Mybatis

第一部分 自定义持久层框架

1.1 分析jdbc操作问题

```
public static void main(String[] args) {
        Connection connection = null;
        PreparedStatement preparedStatement = null;
        ResultSet resultSet = null;
    try {
        // 加载数据库驱动
        Class.forName("com.mysql.jdbc.Driver");
        // 通过驱动管理类获取数据库链接
        connection =
DriverManager.getConnection("jdbc:mysql://localhost:3306/mybatis?
characterEncoding=utf-8", "root", "root");
        // 定义sql语句 ?表示占位符
        String sql = "select * from user where username = ?";
        // 获取预处理statement
        preparedStatement = connection.prepareStatement(sql);
        // 设置参数，第一个参数为sql语句中参数的序号（从1开始），第二个参数为设置的参数值
        preparedStatement.setString(1, "tom");
        // 向数据库发出sql执行查询，查询出结果集
        resultSet = preparedStatement.executeQuery();
        // 遍历查询结果集
        while (resultSet.next()) {
              int id = resultSet.getInt("id");
              String username = resultSet.getString("username");
        // 封装User
              user.setId(id);
              user.setUsername(username);
          }
          System.out.println(user);

          }
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        // 释放资源
        if (resultSet != null) {
          try {
```

```
          resultSet.close();
        } catch (SQLException e) {
        e.printStackTrace();
        }
      }
      if (preparedStatement != null) {
        try {
          preparedStatement.close();
        } catch (SQLException e) {
          e.printStackTrace();
        }
      }
      if (connection != null) {
        try {
          connection.close();
        } catch (SQLException e) {
          e.printStackTrace();
        }
      }
    }
```

JDBC问题总结：

1、 数据库连接创建、释放频繁造成系统资源浪费，从而影响系统性能。

2、 Sql语句在代码中硬编码，造成代码不易维护，实际应用中sql变化的可能较大，sql变动需要改变java代码。

3、 使用preparedStatement向占有位符号传参数存在硬编码，因为sql语句的where条件不一定，可能多也可能少，修改sql还要修改代码，系统不易维护。

4、对结果集解析存在硬编码（查询列名），sql变化导致解析代码变化，系统不易维护，如果能将数据库记录封装成pojo对象解析比较方便

1.2 问题解决思路

数据库频繁创建连接、释放资源：连接池

sql语句及参数硬编码：配置文件

手动解析封装返回结果集：反射、内省

1.3 自定义框架设计

使用端：

提供核心配置文件

sqlMapConfig.xml: 存放数据源信息，引入mapper.xml

Mapper.xml: sql语句的配置文件信息

框架端：

1.读取配置文件

读取完以后以流的形式存在，我们不能讲读取到的配置信息以流的形式存放在内存中，不好操作，可以创建javaBean来存储

（1）Configuration: 存放数据库基本信息、Map<唯一标识,Mapper> 唯一标识：namespace+"."+id

（2）MappedStatement: sql语句、statement类型、输入参数java类型、输出参数java类型

2.解析配置文件

创建SqlSessionFactroyBuilder类：

方法：SqlSessionFactory build():

第一：使用dom4j解析配置文件，将解析出来的内容封装到Configuration和MappedStatement中

第二：创建SqlSessionFactory的实现类DefaultSqlSession

3.创建SqlSessionFactory:

方法：openSession()：获取sqlSession接口的实现类实例对象

4.创建SqlSession接口及实现类: 主要封装CRUD方法

方法：selectList(String StatementId,Object param)：查询所有

selectOne(String StatementId,Object param)：查询单个

close() 释放资源

具体实现：封装JDBC完成对数据库表的查询操作

设计到的设计模式

构建者模式、工厂模式、代理模式

1.4 自定义框架实现

在使用端项目中创建配置配置文件

创建sqlMapConfig.xml

```xml
<configuration>
<!--数据库连接信息-->
<property name="driverClass" value="com.mysql.jdbc.Driver"></property>
<property name="jdbcUrl" value="jdbc:mysql:///zdy_mybatis"></property>
<property name="user" value="root"></property>
<property name="password" value="root"></property>

<!--引入sql配置信息-->
<mapper resource="mapper.xml"></mapper>
</configuration>
```

mapper.xml

```xml
<mapper namespace="User">
    <select id="selectOne" paramterType="com.lagou.pojo.User"
resultType="com.lagou.pojo.User">
        select * from user where id = #{id} and username =#{username}
    </select>

      <select id="selectList" resultType="com.lagou.pojo.User">
        select * from user
    </select>
</mapper>
```

User实体

```java
public class User {
    //主键标识
    private  Integer id;
    //用户名
    private String username;

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    @Override
    public String toString() {
        return "User{" +
                "id=" + id +
                ", username='" + username + '\'' +
                '}';
    }
}
```

再创建一个Maven子工程并且导入需要用到的依赖坐标

```xml
    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <maven.compiler.encoding>UTF-8</maven.compiler.encoding>
        <java.version>1.8</java.version>
        <maven.compiler.source>1.8</maven.compiler.source>
        <maven.compiler.target>1.8</maven.compiler.target>
    </properties>

    <dependencies>
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>5.1.17</version>
        </dependency>
        <dependency>
            <groupId>c3p0</groupId>
            <artifactId>c3p0</artifactId>
            <version>0.9.1.2</version>
        </dependency>
        <dependency>
            <groupId>log4j</groupId>
            <artifactId>log4j</artifactId>
            <version>1.2.12</version>
        </dependency>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.10</version>
        </dependency>
        <dependency>
            <groupId>dom4j</groupId>
            <artifactId>dom4j</artifactId>
            <version>1.6.1</version>
        </dependency>
        <dependency>
            <groupId>jaxen</groupId>
            <artifactId>jaxen</artifactId>
            <version>1.1.6</version>
        </dependency>
    </dependencies>
```

Configuration

```
public class Configuration {

    //数据源
    private DataSource dataSource;
    //map集合: key:statementId   value:MappedStatement
    private Map<String,MappedStatement> mappedStatementMap = new
HashMap<String, MappedStatement>();
```

```
    public DataSource getDataSource() {
        return dataSource;
    }

    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;
    }

    public Map<String, MappedStatement> getMappedStatementMap() {
        return mappedStatementMap;
    }

    public void setMappedStatementMap(Map<String, MappedStatement>
mappedStatementMap) {
        this.mappedStatementMap = mappedStatementMap;
    }
}
```

MappedStatement

```
public class MappedStatement {
    //id
    private Integer id;
    //sql语句
    private String sql;
    //输入参数
    private Class<?> paramterType;
    //输出参数
    private Class<?> resultType;

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getSql() {
        return sql;
    }

    public void setSql(String sql) {
        this.sql = sql;
    }

    public Class<?> getParamterType() {
        return paramterType;
    }

    public void setParamterType(Class<?> paramterType) {
        this.paramterType = paramterType;
    }

    public Class<?> getResultType() {
        return resultType;
    }

    public void setResultType(Class<?> resultType) {
        this.resultType = resultType;
    }
}
```

Resources

```
public class Resources {
   public static InputStream getResourceAsSteam(String path){
       InputStream resourceAsStream =
Resources.class.getResourceAsStream(path);
       return resourceAsStream;
  }
}
```

SqlSessionFactoryBuilder

```
public class SqlSessionFactoryBuilder {

    private Configuration configuration;

    public SqlSessionFactoryBuilder() {
        this.configuration = new Configuration();
    }

    public SqlSessionFactory build(InputStream inputStream) throws
DocumentException, PropertyVetoException, ClassNotFoundException {
        //1.解析配置文件，封装Configuration
        XMLConfigerBuilder xmlConfigerBuilder = new
XMLConfigerBuilder(configuration);
        Configuration configuration =
xmlConfigerBuilder.parseConfiguration(inputStream);

        //2.创建sqlSessionFactory
        SqlSessionFactory sqlSessionFactory = new
DefaultSqlSessionFactory(configuration);

        return sqlSessionFactory;
    }
}
```

XMLConfigerBuilder

```
public class XMLConfigerBuilder {

    private  Configuration configuration;

    public XMLConfigerBuilder(Configuration configuration) {
        this.configuration = new Configuration();
    }


    public Configuration parseConfiguration(InputStream inputStream) throws
DocumentException, PropertyVetoException, ClassNotFoundException {
        Document document  = new SAXReader().read(inputStream);
        //<configuation>
        Element rootElement = document.getRootElement();
        List<Element> propertyElements =
rootElement.selectNodes("//property");
        Properties properties = new Properties();
        for (Element propertyElement : propertyElements) {
            String name = propertyElement.attributeValue("name");
            String value = propertyElement.attributeValue("value");
            properties.setProperty(name,value);
        }
        //连接池
        ComboPooledDataSource comboPooledDataSource = new
ComboPooledDataSource();

 comboPooledDataSource.setDriverClass(properties.getProperty("driverClass"));
        comboPooledDataSource.setJdbcUrl(properties.getProperty("jdbcUrl"));
        comboPooledDataSource.setUser(properties.getProperty("username"));
        comboPooledDataSource.setPassword(properties.getProperty("password"));

        //填充configuration
        configuration.setDataSource(comboPooledDataSource);

        //mapper部分
        List<Element> mapperElements = rootElement.selectNodes("//mapper");
        XMLMapperBuilder xmlMapperBuilder = new
XMLMapperBuilder(configuration);
        for (Element mapperElement : mapperElements) {
            String mapperPath = mapperElement.attributeValue("resource");
            InputStream resourceAsSteam =
Resources.getResourceAsSteam(mapperPath);
            xmlMapperBuilder.parse(resourceAsSteam);
        }
        return configuration;


    }
```

XMLMapperBuilder

```
public class XMLMapperBuilder {

    private Configuration configuration;

    public XMLMapperBuilder(Configuration configuration) {
        this.configuration = configuration;
    }

    public void parse(InputStream inputStream) throws DocumentException,
ClassNotFoundException {
        Document document = new SAXReader().read(inputStream);
        Element rootElement = document.getRootElement();
        String namespace = rootElement.attributeValue("namespace");
        List<Element> select = rootElement.selectNodes("select");

        for (Element element : select) {
            //id的值
            String id = element.attributeValue("id");
            String paramterType = element.attributeValue("paramterType");
            String resultType = element.attributeValue("resultType");
            //输入参数class
            Class<?> paramterTypeClass = getClassType(paramterType);
            //返回结果class
            Class<?> resultTypeClass = getClassType(resultType);
            //statementId
            String key = namespace +"."+id;
            //sql语句
            String textTrim = element.getTextTrim();


            //封装mappedStatement
            MappedStatement mappedStatement = new MappedStatement();
            mappedStatement.setId(id);
            mappedStatement.setParamterType(paramterTypeClass);
            mappedStatement.setResultType(resultTypeClass);
            mappedStatement.setSql(textTrim);
            //填充configuration
            configuration.getMappedStatementMap().put(key,mappedStatement);
        }
    }
    private Class<?> getClassType(String paramterType) throws
ClassNotFoundException {

        Class<?> aClass = Class.forName(paramterType);
        return aClass;


    }
```

sqlSessionFactory接口及DefaultSqlSessionFactory实现类

```
public interface SqlSessionFactory {

    public SqlSession openSession();
}
```

```
public class DefaultSqlSessionFactory implements  SqlSessionFactory {

    private Configuration configuration;

    public DefaultSqlSessionFactory(Configuration configuration) {
        this.configuration = configuration;
    }

    public SqlSession openSession(){
        return new DefaultSqlSession(configuration);
    }
}
```

sqlSession接口及DefaultSqlSession实现类

```
public interface SqlSession {

    public <E> List<E> selectList(String statementId, Object... param) throws
Exception;
    public <T> T selectOne(String statementId,Object... params) throws
Exception;
    public void close() throws SQLException;
}
```

```
public class DefaultSqlSession implements SqlSession {

    private Configuration configuration;

    public DefaultSqlSession(Configuration configuration) {
        this.configuration = configuration;
    }

    //处理器对象
    private Executor simpleExcutor = new SimpleExecutor();

    public <E> List<E> selectList(String statementId,Object... param) throws
Exception{
        MappedStatement mappedStatement =
configuration.getMappedStatementMap().get(statementId);



        List<E> query = simpleExcutor.query(configuration, mappedStatement,
param);

        return query;
    }

    //selectOne中调用selectList
    public <T> T selectOne(String statementId,Object... params) throws
Exception {

        List<Object> objects = selectList(statementId, params);
        if(objects.size() ==1){
            return (T) objects.get(0);
        }else {
            throw new RuntimeException("返回结果过多");
        }

    }
     public void close() throws SQLException {
        simpleExcutor.close();
    }
}
```

Executor

```
public interface Executor {

    <E> List<E> query(Configuration configuration, MappedStatement
mappedStatement,Object[] param) throws Exception;

    void close() throws SQLException;
}
```

SimpleExecutor

```
public class SimpleExecutor implements Executor {
```

```java
    private Connection connection = null;

    public <E> List<E> query(Configuration configuration, MappedStatement
mappedStatement,Object[] param) throws SQLException, NoSuchFieldException,
IllegalAccessException, InstantiationException, IntrospectionException,
InvocationTargetException {

        //获取连接
         connection = configuration.getDataSource().getConnection();

        // select * from user where id = #{id} and username = #{username}
        String sql = mappedStatement.getSql();

        // 对sql进行处理
        BoundSql boundsql =  getBoundSql(sql);

        // select * from where id = ? and username = ?
        String finalSql = boundsql.getSqlText();

        //获取传入参数类型
        Class<?> paramterType = mappedStatement.getParamterType();

        //获取预编译preparedStatement对象
        PreparedStatement preparedStatement =
connection.prepareStatement(finalSql);
        List<ParameterMapping> parameterMappingList =
boundsql.getParameterMappingList();

        for (int i = 0; i < parameterMappingList.size(); i++) {
            ParameterMapping parameterMapping = parameterMappingList.get(i);
            String name = parameterMapping.getName();

            //反射
            Field declaredField = paramterType.getDeclaredField(name);
            declaredField.setAccessible(true);
            //参数的值
            Object o = declaredField.get(param[0]);
            //给占位符赋值
            preparedStatement.setObject(i+1,o);
        }

        ResultSet resultSet = preparedStatement.executeQuery();
        Class<?> resultType = mappedStatement.getResultType();
        ArrayList<E> results = new ArrayList<E>();
        while (resultSet.next()){
            ResultSetMetaData metaData = resultSet.getMetaData();
```

```
                E o = (E) resultType.newInstance();
                int columnCount = metaData.getColumnCount();
                    for (int i = 1; i <= columnCount; i++) {
                        //属性名
                        String columnName = metaData.getColumnName(i);
                        //属性值
                        Object value = resultSet.getObject(columnName);

                        //创建属性描述器，为属性生成读写方法
                        PropertyDescriptor propertyDescriptor = new
PropertyDescriptor(columnName, resultType);
                        //获取写方法
                        Method writeMethod = propertyDescriptor.getWriteMethod();
                        //向类中写入值
                        writeMethod.invoke(o,value);

                    }
                results.add(o);

            }
                return results;

    }

    @Override
    public void close() throws SQLException {
            connection.close();
    }

    private BoundSql getBoundSql(String sql) {
        // 标记处理类：主要是配合通用标记解析器GenericTokenParser类完成对配置文件等的解
析工作，其中TokenHandler主要完成处理
        ParameterMappingTokenHandler parameterMappingTokenHandler = new
ParameterMappingTokenHandler();
        //GenericTokenParser：通用的标记解析器，完成了代码片段中的占位符的解析，然后再根
据给定的标记处理器（TokenHandler）来进行表达式的处理
      //三个参数：分别为openToken（开始标记）、closeToken（结束标记）、handler（标记处
理器）
        GenericTokenParser genericTokenParser = new GenericTokenParser("#
{","}",parameterMappingTokenHandler);
        String parse = genericTokenParser.parse(sql);

        List<ParameterMapping> parameterMappings =
parameterMappingTokenHandler.getParameterMappings();

        BoundSql boundSql = new BoundSql(parse, parameterMappings);

        return boundSql;
    }
```

BoundSql

```
public class BoundSql {

    //解析过后的sql语句
    private String sqlText;

    //解析出来的参数
    private List<ParameterMapping> parameterMappingList  = new
ArrayList<ParameterMapping>();

    public BoundSql(String sqlText, List<ParameterMapping>
parameterMappingList) {
        this.sqlText = sqlText;
        this.parameterMappingList = parameterMappingList;
    }

    public String getSqlText() {
        return sqlText;
    }

    public void setSqlText(String sqlText) {
        this.sqlText = sqlText;
    }

    public List<ParameterMapping> getParameterMappingList() {
        return parameterMappingList;
    }

    public void setParameterMappingList(List<ParameterMapping>
parameterMappingList) {
        this.parameterMappingList = parameterMappingList;
    }
}
```

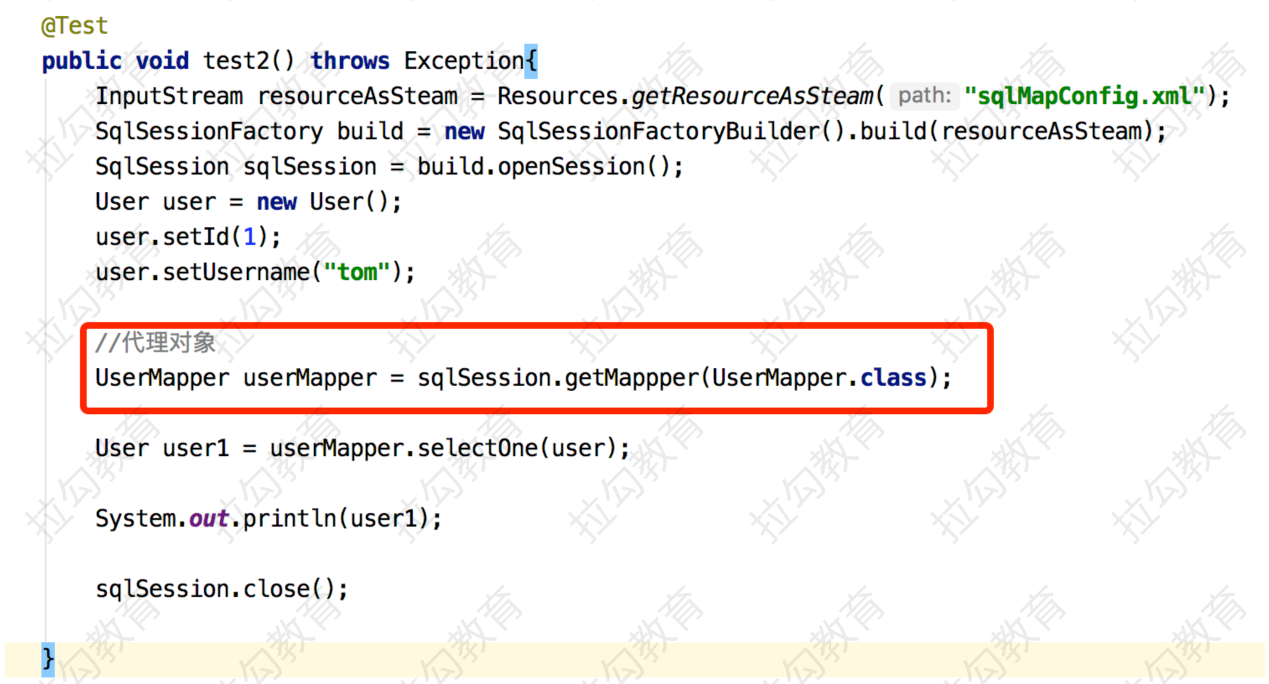
1.5 自定义框架优化

通过上述我们的自定义框架，我们解决了JDBC操作数据库带来的一些问题：例如频繁创建释放数据库连接，硬编码，手动封装返回结果集等问题，但是现在我们继续来分析刚刚完成的自定义框架代码，有没有什么问题?

问题如下：

- dao的实现类中存在重复的代码，整个操作的过程模板重复（创建sqlsession，调用sqlsession方法，关闭sqlsession）
- dao的实现类中存在硬编码，调用sqlsession的方法时，参数 statement的id硬编码

解决：使用代理模式来创建接口的代理对象



```java
@Test
public void test2() throws Exception{
    InputStream resourceAsSteam = Resources.getResourceAsSteam( path: "sqlMapConfig.xml");
    SqlSessionFactory build = new SqlSessionFactoryBuilder().build(resourceAsSteam);
    SqlSession sqlSession = build.openSession();
    User user = new User();
    user.setId(1);
    user.setUsername("tom");

    //代理对象
    UserMapper userMapper = sqlSession.getMappper(UserMapper.class);

    User user1 = userMapper.selectOne(user);

    System.out.println(user1);

    sqlSession.close();

}
```

在sqlSession中添加方法

```java
public interface SqlSession {
    public  <T> T getMappper(Class<?> mapperClass);
```

实现类

```java
@Override
public <T> T getMappper(Class<?> mapperClass) {
    T o = (T) Proxy.newProxyInstance(mapperClass.getClassLoader(), new Class[]
{mapperClass}, new InvocationHandler() {
        @Override
        public Object invoke(Object proxy, Method method, Object[] args)
throws Throwable {
            // selectOne
            String methodName = method.getName();
            // className:namespace
            String className = method.getDeclaringClass().getName();

            //statementid
            String key = className+"."+methodName;

            MappedStatement mappedStatement =
configuration.getMappedStatementMap().get(key);

            Type genericReturnType = method.getGenericReturnType();

            ArrayList arrayList = new ArrayList<> ();
            //判断是否实现泛型类型参数化
            if(genericReturnType instanceof ParameterizedType){
                return selectList(key,args);
```