一、MyBatis使用

概述

1.JDBC编程

什么是JDBC

- JDBC是Java与数据库交互的统一API，实际上它分为两组API，一组是面向Java应用程序开发人员的API，另一组是面向数据库驱动程序开发人员的API。
- Java程序都是通过JDBC(Java Data Base Connectivity)连接数据库的，这样我们就可以通过SQL对数据库编程。

- JDBC是由SUN公司提出的一系列规范，但是它只定义了接口规范，而具体的实现是交由各个数据库厂商去实现的，因为每个数据库都有其特殊性，所以JDBC就是一种典型的桥接模式。

  JDBC编程过程

```
public void func() throws Exception {
    Class.forName("com.mysql.jdbc.Driver");
    String url = "jdbc:mysql://localhost:3306/test";
    String username = "root";
    Connection connection = DriverManager.getConnection(url, username, null);
    PreparedStatement ps = connection.prepareStatement("SELECT * FROM tb_name LIMIT 1
");
    ResultSet rs = ps.executeQuery();
    while (rs.next()) {
        System.out.println("Id = " + rs.getString("id") + "  Name = " + rs.getString(
"name"));
    }
    rs.close();
    ps.close();
    connection.close();
}
```

- 使用JDBC编程需要连接数据库,注册驱动和数据库信息。

- 操作Connection，打开Statement对象。

- 通过Statement执行SQL，返回结果到ResultSet对象。

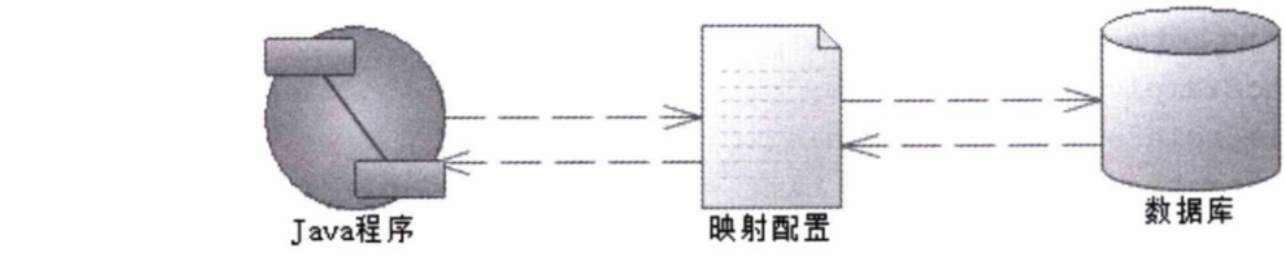
- 使用ResultSet读取数据，并将数据转换成JavaBean对象。
- 关闭ResultSet、Statement对象以及数据库连接，释放相关资源。

传统JDBC的弊端

1. 工作量相对较大，我们需要先连接、处理JDBC底层事务、处理数据类型；还要操作Connection对象、Statement对象、Resultset对象去拿到数据，并准确关闭它们，并且每次数据库操作都需要类似的重复性代码。
2. 我们要对JDBC编程可能产生的异常进行捕捉处理并正确关闭资源。
3. 实践中，为了提高代码的可维护性，可以将上述重复性代码封装到一个类似DBUtils的工具类中，步骤4中完成了关系模型到对象模型的转换，要使用比较通用的方式封装这种复杂的转换是比较困难的。

2.什么是ORM

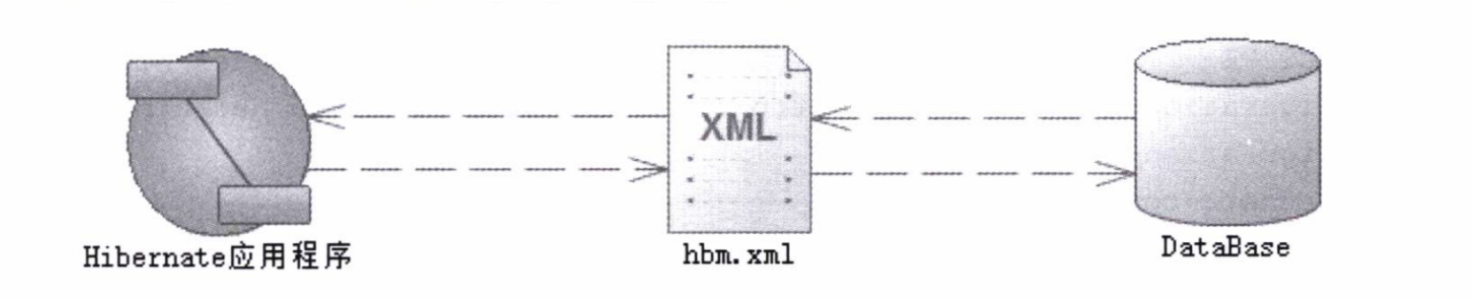
ORM（Object Relational Mapping，对象关系映射）模型就是数据库的表和简单Java对象(POJO)的映射关系模型，它主要解决数据库数据和POJO对象的相互映射。通过这层映射关系就可以简单迅速地把数据库表的数据转化为POJO，以便程序员更加容易理解和应用Java程序。



3.常见的持久化框架

Hibernate

- Hibernate是一款Java世界中最著名的ORM框架之一，作为一个老牌的ORM框架，Hibernate经受住了JavaEE企业级应用的考验，替代了复杂的JavaEE中EJB解决方案，一度成为Java ORM领域的首选框架。
- Hibernate是建立在若干POJO通过XML映射文件（或注解）提供的规则映射到数据库表上的。换句话说，我们可以通过POJO直接操作数据库的数据。它提供的是一种全表映射的模型，使用者不需要编写SQL语言，只要使用HQL语言(Hibernate Query Langurage)就可以了。



Hibernate的优势

- 消除了代码的映射规则，它全部被分离到了XML或者注解里面去配置，实现对象模型与关系模型的映射。
- 无需再管理数据库连接，它也配置在XML里面，同时帮助开发人员屏蔽不同数据库产品中SQL语句的细微差异。
- 一个会话中，不要操作多个对象，只要操作 Session对象即可，关闭资源只需要关闭一个Session便可。
- Hibernate的API没有侵入性，业务逻辑不需要继承Hibernate的任何接口。
- 默认提供一级缓存和二级缓存，这有利于提高系统的性能，降低数据库的压力。
- 支持透明的持久化、延迟加载、由对象模型自动生成数据库表

Hibernate的不足

1. 全表映射带来的不便，比如更新时需要发送所有的字段。
2. 无法根据不同的条件组装不同的SQL。
3. 对多表关联和复杂SQL查询支持较差，需要自己写SQL，返回后自己将数据组装为POJO。
4. 不能有效支持存储过程，对批处理的支持并不是很友好。
5. 虽然有HQL，但是性能较差，大型互联网系统往往需要优化SQL，但我们很难修改Hibernate生成的SQL语句，当数据量比较大、数据结构比较复杂时，Hibernate生成的SQL语句会非常复杂，且要让生成的SQL语句使用正确的索引也比较困难，这就会导致大量慢查询的情况。在有些大数据量、高并发、低延迟的场景下，Hibernate并不是很合适。

JPA

JPA（Java Persistence API)是SUN官方推出的Java持久化规范，它为Java开发人员提供了一种对象/关联映射工具来管理Java应用中的关系数据。它的出现主要是为了简化现有的持久化开发工作和整合ORM技术，结束现在Hibernate，TopLink，JDO等ORM框架各自为营的局面。这里需要了解的是，JPA仅仅是一个持久化的规范，它并没有提供具体的实现，其他持久化厂商会提供JPA规范的具体实现。JPA规范的愿景很美好，但是并没有得到很好的发展，现在在实践中的出场率也不是很高。

Spring JDBC

- 严格来说，Spring JDBC并不能算是一个ORM框架，它仅仅是使用模板方式对原生JDBC进行了

一层非常薄的封装。使用Spring JDBC可以帮助开发人员屏蔽创建数据库连接对、Statement对象、异常处理以及事务管理的重复性代码，提高开发效率。
Spring JDBC中没有映射文件、对象查询语言、缓存等概念，而是直接执行原生SQL语句。

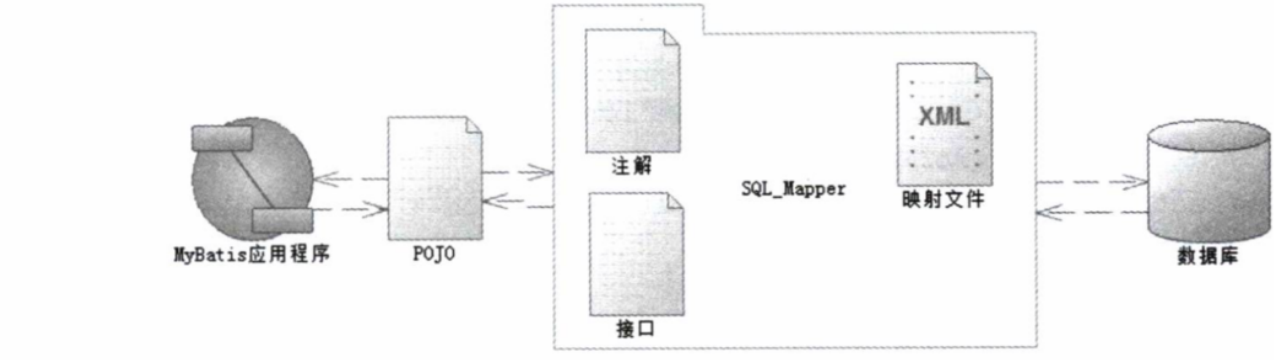
- Spring JDBC中提供了多种Template类，可以将对象中的属性映射成SQL语句中绑定的参数，Spring JDBC还提供了很多ORM化的Callback，这些Callback可以将ResultSet转化成相应的对象列表。在有些场景中，我们需要直接使用JDBC原生对象，例如，操作JDBC原生的ResultSet，则可以直接返回SqlRowSet对象，该对象是原生ResultSet对象的简单封装。
- Spring JDBC在功能上不及Hibernate强大，但它凭借高度的灵活性，也在Java持久化中占有一席之地。

MyBatis

为了解决 Hibernate的不足，一个半自动映射的框架MyBatis应运而生。
MyBatis前身是Apache基金会的开源项目iBatis，在2010年该项目脱离Apache基金会并正式更名为MyBatis。在2013年11月，MyBatis迁移到了GitHub。
目前，越来越多的互联网公司开始使用MyBatis，其中包括阿里巴巴、网易、搜狗、华为等，依赖MyBatis搭建的创业项目更是数不胜数。

什么是半自动

- MyBatis之所以是半自动的，是因为它需要手工匹配提供POJO、SQL和映射关系，而全表映射的Hibernate只需要提供POJO和映射关系便可。
- MyBatis所需要提供的映射文件包含以下三个部分：SQL、映射规则、POJO。



- 相较于 Hibernate，MyBatis 更加轻量级，可控性也更高，在使用 MyBatis 时我们直接在映射配置文件中编写待执行的原生SQL语句，这就给了我们直接优化 SQL 语句的机会，让SQL语句选择合适的索引，能更好地提高系统的性能，比较适合大数据量、高并发等场景。在编写SQL语句时，我们也可以比较方便地指定查询返回的列，而不是查询所有列并映射对象后返回， 这在列比较多的时候也能起到一定的优化效果。

如何选择合适的持久化框架?

- 从性能角度来看，Hibernate 生成的 SQL 语句难以优化，Spring JDBC 和 MyBatis 直接 使用原生SQL语句，优化空间比较大，MyBatis 和 Hibernate有设计良好的缓存机制，三者都可以与第三方数据源配合使用；
- 从可移植性角度来看，Hibernate 帮助开发人员屏蔽了底层数据库 方言，而 Spring JDBC和MyBatis在该方面没有做很好的支持，但实践中很少有项目会来回切 换底层使用的数据库产品，所以这点并不是特别重要；
- 从开发效率的角度来看，Hibernate 和 MyBatis 都提供了XML映射配置文件和注解两种方式实现映射，Spring JDBC 则是通过 ORM 化的 Callback的方式进行映射。在进行技术选型时，可以从更多角度进行比较，权衡性能、可扩展性、开发人员技术栈等多个方面选择合适的框架。

MyBatis使用示例

Idea代码展示

- mybatis-config.xml 配置文件

```
xml <?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration PUBLIC ... >
<configuration>
<properties> <!-- 定义属性值 -->
<property name="username" value="root"/>
<property name="id" value="123"/>
</properties>
<settings><!-- 全局配置信息 --> <setting name="cacheEnabled" value="true"/>
... ...
</settings>
<typeAliases> <!-- 配置别名信息，在映射配置文件中可以直接使用Blog这个别名代替
com.xxx.Blog这个类 -->
<typeAlias type="com.xxx.Blog" alias="Blog"/> ... ...
</typeAliases>
<environments default="development">
<environment id="development"> <!-- 配置事务管理器的类型 -->
<transactionManager type="JDBC"/> <!-- 配置数据源的类型，以及数据库连接的相关信息 -->
<dataSource type="POOLED">
<property name="driver" value="com.mysql.jdbc.Driver"/>
<property name="url" value="jdbc:mysql://localhost:3306/test"/>
<property name="username" value="root"/>
<property name="password" value=""/>
</dataSource>
</environment>
</environments> <!-- 配置映射配置文件的位置 -->
<mappers>
<mapper resource="com/xxx/BlogMapper.xml"/>
</mappers>
</configuration>
```

- UserMapper.xml 映射配置文件

```
xml <?xml version="1.0" encoding="UTF-8"?> <!DOCTYPE mapper PUBLIC "-
//mybatis.org//DTD Mapper 3.0//EN" ... >
<mapper namespace="com.xxx.BlogMapper"> <!-- 定义映射规则 -->
<resultMap id="detailedBlogResultMap" type="Blog">
<constructor> <!-- 构造函数映射 -->
<idArg column="blog_id" javaType="int"/>
</constructor> <!-- 属性映射 -->
<result property="title" column="blog_title"/> <!-- 对象属性的映射，同时也是一个嵌套
映射，后面会详细分析嵌套映射的处理过程 -->
<association property="author" resultMap="authorResult"/> <!-- 集合映射，也是一个
匿名的嵌套映射 -->
<collection property="posts" ofType="Post">
<id property="id" column="post_id"/>
<result property="content" column="post_content"/>
</collection>
</resultMap>
<resultMap id="authorResult" type="Author">
<id property="id" column="author_id"/>
<result property="username" column="username"/>
<result property="password" column="password"/>
<result property="email" column="email"/>
</resultMap>
<!-- 定义SQL语句，除了select节点，还可以定义insert、update、delete节点。为了便于描述， 后
面统称为“SQL节点” -->
<select id="selectBlogDetails" resultMap="detailedBlogResultMap">
select B.id as blog_id, B.title as blog_title, B.author_id as blog_author_id,
A.id as author_id, A.username as author_username, A.password as
author_password, A.email as author_email, P.id as post_id, P.blog_id as
post_blog_id, P.content as post_content
from Blog B
left outer join Author A
on B.author_id = A.id
left outer join Post P
on B.id = P.blog_id
where B.id = #{id}
</select>
</mapper>
```

MyBatis整体架构

MyBatis 的整体架构分为三层，分别是基础支持层、核心处理层和接口层。

- 接口层
  - SqlSession

- 核心处理层
  - 配置解析
  - 参数映射
  - SQL解析
  - SQL执行
  - 结果集映射
  - 插件
- 基础支持层
  - 数据源模块
  - 事务管理模块
  - 缓存模块
  - Binding模块
  - 反射模块
  - 类型转换
  - 日志模块
  - 资源加载
  - 解析器模块

SqlSession执行流程源码分析

1.Mapper的动态代理

我们自定义的Mapper接口想要发挥功能，必须有具体的实现类，在MyBatis中是通过为Mapper每个接口提供一个动态代理类来实现的。整个过程主要有四个类：MapperRegistry、MapperProxyFactory、MapperProxy、MapperMethod。

- MapperRegistry是Mapper接口及其对应的代理对象工厂的注册中心。
- MapperProxyFactory就是MapperProxy的工厂类，主要方法就是包装了Java动态代理的Proxy.newProxyInstance()方法。
- MapperProxy就是一个动态代理类，它实现了InvocationHandler接口。对于代理对象的调用都会被代理到InvocationHandler#invoke方法上。

- MapperMethod包含了具体增删改查方法的实现逻辑。

```
public class MapperRegistry {
//Configuration对象，MyBatis全局唯一的配置对象，其中包含了所有配置信息
private final Configuration config;
//记录了Mapper接口与对应MapperProxyFactory之间的关系
private final Map<Class<?>, MapperProxyFactory<?>> knownMappers = new HashMap();
public <T> T getMapper(Class<T> type, SqlSession sqlSession) {
```

```
//查找指定type对应的MapperProxyFactory对象
    MapperProxyFactory<T> mapperProxyFactory = (MapperProxyFactory)this.knownMappers.
get(type);
    //如果mapperProxyFactory为空，则抛出异常
    if (mapperProxyFactory == null) {
        throw new BindingException("Type " + type + " is not known to the MapperRegis
try.");
    } else {
        try {
        //创建实现了type接口的代理对象
            return mapperProxyFactory.newInstance(sqlSession);
        } catch (Exception var5) {
            throw new BindingException("Error getting mapper instance. Cause: " + var
5, var5);
        }
    }
}
public <T> void addMapper(Class<T> type) {
    //检测type是否为接口
    if (type.isInterface()) {
    //检测是否已经加载过该接口
        if (this.hasMapper(type)) {
            throw new BindingException("Type " + type + " is already known to the Map
perRegistry.");
        }
        boolean loadCompleted = false;
        try {
        //将Mapper接口对应的Class对象和MapperProxyFactory对象添加到knownMappers集合
            this.knownMappers.put(type, new MapperProxyFactory(type));
            //涉及XML解析和注解的处理
            MapperAnnotationBuilder parser = new MapperAnnotationBuilder(this.config,
 type);
            parser.parse();
            loadCompleted = true;
        } finally {
            if (!loadCompleted) {
                this.knownMappers.remove(type);
            }
        }
    }
}
}
```

```
public class MapperProxyFactory<T> {
```

```
  //当前MapperProxyFactory对象可以创建实现了mapperInterface接口的代理对象
  private final Class<T> mapperInterface;
  //缓存，key是mapperInterface接口中某方法对应的Method对象，value是对应的MapperMethod对象
  private final Map<Method, MapperMethod> methodCache = new ConcurrentHashMap();

  // 这里可以看到是通过Java的动态代理来实现的，具体代理的方法被放到来MapperProxy中
  protected T newInstance(MapperProxy<T> mapperProxy) {
    return (T) Proxy.newProxyInstance(mapperInterface.getClassLoader(), new Class[] { mapp
erInterface }, mapperProxy);
  }

  public T newInstance(SqlSession sqlSession) {
    //创建MapperProxy对象，每次调用都会创建新的MapperProxy对象
    final MapperProxy<T> mapperProxy = new MapperProxy<T>(sqlSession, mapperInterface, met
hodCache);
    return newInstance(mapperProxy);
  }
}
```

```
// 实现了InvocationHandler接口
public class MapperProxy<T> implements InvocationHandler, Serializable {
  //记录了关联的SqlSession对象
  private final SqlSession sqlSession;
  //Mapper接口对应的Class对象
  private final Class<T> mapperInterface;
  //用于缓存MapperMethod对象，其中key是Mapper接口中方法对应的Method对象，value是对应的MapperMet
hod对象。MapperMethod对象会完成参数转换以及SQL语句的执行功能。
  //需要注意的是，MapperMethod中并不记录任何状态相关的信息，所以可以在多个代理对象之间共享
  private final Map<Method, MapperMethod> methodCache;

  // 对代理类的所有方法的执行，都会进入到invoke方法中
  @Override
  public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
    // 此处判断是否是Object类的方法，如toString()、clone()，如果是则直接执行不进行代理
    if (Object.class.equals(method.getDeclaringClass())) {
      try {
        return method.invoke(this, args);
      } catch (Throwable t) {
        throw ExceptionUtil.unwrapThrowable(t);
      }
    }
    // 如果不是Object类的方法，则初始化一个MapperMethod并放入缓存中
    // 或者从缓存中取出之前的MapperMethod
    final MapperMethod mapperMethod = cachedMapperMethod(method);
```

```
    // 调用MapperMethod.execute()方法执行SQL语句
    return mapperMethod.execute(sqlSession, args);
  }

   private MapperMethod cachedMapperMethod(Method method) {
        //在缓存中查找MapperMethod，若没有，则创建MapperMethod对象，并添加到methodCache集合中缓
存
        return (MapperMethod)this.methodCache.computeIfAbsent(method, (k) -> {
            return new MapperMethod(this.mapperInterface, method, this.sqlSession.getConfi
guration());
        });
    }
}
```

```
public class MapperMethod {
  //记录了SQL语句的名称和类型
  private final MapperMethod.SqlCommand command;
  //Mapper接口中对应方法的相关信息
  private final MapperMethod.MethodSignature method;

  // MapperMethod采用命令模式运行，根据上下文跳转，它可能跳转到许多方法中
  // 实际上它最后就是通过SqlSession对象去运行对象的SQL。
  public Object execute(SqlSession sqlSession, Object[] args) {
    Object result;
    switch (command.getType()) {
      case INSERT: { //...
      }
      case UPDATE: { //...
      }
      case DELETE: { //...
      }
      case SELECT:
        if (method.returnsVoid() && method.hasResultHandler()) {
          executeWithResultHandler(sqlSession, args);
          result = null;
        } else if (method.returnsMany()) {
          result = executeForMany(sqlSession, args);
        } else if (method.returnsMap()) {
          result = executeForMap(sqlSession, args);
        } else if (method.returnsCursor()) {
          result = executeForCursor(sqlSession, args);
        } else {
          Object param = method.convertArgsToSqlCommandParam(args);
          result = sqlSession.selectOne(command.getName(), param);
```

```
        }
        break;
      case FLUSH:
        //...
      default:
        throw new BindingException("Unknown execution method for: " + command.getName());
    }

    return result;
  }
}
```

2.SqlSession中的对象

Mapper执行的过程是通过Executor、StatementHandler、ParameterHandler和ResultHandler来完成数据库操作和结果返回的：

- Executor代表执行器，由它来调度StatementHandler、ParameterHandler、ResultHandler等来执行对应的SQL。
- StatementHandler的作用是使用数据库的Statement（PreparedStatement）执行操作，起到承上启下的作用。
- ParameterHandler用于SQL对参数的处理。
  ResultHandler是进行最后数据集（ResultSet）的封装返回处理的。

3.执行器Executor

执行器是一个真正执行Java和数据库交互的类，一共有三种执行器，我们可以在MyBatis的配置文件中设置defaultExecutorType属性进行选择。

- SIMPLE（org.apache.ibatis.executor.SimpleExecutor），简易执行器，默认执行器。
- REUSE（org.apache.ibatis.executor.ReuseExecutor），是一种执行器重用预处理语句。
- BATCH（org.apache.ibatis.executor.BatchExecutor），执行器重用语句和批量更新，它是针对批量专用的执行器。

```
java // Configure类中创建Executor的具体逻辑
public Executor newExecutor(Transaction transaction, ExecutorType executorType)
{
executorType = executorType == null ? defaultExecutorType : executorType;
executorType = executorType == null ? ExecutorType.SIMPLE : executorType;
Executor executor;
if (ExecutorType.BATCH == executorType) {
executor = new BatchExecutor(this, transaction);
} else if (ExecutorType.REUSE == executorType) {
executor = new ReuseExecutor(this, transaction);
} else {
executor = new SimpleExecutor(this, transaction);
}
if (cacheEnabled) {
executor = new CachingExecutor(executor);
}
// 在executor完成创建之后，会通过interceptorChain来添加插件
executor = (Executor) interceptorChain.pluginAll(executor);
return executor;
}
```

创建Executor的具体逻辑在Configure类中，可以看到，在Executor创建完成之后，会通过interceptorChain来添加插件，通过代理到方式，在调度真实的Executor方法之前执行插件代码来完成功能。

Executor的具体执行逻辑

我们通过SimpleExecutor来看一下Executor的具体执行逻辑：

1. 根据Configuration来构建StatementHandler
2. 然后使用prepareStatement方法，对SQL编译并对参数进行初始化
3. 在prepareStatement方法中，调用了StatementHandler的prepared进行了预编译和基础设置，然后通过StatementHandler的parameterize来设置参数并执行。

4. 包装好的Statement通过StatementHandler来执行，并把结果传递给resultHandler。

```
public class SimpleExecutor extends BaseExecutor {
@Override
public <E> List<E> doQuery(MappedStatement ms, Object parameter, RowBounds rowBounds,
 ResultHandler resultHandler, BoundSql boundSql) throws SQLException {
Statement stmt = null;
try {
  Configuration configuration = ms.getConfiguration();
  // (1)根据 Configuration来构建Statementhandler
```

```
  StatementHandler handler = configuration.newStatementHandler(wrapper, ms, parameter
, rowBounds, resultHandler, boundSql);
  stmt = prepareStatement(handler, ms.getStatementLog());
  return handler.<E>query(stmt, resultHandler);
} finally {
  closeStatement(stmt);
}
}
private Statement prepareStatement(StatementHandler handler, Log statementLog) throws
 SQLException {
Statement stmt;
Connection connection = getConnection(statementLog);
stmt = handler.prepare(connection, transaction.getTimeout());
handler.parameterize(stmt);
return stmt;
}
}
```

4.数据库会话器StatementHandler

StatementHandler就是专门处理数据库会话的，创建StatementHandler的过程在Configuration中。

```
public StatementHandler newStatementHandler(Executor executor, MappedStatement mapped
Statement, Object parameterObject, RowBounds rowBounds, ResultHandler resultHandler,
BoundSql boundSql) {
StatementHandler statementHandler = new RoutingStatementHandler(executor, mappedState
ment, parameterObject, rowBounds, resultHandler, boundSql);
statementHandler = (StatementHandler) interceptorChain.pluginAll(statementHandler);
return statementHandler;
}
```

```
public RoutingStatementHandler(Executor executor, MappedStatement ms, Object paramete
r, RowBounds rowBounds, ResultHandler resultHandler, BoundSql boundSql) {
switch (ms.getStatementType()) {
  case STATEMENT:
    delegate = new SimpleStatementHandler(executor, ms, parameter, rowBounds, resultH
andler, boundSql);
    break;
  case PREPARED:
    delegate = new PreparedStatementHandler(executor, ms, parameter, rowBounds, resul
tHandler, boundSql);
```

```
    break;
  case CALLABLE:
    delegate = new CallableStatementHandler(executor, ms, parameter, rowBounds, resul
tHandler, boundSql);
    break;
  default:
    throw new ExecutorException("Unknown statement type: " + ms.getStatementType());
}
@Override
public Statement prepare(Connection connection, Integer transactionTimeout) throws SQ
LException {
  return delegate.prepare(connection, transactionTimeout);
}
@Override
public void parameterize(Statement statement) throws SQLException {
  delegate.parameterize(statement);
}
}
```

很显然创建的真实对象是一个RoutingStatementHandler对象，它实现了接口StatementHandler。从RoutingStatementHandler的构造方法来看，它其实是使用来委派模式来把具体的StatementHandler类型隐藏起来，通过RoutingStatementHandler来统一管理。一共用三种具体的StatementHandler类型：SimpleHandler、PreparedStatementHandler、CallableStatementHandler。

通过StatementHandler看执行细节

在Executor的具体执行逻辑中，我们主要关注StatementHandler的prepared、parameterize两个方法。

```
public abstract class BaseStatementHandler implements StatementHandler {
public Statement prepare(Connection connection, Integer transactionTimeout) throws SQ
LException {
ErrorContext.instance().sql(boundSql.getSql());
Statement statement = null;
try {
  // instantiateStatement对SQL进行了预编译
  statement = instantiateStatement(connection);
  // 设置超时时间
  setStatementTimeout(statement, transactionTimeout);
  // 设置获取最大的行数
  setFetchSize(statement);
  return statement;
```

```
} catch (SQLException e) {
  closeStatement(statement);
  throw e;
} catch (Exception e) {
  closeStatement(statement);
  throw new ExecutorException("Error preparing statement.  Cause: " + e, e);
}
}
}
```

```
public class PreparedStatementHandler extends BaseStatementHandler {
// 调用parameterize去设置参数，可以发现是通过parameterHandler来具体执行的
public void parameterize(Statement statement) throws SQLException {
parameterHandler.setParameters((PreparedStatement) statement);
}
}
```

```
public class PreparedStatementHandler extends BaseStatementHandler {
// 具体的查询就是通过PreparedStatement#execute来执行的
public <E> List<E> query(Statement statement, ResultHandler resultHandler) throws SQL
Exception {
PreparedStatement ps = (PreparedStatement) statement;
ps.execute();
return resultSetHandler.<E> handleResultSets(ps);
}
}
```

5.参数处理器ParameterHandler

MyBatis是通过ParameterHandler对预编译的语句进行参数设置的。

```
public interface ParameterHandler {
// 返回参数对象
Object getParameterObject();
// 设置预编译的SQL语句的参数
void setParameters(PreparedStatement ps) throws SQLException;
}
```

```
public class DefaultParameterHandler implements ParameterHandler {
@Override
public void setParameters(PreparedStatement ps) {
```

```
ErrorContext.instance().activity("setting parameters").object(mappedStatement.getPara
meterMap().getId());
List<ParameterMapping> parameterMappings = boundSql.getParameterMappings();
if (parameterMappings != null) {
  for (int i = 0; i < parameterMappings.size(); i++) {
    ParameterMapping parameterMapping = parameterMappings.get(i);
    if (parameterMapping.getMode() != ParameterMode.OUT) {
      Object value;
      String propertyName = parameterMapping.getProperty();
      if (boundSql.hasAdditionalParameter(propertyName)) { // issue #448 ask first fo
r additional params
        value = boundSql.getAdditionalParameter(propertyName);
      } else if (parameterObject == null) {
        value = null;
      } else if (typeHandlerRegistry.hasTypeHandler(parameterObject.getClass())) {
        value = parameterObject;
      } else {
        MetaObject metaObject = configuration.newMetaObject(parameterObject);
        value = metaObject.getValue(propertyName);
      }
      TypeHandler typeHandler = parameterMapping.getTypeHandler();
      JdbcType jdbcType = parameterMapping.getJdbcType();
      if (value == null && jdbcType == null) {
        jdbcType = configuration.getJdbcTypeForNull();
      }
      try {
        typeHandler.setParameter(ps, i + 1, value, jdbcType);
      } catch (TypeException e) {
        throw new TypeException("Could not set parameters for mapping: " + parameterM
apping + ". Cause: " + e, e);
      } catch (SQLException e) {
        throw new TypeException("Could not set parameters for mapping: " + parameterM
apping + ". Cause: " + e, e);
      }
    }
  }
}
}
}
```

MyBatis为ParameterHandler提供了一个实现类DefaultParameterHandler，具体执行过程还是从 parameterObject对象中取参数然后使用typeHandler进行参数处理，而typeHandler也是在My Batis初始化的时候，注册在Configuration里面的，我们需要的时候可以直接拿来用。

6.ResultSetHandler

```
public interface ResultSetHandler {
  // 包装结果集的
  <E> List<E> handleResultSets(Statement stmt) throws SQLException;
  // 处理存储过程输出参数的
  <E> Cursor<E> handleCursorResultSets(Statement stmt) throws SQLException;
  void handleOutputParameters(CallableStatement cs) throws SQLException;
}
```

MyBatis为我们提供了一个DefaultResultSetHandler类，在默认的情况下都是通过这个类进行处理的。这个类JAVASSIST或者CGLIB作为延迟加载，然后通过typeHandler和ObjectFactory进行组装结果再返回。

整体结构：