Sharding-jdbc

Sharding-JDBC是ShardingSphere的第一个产品，也是ShardingSphere的前身。 它定位为轻量级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。

- apache官方使用手册

1.分库分表简述

垂直分表：可以把一个宽表的字段按访问频次、是否是大字段的原则拆分为多个表，这样既能使业务清晰，还能提升部分性能。拆分后，尽量从业务角度避免联查，否则性能方面将得不偿失。

垂直分库：可以把多个表按业务耦合松紧归类，分别存放在不同的库，这些库可以分布在不同服务器，从而使访问压力被多服务器负载，大大提升性能，同时能提高整体架构的业务清晰度，不同的业务库可根据自身情况定制优化方案。但是它需要解决跨库带来的所有复杂问题。

水平分库：可以把一个表的数据(按数据行)分到多个不同的库，每个库只有这个表的部分数据，这些库可以分布在不同服务器，从而使访问压力被多服务器负载，大大提升性能。它不仅需要解决跨库带来的所有复杂问题，还要解决数据路由的问题(数据路由问题后边介绍)。

水平分表：可以把一个表的数据(按数据行)分到多个同一个数据库的多张表中，每个表只有这个表的部分数据，这样做能小幅提升性能，它仅仅作为水平分库的一个补充优化。

一般来说，在系统设计阶段就应该根据业务耦合松紧来确定垂直分库，垂直分表方案，在数据量及访问压力不是特别大的情况，首先考虑缓存、读写分离、索引技术等方案。若数据量极大，且持续增长，再考虑水平分库水平分表方案。

2.使用说明

2.1 maven引入

```
<sharding-jdbc.version>4.0.0-RC1</sharding-jdbc.version>
<dependency>
    <groupId>org.apache.shardingsphere</groupId>
    <artifactId>sharding-jdbc-spring-boot-starter</artifactId>
    <version>${sharding-jdbc.version}</version>
</dependency>
```

2.2分库分表配置

2.2.1数据源配置

```
#sharding-jdbc分片规则
#配置数据源 m0,m1,m2,s0,s1,s2
spring.shardingsphere.datasource.names = m0,m1,m2,s0,s1,s2

spring.shardingsphere.datasource.m0.type = com.alibaba.druid.pool.DruidDataSource
spring.shardingsphere.datasource.m0.driver-class-name = com.mysql.jdbc.Driver
spring.shardingsphere.datasource.m0.url = jdbc:mysql://localhost:3311/store_db?serverTimezone=Asia/Shanghai&useSSL=false&useUnicode
spring.shardingsphere.datasource.m0.username = root
spring.shardingsphere.datasource.m0.password = root@localhost

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:3311/product_db_1?serverTimezone=Asia/Shanghai&useSSL=false&useUni
spring.shardingsphere.datasource.m1.username = root
spring.shardingsphere.datasource.m1.password = root@localhost

spring.shardingsphere.datasource.m2.type = com.alibaba.druid.pool.DruidDataSource
spring.shardingsphere.datasource.m2.driver-class-name = com.mysql.jdbc.Driver
spring.shardingsphere.datasource.m2.url = jdbc:mysql://localhost:3311/product_db_2?serverTimezone=Asia/Shanghai&useSSL=false&useUni
spring.shardingsphere.datasource.m2.username = root
spring.shardingsphere.datasource.m2.password = root@localhost

spring.shardingsphere.datasource.s0.type = com.alibaba.druid.pool.DruidDataSource
spring.shardingsphere.datasource.s0.driver-class-name = com.mysql.jdbc.Driver
spring.shardingsphere.datasource.s0.url = jdbc:mysql://localhost:3311/store_db?serverTimezone=Asia/Shanghai&useSSL=false&useUnicode
spring.shardingsphere.datasource.s0.username = root
spring.shardingsphere.datasource.s0.password = root@localhost

spring.shardingsphere.datasource.s1.type = com.alibaba.druid.pool.DruidDataSource
spring.shardingsphere.datasource.s1.driver-class-name = com.mysql.jdbc.Driver
spring.shardingsphere.datasource.s1.url = jdbc:mysql://localhost:3311/product_db_1?serverTimezone=Asia/Shanghai&useSSL=false&useUni
spring.shardingsphere.datasource.s1.username = root
spring.shardingsphere.datasource.s1.password = root@localhost

spring.shardingsphere.datasource.s2.type = com.alibaba.druid.pool.DruidDataSource
spring.shardingsphere.datasource.s2.driver-class-name = com.mysql.jdbc.Driver
spring.shardingsphere.datasource.s2.url = jdbc:mysql://localhost:3311/product_db_2?serverTimezone=Asia/Shanghai&useSSL=false&useUni
spring.shardingsphere.datasource.s2.username = root
spring.shardingsphere.datasource.s2.password = root@localhost
```

数据源别名，可以自定义

2.2.2读写分离配置

```
# 读写分离，主库从库逻辑数据源定义 ds0为user_db
spring.shardingsphere.sharding.master-slave-rules.ds0.master-data-source-name=m0
spring.shardingsphere.sharding.master-slave-rules.ds0.slave-data-source-names=s0

# t_user分表策略，固定分配至ds0的t_user真实表
spring.shardingsphere.sharding.tables.t_user.actual-data-nodes = ds$->{0}.t_user
# 未做读写分离时配置如下
#spring.shardingsphere.sharding.tables.t_user.actual-data-nodes = m$->{0}.t_user
```

数据源别名

读写分离表别名

分库分表策略配置表达式，此处user表未做分库和分表，如果做了分库分表，eg: ds$->{0..1}.t_user_{1..2}

2.2.3分库分表策略配置

```
# order信息分库策略，以user_id为分片键，分片策略为user_id % 2 + 1，user_id为偶数操作m1数据源，否则操作m2。
spring.shardingsphere.sharding.tables.t_order.database-strategy.inline.sharding-column = user_id
spring.shardingsphere.sharding.tables.t_order.database-strategy.inline.algorithm-expression = m$->{user_id % 2 + 1}

# 指定t_order表的数据分布情况，配置数据节点 m1.t_order_1,m1.t_order_2,m2.t_order_1,m2.t_order_2
spring.shardingsphere.sharding.tables.t_order.actual-data-nodes = m$->{1..2}.t_order_$->{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}
```

分片列名

分库

分片表达式

雪花算法

数据节点，Sharding-JDBC会根据它索引逻辑表对应的真实表

分表

注order表以user_id进行分库`m$->{user_id % 2 + 1}`user_id为偶数分到`m1`库，奇数分到`m2`库。

2.2.4广播表

```
# 指定t_dict为公共表
spring.shardingsphere.sharding.broadcast-tables=t_dict

# 打开sql输出日志
spring.shardingsphere.props.sql.show = true
```

广播表

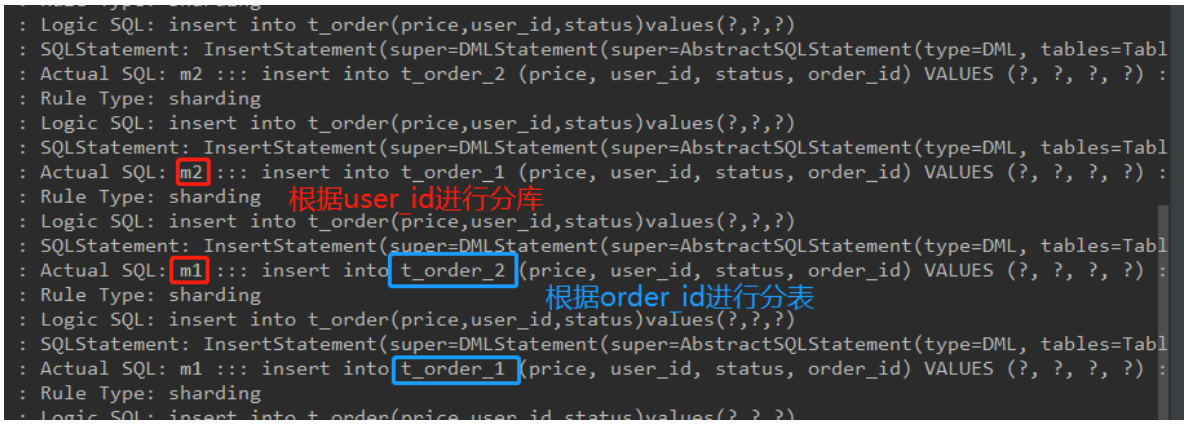
3.SQL使用

3.1 Sql操作

执行示例：

```
    /**
     * 插入订单
     * @param price 价格
     * @param userId 用户id
     * @param status 订单状态
     * @return 执行条数
     */
    @Insert("insert into t_order(price,user_id,status)values(#{price},#{userId},#{status})")
    int insertOrder(@Param("price") BigDecimal price, @Param("userId") Long userId, @Param("status") String status);

    @Test
    public void testInsertOrder() {
        for (int i = 0; i < 10; i++) {
            orderDao.insertOrder(new BigDecimal((i + 1) * 5), 1L, "WAIT_PAY");
        }
        for (int i = 0; i < 10; i++) {
            orderDao.insertOrder(new BigDecimal((i + 1) * 10), 2L, "WAIT_PAY");
        }
    }
```



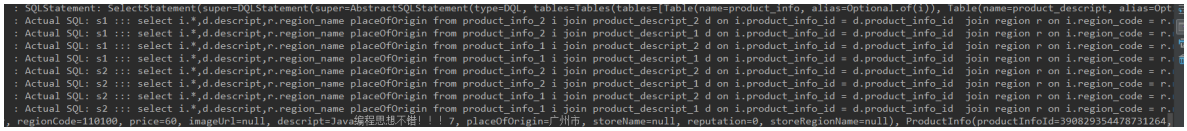
3.2关联表查询--shopping

```
# 设置product_info,product_descript为绑定表
spring.shardingsphere.sharding.binding-tables[0] = product_info,product_descript
```

注 数组下标从0开始。

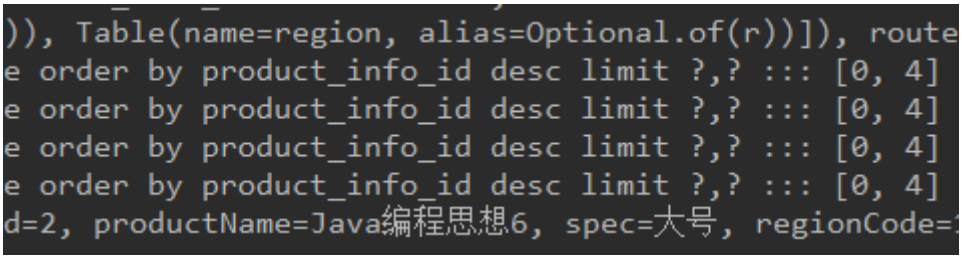
配置后会在本库表中进行关联查询。

取消配置后会进行笛卡尔积查询：



使用limit查询时，sharding-jdbc会将起始数和分页数重新计算

```
List<ProductInfo> productInfos = productService.queryProduct(2, 2);
```



4.sharding-jdbc分布式事务

目前sharding-jdbc支持的分布式事务有两阶段事务(xa)和柔性事务(base)，简单介绍如下，有兴趣的小伙伴可以自己百度了解具体介绍。

```
 * 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。
 * 两阶段事务：
 * --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协议两阶段提交类型的分布式事务并不是最佳选择。
 * 柔性事务：
 * --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和隔离性的要求，只要求当整个事务最终结束的时候，数据是一致的。
 * --而在事务执行期间，任何读取操作得到的数据都有可能被改变。这种弱一致性的设计可以用来换取系统吞吐量的提升。
```

- [sharding-jdbc官方分布式事务解决方案]

4.1引入pom依赖

- XA事务

```
        <!--两段事务-->
        <dependency>
            <groupId>org.apache.shardingsphere</groupId>
            <artifactId>sharding-transaction-xa-core</artifactId>
        </dependency>
```

- 柔性事务官方给出的maven位置是：

```
io.shardingsphere
sharding-transaction-base-saga
${shardingsphere-spi-impl.version}
```

org.apache.shardingsphere
sharding-transaction-base-seata-at
${sharding-sphere.version}

4.2分布式事务实现

在需要回滚的位置添加代码（不需要额外的application.properties配置）：

`TransactionTypeHolder.set(TransactionType.XA);`

```
    @Transactional(rollbackFor = {Exception.class})
    @Override
    public int insertUser1(Long userId, String userNamePlain, String userName,
String userType, String pwd) {
        TransactionTypeHolder.set(TransactionType.XA);
        int i = userDao.insertUser(userId, userNamePlain, userName, userType,
pwd);
        int j = 1 / 0;
        return i;
    }
```

或者像io.shardingshpere事务使用一样

```
@Transactional(rollbackFor = {Exception.class})
    @Override
    public int insertUser(Long userId, String userNamePlain, String userName,
String userType, String pwd) {
        try {
            TransactionTypeHolder.set(TransactionType.XA);
            int i = userDao.insertUser(userId, userNamePlain, userName,
userType, pwd);
            int j = 1 / 0;
            return i;
        } catch (Exception e) {
            // 使用io.shardingsphere的ShardingTransaction需要加上下面方法调用，不然会
回滚失败

 TransactionAspectSupport.currentTransactionStatus().setRollbackOnly();
            LOGGER.error(NestedExceptionUtils.buildMessage("添加新用户[userId={}]
异常：", e), userId);
        }
        return 0;
    }
```

- io.shardingshpere事务

```
    /**
     * ---@ShardingTransactionType: 需要同Spring的@Transactional配套使用，事务才会生
效
     * ---TransactionType.LOCAL 本地事务
     * ---TransactionType.XA 两阶段事务（支持夸库事务）
     * ---TransactionType.BASE 柔性事务
     */
    @ShardingTransactionType(TransactionType.XA)
```

```
    @Transactional(rollbackFor = {Exception.class})
    @Override
    public Integer insertUser(Long userId, String userNamePlain, String
userName, String userType, String pwd) {
        try {
            int result = userDao.insertUser(userId, userNamePlain, userName,
userType, pwd);
            int i = 1 / 0;
            return result;
        } catch (Exception e) {
            // io.shardingsphere ShardingTransaction需要加上下面方法调用，不然会回滚失
败

 TransactionAspectSupport.currentTransactionStatus().setRollbackOnly();
            LOGGER.error(NestedExceptionUtils.buildMessage("添加新用户[userId={}]
异常：", e), userId);
        }
        return null;
    }
```

- 值得一提的是，sharding-jdbc分布式事务需要添加`@Transactional`注解，否则回滚会失败：

具体使用方法可参考[sharding分布式事务]