链滴

基于 spring 的 aop 实现多数据源动态切换

作者：aqjun
原文链接：https://ld246.com/article/1488101038281
来源网站：链滴

一、多数据源动态切换原理

项目中我们经常会遇到多数据源的问题，尤其是数据同步或定时任务等项目更是如此；又例如：读写离数据库配置的系统。

1、多数据源设置：

1）静态数据源切换：

一般情况下，我们可以配置多个数据源，然后为每个数据源写一套对应的sessionFactory和dao层代码以hibernate为例，mybatis同理)，——我们称之为_静态数据源配置_。

2）动态数据源切换：

可看出在Dao层代码中写死了两个SessionFactory，这样日后如果再多一个数据源，还要改代码添加个SessionFactory，显然这并不符合开闭原则。比较好的做法是，配置多个数据源，只对应一套sessinFactory，数据源之间可以动态切换。

2、动态数据源切换时，如何保证数据库的事务：

目前事务最灵活的方式，是使用spring的声明式事务，本质是利用了spring的aop，在执行数据库操前后，加上事务处理。

spring的事务管理，是基于数据源的，所以如果要实现动态数据源切换，而且在同一个数据源中保证务是起作用的话，就需要注意二者的顺序问题，即：在事物起作用之前就要把数据源切换回来。

举一个例子：web开发常见是三层结构：controller、service、dao。一般事务都会在service层添加如果使用spring的声明式事物管理，在调用service层代码之前，spring会通过aop的方式动态添加事控制代码，所以如果要想保证事物是有效的，那么就必须在spring添加事务之前把数据源动态切换过，也就是动态切换数据源的aop要至少在service上添加，而且要在spring声明式事物aop之前添加.根上面分析:

- 最简单的方式是把动态切换数据源的aop加到controller层，这样在controller层里面就可以确定下数据源了。不过，这样有一个缺点就是，每一个controller绑定了一个数据源，不灵活。对于这种：个请求，需要使用两个以上数据源中的数据完成的业务时，就无法实现了。
- 针对上面的这种问题，可以考虑把动态切换数据源的aop放到service层，但要注意一定要在事务ao之前来完成。这样，对于一个需要多个数据源数据的请求，我们只需要在controller里面注入多个servie实现即可。但这种做法的问题在于，controller层里面会涉及到一些不必要的业务代码，例如：合并个数据源中的list...
- 此外，针对上面的问题，还可以再考虑一种方案，就是把事务控制到dao层，然后在service层里面态切换数据源。

二、实例1：

本例子中，对不同数据源分包（package）管理，同一包下的代码使用了同一数据源。

1、写一个DynamicDataSource类继承**AbstractRoutingDataSource**，并实现**determineCurrentookupKey方法**：

```
import org.springframework.jdbc.datasource.lookup.AbstractRoutingDataSource;
public class DynamicDataSource extends AbstractRoutingDataSource {
    @Override
    protected Object determineCurrentLookupKey() {
```

```
        return CustomerContextHolder.getCustomerType();
    }
}
```

2、利用**ThreadLocal**解决线程安全问题：

```
public class CustomerContextHolder {
    public static final String DATA_SOURCE_A = "dataSource";
    public static final String DATA_SOURCE_B = "dataSource2";
    private static final ThreadLocal<String> contextHolder = new ThreadLocal<String>();
    public static void setCustomerType(String customerType) {
        contextHolder.set(customerType);
    }
    public static String getCustomerType() {
        return contextHolder.get();
    }
    public static void clearCustomerType() {
        contextHolder.remove();
    }
}
```

3、定义一个**数据源切面类**，通过aop来控制数据源的切换：

```
import org.aspectj.lang.JoinPoint;

public class DataSourceInterceptor {

    public void setdataSourceMysql(JoinPoint jp) {
        DatabaseContextHolder.setCustomerType("dataSourceMySql");
    }

    public void setdataSourceOracle(JoinPoint jp) {
        DatabaseContextHolder.setCustomerType("dataSourceOracle");
    }
}
```

4、在spring的application.xml中配置多个dataSource：

```
<!-- 数据源1 -->
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">
        <property name="driverClassName" value="net.sourceforge.jtds.jdbc.Driver"></prope
ty>
        <property name="url" value="jdbc:jtds:sqlserver://10.82.81.51:1433;databaseName=sta
dards"></property>
        <property name="username" value="youguess"></property>
        <property name="password" value="youguess"></property>
</bean>
<!-- 数据源2 -->
<bean id="dataSource2" class="org.apache.commons.dbcp.BasicDataSource">
        <property name="driverClassName" value="net.sourceforge.jtds.jdbc.Driver"></prope
ty>
        <property name="url" value="jdbc:jtds:sqlserver://10.82.81.52:1433;databaseName=sta
dards"></property>
        <property name="username" value="youguess"></property>
        <property name="password" value="youguess"></property>
```

```
</bean>

<bean id="dataSource" class="com.core.DynamicDataSource">
    <property name="targetDataSources">
        <map key-type="java.lang.String">
            <entry key="dataSourceMySql" value-ref="dataSourceMySql" />
            <entry key="dataSourceOracle" value-ref="dataSourceOracle" />
        </map>
    </property>
    <property name="defaultTargetDataSource" ref="dataSourceMySql" />
</bean>

<!-- 动态数据源切换aop 先与事务的aop  -->
<bean id="dataSourceInterceptor" class="com.core.DataSourceInterceptor" />
<aop:config>
    <aop:aspect id="dataSourceAspect" ref="dataSourceInterceptor">
        <aop:pointcut id="dsMysql" expression="execution(* com.service.mysql..*.*(..))" />
        <aop:pointcut id="dsOracle" expression="execution(* com.service.oracle..*.*(..))" />
        <aop:before method="setdataSourceMysql" pointcut-ref="dsMysql"/>
        <aop:before method="setdataSourceOracle" pointcut-ref="dsOracle"/>
    </aop:aspect>
</aop:config>

<!-- 事物aop -->
。。。
```

三、实例2：

该例子，实现了在业务逻辑层控制了mysql的读写分离。同样，使用了spring的aop来动态切换读和的数据源。和上个例子不同之处在于：不同数据源没有按照包分类管理，而是使用了自定义注解。

1、首先配置mysql 的主从复制：

详情见，这里。

2、自定义注解：

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Target;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
/**
 * RUNTIME
 * 编译器将把注释记录在类文件中，在运行时 VM 将保留注释，因此可以反射性地读取。
 * @author yangGuang
 *
 */
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface DataSource {
    String value();
}
```

3、基于spring的aop实现多数据源（读和写两个数据源）：

1）写一个ChooseDataSource： 类继承_AbstractRoutingDataSource_，并实现_determineCurrenLookupKey方法_：

```
import org.springframework.jdbc.datasource.lookup.AbstractRoutingDataSource;

public class ChooseDataSource extends AbstractRoutingDataSource {

    @Override
    protected Object determineCurrentLookupKey() {
        return HandleDataSource.getDataSource();
    }

}
```

2）利用ThreadLocal解决线程安全问题：

```
public class HandleDataSource {
    public static final ThreadLocal<String> holder = new ThreadLocal<String>();
    public static void putDataSource(String datasource) {
        holder.set(datasource);
    }

    public static String getDataSource() {
        return holder.get();
    }
}
```

3）定义一个数据源切面类，通过aop访问，获取方法上的自定义注解，然后根据注解内容尽情判断动态设置数据源：

```
import java.lang.reflect.Method;
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;
import org.aspectj.lang.reflect.MethodSignature;
import org.springframework.stereotype.Component;
//@Aspect
//@Component
public class DataSourceAspect {
    //@Pointcut("execution(* com.apc.cms.service.*.*(..))")
    public void pointCut(){};

  //  @Before(value = "pointCut()")
     public void before(JoinPoint point)
        {
            Object target = point.getTarget();
            System.out.println(target.toString());
            String method = point.getSignature().getName();
            System.out.println(method);
            Class<?>[] classz = target.getClass().getInterfaces();
            Class<?>[] parameterTypes = ((MethodSignature) point.getSignature())
                .getMethod().getParameterTypes();
            try {
                Method m = classz[0].getMethod(method, parameterTypes);
```

```
                System.out.println(m.getName());
                if (m != null && m.isAnnotationPresent(DataSource.class)) {
                    DataSource data = m.getAnnotation(DataSource.class);
                    HandleDataSource.putDataSource(data.value());
                }

            } catch (Exception e) {
                e.printStackTrace();
            }
        }
}
```

4）配置applicationContext.xml：

```
<!-- 主库数据源 -->
 <bean id="writeDataSource" class="com.jolbox.bonecp.BoneCPDataSource"  destroy-metho
="close">
    <property name="driverClass" value="com.mysql.jdbc.Driver"/>
    <property name="jdbcUrl" value="jdbc:mysql://172.22.14.6:3306/cpp?autoReconnect=true
/>
    <property name="username" value="root"/>
    <property name="password" value="root"/>
    <property name="partitionCount" value="4"/>
    <property name="releaseHelperThreads" value="3"/>
    <property name="acquireIncrement" value="2"/>
    <property name="maxConnectionsPerPartition" value="40"/>
    <property name="minConnectionsPerPartition" value="20"/>
    <property name="idleMaxAgeInSeconds" value="60"/>
    <property name="idleConnectionTestPeriodInSeconds" value="60"/>
    <property name="poolAvailabilityThreshold" value="5"/>
</bean>

<!-- 从库数据源 -->
<bean id="readDataSource" class="com.jolbox.bonecp.BoneCPDataSource"  destroy-metho
="close">
    <property name="driverClass" value="com.mysql.jdbc.Driver"/>
    <property name="jdbcUrl" value="jdbc:mysql://172.22.14.7:3306/cpp?autoReconnect=true
/>
    <property name="username" value="root"/>
    <property name="password" value="root"/>
    <property name="partitionCount" value="4"/>
    <property name="releaseHelperThreads" value="3"/>
    <property name="acquireIncrement" value="2"/>
    <property name="maxConnectionsPerPartition" value="40"/>
    <property name="minConnectionsPerPartition" value="20"/>
    <property name="idleMaxAgeInSeconds" value="60"/>
    <property name="idleConnectionTestPeriodInSeconds" value="60"/>
    <property name="poolAvailabilityThreshold" value="5"/>
</bean>

<!-- transaction manager, 事务管理 -->
<bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTra
sactionManager">
    <property name="dataSource" ref="dataSource" />
```

```
</bean>


<!-- 注解自动载入 -->
<context:annotation-config />

<!--enale component scanning (beware that this does not enable mapper scanning!)-->
<context:component-scan base-package="com.apc.cms.persistence.rdbms" />
<context:component-scan base-package="com.apc.cms.service">
 <context:include-filter type="annotation"
   expression="org.springframework.stereotype.Component" />
</context:component-scan>

<context:component-scan base-package="com.apc.cms.auth" />

<!-- enable transaction demarcation with annotations -->
<tx:annotation-driven />


<!-- define the SqlSessionFactory -->
<bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
   <property name="dataSource" ref="dataSource" />
   <property name="typeAliasesPackage" value="com.apc.cms.model.domain" />
</bean>

<!-- scan for mappers and let them be autowired -->
<bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
   <property name="basePackage" value="com.apc.cms.persistence" />
   <property name="sqlSessionFactory" ref="sqlSessionFactory" />
</bean>

<bean id="dataSource" class="com.apc.cms.utils.ChooseDataSource">
   <property name="targetDataSources">
        <map key-type="java.lang.String">
           <!-- write -->
          <entry key="write" value-ref="writeDataSource"/>
          <!-- read -->
          <entry key="read" value-ref="readDataSource"/>
        </map>

   </property>
   <property name="defaultTargetDataSource" ref="writeDataSource"/>
</bean>

<!-- 激活自动代理功能 -->
<aop:aspectj-autoproxy proxy-target-class="true"/>

<!-- 配置数据库注解aop -->
<bean id="dataSourceAspect" class="com.apc.cms.utils.DataSourceAspect" />
<aop:config>
   <aop:aspect id="c" ref="dataSourceAspect">
      <aop:pointcut id="tx" expression="execution(* com.apc.cms.service..*.*(..))"/>
      <aop:before pointcut-ref="tx" method="before"/>
   </aop:aspect>
```

```
</aop:config>
<!-- 配置数据库注解aop -->
```

4、测试：

```
@DataSource("write")
public void update(User user) {
    userMapper.update(user);
}

@DataSource("read")
public Document getDocById(long id) {
    return documentMapper.getById(id);
}
```

● 测试写操作：可以通过应用修改数据，修改主库数据，发现从库的数据被同步更新了，所以定义的write操作都是走的写库

● 测试读操作： 后台修改从库数据，查看主库的数据没有被修改，在应用页面中刷新，发现读的是从的数据，说明读写分离ok。