

Mybatis概述

What框架? Why框架

1、框架即framework。

其实就是某种应用的半成品，就是一组组件，供你选用完成你自己的系统。

简单理解就是一套资源，包含jar包、源码、帮助文档、示例等。

2、为什么要用框架开发？

使用别人成熟的框架，就相当于让别人帮你完成一些基础工作，你只需要集中精力完成系统的业务逻辑设计。这样可以节省开发时间，提高代码重用性，让开发变得更简单。

Mybatis简介

MyBatis 是支持定制化 SQL以及高级映射(ORM)的优秀的持久层框架。

MyBatis 发展历史

Mybatis的前身是 Apache的一个开源项目 iBatis，2010年迁移到了google code 改名为 MyBatis，最后又迁移到了Github。

<http://www.mybatis.org/mybatis-3/zh/index.html>

mybatis

最近更新: 26 四月 2021 | 版本: 3.5.7

参考文档

- 简介
- 入门
- XML 配置
- XML 映射文件
- 动态 SQL
- Java API
- SQL 语句构造器
- 日志
- 项目文档
- 项目信息
- 项目报表

Build by 

 **MyBatis**

简介

什么是 MyBatis?

MyBatis 是一款优秀的持久层框架，它支持自定义 SQL、存储过程以及高级映射。MyBatis 免除了几乎所有的 JDBC 代码以及设置参数和获取结果集的工作。MyBatis 可以通过简单的 XML 或注解来配置和映射原始类型、接口和 Java POJO (Plain Old Java Objects, 普通老式 Java 对象) 为数据库中的记录。

帮助改进文档...

如果你发现文档有任何的遗漏，或缺少某一个功能点的说明，最好的解决办法是先自己学习，然后为遗漏的部分补上相应的文档。

该文档 xdoc 格式的源文件可通过[项目的 Git 代码库](#) 来获取。复制该源文件，作出更新，并提交 Pull Request 吧。

还有其他像你一样的人都渴望阅读这份文档，而你，就是这份文档最好的作者。

文档的翻译版本

您可以阅读 MyBatis 文档的其他语言版本：

-  English
-  Español
-  日本語
-  한국어
-  繁体中文

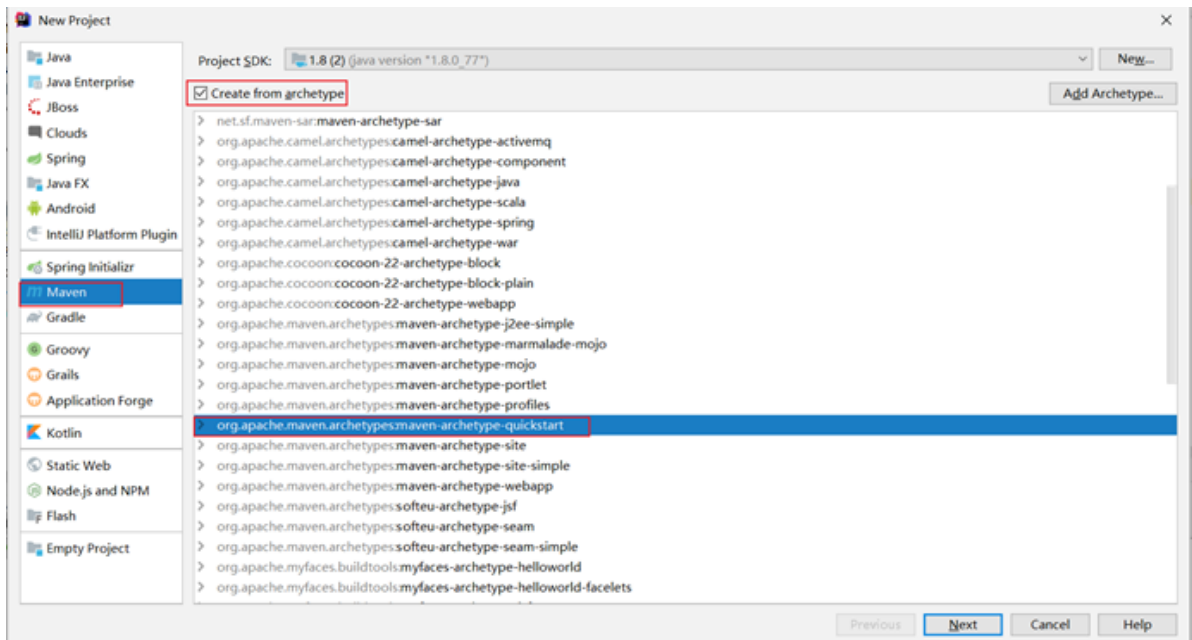
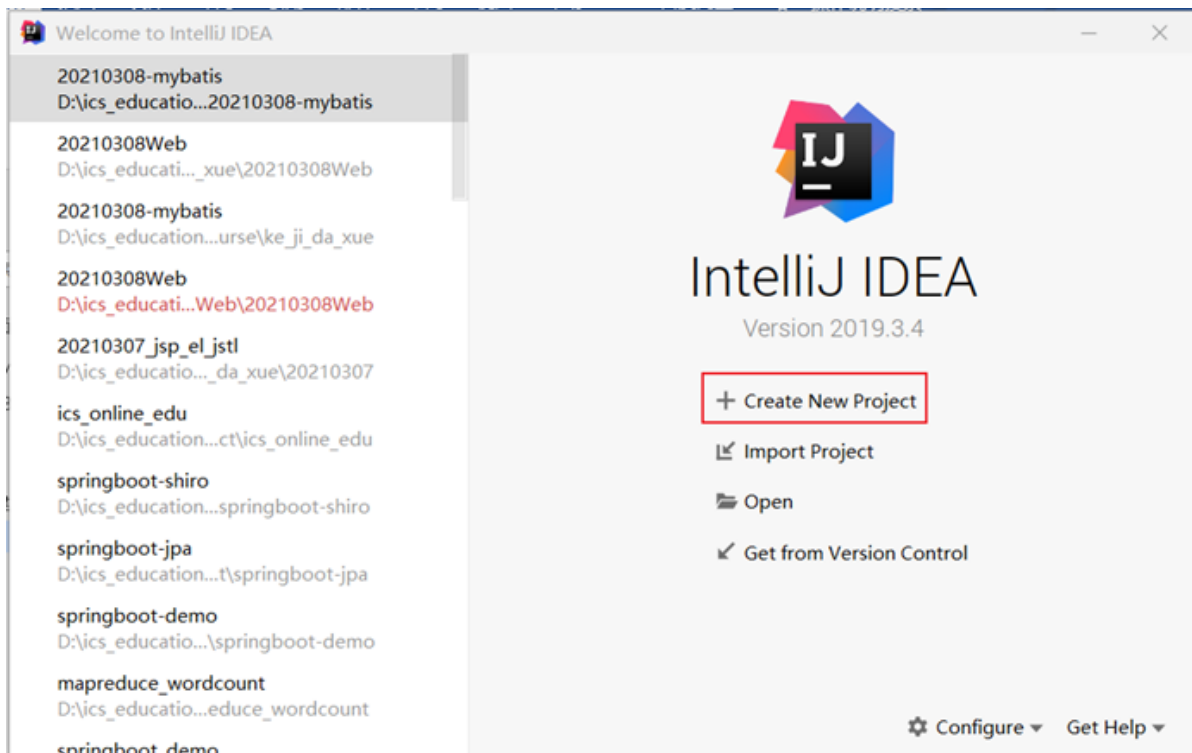
想用你的母语来了解 MyBatis 吗？那就将文档翻译成你的母语并提供给我们吧！

MyBatis的优点

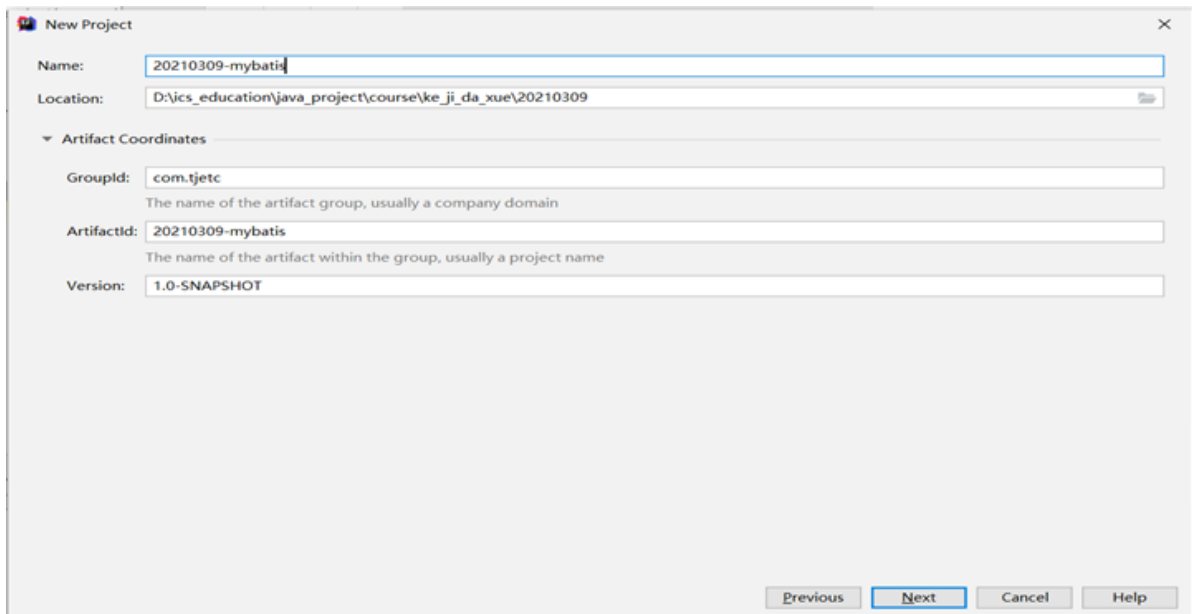
基于SQL语法，简单易学，能了解底层组装过程，SQL语句封装在配置文件中，便于统一管理与维护，降低了程序的耦合度，程序调试方便。

Mybatis配置

创建Mybatis工程



选择maven 输入组织名称、项目名称、版本号和工程名、保存位置



New Project

Name: 20210309-mybatis

Location: D:\ics_education\java_project\course\ke_ji_da_xue\20210309

Artifact Coordinates

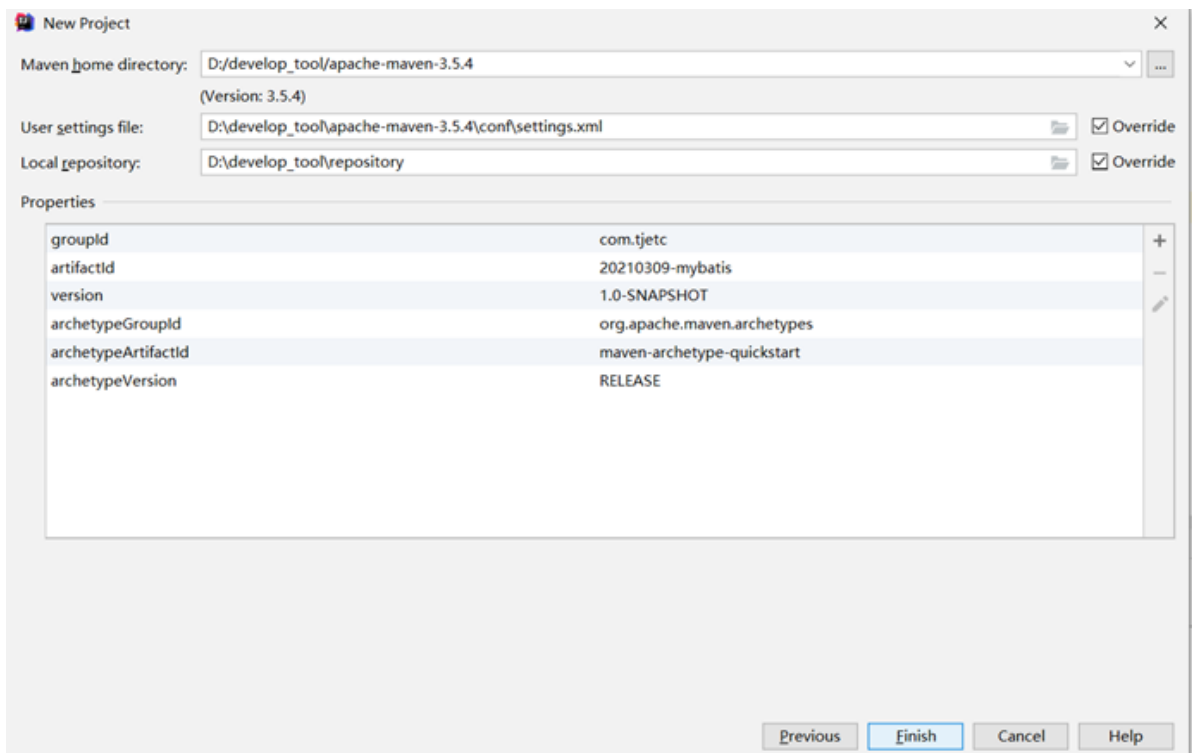
GroupId: com.tjetc
The name of the artifact group, usually a company domain

ArtifactId: 20210309-mybatis
The name of the artifact within the group, usually a project name

Version: 1.0-SNAPSHOT

Previous Next Cancel Help

核对信息并点击Finish按钮



New Project

Maven home directory: D:/develop_tool/apache-maven-3.5.4
(Version: 3.5.4)

User settings file: D:\develop_tool\apache-maven-3.5.4\conf\settings.xml ☒ Override

Local repository: D:\develop_tool\repository ☒ Override

Properties

| | | |
|---------------------|-----------------------------|---|
| groupId | com.tjetc | + |
| artifactId | 20210309-mybatis | - |
| version | 1.0-SNAPSHOT | |
| archetypeGroupId | org.apache.maven.archetypes | |
| archetypeArtifactId | maven-archetype-quickstart | |
| archetypeVersion | RELEASE | |

Previous Finish Cancel Help

引入依赖

```
<dependencies>
  <dependency>
    <groupId>org.mybatis</groupId>
    <artifactId>mybatis</artifactId>
    <version>3.5.13</version>
  </dependency>
  <dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.33</version>
  </dependency>
</dependencies>
```

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.12</version>
</dependency>
</dependencies>
```

mybatis.xml配置文件

mybatis配置顺序如下，不能颠倒，也就是如果有properties属性配置，一定放到最前面，settings放到第二个位置，以此类推

MyBatis 的配置文件包含了会深深影响 MyBatis 行为的设置和属性信息。配置文档的顶层结构如下：

- configuration (配置)
 - properties (属性)
 - settings (设置)
 - typeAliases (类型别名)
 - typeHandlers (类型处理器)
 - objectFactory (对象工厂)
 - plugins (插件)
 - environments (环境配置)
 - environment (环境变量)
 - transactionManager (事务管理器)
 - dataSource (数据源)
 - databaseIdProvider (数据库厂商标识)
 - mappers (映射器)

```
<?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>
  <settings>
    <!--指定 MyBatis 所用日志的具体实现,STDOUT_LOGGING表示控制台输出日志信息-->
    <setting name="logImpl" value="STDOUT_LOGGING"/>
  </settings>
  <!--配置环境（可以配置多个环境）-->
  <environments default="dev">
    <!--开发环境-->
    <environment id="dev">
      <!--使用JDBC事务管理-->
      <transactionManager type="JDBC"></transactionManager>
      <!--使用连接池技术-->
      <dataSource type="POOLED">
        <!--数据库驱动-->
        <property name="driver" value="com.mysql.cj.jdbc.Driver"/>
        <!--连接字符串-->
        <property name="url"
          value="jdbc:mysql://localhost:3306/test?
useUnicode=true&characterEncoding=utf8&useSSL=false&serverTimezone=A
sia/Shanghai"/>
        <!--数据库用户名-->
        <property name="username" value="root"/>
        <!--数据库密码-->
```

```

        <property name="password" value="123456"/>
    </dataSource>
</environment>
</environments>
<!--映射文件-->
<!--使用相对于类路径的资源引用-->
<mappers>
    <mapper resource="mapper/UserMapper.xml"></mapper>
</mappers>
</configuration>

```

写跟数据库表对应的java实体类

```

public class User {
    private Long id;
    private String username;
    private String password;

    public User() {
    }

    public User(Long id, String username, String password) {
        this.id = id;
        this.username = username;
        this.password = password;
    }

    public Long getId() {
        return id;
    }

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

    public String getUsername() {
        return username;
    }

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

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    @Override
    public String toString() {
        return "User{" +

```

```

        "id=" + id +
        ", username='" + username + '\'' +
        ", password='" + password + '\'' +
        '}';
    }
}

```

Mybatis调用接口Dao的接口方法，创建UserMapper接口

```

public interface UserMapper {
    List<User> queryList();
    User queryById(Long id);
    int insert(User user);
    int update(User user);
    int delete(Long id);
}

```

映射文件UserMapper.xml

映射文件位置为：src/main/resources/mapper/UserMapper.xml

```

<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-mapper.dtd" >
<mapper namespace="com.tjetc.mapper.UserMapper">
    <select id="queryList" resultType="com.tjetc.entity.User">
        select * from user
    </select>
    <select id="queryById" parameterType="long"
resultType="com.tjetc.entity.User">
        select * from user where id=#{id}
    </select>
    <insert id="insert" parameterType="com.tjetc.entity.User"
useGeneratedKeys="true" keyProperty="id">
        insert into user(username,password) values(#{username},#{password})
    </insert>
    <update id="update" parameterType="com.tjetc.entity.User">
        update user set username=#{username},password=#{password} where id=#{id}
    </update>
    <delete id="delete" parameterType="long">
        delete from user where id=#{id}
    </delete>
</mapper>

```

测试程序

```

public class TestUser {
    @Test
    public void testQueryList() throws IOException {
        //实例化SqlSessionFactoryBuilder对象
    }
}

```

```

        SqlSessionFactoryBuilder sqlSessionFactoryBuilder = new
SqlSessionFactoryBuilder();
        //创建SqlSessionFactory对象
        SqlSessionFactory sqlSessionFactory =
sqlSessionFactoryBuilder.build(Resources.getResourceAsReader("mybatis.xml"));
        //创建SqlSession对象
        SqlSession sqlSession = sqlSessionFactory.openSession();
        //得到mapper接口的实现对象
        UserMapper mapper = session.getMapper(UserMapper.class);
        List<User> users = mapper.queryList();
        //打印结果
        System.out.println(users);
        //关闭session
        sqlSession.close();
    }

    @Test
    public void testQueryById() throws IOException {
        //实例化SqlSessionFactoryBuilder对象
        SqlSessionFactoryBuilder sqlSessionFactoryBuilder = new
SqlSessionFactoryBuilder();
        //创建SqlSessionFactory对象
        SqlSessionFactory sqlSessionFactory =
sqlSessionFactoryBuilder.build(Resources.getResourceAsReader("mybatis.xml"));
        //创建SqlSession对象
        SqlSession sqlSession = sqlSessionFactory.openSession();
        //得到mapper接口的实现对象
        UserMapper mapper = session.getMapper(UserMapper.class);
        User user = mapper.queryById(1L);
        //打印结果
        System.out.println(user);
        //关闭session
        sqlSession.close();
    }

    @Test
    public void testInsert() throws IOException {
        //实例化SqlSessionFactoryBuilder对象
        SqlSessionFactoryBuilder sqlSessionFactoryBuilder = new
SqlSessionFactoryBuilder();
        //创建SqlSessionFactory对象
        SqlSessionFactory sqlSessionFactory =
sqlSessionFactoryBuilder.build(Resources.getResourceAsReader("mybatis.xml"));
        //创建SqlSession对象
        SqlSession sqlSession = sqlSessionFactory.openSession();
        //得到mapper接口的实现对象
        UserMapper mapper = session.getMapper(UserMapper.class);
        User user = new User("z1", "123");
        int rows = mapper.insert(1L);
        //提交事务
        session.commit();
        System.out.println(rows);
        //打印结果
        System.out.println(user);
        //关闭session
    }

```

```

        sqlSession.close();
    }

    @Test
    public void testUpdate() throws IOException {
        //实例化SqlSessionFactoryBuilder对象
        SqlSessionFactoryBuilder sqlSessionFactoryBuilder = new
        SqlSessionFactoryBuilder();
        //创建SqlSessionFactory对象
        SqlSessionFactory sqlSessionFactory =
        sqlSessionFactoryBuilder.build(Resources.getResourceAsReader("mybatis.xml"));
        //创建SqlSession对象
        SqlSession sqlSession = sqlSessionFactory.openSession();
        //得到mapper接口的实现对象
        UserMapper mapper = session.getMapper(UserMapper.class);
        User user = mapper.queryById(1L);
        user.setPassword("aaaaa");
        int rows = mapper.update(user);
        //提交事务
        session.commit();
        //打印结果
        System.out.println(rows);
        //关闭session
        sqlSession.close();
    }

    @Test
    public void testDelete() throws IOException {
        //实例化SqlSessionFactoryBuilder对象
        SqlSessionFactoryBuilder sqlSessionFactoryBuilder = new
        SqlSessionFactoryBuilder();
        //创建SqlSessionFactory对象
        SqlSessionFactory sqlSessionFactory =
        sqlSessionFactoryBuilder.build(Resources.getResourceAsReader("mybatis.xml"));
        //创建SqlSession对象
        SqlSession sqlSession = sqlSessionFactory.openSession();
        //得到mapper接口的实现对象
        UserMapper mapper = session.getMapper(UserMapper.class);
        int rows = mapper.delete(1L);
        //提交事务
        session.commit();
        //打印结果
        System.out.println(rows);
        //关闭session
        sqlSession.close();
    }
}

```

使用别名

在mybatis.xml文件中配置别名


```
<typeAliases>
    <!--使用别名-->
    <typeAlias type="com.tjetc.entity.User" alias/typeAlias>
</typeAliases>
```

UserMapper.xml resultMap="user"

这里的user大小没有影响

```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-mapper.dtd" >
<mapper namespace="com.tjetc.mapper.UserMapper">
    <select id="queryList" resultMap="user">
        select * from user
    </select>
    <select id="queryById" parameterType="int" resultMap="User">
        select * from user where id=#{id}
    </select>
    <insert id="insert" parameterType="User">
        insert into user(username,password) values(#{username},#{password})
    </insert>
    <update id="update" parameterType="User">
        update user set username=#{username},password=#{password} where id=#{id}
    </update>
    <delete id="delete" parameterType="int">
        delete from user where id=#{id}
    </delete>
</mapper>
```

测试

mybatis.xml配置详细说明

properties配置

resources目录下创建db.properties

```
driverName=com.mysql.cj.jdbc.Driver
url=jdbc:mysql://localhost:3306/test?
useUnicode=true&characterEncoding=utf8&useSSL=false&serverTimezone=Asia/Shanghai
username=root
password=123456
```

修改mybatis.xml配置

```
<?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>
    <properties resource="db.properties"></properties>
    <settings>
        <!--指定 MyBatis 所用日志的具体实现,STDOUT_LOGGING表示控制台输出日志信息-->
```

```

    <setting name="logImpl" value="STDOUT_LOGGING"/>
</settings>
<typeAliases>
    <!--使用别名-->
    <typeAlias type="com.tjetc.entity.User" alias="User"/></typeAlias>
</typeAliases>
<!--配置环境（可以配置多个环境）-->
<environments default="dev">
    <!--开发环境-->
    <environment id="dev">
        <!--使用JDBC事务管理-->
        <transactionManager type="JDBC"/></transactionManager>
        <!--使用连接池技术-->
        <dataSource type="POOLED">
            <!--数据库驱动-->
            <property name="driver" value="${driverName}"/>
            <!--连接字符串-->
            <property name="url" value="${url}"/>
            <!--数据库用户名-->
            <property name="username" value="${username}"/>
            <!--数据库密码-->
            <property name="password" value="${password}"/>
        </dataSource>
    </environment>
</environments>
<!--映射文件-->
<mappers>
    <mapper resource="mapper/UserMapper.xml"/></mapper>
</mappers>
</configuration>

```

typeAliases配置

类型别名是java类型的简写。它仅在XML配置文件中，用于简化合格的class名字

也可以指定一个包名，MyBatis 会在包名下面搜索需要的Java Bean

```

<typeAliases>
    <package name="com.tjetc.entity"/>
</typeAliases>

```

每一个在包 `com.tjetc.entity` 中的Java Bean，在没有注解的情况下，会使用 Bean 的首字母小写的非限定类名来作为它的别名。比如 `com.tjetc.entity.User` 的别名为 `user`；若有注解，则别名为其注解值，例如

```

@Alias("adm")
public class Admin{
    //....
}

```

下面是一些为常见的Java类型内建的类型别名。它们都是不区分大小写的，注意，为了应对原始类型的命名重复，采取了特殊的命名风格

| 别名 | 映射的类型 |
|------------|------------|
| _byte | byte |
| _long | long |
| _short | short |
| _int | int |
| _integer | int |
| _double | double |
| _float | float |
| _boolean | boolean |
| string | String |
| byte | Byte |
| long | Long |
| short | Short |
| int | Integer |
| integer | Integer |
| double | Double |
| float | Float |
| boolean | Boolean |
| date | Date |
| decimal | BigDecimal |
| bigdecimal | BigDecimal |
| object | Object |
| map | Map |
| hashmap | HashMap |
| list | List |
| arraylist | ArrayList |
| collection | Collection |
| iterator | Iterator |

typeHandlers

无论是mybatis设置一个参数给PreparedStatement，还是从PreparedStatement中返回ResultSet，都要面临一个问题：Java类型与JDBC类型之间的转换。

Java类型面向的是内存，JDBC类型面向的是各种关系型数据库，因此两者有很大的不同。

数据类型映射，是ORM框架中最关键的问题。

自定义类型映射器

你可以重写类型映射器，也可以创建自己的非标准类型映射器。

需要实现org.apache.ibatis.type.TypeHandler接口，或继承类org.apache.ibatis.type.BaseTypeHandler

```
@MappedJdbcTypes(JdbcType.VARCHAR)
public class ExampleTypeHandler extends BaseTypeHandler<String> {
    @Override
    public void setNonNullParameter(PreparedStatement ps, int i, String parameter, JdbcType jdbcType)
        throws SQLException {
        ps.setString(i, parameter);
        Log.logger.info("ExampleTypeHandler-->setNonNullParameter");
    }

    @Override
    public String getNullableResult(ResultSet rs, String columnName) throws SQLException {
        Log.logger.info("ExampleTypeHandler-->getNullableResult");
        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);
    }
}
```

```
<!-- mybatis-config.xml -->
<typeHandlers>
  <typeHandler handler="org.mybatis.example.ExampleTypeHandler"/>
</typeHandlers>
```

objectFactory配置

当mybatis每次创建一个新的结果集对象时，都需要使用objectFactory实例。

如果希望重写默认ObjectFactory的行为，可以自定义对象工厂

```
public class ExampleObjectFactory extends DefaultObjectFactory {
    private static final long serialVersionUID = 1L;

    @Override
    public <T> T create(Class<T> type) {
        Log.logger.info("ExampleObjectFactory-->create");
        return super.create(type);
    }

    @Override
    public <T> T create(Class<T> type, List<Class<?>> constructorArgTypes, List<Object> constructorArgs) {
        Log.logger.info("ExampleObjectFactory-->create2");
        return super.create(type, constructorArgTypes, constructorArgs);
    }

    @Override
    public void setProperties(Properties properties) {
        Log.logger.info("ExampleObjectFactory-->setProperties");
        super.setProperties(properties);
    }

    @Override
    public <T> boolean isCollection(Class<T> type) {
        Log.logger.info("ExampleObjectFactory-->isCollection");
        return Collection.class.isAssignableFrom(type);
    }
}
```

environments配置

1、mybatis允许你同时配置多个environments。如开发、测试、生产，可以同时配置三个环境。还可以同时对多个不同类型的数据库操作。

注意：虽然你可以同时配置多个环境，但是对于一个SqlSessionFactory实例，只能选择一个环境使用。因此，如果你想同时连接两个数据库，必须要创建两个SqlSessionFactory。

2、transactionManager配置

这里有两种事务管理类型可选，type=[JDBC | MANAGED]

JDBC：依赖于从dataSource返回的Connection对象，直接使用JDBC管理事务

MANAGED：由JavaEE容器管理整个事务

注意：如果使用Spring整合mybatis，无需配置TransactionManager，因为Spring会用自己的事务环境重写mybatis的配置

3、dataSource配置

使用标准数据源接口javax.sql.DataSource配置数据库的JDBC连接

这里有三种数据源类型可选：type="[UNPOOLED | POOLED | JNDI]"

- **UNPOOLED数据源**

这个数据源的实现会每次请求时打开和关闭连接。虽然有点慢，但对那些数据库连接可用性要求不高的简单应用程序来说，是一个很好的选择。性能表现则依赖于使用的数据库，对某些数据库来说，使用连接池并不重要，这个配置就很适合这种情形

- **POOLED**

采用数据库连接池配置，可以优化性能，是当前web项目配置首选。

- **JNDI**

这个数据源实现是为了能在如 EJB 或应用服务器这类容器中使用，容器可以集中或在外部配置数据源，然后放置一个JNDI上下文的数据源引用

databaseIdProvider配置

DatabaseIdProvider元素主要是为了支持不同厂商的数据库，即同时支持多个数据库

```
<databaseIdProvider type="DB_VENDOR">
  <property name="MySQL" value="mysql"/>
  <property name="Oracle" value="oracle" />
</databaseIdProvider>
```

这个配置非常有用，项目如何同时支持多种数据库？

传统做法是生成多套mapper文件，在mybatis.xml中配置使用那套映射文件。这个做法的很大缺陷是：多套映射文件中，会有很多接口的实现是相同的，如果代码修改，需要同时修改多套文件。这给开发额外增加了很大的工作量。

