Java第四阶段\_day01\_Sharding-jdbc案例

## 1.1 **需求说明**

本章节使用Sharding-JDBC完成对订单表的水平分表，通过快速入门程序的开发，快速体验Sharding-JDBC的使用方法。

人工创建两张表，t\_order\_1和t\_order\_2，这两张表是订单表拆分后的表，通过Sharding-Jdbc向订单表插入数据， 按照一定的分片规则，主键为偶数的进入t\_order\_1，另一部分数据进入t\_order\_2，通过Sharding-Jdbc 查询数据，根据 SQL语句的内容从t\_order\_1或t\_order\_2查询数据。

## 1.2. 环境搭建

### 1.2.1 **环境说明**

操作系统：Win10

数据库：MySQL-5.7.25 JDK：64位 jdk1.8.0\_201

应用框架：spring-boot-2.1.3.RELEASE，Mybatis3.5.0 Sharding-JDBC：sharding-jdbc-spring-boot-starter-4.0.0-RC1

### 1.2.2 **创建数据库**

创建订单库order\_db

CREATE DATABASE `order\_db` CHARACTER SET 'utf8' COLLATE 'utf8\_general\_ci';

在order\_db中创建t\_order\_1、t\_order\_2表

DROP TABLE IF EXISTS `t\_order\_1`;   
CREATE TABLE `t\_order\_1` (  
`order\_id` bigint(20) NOT NULL COMMENT '订单id',  
`price` decimal(10, 2) NOT NULL COMMENT '订单价格',  
`user\_id` bigint(20) NOT NULL COMMENT '下单用户id',  
`status` varchar(50) CHARACTER SET utf8 COLLATE utf8\_general\_ci NOT NULL COMMENT '订单状态',  
PRIMARY KEY (`order\_id`) USING BTREE  
) ENGINE = InnoDB CHARACTER SET = utf8 COLLATE = utf8\_general\_ci ROW\_FORMAT = Dynamic;  
​  
DROP TABLE IF EXISTS `t\_order\_2`;   
CREATE TABLE `t\_order\_2` (  
`order\_id` bigint(20) NOT NULL COMMENT '订单id',  
`price` decimal(10, 2) NOT NULL COMMENT '订单价格',  
`user\_id` bigint(20) NOT NULL COMMENT '下单用户id',  
`status` varchar(50) CHARACTER SET utf8 COLLATE utf8\_general\_ci NOT NULL COMMENT '订单状态',  
PRIMARY KEY (`order\_id`) USING BTREE  
) ENGINE = InnoDB CHARACTER SET = utf8 COLLATE = utf8\_general\_ci ROW\_FORMAT = Dynamic;

### **1.2.3.引入maven依赖**

引入 sharding-jdbc和SpringBoot整合的Jar包：

<dependency>  
    <groupId>org.apache.shardingsphere</groupId>  
    <artifactId>sharding‐jdbc‐spring‐boot‐starter</artifactId>  
    <version>4.0.0‐RC1</version>  
</dependency>

具体spring boot相关依赖及配置请参考资料中sharding-jdbc-simple工程，本指引只说明与Sharding- JDBC相关的内容。

## 1.3 **编写程序**

### 1.3.1 **分片规则配置**

分片规则配置是sharding-jdbc进行对分库分表操作的重要依据，配置内容包括：数据源、主键生成策略、分片策 略等。

在application.properties中配置

server.port=56081  
​  
spring.application.name = sharding‐jdbc‐simple‐demo  
server.servlet.context‐path = /sharding‐jdbc‐simple‐demo spring.http.encoding.enabled = true spring.http.encoding.charset = UTF‐8 spring.http.encoding.force = true  
​  
spring.main.allow‐bean‐definition‐overriding = true  
​  
mybatis.configuration.map‐underscore‐to‐camel‐case = true # 以下是分片规则配置  
# 定义数据源  
spring.shardingsphere.datasource.names = m1  
​  
spring.shardingsphere.datasource.m1.type = com.alibaba.druid.pool.DruidDataSource spring.shardingsphere.datasource.m1.driver‐class‐name = com.mysql.jdbc.Driver spring.shardingsphere.datasource.m1.url = jdbc:mysql://localhost:3306/order\_db?useUnicode=true spring.shardingsphere.datasource.m1.username = root spring.shardingsphere.datasource.m1.password = root  
​  
# 指定t\_order表的数据分布情况，配置数据节点  
spring.shardingsphere.sharding.tables.t\_order.actual‐data‐nodes = m1.t\_order\_$‐>{1..2}  
​  
# 指定t\_order表的主键生成策略为SNOWFLAKE spring.shardingsphere.sharding.tables.t\_order.key‐generator.column=order\_id spring.shardingsphere.sharding.tables.t\_order.key‐generator.type=SNOWFLAKE  
​  
# 指定t\_order表的分片策略，分片策略包括分片键和分片算法spring.shardingsphere.sharding.tables.t\_order.table‐strategy.inline.sharding‐column = order\_id   
spring.shardingsphere.sharding.tables.t\_order.table‐strategy.inline.algorithm‐expression = t\_order\_$‐>{order\_id % 2 + 1}  
​  
# 打开sql输出日志  
spring.shardingsphere.props.sql.show = true swagger.enable = true  
logging.level.root = info logging.level.org.springframework.web = info logging.level.com.bigdata.dbsharding = debug   
logging.level.druid.sql = debug

1. 首先定义数据源m1，并对m1进行实际的参数配置。

2. 指定t\_order表的数据分布情况，他分布在m1.t\_order\_1，m1.t\_order\_2

3. 指定t\_order表的主键生成策略为SNOWFLAKE，SNOWFLAKE是一种分布式自增算法，保证id全局唯一

4. 定义t\_order分片策略，order\_id为偶数的数据落在t\_order\_1，为奇数的落在t\_order\_2，分表策略的表达式为t\_order\_$->{order\_id % 2 + 1}

### 1.3.2. **数据操作**

@Mapper

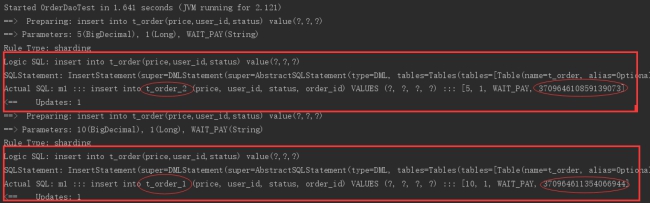
@Component  
public interface OrderDao {  
​  
/\*\*  
\*新增订单  
\*@param price 订单价格  
\*@param userId 用户id  
\*@param status 订单状态  
\*@return  
\*/  
@Insert("insert into t\_order(price,user\_id,status) value(#{price},#{userId},#{status})") int insertOrder(@Param("price") BigDecimal price, @Param("userId")Long userId,  
@Param("status")String status);  
​  
/\*\*  
\*根据id列表查询多个订单  
\*@param orderIds 订单id列表  
\*@return  
\*/ @Select({"<script>" +  
"select " + " \* " +  
" from t\_order t" +  
" where t.order\_id in " +  
"<foreach collection='orderIds' item='id' open='(' separator=',' close=')'>" + " #{id} " +  
"</foreach>"+ "</script>"})  
List<Map> selectOrderbyIds(@Param("orderIds")List<Long> orderIds);  
​  
}

2.3.3. 测试

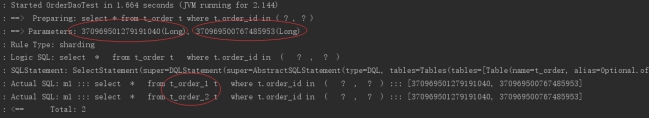
编写单元测试：

@RunWith(SpringRunner.class)  
@SpringBootTest(classes = {ShardingJdbcSimpleDemoBootstrap.class})   
public class OrderDaoTest {  
    @Autowired  
    private OrderDao orderDao;   
    @Test  
    public void testInsertOrder(){ for (int i = 0 ; i<10; i++){  
   orderDao.insertOrder(new BigDecimal((i+1)\*5),1L,"WAIT\_PAY");  
   }  
    @Test  
    public void testSelectOrderbyIds(){   
        List<Long> ids = new ArrayList<>(); ids.add(373771636085620736L);             ids.add(373771635804602369L);  
        List<Map> maps = orderDao.selectOrderbyIds(ids); System.out.println(maps);  
   }  
}

执行testInsertOrder：



通过日志可以发现order\_id为奇数的被插入到t\_order\_2表，为偶数的被插入到t\_order\_1表，达到预期目标。执行testSelectOrderbyIds：



通过日志可以发现，根据传入order\_id的奇偶不同，sharding-jdbc分别去不同的表检索数据，达到预期目标。

## 1.4. **流程分析**

通过日志分析，Sharding-JDBC在拿到用户要执行的sql之后干了哪些事儿：

（1） 解析sql，获取片键值，在本例中是order\_id

（2） Sharding-JDBC通过规则配置 t\_order\_$->{order\_id % 2 + 1}，知道了当order\_id为偶数时，应该往t\_order\_1表插数据，为奇数时，往t\_order\_2插数据。

（3） 于是Sharding-JDBC根据order\_id的值改写sql语句，改写后的SQL语句是真实所要执行的SQL语句。

（4） 执行改写后的真实sql语句

（5） 将所有真正执行sql的结果进行汇总合并，返回。

## 1.5. **其他集成方式**

Sharding-JDBC不仅可以与spring boot良好集成，它还支持其他配置方式，共支持以下四种集成方式。

Spring Boot Yaml 配置

定义application.yml，内容如下：

server:  
 port: 56081  
 servlet:  
   context-path: /sharding-jdbc-simple-demo  
spring:  
 application:  
   name: sharding-jdbc-simple-demo  
 http:  
   encoding:  
     enabled: true  
     charset: utf-8  
     force: true  
 main:  
   allow-bean-definition-overriding: true  
 shardingsphere:  
   datasource:  
     names: m1  
     m1:  
       type: com.alibaba.druid.pool.DruidDataSource  
       driverClassName: com.mysql.jdbc.Driver  
       url: jdbc:mysql://localhost:3306/order\_db?useUnicode=true  
       username: root  
       password: mysql  
   sharding:  
     tables:  
       t\_order:  
         actualDataNodes: m1.t\_order\_$->{1..2}  
         tableStrategy:  
           inline:  
             shardingColumn: order\_id  
             algorithmExpression: t\_order\_$->{order\_id % 2 + 1}  
         keyGenerator:  
           type: SNOWFLAKE  
           column: order\_id  
   props:  
     sql:  
       show: true  
mybatis:  
 configuration:  
   map-underscore-to-camel-case: true  
swagger:  
 enable: true  
logging:  
 level:  
   root: info  
   org.springframework.web: info  
   com.bigdata.dbsharding: debug  
   druid.sql: debug  
​

如果使用application.yml则需要屏蔽原来的application.properties文件。

#### **Java 配置**

添加配置类：

@Configuration  
public class ShardingJdbcConfig {  
​  
    // 定义数据源  
    Map<String, DataSource> createDataSourceMap() {   
        DruidDataSource dataSource1 = new DruidDataSource();           dataSource1.setDriverClassName("com.mysql.jdbc.Driver");  
        dataSource1.setUrl("jdbc:mysql://localhost:3306/order\_db?useUnicode=true");            dataSource1.setUsername("root");  
        dataSource1.setPassword("root");  
        Map<String, DataSource> result = new HashMap<>(); result.put("m1", dataSource1);  
        return result;  
   }  
// 定义主键生成策略  
    private static KeyGeneratorConfiguration getKeyGeneratorConfiguration() { KeyGeneratorConfiguration result = new  
    KeyGeneratorConfiguration("SNOWFLAKE","order\_id"); return result;  
 }  
​  
    // 定义t\_order表的分片策略  
    TableRuleConfiguration getOrderTableRuleConfiguration() {  
        TableRuleConfiguration result = new TableRuleConfiguration("t\_order","m1.t\_order\_$‐>  
       {1..2}");  
        result.setTableShardingStrategyConfig(new InlineShardingStrategyConfiguration("order\_id", "t\_order\_$‐>{order\_id % 2 + 1}"));  
        result.setKeyGeneratorConfig(getKeyGeneratorConfiguration());  
​  
        return result;  
   }  
    // 定义sharding‐Jdbc数据源  
    @Bean  
    DataSource getShardingDataSource() throws SQLException {  
        ShardingRuleConfiguration shardingRuleConfig = new ShardingRuleConfiguration(); shardingRuleConfig.getTableRuleConfigs().add(getOrderTableRuleConfiguration());  
        //spring.shardingsphere.props.sql.show = true   
        Properties properties = new Properties();   
        properties.put("sql.show","true");  
        return ShardingDataSourceFactory.createDataSource(createDataSourceMap(),  
        shardingRuleConfig,properties);  
   }  
}

由于采用了配置类所以需要屏蔽原来application.properties文件中spring.shardingsphere开头的配置信息。还需要在SpringBoot启动类中屏蔽使用spring.shardingsphere配置项的类：

@SpringBootApplication(exclude = {SpringBootConfiguration.class}) public class ShardingJdbcSimpleDemoBootstrap { }

Spring Boot properties配置

此方式同快速入门程序。

# 定义数据源  
spring.shardingsphere.datasource.names = m1  
​  
spring.shardingsphere.datasource.m1.type = com.alibaba.druid.pool.DruidDataSource spring.shardingsphere.datasource.m1.driver‐class‐name = com.mysql.jdbc.Driver spring.shardingsphere.datasource.m1.url = jdbc:mysql://localhost:3306/order\_db?useUnicode=true spring.shardingsphere.datasource.m1.username = root spring.shardingsphere.datasource.m1.password = root  
​  
​  
# 指定t\_order表的主键生成策略为SNOWFLAKE spring.shardingsphere.sharding.tables.t\_order.key‐generator.column=order\_id spring.shardingsphere.sharding.tables.t\_order.key‐generator.type=SNOWFLAKE  
​  
# 指定t\_order表的数据分布情况  
spring.shardingsphere.sharding.tables.t\_order.actual‐data‐nodes = m1.t\_order\_$‐>{1..2}  
​  
# 指定t\_order表的分表策略  
spring.shardingsphere.sharding.tables.t\_order.table‐strategy.inline.sharding‐column = order\_id   
spring.shardingsphere.sharding.tables.t\_order.table‐strategy.inline.algorithm‐expression = t\_order\_$‐>{order\_id % 2 + 1}