1. **多源数据库配置**

这里配置了一个动态数据源routingDataSource，包含两个数据源primaryDataSource和secondaryDataSource根据分库分表规则切换；以及另一个日志数据源。

1. **DataSourceConfig**

package com.scn7th.config;  
  
import com.scn7th.common.ShardingConstant;  
import com.scn7th.sharding.DynamicDataSourceRouter;  
import org.springframework.beans.factory.annotation.Qualifier;  
import org.springframework.boot.autoconfigure.jdbc.DataSourceBuilder;  
import org.springframework.boot.context.properties.ConfigurationProperties;  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
import org.springframework.context.annotation.Primary;  
import org.springframework.jdbc.datasource.lookup.AbstractRoutingDataSource;  
  
import javax.sql.DataSource;  
import java.util.HashMap;  
import java.util.Map;  
  
*/\*\*  
 \* Created by scn7th on 2017/4/16 0016.  
 \*/*@Configuration  
public class DataSourceConfig {  
 @Bean(name = "primaryDataSource")  
 @Qualifier("primaryDataSource")  
 @Primary  
 @ConfigurationProperties(prefix="spring.datasource.primary")  
 public DataSource primaryDataSource() {  
 return DataSourceBuilder.*create*().build();  
 }  
  
 @Bean(name = "secondaryDataSource")  
 @Qualifier("secondaryDataSource")  
 @ConfigurationProperties(prefix="spring.datasource.secondary")  
 public DataSource secondaryDataSource() {  
 return DataSourceBuilder.*create*().build();  
 }  
  
 @Bean(name = "routingDataSource")  
 public AbstractRoutingDataSource routingDataSource(@Qualifier("primaryDataSource") DataSource primaryDataSource,  
 @Qualifier("secondaryDataSource") DataSource secondaryDataSource) {  
 DynamicDataSourceRouter proxy = new DynamicDataSourceRouter();  
 Map<Object, Object> targetDataSources = new HashMap<>(2);  
 targetDataSources.put(ShardingConstant.*PRIMARY*, primaryDataSource);  
 targetDataSources.put(ShardingConstant.*SECONDARY*, secondaryDataSource);  
  
 proxy.setDefaultTargetDataSource(primaryDataSource);  
 proxy.setTargetDataSources(targetDataSources);  
 return proxy;  
 }  
  
  
 @Bean(name = "logDataSource")  
 @Qualifier("logDataSource")  
 @ConfigurationProperties(prefix="spring.datasource.log")  
 public DataSource logDataSource() {  
 return DataSourceBuilder.*create*().build();  
 }  
}

1. **EntityManagerConfig**

package com.scn7th.config;  
  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.boot.autoconfigure.orm.jpa.JpaProperties;  
import org.springframework.boot.orm.jpa.EntityManagerFactoryBuilder;  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
import org.springframework.context.annotation.Primary;  
import org.springframework.data.jpa.repository.config.EnableJpaRepositories;  
import org.springframework.orm.jpa.JpaTransactionManager;  
import org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean;  
import org.springframework.transaction.PlatformTransactionManager;  
import org.springframework.transaction.annotation.EnableTransactionManagement;  
  
import javax.annotation.Resource;  
import javax.persistence.EntityManager;  
import javax.sql.DataSource;  
import java.util.Map;  
  
*/\*\*  
 \* Created by scn7th on 2017/4/16 0016.  
 \*/*@Configuration  
@EnableTransactionManagement  
@EnableJpaRepositories(  
 entityManagerFactoryRef="entityManagerFactoryPrimary",  
 transactionManagerRef="transactionManagerPrimary",  
 basePackages= { "com.scn7th.trade.repository" }) //设置Repository所在位置  
public class EntityManagerConfig {  
  
 @Resource(name = "routingDataSource")  
 private DataSource routingDataSource;  
  
 @Primary  
 @Bean(name = "entityManagerPrimary")  
 public EntityManager entityManager(EntityManagerFactoryBuilder builder) {  
 return entityManagerFactoryPrimary(builder).getObject().createEntityManager();  
 }  
  
 @Primary  
 @Bean(name = "entityManagerFactoryPrimary")  
 public LocalContainerEntityManagerFactoryBean entityManagerFactoryPrimary (EntityManagerFactoryBuilder builder) {  
 return builder  
 .dataSource(routingDataSource)  
 .properties(getVendorProperties(routingDataSource))  
 .packages("com.scn7th.trade.model") //设置实体类所在位置  
 .persistenceUnit("primaryPersistenceUnit")  
 .build();  
 }  
  
 @Autowired  
 private JpaProperties jpaProperties;  
  
 private Map<String, String> getVendorProperties(DataSource dataSource) {  
 return jpaProperties.getHibernateProperties(dataSource);  
 }  
  
 @Primary  
 @Bean(name = "transactionManagerPrimary")  
 public PlatformTransactionManager transactionManagerPrimary(EntityManagerFactoryBuilder builder) {  
 return new JpaTransactionManager(entityManagerFactoryPrimary(builder).getObject());  
 }  
  
}

1. **LogEntityManagerConfig**

package com.scn7th.config;  
  
*/\*\*  
 \* Created by scn7th on 2017/4/16 0016.  
 \*/*import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.beans.factory.annotation.Qualifier;  
import org.springframework.boot.autoconfigure.orm.jpa.JpaProperties;  
import org.springframework.boot.orm.jpa.EntityManagerFactoryBuilder;  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
import org.springframework.data.jpa.repository.config.EnableJpaRepositories;  
import org.springframework.orm.jpa.JpaTransactionManager;  
import org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean;  
import org.springframework.transaction.PlatformTransactionManager;  
import org.springframework.transaction.annotation.EnableTransactionManagement;  
  
import javax.persistence.EntityManager;  
import javax.sql.DataSource;  
import java.util.Map;  
  
@Configuration  
@EnableTransactionManagement  
@EnableJpaRepositories(  
 entityManagerFactoryRef="entityManagerFactoryLog",  
 transactionManagerRef="transactionManagerLog",  
 basePackages= { "com.scn7th.log.repository" }) //设置Repository所在位置  
public class LogEntityManagerConfig {  
  
 @Autowired  
 @Qualifier(value = "logDataSource")  
 private DataSource logDataSource;  
  
 @Bean(name = "entityManagerLog")  
 public EntityManager entityManagerLog(EntityManagerFactoryBuilder builder) {  
 return entityManagerFactoryLog(builder).getObject().createEntityManager();  
 }  
  
 @Autowired  
 private JpaProperties jpaProperties;  
  
 private Map<String, String> getVendorProperties(DataSource dataSource) {  
 return jpaProperties.getHibernateProperties(dataSource);  
 }  
  
 @Bean(name = "transactionManagerLog")  
 PlatformTransactionManager transactionManagerLog(EntityManagerFactoryBuilder builder) {  
 return new JpaTransactionManager(entityManagerFactoryLog(builder).getObject());  
 }  
  
 @Bean(name = "entityManagerFactoryLog")  
 public LocalContainerEntityManagerFactoryBean entityManagerFactoryLog (EntityManagerFactoryBuilder builder) {  
 return builder  
 .dataSource(logDataSource)  
 .properties(getVendorProperties(logDataSource))  
 .packages("com.scn7th.log.model.root") //设置实体类所在位置  
 .persistenceUnit("LogPersistenceUnit")  
 .build();  
 }  
}

1. **分库分表标识——注解**
2. **Sharding——该方法使用分库分表**

package com.scn7th.sharding;  
  
import java.lang.annotation.\*;  
  
*/\*\*  
 \* Created by scn7th on 2020/5/6.  
 \*/*@Documented  
@Retention(RetentionPolicy.*RUNTIME*)  
@Target(value = {ElementType.*METHOD*, ElementType.*TYPE*})  
public @interface Sharding {  
}

1. **ShardingKey——入参指定分库分表的key**

package com.scn7th.sharding;  
  
import java.lang.annotation.\*;  
  
*/\*\*  
 \* Created by scn7th on 2020/5/6.  
 \*/*@Documented  
@Retention(RetentionPolicy.*RUNTIME*)  
@Target(value = {ElementType.*PARAMETER*})  
public @interface ShardingKey {  
 //入参名称  
 String value();  
}

1. **ShardingContextHolder在线程中存储分库分表key**

package com.scn7th.sharding;  
  
*/\*\*  
 \* Created by scn7th on 2020/5/6.  
 \*/*public class ShardingContextHolder {  
 private static ThreadLocal<String> *holder* = new ThreadLocal<>();  
  
 public static void setShardingKey(String shardingKey) {  
 *holder*.set(shardingKey);  
 }  
  
 public static String getShardingKey() {  
 return *holder*.get();  
 }  
  
 public static void remove() {  
 *holder*.remove();  
 }  
}

1. **ShardingAspect**

用于解析ShardingKey对应的参数，使用ShardingContextHolder将其写入线程的threadlocal中。

package com.scn7th.sharding;  
  
import org.aspectj.lang.ProceedingJoinPoint;  
import org.aspectj.lang.Signature;  
import org.aspectj.lang.annotation.Around;  
import org.aspectj.lang.annotation.Aspect;  
import org.aspectj.lang.reflect.MethodSignature;  
import org.springframework.core.annotation.Order;  
import org.springframework.stereotype.Component;  
import org.springframework.util.StringUtils;  
  
import java.lang.annotation.Annotation;  
  
*/\*\*  
 \* Created by scn7th on 2020/5/6.  
 \*/*@Aspect  
@Order(-1)  
@Component  
public class ShardingAspect {  
  
 @Around(value = "@annotation(com.scn7th.sharding.Sharding)")  
 public Object injectShardingKey(ProceedingJoinPoint joinPoint) throws Throwable {  
 String key = null;  
 try {  
 String keyName = getKeyName(joinPoint);  
 //没有指定key则直接执行  
 if(!StringUtils.*isEmpty*(keyName)) {  
 key = getKey(keyName, joinPoint);  
 if(!StringUtils.*isEmpty*(key)) {  
 ShardingContextHolder.*setShardingKey*(key);  
 }  
 }  
 return joinPoint.proceed();  
 } catch (Exception e) {  
 e.printStackTrace();  
 throw new RuntimeException(e);  
 } finally {  
 if(!StringUtils.*isEmpty*(key)) {  
 ShardingContextHolder.*remove*();  
 }  
 }  
 }  
  
 private String getKeyName(ProceedingJoinPoint joinPoint) {  
 Signature signature = joinPoint.getSignature();  
 Annotation[][] parameterAnnotations= ((MethodSignature) signature).getMethod().getParameterAnnotations();  
 String keyName = null;  
 for (Annotation[] parameterAnnotation: parameterAnnotations) {  
 for(Annotation annotation : parameterAnnotation) {  
 if(annotation instanceof ShardingKey) {  
 ShardingKey shardingKey = (ShardingKey) annotation;  
 keyName = shardingKey.value();  
 break;  
 }  
 }  
 if(!StringUtils.*isEmpty*(keyName)) break;  
 }  
 return keyName;  
 }  
  
 private String getKey(String keyName, ProceedingJoinPoint joinPoint) {  
 Signature signature = joinPoint.getSignature();  
 MethodSignature methodSignature = (MethodSignature)signature;  
 String[] parameterNames = methodSignature.getParameterNames();  
 Object[] args = joinPoint.getArgs();  
 String key = null;  
 for(int i = 0; i < parameterNames.length; i++) {  
 if(keyName.equals(parameterNames[i])) {  
 key = (String) args[i];  
 break;  
 }  
 }  
 return key;  
 }  
}

1. **分库分表动态路由**
2. **DynamicDataSourceRouter动态数据源的路由**

package com.scn7th.sharding;  
  
import com.scn7th.common.ShardingConstant;  
import org.springframework.jdbc.datasource.lookup.AbstractRoutingDataSource;  
  
*/\*\*  
 \* 分库规则  
 \* Created by scn7th on 2020/5/6.  
 \*/*public class DynamicDataSourceRouter extends AbstractRoutingDataSource{  
 @Override  
 protected Object determineCurrentLookupKey() {  
 String key = ShardingContextHolder.*getShardingKey*();  
 if(key == null || key.startsWith("TX")) return ShardingConstant.*PRIMARY*;  
 return ShardingConstant.*SECONDARY*;  
 }  
}

1. DynamicTableInterceptor表的路由

继承hibernate EmptyInterceptor就可以修改当前sql。

package com.scn7th.sharding;  
  
import org.hibernate.EmptyInterceptor;  
  
*/\*\*  
 \* 分表规则  
 \* Created by scn7th on 2020/5/6.  
 \*/*public class DynamicTableInterceptor extends EmptyInterceptor{  
 @Override  
 public String onPrepareStatement(String sql) {  
 //*todo 分表* String key = ShardingContextHolder.*getShardingKey*();  
 return sql;  
 }  
  
}

1. **使用**
2. TradeService使用分库分表注解

package com.scn7th.trade;  
  
import com.scn7th.sharding.Sharding;  
import com.scn7th.sharding.ShardingKey;  
import com.scn7th.trade.model.root.TradeStatisticsValue;  
import com.scn7th.trade.repository.TradeStatisticsValueRepository;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.stereotype.Service;  
import org.springframework.transaction.annotation.Transactional;  
  
*/\*\*  
 \* Created by scn7th on 2020/5/6.  
 \*/*@Service  
public class TradeService {  
 @Autowired  
 private TradeStatisticsValueRepository repository;  
  
 @Sharding  
 @Transactional  
 public void save(@ShardingKey("cid") String cid, boolean kaka) {  
// ShardingContextHolder.setShardingKey(cid);  
 TradeStatisticsValue result = new TradeStatisticsValue();  
 result.setCid(cid);  
 result.setAdminId("A245");  
 repository.save(result);  
 }  
}

1. **分库分表后的分页排序查询思考**

如果我们跨表分页查询，并使用Order by，显然会随着offset变大查询越来越慢，因为分页order by要去我们首先对所有表进行排序，然后从所有表(数量为n)排序结果中取出前k=offset + pageSize个数据，执行归并排序，找出前k个数据。显然数据量会越来越大。

一个可行的办法是根据各个表建立一个统一视图，例如ES。

实际使用中，我们订单使用了sharding-jdbc来做自动路由。

1. **有生命周期的业务聚合根分库实践**
2. **按照时间路由**

由于订单之类的单据有明确的生命周期，我们可以将结束的单据迁入到历史库中，查询统一走es聚合视图。这样不需要像会员那样根据店铺编号进行分库分表，大大简化了流程。

订单分库分为：

* 主库（记录近3个月内的数据）
* 归档库（记录3个月外的数据）
* 日志库（记录业务日志，做补偿使用）

1. **迁移到历史库**

当单据结束，最关键的问题是如何将其准确、不丢失、快速地迁入到历史库，然后执行清理。



针对数据迁移三种方案，我们给出了测试如下：

1. **jpa迁移**

对于26w订单，从当前库分页读到本地内存，然后再写入到历史库中，总耗时1200s。；

1. **jdbc-batch迁移**

通过jdbc流fetch，一次查询1w条数据（游标，不是分页），读取速度非常快，对于84个表字段的订单，读取速度2.5w/s。

通过jdbc batch写，每个batch有1w数据，数据库自动提交。

26w订单迁移耗时98s，速度为2600/s，提升12倍。

package com.scn7th.batch;  
  
import com.mysql.cj.api.jdbc.Statement;  
import lombok.extern.slf4j.Slf4j;  
  
import java.sql.\*;  
  
*/\*\*  
 \* Created by scn7th on 2020/11/10.  
 \*/*@Slf4j  
public class FetchThenInsertBatch {  
 public static void main(String[] args) {  
 try {  
 PreparedStatement readPs = *getReadPs*();  
 PreparedStatement writePs = *getWritePs*();  
 ResultSet rs = readPs.executeQuery();  
 *write*(writePs, rs);  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
 }  
  
 private static void write(PreparedStatement ps, ResultSet rs) {  
 try {  
 int batch = 10000;  
 int recordNum = 0;  
 long startTime = System.*currentTimeMillis*();  
 long lastTime = System.*currentTimeMillis*();  
 while (rs.next()) {  
 try {  
 recordNum++;  
 if (recordNum % batch == 0) {  
 *log*.info(rs.getString(1));  
 *log*.info("Current record " + recordNum + " , speed " + (batch \* 1000 / (System.*currentTimeMillis*() - lastTime)) + " record/s");  
 lastTime = System.*currentTimeMillis*();  
 int[] updateCounts = ps.executeBatch();  
 }  
 ResultSetMetaData rsmd = rs.getMetaData();  
 ps.setObject(1, "T" + rs.getObject(1));  
 for (int i = 2; i <= rsmd.getColumnCount(); i++) {  
 ps.setObject(i, rs.getObject(i));  
 }  
 ps.addBatch();  
 // Execute the batch  
 }catch (Exception e) {  
 e.printStackTrace();  
 }  
 }  
 *log*.info("Total record " + recordNum + " , spends " + (System.*currentTimeMillis*() - startTime));  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
 }  
  
  
 private static PreparedStatement getReadPs() {  
 try {  
 Class.*forName*("com.mysql.jdbc.Driver");  
 Connection readConn = DriverManager.*getConnection*(  
 "jdbc:mysql://172.19.28.251:3306/d2ctccashmysql?useUnicode=true&characterEncoding=UTF-8", "d2ctccashmysql", "HY67TG4E");  
 PreparedStatement ps = readConn.prepareStatement( "select \* from cashier\_trade where tid > 'TCC200910000001' and tid < 'TCC200913000001' ", ResultSet.*TYPE\_FORWARD\_ONLY*, ResultSet.*CONCUR\_READ\_ONLY*);  
 ps.setFetchSize(10000);  
 ps.setFetchDirection(ResultSet.*FETCH\_FORWARD*);  
 ((Statement)ps).enableStreamingResults();  
 return ps;  
 } catch (Exception e) {  
 e.printStackTrace();  
 return null;  
 }  
 }  
  
 private static PreparedStatement getWritePs() {  
 try {  
 Connection writeConn = DriverManager.*getConnection*(  
 "jdbc:mysql://172.19.29.151:3306/d2ctccashmysql?useUnicode=tr ue&characterEncoding=UTF-8&rewriteBatchedStatements=true", "d2ctccashmysql", "HY67TG4E");  
 // Create a prepared statement  
 String sql = "REPLACE INTO cashier\_trade VALUES(";  
 int size = 84;  
 for(int i = 0; i < size; i++) {  
 sql += "?";  
 if(i != size - 1) sql += ",";  
 }  
 sql += ")";  
 PreparedStatement insertPs = writeConn.prepareStatement(sql);  
 return insertPs;  
 } catch (Exception e) {  
 e.printStackTrace();  
 return null;  
 }  
 }  
}

1. **kafka监听binlog**

架构团队无法保证canal和kafka不丢失Binlog，所以没有执行。

1. **本地镜像**

我们在历史库所在实例建立主库从机，通过监听binlog实现同步。然后执行：

replace INTO d2ctccashmysql.cashier\_trade select \* from d2ctccashmysql\_bak.cashier\_trade where tid > 'TCC200910000001' and tid < 'TCC200913000001'

26w数据耗时8592ms，速度提升11倍。

*/\*\*  
 \* 本地跨库拷贝  
 \*/*public static void localCopy() {  
 try {  
 Connection writeConn = DriverManager.*getConnection*("jdbc:mysql://172.19.28.251:3306/d2ctccashmysql?useUnicode=true&characterEncoding=UTF-8", "d2ctccashmysql", "HY67TG4E");  
 // Create a prepared statement  
 String sql = "replace INTO d2ctccashmysql.cashier\_trade select \* from d2ctccashmysql\_bak.cashier\_trade where tid > 'TCC200910000001' and tid < 'TCC200913000001'";  
 long start = System.*currentTimeMillis*();  
 PreparedStatement insertPs = writeConn.prepareStatement(sql);  
 insertPs.execute();  
 *log*.info("耗时：{}ms", System.*currentTimeMillis*() - start);  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
}

1. **最终方案**

利用本地数据拷贝，如果checksum失败，则利用远程全量重新拷贝，如果此时checksum再失败，则等待人工处理，sharding游标不移动。



参考

1. 思路<https://hacpai.com/article/1497318811539>
2. Jpa拦截器<https://blog.csdn.net/weixin_34270865/article/details/92079428>
3. 动态数据源<https://blog.csdn.net/sunshine920103/article/details/50363315>
4. <https://www.jianshu.com/p/97ed280204d9>
5. Springboot+jpa+动态数据源<https://www.dazhuanlan.com/2019/12/13/5df32795f2766/>