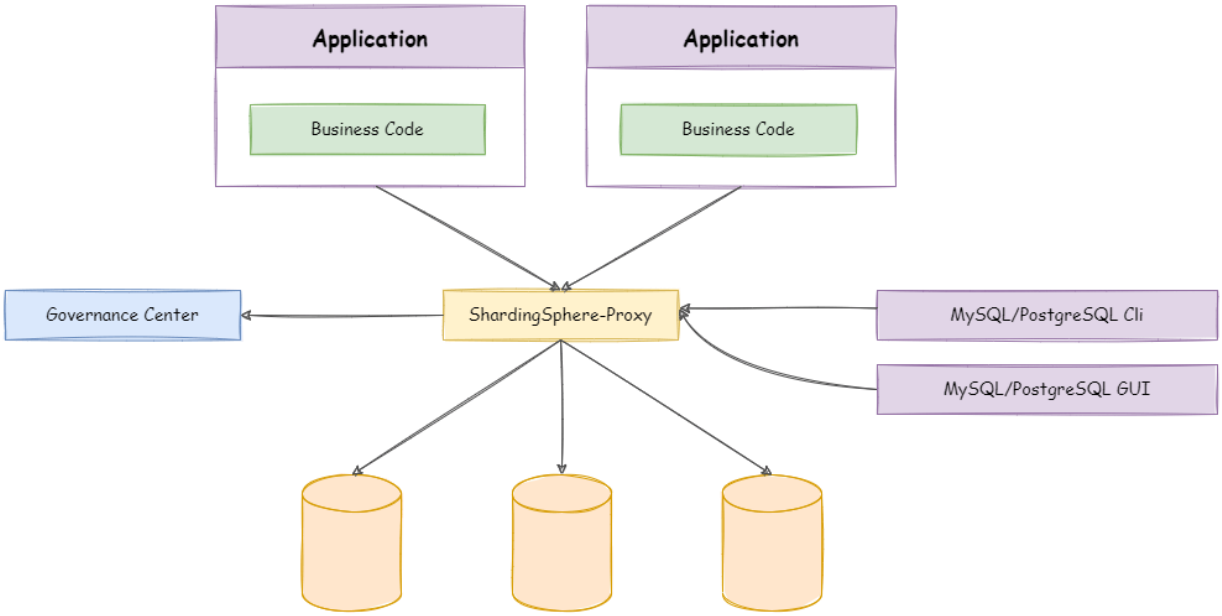
深入理解ShardingProxy服务端分库分表机制

-- 楼兰

之前章节我们已经使用ShardingJDBC完成了分库分表，其中体现了ShardingSphere很多核心的功能机制。这一章节我们来理解一下ShardingSphere中另外一个服务端分库分表场景 ShardingProxy。在使用之前，希望你不要仅仅把ShardingProxy当做一个简单的数据库产品来用，而是结合之前ShardingJDBC的理解，相互印证，相互补足。

一、为什么要有服务端分库分表?

ShardingSphere-Proxy，早前版本就叫做ShardingProxy。定位为一个透明化的数据库代理，目前提供MySQL和PostgreSQL协议，透明化数据库操作。简单理解就是，他会部署成一个MySQL或者PostgreSQL的数据库服务，应用程序只需要像操作单个数据库一样去访问ShardingSphere-proxy，由ShardingProxy去完成分库分表功能。



我们之前已经使用ShardingJDBC进行了很多分库分表的功能，为什么还需要使用ShardingProxy呢？这应该是你在开始使用ShardingProxy需要考虑的问题。 ChatGPT对于这个问题的答案是这样的的。

1、配合 ORM 框架使用更友好

当使用 ShardingSphere-JDBC 时,需要在代码中直接编写分库分表的逻辑,如果使用 ORM 框架,会产生冲突。ShardingSphere-Proxy 作为服务端中间件,可以无缝对接 ORM 框架。

2、对 DBA 更加友好

ShardingSphere-Proxy 作为服务端代理,对 DBA 完全透明。DBA 可以直接操作原始数据源,而不需要了解 ShardingSphere 的功能和 API。这简化了 DBA 的工作,也不会产生额外学习成本。

3、避免在项目中侵入分库分表逻辑

使用 ShardingSphere-JDBC,需要在业务代码中编写分库分表规则配置,这会使代码显得繁琐,且一旦规则变更,需要修改大量代码。ShardingSphere-Proxy 通过外部配置实现规则管理,可以避免这种情况。

4、提供分布式事务支持

ShardingSphere-Proxy 可以与第三方事务管理器对接,提供对分布式数据库的分布式事务支持。而 ShardingSphere-JDBC 仅支持本地事务。

5、 实现无中心化数据治理

通过 ShardingSphere-Proxy,可以将多个数据源注册到同一个代理服务中,实现跨数据源的数据治理、监控和报警等功能。这有利于大规模微服务系统的运维。

所以,ShardingSphere 提供服务端分库分表方案的主要优势是对 ORM、DBA 更加友好,可以避免侵入业务代码,提供分布式事务和数据治理支持。这些功能更加适合大规模企业级应用。

这里提到的很多重要的优势，其实就是后续使用ShardingProxy时需要重点关注的功能。

二、ShardingProxy基础使用

ShardingProxy作为一个第三方服务，部署使用已经非常简化。基本就是三个步骤，下载-配置-启动。

1、部署SharidingProxy

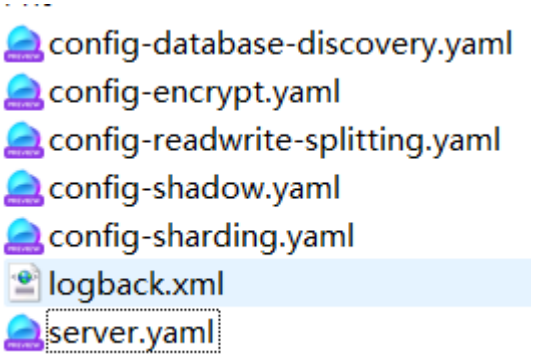
首先，获取ShardingProxy

ShardingProxy在windows和Linux上提供了一套统一的部署发布包。我们可以从ShardingSphere官网下载ShardingProxy发布包apache-shardingsphere-5.2.1-shardingsphere-proxy-bin.tar.gz，解压到本地目录。`注意不要有中文路径`

解压完成后，我们需要把MySQL的JDBC驱动包mysql-connector-java-8.0.20.jar手动复制到ShardingProxy的lib目录下。ShardingProxy默认只附带了PostgreSQL的JDBC驱动包，而不包含MySQL的JDBC驱动包。

然后，配置服务

打开conf目录，在这个目录下就有ShardingProxy的所有配置文件。配置ShardingProxy也非常简单，你基本不需要去记那些复杂的配置，每个配置文件里都给出了配置的示例，照着改改就行。`当前前提是，你理解了ShardingSphere的各种玩法`



这些配置文件的作用一目了然。server.yaml配置一些服务通用的参数。 config-sharding配置数据分片逻辑。config-encrypt配置数据加密逻辑。config-readwrite-splitting配置读写分离逻辑。这些都在之前示例中分析过。如果你熟悉了之前ShardingJDBC的示例，几乎可以零门槛看懂这些配置文件。

先打开server.yaml，把其中的rule部分和props部分注释打开

```yaml
rules:
  - !AUTHORITY
    users:
      - root@%:root
      - sharding@:sharding
    provider:
      type: ALL_PERMITTED
  - !TRANSACTION
    defaultType: XA
    providerType: Atomikos
  - !SQL_PARSER
    sqlCommentParseEnabled: true
    sqlStatementCache:
      initialCapacity: 2000
      maximumSize: 65535
    parseTreeCache:
      initialCapacity: 128
      maximumSize: 1024
```

```
props:
  max-connections-size-per-query: 1
  kernel-executor-size: 16  # Infinite by default.
  proxy-frontend-flush-threshold: 128  # The default value is 128.
  proxy-hint-enabled: false
  sql-show: false
  check-table-metadata-enabled: false
    # Proxy backend query fetch size. A larger value may increase the memory usage
of ShardingSphere Proxy.
    # The default value is -1, which means set the minimum value for different JDBC
drivers.
  proxy-backend-query-fetch-size: -1
  proxy-frontend-executor-size: 0 # Proxy frontend executor size. The default value
is 0, which means let Netty decide.
    # Available options of proxy backend executor suitable: OLAP(default), OLTP. The
OLTP option may reduce time cost of writing packets to client, but it may increase
the latency of SQL execution
    # and block other clients if client connections are more than `proxy-frontend-
executor-size`, especially executing slow SQL.
  proxy-backend-executor-suitable: OLAP
  proxy-frontend-max-connections: 0 # Less than or equal to 0 means no limitation.
    # Available sql federation type: NONE (default), ORIGINAL, ADVANCED
  sql-federation-type: NONE
    # Available proxy backend driver type: JDBC (default), ExperimentalVertx
  proxy-backend-driver-type: JDBC
  proxy-mysql-default-version: 5.7.22 # In the absence of schema name, the default
version will be used.
  proxy-default-port: 3307 # Proxy default port.
  proxy-netty-backlog: 1024 # Proxy netty backlog.
```

rules下的AUTHORITY部分配置ShardingProxy的用户以及权限。 TRANSACTION部分维护的是事务控制器，下一章节再做分析。

props部分配置服务端的一些参数。max-connections-size-per-query参数是对于ShardingProxy最重要的优化参数，在上一章节介绍ShardingSphere的执行引擎以及结果归并时介绍到了。proxy-mysql-default-version表示ShardingProxy所模拟的MySQL服务版本。为了与之前的示例兼容，我们这里可以将它改成8.0.20版本。proxy-default-port表示模拟的MySQL服务的端口。

修改完成后，就可以启动ShardingProxy了。 启动脚本在bin目录下。

```
E:\a-work\shardingsphere\5.2.1\apache-shardingsphere-5.2.1-shardingsphere-proxy-bin\bin>start.bat
we find java version: java8, full_version=1.8.0_45"
Starting the ShardingSphere-Proxy ...
[INFO ] 2023-04-14 14:22:37.557 [main] o.a.s.p.frontend.ShardingSphereProxy - ShardingSphere-Proxy Standalone mode start
ed successfully
```

服务启动后，就可以使用mysql客户端直接访问ShardingProxy了。

```
D:\dev-hook\mysql-8.0.20-winx64\bin>mysql.exe -u root -P 3307 -p
Enter password: ****
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 4
Server version: 8.0.20-ShardingSphere-Proxy 5.2.1

Copyright (c) 2000, 2020, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql>
```

然后，你可以像用MySQL一样去使用shardingProxy

```
mysql> show databases;
+--------------------+
| schema_name        |
+--------------------+
| shardingsphere     |
| information_schema |
| performance_schema |
| mysql              |
| sys                |
+--------------------+
5 rows in set (0.01 sec)

mysql> use shardingsphere
Database changed
mysql> show tables;
+----------------------------+------------+
| Tables_in_shardingsphere   | Table_type |
+----------------------------+------------+
| sharding_table_statistics | BASE TABLE |
+----------------------------+------------+
1 row in set (0.01 sec)

mysql> select * from sharding_table_statistics;
Empty set (1.25 sec)
```

2、配置常用分库分表策略

当然，现在ShardingProxy里还没什么东西，因为还没有配置逻辑表。打开config-sharding.xml，像我们之前章节使用shardingJDBC时一样，配置逻辑表course。

```
databaseName: sharding_db

dataSources:
  m0:
    url: jdbc:mysql://127.0.0.1:3306/coursedb?serverTimezone=UTC&useSSL=false
```

```
    username: root
    password: root
    connectionTimeoutMilliseconds: 30000
    idleTimeoutMilliseconds: 60000
    maxLifetimeMilliseconds: 1800000
    maxPoolSize: 50
    minPoolSize: 1
  m1:
    url: jdbc:mysql://127.0.0.1:3306/coursedb2?serverTimezone=UTC&useSSL=false
    username: root
    password: root
    connectionTimeoutMilliseconds: 30000
    idleTimeoutMilliseconds: 60000
    maxLifetimeMilliseconds: 1800000
    maxPoolSize: 50
    minPoolSize: 1

rules:
- !SHARDING
  tables:
    course:
      actualDataNodes: m${0..1}.course_${1..2}
      databaseStrategy:
        standard:
          shardingColumn: cid
          shardingAlgorithmName: course_db_alg
      tableStrategy:
        standard:
          shardingColumn: cid
          shardingAlgorithmName: course_tbl_alg
      keyGenerateStrategy:
        column: cid
        keyGeneratorName: alg_snowflake

  shardingAlgorithms:
    course_db_alg:
      type: MOD
      props:
        sharding-count: 2
    course_tbl_alg:
      type: INLINE
      props:
        algorithm-expression: course_$->{cid%2+1}

  keyGenerators:
    alg_snowflake:
      type: SNOWFLAKE
```

粗略一看，感觉挺复杂。但是，对照一下之前ShardingJDBC的配置，除了Groovy表达式中，用${}来表达变量外，其他配置几乎是零门槛的。

然后重新启动ShardingProxy服务，再看看服务中有哪些东西。

```
mysql> show databases;
+--------------------+
| schema_name        |
+--------------------+
| information_schema |
| performance_schema |
| sys                |
| shardingsphere     |
| sharding_db        |
| mysql              |
+--------------------+
6 rows in set (0.01 sec)

mysql> use sharding_db;
Database changed
mysql> show tables;
+------------------------+------------+
| Tables_in_sharding_db | Table_type |
+------------------------+------------+
| dict_2                 | BASE TABLE |
| dict_1                 | BASE TABLE |
| user_2                 | BASE TABLE |
| user_1                 | BASE TABLE |
| user_course_info_1     | BASE TABLE |
| user_course_info       | BASE TABLE |
| user_course_info_2     | BASE TABLE |
| course                 | BASE TABLE |
| dict                   | BASE TABLE |
| user                   | BASE TABLE |
+------------------------+------------+
10 rows in set (0.01 sec)

mysql> select * from course;
+---------------------+-------+---------+---------+
| cid                 | cname | user_id | cstatus |
+---------------------+-------+---------+---------+
|  853625816750227456 | java  |    1001 | 1       |
.....
+---------------------+-------+---------+---------+
34 rows in set (0.05 sec)

mysql> select * from course_1;
+--------------------+-------+---------+---------+
| cid                | cname | user_id | cstatus |
+--------------------+-------+---------+---------+
| 853625816750227456 | java  |    1001 | 1       |
.....
+--------------------+-------+---------+---------+
5 rows in set (0.01 sec)
```

> 这也解释了在使用ShardingJDBC时，大家经常会问的一个问题，就是对于数据库中没有配置虚拟表的真实表，要怎么查。这里就给出了答案。

接下来，在之前ShardingJDBC示例中配置的各种分库分表的组件，你都可以行尝试一下。

二、ShardinSphere中的分布式事务机制

如果你比较仔细，会发现，在之前的配置中，server.yaml中的rules部分，还有一个不太眼熟的配置TRANSACTION 分布式事务管理器。

```
rules:
  - !TRANSACTION
    defaultType: XA
    providerType: Atomikos
```

由于ShardingSphere是需要操作分布式的数据库集群，所以数据库内部的本地事务机制是无法保证ShardingProxy中的事务安全的，这就需要引入分布式事务管理机制，保证ShardingProxy中的SQL语句执行的原子性。也就是说，在ShardingProxy中打开分布式事务机制后，你就不需要考虑SQL语句执行时的分布式事务问题了。

1、什么是XA事务?

这其中XA是由X/Open Group组织定义的，处理分布式事务的标准。主流的关系型数据库产品都实现了XA协议。例如，MySQL从5.0.3版本开始，就已经可以直接支持XA事务了。但是要注意，只有InnoDB引擎才提供支持：

```
//1、 XA START|BEGIN 开启事务，这个test就相当于是事务ID，将事务置于ACTIVE状态
XA START 'test';
//2、对一个ACTIVE状态的XA事务，执行构成事务的SQL语句。
insert...//business sql
//3、发布一个XA END指令，将事务置于IDLE状态
XA END 'test'; //事务结束
//4、对于IDLE状态的XACT事务，执行XA PREPARED指令 将事务置于PREPARED状态。
//也可以执行 XA COMMIT 'test' ON PHASE 将预备和提交一起操作。
XA PREPARE 'test'; //准备事务
//PREPARED状态的事务可以用XA RECOVER指令列出。列出的事务ID会包含gtrid,bqual,formatID和data四个字段。
XA RECOVER;
//5、对于PREPARED状态的XA事务，可以进行提交或者回滚。
XA COMMIT 'test'; //提交事务
XA ROLLBACK 'test'; //回滚事务。
```

在这个标准下有多种具体的实现框架。ShardingSphere集成了Atomikos、Bitronix和Narayana三个框架。其中在ShardingProxy中默认只集成了Atomikos实现。

5.10.2 XATransactionManagerProvider

全限定类名

org.apache.shardingsphere.transaction.xa.spi.XATransactionManagerProvider

定义

XA 分布式事务管理器

已知实现

配置标识 *	详细说明	全限定类名 *
Atomikos	基于 Atomikos 的 XA 分布式事务管理器	org.ap ache.shardi ngsphere.transaction.x a.atomikos. manager.Ato mikosTransa ction-Manage rProvider |
Narayana	基于 Narayana 的 XA 分布式事务管理器	org.apac he.sharding sphere.tran saction.xa. narayana.ma nager.Naray anaXATransa ctionManage rProvider |
Bitronix	基于 Bitronix 的 XA 分布式事务管理器	org.apac he.sharding sphere.tran saction.xa. bitronix.ma nager.Bitro nixXA-Transa ctionManage rProvider |

2、实战理解XA事务

回到之前的ShardingJDBC示例项目，我们做一个简单的示例来理解一下XA事务。

引入Maven依赖

```
<!--XA 分布式事务 -->
        <dependency>
            <groupId>org.apache.shardingsphere</groupId>
            <artifactId>shardingsphere-transaction-xa-core</artifactId>
            <version>5.2.1</version>
            <exclusions>
                <exclusion>
                    <artifactId>transactions-jdbc</artifactId>
                    <groupId>com.atomikos</groupId>
                </exclusion>
                <exclusion>
                    <artifactId>transactions-jta</artifactId>
```

```
                    <groupId>com.atomikos</groupId>
                </exclusion>
            </exclusions>
        </dependency>
        <!-- 版本滞后了 -->
        <dependency>
            <artifactId>transactions-jdbc</artifactId>
            <groupId>com.atomikos</groupId>
            <version>5.0.8</version>
        </dependency>
        <dependency>
            <artifactId>transactions-jta</artifactId>
            <groupId>com.atomikos</groupId>
            <version>5.0.8</version>
        </dependency>

<!-- 使用XA事务时，可以引入其他几种事务管理器 -->
<!--         <dependency>-->
<!--             <groupId>org.apache.shardingsphere</groupId>-->
<!--             <artifactId>shardingsphere-transaction-xa-bitronix</artifactId>-->
<!--             <version>5.2.1</version>-->
<!--         </dependency>-->
<!--         <dependency>-->
<!--             <groupId>org.apache.shardingsphere</groupId>-->
<!--             <artifactId>shardingsphere-transaction-xa-narayana</artifactId>-->
<!--             <version>5.2.1</version>-->
<!--         </dependency>-->
```

配置事务管理器

```
@Configuration
@EnableTransactionManagement
public class TransactionConfiguration {

    @Bean
    public PlatformTransactionManager txManager(final DataSource dataSource) {
        return new DataSourceTransactionManager(dataSource);
    }
}
```

然后就可以写一个示例

```
public class MySQLXAConnectionTest {
    public static void main(String[] args) throws SQLException {
        //true表示打印XA语句,, 用于调试
        boolean logXaCommands = true;
        // 获得资源管理器操作接口实例 RM1
        Connection conn1 =
DriverManager.getConnection("jdbc:mysql://localhost:3306/coursedb?
serverTimezone=UTC", "root", "root");
```

```
        XAConnection xaConn1 = new
MysqlXAConnection((com.mysql.cj.jdbc.JdbcConnection) conn1, logXaCommands);
        XAResource rm1 = xaConn1.getXAResource();
        // 获得资源管理器操作接口实例 RM2
        Connection conn2 =
DriverManager.getConnection("jdbc:mysql://localhost:3306/coursedb2?
serverTimezone=UTC", "root", "root");
        XAConnection xaConn2 = new
MysqlXAConnection((com.mysql.cj.jdbc.JdbcConnection) conn2, logXaCommands);
        XAResource rm2 = xaConn2.getXAResource();
        // AP请求TM执行一个分布式事务，TM生成全局事务id
        byte[] gtrid = "g12345".getBytes();
        int formatId = 1;
        try {
            // ==============分别执行RM1和RM2上的事务分支====================
            // TM生成rm1上的事务分支id
            byte[] bqual1 = "b00001".getBytes();
            Xid xid1 = new MysqlXid(gtrid, bqual1, formatId);
            // 执行rm1上的事务分支
            rm1.start(xid1, XAResource.TMNOFLAGS);//One of TMNOFLAGS, TMJOIN, or
TMRESUME.
            PreparedStatement ps1 = conn1.prepareStatement("INSERT INTO `dict`
VALUES (1, 'T', '测试1');");
            ps1.execute();
            rm1.end(xid1, XAResource.TMSUCCESS);
            // TM生成rm2上的事务分支id
            byte[] bqual2 = "b00002".getBytes();
            Xid xid2 = new MysqlXid(gtrid, bqual2, formatId);
            // 执行rm2上的事务分支
            rm2.start(xid2, XAResource.TMNOFLAGS);
            PreparedStatement ps2 = conn2.prepareStatement("INSERT INTO `dict`
VALUES (2, 'F', '测试2');");
            ps2.execute();
            rm2.end(xid2, XAResource.TMSUCCESS);
            // ===================两阶段提交================================
            // phase1：询问所有的RM 准备提交事务分支
            int rm1_prepare = rm1.prepare(xid1);
            int rm2_prepare = rm2.prepare(xid2);
            // phase2：提交所有事务分支
            boolean onePhase = false; //TM判断有2个事务分支，所以不能优化为一阶段提交
            if (rm1_prepare == XAResource.XA_OK
                    && rm2_prepare == XAResource.XA_OK
            ) {//所有事务分支都prepare成功，提交所有事务分支
                rm1.commit(xid1, onePhase);
                rm2.commit(xid2, onePhase);
            } else {//如果有事务分支没有成功，则回滚
                rm1.rollback(xid1);
                rm1.rollback(xid2);
            }
        } catch (XAException e) {
            // 如果出现异常，也要进行回滚
            e.printStackTrace();
```

```
        }
    }
}
```

这其中，XA标准规范了事务XID的格式。有三个部分: gtrid [, bqual [, formatID]] 其中

- gtrid 是一个全局事务标识符 global transaction identifier
- bqual 是一个分支限定符 branch qualifier 。如果没有提供，会使用默认值就是一个空字符串。
- formatID 是一个数字，用于标记gtrid和bqual值的格式，这是一个正整数，最小为0，默认值就是1。

但是使用XA事务时需要注意以下几点：

- XA事务无法自动提交
- XA事务效率非常低下，全局事务的状态都需要持久化。性能非常低下，通常耗时能达到本地事务的10倍。
- XA事务在提交前出现故障的话，很难将问题隔离开。

3、如何在ShardingProxy中使用另外两种事务管理器?

例如如果希望在ShardingProxy中使用narayana事务管理器，只需要两个步骤：

1、将narayana的事务集成Jar包 shardingsphere-transaction-xa-narayana-5.2.1.jar 放入到ShardingProxy的lib目录下。

> 这个jar包可以通过Maven依赖下载

2、在server.yaml中就可以将事务的Provider配置成Narayana

```
rules:
  - !TRANSACTION
    defaultType: XA
    providerType: Narayana
```

> 这个字符串Narayana是哪里来的？按照之前的思路，看一下**XATransactionManagerProvider**的实现类你就知道了。
>
> 但是要注意有些组件版本冲突的问题！最近几个ShardingSphere的版本太新了，有些依赖没有维护好。

三、ShardingProxy集群化部署

1、理解ShardingProxy运行模式

在之前测试中，对于server.yaml文件，还有一段mod配置，没有打开注释。这是干什么用的?

```
#mode:
#  type: Cluster
#  repository:
#    type: ZooKeeper
#    props:
#      namespace: governance_ds
#      server-lists: localhost:2181
#      retryIntervalMilliseconds: 500
#      timeToLiveSeconds: 60
#      maxRetries: 3
#      operationTimeoutMilliseconds: 500
```

这个表示ShardingSphere的运行模式。简单理解也就是ShardingSphere怎么管理这么多复杂的配置信息。ShardingSphere支持两种运行模式，Standalone独立模式和Cluster集群模式。

在Standalone独立模式下，ShardingSphere不需要考虑其他实例的影响，直接在内存中管理核心配置规则就可以了。他是ShardingSphere默认的运行模式。

而在Cluster集群模式下，ShardingSphere不光要考虑自己的配置规则，还需要考虑如何跟集群中的其他实例同步自己的配置规则。这就需要引入第三方组件来提供配置信息同步。ShardingSphere目前支持的配置中心包括：Zookeeper、etcd、Nacos、Consule。但是在ShardingSphere分库分表的场景下，这些配置信息几乎不会变动，访问频率也不会太高。所以，最为推荐的，是基于CP架构的Zookeeper。另外，如果应用的本地和Zookeeper中都有配置信息，那么ShardingSphere会以Zookeeper中的配置为准。

在进行选择时，Standaloe适用于小型项目或对性能要求较高的场景，比较适合配合ShardingJDBC使用。Cluster适合大规模集群环境，比较适合配合ShardingProxy使用。

2、使用Zookeeper进行集群部署

接下来我们可以基于Zookeeper部署一下ShardingProxy集群，看一下ShardingSphere需要同步的配置有哪些。

我们只需要在本地部署一个Zookeeper，然后将server.yaml中的mode部分解除注释：

```
mode:
  type: Cluster
  repository:
    type: ZooKeeper
    props:
      namespace: governance_ds
      server-lists: localhost:2181
      retryIntervalMilliseconds: 500
      timeToLiveSeconds: 60
      maxRetries: 3
      operationTimeoutMilliseconds: 500
```

启动ShardingProxy服务后，可以看到Zookeeper注册中心的信息如下是：

```
namespace
├──rules # 全局规则配置
├──props # 属性配置
```

```
├──metadata # Metadata 配置
├ ├──${databaseName} # 逻辑数据库名称
├ ├ ├──schemas # Schema 列表
├ ├ ├ ├──${schemaName} # 逻辑 Schema 名称
├ ├ ├ ├ ├──tables # 表结构配置
├ ├ ├ ├ ├ ├──${tableName}
├ ├ ├ ├ ├ ├──...
├ ├ ├ ├──...
├ ├ ├──versions # 元数据版本列表
├ ├ ├ ├ ├──views # 视图结构配置
├ ├ ├ ├ ├ ├──${viewName}
├ ├ ├ ├ ├ ├──...
├ ├ ├ ├──${versionNumber} # 元数据版本号
├ ├ ├ ├ ├──dataSources # 数据源配置
├ ├ ├ ├ ├──rules # 规则配置
├ ├ ├ ├──...
├ ├ ├──active_version # 激活的元数据版本号
├ ├──...
├──nodes
├ ├──compute_nodes
├ ├ ├──online
├ ├ ├ ├──proxy
├ ├ ├ ├ ├──UUID # Proxy 实例唯一标识
├ ├ ├ ├ ├──....
├ ├ ├ ├──jdbc
├ ├ ├ ├ ├──UUID # JDBC 实例唯一标识
├ ├ ├ ├ ├──....
├ ├ ├──status
├ ├ ├ ├──UUID
├ ├ ├ ├──....
├ ├ ├──worker_id
├ ├ ├ ├──UUID
├ ├ ├ ├──....
├ ├ ├──process_trigger
├ ├ ├ ├──process_list_id:UUID
├ ├ ├ ├──....
├ ├ ├──labels
├ ├ ├ ├──UUID
├ ├ ├ ├──....
├ ├──storage_nodes
├ ├ ├──${databaseName.groupName.ds}
├ ├ ├──${databaseName.groupName.ds}
```

而在rules部分，就是我们配置的ShardingProxy的核心属性

```
- !AUTHORITY
  provider:
    type: ALL_PERMITTED
  users:
  - root@%:root
  - sharding@%:sharding
```

```
- !TRANSACTION
  defaultType: XA
  providerType: Atomikos
- !SQL_PARSER
  parseTreeCache:
    initialCapacity: 128
    maximumSize: 1024
  sqlCommentParseEnabled: true
  sqlStatementCache:
    initialCapacity: 2000
    maximumSize: 65535
```

而分库分表的信息，则配置在/governance_ds/metadata/sharding_db/versions/0/rules节点下

```
- !SHARDING
  keyGenerators:
    alg_snowflake:
      type: SNOWFLAKE
  shardingAlgorithms:
    course_db_alg:
      props:
        sharding-count: 2
      type: MOD
    course_tbl_alg:
      props:
        algorithm-expression: course_$->{cid%2+1}
      type: INLINE
  tables:
    course:
      actualDataNodes: m${0..1}.course_${1..2}
      databaseStrategy:
        standard:
          shardingAlgorithmName: course_db_alg
          shardingColumn: cid
      keyGenerateStrategy:
        column: cid
        keyGeneratorName: alg_snowflake
      logicTable: course
      tableStrategy:
        standard:
          shardingAlgorithmName: course_tbl_alg
          shardingColumn: cid
```

3、统一ShardingJDBC和ShardingProxy配置信息

这时，回过头来想一想，既然ShardingProxy可以通过Zookeeper同步配置信息，那么我们可不可以在ShardingJDBC中也采用Zookeeper的配置呢？当然是可以的。

1、通过注册中心同步配置

第一种简单的思路就是将ShardingProxy中的mod部分配置移植到之前的ShardingJDBC示例中。

将application.properties中的配置信息全部删除，只配置Zookeeper地址：

```
spring.shardingsphere.mode.type=Cluster
spring.shardingsphere.mode.repository.type=ZooKeeper
spring.shardingsphere.mode.repository.props.namespace=governance_ds
spring.shardingsphere.mode.repository.props.server-lists=localhost:2181
spring.shardingsphere.mode.repository.props.retryIntervalMilliseconds=600
spring.shardingsphere.mode.repository.props.timeToLiveSecoonds=60
spring.shardingsphere.mode.repository.props.maxRetries=3
spring.shardingsphere.mode.repository.props.operationTimeoutMilliseconds=500
```

然后，就可以继续验证对course表的分库分表操作了。

2、直接使用ShardingProxy提供的JDBC驱动读取配置文件

ShardingSphere一直以来都是通过兼容MySQL或者PostgreSQL服务的方式，提供分库分表功能。应用端可以通过MySQL或者PostgreSQL的JDBC驱动来访问ShardignSphereDataSource。而在当前版本中，ShardingSphere则在这条道路上又往前进了一大步。直接提供了自己的JDBC驱动。

例如在之前ShardingJDBC的classpath下增加一个config.xml，然后将我们之前在ShardingProxy中的几个关键配置整合到一起

```
rules:
  - !AUTHORITY
    users:
      - root@%:root
      - sharding@:sharding
    provider:
      type: ALL_PERMITTED
  - !TRANSACTION
    defaultType: XA
    providerType: Atomikos
  - !SQL_PARSER
    sqlCommentParseEnabled: true
    sqlStatementCache:
      initialCapacity: 2000
      maximumSize: 65535
    parseTreeCache:
      initialCapacity: 128
      maximumSize: 1024
  - !SHARDING
    tables:
      course:
        actualDataNodes: m${0..1}.course_${1..2}
        databaseStrategy:
          standard:
            shardingColumn: cid
            shardingAlgorithmName: course_db_alg
        tableStrategy:
          standard:
            shardingColumn: cid
            shardingAlgorithmName: course_tbl_alg
```

```
        keyGenerateStrategy:
          column: cid
          keyGeneratorName: alg_snowflake

    shardingAlgorithms:
      course_db_alg:
        type: MOD
        props:
          sharding-count: 2
      course_tbl_alg:
        type: INLINE
        props:
          algorithm-expression: course_$->{cid%2+1}

    keyGenerators:
      alg_snowflake:
        type: SNOWFLAKE

props:
  max-connections-size-per-query: 1
  kernel-executor-size: 16  # Infinite by default.
  proxy-frontend-flush-threshold: 128  # The default value is 128.
  proxy-hint-enabled: false
  sql-show: false
  check-table-metadata-enabled: false
    # Proxy backend query fetch size. A larger value may increase the memory usage
of ShardingSphere Proxy.
  # The default value is -1, which means set the minimum value for different JDBC
drivers.
  proxy-backend-query-fetch-size: -1
  proxy-frontend-executor-size: 0 # Proxy frontend executor size. The default value
is 0, which means let Netty decide.
    # Available options of proxy backend executor suitable: OLAP(default), OLTP. The
OLTP option may reduce time cost of writing packets to client, but it may increase
the latency of SQL execution
  # and block other clients if client connections are more than `proxy-frontend-
executor-size`, especially executing slow SQL.
  proxy-backend-executor-suitable: OLAP
  proxy-frontend-max-connections: 0 # Less than or equal to 0 means no limitation.
  # Available sql federation type: NONE (default), ORIGINAL, ADVANCED
  sql-federation-type: NONE
  # Available proxy backend driver type: JDBC (default), ExperimentalVertx
  proxy-backend-driver-type: JDBC
  proxy-mysql-default-version: 8.0.20 # In the absence of schema name, the default
version will be used.
  proxy-default-port: 3307 # Proxy default port.
  proxy-netty-backlog: 1024 # Proxy netty backlog.

databaseName: sharding_db
dataSources:
  m0:
    #这个参数必须新增
```

```
    dataSourceClassName: com.zaxxer.hikari.HikariDataSource
    url: jdbc:mysql://127.0.0.1:3306/coursedb?serverTimezone=UTC&useSSL=false
    username: root
    password: root
    connectionTimeoutMilliseconds: 30000
    idleTimeoutMilliseconds: 60000
    maxLifetimeMilliseconds: 1800000
    maxPoolSize: 50
    minPoolSize: 1
  m1:
    #这个参数必须新增
    dataSourceClassName: com.zaxxer.hikari.HikariDataSource
    url: jdbc:mysql://127.0.0.1:3306/coursedb2?serverTimezone=UTC&useSSL=false
    username: root
    password: root
    connectionTimeoutMilliseconds: 30000
    idleTimeoutMilliseconds: 60000
    maxLifetimeMilliseconds: 1800000
    maxPoolSize: 50
    minPoolSize: 1
```

然后，可以直接用JDBC的方式访问带有分库分表的虚拟库。

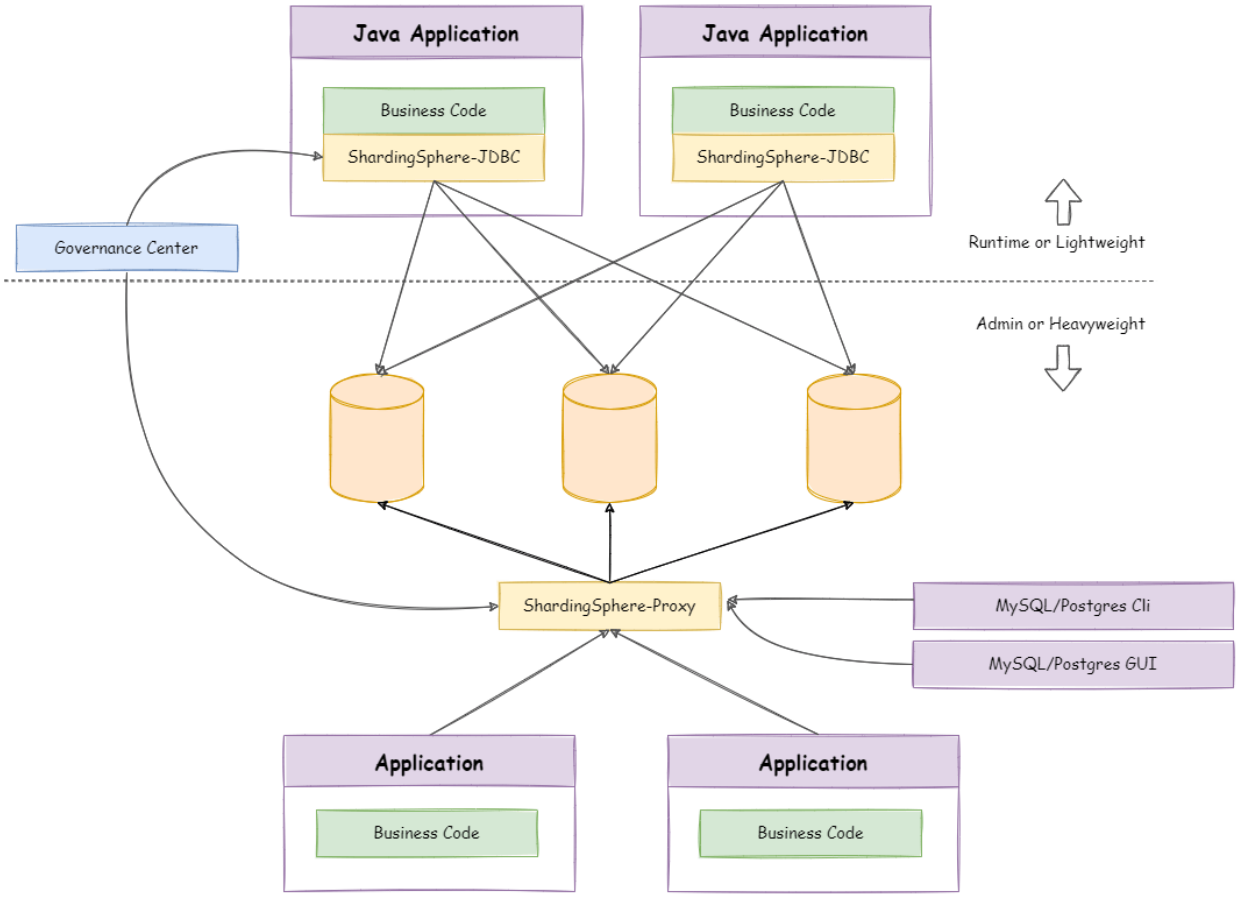
```
public class ShardingJDBCDriverTest {

    @Test
    public void test() throws ClassNotFoundException, SQLException {
        String jdbcDriver = "org.apache.shardingsphere.driver.ShardingSphereDriver";
        String jdbcUrl = "jdbc:shardingsphere:classpath:config.yaml";
        String sql = "select * from sharding_db.course";

        Class.forName(jdbcDriver);
        try(Connection connection = DriverManager.getConnection(jdbcUrl);) {
            Statement statement = connection.createStatement();
            ResultSet resultSet = statement.executeQuery(sql);
            while (resultSet.next()){
                System.out.println("course cid= "+resultSet.getLong("cid"));
            }
        }
    }
}
```

> 官方的说明是ShardingSphereDriver读取config.yaml时， 这个config.yaml配置信息与ShardingProxy中的配置文件完全是相同的，你甚至可以直接将ShardingProxy中的配置文件拿过来用。但是从目前版本来看，还是有不少小问题的。静待后续版本跟踪把。

到这里，你对于之前介绍的ShardingSphere的混合架构，有没有更新的了解？



四、ShardingProxy功能扩展

其实到这，你应该已经对ShardingProxy非常熟练了。最后就补充够一个在ShardingProxy中进行自定义扩展的方式。在ShardingProxy中，只需要将自定义的扩展功能按照SPI机制的要求打成jar包，就可以直接把jar包放入lib目录，然后就配置使用了。

例如，之前在ShardingJDBC章节我们已经创建了一个自己扩展的主键生成策略。MyKeyGeneratorAlgorithm

```
public class MyKeyGeneratorAlgorithm implements KeyGenerateAlgorithm {
    private AtomicLong atom = new AtomicLong(0);
    private Properties props;

    @Override
    public Comparable<?> generateKey() {
        LocalDateTime ldt = LocalDateTime.now();
        String timestampS = DateTimeFormatter.ofPattern("HHmmssSSS").format(ldt);
        return Long.parseLong(""+timestampS+atom.incrementAndGet());
    }
    @Override
    public Properties getProps() {
        return this.props;
    }
    public String getType() {
        return "MYKEY";
    }
```

```
    @Override
    public void init(Properties props) {
        this.props = props;
    }
}
```

然后，我们只需要将这个类以及对应的SPI文件打成一个Jar包，放到ShardingProxy的lib目录下就可以使用了。使用的方式跟在ShardingJDBC中一样，配置到主键生成算法中就行。这里就不多说了。

只补充一个在之前的ShardingJDBC项目中，单独打功能扩展jar包的方式。在pom.xml中引入maven-jar-plugin插件就可以了。

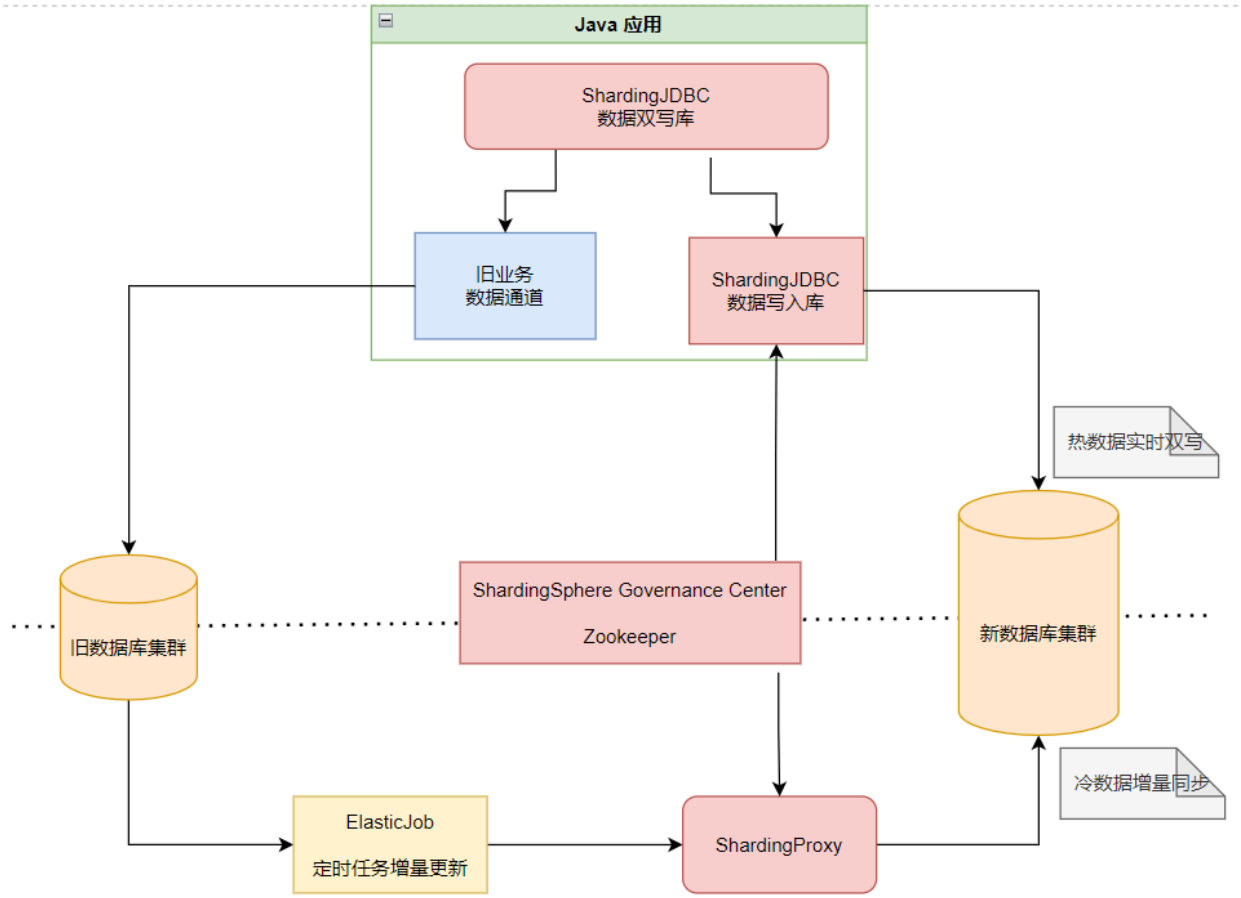
```
<build>
        <plugins>
            <!-- 将SPI扩展功能单独打成jar包 -->
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-jar-plugin</artifactId>
                <version>3.2.0</version>
                <executions>
                    <execution>
                        <id>ShardingSPIDemo</id>
                        <phase>package</phase>
                        <goals>
                            <goal>jar</goal>
                        </goals>
                        <configuration>
                            <classifier>spiextention</classifier>
                            <includes>
                                <include>com/roy/shardingDemo/algorithm/*</include>
                                <include>META-INF/services/*</include>
                            </includes>
                        </configuration>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
```

五、分库分表数据迁移方案

对于分库分表场景，还有一个非常让人头疼的事情，就是数据迁移。之前项目没有分库分表，现在数据大了之后要进行分库分表改造。或者数据采用取模分片，发现数据量太大了，需要增加数据分片数量，等等这些场景都需要进行数据迁移。而分库分表通常面临的就是海量数据的场景，这使得数据迁移通常是一个非常庞大，非常耗时的工作。业界有很多零迁移数据扩缩容方案可以用来预防未来可能需要的数据迁移工作。但是，如果你没有提前进行设计，在调整分片方案时，确实需要进行数据迁移，应该怎么办呢？

数据迁移的难点往往不在于数据怎么转移，而是在数据转移的过程中，如何保证不影响业务正常进行。通常的思路都是冷热数据分开迁移。冷数据是指那些存量的历史数据。这一部分数据往往数据量非常大，不可能一次性迁移完成，那就只能用定时任务的方式，一点点的逐步完成迁移。热数据是指那些业务进行过程当中产生的实时数据。这一部分数据就要保证数据迁移过程中实时双写。在冷数据迁移过程中，既要写入旧数据库当中，保证业务正常运行，同时又要写入新的数据库集群当中，保证数据正常更新。等冷数据迁移完成后，再将旧数据库完全淘汰，用新的数据库集群承载业务。

目标清晰了，具体怎么设计过渡方案呢？实际上，你不是在孤军作战，ShardingSphere也在考虑这个问题。ShardingSphere已经包含了一个子项目ElasticJob可以帮助定制定时任务调度。在ShardingSphere未来的规划中，也设计一个Scaling组件。预计结合ElasticJob定制出一个比较标准的数据迁移指导方案。但是，在这之前，我们其实可以利用ShardingSphere的混合架构来辅助进行分库分表数据迁移。



热数据可以在旧业务数据通道外，通过ShardingJDBC往新的数据库进行实时双写。在这里主要是要考虑尽量少的影响旧业务的数据通道。而我们要做的， 就是用一个ShardingSphereDataSource，去替换旧的DataSource。首先可以在应用中配置一个ShardingJDBC数据双写库。在这个库中主要是让核心业务表能够保持在新旧两套数据库集群中同时进行双写。在这个过程中，旧的业务通道不需要做任何修改，只要在ShardingJDBC数据双写库中针对要迁移的核心业务表配置分片规则这就行。`数据双写可以通过定制分片算法实现。`

配置完数据双写后，就需要针对新数据库集群配置，配置一个ShardingJDBC数据写入库，主要完整针对细新数据库集群的数据分片。这个ShardingJDBC数据写入库可以使用ShardingSphere的JDBC来创建，然后作为一个真实库，配置到之前的数据双写库当中，这样就可以完成针对新数据库集群的数据分片写入

冷数据部分主要是查询旧数据库中的数据，按照新的数据分片规则，转移到新数据库集群当中。这部分数据通常非常巨大，对内存的消耗非常大。所以可以通过定时任务进行增量更新，每次只读一部分数据。然后，为了简化数据转移的逻辑，可以搭建一个ShardingProxy服务，用来完成针对新数据库集群的分片规则。这样定时任务的逻辑就比较简单了，只要从一个MySQL服务读数据，然后写入另一个MySQL服务就可以了。而为了保持在写入新数据库集群时，与热数据保持相同的分片逻辑，ShardingProxy与ShardingJDBC数据写入库之间，可以通过ShardingSphere的管理中心来保持规则同步。

这样，整个数据迁移过程中，旧的业务数据通道几乎不需要做任何改变。等数据迁移完成后，只要把ShardingJDBC数据写入库保留下来，把ShardingJDBC数据双写库和旧数据通道直接删掉即可。在整个迁移过程中，对应用来说，都只是访问一个DataSource，只不过是DataSource的具体实例根据配置做了变动而已，业务层面几乎不需要做任何修改。

当然，具体实施时，还是有很多具体的细节需要你自己去补充的。例如对于业务SQL需要做梳理。如果原本只是访问一个MySQL服务，而SQL语句写得比较放飞自我的话，那就需要按照ShardingJDBC的要求进行调整，尽量不要使用ShardingSphere不支持的SQL语句。

六、章节总结

到这里，ShardingSphere的主要功能模块我们就介绍得差不多了。但是，ShardingSphere是一个生态，所以，还有一些平常开发过程中用得不是太多的功能，比如像影子库，这里就不多做介绍了。有兴趣希望你按照我们这个路线，自己进行尝试。虽然现在新的版本还不太稳定，有很多隐藏的坑，但是，通过你自己的尝试，才能真正加深对于分库分表场景的理解。

另外，还是像之前一直强调的，ShardingSphere的重点是扩展。所以，理解ShardingSphere的这些功能扩展点，并在真实开发过程中进行自定义扩展，这应该是比学习ShardingSphere的API更重要的事情。

> 有道云笔记链接：https://note.youdao.com/s/ZTv0BY60