Mybatis

1. 入门

1.1 Maven配置

```
<dependencies>
    <!-- 添加junit -->
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.11</version>
        <scope>test</scope>
    </dependency>

    <!-- 添加log4j -->
    <dependency>
        <groupId>log4j</groupId>
        <artifactId>log4j</artifactId>
        <version>1.2.16</version>
    </dependency>

    <!-- 添加mybatis -->
    <dependency>
        <groupId>org.mybatis</groupId>
        <artifactId>mybatis</artifactId>
        <version>3.2.6</version>
    </dependency>

    <!-- 添加mysql驱动 -->
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>5.1.12</version>
    </dependency>
</dependencies>
```

1.2 configuration.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration
    PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>

    <!-- 指定properties配置文件， 我这里面配置的是数据库相关 -->
    <properties resource="dbConfig.properties"></properties>

    <!-- 指定Mybatis使用log4j -->
    <settings>
        <setting name="logImpl" value="LOG4J"/>
    </settings>

    <environments default="development">
        <environment id="development">
            <transactionManager type="JDBC"/>
            <dataSource type="POOLED">
                <!--
                 如果上面没有指定数据库配置的properties文件，那么此处可以这样直接配置
                <property name="driver" value="com.mysql.jdbc.Driver"/>
                <property name="url" value="jdbc:mysql://localhost:3306/test1"/>
                <property name="username" value="root"/>
```

```
                <property name="password" value="root"/>
                -->

                <!-- 上面指定了数据库配置文件， 配置文件里面也是对应的这四个属性 -->
                <property name="driver" value="${driver}"/>
                <property name="url" value="${url}"/>
                <property name="username" value="${username}"/>
                <property name="password" value="${password}"/>
            </dataSource>
        </environment>
    </environments>

    <!-- 映射文件，mybatis精髓， 后面才会细讲 -->
    <mappers>
        <mapper resource="com/dy/dao/userDao-mapping.xml"/>
    </mappers>
</configuration>
```

2. 配置

> MyBatis在初始化的时候，会将MyBatis的配置信息全部加载到内存中，使用`org.apache.ibatis.session.Configuration`实例来维护。使用者可以使用`sqlSession.getConfiguration()`方法来获取。

2.1 事务的配置

2.1.1 事务管理形式：

1. 使用**JDBC**的事务管理机制：即利用java.sql.Connection对象完成对事务的提交（commit()）、回滚（rollback()）、关闭（close()）等。
2. 使用**MANAGED**的事务管理机制：这种机制MyBatis自身不会去实现事务管理，而是让程序的**容器**如（JBOSS，Weblogic）来实现对事务的管理。

> 注意：`如果我们使用MyBatis构建本地程序，即不是WEB程序`，若将type设置成"MANAGED"，那么，我们执行的任何update操作，`即使我们最后执行了commit操作，数据也不会保留，不会对数据库造成任何影响`。因为我们将MyBatis配置成了"MANAGED"，`即MyBatis自己不管理事务，而我们又是运行的本地程序，没有事务管理功能`，所以对数据库的update操作都是无效的。

2.1.2 配置：

```
<transactionManager type="JDBC/MANAGED">
```

2.2 properties

```
<!-- 方式一 -->
<properties url="url/database.properties"/>
<!-- 方式二 -->
<properties resource="database.properties"/> <!-- resource路径下 -->
<!-- 方式三 -->
<properties>
    <property name="driver" value="com.mysql.jdbc.Driver"/>
        <property name="url" value="jdbc:mysql://localhost:3306/test1"/>
        <property name="username" value="root"/>
        <property name="password" value="root"/>
</properties>
```

至于优先级问题，自己看源码：

```
if (context != null) {
      Properties defaults = context.getChildrenAsProperties(); // mybatis默认配置
      String resource = context.getStringAttribute("resource"); // resource引进的属性
      String url = context.getStringAttribute("url"); // url引进的属性
      if (resource != null && url != null) {
        throw new BuilderException("The properties element cannot specify both a URL and a resource based
property file reference.  Please specify one or the other.");
      }
      if (resource != null) {
        defaults.putAll(Resources.getResourceAsProperties(resource));
      } else if (url != null) {
        defaults.putAll(Resources.getUrlAsProperties(url));
      }
      Properties vars = configuration.getVariables(); // <property>标签定义的属性
      if (vars != null) {
        defaults.putAll(vars);
      }
```

2.3 别名

```
<typeAliases>
        <!--
        通过package，可以直接指定package的名字， mybatis会自动扫描你指定包下面的javabean，
        并且默认设置一个别名，默认的名字为： javabean 的首字母小写的非限定类名来作为它的别名。
        也可在javabean 加上注解@Alias 来自定义别名， 例如： @Alias(user)
        <package name="com.dy.entity"/>
        -->
        <typeAlias alias="UserEntity" type="com.dy.entity.User"/>
</typeAliases>
```

关于默认注册别名，请查看`TypeAliasRegistry`类；

2.4 TypeHandler

```

<typeHandlers>
    <!--
          当配置package的时候，mybatis会去配置的package扫描TypeHandler
          <package name="com.dy.demo"/>
       -->
        <package name="package.path"></package>

    <!-- handler属性直接配置我们要指定的TypeHandler -->
    <typeHandler handler=""/>

    <!-- javaType 配置java类型，例如String， 如果配上javaType， 那么指定的typeHandler就只作用于指定的类型 -->
    <typeHandler javaType="" handler=""/>

    <!-- jdbcType 配置数据库基本数据类型，例如varchar， 如果配上jdbcType， 那么指定的typeHandler就只作用于指定的类型
-->
    <typeHandler jdbcType="" handler=""/>

    <!-- 也可两者都配置 -->
    <typeHandler javaType="" jdbcType="" handler=""/>
</typeHandlers>
```

关于mybatis默认的类型映射，请查看`TypeHandlerRegistry`类；

2.4.1 自定义TypeHandler

需要实现**BaseTypeHandler**接口：

```
@MappedJdbcTypes(JdbcType.VARCHAR)
//此处如果不用注解指定jdbcType，那么，就可以在配置文件中通过"jdbcType"属性指定， 同理， javaType 也可通过
@MappedTypes指定
public class ExampleTypeHandler extends BaseTypeHandler<String> {

  @Override
  public void setNonNullParameter(PreparedStatement ps, int i, String parameter, JdbcType jdbcType) throws
SQLException {
    ps.setString(i, parameter);
  }

  @Override
  public String getNullableResult(ResultSet rs, String columnName) throws SQLException {
    return rs.getString(columnName);
  }

  @Override
  public String getNullableResult(ResultSet rs, int columnIndex) throws SQLException {
    return rs.getString(columnIndex);
  }

  @Override
  public String getNullableResult(CallableStatement cs, int columnIndex) throws SQLException {
    return cs.getString(columnIndex);
  }
}
```

2.5 ObjectFactory

```
<objectFactory type="org.mybatis.example.ExampleObjectFactory">
        <property name="someProperty" value="100"/>
</objectFactory>
```

2.6 plugin

```
<configuration>
    ......
    <plugins>
      <plugin interceptor="org.mybatis.example.ExamplePlugin">
        <property name="someProperty" value="100"/>
      </plugin>
    </plugins>
    ......
</configuration>
```

2.7 mapper.xml文件

```
<configuration>
    ......
    <mappers>
        <!-- 第一种方式：通过resource指定 -->
        <mapper resource="com/dy/dao/userDao.xml"/>

        <!-- 第二种方式， 通过class指定接口，进而将接口与对应的xml文件形成映射关系
             不过，使用这种方式必须保证 接口与mapper文件同名(不区分大小写)，
             我这儿接口是UserDao,那么意味着mapper文件为UserDao.xml
        -->
        <mapper class="com.dy.dao.UserDao"/>

        <!-- 第三种方式，直接指定包，自动扫描，与方法二同理
        -->
        <package name="com.dy.dao"/>
```

```
        <!-- 第四种方式：通过url指定mapper文件位置
        -->
        <mapper url="file://........"/>
    </mappers>
    ......
</configuration>
```

2.8 settings

setting节点里配置的值会直接改写Configuration对应的变量值，这些变量描述的是Mybatis的全局运行方式；

```
<settings>
    <setting name="cacheEnabled" value="true"/>
    <setting name="lazyLoadingEnabled" value="true"/>
    <setting name="multipleResultSetsEnabled" value="true"/>
    <setting name="useColumnLabel" value="true"/>
    <setting name="useGeneratedKeys" value="false"/>
    <setting name="enhancementEnabled" value="false"/>
    <setting name="defaultExecutorType" value="SIMPLE"/>
    <setting name="defaultStatementTimeout" value="25000"/>
</settings>
```

3. Mapper.xml内容

3.1 cache-ref

> cache-ref引用别的cache。因为每个cache都以namespace为id，所以cache-ref只需要配置一个namespace属性就可以了。需要注意的是，如果cache-ref和cache都配置了，以cache为准。

```
<cache-ref namespace="com.someone.application.data.SomeMapper"/>
```

3.2 flushCache

默认配置为true，将其设置为 true，任何时候只要语句被调用，都会导致本地缓存和二级缓存都会被清空；

3.3 statementType

STATEMENT，PREPARED 或 CALLABLE 的一个。这会让 MyBatis 分别使用 Statement，PreparedStatement 或 CallableStatement，默认值：PREPARED。

3.4 keyProperty

（仅对 insert 和 update 有用）唯一标记一个属性，MyBatis 会通过 getGeneratedKeys 的返回值或者通过 insert 语句的 selectKey 子元素设置它的键值，默认：unset。如果希望得到多个生成的列，也可以是逗号分隔的属性名称列表。

```
<!-- 主要是在主键是自增的情况下，添加成功后可以直接使用主键值，其中keyProperty的值
        是对象的属性值不是数据库表中的字段名-->
    <insert id="saveMsg" parameterType="cn.com.hyddl.smarthome.notice
                                        .core.nano.Notice"
        useGeneratedKeys="true" keyProperty="msgId">
        insert into notice(msg_type,title,content,rec_time,send_time,user_id,
                                                                    deleted,viewed)
        values(#{msgType,jdbcType=INTEGER},
                        #{title,jdbcType=VARCHAR},
                        #{content,jdbcType=VARCHAR},
                #{recTime,jdbcType=BIGINT},
                        #{sendTime,jdbcType=BIGINT},
                        #{userId,jdbcType=VARCHAR},
                #{deleted,jdbcType=TINYINT},
```

```
                        #{viewed,jdbcType=INTEGER}
        )
    </insert>
```

3.5 selectKey

ibatis 插入数据将selectKey放在insert之后，通过LAST_INSERT_ID() 获得刚插入的自动增长的id的值。

mysql中：

```
<insert id="insertUser" parameterClass="ibatis.User">
    <!-- 至于statementType，与前面相同，MyBatis 支持 STATEMENT，PREPARED 和 CALLABLE
                        语句的映射类型，分别代表 PreparedStatement 和 CallableStatement 类型。 -->
          <selectKey resultClass="long" keyProperty="id" order="AFTER"
                     statementType="PREPARED">
             SELECT LAST_INSERT_ID() AS ID
          </selectKey>
          insert into user
          (name,password)
          values
          (#name#,#password#)
</insert>
```

oracle中：

```
<insert id="insertUser" parameterClass="ibatis.User">
     <selectKey resultClass="long" order="BEFORE" keyProperty="id">
            SELECT LAST_INSERT_ID() AS ID
        </selectKey>
          insert into user
          (name,password)
          values
          (#name#,#password#)
</insert>
```

3.6 keyColumn

仅对 insert 和 update 有用）通过生成的键值设置表中的列名，这个设置仅在某些数据库（像 PostgreSQL）是必须的，当主键列不是表中的第一列的时候需要设置。如果希望得到多个生成的列，也可以是逗号分隔的属性名称列表。

3.7 timeout

这个设置是在抛出异常之前，驱动程序等待数据库返回请求结果的秒数。默认值为 unset（依赖驱动）。 → timeout="20"；

3.8 useCache

将其设置为 true，将会导致本条语句的结果被二级缓存，默认值：对 select 元素为 true → useCache="true"；

4. 动态SQL

4.1 where

where可以自动处理开头和结尾多出来的 `AND` 和 `OR` ；

4.2 trim

```
<trim prefix="WHERE" prefixOverrides="AND |OR ">
    ...
</trim>
<!-- 其实，这何where标签的功能一样 -->
```

代码作用：当WHERE后紧随AND或则OR的时候，就去除AND或者OR。

4.3 set

```
<update id="updateUser" parameterType="com.dy.entity.User">
    update user
        <!-- set能够自动处理结尾多余的逗号 -->
        <set>
            <if test="name != null">name = #{name},</if>
            <if test="password != null">password = #{password},</if>
            <if test="age != null">age = #{age},</if>
        </set>
        <where>
            <if test="id != null">
                id = #{id}
            </if>
            and deleteFlag = 0;
        </where>
</update>
```

同样功能的trim标签：

```
<trim prefix="SET" suffixOverrides=",">
  ...
</trim>
```

4.4 foreach

当循环的对象为map的时候，index其实就是map的key。

5. SQL执行流程分析

6. 插件原理

6.1 Interceptor接口

示例：

```
@Intercepts({@Signature(type= Executor.class, method = "update", args =
{MappedStatement.class,Object.class})})
public class ExamplePlugin implements Interceptor {
        public Object intercept(Invocation invocation) throws Throwable {
        /* 最后一个语句一定是return invocation.proceed(), 不然拦截器链就断了 */
        return invocation.proceed();
        }
        public Object plugin(Object target) {
        return Plugin.wrap(target, this);
        }
        public void setProperties(Properties properties) {
        }
}
```

```
<plugins>
    <plugin interceptor="org.format.mybatis.cache.interceptor.ExamplePlugin">
        <property name="key" value="value"></property>
        <property name="key2" value="value2"></property>
    </plugin>
</plugins>
```

6.1.1 接口方法

> plugin方法用于某些处理器(Handler)的构建过程。interceptor方法用于处理代理类的执行。setProperties方法用于拦截器属性的设置。

> MyBatis官网提供的使用 @Interceptors和 @Signature注解以及Plugin类这样处理拦截器的方法，我们不一定要直接这样使用。我们也可以抛弃这3个类，直接在plugin方法内部根据target实例的类型做相应的操作。

6.2 可拦截对象

默认情况下，MyBatis允许使用插件来拦截的方法调用包括：

1. Executor (update, query, flushStatements, commit, rollback, getTransaction, close, isClosed) 拦截执行器的方法
2. ParameterHandler (getParameterObject, setParameters) 拦截参数的处理
3. ResultSetHandler (handleResultSets, handleOutputParameters) 拦截结果集的处理
4. StatementHandler (prepare, parameterize, batch, update, query) 拦截Sql语法构建的处理

6.3 原理

请查看博客

6.4 注意事项

1. 不编写不必要的插件；
2. 实现plugin方法时判断一下目标类型，是本插件要拦截的对象才执行Plugin.wrap方法，否者直接返回目标本省，这样可以减少目标被代理的次数。

7. 分页插件

> MyBatis自身提供的分页功能是通过RowBounds来实现的，它通过rowBounds.offset和rowBounds.limit来过滤查询出来的结果集，这种分页功能是基于查询结果的再过滤，而不是进行数据库的物理分页；

7.1 自定义插件

7.1.1 拦截器签名

```
@Intercepts({@Signature(type =StatementHandler.class, method = "prepare",
                        args ={Connection.class})})
public class PageInterceptor implements Interceptor {
    ...
}
```

7.1.2 实现intercept方法

```
public Object intercept(Invocation invocation) throws Throwable {
     StatementHandler statementHandler = (StatementHandler) invocation.getTarget();
     MetaObject metaStatementHandler = MetaObject.forObject(statementHandler,
     DEFAULT_OBJECT_FACTORY, DEFAULT_OBJECT_WRAPPER_FACTORY);
     // 分离代理对象链(由于目标类可能被多个拦截器拦截，从而形成多次代理，通过下面的两次循环
     // 可以分离出最原始的的目标类)
     while (metaStatementHandler.hasGetter("h")) {
         Object object = metaStatementHandler.getValue("h");
         metaStatementHandler = MetaObject.forObject(object, DEFAULT_OBJECT_FACTORY,
         DEFAULT_OBJECT_WRAPPER_FACTORY);
     }
     // 分离最后一个代理对象的目标类
     while (metaStatementHandler.hasGetter("target")) {
```

```
            Object object = metaStatementHandler.getValue("target");
            metaStatementHandler = MetaObject.forObject(object, DEFAULT_OBJECT_FACTORY,
            DEFAULT_OBJECT_WRAPPER_FACTORY);
        }
        Configuration configuration = (Configuration) metaStatementHandler.
        getValue("delegate.configuration");
        dialect = configuration.getVariables().getProperty("dialect");
        if (null == dialect || "".equals(dialect)) {
            logger.warn("Property dialect is not setted,use default 'mysql' ");
            dialect = defaultDialect;
        }
        pageSqlId = configuration.getVariables().getProperty("pageSqlId");
        if (null == pageSqlId || "".equals(pageSqlId)) {
            logger.warn("Property pageSqlId is not setted,use default '.*Page$' ");
            pageSqlId = defaultPageSqlId;
        }
        MappedStatement mappedStatement = (MappedStatement)
        metaStatementHandler.getValue("delegate.mappedStatement");
        // 只重写需要分页的sql语句。通过MappedStatement的ID匹配，默认重写以Page结尾的
        //  MappedStatement的sql
        if (mappedStatement.getId().matches(pageSqlId)) {
            BoundSql boundSql = (BoundSql) metaStatementHandler
                .getValue("delegate.boundSql");
            Object parameterObject = boundSql.getParameterObject();
            if (parameterObject == null) {
                throw new NullPointerException("parameterObject is null!");
            } else {
                // 分页参数作为参数对象parameterObject的一个属性
                PageParameter page = (PageParameter) metaStatementHandler
                        .getValue("delegate.boundSql.parameterObject.page");
                String sql = boundSql.getSql();
                // 重写sql
                String pageSql = buildPageSql(sql, page);
                metaStatementHandler.setValue("delegate.boundSql.sql", pageSql);
                // 采用物理分页后，就不需要mybatis的内存分页了，所以重置下面的两个参数
                metaStatementHandler.setValue("delegate.rowBounds.offset",
                RowBounds.NO_ROW_OFFSET);
                metaStatementHandler.setValue("delegate.rowBounds.limit",
                                              RowBounds.NO_ROW_LIMIT);
                Connection connection = (Connection) invocation.getArgs()[0];
                // 重设分页参数里的总页数等
                setPageParameter(sql, connection, mappedStatement, boundSql, page);
            }
        }
        // 将执行权交给下一个拦截器
        return invocation.proceed();
    }

    private String buildPageSql(String sql, PageParameter page) {
        if (page != null) {
            StringBuilder pageSql = new StringBuilder();
            if ("mysql".equals(dialect)) {
                pageSql = buildPageSqlForMysql(sql, page);
            } else if ("oracle".equals(dialect)) {
                pageSql = buildPageSqlForOracle(sql, page);
            } else {
                return sql;
            }
            return pageSql.toString();
        } else {
            return sql;
        }
    }

    /* mysql-分页SQL拼接 */
    public StringBuilder buildPageSqlForMysql(String sql, PageParameter page) {
        StringBuilder pageSql = new StringBuilder(100);
        String beginrow = String
```

```
        .valueOf((page.getCurrentPage() - 1) * page.getPageSize());
    pageSql.append(sql);
    pageSql.append(" limit " + beginrow + "," + page.getPageSize());
    return pageSql;
}

/* oracle-分页SQL拼接 */
public StringBuilder buildPageSqlForOracle(String sql, PageParameter page) {
    StringBuilder pageSql = new StringBuilder(100);
    String beginrow = String.valueOf((page.getCurrentPage() - 1) * page.getPageSize());
    String endrow = String.valueOf(page.getCurrentPage() * page.getPageSize());
    pageSql.append("select * from ( select temp.*, rownum row_id from ( ");
    pageSql.append(sql);
    pageSql.append(" ) temp where rownum <= ").append(endrow);
    pageSql.append(") where row_id > ").append(beginrow);
    return pageSql;
}
```

7.1.3 总记录数/总页数

```
/**
 * 从数据库里查询总的记录数并计算总页数，回写进分页参数<code>PageParameter</code>,这样调用
 * 者就可用通过 分页参数<code>PageParameter</code>获得相关信息。
 *
 * @param sql
 * @param connection
 * @param mappedStatement
 * @param boundSql
 * @param page
 */
private void setPageParameter(String sql, Connection connection, MappedStatement mappedStatement,
        BoundSql boundSql, PageParameter page) {
    // 记录总记录数
    String countSql = "select count(0) from (" + sql + ") as total";
    PreparedStatement countStmt = null;
    ResultSet rs = null;
    try {
        countStmt = connection.prepareStatement(countSql);
        BoundSql countBS = new BoundSql(mappedStatement.getConfiguration(), countSql,
                boundSql.getParameterMappings(), boundSql.getParameterObject());
        setParameters(countStmt, mappedStatement, countBS, boundSql.getParameterObject());
        rs = countStmt.executeQuery();
        int totalCount = 0;
        if (rs.next()) {
            totalCount = rs.getInt(1);
        }
        page.setTotalCount(totalCount);
        int totalPage = totalCount / page.getPageSize() + ((totalCount % page.getPageSize() == 0) ? 0 : 1);
        page.setTotalPage(totalPage);
    } catch (SQLException e) {
        logger.error("Ignore this exception", e);
    } finally {
        try {
            rs.close();
        } catch (SQLException e) {
            logger.error("Ignore this exception", e);
        }
        try {
            countStmt.close();
        } catch (SQLException e) {
            logger.error("Ignore this exception", e);
        }
    }
}

/**
```

```
 * 对SQL参数(?)设值
 *
 * @param ps
 * @param mappedStatement
 * @param boundSql
 * @param parameterObject
 * @throws SQLException
 */
private void setParameters(PreparedStatement ps, MappedStatement mappedStatement, BoundSql boundSql,
        Object parameterObject) throws SQLException {
    ParameterHandler parameterHandler = new DefaultParameterHandler(mappedStatement, parameterObject,
boundSql);
    parameterHandler.setParameters(ps);
}
```

7.1.4 plugin实现

```
public Object plugin(Object target) {
    // 当目标类是StatementHandler类型时，才包装目标类，否者直接返回目标本身,减少目标被代理的
    // 次数
    if (target instanceof StatementHandler) {
        return Plugin.wrap(target, this);
    } else {
        return target;
    }
}
```

8. SQL生成插件

9. 改造Cache插件

文章主要解决了`开启cache时，前面所写的分页和SQL生成插件不能正常工作`的问题；

10. Spring集成

10.1 主要配置

需要配置SqlSessionFactoryBean，该配置会加入数据源和mybatis xml配置文件路径等信息：

```
<bean id="sqlSessionFactory"
    class="org.mybatis.spring.SqlSessionFactoryBean">
    <property name="dataSource" ref="datasource"></property>
    <property name="configLocation" value="classpath:context/mybatis-config.xml"></property>
    <!--
    mapperLocations：通过正则表达式，支持mybatis动态扫描添加mapper不用像ibatis，用一个还要蛋疼滴添加一个include
    -->
    <property name="mapperLocations"
              value="classpath*:/com/tx/demo/**/*SqlMap.xml" />
    <!--
    typeHandlersPackage: 由于mybatis默认入参如果为空，又没有指定jdbcType时会抛出异常，在这里通过配置一些默认的类型空
值插入的handle，以便处理mybatis的默认类型为空的情况。
    例如NullAbleStringTypeHandle通过实现当String字符串中为null是调用ps.setString(i,null)其他常用类型雷同。
    -->
    <property name="typeHandlersPackage"
              value="com.tx.core.mybatis.handler"></property>
    <!--
    failFast：开启后将在启动时检查设定的parameterMap,resultMap是否存在，是否合法。个人建议设置为true,这样可以尽快定位
解决问题。不然在调用过程中发现错误，会影响问题定位。
    -->
    <property name="failFast" value="true"></property>
    <property name="plugins">
        <array>
```

```
            <bean class="com.tx.core.mybatis.interceptor
                        .PagedDiclectStatementHandlerInterceptor">
                <property name="dialect">
                    <bean class="org.hibernate.dialect.PostgreSQLDialect"></bean>
                </property>
            </bean>
        </array>
    </property>
</bean>
<!--
myBatisExceptionTranslator：用以支持spring的异常转换，通过配置该translator可以将mybatis异常转换为spring中定义的
DataAccessException。
-->
<bean id="myBatisExceptionTranslator"
      class="org.mybatis.spring.MyBatisExceptionTranslator">
    <property name="dataSource">
        <ref bean="datasource"></ref>
    </property>
</bean>

<bean id="sqlSessionTemplate" class="org.mybatis.spring.SqlSessionTemplate">
    <constructor-arg name="sqlSessionFactory"
                     ref="sqlSessionFactory"></constructor-arg>
    <constructor-arg name="executorType" ref="SIMPLE"></constructor-arg>
    <constructor-arg name="exceptionTranslator"
                     ref="myBatisExceptionTranslator"></constructor-arg>
</bean>

<bean id="myBatisDaoSupport" class="com.tx.core.mybatis.support.MyBatisDaoSupport">
    <property name="sqlSessionTemplate">
        <ref bean="sqlSessionTemplate"/>
    </property>
</bean>
```

10.2 事务配置

```
<!-- 配置事务 -->
<bean id="txManager"
        class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource" />
</bean>
<!-- 配置基于注解的事务aop -->
<tx:annotation-driven transaction-manager="txManager" proxy-target-class="true"/>
```

或

```
<!-- 配置事务管理器，注意这里的dataSource和SqlSessionFactoryBean的dataSource要一致，不然事务就没有作用了 -->
<bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource" />
</bean>

<!-- 配置事务的传播特性 -->
<bean id="baseTransactionProxy"
class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean" abstract="true">
    <property name="transactionManager" ref="transactionManager" />
    <property name="transactionAttributes">
        <props>
            <prop key="add*">PROPAGATION_REQUIRED</prop>
            <prop key="edit*">PROPAGATION_REQUIRED</prop>
            <prop key="remove*">PROPAGATION_REQUIRED</prop>
            <prop key="insert*">PROPAGATION_REQUIRED</prop>
            <prop key="update*">PROPAGATION_REQUIRED</prop>
            <prop key="del*">PROPAGATION_REQUIRED</prop>
            <prop key="*">readOnly</prop>
        </props>
```

```
    </property>
</bean>

<!--把事务控制在Service层-->
<aop:config>
    <aop:pointcut id="pc" expression="execution(public * com.jeasy..service.*.*(..))" />
    <aop:advisor pointcut-ref="pc" advice-ref="baseTransactionProxy" />
</aop:config>
```

10.3 batch模式

定义模板：

```
<!--通过模板定制mybatis的行为 -->
<bean id="sqlSessionTemplateSimple" class="org.mybatis.spring.SqlSessionTemplate">
    <constructor-arg index="0" ref="sqlSessionFactory" />
    <!--更新采用单个模式 -->
    <constructor-arg index="1" value="SIMPLE"/>
</bean>

<!--通过模板定制mybatis的行为 -->
<bean id="sqlSessionTemplateBatch" class="org.mybatis.spring.SqlSessionTemplate">
    <constructor-arg index="0" ref="sqlSessionFactory" />
    <!--更新采用批量模式 -->
    <constructor-arg index="1" value="BATCH"/>
</bean>
```

配置：

```
<bean id="userDao" class="org.mybatis.spring.mapper.MapperFactoryBean">
    <property name="mapperInterface" value="com.xxx.dao.UserDao" />
    <property name="sqlSessionTemplate" ref="sqlSessionTemplateBatch" />
</bean>
```

- 批量设置：

```
<!-- 采用自动扫描方式创建mapper bean(单个更新模式) -->
<bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
    <property name="basePackage" value="com.xxx.dao" />
    <property name="sqlSessionTemplateBeanName"
              value="sqlSessionTemplateSimple" />
    <property name="markerInterface" value="com.xxx.dao.SimpleDao" />
</bean>

<!-- 采用自动扫描方式创建mapper bean(批量更新模式) -->
<bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
    <property name="basePackage" value="com.xxx.dao" />
    <property name="sqlSessionTemplateBeanName"
              value="sqlSessionTemplateBatch" />
    <property name="markerInterface" value="com.xxx.dao.BatchDao" />
</bean>
```

上面的MapperScannerConfigurer三个属性：

- **basePackage**：扫描器开始扫描的基础包名，支持嵌套扫描；
- **sqlSessionTemplateBeanName**：前文提到的模板bean的名称；
- **markerInterface**：`基于接口的过滤器`，实现了该接口的dao才会被扫描器扫描，与basePackage是与的作用。

还要更多属性：

- **annotationClass**：注解过滤，`配置了该注解的dao才会被扫描器扫描`，与basePackage是与的作用。annotationClass与markerInterface，只能二选一；

11. 缓存源码分析

> 为了提高数据利用率和减小服务器和数据库的压力，`MyBatis 会对于一些查询提供会话级别的数据缓存`，会将对某一次查询，放置到SqlSession 中，在允许的时间间隔内，对于完全相同的查询，MyBatis 会直接将缓存结果返回给用户，而不用再到数据库中查找。

> 我们可以进而考虑对SQL执行结果的缓存来提升性能。缓存数据都是key-value的格式，那么这个key怎么来呢？怎么保证唯一呢？即使同一条SQL语句几次访问的过程中由于传入参数的不同，得到的执行SQL语句也是不同的。那么缓存起来的时候是多对。但是SQL语句和传入参数两部分合起来可以作为数据缓存的key值

11.1 缓存概念

11.1.1 一级缓存

对于会话（Session）级别的数据缓存，我们称之为一级数据缓存，简称一级缓存。

> 只是一个MyBatis对外的接口，
> `SqlSession将它的工作交给了Executor执行器这个角色来完成，负责完成对数据库的各种操作`。当创建了一个SqlSession对象时，`MyBatis会为这个SqlSession对象创建一个新的Executor执行器，而缓存信息就被维护在这个Executor执行器中`，MyBatis将缓存和对缓存相关的操作封装成了Cache接口中。Executor接口的实现类BaseExecutor中拥有一个Cache接口的实现类PerpetualCache，`则对于BaseExecutor对象而言，它将使用PerpetualCache对象维护缓存`。

11.1.1.1 一级缓存的生命周期

1. MyBatis在开启一个数据库会话时，会创建一个新的SqlSession对象，SqlSession对象中会有一个新的Executor对象，Executor对象中持有一个新的PerpetualCache对象；
   `当会话结束时，SqlSession对象及其内部的Executor对象还有PerpetualCache对象也一并释放掉`。
2. 如果`SqlSession调用了close()方法`，会释放掉一级缓存PerpetualCache对象，一级缓存将不可用；
3. 如果`SqlSession调用了clearCache()`，会清空PerpetualCache对象中的数据，但是该对象仍可使用；
4. SqlSession中`执行了任何一个update操作(update()、delete()、insert())`，都会清空PerpetualCache对象的数据，`但是该对象可以继续使用`；

11.1.1.2 一级缓存的工作流程

1. 对于某个查询，根据**statementId + rowBounds + 传递给JDBC的SQL + 传递给JDBC的参数值**来构建一个key值，根据这个key值去缓存Cache中取出对应的key值存储的缓存结果；
2. 判断从Cache中根据特定的key值取的数据数据是否为空，即是否命中；
3. 如果命中，则直接将缓存结果返回；
4. 如果没命中：

   4.1 去数据库中查询数据，得到查询结果；

   4.2 将key和查询到的结果分别作为key,value对存储到Cache中；

   4.3 将查询结果返回；
5. 结束。

11.1.1.3 使用注意

1. 对于数据变化频率很大，并且需要高时效准确性的数据要求，我们使用SqlSession查询的时候，要控制好SqlSession的生存时间，SqlSession的生存时间越长，它其中缓存的数据有可能就越旧，从而造成和真实数据库的误差；同时对于这种情

况，用户也可以手动地适时清空SqlSession中的缓存；
2. 对于只执行、并且频繁执行大范围的select操作的SqlSession对象，SqlSession对象的生存时间不应过长。

11.1.2 二级缓存

> MyBatis的二级缓存是`Application级别的缓存`。

> 当开一个会话时，一个SqlSession对象会使用一个Executor对象来完成会话操作，
> `MyBatis的二级缓存机制的关键就是对这个Executor对象做文章`。如果用户配置了`"cacheEnabled=true"`，那么MyBatis在为SqlSession对象创建Executor对象时，`会对Executor对象加上一个装饰者：CachingExecutor`，这时SqlSession使用CachingExecutor对象来完成操作请求。
> `CachingExecutor对于查询请求，会先判断该查询请求在Application级别的二级缓存中是否有缓存结果`，如果有查询结果，则直接返回缓存结果；如果缓存中没有，再交给真正的Executor对象来完成查询操作，
> `之后CachingExecutor会将真正Executor返回的查询结果放置到缓存中`，然后在返回给用户。

> MyBatis并不是简单地对整个Application就只有一个Cache缓存对象，它将缓存划分的更细，即是Mapper级别的，即每一个Mapper都可以拥有一个Cache对象

11.1.2.1 配置

- 全局开关，默认为true：

```
<settings>
    <setting name="cacheEnabled" value="true"/>
</settings>
```

- 命名空间配置(mapper.xml中)，默认开启：

```
<cache
        eviction="FIFO"
        flushInterval="60000"
        size="512"
        readOnly="true"/>
```

eviction(收回策略):

- LRU 最近最少使用的，移除最长时间不被使用的对象，这是默认值
- FIFO 先进先出，按对象进入缓存的顺序来移除它们
- SOFT 软引用，移除基于垃圾回收器状态和软引用规则的对象
- WEAK 弱引用，更积极的移除基于垃圾收集器状态和弱引用规则的对象

flushInterval（刷新间隔）可以被设置为任意的正整数，而且它们代表一个合理的毫秒形式的时间段。 默认情况不设置，即没有刷新间隔，缓存仅仅在调用语句时刷新；

size（引用数目）可以被设置为任意的正整数，要记住缓存的对象数目和运行环境的可用内存资源数目，默认1024；

readOnly（只读）属性可以被设置为true后者false。 只读的缓存会给所有调用者返回缓存对象的相同实例，因此这些对象不能被修改，这提供了很重要的性能优势。 可读写的缓存会通过序列化返回缓存对象的拷贝，这种方式会慢一些，但很安全，因此默认为false；

- 注解方式

```
@CacheNamespace
public interface RoleMapper {
    ....
}
```

- 参照缓存

  > 同时使用注解方式和XML映射文件配置二级缓存时，会抛出如下异常，因为Mapper接口和对应的XML文件是相同的命名空间；

  可以使用配置：

```
@CacheNamespaceRef(RoleMapper.class)
public interface RoleMapper{

}
```

  > MyBatis很少会同时使用Mapper接口注解方式和XML映射文件，所以参照缓存并不是为了解决这个问题而设计的，参照缓存主要是为了解决脏读。

- 要想使某条Select查询支持二级缓存，你需要保证：
  1. MyBatis支持二级缓存的总开关：全局配置变量参数 cacheEnabled=true
  2. 该select语句所在的Mapper，配置了 `<cache>` 或 `<cached-ref>` 节点，并且有效
  3. 该select语句的参数 useCache=true

11.1.2.2 注意

> 只能在【只有**单表操作**】的表上使用缓存，不只是要保证这个表在整个系统中只有单表操作，而且**和该表有关的全部操作必须全部在一个namespace**下。

如果违反了这条原则：

> 例如在UserMapper.xml中有大多数针对user表的操作。但是在一个XXXMapper.xml中，还有针对user单表的操作。
>
> 这会导致user在两个命名空间下的数据不一致。如果在UserMapper.xml中做了刷新缓存的操作，在XXXMapper.xml中缓存仍然有效，如果有针对user的单表查询，使用缓存的结果可能会不正确。

不过，多表关联的情况可以研究一下 `cache-ref`，标记一下// todo；

11.1.2.3 调用顺序

请注意，如果你的MyBatis使用了二级缓存，并且你的Mapper和select语句也配置使用了二级缓存，那么在执行select查询的时候，MyBatis会先从二级缓存中取输入，其次才是一级缓存，即MyBatis查询数据的顺序是：
`二级缓存 ——> 一级缓存 —> 数据库`。

11.1.2.4使用建议

- 多表操作时，不要使用二级缓存；
- 代码生成器生成的代码一般都是单表操作，一般可以使用二级缓存；但是全局开关打开后，`<cache>` 标签默认也是开启的，要注意把多表的mapper的cache关闭；
- 使用 `mybatis-enhanced-cache`

  原理很简单，就是 当执行了某个update操作时，根据配置信息去清空指定的查询语句在Cache中所产生的缓存数据。

```
<plugins>
    <plugin interceptor="org.luanlouis.mybatis
                        .plugin.cache.EnhancedCachingExecutor">
```

```
        <!-- dependencys.xml是StatementId之间的依赖关系的配置文件路径 -->
        <property name="dependency" value="dependencys.xml"/>
        <property name="cacheEnabled" value="true"/>
    </plugin>
</plugins>
```

dependencys.xml如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<dependencies>
    <statements>
        <statement id="com.louis.mybatis.dao
                        .DepartmentsMapper.updateByPrimaryKey">
            <observer id="com.louis.mybatis.dao
                            .EmployeesMapper.selectWithDepartments" />
        </statement>
    </statements>
</dependencies>
```

每个mapper.xml中还需要配置`<cache>`；

11.1.2.5 自定义cache

自己实现`org.apache.ibatis.cache.Cache`接口，并通过`<cache type="">`配置；

12 Mybatis原理篇

12.1 回顾JDBC

12.1.1 代码示例：

```
public static List<Map<String,Object>> queryForList(){
    Connection connection = null;
    ResultSet rs = null;
    PreparedStatement stmt = null;
    List<Map<String,Object>> resultList = new ArrayList<Map<String,Object>>();

    try {
        // 加载JDBC驱动
        Class.forName("oracle.jdbc.driver.OracleDriver").newInstance();
        String url = "jdbc:oracle:thin:@localhost:1521:ORACLEDB";

        String user = "trainer";
        String password = "trainer";

        // 获取数据库连接
        connection = DriverManager.getConnection(url,user,password);

        String sql = "select * from userinfo where user_id = ? ";
        // 创建Statement对象（每一个Statement为一次数据库执行请求）
        stmt = connection.prepareStatement(sql);

        // 设置传入参数
        stmt.setString(1, "zhangsan");

        // 执行SQL语句
        rs = stmt.executeQuery();

        // 处理查询结果（将查询结果转换成List<Map>格式）
        ResultSetMetaData rsmd = rs.getMetaData();
        int num = rsmd.getColumnCount();

        while(rs.next()){
```

```
            Map map = new HashMap();
            for(int i = 0;i < num;i++){
                String columnName = rsmd.getColumnName(i+1);
                map.put(columnName,rs.getString(columnName));
            }
            resultList.add(map);
        }

    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        try {
            // 关闭结果集
            if (rs != null) {
                rs.close();
                rs = null;
            }
            // 关闭执行
            if (stmt != null) {
                stmt.close();
                stmt = null;
            }
            if (connection != null) {
                connection.close();
                connection = null;
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
    return resultList;
}
```

12.1.2 JDBC缺点

- 数据库连接频繁的开启和关闭本身就造成了**资源浪费**；
- 即使使用了数据库连接池，但是面对各种各样的连接池，配置变化太大；
- SQL语句可读性差；不支持动态SQL，使得代码存在大量重复SQL，或存在大量判断语句以实现多条件SQL复用；
- 执行结果处理过程是重复工作，效率低下；

12.2 Mybatis核心类

- **SqlSession** 作为MyBatis工作的主要顶层API，表示和数据库交互的会话，完成必要数据库增删改查功能

> SqlSession有一个重要的方法getMapper，顾名思义，这个方式是用来获取Mapper对象的。什么是Mapper对象？根据Mybatis的官方手册，应用程序除了要初始并启动Mybatis之外，还需要定义一些接口，
> `接口里定义访问数据库的方法，存放接口的包路径下需要放置同名的XML配置文件`。
> `SqlSession的getMapper方法是联系应用程序和Mybatis纽带，应用程序访问getMapper时，Mybatis会根据传入的接口类型和对应的XML配置文件生成一个代理对象，这个代理对象就叫Mapper对象`
> 。应用程序获得Mapper对象后，就应该通过这个Mapper对象来访问Mybatis的SqlSession对象，这样就达到里插入到Mybatis流程的目的。

- **Executor** MyBatis执行器，是MyBatis 调度的核心，负责SQL语句的生成和查询缓存的维护
- **StatementHandler** 封装了JDBC Statement操作，负责对JDBC statement 的操作，如设置参数、将Statement结果集转换成List集合。
- **ParameterHandler** 负责对用户传递的参数转换成JDBC Statement 所需要的参数，
- **ResultSetHandler** 负责将JDBC返回的ResultSet结果集对象转换成List类型的集合；
- **TypeHandler** 负责java数据类型和jdbc数据类型之间的映射和转换

- **MappedStatement** MappedStatement维护了一条<select|update|delete|insert>节点的封装，
- **SqlSource** 负责根据用户传递的parameterObject，动态地生成SQL语句，将信息封装到BoundSql对象中，并返回
- **BoundSql** 表示动态生成的SQL语句以及相应的参数信息
- **Configuration** MyBatis所有的配置信息都维持在Configuration对象之中。
- **TransactionFactory** 事务创建类，有`JdbcTransactionFactory`和`MangedTransactionFactory`两个实现类；
- **XMLConfigBuilder**：解析mybatis中configLocation属性中的全局xml文件，内部会使用`XMLMapperBuilder`解析各个xml文件。
- **XMLStatementBuilder**处理mapper.xml中的每个节点，比如：select、update等；
- **XMLScriptBuilder**解析每个节点中的SQL脚本；

12.3 environment

> MyBatis对节点的解析会生成TransactionFactory实例；而对解析会生成datasouce实例，
> `作为<environment>节点，会根据TransactionFactory和DataSource实例创建一个Environment对象`，Environment表示着一个数据库的连接，`生成后的Environment对象会被设置到Configuration实例中`，以供后续的使用。

12.4 批量模式

在Insert操作时，`在事务没有提交之前，是没有办法获取到自增的id`；

13. 优化

13.1 连接池

- 由于创建一个数据库连接所占用的资源比较大，`对于数据吞吐量大和访问量非常大的应用而言`，连接池的设计就显得非常重要。

13.2 N+1问题

问题：自行了解；

解决：

- 采取嵌套resultMap方式；

```
<resultMap type="com.foo.bean.BlogInfo" id="BlogInfo">
    <id column="blog_id" property="blogId"/>
    <result column="title" property="title"/>
    <association property="author" column="blog_author_id"
             javaType="com.foo.bean.Author">
        <!-- 作者-博客：一对多 -->
        <id column="author_id" property="authorId"/>
        <result column="user_name" property="userName"/>
        <result column="password" property="password"/>
        <result column="email" property="email"/>
        <result column="biography" property="biography"/>
    </association>
    <collection property="posts" column="blog_post_id"
                ofType="com.foo.bean.Post">
        <!-- 博客-评论：一对多 -->
        <id column="post_id" property="postId"/>
        <result column="blog_id" property="blogId"/>
        <result column="create_time" property="createTime"/>
        <result column="subject" property="subject"/>
        <result column="body" property="body"/>
```

```
        <result column="draft" property="draft"/>
    </collection>
</resultMap>
```

这种方式，只查询一次，然后，组装查询到的结果；

- 延迟加载（在使用到关联对象的时候，才进行查询）；

  配置：

```
<setting name="lazyLoadingEnabled" value="true"/>
<setting name="aggressiveLazyLoading" value="false"/>
```

扩展

OGNL

Object-Graph Navigation Language（OGNL）是一种表达式语言；Mybatis的动态SQL就使用到了它；

> 它是一个功能强大的表达式语言，用来获取和设置 java 对象的属性 , 它旨在提供一个更高抽象度语法来对 java 对象图进行导航。 对于开发者来说，使用 OGNL，可以用简洁的语法来完成对 java 对象的导航。通常来说： 通过一个“路径”来完成对象信息的导航，这个“路径”可以是到 java bean 的某个属性，或者集合中的某个索引的对象，等等，而不是直接使用 get 或者 set 方法来完成。

MetaObject

`MetaObject是Mybatis提供的一个的工具类，通过它包装一个对象后可以获取或设置该对象的原本不可访问的属性（比如那些私有属性）`。它有个三个重要方法经常用到：

- MetaObject forObject(Object object,ObjectFactory objectFactory, ObjectWrapperFactory objectWrapperFactory) 用于包装对象；
- Object getValue(String name) 用于获取属性的值（支持OGNL的方法）；
- void setValue(String name, Object value) 用于设置属性的值（支持OGNL的方法）；

系列博客

MyBatis中的N+1问题