Shardingsphere 主从+分表

背景

为了满足公司业务拓展，提升数据库IO性能，需要对数据库数据量巨大的表进行拆表，因此我们引用shardingsphere技术实现分表策略。

引用shardingsphere分表

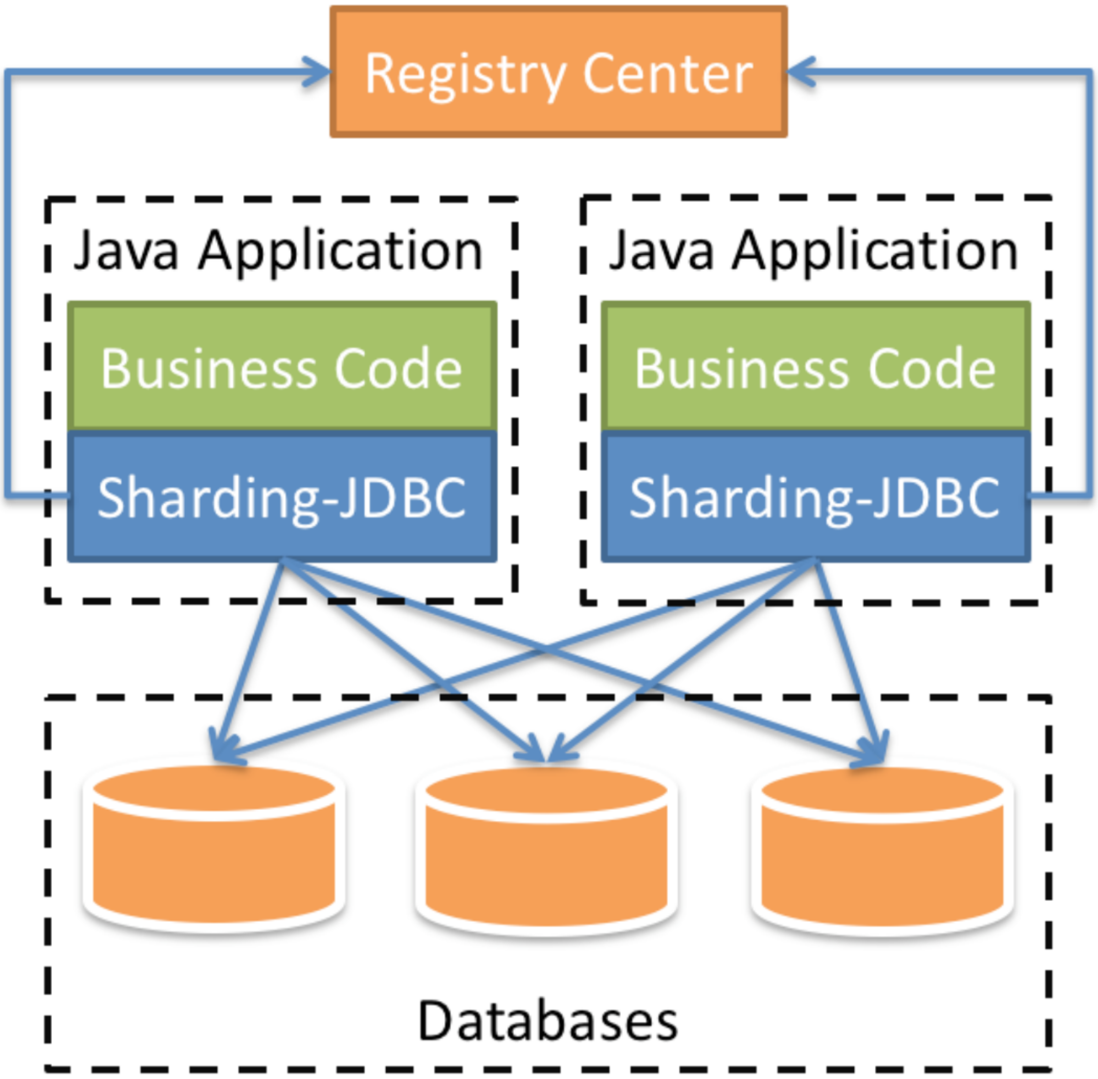
Apache ShardingSphere 是一套开源的分布式数据库中间件解决方案组成的生态圈，它由 JDBC、Proxy

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。

实现原理



分表配置

指定数据库

```
# 指定数据源名称
spring.shardingsphere.datasource.names=ds
```

数据源配置

```
# 数据源配置，在本项目中使用 jdbc-url，但在Fmapp中请使用 url
spring.shardingsphere.datasource.ds.type=com.zaxxer.hikari.HikariDataSource
spring.shardingsphere.datasource.ds.driver-class-name=com.mysql.jdbc.Driver
spring.shardingsphere.datasource.ds.jdbc-url=jdbc:mysql://localhost:3306/demo_ds?serverTimezone=UTC&useSSL=false&useUnicode=true&characterEncoding=UTF-8
spring.shardingsphere.datasource.ds.username=root
spring.shardingsphere.datasource.ds.password=root
```

复合分片自定义分表算法

```
# 标准分表，自定义分表算法，CRC32 hash再取模分表
spring.shardingsphere.sharding.tables.t_order.actual-data-nodes=ds.t_order_$->{0..6}
spring.shardingsphere.sharding.tables.t_order.table-strategy.standard.sharding-column=order_id
spring.shardingsphere.sharding.tables.t_order.table-strategy.standard.precise-algorithm-class-name=com.x2era.sharding.sharding.table.algorithm.TablePreciseShardingAlgorithm
spring.shardingsphere.sharding.tables.t_order_item.actual-data-nodes=ds.t_order_item_$->{0..6}
spring.shardingsphere.sharding.tables.t_order_item.table-strategy.standard.sharding-column=order_id
spring.shardingsphere.sharding.tables.t_order_item.table-strategy.standard.precise-algorithm-class-name=com.x2era.sharding.sharding.table.algorithm.TablePreciseShardingAlgorithm
```

基本数据类型自定义分表表达式方式

```
# 基本数据类型取模分表
spring.shardingsphere.sharding.tables.t_address.actual-data-nodes=ds.t_address_$->{0..2}
spring.shardingsphere.sharding.tables.t_address.table-strategy.inline.sharding-column=user_id
spring.shardingsphere.sharding.tables.t_address.table-strategy.inline.algorithm-expression=t_address_$->{user_id % 3}
```

联表绑定关系

```
# 确定联表绑定关系，绑定表之间的分区键要完全相同
spring.shardingsphere.sharding.binding-tables=t_order,t_order_item
```

自定义分表规则

复合分表可以使开发者自定义分表规则。根据分表的字段的值，hash取模后匹配到真实执行的表

```
public class TablePreciseShardingAlgorithm implements
PreciseShardingAlgorithm<String> {

    @Override
    public String doSharding(Collection<String> availableTargetNames,
PreciseShardingValue<String> shardingValue) {

        final String value = shardingValue.getValue();
        final String columnName = shardingValue.getColumnName();
        final String logicTableName = shardingValue.getLogicTableName();
        final int availableTableSize = availableTargetNames.size();

        System.out.println(String.format("执行业务开始分表流程，开始寻找真实执行的业务表
tableName = %s, columnName = %s, tableColumnValue = %s",
                logicTableName, columnName, value));

        long tableIndex = CRC32Utils.hash(value) % availableTableSize;
        String tableName = logicTableName + "_" + tableIndex;
        System.out.println(String.format("寻找真实的业务表 tableName = %s",
tableName));

        AtomicReference<String> resultTableName = new AtomicReference<>
(logicTableName);
        availableTargetNames.stream().filter(table ->
table.equals(tableName)).findFirst().ifPresent(resultTableName::set);
        return resultTableName.get();
    }
}
```

hash算法

hash算法使用JDK内置自带的算法CRC32，CRC32`mysql`也有自身的内置函数，所以未来如果做数据迁移也是非常好的选择

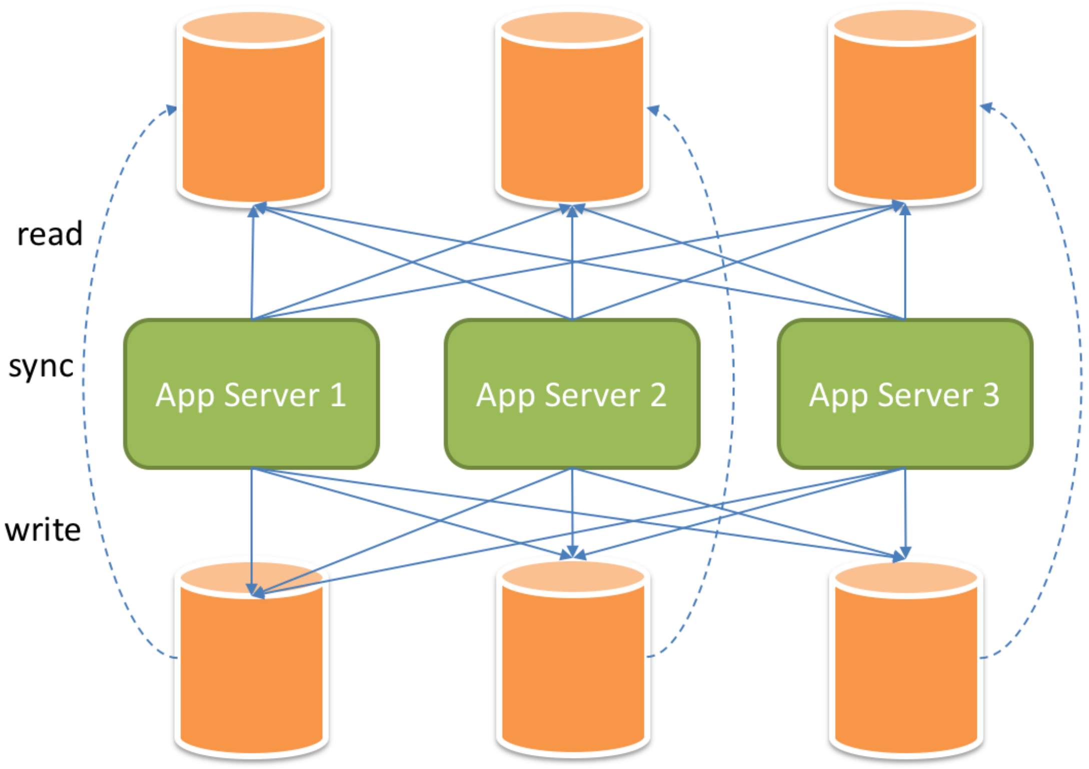
```
/**
  * 计算hash值
  * @param key 需要hash计算的值
  * @return 返回hash值
  */
private long hash(final String key) {
    byte[] b = key.getBytes();
    CRC32 c = new CRC32();
    // Resets CRC-32 to initial value.
    c.reset();
    // 将数据丢入CRC32解码器
    c.update(b, 0, b.length);
    // 获取CRC32 的值  默认返回值类型为long 用于保证返回值是一个正数
    return c.getValue();
}
```

读写分离原理

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。

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。



主从配置

```
# 指定数据源名称
spring.shardingsphere.datasource.names=ds-master,ds-slave-0,ds-slave-1
# 主库
spring.shardingsphere.datasource.ds-
master.type=com.zaxxer.hikari.HikariDataSource
spring.shardingsphere.datasource.ds-master.driver-class-
name=com.mysql.jdbc.Driver
spring.shardingsphere.datasource.ds-master.jdbc-
url=jdbc:mysql://localhost:3306/ds_master?
serverTimezone=UTC&useSSL=false&useUnicode=true&characterEncoding=UTF-8
spring.shardingsphere.datasource.ds-master.username=root
spring.shardingsphere.datasource.ds-master.password=root
# 从库0
spring.shardingsphere.datasource.ds-slave-
0.type=com.zaxxer.hikari.HikariDataSource
spring.shardingsphere.datasource.ds-slave-0.driver-class-
name=com.mysql.jdbc.Driver
```

```
spring.shardingsphere.datasource.ds-slave-0.jdbc-
url=jdbc:mysql://localhost:3306/ds_slave_0?
serverTimezone=UTC&useSSL=false&useUnicode=true&characterEncoding=UTF-8
spring.shardingsphere.datasource.ds-slave-0.username=root
spring.shardingsphere.datasource.ds-slave-0.password=root
# 从库1
spring.shardingsphere.datasource.ds-slave-
1.type=com.zaxxer.hikari.HikariDataSource
spring.shardingsphere.datasource.ds-slave-1.driver-class-
name=com.mysql.jdbc.Driver
spring.shardingsphere.datasource.ds-slave-1.jdbc-
url=jdbc:mysql://localhost:3306/ds_slave_1?
serverTimezone=UTC&useSSL=false&useUnicode=true&characterEncoding=UTF-8
spring.shardingsphere.datasource.ds-slave-1.username=root
spring.shardingsphere.datasource.ds-slave-1.password=root

# 主从核心配置
# 从库查询规则，轮询方式
spring.shardingsphere.masterslave.load-balance-algorithm-type=round_robin
# 此配置主从方式必须配置
spring.shardingsphere.masterslave.name=ds_ms
# 指定主库数据源名称
spring.shardingsphere.masterslave.master-data-source-name=ds-master
# 指定从库数据源名称
spring.shardingsphere.masterslave.slave-data-source-names=ds-slave-0,ds-slave-1
```

主从+分表配置

```
spring.shardingsphere.datasource.names=ds-master,ds-slave
# 主库
spring.shardingsphere.datasource.ds-
master.type=com.zaxxer.hikari.HikariDataSource
spring.shardingsphere.datasource.ds-master.driver-class-
name=com.mysql.jdbc.Driver
spring.shardingsphere.datasource.ds-master.jdbc-
url=jdbc:mysql://localhost:3306/demo_ds_master?
serverTimezone=UTC&useSSL=false&useUnicode=true&characterEncoding=UTF-8
spring.shardingsphere.datasource.ds-master.username=root
spring.shardingsphere.datasource.ds-master.password=root
# 从库
spring.shardingsphere.datasource.ds-
slave.type=com.zaxxer.hikari.HikariDataSource
spring.shardingsphere.datasource.ds-slave.driver-class-
name=com.mysql.jdbc.Driver
spring.shardingsphere.datasource.ds-slave.jdbc-
url=jdbc:mysql://localhost:3306/demo_ds_slave?
serverTimezone=UTC&useSSL=false&useUnicode=true&characterEncoding=UTF-8
spring.shardingsphere.datasource.ds-slave.username=root
spring.shardingsphere.datasource.ds-slave.password=root
# 一主一从规则配置
spring.shardingsphere.sharding.master-slave-rules.ds.master-data-source-name=ds-
master
spring.shardingsphere.sharding.master-slave-rules.ds.slave-data-source-names=ds-
slave
# 标准分表，自定义分表算法，CRC32 hash再取模分表
spring.shardingsphere.sharding.tables.t_order.actual-data-nodes=ds.t_order_$->
{0..3}
```

```
spring.shardingsphere.sharding.tables.t_order.table-strategy.standard.sharding-column=order_id
spring.shardingsphere.sharding.tables.t_order.table-strategy.standard.precise-algorithm-class-name=com.x2era.sharding.sharding.table.algorithm.TablePreciseShardingAlgorithm
spring.shardingsphere.sharding.tables.t_order_item.actual-data-nodes=ds.t_order_item_$->{0..3}
spring.shardingsphere.sharding.tables.t_order_item.table-strategy.standard.sharding-column=order_id
spring.shardingsphere.sharding.tables.t_order_item.table-strategy.standard.precise-algorithm-class-name=com.x2era.sharding.sharding.table.algorithm.TablePreciseShardingAlgorithm
# 基本数据类型取模分表
spring.shardingsphere.sharding.tables.t_address.actual-data-nodes=ds.t_address_$->{0..2}
spring.shardingsphere.sharding.tables.t_address.table-strategy.inline.sharding-column=user_id
spring.shardingsphere.sharding.tables.t_address.table-strategy.inline.algorithm-expression=t_address_$->{user_id % 3}
# 联表查询建立绑定关系，处于在同一个分片节点
spring.shardingsphere.sharding.binding-tables=t_order,t_order_item
```

生产数据分片同步到分片表操作步骤

- 按照之前的方式，需要添加mybatis-plus动态数据源配置
  - 引入`dynamic-datasource-spring-boot-starter`多数据源依赖

    ```
    <dependency>
        <groupId>com.baomidou</groupId>
        <artifactId>dynamic-datasource-spring-boot-starter</artifactId>
        <version>3.1.1</version>
    </dependency>
    ```

  - 多数据源apollo方式

    ```
    spring.datasource.dynamic.primary = master
    spring.datasource.dynamic.datasource.master.type = com.zaxxer.hikari.HikariDataSource
    spring.datasource.dynamic.datasource.master.driver-class-name = com.mysql.cj.jdbc.Driver
    spring.datasource.dynamic.datasource.master.url = jdbc:mysql://10.242.231.7:3306/fmapp_mbr?useSSL=false&useUnicode=true&characterEncoding=utf-8&zeroDateTimeBehavior=convertToNull&transformedBitIsBoolean=true&tinyInt1isBit=false&allowMultiQueries=true&serverTimezone=GMT%2B8
    spring.datasource.dynamic.datasource.master.username = fmapp_dev
    spring.datasource.dynamic.datasource.master.password = KJss98scb$&*Qbrf45
    ```

  - sharding分表策略

```
spring.shardingsphere.datasource.names = member
spring.shardingsphere.datasource.member.type =
com.zaxxer.hikari.HikariDataSource
spring.shardingsphere.datasource.member.driver-class-name =
com.mysql.cj.jdbc.Driver
spring.shardingsphere.datasource.member.jdbc-url =
jdbc:mysql://10.242.231.7:3306/fmapp_mbr?
useSSL=false&useUnicode=true&characterEncoding=utf-
8&zeroDateTimeBehavior=convertToNull&transformedBitIsBoolean=true&tinyInt1is
Bit=false&allowMultiQueries=true&serverTimezone=GMT%2B8
spring.shardingsphere.datasource.member.username = fmapp_dev
spring.shardingsphere.datasource.member.password = KJss98scb$&*Qbrf45
spring.shardingsphere.props.sql.show = true
spring.shardingsphere.sharding.tables.mbr_mili_account.actualDataNodes =
member.mbr_mili_account_$->{0..7}
spring.shardingsphere.sharding.tables.mbr_mili_account.table-
strategy.standard.sharding-column = member_code
spring.shardingsphere.sharding.tables.mbr_mili_account.table-
strategy.standard.precise-algorithm-class-name =
com.x2era.xcloud.member.sharding.PreciseAlgorithm
```

- `Java`多数据源配置

```
@Configuration
@AutoConfigureBefore({DynamicDataSourceAutoConfiguration.class,
SpringBootConfiguration.class})
public class DataSourceConfig {

    /**
     * 分表数据源名称
     */
    private static final String SHARDING_DATA_SOURCE_NAME = "sharding";

    /**
     * 动态数据源配置项
     */
    @Autowired
    private DynamicDataSourceProperties properties;

    /**
     * shardingjdbc有四种数据源，需要根据业务注入不同的数据源
     *
     * <p>1. 未使用分片，脱敏的名称(默认): shardingDataSource;
     * <p>2. 主从数据源: masterSlaveDataSource;
     * <p>3. 脱敏数据源：encryptDataSource;
     * <p>4. 影子数据源：shadowDataSource
     */
    @Lazy
    @Resource(name = "shardingDataSource")
    AbstractDataSourceAdapter shardingDataSource;

    /**
     * 将动态数据源设置为首选的
     * 当spring存在多个数据源时，自动注入的是首选的对象
     * 设置为主要的数据源之后，就可以支持shardingjdbc原生的配置方式了
     *
```

```
      * @return 数据源
      */
    @Primary
    @Bean
    public DataSource dataSource(DynamicDataSourceProvider
dynamicDataSourceProvider) {

        DynamicRoutingDataSource dataSource = new
DynamicRoutingDataSource();
        dataSource.setPrimary(properties.getPrimary());
        dataSource.setStrict(properties.getStrict());
        dataSource.setStrategy(properties.getStrategy());
        dataSource.setProvider(dynamicDataSourceProvider);
        dataSource.setP6spy(properties.getP6spy());
        dataSource.setSeata(properties.getSeata());
        return dataSource;

    }

    @Bean
    public DynamicDataSourceProvider dynamicDataSourceProvider() {

        Map<String, DataSourceProperty> datasourceMap =
properties.getDatasource();
        return new AbstractDataSourceProvider() {
            @Override
            public Map<String, DataSource> loadDataSources() {
                Map<String, DataSource> dataSourceMap =
super.createDataSourceMap(datasourceMap);
                // 将 shardingjdbc 管理的数据源也交给动态数据源管理
                dataSourceMap.put(SHARDING_DATA_SOURCE_NAME,
shardingDataSource);
                return dataSourceMap;
            }
        };

    }
}
```

- Java代码层面需要使用`DS('sharding')`指定数据源

```
@DS("sharding")
@Override
public boolean editMiliBalance(final Integer indexType,
                               final String memberCode,
                               final Integer num,
                               final boolean editCurrent) {

    RLock rLock = this.openMiliAccountLock(memberCode);
    try {
        rLock.lock(ServiceConstants.MILI_ACCOUNT_LOCK_TIME,
TimeUnit.MINUTES);
        boolean result;
        if (indexType == DeductionEnums.IndexType.INCOME.getValue()) {
            result = this.addMili(memberCode,
                                   num);
        } else {
            if (editCurrent) {
                result = this.cutCurrentMili(memberCode,
                                             num);
            } else {
                result = this.cutMili(memberCode,
                                      num);
            }
        }
        return result;
    } catch (Exception e) {
        log.error("MiliAccountServiceImpl.editByMemberCode.用户米粒余额修改失
败，indexType：{}、memberCode：{}、num：{}",
                  indexType, memberCode, num, e);
        throw new CommonException(e.getMessage());
    } finally {
        rLock.unlock();
    }

}
```

注意：默认主数据源不是sharding分片的数据源

- 通过定时任务Job，分片+线程池方式处理生产数据
  - 非分片查询原表中的数据，通过xxl-job分片定时任务跑批

    ```
    ShardingUtil.ShardingVO shardingVo = ShardingUtil.getShardingVo();
    final int index = shardingVo.getIndex();
    final int total = shardingVo.getTotal();
    ```

    sql层

    ```
    SELECT count(id) FROM mbr_member
    WHERE delete_flag = 0 AND MOD(id,#{sTotal}) = #{sIndex}
    ```

    `sTotal` xxl-job注册节点总数量

    `sIndex` 当前节点所处的索引位置
  - 跑批插入到sharding分片表中，在`Service`层方法上田间`@DS('sharding')`注解指向sharding数据源插入分片表

注意：使用shardingsphere分表策略中间后，所有开发人员需要注意编写SQL脚本时，请查看shardingsphere是否支持；强烈要求，所有开发人员使用单表查询，禁止使用联表查询，如果需要联表查询请报备直属leader。