```
/**
* @ClassName: JDBCDemo
* @Author: 99847
* @Description: JDBC连接配置
* @Date: 2021/6/14 19:27
* @Version: 1.0
*/
public class JDBCDemo {

    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.cj.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0个参数为sql语句中参数的序号(从1开始)，第0个参数为设置的参数值
            preparedStatement.setString(1, "tom ");
            // 向数据库发出sql执口查询，查询出结果集
            resultSet = preparedStatement.executeQuery();
            User user = new User();

            // 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();
                }
            }
        }
    }
}
```

存在问题

- 数据库连接配置存在硬编码问题
- 1. Sql语句在代码中硬编码，实际应DHsql优化的可能较D
  2. 使D preparedStatement向占有位符号传参数存在硬编码
- 对结果集解析存在硬编码(查询列名)系统不易维护
- 数据库连接创建、释放频繁造成系统资源浪费，从D影响系统性能。

解决思路

- 配置文件
- 配置文件
- 分别用不同的配置文件配置，因为数据库连接配置不需要频繁改动，sql语句配置会频繁变动
- 反射、内省技术
- 连接池(Pool)

自定义框架思路

- 读取配置D件
- 创建Resources类：getResourceAsSteam(String path)
- 解析配置D件
- 创建 SqlSessionFactoryBuilder 类：build(InputSteam in)
- dom4j解析配置文件，放到容器对象中
- 创建SqlSessionFactory对象
- 创建 SqlSessionFactory
- 生产SqlSession对象：openSession()
- 创建 sqlSession 接D及实现类
- 定义CRUD操作：selectOne(), update(), delete()等
- 创建 Executor 接D及实现类 SimpleExecutor
- 执行方法：query(Configuration conf, MappedStatement mapper, Object... params) 调用JDBC接口

自定义框架实现

实体类

```
public class Configuration {


    private DataSource dataSource;


    /**
     * key: statementId value:封装好的mappedStatement对象
     *
     */
    Map<String,MappedStatement> mappedStatementMap = new HashMap<>();


    public DataSource getDataSource() {
        return dataSource;
    }


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


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

```
// 连接数据库配置<configuration>

    <!--数据库配置信息-->
    <dataSource>
        <property name="driverClass" value="com.mysql.jdbc.Driver"></property>
        <property name="jdbcUrl" value="jdbc:mysql:///zdy_mybatis"></property>
        <property name="username" value="root"></property>
        <property name="password" value="root"></property>
    </dataSource>

    <!--存放mapper.xml的全路径-->
    <mapper resource="UserMapper.xml"></mapper>

</configuration>
```

连接数据库的配置信息

可以实现只调用一次getResources方法

读取配置

```
import java.io.InputStream;


public class Resources {

    // 根据配置文件的路径，将配置文件加载成字节输入流，存储在内存中
    public static InputStream getResourceAsSteam(String path){
    InputStream resourceAsStream = Resources.class.getClassLoader().getResourceAsStream(path);
    return resourceAsStream;
}
```

解析配置

```
// 单个表sql语句配置
<mapper namespace="com.lagou.dao.IUserDao">
    <!--sql的唯一标识：namespace.id来组成 ： statementId-->
    <select id="findAll" resultType="com.lagou.pojo.User" >
        select * from user
    </select>

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

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

同一个业务表的不同sql语句唯一标识

方便运用反射技术，实现实体类的自动封装

实体类

```
@Data
public class MappedStatement {

    //id标识
    private String id;
    //返回值类型
    private String resultType;
    //参数值类型
    private String paramterType;
    //sql语句
    private String sql;


}
```

```
public class SqlSessionFactoryBuilder {

    public SqlSessionFactory build(InputStream in) throws DocumentException, PropertyVetoException {
        // 第一：使用dom4j解析配置文件，将解析出来的内容封装到Configuration中
        XMLConfigBuilder xmlConfigBuilder = new XMLConfigBuilder();
        Configuration configuration = xmlConfigBuilder.parseConfig(in);


        // 第二：创建sqlSessionFactory对象：工厂类：生产sqlSession:会话对象
        DefaultSqlSessionFactory defaultSqlSessionFactory = new DefaultSqlSessionFactory(configuration);

        return defaultSqlSessionFactory;
    }
}
```

1. 解析配置文件

2. 创建sqlSessionFactory

```
public class DefaultSqlSessionFactory implements SqlSessionFactory {

    private Configuration configuration;

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


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

SqlSessionFactory 对象

```
public class DefaultSqlSession implements SqlSession {

    private Configuration configuration;

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

    @Override
    public <E> List<E> selectList(String statementid, Object... params) throws Exception {

        //将要去完成对simpleExecutor里的query方法的调用
        simpleExecutor simpleExecutor = new simpleExecutor();
        MappedStatement mappedStatement = configuration.getMappedStatementMap().get(statementid);
        List<Object> list = simpleExecutor.query(configuration, mappedStatement, params);

        return (List<E>) list;
    }
}
```

SqlSession：CRUD操作

Executor：调用JDBC接口

```
// Configuration 配置文件解析
public class XMLConfigBuilder {

    private Configuration configuration;

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

    /**
     * 该方法就是使用dom4j对配置文件进行解析，封装Configuration
     */
    public Configuration parseConfig(InputStream inputStream) throws DocumentException, PropertyVetoException {

        Document document = new SAXReader().read(inputStream);
        //<configuration>
        Element rootElement = document.getRootElement();
        List<Element> list = rootElement.selectNodes("//property");
        Properties properties = new Properties();
        for (Element element : list) {
            String name = element.attributeValue("name");
            String value = element.attributeValue("value");
            properties.setProperty(name,value);
        }

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

        configuration.setDataSource(comboPooledDataSource);

        //mapper.xml解析：拿到路径--字节输入流---dom4j进行解析
        List<Element> mapperList = rootElement.selectNodes("//mapper");

        for (Element element : mapperList) {
            String mapperPath = element.attributeValue("resource");
            InputStream resourceAsSteam = Resources.getResourceAsSteam(mapperPath);
            XMLMapperBuilder xmlMapperBuilder = new XMLMapperBuilder(configuration);
            xmlMapperBuilder.parse(resourceAsSteam);

        }
        return configuration;
    }
}
```

```
// Mapper配置文件解析
public class XMLMapperBuilder {

    private Configuration configuration;

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

    public void parse(InputStream inputStream) throws DocumentException {

        Document document = new SAXReader().read(inputStream);
        Element rootElement = document.getRootElement();

        String namespace = rootElement.attributeValue("namespace");

        List<Element> list = rootElement.selectNodes("//select");
        for (Element element : list) {
            String id = element.attributeValue("id");
            String resultType = element.attributeValue("resultType");
            String paramterType = element.attributeValue("paramterType");
            String sqlText = element.getTextTrim();
            MappedStatement mappedStatement = new MappedStatement();
            mappedStatement.setId(id);
            mappedStatement.setResultType(resultType);
            mappedStatement.setParamterType(paramterType);
            mappedStatement.setSql(sqlText);
            String key = namespace+"."+id;
            configuration.getMappedStatementMap().put(key,mappedStatement);

        }

    }
}
```

```
public class simpleExecutor implements Executor {



    @Override //user
    public <E> List<E> query(Configuration configuration, MappedStatement mappedStatement, Object... params) throws Exception {
    // 1. 注册驱动，获取连接
    Connection connection = configuration.getDataSource().getConnection();


    // 2. 获取sql语句 : select * from user where id = #{id} and username = #{username}
    //转换sql语句 ： select * from user where id = ? and username = ? ，转换的过程中，还需要对#{}里面的值进行解析存储
    String sql = mappedStatement.getSql();
    BoundSql boundSql = getBoundSql(sql);


    // 3.获取预处理对象：preparedStatement
    PreparedStatement preparedStatement = connection.prepareStatement(boundSql.getSqlText());


    // 4. 设置参数
    //获取到了参数的全路径
    String paramterType = mappedStatement.getParamterType();
    Class<?> paramtertypeClass = getClassType(paramterType);


    List<ParameterMapping> parameterMappingList = boundSql.getParameterMappingList();
    for (int i = 0; i < parameterMappingList.size(); i++) {
    ParameterMapping parameterMapping = parameterMappingList.get(i);
    String content = parameterMapping.getContent();


    //反射
    Field declaredField = paramtertypeClass.getDeclaredField(content);
    //暴力访问
    declaredField.setAccessible(true);
    Object o = declaredField.get(params[0]);


    preparedStatement.setObject(i+1,o);


    }



    // 5. 执行sql
    ResultSet resultSet = preparedStatement.executeQuery();
    String resultType = mappedStatement.getResultType();
    Class<?> resultTypeClass = getClassType(resultType);


    ArrayList<Object> objects = new ArrayList<>();


    // 6. 封装返回结果集
    while (resultSet.next()){
    Object o =resultTypeClass.newInstance();
    //元数据
    ResultSetMetaData metaData = resultSet.getMetaData();
    for (int i = 1; i <= metaData.getColumnCount(); i++) {


    // 字段名
    String columnName = metaData.getColumnName(i);
    // 字段的值
    Object value = resultSet.getObject(columnName);


    //使用反射或者内省，根据数据库表和实体的对应关系，完成封装
    PropertyDescriptor propertyDescriptor = new PropertyDescriptor(columnName, resultTypeClass);
    Method writeMethod = propertyDescriptor.getWriteMethod();
    writeMethod.invoke(o,value);



    }
    objects.add(o);


    }
    return (List<E>) objects;


    }


    private Class<?> getClassType(String paramterType) throws ClassNotFoundException {
    if(paramterType!=null){
    Class<?> aClass = Class.forName(paramterType);
    return aClass;
    }
    return null;


    }
```