Mybatis

1.Mybatis简介

MyBatis 是一款优秀的持久层框架，它支持定制化 SQL、存储过程以及高级映射。MyBatis 避免了几乎所有的 JDBC 代码和手动设置参数以及获取结果集。MyBatis 可以使用简单的 XML 或注解来配置和映射原生类型、接口和 Java 的 POJO（Plain Old Java Objects，普通老式 Java 对象）为数据库中的记录。Mybatis框架本质上就是对JDBC的封装。

2.为什么用框架

```
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?
serverTimezone=Asia/Shanghai", "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, "zhangsan");
            //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()){
                System.out.println(resultSet.getString("id")+"
"+resultSet.getString("username"));
            }
        } 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();
                }
            }

        }
    }
```

问题说明：

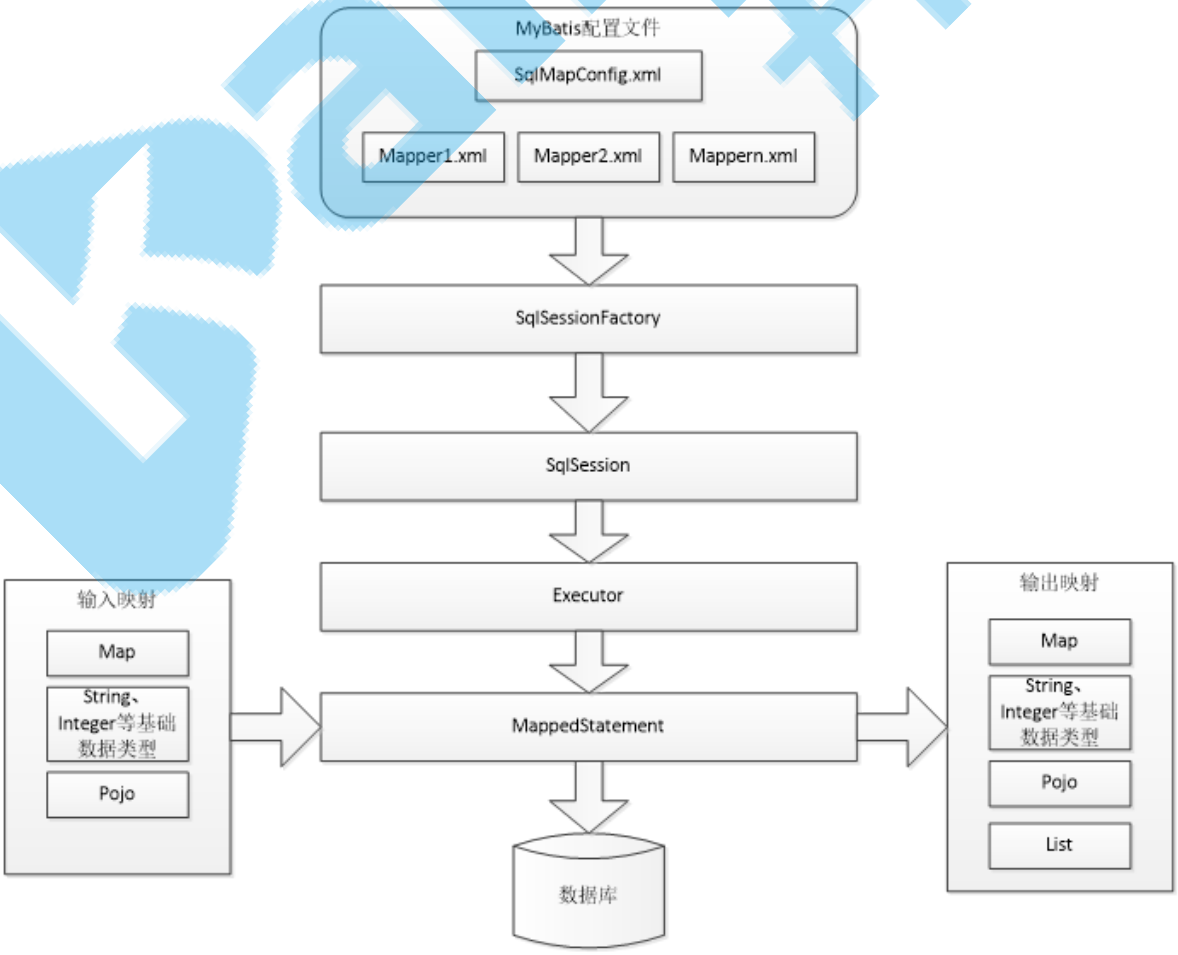
1、 数据库链接创建、释放频繁造成系统资源浪费从而影响系统性能，如果使用数据库链接池可解决此问题。

2、 Sql语句在代码中硬编码，造成代码不易维护，实际应用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。

3.架构说明



4.入门程序

4.1创建工程

maven依赖：

```
<properties>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
</properties>
<dependencies>
    <dependency>
        <groupId>org.mybatis</groupId>
        <artifactId>mybatis</artifactId>
        <version>3.5.3</version>
    </dependency>
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>8.0.17</version>
    </dependency>
    <dependency>
        <groupId>log4j</groupId>
        <artifactId>log4j</artifactId>
        <version>1.2.17</version>
    </dependency>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.12</version>
    </dependency>
</dependencies>
```

log4j.properties

```
# Global logging configuration
log4j.rootLogger=DEBUG, stdout
# Console output...
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%5p [%t] - %m%n
```

mybatis-config.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>
    <environments default="development">
        <environment id="development">
            <transactionManager type="JDBC"/>
            <dataSource type="POOLED">
                <property name="driver" value="com.mysql.cj.jdbc.Driver"/>
```

```
                <property name="url" value="jdbc:mysql:///mybatis?
serverTimezone=Asia/Shanghai"/>
                <property name="username" value="root"/>
                <property name="password" value="root"/>
            </dataSource>
        </environment>
    </environments>
    <mappers>
        <mapper resource="mapper/ProductMapper.xml"/>
    </mappers>
</configuration>
```

ProductMapper.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="test">
</mapper>
```

4.2查询一条数据

需求：从商品表中根据主键查询一个商品。

由于是根据主键查询那么一定是查询到一条数据或者没有数据。如果查询到一条数据，我们所期望的是能够直接返回一个java对象，这样我们使用起来就很方便，而不是返回resultSet对象。我们创建一个pojo类来表示一个商品。

```
public class Products {
    private int pid;
    private String pname;
    private float price;
    private Date pdate;
    private String cid;
    //getter、setter ...
}
```

编写查询商品数据的sql语句：

```
SELECT * FROM `products` where pid=?
```

其中pid是商品表的主键，对应的值应该是一个变量，所以使用`?`来表示，在jdbc中叫做`占位符`。我们把这个sql语句添加到Mapper文件中：

```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="test">
    <select id="selectProductById" parameterType="int"
resultType="com.kkb.pojo.Products">
        select * from products where pid=#{pid}
    </select>
</mapper>
```

在Mapper映射文件中，占位符使用`#{}`表示。

`parameterType`定义输入到sql中的映射类型

`resultType`定义结果映射类型。

测试程序：

```java
@Test
public void testSelectOne() throws Exception {
    //创建初始化builder
    SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();
    //加载配置文件
    Reader reader = Resources.getResourceAsReader("mybatis-config.xml");
    //创建会话工厂对象
    SqlSessionFactory factory = builder.build(reader);
    //从工厂对象获得一个连接
    SqlSession sqlSession = factory.openSession();
    //执行查询
    Products products = sqlSession.selectOne("selectProductById", 1);
    //打印结果
    System.out.println(products);
    //关闭连接
    sqlSession.close();
}
```

4.3查询多条数据

mapper文件：

```xml
<select id="selectProductByName" parameterType="string"
resultType="com.kkb.pojo.Products">
    SELECT * FROM `products` where pname like '%${name}%'
</select>
```

测试代码：

```java
    private SqlSessionFactory factory;

    @Before
    public void init() throws Exception {
        //创建初始化builder
        SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();
```

```
        //加载配置文件
        Reader reader = Resources.getResourceAsReader("mybatis-config.xml");
        //创建会话工厂对象
        factory = builder.build(reader);
    }

    @Test
    public void testSelectMore() throws Exception {
        SqlSession sqlSession = factory.openSession();
        List<Products> products = sqlSession.selectList("selectProductByName",
"新疆");
        products.forEach(p-> System.out.println(p));
    }
```

#{}和${}的区别：

> `#{}`表示一个占位符号，等同于jdbc中的`?`可以实现preparedStatement向占位符中设置值，自动进行java类型和jdbc类型转换。#{}可以接收简单类型值或pojo属性值。 如果parameterType传输单个简单类型值，#{}括号中可以是任意名称。
>
> `${}`表示字符串拼接，通过可以将$parameterType$传入的内容拼接在sql中且不进行$jdbc$类型转换，{}可以接收简单类型值或pojo属性值，如果parameterType传输单个简单类型值，${}括号中可以是任意名称。

4.4插入数据

mapper文件：

```
<insert id="addProduct" parameterType="com.kkb.pojo.Products">
    insert into products(pname, price, pdate, cid)
    values(#{pname},#{price},#{pdate},#{cid})
</insert>
```

测试代码：

```
@Test
public void addProduct() throws Exception {
    SqlSession sqlSession = factory.openSession();
    Products products = new Products();
    products.setPname("新疆葡萄干");
    products.setPrice(15);
    products.setPdate(new Date());
    products.setCid("s001");
    sqlSession.insert("addProduct", products);
    sqlSession.commit();
    sqlSession.close();
}
```

> SQLSession对象不是autoCommit的，如果不手动进行commit则回滚，这样就看不到刚添加的数据。所以使用Mybatis对数据库进行增删改操作后，需要手动commit。

我们使用mysql数据库时，通常主键使用自增主键，如果在一个处理中需要使用刚生成的主键，需要使用sql语句

```
SELECT LAST_INSERT_ID()
```

来获**当前事务**中最后生成的主键的值。在mybatis中可以一气呵成，插入之后可以直接返回生成的主键。那么久需要在`insert`节点中添加`selectKey`节点，来获得主键。

```
<insert id="addProduct" parameterType="com.kkb.pojo.Products">
    <selectKey resultType="int" keyProperty="pid" order="AFTER">
        select last_insert_id()
    </selectKey>
    insert into products(pname, price, pdate, cid)
    values(#{pname},#{price},#{pdate},#{cid})
</insert>
```

4.5修改数据

mapper文件：

```
<update id="updateProduct" parameterType="com.kkb.pojo.Products">
    update products set pname = #{pname},price=#{price},pdate=#{pdate},cid=#
{cid}
    where pid=#{pid}
</update>
```

测试代码：

```
@Test
public void updateProduct() throws Exception {
    SqlSession sqlSession = factory.openSession();
    Products products = sqlSession.selectOne("selectProductById", 9);
    products.setPname("新疆巴坦木");
    sqlSession.update("updateProduct", products);
    sqlSession.commit();
    sqlSession.close();
}
```

4.6删除数据

mapper文件：

```
<delete id="deleteProduct" parameterType="int">
    delete from products where pid=#{id}
</delete>
```

测试代码：

```
@Test
public void deleteProduct() throws Exception {
    SqlSession sqlSession = factory.openSession();
    sqlSession.delete("deleteProduct", 9);
    sqlSession.commit();
    sqlSession.close();
}
```

5.Mapper代理

5.1开发规范

Mapper接口开发方法只需要程序员编写Mapper接口，由Mybatis框架根据接口定义创建接口的动态代理对象，代理对象的方法体同上边Dao接口实现类方法。

Mapper接口开发需要遵循以下规范：

1. Mapper.xml文件中的namespace与mapper接口的类路径相同。
2. Mapper接口方法名和Mapper.xml中定义的每个statement的id相同
3. Mapper接口方法的输入参数类型和mapper.xml中定义的每个sql 的parameterType的类型相同
4. Mapper接口方法的输出参数类型和mapper.xml中定义的每个sql的resultType的类型相同

5.2接口

```
public interface ProductMapper {
    Products selectProductById(int pid);
    List<Products> selectProductByName(String name);
}
```

5.3 测试代码

```
@Test
public void testMapper() throws Exception {
    SqlSession sqlSession = factory.openSession();
    ProductMapper mapper = sqlSession.getMapper(ProductMapper.class);
    Products products = mapper.selectProductById(7);
    System.out.println(products);
    sqlSession.close();
}
```

5.4Mapper文件

```
<?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.kkb.mapper.ProductMapper">
    <select id="selectProductById" parameterType="int" resultType="com.kkb.pojo.Products">
        select * from products where pid=#{pid}
    </select>
    <select id="selectProductByName" parameterType="string" resultType="com.kkb.pojo.Products">
        SELECT * FROM `products` where pname like '%${name}%'
    </select>
    <insert id="addProduct" parameterType="com.kkb.pojo.Products">
        <selectKey resultType="int" keyProperty="pid" order="AFTER">
            select last_insert_id()
```

6.XML配置（了解）

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）

详细介绍：

https://mybatis.org/mybatis-3/zh/configuration.html#

6.1properties

在classpath下定义db.properties文件：

```
jdbc.driver=com.mysql.cj.jdbc.Driver
jdbc.url=jdbc:mysql:///mybatis?serverTimezone=Asia/Shanghai
jdbc.username=root
jdbc.password=root
```

SqlMapConfig.xml引用如下：

```
<?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"/>
    <environments default="development">
        <environment id="development">
            <transactionManager type="JDBC"/>
            <dataSource type="POOLED">
                <property name="driver" value="${jdbc.driver}"/>
                <property name="url" value="${jdbc.url}"/>
                <property name="username" value="${jdbc.username}"/>
                <property name="password" value="${jdbc.password}"/>
            </dataSource>
        </environment>
    </environments>
    <mappers>
        <mapper resource="mapper/ProductMapper.xml"/>
    </mappers>
</configuration>
```

> 注意： MyBatis 将按照下面的顺序来加载属性： 在 properties 元素体内定义的属性首先被读取。 然后会读取properties 元素中resource或 url 加载的属性，它会覆盖已读取的同名属性。

6.2 typeAliases

类型别名是为 Java 类型设置一个短的名字。 它只和 XML 配置有关，存在的意义仅在于用来减少类完全限定名的冗余。例如：

```
<typeAliases>
  <typeAlias alias="Author" type="domain.blog.Author"/>
  <typeAlias alias="Blog" type="domain.blog.Blog"/>
  <typeAlias alias="Comment" type="domain.blog.Comment"/>
  <typeAlias alias="Post" type="domain.blog.Post"/>
  <typeAlias alias="Section" type="domain.blog.Section"/>
  <typeAlias alias="Tag" type="domain.blog.Tag"/>
</typeAliases>
```

当这样配置时，`Blog` 可以用在任何使用 `domain.blog.Blog` 的地方。

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

```
<typeAliases>
  <package name="domain.blog"/>
</typeAliases>
```

每一个在包 `domain.blog` 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 `domain.blog.Author` 的别名为 `author`；若有注解，则别名为其注解值。见下面的例子：

```
@Alias("author")
public class Author {
    ...
}
```

这是一些为常见的 Java 类型内建的相应的类型别名。它们都是不区分大小写的，注意对基本类型名称重复采取的特殊命名风格。

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

6.3 mappers

我们现在就要定义 SQL 映射语句了。 但是首先我们需要告诉 MyBatis 到哪里去找到这些语句。 Java 在自动查找这方面没有提供一个很好的方法，所以最佳的方式是告诉 MyBatis 到哪里去找映射文件。 你可以使用相对于类路径的资源引用， 或完全限定资源定位符（包括 `file:///` 的 URL），或类名和包名等。例如：

```
<!-- 使用相对于类路径的资源引用 -->
<mappers>
  <mapper resource="org/mybatis/builder/AuthorMapper.xml"/>
  <mapper resource="org/mybatis/builder/BlogMapper.xml"/>
  <mapper resource="org/mybatis/builder/PostMapper.xml"/>
</mappers>
<!-- 使用完全限定资源定位符（URL） -->
<mappers>
  <mapper url="file:///var/mappers/AuthorMapper.xml"/>
  <mapper url="file:///var/mappers/BlogMapper.xml"/>
  <mapper url="file:///var/mappers/PostMapper.xml"/>
</mappers>
<!-- 使用映射器接口完全限定类名 -->
<mappers>
  <mapper class="org.mybatis.builder.AuthorMapper"/>
  <mapper class="org.mybatis.builder.BlogMapper"/>
  <mapper class="org.mybatis.builder.PostMapper"/>
</mappers>
<!-- 将包内的映射器接口实现全部注册为映射器 -->
<mappers>
  <package name="org.mybatis.builder"/>
</mappers>
```

> 如果使用映射器接口完全限定类名或者扫描包的形式加载mapper文件，那么久要求接口和mapper文件在同一个目录下，并且名称相同。

7.动态sql

7.1 if

如果想实现不固定条件查询，例如根据pojo对象的属性来生成查询条件，如果属性有值就增加一个查询条件，如果没有值就没有条件。要是想实现这样的功能就需要使用动态sql中的 `if` 节点

```
<select id="selectProductList" parameterType="products" resultType="products">
    select * from products
    where 0=0
    <if test="pid != null and pid != null">
        and pid = #{pid}
    </if>
    <if test="pname != null and pname != null">
        and pname like '%${pname}%'
    </if>
</select>
```

7.2 where

`where` 节点可以删除多余的 `and`

```
<select id="selectProductList" parameterType="products" resultType="products">
    select * from products
    <where>
        <if test="pid != null and pid != null">
            and pid = #{pid}
        </if>
        <if test="pname != null and pname != null">
            and pname like '%${pname}%'
        </if>
    </where>
</select>
```

7.3 foreach

向sql传递数组或List，使用`foreach`解析。例如动态生成`in`条件。

pojo类中增加一个集合属性，并添加get、set方法：

```
import java.util.Date;
public class Products {
    private int pid;
    private String pname;
    private float price;
    private Date pdate;
    private String cid;
    private String[] cids;
    //getter、setter ...

    public int getPid() { return pid; }

    public void setPid(int pid) { this.pid = pid; }
```

mapper文件中：

```
<select id="selectProductList" parameterType="products" resultType="products">
    select * from products
    <where>
        <if test="pid != null and pid != null">
            and pid = #{pid}
        </if>
        <if test="pname != null and pname != null">
            and pname like '%${pname}%'
        </if>
        <if test="cids !=null">
            <foreach collection="cids" open="and cid in(" close=")"
separator="," item="c">
                #{c}
            </foreach>
        </if>
    </where>
</select>
```

7.4 set

做update操作时，根据pojo属性是否有值来更新字段，可能会出现多余的“,”使用`set`节点来去掉多余的逗号。

```
<update id="updateProductSelected" parameterType="com.kkb.pojo.Products">
    update products
    <set>
        <if test="pname != null and pname != ''">
            pname = #{pname},
        </if>
        <if test="price != null and price != ''">
            price = #{price},
        </if>
        <if test="pdate != null">
            pdate = #{pdate},
        </if>
        <if test="cid != null and cid != ''">
            cid = #{cid},
        </if>
    </set>
    where pid=#{pid}
</update>
```

7.5 sql,include

Sql中可将重复的sql提取出来，使用时用include引用即可，最终达到sql重用的目的。

例如可以吧sql语句中的字段列表提取出来作为通用的sql片段。然后在sql语句中使用`include`节点引用这个sql片段。

```
<sql id="filedList">
    pid,pname,price,pdate,cid
</sql>
<select id="selectProductById" parameterType="int" resultType="Products">
    select
    <include refid="filedList"/>
    from products where pid=#{pid}
</select>
```

8.ResultMap

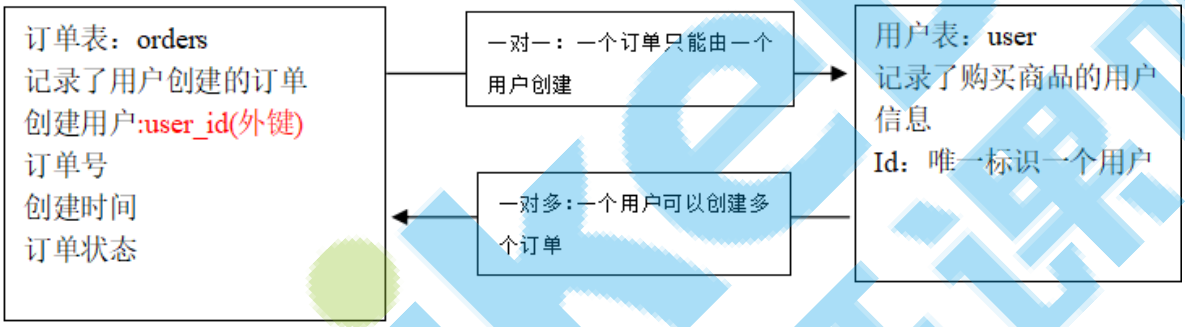
resultType可以指定pojo将查询结果映射为pojo，但需要pojo的属性名和sql查询的列名一致方可映射成功。如果sql查询字段名和pojo的属性名不一致，可以通过resultMap将字段名和属性名作一个对应关系 ，resultMap实质上还需要将查询结果映射到pojo对象中。resultMap可以实现将查询结果映射为复杂类型的pojo，比如在查询结果映射对象中包括pojo和list实现一对一查询和一对多查询。

8.1字段属性映射

```
<resultMap id="rmProduct" type="products">
```

```
    <id column="pro_id" property="pid"/>
    <result column="pro_name" property="pname"/>
    <result column="pro_price" property="price"/>
    <result column="pro_date" property="pdate"/>
    <result column="pro_cid" property="cid"/>
</resultMap>
<select id="selectProductWithMap" resultMap="rmProduct">
    select
    pid pro_id,
    pname pro_name,
    price pro_price,
    pdate pro_date,
    cid pro_cid
    from
    products
</select>
```

8.2关联关系说明



8.3一对一关联

查询所有订单信息，关联查询下单用户信息。使用resultMap，定义专门的resultMap用于映射一对一查询结果。

8.3.1 sql语句

```
SELECT
  orders.*,
  user.username,
  user.address
FROM
  orders,
  user
WHERE orders.user_id = user.id
```

8.3.2 pojo类

Users:

```
public class User {
    private int id;
    private String username;// 用户姓名
    private String sex;// 性别
    private Date birthday;// 生日
    private String address;// 地址
    //get\set...
}
```

在Orders类中加入User属性，user属性中用于存储关联查询的用户信息，因为订单关联查询用户是一对一关系，所以这里使用单个User对象存储关联查询的用户信息。

Orders：

```
public class Orders {
    private Integer id;
    private Integer userId;
    private String number;
    private Date createtime;
    private String note;
    private User user;
    //get\set...
}
```

8.3.3mapper文件

配置ResultMap映射：

```
<resultMap type="Orders" id="rmOrderUser">
    <id column="id" property="id"/>
    <result column="user_id" property="userId"/>
    <result column="number" property="number"/>
    <result column="createtime" property="createtime"/>
    <result column="note" property="note"/>
    <association property="user" javaType="User">
        <id column="user_id" property="id"/>
        <result column="username" property="username"/>
        <result column="address" property="address"/>
    </association>
</resultMap>
<select id="getOrders" resultMap="rmOrderUser">
    SELECT
    o.id,
    o.user_id,
    o.number,
    o.createtime,
    o.note,
    u.username,
    u.address
    FROM
    orders o
    JOIN `user` u ON u.id = o.user_id
</select>
```

8.4一对多关联

查询所有用户信息及用户关联的订单信息。用户信息和订单信息为一对多关系。

8.4.1 sql语句

```
SELECT
    u.*, o.id oid,
    o.number,
    o.createtime,
    o.note
FROM
    `user` u
LEFT JOIN orders o ON u.id = o.user_id
```

8.4.2 pojo类

在User类中添加一个orders属性，这个属性是一个集合属性，代表一个用户关联的多个订单信息。

```
public class User {
    private int id;
    private String username;// 用户姓名
    private String sex;// 性别
    private Date birthday;// 生日
    private String address;// 地址
    private List<Orders> orders;
}
```

8.4.3 Mapper文件

```
<resultMap type="user" id="rmUserOrder">
    <id property="id" column="id"/>
    <result property="username" column="username"/>
    <result property="birthday" column="birthday"/>
    <result property="sex" column="sex"/>
    <result property="address" column="address"/>
    <collection property="orders" ofType="orders">
        <id property="id" column="oid"/>
        <result property="number" column="number"/>
        <result property="createtime" column="createtime"/>
        <result property="note" column="note"/>
    </collection>
</resultMap>
<select id="getUserList" resultMap="rmUserOrder">
    SELECT
    u.*, o.id oid,
    o.number,
    o.createtime,
    o.note
    FROM
    `user` u
    LEFT JOIN orders o ON u.id = o.user_id
</select>
```

9.注解开发

最初设计时，MyBatis 是一个 XML 驱动的框架。配置信息是基于 XML 的，而且映射语句也是定义在 XML 中的。而到了 MyBatis 3，就有新选择了。MyBatis 3 构建在全面且强大的基于 Java 语言的配置 API 之上。这个配置 API 是基于 XML 的 MyBatis 配置的基础，也是新的基于注解配置的基础。注解提供了一种简单的方式来实现简单映射语句，而不会引入大量的开销。

> 不幸的是，Java 注解的的表达力和灵活性十分有限。尽管很多时间都花在调查、设计和试验上，最强大的 MyBatis 映射并不能用注解来构建

9.1单条件查询

在接口中定义一个方法，使用`@Select`注解标注，其中包含sql语句。只有一条查询条件时，参数前可以不用添加`@Param`注解。

```
public interface UserMapper {
    @Select("select * from kuser where id=#{id}")
    User getUserById(int id);
}
```

在mybatis-config.xml中加载此接口：

```
<mappers>
    <mapper class="com.kkb.mapper.UserMapper"/>
</mappers>
```

测试方法：

```
@Test
public void testGetUserById() throws Exception {
    SqlSession sqlSession = factory.openSession();
    UserMapper mapper = sqlSession.getMapper(UserMapper.class);
    User user = mapper.getUserById(1);
    System.out.println(user);
    sqlSession.close();
}
```

9.2多条件查询

```
@Select("select * from kuser where username like '%${name}%' and sex = #{sex}")
List<User> getUserList(@Param("name") String name, @Param("sex") int sex);
```

多条件查询时，每个参数前必须添加`@Param`注解

9.3插入数据

```
@Insert("insert into kuser(username, birthday, sex, address) values(#
{username},#{birthday},#{sex},#{address})")
void addUser(User user);
```

返回主键：

```
@Insert("insert into kuser(username, birthday, sex, address) values(#
{username},#{birthday},#{sex},#{address})")
@SelectKey(statement = "select last_insert_id()", keyProperty = "id", resultType
= int.class, before = false)
void addUser(User user);
```

9.4更新数据

```
@Update("update kuser set username = #{username}, sex=#{sex}, birthday=#
{birthday}, address=#{address} where id=#{id}")
void updateUser(User user);
```

9.5删除数据

```
@Delete("delete from kuser where id=#{id}")
void deleteUser(int id);
```

9.6动态SQL

```
@Select("<script>\n" +
        "SELECT *\n" +
        "FROM\n" +
        "    `kuser`\n" +
        "<where>\n" +
        "    <if test=\"id!=0\">\n" +
        "        and id = #{id}\n" +
        "    </if>\n" +
        "    <if test=\"username != '' and username != null\">\n" +
        "        and username = #{username}\n" +
        "    </if>\n" +
        "</where>\n" +
        "</script>")
List<User> findList(User user);
```

9.7ResultMap

```
@Results({
    @Result(id = true, column = "uid", property = "id"),
    @Result(id = false, column = "uname", property = "username"),
    @Result(id = false, column = "udate", property = "birthday"),
    @Result(id = false, column = "usex", property = "sex"),
    @Result(id = false, column = "uaddr", property = "address")
})
@Select("SELECT id uid,username uname,birthday udate,sex usex,address uaddr FROM
kuser")
List<User> selectAll();
```


```
List<Map> getEmps(Emp emp);

@Select("select * from emp where empno=#{empno}")
@Results(id = "eee", value = {
        @Result(property = "empno", column = "empno", id = true),
        @Result(property = "ename", column = "ename1"),
        @Result(property = "dept", column = "deptno", one = @One(select = "com.aaa.dao.IEmpDao.getDeptByDeptno", fetchType = FetchType.DEFAULT))
})
List<Emp> query1(Emp emp);

@Select("select * from emp where empno=#{empno}")
@ResultMap("eee")
List<Emp> query2(Emp emp);

List<Emp> getEmpByName(Emp emp);

@Select("<script>" +
        "select * from emp " +
        "<where>" +
        "<if test=\"ename!=null and ename!=''\">" +
        "and ename1=#{ename}" +
        "</if>" +
        "</where>" +
        "</script>")
@ResultMap("eee")
List<Emp> query3(Emp emp);

@Select("select * from dept where deptno=#{deptno}")
Dept getDeptByDeptno();
}
```

> 复杂的查询，例如一对一、一对多映射就不推荐使用注解了。直接在Mapper文件中编写更为方便。

10.逆向工程

10.1配置maven插件

```
<build>
    <plugins>
        <plugin>
            <groupId>org.mybatis.generator</groupId>
            <artifactId>mybatis-generator-maven-plugin</artifactId>
            <version>1.3.7</version>
            <configuration>
                <verbose>true</verbose>
                <overwrite>true</overwrite>
            </configuration>
        </plugin>
    </plugins>
</build>
```

10.2 添加 generatorConfig.xml

在resources目录下创建 generatorConfig.xml文件，内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE generatorConfiguration
        PUBLIC "-//mybatis.org//DTD MyBatis Generator Configuration 1.0//EN"
        "http://mybatis.org/dtd/mybatis-generator-config_1_0.dtd">
<generatorConfiguration>
    <!--导入属性配置-->
    <properties resource="db.properties"></properties>
    <!--指定特定数据库的jdbc驱动jar包的位置-->
    <classPathEntry location="${jdbc.location}"/>
    <context id="default" targetRuntime="MyBatis3">
        <!-- optional，旨在创建class时，对注释进行控制 -->
        <commentGenerator>
```

```
            <property name="suppressDate" value="true"/>
            <property name="suppressAllComments" value="true"/>
        </commentGenerator>
        <!--jdbc的数据库连接 -->
        <jdbcConnection
                driverClass="${jdbc.driver}"
                connectionURL="${jdbc.url}"
                userId="${jdbc.username}"
                password="${jdbc.password}">
        </jdbcConnection>
        <!-- 非必需，类型处理器，在数据库类型和java类型之间的转换控制-->
        <javaTypeResolver>
            <property name="forceBigDecimals" value="false"/>
        </javaTypeResolver>
        <!-- Model模型生成器,用来生成含有主键key的类，记录类 以及查询Example类
            targetPackage     指定生成的model生成所在的包名
            targetProject     指定在该项目下所在的路径
        -->
        <javaModelGenerator targetPackage="com.kaikeba.pojo"
targetProject="src/main/java">
            <!-- 是否允许子包，即targetPackage.schemaName.tableName -->
            <property name="enableSubPackages" value="false"/>
            <!-- 是否对model添加 构造函数 -->
            <property name="constructorBased" value="true"/>
            <!-- 是否对类CHAR类型的列的数据进行trim操作 -->
            <property name="trimStrings" value="true"/>
            <!-- 建立的Model对象是否 不可改变  即生成的Model对象不会有 setter方法，只有
构造方法 -->
            <property name="immutable" value="false"/>
        </javaModelGenerator>
        <!--mapper映射文件生成所在的目录 为每一个数据库的表生成对应的SqlMap文件 -->
        <sqlMapGenerator targetPackage="com.kaikeba.dao"
targetProject="src/main/resources">
            <property name="enableSubPackages" value="false"/>
        </sqlMapGenerator>
        <javaClientGenerator type="XMLMAPPER" targetPackage="com.kaikeba.dao"
targetProject="src/main/java">
            <!-- enableSubPackages:是否让schema作为包的后缀 -->
            <property name="enableSubPackages" value="false" />
        </javaClientGenerator>

        <table catalog="" tableName="category"></table>
        <table catalog="" tableName="orders"></table>
        <table catalog="" tableName="products"></table>
        <table catalog="" tableName="kuser"></table>
    </context>
</generatorConfiguration>
```

10.3生成

使用maven 插件命令生成逆向工程代码：

```
mybatis-generator:generate
```