RatuDB

write by: lei.fu

这是什么？

这个项目是在 Opensearch 3.0 进行的扩展。说白了，还是个 Opensearch，但是在 Opensearch 内部集成了Cassandra。

Cassandra作为宽表存储，Elasticsearch存储着Cassandra的索引，每次查询的时候，都会到 Opensearch 进行索引操作，优化提升Cassandra的查询性能。

查看 测试文档 有关运行RatuDB测试的更多内容.

Getting Started

Installation

- [下载] 并且 解压 RatuDB 安装包.
- 运行 `bin/opensearch` 在 Linux 或者 macOS. 运行 `bin\opensearch.bat` 在 Windows.
- 运行 `curl -X GET http://localhost:9200/` 验证 Elasticsearch 是否运行.
- 目前项目没有集成 `cql` 客户端, 可以在随机找一个, 进入目录，运行 `./bin/cqlsh`，cql默认不填写`ip`地址的情况下，就是连接`127.0.0.1`，可以验证Cassandra是否运行。
- 也可以使用`./bin/nodetool` 客户端工具，查看工具变化，RatuDB默认开放 9042端口，用于查看集群状态。目前安装也未集成客户端工具。

1. 创建表

首先，两个服务启动成功后，登录**Cassandra**：

```
$ ./bin/cqlsh
Connected to Test Cluster at 127.0.0.1:9042
[cqlsh 6.1.0 | Cassandra 4.1.3-SNAPSHOT | CQL spec 3.4.6 | Native protocol v5]
Use HELP for help.
cqlsh>
```

然后可以创建一个**keyspaces:**

```
CREATE KEYSPACE lei
WITH REPLICATION = {'class': 'SimpleStrategy', 'replication_factor': 1};
```

`replication_factor` 这个参数建议安装集群数量设置，本身**Cassandra**对于内存的使用有自动扩容的机制，不必考虑内存占用

然后再创建一张表：

```
CREATE TABLE lei.tweets (
   id INT PRIMARY KEY,
   user TEXT,
   body TEXT,
   time TIMESTAMP,
   latitude FLOAT,
   longitude FLOAT
);
```

2. 创建索引

从目前看，跟普通的**Cassandra**操作是没有什么区别的，接下来的创建的索引就不一样了，举例如下：

```
CREATE CUSTOM INDEX tweets_index ON lei.tweets ()
USING 'org.apache.ratu.second.ElasticSecondaryIndex'
WITH OPTIONS = {
   'refresh_seconds': '1',
   'async_write': 'true',
   'schema': '{
      fields: {
         id: {type: "integer"},
         user: {type: "text"},
         body: {type: "text", analyzer: "english"},
         time: {type: "date", pattern: "yyyy-MM-dd"},
         latitude: {type: "float"},
         longitude: {type:"float"}
      }
   }'
};
```

这是一个全字段索引的例子，当然也可以根据自己的需要进行索引配置。

接下来拆解这些内容：

`CREATE CUSTOM INDEX tweets_index ON lei.tweets ()` 这句的意思就是创建一个自定义索引，名字叫 `tweets_index` 在 `lei.tweets` 这张表上。

`USING 'org.apache.ratu.second.ElasticSecondaryIndex'` 这句就是引用了二级索引的操作类。

`WITH OPTIONS` 就是二级索引的参数。参数里 `refresh_seconds` 就是 Opensearch 的索引刷新频率，如果数据写入频繁，这个值不建议设置过大，索引数据写入ES之后，还是要刷新，才会在ES中查询到，这个刷新频率就是靠这个参数控制的。数据写入频繁，刷新过快，会导致ES内存占用过大，导致内存溢出。默认值是`-1`。如果未设置的情况，如果在Opensearch中可见，需要手动刷新，执行如下api:

```
POST 索引名/_refresh
```

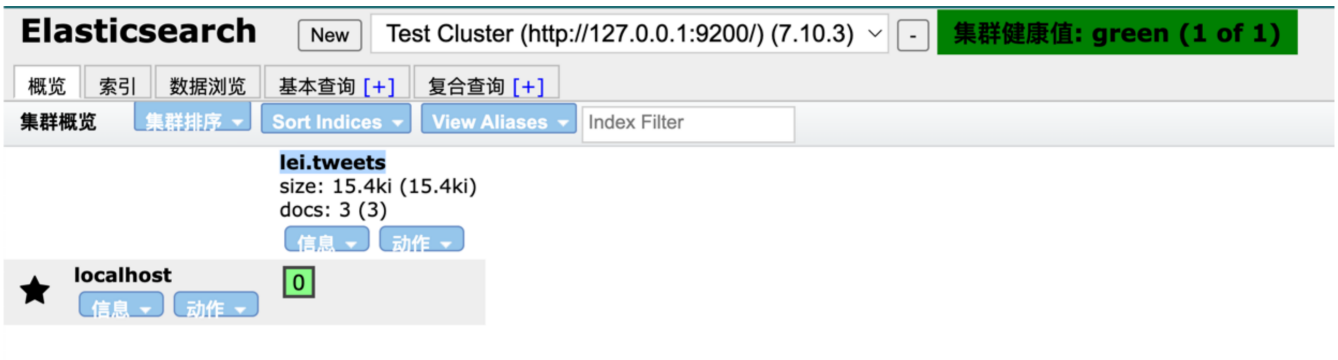
schema 里的 fields 就是配置写入 Opensearch 的字段，举个例子：

body: {type: "text", analyzer: "english"}, 这里边 type 就是类型，analyzer 就是分词器。

time: {type: "date", pattern: "yyyy-MM-dd"}, 这里边的 pattern 就是日期格式。

并且，如果设置 text 也会对应创建子字段。

以上都执行完成后，可以执行在es里看到索引已经创建了：



3. 创建索引补充内容

最近测试发现一个问题，数据量频繁的情况，为了保证性能，异步写入的吞吐量确实高，这是1万条数据，1G内存，未同步的情况的报告：

```
[OVERALL], RunTime(ms), 4167
[OVERALL], Throughput(ops/sec), 2399.808015358771
[TOTAL_GCS_G1_Young_Generation], Count, 4
[TOTAL_GC_TIME_G1_Young_Generation], Time(ms), 16
[TOTAL_GC_TIME_%_G1_Young_Generation], Time(%), 0.3839692824574034
[TOTAL_GCS_G1_Old_Generation], Count, 0
[TOTAL_GC_TIME_G1_Old_Generation], Time(ms), 0
[TOTAL_GC_TIME_%_G1_Old_Generation], Time(%), 0.0
[TOTAL_GCs], Count, 4
[TOTAL_GC_TIME], Time(ms), 16
[TOTAL_GC_TIME_%], Time(%), 0.3839692824574034
[CLEANUP], Operations, 10
[CLEANUP], AverageLatency(us), 222516.6
[CLEANUP], MinLatency(us), 0
[CLEANUP], MaxLatency(us), 2226175
[CLEANUP], 95thPercentileLatency(us), 2226175
[CLEANUP], 99thPercentileLatency(us), 2226175
[INSERT], Operations, 10000
[INSERT], AverageLatency(us), 1289.455
[INSERT], MinLatency(us), 401
[INSERT], MaxLatency(us), 47551
[INSERT], 95thPercentileLatency(us), 3957
[INSERT], 99thPercentileLatency(us), 5307
[INSERT], Return=OK, 10000
```

同步的情况：

```
[OVERALL], RunTime(ms), 4375
[OVERALL], Throughput(ops/sec), 2285.714285714286
[TOTAL_GCS_G1_Young_Generation], Count, 4
[TOTAL_GC_TIME_G1_Young_Generation], Time(ms), 15
[TOTAL_GC_TIME_%_G1_Young_Generation], Time(%), 0.34285714285714286
[TOTAL_GCS_G1_Old_Generation], Count, 0
[TOTAL_GC_TIME_G1_Old_Generation], Time(ms), 0
[TOTAL_GC_TIME_%_G1_Old_Generation], Time(%), 0.0
[TOTAL_GCs], Count, 4
[TOTAL_GC_TIME], Time(ms), 15
[TOTAL_GC_TIME_%], Time(%), 0.34285714285714286
[CLEANUP], Operations, 10
[CLEANUP], AverageLatency(us), 223336.7
[CLEANUP], MinLatency(us), 1
[CLEANUP], MaxLatency(us), 2234367
[CLEANUP], 95thPercentileLatency(us), 2234367
[CLEANUP], 99thPercentileLatency(us), 2234367
[INSERT], Operations, 10000
[INSERT], AverageLatency(us), 1575.29
[INSERT], MinLatency(us), 541
[INSERT], MaxLatency(us), 56607
[INSERT], 95thPercentileLatency(us), 3393
[INSERT], 99thPercentileLatency(us), 7135
[INSERT], Return=OK, 10000
```

基本上是差不多的性能了。

- 但是高频写入大数据量的情况，会触发ES的断路器异常，主要原因是，异步写入，任务是写入到一个 Queue，但是 Queue 的长度如果超限，将会导致`[parent] Data too large`的异常，这就是触发了父级的断路器，ES为了防止内存溢出，专门的设置。
- 所以我在创建索引的时间了，增加了`async_write`
  字段配置，默认是`false`，就是同步写入，同步写入可以保证稳定性。如果量不大，需要快速写入，并且频率也不高，可以设置为`true`，异步写入。

创建索引例子：

```
CREATE CUSTOM INDEX usertable_index ON ycsb.usertable ()
USING 'org.apache.ratu.second.ElasticSecondaryIndex'
WITH OPTIONS = {
   'refresh_seconds': '120',
   'async_write': 'true',
   'schema': '{
      fields: {
         field0: {type: "text"},
         field1: {type: "text"},
         field2: {type: "text"},
         field3: {type: "text"},
         field4: {type: "text"},
         field5: {type: "text"},
         field6: {type: "text"},
```

```
            field7: {type: "text"},
            field8: {type: "text"},
            field9: {type: "text"}
        }
    }'
};
```

如果是异步写入es，其实就是后台线程拉倒了es里边，es里边用一个queue存储，一个一个的进行处理。所以异步情况，要使用如下api观察是否写入成功：

```
GET _cat/thread_pool?v
```

如果queue里边的线程没有执行完毕，就执行其他操作，有可能出现异常。所以异步情况下建议还是要观察一下，es的写入情况。

还有就是`refresh_seconds`参数，如果未设置的情况下，默认是`-1`，为了保证性能，就不刷新可见，但是同样Opensearch在内存不足的情况会出现断路器的异常`[parent] Data too large` Opensearch的内用第一是不会自动扩容，尤其是JVM 堆，一开始都是在`jvm-options`文件里设置好的，一旦不够用，为了避免服务出现异常，就会对占用内存过大的线程进行限制。所以建议横向扩展，对Opensearch分配足够多的内存。

4. 写入数据

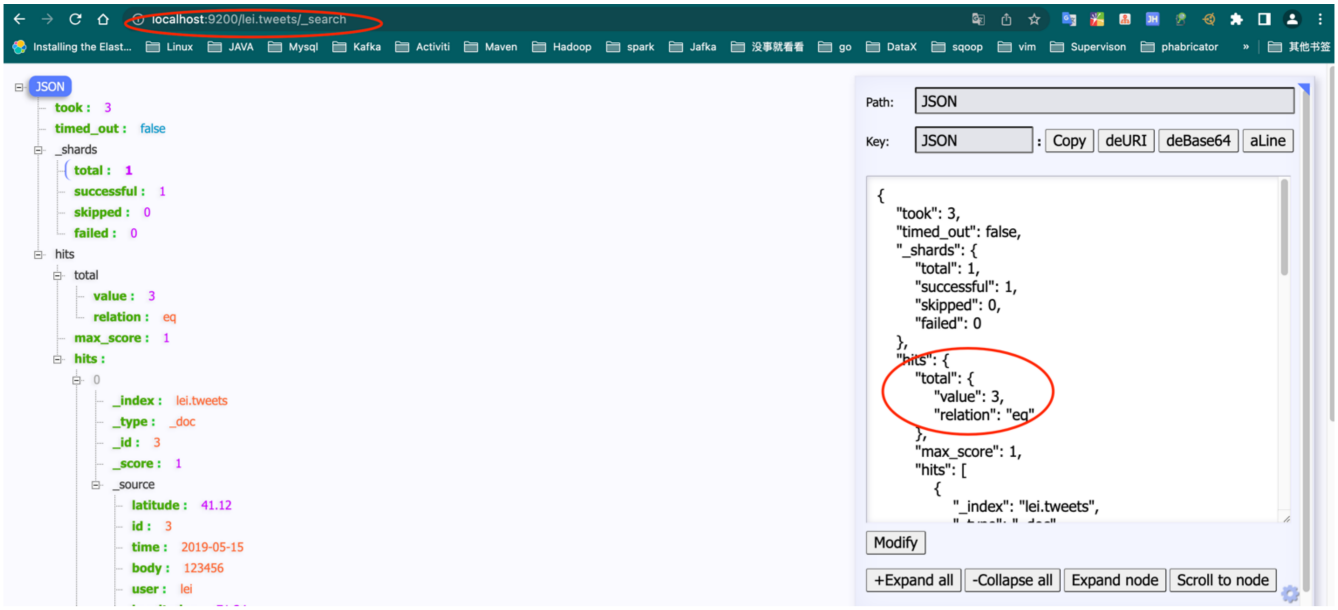
再写入几条数据试试，

```
INSERT INTO lei.tweets (id, user, body, time,latitude,longitude) VALUES (1, 'fu',
'abc', '2015-05-15',41.12,-71.34);

INSERT INTO lei.tweets (id, user, body, time,latitude,longitude) VALUES (2, 'fu',
'123456', '2019-05-15',41.12,-71.34);

INSERT INTO lei.tweets (id, user, body, time,latitude,longitude) VALUES (3, 'lei',
'123456', '2019-05-15',41.12,-71.34);
```

再看看数据：



正好三条数据。

5.查询

既然写入索引变化了，所以在Cassandra中查询数据，也需要一个新的表达式，才能进行二级索引的使用：

```
<!--range查询-->
SELECT * FROM lei.tweets WHERE expr(tweets_index, '{
   query: {type: "range", field: "time", gte: "2014-04-25", lte: "2015-05-21"}
}');
```

```
query:代表的就是普通查询
type:代表的就是DSL的查询函数
field: 代表的是要查询的字段
gte:  大于等于
lte:  小于等于
```

```
<!--match查询-->
SELECT * FROM lei.tweets WHERE expr(tweets_index, '{
   query: {type: "match", field: "user", query: "lei"}
}');
```

```
<!--match查询, value形式-->
SELECT * FROM lei.tweets WHERE expr(tweets_index, '{
   query: {type: "match", field: "user", value: "lei"}
}');
```

```
<!--match_phrase查询-->
```

```
SELECT * FROM lei.tweets WHERE expr(tweets_index, '{
   query: {type: "match_phrase", field: "user", query: "lei"}
}');
```

```
<!--match_phrase, value-->
SELECT * FROM lei.tweets WHERE expr(tweets_index, '{
   query: {type: "match_phrase", field: "user", value: "lei"}
}');
```

```
<!--term -->
SELECT * FROM lei.tweets WHERE expr(tweets_index, '{
   query: {type: "term", field: "user", value: "lei"}
}');
```

refresh: true ES ES

```
<!--range-->
SELECT * FROM lei.tweets WHERE expr(tweets_index, '{
   query: {type: "range", field: "time", gte: "2014-04-25", lte: "2015-05-21"},
   refresh: true
}') limit 100;
```

Cassandra Opensearch

CQL	**Java**	**ES**	
ascii	String	text	asii
bigint	long	long	64
blob	ByteBuffer/byte[]	text	ES
boolean	Boolean	boolean	
decimal	BigDecimal	float	
double	double	double	64
float	float	float	32
inet	String	ip	ipv4ipv6ipipv6
int	int	integer	32
text	String	text	utf-8

CQL 类型	对应**Java**类型	**ES**类型	描述
timestamp	Date	date	日期 Opensearch 支持的日期，yyyy-MM-dd 或者 yyyy-MM-ddTHH:MM:SSZ ,代码内自动转换
uuid	UUID	text	UUID类型
timeuuid	UUID	text	时间相关的UUID
varchar	string	text	text的别名
varint	BigInteger	text	高精度整型
duration	String	text	以纳秒为单位的持续时间
smallint	Integer	integer	16位浮点数
tinyint	Integer	integer	8位浮点数
list<T>	String	text	存入到ES之后是array
time	long	long	纳秒级别的时间戳，格式 hh:mm:ss 的纳秒精准度，存入ES是64位整数
set<T>	Set<T>	text	存入到ES之后是array
map<T,T>	Map<T,T>	nested	复合结构，支持子查询

构建源码

RatuDB 使用 Gradle 构建系统.

Gradle 使用 `8.0.2` 版本，不建议升级 Gradle版本。

完成的发行版将输出到 `distributions/archives` 目录.

Fork并且克隆代码

拉取源代码，这样只是将RatuDB的代码拉取下来了，没有子项目Cassandra的代码。

```
git clone ssh://git@gitlab.ratu.ltd:30022/operation-ratudb/ratudb-opensearch.git
```

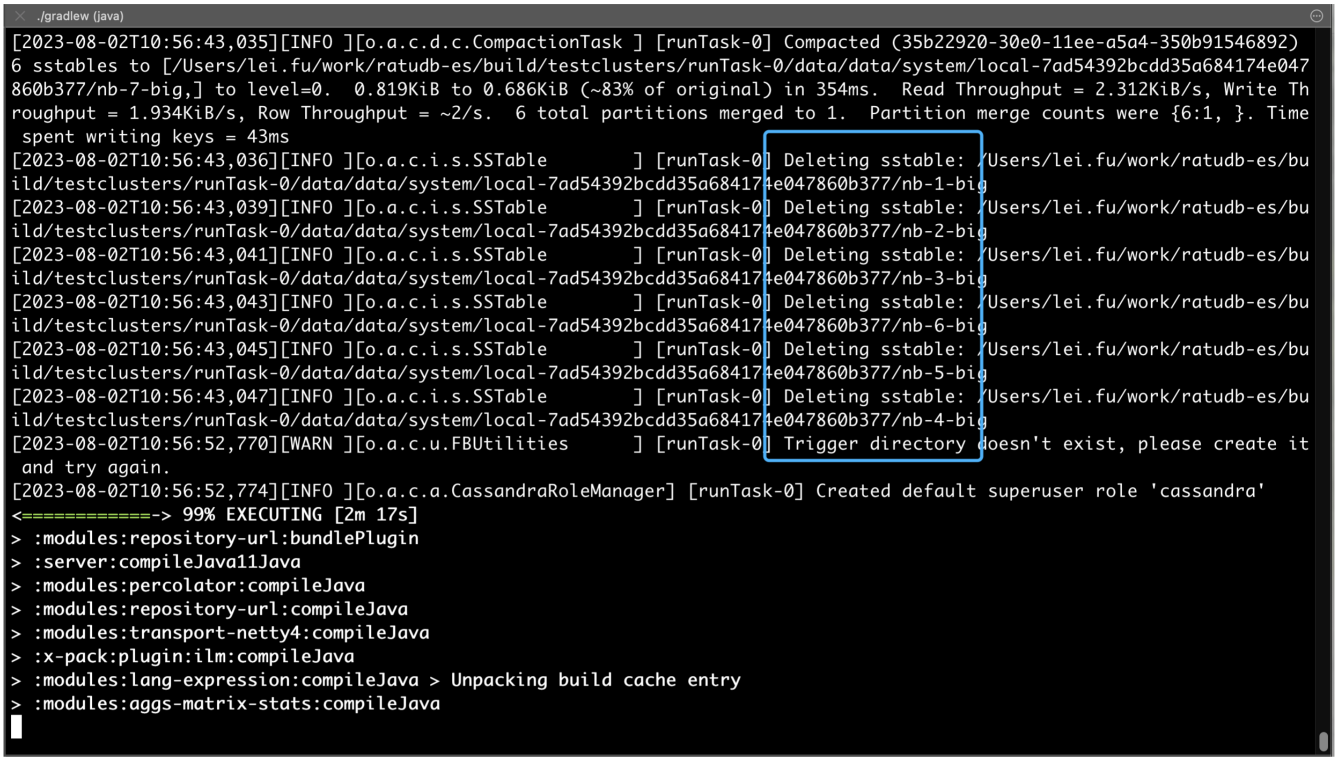
项目里，Cassandra的源码作为项目的 `submodule` 所以拉取的时候需要递归拉取，可以执行下边的命令拉取代码：

```
git clone ssh://git@gitlab.ratu.ltd:30022/operation-ratudb/ratudb-opensearch.git
--recursive
```

代码拉取完成之后，可以执行一下运行：

```
./gradlew run
```

如果配置了JDK11, 就可以运行RatuDB，就会拉取一下相关依赖的包，当然也可以直接在IDEA里边打开做同步。



看到画图的日志，是不是很熟悉，这就是Cassandra启动成功的日志。

1. JDK

本机开发的时候，需要配置两个JDK，

```
export RUNTIME_JAVA_HOME="JDK14路径"
export CASSANDRA_USE_JDK11="jdk11路径"
export JAVA_HOME="JDK11路径"
```

RUNTIME_JAVA_HOME 是ES 的运行时JDK，CASSANDRA_USE_JDK11是Cassandra的运行时JDK。

2. 构建

本身项目里，是导入了Cassandra的所有ant任务，而RatuDB启动后自动加载，Cassandra的api包。

所以在架构项目之前，建议先执行，Cassandra的api包的构建任务，当然，如果没有修改Cassandra，也可以直接使用，源码自带了一个成品包。

构建Cassandra的Api包:

```
./gradlew cassandra-mvn-install
```

这样就更新了Cassandra 的api包。

构建全平台包：

```
./gradlew assemble
```

这个命令不推荐使用，原因有两点：第一，需要连接Docker，如果Docker服务没启动，构建任务会产生失败。再有就是很慢，所以还是构建需要的平台版本，比较好，可以执行如下命令：

1.构建linux版本

```
./gradlew :distribution:archives:linux-tar:assemble
```

linux 版本输出路径：

- distribution
  - archives
    - build
    - darwin-arm64-tar
    - darwin-tar
      - build
        - distributions
          - RatuDB-1.0.0-darwin-x64.tar.gz
        - install
        - tmp
      - build.gradle
      - darwin-tar.iml
    - freebsd-tar
    - integ-test-zip
    - linux-arm64-tar
    - linux-ppc64le-tar
    - linux-s390x-tar
    - linux-tar
      - build
        - distributions
          - RatuDB-1.0.0-linux-x64.tar.gz
        - install
        - tmp
      - build.gradle
      - linux-tar.iml
    - no-jdk-darwin-arm64-tar

2.构建mac版本

```
./gradlew :distribution:archives:darwin-tar:assemble
```

mac 版本输出路径：

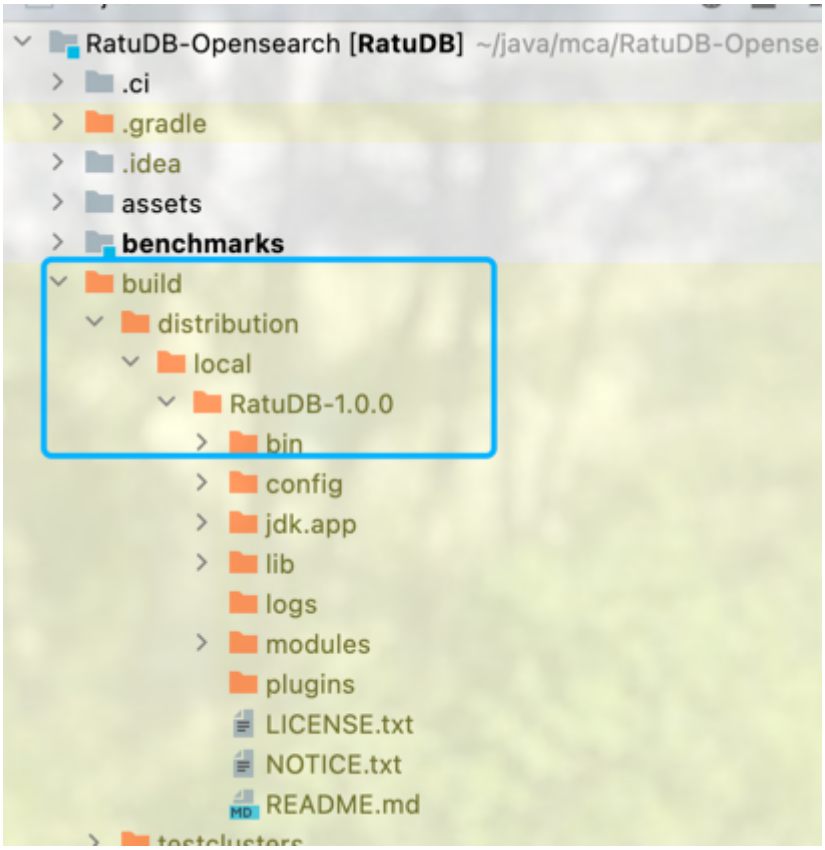
3.构建windows版本

```
./gradlew :distribution:archives:windows-zip:assemble
```

这是指定了平台版本，如果想构建本机系统版本，还有一个命令，也可以使用：

```
./gradlew localDistro
```

这个是构建一个本地系统版本，用mac就是构建mac系统的版本。输出路径：



导入项目到 IntelliJ IDEA

RatuDB 使用JDK11,全局配置JDK11就可以了。

- 选择 **File > Open**
- 在随后的对话框中导航到根目录 `build.gradle` 文件
- 在随后的对话框中选择 **Open as Project**

集群配置

举例：

1.我准备了三个**linux**的虚拟机，分别为：

```
192.168.184.31
192.168.184.32
192.168.184.33
```

JDK 只需要配置**11** 就可以了，**linux**环境不建议使用内置**jdk**，内置**jdk**使用的是 **adoptopenjdk**，内置的是**15**的版本，但是**Cassandra**大量使用了反射，而这款**jdk**在测试过程中反射问题不少。所以**linux**环境不建议使用。

环境变量需要配置在`/etc/profile`里边，如下内容：

```
export JAVA_HOME=/home/elastic/jdk-11
export PATH=$JAVA_HOME/bin:$PATH
export CLASSPATH=.:$JAVA_HOME/lib/dt.jar:$JAVA_HOME/lib/tools.jar
```

2.先打包**liunx**版本的发型包 执行：

```
./gradlew :distribution:archives:linux-tar:assemble
```

3.将**RatuDB**的压缩包上传到三台机器里并解压。

4.接下来修改配置；RatuDB 现在只需要修改 `config` 目录下的 `cassandra.yaml`

需要配置的内容：

```
cluster_name: 'Ratu Cluster'
```

集群名字，这项配置，需要集群内所有机器都是相同的才可以。但是Cassandra有个特点，如果想要修改这个集群名字，需要在Cassandra基础元数据表里先进行修改，然后再修改配置。具体可以参考网上给的配置。所以最终结论就是，当配置好后，正常启动服务了，就不要修改这项值。

```
rpc_address: 192.168.184.31
```

通信地址，这个值配置上之后，es服务的`network.host` 也使用的是这个值。

```
listen_address: 192.168.184.31
```

监听地址。

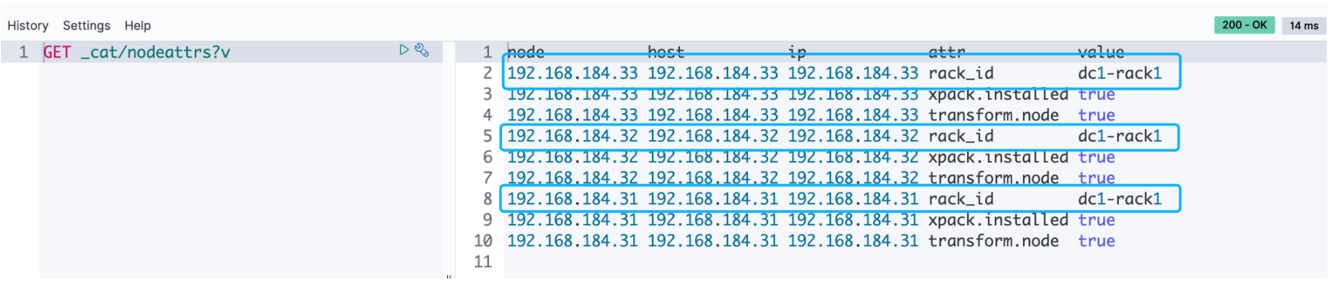
```
- seeds: "192.168.184.31:7000,192.168.184.32:7000,192.168.184.33:7000"
```

集群节点配置。ES的 `discovery.seed_hosts` 和 `cluster.initial_master_nodes` 也使用这个配置，但是 `http.port` 和 `transport.port` 就没有灵活配置了，就是默认的 `9200` 和 `9300` 。

5.数据中心和机架配置 还有一个配置在 `cassandra-rackdc.properties` 文件，这里边就配置了两个值：

```
dc=dc1
rack=rack1
```

`dc` 代表数据中心，`rack` 代表的是机架。 这两个值，对应到ES就是自定义属性，配置好就是这个样子：



```
GET _cat/nodeattrs?v
```

```
node           host           ip             attr            value
192.168.184.33 192.168.184.33 192.168.184.33 rack_id         dc1-rack1
192.168.184.33 192.168.184.33 192.168.184.33 xpack.installed true
192.168.184.33 192.168.184.33 192.168.184.33 transform.node  true
192.168.184.32 192.168.184.32 192.168.184.32 rack_id         dc1-rack1
192.168.184.32 192.168.184.32 192.168.184.32 xpack.installed true
192.168.184.32 192.168.184.32 192.168.184.32 transform.node  true
192.168.184.31 192.168.184.31 192.168.184.31 rack_id         dc1-rack1
192.168.184.31 192.168.184.31 192.168.184.31 xpack.installed true
192.168.184.31 192.168.184.31 192.168.184.31 transform.node  true
```

这个值，预设过来，可以做冷热集群使用。

6.修改数据存储目录 在cassandra.yaml文件内，通过更改`data_file_directories`可以更改cassandra和es的存储路径，如果没有配置的情况下，默认存储在当前程序目录下的`data`目录下。

关于`[parent] Data too large`异常

Opensearch 本身的断路器，是避免内存使用过大，导致的系统崩溃的一种措施。[parent]父级断路器，是在JVM内存达到95%，产生的一种异常错误。

虽然已经在内部进行了优化，但是大量数据还是存在内存进行交互，所以还是不能排除有放入内存中的数据大于本身JVM内存的情况，所以在使用异步同步数据的情况，在过大数据量的情况下，出现此错误的时候，最实在的方法还是建议增大内存。

查看内存分析：

```
GET _cat/nodes?v=true&h=name,node*,heap*
```

此API可以分析，每个集群节点的内存使用情况。

```
GET _nodes/stats/breaker
```

可以查看断路器的内存使用情况。

关于优化es的写入速度

1、客户端线程数。 es本身拥有一个queue，是一个FIFO的队列。客户端10线程写入和100线程写入性能差距是很大的。所以在感觉性能不够的时候。可以适当增加客户端线程提高速度。

虽然数据是先写入Cassandra，但是Cassandra数据，先写入的commitLog，再加载到内存表，性能确实强悍。增加线程的同时完全不用担心Cassandra的性能。

2、translog 优化 es的数据是优先写入内存的缓冲区，然后再写入translog缓冲区，然后写入FileSystemCache。然后再生成segment文件。translog本身也是日志文件，但是同步刷新到缓冲区，这性能就下来了。可以尝试：

```
PUT 索引名字/_settings?preserve_existing
{
  "index.translog.durability" : "async",
"index.translog.flush_threshold_size" : "1024mb",
  "index.translog.sync_interval" : "60s"
}
```

这可以提升性能20%左右，但是不建议使用，有可能有丢数据的情况，所以对性能不是极致要求，不建议使用。