小细节：

TestController.class.getResourceAsStream("/excel/template/test.xlsx");

和

TestController.class.getClassLoader.getResourceAsStream("excel/template/test.xlsx")一样，getClassLoader相当于“/” (根路径的意思)。

使用properties方式添加数据源：

environment讲解：

其中，事务管理器（transactionManager）类型有两种：

•JDBC：这个配置就是直接使用了JDBC 的提交和回滚设置，它依赖于从数据源得到的连接来管理事务作用域。

•MANAGED：这个配置几乎没做什么。它从来不提交或回滚一个连接，而是让容器来管理事务的整个生命周期（比如 JEE 应用服务器的上下文）。默认情况下它会关闭连接，然而一些容器并不希望这样，因此需要将 closeConnection 属性设置为 false 来阻止它默认的关闭行为。

其中，数据源（dataSource）类型有三种：

•UNPOOLED：这个数据源的实现只是每次被请求时打开和关闭连接。

•POOLED：这种数据源的实现利用“池”的概念将 JDBC 连接对象组织起来。

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

typeAliases标签讲解：

4)typeAliases标签

类型别名是为Java 类型设置一个短的名字。原来的类型名称配置如下

配置typeAliases，为com.lagou.domain.User定义别名为user

上面我们是自定义的别名，mybatis框架已经为我们设置好的一些常用的类型的别名

别名	数据类型
string	String
long	Long
int	Integer
double	Double
boolean	Boolean
... ...	... ...

choose，when，otherwise(相当于Java中的switch)

- 只要有一个成立，其他都不执行
- 如果title和content都不为null或都不为""
  - 生成的sql中只有where title=?
- 如果title和content都为null或都为""
  - 生成的sql中只有where owner = “owner1”

```
<select id="dynamicChooseTest" parameterType="Blog" resultType="Blog">
    select * from t_blog where 1=1
    <choose>
        <when test="title != null">
            and title = #{title}
        </when>
        <when test="content != null">
            and content = #{content}
        </when>
        <!-- <otherwise>可以不写 -->
        <otherwise>
            and owner = "owner1"
        </otherwise>
    </choose>
</select>
```

trim(截断 添加)

- prefix 在前面添加内容
- suffix 在后面添加内容
- prefixOverrides 去掉前面内容
- suffixOverrides 去掉后面内容

```xml
<update id="upd" parameterType="log">
    update log
    <!-- 去掉了后面的内容 -->
    <!-- 覆盖了标签后的逗号 -->
    <!-- 适用于存在符号和关键字的参数(例如金钱符号$) -->
    <trim prefix="set" suffixOverrides>
        a=a,
    </trim>
    where id=100
</update>
```

bind（模糊查询）

- 作用：给参数重新赋值
- 场景：模糊查询 | 在原内容前或后添加内容

```xml
<select id="selByLog" parameterType="log" resultType="log">
    select * from log
    <where>
        <!-- 常用语模糊查询(添加%) -->
        <if test="title!=null and title!=''">
            <bind name="title" value="'$'+title+'$'"/>
            and title like #{title}
        </if>
        <!-- bind:给参数附加字符串 -->
        <if test="money!=null and money!=''">
            <bind name="money" value="'$'+money"/>
            and money = #{money}
        </if>
    </where>
</select>
```

foreach讲解:

属性

- collection：添加要遍历的集合
- item：迭代变量，循环内使用#{迭代变量名}来获取内容
- open：循环后左侧添加的内容
- close：循环后右侧添加的内容
- separator：添加每次遍历尾部追加的分割符

```xml
<select id="selIn" parameterType="list" resultType="log">
    select * from log where id in
    <foreach collection="list" item="a" open="(" close=")" separator=",">
        #{a}
    </foreach>
</select>
```

mybatis的一级缓存

- **mybaits中如何维护一级缓存**

  答： BaseExecutor成员变量之一的PerpetualCache，是对Cache接口最基本的实现， 其实现非常简单，内部持有HashMap，对一级缓存的操作实则是对HashMap的操作。

- **一级缓存的生命周期**

  MyBatis一级缓存的生命周期和SqlSession一致;

  MyBatis的一级缓存最大范围是SqlSession内部，有多个SqlSession或者分布式的环境下，数据库写操作会引起脏数据;

  MyBatis一级缓存内部设计简单，只是一个没有容量限定的HashMap，在缓存的功能性上有所欠缺。

- **mybatis 一级缓存何时失效**

  1. MyBatis在开启一个数据库会话时，会创建一个新的SqlSession对象， SqlSession对象中会有一个新的Executor对象， Executor对象中持有一个新的PerpetualCache对象；当会话结束时，SqlSession对象及其内部的Executor对象还有PerpetualCache对象也一并释放掉。
  2. 如果SqlSession调用了close()方法，会释放掉一级缓存PerpetualCache对象，一级缓存将不可用；
  3. 如果SqlSession调用了clearCache()，会清空PerpetualCache对象中的数据，但是该对象仍可使用；
  4. SqlSession中执行了任何一个update操作(update()、delete()、insert()) ，都会清空PerpetualCache对象的数据

- **一级缓存的工作流程?**

  1. 对于某个查询，根据statementId,params,rowBounds来构建一个key值，根据这个key值去缓存Cache中取出对应的key值存储的缓存结果；
  2. 判断从Cache中根据特定的key值取的数据数据是否为空，即是否命中；
  3. 如果命中，则直接将缓存结果返回；
  4. 如果没命中：去数据库中查询数据，得到查询结果；将key和查询到的结果分别作为key,value对存储到Cache中；将查询结果返回.

二级缓存整合redis：

导入依赖：

```
  <dependency>
    <groupId>org.mybatis.caches</groupId>
    <artifactId>mybatis-redis</artifactId>
    <version>1.0.0-beta2</version>
  </dependency>
```

配置文件：

```
<?xml version="1.0" encoding="UTF-8"?> <!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
 "http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<mapper namespace="com.lagou.mapper.IUserMapper">

<cache type="org.mybatis.caches.redis.RedisCache" />//在mapper文件添加该标签

<select id="findAll" resultType="com.lagou.pojo.User" useCache="true">
    select * from user
</select>
```

flushCache、useCache的区别：

如果没有去配置flushCache、useCache，那么默认是启用缓存的

- flushCache默认为false，表示任何时候语句被调用，都不会去清空本地缓存和二级缓存。
- useCache默认为true，表示会将本条语句的结果进行二级缓存。
- 在insert、update、delete语句时： flushCache默认为true，表示任何时候语句被调用，都会导致本地缓存和二级缓存被清空。 useCache属性在该情况下没有。update 的时候如果 flushCache="false"，则当你更新后，查询的数据数据还是老的数据。

下面的cache标签里面的属性代表的含义：

```
<cache eviction="FIFO" flushInterval="600000" size="4096" readOnly="true"/>
```

- eviction：缓存回收策略
  - LRU：最少使用原则，移除最长时间不使用的对象
  - FIFO：先进先出原则，按照对象进入缓存顺序进行回收
  - 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，更积极的移除移除基于垃圾回收器状态和弱引用规则的对象
- flushInterval：刷新时间间隔，单位为毫秒，这里配置的100毫秒。如果不配置，那么只有在进行数据库修改操作才会被动刷新缓存区
- size：引用额数目，代表缓存最多可以存储的对象个数
- readOnly：是否只读，如果为true，则所有相同的sql语句返回的是同一个对象（有助于提高性能，但并发操作同一条数据时，可能不安全），如果设置为false，则相同的sql，后面访问的是cache的clone副本。

redis.properties：注意命名是不能改的

```
redis.host=localhost
redis.port=6379
redis.connectionTimeout=5000
redis.password=
redis.database=0
```

扫描mapper注解的两种方式：

修改MyBatis的核心配置文件，我们使用了注解替代的映射文件，所以我们只需要加载使用了注解的Mapper接口即可

```
<mappers>
    <!--扫描使用注解的类-->
    <mapper class="com.lagou.mapper.UserMapper"></mapper>
</mappers>
```

或者指定扫描包含映射关系的接口所在的包也可以

```
<mappers>
    <!--扫描使用注解的类所在的包-->
    <package name="com.lagou.mapper"></package>
</mappers>
```

第二种用的比较多，可以扫描到该文件夹下的所有注解。不仅仅可以用在扫描mapper注解

mapper加载四种方式：

1.package name="映射文件所在包名"

必须保证接口名（例如IUserDao）和xml名（IUserDao.xml）相同，还必须在同一个包中

```
<package name="com.mybatis.dao"/>
```

2.mapper resource=""

不用保证同接口同包同名

```
<mapper resource="com/mybatis/mappers/EmployeeMapper.xml"/>
```

3.mapper class="接口路径"

保证接口名（例如IUserDao）和xml名（IUserDao.xml）相同，还必须在同一个包中

```
<mapper class="com.mybatis.dao.EmployeeMapper"/>
```

4.mapper url="文件路径名" 不推荐

引用网路路径或者磁盘路径下的sql映射文件 file:///var/mappers/AuthorMapper.xml

```
<mapper url="file:E:/Study/myeclipse/_03_Test/src/cn/sdut/pojo/PersonMapper.xml"/>
```

问题：

Dao接口里的方法，参数不同时，方法能重载吗?

Dao接口里的方法，是不能重载的，因为是全限名+方法名的保存和寻找策略，重载方法时将导致矛盾。对于Mapper接口，Mybatis禁止方法重载（overLoad）。

什么是SOAP

简单对象访问协议是一种数据交换协议规范，是一种轻量的、简单的、基于XML的协议的规范。SOAP协议和HTTP协议一样，都是底层的通信协议，只是请求包的格式不同而已，SOAP包是XML格式的。SOAP的消息是基于xml并封装成了符合http协议，因此，它符合任何路由器、防火墙或代理服务器的要求。SOAP可以使用任何语言来完成，只要发送正确的SOAP请求即可，基于SOAP的服务可以在任何平台无需修改即可正常使用。

#{} 和${}的区别?

- #{}生成的sql，将#{user_ids}替换成?占位符，然后在执行时替换成实际传入的user_id值，并在两边加上单引号，以字符串方式处理

```
select user_id,user_name from t_user where user_id = #{user_id}
```

执行出的日志

```
10:27:20.247 [main] DEBUG william.mybatis.quickstart.mapper.UserMapper.selectById - ==>  Preparing: select id, user_name from t_user where id = ?
10:27:20.285 [main] DEBUG william.mybatis.quickstart.mapper.UserMapper.selectById - ==> Parameters: 1(Long)
```

- ${}生成的sql，即将传入的值直接拼接到SQL语句中，且不会自动加单引号

```
select user_id,user_name from t_user where user_id = ${user_id}
```

执行出的日志

```
10:27:20.247 [main] DEBUG william.mybatis.quickstart.mapper.UserMapper.selectById - ==>  Preparing: select id, user_name from t_user where id = 1
```

可以看到，参数是直接替换的，且没有单引号处理，这样就有SQL注入的风险。

但是在一些特殊情况下，使用${}是更适合的方式，如表名、orderby等。见下面这个例子：

```
select user_id,user_name from ${table_name} where user_id = ${user_id}
这里如果想要动态处理表名，就只能使用"${}"，因为如果使用"#{}"，就会在表名字段两边加上单引号，变成下面这样：
```

```
select user_id,user_name from 't_user' where user_id = ${user_id}
```

这样SQL语句就会报错。

占位符为什么可以防止sql注入?

以下为例：

String sql = "select * from administrator where adminname=?";
psm = con.prepareStatement(sql);

String s_name ="zhangsan' or '1'='1";
psm.setString(1, s_name);

如果zhangsan' or '1'='1直接拼接到sql会查询所有数据，而有占位符会转义：

```
转义后的sql为'zhangsan\' or \'1\'=\'1';这个时候是查不出来的。
```

sql (复用)

- 某些SQL片段如果需要复用，可以使用这个标签

```
<sql id="mysql">
    id,accin,accout,money
</sql>
```

- 在 ▼

```
<select id="">
    select <include refid="mysql"></include> from log
</select>
```

MySQL预编译：

MySQL执行预编译分为如三步：

第一步：执行预编译语句，例如：prepare myperson from 'select * from t_person where name=?'
第二步：设置变量，例如：set @name='Jim'
第三步：执行语句，例如：execute myperson using @name

如果需要再次执行myperson，那么就不再需要第一步，即不需要再编译语句了：

设置变量，例如：set @name='Tom'
执行语句，例如：execute myperson using @name

预编译的好处

1.1、预编译能避免SQL注入

预编译功能可以避免SQL注入，因为SQL已经编译完成，其结构已经固定，用户的输入只能当做参数传入进去，不能再破坏SQL的结果，无法造成曲解SQL原本意思的破坏。

1.2、预编译能提高SQL执行效率

预编译功能除了避免SQL注入，还能提高SQL执行效率。当客户发送一条SQL语句给服务器后，服务器首先需要校验SQL语句的语法格式是否正确，然后把SQL语句编译成可执行的函数，最后才是执行SQL语句。其中校验语法，和编译所花的时间可能比执行SQL语句花的时间还要多。

如果我们需要执行多次insert语句，但只是每次插入的值不同，MySQL服务器也是需要每次都去校验SQL语句的语法格式以及编译，这就浪费了太多的时间。如果**使用预编译功能，那么只对SQL语句进行一次语法校验和编译**，所以效率要高。

JDBC的预编译用法

相信每个人都应该了解JDBC中的PreparedStatement接口，它是用来实现SQL预编译的功能。其用法是这样的：

```
Class.forName("com.mysql.jdbc.Driver");
String url = "jdbc:mysql://127.0.0.1:3306/mybatis";
String user = "root";
String password = "123456";
//建立数据库连接
Connection conn = DriverManager.getConnection(url, user, password);

String sql = "insert into user(username, sex, address) values(?,?,?)";
PreparedStatement ps = conn.preparedStatement(sql);
ps.setString(1, "张三");  //为第一个问号赋值
ps.setInt(2, 2);    //为第二个问号赋值
ps.setString(3, "北京");    //为第三个问号赋值
ps.executeUpdate();
conn.close();
```

mybatis-redis源码：

源码分析：

RedisCache和大家普遍实现Mybatis的缓存方案大同小异，无非是实现Cache接口，并使用jedis操作缓存；不过该项目在设计细节上有一些区别；

```
public final class RedisCache implements Cache {
    public RedisCache(final String id) {
      if (id == null) {
            throw new IllegalArgumentException("Cache instances require an
ID");
      }
      this.id = id;
      RedisConfig redisConfig =

 RedisConfigurationBuilder.getInstance().parseConfiguration();
      pool = new JedisPool(redisConfig, redisConfig.getHost(),
      redisConfig.getPort(),
      redisConfig.getConnectionTimeout(),
      redisConfig.getSoTimeout(), redisConfig.getPassword(),
      redisConfig.getDatabase(), redisConfig.getClientName());
    }
```

RedisCache在mybatis启动的时候，由MyBatis的CacheBuilder创建，创建的方式很简单，就是调用RedisCache的带有String参数的构造方法，即RedisCache(String id)；而在RedisCache的构造方法中，调用了RedisConfigurationBuilder来创建RedisConfig对象，并使用RedisConfig来创建JedisPool。

RedisConfig类继承了JedisPoolConfig，并提供了host,port等属性的包装，简单看一下RedisConfig的属性： public class RedisConfig extends JedisPoolConfig {

```
private String host = Protocol.DEFAULT_HOST;
private int port = Protocol.DEFAULT_PORT;
private int connectionTimeout = Protocol.DEFAULT_TIMEOUT;
private int soTimeout = Protocol.DEFAULT_TIMEOUT;
private String password;
private int database = Protocol.DEFAULT_DATABASE;
private String clientName;
```

RedisConfig对象是由RedisConfigurationBuilder创建的，简单看下这个类的主要方法：

```
public RedisConfig parseConfiguration(ClassLoader classLoader) {
    Properties config = new Properties();
    InputStream input =
classLoader.getResourceAsStream(redisPropertiesFilename);
    if (input != null) {
    try {
      config.load(input);
    } catch (IOException e) {
      throw new RuntimeException(
        "An error occurred while reading classpath property '"
        + redisPropertiesFilename
        + "', see nested exceptions", e);
    } finally {
      try {
        input.close();
      } catch (IOException e) {
        // close quietly
      }
    }
  }

  RedisConfig jedisConfig = new RedisConfig();
  setConfigProperties(config, jedisConfig);
  return jedisConfig;
}
```

核心的方法就是parseConfiguration方法，该方法从classpath中读取一个redis.properties文件：

```
    host=localhost
    port=6379
    connectionTimeout=5000
    soTimeout=5000
    password=
    database=0
    clientName=
```

并将该配置文件中的内容设置到RedisConfig对象中，并返回；接下来，就是RedisCache使用RedisConfig类创建完成JedisPool；在RedisCache中实现了一个简单的模板方法，用来操作Redis：

```
private Object execute(RedisCallback callback) {
    Jedis jedis = pool.getResource();
    try {
      return callback.doWithRedis(jedis);
    } finally {
      jedis.close();
    }
}
```

模板接口为RedisCallback，这个接口中就只需要实现了一个doWithRedis方法而已：

```
public interface RedisCallback {
        Object doWithRedis(Jedis jedis);
    }
```

接下来看看Cache中最重要的两个方法：putObject和getObject，通过这两个方法来查看mybatis-redis储存数据的格式：

```
@Override
public void putObject(final Object key, final Object value) {
    execute(new RedisCallback() {
      @Override
      public Object doWithRedis(Jedis jedis) {
        jedis.hset(id.toString().getBytes(), key.toString().getBytes(),
                SerializeUtil.serialize(value));
        return null;
      }
    });
}

@Override
public Object getObject(final Object key) {
    return execute(new RedisCallback() {
      @Override
      public Object doWithRedis(Jedis jedis) {
        return SerializeUtil.unserialize(jedis.hget(id.toString().getBytes(),
                key.toString().getBytes()));
      }
    });
}
```

可以很清楚的看到，mybatis-redis在存储数据的时候，是使用的hash结构，把cache的id作为这个hash的key（cache的id在mybatis中就是mapper的namespace）；这个mapper中的查询缓存数据作为hash的field，需要缓存的内容直接使用SerializeUtil存储，SerializeUtil和其他的序列化类差不多，负责对象的序列化和反序列化；

mybatis插件源码：

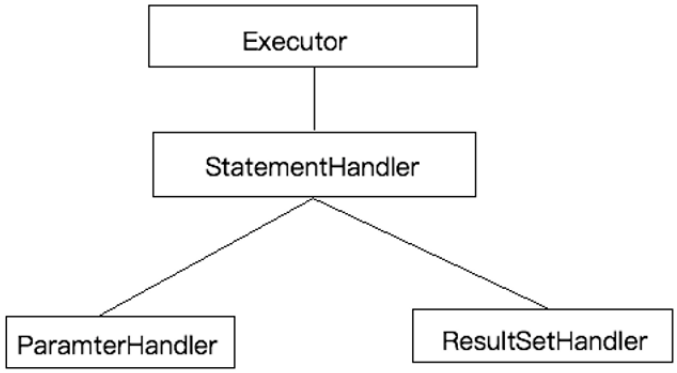
文章链接：https://www.cnblogs.com/summerday152/p/12777458.html

Mybatis插件介绍：

实际上就是对这四个对象里面的方法进行代理。

2. Mybatis插件介绍

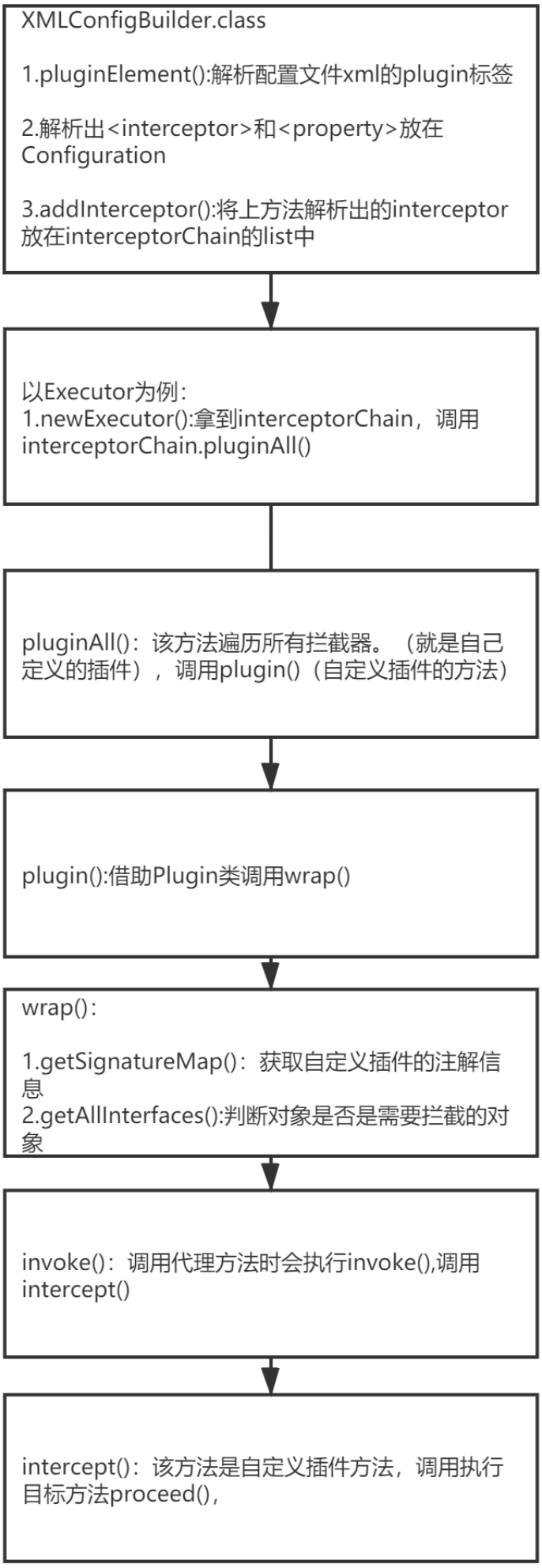
Mybatis作为一个应用广泛的优秀的ORM开源框架，这个框架具有强大的灵活性，在四大组件（Executor、StatementHandler、ParameterHandler、ResultSetHandler）处提供了简单易用的插件扩展机制。Mybatis对持久层的操作就是借助于四大核心对象。MyBatis支持用插件对四大核心对象进行拦截，对mybatis来说插件就是拦截器，用来增强核心对象的功能，增强功能本质上是借助于底层的动态代理实现的，换句话说，MyBatis中的四大对象都是代理对象



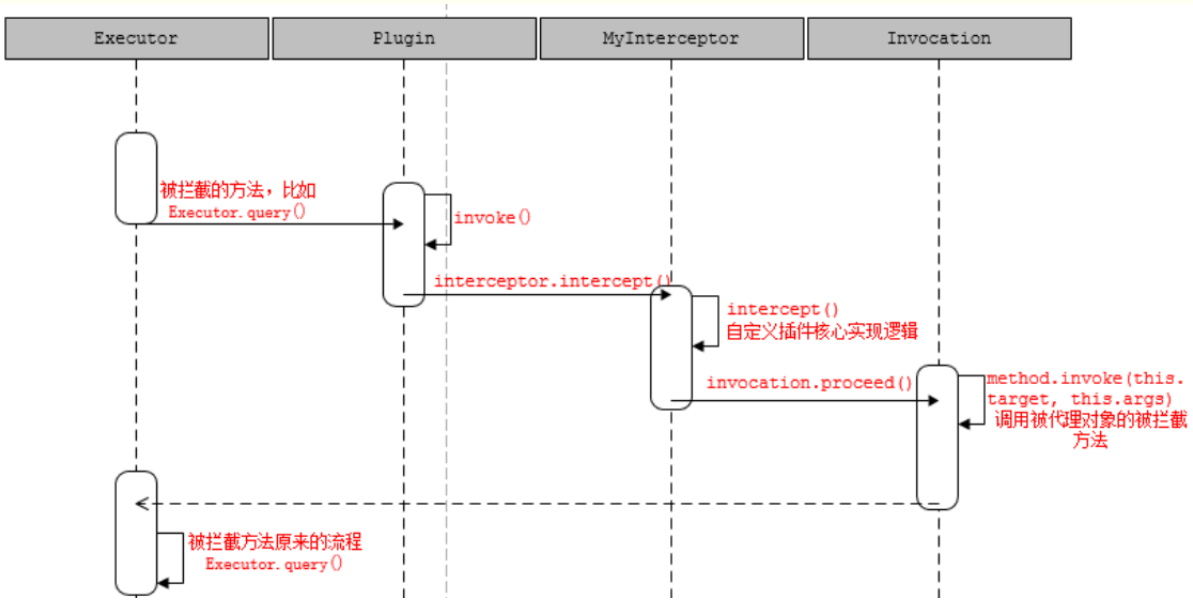
MyBatis 所允许拦截的方法如下：

- 执行器Executor（update、query、commit、rollback等方法）；
- SQL语法构建器StatementHandler（prepare、parameterize、batch、update、query等方法）；
- 参数处理器ParameterHandler（getParameterObject、setParameters方法）；
- 结果集处理器ResultSetHandler（handleResultSets、handleOutputParameters等方法）；

源码执行流程：


XMLConfigBuilder.class
1.pluginElement():解析配置文件xml的plugin标签
2.解析出<interceptor>和<property>放在Configuration
3.addInterceptor():将上方法解析出的interceptor放在interceptorChain的list中
以Executor为例：
1.newExecutor():拿到interceptorChain，调用interceptorChain.pluginAll()
pluginAll()：该方法遍历所有拦截器。（就是自己定义的插件），调用plugin()（自定义插件的方法）
plugin():借助Plugin类调用wrap()
wrap()：
1.getSignatureMap()：获取自定义插件的注解信息
2.getAllInterfaces():判断对象是否是需要拦截的对象
invoke()：调用代理方法时会执行invoke(),调用intercept()
intercept()：该方法是自定义插件方法，调用执行目标方法proceed()，

调用流程时序图：



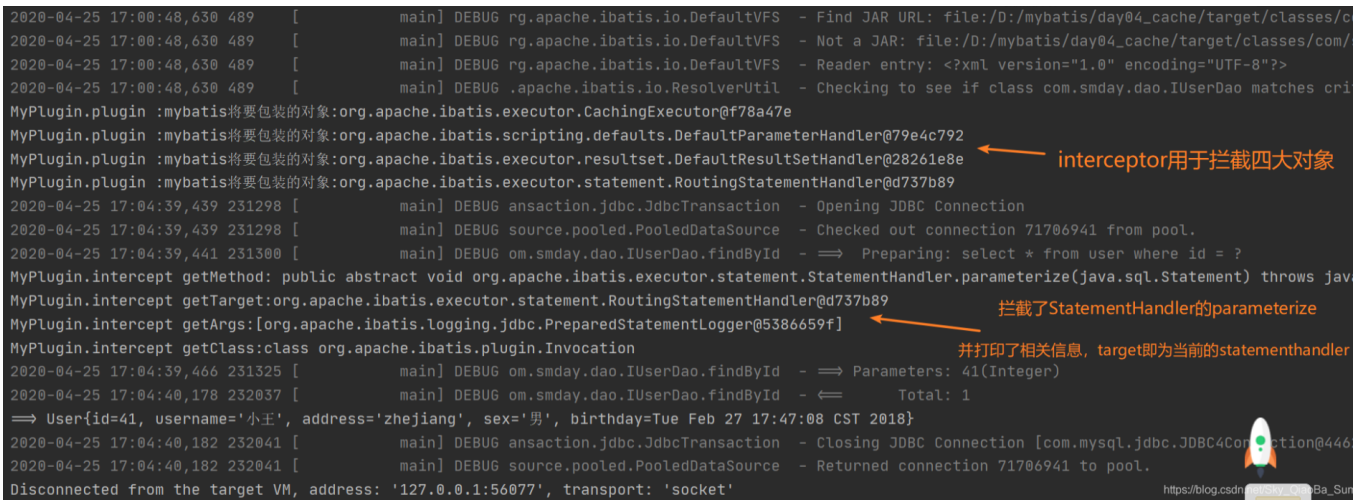
一、自定义插件流程

- 自定义插件，实现Interceptor接口。
- 实现intercept、plugin和setProperties方法。
- 使用@Intercepts注解完成插件签名。
- 在主配置文件注册插件。

```java
/**
 * 自定义插件
 * Intercepts:完成插件签名,告诉mybatis当前插件拦截哪个对象的哪个方法
 *
 * @author Summerday
 */
@Intercepts({
        @Signature(type = StatementHandler.class, method = "parameterize", args = Statement.class)
})
public class MyPlugin implements Interceptor {
    /**
     * 拦截目标方法执行
     *
     * @param invocation
     * @return
     * @throws Throwable
     */
    @Override
    public Object intercept(Invocation invocation) throws Throwable {
        //执行目标方法
        Object proceed = invocation.proceed();
        //返回执行后的返回值
        return proceed;
    }

    /**
     * 包装目标对象,为目标对象创建一个代理对象
     *
     * @param target
     * @return
     */
    @Override
    public Object plugin(Object target) {
        System.out.println("MyPlugin.plugin :mybatis将要包装的对象:"+target);
        //借助Plugin类的wrap方法使用当前拦截器包装目标对象
        Object wrap = Plugin.wrap(target, this);
        //返回为当前target创建的动态代理
        return wrap;
    }

    /**
     * 将插件注册时的properties属性设置进来
     *
     * @param properties
     */
    @Override
    public void setProperties(Properties properties) {
        System.out.println("插件配置的信息:" + properties);
    }
}
```

```xml
<!--注册插件-->
    <plugins>
        <plugin interceptor="com.smday.interceptor.MyPlugin">
            <property name="username" value="root"/>
            <property name="password" value="123456"/>
        </plugin>
    </plugins>
```

测试结果：



二、源码分析

执行插件逻辑Plugin实现了 InvocationHandler接口，因此它的invoke方法会拦截所有的方法调用。invoke方法会对所拦截的方法进行检测，以决定是否执行插件逻辑。该方法的逻辑如下：

```java
public Object invoke(Objectproxy, Methodmethod, Object[] args) throws Throwable {
       try {/**获取被拦截方法列表，比如：*signatureMap.get(Executor.class)，可能返回 [query, update,commit]*/
           Set<Method> methods = signatureMap.get(method.getDeclaringClass());
           //检测方法列表是否包含被拦截的方法
           if (methods != null && methods.contains(method)) {
               //执行插件逻辑
               return interceptor.intercept(new Invocation(target, method, args));
               //执行被拦截的方法
               return method.invoke(target, args);
           } catch(Exception e){
           }
       }
   }
```

invoke方法的代码比较少，逻辑不难理解。首先,invoke方法会检测被拦截方法是否配置在插件的@Signature注解中，若是，则执行插件逻辑，否则执行被拦截方法。插件逻辑封装在intercept中，该方法的参数类型为Invocationo Invocation主要用于存储目标类，方法以及方法参数列表。下面简单看一下该类的定义：

```java
public class Invocation {
    private final Object target;
    private final Method method;
    private final Object[] args;

    public Invocation(Object targetf Method method, Object[] args) {
        this.target = target;
        this.method = method;//省略部分代码
        public Object proceed () throws Invocation TargetException,        IllegalAccessException { //调用被拦截的方法
```

三、插件开发插件pagehelper

插件文档地址：https://github.com/pagehelper/Mybatis-PageHelper

这款插件使分页操作变得更加简便，来一个简单的测试如下：

1、引入相关依赖

```xml
    <dependency>
        <groupId>com.github.pagehelper</groupId>
        <artifactId>pagehelper</artifactId>
        <version>5.1.2</version>
    </dependency>
```

2、全局配置

```
    <!--注册插件-->
    <plugins>
        <plugin interceptor="com.github.pagehelper.PageInterceptor"></plugin>
    </plugins>
```

3、测试分页

```
    @Test
    public void testPlugin(){
        //查询第一页,每页3条记录
        PageHelper.startPage(1,3);
        List<User> all = userDao.findAll();
        for (User user : all) {
            System.out.println(user);
        }
    }
```

```
-04-25 18:38:02,331 579    [           main] DEBUG day.dao.IUserDao.findAll_COUNT  - ==> Parameters:
-04-25 18:38:02,344 592    [           main] DEBUG day.dao.IUserDao.findAll_COUNT  - <==      Total: 1
-04-25 18:38:02,351 599    [           main] DEBUG com.smday.dao.IUserDao.findAll  - ==>  Preparing: select * from user LIMIT ?
-04-25 18:38:02,352 600    [           main] DEBUG com.smday.dao.IUserDao.findAll  - ==> Parameters: 3(Integer)
-04-25 18:38:02,358 606    [           main] DEBUG com.smday.dao.IUserDao.findAll  - <==      Total: 3
```

四、插件总结

参考：《深入浅出MyBatis技术原理与实战》

- 插件生成地是层层代理对象的责任链模式，其中设计反射技术实现动态代理，难免会对性能产生一些影响。
- 插件的定义需要明确需要拦截的对象、拦截的方法、拦截的方法参数。
- 插件将会改变MyBatis的底层设计，使用时务必谨慎。

传统JDBC流程以及存在的问题：

```
public static void main(String[] args) {
        Connectionconnection = null;
        PreparedStatementpreparedStatement = null;
        ResultSetresultSet = null;
        try {
            //1.加载数据库驱动
            Class.forName("com.mysql.jdbc.Driver");
            //2.通过驱动管理类获取数据库连接
            connection = DriverManager.getConnection("jdbc:mysql://localhost:3306/mybatis?characterEncoding = utf - 8", " root", " root");
            //3.定义sql语句 ?表示占位符
            String sql = "select * from user where username = ?";
            //4.获取预处理statement
            preparedStatement = connection.prepareStatement(sql);
            //5.设置参数，第一个参数为sql语句中参数的序号
            preparedStatement.setString(1, "tom");
            //6.向数据库发出sql执行查询，查询出结果集
            resultSet = preparedStatement.executeQuery();
            //7.遍历查询结果集
            while (resultSet.next()) {
                int id = resultSet.getInt("id");
                String username = resultSet.getString("username");
                //封装User
                user.setId(id);
                user.setUsername(username);
            }
            System.out.println(user);
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            //8.释放资源
            if (resultSet != null) {
                try {
                    resultSet.close();
                } catch (SQLException e) {
                    e.printStackTrace();
                }
            }
            if (preparedStatement != null) {
                try {
                    preparedStatement.close();
                } catch (SQLException e) {
                    e.printStackTrace();
                }
            }
            if (connection != null) {
                try {
                    connection.close();
                } catch (SQLException e) {
                    e.printStackTrace();
                }
            }
        }
    }
```

JDBC问题总结：

1、 数据库连接创建、释放频繁造成系统资源浪费，从而影响系统性能。

2、 Sql语句在代码中硬编码，造成代码不易维护，实际应用中sql变化的可能较大，sql变动需要改变java代码。

3、 使用preparedStatement向占有位符号传参数存在硬编码，因为sql语句的where条件不一定，可能多也可能少，修改sql还要修改代码，系统不易维护。

4、对结果集解析存在硬编码（查询列名），sql变化导致解析代码变化，系统不易维护，如果能将数据库记录封装成pojo对象解析比较方便

1.2 问题解决思路

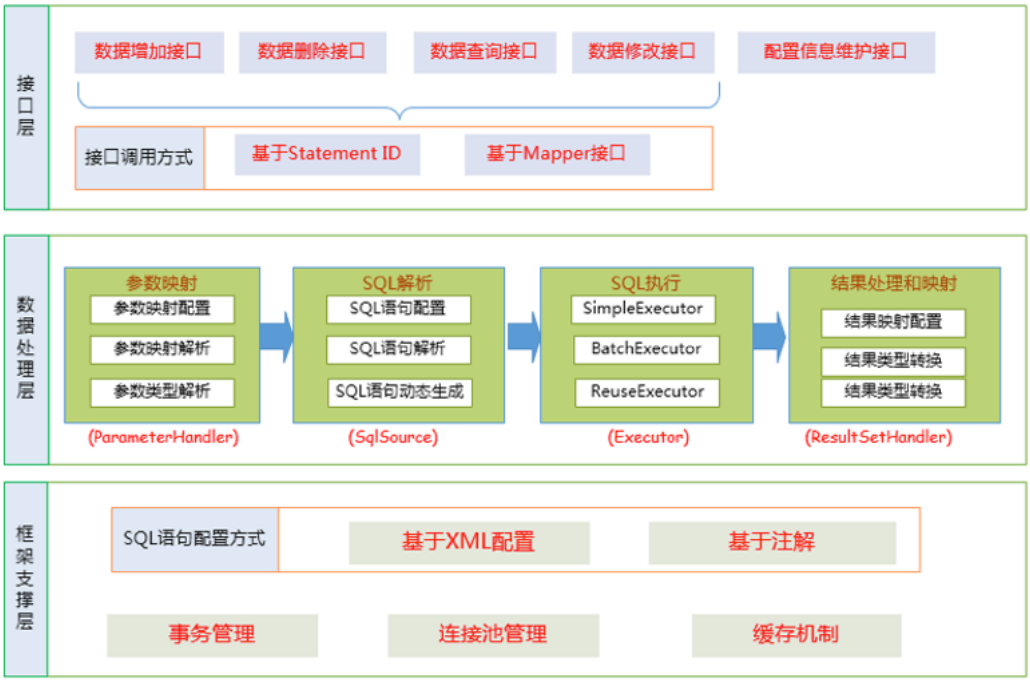
数据库频繁创建连接、释放资源：连接池

sql语句及参数硬编码：配置文件

手动解析封装返回结果集：反射、内省

mybatis总体架构、层次结构、总体流程：

架构设计：



我们把Mybatis的功能架构分为三层：

(1) API接口层：提供给外部使用的接口API，开发人员通过这些本地API来操纵数据库。接口层一接收到调用请求就会调用数据处理层来完成具体的数据处理。

MyBatis和数据库的交互有两种方式：

a.使用传统的MyBatis提供的API；

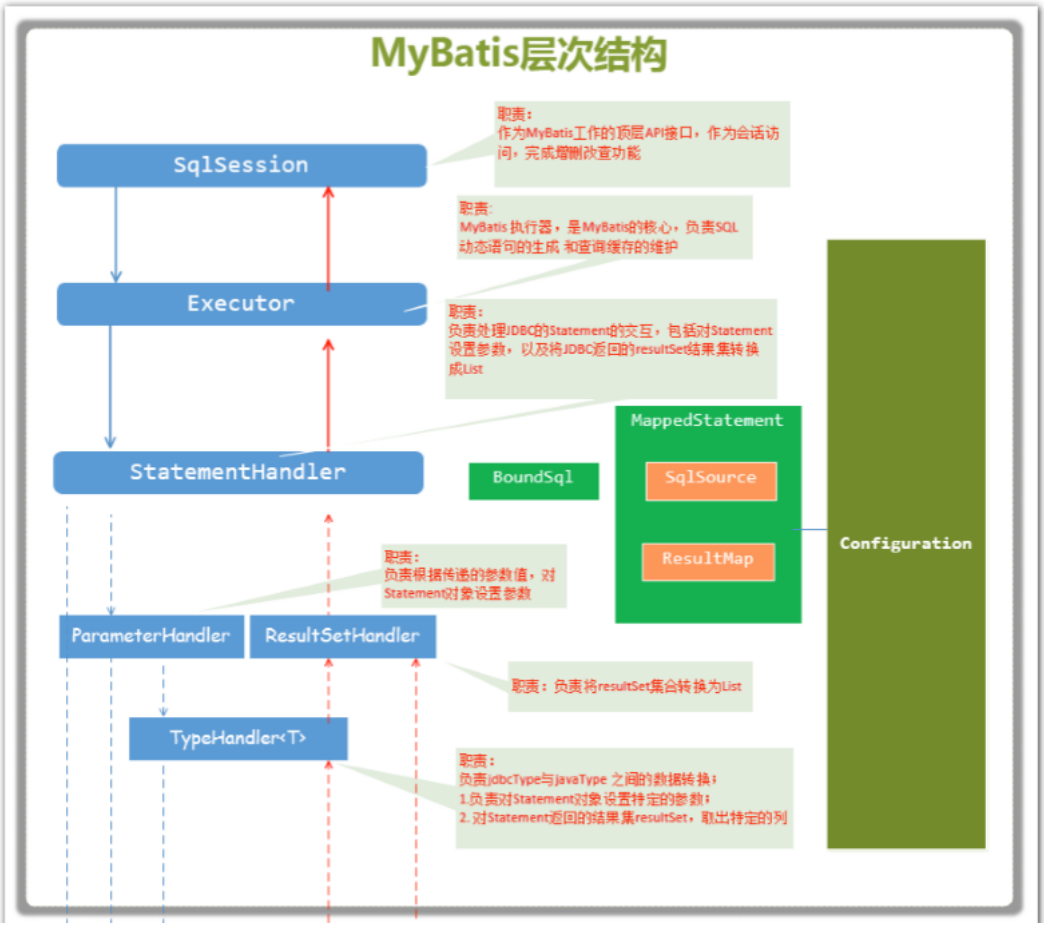
b. 使用Mapper代理的方式

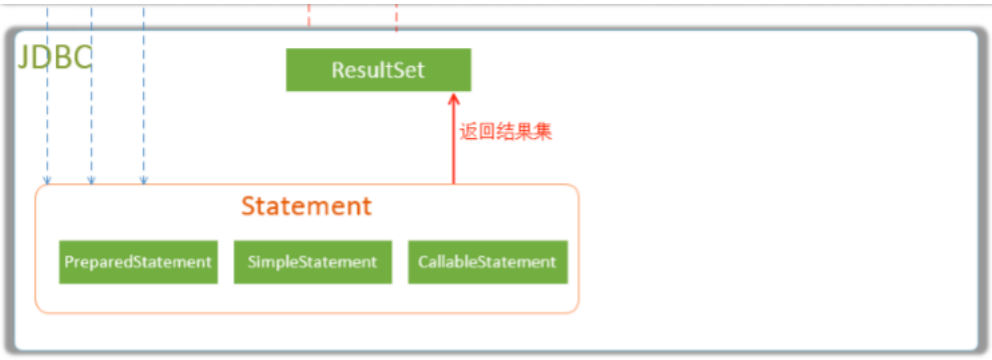
(2) 数据处理层：负责具体的SQL查找、SQL解析、SQL执行和执行结果映射处理等。它主要的目的是根据调用的请求完成一次数据库操作。

(3)基础支撑层：负责最基础的功能支撑，包括连接管理、事务管理、配置加载和缓存处理，这些都是共用的东西，将他们抽取出来作为最基础的组件。为上层的数据处理层提供最基础的支撑

主要构件机相互关系：

构件	描述
SqlSession	作为MyBatis工作的主要顶层API，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	MyBatis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	封装了JDBC Statement操作，负责对JDBC statement 的操作，如设置参数、将Statement结果集转换成List集合。
ParameterHandler	负责对用户传递的参数转换成JDBC Statement 所需要的参数，
ResultSetHandler	负责将JDBC返回的ResultSet结果集对象转换成List类型的集合；
TypeHandler	负责java数据类型和jdbc数据类型之间的映射和转换
MappedStatement	MappedStatement维护了一条<select|update|delete|insert>节点的封装
SqlSource	负责根据用户传递的parameterObject，动态地生成SQL语句，将信息封装到BoundSql对象中，并返回
BoundSql	表示动态生成的SQL语句以及相应的参数信息





9.3 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。

自定义mybatis

使用端：

1. sqlMapConfig.xml：存放数据库配置信息、存放mapper.xml的全路径

```xml
<configuration>
    <!--数据库配置信息-->
    <dataSource>
        <property name="driverClass" value="com.mysql.jdbc.Driver"></property>
        <property name="jdbcUrl" value="jdbc:mysql:///sys"></property>
        <property name="username" value="root"></property>
        <property name="password" value="admin123"></property>
    </dataSource>
    <!--存放mapper.xml的全路径-->
    <mapper resource="UserMapper.xml"></mapper>
</configuration>
```

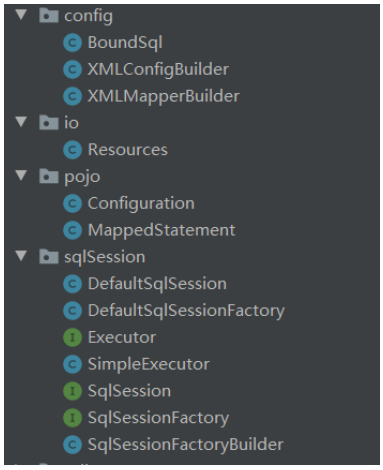
2. mapper.xml：存放sql配置信息

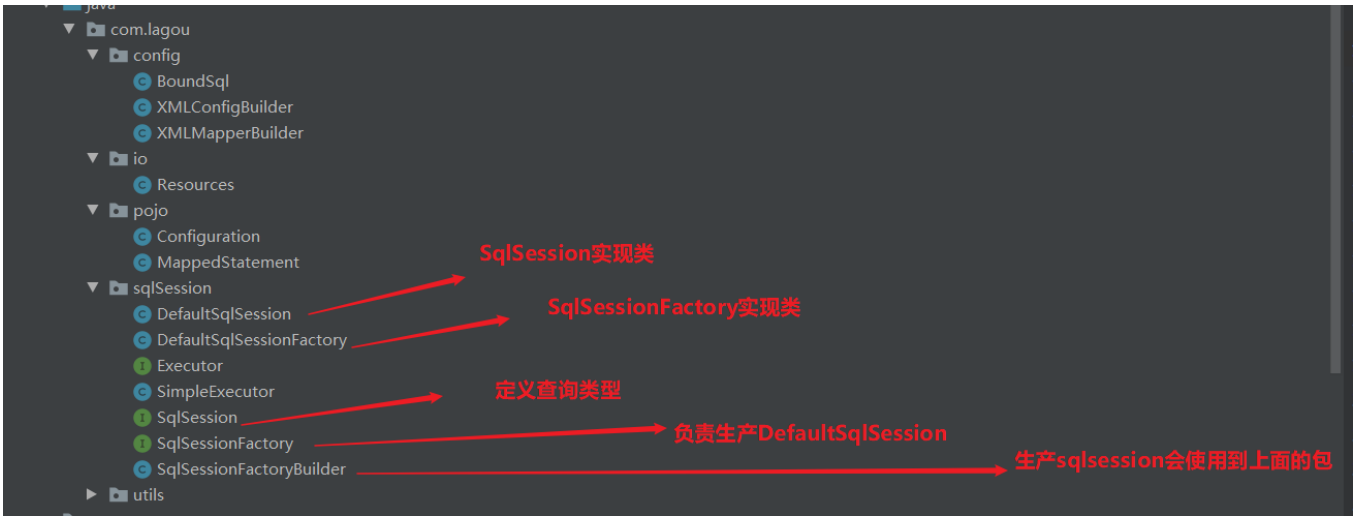
```xml
<mapper namespace="com.lagou.mapper.IOrderMapper">
    <select id="selectList">
        select * from product
    </select>
    <select id="selectOne">
        select * from product where id = ? and username = ?
    </select>
</mapper>
```

3. dao层：

```java
public interface IOrderMapper {
    @Select("select * from orders")
    public List<Order> selectList();
    @Select("select * from orders where uid = #{uid}")
    public List<Order> selectOne(Integer uid);
}
```

自定义持久层框架结构图：





测试类：

```java
InputStream resourceAsSteam = Resources.getResourceAsSteam( path: "sqlMapConfig.xml");
SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(resourceAsSteam);
SqlSession sqlSession = sqlSessionFactory.openSession();

//调用
User user = new User();
user.setId(1);
user.setUsername("232");
User user2 = sqlSession.selectOne( statementid: "User.selectOne", user);
System.out.println(user2);
/*  User user2 = sqlSession.selectOne("user.selectOne", user);

System.out.println(user2);*/

/* List<User> users = sqlSession.selectList("user.selectList");
for (User user1 : users) {
    System.out.println(user1);
}*/

IUserDao userDao = sqlSession.getMapper(IUserDao.class);

List<User> all = userDao.findAll();
for (User user1 : all) {
    System.out.println(user1);
```

流程总结：

1. **Resources：将配置文件转化为输入流：**

InputStream inputStream = Resources.getResourceAsStream("sqlMapConfig.xml")

```java
// 根据配置文件的路径，将配置文件加载成字节输入流，存储在内存中
    public static InputStream getResourceAsSteam(String path){
    InputStream resourceAsStream = Resources.class.getClassLoader().
getResourceAsStream(path);
        return  resourceAsStream;
 }
```

2. **SqlSessionFactoryBuilder：里面的build方法做三件事：**

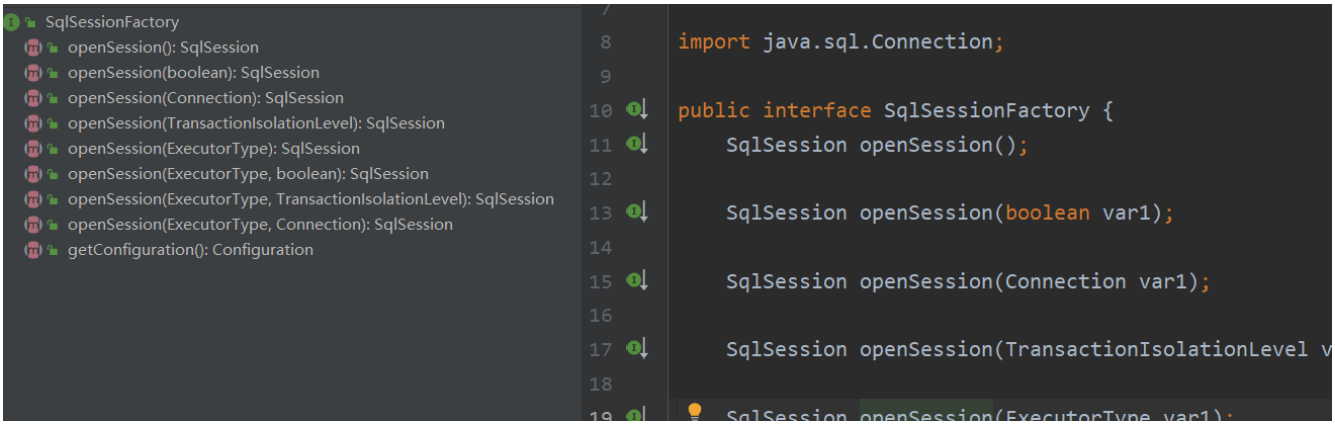
SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream)

> Configuration：包括数据库配置和MappedStatement（MappedStatement存储方式是map形式）。

```java
public class SqlSessionFactoryBuilder {

    public SqlSessionFactory build(InputStream in) throws DocumentException, PropertyVetoException {
        // 第一：使用dom4j解析配置文件，将解析出来的内容封装到Configuration中
        // 第二：使用dom4j解析mapper.xml，将解析出来的内容封装到mappedStatement，并以map形式放入Configuration（如下）：
        // String key = namespace+"."+id;
        //configuration.getMappedStatementMap().put(key,mappedStatement);
        XMLConfigBuilder xmlConfigBuilder = new XMLConfigBuilder();
        Configuration configuration = xmlConfigBuilder.parseConfig(in);

        // 第三：创建sqlSessionFactory对象：工厂类：生产sqlSession:会话对象
        DefaultSqlSessionFactory defaultSqlSessionFactory = new DefaultSqlSessionFactory(configuration);

        return defaultSqlSessionFactory;
    }
}
```

3. **openSession()介绍：**

SqlSessionFactory：负责生产DefaultSqlSession



SqlSession sqlSession = sqlSessionFactory.openSession()：

```
public SqlSession openSession() {
 return this.openSessionFromDataSource(this.configuration.getDefaultExecutorType(), (TransactionIsolationLevel)null, false);
    }
```

```
private SqlSession openSessionFromDataSource(ExecutorType execType, TransactionIsolationLevel level, boolean autoCommit) {
        Transaction tx = null;

        DefaultSqlSession var8;
        try {
            Environment environment = this.configuration.getEnvironment();
            TransactionFactory transactionFactory = this.getTransactionFactoryFromEnvironment(environment);
            tx = transactionFactory.newTransaction(environment.getDataSource(), level, autoCommit);
            Executor executor = this.configuration.newExecutor(tx, execType);
            //主要看这里！！！
            //创建SqlSession实现类DefaultSqlSession的实例
            var8 = new DefaultSqlSession(this.configuration, executor, autoCommit);
        } catch (Exception var12) {
            this.closeTransaction(tx);
            throw ExceptionFactory.wrapException("Error opening session.  Cause: " + var12, var12);
        } finally {
            ErrorContext.instance().reset();
        }
        return var8;
    }
```

4. **使用动态代理getMapper()，通过dao包路径找到mapper.xml文件对应sql以及SqlSession对应的增删改查类型（类型如sqlsession接口，跟据sql标签以及dao包的返回类型确定）：**

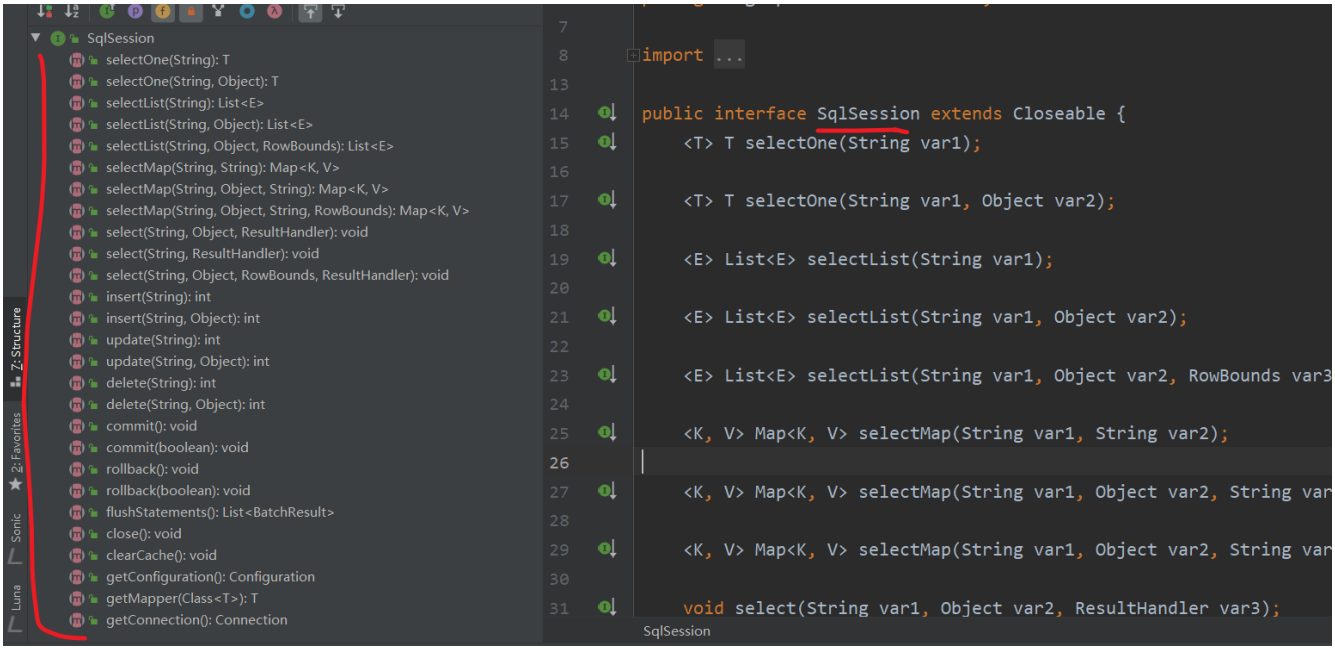
IUserDao userDao = sqlSession.getMapper(IUserDao.class)

```
 @Override
    public <T> T getMapper(Class<?> mapperClass) {
        // 使用JDK动态代理来为Dao接口生成代理对象，并返回
        Object proxyInstance = Proxy.newProxyInstance(DefaultSqlSession.class.getClassLoader(), new Class[]{mapperClass}, new InvocationHandler()
{
            @Override
            public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
                // 底层都还是去执行JDBC代码
                // 根据不同情况，来调用selctList或者selectOne
                // 准备参数 1：statmentid :sql语句的唯一标识：namespace.id= 接口全限定名.方法名
                // 方法名：findAll
                String methodName = method.getName();
                String className = method.getDeclaringClass().getName();
                String statementId = className+"."+methodName;
                // 准备参数2：params:args
                // 获取被调用方法的返回值类型
                Type genericReturnType = method.getGenericReturnType();
                // 判断是否进行了泛型类型参数化
                if(genericReturnType instanceof ParameterizedType){
                    List<Object> objects = selectList(statementId, args);
                    return objects;
                }
                return selectOne(statementId,args);
            }
        });
        return (T) proxyInstance;
    }
```

SqlSession接口介绍：

```
public interface SqlSession {
    //查询所有
    public <E> List<E> selectList(String statementid, Object... params) throws Exception;
    //根据条件查询单个
    public <T> T selectOne(String statementid, Object... params) throws Exception;
    //为Dao接口生成代理实现类
    public <T> T getMapper(Class<?> mapperClass);
}
```

源码对应的SqlSession接口，通过getMapper方式执行下面的方法：



5. **通过上面的动态代理执行sql：**

List users= userDao.findAll()

```
Executor：执行sql。
SimpleExecutor：Executor的实现类

   public class SimpleExecutor implements Executor {
       @Override                                                                    //user
       public <E> List<E> query(Configuration configuration, MappedStatement mappedStatement, Object... params) throws Exception {
           // 1. 注册驱动，获取连接
           Connection connection = configuration.getDataSource().getConnection();

           // 2. 获取sql语句 : select * from user where id = #{id} and username = #{username}
           //转换sql语句：  select * from user where id = ? and username = ? , 转换的过程中，还需要对#{}里面的值进行解析存储
           String sql = mappedStatement.getSql();
           BoundSql boundSql = getBoundSql(sql);

           // 3.获取预处理对象：preparedStatement
           PreparedStatement preparedStatement = connection.prepareStatement(boundSql.getSqlText());

           // 4. 设置参数
           //获取到了参数的全路径
           String paramterType = mappedStatement.getParamterType();
           Class<?> paramtertypeClass = getClassType(paramterType);

           List<ParameterMapping> parameterMappingList = boundSql.getParameterMappingList();
           for (int i = 0; i < parameterMappingList.size(); i++) {
               ParameterMapping parameterMapping = parameterMappingList.get(i);
               String content = parameterMapping.getContent();
               Field declaredField = paramtertypeClass.getDeclaredField(content);
               declaredField.setAccessible(true);
               Object o = declaredField.get(params[0]);

               preparedStatement.setObject(i + 1, o);

           }

           // 5. 执行sql
           ResultSet resultSet = preparedStatement.executeQuery();
           String resultType = mappedStatement.getResultType();
           Class<?> resultTypeClass = getClassType(resultType);

           ArrayList<Object> objects = new ArrayList<>();

           // 6. 封装返回结果集
           while (resultSet.next()) {
               Object o = resultTypeClass.newInstance();
               //元数据
               ResultSetMetaData metaData = resultSet.getMetaData();
               for (int i = 1; i <= metaData.getColumnCount(); i++) {
                   // 字段名
                   String columnName = metaData.getColumnName(i);
                   // 字段的值
                   Object value = resultSet.getObject(columnName);
                   //使用反射或者内省，根据数据库表和实体的对应关系，完成封装
                   PropertyDescriptor propertyDescriptor = new PropertyDescriptor(columnName, resultTypeClass);
                   Method writeMethod = propertyDescriptor.getWriteMethod();
                   writeMethod.invoke(o, value);
               }
               objects.add(o);
           }
           return (List<E>) objects;
       }
       private Class<?> getClassType(String paramterType) throws ClassNotFoundException {
           if (paramterType != null) {
               Class<?> aClass = Class.forName(paramterType);
```

```
                return aClass;
            }
            return null;
        }

        /**
         * 完成对#{}的解析工作：1.将#{}使用？进行代替，2.解析出#{}里面的值进行存储
         *
         * @param sql
         * @return
         */
        private BoundSql getBoundSql(String sql) {
            //标记处理类：配置标记解析器来完成对占位符的解析处理工作
            ParameterMappingTokenHandler parameterMappingTokenHandler = new ParameterMappingTokenHandler();
            GenericTokenParser genericTokenParser = new GenericTokenParser("#{", "}", parameterMappingTokenHandler);
            //解析出来的sql
            String parseSql = genericTokenParser.parse(sql);
            //#{}里面解析出来的参数名称
            List<ParameterMapping> parameterMappings = parameterMappingTokenHandler.getParameterMappings();

            BoundSql boundSql = new BoundSql(parseSql, parameterMappings);
            return boundSql;
        }
    }
```

数据库连接池默认连接数和配置参数?

```
acquireIncrement：声明当连接池中连接耗尽时再一次新生成多少个连接，默认为3个
initialPoolSize：当连接池启动时，初始化连接的个数，必须在minPoolSize~maxPoolSize之间，默认为3
minPoolSize：任何时间连接池中保存的最小连接数，默认3
maxPoolSize：在任何时间连接池中所能拥有的最大连接数，默认15
maxIdleTime：超过多长时间连接自动销毁，默认为0，即永远不会自动销毁
```

mybatis的坑?

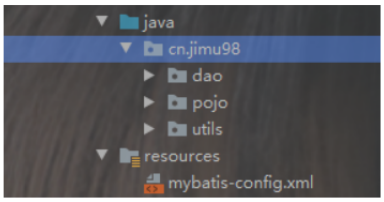
1. 在使用resultMap的时候，要把ID写在第一行，否则的话，就会报错。
2. resultType为Integer没有判断非空：然后在业务代码中很自然的用一个变量int去接当前这个方法的返回，如果按照你传入的条件在数据库中没有找到相关的值，此时selectOne方法的返回值会是一个null，当你使用Java的自动拆箱机制的时候会报出一个无情的NPE。

   原因：Java在自动拆箱的时候会调用Integer类中的intValue方法，如果当前对象为null，则抛出NPE，于是需要改为以下：

   ```
   int selectOne(Object obj);
   改为（注意resultType也要是Integer）：
   Integer selectOne(Object obj);
   ```

3. 包折叠导致xml文件找不到

我的包结构是这样的



之后把他改成这样的 就可以了