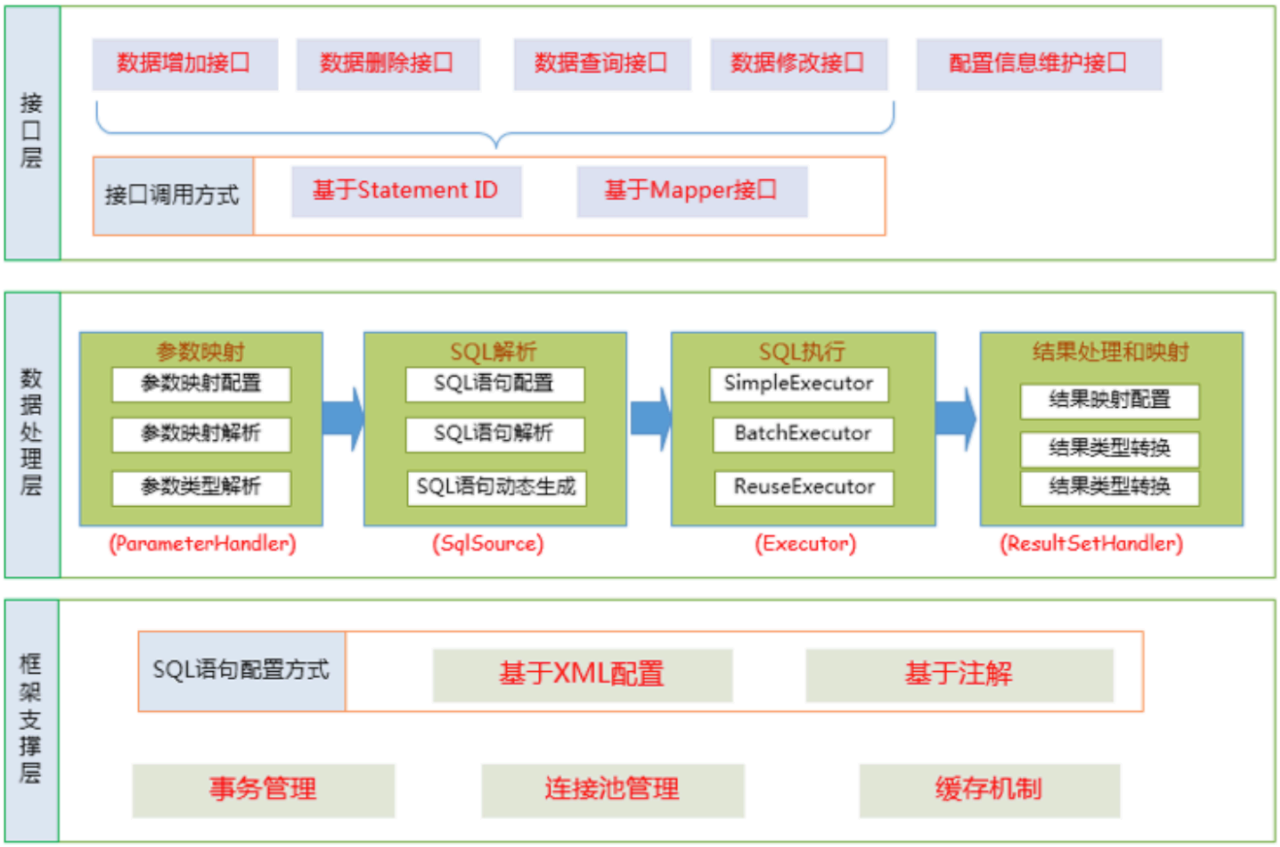
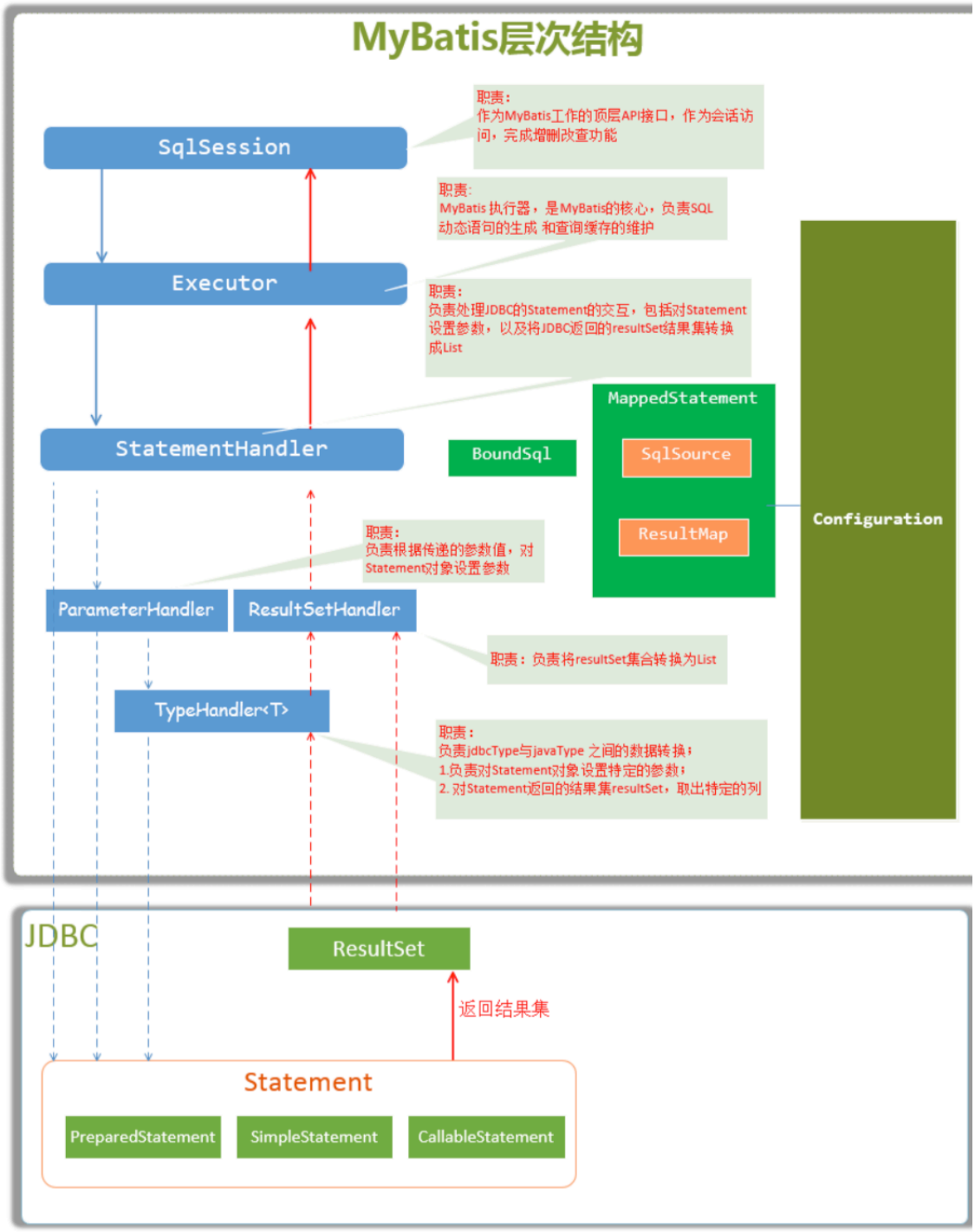
1. Mybatis架构原理

1.1 三层功能架构



1.2 层次结构


MyBatis层次结构
SqlSession
职责：
作为MyBatis工作的顶层API接口，作为会话访问，完成增删改查功能
Executor
职责：
MyBatis 执行器，是MyBatis的核心，负责SQL动态语句的生成 和查询缓存的维护
StatementHandler
职责：
负责处理JDBC的Statement的交互，包括对Statement设置参数，以及将JDBC返回的resultSet结果集转换成List
BoundSql
MappedStatement
SqlSource
ResultMap
Configuration
ParameterHandler
ResultSetHandler
职责：
负责根据传递的参数值，对Statement对象设置参数
职责：负责将resultSet集合转换为List
TypeHandler<T>
职责：
负责jdbcType与javaType 之间的数据转换；
1.负责对Statement对象设置特定的参数；
2. 对Statement返回的结果集resultSet，取出特定的列
JDBC
ResultSet
返回结果集
Statement
PreparedStatement
SimpleStatement
CallableStatement

1.3 总体流程

(1)加载配置并初始化

触发条件：加载配置文件

配置来源于两个地方，一个是配置文件(主配置文件conf.xml,mapper文件*.xml)，一个是java代码中的注解，将主配置文件内容解析封装到Configuration,将sql的配置信息加载成为一个mappedstatement对象，存储在内存之中

---

(2)接收调用请求

触发条件：调用Mybatis提供的API

传入参数：为SQL的ID和传入参数对象

处理过程：将请求传递给下层的请求处理层进行处理。

(3)处理操作请求

触发条件：API接口层传递请求过来

传入参数：为SQL的ID和传入参数对象

处理过程：

(A)根据SQL的ID查找对应的MappedStatement对象。

(B)根据传入参数对象解析MappedStatement对象，得到最终要执行的SQL和执行传入参数。

(C)获取数据库连接，根据得到的最终SQL语句和执行传入参数到数据库执行，并得到执行结果。

(D)根据MappedStatement对象中的结果映射配置对得到的执行结果进行转换处理，并得到最终的处理结果。

(E)释放连接资源。

(4)返回处理结果

将最终的处理结果返回。

2. Mybatis初始化过程

2.1 加载所有配置文件入内存

从2.开始正式初始化工作：

```
// 1. 读取配置文件，读成字节输入流放在内存，注意：现在还没解析
    InputStream resourceAsStream =
Resources.getResourceAsStream("sqlMapConfig.xml");

// 2. 解析配置文件，封装Configuration对象，创建DefaultSqlSessionFactory对象
    SqlSessionFactory sqlSessionFactory = new
SqlSessionFactoryBuilder().build(resourceAsStream);
```

2.2 解析配置文件，封装为Configuration对象

1. parse一切获得Configuration对象
2. 根据Configuration对象创建 DefaultSqlSessionFactory对象

> 这里用到了构建者设计模式

3. SQL语句执行流程（传统方式）

```
/**
 * 传统方式
 * @throws IOException
 */
public void test1() throws IOException {
  // 1. 读取配置文件，读成字节输入流，注意：现在还没解析
  InputStream resourceAsStream =
Resources.getResourceAsStream("sqlMapConfig.xml");

  // 2. 解析配置文件，封装Configuration对象    创建DefaultSqlSessionFactory对象
  SqlSessionFactory sqlSessionFactory = new
SqlSessionFactoryBuilder().build(resourceAsStream);

  // 3. 生产了DefaultSqlsession实例对象    设置了事务不自动提交   完成了executor对象的创建
  SqlSession sqlSession = sqlSessionFactory.openSession();

  // 4.(1)根据statementid来从Configuration中map集合中获取到了指定的MappedStatement对象
     //(2)将查询任务委派了executor执行器
  List<Object> objects = sqlSession.selectList("namespace.id");

  // 5.释放资源
  sqlSession.close();

}
```

1. 关于SqlSession

SqlSession is an Interface, 2个实现类：

- DefaultSqlSession（默认）
- SqlSessionManager（弃用，不介绍）

SqlSession是Mybatis中用于和DB交互的顶层类，通常将它与ThreadLocal绑定，一个会话使用一个SqlSession，线程不安全，使用完毕后要close。（详见Task2-7.随堂测试）

```
public class DefaultSqlSession implements SqlSession {

  private final Configuration configuration;
  private final Executor executor;

}
```

SqlSession在执行任务时，都会委派给自己的executor来执行。

2. 关于Executor

Executor is an Interface，3个常用实现类：

- BatchExecutor（重用语句来批量执行）
- ReuseExecutor（重用预处理语句preparedStatements)
- SimpleExecutor（普通的执行器，默认）

3.1 生产出DefaultSqlsession实例对象

```
// 3. 生产了DefaultSqlsession实例对象，同时设置了事务不自动提交，并完成了executor对象的创建
SqlSession sqlSession = sqlSessionFactory.openSession();
```

在DefaultSqlSessionFactory 类中：

```
// 3.1 进入openSession方法
@Override
public SqlSession openSession() {
    //getDefaultExecutorType()传递的是SimpleExecutor; level指DB数据的隔离级别;
autocommit指是否自动提交事务
    return openSessionFromDataSource(configuration.getDefaultExecutorType(),
null, false);
}
```

```
// 3.2 进入openSessionFromDataSource。
//ExecutorType 为Executor的类型，TransactionIsolationLevel为事务隔离级别，autoCommit是
否开启事务
```

```
//openSession的多个重载方法可以指定获得的SeqSession的Executor类型和事务的处理
private SqlSession openSessionFromDataSource(ExecutorType execType,
TransactionIsolationLevel level, boolean autoCommit) {
    Transaction tx = null;
    try {
        // 获得 Environment 对象
        final Environment environment = configuration.getEnvironment();
        // 创建 Transaction 对象
        final TransactionFactory transactionFactory =
getTransactionFactoryFromEnvironment(environment);
        tx = transactionFactory.newTransaction(environment.getDataSource(),
level, autoCommit);
        // 创建 Executor 对象
        final Executor executor = configuration.newExecutor(tx, execType);
        // 创建 DefaultSqlSession 对象
        return new DefaultSqlSession(configuration, executor, autoCommit);
    } catch (Exception e) {
        // 如果发生异常，则关闭 Transaction 对象
        closeTransaction(tx); // may have fetched a connection so lets call
close()
        throw ExceptionFactory.wrapException("Error opening session.  Cause: " +
e, e);
    } finally {
        ErrorContext.instance().reset();
    }
}
```

3.2 调用SqlSession对象的方法

```
// 4.(1)根据statementid来从Configuration中map集合中获取到了指定的MappedStatement对象
//   (2)将查询任务委派了executor执行器
List<Object> objects = sqlSession.selectList("namespace.id");
```

在DefaultSqlSession类中：

```
@Override
public <E> List<E> selectList(String statement, Object parameter, RowBounds
rowBounds) {
// RowBounds是管分页，这里没用
    try {
        // 获得 MappedStatement 对象
        MappedStatement ms = configuration.getMappedStatement(statement);
        // 是executor在执行查询
```

```
            return executor.query(ms, wrapCollection(parameter), rowBounds,
Executor.NO_RESULT_HANDLER);
        } catch (Exception e) {
            throw ExceptionFactory.wrapException("Error querying database.  Cause: "
+ e, e);
        } finally {
            ErrorContext.instance().reset();
        }
    }
```

3.3 Executor执行任务

这里query方法的实现是在BaseExecutor中，它是SimpleExecutor的父类。

在BaseExecutor类中：

```
//此方法在SimpleExecutor的父类BaseExecutor中实现
@Override
public <E> List<E> query(MappedStatement ms, Object parameter, RowBounds
rowBounds, ResultHandler resultHandler) throws SQLException {
    //根据传入的参数动态获得SQL语句，最后返回用BoundSql对象表示
    BoundSql boundSql = ms.getBoundSql(parameter);
    //为本次查询创建缓存的Key
    CacheKey key = createCacheKey(ms, parameter, rowBounds, boundSql);
    // 查询
    return query(ms, parameter, rowBounds, resultHandler, key, boundSql);
}
```

Query时，先去一级缓存找，如果没有，再去通过`doQuery`方法去数据库找。`doQuery`方法是个Interface，被实现在了SimpleExecutor类中。这里把任务继续向下交给StatementHanlder对象来执行`handler.query(...)`：

```
@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();
        // 传入参数创建StatementHanlder对象来继续执行查询
        StatementHandler handler = configuration.newStatementHandler(wrapper, ms,
parameter, rowBounds, resultHandler, boundSql);
        // 创建jdbc中的statement对象
        stmt = prepareStatement(handler, ms.getStatementLog());
        // 执行 StatementHandler  , 进行读操作
        return handler.query(stmt, resultHandler);
```

```
    } finally {
        // 关闭 StatementHandler 对象
        closeStatement(stmt);
    }
}
```

3.4 StatementHanlder执行任务

3.4.1 parameterHandler.parameterize(stmt)进行参数设置

```
//使用ParameterHandler对象来完成对Statement的设值
@Override
public void parameterize(Statement statement) throws SQLException {
    parameterHandler.setParameters((PreparedStatement) statement);
}
```

在DefaultParameterHandler类中，由typeHandler来设置？占位符的参数和该参数的jdbc类型：

`boundSql` 存储着

- 带“? ”占位符的sql语句
- #{ } 中的content名字

```
@Override
public void setParameters(PreparedStatement ps) {
    ErrorContext.instance().activity("setting
parameters").object(mappedStatement.getParameterMap().getId());
    // 遍历 ParameterMapping 数组
    List<ParameterMapping> parameterMappings = boundSql.getParameterMappings();
    if (parameterMappings != null) {
        for (int i = 0; i < parameterMappings.size(); i++) {
            // 一通操作获得 ParameterMapping 对象 和 值
            ...
                // 获得 typeHandler、jdbcType 属性
                TypeHandler typeHandler = parameterMapping.getTypeHandler();
                JdbcType jdbcType = parameterMapping.getJdbcType();
                if (value == null && jdbcType == null) {
                    jdbcType = configuration.getJdbcTypeForNull();
                }
                // 由typeHandler来设置 ? 占位符的参数和该参数的jdbc类型
                try {
                    typeHandler.setParameter(ps, i + 1, value, jdbcType);
                } catch (TypeException | SQLException e) {
                    throw new TypeException("Could not set parameters for
mapping: " + parameterMapping + ". Cause: " + e, e);
```

```
                }
            }
        }
    }
}
```

3.4.2 preparedStatementHandler.query(stmt, ResultSetHandler)

1. 用刚刚设置好参数的statement执行sql操作
2. ResultSetHandler.handleResultSets 来处理返回结果

3.4.3 ResultSetHandler.handleResultSets 来处理返回结果

```
@Override
public List<Object> handleResultSets(Statement stmt) throws SQLException {
    ErrorContext.instance().activity("handling
results").object(mappedStatement.getId());

    // 多 ResultSet 的结果集合，每个 ResultSet 对应一个 Object 对象。而实际上，每个
Object 是 List<Object> 对象。
    // 在不考虑存储过程的多 ResultSet 的情况，普通的查询，实际就一个 ResultSet ，也就是说，
multipleResults 最多就一个元素。
    final List<Object> multipleResults = new ArrayList<>();

    int resultSetCount = 0;
    // 获得首个 ResultSet 对象，并封装成 ResultSetWrapper 对象
    ResultSetWrapper rsw = getFirstResultSet(stmt);

    // 获得 ResultMap 数组
    // 在不考虑存储过程的多 ResultSet 的情况，普通的查询，实际就一个 ResultSet ，也就是说，
resultMaps 就一个元素。
    List<ResultMap> resultMaps = mappedStatement.getResultMaps();
    int resultMapCount = resultMaps.size();
    validateResultMapsCount(rsw, resultMapCount); // 校验
    while (rsw != null && resultMapCount > resultSetCount) {
        // 获得 ResultMap 对象
        ResultMap resultMap = resultMaps.get(resultSetCount);

        //【关注】 处理 ResultSet ，将结果添加到 multipleResults 中
        handleResultSet(rsw, resultMap, multipleResults, null);
        //【关注】 现在multipleResults中已经含有处理好的结果

        // 获得下一个 ResultSet 对象，并封装成 ResultSetWrapper 对象
        rsw = getNextResultSet(stmt);
```

```
            // 清理
            cleanUpAfterHandlingResultSet();
            // resultSetCount ++
            resultSetCount++;
        }

       ...

        // 如果是 multipleResults 单元素，则取首元素返回
        return collapseSingleResultList(multipleResults);
    }
```

4. Mapper代理方式

```
/**
 * mapper代理方式
 */
public void test2() throws IOException {

  InputStream inputStream = Resources.getResourceAsStream("sqlMapConfig.xml");
  SqlSessionFactory factory = new SqlSessionFactoryBuilder().build(inputStream);
  SqlSession sqlSession = factory.openSession();

  // 使用JDK动态代理对mapper接口产生代理对象
  IUserMapper mapper = sqlSession.getMapper(IUserMapper.class);

  //代理对象调用接口中的任意方法，执行的都是动态代理中的invoke方法
  // invoke方法执行时，是去调用sqlSession里面的增删改查方法来和DB交互
  List<Object> allUser = mapper.findAllUser();

}
```

Mybatis初始化同上。

关注2个问题：

- sqlSession.getMapper方法是如何对UserMapper Interface产生代理对象的
- 代理对象在调用方法时，其底层的invoke方法如何去执行的

4.1 批量加载mapper

加载sqlMapConfig.xml时候，若使用`<package>`：

```
<configuration>
    <mappers>
        <package name="com.lagou.mapper"/>
    </mappers>
</configuration>
```

那么，一通操作后，class MapperRegistry会生成一张hashmap来存映射关系：一个Interface对应一个MapperProxyFactory的对象。

4.2 sqlSession.getMapper

最终是落实到了`mapperRegistry.getMapper()`方法：

```
public class MapperRegistry {
  ...
public <T> T getMapper(Class<T> type, SqlSession sqlSession) {
        // 获得 MapperProxyFactory 对象
        final MapperProxyFactory<T> mapperProxyFactory = (MapperProxyFactory<T>)
knownMappers.get(type);
        // 不存在，则抛出 BindingException 异常
        if (mapperProxyFactory == null) {
            throw new BindingException("Type " + type + " is not known to the
MapperRegistry.");
        }
        /// 存在，通过动态代理工厂生成实例。
        try {
            return mapperProxyFactory.newInstance(sqlSession);
        } catch (Exception e) {
            throw new BindingException("Error getting mapper instance. Cause: " +
e, e);
        }
    }
  ...
```

MapperProxyFactory类中的newInstance方法:

```
    public T newInstance(SqlSession sqlSession) {
        // 创建了JDK动态代理的invocationHandler接口的实现类mapperProxy
        final MapperProxy<T> mapperProxy = new MapperProxy<>(sqlSession,
mapperInterface, methodCache);
        // 调用了重载方法
        return newInstance(mapperProxy); // 最终返回的是代理对象
    }
```

此处的代理对象是属于MapperProxy类，那么代理对象在调用Interface中的任何方法，都是去执行自属类（即MapperProxy类）的invoke方法。

4.3 mapper.任何方法 => MapperProxy.invoke()

```
public class MapperProxy<T> implements InvocationHandler, Serializable {
  ...
     @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
        try {
            // 如果是 Object 定义的方法，直接调用
            if (Object.class.equals(method.getDeclaringClass())) {
                return method.invoke(this, args);

            } else if (isDefaultMethod(method)) {
                return invokeDefaultMethod(proxy, method, args);
            }
        } catch (Throwable t) {
            throw ExceptionUtil.unwrapThrowable(t);
        }
        // 获得 MapperMethod 对象
        final MapperMethod mapperMethod = cachedMapperMethod(method);
        // 重点在这：MapperMethod最终调用了执行的方法
        return mapperMethod.execute(sqlSession, args);
    }
  ...
```