[关于MyBatis sqlSession的一点整理](http://www.xx566.com/detail/186.html)

    工作中，需要学习一下MyBatis sqlSession的产生过程，翻看了mybatis-spring的源码，阅读了一些mybatis的相关doc，对mybatis sqlSession有了一些认知和理解，这里简单的总结和整理一下。

    首先， 通过翻阅源码，我们来整理一下mybatis进行持久化操作时重要的几个类：

* SqlSessionFactoryBuilder：build方法创建SqlSessionFactory实例。
* SqlSessionFactory：创建SqlSession实例的工厂。
* SqlSession：用于执行持久化操作的对象，类似于jdbc中的Connection。
* SqlSessionTemplate：MyBatis提供的持久层访问模板化的工具，线程安全，可通过构造参数或依赖注入SqlSessionFactory实例。

    Hibernate是与MyBatis类似的orm框架，这里与Hibernate进行一下对比，Hibernate中对于connection的管理，是通过以下几个重要的类：

* SessionFactory：创建Session实例的工厂，类似于MyBatis中的SqlSessionFactory。
* Session：用来执行持久化操作的对象，类似于jdbc中的Connection。
* HibernateTemplate：Hibernate提供的持久层访问模板化的工具，线程安全，可通过构造参数或依赖注入SessionFactory实例。

    在日常的开发中，我们经常需要这样对MyBatis和Spring进行集成，把sqlSessionFactory交给Spring管理，通常情况下，我们这样配置：

<bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">

    <property name="dataSource" ref="dataSource" />

</bean>

    通过上面的配置，Spring将自动创建一个SqlSessionFactory对象，其中使用到了org.mybatis.spring.SqlSessionFactoryBean，其 是MyBatis为Spring提供的用于创建SqlSessionFactory的类，将在Spring应用程序的上下文建议一下可共享的 MyBatis SqlSessionFactory实例，我们可以通过依赖注入将SqlSessionFactory传递给MyBatis的一些接口。

    如果通过Spring进行事务的管理，我们需要增加Spring注解的事务管理机制，如下配置：

<bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">

    <property name="dataSource" ref="dataSource" />

</bean>

<tx:annotation-driven/>

    这样，我们就可以使用Spring @Transactional注解，进行事务的控制，表明所注释的方法应该在一个事务中运行。 Spring将在事务成功完成后提交事务，在事务发生错误时进行异常回滚，而且，Spring会将产生的MyBatis异常转换成适当的 DataAccessExceptions，从而提供具体的异常信息。

    下面，我们通过分析SqlSessionUtils中getSession的源码，来详细的了解一下sqlSession的产生过程，源码如下：

public static SqlSession getSqlSession(SqlSessionFactory sessionFactory, ExecutorType executorType, PersistenceExceptionTranslator exceptionTranslator) {

  notNull(sessionFactory, "No SqlSessionFactory specified");

  notNull(executorType, "No ExecutorType specified");

  SqlSessionHolder holder = (SqlSessionHolder) getResource(sessionFactory);

  if (holder != null && holder.isSynchronizedWithTransaction()) {

    if (holder.getExecutorType() != executorType) {

      throw new TransientDataAccessResourceException("Cannot change the ExecutorType when there is an existing transaction");

    }

    holder.requested();

    if (logger.isDebugEnabled()) {

      logger.debug("Fetched SqlSession [" + holder.getSqlSession() + "] from current transaction");

    }

    return holder.getSqlSession();

  }

  if (logger.isDebugEnabled()) {

    logger.debug("Creating a new SqlSession");

  }

  SqlSession session = sessionFactory.openSession(executorType);

  // Register session holder if synchronization is active (i.e. a Spring TX is active)

  //

  // Note: The DataSource used by the Environment should be synchronized with the

  // transaction either through DataSourceTxMgr or another tx synchronization.

  // Further assume that if an exception is thrown, whatever started the transaction will

  // handle closing / rolling back the Connection associated with the SqlSession.

  if (isSynchronizationActive()) {

    Environment environment = sessionFactory.getConfiguration().getEnvironment();

    if (environment.getTransactionFactory() instanceof SpringManagedTransactionFactory) {

      if (logger.isDebugEnabled()) {

        logger.debug("Registering transaction synchronization for SqlSession [" + session + "]");

      }

      holder = new SqlSessionHolder(session, executorType, exceptionTranslator);

      bindResource(sessionFactory, holder);

      registerSynchronization(new SqlSessionSynchronization(holder, sessionFactory));

      holder.setSynchronizedWithTransaction(true);

      holder.requested();

    } else {

      if (getResource(environment.getDataSource()) == null) {

        if (logger.isDebugEnabled()) {

          logger.debug("SqlSession [" + session + "] was not registered for synchronization because DataSource is not transactional");

        }

      } else {

        throw new TransientDataAccessResourceException(

            "SqlSessionFactory must be using a SpringManagedTransactionFactory in order to use Spring transaction synchronization");

      }

    }

  } else {

    if (logger.isDebugEnabled()) {

      logger.debug("SqlSession [" + session + "] was not registered for synchronization because synchronization is not active");

    }

  }

  return session;

}

    上面的getSession方法，会从Spring的事务管理器中获取一个SqlSession或创建一个新的SqlSession，将试图从当前事务中得到一个SqlSession，然后，如果配置有事务管理器的工厂并且Spring 的事务管理器是活跃的，它将会锁定当前事务的SqlSession，保证同步。主要是通过以下几个步骤进行SqlSession的创建：

1. 它会首先获取SqlSessionHolder，SqlSessionHolder用于在TransactionSynchronizationManager中保持当前的SqlSession。
2. 如果holder不为空，并且holder被事务锁定，则可以通过holder.getSqlSession()方法，从当前事务中获取sqlSession，即 Fetched SqlSession from current transaction。
3. 如果不存在holder或没有被事务锁定，则会创建新的sqlSession，即 Creating a new SqlSession，通过sessionFactory.openSession()方法。
4. 如果当前线程的事务是活跃的，将会为SqlSession注册事务同步，即 Registering transaction synchronization for SqlSession。