**Mybatis中的数据源与连接池详解**

<https://blog.csdn.net/majinggogogo/article/details/71715846>

原创 2017年05月14日 15:05:15

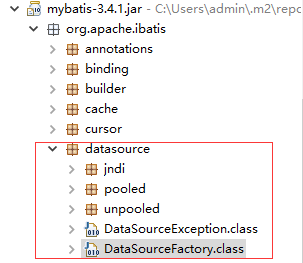
* 标签：
* [mybatis](http://so.csdn.net/so/search/s.do?q=mybatis&t=blog) /
* [架构](http://so.csdn.net/so/search/s.do?q=%E6%9E%B6%E6%9E%84&t=blog) /
* [源码](http://so.csdn.net/so/search/s.do?q=%E6%BA%90%E7%A0%81&t=blog) /
* [数据库](http://so.csdn.net/so/search/s.do?q=%E6%95%B0%E6%8D%AE%E5%BA%93&t=blog)
* 4554

在前面的文章《[Mybatis配置之<environments>配置元素详述](http://blog.csdn.net/majinggogogo/article/details/71616712)》中我们已经知道里面可以配置两个元素，一个是数据源及连接池的配置，一个是事务管理器的配置。在上篇文章中我们只是简单的描述了一下，从这篇文章开始，我们将分两篇博文，分别对这两个问题进行详细说明。

这篇文章我们先来了解一下数据源及连接池的配置。

（1）Mybatis中支持的数据源

在上篇文章中，我们知道Mybatis中支持三种形式数据源的配置，分别为：UNPOOLED、POOLED和JNDI，如下红色区域所示：



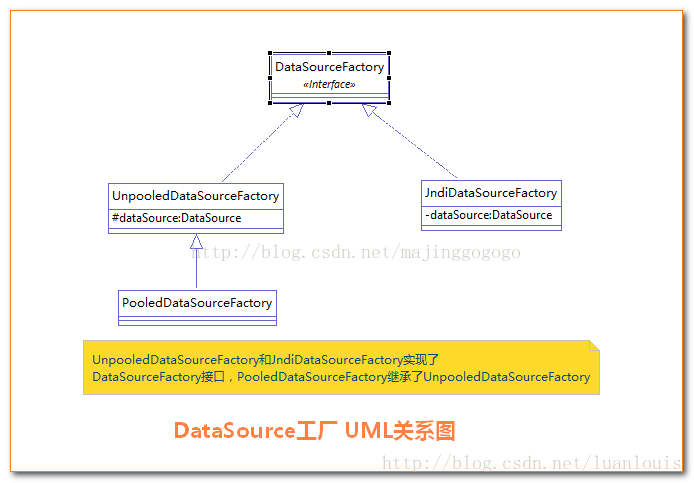
在Mybatis内部定义了一个接口DataSourceFactory，而支持的三种形式都需要实现这个接口。DataSourceFactory接口定义如下：

**[java]** [view plain](https://blog.csdn.net/majinggogogo/article/details/71715846) [copy](https://blog.csdn.net/majinggogogo/article/details/71715846)

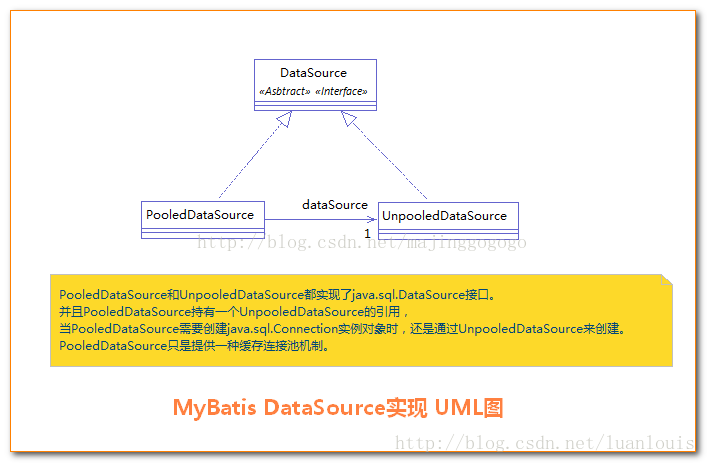
1. <span style="font-family:'KaiTi\_GB2312';font-size:18px;">**package** org.apache.ibatis.datasource;
3. **import** java.util.Properties;
4. **import** javax.sql.DataSource;
6. /\*\*
7. \* @author Clinton Begin
8. \*/
9. **public** **interface** DataSourceFactory {
11. **void** setProperties(Properties props);
13. DataSource getDataSource();
15. }</span>

与UNPOOLED、POOLED和JNDI相对应的，在mybatis内部定义实现了DataSourceFactory接口的三个类，分别为UnpooledDataSourceFactory、PooledDataSourceFactory和JndiDataSourceFactory。

具体结构如下所示：



与这些数据源工厂类相对应的也定义了相应的数据源对象，其中UnpooledDataSourceFactory和PooledDataSourceFactory工厂返回的分别是UnpooledDataSource和PooledDataSource，而JndiDataSourceFactory则是根据配置返回相应的数据源。



（2）mybatis中数据源的创建过程

首先从配置文件开始看起：

**[html]** [view plain](https://blog.csdn.net/majinggogogo/article/details/71715846) [copy](https://blog.csdn.net/majinggogogo/article/details/71715846)

1. **<span** style="font-family:'KaiTi\_GB2312';font-size:18px;"**>**<!-- 数据库连接池 -->
2. **<dataSource** type="POOLED"**>**
3. **<property** name="driver" value="${driver}" **/>**
4. **<property** name="url" value="${url}" **/>**
5. **<property** name="username" value="${username}" **/>**
6. **<property** name="password" value="${password}" **/>**
7. **</dataSource></span>**

（a）在mybatis初始化的时候，在解析到<dataSource>节点时，会根据相应的type类型设置来创建相应的数据源工厂类实例，如下所示：

**[java]** [view plain](https://blog.csdn.net/majinggogogo/article/details/71715846) [copy](https://blog.csdn.net/majinggogogo/article/details/71715846)

1. <span style="font-family:'KaiTi\_GB2312';font-size:18px;">DataSourceFactory dsFactory = dataSourceElement(child.evalNode("dataSource"));</span>

**[java]** [view plain](https://blog.csdn.net/majinggogogo/article/details/71715846) [copy](https://blog.csdn.net/majinggogogo/article/details/71715846)

1. <span style="font-family:'KaiTi\_GB2312';font-size:18px;">**private** DataSourceFactory dataSourceElement(XNode context) **throws** Exception {
2. **if** (context != **null**) {
3. String type = context.getStringAttribute("type");
4. Properties props = context.getChildrenAsProperties();
5. DataSourceFactory factory = (DataSourceFactory) resolveClass(type).newInstance();
6. factory.setProperties(props);
7. **return** factory;
8. }
9. **throw** **new** BuilderException("Environment declaration requires a DataSourceFactory.");
10. }</span>

在上面代码里，根据type类型去寻找相应的数据源工厂类并实例化一个。具体每一个配置对应什么类，在Configuration类中已经进行了声明，如下所示：

**[java]** [view plain](https://blog.csdn.net/majinggogogo/article/details/71715846) [copy](https://blog.csdn.net/majinggogogo/article/details/71715846)

1. <span style="font-family:'KaiTi\_GB2312';font-size:18px;">typeAliasRegistry.registerAlias("JNDI", JndiDataSourceFactory.**class**);
2. typeAliasRegistry.registerAlias("POOLED", PooledDataSourceFactory.**class**);
3. typeAliasRegistry.registerAlias("UNPOOLED", UnpooledDataSourceFactory.**class**);</span>

（b）之后，从数据源工厂类实例中通过getDataSource()方法获取一个DataSource对象；

（c）MyBatis创建了DataSource实例后，会将其放到Configuration对象内的Environment对象中， 供以后使用。如下所示：

**[java]** [view plain](https://blog.csdn.net/majinggogogo/article/details/71715846) [copy](https://blog.csdn.net/majinggogogo/article/details/71715846)

1. <span style="font-family:'KaiTi\_GB2312';font-size:18px;">DataSourceFactory dsFactory = dataSourceElement(child.evalNode("dataSource"));
2. DataSource dataSource = dsFactory.getDataSource();
3. Environment.Builder environmentBuilder = **new** Environment.Builder(id)
4. .transactionFactory(txFactory)
5. .dataSource(dataSource);
6. configuration.setEnvironment(environmentBuilder.build());</span>

（3）数据源DataSource对象什么时候创建数据库连接

当我们需要创建SqlSession对象并需要执行SQL语句时，这时候MyBatis才会去调用dataSource对象来创建java.sql.Connection对象。也就是说，java.sql.Connection对象的创建一直延迟到执行SQL语句的时候。

**[java]** [view plain](https://blog.csdn.net/majinggogogo/article/details/71715846) [copy](https://blog.csdn.net/majinggogogo/article/details/71715846)

1. <span style="font-family:'KaiTi\_GB2312';font-size:18px;">**public** **void** testFindUserById(){
2. SqlSession sqlSession = getSessionFactory().openSession(**true**);
3. UserDao userMapper = sqlSession.getMapper(UserDao.**class**);
5. User user = userMapper.findUserById(10);
6. System.out.println("记录为："+user);
7. }</span>

对于上面这段代码，我们通过调试会发现，在前两句的时候其实是没有创建数据库连接的，而是在执行userMapper.findUserById()方法的时候才触发了数据库连接的创建。

（4）非池化的数据源UnpooledDataSource

我们先直接从代码入手：

**[java]** [view plain](https://blog.csdn.net/majinggogogo/article/details/71715846) [copy](https://blog.csdn.net/majinggogogo/article/details/71715846)

1. <span style="font-family:'KaiTi\_GB2312';font-size:18px;">@Override
2. **public** Connection getConnection() **throws** SQLException {
3. **return** doGetConnection(username, password);
4. }</span>

**[java]** [view plain](https://blog.csdn.net/majinggogogo/article/details/71715846) [copy](https://blog.csdn.net/majinggogogo/article/details/71715846)

1. <span style="font-family:'KaiTi\_GB2312';font-size:18px;">**private** Connection doGetConnection(String username, String password) **throws** SQLException {
2. Properties props = **new** Properties();
3. **if** (driverProperties != **null**) {
4. props.putAll(driverProperties);
5. }
6. **if** (username != **null**) {
7. props.setProperty("user", username);
8. }
9. **if** (password != **null**) {
10. props.setProperty("password", password);
11. }
12. **return** doGetConnection(props);
13. }</span>

**[java]** [view plain](https://blog.csdn.net/majinggogogo/article/details/71715846) [copy](https://blog.csdn.net/majinggogogo/article/details/71715846)

1. <span style="font-family:'KaiTi\_GB2312';font-size:18px;">**private** Connection doGetConnection(Properties properties) **throws** SQLException {
2. initializeDriver();
3. Connection connection = DriverManager.getConnection(url, properties);
4. configureConnection(connection);
5. **return** connection;
6. }</span>

从上面的代码可以知道UnpooledDataSource创建数据库连接的主要流程，具体时序图如下所示：

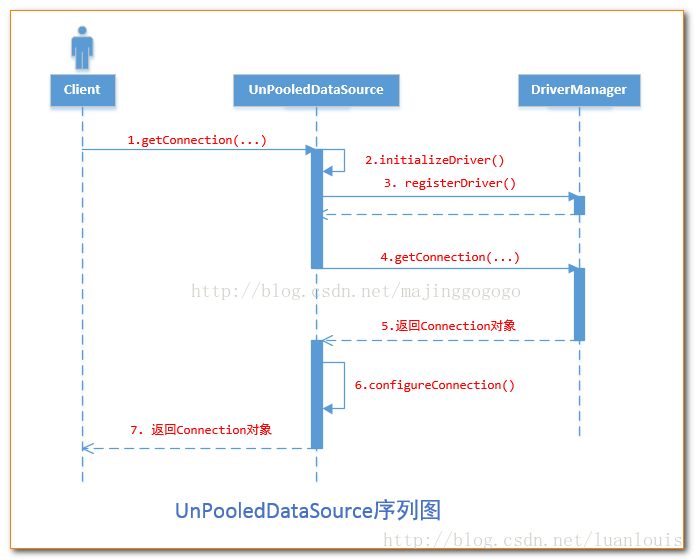
（a）调用initializeDriver()方法进行驱动的初始化；

判断driver驱动是否已经加载到内存中，如果还没有加载，则会动态地加载driver类，并实例化一个Driver对象，使用DriverManager.registerDriver()方法将其注册到内存中，以供后续使用。

（b）调用DriverManager.getConnection()获取数据库连接；

（c）对数据库连接进行一些设置，并返回数据库连接Connection;

设置数据库连接是否自动提交，设置事务级别等。



有人可能会有疑问，这里的username和password是什么传递给数据源的呢？

这个问题其实上面已经提到过了，在mybatis初始化的时候，就已经解析了<dataSource>元素，并将其下相关的<property>配置作为数据源的配置初始化进去了。也就是下面这段逻辑：

**[java]** [view plain](https://blog.csdn.net/majinggogogo/article/details/71715846) [copy](https://blog.csdn.net/majinggogogo/article/details/71715846)

1. <span style="font-family:'KaiTi\_GB2312';font-size:18px;">Properties props = context.getChildrenAsProperties();
2. DataSourceFactory factory = (DataSourceFactory) resolveClass(type).newInstance();
3. factory.setProperties(props);</span>

至此，对于UnpooledDataSource数据源算是有比较清楚的了解了。下面我们看看带连接池的PooledDataSource

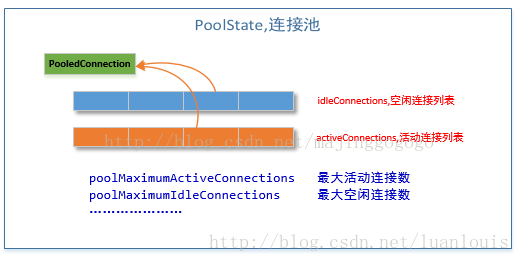
（5）带连接池的PooledDataSource

为什么要使用带连接池的数据源呢，最根本的原因还是因为每次创建连接开销比较大，频繁的创建和关闭数据库连接将会严重的影响性能。因此，常用的做法是维护一个数据库连接池，每次使用完之后并不是直接关闭数据库连接，再后面如果需要创建数据库连接的时候直接拿之前释放的数据库连接使用，避免频繁创建和关闭数据库连接造成的开销。

在mybatis中，定义了一个数据库连接池状态的类PoolState，在这个类里，除维护了数据源实例，还维护着数据库连接。数据库连接被分成了两种状态类型并存放在两个列表中：idleConnections和activeConnections。

idleConnections:

空闲(idle)状态PooledConnection对象被放置到此集合中，表示当前闲置的没有被使用的PooledConnection集合，调用PooledDataSource的getConnection()方法时，会优先从此集合中取PooledConnection对象。当用完一个java.sql.Connection对象时，MyBatis会将其包裹成PooledConnection对象放到此集合中。  
  
activeConnections:活动(active)状态的PooledConnection对象被放置到名为activeConnections的ArrayList中，表示当前正在被使用的PooledConnection集合，调用PooledDataSource的getConnection()方法时，会优先从idleConnections集合中取PooledConnection对象,如果没有，则看此集合是否已满，如果未满，PooledDataSource会创建出一个PooledConnection，添加到此集合中，并返回。



下面我们看看怎么从连接池中获取一个数据库连接，还是从PooledDataSource类开始看起。

**[java]** [view plain](https://blog.csdn.net/majinggogogo/article/details/71715846) [copy](https://blog.csdn.net/majinggogogo/article/details/71715846)

1. @Override
2. **public** Connection getConnection() **throws** SQLException {
3. **return** popConnection(dataSource.getUsername(), dataSource.getPassword()).getProxyConnection();
4. }
6. @Override
7. **public** Connection getConnection(String username, String password) **throws** SQLException {
8. **return** popConnection(username, password).getProxyConnection();
9. }

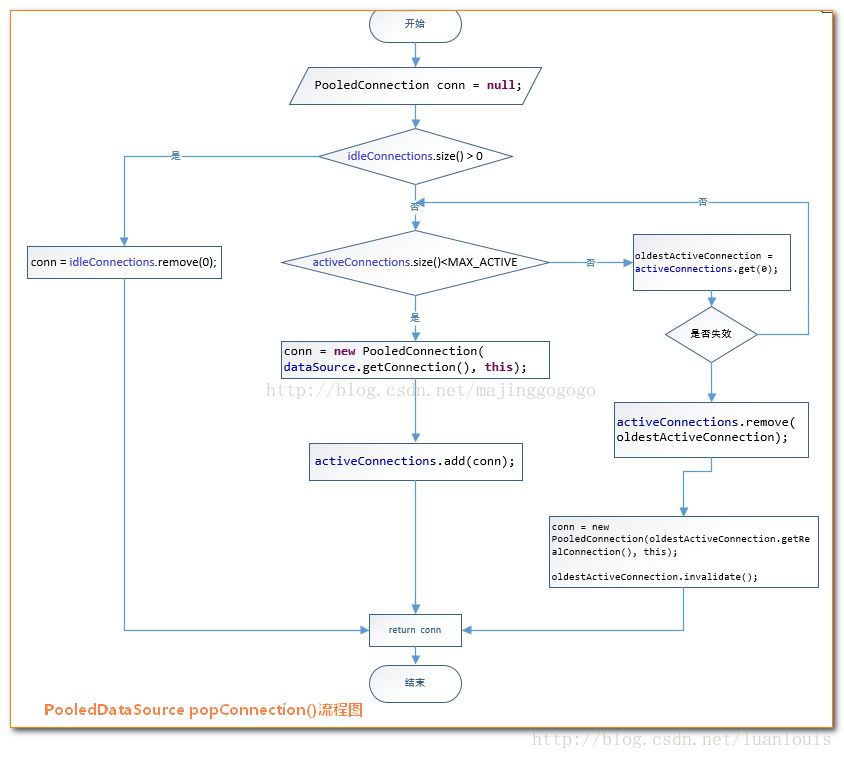
这里都是调用了popConnection()方法，然后返回其代理对象。

**[java]** [view plain](https://blog.csdn.net/majinggogogo/article/details/71715846) [copy](https://blog.csdn.net/majinggogogo/article/details/71715846)

1. **private** PooledConnection popConnection(String username, String password) **throws** SQLException {
2. **boolean** countedWait = **false**;
3. PooledConnection conn = **null**;
4. **long** t = System.currentTimeMillis();
5. **int** localBadConnectionCount = 0;
7. **while** (conn == **null**) {
8. **synchronized** (state) {
9. **if** (!state.idleConnections.isEmpty()) {
10. // Pool has available connection
11. conn = state.idleConnections.remove(0);
12. **if** (log.isDebugEnabled()) {
13. log.debug("Checked out connection " + conn.getRealHashCode() + " from pool.");
14. }
15. } **else** {
16. // Pool does not have available connection
17. **if** (state.activeConnections.size() < poolMaximumActiveConnections) {
18. // Can create new connection
19. conn = **new** PooledConnection(dataSource.getConnection(), **this**);
20. **if** (log.isDebugEnabled()) {
21. log.debug("Created connection " + conn.getRealHashCode() + ".");
22. }
23. } **else** {
24. // Cannot create new connection
25. PooledConnection oldestActiveConnection = state.activeConnections.get(0);
26. **long** longestCheckoutTime = oldestActiveConnection.getCheckoutTime();
27. **if** (longestCheckoutTime > poolMaximumCheckoutTime) {
28. // Can claim overdue connection
29. state.claimedOverdueConnectionCount++;
30. state.accumulatedCheckoutTimeOfOverdueConnections += longestCheckoutTime;
31. state.accumulatedCheckoutTime += longestCheckoutTime;
32. state.activeConnections.remove(oldestActiveConnection);
33. **if** (!oldestActiveConnection.getRealConnection().getAutoCommit()) {
34. **try** {
35. oldestActiveConnection.getRealConnection().rollback();
36. } **catch** (SQLException e) {
37. log.debug("Bad connection. Could not roll back");
38. }
39. }
40. conn = **new** PooledConnection(oldestActiveConnection.getRealConnection(), **this**);
41. oldestActiveConnection.invalidate();
42. **if** (log.isDebugEnabled()) {
43. log.debug("Claimed overdue connection " + conn.getRealHashCode() + ".");
44. }
45. } **else** {
46. // Must wait
47. **try** {
48. **if** (!countedWait) {
49. state.hadToWaitCount++;
50. countedWait = **true**;
51. }
52. **if** (log.isDebugEnabled()) {
53. log.debug("Waiting as long as " + poolTimeToWait + " milliseconds for connection.");
54. }
55. **long** wt = System.currentTimeMillis();
56. state.wait(poolTimeToWait);
57. state.accumulatedWaitTime += System.currentTimeMillis() - wt;
58. } **catch** (InterruptedException e) {
59. **break**;
60. }
61. }
62. }
63. }
64. **if** (conn != **null**) {
65. **if** (conn.isValid()) {
66. **if** (!conn.getRealConnection().getAutoCommit()) {
67. conn.getRealConnection().rollback();
68. }
69. conn.setConnectionTypeCode(assembleConnectionTypeCode(dataSource.getUrl(), username, password));
70. conn.setCheckoutTimestamp(System.currentTimeMillis());
71. conn.setLastUsedTimestamp(System.currentTimeMillis());
72. state.activeConnections.add(conn);
73. state.requestCount++;
74. state.accumulatedRequestTime += System.currentTimeMillis() - t;
75. } **else** {
76. **if** (log.isDebugEnabled()) {
77. log.debug("A bad connection (" + conn.getRealHashCode() + ") was returned from the pool, getting another connection.");
78. }
79. state.badConnectionCount++;
80. localBadConnectionCount++;
81. conn = **null**;
82. **if** (localBadConnectionCount > (poolMaximumIdleConnections + 3)) {
83. **if** (log.isDebugEnabled()) {
84. log.debug("PooledDataSource: Could not get a good connection to the database.");
85. }
86. **throw** **new** SQLException("PooledDataSource: Could not get a good connection to the database.");
87. }
88. }
89. }
90. }
92. }
94. **if** (conn == **null**) {
95. **if** (log.isDebugEnabled()) {
96. log.debug("PooledDataSource: Unknown severe error condition.  The connection pool returned a null connection.");
97. }
98. **throw** **new** SQLException("PooledDataSource: Unknown severe error condition.  The connection pool returned a null connection.");
99. }
101. **return** conn;
102. }

我们看下上面的方法都做了什么：  
1.  先看是否有空闲(idle)状态下的PooledConnection对象，如果有，就直接返回一个可用的PooledConnection对象；否则进行第2步。  
2. 查看活动状态的PooledConnection池activeConnections是否已满；如果没有满，则创建一个新的PooledConnection对象，然后放到activeConnections池中，然后返回此PooledConnection对象；否则进行第三步；  
3.  看最先进入activeConnections池中的PooledConnection对象是否已经过期：如果已经过期，从activeConnections池中移除此对象，然后创建一个新的PooledConnection对象，添加到activeConnections中，然后将此对象返回；否则进行第4步；  
4.  线程等待，循环2步。

流程图如下所示：



当我们拿到数据库连接PooledConnection后，我们在使用完之后一般来说就要关闭这个数据库连接，但是，对于池化来说，我们关闭了一个数据库连接并不是真正意义上想关闭这个连接，而是想把它放回到数据库连接池中。

怎么实现呢？mybatis中使用了代理模式有效的解决了该问题。就是返回给外部使用的数据库连接其实是一个代理对象（通过调用getProxyConnection()返回的对象）。这个代理对象时在真实数据库连接创建的时候被创建的，如下所示：

**[java]** [view plain](https://blog.csdn.net/majinggogogo/article/details/71715846) [copy](https://blog.csdn.net/majinggogogo/article/details/71715846)

1. **public** PooledConnection(Connection connection, PooledDataSource dataSource) {
2. **this**.hashCode = connection.hashCode();
3. **this**.realConnection = connection;
4. **this**.dataSource = dataSource;
5. **this**.createdTimestamp = System.currentTimeMillis();
6. **this**.lastUsedTimestamp = System.currentTimeMillis();
7. **this**.valid = **true**;
8. **this**.proxyConnection = (Connection) Proxy.newProxyInstance(Connection.**class**.getClassLoader(), IFACES, **this**);
9. }

而在调用这个代理对象的各个方法的时候，都是通过反射的方式，从invoke()方法进入，我们来看看：

**[java]** [view plain](https://blog.csdn.net/majinggogogo/article/details/71715846) [copy](https://blog.csdn.net/majinggogogo/article/details/71715846)

1. @Override
2. **public** Object invoke(Object proxy, Method method, Object[] args) **throws** Throwable {
3. String methodName = method.getName();
4. **if** (CLOSE.hashCode() == methodName.hashCode() && CLOSE.equals(methodName)) {
5. dataSource.pushConnection(**this**);
6. **return** **null**;
7. } **else** {
8. **try** {
9. **if** (!Object.**class**.equals(method.getDeclaringClass())) {
10. // issue #579 toString() should never fail
11. // throw an SQLException instead of a Runtime
12. checkConnection();
13. }
14. **return** method.invoke(realConnection, args);
15. } **catch** (Throwable t) {
16. **throw** ExceptionUtil.unwrapThrowable(t);
17. }
18. }
19. }

**[java]** [view plain](https://blog.csdn.net/majinggogogo/article/details/71715846) [copy](https://blog.csdn.net/majinggogogo/article/details/71715846)

1. **private** **static** **final** String CLOSE = "close";

我们可以看到，这里做了一个特殊处理，那就是判断调用的方法名是否是close()方法，如果是的话，就调用数据源对象的pushConnection()方法将数据库连接放回到连接池中，如下所示：

**[java]** [view plain](https://blog.csdn.net/majinggogogo/article/details/71715846) [copy](https://blog.csdn.net/majinggogogo/article/details/71715846)

1. **protected** **void** pushConnection(PooledConnection conn) **throws** SQLException {
3. **synchronized** (state) {
4. state.activeConnections.remove(conn);
5. **if** (conn.isValid()) {
6. **if** (state.idleConnections.size() < poolMaximumIdleConnections && conn.getConnectionTypeCode() == expectedConnectionTypeCode) {
7. state.accumulatedCheckoutTime += conn.getCheckoutTime();
8. **if** (!conn.getRealConnection().getAutoCommit()) {
9. conn.getRealConnection().rollback();
10. }
11. PooledConnection newConn = **new** PooledConnection(conn.getRealConnection(), **this**);
12. state.idleConnections.add(newConn);
13. newConn.setCreatedTimestamp(conn.getCreatedTimestamp());
14. newConn.setLastUsedTimestamp(conn.getLastUsedTimestamp());
15. conn.invalidate();
16. **if** (log.isDebugEnabled()) {
17. log.debug("Returned connection " + newConn.getRealHashCode() + " to pool.");
18. }
19. state.notifyAll();
20. } **else** {
21. state.accumulatedCheckoutTime += conn.getCheckoutTime();
22. **if** (!conn.getRealConnection().getAutoCommit()) {
23. conn.getRealConnection().rollback();
24. }
25. conn.getRealConnection().close();
26. **if** (log.isDebugEnabled()) {
27. log.debug("Closed connection " + conn.getRealHashCode() + ".");
28. }
29. conn.invalidate();
30. }
31. } **else** {
32. **if** (log.isDebugEnabled()) {
33. log.debug("A bad connection (" + conn.getRealHashCode() + ") attempted to return to the pool, discarding connection.");
34. }
35. state.badConnectionCount++;
36. }
37. }
38. }

简单的说下上面这个方法的逻辑：

1. 首先将当前数据库连接从活动数据库连接集合activeConnections中移除；

2. 判断当前数据库连接是否有效，如果无效，则跳转到第4步；如果有效，则继续下面的判断；

3. 判断当前idleConnections集合中的闲置数据库连接数量是否没超过设置的阈值且是当前数据库连接池的创建出来的链接，如果是，则将该数据库连接放回到idleConnections集合中并且通知在此据库连接池上等待的请求对象线程，如果不是，则将数据库连接关闭；

4. 将连接池中的坏数据库连接数+1，并返回；

（6）JNDI类型的数据源工厂JndiDataSourceFactory

对于JNDI类型的数据源的获取比较简单，mybatis中定义了一个JndiDataSourceFactory类用来创建通过JNDI形式创建的数据源。这个类源码如下：

**[java]** [view plain](https://blog.csdn.net/majinggogogo/article/details/71715846) [copy](https://blog.csdn.net/majinggogogo/article/details/71715846)

1. **public** **class** JndiDataSourceFactory **implements** DataSourceFactory {
3. **public** **static** **final** String INITIAL\_CONTEXT = "initial\_context";
4. **public** **static** **final** String DATA\_SOURCE = "data\_source";
5. **public** **static** **final** String ENV\_PREFIX = "env.";
7. **private** DataSource dataSource;
9. @Override
10. **public** **void** setProperties(Properties properties) {
11. **try** {
12. InitialContext initCtx;
13. Properties env = getEnvProperties(properties);
14. **if** (env == **null**) {
15. initCtx = **new** InitialContext();
16. } **else** {
17. initCtx = **new** InitialContext(env);
18. }
20. **if** (properties.containsKey(INITIAL\_CONTEXT)
21. && properties.containsKey(DATA\_SOURCE)) {
22. Context ctx = (Context) initCtx.lookup(properties.getProperty(INITIAL\_CONTEXT));
23. dataSource = (DataSource) ctx.lookup(properties.getProperty(DATA\_SOURCE));
24. } **else** **if** (properties.containsKey(DATA\_SOURCE)) {
25. dataSource = (DataSource) initCtx.lookup(properties.getProperty(DATA\_SOURCE));
26. }
28. } **catch** (NamingException e) {
29. **throw** **new** DataSourceException("There was an error configuring JndiDataSourceTransactionPool. Cause: " + e, e);
30. }
31. }
33. @Override
34. **public** DataSource getDataSource() {
35. **return** dataSource;
36. }
38. **private** **static** Properties getEnvProperties(Properties allProps) {
39. **final** String PREFIX = ENV\_PREFIX;
40. Properties contextProperties = **null**;
41. **for** (Entry<Object, Object> entry : allProps.entrySet()) {
42. String key = (String) entry.getKey();
43. String value = (String) entry.getValue();
44. **if** (key.startsWith(PREFIX)) {
45. **if** (contextProperties == **null**) {
46. contextProperties = **new** Properties();
47. }
48. contextProperties.put(key.substring(PREFIX.length()), value);
49. }
50. }
51. **return** contextProperties;
52. }
54. }

因为这块没看明白，对JNDI不是太熟悉，所以这块就不解释了，回头对这块了解了之后再行补充说明。如果有懂的朋友，也可以留言说明一下，谢谢。

如果想了解我最新的博文，请关注我的博客，谢谢。如果想看到更多相关技术文章并愿意支持我继续写下去，欢迎给我打赏，对您的打赏我表示感谢。吐舌头

