimal(15,5),desc
string,ttime date)
ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
tblproperties(
    "topic"="topic_t2s.*", ①
    "kafka.zookeeper"="idc16:2181,idc17:2181,idc18:2181",
    "kafka.broker.list"="idc16:9092");
```

① topic可以通过正则的方式匹配多个关联到一个流中

创建streamjob

```
//非分区表streamjob
CREATE STREAMJOB job_t1_2
as ("INSERT INTO ss_t1 SELECT /*+USE_BULKLOAD*/ account,btype,SId,amount,price,desc,ttime,TId FROM
stream_scheckpoint distribute by TId") ①
JOBPROPERTIES(
"morphling.job.enable.checkpoint"="true", ②
"morphling.job.checkpoint.interval"="15000"  ③
);


//分区表streamjob
CREATE STREAMJOB job_t2_2
as ("INSERT INTO ss_t2 SELECT /*+USE_BULKLOAD*/ account ,btype ,SId ,amount ,price ,desc ,ttime,TId
FROM stream_scheckpoint distribute by TId,ttime") ④
JOBPROPERTIES(
"morphling.job.enable.checkpoint"="true",
"morphling.job.checkpoint.interval"="15000"
);
```

① slipstream入库目前均通过bulkload方式进行入库

② slipstream checkpoint功能

③ checkpoint时间间隔，即写入间隔，不推荐时间调整的非常小

④ 非分区表distribute需要接scope.key.column,而分区表在此基础上还需要加上所有的分区列

流的运行

```
//任务启动
start streamjob job_t1_2;
start streamjob job_t2_2;

//任务查看
list streamjobs;

//任务停止
stop streamjob job_t1_2;
stop streamjob job_t2_2;
```

3.9.2. 脏数据校验

目前通过流导入数据时，支持对脏数据的校对和检验。支持以下几类不合规范的校验：

1. 列个数不匹配（多列/少列）
2. 字段类型不匹配
3. 日期不符合规范，目前时间类型校验仅支持yyyy-mm-dd hh:MM:ss格式

使用脏数据校验功能需要在stream和streamjob中添加特定参数，以下通过例子表明参数的用法

stream相关参数

```
create stream stream_s2s_dirty1
(key string,cboolean boolean,ctinyint tinyint,csmallint smallint,cint int,cbigint bigint,cfloat
float,cdouble double,cdate date,ctimestamp date,cdecimal decimal(10,2))
ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
tblproperties(
    "topic"="topic_s2s_tdirty",
    "kafka.zookeeper"="idc16:2181,idc17:2181,idc18:2181",
    "kafka.broker.list"="idc16:9092",
    'stream.dirtydata.dump.tohdfs.timeindex'='9', ①
    'stream.dirtydata.checkers'='slipstream.morphling.sideoutput.TableScanColumnCheckerForAll');②
```

① stream.dirtydata.dump.tohdfs.timeindex=<int>:确定进行时间校验列是哪一列

② stream.dirtydata.checkers:固定用法，调用的校验类

streamjob相关参数

```
create streamjob streamjob_s2s_dirty12_1 as
("insert into datatype_t2_1  select /*+USE_BULKLOAD*/
cdate,cboolean,ctinyint,csmallint,key,scope_id(cboolean)
skey,cint,cbigint,cfloat,cdouble,ctimestamp,cdecimal from stream_s2s_dirty1 distribute by
skey,cdate")
JOBPROPERTIES(
"morphling.job.enable.checkpoint"="true",
"morphling.job.checkpoint.interval"="5000",
"stream.allow.dirtydata"="true", ①
"stream.dirtydata.location"="hdfs://nameservice1/slipstream4/wrongdata/");②
```

① stream.allow.dirtydata:是否允许脏数据写入

② stream.dirtydata.location:脏数据写入位置,目前支持写入到hdfs中

3.10. Scope SQL检索语义

从大篇幅文本数据中查询短语或单词时，标准SQL只能通过 like %word% 来查询，查询速度较慢，且检索语义十分单一。对此，Scope SQL在分词的基础上开发了语义检索的功能，其实现过程分为两步：

- 被查询的文本分词

  第一步是将被查询的文本根据指定分词器分词，生成倒排索引，使文本数据标准化，做到文本分词后的每个单词都可查询。这一步可以通过SQL语句在建内表时或者建外表时映射的字段本身已经指定了分词器来实现。

- 查询条件分词

  第二步是查询时，将查询条件根据与倒排索引相同的分词器分词，以同样的标准进行规范，再与倒排索引相匹配，返回符合条件的记录 。这一步会在后文的检索语句的sql中进行展示。

3.10.1. 检索语义的优势

相较于传统的模糊查询，Esdrive SQL的检索语义不仅提升了查询性能，还定义了多种模糊查询的语法，其优势总结如下：

- 查询性能更优

  传统标准SQL查询 like %word% 的算法复杂度是 O(n) ，但是分词查询利用了Scope本身的检索能力，复杂度只有 O(log(n)) ，其查询性能优于前者。

- 检索语义相对会丰富

  传统标准SQL的检索查询语义比较单一，而通过Scope的检索语义可以比较简化的表达出一部分以前仅能通过rest请求表达的检索请求。

3.10.2. 建表指定分词器的语法

Scope支持直接用SQL语句在建表时对列指定分词器，更为简洁易懂。在使用SQL语句对列指定分词器时，有以下几点注意事项：

- 对列指定分词器的语法只可用于建 **内表** 。
- 只可用于内表中 STRING 类型的列。

```
<column> HAS  ANALYZER  <ANALYZER_NAME>
```

<ANALYZER_NAME> ： 用于指定具体的分词器，如iksmart，pinyin,standard等 。

例 20. 分词器建表

```
create table scope_1(
v1 boolean,
v2 int,
v3 tinyint,
v4 smallint,
v5 bigint,
v6 float,
v7 double,
v8 date,
v9 decimal(10, 4),
v10 string,
v11 string has analyzer 'standard'
)
STORED AS scope with shard number 5 replication 3
TBLPROPERTIES('scope.key.column'='v10','scope.partition.type'='range','scope.partition.column.r
ange'='v8','scope.partition.column.value.v8'='2021-10-01,2022-10-01');
```

创建了一张名为 scope_1 的表，其中v11字段用标准分词器standard进行分词

3.10.3. CONTAINS 语法

3.10.3.1. CONTAINS 基础使用方法

建表并对列指定分词器实现被查询文本的标准化后，即可通过 CONTAINS 语法对查询条件进行分词，实现分词检索，具体语法如下：

CONTAINS 函数使用语法

```
CONTAINS(
   [schema.]column, ①
   '<text_query>'   ②
)
```

① **[schema.]column** ： 分词字段。

② **text_query** ： 检索表达式，会作为一个整体被传入 。

例 21. contains语法用例

```
SQL表达：
select * from search_zh_near_ik where contains(content, '南京市长');
select * from search_zh_near_ik where contains(content, '南京, 市长');


SCOPE语义分别为：
GET /_search
{
    "query": {
        "query_string" : {
            "fields" : ["content"],
            "query" : "南京市长"
        }
    }
}

GET /_search
{
    "query": {
        "query_string" : {
            "fields" : ["content"],
            "query" : "南京，市长"
        }
    }
}
```

3.10.3.2. NEAR 操作符

NEAR 操作符是在 CONTAINS 检索语法的基础上，限制所查询单词的间隔，提高查询结果的相关性。

该操作符具体使用规则如下：

near操作符使用规则

```
CONTAINS(<column>, 'NEAR((token1, token2[,token3,...]), slop[, in_order])');
```

总共包含3个参数：

- token1 ： 表示进行查询匹配的词，可以有多个。注意，token表示该单词不分词，必须为分词后的倒排索引中存在的单词，否则查询无意义。
- slop ：表示token1与token2之间允许间隔的最大token数，该值是一个上限。
- in_order ：可选项，表示是否按顺序依次匹配token。boolean型，默认值为true，按照顺序匹配，即token2必须出现在token1之后。若为false，token顺序不匹配的记录也会被检索到。

例 22. near语法用例

```
SQL表达：
select * from search_zh_near_ik where contains(content, 'near((南京,市长),2)');

SCOPE语义分别为：
GET /_search
{
    "query": {
        "span_near" : {
            "clausSCOPE" : [
                { "span_term" : { "content" : "南京" } },
                { "span_term" : { "content" : "市长" } }
            ],
            "slop" : 2,
            "in_order" : true
        }
    }
}
```

3.10.3.3. FUZYY 操作符

不同于 NEAR 操作符， FUZZY 操作符是针对短语而设计的，同样是在基础分词检索的基础上，查询相似的短语。 3 .fuzzy操作符使用规则

```
CONTAINS(<column>, 'fuzzy(phrase, fuzzinSCOPEs)')
```

总共包含2个参数：

- **phrase** ：必须先经过指定分词器分词，得到多个token，查询结果必须包含分词后的所有token。
- **fuzzinSCOPEs** ：fuzzinSCOPEs的值为0，1，2或者auto。最大编辑距离(Levenshtein距离)。计算查询结果中所有相应的token，在经过若干次编辑操作（删除，相邻交换）后，其相对位置与所查询的 **phrase** 分词后所有token的相对位置一致，所返回记录的编辑距离不可超过所设的最大值。

  在所查询的记录中，任意1次编辑操作（删除，相邻交换），其编辑距离都增1。

例 23. fuzzy语法用例

```
SQL表达：
select * from search_zh_near_ik where contains(content, 'fuzzy(南市,1)');

SCOPE语义：
GET /_search
{
    "query": {
        "query_string" : {
            "default_field" : "content",
            "query" : "南市~1"
        }
    }
}
```

这里的 **最大编辑距离** 与 near 操作符中的 **slop** 不同。编辑距离（fuzziness）的计算公式较为复杂，在具体实践中会存在 一定的误差 ，具体算法请参考 Damerau－Levenshtein distance。

针对*中文*字符的分词方式与英文字符不同，导致编辑距离计算难度增大的问题，只能将中文字符的距离描述转化为倾向性。所以中文的情况下，不再是简单的由编辑操作（删除，相邻交换）的步数决定距离的计数。

4. Transwarp Scope API介绍

4.1. Java API

4.1.1. Java Low Level REST Client

4.1.1.1. Getting started

Javadoc

1. 推荐JDK版本为8系列

2. 推荐使用的客户端版本为ES 672系列的客户端

Maven Repository

客户端时使用有多种方式：

1. 产品发布时时配套生成对应的SDK，可以直接从SDK中取出对应的scope-client相关的jar包
2. 产品为了更好的兼容开源生态，满足广大客户的迁移需求，可以直接采用开源的elasticsearch client相关的jar包，我们scope对应支持的elasticsearch接口版本为6.7.2 的REST API

```
    <!-- https://mvnrepository.com/artifact/org.elasticsearch.client/elasticsearch-rest-
high-level-client -->
        <dependency>
            <groupId>org.elasticsearch.client</groupId>
            <artifactId>elasticsearch-rest-client</artifactId>
            <version>6.7.2</version>
        </dependency>
```

```
dependencies {
    compile 'org.elasticsearch.client:elasticsearch-rest-client:6.7.2'
}
```

Initialization

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

4.1.1.2. Common configuration

Timeouts

可以通过在构建 RestClient 时提供 RequestConfigCallback 实例来配置请求超时。 该接口有一个方法接收 org.apache.http.client.config.RequestConfig.Builder 的实例作为参数，并具有相同的返回类型。

可以修改请求配置生成器，然后返回。 在下面的示例中，我们增加了连接超时（默认为 1 秒）和套接字超时（默认为 30 秒）。

```
RestClientBuilder builder = RestClient.builder(
    new HttpHost("localhost", 9200))
    .setRequestConfigCallback(
        new RestClientBuilder.RequestConfigCallback() {
            @Override
            public RequestConfig.Builder customizeRequestConfig(
                    RequestConfig.Builder requestConfigBuilder) {
                return requestConfigBuilder
                    .setConnectTimeout(5000)
                    .setSocketTimeout(60000);
            }
        });
```

Number of threads

默认情况下，Apache Http Async Client 启动一个调度程序线程以及连接管理器使用的多个工作线程，与本地检测到的处理器数量一样多（取决于 Runtime.getRuntime().availableProcessors() 返回的内容）。可以修改线程数如下：

```
RestClientBuilder builder = RestClient.builder(
    new HttpHost("localhost", 9200))
    .setHttpClientConfigCallback(new HttpClientConfigCallback() {
        @Override
        public HttpAsyncClientBuilder customizeHttpClient(
                HttpAsyncClientBuilder httpClientBuilder) {
            return httpClientBuilder.setDefaultIOReactorConfig(
                IOReactorConfig.custom()
                    .setIoThreadCount(1)
                    .build());
        }
    });
```

Basic authentication

可以通过构建 RestClient 时提供 HttpClientConfigCallback 来配置基本身份验证。 该接口有一个方法接收 org.apache.http.impl.nio.client.HttpAsyncClientBuilder 的实例作为参数，并具有相同的返回类型。

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


RestClientBuilder builder = RestClient.builder(
    new HttpHost("localhost", 9200))
    .setHttpClientConfigCallback(new HttpClientConfigCallback() {
        @Override
        public HttpAsyncClientBuilder customizeHttpClient(
                HttpAsyncClientBuilder httpClientBuilder) {
            return httpClientBuilder
                .setDefaultCredentialsProvider(credentialsProvider);
        }
    });
```

Node selector

客户端以循环方式将每个请求发送到配置的节点之一。 对于每个请求，客户端将运行最终配置的节点选择器

来过滤节点候选者，然后从剩余的节点中选择列表中的下一个。

```
RestClientBuilder builder = RestClient.builder(
        new HttpHost("localhost", 9200, "http"));
builder.setNodeSelector(new NodeSelector() {
    @Override
    public void select(Iterable<Node> nodes) {
        /*
         * Prefer any node that belongs to rack_one. If none is around
         * we will go to another rack till it's time to try and revive
         * some of the nodes that belong to rack_one.
         */
        boolean foundOne = false;
        for (Node node : nodes) {
            String rackId = node.getAttributes().get("rack_id").get(0);
            if ("rack_one".equals(rackId)) {
                foundOne = true;
                break;
            }
        }
        if (foundOne) {
            Iterator<Node> nodesIt = nodes.iterator();
            while (nodesIt.hasNext()) {
                Node node = nodesIt.next();
                String rackId = node.getAttributes().get("rack_id").get(0);
                if ("rack_one".equals(rackId) == false) {
                    nodesIt.remove();
                }
            }
        }
    }
});
```

Encrypted communication

加密通信也可以通过 HttpClientConfigCallback 进行配置。 作为参数接收的 org.apache.http.impl.nio.client.HttpAsyncClientBuilder 公开了多个配置加密通信的方法：setSSLContext、setSSLSessionStrategy 和 setConnectionManager。

```
KeyStore truststore = KeyStore.getInstance("jks");
try (InputStream is = Files.newInputStream(keyStorePath)) {
    truststore.load(is, keyStorePass.toCharArray());
}
SSLContextBuilder sslBuilder = SSLContexts.custom()
    .loadTrustMaterial(truststore, null);
final SSLContext sslContext = sslBuilder.build();
RestClientBuilder builder = RestClient.builder(
    new HttpHost("localhost", 9200, "https"))
    .setHttpClientConfigCallback(new HttpClientConfigCallback() {
        @Override
        public HttpAsyncClientBuilder customizeHttpClient(
                HttpAsyncClientBuilder httpClientBuilder) {
            return httpClientBuilder.setSSLContext(sslContext);
        }
    });
```

4.1.2. Java High Level REST Client

4.1.2.1. Getting started

Javadoc

1. 推荐JDK版本为8系列

2. 推荐使用的客户端版本为ES 672系列的客户端

Maven Repository

客户端时使用有多种方式：

1. 产品发布时时配套生成对应的SDK，可以直接从SDK中取出对应的scope-client相关的jar包
2. 产品为了更好的兼容开源生态，满足广大客户的迁移需求，可以直接采用开源的elasticsearch client相关的jar包，我们scope对应支持的elasticsearch接口版本为6.7.2 的highlevel REST API

例 24. Maven Configuration

```
     <!-- https://mvnrepository.com/artifact/org.elasticsearch.client/elasticsearch-rest-
high-level-client -->
        <dependency>
            <groupId>org.elasticsearch.client</groupId>
            <artifactId>elasticsearch-rest-high-level-client</artifactId>
            <version>6.7.2</version>
        </dependency>
```

例 25. Gradle Configuration

```
dependencies {
    compile 'org.elasticsearch.client:elasticsearch-rest-high-level-client:6.7.2'
}
```

为了便于大家理解，这里对于Scope的table定义，我们都用elasticsearch的index予以替换。

这里给出常用的部分api使用方式，细致的内容由于我们兼容开源使用方式，同样也可以参考 开源相关文档。

Initialization

例 26. 客户端实例的初始化与关闭

```
RestHighLevelClient client = new RestHighLevelClient(
        RestClient.builder(
                new HttpHost("localhost", 9200, "http"),
                new HttpHost("localhost", 9201, "http")));
......
client.close();
```

4.1.2.2. Document API

get

例 27. 获取数据

```
GetRequest posts = new GetRequest(index, type, documnetid);
//也可以省略type，对于type类型我们的传入都会被转化成默认的default_type_
// GetRequest posts = new GetRequest(index, documnetid);
GetResponse response = client.get(posts, RequestOptions.DEFAULT)
```

delete

例 28. 删除数据

```
DeleteRequest request = new DeleteRequest(indexName, type, documentid);
DeleteResponse response = client.indices().delete(request, RequestOptions.DEFAULT);
```

update

例 29. 更新操作

```
UpdateRequest request = new UpdateRequest(indexName, type, documentid);
request.doc(XContentType.JSON, field, value);
//支持string in json format,map等方式构建doc
UpdateResponse response = client.update(request, RequestOptions.DEFAULT);
```

bulk

例 30. 批量操作

```
BulkRequest request = new BulkRequest();
request.add(new IndexRequest(indexname, type, documnetid).source(XContentType.JSON,field,
value));
request.add(new IndexRequest("posts", "doc", "1").source(XContentType.JSON,"field", "foo"));
request.add(new IndexRequest("posts", "doc", "2").source(XContentType.JSON,"field", "bar"));
BulkResponse bulkResponses = client.bulk(request);
```

count

例 31. 统计

```
CountRequest request= new CountRequest().indices(indexName);
CountResponse response = client.count(request, RequestOptions.DEFAULT);
```

4.1.2.3. Indices API

create Index

在构建index过程中，Scope支持先构建mapping映射后进行数据读写操作的方式；**暂不支持** 直接插入数据自适应构建mapping的形式。同时，类似于高版本的elasticsearch取消了多type的mapping逻辑，**仅支持default_type_作为mapping的唯一type类型**。

例 32. 构建索引

```
CreateIndexRequest request = new CreateIndexRequest(index);
        request.settings(Settings.builder()
                .put("index.number_of_shards", 1)
                .put("index.number_of_replicas", 0)
        );
        request.mapping(TYPE_NAME,
                "   {\n" +
                        "    \"" + TYPE_NAME + "\": {\n" +
                        "      \"properties\": {\n" +
                        "        \"c_text\": {\n" +
                        "          \"type\": \"text\"\n" +
                        "        },\n" +
                        "        \"c_string_mf\": {\n" +
                        "          \"type\": \"keyword\"\n" +
                        "        }\n" +
                        "      }\n" +
                        "    }\n" +
                        "  }",
                XContentType.JSON);
CreateIndexResponse createIndexResponse = client.indices().create(request);
```

Delete Index

例 33. 删除索引

```
DeleteIndexRequest request = new DeleteIndexRequest(indexname);
AcknowledgedResponse deleteIndexResponse = client.indices().delete(request);
```

Get Mapping

例 34. 获取index schema

```
GetMappingsRequest request = new GetMappingsRequest();
request.indices(indexname);
GetMappingsResponse response = client.indices().getMapping(request, RequestOptions.DEFAULT);
```

Get Setting

例 35. 获取index的配置信息

```
GetSettingsRequest request = new GetSettingsRequest().indices(indexName);
GetSettingsResponse response = client.indices().getSettings(request, RequestOptions.DEFAULT);
```

Refresh

例 36. refresh

```
RefreshRequest request = new RefreshRequest(indexName)
RefreshResponse response = client.indices().refresh(request,RequestOptions.DEFAULT);
```

Update Settings

例 37. 更新index配置

```
UpdateSettingsRequest request = new UpdateSettingsRequest(indexName);
request.settings(Settings.builder().put("index.refresh_interval", "14m").build());
AcknowledgedResponse acknowledgedResponse = client.indices().putSettings(request,
RequestOptions.DEFAULT);
```

4.1.2.4. Cluster API

Get Cluster

例 38. 获取集群信息

```
MainResponse info = client.info(RequestOptions.DEFAULT);
```

4.1.2.5. Search API

查询部分的逻辑其基本模板为：

```
/*根据查询需求自定义对应querybuilder的类型*/
QueryBuilder queryBuilder= new XXXBuilder(argsp[1,...];)
/*将querybuilder放入查询请求中，这部分内容在查询中基本一致，后文不在赘述*/
SearchSourceBuilder sourceBuilder = new SearchSourceBuilder();
sourceBuilder.query(queryBuilder);
SearchRequest searchRequest = new SearchRequest(indexName);
searchRequest.source(sourceBuilder);
SearchResponse response = client.search(searchRequest, RequestOptions.DEFAULT);
```

term

例 39. 精确查询

```
QueryBuilder queryBuilder =
        new TermQueryBuilder(field,value);
```

terms

例 40. 多条件匹配查询

```
QueryBuilder queryBuilder =
        new TermsQueryBuilder(field,value1,value2);
```

range

例 41. 范围查询

```
QueryBuilder queryBuilder =
        new RangeQueryBuilder(field).from(value1).to(value2);
```

match

例 42. 匹配查询

```
QueryBuilder queryBuilder =
        new MatchQueryBuilder(field,value);
```

bool

例 43. 布尔查询

```
QueryBuilder queryBuilder =
        new BoolQueryBuilder().must(new MatchAllQueryBuilder());
```

prefix

例 44. 前模糊查询

```
QueryBuilder queryBuilder =
        new PrefixQueryBuilder(field,value);
```

match_phrase

例 45. 短语匹配查询

```
QueryBuilder queryBuilder =
        new MatchPhraseQueryBuilder(field,value);
```

regexp

例 46. 正则查询

```
QueryBuilder queryBuilder =
        new RegexpQueryBuilder(field,value);
```

wildcard

例 47. 模糊查询

```
QueryBuilder queryBuilder =
        new WildcardQueryBuilder(field,value);
```

scroll

例 48. scroll

```
SearchRequest searchRequest = new SearchRequest(indexName);
SearchRequest.scroll("5m");
SearchResponse response = client.search(searchRequest, RequestOptions.DEFAULT);
scrollID = response.getScrollId();
SearchScrollRequest request = new SearchScrollRequest(scrollID);
SearchResponse response = client.scroll(request, RequestOptions.DEFAULT);
```

script

例 49. 脚本语言查询

```
Script script = new Script("doc[\u0027number\u0027].value > 1");
//等价GET /_search { "query": { "bool" : { "filter" : { "script" : { "script" : { "source":
"doc['num1'].value > 1", "lang": "painless" } } } } } }
SearchResponse searchResponse = search(client, indexName, new ScriptQueryBuilder(script));
```

match_none

例 50. match_none

```
SearchResponse searchResponse =
        search(client, indexName, new
MatchNoneQueryBuilder().queryName("test_match_none").boost(2.0f));
```

terms set

例 51. terms set

```
List<Object> stringList = new ArrayList<>();
stringList.add("snno00000001");
stringList.add("snno00000002");
SearchResponse searchResponse1 =
        search(client, indexName, new TermsSetQueryBuilder(field2,
stringList).setMinimumShouldMatchField(field1));
```

4.1.2.6. 简单查询样例

例 52. 样例

```
import org.apache.http.HttpHost;
import org.elasticsearch.action.admin.indices.create.CreateIndexRequest;
import org.elasticsearch.action.admin.indices.delete.DeleteIndexRequest;
import org.elasticsearch.action.admin.indices.refresh.RefreshRequest;
import org.elasticsearch.action.admin.indices.refresh.RefreshResponse;
import org.elasticsearch.action.bulk.BulkRequest;
import org.elasticsearch.action.bulk.BulkResponse;
import org.elasticsearch.action.delete.DeleteRequest;
import org.elasticsearch.action.delete.DeleteResponse;
import org.elasticsearch.action.index.IndexRequest;
import org.elasticsearch.action.index.IndexResponse;
import org.elasticsearch.action.search.ClearScrollRequest;
import org.elasticsearch.action.search.ClearScrollResponse;
import org.elasticsearch.action.search.SearchRequest;
import org.elasticsearch.action.search.SearchResponse;
import org.elasticsearch.action.search.SearchScrollRequest;
import org.elasticsearch.action.support.IndicesOptions;
import org.elasticsearch.action.support.master.AcknowledgedResponse;
import org.elasticsearch.action.update.UpdateRequest;
import org.elasticsearch.action.update.UpdateResponse;
import org.elasticsearch.client.RequestOptions;
import org.elasticsearch.client.RestClient;
import org.elasticsearch.client.RestHighLevelClient;
import org.elasticsearch.common.network.NetworkService;
import org.elasticsearch.common.settings.Settings;
import org.elasticsearch.common.unit.TimeValue;
import org.elasticsearch.common.xcontent.XContentType;
import org.elasticsearch.index.query.MatchAllQueryBuilder;
import org.elasticsearch.index.query.QueryBuilders;
import org.elasticsearch.search.SearchHit;
import org.elasticsearch.search.SearchHits;
import org.elasticsearch.search.builder.SearchSourceBuilder;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;


import java.io.IOException; import java.util.concurrent.TimeUnit;

public class Sample {


  private static final Logger LOG = LoggerFactory.getLogger(Sample.class);
  private static final String TABLE_NAME_HASH = "search_muti_data_center_test_table_hash";
  private static final String TYPE_NAME = "default_type_";



  public static RestHighLevelClient getClient() {
    return new RestHighLevelClient(
            RestClient.builder(new HttpHost("localhost", 9200, "http")));
  }



  public static void deleteTable(RestHighLevelClient client, String tableName) {
    DeleteIndexRequest request = new DeleteIndexRequest(tableName);//指定要删除的索引名称
    //可选参数：
    request.timeout(TimeValue.timeValueMinutes(2)); //设置超时，等待所有节点确认索引删除（使用
  TimeValue形式）
    request.masterNodeTimeout(TimeValue.timeValueMinutes(1));//连接master节点的超时时间(使用
  TimeValue方式)



  //设置IndicesOptions控制如何解决不可用的索引以及如何扩展通配符表达式
  request.indicesOptions(IndicesOptions.lenientExpandOpen());



  try {
    //同步执行
    AcknowledgedResponse deleteIndexResponse = client.indices().delete(request);
```

```
    boolean acknowledged = deleteIndexResponse.isAcknowledged();//是否所有节点都已确认请求
    System.out.println("delete index[" + tableName + "] response: isAcknowledged is " +
acknowledged + "\t"
            + "deleteIndexResponse is " + deleteIndexResponse);
  } catch (Exception e) {
    System.out.println("table not found" + e);
  }
}
```

```
public static void createTable(RestHighLevelClient client, String tableName) throws IOException
{
  CreateIndexRequest request = new CreateIndexRequest(tableName);//创建索引
  //创建的每个索引都可以有与之关联的特定设置。
  request.settings(Settings.builder()
          .put("index.number_of_shards", 5)
          .put("index.number_of_replicas", 3);
  );
  //创建索引时创建文档类型映射
  request.mapping(TYPE_NAME,//类型定义
          "  {\n" +
                  "    \"" + TYPE_NAME + "\": {\n" +
                  "      \"properties\": {\n" +
                  "        \"c_text\": {\n" +
                  "          \"type\": \"text\"\n" +
                  "        },\n" +
                  "        \"c_string\": {\n" +
                  "          \"type\": \"keyword\"\n" +
                  "        },\n" +
                  "        \"c_integer\": {\n" +
                  "          \"type\": \"integer\"\n" +
                  "        },\n" +
                  "        \"c_string_mf\": {\n" +
                  "          \"type\": \"keyword\"\n" +
                  "        }\n" +
                  "      }\n" +
                  "    }\n" +
                  "  }",//类型映射，需要的是一个JSON字符串
          XContentType.JSON);
```

```
//可选参数
request.timeout(TimeValue.timeValueMinutes(2));//超时,等待所有节点被确认(使用TimeValue方式)
request.masterNodeTimeout(TimeValue.timeValueMinutes(1));//连接master节点的超时时间(使用TimeValue方
式)
request.waitForActiveShards(10);//在创建索引API返回响应之前等待的活动分片副本的数量，以int形式表示。
```

```
//同步执行
org.elasticsearch.client.indices.CreateIndexResponse createIndexResponse =
client.indices().create(request);
//返回的CreateIndexResponse允许检索有关执行的操作的信息，如下所示：
boolean acknowledged = createIndexResponse.isAcknowledged();//指示是否所有节点都已确认请求
boolean shardsAcknowledged =
createIndexResponse.isShardsAcknowledged();//指示是否在超时之前为索引中的每个分片启动了必需的分片副本数
```

```
  System.out.println("create index[" + tableName + "] response: isAcknowledged is " +
acknowledged + "\t"
          + ",isShardsAcknowledged is " + shardsAcknowledged + "\t" + ",createIndexResponse is
" + createIndexResponse);
}
```

```
public static void batchWrite(RestHighLevelClient client, String tableName) throws IOException,
InterruptedException {
  for (int i = 0; i < 10; i++) {
    BulkRequest request = new BulkRequest();
    for (int j = 0; j < 100; j++) {
      String stringValue = "value" + j;
      String textValue = "value" + j;
      String mfValue = "value" + j;
      if (j % 10 == 0) {
        stringValue = "transwarp";
        textValue = "I love transwarp";
        mfValue = "the transwarp";
      }
      request.add(new IndexRequest(tableName).type(TYPE_NAME).id(String.valueOf(i * 100 + j))
              .source(XContentType.JSON, "c_text", textValue, "c_string", stringValue,
"c_string_mf", mfValue));
    }
    BulkResponse bulkResponses = client.bulk(request);
    System.out.println("batch write response is " + bulkResponses);
  }
}
```

```
public static void matchAllQuery(RestHighLevelClient client, String tableName) throws
IOException {
  SearchSourceBuilder sourceBuilder = new SearchSourceBuilder();
  MatchAllQueryBuilder matchQueryBuilder = QueryBuilders.matchAllQuery();
  sourceBuilder.query(matchQueryBuilder); //设置查询，可以是任何类型的QueryBuilder。
  sourceBuilder.from(0); //设置确定结果要从哪个索引开始搜索的from选项，默认为0
  sourceBuilder.size(10000); //设置确定搜素命中返回数的size选项，默认为10
  sourceBuilder.timeout(new TimeValue(60, TimeUnit.SECONDS));
//设置一个可选的超时，控制允许搜索的时间。
```

```
SearchRequest searchRequest = new SearchRequest(tableName); //索引
searchRequest.types(TYPE_NAME); //类型
searchRequest.source(sourceBuilder);
SearchResponse response = client.search(searchRequest);
SearchHits hits = response.getHits();  //SearchHits提供有关所有匹配的全局信息，例如总命中数或最高分数：
SearchHit[] searchHits = hits.getHits();
System.out.println("matchAllQuery search result size is " + searchHits.length + ", content is:
");
for (SearchHit hit : searchHits) {
  System.out.println(hit.getId() + "\t" + hit.getSourceAsString());
}
```

```
  System.out.println("scrollID: " + response.getScrollId());
  if (searchHits.length != 1000) {
    throw new IllegalStateException("matchAllQuery result size is not true");
  }
}
```

```
public static void scrollSearch(RestHighLevelClient client, String tableName) throws Exception
{
  SearchSourceBuilder sourceBuilder = new SearchSourceBuilder();
  MatchAllQueryBuilder matchQueryBuilder = QueryBuilders.matchAllQuery();
  sourceBuilder.query(matchQueryBuilder); //设置查询，可以是任何类型的QueryBuilder。
  sourceBuilder.from(0); //设置确定结果要从哪个索引开始搜索的from选项，默认为0
  sourceBuilder.size(10); //设置确定搜素命中返回数的size选项，默认为10
  sourceBuilder.timeout(new TimeValue(60, TimeUnit.SECONDS));
//设置一个可选的超时，控制允许搜索的时间。
```

```
SearchRequest searchRequest = new SearchRequest(tableName); //索引
searchRequest.types(TYPE_NAME); //类型
searchRequest.source(sourceBuilder);
searchRequest.scroll("5m");//游标查询的过期时间会在每次做查询的时候刷新，所以这个时间只需要足够处理当前批的
结果就可以了
String scrollID = null;
int searchResultLength = 0;
{
  SearchResponse response = client.search(searchRequest);
  SearchHits hits = response.getHits();
//SearchHits提供有关所有匹配的全局信息，例如总命中数或最高分数：
  SearchHit[] searchHits = hits.getHits();
  System.out.println("the 1st search, get data size: " + searchHits.length + ", the detail is:
");
  searchResultLength += searchHits.length;
  for (SearchHit hit : searchHits) {
    System.out.println("search hit id: " + hit.getId() + "\t" + ", search hit content: " +
hit.getSourceAsString()
            + "\t" + ", search hit length: " + searchHits.length);
  }
  scrollID = response.getScrollId();
  System.out.println("scrollID: " + scrollID);
}



Thread.sleep(10000);



int i = 2;
while (true) {
  SearchScrollRequest request = new SearchScrollRequest(scrollID);
  SearchResponse response = client.scroll(request, RequestOptions.DEFAULT);
  SearchHits hits = response.getHits();
//SearchHits提供有关所有匹配的全局信息，例如总命中数或最高分数：
  if (null == hits || hits.getHits() == null || hits.getHits().length == 0) {
    break;
  }
  SearchHit[] searchHits = hits.getHits();
  System.out.println("the " + i + "st search, get data size: " + searchHits.length + ", the
detail is: ");
  searchResultLength += searchHits.length;
  for (SearchHit hit : searchHits) {
    System.out.println("search hit id: " + hit.getId() + "\t" + ", search hit content: " +
hit.getSourceAsString()
            + "\t" + ", search hit length: " + searchHits.length);
  }
  scrollID = response.getScrollId();
  System.out.println("scrollID: " + scrollID);
  i++;
}



if (searchResultLength != 1000) {
  throw new IllegalStateException("scroll search result size is not true");
}



  ClearScrollRequest clearScrollRequest = new ClearScrollRequest();
  clearScrollRequest.addScrollId(scrollID);
  ClearScrollResponse clearScrollResponse = client.clearScroll(clearScrollRequest);
  System.out.println("clear scroll response is " + clearScrollResponse);
}



public static void singleWrite(RestHighLevelClient client, String tableName) throws Exception {
  {
    IndexRequest request = new
IndexRequest(tableName).type(TYPE_NAME).id("4").source(XContentType.JSON, "c_string", "test");
    IndexResponse response = client.index(request);
    System.out.println(response);
  }
```

```
{
  UpdateRequest request = new UpdateRequest(tableName, TYPE_NAME, "4");
  request.doc(XContentType.JSON, "c_string", "test1");
  UpdateResponse response = client.update(request);
  System.out.println(response);
}



{
  UpdateRequest request = new UpdateRequest(tableName, TYPE_NAME, "5");
  request.doc(XContentType.JSON, "c_string", "test1");
  UpdateResponse response = client.update(request);
  System.out.println(response);
}



{
  DeleteRequest request = new DeleteRequest(tableName, TYPE_NAME, "4");
  DeleteResponse response = client.delete(request);
  System.out.println(response);
}



{
  IndexRequest request = new
IndexRequest(tableName).type(TYPE_NAME).id("4").source(XContentType.JSON, "c_string", "test");
  IndexResponse response = client.index(request);
  System.out.println(response);
}



{
  DeleteRequest request = new DeleteRequest(tableName, TYPE_NAME, "6");
  DeleteResponse response = client.delete(request);
  System.out.println(response);
}



  {
    IndexRequest request = new
IndexRequest(tableName).type(TYPE_NAME).id("6").source(XContentType.JSON, "c_string", "test");
    IndexResponse response = client.index(request);
    System.out.println(response);
  }
}



public static void main(String[] args) throws Exception {



// generate client
RestHighLevelClient restHighLevelClient = getClient();



// delete table first
System.out.println("============================delete index============================");
deleteTable(restHighLevelClient, TABLE_NAME_HASH);



// create table
System.out.println("============================create index============================");
createTable(restHighLevelClient, TABLE_NAME_HASH);



// write data
System.out.println("============================write data============================");
batchWrite(restHighLevelClient, TABLE_NAME_HASH);
RefreshResponse refreshResponse = restHighLevelClient.indices().refresh(new
RefreshRequest(TABLE_NAME_HASH), RequestOptions.DEFAULT);
System.out.println("refresh response is " + refreshResponse);
```

```
    // search, matchAllQuery
    System.out.println("============================match all query============================");
    matchAllQuery(restHighLevelClient, TABLE_NAME_HASH);



    // scrollSearch, use matchAllQuery
    System.out.println("============================scroll search============================");
    scrollSearch(restHighLevelClient, TABLE_NAME_HASH);



    // singe write
    System.out.println("============================single write============================");
    singleWrite(restHighLevelClient, TABLE_NAME_HASH);
    refreshResponse = restHighLevelClient.indices().refresh(new RefreshRequest(TABLE_NAME_HASH),
    RequestOptions.DEFAULT);
    System.out.println("refresh response is " + refreshResponse);



    // search again
    System.out.println("============================search again============================");
    matchAllQuery(restHighLevelClient, TABLE_NAME_HASH);



      restHighLevelClient.close();
    }


  }
```

4.2. QueryAPI

本节内容会详细介绍一些Query过程中经常或者可能使用到一些api。

url结尾含有'/'是不被允许的，例如：`curl -X GET <host>:<port>/table/`将会报错

4.2.1. Query and filter context

查询子句的行为取决于它是在query context中使用还是在filter context中使用：

4.2.1.1. Query context

query context中使用的query子句可以得到此文档与此query子句的匹配程度。除了决定文档是否匹配外，query子句还会计算出一个分数score，表示此文档相对于其他文档的匹配程度。

4.2.1.2. Filter context

在Filter context中，query子句会得到此文档是否与此query子句匹配的答案：——是或不是 （不计算分数）。

4.2.2. Match All Query

最简单的查询，匹配所有文档，所有文档的分数均为1.0。

```
curl -XGET <host>:<port>/<index_name>/_search { "query": { "match_all": {} } }
```

```
curl -XGET "localhost:9200/match_all_index/_search?pretty" -H 'Content-Type: application/json'
-d'
{
   "query": {
     "match_all": {

     }
   }
}'
{
  "took" : 181,
  "timed_out" : false,
  "terminated_early" : false,
  "_shards" : {
    "total" : 1,
    "successful" : 1,
    "skipped" : 0,
    "failed" : 0
  },
  "_clusters" : {
    "total" : 1,
    "successful" : 1,
    "skipped" : 0
  },
  "hits" : {
    "total" : 1,
    "max_score" : null,
    "hits" : [
      {
        "_index" : "match_all_index",
        "_type" : "default_type_",
        "_id" : "3",
        "_score" : null,
        "_source" : {
          "msg" : "no scheme",
          "url_address" : "developer.mozilla.org/en-US/search?q=URL"
        }
      }
    ]
  }
}
```

可以使用boost参数更改分数：

```
curl –XGET <host>:<port>/<index_name>/_search { "query": { "match_all": { "boost" : 1.2 } } }
```

4.2.3. full text queries

4.2.3.1. Match Query

match query可以查询文本/数字/日期类型的数据。

```
curl –XGET <host>:<port>/<index_name>/_search { "query": { "match": { "message": "this is a test" } } }
```

注意，message是一个字段的名称，你可以用任何字段的名称来代替。

```
curl -XGET "localhost:9200/match_query_index/_search?pretty" -H 'Content-Type:
application/json' -d'
{
    "query": {
      "match": {
        "url_address": "https://developer.mozilla.org"
      }
  }
}'
{
  "took" : 112,
  "timed_out" : false,
  "terminated_early" : false,
  "_shards" : {
    "total" : 1,
    "successful" : 1,
    "skipped" : 0,
    "failed" : 0
  },
  "_clusters" : {
    "total" : 1,
    "successful" : 1,
    "skipped" : 0
  },
  "hits" : {
    "total" : 1,
    "max_score" : 1.0893298,
    "hits" : [
      {
        "_index" : "match_query_index",
        "_type" : "default_type_",
        "_id" : "3",
        "_score" : 1.0893298,
        "_source" : {
          "msg" : "no scheme",
          "url_address" : "developer.mozilla.org/en-US/search?q=URL"
        }
      }
    ]
  }
}
```

match

match query为一种布尔类型查询。这意味着对所提供的文本进行分析，分析过程根据所提供的文本构造一个布尔查询。operator参数可以设置为OR或AND来控制布尔子句(默认为OR)，通过operator参数可以控制匹配关系。匹配的可选should子句的最小数量可以使用minimum_should_match参数来设置。

下面是一个提供额外参数的示例(注意结构上的变化，message是字段名)：

```
curl –XGET <host>:<port>/<index_name>/_search { "query": { "match": { "message": { "query" : "this is a test",  "operator" : "AND" } } } }
```

analyzer参数可以设置为控制哪个analyzer将对文本执行分析过程。默认的analyzer是字段中的mapping显式定义的analyzer或default search analyzer。

可以将lenient参数设置为true，以忽略由数据类型不匹配引起的异常，例如尝试使用文本查询去查询数字字段。默认值为false。

Fuzziness

fuzziness参数允许基于被查询字段的类型进行模糊匹配。在这种情况下，可以设置prefix_length和max_expansion来控制模糊过程。

注意，模糊匹配不适用于具有同义词的词项。

```
curl –XGET <host>:<port>/<index_name>/_search { "query": { "match": { "message": { "query" : "this is a test",  "fuzziness" : "AUTO" } } } }
```

zero term query

如果所使用的analyzer删除了查询中的所有词元，则不会匹配到任何文档。为了改变zero_terms_query参数的使用，它的默认值是none，若设置为all则对应于match_all查询。

```
curl –XGET <host>:<port>/<index_name>/_search { "query": { "match": { "message": { "query" : "this is a test",  "zero_terms_query": "all" } } } }
```

4.2.3.2. Match Phrase Query

match_phrase查询分析文本并根据分析的文本创建一个短语查询。

```
curl –XGET <host>:<port>/<index_name>/_search { "query": { "match_phrase": { "message": "this is a test" } } }
```

```
curl -X GET "localhost:9200/match_phrase_index/_search?&pretty" -H 'Content-Type:
application/json' -d'
{
    "query": {
      "match_phrase": {
        "mtext": "Brown fox"
      }
    }
}'
{
  ...
  "hits" : {
    "total" : 2,
    "max_score" : 0.19932948,
    "hits" : [
      {
        "_index" : "match_phrase_index",
        "_type" : "default_type_",
        "_id" : "2",
        "_score" : 0.19932948,
        "_source" : {
          "mtext" : "Brown fox eat rabbits",
          "mkeyword" : "Brown fox eat rabbits"
        }
      },
      {
        "_index" : "match_phrase_index",
        "_type" : "default_type_",
        "_id" : "1",
        "_score" : 0.1325215,
        "_source" : {
          "mtext" : "The 2 QUICK Brown-Foxes jumped over the lazy dog's Bone.",
          "mkeyword" : "The 2 QUICK Brown-Foxes jumped over the lazy dog's Bone."
        }
      }
    ]
  }
}
```

slop用于指定查询短语间的词项(term)间的距离，默认值是0。analyzer参数可以设置哪个analyzer将对文本执行分析过程。

curl -XGET <host>:<port>/<index_name>/_search { "query": { "match_phrase": { "message": { "query": "quick brown", "slop": 10 } } } }

和match query一样，match phrase query也可以接受zero_terms_query。

4.2.3.3. Match Phrase Prefix Query

match phrase prefix query与match phrase query相同，但是它多了一个特性，它允许对文本中的最后一个词进行前缀匹配。

curl -XGET <host>:<port>/<index_name>/_search { "query": { "match_phrase_prefix": { "message": "quick brown f" } } }

```
curl -X GET "localhost:9200/match_phrase_prefix_index/_search?&pretty" -H 'Content-Type:
application/json' -d'
{
   "query": {
     "match_phrase_prefix": {
       "mtext": {
         "query": "quick brown f"
       }
     }
   }
}'
{
  ...
  "hits" : {
    "total" : 1,
    "max_score" : 3.2995868,
    "hits" : [
      {
        "_index" : "match_phrase_prefix_index",
        "_type" : "default_type_",
        "_id" : "1",
        "_score" : 3.2995868,
        "_source" : {
          "mtext" : "The 2 QUICK Brown-Foxes jumped over the lazy dog's Bone.",
          "mkeyword" : "The 2 QUICK Brown-Foxes jumped over the lazy dog's Bone."
        }
      }
    ]
  }
}
```

match phrase prefix的max_expansion参数可以控制最后一项将扩展多少个后缀。建议将其设置为可接受的值以控制查询的执行时间。以quick brown f查询为例，它会先查找第一个与前缀f匹配的词，然后依次查找搜集与之匹配的词(按字母顺序)，直到没有更多可匹配的词或当数量超过max_expansions时结束。但是max_expansions是作用在分片级别（shard level）的，这意味着即使设置为1，依然有可能匹配到多个词。

curl -XGET <host>:<port>/<index_name>/_search { "query": { "match_phrase_prefix": { "message": { "query": "quick brown f", "max_expansions": 2 } } } }

match phrase prefix query非常容易使用，可以快速地开始搜索，但它的结果有时可能会不尽人意。

考虑查询字符串quick brown f。该查询通过创建一个由quick和brown组成的查询语句来进行查询(即quick必须存在，并且必须后面跟着brown)。然后，它查看排序后的词项字典，找到以f开头的前50个词项，并将这些词项添加到短语查询中。

问题是前50个词项中可能不包括fox，这样短语quick brown fox就不会被查到。可以通过继续输入更多的字母的方法找到要找到的结果。

4.2.3.4. Multi Match Query

multi match query建立在match query的基础上，允许多字段查询:

```
curl -XGET <host>:<port>/<index_name>/_search { "query": { "multi_match" : { "query": "this is a test", "fields": [ "subject", "message" ] } } }
```

```
curl -X GET "localhost:9200/multi_match_index/_search?pretty" -H 'Content-Type:
application/json' -d'{
    "query": {
      "multi_match" : {
        "query":      "mouse",
        "fields":     [ "lastname","abstract" ]
      }
    }
}'
{
  ...
  "hits" : {
    "total" : 1,
    "max_score" : 1.8032517,
    "hits" : [
      {
        "_index" : "multi_match_index",
        "_type" : "default_type_",
        "_id" : "4",
        "_score" : 1.8032517,
        "_source" : {
          "firstname" : "Industrious",
          "abstract" : "mouse eat rices",
          "title" : "mouse are also Pets",
          "content" : "Why hamster is a pet, mouse is not a pet",
          "lastname" : "Dark"
        }
      }
    ]
  }
}
```

fields and per-field boosting

可以使用通配符指定字段

```
curl -XGET <host>:<port>/<index_name>/_search { "query": { "multi_match" : { "query": "Will Smith", "fields": [ "title", "*_name" ] } } }
```

单个字段可以使用符号(^)来增强，下面的例子表示subject字段的重要性是message字段的三倍。

```
curl -XGET <host>:<port>/<index_name>/_search { "query": { "multi_match" : { "query" : "this is a test", "fields" : [ "subject^3", "message" ] } } }
```

如果没有提供字段，multi match query默认为*。*会将所有字段组合起来以构建一个查询。

Types of multi_match query

multi match query在内部执行的方式取决于type参数，它可以设置为：

best_fields：(默认值)查找匹配任何字段的文档，但取最大的匹配的score。

most_fields：查找匹配任意字段的文档，并把每个字段的score相加。

cross_fields：用相同的analyzer处理字段，查找任何领域的每个单词。

phrase：对每个字段进行match phrase query，并取最大的匹配的score。

phrase_prefix：对每个字段运行一个match phrase prefix query，并将每个字段的score相加。

best_fields

best_fields类型为每个字段生成一个匹配查询，以找到唯一的最佳匹配字段。

```
curl -XGET <host>:<port>/<index_name>/_search { "query": { "multi_match" : { "query": "brown fox", "type": "best_fields", "fields": [ "subject", "message" ] } } }
```

most_fields

与best_fields不同之处在于相关性评分，best_fields取最大匹配得分（max计算），而most_fields取所有匹配之和（sum计算）。

```
curl -XGET <host>:<port>/<index_name>/_search { "query": { "multi_match" : { "query": "quick brown fox", "type": "most_fields", "fields": [ "title", "title.original", "title.shingles" ] } } }
```

相当于执行以下查询语句

```
curl -XGET <host>:<port>/<index_name>/_search { "query": { "bool": { "should": [ { "match": { "title": "quick brown fox" }}, { "match": { "title.original": "quick brown fox" }}, { "match": { "title.shingles": "quick brown fox" }} ] } } }
```

phrase and phrase_prefix

phrase和phrase_prefix与best_fields类似，但它们使用的是match_phrase query或match_phrase_prefix query，而不是match query。

```
curl -XGET <host>:<port>/<index_name>/_search { "query": { "multi_match" : { "query": "quick brown f", "type": "phrase_prefix", "fields": [ "subject", "message" ] } } }
```

cross_fields

cross_fields对于需要匹配多个字段的结构化文档特别有用。例如，当查询“Will Smith”的first_name和last_name字段时，最好的匹配是一个字段中有“Will”，另一个字段中有“Smith”。

但是这种方法有个问题，与相关性有关:first_name和last_name字段中的词项频率不同，可能会产生意想不

到的结果。例如，假设有两个人:“Will Smith”和“Smith Jones”。“Smith”作为last_name是很常见的(因此不太重要)，但“Smith”作为first_name是很少见的(因此很重要)。如果搜索“Will Smith”，“Smith Jones”文档可能会出现在更匹配的“Will Smith”之上，因为first_name: Smith的分数超过了first_name: Will + last_name: Smith的分数之和。

处理这类查询的一种方法是将first_name和last_name字段索引到单个full_name字段中。当然，这只能在创建索引时完成。

cross_field试图在查询时解决这些问题。它首先将查询字符串分析为单个词项，然后在任何字段中查找每个词项，就像它们是一个大字段一样。

```
curl -XGET <host>:<port>/<index_name>/_search { "query": { "multi_match" : { "query": "Will Smith", "type": "cross_fields", "fields": [ "first_name", "last_name" ], "operator": "and" } } }
```

4.2.4. Term level queries

4.2.4.1. Term Query

term query查找包含倒排索引中指定的准确词项的文档。

```
curl -XGET <host>:<port>/<index_name>/_search { "query": { "term" : { "user" : { "value": "Kimchy" } } } }
```

```
curl -XGET "localhost:9200/term_query_index/_search?pretty" -H 'Content-Type: application/json'
-d'
{
    "query": {
      "term": {
        "url_address": {
          "value": "https://www.google.com/"
        }
      }
    }
}'
{
  ...
  "hits" : {
    "total" : 1,
    "max_score" : 0.28004453,
    "hits" : [
      {
        "_index" : "term_query_index",
        "_type" : "default_type_",
        "_id" : "3",
        "_score" : 0.28004453,
        "_source" : {
          "msg" : "long",
          "url_address" : "https://www.google.com/search?q=url&sxsrf=APq-
WBvQC6by6yojrKX42LpO9KpY5IEiBw%3A1646378551496"
        }
      }
    ]
  }
}
```

可以指定一个boost参数来给这个查询一个比其他查询更高的相关性评分。

curl -XGET <host>:<port>/<index_name>/_search { "query": { "term": { "name": "Charlie Paul" "boost": 3.0 } } }

4.2.4.2. Terms Query

terms query可以实现将一个字段，从多个value中检索的效果,类似于sql语句中的in。

curl -XGET <host>:<port>/<index_name>/_search { "query": { "terms" : { "user" : ["kimchy", "Scope"]} } }

```
curl -XGET "localhost:9200/search_index/_search?&pretty" -H 'Content-Type: application/json'
-d'
{
   "query": {
     "terms": {
     "name.keyword": ["Charlie Puth"]
     }
   }
}'
{
  ...
  "hits" : {
    "total" : 1,
    "max_score" : 1.0,
    "hits" : [
      {
        "_index" : "search_index",
        "_type" : "default_type_",
        "_id" : "9",
        "_score" : 1.0,
        "_source" : {
          "name" : "Charlie Puth",
          "age" : 43,
          "hobby" : null
        }
      }
    ]
  }
}
```

4.2.4.3. Range Query

在一定范围内将文档与具有词项的字段进行匹配。Lucene查询的类型取决于字段类型，对于字符串字段，是TermRangeQuery，而对于数字/日期字段，是NumericRangeQuery。

curl -XGET <host>:<port>/<index_name>/_search { "query": { "range" : { "age" : { "gte" : 10, "lte" : 20, "boost" : 2.0 } } } }

```
curl -X GET "localhost:9200/range_query_index/_search?&pretty&" -H 'Content-Type:
application/json' -d'
{
    "query": {
      "range": {
        "age": {
          "gte": 10,
          "lte": 24,
          "boost": 2.0
        }
      }
    }
}'
{
  ...
  "hits" : {
    "total" : 1,
    "max_score" : 2.0,
    "hits" : [
      {
        "_index" : "range_query_index",
        "_type" : "default_type_",
        "_id" : "2",
        "_score" : 2.0,
        "_source" : {
          "name" : "james",
          "age" : 23
        }
      }
    ]
  }
}
```

range query接受以下这些参数:

gte：大于等于

gt：大于

lte：小于等于

lt：小于

boost：设置查询的增强值，默认为1.0

Ranges on date field

当对date类型的字段进行范围查询时，范围可以使用date Math指定:

```
curl −XGET <host>:<port>/<index_name>/_search { "query": { "range" : { "date" : { "gte" : "now−1d/d",
"lt" : "now/d" } } } }
```

Date math and Rounding

当使用date Math将日期四舍五入到最近的日、月、小时等时，四舍五入的日期取决于范围的两端是包含的还是不包含的。

向上四舍五入移动到四舍五入范围的最后一毫秒，向下四舍五入到四舍五入范围的第一毫秒。

gt：大于四舍五入的日期:2014-11-18||/M变成2014-11-30T23:59:59.999，即不包括整个月。

gte：大于或等于四舍五入的日期:2014-11-18||/M为2014-11-01，即包括整个月。

lt：小于四舍五入的日期:2014-11-18||/M变成2014-11-01，即不包括整个月。

lte：小于或等于四舍五入的日期:2014-11-18||/M变为2014-11-30T23:59:59.999，即包括整个月。

Date format in range queries

默认情况下，格式化的日期将使用date字段指定的格式进行解析，但它可以通过将format参数传递给range query来覆盖:

```
curl -XGET <host>:<port>/<index_name>/_search { "query": { "range" : { "born" : { "gte": "01/01/2012", "lte": "2013", "format": "dd/MM/yyyy||yyyy" } } } }
```

注意，如果日期漏掉了年或月或日，则用unix时间的开始(1970年1月1日)来填充缺失的部分。这意味着，例如，当指定dd作为格式时，像"gte": 10这样的值将被转换为1970-01-10T00:00:00.000Z。

Time Zone in range quries

日期可以从另一个时区转换为UTC，要么通过在日期值本身指定时区(如果格式接受)，要么可以指定为time_zone参数:

```
curl -XGET <host>:<port>/<index_name>/_search { "query": { "range" : { "timestamp" : { "gte": "2015-01-01 00:00:00", "lte": "now", "time_zone": "+01:00" } } } }
```

此日期将转换为2014-12-31T23:00:00 UTC。

4.2.4.4. Exists Query

返回原始字段中至少有一个非空值的文档:

```
curl -XGET <host>:<port>/<index_name>/_search { "query": { "exists" : { "field" : "user" } } }
```

missing query

而是在must_not子句中使用exists查询实现missing query

```
curl -XGET <host>:<port>/<index_name>/_search { "query": { "bool": { "must_not": { "exists": { "field": "user" } } } } }
```

这个查询会返回在user这个字段没有值的文档。

4.2.4.5. Prefix Query

匹配具有包含具有指定前缀的词项的字段的文档(未被分析)。

```
curl -XGET <host>:<port>/<index_name>/_search { "query": { "prefix" : { "user" : "ki" } } }
```

```
curl -X GET "localhost:9200/prefix_index/_search?&pretty&" -H 'Content-Type: application/json'
-d'
{
   "query": {
     "prefix": {
       "user": "ki"
     }
   }
}'
{
  ...
  "hits" : {
    "total" : 1,
    "max_score" : 1.0,
    "hits" : [
      {
        "_index" : "prefix_index",
        "_type" : "default_type_",
        "_id" : "3",
        "_score" : 1.0,
        "_source" : {
          "user" : "kimo"
        }
      }
    ]
  }
}
```

4.2.4.6. Wildcard Query

匹配具有匹配通配符表达式(未被分析)的字段的文档。支持的通配符是*，匹配任何字符序列(包括空字符)，和?，匹配任何单个字符。注意，这个查询可能很慢，因为它需要遍历许多词项。为了使得wildcard query速度加快，通配符词项不应该以通配符*或?之一开头。

```
curl −XGET <host>:<port>/<index_name>/_search { "query": { "wildcard" : { "user" : "ki*y" } } }
```

```
curl  -X GET "localhost:9200/wildcard_index/_search?pretty" -H 'Content-Type: application/json'
-d'
{
   "query": {
     "wildcard": {
       "lyricE": {
       "value": "s*"
       }
     }
   }
}'
{
  ...
  "hits" : {
    "total" : 1,
    "max_score" : 1.0,
    "hits" : [
      {
        "_index" : "wildcard_index",
        "_type" : "default_type_",
        "_id" : "1",
        "_score" : 1.0,
        "_source" : {
          "lyricC" : "你的眼里有星河闪烁",
          "lyricE" : "There are stars in your eyes"
        }
      }
    ]
  }
}
```

4.2.4.7. Regexp Query

regexp query允许使用正则表达式进行查询。

注意:regexp query的性能在很大程度上取决于所选择的正则表达式。匹配像.*这样的所有东西和使用查找正则表达式是非常缓慢的。如果可能的话，应该尝试在正则表达式开始之前使用长前缀。

```
curl -XGET <host>:<port>/<index_name>/_search { "query": { "regexp":{ "name.first": "s.*y" } } }
```

```
curl  -X GET "localhost:9200/regexp_index/_search?pretty" -H 'Content-Type: application/json'
-d'
{
   "query": {
     "regexp": {
       "value": {
         "value": ".*.*..*"
       }
     }
   }
}'
{
  ...
  "hits" : {
    "total" : 1,
    "max_score" : 1.0,
    "hits" : [
      {
        "_index" : "regexp_index",
        "_type" : "default_type_",
        "_id" : "3",
        "_score" : 1.0,
        "_source" : {
          "value" : "aaba10aba20ab"
        }
      }
    ]
  }
}
```

可以使用flag参数，为正则表达式启用可选运算符

```
curl -XGET <host>:<port>/<index_name>/_search { "query": { "regexp":{ "name.first": { "value": "s.*y",
"flags" : "INTERSECTION|COMPLEMENT|EMPTY" } } } }
```

特殊标志有ALL(默认)，ANYSTRING, COMPLEMENT, EMPTY, INTERSECTION, INTERVAL或NONE。

正则表达式很容易意外地创建一个看起来无害的表达式，它需要大量的内部确定自动机状态(以及相应的RAM和CPU)供Lucene执行。Lucene使用max_determinized_states设置(默认为10000)来阻止这些，可以提高这个限制以允许执行更复杂的正则表达式。

```
curl -XGET <host>:<port>/<index_name>/_search { "query": { "regexp":{ "name.first": { "value": "s.*y",
"flags" : "INTERSECTION|COMPLEMENT|EMPTY", "max_determinized_states": 20000 } } } }
```

4.2.5. Compound queries

4.2.5.1. Bool Query

一个查询，使用其他查询的布尔值组合进行匹配查询，它使用一个或多个布尔子句构建。

must：子句(查询)必须出现在匹配的文档中。

filter：子句(查询)必须出现在匹配的文档中。

should：子句(查询)应该出现在匹配的文档中。

must_not：子句(查询)不能出现在匹配的文档中。

bool query采用more-matches-is-better方法，因此每个匹配的must或should子句的分数将被加在一起，为每个文档提供最终的分数。

```
curl -XPOST<host>:<port>/<index_name>/_search { "query": { "bool" : { "must" : { "term" : { "user" : "kimchy" } }, "filter": { "term" : { "tag" : "tech" } }, "must_not" : { "range" : { "age" : { "gte" : 10, "lte" : 20 } } }, "should" : [ { "term" : { "tag" : "wow" } }, { "term" : { "tag" : "elasticsearch" } } ], "minimum_should_match" : 1, "boost" : 1.0 } } }
```

```
curl  -X POST "localhost:9200/bool_index/_search?pretty" -H 'Content-Type: application/json'
-d'
{
   "query": {
     "bool" : {
       "must_not" : {
         "range" : {
           "price" : { "lte" : 6.99 }
           }
         },
     "should" : [
       { "term" : { "avaliable" : true } }
      ],
    "boost" : 2.0
     }
   }
}'
{
  ...
  "hits" : {
    "total" : 2,
    "max_score" : 0.26706278,
    "hits" : [
      {
        "_index" : "bool_index",
        "_type" : "default_type_",
        "_id" : "2",
        "_score" : 0.26706278,
        "_source" : {
          "name" : "Jump Ropes",
          "description" : "Lusuroi Jump Rope, Cable Adjustable Skipping Rope for Workout
Fitness Exercise Training, Speed Jumping Rope for Kids Beginners Women Adults, Tangle-Free with
Ball Bearings Gym Rope",
          "productID" : "C08H1P827X",
          "price" : 13.99,
          "avaliable" : true
        }
      },
      {
        "_index" : "bool_index",
        "_type" : "default_type_",
        "_id" : "3",
        "_score" : 0.26706278,
        "_source" : {
          "name" : "Beaded Jump Rope",
          "description" : "Beaded Jump Rope Tangle-Free with Ball Bearings Rapid Adjustable
Skipping Rope Non-Slip Handle Speed Suitable for Fitness Workout Adult Kids Men Women-2 Packs",
          "productID" : "B087681ZJ2",
          "price" : 7.99,
          "avaliable" : true
        }
      }
    ]
  }
}
```

4.2.6. Minimum Should Match

mininum_should_match参数可以有以下几种类型的值：

Integer:表示一个固定值，与可选子句的数量无关。

Negative integer:表示可选子句的总数减去这个数字应该是强制性的。

Percentage:表示该可选子句总数的百分比是必需的。从百分比计算的数字向下舍入并用作最小值。

Negative percentage:表示可能缺少可选子句总数的这个百分比。从百分比计算的数字向下舍入，然后从总数中减去以确定最小值。

Combination:一个正整数，后跟小于号。它表示如果可选子句的数量等于（或小于）整数，则它们都是必需的，但如果大于整数，则适用规范。

Multiple combinations:多个条件说明可以用空格分隔，每个条件仅对大于其前一个的数字有效。

4.2.6.1. Function Score Query

function_score 允许您修改查询检索到的文档的分数。

要使用 function_score，用户必须定义一个查询和一个或多个函数，为查询返回的每个文档计算一个新分数。

```
GET /_search { "query": { "function_score": { "query": { "match_all": {} }, "boost": "5", "random_score": {}, "boost_mode":"multiply" } } }
```

4.3. Cat API

本节内容会详细介绍一些在Scope集群运维过程中经常或者可能使用到一些api。以此帮助大家更好的掌握集群情况并进行运维。

curl -X GET <host>:<port>/help可以获取使用说明;
在query string中加入pretty可以使结果信息更加美观易读，例如：curl -X GET <host>:<port>/tables?pretty

url结尾含有'/'是不被允许的，例如：`curl -X GET <host>:<port>/table/`将会报错

查看CAT命令： curl <host>:<port>/_cat

```
transwarp@transwarp-ThinkPad-T14-Gen-1:~/mygit1/shiva/build/bin$ curl -XGET
localhost:19200/_cat
=^.^=
/_cat/health
/_cat/allocation
/_cat/allocation/{nodes}
/_cat/count
/_cat/count/{index}
/_cat/indices
/_cat/indices/{index}
/_cat/master
/_cat/nodes
/_cat/nodes/{node}
/_cat/shards
/_cat/shards/{index}
/_cat/templates
/_cat/thread_pool
/_cat/thread_pool/{thread_pool_patterns}
```

4.3.1. cat aliases

aliases 显示有关当前配置的索引别名的信息，包括过滤器和路由信息。

```
curl <host>:<port>/_cat/aliases?v
```

```
transwarp@transwarp-ThinkPad-T14-Gen-1:~/mygit1/shiva/build/bin$ curl -XGET
"localhost:19200/_cat/aliases?v"
alias index
```

4.3.2. cat allocation

allocation 提供了每个数据节点分配了多少分片以及它们正在使用多少磁盘空间的快照。

```
curl <host>:<port>/_cat/allocation?v
```

```
curl -XGET "localhost:19200/_cat/allocation?v"
shards disk.indices disk.used disk.avail disk.total disk.percent ip          port
     3       12.8kb         *          *          *            * 192.168.1.8 27751
     3       12.8kb         *          *          *            * 192.168.1.8 27731
     3       15.3kb         *          *          *            * 192.168.1.8 27711
     3       15.3kb         *          *          *            * 192.168.1.8 27721
     3       15.3kb         *          *          *            * 192.168.1.8 27741
```

4.3.3. cat count

count 提供对整个集群或单个索引的文档计数的快速访问。

```
curl <host>:<port>/_cat/count?v
```

```
curl -XGET "localhost:19200/_cat/count?v"
count deleted
1     0
```

或者也可以对单个索引使用

```
curl <host>:<port>/_cat/count/<index name>?v
```

4.3.4. cat health

health用于查看集群健康状态。

```
curl <host>:<port>/_cat/health?v
```

```
curl -XGET "localhost:19200/_cat/health?v"
epoch      timestamp cluster                          status master.total master.jeopardy
node.total node.jeopardy node.decommissioning shards shards.under_replica shards.corrupted
replica.corrupted active_shards_percent
1651892436 03:00:36  309a3d000ef049b6a9b4c2d126cec288 green             3               0
5             0                    0      5                    0               0
0               100.0%
```

4.3.5. cat indices

indices 命令提供每个索引的信息。

curl <host>:<port>/_cat/indices?v

```
curl -XGET "localhost:19200/_cat/indices?v"
epoch      timestamp index   uuid                             pri rep docs.count docs.deleted
store.size
1651892887 03:08:07  twitter d244a3665d5347abb203ef95cb0cc17e   5   3          1            0
0.023359298706054688MB
```

我们可以快速知道有多少分片组成一个索引、文档的数量、已删除的文档和存储大小等信息。

4.3.5.1. Primaries

默认情况下，索引统计信息将显示索引的所有分片，包括副本。 可以提供 pri 标志以查看相关统计信息。

4.3.5.2. Examples

```
curl -X GET "localhost:19200/_cat/indices?v&s=docs.count:desc&pretty"

curl -X GET "localhost:19200/_cat/indices/twitter?pri&v&h=index,pri,rep,docs.count,mt&pretty"
```

4.3.6. cat master

master显示主节点 ID、绑定IP地址和节点名称。

curl <host>:<port>/_cat/master?v

```
curl -X GET "localhost:19200/_cat/master?v&pretty"
epoch      timestamp active.master.host active.master.port master.group
1651894131 03:28:51  192.168.1.8        26711
192.168.1.8:26731,192.168.1.8:26721,192.168.1.8:26711
```

4.3.7. cat nodes

nodes 命令显示集群信息。

```
curl <host>:<port>/_cat/nodes?v
```

```
curl -X GET "localhost:19200/_cat/nodes?v&pretty"
host        port  shards heap.percent doc.count store.size
192.168.1.8 27731 6                              0      26232
192.168.1.8 27741 6                              3      34151
192.168.1.8 27711 6                              3      34151
192.168.1.8 27721 6                              3      34151
192.168.1.8 27751 6                              0      26232
```

4.3.8. cat thread pool

thread_pool 命令显示每个节点的集群范围线程池统计信息。

```
curl <host>:<port>/_cat/thread_pool?v
```

```
curl -X GET "localhost:19200/_cat/thread_pool?pretty"
192.168.1.8 27741 abort_merge 0 0 0
192.168.1.8 27741 commit      0 0 0
192.168.1.8 27741 generic     0 0 0
192.168.1.8 27741 listener    0 0 0
192.168.1.8 27741 merge       0 0 0
192.168.1.8 27741 refresh     0 0 0
192.168.1.8 27741 warmer      0 0 0
192.168.1.8 27751 abort_merge 0 0 0
192.168.1.8 27751 commit      0 0 0
192.168.1.8 27751 generic     0 0 0
192.168.1.8 27751 listener    0 0 0
192.168.1.8 27751 merge       0 0 0
192.168.1.8 27751 refresh     0 0 0
192.168.1.8 27751 warmer      0 0 0
192.168.1.8 27721 abort_merge 0 0 0
192.168.1.8 27721 commit      0 0 0
192.168.1.8 27721 generic     0 0 0
192.168.1.8 27721 listener    0 0 0
192.168.1.8 27721 merge       0 0 0
192.168.1.8 27721 refresh     0 0 0
192.168.1.8 27721 warmer      0 0 0
192.168.1.8 27731 abort_merge 0 0 0
192.168.1.8 27731 commit      0 0 0
192.168.1.8 27731 generic     0 0 0
192.168.1.8 27731 listener    0 0 0
192.168.1.8 27731 merge       0 0 0
192.168.1.8 27731 refresh     0 0 0
192.168.1.8 27731 warmer      0 0 0
192.168.1.8 27711 abort_merge 0 0 0
192.168.1.8 27711 commit      0 0 0
192.168.1.8 27711 generic     0 0 0
192.168.1.8 27711 listener    0 0 0
192.168.1.8 27711 merge       0 0 0
192.168.1.8 27711 refresh     0 0 0
192.168.1.8 27711 warmer      0 0 0
```

第三列为线程池名称，后面三列显示每个线程池的active、queue、rejected统计信息。

cat thread pool API 接受 thread_pool_patterns URL 参数，用于指定以逗号分隔的正则表达式列表以匹配线程池名称。

```
curl -X GET
"localhost:19200/_cat/thread_pool/generic?v&h=name,active,rejected,completed&pretty"
```

4.3.9. cat shards

shards 命令返回哪些节点包含哪些分片等信息。shards命令会告诉你它是主节点还是副本节点、文档数、它在磁盘上占用的字节数以及它所在的节点。

```
curl <host>:<port>/_cat/shards?v
```

```
curl -X GET "localhost:19200/_cat/shards?pretty"
twitter  d244a3665d5347abb203ef95cb0cc17e#2 r 192.168.1.8:27751 0 4372
twitter  d244a3665d5347abb203ef95cb0cc17e#3 p 192.168.1.8:27751 0 4372
twitter  d244a3665d5347abb203ef95cb0cc17e#4 r 192.168.1.8:27751 0 4372
twitter  d244a3665d5347abb203ef95cb0cc17e#2 r 192.168.1.8:27711 0 4372
twitter  d244a3665d5347abb203ef95cb0cc17e#2 p 192.168.1.8:27731 0 4372
twitter  d244a3665d5347abb203ef95cb0cc17e#1 r 192.168.1.8:27741 0 4372
twitter  d244a3665d5347abb203ef95cb0cc17e#0 r 192.168.1.8:27721 2 9647
twitter  d244a3665d5347abb203ef95cb0cc17e#1 r 192.168.1.8:27731 0 4372
twitter  d244a3665d5347abb203ef95cb0cc17e#4 r 192.168.1.8:27741 0 4372
twitter  d244a3665d5347abb203ef95cb0cc17e#4 p 192.168.1.8:27721 0 4372
twitter  d244a3665d5347abb203ef95cb0cc17e#1 p 192.168.1.8:27711 0 4372
twitter  d244a3665d5347abb203ef95cb0cc17e#3 r 192.168.1.8:27731 0 4372
twitter  d244a3665d5347abb203ef95cb0cc17e#3 r 192.168.1.8:27721 0 4372
twitter  d244a3665d5347abb203ef95cb0cc17e#0 p 192.168.1.8:27741 2 9647
twitter  d244a3665d5347abb203ef95cb0cc17e#0 r 192.168.1.8:27711 2 9647
twitter2 7f6d9b501e5f4cc487facdb3ce240694#1 r 192.168.1.8:27741 0 4372
twitter2 7f6d9b501e5f4cc487facdb3ce240694#0 p 192.168.1.8:27741 1 7016
twitter2 7f6d9b501e5f4cc487facdb3ce240694#4 r 192.168.1.8:27741 0 4372
twitter2 7f6d9b501e5f4cc487facdb3ce240694#4 r 192.168.1.8:27751 0 4372
twitter2 7f6d9b501e5f4cc487facdb3ce240694#2 r 192.168.1.8:27751 0 4372
twitter2 7f6d9b501e5f4cc487facdb3ce240694#3 p 192.168.1.8:27751 0 4372
twitter2 7f6d9b501e5f4cc487facdb3ce240694#0 r 192.168.1.8:27711 1 7016
twitter2 7f6d9b501e5f4cc487facdb3ce240694#2 p 192.168.1.8:27731 0 4372
twitter2 7f6d9b501e5f4cc487facdb3ce240694#1 r 192.168.1.8:27731 0 4372
twitter2 7f6d9b501e5f4cc487facdb3ce240694#1 p 192.168.1.8:27711 0 4372
twitter2 7f6d9b501e5f4cc487facdb3ce240694#2 r 192.168.1.8:27711 0 4372
twitter2 7f6d9b501e5f4cc487facdb3ce240694#3 r 192.168.1.8:27731 0 4372
twitter2 7f6d9b501e5f4cc487facdb3ce240694#0 r 192.168.1.8:27721 1 7016
twitter2 7f6d9b501e5f4cc487facdb3ce240694#3 r 192.168.1.8:27721 0 4372
twitter2 7f6d9b501e5f4cc487facdb3ce240694#4 p 192.168.1.8:27721 0 4372
```

4.3.9.1. Index pattern

如果有很多分片，您可能希望限制输出中显示的索引，可以使用grep执行此操作，也可以通过在末尾提供索引模式来节省一些带宽。

```
curl <host>:<port>/_cat/shards/twitt*
```

4.3.10. cat templates

templates 命令提供有关现有模板的信息。

```
curl <host>:<port>/_cat/templates?v&s=name
```

4.4. ClusterAPI

4.4.1. Cluster Health

Cluster Health API 允许获得关于集群健康的状态。

GET _cluster/health

```
curl -X GET "localhost:19200/_cluster/health?pretty"
{
  "cluster_name" : "d1a1cb4a025a4c7b89235909cb4c5717",
  "status" : "green",
  "timed_out" : false,
  "number_of_nodes" : 8,
  "number_of_data_nodes" : 5,
  "active_primary_shards" : 0,
  "active_shards" : 0,
  "relocating_shards" : 0,
  "initializing_shards" : 0,
  "unassigned_shards" : 0,
  "delayed_unassigned_shards" : 0,
  "number_of_pending_tasks" : 0,
  "number_of_in_flight_fetch" : 0,
  "task_max_waiting_in_queue_millis" : 0,
  "active_shards_percent_as_number" : 100.0
}
```

这个API也可以这样执行

GET /_cluster/health/test1,test2

集群健康状态为：绿色、黄色或红色。 在分片级别，红色状态表示集群中未分配特定分片，黄色表示已分配主分片但未分配副本，绿色表示已分配所有分片。 索引级别状态由最差的分片状态控制。 集群状态由最差的索引状态控制。

4.4.1.1. Request Parameters

Cluster health API 接受以下请求参数：

wait_for_status: green、yellow或red中的一种。将等待（直到提供的超时）直到集群的状态更改为提供的或更好的状态，(green > yellow > red)。默认情况下，不会等待任何状态。

wait_for_no_relocating_shards: 一个布尔值，它控制是否等待（直到提供的超时）集群中没有重定位的分片。默认为 false，这意味着它不会等待重新定位分片。

wait_for_no_initializing_shards: 一个布尔值，它控制是否等待（直到提供的超时）集群中没有初始化的分片。默认为 false，这意味着它不会等待初始化分片。

wait_for_active_shards: 一个数字，控制等待多少活动分片，等待集群中的所有分片都处于活动状态。默认为 0。

wait_for_nodes: 请求会一直等待，直到指定数量的 N 个节点可用。它还接受 >=N、 N、>N 和 <N。或者可以使用 ge(N)、le(N)、gt(N) 和 lt(N) 表示法。

wait_for_events: 可以是immediate、urgent、high、normal、low、languid。等到所有当前排队的具有给

定优先级的事件都被处理。

timeout：一个基于时间的参数，如果提供了其中一个 wait_for_XXX，则控制等待多长时间。默认为 30 秒。

master_timeout：一个基于时间的参数，控制在 master 尚未被发现或断开连接时等待多长时间。如果未提供，则使用与 timeout 相同的值。

local：如果为 true，则返回本地节点信息，并且不提供来自主节点的状态。默认值：false。

```
GET /_cluster/health?wait_for_status=yellow&timeout=50s
```

4.5. DocumentAPI

4.5.1. Reading and Writing documents

4.5.1.1. Introduction

Scope中的每个索引都会进行分片，每个分片又都会有多个副本，这些副本称为replication group，在添加或删除文档时这些副本也必须保持同步，否则在数据读取时就会出现数据紊乱，保持分片副本的同步并从中提供读取的过程就是我们所说的data replication model。

Scope的数据复制模型基于 主-备模型，在这个模型下，分片分为主分片和副本分片，主分片是所有索引操作的主要入口点，它负责验证并确保所有操作是正确的，一旦主分片接受了索引操作，主分片在索引操作执行成功后还要负责将操作复制到其他副本。

4.5.1.2. Basic write model

Scope中的每个索引操作首先通过路由解析到replication group，这一操作通常基于文档ID，一旦replication group被确定后，索引操作将在内部转发到replication group的当前主分片上，主分片将负责验证操作并将操作转发到其他副本。由于副本可以离线，因此不需要将主分片复制到所有副本，Scope会维护一个应该接收操作的分片副本列表，这个列表称为同步副本并由主节点维护。顾名思义，这些是“好”分片副本的集合。

主分片遵循以下基本流程：

1. 验证输入操作并在结构无效时拒绝它（例如：想要一个数字结果给了一个对象）

2. 先在本地执行操作，例如索引或删除相关文档，如果执行出错时也将拒绝（例如：关键字值太长，无法在Lucene中进行索引）

3. 将操作转发到当前同步副本集中的每个副本。如果有多个副本，则并行执行该操作

4. 一旦所有副本成功执行了操作并响应给主服务器，主服务器就会确认成功完成对客户端的请求

4.5.1.3. Failure handling

在索引的过程可能会出现各种各样的异常情况，例如：1. 磁盘损坏；2. 节点相互断开连接；3. 由于配置错误导致复制副本上的操作失败，尽管它在主服务器上操作成功，等等。虽然这些问题并不一定常见，但是开发者还是有必要作出相应的预案。 在主分片本身发生故障的情况下，托管主分片的节点将向Master发送有关它的消息，此时索引操作将等待（默认情况下最多1分钟），以便Master将其中一个副本提升为新主分片，然后，该操作将被转发到新的主分片处理。 请注意，Master还会监控节点的运行状况，并可能决定主动对主分片进行降级（这通常是由于网络问题导致的）。一旦在主分片上成功执行了操作，主分片就必须处理在副本

上执行操作时存在的潜在故障，这些潜在的故障可能是由副本上的实际故障或由于网络问题导致操作无法到达副本（或阻止副本响应）引起的。所有这些都具有相同的最终结果：同步副本集中的一部分副本错过了即将被确认的操作。此时，主分片向Master发送消息，请求从同步副本集中删除有问题的分片。只有在Master确认删除了分片后，主分片才会确认操作。注意，Master还将指示另一个节点开始构建新的分片副本，以便将系统还原到正常状态。 在将操作转发到副本时，主分片将使用副本来验证它仍然是活动主分片。如果主分片由于网络原因（或长GC）而被分离，它依然可能会在被降级之前继续处理传入的索引操作，此时副本将拒绝来自旧主分片的操作。主分片收到副本的拒绝请求后会请求Master节点，Master会告诉旧的主分片你已经被替换掉，然后操作会被路由到新的主分片。

4.5.1.4. what happens if there are no replicas

由于索引的配置原因或者所有副本都已失效，在这种情况下，会发生主分片没有副本。此时，主分片处理操作而没有任何外部验证，这可能看起来有问题。另一方面，主分片本身不能使其他分片失效，但可以请求Master代表它执行此操作。这意味着Master知道主分片是唯一的好副本。因此，我们保证Master不会将任何其他（过时的）分片副本提升为新的主分片，并且任何索引到主分片的操作都不会丢失。当然，由于此时我们只使用单个数据副本运行，因此物理硬件问题可能导致数据丢失。

4.5.1.5. Basic read model

Scope中的读取操作，可以是按照ID查找这种非常轻量级的操作，也可以是具有复杂聚合的大量搜索请求，这些聚合操作会占用非常大的CPU算力。 主-备模型的优点之一是它使所有分片副本保持一致（除了飞行中的操作）。基于此，单个同步的副本足以处理读取请求。 当节点收到读取请求时，该节点负责将其转发到保存相关分片的节点，整理响应并对客户端做出响应。此时，我们将该节点称为该请求的协调节点，该节点的基本工作流程如下：

1. 对读取请求进行解析，然后将请求分发到不同分片上。请注意，由于大多数搜索请求将被发送到一个或多个索引，因此它们通常需要从多个分片中读取，每个分片代表数据的不同子集。

2. 从replication group中选择每个相关分片的可用副本，可以是主分片或副本。默认情况下，Scope将简单地在分片副本之间循环。

3. 将分片级读取请求发送到所选副本。

4. 整合请求结果并给客户端作出响应，注意，在通过ID查找的情况下，只有一个分片是相关的，并且可以跳过此步骤（即不需要整合请求结果，用过MyCat的读者，可能会发现这个步骤的作用和MyCat比较类似）。

4.5.1.6. Shard failures

当分片无法响应读取请求时，协调节点将从同一复制组中选择另一个副本，并将分片级别搜索请求发送到该副本，不过要是重复失败可能导致没有可用的分片副本。

4.5.1.7. A few simple implications

由于读取和写入请求可以同时执行，因此这两个基本流程彼此交互，有一些固有的含义：

Efficient reads

在正常操作下，对每个相关的replication group执行一次读取操作。只有在失败的情况下，才会对同一个分片的多个副本执行相同的搜索。

Read unacknowledged

由于主分片首先在本地进行索引，然后将操作发给副本去执行，因此并发读取可能在确认之前就已经看到了更改的数据。

Two copies by default

此模型可以容错，同时只保留两个数据副本。这与基于法定数量的系统形成对比（容错的最小副本数为3）。

4.5.1.8. Failures

在操作失败的情况下，以下是可能的：

1. Efficient reads

由于主分片在每个操作期间等待同步副本集中的所有副本，因此单个分片操作速度慢可能会降低整个replication group的速度。这是我们为上述阅读效率付出的代价，同时，单个慢速分片也会降低已经路由到它的搜索请求。

2. Read unacknowledged

被隔离的主分片可以执行写操作，但是却无法被确认，这是因为隔离的主分片只有在向其副本发送请求或向Master发送请求时才会意识到它是隔离的。此时，操作已经索引到主分片中，并且可以通过并发读取来读取。Scope通过每秒ping一次Master（默认情况下）并在没有Master的情况下拒绝索引操作来减轻这种风险。

4.5.2. Index API

索引API在特定索引中添加或更新类型化JSON文档，使其可以被搜索到。下面的例子将JSON文档插入到“twitter”索引中，类型名为_doc, id为1:

```
PUT twitter/_doc/1 { "user" : "kimchy", "post_date" : "2009-11-15T14:12:12", "message" : "trying out Scope" }
```

```
curl -X POST "localhost:9200/term_query_index/default_type_/2?pretty" -H 'Content-Type:
application/json' -d'
{
   "name": "Charlie Paul",
   "hobby": "play football"
}'
{
  "_index" : "term_query_index",
  "_type" : "default_type_",
  "_id" : "2",
  "_version" : -1,
  "result" : "created",
  "_shards" : {
    "total" : 1,
    "successful" : 1,
    "failed" : 0
  }
}
```

_shards报头提供了关于索引操作的信息:

total：指示索引操作应该在多少个分片副本(主分片和副本分片)上执行。

successful：索引操作成功的分片个数。

failed：当索引操作在副本分片上失败时，包含相关错误的数组。

4.5.2.1. Automatic Index Creation

如果索引不存在，索引操作会自动创建一个索引，并应用配置的任何索引模板。索引操作还为指定类型创建动态类型的mapping(如果不存在)。默认情况下，如果需要，新的字段和对象将自动添加到指定类型的mapping定义中。

自动索引创建由控制action.auto_create_index设置。该设置默认为true，意味着总是自动创建索引。加上+或-也可以显式地允许和禁止它。最后，可以通过将此设置更改为false来完全禁用它。

```
PUT _cluster/settings { "persistent": { "action.auto_create_index": "twitter,index10,-index1*,+ind*" } }

PUT _cluster/settings { "persistent": { "action.auto_create_index": "false" } }

PUT _cluster/settings { "persistent": { "action.auto_create_index": "true" } }
```

4.5.2.2. Automatic ID Generation

不需要指定id就可以执行索引操作。在这种情况下，将自动生成一个id。

```
POST twitter/_doc/ { "user" : "kimchy", "post_date" : "2009-11-15T14:12:12", "message" : "trying out Scope" }
```

4.5.2.3. Routing

默认情况下，分片放置(或路由)是通过对文档id值的哈希值来控制的。为了更明确的控制，输入到路由器使用的哈希函数的值可以使用route参数直接在每个操作的基础上指定。

```
POST twitter/_doc?routing=kimchy { "user" : "kimchy", "post_date" : "2009-11-15T14:12:12", "message" : "trying out Scope" }
```

在上面的例子中，“_doc”文档根据提供的route参数“kimchy”被路由到一个分片。

在设置显式mapping时，可以使用_routing字段指导索引操作从文档本身提取路由值。

4.5.2.4. Distributed

索引操作将根据主分片的route定向到主分片，并在包含该分片的实际节点上执行。主分片完成操作后，如果需要，这些操作更新将分发到适用的副本上。

4.5.2.5. Wait For Active Shards

索引操作可以配置为等待一定数量的活跃的分片副本后再进行操作。如果所需要的活跃的分片副本数量不可用，则写操作必须等待并重试，直到所需要的分片副本启动或超时。默认情况下，写操作只等待主分片激活才能继续(即wait_for_active_shards=1)。通过设置index.write.wait_for_active_shards，可以在索引设置中动态地覆盖这个默认值。有效值为all或任意正整数，最大值为索引中每个分片配置的副本总数(即number_of_replicas + 1)。设置为负数或大于分片副本数量的数字将会抛出错误。

4.5.2.6. Timeout

timeout参数可用于显式指定等待的时间。当执行索引操作时，指定执行索引操作的主分片可能不可用。出现

这种情况的一些原因可能是主分片目前正在从网关恢复或正在进行重新定位。默认情况下，索引操作将在主分片上等待最多1分钟，直到失败并返回错误。

```
PUT twitter/_doc/1?timeout=5m { "user" : "kimchy", "post_date" : "2009-11-15T14:12:12", "message" : "trying out Scope" }
```

4.5.2.7. Versioning

每个索引文档都有一个版本号。默认情况下，使用内部版本控制，从1开始，随着每次更新和删除而递增。版本号可以设置为一个外部值(例如，如果在数据库中维护)。要启用此功能，应该将version_type设置为external。提供的值必须是一个数值，大于或等于0，且小于9.2e+18左右。

当使用external时，系统检查传递给索引请求的版本号是否大于当前存储的文档的版本号。如果为true，则对文档进行索引，并使用新版本号。如果提供的值小于或等于存储文档的版本号，则会发生版本冲突，索引操作将失败。

```
PUT twitter/_doc/1?version=2&version_type=external { "message" : "Scope now has versioning support, double cool!" }
```

4.5.2.8. Version types

internal:只有当给定的版本与存储的文档的版本相同时，才对文档进行索引。

external或external_gt:只有当给定版本严格高于存储文档的版本，或者没有现有文档时，才对文档建立索引。给定的版本将作为新版本使用，并与新文档一起存储。提供的版本必须是非负的。

external_gte:只有当给定版本等于或高于存储文档的版本时，才对文档建立索引。如果没有现有的文档，操作也会成功。给定的版本将作为新版本使用，并与新文档一起存储。提供的版本必须是非负的。

注意:external_gte用于特殊情况，应该谨慎使用。如果使用不当，可能会导致数据丢失。

4.5.3. Delete API

删除API允许根据id从特定索引中删除类型化JSON文档。下面的示例从名为twitter的索引中删除JSON文档，该文档的类型为_doc, id为1:

```
DELETE /twitter/_doc/1
```

```
curl -X DELETE "localhost:9200/term_query_index/?pretty"
{
  "acknowledged" : true
}
```

4.5.3.1. Versioning

索引中的每个文档都被标记了版本，任何的操作如更新删除等，都会导致文档版本的递增，因此，当删除文档时，也可以指定文档的版本号，确保删掉的文档是我们想删掉的文档。已经删除文档的版本号在删除后仍可短时间使用，以便控制并发操作，可以通过index.gc_deletes来设置已经删除文档的版本号的保存时间，

默认为60秒。

4.5.3.2. Routing

如果索引的时候指定了routing参数，那么删除的时候也要指定对应的routing参数，否则删除不了相应的文档。

```
DELETE /twitter/_doc/1?routing=kimchy
```

上例将根据kimchy进行路由删除id为1的文档。请注意，在没有正确路由的情况下将无法正确删除文档。

当在mapping中设置_routing但没有指定routing参数时，delete api会抛RoutingMissingException异常并且拒绝本次请求。

4.5.3.3. Automatic index creation

如果使用了外部版本控制，则删除操作会自动创建一个之前尚未创建过的索引,如果以前未创建过特定类型，还会自动为特定类型创建动态类型mapping。

4.5.3.4. Distributed

delete请求会通过hash路由到某个分片上，然后被重定向到该id组中的主分片上，并复制到该id组中的副本分片上。之后在所有相关分片上执行删除操作。

4.5.3.5. Wait For Active Shards

当执行删除操作时，可以设置 wait_for_active_shards参数要求活跃分片数达到某个值的时候才能执行删除操作。

4.5.3.6. Refresh

设置此请求所做的更改到搜索可见的时间。

4.5.3.7. Timeout

执行删除操作时，执行删除操作的主分片可能不可用。这可能是因为主分片正在恢复中或者主分片正在进行迁移。默认情况下，删除操作会等待一分钟，如果主分片还是不可用就会报错。你可以在delete API 中通过timeout参数显式指定超时时间。下面例子把timeout设置为5分钟：

```
DELETE /twitter/_doc/1?timeout=5m
```

4.5.4. Update API

Update API从索引中获取文档，并对结果进行索引。它使用版本控制来确保在“get”和“reindex”期间没有发生任何更新。

请注意，此操作仍然意味着对文档进行完整的重新索引，它只是删除了一些网络往返并减少了get和索引之间的版本冲突。需要启用 _source字段才能使用此功能。

4.5.4.1. Updates with a partial document

Update API还支持传递部分文档，该部分文档将被合并到现有文档中（简单的递归合并、对象的内部合并、

替换核心“键/值”和数组）。

```
POST test/_doc/1/_update { "doc" : { "name" : "new_name" } }
```

```
curl -XPOST "localhost:9200/test/_doc/1/_update?pretty" -H 'Content-Type: application/json' -d
'
{
     "doc" : {
         "counter" : 2
     }
}'
{
  "_index" : "test",
  "_type" : "default_type_",
  "_id" : "1",
  "_version" : -1,
  "result" : "updated",
  "_shards" : {
    "total" : 3,
    "successful" : 3,
    "failed" : 0
  }
}
```

4.5.5. Bulk API

Bulk API能够在一次API调用完成多个index/delete操作，这样可以大幅提高操作索引的速度

Bulk的端点是/_bulk，它需要遵循以下指定的JSON结构

```
action_and_meta_data\n

optional_source\n

action_and_meta_data\n

optional_source\n

….

action_and_meta_data\n

optional_source\n
```

可以做的操作是：index，create，delete和update。如果提供一个文本文件作为curl的输入一定要用—data-binary标志去取代简单的-d。

```
$ cat requests { "index" : { "_index" : "test", "_type" : "_doc", "_id" : "1" } } { "field1" : "value1" } $ curl -s -H "Content-Type: application/x-ndjson" -XPOST localhost:9200/_bulk --data-binary "@requests"; echo {"took":7, "errors": false, "items":[{"index": {"_index":"test","_type":"_doc","_id":"1","_version":1,"result":"created","forced_refresh":false}}]}
```

因为这个格式使用文字\n作为限制符，请确保JSON操作和资源不会被大量的打印。下面是一个正确的bulk的

例子。

```
POST _bulk { "index" : { "_index" : "test", "_type" : "_doc", "_id" : "1" } } { "field1" : "value1" } { "delete" : { "_index" : "test", "_type" : "_doc", "_id" : "2" } } { "create" : { "_index" : "test", "_type" : "_doc", "_id" : "3" } } { "field1" : "value3" } { "update" : {"_id" : "1", "_type" : "_doc", "_index" : "test"} } { "doc" : {"field2" : "value2"} }
```

```
curl -X POST "localhost:9200/bulk_index/_bulk?pretty" -H 'Content-Type: application/x-ndjson'
--data-binary '
{ "index" : {"_index" : "bulk_index2","_type" : "default_type_", "_id" : "bulk2" } }
{ "name":"blue","age":234} }
{ "delete" : {"_type" : "default_type_","_id" : "a" } }
{ "create" : {"_type" : "default_type_","_id" : "bulk2" } }
{ "songname" : "When You Say Nothing At All","singer":"Alison Krauss"}
{ "update" : {"_id" : "changchangdeid", "_index" : "bulk_index","_type" : "default_type_"} }
{ "doc" : {"publish" : "2006-02-13"}}'
{
  "took" : 0,
  "errors" : false,
  "items" : [
    {
      "index" : {
        "_index" : "bulk_index2",
        "_type" : "default_type_",
        "_id" : "bulk2",
        "_version" : -1,
        "result" : "created",
        "_shards" : {
          "total" : 3,
          "successful" : 3,
          "failed" : 0
        },
        "status" : 201
      }
    },
    {
      "delete" : {
        "_index" : "bulk_index",
        "_type" : "default_type_",
        "_id" : "a",
        "_version" : -1,
        "result" : "deleted",
        "_shards" : {
          "total" : 3,
          "successful" : 3,
          "failed" : 0
        },
        "status" : 200
      }
    },
    {
      "create" : {
        "_index" : "bulk_index",
        "_type" : "default_type_",
        "_id" : "bulk2",
        "_version" : -1,
        "result" : "created",
        "_shards" : {
          "total" : 3,
          "successful" : 3,
          "failed" : 0
        },
        "status" : 201
      }
    },
    {
      "update" : {
        "_index" : "bulk_index",
        "_type" : "default_type_",
        "_id" : "changchangdeid",
        "_version" : -1,
        "result" : "updated",
        "_shards" : {
          "total" : 3,
          "successful" : 3,
          "failed" : 0
        },
        "status" : 200
      }
    }
  ]
}
```

端点有/_bulk，/{index}/_bulk和{index}/{type}/_bulk。

bulk操作的响应是一个大型的JSON结构，每个操作的结果都是按照与请求中出现的操作相同的顺序展现的。单个操作的失败不会影响其余操作。

单个bulk调用含多少操作没有一个“正确”的数量，应该在运行环境中尝试不同的的配置去找到最优操作数的大小。

4.5.5.1. Routing

每个bulk单元可以使用_routing包含路由值。

4.5.5.2. Refresh

控制此请求所做的更改何时对搜索可见。

4.5.5.3. Update

当使用update操作时，retry_on_conflict可以被用作一个字段去指定重试多少次以防版本冲突。

update支持以下选项：doc, upsert, doc_as_upsert, script, params, lang, _source。

```
POST _bulk { "update" : {"_id" : "1", "_type" : "_doc", "_index" : "index1", "retry_on_conflict" : 3} } { "doc" : {"field" : "value"} } { "update" : { "_id" : "0", "_type" : "_doc", "_index" : "index1", "retry_on_conflict" : 3} } { "script" : { "source": "ctx._source.counter += params.param1", "lang" : "painless", "params" : {"param1" : 1}}, "upsert" : {"counter" : 1}} { "update" : {"_id" : "2", "_type" : "_doc", "_index" : "index1", "retry_on_conflict" : 3} } { "doc" : {"field" : "value"}, "doc_as_upsert" : true } { "update" : {"_id" : "3", "_type" : "_doc", "_index" : "index1", "_source" : true} } { "doc" : {"field" : "value"} } { "update" : {"_id" : "4", "_type" : "_doc", "_index" : "index1"} } { "doc" : {"field" : "value"}, "_source": true}
```

4.5.6. ?refresh

Index、Update、Delete和Bulk API均支持设置refresh，以控制此请求所做的更改何时对搜索可见。以下是允许的值：

Empty string or true 操作发生后立即刷新相关的主分片和副本分片（不是整个索引），以便更新的文档立即出现在搜索结果中。这只能在仔细考虑并验证它不会导致性能不佳（从索引和搜索的角度来看）之后进行。

false（默认） 不采取与刷新相关的操作。此请求所做的更改将在请求返回后的某个时间点可见。

```
PUT /test/_doc/1?refresh {"test": "test"}

PUT /test/_doc/2?refresh=true {"test": "test"}

PUT /test/_doc/4?refresh=false {"test": "test"}
```

```
curl -X POST "localhost:9200/term_query_index/default_type_/3?refresh=true" -H 'Content-Type:
application/json' -d'
{
   "name": "Charlie",
   "hobby": "play football"
}'
{"_index":"term_query_index","_type":"default_type_","_id":"3","_version":-
1,"result":"created","_shards":{"total":1,"successful":1,"failed":0}}
```

4.5.7. Delete By Query API

使用_delete_by_query对每个查询匹配的文档执行删除操作

POST twitter/_delete_by_query

```
curl -X POST "localhost:19200/index/_delete_by_query?pretty" -H 'Content-Type:
application/json' -d'
> {
>   "query": {
>     "match": {
>       "user": "kimchy"
>     }
>   }
> }'
{
  "took" : 121,
  "timed_out" : false,
  "total" : 1,
  "updated" : 0,
  "deleted" : 1,
  "batches" : 1,
  "version_conflicts" : 0,
  "noops" : 0,
  "retries" : {
    "bulk" : 0,
    "search" : 0
  },
  "throttled_millis" : 0,
  "requests_per_second" : -1.0,
  "throttled_until_millis" : 0,
  "failures" : [ ]
}
```

_delete_by_query 在启动时获取索引的快照，并使用内部版本控制删除它找到的内容，当版本匹配时，文档被删除。

在 _delete_by_query 执行过程中，会依次执行多个搜索请求，以便找到所有要删除的匹配文档。 每找到一批文档，就会执行相应的批量请求，删除所有这些文档。 如果搜索或批量请求被拒绝，_delete_by_query 将依赖默认策略重试被拒绝的请求（最多 10 次）。 达到最大重试限制会导致 _delete_by_query 中止，并且所有失败都在响应失败中返回。 已执行的删除仍然存在。 换句话说，该过程没有回滚，只是中止。 当第一次失败导致中止时，失败的批量请求返回的所有失败都在 failures 元素中返回； 因此，可能会有相当多的失败实体。

也可以一次删除多个索引和多个类型的文档

POST twitter,twitter1/_delete_by_query

4.5.7.1. Response body

JSON响应如下所示：

```
{
  "took" : 1465,
  "timed_out" : false,
  "total" : 1,
  "deleted" : 0,
  "batches" : 1,
  "version_conflicts" : 0,
  "noops" : 0,
  "retries" : {
    "bulk" : 0,
    "search" : 0
  },
  "throttled_millis" : 0,
  "requests_per_second" : -1.0,
  "throttled_until_millis" : 0,
  "failures" : [ ]
}
```

took 整个操作从开始到结束的毫秒数。

timed_out 如果在通过查询执行更新期间执行的任何请求已超时，则此标志设置为 true。

total 成功处理的文档数。

deleted 成功删除的文档数。

batches 由查询更新拉回的滚动响应数。

version_conflicts 查询更新命中的版本冲突数。

noops 由于用于按查询更新的脚本返回 ctx.op 的 noop 值而被忽略的文档数。

retries 通过查询更新尝试的重试次数。 bulk 是重试批量操作的次数，search 是重试搜索操作的次数。

throttled_millis 请求休眠以符合 requests_per_second 的毫秒数。

requests_per_second 查询更新期间每秒有效执行的请求数。

throttled_until_millis 在_update_by_query 响应中，该字段应始终为零。

failures update by query是使用批处理实现的,任何失败都会导致整个过程中止，但当前批次中的所有失败都会收集到数组中，如果数组是非空的，那么请求会因为这些失败而中止。

4.5.8. Update By Query API

使用_update_by_query对每个查询匹配的文档执行更新操作

```
POST twitter/_update_by_query
```

_update_by_query 在启动时获取索引的快照，并使用内部版本控制索引它找到的内容。 这意味着如果文档在拍摄快照和处理索引请求之间发生变化，您将遇到版本冲突。 当版本匹配时，更新文档并增加版本号。

所有更新和查询失败都会导致 _update_by_query 中止并在响应失败中返回。已执行的更新仍然存在。换句话说，该过程没有回滚，只是中止。当第一次失败导致中止时，失败的批量请求返回的所有失败都在 failures 元素中返回；因此，可能会有相当多的失败实体。

您还可以使用 Query DSL 限制 _update_by_query。 下面的语句这将为用户 kimchy 更新 twitter 索引中

的所有文档：

```
POST twitter/_update_by_query?conflicts=proceed { "query": { "term": { "user": "kimchy" } } }
```

```
curl -XPOST "localhost:19200/index/_update_by_query?pretty" -H 'Content-Type: application/json'
-d '
> {
>   "script": {
>     "source": "ctx._source.age=params.age",
>     "lang": "painless",
>     "params":{
>       "age":32
>     }
>   },
>   "query": {
>     "term": {
>       "user": "kimchy"
>     }
>   }
> }'
{
  "took" : 1465,
  "timed_out" : false,
  "total" : 1,
  "updated" : 1,
  "deleted" : 0,
  "batches" : 1,
  "version_conflicts" : 0,
  "noops" : 0,
  "retries" : {
    "bulk" : 0,
    "search" : 0
  },
  "throttled_millis" : 0,
  "requests_per_second" : -1.0,
  "throttled_until_millis" : 0,
  "failures" : [ ]
}
```

_update_by_query 支持脚本来更新文档。 这将增加所有 kimchy 推文的 likes 字段：

```
POST twitter/_update_by_query { "script": { "source": "ctx._source.likes++", "lang": "painless" }, "query": {
"term": { "user": "kimchy" } } }
```

4.5.8.1. Response body

JSON响应如下所示：

```
{
  "took" : 1465,
  "timed_out" : false,
  "total" : 1,
  "updated" : 1,
  "deleted" : 0,
  "batches" : 1,
  "version_conflicts" : 0,
  "noops" : 0,
  "retries" : {
    "bulk" : 0,
    "search" : 0
  },
  "throttled_millis" : 0,
  "requests_per_second" : -1.0,
  "throttled_until_millis" : 0,
  "failures" : [ ]
}
```

took 整个操作从开始到结束的毫秒数。

timed_out 如果在通过查询执行更新期间执行的任何请求已超时，则此标志设置为 true。

total 成功处理的文档数。

updated 成功更新的文档数。

deleted 成功删除的文档数。

batches 由查询更新拉回的滚动响应数。

version_conflicts 查询更新命中的版本冲突数。

noops 由于用于按查询更新的脚本返回 ctx.op 的 noop 值而被忽略的文档数。

retries 通过查询更新尝试的重试次数。 bulk 是重试批量操作的次数，search 是重试搜索操作的次数。

throttled_millis 请求休眠以符合 requests_per_second 的毫秒数。

requests_per_second 查询更新期间每秒有效执行的请求数。

throttled_until_millis 在_update_by_query 响应中，该字段应始终为零。

failures update by query是使用批处理实现的,任何失败都会导致整个过程中止，但当前批次中的所有失败都会收集到数组中，如果数组是非空的，那么请求会因为这些失败而中止。

4.5.9. Reindex API

_reindex 最基本的形式只是将文档从一个索引复制到另一个索引。 这会将 twitter 索引中的文档复制到 new_twitter 索引中：

```
POST _reindex { "source": { "index": "twitter" }, "dest": { "index": "new_twitter" } }
```

```
curl -X POST "localhost:19200/_reindex?pretty" -H 'Content-Type: application/json' -d'
> {
>   "source": {
>     "index": "my_index2"
>   },
>   "dest": {
>     "index": "new_index"
>   }
> }
> '
{
  "took" : 112,
  "timed_out" : false,
  "total" : 1,
  "updated" : 0,
  "created" : 1,
  "deleted" : 0,
  "batches" : 1,
  "version_conflicts" : 0,
  "noops" : 0,
  "retries" : {
    "bulk" : 0,
    "search" : 0
  },
  "throttled_millis" : 0,
  "requests_per_second" : -1.0,
  "throttled_until_millis" : 0,
  "failures" : [ ]
}
```

就像 _update_by_query 一样，_reindex 获取源索引的快照，但其目标必须是不同的索引，因此不太可能发生版本冲突。省略version_type或将其设置为 internal 将导致 Scope 盲目地将文档转储到目标中，覆盖任何碰巧具有相同类型和 id 的文档：

```
POST _reindex { "source": { "index": "twitter" }, "dest": { "index": "new_twitter", "version_type": "internal" }
}
```

将 version_type 设置为 external 将导致 Scope 保留源中的版本，创建任何丢失的文档，并更新目标索引中版本比源索引中的旧版本的任何文档：

```
POST _reindex { "source": { "index": "twitter" }, "dest": { "index": "new_twitter", "version_type": "external" }
}
```

设置 op_type为create 将导致 _reindex 仅在目标索引中创建缺失的文档。 所有现有文档都会导致版本冲突：

```
POST _reindex { "source": { "index": "twitter" }, "dest": { "index": "new_twitter", "op_type": "create" } }
```

您可以通过向源添加类型或添加查询来限制文档。下面这条语句只会将 kimchy 制作的推文复制到 new_twitter 中：

```
POST _reindex { "source": { "index": "twitter", "type": "_doc", "query": { "term": { "user": "kimchy" } } },
"dest": { "index": "new_twitter" } }
```

index 和 type 都可以是列表，允许您在一个请求中从许多源中复制。下面这条语句将从 twitter 和 blog

索引中复制文档。

```
POST _reindex { "source": { "index": ["twitter", "blog"], "type": ["_doc", "post"] }, "dest": { "index": "all_together", "type": "_doc" } }
```

keeps 将针对每个匹配发送的批量请求的路由设置为匹配的路由。 这是默认值。 discard 将针对每个匹配发送的批量请求的路由设置为 null。 =<some text> 将针对每个匹配发送的批量请求的路由设置为 = 之后的所有文本。 例如，您可以使用以下请求将公司名称为 cat 的源索引中的所有文档复制到路由设置为 cat 的 dest 索引中。

```
POST _reindex { "source": { "index": "source", "query": { "match": { "company": "cat" } } }, "dest": { "index": "dest", "routing": "=cat" } }
```

Reindex 还可以通过指定pipeline来使用 Ingest Node 功能：

```
POST _reindex { "source": { "index": "source" }, "dest": { "index": "dest", "pipeline": "some_ingest_pipeline" } }
```

4.5.9.1. Response body

JSON响应如下所示：

```
{
  "took" : 1465,
  "timed_out" : false,
  "total" : 1,
  "updated" : 1,
  "deleted" : 0,
  "batches" : 1,
  "version_conflicts" : 0,
  "noops" : 0,
  "retries" : {
    "bulk" : 0,
    "search" : 0
  },
  "throttled_millis" : 0,
  "requests_per_second" : -1.0,
  "throttled_until_millis" : 0,
  "failures" : [ ]
}
```

took 整个操作从开始到结束的毫秒数。

timed_out 如果在通过查询执行更新期间执行的任何请求已超时，则此标志设置为 true。

total 成功处理的文档数。

updated 成功更新的文档数。

deleted 成功删除的文档数。

batches 由查询更新拉回的滚动响应数。

version_conflicts 查询更新命中的版本冲突数。

noops 由于用于按查询更新的脚本返回 ctx.op 的 noop 值而被忽略的文档数。

retries 通过查询更新尝试的重试次数。 bulk 是重试批量操作的次数，search 是重试搜索操作的次数。

throttled_millis 请求休眠以符合 requests_per_second 的毫秒数。

requests_per_second 查询更新期间每秒有效执行的请求数。

throttled_until_millis 在_update_by_query 响应中，该字段应始终为零。

failures update by query是使用批处理实现的,任何失败都会导致整个过程中止，但当前批次中的所有失败都会收集到数组中，如果数组是非空的，那么请求会因为这些失败而中止。

4.5.10. Multi Get API

Multi get API 根据索引、类型和 id 返回多个文档。响应包括一个 docs 数组，其中包含与原始multi-get请求相对应的所有获取文档的顺序（如果获取失败，则包含此错误的对象将包含在响应中）。

GET /_mget { "docs" : [{ "_index" : "test", "_type" : "_doc", "_id" : "1" }, { "_index" : "test", "_type" : "_doc", "_id" : "2" }] }

```
curl -X GET "localhost:19200/_mget?pretty" -H 'Content-Type: application/json' -d'
> {
>     "docs" : [
>         {
>             "_index" : "test",
>             "_type" : "_doc",
>             "_id" : "1"
>         },
>         {
>             "_index" : "test",
>             "_type" : "_doc",
>             "_id" : "2"
>         }
>     ]
> }
> '
{
  "docs" : [
    {
      "_index" : "test",
      "_type" : "default_type_",
      "_id" : "2",
      "_version" : 1,
      "_seq_no" : 1,
      "_primary_term" : 1,
      "found" : true,
      "_source" : {
        "test" : "document2"
      }
    },
    {
      "_index" : "test",
      "_type" : "default_type_",
      "_id" : "1",
      "_version" : 0,
      "_seq_no" : 1,
      "_primary_term" : 1,
      "found" : true,
      "_source" : {
        "test" : "document1"
      }
    }
  ]
}
```

mget 也可以用于索引:

```
GET /test/_mget { "docs" : [ { "_type" : "_doc", "_id" : "1" }, { "_type" : "_doc", "_id" : "2" } ] }
```

4.5.10.1. Source Filtering

默认情况下，将为每个文档（如果已存储）返回 _source 字段。 您可以使用 _source 参数仅检索 _source 的一部分（或根本不检索）。 您还可以使用 url 参数 _source、_source_includes 和 _source_excludes 来指定默认值。

```
GET /_mget { "docs" : [ { "_index" : "test", "_type" : "_doc", "_id" : "1", "_source" : false }, { "_index" : "test", "_type" : "_doc", "_id" : "2", "_source" : ["field3", "field4"] }, { "_index" : "test", "_type" : "_doc", "_id" : "3", "_source" : { "include": ["user"], "exclude": ["user.location"] } } ] }
```

4.5.10.2. Fields

可以指定要获取的每个文档检索特定的存储字段，类似于 Get API 的 stored_fields 参数。 例如：

```
GET /_mget { "docs" : [ { "_index" : "test", "_type" : "_doc", "_id" : "1", "stored_fields" : ["field1", "field2"] }, { "_index" : "test", "_type" : "_doc", "_id" : "2", "stored_fields" : ["field3", "field4"] } ] }
```

或者，您可以将查询字符串中的 stored_fields 参数指定为默认应用于所有文档。

```
GET /test/_doc/_mget?stored_fields=field1,field2 { "docs" : [ { "_id" : "1" }, { "_id" : "2", "stored_fields" : ["field3", "field4"] } ] }
```

4.5.10.3. Routing

```
GET /_mget?routing=key1 { "docs" : [ { "_index" : "test", "_type" : "_doc", "_id" : "1", "routing" : "key2" }, { "_index" : "test", "_type" : "_doc", "_id" : "2" } ] }
```

4.6. indices APIs

4.6.1. Create Index

Create Index API用于手动创建索引。Scope中的所有文档都存储在一个或另一个索引中。

最基本的命令如下：

```
curl -X PUT "localhost:9200/twitter?pretty"
```

上述命令创建了一个名为twitter的索引。

```
curl -X PUT "localhost:9200/twitter?pretty"
{
  "acknowledged" : true,
  "shards_acknowledged" : true,
  "index" : "twitter"
}
```

4.6.1.1. Index name limitations

索引的命名有几个限制：

1. 仅小写

2. 不能包含 \、/、*、?、"、<、>、|、` `（空格字符）、,、#

3. 不能以 -、_、+ 开头

4. 不可能是 . 或者 ..

5. 不能超过255个字节

4.6.1.2. Index Settings

创建的每个索引都可以具有与之关联的特定设置：

```
PUT twitter { "settings" : { "index" : { "number_of_shards" : 3, "number_of_replicas" : 2 } } }
```

或者

```
PUT twitter { "settings" : { "number_of_shards" : 3, "number_of_replicas" : 2 } }
```

4.6.1.3. Mappings

Create Index API还允许提供Mapping：

```
PUT test { "settings" : { "number_of_shards" : 1 }, "mappings" : { "_doc" : { "properties" : { "field1" : { "type" : "text" } } } } }
```

4.6.1.4. Aliases

Create Index API还允许提供一组别名：

```
PUT test { "aliases" : { "alias_1" : {}, "alias_2" : { "filter" : { "term" : {"user" : "kimchy" } }, "routing" : "kimchy" } } }
```

4.6.1.5. Wait For Active Shards

默认情况下，索引创建只会在每个分片的主副本已启动或请求超时时才向客户端返回响应

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

acknowledged表明是否在集群中成功创建索引，而 shards_acknowledged 指示是否在超时之前为索引中的每个分片启动了必要数量的分片副本。请注意，即使索引创建成功，acknowledged 或 shards_acknowledged 也仍然可能为 false。这些值仅表示操作是否在超时之前完成。如果 acknowledged 为 false，那么我们在使用新创建的索引更新集群状态之前就超时了，但它可能很快就会被创建。如果 shards_acknowledged 为 false，那么即使集群状态已成功更新以反映新创建的索引（即，acknowledged = true），也会在启动所需数量的分片之前超时（默认情况下只是主分片）。

我们可以通过索引设置 index.write.wait_for_active_shards 更改默认只等待主分片启动（注意更改此设置也会影响所有后续写入操作的 wait_for_active_shards 值）

```
PUT test { "settings": { "index.write.wait_for_active_shards": "2" } }
```

或者也可以这样

```
PUT test?wait_for_active_shards=2
```

4.6.2. Delete Index

Delete Index API 允许删除现有索引。

```
DELETE /twitter
```

```
curl -X DELETE "localhost:9200/test/?pretty"
{
  "acknowledged" : true
}
```

上面的示例删除了一个名为 test 的索引。

4.6.3. Get Index

get index API 允许检索有关一个或多个索引的信息。

```
GET /twitter
```

上面的示例获取名为 twitter 的索引的信息。

```
curl -X GET "localhost:19200/test?pretty"
{
  "test" : {
    "aliases" : { },
    "mappings" : {
      "default_type_" : {
        "properties" : { }
      }
    },
    "settings" : {
      "index" : {
        "number_of_shards" : "5",
        "number_of_replicas" : "3",
        "creation_date" : "2022-05-09 21:04:42"
      }
    }
  }
}
```

expand_wildcards: 控制通配符索引表达式可以扩展到哪种具体索引。 如果指定了 open，则通配符表达式将扩展为仅打开的索引。 如果指定了closed，则通配符表达式仅扩展为关闭索引。 也可以指定两个值（open，closed）以扩展到所有索引。

如果指定为null，则通配符扩展将被禁用。 如果指定了 all，通配符表达式将扩展到所有索引（这相当于指定 open,closed）

4.6.4. Indices Exists

检查index是否存在，HTTP状态码返回200表示存在，404表示不存在。

```
HEAD twitter
```

```
curl -I "localhost:19200/_alias/2016?pretty"
HTTP/1.1 200 OK
content-type: application/json; charset=UTF-8
content-length: 73
```

4.6.5. Put Mapping

put mapping API 允许将字段添加到现有索引。

```
PUT twitter {}

PUT twitter/_mapping/_doc { "properties": { "email": { "type": "keyword" } } }
```

```
curl -X PUT "localhost:19200/twitter/_mapping/_doc?pretty" -H 'Content-Type: application/json'
-d'
{
   "properties": {
     "email": {
       "type": "keyword"
     }
   }
}'
{
  "acknowledged" : true
}
```

4.6.5.1. Multi-index

put mapping API 可以通过单个请求应用于多个索引。 例如，我们可以同时更新 twitter-1 和 twitter-2 的mapping：

PUT twitter-1

PUT twitter-2

PUT /twitter-1,twitter-2/_mapping/_doc { "properties": { "user_name": { "type": "text" } } }

4.6.5.2. Updating field mappings

通常，无法更新现有字段的mapping。但是这条规则有一些例外。 例如：

可以将new properties添加到对象数据类型字段。 可以将新的multi-fields添加到现有字段。
ignore_above参数可以更新

**PUT my_index { "mappings": { "_doc": { "properties": { "name": { "properties": { "first": { "type": "text" } } },
"user_id": { "type": "keyword" } } } } }**

**PUT my_index/_mapping/_doc { "properties": { "name": { "properties": { "last": { "type": "text" } } },
"user_id": { "type": "keyword", "ignore_above": 100 } } }**

4.6.6. Get Mapping

get mapping API 允许检索索引或索引/类型的mapping。

GET /twitter/_mapping

```
curl -X GET "localhost:19200/my_index/_mapping?pretty"
{
  "my_index" : {
    "mappings" : {
      "default_type_" : {
        "properties" : {
          "name" : {
            "type" : "object"
          },
          "user_id" : {
            "type" : "keyword"
          }
        }
      }
    }
  }
}
```

4.6.7. Index Aliases

Scope 中 index aliases API 允许使用名称为索引设置别名，所有 API 都会自动将别名转换为实际的索引名称。 一个别名也可以映射到多个索引，当指定它时，别名会自动扩展为别名索引。 别名也可以与搜索时自动应用的过滤器和路由值相关联。 别名不能与索引同名。

以下是将别名 alias1 与索引 test1 关联的示例：

POST /_aliases { "actions" : [{ "add" : { "index" : "test1", "alias" : "alias1" } }] }

```
curl -X POST "localhost:19200/_aliases?pretty" -H 'Content-Type: application/json' -d'
{
    "actions" : [
        { "add" : { "index" : "test1", "alias" : "alias1" } }
    ]
}'
{
  "acknowledged" : true
}
```

以下是移除别名 alias1 与索引 test1 的关联

POST /_aliases { "actions" : [{ "remove" : { "index" : "test1", "alias" : "alias1" } }] }

重命名别名是在同一个 API 中简单的删除然后添加操作。 这个操作是原子的，不用担心别名不指向索引的短时间内：

POST /_aliases { "actions" : [{ "remove" : { "index" : "test1", "alias" : "alias1" } }, { "add" : { "index" : "test2", "alias" : "alias1" } }] }

将别名与多个索引相关联：

```
POST /_aliases { "actions" : [ { "add" : { "index" : "test1", "alias" : "alias1" } }, { "add" : { "index" : "test2", "alias" : "alias1" } } ] }
```

可以使用索引数组语法为动作指定多个索引：

```
POST /_aliases { "actions" : [ { "add" : { "indices" : ["test1", "test2"], "alias" : "alias1" } } ] }
```

要在一个操作中指定多个别名，相应的别名数组语法也存在。

对于上面的示例，glob 模式也可用于将别名与多个共享同一个名称的索引相关联：

```
POST /_aliases { "actions" : [ { "add" : { "index" : "test*", "alias" : "all_test_indices" } } ] }
```

在这种情况下将对所有匹配的当前索引进行分组，它不会随着与此模式匹配的新索引的添加/删除而自动更新。

4.6.7.1. Filtered Aliases

带有过滤器的别名提供了一种简单的方法来创建同一索引的不同“视图”。

要创建过滤别名，首先我们需要确保mapping中已经存在字段：

```
PUT /test1 { "mappings": { "_doc": { "properties": { "user" : { "type": "keyword" } } } } }
```

现在我们可以创建一个别名，对字段user使用过滤器：

```
POST /_aliases { "actions" : [ { "add" : { "index" : "test1", "alias" : "alias2", "filter" : { "term" : { "user" : "kimchy" } } } } ] }
```

4.6.7.2. Routing

可以将路由值与别名相关联。 此功能可以与过滤别名一起使用，以避免不必要的分片操作。

以下命令创建一个指向索引 test 的新别名 alias1。 创建 alias1 后，所有具有此别名的操作都会自动修改为使用值 1 进行路由：

```
POST /_aliases { "actions" : [ { "add" : { "index" : "test", "alias" : "alias1", "routing" : "1" } } ] }
```

也可以为搜索和索引操作指定不同的路由值：

```
POST /_aliases { "actions" : [ { "add" : { "index" : "test", "alias" : "alias2", "search_routing" : "1,2", "index_routing" : "2" } } ] }
```

4.6.7.3. Write Index

可以将别名指向的索引关联为写索引。 指定后，针对指向多个索引的别名的所有索引和更新请求都将尝试解析为写入索引的一个索引。 每个别名一次只能分配一个索引作为写入索引。 如果没有指定写入索引并且有多个索引被别名引用，则不允许写入。

4.6.7.4. Aliases during index creation

```
PUT /logs_20162801 { "mappings" : { "_doc" : { "properties" : { "year" : {"type" : "integer"} } } }, "aliases" : { "current_day" : {}, "2016" : { "filter" : { "term" : {"year" : 2016 } } } } }
```

4.6.7.5. Retrieving existing aliases

get index alias API 允许按别名和索引名称进行过滤。 此 api 重定向到 master 并获取请求的索引别名（如果可用）。 此 api 仅序列化找到的索引别名。

```
GET /logs_20162801/_alias/*

GET /_alias/2016

GET /_alias/20*
```

4.6.8. Update Indices Settings

实时更改特定的索引级别设置。

REST 端点是 /_settings（更新所有索引）或 {index}/_settings 更新一个（或多个）索引设置。

```
PUT /twitter/_settings { "index" : { "number_of_replicas" : 2 } }
```

```
curl -X PUT "localhost:19200/twitter/_settings?pretty" -H 'Content-Type: application/json' -d'
{
    "index" : {
        "number_of_replicas" : 2
    }
}
'
{
  "acknowledged" : true
}
```

要将设置重置为默认值，请使用 null。

```
PUT /twitter/_settings { "index" : { "refresh_interval" : null } }
```

4.6.9. Get Settings

get settings API 允许检索索引/索引的设置：

GET /twitter/_settings

```
curl -X GET "localhost:19200/twitter/_settings?pretty"
{
  "twitter" : {
    "settings" : {
      "index" : {
        "disk" : {
          "id" : "0"
        },
        "refresh_interval" : "1s",
        "number_of_shards" : "5",
        "creation_date" : "2022-05-09 22:50:10",
        "number_of_replicas" : "3",
        "version" : {
          "created" : "1000001"
        },
        "table" : {
          "id" : "79270321525d4d759a1be032cf1c4a99"
        }
      },
      "path" : {
        "conf" : "../lib/search.yml"
      },
      "disk" : {
        "id" : ""
      },
      "indices" : {
        "plugins" : {
          "config" : {
            "dir" : "/home/transwarp/mygit1/shiva/build/lib/plugins"
          }
        }
      }
    }
  }
}
```

4.6.9.1. Multiple Indices and Types

get settings API 可用于通过一次调用获取多个索引的设置。 API 的一般用法遵循以下语法：host:port/{index}/_settings 其中 {index} 可以代表以逗号分隔的索引名称和别名列表。还支持通配符表达式。 以下是一些示例：

GET /twitter,kimchy/_settings

GET /log_2013_*/_settings

4.6.10. Index Templates

索引模板允许您定义将在创建新索引时自动应用的模板。 模板包括setting和mapping以及控制是否应将模板应用于新索引的简单模式模板。

模板仅在创建索引时应用。 更改模板不会影响现有索引。 使用创建索引 API 时，作为创建索引调用的一部分定义的settings/mappings将优先于模板中定义的任何匹配settings/mappings。

PUT _template/template_1 { "index_patterns": ["te*", "bar*"], "settings": { "number_of_shards": 1 }, "mappings": { "_doc": { "_source": { "enabled": false }, "properties": { "host_name": { "type": "keyword" }, "created_at": { "type": "date", "format": "EEE MMM dd HH:mm:ss Z yyyy" } } } } }

```
curl -X PUT "localhost:19200/_template/template_1?pretty" -H 'Content-Type: application/json'
-d'
{
   "index_patterns": ["te*", "bar*"],
   "settings": {
     "number_of_shards": 1
   },
   "mappings": {
     "_doc": {
       "_source": {
         "enabled": false
       },
       "properties": {
         "host_name": {
           "type": "keyword"
         },
         "created_at": {
           "type": "date",
           "format": "EEE MMM dd HH:mm:ss Z yyyy"
         }
       }
     }
   }
}'
{
  "acknowledged" : true
}
```

定义一个名为 template_1 的模板，模板模式为 te* 或 bar*。 设置和mapping将应用于与 te* 或 bar* 模式匹配的任何索引名称。

也可以在索引模板中包含别名，如下所示：

```
PUT _template/template_1 { "index_patterns" : ["te*"], "settings" : { "number_of_shards" : 1 }, "aliases" : { "alias1" : {}, "alias2" : { "filter" : { "term" : {"user" : "kimchy" } }, "routing" : "kimchy" }, "{index}-alias" : {} }
}
```

4.6.10.1. Deleting a Template

可以使用以下命令删除模板

```
DELETE /_template/template_1
```

4.6.10.2. Multiple Templates Matching

多个索引模板可能会匹配一个索引，在这种情况下，setting和mapping都会合并到索引的最终配置中。 可以使用 order 参数控制合并的顺序，首先应用较低的顺序，然后应用较高的顺序覆盖它们。 例如：

```
PUT /_template/template_1 { "index_patterns" : ["*"], "order" : 0, "settings" : { "number_of_shards" : 1 }, "mappings" : { "_doc" : { "_source" : { "enabled" : false } } } }

PUT /_template/template_2 { "index_patterns" : ["te*"], "order" : 1, "settings" : { "number_of_shards" : 1 }, "mappings" : { "_doc" : { "_source" : { "enabled" : true } } } }
```

以上将禁用存储 _source，但对于以 te* 开头的索引，_source 仍将启用。 请注意，对于mapping，合并是“深度”的，这意味着可以在高阶模板上轻松添加/覆盖基于特定对象/属性的mapping，而低阶模板提供基

础。

4.6.11. Refresh

refresh API 允许显式刷新一个或多个索引，使自上次刷新以来执行的所有操作都可用于搜索。（接近）实时功能取决于所使用的索引引擎。

POST /twitter/_refresh

```
curl -X POST "localhost:19200/test/_refresh?pretty"
{
  "_shards" : {
    "total" : 3,
    "successful" : 3,
    "failed" : 0
  }
}
```

4.6.11.1. Multi Index

refresh API可以支持多个索引同时刷新

POST /kimchy,scope/_refresh

POST /_refresh

4.6.12. Open/Close Index

Open/Close Index API 允许关闭索引，然后再打开它。 一个封闭的索引在集群上几乎没有开销（除了维护它的元数据），并且被阻塞用于读/写操作。 可以打开一个已关闭的索引，然后该索引将通过正常的恢复过程。

最基本的命令如下：

curl -X PUT "localhost:9200/index/_open?pretty"

curl -X PUT "localhost:9200/index/_close?pretty"

上述命令打开/关闭了一个名为index的索引。

```
curl -X POST "localhost:19200/my_index/_close?pretty"
{
  "acknowledged" : true
}

curl -X POST "localhost:19200/my_index/_open?pretty"
{
  "acknowledged" : true
}
```

封闭索引会消耗大量磁盘空间，这可能会导致托管环境出现问题。

4.7. Mapping

4.7.1. Field datatypes

4.7.1.1. Binary datatype

binary 类型接受二进制值作为 Base64 编码的字符串。 该字段默认不存储且不可被搜索：

```
PUT my_index { "mappings": { "_doc": { "properties": { "name": { "type": "text" }, "blob": { "type": "binary" } } } } }

PUT my_index/_doc/1 { "name": "Some binary blob", "blob": "U29tZSBiaW5hcnkgYmxvYg==" }
```

```
curl -X PUT "localhost:19200/my_index?pretty" -H 'Content-Type: application/json' -d' {
   "mappings": {
     "_doc": {
       "properties": {
         "name": {
           "type": "text"
         },
         "blob": {
           "type": "binary"
         }
       }
     }
   }
 }'
{
  "acknowledged" : true,
  "shards_acknowledged" : true,
  "index" : "my_index"
}
```

Parameters for binary fields

binary字段接收以下参数：

doc_values:该字段是否应该以列跨步的方式存储在磁盘上，以便以后可以用于排序、聚合或脚本？ 接受true或false（默认）。

store:字段值是否应与 _source 字段分开存储和检索。 接受true或false（默认）。

4.7.1.2. Boolean datatype

布尔字段接受 true 和 false 值，也可以接受被解释为 true 或 false 的字符。

```
PUT my_index { "mappings": { "_doc": { "properties": { "is_published": { "type": "boolean" } } } } }

POST my_index/_doc/1 { "is_published": "true" }

GET my_index/_search { "query": { "term": { "is_published": true } } }
```

```
curl -X PUT "localhost:19200/my_index?pretty" -H 'Content-Type: application/json' -d'{
   "mappings": {
     "_doc": {
       "properties": {
         "is_published": {
           "type": "boolean"
         }
       }
     }
   }
 }'
{
  "acknowledged" : true,
  "shards_acknowledged" : true,
  "index" : "my_index"
}
```

Parameters for boolean fields

boolean字段接受以下参数：

boost: Mapping字段查询时间提升。 接受浮点数，默认为 1.0。

doc_value: 该字段是否应该以列跨步的方式存储在磁盘上，以便以后可以用于排序、聚合或脚本。 接受true（默认）和false。

index: 该字段应该是可搜索的吗？ 接受true（默认）和false。

null_value: 接受上面列出的任何true或false值。 该值将替换任何显式空值。 默认为 null，这意味着该字段被视为缺失。

store: 字段值是否应与 _source 字段分开存储和检索。 接受true（默认）和false。

4.7.1.3. Date datatype

Scope 中的日期可以是：

包含格式化日期的字符串，例如“2015-01-01”或“2015/01/01 12:10:30”。 一个长数，表示自时代以来的毫秒数。 一个整数，表示自时代以来的秒数。 在内部，日期被转换为 UTC（如果指定了时区）并存储为一个长数，表示自时代以来的毫秒数。

可以自定义日期格式，但如果未指定格式，则使用默认格式：

“strict_date_optional_time||epoch_millis” 这意味着它将接受带有可选时间戳的日期，这些时间戳符合 strict_date_optional_time 或 milliseconds-since-the-epoch 支持的格式。

PUT my_index { "mappings": { "_doc": { "properties": { "date": { "type": "date" } } } } }

PUT my_index/_doc/1 { "date": "2015-01-01" }

PUT my_index/_doc/2 { "date": "2015-01-01T12:10:30Z" }

PUT my_index/_doc/3 { "date": 1420070400001 }

GET my_index/_search { "sort": { "date": "asc"} }

```
curl -X PUT "localhost:19200/my_index?pretty" -H 'Content-Type: application/json' -d' {
    "mappings": {
      "_doc": {
        "properties": {
          "date": {
            "type": "date"
          }
        }
      }
    }
  }'
{
  "acknowledged" : true,
  "shards_acknowledged" : true,
  "index" : "my_index"
}
```

Multiple date formats

可以通过用 || 分隔来指定多种格式作为分隔符。将依次尝试每种格式，直到找到匹配的格式。

PUT my_index { "mappings": { "_doc": { "properties": { "date": { "type": "date", "format": "yyyy-MM-dd HH:mm:ss||yyyy-MM-dd||epoch_millis" } } } } }

Parameters for date fields

date字段接受以下参数：

boost: Mapping字段查询时间提升。接受浮点数，默认为 1.0。

doc_values: 该字段是否应该以列跨步的方式存储在磁盘上，以便以后可以用于排序、聚合或脚本？接受true（默认）或false。

format: 可以解析的日期格式。默认为 strict_date_optional_time||epoch_millis。

locale: 解析自月份以来的日期时使用的语言环境在所有语言中都没有相同的名称和/或缩写。默认为 ROOT 语言环境，

index:该字段应该是可搜索的吗？接受true（默认）或false。

null_value: 接受配置格式之一的日期值作为替换任何显式空值的字段。默认为 null，这意味着该字段被视为缺失。

store: 字段值是否应与 _source 字段分开存储和检索。接受true（默认）或false。

4.7.1.4. IP datatype

PUT my_index { "mappings": { "_doc": { "properties": { "ip_addr": { "type": "ip" } } } } }

PUT my_index/_doc/1 { "ip_addr": "192.168.1.1" }

GET my_index/_search { "query": { "term": { "ip_addr": "192.168.0.0/16" } } }

```
transwarp@transwarp-ThinkPad-T14-Gen-1:~/mygit1/shiva/build/bin$ curl -X PUT
"localhost:19200/my_index?pretty" -H 'Content-Type: application/json' -d'
{
  "mappings": {
    "_doc": {
      "properties": {
        "ip_addr": {
          "type": "ip"
        }
      }
    }
  }
}
'
{
  "acknowledged" : true,
  "shards_acknowledged" : true,
  "index" : "my_index"
}
```

Parameters for ip fields

ip字段接受以下参数：

boost：Mapping字段查询时间提升。接受浮点数，默认为 1.0。

doc_values：该字段是否应该以列跨步的方式存储在磁盘上，以便以后可以用于排序、聚合或脚本？接受true（默认）或false。

index：该字段应该是可搜索的吗？接受true（默认）或false。

null_value：接受配置格式之一的日期值作为替换任何显式空值的字段。默认为 null，这意味着该字段被视为缺失。

store：字段值是否应与 _source 字段分开存储和检索。接受true（默认）或false。

4.7.1.5. Keyword datatype

用于索引结构化内容的字段，例如电子邮件地址、主机名、状态代码、邮政编码或标签。

它们通常用于过滤（查找所有发布状态的博客文章）、排序和聚合。 关键字字段只能按其精确值进行搜索。

如果您需要索引全文内容，例如电子邮件正文或产品描述，您可能应该使用文本字段。

以下是关键字字段的示例：

```
PUT my_index { "mappings": { "_doc": { "properties": { "tags": { "type": "keyword" } } } } }
```

```
curl -X PUT "localhost:19200/my_index?pretty" -H 'Content-Type: application/json' -d'{
   "mappings": {
     "_doc": {
       "properties": {
         "tags": {
           "type":  "keyword"
         }
       }
     }
   }
}'
{
  "acknowledged" : true,
  "shards_acknowledged" : true,
  "index" : "my_index"
}
```

Parameters for keyword fields

boost：mapping字段查询时间提升。接受浮点数，默认为 1.0。

doc_values：该字段是否应该以列跨步的方式存储在磁盘上，以便以后可以用于排序、聚合或脚本？接受true（默认）或false。

fields：多字段允许为不同目的以多种方式索引相同的字符串值，例如一个用于搜索的字段和一个用于排序和聚合的多字段。

ignore_above：不要索引任何长于该值的字符串。默认为 2147483647，以便接受所有值。但是请注意，默认动态映射规则会创建一个子关键字字段，通过设置 ignore_above：256 来覆盖此默认值。

index：该字段应该是可搜索的吗？接受true（默认）或false。

index_options：出于评分目的，应将哪些信息存储在索引中。默认为 docs，但也可以设置为 freqs 以在计算分数时考虑词频。

null_value：接受替换任何显式空值的字符串值。默认为 null，这意味着该字段被视为缺失。

store：字段值是否应与 _source 字段分开存储和检索。接受true（默认）或false。

4.7.1.6. Numeric datatype

支持以下数字类型：

long:一个有符号的 64 位整数，最小值为 -263，最大值为 263-1。

integer:一个带符号的 32 位整数，最小值为 -231，最大值为 231-1。

short:一个带符号的 16 位整数，最小值为 -32,768，最大值为 32,767。

byte:一个有符号的 8 位整数，最小值为 -128，最大值为 127。

double:双精度 64 位 IEEE 754 浮点数，限制为有限值。

float:单精度 32 位 IEEE 754 浮点数，限制为有限值。

PUT my_index { "mappings": { "_doc": { "properties": { "number_of_bytes": { "type": "integer" }, "time_in_seconds": { "type": "float" }, "price": { "type": "scaled_float", "scaling_factor": 100 } } } } }

```
curl -X PUT "localhost:19200/my_index?pretty" -H 'Content-Type: application/json' -d' {
   "mappings": {
     "_doc": {
       "properties": {
         "number_of_bytes": {
           "type": "integer"
         },
         "time_in_seconds": {
           "type": "float"
         },
         "price": {
           "type": "byte"
         }
       }
     }
   }
 }'
{
  "acknowledged" : true,
  "shards_acknowledged" : true,
  "index" : "my_index"
}
```

Parameters for numeric fields

numeric字段可以接收以下参数：

boost: Mapping字段查询时间提升。接受浮点数，默认为 1.0。

doc_values: 该字段是否应该以列跨步的方式存储在磁盘上，以便以后可以用于排序、聚合或脚本？接受true（默认）或false。

index: 该字段应该是可搜索的吗？接受true（默认）或false。

null_value: 接受替换任何显式空值的字符串值。默认为 null，这意味着该字段被视为缺失。

store: 字段值是否应与 _source 字段分开存储和检索。接受true（默认）或false。

4.7.1.7. Text datatype

用于索引全文值的字段，例如电子邮件正文或产品描述。 对这些字段进行分析，也就是说，它们在被索引之前通过analyzer将字符串转换为单个词项的列表。 分析过程允许在每个全文字段中搜索单个单词。 文本字段不用于排序，也很少用于聚合（尽管重要的文本聚合是一个明显的例外）。

下面是一个文本字段的映射示例：

```
PUT my_index { "mappings": { "_doc": { "properties": { "full_name": { "type": "text" } } } } }
```

```
curl -X PUT "localhost:19200/my_index?pretty" -H 'Content-Type: application/json' -d' {
   "mappings": {
     "_doc": {
       "properties": {
         "full_name": {
           "type":  "text"
         }
       }
     }
   }
 }'

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

Parameters for text fields

对于text字段，可以接收以下参数：

analyzer: 在索引时间和搜索时间都应该用于分析的字符串字段的analyzer（除非被 search_analyzer 覆盖）。

boost: Mapping字段查询时间提升。接受浮点数，默认为 1.0。

doc_values: 该字段是否应该以列跨步的方式存储在磁盘上，以便以后可以用于排序、聚合或脚本？接受true（默认）或false。

fields: 多字段允许为不同目的以多种方式索引相同的字符串值，例如一个用于搜索的字段和一个用于排序和聚合的多字段。

index: 该字段应该是可搜索的吗？接受true（默认）或false。

index_options: 出于评分目的，应将哪些信息存储在索引中。默认为 docs，但也可以设置为 freqs 以在计算分数时考虑词频。

null_value: 接受替换任何显式空值的字符串值。默认为 null，这意味着该字段被视为缺失。

store: 字段值是否应与 _source 字段分开存储和检索。接受true（默认）或false。

4.7.2. Mapping parameters

4.7.2.1. analyzer

被分析的字符串字段的值通过 analyzer 传递，以将字符串转换为词元。例如，字符串“The quick Brown Foxes”。根据使用的 analyzer ，可能会被分析为：quick、brown、foxes。这使得可以在大块文本中有效地搜索单个单词。

此分析过程不仅需要在索引时进行，而且还需要在查询时进行：查询字符串需要通过相同（或类似）的analyzer，以便它尝试查找的词项与那些词项具有相同的格式存在于索引中。

Scope 附带了许多预定义的 analyzer，无需进一步配置即可使用。它还附带了许多字符过滤器、标记器和标记过滤器，它们可以组合起来为每个索引配置自定义 analyzer。

可以按查询、按字段或按索引指定 analyzer。

PUT /my_index { "mappings": { "_doc": { "properties": { "text": { "type": "text", "fields": { "english": { "type": "text", "analyzer": "english" } } } } } } }

GET my_index/_analyze { "field": "text", "text": "The quick Brown Foxes." }

GET my_index/_analyze { "field": "text.english", "text": "The quick Brown Foxes." }

```
curl -XPUT "localhost:9200/my_index?pretty" -H 'Content-Type: application/json' -d'{
   "mappings": {
     "_doc": {
       "properties": {
         "text": {
           "type": "text",
           "fields": {
             "english": {
               "type":     "text",
               "analyzer": "english"
             }
           }
         }
       }
     }
   }
}'

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

4.7.2.2. copy_to

copy_to 参数可以将多个字段的值复制到一个组字段中，然后可以将其作为单个字段进行查询。 例如，可以将 first_name 和 last_name 字段复制到 full_name 字段，如下所示：

PUT my_index { "mappings": { "_doc": { "properties": { "first_name": { "type": "text", "copy_to": "full_name" }, "last_name": { "type": "text", "copy_to": "full_name" }, "full_name": { "type": "text" } } } } }

PUT my_index/_doc/1 { "first_name": "John", "last_name": "Smith" }

GET my_index/_search { "query": { "match": { "full_name": { "query": "John Smith", "operator": "and" } } } }

```
curl -XPUT "localhost:9200/my_index?pretty" -H 'Content-Type: application/json' -d' {
"mappings": {
      "_doc": {
        "properties": {
          "first_name": {
            "type": "text",
            "copy_to": "full_name"
          },
          "last_name": {
            "type": "text",
            "copy_to": "full_name"
          },
          "full_name": {
            "type": "text"
          }
        }
      }
    }
}'

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

4.7.2.3. doc_values

默认情况下，大多数字段都已编入索引，这使得它们可搜索。倒排索引允许查询在唯一排序的词项列表中查找搜索词项，并从中立即访问包含该词项的文档列表。

排序、聚合和访问脚本中的字段值需要不同的数据访问模式。需要能够查找文档并找到它在字段中的词项，而不是查找词项和查找文档。

文档值是在文档索引时构建的磁盘数据结构。它们存储与 _source 相同的值，但以面向列的方式存储，这对于排序和聚合更有效。几乎所有字段类型都支持doc_values。

所有支持doc_values的字段都默认启用它们。如果确定不需要对字段进行排序或聚合，或从脚本访问字段值，则可以禁用 doc_values以节省磁盘空间：

PUT my_index { "mappings": { "_doc": { "properties": { "status_code": { "type": "keyword" }, "session_id": { "type": "keyword", "doc_values": false } } } } }

```
curl -X PUT "localhost:9200/my_index?pretty" -H 'Content-Type: application/json' -d' {
   "mappings": {
     "_doc": {
       "properties": {
         "status_code": {
           "type":       "keyword"
         },
         "session_id": {
           "type":       "keyword",
           "doc_values": false
         }
       }
     }
   }
}'

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

4.7.2.4. format

在 JSON 文档中，日期表示为字符串。Scope使用一组预配置的格式来识别这些字符串并将其解析为一个 long 值，该值表示 UTC 中的毫秒数。

除了内置格式之外，还可以使用熟悉的 yyyy/MM/dd 语法指定自定义格式：

```
PUT my_index { "mappings": { "_doc": { "properties": { "date": { "type": "date", "format": "yyyy-MM-dd" } } } } }
```

```
curl -X PUT "localhost:9200/my_index?pretty" -H 'Content-Type: application/json' -d'{
   "mappings": {
     "_doc": {
       "properties": {
         "date": {
           "type":   "date",
           "format": "yyyy-MM-dd"
         }
       }
     }
   }
}'
{
  "acknowledged" : true,
  "shards_acknowledged" : true,
  "index" : "my_index"
}
```

以下大多数格式都有严格的配套格式，例如，像 5/11/1 这样的日期将被视为无效，需要将其重写为 2005/11/01 才能被日期解析器接受。

要使用它们，您需要在日期格式的名称前面加上 strict_，例如 strict_date_optional_time 而不是 date_optional_time。

这些严格的日期格式在动态映射日期字段时特别有用，以确保不会意外地将不相关的字符串映射为日期。

下面列出了支持的所有默认 ISO 格式：

epoch_millis：自纪元以来的毫秒数的格式化程序。请注意，此时间戳受 Java Long.MIN_VALUE 和 Long.MAX_VALUE 的限制。

epoch_second：自纪元以来的秒数的格式化程序。请注意，此时间戳受 Java Long.MIN_VALUE 和 Long 的限制。 MAX_VALUE 除以 1000（一秒中的毫秒数）。

date_optional_time 或 strict_date_optional_time：一个通用的 ISO 日期时间解析器，其中日期是强制性的，时间是可选的。

basic_date:完整日期的基本格式化程序，如四位数年份、两位数月份和两位数月份日期：yyyyMMdd。

basic_date_time:结合了基本日期和时间的基本格式化程序，由 T 分隔：yyyyMMdd’T’HHmmss.SSSZ。

basic_date_time_no_millis: 一个基本的格式化程序，它结合了没有毫秒的基本日期和时间，用 T 分隔：yyyyMMdd’T’HHmmssZ。

basic_ordinal_date:完整序数日期的格式化程序，使用四位数年份和三位数 dayOfYear：yyyyDDD。

basic_ordinal_date_time:一个完整的序数日期和时间的格式化程序，使用四位数的年份和三位数的 dayOfYear：yyyyDDD’T’HHmmss.SSSZ。

basic_ordinal_date_time_no_millis：没有毫秒的完整序数日期和时间的格式化程序，使用四位数的年份和三位数的 dayOfYear：yyyyDDD’T’HHmmssZ。

basic_time:用于一天中两位数的小时、两位数的分钟、两位数的分钟、三位毫秒和时区偏移的基本格式化程序：HHmmss.SSSZ。

basic_time_no_millis:用于一天中两位数的小时、两位数的小时、两位数的分钟和时区偏移的基本格式化程序：HHmmssZ。

basic_t_time:用于一天中两位数小时、两位数分钟数、两位数秒数、三位数毫秒和以 T 为前缀的时区偏移的基本格式化程序：’T’HHmmss.SSSZ。

basic_t_time_no_millis:一个基本的格式化程序，用于一天中的两位数小时、小时的两位数分钟、分钟的两位数秒以及以 T 为前缀的时区偏移量：’T’HHmmssZ。

basic_week_date 或 strict_basic_week_date: 完整日期的基本格式化程序，如四位数的周年、两位数的周年和一位数的星期：xxxx’W’wwe。

basic_week_date_time 或 strict_basic_week_date_time: 一个基本的格式化程序，它结合了基本的星期日期和时间，用 T 分隔：xxxx’W’wwe’T’HHmmss.SSSZ。

basic_week_date_time_no_millis 或 strict_basic_week_date_time_no_millis:一个基本的格式化程序，它结合了基本的星期日期和时间，没有毫秒，用 T 分隔：xxxx’W’wwe’T’HHmmssZ。

date 或 strict_date:将完整日期格式化为四位数年份、两位数月份和两位数月份日期：yyyy-MM-dd。

date_hour 或 strict_date_hour:结合了完整日期和两位数小时的格式化程序：yyyy-MM-dd’T’HH。

date_hour_minute 或 strict_date_hour_minute:结合完整日期、两位数小时和两位数分钟的格式化程序：yyyy-MM-dd’T’HH:mm。

date_hour_minute_second 或 strict_date_hour_minute_second:结合了完整日期、两位数小时、两位数分钟和两位数分钟的格式化程序：yyyy-MM-dd’T’HH:mm:ss。

date_hour_minute_second_fraction 或 strict_date_hour_minute_second_fraction:结合了完整日期、一天中的两位数小时、小时的两位数分钟、分钟的两位数秒和秒的三位小数的格式化程序：yyyy-MM-dd’T’HH:mm:ss.SSS。

date_hour_minute_second_millis 或 strict_date_hour_minute_second_millis:结合了完整日期、一天中的两位数小时、小时的两位数分钟、分钟的两位数秒和秒的三位小数的格式化程序：yyyy-MM-dd’T’HH:mm:ss.SSS。

date_time 或 strict_date_time 结合完整日期和时间的格式化程序，由 T 分隔：yyyy-MM-dd’T’HH:mm:ss.SSSZZ。

date_time_no_millis 或 strict_date_time_no_millis:一个格式化程序，它结合了完整的日期和时间，没有毫秒，用 T 分隔：yyyy-MM-dd’T’HH:mm:ssZZ。

hour 或 strict_hour:一天中两位数小时的格式化程序：HH

hour_minute 或 strict_hour_minute:一天中两位数小时和小时两位数分钟的格式化程序：HH:mm。

hour_minute_second 或 strict_hour_minute_second:用于一天中的两位数小时、小时的两位数分钟和分钟的两位数秒的格式化程序：HH:mm:ss。

hour_minute_second_fraction 或 strict_hour_minute_second_fraction:一天中两位数小时、小时两位数

分钟、分钟两位数秒和秒的三位小数的格式化程序：HH:mm:ss.SSS。

hour_minute_second_millis 或 strict_hour_minute_second_millis:一天中两位数小时、小时两位数分钟、分钟两位数秒和秒的三位小数的格式化程序：HH:mm:ss.SSS。

ordinal_date 或 strict_ordinal_date:完整序数日期的格式化程序，使用四位数年份和三位数 dayOfYear：yyyy-DDD。

ordinal_date_time 或 strict_ordinal_date_time:完整的序数日期和时间的格式化程序，使用四位数的年份和三位数的 dayOfYear：yyyy-DDD’T’HH:mm:ss.SSSZZ。

ordinal_date_time_no_millis 或 strict_ordinal_date_time_no_millis:没有毫秒的完整序数日期和时间的格式化程序，使用四位数的年份和三位数的 dayOfYear：yyyy-DDD’T’HH:mm:ssZZ。

time 或 strict_time:一天中两位数小时、小时两位数分钟、分钟两位数秒、秒的三位小数和时区偏移的格式化程序：HH:mm:ss.SSSZZ。

time_no_millis 或 strict_time_no_millis:一天中两位数小时、小时两位数分钟、分钟两位数秒和时区偏移量的格式化程序：HH:mm:ssZZ。

t_time 或 strict_t_time:用于一天中的两位数小时、小时的两位数分钟、分钟的两位数秒、秒的三位小数以及以 T:’T’HH:mm:ss.SSSZZ 为前缀的时区偏移量的格式化程序。

t_time_no_millis 或 strict_t_time_no_millis:用于一天中的两位数小时、小时的两位数分钟、分钟的两位数秒和以 T 为前缀的时区偏移量的格式化程序：’T’HH:mm:ssZZ。

week_date 或 strict_week_date:将完整日期格式化为四位数的周年、两位数的周数和一位数的周数：xxxx-’W’ww-e。

week_date_time 或 strict_week_date_time:一个组合了完整的星期日期和时间的格式化程序，用 T 分隔：xxxx-’W’ww-e’T’HH:mm:ss.SSSZZ。

week_date_time_no_millis 或 strict_week_date_time_no_millis:一个格式化程序，它结合了一个完整的星期日期和时间，没有毫秒，用 T 分隔：xxxx-’W’ww-e’T’HH:mm:ssZZ。

weekyear 或 strict_weekyear:四位数周年的格式化程序：xxxx。

weekyear_week 或 strict_weekyear_week:用于四位数周年和两位数周的格式化程序：xxxx-’W’ww。

weekyear_week_day 或 strict_weekyear_week_day:一个四位数的星期数、两位数的星期数和一位数的星期几的格式化程序：xxxx-’W’ww-e。

year 或 strict_year:四位数年份的格式化程序：yyyy。

year_month 或 strict_year_month:用于四位数字年份和一年中两位数字月份的格式化程序：yyyy-MM。

year_month_day 或 strict_year_month_day:用于四位数年份、两位数月份和两位数月份日期的格式化程序：yyyy-MM-dd。

4.7.2.5. ignore_above

长于 ignore_above 设置的字符串将不会被索引或存储。 对于字符串数组，ignore_above 将分别应用于每个数组元素，并且长度超过 ignore_above 的字符串元素将不会被索引或存储。

PUT my_index { "mappings": { "_doc": { "properties": { "message": { "type": "keyword", "ignore_above": 20 } } } } }

PUT my_index/_doc/1 { "message": "Syntax error" }

PUT my_index/_doc/2 { "message": "Syntax error with some long stacktrace" }

GET _search { "aggs": { "messages": { "terms": { "field": "message" } } } }

```
curl -X PUT "localhost:9200/my_index?pretty" -H 'Content-Type: application/json' -d'{
   "mappings": {
     "_doc": {
       "properties": {
         "message": {
           "type": "keyword",
           "ignore_above": 20
         }
       }
     }
   }
}'

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

4.7.2.6. index

index 选项控制字段值是否被索引。它接受true或false，默认为true。未编入索引的字段不可查询。

4.7.2.7. fields

出于不同目的以不同方式索引同一字段通常很有用。 这就是多领域的目的。 例如，text字段可以映射为全文搜索的文本字段，以及排序的keyword字段：

PUT my_index { "mappings": { "_doc": { "properties": { "city": { "type": "text", "fields": { "raw": { "type": "keyword" } } } } } } }

PUT my_index/_doc/1 { "city": "New York" }

PUT my_index/_doc/2 { "city": "York" }

GET my_index/_search { "query": { "match": { "city": "york" } }, "sort": { "city.raw": "asc" } }

```
curl -X PUT "localhost:9200/my_index?pretty" -H 'Content-Type: application/json' -d' {
   "mappings": {
     "_doc": {
       "properties": {
         "city": {
           "type": "text",
           "fields": {
             "raw": {
               "type":  "keyword"
             }
           }
         }
       }
     }
   }
}'

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

4.7.2.8. position_increment_gap

分析的文本字段会考虑词项位置，以便能够支持邻近或短语查询。 当索引具有多个值的文本字段时，值之间会添加一个“假”间隙，以防止大多数短语查询在值之间匹配。 此间隙的大小使用 position_increment_gap 配置，默认为 100。

PUT my_index { "mappings": { "_doc": { "properties": { "names": { "type": "text", "position_increment_gap": 0 } } } } }

PUT my_index/_doc/1 { "names": ["John Abraham", "Lincoln Smith"] }

GET my_index/_search { "query": { "match_phrase": { "names": "Abraham Lincoln" } } }

```
curl -XPUT "localhost:9200/my_index?pretty" -H 'Content-Type: application/json' -d' {
   "mappings": {
     "_doc": {
       "properties": {
         "names": {
           "type": "text",
           "position_increment_gap": 0
         }
       }
     }
   }
}'

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

4.7.2.9. search_analyzer

通常，在索引时和搜索时应该应用相同的analyzer，以确保查询中的词项与倒排索引中的词项格式相同。

但是，有时需要在搜索时使用不同的analyzer，例如在使用 edge_ngram 标记器进行自动完成时。

默认情况下，查询将使用字段mapping中定义的analyzer，但这可以用 search_analyzer 设置覆盖：

PUT my_index { "settings": { "analysis": { "filter": { "autocomplete_filter": { "type": "edge_ngram", "min_gram": 1, "max_gram": 20 } }, "analyzer": { "autocomplete": { "type": "custom", "tokenizer": "standard", "filter": ["lowercase", "autocomplete_filter"] } } } }, "mappings": { "_doc": { "properties": { "text": { "type": "text", "analyzer": "autocomplete", "search_analyzer": "standard" } } } } }

PUT my_index/_doc/1 { "text": "Quick Brown Fox" }

GET my_index/_search { "query": { "match": { "text": { "query": "Quick Br", "operator": "and" } } } }

```
curl -X PUT "localhost:9200/my_index?pretty" -H 'Content-Type: application/json' -d' {
   "settings": {
     "analysis": {
       "filter": {
         "autocomplete_filter": {
           "type": "edge_ngram",
           "min_gram": 1,
           "max_gram": 20
         }
       },
       "analyzer": {
         "autocomplete": {
           "type": "custom",
           "tokenizer": "standard",
           "filter": [
             "lowercase",
             "autocomplete_filter"
           ]
         }
       }
     }
   },
   "mappings": {
     "_doc": {
       "properties": {
         "text": {
           "type": "text",
           "analyzer": "autocomplete",
           "search_analyzer": "standard"
         }
       }
     }
   }
}'
{
  "acknowledged" : true,
  "shards_acknowledged" : true,
  "index" : "my_index"
}
```

4.7.2.10. store

默认情况下，对字段值进行索引以使其可搜索，但不会存储它们。 这意味着可以查询该字段，但无法检索原始字段值。

通常这无关紧要。 字段值已经是默认存储的 _source 字段的一部分。 如果您只想检索单个字段或几个字段的值，而不是整个 _source，则可以通过源过滤来实现。

在某些情况下，存储字段是有意义的。 例如，如果有一个包含标题、日期和非常大的内容字段的文档，只想检索标题和日期，而不必从大的 _source 字段中提取这些字段：

PUT my_index { "mappings": { "_doc": { "properties": { "title": { "type": "text", "store": true }, "date": { "type": "date", "store": true }, "content": { "type": "text" } } } } }

PUT my_index/_doc/1 { "title": "Some short title", "date": "2015-01-01", "content": "A very long content field…" }

GET my_index/_search { "stored_fields": ["title", "date"] }

```
curl -X PUT "localhost:9200/my_index?pretty" -H 'Content-Type: application/json' -d' {
   "mappings": {
     "_doc": {
       "properties": {
         "title": {
           "type": "text",
           "store": true
         },
         "date": {
           "type": "date",
           "store": true
         },
         "content": {
           "type": "text"
         }
       }
     }
   }
}'

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

4.8. AggregationAPI

4.8.1. Metrics Aggregation

4.8.1.1. Avg Aggregation

一种单值指标聚合，用于计算从聚合文档中提取的数值的平均值。这些值可以从文档中的特定数字字段中提取。

假设数据由代表学生考试成绩（介于 0 到 100 之间）的文档组成，我们可以将他们的分数平均为：

POST /exams/_search?size=0 { "aggs" : { "avg_grade" : { "avg" : { "field" : "grade" } } } }

上述聚合计算所有文档的平均成绩。聚合类型是 avg ，字段设置定义了将计算平均值的文档的数字字段。

```
curl -X POST "localhost:19200/exams/_search?size=0&pretty" -H 'Content-Type: application/json'
-d' {
     "aggs" : {
         "avg_grade" : { "avg" : { "field" : "grade" } }
     }
}'
{
  "took" : 237,
  "timed_out" : false,
  "terminated_early" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "skipped" : 0,
    "failed" : 0
  },
  "_clusters" : {
    "total" : 5,
    "successful" : 5,
    "skipped" : 0
  },
  "hits" : {
    "total" : 0,
    "max_score" : 0.0,
    "hits" : [ ]
  },
  "aggregations" : {
    "avg_grade" : {
      "value" : 3.0
    }
  }
}
```

Missing value

missing参数定义了应该如何处理缺少值的文档。 默认情况下，它们将被忽略。

```
POST /exams/_search?size=0 { "aggs" : { "grade_avg" : { "avg" : { "field" : "grade", "missing": 10 } } } }
```

4.8.1.2. Max Aggregation

一种单值度量聚合，用于跟踪并返回从聚合文档中提取的数值中的最大值。 这些值可以从文档中的特定数字字段中提取。

```
POST /sales/_search?size=0 { "aggs" : { "max_price" : { "max" : { "field" : "price" } } } }
```

```
curl -X POST "localhost:19200/sales/_search?size=0&pretty" -H 'Content-Type: application/json'
-d'{
    "aggs" : {
        "max_price" : { "max" : { "field" : "price" } }
    }
}
'
```

```
{
  "took" : 8,
  "timed_out" : false,
  "terminated_early" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "skipped" : 0,
    "failed" : 0
  },
  "_clusters" : {
    "total" : 5,
    "successful" : 5,
    "skipped" : 0
  },
  "hits" : {
    "total" : 0,
    "max_score" : 0.0,
    "hits" : [ ]
  },
  "aggregations" : {
    "max_price" : {
      "value" : 4.0
    }
  }
}
```

Missing value

missing参数定义了应该如何处理缺少值的文档。 默认情况下，它们将被忽略。

```
POST /sales/_search { "aggs" : { "grade_max" : { "max" : { "field" : "grade", "missing": 10 } } } }
```

4.8.1.3. Min Aggregation

一种单值度量聚合，用于跟踪并返回从聚合文档中提取的数值中的最小值。 这些值可以从文档中的特定数字字段中提取。

```
POST /sales/_search?size=0 { "aggs" : { "min_price" : { "min" : { "field" : "price" } } } }
```

```
curl -X POST "localhost:19200/sales/_search?size=0&pretty" -H 'Content-Type: application/json'
-d' {
     "aggs" : {
         "min_price" : { "min" : { "field" : "price" } }
     }
}'

{
  "took" : 14,
  "timed_out" : false,
  "terminated_early" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "skipped" : 0,
    "failed" : 0
  },
  "_clusters" : {
    "total" : 5,
    "successful" : 5,
    "skipped" : 0
  },
  "hits" : {
    "total" : 0,
    "max_score" : 0.0,
    "hits" : [ ]
  },
  "aggregations" : {
    "min_price" : {
      "value" : 2.0
    }
  }
}
```

Missing value

missing参数定义了应该如何处理缺少值的文档。 默认情况下，它们将被忽略。

```
POST /sales/_search { "aggs" : { "grade_min" : { "min" : { "field" : "grade", "missing": 10 } } } }
```

4.8.1.4. Sum Aggregation

对从聚合文档中提取的数值进行汇总的单值度量聚合。 这些值可以从文档中的特定数字字段中提取。

```
POST /sales/_search?size=0 { "aggs" : { "hat_prices" : { "sum" : { "field" : "price" } } } }
```

```
curl -X POST "localhost:19200/sales/_search?size=0&pretty" -H 'Content-Type: application/json'
-d'
{
    "aggs" : {
        "hat_prices" : { "sum" : { "field" : "price" } }
    }
}
'
{
  "took" : 3,
  "timed_out" : false,
  "terminated_early" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "skipped" : 0,
    "failed" : 0
  },
  "_clusters" : {
    "total" : 5,
    "successful" : 5,
    "skipped" : 0
  },
  "hits" : {
    "total" : 0,
    "max_score" : 0.0,
    "hits" : [ ]
  },
  "aggregations" : {
    "hat_prices" : {
      "value" : 50.0
    }
  }
}
```

Missing value

missing参数定义了应该如何处理缺少值的文档。 默认情况下，它们将被忽略。

```
POST /sales/_search?size=0 { "aggs" : { "hat_prices" : { "sum" : { "field" : "price", "missing": 100 } } } }
```

4.8.1.5. Top Hits Aggregation

top_hits 度量聚合器跟踪正在聚合的最相关文档。 该聚合器旨在用作子聚合器，以便可以按桶聚合最匹配的文档。

top_hits 聚合器可以有效地用于通过桶聚合器按某些字段对结果集进行分组。

Options

from ： 要获取的第一个结果的偏移量。

size ： 每个桶返回的最大匹配命中数。默认情况下，返回前三个匹配的命中。

sort ： 如何对最匹配的匹配项进行排序。 默认情况下，命中按主查询的分数排序。

```
POST /sales/_search?size=0 { "aggs": { "top_sales_hits": { "top_hits": { "sort": [ { "date": { "order": "desc" } } ]
} } } }
```

```
curl -X POST "localhost:19200/sales/_search?size=0&pretty" -H 'Content-Type: application/json'
-d'
{
            "aggs": {
                "top_sales_hits": {
                    "top_hits": {
                        "sort": [
                            {
                                "price": {
                                    "order": "desc"
                                }
                            }
                        ]
                    }
                }
            }
    }
}'
{
  ...
  "aggregations" : {
    "top_sales_hits" : {
      "hits" : {
        "total" : 3,
        "max_score" : null,
        "hits" : [
          {
            "_type" : "default_type_",
            "_id" : "3",
            "_score" : null,
            "_source" : {
              "type" : "hat",
              "price" : 100
            }
          },
          {
            "_type" : "default_type_",
            "_id" : "2",
            "_score" : null,
            "_source" : {
              "type" : "hat",
              "price" : 30
            }
          },
          {
            "_type" : "default_type_",
            "_id" : "1",
            "_score" : null,
            "_source" : {
              "type" : "hat",
              "price" : 20
            }
          }
        ]
      }
    }
  }
}
```

4.8.1.6. Value Count Aggregation

一种单值指标聚合，用于计算从聚合文档中提取的值的数量。 这些值可以从文档中的特定字段中提取。

```
POST /sales/_search?size=0 { "aggs" : { "types_count" : { "value_count" : { "field" : "type" } } } }
```

```
curl -X POST "localhost:19200/sales/_search?size=0&pretty" -H 'Content-Type: application/json'
-d'
{
     "aggs" : {
         "types_count" : { "value_count" : { "field" : "price" } }
     }
}'
{
  "took" : 78,
  "timed_out" : false,
  "terminated_early" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "skipped" : 0,
    "failed" : 0
  },
  "_clusters" : {
    "total" : 5,
    "successful" : 5,
    "skipped" : 0
  },
  "hits" : {
    "total" : 0,
    "max_score" : 0.0,
    "hits" : [ ]
  },
  "aggregations" : {
    "types_count" : {
      "value" : 2
    }
  }
}
```

4.8.2. Bucket Aggregation

4.8.2.1. Date Histogram Aggregation

这种多桶聚合类似于普通的直方图，但它只能与日期值一起使用。此处可以使用日期/时间表达式指定时间间隔。

POST /sales/_search?size=0 { "aggs" : { "sales_over_time" : { "date_histogram" : { "field" : "date", "interval" : "month" } } } }

```
curl -X POST "localhost:19200/sales/_search?size=0&pretty" -H 'Content-Type: application/json'
-d'
{
    "aggs" : {
        "sales_over_time" : {
            "date_histogram" : {
                "field" : "date",
                "interval" : "month"
            }
        }
    }
}'
{
  ...
  "aggregations" : {
    "sales_over_time" : {
      "buckets" : [
        {
          "key_as_string" : "2015-01-01T00:00:00.000Z",
          "key" : 1420070400000,
          "doc_count" : 3
        }
      ]
    }
  }
}
```

Setting intervals

以下是有效的时间规范及其含义：

seconds ： 1000 毫秒； 固定长度间隔（包含闰秒的一分钟的最后一秒除外，它长 2000 毫秒）； 支持倍数。

minutes ： 所有分钟都从 00 秒开始。一分钟（1m）是指定时区中第一分钟的 00 秒和下一分钟的 00 秒之间的间隔，以补偿任何中间的闰秒，因此一小时后的分钟数和秒数相同 开始和结束。多分钟（nm）是 60x1000=60,000 毫秒的间隔。

hours ： 所有小时都从 00 分 00 秒开始。一小时（1h）是指定时区中第一小时的 00:00 分钟和下一小时的 00:00 分钟之间的时间间隔，用于补偿任何中间的闰秒，因此一小时后的分钟数和秒数为 在开始和结束时相同。多个小时（nh）的间隔正好是 60x60x1000=3,600,000 毫秒。

days ： 所有的日子都从最早的时间开始，通常是 00:00:00（午夜）。一天（1d）是指定时区中一天开始和第二天开始之间的间隔，以补偿任何中间时间变化。多天（nd）是 24x60x60x1000=86,400,000 毫秒的间隔。

weeks ： 一周（1w）是指定时区中开始 day_of_week:hour:minute:second 与一周中的同一天和下一周的时间之间的间隔。 不支持多周（nw）。

months ： 一个月（1M）是指定时区的月份开始日和时间与月份的同一天和下个月的时间之间的间隔，因此月份的日期和时间是 在开始和结束时相同。 不支持多月（nM）。

quarters ： 四分之一（1q）是一个月的开始日期和时间与三个月后的同一天和一天的时间之间的间隔，因此一个月的日期和时间是相同的 开始和结束。 不支持多季度（nq）。

years ： 一年（1y）是指定时区的月份开始日和次年时间的间隔，使得开始时的日期和时间相同 并结束。不支持多年（ny）。

Keys

日期表示为 64 位数字，表示自时代以来的时间戳（以毫秒为单位）（UTC 时间 1970 年 1 月 1 日午夜）。 这些时间戳作为存储桶的键名返回。 key_as_string 是使用格式参数规范转换为格式化日期字符串

的相同时间戳：

```
POST /sales/_search?size=0 { "aggs" : { "sales_over_time" : { "date_histogram" : { "field" : "date", "interval" : "1M", "format" : "yyyy-MM-dd" } } } }
```

Timezone

日期时间以 UTC 格式存储在 Scope 中。 默认情况下，所有分桶和舍入也以 UTC 完成。 使用 time_zone 参数指示分桶应使用不同的时区。

```
PUT my_index/_doc/1?refresh { "date": "2015-10-01T00:30:00Z" }

PUT my_index/_doc/2?refresh { "date": "2015-10-01T01:30:00Z" }

GET my_index/_search?size=0 { "aggs": { "by_day": { "date_histogram": { "field": "date", "interval": "day", "time_zone": "-01:00" } } } }
```

Offset

使用 offset 参数通过指定的正 (+) 或负偏移 (-) 持续时间更改每个存储桶的起始值，例如 1h 表示一小时，或 1d 表示一天。 有关更多可能的持续时间选项，请参阅时间单位。

例如，当使用一天的间隔时，每个存储桶从午夜运行到午夜。 将 offset 参数设置为 +6h 会将每个存储桶更改为从早上 6 点运行到早上 6 点：

```
PUT my_index/_doc/1?refresh { "date": "2015-10-01T05:30:00Z" }

PUT my_index/_doc/2?refresh { "date": "2015-10-01T06:30:00Z" }

GET my_index/_search?size=0 { "aggs": { "by_day": { "date_histogram": { "field": "date", "interval": "day", "offset": "+6h" } } } }
```

Keyed Response

将 keyed 标志设置为 true 会将唯一的字符串键与每个存储桶相关联，并将范围作为哈希而不是数组返回。

```
POST /sales/_search?size=0 { "aggs" : { "sales_over_time" : { "date_histogram" : { "field" : "date", "interval" : "1M", "format" : "yyyy-MM-dd", "keyed": true } } } }
```

Missing value

missing 参数定义如何处理缺少值的文档。 默认情况下，它们被忽略。

```
POST /sales/_search?size=0 { "aggs" : { "sale_date" : { "date_histogram" : { "field" : "date", "interval": "year", "missing": "2000/01/01" } } } }
```

4.8.2.2. Nested Aggregation

一种特殊的单桶聚合，可以聚合嵌套文档。以下聚合将返回可以购买的最低价格产品：

```
GET /_search { "aggs" : { "resellers" : { "nested" : { "path" : "resellers" }, "aggs" : { "min_price" : { "min" : { "field" : "resellers.price" } } } } } }
```

4.8.2.3. Terms Aggregation

一种基于多桶值源的聚合，其中桶是动态构建的——每个唯一值一个。terms query应是keyword类型的字段或任何其他适用于桶聚合的数据类型。

```
GET /_search { "aggs" : { "genres" : { "terms" : { "field" : "genre" } } } }
```

```
curl -X GET "localhost:19200/index/_search?pretty" -H 'Content-Type: application/json' -d'
{
    "aggs" : {
        "genres" : {
            "terms" : { "field" : "genre" }
        }
    }
}'
{
  "took" : 24,
  "timed_out" : false,
  "terminated_early" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "skipped" : 0,
    "failed" : 0
  },
  "_clusters" : {
    "total" : 5,
    "successful" : 5,
    "skipped" : 0
  },
  "hits" : {
    "total" : 0,
    "max_score" : 0.0,
    "hits" : [ ]
  },
  "aggregations" : {
    "genres" : {
      "doc_count_error_upper_bound" : 0,
      "sum_other_doc_count" : 0,
      "buckets" : [
        {
          "key" : "led",
          "doc_count" : 1
        }
      ]
    }
  }
}
```

Size

可以设置 size 参数来定义应该从整个术语列表中返回多少个术语桶。 默认情况下，协调搜索过程的节点将请求每个分片提供自己的最大大小术语桶，一旦所有分片都响应，它会将结果减少到最终列表，然后返回给客户端。 这意味着，如果唯一术语的数量大于大小，则返回的列表会略微偏离且不准确（可能是术语计数略有偏离，甚至可能是本应处于最大尺寸的术语 桶没有被退回）。

Document counts are approximate

如上所述，terms Aggregation中的文档计数（以及任何子聚合的结果）并不总是准确的。 这是因为每个分片都提供了自己的视图。

Shard Size

请求的大小越大，结果越准确，但计算最终结果的成本也越高（这都是由于在分片级别上管理的更大的优先级队列以及由于更大的数据传输 节点和客户端）。

shard_size 参数可用于最小化更大请求大小带来的额外工作。 定义后，它将确定协调节点将从每个分片请求多少个词项。 一旦所有分片都响应，协调节点将根据大小参数将它们减少到最终结果,这样，可以提高返回词项的准确性并避免将大桶列表流回的开销给客户。

Calculating Document Count Error

有两个错误值可以显示在terms Aggregation上。如上面例子所示，第一个给出了作为一个整体的聚合值，它表示未进入最终词项列表的词项的最大潜在文档计数。 这计算为从每个分片返回的最后一个词项的文档计数的总和。

Order

可以通过设置 order 参数自定义存储桶的顺序。 默认情况下，存储桶按其 doc_count 降序排列。 可以更改此行为，如下所述：

以升序方式按 doc_count 对存储桶进行排序：

```
GET /_search { "aggs" : { "genres" : { "terms" : { "field" : "genre", "order" : { "_count" : "asc" } } } } }
```

以升序方式按词项的字母顺序对存储桶进行排序：

```
GET /_search { "aggs" : { "genres" : { "terms" : { "field" : "genre", "order" : { "_key" : "asc" } } } } }
```

Minimum document count

可以使用 min_doc_count 选项仅返回匹配多于配置的命中数的术语：

```
GET /_search { "aggs" : { "tags" : { "terms" : { "field" : "tags", "min_doc_count": 10 } } } }
```

上述聚合只会返回在 10 次或更多命中中找到的标签。默认值为 1。

在分片级别收集和排序词项，并在第二步中与从其他分片收集的词项合并。但是，分片没有可用的全局文档计数信息。是否将词项添加到候选列表的决定仅取决于使用本地分片频率在分片上计算的顺序。
min_doc_count 标准仅在合并所有分片的本地词项统计信息后应用。在某种程度上，决定将词项添加为候选者是在不确定该词项是否实际达到所需的 min_doc_count 的情况下做出的。如果低频项填充候选列表，这可能会导致最终结果中缺少许多（全局）高频项。为了避免这种情况，可以增加 shard_size 参数以允许分片上有更多的候选词。但是，这会增加内存消耗和网络流量。

shard_min_doc_count 参数

参数 shard_min_doc_count 规定了一个分片是否应该将词项实际添加到候选列表中的确定性与

min_doc_count 相关。只有当它们在集合中的本地分片频率高于 shard_min_doc_count 时才会考虑词项。如果字典包含许多低频率词项并且对这些不感兴趣（例如拼写错误），那么可以设置 shard_min_doc_count 参数以过滤掉分片级别的候选词项，即使在之后也有合理的确定性不会达到所需的 min_doc_count合并本地计数。 shard_min_doc_count 默认设置为 0，除非明确设置，否则无效。

Multi-field terms aggregation

terms aggregation不支持从同一文档中的多个字段收集词项。 原因是terms aggregation 本身并不收集字符串词项值，而是使用全局序数来生成字段中所有唯一值的列表。 全局序数导致重要的性能提升，这在多个字段是不可能的。

Collect mode

推迟子聚合的计算

对于具有许多唯一词项和少量所需结果的字段，延迟子聚合的计算直到顶部父级聚合被修剪会更有效。 通常，聚合树的所有分支都在一次深度优先传递中展开，然后才会发生任何修剪。 在某些情况下，这可能非常浪费，并且可能会遇到内存限制。 一个示例问题场景是在电影数据库中查询 10 位最受欢迎的演员及其 5 位最常见的联合主演：

```
GET /_search { "aggs" : { "actors" : { "terms" : { "field" : "actors", "size" : 10 }, "aggs" : { "costars" : { "terms" : { "field" : "actors", "size" : 5 } } } } } }
```

即使参与者的数量可能相对较少并且我们只想要 50 个结果桶，但在计算过程中桶的组合爆炸 - 单个参与者可以产生 n^2 桶，其中 n 是参与者的数量。 明智的选择是首先确定 10 位最受欢迎的演员，然后再检查这 10 位演员的最佳联合主演。 这种替代策略就是breadth_first收集模式，而不是depth_first模式。

```
GET /_search { "aggs" : { "actors" : { "terms" : { "field" : "actors", "size" : 10, "collect_mode" : "breadth_first" }, "aggs" : { "costars" : { "terms" : { "field" : "actors", "size" : 5 } } } } } }
```

当使用breadth_first模式时，落入最上面的存储桶的文档集被缓存以供后续重播，因此这样做会产生与匹配文档数量成线性关系的内存开销。 请注意，在使用 breadth_first 设置时，order 参数仍可用于引用来自子聚合的数据——父聚合了解需要在任何其他子聚合之前首先调用此子聚合。

Missing value

missing参数定义了应该如何处理缺少值的文档。 默认情况下，它们将被忽略。

```
GET /_search { "aggs" : { "tags" : { "terms" : { "field" : "tags", "missing": "N/A" } } } }
```

4.8.3. Caching heavy aggregations

可以缓存经常使用的聚合（例如，用于显示在网站主页上）以加快响应速度。 这些缓存的结果与未缓存的聚合返回的结果相同，那么将永远不会得到陈旧的结果。

4.8.4. Returing only aggregation results

在很多情况下需要聚合但不需要搜索命中。 对于这些情况，可以通过设置 size=0 来忽略命中。 例如：

GET /twitter/_search { "size": 0, "aggregations": { "my_agg": { "terms": { "field": "text" } } } }

```
curl -X GET "localhost:19200/index/_search?pretty" -H 'Content-Type: application/json' -d'
{
   "size": 0,
   "aggregations": {
     "my_agg": {
       "terms": {
         "field": "genre"
       }
     }
   }
}'
{
  "took" : 3,
  "timed_out" : false,
  "terminated_early" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "skipped" : 0,
    "failed" : 0
  },
  "_clusters" : {
    "total" : 5,
    "successful" : 5,
    "skipped" : 0
  },
  "hits" : {
    "total" : 0,
    "max_score" : 0.0,
    "hits" : [ ]
  },
  "aggregations" : {
    "my_agg" : {
      "doc_count_error_upper_bound" : 0,
      "sum_other_doc_count" : 0,
      "buckets" : [
        {
          "key" : "led",
          "doc_count" : 1
        },
        {
          "key" : "zijian",
          "doc_count" : 1
        }
      ]
    }
  }
}
```

4.8.5. Returning the type of the aggregation

有时您需要知道聚合的确切类型才能解析其结果。 typed_keys 参数可用于更改响应中聚合的名称，使其以其内部类型为前缀。

考虑以下名为 tweets_over_time 的 date_histogram aggregation，它有一个名为 top_users 的子“top_hits”聚合：

GET /twitter/_search?typed_keys { "aggregations": { "tweets_over_time": { "date_histogram": { "field": "date", "interval": "year" }, "aggregations": { "top_users": { "top_hits": { "size": 1 } } } } } }

```
curl -X GET "localhost:19200/sales/_search?typed_keys&pretty" -H 'Content-Type:
application/json' -d'
{
  "aggregations": {
    "tweets_over_time": {
      "date_histogram": {
        "field": "date",
        "interval": "year"
      },
      "aggregations": {
        "top_users": {
            "top_hits": {
                "size": 1
            }
        }
      }
    }
  }
}
'
{
  "took" : 77,
  "timed_out" : false,
  "terminated_early" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "skipped" : 0,
    "failed" : 0
  },
  "_clusters" : {
    "total" : 5,
    "successful" : 5,
    "skipped" : 0
  },
  "hits" : {
    "total" : 0,
    "max_score" : 0.0,
    "hits" : [ ]
  },
  "aggregations" : {
    "date_histogram#tweets_over_time" : {
      "buckets" : [
        {
          "key_as_string" : "2015-01-01T00:00:00.000Z",
          "key" : 1420070400000,
          "doc_count" : 3,
          "top_hits#top_users" : {
            "hits" : {
              "total" : 3,
              "max_score" : null,
              "hits" : [
                {
                  "_type" : "default_type_",
                  "_id" : "1",
                  "_score" : null,
                  "_source" : {
                    "date" : "2015-01-01"
                  }
                }
              ]
            }
          }
        }
      ]
    }
  }
}
```

在响应中，聚合名称将分别更改为 date_histogram#tweets_over_time 和 top_hits#top_users，反映每个聚合的内部类型。

4.9. Search API

4.9.1. Preference

控制执行搜索的分片副本的preference。 默认情况下，Scope 会以未指定的顺序从可用的分片副本中进行选择，同时考虑分配感知和自适应副本选择配置。 但是，有时可能需要将某些搜索路由到某些分片副本。

查询字符串参数，可以设置为： _shards:2,3 限制对指定分片的操作。（在这种情况下是对分片2和3操作）。

```
GET <host>:<port>/<index_name>*/_search?pretty&preference=_shards:0,1
```

```
curl -X GET "localhost:19200/preference_index/_search?pretty&preference=_shards:0,1" -H
'Content-Type: application/json' -d'
> {
>   "query": {
>     "term": {
>       "hobby": {
>         "value": "cloud"
>       }
>     }
>   }
> }
> ';
{
  "took" : 239,
  "timed_out" : false,
  "terminated_early" : false,
  "_shards" : {
    "total" : 1,
    "successful" : 1,
    "skipped" : 0,
    "failed" : 0
  },
  "_clusters" : {
    "total" : 1,
    "successful" : 1,
    "skipped" : 0
  },
  "hits" : {
    "total" : 2,
    "max_score" : 0.48436219,
    "hits" : [
      {
        "_index" : "preference_index",
        "_type" : "default_type_",
        "_id" : "1",
        "_score" : 0.48436219,
        "_source" : {
          "name" : "Emma Edgar",
          "age" : 32,
          "hobby" : "Look through the moon to see cloud and snow",
          "job" : {
            "name" : "Product Manager",
            "salary" : 35000,
            "work desc" : "designing product about apples and pinapples"
          }
        }
      },
      {
        "_index" : "preference_index",
        "_type" : "default_type_",
        "_id" : "4",
        "_score" : 0.4132582,
        "_source" : {
          "name" : "Rivera Interpreter",
          "age" : 26,
          "hobby" : "The cloak of golden armour and dark silk has broken tens of thousands of
clouds",
          "job" : {
            "name" : "JavaScript Programmer",
            "salary" : 22000
          }
        }
      }
    ]
  }
}
```

4.10. 问题与说明

本小节主要介绍部分api使用时需要注意的一些事项。

4.10.1. 中文分词器：ik分词 的使用

Scope在2.0版本前支持调用的ik-plugin版本相对较低，在这些版本中，支持 analyzer=ik 但2.1 后ik分词器升级为v5.x+版本，因此沿用分词器内部的规则：移除名为 ik 的analyzer和tokenizer,使用 ik_smart 和

ik_max_word进行替换"

ik_max_word:会将文本做最细粒度的拆分，比如会将“中华人民共和国人民大会堂”拆分为“中华人民共和国、中华人民、中华、华人、人民共和国、人民、共和国、大会堂、大会、会堂等词语 ik_smart:会做最粗粒度的拆分，比如会将“中华人民共和国人民大会堂”拆分为中华人民共和国、人民大会堂。

4.10.2. Index and Mapping

Scope 支持创建index后，再put mapping的做法；但更推荐在创建index时直接带上mapping这类schema信息一并创建。这样做更贴合数据库本身建表的逻辑。

5. Transwarp Scope 运维

5.1. Transwarp Scope运维管理界面

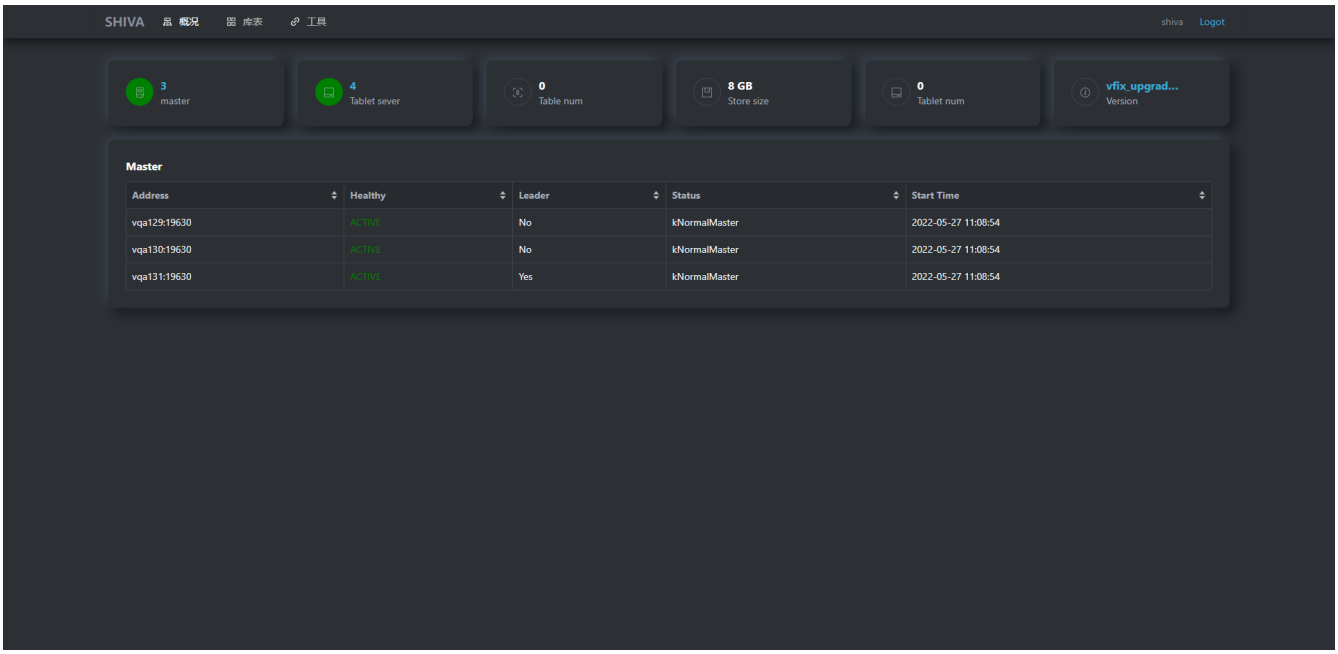
5.1.1. Webserver 页面

Webserver是Scope的运维监控界面,目前Scope提供新旧两个版本的监控来配合用户的使用习惯。

它的访问地址为： **新界面---http://<webserver_ip>/4567；传统界面---http://<webserver_ip>/4567/web1**
下面主要介绍新版本的界面情况，老版本可参考1.x版本使用手册。

Webserver主要由以下几个部分构成

5.1.1.1. 概况



页面展示了Scope集群的基本信息，包括：

1. Master status

   当前的active master，master group，master address，master的健康状态

2. Tabletserver Status

   tabletserver address,健康状态，逻辑机架和数据中心信息，容量使用以及tablet个数。

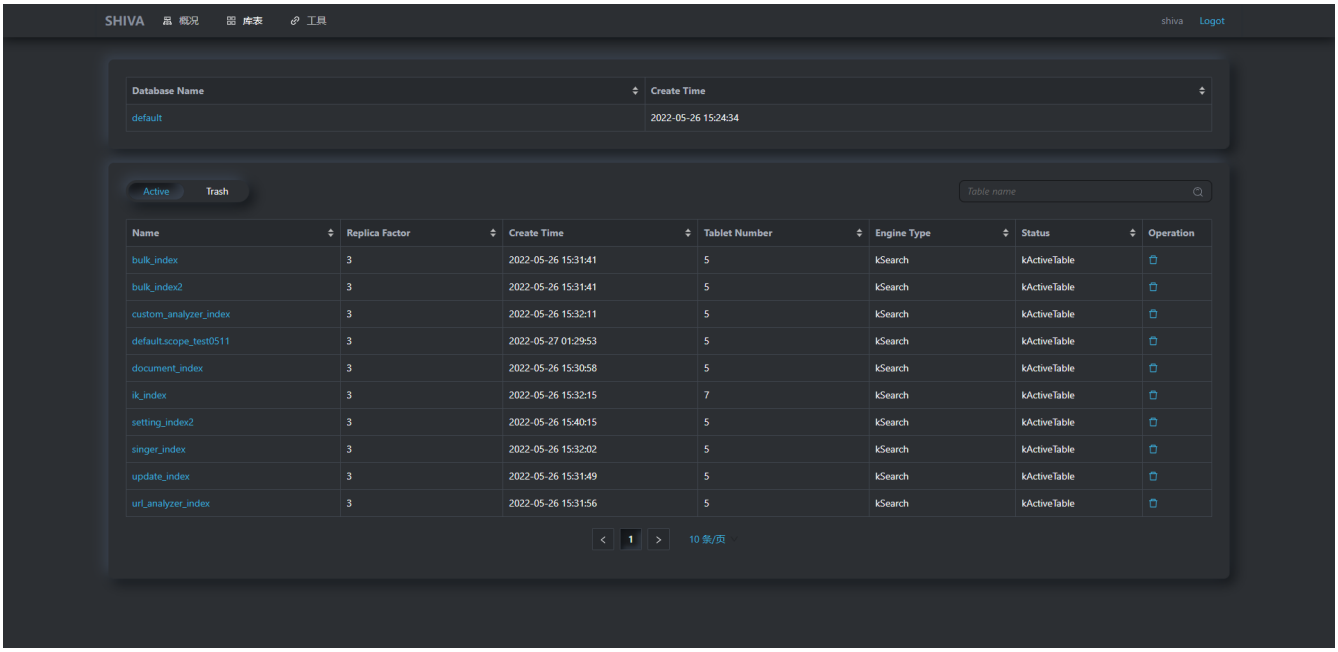
3. Table Num

   当前集群表的个数。

4. Shiva Build Info/Version

   TDDMS 版本信息

5.1.1.2. 库表



页面以库和表的概念集群存储的各类数据信息。

库信息

1. 库名。
2. 库创建时间。

库内的各类表的信息

1. 点击某个库，可以看到库下所有表的信息，主要包括：

   表的id

   表的名字

   表的状态

   表的tablet数量

   表的engine类型

   表的副本数

   表的创建时间
2. 支持针对表名和表的id进行（模糊）搜索。
3. 支持查看回收站内已经回收的表（正常删表操作表会进入回收站）

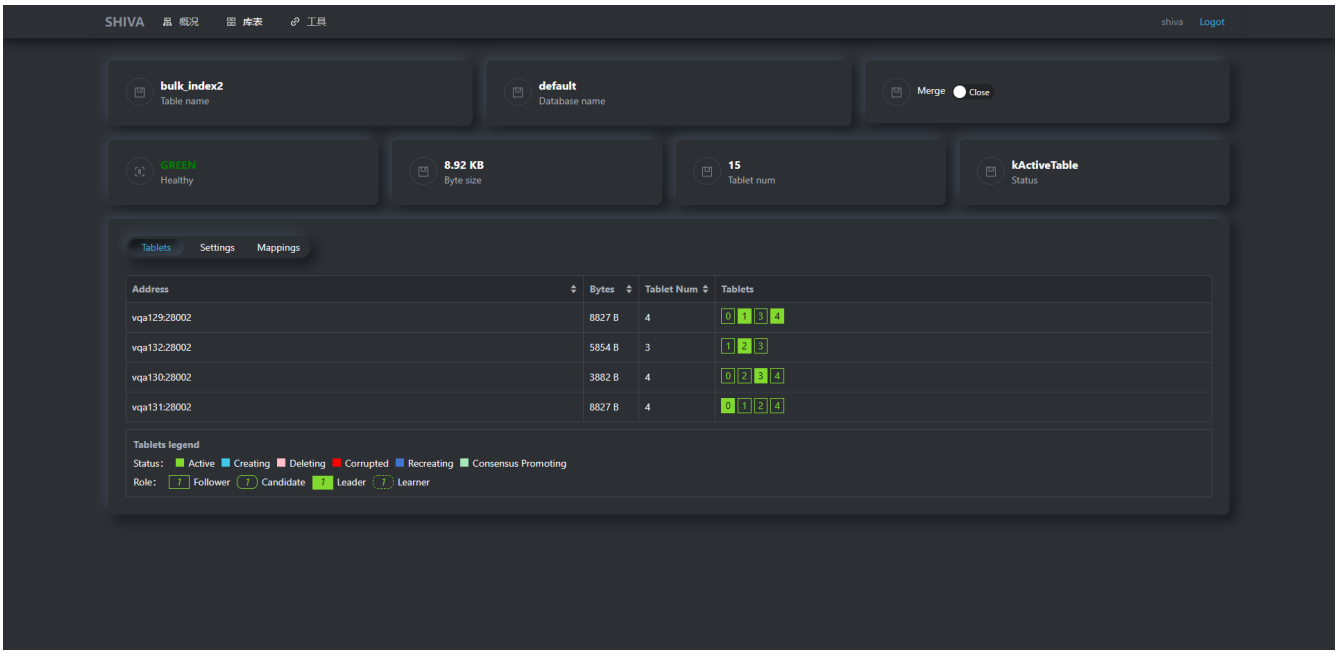


图 17. 表的详细信息

1. 点击库内任意表名，可以看到表的详细信息，主要信息包括：

    表的名字

    所属库

    表的状态

    表大小

    表的tablet数量

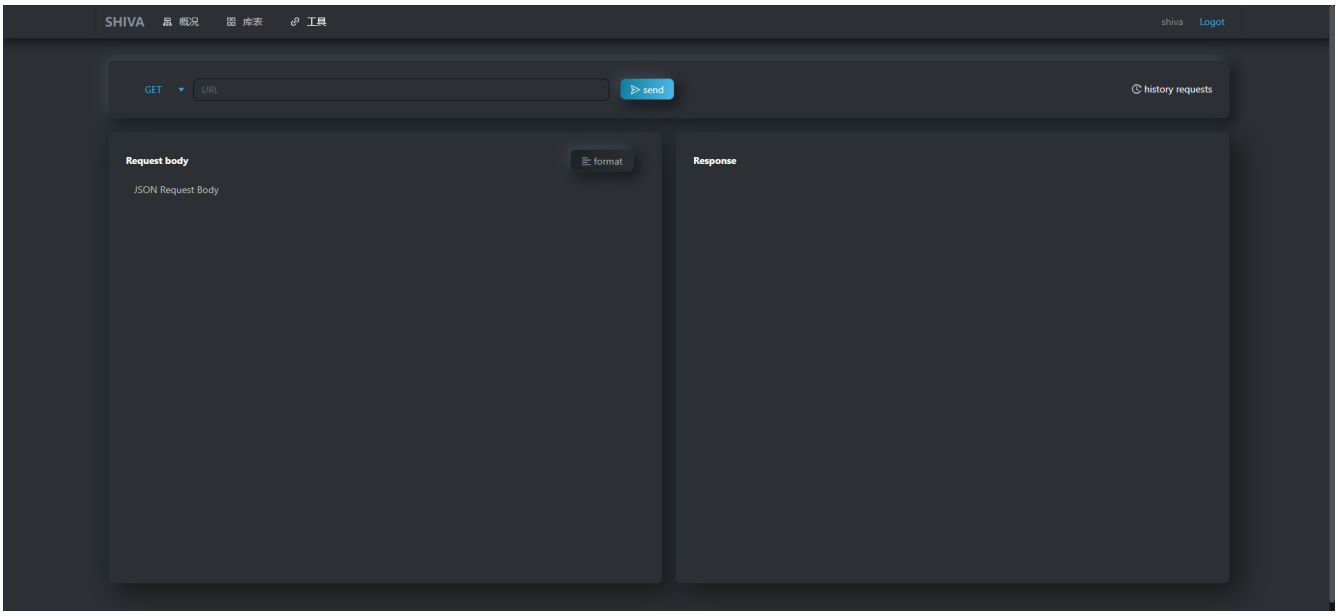
    表的副本数

    表的tablet 分布和状态

    表配置信息

    表的mapping（如果是基于search engine则有对应输出）

5.1.1.3. 工具



1. rest命令界面，支持界面化操作http rest命令与查看结果，以及缓存历史查询请求（仅支持页面刷新前保留）

5.1.2. Shiva Tool

Scope为了简化运维操作，同时提供界面化的几个运维功能用于集群上的操作

它的访问地址默认为 http://<webserver_ip>/4567/tool

以下为工具的基本视图：

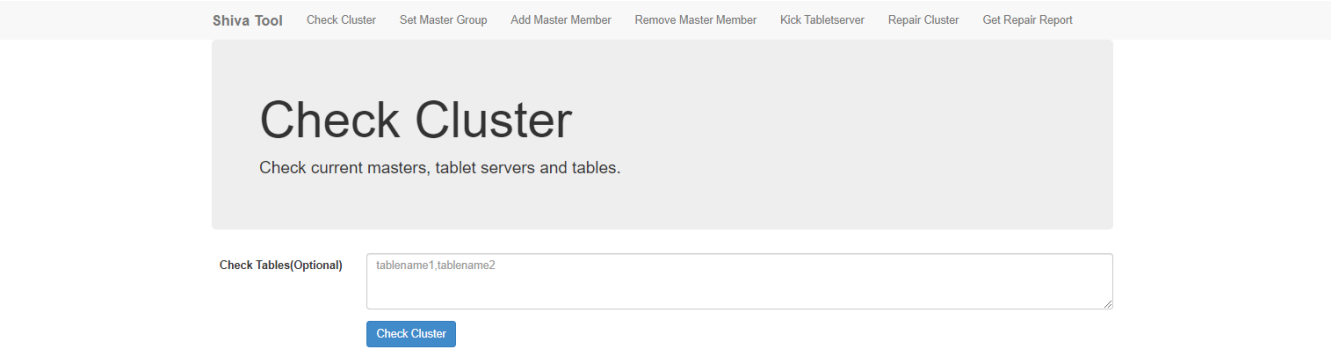


图 18. checkcluster

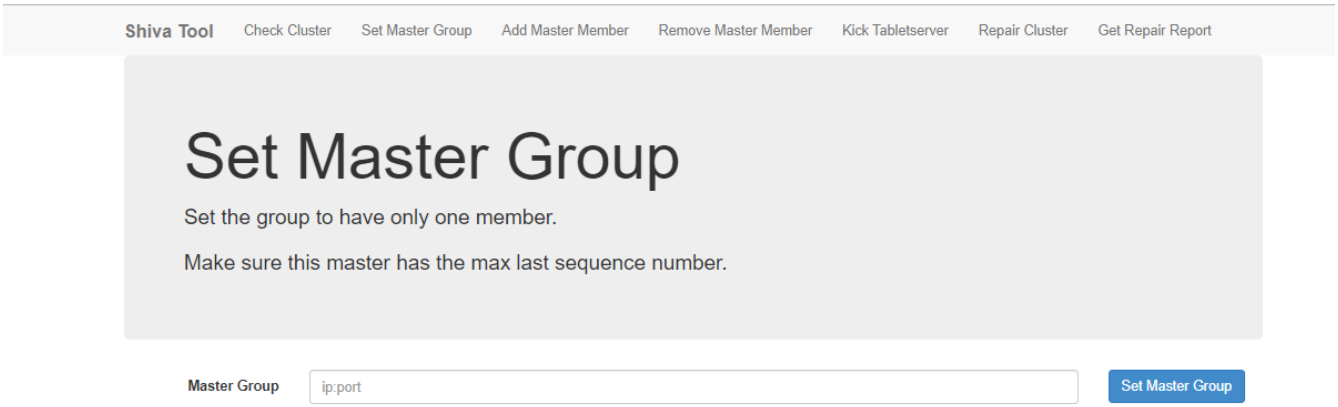


图 19. set MasterGroup

tool提供的操作均为 运维Restful API中介绍的的7类命令的界面操作方式

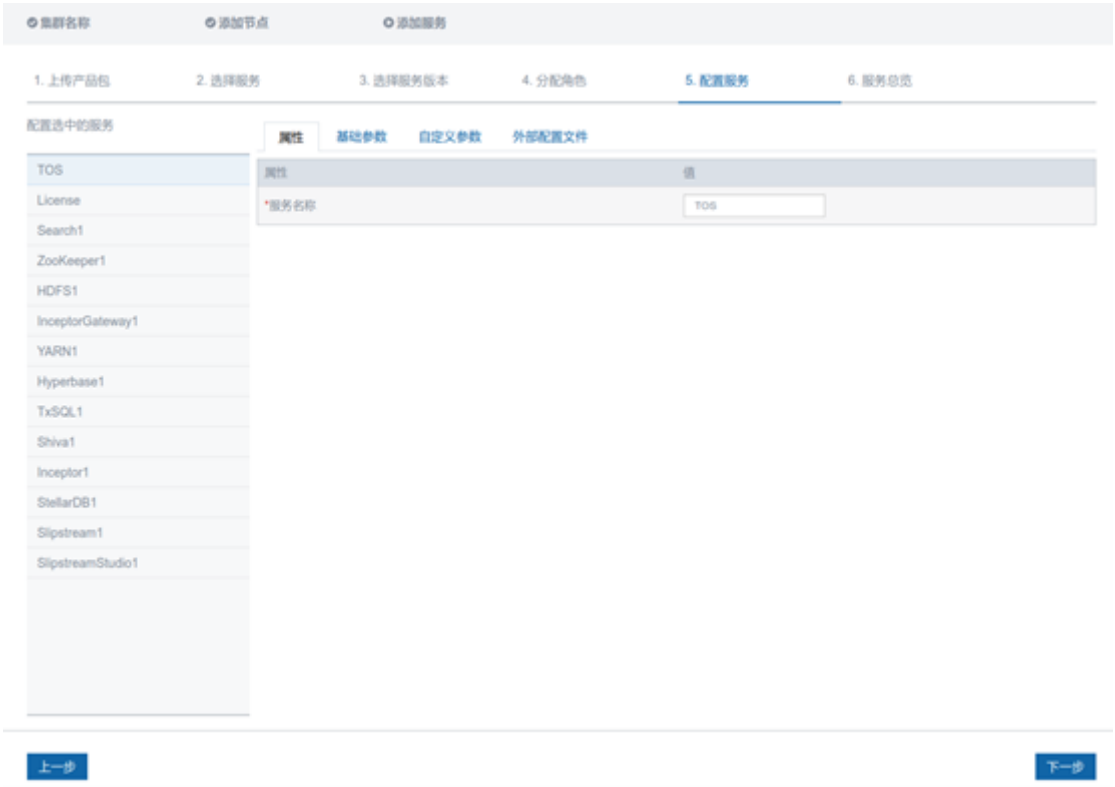
5.2. 常规运维操作

5.2.1. 节点扩容

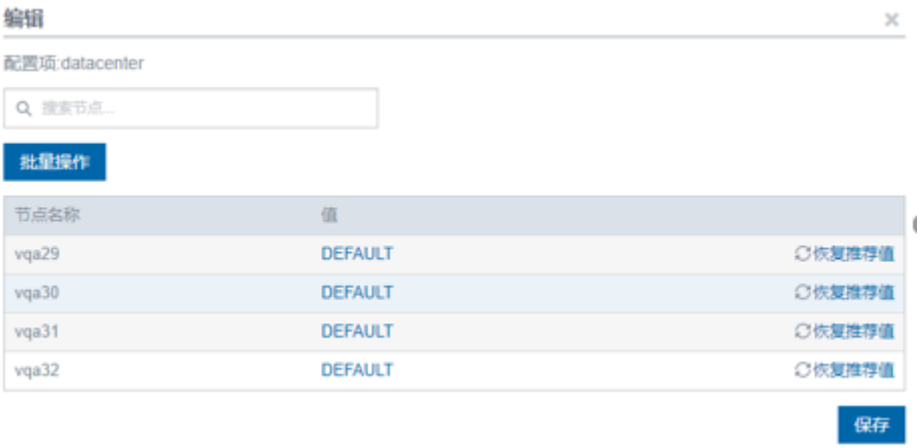
Scope集群的节点扩容和TDH一致，这里可以参阅TDH安装手册的《添加/删除集群服务器》章节。

但这里有额外的一些细节要注意：

1. 配置修改



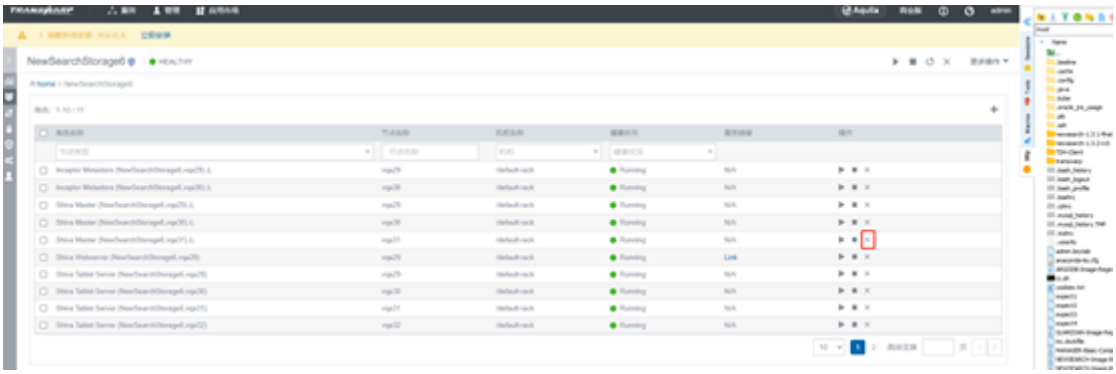
- 添加节点后服务添加的参数配置环节，需要对原集群修改过的参数重新配置，因为新节点会默认走默认值进行配置，比如服务的dir目录配置，memorysize的配置等



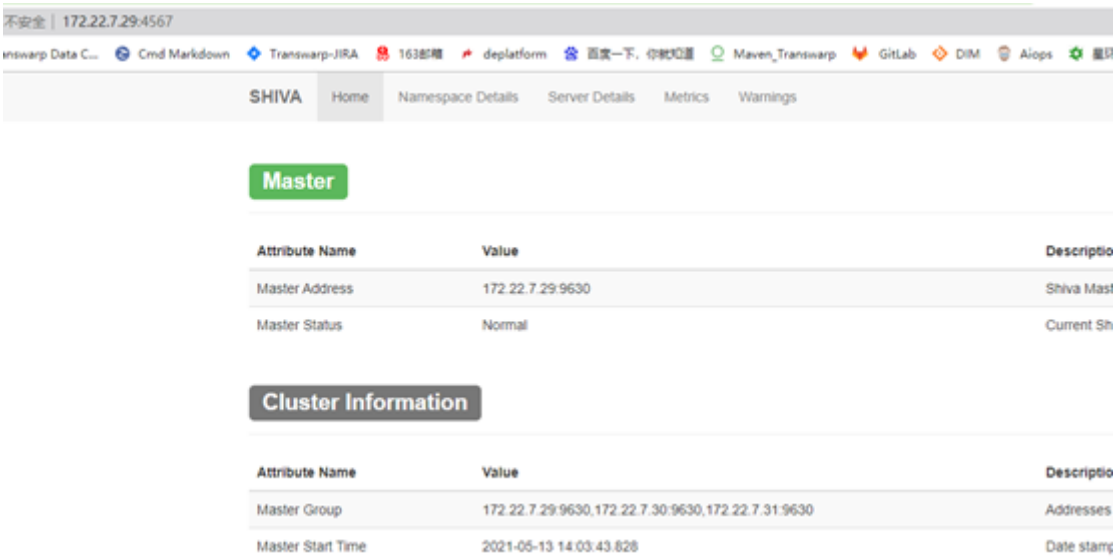
- 一些特殊参数如datacenter，shiva.topology.topology.rack这类参数需要在添加节点过程中进行设置，这些参数涉及到集群安装后的拓扑逻辑，无法二次变更生效

5.2.2. kick tabletserver/master

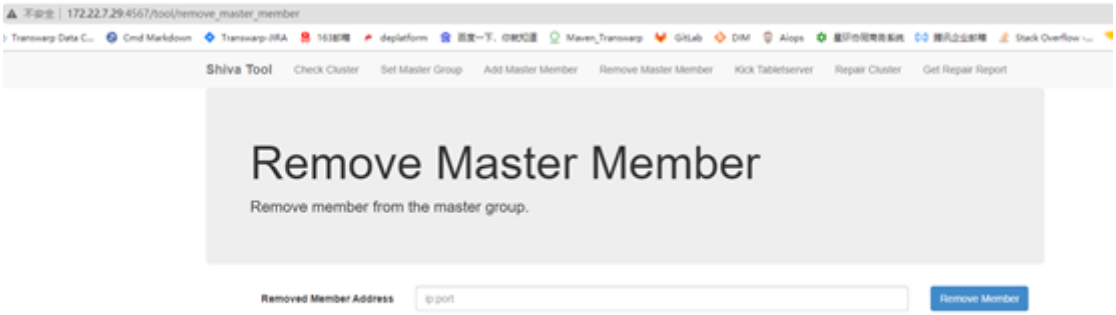
1. 在manager界面删除对应节点master服务



2. 进入到webserver界面，记录下要剔除的服务（master/tabletserver）的节点ip和端口



3. 进入shiva-tool工具,选择remove master/kick tabletserver菜单栏按照需求完成剔除操作



4. 删除服务后，在对应节点删除相关的配置和信息，一般数据目录位置为/mnt1/disk1/shiva-master-newsearchstorage1这种，目录可以参考配置文件找到具体对应目录删除即可

5.3. 集群修复

当多台机器同时故障时，可能会导致Shiva进入无法自动恢复的异常状态，此时需要运维人员人工介入，本节介绍如果使用shiva_tool手工修复集群。

5.3.1. 前置条件

尽量启动宕机的tabletserver/master，后续该文档将假设进行集群修复时，未启动的tabletserver/master已无法恢复。

5.3.2. 限制

Master必须至少存在一份副本，如所有Master副本丢失，集群将无法恢复。

5.3.3. 集群故障操作步骤

5.3.3.1. Step1.检查集群状态

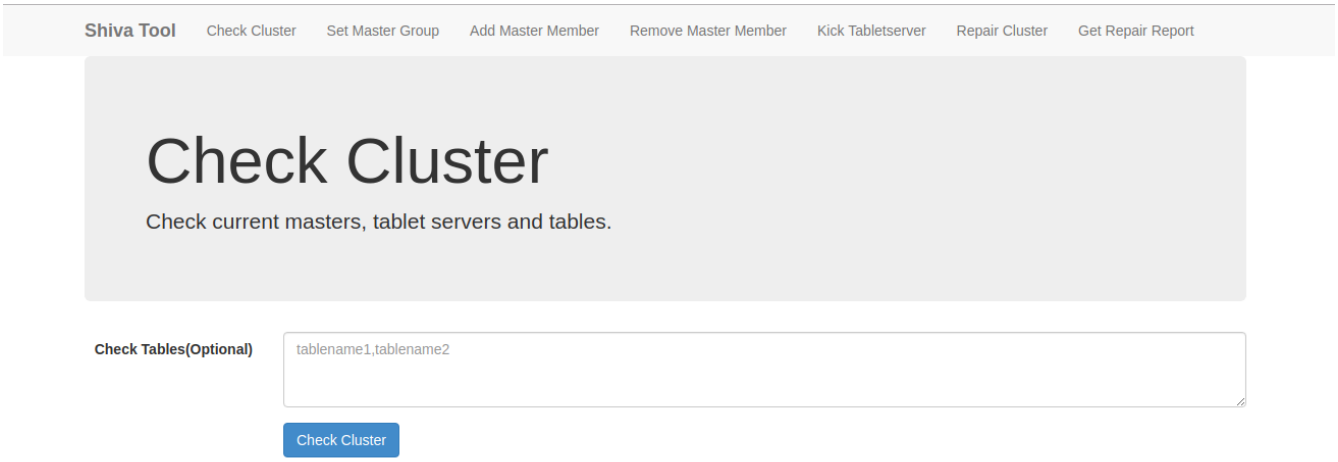
基于命令行方式的具体操作如下：

```
./shiva_tool --master_group=192.168.0.210:9630,192.168.0.211:9630,192.168.0.212:9630
--cmd=check_cluster
```

该命令执行完成后，默认将在当前目录下生成 **shiva.out** 文件，可以通过添加选项 **--check_output=${your_file_path}** 在不同文件路径下生成检查结果文件。

基于webui方式的具体操作如下：

通过webui shiva tool的check cluster界面完成。



对检查结果的解释：

1. 如结果文件显示 **status: kGreen** ，表示当前集群状态一切正常。
2. 如结果文件显示 **status: kYellow** ，表示当前集群服务正常，但存在异常情况，异常情况由 **warning** 项描述。注意，此时并不需要手工修复集群。
3. 如结果文件显示 **status: kRed** ，表示当前集群服务/部分服务已经异常，异常情况由 **fatal** 项描述。此时需要手工修复集群。

5.3.3.2. Step2.修复master

如集群检查结果显示 **fatal: "shiva master is not working properly"** ，则表示master已经无法正常提供服

务，master group已经只剩下少数副本；如果结果显示 **shiva master all dead, gg simida**，表示当前所有master副本已丢失，集群无法恢复；否则所有活着的master副本的 **last seq number** 将被显示在结果中。修复Master逻辑如下：

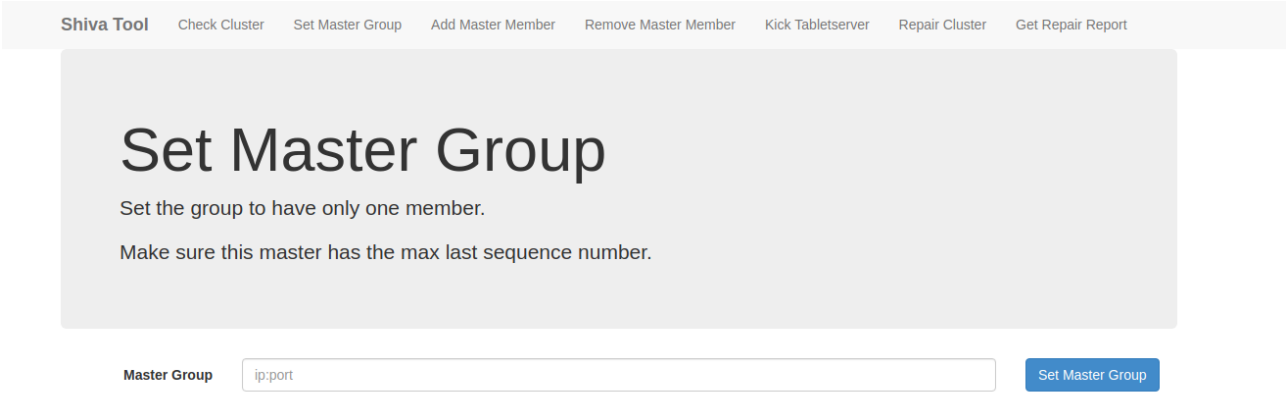
1. 从集群检查结果中，选择last seq number最大的master节点，设该master节点地址是 **192.168.0.210:9630** 。
2. 运行如下命令，将master group设置成只有 **192.168.0.210:9360** 一个成员。

   基于命令行方式的具体操作如下：

   ```
   ./shiva_tool --address=192.168.0.210:9630 --cmd=set_master_group
   ```

   基于webui方式的具体操作如下：

   在set master member界面完成设置master group。输入栏中填入指定master group即可。



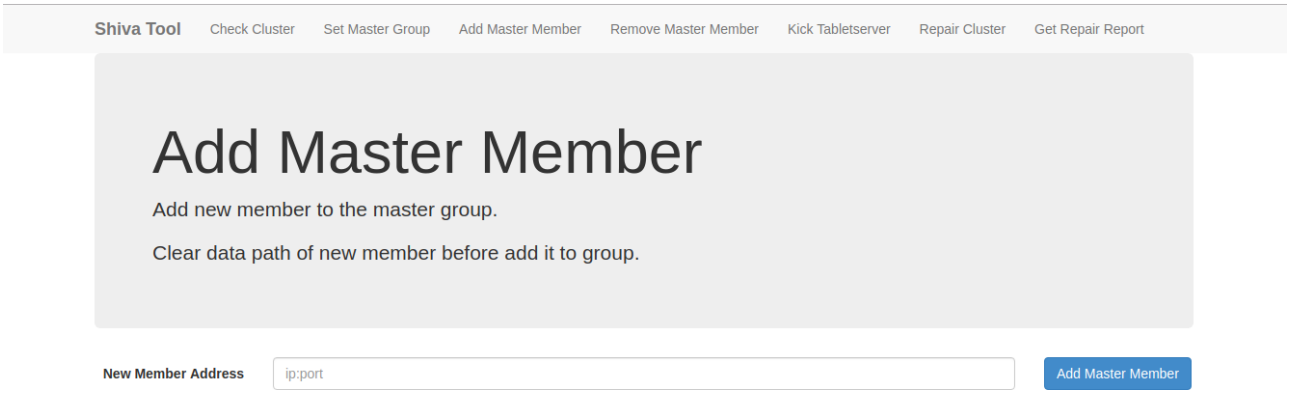
3. 将其余master(如果存在)停掉，清除master的 **data_path** ，并重新启动，作为新的master。
4. 将所有新master通过如下命令加入master group，尽量将master group恢复成至少三台。

   基于命令行方式的具体操作如下：

   ```
   ./shiva_tool --master_group=192.168.0.210:9630 --cmd=add_master_member --address=${new_address}
   ```

   基于webui方式的具体操作如下：

   通过add member界面可以将新的master加入master group。输入栏中填入新加master即可。



此时master修复完成，再次检查集群状态，此时应该不再出现 **shiva master is not working properly** 关键字；如集群状态为 **kGreen** 或 **kYellow** ，集群修复完成，否则进行Step3。

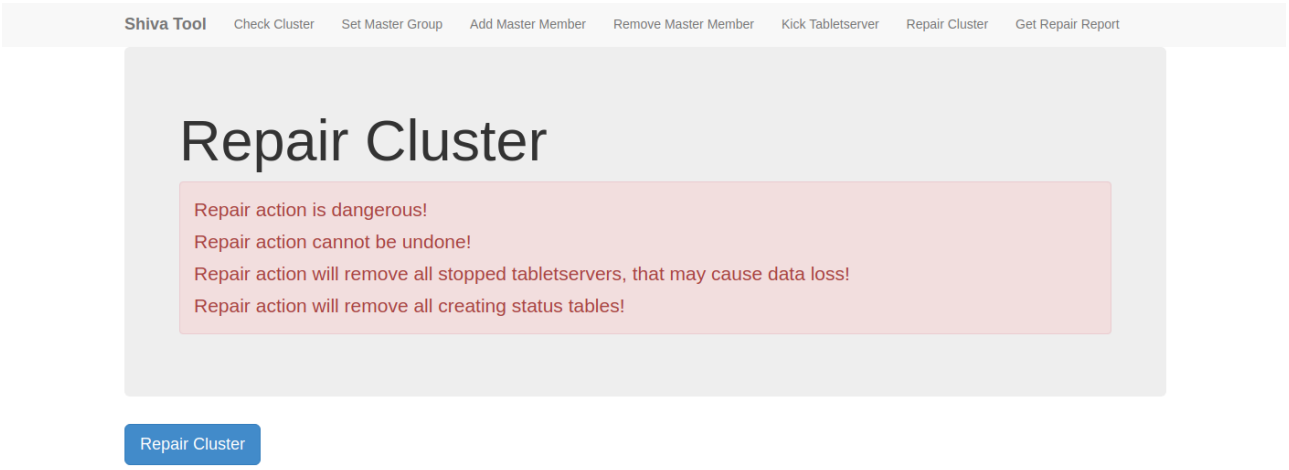
5.3.3.3. Step3.修复数据

1. 修复集群

   以下操作具有一定风险，请谨慎使用！

   基于webui方式的具体操作如下：

   通过点击repair cluster按钮进入修复集群页面。



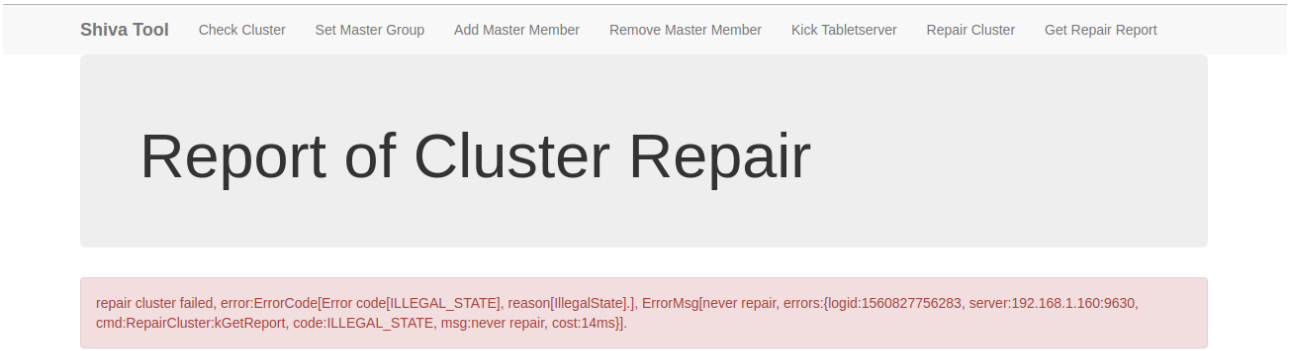
   点击Repair Cluster按钮即可触发集群修复

2. 获取集群修复信息

   默认生成在当前目录的 **shiva_repair_report.out** 文件中，通过 **--record_output** 参数指定文件路径。此时再次检查集群状态，得到结果应该是 **kGreen** 或者 **kYellow** 。

   基于webui方式的具体操作如下：

点击get repair report按钮获取集群修复信息。



5.3.4. 集群状态为yellow状态的处理过程

如果检查集群状态为 **kYellow** ，可能原因是：

1. 存在inactive tabletserver，如tabletserver无法恢复，使用 **kickserver** 将其从集群中去除即可。
2. 存在某些tablet副本数不足，这种情况不需要特殊处理。

5.3.5. 修复master操作

当Master group只剩少数节点存活时，Master无法提供服务，为了保证Master之间的数据一致，修复逻辑是首先选择一台存活的Master，通过 **set_master_group** 让其成为一个只有自己的一个成员的Master group，之后再向该group加入新的master节点，直到master group有足够的节点，一般为3或5。

5.3.6. 修复集群操作

修复集群通过master的 **repaircluster** 功能完成，其逻辑是：

1. 设置Master状态为kPreRepairMaster
2. 等待Master上所有运行的任务停止
3. 设置Master状态为kGcForRepairCluster
4. 收集所有tabletserver的实际tablet信息，处理master与tabletserver之间的不一致情况；GetAllTablets失败的tabletserver会被从集群中移除
5. 设置Master状态为kRepairingMaster
6. 删除所有处于creating状态的表
7. 对活跃副本数少于一半的tablet，将raft group设置成只包括当前存活的副本；对丢失所有副本的tablet，重建一个副本
8. 设置Master状态为kNormalMaster，修复完成
9. 如果需要，开始进行常规的recovery操作。

repaircluster 动作一旦开始，会一直进行下去直到完成，期间无论Master是否发生leader角色的切换，是否宕机重启。

5.4. 运维RestFul API

本节内容会详细介绍一些在Scope集群运维过程中经常或者可能使用到一些api。以此帮助大家更好的掌握集群情况并进行运维。

curl -X GET <host>:<port>/help可以获取使用说明;
在query string中加入pretty可以使结果信息更加美观易读，例如：`curl -X GET <host>:<port>/tables?pretty`

url结尾含有'/'是不被允许的，例如：`curl -X GET <host>:<port>/table/`将会报错

5.4.1. 库相关操作

1. 列出所有database：**curl -X GET "<host>:<port>/database"**

```
[root@vqa129 ~]# curl -ushiva:shiva -XGET 172.22.7.130:14567/database?pretty
{
  "log_id" : 1663139747723,
  "code" : 0,
  "msg" : "ok",
  "databases" : [ {
    "id" : "00000000000000000000000000000000",
    "name" : "default",
    "creation_date_time" : "2022-08-31 09:53:54"
  } ]
}
```

2. 删除database：**curl -X DELETE "<host>:<port>/database/<namespace>"**

不允许删除默认命名空间：default。

5.4.2. 表相关操作

由于篇幅有限，不会将命令的全部结果显示出来，仅会显示部分结果以示参考。

1. 列出所有表：**curl -X GET "<host>:<port>/table?database=<database>&option=all"**

```
[root@vqa129 ~]# curl -ushiva:shiva -XGET
"172.22.7.130:14567/table?database=default&option=all"
{"log_id": 1663139747921,"code": 0,"msg": "ok","tables": [{"name":
"4f97fc97fab94fbe90a66632e7adcffc","status": "kActiveTable","engine_type":
"kSearch","tablet_num": 5,"replication_factor": 3,"creation_date_time": "2022-09-20
12:19:25","id": "0a1d514bc72b44d28221eb2633731acc","type":
"kMainTable","deletion_date_time": "2022-09-20 12:30:33","is_partition_table": true}...]}
```

2. 列出所有被删除的在回收站中的表 **curl -X GET "<host>:<port>/table?database=<database>&option=deleted"**

```
[root@vqa129 ~]# curl -ushiva:shiva -XGET
"172.22.7.130:14567/table?database=default&option=deleted"
{"log_id": 1663139747933,"code": 0,"msg": "ok","tables": [{"name":
"4f97fc97fab94fbe90a66632e7adcffc","status": "kActiveTable","engine_type":
"kSearch","tablet_num": 5,"replication_factor": 3,"creation_date_time": "2022-09-20
12:19:25","id": "0a1d514bc72b44d28221eb2633731acc","type":
"kMainTable","deletion_date_time": "2022-09-20 12:30:33","is_partition_table": true}...]}
```

3. 列出所有常规状态表：**curl −X GET "<host>:<port>/table?database=<database>&option=active"**

```
[root@vqa129 ~]# curl -ushiva:shiva -XGET
"172.22.7.130:14567/table?database=default&option=active"
{"log_id": 1663139747948,"code": 0,"msg": "ok","tables": [{"name":
"06923f6cb37a4862b49733bed6070bb2","status": "kActiveTable","engine_type":
"kSearch","tablet_num": 2,"replication_factor": 3,"creation_date_time": "2022-09-19
19:31:52","id": "d76b639aebf84e0c82c49d3399c12cc8","type":
"kMainTable","is_partition_table": true}...]}
```

4. 获取表的详细信息：**curl −X GET "<host>:<port>/table/description?database=<database>&table_name=<table_name>"**

```
[root@vqa129 ~]# curl -ushiva:shiva -XGET
"172.22.7.130:14567/table/description?database=default&table_name=default.scope_test"
{"table_description": {"identifier": {"table_id":
"ea412ac49dc7403882ed9d478376c8b7","name": {"name": "default.scope_test","database":
"default"}},"version": 0,"schema_version": 0,"properties": {"engine_type":
"kSearch","type": "kMainTable","dialect": "ORACLE","creation_date_time_seconds":
1662701885,"creator": {"identifier": "4fec24f5-7808-417f-ac0d-
5656255a0dde"},"creation_date_time": "2022-09-09 13:38:05","partition_management_type":
"kEngineManagementType","is_replicated_table": false},"schema": {"columns": [{"id":
0,"name": "_uid","type": "STRING","attributes": {"case_sensitivity":
"CASESPECIFIC"},"constraints": {"not_null": true,"unique": true,"primary_key":
true}...}]}}}
```

5. 获取表的分布信息：**curl −X GET "<host>:<port>/table/location?database=<database>&table_name=<table_name>"**

```
[root@vqa129 ~]# curl -ushiva:shiva -XGET
"172.22.7.130:14567/table/location?database=default&table_name=default.test"
{"log_id": 1663657458027,"code": 0,"msg": "ok","table_desc": {"identifier": {"table_id":
"ecd4f4f77ed54d13b6b0ffacf8481b36","name": {"name": "default.test","database":
"default"}},"version": 0,"schema_version": 0,"properties": {"engine_type":
"kSearch","type": "kMainTable","dialect": "ORACLE","creation_date_time_seconds":
1663243811,"creator": {"identifier": "5fafeef3-6a13-46ad-8365-
dabee12d1ed2"},"creation_date_time": "2022-09-15 20:10:11","partition_management_type":
"kEngineManagementType","is_replicated_table": false},"schema": {"columns": [{"id":
0,"name": "_uid","type": "STRING","attributes": {"case_sensitivity":
"CASESPECIFIC"},"constraints": {"not_null": true,"unique": true,"primary_key":
true},"extension": {"shiva.engine.search.column_extension": {"search_field_id":
0,"fundamental_properties": {},"meta_column_pb": {"name": "_uid","enabled":
true},"search_data_type": "string","nested": {"nested": false,"includeInParent":
false,"includeInRoot": false}}},"is_virtual": false}...}}}
```

6. 删除表。删除后表进入回收站，并将在7天后被彻底从回收站清除：**curl −X DELETE "<host>:<port>/table?database=<database>&table_name=<table_name>"**

```
[root@vqa129 ~]# curl -ushiva:shiva -XDELETE
"172.22.7.130:14567/table?database=default&table_name=4f97fc97fab94fbe90a66632e7adcffc"
{
  "code": "OK",
  "msg": "ok"
}
```

7. 更新表的tag：**curl −X PUT "<host>:<port>/table/tag/<tag>?database=<database>&table_name=<table_name>"**

```
[root@vqa129 ~]# curl -ushiva:shiva -XPUT
"172.22.7.130:14567/table/tag/hot?database=default&table_name=default.scope_test"
{"log_id": 1663657458162,"code": 0,"msg": "table:default.scope_test tag updated"}
```

8. 移除标的tag：**curl −X DELETE "<host>:<port>/table/tag?database=<database>&table_name=<table_name>"**

```
[root@vqa129 ~]# curl -ushiva:shiva -XDELETE
"172.22.7.130:14567/table/tag?database=default&table_name=default.scope_test"
{"log_id": 1663657458169,"code": 0,"msg": "table:default.scope_test tag updated"}
```

9. 更新表的副本数：**curl −X PUT "<host>:<port>/table/replica_factor/<replica_factor>?database=<database>&table_name=<table_name>"**

```
[root@vqa129 ~]# curl -ushiva:shiva -XPUT
"172.22.7.130:14567/table/replica_factor/5?database=default&table_name=default.scope_test"
{"log_id": 1663657458211,"code": 0,"msg": "table:default.scope_test replication factor
updated"}
```

副本个数推荐为奇数个3或者5，1副本和偶数副本数不推荐使用，无法满足内部的多数派选举机制而报错

10. 获取tablet的分布信息：**curl −X GET "<host>:<port>/table/tablet/location/?database=<database>&table_name=<table_name>&partition_key=<key>"**

```
[root@vqa129 ~]# curl -ushiva:shiva -XGET
"172.22.7.130:14567/table/location?database=default&table_name=default.test&partition=3"
{"log_id": 1663657458241,"code": 0,"msg": "ok","table_desc": {"identifier": {"table_id":
"ecd4f4f77ed54d13b6b0ffacf8481b36","name": {"name": "default.test","database":
"default"}},"version": 0,"schema_version": 0,"properties": {"engine_type":
"kSearch","type": "kMainTable","dialect": "ORACLE","creation_date_time_seconds":
1663243811,"creator": {"identifier": "5fafeef3-6a13-46ad-8365-
dabee12d1ed2"},"creation_date_time": "2022-09-15 20:10:11","partition_management_type":
"kEngineManagementType","is_replicated_table": false},"schema": {"columns": [{"id":
0,"name": "_uid","type": "STRING","attributes": {"case_sensitivity":
"CASESPECIFIC"},"constraints": {"not_null": true,"unique": true,"primary_key":
true},"extension": {"shiva.engine.search.column_extension": {"search_field_id":
0,"fundamental_properties": {},"meta_column_pb": {"name": "_uid","enabled":
true},"search_data_type": "string","nested": {"nested": false,"includeInParent":
false,"includeInRoot": false}}},"is_virtual": false},{"id": 1,"name": "_source","type":
"BINARY","attributes": {"case_sensitivity": "CASESPECIFIC"}....}]}}}
```

5.4.3. 回收站相关操作

1. 恢复回收站中的表。如果原表名已被占用，请使用new_table_name作为为恢复后表的新表名：**curl -X PUT "<host>:<port>/trash?table_id=<id>&new_table_name=<name>"**

```
[root@vqa129 ~]# curl -ushiva:shiva -XPUT
"172.22.7.130:14567/trash?table_id=0fad53add57f4578a3d3d82d0dbbf633&new_table_name=abc"
{"log_id": 1663657458268,"code": 0,"msg": "ok"}
```

2. 彻底删除回收站中的表：**curl -X DELETE "<host>:<port>/trash?table_id=<id1>&table_id=<id2>&…table_id=<idN>"**

```
[root@vqa129 ~]# curl -ushiva:shiva -XDELETE
"172.22.7.130:14567/trash?table_id=0a1d514bc72b44d28221eb2633731acc"
{"log_id": 1663657458292,"code": 0,"msg": "ok"}
```

3. 清空回收站中的表：**curl -X DELETE "<host>:<port>/trash/all?database=<db_name>"**

```
[root@vqa129 ~]# curl -ushiva:shiva -XDELETE
"172.22.7.130:14567/trash/all?pretty&database=default"
{  "code": "OK",
   "msg": "cleaned table num:5"
}
```

5.4.4. Tabletserver相关操作

1. 列出所有tabletserver：**curl -X GET "<host>:<port>/server"**

```
[root@vqa129 ~]# curl -ushiva:shiva -XGET "172.22.7.130:14567/server"
{"log_id": 1663657458964,"code": 0,"msg": "ok","servers": [{"id":
"f890862d9d934f40ac869754e395ebc6","topology": {"address": "vqa131:18002","rack":
"rack1","datacenter": "DEFAULT"},"status": "kActiveServer","commission_status":
"kCommissioned","server_statistic": {"tablets_num": 1211,"total_store_capacity_units":
898,"used_store_capacity_units": 8,"stores": [{"id": 0,"tablets_num": 606,"capacity_units":
449...}]}}]}
```

2. 剔除tabletserver：**curl -X DELETE "<host>:<port>/server/<server_address>"**

```
[root@vqa29 ~]# curl -X DELETE "vqa30:4567/server/172.22.7.29:8002"
```

5.4.5. 告警相关操作

1. 列出当前的告警信息：**curl -X GET "<host>:<port>/warning"**

```
[root@vqa129 ~]# curl -ushiva:shiva -XGET "172.22.7.130:14567/warning"
{"log_id": 1663657459225,"code": 0,"msg": "ok","table_warns": [{"table_identifier":
{"table_id": "ea412ac49dc7403882ed9d478376c8b7","name": {"name":
"default.scope_test","database": "default"}},"under_replica_tablets": [{"tablet_id":
0,"replica_num": 4},{"tablet_id": 1,"replica_num": 4},{"tablet_id": 2,"replica_num":
4},{"tablet_id": 3,"replica_num": 4},{"tablet_id": 4,"replica_num":
4}]}],"change_config_ongoing": true}
```

5.4.6. 集群信息相关操作

1. get tddms cluster info: **curl -X GET "<host>:<port>/cluster_info"**

```
[root@vqa129 ~]# curl -ushiva:shiva -XGET "172.22.7.130:14567/cluster_info"
{
  "master_group": "vqa130:19630,vqa131:19630,vqa132:19630",
  "master_start_time": "2022-09-20 16:20:21",
  "shiva_build_info": {
    "version_": [
      {
        "bitField0_": 7,
        "name_": {
          "bytes": [
            65...}
```

2. trigger gc: **curl -X POST "<host>:<port>/cluster/gc"**

```
[root@vqa129 ~]# curl -ushiva:shiva -XPOST "172.22.7.130:14567/cluster/gc"
{"log_id": 1663657461184,"code": 0,"msg": "ok"}
```

3. trigger leader/data balance : **curl -X POST "<host>:<port>/cluster/balance?action=leader|data"**

```
[root@vqa129 ~]# curl -X POST "172.22.7.130:14567/cluster/balance?action=data"
{"log_id": 1663825591308,"code": 0,"msg": "ok"}
```

5.4.7. Master group相关操作

1. 剔除成员: **curl -X DELETE "<host>:<port>/master_group/<master_address>"**

```
[root@vqa29 ~]# curl -X DELETE "vqa30:4567/master_group/172.22.7.30:9630"
{"log_id": 1621911871393,"code": 0,"msg": "ok"}
```

2. 指定master group: **curl -X PUT "<host>:<port>/master_group/<master_address>"**

```
[root@vqa29 ~]# curl -X PUT
"vqa30:4567/master_group/172.22.7.29:9630,172.22.7.30:9630,172.22.7.31:9630"
```

3. 添加成员：**curl -X POST "<host>:<port>/master_group/<master_address>"**

```
[root@vqa29 ~]# curl -X POST "vqa30:4567/master_group/172.22.7.29:9630"
```

5.5. Scope安全

Scope产品同样支持基于权限的认证和访问，本章节会介绍产品在安全模式下的集群用户信息管理和各项服务的使用

在本手册中，我们将称部署并启用了认证系统(包括Kerberos、LDAP或者Guaridan)的集群为“安全模式”下的集群。在安全模式下的集群，用户需要通过认证才能够使用服务。如果集群没有部署或者部署了但没有启用任何认证系统，那么我们称其为“非安全模式”下的集群。

5.5.1. 开启安全与访问

Scope安全默认情况下是开启的，不会关闭，但是提供了匿名用的配置来操作集群的访问情况，可以通过在manager的scope参数配置界面上进行参数修改与操作，参数如下:

- master.shiva_permit_no_negotiated_user = true/false(default:true)

  true，表示scope允许匿名用户访问scope，此时scope对所有的用户操作放开权限

  false，表示scope不允许匿名用户访问scope，此时等同于用户的权限限制完全开启

在使用scope sql时，quark对scope的访问同样受到权限的限制，具体相关的配置可以参考《Scope SQL相关配置》章节的内容。

5.5.2. 用户权限

scope的用户的用户有以下几种类型：

- 默认用户

  shiva用户，具备最高权限的内置用户，生产环境以以下两种用户为主。

  `Super User`：超级用户，通过默认用户创建。

  `Normal User`：常规用户，可以通过超级用户和默认用户创建。

5.5.3. 用户操作命令

5.5.3.1. 用户创建

创建用户相关语法

```
curl -u user:password -XPUT ip:port/_security/user/<new_user>?pretty -H 'Content-Type:
application/json' -d '{
  "password" : "<new_user_password>",
  "metadata" : {
    "is_super" : true/fasle
    [,"can_create_role": true/fale]
  }
}';
is_super：用于控制创建的用户是超级用户或者常规用户
can_create_role:用于给用户赋权是否拥有创建用户的权限。默认超级用户支持创建用户，常规用户则无此项权限权限
```

例 53. 创建用户

```
创建超级用户super_test
curl -u shiva:shiva -XPUT localhost:9200/_security/user/super_test?pretty -H 'Content-Type:
application/json' -d '{
  "password" : "super_test",
  "metadata" : {
    "is_super" : true
  }
}';

创建常规用户test
curl -u shiva:shiva -XPUT localhost:9200/_security/user/test?pretty -H 'Content-Type:
application/json' -d '{
  "password" : "test",
  "metadata" : {
    "is_super" : false
  }
}';
```

5.5.3.2. 用户查看

查看用户相关语法

```
curl -u user:password -XPUT ip:port/_security/user?pretty
权限权限
```

例 54. 查看用户

```
curl -u shiva:shiva -XPUT localhost:9200/_security/user?pretty
```

5.5.3.3. 用户删除

查看用户相关语法

```
curl -u user:password -XDELETE ip:port/_security/<delete_user>?pretty
```

例 55. 删除用户

```
curl -u shiva:shiva -XDELETE localhost:9200/_security/user/test?pretty
```

5.5.3.4. 用户权限查看

查看权限语法

```
curl -u user:password -XGET ip:port/_security/user/_privileges?pretty
```

例 56. 删除用户

```
curl -u test:test -XGET localhost:9200/_security/user/_privileges?pretty
```

5.5.3.5. 用户权限修改与用户密码修改

修改用户密码与权限相关语法

```
curl -u user:password -XPUT ip:port/_security/user/<change_user>/_alter?pretty -H 'Content-Type:
application/json' -d '{
  "password" : "new_password",
  "metadata" : {
    "is_super" : true/false
  }
}';
```

例 57. 修改用户密码与权限

```
将super_test用户重置为常规用户并修改其密码
curl -u super_test:super_test -XPUT localhost:9200/_security/user/super_test/_alter?pretty -H
'Content-Type: application/json' -d '{
  "password" : "new_password",
  "metadata" : {
    "is_super" : false
  }
}';
```

5.5.4. 表权限操作命令

5.5.4.1. 表权限查看

查看表权限语法

```
curl -u user:password -XGET ip:port/_security/indices?pretty
```

例 58. 查看表/索引权限

```
curl -u super_test:super_test -XGET localhost:9200/_security/indices?pretty
```

5.5.4.2. 赋权

赋权语法

```
curl -u user:password -X POST "ip:port/_security/role/another_user?pretty" -H 'Content-Type:
application/json' -d'
{
  "indices": [
    {
      "names": [ "<table_name>"],
      "privileges": ["all"]
    }
  ]
}
';
```

例 59. 赋权

```
supertest用户给用户test赋予表testtable all权限，test用户对testtable具备READ/WRITE/DELETE权限

curl -u super_test:super_test -X POST "localhost:9200/_security/role/test?pretty" -H 'Content-
Type: application/json' -d'
{
  "indices": [
    {
      "names": [ "testtable"],
      "privileges": ["all"]
    }
  ]
}';
```

5.5.5. 权限认证与java API

除却Rest api外，通过JAVA 访问也要遵循权限访问的相关规则。这里通过demo提供具体的使用方法。

5.5.5.1. 登录认证

登录认证

```
    CredentialsProvider credentialsProvider = new BasicCredentialsProvider();
    credentialsProvider.setCredentials(AuthScope.ANY,
        new UsernamePasswordCredentials("test", "test"));//用户名与密码
    RestHighLevelClient restHighLevelClient = new RestHighLevelClient(
        RestClient.builder(
            new HttpHost("localhost", 9200, "http")//对应adapter的地址与端口
        ).setHttpClientConfigCallback(f -> f.setDefaultCredentialsProvider(credentialsProvider)));
```

5.5.5.2. 用户创建

用户创建

```
    // 构建RestHighLevelClient
    CredentialsProvider credentialsProvider = new BasicCredentialsProvider();
    credentialsProvider.setCredentials(AuthScope.ANY,
        new UsernamePasswordCredentials("test", "test"));
    RestHighLevelClient restHighLevelClient = new RestHighLevelClient(
        RestClient.builder(
            new HttpHost("localhost", 9200, "http")
        ).setHttpClientConfigCallback(f -> f.setDefaultCredentialsProvider(credentialsProvider)));

    // 构建新增用户请求
    char[] password = new char[]{'p', 'a', 's', 's', 'w', 'o', 'r', 'd'};//密码的要求是char数组
    Map<String, Object> metadata = new HashMap<>();
    metadata.put("is_super", false);//super权限赋予
    metadata.put("can_create_role", true);//创建用户的权限赋予
    User user = new User("newTest", Collections.emptyList(), metadata, null, null);//用户名
    PutUserRequest request = PutUserRequest.withPassword(user, password, true, RefreshPolicy.NONE);

    // 发送请求，获取结果
   PutUserResponse response = restHighLevelClient.security().putUser(request,
RequestOptions.DEFAULT);
```

5.5.5.3. 赋权

赋权

```
// 构建RestHighLevelClient
CredentialsProvider credentialsProvider = new BasicCredentialsProvider();
credentialsProvider.setCredentials(AuthScope.ANY,
    new UsernamePasswordCredentials("test", "test"));
RestHighLevelClient restHighLevelClient = new RestHighLevelClient(
    RestClient.builder(
        new HttpHost("localhost", 9200, "http")
    ).setHttpClientConfigCallback(f -> f.setDefaultCredentialsProvider(credentialsProvider)));

// 构建请求--用户test为newtest用户赋权index1的all权限
final Role role = Role.builder()
      .name(newTest)
      .indicesPrivileges(IndicesPrivileges.builder()
          .indices("index1").privileges(Role.IndexPrivilegeName.ALL).build())
      .build();
  final PutRoleRequest request = new PutRoleRequest(role, RefreshPolicy.NONE);

// 发送请求，获取结果
PutRoleResponse response = client.security().putRole(request, RequestOptions.DEFAULT);
```

6. TDDMS运维文档

6.1. 安装

6.1.1. Master安装

Master负责存储和管理元信息。

6.1.1.1. 角色分配

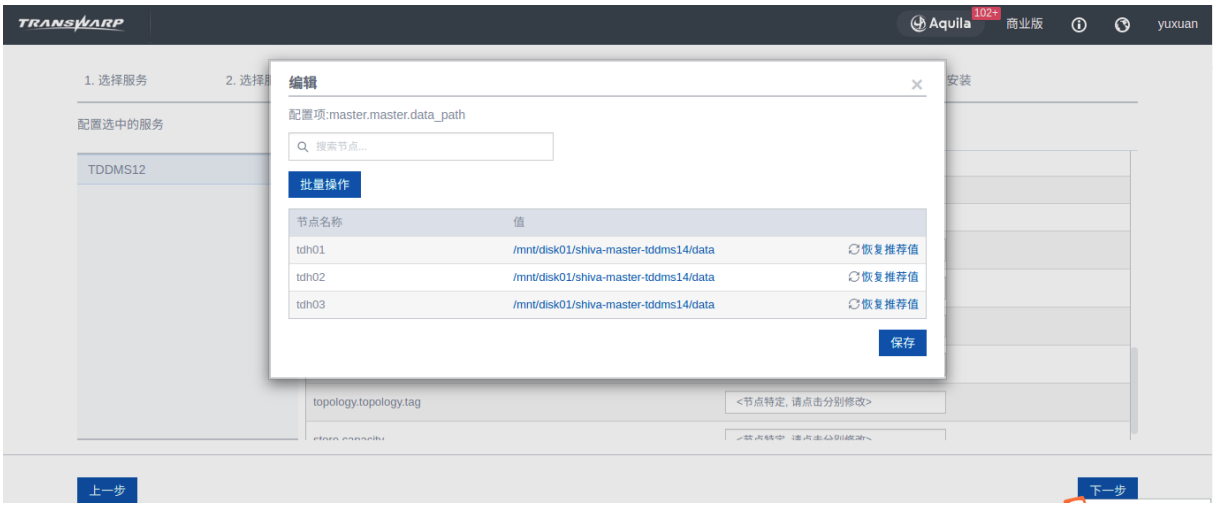
安装时要求至少安装1个，实际生产环境推荐安装3或5个。

6.1.1.2. 配置服务

以下参数请在安装时进行检查和确认：

master.master.datapath：用于存储数据。

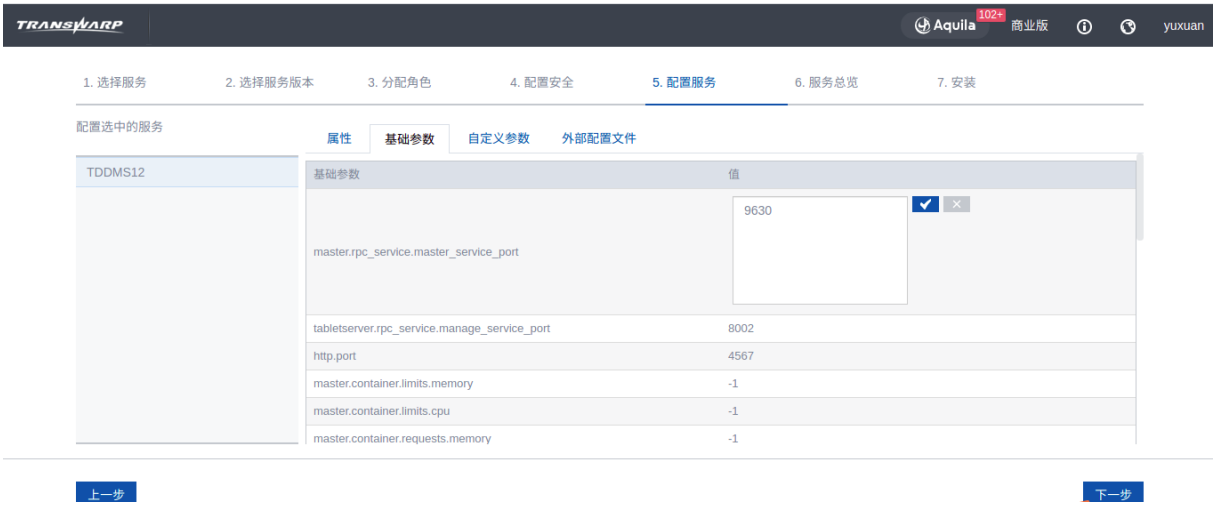
注意：通常情况下数据目录会分配在数据盘而不是根分区。因为存在一定概率会出现数据目录错误地被挂载到根分区的情况，所以请务必确认配置的数据目录的挂载点不是根分区。



master.rpcservice.masterserviceport：用于和其他节点通信。安装后会占用从配置值开始的连续4个端

口。例如安装时配置为9630，则实际会使用9630-9633这4个连续端口。务必在安装时确认端口没有冲突。

注意：端口在安装后不能变更。请在安装时做好规划，避免出现端口冲突。



6.1.2. Tabletserver安装

Tabletserver负责存储和管理数据分片副本。

6.1.2.1. 角色分配

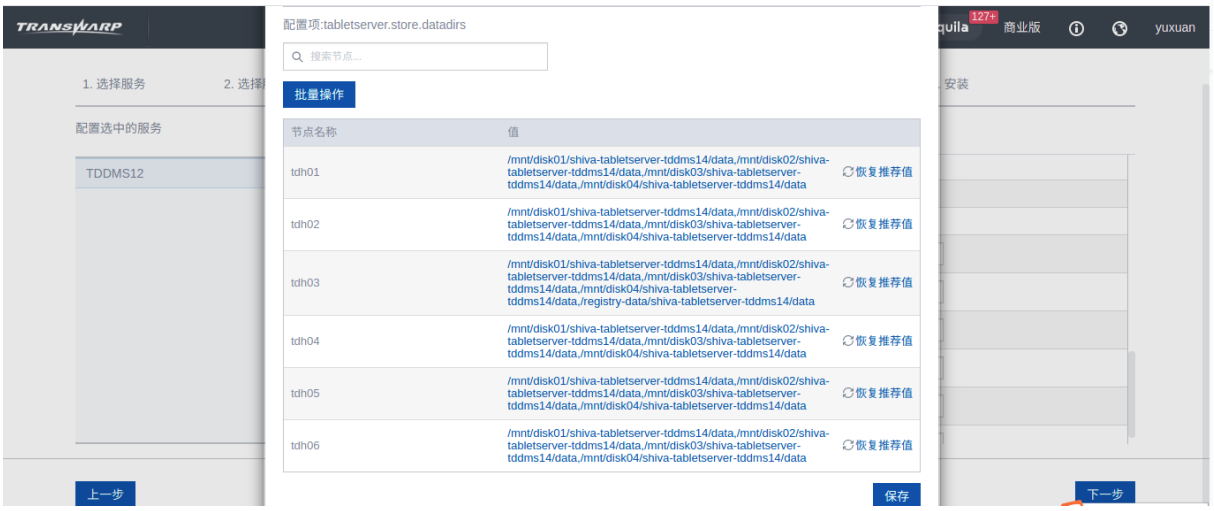
安装时要求至少安装1个，实际生产环境推荐安装至少3个或更多。

6.1.2.2. 配置服务

以下参数请在安装时进行检查和确认：

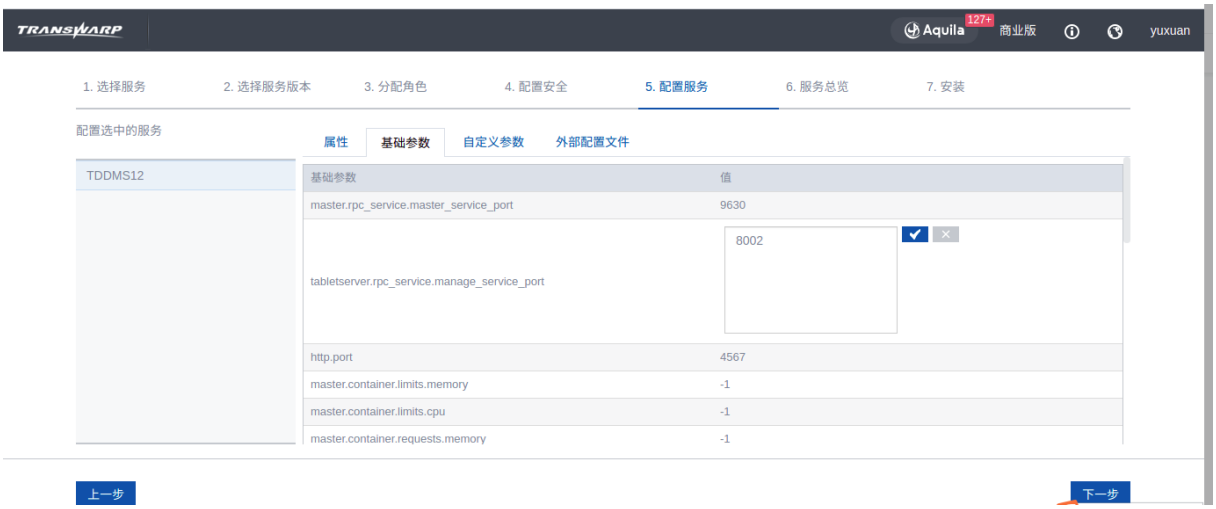
tabletserver.store.datadirs：用于存储数据。

注意：通常情况下数据目录会分配在数据盘而不是根分区。因为存在一定概率会出现数据目录错误地被挂载到根分区的情况，所以请务必确认配置的数据目录的挂载点不是根分区。



tabletserver.rpcservice.manageserviceport：用于和其他节点通信。安装后会占用从配置值开始的连续4个端口。例如安装时配置为8004，则实际会使用8004-8007这4个连续端口。

注意：端口在安装后不能变更。请在安装时做好规划，避免出现端口冲突。



6.1.3. Webserver安装

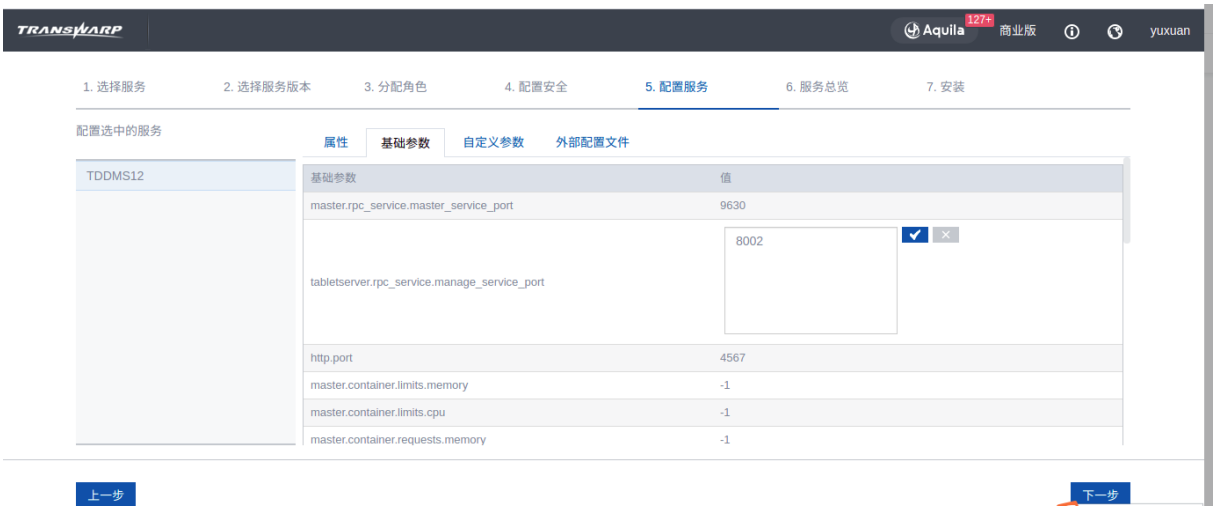
Webserver负责提供Web界面和Restful接口。

6.1.3.1. 角色分配

安装时要求至少安装1个，实际生产环境推荐安装1个。

6.1.3.2. 配置服务

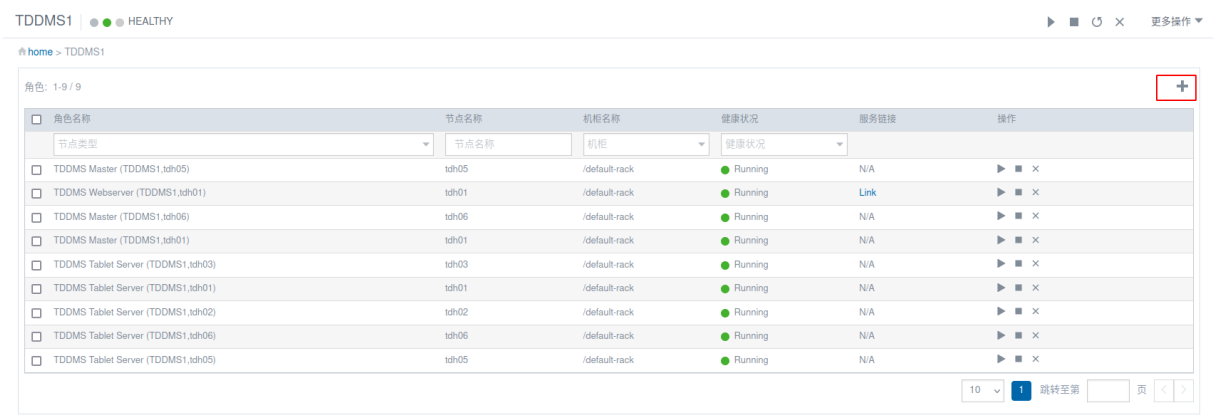
http.port：用于提供Web界面和Restful接口。安装后会占用从配置值的1个端口。例如安装时配置为4567，则实际会使用4567端口。务必在安装时确认端口没有冲突。

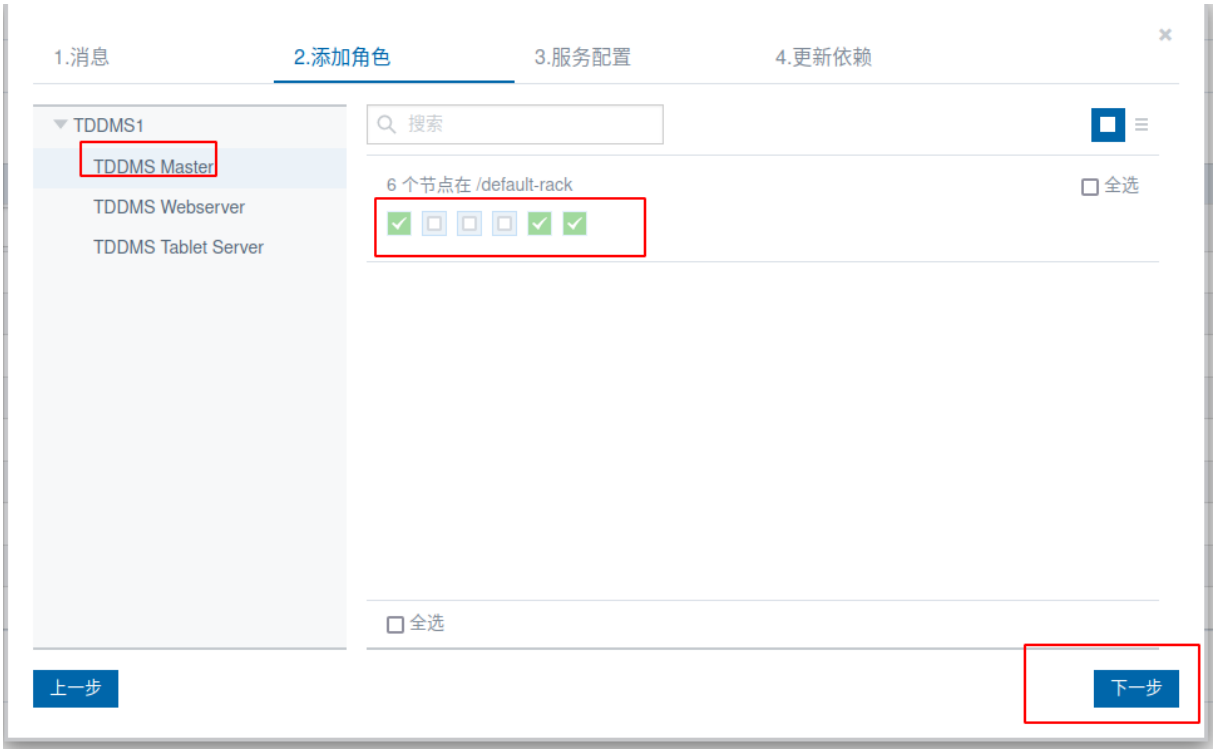


6.2. 扩容

6.2.1. Master节点扩容

6.2.1.1. 通过TDH添加shiva master节点





此处按需修改配置

- master.rpcservice.masterserviceport

默认值是9630，占用连续4个端口，以默认为例，会使用9630-9633。安装后不支持修改，

安装时需检查不与其他服务端口冲突

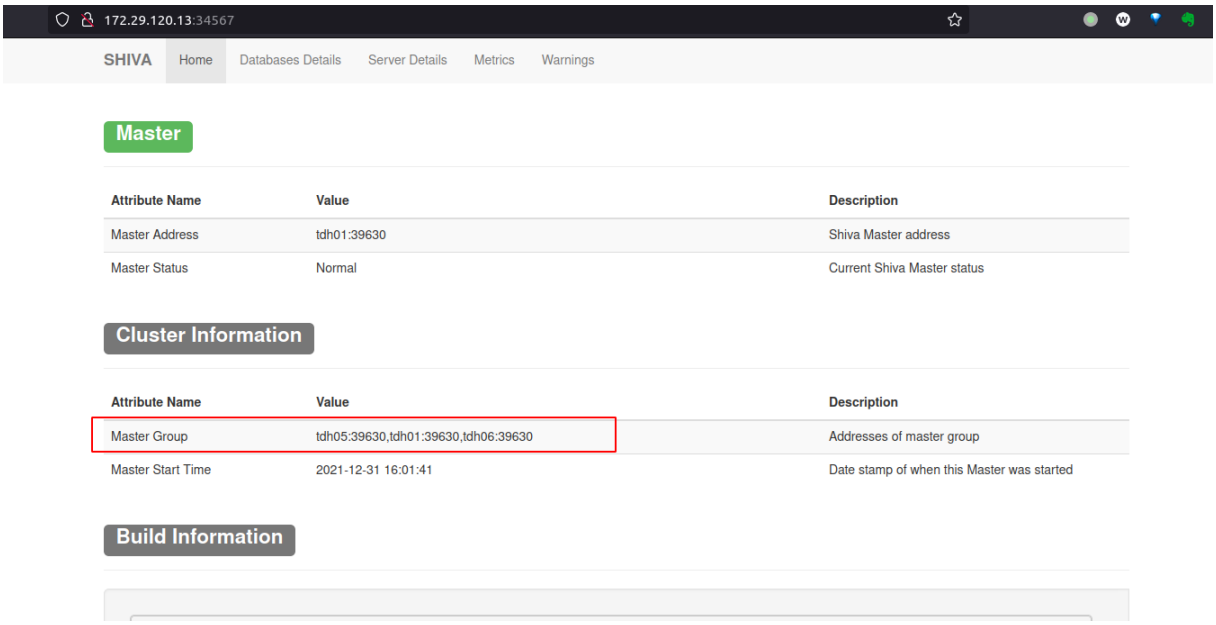
等待shiva master节点安装完成

6.2.1.2. 将新添加的master节点加入到master group

请参考**Shiva Tool使用说明**中，关于add master member的介绍

6.2.1.3. 确认扩容是否成功

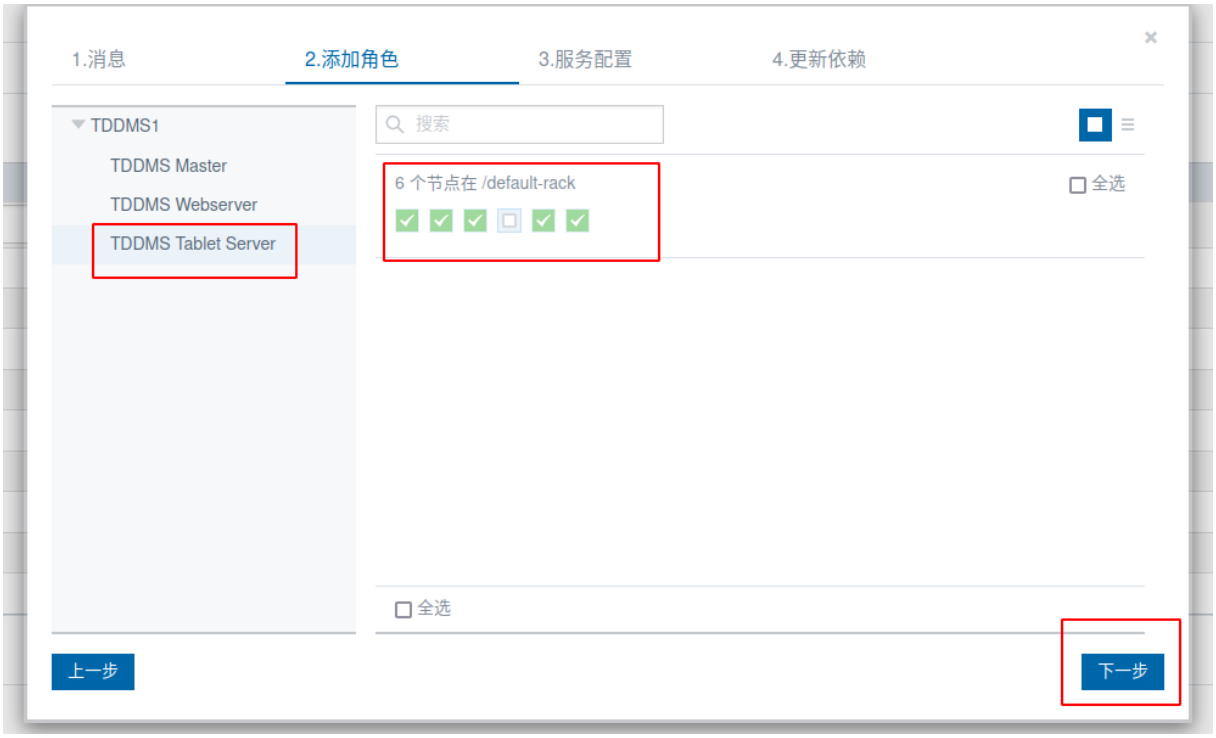
(1)打开webui页面，检查Master Group是否有新添加的master节点



(2) 查看warning页面，确认没有master warning

6.2.2. Tabletserver节点扩容

6.2.2.1. 通过TDH完成



按需修改配置

- tabletserver.store.datadirs

tserver数据目录，确保没有历史数据

注意：通常情况下数据目录会分配在数据盘而不是根分区。因为存在一定概率会出现数据目录错误地被挂载到根分区的情况，所以请务必确认配置的数据目录的挂载点不是根分区。

- topology.topology.tag

tserver一旦创建，不支持修改

- tabletserver.store.qualitytypes

配置项的数目需要与datadirs中磁盘数目保持一致。用逗号分割。可以配置的种类包括：kBronze,kSilver,kGold

支持更改配置，需要重启tserver

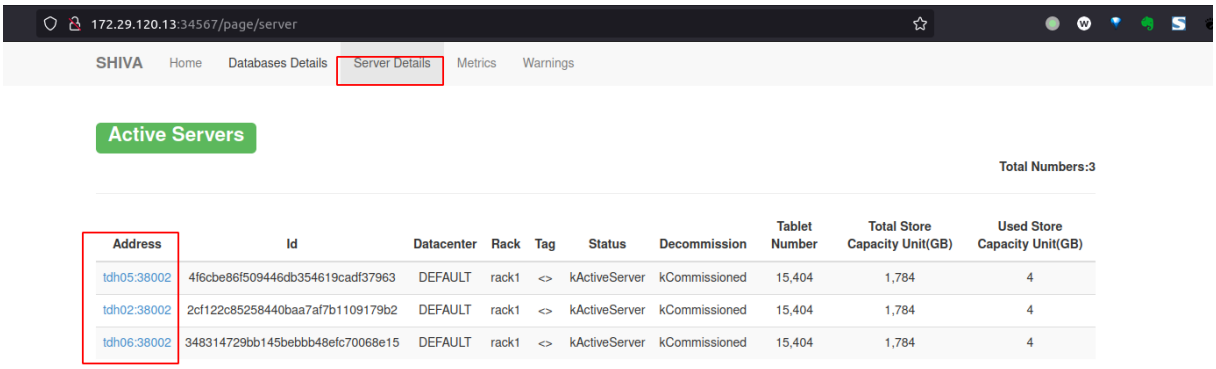
- tabletserver.rpcservice.manageserviceport

默认值是8002，会占用连续4个，以默认为例，会使用8002-8005

注意：端口在安装后不能变更。请在安装时做好规划，避免出现端口冲突。

6.2.2.2. 检查节点是否添加成功

打开webui页面，检查是否有新添加的tserver节点

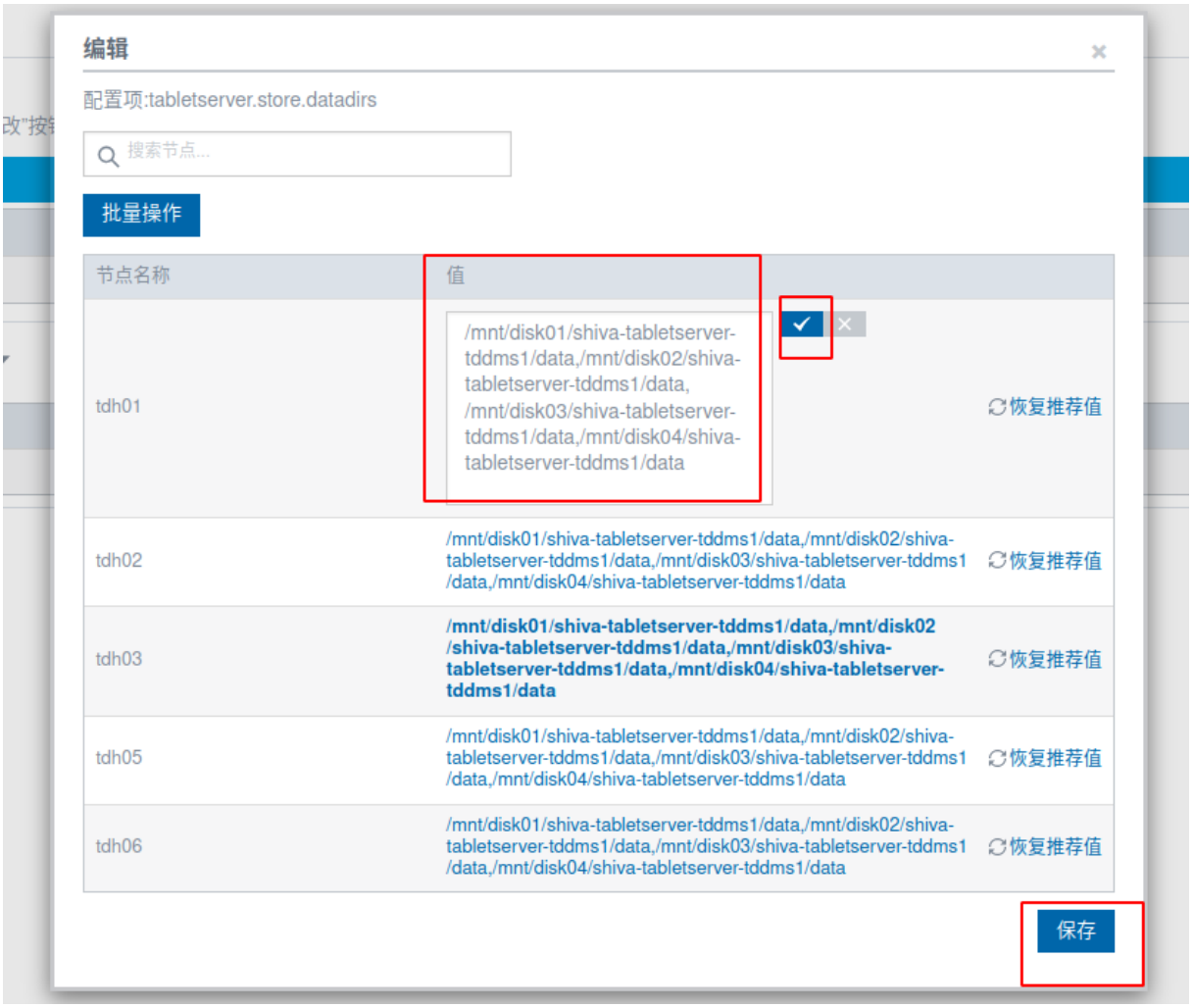


6.2.3. Tabletserver磁盘扩容

6.2.3.1. 通过TDH添加磁盘

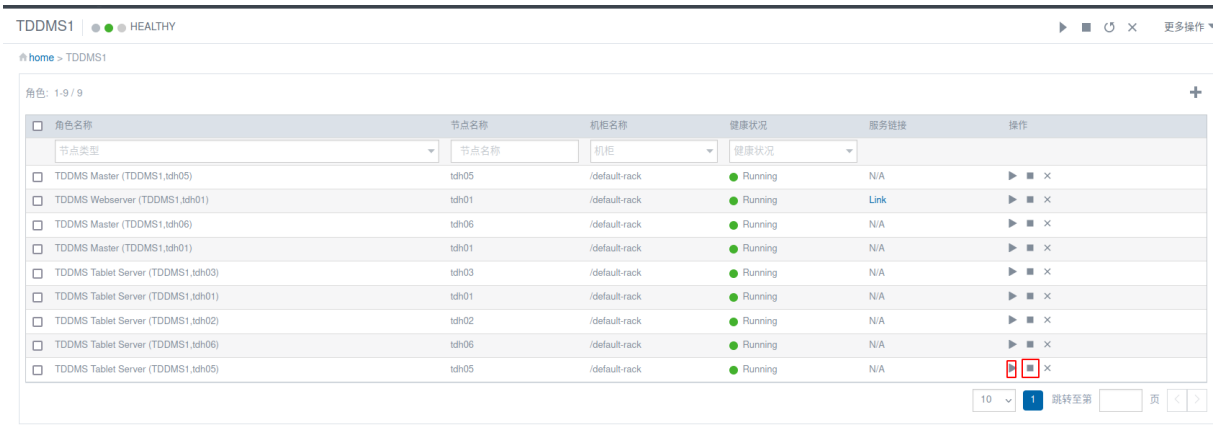
找到对应节点，加入新盘的路经，点击  ，最后进行保存

注意：通常情况下数据目录会分配在数据盘而不是根分区。因为存在一定概率会出现数据目录错误地被挂载到根分区的情况，所以请务必确认配置的数据目录的挂载点不是根分区。



6.2.3.2. 点击配置服务

6.2.3.3. 通过TDH重启tserver

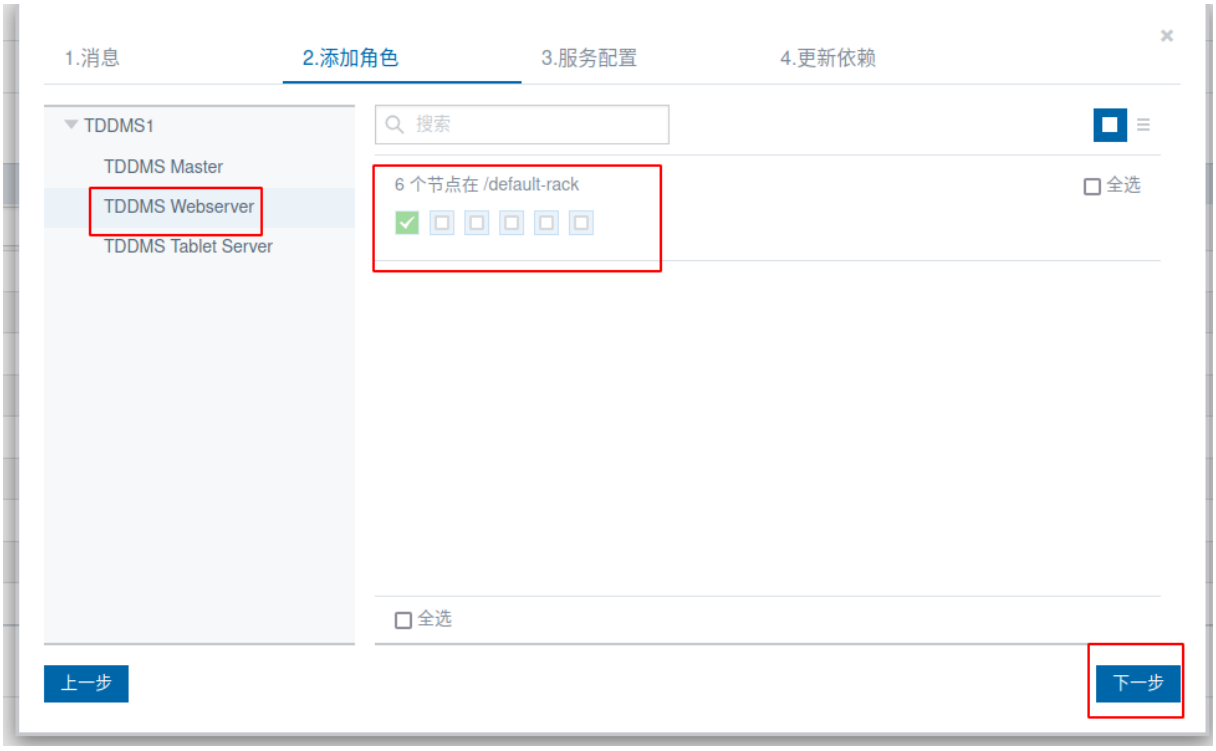


6.2.3.4. 确认磁盘是否添加成功

通过物理节点，查看添加磁盘目录是否有新产生的数据

6.2.4. Webserver节点扩容

6.2.4.1. 通过TDH添加webserver节点



按需修改配置

- http.port

默认值4567，安装时需检查不与其他服务端口冲突

6.2.4.2. 验证是否扩容成功

点击新增webserver节点的webui地址，查看是否成功

6.3. 缩容

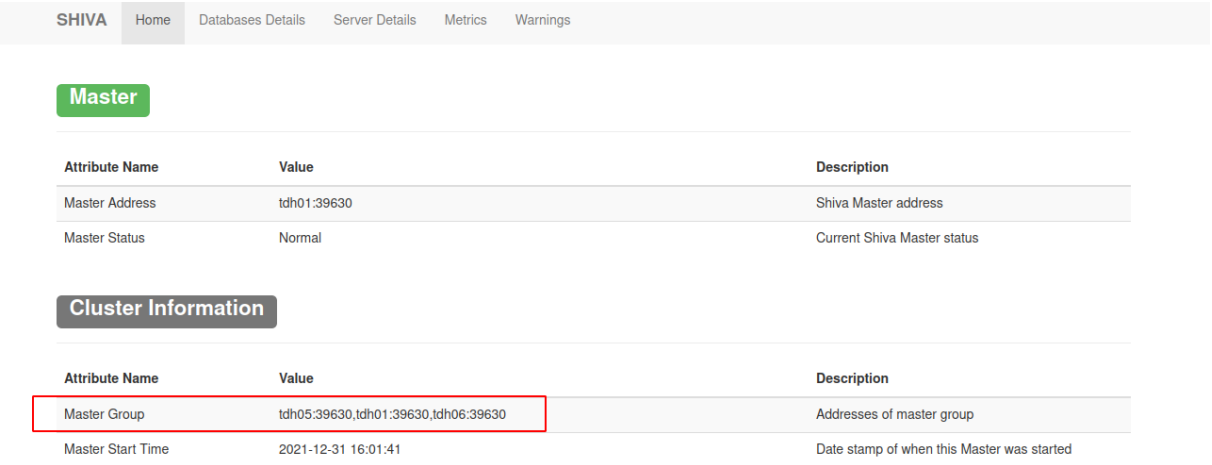
6.3.1. Master缩容

6.3.1.1. 将节点从master group中移除

具体操作说明请参考**Shiva Tool使用说明**，remove master member相关操作

6.3.1.2. 检查是否移除完成

通过webui界面，等待master节点移除成功



6.3.1.3. 通过TDH删除master节点

6.3.1.4. 清理移除节点的shiva数据

通过tdh查找master的数据目录

清空数据目录内容，例如清空/mnt/disk1/shiva-master-argodbstorage1/data目录下所有文件和隐藏文件。

6.3.2. Tabletserver节点缩容

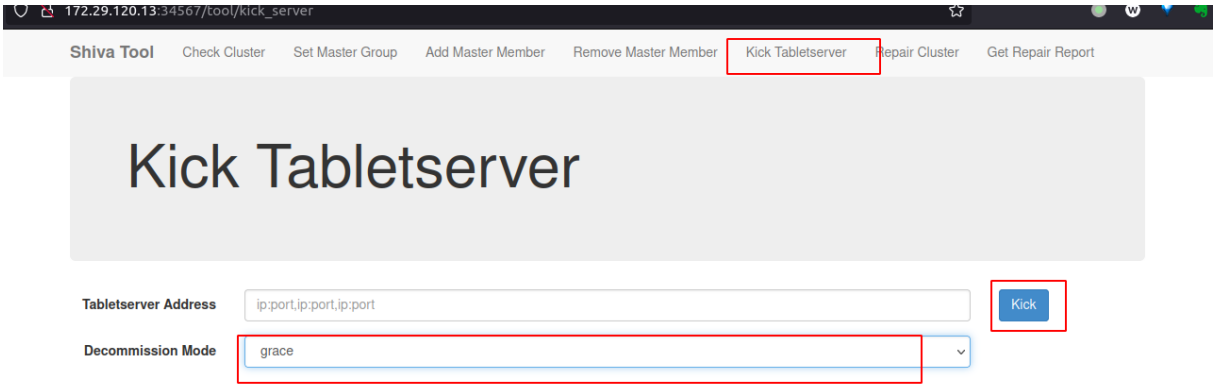
6.3.2.1. 通过webui kick tabletserver

kick tabletserver主要支持grace，force和rollback

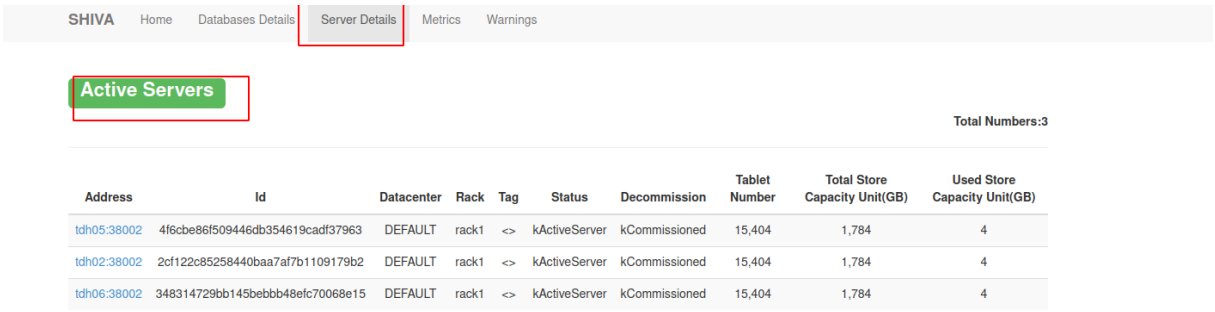
（1）grace模式：通过副本迁移的完成，完成节点退役，需要data balance开启。开启的命令：curl -X PUT "<host>:<port>/config?config=DATABALANCEENABLED,true"

（2）rollback模式：针对grace模式，可以中止掉grace模式触发的退役。

（3）force模式：在满足副本数超过半数的情况下，通过丢弃副本的方式完成退役。



6.3.2.2. 等待webui server details页面中，对应tabletserver从active状态中移除



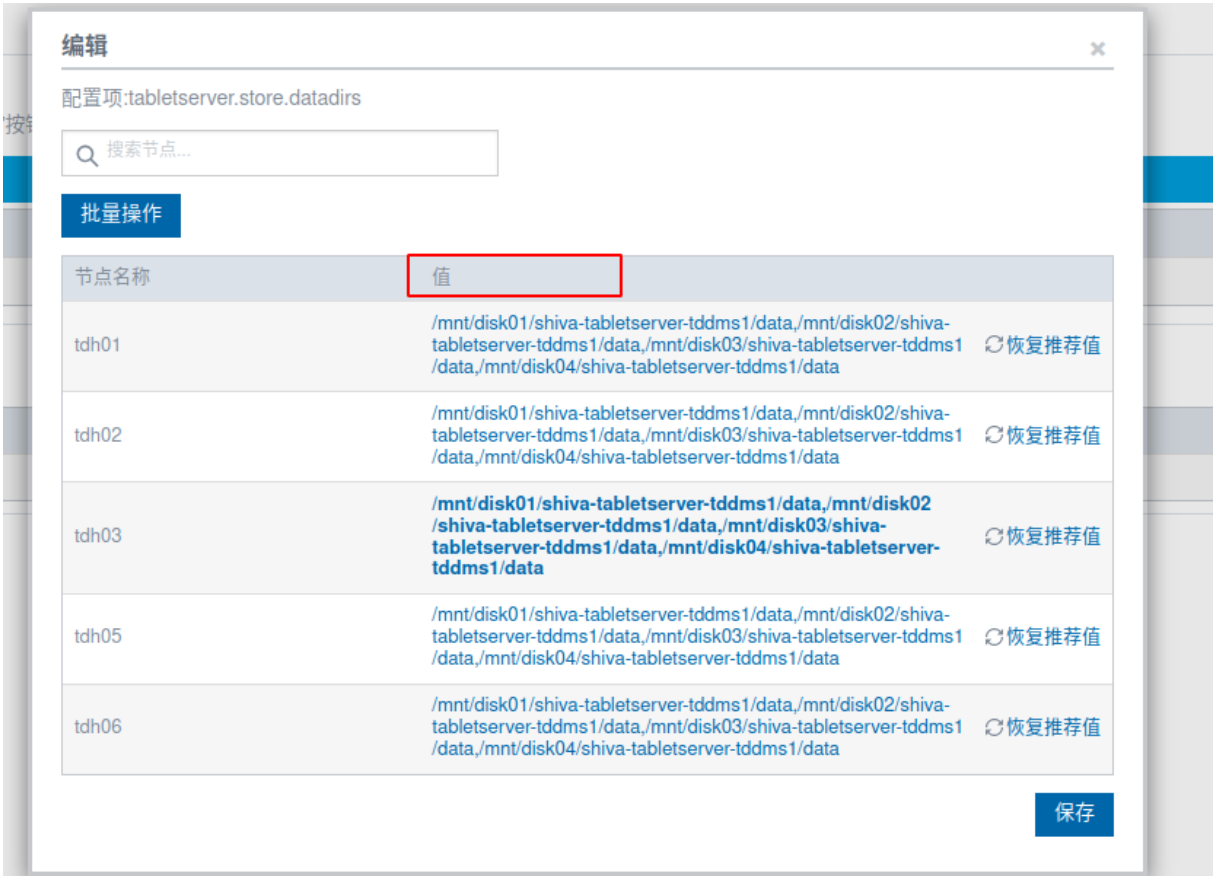
Address	Id	Datacenter	Rack	Tag	Status	Decommission	Tablet Number	Total Store Capacity Unit(GB)	Used Store Capacity Unit(GB)
tdh05:38002	4f6cbe86f509446db354619cadf37963	DEFAULT	rack1	<>	kActiveServer	kCommissioned	15,404	1,784	4
tdh02:38002	2cf122c85258440baa7af7b1109179b2	DEFAULT	rack1	<>	kActiveServer	kCommissioned	15,404	1,784	4
tdh06:38002	348314729bb145bebbb48efc70068e15	DEFAULT	rack1	<>	kActiveServer	kCommissioned	15,404	1,784	4

6.3.2.3. 如果选择了 force 模式，则必须等待Warning页面中，under replica归零

6.3.2.4. 在TDH对应的服务中删除tabletserver角色

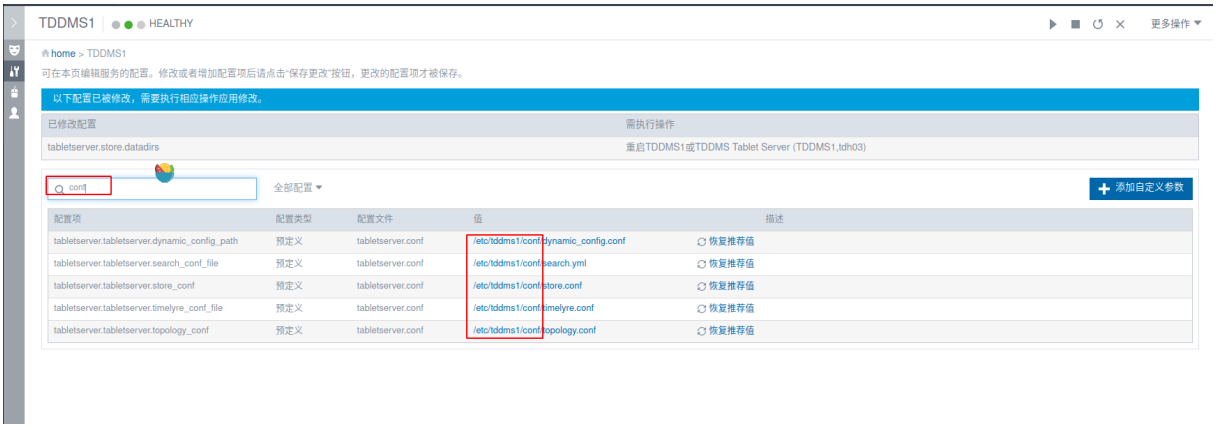
6.3.2.5. 清理tabletserver的数据目录

通过tdh查找shiva.tabletserver.store.datadirs配置项确定tablet数据目录，清空数据目录内容，可能有多个目录，每个目录都需要操作，例如清除/mnt/disk1/shiva-master-argodbstorage1/data目录下的文件和隐藏文件



6.3.2.6. 清除token文件

清除tabletserver的token，通过TDH查找配置服务配置目录



本例子中是 /etc/tddms1/conf，检查 /etc/tddms1/conf和 /etc/tddms1/conf/shiva目录中，是否存在.token文件，存在则需要删除

6.3.3. Tabletserver磁盘缩容

6.3.3.1. 缩容注意事项

1. 检查集群中是否存在单副本的表，如果存在，需要检查该表的副本是否存在于缩容的磁盘上
2. 一次只能处理一个磁盘，等待完成后在处理第二个磁盘

主要分为两种方式：

6.3.3.2. 在不重启Tabletserver时进行磁盘的动态缩容

1. 禁用磁盘

   curl -X DELETE "http://$ip:$port/server/$server/store/$storeid?pretty"

   $ip和$port是WebUI的地址和端口号

   $storeid是磁盘的标识，**必须通过webui查看。具体方法是：进入webui，选择Server details页面，选择对应tablet server，点击进入详细页面。**

   $server是tabletserver的标识

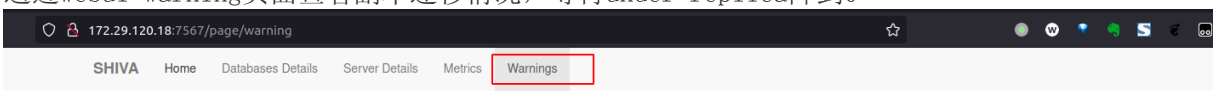
   实例：curl -X DELETE "http://192.168.1.161:4567/server/192.168.1.161:8002/store/2?pretty [http://192.168.1.161:4567/server/192.168.1.161:8002/store/2?pretty]"

2. 触发副本恢复

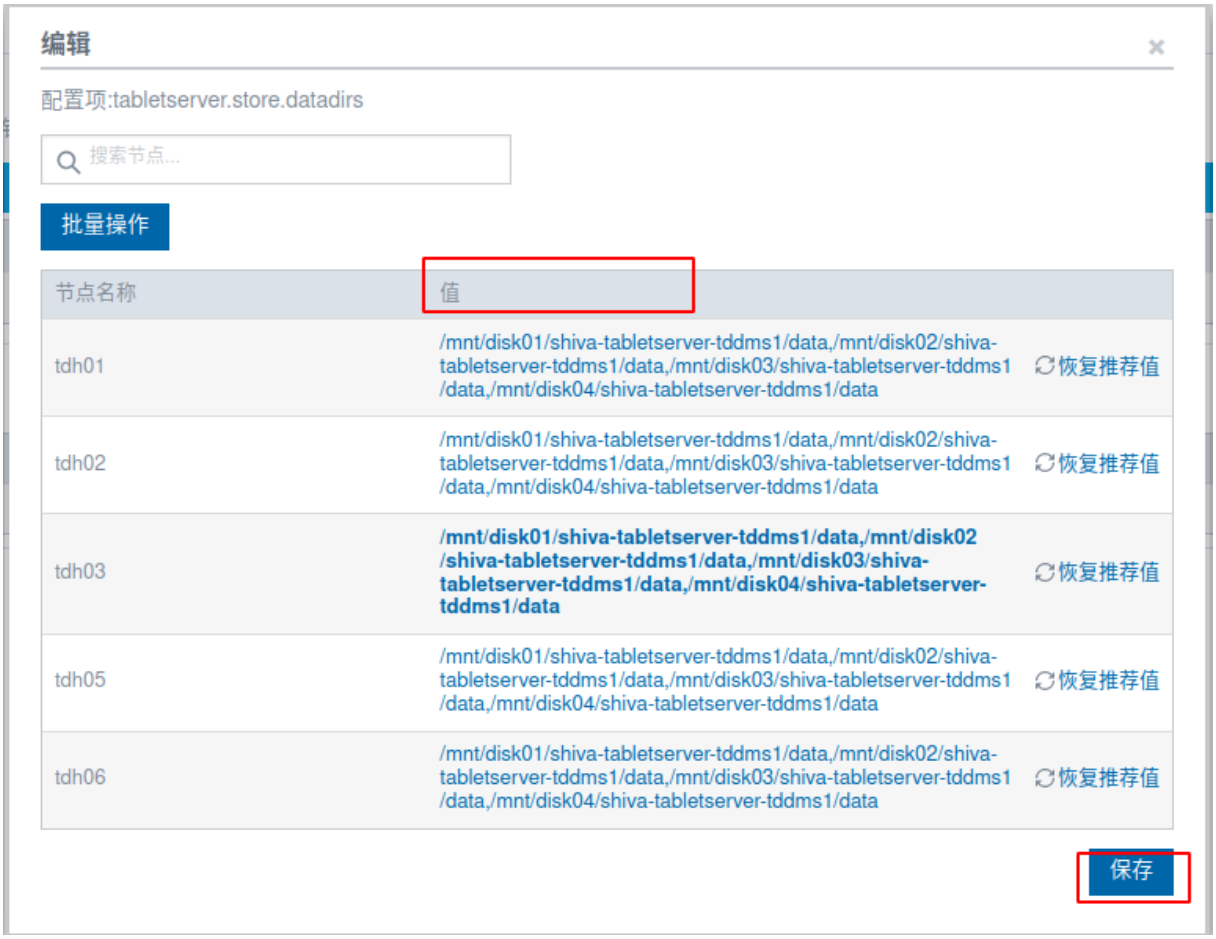
   curl -XPOST "$ip:$port/cluster/gc"

   $ip和$port是WebUI的地址和端口号

3. 通过webui warning页面查看副本迁移情况，等待under replica降到0

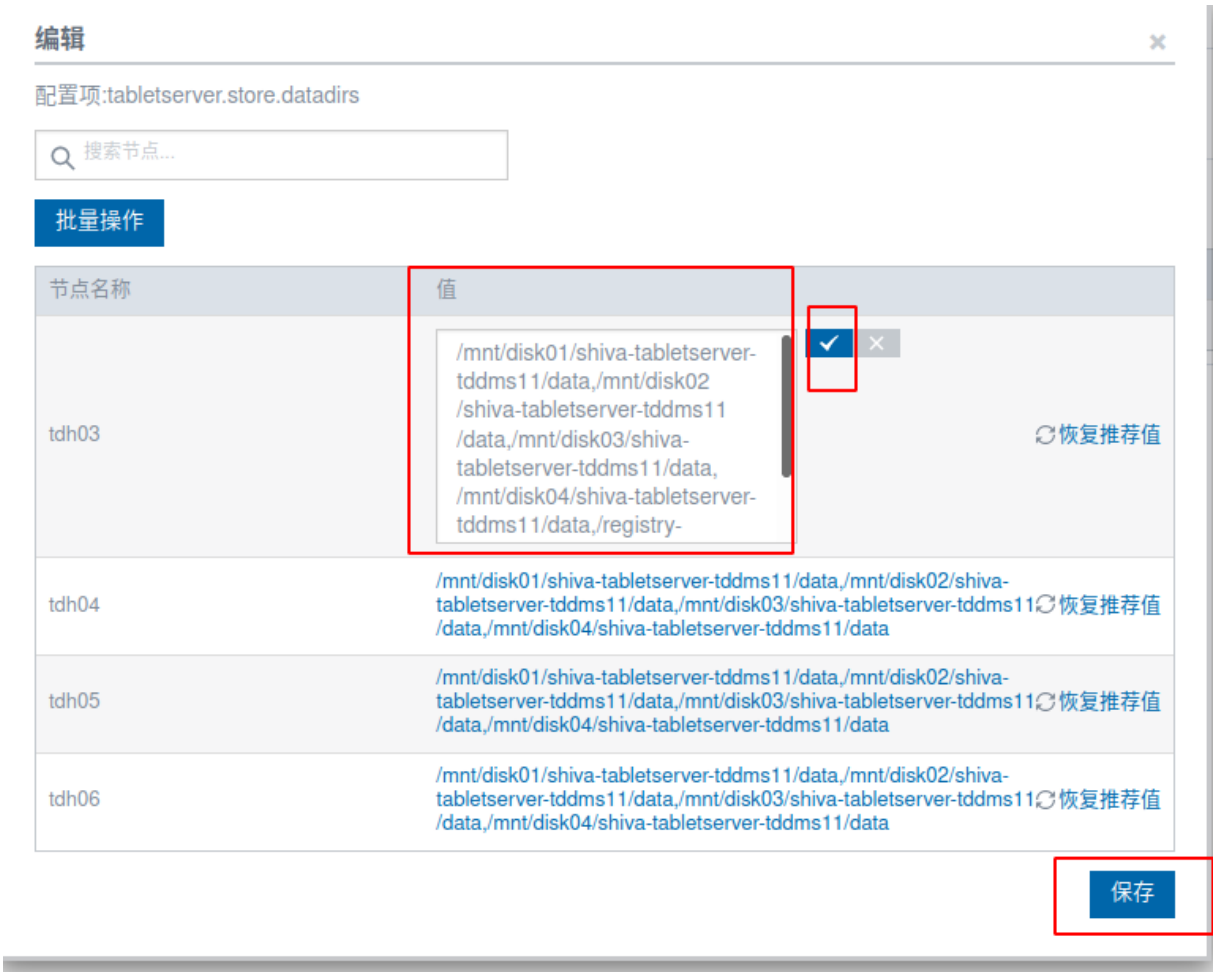


4. 将坏盘从manager的配置路径中删除，配置服务



6.3.3.3. 通过重启Tabletserver完成磁盘的缩容

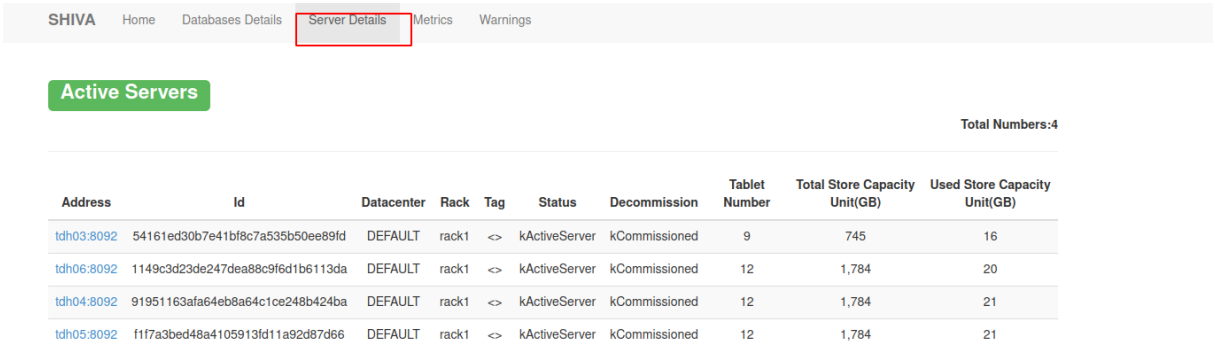
将需要剔除的磁盘从manager的配置路径中删除



点击 "配置服务"

重启tablet server以重新加载配置

通过webui等待tabletserver处于active状态



Address	Id	Datacenter	Rack	Tag	Status	Decommission	Tablet Number	Total Store Capacity Unit(GB)	Used Store Capacity Unit(GB)
tdh03:8092	54161ed30b7e41bf8c7a535b50ee89fd	DEFAULT	rack1	<>	kActiveServer	kCommissioned	9	745	16
tdh06:8092	1149c3d23de247dea88c9f6d1b6113da	DEFAULT	rack1	<>	kActiveServer	kCommissioned	12	1,784	20
tdh04:8092	91951163afa64eb8a64c1ce248b424ba	DEFAULT	rack1	<>	kActiveServer	kCommissioned	12	1,784	21
tdh05:8092	f1f7a3bed48a4105913fd11a92d87d66	DEFAULT	rack1	<>	kActiveServer	kCommissioned	12	1,784	21

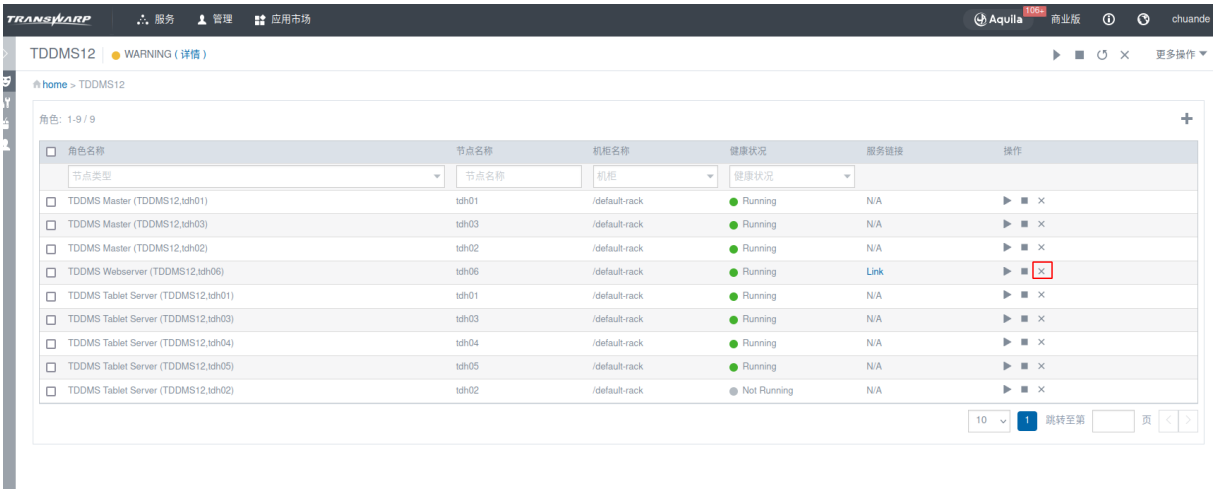
触发副本恢复

curl -XPOST "$ip:$port/cluster/gc"$ip和$port是WebUI的地址和端口号

查看Webui副本迁移情况，等待under replica降到0。

6.3.4. Webserver节点缩容

通过TDH直接删除webserver节点即可



6.4. 维修

6.4.1. 集群级维修

当集群发生异常且无法自动修复时，使用集群级修复使集群能正常工作。

注意：请确保维修前，尽可能启动所有Master和Tabletserver角色

具体操作请参考ShivaTool使用说明。

6.4.1.1. 获取集群修复报告

触发的集群维修完成后，可以获取维修过程中reset或recreate的表信息。

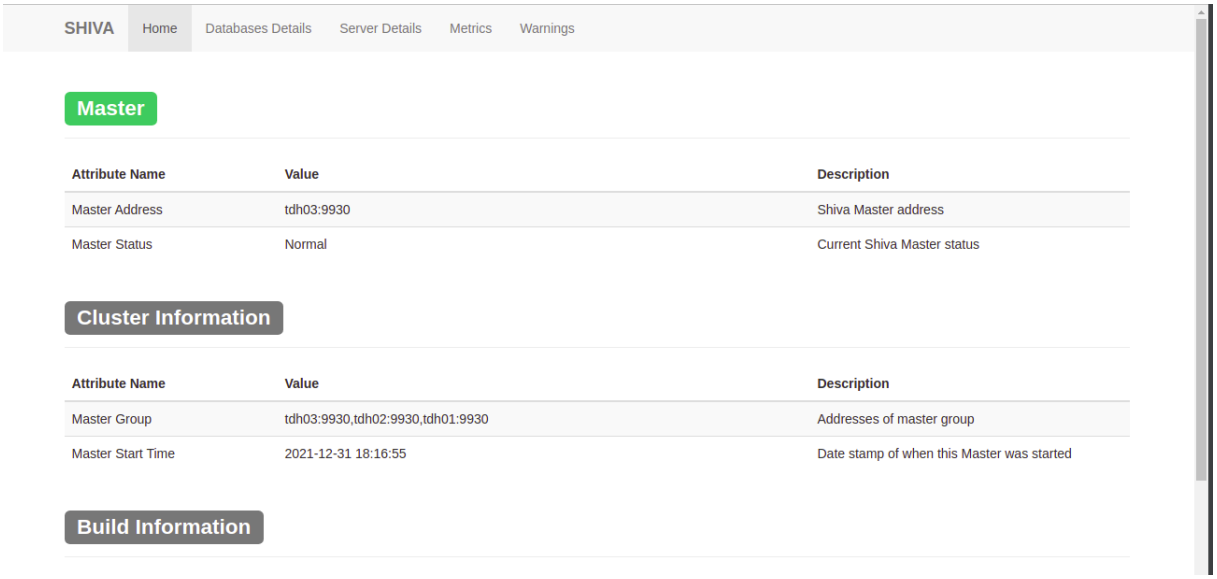
具体操作请参考ShivaTool使用说明。

6.4.1.2. 查看集群状态

1. 介绍：查看集群的状态（master是否inited，是否working）
2. 工具： webui ， shivatool,
3. shivatool用法：

```
shivatool --mastergroup=$mastergroup
```

4. webui查看集群状态



6.4.1.3. 表级别维修

注意：此项操作可能会造成数据丢失，请在操作前与研发进行确认

支持版本：shiva 2.0及以上

在少数表出现问题的时候，可以使用repair table 功能。避免repair cluster带来的大范围服务停止。

具体操作请参考ShivaTool使用说明。

6.5. Shiva Tool使用说明

6.5.1. 基本说明

docker run --network=host $image $shivatool $command

参数说明：

$image和$shivatool：

a. 对于shiva1相关版本，须使用对应版本的shiva-master镜像，Shiva tool位于/usr/shiva-master/bin/shivatool

b. 对于shiva2相关版本，使用对应版本的tddms或scope镜像，Shiva tool为/usr/shiva/bin/shivatool

6.5.2. 详细说明

下面介绍$command中各类操作需要的参数：

6.5.2.1. add master member

解释：对已经存在的master group，增加新的master副本

参数

--cmd=addmastermember

--mastergroup=${mastergroup}

--adddress=${newmasteraddress}

6.5.2.2. remove master member

解释：对已经存在的master group，删除已有的master副本

参数

--cmd=removemastermember

--mastergroup=${mastergroup}

--adddress=${removedmasteraddress}

6.5.2.3. set master member

解释：对已经存在的master group，强制指定副本，使其独立成为新的master group

参数

--cmd=setmastermember

--adddress=${masteraddress}

6.5.2.4. kick tabletserver

解释：触发已经存在的tablet server的退役

参数

--cmd=kicktabletserver

--mastergroup=${mastergroup}

--adddress=${masteraddress}

--mode=grace|force|rollback

6.5.2.5. repair cluster

解释：触发集群维修

参数：

--cmd=repaircluster

--mastergroup=${mastergroup}

--kickinactivetserverwhenrepair=true|false

--waitforrepaircluster=true|false

--recordoutput=${outputpath}

6.5.2.6. get repair report

解释：获取集群维修报告

参数：

--cmd=getrepairreport

--mastergroup=${mastergroup}

--recordoutput=${outputpath}

6.5.2.7. repair table

解释：表级修复

参数：

--cmd=repairtable

--mastergroup=${mastergroup}

--tableid=${repairedtableid}

--recordoutput=${outputpath}

6.6. 高危操作

以下操作可能会导致数据丢失和服务不可用,如在存在实际的运维需求，请及时与研发进行针对性讨论。

6.6.1. 重启

6.6.1.1. 停机时出现掉电

掉电时，可能会导致缓冲区的数据无法落盘。这会导致启动后出现数据丢失或损坏。

6.6.1.2. 未经确认直接重启Shiva Master或Shiva Tablet Server

因为启动时加载信息并恢复数据进度需要一个较长的时间窗口，所以可能会对业务有一定影响。

6.6.2. 角色变更

6.6.2.1. 手工对Master Group成员进行变更

正常情况下，Master group会依据Raft算法自动进行选主，不需要人工进行干预。

在多数派无法满足的情况下，需要准确找到进度最大的存活的Master副本。并且在手工变更后，需要进行的副本数量的恢复。

如操作过程有误，可能会导致Master group分裂，元数据丢失和服务不可用等问题。

6.6.2.2. Master扩容或缩容

增加Master节点

请参考《TDH环境安装文档》中Master的扩容

删除Master节点

请参考《TDH环境安装文档》中Master的缩容相关内容。

6.6.2.3. 严禁直接通过TDH Manager对Master删除

请参考《TDH环境安装文档》中Master的缩容相关内容。

6.6.2.4. Tablet Server扩容或缩容

请参考《TDH环境安装文档》中Tablet Server的扩容相关部分。

6.6.2.5. 删除Tablet Server节点

严禁直接通过TDH Manager对Tablet Server删除

请参考《TDH环境安装文档》中Tablet Server的缩容相关内容。

6.6.3. 集群维修

6.6.3.1. 集群维修前未确认Tablet Server状态

集群维修时，会自动处理失联的Tablet Server。（后续版本会提供开关，默认不开启此功能），如操作时，未等到所有Tablet Server都启动就进行维修，会导致数据丢失。

6.6.4. 变更数据盘

6.6.4.1. 变更Master的数据目录

目前不支持直接操作Master数据目录，包括但不限于：移动，重命名等直接进行Master底层的数据目录，可能会导致Master副本数据丢失和服务不可用。

6.6.4.2. 变更Tablet Server的数据目录

目前不支持直接操作Tablet Server数据目录，包括但不限于：移动，重命名等，直接进行Tablet Server底层的数据目录，可能会导致Tablet副本数据丢失和服务不可用。

6.6.5. 变更IP

6.6.5.1. 变更Master的IP

当前不支持直接变更Master的IP，直接进行IP变更会导致Master副本失联，可能会元数据丢失和服务不可用的风险。

6.6.5.2. 变更Tablet Server的IP变更

当前不支持直接变更Tablet Server的IP，直接进行IP变更会导致Tablet副本失联，可能会数据丢失和服务不可用的风险。

6.7. 常见问题

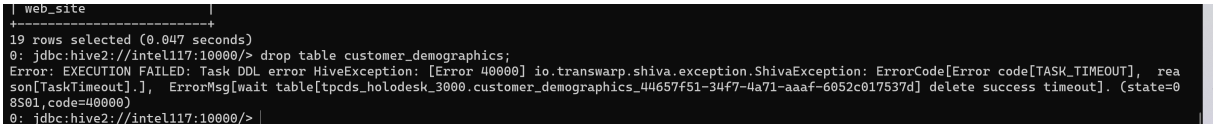
6.7.1. DDL报错

6.7.1.1. 建表失败

arm环境，华为鲲鹏，网络丢包，出现cause:CreateTablet, code:RpcTimeout, msg:Deadline Exceeded, cost:10000475us，报错原因是Tabletserver数量不符合预期，检查安装部署文档

6.7.1.2. 表变更超时

如果drop table或partition变更出现超时，如下图所示：



当出现上面的schema变更无法完成的情况时，需要确认以下几点：

- 是否重启过master

master的重启，可能会带来master group的leadership切换。目前如果leadership切换，会造成有一段时间（默认是10min）无法进行schema变更。只读事务只维护在master leader的内存中并且没有进行持久化. 切换leader后, 新的leader需要通过心跳来获取只读事务的信息

- 如果有事务对目标表有读或写的依赖，那么schema变更的进度就会被延后

curl -XGET "<host>:<port>/bulktransaction?pretty"

例如下图

上图可以看到，有一个读写事务（存在于bulktransactiondescriptions中）和一个只读事务（存在于bulkrotransactiondescriptions中）,id为218的读写事务，状态是active，并且正在对id为65的表进行bulk写入,只读事务正在读id为65的表,如果上面的两个事务，长时间没有结束。会造成id为65的表的schema变更无法结束。

6.7.2. 表写入异常

6.7.2.1. holo表bulk事务begin报错

- 现象

过多的活跃事务 too many uncommitted bulk transactions, current:5000, max:5000

- 排查命令

事务停留在active状态

curl -XGET "ip:port/bulktransaction?pretty&status=active"

事务停留在precommitting

curl -XGET"ip:port/bulktransaction?pretty&status=precommitting"

事务停留在committing状态

curl -XGET "ip:port/bulktransaction?pretty&status=committing"

6.7.2.2. holo表bulk事务precommit阶段报错

- 现象

bulk read response size over limit, max:26843545, current:26847025

rowset number over limit

- 排查命令

sql长时间无法结束，导致多版本无法合并

compact任务不及时

holo engine副本之间数据不一致

6.7.3. 表查询异常

6.7.3.1. Executor报文件不存在

Shiva使用的数据目录错误地挂载到根分区

如何判断数据盘错误地挂载到根分区上

排查和定位的方法参考此文档https://wiki.transwarp.io/pages/viewpage.action?pageId=23471581 [Manager诊断树——挂载链断裂问题排查方案]

如果tabletserver的数据盘挂载到了根分区

A. 确保tabletserver的数据盘挂载到了正确的分区。

B. 停止挂载出现问题的tabletserver角色。

C. 在TDH Manager上，对Shiva服务进行配置服务。

D. 检查/proc/mounts中对应磁盘，确保挂载到了正确的数据盘上。

E. 如果物理节点的数据盘上有数据，说明tabletserver角色是在正常安装并工作一段时间后数据盘被错误挂载到了根分区。此时需要清空或重建数据目录，清除原有的残留数据。

F. 启动出现问题的tabletserver角色。这是tabletserver会使用正确的数据盘。

G. 等待tabletserver启动完成后，shiva会自动恢复丢失的副本。

如果master的数据盘挂载到了根分区

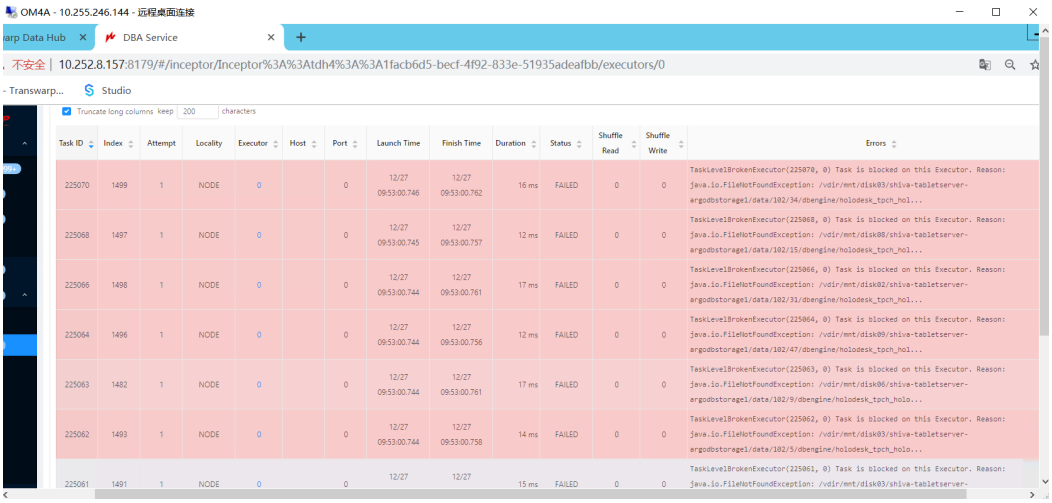
A. 将有问题的master角色从集群中剔除。

B. 确保master使用的数据目录是正确挂载的。

C. 通过新增master角色流程重新将master角色加回集群。

Executor读holo表本地文件报文件不存在—Executor未正确挂载物理机上的tserver数据目录

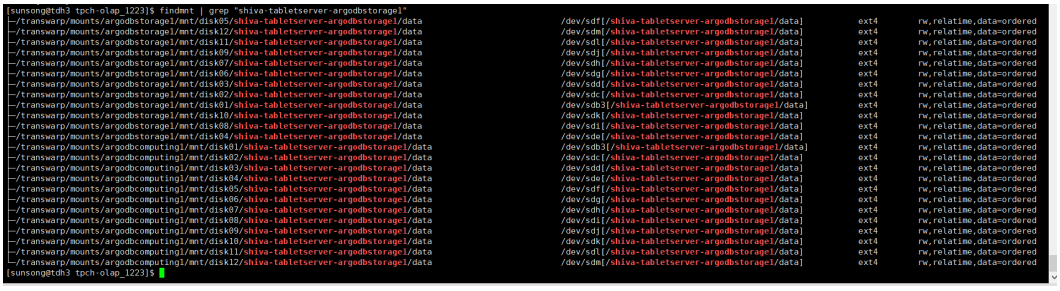
现场任务失败， 看到报错 file not found exception



1.2.2 先确认数据是否存在

去到物理机上的相应目录上查看， 如果数据不存在， 说明确实丢了。 如果数据是存在的，则说明是其它问题。此次SLA中数据在物理机上是存在的。

这个时候怀疑可能是挂载问题， 应在物理机上使用 findmnt | grep "shiva-tabletserver-argodbstorage1" 去查看一下挂载

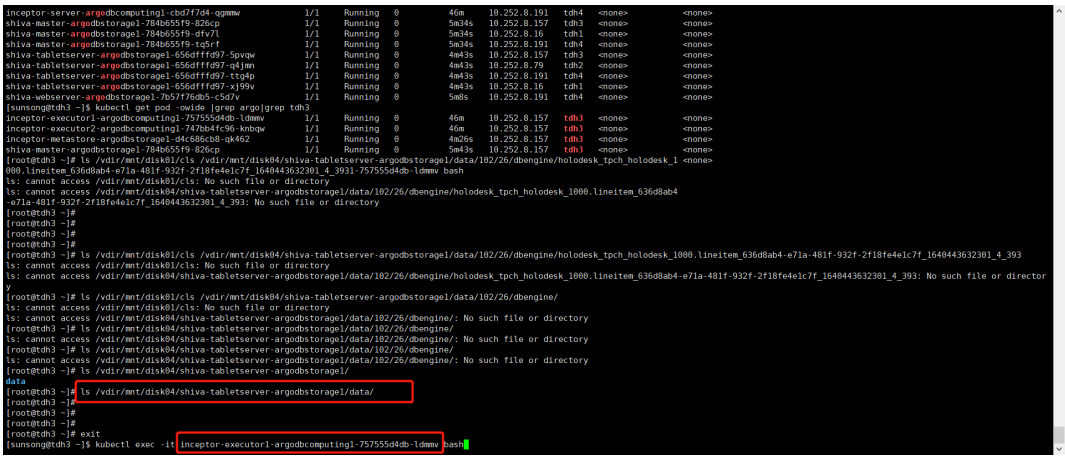


上面这张图的意思是： argodbstorage1 （即tablet server） 和 argodbcomputing1 （即executor）都正确的将目录挂载到物理机对应的路径上 ， 这是正常情况。

如果是挂载问题， 往往是argodbstorage 或者 argodbcomputing 至少有一个没挂好， 即要么没挂载， 要么没挂在到正确的路径上。

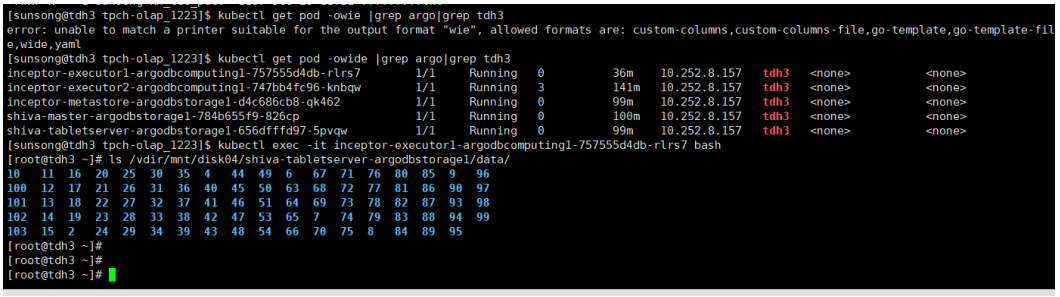
这个SLA是executor没有挂载好。

去检查executor pod 中的数据目录， 发现是空的（注意pod中的路径比物理机中的路径多一个 /vdir 即pod中路径开头为 /vdir/mnt/ ）



为解决这个挂载问题， 配置服务， 重启了该节点的executor pod（配置服务会重新触发挂载）

重启完之后再次查看executor 的数据目录， 已经正常了。



6.7.4. 集群状态异常

6.7.4.1. hiva webui报错 server warning中包含io fails

[[Tabletserver承载的tablet副本数超限导致Too many open files]]

检查tabletserver的ERROR日志

1. 磁盘空间不足：去tabletserver的ERROR中搜索 no space left
2. 磁盘损坏 ： 磁盘无法访问

6.7.4.2. 副本损坏

少数派副本 损坏： 对于3副本的表，数量为1个；对于5副本的表，数量为1个或2个

shiva 1.9 及之前，需要手动介入

curl -X DELETE "<host>:<port>/replica?namespace=<namespace>&tablename=<tablename>&tabletname=<tabletname>&server=<serveraddress>"

shiva 1.10开始， 会定时进行检测并且自动处理损坏的副本。 如需立即触发，可以使用restful触发 data balance ：

curl -XPOST "ip:port/cluster/balance?action=data?pretty"

多数派损坏 ：对于3副本的表，数量到达2个或以上；对于5副本的表，数量到达3个或以上

使用repair cluster进行处理(须在专业人员指导下操作)

手工修改 tablet group(须在专业人员指导下操作)

6.7.4.3. Webserver异常

/usr/lib/shiva-webserver/plugin/install.sh报错：active tabletserver数量不足

init master group failed

6.7.4.4. Master异常

6.7.4.5. Tabletserver异常

- 端口冲突
- 重装的场景下，本地存在没有清理的陈旧数据
- 磁盘损坏导致无法启动：查看对应tserver磁盘情况
- 启动过程中 master无法接受tabletserver心跳：cmd:GetAllTablets报错ExceedLimit。
- tabletserver启动慢，导致tserver不服务/处于失联状态
  1. 查看tabletserver日志 grep -c “open tablet success” tabletservermain.INFO
  2. 使用命令查看tableserver磁盘状态是否打满，判断是否为启动慢问题 iostat -x 1 100

客户服务

技术支持

感谢您使用星环信息科技（上海）股份有限公司的产品和服务。如您在产品使用或服务中有任何技术问题，可以通过以下途径找到我们的技术人员给予解答。

email： support@transwarp.io

技术支持热线电话：4007-676-098

官方网址： http://www.transwarp.cn/

论坛支持： http://support.transwarp.cn/

意见反馈

如果您在系统安装，配置和使用中发现任何产品问题，可以通过以下方式反馈：

email： support@transwarp.io

感谢您的支持和反馈，我们一直在努力！