Apache ShardingSphere document

Apache ShardingSphere

2022 年 03 月 29 日

Contents

1 简介 2
- 1.1 ShardingSphere-JDBC . . . 2
- 1.2 ShardingSphere-Proxy . . . 2
- 1.3 ShardingSphere-Sidecar（TODO） . . . 4
- 1.4 混合架构 . . . 5

2 解决方案 6

3 线路规划 7

4 快速入门 8
- 4.1 ShardingSphere-JDBC . . . 8
  - 4.1.1 引入 maven 依赖 . . . 8
  - 4.1.2 规则配置 . . . 8
  - 4.1.3 创建数据源 . . . 8
- 4.2 ShardingSphere-Proxy . . . 9
  - 4.2.1 规则配置 . . . 9
  - 4.2.2 引入依赖 . . . 9
  - 4.2.3 启动服务 . . . 9
  - 4.2.4 使用 ShardingSphere-Proxy . . . 9
- 4.3 ShardingSphere-Scaling (Experimental) . . . 9
  - 4.3.1 规则配置 . . . 9
  - 4.3.2 引入依赖 . . . 10
  - 4.3.3 启动服务 . . . 10
  - 4.3.4 任务管理 . . . 10
  - 4.3.5 相关文档 . . . 10

5 概念 11
- 5.1 接入端 . . . 11
  - 5.1.1 ShardingSphere-JDBC . . . 11
  - 5.1.2 ShardingSphere-Proxy . . . 12
  - 5.1.3 混合架构 . . . 13

5.2 运行模式 15
5.2.1 背景 15
5.2.2 内存模式 15
5.2.3 单机模式 15
5.2.4 集群模式 15
5.3 DistSQL 15
5.3.1 背景 15
5.3.2 挑战 15
5.3.3 目标 16
5.3.4 注意事项 16
5.4 可插拔架构 16
5.4.1 背景 16
5.4.2 挑战 16
5.4.3 目标 16
5.4.4 实现 17
L1 内核层 17
L2 功能层 18
L3 生态层 18

6 功能 **19**
6.1 数据库兼容 19
6.1.1 背景 19
6.1.2 挑战 19
6.1.3 目标 20
6.1.4 SQL 解析 20
MySQL 20
openGauss 20
PostgreSQL 21
SQLServer 22
Oracle 22
SQL92 22
6.1.5 数据库协议 22
6.1.6 特性支持 22
MySQL 22
PostgreSQL 23
SQLServer 23
Oracle 24
SQL92 24
6.2 管控 24
6.2.1 背景 24
6.2.2 挑战 24
6.2.3 目标 24
6.2.4 核心概念 24
熔断 24
限流 25

6.3 数据分片 25
6.3.1 背景 25
垂直分片 25
水平分片 27
6.3.2 挑战 27
6.3.3 目标 28
6.3.4 核心概念 28
导览 28
表 28
数据节点 29
分片 30
行表达式 32
分布式主键 35
强制分片路由 37
6.3.5 使用规范 37
背景 37
SQL 37
分页 41
6.4 分布式事务 43
6.4.1 背景 43
本地事务 44
两阶段提交 44
柔性事务 44
6.4.2 挑战 45
6.4.3 目标 45
6.4.4 核心概念 45
导览 45
XA 事务 45
柔性事务 46
6.4.5 使用规范 46
背景 46
本地事务 47
XA 事务 47
柔性事务 51
6.5 读写分离 51
6.5.1 背景 51
6.5.2 挑战 51
6.5.3 目标 54
6.5.4 核心概念 54
主库 54
从库 54
主从同步 54
负载均衡策略 54
6.5.5 使用规范 54
支持项 54

不支持项 54
6.6 高可用 55
6.6.1 背景 55
6.6.2 挑战 55
6.6.3 目标 55
6.6.4 核心概念 56
高可用类型 56
动态读写分离 56
6.6.5 使用规范 56
支持项 56
不支持项 56
6.7 弹性伸缩 56
6.7.1 背景 56
6.7.2 挑战 56
6.7.3 目标 58
6.7.4 状态 58
6.7.5 核心概念 58
弹性伸缩作业 58
存量数据 58
增量数据 58
6.7.6 使用规范 58
支持项 58
不支持项 58
6.8 数据加密 59
6.8.1 背景 59
6.8.2 挑战 59
6.8.3 目标 59
6.8.4 核心概念 59
逻辑列 59
密文列 60
查询辅助列 60
明文列 60
6.8.5 使用规范 60
支持项 60
不支持项 60
6.9 影子库压测 60
6.9.1 背景 60
6.9.2 挑战 61
6.9.3 目标 61
6.9.4 核心概念 61
生产库 61
影子库 61
影子算法 61
6.9.5 使用规范 61
支持项 61

不支持项 62
6.10 可观察性 63
6.10.1 背景 63
6.10.2 挑战 63
6.10.3 目标 64
6.10.4 核心概念 64
代理 64
APM 64
Tracing 64
Metrics 64
6.10.5 使用规范 64
源码编译 64
agent 配置 65
ShardingSphere-Proxy 中使用 67

7 用户手册 69
7.1 ShardingSphere-JDBC 69
7.1.1 Java API 69
简介 69
使用步骤 70
模式配置 71
数据源配置 72
规则配置 72
7.1.2 YAML 配置 83
简介 83
使用步骤 83
语法说明 84
模式配置 84
数据源配置 85
规则配置 86
7.1.3 Spring Boot Starter 92
简介 92
使用步骤 92
在 Spring 中使用 ShardingSphere 数据源 92
模式配置 92
数据源配置 93
规则配置 95
7.1.4 Spring 命名空间 102
简介 102
使用步骤 102
配置 Spring Bean 102
在 Spring 中使用 ShardingSphere 数据源 103
模式配置 103
数据源配置 107
规则配置 108

7.1.5 属性配置 . . . 118
配置项说明 . . . 118
7.1.6 内置算法 . . . 118
简介 . . . 118
使用方式 . . . 118
元数据持久化仓库 . . . 118
分片算法 . . . 119
分布式序列算法 . . . 122
负载均衡算法 . . . 122
加密算法 . . . 123
影子算法 . . . 124
7.1.7 特殊 API . . . 125
数据分片 . . . 125
分布式事务 . . . 130
可观察性 . . . 138
7.1.8 不支持项 . . . 143
DataSource 接口 . . . 143
Connection 接口 . . . 143
Statement 和 PreparedStatement 接口 . . . 143
ResultSet 接口 . . . 143
JDBC 4.1 . . . 144
7.2 ShardingSphere-Proxy . . . 144
7.2.1 启动手册 . . . 144
使用二进制发布包 . . . 144
使用 Docker . . . 146
7.2.2 YAML 配置 . . . 147
权限 . . . 148
属性配置 . . . 149
7.2.3 DistSQL . . . 149
语法 . . . 149
使用 . . . 189
7.3 ShardingSphere-Sidecar . . . 196
7.3.1 简介 . . . 196
7.3.2 对比 . . . 197
7.4 ShardingSphere-Scaling . . . 197
7.4.1 简介 . . . 197
7.4.2 运行部署 . . . 197
部署启动 . . . 197
结束 . . . 200
7.4.3 使用手册 . . . 200
使用手册 . . . 200

8 开发者手册 208
8.1 运行模式 . . . 208
8.1.1 StandalonePersistRepository . . . 208

8.1.2 ClusterPersistRepository . . . 208
8.1.3 GovernanceWatcher . . . 209
8.2 配置 . . . 209
8.2.1 RuleBuilder . . . 209
8.2.2 YamlRuleConfigurationSwapper . . . 210
8.2.3 ShardingSphereYamlConstruct . . . 211
8.3 内核 . . . 211
8.3.1 SQLRouter . . . 211
8.3.2 SQLRewriteContextDecorator . . . 211
8.3.3 SQLExecutionHook . . . 212
8.3.4 ResultProcessEngine . . . 212
8.3.5 StoragePrivilegeHandler . . . 212
8.4 数据源 . . . 212
8.4.1 DatabaseType . . . 212
8.4.2 DialectTableMetaDataLoader . . . 213
8.4.3 DataSourcePoolMetaData . . . 213
8.4.4 DataSourcePoolActiveDetector . . . 214
8.5 SQL 解析 . . . 214
8.5.1 DatabaseTypedSQLParserFacade . . . 214
8.5.2 SQLVisitorFacade . . . 214
8.6 代理端 . . . 215
8.6.1 DatabaseProtocolFrontendEngine . . . 215
8.6.2 JDBCDriverURLRecognizer . . . 215
8.6.3 AuthorityProvideAlgorithm . . . 215
8.7 数据分片 . . . 216
8.7.1 ShardingAlgorithm . . . 216
8.7.2 KeyGenerateAlgorithm . . . 216
8.7.3 DatetimeService . . . 217
8.7.4 DatabaseSQLEntry . . . 217
8.8 读写分离 . . . 217
8.8.1 ReadwriteSplittingType . . . 217
8.8.2 ReplicaLoadBalanceAlgorithm . . . 218
8.9 高可用 . . . 218
8.9.1 DatabaseDiscoveryType . . . 218
8.10 分布式事务 . . . 218
8.10.1 ShardingSphereTransactionManager . . . 218
8.10.2 XATransactionManagerProvider . . . 219
8.10.3 XADataSourceDefinition . . . 219
8.10.4 DataSourcePropertyProvider . . . 219
8.11 弹性伸缩 . . . 220
8.11.1 ScalingEntry . . . 220
8.11.2 JobCompletionDetectAlgorithm . . . 220
8.11.3 DataConsistencyCheckAlgorithm . . . 220
8.11.4 SingleTableDataCalculator . . . 221
8.12 SQL 检查 . . . 221

8.12.1 SQLChecker . . . 221
8.13 数据加密 . . . 221
8.13.1 EncryptAlgorithm . . . 221
8.13.2 QueryAssistedEncryptAlgorithm . . . 222
8.14 影子库 . . . 222
8.14.1 ShadowAlgorithm . . . 222
8.15 可观察性 . . . 222
8.15.1 PluginDefinitionService . . . 222
8.15.2 PluginBootService . . . 223

9 技术参考 224
9.1 管控 . . . 224
9.1.1 注册中心数据结构 . . . 224
/rules . . . 225
/props . . . 225
/metadata/${schemaName}/dataSources . . . 226
/metadata/${schemaName}/rules . . . 226
/metadata/${schemaName}/tables . . . 227
/nodes/compute_nodes . . . 227
/nodes/storage_nodes . . . 227
9.2 数据分片 . . . 227
9.2.1 SQL 解析 . . . 228
9.2.2 SQL 路由 . . . 228
9.2.3 SQL 改写 . . . 229
9.2.4 SQL 执行 . . . 229
9.2.5 结果归并 . . . 229
9.2.6 查询优化 . . . 229
9.2.7 解析引擎 . . . 230
抽象语法树 . . . 230
SQL 解析引擎 . . . 231
9.2.8 路由引擎 . . . 235
分片路由 . . . 235
广播路由 . . . 237
9.2.9 改写引擎 . . . 239
正确性改写 . . . 239
优化改写 . . . 244
9.2.10 执行引擎 . . . 244
连接模式 . . . 244
自动化执行引擎 . . . 246
9.2.11 归并引擎 . . . 250
遍历归并 . . . 250
排序归并 . . . 250
分组归并 . . . 251
聚合归并 . . . 254
分页归并 . . . 254

9.3 分布式事务 254
9.3.1 导览 254
9.3.2 XA 事务 255
开启全局事务 255
执行真实分片 SQL 255
提交或回滚事务 256
9.3.3 Seata 柔性事务 257
引擎初始化 257
开启全局事务 258
执行真实分片 SQL 258
提交或回滚事务 258
9.4 弹性伸缩 258
9.4.1 原理说明 258
9.4.2 执行阶段说明 260
准备阶段 260
存量数据迁移阶段 260
增量数据同步阶段 260
规则切换阶段 260
9.5 数据加密 260
9.5.1 处理流程详解 260
整体架构 261
加密规则 261
加密处理过程 262
9.5.2 解决方案详解 264
新上线业务 264
已上线业务改造 265
9.5.3 中间件加密服务优势 268
9.5.4 加密算法解析 269
EncryptAlgorithm 269
QueryAssistedEncryptAlgorithm 269
9.6 影子库 270
9.6.1 整体架构 270
9.6.2 影子规则 271
9.6.3 路由过程 272
9.6.4 影子判定流程 272
DML 语句 272
DDL 语句 272
9.6.5 影子算法 272
9.6.6 使用案例 272
场景需求 272
影子库配置 273
影子库环境 273
影子算法使用 274
9.7 测试 277
9.7.1 整合测试 277

9.7.2 模块测试 277
9.7.3 性能测试 277
9.7.4 集成测试 277
设计 277
使用指南 278
9.7.5 性能测试 281
Sysbench 性能测试 281
BenchmarkSQL 性能测试 290
9.7.6 模块测试 304
SQL 解析测试 304
SQL 改写测试 305
9.8 FAQ 307
9.8.1 [JDBC] 为什么配置了某个数据连接池的 spring-boot-starter（比如 druid）和 shardingsphere-jdbc-spring-boot-starter 时，系统启动会报错? 307
9.8.2 [JDBC] 使用 Spring 命名空间时找不到 xsd? 307
9.8.3 [JDBC] 引入 `shardingsphere-transaction-xa-core` 后，如何避免 spring-boot 自动加载默认的 JtaTransactionManager? 307
9.8.4 [Proxy] Windows 环境下，运行 ShardingSphere-Proxy，找不到或无法加载主类 org.apache.shardingsphere.proxy.Bootstrap，如何解决? 307
9.8.5 [Proxy] 在使用 ShardingSphere-Proxy 的时候，如何动态在添加新的 logic schema? 308
9.8.6 [Proxy] 在使用 ShardingSphere-Proxy 时，怎么使用合适的工具连接到 ShardingSphere-Proxy? 308
9.8.7 [Proxy] 使用 Navicat 等第三方数据库工具连接 ShardingSphere-Proxy 时，如果 ShardingSphere-Proxy 没有创建 Schema 或者没有添加 Resource，连接失败? . 308
9.8.8 [分片] Cloud not resolve placeholder …in string value …异常的解决方法? 309
9.8.9 [分片] inline 表达式返回结果为何出现浮点数? 309
9.8.10 [分片] 如果只有部分数据库分库分表，是否需要将不分库分表的表也配置在分片规则中? 309
9.8.11 [分片] 指定了泛型为 Long 的 `SingleKeyTableShardingAlgorithm`，遇到 `ClassCastException: Integer can not cast to Long`? 309
9.8.12 [分片、PROXY] 实现 `StandardShardingAlgorithm` 自定义算法时，指定了 `Comparable` 的具体类型为 Long，且数据库表中字段类型为 bigint，出现 `ClassCastException: Integer can not cast to Long` 异常。 309
9.8.13 [分片] ShardingSphere 提供的默认分布式自增主键策略为什么是不连续的，且尾数大多为偶数? 310
9.8.14 [分片] 如何在 inline 分表策略时，允许执行范围查询操作（BETWEEN AND、>、<、>=、<=)? 310
9.8.15 [分片] 为什么我实现了 `KeyGenerateAlgorithm` 接口，也配置了 Type，但是自定义的分布式主键依然不生效? 310
9.8.16 [分片] ShardingSphere 除了支持自带的分布式自增主键之外，还能否支持原生的自增主键? 310
9.8.17 [数据加密] JPA 和数据加密无法一起使用，如何解决? 311
9.8.18 [DistSQL] 使用 DistSQL 添加数据源时，如何设置自定义的 JDBC 连接参数或连接池属性? 311

9.8.19 [DistSQL] 使用 DistSQL 删除资源时,出现 Resource [xxx] is still used by [SingleTableRule]。 311
9.8.20 [DistSQL] 使用 DistSQL 添加资源时，出现 Failed to get driver instance for jdbcURL=xxx。 311
9.8.21 [其他] 如果 SQL 在 ShardingSphere 中执行不正确，该如何调试? 312
9.8.22 [其他] 阅读源码时为什么会出现编译错误? IDEA 不索引生成的代码? 312
9.8.23 [其他] 使用 SQLSever 和 PostgreSQL 时，聚合列不加别名会抛异常? 312
9.8.24 [其他] Oracle 数据库使用 Timestamp 类型的 Order By 语句抛出异常提示“Order by value must implements Comparable”? 313
9.8.25 [其他] Windows 环境下，通过 Git 克隆 ShardingSphere 源码时为什么提示文件名过长，如何解决? 314
9.8.26 [其他] Type is required 异常的解决方法? 314
9.8.27 [其他] 服务启动时如何加快 metadata 加载速度? 315
9.8.28 [其他] ANTLR 插件在 src 同级目录下生成代码，容易误提交，如何避免? 315
9.8.29 [其他] 使用 Proxool 时分库结果不正确? 316
9.8.30 [其他] 使用 Spring Boot 2.x 集成 ShardingSphere 时，配置文件中的属性设置不生效? 316
9.9 API 变更历史 318
9.9.1 ShardingSphere-JDBC 318
YAML 配置 318
Java API 333
Spring 命名空间配置 355
Spring Boot Start 配置 378
9.9.2 ShardingSphere-Proxy 390
5.0.0-beta 390
5.0.0-alpha 392
ShardingSphere-4.x 393
ShardingSphere-3.x 395

10 下载 **398**
10.1 最新版本 398
10.1.1 Apache ShardingSphere - 版本: 5.1.0 (发布日期: Feb 16th, 2022) 398
10.2 全部版本 398
10.3 校验版本 398

星评增长时间线

贡献者增长时间线

Apache ShardingSphere 产品定位为 `Database Plus`，旨在构建异构数据库上层的标准和生态。它关注如何充分合理地利用数据库的计算和存储能力，而并非实现一个全新的数据库。ShardingSphere 站在数据库的上层视角，关注他们之间的协作多于数据库自身。

连接、增量和 可插拔是 Apache ShardingSphere 的核心概念。

- 连接：通过对数据库协议、SQL 方言以及数据库存储的灵活适配，快速的连接应用与多模式的异构数据库；
- 增量：获取数据库的访问流量，并提供流量重定向（数据分片、读写分离、影子库）、流量变形（数据加密、数据脱敏）、流量鉴权（安全、审计、权限）、流量治理（熔断、限流）以及流量分析（服务质量分析、可观察性）等透明化增量功能；
- 可插拔：项目采用微内核 + 三层可插拔模型，使内核、功能组件以及生态对接完全能够灵活的方式进行插拔式扩展，开发者能够像使用积木一样定制属于自己的独特系统。

ShardingSphere 已于 2020 年 4 月 16 日成为 Apache 软件基金会的顶级项目。欢迎通过邮件列表参与讨论。

图 1: Overview

1 简介

Apache ShardingSphere 由 JDBC、Proxy 和 Sidecar（规划中）这 3 款既能够独立部署，又支持混合部署配合使用的产品组成。它们均提供标准化的基于数据库作为存储节点的增量功能，可适用于如 Java 同构、异构语言、云原生等各种多样化的应用场景。

关系型数据库当今依然占有巨大市场份额，是企业核心系统的基石，未来也难于撼动，我们更加注重在原有基础上提供增量，而非颠覆。

1.1 ShardingSphere-JDBC

定位为轻量级 Java 框架，在 Java 的 JDBC 层提供的额外服务。它使用客户端直连数据库，以 jar 包形式提供服务，无需额外部署和依赖，可理解为增强版的 JDBC 驱动，完全兼容 JDBC 和各种 ORM 框架。

- 适用于任何基于 JDBC 的 ORM 框架，如：JPA, Hibernate, Mybatis, Spring JDBC Template 或直接使用 JDBC；
- 支持任何第三方的数据库连接池，如：DBCP, C3P0, BoneCP, HikariCP 等；
- 支持任意实现 JDBC 规范的数据库，目前支持 MySQL，PostgreSQL，Oracle，SQLServer 以及任何可使用 JDBC 访问的数据库。

1.2 ShardingSphere-Proxy

定位为透明化的数据库代理端，提供封装了数据库二进制协议的服务端版本，用于完成对异构语言的支持。目前提供 MySQL 和 PostgreSQL（兼容 openGauss 等基于 PostgreSQL 的数据库）版本，它可以使用任何兼容 MySQL/PostgreSQL 协议的访问客户端（如：MySQL Command Client, MySQL Workbench, Navicat 等）操作数据，对 DBA 更加友好。

- 向应用程序完全透明，可直接当做 MySQL/PostgreSQL 使用；
- 适用于任何兼容 MySQL/PostgreSQL 协议的的客户端。

图 1: ShardingSphere-JDBC Architecture

图 2: ShardingSphere-Proxy Architecture

1.3 ShardingSphere-Sidecar（TODO）

定位为 Kubernetes 的云原生数据库代理，以 Sidecar 的形式代理所有对数据库的访问。通过无中心、零侵入的方案提供与数据库交互的啮合层，即 `Database Mesh`，又可称数据库网格。

Database Mesh 的关注重点在于如何将分布式的数据访问应用与数据库有机串联起来，它更加关注的是交互，是将杂乱无章的应用与数据库之间的交互进行有效地梳理。使用 Database Mesh，访问数据库的应用和数据库终将形成一个巨大的网格体系，应用和数据库只需在网格体系中对号入座即可，它们都是被啮合层所治理的对象。

图 3: ShardingSphere-Sidecar Architecture

	Shard ingSphere-JDBC	*Shardi ngSphere-Proxy*	*Sharding Sphere-Sidecar*
数据库	任意	M ySQL/PostgreSQL	M ySQL/PostgreSQL
连接消耗数	高	低	高
异构语言	仅 Java	任意	任意
性能	损耗低	损耗略高	损耗低
无中心化	是	否	是
静态入口	无	有	无

1.4 混合架构

ShardingSphere-JDBC 采用无中心化架构，与应用程序共享资源，适用于 Java 开发的高性能的轻量级 OLTP 应用；ShardingSphere-Proxy 提供静态入口以及异构语言的支持，独立于应用程序部署，适用于 OLAP 应用以及对分片数据库进行管理和运维的场景。

Apache ShardingSphere 是多接入端共同组成的生态圈。通过混合使用 ShardingSphere-JDBC 和 ShardingSphere-Proxy，并采用同一注册中心统一配置分片策略，能够灵活的搭建适用于各种场景的应用系统，使得架构师更加自由地调整适合于当前业务的最佳系统架构。

图 4: ShardingSphere Hybrid Architecture

2 解决方案

解决方案/功能	分布式数据库	数据安全	• 数据库网关 *	• 全链路压测 *
	数据分片	数据加密	异构数据库支持	影子库
	读写分离	行级权限（TODO）	SQL 方言转换（TODO）	可观测性
	分布式事务	SQL 审计（TODO）		
	弹性伸缩	SQL 防火墙（TODO）		
	高可用			

3 线路规划

图 1: Roadmap

4 快速入门

本章节以尽量短的时间，为使用者提供最简单的 Apache ShardingSphere 的快速入门。

4.1 ShardingSphere-JDBC

4.1.1 引入 maven 依赖

```
<dependency>
    <groupId>org.apache.shardingsphere</groupId>
    <artifactId>shardingsphere-jdbc-core</artifactId>
    <version>${latest.release.version}</version>
</dependency>
```

注意：请将 `${latest.release.version}` 更改为实际的版本号。

4.1.2 规则配置

ShardingSphere-JDBC 可以通过 `Java`，`YAML`，`Spring` 命名空间和 `Spring Boot Starter` 这 4 种方式进行配置，开发者可根据场景选择适合的配置方式。详情请参见用户手册。

4.1.3 创建数据源

通过 `ShardingSphereDataSourceFactory` 工厂和规则配置对象获取 `ShardingSphereDataSource`。该对象实现自 JDBC 的标准 DataSource 接口，可用于原生 JDBC 开发，或使用 JPA, Hibernate, MyBatis 等 ORM 类库。

```
DataSource dataSource = ShardingSphereDataSourceFactory.
createDataSource(schemaName, modeConfig, dataSourceMap, ruleConfigs, props);
```

4.2 ShardingSphere-Proxy

4.2.1 规则配置

编辑 `%SHARDINGSPHERE_PROXY_HOME%/conf/config-xxx.yaml`。

编辑 `%SHARDINGSPHERE_PROXY_HOME%/conf/server.yaml`。

> %SHARDINGSPHERE_PROXY_HOME% 为 Proxy 解压后的路径，例：/opt/shardingsphere-proxy-bin/

详情请参见 配置手册。

4.2.2 引入依赖

如果后端连接 PostgreSQL 数据库，不需要引入额外依赖。

如果后端连接 MySQL 数据库，请下载 mysql-connector-java-5.1.47.jar 或者 mysql-connector-java-8.0.11.jar，并将其放入 `%SHARDINGSPHERE_PROXY_HOME%/ext-lib` 目录。

4.2.3 启动服务

- 使用默认配置项

```
sh %SHARDINGSPHERE_PROXY_HOME%/bin/start.sh
```

默认启动端口为 `3307`，默认配置文件目录为：`%SHARDINGSPHERE_PROXY_HOME%/conf/`。

- 自定义端口和配置文件目录

```
sh %SHARDINGSPHERE_PROXY_HOME%/bin/start.sh ${proxy_port} ${proxy_conf_directory}
```

4.2.4 使用 ShardingSphere-Proxy

执行 MySQL 或 PostgreSQL 的客户端命令直接操作 ShardingSphere-Proxy 即可。以 MySQL 举例：

```
mysql -u${proxy_username} -p${proxy_password} -h${proxy_host} -P${proxy_port}
```

4.3 ShardingSphere-Scaling (Experimental)

4.3.1 规则配置

编辑 `%SHARDINGSPHERE_PROXY_HOME%/conf/server.yaml`。

> %SHARDINGSPHERE_PROXY_HOME% 为 Proxy 解压后的路径，例：/opt/shardingsphere-proxy-bin/

详情请参见运行部署。

4.3.2 引入依赖

如果后端连接 PostgreSQL 数据库，不需要引入额外依赖。

如果后端连接 MySQL 数据库，请下载 mysql-connector-java-5.1.47.jar，并将其放入 `%SHARDINGSPHERE_PROXY_HOME%/lib` 目录。

4.3.3 启动服务

```
sh %SHARDINGSPHERE_PROXY_HOME%/bin/start.sh
```

4.3.4 任务管理

通过相应的 DistSQL 接口管理迁移任务。

详情请参见使用手册。

4.3.5 相关文档

- 功能 # 弹性伸缩：核心概念、使用规范
- 用户手册 # 弹性伸缩：运行部署、使用手册
- RAL# 弹性伸缩：弹性伸缩的 DistSQL
- 开发者手册 # 弹性伸缩：SPI 接口及实现类

5 概念

Apache ShardingSphere 功能十分复杂，有数百模块之多，但众多模块间的概念却简单明了。大部分模块都是面向这几个概念的横向扩展。

它的概念主要包括：面向独立产品的接入端、面向启动的运行模式、面向使用者操作的 DistSQL 以及面向开发者的可插拔架构。

本章节将详细阐述 Apache ShardingSphere 相关的概念。

5.1 接入端

Apache ShardingSphere 由 ShardingSphere-JDBC 和 ShardingSphere-Proxy 这 2 款既能够独立部署，又支持混合部署配合使用的产品组成。它们均提供标准化的基于数据库作为存储节点的增量功能，可适用于如 Java 同构、异构语言、云原生等各种多样化的应用场景。

5.1.1 ShardingSphere-JDBC

ShardingSphere-JDBC 是 Apache ShardingSphere 的第一个产品，也是 Apache ShardingSphere 的前身。定位为轻量级 Java 框架，在 Java 的 JDBC 层提供的额外服务。它使用客户端直连数据库，以 jar 包形式提供服务，无需额外部署和依赖，可理解为增强版的 JDBC 驱动，完全兼容 JDBC 和各种 ORM 框架。

- 适用于任何基于 JDBC 的 ORM 框架，如：JPA, Hibernate, Mybatis, Spring JDBC Template 或直接使用 JDBC；
- 支持任何第三方的数据库连接池，如：DBCP, C3P0, BoneCP, HikariCP 等；
- 支持任意实现 JDBC 规范的数据库，目前支持 MySQL，PostgreSQL，Oracle，SQLServer 以及任何可使用 JDBC 访问的数据库。

图 1: ShardingSphere-JDBC Architecture

	ShardingSphere-JDBC	*ShardingSphere-Proxy*
数据库	任意	MySQL/PostgreSQL
连接消耗数	高	低
异构语言	仅 `Java`	任意
性能	损耗低	损耗略高
无中心化	是	否
静态入口	无	有

ShardingSphere-JDBC 的优势在于对 Java 应用的友好度。

5.1.2 ShardingSphere-Proxy

ShardingSphere-Proxy 是 Apache ShardingSphere 的第二个产品。它定位为透明化的数据库代理端，提供封装了数据库二进制协议的服务端版本，用于完成对异构语言的支持。目前提供 MySQL 和 PostgreSQL（兼容 openGauss 等基于 PostgreSQL 的数据库）版本，它可以使用任何兼容 MySQL/PostgreSQL 协议的访问客户端（如：MySQL Command Client, MySQL Workbench, Navicat 等）操作数据，对 DBA 更加友好。

- 向应用程序完全透明，可直接当做 MySQL/PostgreSQL 使用；
- 适用于任何兼容 MySQL/PostgreSQL 协议的的客户端。

图 2: ShardingSphere-Proxy Architecture

	ShardingSphere-JDBC	*ShardingSphere-Proxy*
数据库	任意	`MySQL/PostgreSQL`
连接消耗数	高	低
异构语言	仅 Java	任意
性能	损耗低	损耗略高
无中心化	是	否
静态入口	无	有

ShardingSphere-Proxy 的优势在于对异构语言的支持，以及为 DBA 提供可操作入口。

5.1.3 混合架构

ShardingSphere-JDBC 采用无中心化架构，与应用程序共享资源，适用于 Java 开发的高性能的轻量级 OLTP 应用；ShardingSphere-Proxy 提供静态入口以及异构语言的支持，独立于应用程序部署，适用于 OLAP 应用以及对分片数据库进行管理和运维的场景。

Apache ShardingSphere 是多接入端共同组成的生态圈。通过混合使用 ShardingSphere-JDBC 和 ShardingSphere-Proxy，并采用同一注册中心统一配置分片策略，能够灵活的搭建适用于各种场景的应用系统，使得架构师更加自由地调整适合于当前业务的最佳系统架构。

图 3: Hybrid Architecture

5.2 运行模式

5.2.1 背景

Apache ShardingSphere 是一套完善的产品，使用场景非常广泛。除生产环境的集群部署之外，还为工程师在开发和自动化测试等场景提供相应的运行模式。Apache ShardingSphere 提供的 3 种运行模式分别是内存模式、单机模式和集群模式。

5.2.2 内存模式

初始化配置或执行 SQL 等造成的元数据结果变更的操作，仅在当前进程中生效。适用于集成测试的环境启动，方便开发人员在整合功能测试中集成 Apache ShardingSphere 而无需清理运行痕迹。

5.2.3 单机模式

能够将数据源和规则等元数据信息持久化，但无法将元数据同步至多个 Apache ShardingSphere 实例，无法在集群环境中相互感知。通过某一实例更新元数据之后，会导致其他实例由于获取不到最新的元数据而产生不一致的错误。适用于工程师在本地搭建 Apache ShardingSphere 环境。

5.2.4 集群模式

提供了多个 Apache ShardingSphere 实例之间的元数据共享和分布式场景下状态协调的能力。在真实部署上线的生产环境，必须使用集群模式。它能够提供计算能力水平扩展和高可用等分布式系统必备的能力。集群环境需要通过独立部署的注册中心来存储元数据和协调节点状态。

5.3 DistSQL

5.3.1 背景

DistSQL（Distributed SQL）是 Apache ShardingSphere 特有的操作语言。它与标准 SQL 的使用方式完全一致，用于提供增量功能的 SQL 级别操作能力。

5.3.2 挑战

灵活的规则配置和资源管控能力是 Apache ShardingSphere 的特点之一。在使用 4.x 及其之前版本时，开发者虽然可以像使用原生数据库一样操作数据，但却需要通过本地文件或注册中心配置资源和规则。然而，操作习惯变更，对于运维工程师并不友好。

DistSQL 让用户可以像操作数据库一样操作 Apache ShardingSphere，使其从面向开发人员的框架和中间件转变为面向运维人员的数据库产品。

DistSQL 细分为 RDL、RQL 和 RAL 三种类型。

- RDL（Resource & Rule Definition Language）负责资源和规则的创建、修改和删除；
- RQL（Resource & Rule Query Language）负责资源和规则的查询和展现；
- RAL（Resource & Rule Administration Language）负责 Hint、事务类型切换、分片执行计划查询等管理功能。

5.3.3 目标

打破中间件和数据库之间的界限，让开发者像使用数据库一样使用 Apache ShardingSphere，是 DistSQL 的设计目标。

5.3.4 注意事项

DistSQL 只能用于 ShardingSphere-Proxy，ShardingSphere-JDBC 暂不提供。

5.4 可插拔架构

5.4.1 背景

在 Apache ShardingSphere 中，很多功能实现类的加载方式是通过 SPI（Service Provider Interface）注入的方式完成的。SPI 是一种为了被第三方实现或扩展的 API，它可以用于实现框架扩展或组件替换。

5.4.2 挑战

可插拔架构对程序架构设计的要求非常高，需要将各个模块相互独立，互不感知，并且通过一个可插拔内核，以叠加的方式将各种功能组合使用。设计一套将功能开发完全隔离的架构体系，既可以最大限度的将开源社区的活力激发出来，也能够保障项目的质量。

Apache ShardingSphere 5.x 版本开始致力于可插拔架构，项目的功能组件能够灵活的以可插拔的方式进行扩展。目前，数据分片、读写分离、数据库高可用、数据加密、影子库压测等功能，以及对 MySQL、PostgreSQL、SQLServer、Oracle 等 SQL 与协议的支持，均通过插件的方式织入项目。Apache ShardingSphere 目前已提供数十个 SPI 作为系统的扩展点，而且仍在不断增加中。

5.4.3 目标

让开发者能够像使用积木一样定制属于自己的独特系统，是 Apache ShardingSphere 可插拔架构的设计目标。

图 4: Pluggable Platform

5.4.4 实现

Apache ShardingSphere 的可插拔架构划分为 3 层，它们是：L1 内核层、L2 功能层、L3 生态层。

L1 内核层

是数据库基本能力的抽象，其所有组件均必须存在，但具体实现方式可通过可插拔的方式更换。主要包括查询优化器、分布式事务引擎、分布式执行引擎、权限引擎和调度引擎等。

L2 功能层

用于提供增量能力，其所有组件均是可选的，可以包含零至多个组件。组件之间完全隔离，互无感知，多组件可通过叠加的方式相互配合使用。主要包括数据分片、读写分离、数据库高可用、数据加密、影子库等。用户自定义功能可完全面向 Apache ShardingSphere 定义的顶层接口进行定制化扩展，而无需改动内核代码。

L3 生态层

用于对接和融入现有数据库生态，包括数据库协议、SQL 解析器和存储适配器，分别对应于 Apache ShardingSphere 以数据库协议提供服务的方式、SQL 方言操作数据的方式以及对接存储节点的数据库类型。

6 功能

Apache ShardingSphere 提供了多样化的功能，涵盖范围从数据库内核、数据库分布式到贴近数据库上层的应用，为用户提供了大量的功能池。

功能并无边界，只要满足数据库服务和生态的共性需求即可，期待更多的开源工程师参与 Apache ShardingSphere 社区，提供新颖思路和令人兴奋的功能。

6.1 数据库兼容

6.1.1 背景

随着通信技术的革新，全新领域的应用层出不穷，推动和颠覆整个人类社会协作模式的革新。数据存量随着应用的探索不断增加，数据的存储和计算模式无时无刻面临着创新。

面向交易、大数据、关联分析、物联网等场景越来越细分，单一数据库再也无法适用于所有的应用场景。与此同时，场景内部也愈加细化，相似场景使用不同数据库已成为常态。由此可见，数据库碎片化的趋势已经不可逆转。

6.1.2 挑战

并无统一标准的数据库的访问协议和 SQL 方言，以及各种数据库带来的不同运维方法和监控工具的异同，让开发者的学习成本和 DBA 的运维成本不断增加。提升与原有数据库兼容度，是在其之上提供增量服务的前提。

SQL 方言和数据库协议的兼容，是数据库兼容度提升的关键点。

6.1.3 目标

尽量多的兼容各种数据库，让用户零使用成本，是 Apache ShardingSphere 数据库兼容度希望达成的主要目标。

6.1.4 SQL 解析

SQL 是使用者与数据库交流的标准语言。SQL 解析引擎负责将 SQL 字符串解析为抽象语法树，供 Apache ShardingSphere 理解并实现其增量功能。

目前支持 MySQL, PostgreSQL, SQLServer, Oracle, openGauss 以及符合 SQL92 规范的 SQL 方言。由于 SQL 语法的复杂性，目前仍然存在少量不支持的 SQL。

本章节详细罗列出目前不支持的 SQL 种类，供使用者参考。

其中有未涉及到的 SQL 欢迎补充，未支持的 SQL 也尽量会在未来的版本中支持。

MySQL

MySQL 不支持的 SQL 清单如下：

SQL
CLONE LOCAL DATA DIRECTORY = ‘clone_dir’
INSTALL COMPONENT ‘file://component1’, ‘file://component2’
UNINSTALL COMPONENT ‘file://component1’, ‘file://component2’
REPAIR TABLE t_order
OPTIMIZE TABLE t_order
CHECKSUM TABLE t_order
CHECK TABLE t_order
SET RESOURCE GROUP group_name
DROP RESOURCE GROUP group_name
CREATE RESOURCE GROUP group_name TYPE = SYSTEM
ALTER RESOURCE GROUP rg1 VCPU = 0-63

openGauss

openGauss 不支持的 SQL 清单如下：

SQL
CREATE type avg_state AS (total bigint, count bigint);
CREATE AGGREGATE my_avg(int4) (stype = avg_state, sfunc = avg_transfn, finalfunc = avg_finalfn)
CREATE TABLE agg_data_2k AS SELECT g FROM generate_series(0, 1999) g;
CREATE SCHEMA alt_nsp1;
ALTER AGGREGATE alt_agg3(int) OWNER TO regress_alter_generic_user2;
CREATE CONVERSION alt_conv1 FOR ‘LATIN1’ TO ‘UTF8’ FROM iso8859_1_to_utf8;
CREATE FOREIGN DATA WRAPPER alt_fdw1
CREATE SERVER alt_fserv1 FOREIGN DATA WRAPPER alt_fdw1
CREATE LANGUAGE alt_lang1 HANDLER plpgsql_call_handler
CREATE STATISTICS alt_stat1 ON a, b FROM alt_regress_1
CREATE TEXT SEARCH DICTIONARY alt_ts_dict1 (template=simple)
CREATE RULE def_view_test_ins AS ON INSERT TO def_view_test DO INSTEAD INSERT INTO def_test SELECT new.*
ALTER TABLE alterlock SET (toast.autovacuum_enabled = off)
CREATE PUBLICATION pub1 FOR TABLE alter1.t1, ALL TABLES IN SCHEMA alter2

PostgreSQL

PostgreSQL 不支持的 SQL 清单如下：

SQL
CREATE type avg_state AS (total bigint, count bigint);
CREATE AGGREGATE my_avg(int4) (stype = avg_state, sfunc = avg_transfn, finalfunc = avg_finalfn)
CREATE TABLE agg_data_2k AS SELECT g FROM generate_series(0, 1999) g;
CREATE SCHEMA alt_nsp1;
ALTER AGGREGATE alt_agg3(int) OWNER TO regress_alter_generic_user2;
CREATE CONVERSION alt_conv1 FOR ‘LATIN1’ TO ‘UTF8’ FROM iso8859_1_to_utf8;
CREATE FOREIGN DATA WRAPPER alt_fdw1
CREATE SERVER alt_fserv1 FOREIGN DATA WRAPPER alt_fdw1
CREATE LANGUAGE alt_lang1 HANDLER plpgsql_call_handler
CREATE STATISTICS alt_stat1 ON a, b FROM alt_regress_1
CREATE TEXT SEARCH DICTIONARY alt_ts_dict1 (template=simple)
CREATE RULE def_view_test_ins AS ON INSERT TO def_view_test DO INSTEAD INSERT INTO def_test SELECT new.*
ALTER TABLE alterlock SET (toast.autovacuum_enabled = off)
CREATE PUBLICATION pub1 FOR TABLE alter1.t1, ALL TABLES IN SCHEMA alter2

SQLServer

SQLServer 不支持的 SQL 清单如下:

TODO

Oracle

Oracle 不支持的 SQL 清单如下:

TODO

SQL92

SQL92 不支持的 SQL 清单如下:

TODO

6.1.5 数据库协议

Apache ShardingSphere 目前实现了 MySQL 和 PostgreSQL 协议。

6.1.6 特性支持

Apache ShardingSphere 为数据库提供了分布式协作的能力，同时将一部分数据库特性抽象到了上层，进行统一管理，以降低用户的使用难度。

因此，对于统一提供的特性，原生的 SQL 将不再下发到数据库，并提示该操作不被支持，用户可使用 ShardingSphere 提供的的方式进行代替。

本章节详细罗列出目前不支持的数据库特性和相关的 SQL 语句，供使用者参考。

其中有未涉及到的 SQL 欢迎补充。

MySQL

MySQL 不支持的 SQL 清单如下:

用户和角色

SQL
CREATE USER ‘finley’@‘localhost’IDENTIFIED BY ‘password’
ALTER USER ‘finley’@‘localhost’IDENTIFIED BY ‘new_password’
DROP USER ‘finley’@‘localhost’;
CREATE ROLE ‘app_read’
DROP ROLE ‘app_read’
SHOW CREATE USER finley
SET PASSWORD = ‘auth_string’
SET ROLE DEFAULT;

授权

SQL
GRANT ALL ON db1.* TO ‘jeffrey’@‘localhost’
GRANT SELECT ON world.* TO ‘role3’;
GRANT ‘role1’, ‘role2’TO ‘user1’@‘localhost’
REVOKE INSERT ON . FROM ‘jeffrey’@‘localhost’
REVOKE ‘role1’, ‘role2’FROM ‘user1’@‘localhost’
REVOKE ALL PRIVILEGES, GRANT OPTION FROM user_or_role
SHOW GRANTS FOR ‘jeffrey’@‘localhost’
SHOW GRANTS FOR CURRENT_USER
FLUSH PRIVILEGES

PostgreSQL

PostgreSQL 不支持的 SQL 清单如下:

TODO

SQLServer

SQLServer 不支持的 SQL 清单如下:

TODO

Oracle

Oracle 不支持的 SQL 清单如下：

TODO

SQL92

SQL92 不支持的 SQL 清单如下：

TODO

6.2 管控

6.2.1 背景

随着数据规模的不断膨胀，使用多节点集群的分布式方式逐渐成为趋势。对集群整体视角的统一管理能力，和针对单独组件细粒度的控制能力，是基于存算分离的现代数据库体系中不可或缺的功能。

6.2.2 挑战

管控的挑战，在于对集群的集中化管理的统一管理能力以及在单点出现故障时精细化的操作能力。

集中化管理的挑战体现在将包括数据库存储节点和中间件计算节点的状态统一管理，并且能够实时的探测到分布式环境下最新的变动情况，进一步为集群的控制和调度提供依据。

面对超负荷的流量下，针对某一节点进行熔断和限流，以保证整个数据库集群得以继续运行，是分布式系统下对单一节点控制能力的挑战。

6.2.3 目标

实现从数据库到计算节点打通的一体化管理能力，在故障中为组件提供细粒度的控制能力，并尽可能的提供自愈的可能，是 Apache ShardingSphere 管控模块的主要设计目标。

6.2.4 核心概念

熔断

阻断 Apache ShardingSphere 和数据库的连接。当某个 Apache ShardingSphere 节点超过负载后，停止该节点对数据库的访问，使数据库能够保证足够的资源为其他节点提供服务。

限流

面对超负荷的请求开启限流，以保护部分请求可以得以高质量的响应。

6.3 数据分片

6.3.1 背景

传统的将数据集中存储至单一节点的解决方案，在性能、可用性和运维成本这三方面已经难于满足海量数据的场景。

从性能方面来说，由于关系型数据库大多采用 B+ 树类型的索引，在数据量超过阈值的情况下，索引深度的增加也将使得磁盘访问的 IO 次数增加，进而导致查询性能的下降；同时，高并发访问请求也使得集中式数据库成为系统的最大瓶颈。

从可用性的方面来讲，服务化的无状态性，能够达到较小成本的随意扩容，这必然导致系统的最终压力都落在数据库之上。而单一的数据节点，或者简单的主从架构，已经越来越难以承担。数据库的可用性，已成为整个系统的关键。

从运维成本方面考虑，当一个数据库实例中的数据达到阈值以上，对于 DBA 的运维压力就会增大。数据备份和恢复的时间成本都将随着数据量的大小而愈发不可控。一般来讲，单一数据库实例的数据的阈值在 1TB 之内，是比较合理的范围。

在传统的关系型数据库无法满足互联网场景需要的情况下，将数据存储至原生支持分布式的 NoSQL 的尝试越来越多。但 NoSQL 对 SQL 的不兼容性以及生态圈的不完善，使得它们在与关系型数据库的博弈中始终无法完成致命一击，而关系型数据库的地位却依然不可撼动。

数据分片指按照某个维度将存放在单一数据库中的数据分散地存放至多个数据库或表中以达到提升性能瓶颈以及可用性的效果。数据分片的有效手段是对关系型数据库进行分库和分表。分库和分表均可以有效的避免由数据量超过可承受阈值而产生的查询瓶颈。除此之外，分库还能够用于有效的分散对数据库单点的访问量；分表虽然无法缓解数据库压力，但却能够提供尽量将分布式事务转化为本地事务的可能，一旦涉及到跨库的更新操作，分布式事务往往会使问题变得复杂。使用多主多从的分片方式，可以有效的避免数据单点，从而提升数据架构的可用性。

通过分库和分表进行数据的拆分来使得各个表的数据量保持在阈值以下，以及对流量进行疏导应对高访问量，是应对高并发和海量数据系统的有效手段。数据分片的拆分方式又分为垂直分片和水平分片。

垂直分片

按照业务拆分的方式称为垂直分片，又称为纵向拆分，它的核心理念是专库专用。在拆分之前，一个数据库由多个数据表构成，每个表对应着不同的业务。而拆分之后，则是按照业务将表进行归类，分布到不同的数据库中，从而将压力分散至不同的数据库。下图展示了根据业务需要，将用户表和订单表垂直分片到不同的数据库的方案。

垂直分片往往需要对架构和设计进行调整。通常来讲，是来不及应对互联网业务需求快速变化的；而且，它也并无法真正的解决单点瓶颈。垂直拆分可以缓解数据量和访问量带来的问题，但无法根治。如果垂直拆分之后，表中的数据量依然超过单节点所能承载的阈值，则需要水平分片来进一步处理。

SELECT * FROM t_user
SELECT * FROM t_order
SELECT * FROM t_user
SELECT * FROM t_order

图 1: 垂直分片

水平分片

水平分片又称为横向拆分。相对于垂直分片，它不再将数据根据业务逻辑分类，而是通过某个字段（或某几个字段），根据某种规则将数据分散至多个库或表中，每个分片仅包含数据的一部分。例如：根据主键分片，偶数主键的记录放入 0 库（或表），奇数主键的记录放入 1 库（或表），如下图所示。

图 2: 水平分片

水平分片从理论上突破了单机数据量处理的瓶颈，并且扩展相对自由，是数据分片的标准解决方案。

6.3.2 挑战

虽然数据分片解决了性能、可用性以及单点备份恢复等问题，但分布式的架构在获得了收益的同时，也引入了新的问题。

面对如此散乱的分片之后的数据，应用开发工程师和数据库管理员对数据库的操作变得异常繁重就是其中的重要挑战之一。他们需要知道数据需要从哪个具体的数据库的子表中获取。

另一个挑战则是，能够正确的运行在单节点数据库中的 SQL，在分片之后的数据库中并不一定能够正确运行。例如，分表导致表名称的修改，或者分页、排序、聚合分组等操作的不正确处理。

跨库事务也是分布式的数据库集群要面对的棘手事情。合理采用分表，可以在降低单表数据量的情况下，尽量使用本地事务，善于使用同库不同表可有效避免分布式事务带来的麻烦。在不能避免跨库事务的场

景，有些业务仍然需要保持事务的一致性。而基于 XA 的分布式事务由于在并发度高的场景中性能无法满足需要，并未被互联网巨头大规模使用，他们大多采用最终一致性的柔性事务代替强一致事务。

6.3.3 目标

尽量透明化分库分表所带来的影响，让使用方尽量像使用一个数据库一样使用水平分片之后的数据库集群，是 Apache ShardingSphere 数据分片模块的主要设计目标。

6.3.4 核心概念

导览

本小节主要介绍数据分片的核心概念。

表

表是透明化数据分片的关键概念。Apache ShardingSphere 通过提供多样化的表类型，适配不同场景下的数据分片需求。

逻辑表

相同结构的水平拆分数据库（表）的逻辑名称，是 SQL 中表的逻辑标识。例：订单数据根据主键尾数拆分为 10 张表，分别是 `t_order_0` 到 `t_order_9`，他们的逻辑表名为 `t_order`。

真实表

在水平拆分的数据库中真实存在的物理表。即上个示例中的 `t_order_0` 到 `t_order_9`。

绑定表

指分片规则一致的主表和子表。使用绑定表进行多表关联查询时，必须使用分片键进行关联，否则会出现笛卡尔积关联或跨库关联，从而影响查询效率。例如：`t_order` 表和 `t_order_item` 表，均按照 `order_id` 分片，并且使用 `order_id` 进行关联，则此两张表互为绑定表关系。绑定表之间的多表关联查询不会出现笛卡尔积关联，关联查询效率将大大提升。举例说明，如果 SQL 为：

```
SELECT i.* FROM t_order o JOIN t_order_item i ON o.order_id=i.order_id WHERE o.
order_id in (10, 11);
```

在不配置绑定表关系时，假设分片键 `order_id` 将数值 10 路由至第 0 片，将数值 11 路由至第 1 片，那么路由后的 SQL 应该为 4 条，它们呈现为笛卡尔积：

```
SELECT i.* FROM t_order_0 o JOIN t_order_item_0 i ON o.order_id=i.order_id WHERE o.
order_id in (10, 11);

SELECT i.* FROM t_order_0 o JOIN t_order_item_1 i ON o.order_id=i.order_id WHERE o.
order_id in (10, 11);

SELECT i.* FROM t_order_1 o JOIN t_order_item_0 i ON o.order_id=i.order_id WHERE o.
order_id in (10, 11);

SELECT i.* FROM t_order_1 o JOIN t_order_item_1 i ON o.order_id=i.order_id WHERE o.
order_id in (10, 11);
```

在配置绑定表关系，并且使用 `order_id` 进行关联后，路由的 SQL 应该为 2 条：

```
SELECT i.* FROM t_order_0 o JOIN t_order_item_0 i ON o.order_id=i.order_id WHERE o.
order_id in (10, 11);

SELECT i.* FROM t_order_1 o JOIN t_order_item_1 i ON o.order_id=i.order_id WHERE o.
order_id in (10, 11);
```

其中 `t_order` 在 FROM 的最左侧，ShardingSphere 将会以它作为整个绑定表的主表。所有路由计算将会只使用主表的策略，那么 `t_order_item` 表的分片计算将会使用 `t_order` 的条件。因此，绑定表间的分区键需要完全相同。

广播表

指所有的分片数据源中都存在的表，表结构及其数据在每个数据库中均完全一致。适用于数据量不大且需要与海量数据的表进行关联查询的场景，例如：字典表。

单表

指所有的分片数据源中仅唯一存在的表。适用于数据量不大且无需分片的表。

数据节点

数据分片的最小单元，由数据源名称和真实表组成。例：`ds_0.t_order_0`。

逻辑表与真实表的映射关系，可分为均匀分布和自定义分布两种形式。

均匀分布

指数据表在每个数据源内呈现均匀分布的态势，例如：

```
db0
  ├── t_order0
  └── t_order1
db1
  ├── t_order0
  └── t_order1
```

数据节点的配置如下：

```
db0.t_order0, db0.t_order1, db1.t_order0, db1.t_order1
```

自定义分布

指数据表呈现有特定规则的分布，例如：

```
db0
  ├── t_order0
  └── t_order1
db1
  ├── t_order2
  ├── t_order3
  └── t_order4
```

数据节点的配置如下：

```
db0.t_order0, db0.t_order1, db1.t_order2, db1.t_order3, db1.t_order4
```

分片

分片键

用于将数据库（表）水平拆分的数据库字段。例：将订单表中的订单主键的尾数取模分片，则订单主键为分片字段。SQL 中如果无分片字段，将执行全路由，性能较差。除了对单分片字段的支持，Apache ShardingSphere 也支持根据多个字段进行分片。

分片算法

用于将数据分片的算法，支持 =、>=、<=、>、<、BETWEEN 和 IN 进行分片。分片算法可由开发者自行实现，也可使用 Apache ShardingSphere 内置的分片算法语法糖，灵活度非常高。

自动化分片算法

分片算法语法糖，用于便捷的托管所有数据节点，使用者无需关注真实表的物理分布。包括取模、哈希、范围、时间等常用分片算法的实现。

自定义分片算法

提供接口让应用开发者自行实现与业务实现紧密相关的分片算法，并允许使用者自行管理真实表的物理分布。自定义分片算法又分为：

- 标准分片算法

用于处理使用单一键作为分片键的 =、IN、BETWEEN AND、>、<、>=、<= 进行分片的场景。

- 复合分片算法

用于处理使用多键作为分片键进行分片的场景，包含多个分片键的逻辑较复杂，需要应用开发者自行处理其中的复杂度。

- Hint 分片算法

用于处理使用 Hint 行分片的场景。

分片策略

包含分片键和分片算法，由于分片算法的独立性，将其独立抽离。真正可用于分片操作的是分片键 + 分片算法，也就是分片策略。

强制分片路由

对于分片字段并非由 SQL 而是其他外置条件决定的场景，可使用 SQL Hint 注入分片值。例：按照员工登录主键分库，而数据库中并无此字段。SQL Hint 支持通过 Java API 和 SQL 注释（待实现）两种方式使用。详情请参见强制分片路由。

行表达式

实现动机

配置的简化与一体化是行表达式所希望解决的两个主要问题。

在繁琐的数据分片规则配置中，随着数据节点的增多，大量的重复配置使得配置本身不易被维护。通过行表达式可以有效地简化数据节点配置工作量。

对于常见的分片算法，使用 Java 代码实现并不有助于配置的统一管理。通过行表达式书写分片算法，可以有效地将规则配置一同存放，更加易于浏览与存储。

语法说明

行表达式的使用非常直观，只需要在配置中使用 `${ expression }` 或 `$->{ expression }` 标识行表达式即可。目前支持数据节点和分片算法这两个部分的配置。行表达式的内容使用的是 Groovy 的语法，Groovy 能够支持的所有操作，行表达式均能够支持。例如：

`${begin..end}` 表示范围区间

`${[unit1, unit2, unit_x]}` 表示枚举值

行表达式中如果出现连续多个 `${ expression }` 或 `$->{ expression }` 表达式，整个表达式最终的结果将会根据每个子表达式的结果进行笛卡尔组合。

例如，以下行表达式：

```
${['online', 'offline']}_table${1..3}
```

最终会解析为：

```
online_table1, online_table2, online_table3, offline_table1, offline_table2,
offline_table3
```

配置

数据节点

对于均匀分布的数据节点，如果数据结构如下：

```
db0
  ├── t_order0
  └── t_order1
db1
  ├── t_order0
  └── t_order1
```

用行表达式可以简化为：

```
db${0..1}.t_order${0..1}
```

或者:

```
db$->{0..1}.t_order$->{0..1}
```

对于自定义的数据节点，如果数据结构如下:

```
db0
  ├── t_order0
  └── t_order1
db1
  ├── t_order2
  ├── t_order3
  └── t_order4
```

用行表达式可以简化为:

```
db0.t_order${0..1},db1.t_order${2..4}
```

或者:

```
db0.t_order$->{0..1},db1.t_order$->{2..4}
```

对于有前缀的数据节点，也可以通过行表达式灵活配置，如果数据结构如下:

```
db0
  ├── t_order_00
  ├── t_order_01
  ├── t_order_02
  ├── t_order_03
  ├── t_order_04
  ├── t_order_05
  ├── t_order_06
  ├── t_order_07
  ├── t_order_08
  ├── t_order_09
  ├── t_order_10
  ├── t_order_11
  ├── t_order_12
  ├── t_order_13
  ├── t_order_14
  ├── t_order_15
  ├── t_order_16
  ├── t_order_17
  ├── t_order_18
  ├── t_order_19
  └── t_order_20
db1
```

```
  ├── t_order_00
  ├── t_order_01
  ├── t_order_02
  ├── t_order_03
  ├── t_order_04
  ├── t_order_05
  ├── t_order_06
  ├── t_order_07
  ├── t_order_08
  ├── t_order_09
  ├── t_order_10
  ├── t_order_11
  ├── t_order_12
  ├── t_order_13
  ├── t_order_14
  ├── t_order_15
  ├── t_order_16
  ├── t_order_17
  ├── t_order_18
  ├── t_order_19
  └── t_order_20
```

可以使用分开配置的方式，先配置包含前缀的数据节点，再配置不含前缀的数据节点，再利用行表达式笛卡尔积的特性，自动组合即可。上面的示例，用行表达式可以简化为：

```
db${0..1}.t_order_0${0..9}, db${0..1}.t_order_${10..20}
```

或者：

```
db$->{0..1}.t_order_0$->{0..9}, db$->{0..1}.t_order_$->{10..20}
```

分片算法

对于只有一个分片键的使用 = 和 `IN` 进行分片的 SQL，可以使用行表达式代替编码方式配置。

行表达式内部的表达式本质上是一段 Groovy 代码，可以根据分片键进行计算的方式，返回相应的真实数据源或真实表名称。

例如：分为 10 个库，尾数为 0 的路由到后缀为 0 的数据源，尾数为 1 的路由到后缀为 1 的数据源，以此类推。用于表示分片算法的行表达式为：

```
ds${id % 10}
```

或者：

```
ds$->{id % 10}
```

分布式主键

实现动机

传统数据库软件开发中，主键自动生成技术是基本需求。而各个数据库对于该需求也提供了相应的支持，比如 MySQL 的自增键，Oracle 的自增序列等。数据分片后，不同数据节点生成全局唯一主键是非常棘手的问题。同一个逻辑表内的不同实际表之间的自增键由于无法互相感知而产生重复主键。虽然可通过约束自增主键初始值和步长的方式避免碰撞，但需引入额外的运维规则，使解决方案缺乏完整性和可扩展性。

目前有许多第三方解决方案可以完美解决这个问题，如 UUID 等依靠特定算法自生成不重复键，或者通过引入主键生成服务等。为了方便用户使用、满足不同用户不同使用场景的需求，Apache ShardingSphere 不仅提供了内置的分布式主键生成器，例如 UUID、SNOWFLAKE，还抽离出分布式主键生成器的接口，方便用户自行实现自定义的自增主键生成器。

内置的主键生成器

UUID

采用 `UUID.randomUUID()` 的方式产生分布式主键。

SNOWFLAKE

在分片规则配置模块可配置每个表的主键生成策略，默认使用雪花算法（snowflake）生成 64bit 的长整型数据。

雪花算法是由 Twitter 公布的分布式主键生成算法，它能够保证不同进程主键的不重复性，以及相同进程主键的有序性。

实现原理

在同一个进程中，它首先是通过时间位保证不重复，如果时间相同则是通过序列位保证。同时由于时间位是单调递增的，且各个服务器如果大体做了时间同步，那么生成的主键在分布式环境可以认为是总体有序的，这就保证了对索引字段的插入的高效性。例如 MySQL 的 Innodb 存储引擎的主键。

使用雪花算法生成的主键，二进制表示形式包含 4 部分，从高位到低位分表为：1bit 符号位、41bit 时间戳位、10bit 工作进程位以及 12bit 序列号位。

- 符号位（1bit）

预留的符号位，恒为零。

- 时间戳位（41bit）

41 位的时间戳可以容纳的毫秒数是 2 的 41 次幂，一年所使用的毫秒数是：`365 * 24 * 60 * 60 * 1000`。通过计算可知：

```
Math.pow(2, 41) / (365 * 24 * 60 * 60 * 1000L);
```

结果约等于 69.73 年。Apache ShardingSphere 的雪花算法的时间纪元从 `2016 年 11 月 1 日零点开始`，可以使用到 2086 年，相信能满足绝大部分系统的要求。

- 工作进程位（10bit）

该标志在 Java 进程内是唯一的，如果是分布式应用部署应保证每个工作进程的 id 是不同的。该值默认为 0，可通过属性设置。

- 序列号位（12bit）

该序列是用来在同一个毫秒内生成不同的 ID。如果在这个毫秒内生成的数量超过 4096 (2 的 12 次幂)，那么生成器会等待到下个毫秒继续生成。

雪花算法主键的详细结构见下图。

图 3: 雪花算法

时钟回拨

服务器时钟回拨会导致产生重复序列，因此默认分布式主键生成器提供了一个最大容忍的时钟回拨毫秒数。如果时钟回拨的时间超过最大容忍的毫秒数阈值，则程序报错；如果在可容忍的范围内，默认分布式主键生成器会等待时钟同步到最后一次主键生成的时间后再继续工作。最大容忍的时钟回拨毫秒数的默认值为 0，可通过属性设置。

强制分片路由

实现动机

通过解析 SQL 语句提取分片键列与值并进行分片是 Apache ShardingSphere 对 SQL 零侵入的实现方式。若 SQL 语句中没有分片条件，则无法进行分片，需要全路由。

在一些应用场景中，分片条件并不存在于 SQL，而存在于外部业务逻辑。因此需要提供一种通过外部指定分片结果的方式，在 Apache ShardingSphere 中叫做 Hint。

实现机制

Apache ShardingSphere 使用 `ThreadLocal` 管理分片键值。可以通过编程的方式向 `HintManager` 中添加分片条件，该分片条件仅在当前线程内生效。

除了通过编程的方式使用强制分片路由，Apache ShardingSphere 还可以通过 SQL 中的特殊注释的方式引用 Hint，使开发者可以采用更加透明的方式使用该功能。

指定了强制分片路由的 SQL 将会无视原有的分片逻辑，直接路由至指定的真实数据节点。

6.3.5 使用规范

背景

虽然 Apache ShardingSphere 希望能够完全兼容所有的 SQL 以及单机数据库，但分布式为数据库带来了更加复杂的场景。Apache ShardingSphere 希望能够优先解决海量数据 OLTP 的问题，OLAP 的相关支持，会一点一点的逐渐完善。

SQL

SQL 支持程度

兼容全部常用的**路由至单数据节点**的 SQL；**路由至多数据节点**的 SQL 由于场景复杂，分为稳定支持、实验性支持和不支持这三种情况。

稳定支持

全面支持 DML、DDL、DCL、TCL 和常用 DAL。支持分页、去重、排序、分组、聚合、表关联等复杂查询。

常规查询

- SELECT 主语句

```
SELECT select_expr [, select_expr ...] FROM table_reference [, table_reference ...]
[WHERE predicates]
[GROUP BY {col_name | position} [ASC | DESC], ...]
[ORDER BY {col_name | position} [ASC | DESC], ...]
[LIMIT {[offset,] row_count | row_count OFFSET offset}]
```

- select_expr

```
* |
[DISTINCT] COLUMN_NAME [AS] [alias] |
(MAX | MIN | SUM | AVG)(COLUMN_NAME | alias) [AS] [alias] |
COUNT(* | COLUMN_NAME | alias) [AS] [alias]
```

- table_reference

```
tbl_name [AS] alias] [index_hint_list]
| table_reference ([INNER] | {LEFT|RIGHT} [OUTER]) JOIN table_factor [JOIN ON
conditional_expr | USING (column_list)]
```

子查询

子查询和外层查询同时指定分片键，且分片键的值保持一致时，由内核提供稳定支持。

例如：

```
SELECT * FROM (SELECT * FROM t_order WHERE order_id = 1) o WHERE o.order_id = 1;
```

用于分页的子查询，由内核提供稳定支持。

例如：

```
SELECT * FROM (SELECT row_.*, rownum rownum_ FROM (SELECT * FROM t_order) row_
WHERE rownum <= ?) WHERE rownum > ?;
```

运算表达式中包含分片键

当分片键处于运算表达式中时，无法通过 SQL 字面提取用于分片的值，将导致全路由。

例如，假设 `create_time` 为分片键：

```
SELECT * FROM t_order WHERE to_date(create_time, 'yyyy-mm-dd') = '2019-01-01';
```

实验性支持

实验性支持特指使用 Federation 执行引擎提供支持。该引擎处于快速开发中，用户虽基本可用，但仍需大量优化，是实验性产品。

子查询

子查询和外层查询未同时指定分片键，或分片键的值不一致时，由 Federation 执行引擎提供支持。

例如：

```
SELECT * FROM (SELECT * FROM t_order) o;

SELECT * FROM (SELECT * FROM t_order) o WHERE o.order_id = 1;

SELECT * FROM (SELECT * FROM t_order WHERE order_id = 1) o;

SELECT * FROM (SELECT * FROM t_order WHERE order_id = 1) o WHERE o.order_id = 2;
```

跨库关联查询

当关联查询中的多个表分布在不同的数据库实例上时，由 Federation 执行引擎提供支持。假设 `t_order` 和 `t_order_item` 是多数据节点的分片表，并且未配置绑定表规则，`t_user` 和 `t_user_role` 是分布在不同的数据库实例上的单表，那么 Federation 执行引擎能够支持如下常用的关联查询：

```
SELECT * FROM t_order o INNER JOIN t_order_item i ON o.order_id = i.order_id WHERE
o.order_id = 1;

SELECT * FROM t_order o INNER JOIN t_user u ON o.user_id = u.user_id WHERE o.user_
id = 1;

SELECT * FROM t_order o LEFT JOIN t_user_role r ON o.user_id = r.user_id WHERE o.
user_id = 1;

SELECT * FROM t_order_item i LEFT JOIN t_user u ON i.user_id = u.user_id WHERE i.
user_id = 1;

SELECT * FROM t_order_item i RIGHT JOIN t_user_role r ON i.user_id = r.user_id
WHERE i.user_id = 1;

SELECT * FROM t_user u RIGHT JOIN t_user_role r ON u.user_id = r.user_id WHERE u.
user_id = 1;
```

不支持

以下 CASE WHEN 语句不支持：

- `CASE WHEN` 中包含子查询
- `CASE WHEN` 中使用逻辑表名（请使用表别名）

SQL 示例

稳定支持的 SQL	必要条件
SELECT * FROM tbl_name	
SELECT * FROM tbl_name WHERE (col1 = ? or col2 = ?) and col3 = ?	
SELECT * FROM tbl_name WHERE col1 = ? ORDER BY col2 DESC LIMIT ?	
SELECT COUNT(*), SUM(col1), MIN(col1), MAX(col1), AVG(col1) FROM tbl_name WHERE col1 = ?	
SELECT COUNT(col1) FROM tbl_name WHERE col2 = ? GROUP BY col1 ORDER BY col3 DESC LIMIT ?, ?	
SELECT DISTINCT * FROM tbl_name WHERE col1 = ?	
SELECT COUNT(DISTINCT col1), SUM(DISTINCT col1) FROM tbl_name	
(SELECT * FROM tbl_name)	
SELECT * FROM (SELECT * FROM tbl_name WHERE col1 = ?) o WHERE o.col1 = ?	子查询和外层查询在同一分片后的数据节点
INSERT INTO tbl_name (col1, col2,⋯) VALUES (?, ?, ⋯.)	
INSERT INTO tbl_name VALUES (?, ?,⋯.)	
INSERT INTO tbl_name (col1, col2, ⋯) VALUES(1 + 2, ?, ⋯)	
INSERT INTO tbl_name (col1, col2, ⋯) VALUES (?, ?, ⋯.), (?, ?, ⋯.)	
INSERT INTO tbl_name (col1, col2, ⋯) SELECT col1, col2, ⋯FROM tbl_name WHERE col3 = ?	INSERT 表和 SELECT 表相同表或绑定表
REPLACE INTO tbl_name (col1, col2, ⋯) SELECT col1, col2, ⋯FROM tbl_name WHERE col3 = ?	REPLACE 表和 SELECT 表相同表或绑定表
UPDATE tbl_name SET col1 = ? WHERE col2 = ?	
DELETE FROM tbl_name WHERE col1 = ?	
CREATE TABLE tbl_name (col1 int, ⋯)	
ALTER TABLE tbl_name ADD col1 varchar(10)	
DROP TABLE tbl_name	
TRUNCATE TABLE tbl_name	
CREATE INDEX idx_name ON tbl_name	
DROP INDEX idx_name ON tbl_name	
DROP INDEX idx_name	

实验性支持的 SQL	必要条件
SELECT * FROM (SELECT * FROM tbl_name) o	
SELECT * FROM (SELECT * FROM tbl_name) o WHERE o.col1 = ?	
SELECT * FROM (SELECT * FROM tbl_name WHERE col1 = ?) o	
SELECT * FROM (SELECT * FROM tbl_name WHERE col1 = ?) o WHERE o.col1 = ?	子查询和外层查询不在同一分片后的数据节点
SELECT (SELECT MAX(col1) FROM tbl_name) a, col2 from tbl_name	
SELECT SUM(DISTINCT col1), SUM(col1) FROM tbl_name	
SELECT col1, SUM(col2) FROM tbl_name GROUP BY col1 HAVING SUM(col2) > ?	
SELECT col1, col2 FROM tbl_name UNION SELECT col1, col2 FROM tbl_name	
SELECT col1, col2 FROM tbl_name UNION ALL SELECT col1, col2 FROM tbl_name	

慢 SQL	原因
SELECT * FROM tbl_name WHERE to_date(create_time, ‘yyyy-mm-dd’) = ?	分片键在运算表达式中，导致全路由

不支持的 SQL	原因	解决方案
INSERT INTO tbl_name (col1, col2, …) SELECT * FROM tbl_name WHERE col3 = ?	SELECT 子句不支持 * 和内置分布式主键生成器	无
REPLACE INTO tbl_name (col1, col2, …) SELECT * FROM tbl_name WHERE col3 = ?	SELECT 子句不支持 * 和内置分布式主键生成器	无
SELECT MAX(tbl_name.col1) FROM tbl_name	查询列是函数表达式时，查询列前不能使用表名	使用表别名

分页

完全支持 MySQL、PostgreSQL 和 Oracle 的分页查询，SQLServer 由于分页查询较为复杂，仅部分支持。

分页性能

性能瓶颈

查询偏移量过大的分页会导致数据库获取数据性能低下，以 MySQL 为例：

```
SELECT * FROM t_order ORDER BY id LIMIT 1000000, 10
```

这句 SQL 会使得 MySQL 在无法利用索引的情况下跳过 1,000,000 条记录后，再获取 10 条记录，其性能可想而知。而在分库分表的情况下（假设分为 2 个库），为了保证数据的正确性，SQL 会改写为：

```
SELECT * FROM t_order ORDER BY id LIMIT 0, 1000010
```

即将偏移量前的记录全部取出，并仅获取排序后的最后 10 条记录。这会在数据库本身就执行很慢的情况下，进一步加剧性能瓶颈。因为原 SQL 仅需要传输 10 条记录至客户端，而改写之后的 SQL 则会传输 `1,000,010 * 2` 的记录至客户端。

ShardingSphere 的优化

ShardingSphere 进行了 2 个方面的优化。

首先，采用流式处理 + 归并排序的方式来避免内存的过量占用。由于 SQL 改写不可避免的占用了额外的带宽，但并不会导致内存暴涨。与直觉不同，大多数人认为 ShardingSphere 会将 `1,000,010 * 2` 记录全部加载至内存，进而占用大量内存而导致内存溢出。但由于每个结果集的记录是有序的，因此 ShardingSphere 每次比较仅获取各个分片的当前结果集记录，驻留在内存中的记录仅为当前路由到的分片的结果集的当前游标指向而已。按归并思想合并 m 个长度为 n 的已排序数组，时间复杂度为 `O(mn(log m))`，一般分片数量 m 都较小，可以认为时间复杂度为 `O(n)`，性能损耗很小。

其次，ShardingSphere 对仅落至单分片的查询进行进一步优化。落至单分片查询的请求并不需要改写 SQL 也可以保证记录的正确性，因此在此种情况下，ShardingSphere 并未进行 SQL 改写，从而达到节省带宽的目的。

分页方案优化

由于 LIMIT 并不能通过索引查询数据，因此如果可以保证 ID 的连续性，通过 ID 进行分页是比较好的解决方案：

```
SELECT * FROM t_order WHERE id > 100000 AND id <= 100010 ORDER BY id
```

或通过记录上次查询结果的最后一条记录的 ID 进行下一页的查询：

```
SELECT * FROM t_order WHERE id > 100000 LIMIT 10
```

分页子查询

Oracle 和 SQLServer 的分页都需要通过子查询来处理，ShardingSphere 支持分页相关的子查询。

- Oracle

支持使用 rownum 进行分页：

```
SELECT * FROM (SELECT row_.*, rownum rownum_ FROM (SELECT o.order_id as order_id
FROM t_order o JOIN t_order_item i ON o.order_id = i.order_id) row_ WHERE rownum <=
?) WHERE rownum > ?
```

目前不支持 rownum + BETWEEN 的分页方式。

- SQLServer

支持使用 TOP + ROW_NUMBER() OVER 配合进行分页：

```
SELECT * FROM (SELECT TOP (?) ROW_NUMBER() OVER (ORDER BY o.order_id DESC) AS
rownum, * FROM t_order o) AS temp WHERE temp.rownum > ? ORDER BY temp.order_id
```

支持 SQLServer 2012 之后的 OFFSET FETCH 的分页方式：

```
SELECT * FROM t_order o ORDER BY id OFFSET ? ROW FETCH NEXT ? ROWS ONLY
```

目前不支持使用 WITH xxx AS (SELECT …) 的方式进行分页。由于 Hibernate 自动生成的 SQLServer 分页语句使用了 WITH 语句，因此目前并不支持基于 Hibernate 的 SQLServer 分页。目前也不支持使用两个 TOP + 子查询的方式实现分页。

- MySQL, PostgreSQL

MySQL 和 PostgreSQL 都支持 LIMIT 分页，无需子查询：

```
SELECT * FROM t_order o ORDER BY id LIMIT ? OFFSET ?
```

6.4 分布式事务

6.4.1 背景

数据库事务需要满足 ACID（原子性、一致性、隔离性、持久性）四个特性。

- 原子性（Atomicity）指事务作为整体来执行，要么全部执行，要么全不执行；
- 一致性（Consistency）指事务应确保数据从一个一致的状态转变为另一个一致的状态；
- 隔离性（Isolation）指多个事务并发执行时，一个事务的执行不应影响其他事务的执行；
- 持久性（Durability）指已提交的事务修改数据会被持久保存。

在单一数据节点中，事务仅限于对单一数据库资源的访问控制，称之为本地事务。几乎所有的成熟的关系型数据库都提供了对本地事务的原生支持。但是在基于微服务的分布式应用环境下，越来越多的应用场景要求对多个服务的访问及其相对应的多个数据库资源能纳入到同一个事务当中，分布式事务应运而生。

关系型数据库虽然对本地事务提供了完美的 ACID 原生支持。但在分布式的场景下，它却成为系统性能的桎梏。如何让数据库在分布式场景下满足 ACID 的特性或找寻相应的替代方案，是分布式事务的重点工作。

本地事务

在不开启任何分布式事务管理器的前提下，让每个数据节点各自管理自己的事务。它们之间没有协调以及通信的能力，也并不互相知晓其他数据节点事务的成功与否。本地事务在性能方面无任何损耗，但在强一致性以及最终一致性方面则力不从心。

两阶段提交

XA 协议最早的分布式事务模型是由 `X/Open` 国际联盟提出的 `X/Open Distributed Transaction Processing (DTP)` 模型，简称 XA 协议。

基于 XA 协议实现的分布式事务对业务侵入很小。它最大的优势就是对使用方透明，用户可以像使用本地事务一样使用基于 XA 协议的分布式事务。XA 协议能够严格保障事务 ACID 特性。

严格保障事务 ACID 特性是一把双刃剑。事务执行在过程中需要将所需资源全部锁定，它更加适用于执行时间确定的短事务。对于长事务来说，整个事务进行期间对数据的独占，将导致对热点数据依赖的业务系统并发性能衰退明显。因此，在高并发的性能至上场景中，基于 XA 协议的分布式事务并不是最佳选择。

柔性事务

如果将实现了 ACID 的事务要素的事务称为刚性事务的话，那么基于 BASE 事务要素的事务则称为柔性事务。BASE 是基本可用、柔性状态和最终一致性这三个要素的缩写。

- 基本可用（Basically Available）保证分布式事务参与方不一定同时在线；
- 柔性状态（Soft state）则允许系统状态更新有一定的延时，这个延时对客户来说不一定能够察觉；
- 最终一致性（Eventually consistent）通常是通过消息传递的方式保证系统的最终一致性。

在 ACID 事务中对隔离性的要求很高，在事务执行过程中，必须将所有的资源锁定。柔性事务的理念则是通过业务逻辑将互斥锁操作从资源层面上移至业务层面。通过放宽对强一致性要求，来换取系统吞吐量的提升。

基于 ACID 的强一致性事务和基于 BASE 的最终一致性事务都不是银弹，只有在最适合的场景中才能发挥它们的最大长处。可通过下表详细对比它们之间的区别，以帮助开发者进行技术选型。

	本地事务	两（三）阶段事务	柔性事务
业务改造	无	无	实现相关接口
一致性	不支持	支持	最终一致
隔离性	不支持	支持	业务方保证
并发性能	无影响	严重衰退	略微衰退
适合场景	业务方处理不一致	短事务 & 低并发	长事务 & 高并发

6.4.2 挑战

由于应用的场景不同，需要开发者能够合理的在性能与功能之间权衡各种分布式事务。

强一致的事务与柔性事务的 API 和功能并不完全相同，在它们之间并不能做到自由的透明切换。在开发决策阶段，就不得不在强一致的事务和柔性事务之间抉择，使得设计和开发成本被大幅增加。

基于 XA 的强一致事务使用相对简单，但是无法很好的应对互联网的高并发或复杂系统的长事务场景；柔性事务则需要开发者对应用进行改造，接入成本非常高，并且需要开发者自行实现资源锁定和反向补偿。

6.4.3 目标

整合现有的成熟事务方案，为本地事务、两阶段事务和柔性事务提供统一的分布式事务接口，并弥补当前方案的不足，提供一站式的分布式事务解决方案是 Apache ShardingSphere 分布式事务模块的主要设计目标。

6.4.4 核心概念

导览

本小节主要介绍分布式事务的核心概念，主要包括：

- 基于 XA 协议的两阶段事务
- 基于最终一致性的柔性事务

XA 事务

两阶段事务提交采用的是 X/OPEN 组织所定义的 DTP 模型所抽象的 AP（应用程序），TM（事务管理器）和 RM（资源管理器）概念来保证分布式事务的强一致性。其中 TM 与 RM 间采用 XA 的协议进行双向通信。与传统的本地事务相比，XA 事务增加了准备阶段，数据库除了被动接受提交指令外，还可以反向通知调用方事务是否可以被提交。`TM` 可以收集所有分支事务的准备结果，并于最后进行原子提交，以保证事务的强一致性。

Java 通过定义 JTA 接口实现了 XA 模型，JTA 接口中的 `ResourceManager` 需要数据库厂商提供 XA 驱动实现，`TransactionManager` 则需要事务管理器的厂商实现，传统的事务管理器需要同应用服务器绑定，因此使用的成本很高。而嵌入式的事务管器可以通过 jar 形式提供服务，同 Apache ShardingSphere 集成后，可保证分片后跨库事务强一致性。

通常，只有使用了事务管理器厂商所提供的 XA 事务连接池，才能支持 XA 的事务。Apache ShardingSphere 在整合 XA 事务时，采用分离 XA 事务管理和连接池管理的方式，做到对应用程序的零侵入。

图 4: 两阶段提交模型

柔性事务

柔性事务在 2008 年发表的一篇论文中被最早提到，它提倡采用最终一致性放宽对强一致性的要求，以达到事务处理并发度的提升。

TCC 和 Saga 是两种常见实现方案。他们主张开发者自行实现对数据库的反向操作，来达到数据在回滚时仍能够保证最终一致性。SEATA 实现了 SQL 反向操作的自动生成，可以使柔性事务不再必须由开发者介入才能使用。

Apache ShardingSphere 集成了 SEATA 作为柔性事务的使用方案。

6.4.5 使用规范

背景

虽然 Apache ShardingSphere 希望能够完全兼容所有的分布式事务场景，并在性能上达到最优，但在 CAP 定理所指导下，分布式事务必然有所取舍。Apache ShardingSphere 希望能够将分布式事务的选择权交给使用者，在不同的场景用使用最适合的分布式事务解决方案。

本地事务

支持项

- 完全支持非跨库事务，例如：仅分表，或分库但是路由的结果在单库中；
- 完全支持因逻辑异常导致的跨库事务。例如：同一事务中，跨两个库更新。更新完毕后，抛出空指针，则两个库的内容都能够回滚。

不支持项

- 不支持因网络、硬件异常导致的跨库事务。例如：同一事务中，跨两个库更新，更新完毕后、未提交之前，第一个库宕机，则只有第二个库数据提交，且无法回滚。

XA 事务

支持项

- 支持数据分片后的跨库事务；
- 两阶段提交保证操作的原子性和数据的强一致性；
- 服务宕机重启后，提交/回滚中的事务可自动恢复；
- 支持同时使用 XA 和非 XA 的连接池。

不支持项

- 服务宕机后，在其它机器上恢复提交/回滚中的数据。
- Savepoint。
- 事务块内，SQL 执行出现异常，执行 Commit，数据保持一致。

通过 XA 语句控制的分布式事务

- 通过 XA START 可以手动开启 XA 事务，注意该事务完全由用户管理，ShardingSphere 只负责将语句转发至后端数据库；
- 服务宕机后，需要通过 XA RECOVER 获取未提交或回滚的事务，也可以在 COMMIT 时使用 ONE PHASE 跳过 PERPARE。

```
MySQL [(none)]> use test1
                    |MySQL [(none)]> use test2
Reading table information for completion of table and column names
                    |Reading table information for completion of table and column
names
You can turn off this feature to get a quicker startup with -A
                    |You can turn off this feature to get a quicker startup with -A
```

```
                    |
Database changed
                    |Database changed
MySQL [test1]> XA START '61c052438d3eb';
                    |MySQL [test2]> XA START '61c0524390927';
Query OK, 0 rows affected (0.030 sec)
                    |Query OK, 0 rows affected (0.009 sec)

                    |
MySQL [test1]> update test set val = 'xatest1' where id = 1;
                    |MySQL [test2]> update test set val = 'xatest2' where id = 1;
Query OK, 1 row affected (0.077 sec)
                    |Query OK, 1 row affected (0.010 sec)

                    |
MySQL [test1]> XA END '61c052438d3eb';
                    |MySQL [test2]> XA END '61c0524390927';
Query OK, 0 rows affected (0.006 sec)
                    |Query OK, 0 rows affected (0.008 sec)

                    |
MySQL [test1]> XA PREPARE '61c052438d3eb';
                    |MySQL [test2]> XA PREPARE '61c0524390927';
Query OK, 0 rows affected (0.018 sec)
                    |Query OK, 0 rows affected (0.011 sec)

                    |
MySQL [test1]> XA COMMIT '61c052438d3eb';
                    |MySQL [test2]> XA COMMIT '61c0524390927';
Query OK, 0 rows affected (0.011 sec)
                    |Query OK, 0 rows affected (0.018 sec)

                    |
MySQL [test1]> select * from test where id = 1;
                    |MySQL [test2]> select * from test where id = 1;
+----+---------+
                    |+----+---------+
| id | val     |
                    || id | val     |
+----+---------+
                    |+----+---------+
|  1 | xatest1 |
                    ||  1 | xatest2 |
+----+---------+
                    |+----+---------+
1 row in set (0.016 sec)
                    |1 row in set (0.129 sec)
```

```
MySQL [test1]> XA START '61c05243994c3';
                    |MySQL [test2]> XA START '61c052439bd7b';
Query OK, 0 rows affected (0.047 sec)
                    |Query OK, 0 rows affected (0.006 sec)

                    |
MySQL [test1]> update test set val = 'xarollback' where id = 1;
                    |MySQL [test2]> update test set val = 'xarollback' where id =
1;
Query OK, 1 row affected (0.175 sec)
                    |Query OK, 1 row affected (0.008 sec)

                    |
MySQL [test1]> XA END '61c05243994c3';
                    |MySQL [test2]> XA END '61c052439bd7b';
Query OK, 0 rows affected (0.007 sec)
                    |Query OK, 0 rows affected (0.014 sec)

                    |
MySQL [test1]> XA PREPARE '61c05243994c3';
                    |MySQL [test2]> XA PREPARE '61c052439bd7b';
Query OK, 0 rows affected (0.013 sec)
                    |Query OK, 0 rows affected (0.019 sec)

                    |
MySQL [test1]> XA ROLLBACK '61c05243994c3';
                    |MySQL [test2]> XA ROLLBACK '61c052439bd7b';
Query OK, 0 rows affected (0.010 sec)
                    |Query OK, 0 rows affected (0.010 sec)

                    |
MySQL [test1]> select * from test where id = 1;
                    |MySQL [test2]> select * from test where id = 1;
+----+---------+
                    |+----+---------+
| id | val     |
                    || id | val     |
+----+---------+
                    |+----+---------+
|  1 | xatest1 |
                    ||  1 | xatest2 |
+----+---------+
                    |+----+---------+
1 row in set (0.009 sec)
                    |1 row in set (0.083 sec)

MySQL [test1]>  XA START '61c052438d3eb';
```

```
Query OK, 0 rows affected (0.030 sec)

MySQL [test1]> update test set val = 'recover' where id = 1;
Query OK, 1 row affected (0.072 sec)

MySQL [test1]> select * from test where id = 1;
+----+---------+
| id | val     |
+----+---------+
|  1 | recover |
+----+---------+
1 row in set (0.039 sec)

MySQL [test1]>  XA END '61c052438d3eb';
Query OK, 0 rows affected (0.005 sec)

MySQL [test1]> XA PREPARE '61c052438d3eb';
Query OK, 0 rows affected (0.020 sec)

MySQL [test1]> XA RECOVER;
+----------+--------------+--------------+---------------+
| formatID | gtrid_length | bqual_length | data          |
+----------+--------------+--------------+---------------+
|        1 |           13 |            0 | 61c052438d3eb |
+----------+--------------+--------------+---------------+
1 row in set (0.010 sec)

MySQL [test1]> XA RECOVER CONVERT XID;
+----------+--------------+--------------+------------------------------+
| formatID | gtrid_length | bqual_length | data                         |
+----------+--------------+--------------+------------------------------+
|        1 |           13 |            0 | 0x3631633035323433386433656 2 |
+----------+--------------+--------------+------------------------------+
1 row in set (0.011 sec)

MySQL [test1]> XA COMMIT 0x3631633035323433386433656 2;
Query OK, 0 rows affected (0.029 sec)

MySQL [test1]> XA RECOVER;
Empty set (0.011 sec)
```

柔性事务

支持项

- 支持数据分片后的跨库事务；
- 支持 RC 隔离级别；
- 通过 undo 快照进行事务回滚；
- 支持服务宕机后的，自动恢复提交中的事务。

不支持项

- 不支持除 RC 之外的隔离级别。

待优化项

- Apache ShardingSphere 和 SEATA 重复 SQL 解析。

6.5 读写分离

6.5.1 背景

面对日益增加的系统访问量，数据库的吞吐量面临着巨大瓶颈。对于同一时刻有大量并发读操作和较少写操作类型的应用系统来说，将数据库拆分为主库和从库，主库负责处理事务性的增删改操作，从库负责处理查询操作，能够有效的避免由数据更新导致的行锁，使得整个系统的查询性能得到极大的改善。

通过一主多从的配置方式，可以将查询请求均匀的分散到多个数据副本，能够进一步的提升系统的处理能力。使用多主多从的方式，不但能够提升系统的吞吐量，还能够提升系统的可用性，可以达到在任何一个数据库宕机，甚至磁盘物理损坏的情况下仍然不影响系统的正常运行。

与将数据根据分片键打散至各个数据节点的水平分片不同，读写分离则是根据 SQL 语义的分析，将读操作和写操作分别路由至主库与从库。

读写分离的数据节点中的数据内容是一致的，而水平分片的每个数据节点的数据内容却并不相同。将水平分片和读写分离联合使用，能够更加有效的提升系统性能。

6.5.2 挑战

读写分离虽然可以提升系统的吞吐量和可用性，但同时也带来了数据不一致的问题。这包括多个主库之间的数据一致性，以及主库与从库之间的数据一致性的问题。并且，读写分离也带来了与数据分片同样的问题，它同样会使得应用开发和运维人员对数据库的操作和运维变得更加复杂。下图展现了将数据分片与读写分离一同使用时，应用程序与数据库集群之间的复杂拓扑关系。

UPDATE t_user SET status='OK' WHERE id=1
SELECT * FROM t_user WHERE id=1
UPDATE t_user SET status='OK' WHERE id=1
Primary
SELECT * FROM t_user WHERE id=1
Replica

图 5: 背景

图 6: 挑战

6.5.3 目标

透明化读写分离所带来的影响，让使用方尽量像使用一个数据库一样使用主从数据库集群，是 Apache ShardingSphere 读写分离模块的主要设计目标。

6.5.4 核心概念

主库

添加、更新以及删除数据操作所使用的数据库，目前仅支持单主库。

从库

查询数据操作所使用的数据库，可支持多从库。

主从同步

将主库的数据异步的同步到从库的操作。由于主从同步的异步性，从库与主库的数据会短时间内不一致。

负载均衡策略

通过负载均衡策略将查询请求疏导至不同从库。

6.5.5 使用规范

支持项

- 提供一主多从的读写分离配置，可独立使用，也可配合数据分片使用；
- 事务中的数据读写均用主库；
- 基于 Hint 的强制主库路由。

不支持项

- 主库和从库的数据同步；
- 主库和从库的数据同步延迟导致的数据不一致；
- 主库多写；
- 主从库间的事务一致性。主从模型中，事务中的数据读写均用主库。

6.6 高可用

6.6.1 背景

高可用是现代系统的最基本诉求，作为系统基石的数据库，对于高可用的要求也是必不可少的。

在存算分离的分布式数据库体系中，存储节点和计算节点的高可用方案是不同的。对于有状态的存储节点来说，需要其自身具备数据一致性同步、探活、主节点选举等能力；对于无状态的计算节点来说，需要感知存储节点的变化的同时，还需要独立架设负载均衡器，并具备服务发现和请求分发的能力。

Apache ShardingSphere 自身提供计算节点，并通过数据库作为存储节点。因此，它采用的高可用方案是利用数据库自身的高可用方案做存储节点高可用，并自动识别其变化。

6.6.2 挑战

Apache ShardingSphere 需要自动感知多样化的存储节点高可用方案的同时，也能够动态集成对读写分离方案，是实现的主要挑战。

图 7: 概述

6.6.3 目标

** 尽可能的保证 7*24 小时不间断的数据库服务，是 Apache ShardingSphere 高可用模块的主要设计目标。**

6.6.4 核心概念

高可用类型

Apache ShardingSphere 不提供数据库高可用的能力，它通过第三方提供的高可用方案感知数据库主从关系的切换。确切来说，Apache ShardingSphere 提供数据库发现的能力，自动感知数据库主从关系，并修正计算节点对数据库的连接。

动态读写分离

高可用和读写分离一起使用时，读写分离无需配置具体的主库和从库。高可用的数据源会动态的修正读写分离的主从关系，并正确的疏导读写流量。

6.6.5 使用规范

支持项

- MySQL MGR 单主模式。

不支持项

- MySQL MGR 多主模式。

6.7 弹性伸缩

6.7.1 背景

对于使用单数据库运行的系统来说，如何安全简单地将数据迁移至水平分片的数据库上，一直以来都是一个迫切的需求；对于已经使用了 Apache ShardingSphere 的用户来说，随着业务规模的快速变化，也可能需要对现有的分片集群进行弹性扩容或缩容。

6.7.2 挑战

Apache ShardingSphere 在分片算法上提供给用户极大的自由度，但却给弹性伸缩造成了极大的挑战。找寻既能支持自定义的分片算法，又能高效地将数据节点进行扩缩容的方式，是弹性伸缩面临的第一个挑战；

同时，在伸缩过程中，不应该对正在运行的业务造成影响。尽可能减少伸缩时数据不可用的时间窗口，甚至做到用户完全无感知，是弹性伸缩的另一个挑战；

最后，弹性伸缩不应该对现有的数据造成影响，如何保证数据的正确性，是弹性伸缩的第三个挑战。

ShardingSphere-Scaling 是一个提供给用户的通用数据接入迁移及弹性伸缩的解决方案。

用户
更新
查询
终端 & 图形界面
规则配置
迁移状态
创建迁移作业
查询
Scaling-Core
存量数据迁移
增量数据迁移
检查增量数据进度
迁移进度
持久化
注册中心
自动切换
Resource Manager
人工切换
设置只读模式
数据比对
切换数据源
持久化

图 8: 概述

6.7.3 目标

支持自定义分片算法，减少数据伸缩及迁移时的业务影响，提供一站式的通用弹性伸缩解决方案，是 Apache ShardingSphere 弹性伸缩的主要设计目标。

6.7.4 状态

ShardingSphere-Scaling 从 **4.1.0** 版本开始向用户提供。当前处于 **alpha** 开发阶段。

6.7.5 核心概念

弹性伸缩作业

指一次将数据由旧规则迁移至新规则的完整流程。

存量数据

在弹性伸缩作业开始前，数据节点中已有的数据。

增量数据

在弹性伸缩作业执行过程中，业务系统所产生的新数据。

6.7.6 使用规范

支持项

- 将外围数据迁移至 Apache ShardingSphere 所管理的数据库；
- 将 Apache ShardingSphere 的数据节点进行扩容或缩容。

不支持项

- 无主键表扩缩容；
- 复合主键表扩缩容；
- 不支持在当前存储节点之上做迁移，需要准备一个全新的数据库集群作为迁移目标库。

6.8 数据加密

6.8.1 背景

安全控制一直是治理的重要环节，数据加密属于安全控制的范畴。无论对互联网公司还是传统行业来说，数据安全一直是极为重视和敏感的话题。数据加密是指对某些敏感信息通过加密规则进行数据的变形，实现敏感隐私数据的可靠保护。涉及客户安全数据或者一些商业性敏感数据，如身份证号、手机号、卡号、客户号等个人信息按照相关部门规定，都需要进行数据加密。

对于数据加密的需求，在现实的业务场景中一般分为两种情况：

1. 新业务上线，安全部门规定需将涉及用户敏感信息，例如银行、手机号码等进行加密后存储到数据库，在使用的时候再进行解密处理。因为是全新系统，因而没有存量数据清洗问题，所以实现相对简单。
2. 已上线业务，之前一直将明文存储在数据库中。相关部门突然需要对已上线业务进行加密整改。这种场景一般需要处理 3 个问题：

- 历史数据需要如何进行加密处理，即洗数。
- 如何能在不改动业务 SQL 和逻辑情况下，将新增数据进行加密处理，并存储到数据库；在使用时，再进行解密取出。
- 如何较为安全、无缝、透明化地实现业务系统在明文与密文数据间的迁移。

6.8.2 挑战

在真实业务场景中，相关业务开发团队则往往需要针对公司安全部门需求，自行实行并维护一套加解密系统。而当加密场景发生改变时，自行维护的加密系统往往又面临着重构或修改风险。此外，对于已经上线的业务，在不修改业务逻辑和 SQL 的情况下，透明化、安全低风险地实现无缝进行加密改造也相对复杂。

6.8.3 目标

根据业界对加密的需求及业务改造痛点，提供了一套完整、安全、透明化、低改造成本的数据加密整合解决方案，是 Apache ShardingSphere 数据加密模块的主要设计目标。

6.8.4 核心概念

逻辑列

用于计算加解密列的逻辑名称，是 SQL 中列的逻辑标识。逻辑列包含密文列（必须）、查询辅助列（可选）和明文列（可选）。

密文列

加密后的数据列。

查询辅助列

用于查询的辅助列。对于一些安全级别更高的非幂等加密算法，提供不可逆的幂等列用于查询。

明文列

存储明文的列，用于在加密数据迁移过程中仍旧提供服务。在洗数结束后可以删除。

6.8.5 使用规范

支持项

- 对数据库表中某个或多个列进行加解密；
- 兼容所有常用 SQL。

不支持项

- 需自行处理数据库中原始的存量数据；
- 加密字段无法支持查询不区分大小写功能；
- 加密字段无法支持比较操作，如：大于、小于、ORDER BY、BETWEEN、LIKE 等；
- 加密字段无法支持计算操作，如：AVG、SUM 以及计算表达式。

6.9 影子库压测

6.9.1 背景

在基于微服务的分布式应用架构下，业务需要多个服务是通过一系列的服务、中间件的调用来完成，所以单个服务的压力测试已无法代表真实场景。在测试环境中，如果重新搭建一整套与生产环境类似的压测环境，成本过高，并且往往无法模拟线上环境的复杂度以及流量。因此，业内通常选择全链路压测的方式，即在生产环境进行压测，这样所获得的测试结果能够准确地反应系统真实容量和性能水平。

6.9.2 挑战

全链路压测是一项复杂而庞大的工作。需要各个微服务、中间件之间配合与调整，以应对不同流量以及压测标识的透传。通常会搭建一整套压测平台以适用不同测试计划。在数据库层面需要做好数据隔离，为了保证生产数据的可靠性与完整性，需要将压测产生的数据路由到压测环境数据库，防止压测数据对生产数据库中真实数据造成污染。这就要求业务应用在执行 SQL 前，能够根据透传的压测标识，做好数据分类，将相应的 SQL 路由到与之对应的数据源。

6.9.3 目标

Apache ShardingSphere 关注于全链路压测场景下，数据库层面的解决方案。将压测数据自动路由至用户指定的数据库，是 Apache ShardingSphere 影子库模块的主要设计目标。

6.9.4 核心概念

生产库

生产环境使用的数据库。

影子库

压测数据隔离的影子数据库，与生产数据库应当使用相同的配置。

影子算法

影子算法和业务实现紧密相关，目前提供 2 种类型影子算法。

- 基于列的影子算法

通过识别 SQL 中的数据，匹配路由至影子库的场景。适用于由压测数据名单驱动的压测场景。

- 基于 Hint 的影子算法

通过识别 SQL 中的注释，匹配路由至影子库的场景。适用于由上游系统透传标识驱动的压测场景。

6.9.5 使用规范

支持项

- 基于 Hint 的影子算法支持全部 SQL；
- 基于列的影子算法仅支持部分 SQL。

不支持项

基于 Hint 的影子算法

- 无。

基于列的影子算法

- 不支持 DDL；
- 不支持范围、分组和子查询，如：BETWEEN、GROUP BY …HAVING 等。

SQL 支持列表：

- INSERT

SQL	是否支持
INSERT INTO table (column,…) VALUES (value,…)	支持
INSERT INTO table (column,…) VALUES (value,…),(value,…),…	支持
INSERT INTO table (column,…) SELECT column1 from table1 where column1 = value1	不支持

- SELECT/UPDATE/DELETE

条件类型	*SQL*	• 是否支持 *
=	SELECT/UPDATE/DELETE … WHERE column = value	支持
LIKE/NOT LIKE	SELECT/UPDATE/DELETE … WHERE column LIKE/NOT LIKE value	支持
IN/NOT IN	SELECT/UPDATE/DELETE … WHERE column IN/NOT IN (value1,value2,…)	支持
BETWEEN	SELECT/UPDATE/DELETE … WHERE column BETWEEN value1 AND value2	不支持
GROUP BY …HAVING…	SELECT/UPDATE/DELETE … WHERE …GROUP BY column HAVING column > value	不支持
子查询	SELECT/UPDATE/DELETE … WHERE column = (SELECT column FROM table WHERE column = value)	不支持

6.10 可观察性

6.10.1 背景

如何观测集群的运行状态，使运维人员可以快速掌握当前系统现状，并进行进一步的维护工作，是分布式系统的全新挑战。登录到具体服务器的点对点运维方式，无法适用于面向大量分布式服务器的场景。通过对可系统观察性数据的遥测是分布式系统推荐的运维方式。Tracing（链路跟踪）、Metrics（指标监控）和 Logging （日志）是系统运行状况的可观察性数据重要的获取手段。

APM（应用性能监控）是通过对系统可观察性数据进行采集、存储和分析，进行系统的性能监控与诊断，主要功能包括性能指标监控、调用链分析，应用拓扑图等。

Apache ShardingSphere 并不负责如何采集、存储以及展示应用性能监控的相关数据，而是为应用监控系统提供必要的指标数据。换句话说，Apache ShardingSphere 仅负责产生具有价值的数据，并通过标准协议或插件化的方式递交给相关系统。

Tracing 用于获取 SQL 解析与 SQL 执行的链路跟踪信息。Apache ShardingSphere 默认提供了对 SkyWalking，Zipkin，Jaeger 和 OpenTelemetry 的支持，也支持用户通过插件化的方式开发自定义的 Tracing 组件。

- 使用 Zipkin 和 Jaeger

通过在 agent 配置文件中开启对应的插件，并配置好 Zipkin 或者 Jaeger 服务器信息即可。

- 使用 OpenTelemetry

OpenTelemetry 在 2019 年由 OpenTracing 和 OpenCencus 合并而来。使用这种方式，只需要在 agent 配置文件中，根据 OpenTelemetry SDK 自动配置说明 ，填写合适的配置即可。

- 使用 SkyWalking

需要在 agent 配置中配置启用对应插件，并且需要同时配置使用 SkyWalking 的 apm-toolkit 工具。

- 使用 SkyWalking 的内置自动探针

Apache ShardingSphere 团队与 Apache SkyWalking 团队共同合作，在 SkyWalking 中实现了 Apache ShardingSphere 自动探针，可以将相关的应用性能数据自动发送到 SkyWalking 中。注意这种方式的自动探针不能与 Apache ShardingSphere 插件探针同时使用。

Metrics 则用于收集和展示整个集群的统计指标。Apache ShardingSphere 默认提供了对 Prometheus 的支持。

6.10.2 挑战

Tracing 和 Metrics 需要通过埋点来收集系统信息。大量的埋点使项目核心代码支离破碎，难于维护，且不易定制化统计指标。

6.10.3 目标

提供尽量多的性能和统计指标，并隔离核心代码和埋点代码，是 Apache ShardingSphere 可观察性模块的设计目标。

6.10.4 核心概念

代理

基于字节码增强和插件化设计，以提供 Tracing 和 Metrics 埋点，以及日志输出功能。需要开启代理的插件功能后，才能将监控指标数据输出至第三方 APM 中展示。

APM

APM 是应用性能监控的缩写。着眼于分布式系统的性能诊断，其主要功能包括调用链展示，应用拓扑分析等。

Tracing

链路跟踪，通过探针收集调用链数据，并发送到第三方 APM 系统。

Metrics

系统统计指标，通过探针收集，并且写入到时序数据库，供第三方应用展示。

6.10.5 使用规范

源码编译

从 Github 下载 Apache ShardingSphere 源码，对源码进行编译，操作命令如下。

```
git clone --depth 1 https://github.com/apache/shardingsphere.git
cd shardingsphere
mvn clean install -Dmaven.javadoc.skip=true -Dcheckstyle.skip=true -Drat.skip=true
-Djacoco.skip=true -DskipITs -DskipTests -Prelease
```

agent 包输出目录为 shardingsphere-agent/shardingsphere-agent-distribution/target/apache-shardingsphere-${latest.release.version}-shardingsphere-agent-bin.tar.gz

agent 配置

- 目录说明

创建 agent 目录，解压 agent 二进制包到 agent 目录。

```
mkdir agent
tar -zxvf apache-shardingsphere-${latest.release.version}-shardingsphere-agent-bin.
tar.gz -C agent
cd agent
tree
.
├── conf
│   ├── agent.yaml
│   └── logback.xml
├── plugins
│   ├── shardingsphere-agent-logging-base-${latest.release.version}.jar
│   ├── shardingsphere-agent-metrics-prometheus-${latest.release.version}.jar
│   ├── shardingsphere-agent-tracing-jaeger-${latest.release.version}.jar
│   ├── shardingsphere-agent-tracing-opentelemetry-${latest.release.version}.jar
│   ├── shardingsphere-agent-tracing-opentracing-${latest.release.version}.jar
│   └── shardingsphere-agent-tracing-zipkin-${latest.release.version}.jar
└── shardingsphere-agent.jar
```

- 配置说明

agent.yaml 是配置文件，插件有 Jaeger、OpenTracing、Zipkin、OpenTelemetry、Logging、Prometheus。如果需要开启插件时，只需要注释掉 ignoredPluginNames 中对应的插件即可。

```
applicationName: shardingsphere-agent
ignoredPluginNames:
  - Jaeger
  - OpenTracing
  - Zipkin
  - OpenTelemetry
  - Logging
  - Prometheus

plugins:
  Prometheus:
    host:  "localhost"
    port: 9090
    props:
      JVM_INFORMATION_COLLECTOR_ENABLED : "true"
  Jaeger:
    host: "localhost"
    port: 5775
    props:
      SERVICE_NAME: "shardingsphere-agent"
      JAEGER_SAMPLER_TYPE: "const"
```

```
      JAEGER_SAMPLER_PARAM: "1"
  Zipkin:
    host: "localhost"
    port: 9411
    props:
      SERVICE_NAME: "shardingsphere-agent"
      URL_VERSION: "/api/v2/spans"
      SAMPLER_TYPE: "const"
      SAMPLER_PARAM: "1"
  OpenTracing:
    props:
      OPENTRACING_TRACER_CLASS_NAME: "org.apache.skywalking.apm.toolkit.
opentracing.SkywalkingTracer"
  OpenTelemetry:
    props:
      otel.resource.attributes: "service.name=shardingsphere-agent"
      otel.traces.exporter: "zipkin"
  Logging:
    props:
      LEVEL: "INFO"
```

- 参数说明;

名称	说明	取值范围	默认值
JVM_INFORMATION_COLLECTOR_ENABLED	是否开启 JVM 采集器	true、false	true
SERVICE_NAME	链路跟踪的服务名称	自定义	shardingsphere-agent
JAEGER_SAMPLER_TYPE	Jaeger 采样率类型	const、probabilistic、ratelimiting、remote	const
JAEGER_SAMPLER_PARAM	Jaeger 采样率参数	const：0、1，probabilistic：0.0 - 1.0，ratelimiting：> 0，自定义每秒采集数量，remote：需要自定义配置远程采样率管理服务地址，JAEGER_SAMPLER_MANAGER_HOST_PORT	1（const 类型）
SAMPLER_TYPE	Zipkin 采样率类型	const、counting、ratelimiting、boundary	const
SAMPLER_PARAM	Zipkin 采样率参数	const：0、1，counting：0.01 - 1.0，ratelimiting：> 0，自定义每秒采集数量，boundary: 0.0001 - 1.0	1（const 类型）
otel.resource.attributes	open-telemetry 资源属性	字符串键值对（, 分割）	service.name=shardingsphere-agent
otel.traces.exporter	Tracing expoter	zipkin、jaeger	zipkin
otel.traces.sampler	open-telemetry 采样率类型	always_on、always_off、traceidratio	always_on
otel.traces.sampler.arg	open-telemetry 采样率参数	traceidratio：0.0 - 1.0	1.0

ShardingSphere-Proxy 中使用

- 启动脚本

  配置 shardingsphere-agent.jar 的绝对路径到 ShardingSphere-Proxy 的 start.sh 启动脚本中，请注意配置自己对应的绝对路径。

```
nohup java ${JAVA_OPTS} ${JAVA_MEM_OPTS} \
-javaagent:/xxxxx/agent/shardingsphere-agent.jar \
```

```
-classpath ${CLASS_PATH} ${MAIN_CLASS} >> ${STDOUT_FILE} 2>&1 &
```

- 启动插件

通过改造后的 ShardingSphere-Proxy 的启动脚本启动。

```
bin/start.sh
```

正常启动可以在对应的 ShardingSphere-Proxy 日志查看到 plugin 的启动日志，访问 Proxy 后，可以通过配置的地址查看到 Metric 和 Tracing 的数据。

7 用户手册

本章节面向 Apache ShardingSphere 的用户，详细阐述项目的使用说明。

7.1 ShardingSphere-JDBC

配置是 ShardingSphere-JDBC 中唯一与应用开发者交互的模块，通过它可以快速清晰的理解 ShardingSphere-JDBC 所提供的功能。

本章节是 ShardingSphere-JDBC 的配置参考手册，需要时可当做字典查阅。

ShardingSphere-JDBC 提供了 4 种配置方式，用于不同的使用场景。通过配置，应用开发者可以灵活的使用数据分片、读写分离、数据加密、影子库等功能，并且能够叠加使用。

混合规则配置与单一规则配置一脉相承，只是从配置单一的规则项到配置多个规则项的异同。

需要注意的是，规则项之间的叠加使用是通过数据源名称和表名称关联的。如果前一个规则是面向数据源聚合的，下一个规则在配置数据源时，则需要使用前一个规则配置的聚合后的逻辑数据源名称；同理，如果前一个规则是面向表聚合的，下一个规则在配置表时，则需要使用前一个规则配置的聚合后的逻辑表名称。

更多使用细节请参见使用示例。

7.1.1 Java API

简介

Java API 是 ShardingSphere-JDBC 中所有配置方式的基础，其他配置最终都将转化成为 Java API 的配置方式。

Java API 是最繁琐也是最灵活的配置方式，适合需要通过编程进行动态配置的场景下使用。

使用步骤

引入 Maven 依赖

```
<dependency>
    <groupId>org.apache.shardingsphere</groupId>
    <artifactId>shardingsphere-jdbc-core</artifactId>
    <version>${shardingsphere.version}</version>
</dependency>
```

构建数据源

ShardingSphere-JDBC 的 Java API 通过 Schema 名称、运行模式、数据源集合、规则集合以及属性配置组成。

通过 ShardingSphereDataSourceFactory 工厂创建的 ShardingSphereDataSource 实现自 JDBC 的标准接口 DataSource。

```
String schemaName = "foo_schema"; // 指定逻辑 Schema 名称
ModeConfiguration modeConfig = ... // 构建运行模式
Map<String, DataSource> dataSourceMap = ... // 构建真实数据源
Collection<RuleConfiguration> ruleConfigs = ... // 构建具体规则
Properties props = ... // 构建属性配置
DataSource dataSource = ShardingSphereDataSourceFactory.
createDataSource(schemaName, modeConfig, dataSourceMap, ruleConfigs, props);
```

模式详情请参见模式配置。

数据源详情请参见数据源配置。

规则详情请参见规则配置。

使用数据源

可通过 DataSource 选择使用原生 JDBC，或 JPA、Hibernate、MyBatis 等 ORM 框架。

以原生 JDBC 使用方式为例：

```
// 创建 ShardingSphereDataSource
DataSource dataSource = ShardingSphereDataSourceFactory.
createDataSource(schemaName, modeConfig, dataSourceMap, ruleConfigs, props);

String sql = "SELECT i.* FROM t_order o JOIN t_order_item i ON o.order_id=i.order_
id WHERE o.user_id=? AND o.order_id=?";
try (
        Connection conn = dataSource.getConnection();
        PreparedStatement ps = conn.prepareStatement(sql)) {
    ps.setInt(1, 10);
```

```
    ps.setInt(2, 1000);
    try (ResultSet rs = preparedStatement.executeQuery()) {
        while(rs.next()) {
            // ...
        }
    }
}
```

模式配置

配置入口

类名称：org.apache.shardingsphere.infra.config.mode.ModeConfiguration

可配置属性：

Standalone 持久化配置

类名称：org.apache.shardingsphere.mode.repository.standalone.StandalonePersistRepositoryConfiguration

可配置属性：

名称	数据类型	说明
type	String	持久化仓库类型
props	Properties	持久化仓库所需属性

Cluster 持久化配置

类名称：org.apache.shardingsphere.mode.repository.cluster.ClusterPersistRepositoryConfiguration

可配置属性：

名称	数据类型	说明
type	String	持久化仓库类型
namespace	String	注册中心命名空间
serverLists	String	注册中心连接地址
props	Properties	持久化仓库所需属性

持久化仓库类型的详情，请参见内置持久化仓库类型列表。

数据源配置

ShardingSphere-JDBC 支持所有的数据库 JDBC 驱动和连接池。

配置示例

示例的数据库驱动为 MySQL，连接池为 HikariCP，可以更换为其他数据库驱动和连接池。

```
Map<String, DataSource> dataSourceMap = new HashMap<>();

// 配置第 1 个数据源
HikariDataSource dataSource1 = new HikariDataSource();
dataSource1.setDriverClassName("com.mysql.jdbc.Driver");
dataSource1.setJdbcUrl("jdbc:mysql://localhost:3306/ds_1");
dataSource1.setUsername("root");
dataSource1.setPassword("");
dataSourceMap.put("ds_1", dataSource1);

// 配置第 2 个数据源
HikariDataSource dataSource2 = new HikariDataSource();
dataSource2.setDriverClassName("com.mysql.jdbc.Driver");
dataSource2.setJdbcUrl("jdbc:mysql://localhost:3306/ds_2");
dataSource2.setUsername("root");
dataSource2.setPassword("");
dataSourceMap.put("ds_2", dataSource2);

// 配置其他数据源
...
```

规则配置

规则是 Apache ShardingSphere 面向可插拔的一部分。本章节是 ShardingSphere-JDBC 的 Java 规则配置参考手册。

数据分片

配置入口

类名称：org.apache.shardingsphere.sharding.api.config.ShardingRuleConfiguration

可配置属性：

分片表配置

类名称：org.apache.shardingsphere.sharding.api.config.ShardingTableRuleConfiguration

可配置属性：

名称	数据类型	说明	默认值
logi cTable	String	分片逻辑表名称	•
act ualDat aNodes (?)	String	由数据源名 + 表名组成，以小数点分隔。多个表以逗号分隔，支持行表达式	使用已知数据源与逻辑表名称生成数据节点，用于广播表或只分库不分表且所有库的表结构完全一致的情况
databa seShar dingSt rategy (?)	S harding Strateg yConfig uration	分库策略	使用默认分库策略
tab leShar dingSt rategy (?)	S harding Strateg yConfig uration	分表策略	使用默认分表策略
k eyGene rateSt rategy (?)	KeyG enerato rConfig uration	自增列生成器	使用默认自增主键生成器

自动分片表配置

类名称：org.apache.shardingsphere.sharding.api.config.ShardingAutoTableRuleConfiguration

可配置属性：

名称	数据类型	说明	默认值
logicTable	String	分片逻辑表名称	•
act ualDataSources (?)	String	数据源名称，多个数据源以逗号分隔	使用全部配置的数据源
sh ardingStrategy (?)	S hardingStrateg yConfiguration	分片策略	使用默认分片策略
keyGe nerateStrategy (?)	KeyGenerato rConfiguration	自增列生成器	使用默认自增主键生成器

分片策略配置

标准分片策略配置

类名称:org.apache.shardingsphere.sharding.api.config.strategy.sharding.StandardShardingStrategyConfiguration

可配置属性:

名称	数据类型	说明
shardingColumn	String	分片列名称
shardingAlgorithmName	String	分片算法名称

复合分片策略配置

类名称:org.apache.shardingsphere.sharding.api.config.strategy.sharding.ComplexShardingStrategyConfiguration

可配置属性:

名称	数据类型	说明
shardingColumns	String	分片列名称，多个列以逗号分隔
shardingAlgorithmName	String	分片算法名称

Hint 分片策略配置

类名称:org.apache.shardingsphere.sharding.api.config.strategy.sharding.HintShardingStrategyConfiguration

可配置属性:

名称	数据类型	说明
shardingAlgorithmName	String	分片算法名称

不分片策略配置

类名称:org.apache.shardingsphere.sharding.api.config.strategy.sharding.NoneShardingStrategyConfiguration

可配置属性：无

算法类型的详情，请参见内置分片算法列表。

分布式序列策略配置

类名称:org.apache.shardingsphere.sharding.api.config.strategy.keygen.KeyGenerateStrategyConfiguration

可配置属性:

名称	数据类型	说明
column	String	分布式序列列名称
keyGeneratorName	String	分布式序列算法名称

算法类型的详情，请参见内置分布式序列算法列表。

读写分离

配置入口

类名称：org.apache.shardingsphere.readwritesplitting.api.ReadwriteSplittingRuleConfiguration

可配置属性:

名称	数据类型	说明
d ataSources (+)	Collection<Read writeSplittingDataSourceRuleConfiguration>	读写数据源配置
loa dBalancers (*)	Map<String, ShardingSphereAlgorithmConfiguration>	从库负载均衡算法配置

主从数据源配置

类名称:org.apache.shardingsphere.readwritesplitting.api.rule.ReadwriteSplittingDataSourceRuleConfiguration

可配置属性:

名称	数据类型	说明	默认值
name	S tring	读写分离数据源名称	•
type	S tring	读写分离类型，分为静态和动态。如 Static、Dynamic	•
props	Prope rties	读写分离所需属性，如静态：write-data-source-name、read-data-sourc e-names，动态：auto-aware-data-source-name	•
loadBala ncerName (?)	S tring	读库负载均衡算法名称	轮询负载均衡算法

算法类型的详情，请参见内置负载均衡算法列表。查询一致性路由的详情，请参见使用规范。

高可用

配置入口

类名称：org.apache.shardingsphere.dbdiscovery.api.config.DatabaseDiscoveryRuleConfiguration

可配置属性：

名称	数据类型	说明
dataSources (+)	Collection<Databa seDiscoveryDataSourceRuleConfiguration>	数据源配置
disco veryHeartbeats (+)	Map<String, D atabaseDiscoveryHeartBeatConfiguration>	监听心跳配置
discoveryTypes (+)	Map<String, ShardingSphereAlgorithmConfiguration>	数据库发现类型配置

数据源配置

类名称：org.apache.shardingsphere.dbdiscovery.api.config.rule.DatabaseDiscoveryDataSourceRuleConfiguration

可配置属性：

名称	数据类型	说明	默认值
groupName (+)	String	数据库发现组名称	•
dataSourceNames (+)	Co llection<String>	数据源名称，多个数据源用逗号分隔如：ds_0, ds_1	•
discov eryHeartbeatName (+)	String	监听心跳名称	•
d iscoveryTypeName (+)	String	数据库发现类型名称	•

监听心跳配置

类名称：org.apache.shardingsphere.dbdiscovery.api.config.rule.DatabaseDiscoveryHeartBeatConfiguration

可配置属性：

名称	数据类型	说明	• 默认值 *
props (+)	Pr operties	监听心跳属性配置，keep-alive-cron 属性配置 cron 表达式，如：'0/5 * * * * ?'	•

数据库发现类型配置

类名称：org.apache.shardingsphere.infra.config.algorithm.ShardingSphereAlgorithmConfiguration

名称	数据类型	说明	默认值
type (+)	String	数据库发现类型，如：MGR、openGauss	•
props (?)	Properties	数据库发现类型配置，如 MGR 的 group-name 属性配置	•

数据加密

配置入口

类名称：org.apache.shardingsphere.encrypt.api.config.EncryptRuleConfiguration

可配置属性：

加密表规则配置

类名称：org.apache.shardingsphere.encrypt.api.config.rule.EncryptTableRuleConfiguration

可配置属性：

名称	数据类型	说明
name	String	表名称
columns (+)	Co llection<EncryptColu mnRuleConfiguration>	加密列规则配置列表
q ueryWithCipherColumn (?)	boolean	该表是否使用加密列进行查询

加密列规则配置

类名称：org.apache.shardingsphere.encrypt.api.config.rule.EncryptColumnRuleConfiguration

可配置属性：

名称	数据类型	说明
logicColumn	String	逻辑列名称
cipherColumn	String	密文列名称
assistedQueryColumn (?)	String	查询辅助列名称
plainColumn (?)	String	原文列名称
encryptorName	String	加密算法名称

加解密算法配置

类名称：org.apache.shardingsphere.infra.config.algorithm.ShardingSphereAlgorithmConfiguration

可配置属性：

名称	数据类型	说明
name	String	加解密算法名称
type	String	加解密算法类型
properties	Properties	加解密算法属性配置

算法类型的详情，请参见内置加密算法列表。

影子库

配置入口

类名称：org.apache.shardingsphere.shadow.api.config.ShadowRuleConfiguration

可配置属性：

影子数据源配置

类名称：org.apache.shardingsphere.shadow.api.config.datasource.ShadowDataSourceConfiguration

可配置属性：

名称	数据类型	说明
sourceDataSourceName	String	生产数据源名称
shadowDataSourceName	String	影子数据源名称

影子表配置

类名称：org.apache.shardingsphere.shadow.api.config.table.ShadowTableConfiguration

可配置属性：

名称	数据类型	说明
dataSourceNames	Collection<String>	影子表关联影子数据源映射名称列表
shadowAlgorithmNames	Collection<String>	影子表关联影子算法名称列表

影子算法配置

算法类型的详情，请参见内置影子算法列表。

SQL 解析

配置入口

类名称：org.apache.shardingsphere.parser.config.SQLParserRuleConfiguration

可配置属性：

名称	数据类型	说明
sqlCommentParseEnabled (?)	boolean	是否解析 SQL 注释
parseTreeCache (?)	CacheOption	解析语法树本地缓存配置
sqlStatementCache (?)	CacheOption	SQL 语句本地缓存配置

本地缓存配置

类名称：org.apache.shardingsphere.sql.parser.api.CacheOption

可配置属性：

名称	数据类型	说明	默认值
initialCapacity	int	本地缓存初始容量	语法树本地缓存默认值 128，SQL 语句缓存默认值 2000
maximumSize	long	本地缓存最大容量	语法树本地缓存默认值 1024，SQL 语句缓存默认值 65535
concurrencyLevel	int	本地缓存并发级别，最多允许线程并发更新的个数	4

混合规则

混合配置的规则项之间的叠加使用是通过数据源名称和表名称关联的。

如果前一个规则是面向数据源聚合的，下一个规则在配置数据源时，则需要使用前一个规则配置的聚合后的逻辑数据源名称；同理，如果前一个规则是面向表聚合的，下一个规则在配置表时，则需要使用前一个规则配置的聚合后的逻辑表名称。

配置项说明

```
/* 数据源配置 */
HikariDataSource writeDataSource0 = new HikariDataSource();
writeDataSource0.setDriverClassName("com.mysql.jdbc.Driver");
writeDataSource0.setJdbcUrl("jdbc:mysql://localhost:3306/db0?serverTimezone=UTC&
useSSL=false&useUnicode=true&characterEncoding=UTF-8");
writeDataSource0.setUsername("root");
writeDataSource0.setPassword("");

HikariDataSource writeDataSource1 = new HikariDataSource();
// ... 忽略其他数据库配置项

HikariDataSource read0OfwriteDataSource0 = new HikariDataSource();
// ... 忽略其他数据库配置项

HikariDataSource read1OfwriteDataSource0 = new HikariDataSource();
// ... 忽略其他数据库配置项

HikariDataSource read0OfwriteDataSource1 = new HikariDataSource();
// ... 忽略其他数据库配置项

HikariDataSource read1OfwriteDataSource1 = new HikariDataSource();
// ... 忽略其他数据库配置项

Map<String, DataSource> datasourceMaps = new HashMap<>(6);

datasourceMaps.put("write_ds0", writeDataSource0);
datasourceMaps.put("write_ds0_read0", read0OfwriteDataSource0);
datasourceMaps.put("write_ds0_read1", read1OfwriteDataSource0);

datasourceMaps.put("write_ds1", writeDataSource1);
datasourceMaps.put("write_ds1_read0", read0OfwriteDataSource1);
datasourceMaps.put("write_ds1_read1", read1OfwriteDataSource1);

/* 分片规则配置 */
// 表达式 ds_${0..1} 枚举值表示的是主从配置的逻辑数据源名称列表
ShardingTableRuleConfiguration tOrderRuleConfiguration = new
ShardingTableRuleConfiguration("t_order", "ds_${0..1}.t_order_${[0, 1]}");
tOrderRuleConfiguration.setKeyGenerateStrategy(new
KeyGenerateStrategyConfiguration("order_id", "snowflake"));
```

```
tOrderRuleConfiguration.setTableShardingStrategy(new
StandardShardingStrategyConfiguration("order_id", "tOrderInlineShardingAlgorithm
"));
Properties tOrderShardingInlineProps = new Properties();
tOrderShardingInlineProps.setProperty("algorithm-expression", "t_order_${order_id %
2}");
tOrderRuleConfiguration.getShardingAlgorithms().putIfAbsent(
"tOrderInlineShardingAlgorithm", new ShardingSphereAlgorithmConfiguration("INLINE",
tOrderShardingInlineProps));

ShardingTableRuleConfiguration tOrderItemRuleConfiguration = new
ShardingTableRuleConfiguration("t_order_item", "ds_${0..1}.t_order_item_${[0, 1]}
");
tOrderItemRuleConfiguration.setKeyGenerateStrategy(new
KeyGenerateStrategyConfiguration("order_item_id", "snowflake"));
tOrderRuleConfiguration.setTableShardingStrategy(new
StandardShardingStrategyConfiguration("order_item_id",
"tOrderItemInlineShardingAlgorithm"));
Properties tOrderItemShardingInlineProps = new Properties();
tOrderItemShardingInlineProps.setProperty("algorithm-expression", "t_order_item_$
{order_item_id % 2}");
tOrderRuleConfiguration.getShardingAlgorithms().putIfAbsent(
"tOrderItemInlineShardingAlgorithm", new ShardingSphereAlgorithmConfiguration(
"INLINE",tOrderItemShardingInlineProps));

ShardingRuleConfiguration shardingRuleConfiguration = new
ShardingRuleConfiguration();
shardingRuleConfiguration.getTables().add(tOrderRuleConfiguration);
shardingRuleConfiguration.getTables().add(tOrderItemRuleConfiguration);
shardingRuleConfiguration.getBindingTableGroups().add("t_order, t_order_item");
shardingRuleConfiguration.getBroadcastTables().add("t_bank");
// 默认分库策略
shardingRuleConfiguration.setDefaultDatabaseShardingStrategy(new
StandardShardingStrategyConfiguration("user_id", "default_db_strategy_inline"));
Properties defaultDatabaseStrategyInlineProps = new Properties();
defaultDatabaseStrategyInlineProps.setProperty("algorithm-expression", "ds_${user_
id % 2}");
shardingRuleConfiguration.getShardingAlgorithms().put("default_db_strategy_inline",
new ShardingSphereAlgorithmConfiguration("INLINE",
defaultDatabaseStrategyInlineProps));
// 分布式序列算法配置
Properties snowflakeProperties = new Properties();
shardingRuleConfiguration.getKeyGenerators().put("snowflake", new
ShardingSphereAlgorithmConfiguration("SNOWFLAKE", snowflakeProperties));

/* 数据加密规则配置 */
Properties encryptProperties = new Properties();
encryptProperties.setProperty("aes-key-value", "123456");
```

```
EncryptColumnRuleConfiguration columnConfigAes = new
EncryptColumnRuleConfiguration("username", "username", "", "username_plain", "name_
encryptor");
EncryptColumnRuleConfiguration columnConfigTest = new
EncryptColumnRuleConfiguration("pwd", "pwd", "assisted_query_pwd", "", "pwd_
encryptor");
EncryptTableRuleConfiguration encryptTableRuleConfig = new
EncryptTableRuleConfiguration("t_user", Arrays.asList(columnConfigAes,
columnConfigTest));

Map<String, ShardingSphereAlgorithmConfiguration> encryptAlgorithmConfigs = new
LinkedHashMap<>(2, 1);
encryptAlgorithmConfigs.put("name_encryptor", new
ShardingSphereAlgorithmConfiguration("AES", encryptProperties));
encryptAlgorithmConfigs.put("pwd_encryptor", new
ShardingSphereAlgorithmConfiguration("assistedTest", encryptProperties));
EncryptRuleConfiguration encryptRuleConfiguration = new
EncryptRuleConfiguration(Collections.singleton(encryptTableRuleConfig),
encryptAlgorithmConfigs);

/* 读写分离规则配置 */
Properties readwriteProps1 = new Properties();
readwriteProps1.setProperty("write-data-source-name", "write_ds0");
readwriteProps1.setProperty("read-data-source-names", "write_ds0_read0, write_ds0_
read1");
ReadwriteSplittingDataSourceRuleConfiguration dataSourceConfiguration1 = new
ReadwriteSplittingDataSourceRuleConfiguration("ds_0", "Static", readwriteProps1,
"roundRobin");
Properties readwriteProps2 = new Properties();
readwriteProps2.setProperty("write-data-source-name", "write_ds0");
readwriteProps2.setProperty("read-data-source-names", "write_ds1_read0, write_ds1_
read1");
ReadwriteSplittingDataSourceRuleConfiguration dataSourceConfiguration2 = new
ReadwriteSplittingDataSourceRuleConfiguration("ds_1", "Static", readwriteProps2,
"roundRobin");

// 负载均衡算法
Map<String, ShardingSphereAlgorithmConfiguration> loadBalanceMaps = new HashMap<>
(1);
loadBalanceMaps.put("roundRobin", new ShardingSphereAlgorithmConfiguration("ROUND_
ROBIN", new Properties()));

ReadwriteSplittingRuleConfiguration readWriteSplittingyRuleConfiguration = new
ReadwriteSplittingRuleConfiguration(Arrays.asList(dataSourceConfiguration1,
dataSourceConfiguration2), loadBalanceMaps);

/* 其他配置 */
Properties otherProperties = new Properties();
```

```
otherProperties.setProperty("sql-show", "true");

/* shardingDataSource 就是最终被 ORM 框架或其他 jdbc 框架引用的数据源名称 */
DataSource shardingDataSource = ShardingSphereDataSourceFactory.
createDataSource(datasourceMaps, Arrays.asList(shardingRuleConfiguration,
readWriteSplittingyRuleConfiguration, encryptRuleConfiguration), otherProperties);
```

7.1.2 YAML 配置

简介

YAML 提供通过配置文件的方式与 ShardingSphere-JDBC 交互。配合治理模块一同使用时，持久化在配置中心的配置均为 YAML 格式。

YAML 配置是最常见的配置方式，可以省略编程的复杂度，简化用户配置。

使用步骤

引入 Maven 依赖

```
<dependency>
    <groupId>org.apache.shardingsphere</groupId>
    <artifactId>shardingsphere-jdbc-core</artifactId>
    <version>${shardingsphere.version}</version>
</dependency>
```

配置 YAML

ShardingSphere-JDBC 的 YAML 配置文件通过 Schema 名称、运行模式、数据源集合、规则集合以及属性配置组成。

```
# JDBC 中的数据源的别名。在集群模式，使用该参数联通 ShardingSphere-JDBC 与 ShardingSphere-
Proxy 共同使用。
# 默认值: logic_db
schemaName (?):

mode:

dataSources:

rules:
- !FOO_XXX
    ...
- !BAR_XXX
    ...
```

```
props:
  key_1: value_1
  key_2: value_2
```

模式详情请参见模式配置。

数据源详情请参见数据源配置。

规则详情请参见规则配置。

构建数据源

通过 YamlShardingSphereDataSourceFactory 工厂创建的 ShardingSphereDataSource 实现自 JDBC 的标准接口 DataSource。

```
File yamlFile = // 指定 YAML 文件路径
DataSource dataSource = YamlShardingSphereDataSourceFactory.
createDataSource(yamlFile);
```

使用数据源

使用方式同 Java API。

语法说明

!! 表示实例化该类

! 表示自定义别名

- 表示可以包含一个或多个

[] 表示数组，可以与减号相互替换使用

模式配置

配置项说明

```
mode (?): # 不配置则默认内存模式
  type: # 运行模式类型。可选配置：Memory、Standalone、Cluster
  repository (?): # 久化仓库配置。Memory 类型无需持久化
  overwrite: # 是否使用本地配置覆盖持久化配置
```

内存模式

```
mode:
  type: Memory
```

单机模式

```
mode:
  type: Standalone
  repository:
    type: # 持久化仓库类型
    props: # 持久化仓库所需属性
      foo_key: foo_value
      bar_key: bar_value
  overwrite: # 是否使用本地配置覆盖持久化配置
```

集群模式

```
mode:
  type: Cluster
  repository:
    type: # 持久化仓库类型
    props: # 持久化仓库所需属性
      namespace: # 注册中心命名空间
      server-lists: # 注册中心连接地址
      foo_key: foo_value
      bar_key: bar_value
  overwrite: # 是否使用本地配置覆盖持久化配置
```

持久化仓库类型的详情，请参见内置持久化仓库类型列表。

数据源配置

数据源配置分为单数据源配置和多数据源配置。ShardingSphere-JDBC 支持所有的数据库 JDBC 驱动和连接池。

示例的数据库驱动为 MySQL，连接池为 HikariCP，可以更换为其他数据库驱动和连接池。

配置项说明

```
dataSources: # 数据源配置，可配置多个 <data-source-name>
  <data-source-name>: # 数据源名称
    dataSourceClassName: # 数据源完整类名
    driverClassName: # 数据库驱动类名，以数据库连接池自身配置为准
    jdbcUrl: # 数据库 URL 连接，以数据库连接池自身配置为准
    username: # 数据库用户名，以数据库连接池自身配置为准
    password: # 数据库密码，以数据库连接池自身配置为准
    # ... 数据库连接池的其它属性
```

配置示例

```
dataSources:
  ds_1:
    dataSourceClassName: com.zaxxer.hikari.HikariDataSource
    driverClassName: com.mysql.jdbc.Driver
    jdbcUrl: jdbc:mysql://localhost:3306/ds_1
    username: root
    password:
  ds_2:
    dataSourceClassName: com.zaxxer.hikari.HikariDataSource
    driverClassName: com.mysql.jdbc.Driver
    jdbcUrl: jdbc:mysql://localhost:3306/ds_2
    username: root
    password:

  # 配置其他数据源
```

规则配置

规则是 Apache ShardingSphere 面向可插拔的一部分。本章节是 ShardingSphere-JDBC 的 YAML 规则配置参考手册。

数据分片

配置项说明

```
rules:
- !SHARDING
  tables: # 数据分片规则配置
    <logic-table-name> (+): # 逻辑表名称
      actualDataNodes (?): # 由数据源名 + 表名组成（参考 Inline 语法规则）
      databaseStrategy (?): # 分库策略，缺省表示使用默认分库策略，以下的分片策略只能选其一
```

```
      standard: # 用于单分片键的标准分片场景
        shardingColumn: # 分片列名称
        shardingAlgorithmName: # 分片算法名称
      complex: # 用于多分片键的复合分片场景
        shardingColumns: # 分片列名称，多个列以逗号分隔
        shardingAlgorithmName: # 分片算法名称
      hint: # Hint 分片策略
        shardingAlgorithmName: # 分片算法名称
      none: # 不分片
    tableStrategy: # 分表策略，同分库策略
    keyGenerateStrategy: # 分布式序列策略
      column: # 自增列名称，缺省表示不使用自增主键生成器
      keyGeneratorName: # 分布式序列算法名称
autoTables: # 自动分片表规则配置
  t_order_auto: # 逻辑表名称
    actualDataSources (?): # 数据源名称
    shardingStrategy: # 切分策略
      standard: # 用于单分片键的标准分片场景
        shardingColumn: # 分片列名称
        shardingAlgorithmName: # 自动分片算法名称
bindingTables (+): # 绑定表规则列表
  - <logic_table_name_1, logic_table_name_2, ...>
  - <logic_table_name_1, logic_table_name_2, ...>
broadcastTables (+): # 广播表规则列表
  - <table-name>
  - <table-name>
defaultDatabaseStrategy: # 默认数据库分片策略
defaultTableStrategy: # 默认表分片策略
defaultKeyGenerateStrategy: # 默认的分布式序列策略
defaultShardingColumn: # 默认分片列名称

# 分片算法配置
shardingAlgorithms:
  <sharding-algorithm-name> (+): # 分片算法名称
    type: # 分片算法类型
    props: # 分片算法属性配置
    # ...

# 分布式序列算法配置
keyGenerators:
  <key-generate-algorithm-name> (+): # 分布式序列算法名称
    type: # 分布式序列算法类型
    props: # 分布式序列算法属性配置
    # ...
```

读写分离

配置项说明

```
rules:
- !READWRITE_SPLITTING
  dataSources:
    <data-source-name> (+): # 读写分离逻辑数据源名称
        type: # 读写分离类型，比如：Static，Dynamic
        props:
          auto-aware-data-source-name: # 自动发现数据源名称（与数据库发现配合使用）
          write-data-source-name: # 写库数据源名称
          read-data-source-names: # 读库数据源名称，多个从数据源用逗号分隔
      loadBalancerName: # 负载均衡算法名称

  # 负载均衡算法配置
  loadBalancers:
    <load-balancer-name> (+): # 负载均衡算法名称
      type: # 负载均衡算法类型
      props: # 负载均衡算法属性配置
        # ...
```

算法类型的详情，请参见内置负载均衡算法列表。查询一致性路由的详情，请参见使用规范。

高可用

配置项说明

```
rules:
- !DB_DISCOVERY
  dataSources:
    <data-source-name> (+): # 逻辑数据源名称
      dataSourceNames: # 数据源名称列表
        - <data-source>
        - <data-source>
      discoveryHeartbeatName: # 检测心跳名称
      discoveryTypeName: # 数据库发现类型名称

  # 心跳检测配置
  discoveryHeartbeats:
    <discovery-heartbeat-name> (+): # 心跳名称
      props:
        keep-alive-cron: # cron 表达式，如：'0/5 * * * * ?'

  # 数据库发现类型配置
  discoveryTypes:
    <discovery-type-name> (+): # 数据库发现类型名称
```

```
      type: # 数据库发现类型，如：MGR、openGauss
      props (?):
        group-name: 92504d5b-6dec-11e8-91ea-246e9612aaf1 # 数据库发现类型必要参数，如
MGR 的 group-name
```

数据加密

配置项说明

```
rules:
- !ENCRYPT
  tables:
    <table-name> (+): # 加密表名称
      columns:
        <column-name> (+): # 加密列名称
          cipherColumn: # 密文列名称
          assistedQueryColumn (?):  # 查询辅助列名称
          plainColumn (?): # 原文列名称
          encryptorName: # 加密算法名称
      queryWithCipherColumn(?): # 该表是否使用加密列进行查询

  # 加密算法配置
  encryptors:
    <encrypt-algorithm-name> (+): # 加解密算法名称
      type: # 加解密算法类型
      props: # 加解密算法属性配置
        # ...

  queryWithCipherColumn: # 是否使用加密列进行查询。在有原文列的情况下，可以使用原文列进行查询
```

算法类型的详情，请参见内置加密算法列表。

影子库

配置项说明

```
rules:
- !SHADOW
  dataSources:
    shadowDataSource:
      sourceDataSourceName: # 生产数据源名称
      shadowDataSourceName: # 影子数据源名称
  tables:
    <table-name>:
      dataSourceNames: # 影子表关联影子数据源名称列表
```

```
        - <shadow-data-source>
      shadowAlgorithmNames: # 影子表关联影子算法名称列表
        - <shadow-algorithm-name>
  defaultShadowAlgorithmName: # 默认影子算法名称
  shadowAlgorithms:
    <shadow-algorithm-name> (+): # 影子算法名称
      type: # 影子算法类型
      props: # 影子算法属性配置
        # ...
```

混合规则

混合配置的规则项之间的叠加使用是通过数据源名称和表名称关联的。

如果前一个规则是面向数据源聚合的，下一个规则在配置数据源时，则需要使用前一个规则配置的聚合后的逻辑数据源名称；同理，如果前一个规则是面向表聚合的，下一个规则在配置表时，则需要使用前一个规则配置的聚合后的逻辑表名称。

配置项说明

```
dataSources: # 配置真实存在的数据源作为名称
  write_ds:
    # ... 省略具体配置
  read_ds_0:
    # ... 省略具体配置
  read_ds_1:
    # ... 省略具体配置

rules:
  - !SHARDING # 配置数据分片规则
    tables:
      t_user:
        actualDataNodes: ds.t_user_${0..1} # 数据源名称 `ds` 使用读写分离配置的逻辑数据源
名称
        tableStrategy:
          standard:
            shardingColumn: user_id
            shardingAlgorithmName: t_user_inline
    shardingAlgorithms:
      t_user_inline:
        type: INLINE
        props:
          algorithm-expression: t_user_${user_id % 2}

  - !ENCRYPT # 配置数据加密规则
    tables:
```

```
      t_user: # 表名称 `t_user` 使用数据分片配置的逻辑表名称
        columns:
          pwd:
            plainColumn: plain_pwd
            cipherColumn: cipher_pwd
            encryptorName: encryptor_aes
    encryptors:
      encryptor_aes:
        type: aes
        props:
          aes-key-value: 123456abc

  - !READWRITE_SPLITTING # 配置读写分离规则
    dataSources:
      ds: # 读写分离的逻辑数据源名称 `ds` 用于在数据分片中使用
        type: Static
        props:
          write-data-source-name: write_ds  # 使用真实存在的数据源名称 `write_ds`
          read-data-source-names: read_ds_0, read_ds_1 # 使用真实存在的数据源名称
`read_ds_0` `read_ds_1`
        loadBalancerName: roundRobin
    loadBalancers:
      roundRobin:
        type: ROUND_ROBIN

props:
  sql-show: true
```

SQL 解析

配置项说明

```
rules:
- !SQL_PARSER
  sqlCommentParseEnabled: # 是否解析 SQL 注释
  sqlStatementCache: # SQL 语句本地缓存配置项
    initialCapacity: # 本地缓存初始容量
    maximumSize: # 本地缓存最大容量
    concurrencyLevel: # 本地缓存并发级别，最多允许线程并发更新的个数
  parseTreeCache: # 解析树本地缓存配置项
    initialCapacity: # 本地缓存初始容量
    maximumSize: # 本地缓存最大容量
    concurrencyLevel: # 本地缓存并发级别，最多允许线程并发更新的个数
```

7.1.3 Spring Boot Starter

简介

ShardingSphere-JDBC 提供官方的 Spring Boot Starter，使开发者可以非常便捷的整合 ShardingSphere-JDBC 和 Spring Boot。

使用步骤

引入 Maven 依赖

```
<dependency>
    <groupId>org.apache.shardingsphere</groupId>
    <artifactId>shardingsphere-jdbc-core-spring-boot-starter</artifactId>
    <version>${shardingsphere.version}</version>
</dependency>
```

在 Spring 中使用 ShardingSphere 数据源

直接通过注入的方式即可使用 ShardingSphereDataSource；或者将 ShardingSphereDataSource 配置在 JPA、Hibernate、MyBatis 等 ORM 框架中配合使用。

```
@Resource
private DataSource dataSource;
```

模式配置

缺省配置为使用内存模式。

配置项说明

```
spring.shardingsphere.mode.type= # 运行模式类型。可选配置：Memory、Standalone、Cluster
spring.shardingsphere.mode.repository= # 持久化仓库配置。Memory 类型无需持久化
spring.shardingsphere.mode.overwrite= # 是否使用本地配置覆盖持久化配置
```

内存模式

```
spring.shardingsphere.mode.type=Memory
```

单机模式

```
spring.shardingsphere.mode.type=Standalone
spring.shardingsphere.mode.repository.type= # 持久化仓库类型
spring.shardingsphere.mode.repository.props.<key>= # 持久化仓库所需属性
spring.shardingsphere.mode.overwrite= # 是否使用本地配置覆盖持久化配置
```

集群模式

```
spring.shardingsphere.mode.type=Cluster
spring.shardingsphere.mode.repository.type= # 持久化仓库类型
spring.shardingsphere.mode.repository.props.namespace= # 注册中心命名空间
spring.shardingsphere.mode.repository.props.server-lists= # 注册中心连接地址
spring.shardingsphere.mode.repository.props.<key>= # 持久化仓库所需属性
spring.shardingsphere.mode.overwrite= # 是否使用本地配置覆盖持久化配置
```

持久化仓库类型的详情，请参见内置持久化仓库类型列表。

数据源配置

使用本地数据源

配置项说明

```
spring.shardingsphere.datasource.names= # 真实数据源名称，多个数据源用逗号区分

# <actual-data-source-name> 表示真实数据源名称
spring.shardingsphere.datasource.<actual-data-source-name>.type= # 数据库连接池全类名
spring.shardingsphere.datasource.<actual-data-source-name>.driver-class-name= # 数据
库驱动类名，以数据库连接池自身配置为准
spring.shardingsphere.datasource.<actual-data-source-name>.jdbc-url= # 数据库 URL 连
接，以数据库连接池自身配置为准
spring.shardingsphere.datasource.<actual-data-source-name>.username= # 数据库用户名，
以数据库连接池自身配置为准
spring.shardingsphere.datasource.<actual-data-source-name>.password= # 数据库密码，以
数据库连接池自身配置为准
spring.shardingsphere.datasource.<actual-data-source-name>.<xxx>= # ... 数据库连接池的
其它属性
```

配置示例

示例的数据库驱动为 MySQL，连接池为 HikariCP，可以更换为其他数据库驱动和连接池。

```
# 配置真实数据源
spring.shardingsphere.datasource.names=ds1,ds2

# 配置第 1 个数据源
spring.shardingsphere.datasource.ds1.type=com.zaxxer.hikari.HikariDataSource
spring.shardingsphere.datasource.ds1.driver-class-name=com.mysql.jdbc.Driver
spring.shardingsphere.datasource.ds1.jdbc-url=jdbc:mysql://localhost:3306/ds1
spring.shardingsphere.datasource.ds1.username=root
spring.shardingsphere.datasource.ds1.password=

# 配置第 2 个数据源
spring.shardingsphere.datasource.ds2.type=com.zaxxer.hikari.HikariDataSource
spring.shardingsphere.datasource.ds2.driver-class-name=com.mysql.jdbc.Driver
spring.shardingsphere.datasource.ds2.jdbc-url=jdbc:mysql://localhost:3306/ds2
spring.shardingsphere.datasource.ds2.username=root
spring.shardingsphere.datasource.ds2.password=
```

使用 JNDI 数据源

如果计划使用 JNDI 配置数据库，在应用容器（如 Tomcat）中使用 ShardingSphere-JDBC 时，可使用 `spring.shardingsphere.datasource.${datasourceName}.jndiName` 来代替数据源的一系列配置。

配置项说明

```
spring.shardingsphere.datasource.names= # 真实数据源名称，多个数据源用逗号区分

# <actual-data-source-name> 表示真实数据源名称
spring.shardingsphere.datasource.<actual-data-source-name>.jndi-name= # 数据源 JNDI
```

配置示例

```
# 配置真实数据源
spring.shardingsphere.datasource.names=ds1,ds2

# 配置第 1 个数据源
spring.shardingsphere.datasource.ds1.jndi-name=java:comp/env/jdbc/ds1
# 配置第 2 个数据源
spring.shardingsphere.datasource.ds2.jndi-name=java:comp/env/jdbc/ds2
```

规则配置

规则是 Apache ShardingSphere 面向可插拔的一部分。本章节是 ShardingSphere-JDBC 的 Spring Boot Starter 规则配置参考手册。

数据分片

配置项说明

```
spring.shardingsphere.datasource.names= # 省略数据源配置，请参考使用手册

# 标准分片表配置
spring.shardingsphere.rules.sharding.tables.<table-name>.actual-data-nodes= # 由数据源名 + 表名组成，以小数点分隔。多个表以逗号分隔，支持 inline 表达式。缺省表示使用已知数据源与逻辑表名称生成数据节点，用于广播表（即每个库中都需要一个同样的表用于关联查询，多为字典表）或只分库不分表且所有库的表结构完全一致的情况

# 分库策略，缺省表示使用默认分库策略，以下的分片策略只能选其一

# 用于单分片键的标准分片场景
spring.shardingsphere.rules.sharding.tables.<table-name>.database-strategy.
standard.sharding-column= # 分片列名称
spring.shardingsphere.rules.sharding.tables.<table-name>.database-strategy.
standard.sharding-algorithm-name= # 分片算法名称

# 用于多分片键的复合分片场景
spring.shardingsphere.rules.sharding.tables.<table-name>.database-strategy.complex.
sharding-columns= # 分片列名称，多个列以逗号分隔
spring.shardingsphere.rules.sharding.tables.<table-name>.database-strategy.complex.
sharding-algorithm-name= # 分片算法名称

# 用于 Hint 的分片策略
spring.shardingsphere.rules.sharding.tables.<table-name>.database-strategy.hint.
sharding-algorithm-name= # 分片算法名称

# 分表策略，同分库策略
spring.shardingsphere.rules.sharding.tables.<table-name>.table-strategy.xxx= # 省略

# 自动分片表配置
spring.shardingsphere.rules.sharding.auto-tables.<auto-table-name>.actual-data-
sources= # 数据源名

spring.shardingsphere.rules.sharding.auto-tables.<auto-table-name>.sharding-
strategy.standard.sharding-column= # 分片列名称
spring.shardingsphere.rules.sharding.auto-tables.<auto-table-name>.sharding-
strategy.standard.sharding-algorithm-name= # 自动分片算法名称
```

```
# 分布式序列策略配置
spring.shardingsphere.rules.sharding.tables.<table-name>.key-generate-strategy.
column= # 分布式序列列名称
spring.shardingsphere.rules.sharding.tables.<table-name>.key-generate-strategy.key-
generator-name= # 分布式序列算法名称

spring.shardingsphere.rules.sharding.binding-tables[0]= # 绑定表规则列表
spring.shardingsphere.rules.sharding.binding-tables[1]= # 绑定表规则列表
spring.shardingsphere.rules.sharding.binding-tables[x]= # 绑定表规则列表

spring.shardingsphere.rules.sharding.broadcast-tables[0]= # 广播表规则列表
spring.shardingsphere.rules.sharding.broadcast-tables[1]= # 广播表规则列表
spring.shardingsphere.rules.sharding.broadcast-tables[x]= # 广播表规则列表

spring.shardingsphere.sharding.default-database-strategy.xxx= # 默认数据库分片策略
spring.shardingsphere.sharding.default-table-strategy.xxx= # 默认表分片策略
spring.shardingsphere.sharding.default-key-generate-strategy.xxx= # 默认分布式序列策略
spring.shardingsphere.sharding.default-sharding-column= # 默认分片列名称

# 分片算法配置
spring.shardingsphere.rules.sharding.sharding-algorithms.<sharding-algorithm-name>.
type= # 分片算法类型
spring.shardingsphere.rules.sharding.sharding-algorithms.<sharding-algorithm-name>.
props.xxx= # 分片算法属性配置

# 分布式序列算法配置
spring.shardingsphere.rules.sharding.key-generators.<key-generate-algorithm-name>.
type= # 分布式序列算法类型
spring.shardingsphere.rules.sharding.key-generators.<key-generate-algorithm-name>.
props.xxx= # 分布式序列算法属性配置
```

算法类型的详情，请参见内置分片算法列表和内置分布式序列算法列表。

注意事项

行表达式标识符可以使用 `${...}` 或 `$->{...}`，但前者与 Spring 本身的属性文件占位符冲突，因此在 Spring 环境中使用行表达式标识符建议使用 `$->{...}`。

读写分离

配置项说明

```
spring.shardingsphere.datasource.names= # 省略数据源配置，请参考使用手册

spring.shardingsphere.rules.readwrite-splitting.data-sources.<readwrite-splitting-data-source-name>.type= # 读写分离类型，如：Static，Dynamic
spring.shardingsphere.rules.readwrite-splitting.data-sources.<readwrite-splitting-data-source-name>.props.auto-aware-data-source-name= # 自动发现数据源名称（与数据库发现配合使用）
spring.shardingsphere.rules.readwrite-splitting.data-sources.<readwrite-splitting-data-source-name>.props.write-data-source-name= # 写数据源名称
spring.shardingsphere.rules.readwrite-splitting.data-sources.<readwrite-splitting-data-source-name>.props.read-data-source-names= # 读数据源名称，多个从数据源用逗号分隔
spring.shardingsphere.rules.readwrite-splitting.data-sources.<readwrite-splitting-data-source-name>.load-balancer-name= # 负载均衡算法名称

# 负载均衡算法配置
spring.shardingsphere.rules.readwrite-splitting.load-balancers.<load-balance-algorithm-name>.type= # 负载均衡算法类型
spring.shardingsphere.rules.readwrite-splitting.load-balancers.<load-balance-algorithm-name>.props.xxx= # 负载均衡算法属性配置
```

算法类型的详情，请参见内置负载均衡算法列表。查询一致性路由的详情，请参见使用规范。

高可用

配置项说明

```
spring.shardingsphere.datasource.names= # 省略数据源配置，请参考使用手册

spring.shardingsphere.rules.database-discovery.data-sources.<database-discovery-data-source-name>.data-source-names= # 数据源名称，多个数据源用逗号分隔 如：ds_0，ds_1
spring.shardingsphere.rules.database-discovery.data-sources.<database-discovery-data-source-name>.discovery-heartbeat-name= # 检测心跳名称
spring.shardingsphere.rules.database-discovery.data-sources.<database-discovery-data-source-name>.discovery-type-name= # 数据库发现类型名称

spring.shardingsphere.rules.database-discovery.discovery-heartbeats.<discovery-heartbeat-name>.props.keep-alive-cron= # cron 表达式，如：'0/5 * * * * ?'

spring.shardingsphere.rules.database-discovery.discovery-types.<discovery-type-name>.type= # 数据库发现类型，如：MGR、openGauss
spring.shardingsphere.rules.database-discovery.discovery-types.<discovery-type-name>.props.group-name= # 数据库发现类型必要参数，如 MGR 的 group-name
```

数据加密

配置项说明

```
spring.shardingsphere.datasource.names= # 省略数据源配置，请参考使用手册

spring.shardingsphere.rules.encrypt.tables.<table-name>.query-with-cipher-column= # 该表是否使用加密列进行查询
spring.shardingsphere.rules.encrypt.tables.<table-name>.columns.<column-name>.cipher-column= # 加密列名称
spring.shardingsphere.rules.encrypt.tables.<table-name>.columns.<column-name>.assisted-query-column= # 查询列名称
spring.shardingsphere.rules.encrypt.tables.<table-name>.columns.<column-name>.plain-column= # 原文列名称
spring.shardingsphere.rules.encrypt.tables.<table-name>.columns.<column-name>.encryptor-name= # 加密算法名称

# 加密算法配置
spring.shardingsphere.rules.encrypt.encryptors.<encrypt-algorithm-name>.type= # 加密算法类型
spring.shardingsphere.rules.encrypt.encryptors.<encrypt-algorithm-name>.props.xxx= # 加密算法属性配置

spring.shardingsphere.rules.encrypt.queryWithCipherColumn= # 是否使用加密列进行查询。在有原文列的情况下，可以使用原文列进行查询
```

算法类型的详情，请参见内置加密算法列表。

影子库

配置项说明

```
spring.shardingsphere.datasource.names= # 省略数据源配置，请参考使用手册

spring.shardingsphere.rules.shadow.data-sources.shadow-data-source.source-data-source-name= # 生产数据源名称
spring.shardingsphere.rules.shadow.data-sources.shadow-data-source.shadow-data-source-name= # 影子数据源名称

spring.shardingsphere.rules.shadow.tables.<table-name>.data-source-names= # 影子表关联影子数据源名称列表（多个值用"," 隔开）
spring.shardingsphere.rules.shadow.tables.<table-name>.shadow-algorithm-names= # 影子表关联影子算法名称列表（多个值用"," 隔开）

spring.shardingsphere.rules.shadow.defaultShadowAlgorithmName= # 默认影子算法名称，选配项。
```

```
spring.shardingsphere.rules.shadow.shadow-algorithms.<shadow-algorithm-name>.type=
# 影子算法类型
spring.shardingsphere.rules.shadow.shadow-algorithms.<shadow-algorithm-name>.props.
xxx= # 影子算法属性配置
```

混合规则

混合配置的规则项之间的叠加使用是通过数据源名称和表名称关联的。

如果前一个规则是面向数据源聚合的，下一个规则在配置数据源时，则需要使用前一个规则配置的聚合后的逻辑数据源名称；同理，如果前一个规则是面向表聚合的，下一个规则在配置表时，则需要使用前一个规则配置的聚合后的逻辑表名称。

配置项说明

```
# 数据源配置
# 数据源名称，多数据源以逗号分隔
spring.shardingsphere.datasource.names= write-ds0,write-ds1,write-ds0-read0,write-
ds1-read0

spring.shardingsphere.datasource.write-ds0.jdbc-url= # 数据库 URL 连接
spring.shardingsphere.datasource.write-ds0.type=  # 数据库连接池类名称
spring.shardingsphere.datasource.write-ds0.driver-class-name= # 数据库驱动类名
spring.shardingsphere.datasource.write-ds0.username= # 数据库用户名
spring.shardingsphere.datasource.write-ds0.password= # 数据库密码
spring.shardingsphere.datasource.write-ds0.xxx=  # 数据库连接池的其它属性

spring.shardingsphere.datasource.write-ds1.jdbc-url= # 数据库 URL 连接
# 忽略其他数据库配置项

spring.shardingsphere.datasource.write-ds0-read0.jdbc-url= # 数据库 URL 连接
# 忽略其他数据库配置项

spring.shardingsphere.datasource.write-ds1-read0.jdbc-url= # 数据库 URL 连接
# 忽略其他数据库配置项

# 分片规则配置
# 分库策略
spring.shardingsphere.rules.sharding.default-database-strategy.standard.sharding-
column=user_id
spring.shardingsphere.rules.sharding.default-database-strategy.standard.sharding-
algorithm-name=default-database-strategy-inline
# 绑定表规则，多组绑定规则使用数组形式配置
spring.shardingsphere.rules.sharding.binding-tables[0]=t_user,t_user_detail # 绑定表
名称，多个表之间以逗号分隔
spring.shardingsphere.rules.sharding.binding-tables[1]= # 绑定表名称，多个表之间以逗号分
隔
```

```
spring.shardingsphere.rules.sharding.binding-tables[x]= # 绑定表名称，多个表之间以逗号分隔
# 广播表规则配置
spring.shardingsphere.rules.sharding.broadcast-tables= # 广播表名称，多个表之间以逗号分隔

# 分表策略
# 表达式 `ds_$->{0..1}`枚举的数据源为读写分离配置的逻辑数据源名称
spring.shardingsphere.rules.sharding.tables.t_user.actual-data-nodes=ds_$->{0..1}.t_user_$->{0..1}
spring.shardingsphere.rules.sharding.tables.t_user.table-strategy.standard.sharding-column=user_id
spring.shardingsphere.rules.sharding.tables.t_user.table-strategy.standard.sharding-algorithm-name=user-table-strategy-inline

spring.shardingsphere.rules.sharding.tables.t_user_detail.actual-data-nodes=ds_$->{0..1}.t_user_detail_$->{0..1}
spring.shardingsphere.rules.sharding.tables.t_user_detail.table-strategy.standard.sharding-column=user_id
spring.shardingsphere.rules.sharding.tables.t_user_detail.table-strategy.standard.sharding-algorithm-name=user-detail-table-strategy-inline

# 数据加密配置
# `t_user` 使用分片规则配置的逻辑表名称
spring.shardingsphere.rules.encrypt.tables.t_user.columns.username.cipher-column=username
spring.shardingsphere.rules.encrypt.tables.t_user.columns.username.encryptor-name=name-encryptor
spring.shardingsphere.rules.encrypt.tables.t_user.columns.pwd.cipher-column=pwd
spring.shardingsphere.rules.encrypt.tables.t_user.columns.pwd.encryptor-name=pwd-encryptor

# 数据加密算法配置
spring.shardingsphere.rules.encrypt.encryptors.name-encryptor.type=AES
spring.shardingsphere.rules.encrypt.encryptors.name-encryptor.props.aes-key-value=123456abc
spring.shardingsphere.rules.encrypt.encryptors.pwd-encryptor.type=AES
spring.shardingsphere.rules.encrypt.encryptors.pwd-encryptor.props.aes-key-value=123456abc

# 分布式序列策略配置
spring.shardingsphere.rules.sharding.tables.t_user.key-generate-strategy.column=user_id
spring.shardingsphere.rules.sharding.tables.t_user.key-generate-strategy.key-generator-name=snowflake

# 分片算法配置
spring.shardingsphere.rules.sharding.sharding-algorithms.default-database-strategy-inline.type=INLINE
```

```
# 表达式 `ds_$->{user_id % 2}` 枚举的数据源为读写分离配置的逻辑数据源名称
spring.shardingsphere.rules.sharding.sharding-algorithms.default-database-strategy-
inline.algorithm-expression=ds_$->{user_id % 2}
spring.shardingsphere.rules.sharding.sharding-algorithms.user-table-strategy-
inline.type=INLINE
spring.shardingsphere.rules.sharding.sharding-algorithms.user-table-strategy-
inline.algorithm-expression=t_user_$->{user_id % 2}

spring.shardingsphere.rules.sharding.sharding-algorithms.user-detail-table-
strategy-inline.type=INLINE
spring.shardingsphere.rules.sharding.sharding-algorithms.user-detail-table-
strategy-inline.algorithm-expression=t_user_detail_$->{user_id % 2}

# 分布式序列算法配置
spring.shardingsphere.rules.sharding.key-generators.snowflake.type=SNOWFLAKE

# 读写分离策略配置
# ds_0,ds_1 为读写分离配置的逻辑数据源名称
spring.shardingsphere.rules.readwrite-splitting.data-sources.ds_0.type=Static
spring.shardingsphere.rules.readwrite-splitting.data-sources.ds_0.props.write-data-
source-name=write-ds0
spring.shardingsphere.rules.readwrite-splitting.data-sources.ds_0.props.read-data-
source-names=write-ds0-read0
spring.shardingsphere.rules.readwrite-splitting.data-sources.ds_0.load-balancer-
name=read-random
spring.shardingsphere.rules.readwrite-splitting.data-sources.ds_1.type=Static
spring.shardingsphere.rules.readwrite-splitting.data-sources.ds_1.props.write-data-
source-name=write-ds1
spring.shardingsphere.rules.readwrite-splitting.data-sources.ds_1.props.read-data-
source-names=write-ds1-read0
spring.shardingsphere.rules.readwrite-splitting.data-sources.ds_1.load-balancer-
name=read-random

# 负载均衡算法配置
spring.shardingsphere.rules.readwrite-splitting.load-balancers.read-random.
type=RANDOM
```

SQL 解析

配置项说明

```
spring.shardingsphere.rules.sql-parser.sql-comment-parse-enabled= # 是否解析 SQL 注释

spring.shardingsphere.rules.sql-parser.sql-statement-cache.initial-capacity= # SQL
语句本地缓存初始容量
spring.shardingsphere.rules.sql-parser.sql-statement-cache.maximum-size= # SQL 语句
本地缓存最大容量
```

```
spring.shardingsphere.rules.sql-parser.sql-statement-cache.concurrency-level= # SQL 语句本地缓存并发级别，最多允许线程并发更新的个数

spring.shardingsphere.rules.sql-parser.parse-tree-cache.initial-capacity= # 解析树本地缓存初始容量
spring.shardingsphere.rules.sql-parser.parse-tree-cache.maximum-size= # 解析树本地缓存最大容量
spring.shardingsphere.rules.sql-parser.parse-tree-cache.concurrency-level= # 解析树本地缓存并发级别，最多允许线程并发更新的个数
```

7.1.4 Spring 命名空间

简介

ShardingSphere-JDBC 提供官方的 Spring 命名空间，使开发者可以非常便捷的整合 ShardingSphere-JDBC 和 Spring。

使用步骤

引入 Maven 依赖

```
<dependency>
    <groupId>org.apache.shardingsphere</groupId>
    <artifactId>shardingsphere-jdbc-core-spring-namespace</artifactId>
    <version>${shardingsphere.version}</version>
</dependency>
```

配置 Spring Bean

配置项说明

命名空间：http://shardingsphere.apache.org/schema/shardingsphere/datasource/datasource-5.0.0.xsd

<shardingsphere:data-source />

名称	类型	说明
id	属性	Spring Bean Id
schema-name (?)	属性	JDBC 数据源别名
data-source-names	标签	数据源名称，多个数据源以逗号分隔
rule-refs	标签	规则名称，多个规则以逗号分隔
mode (?)	标签	运行模式配置
props (?)	标签	属性配置，详情请参见属性配置

配置示例

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:shardingsphere="http://shardingsphere.apache.org/schema/
shardingsphere/datasource"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-
beans.xsd
                           http://shardingsphere.apache.org/schema/shardingsphere/
datasource
                           http://shardingsphere.apache.org/schema/shardingsphere/
datasource/datasource.xsd
                           ">
    <shardingsphere:data-source id="ds" schema-name="foo_schema" data-source-names=
"..." rule-refs="...">
        <shardingsphere:mode type="..." />
        <props>
            <prop key="xxx.xxx">${xxx.xxx}</prop>
        </props>
    </shardingsphere:data-source>
</beans>
```

在 Spring 中使用 ShardingSphere 数据源

使用方式同 Spring Boot Starter。

模式配置

配置项说明

命名空间：http://shardingsphere.apache.org/schema/shardingsphere/datasource/datasource-5.1.0.xsd

<shardingsphere:mode />

名称	类型	说明	默认值
type	属性	运行模式类型。可选配置：Memory 、Standalone、Cluster	
repository-ref (?)	属性	持久化仓库 Bean 引用。Memory 类型无需持久化	
overwrite (?)	属性	是否使用本地配置覆盖持久化配置	false

内存模式

缺省配置。

配置示例

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:shardingsphere="http://shardingsphere.apache.org/schema/
shardingsphere/datasource"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-
beans.xsd
                           http://shardingsphere.apache.org/schema/shardingsphere/
datasource
                           http://shardingsphere.apache.org/schema/shardingsphere/
datasource/datasource.xsd">

    <shardingsphere:data-source id="ds" schema-name="foo_schema" data-source-names=
"..." rule-refs="..." />
</beans>
```

单机模式

配置项说明

命名空间：http://shardingsphere.apache.org/schema/shardingsphere/mode-repository/standalone/repository-5.0.0.xsd

名称	类型	说明
id	属性	持久化仓库 Bean 名称
type	属性	持久化仓库类型
props (?)	标签	持久化仓库所需属性

配置示例

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:shardingsphere="http://shardingsphere.apache.org/schema/
shardingsphere/datasource"
       xmlns:standalone="http://shardingsphere.apache.org/schema/shardingsphere/
mode-repository/standalone"
```

```
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-
beans.xsd
                           http://shardingsphere.apache.org/schema/shardingsphere/
datasource
                           http://shardingsphere.apache.org/schema/shardingsphere/
datasource/datasource.xsd
                           http://shardingsphere.apache.org/schema/shardingsphere/
mode-repository/standalone
                           http://shardingsphere.apache.org/schema/shardingsphere/
mode-repository/standalone/repository.xsd">
    <standalone:repository id="standaloneRepository" type="File">
        <props>
            <prop key="path">target</prop>
        </props>
    </standalone:repository>

    <shardingsphere:data-source id="ds" schema-name="foo_schema" data-source-names=
"..." rule-refs="..." >
        <shardingsphere:mode type="Standalone" repository-ref="standaloneRepository
" overwrite="true" />
    </shardingsphere:data-source>
</beans>
```

集群模式

配置项说明

命名空间：http://shardingsphere.apache.org/schema/shardingsphere/mode-repository/cluster/repository-5.0.0.xsd

名称	类型	说明
id	属性	持久化仓库 Bean 名称
type	属性	持久化仓库类型
namespace	属性	注册中心命名空间
server-lists	属性	注册中心连接地址
props (?)	标签	持久化仓库所需属性

配置示例

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:shardingsphere="http://shardingsphere.apache.org/schema/
shardingsphere/datasource"
       xmlns:cluster="http://shardingsphere.apache.org/schema/shardingsphere/mode-
repository/cluster"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-
beans.xsd
                           http://shardingsphere.apache.org/schema/shardingsphere/
datasource
                           http://shardingsphere.apache.org/schema/shardingsphere/
datasource/datasource.xsd
                           http://shardingsphere.apache.org/schema/shardingsphere/
mode-repository/cluster
                           http://shardingsphere.apache.org/schema/shardingsphere/
mode-repository/cluster/repository.xsd">
    <cluster:repository id="clusterRepository" type="Zookeeper" namespace=
"regCenter" server-lists="localhost:3182">
        <props>
            <prop key="max-retries">3</prop>
            <prop key="operation-timeout-milliseconds">1000</prop>
        </props>
    </cluster:repository>

    <shardingsphere:data-source id="ds" schema-name="foo_schema" data-source-names=
"..." rule-refs="...">
        <shardingsphere:mode type="Cluster" repository-ref="clusterRepository"
overwrite="true" />
    </shardingsphere:data-source>
</beans>
```

持久化仓库类型的详情，请参见内置持久化仓库类型列表。

数据源配置

任何配置成为 Spring Bean 的数据源对象即可与 ShardingSphere-JDBC 的 Spring 命名空间配合使用。

配置示例

示例的数据库驱动为 MySQL，连接池为 HikariCP，可以更换为其他数据库驱动和连接池。

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:shardingsphere="http://shardingsphere.apache.org/schema/
shardingsphere/datasource"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-
beans.xsd
                           http://shardingsphere.apache.org/schema/shardingsphere/
datasource
                           http://shardingsphere.apache.org/schema/shardingsphere/
datasource/datasource.xsd
                           ">
    <bean id="ds1" class="com.zaxxer.hikari.HikariDataSource" destroy-method="close
">
        <property name="driverClassName" value="com.mysql.jdbc.Driver" />
        <property name="jdbcUrl" value="jdbc:mysql://localhost:3306/ds1" />
        <property name="username" value="root" />
        <property name="password" value="" />
    </bean>

    <bean id="ds2" class="com.zaxxer.hikari.HikariDataSource" destroy-method="close
">
        <property name="driverClassName" value="com.mysql.jdbc.Driver" />
        <property name="jdbcUrl" value="jdbc:mysql://localhost:3306/ds2" />
        <property name="username" value="root" />
        <property name="password" value="" />
    </bean>

    <shardingsphere:data-source id="ds" schema-name="foo_schema" data-source-names=
"ds1,ds2" rule-refs="..." />
</beans>
```

规则配置

规则是 Apache ShardingSphere 面向可插拔的一部分。本章节是 ShardingSphere-JDBC 的 Spring 命名空间规则配置参考手册。

数据分片

配置项说明

命名空间：http://shardingsphere.apache.org/schema/shardingsphere/sharding/sharding-5.1.0.xsd

<sharding:rule />

名称	类型	说明
id	属性	Spring Bean Id
table-rules (?)	标签	分片表规则配置
auto-table-rules (?)	标签	自动化分片表规则配置
binding-table-rules (?)	标签	绑定表规则配置
broadcast-table-rules (?)	标签	广播表规则配置
default-database-strategy-ref (?)	属性	默认分库策略名称
default-table-strategy-ref (?)	属性	默认分表策略名称
default-key-generate-strategy-ref (?)	属性	默认分布式序列策略名称
default-sharding-column (?)	属性	默认分片列名称

<sharding:table-rule />

名称	• 类型 *	说明
logic-table	属性	逻辑表名称
actual-data-nodes	属性	由数据源名 + 表名组成，以小数点分隔。多个表以逗号分隔，支持 inline 表达式。缺省表示使用已知数据源与逻辑表名称生成数据节点，用于广播表（即每个库中都需要一个同样的表用于关联查询，多为字典表）或只分库不分表且所有库的表结构完全一致的情况
actual-data-sources	属性	自动分片表数据源名
database-strategy-ref	属性	标准分片表分库策略名称
table-strategy-ref	属性	标准分片表分表策略名称
sharding-strategy-ref	属性	自动分片表策略名称
key-generate-strategy-ref	属性	分布式序列策略名称

<sharding:binding-table-rules />

名称	类型	说明
binding-table-rule (+)	标签	绑定表规则配置

<sharding:binding-table-rule />

名称	类型	说明
logic-tables	属性	绑定表名称，多个表以逗号分隔

<sharding:broadcast-table-rules />

名称	类型	说明
broadcast-table-rule (+)	标签	广播表规则配置

<sharding:broadcast-table-rule />

名称	类型	说明
table	属性	广播表名称

<sharding:standard-strategy />

名称	类型	说明
id	属性	标准分片策略名称
sharding-column	属性	分片列名称
algorithm-ref	属性	分片算法名称

<sharding:complex-strategy />

名称	类型	说明
id	属性	复合分片策略名称
sharding-columns	属性	分片列名称，多个列以逗号分隔
algorithm-ref	属性	分片算法名称

<sharding:hint-strategy />

名称	类型	说明
id	属性	Hint 分片策略名称
algorithm-ref	属性	分片算法名称

<sharding:none-strategy />

名称	类型	说明
id	属性	分片策略名称

<sharding:key-generate-strategy />

名称	类型	说明
id	属性	分布式序列策略名称
column	属性	分布式序列列名称
algorithm-ref	属性	分布式序列算法名称

<sharding:sharding-algorithm />

名称	类型	说明
id	属性	分片算法名称
type	属性	分片算法类型
props (?)	标签	分片算法属性配置

<sharding:key-generate-algorithm />

名称	类型	说明
id	属性	分布式序列算法名称
type	属性	分布式序列算法类型
props (?)	标签	分布式序列算法属性配置

算法类型的详情，请参见内置分片算法列表和内置分布式序列算法列表。

注意事项

行表达式标识符可以使用 `${...}` 或 `$->{...}`，但前者与 Spring 本身的属性文件占位符冲突，因此在 Spring 环境中使用行表达式标识符建议使用 `$->{...}`。

读写分离

配置项说明

命名空间：http://shardingsphere.apache.org/schema/shardingsphere/readwrite-splitting/readwrite-splitting-5.1.0.xsd

<readwrite-splitting:rule />

名称	类型	说明
id	属性	Spring Bean Id
data-source-rule (+)	标签	读写分离数据源规则配置

<readwrite-splitting:data-source-rule />

<readwrite-splitting:load-balance-algorithm />

名称	类型	说明
id	属性	负载均衡算法名称
type	属性	负载均衡算法类型
props (?)	标签	负载均衡算法属性配置

算法类型的详情，请参见内置负载均衡算法列表。查询一致性路由的详情，请参见使用规范。

高可用

配置项说明

命名空间：http://shardingsphere.apache.org/schema/shardingsphere/database-discovery/database-discovery-5.1.0.xsd

<database-discovery:rule />

名称	类型	说明
id	属性	Spring Bean Id
data-source-rule (+)	标签	数据源规则配置
discovery-heartbeat (+)	标签	检测心跳规则配置

<database-discovery:data-source-rule />

名称	类型	说明
id	属性	数据源规则名称
data-source-names	属性	数据源名称，多个数据源用逗号分隔如：ds_0, ds_1
discovery-heartbeat-name	属性	检测心跳名称
discovery-type-name	属性	数据库发现类型名称

<database-discovery:discovery-heartbeat />

名称	类型	说明
id	属性	监听心跳名称
props	标签	监听心跳属性配置，keep-alive-cron 属性配置 cron 表达式，如：‘0/5 * * * * ?’

<database-discovery:discovery-type />

名称	类型	说明
id	属性	数据库发现类型名称
type	属性	数据库发现类型，如：MGR、openGauss
props (?)	标签	数据库发现类型配置，如 MGR 的 group-name 属性配置

数据加密

配置项说明

命名空间：http://shardingsphere.apache.org/schema/shardingsphere/encrypt/encrypt-5.1.0.xsd

<encrypt:rule />

名称	类型	说明	默认值
id	属性	Spring Bean Id	
queryWithCipherColumn (?)	属性	是否使用加密列进行查询。在有原文列的情况下，可以使用原文列进行查询	true
table (+)	标签	加密表配置	

<encrypt:table />

名称	类型	说明
name	属性	加密表名称
column (+)	标签	加密列配置
query-with-cipher-column(?)	属性	该表是否使用加密列进行查询。在有原文列的情况下，可以使用原文列进行查询

<encrypt:column />

名称	类型	说明
logic-column	属性	加密列逻辑名称
cipher-column	属性	加密列名称
assisted-query-column (?)	属性	查询辅助列名称
plain-column (?)	属性	原文列名称
encrypt-algorithm-ref	属性	加密算法名称

<encrypt:encrypt-algorithm />

名称	类型	说明
id	属性	加密算法名称
type	属性	加密算法类型
props (?)	标签	加密算法属性配置

算法类型的详情，请参见内置加密算法列表。

影子库

配置项说明

命名空间：http://shardingsphere.apache.org/schema/shardingsphere/shadow/shadow-5.1.0.xsd

<shadow:rule />

名称	类型	说明
id	属性	Spring Bean Id
data-source(?)	标签	影子数据源配置
default-shadow-algorithm-name(?)	标签	默认影子算法配置
shadow-table(?)	标签	影子表配置

<shadow:data-source />

名称	类型	说明
id	属性	Spring Bean Id
source-data-source-name	属性	生产数据源名称
shadow-data-source-name	属性	影子数据源名称

<shadow:default-shadow-algorithm-name />

名称	类型	说明
name	属性	默认影子算法名称

<shadow:shadow-table />

名称	类型	说明
name	属性	影子表名称
data-sources	属性	影子表关联影子数据源名称列表（多个值用”,“隔开）
algorithm (?)	标签	影子表关联影子算法配置

<shadow:algorithm />

名称	类型	说明
shadow-algorithm-ref	属性	影子表关联影子算法名称

<shadow:shadow-algorithm />

名称	类型	说明
id	属性	影子算法名称
type	属性	影子算法类型
props (?)	标签	影子算法属性配置

SQL 解析

配置项说明

命名空间：http://shardingsphere.apache.org/schema/shardingsphere/sql-parser/sql-parser-5.1.0.xsd

<sql-parser:rule />

名称	类型	说明
id	属性	Spring Bean Id
sql-comment-parse-enable	属性	是否解析 SQL 注释
parse-tree-cache-ref	属性	解析树本地缓存名称
sql-statement-cache-ref	属性	SQL 语句本地缓存名称

<sql-parser:cache-option />

名称	类型	说明
id	属性	本地缓存配置项名称
initial-capacity	属性	本地缓存初始容量
maximum-size	属性	本地缓存最大容量
concurrency-level	属性	本地缓存并发级别，最多允许线程并发更新的个数

混合规则

混合配置的规则项之间的叠加使用是通过数据源名称和表名称关联的。

如果前一个规则是面向数据源聚合的，下一个规则在配置数据源时，则需要使用前一个规则配置的聚合后的逻辑数据源名称；同理，如果前一个规则是面向表聚合的，下一个规则在配置表时，则需要使用前一个规则配置的聚合后的逻辑表名称。

配置项说明

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:shardingsphere="http://shardingsphere.apache.org/schema/
shardingsphere/datasource"
       xmlns:readwrite-splitting="http://shardingsphere.apache.org/schema/
shardingsphere/readwrite-splitting"
       xmlns:encrypt="http://shardingsphere.apache.org/schema/shardingsphere/
encrypt"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-
beans.xsd
                           http://shardingsphere.apache.org/schema/shardingsphere/
datasource
```

```
                            http://shardingsphere.apache.org/schema/shardingsphere/
datasource/datasource.xsd
                            http://shardingsphere.apache.org/schema/shardingsphere/
readwrite-splitting
                            http://shardingsphere.apache.org/schema/shardingsphere/
readwrite-splitting/readwrite-splitting.xsd
                            http://shardingsphere.apache.org/schema/shardingsphere/
encrypt
                            http://shardingsphere.apache.org/schema/shardingsphere/
encrypt/encrypt.xsd
                            ">
    <bean id="write_ds0" class="  com.zaxxer.hikari.HikariDataSource" init-method=
"init" destroy-method="close">
        <property name="driverClassName" value="com.mysql.jdbc.Driver" />
        <property name="jdbcUrl" value="jdbc:mysql://localhost:3306/write_ds?
useSSL=false&amp;useUnicode=true&amp;characterEncoding=UTF-8" />
        <property name="username" value="root" />
        <property name="password" value="" />
    </bean>

    <bean id="read_ds0_0" class="  com.zaxxer.hikari.HikariDataSource" init-method=
"init" destroy-method="close">
        <!-- 省略详细数据源配置详情 -->
    </bean>

    <bean id="read_ds0_1" class="  com.zaxxer.hikari.HikariDataSource" init-method=
"init" destroy-method="close">
        <!-- 省略详细数据源配置详情 -->
    </bean>

    <bean id="write_ds1" class="  com.zaxxer.hikari.HikariDataSource" init-method=
"init" destroy-method="close">
        <!-- 省略详细数据源配置详情 -->
    </bean>

    <bean id="read_ds1_0" class="  com.zaxxer.hikari.HikariDataSource" init-method=
"init" destroy-method="close">
        <!-- 省略详细数据源配置详情 -->
    </bean>

    <bean id="read_ds1_1" class="  com.zaxxer.hikari.HikariDataSource" init-method=
"init" destroy-method="close">
        <!-- 省略详细数据源配置详情 -->
    </bean>

    <!-- 主从配置负载均衡策略 -->
    <readwrite-splitting:load-balance-algorithm id="randomStrategy" type="RANDOM" /
>
```

```
    <!-- 主从规则配置 -->
    <readwrite-splitting:rule id="readWriteSplittingRule">
        <readwrite-splitting:data-source-rule id="ds_0" type="Static" load-balance-
algorithm-ref="randomStrategy">
            <props>
                <prop key="write-data-source-name">write_ds0</prop>
                <prop key="read-data-source-names">read_ds0_0, read_ds0_1</prop>
            </props>
        </readwrite-splitting:data-source-rule>
        <readwrite-splitting:data-source-rule id="ds_1" type="Static" load-balance-
algorithm-ref="randomStrategy">
            <props>
                <prop key="write-data-source-name">write_ds1</prop>
                <prop key="read-data-source-names">read_ds1_0, read_ds1_1</prop>
            </props>
        </readwrite-splitting:data-source-rule>
    </readwrite-splitting:rule>

    <!-- 分片策略配置 -->
    <sharding:standard-strategy id="databaseStrategy" sharding-column="user_id"
algorithm-ref="inlineDatabaseStrategyAlgorithm" />
    <sharding:standard-strategy id="orderTableStrategy" sharding-column="order_id"
algorithm-ref="inlineOrderTableStrategyAlgorithm" />
    <sharding:standard-strategy id="orderItemTableStrategy" sharding-column="order_
item_id" algorithm-ref="inlineOrderItemTableStrategyAlgorithm" />

    <sharding:sharding-algorithm id="inlineDatabaseStrategyAlgorithm" type="INLINE
">
        <props>
            <!-- 表达式枚举的数据源名称为主从配置的逻辑数据源名称  -->
            <prop key="algorithm-expression">ds_${user_id % 2}</prop>
        </props>
    </sharding:sharding-algorithm>
    <sharding:sharding-algorithm id="inlineOrderTableStrategyAlgorithm" type=
"INLINE">
        <props>
            <prop key="algorithm-expression">t_order_${order_id % 2}</prop>
        </props>
    </sharding:sharding-algorithm>
    <sharding:sharding-algorithm id="inlineOrderItemTableStrategyAlgorithm" type=
"INLINE">
        <props>
            <prop key="algorithm-expression">t_order_item_${order_item_id % 2}</
prop>
        </props>
    </sharding:sharding-algorithm>
```

```
    <!-- 分片规则配置 -->
    <sharding:rule id="shardingRule">
        <sharding:table-rules>
            <!-- 表达式 ds_${0..1} 枚举的数据源名称为主从配置的逻辑数据源名称  -->
            <sharding:table-rule logic-table="t_order" actual-data-nodes="ds_${0..
1}.t_order_${0..1}" database-strategy-ref="databaseStrategy" table-strategy-ref=
"orderTableStrategy" key-generate-strategy-ref="orderKeyGenerator"/>
            <sharding:table-rule logic-table="t_order_item" actual-data-nodes="ds_$
{0..1}.t_order_item_${0..1}" database-strategy-ref="databaseStrategy" table-
strategy-ref="orderItemTableStrategy" key-generate-strategy-ref="itemKeyGenerator"/
>
        </sharding:table-rules>
        <sharding:binding-table-rules>
            <sharding:binding-table-rule logic-tables="t_order, t_order_item"/>
        </sharding:binding-table-rules>
        <sharding:broadcast-table-rules>
            <sharding:broadcast-table-rule table="t_address"/>
        </sharding:broadcast-table-rules>
    </sharding:rule>

    <!-- 数据加密规则配置 -->
    <encrypt:encrypt-algorithm id="name_encryptor" type="AES">
        <props>
            <prop key="aes-key-value">123456</prop>
        </props>
    </encrypt:encrypt-algorithm>
    <encrypt:encrypt-algorithm id="pwd_encryptor" type="assistedTest" />

    <encrypt:rule id="encryptRule">
        <encrypt:table name="t_user">
            <encrypt:column logic-column="username" cipher-column="username" plain-
column="username_plain" encrypt-algorithm-ref="name_encryptor" />
            <encrypt:column logic-column="pwd" cipher-column="pwd" assisted-query-
column="assisted_query_pwd" encrypt-algorithm-ref="pwd_encryptor" />
        </encrypt:table>
    </encrypt:rule>

    <!-- 数据源配置 -->
    <!-- data-source-names 数据源名称为所有的数据源节点名称 -->
    <shardingsphere:data-source id="readQueryDataSource" data-source-names="write_
ds0, read_ds0_0, read_ds0_1, write_ds1, read_ds1_0, read_ds1_1"
        rule-refs="readWriteSplittingRule, shardingRule, encryptRule" >
        <props>
            <prop key="sql-show">true</prop>
        </props>
    </shardingsphere:data-source>
</beans>
```

7.1.5 属性配置

Apache ShardingSphere 提供属性配置的方式配置系统级配置。

配置项说明

7.1.6 内置算法

简介

Apache ShardingSphere 通过 SPI 方式允许开发者扩展算法；与此同时，Apache ShardingSphere 也提供了大量的内置算法以便于开发者使用。

使用方式

内置算法均通过 type 和 props 进行配置，其中 type 由算法定义在 SPI 中，props 用于传递算法的个性化参数配置。

无论使用哪种配置方式，均是将配置完毕的算法命名，并传递至相应的规则配置中。本章节根据功能区分并罗列 Apache ShardingSphere 全部的内置算法，供开发者参考。

元数据持久化仓库

文件持久化

类型：File

适用模式：Standalone

可配置属性：

名称	数据类型	说明	默认值
path	String	元数据存储路径	.shardingsphere

ZooKeeper 持久化

类型：ZooKeeper

适用模式：Cluster

可配置属性：

名称	数据类型	说明	默认值
retryIntervalMilliseconds	int	重试间隔毫秒数	500
maxRetries	int	客户端连接最大重试次数	3
timeToLiveSeconds	int	临时数据失效的秒数	60
operationTimeoutMilliseconds	int	客户端操作超时的毫秒数	500
digest	String	登录认证密码	

Etcd 持久化

类型：Etcd

适用模式：Cluster

可配置属性：

名称	数据类型	说明	默认值
timeToLiveSeconds	long	临时数据失效的秒数	30
connectionTimeout	long	连接超时秒数	30

分片算法

自动分片算法

取模分片算法

类型：MOD

可配置属性：

属性名称	数据类型	说明
sharding-count	int	分片数量

哈希取模分片算法

类型：HASH_MOD

可配置属性：

属性名称	数据类型	说明
sharding-count	int	分片数量

基于分片容量的范围分片算法

类型：VOLUME_RANGE

可配置属性：

属性名称	数据类型	说明
range-lower	long	范围下界，超过边界的数据会报错
range-upper	long	范围上界，超过边界的数据会报错
sharding-volume	long	分片容量

基于分片边界的范围分片算法

类型：BOUNDARY_RANGE

可配置属性：

属性名称	数据类型	说明
sharding-ranges	String	分片的范围边界，多个范围边界以逗号分隔

自动时间段分片算法

类型：AUTO_INTERVAL

可配置属性：

属性名称	• 数据类型 *	说明
datetim e-lower	St ri ng	分片的起始时间范围，时间戳格式：yyyy-MM-dd HH:mm:ss
datetim e-upper	St ri ng	分片的结束时间范围，时间戳格式：yyyy-MM-dd HH:mm:ss
sh arding- seconds	lo ng	单一分片所能承载的最大时间，单位：秒，允许分片键的时间戳格式的秒带有时间精度，但秒后的时间精度会被自动抹去

标准分片算法

Apache ShardingSphere 内置的标准分片算法实现类包括：

行表达式分片算法

使用 Groovy 的表达式，提供对 SQL 语句中的 = 和 `IN` 的分片操作支持，只支持单分片键。对于简单的分片算法，可以通过简单的配置使用，从而避免繁琐的 Java 代码开发，如: `t_user_$->{u_id % 8}` 表示 `t_user` 表根据 `u_id` 模 8，而分成 8 张表，表名称为 `t_user_0` 到 `t_user_7`。详情请参见行表达式。

类型：INLINE

可配置属性：

时间范围分片算法

类型：INTERVAL

可配置属性：

复合分片算法

复合行表达式分片算法

详情请参见行表达式。

类型：COMPLEX_INLINE

Hint 分片算法

Hint 行表达式分片算法

详情请参见行表达式。

类型：HINT_INLINE

属性名称	数据类型	说明	默认值
algorithm-expression (?)	String	分片算法的行表达式	${value}

自定义类分片算法

通过配置分片策略类型和算法类名，实现自定义扩展。

类型：CLASS_BASED

可配置属性：

属性名称	数据类型	说明
strategy	String	分片策略类型，支持 STANDARD、COMPLEX 或 HINT（不区分大小写）
algorithmClassName	String	分片算法全限定名

分布式序列算法

雪花算法

类型：SNOWFLAKE

可配置属性：

UUID

类型：UUID

可配置属性：无

负载均衡算法

轮询算法

类型：ROUND_ROBIN

可配置属性：无

随机访问算法

类型：RANDOM

可配置属性：无

权重访问算法

类型：WEIGHT

可配置属性：

使用中的读库都必须配置权重

属性名称	数据类型	说明
• (+)	d ouble	属性名字使用读库名字，参数填写读库对应的权重值。权重参数范围最小值 > 0，合计 <= Double.MAX_VALUE。

加密算法

MD5 加密算法

类型：MD5

可配置属性：无

AES 加密算法

类型：AES

可配置属性：

名称	数据类型	说明
aes-key-value	String	AES 使用的 KEY

RC4 加密算法

类型：RC4

可配置属性：

名称	数据类型	说明
rc4-key-value	String	RC4 使用的 KEY

SM3 加密算法

类型：SM3

可配置属性：

名称	数据类型	说明
sm3-salt	String	SM3 使用的 SALT（空或 8 Bytes）

SM4 加密算法

类型：SM4

可配置属性：

名称	数据类型	说明
sm4-key	String	SM4 使用的 KEY （16 Bytes）
sm4-mode	String	SM4 使用的 MODE （CBC 或 ECB）
sm4-iv	String	SM4 使用的 IV （MODE 为 CBC 时需指定，16 Bytes）
sm4-padding	String	SM4 使用的 PADDING （PKCS5Padding 或 PKCS7Padding，暂不支持 NoPadding）

影子算法

列影子算法

列值匹配影子算法

类型：VALUE_MATCH

可配置属性：

属性名称	数据类型	说明
column	String	影子列
operation	String	SQL 操作类型（INSERT, UPDATE, DELETE, SELECT）
value	String	影子列匹配的值

列正则表达式匹配影子算法

类型：REGEX_MATCH

可配置属性：

属性名称	数据类型	说明
column	String	匹配列
operation	String	SQL 操作类型（INSERT, UPDATE, DELETE, SELECT）
regex	String	影子列匹配正则表达式

Hint 影子算法

简单 Hint 匹配影子算法

类型：SIMPLE_HINT

可配置属性：

至少配置一组任意的键值对。比如：foo:bar

属性名称	数据类型	说明
foo	String	bar

7.1.7 特殊 API

本章节将介绍 ShardingSphere-JDBC 的特殊场景 API。

数据分片

本章节将介绍 ShardingSphere-JDBC 的分片场景 API。

强制路由

简介

Apache ShardingSphere 使用 ThreadLocal 管理分片键值进行强制路由。可以通过编程的方式向 HintManager 中添加分片值，该分片值仅在当前线程内生效。
Apache ShardingSphere 还可以通过 SQL 中增加注释的方式进行强制路由。

Hint 的主要使用场景：

- 分片字段不存在 SQL 和数据库表结构中，而存在于外部业务逻辑。
- 强制在主库进行某些数据操作。

- 强制在指定数据库进行某些数据操作。

使用方法

使用 Hint 分片

规则配置

Hint 分片算法需要用户实现 `org.apache.shardingsphere.sharding.api.sharding.hint.HintShardingAlgorithm` 接口。Apache ShardingSphere 在进行路由时，将会从 HintManager 中获取分片值进行路由操作。

参考配置如下：

```
rules:
- !SHARDING
  tables:
    t_order:
      actualDataNodes: demo_ds_${0..1}.t_order_${0..1}
      databaseStrategy:
        hint:
          algorithmClassName: xxx.xxx.xxx.HintXXXAlgorithm
      tableStrategy:
        hint:
          algorithmClassName: xxx.xxx.xxx.HintXXXAlgorithm
  defaultTableStrategy:
    none:
  defaultKeyGenerateStrategy:
    type: SNOWFLAKE
    column: order_id

props:
    sql-show: true
```

获取 HintManager

```
HintManager hintManager = HintManager.getInstance();
```

添加分片键值

- 使用 `hintManager.addDatabaseShardingValue` 来添加数据源分片键值。
- 使用 `hintManager.addTableShardingValue` 来添加表分片键值。

  分库不分表情况下，强制路由至某一个分库时，可使用 `hintManager.setDatabaseShardingValue` 方式添加分片。

清除分片键值

分片键值保存在 ThreadLocal 中，所以需要在操作结束时调用 `hintManager.close()` 来清除 ThreadLocal 中的内容。

hintManager 实现了 AutoCloseable 接口，可推荐使用 try with resource 自动关闭。

完整代码示例

```
// Sharding database and table with using HintManager
String sql = "SELECT * FROM t_order";
try (HintManager hintManager = HintManager.getInstance();
     Connection conn = dataSource.getConnection();
     PreparedStatement preparedStatement = conn.prepareStatement(sql)) {
    hintManager.addDatabaseShardingValue("t_order", 1);
    hintManager.addTableShardingValue("t_order", 2);
    try (ResultSet rs = preparedStatement.executeQuery()) {
        while (rs.next()) {
            // ...
        }
    }
}

// Sharding database without sharding table and routing to only one database with
using HintManager
String sql = "SELECT * FROM t_order";
try (HintManager hintManager = HintManager.getInstance();
     Connection conn = dataSource.getConnection();
     PreparedStatement preparedStatement = conn.prepareStatement(sql)) {
    hintManager.setDatabaseShardingValue(3);
    try (ResultSet rs = preparedStatement.executeQuery()) {
        while (rs.next()) {
            // ...
```

```
        }
    }
}
```

使用 Hint 强制主库路由

使用手动编程的方式

获取 HintManager

与基于 Hint 的数据分片相同。

设置主库路由

- 使用 `hintManager.setWriteRouteOnly` 设置主库路由。

清除分片键值

与基于 Hint 的数据分片相同。

完整代码示例

```
String sql = "SELECT * FROM t_order";
try (HintManager hintManager = HintManager.getInstance();
     Connection conn = dataSource.getConnection();
     PreparedStatement preparedStatement = conn.prepareStatement(sql)) {
    hintManager.setWriteRouteOnly();
    try (ResultSet rs = preparedStatement.executeQuery()) {
        while (rs.next()) {
            // ...
        }
    }
}
```

使用 SQL 注释的方式

使用规范

SQL Hint 功能需要用户提前开启解析注释的配置，设置 `sqlCommentParseEnabled` 为 `true`。注释格式暂时只支持 `/* */`，内容需要以 `ShardingSphere hint:` 开始，属性名为 `writeRouteOnly`。

完整示例

```
/* ShardingSphere hint: writeRouteOnly=true */
SELECT * FROM t_order;
```

使用 Hint 路由至指定数据库

使用手动编程的方式

获取 HintManager

与基于 Hint 的数据分片相同。

设置路由至指定数据库

- 使用 `hintManager.setDataSourceName` 设置数据库名称。

完整代码示例

```
String sql = "SELECT * FROM t_order";
try (HintManager hintManager = HintManager.getInstance();
     Connection conn = dataSource.getConnection();
     PreparedStatement preparedStatement = conn.prepareStatement(sql)) {
    hintManager.setDataSourceName("ds_0");
    try (ResultSet rs = preparedStatement.executeQuery()) {
        while (rs.next()) {
            // ...
        }
    }
}
```

使用 SQL 注释的方式

使用规范

SQL Hint 功能需要用户提前开启解析注释的配置，设置 `sqlCommentParseEnabled` 为 `true`，目前只支持路由至一个数据源。注释格式暂时只支持 `/* */`，内容需要以 `ShardingSphere hint:` 开始，属性名为 `dataSourceName`。

完整示例

```
/* ShardingSphere hint: dataSourceName=ds_0 */
SELECT * FROM t_order;
```

分布式事务

通过 Apache ShardingSphere 使用分布式事务，与本地事务并无区别。除了透明化分布式事务的使用之外，Apache ShardingSphere 还能够在每次数据库访问时切换分布式事务类型。支持的事务类型包括：本地事务、XA 事务和柔性事务。可在创建数据库连接之前设置，缺省为 Apache ShardingSphere 启动时的默认事务类型。

使用 Java API

引入 Maven 依赖

```
<dependency>
    <groupId>org.apache.shardingsphere</groupId>
    <artifactId>shardingsphere-jdbc-core</artifactId>
    <version>${shardingsphere.version}</version>
</dependency>

<!-- 使用 XA 事务时，需要引入此模块 -->
<dependency>
    <groupId>org.apache.shardingsphere</groupId>
    <artifactId>shardingsphere-transaction-xa-core</artifactId>
    <version>${shardingsphere.version}</version>
</dependency>

<!-- 使用 BASE 事务时，需要引入此模块 -->
<dependency>
    <groupId>org.apache.shardingsphere</groupId>
    <artifactId>shardingsphere-transaction-base-seata-at</artifactId>
    <version>${shardingsphere.version}</version>
</dependency>
```

使用分布式事务

```
TransactionTypeHolder.set(TransactionType.XA); // 支持 TransactionType.LOCAL,
TransactionType.XA, TransactionType.BASE
try (Connection conn = dataSource.getConnection()) { // 使用
ShardingSphereDataSource
    conn.setAutoCommit(false);
    PreparedStatement ps = conn.prepareStatement("INSERT INTO t_order (user_id,
status) VALUES (?, ?)");
```

```
    ps.setObject(1, 1000);
    ps.setObject(2, "init");
    ps.executeUpdate();
    conn.commit();
}
```

使用 Spring Boot Starter

引入 Maven 依赖

```
<dependency>
    <groupId>org.apache.shardingsphere</groupId>
    <artifactId>shardingsphere-jdbc-core-spring-boot-starter</artifactId>
    <version>${shardingsphere.version}</version>
</dependency>

<!-- 使用 XA 事务时，需要引入此模块 -->
<dependency>
    <groupId>org.apache.shardingsphere</groupId>
    <artifactId>shardingsphere-transaction-xa-core</artifactId>
    <version>${shardingsphere.version}</version>
</dependency>

<!-- 使用 BASE 事务时，需要引入此模块 -->
<dependency>
    <groupId>org.apache.shardingsphere</groupId>
    <artifactId>shardingsphere-transaction-base-seata-at</artifactId>
    <version>${shardingsphere.version}</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);
    }

    @Bean
    public JdbcTemplate jdbcTemplate(final DataSource dataSource) {
        return new JdbcTemplate(dataSource);
```

```
    }
}
```

使用分布式事务

```
@Transactional
@ShardingSphereTransactionType(TransactionType.XA)  // 支持 TransactionType.LOCAL,
TransactionType.XA, TransactionType.BASE
public void insert() {
    jdbcTemplate.execute("INSERT INTO t_order (user_id, status) VALUES (?, ?)",
(PreparedStatementCallback<Object>) ps -> {
        ps.setObject(1, i);
        ps.setObject(2, "init");
        ps.executeUpdate();
    });
}
```

使用 Spring 命名空间

引入 Maven 依赖

```
<dependency>
    <groupId>org.apache.shardingsphere</groupId>
    <artifactId>shardingsphere-jdbc-core-spring-namespace</artifactId>
    <version>${shardingsphere.version}</version>
</dependency>

<!-- 使用 XA 事务时，需要引入此模块 -->
<dependency>
    <groupId>org.apache.shardingsphere</groupId>
    <artifactId>shardingsphere-transaction-xa-core</artifactId>
    <version>${shardingsphere.version}</version>
</dependency>

<!-- 使用 BASE 事务时，需要引入此模块 -->
<dependency>
    <groupId>org.apache.shardingsphere</groupId>
    <artifactId>shardingsphere-transaction-base-seata-at</artifactId>
    <version>${shardingsphere.version}</version>
</dependency>
```

配置事务管理器

```
<!-- ShardingDataSource 的相关配置 -->
<!-- ...  -->

<bean id="transactionManager" class="org.springframework.jdbc.datasource.
DataSourceTransactionManager">
    <property name="dataSource" ref="shardingDataSource" />
</bean>
<bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
    <property name="dataSource" ref="shardingDataSource" />
</bean>
<tx:annotation-driven />

<!-- 开启自动扫描 @ShardingSphereTransactionType 注解，使用 Spring 原生的 AOP 在类和方法上
进行增强 -->
<sharding:tx-type-annotation-driven />
```

使用分布式事务

```
@Transactional
@ShardingSphereTransactionType(TransactionType.XA)  // 支持 TransactionType.LOCAL,
TransactionType.XA, TransactionType.BASE
public void insert() {
    jdbcTemplate.execute("INSERT INTO t_order (user_id, status) VALUES (?, ?)",
(PreparedStatementCallback<Object>) ps -> {
        ps.setObject(1, i);
        ps.setObject(2, "init");
        ps.executeUpdate();
    });
}
```

Atomikos 事务

Apache ShardingSphere 默认的 XA 事务管理器为 Atomikos。

数据恢复

在项目的 `logs` 目录中会生成 `xa_tx.log`, 这是 XA 崩溃恢复时所需的日志，请勿删除。

修改配置

可以通过在项目的 classpath 中添加 `jta.properties` 来定制化 Atomikos 配置项。

详情请参见 Atomikos 官方文档。

Narayana 事务

引入 Maven 依赖

```
<properties>
    <narayana.version>5.9.1.Final</narayana.version>
    <jboss-transaction-spi.version>7.6.0.Final</jboss-transaction-spi.version>
    <jboss-logging.version>3.2.1.Final</jboss-logging.version>
</properties>

<dependency>
    <groupId>org.apache.shardingsphere</groupId>
    <artifactId>shardingsphere-jdbc-core</artifactId>
    <version>${shardingsphere.version}</version>
</dependency>

<!-- 使用 XA 事务时，需要引入此模块 -->
<dependency>
    <groupId>org.apache.shardingsphere</groupId>
    <artifactId>shardingsphere-transaction-xa-core</artifactId>
    <version>${shardingsphere.version}</version>
</dependency>

<dependency>
      <groupId>org.apache.shardingsphere</groupId>
      <artifactId>shardingsphere-transaction-xa-narayana</artifactId>
      <version>${shardingsphere.version}</version>
</dependency>
<dependency>
      <groupId>org.jboss.narayana.jta</groupId>
      <artifactId>jta</artifactId>
      <version>${narayana.version}</version>
</dependency>
<dependency>
       <groupId>org.jboss.narayana.jts</groupId>
       <artifactId>narayana-jts-integration</artifactId>
       <version>${narayana.version}</version>
</dependency>
<dependency>
       <groupId>org.jboss</groupId>
       <artifactId>jboss-transaction-spi</artifactId>
       <version>${jboss-transaction-spi.version}</version>
```

```
</dependency>
<dependency>
       <groupId>org.jboss.logging</groupId>
       <artifactId>jboss-logging</artifactId>
       <version>${jboss-logging.version}</version>
</dependency>
```

定制化配置项

可以通过在项目的 classpath 中添加 `jbossts-properties.xml` 来定制化 Narayana 配置项。

详情请参见 Narayana 官方文档。

设置 XA 事务管理类型

Yaml:

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

SpringBoot:

```
spring:
  shardingsphere:
    props:
      xa-transaction-manager-type: Narayana
```

Spring Namespace:

```
<shardingsphere:data-source id="xxx" data-source-names="xxx" rule-refs="xxx">
    <props>
        <prop key="xa-transaction-manager-type">Narayana</prop>
    </props>
</shardingsphere:data-source>
```

Bitronix 事务

引入 Maven 依赖

```
<properties>
    <btm.version>2.1.3</btm.version>
</properties>

<dependency>
```

```
    <groupId>org.apache.shardingsphere</groupId>
    <artifactId>shardingsphere-jdbc-core</artifactId>
    <version>${shardingsphere.version}</version>
</dependency>

<!-- 使用 XA 事务时，需要引入此模块 -->
<dependency>
    <groupId>org.apache.shardingsphere</groupId>
    <artifactId>shardingsphere-transaction-xa-core</artifactId>
    <version>${shardingsphere.version}</version>
</dependency>

<dependency>
    <groupId>org.apache.shardingsphere</groupId>
    <artifactId>shardingsphere-transaction-xa-bitronix</artifactId>
    <version>${shardingsphere.version}</version>
</dependency>

<dependency>
    <groupId>org.codehaus.btm</groupId>
    <artifactId>btm</artifactId>
    <version>${btm.version}</version>
</dependency>
```

定制化配置项

详情请参见 Bitronix 官方文档。

设置 XA 事务管理类型

Yaml:

```
- !TRANSACTION
  defaultType: XA
  providerType: Bitronix
```

SpringBoot:

```
spring:
  shardingsphere:
    props:
      xa-transaction-manager-type: Bitronix
```

Spring Namespace:

```
<shardingsphere:data-source id="xxx" data-source-names="xxx" rule-refs="xxx">
    <props>
```

```
        <prop key="xa-transaction-manager-type">Bitronix</prop>
    </props>
</shardingsphere:data-source>
```

Seata 事务

启动 Seata 服务

按照 seata-work-shop 中的步骤，下载并启动 Seata 服务器。

创建日志表

在每一个分片数据库实例中执创建 `undo_log` 表（以 MySQL 为例）。

```
CREATE TABLE IF NOT EXISTS `undo_log`
(
  `id`            BIGINT(20)   NOT NULL AUTO_INCREMENT COMMENT 'increment id',
  `branch_id`     BIGINT(20)   NOT NULL COMMENT 'branch transaction id',
  `xid`           VARCHAR(100) NOT NULL COMMENT 'global transaction id',
  `context`       VARCHAR(128) NOT NULL COMMENT 'undo_log context,such as
serialization',
  `rollback_info` LONGBLOB     NOT NULL COMMENT 'rollback info',
  `log_status`    INT(11)      NOT NULL COMMENT '0:normal status,1:defense status',
  `log_created`   DATETIME     NOT NULL COMMENT 'create datetime',
  `log_modified`  DATETIME     NOT NULL COMMENT 'modify datetime',
  PRIMARY KEY (`id`),
  UNIQUE KEY `ux_undo_log` (`xid`, `branch_id`)
) ENGINE = InnoDB
  AUTO_INCREMENT = 1
  DEFAULT CHARSET = utf8 COMMENT ='AT transaction mode undo table';
```

修改配置

在 classpath 中增加 `seata.conf` 文件。

```
client {
    application.id = example    ## 应用唯一主键
    transaction.service.group = my_test_tx_group   ## 所属事务组
}
```

根据实际场景修改 Seata 的 `file.conf` 和 `registry.conf` 文件。

可观察性

介绍如何使用可观察性探针和集成第三方应用。

使用探针

如何获取

本地构建

```
> cd shardingsphere/shardingsphere-agent
> mvn clean install
```

远程下载（暂未发布）

```
> weget http://xxxxx/shardingsphere-agent.tar.gz
> tar -zxvcf shardingsphere-agent.tar.gz
```

配置

找到 `agent.yaml` 文件:

```yaml
applicationName: shardingsphere-agent
ignoredPluginNames: # 忽略的插件集合
  - Opentracing
  - Jaeger
  - Zipkin
  - Prometheus
  - OpenTelemetry
  - Logging

plugins:
  Prometheus:
    host:  "localhost"
    port: 9090
    props:
      JVM_INFORMATION_COLLECTOR_ENABLED : "true"
  Jaeger:
    host: "localhost"
    port: 5775
    props:
      SERVICE_NAME: "shardingsphere-agent"
      JAEGER_SAMPLER_TYPE: "const"
      JAEGER_SAMPLER_PARAM: "1"
```

```
      JAEGER_REPORTER_LOG_SPANS: "true"
      JAEGER_REPORTER_FLUSH_INTERVAL: "1"
  Zipkin:
    host: "localhost"
    port: 9411
    props:
      SERVICE_NAME: "shardingsphere-agent"
      URL_VERSION: "/api/v2/spans"
  Opentracing:
    props:
      OPENTRACING_TRACER_CLASS_NAME: "org.apache.skywalking.apm.toolkit.
opentracing.SkywalkingTracer"
  OpenTelemetry:
    props:
      otel.resource.attributes: "service.name=shardingsphere-agent" # 多个配置用','分
隔
      otel.traces.exporter: "zipkin"
  Logging:
    props:
      LEVEL: "INFO"
```

启动

在启动脚本中添加参数:

```
-javaagent:\absolute path\shardingsphere-agent.jar
```

应用性能监控集成

使用方法

使用 OpenTracing 协议

- 方法 1：通过读取系统参数注入 APM 系统提供的 Tracer 实现类

启动时添加参数

```
-Dorg.apache.shardingsphere.tracing.opentracing.tracer.class=org.apache.skywalking.
apm.toolkit.opentracing.SkywalkingTracer
```

调用初始化方法

```
ShardingTracer.init();
```

- 方法 2：通过参数注入 APM 系统提供的 Tracer 实现类。

```
ShardingTracer.init(new SkywalkingTracer());
```

注意：使用 *SkyWalking* 的 *OpenTracing* 探针时，不能同时使用 *Apache ShardingSphere Agent* 的 *OpenTracing* 插件，以防止两种插件互相冲突。

使用 SkyWalking 自动探针

请参考 SkyWalking 部署手册。

使用 OpenTelemetry

在 agent.yaml 中填写好配置即可，例如将 Traces 数据导出到 Zipkin。

```
OpenTelemetry:
    props:
      otel.resource.attributes: "service.name=shardingsphere-agent"
      otel.traces.exporter: "zipkin"
      otel.exporter.zipkin.endpoint: "http://127.0.0.1:9411/api/v2/spans"
```

效果展示

无论使用哪种方式，都可以方便的将 APM 信息展示在对接的系统中，以下以 SkyWalking 为例。

应用架构

使用 ShardingSphere-Proxy 访问两个数据库 `192.168.0.1:3306` 和 `192.168.0.2:3306`，且每个数据库中有两个分表。

拓扑图展示

从图中看，用户访问 18 次 ShardingSphere-Proxy 应用，每次每个数据库访问了两次。这是由于每次访问涉及到每个库中的两个分表，所以每次访问了四张表。

跟踪数据展示

从跟踪图中可以能够看到 SQL 解析和执行的情况。

`/Sharding-Sphere/parseSQL/`: 表示本次 SQL 的解析性能。

`/Sharding-Sphere/executeSQL/`: 表示具体执行的实际 SQL 的性能。

图 1: 拓扑图

图 2: 跟踪图

图 3: 解析节点

图 4: 实际访问节点

异常情况展示

图 5: 异常跟踪图

从跟踪图中可以能够看到发生异常的节点。

`/Sharding-Sphere/executeSQL/`: 表示执行 SQL 异常的结果。

图 6: 异常节点

`/Sharding-Sphere/executeSQL/`: 表示执行 SQL 异常的日志。

图 7: 异常节点日志

7.1.8 不支持项

DataSource 接口

- 不支持 timeout 相关操作。

Connection 接口

- 不支持存储过程，函数，游标的操作；
- 不支持执行 native SQL；
- 不支持 savepoint 相关操作；
- 不支持 Schema/Catalog 的操作；
- 不支持自定义类型映射。

Statement 和 PreparedStatement 接口

- 不支持返回多结果集的语句（即存储过程，非 SELECT 多条数据）；
- 不支持国际化字符的操作。

ResultSet 接口

- 不支持对于结果集指针位置判断；
- 不支持通过非 next 方法改变结果指针位置；
- 不支持修改结果集内容；
- 不支持获取国际化字符；
- 不支持获取 Array。

JDBC 4.1

- 不支持 JDBC 4.1 接口新功能。

查询所有未支持方法，请阅读 `org.apache.shardingsphere.driver.jdbc.unsupported` 包。

7.2 ShardingSphere-Proxy

配置是 ShardingSphere-Proxy 中唯一与开发者交互的模块,通过它可以快速清晰的理解 ShardingSphere-Proxy 所提供的功能。

本章节是 ShardingSphere-Proxy 的配置参考手册，需要时可当做字典查阅。

ShardingSphere-Proxy 提供基于 YAML 的配置方式，并使用 DistSQL 进行交互。通过配置，应用开发者可以灵活的使用数据分片、读写分离、数据加密、影子库等功能，并且能够叠加使用。

规则配置部分与 ShardingSphere-JDBC 的 YAML 配置完全一致。DistSQL 与 YAML 配置能够相互取代。

更多使用细节请参见使用示例。

7.2.1 启动手册

本章节将介绍 ShardingSphere-Proxy 相关部署和启动等相关操作。

使用二进制发布包

启动步骤

1. 通过下载页面获取 ShardingSphere-Proxy 安装包。
2. 解压缩后修改 `conf/server.yaml` 和以 `config-` 前缀开头的文件，如：`conf/config-xxx.yaml` 文件，进行分片规则、读写分离规则配置。配置方式请参考配置手册。
3. Linux 操作系统请运行 `bin/start.sh`，Windows 操作系统请运行 `bin/start.bat` 启动 ShardingSphere-Proxy。如需配置启动端口、配置文件位置，可参考快速入门。

选择数据库协议

使用 PostgreSQL

1. 使用任何 PostgreSQL 的客户端连接。如: `psql -U root -h 127.0.0.1 -p 3307`。

使用 MySQL

1. 将 MySQL 的 JDBC 驱动程序复制至目录 `ext-lib/`。
2. 使用任何 MySQL 的客户端连接。如: `mysql -u root -h 127.0.0.1 -P 3307`。

使用 openGauss

1. 将以 `org.opengauss` 包名为前缀的 openGauss 的 JDBC 驱动程序复制至目录 `ext-lib/`。
2. 使用任何 openGauss 的客户端连接。如: `gsql -U root -h 127.0.0.1 -p 3307`。

选择元数据持久化仓库

使用 ZooKeeper

默认集成 ZooKeeper Curator 客户端。

使用 Etcd

1. 将 Etcd 的客户端驱动程序复制至目录 `ext-lib/`。

使用分布式事务

与 ShardingSphere-JDBC 使用方式相同。具体可参考分布式事务。

使用自定义算法

当用户需要使用自定义的算法类时，可通过以下方式配置使用自定义算法，以分片为例:

1. 实现 `ShardingAlgorithm` 接口定义的算法实现类。
2. 在项目 `resources` 目录下创建 `META-INF/services` 目录。
3. 在 `META-INF/services` 目录下新建文件 `org.apache.shardingsphere.sharding.spi.ShardingAlgorithm`
4. 将实现类的绝对路径写入至文件 `org.apache.shardingsphere.sharding.spi.ShardingAlgorithm`
5. 将上述 Java 文件打包成 jar 包。
6. 将上述 jar 包拷贝至 ShardingSphere-Proxy 解压后的 `ext-lib/` 目录。
7. 将上述自定义算法实现类的 Java 文件引用配置在 YAML 文件中，具体可参考配置规则。

注意事项

1. ShardingSphere-Proxy 默认使用 3307 端口，可以通过启动脚本追加参数作为启动端口号。如: `bin/start.sh 3308`。
2. ShardingSphere-Proxy 使用 `conf/server.yaml` 配置注册中心、认证信息以及公用属性。
3. ShardingSphere-Proxy 支持多逻辑数据源，每个以 `config-` 前缀命名的 YAML 配置文件，即为一个逻辑数据源。

使用 Docker

拉取官方 Docker 镜像

```
docker pull apache/shardingsphere-proxy
```

手动构建 Docker 镜像（可选）

```
git clone https://github.com/apache/shardingsphere
mvn clean install
cd shardingsphere-distribution/shardingsphere-proxy-distribution
mvn clean package -Prelease,docker
```

配置 ShardingSphere-Proxy

在 `/${your_work_dir}/conf/` 创建 `server.yaml` 和 `config-xxx.yaml` 文件，进行服务器和分片规则配置。配置规则，请参考配置手册。配置模板，请参考配置模板

运行 Docker

```
docker run -d -v /${your_work_dir}/conf:/opt/shardingsphere-proxy/conf -e PORT=3308
-p13308:3308 apache/shardingsphere-proxy:latest
```

说明

- 可以自定义端口 3308 和 13308。3308 表示 docker 容器端口，13308 表示宿主机端口。
- 必须挂载配置路径到 `/opt/shardingsphere-proxy/conf`。

```
docker run -d -v /${your_work_dir}/conf:/opt/shardingsphere-proxy/conf -e JVM_OPTS=
"-Djava.awt.headless=true" -e PORT=3308 -p13308:3308 apache/shardingsphere-
proxy:latest
```

说明

- 可以自定义 JVM 相关参数到环境变量 `JVM_OPTS` 中。

```
docker run -d -v /${your_work_dir}/conf:/opt/shardingsphere-proxy/conf -v /${your_work_dir}/ext-lib:/opt/shardingsphere-proxy/ext-lib -p13308:3308 apache/shardingsphere-proxy:latest
```

说明

- 如需使用外部 jar 包（例如 MySQL/openGauss JDBC 驱动、自定义算法等），可将其所在目录挂载到 `/opt/shardingsphere-proxy/ext-lib`。

访问 ShardingSphere-Proxy

与连接 PostgreSQL 的方式相同。

```
psql -U ${your_username} -h ${your_host} -p 13308
```

FAQ

问题 1：I/O exception (java.io.IOException) caught when processing request to {}->unix://localhost:80: Connection refused?

回答：在构建镜像前，请确保 docker daemon 进程已经运行。

问题 2：启动时报无法连接到数据库错误?

回答：请确保 `/${your_work_dir}/conf/config-xxx.yaml` 配置文件中指定的 PostgreSQL 数据库的 IP 可以被 Docker 容器内部访问到。

问题 3：如何使用后端数据库为 MySQL/openGauss 的 ShardingSphere-Proxy?

回答：将 `mysql-connector.jar` 或 `opengauss-jdbc.jar` 所在目录挂载到 `/opt/shardingsphere-proxy/ext-lib`。

问题 4：如何使用自定义分片算法?

回答：实现对应的分片算法接口，将编译出的分片算法 jar 所在目录挂载到 `/opt/shardingsphere-proxy/ext-lib`。

7.2.2 YAML 配置

ShardingSphere-JDBC 的 YAML 配置是 ShardingSphere-Proxy 的子集。在 `server.yaml` 文件中，ShardingSphere-Proxy 能够额外配置权限功能和更多的 Proxy 专有属性。

本章节将介绍 ShardingSphere-Proxy 的 YAML 额外配置。

权限

用于配置登录计算节点的初始用户，和存储节点数据授权。

配置项说明

```
rules:
  - !AUTHORITY
    users:
      - # 用于登录计算节点的用户名，授权主机和密码的组合。格式：<username>@<hostname>:
<password>，hostname 为 % 或空字符串表示不限制授权主机
    provider:
      type: # 存储节点数据授权的权限提供者类型，缺省值为 ALL_PRIVILEGES_PERMITTED
```

配置示例

ALL_PRIVILEGES_PERMITTED

```
rules:
  - !AUTHORITY
    users:
      - root@localhost:root
      - my_user@:pwd
    provider:
      type: ALL_PRIVILEGES_PERMITTED
```

SCHEMA_PRIVILEGES_PERMITTED

```
rules:
  - !AUTHORITY
    users:
      - root@:root
      - my_user@:pwd
    provider:
      type: SCHEMA_PRIVILEGES_PERMITTED
      props:
        user-schema-mappings: root@=sharding_db, root@=test_db, my_user@127.0.0.
1=sharding_db
```

以上配置表示：- root 用户从任意主机连接时，可访问 `sharding_db`。- root 用户从任意主机连接时，可访问 `test_db` 。- my_user 用户仅当从 127.0.0.1 连接时，可访问 `sharding_db`。

权限提供者具体实现可以参考 权限提供者。

属性配置

简介

Apache ShardingSphere 提供属性配置的方式配置系统级配置。

配置项说明

属性配置可以通过 DistSQL#RAL 修改。支持动态修改的属性可以立即生效，不支持动态修改的属性需要重启后生效。

7.2.3 DistSQL

本章节将介绍 DistSQL 的详细语法。

语法

本章节将对 DistSQL 的语法进行详细说明，并以实际的列子介绍 DistSQL 的使用。

RDL 语法

RDL (Resource & Rule Definition Language) 为 Apache ShardingSphere 的资源和规则定义语言。

资源定义

语法说明

```
ADD RESOURCE dataSource [, dataSource] ...

ALTER RESOURCE dataSource [, dataSource] ...

DROP RESOURCE dataSourceName [, dataSourceName] ... [ignore single tables]

dataSource:
    simpleSource | urlSource

simpleSource:
    dataSourceName(HOST=hostname,PORT=port,DB=dbName,USER=user [,PASSWORD=password]
[,PROPERTIES(poolProperty [,poolProperty]) ...])

urlSource:
    dataSourceName(URL=url,USER=user [,PASSWORD=password] [,PROPERTIES(poolProperty
[,poolProperty]) ...])
```

```
poolProperty:
    "key"= ("value" | value)
```

- 添加资源前请确认已经创建分布式数据库，并执行 use 命令成功选择一个数据库；
- 确认增加的资源是可以正常连接的，否则将不能添加成功；
- 重复的 dataSourceName 不允许被添加；
- 在同一 dataSource 的定义中，simpleSource 和 urlSource 语法不可混用；
- poolProperty 用于自定义连接池参数，key 必须和连接池参数名一致，value 支持 int 和 String 类型；
- ALTER RESOURCE 修改资源时不允许改变该资源关联的真实数据源；
- ALTER RESOURCE 修改资源时会发生连接池的切换，这个操作可能对进行中的业务造成影响，请谨慎使用；
- DROP RESOURCE 只会删除逻辑资源，不会删除真实的数据源；
- 被规则引用的资源将无法被删除；
- 若资源只被 single table rule 引用，且用户确认可以忽略该限制，则可以添加可选参数 ignore single tables 进行强制删除。

示例

```
ADD RESOURCE resource_0 (
    HOST=127.0.0.1,
    PORT=3306,
    DB=db0,
    USER=root,
    PASSWORD=root
),resource_1 (
    HOST=127.0.0.1,
    PORT=3306,
    DB=db1,
    USER=root
),resource_2 (
    HOST=127.0.0.1,
    PORT=3306,
    DB=db2,
    USER=root,
    PROPERTIES("maximumPoolSize"=10)
),resource_3 (
    URL="jdbc:mysql://127.0.0.1:3306/db3?serverTimezone=UTC&useSSL=false",
    USER=root,
    PASSWORD=root,
    PROPERTIES("maximumPoolSize"=10,"idleTimeout"="30000")
);
```

```
ALTER RESOURCE resource_0 (
    HOST=127.0.0.1,
    PORT=3309,
    DB=db0,
    USER=root,
    PASSWORD=root
),resource_1 (
    URL="jdbc:mysql://127.0.0.1:3309/db1?serverTimezone=UTC&useSSL=false",
    USER=root,
    PASSWORD=root,
    PROPERTIES("maximumPoolSize"=10,"idleTimeout"="30000")
);

DROP RESOURCE resource_0, resource_1;
DROP RESOURCE resource_2, resource_3 ignore single tables;
```

规则定义

本章节将对规则定义的语法进行详细说明。

数据分片

语法说明

Sharding Table Rule

```
CREATE SHARDING TABLE RULE shardingTableRuleDefinition [,
shardingTableRuleDefinition] ...

ALTER SHARDING TABLE RULE shardingTableRuleDefinition [,
shardingTableRuleDefinition] ...

DROP SHARDING TABLE RULE tableName [, tableName] ...

CREATE DEFAULT SHARDING shardingScope STRATEGY (shardingStrategy)

ALTER DEFAULT SHARDING shardingScope STRATEGY (shardingStrategy)

DROP DEFAULT SHARDING shardingScope STRATEGY;

CREATE SHARDING ALGORITHM shardingAlgorithmDefinition [,
shardingAlgorithmDefinition] ...

ALTER SHARDING ALGORITHM shardingAlgorithmDefinition [,
shardingAlgorithmDefinition] ...
```

```
DROP SHARDING ALGORITHM algorithmName [, algorithmName] ...

CREATE SHARDING KEY GENERATOR keyGeneratorDefinition [, keyGeneratorDefinition] ...

ALTER SHARDING KEY GENERATOR keyGeneratorDefinition [, keyGeneratorDefinition] ...

DROP SHARDING KEY GENERATOR keyGeneratorName [, keyGeneratorName] ...

shardingTableRuleDefinition:
    shardingAutoTableRule | shardingTableRule

shardingAutoTableRule:
    tableName(resources, shardingColumn, algorithmDefinition [,
keyGenerateDeclaration])

shardingTableRule:
    tableName(dataNodes [, databaseStrategy] [, tableStrategy] [,
keyGenerateDeclaration])

resources:
    RESOURCES(resource [, resource] ...)

dataNodes:
    DATANODES(dataNode [, dataNode] ...)

resource:
    resourceName | inlineExpression

dataNode:
    resourceName | inlineExpression

shardingColumn:
    SHARDING_COLUMN=columnName

algorithmDefinition:
    TYPE(NAME=shardingAlgorithmType [, PROPERTIES([algorithmProperties])])

keyGenerateDeclaration:
    keyGenerateDefinition | keyGenerateConstruction

keyGenerateDefinition:
    KEY_GENERATE_STRATEGY(COLUMN=columnName, strategyDefinition)

shardingScope:
    DATABASE | TABLE

databaseStrategy:
```

```
    DATABASE_STRATEGY(shardingStrategy)

tableStrategy:
    TABLE_STRATEGY(shardingStrategy)

keyGenerateConstruction
    KEY_GENERATE_STRATEGY(COLUMN=columnName, KEY_
GENERATOR=keyGenerateAlgorithmName)

shardingStrategy:
    TYPE=strategyType, shardingColumn, shardingAlgorithm

shardingAlgorithm:
    existingAlgorithm | autoCreativeAlgorithm

existingAlgorithm:
    SHARDING_ALGORITHM=shardingAlgorithmName

autoCreativeAlgorithm:
    SHARDING_ALGORITHM(algorithmDefinition)

strategyDefinition:
    TYPE(NAME=keyGenerateStrategyType [, PROPERTIES([algorithmProperties])])

shardingAlgorithmDefinition:
    shardingAlgorithmName(algorithmDefinition)

algorithmProperties:
    algorithmProperty [, algorithmProperty] ...

algorithmProperty:
    key=value

keyGeneratorDefinition:
    keyGeneratorName (algorithmDefinition)
```

- RESOURCES 需使用 RDL 管理的数据源资源；
- shardingAlgorithmType 指定自动分片算法类型，请参考 自动分片算法；
- keyGenerateStrategyType 指定分布式主键生成策略，请参考 分布式主键；
- 重复的 tableName 将无法被创建；
- shardingAlgorithm 能够被不同的 Sharding Table Rule 复用，因此在执行 DROP SHARDING TABLE RULE 时，对应的 shardingAlgorithm 不会被移除；
- 如需移除 shardingAlgorithm，请执行 DROP SHARDING ALGORITHM；
- strategyType 指定分片策略，请参考分片策略；
- Sharding Table Rule 同时支持 Auto Table 和 Table 两种类型，两者在语法上有所差异，对

应配置文件请参考 数据分片；

- 使用 autoCreativeAlgorithm 方式指定 shardingStrategy 时，将会自动创建新的分片算法，算法命名规则为 tableName_strategyType_shardingAlgorithmType，如 t_order_database_inline。

Sharding Binding Table Rule

```
CREATE SHARDING BINDING TABLE RULES bindTableRulesDefinition [,
bindTableRulesDefinition] ...

ALTER SHARDING BINDING TABLE RULES bindTableRulesDefinition [,
bindTableRulesDefinition] ...

DROP SHARDING BINDING TABLE RULES bindTableRulesDefinition [,
bindTableRulesDefinition] ...

bindTableRulesDefinition:
    (tableName [, tableName] ... )
```

- ALTER 会使用新的配置直接覆盖数据库内的绑定表配置

Sharding Broadcast Table Rule

```
CREATE SHARDING BROADCAST TABLE RULES (tableName [, tableName] ...)

ALTER SHARDING BROADCAST TABLE RULES (tableName [, tableName] ...)

DROP SHARDING BROADCAST TABLE RULES (tableName [, tableName] ...)
```

- ALTER 会使用新的配置直接覆盖数据库内的广播表配置

Sharding Scaling Rule

```
CREATE SHARDING SCALING RULE scalingName [scalingRuleDefinition]

DROP SHARDING SCALING RULE scalingName

ENABLE SHARDING SCALING RULE scalingName

DISABLE SHARDING SCALING RULE scalingName

scalingRuleDefinition:
    [inputDefinition] [, outputDefinition] [, streamChannel] [, completionDetector]
[, dataConsistencyChecker]
```

```
inputDefinition:
    INPUT ([workerThread] [, batchSize] [, rateLimiter])

outputDefinition:
    OUTPUT ([workerThread] [, batchSize] [, rateLimiter])

completionDetector:
    COMPLETION_DETECTOR (algorithmDefinition)

dataConsistencyChecker:
    DATA_CONSISTENCY_CHECKER (algorithmDefinition)

rateLimiter:
    RATE_LIMITER (algorithmDefinition)

streamChannel:
    STREAM_CHANNEL (algorithmDefinition)

workerThread:
    WORKER_THREAD=intValue

batchSize:
    BATCH_SIZE=intValue

intValue:
    INT
```

- ENABLE 用于设置启用哪个弹性伸缩配置；
- DISABLE 将禁用当前正在使用的配置；
- 创建 schema 中第一个弹性伸缩配置时，默认启用。

示例

Sharding Table Rule

Key Generator

```
CREATE SHARDING KEY GENERATOR snowflake_key_generator (
TYPE(NAME=SNOWFLAKE)
);

ALTER SHARDING KEY GENERATOR snowflake_key_generator (
TYPE(NAME=SNOWFLAKE)
);

DROP SHARDING KEY GENERATOR snowflake_key_generator;
```

Auto Table

```
CREATE SHARDING TABLE RULE t_order (
RESOURCES(resource_0,resource_1),
SHARDING_COLUMN=order_id,TYPE(NAME=hash_mod,PROPERTIES("sharding-count"=4)),
KEY_GENERATE_STRATEGY(COLUMN=another_id,TYPE(NAME=snowflake))
);

ALTER SHARDING TABLE RULE t_order (
RESOURCES(resource_0,resource_1,resource_2,resource_3),
SHARDING_COLUMN=order_id,TYPE(NAME=hash_mod,PROPERTIES("sharding-count"=16)),
KEY_GENERATE_STRATEGY(COLUMN=another_id,TYPE(NAME=snowflake))
);

DROP SHARDING TABLE RULE t_order;

DROP SHARDING ALGORITHM t_order_hash_mod;
```

Table

```
CREATE SHARDING ALGORITHM table_inline (
TYPE(NAME=inline,PROPERTIES("algorithm-expression"="t_order_item_${order_id % 2}"))
);

CREATE SHARDING TABLE RULE t_order_item (
DATANODES("resource_${0..1}.t_order_item_${0..1}"),
DATABASE_STRATEGY(TYPE=standard,SHARDING_COLUMN=user_id,SHARDING_
ALGORITHM(TYPE(NAME=inline,PROPERTIES("algorithm-expression"="resource_${user_id %
2}")))),
TABLE_STRATEGY(TYPE=standard,SHARDING_COLUMN=order_id,SHARDING_ALGORITHM=table_
inline),
KEY_GENERATE_STRATEGY(COLUMN=another_id,KEY_GENERATOR=snowflake_key_generator)
);

ALTER SHARDING ALGORITHM database_inline (
TYPE(NAME=inline,PROPERTIES("algorithm-expression"="resource_${user_id % 4}"))
),table_inline (
TYPE(NAME=inline,PROPERTIES("algorithm-expression"="t_order_item_${order_id % 4}"))
);

ALTER SHARDING TABLE RULE t_order_item (
DATANODES("resource_${0..3}.t_order_item${0..3}"),
DATABASE_STRATEGY(TYPE=standard,SHARDING_COLUMN=user_id,SHARDING_
ALGORITHM=database_inline),
TABLE_STRATEGY(TYPE=standard,SHARDING_COLUMN=order_id,SHARDING_ALGORITHM=table_
inline),
KEY_GENERATE_STRATEGY(COLUMN=another_id,KEY_GENERATOR=snowflake_key_generator)
);
```

```
DROP SHARDING TABLE RULE t_order_item;

DROP SHARDING ALGORITHM database_inline;

CREATE DEFAULT SHARDING DATABASE STRATEGY (
TYPE = standard,SHARDING_COLUMN=order_id,SHARDING_ALGORITHM=database_inline
);

ALTER DEFAULT SHARDING DATABASE STRATEGY (
TYPE = standard,SHARDING_COLUMN=another_id,SHARDING_ALGORITHM=database_inline
);

DROP DEFAULT SHARDING DATABASE STRATEGY;
```

Sharding Binding Table Rule

```
CREATE SHARDING BINDING TABLE RULES (t_order,t_order_item),(t_1,t_2);

ALTER SHARDING BINDING TABLE RULES (t_order,t_order_item);

DROP SHARDING BINDING TABLE RULES;

DROP SHARDING BINDING TABLE RULES (t_order,t_order_item);
```

Sharding Broadcast Table Rule

```
CREATE SHARDING BROADCAST TABLE RULES (t_b,t_a);

ALTER SHARDING BROADCAST TABLE RULES (t_b,t_a,t_3);

DROP SHARDING BROADCAST TABLE RULES;

DROP SHARDING BROADCAST TABLE RULES t_b;
```

Sharding Scaling Rule

```
CREATE SHARDING SCALING RULE sharding_scaling(
INPUT(
  WORKER_THREAD=40,
  BATCH_SIZE=1000
),
OUTPUT(
  WORKER_THREAD=40,
```

```
  BATCH_SIZE=1000
),
STREAM_CHANNEL(TYPE(NAME=MEMORY, PROPERTIES("block-queue-size"=10000))),
COMPLETION_DETECTOR(TYPE(NAME=IDLE, PROPERTIES("incremental-task-idle-minute-
threshold"=30))),
DATA_CONSISTENCY_CHECKER(TYPE(NAME=DATA_MATCH, PROPERTIES("chunk-size"=1000)))
);

ENABLE SHARDING SCALING RULE sharding_scaling;

DISABLE SHARDING SCALING RULE sharding_scaling;

DROP SHARDING SCALING RULE sharding_scaling;
```

单表

定义

```
CREATE DEFAULT SINGLE TABLE RULE singleTableRuleDefinition

ALTER DEFAULT SINGLE TABLE RULE singleTableRuleDefinition

DROP DEFAULT SINGLE TABLE RULE

singleTableRuleDefinition:
    RESOURCE = resourceName
```

- RESOURCE 需使用 RDL 管理的数据源资源。

示例

Single Table Rule

```
CREATE DEFAULT SINGLE TABLE RULE RESOURCE = ds_0

ALTER DEFAULT SINGLE TABLE RULE RESOURCE = ds_1

DROP DEFAULT SINGLE TABLE RULE
```

读写分离

语法说明

```
CREATE READWRITE_SPLITTING RULE readwriteSplittingRuleDefinition [,
readwriteSplittingRuleDefinition] ...

ALTER READWRITE_SPLITTING RULE readwriteSplittingRuleDefinition [,
readwriteSplittingRuleDefinition] ...

DROP READWRITE_SPLITTING RULE ruleName [, ruleName] ...

readwriteSplittingRuleDefinition:
    ruleName ([staticReadwriteSplittingRuleDefinition |
dynamicReadwriteSplittingRuleDefinition]
              [, loadBanlancerDefinition])

staticReadwriteSplittingRuleDefinition:
    WRITE_RESOURCE=writeResourceName, READ_RESOURCES(resourceName [, resourceName]
... )

dynamicReadwriteSplittingRuleDefinition:
    AUTO_AWARE_RESOURCE=resourceName

loadBanlancerDefinition:
    TYPE(NAME=loadBanlancerType [, PROPERTIES([algorithmProperties] )] )

algorithmProperties:
    algorithmProperty [, algorithmProperty] ...

algorithmProperty:
    key=value
```

- 支持创建静态读写分离规则和动态读写分离规则；
- 动态读写分离规则依赖于数据库发现规则；
- `loadBanlancerType` 指定负载均衡算法类型，请参考 负载均衡算法；
- 重复的 `ruleName` 将无法被创建。

示例

```
// Static
CREATE READWRITE_SPLITTING RULE ms_group_0 (
WRITE_RESOURCE=write_ds,
READ_RESOURCES(read_ds_0,read_ds_1),
TYPE(NAME=random)
);

// Dynamic
CREATE READWRITE_SPLITTING RULE ms_group_1 (
AUTO_AWARE_RESOURCE=group_0,
TYPE(NAME=random,PROPERTIES(read_weight='2:1'))
);

ALTER READWRITE_SPLITTING RULE ms_group_1 (
WRITE_RESOURCE=write_ds,
READ_RESOURCES(read_ds_0,read_ds_1,read_ds_2),
TYPE(NAME=random,PROPERTIES(read_weight='2:0'))
);

DROP READWRITE_SPLITTING RULE ms_group_1;
```

数据库发现

语法说明

```
CREATE DB_DISCOVERY RULE ruleDefinition [, ruleDefinition] ...

ALTER DB_DISCOVERY RULE ruleDefinition [, ruleDefinition] ...

DROP DB_DISCOVERY RULE ruleName [, ruleName] ...

CREATE DB_DISCOVERY TYPE databaseDiscoveryTypeDefinition [,
databaseDiscoveryTypeDefinition] ...

ALTER DB_DISCOVERY TYPE databaseDiscoveryTypeDefinition [,
databaseDiscoveryTypeDefinition] ...

DROP DB_DISCOVERY TYPE discoveryTypeName [, discoveryTypeName] ...

CREATE DB_DISCOVERY HEARTBEAT databaseDiscoveryHeartbaetDefinition [,
databaseDiscoveryHeartbaetDefinition] ...

ALTER DB_DISCOVERY HEARTBEAT databaseDiscoveryHeartbaetDefinition [,
databaseDiscoveryHeartbaetDefinition] ...
```

```
DROP DB_DISCOVERY HEARTBEAT discoveryHeartbeatName [, discoveryHeartbeatName] ...

ruleDefinition:
    (databaseDiscoveryRuleDefinition | databaseDiscoveryRuleConstruction)

databaseDiscoveryRuleDefinition
    ruleName (resources, typeDefinition, heartbeatDefinition)

databaseDiscoveryRuleConstruction
    ruleName (resources, TYPE = discoveryTypeName, HEARTBEAT =
discoveryHeartbeatName)

databaseDiscoveryTypeDefinition
    discoveryTypeName (typeDefinition)

databaseDiscoveryHeartbaetDefinition
    discoveryHeartbeatName (PROPERTIES (properties))

resources:
    RESOURCES(resourceName [, resourceName] ...)

typeDefinition:
    TYPE(NAME=typeName [, PROPERTIES([properties] )] )

heartbeatDefinition
    HEARTBEAT (PROPERTIES (properties))

properties:
    property [, property] ...

property:
    key=value
```

- discoveryType 指定数据库发现服务类型，ShardingSphere 内置支持 MGR；
- 重复的 ruleName 将无法被创建；
- 正在被使用的 discoveryType 和 discoveryHeartbeat 无法被删除；
- 带有 - 的命名在改动时需要使用 " "；
- 移除 discoveryRule 时不会移除被该 discoveryRule 使用的 discoveryType 和 discoveryHeartbeat。

示例

创建 discoveryRule 时同时创建 discoveryType 和 discoveryHeartbeat

```
CREATE DB_DISCOVERY RULE db_discovery_group_0 (
RESOURCES(ds_0, ds_1, ds_2),
TYPE(NAME=mgr,PROPERTIES('group-name'='92504d5b-6dec')),
HEARTBEAT(PROPERTIES('keep-alive-cron'='0/5 * * * * ?'))
);

ALTER DB_DISCOVERY RULE db_discovery_group_0 (
RESOURCES(ds_0, ds_1, ds_2),
TYPE(NAME=mgr,PROPERTIES('group-name'='246e9612-aaf1')),
HEARTBEAT(PROPERTIES('keep-alive-cron'='0/5 * * * * ?'))
);

DROP DB_DISCOVERY RULE db_discovery_group_0;

DROP DB_DISCOVERY TYPE db_discovery_group_0_mgr;

DROP DB_DISCOVERY HEARTBEAT db_discovery_group_0_heartbeat;
```

使用已有的 discoveryType 和 discoveryHeartbeat 创建 discoveryRule

```
CREATE DB_DISCOVERY TYPE db_discovery_group_1_mgr(
  TYPE(NAME=mgr,PROPERTIES('group-name'='92504d5b-6dec'))
);

CREATE DB_DISCOVERY HEARTBEAT db_discovery_group_1_heartbeat(
  PROPERTIES('keep-alive-cron'='0/5 * * * * ?')
);

CREATE DB_DISCOVERY RULE db_discovery_group_1 (
RESOURCES(ds_0, ds_1, ds_2),
TYPE=db_discovery_group_1_mgr,
HEARTBEAT=db_discovery_group_1_heartbeat
);

ALTER DB_DISCOVERY TYPE db_discovery_group_1_mgr(
  TYPE(NAME=mgr,PROPERTIES('group-name'='246e9612-aaf1'))
);

ALTER DB_DISCOVERY HEARTBEAT db_discovery_group_1_heartbeat(
  PROPERTIES('keep-alive-cron'='0/10 * * * * ?')
);
```

```
ALTER DB_DISCOVERY RULE db_discovery_group_1 (
RESOURCES(ds_0, ds_1),
TYPE=db_discovery_group_1_mgr,
HEARTBEAT=db_discovery_group_1_heartbeat
);

DROP DB_DISCOVERY RULE db_discovery_group_1;

DROP DB_DISCOVERY TYPE db_discovery_group_1_mgr;

DROP DB_DISCOVERY HEARTBEAT db_discovery_group_1_heartbeat;
```

数据加密

语法说明

```
CREATE ENCRYPT RULE encryptRuleDefinition [, encryptRuleDefinition] ...

ALTER ENCRYPT RULE encryptRuleDefinition [, encryptRuleDefinition] ...

DROP ENCRYPT RULE tableName [, tableName] ...

encryptRuleDefinition:
    tableName(COLUMNS(columnDefinition [, columnDefinition] ...), QUERY_WITH_
CIPHER_COLUMN=queryWithCipherColumn)

columnDefinition:
    (NAME=columnName [, PLAIN=plainColumnName] , CIPHER=cipherColumnName,
encryptAlgorithm)

encryptAlgorithm:
    TYPE(NAME=encryptAlgorithmType [, PROPERTIES([algorithmProperties] )] )

algorithmProperties:
    algorithmProperty [, algorithmProperty] ...

algorithmProperty:
    key=value
```

- `PLAIN` 指定明文数据列，`CIPHER` 指定密文数据列；
- `encryptAlgorithmType` 指定加密算法类型，请参考 加密算法；
- 重复的 `tableName` 将无法被创建；
- `queryWithCipherColumn` 支持大写或小写的 true 或 false。

示例

```
CREATE ENCRYPT RULE t_encrypt (
COLUMNS(
(NAME=user_id,PLAIN=user_plain,CIPHER=user_cipher,TYPE(NAME=AES,PROPERTIES('aes-key-value'='123456abc'))),
(NAME=order_id, CIPHER =order_cipher,TYPE(NAME=MD5))
),QUERY_WITH_CIPHER_COLUMN=true),
t_encrypt_2 (
COLUMNS(
(NAME=user_id,PLAIN=user_plain,CIPHER=user_cipher,TYPE(NAME=AES,PROPERTIES('aes-key-value'='123456abc'))),
(NAME=order_id, CIPHER=order_cipher,TYPE(NAME=MD5))
), QUERY_WITH_CIPHER_COLUMN=FALSE);

ALTER ENCRYPT RULE t_encrypt (
COLUMNS(
(NAME=user_id,PLAIN=user_plain,CIPHER=user_cipher,TYPE(NAME=AES,PROPERTIES('aes-key-value'='123456abc'))),
(NAME=order_id,CIPHER=order_cipher,TYPE(NAME=MD5))
), QUERY_WITH_CIPHER_COLUMN=TRUE);

DROP ENCRYPT RULE t_encrypt,t_encrypt_2;
```

影子库压测

语法说明

```
CREATE SHADOW RULE shadowRuleDefinition [, shadowRuleDefinition] ...

ALTER SHADOW RULE shadowRuleDefinition [, shadowRuleDefinition] ...

CREATE SHADOW ALGORITHM shadowAlgorithm [, shadowAlgorithm] ...

ALTER SHADOW ALGORITHM shadowAlgorithm [, shadowAlgorithm] ...

DROP SHADOW RULE ruleName [, ruleName] ...

DROP SHADOW ALGORITHM algorithmName [, algorithmName] ...

CREATE DEFAULT SHADOW ALGORITHM NAME = algorithmName

shadowRuleDefinition: ruleName(resourceMapping, shadowTableRule [, shadowTableRule] ...)

resourceMapping: SOURCE=resourceName, SHADOW=resourceName
```

```
shadowTableRule: tableName(shadowAlgorithm [, shadowAlgorithm] ...)

shadowAlgorithm: ([algorithmName, ] TYPE(NAME=shadowAlgorithmType,
PROPERTIES([algorithmProperties] ...)))

algorithmProperties: algorithmProperty [, algorithmProperty] ...

algorithmProperty: key=value
```

- 重复的 `ruleName` 无法被创建；
- `resourceMapping` 指定源数据库和影子库的映射关系，需使用 RDL 管理的 `resource`，请参考数据源资源；
- `shadowAlgorithm` 可同时作用于多个 `shadowTableRule`；
- `algorithmName` 未指定时会根据 `ruleName`、`tableName` 和 `shadowAlgorithmType` 自动生成；
- `shadowAlgorithmType` 目前支持 `VALUE_MATCH`、`REGEX_MATCH` 和 `SIMPLE_HINT`；
- `shadowTableRule` 能够被不同的 `shadowRuleDefinition` 复用，因此在执行 `DROP SHADOW RULE` 时，对应的 `shadowTableRule` 不会被移除；
- `shadowAlgorithm` 能够被不同的 `shadowTableRule` 复用，因此在执行 `ALTER SHADOW RULE` 时，对应的 `shadowAlgorithm` 不会被移除。

示例

```
CREATE SHADOW RULE shadow_rule(
SOURCE=demo_ds,
SHADOW=demo_ds_shadow,
t_order((simple_hint_algorithm, TYPE(NAME=SIMPLE_HINT, PROPERTIES("shadow"="true",
foo="bar"))),(TYPE(NAME=REGEX_MATCH, PROPERTIES("operation"="insert","column"=
"user_id", "regex"='[1]'))),
t_order_item((TYPE(NAME=VALUE_MATCH, PROPERTIES("operation"="insert","column"=
"user_id", "value"='1'))));

ALTER SHADOW RULE shadow_rule(
SOURCE=demo_ds,
SHADOW=demo_ds_shadow,
t_order((simple_hint_algorithm, TYPE(NAME=SIMPLE_HINT, PROPERTIES("shadow"="true",
foo="bar"))),(TYPE(NAME=REGEX_MATCH, PROPERTIES("operation"="insert","column"=
"user_id", "regex"='[1]'))),
t_order_item((TYPE(NAME=VALUE_MATCH, PROPERTIES("operation"="insert","column"=
"user_id", "value"='1'))));

CREATE SHADOW ALGORITHM
(simple_hint_algorithm, TYPE(NAME=SIMPLE_HINT, PROPERTIES("shadow"="true", "foo"=
"bar"))),
```

```
(user_id_match_algorithm, TYPE(NAME=REGEX_MATCH,PROPERTIES("operation"="insert",
"column"="user_id", "regex"='[1]')));


ALTER SHADOW ALGORITHM
(simple_hint_algorithm, TYPE(NAME=SIMPLE_HINT, PROPERTIES("shadow"="false", "foo"=
"bar"))),
(user_id_match_algorithm, TYPE(NAME=VALUE_MATCH,PROPERTIES("operation"="insert",
"column"="user_id", "value"='1')));

DROP SHADOW RULE shadow_rule;

DROP SHADOW ALGORITHM simple_note_algorithm;

CREATE DEFAULT SHADOW ALGORITHM NAME = simple_hint_algorithm;
```

RQL 语法

RQL (Resource & Rule Query Language) 为 Apache ShardingSphere 的资源和规则查询语言。

资源查询

语法说明

```
SHOW SCHEMA RESOURCES [FROM schemaName]
```

返回值说明

列	说明
name	数据源名称
type	数据源类型
host	数据源地址
port	数据源端口
db	数据库名称
attribute	数据源参数

示例

```
mysql> show schema resources;
+------+-------+-----------+------+------+-------------------------------------
-------------------------------------------------------------------------------
---------------------------------+
| name | type  | host      | port | db   | attribute

                                |
+------+-------+-----------+------+------+-------------------------------------
-------------------------------------------------------------------------------
---------------------------------+
| ds_0 | MySQL | 127.0.0.1 | 3306 | ds_0 | {"minPoolSize":1,
"connectionTimeoutMilliseconds":30000,"maxLifetimeMilliseconds":1800000,"readOnly
":false,"idleTimeoutMilliseconds":60000,"maxPoolSize":50} |
| ds_1 | MySQL | 127.0.0.1 | 3306 | ds_1 | {"minPoolSize":1,
"connectionTimeoutMilliseconds":30000,"maxLifetimeMilliseconds":1800000,"readOnly
":false,"idleTimeoutMilliseconds":60000,"maxPoolSize":50} |
+------+-------+-----------+------+------+-------------------------------------
-------------------------------------------------------------------------------
---------------------------------+
2 rows in set (0.84 sec)
```

规则查询

本章节将对规则查询的语法进行详细说明。

数据分片

语法说明

Sharding Table Rule

```
SHOW SHARDING TABLE tableRule | RULES [FROM schemaName]

SHOW SHARDING ALGORITHMS [FROM schemaName]

SHOW UNUSED SHARDING ALGORITHMS [FROM schemaName]

SHOW SHARDING KEY GENERATORS [FROM schemaName]

SHOW UNUSED SHARDING KEY GENERATORS [FROM schemaName]

SHOW SHARDING TABLE RULES USED KEY GENERATOR keyGeneratorName [FROM schemaName]
```

```
SHOW DEFAULT SHARDING STRATEGY

SHOW SHARDING TABLE NODES;

tableRule:
    RULE tableName
```

- 支持查询所有数据分片规则和指定表查询;
- 支持查询所有分片算法。

Sharding Binding Table Rule

```
SHOW SHARDING BINDING TABLE RULES [FROM schemaName]
```

Sharding Broadcast Table Rule

```
SHOW SHARDING BROADCAST TABLE RULES [FROM schemaName]
```

Sharding Scaling Rule

```
SHOW SHARDING SCALING RULES [FROM schemaName]
```

返回值说明

Sharding Table Rule

列	说明
table	逻辑表名
actual_data_nodes	实际的数据节点
actual_data_sources	实际的数据源（通过 RDL 创建的规则时显示）
database_strategy_type	数据库分片策略类型
database_sharding_column	数据库分片键
database_sharding_algorithm_type	数据库分片算法类型
d atabase_sharding_algorithm_props	数据库分片算法参数
table_strategy_type	表分片策略类型
table_sharding_column	表分片键
table_sharding_algorithm_type	表分片算法类型
table_sharding_algorithm_props	表分片算法参数
key_generate_column	分布式主键生成列
key_generator_type	分布式主键生成器类型
key_generator_props	分布式主键生成器参数

Sharding Algorithms

列	说明
name	分片算法名称
type	分片算法类型
props	分片算法参数

Unused Sharding Algorithms

列	说明
name	分片算法名称
type	分片算法类型
props	分片算法参数

Sharding Key Generators

列	说明
name	分片列生成器名称
type	分片列生成器类型
props	分片列生成器参数

Unused Sharding Key Generators

列	说明
name	分片列生成器名称
type	分片列生成器类型
props	分片列生成器参数

Default Sharding Strategy

列	说明
name	策略名称
type	分片策略类型
sharding_column	分片键
sharding_algorithm_name	分片算法名称
sharding_algorithm_type	分片算法类型
sharding_algorithm_props	分片算法参数

Sharding Table Nodes

列	说明
name	分片规则名称
nodes	分片节点

Sharding Binding Table Rule

列	说明
sharding_binding_tables	绑定表名称

Sharding Broadcast Table Rule

列	说明
sharding_broadcast_tables	广播表名称

Sharding Scaling Rule

列	说明
name	弹性伸缩配置名称
input	数据读取配置
output	数据写入配置
stream_channel	数据通道配置
completion_detector	作业完成检测算法配置
data_consistency_checker	数据一致性校验算法配置

示例

Sharding Table Rule

SHOW SHARDING TABLE RULES

```
mysql> SHOW SHARDING TABLE RULES;
+--------------+---------------------------------+-------------------+-------------
---------+------------------------+-------------------------------+----------------
------------------------+-------------------+---------------------+----------------
------------+-------------------------------------------------------+------------------
-+------------------+-------------------+
| table        | actual_data_nodes               | actual_data_sources | database_
strategy_type | database_sharding_column | database_sharding_algorithm_type |
database_sharding_algorithm_props         | table_strategy_type | table_sharding_
column | table_sharding_algorithm_type | table_sharding_algorithm_props
          | key_generate_column | key_generator_type | key_generator_props |
+--------------+---------------------------------+-------------------+-------------
---------+------------------------+-------------------------------+----------------
------------------------+-------------------+---------------------+----------------
------------+-------------------------------------------------------+------------------
-+------------------+-------------------+
| t_order      | ds_${0..1}.t_order_${0..1}      |                   | INLINE
        | user_id                | INLINE                        | algorithm-
expression:ds_${user_id % 2} | INLINE            | order_id            | INLINE
                | algorithm-expression:t_order_${order_id % 2}      | order_id
    | SNOWFLAKE        |                   |
| t_order_item | ds_${0..1}.t_order_item_${0..1} |                   | INLINE
        | user_id                | INLINE                        | algorithm-
expression:ds_${user_id % 2} | INLINE            | order_id            | INLINE
                | algorithm-expression:t_order_item_${order_id % 2} | order_item_id
    | SNOWFLAKE        |                   |
| t2           |                                 | ds_0,ds_1         |
        |                        |                               |
                        | mod                | id                  | mod
         | sharding-count:10                                |                   |
                |                   |
```

```
+--------------+--------------------------------+-------------------+-------------
---------+------------------------+-------------------------------+----------------
------------------------+-------------------+---------------------+----------------
------------+-----------------------------------------------------+------------------
-+------------------+-------------------+
3 rows in set (0.02 sec)
```

SHOW SHARDING TABLE RULE tableName

```
mysql> SHOW SHARDING TABLE RULE t_order;
+---------+---------------------------+-------------------+----------------------
+------------------------+-------------------------------+-------------------------
---------------+-------------------+---------------------+-------------------------
---+-----------------------------------------------+-------------------+------------
------+-------------------+
| table   | actual_data_nodes          | actual_data_sources | database_strategy_
type | database_sharding_column | database_sharding_algorithm_type | database_
sharding_algorithm_props         | table_strategy_type | table_sharding_column |
table_sharding_algorithm_type | table_sharding_algorithm_props                  |
key_generate_column | key_generator_type | key_generator_props |
+---------+---------------------------+-------------------+----------------------
+------------------------+-------------------------------+-------------------------
---------------+-------------------+---------------------+-------------------------
---+-----------------------------------------------+-------------------+------------
------+-------------------+
| t_order | ds_${0..1}.t_order_${0..1} |                   | INLINE               |
user_id                | INLINE                        | algorithm-expression:ds_$
{user_id % 2} | INLINE            | order_id            | INLINE
 | algorithm-expression:t_order_${order_id % 2} | order_id          | SNOWFLAKE
   |                   |
+---------+---------------------------+-------------------+----------------------
+------------------------+-------------------------------+-------------------------
---------------+-------------------+---------------------+-------------------------
---+-----------------------------------------------+-------------------+------------
------+-------------------+
1 row in set (0.01 sec)
```

SHOW SHARDING ALGORITHMS

```
mysql> SHOW SHARDING ALGORITHMS;
+-------------------------+--------+-----------------------------------------------
------+
| name                    | type   | props
     |
+-------------------------+--------+-----------------------------------------------
------+
| t_order_inline          | INLINE | algorithm-expression=t_order_${order_id % 2}
     |
| t_order_item_inline     | INLINE | algorithm-expression=t_order_item_${order_id %
2}   |
```

```
+-------------------------+--------+-----------------------------------------------
------+
2 row in set (0.01 sec)
```

SHOW UNUSED SHARDING ALGORITHMS

```
mysql> SHOW UNUSED SHARDING ALGORITHMS;
+---------------+--------+-----------------------------------------------------+
| name          | type   | props                                               |
+---------------+--------+-----------------------------------------------------+
| t1_inline     | INLINE | algorithm-expression=t_order_${order_id % 2}        |
+---------------+--------+-----------------------------------------------------+
1 row in set (0.01 sec)
```

SHOW SHARDING KEY GENERATORS

```
mysql> SHOW SHARDING KEY GENERATORS;
+------------------------+-----------+-----------------+
| name                   | type      | props           |
+------------------------+-----------+-----------------+
| t_order_snowflake      | snowflake |                 |
| t_order_item_snowflake | snowflake |                 |
| uuid_key_generator     | uuid      |                 |
+------------------------+-----------+-----------------+
3 row in set (0.01 sec)
```

SHOW UNUSED SHARDING KEY GENERATORS

```
mysql> SHOW UNUSED SHARDING KEY GENERATORS;
+------------------------+-----------+-----------------+
| name                   | type      | props           |
+------------------------+-----------+-----------------+
| uuid_key_generator     | uuid      |                 |
+------------------------+-----------+-----------------+
1 row in set (0.01 sec)
```

SHOW SHARDING TABLE RULES USED KEY GENERATOR keyGeneratorName

```
mysql> SHOW SHARDING TABLE RULES USED KEY GENERATOR keyGeneratorName;
+------------------------+-----------+-----------------+
| schema                 | type      | name            |
+------------------------+-----------+-----------------+
| sharding_db            | table     | t_order         |
+------------------------+-----------+-----------------+
1 row in set (0.01 sec)
```

SHOW DEFAULT SHARDING STRATEGY

```
mysql> SHOW DEFAULT SHARDING STRATEGY ;

+----------+---------+--------------------+------------------------+--------------
-----------+-----------------------------------------+
| name     | type    | sharding_column    | sharding_algorithm_name | sharding_
algorithm_type | sharding_algorithm_props                |
+----------+---------+--------------------+------------------------+--------------
-----------+-----------------------------------------+
| TABLE    | NONE    |                    |                        |
          |                                         |
| DATABASE | STANDARD| order_id           | database_inline        | INLINE
          | {algorithm-expression=ds_${user_id % 2}} |
+----------+---------+--------------------+------------------------+--------------
-----------+-----------------------------------------+
2 rows in set (0.07 sec)
```

SHOW SHARDING TABLE NODES

```
mysql> SHOW SHARDING TABLE NODES;
+---------+-------------------------------------------------------------------+
| name    | nodes                                                             |
+---------+-------------------------------------------------------------------+
| t_order | ds_0.t_order_0, ds_1.t_order_1, ds_0.t_order_2, ds_1.t_order_3 |
+---------+-------------------------------------------------------------------+
1 row in set (0.02 sec)
```

Sharding Binding Table Rule

```
mysql> SHOW SHARDING BINDING TABLE RULES;
+----------------------+
| sharding_binding_tables |
+----------------------+
| t_order,t_order_item |
| t1,t2                |
+----------------------+
2 rows in set (0.00 sec)
```

Sharding Broadcast Table Rule

```
mysql> SHOW SHARDING BROADCAST TABLE RULES;
+------------------------+
| sharding_broadcast_tables |
+------------------------+
| t_1                    |
| t_2                    |
```

```
+------------------------+
2 rows in set (0.00 sec)
```

Sharding Scaling Rule

```
mysql> SHOW SHARDING SCALING RULES;
+------------------+-------------------------------------------------------------------
-------------------------+-------------------------------------------------------------
---------------------------------+-----------------------------------------------
-------+-------------------------------------------------------------------------+-
---------------------------------------------------+
| name             | input
                        | output
                               | stream_channel
    | completion_detector                                                       |
data_consistency_checker                            |
+------------------+-------------------------------------------------------------------
-------------------------+-------------------------------------------------------------
---------------------------------+-----------------------------------------------
-------+-------------------------------------------------------------------------+-
---------------------------------------------------+
| sharding_scaling | {"workerThread":40,"batchSize":1000} | {"workerThread":40,
"batchSize":1000} | {"type":"MEMORY","props":{"block-queue-size":"10000"}} | {"type
":"IDLE","props":{"incremental-task-idle-minute-threshold":"30"}} | {"type":"DATA_
MATCH","props":{"chunk-size":"1000"}} |
+------------------+-------------------------------------------------------------------
-------------------------+-------------------------------------------------------------
---------------------------------+-----------------------------------------------
-------+-------------------------------------------------------------------------+-
---------------------------------------------------+
1 row in set (0.00 sec)
```

单表

语法说明

```
SHOW SINGLE TABLE (tableRule | RULES) [FROM schemaName]

SHOW SINGLE TABLES

tableRule:
    RULE tableName
```

返回值说明

Single Table Rule

列	说明
name	规则名称
resource_name	数据源名称

Single Table

列	说明
table_name	单表名称
resource_name	单表所在的数据源名称

示例

single table rules

```
sql> show single table rules;
+---------+---------------+
| name    | resource_name |
+---------+---------------+
| default | ds_1          |
+---------+---------------+
1 row in set (0.01 sec)
```

single tables

```
mysql> show single tables;
+--------------+---------------+
| table_name   | resource_name |
+--------------+---------------+
| t_single_0   | ds_0          |
| t_single_1   | ds_1          |
+--------------+---------------+
2 rows in set (0.02 sec)
```

读写分离

语法说明

```
SHOW READWRITE_SPLITTING RULES [FROM schemaName]
```

返回值说明

列	说明
name	规则名称
auto_aware_data_source_name	自动发现数据源名称（配置动态读写分离规则显示）
write_data_source_name	写数据源名称
read_data_source_names	读数据源名称列表
load_balancer_type	负载均衡算法类型
load_balancer_props	负载均衡算法参数

示例

静态读写分离规则

```
mysql> show readwrite_splitting rules;
+------------+----------------------------+------------------------+--------------
----------+--------------------+---------------------+
| name       | auto_aware_data_source_name | write_data_source_name | read_data_
source_names | load_balancer_type | load_balancer_props |
+------------+----------------------------+------------------------+--------------
----------+--------------------+---------------------+
| ms_group_0 |                            | ds_primary             | ds_slave_0,
ds_slave_1 | random             |                     |
+------------+----------------------------+------------------------+--------------
----------+--------------------+---------------------+
1 row in set (0.00 sec)
```

动态读写分离规则

```
mysql> show readwrite_splitting rules from readwrite_splitting_db;
+--------------+----------------------------+------------------------+------------
------------+--------------------+---------------------+
| name         | auto_aware_data_source_name | write_data_source_name | read_data_
source_names | load_balancer_type | load_balancer_props |
+--------------+----------------------------+------------------------+------------
------------+--------------------+---------------------+
| readwrite_ds | ms_group_0                 |                        |
           | random             | read_weight=2:1     |
```

```
+-------+----------------------------+-----------------------+-------------------
-----+--------------------+---------------------+
1 row in set (0.01 sec)
```

静态读写分离规则和动态读写分离规则

```
mysql> show readwrite_splitting rules from readwrite_splitting_db;
+--------------+----------------------------+-----------------------+------------
------------+--------------------+---------------------+
| name         | auto_aware_data_source_name | write_data_source_name | read_data_
source_names | load_balancer_type | load_balancer_props |
+--------------+----------------------------+-----------------------+------------
------------+--------------------+---------------------+
| readwrite_ds | ms_group_0                  | write_ds               | read_ds_0,
read_ds_1    | random             | read_weight=2:1     |
+-------+----------------------------+-----------------------+-------------------
-----+--------------------+---------------------+
1 row in set (0.00 sec)
```

数据库发现

语法说明

```
SHOW DB_DISCOVERY RULES [FROM schemaName]

SHOW DB_DISCOVERY TYPES [FROM schemaName]

SHOW DB_DISCOVERY HEARTBEATS [FROM schemaName]
```

返回值说明

DB Discovery Rule

列	说明
group_name	规则名称
data_source_names	数据源名称列表
primary_data_source_name	主数据源名称
discovery_type	数据库发现服务类型
discovery_heartbeat	数据库发现服务心跳

DB Discovery Type

列	说明
name	类型名称
type	类型种类
props	类型参数

DB Discovery Heartbeat

列	说明
name	心跳名称
props	心跳参数

示例

DB Discovery Rule

```
mysql> show db_discovery rules;
+----------------------+-------------------+-------------------------+------------
-----------------------------------------------------------------+-----------------
-------------------------------------------------------------+
| group_name           | data_source_names | primary_data_source_name | discovery_
type                                                              | discovery_
heartbeat                                                          |
+----------------------+-------------------+-------------------------+------------
-----------------------------------------------------------------+-----------------
-------------------------------------------------------------+
| db_discovery_group_0 | ds_0,ds_1,ds_2    |        ds_0              | {name=db_
discovery_group_0_mgr, type=mgr, props={group-name=92504d5b-6dec}} | {name=db_
discovery_group_0_heartbeat, props={keep-alive-cron=0/5 * * * * ?}} |
+----------------------+-------------------+-------------------------+------------
-----------------------------------------------------------------+----------------
-------------------------------------------------------------+
1 row in set (0.20 sec)
```

DB Discovery Type

```
mysql> show db_discovery types;
+--------------------------+------+----------------------------+
| name                     | type | props                      |
+--------------------------+------+----------------------------+
| db_discovery_group_0_mgr | mgr  | {group-name=92504d5b-6dec} |
+--------------------------+------+----------------------------+
1 row in set (0.01 sec)
```

DB Discovery Heartbeat

```
mysql> show db_discovery heartbeats;
+--------------------------------+---------------------------------+
| name                           | props                           |
+--------------------------------+---------------------------------+
| db_discovery_group_0_heartbeat | {keep-alive-cron=0/5 * * * * ?} |
+--------------------------------+---------------------------------+
1 row in set (0.01 sec)
```

数据加密

语法说明

```
SHOW ENCRYPT RULES [FROM schemaName]

SHOW ENCRYPT TABLE RULE tableName [from schemaName]
```

- 支持查询所有的数据加密规则和指定逻辑表名查询。

返回值说明

列	说明
table	逻辑表名
logic_column	逻辑列名
logic_data_type	逻辑列数据类型
cipher_column	密文列名
cipher_data_type	密文列数据类型
plain_column	明文列名
plain_data_type	明文列数据类型
assisted_query_column	辅助查询列名
assisted_query_data_type	辅助查询列数据类型
encryptor_type	加密算法类型
encryptor_props	加密算法参数
query_with_cipher_column	是否使用加密列进行查询

示例

显示加密规则

```
mysql> show encrypt rules from encrypt_db;
+-------------+--------------+-----------------+---------------+------------------
+--------------+-----------------+----------------------+-------------------------
-+----------------+-------------------------+-------------------------+
| table       | logic_column | logic_data_type | cipher_column | cipher_data_type |
plain_column | plain_data_type | assisted_query_column | assisted_query_data_type |
encryptor_type | encryptor_props         | query_with_cipher_column |
+-------------+--------------+-----------------+---------------+------------------
+--------------+-----------------+----------------------+-------------------------
-+----------------+-------------------------+-------------------------+
| t_encrypt   | user_id      |                 | user_cipher   |                  |
user_plain   |                 |                      |                         |
AES            | aes-key-value=123456abc | true                     |
| t_encrypt   | order_id     |                 | order_cipher  |                  |
             |                 |                      |                         |
MD5            |                         | true                     |
| t_encrypt_2 | user_id      |                 | user_cipher   |                  |
user_plain   |                 |                      |                         |
AES            | aes-key-value=123456abc | false                    |
| t_encrypt_2 | order_id     |                 | order_cipher  |                  |
             |                 |                      |                         |
MD5            |                         | false                    |
+-------------+--------------+-----------------+---------------+------------------
+--------------+-----------------+----------------------+-------------------------
-+----------------+-------------------------+-------------------------+
4 rows in set (0.78 sec)
```

显示加密表规则表名

```
mysql> show encrypt table rule t_encrypt;
+-------------+--------------+-----------------+---------------+------------------
+--------------+-----------------+----------------------+-------------------------
-+----------------+-------------------------+-------------------------+
| table       | logic_column | logic_data_type | cipher_column | cipher_data_type |
plain_column | plain_data_type | assisted_query_column | assisted_query_data_type |
encryptor_type | encryptor_props         | query_with_cipher_column |
+-------------+--------------+-----------------+---------------+------------------
+--------------+-----------------+----------------------+-------------------------
-+----------------+-------------------------+-------------------------+
| t_encrypt   | user_id      |                 | user_cipher   |                  |
user_plain   |                 |                      |                         |
AES            | aes-key-value=123456abc | true                     |
| t_encrypt   | order_id     |                 | order_cipher  |                  |
             |                 |                      |                         |
MD5            |                         | true                     |
```

```
+-------------+--------------+-----------------+---------------+-----------------
+--------------+-----------------+----------------------+------------------------
-+----------------+------------------------+------------------------+
2 rows in set (0.01 sec)

mysql> show encrypt table rule t_encrypt from encrypt_db;
+-------------+--------------+-----------------+---------------+-----------------
+--------------+-----------------+----------------------+------------------------
-+----------------+------------------------+------------------------+
| table       | logic_column | logic_data_type | cipher_column | cipher_data_type |
plain_column | plain_data_type | assisted_query_column | assisted_query_data_type |
encryptor_type | encryptor_props         | query_with_cipher_column |
+-------------+--------------+-----------------+---------------+-----------------
+--------------+-----------------+----------------------+------------------------
-+----------------+------------------------+------------------------+
| t_encrypt   | user_id      |                 | user_cipher   |                  |
user_plain   |                 |                      |                          |
AES            | aes-key-value=123456abc | true                     |
| t_encrypt   | order_id     |                 | order_cipher  |                  |
             |                 |                      |                          |
MD5            |                         | true                     |
+-------------+--------------+-----------------+---------------+-----------------
+--------------+-----------------+----------------------+------------------------
-+----------------+------------------------+------------------------+
2 rows in set (0.01 sec))
```

影子库压测

语法说明

```
SHOW SHADOW shadowRule | RULES [FROM schemaName]

SHOW SHADOW TABLE RULES [FROM schemaName]

SHOW SHADOW ALGORITHMS [FROM schemaName]

shadowRule:
    RULE ruleName
```

- 支持查询所有影子规则和指定表查询;
- 支持查询所有表规则;
- 支持查询所有影子算法。

返回值说明

Shadow Rule

列	说明
rule_name	规则名称
source_name	源数据库
shadow_name	影子数据库
shadow_table	影子表

Shadow Table Rule

列	说明
shadow_table	影子表
shadow_algorithm_name	影子算法名称

Shadow Algorithms

列	说明
shadow_algorithm_name	影子算法名称
type	算法类型
props	算法参数
is_default	是否默认

Shadow Rule status

列	说明
status	是否启用

示例

SHOW SHADOW RULES

```
mysql> show shadow rules;
+--------------------+-------------+-------------+--------------+
| rule_name          | source_name | shadow_name | shadow_table |
+--------------------+-------------+-------------+--------------+
| shadow_rule_1      | ds_1        | ds_shadow_1 | t_order      |
| shadow_rule_2      | ds_2        | ds_shadow_2 | t_order_item |
```

```
+--------------------+-------------+-------------+--------------+
2 rows in set (0.02 sec)
```

SHOW SHADOW RULE ruleName

```
mysql> show shadow rule shadow_rule_1;
+------------------+-------------+-------------+--------------+
| rule_name        | source_name | shadow_name | shadow_table |
+------------------+-------------+-------------+--------------+
| shadow_rule_1    | ds_1        | ds_shadow_1 | t_order      |
+------------------+-------------+-------------+--------------+
1 rows in set (0.01 sec)
```

SHOW SHADOW TABLE RULES

```
mysql> show shadow table rules;
+--------------+-------------------------------------------------------------------
-------------+
| shadow_table | shadow_algorithm_name
            |
+--------------+-------------------------------------------------------------------
-------------+
| t_order_1    | user_id_match_algorithm,simple_note_algorithm_1
            |
+--------------+-------------------------------------------------------------------
-------------+
1 rows in set (0.01 sec)
```

SHOW SHADOW ALGORITHMS

```
mysql> show shadow algorithms;
+-------------------------+--------------------+----------------------------------
--------+----------------+
| shadow_algorithm_name   | type               | props
       | is_default     |
+-------------------------+--------------------+----------------------------------
--------+----------------+
| user_id_match_algorithm | COLUMN_REGEX_MATCH | operation=insert,column=user_id,
regex=[1] | false          |
| simple_note_algorithm_1 | SIMPLE_NOTE        | shadow=true,foo=bar
       | false          |
+-------------------------+--------------------+----------------------------------
--------+----------------+
2 rows in set (0.01 sec)
```

RAL 语法

RAL (Resource & Rule Administration Language) 为 Apache ShardingSphere 的管理语言，负责强制路由、事务类型切换、弹性伸缩、分片执行计划查询等增量功能的操作。

强制路由

语句	说明	示例
set readwrite_splitting hint source = [auto / write]	针对当前连接, 设置读写分离的路由策略（自动路由或强制到写库）	set re adwrite_splitting hint source = write
set sharding hint database_value = yy	针对当前连接，设置 hint 仅对数据库分片有效，并添加分片值，yy：数据库分片值	set sharding hint database_value = 100
add sharding hint database_value xx = yy	针对当前连接，为表 xx 添加分片值 yy，xx：逻辑表名称，yy：数据库分片值	add sharding hint database_value t_order= 100
add sharding hint table_value xx = yy	针对当前连接，为表 xx 添加分片值 yy，xx：逻辑表名称，yy：表分片值	add sharding hint table_value t_order = 100
clear hint	针对当前连接，清除 hint 所有设置	clear hint
clear [sharding hint / readwrite_splitting hint]	针对当前连接，清除 sharding 或 readwrite_splitting 的 hint 设置	clear re adwrite_splitting hint
show [sharding / readwrite_splitting] hint status	针对当前连接，查询 sharding 或 readwrite_splitting 的 hint 设置	show re adwrite_splitting hint status

弹性伸缩

语句	说明	示例
show scaling list	查询运行列表	show scaling list
show scaling status xx	查询任务状态，xx：任务 id	show scaling status 1234
start scaling xx	开始运行任务，xx：任务 id	start scaling 1234
stop scaling xx	停止运行任务，xx：任务 id	stop scaling 12345
drop scaling xx	移除任务，xx：任务 id	drop scaling 1234
reset scaling xx	重置任务进度，xx：任务 id	reset scaling 1234
check scaling xx	数据一致性校验，使用 `server.yaml` 里的校验算法，xx：任务 id	check scaling 1234
show scaling check algorithms	展示可用的一致性校验算法	show scaling check algorithms
check scaling {jobId} by type(name={algorithmType})	数据一致性校验，使用指定的校验算法	check scaling 1234 by type(name=DEFAULT)
stop scaling source writing xx	旧的 ShardingSphere 数据源停写，xx：任务 id	stop scaling source writing 1234
restore scaling source writing xx	旧的 ShardingSphere 数据源恢复写，xx：任务 id	restore scaling source writing 1234
apply scaling xx	切换至新的 ShardingSphere 元数据，xx：任务 id	apply scaling 1234

熔断

语句	说明	示例
[enable / disable] readwrite_splitting read xxx [from schema]	启用 / 禁用读库	enable r eadwrite_splitting read resource_0
[enable / disable] instance [IP=xxx, PORT=xxx / instanceId]	启用 / 禁用 proxy 实例	disable instance 127.0.0.1@3307
show instance list	查询 proxy 实例信息	show instance list
show readwrite_splitting read resources [from schema]	查询所有读库的状态	show r eadwrite_splitting read resources

全局规则

语句	说明	示例
SHOW AUTHORITY RULE	查询权限规则配置	SHOW AUTHORITY RULE
SHOW TRANSACTION RULE	查询事务规则配置	SHOW TRANSACTION RULE
SHOW SQL_PARSER RULE	查询解析引擎规则配置	SHOW SQL_PARSER RULE
ALTER TRANSACTION RULE(DEFAULT=xx,TYPE(NAME=xxx, PROPERTIES(“key1”=“val ue1”,“key2”=“value2”…)))	更新事务规则配置，DEFAULT：默认事务类型，支持 LOCAL、XA、BASE；N AME：事务管理器名称，支持 Ato mikos、Narayana 和 Bitronix	ALTER TRANSACTION RULE(DEFAULT=XA,TYPE(NAME=Narayana, PROPERTIE S(“databaseName”=“jbosst s”,“host”=“127.0.0.1”)))
ALTER SQL_PARSER RULE **SQL_**COMMENT_PARSE_ENABLE=xx, PARSE_TREE_CACHE(INITIAL_CAPACITY=xx, MAXIMUM_SIZE=xx, CONCURRENCY_LEVEL=xx), SQL_STATEMENT_CACHE(INITIAL_CAPACITY=xxx, MAXIMUM_SIZE=xxx, CONCURRENCY_LEVEL=xxx)	更新解析引擎规则配置，SQL_COMMENT_PARSE_ENABLE：是否解析 SQL 注释，“ PARSE_TREE_CACHE“：语法树本地缓存配置，SQL_STATEMENT_CACHE：SQL 语句本地缓存配置项	ALTER SQL_PARSER RULE SQL_COMMENT_PARSE_ENABLE=false, PARSE_TREE_CACHE(INITIAL_CAPACITY=10, MAXIMUM_SIZE=11, CONCURRENCY_LEVEL=1), SQL_STATEMENT_CACHE(INITIAL_CAPACITY=11, MAXIMUM_SIZE=11, CONCURRENCY_LEVEL=100)

其他

语句	说明	示例
show instance mode	查询当前 proxy 的 mode 配置	show instance mode
count schema rules [from schema]	查询 schema 中的规则数量	count schema rules
set variable proxy_property_name = xx	proxy_property_name 为 proxy 的属性配置，需使用下划线命名	set variable sql_show = true
set variable transaction_type = xx	修改当前连接的事务类型，支持 LOCAL，XA，BASE	set variable t ransaction_type = XA
set variable agent_plugins_enabled = [true / false]	设置 agent 插件的启用状态，默认值 false	set variable **agent_** plugins_enabled = true
show all variables	查询 proxy 所有的属性配置	show all variables
show variable variable_name	查询 proxy 属性，需使用下划线命名	show variable sql_show
preview SQL	预览实际 SQL	preview select * from t_order
parse SQL	解析 SQL 并输出抽象语法树 parse select * from t_order	
refresh table metadata	刷新所有表的元数据	refresh table metadata
refresh table metadata [tableName / tableName from resource resourceName]	刷新指定表的元数据	refresh table metadata t_order from resource ds_1
show table metadata tableName [, tableName] …	查询表的元数据	show table metadata t_order
export schema config [from schema_name] [, file= “file_path”]	查询 / 导出 schema 中的资源和规则配置	export schema config from readwrite_splitting_db
show rules used resource resourceName [from schema]	查询 schema 中使用指定资源的规则	show rules used resource ds_0 from schemaName

注意事项

ShardingSphere-Proxy 默认不支持 hint，如需支持，请在 `conf/server.yaml` 中，将 `properties` 的属性 `proxy-hint-enabled` 设置为 true。

使用

本章节将结合 DistSQL 的语法，并以实战的形式分别介绍如何使用 DistSQL 管理分布式数据库下的资源和规则。

前置工作

以 MySQL 为例，其他数据库可直接替换。

1. 启动 MySQL 服务；
2. 创建待注册资源的 MySQL 数据库；
3. 在 MySQL 中为 ShardingSphere-Proxy 创建一个拥有创建权限的角色或者用户；
4. 启动 ZooKeeper 服务；
5. 添加 `mode` 和 `authentication` 配置参数到 `server.yaml`；
6. 启动 ShardingSphere-Proxy；
7. 通过应用程序或终端连接到 ShardingSphere-Proxy；

创建数据库

1. 创建逻辑库。

```
CREATE DATABASE foo_db;
```

2. 使用新创建的逻辑库。

```
USE foo_db;
```

资源操作

详见具体规则示例。

规则操作

详见具体规则示例。

注意事项

1. 当前, DROP DATABASE 只会移除 逻辑的分布式数据库，不会删除用户真实的数据库；
2. DROP TABLE 会将逻辑分片表和数据库中真实的表全部删除；
3. CREATE DATABASE 只会创建 逻辑的分布式数据库，所以需要用户提前创建好真实的数据库。

数据分片

资源操作

```
ADD RESOURCE ds_0 (
HOST=127.0.0.1,
PORT=3306,
DB=ds_1,
USER=root,
PASSWORD=root
);

ADD RESOURCE ds_1 (
HOST=127.0.0.1,
PORT=3306,
DB=ds_2,
USER=root,
PASSWORD=root
);
```

规则操作

- 创建分片规则

```
CREATE SHARDING TABLE RULE t_order(
RESOURCES(ds_0,ds_1),
SHARDING_COLUMN=order_id,
TYPE(NAME=hash_mod,PROPERTIES("sharding-count"=4)),
KEY_GENERATE_STRATEGY(COLUMN=order_id,TYPE(NAME=snowflake))
);
```

- 创建切分表

```
CREATE TABLE `t_order` (
  `order_id` int NOT NULL,
  `user_id` int NOT NULL,
  `status` varchar(45) DEFAULT NULL,
  PRIMARY KEY (`order_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4
```

- 删除切分表

```
DROP TABLE t_order;
```

- 删除分片规则

```
DROP SHARDING TABLE RULE t_order;
```

- 删除数据源

```
DROP RESOURCE ds_0, ds_1;
```

- 删除分布式数据库

```
DROP DATABASE foo_db;
```

读写分离

资源操作

```
ADD RESOURCE write_ds (
HOST=127.0.0.1,
PORT=3306,
DB=ds_0,
USER=root,
PASSWORD=root
),read_ds (
HOST=127.0.0.1,
PORT=3307,
DB=ds_0,
USER=root,
PASSWORD=root
);
```

规则操作

- 创建读写分离规则

```
CREATE READWRITE_SPLITTING RULE group_0 (
WRITE_RESOURCE=write_ds,
READ_RESOURCES(read_ds),
TYPE(NAME=random)
);
```

- 修改读写分离规则

```
ALTER READWRITE_SPLITTING RULE group_0 (
WRITE_RESOURCE=write_ds,
READ_RESOURCES(read_ds),
TYPE(NAME=random,PROPERTIES(read_weight='2:0'))
);
```

- 删除读写分离规则

```
DROP READWRITE_SPLITTING RULE group_0;
```

- 删除数据源

```
DROP RESOURCE write_ds,read_ds;
```

- 删除分布式数据库

```
DROP DATABASE readwrite_splitting_db;
```

数据加密

资源操作

```
ADD RESOURCE ds_0 (
    HOST=127.0.0.1,
    PORT=3306,
    DB=ds_0,
    USER=root,
    PASSWORD=root
);
```

规则操作

- 创建加密规则

```
CREATE ENCRYPT RULE t_encrypt (
    COLUMNS(
        (NAME=user_id,PLAIN=user_plain,CIPHER=user_cipher,TYPE(NAME=AES,PROPERTIES(
'aes-key-value'='123456abc'))),
        (NAME=order_id,PLAIN=order_plain,CIPHER =order_cipher,TYPE(NAME=RC4,
PROPERTIES('rc4-key-value'='123456abc')))
));
```

- 创建加密表

```
CREATE TABLE `t_encrypt` (
    `id` int(11) NOT NULL,
```

```
    `user_id` varchar(45) DEFAULT NULL,
    `order_id` varchar(45) DEFAULT NULL,
    PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
```

- 修改加密规则

```
ALTER ENCRYPT RULE t_encrypt (
    COLUMNS(
        (NAME=user_id,PLAIN=user_plain,CIPHER=user_cipher,TYPE(NAME=AES,PROPERTIES(
'aes-key-value'='123456abc')))
));
```

- 删除加密规则

```
DROP ENCRYPT RULE t_encrypt;
```

- 删除数据源

```
DROP RESOURCE ds_0;
```

- 删除分布式数据库

```
DROP DATABASE encrypt_db;
```

数据库发现

资源操作

```
ADD RESOURCE ds_0 (
HOST=127.0.0.1,
PORT=3306,
DB=ds_0,
USER=root,
PASSWORD=root
),ds_1 (
HOST=127.0.0.1,
PORT=3306,
DB=ds_1,
USER=root,
PASSWORD=root
),ds_2 (
HOST=127.0.0.1,
PORT=3306,
DB=ds_2,
USER=root,
```

```
PASSWORD=root
);
```

规则操作

- 创建数据库发现规则

```
CREATE DB_DISCOVERY RULE db_discovery_group_0 (
RESOURCES(ds_0, ds_1),
TYPE(NAME=mgr,PROPERTIES('group-name'='92504d5b-6dec')),
HEARTBEAT(PROPERTIES('keep-alive-cron'='0/5 * * * * ?'))
);
```

- 修改数据库发现规则

```
ALTER DB_DISCOVERY RULE db_discovery_group_0 (
RESOURCES(ds_0, ds_1, ds_2),
TYPE(NAME=mgr,PROPERTIES('group-name'='92504d5b-6dec')),
HEARTBEAT(PROPERTIES('keep-alive-cron'='0/5 * * * * ?'))
);
```

- 删除数据库发现规则

```
DROP DB_DISCOVERY RULE db_discovery_group_0;
```

- 删除数据库发现类型

```
DROP DB_DISCOVERY TYPE db_discovery_group_0_mgr;
```

- 删除数据库发现心跳

```
DROP DB_DISCOVERY HEARTBEAT db_discovery_group_0_heartbeat;
```

- 删除数据源

```
DROP RESOURCE ds_0,ds_1,ds_2;
```

- 删除分布式数据库

```
DROP DATABASE discovery_db;
```

影子库压测

资源操作

```
ADD RESOURCE ds_0 (
HOST=127.0.0.1,
PORT=3306,
DB=ds_0,
USER=root,
PASSWORD=root
),ds_1 (
HOST=127.0.0.1,
PORT=3306,
DB=ds_1,
USER=root,
PASSWORD=root
),ds_2 (
HOST=127.0.0.1,
PORT=3306,
DB=ds_2,
USER=root,
PASSWORD=root
);
```

规则操作

- 创建影子库压测规则

```
CREATE SHADOW RULE group_0(
SOURCE=ds_0,
SHADOW=ds_1,
t_order((simple_hint_algorithm, TYPE(NAME=SIMPLE_HINT, PROPERTIES("foo"="bar"))),
(TYPE(NAME=REGEX_MATCH, PROPERTIES("operation"="insert","column"="user_id", "regex
"='[1]')))),
t_order_item((TYPE(NAME=SIMPLE_HINT, PROPERTIES("foo"="bar")))));
```

- 修改影子库压测规则

```
ALTER SHADOW RULE group_0(
SOURCE=ds_0,
SHADOW=ds_2,
t_order_item((TYPE(NAME=SIMPLE_HINT, PROPERTIES("foo"="bar")))));
```

- 删除影子库压测规则

```
DROP SHADOW RULE group_0;
```

- 删除数据源

```
DROP RESOURCE ds_0,ds_1,ds_2;
```

9. 删除分布式数据库

```
DROP DATABASE foo_db;
```

7.3 ShardingSphere-Sidecar

7.3.1 简介

ShardingSphere-Sidecar 是 ShardingSphere 的第三个产品，目前仍然在 规划中。定位为 Kubernetes 或 Mesos 的云原生数据库代理，以 DaemonSet 的形式代理所有对数据库的访问。

通过无中心、零侵入的方案提供与数据库交互的的啮合层，即 Database Mesh，又可称数据网格。Database Mesh 的关注重点在于如何将分布式的数据访问应用与数据库有机串联起来，它更加关注的是交互，是将杂乱无章的应用与数据库之间的交互进行有效地梳理。使用 Database Mesh，访问数据库的应用和数据库终将形成一个巨大的网格体系，应用和数据库只需在网格体系中对号入座即可，它们都是被啮合层所治理的对象。

图 8: ShardingSphere-Sidecar Architecture

7.3.2 对比

	Shard ingSphere-JDBC	*Shardi ngSphere-Proxy*	*Sharding Sphere-Sidecar*
数据库	任意	M ySQL/PostgreSQL	`MyS QL/PostgreSQL`
连接消耗数	高	低	高
异构语言	仅 Java	任意	任意
性能	损耗低	损耗略高	损耗低
无中心化	是	否	是
静态入口	无	有	无

ShardingSphere-Sidecar 的优势在于对 Kubernetes 和 Mesos 的云原生支持。

7.4 ShardingSphere-Scaling

7.4.1 简介

ShardingSphere-Scaling 是一个提供给用户的通用的 ShardingSphere 数据接入迁移，及弹性伸缩的解决方案。

于 **4.1.0** 开始向用户提供，目前仍处于 **实验室**版本。

7.4.2 运行部署

部署启动

1. 执行以下命令，编译生成 ShardingSphere-Proxy 二进制包：

```
git clone --depth 1 https://github.com/apache/shardingsphere.git
cd shardingsphere
mvn clean install -Dmaven.javadoc.skip=true -Dcheckstyle.skip=true -Drat.skip=true
-Djacoco.skip=true -DskipITs -DskipTests -Prelease
```

发布包：- /shardingsphere-distribution/shardingsphere-proxy-distribution/target/apache-shardingsphere-${latest.release.version}-shardingsphere-proxy-bin.tar.gz

或者通过下载页面获取安装包。

> Scaling 还是实验性质的功能，建议使用 master 分支最新版本，点击此处下载每日构建版本

2. 解压缩 proxy 发布包，修改配置文件 `conf/config-sharding.yaml`。详情请参见 proxy 启动手册。
3. 修改配置文件 `conf/server.yaml`，详情请参见模式配置。

目前 `mode` 必须是 `Cluster`，需要提前启动对应的注册中心。

配置示例：

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

4. 开启 scaling。

方法 1：修改配置文件 `conf/config-sharding.yaml` 的 `scalingName` 和 `scaling` 部分。

配置项说明：

```
rules:
- !SHARDING
  # 忽略的配置

  scalingName: # 启用的弹性伸缩配置名称
  scaling:
    <scaling-action-config-name> (+):
      input: # 数据读取配置。如果不配置则部分参数默认生效。
        workerThread: # 从源端摄取全量数据的线程池大小。如果不配置则使用默认值。
        batchSize: # 一次查询操作返回的最大记录数。如果不配置则使用默认值。
        rateLimiter: # 限流算法。如果不配置则不限流。
          type: # 算法类型。可选项:
          props: # 算法属性
      output: # 数据写入配置。如果不配置则部分参数默认生效。
        workerThread: # 数据写入到目标端的线程池大小。如果不配置则使用默认值。
        batchSize: # 一次批量写入操作的最大记录数。如果不配置则使用默认值。
        rateLimiter: # 限流算法。如果不配置则不限流。
          type: # 算法类型。可选项:
          props: # 算法属性
      streamChannel: # 数据通道，连接生产者和消费者，用于 input 和 output 环节。如果不配置则默认使用 MEMORY 类型
        type: # 算法类型。可选项: MEMORY
        props: # 算法属性
          block-queue-size: # 属性: 阻塞队列大小
      completionDetector: # 作业是否接近完成检测算法。如果不配置则无法自动进行后续步骤，可以通过 DistSQL 手动操作。
        type: # 算法类型。可选项: IDLE
        props: # 算法属性
          incremental-task-idle-minute-threshold: # 如果增量同步任务不再活动超过一定时间，那么可以认为增量同步任务接近完成。适用算法类型: IDLE
```

```
      dataConsistencyChecker: # 数据一致性校验算法。如果不配置则跳过这个步骤。
        type: # 算法类型。可选项: DATA_MATCH, CRC32_MATCH
        props: # 算法属性
          chunk-size: # 一次查询操作返回的最大记录数
```

配置示例:

```
rules:
- !SHARDING
  # 忽略的配置

  scalingName: default_scaling
  scaling:
    default_scaling:
      input:
        workerThread: 40
        batchSize: 1000
      output:
        workerThread: 40
        batchSize: 1000
      streamChannel:
        type: MEMORY
        props:
          block-queue-size: 10000
      completionDetector:
        type: IDLE
        props:
          incremental-task-idle-minute-threshold: 30
      dataConsistencyChecker:
        type: DATA_MATCH
        props:
          chunk-size: 1000
```

以上的 completionDetector, dataConsistencyChecker 都可以通过实现 SPI 自定义。可以参考现有实现，详情请参见开发者手册 # 弹性伸缩。

方法 2：通过 DistSQL 配置 scaling

创建 scaling 配置示例:

```
CREATE SHARDING SCALING RULE default_scaling (
INPUT(
  WORKER_THREAD=40,
  BATCH_SIZE=1000
),
OUTPUT(
  WORKER_THREAD=40,
  BATCH_SIZE=1000
),
```

```
STREAM_CHANNEL(TYPE(NAME=MEMORY, PROPERTIES("block-queue-size"=10000))),
COMPLETION_DETECTOR(TYPE(NAME=IDLE, PROPERTIES("incremental-task-idle-minute-
threshold"=3))),
DATA_CONSISTENCY_CHECKER(TYPE(NAME=DATA_MATCH, PROPERTIES("chunk-size"=1000)))
);
```

详情请参见 RDL# 数据分片。

5. 启动 ShardingSphere-Proxy:

```
sh bin/start.sh
```

6. 查看 proxy 日志 `logs/stdout.log`，看到日志中出现:

```
[INFO ] [main] o.a.s.p.frontend.ShardingSphereProxy - ShardingSphere-Proxy start
success
```

确认启动成功。

结束

```
sh bin/stop.sh
```

7.4.3 使用手册

使用手册

环境要求

纯 JAVA 开发，JDK 建议 1.8 以上版本。

支持迁移场景如下:

源端	目标端
MySQL(5.1.15 ~ 5.7.x)	MySQL(5.1.15 ~ 5.7.x)
PostgreSQL(9.4 ~)	PostgreSQL(9.4 ~)
openGauss(2.1.0)	openGauss(2.1.0)

注意:

如果后端连接以下数据库,请下载相应 JDBC 驱动 jar 包,并将其放入 `${shardingsphere-proxy}/lib` 目录。

数据库	JDBC 驱动	参考
MySQL	**'mysql-connector-java-5.1.47.jar < https://repo1.maven.org/maven2/mysql/mysql-connector-java/5.1.47/mysql-connector-java-5.1.47.jar>'__**	Connector/J Versions
openGauss	opengauss-jdbc-2.0.1-compatibility.jar	

功能支持情况：

功能	MySQL	PostgreSQL	openGauss
全量迁移	支持	支持	支持
增量迁移	支持	支持	支持
自动建表	支持	不支持	支持
DATA_MATCH 一致性校验	支持	支持	支持
CRC32_MATCH 一致性校验	支持	不支持	不支持

注意：

还没开启 自动建表的数据库需要手动创建分表。

权限要求

MySQL

1. 开启 `binlog`

MySQL 5.7 `my.cnf` 示例配置：

```
[mysqld]
server-id=1
log-bin=mysql-bin
binlog-format=row
binlog-row-image=full
max_connections=600
```

执行以下命令，确认是否有开启 binlog：

```
show variables like '%log_bin%';
show variables like '%binlog%';
```

如以下显示，则说明 binlog 已开启

```
+-----------------------------------------+---------------------------------------+
| Variable_name                           | Value                                 |
+-----------------------------------------+---------------------------------------+
```

```
| log_bin                                  | ON                                      |
| binlog_format                            | ROW                                     |
| binlog_row_image                         | FULL                                    |
+------------------------------------------+-----------------------------------------+
```

2. 赋予 MySQL 账号 Replication 相关权限。

执行以下命令，查看该用户是否有迁移权限：

```
SHOW GRANTS 'user';
```

示例结果：

```
+-------------------------------------------------------------------------------+
|Grants for ${username}@${host}                                                 |
+-------------------------------------------------------------------------------+
|GRANT REPLICATION SLAVE, REPLICATION CLIENT ON *.* TO ${username}@${host}      |
|.......                                                                        |
+-------------------------------------------------------------------------------+
```

PostgreSQL

1. 开启 test_decoding。
2. 调整 WAL 配置。

`postgresql.conf` 示例配置：

```
wal_level = logical
max_replication_slots = 10
```

详情请参见 Write Ahead Log 和 Replication。

DistSQL 自动模式接口

预览当前分片规则

示例：

```
preview select count(1) from t_order;
```

返回信息：

```
mysql> preview select count(1) from t_order;
+------------------+--------------------------------+
| data_source_name | sql                            |
+------------------+--------------------------------+
| ds_0             | select count(1) from t_order_0 |
```

```
| ds_0              | select count(1) from t_order_1 |
| ds_1              | select count(1) from t_order_0 |
| ds_1              | select count(1) from t_order_1 |
+-------------------+--------------------------------+
4 rows in set (0.00 sec)
```

创建迁移任务

1. 添加新的数据源。

详情请参见 RDL # 数据源资源。

先在底层数据库系统创建需要的分库，下面的 `DistSQL` 需要用到。

示例：

```
ADD RESOURCE ds_2 (
    URL="jdbc:mysql://127.0.0.1:3306/scaling_ds_2?serverTimezone=UTC&useSSL=false",
    USER=root,
    PASSWORD=root,
    PROPERTIES("maximumPoolSize"=10,"idleTimeout"="30000")
), ds_3 (
    URL="jdbc:mysql://127.0.0.1:3306/scaling_ds_3?serverTimezone=UTC&useSSL=false",
    USER=root,
    PASSWORD=root,
    PROPERTIES("maximumPoolSize"=10,"idleTimeout"="30000")
), ds_4 (
    URL="jdbc:mysql://127.0.0.1:3306/scaling_ds_4?serverTimezone=UTC&useSSL=false",
    USER=root,
    PASSWORD=root,
    PROPERTIES("maximumPoolSize"=10,"idleTimeout"="30000")
);
```

2. 修改待迁移表的分片规则。

待迁移表可以是所有表，也可以是部分表。绑定表只能一块迁移。

目前只有通过执行 `ALTER SHARDING TABLE RULE` DistSQL 来触发迁移。

详情请参见 RDL # 数据分片。

`SHARDING TABLE RULE` 支持 2 种类型：`TableRule` 和 `AutoTableRule`。以下是两种分片规则的对比：

类型	AutoTableRule（自动分片）	TableRule（自定义分片）
定义	自动化分片算法	自定义分片算法

DistSQL 字段含义和 YAML 配置保持一致，详情请参见 YAML 配置 # 数据分片。

`AutoTableRule` 修改示例：

```
ALTER SHARDING TABLE RULE t_order (
RESOURCES(ds_2, ds_3, ds_4),
SHARDING_COLUMN=order_id,
TYPE(NAME=hash_mod,PROPERTIES("sharding-count"=6)),
KEY_GENERATE_STRATEGY(COLUMN=order_id,TYPE(NAME=snowflake))
);
```

RESOURCES 从 (ds_0, ds_1) 改为了 (ds_2, ds_3, ds_4)，sharding-count 从 4 改为了 6，会触发迁移。

TableRule 修改示例：

```
ALTER SHARDING ALGORITHM database_inline (
TYPE(NAME=INLINE,PROPERTIES("algorithm-expression"="ds_${user_id % 3 + 2}"))
);

ALTER SHARDING TABLE RULE t_order (
DATANODES("ds_${2..4}.t_order_${0..1}"),
DATABASE_STRATEGY(TYPE=standard,SHARDING_COLUMN=user_id,SHARDING_
ALGORITHM=database_inline),
TABLE_STRATEGY(TYPE=standard,SHARDING_COLUMN=order_id,SHARDING_ALGORITHM=t_order_
inline),
KEY_GENERATE_STRATEGY(COLUMN=order_id,TYPE(NAME=snowflake))
), t_order_item (
DATANODES("ds_${2..4}.t_order_item_${0..1}"),
DATABASE_STRATEGY(TYPE=standard,SHARDING_COLUMN=user_id,SHARDING_
ALGORITHM=database_inline),
TABLE_STRATEGY(TYPE=standard,SHARDING_COLUMN=order_id,SHARDING_ALGORITHM=t_order_
item_inline),
KEY_GENERATE_STRATEGY(COLUMN=order_item_id,TYPE(NAME=snowflake))
);
```

database_inline 的 algorithm-expression 从 ds_${user_id % 2} 改为 ds_${user_id % 3 + 2}，t_order 的 DATANODES 从 ds_${0..1}.t_order_${0..1} 改为 ds_${2..4}.t_order_${0..1}，会触发迁移。

目前 ALTER SHARDING ALGORITHM 会即时生效、但是规则还没生效，可能会导致源端 insert 异常，所以建议优先修改为 AutoTableRule。

查询所有迁移任务

详情请参见 RAL # 弹性伸缩。

示例：

```
show scaling list;
```

返回信息：

```
mysql> show scaling list;
+--------------------+----------------------+---------------------+--------+-----
----------------+---------------------+
| id                 | tables               | sharding_total_count | active |
create_time         | stop_time           |
+--------------------+----------------------+---------------------+--------+-----
----------------+---------------------+
| 659853312085983232 | t_order_item, t_order | 2                   | false  |
2021-10-26 20:21:31 | 2021-10-26 20:24:01 |
| 660152090995195904 | t_order_item, t_order | 2                   | false  |
2021-10-27 16:08:43 | 2021-10-27 16:11:00 |
+--------------------+----------------------+---------------------+--------+-----
----------------+---------------------+
2 rows in set (0.04 sec)
```

查询迁移任务进度

示例:

```
show scaling status {jobId};
```

返回信息:

```
mysql> show scaling status 660152090995195904;
+------+-------------+----------+-------------------------------+------------------
--------+
| item | data_source | status   | inventory_finished_percentage | incremental_idle_
seconds |
+------+-------------+----------+-------------------------------+------------------
--------+
| 0    | ds_1        | FINISHED | 100                           | 2834
       |
| 1    | ds_0        | FINISHED | 100                           | 2834
       |
+------+-------------+----------+-------------------------------+------------------
--------+
2 rows in set (0.00 sec)
```

当前迁移任务已完成，新的分片规则已生效。如果迁移失败，新的分片规则不会生效。

`status` 的取值:

取值	描述
PREPARING	准备中
RUNNING	运行中
EXECUTE_INVENTORY_TASK	全量迁移中
EXECUTE_INCREMENTAL_TASK	增量迁移中
FINISHED	已完成（整个流程完成了，新规则已生效）
PREPARING_FAILURE	准备阶段失败
EXECUTE_INVENTORY_TASK_FAILURE	全量迁移阶段失败
EXECUTE_INCREMENTAL_TASK_FAILURE	增量迁移阶段失败

如果 `status` 出现失败的情况，可以查看 `proxy` 的日志查看错误堆栈分析问题。

预览新的分片规则是否生效

示例：

```
preview select count(1) from t_order;
```

返回信息：

```
mysql> preview select count(1) from t_order;
+------------------+-------------------------------+
| data_source_name | sql                           |
+------------------+-------------------------------+
| ds_2             | select count(1) from t_order_0 |
| ds_2             | select count(1) from t_order_1 |
| ds_3             | select count(1) from t_order_0 |
| ds_3             | select count(1) from t_order_1 |
| ds_4             | select count(1) from t_order_0 |
| ds_4             | select count(1) from t_order_1 |
+------------------+-------------------------------+
6 rows in set (0.01 sec)
```

其他 DistSQL

详情请参见 RAL # 弹性伸缩。

DistSQL 手动模式接口

数据校验、切换配置等操作可以手动执行。详情请参见：RAL # 弹性伸缩。

8 开发者手册

Apache ShardingSphere 可插拔架构提供了数十个基于 SPI 的扩展点。对于开发者来说，可以十分方便的对功能进行定制化扩展。

本章节将 Apache ShardingSphere 的 SPI 扩展点悉数列出。如无特殊需求，用户可以使用 Apache ShardingSphere 提供的内置实现；高级用户则可以参考各个功能模块的接口进行自定义实现。

Apache ShardingSphere 社区非常欢迎开发者将自己的实现类反馈至开源社区，让更多用户从中收益。

8.1 运行模式

8.1.1 StandalonePersistRepository

SPI 名称	详细说明
StandalonePersistRepository	Standalone 模式配置信息持久化

已知实现类	详细说明
FileRepository	基于 File 的持久化

8.1.2 ClusterPersistRepository

SPI 名称	详细说明
ClusterPersistRepository	Cluster 模式配置信息持久化

已知实现类	详细说明
CuratorZookeeperRepository	基于 ZooKeeper 的持久化
EtcdRepository	基于 etcd 的持久化

8.1.3 GovernanceWatcher

SPI 名称	详细说明
GovernanceWatcher	治理监听器

已知实现类	详细说明
StorageNodeStateChangedWatcher	存储节点状态变化监听器
ComputeNodeStateChangedWatcher	计算节点状态变化监听器
PropertiesChangedWatcher	属性变化监听器
PrivilegeNodeChangedWatcher	权限变化监听器
GlobalRuleChangedWatcher	全局规则配置变化监听器
MetaDataChangedWatcher	元数据变化监听器

8.2 配置

8.2.1 RuleBuilder

SPI 名称	详细说明
RuleBuilder	用于将用户配置转化为规则对象

已知实现类	详细说明
AlgorithmProvidedReadwriteSplittingRuleBuilder	用于将基于算法的读写分离用户配置转化为读写分离规则对象
AlgorithmProvidedDatabaseDiscoveryRuleBuilder	用于将基于算法的数据库发现用户配置转化为数据库发现规则对象
AlgorithmProvidedShardingRuleBuilder	用于将基于算法的分片用户配置转化为分片规则对象
AlgorithmProvidedEncryptRuleBuilder	用于将基于算法的加密用户配置转化为加密规则对象
AlgorithmProvidedShadowRuleBuilder	用于将基于算法的影子库用户配置转化为影子库规则对象
ReadwriteSplittingRuleBuilder	用于将读写分离用户配置转化为读写分离规则对象
DatabaseDiscoveryRuleBuilder	用于将数据库发现用户配置转化为数据库发现规则对象
SingleTableRuleBuilder	用于将单表用户配置转化为单表规则对象
AuthorityRuleBuilder	用于将权限用户配置转化为权限规则对象
ShardingRuleBuilder	用于将分片用户配置转化为分片规则对象
EncryptRuleBuilder	用于将加密用户配置转化为加密规则对象
ShadowRuleBuilder	用于将影子库用户配置转化为影子库规则对象
TransactionRuleBuilder	用于将事务用户配置转化为事务规则对象
SQLParserRuleBuilder	用于将 SQL 解析用户配置转化为 SQL 解析规则对象

8.2.2 YamlRuleConfigurationSwapper

SPI 名称	详细说明
YamlRuleConfigurationSwapper	用于将 YAML 配置转化为标准用户配置

已知实现类	详细说明
ReadwriteSplittingRu leAlgorithmProviderConfigurationYamlSwapper	用于将基于算法的读写分离配置转化为读写分离标准配置
DatabaseDiscoveryRu leAlgorithmProviderConfigurationYamlSwapper	用于将基于算法的数据库发现配置转化为数据库发现标准配置
ShardingRu leAlgorithmProviderConfigurationYamlSwapper	用于将基于算法的分片配置转化为分片标准配置
EncryptRu leAlgorithmProviderConfigurationYamlSwapper	用于将基于算法的加密配置转化为加密标准配置
ShadowRu leAlgorithmProviderConfigurationYamlSwapper	用于将基于算法的影子库配置转化为影子库标准配置
Rea dwriteSplittingRuleConfigurationYamlSwapper	用于将读写分离的 YAML 配置转化为读写分离标准配置
Da tabaseDiscoveryRuleConfigurationYamlSwapper	用于将数据库发现的 YAML 配置转化为数据库发现标准配置
AuthorityRuleConfigurationYamlSwapper	用于将权限规则的 YAML 配置转化为权限规则标准配置
ShardingRuleConfigurationYamlSwapper	用于将分片的 YAML 配置转化为分片标准配置
EncryptRuleConfigurationYamlSwapper	用于将加密的 YAML 配置转化为加密标准配置
ShadowRuleConfigurationYamlSwapper	用于将影子库的 YAML 配置转化为影子库标准配置
TransactionRuleConfigurationYamlSwapper	用于将事务的 YAML 配置转化为事务标准配置
SingleTableRuleConfigurationYamlSwapper	用于将单表的 YAML 配置转化为单表标准配置
SQLParserRuleConfigurationYamlSwapper	用于将 SQL 解析的 YAML 配置转化为 SQL 解析标准配置

8.2.3 ShardingSphereYamlConstruct

SPI 名称	详细说明
ShardingSphereYamlConstruct	用于将定制化对象和 YAML 相互转化

已知实现类	详细说明
NoneShardingSt rategyConfigurationYamlConstruct	用于将不分片策略对象和 YAML 相互转化

8.3 内核

8.3.1 SQLRouter

SPI 名称	详细说明
SQLRouter	用于处理路由结果

已知实现类	详细说明
ReadwriteSplittingSQLRouter	用于处理读写分离路由结果
DatabaseDiscoverySQLRouter	用于处理数据库发现路由结果
SingleTableSQLRouter	用于处理单表路由结果
ShardingSQLRouter	用于处理分片路由结果
ShadowSQLRouter	用于处理影子库路由结果

8.3.2 SQLRewriteContextDecorator

SPI 名称	详细说明
SQLRewriteContextDecorator	用于处理 SQL 改写结果

已知实现类	详细说明
ShardingSQLRewriteContextDecorator	用于处理分片 SQL 改写结果
EncryptSQLRewriteContextDecorator	用于处理加密 SQL 改写结果

8.3.3 SQLExecutionHook

SPI 名称	详细说明
SQLExecutionHook	SQL 执行过程监听器

已知实现类	详细说明
TransactionalSQLExecutionHook	基于事务的 SQL 执行过程监听器

8.3.4 ResultProcessEngine

SPI 名称	详细说明
ResultProcessEngine	用于处理结果集

已知实现类	详细说明
ShardingResultMergerEngine	用于处理分片结果集归并
EncryptResultDecoratorEngine	用于处理加密结果集改写

8.3.5 StoragePrivilegeHandler

SPI 名称	详细说明
StoragePrivilegeHandler	使用数据库方言处理权限信息

已知实现类	详细说明
PostgreSQLPrivilegeHandler	使用 PostgreSQL 方言处理权限信息
SQLServerPrivilegeHandler	使用 SQLServer 方言处理权限信息
OraclePrivilegeHandler	使用 Oracle 方言处理权限信息
MySQLPrivilegeHandler	使用 MySQL 方言处理权限信息

8.4 数据源

8.4.1 DatabaseType

SPI 名称	详细说明
DatabaseType	支持的数据库类型

已知实现类	详细说明
SQL92DatabaseType	遵循 SQL92 标准的数据库类型
MySQLDatabaseType	MySQL 数据库
MariaDBDatabaseType	MariaDB 数据库
PostgreSQLDatabaseType	PostgreSQL 数据库
OracleDatabaseType	Oracle 数据库
SQLServerDatabaseType	SQLServer 数据库
H2DatabaseType	H2 数据库
OpenGaussDatabaseType	OpenGauss 数据库

8.4.2 DialectTableMetaDataLoader

SPI 名称	详细说明
DialectTableMetaDataLoader	用于使用数据库方言快速加载元数据

已知实现类	详细说明
MySQLTableMetaDataLoader	使用 MySQL 方言加载元数据
OracleTableMetaDataLoader	使用 Oracle 方言加载元数据
PostgreSQLTableMetaDataLoader	使用 PostgreSQL 方言加载元数据
SQLServerTableMetaDataLoader	使用 SQLServer 方言加载元数据
H2TableMetaDataLoader	使用 H2 方言加载元数据
OpenGaussTableMetaDataLoader	使用 OpenGauss 方言加载元数据

8.4.3 DataSourcePoolMetaData

SPI 名称	详细说明
DataSourcePoolMetaData	数据源连接池元数据

已知实现类	详细说明
DBCPDataSourcePoolMetaData	DBCP 数据库连接池元数据
HikariDataSourcePoolMetaData	Hikari 数据源连接池元数据

8.4.4 DataSourcePoolActiveDetector

SPI 名称	详细说明
DataSourcePoolActiveDetector	数据源连接池活跃探测器

已知实现类	详细说明
DefaultDataSourcePoolActiveDetector	默认数据源连接池活跃探测器
HikariDataSourcePoolActiveDetector	Hikari 数据源连接池活跃探测器

8.5 SQL 解析

8.5.1 DatabaseTypedSQLParserFacade

SPI 名称	详细说明
DatabaseTypedSQLParserFacade	配置用于 SQL 解析的词法分析器和语法分析器入口

Implementation Class	*Description*
MySQLParserFacade	基于 MySQL 的 SQL 解析器入口
PostgreSQLParserFacade	基于 PostgreSQL 的 SQL 解析器入口
SQLServerParserFacade	基于 SQLServer 的 SQL 解析器入口
OracleParserFacade	基于 Oracle 的 SQL 解析器入口
SQL92ParserFacade	基于 SQL92 的 SQL 解析器入口
OpenGaussParserFacade	基于 openGauss 的 SQL 解析器入口

8.5.2 SQLVisitorFacade

SPI 名称	详细说明
SQLVisitorFacade	SQL 语法树访问器入口

Implementation Class	*Description*
MySQLStatementSQLVisitorFacade	基于 MySQL 的提取 SQL 语句的语法树访问器
Pos tgreSQLStatementSQLVisitorFacade	基于 PostgreSQL 的提取 SQL 语句的语法树访问器
SQ LServerStatementSQLVisitorFacade	基于 SQLServer 的提取 SQL 语句的语法树访问器
OracleStatementSQLVisitorFacade	基于 Oracle 的提取 SQL 语句的语法树访问器
SQL92StatementSQLVisitorFacade	基于 SQL92 的提取 SQL 语句的语法树访问器
Op enGaussStatementSQLVisitorFacade	基于 openGauss 的提取 SQL 语句的语法树访问器

8.6 代理端

8.6.1 DatabaseProtocolFrontendEngine

SPI 名称	详细说明
DatabaseProtocolFrontendEngine	用于 ShardingSphere-Proxy 解析与适配访问数据库的协议

已知实现类	详细说明
MySQLFrontendEngine	基于 MySQL 的数据库协议实现
PostgreSQLFrontendEngine	基于 PostgreSQL 的数据库协议实现
OpenGaussFrontendEngine	基于 openGauss 的数据库协议实现

8.6.2 JDBCDriverURLRecognizer

SPI 名称	详细说明
JDBCDriverURLRecognizer	使用 JDBC 驱动执行 SQL

已知实现类	详细说明
MySQLRecognizer	使用 MySQL 的 JDBC 驱动执行 SQL
PostgreSQLRecognizer	使用 PostgreSQL 的 JDBC 驱动执行 SQL
OracleRecognizer	使用 Oracle 的 JDBC 驱动执行 SQL
SQLServerRecognizer	使用 SQLServer 的 JDBC 驱动执行 SQL
H2Recognizer	使用 H2 的 JDBC 驱动执行 SQL
P6SpyDriverRecognizer	使用 P6Spy 的 JDBC 驱动执行 SQL
OpenGaussRecognizer	使用 openGauss 的 JDBC 驱动执行 SQL

8.6.3 AuthorityProvideAlgorithm

SPI 名称	详细说明
AuthorityProvideAlgorithm	用户权限加载逻辑

已知实现类	*Type*	详细说明
NativeAuthorityProviderAlgorithm（已弃用）	NATIVE	基于后端数据库存取 server.yaml 中配置的权限信息。如果用户不存在，则自动创建用户并默认赋予最高权限。
AllPrivilegesPermittedAuthorityProviderAlgorithm	ALL_PRIVILEGES_PERMITTED	默认授予所有权限（不鉴权），不会与实际数据库交互。
SchemaPrivilegesPermittedAuthorityProviderAlgorithm	SCHEMA_PRIVILEGES_PERMITTED	通过属性 user-schema-mappings 配置的权限。

8.7 数据分片

8.7.1 ShardingAlgorithm

SPI 名称	详细说明
ShardingAlgorithm	分片算法

已知实现类	详细说明
BoundaryBasedRangeShardingAlgorithm	基于分片边界的范围分片算法
VolumeBasedRangeShardingAlgorithm	基于分片容量的范围分片算法
ComplexInlineShardingAlgorithm	基于行表达式的复合分片算法
AutoIntervalShardingAlgorithm	基于可变时间范围的分片算法
ClassBasedShardingAlgorithm	基于自定义类的分片算法
HintInlineShardingAlgorithm	基于行表达式的 Hint 分片算法
IntervalShardingAlgorithm	基于固定时间范围的分片算法
HashModShardingAlgorithm	基于哈希取模的分片算法
InlineShardingAlgorithm	基于行表达式的分片算法
ModShardingAlgorithm	基于取模的分片算法

8.7.2 KeyGenerateAlgorithm

SPI 名称	详细说明
KeyGenerateAlgorithm	分布式主键生成算法

已知实现类	详细说明
SnowflakeKeyGenerateAlgorithm	基于雪花算法的分布式主键生成算法
UUIDKeyGenerateAlgorithm	基于 UUID 的分布式主键生成算法

8.7.3 DatetimeService

SPI 名称	详细说明
DatetimeService	获取当前时间进行路由

已知实现类	详细说明
DatabaseDatetimeServiceDelegate	从数据库中获取当前时间进行路由
SystemDatetimeService	从应用系统时间中获取当前时间进行路由

8.7.4 DatabaseSQLEntry

SPI 名称	详细说明
DatabaseSQLEntry	获取当前时间的数据库方言

已知实现类	详细说明
MySQLDatabaseSQLEntry	从 MySQL 获取当前时间的数据库方言
PostgreSQLDatabaseSQLEntry	从 PostgreSQL 获取当前时间的数据库方言
OracleDatabaseSQLEntry	从 Oracle 获取当前时间的数据库方言
SQLServerDatabaseSQLEntry	从 SQLServer 获取当前时间的数据库方言

8.8 读写分离

8.8.1 ReadwriteSplittingType

SPI 名称	详细说明
ReadwriteSplittingType	读写分离类型

已知实现类	详细说明
StaticReadwriteSplittingType	静态读写分离类型
DynamicReadwriteSplittingType	动态读写分离类型

8.8.2 ReplicaLoadBalanceAlgorithm

SPI 名称	详细说明
ReplicaLoadBalanceAlgorithm	读库负载均衡算法

已知实现类	详细说明
RoundRobinReplicaLoadBalanceAlgorithm	基于轮询的读库负载均衡算法
RandomReplicaLoadBalanceAlgorithm	基于随机的读库负载均衡算法
WeightReplicaLoadBalanceAlgorithm	基于权重的读库负载均衡算法

8.9 高可用

8.9.1 DatabaseDiscoveryType

SPI 名称	详细说明
DatabaseDiscoveryType	数据库发现类型

已知实现类	详细说明
MGRDatabaseDiscoveryType	基于 MySQL MGR 的数据库发现
ShowSlaveStatusDatabaseDiscoveryType	基于 MySQL 主从延迟的数据库发现
OpenGaussDatabaseDiscoveryType	基于 openGauss 的数据库发现

8.10 分布式事务

8.10.1 ShardingSphereTransactionManager

SPI 名称	详细说明
ShardingSphereTransactionManager	分布式事务管理器

已知实现类	详细说明
XAShardingSphereTransactionManager	基于 XA 的分布式事务管理器
SeataATShardingSphereTransactionManager	基于 Seata 的分布式事务管理器

8.10.2 XATransactionManagerProvider

SPI 名称	详细说明
XATransactionManagerProvider	XA 分布式事务管理器

已知实现类	详细说明
AtomikosTransactionManagerProvider	基于 Atomikos 的 XA 分布式事务管理器
NarayanaXATransactionManagerProvider	基于 Narayana 的 XA 分布式事务管理器
BitronixXATransactionManagerProvider	基于 Bitronix 的 XA 分布式事务管理器

8.10.3 XADataSourceDefinition

SPI 名称	详细说明
XADataSourceDefinition	非 XA 数据源自动转化为 XA 数据源

已知实现类	详细说明
MySQ LXADataSourceDefinition	非 XA 的 MySQL 数据源自动转化为 XA 的 MySQL 数据源
MariaD BXADataSourceDefinition	非 XA 的 MariaDB 数据源自动转化为 XA 的 MariaDB 数据源
PostgreSQ LXADataSourceDefinition	非 XA 的 PostgreSQL 数据源自动转化为 XA 的 PostgreSQL 数据源
Oracl eXADataSourceDefinition	非 XA 的 Oracle 数据源自动转化为 XA 的 Oracle 数据源
SQLServe rXADataSourceDefinition	非 XA 的 SQLServer 数据源自动转化为 XA 的 SQLServer 数据源
H 2XADataSourceDefinition	非 XA 的 H2 数据源自动转化为 XA 的 H2 数据源

8.10.4 DataSourcePropertyProvider

SPI 名称	详细说明
DataSourcePropertyProvider	用于获取数据源连接池的标准属性

已知实现类	详细说明
HikariCPPropertyProvider	用于获取 HikariCP 连接池的标准属性

8.11 弹性伸缩

8.11.1 ScalingEntry

SPI 名称	详细说明
ScalingEntry	弹性伸缩入口

已知实现类	详细说明
MySQLScalingEntry	基于 MySQL 的弹性伸缩入口
PostgreSQLScalingEntry	基于 PostgreSQL 的弹性伸缩入口
OpenGaussScalingEntry	基于 openGauss 的弹性伸缩入口

8.11.2 JobCompletionDetectAlgorithm

SPI 名称	详细说明
JobCompletionDetectAlgorithm	作业是否接近完成检测算法

已知实现类	详细说明
IdleRuleAlt eredJobCompletionDetectAlgorithm	基于增量迁移任务空闲时长的检测算法

8.11.3 DataConsistencyCheckAlgorithm

SPI 名称	详细说明
DataConsistencyCheckAlgorithm	数据一致性校验算法

已知实现类	详细说明
DataMatch DataConsistencyCheckAlgorithm	基于数据匹配的一致性校验算法。类型名：DATA_MATCH
CRC32Match DataConsistencyCheckAlgorithm	基于数据 CRC32 匹配的一致性校验算法。类型名：CRC32_MATCH

8.11.4 SingleTableDataCalculator

SPI 名称	详细说明
SingleTableDataCalculator	给数据一致性校验使用的单表数据计算算法

已知实现类	详细说明
DataMatchS ingleTableDataCalculator	给 DATA_MATCH 数据一致性校验算法使用的单表数据计算算法。适用于所有数据库
CRC32MatchMySQLS ingleTableDataCalculator	给 CRC32_MATCH 数据一致性校验算法使用的单表数据计算算法。适用于 MySQL

8.12 SQL 检查

8.12.1 SQLChecker

SPI 名称	详细说明
SQLChecker	SQL 检查器

已知实现类	详细说明
AuthorityChecker	权限检查器

8.13 数据加密

8.13.1 EncryptAlgorithm

SPI 名称	详细说明
EncryptAlgorithm	数据加密算法

已知实现类	详细说明
MD5EncryptAlgorithm	基于 MD5 的数据加密算法
AESEncryptAlgorithm	基于 AES 的数据加密算法
RC4EncryptAlgorithm	基于 RC4 的数据加密算法
SM4EncryptAlgorithm	基于 SM4 的数据加密算法
SM3EncryptAlgorithm	基于 SM3 的数据加密算法

8.13.2 QueryAssistedEncryptAlgorithm

SPI 名称	详细说明
QueryAssistedEncryptAlgorithm	包含查询辅助列的数据加密算法

已知实现类	详细说明
无	

8.14 影子库

8.14.1 ShadowAlgorithm

SPI 名称	详细说明
ShadowAlgorithm	影子库路由算法

已知实现类	详细说明
ColumnValueMatchShadowAlgorithm	基于字段值匹配影子算法
ColumnRegexMatchShadowAlgorithm	基于字段值正则匹配影子算法
SimpleHintShadowAlgorithm	基于 Hint 简单匹配影子算法

8.15 可观察性

8.15.1 PluginDefinitionService

SPI 名称	详细说明
PluginDefinitionService	Agent 插件定义

已知实现类	详细说明
PrometheusPluginDefinitionService	Prometheus plugin
BaseLoggingPluginDefinitionService	Logging plugin
JaegerPluginDefinitionService	Jaeger plugin
OpenTelemetryTracingPluginDefinitionService	OpenTelemetryTracing plugin
OpenTracingPluginDefinitionService	OpenTracing plugin
ZipkinPluginDefinitionService	Zipkin plugin

8.15.2 PluginBootService

SPI 名称	详细说明
PluginBootService	插件启动服务定义

已知实现类	详细说明
PrometheusPluginBootService	Prometheus plugin 启动类
BaseLoggingPluginBootService	Logging plugin 启动类
JaegerTracingPluginBootService	Jaeger plugin 启动类
OpenTelemetryTracingPluginBootService	OpenTelemetryTracing plugin 启动类
OpenTracingPluginBootService	OpenTracing plugin 启动类
ZipkinTracingPluginBootService	Zipkin plugin 启动类

9 技术参考

本章包含了 Apache ShardingSphere 的技术实现细节和测试流程，供开发者和用户参考。

9.1 管控

9.1.1 注册中心数据结构

在定义的命名空间下，`rules`、`props` 和 `metadata` 节点以 YAML 格式存储配置，可通过修改节点来实现对于配置的动态管理。`nodes` 存储数据库访问对象运行节点，用于区分不同数据库访问实例。

```
namespace
   ├──rules                                    # 全局规则配置
   ├──props                                    # 属性配置
   ├──metadata                                 # Metadata 配置
   ├     ├──${schema_1}                        # Schema 名称 1
   ├     ├     ├──dataSources                  # 数据源配置
   ├     ├     ├──rules                        # 规则配置
   ├     ├     ├──tables                       # 表结构配置
   ├     ├     ├     ├──t_1
   ├     ├     ├     ├──t_2
   ├     ├──${schema_2}                        # Schema 名称 2
   ├     ├     ├──dataSources                  # 数据源配置
   ├     ├     ├──rules                        # 规则配置
   ├     ├     ├──tables                       # 表结构配置
   ├──nodes
   ├    ├──compute_nodes
   ├    ├     ├──online
   ├    ├     ├     ├──proxy
   ├    ├     ├     ├     ├──${your_instance_ip_a}@${your_instance_port_x}
   ├    ├     ├     ├     ├──${your_instance_ip_b}@${your_instance_port_y}
   ├    ├     ├     ├     ├──....
   ├    ├     ├     ├──jdbc
   ├    ├     ├     ├     ├──${your_instance_ip_a}@${your_instance_pid_x}
```

```
    ├    ├     ├     ├     ├──${your_instance_ip_b}@${your_instance_pid_y}
    ├    ├     ├     ├     ├──....
    ├    ├     ├──attributes
    ├    ├     ├     ├──${your_instance_ip_a}@${your_instance_port_x}
    ├    ├     ├     ├     ├──status
    ├    ├     ├     ├     ├──label
    ├    ├     ├     ├──${your_instance_ip_b}@${your_instance_pid_y}
    ├    ├     ├     ├     ├──status
    ├    ├     ├     ├──....
    ├    ├──storage_nodes
    ├    ├     ├──disable
    ├    ├     ├      ├──${schema_1.ds_0}
    ├    ├     ├      ├──${schema_1.ds_1}
    ├    ├     ├      ├──....
    ├    ├     ├──primary
    ├    ├     ├      ├──${schema_2.ds_0}
    ├    ├     ├      ├──${schema_2.ds_1}
    ├    ├     ├      ├──....
```

/rules

全局规则配置，可包括访问 ShardingSphere-Proxy 用户名和密码的权限配置。

```
- !AUTHORITY
users:
  - root@%:root
  - sharding@127.0.0.1:sharding
provider:
  type: ALL_PRIVILEGES_PERMITTED
```

/props

属性配置，详情请参见配置手册。

```
kernel-executor-size: 20
sql-show: true
```

/metadata/${schemaName}/dataSources

多个数据库连接池的集合，不同数据库连接池属性自适配（例如：DBCP，C3P0，Druid，HikariCP）。

```
ds_0:
  initializationFailTimeout: 1
  validationTimeout: 5000
  maxLifetime: 1800000
  leakDetectionThreshold: 0
  minimumIdle: 1
  password: root
  idleTimeout: 60000
  jdbcUrl: jdbc:mysql://127.0.0.1:3306/ds_0?serverTimezone=UTC&useSSL=false
  dataSourceClassName: com.zaxxer.hikari.HikariDataSource
  maximumPoolSize: 50
  connectionTimeout: 30000
  username: root
  poolName: HikariPool-1
ds_1:
  initializationFailTimeout: 1
  validationTimeout: 5000
  maxLifetime: 1800000
  leakDetectionThreshold: 0
  minimumIdle: 1
  password: root
  idleTimeout: 60000
  jdbcUrl: jdbc:mysql://127.0.0.1:3306/ds_1?serverTimezone=UTC&useSSL=false
  dataSourceClassName: com.zaxxer.hikari.HikariDataSource
  maximumPoolSize: 50
  connectionTimeout: 30000
  username: root
  poolName: HikariPool-2
```

/metadata/${schemaName}/rules

规则配置，可包括数据分片、读写分离、数据加密、影子库压测等配置。

```
- !SHARDING
  xxx

- !READWRITE_SPLITTING
  xxx

- !ENCRYPT
  xxx
```

/metadata/${schemaName}/tables

表结构配置，每个表使用单独节点存储，暂不支持动态修改。

```
name: t_order                                   # 表名
columns:                                        # 列
  id:                                           # 列名
    caseSensitive: false
    dataType: 0
    generated: false
    name: id
    primaryKey: trues
  order_id:
    caseSensitive: false
    dataType: 0
    generated: false
    name: order_id
    primaryKey: false
indexs:                                         # 索引
  t_user_order_id_index:                        # 索引名
    name: t_user_order_id_index
```

/nodes/compute_nodes

数据库访问对象运行实例信息，子节点是当前运行实例的标识。运行实例标识由运行服务器的 IP 地址和 PORT 构成。运行实例标识均为临时节点，当实例上线时注册，下线时自动清理。注册中心监控这些节点的变化来治理运行中实例对数据库的访问等。

/nodes/storage_nodes

可以治理读写分离从库，可动态添加删除以及禁用。

9.2 数据分片

ShardingSphere 的 3 个产品的数据分片主要流程是完全一致的，按照是否进行查询优化，可以分为 Standard 内核流程和 Federation 执行引擎流程。Standard 内核流程由 `SQL 解析 => SQL 路由 => SQL 改写 => SQL 执行 => 结果归并组成`，主要用于处理标准分片场景下的 SQL 执行。Federation 执行引擎流程由 `SQL 解析 => 逻辑优化 => 物理优化 => 优化执行 => Standard 内核流程组成`，Federation 执行引擎内部进行逻辑优化和物理优化，在优化执行阶段依赖 Standard 内核流程，对优化后的逻辑 SQL 进行路由、改写、执行和归并。

图 1: 分片架构图

9.2.1 SQL 解析

分为词法解析和语法解析。先通过词法解析器将 SQL 拆分为一个个不可再分的单词。再使用语法解析器对 SQL 进行理解，并最终提炼出解析上下文。解析上下文包括表、选择项、排序项、分组项、聚合函数、分页信息、查询条件以及可能需要修改的占位符的标记。

9.2.2 SQL 路由

根据解析上下文匹配用户配置的分片策略，并生成路由路径。目前支持分片路由和广播路由。

9.2.3 SQL 改写

将 SQL 改写为在真实数据库中可以正确执行的语句。SQL 改写分为正确性改写和优化改写。

9.2.4 SQL 执行

通过多线程执行器异步执行。

9.2.5 结果归并

将多个执行结果集归并以便于通过统一的 JDBC 接口输出。结果归并包括流式归并、内存归并和使用装饰者模式的追加归并这几种方式。

9.2.6 查询优化

由 Federation 执行引擎（开发中）提供支持，对关联查询、子查询等复杂查询进行优化，同时支持跨多个数据库实例的分布式查询，内部使用关系代数优化查询计划，通过最优计划查询出结果。

9.2.7 解析引擎

相对于其他编程语言，SQL 是比较简单的。不过，它依然是一门完善的编程语言，因此对 SQL 的语法进行解析，与解析其他编程语言（如：Java 语言、C 语言、Go 语言等）并无本质区别。

抽象语法树

解析过程分为词法解析和语法解析。词法解析器用于将 SQL 拆解为不可再分的原子符号，称为 Token。并根据不同数据库方言所提供的字典，将其归类为关键字，表达式，字面量和操作符。再使用语法解析器将词法解析器的输出转换为抽象语法树。

例如，以下 SQL：

```
SELECT id, name FROM t_user WHERE status = 'ACTIVE' AND age > 18
```

解析之后的为抽象语法树见下图。

图 2: SQL 抽象语法树

为了便于理解，抽象语法树中的关键字的 Token 用绿色表示，变量的 Token 用红色表示，灰色表示需要进一步拆分。

最后，通过 `visitor` 对抽象语法树遍历构造域模型，通过域模型（`SQLStatement`）去提炼分片所需的上下文，并标记有可能需要改写的位置。供分片使用的解析上下文包含查询选择项（Select Items）、表信息（Table）、分片条件（Sharding Condition）、自增主键信息（Auto increment Primary Key）、排序信息（Order By）、分组信息（Group By）以及分页信息（Limit、Rownum、Top）。SQL 的一次解析过程是不可逆的，一个个 Token 按 SQL 原本的顺序依次进行解析，性能很高。考虑到各种数据库 SQL 方言的异同，在解析模块提供了各类数据库的 SQL 方言字典。

SQL 解析引擎

历史

SQL 解析作为分库分表类产品的核心，其性能和兼容性是最重要的衡量指标。ShardingSphere 的 SQL 解析器经历了 3 代产品的更新迭代。

第一代 SQL 解析器为了追求性能与快速实现，在 1.4.x 之前的版本使用 Druid 作为 SQL 解析器。经实际测试，它的性能远超其它解析器。

第二代 SQL 解析器从 1.5.x 版本开始，ShardingSphere 采用完全自研的 SQL 解析引擎。由于目的不同，ShardingSphere 并不需要将 SQL 转为一颗完全的抽象语法树，也无需通过访问器模式进行二次遍历。它采用对 SQL 半理解的方式，仅提炼数据分片需要关注的上下文，因此 SQL 解析的性能和兼容性得到了进一步的提高。

第三代 SQL 解析器从 3.0.x 版本开始，尝试使用 ANTLR 作为 SQL 解析引擎的生成器，并采用 Visit 的方式从 AST 中获取 SQL Statement。从 5.0.x 版本开始，解析引擎的架构已完成重构调整，同时通过将第一次解析得到的 AST 放入缓存，方便下次直接获取相同 SQL 的解析结果，来提高解析效率。因此我们建议用户采用 `PreparedStatement` 这种 SQL 预编译的方式来提升性能。

功能点

- 提供独立的 SQL 解析功能
- 可以非常方便的对语法规则进行扩充和修改（使用了 `ANTLR`）
- 支持多种方言的 SQL 解析

数据库	支持状态
MySQL	支持，完善
PostgreSQL	支持，完善
SQLServer	支持
Oracle	支持
SQL92	支持
openGauss	支持

- 提供 SQL 格式化功能（开发中）
- 提供 SQL 模板化功能（开发中）

API 使用

引入 Maven 依赖

```
<dependency>
    <groupId>org.apache.shardingsphere</groupId>
    <artifactId>shardingsphere-sql-parser-engine</artifactId>
    <version>${project.version}</version>
</dependency>
<!-- 根据需要引入指定方言的解析模块（以 MySQL 为例），可以添加所有支持的方言，也可以只添加使用到的
-->
<dependency>
    <groupId>org.apache.shardingsphere</groupId>
    <artifactId>shardingsphere-sql-parser-mysql</artifactId>
    <version>${project.version}</version>
</dependency>
```

例子

- 获取语法树

```
/**
 * databaseType type:String 可能值 MySQL, Oracle, PostgreSQL, SQL92, SQLServer,
openGauss
 * sql type:String 解析的 SQL
 * useCache type:boolean 是否使用缓存
 * @return parse context
 */
ParseContext parseContext = new SQLParserEngine(databaseType).parse(sql, useCache);
```

- 获取 SQLStatement

```
/**
 * databaseType type:String 可能指 MySQL, Oracle, PostgreSQL, SQL92, SQLServer,
openGauss
 * useCache type:boolean 是否使用缓存
 * @return SQLStatement
 */
ParseContext parseContext = new SQLParserEngine(databaseType).parse(sql, useCache);
SQLVisitorEngine sqlVisitorEngine = new SQLVisitorEngine(databaseType, "STATEMENT
");
SQLStatement sqlStatement = sqlVisitorEngine.visit(parseContext);
```

- SQL 格式化

```
/**
 * databaseType type:String 可能指 MySQL
 * useCache type:boolean 是否使用缓存
 * @return String
 */
```

```
ParseContext parseContext = new SQLParserEngine(databaseType).parse(sql, useCache);
SQLVisitorEngine sqlVisitorEngine = new SQLVisitorEngine(databaseType, "FORMAT",
new Properties());
String formatedSql = sqlVisitorEngine.visit(parseContext);
```

例子:

sql	formatedSql
select a+1 as b, name n from table1 join table2 where id=1 and name= ‘lu’ ;	SELECT a + 1 AS b, name nFROM table1 JOIN table2WHERE id = 1 and name = ‘lu’ ;
select id, name, age, sex, ss, yy from table1 where id=1;	SELECT id , name , age , sex , ss , yy FROM table1WHERE id = 1;
select id, name, age, count(*) as n, (select id, name, age, sex from table2 where id=2) as sid, yyyy from table1 where id=1;	SELECT id , name , age , COUNT(*) AS n, (SELECT id , name , age , sex FROM table2 WHERE id = 2) AS sid, yyyy FROM table1WHERE id = 1;
select id, name, age, sex, ss, yy from table1 where id=1 and name=1 and a=1 and b=2 and c=4 and d=3;	SELECT id , name , age , sex , ss , yy FROM table1WHERE id = 1 and name = 1 and a = 1 and b = 2 and c = 4 and d = 3;
ALTER TABLE t_order ADD column4 DATE, ADD column5 DATETIME, engine ss max_rows 10,min_rows 2, ADD column6 TIMESTAMP, ADD column7 TIME;	ALTER TABLE t_order ADD column4 DATE, ADD column5 DATETIME, ENGINE ss MAX_ROWS 10, MIN_ROWS 2, ADD column6 TIMESTAMP, ADD column7 TIME
CREATE TABLE IF NOT EXISTS “ runoob_tbl“(runoob_id INT UNSIGNED **AUTO_** INCREMENT,runoob_title VARCHAR(100) NOT NULL,runoob_author VARCHAR(40) NOT NULL,runoob_test NATIONAL CHAR(40),submission_date DATE,PRIMARY KEY (runoob_id))ENGINE=InnoDB DEFAULT CHARSET=utf8;	CREATE TABLE IF NOT EXISTS runoob_tbl (runoob_id INT UNSIGNED AUTO_INCREMENT, runoob_title VARCHAR(100) NOT NULL, runoob_author VARCHAR(40) NOT NULL, runoob_test NATIONAL CHAR(40), submission_date DATE, PRIMARY KEY (runoob_id)) ENGINE = InnoDB DEFAULT CHARSET = utf8;
INSERT INTO t_order_item(order_id, user_id, status, creation_date) values (1, 1, ‘insert’ , ‘2017-08-08’), (2, 2, ‘insert’ , ‘2017-08-08’) ON DUPLICATE KEY UPDATE status = ‘init’ ;	INSERT INTO t_order_item (order_id , user_id , status , creation_date)VALUES (1, 1, ‘insert’ , ‘2017-08-08’), (2, 2, ‘insert’ , ‘2017-08-08’)ON DUPLICATE KEY UPDATE status = ‘init’ ;
INSERT INTO t_order SET order_id = 1, user_id = 1, status = convert(to_base64(aes_encrypt(1, ‘key’)) USING utf8) ON DUPLICATE KEY UPDATE status = VALUES(status);	INSERT INTO t_order SET order_id = 1, user_id = 1, status = CONVERT(to_base64(aes_encrypt(1 , ‘key’)) USING utf8)ON DUPLICATE KEY UPDATE status = VALUES(status);
INSERT INTO t_order (order_id, user_id, status) SELECT order_id, user_id, status FROM t_order WHERE order_id = 1;	INSERT INTO t_order (order_id , user_id , status) SELECT order_id , user_id , status FROM t_orderWHERE order_id = 1;

9.2.8 路由引擎

根据解析上下文匹配数据库和表的分片策略，并生成路由路径。对于携带分片键的 SQL，根据分片键的不同可以划分为单片路由（分片键的操作符是等号）、多片路由（分片键的操作符是 IN）和范围路由（分片键的操作符是 BETWEEN）。不携带分片键的 SQL 则采用广播路由。

分片策略通常可以采用由数据库内置或由用户方配置。数据库内置的方案较为简单，内置的分片策略大致可分为尾数取模、哈希、范围、标签、时间等。由用户方配置的分片策略则更加灵活，可以根据使用方需求定制复合分片策略。如果配合数据自动迁移来使用，可以做到无需用户关注分片策略，自动由数据库中间层分片和平衡数据即可，进而做到使分布式数据库具有的弹性伸缩的能力。在 ShardingSphere 的线路规划中，弹性伸缩将于 4.x 开启。

分片路由

用于根据分片键进行路由的场景，又细分为直接路由、标准路由和笛卡尔积路由这 3 种类型。

直接路由

满足直接路由的条件相对苛刻，它需要通过 Hint（使用 HintAPI 直接指定路由至库表）方式分片，并且是只分库不分表的前提下，则可以避免 SQL 解析和之后的结果归并。因此它的兼容性最好，可以执行包括子查询、自定义函数等复杂情况的任意 SQL。直接路由还可以用于分片键不在 SQL 中的场景。例如，设置用于数据库分片的键为 3，

```
hintManager.setDatabaseShardingValue(3);
```

假如路由算法为 `value % 2`，当一个逻辑库 `t_order` 对应 2 个真实库 `t_order_0` 和 `t_order_1` 时，路由后 SQL 将在 `t_order_1` 上执行。下方是使用 API 的代码样例：

```
String sql = "SELECT * FROM t_order";
try (
        HintManager hintManager = HintManager.getInstance();
        Connection conn = dataSource.getConnection();
        PreparedStatement pstmt = conn.prepareStatement(sql)) {
    hintManager.setDatabaseShardingValue(3);
    try (ResultSet rs = pstmt.executeQuery()) {
        while (rs.next()) {
            //...
        }
    }
}
```

标准路由

标准路由是 ShardingSphere 最为推荐使用的分片方式，它的适用范围是不包含关联查询或仅包含绑定表之间关联查询的 SQL。当分片运算符是等于号时，路由结果将落入单库（表），当分片运算符是 BETWEEN 或 IN 时，则路由结果不一定落入唯一的库（表），因此一条逻辑 SQL 最终可能被拆分为多条用于执行的真实 SQL。举例说明，如果按照 `order_id` 的奇数和偶数进行数据分片，一个单表查询的 SQL 如下：

```
SELECT * FROM t_order WHERE order_id IN (1, 2);
```

那么路由的结果应为：

```
SELECT * FROM t_order_0 WHERE order_id IN (1, 2);
SELECT * FROM t_order_1 WHERE order_id IN (1, 2);
```

绑定表的关联查询与单表查询复杂度和性能相当。举例说明，如果一个包含绑定表的关联查询的 SQL 如下：

```
SELECT * FROM t_order o JOIN t_order_item i ON o.order_id=i.order_id  WHERE order_
id IN (1, 2);
```

那么路由的结果应为：

```
SELECT * FROM t_order_0 o JOIN t_order_item_0 i ON o.order_id=i.order_id  WHERE
order_id IN (1, 2);
SELECT * FROM t_order_1 o JOIN t_order_item_1 i ON o.order_id=i.order_id  WHERE
order_id IN (1, 2);
```

可以看到，SQL 拆分的数目与单表是一致的。

笛卡尔路由

笛卡尔路由是最复杂的情况，它无法根据绑定表的关系定位分片规则，因此非绑定表之间的关联查询需要拆解为笛卡尔积组合执行。如果上个示例中的 SQL 并未配置绑定表关系，那么路由的结果应为：

```
SELECT * FROM t_order_0 o JOIN t_order_item_0 i ON o.order_id=i.order_id  WHERE
order_id IN (1, 2);
SELECT * FROM t_order_0 o JOIN t_order_item_1 i ON o.order_id=i.order_id  WHERE
order_id IN (1, 2);
SELECT * FROM t_order_1 o JOIN t_order_item_0 i ON o.order_id=i.order_id  WHERE
order_id IN (1, 2);
SELECT * FROM t_order_1 o JOIN t_order_item_1 i ON o.order_id=i.order_id  WHERE
order_id IN (1, 2);
```

笛卡尔路由查询性能较低，需谨慎使用。

广播路由

对于不携带分片键的 SQL，则采取广播路由的方式。根据 SQL 类型又可以划分为全库表路由、全库路由、全实例路由、单播路由和阻断路由这 5 种类型。

全库表路由

全库表路由用于处理对数据库中与其逻辑表相关的所有真实表的操作，主要包括不带分片键的 DQL 和 DML，以及 DDL 等。例如：

```
SELECT * FROM t_order WHERE good_prority IN (1, 10);
```

则会遍历所有数据库中的所有表，逐一匹配逻辑表和真实表名，能够匹配得上则执行。路由后成为

```
SELECT * FROM t_order_0 WHERE good_prority IN (1, 10);
SELECT * FROM t_order_1 WHERE good_prority IN (1, 10);
SELECT * FROM t_order_2 WHERE good_prority IN (1, 10);
SELECT * FROM t_order_3 WHERE good_prority IN (1, 10);
```

全库路由

全库路由用于处理对数据库的操作，包括用于库设置的 SET 类型的数据库管理命令，以及 TCL 这样的事务控制语句。在这种情况下，会根据逻辑库的名字遍历所有符合名字匹配的真实库，并在真实库中执行该命令，例如：

```
SET autocommit=0;
```

在 `t_order` 中执行，`t_order` 有 2 个真实库。则实际会在 `t_order_0` 和 `t_order_1` 上都执行这个命令。

全实例路由

全实例路由用于 DCL 操作，授权语句针对的是数据库的实例。无论一个实例中包含多少个 Schema，每个数据库的实例只执行一次。例如：

```
CREATE USER customer@127.0.0.1 identified BY '123';
```

这个命令将在所有的真实数据库实例中执行，以确保 customer 用户可以访问每一个实例。

单播路由

单播路由用于获取某一真实表信息的场景，它仅需要从任意库中的任意真实表中获取数据即可。例如：

```
DESCRIBE t_order;
```

t_order 的两个真实表 t_order_0，t_order_1 的描述结构相同，所以这个命令在任意真实表上选择执行一次。

阻断路由

阻断路由用于屏蔽 SQL 对数据库的操作，例如：

```
USE order_db;
```

这个命令不会在真实数据库中执行，因为 ShardingSphere 采用的是逻辑 Schema 的方式，无需将切换数据库 Schema 的命令发送至数据库中。

路由引擎的整体结构划分如下图。

图 3: 路由引擎结构

9.2.9 改写引擎

工程师面向逻辑库与逻辑表书写的 SQL，并不能够直接在真实的数据库中执行，SQL 改写用于将逻辑 SQL 改写为在真实数据库中可以正确执行的 SQL。它包括正确性改写和优化改写两部分。

正确性改写

在包含分表的场景中，需要将分表配置中的逻辑表名称改写为路由之后所获取的真实表名称。仅分库则不需要表名称的改写。除此之外，还包括补列和分页信息修正等内容。

标识符改写

需要改写的标识符包括表名称、索引名称以及 Schema 名称。

表名称改写是指将找到逻辑表在原始 SQL 中的位置，并将其改写为真实表的过程。表名称改写是一个典型的需要对 SQL 进行解析的场景。从一个最简单的例子开始，若逻辑 SQL 为：

```
SELECT order_id FROM t_order WHERE order_id=1;
```

假设该 SQL 配置分片键 order_id，并且 order_id=1 的情况，将路由至分片表 1。那么改写之后的 SQL 应该为：

```
SELECT order_id FROM t_order_1 WHERE order_id=1;
```

在这种最简单的 SQL 场景中，是否将 SQL 解析为抽象语法树似乎无关紧要，只要通过字符串查找和替换就可以达到 SQL 改写的效果。但是下面的场景，就无法仅仅通过字符串的查找替换来正确的改写 SQL 了：

```
SELECT order_id FROM t_order WHERE order_id=1 AND remarks=' t_order xxx';
```

正确改写的 SQL 应该是：

```
SELECT order_id FROM t_order_1 WHERE order_id=1 AND remarks=' t_order xxx';
```

而非：

```
SELECT order_id FROM t_order_1 WHERE order_id=1 AND remarks=' t_order_1 xxx';
```

由于表名之外可能含有表名称的类似字符，因此不能通过简单的字符串替换的方式去改写 SQL。

下面再来看一个更加复杂的 SQL 改写场景：

```
SELECT t_order.order_id FROM t_order WHERE t_order.order_id=1 AND remarks=' t_order
xxx';
```

上面的 SQL 将表名作为字段的标识符，因此在 SQL 改写时需要一并修改：

```
SELECT t_order_1.order_id FROM t_order_1 WHERE t_order_1.order_id=1 AND remarks='
t_order xxx';
```

而如果 SQL 中定义了表的别名，则无需连同别名一起修改，即使别名与表名相同亦是如此。例如：

```
SELECT t_order.order_id FROM t_order AS t_order WHERE t_order.order_id=1 AND
remarks=' t_order xxx';
```

SQL 改写则仅需要改写表名称就可以了：

```
SELECT t_order.order_id FROM t_order_1 AS t_order WHERE t_order.order_id=1 AND
remarks=' t_order xxx';
```

索引名称是另一个有可能改写的标识符。在某些数据库中（如 MySQL、SQLServer），索引是以表为维度创建的，在不同的表中的索引是可以重名的；而在另外的一些数据库中（如 PostgreSQL、Oracle），索引是以数据库为维度创建的，即使是作用在不同表上的索引，它们也要求其名称的唯一性。

在 ShardingSphere 中，管理 Schema 的方式与管理表如出一辙，它采用逻辑 Schema 去管理一组数据源。因此，ShardingSphere 需要将用户在 SQL 中书写的逻辑 Schema 替换为真实的数据库 Schema。

ShardingSphere 目前还不支持在 DQL 和 DML 语句中使用 Schema。它目前仅支持在数据库管理语句中使用 Schema，例如：

```
SHOW COLUMNS FROM t_order FROM order_ds;
```

Schema 的改写指的是将逻辑 Schema 采用单播路由的方式，改写为随机查找到的一个正确的真实 Schema。

补列

需要在查询语句中补列通常由两种情况导致。第一种情况是 ShardingSphere 需要在结果归并时获取相应数据，但该数据并未能通过查询的 SQL 返回。这种情况主要是针对 GROUP BY 和 ORDER BY。结果归并时，需要根据 `GROUP BY` 和 `ORDER BY` 的字段项进行分组和排序，但如果原始 SQL 的选择项中若并未包含分组项或排序项，则需要对原始 SQL 进行改写。先看一下原始 SQL 中带有结果归并所需信息的场景：

```
SELECT order_id, user_id FROM t_order ORDER BY user_id;
```

由于使用 user_id 进行排序，在结果归并中需要能够获取到 user_id 的数据，而上面的 SQL 是能够获取到 user_id 数据的，因此无需补列。

如果选择项中不包含结果归并时所需的列，则需要进行补列，如以下 SQL：

```
SELECT order_id FROM t_order ORDER BY user_id;
```

由于原始 SQL 中并不包含需要在结果归并中需要获取的 user_id，因此需要对 SQL 进行补列改写。补列之后的 SQL 是：

```
SELECT order_id, user_id AS ORDER_BY_DERIVED_0 FROM t_order ORDER BY user_id;
```

值得一提的是，补列只会补充缺失的列，不会全部补充，而且，在 SELECT 语句中包含 * 的 SQL，也会根据表的元数据信息选择性补列。下面是一个较为复杂的 SQL 补列场景：

```
SELECT o.* FROM t_order o, t_order_item i WHERE o.order_id=i.order_id ORDER BY
user_id, order_item_id;
```

我们假设只有 t_order_item 表中包含 order_item_id 列，那么根据表的元数据信息可知，在结果归并时，排序项中的 user_id 是存在于 t_order 表中的，无需补列；order_item_id 并不在 t_order 中，因此需要补列。补列之后的 SQL 是：

```
SELECT o.*, order_item_id AS ORDER_BY_DERIVED_0 FROM t_order o, t_order_item i
WHERE o.order_id=i.order_id ORDER BY user_id, order_item_id;
```

补列的另一种情况是使用 AVG 聚合函数。在分布式的场景中，使用 avg1 + avg2 + avg3 / 3 计算平均值并不正确，需要改写为 (sum1 + sum2 + sum3) / (count1 + count2 + count3)。这就需要将包含 AVG 的 SQL 改写为 SUM 和 COUNT，并在结果归并时重新计算平均值。例如以下 SQL：

```
SELECT AVG(price) FROM t_order WHERE user_id=1;
```

需要改写为：

```
SELECT COUNT(price) AS AVG_DERIVED_COUNT_0, SUM(price) AS AVG_DERIVED_SUM_0 FROM t_
order WHERE user_id=1;
```

然后才能够通过结果归并正确的计算平均值。

最后一种补列是在执行 INSERT 的 SQL 语句时，如果使用数据库自增主键，是无需写入主键字段的。但数据库的自增主键是无法满足分布式场景下的主键唯一的，因此 ShardingSphere 提供了分布式自增主键的生成策略，并且可以通过补列，让使用方无需改动现有代码，即可将分布式自增主键透明的替换数据库现有的自增主键。分布式自增主键的生成策略将在下文中详述，这里只阐述与 SQL 改写相关的内容。举例说明，假设表 t_order 的主键是 order_id，原始的 SQL 为：

```
INSERT INTO t_order (`field1`, `field2`) VALUES (10, 1);
```

可以看到，上述 SQL 中并未包含自增主键，是需要数据库自行填充的。ShardingSphere 配置自增主键后，SQL 将改写为：

```
INSERT INTO t_order (`field1`, `field2`, order_id) VALUES (10, 1, xxxxx);
```

改写后的 SQL 将在 INSERT FIELD 和 INSERT VALUE 的最后部分增加主键列名称以及自动生成的自增主键值。上述 SQL 中的 `xxxxx` 表示自动生成的自增主键值。

如果 INSERT 的 SQL 中并未包含表的列名称，ShardingSphere 也可以根据判断参数个数以及表元信息中的列数量对比，并自动生成自增主键。例如，原始的 SQL 为：

```
INSERT INTO t_order VALUES (10, 1);
```

改写的 SQL 将只在主键所在的列顺序处增加自增主键即可：

```
INSERT INTO t_order VALUES (xxxxx, 10, 1);
```

自增主键补列时，如果使用占位符的方式书写 SQL，则只需要改写参数列表即可，无需改写 SQL 本身。

分页修正

从多个数据库获取分页数据与单数据库的场景是不同的。假设每 10 条数据为一页，取第 2 页数据。在分片环境下获取 LIMIT 10, 10，归并之后再根据排序条件取出前 10 条数据是不正确的。举例说明，若 SQL 为：

```
SELECT score FROM t_score ORDER BY score DESC LIMIT 1, 2;
```

下图展示了不进行 SQL 的改写的分页执行结果。

图 4: 不改写 SQL 的分页执行结果

通过图中所示，想要取得两个表中共同的按照分数排序的第 2 条和第 3 条数据，应该是 `95` 和 `90`。由于执行的 SQL 只能从每个表中获取第 2 条和第 3 条数据，即从 t_score_0 表中获取的是 `90` 和 `80`；从 t_score_1 表中获取的是 `85` 和 `75`。因此进行结果归并时，只能从获取的 `90`，`80`，`85` 和 `75` 之中进行归并，那么结果归并无论怎么实现，都不可能获得正确的结果。

正确的做法是将分页条件改写为 `LIMIT 0, 3`，取出所有前两页数据，再结合排序条件计算出正确的数据。下图展示了进行 SQL 改写之后的分页执行结果。

越获取偏移量位置靠后数据，使用 LIMIT 分页方式的效率就越低。有很多方法可以避免使用 LIMIT 进行分页。比如构建行记录数量与行偏移量的二级索引，或使用上次分页数据结尾 ID 作为下次查询条件的分页方式等。

图 5: 改写 SQL 的分页执行结果

分页信息修正时，如果使用占位符的方式书写 SQL，则只需要改写参数列表即可，无需改写 SQL 本身。

批量拆分

在使用批量插入的 SQL 时，如果插入的数据是跨分片的，那么需要对 SQL 进行改写来防止将多余的数据写入到数据库中。插入操作与查询操作的不同之处在于，查询语句中即使用了不存在于当前分片的分片键，也不会对数据产生影响；而插入操作则必须将多余的分片键删除。举例说明，如下 SQL：

```
INSERT INTO t_order (order_id, xxx) VALUES (1, 'xxx'), (2, 'xxx'), (3, 'xxx');
```

假设数据库仍然是按照 order_id 的奇偶值分为两片的，仅将这条 SQL 中的表名进行修改，然后发送至数据库完成 SQL 的执行，则两个分片都会写入相同的记录。虽然只有符合分片查询条件的数据才能够被查询语句取出，但存在冗余数据的实现方案并不合理。因此需要将 SQL 改写为：

```
INSERT INTO t_order_0 (order_id, xxx) VALUES (2, 'xxx');
INSERT INTO t_order_1 (order_id, xxx) VALUES (1, 'xxx'), (3, 'xxx');
```

使用 IN 的查询与批量插入的情况相似，不过 IN 操作并不会导致数据查询结果错误。通过对 IN 查询的改写，可以进一步的提升查询性能。如以下 SQL：

```
SELECT * FROM t_order WHERE order_id IN (1, 2, 3);
```

改写为：

```
SELECT * FROM t_order_0 WHERE order_id IN (2);
SELECT * FROM t_order_1 WHERE order_id IN (1, 3);
```

可以进一步的提升查询性能。ShardingSphere 暂时还未实现此改写策略，目前的改写结果是：

```
SELECT * FROM t_order_0 WHERE order_id IN (1, 2, 3);
SELECT * FROM t_order_1 WHERE order_id IN (1, 2, 3);
```

虽然 SQL 的执行结果是正确的，但并未达到最优的查询效率。

优化改写

优化改写的目的是在不影响查询正确性的情况下，对性能进行提升的有效手段。它分为单节点优化和流式归并优化。

单节点优化

路由至单节点的 SQL，则无需优化改写。当获得一次查询的路由结果后，如果是路由至唯一的数据节点，则无需涉及到结果归并。因此补列和分页信息等改写都没有必要进行。尤其是分页信息的改写，无需将数据从第 1 条开始取，大量的降低了对数据库的压力，并且节省了网络带宽的无谓消耗。

流式归并优化

它仅为包含 `GROUP BY` 的 SQL 增加 `ORDER BY` 以及和分组项相同的排序项和排序顺序，用于将内存归并转化为流式归并。在结果归并的部分中，将对流式归并和内存归并进行详细说明。

改写引擎的整体结构划分如下图所示。

9.2.10 执行引擎

ShardingSphere 采用一套自动化的执行引擎，负责将路由和改写完成之后的真实 SQL 安全且高效发送到底层数据源执行。它不是简单地将 SQL 通过 JDBC 直接发送至数据源执行；也并非直接将执行请求放入线程池去并发执行。它更关注平衡数据源连接创建以及内存占用所产生的消耗，以及最大限度地合理利用并发等问题。执行引擎的目标是自动化的平衡资源控制与执行效率。

连接模式

从资源控制的角度看，业务方访问数据库的连接数量应当有所限制。它能够有效地防止某一业务操作过多的占用资源，从而将数据库连接的资源耗尽，以致于影响其他业务的正常访问。特别是在一个数据库实例中存在较多分表的情况下，一条不包含分片键的逻辑 SQL 将产生落在同库不同表的大量真实 SQL，如果每条真实 SQL 都占用一个独立的连接，那么一次查询无疑将会占用过多的资源。

从执行效率的角度看，为每个分片查询维持一个独立的数据库连接，可以更加有效的利用多线程来提升执行效率。为每个数据库连接开启独立的线程，可以将 I/O 所产生的消耗并行处理。为每个分片维持一个

改写引擎
正确性改写
优化改写
标识符改写
补列
分页修正
批量拆分
表名称改写
索引名称改写
Schema改写
排序补列
分组补列
聚合补列
自增主键补列
批量插入拆分
IN查询拆分
单节点优化
流式归并优化

图 6: 改写引擎结构

独立的数据库连接，还能够避免过早的将查询结果数据加载至内存。独立的数据库连接，能够持有查询结果集游标位置的引用，在需要获取相应数据时移动游标即可。

以结果集游标下移进行结果归并的方式，称之为流式归并，它无需将结果数据全数加载至内存，可以有效的节省内存资源，进而减少垃圾回收的频次。当无法保证每个分片查询持有一个独立数据库连接时，则需要在复用该数据库连接获取下一张分表的查询结果集之前，将当前的查询结果集全数加载至内存。因此，即使可以采用流式归并，在此场景下也将退化为内存归并。

一方面是对数据库连接资源的控制保护，一方面是采用更优的归并模式达到对中间件内存资源的节省，如何处理好两者之间的关系，是 ShardingSphere 执行引擎需要解决的问题。具体来说，如果一条 SQL 在经过 ShardingSphere 的分片后，需要操作某数据库实例下的 200 张表。那么，是选择创建 200 个连接并行执行，还是选择创建一个连接串行执行呢？效率与资源控制又应该如何抉择呢？

针对上述场景，ShardingSphere 提供了一种解决思路。它提出了连接模式（Connection Mode）的概念，将其划分为内存限制模式（MEMORY_STRICTLY）和连接限制模式（CONNECTION_STRICTLY）这两种类型。

内存限制模式

使用此模式的前提是，ShardingSphere 对一次操作所耗费的数据库连接数量不做限制。如果实际执行的 SQL 需要对某数据库实例中的 200 张表做操作，则对每张表创建一个新的数据库连接，并通过多线程的方式并发处理，以达成执行效率最大化。并且在 SQL 满足条件情况下，优先选择流式归并，以防止出现内存溢出或避免频繁垃圾回收情况。

连接限制模式

使用此模式的前提是，ShardingSphere 严格控制对一次操作所耗费的数据库连接数量。如果实际执行的 SQL 需要对某数据库实例中的 200 张表做操作，那么只会创建唯一的数据库连接，并对其 200 张表串行处理。如果一次操作中的分片散落在不同的数据库，仍然采用多线程处理对不同库的操作，但每个库的每次操作仍然只创建一个唯一的数据库连接。这样即可以防止对一次请求对数据库连接占用过多所带来的问题。该模式始终选择内存归并。

内存限制模式适用于 OLAP 操作，可以通过放宽对数据库连接的限制提升系统吞吐量；连接限制模式适用于 OLTP 操作，OLTP 通常带有分片键，会路由到单一的分片，因此严格控制数据库连接，以保证在线系统数据库资源能够被更多的应用所使用，是明智的选择。

自动化执行引擎

ShardingSphere 最初将使用何种模式的决定权交由用户配置，让开发者依据自己业务的实际场景需求选择使用内存限制模式或连接限制模式。

这种解决方案将两难的选择的决定权交由用户，使得用户必须要了解这两种模式的利弊，并依据业务场景需求进行选择。这无疑增加了用户对 ShardingSphere 的学习和使用的成本，并非最优方案。

这种一分为二的处理方案，将两种模式的切换交由静态的初始化配置，是缺乏灵活应对能力的。在实际的使用场景中，面对不同 SQL 以及占位符参数，每次的路由结果是不同的。这就意味着某些操作可能需要使用内存归并，而某些操作则可能选择流式归并更优，具体采用哪种方式不应该由用户在 ShardingSphere 启动之前配置好，而是应该根据 SQL 和占位符参数的场景，来动态的决定连接模式。

为了降低用户的使用成本以及连接模式动态化这两个问题，ShardingSphere 提炼出自动化执行引擎的思路，在其内部消化了连接模式概念。用户无需了解所谓的内存限制模式和连接限制模式是什么，而是交由执行引擎根据当前场景自动选择最优的执行方案。

自动化执行引擎将连接模式的选择粒度细化至每一次 SQL 的操作。针对每次 SQL 请求，自动化执行引擎都将根据其路由结果，进行实时的演算和权衡，并自主地采用恰当的连接模式执行，以达到资源控制和效率的最优平衡。针对自动化的执行引擎，用户只需配置 maxConnectionSizePerQuery 即可，该参数表示一次查询时每个数据库所允许使用的最大连接数。

执行引擎分为准备和执行两个阶段。

准备阶段

顾名思义，此阶段用于准备执行的数据。它分为结果集分组和执行单元创建两个步骤。

结果集分组是实现内化连接模式概念的关键。执行引擎根据 maxConnectionSizePerQuery 配置项，结合当前路由结果，选择恰当的连接模式。具体步骤如下：

1. 将 SQL 的路由结果按照数据源的名称进行分组。
2. 通过下图的公式，可以获得每个数据库实例在 `maxConnectionSizePerQuery` 的允许范围内，每个连接需要执行的 SQL 路由结果组，并计算出本次请求的最优连接模式。

图 7: 连接模式计算公式

在 maxConnectionSizePerQuery 允许的范围内，当一个连接需要执行的请求数量大于 1 时，意味着当前的数据库连接无法持有相应的数据结果集，则必须采用内存归并；反之，当一个连接需要执行的请求数量等于 1 时，意味着当前的数据库连接可以持有相应的数据结果集，则可以采用流式归并。

每一次的连接模式的选择，是针对每一个物理数据库的。也就是说，在同一次查询中，如果路由至一个以上的数据库，每个数据库的连接模式不一定一样，它们可能是混合存在的形态。

通过上一步骤获得的路由分组结果创建执行的单元。当数据源使用数据库连接池等控制数据库连接数量的技术时，在获取数据库连接时，如果不妥善处理并发，则有一定几率发生死锁。在多个请求相互等待对方释放数据库连接资源时，将会产生饥饿等待，造成交叉的死锁问题。

举例说明，假设一次查询需要在某一数据源上获取两个数据库连接，并路由至同一个数据库的两个分表查询。则有可能出现查询 A 已获取到该数据源的 1 个数据库连接，并等待获取另一个数据库连接；而查

询 B 也已经在该数据源上获取到的一个数据库连接，并同样等待另一个数据库连接的获取。如果数据库连接池的允许最大连接数是 2，那么这 2 个查询请求将永久的等待下去。下图描绘了死锁的情况。

图 8: 获取资源死锁

ShardingSphere 为了避免死锁的出现，在获取数据库连接时进行了同步处理。它在创建执行单元时，以原子性的方式一次性获取本次 SQL 请求所需的全部数据库连接，杜绝了每次查询请求获取到部分资源的可能。由于对数据库的操作非常频繁，每次获取数据库连接时时都进行锁定，会降低 ShardingSphere 的并发。因此，ShardingSphere 在这里进行了 2 点优化：

1. 避免锁定一次性只需要获取 1 个数据库连接的操作。因为每次仅需要获取 1 个连接，则不会发生两个请求相互等待的场景，无需锁定。对于大部分 OLTP 的操作，都是使用分片键路由至唯一的数据节点，这会使得系统变为完全无锁的状态，进一步提升了并发效率。除了路由至单分片的情况，读写分离也在此范畴之内。
2. 仅针对内存限制模式时才进行资源锁定。在使用连接限制模式时，所有的查询结果集将在装载至内存之后释放掉数据库连接资源，因此不会产生死锁等待的问题。

执行阶段

该阶段用于真正的执行 SQL，它分为分组执行和归并结果集生成两个步骤。

分组执行将准备执行阶段生成的执行单元分组下发至底层并发执行引擎，并针对执行过程中的每个关键步骤发送事件。如：执行开始事件、执行成功事件以及执行失败事件。执行引擎仅关注事件的发送，它并不关心事件的订阅者。ShardingSphere 的其他模块，如：分布式事务、调用链路追踪等，会订阅感兴趣的事件，并进行相应的处理。

ShardingSphere 通过在执行准备阶段的获取的连接模式，生成内存归并结果集或流式归并结果集，并将其传递至结果归并引擎，以进行下一步的工作。

执行引擎的整体结构划分如下图所示。

图 9: 执行引擎流程图

9.2.11 归并引擎

将从各个数据节点获取的多数据结果集，组合成为一个结果集并正确的返回至请求客户端，称为结果归并。

ShardingSphere 支持的结果归并从功能上分为遍历、排序、分组、分页和聚合 5 种类型，它们是组合而非互斥的关系。从结构划分，可分为流式归并、内存归并和装饰者归并。流式归并和内存归并是互斥的，装饰者归并可以在流式归并和内存归并之上做进一步的处理。

由于从数据库中返回的结果集是逐条返回的，并不需要将所有的数据一次性加载至内存中，因此，在进行结果归并时，沿用数据库返回结果集的方式进行归并，能够极大减少内存的消耗，是归并方式的优先选择。

流式归并是指每一次从结果集中获取到的数据，都能够通过逐条获取的方式返回正确的单条数据，它与数据库原生的返回结果集的方式最为契合。遍历、排序以及流式分组都属于流式归并的一种。

内存归并则是需要将结果集的所有数据都遍历并存储在内存中，再通过统一的分组、排序以及聚合等计算之后，再将其封装成为逐条访问的数据结果集返回。

装饰者归并是对所有的结果集归并进行统一的功能增强，目前装饰者归并有分页归并和聚合归并这 2 种类型。

遍历归并

它是最为简单的归并方式。只需将多个数据结果集合并为一个单向链表即可。在遍历完成链表中当前数据结果集之后，将链表元素后移一位，继续遍历下一个数据结果集即可。

排序归并

由于在 SQL 中存在 `ORDER BY` 语句，因此每个数据结果集自身是有序的，因此只需要将数据结果集当前游标指向的数据值进行排序即可。这相当于对多个有序的数组进行排序，归并排序是最适合此场景的排序算法。

ShardingSphere 在对排序的查询进行归并时，将每个结果集的当前数据值进行比较（通过实现 Java 的 Comparable 接口完成），并将其放入优先级队列。每次获取下一条数据时，只需将队列顶端结果集的游标下移，并根据新游标重新进入优先级排序队列找到自己的位置即可。

通过一个例子来说明 ShardingSphere 的排序归并，下图是一个通过分数进行排序的示例图。图中展示了 3 张表返回的数据结果集，每个数据结果集已经根据分数排序完毕，但是 3 个数据结果集之间是无序的。将 3 个数据结果集的当前游标指向的数据值进行排序，并放入优先级队列，t_score_0 的第一个数据值最大，t_score_2 的第一个数据值次之，t_score_1 的第一个数据值最小，因此优先级队列根据 t_score_0，t_score_2 和 t_score_1 的方式排序队列。

下图则展现了进行 next 调用的时候，排序归并是如何进行的。通过图中我们可以看到，当进行第一次 next 调用时，排在队列首位的 t_score_0 将会被弹出队列，并且将当前游标指向的数据值（也就是 100）返回至查询客户端，并且将游标下移一位之后，重新放入优先级队列。而优先级队列也会根据 t_score_0 的当前数据结果集指向游标的数据值（这里是 90）进行排序，根据当前数值，t_score_0 排列在队列的最后一位。之前队列中排名第二的 t_score_2 的数据结果集则自动排在了队列首位。

图 10: 排序归并示例 1

在进行第二次 next 时，只需要将目前排列在队列首位的 t_score_2 弹出队列，并且将其数据结果集游标指向的值返回至客户端，并下移游标，继续加入队列排队，以此类推。当一个结果集中已经没有数据了，则无需再次加入队列。

可以看到，对于每个数据结果集中的数据有序，而多数据结果集整体无序的情况下，ShardingSphere 无需将所有的数据都加载至内存即可排序。它使用的是流式归并的方式，每次 next 仅获取唯一正确的一条数据，极大的节省了内存的消耗。

从另一个角度来说，ShardingSphere 的排序归并，是在维护数据结果集的纵轴和横轴这两个维度的有序性。纵轴是指每个数据结果集本身，它是天然有序的，它通过包含 `ORDER BY` 的 SQL 所获取。横轴是指每个数据结果集当前游标所指向的值，它需要通过优先级队列来维护其正确顺序。每一次数据结果集当前游标的下移，都需要将该数据结果集重新放入优先级队列排序，而只有排列在队列首位的数据结果集才可能发生游标下移的操作。

分组归并

分组归并的情况最为复杂，它分为流式分组归并和内存分组归并。流式分组归并要求 SQL 的排序项与分组项的字段以及排序类型（ASC 或 DESC）必须保持一致，否则只能通过内存归并才能保证其数据的正确性。

举例说明，假设根据科目分片，表结构中包含考生的姓名（为了简单起见，不考虑重名的情况）和分数。通过 SQL 获取每位考生的总分，可通过如下 SQL：

```
SELECT name, SUM(score) FROM t_score GROUP BY name ORDER BY name;
```

在分组项与排序项完全一致的情况下，取得的数据是连续的，分组所需的数据全数存在于各个数据结果集的当前游标所指向的数据值，因此可以采用流式归并。如下图所示。

图 11: 排序归并示例 2

图 12: 分组归并示例 1

进行归并时，逻辑与排序归并类似。下图展现了进行 next 调用的时候，流式分组归并是如何进行的。

图 13: 分组归并示例 2

通过图中我们可以看到，当进行第一次 next 调用时，排在队列首位的 t_score_java 将会被弹出队列，并且将分组值同为“Jerry”的其他结果集中的数据一同弹出队列。在获取了所有的姓名为“Jerry”的同学的分数之后，进行累加操作，那么，在第一次 next 调用结束后，取出的结果集是“Jerry”的分数总和。与此同时，所有的数据结果集中的游标都将下移至数据值“Jerry”的下一个不同的数据值，并且根据数据结果集当前游标指向的值进行重排序。因此，包含名字顺着第二位的“John”的相关数据结果集则排在的队列的前列。

流式分组归并与排序归并的区别仅仅在于两点：

1. 它会一次性的将多个数据结果集中的分组项相同的数据全数取出。
2. 它需要根据聚合函数的类型进行聚合计算。

对于分组项与排序项不一致的情况，由于需要获取分组的相关的数据值并非连续的，因此无法使用流式归并，需要将所有的结果集数据加载至内存中进行分组和聚合。例如，若通过以下 SQL 获取每位考生的总分并按照分数从高至低排序：

```
SELECT name, SUM(score) FROM t_score GROUP BY name ORDER BY score DESC;
```

那么各个数据结果集中取出的数据与排序归并那张图的上半部分的表结构的原始数据一致，是无法进行流式归并的。

当 SQL 中只包含分组语句时，根据不同数据库的实现，其排序的顺序不一定与分组顺序一致。但由于排序语句的缺失，则表示此 SQL 并不在意排序顺序。因此，ShardingSphere 通过 SQL 优化的改写，自动增加与分组项一致的排序项，使其能够从消耗内存的内存分组归并方式转化为流式分组归并方案。

聚合归并

无论是流式分组归并还是内存分组归并，对聚合函数的处理都是一致的。除了分组的 SQL 之外，不进行分组的 SQL 也可以使用聚合函数。因此，聚合归并是在之前介绍的归并类的之上追加的归并能力，即装饰者模式。聚合函数可以归类为比较、累加和求平均值这 3 种类型。

比较类型的聚合函数是指 `MAX` 和 `MIN`。它们需要对每一个同组的结果集数据进行比较，并且直接返回其最大或最小值即可。

累加类型的聚合函数是指 `SUM` 和 `COUNT`。它们需要将每一个同组的结果集数据进行累加。

求平均值的聚合函数只有 `AVG`。它必须通过 SQL 改写的 `SUM` 和 `COUNT` 进行计算，相关内容已在 SQL 改写的内容中涵盖，不再赘述。

分页归并

上文所述的所有归并类型都可能进行分页。分页也是追加在其他归并类型之上的装饰器，ShardingSphere 通过装饰者模式来增加对数据结果集进行分页的能力。分页归并负责将无需获取的数据过滤掉。

ShardingSphere 的分页功能比较容易让使用者误解，用户通常认为分页归并会占用大量内存。在分布式的场景中，将 `LIMIT 10000000, 10` 改写为 `LIMIT 0, 10000010`，才能保证其数据的正确性。用户非常容易产生 ShardingSphere 会将大量无意义的数据加载至内存中，造成内存溢出风险的错觉。其实，通过流式归并的原理可知，会将数据全部加载到内存中的只有内存分组归并这一种情况。而通常来说，进行 OLAP 的分组 SQL，不会产生大量的结果数据，它更多的用于大量的计算，以及少量结果产出的场景。除了内存分组归并这种情况之外，其他情况都通过流式归并获取数据结果集，因此 ShardingSphere 会通过结果集的 next 方法将无需取出的数据全部跳过，并不会将其存入内存。

但同时需要注意的是，由于排序的需要，大量的数据仍然需要传输到 ShardingSphere 的内存空间。因此，采用 LIMIT 这种方式分页，并非最佳实践。由于 LIMIT 并不能通过索引查询数据，因此如果可以保证 ID 的连续性，通过 ID 进行分页是比较好的解决方案，例如：

```
SELECT * FROM t_order WHERE id > 100000 AND id <= 100010 ORDER BY id;
```

或通过记录上次查询结果的最后一条记录的 ID 进行下一页的查询，例如：

```
SELECT * FROM t_order WHERE id > 10000000 LIMIT 10;
```

归并引擎的整体结构划分如下图。

9.3 分布式事务

9.3.1 导览

本小节主要介绍 Apache ShardingSphere 分布式事务的实现原理

- 基于 XA 协议的两阶段事务
- 基于 Seata 的柔性事务

图 14: 归并引擎结构

9.3.2 XA 事务

`XAShardingSphereTransactionManager` 为 Apache ShardingSphere 的分布式事务的 XA 实现类。它主要负责对多数据源进行管理和适配，并且将相应事务的开启、提交和回滚操作委托给具体的 XA 事务管理器。

开启全局事务

收到接入端的 `set autoCommit=0` 时，`XAShardingSphereTransactionManager` 将调用具体的 XA 事务管理器开启 XA 全局事务，以 XID 的形式进行标记。

执行真实分片 SQL

`XAShardingSphereTransactionManager` 将数据库连接所对应的 XAResource 注册到当前 XA 事务中之后，事务管理器会在此阶段发送 `XAResource.start` 命令至数据库。数据库在收到 `XAResource.end` 命令之前的所有 SQL 操作，会被标记为 XA 事务。

例如:

```
XAResource1.start              ## Enlist 阶段执行
statement.execute("sql1");     ## 模拟执行一个分片 SQL1
statement.execute("sql2");     ## 模拟执行一个分片 SQL2
XAResource1.end                ## 提交阶段执行
```

示例中的 `sql1` 和 `sql2` 将会被标记为 XA 事务。

图 15: XA 事务实现原理

提交或回滚事务

`XAShardingSphereTransactionManager` 在接收到接入端的提交命令后，会委托实际的 XA 事务管理进行提交动作，事务管理器将收集到的当前线程中所有注册的 XAResource，并发送 `XAResource.end` 指令，用以标记此 XA 事务边界。接着会依次发送 `prepare` 指令，收集所有参与 XAResource 投票。若所有 XAResource 的反馈结果均为正确，则调用 `commit` 指令进行最终提交；若有任意 XAResource 的反馈结果不正确，则调用 `rollback` 指令进行回滚。在事务管理器发出提交指令后，任何 XAResource 产生的异常都会通过恢复日志进行重试，以保证提交阶段的操作原子性，和数据强一致性。

例如:

```
XAResource1.prepare           ## ack: yes
XAResource2.prepare           ## ack: yes
XAResource1.commit
XAResource2.commit

XAResource1.prepare           ## ack: yes
XAResource2.prepare           ## ack: no
XAResource1.rollback
XAResource2.rollback
```

9.3.3 Seata 柔性事务

整合 Seata AT 事务时，需要将 TM，RM 和 TC 的模型融入 Apache ShardingSphere 的分布式事务生态中。在数据库资源上，Seata 通过对接 `DataSource` 接口，让 JDBC 操作可以同 TC 进行远程通信。同样，Apache ShardingSphere 也是面向 `DataSource` 接口，对用户配置的数据源进行聚合。因此，将 `DataSource` 封装为基于 Seata 的 `DataSource` 后，就可以将 Seata AT 事务融入到 Apache ShardingSphere 的分片生态中。

图 16: 柔性事务 Seata

引擎初始化

包含 Seata 柔性事务的应用启动时，用户配置的数据源会根据 `seata.conf` 的配置，适配为 Seata 事务所需的 `DataSourceProxy`，并且注册至 RM 中。

开启全局事务

TM 控制全局事务的边界，TM 通过向 TC 发送 Begin 指令，获取全局事务 ID，所有分支事务通过此全局事务 ID，参与到全局事务中；全局事务 ID 的上下文存放在当前线程变量中。

执行真实分片 SQL

处于 Seata 全局事务中的分片 SQL 通过 RM 生成 undo 快照，并且发送 `participate` 指令至 TC，加入到全局事务中。由于 Apache ShardingSphere 的分片物理 SQL 采取多线程方式执行，因此整合 Seata AT 事务时，需要在主线程和子线程间进行全局事务 ID 的上下文传递。

提交或回滚事务

提交 Seata 事务时，TM 会向 TC 发送全局事务的提交或回滚指令，TC 根据全局事务 ID 协调所有分支事务进行提交或回滚。

9.4 弹性伸缩

9.4.1 原理说明

考虑到 Apache ShardingSphere 的弹性伸缩模块的几个挑战，目前的弹性伸缩解决方案为：临时地使用两个数据库集群，伸缩完成后切换的方式实现。

这种实现方式有以下优点：

1. 伸缩过程中，原始数据没有任何影响；
2. 伸缩失败无风险；
3. 不受分片策略限制。

同时也存在一定的缺点：

1. 在一定时间内存在冗余服务器；
2. 所有数据都需要移动。

弹性伸缩模块会通过解析旧分片规则，提取配置中的数据源、数据节点等信息，之后创建伸缩作业工作流，将一次弹性伸缩拆解为 4 个主要阶段：

1. 准备阶段；
2. 存量数据迁移阶段；
3. 增量数据同步阶段；
4. 规则切换阶段。

图 17: 伸缩总揽

图 18: 伸缩工作流

9.4.2 执行阶段说明

准备阶段

在准备阶段，弹性伸缩模块会进行数据源连通性及权限的校验，同时进行存量数据的统计、日志位点的记录，最后根据数据量和用户设置的并行度，对任务进行分片。

存量数据迁移阶段

执行在准备阶段拆分好的存量数据迁移作业，存量迁移阶段采用 JDBC 查询的方式，直接从数据节点中读取数据，并使用新规则写入到新集群中。

增量数据同步阶段

由于存量数据迁移耗费的时间受到数据量和并行度等因素影响，此时需要对这段时间内业务新增的数据进行同步。不同的数据库使用的技术细节不同，但总体上均为基于复制协议或 WAL 日志实现的变更数据捕获功能。

- MySQL：订阅并解析 binlog；
- PostgreSQL：采用官方逻辑复制 test_decoding。

这些捕获的增量数据，同样会由弹性伸缩模块根据新规则写入到新数据节点中。当增量数据基本同步完成时（由于业务系统未停止，增量数据是不断的），则进入规则切换阶段。

规则切换阶段

在此阶段，可能存在一定时间的业务只读窗口期，通过设置数据库只读或 ShardingSphere 的熔断机制，让旧数据节点中的数据短暂静态，确保增量同步已完全完成。

这个窗口期时间短则数秒，长则数分钟，取决于数据量和用户是否需要对数据进行强校验。确认完成后，Apache ShardingSphere 可通过配置中心修改配置，将业务导向新规则的集群，弹性伸缩完成。

9.5 数据加密

9.5.1 处理流程详解

Apache ShardingSphere 通过对用户输入的 SQL 进行解析，并依据用户提供的加密规则对 SQL 进行改写，从而实现对原文数据进行加密，并将原文数据（可选）及密文数据同时存储到底层数据库。在用户查询数据时，它仅从数据库中取出密文数据，并对其解密，最终将解密后的原始数据返回给用户。Apache ShardingSphere 自动化 & 透明化了数据加密过程，让用户无需关注数据加密的实现细节，像使用普通数据那样使用加密数据。此外，无论是已在线业务进行加密改造，还是新上线业务使用加密功能，Apache ShardingSphere 都可以提供一套相对完善的解决方案。

整体架构

图 19: 1

加密模块将用户发起的 SQL 进行拦截，并通过 SQL 语法解析器进行解析、理解 SQL 行为，再依据用户传入的加密规则，找出需要加密的字段和所使用的加解密算法对目标字段进行加解密处理后，再与底层数据库进行交互。Apache ShardingSphere 会将用户请求的明文进行加密后存储到底层数据库；并在用户查询时，将密文从数据库中取出进行解密后返回给终端用户。通过屏蔽对数据的加密处理，使用户无需感知解析 SQL、数据加密、数据解密的处理过程，就像在使用普通数据一样使用加密数据。

加密规则

在详解整套流程之前，我们需要先了解下加密规则与配置，这是认识整套流程的基础。加密配置主要分为四部分：数据源配置，加密算法配置，加密表配置以及查询属性配置，其详情如下图所示：

数据源配置：指数据源配置。

加密算法配置：指使用什么加密算法进行加解密。目前 ShardingSphere 内置了三种加解密算法：AES，MD5 和 RC4。用户还可以通过实现 ShardingSphere 提供的接口，自行实现一套加解密算法。

加密表配置：用于告诉 ShardingSphere 数据表里哪个列用于存储密文数据（cipherColumn）、哪个列用于存储明文数据（plainColumn）以及用户想使用哪个列进行 SQL 编写（logicColumn）。

> 如何理解 用户想使用哪个列进行 `SQL` 编写（`logicColumn`）?
>
> 我们可以从加密模块存在的意义来理解。加密模块最终目的是希望屏蔽底层对数据的加密处理，也就是说我们不希望用户知道数据是如何被加解密的、如何将明文数据存储到 plainColumn，将密文数据存储到 cipherColumn。换句话说，我们不希望用户知道 plainColumn 和 cipherColumn 的存在和使用。所以，我们需要给用户提供一个概念意义上的列，这个列可以脱离底层数据库的真实列，它可以是数据库表里的一个真实列，也可以不是，从而使得用户

图 20: 2

可以随意改变底层数据库的 plainColumn 和 cipherColumn 的列名。或者删除 plainColumn，选择永远不再存储明文，只存储密文。只要用户的 SQL 面向这个逻辑列进行编写，并在加密规则里给出 logicColumn 和 plainColumn、cipherColumn 之间正确的映射关系即可。

为什么要这么做呢？答案在文章后面，即为了让已上线的业务能无缝、透明、安全地进行数据加密迁移。

查询属性的配置：当底层数据库表里同时存储了明文数据、密文数据后，该属性开关用于决定是直接查询数据库表里的明文数据进行返回，还是查询密文数据通过 Apache ShardingSphere 解密后返回。该属性开关支持表级别和整个规则级别配置，表级别优先级最高。

加密处理过程

举例说明，假如数据库里有一张表叫做 `t_user`，这张表里实际有两个字段 `pwd_plain`，用于存放明文数据、`pwd_cipher`，用于存放密文数据，同时定义 logicColumn 为 pwd。那么，用户在编写 SQL 时应该面向 logicColumn 进行编写，即 `INSERT INTO t_user SET pwd = '123'`。Apache ShardingSphere 接收到该 SQL，通过用户提供的加密配置，发现 pwd 是 logicColumn，于是便对逻辑列及其对应的明文数据进行加密处理。**Apache ShardingSphere 将面向用户的逻辑列与面向底层数据库的明文列和密文列进行了列名以及数据的加密映射转换**。如下图所示：

即依据用户提供的加密规则，将用户 SQL 与底层数据表结构割裂开来，使得用户的 SQL 编写不再依赖于真实的数据库表结构。而用户与底层数据库之间的衔接、映射、转换交由 Apache ShardingSphere 进行处理。

模型
举例
应用程序
交互
logicColumn
ShardingSphere
映射
plainColumn
cipherColumn
真实数据表
应用程序
交互
pwd
t_user
映射
pwd_plain
pwd_cipher
t_user

图 21: 3

下方图片展示了使用加密模块进行增删改查时，其中的处理流程和转换逻辑，如下图所示。

图 22: 4

9.5.2 解决方案详解

在了解了 Apache ShardingSphere 加密处理流程后，即可将加密配置、加密处理流程与实际场景进行结合。所有的设计开发都是为了解决业务场景遇到的痛点。那么面对之前提到的业务场景需求，又应该如何使用 Apache ShardingSphere 这把利器来满足业务需求呢?

新上线业务

业务场景分析：新上线业务由于一切从零开始，不存在历史数据清洗问题，所以相对简单。

解决方案说明：选择合适的加密算法，如 AES 后，只需配置逻辑列（面向用户编写 SQL ）和密文列（数据表存密文数据）即可，**逻辑列和密文列可以相同也可以不同**。建议配置如下（YAML 格式展示）:

```
-!ENCRYPT
  encryptors:
    aes_encryptor:
      type: AES
      props:
        aes-key-value: 123456abc
  tables:
    t_user:
```

```
    columns:
      pwd:
        cipherColumn: pwd
        encryptorName: aes_encryptor
```

使用这套配置，Apache ShardingSphere 只需将 logicColumn 和 cipherColumn 进行转换，底层数据表不存储明文，只存储了密文，这也是安全审计部分的要求所在。如果用户希望将明文、密文一同存储到数据库，只需添加 plainColumn 配置即可。整体处理流程如下图所示：

图 23: 5

已上线业务改造

业务场景分析：由于业务已经在线上运行，数据库里必然存有大量明文历史数据。现在的问题是如何让历史数据得以加密清洗、如何让增量数据得以加密处理、如何让业务在新旧两套数据系统之间进行无缝、透明化迁移。

解决方案说明：在提供解决方案之前，我们先来头脑风暴一下：首先，既然是旧业务需要进行加密改造，那一定存储了非常重要且敏感的信息。这些信息含金量高且业务相对基础重要。不应该采用停止业务禁止新数据写入，再找个加密算法把历史数据全部加密清洗，再把之前重构的代码部署上线，使其能把存量和增量数据进行在线加密解密。

那么另一种相对安全的做法是：重新搭建一套和生产环境一模一样的预发环境，然后通过相关迁移洗数工具把生产环境的**存量原文数据**加密后存储到预发环境，而**新增数据**则通过例如 MySQL 主从复制及业务方自行开发的工具加密后存储到预发环境的数据库里，再把重构后可以进行加解密的代码部署到预发环境。这样生产环境是一套**以明文为核心的查询修改**的环境；预发环境是一套**以密文为核心加解密查询修改**的环境。在对比一段时间无误后，可以夜间操作将生产流量切到预发环境中。此方案相对安全可靠，只是时间、人力、资金、成本较高，主要包括：预发环境搭建、生产代码整改、相关辅助工具开发等。

业务开发人员最希望的做法是：减少资金费用的承担、最好不要修改业务代码、能够安全平滑迁移系统。于是，ShardingSphere 的加密功能模块便应运而生。可分为 3 步进行：

1. 系统迁移前

假设系统需要对 `t_user` 的 pwd 字段进行加密处理，业务方使用 Apache ShardingSphere 来代替标准化的 JDBC 接口，此举基本不需要额外改造（我们还提供了 Spring Boot Starter，Spring 命名空间，YAML 等接入方式，满足不同业务方需求）。另外，提供一套加密配置规则，如下所示：

```
-!ENCRYPT
  encryptors:
    aes_encryptor:
      type: AES
      props:
        aes-key-value: 123456abc
  tables:
    t_user:
      columns:
        pwd:
          plainColumn: pwd
          cipherColumn: pwd_cipher
          encryptorName: aes_encryptor
  queryWithCipherColumn: false
```

依据上述加密规则可知，首先需要在数据库表 `t_user` 里新增一个字段叫做 `pwd_cipher`，即 cipherColumn，用于存放密文数据，同时我们把 plainColumn 设置为 pwd，用于存放明文数据，而把 logicColumn 也设置为 pwd。由于之前的代码 SQL 就是使用 pwd 进行编写，即面向逻辑列进行 SQL 编写，所以业务代码无需改动。通过 Apache ShardingSphere，针对新增的数据，会把明文写到 pwd 列，并同时把明文进行加密存储到 `pwd_cipher` 列。此时，由于 `queryWithCipherColumn` 设置为 false，对业务应用来说，依旧使用 pwd 这一明文列进行查询存储，却在底层数据库表 `pwd_cipher` 上额外存储了新增数据的密文数据，其处理流程如下图所示：

图 24: 6

新增数据在插入时，就通过 Apache ShardingSphere 加密为密文数据，并被存储到了 cipherColumn。而现在就需要处理历史明文存量数据。**由于 Apache ShardingSphere 目前并未提供相关迁移洗数工具，此时需要业务方自行将 “pwd“ 中的明文数据进行加密处理存储到 “pwd_cipher“**。

2. 系统迁移中

新增的数据已被 Apache ShardingSphere 将密文存储到密文列，明文存储到明文列；历史数据被业务方自行加密清洗后，将密文也存储到密文列。也就是说现在的数据库里即存放着明文也存放着密文，只是

由于配置项中的 `queryWithCipherColumn = false`，所以密文一直没有被使用过。现在我们为了让系统能切到密文数据进行查询，需要将加密配置中的 queryWithCipherColumn 设置为 true。在重启系统后，我们发现系统业务一切正常，但是 Apache ShardingSphere 已经开始从数据库里取出密文列的数据，解密后返回给用户；而对于用户的增删改需求，则依旧会把原文数据存储到明文列，加密后密文数据存储到密文列。

虽然现在业务系统通过将密文列的数据取出，解密后返回；但是，在存储的时候仍旧会存一份原文数据到明文列，这是为什么呢？答案是：为了能够进行系统回滚。**因为只要密文和明文永远同时存在，我们就可以通过开关项配置自由将业务查询切换到 cipherColumn 或 plainColumn**。也就是说，如果将系统切到密文列进行查询时，发现系统报错，需要回滚。那么只需将 `queryWithCipherColumn = false`，Apache ShardingSphere 将会还原，即又重新开始使用 plainColumn 进行查询。处理流程如下图所示：

图 25: 7

3. 系统迁移后

由于安全审计部门要求，业务系统一般不可能让数据库的明文列和密文列永久同步保留，我们需要在系统稳定后将明文列数据删除。即我们需要在系统迁移后将 plainColumn，即 pwd 进行删除。那问题来了，现在业务代码都是面向 pwd 进行编写 SQL 的，把底层数据表中的存放明文的 pwd 删除了，换用 pwd_cipher 进行解密得到原文数据，那岂不是意味着业务方需要整改所有 SQL，从而不使用即将要被删除的 pwd 列？还记得我们 Apache ShardingSphere 的核心意义所在吗？

> 这也正是 Apache ShardingSphere 核心意义所在，即依据用户提供的加密规则，将用户 SQL 与底层数据库表结构割裂开来，使得用户的 SQL 编写不再依赖于真实的数据库表结构。而用户与底层数据库之间的衔接、映射、转换交由 Apache ShardingSphere 进行处理。

是的，因为有 logicColumn 存在，用户的编写 SQL 都面向这个虚拟列，Apache ShardingSphere 就可以把这个逻辑列和底层数据表中的密文列进行映射转换。于是迁移后的加密配置即为：

```
-!ENCRYPT
  encryptors:
    aes_encryptor:
      type: AES
      props:
```

```
      aes-key-value: 123456abc
tables:
  t_user:
    columns:
      pwd: # pwd 与 pwd_cipher 的转换映射
        cipherColumn: pwd_cipher
        encryptorName: aes_encryptor
```

其处理流程如下：

图 26: 8

至此，已在线业务加密整改解决方案全部叙述完毕。我们提供了 Java、YAML、Spring Boot Starter、Spring 命名空间多种方式供用户选择接入，力求满足业务不同的接入需求。该解决方案目前已在京东数科不断落地上线，提供对内基础服务支撑。

9.5.3 中间件加密服务优势

1. 自动化 & 透明化数据加密过程，用户无需关注加密中间实现细节。
2. 提供多种内置、第三方（AKS）的加密算法，用户仅需简单配置即可使用。
3. 提供加密算法 API 接口，用户可实现接口，从而使用自定义加密算法进行数据加密。
4. 支持切换不同的加密算法。
5. 针对已上线业务，可实现明文数据与密文数据同步存储，并通过配置决定使用明文列还是密文列进行查询。可实现在不改变业务查询 SQL 前提下，已上线系统对加密前后数据进行安全、透明化迁移。

9.5.4 加密算法解析

Apache ShardingSphere 提供了两种加密算法用于数据加密，这两种策略分别对应 Apache ShardingSphere 的两种加解密的接口，即 `EncryptAlgorithm` 和 `QueryAssistedEncryptAlgorithm`。

一方面，Apache ShardingSphere 为用户提供了内置的加解密实现类，用户只需进行配置即可使用；另一方面，为了满足用户不同场景的需求，我们还开放了相关加解密接口，用户可依据这两种类型的接口提供具体实现类。再进行简单配置，即可让 Apache ShardingSphere 调用用户自定义的加解密方案进行数据加密。

EncryptAlgorithm

该解决方案通过提供 `encrypt()`，`decrypt()` 两种方法对需要加密的数据进行加解密。在用户进行 `INSERT`，`DELETE`，`UPDATE` 时，ShardingSphere 会按照用户配置，对 SQL 进行解析、改写、路由，并调用 `encrypt()` 将数据加密后存储到数据库，而在 `SELECT` 时，则调用 `decrypt()` 方法将从数据库中取出的加密数据进行逆向解密，最终将原始数据返回给用户。

当前，Apache ShardingSphere 针对这种类型的加密解决方案提供了三种具体实现类，分别是 MD5（不可逆），AES（可逆），RC4（可逆），用户只需配置即可使用这三种内置的方案。

QueryAssistedEncryptAlgorithm

相比较于第一种加密方案，该方案更为安全和复杂。它的理念是：即使是相同的数据，如两个用户的密码相同，它们在数据库里存储的加密数据也应当是不一样的。这种理念更有利于保护用户信息，防止撞库成功。

它提供三种函数进行实现，分别是 `encrypt()`，`decrypt()`，`queryAssistedEncrypt()`。在 `encrypt()` 阶段，用户通过设置某个变动种子，例如时间戳。针对原始数据 + 变动种子组合的内容进行加密，就能保证即使原始数据相同，也因为有变动种子的存在，致使加密后的加密数据是不一样的。在 `decrypt()` 可依据之前规定的加密算法，利用种子数据进行解密。

虽然这种方式确实可以增加数据的保密性，但是另一个问题却随之出现：相同的数据在数据库里存储的内容是不一样的，那么当用户按照这个加密列进行等值查询（`SELECT FROM table WHERE encryptedColumnn = ?`）时会发现无法将所有相同的原始数据查询出来。为此，我们提出了辅助查询列的概念。该辅助查询列通过 `queryAssistedEncrypt()` 生成，与 `decrypt()` 不同的是，该方法通过对原始数据进行另一种方式的加密，但是针对原始数据相同的数据，这种加密方式产生的加密数据是一致的。将 `queryAssistedEncrypt()` 后的数据存储到数据中用于辅助查询真实数据。因此，数据库表中多出这一个辅助查询列。

由于 `queryAssistedEncrypt()` 和 `encrypt()` 产生不同加密数据进行存储，而 `decrypt()` 可逆，`queryAssistedEncrypt()` 不可逆。在查询原始数据的时候，我们会自动对 SQL 进行解析、改写、路由，利用辅助查询列进行 `WHERE` 条件的查询，却利用 `decrypt()` 对 `encrypt()` 加密后的数据进行解密，并将原始数据返回给用户。这一切都是对用户透明化的。

当前，Apache ShardingSphere 针对这种类型的加密解决方案并没有提供具体实现类，却将该理念抽象成接口，提供给用户自行实现。ShardingSphere 将调用用户提供的该方案的具体实现类进行数据加密。

9.6 影子库

9.6.1 整体架构

Apache ShardingSphere 通过解析 SQL，对传入的 SQL 进行影子判定，根据配置文件中用户设置的影子规则，路由到生产库或者影子库。

图 27: 执行流程

9.6.2 影子规则

影子规则包含影子数据源映射关系，影子表以及影子算法。

图 28: 规则

影子库映射：生产数据源名称和影子数据源名称映射关系。

影子表：压测相关的影子表。影子表必须存在于指定的影子库中，并且需要指定影子算法。

影子算法：SQL 路由影子算法。

默认影子算法：默认影子算法。选配项，对于没有配置影子算法表的默认匹配算法。

9.6.3 路由过程

以 INSERT 语句为例，在写入数据时，Apache ShardingSphere 会对 SQL 进行解析，再根据配置文件中的规则，构造一条路由链。在当前版本的功能中，影子功能处于路由链中的最后一个执行单元，即，如果有其他需要路由的规则存在，如分片，Apache ShardingSphere 会首先根据分片规则，路由到某一个数据库，再执行影子路由判定流程，判定执行 SQL 满足影子规则的配置，数据路由到与之对应的影子库，生产数据则维持不变。

9.6.4 影子判定流程

影子库功能对执行的 SQL 语句进行影子判定。影子判定支持两种类型算法，用户可根据实际业务需求选择一种或者组合使用。

DML 语句

支持两种算法。影子判定会首先判断执行 SQL 相关表与配置的影子表是否有交集。如果有交集，依次判定交集部分影子表关联的影子算法，有任何一个判定成功。SQL 语句路由到影子库。影子表没有交集或者影子算法判定不成功，SQL 语句路由到生产库。

DDL 语句

仅支持注解影子算法。在压测场景下，DDL 语句一般不需要测试。主要在初始化或者修改影子库中影子表时使用。

影子判定会首先判断执行 SQL 是否包含注解。如果包含注解，影子规则中配置的 HINT 影子算法依次判定。有任何一个判定成功。SQL 语句路由到影子库。执行 SQL 不包含注解或者 HINT 影子算法判定不成功，SQL 语句路由到生产库。

9.6.5 影子算法

影子算法详情，请参见内置影子算法列表

9.6.6 使用案例

场景需求

假设一个电商网站要对下单业务进行压测。压测相关表 `t_order` 为影子表，生产数据执行到 `ds` 生产数据库，压测数据执行到数据库 `ds_shadow` 影子库。

影子库配置

建议配置如下（YAML 格式展示）:

```
data-sources:
  shadow-data-source:
    source-data-source-name: ds
    shadow-data-source-name: ds-shadow
tables:
  t_order:
    data-source-names: shadow-data-source
    shadow-algorithm-names:
      - simple-hint-algorithm
      - user-id-value-match-algorithm
shadow-algorithms:
  simple-hint-algorithm:
    type: SIMPLE_HINT
    props:
      foo: bar
  user-id-value-match-algorithm:
    type: VALUE_MATCH
    props:
      operation: insert
      column: user_id
      value: 0

sql-parser:
  sql-comment-parse-enabled: true
```

注意: 如果使用注解影子算法, 需要开启解析 SQL 注释配置项 `sql-comment-parse-enabled: true`。默认关闭。请参考 SQL 解析配置

影子库环境

- 创建影子库 `ds_shadow`。
- 创建影子表，表结构与生产环境必须一致。假设在影子库创建 `t_order` 表。创建表语句需要添加 SQL 注释 `/*foo:bar,...*/`。即:

```
CREATE TABLE t_order (order_id INT(11) primary key, user_id int(11) not null, ...)
/*foo:bar,...*/
```

执行到影子库。

注意：如果使用 MySQL 客户端进行测试，链接需要使用参数：`-c` 例如:

```
mysql> mysql -u root -h127.0.0.1 -P3306 -proot -c
```

参数说明：保留注释，发送注释到服务端。

执行包含注解 SQL 例如:

```
SELECT * FROM table_name /*shadow:true,foo:bar*/;
```

不使用参数 -c 会被 MySQL 客户端截取注释语句变为:

```
SELECT * FROM table_name;
```

影响测试结果。

影子算法使用

1. 列影子算法使用

假设 `t_order` 表中包含下单用户 ID 的 `user_id` 列。实现的效果，当用户 ID 为 `0` 的用户创建订单产生的数据。即:

```
INSERT INTO t_order (order_id, user_id, ...) VALUES (xxx..., 0, ...)
```

会执行到影子库，其他数据执行到生产库。

无需修改任何 SQL 或者代码，只需要对压力测试的数据进行控制就可以实现在线的压力测试。

算法配置如下（YAML 格式展示）:

```
shadow-algorithms:
  user-id-value-match-algorithm:
    type: VALUE_MATCH
    props:
      operation: insert
      column: user_id
      value: 0
```

注意: 影子表使用列影子算法时，相同类型操作（INSERT, UPDATE, DELETE, SELECT）目前仅支持单个字段。

2. 使用 Hint 影子算法

假设 `t_order` 表中不包含可以对值进行匹配的列。添加注解 `/*foo:bar,...*/` 到执行 SQL 中，即:

```
SELECT * FROM t_order WHERE order_id = xxx /*foo:bar,...*/
```

会执行到影子库，其他数据执行到生产库。

算法配置如下（YAML 格式展示）:

```
shadow-algorithms:
  simple-hint-algorithm:
    type: SIMPLE_HINT
    props:
      foo: bar
```

3. 混合使用影子模式

假设对 t_order 表压测需要覆盖以上两种场景，即，

```
INSERT INTO t_order (order_id, user_id, ...) VALUES (xxx..., 0, ...);

SELECT * FROM t_order WHERE order_id = xxx /*foo:bar,...*/;
```

都会执行到影子库，其他数据执行到生产库。

算法配置如下（YAML 格式展示）：

```
shadow-algorithms:
  user-id-value-match-algorithm:
    type: VALUE_MATCH
    props:
      operation: insert
      column: user_id
      value: 0
  simple-hint-algorithm:
    type: SIMPLE_HINT
    props:
      foo: bar
```

4. 使用默认影子算法

假设对 t_order 表压测使用列影子算法，其他相关其他表都需要使用 Hint 影子算法。即，

```
INSERT INTO t_order (order_id, user_id, ...) VALUES (xxx..., 0, ...);

INSERT INTO t_xxx_1 (order_item_id, order_id, ...) VALUES (xxx..., xxx..., ...) /
*foo:bar,...*/;

SELECT * FROM t_xxx_2 WHERE order_id = xxx /*foo:bar,...*/;

SELECT * FROM t_xxx_3 WHERE order_id = xxx /*foo:bar,...*/;
```

都会执行到影子库，其他数据执行到生产库。

配置如下（YAML 格式展示）：

```
data-sources:
  shadow-data-source:
    source-data-source-name: ds
    shadow-data-source-name: ds-shadow
tables:
  t_order:
    data-source-names: shadow-data-source
    shadow-algorithm-names:
      - simple-hint-algorithm
      - user-id-value-match-algorithm
```

```
default-shadow-algorithm-name: simple-note-algorithm
shadow-algorithms:
  simple-hint-algorithm:
    type: SIMPLE_HINT
    props:
      foo: bar
  user-id-value-match-algorithm:
    type: VALUE_MATCH
    props:
      operation: insert
      column: user_id
      value: 0

sql-parser:
  sql-comment-parse-enabled: true
```

注意默认影子算法仅支持 Hint 影子算法。使用时必须确保配置文件中 `props` 的配置项小于等于 SQL 注释中的配置项，且配置文件的具体配置要和 SQL 注释中写的配置一样，配置文件中配置项越少，匹配条件越宽松

```
simple-note-algorithm:
  type: SIMPLE_HINT
  props:
    foo: bar
    foo1: bar1
```

如当前 `props` 项中配置了 2 条配置，在 SQL 中可以匹配的写法有如下：

```
SELECT * FROM t_xxx_2 WHERE order_id = xxx /*foo:bar, foo1:bar1*/
```

```
SELECT * FROM t_xxx_2 WHERE order_id = xxx /*foo:bar, foo1:bar1, foo2:bar2, ...*/
```

```
simple-note-algorithm:
  type: SIMPLE_HINT
  props:
    foo: bar
```

如当前 props 项中配置了 1 条配置，在 SQL 中可以匹配的写法有如下：

```
SELECT * FROM t_xxx_2 WHERE order_id = xxx /*foo:foo*/
```

```
SELECT * FROM t_xxx_2 WHERE order_id = xxx /*foo:foo, foo1:bar1, ...*/
```

9.7 测试

Apache ShardingSphere 提供了完善的整合测试、模块测试和性能测试。

9.7.1 整合测试

通过真实的 Apache ShardingSphere 和数据库的连接，提供端到端的测试。

整合测试引擎以 XML 方式定义 SQL，分别为各个数据库独立运行测试用例。为了方便上手，测试引擎无需修改任何 **Java** 代码，只需修改相应的配置文件即可运行断言。测试引擎不依赖于任何第三方环境，用于测试的 ShardingSphere-Proxy 计算节点和数据库均由 Docker 镜像提供。

9.7.2 模块测试

将复杂的模块单独提炼成为测试引擎。

模块测试引擎同样以 XML 方式定义 SQL，分别为各个数据库独立运行测试用例，包括 SQL 解析和 SQL 改写模块。

9.7.3 性能测试

提供多样性的性能测试方法，包括 Sysbench、JMH、TPCC 等。

9.7.4 集成测试

设计

集成测试包括 3 个模块：测试用例、测试环境以及测试引擎。

测试用例

用于定义待测试的 SQL 以及测试结果的断言数据。每个用例定义一条 SQL，SQL 可定义多种数据库执行类型。

测试环境

用于搭建运行测试用例的数据库和 ShardingSphere-Proxy 环境。环境又具体分为环境准备方式，数据库类型和场景。

环境准备方式分为 Native 和 Docker，未来还将增加 Embed 类型的支持。

- Native 环境用于测试用例直接运行在开发者提供的测试环境中，适于调试场景；
- Docker 环境由 Maven 运行 Docker-Compose 插件直接搭建，适用于云编译环境和测试 ShardingSphere-Proxy 的场景，如：GitHub Action；

- Embed 环境由测试框架自动搭建嵌入式 MySQL，适用于 ShardingSphere-JDBC 的本地环境测试。

当前默认采用 Native 环境，使用 ShardingSphere-JDBC + H2 数据库运行测试用例。通过 Maven 的 `-P -Pit.env.docker` 参数可以指定 Docker 环境的运行方式。未来将采用 Embed 环境的 ShardingSphere-JDBC + MySQL，替换 Native 执行测试用例的默认环境类型。

数据库类型目前支持 MySQL、PostgreSQL、SQLServer 和 Oracle，并且可以支持使用 ShardingSphere-JDBC 或是使用 ShardingSphere-Proxy 执行测试用例。

场景用于对 ShardingSphere 支持规则进行测试，目前支持数据分片和读写分离的相关场景，未来会不断完善场景的组合。

测试引擎

用于批量读取测试用例，并逐条执行和断言测试结果。

测试引擎通过将用例和环境进行排列组合，以达到用最少的用例测试尽可能多场景的目的。

每条 SQL 会以 `数据库类型 * 接入端类型 * SQL 执行模式 * JDBC 执行模式 * 场景`的组合方式生成测试报告，目前各个维度的支持情况如下：

- 数据库类型：H2、MySQL、PostgreSQL、SQLServer 和 Oracle；
- 接入端类型：ShardingSphere-JDBC 和 ShardingSphere-Proxy；
- SQL 执行模式：Statement 和 PreparedStatement；
- JDBC 执行模式：execute 和 executeQuery（查询）/ executeUpdate（更新）；
- 场景：分库、分表、读写分离和分库分表 + 读写分离。

因此，1 条 SQL 会驱动：`数据库类型（5）* 接入端类型（2）* SQL 执行模式（2）* JDBC 执行模式（2）* 场景（4）= 160` 个测试用例运行，以达到项目对于高质量的追求。

使用指南

模块路径：`shardingsphere-test/shardingsphere-integration-test/shardingsphere-integration-test-suite`

测试用例配置

SQL 用例在 `resources/cases/${SQL-TYPE}/${SQL-TYPE}-integration-test-cases.xml`。

用例文件格式如下：

```
<integration-test-cases>
    <test-case sql="${SQL}">
        <assertion parameters="${value_1}:${type_1}, ${value_2}:${type_2}"
expected-data-file="${dataset_file_1}.xml" />
        <!-- ... more assertions -->
        <assertion parameters="${value_3}:${type_3}, ${value_4}:${type_4}"
expected-data-file="${dataset_file_2}.xml" />
```

```
    </test-case>

    <!-- ... more test cases -->
</integration-test-cases>
```

expected-data-file 的查找规则是：1. 查找同级目录中 dataset\${SCENARIO_NAME}\${DATABASE_TYPE}\${dataset_file}.xml 文件；2. 查找同级目录中 dataset\${SCENARIO_NAME}\${dataset_file}.xml 文件；3. 查找同级目录中 dataset\${dataset_file}.xml 文件；4. 都找不到则报错。

断言文件格式如下：

```
<dataset>
    <metadata>
        <column name="column_1" />
        <!-- ... more columns -->
        <column name="column_n" />
    </metadata>
    <row values="value_01, value_02" />
    <!-- ... more rows -->
    <row values="value_n1, value_n2" />
</dataset>
```

环境配置

${SCENARIO-TYPE} 表示场景名称，在测试引擎运行中用于标识唯一场景。${DATABASE-TYPE} 表示数据库类型。

Native 环境配置

目录：src/test/resources/env/${SCENARIO-TYPE}

- scenario-env.properties: 数据源配置；
- rules.yaml: 规则配置；
- databases.xml: 真实库名称；
- dataset.xml: 初始化数据；
- init-sql\${DATABASE-TYPE}\init.sql: 初始化数据库表结构；
- authority.xml: 待补充。

Docker 环境配置

目录：`src/test/resources/docker/${SCENARIO-TYPE}`

- `docker-compose.yml`: Docker-Compose 配置文件，用于 Docker 环境启动；
- `proxy/conf/config-${SCENARIO-TYPE}.yaml`: 规则配置。

Docker 环境配置为 ShardingSphere-Proxy 提供了远程调试端口，可以在 "docker-compose.yml" 文件的 "shardingsphere-proxy" 中找到第 2 个暴露的端口用于远程调试。

运行测试引擎

配置测试引擎运行环境

通过配置 `src/test/resources/env/engine-env.properties` 控制测试引擎。

所有的属性值都可以通过 Maven 命令行 `-D` 的方式动态注入。

```
# 场景类型，多个值可用逗号分隔。可选值：db，tbl，dbtbl_with_replica_query，replica_query
it.scenarios=db,tbl,dbtbl_with_replica_query,replica_query

# 是否运行附加测试用例
it.run.additional.cases=false

# 配置环境类型，只支持单值。可选值：docker 或空，默认值：空
it.cluster.env.type=${it.env}
# 待测试的接入端类型，多个值可用逗号分隔。可选值：jdbc，proxy，默认值：jdbc
it.cluster.adapters=jdbc

# 场景类型，多个值可用逗号分隔。可选值：H2，MySQL，Oracle，SQLServer，PostgreSQL
it.cluster.databases=H2,MySQL,Oracle,SQLServer,PostgreSQL
```

运行调试模式

- 标准测试引擎运行 `org.apache.shardingsphere.test.integration.engine.${SQL-TYPE}.General${SQL-TYPE}IT` 以启动不同 SQL 类型的测试引擎。
- 批量测试引擎运行 `org.apache.shardingsphere.test.integration.engine.dml.BatchDMLIT`，以启动为 DML 语句提供的测试 `addBatch()` 的批量测试引擎。
- 附加测试引擎运行 `org.apache.shardingsphere.test.integration.engine.${SQL-TYPE}.Additional${SQL-TYPE}IT` 以启动使用更多 JDBC 方法调用的测试引擎。附加测试引擎需要通过设置 `it.run.additional.cases=true` 开启。

运行 Docker 模式

```
./mvnw -B clean install -f shardingsphere-test/shardingsphere-integration-test/pom.
xml -Pit.env.docker -Dit.cluster.adapters=proxy,jdbc -Dit.scenarios=${scenario_
name_1,scenario_name_1,scenario_name_n} -Dit.cluster.databases=MySQL
```

注意事项

1. 如需测试 Oracle，请在 pom.xml 中增加 Oracle 驱动依赖；
2. 为了保证测试数据的完整性和易读性，整合测试中的分库分表采用了 10 库 10 表的方式，完全运行测试用例所需时间较长。

9.7.5 性能测试

提供各个压测工具的性能测试结果。

Sysbench 性能测试

目标

对 ShardingSphere-JDBC，ShardingSphere-Proxy 及 MySQL 进行性能对比。从业务角度考虑，在基本应用场景（单路由，主从 + 加密 + 分库分表，全路由）下，INSERT+UPDATE+DELETE 通常用作一个完整的关联操作，用于性能评估，而 SELECT 关注分片优化可用作性能评估的另一个操作；而主从模式下，可将 INSERT+SELECT+DELETE 作为一组评估性能的关联操作。为了更好的观察效果，设计在一定数据量的基础上，使用 jmeter 20 并发线程持续压测半小时，进行增删改查性能测试，且每台机器部署一个 MySQL 实例，而对比 MySQL 场景为单机单实例部署。

测试场景

单路由

在 1000 数据量的基础上分库分表，根据 `id` 分为 4 个库，部署在同一台机器上，根据 `k` 分为 1024 个表，查询操作路由到单库单表；作为对比，MySQL 运行在 1000 数据量的基础上，使用 INSERT+UPDATE+DELETE 和单路由查询语句。

主从

基本主从场景，设置一主库一从库，部署在两台不同的机器上，在 10000 数据量的基础上，观察读写性能；作为对比，MySQL 运行在 10000 数据量的基础上，使用 INSERT+SELECT+DELETE 语句。

主从 + 加密 + 分库分表

在 1000 数据量的基础上，根据 `id` 分为 4 个库，部署在四台不同的机器上，根据 `k` 分为 1024 个表，`c` 使用 aes 加密，`pad` 使用 md5 加密，查询操作路由到单库单表；作为对比，MySQL 运行在 1000 数据量的基础上，使用 INSERT+UPDATE+DELETE 和单路由查询语句。

全路由

在 1000 数据量的基础上，分库分表，根据 `id` 分为 4 个库，部署在四台不同的机器上，根据 `k` 分为 1 个表，查询操作使用全路由。作为对比，MySQL 运行在 1000 数据量的基础上，使用 INSERT+UPDATE+DELETE 和全路由查询语句。

测试环境搭建

数据库表结构

此处表结构参考 sysbench 的 sbtest 表。

```
CREATE TABLE `tbl` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT,
  `k` int(11) NOT NULL DEFAULT 0,
  `c` char(120) NOT NULL DEFAULT '',
  `pad` char(60) NOT NULL DEFAULT '',
  PRIMARY KEY (`id`)
);
```

测试场景配置

ShardingSphere-JDBC 使用与 ShardingSphere-Proxy 一致的配置，MySQL 直连一个库用作性能对比，下面为四个场景的具体配置：

单路由配置

```
schemaName: sharding_db

dataSources:
  ds_0:
    url: jdbc:mysql://***.***.***.***:****/ds?serverTimezone=UTC&useSSL=false
    username: test
    password:
    connectionTimeoutMilliseconds: 30000
    idleTimeoutMilliseconds: 60000
    maxLifetimeMilliseconds: 1800000
    maxPoolSize: 200
  ds_1:
    url: jdbc:mysql://***.***.***.***:****/ds?serverTimezone=UTC&useSSL=false
    username: test
    password:
    connectionTimeoutMilliseconds: 30000
    idleTimeoutMilliseconds: 60000
    maxLifetimeMilliseconds: 1800000
    maxPoolSize: 200
  ds_2:
    url: jdbc:mysql://***.***.***.***:****/ds?serverTimezone=UTC&useSSL=false
    username: test
    password:
    connectionTimeoutMilliseconds: 30000
    idleTimeoutMilliseconds: 60000
    maxLifetimeMilliseconds: 1800000
    maxPoolSize: 200
  ds_3:
    url: jdbc:mysql://***.***.***.***:****/ds?serverTimezone=UTC&useSSL=false
    username: test
    password:
    connectionTimeoutMilliseconds: 30000
    idleTimeoutMilliseconds: 60000
    maxLifetimeMilliseconds: 1800000
    maxPoolSize: 200
rules:
- !SHARDING
  tables:
    tbl:
      actualDataNodes: ds_${0..3}.tbl${0..1023}
      tableStrategy:
        standard:
          shardingColumn: k
          shardingAlgorithmName: tbl_table_inline
      keyGenerateStrategy:
          column: id
```

```
          keyGeneratorName: snowflake
  defaultDatabaseStrategy:
    standard:
      shardingColumn: id
      shardingAlgorithmName: default_db_inline
  defaultTableStrategy:
    none:
  shardingAlgorithms:
    tbl_table_inline:
      type: INLINE
      props:
        algorithm-expression: tbl${k % 1024}
    default_db_inline:
      type: INLINE
      props:
        algorithm-expression: ds_${id % 4}
  keyGenerators:
    snowflake:
      type: SNOWFLAKE
```

主从配置

```
schemaName: sharding_db

dataSources:
  primary_ds:
    url: jdbc:mysql://***.***.***.***:****/ds?serverTimezone=UTC&useSSL=false
    username: test
    password:
    connectionTimeoutMilliseconds: 30000
    idleTimeoutMilliseconds: 60000
    maxLifetimeMilliseconds: 1800000
    maxPoolSize: 200
  replica_ds_0:
    url: jdbc:mysql://***.***.***.***:****/ds?serverTimezone=UTC&useSSL=false
    username: test
    password:
    connectionTimeoutMilliseconds: 30000
    idleTimeoutMilliseconds: 60000
    maxLifetimeMilliseconds: 1800000
    maxPoolSize: 200
rules:
- !READWRITE_SPLITTING
  dataSources:
    readwrite_ds:
      type: Static
      props:
```

```
        write-data-source-name: primary_ds
        read-data-source-names: replica_ds_0
```

主从 + 加密 + 分库分表配置

```
schemaName: sharding_db

dataSources:
  primary_ds_0:
    url: jdbc:mysql://***.***.***.***:****/ds?serverTimezone=UTC&useSSL=false
    username: test
    password:
    connectionTimeoutMilliseconds: 30000
    idleTimeoutMilliseconds: 60000
    maxLifetimeMilliseconds: 1800000
    maxPoolSize: 200
  replica_ds_0:
    url: jdbc:mysql://***.***.***.***:****/ds?serverTimezone=UTC&useSSL=false
    username: test
    password:
    connectionTimeoutMilliseconds: 30000
    idleTimeoutMilliseconds: 60000
    maxLifetimeMilliseconds: 1800000
    maxPoolSize: 200
  primary_ds_1:
    url: jdbc:mysql://***.***.***.***:****/ds?serverTimezone=UTC&useSSL=false
    username: test
    password:
    connectionTimeoutMilliseconds: 30000
    idleTimeoutMilliseconds: 60000
    maxLifetimeMilliseconds: 1800000
    maxPoolSize: 200
  replica_ds_1:
    url: jdbc:mysql://***.***.***.***:****/ds?serverTimezone=UTC&useSSL=false
    username: test
    password:
    connectionTimeoutMilliseconds: 30000
    idleTimeoutMilliseconds: 60000
    maxLifetimeMilliseconds: 1800000
    maxPoolSize: 200
  primary_ds_2:
    url: jdbc:mysql://***.***.***.***:****/ds?serverTimezone=UTC&useSSL=false
    username: test
    password:
    connectionTimeoutMilliseconds: 30000
    idleTimeoutMilliseconds: 60000
    maxLifetimeMilliseconds: 1800000
```

```
    maxPoolSize: 200
  replica_ds_2:
    url: jdbc:mysql://***.***.***.***:****/ds?serverTimezone=UTC&useSSL=false
    username: test
    password:
    connectionTimeoutMilliseconds: 30000
    idleTimeoutMilliseconds: 60000
    maxLifetimeMilliseconds: 1800000
    maxPoolSize: 200
  primary_ds_3:
    url: jdbc:mysql://***.***.***.***:****/ds?serverTimezone=UTC&useSSL=false
    username: test
    password:
    connectionTimeoutMilliseconds: 30000
    idleTimeoutMilliseconds: 60000
    maxLifetimeMilliseconds: 1800000
    maxPoolSize: 200
  replica_ds_3:
    url: jdbc:mysql://***.***.***.***:****/ds?serverTimezone=UTC&useSSL=false
    username: test
    password:
    connectionTimeoutMilliseconds: 30000
    idleTimeoutMilliseconds: 60000
    maxLifetimeMilliseconds: 1800000
    maxPoolSize: 200
rules:
- !SHARDING
  tables:
    tbl:
      actualDataNodes: readwrite_ds_${0..3}.tbl${0..1023}
      databaseStrategy:
        standard:
          shardingColumn: id
          shardingAlgorithmName: tbl_database_inline
      tableStrategy:
        standard:
          shardingColumn: k
          shardingAlgorithmName: tbl_table_inline
      keyGenerateStrategy:
        column: id
        keyGeneratorName: snowflake
  bindingTables:
    - tbl
  defaultDataSourceName: primary_ds_1
  defaultTableStrategy:
    none:
  shardingAlgorithms:
    tbl_database_inline:
```

```
      type: INLINE
      props:
        algorithm-expression: readwrite_ds_${id % 4}
    tbl_table_inline:
      type: INLINE
      props:
        algorithm-expression: tbl${k % 1024}
  keyGenerators:
    snowflake:
      type: SNOWFLAKE
- !READWRITE_SPLITTING
  dataSources:
    readwrite_ds_0:
      type: Static
      props:
        write-data-source-name: primary_ds_0
        read-data-source-names: replica_ds_0
    readwrite_ds_1:
      type: Static
      props:
        write-data-source-name: primary_ds_1
        read-data-source-names: replica_ds_1
      loadBalancerName: round_robin
    readwrite_ds_2:
      type: Static
      props:
        write-data-source-name: primary_ds_2
        read-data-source-names: replica_ds_2
      loadBalancerName: round_robin
    readwrite_ds_3:
      type: Static
      props:
        write-data-source-name: primary_ds_3
        read-data-source-names: replica_ds_3
      loadBalancerName: round_robin
  loadBalancers:
    round_robin:
      type: ROUND_ROBIN
- !ENCRYPT:
  encryptors:
    aes_encryptor:
      type: AES
      props:
        aes-key-value: 123456abc
    md5_encryptor:
      type: MD5
  tables:
    sbtest:
```

```
      columns:
        c:
          plainColumn: c_plain
          cipherColumn: c_cipher
          encryptorName: aes_encryptor
        pad:
          cipherColumn: pad_cipher
          encryptorName: md5_encryptor
```

全路由

```
schemaName: sharding_db

dataSources:
  ds_0:
    url: jdbc:mysql://***.***.***.***:****/ds?serverTimezone=UTC&useSSL=false
    username: test
    password:
    connectionTimeoutMilliseconds: 30000
    idleTimeoutMilliseconds: 60000
    maxLifetimeMilliseconds: 1800000
    maxPoolSize: 200
  ds_1:
    url: jdbc:mysql://***.***.***.***:****/ds?serverTimezone=UTC&useSSL=false
    username: test
    password:
    connectionTimeoutMilliseconds: 30000
    idleTimeoutMilliseconds: 60000
    maxLifetimeMilliseconds: 1800000
    maxPoolSize: 200
  ds_2:
    url: jdbc:mysql://***.***.***.***:****/ds?serverTimezone=UTC&useSSL=false
    username: test
    password:
    connectionTimeoutMilliseconds: 30000
    idleTimeoutMilliseconds: 60000
    maxLifetimeMilliseconds: 1800000
    maxPoolSize: 200
  ds_3:
    url: jdbc:mysql://***.***.***.***:****/ds?serverTimezone=UTC&useSSL=false
    username: test
    password:
    connectionTimeoutMilliseconds: 30000
    idleTimeoutMilliseconds: 60000
    maxLifetimeMilliseconds: 1800000
    maxPoolSize: 200
rules:
```

```yaml
- !SHARDING
  tables:
    tbl:
      actualDataNodes: ds_${0..3}.tbl1
      tableStrategy:
        standard:
          shardingColumn: k
          shardingAlgorithmName: tbl_table_inline
      keyGenerateStrategy:
        column: id
        keyGeneratorName: snowflake
  defaultDatabaseStrategy:
    standard:
      shardingColumn: id
      shardingAlgorithmName: default_database_inline
  defaultTableStrategy:
    none:
  shardingAlgorithms:
    default_database_inline:
      type: INLINE
      props:
        algorithm-expression: ds_${id % 4}
    tbl_table_inline:
      type: INLINE
      props:
        algorithm-expression: tbl1
  keyGenerators:
    snowflake:
      type: SNOWFLAKE
```

测试结果验证

压测语句

```
INSERT+UPDATE+DELETE 语句:
INSERT INTO tbl(k, c, pad) VALUES(1, '###-###-###', '###-###');
UPDATE tbl SET c='####-####-####', pad='####-####' WHERE id=?;
DELETE FROM tbl WHERE id=?

全路由查询语句:
SELECT max(id) FROM tbl WHERE id%4=1

单路由查询语句:
SELECT id, k FROM tbl ignore index(`PRIMARY`) WHERE id=1 AND k=1

INSERT+SELECT+DELETE 语句:
```

```
INSERT INTO tbl1(k, c, pad) VALUES(1, '###-###-###', '###-###');
SELECT count(id) FROM tbl1;
SELECT max(id) FROM tbl1 ignore index(`PRIMARY`);
DELETE FROM tbl1 WHERE id=?
```

压测类

参考 shardingsphere-benchmark 实现，注意阅读其中的注释。

编译

```
git clone https://github.com/apache/shardingsphere-benchmark.git
cd shardingsphere-benchmark/shardingsphere-benchmark
mvn clean install
```

压测执行

```
cp target/shardingsphere-benchmark-1.0-SNAPSHOT-jar-with-dependencies.jar apache-
jmeter-4.0/lib/ext
jmeter -n -t test_plan/test.jmx
test.jmx 参考 https://github.com/apache/shardingsphere-benchmark/tree/master/report/
script/test_plan/test.jmx
```

压测结果处理

注意修改为上一步生成的 result.jtl 的位置。

```
sh shardingsphere-benchmark/report/script/gen_report.sh
```

历史压测数据展示

正在进行中，请等待。

BenchmarkSQL 性能测试

测试方法

ShardingSphere Proxy 支持通过 BenchmarkSQL 5.0 进行 TPC-C 测试。除本文说明的内容外，BenchmarkSQL 操作步骤按照原文档 `HOW-TO-RUN.txt` 即可。

测试工具微调

与单机数据库压测不同，分布式数据库解决方案难免在功能支持上有所取舍。使用 BenchmarkSQL 压测 ShardingSphere Proxy 建议进行如下调整。

移除外键与 extraHistID

修改 BenchmarkSQL 目录下 `run/runDatabaseBuild.sh`，文件第 17 行。

修改前：

```
AFTER_LOAD="indexCreates foreignKeys extraHistID buildFinish"
```

修改后：

```
AFTER_LOAD="indexCreates buildFinish"
```

压测环境或参数建议

注意：本节中提到的任何参数都不是绝对值，都需要根据实际测试结果进行调整或取舍。

建议使用 Java 17 运行 ShardingSphere

编译 ShardingSphere 可以使用 Java 8。

使用 Java 17 可以在默认情况下尽量提升 ShardingSphere 的性能。

ShardingSphere 数据分片建议

对 BenchmarkSQL 的数据分片，可以考虑以各个表中的 warehouse id 作为分片键。

其中一个表 `bmsql_item` 没有 warehouse id，数据量固定 10 万行：- 可以取 `i_id` 作为分片键。但可能会导致同一个 Proxy 连接同时持有多个不同数据源的连接。- 或考虑不做分片，存在单个数据源内。可能会导致某一数据源压力较大。- 或对 `i_id` 进行范围分片，例如 1-50000 分布在数据源 0、50001-100000 分布在数据源 1。

BenchmarkSQL 中有如下 SQL 涉及多表：

```
SELECT c_discount, c_last, c_credit, w_tax
FROM bmsql_customer
    JOIN bmsql_warehouse ON (w_id = c_w_id)
WHERE c_w_id = ? AND c_d_id = ? AND c_id = ?
```

```
SELECT o_id, o_entry_d, o_carrier_id
FROM bmsql_oorder
WHERE o_w_id = ? AND o_d_id = ? AND o_c_id = ?
  AND o_id = (
```

```
    SELECT max(o_id)
    FROM bmsql_oorder
    WHERE o_w_id = ? AND o_d_id = ? AND o_c_id = ?
    )
```

如果以 warehouse id 作为分片键，以上 SQL 涉及的表可以配置为 bindingTable：

```
rules:
  - !SHARDING
    bindingTables:
      - bmsql_warehouse, bmsql_customer
      - bmsql_stock, bmsql_district, bmsql_order_line
```

以 warehouse id 为分片键的数据分片配置可以参考本文附录。

PostgreSQL JDBC URL 参数建议

对 BenchmarkSQL 所使用的配置文件中的 JDBC URL 进行调整，即参数名 `conn` 的值：- 增加参数 `defaultRowFetchSize=50` 可能减少多行结果集的 fetch 次数，需要根据实际测试结果适当增大或减小。- 增加参数 `reWriteBatchedInserts=true` 可能减少批量插入的耗时，例如准备数据或 New Order 业务的批量插入，需要根据实际测试结果决定是否启用。

props.pg 文件节选，建议修改的位置为第 3 行 `conn` 的参数值：

```
db=postgres
driver=org.postgresql.Driver
conn=jdbc:postgresql://localhost:5432/postgres?defaultRowFetchSize=50&
reWriteBatchedInserts=true
user=benchmarksql
password=PWbmsql
```

ShardingSphere Proxy server.yaml 参数建议

`proxy-backend-query-fetch-size` 参数值默认值为 -1，修改为 `50` 左右可以尽量减少多行结果集的 fetch 次数。`proxy-frontend-executor-size` 参数默认值为 CPU * 2，可以根据实际测试结果减少至 CPU * 0.5 左右；如果涉及 NUMA，可以根据实际测试结果设置为单个 CPU 的物理核数。

`server.yaml` 文件节选：

```
props:
  proxy-backend-query-fetch-size: 50
  # proxy-frontend-executor-size: 32 # 4 路 32C aarch64
  # proxy-frontend-executor-size: 12 # 2 路 12C24T x86
```

附录

BenchmarkSQL 数据分片参考配置

Pool size 请根据实际压测情况适当调整。

```
schemaName: bmsql_sharding
dataSources:
  ds_0:
    url: jdbc:postgresql://db0.ip:5432/bmsql
    username: postgres
    password: postgres
    connectionTimeoutMilliseconds: 3000
    idleTimeoutMilliseconds: 60000
    maxLifetimeMilliseconds: 1800000
    maxPoolSize: 1000
    minPoolSize: 1000
  ds_1:
    url: jdbc:postgresql://db1.ip:5432/bmsql
    username: postgres
    password: postgres
    connectionTimeoutMilliseconds: 3000
    idleTimeoutMilliseconds: 60000
    maxLifetimeMilliseconds: 1800000
    maxPoolSize: 1000
    minPoolSize: 1000
  ds_2:
    url: jdbc:postgresql://db2.ip:5432/bmsql
    username: postgres
    password: postgres
    connectionTimeoutMilliseconds: 3000
    idleTimeoutMilliseconds: 60000
    maxLifetimeMilliseconds: 1800000
    maxPoolSize: 1000
    minPoolSize: 1000
  ds_3:
    url: jdbc:postgresql://db3.ip:5432/bmsql
    username: postgres
    password: postgres
    connectionTimeoutMilliseconds: 3000
    idleTimeoutMilliseconds: 60000
    maxLifetimeMilliseconds: 1800000
    maxPoolSize: 1000
    minPoolSize: 1000

rules:
  - !SHARDING
    bindingTables:
```

```
    - bmsql_warehouse, bmsql_customer
    - bmsql_stock, bmsql_district, bmsql_order_line
  defaultDatabaseStrategy:
    none:
  defaultTableStrategy:
    none:
  keyGenerators:
    snowflake:
      type: SNOWFLAKE
  tables:
    bmsql_config:
      actualDataNodes: ds_0.bmsql_config

    bmsql_warehouse:
      actualDataNodes: ds_${0..3}.bmsql_warehouse
      databaseStrategy:
        standard:
          shardingColumn: w_id
          shardingAlgorithmName: mod_4

    bmsql_district:
      actualDataNodes: ds_${0..3}.bmsql_district
      databaseStrategy:
        standard:
          shardingColumn: d_w_id
          shardingAlgorithmName: mod_4

    bmsql_customer:
      actualDataNodes: ds_${0..3}.bmsql_customer
      databaseStrategy:
        standard:
          shardingColumn: c_w_id
          shardingAlgorithmName: mod_4

    bmsql_item:
      actualDataNodes: ds_${0..3}.bmsql_item
      databaseStrategy:
        standard:
          shardingColumn: i_id
          shardingAlgorithmName: mod_4

    bmsql_history:
      actualDataNodes: ds_${0..3}.bmsql_history
      databaseStrategy:
        standard:
          shardingColumn: h_w_id
          shardingAlgorithmName: mod_4
```

```
      bmsql_oorder:
        actualDataNodes: ds_${0..3}.bmsql_oorder
        databaseStrategy:
          standard:
            shardingColumn: o_w_id
            shardingAlgorithmName: mod_4

      bmsql_stock:
        actualDataNodes: ds_${0..3}.bmsql_stock
        databaseStrategy:
          standard:
            shardingColumn: s_w_id
            shardingAlgorithmName: mod_4

      bmsql_new_order:
        actualDataNodes: ds_${0..3}.bmsql_new_order
        databaseStrategy:
          standard:
            shardingColumn: no_w_id
            shardingAlgorithmName: mod_4

      bmsql_order_line:
        actualDataNodes: ds_${0..3}.bmsql_order_line
        databaseStrategy:
          standard:
            shardingColumn: ol_w_id
            shardingAlgorithmName: mod_4

    shardingAlgorithms:
      mod_4:
        type: MOD
        props:
          sharding-count: 4
```

BenchmarkSQL 5.0 PostgreSQL 语句列表

Create tables

```
create table bmsql_config (
  cfg_name    varchar(30) primary key,
  cfg_value   varchar(50)
);

create table bmsql_warehouse (
  w_id        integer   not null,
  w_ytd       decimal(12,2),
```

```
  w_tax       decimal(4,4),
  w_name      varchar(10),
  w_street_1  varchar(20),
  w_street_2  varchar(20),
  w_city      varchar(20),
  w_state     char(2),
  w_zip       char(9)
);

create table bmsql_district (
  d_w_id       integer       not null,
  d_id         integer       not null,
  d_ytd        decimal(12,2),
  d_tax        decimal(4,4),
  d_next_o_id  integer,
  d_name       varchar(10),
  d_street_1   varchar(20),
  d_street_2   varchar(20),
  d_city       varchar(20),
  d_state      char(2),
  d_zip        char(9)
);

create table bmsql_customer (
  c_w_id         integer        not null,
  c_d_id         integer        not null,
  c_id           integer        not null,
  c_discount     decimal(4,4),
  c_credit       char(2),
  c_last         varchar(16),
  c_first        varchar(16),
  c_credit_lim   decimal(12,2),
  c_balance      decimal(12,2),
  c_ytd_payment  decimal(12,2),
  c_payment_cnt  integer,
  c_delivery_cnt integer,
  c_street_1     varchar(20),
  c_street_2     varchar(20),
  c_city         varchar(20),
  c_state        char(2),
  c_zip          char(9),
  c_phone        char(16),
  c_since        timestamp,
  c_middle       char(2),
  c_data         varchar(500)
);

create sequence bmsql_hist_id_seq;
```

```
create table bmsql_history (
  hist_id  integer,
  h_c_id   integer,
  h_c_d_id integer,
  h_c_w_id integer,
  h_d_id   integer,
  h_w_id   integer,
  h_date   timestamp,
  h_amount decimal(6,2),
  h_data   varchar(24)
);

create table bmsql_new_order (
  no_w_id  integer   not null,
  no_d_id  integer   not null,
  no_o_id  integer   not null
);

create table bmsql_oorder (
  o_w_id       integer      not null,
  o_d_id       integer      not null,
  o_id         integer      not null,
  o_c_id       integer,
  o_carrier_id integer,
  o_ol_cnt     integer,
  o_all_local  integer,
  o_entry_d    timestamp
);

create table bmsql_order_line (
  ol_w_id         integer   not null,
  ol_d_id         integer   not null,
  ol_o_id         integer   not null,
  ol_number       integer   not null,
  ol_i_id         integer   not null,
  ol_delivery_d   timestamp,
  ol_amount       decimal(6,2),
  ol_supply_w_id  integer,
  ol_quantity     integer,
  ol_dist_info    char(24)
);

create table bmsql_item (
  i_id     integer      not null,
  i_name   varchar(24),
  i_price  decimal(5,2),
  i_data   varchar(50),
```

```
  i_im_id  integer
);

create table bmsql_stock (
  s_w_id       integer       not null,
  s_i_id       integer       not null,
  s_quantity   integer,
  s_ytd        integer,
  s_order_cnt  integer,
  s_remote_cnt integer,
  s_data       varchar(50),
  s_dist_01    char(24),
  s_dist_02    char(24),
  s_dist_03    char(24),
  s_dist_04    char(24),
  s_dist_05    char(24),
  s_dist_06    char(24),
  s_dist_07    char(24),
  s_dist_08    char(24),
  s_dist_09    char(24),
  s_dist_10    char(24)
);
```

Create indexes

```
alter table bmsql_warehouse add constraint bmsql_warehouse_pkey
  primary key (w_id);

alter table bmsql_district add constraint bmsql_district_pkey
  primary key (d_w_id, d_id);

alter table bmsql_customer add constraint bmsql_customer_pkey
  primary key (c_w_id, c_d_id, c_id);

create index bmsql_customer_idx1
  on  bmsql_customer (c_w_id, c_d_id, c_last, c_first);

alter table bmsql_oorder add constraint bmsql_oorder_pkey
  primary key (o_w_id, o_d_id, o_id);

create unique index bmsql_oorder_idx1
  on  bmsql_oorder (o_w_id, o_d_id, o_carrier_id, o_id);

alter table bmsql_new_order add constraint bmsql_new_order_pkey
  primary key (no_w_id, no_d_id, no_o_id);

alter table bmsql_order_line add constraint bmsql_order_line_pkey
```

```
  primary key (ol_w_id, ol_d_id, ol_o_id, ol_number);

alter table bmsql_stock add constraint bmsql_stock_pkey
  primary key (s_w_id, s_i_id);

alter table bmsql_item add constraint bmsql_item_pkey
  primary key (i_id);
```

New Order 业务

stmtNewOrderSelectWhseCust

```
UPDATE bmsql_district
    SET d_next_o_id = d_next_o_id + 1
    WHERE d_w_id = ? AND d_id = ?
```

stmtNewOrderSelectDist

```
SELECT d_tax, d_next_o_id
    FROM bmsql_district
    WHERE d_w_id = ? AND d_id = ?
    FOR UPDATE
```

stmtNewOrderUpdateDist

```
UPDATE bmsql_district
    SET d_next_o_id = d_next_o_id + 1
    WHERE d_w_id = ? AND d_id = ?
```

stmtNewOrderInsertOrder

```
INSERT INTO bmsql_oorder (
    o_id, o_d_id, o_w_id, o_c_id, o_entry_d,
    o_ol_cnt, o_all_local)
VALUES (?, ?, ?, ?, ?, ?, ?)
```

stmtNewOrderInsertNewOrder

```
INSERT INTO bmsql_new_order (
    no_o_id, no_d_id, no_w_id)
VALUES (?, ?, ?)
```

stmtNewOrderSelectStock

```
SELECT s_quantity, s_data,
       s_dist_01, s_dist_02, s_dist_03, s_dist_04,
       s_dist_05, s_dist_06, s_dist_07, s_dist_08,
       s_dist_09, s_dist_10
```

```
    FROM bmsql_stock
    WHERE s_w_id = ? AND s_i_id = ?
    FOR UPDATE
```

stmtNewOrderSelectItem

```
SELECT i_price, i_name, i_data
    FROM bmsql_item
    WHERE i_id = ?
```

stmtNewOrderUpdateStock

```
UPDATE bmsql_stock
    SET s_quantity = ?, s_ytd = s_ytd + ?,
        s_order_cnt = s_order_cnt + 1,
        s_remote_cnt = s_remote_cnt + ?
    WHERE s_w_id = ? AND s_i_id = ?
```

stmtNewOrderInsertOrderLine

```
INSERT INTO bmsql_order_line (
    ol_o_id, ol_d_id, ol_w_id, ol_number,
    ol_i_id, ol_supply_w_id, ol_quantity,
    ol_amount, ol_dist_info)
VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?)
```

Payment 业务

stmtPaymentSelectWarehouse

```
SELECT w_name, w_street_1, w_street_2, w_city,
       w_state, w_zip
    FROM bmsql_warehouse
    WHERE w_id = ?
```

stmtPaymentSelectDistrict

```
SELECT d_name, d_street_1, d_street_2, d_city,
       d_state, d_zip
    FROM bmsql_district
    WHERE d_w_id = ? AND d_id = ?
```

stmtPaymentSelectCustomerListByLast

```
SELECT c_id
    FROM bmsql_customer
    WHERE c_w_id = ? AND c_d_id = ? AND c_last = ?
    ORDER BY c_first
```

stmtPaymentSelectCustomer

```
SELECT c_first, c_middle, c_last, c_street_1, c_street_2,
       c_city, c_state, c_zip, c_phone, c_since, c_credit,
       c_credit_lim, c_discount, c_balance
    FROM bmsql_customer
    WHERE c_w_id = ? AND c_d_id = ? AND c_id = ?
    FOR UPDATE
```

stmtPaymentSelectCustomerData

```
SELECT c_data
    FROM bmsql_customer
    WHERE c_w_id = ? AND c_d_id = ? AND c_id = ?
```

stmtPaymentUpdateWarehouse

```
UPDATE bmsql_warehouse
    SET w_ytd = w_ytd + ?
    WHERE w_id = ?
```

stmtPaymentUpdateDistrict

```
UPDATE bmsql_district
    SET d_ytd = d_ytd + ?
    WHERE d_w_id = ? AND d_id = ?
```

stmtPaymentUpdateCustomer

```
UPDATE bmsql_customer
    SET c_balance = c_balance - ?,
        c_ytd_payment = c_ytd_payment + ?,
        c_payment_cnt = c_payment_cnt + 1
    WHERE c_w_id = ? AND c_d_id = ? AND c_id = ?
```

stmtPaymentUpdateCustomerWithData

```
UPDATE bmsql_customer
    SET c_balance = c_balance - ?,
        c_ytd_payment = c_ytd_payment + ?,
        c_payment_cnt = c_payment_cnt + 1,
        c_data = ?
    WHERE c_w_id = ? AND c_d_id = ? AND c_id = ?
```

stmtPaymentInsertHistory

```
INSERT INTO bmsql_history (
    h_c_id, h_c_d_id, h_c_w_id, h_d_id, h_w_id,
    h_date, h_amount, h_data)
VALUES (?, ?, ?, ?, ?, ?, ?, ?)
```

Order Status 业务

stmtOrderStatusSelectCustomerListByLast

```
SELECT c_id
    FROM bmsql_customer
    WHERE c_w_id = ? AND c_d_id = ? AND c_last = ?
    ORDER BY c_first
```

stmtOrderStatusSelectCustomer

```
SELECT c_first, c_middle, c_last, c_balance
    FROM bmsql_customer
    WHERE c_w_id = ? AND c_d_id = ? AND c_id = ?
```

stmtOrderStatusSelectLastOrder

```
SELECT o_id, o_entry_d, o_carrier_id
    FROM bmsql_oorder
    WHERE o_w_id = ? AND o_d_id = ? AND o_c_id = ?
      AND o_id = (
          SELECT max(o_id)
              FROM bmsql_oorder
              WHERE o_w_id = ? AND o_d_id = ? AND o_c_id = ?
          )
```

stmtOrderStatusSelectOrderLine

```
SELECT ol_i_id, ol_supply_w_id, ol_quantity,
       ol_amount, ol_delivery_d
    FROM bmsql_order_line
    WHERE ol_w_id = ? AND ol_d_id = ? AND ol_o_id = ?
    ORDER BY ol_w_id, ol_d_id, ol_o_id, ol_number
```

Stock level 业务

stmtStockLevelSelectLow

```
SELECT count(*) AS low_stock FROM (
    SELECT s_w_id, s_i_id, s_quantity
        FROM bmsql_stock
        WHERE s_w_id = ? AND s_quantity < ? AND s_i_id IN (
            SELECT ol_i_id
                FROM bmsql_district
                JOIN bmsql_order_line ON ol_w_id = d_w_id
                 AND ol_d_id = d_id
                 AND ol_o_id >= d_next_o_id - 20
                 AND ol_o_id < d_next_o_id
```

```
                WHERE d_w_id = ? AND d_id = ?
        )
    ) AS L
```

Delivery BG 业务

stmtDeliveryBGSelectOldestNewOrder

```
SELECT no_o_id
    FROM bmsql_new_order
    WHERE no_w_id = ? AND no_d_id = ?
    ORDER BY no_o_id ASC
```

stmtDeliveryBGDeleteOldestNewOrder

```
DELETE FROM bmsql_new_order
    WHERE no_w_id = ? AND no_d_id = ? AND no_o_id = ?
```

stmtDeliveryBGSelectOrder

```
SELECT o_c_id
    FROM bmsql_oorder
    WHERE o_w_id = ? AND o_d_id = ? AND o_id = ?
```

stmtDeliveryBGUpdateOrder

```
UPDATE bmsql_oorder
    SET o_carrier_id = ?
    WHERE o_w_id = ? AND o_d_id = ? AND o_id = ?
```

stmtDeliveryBGSelectSumOLAmount

```
SELECT sum(ol_amount) AS sum_ol_amount
    FROM bmsql_order_line
    WHERE ol_w_id = ? AND ol_d_id = ? AND ol_o_id = ?
```

stmtDeliveryBGUpdateOrderLine

```
UPDATE bmsql_order_line
SET ol_delivery_d = ?
WHERE ol_w_id = ? AND ol_d_id = ? AND ol_o_id = ?
```

stmtDeliveryBGUpdateCustomer

```
UPDATE bmsql_customer
    SET c_balance = c_balance + ?,
        c_delivery_cnt = c_delivery_cnt + 1
    WHERE c_w_id = ? AND c_d_id = ? AND c_id = ?
```

9.7.6 模块测试

提供复杂模块的测试引擎。

SQL 解析测试

数据准备

SQL 解析无需真实的测试环境，开发者只需定义好待测试的 SQL，以及解析后的断言数据即可：

SQL 数据

在集成测试的部分提到过 `sql-case-id`，其对应的 SQL，可以在不同模块共享。开发者只需要在 `shardingsphere-sql-parser/shardingsphere-sql-parser-test/src/main/resources/sql/supported/${SQL-TYPE}/*.xml` 添加待测试的 SQL 即可。

断言数据

断言的解析数据保存在 `shardingsphere-sql-parser/shardingsphere-sql-parser-test/src/main/resources/case/${SQL-TYPE}/*.xml` 在 `xml` 文件中，可以针对表名，token，SQL 条件等进行断言，例如如下的配置：

```
<parser-result-sets>
    <parser-result sql-case-id="insert_with_multiple_values">
        <tables>
            <table name="t_order" />
        </tables>
        <tokens>
            <table-token start-index="12" table-name="t_order" length="7" />
        </tokens>
        <sharding-conditions>
            <and-condition>
                <condition column-name="order_id" table-name="t_order" operator=
"EQUAL">
                    <value literal="1" type="int" />
                </condition>
                <condition column-name="user_id" table-name="t_order" operator=
"EQUAL">
                    <value literal="1" type="int" />
                </condition>
            </and-condition>
            <and-condition>
                <condition column-name="order_id" table-name="t_order" operator=
"EQUAL">
                    <value literal="2" type="int" />
                </condition>
```

```
                <condition column-name="user_id" table-name="t_order" operator=
"EQUAL">
                    <value literal="2" type="int" />
                </condition>
            </and-condition>
        </sharding-conditions>
    </parser-result>
</parser-result-sets>
```

设置好上面两类数据，开发者就可以通过 `shardingsphere-sql-parser/shardingsphere-sql-parser-test` 下对应的测试引擎启动 SQL 解析的测试了。

SQL 改写测试

目标

面向逻辑库与逻辑表书写的 SQL，并不能够直接在真实的数据库中执行，SQL 改写用于将逻辑 SQL 改写为在真实数据库中可以正确执行的 SQL。它包括正确性改写和优化改写两部分，所以 SQL 改写的测试都是基于这些改写方向进行校验的。

测试

SQL 改写测试用例位于 `sharding-core/sharding-core-rewrite` 下的 test 中。SQL 改写的测试主要依赖如下几个部分：

- 测试引擎
- 环境配置
- 验证数据

测试引擎是 SQL 改写测试的入口，跟其他引擎一样，通过 Junit 的 Parameterized 逐条读取 `test\resources` 目录中测试类型下对应的 xml 文件，然后按读取顺序一一进行验证。

环境配置存放在 `test\resources\yaml` 路径中测试类型下对应的 yaml 中。配置了 dataSources,shardingRule，encryptRule 等信息，例子如下：

```
dataSources:
  db: !!com.zaxxer.hikari.HikariDataSource
    driverClassName: org.h2.Driver
    jdbcUrl: jdbc:h2:mem:db;DB_CLOSE_DELAY=-1;DATABASE_TO_UPPER=false;MODE=MYSQL
    username: sa
    password:

## sharding 规则
rules:
- !SHARDING
  tables:
    t_account:
```

```
      actualDataNodes: db.t_account_${0..1}
      tableStrategy:
        standard:
          shardingColumn: account_id
          shardingAlgorithmName: account_table_inline
      keyGenerateStrategy:
        column: account_id
        keyGeneratorName: snowflake
    t_account_detail:
      actualDataNodes: db.t_account_detail_${0..1}
      tableStrategy:
        standard:
          shardingColumn: order_id
          shardingAlgorithmName: account_detail_table_inline
  bindingTables:
    - t_account, t_account_detail
  shardingAlgorithms:
    account_table_inline:
      type: INLINE
      props:
        algorithm-expression: t_account_${account_id % 2}
    account_detail_table_inline:
      type: INLINE
      props:
        algorithm-expression: t_account_detail_${account_id % 2}
  keyGenerators:
    snowflake:
      type: SNOWFLAKE
```

验证数据存放在 `test\resources` 路径中测试类型下对应的 xml 文件中。验证数据中，`yaml-rule` 指定了环境以及 rule 的配置文件，`input` 指定了待测试的 SQL 以及参数，`output` 指定了期待的 SQL 以及参数。其中 `db-type` 决定了 SQL 解析的类型，默认为 `SQL92`，例如：

```
<rewrite-assertions yaml-rule="yaml/sharding/sharding-rule.yaml">
    <!-- 替换数据库类型需要在这里更改 db-type -->
    <rewrite-assertion id="create_index_for_mysql" db-type="MySQL">
        <input sql="CREATE INDEX index_name ON t_account ('status')" />
        <output sql="CREATE INDEX index_name ON t_account_0 ('status')" />
        <output sql="CREATE INDEX index_name ON t_account_1 ('status')" />
    </rewrite-assertion>
</rewrite-assertions>
```

只需在 xml 文件中编写测试数据，配置好相应的 yaml 配置文件，就可以在不更改任何 Java 代码的情况下校验对应的 SQL 了。

9.8 FAQ

9.8.1 [JDBC] 为什么配置了某个数据连接池的 spring-boot-starter（比如 druid）和 shardingsphere-jdbc-spring-boot-starter 时，系统启动会报错?

回答:

1. 因为数据连接池的 starter（比如 druid）可能会先加载并且其创建一个默认数据源，这将会使得 ShardingSphere-JDBC 创建数据源时发生冲突。
2. 解决办法为，去掉数据连接池的 starter 即可，ShardingSphere-JDBC 自己会创建数据连接池。

9.8.2 [JDBC] 使用 Spring 命名空间时找不到 xsd?

回答:

Spring 命名空间使用规范并未强制要求将 xsd 文件部署至公网地址，但考虑到部分用户的需求，我们也将相关 xsd 文件部署至 ShardingSphere 官网。

实际上 shardingsphere-jdbc-spring-namespace 的 jar 包中 META-INF:raw-latex:*spring*.schemas 配置了 xsd 文件的位置：META-INF:raw-latex:*namespace*:raw-latex:'\sharding'.xsd 和 META-INF:raw-latex:*namespace*:raw-latex:'\replica'-query.xsd，只需确保 jar 包中该文件存在即可。

9.8.3 [JDBC] 引入 `shardingsphere-transaction-xa-core` 后，如何避免 spring-boot 自动加载默认的 JtaTransactionManager?

回答:

1. 需要在 spring-boot 的引导类中添加 `@SpringBootApplication(exclude = JtaAutoConfiguration.class)`。

9.8.4 [Proxy] Windows 环境下，运行 ShardingSphere-Proxy，找不到或无法加载主类 org.apache.shardingsphere.proxy.Bootstrap，如何解决?

回答:

某些解压缩工具在解压 ShardingSphere-Proxy 二进制包时可能将文件名截断，导致找不到某些类。

解决方案:

打开 cmd.exe 并执行下面的命令:

```
tar zxvf apache-shardingsphere-${RELEASE.VERSION}-shardingsphere-proxy-bin.tar.gz
```

9.8.5 [Proxy] 在使用 ShardingSphere-Proxy 的时候,如何动态在添加新的 logic schema?

回答:

使用 ShardingSphere-Proxy 时，可以通过 `DistSQL` 动态的创建或移除 logic schema，语法如下:

```
CREATE (DATABASE | SCHEMA) [IF NOT EXISTS] schemaName;

DROP (DATABASE | SCHEMA) [IF EXISTS] schemaName;
```

例:

```
CREATE DATABASE sharding_db;

DROP SCHEMA sharding_db;
```

9.8.6 [Proxy] 在使用 ShardingSphere-Proxy 时，怎么使用合适的工具连接到 ShardingSphere-Proxy?

回答:

1. ShardingSphere-Proxy 可以看做是一个 database server，所以首选支持 SQL 命令连接和操作。
2. 如果使用其他第三方数据库工具，可能由于不同工具的特定实现导致出现异常。
3. 目前已测试的第三方数据库工具如下:
   - Navicat：11.1.13、15.0.20。
   - DataGrip：2020.1、2021.1（使用 IDEA/DataGrip 时打开 `introspect using JDBC metadata` 选项)。
   - WorkBench：8.0.25。

9.8.7 [Proxy] 使用 Navicat 等第三方数据库工具连接 ShardingSphere-Proxy 时，如果 ShardingSphere-Proxy 没有创建 Schema 或者没有添加 Resource，连接失败?

回答:

1. 第三方数据库工具在连接 ShardingSphere-Proxy 时会发送一些 SQL 查询元数据，当 ShardingSphere-Proxy 没有创建 `schema` 或者没有添加 `resource` 时，ShardingSphere-Proxy 无法执行 SQL。
2. 推荐先创建 `schema` 和 `resource` 之后再使用第三方数据库工具连接。
3. 有关 `resource` 的详情请参考。相关介绍

9.8.8 [分片] Cloud not resolve placeholder …in string value …异常的解决方法?

回答:

行表达式标识符可以使用 `${...}` 或 `$->{...}`，但前者与 Spring 本身的属性文件占位符冲突，因此在 Spring 环境中使用行表达式标识符建议使用 `$->{...}`。

9.8.9 [分片] inline 表达式返回结果为何出现浮点数?

回答:

Java 的整数相除结果是整数，但是对于 inline 表达式中的 Groovy 语法则不同，整数相除结果是浮点数。想获得除法整数结果需要将 A/B 改为 A.intdiv(B)。

9.8.10 [分片] 如果只有部分数据库分库分表，是否需要将不分库分表的表也配置在分片规则中?

回答:

不需要，ShardingSphere 会自动识别。

9.8.11 [分片] 指定了泛型为 Long 的 `SingleKeyTableShardingAlgorithm`，遇到 `ClassCastException: Integer can not cast to Long`?

回答:

必须确保数据库表中该字段和分片算法该字段类型一致，如：数据库中该字段类型为 int(11)，泛型所对应的分片类型应为 Integer，如果需要配置为 Long 类型，请确保数据库中该字段类型为 bigint。

9.8.12 [分片、PROXY] 实现 `StandardShardingAlgorithm` 自定义算法时，指定了 `Comparable` 的具体类型为 Long, 且数据库表中字段类型为 bigint，出现 `ClassCastException: Integer can not cast to Long` 异常。

回答:

实现 `doSharding` 方法时，不建议指定方法声明中 `Comparable` 具体的类型，而是在 `doSharding` 方法实现中对类型进行转换，可以参考 `ModShardingAlgorithm#doSharding` 方法

9.8.13 [分片] ShardingSphere 提供的默认分布式自增主键策略为什么是不连续的，且尾数大多为偶数?

回答:

ShardingSphere 采用 snowflake 算法作为默认的分布式自增主键策略，用于保证分布式的情况下可以无中心化的生成不重复的自增序列。因此自增主键可以保证递增，但无法保证连续。

而 snowflake 算法的最后 4 位是在同一毫秒内的访问递增值。因此，如果毫秒内并发度不高，最后 4 位为零的几率则很大。因此并发度不高的应用生成偶数主键的几率会更高。

在 3.1.0 版本中，尾数大多为偶数的问题已彻底解决，参见：https://github.com/apache/shardingsphere/issues/1617

9.8.14 [分片] 如何在 inline 分表策略时，允许执行范围查询操作（BETWEEN AND、>、<、>=、<=）?

回答:

1. 需要使用 4.1.0 或更高版本。
2. 调整以下配置项（需要注意的是，此时所有的范围查询将会使用广播的方式查询每一个分表）:

- 4.x 版本：`allow.range.query.with.inline.sharding` 设置为 true 即可（默认为 false）。
- 5.x 版本：在 InlineShardingStrategy 中将 `allow-range-query-with-inline-sharding` 设置为 true 即可（默认为 false）。

9.8.15 [分片] 为什么我实现了 `KeyGenerateAlgorithm` 接口，也配置了 Type，但是自定义的分布式主键依然不生效?

回答:

Service Provider Interface (SPI) 是一种为了被第三方实现或扩展的 API，除了实现接口外，还需要在 META-INF/services 中创建对应文件来指定 SPI 的实现类，JVM 才会加载这些服务。

具体的 SPI 使用方式，请大家自行搜索。

与分布式主键 `KeyGenerateAlgorithm` 接口相同，其他 ShardingSphere 的扩展功能也需要用相同的方式注入才能生效。

9.8.16 [分片] ShardingSphere 除了支持自带的分布式自增主键之外，还能否支持原生的自增主键?

回答：是的，可以支持。但原生自增主键有使用限制，即不能将原生自增主键同时作为分片键使用。

由于 ShardingSphere 并不知晓数据库的表结构,而原生自增主键是不包含在原始 SQL 中内的,因此 ShardingSphere 无法将该字段解析为分片字段。如自增主键非分片键，则无需关注，可正常返回；若自增主键同时作为分片键使用，ShardingSphere 无法解析其分片值，导致 SQL 路由至多张表，从而影响应用的正确性。

而原生自增主键返回的前提条件是 INSERT SQL 必须最终路由至一张表，因此，面对返回多表的 INSERT SQL，自增主键则会返回零。

9.8.17 [数据加密] JPA 和数据加密无法一起使用，如何解决?

回答:

由于数据加密的 DDL 尚未开发完成，因此对于自动生成 DDL 语句的 JPA 与数据加密一起使用时，会导致 JPA 的实体类（Entity）无法同时满足 DDL 和 DML 的情况。

解决方案如下:

1. 以需要加密的逻辑列名编写 JPA 的实体类（Entity）。
2. 关闭 JPA 的 auto-ddl，如 auto-ddl=none。
3. 手动建表，建表时应使用数据加密配置的 `cipherColumn,plainColumn` 和 `assistedQueryColumn` 代替逻辑列。

9.8.18 [DistSQL] 使用 DistSQL 添加数据源时，如何设置自定义的 JDBC 连接参数或连接池属性?

回答:

1. 如需自定义 JDBC 参数，请使用 `urlSource` 的方式定义 `dataSource`。
2. ShardingSphere 预置了必要的连接池参数，如 `maxPoolSize`、`idleTimeout` 等。如需增加或覆盖参数配置，请在 `dataSource` 中通过 `PROPERTIES` 指定。
3. 以上规则请参考 相关介绍。

9.8.19 [DistSQL] 使用 DistSQL 删除资源时，出现 Resource [xxx] is still used by [SingleTableRule]。

回答:

1. 被规则引用的资源将无法被删除。
2. 若资源只被 single table rule 引用，且用户确认可以忽略该限制，则可以添加可选参数 ignore single tables 进行强制删除。

9.8.20 [DistSQL] 使用 DistSQL 添加资源时，出现 Failed to get driver instance for jdbcURL=xxx。

回答:

ShardingSphere-Proxy 在部署过程中没有添加 jdbc 驱动，需要将 jdbc 驱动放入 ShardingSphere-Proxy 解压后的 ext-lib 目录，例如：`mysql-connector`。

9.8.21 [其他] 如果 SQL 在 ShardingSphere 中执行不正确，该如何调试?

回答:

在 ShardingSphere-Proxy 以及 ShardingSphere-JDBC 1.5.0 版本之后提供了 `sql.show` 的配置，可以将解析上下文和改写后的 SQL 以及最终路由至的数据源的细节信息全部打印至 info 日志。`sql.show` 配置默认关闭，如果需要请通过配置开启。

> 注意：5.x 版本以后，`sql.show` 参数调整为 `sql-show`。

9.8.22 [其他] 阅读源码时为什么会出现编译错误? IDEA 不索引生成的代码?

回答:

ShardingSphere 使用 lombok 实现极简代码。关于更多使用和安装细节，请参考 lombok 官网。

`org.apache.shardingsphere.sql.parser.autogen` 包下的代码由 ANTLR 生成，可以执行以下命令快速生成:

```
./mvnw -Dcheckstyle.skip=true -Drat.skip=true -Dmaven.javadoc.skip=true -Djacoco.skip=true -DskipITs -DskipTests install -T1C
```

生成的代码例如 `org.apache.shardingsphere.sql.parser.autogen.PostgreSQLStatementParser` 等 Java 文件由于较大，默认配置的 IDEA 可能不会索引该文件。可以调整 IDEA 的属性：`idea.max.intellisense.filesize=10000`。

9.8.23 [其他] 使用 SQLSever 和 PostgreSQL 时，聚合列不加别名会抛异常?

回答:

SQLServer 和 PostgreSQL 获取不加别名的聚合列会改名。例如，如下 SQL:

```
SELECT SUM(num), SUM(num2) FROM tablexxx;
```

SQLServer 获取到的列为空字符串和 (2)，PostgreSQL 获取到的列为空 sum 和 sum(2)。这将导致 ShardingSphere 在结果归并时无法找到相应的列而出错。

正确的 SQL 写法应为:

```
SELECT SUM(num) AS sum_num, SUM(num2) AS sum_num2 FROM tablexxx;
```

9.8.24 [其他] Oracle 数据库使用 Timestamp 类型的 Order By 语句抛出异常提示“Order by value must implements Comparable”？

回答:

针对上面问题解决方式有两种:

1. 配置启动 JVM 参数“-oracle.jdbc.J2EE13Compliant=true”
2. 通过代码在项目初始化时设置 System.getProperties().setProperty(“oracle.jdbc.J2EE13Compliant”, “true”);

原因如下:

`org.apache.shardingsphere.sharding.merge.dql.orderby.OrderByValue#getOrderValues()` 方法如下:

```
private List<Comparable<?>> getOrderValues() throws SQLException {
    List<Comparable<?>> result = new ArrayList<>(orderByItems.size());
    for (OrderItem each : orderByItems) {
        Object value = resultSet.getObject(each.getIndex());
        Preconditions.checkState(null == value || value instanceof Comparable,
"Order by value must implements Comparable");
        result.add((Comparable<?>) value);
    }
    return result;
}
```

使用了 resultSet.getObject(int index) 方法, 针对 TimeStamp oracle 会根据 oracle.jdbc.J2EE13Compliant 属性判断返回 java.sql.TimeStamp 还是自定义 oralce.sql.TIMESTAMP 详见 ojdbc 源码 oracle.jdbc.driver.TimestampAccessor#getObject(int var1) 方法:

```
Object getObject(int var1) throws SQLException {
    Object var2 = null;
    if(this.rowSpaceIndicator == null) {
        DatabaseError.throwSqlException(21);
    }

    if(this.rowSpaceIndicator[this.indicatorIndex + var1] != -1) {
        if(this.externalType != 0) {
            switch(this.externalType) {
            case 93:
                return this.getTimestamp(var1);
            default:
                DatabaseError.throwSqlException(4);
                return null;
            }
        }

        if(this.statement.connection.j2ee13Compliant) {
            var2 = this.getTimestamp(var1);
```

```
        } else {
            var2 = this.getTIMESTAMP(var1);
        }
    }

    return var2;
}
```

9.8.25 [其他] Windows 环境下，通过 Git 克隆 ShardingSphere 源码时为什么提示文件名过长，如何解决?

回答:

为保证源码的可读性，ShardingSphere 编码规范要求类、方法和变量的命名要做到顾名思义，避免使用缩写，因此可能导致部分源码文件命名较长。由于 Windows 版本的 Git 是使用 msys 编译的，它使用了旧版本的 Windows Api，限制文件名不能超过 260 个字符。

解决方案如下:

打开 cmd.exe（你需要将 git 添加到环境变量中）并执行下面的命令，可以让 git 支持长文件名:

```
git config --global core.longpaths true
```

如果是 Windows 10，还需要通过注册表或组策略，解除操作系统的文件名长度限制（需要重启） ：> 在注册表编辑器中创建 `HKLM\SYSTEM\CurrentControlSet\Control\FileSystem LongPathsEnabled`，类型为 `REG_DWORD`，并设置为 1。> 或者从系统菜单点击设置图标，输入“编辑组策略”，然后在打开的窗口依次进入“计算机管理”> “管理模板”> “系统”> “文件系统”，在右侧双击“启用 win32 长路径”。

参考资料：https://docs.microsoft.com/zh-cn/windows/desktop/FileIO/naming-a-file https://ourcodeworld.com/articles/read/109/how-to-solve-filename-too-long-error-in-git-powershell-and-github-application-for-windows

9.8.26 [其他] Type is required 异常的解决方法?

回答:

ShardingSphere 中很多功能实现类的加载方式是通过 SPI 注入的方式完成的，如分布式主键，注册中心等；这些功能通过配置中 type 类型来寻找对应的 SPI 实现，因此必须在配置文件中指定类型。

9.8.27 [其他] 服务启动时如何加快 metadata 加载速度?

回答:

1. 升级到 `4.0.1` 以上的版本，以提高 metadata 的加载速度。
2. 参照你采用的连接池，将:

- 配置项 `max.connections.size.per.query`(默认值为 1)调高(版本 >= 3.0.0.M3 且低于 5.0.0)。
- 配置项 `max-connections-size-per-query`（默认值为 1）调高（版本 >= 5.0.0)。

9.8.28 [其他] ANTLR 插件在 src 同级目录下生成代码，容易误提交，如何避免?

回答:

进入 Settings -> Languages & Frameworks -> ANTLR v4 default project settings 配置生成代码的输出目录为 `target/gen`，如图:

图 29: Configure ANTLR plugin

9.8.29 [其他] 使用 Proxool 时分库结果不正确?

回答:

使用 Proxool 配置多个数据源时，应该为每个数据源设置 alias，因为 Proxool 在获取连接时会判断连接池中是否包含已存在的 alias，不配置 alias 会造成每次都只从一个数据源中获取连接。

以下是 Proxool 源码中 ProxoolDataSource 类 getConnection 方法的关键代码:

```
if(!ConnectionPoolManager.getInstance().isPoolExists(this.alias)) {
    this.registerPool();
}
```

更多关于 alias 使用方法请参考 Proxool 官网。

PS：sourceforge 网站需要翻墙访问。

9.8.30 [其他] 使用 Spring Boot 2.x 集成 ShardingSphere 时，配置文件中的属性设置不生效?

回答:

需要特别注意，Spring Boot 2.x 环境下配置文件的属性名称约束为仅允许小写字母、数字和短横线，即 `[a-z][0-9]` 和 `-`。

原因如下:

Spring Boot 2.x 环境下，ShardingSphere 通过 Binder 来绑定配置文件，属性名称不规范（如：驼峰或下划线等）会导致属性设置不生效从而校验属性值时抛出 `NullPointerException` 异常。参考以下错误示例:

下划线示例：database_inline

```
spring.shardingsphere.rules.sharding.sharding-algorithms.database_inline.
type=INLINE
spring.shardingsphere.rules.sharding.sharding-algorithms.database_inline.props.
algorithm-expression=ds-$->{user_id % 2}
```

```
Caused by: org.springframework.beans.factory.BeanCreationException: Error creating
bean with name 'database_inline': Initialization of bean failed; nested exception
is java.lang.NullPointerException: Inline sharding algorithm expression cannot be
null.
    ...
Caused by: java.lang.NullPointerException: Inline sharding algorithm expression
cannot be null.
    at com.google.common.base.Preconditions.checkNotNull(Preconditions.java:897)
    at org.apache.shardingsphere.sharding.algorithm.sharding.inline.
InlineShardingAlgorithm.getAlgorithmExpression(InlineShardingAlgorithm.java:58)
    at org.apache.shardingsphere.sharding.algorithm.sharding.inline.
InlineShardingAlgorithm.init(InlineShardingAlgorithm.java:52)
    at org.apache.shardingsphere.spring.boot.registry.
AbstractAlgorithmProvidedBeanRegistry.
postProcessAfterInitialization(AbstractAlgorithmProvidedBeanRegistry.java:98)
```

```
    at org.springframework.beans.factory.support.
AbstractAutowireCapableBeanFactory.
applyBeanPostProcessorsAfterInitialization(AbstractAutowireCapableBeanFactory.
java:431)
    ...
```

驼峰示例：databaseInline

```
spring.shardingsphere.rules.sharding.sharding-algorithms.databaseInline.type=INLINE
spring.shardingsphere.rules.sharding.sharding-algorithms.databaseInline.props.
algorithm-expression=ds-$->{user_id % 2}
```

```
Caused by: org.springframework.beans.factory.BeanCreationException: Error creating
bean with name 'databaseInline': Initialization of bean failed; nested exception is
java.lang.NullPointerException: Inline sharding algorithm expression cannot be
null.
    ...
Caused by: java.lang.NullPointerException: Inline sharding algorithm expression
cannot be null.
    at com.google.common.base.Preconditions.checkNotNull(Preconditions.java:897)
    at org.apache.shardingsphere.sharding.algorithm.sharding.inline.
InlineShardingAlgorithm.getAlgorithmExpression(InlineShardingAlgorithm.java:58)
    at org.apache.shardingsphere.sharding.algorithm.sharding.inline.
InlineShardingAlgorithm.init(InlineShardingAlgorithm.java:52)
    at org.apache.shardingsphere.spring.boot.registry.
AbstractAlgorithmProvidedBeanRegistry.
postProcessAfterInitialization(AbstractAlgorithmProvidedBeanRegistry.java:98)
    at org.springframework.beans.factory.support.
AbstractAutowireCapableBeanFactory.
applyBeanPostProcessorsAfterInitialization(AbstractAutowireCapableBeanFactory.
java:431)
    ...
```

从异常堆栈中分析可知：`AbstractAlgorithmProvidedBeanRegistry.registerBean` 方法调用 `PropertyUtil.containPropertyPrefix(environment, prefix)` 方法判断指定前缀 `prefix` 的配置是否存在，而 `PropertyUtil.containPropertyPrefix(environment, prefix)` 方法，在 Spring Boot 2.x 环境下使用了 Binder，不规范的属性名称（如：驼峰或下划线等）会导致属性设置不生效。

9.9 API 变更历史

本章包含 Apache ShardingSphere 项目 API 的变更历史记录。

9.9.1 ShardingSphere-JDBC

本章包含 Apache ShardingSphere-JDBC API 的变更历史。

YAML 配置

5.0.0-alpha

数据分片

配置项说明

```
dataSources: # 省略数据源配置，请参考使用手册

rules:
- !SHARDING
  tables: # 数据分片规则配置
    <logic-table-name> (+): # 逻辑表名称
      actualDataNodes (?): # 由数据源名 + 表名组成（参考 Inline 语法规则）
      databaseStrategy (?): # 分库策略，缺省表示使用默认分库策略，以下的分片策略只能选其一
        standard: # 用于单分片键的标准分片场景
          shardingColumn: # 分片列名称
          shardingAlgorithmName: # 分片算法名称
        complex: # 用于多分片键的复合分片场景
          shardingColumns: # 分片列名称，多个列以逗号分隔
          shardingAlgorithmName: # 分片算法名称
        hint: # Hint 分片策略
          shardingAlgorithmName: # 分片算法名称
        none: # 不分片
      tableStrategy: # 分表策略，同分库策略
      keyGenerateStrategy: # 分布式序列策略
        column: # 自增列名称，缺省表示不使用自增主键生成器
        keyGeneratorName: # 分布式序列算法名称
  autoTables: # 自动分片表规则配置
    t_order_auto: # 逻辑表名称
      actualDataSources (?): # 数据源名称
      shardingStrategy: # 切分策略
        standard: # 用于单分片键的标准分片场景
          shardingColumn: # 分片列名称
          shardingAlgorithmName: # 自动分片算法名称
  bindingTables (+): # 绑定表规则列表
    - <logic_table_name_1, logic_table_name_2, ...>
```

```
    - <logic_table_name_1, logic_table_name_2, ...>
  broadcastTables (+): # 广播表规则列表
    - <table-name>
    - <table-name>
  defaultDatabaseStrategy: # 默认数据库分片策略
  defaultTableStrategy: # 默认表分片策略
  defaultKeyGenerateStrategy: # 默认的分布式序列策略

  # 分片算法配置
  shardingAlgorithms:
    <sharding-algorithm-name> (+): # 分片算法名称
      type: # 分片算法类型
      props: # 分片算法属性配置
      # ...

  # 分布式序列算法配置
  keyGenerators:
    <key-generate-algorithm-name> (+): # 分布式序列算法名称
      type: # 分布式序列算法类型
      props: # 分布式序列算法属性配置
      # ...

props:
  # ...
```

读写分离

配置项说明

```
dataSources: # 省略数据源配置，请参考使用手册

rules:
- !REPLICA_QUERY
  dataSources:
    <data-source-name> (+): # 读写分离逻辑数据源名称
      primaryDataSourceName: # 主库数据源名称
      replicaDataSourceNames:
        - <replica-data_source-name> (+) # 从库数据源名称
      loadBalancerName: # 负载均衡算法名称

  # 负载均衡算法配置
  loadBalancers:
    <load-balancer-name> (+): # 负载均衡算法名称
      type: # 负载均衡算法类型
      props: # 负载均衡算法属性配置
        # ...
```

```
props:
  # ...
```

算法类型的详情，请参见内置负载均衡算法列表。

数据加密

配置项说明

```
dataSource: # 省略数据源配置，请参考使用手册

rules:
- !ENCRYPT
  tables:
    <table-name> (+): # 加密表名称
      columns:
        <column-name> (+): # 加密列名称
          cipherColumn: # 密文列名称
          assistedQueryColumn (?):  # 查询辅助列名称
          plainColumn (?): # 原文列名称
          encryptorName: # 加密算法名称

  # 加密算法配置
  encryptors:
    <encrypt-algorithm-name> (+): # 加解密算法名称
      type: # 加解密算法类型
      props: # 加解密算法属性配置
        # ...

  queryWithCipherColumn: # 是否使用加密列进行查询。在有原文列的情况下，可以使用原文列进行查询
```

算法类型的详情，请参见内置加密算法列表。

影子库

配置项说明

```
dataSources: # 省略数据源配置，请参考使用手册

rules:
- !SHADOW
  column: # 影子字段名
  sourceDataSourceNames: # 影子前数据库名
     # ...
  shadowDataSourceNames: # 对应的影子库名
```

```
    # ...

props:
  # ...
```

分布式治理

配置项说明

```
governance:
  name: # 治理名称
  registryCenter: # 注册中心
    type: # 治理持久化类型。如: Zookeeper, etcd
    serverLists: # 治理服务列表。包括 IP 地址和端口号。多个地址用逗号分隔。如: host1:2181,
host2:2181
  overwrite: # 本地配置是否覆盖配置中心配置。如果可覆盖, 每次启动都以本地配置为准
```

ShardingSphere-4.x

数据分片

配置项说明

```
dataSources: # 数据源配置, 可配置多个 data_source_name
  <data_source_name>: # <!! 数据库连接池实现类> `!!`表示实例化该类
    driverClassName: # 数据库驱动类名
    url: # 数据库 url 连接
    username: # 数据库用户名
    password: # 数据库密码
    # ... 数据库连接池的其它属性

shardingRule:
  tables: # 数据分片规则配置, 可配置多个 logic_table_name
    <logic_table_name>: # 逻辑表名称
      actualDataNodes: # 由数据源名 + 表名组成, 以小数点分隔。多个表以逗号分隔, 支持 inline 表
达式。缺省表示使用已知数据源与逻辑表名称生成数据节点, 用于广播表 (即每个库中都需要一个同样的表用于关
联查询, 多为字典表) 或只分库不分表且所有库的表结构完全一致的情况

      databaseStrategy: # 分库策略, 缺省表示使用默认分库策略, 以下的分片策略只能选其一
        standard: # 用于单分片键的标准分片场景
          shardingColumn: # 分片列名称
          preciseAlgorithmClassName: # 精确分片算法类名称, 用于 = 和 IN。该类需实现
PreciseShardingAlgorithm 接口并提供无参数的构造器
          rangeAlgorithmClassName: # 范围分片算法类名称, 用于 BETWEEN, 可选。该类需实现
RangeShardingAlgorithm 接口并提供无参数的构造器
```

```
        complex: # 用于多分片键的复合分片场景
          shardingColumns: # 分片列名称，多个列以逗号分隔
          algorithmClassName: # 复合分片算法类名称。该类需实现
ComplexKeysShardingAlgorithm 接口并提供无参数的构造器
        inline: # 行表达式分片策略
          shardingColumn: # 分片列名称
          algorithmInlineExpression: # 分片算法行表达式，需符合 groovy 语法
        hint: # Hint 分片策略
          algorithmClassName: # Hint 分片算法类名称。该类需实现 HintShardingAlgorithm 接
口并提供无参数的构造器
        none: # 不分片
      tableStrategy: # 分表策略，同分库策略
      keyGenerator:
        column: # 自增列名称，缺省表示不使用自增主键生成器
        type: # 自增列值生成器类型，缺省表示使用默认自增列值生成器。可使用用户自定义的列值生成器
或选择内置类型：SNOWFLAKE/UUID
        props: # 属性配置，  注意：使用 SNOWFLAKE 算法，需要配置 max.tolerate.time.
difference.milliseconds 属性。若使用此算法生成值作分片值，建议配置 max.vibration.offset 属性
          <property-name>: # 属性名称

  bindingTables: # 绑定表规则列表
    - <logic_table_name1, logic_table_name2, ...>
    - <logic_table_name3, logic_table_name4, ...>
    - <logic_table_name_x, logic_table_name_y, ...>
  broadcastTables: # 广播表规则列表
    - table_name1
    - table_name2
    - table_name_x

  defaultDataSourceName: # 未配置分片规则的表将通过默认数据源定位
  defaultDatabaseStrategy: # 默认数据库分片策略，同分库策略
  defaultTableStrategy: # 默认表分片策略，同分库策略
  defaultKeyGenerator: # 默认的主键生成算法，如果没有设置，默认为 SNOWFLAKE 算法
    type: # 默认自增列值生成器类型，缺省将使用 org.apache.shardingsphere.core.keygen.
generator.impl.SnowflakeKeyGenerator。可使用用户自定义的列值生成器或选择内置类型：
SNOWFLAKE/UUID
    props:
      <property-name>: # 自增列值生成器属性配置，比如 SNOWFLAKE 算法的 max.tolerate.time.
difference.milliseconds

  masterSlaveRules: # 读写分离规则，详见读写分离部分
    <data_source_name>: # 数据源名称，需要与真实数据源匹配，可配置多个 data_source_name
      masterDataSourceName: # 详见读写分离部分
      slaveDataSourceNames: # 详见读写分离部分
      loadBalanceAlgorithmType: # 详见读写分离部分
      props: # 读写分离负载算法的属性配置
        <property-name>: # 属性值
```

```
props: # 属性配置
  sql.show: # 是否开启 SQL 显示，默认值：false
  executor.size: # 工作线程数量，默认值：CPU 核数
  max.connections.size.per.query: # 每个查询可以打开的最大连接数量，默认为 1
  check.table.metadata.enabled: # 是否在启动时检查分表元数据一致性，默认值：false
```

读写分离

配置项说明

```
dataSources: # 省略数据源配置，与数据分片一致

masterSlaveRule:
  name: # 读写分离数据源名称
  masterDataSourceName: # 主库数据源名称
  slaveDataSourceNames: # 从库数据源名称列表
    - <data_source_name1>
    - <data_source_name2>
    - <data_source_name_x>
  loadBalanceAlgorithmType: # 从库负载均衡算法类型，可选值：ROUND_ROBIN，RANDOM。若
`loadBalanceAlgorithmClassName` 存在则忽略该配置
  props: # 读写分离负载算法的属性配置
    <property-name>: # 属性值
```

通过 YamlMasterSlaveDataSourceFactory 工厂类创建 DataSource：

```
DataSource dataSource = YamlMasterSlaveDataSourceFactory.
createDataSource(yamlFile);
```

数据脱敏

配置项说明

```
dataSource: # 省略数据源配置

encryptRule:
  encryptors:
    <encryptor-name>:
      type: # 加解密器类型，可自定义或选择内置类型：MD5/AES
      props: # 属性配置，注意：使用 AES 加密器，需要配置 AES 加密器的 KEY 属性：aes.key.value
        aes.key.value:
  tables:
    <table-name>:
      columns:
        <logic-column-name>:
```

```
          plainColumn: # 存储明文的字段
          cipherColumn: # 存储密文的字段
          assistedQueryColumn: # 辅助查询字段，针对 ShardingQueryAssistedEncryptor 类型的加解密器进行辅助查询
          encryptor: # 加密器名字
```

治理

配置项说明

```
dataSources: # 省略数据源配置
shardingRule: # 省略分片规则配置
masterSlaveRule: # 省略读写分离规则配置
encryptRule: # 省略数据脱敏规则配置

orchestration:
  name: # 治理实例名称
  overwrite: # 本地配置是否覆盖注册中心配置。如果可覆盖，每次启动都以本地配置为准
  registry: # 注册中心配置
    type: # 配置中心类型。如：zookeeper
    serverLists: # 连接注册中心服务器的列表。包括 IP 地址和端口号。多个地址用逗号分隔。如：host1:2181,host2:2181
    namespace: # 注册中心的命名空间
    digest: # 连接注册中心的权限令牌。缺省为不需要权限验证
    operationTimeoutMilliseconds: # 操作超时的毫秒数，默认 500 毫秒
    maxRetries: # 连接失败后的最大重试次数，默认 3 次
    retryIntervalMilliseconds: # 重试间隔毫秒数，默认 500 毫秒
    timeToLiveSeconds: # 临时节点存活秒数，默认 60 秒
```

ShardingSphere-3.x

数据分片

配置项说明

```
# 以下配置截止版本为 3.1
# 配置文件中，必须配置的项目为 schemaName,dataSources，并且 shardingRule，masterSlaveRule，配置其中一个（注意，除非 server.yaml 中定义了 Orchestration，否则必须至少有一个 config-xxxx 配置文件），除此之外的其他项目为可选项
schemaName: test # schema 名称，每个文件都是单独的 schema，多个 schema 则是多个 yaml 文件，yaml 文件命名要求是 config-xxxx.yaml 格式，虽然没有强制要求，但推荐名称中的 xxxx 与配置的 schemaName 保持一致，方便维护

dataSources: # 配置数据源列表，必须是有效的 jdbc 配置，目前仅支持 MySQL 与 PostgreSQL，另外通过一些未公开（代码中可查，但可能会在未来有变化）的变量，可以配置来兼容其他支持 JDBC 的数据库，但由于没有足够的测试支持，可能会有严重的兼容性问题，配置时候要求至少有一个
```

```
  master_ds_0: # 数据源名称，可以是合法的字符串，目前的校验规则中，没有强制性要求，只要是合法的
yaml 字符串即可，但如果要用于分库分表配置，则需要有有意义的标志 (在分库分表配置中详述)，以下为目
前公开的合法配置项目，不包含内部配置参数
    # 以下参数为必备参数
    url: jdbc:mysql://127.0.0.1:3306/demo_ds_slave_1?serverTimezone=UTC&
useSSL=false # 这里的要求合法的 jdbc 连接串即可，目前尚未兼容 MySQL 8.x，需要在 maven 编译时
候，升级 MySQL JDBC 版本到 5.1.46 或者 47 版本 (不建议升级到 JDBC 的 8.x 系列版本，需要修改
源代码，并且无法通过很多测试 case)
    username: root # MySQL 用户名
    password: password # MySQL 用户的明文密码
    # 以下参数为可选参数，给出示例为默认配置，主要用于连接池控制
    connectionTimeoutMilliseconds: 30000 # 连接超时控制
    idleTimeoutMilliseconds: 60000 # 连接空闲时间设置
    maxLifetimeMilliseconds: 0 # 连接的最大持有时间,0 为无限制
    maxPoolSize: 50 # 连接池中最大维持的连接数量
    minPoolSize: 1 # 连接池的最小连接数量
    maintenanceIntervalMilliseconds: 30000 # 连接维护的时间间隔 atomikos 框架需求
  # 以下配置的假设是,3307 是 3306 的从库,3309,3310 是 3308 的从库
  slave_ds_0:
    url: jdbc:mysql://127.0.0.1:3307/demo_ds_slave_1?serverTimezone=UTC&
useSSL=false
    username: root
    password: password
  master_ds_1:
    url: jdbc:mysql://127.0.0.1:3308/demo_ds_slave_1?serverTimezone=UTC&
useSSL=false
    username: root
    password: password
  slave_ds_1:
    url: jdbc:mysql://127.0.0.1:3309/demo_ds_slave_1?serverTimezone=UTC&
useSSL=false
    username: root
    password: password
  slave_ds_1_slave2:
    url: jdbc:mysql://127.0.0.1:3310/demo_ds_slave_1?serverTimezone=UTC&
useSSL=false
    username: root
    password: password
masterSlaveRule: # 这里配置这个规则的话，相当于是全局读写分离配置
  name: ds_rw # 名称，合法的字符串即可，但如果涉及到在读写分离的基础上设置分库分表，则名称需要有
意义才可以，另外，虽然目前没有强制要求，但主从库配置需要配置在实际关联的主从库上，如果配置的数据源
之间主从是断开的状态，那么可能会发生写入的数据对于只读会话无法读取到的问题
  # 如果一个会话发生了写入并且没有提交 (显式打开事务)， ShardingSphere 在后续的路由中,select
都会在主库执行，直到会话提交
  masterDataSourceName: master_ds_0 # 主库的 DataSource 名称
  slaveDataSourceNames: # 从库的 DataSource 列表，至少需要有一个
    - slave_ds_0
  loadBalanceAlgorithmClassName: io.shardingsphere.api.algorithm.masterslave #
MasterSlaveLoadBalanceAlgorithm 接口的实现类，允许自定义实现 默认提供两个，配置路径为 io.
shardingsphere.api.algorithm.masterslave 下的
RandomMasterSlaveLoadBalanceAlgorithm(随机 Random) 与
RoundRobinMasterSlaveLoadBalanceAlgorithm(轮询: 次数 % 从库数量)
```

```
  loadBalanceAlgorithmType: # 从库负载均衡算法类型，可选值：ROUND_ROBIN，RANDOM。若
loadBalanceAlgorithmClassName 存在则忽略该配置，默认为 ROUND_ROBIN

shardingRule: # sharding 的配置
  # 配置主要分两类，一类是对整个 sharding 规则所有表生效的默认配置，一个是 sharing 具体某张表时
候的配置
  # 首先说默认配置
  masterSlaveRules: # 在 shardingRule 中也可以配置 shardingRule，对分片生效，具体内容与全
局 masterSlaveRule 一致，但语法为：
    master_test_0:
      masterDataSourceName: master_ds_0
      slaveDataSourceNames:
        - slave_ds_0
    master_test_1:
      masterDataSourceName: master_ds_1
      slaveDataSourceNames:
        - slave_ds_1
        - slave_ds_1_slave2
  defaultDataSourceName: master_test_0 # 这里的数据源允许是 dataSources 的配置项目或者
masterSlaveRules 配置的名称，配置为 masterSlaveRule 的话相当于就是配置读写分离了
  broadcastTables: # 广播表 这里配置的表列表，对于发生的所有数据变更，都会不经 sharding 处理，
而是直接发送到所有数据节点，注意此处为列表，每个项目为一个表名称
    - broad_1
    - broad_2
  bindingTables: # 绑定表，也就是实际上哪些配置的 sharding 表规则需要实际生效的列表，配置为
yaml 列表，并且允许单个条目中以逗号切割，所配置表必须已经配置为逻辑表
    - sharding_t1
    - sharding_t2,sharding_t3
  defaultDatabaseShardingStrategy: # 默认库级别 sharding 规则，对应代码中
ShardingStrategy 接口的实现类，目前支持 none,inline,hint,complex,standard 五种配置 注意此
处默认配置仅可以配置五个中的一个
    # 规则配置同样适合表 sharding 配置，同样是在这些算法中选择
    none: # 不配置任何规则,SQL 会被发给所有节点去执行，这个规则没有子项目可以配置
    inline: # 行表达式分片
      shardingColumn: test_id # 分片列名称
      algorithmExpression: master_test_${test_id % 2} # 分片表达式，根据指定的表达式计算
得到需要路由到的数据源名称 需要是合法的 groovy 表达式，示例配置中，取余为 0 则语句路由到
master_test_0，取余为 1 则路由到 master_test_1
    hint: # 基于标记的 sharding 分片
      shardingAlgorithm: # 需要是 HintShardingAlgorithm 接口的实现，目前代码中，仅有为测
试目的实现的 OrderDatabaseHintShardingAlgorithm，没有生产环境可用的实现
    complex: # 支持多列的 sharding，目前无生产可用实现
      shardingColumns: # 逗号切割的列
      shardingAlgorithm: # ComplexKeysShardingAlgorithm 接口的实现类
    standard: # 单列 sharding 算法，需要配合对应的 preciseShardingAlgorithm,
rangeShardingAlgorithm 接口的实现使用，目前无生产可用实现
      shardingColumn: # 列名，允许单列
      preciseShardingAlgorithm: # preciseShardingAlgorithm 接口的实现类
```

```
      rangeShardingAlgorithm: # rangeShardingAlgorithm 接口的实现类
  defaultTableStrategy: # 配置参考 defaultDatabaseShardingStrategy，区别在于,inline 算法的配置中,algorithmExpression 的配置算法结果需要是实际的物理表名称，而非数据源名称
  defaultKeyGenerator: # 默认的主键生成算法，如果没有设置，默认为 SNOWFLAKE 算法
    column: # 自增键对应的列名称
    type: # 自增键的类型，主要用于调用内置的主键生成算法有三个可用值:SNOWFLAKE(时间戳 +worker id+ 自增 id),UUID(java.util.UUID 类生成的随机 UUID),LEAF，其中 Snowflake 算法与 UUID 算法已经实现,LEAF 目前 (2018-01-14) 尚未实现
    className: # 非内置的其他实现了 KeyGenerator 接口的类，需要注意，如果设置这个，就不能设置 type，否则 type 的设置会覆盖 class 的设置
    props:
    # 定制算法需要设置的参数，比如 SNOWFLAKE 算法的 max.tolerate.time.difference.milliseconds
  tables: # 配置表 sharding 的主要位置
    sharding_t1:
      actualDataNodes: master_test_${0..1}.t_order${0..1} # sharding 表对应的数据源以及物理名称，需要用表达式处理，表示表实际上在哪些数据源存在，配置示例中，意思是总共存在 4 个分片 master_test_0.t_order0,master_test_0.t_order1,master_test_1.t_order0,master_test_1.t_order1
      # 需要注意的是，必须保证设置 databaseStrategy 可以路由到唯一的 dataSource，tableStrategy 可以路由到 dataSource 中唯一的物理表上，否则可能导致错误：一个 insert 语句被插入到多个实际物理表中
      databaseStrategy: # 局部设置会覆盖全局设置，参考 defaultDatabaseShardingStrategy
      tableStrategy: # 局部设置会覆盖全局设置，参考 defaultTableStrategy
      keyGenerator: # 局部设置会覆盖全局设置，参考 defaultKeyGenerator
      logicIndex: # 逻辑索引名称 由于 Oracle,PG 这种数据库中，索引与表共用命名空间，如果接受到 drop index 语句，执行之前，会通过这个名称配置的确定对应的实际物理表名称
props:
  sql.show: # 是否开启 SQL 显示，默认值：false
  acceptor.size: # accept 连接的线程数量，默认为 cpu 核数 2 倍
  executor.size: # 工作线程数量最大，默认值：无限制
  max.connections.size.per.query: # 每个查询可以打开的最大连接数量，默认为 1
  proxy.frontend.flush.threshold: # proxy 的服务时候，对于单个大查询，每多少个网络包返回一次
  check.table.metadata.enabled: # 是否在启动时检查分表元数据一致性，默认值：false
  proxy.transaction.type: # 默认 LOCAL,proxy 的事务模型 允许 LOCAL,XA,BASE 三个值 LOCAL 无分布式事务,XA 则是采用 atomikos 实现的分布式事务 BASE 目前尚未实现
  proxy.opentracing.enabled: # 是否启用 opentracing
  proxy.backend.use.nio: # 是否采用 netty 的 NIO 机制连接后端数据库，默认 False ，使用 epoll 机制
  proxy.backend.max.connections: # 使用 NIO 而非 epoll 的话,proxy 后台连接每个 netty 客户端允许的最大连接数量 (注意不是数据库连接限制) 默认为 8
  proxy.backend.connection.timeout.seconds: # 使用 nio 而非 epoll 的话,proxy 后台连接的超时时间，默认 60s
  check.table.metadata.enabled: # 是否在启动时候，检查 sharing 的表的实际元数据是否一致，默认 False

configMap: # 用户自定义配置
```

```
  key1: value1
  key2: value2
  keyx: valuex
```

读写分离

配置项说明

```
dataSources: # 省略数据源配置，与数据分片一致

masterSlaveRule:
  name: # 读写分离数据源名称
  masterDataSourceName: # 主库数据源名称
  slaveDataSourceNames: # 从库数据源名称列表
    - <data_source_name1>
    - <data_source_name2>
    - <data_source_name_x>
  loadBalanceAlgorithmClassName: # 从库负载均衡算法类名称。该类需实现
MasterSlaveLoadBalanceAlgorithm 接口且提供无参数构造器
  loadBalanceAlgorithmType: # 从库负载均衡算法类型，可选值：ROUND_ROBIN，RANDOM。若
`loadBalanceAlgorithmClassName` 存在则忽略该配置

props: # 属性配置
  sql.show: # 是否开启 SQL 显示，默认值：false
  executor.size: # 工作线程数量，默认值：CPU 核数
  check.table.metadata.enabled: # 是否在启动时检查分表元数据一致性，默认值：false

configMap: # 用户自定义配置
  key1: value1
  key2: value2
  keyx: valuex
```

通过 `MasterSlaveDataSourceFactory` 工厂类创建 `DataSource`：

```
DataSource dataSource = MasterSlaveDataSourceFactory.createDataSource(yamlFile);
```

治理

配置项说明

```
dataSources: # 省略数据源配置
shardingRule: # 省略分片规则配置
masterSlaveRule: # 省略读写分离规则配置

orchestration:
```

```
  name: # 数据治理实例名称
  overwrite: # 本地配置是否覆盖注册中心配置。如果可覆盖，每次启动都以本地配置为准
  registry: # 注册中心配置
    serverLists: # 连接注册中心服务器的列表。包括 IP 地址和端口号。多个地址用逗号分隔。如:
host1:2181,host2:2181
    namespace: # 注册中心的命名空间
    digest: # 连接注册中心的权限令牌。缺省为不需要权限验证
    operationTimeoutMilliseconds: # 操作超时的毫秒数，默认 500 毫秒
    maxRetries: # 连接失败后的最大重试次数，默认 3 次
    retryIntervalMilliseconds: # 重试间隔毫秒数，默认 500 毫秒
    timeToLiveSeconds: # 临时节点存活秒数，默认 60 秒
```

ShardingSphere-2.x

数据分片

配置项说明

```
dataSources:
  db0: !!org.apache.commons.dbcp.BasicDataSource
    driverClassName: org.h2.Driver
    url: jdbc:h2:mem:db0;DB_CLOSE_DELAY=-1;DATABASE_TO_UPPER=false;MODE=MYSQL
    username: sa
    password:
    maxActive: 100
  db1: !!org.apache.commons.dbcp.BasicDataSource
    driverClassName: org.h2.Driver
    url: jdbc:h2:mem:db1;DB_CLOSE_DELAY=-1;DATABASE_TO_UPPER=false;MODE=MYSQL
    username: sa
    password:
    maxActive: 100

shardingRule:
  tables:
    config:
      actualDataNodes: db${0..1}.t_config
    t_order:
      actualDataNodes: db${0..1}.t_order_${0..1}
      databaseStrategy:
        standard:
          shardingColumn: user_id
          preciseAlgorithmClassName: io.shardingjdbc.core.yaml.fixture.
SingleAlgorithm
      tableStrategy:
        inline:
          shardingColumn: order_id
```

```
          algorithmInlineExpression: t_order_${order_id % 2}
      keyGeneratorColumnName: order_id
      keyGeneratorClass: io.shardingjdbc.core.yaml.fixture.IncrementKeyGenerator
    t_order_item:
      actualDataNodes: db${0..1}.t_order_item_${0..1}
      # 绑定表中其余的表的策略与第一张表的策略相同
      databaseStrategy:
        standard:
          shardingColumn: user_id
          preciseAlgorithmClassName: io.shardingjdbc.core.yaml.fixture.
SingleAlgorithm
      tableStrategy:
        inline:
          shardingColumn: order_id
          algorithmInlineExpression: t_order_item_${order_id % 2}
  bindingTables:
    - t_order,t_order_item
  # 默认数据库分片策略
  defaultDatabaseStrategy:
    none:
  defaultTableStrategy:
    complex:
      shardingColumns: id, order_id
      algorithmClassName: io.shardingjdbc.core.yaml.fixture.MultiAlgorithm
  props:
    sql.show: true
```

读写分离

概念

为了缓解数据库压力，将写入和读取操作分离为不同数据源，写库称为主库，读库称为从库，一主库可配置多从库。

支持项

1. 提供了一主多从的读写分离配置，可独立使用，也可配合分库分表使用。
2. 独立使用读写分离支持 SQL 透传。
3. 同一线程且同一数据库连接内，如有写入操作，以后的读操作均从主库读取，用于保证数据一致性。
4. Spring 命名空间。
5. 基于 Hint 的强制主库路由。

不支持范围

1. 主库和从库的数据同步。
2. 主库和从库的数据同步延迟导致的数据不一致。
3. 主库双写或多写。

配置规则

```
dataSources:
  db_master: !!org.apache.commons.dbcp.BasicDataSource
    driverClassName: org.h2.Driver
    url: jdbc:h2:mem:db_master;DB_CLOSE_DELAY=-1;DATABASE_TO_UPPER=false;MODE=MYSQL
    username: sa
    password:
    maxActive: 100
  db_slave_0: !!org.apache.commons.dbcp.BasicDataSource
    driverClassName: org.h2.Driver
    url: jdbc:h2:mem:db_slave_0;DB_CLOSE_DELAY=-1;DATABASE_TO_UPPER=false;
MODE=MYSQL
    username: sa
    password:
    maxActive: 100
  db_slave_1: !!org.apache.commons.dbcp.BasicDataSource
      driverClassName: org.h2.Driver
      url: jdbc:h2:mem:db_slave_1;DB_CLOSE_DELAY=-1;DATABASE_TO_UPPER=false;
MODE=MYSQL
      username: sa
      password:
      maxActive: 100

masterSlaveRule:
  name: db_ms
  masterDataSourceName: db_master
  slaveDataSourceNames: [db_slave_0, db_slave_1]
  configMap:
    key1: value1
```

通过 `MasterSlaveDataSourceFactory` 工厂类创建 `DataSource`：

```
DataSource dataSource = MasterSlaveDataSourceFactory.createDataSource(yamlFile);
```

治理

配置项说明

Zookeeper 分库分表编排配置项说明

```
dataSources: # 数据源配置

shardingRule: # 分片规则配置

orchestration: # Zookeeper 编排配置
  name: # 编排服务节点名称
  overwrite: # 本地配置是否可覆盖注册中心配置。如果可覆盖，每次启动都以本地配置为准
  zookeeper: # Zookeeper 注册中心配置
    namespace: # Zookeeper 的命名空间
    serverLists: # 连接 Zookeeper 服务器的列表。包括 IP 地址和端口号。多个地址用逗号分隔。如: host1:2181,host2:2181
    baseSleepTimeMilliseconds: # 等待重试的间隔时间的初始值。单位：毫秒
    maxSleepTimeMilliseconds: # 等待重试的间隔时间的最大值。单位：毫秒
    maxRetries: # 最大重试次数
    sessionTimeoutMilliseconds: # 会话超时时间。单位：毫秒
    connectionTimeoutMilliseconds: # 连接超时时间。单位：毫秒
    digest: # 连接 Zookeeper 的权限令牌。缺省为不需要权限验证
```

Etcd 分库分表编排配置项说明

```
dataSources: # 数据源配置

shardingRule: # 分片规则配置

orchestration: # Etcd 编排配置
  name: # 编排服务节点名称
  overwrite: # 本地配置是否可覆盖注册中心配置。如果可覆盖，每次启动都以本地配置为准
  etcd: # Etcd 注册中心配置
    serverLists: # 连接 Etcd 服务器的列表。包括 IP 地址和端口号。多个地址用逗号分隔。如: http://host1:2379,http://host2:2379
    timeToLiveSeconds: # 临时节点存活时间。单位：秒
    timeoutMilliseconds: # 每次请求的超时时间。单位：毫秒
    maxRetries: # 每次请求的最大重试次数
    retryIntervalMilliseconds: # 重试间隔时间。单位：毫秒
```

分库分表编排数据源构建方式

```
DataSource dataSource = OrchestrationShardingDataSourceFactory.
createDataSource(yamlFile);
```

读写分离数据源构建方式

```
DataSource dataSource = OrchestrationMasterSlaveDataSourceFactory.
createDataSource(yamlFile);
```

Java API

5.0.0-beta

数据分片

配置入口

类名称：org.apache.shardingsphere.sharding.api.config.ShardingRuleConfiguration

可配置属性：

分片表配置

类名称：org.apache.shardingsphere.sharding.api.config.ShardingTableRuleConfiguration

可配置属性：

名称	数据类型	说明	默认值
logi cTable	String	分片逻辑表名称	•
act ualDat aNodes (?)	String	由数据源名 + 表名组成，以小数点分隔。多个表以逗号分隔，支持行表达式	使用已知数据源与逻辑表名称生成数据节点，用于广播表或只分库不分表且所有库的表结构完全一致的情况
databa seShar dingSt rategy (?)	S harding Strateg yConfig uration	分库策略	使用默认分库策略
tab leShar dingSt rategy (?)	S harding Strateg yConfig uration	分表策略	使用默认分表策略
k eyGene rateSt rategy (?)	KeyG enerato rConfig uration	自增列生成器	使用默认自增主键生成器

自动分片表配置

类名称：org.apache.shardingsphere.sharding.api.config.ShardingAutoTableRuleConfiguration

可配置属性：

名称	数据类型	说明	默认值
logicTable	String	分片逻辑表名称	•
act ualDataSources (?)	String	数据源名称，多个数据源以逗号分隔	使用全部配置的数据源
sh ardingStrategy (?)	S hardingStrateg yConfiguration	分片策略	使用默认分片策略
keyGe nerateStrategy (?)	KeyGenerato rConfiguration	自增列生成器	使用默认自增主键生成器

分片策略配置

标准分片策略配置

类名称：org.apache.shardingsphere.sharding.api.config.strategy.sharding.StandardShardingStrategyConfiguration

可配置属性：

名称	数据类型	说明
shardingColumn	String	分片列名称
shardingAlgorithmName	String	分片算法名称

复合分片策略配置

类名称：org.apache.shardingsphere.sharding.api.config.strategy.sharding.ComplexShardingStrategyConfiguration

可配置属性：

名称	数据类型	说明
shardingColumns	String	分片列名称，多个列以逗号分隔
shardingAlgorithmName	String	分片算法名称

Hint 分片策略配置

类名称:org.apache.shardingsphere.sharding.api.config.strategy.sharding.HintShardingStrategyConfiguration

可配置属性:

名称	数据类型	说明
shardingAlgorithmName	String	分片算法名称

不分片策略配置

类名称:org.apache.shardingsphere.sharding.api.config.strategy.sharding.NoneShardingStrategyConfiguration

可配置属性：无

算法类型的详情，请参见 内置分片算法列表。

分布式序列策略配置

类名称:org.apache.shardingsphere.sharding.api.config.strategy.keygen.KeyGenerateStrategyConfiguration

可配置属性:

名称	数据类型	说明
column	String	分布式序列列名称
keyGeneratorName	String	分布式序列算法名称

算法类型的详情，请参见 内置分布式序列算法列表。

读写分离

配置入口

类名称：ReadwriteSplittingRuleConfiguration

可配置属性:

名称	数据类型	说明
d ataSources (+)	Collection<Read writeSplittingDataSourceRuleConfiguration>	读写数据源配置
loa dBalancers (*)	Map<String, ShardingSphereAlgorithmConfiguration>	从库负载均衡算法配置

读写分离数据源配置

类名称：ReadwriteSplittingDataSourceRuleConfiguration

可配置属性：

名称	数据类型	说明	默认值
name	String	读写分离数据源名称	•
write DataSource-Name	String	写库数据源名称	•
readD ataSource-Names (+)	Coll ection<String>	读库数据源名称列表	•
lo adBalancerName (?)	String	读库负载均衡算法名称	轮询负载均衡算法

算法类型的详情，请参见 内置负载均衡算法列表。

数据加密

配置入口

类名称：org.apache.shardingsphere.encrypt.api.config.EncryptRuleConfiguration

可配置属性：

加密表规则配置

类名称：org.apache.shardingsphere.encrypt.api.config.rule.EncryptTableRuleConfiguration

可配置属性：

名称	数据类型	说明
name	String	表名称
columns (+)	Collection<EncryptColumnRuleConfiguration>	加密列规则配置列表

加密列规则配置

类名称：org.apache.shardingsphere.encrypt.api.config.rule.EncryptColumnRuleConfiguration

可配置属性：

名称	数据类型	说明
logicColumn	String	逻辑列名称
cipherColumn	String	密文列名称
assistedQueryColumn (?)	String	查询辅助列名称
plainColumn (?)	String	原文列名称
encryptorName	String	加密算法名称

加解密算法配置

类名称：org.apache.shardingsphere.infra.config.algorithm.ShardingSphereAlgorithmConfiguration

可配置属性：

名称	数据类型	说明
name	String	加解密算法名称
type	String	加解密算法类型
properties	Properties	加解密算法属性配置

算法类型的详情，请参见 内置加密算法列表。

影子库

配置入口

类名称：org.apache.shardingsphere.shadow.api.config.ShadowRuleConfiguration

可配置属性：

名称	数据类型	说明
column	String	SQL 中的影子字段名，该值为 true 的 SQL 会路由到影子库执行
sourceDataSourceNames	List<String>	生产数据库名称
shadowDataSourceNames	List<String>	影子数据库名称，与上面一一对应

分布式治理

配置项说明

治理

配置入口

类名称：org.apache.shardingsphere.governance.repository.api.config.GovernanceConfiguration

可配置属性：

名称	数据类型	说明
name	String	注册中心实例名称
registr yCenterConfiguration	Registr yCenterConfiguration	注册中心实例的配置
overwrite	boolean	本地配置是否覆盖配置中心配置，如果可覆盖，每次启动都以本地配置为准

注册中心的类型可以为 Zookeeper 或 etcd。

治理实例配置

类名称：org.apache.shardingsphere.governance.repository.api.config.RegistryCenterConfiguration

可配置属性：

名称	数据类型	说明
type	String	治理实例类型，如：Zookeeper, etcd
serverLists	String	治理服务列表，包括 IP 地址和端口号，多个地址用逗号分隔，如: host1:2181,host2:2181
props	Properties	配置本实例需要的其他参数，例如 ZooKeeper 的连接参数等
overwrite	boolean	本地配置是否覆盖配置中心配置；如果覆盖，则每次启动都参考本地配置

ZooKeeper 属性配置

名称	数据类型	说明	默认值
digest (?)	String	连接注册中心的权限令牌	无需验证
operation TimeoutMilliseconds (?)	int	操作超时的毫秒数	500 毫秒
maxRetries (?)	int	连接失败后的最大重试次数	3 次
retryI ntervalMilliseconds (?)	int	重试间隔毫秒数	500 毫秒
timeToLiveSeconds (?)	int	临时节点存活秒数	60 秒

Etcd 属性配置

名称	数据类型	说明	默认值
timeToLiveSeconds (?)	long	数据存活秒数	30 秒

ShardingSphere-4.x

数据分片

ShardingDataSourceFactory

名称	数据类型	说明
dataSourceMap	Map<String, DataSource>	数据源配置
shardingRuleConfig	ShardingRuleConfiguration	数据分片规则配置
props (?)	Properties	属性配置

ShardingRuleConfiguration

名称	数据类型	说明
tableRuleConfigs	Collection	分片规则列表
bindingTableGroups (?)	Collection	绑定表规则列表
broadcastTables (?)	Collection	广播表规则列表
defaultDataSourceName (?)	String	未配置分片规则的表将根据默认数据源定位
defaultDatabaseShardingStrategyConfig (?)	ShardingStrategyConfiguration	默认数据库分片策略
defaultTableShardingStrategyConfig (?)	ShardingStrategyConfiguration	默认分表策略
defaultKeyGeneratorConfig (?)	KeyGeneratorConfiguration	默认密钥生成器配置，使用用户定义的或内置的，例如雪花/UUID。默认密钥生成器是 `org.apache.shardingsphere.core.keygen.generator.impl.SnowflakeKeyGenerator`
masterSlaveRuleConfigs (?)	Collection	读写分离规则，默认值表示不使用读写分离

TableRuleConfiguration

名称	数据类型	说明
logicTable	String	逻辑表名称
actualDataNodes (?)	String	描述数据源名称和实际表，分隔符为点，多个数据节点用逗号分割，支持内联表达式。不存在意味着仅分片数据库。示例：ds: math:*{0..7}.tbl*{0..7}
databaseShardingStrategyConfig (?)	ShardingStrategyConfiguration	数据库分片策略，如果不存在则使用默认的数据库分片策略
tableShardingStrategyConfig (?)	ShardingStrategyConfiguration	表分片策略，如果不存在则使用默认的数据库分片策略
keyGeneratorConfig (?)	KeyGeneratorConfiguration	主键生成器配置，如果不存在则使用默认主键生成器
encryptorConfiguration (?)	EncryptorConfiguration	加密生成器配置

StandardShardingStrategyConfiguration

ShardingStrategyConfiguration 的实现类

名称	数据类型	说明
shardingColumn	String	分片键
preciseShardingAlgorithm	PreciseShardingAlgorithm	= 和 `IN` 中使用的精确分片算法
rangeShardingAlgorithm (?)	RangeShardingAlgorithm	`BETWEEN` 中使用的范围分片算法

ComplexShardingStrategyConfiguration

`ShardingStrategyConfiguration` 的实现类，用于具有多个分片键的复杂分片情况。

名称	数据类型	说明
shardingColumns	String	分片键，以逗号分隔
shardingAlgorithm	ComplexKeysShardingAlgorithm	复杂分片算法

InlineShardingStrategyConfiguration

`ShardingStrategyConfiguration` 的实现类，用于行表达式的分片策略。

名称	数据类型	说明
shardingColumns	String	分片列名，以逗号分隔
algorithmExpression	String	行表达式的分片策略，应符合 groovy 语法；有关更多详细信息，请参阅行表达式

HintShardingStrategyConfiguration

`ShardingStrategyConfiguration` 的实现类，用于配置强制分片策略。

名称	数据类型	说明
shardingAlgorithm	HintShardingAlgorithm	强制分片算法

NoneShardingStrategyConfiguration

`ShardingStrategyConfiguration` 的实现类，用于配置非分片策略。

自增主键生成器

名称	数据类型	说明
column	String	主键
type	String	主键生成器的类型，使用用户定义的或内置的，例如雪花，UUID
props	Properties	主键生成器的属性配置

属性配置

属性配置项，可以是以下属性。

SNOWFLAKE

名称	数据类型	说明
max.tolerate.time.difference.milliseconds (?)	long	最大容忍时钟回退时间，单位：毫秒。默认为 10 毫秒
max.vibration.offset (?)	int	最大抖动上限值，范围 [0, 4096)，默认为 1。注：若使用此算法生成值作分片值，建议配置此属性。此算法在不同毫秒内所生成的 key 取模 2^n (2^n 一般为分库或分表数) 之后结果总为 0 或 1。为防止上述分片问题，建议将此属性值配置为 (2^n)-1

读写分离

MasterSlaveDataSourceFactory

名称	数据类型	说明
dataSourceMap	Map<String, DataSource>	数据源及其名称的映射
masterSlaveRuleConfig	MasterSlaveRuleConfiguration	读写分离规则配置
props (?)	Properties	属性配置

MasterSlaveRuleConfiguration

名称	数据类型	说明
name	String	读写分离数据源名称
masterDataSourceName	String	主数据库源名称
slaveDataSourceNames	Collection	从数据库源名称列表
loadBalanceAlgorithm (?)	MasterSlav eLoadBalanceAlgorithm	从库负载均衡算法

属性配置

属性配置项，可以是以下属性。

名称	数据类型	说明
sql.show (?)	boolean	是否打印 SQL 日志，默认值：false
executor.size (?)	int	用于 SQL 实现的工作线程号；如果为 0，则没有限制。默认值：0
max.connections.size.per.query (?)	int	每个物理数据库每次查询分配的最大连接数，默认值：1
check.table.metadata.enabled (?)	boolean	初始化时是否检查元数据的一致性，默认值：false

数据脱敏

EncryptDataSourceFactory

名称	数据类型	说明
dataSource	DataSource	数据源
encryptRuleConfig	EncryptRuleConfiguration	加密规则配置
props (?)	Properties	属性配置

EncryptRuleConfiguration

名称	数据类型	说明
encryptors	Map<String, EncryptorRuleConfiguration>	加密器名称和加密器
tables	Map<String, EncryptTableRuleConfiguration>	加密表名和加密表

属性配置

属性配置项，可以是以下属性。

名称	数据类型	说明
sql.show (?)	boolean	是否打印 SQL 日志，默认值：false
query.with.cipher.column (?)	boolean	有普通列时，是否使用加密列查询，默认值：true

编排

OrchestrationShardingDataSourceFactory

名称	数据类型	说明
dataSourceMap	Map<String, DataSource>	与 `S hardingDataSourceFactory` 相同
sha rdingRuleConfig	Shar dingRuleConfiguration	与 `S hardingDataSourceFactory` 相同
props (?)	Properties	与 `S hardingDataSourceFactory` 相同
orch estrationConfig	Orche strationConfiguration	编排规则配置

OrchestrationMasterSlaveDataSourceFactory

名称	数据类型	说明
dataSourceMap	Map<String, DataSource>	与 `Mas terSlaveDataSourceFactory` 相同
master SlaveRuleConfig	MasterS laveRuleConfiguration	与 `Mas terSlaveDataSourceFactory` 相同
configMap (?)	Map<String, Object>	与 `Mas terSlaveDataSourceFactory` 相同
props (?)	Properties	与 `Mas terSlaveDataSourceFactory` 相同
orch estrationConfig	Orche strationConfiguration	编排规则配置

OrchestrationEncryptDataSourceFactory

名称	数据类型	说明
dataSource	DataSource	与 `Encryp tDataSourceFactory` 相同
encryptRuleConfig	Encr yptRuleConfiguration	与 `Encryp tDataSourceFactory` 相同
props (?)	Properties	与 `Encryp tDataSourceFactory` 相同
orchestrationConfig	Orches trationConfiguration	编排规则配置

OrchestrationConfiguration

名称	数据类型	说明
instanceC onfigurationMap	Map<String, CenterConfiguration>	co nfig-center&registry-center 的配置，key 是 center 的名称，value 是 co nfig-center/registry-center

CenterConfiguration

名称	数据类型	说明
type	String	注册中心类型 (z ookeeper/etcd/apollo/nacos)
properties	String	注册中心的配置属性，例如 zookeeper 的配置属性
orchestrationType	String	编排中心的类型：config-center 或 registry-center，如果两者都使用 `setOrchestrationType("registry_center,config_center");`
serverLists	String	注册中心服务列表，包括 IP 地址和端口号，多个地址用逗号分隔，如：host1:2181,host2:2181
namespace (?)	String	命名空间

属性配置

属性配置项，可以是以下属性。

名称	数据类型	说明
overwrite	boolean	本地配置是否覆盖配置中心配置；如果覆盖，则每次启动都参考本地配置

如果注册中心类型是 `zookeeper`，则可以使用以下选项设置属性：

名称	数据类型	说明
digest (?)	String	连接注册中心的权限令牌；默认表示不需要权限
operationTimeoutMilliseconds (?)	int	操作超时毫秒数，默认为 500 毫秒
maxRetries (?)	int	最大重试次数，默认为 3 次
retryIntervalMilliseconds (?)	int	重试间隔毫秒数，默认为 500 毫秒
timeToLiveSeconds (?)	int	临时节点的存活时间，默认 60 秒

如果注册中心类型是 `etcd`，则可以使用以下选项设置属性：

名称	数据类型	说明
timeToLiveSeconds (?)	long	etcd TTL 秒，默认为 30 秒

如果注册中心类型是 `apollo`，则可以使用以下选项设置属性：

名称	数据类型	说明
appId (?)	String	Apollo appId，默认为 `APOLLO_SHADINGSPHERE`
env (?)	String	Apollo env，默认为 `DEV`
clusterName (?)	String	Apollo clusterName，默认为 `default`
administrator (?)	String	Apollo administrator，默认为“| | token (?) String Apollo token，默认为 “
portalUrl (?)	String	Apollo portalUrl，默认为 “
connectTimeout (?)	int	Apollo connectTimeout，默认为 1000 毫秒
readTimeout (?)	int	Apollo readTimeout，默认为 5000 毫秒

如果注册中心类型是 `nacos`，则可以使用以下选项设置属性：

名称	数据类型	说明
group (?)	String	Nacos 组，默认为 `SHADING_SPHERE_DEFAULT_GROUP`
timeout (?)	long	Nacos 超时时间，默认为 3000 毫秒

ShardingSphere-3.x

数据分片

ShardingDataSourceFactory

名称	数据类型	说明
dataSourceMap	Map<String, DataSource>	数据源配置
shardingRuleConfig	ShardingRuleConfiguration	数据分片规则配置
configMap (?)	Map<String, Object>	用户自定义的参数
props (?)	Properties	属性配置

ShardingRuleConfiguration

名称	数据类型	说明
tableRuleConfigs	Collection	分片规则列表
bindingTableGroups (?)	Collection	绑定表规则列表
broadcastTables (?)	Collection	广播表规则列表
defaultDataSourceName (?)	String	未配置分片规则的表将根据默认数据源定位
defaultDatabaseShardingStrategyConfig (?)	ShardingStrategyConfiguration	默认数据库分片策略
defaultTableShardingStrategyConfig (?)	ShardingStrategyConfiguration	默认分表策略
def aultKeyGeneratorConfig (?)	KeyG eneratorConfiguration	默认密钥生成器配置，使用用户定义的或内置的，例如雪花/UUID。默认密钥生成器是 “ org.apache.sharding sphere.core.keygen.generator.impl.Snow flakeKeyGenerator“
masterSlaveRuleConfigs (?)	Collection	读写分离规则，默认值表示不使用读写分离

TableRuleConfiguration

名称	数据类型	说明
logicTable	String	逻辑表名称
actualDataNodes (?)	String	描述数据源名称和实际表，分隔符为点，多个数据节点用逗号分割，支持内联表达式。不存在意味着仅分片数据库。示例：ds0..7.*tbl*{0..7}
databaseShardingStrategyConfig (?)	ShardingStrategyConfiguration	数据库分片策略，如果不存在则使用默认的数据库分片策略
tableShardingStrategyConfig (?)	ShardingStrategyConfiguration	表分片策略，如果不存在则使用默认的数据库分片策略
logicIndex (?)	String	逻辑索引，如果在 Oracle/PostgreSQL 中使用 DROP INDEX XXX SQL，则需要设置此属性以查找实际表
keyGeneratorConfig (?)	String	主键列配置，如果不存在则使用默认主键列
keyGenerator (?)	KeyGenerator	主键生成器配置，如果不存在则使用默认主键生成器

StandardShardingStrategyConfiguration

ShardingStrategyConfiguration 的实现类

名称	数据类型	说明
shardingColumn	String	分片键
prec iseShardingAlgorithm	Prec iseShardingAlgorithm	= 和 `IN` 中使用的精确分片算法
ra ngeShardingAlgorithm (?)	Ra ngeShardingAlgorithm	`BETWEEN` 中使用的范围分片算法

ComplexShardingStrategyConfiguration

`ShardingStrategyConfiguration` 的实现类，用于具有多个分片键的复杂分片策略。

名称	数据类型	说明
shardingColumns	String	分片键，以逗号分隔
shardingAlgorithm	ComplexKeysShardingAlgorithm	复杂分片算法

InlineShardingStrategyConfiguration

`ShardingStrategyConfiguration` 的实现类，用于行表达式的分片策略。

名称	数据类型	说明
shardingColumns	String	分片列名，以逗号分隔
algorithmExpression	String	行表达式的分片策略，应符合 groovy 语法；有关更多详细信息，请参阅行表达式

HintShardingStrategyConfiguration

`ShardingStrategyConfiguration` 的实现类，用于配置强制分片策略。

名称	数据类型	说明
shardingAlgorithm	HintShardingAlgorithm	强制分片算法

NoneShardingStrategyConfiguration

`ShardingStrategyConfiguration` 的实现类，用于配置非分片策略。

属性配置

枚举属性

名称	数据类型	说明
sql.show (?)	boolean	是否打印 SQL 日志，默认值：false
executor.size (?)	int	用于 SQL 实现的工作线程号；如果为 0，则没有限制。默认值：0
max.connections.size.per.query (?)	int	每个物理数据库每次查询分配的最大连接数，默认值：1
check.table.metadata.enabled (?)	boolean	初始化时是否检查元数据的一致性，默认值：false

configMap

用户定义的参数。

读写分离

MasterSlaveDataSourceFactory

名称	数据类型	说明
dataSourceMap	Map<String, DataSource>	数据源及其名称的映射
masterSlaveRuleConfig	MasterSlaveRuleConfiguration	读写分离规则配置
configMap (?)	Map<String, Object>	用户自定义的参数
props (?)	Properties	属性配置

MasterSlaveRuleConfiguration

名称	数据类型	说明
name	String	读写分离数据源名称
masterDataSourceName	String	主数据库源名称
slaveDataSourceNames	Collection	从数据库源名称列表
loadBalanceAlgorithm (?)	MasterSlav eLoadBalanceAlgorithm	从库负载均衡算法

configMap

用户定义的参数。

PropertiesConstant

枚举属性。

名称	数据类型	说明
sql.show (?)	boolean	是否打印 SQL 日志，默认值：false
executor.size (?)	int	用于 SQL 实现的工作线程号；如果为 0，则没有限制。默认值：0
max.c onnections.size.per.query (?)	int	每个物理数据库每次查询分配的最大连接数，默认值：1
che ck.table.metadata.enabled (?)	boolean	初始化时是否检查元数据的一致性，默认值：false

编排

OrchestrationShardingDataSourceFactory

名称	数据类型	说明
dataSourceMap	Map<String, DataSource>	ShardingDataSourceFactory“ 相同
sha rdingRuleConfig	Shar dingRuleConfiguration	ShardingDataSourceFactory“ 相同
configMap (?)	Map<String, Object>	ShardingDataSourceFactory“ 相同
props (?)	Properties	ShardingDataSourceFactory“ 相同
orch estrationConfig	Orche strationConfiguration	编排规则配置

OrchestrationMasterSlaveDataSourceFactory

名称	数据类型	说明
dataSourceMap	Map<String, DataSource>	与 `Mas terSlaveDataSourceFactory` 相同
master SlaveRuleConfig	MasterS laveRuleConfiguration	与 `Mas terSlaveDataSourceFactory` 相同
configMap (?)	Map<String, Object>	与 `Mas terSlaveDataSourceFactory` 相同
props (?)	Properties	与 `Mas terSlaveDataSourceFactory` 相同
orch estrationConfig	Orche strationConfiguration	编排规则配置

OrchestrationConfiguration

名称	数据类型	说明
name	String	编排实例名称
overwrite	boolean	本地配置是否覆盖配置中心配置；如果覆盖，则每次启动都参考本地配置
regCenterConfig	Regis tryCenterConfiguration	注册中心配置

RegistryCenterConfiguration

名称	数据类型	说明
serverLists	String	注册中心服务列表，包括 IP 地址和端口号，多个地址用逗号分隔，如: host1:2181,host2:2181
namespace (?)	String	命名空间
digest (?)	String	连接注册中心的权限令牌；默认表示不需要权限
operationTimeoutMilliseconds (?)	int	操作超时毫秒数，默认为 500 毫秒
maxRetries (?)	int	最大重试次数，默认为 3 次
retryIntervalMilliseconds (?)	int	重试间隔毫秒数，默认为 500 毫秒
timeToLiveSeconds (?)	int	临时节点的存活时间，默认 60 秒

ShardingSphere-2.x

读写分离

概念

为了缓解数据库压力，将写入和读取操作分离为不同数据源，写库称为主库，读库称为从库，一主库可配置多从库。

支持项

1. 提供了一主多从的读写分离配置，可独立使用，也可配合分库分表使用。
2. 独立使用读写分离支持 SQL 透传。
3. 同一线程且同一数据库连接内，如有写入操作，以后的读操作均从主库读取，用于保证数据一致性。
4. Spring 命名空间。
5. 基于 Hint 的强制主库路由。

不支持项

1. 主库和从库的数据同步。
2. 主库和从库的数据同步延迟导致的数据不一致。
3. 主库双写或多写。

代码开发示例

读写分离

```
// 构造一个读写分离数据源，读写分离数据源实现了 DataSource 接口，可以直接作为数据源进行处理。masterDataSource、slaveDataSource0、slaveDataSource1 等都是使用 DBCP 等连接池配置的真实数据源。
Map<String, DataSource> dataSourceMap = new HashMap<>();
dataSourceMap.put("masterDataSource", masterDataSource);
dataSourceMap.put("slaveDataSource0", slaveDataSource0);
dataSourceMap.put("slaveDataSource1", slaveDataSource1);

// 构建读写分离配置
MasterSlaveRuleConfiguration masterSlaveRuleConfig = new
MasterSlaveRuleConfiguration();
masterSlaveRuleConfig.setName("ms_ds");
masterSlaveRuleConfig.setMasterDataSourceName("masterDataSource");
masterSlaveRuleConfig.getSlaveDataSourceNames().add("slaveDataSource0");
masterSlaveRuleConfig.getSlaveDataSourceNames().add("slaveDataSource1");

DataSource dataSource = MasterSlaveDataSourceFactory.
createDataSource(dataSourceMap, masterSlaveRuleConfig);
```

分库分表 + 读写分离

```
// 构造一个读写分离数据源，读写分离数据源实现了 DataSource 接口，可以直接作为数据源进行处理。masterDataSource、slaveDataSource0、slaveDataSource1 等都是使用 DBCP 等连接池配置的真实数据源。
Map<String, DataSource> dataSourceMap = new HashMap<>();
dataSourceMap.put("masterDataSource0", masterDataSource0);
dataSourceMap.put("slaveDataSource00", slaveDataSource00);
dataSourceMap.put("slaveDataSource01", slaveDataSource01);

dataSourceMap.put("masterDataSource1", masterDataSource1);
dataSourceMap.put("slaveDataSource10", slaveDataSource10);
dataSourceMap.put("slaveDataSource11", slaveDataSource11);

// 构建读写分离配置
MasterSlaveRuleConfiguration masterSlaveRuleConfig0 = new
MasterSlaveRuleConfiguration();
masterSlaveRuleConfig0.setName("ds_0");
masterSlaveRuleConfig0.setMasterDataSourceName("masterDataSource0");
masterSlaveRuleConfig0.getSlaveDataSourceNames().add("slaveDataSource00");
masterSlaveRuleConfig0.getSlaveDataSourceNames().add("slaveDataSource01");

MasterSlaveRuleConfiguration masterSlaveRuleConfig1 = new
MasterSlaveRuleConfiguration();
```

```
masterSlaveRuleConfig1.setName("ds_1");
masterSlaveRuleConfig1.setMasterDataSourceName("masterDataSource1");
masterSlaveRuleConfig1.getSlaveDataSourceNames().add("slaveDataSource10");
masterSlaveRuleConfig1.getSlaveDataSourceNames().add("slaveDataSource11");

// 继续通过 ShardingSlaveDataSourceFactory 创建 ShardingDataSource
ShardingRuleConfiguration shardingRuleConfig = new ShardingRuleConfiguration();
shardingRuleConfig.getMasterSlaveRuleConfigs().add(masterSlaveRuleConfig0);
shardingRuleConfig.getMasterSlaveRuleConfigs().add(masterSlaveRuleConfig1);

DataSource dataSource = ShardingDataSourceFactory.createDataSource(dataSourceMap,
shardingRuleConfig);
```

ShardingSphere-1.x

读写分离

概念

为了缓解数据库压力，将写入和读取操作分离为不同数据源，写库称为主库，读库称为从库，一主库可配置多从库。

支持项

1. 提供了一主多从的读写分离配置，可独立使用，也可配合分库分表使用。
2. 同一线程且同一数据库连接内，如有写入操作，以后的读操作均从主库读取，用于保证数据一致性。
3. Spring 命名空间。
4. 基于 Hint 的强制主库路由。

不支持项

1. 主库和从库的数据同步。
2. 主库和从库的数据同步延迟导致的数据不一致。
3. 主库双写或多写。

代码开发示例

```
// 构造一个读写分离数据源，读写分离数据源实现了 DataSource 接口，可以直接作为数据源进行处理。masterDataSource、slaveDataSource0、slaveDataSource1 等都是使用 DBCP 等连接池配置的真实数据源。
Map<String, DataSource> slaveDataSourceMap0 = new HashMap<>();
slaveDataSourceMap0.put("slaveDataSource00", slaveDataSource00);
slaveDataSourceMap0.put("slaveDataSource01", slaveDataSource01);
// You can choose the master-slave library load balancing strategy, the default is ROUND_ROBIN, and there is RANDOM to choose from, or customize the load strategy
DataSource masterSlaveDs0 = MasterSlaveDataSourceFactory.createDataSource("ms_0", "masterDataSource0", masterDataSource0, slaveDataSourceMap0, MasterSlaveLoadBalanceStrategyType.ROUND_ROBIN);

Map<String, DataSource> slaveDataSourceMap1 = new HashMap<>();
slaveDataSourceMap1.put("slaveDataSource10", slaveDataSource10);
slaveDataSourceMap1.put("slaveDataSource11", slaveDataSource11);
DataSource masterSlaveDs1 = MasterSlaveDataSourceFactory.createDataSource("ms_1", "masterDataSource1", masterDataSource1, slaveDataSourceMap1, MasterSlaveLoadBalanceStrategyType.ROUND_ROBIN);

// 构建读写分离配置
Map<String, DataSource> dataSourceMap = new HashMap<>();
dataSourceMap.put("ms_0", masterSlaveDs0);
dataSourceMap.put("ms_1", masterSlaveDs1);
```

Spring 命名空间配置

ShardingSphere-5.0.0-beta

数据分片

配置项说明

命名空间：http://shardingsphere.apache.org/schema/shardingsphere/sharding/sharding-5.0.0.xsd

<sharding:rule />

名称	类型	说明
id	属性	Spring Bean Id
table-rules (?)	标签	分片表规则配置
auto-table-rules (?)	标签	自动化分片表规则配置
binding-table-rules (?)	标签	绑定表规则配置
broadcast-table-rules (?)	标签	广播表规则配置
default-database-strategy-ref (?)	属性	默认分库策略名称
default-table-strategy-ref (?)	属性	默认分表策略名称
default-key-generate-strategy-ref (?)	属性	默认分布式序列策略名称

<sharding:table-rules />

名称	类型	说明
table-rule (+)	标签	分片表规则配置

<sharding:table-rule />

名称	• 类型 *	说明
logic-table	属性	逻辑表名称
actual-data-nodes	属性	由数据源名 + 表名组成，以小数点分隔。多个表以逗号分隔，支持 inline 表达式。缺省表示使用已知数据源与逻辑表名称生成数据节点，用于广播表（即每个库中都需要一个同样的表用于关联查询，多为字典表）或只分库不分表且所有库的表结构完全一致的情况
database-strategy-ref	属性	标准分片表分库策略名称
table-strategy-ref	属性	标准分片表分表策略名称
key-generate-strategy-ref	属性	分布式序列策略名称

<auto-table-rules/>

名称	类型	说明
auto-table-rule (+)	标签	自动化分片表规则配置

<auto-table-rule/> | 名称 | 类型 | 说明 | | ————————- | ——| ————| | logic-table | 属性 | 逻辑表名称 | | actual-data-sources | 属性 | 自动分片表数据源名 | | sharding-strategy-ref | 属性 | 自动分片表策略名称 | | key-generate-strategy-ref | 属性 | 分布式序列策略名称 |

<sharding:binding-table-rules />

名称	类型	说明
binding-table-rule (+)	标签	绑定表规则配置

<sharding:binding-table-rule />

名称	类型	说明
logic-tables	属性	绑定表名称，多个表以逗号分隔

<sharding:broadcast-table-rules />

名称	类型	说明
broadcast-table-rule (+)	标签	广播表规则配置

<sharding:broadcast-table-rule />

名称	类型	说明
table	属性	广播表名称

<sharding:standard-strategy />

名称	类型	说明
id	属性	标准分片策略名称
sharding-column	属性	分片列名称
algorithm-ref	属性	分片算法名称

<sharding:complex-strategy />

名称	类型	说明
id	属性	复合分片策略名称
sharding-columns	属性	分片列名称，多个列以逗号分隔
algorithm-ref	属性	分片算法名称

<sharding:hint-strategy />

名称	类型	说明
id	属性	Hint 分片策略名称
algorithm-ref	属性	分片算法名称

<sharding:none-strategy />

名称	类型	说明
id	属性	分片策略名称

<sharding:key-generate-strategy />

名称	类型	说明
id	属性	分布式序列策略名称
column	属性	分布式序列列名称
algorithm-ref	属性	分布式序列算法名称

<sharding:sharding-algorithm />

名称	类型	说明
id	属性	分片算法名称
type	属性	分片算法类型
props (?)	标签	分片算法属性配置

<sharding:key-generate-algorithm />

名称	类型	说明
id	属性	分布式序列算法名称
type	属性	分布式序列算法类型
props (?)	标签	分布式序列算法属性配置

算法类型的详情，请参见内置分片算法列表和内置分布式序列算法列表。

注意事项

行表达式标识符可以使用 `${...}` 或 `$->{...}`，但前者与 Spring 本身的属性文件占位符冲突，因此在 Spring 环境中使用行表达式标识符建议使用 `$->{...}`。

读写分离

配置项说明

命名空间：http://shardingsphere.apache.org/schema/shardingsphere/readwrite-splitting/readwrite-splitting-5.0.0.xsd

<readwrite-splitting:rule />

名称	类型	说明
id	属性	Spring Bean Id
data-source-rule (+)	标签	读写分离数据源规则配置

<readwrite-splitting:data-source-rule />

名称	类型	说明
id	属性	读写分离数据源规则名称
auto-aware-data-source-name (?)	属性	自动感知数据源名称
write-data-source-name	属性	写数据源名称
read-data-source-names	属性	读数据源名称，多个读数据源用逗号分隔
load-balance-algorithm-ref (?)	属性	负载均衡算法名称

<readwrite-splitting:load-balance-algorithm />

名称	类型	说明
id	属性	负载均衡算法名称
type	属性	负载均衡算法类型
props (?)	标签	负载均衡算法属性配置

算法类型的详情，请参见内置负载均衡算法列表。

数据加密

配置项说明

命名空间：http://shardingsphere.apache.org/schema/shardingsphere/encrypt/encrypt-5.0.0.xsd

<encrypt:rule />

名称	类型	说明	默认值
id	属性	Spring Bean Id	
queryWithCipherColumn (?)	属性	是否使用加密列进行查询。在有原文列的情况下，可以使用原文列进行查询	true
table (+)	标签	加密表配置	

<encrypt:table />

名称	类型	说明
name	属性	加密表名称
column (+)	标签	加密列配置

<encrypt:column />

名称	类型	说明
logic-column	属性	加密列逻辑名称
cipher-column	属性	加密列名称
assisted-query-column (?)	属性	查询辅助列名称
plain-column (?)	属性	原文列名称
encrypt-algorithm-ref	属性	加密算法名称

<encrypt:encrypt-algorithm />

名称	类型	说明
id	属性	加密算法名称
type	属性	加密算法类型
props (?)	标签	加密算法属性配置

算法类型的详情，请参见内置加密算法列表。

影子库

配置项说明

命名空间：http://shardingsphere.apache.org/schema/shardingsphere/shadow/shadow-5.0.0.xsd

<shadow:rule />

名称	类型	说明
id	属性	Spring Bean Id
column	属性	影子字段名称
mappings(?)	标签	生产数据库与影子数据库的映射关系配置

<shadow:mapping />

名称	类型	说明
product-data-source-name	属性	生产数据库名称
shadow-data-source-name	属性	影子数据库名称

分布式治理

配置项说明

命名空间: http://shardingsphere.apache.org/schema/shardingsphere/governance/governance-5.0.0.xsd

<governance:reg-center />

名称	类型	说明
id	属性	注册中心实例名称
schema-name (?)	属性	JDBC 数据源别名，该参数可实现 JDBC 与 PROXY 共享配置
type	属性	注册中心类型。如：ZooKeeper, etcd
namespace	属性	注册中心命名空间
server-lists	属性	注册中心服务列表。包括 IP 地址和端口号。多个地址用逗号分隔。如: host1:2181,host2:2181
props (?)	属性	配置本实例需要的其他参数，例如 ZooKeeper 的连接参数等

ShardingSphere-4.x

数据分片

配置项说明

命名空间：http://shardingsphere.apache.org/schema/shardingsphere/sharding/sharding-4.0.0.xsd

<sharding:data-source />

名称	类型	说明
id	属性	Spring Bean Id
sharding-rule	标签	数据分片配置规则
props (?)	标签	属性配置

<sharding:sharding-rule />

名称	• 说明 类 | 型 | * |
data-source-names	属 | 数据源 Bean 列表，多个 Bean 以逗号分隔性 |
table-rules	标 | 表分片规则配置对象签 |
binding-table-rules (?)	标 | 绑定表规则列表签 |
broadcast-table-rules (?)	标 | 广播表规则列表签 |
def ault-data-source-name (?)	属 | 未配置分片规则的表将通过默认数据源定位性 |
default- database-strategy-ref (?)	属 | 默认数据库分片策略，对应性 | <sharding:xxx-strategy> 中的策略 Id，缺省表示不分库
defau lt-table-strategy-ref (?)	属 | 默认表分片策略，对应性 | <sharding:xxx-strategy> 中的策略 Id，缺省表示不分表
defa ult-key-generator-ref (?)	属 | 默认自增列值生成器引用，缺省使用性 | org.apache.shardingsphere.core.key gen.generator.impl.SnowflakeKeyGenerator
encrypt-rule (?)	标 | 脱敏规则签 |

<sharding:table-rules />

名称	类型	说明
table-rule (+)	标签	表分片规则配置对象

<sharding:table-rule />

<sharding:binding-table-rules />

名称	类型	说明
binding-table-rule (+)	标签	绑定表规则

<sharding:binding-table-rule />

名称	类型	说明
logic-tables	属性	绑定规则的逻辑表名，多表以逗号分隔

<sharding:broadcast-table-rules />

名称	类型	说明
broadcast-table-rule (+)	标签	广播表规则

<sharding:broadcast-table-rule />

名称	类型	说明
table	属性	广播规则的表名

<sharding:standard-strategy />

名称	• 说明 类 | 型 | * |
id	属 | Spring Bean Id 性 |
sharding-column	属 | 分片列名称性 |
precis e-algorithm-ref	属 | 精确分片算法引用，用于 = 和 IN。该类需实现性 | Precise-ShardingAlgorithm 接口
rang e-algorithm-ref (?)	属 | 范围分片算法引用，用于 BETWEEN。该类需实现性 | Range-ShardingAlgorithm 接口

<sharding:complex-strategy />

名称	类型	说明
id	属性	Spring Bean Id
sharding-columns	属性	分片列名称，多个列以逗号分隔
algorithm-ref	属性	复合分片算法引用。该类需实现 ComplexKeysShardingAlgorithm 接口

<sharding:inline-strategy />

名称	类型	说明
id	属性	Spring Bean Id
sharding-column	属性	分片列名称
algorithm-expression	属性	分片算法行表达式，需符合 groovy 语法

<sharding:hint-database-strategy />

名称	类型	说明
id	属性	Spring Bean Id
algorithm-ref	属性	Hint 分片算法。该类需实现 HintShardingAlgorithm 接口

<sharding:none-strategy />

名称	类型	说明
id	属性	Spring Bean Id

<sharding:key-generator />

名称	类型	说明
column	属性	自增列名称
type	属性	自增列值生成器类型，可自定义或选择内置类型：SNOWFLAKE/UUID
props-ref	属性	自增列值生成器的属性配置引用

Properties

属性配置项，可以为以下自增列值生成器的属性。

SNOWFLAKE

名称	• 说明 类 | 型 | * |	
max.tolerate.time.difference.milliseconds (?)	l o n g	最大容忍时钟回退时间,单位:毫秒。默认为 10 毫秒
max.vibration.offset (?)	i n t	最大抖动上限值，范围 [0, 4096)，默认为 1。注: 若使用此算法生成值作分片值,建议配置此属性。此算法在不同毫秒内所生成的 key 取模 2^n (2^n 一般为分库或分表数) 之后结果总为 0 或 1。为防止上述分片问题，建议将此属性值配置为 (2^n)-1

<sharding:encrypt-rule />

名称	类型	说明
encrypt:encrypt-rule (?)	标签	加解密规则

名称	类型	说明
sql.show (?)	属性	是否开启 SQL 显示，默认值: false
executor.size (?)	属性	工作线程数量，默认值: CPU 核数
max .connections.size.per.query (?)	属性	每个物理数据库为每次查询分配的最大连接数量。默认值: 1
c heck.table.metadata.enabled (?)	属性	是否在启动时检查分表元数据一致性，默认值: false
query.with.cipher.column (?)	属性	当存在明文列时，是否使用密文列查询，默认值: true

读写分离

配置项说明

命名空间：http://shardingsphere.apache.org/schema/shardingsphere/masterslave/master-slave.xsd

<master-slave:data-source />

<master-slave:props />

名称	类型	说明
sql.show (?)	属性	是否开启 SQL 显示，默认值: false
executor.size (?)	属性	工作线程数量，默认值: CPU 核数
max .connections.size.per.query (?)	属性	每个物理数据库为每次查询分配的最大连接数量。默认值: 1
c heck.table.metadata.enabled (?)	属性	是否在启动时检查分表元数据一致性，默认值: false

<master-slave:load-balance-algorithm /> 4.0.0-RC2 版本添加

名称	类型	说明
id	属性	Spring Bean Id
type (?)	属性	负载均衡算法类型，‘RANDOM’或’ROUND_ROBIN’，支持自定义拓展
props-ref (?)	属性	负载均衡算法配置参数

数据脱敏

配置项说明

命名空间：http://shardingsphere.apache.org/schema/shardingsphere/encrypt/encrypt.xsd

<encrypt:data-source />

名称	类型	说明
id	属性	Spring Bean Id
data-source-name	属性	加密数据源 Bean Id
props (?)	标签	属性配置

<encrypt:encryptors />

名称	类型	说明
encryptor (+)	标签	加密器配置

<encrypt:encryptor />

名称	类型	说明
id	属性	加密器的名称
type	属性	加解密器类型，可自定义或选择内置类型：MD5/AES
props-ref	属性	属性配置, 注意：使用 AES 加密器, 需要配置 AES 加密器的 KEY 属性：aes.key.value

<encrypt:tables />

名称	类型	说明
table (+)	标签	加密表配置

<encrypt:table />

名称	类型	说明
column (+)	标签	加密列配置

<encrypt:column />

<encrypt:props />

名称	类型	说明
sql.show (?)	属性	是否开启 SQL 显示，默认值: false
query.with.cipher.column (?)	属性	当存在明文列时，是否使用密文列查询，默认值: true

治理

数据分片 + 治理

配置项说明

命名空间：http://shardingsphere.apache.org/schema/shardingsphere/orchestration/orchestration.xsd

<orchestration:sharding-data-source />

名称	类型	说明
id	属性	ID
data-source-ref (?)	属性	被治理的数据库 id
registry-center-ref	属性	注册中心 id
overwrite	属性	本地配置是否覆盖注册中心配置。如果可覆盖，每次启动都以本地配置为准。缺省为不覆盖

读写分离 + 治理

配置项说明

命名空间：http://shardingsphere.apache.org/schema/shardingsphere/orchestration/orchestration.xsd

<orchestration:master-slave-data-source />

名称	类型	说明
id	属性	ID
data-source-ref (?)	属性	被治理的数据库 id
registry-center-ref	属性	注册中心 id
overwrite	属性	本地配置是否覆盖注册中心配置。如果可覆盖，每次启动都以本地配置为准。缺省为不覆盖

数据脱敏 + 治理

配置项说明

命名空间：http://shardingsphere.apache.org/schema/shardingsphere/orchestration/orchestration.xsd

<orchestration:encrypt-data-source />

名称	类型	说明
id	属性	ID
data-source-ref (?)	属性	被治理的数据库 id
registry-center-ref	属性	注册中心 id
overwrite	属性	本地配置是否覆盖注册中心配置。如果可覆盖，每次启动都以本地配置为准。缺省为不覆盖

治理注册中心

配置项说明

命名空间：http://shardingsphere.apache.org/schema/shardingsphere/orchestration/orchestration.xsd

<orchestration:registry-center />

名称	类型	说明
id	属性	注册中心的 Spring Bean Id
type	属性	注册中心类型。如：zookeeper
server-lists	属性	连接注册中心服务器的列表，包括 IP 地址和端口号，多个地址用逗号分隔。如: host1:2181,host2:2181
namespace (?)	属性	注册中心的命名空间
digest (?)	属性	连接注册中心的权限令牌。缺省为不需要权限验证
ope ration-timeout-milliseconds (?)	属性	操作超时的毫秒数，默认 500 毫秒
max-retries (?)	属性	连接失败后的最大重试次数，默认 3 次
retry-interval-milliseconds (?)	属性	重试间隔毫秒数，默认 500 毫秒
time-to-live-seconds (?)	属性	临时节点存活秒数，默认 60 秒
props-ref (?)	属性	配置中心其它属性

ShardingSphere-3.x

数据分片

配置项说明

命名空间：http://shardingsphere.apache.org/schema/shardingsphere/sharding/sharding.xsd

<sharding:data-source />

名称	类型	说明
id	属性	Spring Bean Id
sharding-rule	标签	数据分片配置规则
config-map (?)	标签	用户自定义配置
props (?)	标签	属性配置

<sharding:sharding-rule />

名称	• 说明 类 | 型 | * |
data-source-names	属 | 数据源 Bean 列表，多个 Bean 以逗号分隔性 |
table-rules	标 | 表分片规则配置对象签 |
binding-table-rules (?)	标 | 绑定表规则列表签 |
broadcast-table-rules (?)	标 | 广播表规则列表签 |
def ault-data-source-name (?)	属 | 未配置分片规则的表将通过默认数据源定位性 |
default- database-strategy-ref (?)	属 | 默认数据库分片策略，对应性 | <sharding:xxx-strategy> 中的策略 Id，缺省表示不分库
defau lt-table-strategy-ref (?)	属 | 默认表分片策略，对应性 | <sharding:xxx-strategy> 中的策略 Id，缺省表示不分表
defa ult-key-generator-ref (?)	属 | 默认自增列值生成器引用，缺省使用性 | io.shardingsphere.core.keygen.Default KeyGenerator。该类需实现 KeyGenerator 接口

<sharding:table-rules />

名称	类型	说明
table-rule (+)	标签	表分片规则配置对象

<sharding:table-rule />

<sharding:binding-table-rules />

名称	类型	说明
binding-table-rule (+)	标签	绑定表规则

名称	类型	说明
table	属性	广播规则的表名

<sharding:standard-strategy />

<sharding:complex-strategy />

名称	类型	说明
id	属性	Spring Bean Id
sharding-columns	属性	分片列名称，多个列以逗号分隔
algorithm-ref	属性	复合分片算法引用。该类需实现 ComplexKeysShardingAlgorithm 接口

<sharding:inline-strategy />

名称	类型	说明
id	属性	Spring Bean Id
sharding-column	属性	分片列名称
algorithm-expression	属性	分片算法行表达式，需符合 groovy 语法

<sharding:hint-database-strategy />

名称	类型	说明
id	属性	Spring Bean Id
algorithm-ref	属性	Hint 分片算法。该类需实现 HintShardingAlgorithm 接口

<sharding:none-strategy />

名称	类型	说明
id	属性	Spring Bean Id

<sharding:props />

名称	类型	说明
sql.show (?)	属性	是否开启 SQL 显示，默认值: false
executor.size (?)	属性	工作线程数量，默认值: CPU 核数
max .connections.size.per.query (?)	属性	每个物理数据库为每次查询分配的最大连接数量。默认值: 1
c heck.table.metadata.enabled (?)	属性	是否在启动时检查分表元数据一致性，默认值: false

<sharding:config-map />

读写分离

配置项说明

命名空间：http://apache.shardingsphere.org/schema/shardingsphere/masterslave/master-slave.xsd

<master-slave:data-source />

名称	类型	说明
id	属性	Spring Bean Id
master-da ta-source-name	属性	主库数据源 Bean Id
slave-dat a-source-names	属性	从库数据源 Bean Id 列表，多个 Bean 以逗号分隔
strategy-ref (?)	属性	从库负载均衡算法引用。该类需实现 MasterSlaveLoadBalanceAlgorithm 接口
strategy-type (?)	属性	从库负载均衡算法类型，可选值：ROUND_ROBI N，RANDOM。若 strategy-ref 存在则忽略该配置
config-map (?)	属性	用户自定义配置
props (?)	属性	属性配置

<master-slave:config-map />

<master-slave:props />

名称	类型	说明
sql.show (?)	属性	是否开启 SQL 显示，默认值: false
executor.size (?)	属性	工作线程数量，默认值: CPU 核数
max .connections.size.per.query (?)	属性	每个物理数据库为每次查询分配的最大连接数量。默认值: 1
c heck.table.metadata.enabled (?)	属性	是否在启动时检查分表元数据一致性，默认值: false

治理

数据分片 + 数据治理

配置项说明

命名空间：http://shardingsphere.apache.org/schema/shardingsphere/orchestration/orchestration.xsd

<orchestration:sharding-data-source />

名称	类型	说明
id	属性	ID
data-source-ref (?)	属性	被治理的数据库 id
registry-center-ref	属性	注册中心 id
overwrite (?)	属性	本地配置是否覆盖注册中心配置。如果可覆盖，每次启动都以本地配置为准。缺省为不覆盖

读写分离 + 数据治理

配置项说明

命名空间：http://shardingsphere.apache.org/schema/shardingsphere/orchestration/orchestration.xsd

<orchestration:master-slave-data-source />

名称	类型	说明
id	属性	ID
data-source-ref (?)	属性	被治理的数据库 id
registry-center-ref	属性	注册中心 id
overwrite (?)	属性	本地配置是否覆盖注册中心配置。如果可覆盖，每次启动都以本地配置为准。缺省为不覆盖

数据治理注册中心

配置项说明

命名空间：http://shardingsphere.apache.org/schema/shardingsphere/orchestration/orchestration.xsd

<orchestration:registry-center />

名称	类型	说明
id	属性	注册中心的 Spring Bean Id
server-lists	属性	连接注册中心服务器的列表，包括 IP 地址和端口号，多个地址用逗号分隔。如: host1:2181,host2:2181
namespace (?)	属性	注册中心的命名空间
digest (?)	属性	连接注册中心的权限令牌。缺省为不需要权限验证
operation-timeout-milliseconds (?)	属性	操作超时的毫秒数，默认 500 毫秒
max-retries (?)	属性	连接失败后的最大重试次数，默认 3 次
retry-interval-milliseconds (?)	属性	重试间隔毫秒数，默认 500 毫秒
time-to-live-seconds (?)	属性	临时节点存活秒数，默认 60 秒

ShardingSphere-2.x

数据分片

配置项说明

<sharding:data-source/>

定义 sharding-jdbc 数据源

名称	类型	数据类型	必填	说明
id	属性	String	是	Spring Bean ID
sharding-rule	标签	•	是	分片规则
binding-table-rules (?)	标签	•	否	分片规则
props (?)	标签	•	否	相关属性配置

<sharding:sharding-rule/>

<sharding:table-rules/>

名称	类型	数据类型	必填	说明
table-rule(+)	标签	•	是	分片规则

<sharding:table-rule/>

<sharding:binding-table-rules/>

名称	类型	数据类型	必填	说明
binding-table-rule	标签	•	是	绑定规则

<sharding:binding-table-rule/>

名称	类型	数据类型	必填	说明
logic-tables	属性	String	是	逻辑表名，多个表名以逗号分隔

<sharding:standard-strategy/> 标准分片策略，用于单分片键的场景

名称	• *数 类 | 据型 | 类 * | 型 *	• 说明 必 | 填 | * |
sha rding-column	属 | Str | 是 | 分片列名性 | ing | |	
precise-alg orithm-class	属 | Str | 是 | 精性 | ing | | 确的分片算法类名称，用于 = 和 IN。该类需 | | 使用默认的构造器或者提供无参数的构造器	
range-alg orithm-class (?)	属 | Str | 否 | 范围性 | ing | | 的分片算法类名称，用于 BETWEEN。该类需 | | 使用默认的构造器或者提供无参数的构造器	

<sharding:complex-strategy/> 复合分片策略，用于多分片键的场景

名称	类型	数据类型	必填	说明
sharding-columns	属性	String	是	分片列名，多个列以逗号分隔
algorithm-class	属性	String	是	分片算法全类名，该类需使用默认的构造器或者提供无参数的构造器

<sharding:inline-strategy/> inline 表达式分片策略

名称	类型	数据类型	必填	说明
sharding-column	属性	String	是	分片列名
algorithm-expression	属性	String	是	分片算法表达式

<sharding:hint-database-strategy/> Hint 方式分片策略

名称	类型	数据类型	必填	说明
algorithm-class	属性	String	是	分片算法全类名，该类需使用默认的构造器或者提供无参数的构造器

<sharding:none-strategy/> 不分片的策略

<sharding:props/>

名称	类型	数据类型	必填	说明
sql.show	属性	boolean	是	是否开启 SQL 显示，默认为 false 不开启
executor.size (?)	属性	int	否	最大工作线程数量

读写分离

配置项说明

<master-slave:data-source/> 定义 sharding-jdbc 读写分离的数据源

Spring 格式特别说明

如需使用 inline 表达式，需配置 ignore-unresolvable 为 true，否则 placeholder 会把 inline 表达式当成属性 key 值导致出错。

分片算法表达式语法说明

inline 表达式特别说明

${begin..end} 表示范围区间

${[unit1, unit2, unitX]} 表示枚举值

inline 表达式中连续多个 ${…} 表达式，整个 inline 最终的结果将会根据每个子表达式的结果进行笛卡尔组合，例如正式表 inline 表达式如下：

```
dbtbl_${['online', 'offline']}_${1..3}
```

最终会解析为 dbtbl_online_1，dbtbl_online_2，dbtbl_online_3，dbtbl_offline_1，dbtbl_offline_2 和 dbtbl_offline_3 这 6 张表。

字符串内嵌 groovy 代码

表达式本质上是一段字符串，字符串中使用 ${} 来嵌入 groovy 代码。

```
data_source_${id % 2 + 1}
```

上面的表达式中 data_source_ 是字符串前缀，id % 2 + 1 是 groovy 代码。

治理

Zookeeper 标签说明

名称	类型	是否必填	缺省值	描述
id	String	是		注册中心在 Spring 容器中的主键
se rver-lists	String	是		连接 Zookeeper 服务器的列表包括 IP 地址和端口号多个地址用逗号分隔如: host1:2181,host2:2181
namespace	String	是		Zookeeper 的命名空间
base-sle ep-time-milliseconds (?)	int	否	1 000	等待重试的间隔时间的初始值单位：毫秒
max-sle ep-time-milliseconds (?)	int	否	3 000	等待重试的间隔时间的最大值单位：毫秒
m ax-retries (?)	int	否	3	最大重试次数
session- timeout-milliseconds (?)	int	否	60 000	会话超时时间单位：毫秒
c onnection-timeout-mi llisec-onds (?)	int	否	15 000	连接超时时间单位：毫秒
digest (?)	String	否		连接 Zookeep er 的权限令牌缺省为不需要权限验证

Etcd 配置示例

名称	• *是 类 | 否型 | 必 * | 填 *		• 缺省值 *	描述
id	String	是		注册中心在 Spring 容器中的主键
server-lists	String	是		连接 Etcd 服务器的列表包括 IP 地址和端口号多个地址用逗号分隔如: http://host1:2379,http://host2:2379
time-to-live-seconds (?)	int	否	60	临时节点存活时间单位: 秒
timeout-milliseconds (?)	int	否	500	每次请求的超时时间单位: 毫秒
max-retries (?)	int	否	3	每次请求的最大重试次数
retry-interval-milliseconds (?)	int	否	200	重试间隔时间单位: 毫秒

Spring Boot Start 配置

5.0.0-alpha

数据分片

配置项说明

```
spring.shardingsphere.datasource.names= # 省略数据源配置

# 标准分片表配置
spring.shardingsphere.rules.sharding.tables.<table-name>.actual-data-nodes= # 由数据源名 + 表名组成，以小数点分隔。多个表以逗号分隔，支持 inline 表达式。缺省表示使用已知数据源与逻辑表名称生成数据节点，用于广播表（即每个库中都需要一个同样的表用于关联查询，多为字典表）或只分库不分表且所有库的表结构完全一致的情况

# 分库策略，缺省表示使用默认分库策略，以下的分片策略只能选其一
```

```
# 用于单分片键的标准分片场景
spring.shardingsphere.rules.sharding.tables.<table-name>.database-strategy.
standard.<sharding-algorithm-name>.sharding-column= # 分片列名称
spring.shardingsphere.rules.sharding.tables.<table-name>.database-strategy.
standard.<sharding-algorithm-name>.sharding-algorithm-name= # 分片算法名称

# 用于多分片键的复合分片场景
spring.shardingsphere.rules.sharding.tables.<table-name>.database-strategy.complex.
<sharding-algorithm-name>.sharding-columns= # 分片列名称，多个列以逗号分隔
spring.shardingsphere.rules.sharding.tables.<table-name>.database-strategy.complex.
<sharding-algorithm-name>.sharding-algorithm-name= # 分片算法名称

# 用于 Hint 的分片策略
spring.shardingsphere.rules.sharding.tables.<table-name>.database-strategy.hint.
<sharding-algorithm-name>.sharding-algorithm-name= # 分片算法名称

# 分表策略，同分库策略
spring.shardingsphere.rules.sharding.tables.<table-name>.table-strategy.xxx= # 省略

# 自动分片表配置
spring.shardingsphere.rules.sharding.auto-tables.<auto-table-name>.actual-data-
sources= # 数据源名

spring.shardingsphere.rules.sharding.auto-tables.<auto-table-name>.sharding-
strategy.standard.sharding-column= # 分片列名称
spring.shardingsphere.rules.sharding.auto-tables.<auto-table-name>.sharding-
strategy.standard.sharding-algorithm= # 自动分片算法名称

# 分布式序列策略配置
spring.shardingsphere.rules.sharding.tables.<table-name>.key-generate-strategy.
column= # 分布式序列列名称
spring.shardingsphere.rules.sharding.tables.<table-name>.key-generate-strategy.key-
generator-name= # 分布式序列算法名称

spring.shardingsphere.rules.sharding.binding-tables[0]= # 绑定表规则列表
spring.shardingsphere.rules.sharding.binding-tables[1]= # 绑定表规则列表
spring.shardingsphere.rules.sharding.binding-tables[x]= # 绑定表规则列表

spring.shardingsphere.rules.sharding.broadcast-tables[0]= # 广播表规则列表
spring.shardingsphere.rules.sharding.broadcast-tables[1]= # 广播表规则列表
spring.shardingsphere.rules.sharding.broadcast-tables[x]= # 广播表规则列表

spring.shardingsphere.sharding.default-database-strategy.xxx= # 默认数据库分片策略
spring.shardingsphere.sharding.default-table-strategy.xxx= # 默认表分片策略
spring.shardingsphere.sharding.default-key-generate-strategy.xxx= # 默认分布式序列策略

# 分片算法配置
spring.shardingsphere.rules.sharding.sharding-algorithms.<sharding-algorithm-name>.
type= # 分片算法类型
```

```
spring.shardingsphere.rules.sharding.sharding-algorithms.<sharding-algorithm-name>.
props.xxx=# 分片算法属性配置

# 分布式序列算法配置
spring.shardingsphere.rules.sharding.key-generators.<key-generate-algorithm-name>.
type= # 分布式序列算法类型
spring.shardingsphere.rules.sharding.key-generators.<key-generate-algorithm-name>.
props.xxx= # 分布式序列算法属性配置
```

读写分离

配置项说明

```
spring.shardingsphere.datasource.names= # 省略数据源配置

spring.shardingsphere.rules.replica-query.data-sources.<replica-query-data-source-
name>.primary-data-source-name= # 主数据源名称
spring.shardingsphere.rules.replica-query.data-sources.<replica-query-data-source-
name>.replica-data-source-names= # 从数据源名称，多个从数据源用逗号分隔
spring.shardingsphere.rules.replica-query.data-sources.<replica-query-data-source-
name>.load-balancer-name= # 负载均衡算法名称

# 负载均衡算法配置
spring.shardingsphere.rules.replica-query.load-balancers.<load-balance-algorithm-
name>.type= # 负载均衡算法类型
spring.shardingsphere.rules.replica-query.load-balancers.<load-balance-algorithm-
name>.props.xxx= # 负载均衡算法属性配置
```

数据加密

配置项说明

```
spring.shardingsphere.datasource.names= # 省略数据源配置

spring.shardingsphere.rules.encrypt.tables.<table-name>.columns.<column-name>.
cipher-column= # 加密列名称
spring.shardingsphere.rules.encrypt.tables.<table-name>.columns.<column-name>.
assisted-query-column= # 查询列名称
spring.shardingsphere.rules.encrypt.tables.<table-name>.columns.<column-name>.
plain-column= # 原文列名称
spring.shardingsphere.rules.encrypt.tables.<table-name>.columns.<column-name>.
encryptor-name= # 加密算法名称

# 加密算法配置
```

```
spring.shardingsphere.rules.encrypt.encryptors.<encrypt-algorithm-name>.type= # 加密算法类型
spring.shardingsphere.rules.encrypt.encryptors.<encrypt-algorithm-name>.props.xxx= # 加密算法属性配置
```

影子库

配置项说明

```
spring.shardingsphere.datasource.names= # 省略数据源配置

spring.shardingsphere.rules.shadow.column= # 影子字段名称
spring.shardingsphere.rules.shadow.shadow-mappings.<product-data-source-name>= # 影子数据库名称
```

分布式治理

配置项说明

```
spring.shardingsphere.governance.name= # 治理名称
spring.shardingsphere.governance.registry-center.type= # 治理持久化类型。如: Zookeeper, etcd, Apollo, Nacos
spring.shardingsphere.governance.registry-center.server-lists= # 治理服务列表。包括 IP 地址和端口号。多个地址用逗号分隔。如: host1:2181,host2:2181
spring.shardingsphere.governance.registry-center.props= # 其它配置
spring.shardingsphere.governance.additional-config-center.type= # 可选的配置中心类型。如: Zookeeper, etcd, Apollo, Nacos
spring.shardingsphere.governance.additional-config-center.server-lists= # 可选的配置中心服务列表。包括 IP 地址和端口号。多个地址用逗号分隔。如: host1:2181,host2:2181
spring.shardingsphere.governance.additional-config-center.props= # 可选的配置中心其它配置
spring.shardingsphere.governance.overwrite= # 本地配置是否覆盖配置中心配置。如果可覆盖, 每次启动都以本地配置为准.
```

ShardingSphere-4.x

数据分片

配置项说明

```
spring.shardingsphere.datasource.names= # 数据源名称, 多数据源以逗号分隔

spring.shardingsphere.datasource.<data-source-name>.type= # 数据库连接池类名称
```

```
spring.shardingsphere.datasource.<data-source-name>.driver-class-name= # 数据库驱动类名
spring.shardingsphere.datasource.<data-source-name>.url= # 数据库 url 连接
spring.shardingsphere.datasource.<data-source-name>.username= # 数据库用户名
spring.shardingsphere.datasource.<data-source-name>.password= # 数据库密码
spring.shardingsphere.datasource.<data-source-name>.xxx= # 数据库连接池的其它属性

spring.shardingsphere.sharding.tables.<logic-table-name>.actual-data-nodes= # 由数据源名 + 表名组成，以小数点分隔。多个表以逗号分隔，支持 inline 表达式。缺省表示使用已知数据源与逻辑表名称生成数据节点，用于广播表（即每个库中都需要一个同样的表用于关联查询，多为字典表）或只分库不分表且所有库的表结构完全一致的情况

# 分库策略，缺省表示使用默认分库策略，以下的分片策略只能选其一

# 用于单分片键的标准分片场景
spring.shardingsphere.sharding.tables.<logic-table-name>.database-strategy.standard.sharding-column= # 分片列名称
spring.shardingsphere.sharding.tables.<logic-table-name>.database-strategy.standard.precise-algorithm-class-name= # 精确分片算法类名称，用于 = 和 IN。该类需实现 PreciseShardingAlgorithm 接口并提供无参数的构造器
spring.shardingsphere.sharding.tables.<logic-table-name>.database-strategy.standard.range-algorithm-class-name= # 范围分片算法类名称，用于 BETWEEN，可选。该类需实现 RangeShardingAlgorithm 接口并提供无参数的构造器

# 用于多分片键的复合分片场景
spring.shardingsphere.sharding.tables.<logic-table-name>.database-strategy.complex.sharding-columns= # 分片列名称，多个列以逗号分隔
spring.shardingsphere.sharding.tables.<logic-table-name>.database-strategy.complex.algorithm-class-name= # 复合分片算法类名称。该类需实现 ComplexKeysShardingAlgorithm 接口并提供无参数的构造器

# 行表达式分片策略
spring.shardingsphere.sharding.tables.<logic-table-name>.database-strategy.inline.sharding-column= # 分片列名称
spring.shardingsphere.sharding.tables.<logic-table-name>.database-strategy.inline.algorithm-expression= # 分片算法行表达式，需符合 groovy 语法

# Hint 分片策略
spring.shardingsphere.sharding.tables.<logic-table-name>.database-strategy.hint.algorithm-class-name= # Hint 分片算法类名称。该类需实现 HintShardingAlgorithm 接口并提供无参数的构造器

# 分表策略，同分库策略
spring.shardingsphere.sharding.tables.<logic-table-name>.table-strategy.xxx= # 省略

spring.shardingsphere.sharding.tables.<logic-table-name>.key-generator.column= # 自增列名称，缺省表示不使用自增主键生成器
spring.shardingsphere.sharding.tables.<logic-table-name>.key-generator.type= # 自增列值生成器类型，缺省表示使用默认自增列值生成器。可使用用户自定义的列值生成器或选择内置类型：SNOWFLAKE/UUID
```

```
spring.shardingsphere.sharding.tables.<logic-table-name>.key-generator.props.
<property-name>= # 属性配置，注意：使用 SNOWFLAKE 算法，需要配置 max.tolerate.time.
difference.milliseconds 属性。若使用此算法生成值作分片值，建议配置 max.vibration.offset 属性

spring.shardingsphere.sharding.binding-tables[0]= # 绑定表规则列表
spring.shardingsphere.sharding.binding-tables[1]= # 绑定表规则列表
spring.shardingsphere.sharding.binding-tables[x]= # 绑定表规则列表

spring.shardingsphere.sharding.broadcast-tables[0]= # 广播表规则列表
spring.shardingsphere.sharding.broadcast-tables[1]= # 广播表规则列表
spring.shardingsphere.sharding.broadcast-tables[x]= # 广播表规则列表

spring.shardingsphere.sharding.default-data-source-name= # 未配置分片规则的表将通过默认
数据源定位
spring.shardingsphere.sharding.default-database-strategy.xxx= # 默认数据库分片策略，同分
库策略
spring.shardingsphere.sharding.default-table-strategy.xxx= # 默认表分片策略，同分表策略
spring.shardingsphere.sharding.default-key-generator.type= # 默认自增列值生成器类型，缺
省将使用 org.apache.shardingsphere.core.keygen.generator.impl.SnowflakeKeyGenerator。
可使用用户自定义的列值生成器或选择内置类型：SNOWFLAKE/UUID
spring.shardingsphere.sharding.default-key-generator.props.<property-name>= # 自增列
值生成器属性配置，比如 SNOWFLAKE 算法的 max.tolerate.time.difference.milliseconds

spring.shardingsphere.sharding.master-slave-rules.<master-slave-data-source-name>.
master-data-source-name= # 详见读写分离部分
spring.shardingsphere.sharding.master-slave-rules.<master-slave-data-source-name>.
slave-data-source-names[0]= # 详见读写分离部分
spring.shardingsphere.sharding.master-slave-rules.<master-slave-data-source-name>.
slave-data-source-names[1]= # 详见读写分离部分
spring.shardingsphere.sharding.master-slave-rules.<master-slave-data-source-name>.
slave-data-source-names[x]= # 详见读写分离部分
spring.shardingsphere.sharding.master-slave-rules.<master-slave-data-source-name>.
load-balance-algorithm-class-name= # 详见读写分离部分
spring.shardingsphere.sharding.master-slave-rules.<master-slave-data-source-name>.
load-balance-algorithm-type= # 详见读写分离部分

spring.shardingsphere.props.sql.show= # 是否开启 SQL 显示，默认值：false
spring.shardingsphere.props.executor.size= # 工作线程数量，默认值：CPU 核数
```

读写分离

配置项说明

```
# 省略数据源配置，与数据分片一致

spring.shardingsphere.sharding.master-slave-rules.<master-slave-data-source-name>.
master-data-source-name= # 主库数据源名称
spring.shardingsphere.sharding.master-slave-rules.<master-slave-data-source-name>.
slave-data-source-names[0]= # 从库数据源名称列表
spring.shardingsphere.sharding.master-slave-rules.<master-slave-data-source-name>.
slave-data-source-names[1]= # 从库数据源名称列表
spring.shardingsphere.sharding.master-slave-rules.<master-slave-data-source-name>.
slave-data-source-names[x]= # 从库数据源名称列表
spring.shardingsphere.sharding.master-slave-rules.<master-slave-data-source-name>.
load-balance-algorithm-class-name= # 从库负载均衡算法类名称。该类需实现
MasterSlaveLoadBalanceAlgorithm 接口且提供无参数构造器
spring.shardingsphere.sharding.master-slave-rules.<master-slave-data-source-name>.
load-balance-algorithm-type= # 从库负载均衡算法类型，可选值：ROUND_ROBIN，RANDOM。若
`load-balance-algorithm-class-name` 存在则忽略该配置

spring.shardingsphere.props.sql.show= # 是否开启 SQL 显示，默认值：false
spring.shardingsphere.props.executor.size= # 工作线程数量，默认值：CPU 核数
spring.shardingsphere.props.check.table.metadata.enabled= # 是否在启动时检查分表元数据一
致性，默认值：false
```

数据脱敏

配置项说明

```
# 省略数据源配置，与数据分片一致

spring.shardingsphere.encrypt.encryptors.<encryptor-name>.type= # 加解密器类型，可自定
义或选择内置类型：MD5/AES
spring.shardingsphere.encrypt.encryptors.<encryptor-name>.props.<property-name>= #
属性配置，注意：使用 AES 加密器，需要配置 AES 加密器的 KEY 属性：aes.key.value
spring.shardingsphere.encrypt.tables.<table-name>.columns.<logic-column-name>.
plainColumn= # 存储明文的字段
spring.shardingsphere.encrypt.tables.<table-name>.columns.<logic-column-name>.
cipherColumn= # 存储密文的字段
spring.shardingsphere.encrypt.tables.<table-name>.columns.<logic-column-name>.
assistedQueryColumn= # 辅助查询字段，针对 ShardingQueryAssistedEncryptor 类型的加解密器进
行辅助查询
spring.shardingsphere.encrypt.tables.<table-name>.columns.<logic-column-name>.
encryptor= # 加密器名字
```

治理

配置项说明

```
# 省略数据源、数据分片、读写分离和数据脱敏配置

spring.shardingsphere.orchestration.name= # 治理实例名称
spring.shardingsphere.orchestration.overwrite= # 本地配置是否覆盖注册中心配置。如果可覆盖，每次启动都以本地配置为准
spring.shardingsphere.orchestration.registry.type= # 配置中心类型。如：zookeeper
spring.shardingsphere.orchestration.registry.server-lists= # 连接注册中心服务器的列表。包括 IP 地址和端口号。多个地址用逗号分隔。如：host1:2181,host2:2181
spring.shardingsphere.orchestration.registry.namespace= # 注册中心的命名空间
spring.shardingsphere.orchestration.registry.digest= # 连接注册中心的权限令牌。缺省为不需要权限验证
spring.shardingsphere.orchestration.registry.operation-timeout-milliseconds= # 操作超时的毫秒数，默认 500 毫秒
spring.shardingsphere.orchestration.registry.max-retries= # 连接失败后的最大重试次数，默认 3 次
spring.shardingsphere.orchestration.registry.retry-interval-milliseconds= # 重试间隔毫秒数，默认 500 毫秒
spring.shardingsphere.orchestration.registry.time-to-live-seconds= # 临时节点存活秒数，默认 60 秒
spring.shardingsphere.orchestration.registry.props= # 配置中心其它属性
```

ShardingSphere-3.x

数据分片

配置项说明

```
sharding.jdbc.datasource.names= # 数据源名称，多数据源以逗号分隔

sharding.jdbc.datasource.<data-source-name>.type= # 数据库连接池类名称
sharding.jdbc.datasource.<data-source-name>.driver-class-name= # 数据库驱动类名
sharding.jdbc.datasource.<data-source-name>.url= # 数据库 url 连接
sharding.jdbc.datasource.<data-source-name>.username= # 数据库用户名
sharding.jdbc.datasource.<data-source-name>.password= # 数据库密码
sharding.jdbc.datasource.<data-source-name>.xxx= # 数据库连接池的其它属性

sharding.jdbc.config.sharding.tables.<logic-table-name>.actual-data-nodes= # 由数据源名 + 表名组成，以小数点分隔。多个表以逗号分隔，支持 inline 表达式。缺省表示使用已知数据源与逻辑表名称生成数据节点。用于广播表（即每个库中都需要一个同样的表用于关联查询，多为字典表）或只分库不分表且所有库的表结构完全一致的情况

# 分库策略，缺省表示使用默认分库策略，以下的分片策略只能选其一
```

```
# 用于单分片键的标准分片场景
sharding.jdbc.config.sharding.tables.<logic-table-name>.database-strategy.standard.
sharding-column= # 分片列名称
sharding.jdbc.config.sharding.tables.<logic-table-name>.database-strategy.standard.
precise-algorithm-class-name= # 精确分片算法类名称，用于 = 和 IN。该类需实现
PreciseShardingAlgorithm 接口并提供无参数的构造器
sharding.jdbc.config.sharding.tables.<logic-table-name>.database-strategy.standard.
range-algorithm-class-name= # 范围分片算法类名称，用于 BETWEEN，可选。该类需实现
RangeShardingAlgorithm 接口并提供无参数的构造器

# 用于多分片键的复合分片场景
sharding.jdbc.config.sharding.tables.<logic-table-name>.database-strategy.complex.
sharding-columns= # 分片列名称，多个列以逗号分隔
sharding.jdbc.config.sharding.tables.<logic-table-name>.database-strategy.complex.
algorithm-class-name= # 复合分片算法类名称。该类需实现 ComplexKeysShardingAlgorithm 接口并
提供无参数的构造器

# 行表达式分片策略
sharding.jdbc.config.sharding.tables.<logic-table-name>.database-strategy.inline.
sharding-column= # 分片列名称
sharding.jdbc.config.sharding.tables.<logic-table-name>.database-strategy.inline.
algorithm-expression= # 分片算法行表达式，需符合 groovy 语法

# Hint 分片策略
sharding.jdbc.config.sharding.tables.<logic-table-name>.database-strategy.hint.
algorithm-class-name= # Hint 分片算法类名称。该类需实现 HintShardingAlgorithm 接口并提供无
参数的构造器

# 分表策略，同分库策略
sharding.jdbc.config.sharding.tables.<logic-table-name>.table-strategy.xxx= # 省略

sharding.jdbc.config.sharding.tables.<logic-table-name>.key-generator-column-name=
# 自增列名称，缺省表示不使用自增主键生成器
sharding.jdbc.config.sharding.tables.<logic-table-name>.key-generator-class-name= #
自增列值生成器类名称，缺省表示使用默认自增列值生成器。该类需提供无参数的构造器

sharding.jdbc.config.sharding.tables.<logic-table-name>.logic-index= # 逻辑索引名称，
对于分表的 Oracle/PostgreSQL 数据库中 DROP INDEX XXX 语句，需要通过配置逻辑索引名称定位所执行
SQL 的真实分表

sharding.jdbc.config.sharding.binding-tables[0]= # 绑定表规则列表
sharding.jdbc.config.sharding.binding-tables[1]= # 绑定表规则列表
sharding.jdbc.config.sharding.binding-tables[x]= # 绑定表规则列表

sharding.jdbc.config.sharding.broadcast-tables[0]= # 广播表规则列表
sharding.jdbc.config.sharding.broadcast-tables[1]= # 广播表规则列表
sharding.jdbc.config.sharding.broadcast-tables[x]= # 广播表规则列表
```

```
sharding.jdbc.config.sharding.default-data-source-name= # 未配置分片规则的表将通过默认数据源定位
sharding.jdbc.config.sharding.default-database-strategy.xxx= # 默认数据库分片策略，同分库策略
sharding.jdbc.config.sharding.default-table-strategy.xxx= # 默认表分片策略，同分表策略
sharding.jdbc.config.sharding.default-key-generator-class-name= # 默认自增列值生成器类名称，缺省使用 io.shardingsphere.core.keygen.DefaultKeyGenerator。该类需实现 KeyGenerator 接口并提供无参数的构造器

sharding.jdbc.config.sharding.master-slave-rules.<master-slave-data-source-name>.master-data-source-name= # 详见读写分离部分
sharding.jdbc.config.sharding.master-slave-rules.<master-slave-data-source-name>.slave-data-source-names[0]= # 详见读写分离部分
sharding.jdbc.config.sharding.master-slave-rules.<master-slave-data-source-name>.slave-data-source-names[1]= # 详见读写分离部分
sharding.jdbc.config.sharding.master-slave-rules.<master-slave-data-source-name>.slave-data-source-names[x]= # 详见读写分离部分
sharding.jdbc.config.sharding.master-slave-rules.<master-slave-data-source-name>.load-balance-algorithm-class-name= # 详见读写分离部分
sharding.jdbc.config.sharding.master-slave-rules.<master-slave-data-source-name>.load-balance-algorithm-type= # 详见读写分离部分
sharding.jdbc.config.config.map.key1= # 详见读写分离部分
sharding.jdbc.config.config.map.key2= # 详见读写分离部分
sharding.jdbc.config.config.map.keyx= # 详见读写分离部分

sharding.jdbc.config.props.sql.show= # 是否开启 SQL 显示，默认值：false
sharding.jdbc.config.props.executor.size= # 工作线程数量，默认值：CPU 核数

sharding.jdbc.config.config.map.key1= # 用户自定义配置
sharding.jdbc.config.config.map.key2= # 用户自定义配置
sharding.jdbc.config.config.map.keyx= # 用户自定义配置
```

读写分离

配置项说明

```
# 省略数据源配置，与数据分片一致

sharding.jdbc.config.sharding.master-slave-rules.<master-slave-data-source-name>.master-data-source-name= # 主库数据源名称
sharding.jdbc.config.sharding.master-slave-rules.<master-slave-data-source-name>.slave-data-source-names[0]= # 从库数据源名称列表
sharding.jdbc.config.sharding.master-slave-rules.<master-slave-data-source-name>.slave-data-source-names[1]= # 从库数据源名称列表
sharding.jdbc.config.sharding.master-slave-rules.<master-slave-data-source-name>.slave-data-source-names[x]= # 从库数据源名称列表
```

```
sharding.jdbc.config.sharding.master-slave-rules.<master-slave-data-source-name>.
load-balance-algorithm-class-name= # 从库负载均衡算法类名称。该类需实现
MasterSlaveLoadBalanceAlgorithm 接口且提供无参数构造器
sharding.jdbc.config.sharding.master-slave-rules.<master-slave-data-source-name>.
load-balance-algorithm-type= # 从库负载均衡算法类型，可选值：ROUND_ROBIN，RANDOM。若
`load-balance-algorithm-class-name` 存在则忽略该配置

sharding.jdbc.config.config.map.key1= # 用户自定义配置
sharding.jdbc.config.config.map.key2= # 用户自定义配置
sharding.jdbc.config.config.map.keyx= # 用户自定义配置

sharding.jdbc.config.props.sql.show= # 是否开启 SQL 显示，默认值：false
sharding.jdbc.config.props.executor.size= # 工作线程数量，默认值：CPU 核数
sharding.jdbc.config.props.check.table.metadata.enabled= # 是否在启动时检查分表元数据一
致性，默认值：false
```

治理

配置项说明

```
# 省略数据源、数据分片和读写分离配置

sharding.jdbc.config.sharding.orchestration.name= # 数据治理实例名称
sharding.jdbc.config.sharding.orchestration.overwrite= # 本地配置是否覆盖注册中心配置。如
果可覆盖，每次启动都以本地配置为准
sharding.jdbc.config.sharding.orchestration.registry.server-lists= # 连接注册中心服务
器的列表。包括 IP 地址和端口号。多个地址用逗号分隔。如：host1:2181,host2:2181
sharding.jdbc.config.sharding.orchestration.registry.namespace= # 注册中心的命名空间
sharding.jdbc.config.sharding.orchestration.registry.digest= # 连接注册中心的权限令牌。
缺省为不需要权限验证
sharding.jdbc.config.sharding.orchestration.registry.operation-timeout-
milliseconds= # 操作超时的毫秒数，默认 500 毫秒
sharding.jdbc.config.sharding.orchestration.registry.max-retries= # 连接失败后的最大重
试次数，默认 3 次
sharding.jdbc.config.sharding.orchestration.registry.retry-interval-milliseconds= #
重试间隔毫秒数，默认 500 毫秒
sharding.jdbc.config.sharding.orchestration.registry.time-to-live-seconds= # 临时节点
存活秒数，默认 60 秒
```

ShardingSphere-2.x

分库分表

配置项说明

```
# 忽略数据源配置

sharding.jdbc.config.sharding.default-data-source-name= # 未配置分片规则的表将通过默认数据源定位
sharding.jdbc.config.sharding.default-database-strategy.inline.sharding-column= # 分片列名称
sharding.jdbc.config.sharding.default-database-strategy.inline.algorithm-expression= # 分片算法行表达式，需符合 groovy 语法
sharding.jdbc.config.sharding.tables.t_order.actualDataNodes= # 由数据源名 + 表名组成，以小数点分隔。多个表以逗号分隔，支持 inline 表达式。缺省表示使用已知数据源与逻辑表名称生成数据节点，用于广播表（即每个库中都需要一个同样的表用于关联查询，多为字典表）或只分库不分表且所有库的表结构完全一致的情况
sharding.jdbc.config.sharding.tables.t_order.tableStrategy.inline.shardingColumn= # 分片列名称
sharding.jdbc.config.sharding.tables.t_order.tableStrategy.inline.algorithmInlineExpression= # 分片算法行表达式，需符合 groovy 语法
sharding.jdbc.config.sharding.tables.t_order.keyGeneratorColumnName= # 自增列名称，缺省表示不使用自增主键生成器

sharding.jdbc.config.sharding.tables.<logic-table-name>.key-generator-column-name= # 自增列名称，缺省表示不使用自增主键生成器
sharding.jdbc.config.sharding.tables.<logic-table-name>.key-generator-class-name= # 默认自增列值生成器类型
```

读写分离

配置项说明

```
# 忽略数据源配置

sharding.jdbc.config.masterslave.load-balance-algorithm-type= # 从库负载均衡算法类型，可选值：ROUND_ROBIN，RANDOM。若 `load-balance-algorithm-class-name` 存在则忽略该配置
sharding.jdbc.config.masterslave.name= # 主节点名称
sharding.jdbc.config.masterslave.master-data-source-name= # 主数据源的名称
sharding.jdbc.config.masterslave.slave-data-source-names= # 从数据源的名称
```

编排治理

配置项说明

```
# 忽略数据源配置

sharding.jdbc.config.orchestration.name= # 数据治理实例名称
sharding.jdbc.config.orchestration.overwrite= # 本地配置是否覆盖注册中心配置。如果可覆盖，每次启动都以本地配置为准

sharding.jdbc.config.sharding.orchestration.name= # 数据治理实例名称
sharding.jdbc.config.sharding.orchestration.overwrite= # 本地配置是否覆盖注册中心配置。如果可覆盖，每次启动都以本地配置为准
sharding.jdbc.config.sharding.orchestration.registry.server-lists= # 连接注册中心服务器的列表。包括 IP 地址和端口号。多个地址用逗号分隔。如: host1:2181,host2:2181
sharding.jdbc.config.sharding.orchestration.registry.namespace= # 注册中心的命名空间
sharding.jdbc.config.sharding.orchestration.registry.digest= # 连接注册中心的权限令牌。缺省为不需要权限验证
sharding.jdbc.config.sharding.orchestration.registry.operation-timeout-milliseconds= # 操作超时的毫秒数，默认 500 毫秒
sharding.jdbc.config.sharding.orchestration.registry.max-retries= # 连接失败后的最大重试次数，默认 3 次
sharding.jdbc.config.sharding.orchestration.registry.retry-interval-milliseconds= # 重试间隔毫秒数，默认 500 毫秒
sharding.jdbc.config.sharding.orchestration.registry.time-to-live-seconds= # 临时节点存活秒数，默认 60 秒

# Zookeeper 配置
sharding.jdbc.config.orchestration.zookeeper.namespace= # Zookeeper 注册中心的命名空间
sharding.jdbc.config.orchestration.zookeeper.server-lists= # Zookeeper 注册中心服务器的列表。包括 IP 地址和端口号。多个地址用逗号分隔。如: host1:2181,host2:2181

# Etcd 配置
sharding.jdbc.config.orchestration.etcd.server-lists= # Etcd 注册中心服务器的列表。包括 IP 地址和端口号。多个地址用逗号分隔。如: host1:2181,host2:2181
```

9.9.2 ShardingSphere-Proxy

5.0.0-beta

数据源配置项说明

```
schemaName: # 逻辑数据源名称

dataSources: # 数据源配置，可配置多个 <data-source-name>
  <data-source-name>: # 与 ShardingSphere-JDBC 配置不同，无需配置数据库连接池
```

```
    url: # 数据库 URL 连接
    username: # 数据库用户名
    password: # 数据库密码
    connectionTimeoutMilliseconds: # 连接超时毫秒数
    idleTimeoutMilliseconds: # 空闲连接回收超时毫秒数
    maxLifetimeMilliseconds: # 连接最大存活时间毫秒数
    maxPoolSize: 50 # 最大连接数
    minPoolSize: 1  # 最小连接数

rules: # 与 ShardingSphere-JDBC 配置一致
  # ...
```

权限配置

用于执行登录 Sharding Proxy 的权限验证。配置用户名、密码、可访问的数据库后，必须使用正确的用户名、密码才可登录。

```
rules:
  - !AUTHORITY
    users:
      - root@localhost:root  # <username>@<hostname>:<password>, hostname 为 % 或空字符串，则代表不限制 host。
      - sharding@:sharding
    provider:
      type: NATIVE  # 必须显式指定
```

hostname 为% 或空字符串，则代表不限制 host。

provider 的 type 必须显式指定，具体实现可以参考 5.11 Proxy

Proxy 属性

```
props:
  sql-show: # 是否在日志中打印 SQL。打印 SQL 可以帮助开发者快速定位系统问题。日志内容包含：逻辑 SQL，真实 SQL 和 SQL 解析结果。如果开启配置，日志将使用 Topic ShardingSphere-SQL，日志级别是 INFO。
  sql-simple: # 是否在日志中打印简单风格的 SQL。
  executor-size: # 用于设置任务处理线程池的大小。每个 ShardingSphereDataSource 使用一个独立的线程池，同一个 JVM 的不同数据源不共享线程池。
  max-connections-size-per-query: # 一次查询请求在每个数据库实例中所能使用的最大连接数。
  check-table-metadata-enabled: # 是否在程序启动和更新时检查分片元数据的结构一致性。
  proxy-frontend-flush-threshold: # 在 ShardingSphere-Proxy 中设置传输数据条数的 IO 刷新阈值。
  proxy-transaction-type: # ShardingSphere-Proxy 中使用的默认事务类型。包括：LOCAL、XA 和 BASE。
  proxy-opentracing-enabled: # 是否允许在 ShardingSphere-Proxy 中使用 OpenTracing。
```

```
  proxy-hint-enabled: # 是否允许在 ShardingSphere-Proxy 中使用 Hint。使用 Hint 会将
Proxy 的线程处理模型由 IO 多路复用变更为每个请求一个独立的线程，会降低 Proxy 的吞吐量。
  xa-transaction-manager-type: # XA 事务管理器类型。列如：Atomikos，Narayana，Bitronix。
```

5.0.0-alpha

数据源配置项说明

```
schemaName: # 逻辑数据源名称

dataSourceCommon:
  username: # 数据库用户名
  password: # 数据库密码
  connectionTimeoutMilliseconds: # 连接超时毫秒数
  idleTimeoutMilliseconds: # 空闲连接回收超时毫秒数
  maxLifetimeMilliseconds: # 连接最大存活时间毫秒数
  maxPoolSize: 50 # 最大连接数
  minPoolSize: 1  # 最小连接数

dataSources: # 数据源配置，可配置多个 <data-source-name>
  <data-source-name>: # 与 ShardingSphere-JDBC 配置不同，无需配置数据库连接池
    url: # 数据库 URL 连接
rules: # 与 ShardingSphere-JDBC 配置一致
  # ...
```

覆盖 dataSourceCommon 说明

上面配置了每个库的公共数据源配置，如果你想覆盖 dataSourceCommon 属性，请在每个数据源单独配置。

```
dataSources: # 数据源配置，可配置多个 <data-source-name>
  <data-source-name>: # 与 ShardingSphere-JDBC 配置不同，无需配置数据库连接池
    url: # 数据库 URL 连接
    username: # 数据库用户名，覆盖 dataSourceCommon 配置
    password: # 数据库密码，覆盖 dataSourceCommon 配置
    connectionTimeoutMilliseconds: # 连接超时毫秒数，覆盖 dataSourceCommon 配置
    idleTimeoutMilliseconds: # 空闲连接回收超时毫秒数，覆盖 dataSourceCommon 配置
    maxLifetimeMilliseconds: # 连接最大存活时间毫秒数，覆盖 dataSourceCommon 配置
    maxPoolSize: # 最大连接数，覆盖 dataSourceCommon 配置
```

权限配置

用于执行登录 Sharding Proxy 的权限验证。配置用户名、密码、可访问的数据库后，必须使用正确的用户名、密码才可登录。

```
authentication:
  users:
    root: # 自定义用户名
      password: root # 自定义用户名
    sharding: # 自定义用户名
      password: sharding # 自定义用户名
      authorizedSchemas: sharding_db, replica_query_db # 该用户授权可访问的数据库，多个用逗号分隔。缺省将拥有 root 权限，可访问全部数据库。
```

Proxy 属性

```
props:
  sql-show: # 是否在日志中打印 SQL。打印 SQL 可以帮助开发者快速定位系统问题。日志内容包含：逻辑 SQL，真实 SQL 和 SQL 解析结果。如果开启配置，日志将使用 Topic ShardingSphere-SQL，日志级别是 INFO。
  sql-simple: # 是否在日志中打印简单风格的 SQL。
  acceptor-size: # 用于设置接收 TCP 请求线程池的大小。
  executor-size: # 用于设置任务处理线程池的大小。每个 ShardingSphereDataSource 使用一个独立的线程池，同一个 JVM 的不同数据源不共享线程池。
  max-connections-size-per-query: # 一次查询请求在每个数据库实例中所能使用的最大连接数。
  check-table-metadata-enabled: # 是否在程序启动和更新时检查分片元数据的结构一致性。
  query-with-cipher-column: # 是否使用加密列进行查询。在有原文列的情况下，可以使用原文列进行查询。
  proxy-frontend-flush-threshold: # 在 ShardingSphere-Proxy 中设置传输数据条数的 IO 刷新阈值。
  proxy-transaction-type: # ShardingSphere-Proxy 中使用的默认事务类型。包括：LOCAL、XA 和 BASE。
  proxy-opentracing-enabled: # 是否允许在 ShardingSphere-Proxy 中使用 OpenTracing。
  proxy-hint-enabled: # 是否允许在 ShardingSphere-Proxy 中使用 Hint。使用 Hint 会将 Proxy 的线程处理模型由 IO 多路复用变更为每个请求一个独立的线程，会降低 Proxy 的吞吐量。
```

ShardingSphere-4.x

数据源与分片配置项说明

数据分片

```
schemaName: # 逻辑数据源名称

dataSources: # 数据源配置，可配置多个 data_source_name
```

```
  <data_source_name>: # 与 Sharding-JDBC 配置不同，无需配置数据库连接池
    url: # 数据库 url 连接
    username: # 数据库用户名
    password: # 数据库密码
    connectionTimeoutMilliseconds: 30000 # 连接超时毫秒数
    idleTimeoutMilliseconds: 60000 # 空闲连接回收超时毫秒数
    maxLifetimeMilliseconds: 1800000 # 连接最大存活时间毫秒数
    maxPoolSize: 65 # 最大连接数

shardingRule: # 省略数据分片配置，与 Sharding-JDBC 配置一致
```

读写分离

```
schemaName: # 逻辑数据源名称

dataSources: # 省略数据源配置，与数据分片一致

masterSlaveRule: # 省略读写分离配置，与 Sharding-JDBC 配置一致
```

数据脱敏

```
dataSource: # 省略数据源配置

encryptRule:
  encryptors:
    <encryptor-name>:
      type: # 加解密器类型，可自定义或选择内置类型：MD5/AES
      props: # 属性配置，注意：使用 AES 加密器，需要配置 AES 加密器的 KEY 属性：aes.key.value
        aes.key.value:
  tables:
    <table-name>:
      columns:
        <logic-column-name>:
          plainColumn: # 存储明文的字段
          cipherColumn: # 存储密文的字段
          assistedQueryColumn: # 辅助查询字段，针对 ShardingQueryAssistedEncryptor 类型
的加解密器进行辅助查询
          encryptor: # 加密器名字
props:
  query.with.cipher.column: true # 是否使用密文列查询
```

全局配置项说明

治理

与 Sharding-JDBC 配置一致。

Proxy 属性

```
# 省略与 Sharding-JDBC 一致的配置属性

props:
  acceptor.size: # 用于设置接收客户端请求的工作线程个数，默认为 CPU 核数 * 2
  proxy.transaction.type: # 默认为 LOCAL 事务，允许 LOCAL，XA，BASE 三个值，XA 采用 Atomikos 作为事务管理器，BASE 类型需要拷贝实现 ShardingTransactionManager 的接口的 jar 包至 lib 目录中
  proxy.opentracing.enabled: # 是否开启链路追踪功能，默认为不开启。详情请参见 [链路追踪](https://shardingsphere.apache.org/document/current/cn/features/orchestration/apm/)
  check.table.metadata.enabled: # 是否在启动时检查分表元数据一致性，默认值：false
  proxy.frontend.flush.threshold: # 对于单个大查询，每多少个网络包返回一次
```

权限验证

用于执行登录 Sharding Proxy 的权限验证。配置用户名、密码、可访问的数据库后，必须使用正确的用户名、密码才可登录 Proxy。

```
authentication:
  users:
    root: # 自定义用户名
      password: root # 自定义用户名
    sharding: # 自定义用户名
      password: sharding # 自定义用户名
      authorizedSchemas: sharding_db, masterslave_db # 该用户授权可访问的数据库，多个用逗号分隔。缺省将拥有 root 权限，可访问全部数据库。
```

ShardingSphere-3.x

数据源与分片配置项说明

数据分片

```
schemaName: # 逻辑数据源名称

dataSources: # 数据源配置，可配置多个 data_source_name
```

```
  <data_source_name>: # 与 Sharding-JDBC 配置不同，无需配置数据库连接池
    url: # 数据库 url 连接
    username: # 数据库用户名
    password: # 数据库密码
    autoCommit: true # hikari 连接池默认配置
    connectionTimeout: 30000 # hikari 连接池默认配置
    idleTimeout: 60000 # hikari 连接池默认配置
    maxLifetime: 1800000 # hikari 连接池默认配置
    maximumPoolSize: 65 # hikari 连接池默认配置

shardingRule: # 省略数据分片配置，与 Sharding-JDBC 配置一致
```

读写分离

```
schemaName: # 逻辑数据源名称

dataSources: # 省略数据源配置，与数据分片一致

masterSlaveRule: # 省略读写分离配置，与 Sharding-JDBC 配置一致
```

全局配置项说明

数据治理

与 Sharding-JDBC 配置一致。

Proxy 属性

```
# 省略与 Sharding-JDBC 一致的配置属性

props:
  acceptor.size: # 用于设置接收客户端请求的工作线程个数，默认为 CPU 核数 * 2
  proxy.transaction.enabled: # 是否开启事务，目前仅支持 XA 事务，默认为不开启
  proxy.opentracing.enabled: # 是否开启链路追踪功能，默认为不开启。详情请参见 [链路追
踪](https://shardingsphere.apache.org/document/current/cn/features/orchestration/
apm/)
  check.table.metadata.enabled: # 是否在启动时检查分表元数据一致性，默认值: false
```

权限验证

用于执行登录 Sharding Proxy 的权限验证。配置用户名、密码后，必须使用正确的用户名、密码才可登录 Proxy。

```
authentication:
  username: root
  password:
```

10 下载

10.1 最新版本

Apache ShardingSphere 的发布版包括源码包及其对应的二进制包。由于下载内容分布在镜像服务器上，所以下载后应该进行 GPG 或 SHA-512 校验，以此来保证内容没有被篡改。

10.1.1 Apache ShardingSphere - 版本: 5.1.0 (发布日期: Feb 16th, 2022)

- 源码: [SRC] [ASC] [SHA512]
- ShardingSphere-JDBC 二进制包: [TAR] [ASC] [SHA512]
- ShardingSphere-Proxy 二进制包: [TAR] [ASC] [SHA512]
- ShardingSphere-Agent 二进制包: [TAR] [ASC] [SHA512]

10.2 全部版本

全部版本请到 Archive repository 查看。全部孵化器版本请到 Archive incubator repository 查看。

10.3 校验版本

PGP 签名文件

使用 PGP 或 SHA 签名验证下载文件的完整性至关重要。可以使用 GPG 或 PGP 验证 PGP 签名。请下载 KEYS 以及发布的 asc 签名文件。建议从主发布目录而不是镜像中获取这些文件。

```
gpg -i KEYS
```

或者

```
pgpk -a KEYS
```

或者

```
pgp -ka KEYS
```

要验证二进制文件或源代码，您可以从主发布目录下载相关的 asc 文件，并按照以下指南进行操作。

```
gpg --verify apache-shardingsphere-********.asc apache-shardingsphere-*********
```

或者

```
pgpv apache-shardingsphere-********.asc
```

或者

```
pgp apache-shardingsphere-********.asc
```