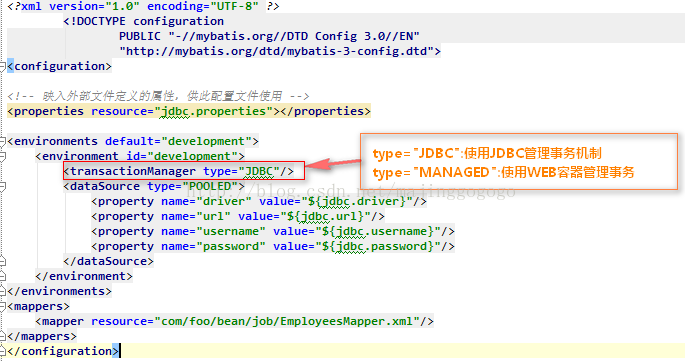
**Mybatis中的事务管理器详述**

<https://blog.csdn.net/majinggogogo/article/details/72026693>

在上篇文章<[Mybatis中的数据源与连接池详解](http://blog.csdn.net/majinggogogo/article/details/71715846)>中，我们结合源码对mybatis中的数据源和连接池进行了比较详细的说明。在这篇文章中，我们讲讲相关的另外一个主题——事务管理器。

在前面的文章中，我们知道mybatis支持两种事务类型，分别为JdbcTransaction和ManagedTransaction。接下来，我们从mybatis的xml配置文件入手，讲解事务管理器工厂的创建，然后讲述事务的创建和使用，最后分析这两种事务的实现和两者的区别。

我们先看看配置文件中相关的配置：



Mybatis定义了一个事务类型接口Transaction，JdbcTransaction和ManagedTransaction两种事务类型都实现了Transaction接口。我们看看Transaction这个接口的定义：

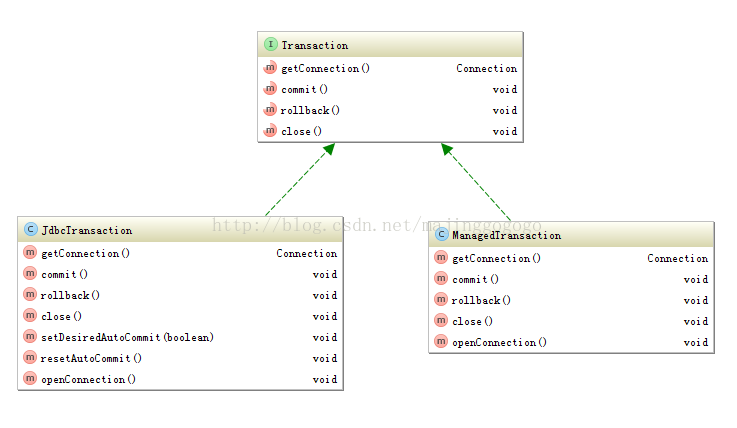
**[java]** [view plain](https://blog.csdn.net/majinggogogo/article/details/72026693) [copy](https://blog.csdn.net/majinggogogo/article/details/72026693)

1. <span style="font-family:'KaiTi\_GB2312';font-size:18px;">**public** **interface** Transaction {
3. /\*\*
4. \* Retrieve inner database connection
5. \* @return DataBase connection
6. \* @throws SQLException
7. \*/
8. Connection getConnection() **throws** SQLException;
10. /\*\*
11. \* Commit inner database connection.
12. \* @throws SQLException
13. \*/
14. **void** commit() **throws** SQLException;
16. /\*\*
17. \* Rollback inner database connection.
18. \* @throws SQLException
19. \*/
20. **void** rollback() **throws** SQLException;
22. /\*\*
23. \* Close inner database connection.
24. \* @throws SQLException
25. \*/
26. **void** close() **throws** SQLException;
28. /\*\*
29. \* Get transaction timeout if set
30. \* @throws SQLException
31. \*/
32. Integer getTimeout() **throws** SQLException;
34. }</span>

在事务接口中，定义了若干方法，如下结构所示：



事务的继承关系如下：



JdbcTransaction和ManagedTransaction的区别如下：

JdbcTransaction：利用java.sql.Connection对象完成对事务的提交（commit()）、回滚（rollback()）、关闭（close()）等；

ManagedTransaction：MyBatis自身不会去实现事务管理，而是让程序的容器来实现对事务的管理；

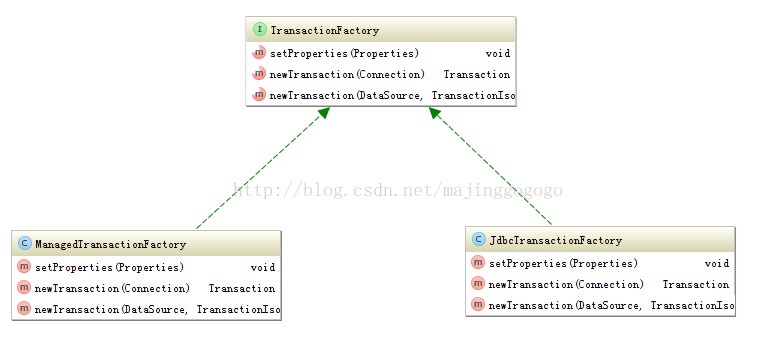
那么在mybatis中又是怎么使用事务管理器的呢？首先需要根据xml中的配置确定需要创建什么样的事务管理器，然后从事务管理器中获取相应的事务。

在mybatis初始化的时候，在解析<transactionManager>节点的时候，根据设置的type类型去初始化相应的事务管理器，解析源码如下所示：

**[java]** [view plain](https://blog.csdn.net/majinggogogo/article/details/72026693) [copy](https://blog.csdn.net/majinggogogo/article/details/72026693)

1. <span style="font-family:'KaiTi\_GB2312';font-size:18px;"> /\*\*
2. \* 解析<transactionManager>节点，创建对应的TransactionFactory
3. \* @param context
4. \* @return
5. \* @throws Exception
6. \*/
7. **private** TransactionFactory transactionManagerElement(XNode context) **throws** Exception {
8. **if** (context != **null**) {
9. String type = context.getStringAttribute("type");
10. Properties props = context.getChildrenAsProperties();
11. /\*
12. 在Configuration初始化的时候，会通过以下语句，给JDBC和MANAGED对应的工厂类
13. typeAliasRegistry.registerAlias("JDBC", JdbcTransactionFactory.class);
14. typeAliasRegistry.registerAlias("MANAGED", ManagedTransactionFactory.class);
15. 下述的resolveClass(type).newInstance()会创建对应的工厂实例
16. \*/
17. TransactionFactory factory = (TransactionFactory) resolveClass(type).newInstance();
18. factory.setProperties(props);
19. **return** factory;
20. }
21. **throw** **new** BuilderException("Environment declaration requires a TransactionFactory.");
22. }  </span>

从代码可以看出来，如果type配置成JDBC，则创建一个JdbcTransactionFactory实例，如果type配置成MANAGED，则会创建一个ManagedTransactionFactory实例。这两个事务管理器类型都实现了mybatis定义的TransactionFactory接口。



事务管理器工厂接口的定义如下所示：

**[java]** [view plain](https://blog.csdn.net/majinggogogo/article/details/72026693) [copy](https://blog.csdn.net/majinggogogo/article/details/72026693)

1. <span style="font-family:'KaiTi\_GB2312';font-size:18px;">**public** **interface** TransactionFactory {
3. /\*\*
4. \* Sets transaction factory custom properties.
5. \* @param props
6. \*/
7. **void** setProperties(Properties props);
9. /\*\*
10. \* Creates a {@link Transaction} out of an existing connection.
11. \* @param conn Existing database connection
12. \* @return Transaction
13. \* @since 3.1.0
14. \*/
15. Transaction newTransaction(Connection conn);
17. /\*\*
18. \* Creates a {@link Transaction} out of a datasource.
19. \* @param dataSource DataSource to take the connection from
20. \* @param level Desired isolation level
21. \* @param autoCommit Desired autocommit
22. \* @return Transaction
23. \* @since 3.1.0
24. \*/
25. Transaction newTransaction(DataSource dataSource, TransactionIsolationLevel level, **boolean** autoCommit);
27. }</span>

从接口定义看，不管是JdbcTransactionFactory，还是ManagedTransactionFactory，都需要自行实现事务Transaction的创建工作。我们从源码上看看这两个类都是怎么定义实现的。

JdbcTransactionFactory定义如下：

**[java]** [view plain](https://blog.csdn.net/majinggogogo/article/details/72026693) [copy](https://blog.csdn.net/majinggogogo/article/details/72026693)

1. <span style="font-family:'KaiTi\_GB2312';font-size:18px;">**public** **class** JdbcTransactionFactory **implements** TransactionFactory {
3. @Override
4. **public** **void** setProperties(Properties props) {
5. }
7. @Override
8. **public** Transaction newTransaction(Connection conn) {
9. **return** **new** JdbcTransaction(conn);
10. }
12. @Override
13. **public** Transaction newTransaction(DataSource ds, TransactionIsolationLevel level, **boolean** autoCommit) {
14. **return** **new** JdbcTransaction(ds, level, autoCommit);
15. }
16. }</span>

ManagedTransactionFactory定义如下：

**[java]** [view plain](https://blog.csdn.net/majinggogogo/article/details/72026693) [copy](https://blog.csdn.net/majinggogogo/article/details/72026693)

1. <span style="font-family:'KaiTi\_GB2312';font-size:18px;">**public** **class** ManagedTransactionFactory **implements** TransactionFactory {
3. **private** **boolean** closeConnection = **true**;
5. @Override
6. **public** **void** setProperties(Properties props) {
7. **if** (props != **null**) {
8. String closeConnectionProperty = props.getProperty("closeConnection");
9. **if** (closeConnectionProperty != **null**) {
10. closeConnection = Boolean.valueOf(closeConnectionProperty);
11. }
12. }
13. }
15. @Override
16. **public** Transaction newTransaction(Connection conn) {
17. **return** **new** ManagedTransaction(conn, closeConnection);
18. }
20. @Override
21. **public** Transaction newTransaction(DataSource ds, TransactionIsolationLevel level, **boolean** autoCommit) {
22. // Silently ignores autocommit and isolation level, as managed transactions are entirely
23. // controlled by an external manager.  It's silently ignored so that
24. // code remains portable between managed and unmanaged configurations.
25. **return** **new** ManagedTransaction(ds, level, closeConnection);
26. }
27. }</span>

从源码看，JdbcTransactionFactory会创建JDBC类型的事务JdbcTransaction，而ManagedTransactionFactory则会创建ManagedTransaction。

接下来，我们再来看看这两种类型的事务类型。

先从JdbcTransaction看起吧，这种类型的事务就是使用Java自带的Connection来实现事务的管理。connection对象的获取被延迟到调用getConnection()方法。如果autocommit设置为on，开启状态的话，它会忽略commit和rollback。直观地讲，就是JdbcTransaction是使用的java.sql.Connection 上的commit和rollback功能，JdbcTransaction只是相当于对java.sql.Connection事务处理进行了一次包装（wrapper），Transaction的事务管理都是通过java.sql.Connection实现的。

JdbcTransaction的代码实现如下：

**[java]** [view plain](https://blog.csdn.net/majinggogogo/article/details/72026693) [copy](https://blog.csdn.net/majinggogogo/article/details/72026693)

1. <span style="font-family:'KaiTi\_GB2312';font-size:18px;">**public** **class** JdbcTransaction **implements** Transaction {
3. **private** **static** **final** Log log = LogFactory.getLog(JdbcTransaction.**class**);
5. **protected** Connection connection;
6. **protected** DataSource dataSource;
7. **protected** TransactionIsolationLevel level;
8. **protected** **boolean** autoCommmit;
10. **public** JdbcTransaction(DataSource ds, TransactionIsolationLevel desiredLevel, **boolean** desiredAutoCommit) {
11. dataSource = ds;
12. level = desiredLevel;
13. autoCommmit = desiredAutoCommit;
14. }
16. **public** JdbcTransaction(Connection connection) {
17. **this**.connection = connection;
18. }
20. @Override
21. **public** Connection getConnection() **throws** SQLException {
22. **if** (connection == **null**) {
23. openConnection();
24. }
25. **return** connection;
26. }
28. @Override
29. **public** **void** commit() **throws** SQLException {
30. **if** (connection != **null** && !connection.getAutoCommit()) {
31. **if** (log.isDebugEnabled()) {
32. log.debug("Committing JDBC Connection [" + connection + "]");
33. }
34. connection.commit();
35. }
36. }
38. @Override
39. **public** **void** rollback() **throws** SQLException {
40. **if** (connection != **null** && !connection.getAutoCommit()) {
41. **if** (log.isDebugEnabled()) {
42. log.debug("Rolling back JDBC Connection [" + connection + "]");
43. }
44. connection.rollback();
45. }
46. }
48. @Override
49. **public** **void** close() **throws** SQLException {
50. **if** (connection != **null**) {
51. resetAutoCommit();
52. **if** (log.isDebugEnabled()) {
53. log.debug("Closing JDBC Connection [" + connection + "]");
54. }
55. connection.close();
56. }
57. }
59. **protected** **void** setDesiredAutoCommit(**boolean** desiredAutoCommit) {
60. **try** {
61. **if** (connection.getAutoCommit() != desiredAutoCommit) {
62. **if** (log.isDebugEnabled()) {
63. log.debug("Setting autocommit to " + desiredAutoCommit + " on JDBC Connection [" + connection + "]");
64. }
65. connection.setAutoCommit(desiredAutoCommit);
66. }
67. } **catch** (SQLException e) {
68. // Only a very poorly implemented driver would fail here,
69. // and there's not much we can do about that.
70. **throw** **new** TransactionException("Error configuring AutoCommit.  "
71. + "Your driver may not support getAutoCommit() or setAutoCommit(). "
72. + "Requested setting: " + desiredAutoCommit + ".  Cause: " + e, e);
73. }
74. }
76. **protected** **void** resetAutoCommit() {
77. **try** {
78. **if** (!connection.getAutoCommit()) {
79. // MyBatis does not call commit/rollback on a connection if just selects were performed.
80. // Some databases start transactions with select statements
81. // and they mandate a commit/rollback before closing the connection.
82. // A workaround is setting the autocommit to true before closing the connection.
83. // Sybase throws an exception here.
84. **if** (log.isDebugEnabled()) {
85. log.debug("Resetting autocommit to true on JDBC Connection [" + connection + "]");
86. }
87. connection.setAutoCommit(**true**);
88. }
89. } **catch** (SQLException e) {
90. **if** (log.isDebugEnabled()) {
91. log.debug("Error resetting autocommit to true "
92. + "before closing the connection.  Cause: " + e);
93. }
94. }
95. }
97. **protected** **void** openConnection() **throws** SQLException {
98. **if** (log.isDebugEnabled()) {
99. log.debug("Opening JDBC Connection");
100. }
101. connection = dataSource.getConnection();
102. **if** (level != **null**) {
103. connection.setTransactionIsolation(level.getLevel());
104. }
105. setDesiredAutoCommit(autoCommmit);
106. }
108. @Override
109. **public** Integer getTimeout() **throws** SQLException {
110. **return** **null**;
111. }
113. }</span>

最后我们再来看看ManagedTransaction对象，这个对象因为是将事务的管理交给容器去控制，所以，这里的ManagedTransaction是没有做任何控制的。我们先来看看源码：

**[java]** [view plain](https://blog.csdn.net/majinggogogo/article/details/72026693) [copy](https://blog.csdn.net/majinggogogo/article/details/72026693)

1. <span style="font-family:'KaiTi\_GB2312';font-size:18px;">**public** **class** ManagedTransaction **implements** Transaction {
3. **private** **static** **final** Log log = LogFactory.getLog(ManagedTransaction.**class**);
5. **private** DataSource dataSource;
6. **private** TransactionIsolationLevel level;
7. **private** Connection connection;
8. **private** **boolean** closeConnection;
10. **public** ManagedTransaction(Connection connection, **boolean** closeConnection) {
11. **this**.connection = connection;
12. **this**.closeConnection = closeConnection;
13. }
15. **public** ManagedTransaction(DataSource ds, TransactionIsolationLevel level, **boolean** closeConnection) {
16. **this**.dataSource = ds;
17. **this**.level = level;
18. **this**.closeConnection = closeConnection;
19. }
21. @Override
22. **public** Connection getConnection() **throws** SQLException {
23. **if** (**this**.connection == **null**) {
24. openConnection();
25. }
26. **return** **this**.connection;
27. }
29. @Override
30. **public** **void** commit() **throws** SQLException {
31. // Does nothing
32. }
34. @Override
35. **public** **void** rollback() **throws** SQLException {
36. // Does nothing
37. }
39. @Override
40. **public** **void** close() **throws** SQLException {
41. **if** (**this**.closeConnection && **this**.connection != **null**) {
42. **if** (log.isDebugEnabled()) {
43. log.debug("Closing JDBC Connection [" + **this**.connection + "]");
44. }
45. **this**.connection.close();
46. }
47. }
49. **protected** **void** openConnection() **throws** SQLException {
50. **if** (log.isDebugEnabled()) {
51. log.debug("Opening JDBC Connection");
52. }
53. **this**.connection = **this**.dataSource.getConnection();
54. **if** (**this**.level != **null**) {
55. **this**.connection.setTransactionIsolation(**this**.level.getLevel());
56. }
57. }
59. @Override
60. **public** Integer getTimeout() **throws** SQLException {
61. **return** **null**;
62. }
64. }</span>

从源码我们可以看到，ManagedTransaction的commit和rollback方法是没有做任何事情的，它将事务交由了更上层的容易来进行控制和实现。至此，关于事务管理器我们描述的已经差不多了，如果需要深究可以自己再去研究研究。

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

