MyBatis

MyBatis 实战

叩丁狼教育：任小龙

Simple is Beautiful

1. 第一章 走进 MyBatis

1.1. 楔子

学习 MyBatis 之前，我们可以通过从查看原来使用 JDBC 操作持久层的代码，再次重温一下 JDBC 的噩梦。

1.1.1. JDBC 编程

保存操作：

```
public void save(Student stu) {
    String sql = "INSERT INTO t_student(name,age) VALUES (?,?)";
    Connection conn = null;
    PreparedStatement ps  = null;
    try {
        conn = JdbcUtil.getConn();
        ps = conn.prepareStatement(sql);
        //设置占位符参数
        ps.setString(1, stu.getName());
        ps.setInt(2, stu.getAge());
        ps.executeUpdate();//没有参数
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        JdbcUtil.close(conn, ps, null);
    }
}
```

查询操作：

```
public Student get(Long id) {
    String sql = "SELECT * FROM t_student WHERE id = ?";
    Connection conn = null;
    PreparedStatement ps = null;
    ResultSet rs = null;
    try {
        conn = JdbcUtil.getConn();
        ps = conn.prepareStatement(sql);
        ps.setLong(1, id);
        rs = ps.executeQuery();//没有参数
        if (rs.next()) {
            Student stu = new Student();

            Long sid = rs.getLong("id");
            String name = rs.getString("name");
            Integer age = rs.getInt("age");
            stu.setId(sid);
            stu.setName(name);
            stu.setAge(age);
            return stu;
        }
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        JdbcUtil.close(conn, ps, rs);
    }
    return null;
}
```

1.1.2.框架

什么是框架，框架从何而来，为什么使用框架？

框架（framework）：

1，是一系列 jar 包，其本质是对 JDK 功能的拓展。

2，框架是一组程序的集合，包含了一系列的最佳实践，作用是解决某一个领域的问题。

1.1.3.最佳实践

最佳实践（Best Practice）：

实际上是无数程序员经历过无数次尝试之后，总结出来的处理特定问题的特定方法。

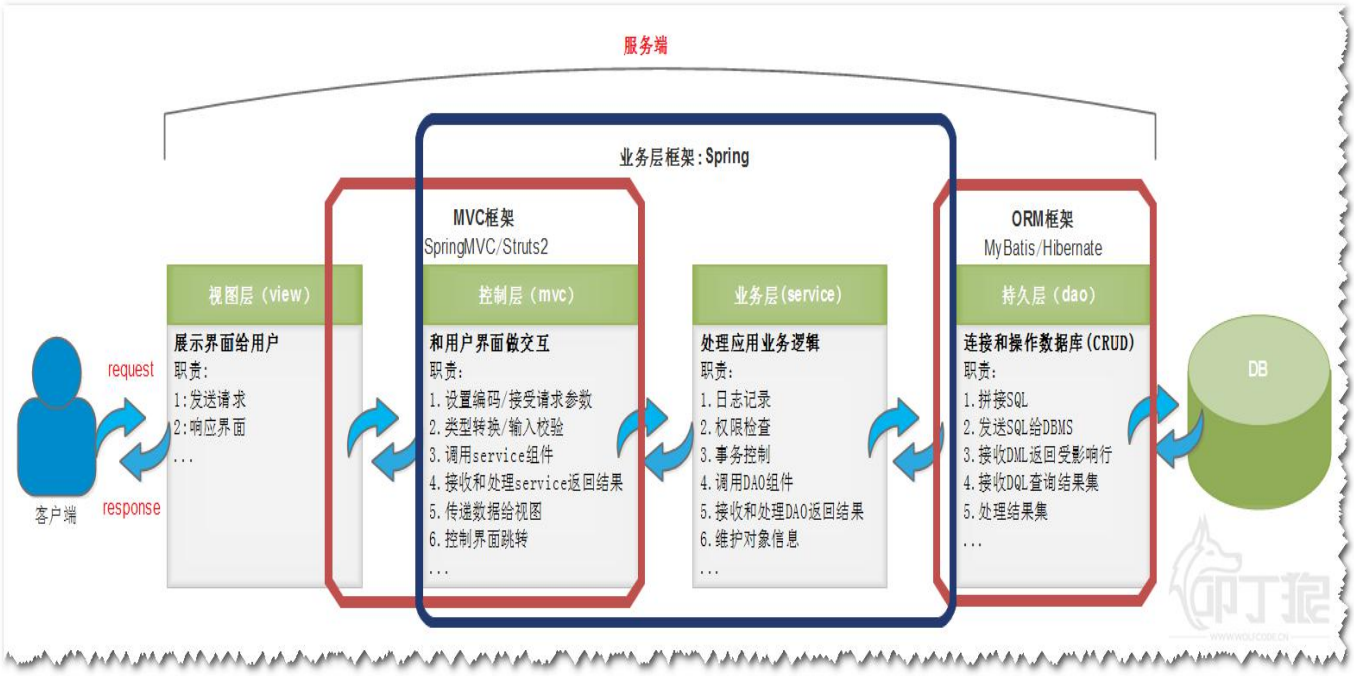
如果把程序员的自由发挥看作是一条通往成功的途径，最佳实践就是其中的最短路径，能极大的解放生产力。

Web 开发中的最佳实践：根据职责的纵向划分：控制层、业务层、持久层：

控制层 ：web/mvc: 负责处理与界面交互的相关操作 (Struts2/Spring MVC)

业务层 ：service: 负责复杂的业务逻辑计算和判断 (Spring)

持久层 ：dao: 负责将业务逻辑数据进行持久化存储 (MyBatis/Hibernate)



1.2. ORM 思想

借助之前编写的 JdbcTemplate 和 BeanHandler 以及 BeanListHandler 类。

需求：使用 JDBC 的方式查询 id 为 10 的记录并封装到 User 对象中。

此时在查询的时候，会发现如果对象中属性名称和表中列名不匹配，就不能使用通用的结果集处理器。

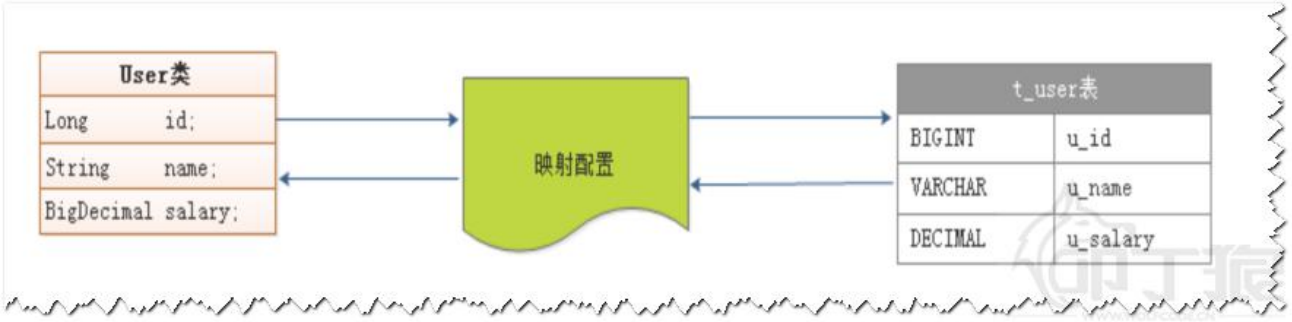
```
//把结果集中的一行数据封装成一个对象
public class BeanHandler<T> implements IResultSetHandler<T>{
    private Class<T> beanClass;

    public BeanHandler(Class<T> beanClass) {
        this.beanClass = beanClass;
    }
    public T handle(ResultSet rs) throws Exception {
        if(rs.next()){
            T obj = beanClass.newInstance();
            BeanInfo beanInfo = Introspector.getBeanInfo(beanClass,Object.class);
            PropertyDescriptor[] pds = beanInfo.getPropertyDescriptors();
            for (PropertyDescriptor pd : pds) {
                String name = pd.getName();//属性名称(列名)
                Object value = rs.getObject(name);
                //调用该对象的setter,把列的值设置到对象中指定的同名的属性中
                pd.getWriteMethod().invoke(obj, value);
            }
            return obj;
        }
        return null;
    }
}
```

1.2.1. ORM 思想

对象关系映射（Object Relational Mapping，简称 ORM）：

是一种为了解决面向对象与关系数据库存在的互不匹配的问题的技术。

简单的说，ORM 是通过使用描述对象和数据库之间映射的元数据，将 Java 程序中的对象自动持久化到关系数据库中。



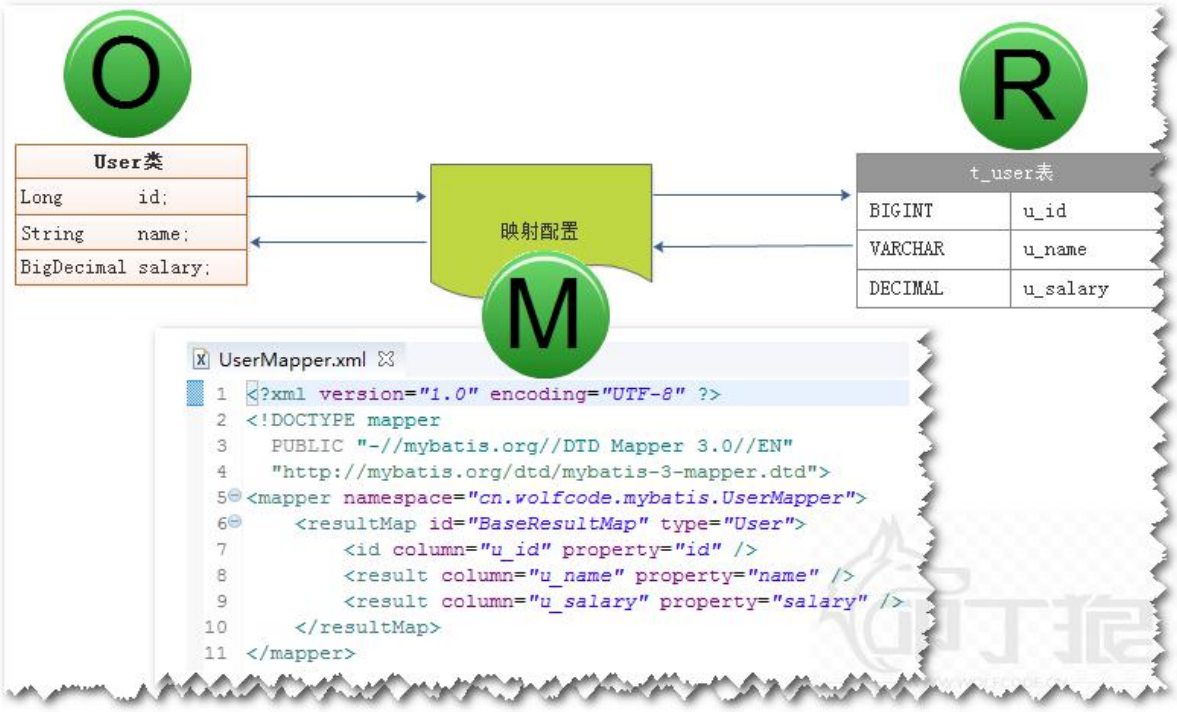
ORM 主要解决对象-关系的映射：

面向对象概念　　面向关系概念

类　　表

对象　　表的行（记录）

属性　　表的列（字段）



1.2.2. 常见 ORM 框架

把对象数据库的操作封装成一套 API，具有操作数据库的增删改查等操作，而且具有独立性，当持久层技术层发生改变时，不用修改任何业务层代码。

1，JPA：本身是一种 ORM 规范,不是 ORM 框架。由各大 ORM 框架提供实现。

2，Hibernate：目前最流行的 ORM 框架。设计灵巧,性能优秀,文档丰富。

3，MyBatis 目前最受欢迎的持久层解决方案。

等

1.3. MyBatis 概述

1.3.1.认识 MyBatis

MyBatis 是支持普通 SQL 查询，存储过程和高级映射的持久层框架，严格上说应该是一个 SQL 映射框架。

其前身是 iBatis，也就是淘宝使用的持久层框架。

几乎所有的 JDBC 代码和参数的手工设置以及结果集的处理都可以交给 MyBatis 完成，

而这只需要简单的使用 XML 或注解配置就可以完成。

和 Hibernate 相比更简单、更底层、性能更优异，因此更深入人心，更受企业的青睐。

GitHub 地址为 https://github.com/mybatis，

包括了 MyBatis 很多子项目：

mybatis-3:MyBatis 框架

generator：代码生成器

spring:MyBatis 和 Spring 集成的组件

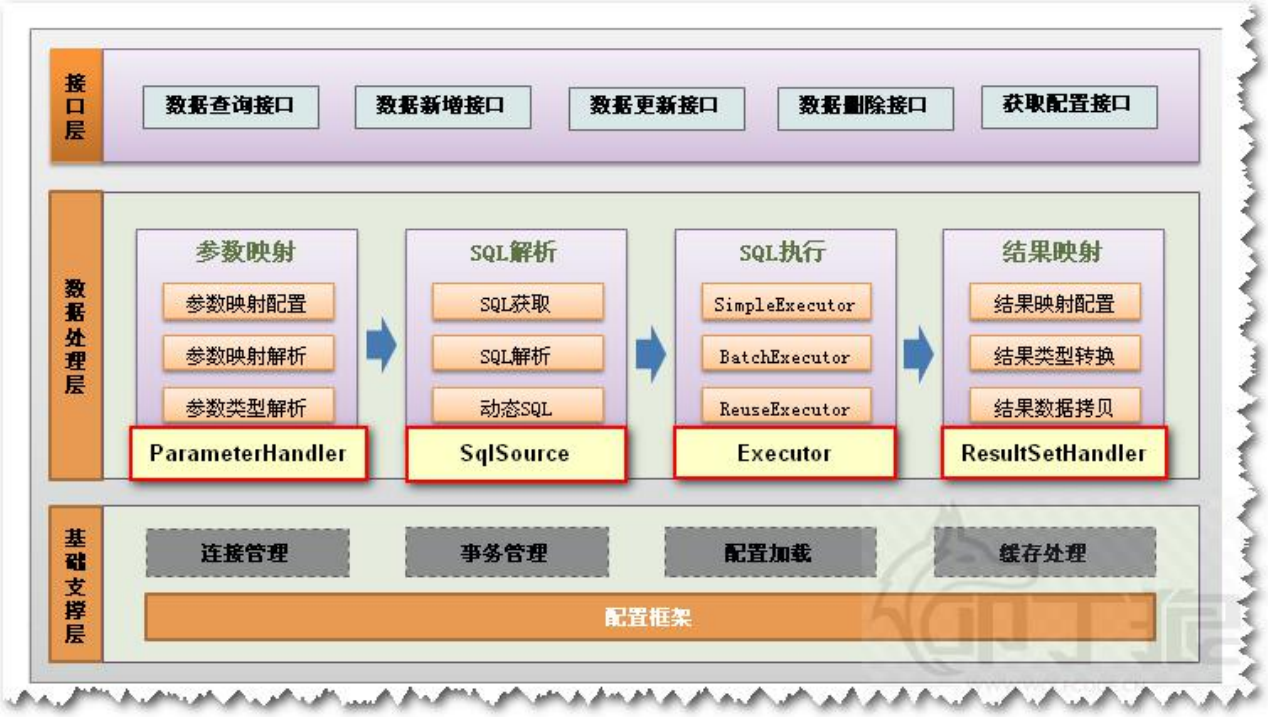
spring-boot-starter：MyBatis 和 Spring Boot 集成的组件

mybatipse：MyBatis 的 Eclipse 插件

中文文档：

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

1.3.2. MyBatis 架构图



1.3.3. MyBatis 核心组件

SqlSessionFactoryBuilder（构建器）：根据配置信息或 Java 代码来构建 SqlSessionFactory 对象。

作用：创建 SqlSessionFactory 对象。

SqlSessionFactory（会话工厂）：好比是 DataSource，线程安全的，在应用运行期间不要重复创建多次，建议使用单例模式。

作用：创建 SqlSession 对象

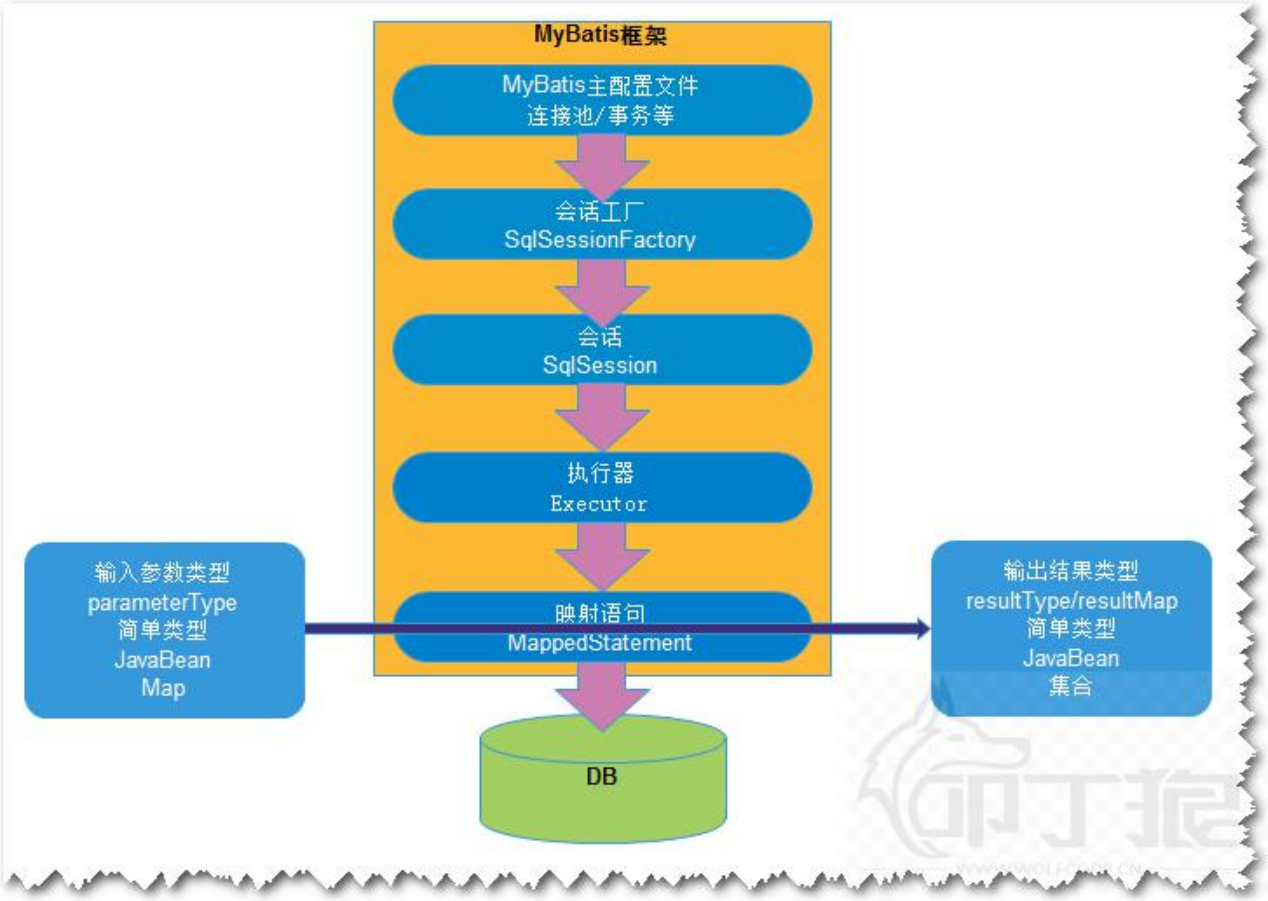
SqlSession（会话）：好比是 Connection，线程不安全的，每次使用开启新的 SqlSession 对象，使用完毕正常关闭，默认使用 DefaultSqlSession。提供操作数据库的增删改查方法，可以调用操作方法，也可以操作 Mapper 组件。

Executor（执行器）：

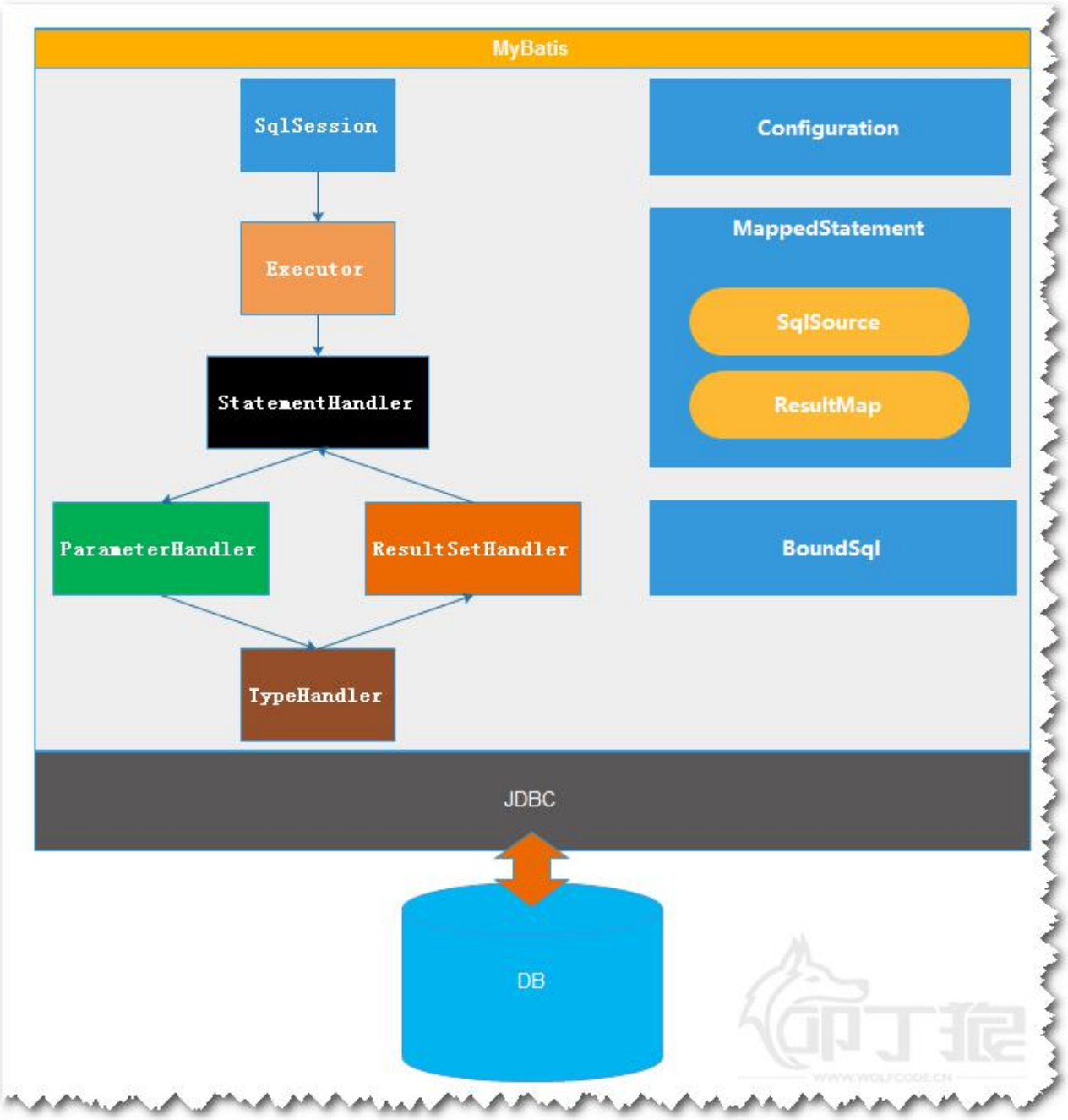
SqlSession 本身不能直接操作数据库，需要 Executor 来完成，该接口有两个实现：缓存执行器（缺省）、基本执行器。

MappedStatement：映射语句封装执行语句时的信息如 SQL、输入参数、输出结果。

1.3.4. MyBatis 原理图



更具体的底层原理图：



涉及到的对象讲解：

SqlSession：表示和数据库交互的会话，完成必要数据库增删改查功能。

Executor：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：语句处理器，封装了 JDBC 的 DML、DQL 操作、参数设置。

ParameterHandler：参数处理器，把用户传入参数转换为 JDBC 需要的参数值。

ResultSetHandler：结果集处理器，把结果集中的数据封装到 List 集合。

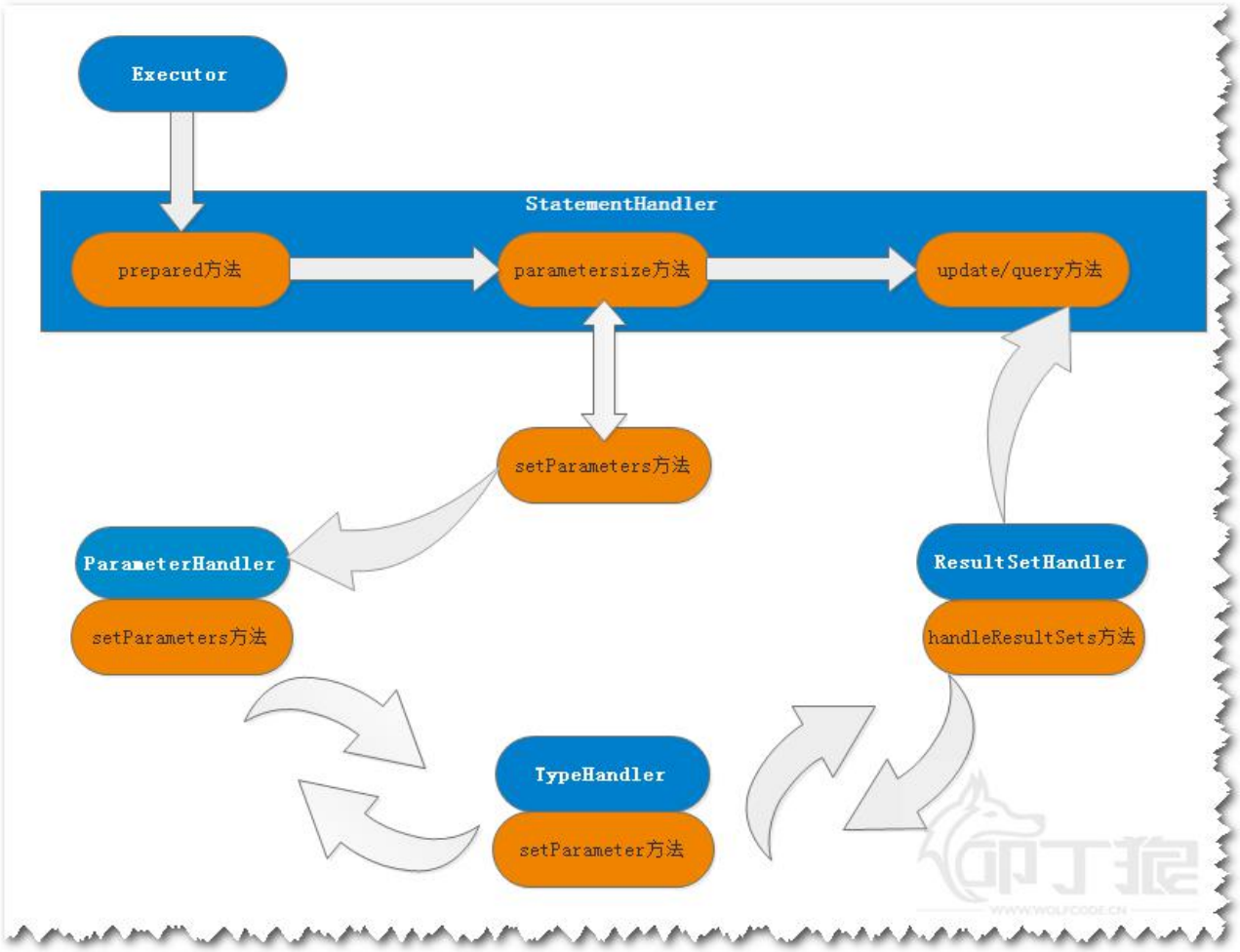
TypeHandler：类型转换器，Java 类型和 JDBC 类型的相互转换。

MappedStatement：映射语句对象，维护了一条<insert|update|delete|select>节点的封装。

SqlSource：SQL 源，根据用户传入的参数生成 SQL 语句，并封装到 BoundSql 中。

BoundSql：SQL 绑定，封装 SQL 语句和对应的参数信息。

Configuration：MyBatis 全局配置对象，封装所有配置信息。



2. 第二章 MyBatis 基础

开发准备：

MyBatis 依赖 jar 包:

1），MySQL 驱动包：mysql-connector-java-5.1.*.jar

2），核心包：mybatis-3.4.5.jar

3），其他依赖，lib 目录中所有的 jar（有需要，再拷贝）

操作表：

```
CREATE TABLE `t_user` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT,
  `name` varchar(20) DEFAULT NULL,
  `salary` decimal(8,2) DEFAULT NULL
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

操作对象：

```
@Data
public class User {
  private Long id;
  private String name;
  private BigDecimal salary;
}
```

2.1. 配置文件

MyBatis 自身的配置文件有两种：

一是、MyBatis 主配置文件，只有一份，名字任意，起名为 mybatis-config.xml

二是、MyBatis 的映射文件，有多份，名字一般 XxxMapper.xml，Xxx 表示模型对象。

2.1.1. 主配置文件

主配置文件主要包括数据库的信息，如连接池、事务等和全局的配置如关联映射日志、插件等。

```
<?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>
```

```
    <setting name="logImpl" value="LOG4J"/>
  </settings>
  <environments default="dev">
    <environment id="dev">
      <transactionManager type="JDBC" />
      <dataSource type="POOLED">
        <property name="driver" value="com.mysql.jdbc.Driver" />
        <property name="url" value="jdbc:mysql://localhost:3306/mybatisdemo" />
        <property name="username" value="root" />
        <property name="password" value="admin" />
      </dataSource>
    </environment>
  </environments>
  <mappers>
    <mapper resource="cn/wolfcode/mybatis/UserMapper.xml"/>
  </mappers>
</configuration>
```

2.1.2. 映射文件

映射文件只要是包含该对象的 CRUD 操作的配置和 SQL。

```
<?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="cn.wolfcode.mybatis.UserMapper">


</mapper>
```

2.1.3. Eclipse 提示 XML 语法

MyBatis 主配置文件的约束文件，mybatis-3.x.x.jar 中：

```
org/apache/ibatis/builder/xml/mybatis-3-config.dtd
```

MyBatis 映射文件的约束文件，mybatis-3.x.x.jar 中：

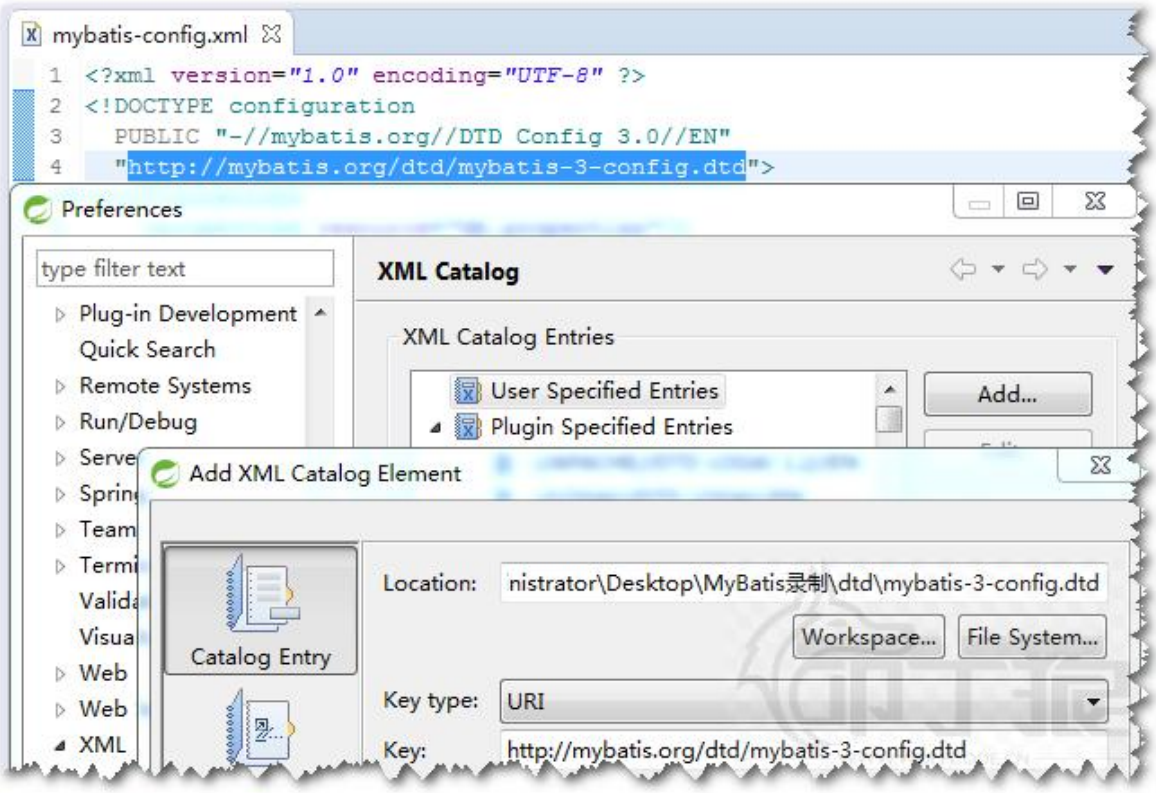
```
org/apache/ibatis/builder/xml/mybatis-3-mapper.dtd
```

XML 需要导入 dtd 约束，约束指向网络路径：

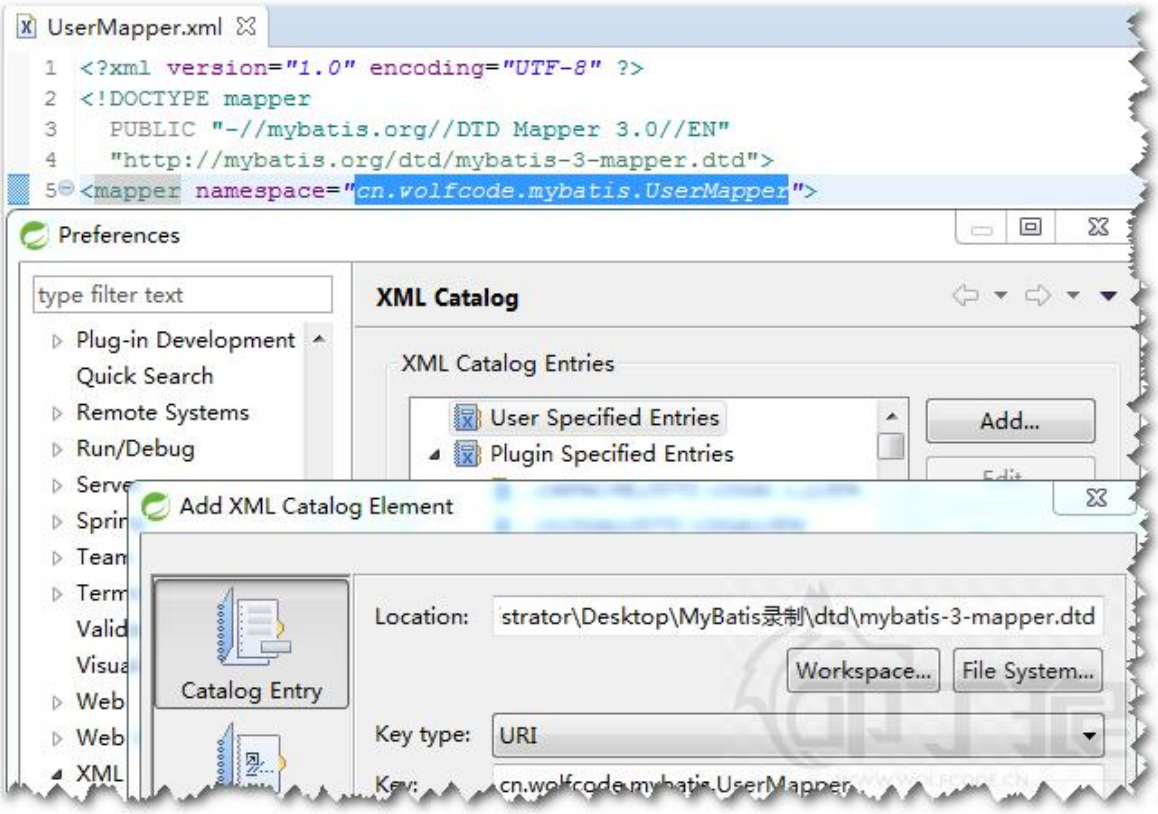
方式一、联网后，自动缓存路径文件到本地，提供提示功能；

方式二、无法联网 ，需要配置 dtd 文件位置（操作如下图）：

一、关联主配置文件


mybatis-config.xml
1 <?xml version="1.0" encoding="UTF-8" ?>
2 <!DOCTYPE configuration
3 PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
4 "http://mybatis.org/dtd/mybatis-3-config.dtd">
Preferences
type filter text
Plug-in Development
Quick Search
Remote Systems
Run/Debug
XML Catalog
XML Catalog Entries
User Specified Entries
Plugin Specified Entries
Add...
Add XML Catalog Element
Catalog Entry
Location:
nistrator\Desktop\MyBatis录制\dtd\mybatis-3-config.dtd
Workspace...
File System...
Key type:
URI
Key:
http://mybatis.org/dtd/mybatis-3-config.dtd

二、关联映射文件


UserMapper.xml
1 <?xml version="1.0" encoding="UTF-8" ?>
2 <!DOCTYPE mapper
3 PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
4 "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
5 <mapper namespace="cn.wolfcode.mybatis.UserMapper">
Preferences
type filter text
Plug-in Development
Quick Search
Remote Systems
Run/Debug
XML Catalog
XML Catalog Entries
User Specified Entries
Plugin Specified Entries
Add...
Add XML Catalog Element
Catalog Entry
Location:
strator\Desktop\MyBatis录制\dtd\mybatis-3-mapper.dtd
Workspace...
File System...
Key type:
URI
Key:
cn.wolfcode.mybatis.UserMapper

2.1.4.OGNL

OGNL（Object-Graph Navigation Language）对象图形导航语言，一种功能强大的表达式语言。

通过 OGNL，可以存取对象的属性和调用对象的方法，遍历整个对象的结构图等。

OGNL 可以直接获取根对象的属性，语法为：#{}：

如上下文的根对象为 employee 对象，则表达式

```
#{dept.name}
```

表示访问了 employee 对象的 dept 属性的 name 属性，就好比是：

```
employee.getDept().getName();
```

2.2. 日志框架

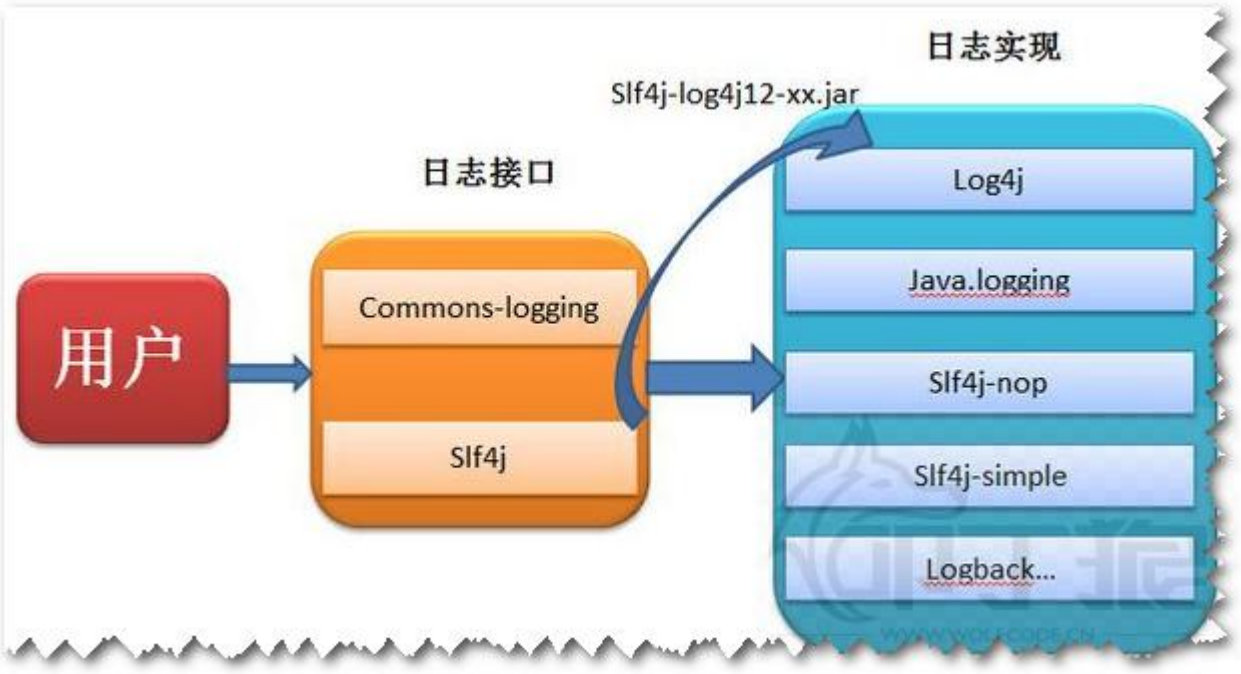
为什么要用日志？

1，比起 System.out.println，日志框架可以把日志的输出和代码分离；

2，日志框架可以方便的定义日志的输出环境，控制台，文件，数据库；

3，日志框架可以方便的定义日志的输出格式和输出级别；

2.2.1.日志江湖



常见的日志框架：

JDK-Logging：JDK1.4 版本以后开始提供的一个自带的日志库实现，太简单，不支持占位符显示，拓展性差，使用很少。

Commons-logging ：Apache 提供的日志规范，需要用户可以选择第三方的日志组件作为具体实现，本身会通过动态查找的机制找出真正日志的实现库。

Log4j：Apache 下功能非常丰富的日志库实现，功能强大，可以把日志输出到控制台、文件中，是出现比较早且最受欢迎日志组件。并且 Log4j 可以允许自定义日志格式和日志等级，帮助开发人员全方位的掌控日志信息。

Log4j2：是 Log4j 的升级，基本上把 Log4j 版本的核心全部重构，而且基于 Log4j 做了很多优化和升级。

SLF4J：好比是 Commons-logging 是日志门面，本身并无日志的实现，制定了日志的规范，使用时得拷贝整合包。

Logback：由 Log4j 创始人设计的另一个开源日志组件，也是作为 Log4j 的替代者出现的。

速度和效率都比 LOG4J 高，而且官方是建议和 SLF4j 一起使用，Logback、slf4j、Log4j 都是出自同一个人，所以默认对 SLF4J 无缝结合。

这里有个小故事：

当年 Apache 说服 Log4j 以及其他的日志框架按照 Commons-Logging 的标准来编写，但是由于 Commons-Logging 的类加载有点问题，实现起来不友好。因此 Log4j 的作者就创作了 SLF4j，也因此与 Commons-Logging 二分天下。

2.2.2. 日志级别

ERROR > WARN > INFO > DEBUG > TRACE

如果设置级别为 INFO，则优先级高于等于 INFO 级别（如：INFO、WARN、ERROR）的日志信息将可以被输出，小于该级别的如 DEBUG 和 TRACE 将不会被输出。

总结：日志级别越低，输出的日志越详细。

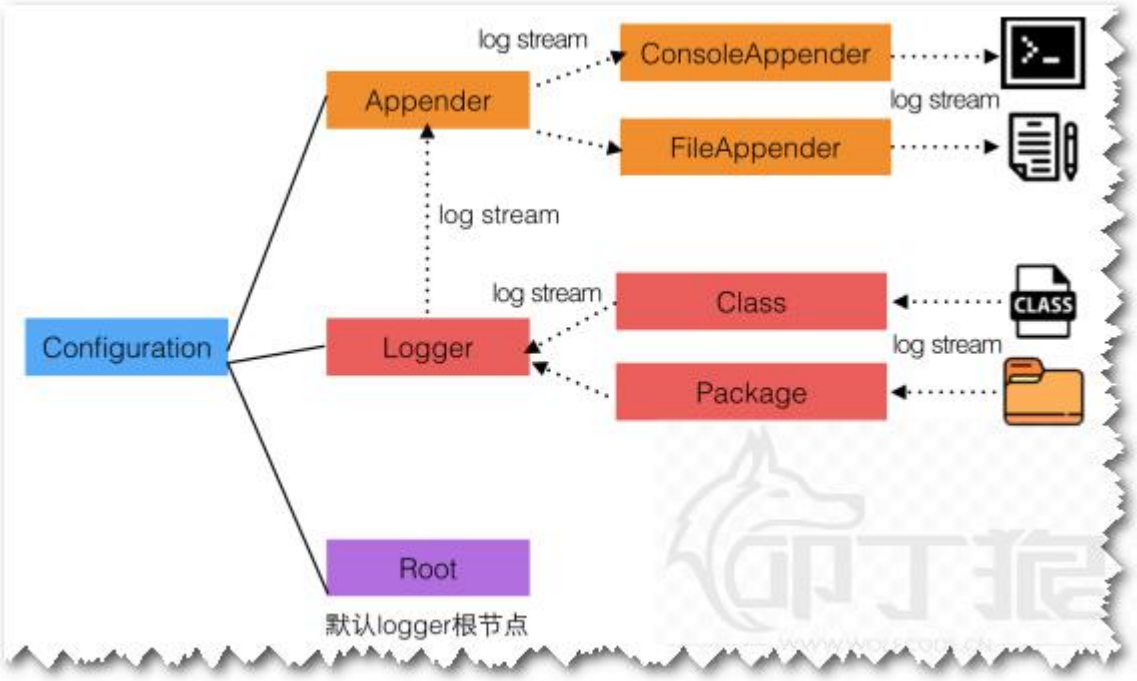
2.2.3. 日志文件

日志文件由三部分组成：

root：设置默认的日志输出级别和风格。

logger：设置自定义日志级别和风格。

appender：可以把日志输出到控制台或文件中去。



Log4.properties

```
# Global logging configuration
log4j.rootLogger=ERROR, stdout
# MyBatis logging configuration...
log4j.logger.cn.wolfcode.mybatis=TRACE
# 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
```

Log4j 采用类似 C 语言中的 printf 函数的打印格式格式化日志信息。打印参数如下：

%m：输出代码中指定的消息。

%p：输出优先级。

%r：输入自应用启动到输出该 log 信息耗费的毫秒数。

%c：输出所属的类目，通常就是所在类的全名。

%t：输出产生该日志线程的线程名。

%n：输出一个回车换行符。Windows 平台为“\r\n”，UNIX 为“\n”。

%d：输出日志时间点的日期或时间，默认格式为 ISO8601，推荐使用“%d{ABSOLUTE}”，这个输出格式形如：“2007-05-07 18:23:23,500”，符合中国人习惯。

%l：输出日志事件发生的位置，包括类名、线程名，以及所在代码的行数。

2.2.4. 选用日志框架

拷贝日志库:

```
log4j-1.2.17.jar
```

mybatis-config.xml 文件：

```
<configuration>
  <settings>
    <setting name="logImpl" value="LOG4J"/>
  </settings>
</configuration>
```

2.3. 保存操作

insert 元素用来封装插入操作的 SQL。

```
<insertid="" parameterType=””></insert>
```

parameterTypes 属性表示传入语句参数的类型，建议不配置，MyBatis 可以自行推导出来。

2.3.1.保存操作

Java 代码：

```
    @Test
    void testSave() throws Exception {
        User u = new User();
        u.setName("乔峰");
        u.setSalary(new BigDecimal("800"));

        InputStream config = Resources.getResourceAsStream("mybatis-config.xml");
        SqlSessionFactory factory = new SqlSessionFactoryBuilder().build(config);
        SqlSession session = factory.openSession();
        session.insert("cn.wolfcode.mybatis.UserMapper.save", u);
        session.commit();
        session.close();

        System.out.println(u);
    }
```

UserMapper.xml 文件：

```
    <insert id="save" parameterType="cn.wolfcode.mybatis.User">
        INSERT INTO t_user (name,salary) VALUES(#{name},#{salary})
    </insert>
```

2.3.2.获取自动生成主键

```
    <insert id="save" parameterType="cn。wolfcode。mybatis。User"
            useGeneratedKeys="true" keyProperty="id">
```

```
    INSERT INTO t_user (name,salary) VALUES(#{name},#{salary})
</insert>
```

userGeneratedKeys 属性 :MyBatis 是否使用 JDBC 的 getGeneratedKeys 方法获取数据库自动生成的主键，缺省值为 false。

keyProperty 属性：MyBatis 通过 getGeneratedKeys 方法获取主键值后给对象的哪一个属性值赋值。

keyColumn 属性：通过生成的键值设置表中的列名，PostgreSQL 中必须，MySQL 不需要设置。

2.4. 更新和删除操作

update 元素用来封装更新操作的 SQL

```
<update id=""></update>
```

delete 元素用来封装删除操作的 SQL

```
<delete id=""></delete>
```

2.4.1. 更新操作

Java 代码：

```
@Test
void testUpdate() throws Exception {
    User u = new User();
    u.setName("陆小凤");
    u.setSalary(new BigDecimal("800"));
    u.setHiredate(new Date());
    U.setId(3L);

    SqlSession session = MyBatisUtil.getSession();
    session.update("cn.wolfcode.mybatis.UserMapper.update", u);
    session.commit();
    session.close();
 }
```

UserMapper.xml 文件：

```
<update id="update">
    UPDATE t_user SET name = #{name},salary = #{salary},hiredate = #{hiredate}
     WHERE id = #{id}
</update>
```

2.4.2. 删除操作

Java 代码：

```
@Test
void testDelete() throws Exception {
    @Cleanup
    SqlSession session = MyBatisUtil.getSession();
    session.update("cn.wolfcode.mybatis.UserMapper.delete", 3L);
    session.commit();
}
```

UserMapper.xml 文件：

```
<update id="delete">
    DELETE FROM t_user SET  WHERE id = #{id}
</update>
```

2.5. 作用域和生命周期

理解我们目前已经讨论过的不同作用域和生命周期类是至关重要的，因为错误的使用会导致非常严重的并发问题。

提示 对象生命周期和依赖注入框架

依赖注入框架可以创建线程安全的、基于事务的 SqlSession 和映射器（mapper）并将它们直接注入到你的 bean 中，因此可以直接忽略它们的生命周期。如果对如何通过依赖注入框架来使用 MyBatis 感兴趣可以研究一下 MyBatis-Spring 或 MyBatis-Guice 两个子项目。

2.5.1. SqlSessionFactoryBuilder

这个类可以被实例化、使用和丢弃，一旦创建了 SqlSessionFactory，就不再需要它了。因此 SqlSessionFactoryBuilder 实例的最佳作用域是方法作用域（也就是局部方法变量）。你可以重用 SqlSessionFactoryBuilder 来创建多个 SqlSessionFactory 实例，但是最好还是不要让其一直存在以保证所有的 XML 解析资源开放给更重要的事情。

2.5.2. SqlSessionFactory

SqlSessionFactory 一旦被创建就应该在应用的运行期间一直存在，没有任何理由对它进行清除或重建。使用 SqlSessionFactory 的最佳实践是在应用运行期间不要重复创建多次，多次重建 SqlSessionFactory 被视为一种代码“坏味道（bad smell）”。因此 SqlSessionFactory 的最佳作用域是应用作用域。有很多方法可以做到，最简单的就是使用单例模式或者静态单例模式。

2.5.3. SqlSession

每个线程都应该有它自己的 SqlSession 实例。SqlSession 的实例不是线程安全的，因此是不能被共享的，所以它的最佳的作用域是请求或方法作用域。绝对不能将 SqlSession 实例的引用放在一个类的静态域，甚至一个类的实例变量也不行。也绝不能将 SqlSession 实例的引用放在任何类型的管理作用域中，比如 Servlet 架构中的 HttpSession。如果你现在正在使用一种 Web 框架，要考虑 SqlSession 放在一个和 HTTP 请求对象相似的作用域中。换句话说，每次收到的 HTTP 请求，就可以打开一个 SqlSession，返回一个响应，就关闭它。这个关闭操作是很重要的，你应该把这个关闭操作放到 finally 块中以确保每次都能执行关闭。下面的示例就是一个确保 SqlSession 关闭的标准模式：

```
SqlSession session = sqlSessionFactory.openSession();
try {
  //do work
} finally {
 session.close();
}
```

在你的所有的代码中一致性地使用这种模式来保证所有数据库资源都能被正确地关闭。

2.5.4. 映射器实例（Mapper Instances）

映射器是创建用来绑定映射语句的接口。映射器接口的实例是从 SqlSession 中获得的。因此从技术层面讲，映射器实例的最大作用域是和 SqlSession 相同的，因为它们都是从 SqlSession 里被请求的。尽管如此，映射器实例的最佳作用域是方法作用域。也就是说，映射器实例应该在调用它们的方法中被请求，用过之后即可废弃。并不需要显式地关闭映射器实例，尽管在整个请求作用域（request scope）保持映射器实例也不会有什么问题，但是很快你会发现，像 SqlSession 一样，在这个作用域上管理太多的资源的话会难于控制。所以要保持简单，最好把映射器放在方法作用域（method scope）内。下面的示例就展示了这个实践：

```
SqlSession session = sqlSessionFactory.openSession();
try {
   BlogMapper mapper = session.getMapper(BlogMapper.class);
   // do work
} finally {
   session.close();
}
```

2.5.5. MyBatis 工具类

```
public class MyBatisUtil {
  private MyBatisUtil() {
```

```
  }
  private static SqlSessionFactory factory = null;
  static {
    InputStream config;
    try {
       config = Resources.getResourceAsStream("mybatis-config.xml");
       factory = new SqlSessionFactoryBuilder().build(config);
    } catch (IOException e) {
       e.printStackTrace();
    }
  }

  public static SqlSession getSession() {
    return factory.openSession();
  }
}
```

2.6. 查询操作

select 元素用来封装查询操作的 SQL。

```
<select id="" resultType=""></select>
```

resultType 属性，表示使用什么类型的对象封装结果集中的每一行数据。

2.6.1. 查询单条数据

Java 代码：

```
@Test
void testGet() throws Exception {
   @Cleanup
   SqlSession session = MyBatisUtil.getSession();
   User u = session.selectOne("cn.wolfcode.mybatis.UserMapper.get", 1L);
   System.out.println(u);
}
```

UserMapper.xml 文件：

```
<select id="get" resultType="cn.wolfcode.mybatis.User">
   SELECT id,name,salary,hiredate FROM t_user WHERE id = #{id}
 </select>
```

2.6.2. 查询多条数据

Java 代码：

```
    @Test
    void testListAll() throws Exception {
        @Cleanup
        SqlSession session = MyBatisUtil.getSession();
        List<Object> list = session.selectList("cn.wolfcode.mybatis.UserMapper.listAll");
        for (Object o :  list) {
           System.out.println(o);
        }
    }
```

UserMapper.xml 文件：

```
    <select id="listAll" resultType="cn.wolfcode.mybatis.User">
        SELECT id,name,salary,hiredate FROM t_user
    </select>
```

3. 第三章 MyBatis 拓展

3.1. 别名配置处理

Mapper 文件中查询操作：

```
<select id="listAll" resultType="cn.wolfcode.mybatis.User">
    SELECT id,name,salary,hiredate FROM t_user
</select>
<select id="get" resultType="cn.wolfcode.mybatis.User">
    SELECT id,name,salary,hiredate FROM t_user WHERE id = #{id}
</select>
```

User 类型每次都使用的全限定名，可以使用类型别名。

别名（typeAlias）是一个类因为全限定名太长，使用一个简短名称去指代。

MyBatis 中别名不区分大小写。

3.1.1. 自定义别名

修改 mybatis-config.xml 文件：

直接给某一个类起别名：

```
<typeAlias type="cn.wolfcode.mybatis.User" alias="User"/>
```

给一个包（包含子包）中所有的类起别名。

```
<package name="cn.wolfcode.mybatis"/>
```

@Alias 注解可以用于设置类的别名。

修改 Mapper 文件：

```
<select id="listAll" resultType="User">
	SELECT id,name,salary,hiredate FROM t_user
</select>

<select id="get" resultType="User">
	SELECT id,name,salary,hiredate FROM t_user WHERE id = #{id}
</select>
```

3.1.2. 系统自带别名

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

这是一些为常见的 Java 类型内建的相应的类型别名。它们都是大小写不敏感的，需要注意的是由基本类型名称重复导致的特殊处理。

需求一、查询表中结果总数：

```
    <select id="queryCount" resultType="int">
            SELECT count(*) FROM t_user
    </select>
```

需求二、查询表中数据并封装到 map 中：

```
    <select id="queryMap" resultType="map">
            SELECT id AS uid,name AS uname,salary AS usalary FROM t_user
    </select>
```

3.2. 属性配置处理

3.2.1. property 元素

修改 mybatis-config.xml 文件，新增：

```
<properties>
    <property name="jdbc.driver" value="com.mysql.jdbc.Driver" />
    <property name="jdbc.url" value="jdbc:mysql://localhost:3306/mybatisdemo" />
    <property name="jdbc.username" value="root" />
    <property name="jdbc.password" value="admin" />
</properties>
```

在数据库配置中，使用$表达式获取属性值：

```
<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>
```

3.2.2. Properties 文件

新增 db.properties 文件：

```
jdbc.driver=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/mybatisdemo
jdbc.username=root
jdbc.password=admin
```

修改 mybatis-config.xml 文件：

```
<properties resource="db.properties"/>
```

3.3. 查询结果映射

在 CRUD 示例中 get 和 listAll 方法直接通过 resultType=“User”来设置 MyBatis 把结果集中的每一行数据包装为 User 对象，此时要求数据表的列名和对象的属性名要对应。

如果表中的列名和对象属性名不匹配，此时怎么办？

修改表中的列名，让列名和属性名不同 ：

```
CREATE TABLE `t_user` (
   `u_id` bigint(20) NOT NULL AUTO_INCREMENT,
   `u_name` varchar(20) DEFAULT NULL,
   `u_salary` decimal(8,2) DEFAULT NULL,
   PRIMARY KEY (`u_id`)
) ;
```

3.3.1.使用别名

处理结果集的时候使用的结果集中的列名，而并不是表中的列名。所以我们可以设置查询结果集中列的别名，把列的别名设置和对象中属性同名。

```
<select id="listAll" resultMap="BaseResultMap">
      SELECT u_id AS id,u_name AS name,u_salary AS salary FROM t_user
</select>

<select id="get" resultMap="BaseResultMap">
      SELECT u_id AS id,u_name AS name,u_salary AS salary FROM t_user WHERE u_id = #{id}
</select>
```

3.3.2.resultMap 元素

resultMap 元素是 MyBatis 中最重要最强大的元素。可以让我们远离 90%的需要从结果集中取出数据并处理结果集的代码， 且支持 JDBC 不支持的操作，比如对象关系匹配，这也许可以跨过上千行的代码。

```
<resultMap id="BaseResultMap" type="User">

</resultMap>
```

resultMap 元素定义了一个 ORM 的具体映射规则。

1，type 属性：最终返回的对象类型。

2，id 属性：该结果映射名称，这个名称就是在 get 或 listAll 中使用的 resultMap 对应的 id。

子元素 result：把结果集中的哪一个列设置到对象中的哪一个属性上，处理普通列。

子元素 id：和 result 功能相同，处理主键时使用 id 元素提升性能，处理主键列。

```
<resultMap id="BaseResultMap" type="User">
    <id column="u_id" property="id" />
    <result column="u_name" property="name" />
    <result column="u_salary" property="salary" />
</resultMap>

<select id="listAll" resultMap="BaseResultMap">
    SELECT u_id,u_name,u_salary FROM t_user
</select>

<select id="get" resultMap="BaseResultMap">
    SELECT u_id,u_name,u_salary FROM t_user WHERE u_id = #{id}
</select>
```

3.4. Mapper 组件

使用 namespace.id 的方式去找到 SQL 元素的方式,会有几个问题：

1，namespace.id 使用的是 String 类型，一旦编写错误，只有等到运行代码才能报错。

2，传入的实际参数类型不能被检查。

3，每一个操作的代码模板类似。

Mapper 组件：MyBatis 中的组件由 Java 接口和 XML 映射文件组成。

3.4.1. Mapper 接口

使用 Mapper 组件：

1，创建一个 Mapper 接口（类似于 DAO 接口），接口的要求：

①，这个接口的全限定名称——对应的 Mapper 文件的 namespace；

②，这个接口中的方法和 Mapper 文件中的 SQL 元素一一对应；

1，方法的名字对应 SQL 元素的 id；

2，方法的参数类型对应 SQL 元素中定义的 paramterType 类型（一般不写）；

3，方法的返回类型对应 SQL 元素中定义的 resultType/resultMap 类型；

2，创建 SqlSession；

3，通过 SqlSession.getMapper(XxxMapper.class)方法得到一个 Mapper 对象；

4，调用 Mapper 对象上的方法完成对象的 CURD；

```
public interface UserMapper {
  void save(User u);
  void update(User u);
  void delete(Long id);
  User get(Long id);
  List<User> listAll();
}
```

使用 Mapper 接口：

```
@Test
void testSave() throws Exception {
    User u  = new User();
    u.setName("乔峰");
    u.setSalary(new BigDecimal("800"));
    SqlSession session = MyBatisUtil.getSession();
    UserMapper mapper = session.getMapper(UserMapper.class);
    mapper.save(u);
    session.commit();
    session.close();
}
```

3.4.2. Mapper 接口原理

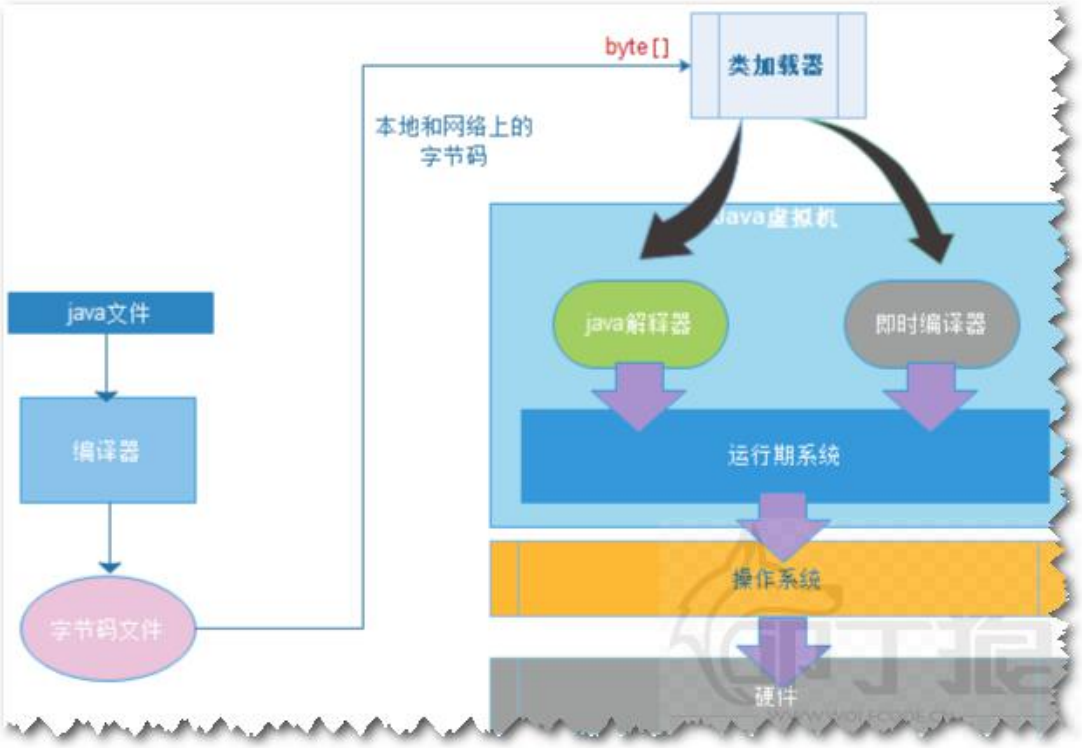
Mapper 的实现原理是动态代理 。

动态代理：动态代理类是在程序运行期间由 JVM 通过反射等机制动态的生成的，所以不存在代理类的字节码文件，代理对象和真实对象的关系是在程序运行时期才确定的。

分析字节码加载过程，思考如何动态的加载一份字节码。

由于 JVM 通过字节码的二进制信息加载类的，如果我们在运行期系统中，遵循 Java 编译系统组织.class 文件的格式和结构，生成相应的二进制数据，然后再把这个二进制数据加载转换成对应的类。

如此，就完成了在代码中动态创建一个类的能力了。




```
@Setter
public class MyMapperProxy<T> implements InvocationHandler {
    //需要被代理的接口
    private Class<T> mapperInterface;
    private SqlSession session;

    public T getProxyObject() {
        return (T) Proxy.newProxyInstance(
                mapperInterface.getClassLoader(),
                new Class[] { mapperInterface },
                this);
    }

    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        String namespaceName = mapperInterface.getName();
        String methodName = method.getName();
        String statementName  = namespaceName + "." + methodName;
        //根据不同的 Mapper 方法调用不同的方法
        return session.selectList(statementName);
    }
}
```

3.5. 参数处理

MyBatis 中的参数

在 SqlSession 中的 insert/update/delete/selectOne/selectList 方法只能至多传入一个参数。

如何完成一个登陆功能方法？

```
void login(String username,String password)
```

3.5.1.封装 POJO 和 Map

如果要在 MyBatis 中传入多个参数，此时可以把参数封装到 POJO 对象或 Map 中，在 Mapper 文件的 SQL 中#{}里面引用的就是对象的属性或 Map 里面的 key。

MyBatis 在#{}中参数流程：

1，会去 POJO 对象中，按照属性名或者 Map 的 key 去查询。

2，如果找不到，尝试直接把方法的实际参数作为查询参数值。

封装 POJO：

```
@Getter@Setter
public class LoginVO {
  private String usename;
  private String password;
}
```

Mapper 接口：

```
public interface UserMapper {
  User login1(LoginVO vo);
  User login2(Map<String,Object> map);
}
```

Mapper 文件：

```
<select id="login1" resultType="User">
    SELECT id,username,password FROM t_user
    WHERE username = #{username} AND password = #{password}
</select>

<select id="login2" resultType="User">
    SELECT id,username,password FROM t_user
    WHERE username = #{username} AND password = #{password}
</select>
```

3.5.2. Param 注解

如果要在 Mapper 接口上的一个方法中添加多个参数，此时一定要在每个参数前使用@Param 标签。

原理：MyBatis 底层自动把这些参数包装成一个 Map 对象，@Param 中的 value 就会作为这个 Map 的 key，对应的参数值，就会作为这个 Key 的 value。

Mapper 接口：

```
User login3(@Param("username") String username,
            @Param("password") String password);
```

Mapper 文件：

```
<select id="login3" resultType="User">
   SELECT id,username,password FROM t_user
   WHERE username = #{username} AND password = #{password}
</select>
```

3.5.3. 集合/数组

当传递一个 List 对象或数组对象参数给 MyBatis 时，MyBatis 会自动把它包装到一个 Map 中，

此时 List 对象会以 list 作为 key，数组对象会以 array 作为 key，也可以使用 Param 注解设置 key 名。

在动态 SQL 中 foreach 元素再讲。

3.6. 使用注解开发

之前的开发 MyBatis 都使用 XML 来做配置，也可以是直接来开发，考虑到写 SQL 的复杂性，推荐使用 XML。

3.6.1. Insert 注解

@Insert：定义 INSERT 方法；配合@Options 使用;

```
@Insert({ "INSERT INTO t_user (name,salary) VALUES(#{name},#{salary})" })
@Options(useGeneratedKeys = true, keyProperty = "id")
void save(User u);
```

3.6.2. Delete 注解

```
@Delete("DELETE FROM t_user SET WHERE id = #{id}")
void delete(Long id);
```

3.6.3. Update 注解

```
@Update("UPDATE t_user SET name = #{name},salary = #{salary} WHERE id = #{id}")
void update(User u);
```

3.6.4. Select 注解

```
@Select("SELECT id AS u_id,name AS u_name ,salary AS u_salary FROM t_user WHERE id = #{id}")
@ResultMap("BaseResultMap")
User get(Long id);

@Select("SELECT id AS u_id,name AS u_name ,salary AS u_salary FROM t_user")
@Results(id = "BaseResultMap", value = { //
      @Result(column = "u_id", property = "id"), //
      @Result(column = "u_name", property = "name"), //
      @Result(column = "u_salary", property = "salary")//
})
List<User> listAll();
```

3.7. 其他

3.7.1. #和$

1：通过#和$都可以获取对象中的属性。

2：区别：

使用#传递的参数会先转换为，无论传递是什么类型数据都会带一个单引号。

使用$传递的参数，直接把值作为 SQL 语句的一部分。

使用$容易导致 SQL 注入问题 ----->使用#更安全。

但是在做排序的时候，如果排序列存在引号，就不能排序。

结论：

如果需要设置占位符参数全部使用#，也就是 SQL 中可以使用?的地方。

如果需要拼接成 SQL 的一部分使用$，比如排序。

如：

```
SELECT id ,name ,salary  FROM t_user
where name  = #{name} AND salary >= #{salary} ORDER BY ${orderBy}
```

3.7.2. MyBatipse 插件

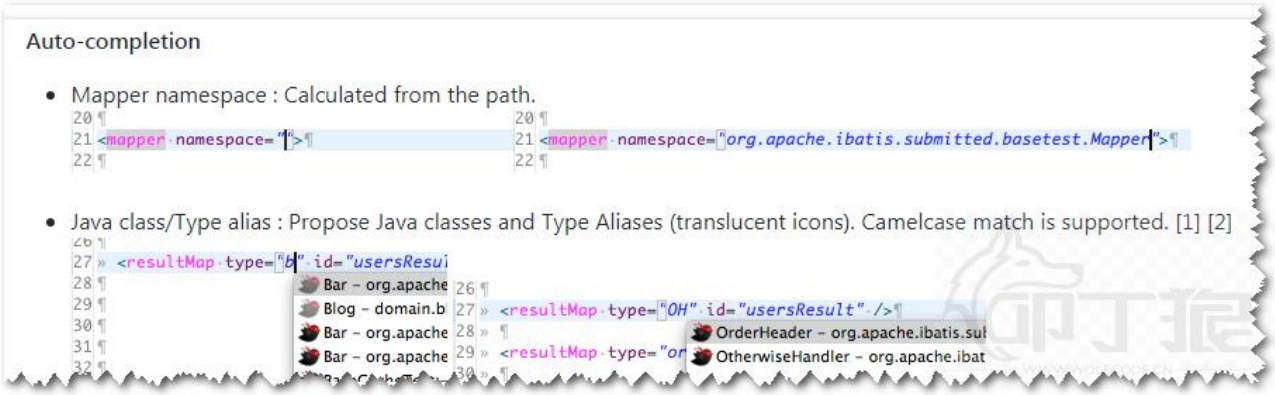
MyBatipse 是 Eclipse 的一个插件，提供了内容提示和 Mybatis 的配置文件验证功能；

官方英文文档：

GitHub - mybatis_mybatipse_ Eclipse plugin adding support for MyBatis SQL Mapper Framework.mhtml

中文文档：

Eclipse 开发工具安装 MyBatipse.mhtml



4. 第四章 动态 SQL

MyBatis 的强大特性之一便是它的动态 SQL。

如果你有使用 JDBC 或其他类似框架的经验，你就能体会到根据不同条件拼接 SQL 语句有多么痛苦。拼接的时候要确保不能忘了必要的空格，还要注意省掉列名列表最后的逗号。利用动态 SQL 这一特性可以彻底摆脱这种痛苦。

通常使用动态 SQL 不可能是独立的一部分，MyBatis 当然使用一种强大的动态 SQL 语言来改进这种情形,这种语言可以被用在任意的 SQL 映射语句中。

4.1. if、choose

选择语句，分支语句。

4.1.1.if

if 元素用于判断，一般用作是否应该包含某一个查询条件。

```
<if test=”boolean 表达式”></if>
```

需求：查询工资高于等于 1000 的。

```
<select id="query" resultType="Employee">
    SELECT id,name,sn,salary FROM employee
    <if test="minSalary!=null">
     WHERE salary >= #{minSalary}
    </if>
</select>
```

需求：查询工资在 1000~2000 的。

```
<select id="query" resultType="Employee">
    SELECT id,name,sn,salary FROM employee
    <if test="minSalary!=null">
      WHERE salary >= #{minSalary}
    </if>
    <if test="maxSalary!=null">
      AND salary &lt;= #{maxSalary}
    </if>
</select>
```

如果 minSalary 和 maxSalary 条件是可选择的此时 SQL 不确定使用 WHERE 还是 AND 来连接查询条件。解决方案使用 WHERE 1=1 方式，其他的查询条件统统使用 AND 或 OR 连接。

```
<select id="query" resultType="Employee">
    SELECT id,name,sn,salary FROM employee WHERE 1 = 1
    <if test="minSalary!=null">
      AND salary >= #{minSalary}
    </if>
    <if test="maxSalary!=null">
      AND salary &lt;= #{maxSalary}
    </if>
</select>
```

但是 WHERE 1=1 会影响查询性能 。

4.1.2.choose、when、otherwise

需求：查询指定部门的员工信息。

```
<select id="query" resultType="Employee">
    SELECT id,name,sn,salary FROM employee WHERE 1 = 1
    <if test="deptId > 0">
      AND dept_id = #{deptId}
    </if>
    <if test="deptId &lt;= 0">
      AND dept_id IS NOT NULL
    </if>
</select>
```

choose 元素类似于 if-else if-else 或者 switch(使用不多)，只允许一个条件存在：

```
<select id="query" resultType="Employee">
    SELECT id,name,sn,salary FROM employee WHERE 1 = 1
    <choose>
      <when test="deptId > 0">
        AND dept_id = #{deptId}
      </when>
      <otherwise>
        AND dept_id IS NOT NULL
      </otherwise>
    </choose>
</select>
```

4.2. where、trim、set

4.2.1.where

where 元素，如果查询条件没有”WHERE“关键字，则自动在查询条件之前插入”WHERE“。

如果查询条件以“AND”或“OR”开头，那么就会使用 WHERE 关键字替换。

where 和 if 连用：

```
<select id="query" resultType="Employee">
   SELECT id,name,sn,salary FROM employee
   <where>
     <if test="minSalary!=null">
       AND salary >= #{minSalary}
     </if>
     <if test="maxSalary!=null">
       AND salary &lt;= #{maxSalary}
     </if>
   </where>
</select>
```

where 和 choose 连用：

```
<select id="query" resultType="Employee">
   SELECT id,name,sn,salary FROM employee
   <where>
     <choose>
       <when test="deptId > 0">
         AND dept_id = #{deptId}
       </when>
       <otherwise>
         AND dept_id IS NOT NULL
       </otherwise>
     </choose>
   </where>
</select>
```

4.2.2.set

set 元素同 where 元素相似，也能根据 set 中的 sql 动态的去掉最后的逗号，并在前面添加 set 关键字，如果没有内容，也会选择忽略 set 语句。

```
<update id="update">
    UPDATE employee
    <set>
      <if test="name!=null">
        name = #{name},
      </if>
      <if test="sn!=null">
        sn = #{sn},
      </if>
      <if test="name!=null">
        salary = #{salary},
      </if>
    </set>
    WHERE id = #{id}
</update>
```

4.2.3.trim

trim 是更为强大的格式化 SQL 的标签：

```
<trim prefix="" prefixOverrides="" suffix="" suffixOverrides="">
      <!--trim 包含的动态 SQL-->
</trim>
```

前提如果 trim 元素包含内容返回了一个字符串，则：

- prefix – 在这个字符串之前插入 prefix 属性值。
- prefixOverrides – 并且字符串的内容以 prefixOverrides 中的内容开头(可以包含管道符号)，那么使用 prefix 属性值替换内容的开头。
- suffix – 在这个字符串之后插入 suffix 属性值。
- suffixOverrides –并且字符串的内容以 suffixOverrides 中的内容结尾(可以包含管道符号) ,那么使用 suffix 属性值替换内容的结尾。

使用 where 等价于：

```
<trim prefix="WHERE" prefixOverrides="AND |OR ">
</trim>
```

注意：此时 AND 后面有一个空格。

使用 set 等价于：

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

4.3. foreach

SQL 中有时候使用 IN 关键字，如 WHERE id IN(10,20,30),此时可以使用${ids}直接拼接 SQL，但是会导致 SQL 注入问题，要避免 SQL 注入，只能使用#{}方式，此时就可以配合使用 foreach 元素了。

foreach 元素用于迭代一个集合/数组，通常是构建在 IN 运算符条件中。

4.3.1.批量删除

需求：删除 ID 为 10,20,30 的数据。

SQL：DELETE FROM employee WHERE id IN(10,20,30);

Mapper 接口：

```
void batchDelete(@Param("ids")Long[] ids);
```

Mapper 文件：

```
<delete id="batchDelete">
    DELETE FROM employee
    WHERE id IN
    <foreach collection="ids" open="(" close=")" separator="," item="id">
      #{id}
    </foreach>
</delete>
```

Mapper 接口中参数如果是数组类型和 List 类型，如何处理？

思考，查询部门 ID 为 10 和 20 的员工。

4.3.2.批量插入

批量插入语法（Oracle 不支持）：

```
INSERT INTO employee (name,sn,salary,dept_id)  VALUES    ('小A','aaa',800.00,10),
                                                         ('小B','bbb',700.00,10),
                                                         ('小C','ccc',600.00,10)
```

Mapper 文件：

```
<insert id="batchInsert">
    INSERT INTO employee (name,sn,salary,dept_id)
    VALUES
    <foreach collection="list" open="(" close=")" separator="," item="u">
      (#{u.name},#{u.sn},#{u.salary},#{u.deptId})
    </foreach>
</insert>
```

4.4. bind、sql、include

4.4.1. bind

使用 OGNL 表达式创建一个变量并将其绑定在上下文中

```
<select id="query" resultType="Employee">
    SELECT id,name,sn,salary FROM employee
    <where>
      <if test="keyword!=null">
        AND (name LIKE concat('%',#{keyword},'%') OR sn LIKE concat('%',#{keyword},'%'))
      </if>
    </where>
</select>
```

使用 bind：

```
<select id="query" resultType="Employee">
    SELECT id,name,sn,salary FROM employee
    <where>
      <if test="keyword!=null">
        <bind name="keywordLike" value="'%' + keyword + '%'"/>
        AND (name LIKE #{keywordLike} OR sn LIKE #{keywordLike})
      </if>
    </where>
</select>
```

4.4.2. sql 和 include

使用 sql 可以把相同的 sql 片段起一个名字，并使用 include 元素在 sql 任意位置使用。

```
<sql id="base_where">
    <where>
      <if test="keyword!=null">
        <bind name="keywordLike" value="'%' + keyword + '%'" />
        AND (name LIKE #{keywordLike} OR sn LIKE #{keywordLike})
      </if>
      <if test="minSalary!=null">
        AND salary >= #{minSalary}
      </if>
      <if test="maxSalary!=null">
        AND salary &lt;= #{maxSalary}
      </if>
```

```
    </where>
</sql>
<select id="query" resultType="Employee">
    SELECT id,name,sn,salary FROM employee
    <include refid="base_where"/>
</select>
```

5. 第五章 对象关系设计

在面向对象中，我们说面向对象是符合人们对现实世界的思维模式，即人们采用针对非程序设计领域存在的复杂问题的解决方式，来解决软件设计过程中各种错综复杂的关系。利用面向对象设计，特别是采用各种设计模式来解决问题时，会设计多个类，然后创建多个对象，这些对象，有些主要是数据模型，有些则是行为描述占主体。一个设计良好的类，应该是兼顾信息和行为，并且是高内聚。而不同的类之间，应该尽量做到松耦合。

由于我们面对的系统或者需要解决的问题经常是复杂的、高度抽象的，我们创建的多个对象往往是有联系的，通常对象之间的关系可以分为以下几类：

- 泛化关系
- 实现关系
- 依赖关系
- 关联关系
- 聚合关系
- 组合关系

对于继承、实现这两种关系比较简单，他们体现的是一种类与类、或者类与接口间的纵向关系。其他的四者关系则体现的是类与类、或者类与接口间的引用、横向关系，这几种关系都是语义级别的，所以从代码层面并不能完全区分各种关系。

从后几种关系所表现的强弱程度来看，依次为：组合>聚合>关联>依赖。在面向对象的设计过程中，能采取强度较大的关系，决不能采取强度小的关系。

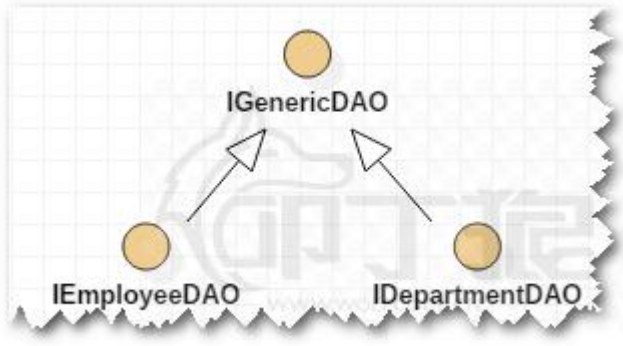
5.1. 泛化关系

泛化关系（generalization）：

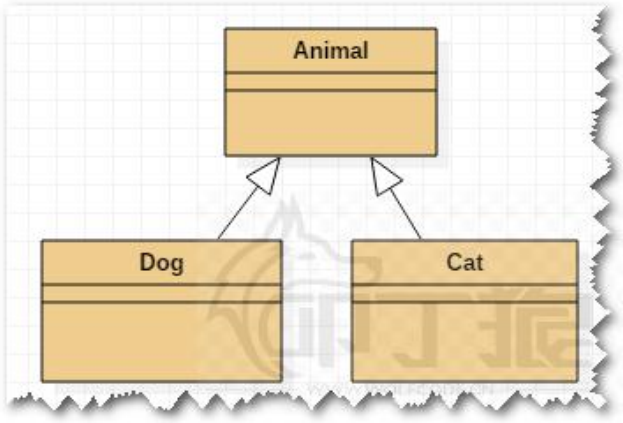
其实就是继承关系，比如类和类之间、接口和接口之间，在代码中使用 extends 表示。

在 UML 中，继承通常是使用空心三角+实线来表示。

接口之间的泛化关系：



类之间的泛化关系：

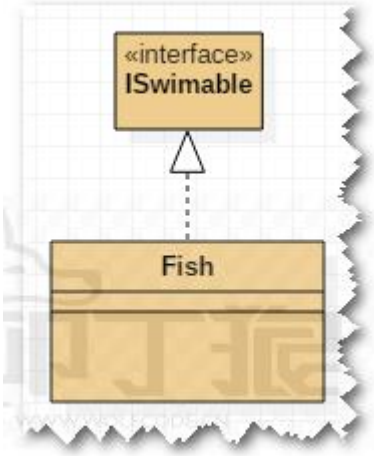


5.2. 实现关系

泛化关系（realization）：

其实就是实现关系，存在于类和接口之间，在代码中使用 implements 表示。

在 UML 中，继承通常是使用空心三角+虚线来表示。



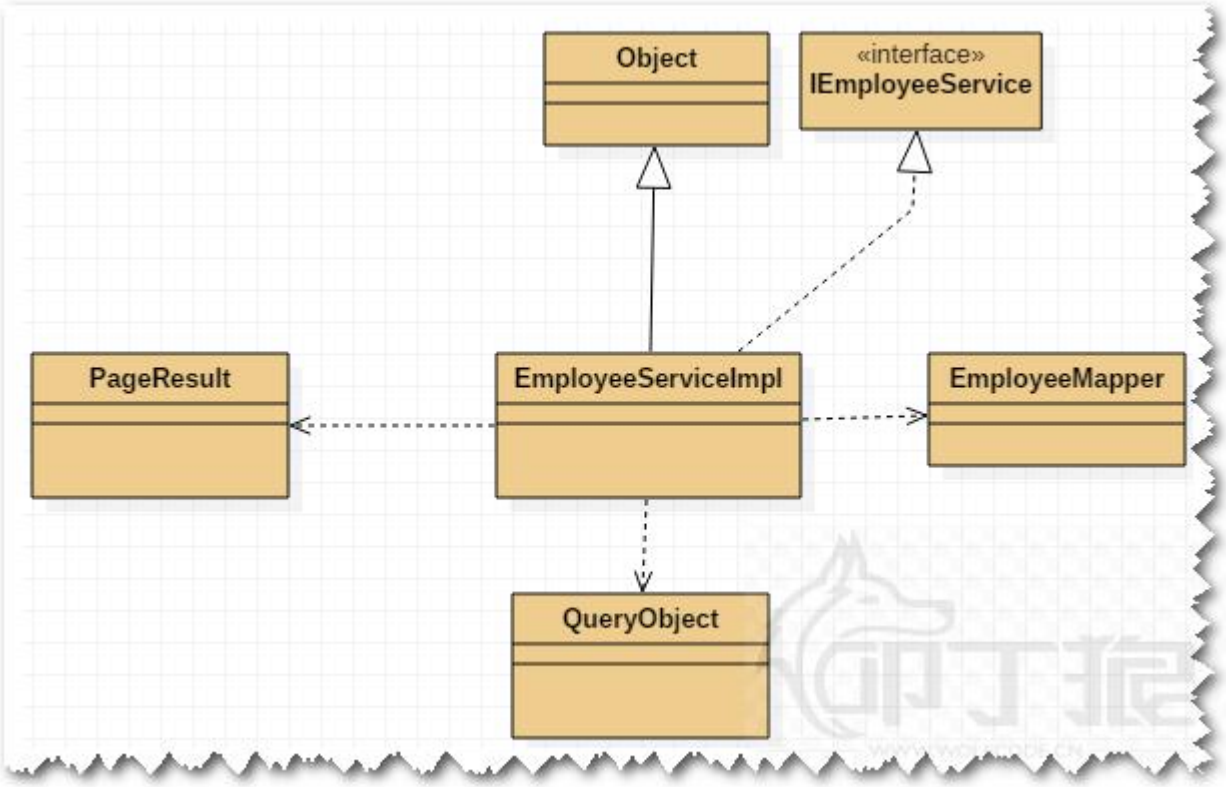
5.3. 依赖关系

依赖关系(dependent)：

表示一个 A 类依赖于另一个 B 类的定义，如果 A 对象离开了 B 对象，A 对象就不能正常编译，则 A 对象依赖 B 对象(A 类中使用到了 B 对象)。

在 UML 中依赖通常使用虚线箭头表示。

```
public class EmployeeServiceImpl implements IEmployeeService {
  private EmployeeMapper employeeMapper;

  public PageResult query(QueryObject qo) {
    //TODO
    return null;
  }
}
```



5.4. 关联关系

关联关系（association）：

A 对象依赖 B 对象，并且把 B 作为 A 的一个成员变量，则 A 和 B 存在关联关系。

在 UML 中依赖通常使用实线箭头表示。

- 按照多重性分:
  - 一对一：一个 A 对象属于一个 B 对象，一个 B 对象属于一个 A 对象。
  - 一对多：一个 A 对象包含多个 B 对象。
  - 多对一：多个 A 对象属于一个 B 对象，并且每个 A 对象只能属于一个 B 对象。
  - 多对多：一个 A 对象属于多个 B 对象，一个 B 对象属于多个 A 对象。
- 按照导航性分：如果通过 A 对象中的某一个属性可以访问到 B 对象，则说 A 可以导航到 B。
  - 单向：只能从 A 通过属性导航到 B，B 不能导航到 A。
  - 双向：A 可以通过属性导航到 B，B 也可以通过属性导航到 A。

关联关系的判断方法：

1，判断都是从对象的实例上面来看的。

2，判断关系必须确定一对属性。

3，判断关系必须确定具体需求。

5.4.1.一对一

一对一：一个 A 对象属于一个 B 对象，一个 B 对象属于一个 A 对象。

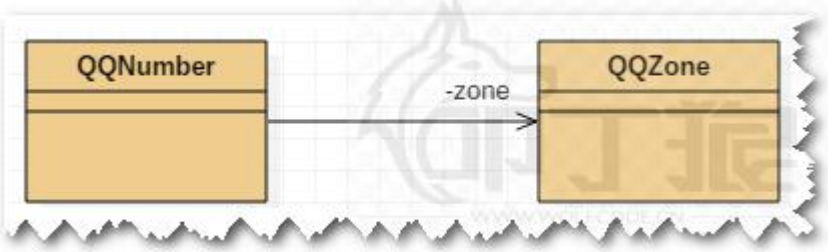
如 QQNumber 和 QQZone 的案例。

单向关系：

```
@Data
public class QQNumber {
  private QQZone zone;
}
```

```
@Data
public class QQZone {
}
```

UML 图：

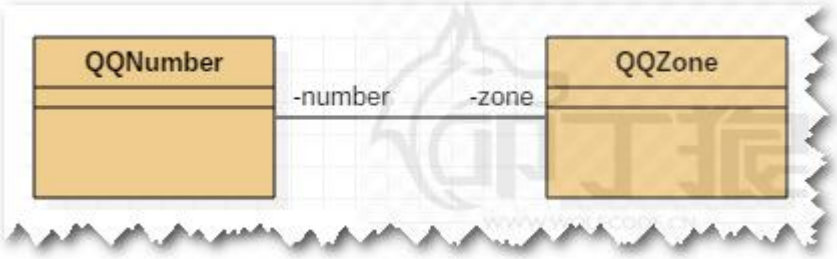


双向关系：

```
@Data
public class QQNumber {
  private QQZone zone;
}
```

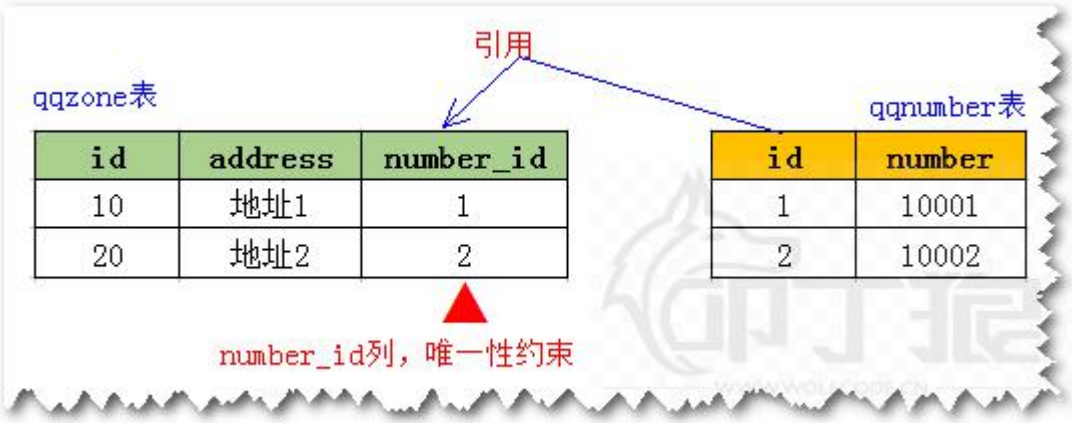
```
@Data
public class QQZone {
  private QQNumber number;
}
```

UML 图：

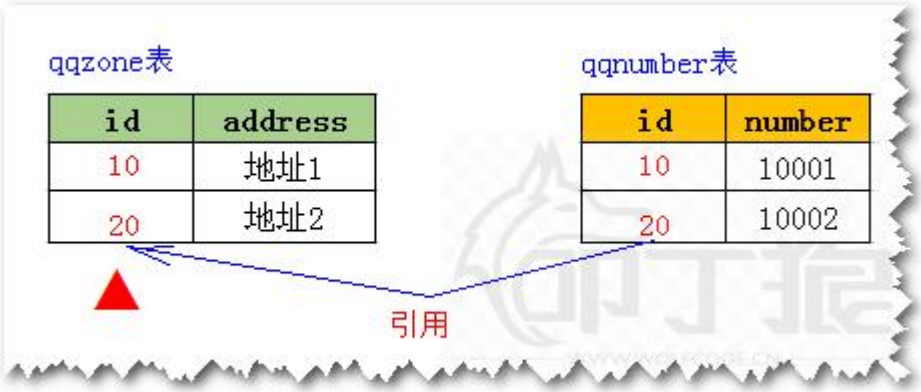


表设计：

方式一，唯一外检约束：



方式二，共享主键（主外键约束）：



此时 qqzone 表中的 id 是一个主键列，但是数据来源于 qqnumber 表中的 id 列。

此时我们可以看出 QQNumber 是主对象，QQZone 是从对象。

需求：保存一个 QQNumber 和一个 QQZone 对象。

如果先保存 QQNumber，再保存 QQZone，此时只会发两条 SQL 语句（性能高）。

如果先保存 QQZone，此时没有 number_id 列只能设置为 NULL，再保存 QQNumber 数据，最后还得再发送一条额外的 UPDATE 的 SQL 去更新 QQZone 表中的 number_id 列的数据。

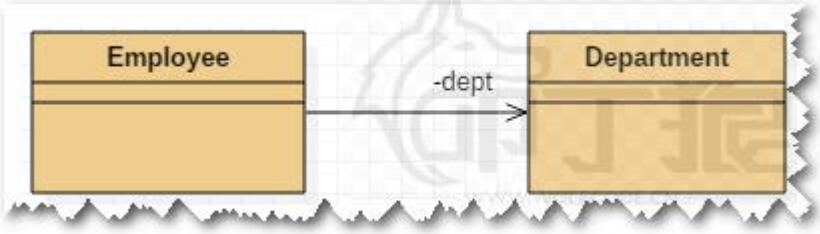
5.4.2.单向多对一

单向多对一：多个 A 对象属于一个 B 对象，并且每个 A 对象只能属于一个 B 对象，只能从 A 对象导航到 B 对象。

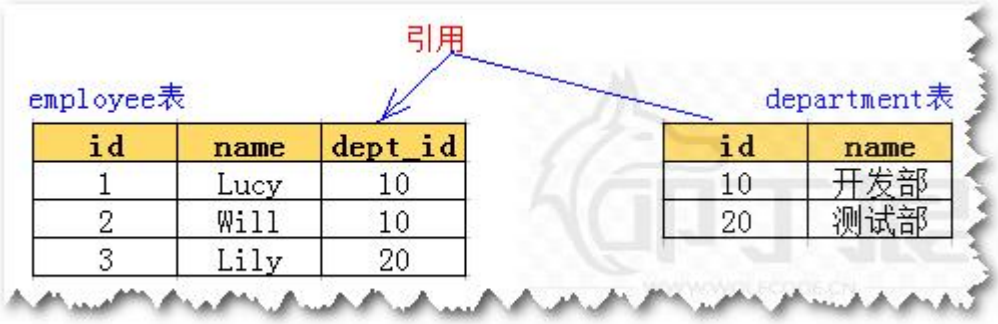
```
@Data
public class Department {
}
```

```
@Data
public class Employee {
  private Department dept;
}
```

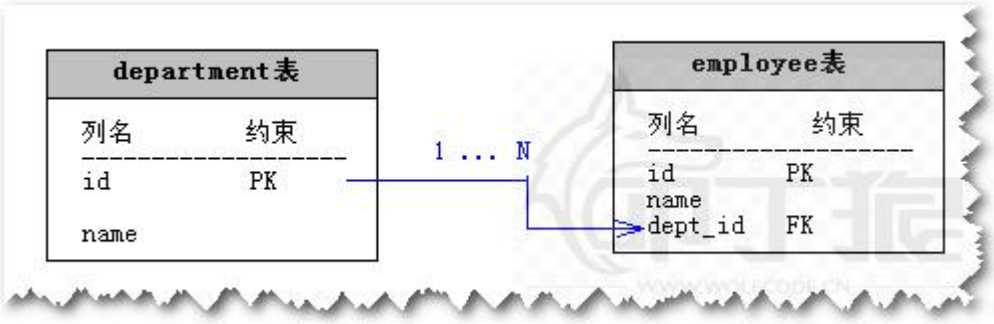
UML 图：



表设计：



或者



注意：外键在 many 方。

需求：保存一个 Department 对象两个 Employee 对象。

如果先保存 department，再保存 employee，此时只需要三条 SQL 语句即可。

如果先保存 employee，此时会发送两天 INSERT 语句，但是 employee 表中的 dept_id 列为 NULL，

再保存 department，最后还得额外发送两天 UPDATE 的 SQL 去维护 dept_id 列的值。

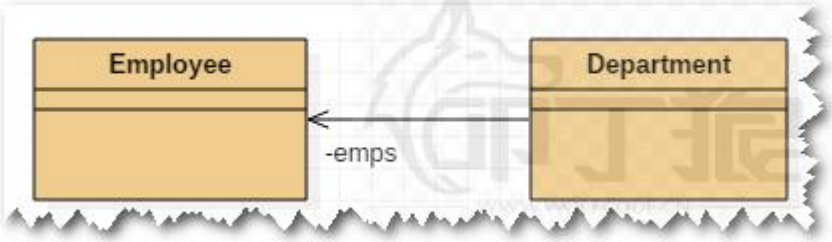
5.4.3.单向一对多

多对一：多个 A 对象属于一个 B 对象，并且每个 A 对象只能属于一个 B 对象。

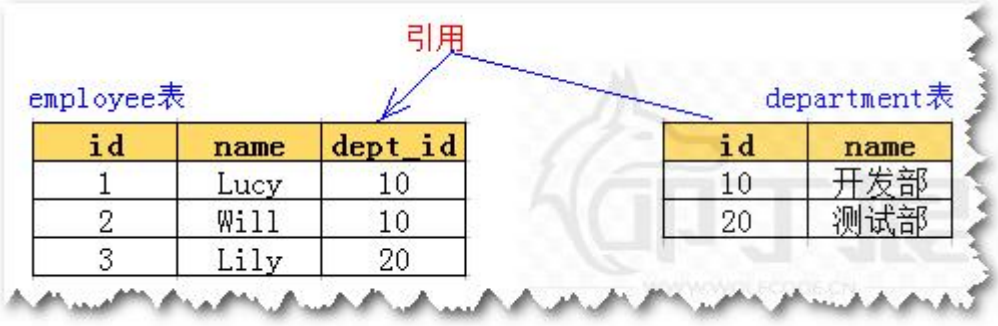
```
@Data
public class Department {
  private List<Employee> emps = new ArrayList<>();
}
```

```
@Data
public class Employee {
}
```

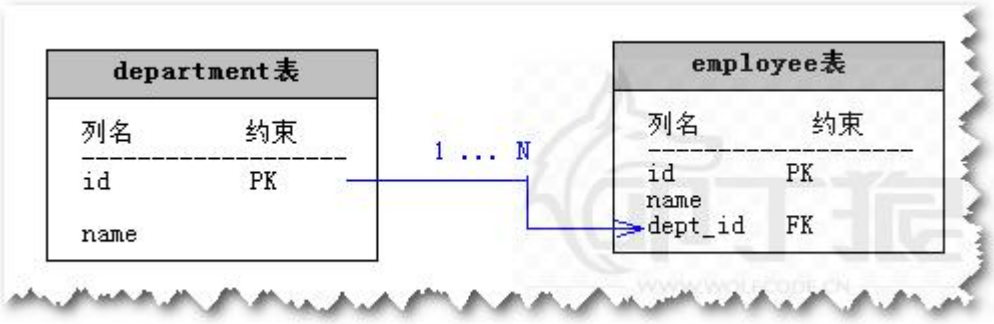
UML 图：



表设计，此时表设计和 many2one 一致，且外键在 many 方。



或者



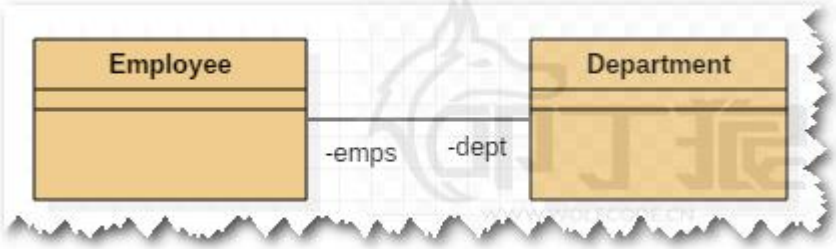
5.4.4.双向多对一和一对多

双向：A 可以通过属性导航到 B，B 也可以通过属性导航到 A，其实就是把两个单向关系组合起来。

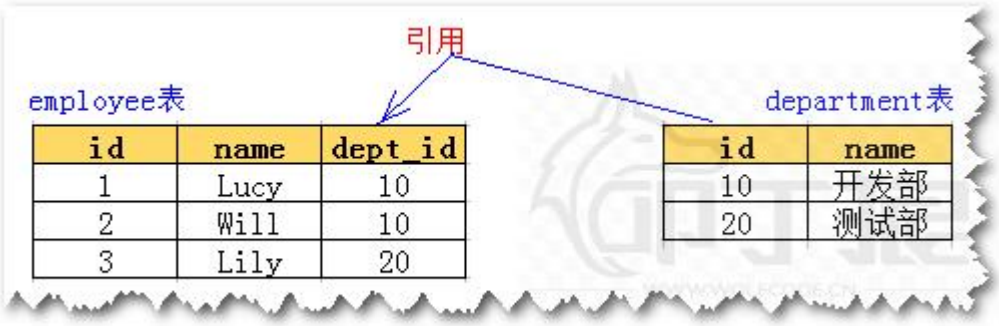
```
@Data
public class Department {
  private List<Employee> emps = new ArrayList<>();
}
```

```
@Data
public class Employee {
  private Department dept;
}
```

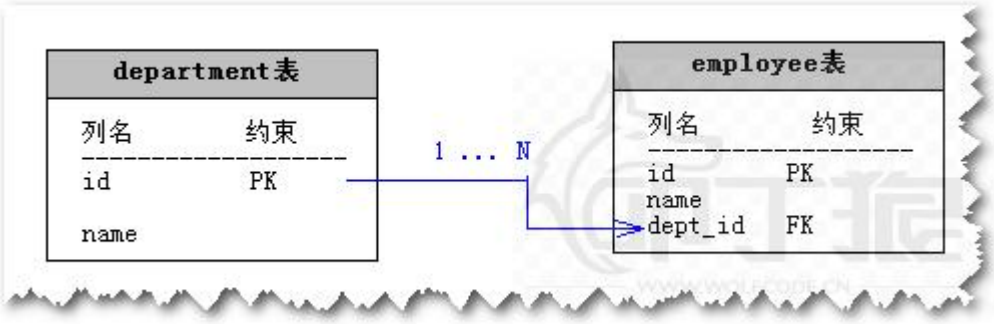
UML 图：



表设计，此时和单向关系一致。



或者



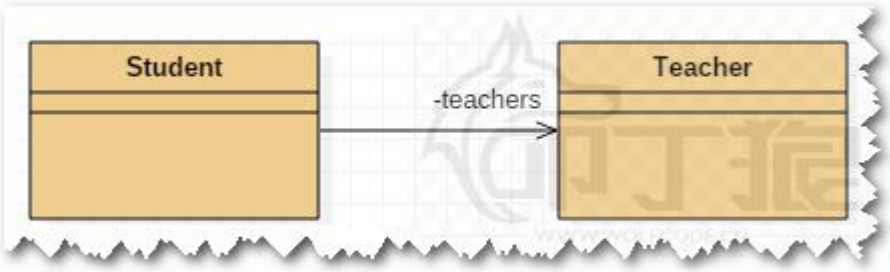
5.4.5.多对多

多对多：一个 A 对象属于多个 B 对象，一个 B 对象属于多个 A 对象（单向关系）。

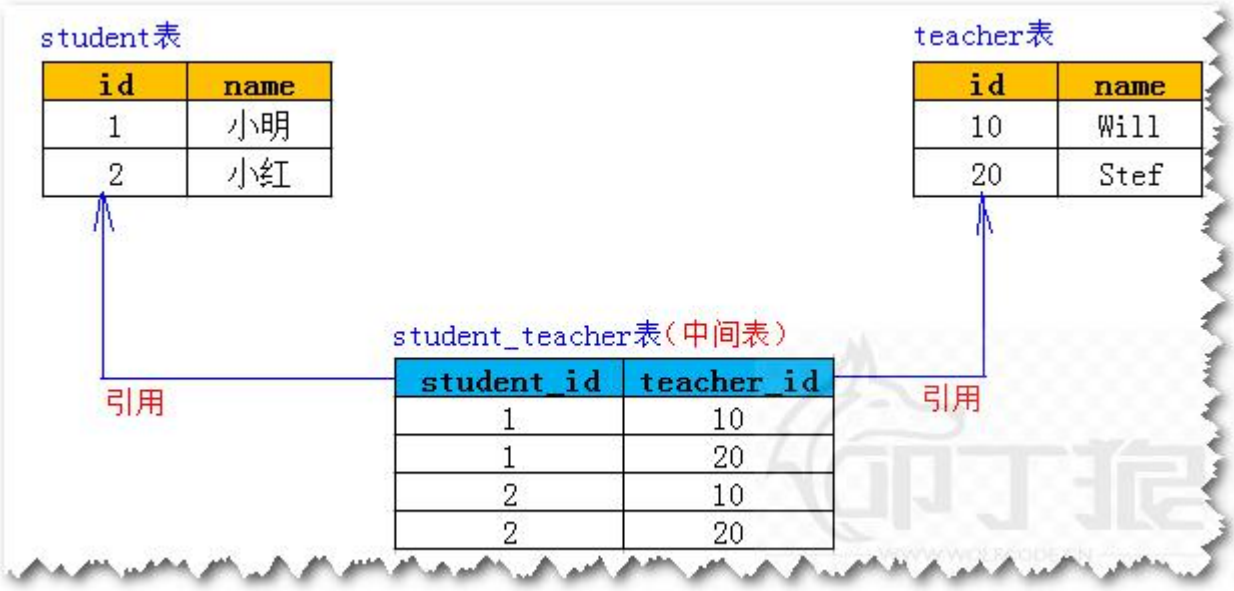
```
@Data
public class Student {
  private List<Teacher> teachers =
                          new ArrayList<>();
}
```

```
@Data
public class Teacher {
}
```

UML 图：



表设计：



中间表的主键设计：

方式一、中间表不设置主键

方式二、把 student_id 和 teacher_id 列设计为联合主键

需求：保存两个学生和两个老师对象。

此时需要发送 8 条 SQL 语句，其中四条 SQL 是保存在中间表中数据。

5.5. 聚合关系

聚会关系（aggregation）：

是一种“弱拥有”关系，表示为 has-a。

表示整体和个体的关系，整体和个体之间可以相互独立存在，一定是有两个模块来分别管理整体和个体。

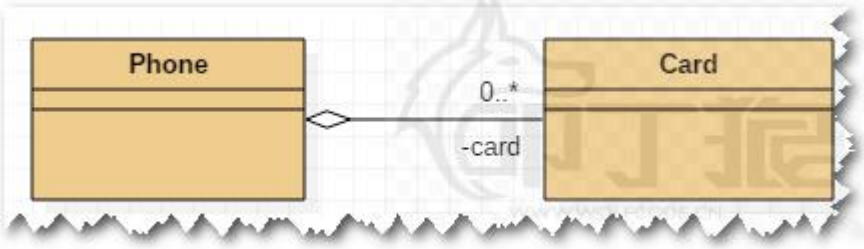
如果 A 和 B 是聚合关系，它们并不是一个独立的整体，A 和 B 的生命周期可以是不同的，通常 B 也是会作为 A 的成员变量存在。

在 UML 中，聚合通常是使用空心菱形+实线箭头来表示。

```
@Data
public class Phone{
  private Card card;
}
```

```
@Data
public class Card{
}
```

UML 图：



5.6. 组合关系

组合关系（composition）：

是一种强聚合关系，是一种“强拥有”关系，表示为 contains-a。

整体和个体不能独立存在，一定是在一个模块中同时管理整体和个体，生命周期必须相同(级联)。

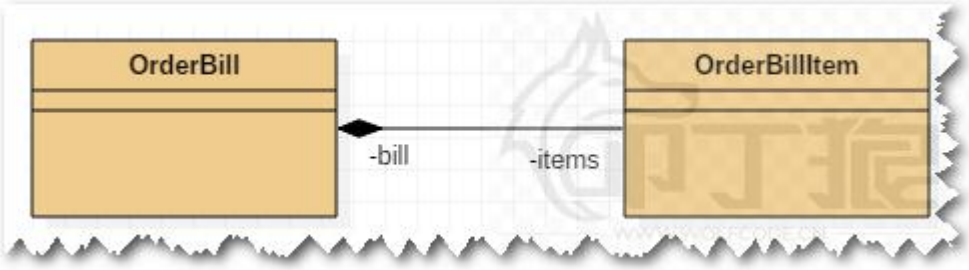
在 UML 中，组合通常是使用实心菱形+实线箭头表示。

一般的，组合关系设计为双向关联。

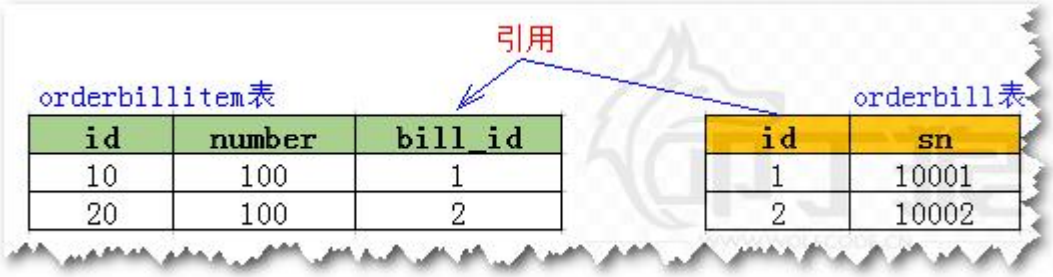
```
@Data
public class OrderBill {
  private List<OrderBillItem> items;
}
```

```
@Data
public class OrderBillItem {
  private OrderBill bill;
}
```

UML 图：



表设计：



id	number	bill_id
10	100	1
20	100	2

id	sn
1	10001
2	10002

6. 第六章 对象关系映射

6.1. 多对一

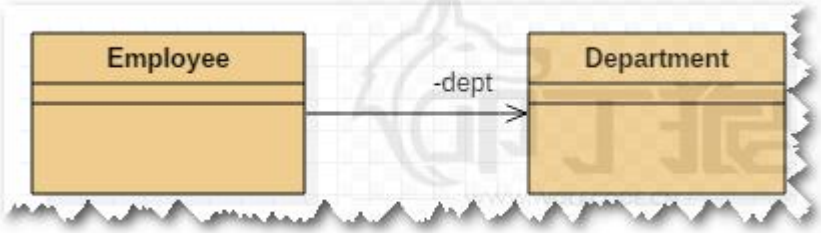
案例：多个员工对象同属于同一个部门对象。

模型对象设计：

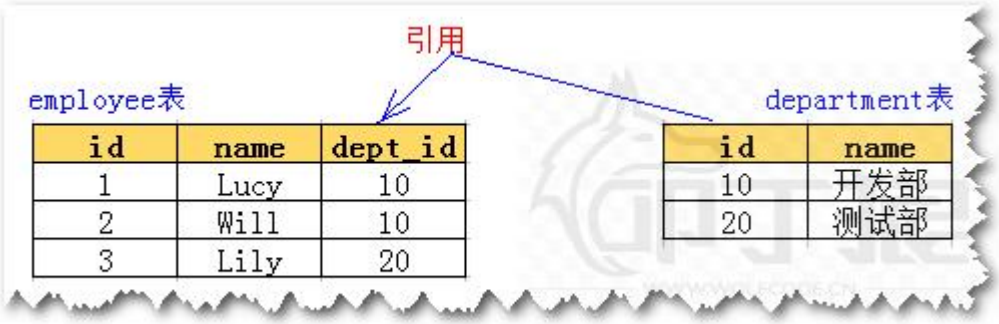
```java
@Getter
@Setter
@ToString
public class Department {
    private Long id;
    private String name;
}
```

```java
@Getter
@Setter
@ToString(exclude = "dept")
public class Employee {
    private Long id;
    private String name;
    private Department dept;
}
```

UML 图：



表设计：



id	name	dept_id
1	Lucy	10
2	Will	10
3	Lily	20

id	name
10	开发部
20	测试部

6.1.1.保存操作

DepartmentMapper.xml

```
<insert id="save" useGeneratedKeys="true" keyProperty="id">
   INSERT INTO department (name) VALUES(#{name})
 </insert>
```

EmployeeMapper.xml

```
<insert id="save" useGeneratedKeys="true" keyProperty="id">
   INSERT INTO employee (name,dept_id) VALUES(#{name},#{dept.id})
</insert>
```

Java 代码：

```
@Test
 void testSave() throws Exception {
   Department d = new Department();
   d.setName("开发部");

   Employee e1 = new Employee();
   e1.setName("小 A");
   e1.setDept(d);

   Employee e2 = new Employee();
   e2.setName("小 B");
   e2.setDept(d);

   SqlSession session = MyBatisUtil.getSession();
   DepartmentMapper departmentMapper = session.getMapper(DepartmentMapper.class);
   EmployeeMapper employeeMapper = session.getMapper(EmployeeMapper.class);


   departmentMapper.save(d);
   employeeMapper.save(e1);
   employeeMapper.save(e2);

   session.commit();
   session.close();
  }
```

6.1.2.内联映射

association 元素：

property 属性：关联对象属性名

javaType 属性：关联对象属性类型

多表连接查询的 SQL：

```
<select id="get" resultMap="BaseResultMap">
   SELECT e.id,e.name,d.id AS d_id,d.name AS d_name
   FROM employee e LEFT JOIN department
   d ON e.dept_id = d.id
   WHERE e.id = #{id}
</select>
```

方式一， 使用级联方式来封装对象（不用）。

```
<resultMap id="BaseResultMap" type="Employee">
   <id column="id" property="id" />
   <result column="name" property="name" />
   <result column="d_id" property="dept.id" />
   <result column="d_name" property="dept.name" />
</resultMap>
```

方式二，使用 association 元素（常用）。

```
<resultMap id="BaseResultMap" type="Employee">
   <id column="id" property="id" />
   <result column="name" property="name" />
   <association property="dept" javaType="Department">
     <id column="d_id" property="id" />
     <id column="d_name" property="name" />
   </association>
</resultMap>
```

使用 column_prefix 统一设置列的别名。

```
<resultMap id="BaseResultMap" type="Employee">
   <id column="id" property="id" />
   <result column="name" property="name" />
   <association property="dept" javaType="Department" columnPrefix="d_">
     <id column="id" property="id" />
     <result column="name" property="name" />
   </association>
</resultMap>
```

6.1.3. 额外 SQL

association 元素

select 属性 ：发送的额外 SQL 语句

column 属性 ：将指定列的值传给额外 SQL

EmployeeMapper.xml

```
<resultMap id="BaseResultMap" type="Employee">
   <id column="id" property="id" />
   <id column="name" property="name" />
   <id column="dept_id" property="dept.id" />
</resultMap>
```

单独发送额外 SQL 方式：

```
<select id="get" resultMap="BaseResultMap">
   SELECT id,name,dept_id FROM employee WHERE id = #{id}
</select>
```

DepartmentMapper.xml

```
<select id="get" resultType="Department">
   SELECT id,name FROM department WHERE id = #{id}
</select>
```

Java 代码，分别查询两个对象，再组装。

```
Employee e = employeeMapper.get(1L);
Department d = departmentMapper.get(e.getDept().getId());
e.setDept(d);
```

让 MyBatis 帮我们主动发送额外的 SQL 配置：

```
<resultMap id="BaseResultMap" type="Employee">
   <id column="id" property="id" />
   <id column="name" property="name" />
   <id column="dept_id" property="dept.id" />
   <association property="dept"
     column="dept_id"
     select="cn.wolfcode.mybatis.many2one.DepartmentMapper.get"/>
</resultMap>
```

6.1.4. N+1 问题

在 employee 表中有 N 条数据，每一个员工都关联着一个不同的部门 ID。

当在查询员工和部门信息列表时，就会发送 N+1 条语句，严格上说应该是 1+N 条 SQL。

- 1 条：SELECT * FROM employee;
- N 条：SELECT * FROM department WHERE id = ？（？表示 1 到 N）

分析：在列表中查询员工信息和部门信息的时候性能超低，导致的原因是使用额外的 SQL 做的映射配置。

解决方案：使用内联映射(多表查询)，此时一条 SQL 语句搞定。

内联映射和额外 SQL 的选择：

在开发中，多对一的关联关系，一般的都是在列表中显示，通常直接使用多表查询，也就是内联查询处理。

如果在当前页面不显示数据，需要进去另一个页面再显示的数据，此时选用额外 SQL 方式。

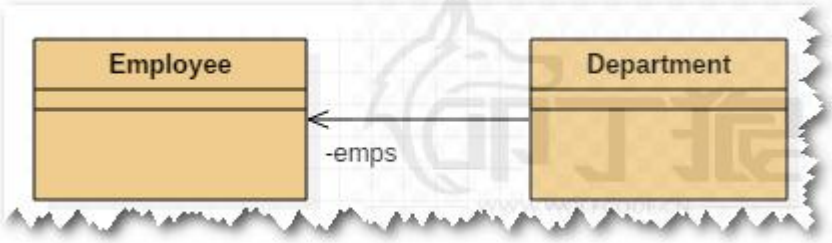
6.2. 一对多

模型对象设计：

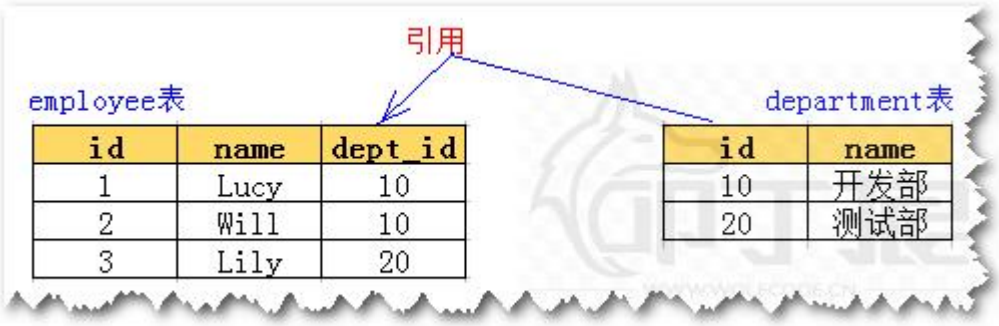
```java
@Getter
@Setter
@ToString(exclude = "emps")
public class Department {
    private Long id;
    private String name;
    private List<Employee> emps = new ArrayList<>();
}
```

```java
@Getter
@Setter
public class Employee {
    private Long id;
    private String name;
}
```

UML 图：



表设计（外键在 many 方）：



id	name	dept_id
1	Lucy	10
2	Will	10
3	Lily	20

id	name
10	开发部
20	测试部

6.2.1. 保存操作

单向的 one2many 关系，many 方不会去去维护外键，只能是 one 方去维护。在保存完 one 方和 many 方之后，one 方会再次发送额外的 UPDATE 的 SQL 去设置外键值。因为额外的 SQL 此时会造成性能降低，所以在开发中往往设计为单向的 many2one 的关系。

DepartmentMapper.xml

```
<insert id="save" useGeneratedKeys="true" keyProperty="id">
   INSERT INTO department (name) VALUES(#{name})
</insert>
```

EmployeeMapper.xml

```
<insert id="save" useGeneratedKeys="true" keyProperty="id">
   INSERT INTO employee (name) VALUES(#{name})
</insert>
<update id="updateRelationWithEmployee">
    UPDATE employee SET dept_id = #{deptId} WHERE id = #{employeeId}
</update>
```

Java 代码：

```
@Test
    void testSave() throws Exception {
        Department d = new Department();
        d.setName("开发部");

        Employee e1 = new Employee();
        e1.setName("小 A");

        Employee e2 = new Employee();
        e2.setName("小 B");

        d.getEmps().add(e1);
        d.getEmps().add(e2);

        SqlSession session = MyBatisUtil.getSession();
        DepartmentMapper departmentMapper = session.getMapper(DepartmentMapper.class);
        EmployeeMapper employeeMapper = session.getMapper(EmployeeMapper.class);

        departmentMapper.save(d);
        employeeMapper.save(e1);
        employeeMapper.save(e2);
        //one 方维护对象关系（维护外键列数据）
        for (Employee e : d.getEmps()) {
            departmentMapper.updateRelationWithEmployee(d.getId(), e.getId());
        }
        session.commit();
        session.close();
    }
```

因为额外的 SQL 此时会造成性能降低，所以在开发中往往设计为单向的 many2one 的关系。

上述设计，了解即可，在开发中不会出现，仅仅是演示。

6.2.2. 额外 SQL

DepartmentMapper.xml

```
<select id="get" resultType="Department">
   SELECT id,name FROM department WHERE id = #{id}
</select>
```

EmployeeMapper.xml

```
<select id="selectByDeptId" resultType="Employee">
   SELECT id,name FROM employee WHERE dept_id = #{deptId}
</select>
```

Java 代码，分别查询两个对象，再组装。

```
Department dept = departmentMapper.get(10L);
List<Employee> list = employeeMapper.selectByDeptId(10L);
dept.setEmps(list);
```

让 MyBatis 帮我们发送额外的 SQL，配置：

```
<resultMap id="BaseResultMap" type="Department">
   <id column="id" property="id"/>
   <result column="name" property="name"/>
   <collection property="emps" ofType="Employee"
     column="id"
     select="cn.wolfcode.mybatis.one2many.EmployeeMapper.selectByDeptId"/>
</resultMap>
```

6.2.3. 内联映射

使用一条 SQL 语句查询出部门和该部门对应的员工（一般不用）：

```
<resultMap id="BaseResultMap" type="Department">
   <id column="id" property="id" />
   <result column="name" property="name" />
   <collection property="emps" ofType="Employee">
     <id column="e_id" property="id" />
     <result column="e_name" property="name" />
   </collection>
</resultMap>

<select id="get" resultMap="BaseResultMap">
   SELECT d.id,d.name,e.id AS e_id,e.name AS e_name
   FROM department d LEFT JOIN employee e ON d.id = e.dept_id
   WHERE d.id = #{id}
</select>
```

内联映射和额外 SQL 的选择：

内联映射： 使用多表查询，一次性查询出所有数据，在列表中一起显示的数据。

额外 SQL：使用额外 SQL，分步查询出所有数据，在另一个页面单独显示的数据。

6.3. 延迟加载

延迟加载(lazy load)：也称为懒加载。

为了避免一些无谓的性能开销而提出来的，所谓延迟加载就是当在真正需要数据的时候，才会发出 sql 语句进行查询该数据。

MyBatis 运行时期的属性配置，放在主配置文件中的<settings>元素中。

MyBatis 中直接查询出来的 many 方对象其实就已经是一个代理对象，当使用 many 方对象的任意一个属性时，立刻很积极的把关联对象也查询出来，此时性能不会太好。

配置细节：

①、MyBatis 缺省情况下，禁用了延迟加载。

②、MyBatis 会很积极的去查询关联对象。

③、MyBatis 中缺省情况下调用 equals、clone、hashCode、toString 都会触发延迟加载，一般的我们保留 clone 就可以了，也就是说调用 many 方对象的 toString、hashCode、equals 方法依然不会去发送查询 one 方的 SQL。

在 mybatis-config.xml 中延迟加载的配置如下：

```
    <settings>
          <!-- 启用延迟加载 -->
          <setting name="lazyLoadingEnabled" value="true" />
          <!-- 禁用积极延迟加载 -->
          <setting name="aggressiveLazyLoading" value="false" />
          <!-- 延迟加载触发方法 -->
          <setting name="lazyLoadTriggerMethods" value="clone" />
   </settings>
```

分步骤演示。

6.4. 关联对象配置选择

在开发中，

针对单属性对象，使用 association 元素，通常直接使用多表查询操作，也就是使用内联处理。

针对集合属性对象，使用 collection 元素，通常使用延迟加载，也就是额外 SQL 查询处理。

6.5. 多对多

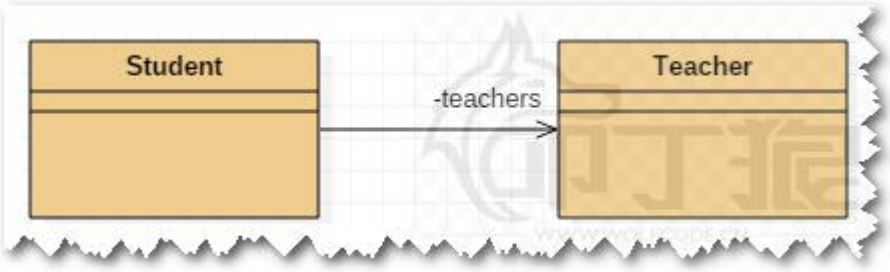
多对多：一个 A 对象属于多个 B 对象，一个 B 对象属于多个 A 对象（单向关系）。

模型对象设计：

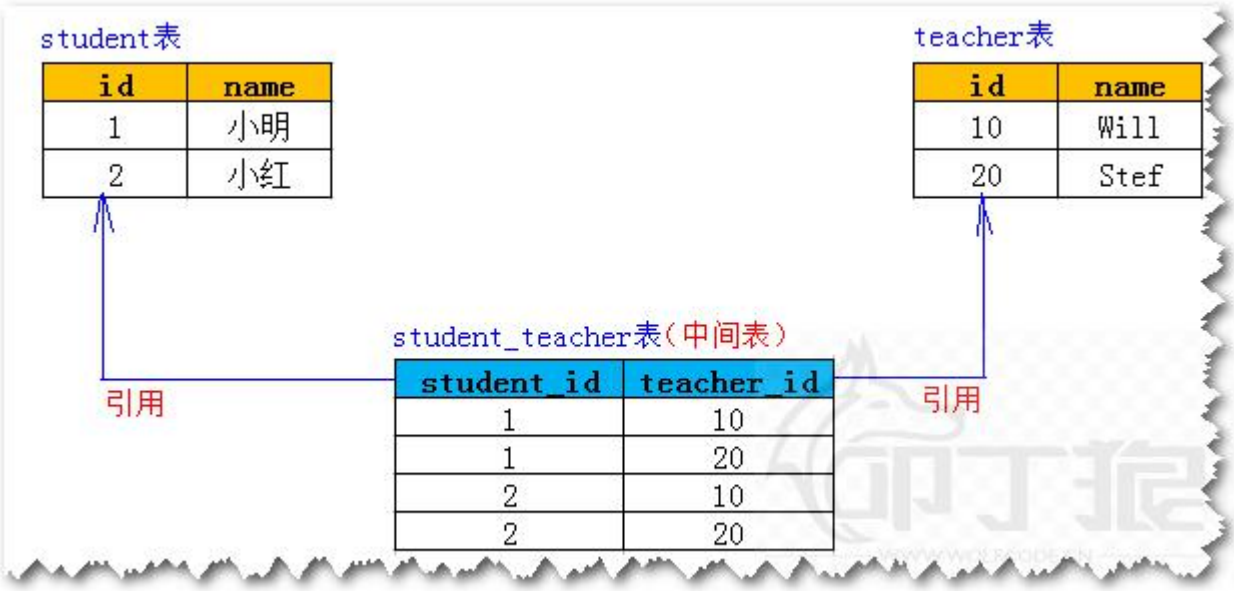
```java
@Getter
@Setter
@ToString
public class Teacher {
    private Long id;
    private String name;
}
```

```java
@Getter
@Setter
@ToString(exclude="teachers")
public class Student {
    private Long id;
    private String name;
    private List<Teacher> teachers = new ArrayList<>();
}
```

UML 图：



表设计：



中间表的主键设计：

方式一、中间表不设置主键

方式二、把 student_id 和 teacher_id 列设计为联合主键

6.5.1.保存操作

因为存在中间表的缘故，所以必须发送额外的 SQL 去维护中间表的关系。

StudentMapper.xml：

```
<insert id="insertRelationWithTeacher">
    INSERT INTO student_teacher (student_id,teacher_id) VALUES(#{studentId},#{teacherId})
</insert>
```

Java 代码：

```
@Test
void testSave() throws Exception {
        SqlSession session = MyBatisUtil.getSession();
        StudentMapper studentMapper = session.getMapper(StudentMapper.class);
        TeacherMapper teacherMapper = session.getMapper(TeacherMapper.class);
        Student s1 = new Student();
        s1.setName("学 1");
        Student s2 = new Student();
        s2.setName("学 2");
        Teacher t1 = new Teacher();
        t1.setName("师 1");
        Teacher t2 = new Teacher();
        t2.setName("师 2");

        studentMapper.save(s1);
        studentMapper.save(s2);
        teacherMapper.save(t1);
        teacherMapper.save(t2);

        s1.getTeachers().add(t1);
        s1.getTeachers().add(t2);
        s2.getTeachers().add(t1);
        s2.getTeachers().add(t2);
        //维护中间表关系
        for (Teacher t : s1.getTeachers()) {
            studentMapper.insertRelationWithTeacher(s1.getId(), t.getId());
        }
        for (Teacher t : s2.getTeachers()) {
            studentMapper.insertRelationWithTeacher(s2.getId(), t.getId());
        }
        session.commit();
        session.close();
}
```

6.5.2.查询操作

因为此时 teachers 属性是集合类型，所以使用额外 SQL 是合理的，使用内联查询时不合理的。所以这里单讲额外 SQL 方式。

StudentMapper.xml

```
<resultMap id="BaseResultMap" type="Student">
    <id column="id" property="name"/>
    <result column="name" property="name"/>
    <collection property="teachers" ofType="Teacher"
      column="id"
      select="cn.wolfcode.mybatis.many2many.mapper.TeacherMapper.selectByStudentId"/>
</resultMap>

<select id="get" resultMap="BaseResultMap">
    SELECT id,name FROM student WHERE id = #{id}
</select>
 TeacherMapper.xml

<select id="selectByStudentId" resultType="Teacher">
    SELECT t.id,t.name  FROM teacher t JOIN student_teacher st ON t.id = st.teacher_id
    WHERE st.student_id = #{studentId}
</select>
```

6.5.3.删除操作

删除操作之前，必须先删除中间表中关联的数据。

```
<delete id="deleteRelationWithTeacher">
    DELETE FROM student_teacher WHERE student_id = #{id}
 </delete>
<delete id="delete">
    DELETE FROM student WHERE id = #{id}
</delete>
```

7. 第七章 缓存机制

使用缓存可以使应用更快地获取数据，避免频繁的数据库交互操作，尤其是在查询越多，缓存命中率越高的情况下，缓存的作用就越明显。

MyBatis 的缓存，包括一级缓存和二级缓存。

1、一级缓存（也称为本地缓存），默认已经开启。

2、二级缓存（也称为查询缓存），需要手动开启和配置，基于其他的缓存框架技术。

7.1. 一级缓存

在 SqlSession 中存在一个 Map 用于缓存查询出来的对象，可以提升性能。

MyBatis 把执行的方法和参数通过算法生成缓存的 key 将 key 和查询的结果 value 存入一个 Map 对象中。

注意：一级缓存提升性能有限，因为 SqlSession 生命周期太短(和 SqlSession 相同)，真正需要提升性能我们得使用二级缓存。

```
SqlSession session = MyBatisUtil.getSession();
UserMapper userMapper = session.getMapper(UserMapper.class);
userMapper.get(1L);
session.clearCache();
userMapper.get(1L);
```

7.2. 二级缓存

二级缓存相关的概念：

不是所有的对象都适合放到二级缓存中。

通常情况下会放入到二级缓存中情况：

- 经常查询的数据 。
- 很少被修改的数据。
- 不会被并发访问的数据。

不适合放在二级缓存中的情况：

- 经常被修改的数据。

二级缓存有一些性能相关属性：

- 命中率（总的从二级缓存中取得的数量/总的取的数量）
- 最大对象数量；
- 最大空闲时间；

7.2.1. 基本使用

一级缓存是 SqlSession 级别的，二级缓存是 mapper 级别的。

二级缓存是多个 SqlSession 共享的，作用域是 mapper 的同一个 namespace。

1、二级缓存默认是开启的：

```
<settings>
```

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

2、Mapper 文件中设置二级缓存，使用 cache 元素：

```
<?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="cn.wolfcode.mybatis.cache.mapper.UserMapper">
  <cache/>
  <select id="get" resultType="User">
    SELECT id,name FROM student WHERE id = #{id}
  </select>
</mapper>
```

3、存于缓存中的对象必须实现 java.io.Serializable 接口：

```
@Data
public class User implements java.io.Serializable{
  private static final long serialVersionUID = 1L;
  private Long id;
  private String name;
  private int age;
}
```

7.2.2.缓存配置细节

1、映射文件中所有的 select 都会使用查询缓存。

2、大多数情况下，对于列表查询的 select 不缓存，因为只有 SQL 和查询参数完全相同，才会使用到查询缓存（设置 useCache="false"）。

```
<select id="list" resultType="User" useCache="false">
</select>
```

3、一般的，只会对 get 方法做查询缓存。

4、文件中所有的 insert、update、delete 语句都会刷新缓存。一般的，针对 insert 操作不需要刷新缓存（设置 flushCache="false"）。

```
<insert id="save" useGeneratedKeys="true" keyProperty="id" flushCache="false">
</insert>
```

7.2.3.常见缓存属性

cache 元素的常用属性：

- eviction 属性：缓存回收策略：
  - LRU　最近最少使用的：移除最长时间不被使用的对象，缺省。
  - 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：更积极地移除基于垃圾收集器状态和弱引用规则的对象。
- flushInterval 属性：刷新间隔，单位毫秒，可以设置为任意正整数，缺省为没有刷新间隔，缓存仅仅调用语句时刷新。
- size 属性：引用数目，正整数，缺省为 1024，代表缓存最多可以存储多少个对象，太大容易导致内存溢出
- readOnly 属性：只读，true/false，默认为 false。
  - true：只读缓存；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。
  - false：读写缓存；会返回缓存对象的拷贝（通过序列化），这会慢一些，但是安全。

7.3. EhCache

Ehcache 是一种广泛使用的开源 Java 分布式缓存，具有快速、简单、多种缓存策略、缓存数据有内存和硬盘两级的特点。缓存数据会在 JVM 重启的过程中写入硬盘，支持多缓存管理器实例以及一个实例的多个缓存区域。

也可以在 MyBatis 中使用 EhCache 作为二级缓存技术的实现。

在 MyBatis 中使用 EhCache 依赖的 jar：

```
ehcache-core-2.6.8.jar
mybatis-ehcache-1.0.3.jar
slf4j-api-1.7.25.jar
slf4j-log4j12-1.7.25.jar
```

7.3.1. EhCache 配置文件

EhCache 缓存技术的配置文件是 ehcache.xml，配置细节如下配置:

<defaultCache>：默认的 cache，相当于公用 cache；

<cache>：自定义的 cache；

共同的属性配置：

1，maxElementsInMemory：该缓存池放在内存中最大的缓存对象个数；

2，eternal：是否永久有效，如果设置为 true，内存中对象永不过期；

3，timeToIdleSeconds：缓存对象最大空闲时间，单位：秒；

4，timeToLiveSeconds：缓存对象最大生存时间，单位：秒；

5，overflowToDisk：当内存中对象超过最大值，是否临时保存到磁盘；

6，maxElementsOnDisk：能保存到磁盘上最大对象数量；

7，diskExpiryThreadIntervalSeconds:磁盘失效线程运行时间间隔，默认是 120 秒

8，memoryStoreEvictionPolicy：当达到 maxElementsInMemory 限制时，Ehcache 将会根据指定的策略去清理内存。默认策略是 LRU（最近最少使用），可以设置为 FIFO（先进先出）或是 LFU（较少使用）

7.3.2. EhCache 使用

在 MyBatis 的 Mapper 文件中设置查询缓存选用的缓存技术：

```
    <cache type="org.mybatis.caches.ehcache.EhcacheCache"/>
```

通过 cache 元素的 name 属性，设置不同 Mapper 的缓存区域：

```
    <cache name="cn.wolfcode.mybatis.cache.mapper.UserMapper"
           maxElementsInMemory="10000"
           eternal="true"
           timeToIdleSeconds="300"
           timeToLiveSeconds="600"
           overflowToDisk="true"
           />
```

8. 第八章 MyBatis Generator

MyBatis Generator 简称 MBG，是一个专门为 iBatis、MyBatis 框架使用者提供的代码生成器，可以快速的根据表生成对应的模型对象、Mapper 接口、Mapper 文件，甚至生成 QBC 风格查询对象。

MBG 支持基本的增删改查操作，也支持 QBC 风格的条件查询，但是复杂的查询还是需要我们写 SQL 完成。

- 官方文档地址

http://www.mybatis.org/generator/

- 官方项目地址

https://github.com/mybatis/generator/releases

运行 MyBatis Generator：

- 方式一：使用 Java 代码运行（目前选用）
- 方式二：使用 Maven 插件运行（项目中选用）

8.1. 配置详解

MBG 配置文件，一般叫做 generatorConfig.xml。

看文档 Mybatis Generator 详细配置.docx 文档。

常用配置：

```
<?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>
  <context id="mysql" defaultModelType="hierarchical" targetRuntime="MyBatis3Simple">
    <!-- 生成的 Java 文件的编码 -->
    <property name="javaFileEncoding" value="UTF-8" />
    <!-- beginningDelimiter 和 endingDelimiter：指明数据库的用于标记数据库对象名的符号 -->
    <property name="beginningDelimiter" value="`" />
    <property name="endingDelimiter" value="`" />

    <!-- 注释生成器 -->
    <commentGenerator>
      <property name="suppressDate" value="true" />
      <property name="suppressAllComments" value="true" />
    </commentGenerator>

    <!-- 必须要有的，使用这个配置链接数据库 @TODO:是否可以扩展 -->
    <jdbcConnection driverClass="com.mysql.jdbc.Driver"
            connectionURL="jdbc:mysql://localhost:3306/mybatisdemo"
            userId="root"
            password="admin">
    </jdbcConnection>

    <!-- 生成 domain 对象 -->
    <javaModelGenerator
```

```
        targetPackage="cn.wolfcode.mybatis.domain" targetProject="src/main/java">
        <property name="enableSubPackages" value="true" />
      </javaModelGenerator>

      <!-- 生成 Mapper 文件 -->
      <sqlMapGenerator
          targetPackage="cn.wolfcode.mybatis.mapper" targetProject="src/main/resources">
        <property name="enableSubPackages" value="true" />
      </sqlMapGenerator>

      <!-- 生成 Mapper 接口 -->
      <javaClientGenerator targetPackage="cn.wolfcode.mybatis.mapper"
                           type="XMLMAPPER" targetProject="src/main/java">
        <property name="enableSubPackages" value="true" />
      </javaClientGenerator>


      <!-- ============================================================== -->
      <table tableName="logininfo" delimitIdentifiers="true" domainObjectName="Logininfo">
        <property name="useActualColumnNames" value="true" />
        <generatedKey column="id" sqlStatement="JDBC" />
      </table>
      <!-- ============================================================== -->
    </context>
  </generatorConfiguration>
```

8.2. 启动类

使用 Java 代码启动 MBG：

```
  public class Generator {
    public static void main(String[] args) throws Exception {
      //MBG 执行过程中的警告信息
      List<String> warnings = new ArrayList<String>();
      //生成代码重复时，是否覆盖源代码
      boolean override = false;
      InputStream in = Thread.currentThread().getContextClassLoader()
                              .getResourceAsStream("generatorConfig.xml");
      ConfigurationParser cp = new ConfigurationParser(warnings);
      Configuration config = cp.parseConfiguration(in);

      DefaultShellCallback callback = new DefaultShellCallback(override);
```

```
        //创建 MBG
        MyBatisGenerator mbg = new MyBatisGenerator(config, callback, warnings);
        mbg.generate(null);
        //输出警告信息
        for (String warn : warnings) {
          System.out.println(warn);
        }
      }
    }
```

8.3. 基本案例

MBG 生成基本的 CRUD 代码。

Context 元素：targetRuntime= “MyBatis3Simple”可以生成基本的增删改查。

```
<context id="mysql" defaultModelType="hierarchical" targetRuntime="MyBatis3Simple">
```

8.4. QBC 案例

MBG 生成 QBC 风格的 CRUD 代码。

QBC(Query By Criteria) API 提供了检索对象的另一种方式，它主要由 Criteria 接口和 Example 类组成，它支持在运行时动态生成查询语句，适用于简单的查询。

Context 元素：targetRuntime="MyBatis3"可以生成带条件的增删改查，缺省。

```
<context id="mysql" defaultModelType="hierarchical" targetRuntime="MyBatis3">
```

演示 QBC 查询操作。

查询 ID 为 10 的员工

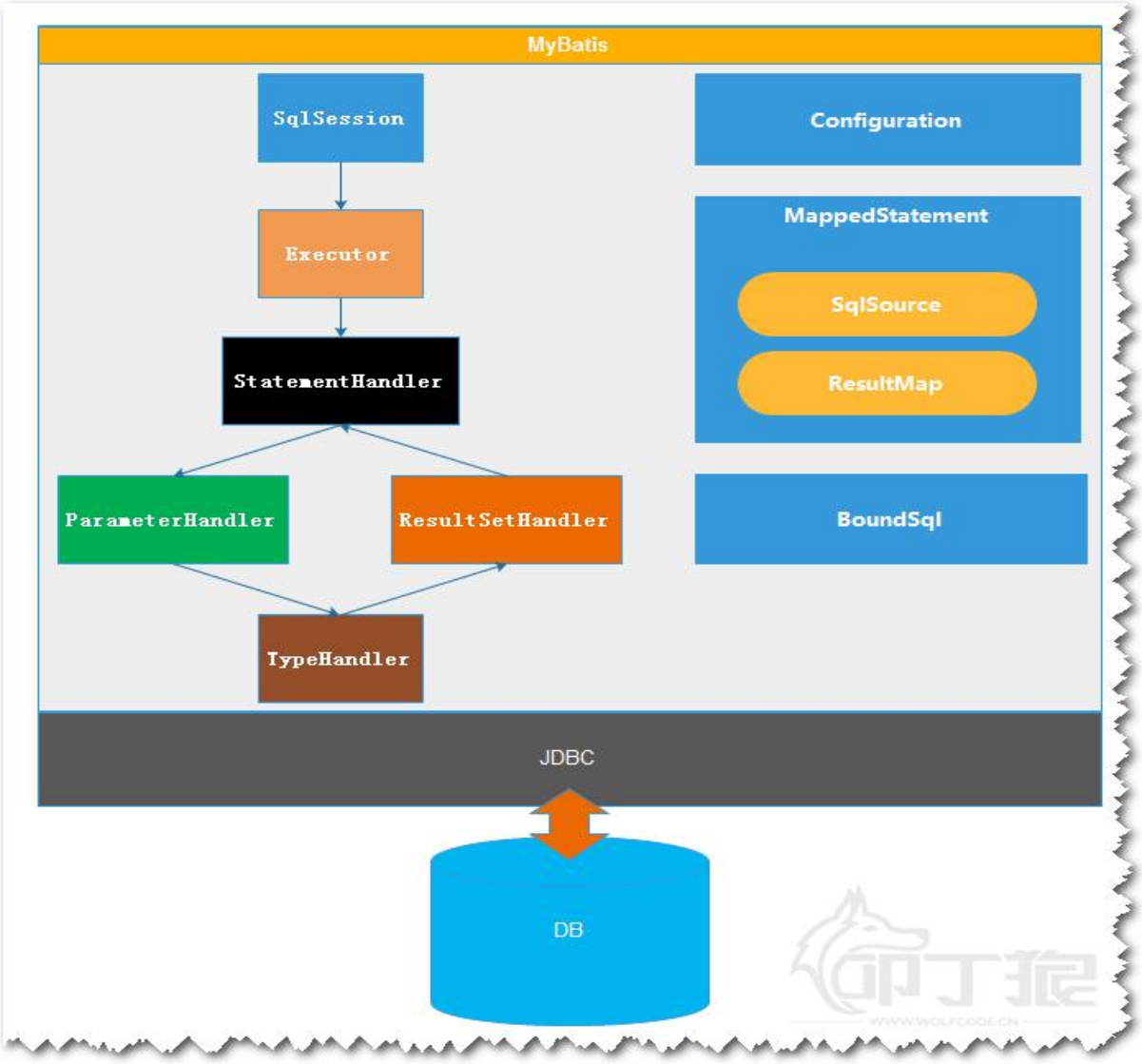
```
EmployeeKey key = new EmployeeKey();
key.setId(10L);
Employee e = employeeMapper.selectByPrimaryKey(key );
```

查询名字带有 a 且工资在 800~100 之间的员工

```
EmployeeExample example = new EmployeeExample();
Criteria criteria = example.createCriteria();
criteria.andNameLike("%a%");
criteria.andSalaryBetween(new BigDecimal("800"), new BigDecimal("1000"));
List<Employee> list = employeeMapper.selectByExample(example );
```

9. 第九章 插件开发

查看 MyBatis 执行的原理图，找出四大组件（Executor、StatementHandler、ParameterHandler、ResultSetHandler）。



MyBatis 在四大组件对象的创建过程中，都会有插件进行调用执行。我们可以利用动态机制对目标对象实施拦截增强操作，也就是在目标对象执行目标方法之前进行拦截增强的效果。

MyBatis 允许在已映射语句执行过程中的某一个时机来进行拦截增强，我们把这种机制称之为插件，其实就是动态代理。

默认情况下，MyBatis 允许使用插件来拦截的接口和包含方法的包括：

- Executor(update, query, flushStatements, commit, rollback, getTransaction, close, isClosed)
- ParameterHandler(getParameterObject, setParameters)

- ResultSetHandler(handleResultSets, handleOutputParameters)
- StatementHandler(prepare, parameterize, batch, update, query)

Executor 接口:

int update 方法会在所有的 INSERT、UPDATE、DELETE 执行时被调用。

query 方法在所有 SELECT 查询方法执行时被调用，是最常被拦截的方法。

ParameterHandler 接口：

setParameters 方法：设置 SQL 参数，在执行 SQL 前调用

ResultSetHandler 接口：

handleResultSets 方法：处理结果集，会在查询方法中会被调用

StatementHandler 接口：

prepare 方法：创建预编译语句，在数据库执行前被调用

parameterize 方法：在 prepare 方法之后执行，用于处理参数信息

query 方法：执行 SELECT 方法时被调用

9.1.1.插件开发

开发步骤：

1、编写插件实现 Interceptor 接口，并使用@Intercepts 注解完成插件签名

2、在全局配置文件中使用<plugin>元素注册插件

@Signature 注解：

type:拦截接口，只能是四种接口

method:设置拦截接口中的方法名，只能是四种接口中的方法，且要和接口匹配。

args:设置拦截方法的参数类型

9.1.2.把 Map 中 key 的下划线风格转成驼峰表示法

```
@Intercepts({ @Signature(type = ResultSetHandler.class, //
        method = "handleResultSets", //
        args = { Statement.class })//
})
public class CamelCaseInterceptor implements Interceptor {
    public Object plugin(Object target) {
        return Plugin.wrap(target, this);
    }

    public void setProperties(Properties properties) {
    }
```

```
    public Object intercept(Invocation invocation) throws Throwable {
        List<Object> ret = (List<Object>) invocation.proceed();
        for (Object obj : ret) {
            if (!(obj instanceof Map)) {
                break;
            }
            handleMap((Map<String, Object>) obj);
        }
        return ret;
    }

    private void handleMap(Map<String, Object> map) {
        Set<String> keySet = new HashSet<>(map.keySet());
        for (String key : keySet) {
            if (key.startsWith("A") && key.endsWith("Z") || key.contains("_")) {
                Object val = map.get(key);
                map.remove(key);
                String newKey = handleKey(key);
                map.put(newKey, val);
            }
        }
    }

    private String handleKey(String key) {
        StringBuilder sb = new StringBuilder(40);
        boolean findUnderLine = false;
        System.out.println(key);
        for (int index = 0; index < key.length(); index++) {
            char ch = key.charAt(index);
            if (ch == '_') {
                findUnderLine = true;
            } else {
                if(findUnderLine) {
                    sb.append(Character.toUpperCase(ch));
                    findUnderLine = false;
                }else {
                    sb.append(Character.toLowerCase(ch));
                }
            }
        }
        return sb.toString();
    }
}
```