* 学习目标

  * 能够掌握自定义MyBatis的核心框架

    * 前期准备

      * 数据库表--实体类--UserDao

      * 全局配置文件

        * SqlMapConfig

          * configuration

            * environments

              * environment

                * dataSource

                  * property

            * mappers

              * mapper:resource,class

      * DaoXML配置

        * mapper:namespace

          <select id="" resultType="">select * from t_user</select>

      * Dao注解配置

        UserDao

        @Select("select * from t_user");

        * List<User> getUsers();

    * 封装配置类

      * Configuration：driver，url，username，password，mappers

      * Mapper：sql，resultType

    * 编写工具类

      * ConnectionUtils(Configuration)

      * Resources

      * XMLUtils

* Xpath

* 可以分xml解析和注解解析

* Executor:真正操作数据库

<E> List<E> selectList(Connection,Mapper)

* 元数据，BeanUtils，反射

* 核心编写

* 两个接口

* SqlSession：<T> T getMapper(Class<T> clazz),close

* SqlSessionFactory:openSession();

* 两个具体实现类

* DefaultSqlSession:Proxy:动态代理对象

* MapperProxy---InvocationHandler

invoke--->key--mappers---Executor--selectList

* DefaultSqlSessionFactory：openSession();

* Configuration配置类

· * SqlSessionFactoryBuilder

SqlSessionFactory ssf= build(InputStream is);

Configuration cfg=XMLUtils.loadConfiguration(is);

* 准备单元测试

* 发现一问题：我们框架不支持toString

* SqlSessionFactory ssf=SqlSessionFactoryBuilder.build(is);

* SqlSession sqlSession=ssf.openSession();

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

List<User> users=User.getUsers();

* sqlSession.close();

---

* 回顾

* Maven

  * 仓库管理：本地仓库，远程仓库（中央仓库，私服）

  * 生命周期：clean，default(compile,test,install) site

  * 插件

* Maven与Idea进行结合开发

  * Java项目，JavaWeb项目

* Maven的聚合和继承

  * Project---Module

* Maven实战

  * 按开发方式:MVC和Web三层架构

    * domain，dao，service，web

  * 按功能划分

    * common （Employee）

    * customer（domain，dao，service，web）

    * employee

    * pay

  * 私服

    * 部署我自己项目依赖

    * 在局域网内使用，有一台有访问外网权限服务器

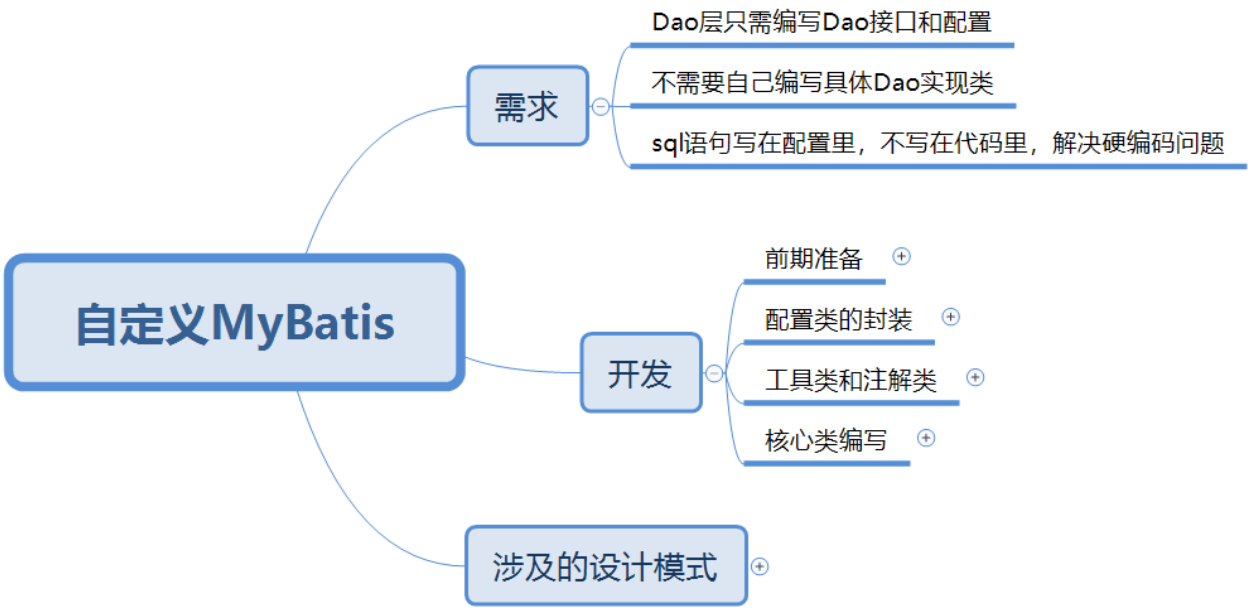
    * nexus

    * group，hosted，proxy

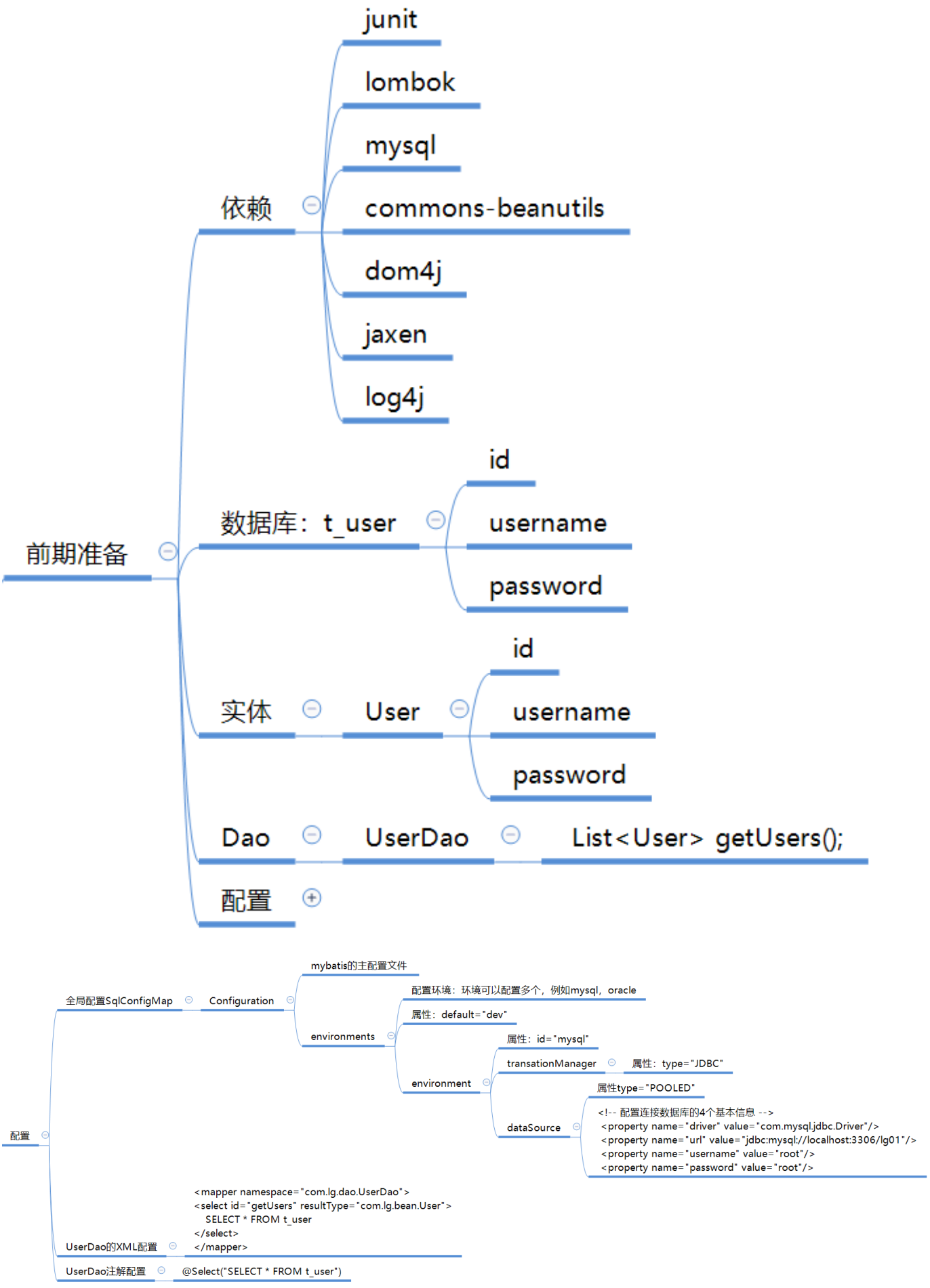
    * 在setting文件中配置

* 能够自定义MyBatis的核心框架

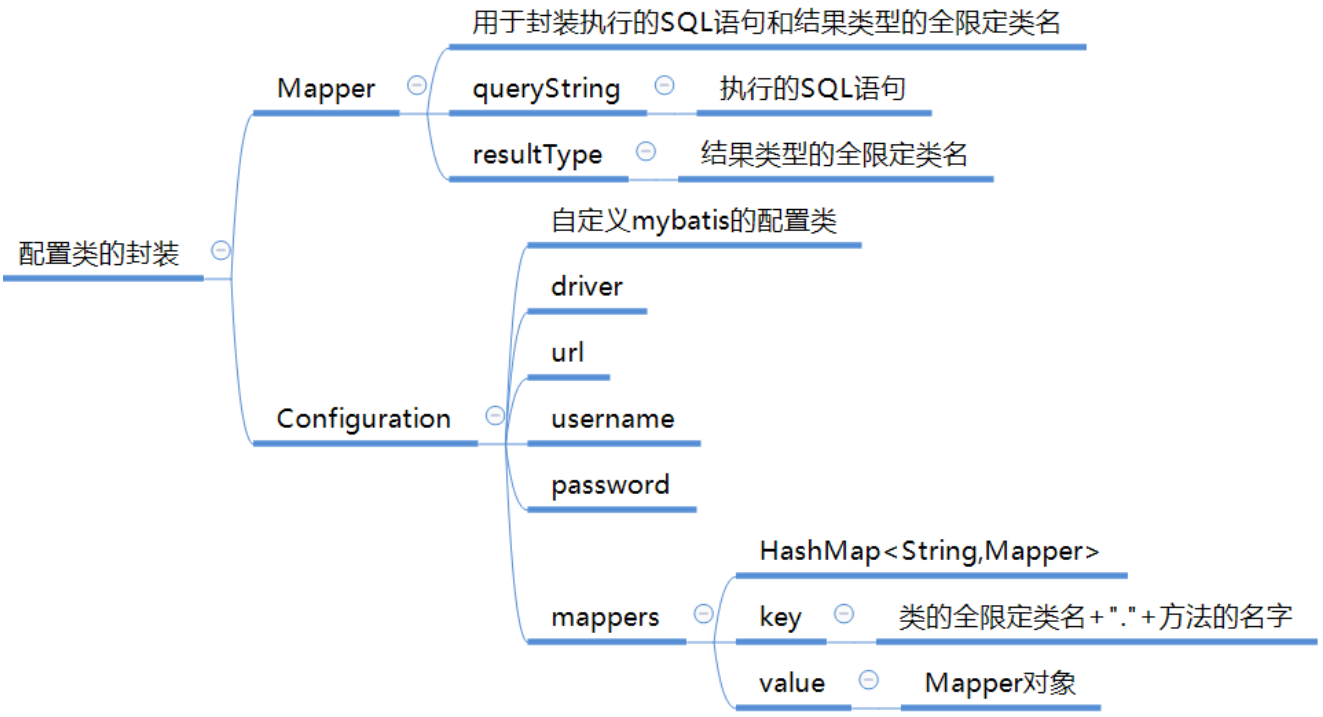
  * 概述


自定义MyBatis
需求
Dao层只需编写Dao接口和配置
不需要自己编写具体Dao实现类
sql语句写在配置里，不写在代码里，解决硬编码问题
开发
前期准备
配置类的封装
工具类和注解类
核心类编写
涉及的设计模式

* 前期准备


前期准备
依赖
junit
lombok
mysql
commons-beanutils
dom4j
jaxen
log4j
数据库：t_user
id
username
password
实体
User
id
username
password
Dao
UserDao
List<User> getUsers();
配置
配置
全局配置SqlConfigMap
Configuration
mybatis的主配置文件
environments
配置环境：环境可以配置多个，例如mysql，oracle
属性：default="dev"
environment
属性：id="mysql"
transationManager
属性：type="JDBC"
dataSource
属性type="POOLED"
<!-- 配置连接数据库的4个基本信息 -->
<property name="driver" value="com.mysql.jdbc.Driver"/>
<property name="url" value="jdbc:mysql://localhost:3306/lg01"/>
<property name="username" value="root"/>
<property name="password" value="root"/>
UserDao的XML配置
<mapper namespace="com.lg.dao.UserDao">
<select id="getUsers" resultType="com.lg.bean.User">
SELECT * FROM t_user
</select>
</mapper>
UserDao注解配置
@Select("SELECT * FROM t_user")

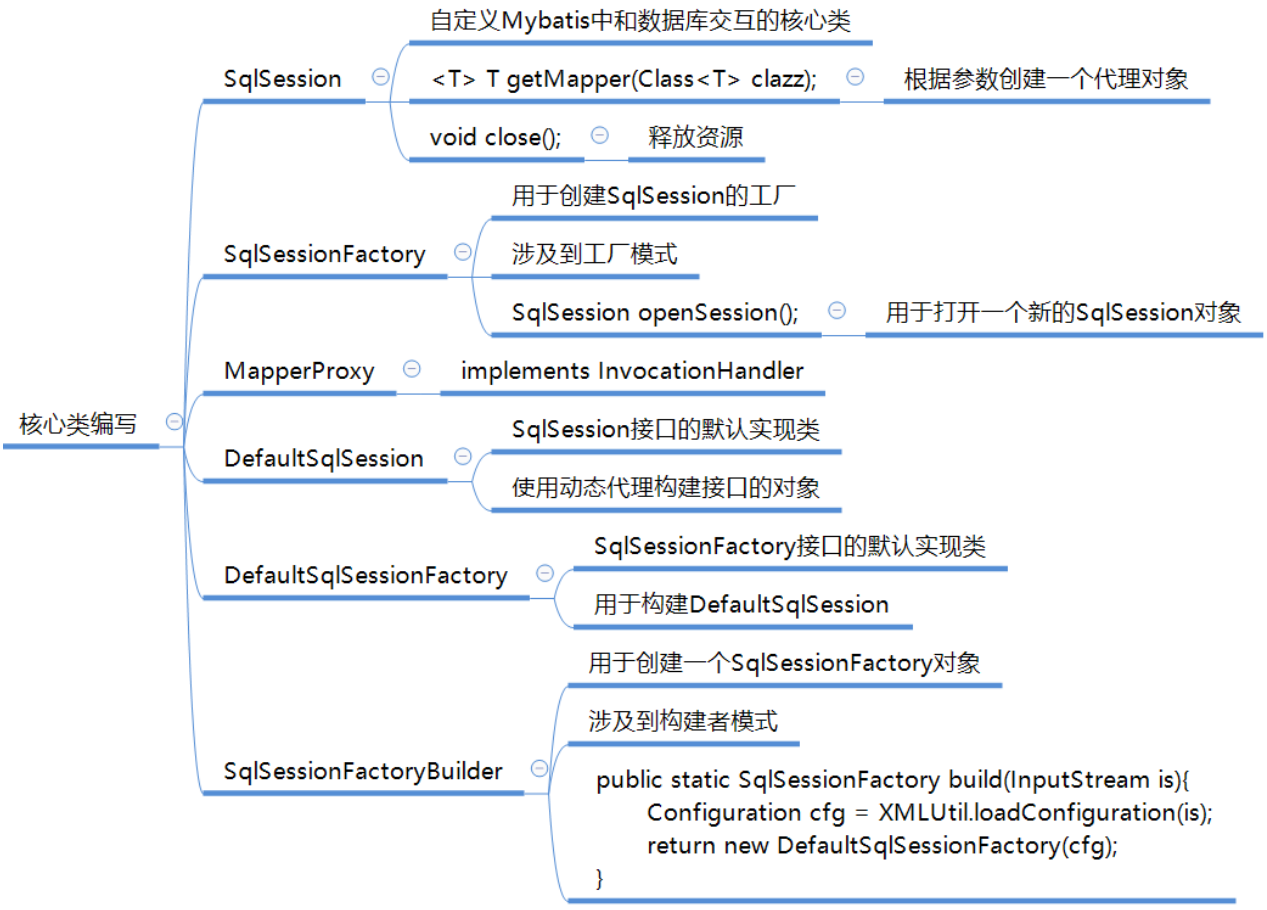
* 配置类的封装


配置类的封装
Mapper
用于封装执行的SQL语句和结果类型的全限定类名
queryString
执行的SQL语句
resultType
结果类型的全限定类名
Configuration
自定义mybatis的配置类
driver
url
username
password
mappers
HashMap<String,Mapper>
key
类的全限定类名+"."+方法的名字
value
Mapper对象

* 工具类和注解类


@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Select {
String value();
}
工具类和注解类
Select
ConnectionUtil
得数据库链接的工具类
Resources
使用类加载器读取配置文件的类
XMLUtil
解析主配置文件和mapper配置
Executor
负责执行SQL语句，并且封装结果集

* 核心类


核心类编写
SqlSession
自定义Mybatis中和数据库交互的核心类
<T> T getMapper(Class<T> clazz);
根据参数创建一个代理对象
void close();
释放资源
SqlSessionFactory
用于创建SqlSession的工厂
涉及到工厂模式
SqlSession openSession();
用于打开一个新的SqlSession对象
MapperProxy
implements InvocationHandler
DefaultSqlSession
SqlSession接口的默认实现类
使用动态代理构建接口的对象
DefaultSqlSessionFactory
SqlSessionFactory接口的默认实现类
用于构建DefaultSqlSession
SqlSessionFactoryBuilder
用于创建一个SqlSessionFactory对象
涉及到构建者模式
public static SqlSessionFactory build(InputStream is){
Configuration cfg = XMLUtil.loadConfiguration(is);
return new DefaultSqlSessionFactory(cfg);
}

* 涉及工厂模式


涉及的设计模式
动态代理模式（已经讲过）
工厂模式
构建者模式

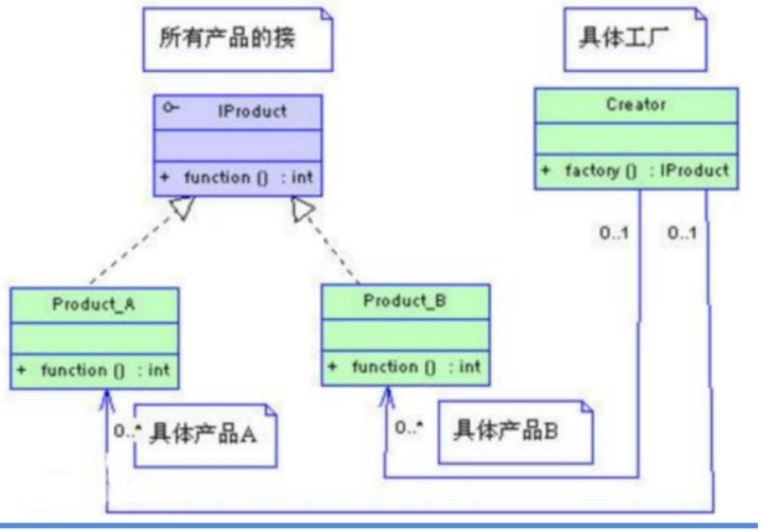
* 工厂模式

工厂模式

概述

工厂模式（Factory Pattern）是 Java 中最常用的设计模式之一。这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式

在工厂模式中，我们在创建对象时不会对客户端暴露创建逻辑，并且是通过使用一个共同的接口来指向新创建的对象。



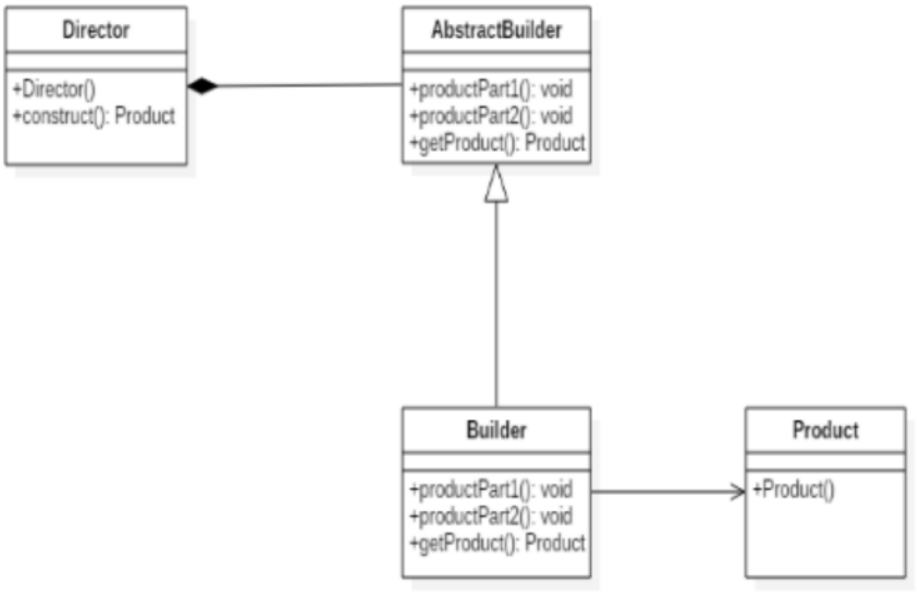
* 构建者模式

构建者模式

概述

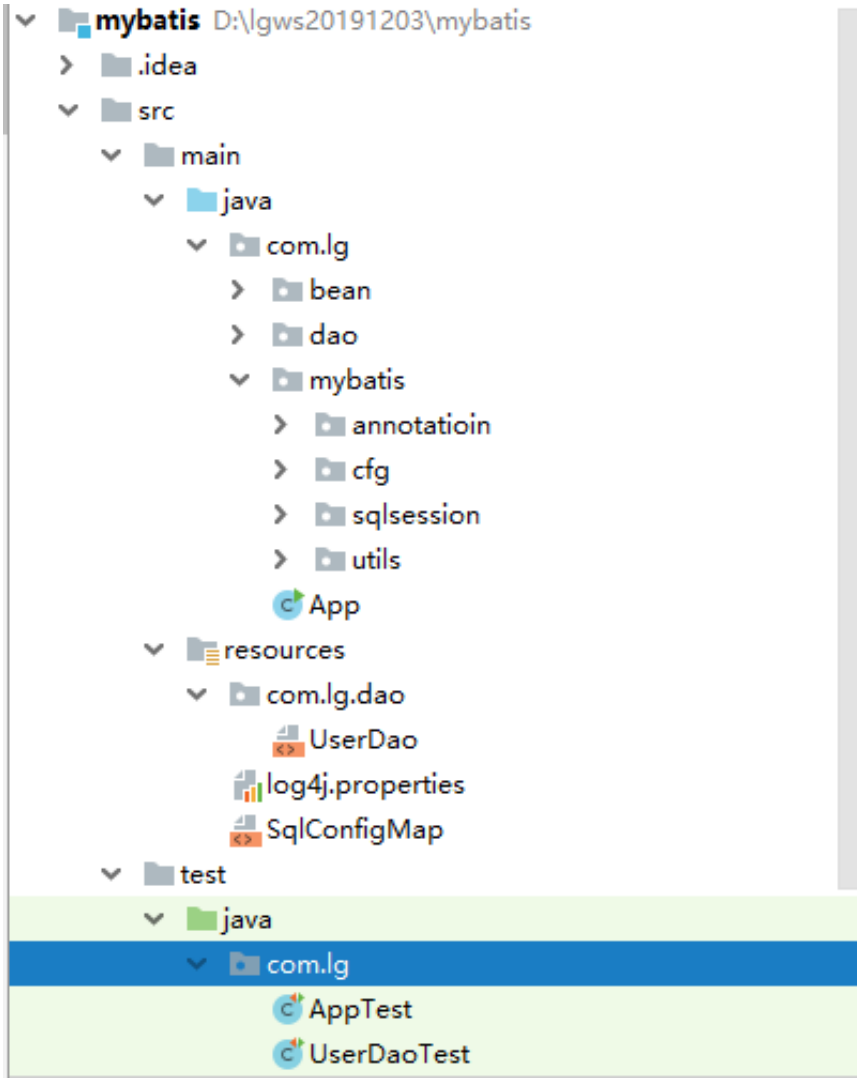
建造者模式（Builder Pattern）使用多个简单的对象一步一步构建成一个复杂的对象。这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。

一个 Builder 类会一步一步构造最终的对象。该 Builder 类是独立于其他对象的。



* 案例

* 项目结构




```
* 依赖
  <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.11</version>
      <scope>test</scope>
    </dependency>
    <!-- https://mvnrepository.com/artifact/org.projectlombok/lombok -->
    <dependency>
      <groupId>org.projectlombok</groupId>
      <artifactId>lombok</artifactId>
      <version>1.18.10</version>
      <scope>provided</scope>
    </dependency>
    <!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->
    <dependency>
      <groupId>mysql</groupId>
```

```
      <artifactId>mysql-connector-java</artifactId>
      <version>5.1.47</version>
    </dependency>
    <!-- https://mvnrepository.com/artifact/commons-beanutils/commons-beanutils
    <dependency>
      <groupId>commons-beanutils</groupId>
      <artifactId>commons-beanutils</artifactId>
      <version>1.9.4</version>
    </dependency>
    <dependency>
      <groupId>dom4j</groupId>
      <artifactId>dom4j</artifactId>
      <version>1.6.1</version>
    </dependency>
    <dependency>
      <groupId>jaxen</groupId>
      <artifactId>jaxen</artifactId>
      <version>1.1.6</version>
    </dependency>
    <dependency>
      <groupId>log4j</groupId>
      <artifactId>log4j</artifactId>
      <version>1.2.12</version>
 </dependency>
 * 实体类
@Data
@AllArgsConstructor
@NoArgsConstructor
public class User {
    private int id;
    private String username;
    private String password;
}
 * Dao
public interface UserDao {
//     @Select("SELECT * FROM t_user")
     List<User> getUsers();
}
 * 全局配置SqlConfigMap
<?xml version="1.0" encoding="UTF-8"?>
```

```
<!-- mybatis的主配置文件 -->
<Configuration>
    <!-- 配置环境 -->
    <environments default="dev">
        <!-- 配置mysql的环境-->
        <environment id="mysql">
        <!-- 配置事务的类型-->
        <transationManager type="JDBC"></transationManager>
        <!-- 配置数据源（连接池） -->
        <dataSource type="POOLED">
            <!-- 配置连接数据库的4个基本信息 -->
            <property name="driver" value="com.mysql.jdbc.Driver"/>
            <property name="url" value="jdbc:mysql://localhost:3306/lg01"/>
            <property name="username" value="root"/>
            <property name="password" value="root"/>
        </dataSource>
    </environment>
    </environments>
    <mappers>
      <mapper resource="com/lg/dao/UserDao"></mapper>
      <!--  <mapper class="com.lg.dao.UserDao"></mapper>-->
    </mappers>
</Configuration>
 * UserDao的XML配置
 <?xml version="1.0" encoding="UTF-8"?>
<mapper namespace="com.lg.dao.UserDao">
<select id="getUsers" resultType="com.lg.bean.User">
    SELECT * FROM t_user
</select>
</mapper>
 * UserDao注解配置
  * @Select("SELECT * FROM t_user")
 * 配置类的封装
@Data
public class Mapper {
    /**
     * SQL
     */
    private String queryString;

```

```
    /**
     * 返回结果类型
     */
    private String resultType;
}
@Data
public class Configuration {
    private String driver;
    private String url;
    private String username;
    private String password;
    private Map<String,Mapper> mappers=new HashMap<String,Mapper>();
}

* 工具类和注解类
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Select {
    String value();
}
public class ConnectionUtil {
    /**
     * @param cfg
     * @return
     * 获得数据库链接
     */
    public static Connection getConnection(Configuration cfg){
            try {
                String driver = cfg.getDriver();
                if(driver!=null){
                    Class.forName(driver);
                }
                return DriverManager.getConnection(cfg.getUrl(),cfg.getUsername
            } catch (Exception e) {
                throw  new RuntimeException(e);
            }
        }
}

public class Resources {
```

```
138     /**
139      * @param filePath
140      * @return
141      * 根据传入的参数，获取一个字节输入流
142      */
143     public static InputStream getResourcesAsStream(String filePath){
144         return Resources.class.getClassLoader().getResourceAsStream(filePath);
145     }
146 }
147 
148 /**
149  * admin
150  */
151 public class XMLUtils {
152 
153     /**
154      * @param is
155      * @return
156      * 通过文件输入流，获取Configuration配置类
157      */
158     public static Configuration loadConfiguration(InputStream is){
159         try {
160             Configuration cfg=new Configuration();
161             SAXReader reader=new SAXReader();
162             Document doc = reader.read(is);
163             Element rootElement = doc.getRootElement();
164             // Xpath
165             List<Element> propertyElements = rootElement.selectNodes("//propert
166             for (Element propertyElement : propertyElements) {
167                 String name = propertyElement.attributeValue("name");
168                 final String value = propertyElement.attributeValue("value");
169                 switch (name){
170                     case "driver":
171                         cfg.setDriver(value);
172                         break;
173                     case "url":
174                         cfg.setUrl(value);
175                         break;
176                     case "username":
177                         cfg.setUsername(value);
```

```
                        break;
                    case "password":
                        cfg.setPassword(value);
                        break;
                }
            }
            // 处理Mappers
            List<Element> mappersElement = rootElement.selectNodes("//mappers/m
            Map<String, Mapper> mappers=new HashMap<String, Mapper>();
            for (Element mapperElement : mappersElement) {
                String resource = mapperElement.attributeValue("resource");
                if(resource!=null){
                    System.out.println("XML配置");
                    loadMapperXMLConfiguration(mappers,resource);
                }else{
                    System.out.println("注解配置");
                    String clazz = mapperElement.attributeValue("class");
                   loadMapperAnnotationConfiguration(mappers,clazz);
                }

            }
            cfg.setMappers(mappers);

            return cfg;
        } catch (Exception e) {
            throw  new RuntimeException(e);
        }
    }
    private static void loadMapperXMLConfiguration(Map<String,Mapper> mappers,S
        try {
            InputStream is = Resources.getResourcesAsStream(resource);
            SAXReader reader=new SAXReader();
            Document document = reader.read(is);
            Element rootElement = document.getRootElement();
            String namespace = rootElement.attributeValue("namespace");
            List<Element> selectsElement = rootElement.selectNodes("//select");
            for (Element selectElement : selectsElement) {
                String id = selectElement.attributeValue("id");
                String resultType = selectElement.attributeValue("resultType");
                String sql = selectElement.getText();
```

```
                String key=namespace+"."+id;
                Mapper mapper=new Mapper();
                mapper.setQueryString(sql);
                mapper.setResultType(resultType);
                mappers.put(key,mapper);
            }
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
    private static void loadMapperAnnotationConfiguration(Map<String,Mapper> ma
          try{
              Class<?> clazz = Class.forName(clazzPath);
              Method[] methods = clazz.getMethods();
              for (Method method : methods) {
                  boolean annotationPresent = method.isAnnotationPresent(Select
                  if(annotationPresent){
                      Select selectAnnotation = method.getAnnotation(Select.cla
                      String sql = selectAnnotation.value();
                      Mapper mapper=new Mapper();
                      mapper.setQueryString(sql);
                      // List<User>:带有泛型
//                      mapper.setResultType();
                      Type type = method.getGenericReturnType();
                      if(type instanceof ParameterizedType){
                          ParameterizedType parameterizedType = (ParameterizedT
                          Type[] types = parameterizedType.getActualTypeArgumer
                          Class tClazz = (Class)types[0];
                          String className=tClazz.getName();
                          mapper.setResultType(className);
                      }
                      String key=clazz.getName()+"."+method.getName();
                      mappers.put(key,mapper);
                  }
              }
          }catch (Exception e){
              throw new RuntimeException(e);
          }
    }
```

```
258
259 }
260
261
262 /**
263  * admin
264  * 负责执行SQL语句，并且封装结果集
265  */
266 public class Executor {
267
268     public <E> List<E> selectList(Connection connection, Mapper mapper){
269         PreparedStatement pmst=null;
270         ResultSet rs=null;
271         try {
272             //1.取出mapper中的数据
273             String queryString = mapper.getQueryString();
274             String resultType = mapper.getResultType();
275             Class<?> clazz = Class.forName(resultType);
276             //2.获取PreparedStatement对象
277             pmst = connection.prepareStatement(queryString);
278             //3.执行SQL语句，获取结果集
279             rs = pmst.executeQuery();
280             List<E> list=new ArrayList<E>();
281             //4.封装结果集
282             while(rs.next()){
283                 E e = (E) clazz.newInstance();
284                 //取出结果集的元信息：ResultSetMetaData
285                 ResultSetMetaData rsmd = rs.getMetaData();
286                 //取出总列数
287                 int columnCount = rsmd.getColumnCount();
288                 Map<String,Object> map=new HashMap<String,Object>();
289                 for (int i = 1; i <=columnCount ; i++) {
290                     //获取每列的名称，列名的序号是从1开始的
291                     String columnName = rsmd.getColumnName(i);
292                     //根据得到列名，获取每列的值
293                     Object value = rs.getObject(columnName);
294                     map.put(columnName,value);
295                 }
296                 BeanUtils.populate(e,map);
297                 //把赋好值的对象加入到集合中
```

```
                list.add(e);
            }
            return list;
        } catch (Exception e) {
            throw  new RuntimeException(e);
        }finally {
            release(pmst, rs);
        }
    }

    private void release(PreparedStatement pmst, ResultSet rs) {
        if(rs!=null){
            try {
                rs.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
        if(pmst!=null){
            try {
                pmst.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }
}
/**
 * admin
 * 与数据库操作的核心类
 */
public interface SqlSession {
    <T> T getMapper(Class<T> clazz);
}


/**
 * admin
 * 用于创建SqlSession的工厂
 */
```

```java
public interface SqlSessionFactory {

    /**
     * 用于打开一个新的SqlSession对象
     * @return
     */
    SqlSession openSession();
}

/**
 * admin
 * SqlSession接口的实现类
 */
public class DefaultSqlSession implements SqlSession {
    private Configuration cfg;
    private Connection connection;

    public DefaultSqlSession(Configuration cfg){
        this.cfg=cfg;
        connection= ConnectionUtil.getConnection(cfg);
    }
    @Override
    public <T> T getMapper(Class<T> clazz) {
        return (T)Proxy.newProxyInstance(clazz.getClassLoader(),new Class[]{cla
    }

    @Override
    public void close() {
        if(connection != null) {
            try {
                connection.close();
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
}

/**
 * admin
```

```
 */
public class MapperProxy implements InvocationHandler {
    //map的key是全限定类名+方法名
    private Map<String, Mapper> mappers;
    private Connection connection;

    public MapperProxy(Map<String, Mapper> mappers,Connection connection){
        this.mappers=mappers;
        this.connection=connection;
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Thr
        //1.获取方法名
        String methodName = method.getName();
        //2.获取方法所在类的名称
        String clazzName = method.getDeclaringClass().getName();
        //3.组合key
        String key=clazzName+"."+methodName;
        //4.获取mappers中的Mapper对象
        Mapper mapper = mappers.get(key);
        //5.判断是否有mapper
        if(mapper==null){
            throw new IllegalArgumentException("mapper is null");
        }
        //6.调用工具类执行查询所有
        Executor executor=new Executor();
        List<Object> results = executor.selectList(connection, mapper);
        return results;
    }
}

/**
 * admin
 */
public class DefaultSqlSessionFactory implements SqlSessionFactory {
    private Configuration cfg;
    public DefaultSqlSessionFactory(Configuration cfg){
        this.cfg=cfg;
    }
```

```
418     @Override
419     public SqlSession openSession() {
420         return new DefaultSqlSession(cfg);
421     }
422 }
423 
424 /**
425  * admin
426  * 用于创建一个SqlSessionFactory对象
427  */
428 public class SqlSessionFactoryBuilder {
429 
430     /**
431      * @param is
432      * @return
433      * 根据参数的字节输入流来构建一个SqlSessionFactory工厂
434      */
435     public static SqlSessionFactory build(InputStream is){
436         Configuration cfg = XMLUtil.loadConfiguration(is);
437         return new DefaultSqlSessionFactory(cfg);
438     }
439 }
440 
441 * Dao测试
442 public class UserDaoTest {
443     @Test
444     public void test1(){
445         InputStream is = Resources.getResourcesAsStream("SqlConfigMap");
446         SqlSessionFactory ssf = SqlSessionFactoryBuilder.build(is);
447         SqlSession sqlSession = ssf.openSession();
448         UserDao userDao = sqlSession.getMapper(UserDao.class);
449         List<User> users = userDao.getUsers();
450         System.out.println(users);
451         sqlSession.close();
452     }
453 }
454 
```

```
[User(id=1, username=xiaohei, password=123), User(id=2, username=xiaobai, password=123)]

Process finished with exit code 0
```