<http://blog.csdn.net/yizhenn/article/details/53965552>

<http://blog.csdn.net/llhwin2010/article/details/11695781>

[**Spring代码中动态切换数据源**](http://blog.csdn.net/yizhenn/article/details/53965552)

2017-01-02 14:59 5422人阅读 [评论](http://blog.csdn.net/yizhenn/article/details/53965552#comments)(13) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/yizhenn/article/details/53965552#report)

 分类：

Spring（4）  数据库（12） 

版权声明：本文为博主原创文章，未经博主允许不得转载。

最近项目中遇到一个场景，需要能够在一个方法中操作多个具有相同表结构数据库(你可以理解为一个表中的数据被水平拆分到多个库中，查询时需要遍历这多个库)。经过笔者几天的研究，最终解决了问题，并且写了一个demo共享到[我的github](https://github.com/ZhenShiErGe/test/tree/dev_add_multi_datasource)。

关注笔者博客的小伙伴一定知道之前的这篇文章[点击打开链接](http://blog.csdn.net/yizhenn/article/details/52384607)，这篇博客中的解决方案仅仅适用读写分离的场景。就是说，当你在开发的时候已经确定了使用写库一读库的形式。笔者今天要写的这篇文章具有普适性，适合所有需要在Spring工程中动态切换数据源的场景，而且本文中的解决方案对工程的代码基本没有侵入性。下面就来说下该方案的实现原理：

在Spring-Mybatis中，有这样一个类AbstractRoutingDataSource，根据名字可以猜到，这是一个框架提供的用于动态选择数据源的类。这个类有两个重要的参数，分别叫

defaultTargetDataSource和targetDataSources。一般的工程都是一个数据源，所以不太接触到这个类。在作者之前的博客[自动切换多个数据源](http://blog.csdn.net/yizhenn/article/details/52384607)中，可以看到这个类的xml配置如下：

**[html]** [view plain](http://blog.csdn.net/yizhenn/article/details/53965552#) [copy](http://blog.csdn.net/yizhenn/article/details/53965552#)

1. **<bean** id="myoneDataSource" class="org.apache.commons.dbcp2.BasicDataSource" destroy-method="close"**>**

2. **<property** name="driverClassName" value="${jdbc.myone.driver}"**/>**

3. **<property** name="url" value="${jdbc.myone.url}"**/>**

4. **<property** name="username" value="${jdbc.myone.username}"**/>**

5. **<property** name="password" value="${jdbc.myone.password}"**/>**

6. **</bean>**

7. **<bean** id="mytwoDataSource" class="org.apache.commons.dbcp2.BasicDataSource" destroy-method="close"**>**

8. **<property** name="driverClassName" value="${jdbc.mytwo.driver}"**/>**

9. **<property** name="url" value="${jdbc.mytwo.url}"**/>**

10. **<property** name="username" value="${jdbc.mytwo.username}"**/>**

11. **<property** name="password" value="${jdbc.mytwo.password}"**/>**

12. **</bean>**

13.

14. **<bean** id="multipleDataSource" class="dal.datasourceswitch.MultipleDataSource"**>**

15. **<property** name="defaultTargetDataSource" ref="myoneDataSource"**/>** <!--默认主库-->

16. **<property** name="targetDataSources"**>**

17. **<map>**

18. **<entry** key="myone" value-ref="myoneDataSource"**/>** <!--辅助aop完成自动数据库切换-->

19. **<entry** key="mytwo" value-ref="mytwoDataSource"**/>**

20. **</map>**

21. **</property>**

22. **</bean>**

上面的配置文件对这两个参数的描述已经很清楚了，但这是多个数据源已经确定的场景。我们这篇博客中的场景是多个数据源的信息存在于数据库中，可能数据库中的数据源信息会动态的增加或者减少。这样的话，就不能像上面这样配置了。那怎么办呢？

我们仅仅需要设定默认的数据源，即defaultDataSource参数，至于targetDataSources参数我们需要在代码中动态的设定。来看下具体的xml配置：

**[html]** [view plain](http://blog.csdn.net/yizhenn/article/details/53965552#) [copy](http://blog.csdn.net/yizhenn/article/details/53965552#)

1. **<bean** id="defaultDataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close"

2. p:driverClassName="${db\_driver}"

3. p:url="${db\_url}"

4. p:username="${db\_user}"

5. p:password="${db\_pass}"

6. p:validationQuery="select 1"

7. p:testOnBorrow="true"**/>**

8.

9. <!--动态数据源相关-->

10. **<bean** id="dynamicDataSource" class="org.xyz.test.service.datasourceswitch.impl.DynamicDataSource"**>**

11. **<property** name="targetDataSources"**>**

12. **<map** key-type="java.lang.String"**>**

13. **<entry** key="defaultDataSource" value-ref="defaultDataSource"**/>**

14. **</map>**

15. **</property>**

16. **<property** name="defaultTargetDataSource" ref="defaultDataSource"**/>**

17. **</bean>**

从上面的配置文件中可以看到，我们仅仅配置了默认的数据源defaultDataSource。至于其他的数据源targetDataSources，我们没有配置，需要在代码中动态的创建。关于配置就讲清楚啦！但我们注意到，支持动态数据源的不应该是AbstractRoutingDataSource类吗？怎么上面的配置中是DynamicDataSource类。没错，这个是我们自定义的继承自AbstractRoutingDataSource类的类，也只最重要的类，来看下：（理解这个类，你需要熟练掌握JAVA反射，以及ThreadLocal变量，和Spring的注入机制。别退缩，大家都是这样一步步学过来的！）（下面仅仅是看下全貌，代码的下面会有详细的说明）

**[java]** [view plain](http://blog.csdn.net/yizhenn/article/details/53965552#) [copy](http://blog.csdn.net/yizhenn/article/details/53965552#)

1. **final** **class** DynamicDataSource **extends** AbstractRoutingDataSource **implements** ApplicationContextAware{

2.

3. **private** **static** **final** String DATA\_SOURCES\_NAME = "targetDataSources";

4.

5. **private** ApplicationContext applicationContext;

6.

7. @Override

8. **protected** Object determineCurrentLookupKey() {

9. DataSourceBeanBuilder dataSourceBeanBuilder = DataSourceHolder.getDataSource();

10. System.out.println("----determineCurrentLookupKey---"+dataSourceBeanBuilder);

11. **if** (dataSourceBeanBuilder == **null**) {

12. **return** **null**;

13. }

14. DataSourceBean dataSourceBean = **new** DataSourceBean(dataSourceBeanBuilder);

15. //查看当前容器中是否存在

16. **try** {

17. Map<Object,Object> map=getTargetDataSources();

18. **synchronized** (**this**) {

19. **if** (!map.keySet().contains(dataSourceBean.getBeanName())) {

20. map.put(dataSourceBean.getBeanName(), createDataSource(dataSourceBean));

21. **super**.afterPropertiesSet();//通知spring有bean更新

22. }

23. }

24. **return** dataSourceBean.getBeanName();

25. } **catch** (NoSuchFieldException | IllegalAccessException e) {

26. **throw** **new** SystemException(ErrorEnum.MULTI\_DATASOURCE\_SWITCH\_EXCEPTION);

27. }

28. }

29.

30.

31. **private** Object createDataSource(DataSourceBean dataSourceBean) **throws** IllegalAccessException {

32. //在spring容器中创建并且声明bean

33. ConfigurableApplicationContext context = (ConfigurableApplicationContext) applicationContext;

34. DefaultListableBeanFactory beanFactory = (DefaultListableBeanFactory) context.getBeanFactory();

35. BeanDefinitionBuilder beanDefinitionBuilder = BeanDefinitionBuilder.genericBeanDefinition(BasicDataSource.**class**);

36. //将dataSourceBean中的属性值赋给目标bean

37. Map<String, Object> properties = getPropertyKeyValues(DataSourceBean.**class**, dataSourceBean);

38. **for** (Map.Entry<String, Object> entry : properties.entrySet()) {

39. beanDefinitionBuilder.addPropertyValue((String) entry.getKey(), entry.getValue());

40. }

41. beanFactory.registerBeanDefinition(dataSourceBean.getBeanName(), beanDefinitionBuilder.getBeanDefinition());

42. **return** applicationContext.getBean(dataSourceBean.getBeanName());

43. }

44.

45. **private** Map<Object, Object> getTargetDataSources() **throws** NoSuchFieldException, IllegalAccessException {

46. Field field = AbstractRoutingDataSource.**class**.getDeclaredField(DATA\_SOURCES\_NAME);

47. field.setAccessible(**true**);

48. **return** (Map<Object, Object>) field.get(**this**);

49. }

50.

51.

52. **private** <T> Map<String, Object> getPropertyKeyValues(Class<T> clazz, Object object) **throws** IllegalAccessException {

53. Field[] fields = clazz.getDeclaredFields();

54. Map<String, Object> result = **new** HashMap<>();

55. **for** (Field field : fields) {

56. field.setAccessible(**true**);

57. result.put(field.getName(), field.get(object));

58. }

59. result.remove("beanName");

60. **return** result;

61. }

62.

63. @Override

64. **public** **void** setApplicationContext(ApplicationContext applicationContext) **throws** BeansException {

65. **this**.applicationContext=applicationContext;

66. }

67. }

首先来看覆盖方法determineCurrentLookupKey()，框架在每次调用数据源时会先调用这个方法，以便知道使用哪个数据源。在本文的场景中，数据源是由程序员在即将切换数据源之前，将要使用的那个数据源的名称放到当前线程的ThreadLocal中，这样在determineCurrentLookupKey()方法中就可以从ThreadLocal中拿到当前请求钥匙用的数据源，从而进行初始化数据源并返回该数据源的操作。在ThreadLocal变量中，我们保存了一个DataSourceBuilder，这是一个建造者模式，不是本文的关键。我们在后面说。读者直接把他理解为是一个数据源的描述就好。因此，determineCurrentLookupKey()方法的流程就是：先从ThreadLocal中拿出要使用的数据源信息，然后看当前的targetDataSources中是否有了这个数据源。如果有直接返回。如果没有，创建一个这样的数据源，放到targetDataSources中然后返回。这个过程需要加锁，为何？这是典型的判断后插入场景，在多线程中会有线程安全问题，所以要加锁！至嘱！

由于targetDataSources是父类AbstractRoutingDataSource中的一个私有域，因此想要获得他的实例只能通过Java反射机制。这也是下面的方法存在的意义！

**[java]** [view plain](http://blog.csdn.net/yizhenn/article/details/53965552#) [copy](http://blog.csdn.net/yizhenn/article/details/53965552#)

1. **private** Map<Object, Object> getTargetDataSources() **throws** NoSuchFieldException, IllegalAccessException {

2. Field field = AbstractRoutingDataSource.**class**.getDeclaredField(DATA\_SOURCES\_NAME);

3. field.setAccessible(**true**);

4. **return** (Map<Object, Object>) field.get(**this**);

5. }

然后，我们来看具体是怎么创建数据源的。

**[java]** [view plain](http://blog.csdn.net/yizhenn/article/details/53965552#) [copy](http://blog.csdn.net/yizhenn/article/details/53965552#)

1. **private** Object createDataSource(DataSourceBean dataSourceBean) **throws** IllegalAccessException {

2. //在spring容器中创建并且声明bean

3. ConfigurableApplicationContext context = (ConfigurableApplicationContext) applicationContext;

4. DefaultListableBeanFactory beanFactory = (DefaultListableBeanFactory) context.getBeanFactory();

5. BeanDefinitionBuilder beanDefinitionBuilder = BeanDefinitionBuilder.genericBeanDefinition(BasicDataSource.**class**);

6. //将dataSourceBean中的属性值赋给目标bean

7. Map<String, Object> properties = getPropertyKeyValues(DataSourceBean.**class**, dataSourceBean);

8. **for** (Map.Entry<String, Object> entry : properties.entrySet()) {

9. beanDefinitionBuilder.addPropertyValue((String) entry.getKey(), entry.getValue());

10. }

11. beanFactory.registerBeanDefinition(dataSourceBean.getBeanName(), beanDefinitionBuilder.getBeanDefinition());

12. **return** applicationContext.getBean(dataSourceBean.getBeanName());

13. }

大家知道，Spring最主要的功能是作为bean容器，即他负责bean生命周期的管理。因此，我们自定义的datasource也不能“逍遥法外”，必须交给Spring容器来管理。这也正是DynamicDataSource类需要实现ApplicationContextAware并且注入ApplicationContext的原因。上面的代码就是根据指定的信息创建一个数据源。这种创建是Spring容器级别的创建。创建完毕之后，需要把刚刚创建的这个数据源放到targetDataSources中，并且还要通知Spring容器，targetDataSources对象变了。下面的方法就是在做这样的事情：

**[java]** [view plain](http://blog.csdn.net/yizhenn/article/details/53965552#) [copy](http://blog.csdn.net/yizhenn/article/details/53965552#)

1. **private** **void** addNewDataSourceToTargerDataSources(DataSourceBean dataSourceBean) **throws** NoSuchFieldException, IllegalAccessException {

2. getTargetDataSources().put(dataSourceBean.getBeanName(), createDataSource(dataSourceBean));

3. **super**.afterPropertiesSet();//通知spring有bean更新

4. }

上面的这一步很重要。没有这一步的话，Spring压根就不会知道targetDataSources中多了一个数据源。至此DynamicDataSource类就讲完了。其实仔细想想，思路还是很清晰的。啃掉了DynamicDataSource类这块硬骨头，下面就是一些辅助类了。比如说DataSourceHolder，业务代码通过使用这个类来通知DynamicDataSource中的determineCurrentLookupKey()方法到底使用那个数据源：

**[java]** [view plain](http://blog.csdn.net/yizhenn/article/details/53965552#) [copy](http://blog.csdn.net/yizhenn/article/details/53965552#)

1. **public** **final** **class** DataSourceHolder {

2. **private** **static** ThreadLocal<DataSourceBeanBuilder> threadLocal=**new** ThreadLocal<DataSourceBeanBuilder>(){

3. @Override

4. **protected** DataSourceBeanBuilder initialValue() {

5. **return** **null**;

6. }

7. };

8.

9. **static** DataSourceBeanBuilder getDataSource(){

10. **return** threadLocal.get();

11. }

12.

13. **public** **static** **void** setDataSource(DataSourceBeanBuilder dataSourceBeanBuilder){

14. threadLocal.set(dataSourceBeanBuilder);

15. }

16.

17.

18. **public** **static** **void** clearDataSource(){

19. threadLocal.remove();

20. }

21. }

再比如这个DataSourceBean类，实际上就是用于保存从那个默认数据库中拿出来的数据源信息，只不过为了安全起见，使用了builder模式，关于builder模式，可参见[构建器模式](http://6230217.blog.51cto.com/6220217/1259621/)。

**[java]** [view plain](http://blog.csdn.net/yizhenn/article/details/53965552#) [copy](http://blog.csdn.net/yizhenn/article/details/53965552#)

1. **final** **class** DataSourceBean {

2. **private** **final** String beanName;

3. **private** **final** String driverClassName;

4. **private** **final** String url;

5. **private** **final** String username;

6. **private** **final** String password;

7. **private** **final** String validationQuery;

8. **private** **final** **boolean** testOnBorrow;

9.

10. **public** DataSourceBean(DataSourceBeanBuilder beanBuilder){

11. **this**.beanName=beanBuilder.getBeanName();

12. **this**.driverClassName=beanBuilder.getDriverClassName();

13. **this**.url=beanBuilder.getUrl();

14. **this**.password=beanBuilder.getPassword();

15. **this**.testOnBorrow=beanBuilder.isTestOnBorrow();

16. **this**.username=beanBuilder.getUsername();

17. **this**.validationQuery=beanBuilder.getValidationQuery();

18. }

19.

20. **public** String getBeanName() {

21. **return** beanName;

22. }

23.

24. **public** String getDriverClassName() {

25. **return** driverClassName;

26. }

27.

28. **public** String getUrl() {

29. **return** url;

30. }

31.

32. **public** String getUsername() {

33. **return** username;

34. }

35.

36. **public** String getPassword() {

37. **return** password;

38. }

39.

40. **public** String getValidationQuery() {

41. **return** validationQuery;

42. }

43.

44. **public** **boolean** isTestOnBorrow() {

45. **return** testOnBorrow;

46. }

47.

48.

49. @Override

50. **public** String toString() {

51. **return** "DataSourceBean{" +

52. "driverClassName='" + driverClassName + '\'' +

53. ", url='" + url + '\'' +

54. ", username='" + username + '\'' +

55. ", password='" + password + '\'' +

56. ", validationQuery='" + validationQuery + '\'' +

57. ", testOnBorrow=" + testOnBorrow +

58. '}';

59. }

60. }

自此，多数据源动态切换的组件就搞完了，有木有赶脚身体被掏空。那么身体掏空被掏空后是谁在受益呢？当然是......

额，不要想多，我是说业务代码。来看下业务代码如何切换数据源：

**[java]** [view plain](http://blog.csdn.net/yizhenn/article/details/53965552#) [copy](http://blog.csdn.net/yizhenn/article/details/53965552#)

1. @Override

2. **public** HttpResult<Boolean> testMultiDataSource(UserCreateReqDTO userCreateReqDTO) {

3. **if** (userCreateReqDTO == **null**) {

4. **return** HttpResult.successResult(Boolean.FALSE);

5. }

6.

7. UserDO userDO = UserConvent.conventToUserDO(userCreateReqDTO);

8. //先向默认数据源插入

9. **if** (!userDao.createUser(userDO)) {

10. **throw** **new** BusinessException(ErrorEnum.TEST\_MULTI\_DATASOURCE\_EXCEPTION);

11. }

12.

13. //再向起他数据源插入

14. List<DataSourceDO> dataSourceDOList = **this**.dataSourceDao.query();

15. **for** (DataSourceDO dataSourceDO : dataSourceDOList) {

16. DataSourceBeanBuilder builder = **new** DataSourceBeanBuilder(

17. dataSourceDO.getDatabaseName(),

18. dataSourceDO.getDatabaseIp(),

19. dataSourceDO.getDatabasePort(),

20. dataSourceDO.getDatasourceName(),

21. dataSourceDO.getUsername(),

22. dataSourceDO.getPassword());

23. DataSourceContext.setDataSource(builder);

24. **if** (!userDao.createUser(userDO)) {

25. **throw** **new** BusinessException(ErrorEnum.TEST\_MULTI\_DATASOURCE\_EXCEPTION);

26. }

27. DataSourceContext.clearDataSource();

28. }

29.

30. **return** HttpResult.successResult(Boolean.TRUE);

31. }

可以看到，当不适用DataSourceContext.setDataSource()方法设置数据源的时候，框架使用默认的数据源，即defaultDataSource参数配置的数据源。当时用DataSource.setDataSource()方法设置数据源之后，框架会使用指定的数据源。使用完毕后执行DataSource.clearDataSource()就又会切回到默认的数据源。

笔者已经有了一个是实现好的例子在我的github上，具体地址为[点击打开链接](https://github.com/ZhenShiErGe/test)。该工程是一个完整的ssm demo，并且其中包含了一些常用的组件，笔者还将继续增强他的功能。