day03

内容复习

Mybatis整体重点

1. 配置文件中settings标签，修改mybatis运行行为，日志，驼峰映射
2. 映射文件中的resultMap(id,result,association,collection)
3. 映射文件中的select insert update delete （parameterType="实体类"，map，简单类型）
4. 映射文件中#{} 的写法
5. 映射文件中的if,where,set foreach(collection,open,close,item,seperator)
6. 支持插件扩展

1. 代理模式

实用编程基础的书籍

敏捷软件开发，原则，模式，实践

<UML模型设计>

微服务架构

spring响应式编程

java高并发编程

使用场景：在不改变源码的情况下，对一个已经存在的对象的方法进行扩展（做日常项目用不上，架构组/中间件开发/代理模式，反射，观察者模式）

参与角色：被代理对象(已经存在的对象)，代理对象(新创建的对象)，调用者(调用代理对象)----

- 静态代理：缺点：需要手动派生代理类，只对需要扩展的方法进行特殊操作
- **动态代理**

  - 动态代理实现方式：:Proxy,cglib组件
  - Proxy工具类要求被代理对象必须有接口.因为$Prxoy0代理类已经使用了extends关键字;
  - $Proxy0 类中持有的是invocationHandle对象
- Proxy工具类生成的代理类结构

```java
public class $Proxy0 extends Proxy implements Connection{

  protected InvocationHandler h;
  private Method m38;

  static{

 m38=Class.forName("java.sql.Connection").getMethod("prepareStatement", new Class[] { Class.forName("java.lang.String") });
  }

  public $Proxy0(InvocationHandler paramInvocationHandler)
  {
    this.h=h;
  }

  public final PreparedStatement prepareStatement(String sql) throws Exception
  {
      return (PreparedStatement)this.h.invoke(this, m38, new Object[]{sql});
  }
}
```

动态代理的操作流程分析

- 获取类加载器对象 ClassLoader l = souce.getClass.getClassLoader()
- 获取需要（被代理对象所实现的接口）的对象 本例是：class[] interfaces = new class[] {Connection.class}
- 获取InvocationHandle 对象，该对象中定义了一个匿名内部类

```
//Proxy.newProxyInstance()创建代理类$Proxy0,并实例化对象的过程.
ClassLoader l = source.getClass().getClassLoader();
Class[] interfaces=new Class[]{Connection.class};
//回调处理器
/**
* proxy: 代理对象,该对象绝对不能调用,只能传递引用.
* method: 正在调用的方法Method对象.
* args: 正在调用的方法的实参
*/
InvocationHandler h = new InvocationHandler() {
  @Override
  public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
    String name = method.getName();
    if(name.equals("close")){
      System.out.println("不在关闭,放回连接池");
    }else{
      Object result= method.invoke(source, args);
      return result;
    }
    return null;
  }
};
```

- 利用Proxy中的newProxyInstance方法创建一个代理对象。`Connection o = (Connection) Proxy.newProxyInstance(l, interfaces, h);`

动态代理的原理分析

- 执行newProxyInstance()方法时会创建一个$Proxy0的类，该类继承了Proxy,实现了数组interfaces中的所有接口
- $Proxy0中的构造器是一个以InvocationHandler对象为参数的有参构造，该构造的有参构造又调用了父类Proxy中的有参构造

```
public class $Proxy0 extends Proxy implements Connection{

  protected InvocationHandler h;// 成员变量
  private Method m38; // 方法对象
```

```
   static{
     m38=Class.forName("java.sql.Connection").getMethod("prepareStatement", new Class[] { Class.forName("java.lang.String") });
   }
   public $Proxy0(InvocationHandler paramInvocationHandler)
  {
    super(paramInvocationHandler)
  }
   public final PreparedStatement prepareStatement(String sql) throws Exception
  {
      return (PreparedStatement)this.h.invoke(this, m38, new Object[]{sql});
  }
}
```

- Proxy中的有参构造

```
protected Proxy(InvocationHandle h){
  Ojbects.requireNonNull(h);
  this.h = h; // this指的是调用者，这里的调用者是指子类的对象，即：$Proxy0 将 newProxyInstance() 方法传入的h对象赋值给成员变量h，即：$Proxy0对象持有了一个InvocationHandler对象
}
```

- 此时$Proxy0初始化完毕，建立了一个 connection对象proxy
- 当o去执行相关方法时，例如：proxy.parepareStatement("****")时，会执行Proxy0中重写的parepareStatement方法，该方法中的this--------->代理对象proxy, proxy.h------->上述的成员属性h-------->执行newProxyInstance()方法时所传的参数h，invoke--------->我们在new InvocationHandler()对象时所定义的匿名内部类中 我们重写的invoke方法。

```
public final PreparedStatement prepareStatement(String sql) throws Exception
  {
      return (PreparedStatement)this.h.invoke(this, m38, new Object[]{sql});
  }
```

- 上一步骤中执行的Invoke(this, m38, new Object[]{sql})方法中，this------>proxy对象，m38-------->

  `Class.forName("java.sql.Connection").getMethod("prepareStatement", new Class[] { Class.forName("java.lang.String") })` new Object[]{sql}------------>调用者所传递的实参

- 执行我们自定义的invoke时，可以通过method.getName() 获取要执行的方法的方法名，若满足一定条件(name.equals("close"))，则我们可以自定义其中的逻辑功能。若不满足，则默认调用 被代理对象中的初始方法

```
InvocationHandler h = new InvocationHandler() {
  @Override
  public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
    String name = method.getName();
    if(name.equals("close")){
      System.out.println("不在关闭,放回连接池");
    }else{
      // source 为被代理对象
      Object result= method.invoke(source, args);
      // 将结果返回
      return result;
    }
    return null;
  }
};
```

2.Configuration对象的构建过程

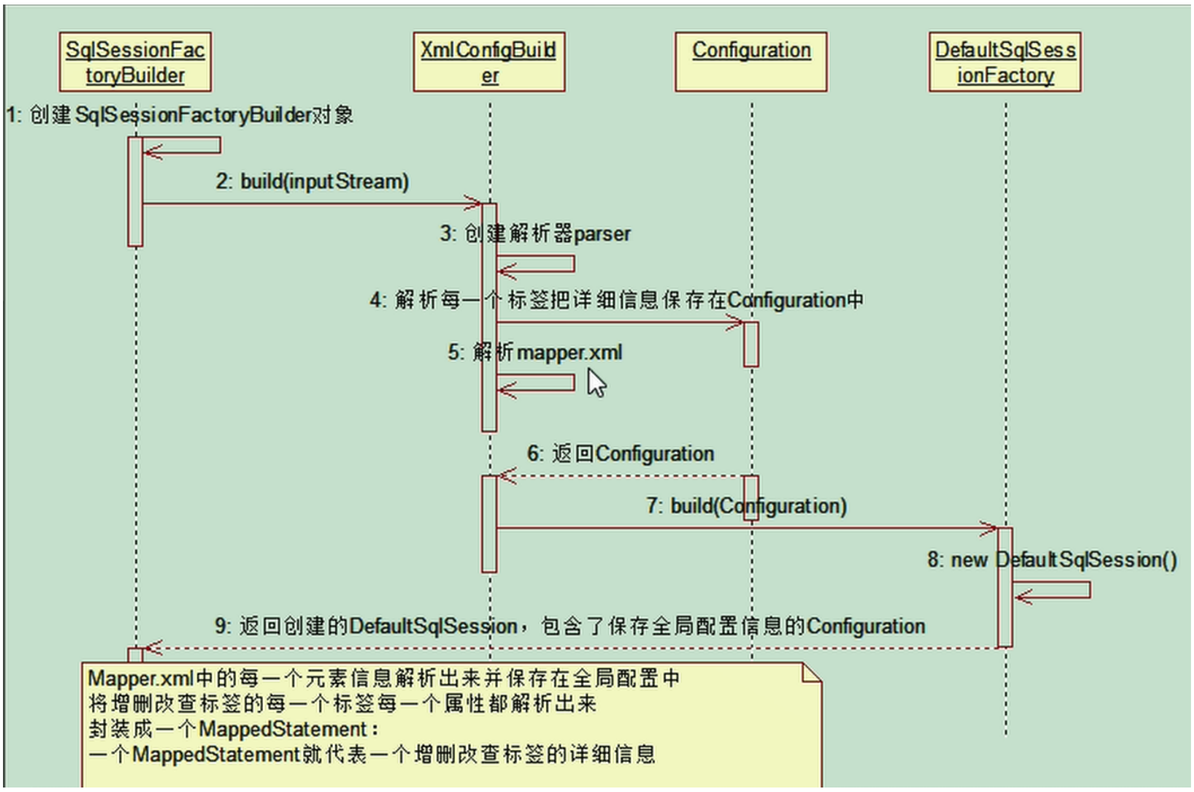
了解配置文件与映射文件的加载过程

- 获取sqlSessionFactory对象：解析每一个信息保存在Configuration中，返回包含了Configuration的Defaults 注意：MappedStatement：代表一个增删改查的详细信息
- 获取sqlSession对象：返回一个DefaultSQLSession对象，包含了Executor 和Configuration;
- 获取代理对象getMapper（MapperProxy）getMapper,使用MapperProxyFactory创建了一个MapperProxy的代理对象，代理对象包含了DefautltSqlSession（Executor）
- 执行增删改查方法

configuration对象的构建的整个流程

根据配置文件创建SqlSessionFactory对象----------->Configuration对象中封装了所有的文件详细信息

总结：把配置文件的信息解析并保存在Configuration对象中，返回DefaultSqlSession对象。


SqlSessionFactoryBuilder
XmlConfigBuilder
Configuration
DefaultSqlSessionFactory
1: 创建 SqlSessionFactoryBuilder对象
2: build(inputStream)
3: 创建解析器parser
4: 解析每一个标签把详细信息保存在Configuration中
5: 解析mapper.xml
6: 返回Configuration
7: build(Configuration)
8: new DefaultSqlSession()
9: 返回创建的DefaultSqlSession，包含了保存全局配置信息的Configuration

Mapper.xml中的每一个元素信息解析出来并保存在全局配置中
将增删改查标签的每一个标签每一个属性都解析出来
封装成一个MappedStatement：
一个MappedStatement就代表一个增删改查标签的详细信息

从SqlSessionFactoryBuilder对象的build(in)方法中获取一个SqlSessionFactory对象

- 通过SqlSessionFactoryBuilder()类的build(in)方法创建一个SqlSessionFactory对象

```java
factory = new SqlSessionFactoryBuilder().build(in);
```

- builder方法

```java
public SqlSessionFactory build(InputStream inputStream, String environment, Properties properties) {
  SqlSessionFactory var5;
  try {
    XMLConfigBuilder parser = new XMLConfigBuilder(inputStream, environment, properties);
    var5 = this.build(parser.parse());
  } catch (Exception var14) {
    throw ExceptionFactory.wrapException("Error building SqlSession.", var14);
  } finally {
    ErrorContext.instance().reset();
    try {
      inputStream.close();
    } catch (IOException var13) {
      ;
    }
  }
  return var5;
}
```

- 新建XMLConfigBuilder对象时调用两个构造器， 在一个构造器中对Configuration进行了初始化操作

```
public XMLConfigBuilder(InputStream inputStream, String environment, Properties props) {
  this(new XPathParser(inputStream, true, props, new XMLMapperEntityResolver()), environment, props);
}

private XMLConfigBuilder(XPathParser parser, String environment, Properties props) {
  super(new Configuration());
  this.localReflectorFactory = new DefaultReflectorFactory();
  ErrorContext.instance().resource("SQL Mapper Configuration");
  this.configuration.setVariables(props);
  this.parsed = false;
  this.environment = environment;
  this.parser = parser;
}
```

- XMLConfigBuilder对象中parse()方法为：配置文件构建者的解析方法，

```
public class XMLConfigBuilder extends BaseBuilder {
  // Parsed为判断标记，创建configuration对象是会将parsed置为true
  // 保证configuration对象的全局唯一性
  private boolean parsed;
  private XPathParser parser;  // xpath解析对象
  private String environment;
  private ReflectorFactory localReflectorFactory;
  //在父类BaseBuilder中的成员变量，且是final类型，即：地址不可变（全局唯一），  类型为protected,即：子类可以使用父类的该属性
  protected final Configuration configuration;
  public Configuration parse() {
    if (this.parsed) {
      throw new BuilderException("Each XMLConfigBuilder can only be used once.");
    } else {
      this.parsed = true;
      this.parseConfiguration(this.parser.evalNode("/configuration"));
      return this.configuration;
    }
  }
}
```

- XpathParser对象中的evalNode()方法，调用了XpathImpl实现类中的evaluate（ ）方法，可以获取到一个XNode root 对象

```
public class XPathImpl implements javax.xml.xpath.XPath {

  public Object evaluate(String expression, Object item, QName returnType)
    throws XPathExpressionException {
    if ( expression == null ) {
      String fmsg = XSLMessages.createXPATHMessage(
        XPATHErrorResources.ER_ARG_CANNOT_BE_NULL,
        new Object[] {"XPath expression"} );
      throw new NullPointerException ( fmsg );
    }
    if ( returnType == null ) {
      String fmsg = XSLMessages.createXPATHMessage(
        XPATHErrorResources.ER_ARG_CANNOT_BE_NULL,
        new Object[] {"returnType"} );
      throw new NullPointerException ( fmsg );
    }
    // Checking if requested returnType is supported. returnType need to
    // be defined in XPathConstants
    if ( !isSupported ( returnType ) ) {
      String fmsg = XSLMessages.createXPATHMessage(
        XPATHErrorResources.ER_UNSUPPORTED_RETURN_TYPE,
        new Object[] { returnType.toString() } );
      throw new IllegalArgumentException ( fmsg );
    }

    try {
      // 核心业务代码
      XObject resultObject = eval( expression, item );
      return getResultAsType( resultObject, returnType );
    } catch ( java.lang.NullPointerException npe ) {
      // If VariableResolver returns null Or if we get
      // NullPointerException at this stage for some other reason
      // then we have to reurn XPathException
      throw new XPathExpressionException ( npe );
    } catch ( javax.xml.transform.TransformerException te ) {
      Throwable nestedException = te.getException();
      if ( nestedException instanceof javax.xml.xpath.XPathFunctionException ) {
        throw (javax.xml.xpath.XPathFunctionException)nestedException;
      } else {
        // For any other exceptions we need to throw
        // XPathExpressionException ( as per spec )
        throw new XPathExpressionException ( te );
      }
    }
  }
}
```

- 调用解析配置文件的方法 parseConfiguration(XNode root) 对上一步得到的xnode对象进行解析

```
public class XMLConfigBuilder extends BaseBuilder{
  private boolean parsed;
  private XPathParser parser;
  private String environment;
  private ReflectorFactory localReflectorFactory;
  private void parseConfiguration(XNode root) {
    try {
      Properties settings = this.settingsAsPropertiess(root.evalNode("settings"));
      this.propertiesElement(root.evalNode("properties"));
      this.loadCustomVfs(settings);
      this.typeAliasesElement(root.evalNode("typeAliases"));
      this.pluginElement(root.evalNode("plugins"));
      this.objectFactoryElement(root.evalNode("objectFactory"));
      this.objectWrapperFactoryElement(root.evalNode("objectWrapperFactory"));
      this.reflectorFactoryElement(root.evalNode("reflectorFactory"));
      this.settingsElement(settings);
      this.environmentsElement(root.evalNode("environments"));
      this.databaseIdProviderElement(root.evalNode("databaseIdProvider"));
      this.typeHandlerElement(root.evalNode("typeHandlers"));
      this.mapperElement(root.evalNode("mappers"));
    } catch (Exception var3) {
      throw new BuilderException("Error parsing SQL Mapper Configuration. Cause: " + var3, var3);
    }
  }
}
```

- (以xml配置文件中mapper标签举例)通过调用mapperElement()方法，将mapper中的接口对象（Class<?> mapperInterface）放入configuration对象中

```
private void mapperElement(XNode parent) throws Exception {
  if (parent != null) {
    Iterator i$ = parent.getChildren().iterator();

  while(true) {
   while(i$.hasNext()) {
    XNode child = (XNode)i$.next();
    String resource;
    if ("package".equals(child.getName())) {
     resource = child.getStringAttribute("name");
     this.configuration.addMappers(resource);
    } else {
     resource = child.getStringAttribute("resource");
     String url = child.getStringAttribute("url");
     String mapperClass = child.getStringAttribute("class");
     XMLMapperBuilder mapperParser;
     InputStream inputStream;
     if (resource != null && url == null && mapperClass == null) {
      ErrorContext.instance().resource(resource);
      inputStream = Resources.getResourceAsStream(resource);
      mapperParser = new XMLMapperBuilder(inputStream, this.configuration, resource, this.configuration.getSqlFragments());
      mapperParser.parse();
     } else if (resource == null && url != null && mapperClass == null) {
      ErrorContext.instance().resource(url);
      inputStream = Resources.getUrlAsStream(url);
      mapperParser = new XMLMapperBuilder(inputStream, this.configuration, url, this.configuration.getSqlFragments());
      mapperParser.parse();
     } else {
      if (resource != null || url != null || mapperClass == null) {
       throw new BuilderException("A mapper element may only specify a url, resource or class, but not more than one.");
```

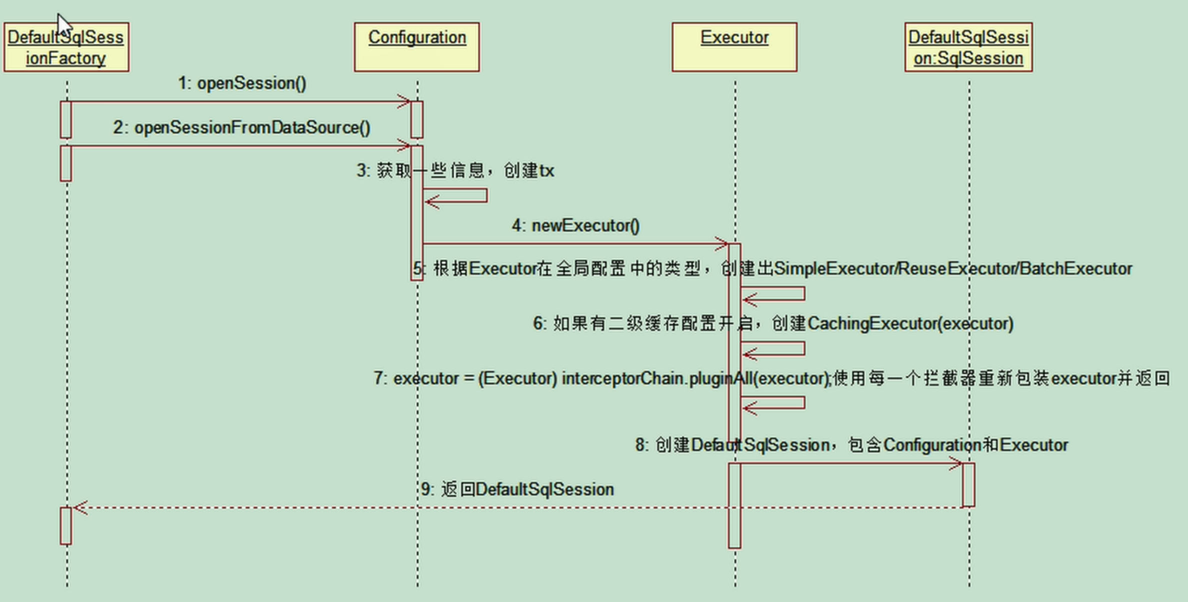
```
            }

            Class<?> mapperInterface = Resources.classForName(mapperClass);
                this.configuration.addMapper(mapperInterface);
          }
        }
      }
      return;
    }
  }
}
```

- 将装配完成的Configuration对象，利用this.build( ) 方法，装配为SqlSessionFactory对象

```
public SqlSessionFactory build(Configuration config) {
  return new DefaultSqlSessionFactory(config);
}
```

- 可以得到一个 SqlSessionFactory factory = new SqlSessionFactoryBuilder().build(in);对象

从SqlSessionFactory 对象得到一个SqlSession对象的实现类DefaultSqlSession对象，它包含了Executor和Configuration；Executor会在这一步被创建


DefaultSqlSessionFactory
Configuration
Executor
DefaultSqlSession:SqlSession
1: openSession()
2: openSessionFromDataSource()
3: 获取一些信息，创建tx
4: newExecutor()
5: 根据Executor在全局配置中的类型，创建出SimpleExecutor/ReuseExecutor/BatchExecutor
6: 如果有二级缓存配置开启，创建CachingExecutor(executor)
7: executor = (Executor) interceptorChain.pluginAll(executor);使用每一个拦截器重新包装executor并返回
8: 创建DefaultSqlSession，包含Configuration和Executor
9: 返回DefaultSqlSession

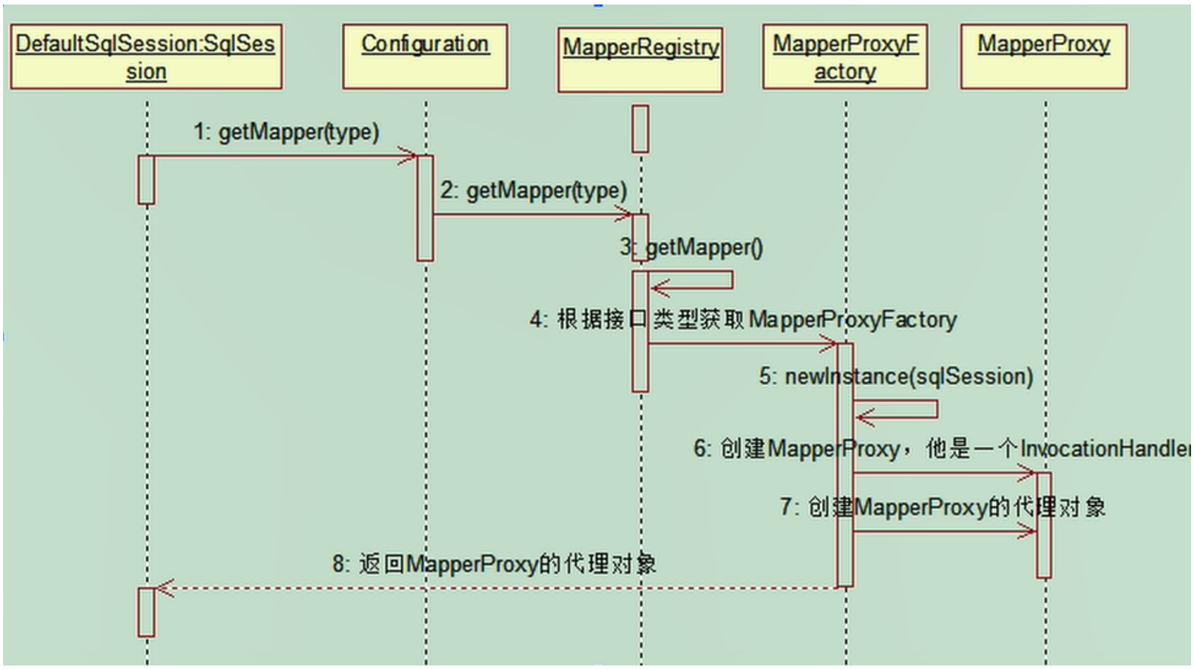
- 调用SqlSessionFactory(接口)对象的openSession()方法，SqlSessionFactory 是一个 接口，调用的是DefaultSqlSessionFactory()实现类

```java
public class DefaultSqlSessionFactory implements SqlSessionFactory {
  private final Configuration configuration;
  // 构造器，SqlSessionFactory对象中的configuration已经赋值了
  public DefaultSqlSessionFactory(Configuration configuration) {
    this.configuration = configuration;
  }

  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);
      // 建立一个DefaultSqlSession对象，将configuration传入
      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();
```

```java
        }
        return var8;
    }
}
```

- 在DefaultSqlSessionFactory对象中通过openSessionFromDataSource方法创建了一个DefaultSqlSession对象，并将其返回

从SqlSession(接口)对象中执行getMapper()方法


DefaultSqlSession:SqlSession
Configuration
MapperRegistry
MapperProxyFactory
MapperProxy
1: getMapper(type)
2: getMapper(type)
3: getMapper()
4: 根据接口类型获取MapperProxyFactory
5: newInstance(sqlSession)
6: 创建MapperProxy，他是一个InvocationHandle
7: 创建MapperProxy的代理对象
8: 返回MapperProxy的代理对象

Mapper=$Proxy4 代理对象中包含了 MapperProxy 和sqlSession

> SqlSession是一个接口，调用的是实现类(DefaultSqlSession)中的getMapper方法

```java
public class DefaultSqlSession implements SqlSession {
  private Configuration configuration;
  private Executor executor;
  private boolean autoCommit;
  private boolean dirty;
  private List<Cursor<?>> cursorList;
  // 上一步openSessionFromDataSource通过该构造生成了一个SqlSession对象
  public DefaultSqlSession(Configuration configuration, Executor executor, boolean autoCommit) {
    this.configuration = configuration;
    this.executor = executor;
    this.dirty = false;
    this.autoCommit = autoCommit;
  }

  public DefaultSqlSession(Configuration configuration, Executor executor) {
    this(configuration, executor, false);
  }
  public <T> T getMapper(Class<T> type) {
    return this.configuration.getMapper(type, this);
  }
}
```

- getMapper方法返回 的是configuration对象中的getMapper()方法的返回值，传递的参数 是 ：类对象class,和this(当前的sqlSession对象)

```java
public class Configuration {
  public <T> T getMapper(Class<T> type, SqlSession sqlSession) {
    return this.mapperRegistry.getMapper(type, sqlSession);
  }
}
```

- 调用的是mapperRegistry对象中getMapper方法

```java
public class MapperRegistry {
  private final Configuration config;
  private final Map<Class<?>, MapperProxyFactory<?>> knownMappers = new HashMap();

  public MapperRegistry(Configuration config) {
    this.config = config;
  }
  public <T> T getMapper(Class<T> type, SqlSession sqlSession) {
    MapperProxyFactory<T> mapperProxyFactory = (MapperProxyFactory)this.knownMappers.get(type);
    if (mapperProxyFactory == null) {
      throw new BindingException("Type " + type + " is not known to the MapperRegistry.");
    } else {
      try {
        return mapperProxyFactory.newInstance(sqlSession);
      } catch (Exception var5) {
        throw new BindingException("Error getting mapper instance. Cause: " + var5, var5);
      }
    }
  }
}
```

- MapperRegistry对象中的getMapper方法执行的是mapperProxyFactory对象中的newInstance方法

```
public class MapperProxyFactory<T> {
  public T newInstance(SqlSession sqlSession) {
  MapperProxy<T> mapperProxy = new MapperProxy(sqlSession, this.mapperInterface, this.methodCache);
  return this.newInstance(mapperProxy);
  }
}
```

- 创建了一个mapperProxy对象(mapperProxy)，MapperProxy对象，实现了InvocationHandler接口

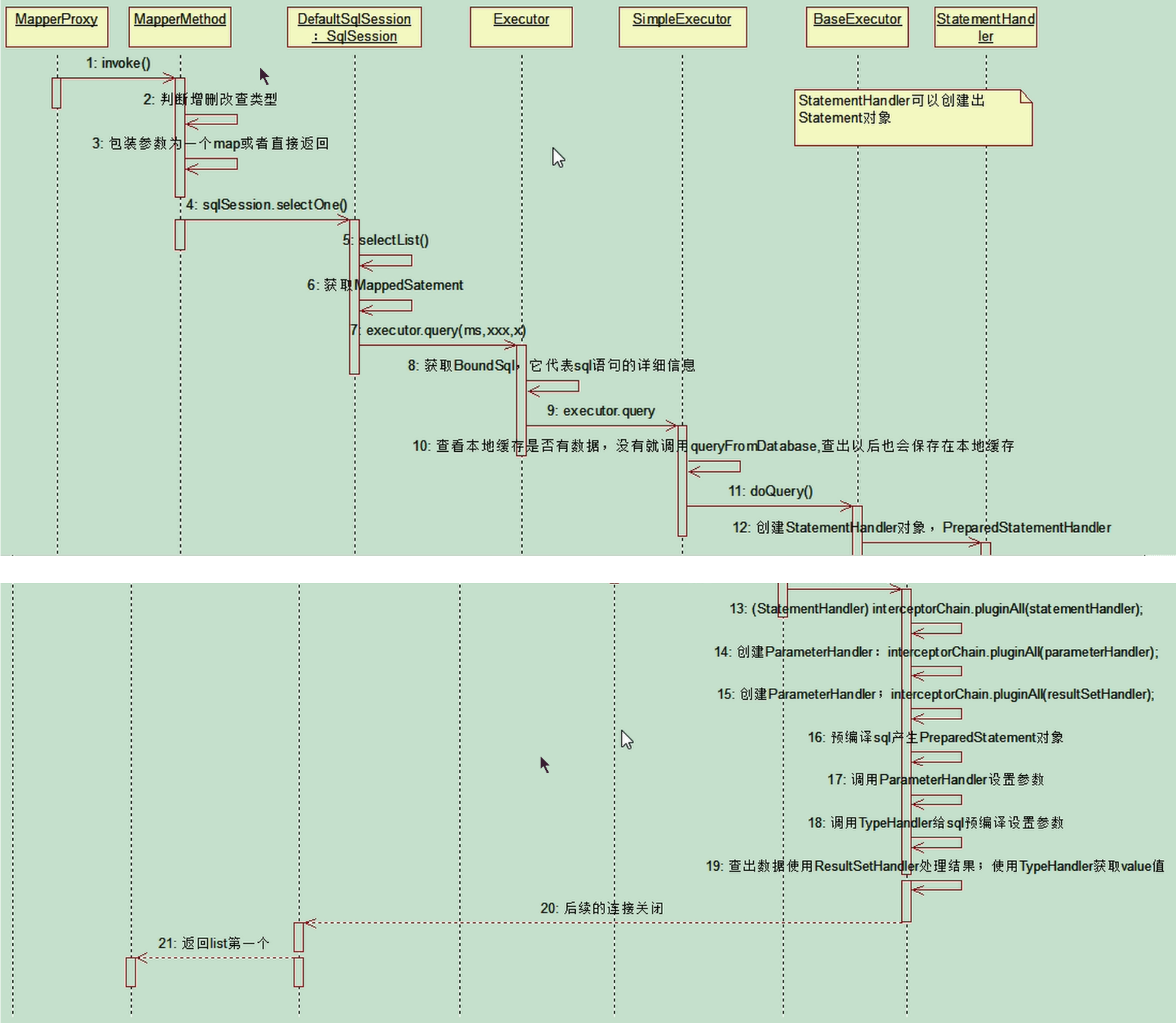
```
public class MapperProxy<T> implements InvocationHandler, Serializable {
  private static final long serialVersionUID = -6424540398559729838L;
  private final SqlSession sqlSession;
  private final Class<T> mapperInterface;
  private final Map<Method, MapperMethod> methodCache;

  public MapperProxy(SqlSession sqlSession, Class<T> mapperInterface, Map<Method, MapperMethod> methodCache) {
    this.sqlSession = sqlSession;
    this.mapperInterface = mapperInterface;
    this.methodCache = methodCache;
  }

  public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
    if (Object.class.equals(method.getDeclaringClass())) {
      try {
        return method.invoke(this, args);
      } catch (Throwable var5) {
        throw ExceptionUtil.unwrapThrowable(var5);
      }
    } else {
      MapperMethod mapperMethod = this.cachedMapperMethod(method);
      return mapperMethod.execute(this.sqlSession, args);
    }
  }

  private MapperMethod cachedMapperMethod(Method method) {
    MapperMethod mapperMethod = (MapperMethod)this.methodCache.get(method);
    if (mapperMethod == null) {
      mapperMethod = new MapperMethod(this.mapperInterface, method, this.sqlSession.getConfiguration());
      this.methodCache.put(method, mapperMethod);
    }

    return mapperMethod;
  }
}
```

- mapperProxyFactory对象中的newInstance方法执行了MapperProxy构造，创建了一个mapperProxy对象(mapperProxy)，将其作为参数，this.newInstance(mapperProxy);

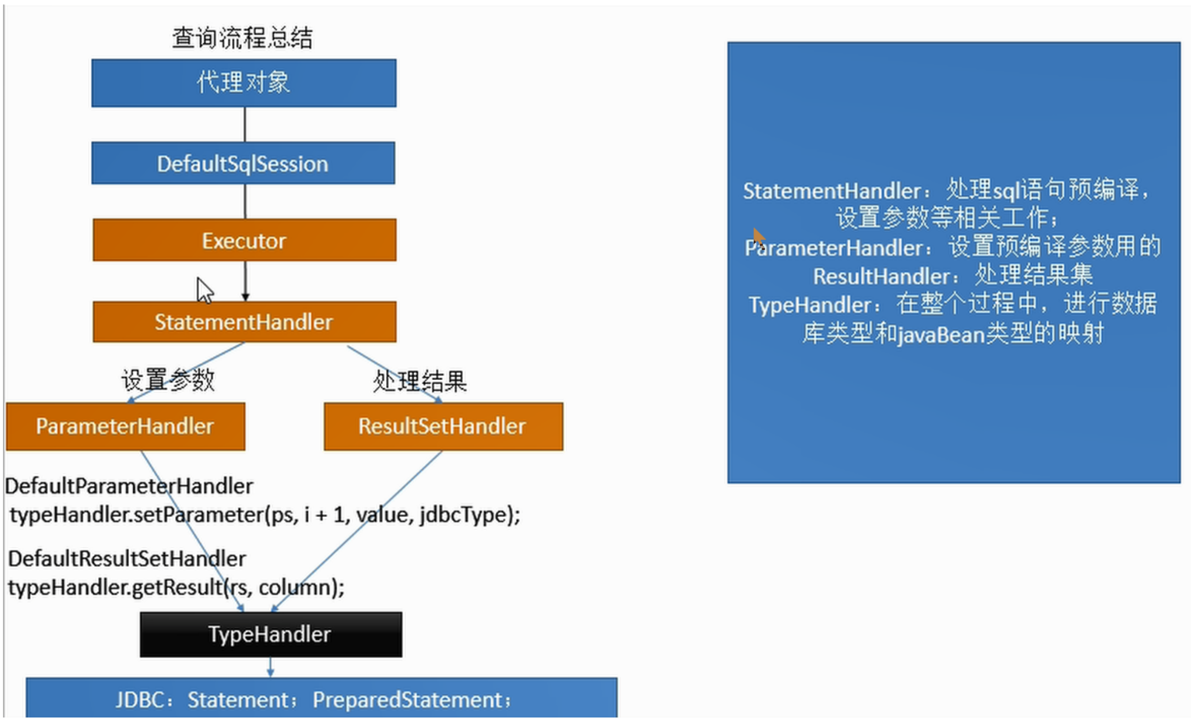
```java
public class MapperProxyFactory<T> {
  protected T newInstance(MapperProxy<T> mapperProxy) {
    return
Proxy.newProxyInstance(this.mapperInterface.getClassLoader(), new Class[]{this.mapperInterface}, mapperProxy);
  }
}
```

- 通过工具类Proxy中的newProxyInstance方法，建立了一个动态代理对象，并将其返回。

Mybatis(invoke)执行增删改查


MapperProxy
MapperMethod
DefaultSqlSession : SqlSession
Executor
SimpleExecutor
BaseExecutor
StatementHandler
StatementHandler可以创建出Statement对象
1: invoke()
2: 判断增删改查类型
3: 包装参数为一个map或者直接返回
4: sqlSession.selectOne()
5: selectList()
6: 获取MappedSatement
7: executor.query(ms,xxx,x)
8: 获取BoundSql，它代表sql语句的详细信息
9: executor.query
10: 查看本地缓存是否有数据，没有就调用queryFromDatabase,查出以后也会保存在本地缓存
11: doQuery()
12: 创建StatementHandler对象，PreparedStatementHandler
13: (StatementHandler) interceptorChain.pluginAll(statementHandler);
14: 创建ParameterHandler：interceptorChain.pluginAll(parameterHandler);
15: 创建ParameterHandler；interceptorChain.pluginAll(resultSetHandler);
16: 预编译sql产生PreparedStatement对象
17: 调用ParameterHandler设置参数
18: 调用TypeHandler给sql预编译设置参数
19: 查出数据使用ResultSetHandler处理结果；使用TypeHandler获取value值
20: 后续的连接关闭
21: 返回list第一个

查询流程总结：



查询流程总结

代理对象

DefaultSqlSession

Executor

StatementHandler

设置参数

处理结果

ParameterHandler

ResultSetHandler

DefaultParameterHandler
typeHandler.setParameter(ps, i + 1, value, jdbcType);

DefaultResultSetHandler
typeHandler.getResult(rs, column);

TypeHandler

JDBC：Statement；PreparedStatement；

StatementHandler：处理sql语句预编译，设置参数等相关工作；
ParameterHandler：设置预编译参数用的
ResultHandler：处理结果集
TypeHandler：在整个过程中，进行数据库类型和javaBean类型的映射

整个流程总结

1. 根据配置文件（全局，sql映射）初始化Configuration对象
2. 创建一个DefaultSqlSession对象，里面包含了Configration以及Executor(根据全局配置文件中的defaultExecutroyType创建出对应的Executor)
3. DefaultSqlSession.getMapper():拿到Mapper接口对应的MapperProxy对象
4. MapperProxy里面有(DefaultSqlSession)
5. 执行增删改查方法：
    1. 调用DefaultSqlSession的增删改查(Executro);
    2. 创建一个StatementHandle对象(同时也会创建出ParameterHandle 和 ResultSetHandler)
    3. 调用StatementHandler预编译参数以及设置参数值，使用ParameterHandler来给sql设置参数
    4. 调用StatementHandler的增删改查方法
    5. ResultSetHandler封装结果
6. 注意：

   四大对象每个创建的过程中都有一个interceptorChain.pluginAll(parameterHandler)

师讲的

```
new Configuration(){
    Properties valiableus;//放的是settings配置
    Map<String,Class> typeAlias;//放的是别名配置
    Map<String,MappedStatent> mappedStataments;//放的是映射文件的select|insert|update|delte
    Map<String,ResultMap> resultMaps;//放的是映射文件中的resultMap标签的解析
}

new XMLConfigBuilder(InputStream in);//解析配置文件,new Configuration()
Configuration config = XMLConfigBuilder.parse();//向Configuration对象填充数据
SqlSessionFactory factory = new DefaultSqlSessionFactory(config);//共产模式的应用


new XMLMapperBuilder(InputStream in,config);//解析映射文件
XMLMapperBuilder.parse()


new XMLStatementBuilder(config,"namespace","select标签")
XMLStatementBuilder.parseStatementNode()
```

```
XMLConfigBuilder parser = new XMLConfigBuilder(inputStream, environment, properties);
var5 = this.build(parser.parse());
this.parseConfiguration(this.parser.evalNode("/configuration"));
private void environmentsElement(XNode context) throws Exception {
Builder environmentBuilder = (new Builder(id)).transactionFactory(txFactory).dataSource(dataSource);}
private void mapperElement(XNode parent) throws Exception {

}
```

```
inputStream = Resources.getResourceAsStream(resource);
mapperParser = new XMLMapperBuilder(inputStream, this.configuration, resource, this.configuration.getSqlFragments());
mapperParser.parse();
this.configurationElement(this.parser.evalNode("/mapper"));
this.parameterMapElement(context.evalNodes("/mapper/parameterMap"));
 this.resultMapElements(context.evalNodes("/mapper/resultMap"));
 this.sqlElement(context.evalNodes("/mapper/sql"));
 this.buildStatementFromContext(context.evalNodes("select|insert|update|delete"));

private void buildStatementFromContext(List<XNode> list, String requiredDatabaseId){
  while(i$.hasNext()) {
    XNode context = (XNode)i$.next();
    XMLStatementBuilder statementParser = new XMLStatementBuilder(this.configuration, this.builderAssistant, context, requiredDatabaseId);
    try {
      statementParser.parseStatementNode();
    } catch (IncompleteElementException var7) {
 this.configuration.addIncompleteStatement(statementParser);
    }
  }

}

MappedStatement statement = statementBuilder.build();
this.configuration.addMappedStatement(statement);
id=com.javasm.mapper.SysUserMapper.addObj

```

```
var8 = new DefaultSqlSession(this.configuration, executor, autoCommit);
Environment environment = this.configuration.getEnvironment();
            TransactionFactory transactionFactory = this.getTransactionFactoryFromEnvironment(environment);
            tx = transactionFactory.newTransaction(environment.getDataSource(), level, autoCommit);
            Executor executor = this.configuration.newExecutor(tx, execType);
            var8 = new DefaultSqlSession(this.configuration, executor, autoCommit);
```

构建者模式:构建复杂对象的对象.该对象的职责就是用来构建另外一个单例对象.

XXXBuilder{

new duixiang()

build(){}

parse(){}

}

工厂模式:构建复杂对象,该对象的职责时用来构建一系列的对象.

XXXFactory(){

对象 create(){}

对象 parse(){}

}

//不把对象的new的过程,散乱在代码不同位置.而统一放在工厂类或构建器类中来创建对象.

3. getMapper方法的执行原理

getMapper方法,返回的是接口的代理对象(接口的实现类实例化对象$Proxy8)

```
//回调处理器对象,该对象内的invoke方法会在代理对象的方法执行时被调用.
class MapperProxy implements InvocationHandler{

  public Object invoke(Object proxy,Method method,Object[] args){

  }
}
```

```java
public T newInstance(SqlSession sqlSession) {
  MapperProxy<T> mapperProxy = new MapperProxy(sqlSession, this.mapperInterface, this.methodCache);
  //创建$Proxy1代理类,并实例化,代理对象
  return Proxy.newProxyInstance(this.mapperInterface.getClassLoader(), new Class[]{this.mapperInterface}, mapperProxy);
}
```

总结:

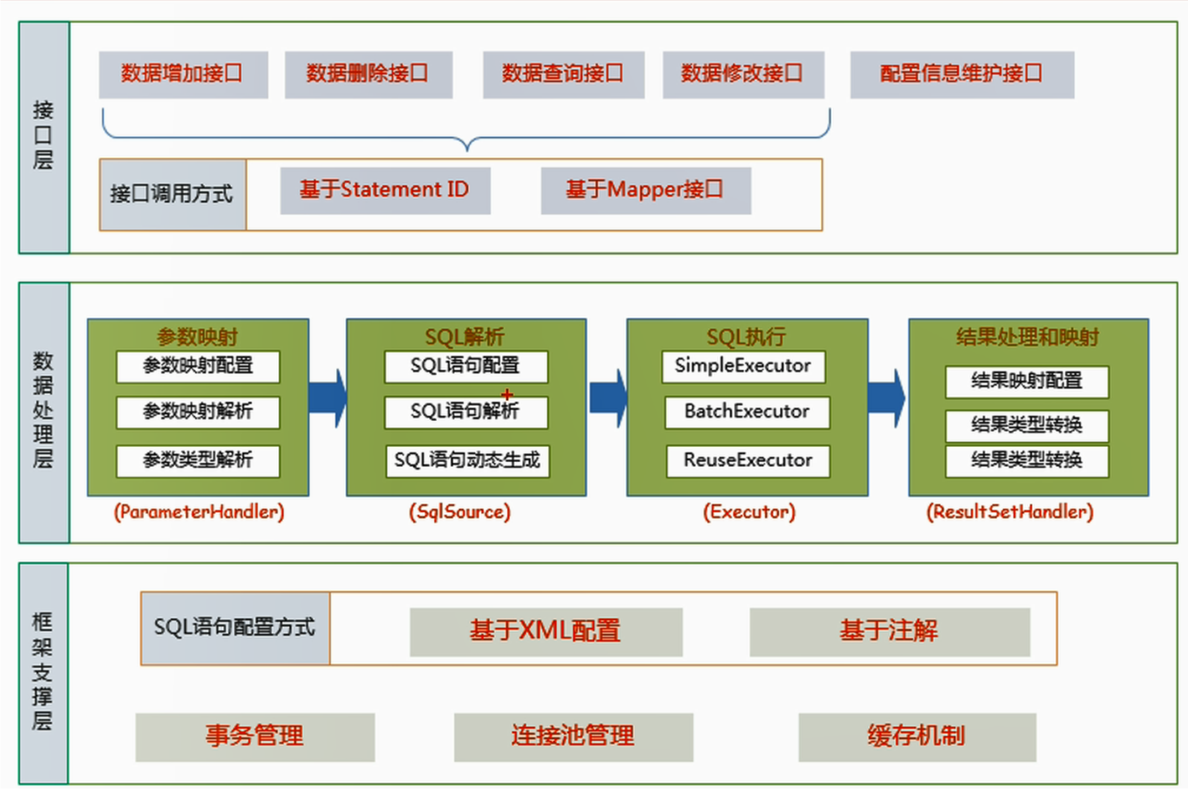
认识构建器模式,工厂模式;

认识Configuration对象;

认识MappedStatement对象;

认识代理模式.了解getMapper方法内返回的到底是什么对象

Mybatis框架的分层架构



接口层是用户进行增删改查操作

数据处理层是与jdbc进行结合，进行操作

框架支持层：配置文件的解析与事务管理

引导层：基于xml配置方式，基于javaAPI的方式。

插件的开发

在四大对象的创建的过程（都允许进行插件的开发）

- 每个对象的创建出来的对象不是直接返回的，而是经过了 interceptorChain.pluginAll(parameterHandler)；
- 获取到所有的Interceptor(拦截器) (插件需要实现接口) 调用 interceptor.plugin(target) 返回target包装后的对象
- 插件机制，可以使用插件为目标对象创建一个代理对象：AOP（面向切面），我们的插件可以为四大对象创建一个代理对象，代理对象可以拦截到四大对象的每一个的执行。

```
public Object pluginAll(Object target){
  for (Interceptor interceptor : interceptors){
    target = interceptor.plugin(target);
  }
  return target;
}
```

插件的编写步骤：

- 编写Interceptor的实现类，重写接口中的方法
- 使用Intercepts注解来完成插件的签名
- 将写好的插件注册到全局配置文件中