Apache ShardingSphere document

Apache ShardingSphere

2020 年 08 月 26 日

Contents

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

2 功能列表 6
2.1 数据分片 6
2.2 分布式事务 6
2.3 数据库治理 6

3 快速入门 7
3.1 ShardingSphere-JDBC 7
3.1.1 1. 引入 maven 依赖 7
3.1.2 2. 规则配置 7
3.1.3 3. 创建数据源 7
3.2 ShardingSphere-Proxy 8
3.2.1 1. 规则配置 8
3.2.2 2. 引入依赖 8
3.2.3 3. 启动服务 8
3.2.4 4. 使用 ShardingSphere-Proxy 8
3.3 ShardingSphere-Scaling(Alpha) 8
3.3.1 1. 规则配置 8
3.3.2 2. 引入依赖 9
3.3.3 3. 启动服务 9
3.3.4 4. 创建迁移任务 9

4 概念 & 功能 10
4.1 数据分片 10
4.1.1 背景 10
垂直分片 11
水平分片 11

4.1.2 挑战 12
4.1.3 目标 13
4.1.4 核心概念 13
导览 13
SQL 13
分片 14
配置 16
行表达式 17
分布式主键 20
强制分片路由 22
4.1.5 内核剖析 23
SQL 解析 24
执行器优化 24
SQL 路由 24
SQL 改写 24
SQL 执行 24
结果归并 24
解析引擎 24
路由引擎 26
改写引擎 31
执行引擎 36
归并引擎 42
4.1.6 使用规范 46
背景 46
SQL 47
分页 49
解析器 51
4.2 分布式事务 52
4.2.1 背景 52
本地事务 52
两阶段提交 52
柔性事务 53
4.2.2 挑战 53
4.2.3 目标 53
4.2.4 核心概念 54
导览 54
XA 两阶段事务 54
Seata 柔性事务 55
4.2.5 实现原理 56
导览 56
XA 两阶段事务 56
Seata 柔性事务 57
4.2.6 使用规范 59
背景 59
本地事务 59

XA 两阶段事务 59
Seata 柔性事务 60
4.3 读写分离 60
4.3.1 背景 60
4.3.2 挑战 60
4.3.3 目标 62
4.3.4 核心概念 62
主库 62
从库 62
主从同步 62
负载均衡策略 62
4.3.5 使用规范 62
支持项 62
不支持项 62
4.4 分布式治理 63
4.4.1 背景 63
4.4.2 挑战 63
4.4.3 目标 63
4.4.4 治理 64
导览 64
配置中心 64
注册中心 66
元数据中心 68
第三方组件依赖 70
4.4.5 可观察性 70
导览 70
应用性能监控集成 71
度量指标监控 75
4.5 弹性伸缩 77
4.5.1 背景 77
4.5.2 简介 77
4.5.3 挑战 77
4.5.4 目标 77
4.5.5 状态 79
4.5.6 核心概念 79
弹性伸缩作业 79
数据节点 79
存量数据 79
增量数据 79
4.5.7 实现原理 80
原理说明 80
执行阶段说明 81
4.5.8 使用规范 82
支持项 82
不支持项 82

4.6 数据加密 82
4.6.1 背景 82
4.6.2 挑战 83
4.6.3 目标 83
4.6.4 核心概念 83
4.6.5 实现原理 83
处理流程详解 83
解决方案详解 87
中间件加密服务优势 91
加密算法解析 92
4.6.6 使用规范 93
支持项 93
不支持项 93
4.7 影子库压测 93
4.7.1 背景 93
4.7.2 挑战 93
4.7.3 目标 94
4.7.4 核心概念 94
影子字段 94
生产数据库 94
影子数据库 94
4.7.5 实现原理 94
整体架构 94
影子规则 94
处理过程 94
4.8 多数据副本 97
4.8.1 导览 97
4.9 可插拔架构 97
4.9.1 背景 97
4.9.2 挑战 97
4.9.3 目标 97
4.10 测试引擎 97
4.10.1 SQL 测试用例 98
目标 98
4.10.2 整合测试引擎 98
流程 98
注意事项 103
4.10.3 SQL 解析测试引擎 103
数据准备 103
4.10.4 SQL 改写测试引擎 105
目标 105
4.10.5 性能测试 106
目标 106
测试场景 106
测试环境搭建 107

测试结果验证 113

5 用户手册 116
5.1 ShardingSphere-JDBC 116
5.1.1 简介 116
5.1.2 对比 116
5.1.3 使用手册 118
数据分片 118
分布式事务 129
分布式治理 134
5.1.4 配置手册 140
Java API 140
YAML 配置 147
Spring Boot Start 配置 154
Spring 命名空间配置 159
内置算法 166
属性配置 169
5.1.5 不支持项 169
DataSource 接口 169
Connection 接口 170
Statement 和 PreparedStatement 接口 170
ResultSet 接口 170
JDBC 4.1 170
5.2 ShardingSphere-Proxy 170
5.2.1 简介 170
5.2.2 对比 171
5.2.3 使用手册 171
Proxy 启动 172
分布式治理 173
分布式事务 173
SCTL 173
5.2.4 配置手册 174
数据源配置 174
权限配置 175
属性配置 175
YAML 语法说明 175
5.2.5 Docker 镜像 175
拉取官方 Docker 镜像 175
手动构建 Docker 镜像（可选） 176
配置 ShardingSphere-Proxy 176
运行 Docker 176
访问 ShardingSphere-Proxy 176
FAQ 177
5.3 ShardingSphere-Sidecar 177
5.3.1 简介 177

5.3.2 对比 . . . 177
5.4 ShardingSphere-Scaling . . . 179
5.4.1 简介 . . . 179
5.4.2 运行部署 . . . 179
部署启动 . . . 179
结束 ShardingSphere-Scaling . . . 180
应用配置项 . . . 180
5.4.3 使用手册 . . . 180
使用手册 . . . 180
通过 UI 界面来操作 . . . 184
5.5 ShardingSphere-UI . . . 185
5.5.1 简介 . . . 185
5.5.2 使用手册 . . . 185
导览 . . . 185
部署运行 . . . 185
配置中心 . . . 187
注册中心 . . . 187

6 开发者手册 188
6.1 SQL 解析 . . . 188
6.1.1 SQLParserConfiguration . . . 188
6.1.2 ParsingHook . . . 189
6.2 配置 . . . 189
6.2.1 ShardingSphereRuleBuilder . . . 189
6.2.2 YamlRuleConfigurationSwapper . . . 189
6.2.3 ShardingSphereYamlConstruct . . . 190
6.3 内核 . . . 190
6.3.1 DatabaseType . . . 190
6.3.2 RuleMetaDataLoader . . . 190
6.3.3 RuleMetaDataDecorator . . . 191
6.3.4 RouteDecorator . . . 191
6.3.5 SQLRewriteContextDecorator . . . 191
6.3.6 ExecuteGroupDecorator . . . 191
6.3.7 SQLExecutionHook . . . 192
6.3.8 ResultProcessEngine . . . 192
6.4 数据分片 . . . 192
6.4.1 ShardingAlgorithm . . . 192
6.4.2 KeyGenerateAlgorithm . . . 193
6.4.3 TimeService . . . 193
6.4.4 DatabaseSQLEntry . . . 193
6.5 读写分离 . . . 193
6.5.1 MasterSlaveLoadBalanceAlgorithm . . . 193
6.6 数据加密 . . . 194
6.6.1 EncryptAlgorithm . . . 194
6.6.2 QueryAssistedEncryptAlgorithm . . . 194

6.7 分布式事务 . . . 194
6.7.1 ShardingTransactionManager . . . 194
6.7.2 XATransactionManager . . . 195
6.7.3 XADataSourceDefinition . . . 195
6.7.4 DataSourcePropertyProvider . . . 195
6.8 分布式治理 . . . 196
6.8.1 ConfigurationRepository . . . 196
6.8.2 RegistryRepository . . . 196
6.8.3 RootInvokeHook . . . 196
6.8.4 MetricsTrackerManager . . . 197
6.9 弹性伸缩 . . . 197
6.9.1 ScalingEntry . . . 197
6.10 Proxy . . . 197
6.10.1 DatabaseProtocolFrontendEngine . . . 197
6.10.2 JDBCDriverURLRecognizer . . . 198

7 下载 199
7.1 最新版本 . . . 199
7.1.1 Apache ShardingSphere - 版本: 4.1.1 (发布日期: Jun 5, 2020) . . . 199
7.1.2 ShardingSphere UI - 版本: 4.1.1 (发布日期: Jun 9, 2020) . . . 199
7.2 全部版本 . . . 199
7.3 校验版本 . . . 200

8 FAQ 201
8.1 如果 SQL 在 ShardingSphere 中执行不正确，该如何调试? . . . 201
8.2 阅读源码时为什么会出现编译错误? . . . 201
8.3 使用 Spring 命名空间时找不到 xsd? . . . 201
8.4 Cloud not resolve placeholder …in string value …异常的解决方法? . . . 202
8.5 inline 表达式返回结果为何出现浮点数? . . . 202
8.6 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? . . . 202
8.7 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? . . . 202
8.8 指定了泛型为 Long 的 `SingleKeyTableShardingAlgorithm`，遇到 `ClassCastException: Integer can not cast to Long`? . . . 203
8.9 使用 SQLSever 和 PostgreSQL 时，聚合列不加别名会抛异常? . . . 203
8.10 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”? . . . 203
8.11 使用 `Proxool` 时分库结果不正确? . . . 204
8.12 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? 205
8.13 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? . . . 205
8.14 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? . . . 205
8.15 Type is required 异常的解决方法? . . . 206
8.16 为什么我实现了 `ShardingKeyGenerator` 接口，也配置了 Type，但是自定义的分布式主键依然不生效? . . . 206

8.17 JPA 和数据加密无法一起使用，如何解决？ 206
8.18 服务启动时如何加快 `metadata` 加载速度？ 207
8.19 如何在 inline 分表策略时，允许执行范围查询操作（BETWEEN AND、>、<、>=、<=）？ 207
8.20 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？ 207
8.21 在使用 sharing-proxy 的时候，如何动态在 ShardingSphere-UI 上添加新的 logic schema？ 207
8.22 在使用 ShardingSphere-Proxy 时，怎么使用合适的工具连接到 ShardingSphere-Proxy？ 208
8.23 引入 `shardingsphere-transaction-xa-core` 后，如何避免 spring-boot 自动加载默认的 JtaTransactionManager？ 208

Apache ShardingSphere 是一套开源的分布式数据库中间件解决方案组成的生态圈，它由 JDBC、Proxy 和 Sidecar（规划中）这 3 款相互独立，却又能够混合部署配合使用的产品组成。它们均提供标准化的数据分片、分布式事务和数据库治理功能，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。

Apache ShardingSphere 定位为关系型数据库中间件，旨在充分合理地在分布式的场景下利用关系型数据库的计算和存储能力，而并非实现一个全新的关系型数据库。它通过关注不变，进而抓住事物本质。关系型数据库当今依然占有巨大市场，是各个公司核心业务的基石，未来也难于撼动，我们目前阶段更加关注在原有基础上的增量，而非颠覆。

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、多数据副本、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。开发者能够像使用积木一样定制属于自己的独特系统。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。

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: ShardingSphere Scope

1 简介

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, Druid, HikariCP 等。
- 支持任意实现 JDBC 规范的数据库，目前支持 MySQL，Oracle，SQLServer，PostgreSQL 以及任何遵循 SQL92 标准的数据库。

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 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。

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 1: ShardingSphere-JDBC Architecture

图 2: ShardingSphere-Proxy Architecture

图 3: ShardingSphere-Sidecar Architecture

	ShardingSphere-JDBC	*ShardingSphere-Proxy*	*ShardingSphere-Sidecar*
数据库	任意	MySQL/PostgreSQL	MySQL/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，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，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与当前业务的最佳系统架构。

图 4: ShardingSphere Hybrid Architecture

2 功能列表

2.1 数据分片

- 分库 & 分表
- 读写分离
- 分片策略定制化
- 无中心化分布式主键

2.2 分布式事务

- 标准化事务接口
- XA 强一致事务
- 柔性事务

2.3 数据库治理

- 分布式治理
- 弹性伸缩
- 可视化链路追踪
- 数据加密

3 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。

3.1 ShardingSphere-JDBC

3.1.1 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。

3.1.2 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。详情请参见配置手册。

3.1.3 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, MyBatis 等 ORM 类库。

```
DataSource dataSource = ShardingSphereDataSourceFactory.createDataSource(dataSourceMap,
 configurations, properties);
```

3.2 ShardingSphere-Proxy

3.2.1 1. 规则配置

编辑`%SHARDINGSPHERE_PROXY_HOME%/conf/config-xxx.yaml`。详情请参见配置手册。

编辑`%SHARDINGSPHERE_PROXY_HOME%/conf/server.yaml`。详情请参见配置手册。

3.2.2 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,需要下载 MySQL Connector/J,解压缩后,将`mysql-connector-java-5.1.47.jar`拷贝到`%SHARDINGSPHERE_PROXY_HOME%/lib`目录。

3.2.3 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}
```

3.2.4 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}
```

3.3 ShardingSphere-Scaling(Alpha)

3.3.1 1. 规则配置

编辑`%SHARDINGSPHERE_SCALING_HOME%/conf/server.yaml`。详情请参见使用手册。

3.3.2 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,需要下载 MySQL Connector/J,解压缩后,将 `mysql-connector-java-5.1.47.jar` 拷贝到 `%SHARDINGSPHERE_SCALING_HOME%/lib` 目录。

3.3.3 3. 启动服务

```
sh %SHARDINGSPHERE_SCALING_HOME%/bin/start.sh
```

3.3.4 4. 创建迁移任务

通过相应的 HTTP 接口管理迁移任务。

创建迁移任务:

```
curl -X POST \
  http://localhost:8888/shardingscaling/job/start \
  -H 'content-type: application/json' \
  -d '{
   "ruleConfiguration": {
      "sourceDatasource": "ds_0: !!org.apache.shardingsphere.orchestration.core.
common.yaml.config.YamlDataSourceConfiguration\n  dataSourceClassName: com.zaxxer.
hikari.HikariDataSource\n  props:\n    jdbcUrl: jdbc:mysql://127.0.0.1:3306/test?
serverTimezone=UTC&useSSL=false\n    username: root\n    password: '\''123456'\
'keyGenerateStrategy
```

更多管理接口详情请参见使用手册。

4 概念 & 功能

本章节阐述 Apache ShardingSphere 相关的概念与功能，更多使用细节请阅读用户手册。

4.1 数据分片

4.1.1 背景

传统的将数据集中存储至单一数据节点的解决方案，在性能、可用性和运维成本这三方面已经难于满足互联网的海量数据场景。

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，服务化的无状态型，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。

图 1: 垂直分片

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。

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。

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，是分库分表的标准解决方案。

图 2: 水平分片

4.1.2 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。

面对如此散乱的分库分表之后的数据，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分表中获取。

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。

4.1.3 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。

4.1.4 核心概念

导览

本小节主要介绍数据分片的核心概念，主要包括：

- SQL 核心概念
- 分片核心概念
- 配置核心概念

SQL

逻辑表

水平拆分的数据库（表）的相同逻辑和数据结构表的总称。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

在分片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`。

数据节点

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

绑定表

指分片规则一致的主表和子表。例如：`t_order` 表和 `t_order_item` 表，均按照 `order_id` 分片，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。故绑定表之间的分区键要完全相同。

广播表

指所有的分片数据源中都存在的表，表结构和表中的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。

分片

分片键

用于分片的数据库字段，是将数据库（表）水平拆分的关键字段。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

通过分片算法将数据分片，支持通过 `=`、`>=`、`<=`、`>`、`<`、`BETWEEN` 和 `IN` 分片。分片算法需要应用方开发者自行实现，可实现的灵活度非常高。

目前提供 4 种分片算法。由于分片算法和业务实现紧密相关，因此并未提供内置分片算法，而是通过分片策略将各种场景提炼出来，提供更高层级的抽象，并提供接口让应用开发者自行实现分片算法。

- 精确分片算法

对应 PreciseShardingAlgorithm，用于处理使用单一键作为分片键的 = 与 `IN` 进行分片的场景。需要配合 StandardShardingStrategy 使用。

- 范围分片算法

对应 RangeShardingAlgorithm，用于处理使用单一键作为分片键的 `BETWEEN AND`、>、<、>=、<= 进行分片的场景。需要配合 StandardShardingStrategy 使用。

- 复合分片算法

对应 ComplexKeysShardingAlgorithm，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。需要配合 ComplexShardingStrategy 使用。

- Hint 分片算法

对应 HintShardingAlgorithm，用于处理使用 `Hint` 行分片的场景。需要配合 HintShardingStrategy 使用。

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。目前提供 5 种分片策略。

- 标准分片策略

对应 StandardShardingStrategy。提供对 SQ L 语句中的 =, >, <, >=, <=, `IN` 和 `BETWEEN AND` 的分片操作支持。StandardShardingStrategy 只支持单分片键，提供 PreciseShardingAlgorithm 和 RangeShardingAlgorithm 两个分片算法。PreciseShardingAlgorithm 是必选的，用于处理 = 和 `IN` 的分片。RangeShardingAlgorithm 是可选的，用于处理 `BETWEEN AND`, >, <, >=, <= 分片，如果不配置 RangeShardingAlgorithm，SQL 中的 `BETWEEN AND` 将按照全库路由处理。

- 复合分片策略

对应 ComplexShardingStrategy。复合分片策略。提供对 SQL 语句中的 =, >, <, >=, <=, `IN` 和 `BETWEEN AND` 的分片操作支持。ComplexShardingStrategy 支持多分片键，由于多分片键之间的关系复杂，因此并未进行过多的封装，而是直接将分片键值组合以及分片操作符透传至分片算法，完全由应用开发者实现，提供最大的灵活度。

- 行表达式分片策略

对应 InlineShardingStrategy。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。

- Hint 分片策略

对应 HintShardingStrategy。通过 Hint 指定分片值而非从 SQL 中提取分片值的方式进行分片的策略。

- 不分片策略

对应 NoneShardingStrategy。不分片的策略。

SQL Hint

对于分片字段非 SQL 决定，而由其他外置条件决定的场景，可使用 SQL Hint 灵活的注入分片字段。例：内部系统，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。

配置

分片规则

分片规则配置的总入口。包含数据源配置、表配置、绑定表配置以及读写分离配置等。

数据源配置

真实数据源列表。

表配置

逻辑表名称、数据节点与分表规则的配置。

数据节点配置

用于配置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
```

那么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
```

(下页继续)

(续上页)

```
db1
  ├── t_order2
  ├── t_order3
  └── t_order4
```

那么数据节点的配置如下:

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

分片策略配置

对于分片策略存有数据源分片策略和表分片策略两种维度。

- 数据源分片策略

对应于 DatabaseShardingStrategy。用于配置数据被分配的目标数据源。

- 表分片策略

对应于 TableShardingStrategy。用于配置数据被分配的目标表，该目标表存在与该数据的目标数据源内。故表分片策略是依赖与数据源分片策略的结果的。

两种策略的 API 完全相同。

自增主键生成策略

通过在客户端生成自增主键替换以数据库原生自增主键的方式，做到分布式主键无重复。

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
```

(下页继续)

(续上页)

```
  ├── 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}
```

配置分片算法

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 还计划通过 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。

4.1.5 内核剖析

ShardingSphere 的 3 个产品的数据分片主要流程是完全一致的。核心由 `SQL 解析 => 执行器优化 => SQL 路由 => SQL 改写 => SQL 执行 => 结果归并`的流程组成。

图 4: 分片架构图

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。

执行器优化

合并和优化分片条件，如 OR 等。

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。

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。

SQL 执行

通过多线程执行器异步执行。

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。

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 SQL 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
```

图 5: SQL 抽象语法树

解析之后的为抽象语法树见下图。

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，通过对抽象语法树的遍历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

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 解析器则从 3.0.x 版本开始，ShardingSphere 尝试使用 ANTLR 作为 SQL 解析的引擎，并计划根据 `DDL -> TCL -> DAL -> DCL -> DML ->DQL` 这个顺序，依次替换原有的解析引擎，目前仍

处于替换迭代中。使用 ANTLR 的原因是希望 ShardingSphere 的解析引擎能够更好的对 SQL 进行兼容。对于复杂的表达式、递归、子查询等语句，虽然 ShardingSphere 的分片核心并不关注，但是会影响对于 SQL 理解的友好度。经过实例测试，ANTLR 解析 SQL 的性能比自研的 SQL 解析引擎慢 3-10 倍左右。为了弥补这一差距，ShardingSphere 将使用 `PreparedStatement` 的 SQL 解析的语法树放入缓存。因此建议采用 `PreparedStatement` 这种 SQL 预编译的方式提升性能。

第三代 SQL 解析引擎的整体结构划分如下图所示。

图 6: 解析引擎结构

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。

路由引擎
携带分片键
分片路由
使用hint
直接路由
单表 || 绑定表
标准路由
多表 & 无绑定表
笛卡尔路由
不携带分片键
广播路由
设置类DAL & TCL
全库路由
DQL & DML & DDL
全库表路由
DCL
全实例路由
查询类DAL
单播路由
use database
阻断路由

图 7: 路由引擎结构

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 8: 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_0 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 9: 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。

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 10: 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 11: 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 12: 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 13: 执行引擎流程图

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 14: 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 15: 排序归并示例 2

图 16: 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 17: 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“Jetty”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“Jetty”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“Jetty”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“Jetty”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。

4.1.6 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。

图 18: 归并引擎结构

SQL

由于 SQL 语法灵活复杂，分布式数据库和单机数据库的查询场景又不完全相同，难免有和单机数据库不兼容的 SQL 出现。

本文详细罗列出已明确可支持的 SQL 种类以及已明确不支持的 SQL 种类，尽量让使用者避免踩坑。

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

支持项

路由至单数据节点

- 100% 全兼容（目前仅 MySQL，其他数据库完善中）。

路由至多数据节点

全面支持 DML、DDL、DCL、TCL 和部分 DAL。支持分页、去重、排序、分组、聚合、关联查询（不支持跨库关联）。以下用最为复杂的 DML 举例：

- 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)]
```

不支持项

路由至多数据节点

不支持 CASE WHEN、HAVING、UNION (ALL)，有限支持子查询。

除了分页子查询的支持之外 (详情请参考分页)，也支持同等模式的子查询。无论嵌套多少层，ShardingSphere 都可以解析至第一个包含数据表的子查询，一旦在下层嵌套中再次找到包含数据表的子查询将直接抛出解析异常。

例如，以下子查询可以支持：

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

以下子查询不支持：

```
SELECT COUNT(*) FROM (SELECT * FROM t_order o WHERE o.id IN (SELECT id FROM t_order
WHERE status = ?))
```

简单来说，通过子查询进行非功能需求，在大部分情况下是可以支持的。比如分页、统计总数等；而通过子查询实现业务查询当前并不能支持。

由于归并的限制，子查询中包含聚合函数目前无法支持。

不支持包含 schema 的 SQL。因为 ShardingSphere 的理念是像使用一个数据源一样使用多数据源，因此对 SQL 的访问都是在同一个逻辑 schema 之上。

对分片键进行操作

运算表达式和函数中的分片键会导致全路由。

假设 `create_time` 为分片键，则无法精确路由形如 SQL：

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

由于 ShardingSphere 只能通过 SQL 字面提取用于分片的值，因此当分片键处于运算表达式或函数中时，ShardingSphere 无法提前获取分片键位于数据库中的值，从而无法计算出真正的分片值。

当出现此类分片键处于运算表达式或函数中的 SQL 时，ShardingSphere 将采用全路由的形式获取结果。

示例

支持的 SQL

不支持的 SQL

DISTINCT 支持情况详细说明

支持的 SQL

SQL
SELECT DISTINCT * FROM tbl_name WHERE col1 = ?
SELECT DISTINCT col1 FROM tbl_name
SELECT DISTINCT col1, col2, col3 FROM tbl_name
SELECT DISTINCT col1 FROM tbl_name ORDER BY col1
SELECT DISTINCT col1 FROM tbl_name ORDER BY col2
SELECT DISTINCT(col1) FROM tbl_name
SELECT AVG(DISTINCT col1) FROM tbl_name
SELECT SUM(DISTINCT col1) FROM tbl_name
SELECT COUNT(DISTINCT col1) FROM tbl_name
SELECT COUNT(DISTINCT col1) FROM tbl_name GROUP BY col1
SELECT COUNT(DISTINCT col1 + col2) FROM tbl_name
SELECT COUNT(DISTINCT col1), SUM(DISTINCT col1) FROM tbl_name
SELECT COUNT(DISTINCT col1), col1 FROM tbl_name GROUP BY col1
SELECT col1, COUNT(DISTINCT col1) FROM tbl_name GROUP BY col1

不支持的 SQL

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 1000000 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。对于本身即有序的待排序对象，归并排序的时间复杂度仅为 `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 ?
```

解析器

ShardingSphere 使用不同解析器支持 SQL 多种方言。对于未实现解析器的特定 SQL 方言，默认采用 SQL92 标准进行解析。

特定 SQL 方言解析器

- PostgreSQL 解析器
- MySQL 解析器
- Oracle 解析器
- SQLServer 解析器

注：MySQL 解析器支持的方言包括 MySQL、H2 和 MariaDB。

默认 SQL 方言解析器

其他 SQL 方言，如 SQLite、Sybase、DB2 和 Informix 等，默认采用 SQL92 标准进行解析。

4.2 分布式事务

4.2.1 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。
- 而最终一致性（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

4.2.2 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。

4.2.3 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。

4.2.4 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 Seata 的柔性事务

XA 两阶段事务

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。

图 19: 两阶段提交模型

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以 jar 包的形式提供服务，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。

Seata 柔性事务

Seata是阿里集团和蚂蚁金服联合打造的分布式事务框架。其 AT 事务的目标是在微服务架构下，提供增量的事务 ACID 语意，让开发者像使用本地事务一样，使用分布式事务，核心理念同 Apache ShardingSphere 一脉相承。

Seata AT 事务模型包含 TM (事务管理器)，RM (资源管理器) 和 TC (事务协调器)。TC 是一个独立部署的服务，TM 和 RM 以 jar 包的方式同业务应用一同部署，它们同 TC 建立长连接，在整个事务生命周期内，保持远程通信。TM 是全局事务的发起方，负责全局事务的开启，提交和回滚。RM 是全局事务的参与者，负责分支事务的执行结果上报，并且通过 TC 的协调进行分支事务的提交和回滚。

Seata 管理的分布式事务的典型生命周期：

1. TM 要求 TC 开始一个全新的全局事务。TC 生成一个代表该全局事务的 XID。
2. XID 贯穿于微服务的整个调用链。
3. 作为该 XID 对应到的 TC 下的全局事务的一部分，RM 注册本地事务。
4. TM 要求 TC 提交或回滚 XID 对应的全局事务。
5. TC 驱动 XID 对应的全局事务下的所有分支事务完成提交或回滚。

图 20: Seata AT 事务模型

4.2.5 实现原理

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

XA 两阶段事务

`XAShardingTransactionManager` 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。

图 21: XA 事务实现原理

开启全局事务

收到接入端的 `set autoCommit=0` 时，`XAShardingTransactionManager` 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

XAShardingTransactionManager 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。

提交或回滚事务

XAShardingTransactionManager 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
```

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 22: 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。

4.2.6 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回滚。

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。

XA 两阶段事务

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。

Seata 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。

4.3 读写分离

4.3.1 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。

4.3.2 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。下图展现了将分库分表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。

图 23: 读写分离

图 24: 数据分片 + 读写分离

4.3.3 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 ShardingSphere 读写分离模块的主要设计目标。

4.3.4 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。

4.3.5 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，也可配合分库分表使用；
- 独立使用读写分离支持 SQL 透传；
- 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；
- 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；
- 主库双写或多写；
- 跨主库和从库之间的事务的数据不一致。主从模型中，事务中读写均用主库。

4.4 分布式治理

4.4.1 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。在这种情况下，如何高效、自动化管理集群节点，实现不同节点的协同工作，配置一致性，状态一致性，高可用性，可观测性等，就成为一个重要的挑战。

本部分包括三个模块：治理、集群管理、可观测性。

4.4.2 挑战

分布式治理的挑战，主要在于集群管理的复杂性，以及如果以统一和标准的方式对接各种第三方集成组件。

集成管理的复杂性体现在，一方面我们需要把所有的节点，不管是底层数据库节点，还是中间件或者业务系统节点，它们的状态都统一管理起来，并且能够实时的探测到最新的变动情况，进一步为集群的控制和调度提供依据。这方面我们使用集群拓扑状态图来管理集群状态，同时使用心跳检测机制实现状态检测与更新。

另一方面，不同节点节点之间的统一协调，策略与规则的同步，也需要我们能够设计一套在分布式情况下，进行全局事件通知机制，以及独占性操作的分布式协调锁机制。这方面，我们使用 Zookeeper/Etcd 等实现配置的同步，状态变更的通知，以及分布式锁来控制排他性操作。

同时，由于治理功能本身可以采用合适的第三方组件作为基础服务，需要我们抽象统一的接口，统一各种不同的组件的标准调用 API，对接到治理功能模块。

最后对于可管理性和可观测性的要求，我们需要完善通过 UI 查询、操作和控制系统的功能，进一步完善对于 tracing 和 APM 的支持，实现 Metric 指标监控以及与 Prometheus、Grafana 的支持，实现实时监控。

4.4.3 目标

对于治理功能，目标如下：

- 实现配置中心：支持 Zookeeper/etcd/Apollo/Nacos，管理数据源、规则和策略的配置。
- 实现注册中心：支持 Zookeeper/etcd，管理各个 Proxy 示例的状态。
- 实现元数据中心：支持 Zookeeper/etcd，管理 LogicSchema 的元数据。

对于可观测性，目标如下：

- 支持 OpenTracing/Skywalking 集成，实现调用链的跟踪；
- 实现 Metric 支持，对接 Prometheus、Grafana，实现监控指标的可视化展示。

4.4.4 治理

导览

本小节主要介绍 Apache ShardingSphere 分布式治理的相关功能

- 配置中心
- 注册中心
- 元数据中心
- 第三方组件依赖

配置中心

实现动机

- 配置集中化：越来越多的运行时实例，使得散落的配置难于管理，配置不同步导致的问题十分严重。将配置集中于配置中心，可以更加有效进行管理。
- 配置动态化：配置修改后的分发，是配置中心可以提供的另一个重要能力。它可支持数据源和规则的动态切换。

配置中心数据结构

配置中心在定义的命名空间的 `config` 节点下，以 YAML 格式存储，包括数据源信息，规则信息、权限配置和属性配置，可通过修改节点来实现对于配置的动态管理。

```
config
    ├──authentication                          # 权限配置
    ├──props                                   # 属性配置
    ├──schema                                  # Schema 配置
    ├      ├──schema_1                         # Schema 名称 1
    ├      ├      ├──datasource                # 数据源配置
    ├      ├      ├──rule                      # 规则配置
    ├      ├──schema_2                         # Schema 名称 2
    ├      ├      ├──datasource                # 数据源配置
    ├      ├      ├──rule                      # 规则配置
```

config/authentication

权限配置，可配置访问 ShardingSphere-Proxy 的用户名和密码。

```
username: root
password: root
```

config/props

属性配置，详情请参见配置手册。

```
executor.size: 20
sql.show: true
```

config/schema/schemeName/datasource

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: !!org.apache.shardingsphere.orchestration.core.common.yaml.config.
YamlDataSourceConfiguration
  dataSourceClassName: com.zaxxer.hikari.HikariDataSource
  props:
    url: jdbc:mysql://127.0.0.1:3306/demo_ds_0?serverTimezone=UTC&useSSL=false
    password: null
    maxPoolSize: 50
    maintenanceIntervalMilliseconds: 30000
    connectionTimeoutMilliseconds: 30000
    idleTimeoutMilliseconds: 60000
    minPoolSize: 1
    username: root
    maxLifetimeMilliseconds: 1800000
ds_1: !!org.apache.shardingsphere.orchestration.core.common.yaml.
configYamlDataSourceConfiguration
  dataSourceClassName: com.zaxxer.hikari.HikariDataSource
  props:
    url: jdbc:mysql://127.0.0.1:3306/demo_ds_1?serverTimezone=UTC&useSSL=false
    password: null
    maxPoolSize: 50
    maintenanceIntervalMilliseconds: 30000
    connectionTimeoutMilliseconds: 30000
    idleTimeoutMilliseconds: 60000
    minPoolSize: 1
    username: root
    maxLifetimeMilliseconds: 1800000
```

config/schema/sharding_db/rule

规则配置，可包括数据分片、读写分离、数据加密、影子库压测、多副本等配置。

```
rules:
- !SHARDING
  xxx

- !MASTERSLAVE
  xxx

- !ENCRYPT
  xxx
```

动态生效

在注册中心上修改、删除、新增相关配置，会动态推送到生产环境并立即生效。

注册中心

实现动机

- 相对于配置中心管理配置数据，注册中心存放运行时的动态/临时状态数据，比如可用的 ShardingSphere 的实例，需要禁用或熔断的数据源等。
- 通过注册中心，可以提供熔断数据库访问程序对数据库的访问和禁用从库的访问的编排治理能力。治理模块仍然有大量未完成的功能（比如流控等）。

注册中心数据结构

注册中心在定义的命名空间的 `state` 节点下，创建数据库访问对象运行节点，用于区分不同数据库访问实例。包括 `instances` 和 `datasources` 节点。

```
instances
    ├──your_instance_ip_a@-@your_instance_pid_x
    ├──your_instance_ip_b@-@your_instance_pid_y
    ├──....
datasources
    ├──ds0
    ├──ds1
    ├──....
```

ShardingSphere-Proxy 支持多逻辑数据源，因此 `datasources` 子节点的名称采用 `schema_name.data_source_name` 的形式。

```
instances
    ├──your_instance_ip_a@-@your_instance_pid_x
    ├──your_instance_ip_b@-@your_instance_pid_y
    ├──....
datasources
    ├──sharding_db.ds0
    ├──sharding_db.ds1
    ├──....
```

state/instances

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 PID 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。

state/datasources

可以治理读写分离从库，可动态添加删除以及禁用。

操作指南

熔断实例

可在 `IP 地址 @-@PID` 节点写入 `DISABLED`（忽略大小写）表示禁用该实例，删除 `DISABLED` 表示启用。

Zookeeper 命令如下：

```
[zk: localhost:2181(CONNECTED) 0] set /your_zk_namespace/your_app_name/state/
instances/your_instance_ip_a@-@your_instance_pid_x DISABLED
```

禁用从库

在读写分离场景下，可在数据源名称子节点中写入 `DISABLED`（忽略大小写）表示禁用从库数据源，删除 `DISABLED` 或节点表示启用。

Zookeeper 命令如下：

```
[zk: localhost:2181(CONNECTED) 0] set /your_zk_namespace/your_app_name/state/
datasources/your_slave_datasource_name DISABLED
```

元数据中心

实现动机

- 元数据是 ShardingSphere 所使用的数据源的表、列和索引等核心数据, 元数据保障 ShardingSphere 各个组件能够正确运行。
- 元数据中心对元数据进行统一组织和管理，实现元数据的统一加载、变更通知和数据同步。

元数据中心数据结构

元数据中心在定义的命名空间和治理节点的 `metadata` 节点下，以 YAML 格式存储，每个逻辑数据源独立存储。

```
├─orchestration-namespace
│   ├─orchestration-name
│   │   ├──metadata
│   │   │    ├──schema_1
│   │   │    │    ├── [YAML text contents]
│   │   │    ├──schema_2
│   │   │    │    ├── [YAML text contents]
│   │   │    ├──....
```

YAML Text Contents

在元数据中心中,元数据分为 `configuredSchemaMetaData` 和 `unconfiguredSchemaMetaDataMap` 两部分。

元数据内容目前不支持动态修改。

```
configuredSchemaMetaData:
  tables:                                       # 表
    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
```

(下页继续)

(续上页)

```
        t_user_order_id_index:                  # 索引名
          name: t_user_order_id_index
    t_order_item:
      columns:
        order_id:
          caseSensitive: false
          dataType: 0
          generated: false
          name: order_id
          primaryKey: false
unconfiguredSchemaMetaDataMap:
  ds_0:                                         # 数据源
    tables:                                     # 表
      t_user:                                   # 表名
        columns:                                # 列
          user_id:                              # 列名
            caseSensitive: false
            dataType: 0
            generated: false
            name: user_id
            primaryKey: false
          id:
            caseSensitive: false
            dataType: 0
            generated: false
            name: id
            primaryKey: true
          order_id:
            caseSensitive: false
            dataType: 0
            generated: false
            name: order_id
            primaryKey: false
        indexes:                                # 索引
          t_user_order_id_index:                # 索引名
            name: t_user_order_id_index
          t_user_user_id_index:
            name: t_user_user_id_index
          primary:
            name: PRIMARY
```

configuredSchemaMetaData

存储所有配置了分片规则的数据源的元数据。

unconfiguredSchemaMetaDataMap

存储没有配置分片规则的数据源的元数据。

变更通知

通过某一个 ShardingSphere 实例执行 DDL 之后，ShardingSphere 先将新的元数据存储到元数据中心，然后通过事件广播机制，通知其它 ShardingSphere 实例从元数据中心同步元数据，保证元数据一致。

第三方组件依赖

Apache ShardingSphere 在数据库治理模块使用 SPI 方式载入数据到配置中心/注册/元数据中心，进行实例熔断和数据库禁用。目前，Apache ShardingSphere 内部支持 ZooKeeper，Etcd，Apollo 和 Nacos 等常用的配置中心/注册中心。此外，开发者可以使用其他第三方组件，并通过 SPI 的方式注入到 Apache ShardingSphere，从而使用该配置中心/注册/元数据中心，实现数据库治理功能。

	实现驱动	版本	配置中心	注册中心	元数据中心
Zookeeper	Apache Curator	3.6.x	支持	支持	支持
Etcd	jetcd	v3	支持	支持	支持
Apollo	Apollo Client	1.5.0	支持	不支持	不支持
Nacos	Nacos Client	1.0.0	支持	不支持	不支持

4.4.5 可观察性

导览

本小节主要介绍 Apache ShardingSphere 可观察性的相关功能

- 应用性能监控集成
- 度量指标监控

应用性能监控集成

背景

APM 是应用性能监控的缩写。目前 APM 的主要功能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。

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，而是将 SQL 解析与 SQL 执行这两块数据分片的最核心的相关信息发送至应用性能监控系统,并交由其处理。换句话说,Apache ShardingSphere 仅负责产生具有价值的数据，并通过标准协议递交至相关系统。Apache ShardingSphere 可以通过两种方式对接应用性能监控系统。

第一种方式是使用 OpenTracing API 发送性能追踪数据。面向 OpenTracing 协议的 APM 产品都可以与 Apache ShardingSphere 自动对接，比如 SkyWalking，Zipkin 和 Jaeger。使用这种方式只需要在启动时配置 OpenTracing 协议的实现者即可。它的优点是可以兼容所有的与 OpenTracing 协议兼容的产品作为 APM 的展现系统，如果采用公司愿意实现自己的 APM 系统，也只需要实现 OpenTracing 协议，即可自动展示 Apache ShardingSphere 的链路追踪信息。缺点是 OpenTracing 协议发展并不稳定，较新的版本实现者较少，且协议本身过于中立，对于个性化的相关产品的实现不如原生支持强大。

第二种方式是使用 SkyWalking 的自动探针。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。

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.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，应将原 *Apache ShardingSphere* 探针插件禁用，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。

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

图 25: 拓扑图

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

图 26: 跟踪图

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。

图 27: 解析节点

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

图 28: 实际访问节点

异常情况展示

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。

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

图 29: 异常跟踪图

图 30: 异常节点

图 31: 异常节点日志

度量指标监控

背景

Metrics 是一种评估系统在运行时的度量指标，通常一组度量指标可以帮助我们评估系统性能，为系统优化或业务策略给出相关的意见。Apache ShardingSphere 旨在打造一款分布式数据库解决方案，而对于数据库来说，分析它的运行状态，连接数据，事务数，吞吐量等相关指标，为数据库的调度，数据平滑迁移，分库分表等策略提供可视化的建议与帮助尤为重要。

方案

Apache ShardingSphere 遵循 Metrics 标准，定义了一套可插拔的 SPI 标准，它并不存储, 收集, 展现 Metrics 信息，只是负责在程序中对 Metrics 进行埋点，目前默认的实现方案为: Prometheus 客户端 API 埋点，服务端通过 http 协议来定时抓取 Metrics 数据。

图 32: 流程图

指标

目前定义了 4 种类型的指标

- Counter : 计数器是一个累积度量，它表示单个单调递增的计数器，其值在重新启动时只能增加或重置为零。
- Gauge : 仪表盘是一种度量标准，它表示一个可以任意上下移动的数值。
- Histogram : 直方图对观察结果 (通常是请求持续时间或响应大小等) 进行采样，并按可配置的桶进行计数。它还提供了所有观测值的和。
- Summary : 摘要是类似于柱状图的观察结果 (通常是请求持续时间和响应大小之类的内容)。虽然它还提供了观察值的总数和所有观察值的总和，但它计算了一个滑动时间窗口上的可配置分位数。

使用

在 ShardingSphere-Proxy 的 server.yaml 文件中新增以下配置:

```
metrics:
   name: prometheus # 指定类型为 prometheus.
   host:  127.0.0.1 # 指定 host，如果为空，则获取默认
   port:  9190  # 指定 prometheus 服务端抓取 metrics 端口
   enable : true # 配置为 true 代表开启，设置为 false 代表关闭 ，此字段不配置时候，默认开启.
```

用户自己搭建 `Prometheus` 服务，在 prometheus.yml 文件中新增如下配置:

```
scrape_configs:
  # The job name is added as a label `job=<job-name>` to any time series scraped
from this config.
  - job_name: 'shardingSphere-proxy'
    # metrics_path defaults to '/metrics'
    # scheme defaults to 'http'.
    static_configs:
    - targets: ['localhost:9190']
```

面板

推荐使用 `Granfana`，用户可以自定义查询来个性化显示面板盘，后续我们会提供默认的面板盘配置。

4.5 弹性伸缩

4.5.1 背景

Apache ShardingSphere 提供了数据分片的能力，可以将数据分散到不同的数据库节点上，提升整体处理能力。但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；同时，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。

4.5.2 简介

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。

从 **4.1.0** 开始向用户提供。

4.5.3 挑战

Apache ShardingSphere 在分片策略和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。如何找到一种方式，即能支持各类不同用户的分片策略和算法，又能高效地将数据节点进行伸缩，是弹性伸缩面临的第一个挑战；

同时，弹性伸缩过程中，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，如何保证数据的可用性和正确性，是弹性伸缩的第三个挑战。

4.5.4 目标

支持各类用户自定义的分片策略，减少用户在数据伸缩及迁移时的重复工作及业务影响，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。

用户
查询
UI
进度
修改ShardingSphere配置
自动创建伸缩作业
弹性伸缩
存量数据迁移
增量数据同步
数据校验
持久化
注册中心
(Zookeeper)
自动切换
UI
手动
切换
设置只读
校验增量日志进度
切换ShardingSphere配置

图 33: 结构总揽

4.5.5 状态

当前处于 **alpha** 开发阶段。

图 34: 路线图

4.5.6 核心概念

弹性伸缩作业

指一次将数据由旧分片规则伸缩至新分片规则的完整流程。

数据节点

同数据分片中的数据节点

存量数据

在弹性伸缩作业开始前，数据分片中已有的数据。

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。

4.5.7 实现原理

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。

图 35: 伸缩总揽

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 36: 伸缩工作流

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。

4.5.8 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

- 不支持无主键表的扩容和缩容。

4.6 数据加密

4.6.1 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。

4.6.2 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。

4.6.3 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。

4.6.4 核心概念

TODO

4.6.5 实现原理

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

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。

图 37: 1

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 内置了两种加解密算法：AES/MD5。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。

图 38: 2

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。

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。

模型
应用程序
交互
logicColumn
ShardingSphere
映射
plainColumn
cipherColumn
真实数据表
举例
应用程序
交互
pwd
t_user
映射
pwd_plain
pwd_cipher
t_user

图 39: 3

图 40: 4

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:
```

(下页继续)

(续上页)

```
        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 41: 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
props:
  query.with.cipher.column: 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 `query.with.cipher.column` 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 42: 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 query.with.cipher.column=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 query.with.cipher.column 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 query.with.cipher.column=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 43: 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:
```

(下页继续)

(续上页)

```
      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
props:
  query.with.cipher.column: true
```

其处理流程如下:

图 44: 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。

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。

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两种具体实现类，分别是 MD5(不可逆)，AES(可逆)，用户只需配置即可使用这两种内置的方案。

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。

4.6.6 使用规范

支持项

- 后端数据库为 MySQL、Oracle、PostgreSQL、SQLServer；
- 用户需要对数据库表中某个或多个列进行加密（数据加密 & 解密）；
- 兼容所有常用 SQL。

不支持项

- 用户需要自行处理数据库中原始的存量数据、洗数；
- 使用加密功能 + 分库分表功能，部分特殊 SQL 不支持，请参考SQL 使用规范；
- 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。

4.7 影子库压测

4.7.1 背景

在基于微服务的分布式应用架构下，由于整体服务都是通过一系列的微服务调用、中间件调用来完成业务需求，所以对于单个服务的压测已经不能代表真实场景。而在线下环境中，如果重新搭建一整套与生产环境类似的压测环境，成本太高，并且往往无法模拟线上环境的体量以及复杂度。这种场景下，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较为准确地反应系统真实容量水平和性能。

4.7.2 挑战

全链路压测是一项复杂而庞大的工作，需要各个中间件、微服务之间相应的调整与配合，以应对不同流量以及压测标识的透传，通常应该有一整套压测平台与测试计划。其中，在数据库层面，为了保证生产数据的可靠性与完整性，做好数据隔离，需要将压测的数据请求打入影子库，以防压测数据写入生产数据库而对真实数据造成污染。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。

4.7.3 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。基于内核的 SQL 解析能力，以及可插拔平台架构，实现压测数据与生产数据的隔离，帮助应用自动路由，支持全链路压测，是 Apache ShardingSphere 影子数据库模块的主要设计目标。

4.7.4 核心概念

影子字段

判断该条 SQL 是否需要路由到影子数据库，为逻辑字段，数据库中不存在。

生产数据库

生产数据使用的数据库。

影子数据库

进行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。

4.7.5 实现原理

整体架构

Apache ShardingSphere 通过解析 SQL，根据配置文件中用户设置的影子规则，对传入的 SQL 进行路由并改写，删除影子字段与字段值。用户无需关注具体过程，使用时仅对 SQL 进行相应改造，添加影子字段与相应的配置即可。

影子规则

影子规则包含影子字段及映射关系。

处理过程

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，将影子数据路由到与之对应的影子库，生产数据则维持不变。

接着对 SQL 进行改写，由于影子字段为逻辑字段，在数据库中实际不存在，所以在改写过程中会删除这个字段及其对应的参数。

DML 语句的处理过程同理，对于非 DML 语句，如创建数据表等，会在生产数据库与影子数据库分别执行。

Logic SQL
JDBC
SQL Parse
SQL Route
SQL Rewrite
SQL Execute
ShardingSphere-Shadow
DB

图 45: 执行流程

图 46: 规则

图 47: sql 过程

4.8 多数据副本

4.8.1 导览

TODO

4.9 可插拔架构

4.9.1 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。

4.9.2 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、多数据副本、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。

4.9.3 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。

4.10 测试引擎

ShardingSphere 提供了完善的测试引擎。它以 XML 方式定义 SQL，每条 SQL 由 SQL 解析单元测试引擎和整合测试引擎驱动，每个引擎分别为 H2、MySQL、PostgreSQL、SQLServer 和 Oracle 数据库运行测试用例。

为了使测试更容易上手，shardingsphere 中的测试引擎不必修改任何 **Java** 代码，只需要修改相应的配置文件即可运行断言。

SQL 解析单元测试全面覆盖 SQL 占位符和字面量维度。整合测试进一步拆分为策略和 JDBC 两个维度，策略维度包括分库分表、仅分表、仅分库、读写分离等策略，JDBC 维度包括 Statement、PreparedStatement。

因此，1 条 SQL 会驱动 5 种数据库的解析 * 2 种参数传递类型 + 5 种数据库 * 5 种分片策略 * 2 种 JDBC 运行方式 = 60 个测试用例，以达到 ShardingSphere 对于高质量的追求。

鉴于表述路径时，子路径可能不止一个，名称为某一类术语的集合，这里用 SQL-TYPE 以及 SHARDING-TYPE 表述如下：

SQL-TYPE : 是 DAL，DCL，DDL，DML，DQL，TCL 中的某一个或者集合

SHARDING-TYPE : 是 db，dbtbl_with_masterslave，masterslave，tbl 中的某一个或者集合

4.10.1 SQL 测试用例

目标

SQL 测试用例的代码位于 `sharding-sql-test` 模块下。该测试用例的作用主要有两个：

1. 通过单元测试，测试通配符的替换以及 `SQLCasesLoader` 的稳定性。
2. 将 SQL 测试用例中 `resources` 下定义的所有 SQL 共享给其他项目。

待测试的 SQL 存放在 `/sharding-sql-test/src/main/resources/sql/sharding/SQL-TYPE/*.xml` 文件中。例如：

```
<sql-cases>
    <sql-case id="select_constant_without_table" value="SELECT 1 as a" />
    <sql-case id="select_with_same_table_name_and_alias" value="SELECT t_order.*
FROM t_order t_order WHERE user_id = ? AND order_id = ?" />
    <sql-case id="select_with_same_table_name_and_alias_column_with_owner" value=
"SELECT t_order.order_id,t_order.user_id,status FROM t_order t_order WHERE t_order.
user_id = ? AND order_id = ?" db-types="MySQL,H2"/>
</sql-cases>
```

开发者通过该文件指定待断言的 SQL 以及该 SQL 所适配的数据库类型。将 `sharding-sql-test` 提取为单独的模块，以保证每个 SQL 用例可以在不同模块的测试引擎中共享。

流程

如下图为 SQL 测试用例的数据流程：

4.10.2 整合测试引擎

流程

Junit 中的 `Parameterized` 会聚合起所有的测试数据，并将测试数据一一传递给测试方法进行断言。数据处理就像是沙漏中的流沙：

ShardingSQLCasesRegistry
SQLCasesLoader
sql_case_id
sql_case_value
sql case
sql_case_id
sql_case_value
sql case
sql_case_id
sql_case_value
sql case
sql_case_id
sql_case_value
sql case
Load
dal
dml
dcl
ddl
dql
tcl
XML

图 48: 测试引擎

IntegrateTestParameters
*-integrate-test-cases.xml
/sql/*.xml
env.properties
Integrate test cases
sql cases
test environment
Collection<Object[]>
object[0] = sql_case_id
object[1] = expected_file
object[2] = sharding_type
object[3] = environment
object[4] = sql_type
Create DB
Create Table
BaseIT
Inheritance
BatchIT
dataSetEnvironmentManager
expectedFiles
dataSetEnvironmentManager
expectedFiles
SingleIT
Inheritance
Inheritance
specific
BatchIT
dmlIT
dqlIT
ddlIT
...
dmlIT
dqlIT
ddlIT
...
specific
SingleIT
1.assert count array length through connection
2.assert each column name
3.assert every column value in each row
1.assert count through connection
2.assert each column name
3.assert every column value in each row

配置

- 环境类文件
  - /shardingsphere-test-suite/src/test/resources/integrate/env.properties
  - /shardingsphere-test-suite/src/test/resources/integrate/env/SQL-TYPE/dataset.xml
  - /shardingsphere-test-suite/src/test/resources/integrate/env/SQL-TYPE/schema.xml
- 测试用例类文件
  - /shardingsphere-test-suite/src/test/resources/integrate/cases/SQL-TYPE/SQL-TYPE-integrate-test-cases.xml
  - /shardingsphere-test-suite/src/test/resources/integrate/cases/SQL-TYPE/dataset/FEATURE-TYPE/*.xml
- sql-case 文件
  - /sharding-sql-test/src/main/resources/sql/sharding/SQL-TYPE/*.xml

环境配置

集成测试需要真实的数据库环境，根据相应的配置文件创建测试环境：

首先，修改配置文件 `/shardingsphere-test-suite/src/test/resources/integrate/env.properties`，例子如下：

```
# 测试主键，并发，column index 等的开关
run.additional.cases=false

# 分片策略，可指定多种策略
sharding.rule.type=db,tbl,dbtbl_with_masterslave,masterslave

# 要测试的数据库，可以指定多种数据库 (H2,MySQL,Oracle,SQLServer,PostgreSQL)
databases=MySQL,PostgreSQL

# MySQL 配置
mysql.host=127.0.0.1
mysql.port=13306
mysql.username=root
mysql.password=root

## PostgreSQL 配置
postgresql.host=db.psql
postgresql.port=5432
postgresql.username=postgres
postgresql.password=

## SQLServer 配置
sqlserver.host=db.mssql
```

(下页继续)

(续上页)

```
sqlserver.port=1433
sqlserver.username=sa
sqlserver.password=Jdbc1234

## Oracle 配置
oracle.host=db.oracle
oracle.port=1521
oracle.username=jdbc
oracle.password=jdbc
```

其次，修改文件 `/shardingsphere-test-suite/src/test/resources/integrate/env/SQL-TYPE/dataset.xml` 在 `dataset.xml` 文件中定义元数据和测试数据。例如：

```
<dataset>
    <metadata data-nodes="tbl.t_order_${0..9}">
        <column name="order_id" type="numeric" />
        <column name="user_id" type="numeric" />
        <column name="status" type="varchar" />
    </metadata>
    <row data-node="tbl.t_order_0" values="1000, 10, init" />
    <row data-node="tbl.t_order_1" values="1001, 10, init" />
    <row data-node="tbl.t_order_2" values="1002, 10, init" />
    <row data-node="tbl.t_order_3" values="1003, 10, init" />
    <row data-node="tbl.t_order_4" values="1004, 10, init" />
    <row data-node="tbl.t_order_5" values="1005, 10, init" />
    <row data-node="tbl.t_order_6" values="1006, 10, init" />
    <row data-node="tbl.t_order_7" values="1007, 10, init" />
    <row data-node="tbl.t_order_8" values="1008, 10, init" />
    <row data-node="tbl.t_order_9" values="1009, 10, init" />
</dataset>
```

开发者可以在 `schema.xml` 中自定义建库与建表语句。

断言配置

`env.properties` 与 `dataset.xml` 确定了什么 SQL 在什么环境执行，下面是断言数据的配置：

断言的配置，需要两种文件，第一类文件位于 `/shardingsphere-test-suite/src/test/resources/integrate/cases/SQL-TYPE/SQL-TYPE-integrate-test-cases.xml` 这个文件类似于一个索引，定义了要执行的 SQL，参数以及期待的数据的文件位置。这里的 test-case 引用的就是 `sharding-sql-test` 中 SQL 对应的 `sql-case-id`，例子如下：

```
<integrate-test-cases>
    <dml-test-case sql-case-id="insert_with_all_placeholders">
       <assertion parameters="1:int, 1:int, insert:String" expected-data-file=
"insert_for_order_1.xml" />
       <assertion parameters="2:int, 2:int, insert:String" expected-data-file=
"insert_for_order_2.xml" />
```

(下页继续)

(续上页)

```
    </dml-test-case>
</integrate-test-cases>
```

还有一类文件 – 断言数据，也就是上面配置中的 expected-data-file 对应的文件，文件在 `/shardingsphere-test-suite/src/test/resources/integrate/cases/SQL-TYPE/dataset/FEATURE-TYPE/*.xml` 这个文件内容跟 dataset.xml 很相似，只不过 `expected-data-file` 文件中不仅定义了断言的数据，还有相应 SQL 执行后的返回值等。例如：

```
<dataset update-count="1">
    <metadata data-nodes="db_${0..9}.t_order">
        <column name="order_id" type="numeric" />
        <column name="user_id" type="numeric" />
        <column name="status" type="varchar" />
    </metadata>
    <row data-node="db_0.t_order" values="1000, 10, update" />
    <row data-node="db_0.t_order" values="1001, 10, init" />
    <row data-node="db_0.t_order" values="2000, 20, init" />
    <row data-node="db_0.t_order" values="2001, 20, init" />
</dataset>
```

所有需要配置的数据，都已经配置完毕，启动相应的集成测试类即可，全程不需要修改任何 `Java` 代码，只需要在 `xml` 中做数据初始化以及断言，极大的降低了 ShardingSphere 数据测试的门槛以及复杂度。

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，整合测试中的分库分表采用了 10 库 10 表的方式，因此运行测试用例的时间会比较长。

4.10.3 SQL 解析测试引擎

数据准备

SQL 解析不需要真实的测试环境，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 `/sharding-sql-test/src/main/resources/sql/sharding/SQL-TYPE/*.xml` 添加待测试的 SQL 即可。

断言解析数据

断言的解析数据保存在 `/sharding-core/sharding-core-parse/sharding-core-parse-test/src/test/resources/sharding/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 `sharding-core-parse-test` 下对应的 engine 启动 SQL 解析的测试了。

4.10.4 SQL 改写测试引擎

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：

```yaml
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
shardingRule:
  tables:
    t_account:
      actualDataNodes: db.t_account_${0..1}
      tableStrategy:
        inline:
          shardingColumn: account_id
          algorithmExpression: t_account_${account_id % 2}
      keyGenerateStrategy:
        type: TEST
        column: account_id
    t_account_detail:
      actualDataNodes: db.t_account_detail_${0..1}
      tableStrategy:
        inline:
          shardingColumn: order_id
```

(下页继续)

(续上页)

```
        algorithmExpression: t_account_detail_${account_id % 2}
  bindingTables:
    - t_account, t_account_detail
```

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。

4.10.5 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

```yaml
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
shardingRule:
    tables:
      tbl:
        actualDataNodes: ds_${0..3}.tbl${0..1023}
        tableStrategy:
          inline:
            shardingColumn: k
            algorithmExpression: tbl${k % 1024}
        keyGenerateStrategy:
            type: SNOWFLAKE
            column: id
```

(下页继续)

(续上页)

```
    defaultDatabaseStrategy:
      inline:
        shardingColumn: id
        algorithmExpression: ds_${id % 4}
    defaultTableStrategy:
      none:
```

主从配置

```
schemaName: sharding_db

dataSources:
  master_ds:
    url: jdbc:mysql://***.***.***.***:****/ds?serverTimezone=UTC&useSSL=false
    username: test
    password:
    connectionTimeoutMilliseconds: 30000
    idleTimeoutMilliseconds: 60000
    maxLifetimeMilliseconds: 1800000
    maxPoolSize: 200
  slave_ds_0:
    url: jdbc:mysql://***.***.***.***:****/ds?serverTimezone=UTC&useSSL=false
    username: test
    password:
    connectionTimeoutMilliseconds: 30000
    idleTimeoutMilliseconds: 60000
    maxLifetimeMilliseconds: 1800000
    maxPoolSize: 200
masterSlaveRule:
  name: ms_ds
  masterDataSourceName: master_ds
  slaveDataSourceNames:
    - slave_ds_0
```

主从 + 加密 + 分库分表配置

```
schemaName: sharding_db

dataSources:
  master_ds_0:
    url: jdbc:mysql://***.***.***.***:****/ds?serverTimezone=UTC&useSSL=false
    username: test
    password:
    connectionTimeoutMilliseconds: 30000
```

(下页继续)

(续上页)

```
    idleTimeoutMilliseconds: 60000
    maxLifetimeMilliseconds: 1800000
    maxPoolSize: 200
  slave_ds_0:
    url: jdbc:mysql://***.***.***.***:****/ds?serverTimezone=UTC&useSSL=false
    username: test
    password:
    connectionTimeoutMilliseconds: 30000
    idleTimeoutMilliseconds: 60000
    maxLifetimeMilliseconds: 1800000
    maxPoolSize: 200
  master_ds_1:
    url: jdbc:mysql://***.***.***.***:****/ds?serverTimezone=UTC&useSSL=false
    username: test
    password:
    connectionTimeoutMilliseconds: 30000
    idleTimeoutMilliseconds: 60000
    maxLifetimeMilliseconds: 1800000
    maxPoolSize: 200
  slave_ds_1:
    url: jdbc:mysql://***.***.***.***:****/ds?serverTimezone=UTC&useSSL=false
    username: test
    password:
    connectionTimeoutMilliseconds: 30000
    idleTimeoutMilliseconds: 60000
    maxLifetimeMilliseconds: 1800000
    maxPoolSize: 200
  master_ds_2:
    url: jdbc:mysql://***.***.***.***:****/ds?serverTimezone=UTC&useSSL=false
    username: test
    password:
    connectionTimeoutMilliseconds: 30000
    idleTimeoutMilliseconds: 60000
    maxLifetimeMilliseconds: 1800000
    maxPoolSize: 200
  slave_ds_2:
    url: jdbc:mysql://***.***.***.***:****/ds?serverTimezone=UTC&useSSL=false
    username: test
    password:
    connectionTimeoutMilliseconds: 30000
    idleTimeoutMilliseconds: 60000
    maxLifetimeMilliseconds: 1800000
    maxPoolSize: 200
  master_ds_3:
    url: jdbc:mysql://***.***.***.***:****/ds?serverTimezone=UTC&useSSL=false
    username: test
    password:
```

(下页继续)

(续上页)

```
    connectionTimeoutMilliseconds: 30000
    idleTimeoutMilliseconds: 60000
    maxLifetimeMilliseconds: 1800000
    maxPoolSize: 200
  slave_ds_3:
    url: jdbc:mysql://***.***.***.***:****/ds?serverTimezone=UTC&useSSL=false
    username: test
    password:
    connectionTimeoutMilliseconds: 30000
    idleTimeoutMilliseconds: 60000
    maxLifetimeMilliseconds: 1800000
    maxPoolSize: 200
shardingRule:
  tables:
    tbl:
      actualDataNodes: ms_ds_${0..3}.tbl${0..1023}
      databaseStrategy:
        inline:
          shardingColumn: id
          algorithmExpression: ms_ds_${id % 4}
      tableStrategy:
        inline:
          shardingColumn: k
          algorithmExpression: tbl${k % 1024}
      keyGenerateStrategy:
        type: SNOWFLAKE
        column: id
  bindingTables:
    - tbl
  defaultDataSourceName: master_ds_1
  defaultTableStrategy:
    none:
  masterSlaveRules:
    ms_ds_0:
      masterDataSourceName: master_ds_0
      slaveDataSourceNames:
        - slave_ds_0
      loadBalanceAlgorithmType: ROUND_ROBIN
    ms_ds_1:
      masterDataSourceName: master_ds_1
      slaveDataSourceNames:
        - slave_ds_1
      loadBalanceAlgorithmType: ROUND_ROBIN
    ms_ds_2:
      masterDataSourceName: master_ds_2
      slaveDataSourceNames:
        - slave_ds_2
```

(下页继续)

(续上页)

```
      loadBalanceAlgorithmType: ROUND_ROBIN
    ms_ds_3:
      masterDataSourceName: master_ds_3
      slaveDataSourceNames:
        - slave_ds_3
      loadBalanceAlgorithmType: ROUND_ROBIN
encryptRule:
  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
```

(下页继续)

(续上页)

```
    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
shardingRule:
  tables:
    tbl:
      actualDataNodes: ds_${0..3}.tbl1
      tableStrategy:
        inline:
          shardingColumn: k
          algorithmExpression: tbl1
      keyGenerateStrategy:
          type: SNOWFLAKE
          column: id
  defaultDatabaseStrategy:
    inline:
      shardingColumn: id
      algorithmExpression: ds_${id % 4}
  defaultTableStrategy:
    none:
```

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:
```

(下页继续)

(续上页)

```
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。

5 用户手册

本章节详细阐述 Apache ShardingSphere 的 3 个相关产品 ShardingSphere-JDBC、ShardingSphere-Proxy 和 ShardingSphere-Sidecar，及其衍生产品 ShardingSphere-Scaling 和 ShardingSphere-UI 的使用。

5.1 ShardingSphere-JDBC

5.1.1 简介

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, Druid, HikariCP 等。
- 支持任意实现 JDBC 规范的数据库，目前支持 MySQL，Oracle，SQLServer，PostgreSQL 以及任何遵循 SQL92 标准的数据库。

5.1.2 对比

	ShardingSphere-JDBC	*ShardingSphere-Proxy*	*ShardingSphere-Sidecar*
数据库	任意	MySQL/PostgreSQL	MySQL/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-JDBC 的优势在于对 Java 应用的友好度。

图 1: ShardingSphere-JDBC Architecture

5.1.3 使用手册

本章节将介绍 ShardingSphere-JDBC 相关使用。更多使用细节请参见使用示例。

数据分片

数据分片是 Apache ShardingSphere 的基础能力，本节以数据分片的使用举例。除数据分片之外，读写分离、多副本、数据加密、影子库压测等功能的使用方法完全一致，只要配置相应的规则即可。多规则可以叠加配置。

详情请参见配置手册。

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>
```

规则配置

ShardingSphere-JDBC 的 Java API 通过数据源集合、规则集合以及属性配置组成。以下示例是根据 `user_id` 取模分库, 且根据 `order_id` 取模分表的 2 库 2 表的配置。

```
// 配置真实数据源
Map<String, DataSource> dataSourceMap = new HashMap<>();

// 配置第 1 个数据源
BasicDataSource dataSource1 = new BasicDataSource();
dataSource1.setDriverClassName("com.mysql.jdbc.Driver");
dataSource1.setUrl("jdbc:mysql://localhost:3306/ds0");
dataSource1.setUsername("root");
dataSource1.setPassword("");
dataSourceMap.put("ds0", dataSource1);

// 配置第 2 个数据源
BasicDataSource dataSource2 = new BasicDataSource();
dataSource2.setDriverClassName("com.mysql.jdbc.Driver");
dataSource2.setUrl("jdbc:mysql://localhost:3306/ds1");
dataSource2.setUsername("root");
dataSource2.setPassword("");
dataSourceMap.put("ds1", dataSource2);
```

(下页继续)

(续上页)

```
// 配置 t_order 表规则
ShardingTableRuleConfiguration orderTableRuleConfig = new ShardingTableRuleConfiguration(
"t_order", "ds${0..1}.t_order${0..1}");

// 配置分库策略
orderTableRuleConfig.setDatabaseShardingStrategy(new StandardShardingStrategyConfiguration(
"user_id", "dbShardingAlgorithm"));

// 配置分表策略
orderTableRuleConfig.setTableShardingStrategy(new StandardShardingStrategyConfiguration(
"order_id", "tableShardingAlgorithm"));

// 省略配置 t_order_item 表规则...
// ...

// 配置分片规则
ShardingRuleConfiguration shardingRuleConfig = new ShardingRuleConfiguration();
shardingRuleConfig.getTables().add(orderTableRuleConfig);

// 配置分库算法
Properties dbShardingAlgorithmrProps = new Properties();
dbShardingAlgorithmrProps.setProperty("algorithm.expression", "ds${user_id % 2}");
shardingRuleConfig.getShardingAlgorithms().put("dbShardingAlgorithm", new
ShardingSphereAlgorithmConfiguration("INLINE", dbShardingAlgorithmrProps));

// 配置分表算法
Properties tableShardingAlgorithmrProps = new Properties();
tableShardingAlgorithmrProps.setProperty("algorithm.expression", "t_order${order_id
% 2}");
shardingRuleConfig.getShardingAlgorithms().put("tableShardingAlgorithm", new
ShardingSphereAlgorithmConfiguration("INLINE", tableShardingAlgorithmrProps));

// 创建 ShardingSphereDataSource
DataSource dataSource = ShardingSphereDataSourceFactory.createDataSource(dataSourceMap,
 Collections.singleton((shardingRuleConfig), new Properties());
```

使用 ShardingSphereDataSource

通过 ShardingSphereDataSourceFactory 工厂创建的 ShardingSphereDataSource 实现自 JDBC 的标准接口 DataSource。可通过 DataSource 选择使用原生 JDBC，或 JPA，MyBatis 等 ORM 框架。

以原生 JDBC 使用方式为例：

```
DataSource dataSource = ShardingSphereDataSourceFactory.createDataSource(dataSourceMap,
 Collections.singleton((shardingRuleConfig), new Properties());
```

(下页继续)

(续上页)

```
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()) {
            // ...
        }
    }
}
```

使用 YAML 配置

引入 Maven 依赖

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

规则配置

ShardingSphere-JDBC 的 YAML 配置文件通过数据源集合、规则集合以及属性配置组成。以下示例是根据 `user_id` 取模分库, 且根据 `order_id` 取模分表的 2 库 2 表的配置。

```
# 配置真实数据源
dataSources:
  # 配置第 1 个数据源
  ds0: !!org.apache.commons.dbcp2.BasicDataSource
    driverClassName: com.mysql.jdbc.Driver
    url: jdbc:mysql://localhost:3306/ds0
    username: root
    password:
  # 配置第 2 个数据源
  ds1: !!org.apache.commons.dbcp2.BasicDataSource
    driverClassName: com.mysql.jdbc.Driver
    url: jdbc:mysql://localhost:3306/ds1
    username: root
    password:
```

(下页继续)

(续上页)

```yaml
rules:
# 配置分片规则
- !SHARDING
  tables:
    # 配置 t_order 表规则
    t_order:
      actualDataNodes: ds${0..1}.t_order${0..1}
      # 配置分库策略
      databaseStrategy:
        standard:
          shardingColumn: user_id
          shardingAlgorithmName: database_inline
      # 配置分表策略
      tableStrategy:
        standard:
          shardingColumn: order_id
          shardingAlgorithmName: table_inline
    t_order_item:
    # 省略配置 t_order_item 表规则...
    # ...

  # 配置分片算法
  shardingAlgorithms:
    database_inline:
      type: INLINE
      props:
        algorithm.expression: ds${user_id % 2}
    table_inline:
      type: INLINE
      props:
        algorithm.expression: t_order_${order_id % 2}
```

```java
// 创建 ShardingSphereDataSource
DataSource dataSource = YamlShardingSphereDataSourceFactory.createDataSource(yamlFile);
```

使用 ShardingSphereDataSource

通过 YamlShardingSphereDataSourceFactory 工厂创建的 ShardingSphereDataSource 实现自 JDBC 的标准接口 DataSource。可通过 DataSource 选择使用原生 JDBC，或 JPA，MyBatis 等 ORM 框架。

```java
DataSource dataSource = YamlShardingSphereDataSourceFactory.createDataSource(yamlFile);

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 (
```

(下页继续)

(续上页)

```
        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()) {
            // ...
        }
    }
}
```

使用 Spring Boot Starter

引入 Maven 依赖

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

规则配置

```
# 配置真实数据源
spring.shardingsphere.datasource.names=ds0,ds1

# 配置第 1 个数据源
spring.shardingsphere.datasource.ds0.type=org.apache.commons.dbcp2.BasicDataSource
spring.shardingsphere.datasource.ds0.driver-class-name=com.mysql.jdbc.Driver
spring.shardingsphere.datasource.ds0.url=jdbc:mysql://localhost:3306/ds0
spring.shardingsphere.datasource.ds0.username=root
spring.shardingsphere.datasource.ds0.password=

# 配置第 2 个数据源
spring.shardingsphere.datasource.ds1.type=org.apache.commons.dbcp2.BasicDataSource
spring.shardingsphere.datasource.ds1.driver-class-name=com.mysql.jdbc.Driver
spring.shardingsphere.datasource.ds1.url=jdbc:mysql://localhost:3306/ds1
spring.shardingsphere.datasource.ds1.username=root
spring.shardingsphere.datasource.ds1.password=

# 配置 t_order 表规则
spring.shardingsphere.rules.sharding.tables.t_order.actual-data-nodes=ds$->{0..1}.
t_order$->{0..1}
```

(下页继续)

(续上页)

```
# 配置分库策略
spring.shardingsphere.rules.sharding.tables.t_order.database-strategy.standard.
sharding-column=user_id
spring.shardingsphere.rules.sharding.tables.t_order.database-strategy.standard.
sharding-algorithm-name=database_inline

# 配置分表策略
spring.shardingsphere.rules.sharding.tables.t_order.table-strategy.standard.
sharding-column=order_id
spring.shardingsphere.rules.sharding.tables.t_order.table-strategy.standard.
sharding-algorithm-name=table_inline

# 省略配置 t_order_item 表规则...
# ...

# 配置 分片算法
spring.shardingsphere.rules.sharding.sharding-algorithms.database_inline.
type=INLINE
spring.shardingsphere.rules.sharding.sharding-algorithms.database_inline.props.
algorithm.expression=ds_${user_id % 2}
spring.shardingsphere.rules.sharding.sharding-algorithms.table_inline.type=INLINE
spring.shardingsphere.rules.sharding.sharding-algorithms.table_inline.props.
algorithm.expression=t_order_${order_id % 2}
```

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。如：

```
# 配置真实数据源
spring.shardingsphere.datasource.names=ds0,ds1

# 配置第 1 个数据源
spring.shardingsphere.datasource.ds0.jndi-name=java:comp/env/jdbc/ds0
# 配置第 2 个数据源
spring.shardingsphere.datasource.ds1.jndi-name=java:comp/env/jdbc/ds1

# 省略规则配置...
# ...
```

在 Spring 中使用 ShardingSphereDataSource

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

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

使用 Spring 命名空间

引入 Maven 依赖

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

规则配置

```
<?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:sharding="http://shardingsphere.apache.org/schema/shardingsphere/sharding
"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
                        http://www.springframework.org/schema/beans/spring-beans.
xsd
                        http://shardingsphere.apache.org/schema/shardingsphere/
sharding
                        http://shardingsphere.apache.org/schema/shardingsphere/
sharding/sharding.xsd
                        ">
    <!-- 配置真实数据源 -->
    <!-- 配置第 1 个数据源 -->
    <bean id="ds0" class="org.apache.commons.dbcp2.BasicDataSource" destroy-method=
"close">
        <property name="driverClassName" value="com.mysql.jdbc.Driver" />
        <property name="url" value="jdbc:mysql://localhost:3306/ds0" />
        <property name="username" value="root" />
        <property name="password" value="" />
    </bean>
    <!-- 配置第 2 个数据源 -->
    <bean id="ds1" class="org.apache.commons.dbcp2.BasicDataSource" destroy-method=
"close">
```

(下页继续)

(续上页)

```
        <property name="driverClassName" value="com.mysql.jdbc.Driver" />
        <property name="url" value="jdbc:mysql://localhost:3306/ds1" />
        <property name="username" value="root" />
        <property name="password" value="" />
    </bean>

    <!-- 配置分库策略 -->
    <sharding:sharding-algorithm id="dbShardingAlgorithm" type="INLINE">
        <properties>
            <prop key="algorithm.expression">ds$->{user_id % 2}</prop>
        </properties>
    </sharding:sharding-algorithm>
    <sharding:standard-strategy id="dbStrategy" sharding-column="user_id"
algorithm-ref="dbShardingAlgorithm" />

    <!-- 配置分表策略 -->
    <sharding:sharding-algorithm id="tableShardingAlgorithm" type="INLINE">
        <properties>
            <prop key="algorithm.expression">t_order$->{order_id % 2}</prop>
        </properties>
    </sharding:sharding-algorithm>
    <sharding:standard-strategy id="tableStrategy" sharding-column="user_id"
algorithm-ref="tableShardingAlgorithm" />

    <!-- 配置 ShardingSphereDataSource -->
    <sharding:data-source id="shardingDataSource">
        <!-- 配置分片规则 -->
        <sharding:sharding-rule data-source-names="ds0,ds1">
            <sharding:table-rules>
                <!-- 配置 t_order 表规则 -->
                <sharding:table-rule logic-table="t_order" actual-data-nodes="ds
$->{0..1}.t_order$->{0..1}" database-strategy-ref="dbStrategy" table-strategy-ref=
"tableStrategy" />
                <!-- 省略配置 t_order_item 表规则... -->
                <!-- ... -->
            </sharding:table-rules>
        </sharding:sharding-rule>
    </sharding:data-source>
</beans>
```

在 Spring 中使用 ShardingSphereDataSource

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

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

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。

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。

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:
```

(下页继续)

(续上页)

```
    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。通过此方式添加分片键值后，将跳过 SQL 解析和改写阶段，从而提高整体执行效率。

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()) {
            // ...
        }
    }
}
```

(下页继续)

(续上页)

```
// 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

获取 HintManager

与基于 Hint 的数据分片相同。

设置主库路由

- 使用 `hintManager.setMasterRouteOnly` 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.setMasterRouteOnly();
    try (ResultSet rs = preparedStatement.executeQuery()) {
        while (rs.next()) {
            // ...
        }
    }
}
```

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-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
@ShardingTransactionType(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-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 @ShardingTransactionType 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
@ShardingTransactionType(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。

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。

分布式治理

使用治理功能需要指定配置中心、注册中心和元数据中心。配置将全部存入配置中心，可以在每次启动时使用本地配置覆盖配置中心配置，也可以只通过配置中心读取配置。

使用 Java API

引入 Maven 依赖

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

<!-- 使用 ZooKeeper 时，需要引入此模块 -->
<dependency>
    <groupId>org.apache.shardingsphere</groupId>
    <artifactId>shardingsphere-orchestration-repository-zookeeper-curator</
artifactId>
    <version>${shardingsphere.version}</version>
</dependency>

<!-- 使用 Etcd 时，需要引入此模块 -->
<dependency>
    <groupId>org.apache.shardingsphere</groupId>
    <artifactId>shardingsphere-orchestration-repository-etcd</artifactId>
    <version>${shardingsphere.version}</version>
</dependency>
```

规则配置

以下示例将 ZooKeeper 作为配置中心和注册中心。

```
// 省略配置数据源以及规则
// ...

// 配置配置/注册/元数据中心
OrchestrationCenterConfiguration configuration = new OrchestrationCenterConfiguration(
"Zookeeper", "localhost:2181", new Properties());

// 配置治理
Map<String, CenterConfiguration> configurationMap = new HashMap<String,
CenterConfiguration>();
configurationMap.put("orchestration-shardingsphere-data-source", configuration);
```

(下页继续)

(续上页)

```
// 创建 OrchestrationShardingSphereDataSource
DataSource dataSource = OrchestrationShardingSphereDataSourceFactory.
createDataSource(
        createDataSourceMap(), createShardingRuleConfig(), new HashMap<String,
Object>(), new Properties(),
        new OrchestrationConfiguration("shardingsphere-orchestration",
configurationMap, true));
```

使用 OrchestrationShardingSphereDataSource

通过 OrchestrationShardingSphereDataSourceFactory 工厂创建的 OrchestrationShardingSphereDataSource 实现自 JDBC 的标准接口 DataSource。可通过 DataSource 选择使用原生 JDBC，或 JPA，MyBatis 等 ORM 框架。

以原生 JDBC 使用方式为例：

```
DataSource dataSource = OrchestrationShardingSphereDataSourceFactory.
createDataSource(
        createDataSourceMap(), createShardingRuleConfig(), new HashMap<String,
Object>(), new Properties(),
        new OrchestrationConfiguration("shardingsphere-orchestration",
configurationMap, true));
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()) {
            // ...
        }
    }
}
```

使用 YAML 配置

引入 Maven 依赖

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

(下页继续)

(续上页)

```
</dependency>

<!-- 使用 ZooKeeper 时，需要引入此模块 -->
<dependency>
    <groupId>org.apache.shardingsphere</groupId>
    <artifactId>shardingsphere-orchestration-repository-zookeeper-curator</
artifactId>
    <version>${shardingsphere.version}</version>
</dependency>

<!-- 使用 Etcd 时，需要引入此模块 -->
<dependency>
    <groupId>org.apache.shardingsphere</groupId>
    <artifactId>shardingsphere-orchestration-repository-etcd</artifactId>
    <version>${shardingsphere.version}</version>
</dependency>
```

规则配置

以下示例将 ZooKeeper 作为配置中心和注册中心。

```
orchestration:
  name: orchestration_ds
  registryCenter:
      type: Zookeeper
      serverLists: localhost:2181
  overwrite: true
```

```
// 创建 OrchestrationShardingSphereDataSource
DataSource dataSource = YamlOrchestrationShardingSphereDataSourceFactory.
createDataSource(yamlFile);
```

使用 OrchestrationShardingSphereDataSource

通过 YamlOrchestrationShardingSphereDataSourceFactory 工厂创建的 OrchestrationShardingSphere-DataSource 实现自 JDBC 的标准接口 DataSource。可通过 DataSource 选择使用原生 JDBC,或 JPA,MyBatis 等 ORM 框架。

以原生 JDBC 使用方式为例:

```
DataSource dataSource = YamlOrchestrationShardingSphereDataSourceFactory.
createDataSource(yamlFile);
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 (
```

(下页继续)

(续上页)

```
        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()) {
            // ...
        }
    }
}
```

使用 Spring Boot Starter

引入 Maven 依赖

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

<!-- 使用 ZooKeeper 时，需要引入此模块 -->
<dependency>
    <groupId>org.apache.shardingsphere</groupId>
    <artifactId>shardingsphere-orchestration-repository-zookeeper-curator</
artifactId>
    <version>${shardingsphere.version}</version>
</dependency>

<!-- 使用 Etcd 时，需要引入此模块 -->
<dependency>
    <groupId>org.apache.shardingsphere</groupId>
    <artifactId>shardingsphere-orchestration-repository-etcd</artifactId>
    <version>${shardingsphere.version}</version>
</dependency>
```

规则配置

```
spring.shardingsphere.orchestration.name=orchestration-spring-boot-shardingsphere-
test
spring.shardingsphere.orchestration.registry-center.type=Zookeeper
spring.shardingsphere.orchestration.registry-center.server-lists=localhost:2181
spring.shardingsphere.orchestration.additional-config-center.type=Zookeeper
```

(下页继续)

(续上页)

```
spring.shardingsphere.orchestration.additional-config-center.server-
lists=localhost:2182
spring.shardingsphere.orchestration.overwrite=true
```

在 Spring 中使用 OrchestrationShardingSphereDataSource

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

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

使用 Spring 命名空间

引入 Maven 依赖

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

<!-- 使用 ZooKeeper 时，需要引入此模块 -->
<dependency>
    <groupId>org.apache.shardingsphere</groupId>
    <artifactId>shardingsphere-orchestration-repository-zookeeper-curator</
artifactId>
    <version>${shardingsphere.version}</version>
</dependency>

<!-- 使用 Etcd 时，需要引入此模块 -->
<dependency>
    <groupId>org.apache.shardingsphere</groupId>
    <artifactId>shardingsphere-orchestration-repository-etcd</artifactId>
    <version>${shardingsphere.version}</version>
</dependency>
```

规则配置

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:orchestration="http://shardingsphere.apache.org/schema/shardingsphere/
orchestration"
       xmlns="http://www.springframework.org/schema/beans"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-
beans.xsd
                           http://shardingsphere.apache.org/schema/shardingsphere/
orchestration
                           http://shardingsphere.apache.org/schema/shardingsphere/
orchestration/orchestration.xsd">
     <util:properties id="instance-properties">
         <prop key="max-retries">3</prop>
         <prop key="operation-timeout-milliseconds">3000</prop>
     </util:properties>
     <orchestration:reg-center id="regCenter" type="Zookeeper" server-lists=
"localhost:2181" />
     <orchestration:config-center id="configCenter" type="ZooKeeper" server-lists=
"localhost:2182" />
     <orchestration:data-source id="shardingDatabasesTablesDataSource" data-source-
ref="realShardingDatabasesTablesDataSource" reg-center-ref="regCenter" config-
center-ref="configCenter" overwrite="true" />
     <orchestration:slave-data-source id="masterSlaveDataSource" data-source-
ref="realMasterSlaveDataSource" reg-center-ref="regCenter" config-center-ref=
"configCenter" overwrite="true" />
     <orchestration:data-source id="encryptDataSource" data-source-ref=
"realEncryptDataSource" reg-center-ref="regCenter" config-center-ref="configCenter"
overwrite="true" />
</beans>
```

在 Spring 中使用 OrchestrationShardingSphereDataSource

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

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

5.1.4 配置手册

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、多副本、数据加密、影子库等功能，并且能够叠加使用。

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 是最复杂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。

使用方式

创建简单数据源

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

```
// 构建数据源
Map<String, DataSource> dataSourceMap = // ...

// 构建配置规则
Collection<RuleConfiguration> configurations = // ...

// 构建属性配置
Properties props = // ...

DataSource dataSource = ShardingSphereDataSourceFactory.createDataSource(dataSourceMap,
 configurations, props);
```

创建携带治理功能的数据源

通过 OrchestrationShardingSphereDataSourceFactory 工厂创建的 OrchestrationShardingSphereDataSource 实现自 JDBC 的标准接口 DataSource。

```
// 构建数据源
Map<String, DataSource> dataSourceMap = // ...
```

(下页继续)

(续上页)

```
// 构建配置规则
Collection<RuleConfiguration> configurations = // ...

// 构建属性配置
Properties props = // ...

// 构建注册中心配置对象
OrchestrationConfiguration orchestrationConfig = // ...

DataSource dataSource = OrchestrationShardingSphereDataSourceFactory.
createDataSource(dataSourceMap, configurations, props, orchestrationConfig);
```

使用数据源

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

以原生 JDBC 使用方式为例：

```
DataSource dataSource = // 通过 Apache ShardingSphere 工厂创建的数据源
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()) {
            // ...
        }
    }
}
```

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：

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	•
actualDataSources (?)	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
shardingStrategy (?)	ShardingStrategyConfiguration	分片策略	使用默认分片策略
keyGenerateStrategy (?)	KeyGeneratorConfiguration	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.masterslave.api.config.MasterSlaveRuleConfiguration

可配置属性:

主从数据源配置

类名称:org.apache.shardingsphere.masterslave.api.config.rule.MasterSlaveDataSourceRuleConfiguration

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	•
masterDataSourceName	String	主库数据源名称	•
slaveDataSourceNames (+)	Collection<String>	从库数据源名称列表	•
loadBalancerName (?)	String	从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
shadowMappings	Map<String, String>	生产数据库与影子数据库的映射关系集合，key 为生产库名称，value 为影子库名称

多数据副本

配置入口

类名称：org.apache.shardingsphere.replica.api.config.ReplicaRuleConfiguration

TODO

分布式治理

配置项说明

治理

配置入口

类名称：org.apache.shardingsphere.orchestration.repository.api.config.OrchestrationConfiguration

可配置属性：

注册中心的类型可以为 Zookeeper 或 etcd。配置中心的类型可以为 Zookeeper 或 etcd、Apollo、Nacos。

治理实例配置

类名称：org.apache.shardingsphere.orchestration.repository.api.config.OrchestrationCenterConfiguration

可配置属性：

名称	数据类型	说明
type	String	治理实例类型，如：Zookeeper, etcd, Apollo, Nacos
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

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
operationTimeoutMilliseconds (?)	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
retryIntervalMilliseconds (?)	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

Apollo 属性配置

名称	数据类型	说明	默认值
appId (?)	String	Apollo appId	APOLLO_SHARDINGSPHERE
env (?)	String	Apollo env	DEV
clusterName (?)	String	Apollo clusterName	default
administrator (?)	String	Apollo administrator	空
token (?)	String	Apollo token	空
portalUrl (?)	String	Apollo portalUrl	空
connectTimeout (?)	int	连接超时毫秒数	1000 毫秒
readTimeout (?)	int	读取超时毫秒数	5000 毫秒

Nacos 属性配置

名称	数据类型	说明	默认值
group (?)	String	nacos group 配置	SHARDING_SPHERE_DEFAULT_GROUP
timeout (?)	long	nacos 获取数据超时毫秒数	3000 毫秒

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。

使用方式

创建简单数据源

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

```
// 指定 YAML 文件路径
File yamlFile = // ...

DataSource dataSource = YamlShardingSphereDataSourceFactory.createDataSource(yamlFile);
```

创建携带治理功能的数据源

通过 YamlOrchestrationShardingSphereDataSourceFactory 工厂创建的 OrchestrationShardingSphere-DataSource 实现自 JDBC 的标准接口 DataSource。

```
// 指定 YAML 文件路径
File yamlFile = // ...

DataSource dataSource = YamlOrchestrationShardingSphereDataSourceFactory.
createDataSource(yamlFile);
```

使用数据源

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

以原生 JDBC 使用方式为例：

```
DataSource dataSource = // 通过 Apache ShardingSphere 工厂创建的数据源
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()) {
            // ...
        }
    }
}
```

YAML 配置项

数据源配置

分为单数据源配置和多数据源配置。单数据源配置用于数据加密规则；多数据源配置用于分片、读写分离等规则。如果加密和分片等功能混合使用，则应该使用多数据源配置。

单数据源配置

配置示例

```
dataSource: !!org.apache.commons.dbcp2.BasicDataSource
  driverClassName: com.mysql.jdbc.Driver
  url: jdbc:mysql://127.0.0.1:3306/ds_name
  username: root
  password: root
```

配置项说明

```
dataSource: # <!! 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
```

多数据源配置

配置示例

```
dataSources:
  ds_0: !!org.apache.commons.dbcp2.BasicDataSource
    driverClassName: org.h2.Driver
    url: jdbc:h2:mem:ds_m;DB_CLOSE_DELAY=-1;DATABASE_TO_UPPER=false;MODE=MYSQL
    username: sa
    password:
  ds_1: !!org.apache.commons.dbcp2.BasicDataSource
    driverClassName: org.h2.Driver
    url: jdbc:h2:mem:ds_s;DB_CLOSE_DELAY=-1;DATABASE_TO_UPPER=false;MODE=MYSQL
    username: sa
    password:
```

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
```

规则配置

以规则别名开启配置，可配置多个规则。

配置示例

```
rules:
-! XXX_RULE_0
  xxx
-! XXX_RULE_1
  xxx
```

配置项说明

```
rules:
-! XXX_RULE # 规则别名，`-` 表示可配置多个规则
  # ... 具体的规则配置
```

更多详细配置请参见具体的规则配置部分。

属性配置

配置示例

```
props:
  xxx: xxx
```

配置项说明

```
props:
  xxx: xxx # 属性名称以及对应的值
```

更多详细配置请参见具体的规则配置部分。

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

数据分片

配置项说明

```
dataSources: # 省略数据源配置

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，以小数点分隔。多个表以逗号分隔，支持行表达式。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
          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
  bindingTables (+): # 绑定表规则列表
    - <logic_table_name_1, logic_table_name_2, ...>
  broadcastTables (+): # 广播表规则列表
    - <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
```

(下页继续)

(续上页)

```
      # ...

props:
  # ...
```

读写分离

配置项说明

```
dataSources: # 省略数据源配置

rules:
- !MASTER_SLAVE
  dataSources:
    <data-source-name> (+): # 读写分离逻辑数据源名称
      masterDataSourceName: # 主库数据源名称
      slaveDataSourceNames:
        - <slave-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

rules:
- !ENCRYPT
  tables:
    <table-name> (+): # 加密表名称
      columns:
        <column-name> (+): # 加密列名称
```

(下页继续)

(续上页)

```
          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
        # ...

props:
  # ...
```

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

rules:
- !SHADOW
  column: # 影子字段名
  shadowMappings: # 生产库与影子库的映射关系，key 为生产库名，value 为逻辑库名
    # ...

props:
  # ...
```

多数据副本

配置项说明

TODO

分布式治理

配置项说明

治理

```
orchestration:
  name: # 治理名称
  registryCenter: # 配置中心
    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
  additionalConfigCenter:
    type: # 治理持久化类型。如: Zookeeper, etcd, Nacos, Apollo
    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
```

Spring Boot Start 配置

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。

数据源配置

```
spring.shardingsphere.datasource.names= # 数据源名称，多数据源以逗号分隔

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

spring.shardingsphere.datasource.<datasource-name>.url= # 数据库 URL 连接
```

覆盖数据源 common 配置

```
spring.shardingsphere.datasource.<datasource-name>.url= # 数据库 URL 连接
spring.shardingsphere.datasource.<datasource-name>.type= # 数据库连接池类名称，覆盖 common 中的 type 配置
spring.shardingsphere.datasource.<datasource-name>.driver-class-name= # 数据库驱动类名，覆盖 common 中的 driver-class-name 配置
spring.shardingsphere.datasource.<datasource-name>.username= # 数据库用户名，覆盖 common 中的 username 配置
spring.shardingsphere.datasource.<datasource-name>.password= # 数据库密码 ，覆盖 common 中的 password 配置
spring.shardingsphere.datasource.<data-source-name>.xxx= # 数据库连接池的其它属性 ，覆盖 common 中其他属性配置
```

规则配置

```
spring.shardingsphere.rules.<rule-type>.xxx= # 规则配置
  # ... 具体的规则配置
```

更多详细配置请参见具体的规则配置部分。

属性配置

```
spring.shardingsphere.props.xxx.xxx= # 具体的属性配置
```

详情请参见属性配置。

数据分片

配置项说明

```
spring.shardingsphere.datasource.names= # 省略数据源配置

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
```

(下页继续)

(续上页)

```
# 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

# 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
```

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

spring.shardingsphere.rules.master-slave.data-sources.<master-slave-data-source-name>.master-data-source-name= # 主数据源名称
spring.shardingsphere.rules.master-slave.data-sources.<master-slave-data-source-name>.slave-data-source-names= # 从数据源名称，多个从数据源用逗号分隔
spring.shardingsphere.rules.master-slave.data-sources.<master-slave-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.master-slave.load-balancers.<load-balance-algorithm-name>.type= # 负载均衡算法类型
spring.shardingsphere.rules.master-slave.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。

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
```

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

spring.shardingsphere.rules.shadow.column= # 影子字段名称名称
spring.shardingsphere.rules.shadow.shadow-mappings.<product-data-source-name>= # 影子数据库名称
```

多数据副本

配置项说明

TODO

分布式治理

配置项说明

治理

```
spring.shardingsphere.orchestration.name= # 治理名称
spring.shardingsphere.orchestration.registry-center.type= # 治理持久化类型。如: Zookeeper, etcd, Apollo, Nacos
spring.shardingsphere.orchestration.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.orchestration.registry-center.props= # 其它配置
spring.shardingsphere.orchestration.additional-config-center.type= # 可选的配置中心类型。如: Zookeeper, etcd, Apollo, Nacos
spring.shardingsphere.orchestration.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.orchestration.additional-config-center.props= # 可选的配置中心其它配置
spring.shardingsphere.orchestration.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.
```

Spring 命名空间配置

简介

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

Spring 命名空间配置项

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
                           ">
    <bean id="ds0" class="org.apache.commons.dbcp2.BasicDataSource" destroy-method=
"close">
        <property name="driverClassName" value="com.mysql.jdbc.Driver" />
        <property name="url" value="jdbc:mysql://localhost:3306/ds0" />
        <property name="username" value="root" />
        <property name="password" value="" />
    </bean>

    <bean id="ds1" class="org.apache.commons.dbcp2.BasicDataSource" destroy-method=
"close">
        <property name="driverClassName" value="com.mysql.jdbc.Driver" />
        <property name="url" value="jdbc:mysql://localhost:3306/ds1" />
        <property name="username" value="root" />
        <property name="password" value="" />
    </bean>

    <!-- 配置规则，更多详细配置请参见具体的规则配置部分。 -->
    <!-- ... -->

    <shardingsphere:data-source id="shardingDataSource" data-source-names="ds0,ds1"
rule-refs="..." >
        <props>
            <prop key="xxx.xxx">${xxx.xxx}</prop>
```

(下页继续)

(续上页)

```
        </props>
    </shardingsphere:data-source>
</beans>
```

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
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
props (?)	标签	属性配置，详情请参见属性配置

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-rule />

<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/masterslave/master-slave-5.0.0.xsd

<master-slave: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

<master-slave:data-source-rule />

名称	类型	说明
id	属性	读写分离数据源规则名称
master-data-source-name	属性	主数据源名称
slave-data-source-names	属性	从数据源名称，多个从数据源用逗号分隔
load-balance-algorithm-ref	属性	负载均衡算法名称

<master-slave: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
id	属性	Spring Bean Id
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

多数据副本

配置项说明

TODO

分布式治理

配置项说明

治理

```
<?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:orchestration="http://shardingsphere.apache.org/schema/shardingsphere/
orchestration"
```

(下页继续)

(续上页)

```
        xsi:schemaLocation="http://www.springframework.org/schema/beans
                            http://www.springframework.org/schema/beans/spring-
beans.xsd
                            http://shardingsphere.apache.org/schema/shardingsphere/
orchestration
                            http://shardingsphere.apache.org/schema/shardingsphere/
orchestration/orchestration.xsd
">

    <orchestration:reg-center id="regCenter" type="ZooKeeper" server-lists=
"localhost:2181" />
    <orchestration:config-center id="configCenter" type="ZooKeeper" server-lists=
"localhost:2182" />
    <orchestration:data-source id="shardingDatabasesTablesDataSource" data-source-
ref="realShardingDatabasesTablesDataSource" reg-center-ref="regCenter" config-
center-ref="configCenter" overwrite="true" />
</beans>
```

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

名称	类型	说明
id	属性	注册中心实例名称
type	属性	注册中心类型。如：ZooKeeper, etcd
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

名称	类型	说明
id	属性	配置中心实例名称
type	属性	配置中心类型。如：ZooKeeper, etcd, Apollo, Nacos
server-lists	属性	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
props (?)	属性	配置本实例需要的其他参数，例如 ZooKeeper 的连接参数等

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。

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
datetime.lower	String	分片的起始时间范围，时间戳格式：yyyy-MM-dd HH:mm:ss
datetime.upper	String	分片的结束时间范围，时间戳格式：yyyy-MM-dd HH:mm:ss
sharding.seconds	long	单一分片所能承载的最大时间，单位：秒

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

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

Apache ShardingSphere 暂无内置复合分片算法。

Hint 分片算法

Apache ShardingSphere 暂无内置 Hint 分片算法。

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

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

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

5.1.5 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。

5.2 ShardingSphere-Proxy

5.2.1 简介

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 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

5.2.2 对比

	ShardingSphere-JDBC	*ShardingSphere-Proxy*	*ShardingSphere-Sidecar*
数据库	任意	`MySQL/PostgreSQL`	MySQL/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-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。

5.2.3 使用手册

本章节将介绍 ShardingSphere-Proxy 相关使用。更多使用细节请参见使用示例。

Proxy 启动

启动步骤

1. 下载 ShardingSphere-Proxy 的最新发行版。
2. 如果使用 docker，可以执行 `docker pull shardingsphere/shardingsphere-proxy` 获取镜像。详细信息请参考Docker 镜像。
3. 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。
4. 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。

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`

使用自定义分片算法

当用户需要使用自定义的分片算法类时，无法再通过简单的行表达式在 YAML 文件进行配置。可通过以下方式配置使用自定义分片算法。

1. 实现 `ShardingAlgorithm` 接口定义的算法实现类。
2. 将上述 Java 文件打包成 jar 包。
3. 将上述 jar 包拷贝至 ShardingSphere-Proxy 解压后的 `conf/lib-ext` 目录。
4. 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。

分布式治理

ShardingSphere-Proxy 支持使用 SPI 方式接入分布式治理，实现配置和元数据统一管理以及实例熔断和从库禁用等功能。

Zookeeper

ShardingSphere-Proxy 默认提供了 Zookeeper 解决方案，实现了配置中心、注册中心和元数据中心功能。配置规则同 ShardingSphere-JDBC YAML 保持一致。

其他第三方组件

详情请参考支持的第三方组件。

1. 使用 SPI 方式实现相关逻辑编码，并将生成的 jar 包复制至 ShardingSphere-Proxy 的 lib 目录。
2. 按照配置规则进行配置，即可使用。

分布式事务

ShardingSphere-Proxy 接入的分布式事务 API 同 ShardingSphere-JDBC 保持一致，支持 LOCAL，XA，BASE 类型的事务。

XA 事务

ShardingSphere-Proxy 原生支持 XA 事务，默认的事务管理器为 Atomikos。可以通过在 ShardingSphere-Proxy 的 conf 目录中添加 `jta.properties` 来定制化 Atomikos 配置项。具体的配置规则请参考 Atomikos 的官方文档。

BASE 事务

BASE 目前没有集成至 ShardingSphere-Proxy 的二进制发布包中，使用时需要将实现了 `ShardingTransactionManager` SPI 的 jar 拷贝至 conf/lib 目录，然后切换事务类型为 BASE。

SCTL

SCTL (ShardingSphere Control Language) 为 ShardingSphere 特有的控制语句，可以在运行时修改和查询 ShardingSphere-Proxy 的状态，目前支持的语法为：

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。

5.2.4 配置手册

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 只提供基于 YAML 的配置方式。通过配置，应用开发者可以灵活的使用数据分片、读写分离、多副本、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。

数据源配置

配置项说明

```yaml
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。

```yaml
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, 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。
```

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

YAML 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

5.2.5 Docker 镜像

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，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_user_name} -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 的 ShardingSphere-Proxy?

回答：将 `mysql-connector.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。

5.3 ShardingSphere-Sidecar

5.3.1 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。

5.3.2 对比

	ShardingSphere-JDBC	*ShardingSphere-Proxy*	*ShardingSphere-Sidecar*
数据库	任意	MySQL/PostgreSQL	`MySQL/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。

图 3: ShardingSphere-Sidecar Architecture

5.4 ShardingSphere-Scaling

5.4.1 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 **Alpha** 版本。

5.4.2 运行部署

部署启动

1. 执行以下命令，编译生成 ShardingSphere-Scaling 二进制包：

```
git clone https://github.com/apache/shardingsphere.git;
cd shardingsphere;
mvn clean install -Prelease;
```

发布包所在目录为:`/shardingsphere-distribution/shardingsphere-scaling-distribution/target/apache-shardingsphere-${latest.release.version}-shardingsphere-scaling-bin.tar.gz`。

2. 解压缩发布包，修改配置文件 `conf/server.yaml`，这里主要修改启动端口，保证不与本机其他端口冲突，其他值保持默认即可：

```
port: 8888
blockQueueSize: 10000
pushTimeout: 1000
workerThread: 30
```

3. 编译生成 ShardingSphere-Scaling：

```
sh bin/start.sh
```

4. 查看日志 `logs/stdout.log`，确保启动成功。
5. 使用 curl 命令再次确认正常运行。

```
curl -X GET http://localhost:8888/shardingscaling/job/list
```

响应应为：

```
{"success":true,"errorCode":0,"errorMsg":null,"model":[]}
```

结束 ShardingSphere-Scaling

```
sh bin/stop.sh
```

应用配置项

应用现有配置项如下，相应的配置可在 `conf/server.yaml` 中修改：

名称	说明	默认值
port	HTTP 服务监听端口	8888
blockQueueSize	数据传输通道队列大小	10000
pushTimeout	数据推送超时时间，单位：毫秒	1000
workerThread	工作线程池大小，允许同时运行的迁移任务线程数	30

5.4.3 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	是否支持
MySQL(5.1.15 ~ 5.7.x)	ShardingSphere-Proxy	是
PostgreSQL(9.4 ~)	ShardingSphere-Proxy	是

注意：

如果后端连接 MySQL 数据库,需要下载MySQL Connector/J,解压缩后,将 mysql-connector-java-5.1.47.jar 拷贝到 `${shardingsphere-scaling}\lib` 目录。

权限要求

MySQL 需要开启 `binlog`，`binlog format` 为 Row 模式，且迁移时所使用用户需要赋予 Replication 相关权限。

```
+----------------------------------------+----------------------------------------+
| Variable_name                          | Value                                  |
+----------------------------------------+----------------------------------------+
| log_bin                                | ON                                     |
| binlog_format                          | ROW                                    |
+----------------------------------------+----------------------------------------+
```

(下页继续)

(续上页)

```
+-------------------------------------------------------------------------------+
|Grants for ${username}@${host}                                                 |
+-------------------------------------------------------------------------------+
|GRANT REPLICATION SLAVE, REPLICATION CLIENT ON *.* TO ${username}@${host}      |
|.......                                                                        |
+-------------------------------------------------------------------------------+
```

PostgreSQL 需要开启 test_decoding

API 接口

弹性迁移组件提供了简单的 HTTP API 接口

创建迁移任务

接口描述：POST /shardingscaling/job/start

请求体：

示例：

```
curl -X POST \
  http://localhost:8888/shardingscaling/job/start \
  -H 'content-type: application/json' \
  -d '{
   "ruleConfiguration": {
      "sourceDatasource": "ds_0: !!org.apache.shardingsphere.orchestration.core.
common.yaml.config.YamlDataSourceConfiguration\n  dataSourceClassName: com.zaxxer.
hikari.HikariDataSource\n  props:\n    jdbcUrl: jdbc:mysql://127.0.0.1:3306/test?
serverTimezone=UTC&useSSL=false\n    username: root\n    password: '\''123456'\''\n
   connectionTimeout: 30000\n    idleTimeout: 60000\n    maxLifetime: 1800000\n
maxPoolSize: 50\n    minPoolSize: 1\n    maintenanceIntervalMilliseconds: 30000\n
 readOnly: false\n",
      "sourceRule": "defaultDatabaseStrategy:\n  inline:\n    algorithmExpression:
ds_${user_id % 2}\n    shardingColumn: user_id\ntables:\n  t1:\n
actualDataNodes: ds_0.t1\n    keyGenerateStrategy:\n      column: order_id\n
     type: SNOWFLAKE\n    logicTable: t1\n    tableStrategy:\n      inline:\
n        algorithmExpression: t1\n        shardingColumn: order_id\n  t2:\n
actualDataNodes: ds_0.t2\n    keyGenerateStrategy:\n      column: order_item_id\n
     type: SNOWFLAKE\n    logicTable: t2\n    tableStrategy:\n      inline:\n
 algorithmExpression: t2\n        shardingColumn: order_id\n",
      "destinationDataSources": {
         "name": "dt_0",
         "password": "123456",
         "url": "jdbc:mysql://127.0.0.1:3306/test2?serverTimezone=UTC&useSSL=false
",
         "username": "root"
```

(下页继续)

(续上页)

```
      }
   },
   "jobConfiguration": {
      "concurrency": 3
   }
}'
```

返回信息:

```
{
   "success": true,
   "errorCode": 0,
   "errorMsg": null,
   "model": null
}
```

查询迁移任务进度

接口描述：GET /shardingscaling/job/progress/{jobId}

示例:

```
curl -X GET \
  http://localhost:8888/shardingscaling/job/progress/1
```

返回信息:

```
{
   "success": true,
   "errorCode": 0,
   "errorMsg": null,
   "model": {
        "id": 1,
        "jobName": "Local Sharding Scaling Job",
        "status": "RUNNING/STOPPED"
        "syncTaskProgress": [{
            "id": "127.0.0.1-3306-test",
            "status": "PREPARING/MIGRATE_HISTORY_DATA/SYNCHRONIZE_REALTIME_DATA/
STOPPING/STOPPED",
            "historySyncTaskProgress": [{
                "id": "history-test-t1#0",
                "estimatedRows": 41147,
                "syncedRows": 41147
            }, {
                "id": "history-test-t1#1",
                "estimatedRows": 42917,
                "syncedRows": 42917
```

(下页继续)

(续上页)

```
            }, {
                "id": "history-test-t1#2",
                "estimatedRows": 43543,
                "syncedRows": 43543
            }, {
                "id": "history-test-t2#0",
                "estimatedRows": 39679,
                "syncedRows": 39679
            }, {
                "id": "history-test-t2#1",
                "estimatedRows": 41483,
                "syncedRows": 41483
            }, {
                "id": "history-test-t2#2",
                "estimatedRows": 42107,
                "syncedRows": 42107
            }],
            "realTimeSyncTaskProgress": {
                "id": "realtime-test",
                "delayMillisecond": 1576563771372,
                "position": {
                    "filename": "ON.000007",
                    "position": 177532875,
                    "serverId": 0
                }
            }
        }]
    }
}
```

查询所有迁移任务

接口描述：GET /shardingscaling/job/list

示例：

```
curl -X GET \
  http://localhost:8888/shardingscaling/job/list
```

返回信息：

```
{
  "success": true,
  "errorCode": 0,
  "model": [
    {
      "jobId": 1,
```

(下页继续)

(续上页)

```
      "jobName": "Local Sharding Scaling Job",
      "status": "RUNNING"
    }
  ]
}
```

停止迁移任务

接口描述：POST /shardingscaling/job/stop

请求体：

Parameter	Describe
jobId	job id

示例：

```
curl -X POST \
  http://localhost:8888/shardingscaling/job/stop \
  -H 'content-type: application/json' \
  -d '{
   "jobId":1
}'
```

返回信息：

```
{
   "success": true,
   "errorCode": 0,
   "errorMsg": null,
   "model": null
}
```

通过 UI 界面来操作

ShardingSphere-Scaling 与 ShardingSphere-UI 集成了用户界面，所以上述所有任务相关的操作都可以通过 UI 界面点点鼠标来实现，当然本质上还是调用了上述基本接口。

更多信息请参考 ShardingSphere-UI 项目。

5.5 ShardingSphere-UI

5.5.1 简介

ShardingSphere-UI 是 ShardingSphere 的一个简单而有用的 web 管理控制台。它用于帮助用户更简单的使用 ShardingSphere 的相关功能，目前提供注册中心管理、动态配置管理、数据库编排等功能。

项目结构上采取了前后端分离的方式, 前端使用 Vue 框架, 后端采用 Spring Boot 框架。使用标准的 Maven 方式进行打包，部署，同时也可以采用前后端分离的方式本地运行，方便开发调试。

5.5.2 使用手册

导览

本章节将介绍 ShardingSphere-UI 相关使用。

部署运行

二进制运行

1. `git clone https://github.com/apache/shardingsphere.git`;
2. 运行 `mvn clean install -Prelease`;
3. 获取安装包 `/shardingsphere-ui/shardingsphere-ui-distribution/target/apache-shardingsphere-${latest.release.version}-shardingsphere-ui-bin.tar.gz`;
4. 解压缩后运行 `bin/start.sh`;
5. 访问 `http://localhost:8088/`。

源码调试模式

ShardingSphere-UI 采用前后端分离的方式。

后端

1. 后端程序执行入口为 `org.apache.shardingsphere.ui.Bootstrap`;
2. 访问 `http://localhost:8088/`。

前端

1. 进入 `shardingsphere-ui-frontend/` 目录;
2. 执行 `npm install`;
3. 执行 `npm run dev`;
4. 访问 `http://localhost:8080/`。

配置

ShardingSphere-UI 的配置文件为 `conf/application.properties`, 它由两部分组成。

1. 程序监听端口;
2. 登录身份验证信息。

```
server.port=8088

user.admin.username=admin
user.admin.password=admin
```

注意事项

1. 若使用 maven 构建后，再进行本地运行前端项目时，可能因为 node 版本不一致导致运行失败，可以清空 `node_modules/` 目录后重新运行。错误日志如下:

```
ERROR  Failed to compile with 17 errors
error  in ./src/views/orchestration/module/instance.vue?vue&type=style&index=0&
id=9e59b740&lang=scss&scoped=true&
Module build failed (from ./node_modules/sass-loader/dist/cjs.js):
Error: Missing binding /shardingsphere/shardingsphere-ui/shardingsphere-ui-
frontend/node_modules/node-sass/vendor/darwin-x64-57/binding.node
Node Sass could not find a binding for your current environment: OS X 64-bit with
Node.js 8.x
Found bindings for the following environments:
  - OS X 64-bit with Node.js 6.x
This usually happens because your environment has changed since running `npm
install`.
Run `npm rebuild node-sass` to download the binding for your current environment.
```

配置中心

配置中心配置

首先需要添加并激活配置中心。可以添加多个配置中心，但只能有一个处于激活状态，后面的规则配置功能是针对当前已激活的配置中心进行操作。目前提供 Zookeeper 的支持，后续会添加第三方配置中心的支持。

规则配置

- 添加激活配置中心后，可以获取当前配置中心中所有数据源的相关配置，包括数据分片，读写分离、Properties 配置等。
- 可以通过 YAML 格式对相关配置信息进行修改。
- 点击 + 按钮可以添加新的数据源和分片规则。

注册中心

注册中心配置

首先需要添加并激活注册中心。可以添加多个注册中心，但只能有一个处于激活状态，后面的运行状态功能都是针对当前已激活的注册中心进行操作。目前提供 Zookeeper 的支持，后续会添加第三方注册中心的支持。

运行状态

- 添加激活注册中心后，可以查看当前注册中心所有运行实例信息。
- 可以通过操作按钮对运行实例进行熔断与恢复操作。
- 可以查看所有从库信息，并进行从库禁用与恢复操作。

6 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。

6.1 SQL 解析

6.1.1 SQLParserConfiguration

SPI 名称	详细说明
SQLParserConfiguration	用于规定用于解析 SQL 的 ANTLR 语法文件及其语法树访问器

已知实现类	详细说明
MySQLParserConfiguration	基于 MySQL 的 SQL 解析器实现
PostgreSQLParserConfiguration	基于 PostgreSQL 的 SQL 解析器实现
SQLServerParserConfiguration	基于 SQLServer 的 SQL 解析器实现
OracleParserConfiguration	基于 Oracle 的 SQL 解析器实现
SQL92ParserConfiguration	基于 SQL92 的 SQL 解析器实现

6.1.2 ParsingHook

SPI 名称	详细说明
ParsingHook	用于 SQL 解析过程追踪

已知实现类	详细说明
OpenTracingParsingHook	使用 OpenTracing 协议追踪 SQL 解析过程

6.2 配置

6.2.1 ShardingSphereRuleBuilder

SPI 名称	详细说明
ShardingSphereRuleBuilder	用于将用户配置转化为规则对象

已知实现类	详细说明
ShardingRuleBuilder	用于将分片用户配置转化为分片规则对象
MasterSlaveRuleBuilder	用于将读写分离用户配置转化为读写分离规则对象
ReplicaRuleBuilder	用于将多副本用户配置转化为多副本规则对象
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

6.2.2 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
ShardingRuleConfigurationYamlSwapper	用于将分片的 YAML 配置转化为分片标准配置
MasterSlaveRuleConfigurationYamlSwapper	用于将读写分离的 YAML 配置转化为读写分离标准配置
ReplicaRuleConfigurationYamlSwapper	用于将多副本的 YAML 分片配置转化为多副本标准配置
EncryptRuleConfigurationYamlSwapper	用于将加密的 YAML 分片配置转化为加密标准配置
ShadowRuleConfigurationYamlSwapper	用于将影子库的 YAML 分片配置转化为影子库标准配置

6.2.3 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
NoneShardingStrategyConfigurationYamlConstruct	用于将不分片策略对象和 YAML 相互转化

6.3 内核

6.3.1 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

6.3.2 RuleMetaDataLoader

SPI 名称	详细说明
RuleMetaDataLoader	用于元数据初始化

已知实现类	详细说明
ShardingMetaDataLoader	用于分片元数据初始化
EncryptMetaDataLoader	用于加密元数据初始化

6.3.3 RuleMetaDataDecorator

SPI 名称	详细说明
RuleMetaDataDecorator	用于元数据更新

已知实现类	详细说明
ShardingMetaDataDecorator	用于分片元数据更新
EncryptMetaDataDecorator	用于加密元数据更新

6.3.4 RouteDecorator

SPI 名称	详细说明
RouteDecorator	用于处理路由结果

已知实现类	详细说明
ShardingRouteDecorator	用于处理分片路由结果
MasterSlaveRouteDecorator	用于处理读写分离路由结果
ReplicaRouteDecorator	用于处理多副本路由结果
ShadowRouteDecorator	用于处理影子库路由结果

6.3.5 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
ShadowSQLRewriteContextDecorator	用于处理影子库 SQL 改写结果

6.3.6 ExecuteGroupDecorator

SPI 名称	详细说明
ExecuteGroupDecorator	用于修改数据节点分组结果

已知实现类	详细说明
ReplicaExecuteGroupDecorator	用于多副本数据节点分组

6.3.7 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
OpenTracingSQLExecutionHook	基于 OpenTracing 的 SQL 执行过程监听器

6.3.8 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

6.4 数据分片

6.4.1 ShardingAlgorithm

SPI 名称	详细说明
ShardingAlgorithm	分片算法

已知实现类	详细说明
InlineShardingAlgorithm	基于行表达式的分片算法
ModShardingAlgorithm	基于取模的分片算法
HashModShardingAlgorithm	基于哈希取模的分片算法
FixedIntervalShardingAlgorithm	基于固定时间范围的分片算法
MutableIntervalShardingAlgorithm	基于可变时间范围的分片算法
VolumeBasedRangeShardingAlgorithm	基于分片容量的范围分片算法
BoundaryBasedRangeShardingAlgorithm	基于分片边界的范围分片算法

6.4.2 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

6.4.3 TimeService

SPI 名称	详细说明
TimeService	获取当前时间进行路由

已知实现类	详细说明
DefaultTimeService	从应用系统时间中获取当前时间进行路由
DatabaseTimeServiceDelegate	从数据库中获取当前时间进行路由

6.4.4 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

6.5 读写分离

6.5.1 MasterSlaveLoadBalanceAlgorithm

SPI 名称	详细说明
MasterSlaveLoadBalanceAlgorithm	读库负载均衡算法

已知实现类	详细说明
RoundRobinMasterSlaveLoadBalanceAlgorithm	基于轮询的读库负载均衡算法
RandomMasterSlaveLoadBalanceAlgorithm	基于随机的读库负载均衡算法

6.6 数据加密

6.6.1 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

6.6.2 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	

6.7 分布式事务

6.7.1 ShardingTransactionManager

SPI 名称	详细说明
ShardingTransactionManager	分布式事务管理器

已知实现类	详细说明
XAShardingTransactionManager	基于 XA 的分布式事务管理器
SeataATShardingTransactionManager	基于 Seata 的分布式事务管理器

6.7.2 XATransactionManager

SPI 名称	详细说明
XATransactionManager	XA 分布式事务管理器

已知实现类	详细说明
AtomikosTransactionManager	基于 Atomikos 的 XA 分布式事务管理器
NarayanaXATransactionManager	基于 Narayana 的 XA 分布式事务管理器
BitronixXATransactionManager	基于 Bitronix 的 XA 分布式事务管理器

6.7.3 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
MySQLXADataSourceDefinitio n	非 XA 的 MySQL 数据源自动转化为 XA 的 MySQL 数据源
MariaDBXADataSourceDefinit ion	非 XA 的 MariaDB 数据源自动转化为 XA 的 MariaDB 数据源
PostgreSQLXADataSourceDefi nition	非 XA 的 PostgreSQL 数据源自动转化为 XA 的 PostgreSQL 数据源
OracleXADataSourceDefiniti on	非 XA 的 Oracle 数据源自动转化为 XA 的 Oracle 数据源
SQLServerXADataSourceDefin ition	非 XA 的 SQLServer 数据源自动转化为 XA 的 SQLServer 数据源
H2XADataSourceDefinition	非 XA 的 H2 数据源自动转化为 XA 的 H2 数据源

6.7.4 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

6.8 分布式治理

6.8.1 ConfigurationRepository

SPI 名称	详细说明
ConfigurationRepository	配置中心

已知实现类	详细说明
CuratorZookeeperRepository	基于 ZooKeeper 的配置中心
EtcdRepository	基于 etcd 的配置中心
NacosRepository	基于 Nacos 的配置中心
ApolloRepository	基于 Apollo 的配置中心

6.8.2 RegistryRepository

SPI 名称	详细说明
RegistryRepository	注册中心

已知实现类	详细说明
CuratorZookeeperRepository	基于 ZooKeeper 的注册中心
EtcdRepository	基于 etcd 的注册中心

6.8.3 RootInvokeHook

SPI 名称	详细说明
RootInvokeHook	请求调用入口追踪

已知实现类	详细说明
OpenTracingRootInvokeHook	基于 OpenTracing 协议的请求调用入口追踪

6.8.4 MetricsTrackerManager

SPI 名称	详细说明
MetricsTrackerManager	度量指标追踪

已知实现类	详细说明
PrometheusMetricsTrackerManager	基于 Prometheus 的度量指标追踪

6.9 弹性伸缩

6.9.1 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

6.10 Proxy

6.10.1 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
MySQLProtocolFrontendEngine	基于 MySQL 的数据库协议实现
PostgreSQLProtocolFrontendEngine	基于 PostgreSQL 的 SQL 解析器实现

6.10.2 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

7 下载

7.1 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。

7.1.1 Apache ShardingSphere - 版本: 4.1.1 (发布日期: Jun 5, 2020)

- 源码: [SRC] [ASC] [SHA512]
- ShardingSphere-JDBC 二进制包: [TAR] [ASC] [SHA512]
- ShardingSphere-Proxy 二进制包: [TAR] [ASC] [SHA512]
- ShardingSphere-Scaling 二进制包: [TAR] [ASC] [SHA512]

7.1.2 ShardingSphere UI - 版本: 4.1.1 (发布日期: Jun 9, 2020)

- 源码: [SRC] [ASC] [SHA512]
- ShardingSphere-UI 二进制包: [TAR] [ASC] [SHA512]

7.2 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。

7.3 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
```

8 FAQ

8.1 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。

8.2 阅读源码时为什么会出现编译错误?

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。

shardingsphere-orchestration-reg 模块需要先执行 `mvn install` 命令，根据 protobuf 文件生成 gRPC 相关的 java 文件。

8.3 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:'\master'**-slave.xsd，只需确保 jar 包中该文件存在即可。

8.4 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 `$->{...}`。

8.5 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)。

8.6 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:

是的。因为 ShardingSphere 是将多个数据源合并为一个统一的逻辑数据源。因此即使不分库分表的部分，不配置分片规则 ShardingSphere 即无法精确的断定应该路由至哪个数据源。但是 ShardingSphere 提供了两种变通的方式，有助于简化配置。

方法 1: 配置 default-data-source, 凡是在默认数据源中的表可以无需配置在分片规则中, ShardingSphere 将在找不到分片数据源的情况下将表路由至默认数据源。

方法 2：将不参与分库分表的数据源独立于 ShardingSphere 之外，在应用中使用多个数据源分别处理分片和不分片的情况。

8.7 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。

8.8 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。

8.9 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;
```

8.10 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：

com.dangdang.ddframe.rdb.sharding.merger.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;
}
```

8.11 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。

8.12 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

8.13 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-

8.14 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
```

8.15 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。

8.16 为什么我实现了 `ShardingKeyGenerator` 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 `ShardingKeyGenerator` 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。

8.17 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。

8.18 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 `default dataSource` 的 table 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）。

8.19 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 以上版本。
2. 将配置项 `allow.range.query.with.inline.sharding` 设置为 true 即可（默认为 false）。
3. 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。

8.20 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，sharing-jdbc 自己会创建数据连接池。

8.21 在使用 sharing-proxy 的时候，如何动态在 ShardingSphere-UI 上添加新的 logic schema?

回答:

1. 4.1.0 之前的版本不支持动态添加或删除 logic schema 的功能，例如一个 proxy 启动的时候有 2 个 logic schema，就会一直持有这 2 个 schema，只能感知这两个 schema 内部的表和 rule 的变更事件。
2. 4.1.0 版本支持在 ShardingSphere-UI 或直接在 zookeeper 上增加新的 logic schema，删除 logic schema 的功能计划在 5.0.0 版本支持。

8.22 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 mysql server，所以首选支持 mysql 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。建议选择特定版本的工具或者打开特定参数，例如使用 Navicat 11.1.13 版本 (不建议 12.x)，使用 IDEA/DataGrip 时打开 `introspect using JDBC metadata` 选项。

8.23 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)`。