

一、ShardingSphere 介绍

1. Apache ShardingSphere

Apache ShardingSphere 是一套开源的分布式数据库解决方案组成的生态圈，它由 JDBC、Proxy 和 Sidecar（规划中）这 3 款既能够独立部署，又支持混合部署配合使用的产品组成。它们均提供标准化的数据水平扩展、分布式事务和分布式治理等功能，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。

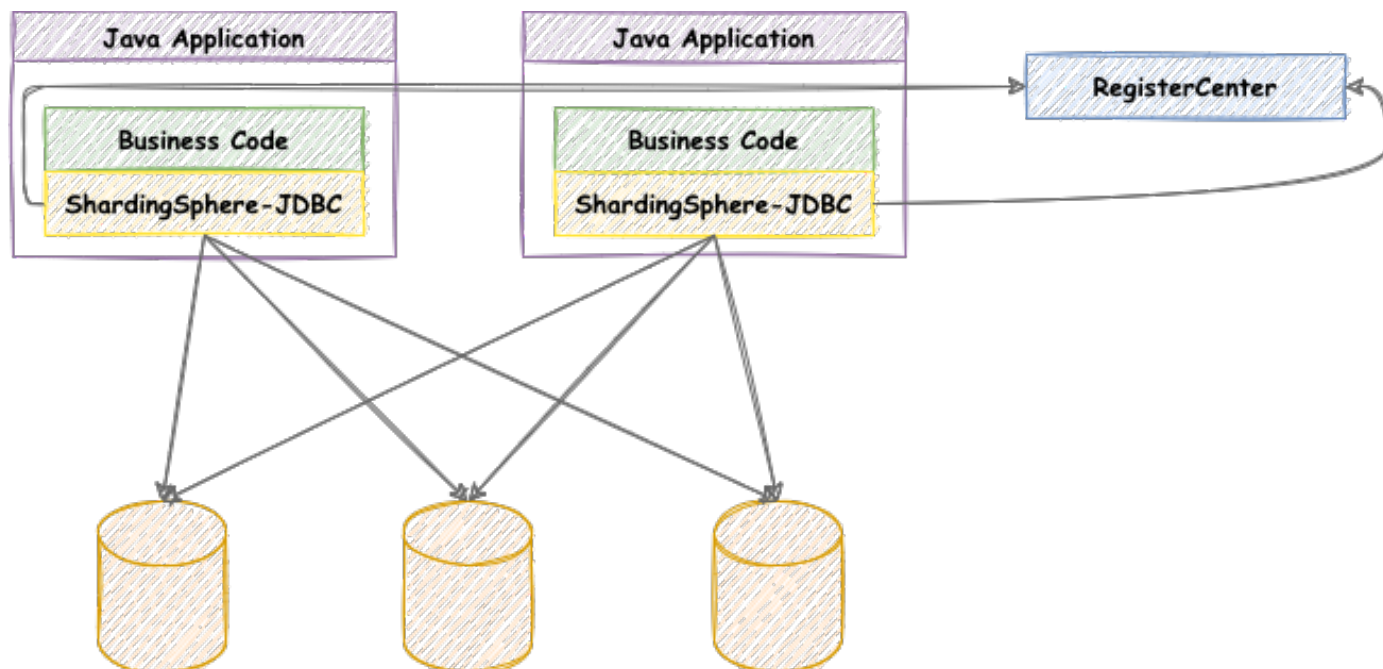
Apache ShardingSphere 旨在充分合理地在分布式的场景下利用关系型数据库的计算和存储能力，而并非实现一个全新的关系型数据库。关系型数据库当今依然占有巨大市场份额，是企业核心系统的基石，未来也难于撼动，我们更加注重在原有基础上提供增量，而非颠覆。

ShardingSphere 已于 2020 年 4 月 16 日成为 Apache 软件基金会的顶级项目。

2. ShardingSphere-JDBC

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

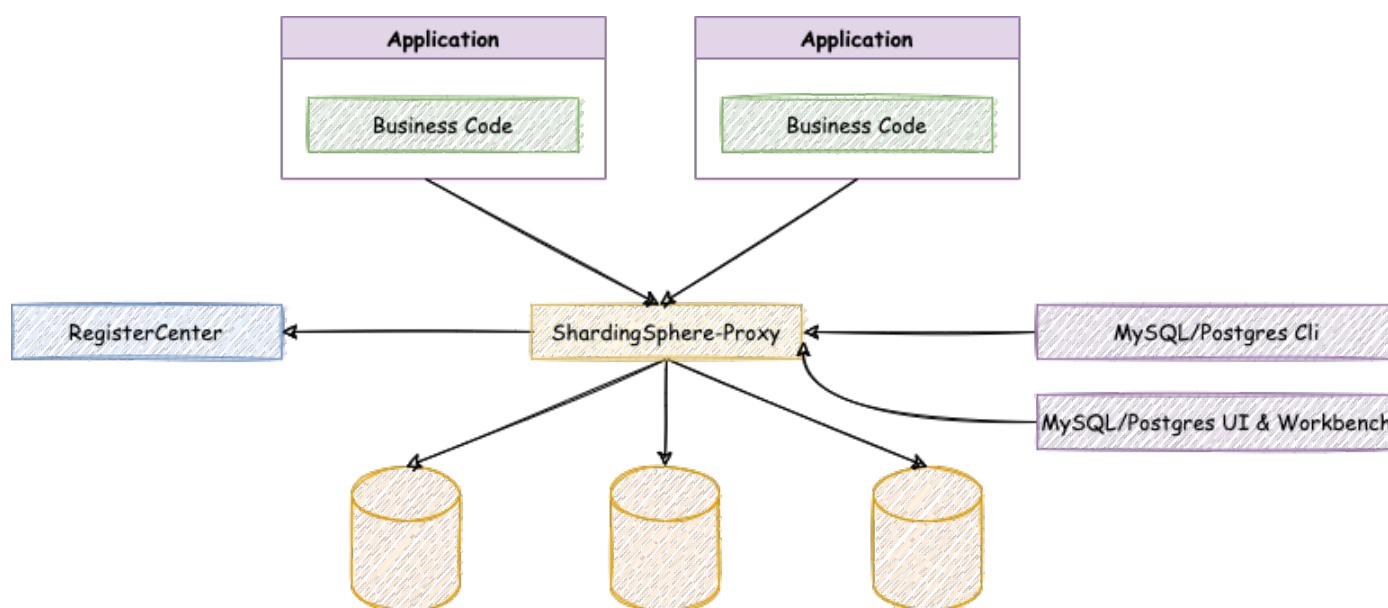
- 适用于任何基于 JDBC 的 ORM 框架，如：JPA, Hibernate, Mybatis, Spring JDBC Template 或直接使用 JDBC；
- 支持任何第三方的数据库连接池，如：DBCP, C3P0, BoneCP, HikariCP 等；
- 支持任意实现 JDBC 规范的数据库，目前支持 MySQL, PostgreSQL, Oracle, SQLServer 以及任何可使用 JDBC 访问的数据库。



3.ShardingSphere-Proxy

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

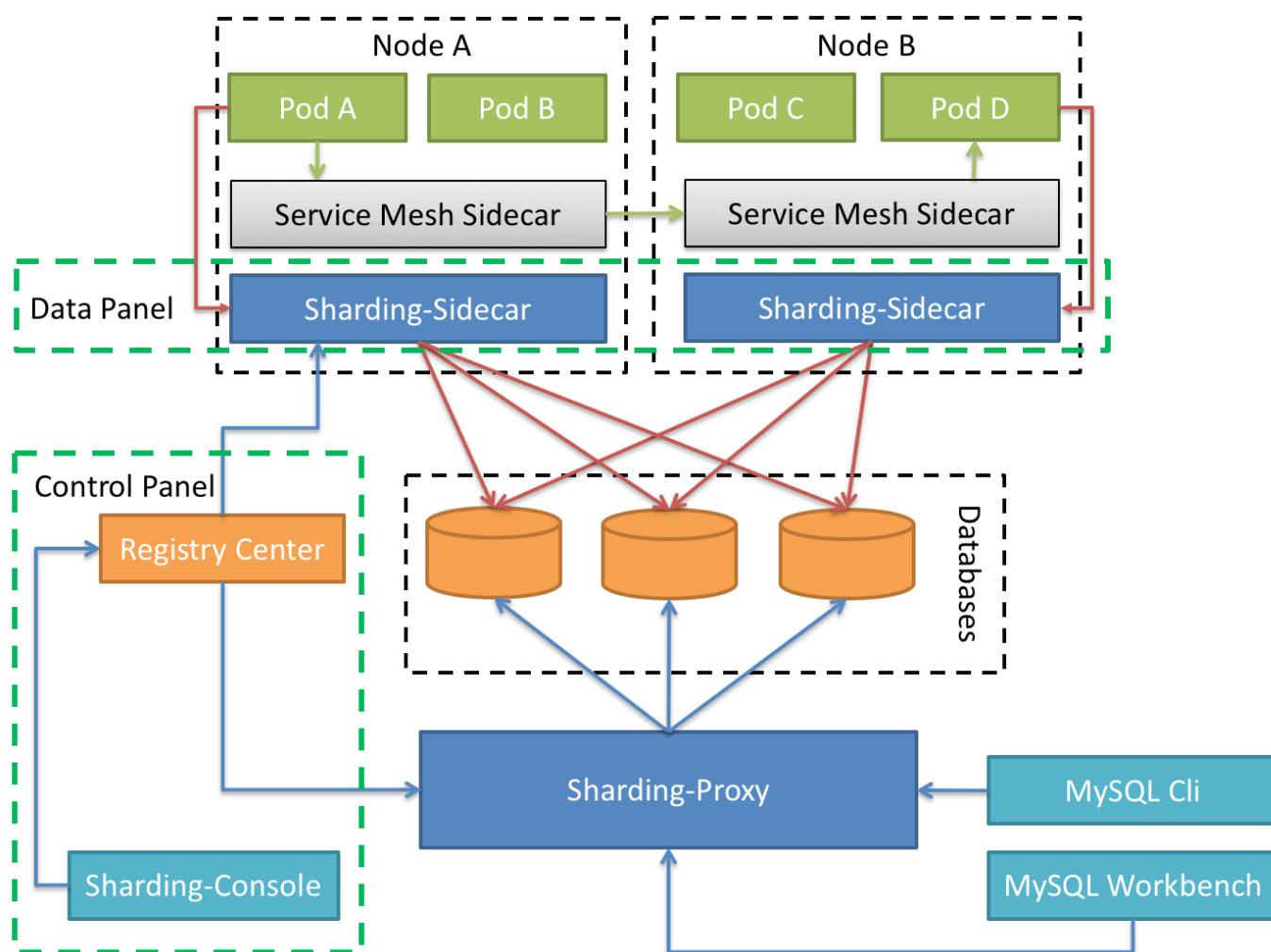
- 向应用程序完全透明，可直接当做 MySQL/PostgreSQL 使用；
- 适用于任何兼容 MySQL/PostgreSQL 协议的客户端。



4.ShardingSphere-Sidecar (TODO)

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

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



	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

二、快速开始

1.部署第一台MySQL服务器

使用docker部署第一台MySQL服务器，先来快速体验ShardingSphere带来的分表功能。在之后的分库分表中，再加入第二台MySQL服务器。

```
1  version: '3.1'
2  services:
3    mysql-0:
4      image: mysql
5      container_name: mysql-0
6      environment:
7        MYSQL_ROOT_PASSWORD: 123456
8      command:
9        --default-authentication-plugin=mysql_native_password
10       --character-set-server=utf8mb4
11       --collation-server=utf8mb4_general_ci
12       --explicit_defaults_for_timestamp=true
13       --lower_case_table_names=1
14      ports:
15        - 3306:3306
16      volumes:
17        - ./data:/var/lib/mysql
```

2. 创建数据库

创建名为db_device_0的数据库。

3. 创建物理表

逻辑上tb_device表示的是描述设备信息的表，为了体现分表的概念，把tb_device表分成了两张。于是tb_device就是逻辑表，而tb_device_0和tb_device_1就是该逻辑表的物理表。

- 创建tb_device_0表

```
1 CREATE TABLE `tb_device_0` (  
2     `device_id` bigint NOT NULL AUTO_INCREMENT,  
3     `device_type` int DEFAULT NULL,  
4     PRIMARY KEY (`device_id`)  
5 ) ENGINE=InnoDB AUTO_INCREMENT=9 DEFAULT CHARSET=utf8mb3;  
6
```

- 创建tb_device_1表

```
1 CREATE TABLE `tb_device_1` (  
2     `device_id` bigint NOT NULL AUTO_INCREMENT,  
3     `device_type` int DEFAULT NULL,  
4     PRIMARY KEY (`device_id`)  
5 ) ENGINE=InnoDB AUTO_INCREMENT=10 DEFAULT CHARSET=utf8mb3;
```

4. 创建ShardingSphere项目并引入依赖

```
1 <?xml version="1.0" encoding="UTF-8"?>  
2 <project xmlns="http://maven.apache.org/POM/4.0.0"  
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
3     xsi:schemaLocation="http://maven.apache.org/POM/4.0.0  
   http://maven.apache.org/xsd/maven-4.0.0.xsd">  
4     <modelVersion>4.0.0</modelVersion>  
5     <groupId>com.qf</groupId>  
6     <artifactId>my-sharding-jdbc-demo</artifactId>  
7     <version>0.0.1-SNAPSHOT</version>  
8     <name>my-sharding-jdbc-demo</name>  
9     <description>Demo project for Spring Boot</description>  
10  
11     <properties>
```

```
12         <java.version>1.8</java.version>
13         <project.build.sourceEncoding>UTF-
14 8</project.build.sourceEncoding>
15         <project.reporting.outputEncoding>UTF-
16 8</project.reporting.outputEncoding>
17         <spring-boot.version>2.3.7.RELEASE</spring-boot.version>
18     </properties>
19
20     <dependencies>
21
22         <dependency>
23             <groupId>org.projectlombok</groupId>
24             <artifactId>lombok</artifactId>
25             <version>1.18.16</version>
26         </dependency>
27
28         <dependency>
29             <groupId>com.alibaba</groupId>
30             <artifactId>druid</artifactId>
31             <version>1.1.22</version>
32         </dependency>
33
34         <dependency>
35             <groupId>mysql</groupId>
36             <artifactId>mysql-connector-java</artifactId>
37         </dependency>
38
39         <dependency>
40             <groupId>com.baomidou</groupId>
41             <artifactId>mybatis-plus-boot-starter</artifactId>
42             <version>3.0.5</version>
43         </dependency>
44
45         <dependency>
46             <groupId>org.apache.shardingsphere</groupId>
47             <artifactId>sharding-jdbc-spring-boot-starter</artifactId>
48             <version>4.1.1</version>
49         </dependency>
50
51         <dependency>
52             <groupId>org.springframework.boot</groupId>
53             <artifactId>spring-boot-starter-web</artifactId>
```

```
52         </dependency>
53
54         <dependency>
55             <groupId>org.springframework.boot</groupId>
56             <artifactId>spring-boot-starter-test</artifactId>
57             <scope>test</scope>
58             <exclusions>
59                 <exclusion>
60                     <groupId>org.junit.vintage</groupId>
61                     <artifactId>junit-vintage-engine</artifactId>
62                 </exclusion>
63             </exclusions>
64         </dependency>
65     </dependencies>
66
67     <dependencyManagement>
68         <dependencies>
69             <dependency>
70                 <groupId>org.springframework.boot</groupId>
71                 <artifactId>spring-boot-dependencies</artifactId>
72                 <version>${spring-boot.version}</version>
73                 <type>pom</type>
74                 <scope>import</scope>
75             </dependency>
76         </dependencies>
77     </dependencyManagement>
78
79     <build>
80         <plugins>
81             <plugin>
82                 <groupId>org.apache.maven.plugins</groupId>
83                 <artifactId>maven-compiler-plugin</artifactId>
84                 <version>3.8.1</version>
85                 <configuration>
86                     <source>1.8</source>
87                     <target>1.8</target>
88                     <encoding>UTF-8</encoding>
89                 </configuration>
90             </plugin>
91             <plugin>
92                 <groupId>org.springframework.boot</groupId>
93                 <artifactId>spring-boot-maven-plugin</artifactId>
```

```
94         <version>2.3.7.RELEASE</version>
95         <configuration>
96
97         <mainClass>com.qf.my.sharding.jdbc.demo.MyShardingJdbcDemoApplication
98         </mainClass>
99
100        </configuration>
101        <executions>
102            <execution>
103                <id>repackage</id>
104                <goals>
105                    <goal>repackage</goal>
106                </goals>
107            </execution>
108        </executions>
109    </plugin>
110 </plugins>
111 </build>
112
113 </project>
```

5.编写配置文件application.properties

```
1  # 配置真实数据源
2  spring.shardingsphere.datasource.names=ds1
3
4  # 配置第 1 个数据源
5  spring.shardingsphere.datasource.ds1.type=com.alibaba.druid.pool.DruidDataSource
6  spring.shardingsphere.datasource.ds1.driver-class-name=com.mysql.cj.jdbc.Driver
7  spring.shardingsphere.datasource.ds1.url=jdbc:mysql://172.16.253.84:3306/db_device_0?serverTimezone=Asia/Shanghai
8  spring.shardingsphere.datasource.ds1.username=root
9  spring.shardingsphere.datasource.ds1.password=123456
10
11 # 配置物理表
12 spring.shardingsphere.sharding.tables.tb_device.actual-data-nodes=ds1.tb_device_${0..1}
13
14 # 配置分表策略：根据device_id作为分片的依据（分片键）
```



```
15 spring.shardingsphere.sharding.tables.tb_device.table-  
strategy.inline.sharding-column=device_id  
16 spring.shardingsphere.sharding.tables.tb_device.table-  
strategy.inline.algorithm-expression=tb_device_${device_id%2}  
17 # 开启SQL显示  
18 spring.shardingsphere.props.sql.show = true  
19
```

6.创建逻辑表对应的实体

```
1 package com.qf.my.sharding.jdbc.demo.entity;  
2  
3 import lombok.Data;  
4  
5 /**  
6  * @author Thor  
7  * @公众号 Java架构栈  
8  */  
9 @Data  
10 public class TbDevice {  
11  
12     private Long deviceId;  
13  
14     private int deviceType;  
15  
16 }  
17
```

7.使用MyBatis-Plus做映射

```
1 package com.qf.my.sharding.jdbc.demo.mapper;
2
3 import com.baomidou.mybatisplus.core.mapper.BaseMapper;
4 import com.qf.my.sharding.jdbc.demo.entity.TbDevice;
5
6 /**
7  * @author Thor
8  * @公众号 Java架构栈
9  */
10 public interface DeviceMapper extends BaseMapper<TbDevice> {
11 }
12
```

8.配置Springboot启动类

关键是配置MapperScan

```
1 package com.qf.my.sharding.jdbc.demo;
2
3 import org.mybatis.spring.annotation.MapperScan;
4 import org.springframework.boot.SpringApplication;
5 import org.springframework.boot.autoconfigure.SpringBootApplication;
6
7 @SpringBootApplication
8 @MapperScan("com.qf.my.sharding.jdbc.demo.mapper")
9 public class MyShardingJdbcDemoApplication {
10
11     public static void main(String[] args) {
12         SpringApplication.run(MyShardingJdbcDemoApplication.class,
13             args);
14     }
15 }
16
```

9.编写单元测试

尝试插入10条device数据，因为分片键是device_id，且分片策略是 `tb_device_${device_id%2}`，因此会根据device_id来决定该条数据会插入到哪张物理表中。

```
1 package com.qf.my.sharding.jdbc.demo;
2
3 import com.qf.my.sharding.jdbc.demo.entity.TbDevice;
4 import com.qf.my.sharding.jdbc.demo.mapper.DeviceMapper;
5 import org.junit.jupiter.api.Test;
6 import org.springframework.beans.factory.annotation.Autowired;
7 import org.springframework.boot.test.context.SpringBootTest;
8
9 @SpringBootTest
10 class MyShardingJdbcDemoApplicationTests {
11
12     @Autowired
13     private DeviceMapper deviceMapper;
14
15     @Test
16     void initData(){
17         for (int i = 0; i < 10; i++) {
18             TbDevice device = new TbDevice();
19             device.setDeviceId((long) i);
20             device.setDeviceType(i);
21             deviceMapper.insert(device);
22         }
23     }
24 }
25
```

发现，根据分片策略，这10条数据中id是奇数的数据将会被插入到tb_device_1表中，id是偶数的数据将会被插入到tb_device_0表中。

三、尝试分库分表

1.准备第二台MySQL服务器

使用docker创建第二台MySQL服务器。

```
1 version: '3.1'
2 services:
3   mysql-1:
4     image: mysql
5     container_name: mysql-1
6     environment:
7       MYSQL_ROOT_PASSWORD: 123456
8     command:
9       --default-authentication-plugin=mysql_native_password
10      --character-set-server=utf8mb4
11      --collation-server=utf8mb4_general_ci
12      --explicit_defaults_for_timestamp=true
13      --lower_case_table_names=1
14     ports:
15       - 3306:3306
16     volumes:
17       - ./data:/var/lib/mysql
```

创建db_device_1数据库。并在数据库中创建两张物理表：

- 创建tb_device_0表

```
1 CREATE TABLE `tb_device_0` (
2   `device_id` bigint NOT NULL AUTO_INCREMENT,
3   `device_type` int DEFAULT NULL,
4   PRIMARY KEY (`device_id`)
5 ) ENGINE=InnoDB AUTO_INCREMENT=9 DEFAULT CHARSET=utf8mb3;
6
```

- 创建tb_device_1表

```
1 CREATE TABLE `tb_device_1` (
2   `device_id` bigint NOT NULL AUTO_INCREMENT,
3   `device_type` int DEFAULT NULL,
4   PRIMARY KEY (`device_id`)
5 ) ENGINE=InnoDB AUTO_INCREMENT=10 DEFAULT CHARSET=utf8mb3;
```

2.配置数据库源

提供两个数据源，将之前搭建的两台MySQL服务器作为数据源。

```
1  # 配置真实数据源
2  spring.shardingsphere.datasource.names=ds0,ds1
3
4  # 配置第 1 个数据源
5  spring.shardingsphere.datasource.ds0.type=com.alibaba.druid.pool.DruidD
   ataSource
6  spring.shardingsphere.datasource.ds0.driver-class-
   name=com.mysql.cj.jdbc.Driver
7  spring.shardingsphere.datasource.ds0.url=jdbc:mysql://172.16.253.84:330
   6/db_device_0?serverTimezone=Asia/Shanghai
8  spring.shardingsphere.datasource.ds0.username=root
9  spring.shardingsphere.datasource.ds0.password=123456
10
11 # 配置第 2 个数据源
12 spring.shardingsphere.datasource.ds1.type=com.alibaba.druid.pool.DruidD
   ataSource
13 spring.shardingsphere.datasource.ds1.driver-class-
   name=com.mysql.cj.jdbc.Driver
14 spring.shardingsphere.datasource.ds1.url=jdbc:mysql://172.16.253.85:330
   6/db_device_1?serverTimezone=Asia/Shanghai
15 spring.shardingsphere.datasource.ds1.username=root
16 spring.shardingsphere.datasource.ds1.password=123456
```

3.配置数据库表的分片策略

```

1 # 配置分库的分片策略, 根据device_id进行分片, 奇偶不同进入不同的数据库: ds1,ds2
2 spring.shardingsphere.sharding.default-database-
  strategy.inline.sharding-column=device_id
3 spring.shardingsphere.sharding.default-database-
  strategy.inline.algorithm-expression=ds$->{device_id % 2}
4
5 # 根据groovy脚本配置数据源+表名
6 spring.shardingsphere.sharding.tables.tb_device.actual-data-nodes=ds$->
  {0..1}.tb_device_$->{0..1}
7
8 # 配置分表策略
9 spring.shardingsphere.sharding.tables.tb_device.table-
  strategy.inline.sharding-column=device_id
10 spring.shardingsphere.sharding.tables.tb_device.table-
  strategy.inline.algorithm-expression=tb_device_$->{device_id%2}
11 # 开启显示sql语句
12 spring.shardingsphere.props.sql.show = true

```

相比之前的配置, 这次加入了两个数据库的分片策略, 根据device_id的奇偶特性决定存入哪个数据库中。同时, 使用groovy脚本确定了数据库和表之间的关系。

```

1 ds$->{0..1}.tb_device_$->{0..1}

```

等效于:

```

1 ds0.tb_device_0
2 ds0.tb_device_1
3 ds1.tb_device_0
4 ds1.tb_device_1

```

此时再运行测试用例, 发现device_id的奇数数据会存入 ds1.tb_device_1 表中, 偶数数据会存入 ds0.tb_device_0 表中。

4.在分库分表下做查询

```
1      @Test
2      void queryDeviceByID(){
3          QueryWrapper< TbDevice> queryWrapper = new QueryWrapper<>();
4          queryWrapper.eq("device_id", 1L);
5          List< TbDevice> deviceList =
deviceMapper.selectList(queryWrapper);
6          System.out.println(deviceList);
7      }
```

5.分库分表存在的问题

```
1      @Test
2      void queryDeviceByRange(){
3          QueryWrapper< TbDevice> queryWrapper = new QueryWrapper<>();
4          queryWrapper.between("device_id", 1, 10);
5          List< TbDevice> deviceList =
deviceMapper.selectList(queryWrapper);
6          System.out.println(deviceList);
7      }
```

使用device_id做范围查询时，发现报错了：inline的分片策略没有办法支持范围查询。

```
1  ### Cause: java.lang.IllegalStateException: Inline strategy cannot
support this type sharding:RangeRouteValue(columnName=device_id,
tableName=tb_device, valueRange=[1..10])
```

接下来需要掌握的是分片策略

四、分库分表核心知识点

1.核心概念

在了解分片策略之前，先来了解以下几个重点概念：逻辑表、真实表、数据节点、绑定表、广播表。

- 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：

```
1 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：

```
1 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);
2
3 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);
4
5 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);
6
7 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：

```
1 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);
2
3 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。

2.分片及分片策略

1) 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，ShardingSphere也支持根据多个字段进行分片。

2) 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。

3) 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SQL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，提供最大的灵活性。

- 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。

- 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。

3.分片策略的实现

1) Standard标准分片策略的精准分片

在Standard标准分片策略可以分别配置在分库和分表中。配置时需要指明分片键，精确分片或范围分片。

- 配置分库的精确分片

```
1 spring.shardingsphere.sharding.default-database-  
  strategy.standard.sharding-column=device_id  
2 spring.shardingsphere.sharding.default-database-  
  strategy.standard.precise-algorithm-class-  
  name=com.qf.my.sharding.jdbc.demo.sharding.algorithm.database.MyDatabase  
  StandardPreciseAlgorithm
```

需要提供一个实现精确分片算法的实现类，其中精确分片的逻辑可以与inline中的行表达式用意相同。

```
1 package com.qf.my.sharding.jdbc.demo.sharding.algorithm.database;  
2  
3 import  
  org.apache.shardingsphere.api.sharding.standard.PreciseShardingAlgorith  
  m;  
4 import  
  org.apache.shardingsphere.api.sharding.standard.PreciseShardingValue;  
5  
6 import java.util.Collection;  
7  
8 /**  
9  * @author Thor  
10  * @公众号 Java架构栈  
11  */  
12 public class MyDatabaseStandardPreciseAlgorithm implements  
  PreciseShardingAlgorithm<Long> {  
13     /**  
14      * 数据库的标准分片策略  
15      * @param collection 具体的物理库  
16      * @param preciseShardingValue 分片条件  
17      * @return  
18      */  
19     @Override
```

```

20     public String doSharding(Collection<String> collection,
PreciseShardingValue<Long> preciseShardingValue) {
21         //获得逻辑表名
22         String logicTableName =
preciseShardingValue.getLogicTableName();
23         //分片键, 列名
24         String columnName = preciseShardingValue.getColumnName();
25         //分片键的具体值
26         Long value = preciseShardingValue.getValue();
27         //根据分片策略: ds$->{device_id % 2} 做精确分片
28         String shardingKey = "ds" + (value % 2);
29         if (!collection.contains(shardingKey)){
30             throw new UnsupportedOperationException("数据
库:"+shardingKey+"不存在");
31         }
32         return shardingKey;
33     }
34 }
35

```

- 配置分表的精确分片

```

1 # standard
2 spring.shardingsphere.sharding.tables.tb_device.table-
strategy.standard.sharding-column=device_id
3 # 精确查找的算法实现类
4 spring.shardingsphere.sharding.tables.tb_device.table-
strategy.standard.precise-algorithm-class-
name=com.qf.my.sharding.jdbc.demo.sharding.algorithm.table.MyTableStanda
rdPreciseAlgorithm

```

同时, 需要提供分表的精确分片算法的实现类。

```

1 package com.qf.my.sharding.jdbc.demo.sharding.algorithm.table;
2
3 import
org.apache.shardingsphere.api.sharding.standard.PreciseShardingAlgorith
m;
4 import
org.apache.shardingsphere.api.sharding.standard.PreciseShardingValue;
5
6 import java.util.Collection;

```

```

7
8  /**
9   * @author Thor
10  * @公众号 Java架构栈
11  */
12  public class MyTableStandardPreciseAlgorithm implements
PreciseShardingAlgorithm<Long> {
13      @Override
14      public String doSharding(Collection<String> collection,
PreciseShardingValue<Long> preciseShardingValue) {
15          //获得逻辑表名
16          String logicTableName =
preciseShardingValue.getLogicTableName();
17          //分片键, 列名
18          String columnName = preciseShardingValue.getColumnName();
19          //分片键的具体值
20          Long value = preciseShardingValue.getValue();
21          //根据分片策略: tb_device_${device_id%2} 做精确分片
22          String shardingKey = logicTableName + "_" + (value % 2);
23          if (!collection.contains(shardingKey)) {
24              throw new UnsupportedOperationException("表:" + shardingKey
+ "不存在");
25          }
26          return shardingKey;
27      }
28  }
29

```

尝试再之前的精确查找测试用例，发现与之前的效果相同，根据id定位到某个库的某张表中。

```

1      @Test
2      void queryDeviceByID(){
3          QueryWrapper<TbDevice> queryWrapper = new QueryWrapper<>();
4          queryWrapper.eq("device_id", 1L);
5          List<TbDevice> deviceList =
deviceMapper.selectList(queryWrapper);
6          System.out.println(deviceList);
7      }

```

2) Standard 标准分片策略的范围分片

- 配置分库的范围分片

```
1 spring.shardingsphere.sharding.default-database-strategy.standard.range-
  algorithm-class-
  name=com.qf.my.sharding.jdbc.demo.sharding.algorithm.database.MyDatabase
  StandardRangeAlgorithm
```

提供范围查询算法的实现类。

```
1 package com.qf.my.sharding.jdbc.demo.sharding.algorithm.database;
2
3 import
  org.apache.shardingsphere.api.sharding.standard.RangeShardingAlgorithm;
4 import
  org.apache.shardingsphere.api.sharding.standard.RangeShardingValue;
5
6 import java.util.Arrays;
7 import java.util.Collection;
8
9 /**
10  * @author Thor
11  * @公众号 Java架构栈
12  */
13 public class MyDatabaseStandardRangeAlgorithm implements
  RangeShardingAlgorithm<Long> {
14     /**
15      * select * from tb_device where id between (1,10);
16      * 由于范围查询，需要在两个库的两张表中查。
17      * @param collection
18      * @param rangeShardingValue 提供了这次查询的条件 1,10
19      * @return 返回要进行范围查询的库名
20      */
21     @Override
22     public Collection<String> doSharding(Collection<String> collection,
  RangeShardingValue<Long> rangeShardingValue) {
23         return Arrays.asList("ds0", "ds1");
24     }
25 }
26
```

- 配置分表的范围分片

```
1 spring.shardingsphere.sharding.tables.tb_device.table-  
strategy.standard.range-algorithm-class-  
name=com.qf.my.sharding.jdbc.demo.sharding.algorithm.table.MyTableStandar-  
dRangeAlgorithm
```

提供范围查询算法的实现类：

```
1 package com.qf.my.sharding.jdbc.demo.sharding.algorithm.table;  
2  
3 import  
org.apache.shardingsphere.api.sharding.standard.RangeShardingAlgorithm;  
4 import  
org.apache.shardingsphere.api.sharding.standard.RangeShardingValue;  
5  
6 import java.util.Arrays;  
7 import java.util.Collection;  
8  
9 /**  
10  * @author Thor  
11  * @公众号 Java架构栈  
12  */  
13 public class MyTableStandardRangeAlgorithm implements  
RangeShardingAlgorithm<Long> {  
14  
15     @Override  
16     public Collection<String> doSharding(Collection<String> collection,  
RangeShardingValue<Long> rangeShardingValue) {  
17         //返回两种物理表  
18         String logicTableName = rangeShardingValue.getLogicTableName();  
19         return Arrays.asList(logicTableName+"_0",logicTableName+"_1");  
20     }  
21 }
```

此时，再运行范围查询的测试用例，发现成功了。

```
1      @Test
2      void queryDeviceByRange(){
3          QueryWrapper<TbDevice> queryWrapper = new QueryWrapper<>();
4          queryWrapper.between("device_id",1,10);
5          List<TbDevice> deviceList =
deviceMapper.selectList(queryWrapper);
6          System.out.println(deviceList);
7      }
```

3) Complex分片策略

- 问题的出现

```
1      @Test
2      void queryDeviceByRangeAndDeviceType(){
3
4          QueryWrapper<TbDevice> queryWrapper = new QueryWrapper<>();
5          queryWrapper.between("device_id",1,10);
6          queryWrapper.eq("device_type", 5);
7          List<TbDevice> deviceList =
deviceMapper.selectList(queryWrapper);
8          System.out.println(deviceList);
9
10     }
```

在对device_id进行范围查询的同时，需要根据device_type做精确查找，发现此时也需要查两个库的三张表，但是奇数的device_type只会在奇数库的奇数表中，此时冗余了多次不必要的查询。


```

1 INFO 5879 --- [          main] ShardingSphere-SQL
  : Actual SQL: ds0 ::: SELECT device_id,device_type FROM tb_device_0
2 WHERE device_id BETWEEN ? AND ? AND device_type = ? ::: [1, 10, 5]
3 INFO 5879 --- [          main] ShardingSphere-SQL
  : Actual SQL: ds0 ::: SELECT device_id,device_type FROM tb_device_1
4 WHERE device_id BETWEEN ? AND ? AND device_type = ? ::: [1, 10, 5]
5 INFO 5879 --- [          main] ShardingSphere-SQL
  : Actual SQL: ds1 ::: SELECT device_id,device_type FROM tb_device_0
6 WHERE device_id BETWEEN ? AND ? AND device_type = ? ::: [1, 10, 5]
7 INFO 5879 --- [          main] ShardingSphere-SQL
  : Actual SQL: ds1 ::: SELECT device_id,device_type FROM tb_device_1
8 WHERE device_id BETWEEN ? AND ? AND device_type = ? ::: [1, 10, 5]

```

为了解决冗余的多次查找，可以使用complex的分片策略。

- complex的分片策略

支持多个字段的分片策略。

```

1 # 分库的分片策略
2 spring.shardingsphere.sharding.default-database-
  strategy.complex.sharding-columns=device_id,device_type
3 spring.shardingsphere.sharding.default-database-
  strategy.complex.algorithm-class-
  name=com.qf.my.sharding.jdbc.demo.sharding.algorithm.database.MyDatabase
  ComplexAlgorithm
4 # 分表的分片策略
5 spring.shardingsphere.sharding.tables.tb_device.table-
  strategy.complex.sharding-columns=device_id,device_type
6 spring.shardingsphere.sharding.tables.tb_device.table-
  strategy.complex.algorithm-class-
  name=com.qf.my.sharding.jdbc.demo.sharding.algorithm.table.MyTableComple
  xAlgorithm

```

配置分库的算法实现类

```

1 package com.qf.my.sharding.jdbc.demo.sharding.algorithm.database;
2
3 import com.google.common.collect.Range;
4 import
  org.apache.shardingsphere.api.sharding.complex.ComplexKeysShardingAlgor
  ithm;

```

```

5  import
   org.apache.shardingsphere.api.sharding.complex.ComplexKeysShardingValue
   ;
6
7  import java.util.ArrayList;
8  import java.util.Collection;
9  import java.util.Map;
10
11 /**
12  * @author Thor
13  * @公众号 Java架构栈
14  */
15 public class MyDatabaseComplexAlgorithm implements
   ComplexKeysShardingAlgorithm<Integer> {
16
17     @Override
18     public Collection<String> doSharding(Collection<String> collection,
   ComplexKeysShardingValue<Integer> complexKeysShardingValue) {
19         //获得这一次查询的device_type的所有值。
20         Collection<Integer> deviceTypes =
   complexKeysShardingValue.getColumnNamesAndShardingValuesMap().get("device_type");
21         //存放指定的库
22         Collection<String> databases = new ArrayList<>();
23         for (Integer deviceType : deviceTypes) {
24             //根据deviceType的奇偶选择哪个数据库
25             String database = "ds" + (deviceType % 2);
26             databases.add(database);
27         }
28         return databases;
29     }
30 }
31

```

配置分表的算法实现类

```

1  package com.qf.my.sharding.jdbc.demo.sharding.algorithm.table;
2
3  import
   org.apache.shardingsphere.api.sharding.complex.ComplexKeysShardingAlgorithm;

```

```
4  import
   org.apache.shardingsphere.api.sharding.complex.ComplexKeysShardingValue
   ;
5
6  import java.util.ArrayList;
7  import java.util.Collection;
8
9  /**
10   * @author Thor
11   * @公众号 Java架构栈
12   */
13  public class MyTableComplexAlgorithm implements
   ComplexKeysShardingAlgorithm<Integer> {
14      @Override
15      public Collection<String> doSharding(Collection<String> collection,
   ComplexKeysShardingValue<Integer> complexKeysShardingValue) {
16          //获得这一次查询的device_type的所有值。
17          Collection<Integer> deviceTypes =
   complexKeysShardingValue.getColumnNamesAndShardingValuesMap().get("device_type");
18          //存放指定的表
19          Collection<String> tables = new ArrayList<>();
20          for (Integer deviceType : deviceTypes) {
21              //根据deviceType的奇偶选择哪个数据库
22              String tableName =
   complexKeysShardingValue.getLogicTableName() + "_" + (deviceType % 2);
23              tables.add(tableName);
24          }
25          return tables;
26      }
27  }
28
```

4) Hint强制路由策略

hint可以不根据sql语句特性，强制路由到某个库的某个表中。

```

1 #hint
2 spring.shardingsphere.sharding.tables.tb_device.table-
  strategy.hint.algorithm-class-
  name=com.qf.my.sharding.jdbc.demo.sharding.algorithm.table.MyTableHintAl
  gorithm

```

配置hint算法的实现类

```

1 package com.qf.my.sharding.jdbc.demo.sharding.algorithm.table;
2
3 import
  org.apache.shardingsphere.api.sharding.hint.HintShardingAlgorithm;
4 import org.apache.shardingsphere.api.sharding.hint.HintShardingValue;
5
6 import java.util.Arrays;
7 import java.util.Collection;
8
9 /**
10  * @author Thor
11  * @公众号 Java架构栈
12  */
13 public class MyTableHintAlgorithm implements
  HintShardingAlgorithm<Long> {
14     @Override
15     public Collection<String> doSharding(Collection<String> collection,
  HintShardingValue<Long> hintShardingValue) {
16         //根据指定参数强制路由
17         String tableName = hintShardingValue.getLogicTableName() + "_"
  + hintShardingValue.getValues().toArray()[0];
18         if (!collection.contains(tableName)) {
19             throw new UnsupportedOperationException("表: " + tableName
  + ", 不存在");
20         }
21         return Arrays.asList(tableName);
22     }
23 }
24

```

编写单元测试。这次查询都会查询两个数据库的tb_device_0这张表。

```
1      @Test
2      void queryByHint(){
3          HintManager hintManager = HintManager.getInstance();
4          //指定强制路由的表
5          hintManager.addTableShardingValue("tb_device",0);
6          List< TbDevice> deviceList = deviceMapper.selectList(null);
7          System.out.println(deviceList);
8          hintManager.close();
9      }
```

4.绑定表

先来模拟笛卡尔积的出现。

- 创建 `tb_device_info` 表:

```
1  CREATE TABLE `tb_device_info_0` (
2      `id` bigint NOT NULL,
3      `device_id` bigint DEFAULT NULL,
4      `device_intro` varchar(255) COLLATE utf8mb4_general_ci DEFAULT NULL,
5      PRIMARY KEY (`id`)
6  ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci;
```

- 配置 `tb_device` 和 `tb_device_info` 表的分片策略。

```

1  # tb_device表的分片策略
2  spring.shardingsphere.sharding.tables.tb_device.actual-data-nodes=ds$->
   {0..1}.tb_device_$->{0..1}
3  spring.shardingsphere.sharding.tables.tb_device.table-
   strategy.inline.sharding-column=device_id
4  spring.shardingsphere.sharding.tables.tb_device.table-
   strategy.inline.algorithm-expression=tb_device_$->{device_id%2}
5
6
7  # # tb_device_info表的分片策略
8  spring.shardingsphere.sharding.tables.tb_device_info.actual-data-
   nodes=ds$->{0..1}.tb_device_info_$->{0..1}
9  spring.shardingsphere.sharding.tables.tb_device_info.table-
   strategy.inline.sharding-column=device_id
10 spring.shardingsphere.sharding.tables.tb_device_info.table-
   strategy.inline.algorithm-expression=tb_device_info_$->{device_id%2}

```

两张表的分片键都是device_id。

- 编写测试用例，插入数据

```

1  @Test
2  void testInsertType(){
3
4      for (int i = 0; i < 10; i++) {
5          TbDevice device = new TbDevice();
6          device.setDeviceId((long) i);
7          device.setDeviceType(i);
8          deviceMapper.insert(device);
9
10         TbDeviceInfo deviceInfo = new TbDeviceInfo();
11         deviceInfo.setDeviceId((long) i);
12         deviceInfo.setDeviceIntro(" "+i);
13         deviceInfoMapper.insert(deviceInfo);
14
15     }
16
17 }

```

- join查询时出现笛卡尔积

```

1  package com.qf.my.sharding.jdbc.demo.mapper;

```

```

2
3 import com.baomidou.mybatisplus.core.mapper.BaseMapper;
4 import com.qf.my.sharding.jdbc.demo.entity.TbDeviceInfo;
5 import org.apache.ibatis.annotations.Select;
6
7 import java.util.List;
8
9 /**
10  * @author Thor
11  * @公众号 Java架构栈
12  */
13 public interface DeviceInfoMapper extends BaseMapper<TbDeviceInfo> {
14
15     @Select("select a.id,a.device_id,a.device_intro,b.device_type from
16     tb_device_info a left join tb_device b on a.device_id = b.device_id ")
17     public List<TbDeviceInfo> queryDeviceInfo();
18 }

```

出现笛卡尔积的查询

```

1 @Test
2 void testQueryDeviceInfo(){
3     List<TbDeviceInfo> deviceInfos =
4     deviceInfoMapper.queryDeviceInfo();
5     deviceInfos.forEach( deviceInfo ->
6     System.out.println(deviceInfo));
7 }

```

结果：

```

1 2022-03-28 21:09:17.413 INFO 14006 --- [          main]
   ShardingSphere-SQL           : Actual SQL: ds0 ::: select
   a.id,a.device_id,a.device_intro,b.device_type from tb_device_info_1 a
   left join tb_device_0 b on a.device_id = b.device_id
2 2022-03-28 21:09:17.413 INFO 14006 --- [          main]
   ShardingSphere-SQL           : Actual SQL: ds0 ::: select
   a.id,a.device_id,a.device_intro,b.device_type from tb_device_info_1 a
   left join tb_device_1 b on a.device_id = b.device_id

```

```

3 2022-03-28 21:09:17.414 INFO 14006 --- [          main]
   ShardingSphere-SQL                : Actual SQL: ds0 ::: select
   a.id,a.device_id,a.device_intro,b.device_type from tb_device_info_0 a
   left join tb_device_0 b on a.device_id = b.device_id
4 2022-03-28 21:09:17.414 INFO 14006 --- [          main]
   ShardingSphere-SQL                : Actual SQL: ds0 ::: select
   a.id,a.device_id,a.device_intro,b.device_type from tb_device_info_0 a
   left join tb_device_1 b on a.device_id = b.device_id
5 2022-03-28 21:09:17.414 INFO 14006 --- [          main]
   ShardingSphere-SQL                : Actual SQL: ds1 ::: select
   a.id,a.device_id,a.device_intro,b.device_type from tb_device_info_1 a
   left join tb_device_0 b on a.device_id = b.device_id
6 2022-03-28 21:09:17.414 INFO 14006 --- [          main]
   ShardingSphere-SQL                : Actual SQL: ds1 ::: select
   a.id,a.device_id,a.device_intro,b.device_type from tb_device_info_1 a
   left join tb_device_1 b on a.device_id = b.device_id
7 2022-03-28 21:09:17.414 INFO 14006 --- [          main]
   ShardingSphere-SQL                : Actual SQL: ds1 ::: select
   a.id,a.device_id,a.device_intro,b.device_type from tb_device_info_0 a
   left join tb_device_0 b on a.device_id = b.device_id
8 2022-03-28 21:09:17.414 INFO 14006 --- [          main]
   ShardingSphere-SQL                : Actual SQL: ds1 ::: select
   a.id,a.device_id,a.device_intro,b.device_type from tb_device_info_0 a
   left join tb_device_1 b on a.device_id = b.device_id
9 TbDeviceInfo(id=1508423776753684482, deviceId=0, deviceIntro=0)
10 TbDeviceInfo(id=1508423776938233858, deviceId=2, deviceIntro=2)
11 TbDeviceInfo(id=1508423777097617410, deviceId=4, deviceIntro=4)
12 TbDeviceInfo(id=1508423777210863618, deviceId=6, deviceIntro=6)
13 TbDeviceInfo(id=1508423777328304130, deviceId=8, deviceIntro=8)
14 TbDeviceInfo(id=1508423776753684482, deviceId=0, deviceIntro=0)
15 TbDeviceInfo(id=1508423776938233858, deviceId=2, deviceIntro=2)
16 TbDeviceInfo(id=1508423777097617410, deviceId=4, deviceIntro=4)
17 TbDeviceInfo(id=1508423777210863618, deviceId=6, deviceIntro=6)
18 TbDeviceInfo(id=1508423777328304130, deviceId=8, deviceIntro=8)
19 TbDeviceInfo(id=1508423776858542081, deviceId=1, deviceIntro=1)
20 TbDeviceInfo(id=1508423777030508546, deviceId=3, deviceIntro=3)
21 TbDeviceInfo(id=1508423777152143362, deviceId=5, deviceIntro=5)
22 TbDeviceInfo(id=1508423777273778177, deviceId=7, deviceIntro=7)
23 TbDeviceInfo(id=1508423777378635778, deviceId=9, deviceIntro=9)
24 TbDeviceInfo(id=1508423776858542081, deviceId=1, deviceIntro=1)
25 TbDeviceInfo(id=1508423777030508546, deviceId=3, deviceIntro=3)
26 TbDeviceInfo(id=1508423777152143362, deviceId=5, deviceIntro=5)

```



```

27 TbDeviceInfo(id=1508423777273778177, deviceId=7, deviceIntro=7)
28 TbDeviceInfo(id=1508423777378635778, deviceId=9, deviceIntro=9)

```

● 配置绑定表

配置绑定表：

```

1 spring.shardingsphere.sharding.binding-
  tables[0]=tb_device,tb_device_info

```

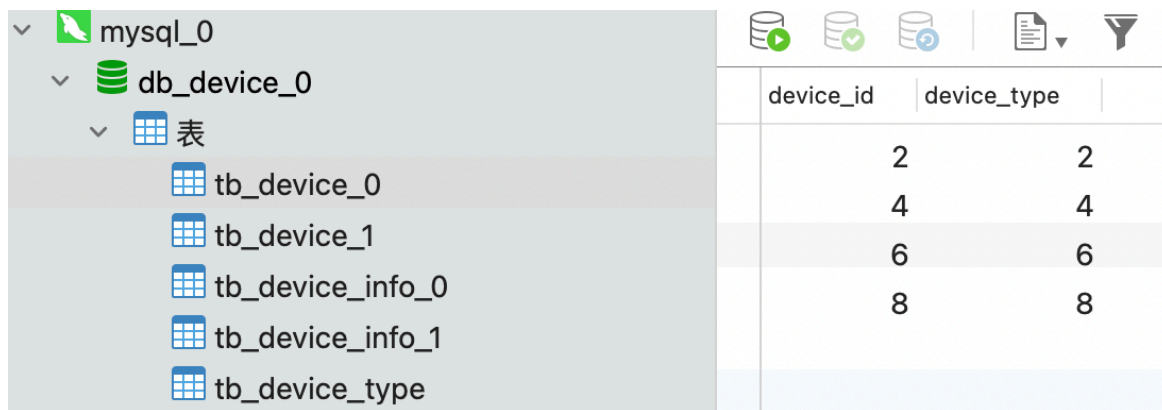
再次查询，不再出现笛卡尔积：

```

1 2022-03-28 21:10:48.549 INFO 14661 --- [          main]
  ShardingSphere-SQL           : Actual SQL: ds0 ::: select
  a.id,a.device_id,a.device_intro,b.device_type from tb_device_info_0 a
  left join tb_device_0 b on a.device_id = b.device_id
2 2022-03-28 21:10:48.550 INFO 14661 --- [          main]
  ShardingSphere-SQL           : Actual SQL: ds0 ::: select
  a.id,a.device_id,a.device_intro,b.device_type from tb_device_info_1 a
  left join tb_device_1 b on a.device_id = b.device_id
3 2022-03-28 21:10:48.550 INFO 14661 --- [          main]
  ShardingSphere-SQL           : Actual SQL: ds1 ::: select
  a.id,a.device_id,a.device_intro,b.device_type from tb_device_info_0 a
  left join tb_device_0 b on a.device_id = b.device_id
4 2022-03-28 21:10:48.550 INFO 14661 --- [          main]
  ShardingSphere-SQL           : Actual SQL: ds1 ::: select
  a.id,a.device_id,a.device_intro,b.device_type from tb_device_info_1 a
  left join tb_device_1 b on a.device_id = b.device_id
5 TbDeviceInfo(id=1508423776753684482, deviceId=0, deviceIntro=0)
6 TbDeviceInfo(id=1508423776938233858, deviceId=2, deviceIntro=2)
7 TbDeviceInfo(id=1508423777097617410, deviceId=4, deviceIntro=4)
8 TbDeviceInfo(id=1508423777210863618, deviceId=6, deviceIntro=6)
9 TbDeviceInfo(id=1508423777328304130, deviceId=8, deviceIntro=8)
10 TbDeviceInfo(id=1508423776858542081, deviceId=1, deviceIntro=1)
11 TbDeviceInfo(id=1508423777030508546, deviceId=3, deviceIntro=3)
12 TbDeviceInfo(id=1508423777152143362, deviceId=5, deviceIntro=5)
13 TbDeviceInfo(id=1508423777273778177, deviceId=7, deviceIntro=7)
14 TbDeviceInfo(id=1508423777378635778, deviceId=9, deviceIntro=9)

```

5.广播表



device_id	device_type
2	2
4	4
6	6
8	8

现在有这么一个场景，device_type列对应的tb_device_type表中的数据，不应该被分表，两个库中都应该有全量的该表的数据。

- 在两个数据库中创建 tb_device_type 表

```
1 CREATE TABLE `tb_device_type` (
2   `type_id` int NOT NULL AUTO_INCREMENT,
3   `type_name` varchar(255) COLLATE utf8mb4_general_ci DEFAULT NULL,
4   PRIMARY KEY (`type_id`)
5 ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci;
```

- 配置广播表

```
1 #广播表配置
2 spring.shardingsphere.sharding.broadcast-tables=tb_device_type
3 spring.shardingsphere.sharding.tables.t_dict.key-
4 generator.column=type_id
5 spring.shardingsphere.sharding.tables.t_dict.key-
6 generator.type=SNOWFLAKE
```

- 编写测试用例

```
1 @Test
2 void testAddDeviceType(){
3
```

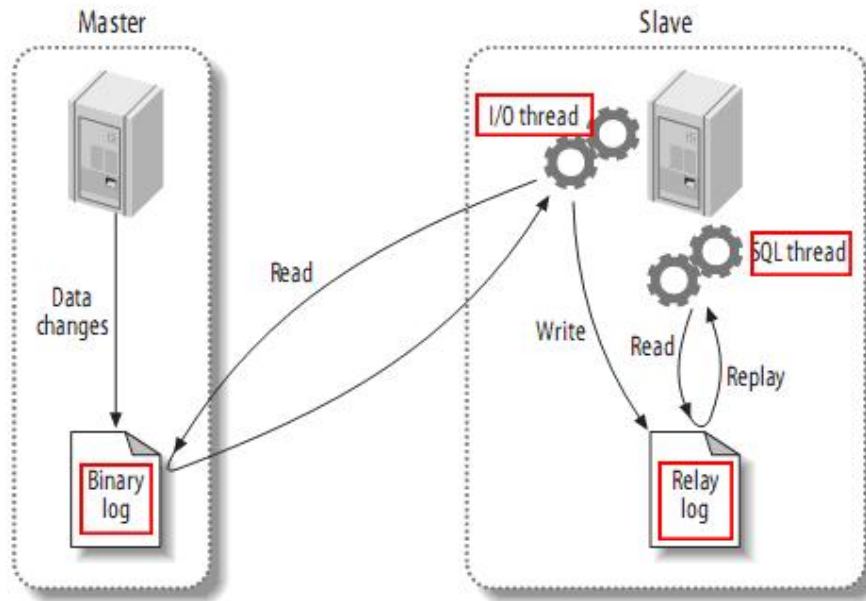
```
4      TbDeviceType deviceType1 = new TbDeviceType();
5      deviceType1.setTypeId(1);
6      deviceType1.setTypeName("人脸考勤");
7      deviceTypeMapper.insert(deviceType1);
8
9      TbDeviceType deviceType2 = new TbDeviceType();
10     deviceType2.setTypeId(2);
11     deviceType2.setTypeName("人脸通道");
12     deviceTypeMapper.insert(deviceType2);
13
14 }
```

五、实现读写分离

1.搭建主从同步数据库

- 主从同步原理

Master将数据写入到binlog日志中。Slave读取主节点的Binlog数据到本地的relaylog日志文件中。此时，Slave持续不断的与Master同步，且数据存在于relaylog中，而并非落在数据库。于是Slave开启一条线程，专门讲relaylog中的数据写入到数据库中。



- 准备Master主库

```
1 version: '3.1'
2 services:
3   mysql:
4     restart: always
5     image: mysql:5.7.25
6     container_name: mysql
7     ports:
8       - 3306:3306
9     environment:
10      TZ: Asia/Shanghai
11      MYSQL_ROOT_PASSWORD: 123456
12     command:
13       --character-set-server=utf8mb4
14       --collation-server=utf8mb4_general_ci
15       --explicit_defaults_for_timestamp=true
16       --lower_case_table_names=1
17       --max_allowed_packet=128M
18       --server-id=47
19       --log_bin=master-bin
20       --log_bin-index=master-bin.index
21       --skip-name-resolve
```

```

22      --sql-
mode="STRICT_TRANS_TABLES,NO_AUTO_CREATE_USER,NO_ENGINE_SUBSTITUTION,NO
_ZERO_DATE,NO_ZERO_IN_DATE,ERROR_FOR_DIVISION_BY_ZERO"
23      volumes:
24          - mysql-data:/var/lib/mysql
25
26      volumes:
27          mysql-data:

```

注意其中的配置：

```

1  服务id: server-id=47
2  开启binlog: log_bin=master-bin
3  binlog索引: log_bin_index=master-bin.index

```

通过 `show master status` 命令查看并记录文件名和偏移量。

```

mysql> show master status;
+-----+-----+-----+-----+-----+
| File           | Position | Binlog_Do_DB | Binlog_Ignore_DB | Executed_Gtid_Set |
+-----+-----+-----+-----+-----+
| master-bin.000004 | 156      |              |                  |                   |
+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)

```

- 准备Slave从库：

```

1  version: '3.1'
2  services:
3      mysql:
4          restart: always
5          image: mysql:5.7.25
6          container_name: mysql
7          ports:
8              - 3306:3306
9          environment:
10             TZ: Asia/Shanghai
11             MYSQL_ROOT_PASSWORD: 123456
12         command:
13             --character-set-server=utf8mb4
14             --collation-server=utf8mb4_general_ci
15             --explicit_defaults_for_timestamp=true
16             --lower_case_table_names=1
17             --max_allowed_packet=128M
18             --server-id=48

```

```

19     --relay-log-index=slave-relay-bin.index
20     --relay-log=slave-relay-bin
21     --log-bin=mysql-bin
22     --log-slave-updates=1
23     --sql-
mode="STRICT_TRANS_TABLES,NO_AUTO_CREATE_USER,NO_ENGINE_SUBSTITUTION,NO
_ZERO_DATE,NO_ZERO_IN_DATE,ERROR_FOR_DIVISION_BY_ZERO"
24     volumes:
25         - mysql-data:/var/lib/mysql
26
27 volumes:
28     mysql-data:

```

注意其中的关键配置：

```

1  服务id: server-id=48
2  开启中继日志: relay-log-index=slave-relay-bin.index
3  开启中继日志: relay-log=slave-relay-bin

```

启动从库后进入到从库，并依次执行如下命令：

```

1  #登录从服务
2  mysql -u root -p;
3  #设置同步主节点:
4  CHANGE MASTER TO
5  MASTER_HOST='172.16.253.31',
6  MASTER_PORT=3306,
7  MASTER_USER='root',
8  MASTER_PASSWORD='123456',
9  MASTER_LOG_FILE='master-bin.000003',
10 MASTER_LOG_POS=154;
11 #开启slave
12 start slave;

```

至此，主从同步集群搭建完成。

在主库中创建 `db_device` 数据库，并在库中创建表：

```
1 CREATE TABLE `tb_user` (  
2   `id` bigint(20) NOT NULL AUTO_INCREMENT,  
3   `name` varchar(255) DEFAULT NULL,  
4   PRIMARY KEY (`id`)  
5 ) ENGINE=InnoDB AUTO_INCREMENT=4 DEFAULT CHARSET=utf8mb4;
```

2.使用sharding-jdbc实现读写分离

- 编写配置文件

```
1 # 配置真实数据源  
2 spring.shardingsphere.datasource.names=m0,s0  
3  
4 # 配置主数据源  
5 spring.shardingsphere.datasource.m0.type=com.alibaba.druid.pool.DruidData  
6   taSource  
7 spring.shardingsphere.datasource.m0.driver-class-  
8   name=com.mysql.cj.jdbc.Driver  
9 spring.shardingsphere.datasource.m0.url=jdbc:mysql://172.16.253.73:3306  
10  /db_device?serverTimezone=Asia/Shanghai  
11 spring.shardingsphere.datasource.m0.username=root  
12 spring.shardingsphere.datasource.m0.password=123456  
13  
14 # 配置从数据源  
15 spring.shardingsphere.datasource.s0.type=com.alibaba.druid.pool.DruidData  
16   taSource  
17 spring.shardingsphere.datasource.s0.driver-class-  
18   name=com.mysql.cj.jdbc.Driver  
19 spring.shardingsphere.datasource.s0.url=jdbc:mysql://172.16.253.74:3306  
20  /db_device?serverTimezone=Asia/Shanghai  
21 spring.shardingsphere.datasource.s0.username=root  
22 spring.shardingsphere.datasource.s0.password=123456  
23  
24 # 分配读写规则  
25 spring.shardingsphere.sharding.master-slave-rules.ds0.master-data-  
26   source-name=m0  
27 spring.shardingsphere.sharding.master-slave-rules.ds0.slave-data-  
28   source-names[0]=s0  
29  
30 # 确定实际表
```

```

23 spring.shardingsphere.sharding.tables.tb_user.actual-data-
    nodes=ds0.tb_user
24 # 确定主键生成策略
25 spring.shardingsphere.sharding.tables.t_dict.key-generator.column=id
26 spring.shardingsphere.sharding.tables.t_dict.key-
    generator.type=SNOWFLAKE
27
28 # 开启显示sql语句
29 spring.shardingsphere.props.sql.show = true

```

- 测试写数据

```

1      @Test
2      void testInsertUser(){
3          for (int i = 0; i < 10; i++) {
4              TbUser user = new TbUser();
5              user.setName(""+i);
6              userMapper.insert(user);
7          }
8      }

```

此时，所有的数据只会往主库中写，然后再同步到从库。

- 测试读数据

```

1      @Test
2      void testQueryUser(){
3          List<TbUser> tbUsers = userMapper.selectList(null);
4          tbUsers.forEach( tbUser -> System.out.println(tbUser));
5      }

```

此时，所有的数据都读自于从库。

六、实现原理-连接模式

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。

1.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。

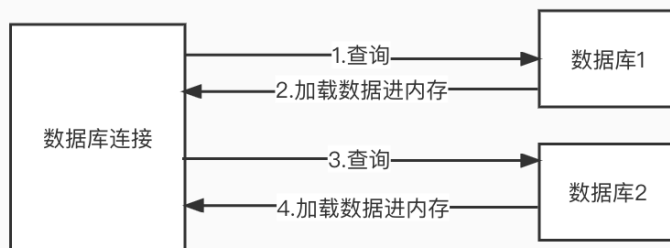
流式归并

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。



内存归并

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。



1) 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。

2) 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。

2. 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。

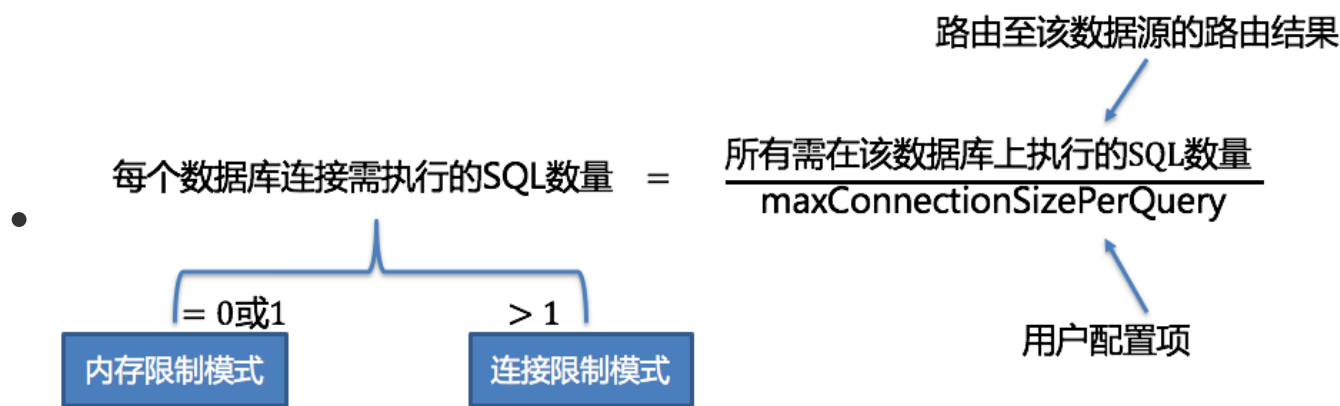
执行引擎分为准备和执行两个阶段。

3. 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：

- 将 SQL 的路由结果按照数据源的名称进行分组。
- 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。



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。

七、实现原理-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。

1. 遍历归并

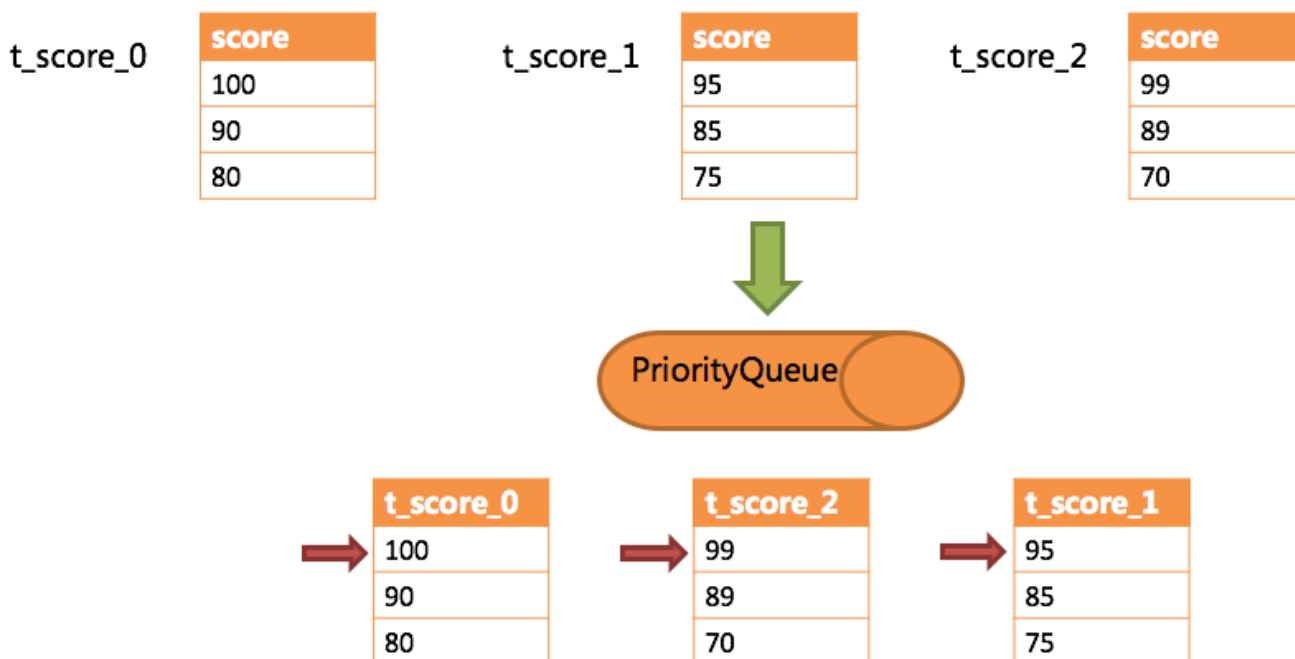
它是最为简单的归并方式。只需将多个数据结果集合并为一个单向链表即可。在遍历完成链表中当前数据结果集之后，将链表元素后移一位，继续遍历下一个数据结果集即可。

2. 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。

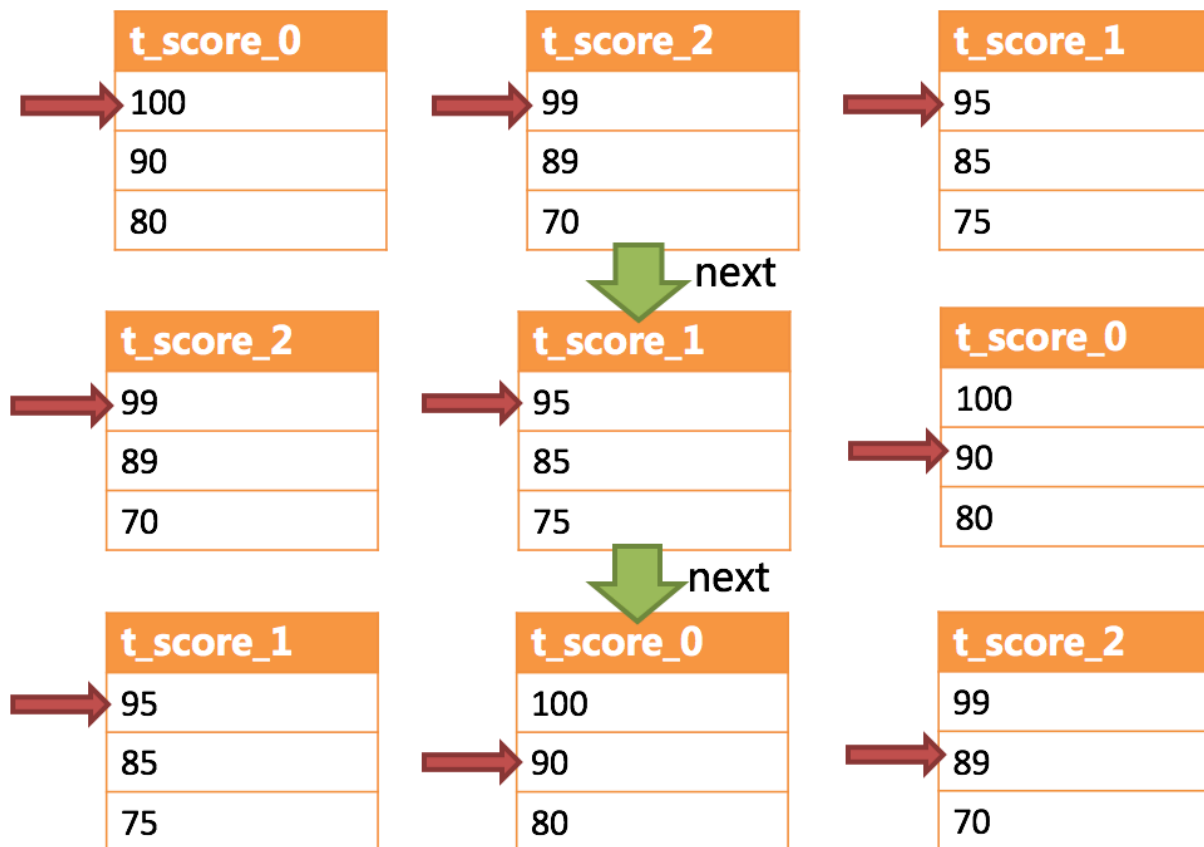
ShardingSphere 在对排序的查询进行归并时，将每个结果集的当前数据值进行比较（通过实现 Java 的 `Comparable` 接口完成），并将其放入优先级队列。每次获取下一条数据时，只需将队列顶端结果集的游标下移，并根据新游标重新进入优先级排序队列找到自己的位置即可。

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。

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，而多数数据结果集整体无序的情况下，ShardingSphere 无需将所有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。

3. 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数据 correctness。

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：

```
1 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。

t_score_java

name	score
Tom	100
Jerry	90
Mary	80

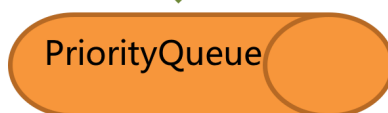
t_score_go

name	score
Jerry	95
Tom	85
John	75

t_score_python

name	score
John	99
Mary	89
Tom	70

```
SELECT name, SUM(score) FROM t_score GROUP BY name ORDER BY name;
```



t_score_java

name	score
Jerry	90
Mary	80
Tom	100



t_score_go

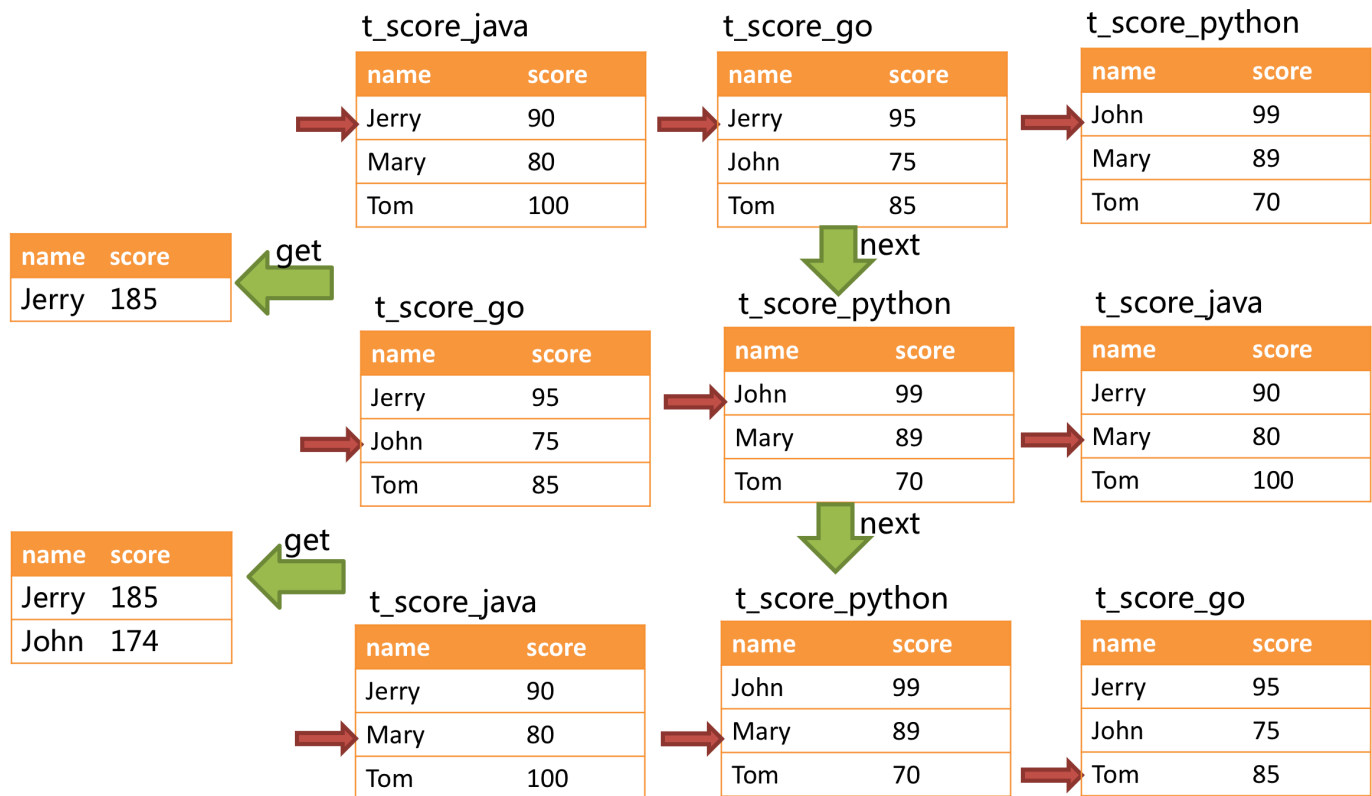
name	score
Jerry	95
John	75
Tom	85



t_score_python

name	score
John	99
Mary	89
Tom	70

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。



通过图中我们可以看到，当进行第一次 `next` 调用时，排在队列首位的 `t_score_java` 将会被弹出队列，并且将分组值同为 “Jerry” 的其他结果集中的数据一同弹出队列。在获取了所有的姓名为 “Jerry” 的同学的分数之后，进行累加操作，那么，在第一次 `next` 调用结束后，取出的结果集是 “Jerry” 的分数总和。与此同时，所有的数据结果集中的游标都将下移至数据值 “Jerry” 的下一个不同的数据值，并且根据数据结果集当前游标指向的值进行重排序。因此，包含名字顺着第二位的 “John” 的相关数据结果集则排在的队列的前列。

流式分组归并与排序归并的区别仅仅在于两点：

- 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。
- 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：

```
1 | 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。

4. 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。

5. 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。

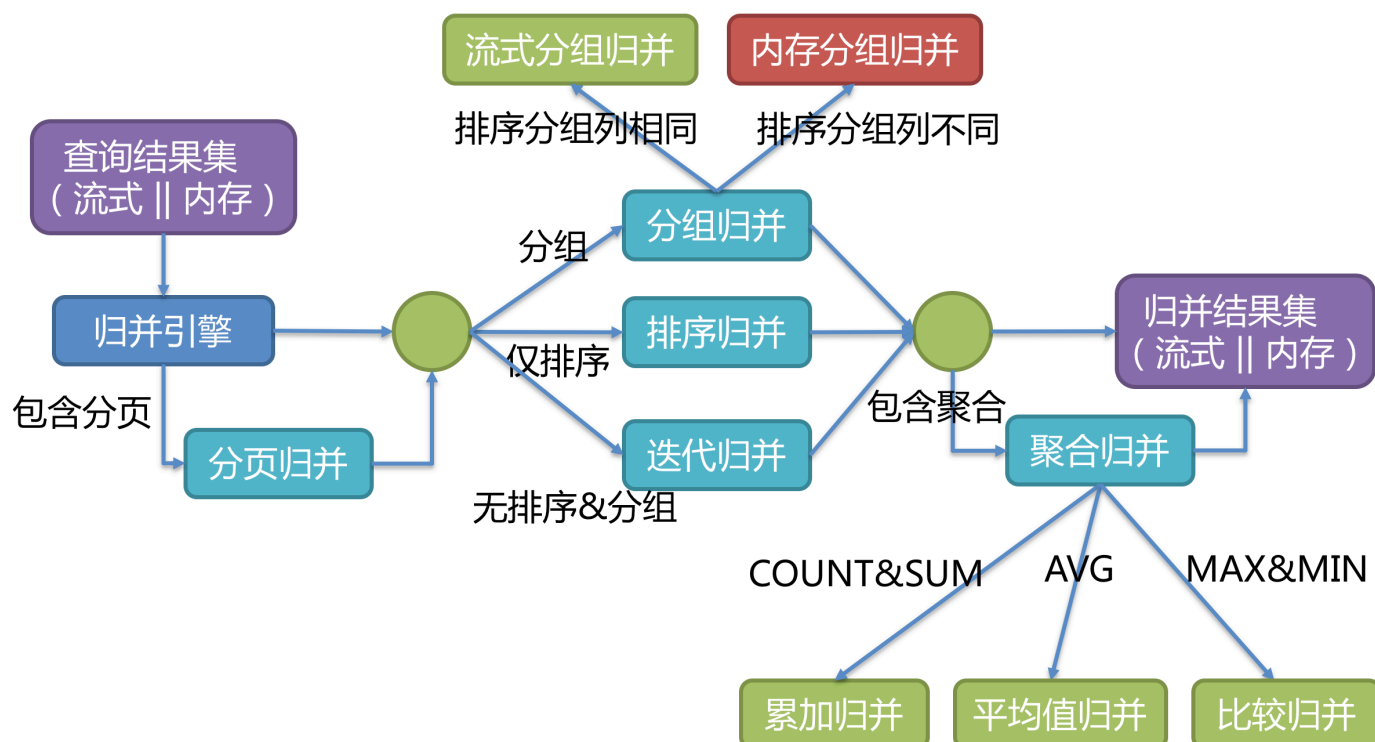
但同时需要注意的是，由于排序的需要，大量的数据仍然需要传输到 ShardingSphere 的内存空间。因此，采用 `LIMIT` 这种方式分页，并非最佳实践。由于 `LIMIT` 并不能通过索引查询数据，因此如果可以保证 ID 的连续性，通过 ID 进行分页是比较好的解决方案，例如：

```
1 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：

```
1 SELECT * FROM t_order WHERE id > 10000000 LIMIT 10;
```

归并引擎的整体结构划分如下图。



千锋教育Java教研院 关注公众号【Java架构栈】下载所有课程代码课件及工具 让技术回归本该有的纯静！