

# 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,separator)
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\$Proxy0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

```

1  public class $Proxy0 extends Proxy implements
    Connection{
2
3      protected InvocationHandler h;
4      private Method m38;
5
6      static{
7
8          m38=Class.forName("java.sql.Connection").getMethod("p
repareStatement", new Class[] {
Class.forName("java.lang.String") });
9
10     public $Proxy0(InvocationHandler
paramInvocationHandler)
11     {
12         this.h=h;
13     }
14
15     public final PreparedStatement
prepareStatement(String sql) throws Exception
16     {
17         return (PreparedStatement)this.h.invoke(this,
m38, new Object[]{sql});
18     }
19 }

```

### 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

```

1 //Proxy.newProxyInstance() 创建代理类$Proxy0,并实例化对象的过程.
2 ClassLoader l = source.getClass().getClassLoader();
3 Class[] interfaces=new Class[]{Connection.class};
4 //回调处理器
5 /**
6  * proxy: 代理对象,该对象绝对不能调用,只能传递引用.
7  * method: 正在调用的方法Method对象.
8  * args: 正在调用的方法的实参
9  */
10 InvocationHandler h = new InvocationHandler() {
11     @Override
12     public Object invoke(Object proxy, Method method,
13     Object[] args) throws Throwable {
14         String name = method.getName();
15         if(name.equals("close")){
16             System.out.println("不在关闭,放回连接池");
17         }else{
18             Object result= method.invoke(source, args);
19             return result;
20         }
21     }
22 };

```

- 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

```

1 public class $Proxy0 extends Proxy implements
    Connection{
2
3     protected InvocationHandler h; // 成员变量
4     private Method m38; // 方法对象

```

```

5     static{
6
7         m38=Class.forName("java.sql.Connection").getMethod("p
            repareStatement", new Class[] {
                Class.forName("java.lang.String") });
8     }
9     public $Proxy0(InvocationHandler
        paramInvocationHandler)
10    {
11        super(paramInvocationHandler)
12    }
13    public final PreparedStatement
        prepareStatement(String sql) throws Exception
14    {
15        return (PreparedStatement)this.h.invoke(this,
            m38, new Object[]{sql});
16    }

```

- Proxy中的有参构造

```

1  protected Proxy(InvocationHandle h){
2      objects.requireNonNull(h);
3      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
4  }

```

- 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。

```

1 public final PreparedStatement prepareStatement(String
  sql) throws Exception
2 {
3     return (PreparedStatement)this.h.invoke(this,
  m38, new Object[]{sql});
4 }

```

- 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("prepareSt
atement", 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

```

1 InvocationHandler h = new InvocationHandler() {
2     @Override
3     public Object invoke(Object proxy, Method method,
  Object[] args) throws Throwable {
4         String name = method.getName();
5         if(name.equals("close")){
6             System.out.println("不在关闭,放回连接池");
7         }else{
8             // source 为被代理对象
9             Object result= method.invoke(source, args);
10            // 将结果返回
11            return result;
12        }
13        return null;
14    }
15 };

```

## 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

configuration对象的构建的整个流程

### 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

```
1 | factory = new SqlSessionFactoryBuilder().build(in);
```

- builder方法

```
1 | public SqlSessionFactory build(InputStream  
   | inputStream, String environment, Properties  
   | properties) {  
2 |     SqlSessionFactory var5;  
3 |     try {  
4 |         XMLConfigBuilder parser = new  
   | XMLConfigBuilder(inputStream, environment,  
   | properties);  
5 |         var5 = this.build(parser.parse());  
6 |     } catch (Exception var14) {  
7 |         throw ExceptionFactory.wrapException("Error  
   | building SqlSession.", var14);  
8 |     } finally {  
9 |         ErrorContext.instance().reset();  
10 |        try {  
11 |            inputStream.close();  
12 |        } catch (IOException var13) {  
13 |            ;  
14 |        }  
15 |    }  
16 |    return var5;  
17 | }
```

- 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

```

1 public XMLConfigBuilder(InputStream inputStream,
2   String environment, Properties props) {
3     this(new XPathParser(inputStream, true, props, new
4       XMLMapperEntityResolver()), environment, props);
5 }
6
7 private XMLConfigBuilder(XPathParser parser, String
8   environment, Properties props) {
9     super(new Configuration());
10    this.localReflectorFactory = new
11      DefaultReflectorFactory();
12    ErrorContext.instance().resource("SQL Mapper
13      Configuration");
14    this.configuration.setVariables(props);
15    this.parsed = false;
16    this.environment = environment;
17    this.parser = parser;
18 }

```

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

```

1 public class XMLConfigBuilder extends BaseBuilder {
2   // Parsed为判断标记，创建configuration对象是会将parsed置为
3   true
4   // 保证configuration对象的全局唯一性
5   private boolean parsed;
6   private XPathParser parser; // xpath解析对象
7   private String environment;
8   private ReflectorFactory localReflectorFactory;
9   //在父类BaseBuilder中的成员变量，且是final类型，即：地址不可
10  变（全局唯一）， 类型为protected,即：子类可以使用父类的该属性
11  protected final Configuration configuration;
12  public Configuration parse() {
13    if (this.parsed) {
14      throw new BuilderException("Each
15        XMLConfigBuilder can only be used once.");
16    } else {
17      this.parsed = true;
18    }
19  }
20 }

```

```

15     this.parseConfiguration(this.parser.evalNode("/configuration"));
16     return this.configuration;
17 }
18 }
19 }

```

- 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

```

1  public class XPathImpl implements
   javax.xml.xpath.XPath {
2
3      public Object evaluate(String expression, Object
   item, QName returnType)
4          throws XPathExpressionException {
5          if ( expression == null ) {
6              String fmsg = XSLMessages.createXPathMessage(
7                  XPATHErrorResources.ER_ARG_CANNOT_BE_NULL,
8                  new Object[] { "XPath expression" } );
9              throw new NullPointerException ( fmsg );
10         }
11         if ( returnType == null ) {
12             String fmsg = XSLMessages.createXPathMessage(
13                 XPATHErrorResources.ER_ARG_CANNOT_BE_NULL,
14                 new Object[] { "returnType" } );
15             throw new NullPointerException ( fmsg );
16         }
17         // Checking if requested returnType is supported.
   returnType need to
18         // be defined in XPathConstants
19         if ( !isSupported ( returnType ) ) {
20             String fmsg = XSLMessages.createXPathMessage(
21
22                 XPATHErrorResources.ER_UNSUPPORTED_RETURN_TYPE,
23                 new Object[] { returnType.toString() } );
24             throw new IllegalArgumentException ( fmsg );
25         }
26         try {

```



```

27      // 核心业务代码
28     XObject resultObject = eval( expression, item );
29      return getResultAsType( resultObject, returnType
);
30  } catch ( java.lang.NullPointerException npe ) {
31      // If VariableResolver returns null Or if we get
32      // NullPointerException at this stage for some
other reason
33      // then we have to reurn XPathException
34      throw new XPathExpressionException ( npe );
35  } catch ( javax.xml.transform.TransformerException
te ) {
36      Throwable nestedException = te.getException();
37      if ( nestedException instanceof
javax.xml.xpath.XPathFunctionException ) {
38          throw
(javax.xml.xpath.XPathFunctionException)nestedExceptio
n;
39      } else {
40          // For any other exceptions we need to throw
41          // XPathExpressionException ( as per spec )
42          throw new XPathExpressionException ( te );
43      }
44  }
45  }
46  }

```

- 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

```

1  public class XMLConfigBuilder extends BaseBuilder{
2      private boolean parsed;
3      private XPathParser parser;
4      private String environment;
5      private ReflectorFactory localReflectorFactory;
6      private void parseConfiguration(XNode root) {
7          try {
8              Properties settings =
this.settingsAsPropertiess(root.evalNode("settings"));
9              this.propertiesElement(root.evalNode("properties"));

```

```

10         this.loadCustomVfs(settings);
11
12         this.typeAliasesElement(root.evalNode("typeAliases"))
13         ;
14         this.pluginElement(root.evalNode("plugins"));
15
16         this.objectFactoryElement(root.evalNode("objectFactory
17         y"));
18
19         this.objectWrapperFactoryElement(root.evalNode("objec
20         tWrapperFactory"));
21
22         this.reflectorFactoryElement(root.evalNode("reflector
23         Factory"));
24         this.settingsElement(settings);
25
26         this.environmentsElement(root.evalNode("environments"
27         ));
28
29         this.databaseIdProviderElement(root.evalNode("databas
30         eIdProvider"));
31
32         this.typeHandlerElement(root.evalNode("typeHandlers")
33         );
34         this.mapperElement(root.evalNode("mappers"));
35     } catch (Exception var3) {
36         throw new BuilderException("Error parsing SQL
37         Mapper Configuration. Cause: " + var3, var3);
38     }
39 }

```

- (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

```

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

```

```
5     while(true) {
6         while(i$.hasNext()) {
7             XNode child = (XNode)i$.next();
8             String resource;
9             if ("package".equals(child.getName())) {
10                 resource = child.getStringAttribute("name");
11                 this.configuration.addMappers(resource);
12             } else {
13                 resource =
14 child.getStringAttribute("resource");
15                 String url =
16 child.getStringAttribute("url");
17                 String mapperClass =
18 child.getStringAttribute("class");
19                 XMLMapperBuilder mapperParser;
20                 InputStream inputStream;
21                 if (resource != null && url == null &&
22 mapperClass == null) {
23                     ErrorContext.instance().resource(resource);
24                     inputStream =
25 Resources.getResourceAsStream(resource);
26                     mapperParser = new
27 XMLMapperBuilder(inputStream, this.configuration,
28 resource, this.configuration.getSqlFragments());
29                     mapperParser.parse();
30                 } else if (resource == null && url != null
31 && mapperClass == null) {
32                     ErrorContext.instance().resource(url);
33                     inputStream =
34 Resources.getUrlAsStream(url);
35                     mapperParser = new
36 XMLMapperBuilder(inputStream, this.configuration, url,
37 this.configuration.getSqlFragments());
38                     mapperParser.parse();
39                 } else {
40                     if (resource != null || url != null ||
41 mapperClass != null) {
42                         throw new BuilderException("A mapper
43 element may only specify a url, resource or class, but
44 not more than one.");
45                     }
46                 }
47             }
48         }
49     }
```

```

31         }
32
33         Class<?> mapperInterface =
Resources.classForName(mapperClass);
34
    this.configuration.addMapper(mapperInterface);
35
    }
36
    }
37
    }
38     return;
39 }
40 }
41 }

```

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

```

1 public SqlSessionFactory build(Configuration config) {
2     return new DefaultSqlSessionFactory(config);
3 }

```

- 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

- 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

```

1 public class DefaultSqlSessionFactory implements
SqlSessionFactory {
2     private final Configuration configuration;
3     // 构造器，SqlSessionFactory对象中的configuration已经赋值
    了
4     public DefaultSqlSessionFactory(Configuration
configuration) {
5         this.configuration = configuration;
6     }
7

```

```

8      private SqlSession
      openSessionFromDataSource(ExecutorType execType,
      TransactionIsolationLevel level, boolean autoCommit) {
9          Transaction tx = null;
10
11         DefaultSqlSession var8;
12         try {
13             Environment environment =
14             this.configuration.getEnvironment();
15             TransactionFactory transactionFactory =
16             this.getTransactionFactoryFromEnvironment(environment)
17             ;
18             tx =
19             transactionFactory.newTransaction(environment.getDataSource(), level, autoCommit);
20             Executor executor =
21             this.configuration.newExecutor(tx, execType);
22             // 建立一个DefaultSqlSession对象, 将configuration传入
23             var8 = new DefaultSqlSession(this.configuration,
24             executor, autoCommit);
25         } catch (Exception var12) {
26             this.closeTransaction(tx);
27             throw ExceptionFactory.wrapException("Error
      opening session. Cause: " + var12, var12);
28         } finally {
29             ErrorContext.instance().reset();
30         }
31         return var8;
32     }
33 }

```

- 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

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

```

1 public class DefaultSqlSession implements SqlSession {
2     private Configuration configuration;
3     private Executor executor;
4     private boolean autoCommit;
5     private boolean dirty;
6     private List<Cursor<?>> cursorList;
7     // 上一步openSessionFromDataSource通过该构造生成了一个
    SqlSession对象
8     public DefaultSqlSession(Configuration
configuration, Executor executor, boolean autoCommit)
    {
9         this.configuration = configuration;
10        this.executor = executor;
11        this.dirty = false;
12        this.autoCommit = autoCommit;
13    }
14
15    public DefaultSqlSession(Configuration
configuration, Executor executor) {
16        this(configuration, executor, false);
17    }
18    public <T> T getMapper(Class<T> type) {
19        return this.configuration.getMapper(type, this);
20    }
21 }

```

- 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)

```

1 public class Configuration {
2     public <T> T getMapper(Class<T> type, SqlSession
sqlSession) {
3         return this.mapperRegistry.getMapper(type,
sqlSession);
4     }
5 }

```

- 调用的是mapperRegistry对象中getMapper方法

```

1 public class MapperRegistry {
2     private final Configuration config;

```

```

3     private final Map<Class<?>, MapperProxyFactory<?>>
      knownMappers = new HashMap();
4
5     public MapperRegistry(Configuration config) {
6         this.config = config;
7     }
8     public <T> T getMapper(Class<T> type, SqlSession
      sqlSession) {
9         MapperProxyFactory<T> mapperProxyFactory =
      (MapperProxyFactory)this.knownMappers.get(type);
10        if (mapperProxyFactory == null) {
11            throw new BindingException("Type " + type + " is
      not known to the MapperRegistry.");
12        } else {
13            try {
14                return
      mapperProxyFactory.newInstance(sqlSession);
15            } catch (Exception var5) {
16                throw new BindingException("Error getting
      mapper instance. Cause: " + var5, var5);
17            }
18        }
19    }
20 }

```

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

```

1 public class MapperProxyFactory<T> {
2     public T newInstance(SqlSession sqlSession) {
3         MapperProxy<T> mapperProxy = new
      MapperProxy(sqlSession, this.mapperInterface,
      this.methodCache);
4         return this.newInstance(mapperProxy);
5     }
6 }

```

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

```
1 public class MapperProxy<T> implements
  InvocationHandler, Serializable {
2     private static final long serialVersionUID =
    -6424540398559729838L;
3     private final SqlSession sqlSession;
4     private final Class<T> mapperInterface;
5     private final Map<Method, MapperMethod> methodCache;
6
7     public MapperProxy(SqlSession sqlSession, Class<T>
    mapperInterface, Map<Method, MapperMethod>
    methodCache) {
8         this.sqlSession = sqlSession;
9         this.mapperInterface = mapperInterface;
10        this.methodCache = methodCache;
11    }
12
13    public Object invoke(Object proxy, Method method,
    Object[] args) throws Throwable {
14        if
    (Object.class.equals(method.getDeclaringClass())) {
15            try {
16                return method.invoke(this, args);
17            } catch (Throwable var5) {
18                throw ExceptionUtil.unwrapThrowable(var5);
19            }
20        } else {
21            MapperMethod mapperMethod =
    this.cachedMapperMethod(method);
22            return mapperMethod.execute(this.sqlSession,
    args);
23        }
24    }
25
26    private MapperMethod cachedMapperMethod(Method
    method) {
27        MapperMethod mapperMethod =
    (MapperMethod) this.methodCache.get(method);
28        if (mapperMethod == null) {
29            mapperMethod = new
    MapperMethod(this.mapperInterface, method,
    this.sqlSession.getConfiguration());
```



```

30         this.methodCache.put(method, mapperMethod);
31     }
32
33     return mapperMethod;
34 }
35 }

```

- 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);

```

1 public class MapperProxyFactory<T> {
2     protected T newInstance(MapperProxy<T> mapperProxy) {
3         return
4         Proxy.newProxyInstance(this.mapperInterface.getClassLoader(), new Class[]{this.mapperInterface}, mapperProxy);
5     }
6 }

```

- 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。

## 老师讲的

```

1 new Configuration(){
2     Properties variableus;//放的是settings配置
3     Map<String,Class> typeAlias;//放的是别名配置
4     Map<String,MappedStatent> mappedStataments;//放的是
    映射文件的select|insert|update|delte
5     Map<String,ResultMap> resultMap;//放的是映射文件中的
    resultMap标签的解析
6 }
7
8 new XMLConfigBuilder(InputStream in);//解析配置文件,new
    Configuration()
9 Configuration config = XMLConfigBuilder.parse();//向
    Configuration对象填充数据
10 SqlSessionFactory factory = new
    DefaultSqlSessionFactory(config);//共产模式的应用

```

```

11
12
13 new XMLMapperBuilder(InputStream in,config);//解析映射文件
14 XMLMapperBuilder.parse()
15
16
17 new XMLStatementBuilder(config,"namespace","select标签")
18 XMLStatementBuilder.parseStatementNode()

```

```

1 XMLConfigBuilder parser = new
XMLConfigBuilder(inputStream, environment, properties);
2 var5 = this.build(parser.parse());
3 this.parseConfiguration(this.parser.evalNode("/configuration"));
4 private void environmentsElement(XNode context) throws
Exception {
5 Builder environmentBuilder = (new
Builder(id)).transactionFactory(txFactory).dataSource(dataSource);}
6 private void mapperElement(XNode parent) throws
Exception {
7
8 }

```

```

1 inputStream = Resources.getResourceAsStream(resource);
2 mapperParser = new XMLMapperBuilder(inputStream,
this.configuration, resource,
this.configuration.getSqlFragments());
3 mapperParser.parse();
4 this.configurationElement(this.parser.evalNode("/mapper"));
5 this.parameterMapElement(context.evalNodes("/mapper/parameterMap"));
6
this.resultMapElements(context.evalNodes("/mapper/resultMap"));

```

```
7      this.sqlElement(context.evalNodes("/mapper/sql"));
8
9      this.buildStatementFromContext(context.evalNodes("select|insert|update|delete"));
10
11 private void buildStatementFromContext(List<XNode>
12 list, String requiredDatabaseId){
13     while(i$.hasNext()) {
14         XNode context = (XNode)i$.next();
15         XMLStatementBuilder statementParser = new
16 XMLStatementBuilder(this.configuration,
17 this.builderAssistant, context, requiredDatabaseId);
18         try {
19             statementParser.parseStatementNode();
20         } catch (IncompleteElementException var7) {
21
22             this.configuration.addIncompleteStatement(statementParser);
23         }
24     }
25 }
26
27 MappedStatement statement = statementBuilder.build();
28 this.configuration.addMappedStatement(statement);
29 id=com.javasm.mapper.SysUserMapper.addObject
```

```

1  var8 = new DefaultSqlSession(this.configuration,
    executor, autoCommit);
2  Environment environment =
    this.configuration.getEnvironment();
3      TransactionFactory transactionFactory =
    this.getTransactionFactoryFromEnvironment(environment);
4      tx =
    transactionFactory.newTransaction(environment.getDataSo
    urce(), level, autoCommit);
5      Executor executor =
    this.configuration.newExecutor(tx, execType);
6      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)

```
1 //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.
2 class MapperProxy implements InvocationHandler{
3
4     public Object invoke(Object proxy,Method
      method,Object[] args){
5
6     }
7 }
```

```
1 public T newInstance(SqlSession sqlSession) {
2     MapperProxy<T> mapperProxy = new
      MapperProxy(sqlSession, this.mapperInterface,
      this.methodCache);
3     //创建$Proxy1代理类,并实例化,代理对象
4     return
      Proxy.newProxyInstance(this.mapperInterface.getClassLo
      ader(), new Class[]{this.mapperInterface},
      mapperProxy);
5 }
```

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

