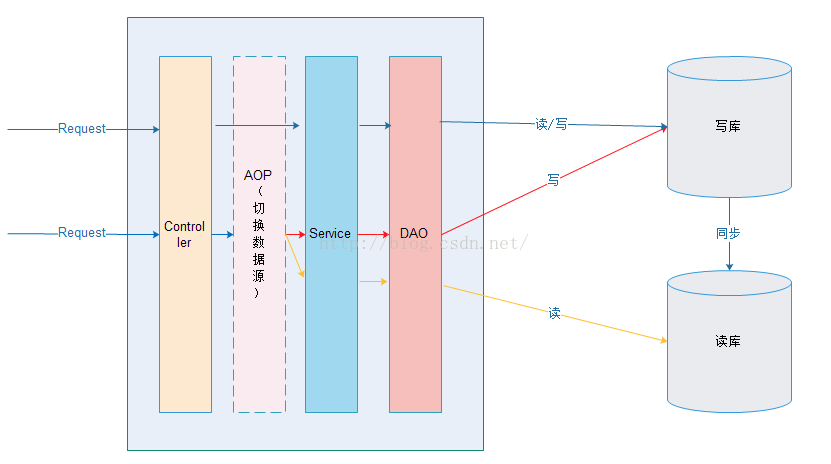
# 使用Spring基于应用层实现

## 3.1. 原理



在进入Service之前，使用AOP来做出判断，是使用写库还是读库，判断依据可以根据方法名判断，比如说以query、find、get等开头的就走读库，其他的走写库。

## 3.2. DynamicDataSource

**[java]** [view plain](http://blog.csdn.net/jack85986370/article/details/51559232" \o "view plain" \t "http://blog.csdn.net/jack85986370/article/details/_blank) [copy](http://blog.csdn.net/jack85986370/article/details/51559232" \o "copy" \t "http://blog.csdn.net/jack85986370/article/details/_blank)

1. **import** org.springframework.jdbc.datasource.lookup.AbstractRoutingDataSource;
3. /\*\*
4. \* 定义动态数据源，实现通过集成Spring提供的AbstractRoutingDataSource，只需要实现determineCurrentLookupKey方法即可
5. \*
6. \* 由于DynamicDataSource是单例的，线程不安全的，所以采用ThreadLocal保证线程安全，由DynamicDataSourceHolder完成。
7. \*
8. \* @author zhijun
9. \*
10. \*/
11. **public** **class** DynamicDataSource **extends** AbstractRoutingDataSource{
13. @Override
14. **protected** Object determineCurrentLookupKey() {
15. // 使用DynamicDataSourceHolder保证线程安全，并且得到当前线程中的数据源key
16. **return** DynamicDataSourceHolder.getDataSourceKey();
17. }
19. }

## 3.3. DynamicDataSourceHolder

**[java]** [view plain](http://blog.csdn.net/jack85986370/article/details/51559232" \o "view plain" \t "http://blog.csdn.net/jack85986370/article/details/_blank) [copy](http://blog.csdn.net/jack85986370/article/details/51559232" \o "copy" \t "http://blog.csdn.net/jack85986370/article/details/_blank)

1. <pre name="code" **class**="java">/\*\*
2. \*
3. \* 使用ThreadLocal技术来记录当前线程中的数据源的key
4. \*
5. \* @author zhijun
6. \*
7. \*/
8. **public** **class** DynamicDataSourceHolder {
10. //写库对应的数据源key
11. **private** **static** **final** String MASTER = "master";
13. //读库对应的数据源key
14. **private** **static** **final** String SLAVE = "slave";
16. //使用ThreadLocal记录当前线程的数据源key
17. **private** **static** **final** ThreadLocal<String> holder = **new** ThreadLocal<String>();
19. /\*\*
20. \* 设置数据源key
21. \* @param key
22. \*/
23. **public** **static** **void** putDataSourceKey(String key) {
24. holder.set(key);
25. }
27. /\*\*
28. \* 获取数据源key
29. \* @return
30. \*/
31. **public** **static** String getDataSourceKey() {
32. **return** holder.get();
33. }
35. /\*\*
36. \* 标记写库
37. \*/
38. **public** **static** **void** markMaster(){
39. putDataSourceKey(MASTER);
40. }
42. /\*\*
43. \* 标记读库
44. \*/
45. **public** **static** **void** markSlave(){
46. putDataSourceKey(SLAVE);
47. }
49. }

## 3.4. DataSourceAspect

**[java]** [view plain](http://blog.csdn.net/jack85986370/article/details/51559232" \o "view plain" \t "http://blog.csdn.net/jack85986370/article/details/_blank) [copy](http://blog.csdn.net/jack85986370/article/details/51559232" \o "copy" \t "http://blog.csdn.net/jack85986370/article/details/_blank)

1. **import** org.apache.commons.lang3.StringUtils;
2. **import** org.aspectj.lang.JoinPoint;
4. /\*\*
5. \* 定义数据源的AOP切面，通过该Service的方法名判断是应该走读库还是写库
6. \*
7. \* @author zhijun
8. \*
9. \*/
10. **public** **class** DataSourceAspect {
12. /\*\*
13. \* 在进入Service方法之前执行
14. \*
15. \* @param point 切面对象
16. \*/
17. **public** **void** before(JoinPoint point) {
18. // 获取到当前执行的方法名
19. String methodName = point.getSignature().getName();
20. **if** (isSlave(methodName)) {
21. // 标记为读库
22. DynamicDataSourceHolder.markSlave();
23. } **else** {
24. // 标记为写库
25. DynamicDataSourceHolder.markMaster();
26. }
27. }
29. /\*\*
30. \* 判断是否为读库
31. \*
32. \* @param methodName
33. \* @return
34. \*/
35. **private** Boolean isSlave(String methodName) {
36. // 方法名以query、find、get开头的方法名走从库
37. **return** StringUtils.startsWithAny(methodName, "query", "find", "get");
38. }
40. }

## 3.5. 配置2个数据源

### 3.5.1.  jdbc.properties

**[java]** [view plain](http://blog.csdn.net/jack85986370/article/details/51559232" \o "view plain" \t "http://blog.csdn.net/jack85986370/article/details/_blank) [copy](http://blog.csdn.net/jack85986370/article/details/51559232" \o "copy" \t "http://blog.csdn.net/jack85986370/article/details/_blank)

1. jdbc.master.driver=com.mysql.jdbc.Driver
2. jdbc.master.url=jdbc:mysql://127.0.0.1:3306/mybatis\_1128?useUnicode=true&characterEncoding=utf8&autoReconnect=true&allowMultiQueries=true
3. jdbc.master.username=root
4. jdbc.master.password=123456

7. jdbc.slave01.driver=com.mysql.jdbc.Driver
8. jdbc.slave01.url=jdbc:mysql://127.0.0.1:3307/mybatis\_1128?useUnicode=true&characterEncoding=utf8&autoReconnect=true&allowMultiQueries=true
9. jdbc.slave01.username=root
10. jdbc.slave01.password=123456

### 3.5.2.  定义连接池

**[html]** [view plain](http://blog.csdn.net/jack85986370/article/details/51559232" \o "view plain" \t "http://blog.csdn.net/jack85986370/article/details/_blank) [copy](http://blog.csdn.net/jack85986370/article/details/51559232" \o "copy" \t "http://blog.csdn.net/jack85986370/article/details/_blank)

1. <!-- 配置连接池 -->
2. **<bean** id="masterDataSource" class="com.jolbox.bonecp.BoneCPDataSource"
3. destroy-method="close"**>**
4. <!-- 数据库驱动 -->
5. **<property** name="driverClass" value="${jdbc.master.driver}" **/>**
6. <!-- 相应驱动的jdbcUrl -->
7. **<property** name="jdbcUrl" value="${jdbc.master.url}" **/>**
8. <!-- 数据库的用户名 -->
9. **<property** name="username" value="${jdbc.master.username}" **/>**
10. <!-- 数据库的密码 -->
11. **<property** name="password" value="${jdbc.master.password}" **/>**
12. <!-- 检查数据库连接池中空闲连接的间隔时间，单位是分，默认值：240，如果要取消则设置为0 -->
13. **<property** name="idleConnectionTestPeriod" value="60" **/>**
14. <!-- 连接池中未使用的链接最大存活时间，单位是分，默认值：60，如果要永远存活设置为0 -->
15. **<property** name="idleMaxAge" value="30" **/>**
16. <!-- 每个分区最大的连接数 -->
17. **<property** name="maxConnectionsPerPartition" value="150" **/>**
18. <!-- 每个分区最小的连接数 -->
19. **<property** name="minConnectionsPerPartition" value="5" **/>**
20. **</bean>**
22. <!-- 配置连接池 -->
23. **<bean** id="slave01DataSource" class="com.jolbox.bonecp.BoneCPDataSource"
24. destroy-method="close"**>**
25. <!-- 数据库驱动 -->
26. **<property** name="driverClass" value="${jdbc.slave01.driver}" **/>**
27. <!-- 相应驱动的jdbcUrl -->
28. **<property** name="jdbcUrl" value="${jdbc.slave01.url}" **/>**
29. <!-- 数据库的用户名 -->
30. **<property** name="username" value="${jdbc.slave01.username}" **/>**
31. <!-- 数据库的密码 -->
32. **<property** name="password" value="${jdbc.slave01.password}" **/>**
33. <!-- 检查数据库连接池中空闲连接的间隔时间，单位是分，默认值：240，如果要取消则设置为0 -->
34. **<property** name="idleConnectionTestPeriod" value="60" **/>**
35. <!-- 连接池中未使用的链接最大存活时间，单位是分，默认值：60，如果要永远存活设置为0 -->
36. **<property** name="idleMaxAge" value="30" **/>**
37. <!-- 每个分区最大的连接数 -->
38. **<property** name="maxConnectionsPerPartition" value="150" **/>**
39. <!-- 每个分区最小的连接数 -->
40. **<property** name="minConnectionsPerPartition" value="5" **/>**
41. **</bean>**

### 3.5.3.  定义DataSource

**[html]** [view plain](http://blog.csdn.net/jack85986370/article/details/51559232" \o "view plain" \t "http://blog.csdn.net/jack85986370/article/details/_blank) [copy](http://blog.csdn.net/jack85986370/article/details/51559232" \o "copy" \t "http://blog.csdn.net/jack85986370/article/details/_blank)

1. <!-- 定义数据源，使用自己实现的数据源 -->
2. **<bean** id="dataSource" class="cn.itcast.usermanage.spring.DynamicDataSource"**>**
3. <!-- 设置多个数据源 -->
4. **<property** name="targetDataSources"**>**
5. **<map** key-type="java.lang.String"**>**
6. <!-- 这个key需要和程序中的key一致 -->
7. **<entry** key="master" value-ref="masterDataSource"**/>**
8. **<entry** key="slave" value-ref="slave01DataSource"**/>**
9. **</map>**
10. **</property>**
11. <!-- 设置默认的数据源，这里默认走写库 -->
12. **<property** name="defaultTargetDataSource" ref="masterDataSource"**/>**
13. **</bean>**

## 3.6. 配置事务管理以及动态切换数据源切面

### 3.6.1.  定义事务管理器

**[html]** [view plain](http://blog.csdn.net/jack85986370/article/details/51559232" \o "view plain" \t "http://blog.csdn.net/jack85986370/article/details/_blank) [copy](http://blog.csdn.net/jack85986370/article/details/51559232" \o "copy" \t "http://blog.csdn.net/jack85986370/article/details/_blank)

1. <!-- 定义事务管理器 -->
2. **<bean** id="transactionManager"
3. class="org.springframework.jdbc.datasource.DataSourceTransactionManager"**>**
4. **<property** name="dataSource" ref="dataSource" **/>**
5. **</bean>**

### 3.6.2.  定义事务策略

**[html]** [view plain](http://blog.csdn.net/jack85986370/article/details/51559232" \o "view plain" \t "http://blog.csdn.net/jack85986370/article/details/_blank) [copy](http://blog.csdn.net/jack85986370/article/details/51559232" \o "copy" \t "http://blog.csdn.net/jack85986370/article/details/_blank)

1. <!-- 定义事务策略 -->
2. **<tx:advice** id="txAdvice" transaction-manager="transactionManager"**>**
3. **<tx:attributes>**
4. <!--定义查询方法都是只读的 -->
5. **<tx:method** name="query\*" read-only="true" **/>**
6. **<tx:method** name="find\*" read-only="true" **/>**
7. **<tx:method** name="get\*" read-only="true" **/>**
9. <!-- 主库执行操作，事务传播行为定义为默认行为 -->
10. **<tx:method** name="save\*" propagation="REQUIRED" **/>**
11. **<tx:method** name="update\*" propagation="REQUIRED" **/>**
12. **<tx:method** name="delete\*" propagation="REQUIRED" **/>**
14. <!--其他方法使用默认事务策略 -->
15. **<tx:method** name="\*" **/>**
16. **</tx:attributes>**
17. **</tx:advice>**

### 3.6.3.  定义切面

**[html]** [view plain](http://blog.csdn.net/jack85986370/article/details/51559232" \o "view plain" \t "http://blog.csdn.net/jack85986370/article/details/_blank) [copy](http://blog.csdn.net/jack85986370/article/details/51559232" \o "copy" \t "http://blog.csdn.net/jack85986370/article/details/_blank)

1. <!-- 定义AOP切面处理器 -->
2. **<bean** class="cn.itcast.usermanage.spring.DataSourceAspect" id="dataSourceAspect" **/>**
4. **<aop:config>**
5. <!-- 定义切面，所有的service的所有方法 -->
6. **<aop:pointcut** id="txPointcut" expression="execution(\* xx.xxx.xxxxxxx.service.\*.\*(..))" **/>**
7. <!-- 应用事务策略到Service切面 -->
8. **<aop:advisor** advice-ref="txAdvice" pointcut-ref="txPointcut"**/>**
10. <!-- 将切面应用到自定义的切面处理器上，-9999保证该切面优先级最高执行 -->
11. **<aop:aspect** ref="dataSourceAspect" order="-9999"**>**
12. **<aop:before** method="before" pointcut-ref="txPointcut" **/>**
13. **</aop:aspect>**
14. **</aop:config>**

# 4.  改进切面实现，使用事务策略规则匹配

之前的实现我们是将通过方法名匹配，而不是使用事务策略中的定义，我们使用事务管理策略中的规则匹配。

## 4.1. 改进后的配置

**[html]** [view plain](http://blog.csdn.net/jack85986370/article/details/51559232" \o "view plain" \t "http://blog.csdn.net/jack85986370/article/details/_blank) [copy](http://blog.csdn.net/jack85986370/article/details/51559232" \o "copy" \t "http://blog.csdn.net/jack85986370/article/details/_blank)

1. **<span** style="white-space:pre"**>**    **</span>**<!-- 定义AOP切面处理器 -->
2. **<bean** class="cn.itcast.usermanage.spring.DataSourceAspect" id="dataSourceAspect"**>**
3. <!-- 指定事务策略 -->
4. **<property** name="txAdvice" ref="txAdvice"**/>**
5. <!-- 指定slave方法的前缀（非必须） -->
6. **<property** name="slaveMethodStart" value="query,find,get"**/>**
7. **</bean>**

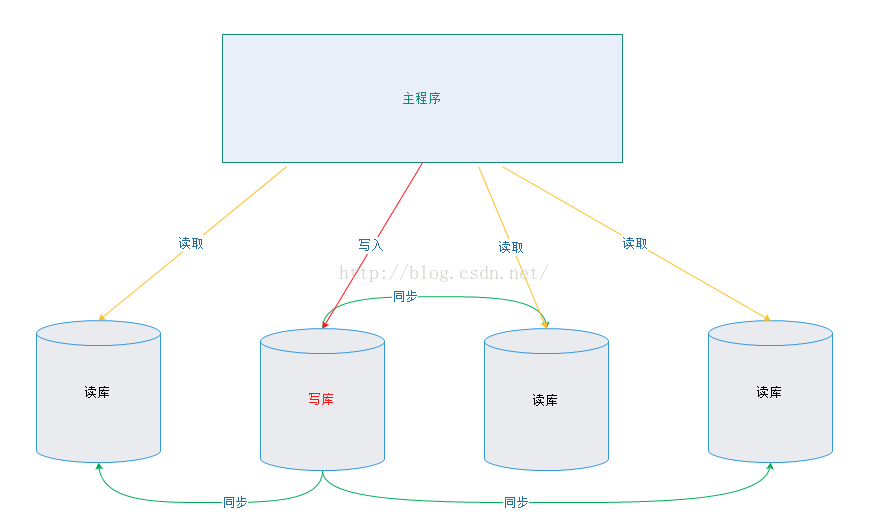
## 4.2. 改进后的实现

**[java]** [view plain](http://blog.csdn.net/jack85986370/article/details/51559232" \o "view plain" \t "http://blog.csdn.net/jack85986370/article/details/_blank) [copy](http://blog.csdn.net/jack85986370/article/details/51559232" \o "copy" \t "http://blog.csdn.net/jack85986370/article/details/_blank)

1. **import** java.lang.reflect.Field;
2. **import** java.util.ArrayList;
3. **import** java.util.List;
4. **import** java.util.Map;
6. **import** org.apache.commons.lang3.StringUtils;
7. **import** org.aspectj.lang.JoinPoint;
8. **import** org.springframework.transaction.interceptor.NameMatchTransactionAttributeSource;
9. **import** org.springframework.transaction.interceptor.TransactionAttribute;
10. **import** org.springframework.transaction.interceptor.TransactionAttributeSource;
11. **import** org.springframework.transaction.interceptor.TransactionInterceptor;
12. **import** org.springframework.util.PatternMatchUtils;
13. **import** org.springframework.util.ReflectionUtils;
15. /\*\*
16. \* 定义数据源的AOP切面，该类控制了使用Master还是Slave。
17. \*
18. \* 如果事务管理中配置了事务策略，则采用配置的事务策略中的标记了ReadOnly的方法是用Slave，其它使用Master。
19. \*
20. \* 如果没有配置事务管理的策略，则采用方法名匹配的原则，以query、find、get开头方法用Slave，其它用Master。
21. \*
22. \* @author zhijun
23. \*
24. \*/
25. **public** **class** DataSourceAspect {
27. **private** List<String> slaveMethodPattern = **new** ArrayList<String>();
29. **private** **static** **final** String[] defaultSlaveMethodStart = **new** String[]{ "query", "find", "get" };
31. **private** String[] slaveMethodStart;
33. /\*\*
34. \* 读取事务管理中的策略
35. \*
36. \* @param txAdvice
37. \* @throws Exception
38. \*/
39. @SuppressWarnings("unchecked")
40. **public** **void** setTxAdvice(TransactionInterceptor txAdvice) **throws** Exception {
41. **if** (txAdvice == **null**) {
42. // 没有配置事务管理策略
43. **return**;
44. }
45. //从txAdvice获取到策略配置信息
46. TransactionAttributeSource transactionAttributeSource = txAdvice.getTransactionAttributeSource();
47. **if** (!(transactionAttributeSource **instanceof** NameMatchTransactionAttributeSource)) {
48. **return**;
49. }
50. //使用反射技术获取到NameMatchTransactionAttributeSource对象中的nameMap属性值
51. NameMatchTransactionAttributeSource matchTransactionAttributeSource = (NameMatchTransactionAttributeSource) transactionAttributeSource;
52. Field nameMapField = ReflectionUtils.findField(NameMatchTransactionAttributeSource.**class**, "nameMap");
53. nameMapField.setAccessible(**true**); //设置该字段可访问
54. //获取nameMap的值
55. Map<String, TransactionAttribute> map = (Map<String, TransactionAttribute>) nameMapField.get(matchTransactionAttributeSource);
57. //遍历nameMap
58. **for** (Map.Entry<String, TransactionAttribute> entry : map.entrySet()) {
59. **if** (!entry.getValue().isReadOnly()) {//判断之后定义了ReadOnly的策略才加入到slaveMethodPattern
60. **continue**;
61. }
62. slaveMethodPattern.add(entry.getKey());
63. }
64. }
66. /\*\*
67. \* 在进入Service方法之前执行
68. \*
69. \* @param point 切面对象
70. \*/
71. **public** **void** before(JoinPoint point) {
72. // 获取到当前执行的方法名
73. String methodName = point.getSignature().getName();
75. **boolean** isSlave = **false**;
77. **if** (slaveMethodPattern.isEmpty()) {
78. // 当前Spring容器中没有配置事务策略，采用方法名匹配方式
79. isSlave = isSlave(methodName);
80. } **else** {
81. // 使用策略规则匹配
82. **for** (String mappedName : slaveMethodPattern) {
83. **if** (isMatch(methodName, mappedName)) {
84. isSlave = **true**;
85. **break**;
86. }
87. }
88. }
90. **if** (isSlave) {
91. // 标记为读库
92. DynamicDataSourceHolder.markSlave();
93. } **else** {
94. // 标记为写库
95. DynamicDataSourceHolder.markMaster();
96. }
97. }
99. /\*\*
100. \* 判断是否为读库
101. \*
102. \* @param methodName
103. \* @return
104. \*/
105. **private** Boolean isSlave(String methodName) {
106. // 方法名以query、find、get开头的方法名走从库
107. **return** StringUtils.startsWithAny(methodName, getSlaveMethodStart());
108. }
110. /\*\*
111. \* 通配符匹配
112. \*
113. \* Return if the given method name matches the mapped name.
114. \* <p>
115. \* The default implementation checks for "xxx\*", "\*xxx" and "\*xxx\*" matches, as well as direct
116. \* equality. Can be overridden in subclasses.
117. \*
118. \* @param methodName the method name of the class
119. \* @param mappedName the name in the descriptor
120. \* @return if the names match
121. \* @see org.springframework.util.PatternMatchUtils#simpleMatch(String, String)
122. \*/
123. **protected** **boolean** isMatch(String methodName, String mappedName) {
124. **return** PatternMatchUtils.simpleMatch(mappedName, methodName);
125. }
127. /\*\*
128. \* 用户指定slave的方法名前缀
129. \* @param slaveMethodStart
130. \*/
131. **public** **void** setSlaveMethodStart(String[] slaveMethodStart) {
132. **this**.slaveMethodStart = slaveMethodStart;
133. }
135. **public** String[] getSlaveMethodStart() {
136. **if**(**this**.slaveMethodStart == **null**){
137. // 没有指定，使用默认
138. **return** defaultSlaveMethodStart;
139. }
140. **return** slaveMethodStart;
141. }
143. }

# 5.  一主多从的实现

很多实际使用场景下都是采用“一主多从”的架构的，所有我们现在对这种架构做支持，目前只需要修改DynamicDataSource即可。



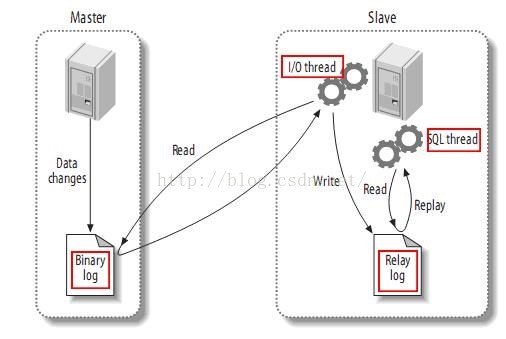
## 5.1. 实现

**[java]** [view plain](http://blog.csdn.net/jack85986370/article/details/51559232" \o "view plain" \t "http://blog.csdn.net/jack85986370/article/details/_blank) [copy](http://blog.csdn.net/jack85986370/article/details/51559232" \o "copy" \t "http://blog.csdn.net/jack85986370/article/details/_blank)

1. **import** java.lang.reflect.Field;
2. **import** java.util.ArrayList;
3. **import** java.util.List;
4. **import** java.util.Map;
5. **import** java.util.concurrent.atomic.AtomicInteger;
7. **import** javax.sql.DataSource;
9. **import** org.slf4j.Logger;
10. **import** org.slf4j.LoggerFactory;
11. **import** org.springframework.jdbc.datasource.lookup.AbstractRoutingDataSource;
12. **import** org.springframework.util.ReflectionUtils;
14. /\*\*
15. \* 定义动态数据源，实现通过集成Spring提供的AbstractRoutingDataSource，只需要实现determineCurrentLookupKey方法即可
16. \*
17. \* 由于DynamicDataSource是单例的，线程不安全的，所以采用ThreadLocal保证线程安全，由DynamicDataSourceHolder完成。
18. \*
19. \* @author zhijun
20. \*
21. \*/
22. **public** **class** DynamicDataSource **extends** AbstractRoutingDataSource {
24. **private** **static** **final** Logger LOGGER = LoggerFactory.getLogger(DynamicDataSource.**class**);
26. **private** Integer slaveCount;
28. // 轮询计数,初始为-1,AtomicInteger是线程安全的
29. **private** AtomicInteger counter = **new** AtomicInteger(-1);
31. // 记录读库的key
32. **private** List<Object> slaveDataSources = **new** ArrayList<Object>(0);
34. @Override
35. **protected** Object determineCurrentLookupKey() {
36. // 使用DynamicDataSourceHolder保证线程安全，并且得到当前线程中的数据源key
37. **if** (DynamicDataSourceHolder.isMaster()) {
38. Object key = DynamicDataSourceHolder.getDataSourceKey();
39. **if** (LOGGER.isDebugEnabled()) {
40. LOGGER.debug("当前DataSource的key为: " + key);
41. }
42. **return** key;
43. }
44. Object key = getSlaveKey();
45. **if** (LOGGER.isDebugEnabled()) {
46. LOGGER.debug("当前DataSource的key为: " + key);
47. }
48. **return** key;
50. }
52. @SuppressWarnings("unchecked")
53. @Override
54. **public** **void** afterPropertiesSet() {
55. **super**.afterPropertiesSet();
57. // 由于父类的resolvedDataSources属性是私有的子类获取不到，需要使用反射获取
58. Field field = ReflectionUtils.findField(AbstractRoutingDataSource.**class**, "resolvedDataSources");
59. field.setAccessible(**true**); // 设置可访问
61. **try** {
62. Map<Object, DataSource> resolvedDataSources = (Map<Object, DataSource>) field.get(**this**);
63. // 读库的数据量等于数据源总数减去写库的数量
64. **this**.slaveCount = resolvedDataSources.size() - 1;
65. **for** (Map.Entry<Object, DataSource> entry : resolvedDataSources.entrySet()) {
66. **if** (DynamicDataSourceHolder.MASTER.equals(entry.getKey())) {
67. **continue**;
68. }
69. slaveDataSources.add(entry.getKey());
70. }
71. } **catch** (Exception e) {
72. LOGGER.error("afterPropertiesSet error! ", e);
73. }
74. }
76. /\*\*
77. \* 轮询算法实现
78. \*
79. \* @return
80. \*/
81. **public** Object getSlaveKey() {
82. // 得到的下标为：0、1、2、3……
83. Integer index = counter.incrementAndGet() % slaveCount;
84. **if** (counter.get() > 9999) { // 以免超出Integer范围
85. counter.set(-1); // 还原
86. }
87. **return** slaveDataSources.get(index);
88. }
90. }

# 6.  MySQL主从复制

## 6.1. 原理



mysql主(称master)从(称slave)复制的原理：

1、master将数据改变记录到二进制日志(binarylog)中,也即是配置文件log-bin指定的文件(这些记录叫做二进制日志事件，binary log events)

2、slave将master的binary logevents拷贝到它的中继日志(relay log)

3、slave重做中继日志中的事件,将改变反映它自己的数据(数据重演)

## 6.2. 主从配置需要注意的地方

1、主DB server和从DB server数据库的版本一致

2、主DB server和从DB server数据库数据一致[ 这里就会可以把主的备份在从上还原，也可以直接将主的数据目录拷贝到从的相应数据目录]

3、主DB server开启二进制日志,主DB server和从DB server的server\_id都必须唯一

## 6.3. 主库配置（windows，Linux下也类似）

在my.ini修改：

#开启主从复制，主库的配置

log-bin= mysql3306-bin

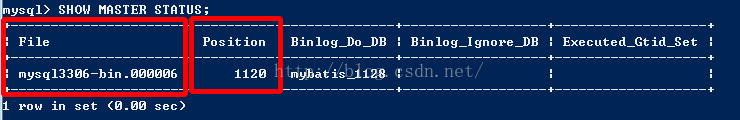
#指定主库serverid

server-id=101

#指定同步的数据库，如果不指定则同步全部数据库

binlog-do-db=mybatis\_1128

执行SQL语句查询状态：  
SHOW MASTER STATUS



需要记录下Position值，需要在从库中设置同步起始值。

## 6.4. 在主库创建同步用户

#授权用户slave01使用123456密码登录mysql

grant replication slave on \*.\* to 'slave01'@'127.0.0.1'identified by '123456';

flush privileges;

## 6.5. 从库配置

在my.ini修改：

#指定serverid，只要不重复即可，从库也只有这一个配置，其他都在SQL语句中操作

server-id=102

以下执行SQL：

CHANGEMASTER TO

 master\_host='127.0.0.1',

 master\_user='slave01',

 master\_password='123456',

 master\_port=3306,

 master\_log\_file='mysql3306-bin.000006',

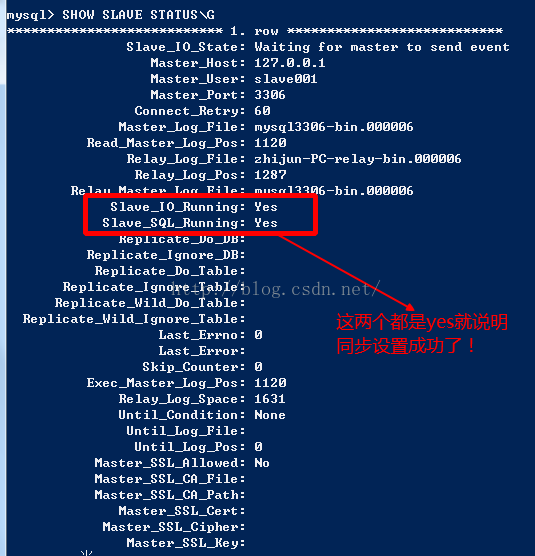
 master\_log\_pos=1120;

#启动slave同步

STARTSLAVE;

#查看同步状态

SHOWSLAVE STATUS;



# 7.  参考资料

[http://www.iteye.com/topic/1127642](http://www.iteye.com/topic/1127642" \t "http://blog.csdn.net/jack85986370/article/details/_blank)

[http://634871.blog.51cto.com/624871/1329301](http://634871.blog.51cto.com/624871/1329301" \t "http://blog.csdn.net/jack85986370/article/details/_blank)