

# MyBatis 事务管理解析：颠覆你心中对事务的理解！

祖大俊 Java后端 1月19日

点击上方 Java后端, 选择 [设为星标](#)

优质文章, 及时送达

作者 | 祖大俊

编辑 | 程序员DD

来源 | [my.oschina.net/zudajun/blog/666764](http://my.oschina.net/zudajun/blog/666764)

## 1.说到数据库事务，人们脑海里自然不自然的就会浮现出事务的四大特性、四大隔离级别、七大传播特性。

四大还好说，问题是七大传播特性是哪儿来的？是Spring在当前线程内，处理多个数据库操作方法事务时所做的一种事务应用策略。

事务本身并不存在什么传播特性，不要混淆事务本身和Spring的事务应用策略。（当然，找工作面试时，还是可以巧妙的描述传播特性的）

## 2.一说到事务，人们可能又会想起**create**、**begin**、**commit**、**rollback**、**close**、**suspend**。

可实际上，只有**commit**、**rollback**是实际存在的，剩下的**create**、**begin**、**close**、**suspend**都是虚幻的，是业务层或数据库底层应用语意，而非JDBC事务的真实命令。

**create (事务创建)** : 不存在。

**begin (事务开始)** : 姑且认为存在于DB的命令行中，比如Mysql的start transaction命令，以及其他数据库中的begin transaction命令。JDBC中不存在。

**close (事务关闭)** : 不存在。应用程序接口中的close()方法，是为了把connection放回数据库连接池中，供下一次使用，与事务毫无关系。

**suspend (事务挂起)** : 不存在。

Spring中事务挂起的含义是，需要新事务时，将现有的connection1保存起来（它还有尚未提交的事务），然后创建connection2，connection2提交、回滚、关闭完毕后，再把connection1取出来，完成提交、回滚、关闭等动作，保存connection1的动作称之为事务挂起。

在JDBC中，是根本不存在事务挂起的说法的，也不存在这样的接口方法。

因此，记住事务的三个真实存在的方法，不要被各种事务状态名词所迷惑，它们分别是：`conn.setAutoCommit()`、`conn.commit()`、`conn.rollback()`。

`conn.close()`含义为关闭一个数据库连接，这已经不再是事务方法了。

## 1. Mybatis中的事务接口Transaction

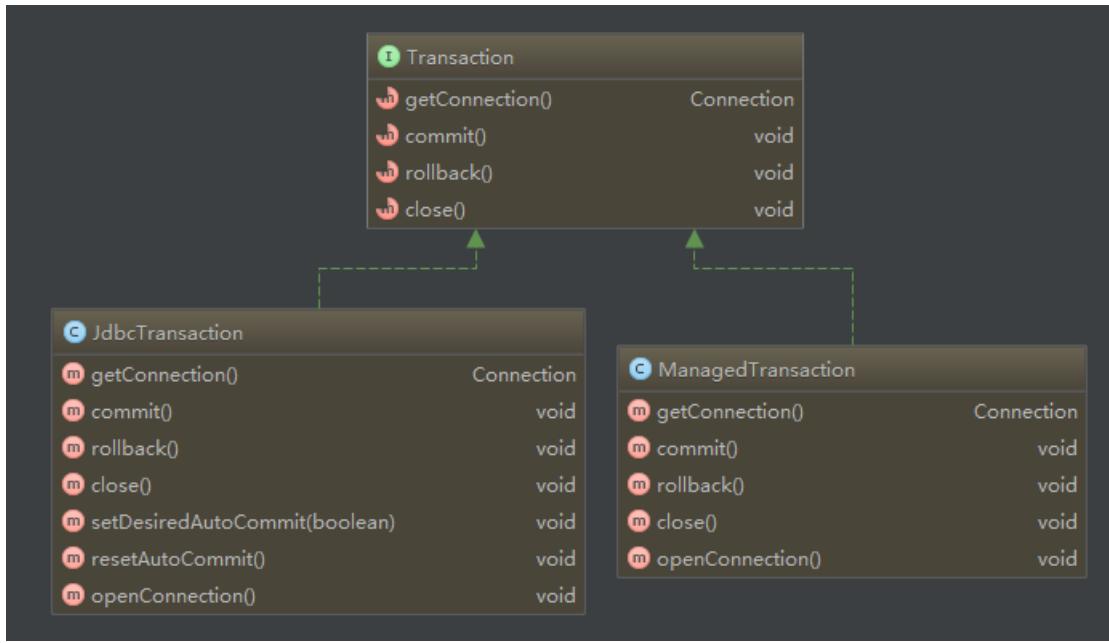
```

public interface Transaction {
    Connection getConnection() throws SQLException ;
    void commit() throws SQLException ;
    void rollback() throws SQLException ;
    void close() throws SQLException ;
}

```

有了文章开头的分析，当你再次看到close()方法时，千万别再认为是关闭一个事务了，而是关闭一个conn连接，或者是把conn连接放回连接池内。

事务类层次结构图：



**JdbcTransaction:** 单独使用Mybatis时，默认的事务管理实现类，就和它的名字一样，它就是我们常说的JDBC事务的极简封装，和编程使用mysql-connector-java-5.1.38-bin.jar事务驱动没啥差别。其极简封装，仅是让connection支持连接池而已。

**ManagedTransaction:** 含义为托管事务，空壳事务管理器，皮包公司。仅是提醒用户，在其它环境中应用时，把事务托管给其它框架，比如托管给Spring，让Spring去管理事务。

org.apache.ibatis.transaction.jdbc.JdbcTransaction.java部分源码。

```

@Override
public void close() throws SQLException {
    if(connection!= null){
        resetAutoCommit();
        if(log.isDebugEnabled()){
            log.debug( "Closing JDBCConnection["+connection+ "]");
        }
        connection.close();
    }
}

```

面对上面这段代码，我们不禁好奇，connection.close()之前，居然调用了一个resetAutoCommit()，含义为重置autoCommit属性值。

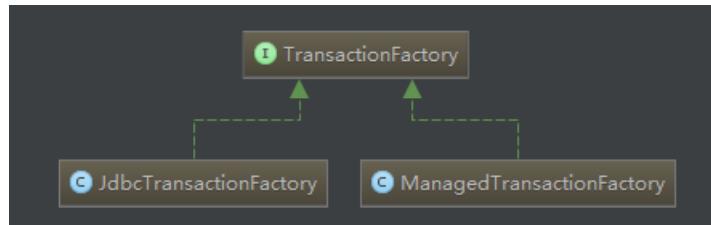
connection.close()含义为销毁conn，既然要销毁conn，为何还多此一举的调用一个resetAutoCommit()呢？消失之前多喝水，真的没有必要。

其实，原因是这样的，connection.close()不意味着真的要销毁conn，而是要把conn放回连接池，供下一次使用，既然还要使

用，自然就需要重置AutoCommit属性了。

通过生成connection代理类，来实现重回连接池的功能。如果connection是普通的Connection实例，那么代码也是没有问题的，双重支持。

## 2. 事务工厂TransactionFactory



顾名思义，一个生产JdbcTransaction实例，一个生产ManagedTransaction实例。两个毫无实际意义的工厂类，除了new之外，没有其他代码。

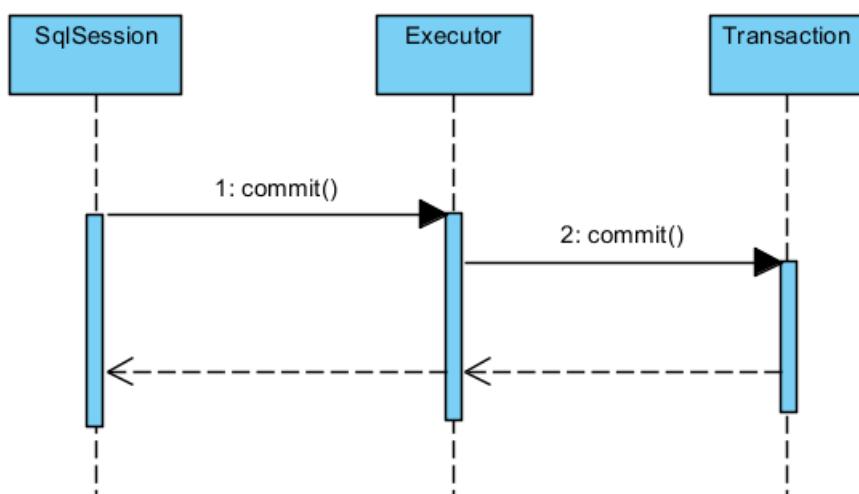
```
<transactionmanagertype="JDBC"/>
```

mybatis-config.xml配置文件内，可配置事务管理类型。

## 3. Transaction的用法

无论是SqlSession，还是Executor，它们的事务方法，最终都指向了Transaction的事务方法，即都是由Transaction来完成事务提交、回滚的。

配一个简单的时序图。



代码样例：

```

public static void main(String[]args){
    SqlSessionsqlSession=MybatisSqlSessionFactory.openSession();
    try{
        StudentMapperstudentMapper=sqlSession.getMapper(StudentMapper.class);

        Studentstudent= newStudent();
        student.setName( "yy");
        student.setEmail( "email@email.com");
        student.setDob( newDate());
        student.setPhone( newPhoneNumber( "123-2568-8947"));

        studentMapper.insertStudent(student);
        sqlSession.commit();
    } catch(Exceptione){
        sqlSession.rollback();
    } finally{
        sqlSession.close();
    }
}

```

注：Executor在执行insertStudent(student)方法时，与事务的提交、回滚、关闭毫无瓜葛（方法内部不会提交、回滚事务），需要像上面的代码一样，手动显示调用commit()、 rollback()、 close()等方法。

因此，后续在分析到类似insert()、 update()等方法内部时，需要忘记事务的存在，不要试图在insert()等方法内部寻找有关事务的任何方法。

## 4. 你可能关心的有关事务的几种特殊场景表现（重要）

### 1. 一个conn生命周期内，可以存在无数多个事务。

```

//执行了connection.setAutoCommit(false), 并返回
SqlSessionsqlSession=MybatisSqlSessionFactory.openSession();
try{
    StudentMapperstudentMapper=sqlSession.getMapper(StudentMapper.class);

    Studentstudent= newStudent();
    student.setName( "yy");
    student.setEmail( "email@email.com");
    student.setDob( new Date());
    student.setPhone( newPhoneNumber( "123-2568-8947"));

    studentMapper.insertStudent(student);
    //提交
    sqlSession.commit();

    studentMapper.insertStudent(student);
    //多次提交
    sqlSession.commit();
} catch(Exceptione){
    //回滚, 只能回滚当前未提交的事务
    sqlSession.rollback();
} finally{
    sqlSession.close();
}

```

对于JDBC来说，autoCommit=false时，是自动开启事务的，执行commit()后，该事务结束。

以上代码正常情况下，开启了2个事务，向数据库插入了2条数据。

JDBC中不存在Hibernate中的session的概念，在JDBC中，insert了几次，数据库就会有几条记录，切勿混淆。而rollback()，只能回滚当前未提交的事务。

## 2. autoCommit=false，没有执行commit()，仅执行close()，会发生什么？

```
try{
    studentMapper.insertStudent(student);
}finally{
    sqlSession.close();
}
```

就像上面这样的代码，没有commit()，固执的程序员总是好奇这样的特例。

insert后，close之前，如果数据库的事务隔离级别是read uncommitted，那么，我们可以在数据库中查询到该条记录。

接着执行sqlSession.close()时，经过SqlSession的判断，决定执行rollback()操作，于是，事务回滚，数据库记录消失。

下面，我们看看org.apache.ibatis.session.defaults.DefaultSqlSession.java中的close()方法源码。

```
@Override
public void close() {
    try{
        executor.close(isCommitOrRollbackRequired(false));
        dirty=false;
    } finally{
        ErrorContext.instance().reset();
    }
}
```

事务是否回滚，依靠isCommitOrRollbackRequired(false)方法来判断。

```
private boolean isCommitOrRollbackRequired(boolean force) {
    return(!autoCommit&&dirty)||force;
}
```

在上面的条件判断中，!autoCommit=true（取反当然是true了），force=false，最终是否回滚事务，只有dirty参数了，dirty含义为是否是脏数据。

```

@Override
public int insert(String statement, Object parameter) {
    return update(statement, parameter);
}

@Override
public int update(String statement, Object parameter) {
    try{
        dirty=true;
        MappedStatement ms=configuration.getMappedStatement(statement);
        return executor.update(ms, wrapCollection(parameter));
    } catch(Exception e){
        throwExceptionFactory.wrapException("Error updating database. Cause:" + e, e);
    } finally{
        ErrorContext.instance().reset();
    }
}

```

源码很明确，只要执行update操作，就设置dirty=true。insert、delete最终也是执行update操作。

只有在执行完commit()、 rollback()、 close()等方法后，才会再次设置dirty=false。

```

@Override
public void commit(boolean force) {
    try{
        executor.commit(isCommitOrRollbackRequired(force));
        dirty=false;
    } catch(Exception e){
        throwExceptionFactory.wrapException("Error committing transaction. Cause:" + e, e);
    } finally{
        ErrorContext.instance().reset();
    }
}

```

因此，得出结论：**autoCommit=false，但是没有手动commit，在sqlSession.close()时，Mybatis会将事务进行rollback()操作，然后才执行conn.close()关闭连接，当然数据最终也就没能持久化到数据库中了。**

### 3. autoCommit=false，没有commit，也没有close，会发生什么？

```
studentMapper.insertStudent(student);
```

干脆，就这一句话，即不commit，也不close。

结论：**insert后，jvm结束前，如果事务隔离级别是read uncommitted，我们可以查到该条记录。jvm结束后，事务被rollback()，记录消失。通过断点debug方式，你可以看到效果。**

这说明JDBC驱动实现，已经考虑到这样的特例情况，底层已经有相应的处理机制了。这也超出了我们的探究范围。

但是，一万个屌丝程序员会对你说：Don't do it like this. Go right way。

**警告：请按正确的try-catch-finally编程方式处理事务，若不从，本人概不负责后果。**

**注：无参的openSession()方法，会自动设置autoCommit=false。**

总结：Mybatis的JdbcTransaction，和纯粹的Jdbc事务，几乎没有差别，它仅是扩展支持了连接池的connection。

另外，需要明确，无论你是否手动处理了事务，只要是对数据库进行任何update操作（update、delete、insert），都一定是在事务中进行的，这是数据库的设计规范之一。

粉丝福利：

赠送一台苹果 iPad 过年如何呢？



微信扫描二维码，关注我的公众号

声明：pdf仅供学习使用，一切版权归原创公众号所有；建议持续关注原创公众号获取最新文章，学习愉快！

# MyBatis 代码生成器配置详解（IDEA）

阿进的写字台 Java后端 1月18日

点击上方 Java后端, 选择 **设为星标**

优质文章, 及时送达

作者 | 阿进的写字台

链接 | [www.cnblogs.com/homejim/p/9782403.html](http://www.cnblogs.com/homejim/p/9782403.html)

在使用 **mybatis** 过程中, 当手写 **JavaBean**和**XML** 写的越来越多的时候, 就越来越容易出错。这种重复性的工作, 我们当然不希望做那么多。

还好, **mybatis** 为我们提供了强大的代码生成--**MybatisGenerator**。

通过简单的配置, 我们就可以生成各种类型的实体类, Mapper接口, MapperXML文件, Example对象等。通过这些生成的文件, 我们就可以方便的进行单表进行增删改查的操作。

Tips: 关注微信公众号: Java后端, 获取每日推送。

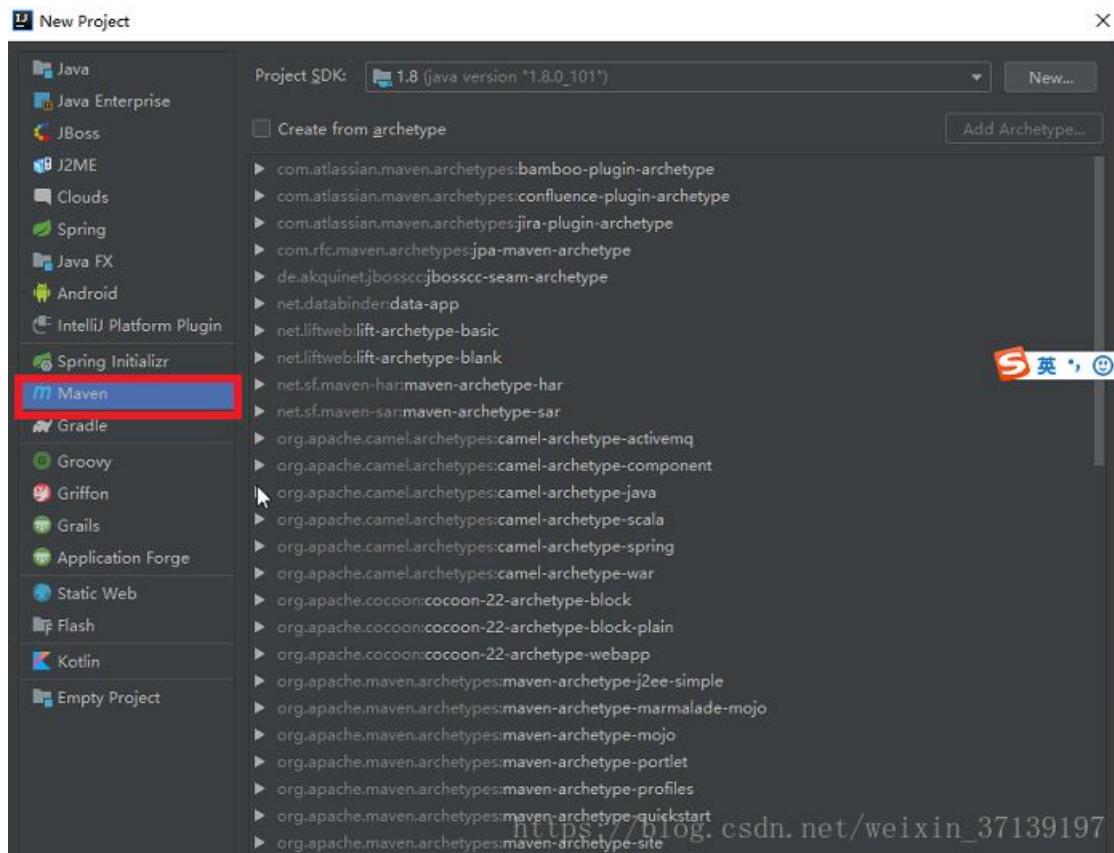
以下的工具使用的都是 **IDEA**

## 1.1 创建Maven项目

### 1.1.1 菜单上选择新建项目

File | New | Project

### 1.1.2 选择左侧的Maven



由于我们只是创建一个普通的项目，此处点击 Next即可。

### 1.1.3 输入GroupId和ArtifactId

- 在我的项目中，

GroupId 填 com.homejim.mybatisplus

ArtifactId 填 mybatis-generator

点击 Next。

### 1.1.4 Finish

通过以上步骤，一个普通的Maven项目就创建好了。

## 1.2 配置 generator.xml

其实名字无所谓，只要跟下面的 **pom.xml** 文件中的对应上就好了。

```

<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE generatorConfiguration PUBLIC
"-//mybatis.org//DTD MyBatis Generator Configuration 1.0//EN"
"http://mybatis.org/dtd/mybatis-generator-config_1_0.dtd" >
<generatorConfiguration>

<classPathEntry location="C:\Users\Administrator\.m2\repository\mysql\mysql-connector-java\8.0.12\mysql-connector-java-8.0.12.jar" />
<context id="context" targetRuntime="MyBatis3">
    <commentGenerator>
        <property name="suppressAllComments" value="false"/>
        <property name="suppressDate" value="true"/>
    </commentGenerator>

    <jdbcConnection
        driverClass="com.mysql.jdbc.Driver"
        connectionURL="jdbc:mysql://localhost:3306/mybatis"
        userId="root"
        password="jim777"/>

    <javaTypeResolver>
        <property name="forceBigDecimals" value="false"/>
    </javaTypeResolver>

    <javaModelGenerator
        targetPackage="com.homejim.mybatis.entity"
        targetProject=".src\main\java">
        <property name="enableSubPackages" value="false"/>
        <property name="trimStrings" value="true"/>
    </javaModelGenerator>

    <sqlMapGenerator
        targetPackage="mybatis/mapper"
        targetProject=".src\main\resources">
        <property name="enableSubPackages" value="false"/>
    </sqlMapGenerator>

    <javaClientGenerator type="XMLMAPPER"
        targetPackage="com.homejim.mybatis.mapper"
        targetProject=".src\main\java">
        <property name="enableSubPackages" value="false"/>
    </javaClientGenerator>

    <table tableName="blog" />
</context>
</generatorConfiguration>

```

需要改一些内容：

1. 本地数据库驱动程序jar包的全路径（**必须要改**）。
2. 数据库的相关配置（**必须要改**）
3. 相关表的配置（**必须要改**）
4. 实体类生成存放的位置。
5. MapperXML 生成文件存放的位置。
6. Mapper 接口存放的位置。

如果不知道怎么改，请看后面的[配置详解](#)。

## 1.3 配置 pom.xml

在原基础上添加一些内容。

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
<modelVersion>4.0.0</modelVersion>

<groupId>com.homejim.mybatisplus</groupId>
<artifactId>mybatis-generator</artifactId>
<version>1.0-SNAPSHOT</version>

<build>
  <finalName>mybatis-generator</finalName>
  <plugins>
    <plugin>
      <groupId>org.mybatis.generator</groupId>
      <artifactId>mybatis-generator-maven-plugin</artifactId>
      <version>1.3.7</version>
      <configuration>

        <configurationFile>src/main/resources/generator.xml</configurationFile>
        <verbose>true</verbose>
        <overwrite>true</overwrite>
      </configuration>
    <executions>
      <execution>
        <id>Generate MyBatis Artifacts</id>
        <goals>
          <goal>generate</goal>
        </goals>
      </execution>
    </executions>
    <dependencies>
      <dependency>
        <groupId>org.mybatis.generator</groupId>
        <artifactId>mybatis-generator-core</artifactId>
        <version>1.3.7</version>
      </dependency>
    </dependencies>
  </plugin>
</plugins>
</build>

</project>
```

需要注意的是 **configurationFile** 中的文件指的是 **generator.xml**。因此路径写的是该文件的相对路径，名称也跟该文件相同。

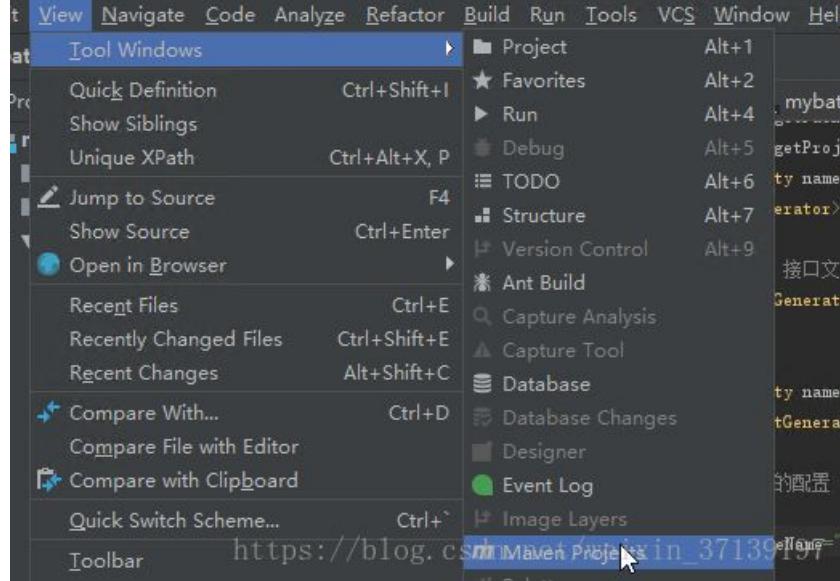
到此， **mybatis-generator** 就可以使用啦。

## 1.4 使用及测试

### 1.4.1 打开 Maven Projects 视图

在 IDEA 上，打开：

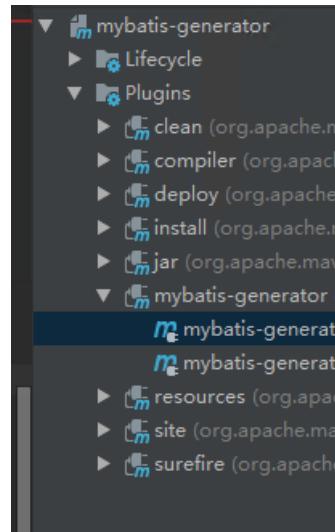
View | Tools | Windwos | Maven Projects



#### 1.4.2 Maven Projects 中双击 mybatis-generator

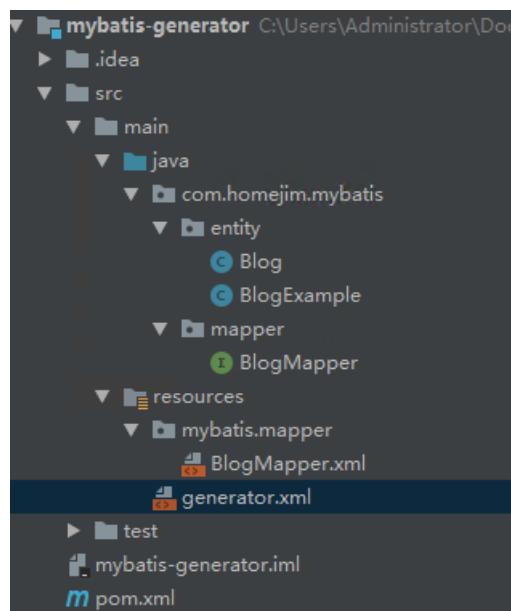
在右侧此时可以看到 Maven Projects 了。找到 mybatis-generator 插件。

mybatis-generator | Plugins | mybatis-generator | mybatis-generator



#### 1.4.3 双击运行

运行正确后，生成代码，得到如下的结构



仅仅是上面那么简单的使用还不够爽。那么我们就可以通过更改 generator.xml 配置文件的方式进行生成的配置。

## 2.1 文档

推荐查看官方的文档。

英文不错的：<http://www.mybatis.org/generator/configreference/xmlconfig.html>

中文翻译版：<http://mbg.cndocs.ml/index.html>

## 2.2 官网没有的

### 2.2.1 property 标签

该标签在官网中只是说用来指定元素的属性，至于怎么用没有详细的讲解。

#### 2.2.1.1 分隔符相关

```
<property name="autoDelimitKeywords" value="true"/>
<property name="beginningDelimiter" value="`"/>
<property name="endingDelimiter" value="`"/>
```

以上的配置对应的是 **mysql**，当数据库中的字段和数据库的关键字一样时，就会使用分隔符。

比如我们的数据列是 delete，按以上的配置后，在它出现的地方，就变成 `delete`。

#### 2.2.1.2 编码

默认是使用当前的系统环境的编码，可以配置为 GBK 或 UTF-8。

```
<property name="javaFileEncoding" value="UTF-8"/>
```

我想项目为 UTF-8，如果指定生成 GBK，则自动生成的中文就是乱码。

#### 2.2.1.3 格式化

```
<property name="javaFormatter" value="org.mybatis.generator.api.dom.DefaultJavaFormatter"/>

<property name="xmlFormatter" value="org.mybatis.generator.api.dom.DefaultXmlFormatter"/>
```

这些显然都是可以自定义实现的。

### 2.2.2 plugins 标签

plugins 标签用来扩展或修改代码生成器生成的代码。

在生成的 XML 中，是没有 **<cache>** 这个标签的。该标签是配置缓存的。

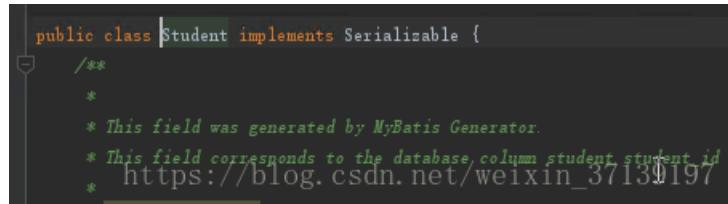
如果我们想生成这个标签，那么可以在 **plugins** 中进行配置。

```
<plugin type="org.mybatis.generator.plugins.CachePlugin">
  <property name="cache_eviction" value="LRU"/>
</plugin>
```

```
<cache eviction="LRU">
<!--
  WARNING - @mbg generated
  This element is automatically generated by MyBatis Generator, do not modify.
-->
  https://blog.csdn.net/weixin_37139197
</cache>
```

比如你想生成的 **JavaBean** 中自行实现 **Serializable** 接口。

```
<plugin type="org.mybatis.generator.plugins.SerializablePlugin" />
```



还能自定义插件。

这些插件都蛮有用的，感觉后续可以专门开一篇文章来讲解。

看名称，就知道是用来生成注释用的。

默认配置：

```
<commentGenerator>
<property name="suppressAllComments" value="false"/>
<property name="suppressDate" value="false"/>
<property name="addRemarkComments" value="false"/>
</commentGenerator>
```

suppressAllComments：阻止生成注释， 默认值是false。

suppressDate：阻止生成的注释包含时间戳， 默认为false。

addRemarkComments：注释中添加数据库的注释， 默认为 false。

还有一个就是我们可以通过 **type** 属性指定我们自定义的注解实现类，生成我们自己想要的注解。自定义的实现类需要实现 **org.mybatis.generator.api.CommentGenerator**。

#### 2.2.4 源码

<https://github.com/homejim/mybatis-cn>

homejim / mybatis-cn

Code Issues 0 Pull requests 0 Projects 0 Wiki Insights Settings

mybatis源码的中文注释以及mybatis的使用和源码解析

mybatis mybatis-sources mybatis-chinese Manage topics

83 commits 1 branch 0 releases 1 contributor View license

Branch: master ▾ New pull request Create new file Upload files Find file Clone or download



微信扫描二维码，关注我的公众号

[阅读原文](#)

声明：pdf仅供学习使用，一切版权归原创公众号所有；建议持续关注原创公众号获取最新文章，学习愉快！

# MyBatis 多数据源读写分离（注解实现）

Java后端 2019-11-09

点击上方 Java后端, 选择 [设为星标](#)

优质文章, 及时送达

作者 | hy\_xiaobin

来源 | [juejin.im/post/5d8705e65188253f4b629f47](https://juejin.im/post/5d8705e65188253f4b629f47)

首先需要建立两个库进行测试，我这里使用的是master\_test和slave\_test两个库，两张库都有一张同样的表（偷懒，嘻嘻），表结构

表名 t\_user

字段名	类型	备注
id	int	主键自增ID
name	varchar	名称

表中分别添加两条不同数据，方便测试 主数据库记录name为xiaobin, 从库为xiaoliu 开始使用Springboot 整合mybatis, 首先引入pom文件

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
<modelVersion>4.0.0</modelVersion>
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.1.4.RELEASE</version>
</parent>
<groupId>com.xiaobin</groupId>
<artifactId>mysql_master_slave</artifactId>
<version>1.0-SNAPSHOT</version>

<properties>
  <java.version>1.8</java.version>
  <lombok.version>1.18.6</lombok.version>
  <mybatis.version>1.3.2</mybatis.version>
  <lombok.version>1.18.6</lombok.version>
</properties>

<dependencies>
  <!-- 添加web启动坐标 -->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
  <!-- 添加lombok工具坐标 -->
  <dependency>
    <groupId>org.projectlombok</groupId>
```

```
        <groupId>org.springframework.boot</groupId>
        <artifactId>lombok</artifactId>
        <version>${lombok.version}</version>
    </dependency>
    <!-- 添加springboot 测试坐标 -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
    </dependency>
    <!-- 添加lombok 测试坐标 -->
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <version>${lombok.version}</version>
    </dependency>
    <!-- 添加mybatis依赖坐标 -->
    <dependency>
        <groupId>org.mybatis.spring.boot</groupId>
        <artifactId>mybatis-spring-boot-starter</artifactId>
        <version>${mybatis.version}</version>
    </dependency>
    <!-- 添加mysql驱动器坐标 -->
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-jdbc</artifactId>
    </dependency>
    <!-- 添加druid数据源坐标 -->
    <dependency>
        <groupId>com.alibaba</groupId>
        <artifactId>druid-spring-boot-starter</artifactId>
        <version>1.1.10</version>
    </dependency>
    <!-- 添加AOP坐标 -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-aop</artifactId>
    </dependency>
</dependencies>
</project>
```

## 动态数据源配置

这里使用的数据源为druid，实现数据源之间的切换用@DataSource自定义注解，配置Aop进行切换 application.yml 配置文件

```

spring:
  datasource:
    type: com.alibaba.druid.pool.DruidDataSource
    druid:
      xiaobin-master:
        driverClassName: com.mysql.jdbc.Driver
        username: root
        password: root
        url: jdbc:mysql://localhost:3306/master_test?serverTimezone=GMT%2B8&useUnicode=true&characterEncoding=utf8
      xiaobin-slave:
        driverClassName: com.mysql.jdbc.Driver
        username: root
        password: root
        url: jdbc:mysql://localhost:3306/slave_test?serverTimezone=GMT%2B8&useUnicode=true&characterEncoding=utf8
  mybatis:
    mapper-locations: classpath:mapper/*.xml

```

多数据源配置类

```

@Configuration
@Component
public class DynamicDataSourceConfig {

  @Bean
  @ConfigurationProperties("spring.datasource.druid.xiaobin-master")
  public DataSource xiaobinMasterDataSource(){
    return DruidDataSourceBuilder.create().build();
  }

  @Bean
  @ConfigurationProperties("spring.datasource.druid.xiaobin-slave")
  public DataSource xiaobinSlaveDataSource(){
    return DruidDataSourceBuilder.create().build();
  }

  @Bean
  @Primary
  public DynamicDataSource dataSource(DataSource xiaobinMasterDataSource, DataSource xiaobinSlaveDataSource) {
    Map<Object, Object> targetDataSources = new HashMap<>();
    targetDataSources.put("xiaobin-master", xiaobinMasterDataSource);
    targetDataSources.put("xiaobin-slave", xiaobinSlaveDataSource);
    return new DynamicDataSource(xiaobinMasterDataSource, targetDataSources);
  }

}

```

动态数据源切换类

```

public class DynamicDataSource extends AbstractRoutingDataSource {

    private static final ThreadLocal<String> contextHolder = new ThreadLocal<>();

    public DynamicDataSource(DataSource defaultTargetDataSource, Map<Object, Object> targetDataSources) {
        super.setDefaultTargetDataSource(defaultTargetDataSource);
        super.setTargetDataSources(targetDataSources);
        super.afterPropertiesSet();
    }

    @Override
    protected Object determineCurrentLookupKey() {
        return getDataSource();
    }

    public static void setDataSource(String dataSource) {
        contextHolder.set(dataSource);
    }

    public static String getDataSource() {
        return contextHolder.get();
    }

    public static void clearDataSource() {
        contextHolder.remove();
    }
}

```

自定义@DataSource注解

在需要切换数据的Dao添加此注解

```

package com.xiaobin.annotation;

import java.lang.annotation.*;

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface DataSource {
    String name() default "";
}

```

Aop切面类配置

```

@Aspect
@Component
public class DataSourceAspect {

    @Pointcut("@annotation(com.xiaobin.annotation.DataSource)")
    public void dataSourcePointCut() {
    }

    @Around("dataSourcePointCut()")
    public Object around(ProceedingJoinPoint point) throws Throwable {
        MethodSignature signature = (MethodSignature) point.getSignature();
        Method method = signature.getMethod();

        DataSource dataSource = method.getAnnotation(DataSource.class);
        if(dataSource == null){
            DynamicDataSource.setDataSource("xiaobin-master");
        }else {
            DynamicDataSource.setDataSource(dataSource.name());
        }

        try {
            return point.proceed();
        } finally {
            DynamicDataSource.clearDataSource();
        }
    }
}

```

启动配置注解信息，重要（不然运行会报错）

```

@SpringBootApplication(exclude= {DataSourceAutoConfiguration.class})
@MapperScan(basePackages = "com.xiaobin.mapper")
@Import({DynamicDataSourceConfig.class})
public class StartApp {
    public static void main(String[] args) {
        SpringApplication.run(StartApp.class);
    }
}

```

测试controller

```

@RestController
@RequestMapping
public class UserController {

    @Autowired
    private UserMapper userMapper;

    @GetMapping("/{name}/list")
    public List<TUser> list(@PathVariable("name")String name){
        if(name.equals("master")){
            return userMapper.queryAllWithMaster();
        }else{
            return userMapper.queryAllWithSlave();
        }
    }
}

```

**效果图**

更具路径传值，进行主从数据源切换

localhost:8080/master/list

GET ▾ localhost:8080/master/list

Params Authorization Headers (9) Body Pre-request Script Tests

Query Params

KEY	VALUE
Key	Value

Body Cookies (1) Headers (3) Test Results

Pretty Raw Preview JSON ▾

```
1 [  
2 {  
3   "id": 1,  
4   "name": "xiaobin"  
5 }  
6 ]
```

localhost:8080/slave/list

GET ▾ localhost:8080/slave/list

Params Authorization Headers (9) Body Pre-request Script Tests

Query Params

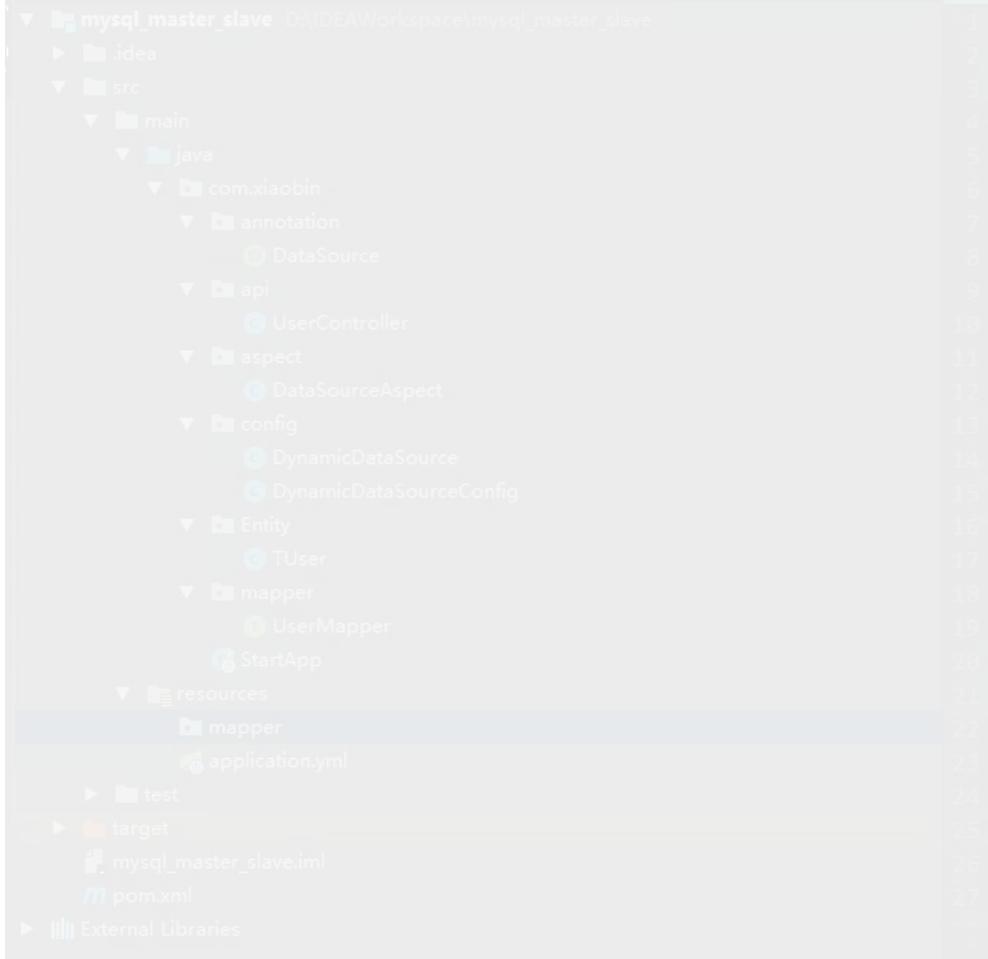
KEY	VALUE
Key	Value

Body Cookies (1) Headers (3) Test Results

Pretty Raw Preview JSON ▾

```
1 [  
2 {  
3   "id": 1,  
4   "name": "xiaoliu"  
5 }  
6 ]
```

目录结构



源码地址（数据库需要自己创建）

[https://gitee.com/MyXiaoXiaoBin/learning-to-share/tree/master/mysql\\_master\\_slave](https://gitee.com/MyXiaoXiaoBin/learning-to-share/tree/master/mysql_master_slave)

- E N D -



学Java,请关注公众号:Java后端

喜欢文章,点个在看

阅读原文

声明：pdf仅供学习使用，一切版权归原创公众号所有；建议持续关注原创公众号获取最新文章，学习愉快！

# MyBatis大揭秘：Plugin 插件设计原理

Java后端 2019-12-02

点击上方 Java后端, 选择 [设为星标](#)

优质文章, 及时送达

作者 | 祖大俊

链接 | [my.oschina.net/zudajun/blog/738973](http://my.oschina.net/zudajun/blog/738973)

大多数框架，都支持插件，用户可通过编写插件来自行扩展功能，Mybatis也不例外。

我们从插件配置、插件编写、插件运行原理、插件注册与执行拦截的时机、初始化插件、分页插件的原理等六个方面展开阐述。

## 1. 插件配置

Mybatis的插件配置在configuration内部，初始化时，会读取这些插件，保存于Configuration对象的InterceptorChain中。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE configuration PUBLIC "-//mybatis.org//DTD Config 3.0//EN" "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
  <plugins>
    <plugin interceptor="com.mybatis3.interceptor.MyBatisInterceptor">
      <property name="value" value="100" />
    </plugin>
  </plugins>
</configuration>

public class Configuration {
  protected final InterceptorChain interceptorChain = new InterceptorChain();
}
```

org.apache.ibatis.plugin.InterceptorChain.java源码。

```
public class InterceptorChain {

  private final List<Interceptor> interceptors = new ArrayList<Interceptor>();

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

  public void addInterceptor(Interceptor interceptor) {
    interceptors.add(interceptor);
  }

  public List<Interceptor> getInterceptors() {
    return Collections.unmodifiableList(interceptors);
  }
}
```

上面的for循环代表了只要是插件，都会以责任链的方式逐一执行（别指望它能跳过某个节点），所谓插件，其实就类似于拦截

## 2. 如何编写一个插件

插件必须实现org.apache.ibatis.plugin.Interceptor接口。

```
public interface Interceptor {
    Object intercept(Invocation invocation) throws Throwable;
    Object plugin(Object target);
    void setProperties(Properties properties);
}
```

- **intercept()方法：**执行拦截内容的地方，比如想收点保护费。由plugin()方法触发，interceptor.plugin(target)足以证明。
- **plugin()方法：**决定是否触发intercept()方法。
- **setProperties()方法：**给自定义的拦截器传递xml配置的属性参数。

下面自定义一个拦截器：

```
@Intercepts({
    @Signature(type = Executor.class, method = "query", args = { MappedStatement.class, Object.class,
        RowBounds.class, ResultHandler.class }),
    @Signature(type = Executor.class, method = "close", args = { boolean.class }) })
public class MyBatisInterceptor implements Interceptor {

    private Integer value;

    @Override
    public Object intercept(Invocation invocation) throws Throwable {
        return invocation.proceed();
    }

    @Override
    public Object plugin(Object target) {
        System.out.println(value);
        // Plugin类是插件的核心类，用于给target创建一个JDK的动态代理对象，触发intercept()方法
        return Plugin.wrap(target, this);
    }

    @Override
    public void setProperties(Properties properties) {
        value = Integer.valueOf((String) properties.get("value"));
    }
}
```

面对上面的代码，我们需要解决两个疑问：

### 1. 为什么要写Annotation注解？注解都是什么含义？

答：Mybatis规定插件必须编写Annotation注解，是必须，而不是可选。

@Intercepts注解：装载一个@Signature列表，一个@Signature其实就是一个需要拦截的方法封装。那么，一个拦截器要拦截

多个方法，自然就是一个@Signature列表。

```
type = Executor.class, method = "query", args = { MappedStatement.class, Object.class, RowBounds.class,
ResultHandler.class }
```

解释：要拦截Executor接口内的query()方法，参数类型为args列表。

## 2. Plugin.wrap(target, this)是什么的？

答：使用JDK的动态代理，给target对象创建一个delegate代理对象，以此来实现方法拦截和增强功能，它会回调intercept()方法。

org.apache.ibatis.plugin.Plugin.java源码：

```
public class Plugin implements InvocationHandler {

    private Object target;
    private Interceptor interceptor;
    private Map<Class<?>, Set<Method>> signatureMap;

    private Plugin(Object target, Interceptor interceptor, Map<Class<?>, Set<Method>> signatureMap) {
        this.target = target;
        this.interceptor = interceptor;
        this.signatureMap = signatureMap;
    }

    public static Object wrap(Object target, Interceptor interceptor) {
        Map<Class<?>, Set<Method>> signatureMap = getSignatureMap(interceptor);
        Class<?> type = target.getClass();
        Class<?>[] interfaces = getAllInterfaces(type, signatureMap);
        if (interfaces.length > 0) {
            // 创建JDK动态代理对象
            return Proxy.newProxyInstance(
                type.getClassLoader(),
                interfaces,
                new Plugin(target, interceptor, signatureMap));
        }
        return target;
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        try {
            Set<Method> methods = signatureMap.get(method.getDeclaringClass());
            // 判断是否是需要拦截的方法(很重要)
            if (methods != null && methods.contains(method)) {
                // 调用intercept()方法
                return interceptor.intercept(new Invocation(target, method, args));
            }
            return method.invoke(target, args);
        } catch (Exception e) {
            throw ExceptionUtil.unwrapThrowable(e);
        }
    }
}
```

Map<Class<?>, Set<Method>> signatureMap：缓存需拦截对象的反射结果，避免多次反射，即target的反射结果。

所以，我们不要动不动就说反射性能很差，那是因为你没有像Mybatis一样去缓存一个对象的反射结果。

判断是否是需要拦截的方法，这句注释很重要，一旦忽略了，都不知道Mybatis是怎么判断是否执行拦截内容的，要记住。

### 3. Mybatis可以拦截哪些接口对象？

```
public class Configuration {  
    //...  
    public ParameterHandler newParameterHandler(MappedStatement mappedStatement, Object parameterObject, BoundSql boundSql) {  
        ParameterHandler parameterHandler = mappedStatement.getLang().createParameterHandler(mappedStatement, parameterObject, boundSql);  
        parameterHandler = (ParameterHandler) interceptorChain.pluginAll(parameterHandler); // 1  
        return parameterHandler;  
    }  
  
    public ResultSetHandler newResultSetHandler(Executor executor, MappedStatement mappedStatement, RowBounds rowBounds, ParameterHandler resultHandler, BoundSql boundSql) {  
        ResultSetHandler resultSetHandler = new DefaultResultSetHandler(executor, mappedStatement, parameterHandler, resultHandler, boundSql);  
        resultSetHandler = (ResultSetHandler) interceptorChain.pluginAll(resultSetHandler); // 2  
        return resultSetHandler;  
    }  
  
    public StatementHandler newStatementHandler(Executor executor, MappedStatement mappedStatement, Object parameterObject, RowBounds rowBounds, ResultHandler statementHandler) {  
        StatementHandler statementHandler = new RoutingStatementHandler(executor, mappedStatement, parameterObject, rowBounds, resultHandler);  
        statementHandler = (StatementHandler) interceptorChain.pluginAll(statementHandler); // 3  
        return statementHandler;  
    }  
  
    public Executor newExecutor(Transaction transaction) {  
        return newExecutor(transaction, defaultExecutorType);  
    }  
  
    public Executor newExecutor(Transaction transaction, ExecutorType executorType) {  
        executorType = executorType == null ? defaultExecutorType : executorType;  
        executorType = executorType == null ? ExecutorType.SIMPLE : executorType;  
        Executor executor;  
        if (ExecutorType.BATCH == executorType) {  
            executor = new BatchExecutor(this, transaction);  
        } else if (ExecutorType.REUSE == executorType) {  
            executor = new ReuseExecutor(this, transaction);  
        } else {  
            executor = new SimpleExecutor(this, transaction);  
        }  
        if (cacheEnabled) {  
            executor = new CachingExecutor(executor);  
        }  
        executor = (Executor) interceptorChain.pluginAll(executor); // 4  
        return executor;  
    }  
    //...  
}
```

Mybatis只能拦截ParameterHandler、ResultSetHandler、StatementHandler、Executor共4个接口对象内的方法。

**重新审视interceptorChain.pluginAll()方法：**该方法在创建上述4个接口对象时调用，其含义为给这些接口对象注册拦截器功能，注意是注册，而不是执行拦截。

**拦截器执行时机：**plugin()方法注册拦截器后，那么，在执行上述4个接口对象内的具体方法时，就会自动触发拦截器的执行，也就是插件的执行。

所以，一定要分清，何时注册，何时执行。切不可认为pluginAll()或plugin()就是执行，它只是注册。

## 4. Invocation

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

`intercept(Invocation invocation)` 方法的参数`Invocation`，我相信你一定可以看得懂，不解释。

## 5. 初始化插件源码解析

`org.apache.ibatis.builder.xml.XMLConfigBuilder.parseConfiguration(XNode)` 方法部分源码。

```
pluginElement(root.evalNode("plugins"));  
  
private void pluginElement(XNode parent) throws Exception {  
    if (parent != null) {  
        for (XNode child : parent.getChildren()) {  
            String interceptor = child.getStringAttribute("interceptor");  
            Properties properties = child.getChildrenAsProperties();  
            Interceptor interceptorInstance = (Interceptor) resolveClass(interceptor).newInstance();  
            // 这里展示了setProperties()方法的调用时机  
            interceptorInstance.setProperties(properties);  
            configuration.addInterceptor(interceptorInstance);  
        }  
    }  
}
```

对于Mybatis，它并不区分是何种拦截器接口，所有的插件都是`Interceptor`，Mybatis完全依靠`Annotation`去标识对谁进行拦截，所以，具备接口一致性。

## 6. 分页插件原理

由于Mybatis采用的是逻辑分页，而非物理分页，那么，市场上就出现了可以实现物理分页的Mybatis的分页插件。

要实现物理分页，就需要对`String sql`进行拦截并增强，Mybatis通过`BoundSql`对象存储`String sql`，而`BoundSql`则由`StatementHandler`对象获取。

```
public interface StatementHandler {  
    <E> List<E> query(Statement statement, ResultHandler resultHandler) throws SQLException;  
  
    BoundSql getBoundSql();  
}  
  
public class BoundSql {  
    public String getSql() {  
        return sql;  
    }  
}
```

因此，就需要编写一个针对`StatementHandler`的`query`方法拦截器，然后获取到`sql`，对`sql`进行重写增强。

任它天高海阔，任它变化无穷，我们只要懂得原理，再多插件，我们都可以对其投送王之蔑视。

如果看到这里，说明你喜欢这篇文章，请[转发、点赞](#)。微信搜索「web\_resource」，关注后回复「进群」或者扫描下方二维码即可进入无广告交流群。

↓ 扫描二维码进群 ↓



#### 推荐阅读

1. 你在公司项目里面看过哪些操蛋的代码？
2. 你知道 Spring Batch 吗？
3. 面试题：Lucene、Solr、ElasticSearch
4. 3 分钟带你彻底搞懂 Java 泛型背后的秘密
5. 团队开发中 Git 最佳实践



喜欢文章，点个在看

# MyBatis 源码：原来 resultMap 解析完是这样

阿进的写字台 Java后端 2019-11-16

点击上方 Java后端, 选择 [设为星标](#)

优质文章, 及时送达

作者 | 阿进的写字台

链接 | [cnblogs.com/homejim/p/9853703.html](http://cnblogs.com/homejim/p/9853703.html)

## 目录

在 **select** 语句中查询得到的是一张二维表, 水平方向上看是一个个字段, 垂直方向上看是一条条记录。

作为面向对象的语言, **Java** 中的对象是根据类定义创建的。类之间的引用关系可以认为是嵌套的关系。

在 mybatis 中, **resultMap** 节点定义了结果集和结果对象 (**JavaBean**) 之间的映射规则。

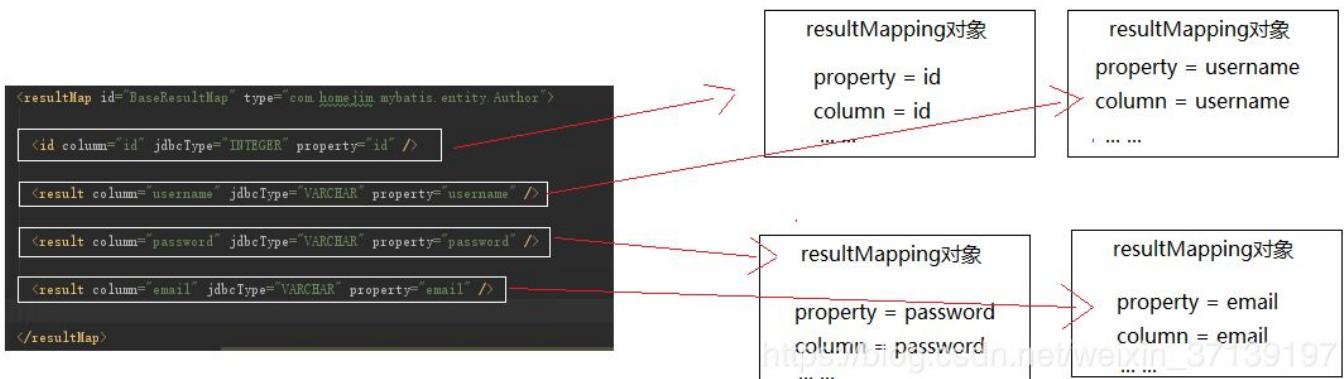
本文主要讲解的是 **resultMap** 的解析。

## 1 两个基础类

在阅读本文之前, 最好能对这两个类有相应的理解。

### 1.1 列映射类ResultMapping

**ResultMapping** 对象记录了结果集中一列与队友**JavaBean**中一个属性的对应关系。



更多详情, 请参考[mybatis百科-列映射类ResultMapping](#)

### 1.2 结果集映射类ResultMap

**ResultMap** 对应的是结果集 **<resultMap>** 中的一个结果集。其基本组成部分中, 含有 **ResultMapping** 对象。

其组成大致如下:



更多详情，请参考mybatis百科-结果集映射类ResultMap

## 2. 解析

### 2.1 入口函数

resultMap 是 mapper.xml 文件下的，因此其是解析 Mapper 的一个环节。

```
resultMapElements(context.evalNodes("/mapper/resultMap"));
```

解析<resultMap>，由于<resultMap>是可以有多个的，因此，`context.evalNodes("/mapper/resultMap")`返回的是一个 List。

```
private void resultMapElements(List<XNode> list) throws Exception {
    // 遍历，解析
    for (XNode resultMapNode : list) {
        try {
            resultMapElement(resultMapNode);
        } catch (IncompleteElementException e) {
            // ignore, it will be retried
        }
    }
}
```

### 2.2 解析流程

整个过程就是 resultMapElement 这个函数。其流程大体如下



对应的代码

```

private ResultMap resultMapElement(XNode resultMapNode) throws Exception {
    return resultMapElement(resultMapNode, Collections.<ResultMapping> emptyList());
}

/**
 * 处理<resultMap>节点，将节点解析成ResultMap对象，下面包含有ResultMapping对象组成的列表
 * @param resultMapNode resultMap 节点
 * @param additionalResultMappings 另外的ResultMapping列表
 * @return ResultMap 对象
 * @throws Exception
 */
private ResultMap resultMapElement(XNode resultMapNode, List<ResultMapping> additionalResultMappings) throws Exception {
    ErrorContext.instance().activity("processing " + resultMapNode.getValueBasedIdentifier());
    // 获取ID，默认值会拼装所有父节点的id或value或property
    String id = resultMapNode.getStringAttribute("id",
        resultMapNode.getValueBasedIdentifier());
    // 获取type属性，表示结果集将被映射为type指定类型的对象
    String type = resultMapNode.getStringAttribute("type",
        resultMapNode.getStringAttribute(" ofType",
            resultMapNode.getStringAttribute("resultType",
                resultMapNode.getStringAttribute("javaType"))));
    // 获取extends属性，其表示结果集的继承
    String extend = resultMapNode.getStringAttribute("extends");
    // 自动映射属性。将列名自动映射为属性
    Boolean autoMapping = resultMapNode.getBooleanAttribute("autoMapping");
    // 解析type，获取其类型
    Class<?> typeClass = resolveClass(type);
    Discriminator discriminator = null;
    // 记录解析的结果
    List<ResultMapping> resultMappings = new ArrayList<>();
    resultMappings.addAll(additionalResultMappings);
    // 处理子节点
    List<XNode> resultChildren = resultMapNode.getChildren();
    for (XNode resultChild : resultChildren) {
        // 处理constructor节点
        if ("constructor".equals(resultChild.getName())) {
            // 解析构造函数元素，其下的每一个子节点都会生产一个ResultMapping对象
            processConstructorElement(resultChild, typeClass, resultMappings);
            // 处理discriminator节点
        } else if ("discriminator".equals(resultChild.getName())) {
            discriminator = processDiscriminatorElement(resultChild, typeClass, resultMappings);
            // 处理其余节点，如id, result, assocation等
        } else {
            List<ResultFlag> flags = new ArrayList<>();
            if ("id".equals(resultChild.getName())) {
                flags.add(ResultFlag.ID);
            }
            // 创建resultMapping对象，并添加到resultMappings中
            resultMappings.add(buildResultMappingFromContext(resultChild, typeClass, flags));
        }
    }
    // 创建ResultMapResolver对象，该对象可以生成ResultMap对象
    ResultMapResolver resultMapResolver = new ResultMapResolver(builderAssistant, id, typeClass, extend, discriminator, resultMappings);
    try {
        return resultMapResolver.resolve();
    } catch (IncompleteElementException e) {
        // 如果无法创建ResultMap对象，则将该结果添加到incompleteResultMaps集合中
        configuration.addIncompleteResultMap(resultMapResolver);
        throw e;
    }
}

```

## 2.3 获取 id

id 对于 resultMap 来说是很重要的， 它是一个身份标识。具有唯一性

```
// 获取 ID, 默认值会拼装所有父节点的 id 或 value 或 property.  
String id = resultMapNode.getStringAttribute("id", resultMapNode.getValueBasedIdentifier());
```

这里涉及到 XNode 对象中的两个函数

```
public String getStringAttribute(String name, String def) {  
    String value = attributes.getProperty(name);  
    if (value == null) {  
        return def;  
    } else {  
        return value;  
    }  
}
```

该函数是获取 XNode 对象对应 XML 节点的 name 属性值， 如果该属性不存在，则返回传入的默认值 def。

而在获取 id 的过程中，默认值是下面这个函数

```
/**  
 * 生成元素节点的基础 id  
 * @return  
 */  
public String getValueBasedIdentifier() {  
    StringBuilder builder = new StringBuilder();  
    XNode current = this;  
    // 当前的节点不为空  
    while (current != null) {  
        // 如果节点不等于 this，则在 0 之前插入 _ 符号，因为是不断的获取父节点的，因此是插在前面  
        if (current != this) {  
            builder.insert(0, "_");  
        }  
        // 获取 id, id 不存在则获取 value, value 不存在则获取 property。  
        String value = current.getStringAttribute("id",  
            current.getStringAttribute("value",  
                current.getStringAttribute("property", null)));  
        // value 非空，则将 . 替换为 _，并将 value 的值加上 []  
        if (value != null) {  
            value = value.replace('.', '_');  
            builder.insert(0, "[" + value);  
            builder.insert(0, "]");  
        }  
        // 不管 value 是否存在，前面都添加上节点的名称  
        builder.insert(0, current.getName());  
        // 获取父节点  
        current = current.getParent();  
    }  
    return builder.toString();  
}
```

该函数是生成元素节点的 id，如果是这样子的 XML。

```
<employee id="${id_var}">
<blah something="that"/>
<first_name>Jim</first_name>
<last_name>Smith</last_name>
<birth_date>
  <year>1970</year>
  <month>6</month>
  <day>15</day>
</birth_date>
<height units="ft">5.8</height>
<weight units="lbs">200</weight>
<active>true</active>
</employee>
```

我们调用

```
XNode node = parser.evalNode("/employee/height");
node.getValueBasedIdentifier();
```

则，返回值应该是

```
employee[id_var]_height
```

## 2.4 解析结果集的类型

结果集的类型，对应的是一个 JavaBean 对象。通过反射来获得该类型。

```
// 获取type, type 不存在则获取 ofType, ofType
// 不存在则获取 resultType, resultType 不存在则获取 javaType
String type = resultMapNode.getStringAttribute("type",
  resultMapNode.getStringAttribute("ofType",
    resultMapNode.getStringAttribute("resultType",
      resultMapNode.getStringAttribute("javaType"))));
// ...
// 获取 type 对应的 Class 对象
Class<?> typeClass = resolveClass(type);
```

看源码，有很多个 **def** 值，也就是说，我们在配置结果集的类型的时候都是有优先级的。但是，这里有一个奇怪的地方，我源代码版本（3.5.0-SNAPSHOT）的 **<resultMap>** 的属性，只有 **type**，没有 **ofType/resultType/javaType**。以下为相应的 DTD 约束：

```
<!ELEMENT resultMap (constructor?,id*,result*,association*,collection*,discriminator?)>
<!ATTLIST resultMap
id CDATA #REQUIRED
type CDATA #REQUIRED
extends CDATA #IMPLIED
autoMapping (true|false) #IMPLIED
>
```

我怀疑是兼容以前的版本。

## 2.5 获取继承结果集和自动映射

```
String extend = resultMapNode.getStringAttribute("extends");
Boolean autoMapping = resultMapNode.getBooleanAttribute("autoMapping");
```

这两个属性都是在配置 XML 的时候可有可无的。

## 2.6 解析的子节点

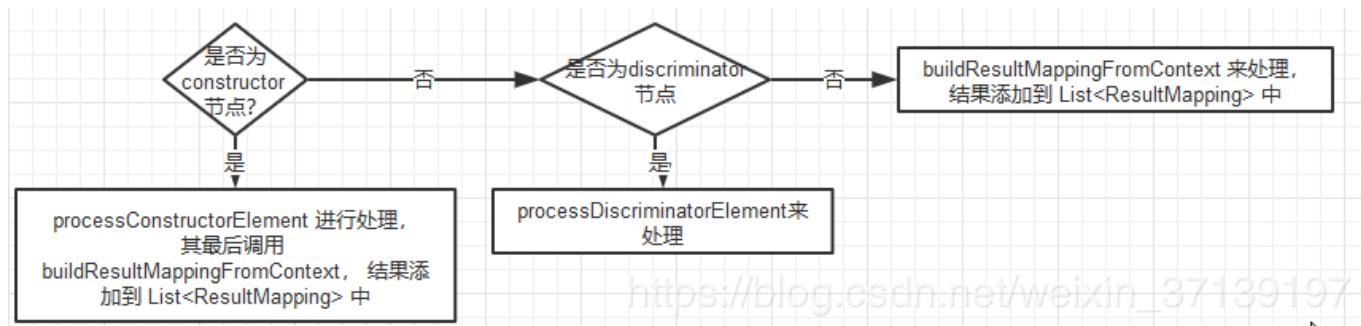
先看 DTD 约束

```
<!ELEMENT resultMap (constructor?,id*,result*,association*,collection*,discriminator?)>
```

可以有以下几个子节点：

子节点	数量
constructor	出现 0 次或 1 次
id	出现 0 次或 多 次
result	出现 0 次或 多 次
association	出现 0 次或 多 次
collection	出现 0 次或 多 次
discriminator	出现 0次或 1 次

子节点解析过程很简单



根据类型进行解析，最后获得 resultMappings (List) ，有可能会获得 discriminator (Discriminator)。

```
// 创建一个 resultMappings 的链表
List<ResultMapping> resultMappings = new ArrayList<>();
// 将从其他地方传入的 additionalResultMappings 添加到该链表中
resultMappings.addAll(additionalResultMappings);
// 获取子节点
List<XNode> resultChildren = resultMapNode.getChildren();
// 遍历解析子节点
for (XNode resultChild : resultChildren) {
    if ("constructor".equals(resultChild.getName())) {
        // 解析构造函数元素，其下的没每一个子节点都会生产一个 ResultMapping 对象
        processConstructorElement(resultChild, typeClass, resultMappings);
    } else if ("discriminator".equals(resultChild.getName())) {
        // 解析 discriminator 节点
        discriminator = processDiscriminatorElement(resultChild, typeClass, resultMappings);
    } else {
        // 解析其余的节点
        List<ResultFlag> flags = new ArrayList<>();
        if ("id".equals(resultChild.getName())) {
            flags.add(ResultFlag.ID);
        }
        resultMappings.add(buildResultMappingFromContext(resultChild, typeClass, flags));
    }
}
```

除了 discriminator 节点，其余节点最后都会回到 buildResultMappingFromContext 方法上，该方法是创建 ResultMapping 对象。

```
/*
 * 获取一行，如result等，取得他们所有的属性，通过这些属性建立ResultMapping对象
 * @param context 对于节点本身
 * @param resultType resultMap 的结果类型
 * @param flags flag 属性，对应ResultFlag 枚举中的属性。一般情况下为空
 * @return 返回ResultMapping
 * @throws Exception
 */
private ResultMapping buildResultMappingFromContext(XNode context, Class<?> resultType, List<ResultFlag> flags) throws Ex
String property;
// 获取节点的属性，如果节点是构造函数（只有name属性，没有property），
// 则获取的是 name, 否则获取 property
if(flags.contains(ResultFlag.CONSTRUCTOR)) {
    property = context.getStringAttribute("name");
} else {
    property = context.getStringAttribute("property");
}
String column = context.getStringAttribute("column");
String javaType = context.getStringAttribute("javaType");
String jdbcType = context.getStringAttribute("jdbcType");
String nestedSelect = context.getStringAttribute("select");
// 获取嵌套的结果集
String nestedResultMap = context.getStringAttribute("resultMap",
    processNestedResultMappings(context, Collections.<ResultMapping> emptyList()));
String notNullColumn = context.getStringAttribute("notNullColumn");
String columnPrefix = context.getStringAttribute("columnPrefix");
String typeHandler = context.getStringAttribute("typeHandler");
String resultSet = context.getStringAttribute("resultSet");
String foreignColumn = context.getStringAttribute("foreignColumn");
boolean lazy = "lazy".equals(context.getStringAttribute("fetchType", configuration.isLazyLoadingEnabled() ? "lazy" : "eager"))

// 以上获取各个属性节点
// 解析javaType, typeHandler, jdbcType
Class<?> javaTypeClass = resolveClass(javaType);
@SuppressWarnings("unchecked")
Class<? extends TypeHandler<?>> typeHandlerClass = (Class<? extends TypeHandler<?>>) resolveClass(typeHandler);
JdbcType jdbcTypeEnum = resolveJdbcType(jdbcType);
// 创建resultMapping对象
return builderAssistant.buildResultMapping(resultType, property, column, javaTypeClass, jdbcTypeEnum, nestedSelect, nestedResultMap)
}
```

如果是 discriminator，则处理该元素并创建鉴别器。

```

/**
 * 处理鉴别器
 * @param context 节点
 * @param resultType 结果类型
 * @param resultMappings 列结果集合
 * @return 鉴别器
 * @throws Exception
 */
private Discriminator processDiscriminatorElement(XNode context, Class<?> resultType, List<ResultMapping> resultMappings)
String column = context.getStringAttribute("column");
String javaType = context.getStringAttribute("javaType");
String jdbcType = context.getStringAttribute("jdbcType");
String typeHandler = context.getStringAttribute("typeHandler");
// 先获取各个属性
// 取得 javaType 对应的类型
Class<?> javaTypeClass = resolveClass(javaType);
// 取得 typeHandler 对应的类型
@SuppressWarnings("unchecked")
Class<? extends TypeHandler<?>> typeHandlerClass = (Class<? extends TypeHandler<?>>) resolveClass(typeHandler);
// 取得 jdbcType 对应的类型
JdbcType jdbcTypeEnum = resolveJdbcType(jdbcType);
// 创建 discriminatorMap，并遍历子节点，以 value->resultMap 的方式放入 discriminatorMap 中
Map<String, String> discriminatorMap = new HashMap<>();
for (XNode caseChild : context.getChildren()) {
    String value = caseChild.getStringAttribute("value");
    String resultMap = caseChild.getStringAttribute("resultMap", processNestedResultMappings(caseChild, resultMappings));
    discriminatorMap.put(value, resultMap);
}
// 创建鉴别器
return builderAssistant.buildDiscriminator(resultType, column, javaTypeClass, jdbcTypeEnum, typeHandlerClass, discriminatorMap);

```

鉴别器内部，也是含有 ResultMapping 的

```

public class Discriminator {

private ResultMapping resultMapping;
private Map<String, String> discriminatorMap;
.....
}

```

## 2.7 创建 ResultMap 对象

在解析完 <resultMap> 的各个属性和子节点之后。创建 ResultMapResolver 对象，通过对象可以生成 ResultMap。

```

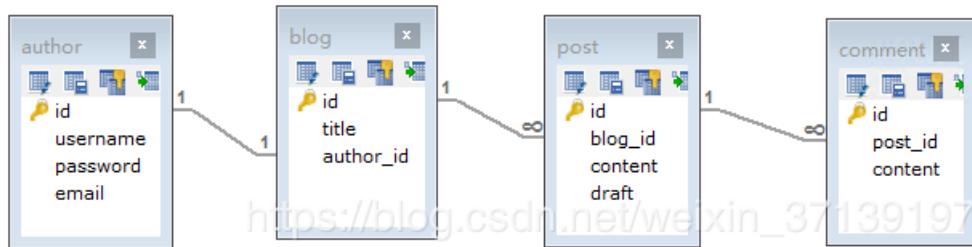
/**
 * 创建并添加 ResultMap 到 Configuration 对象中
 * @param id id, 配置了 id 可以提高效率
 * @param type 类型
 * @param extend 继承
 * @param discriminator 鉴别器
 * @param resultMappings 列集
 * @param autoMapping 是否自动映射
 * @return 返回创建的 ResultMap 对象
 */
public ResultMap addResultMap(
    String id,
    Class<?> type,
    String extend,
    Discriminator discriminator,
    List<ResultMapping> resultMappings,
    Boolean autoMapping) {
    id = applyCurrentNamespace(id, false);
    extend = applyCurrentNamespace(extend, true);

    if (extend != null) {
        if (!configuration.hasResultMap(extend)) {
            throw new IncompleteElementException("Could not find a parent resultmap with id '" + extend + "'");
        }
        // 从 configuration 中获取继承的结果集
        ResultMap resultMap = configuration.getResultMap(extend);
        // 获取所集成结果集的所有 ResultMapping 集合
        List<ResultMapping> extendedResultMappings = new ArrayList<>(resultMap.getResultMappings());
        // 移除需要覆盖的 ResultMapping 集合
        extendedResultMappings.removeAll(resultMappings);
        // 如果该 resultMap 中定义了构造节点, 则移除其父节点的构造器
        boolean declaresConstructor = false;
        for (ResultMapping resultMapping : resultMappings) {
            if (resultMapping.getFlags().contains(ResultFlag.CONSTRUCTOR)) {
                declaresConstructor = true;
                break;
            }
        }
        if (declaresConstructor) {
            Iterator<ResultMapping> extendedResultMappingsIter = extendedResultMappings.iterator();
            while (extendedResultMappingsIter.hasNext()) {
                if (extendedResultMappingsIter.next().getFlags().contains(ResultFlag.CONSTRUCTOR)) {
                    extendedResultMappingsIter.remove();
                }
            }
        }
        // 添加需要被继承的 ResultMapping 集合
        resultMappings.addAll(extendedResultMappings);
    }
    // 通过建造者模式创建 ResultMap 对象
    ResultMap resultMap = new ResultMap.Builder(configuration, id, type, resultMappings, autoMapping)
        .discriminator(discriminator)
        .build();
    // 添加到 Configuration 对象中
    configuration.addResultMap(resultMap);
    return resultMap;
}

```

### 3 解析结果

有如下的数据库表



通过代码生成器生成 XML 和 Mapper。

### 添加结果集

```

<resultMap id="detailedBlogResultMap" type="Blog">
    <!-- 定义映射中使用的构造函数 -->
    <constructor>
        <idArg column="blog_id" javaType="int"/>
    </constructor>
    <!-- 映射普通属性 -->
    <result property="title" column="blog_title"/>
    <!-- 嵌套映射 JavaBean 类型的属性 -->
    <association property="author" resultMap="authorResult"/>
    <collection property="posts" ofType="Post">
        <id property="id" column="post_id"/>
        <result property="content" column="post_content"/>
        <!-- 嵌套查询 -->
        <collection property="comments" column="post_id"
            javaType="ArrayList" ofType="Post" select="selectComment"/>
        <discriminator javaType="int" column="draft">
            <case value="1" resultType="DraftPost"/>
        </discriminator>
    </collection>
</resultMap>

```

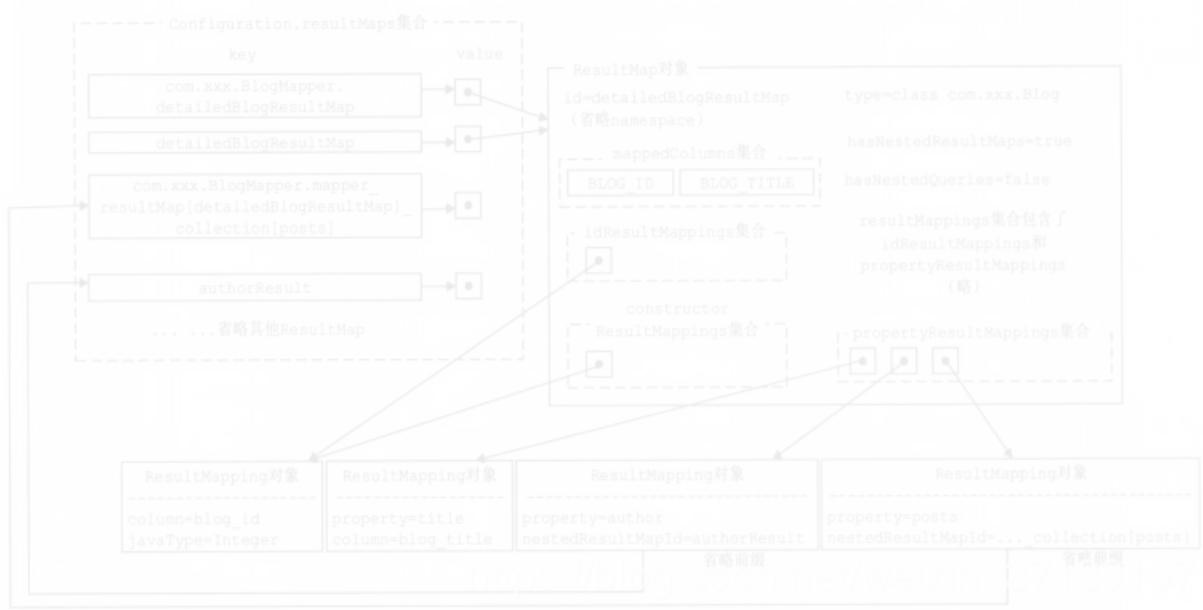
### 对应的 sql

```

<select id="selectBlogDetails" resultMap="detailedBlogResultMap">
    select B.id as blog_id, B.title as blog_title, B.author_id as blog_author_id,
        A.id as author_id, A.username as author_username,
        A.password as author_password, A.email as author_email, P.id as post_id,
        P.blog_id as post_blog_id, P.content as post_content, P.draft as draft
    from Blog B
        left outer join Author A on B.author_id = A.id
        left outer join Post P on B.id = P.blog_id
    where B.id = #{id}
</select>

```

则最后解析出的结果



-END-

如果看到这里，说明你喜欢这篇文章，请转发、点赞。微信搜索「web\_resource」，关注后回复「进群」或者扫描下方二维码即可进入无广告交流群。

↓ 扫描二维码进群 ↓



## 推荐阅读

1. 从零搭建一个 Spring Boot 开发环境
2. Redis 实现「附近的人」这个功能
3. 一个秒杀系统的设计思考
4. 零基础认识 Spring Boot
5. 团队开发中 Git 最佳实践



喜欢文章, 点个在看 

[阅读原文](#)

声明：pdf仅供学习使用，一切版权归原创公众号所有；建议持续关注原创公众号获取最新文章，学习愉快！

# Spring Boot + MyBatis + Druid + PageHelper 实现多数据源并分页

虚无境 Java后端 2019-11-12

点击上方 Java后端, 选择 [设为星标](#)

接收文章, 更加及时

作者 | 虚无境

链接 | [cnblogs.com/xuwujing/p/8964927.html](http://cnblogs.com/xuwujing/p/8964927.html)

## 前言

本篇文章主要讲述的是 Spring Boot整合Mybatis、Druid和PageHelper 并实现多数据源和分页。其中Spring Boot整合Mybatis这块, 在之前的的一篇文章中已经讲述了, 这里就不过多说明了。重点是讲述在多数据源下的如何配置使用Druid和PageHelper。

<http://www.cnblogs.com/xuwujing/p/8260935.html>

## Druid介绍和使用

在使用Druid之前, 先来简单的了解下Druid。

Druid是一个数据库连接池。Druid可以说是目前最好的数据库连接池! 因其优秀功能、性能和扩展性方面, 深受开发人员的青睐。

Druid已经在阿里巴巴部署了超过600个应用, 经过一年多生产环境大规模部署的严苛考验。Druid是阿里巴巴开发的号称“为监控而生”的数据库连接池!

同时Druid不仅仅是一个数据库连接池, Druid 核心主要包括三部分:

- 基于Filter-Chain模式的插件体系。
- DruidDataSource 高效可管理的数据库连接池。
- SQLParser

Druid的主要功能如下:

- 是一个高效、功能强大、可扩展性好的数据库连接池。
- 可以监控数据库访问性能。
- 数据库密码加密
- 获得SQL执行日志
- 扩展JDBC

介绍方面这块就不再多说, 具体的可以看官方文档。那么开始介绍Druid如何使用。

首先是Maven依赖, 只需要添加druid这一个jar就行了。

```
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid</artifactId>
    <version>1.1.8</version>
</dependency>
```

Tips: 可以关注微信公众号: Java后端, 获取Maven教程和每日技术博文推送。

配置方面, 主要的只需要在application.properties或application.yml添加如下就可以了。

说明: 因为这里我是用来两个数据源, 所以稍微有些不同而已。Druid 配置的说明在下面中已经说的很详细了, 这里我就不在说明了。

```
## 默认的数据源
master.datasource.url=jdbc:mysql://localhost:3306/springBoot?useUnicode=true&characterEncoding=utf8&allowMultiQueries=true
master.datasource.username=root
master.datasource.password=123456
master.datasource.driverClassName=com.mysql.jdbc.Driver

## 另一个的数据源
cluster.datasource.url=jdbc:mysql://localhost:3306/springBoot_test?useUnicode=true&characterEncoding=utf8
cluster.datasource.username=root
cluster.datasource.password=123456
cluster.datasource.driverClassName=com.mysql.jdbc.Driver

# 连接池的配置信息
# 初始化大小, 最小, 最大
spring.datasource.type=com.alibaba.druid.pool.DruidDataSource
spring.datasource.initialSize=5
spring.datasource.minIdle=5
spring.datasource.maxActive=20
# 配置获取连接等待超时的时间
spring.datasource.maxWait=60000
# 配置间隔多久才进行一次检测, 检测需要关闭的空闲连接, 单位是毫秒
spring.datasource.timeBetweenEvictionRunsMillis=60000
# 配置一个连接在池中最小生存的时间, 单位是毫秒
spring.datasource.minEvictableIdleTimeMillis=300000
spring.datasource.validationQuery=SELECT 1 FROM DUAL
spring.datasource.testWhileIdle=true
spring.datasource.testOnBorrow=false
spring.datasource.testOnReturn=false
# 打开PSCache, 并且指定每个连接上PSCache的大小
spring.datasource.poolPreparedStatements=true
spring.datasource.maxPoolPreparedStatementPerConnectionSize=20
# 配置监控统计拦截的filters, 去掉后监控界面sql无法统计, 'wall'用于防火墙
spring.datasource.filters=stat,wall,log4j
# 通过connectProperties属性来打开mergeSql功能; 慢SQL记录
spring.datasource.connectionProperties=druid.stat.mergeSql=true;druid.stat.slowSqlMillis=5000
```

成功添加了配置文件之后, 我们再来编写Druid相关的类。

首先是MasterDataSourceConfig.java这个类, 这个是默认的数据源配置类。

```
@Configuration
@MapperScan(basePackages = MasterDataSourceConfig.PACKAGE, sqlSessionFactoryRef = "masterSqlSessionFactory")
public class MasterDataSourceConfig {

    static final String PACKAGE = "com.pancm.dao.master";
    static final String MAPPER_LOCATION = "classpath:mapper/master/*.xml";
```

```
@Value("${master.datasource.url}")
private String url;

@Value("${master.datasource.username}")
private String username;

@Value("${master.datasource.password}")
private String password;

@Value("${master.datasource.driverClassName}")
private String driverClassName;

@Value("${spring.datasource.initialSize}")
private int initialSize;

@Value("${spring.datasource.minIdle}")
private int minIdle;

@Value("${spring.datasource.maxActive}")
private int maxActive;

@Value("${spring.datasource.maxWait}")
private int maxWait;

@Value("${spring.datasource.timeBetweenEvictionRunsMillis}")
private int timeBetweenEvictionRunsMillis;

@Value("${spring.datasource.minEvictableIdleTimeMillis}")
private int minEvictableIdleTimeMillis;

@Value("${spring.datasource.validationQuery}")
private String validationQuery;

@Value("${spring.datasource.testWhileIdle}")
private boolean testWhileIdle;

@Value("${spring.datasource.testOnBorrow}")
private boolean testOnBorrow;

@Value("${spring.datasource.testOnReturn}")
private boolean testOnReturn;

@Value("${spring.datasource.poolPreparedStatements}")
private boolean poolPreparedStatements;

@Value("${spring.datasource.maxPoolPreparedStatementPerConnectionSize}")
private int maxPoolPreparedStatementPerConnectionSize;

@Value("${spring.datasource.filters}")
private String filters;

@Value("{spring.datasource.connectionProperties}")
private String connectionProperties;

@Bean(name = "masterDataSource")
@Primary
public DataSource masterDataSource() {
    DruidDataSource dataSource = new DruidDataSource();
    dataSource.setUrl(url);
    dataSource.setUsername(username);
    dataSource.setPassword(password);
    dataSource.setDriverClassName(driverClassName);
    dataSource.setInitialSize(initialSize);
    dataSource.setMinIdle(minIdle);
    dataSource.setMaxActive(maxActive);
    dataSource.setMaxWait(maxWait);
    dataSource.setTimeBetweenEvictionRunsMillis(timeBetweenEvictionRunsMillis);
    dataSource.setMinEvictableIdleTimeMillis(minEvictableIdleTimeMillis);
    dataSource.setValidationQuery(validationQuery);
    dataSource.setTestWhileIdle(testWhileIdle);
    dataSource.setTestOnBorrow(testOnBorrow);
    dataSource.setTestOnReturn(testOnReturn);
    dataSource.setPoolPreparedStatements(poolPreparedStatements);
    dataSource.setMaxPoolPreparedStatementPerConnectionSize(maxPoolPreparedStatementPerConnectionSize);
    dataSource.setFilters(filters);
    return dataSource;
}
```

```

dataSource.setUsername(username);
dataSource.setPassword(password);
dataSource.setDriverClassName(driverClassName);

//具体配置
dataSource.setInitialSize(initialSize);
dataSource.setMinIdle(minIdle);
dataSource.setMaxActive(maxActive);
dataSource.setMaxWait(maxWait);
dataSource.setTimeBetweenEvictionRunsMillis(timeBetweenEvictionRunsMillis);
dataSource.setMinEvictableIdleTimeMillis(minEvictableIdleTimeMillis);
dataSource.setValidationQuery(validationQuery);
dataSource.setTestWhileIdle(testWhileIdle);
dataSource.setTestOnBorrow(testOnBorrow);
dataSource.setTestOnReturn(testOnReturn);
dataSource.setPoolPreparedStatements(poolPreparedStatements);
dataSource.setMaxPoolPreparedStatementPerConnectionSize(maxPoolPreparedStatementPerConnectionSize);
try {
    dataSource.setFilters(filters);
} catch (SQLException e) {
    e.printStackTrace();
}
dataSource.setConnectionProperties(connectionProperties);
return dataSource;
}

@Bean(name = "masterTransactionManager")
@Primary
public DataSourceTransactionManager masterTransactionManager() {
    return new DataSourceTransactionManager(masterDataSource());
}

@Bean(name = "masterSqlSessionFactory")
@Primary
public SqlSessionFactory masterSqlSessionFactory(@Qualifier("masterDataSource") DataSource masterDataSource)
    throws Exception {
    final SqlSessionFactoryBean sessionFactory = new SqlSessionFactoryBean();
    sessionFactory.setDataSource(masterDataSource);
    sessionFactory.setMapperLocations(new PathMatchingResourcePatternResolver()
        .getResources(MasterDataSourceConfig.MAPPER_LOCATION));
    return sessionFactory.getObject();
}
}

```

其中这两个注解说明下：

- **@Primary** : 标志这个 Bean 如果在多个同类 Bean 候选时, 该 Bean 优先被考虑。多数据源配置的时候注意, 必须要有一个主数据源, 用 @Primary 标志该 Bean。
- **@MapperScan**: 扫描 Mapper 接口并容器管理。

需要注意的是sqlSessionFactoryRef 表示定义一个唯一 SqlSessionFactory 实例。

上面的配置完之后, 就可以将Druid作为连接池使用了。但是Druid并不简简单单的是个连接池, 它也可以说是一个监控应用, 它自带了web监控界面, 可以很清晰的看到SQL相关信息。

在SpringBoot中运用Druid的监控作用, 只需要编写StatViewServlet和WebStatFilter类, 实现注册服务和过滤规则。这里我们可以将这两个写在一起, 使用@Configuration和@Bean。

为了方便理解，相关的配置说明也写在代码中了，这里就不再过多赘述了。

代码如下：

```
@Configuration
public class DruidConfiguration {

    @Bean
    public ServletRegistrationBean druidStatViewServlet() {
        //注册服务
        ServletRegistrationBean servletRegistrationBean = new ServletRegistrationBean(
            new StatViewServlet(), "/druid/*");
        //白名单(为空表示,所有的都可以访问,多个IP的时候用逗号隔开)
        servletRegistrationBean.addInitParameter("allow", "127.0.0.1");
        //IP黑名单(存在共同时, deny优先于allow)
        servletRegistrationBean.addInitParameter("deny", "127.0.0.2");
        //设置登录的用户名和密码
        servletRegistrationBean.addInitParameter("loginUsername", "pancm");
        servletRegistrationBean.addInitParameter("loginPassword", "123456");
        //是否能够重置数据.
        servletRegistrationBean.addInitParameter("resetEnable", "false");
        return servletRegistrationBean;
    }

    @Bean
    public FilterRegistrationBean druidStatFilter() {
        FilterRegistrationBean filterRegistrationBean = new FilterRegistrationBean(
            new WebStatFilter());
        //添加过滤规则
        filterRegistrationBean.addUrlPatterns("/*");
        //添加不需要忽略的格式信息
        filterRegistrationBean.addInitParameter("exclusions",
            "*.js,*.gif,*.jpg,*.png,*.css,*.ico,/druid/*");
        System.out.println("druid初始化成功!");
        return filterRegistrationBean;
    }
}
```

编写完之后，启动程序，在浏览器输入：<http://127.0.0.1:8084/druid/index.html>，然后输入设置的用户名和密码，便可以访问Web界面了。

## 多数据源配置

在进行多数据源配置之前，先分别在springBoot和springBoot\_test的mysql数据库中执行如下脚本。

```
-- springBoot库的脚本
```

```
CREATE TABLE `t_user` (
  `id` int(11) NOT NULL AUTO_INCREMENT COMMENT '自增id',
  `name` varchar(10) DEFAULT NULL COMMENT '姓名',
  `age` int(2) DEFAULT NULL COMMENT '年龄',
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=15 DEFAULT CHARSET=utf8
```

```
-- springBoot_test库的脚本
```

```
CREATE TABLE `t_student` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `name` varchar(16) DEFAULT NULL,
  `age` int(11) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=2 DEFAULT CHARSET=utf8
```

注:为了偷懒,将两张表的结构弄成一样了!不过不影响测试!

在application.properties中已经配置这两个数据源的信息,上面已经贴出了一次配置,这里就不再贴了。

这里重点说下 第二个数据源的配置。和上面的MasterDataSourceConfig.java差不多,区别在于没有使用@Primary注解和名称不同而已。需要注意的是MasterDataSourceConfig.java对package和mapper的扫描是精确到目录的,这里的第二个数据源也是如此。

那么代码如下:

```

@Configuration
@MapperScan(basePackages = ClusterDataSourceConfig.PACKAGE, sqlSessionFactoryRef = "clusterSqlSessionFactory")
public class ClusterDataSourceConfig {

    static final String PACKAGE = "com.pancm.dao.cluster";
    static final String MAPPER_LOCATION = "classpath:mapper/cluster/*.xml";

    @Value("${cluster.datasource.url}")
    private String url;

    @Value("${cluster.datasource.username}")
    private String username;

    @Value("${cluster.datasource.password}")
    private String password;

    @Value("${cluster.datasource.driverClassName}")
    private String driverClass;

    // 和MasterDataSourceConfig一样，这里略

    @Bean(name = "clusterDataSource")
    public DataSource clusterDataSource() {
        DruidDataSource dataSource = new DruidDataSource();
        dataSource.setUrl(url);
        dataSource.setUsername(username);
        dataSource.setPassword(password);
        dataSource.setDriverClassName(driverClass);

        // 和MasterDataSourceConfig一样，这里略 ...
        return dataSource;
    }

    @Bean(name = "clusterTransactionManager")
    public DataSourceTransactionManager clusterTransactionManager() {
        return new DataSourceTransactionManager(clusterDataSource());
    }

    @Bean(name = "clusterSqlSessionFactory")
    public SqlSessionFactory clusterSqlSessionFactory(@Qualifier("clusterDataSource") DataSource clusterDataSource)
        throws Exception {
        final SqlSessionFactoryBean sessionFactory = new SqlSessionFactoryBean();
        sessionFactory.setDataSource(clusterDataSource);
        sessionFactory.setMapperLocations(new PathMatchingResourcePatternResolver().getResources(ClusterDataSourceConfig.M/return sessionFactory.getObject();
    }
}

```

成功写完配置之后，启动程序，进行测试。

分别在springBoot和springBoot\_test库中使用接口进行添加数据。

#### t\_user

```

POST http://localhost:8084/api/user
{"name":"张三","age":25}
{"name":"李四","age":25}
{"name":"王五","age":25}

```

#### t\_student

```
POST http://localhost:8084/api/student
```

```
{"name":"学生A","age":16}  
 {"name":"学生B","age":17}  
 {"name":"学生C","age":18}
```

成功添加数据之后，然后进行调用不同的接口进行查询。

**请求:**

```
GET http://localhost:8084/api/user?name=李四
```

**返回:**

```
{  
    "id": 2,  
    "name": "李四",  
    "age": 25  
}
```

**请求:**

```
GET http://localhost:8084/api/student?name=学生C
```

**返回:**

```
{  
    "id": 1,  
    "name": "学生C",  
    "age": 16  
}
```

通过数据可以看出，成功配置了多数据源了。

## PageHelper 分页实现

PageHelper是Mybatis的一个分页插件，非常的好用！这里强烈推荐！！！

PageHelper的使用很简单，只需要在Maven中添加pagehelper这个依赖就可以了。

Maven的依赖如下：

```
<dependency>  
    <groupId>com.github.pagehelper</groupId>  
    <artifactId>pagehelper-spring-boot-starter</artifactId>  
    <version>1.2.3</version>  
</dependency>
```

注：这里我是用springBoot版的！也可以使用其它版本的。

添加依赖之后，只需要添加如下配置或代码就可以了。

第一种，在application.properties或application.yml添加

```
pagehelper:  
    helperDialect: mysql  
    offsetAspageNum: true  
    rowBoundsWithCount: true  
    reasonable: false
```

第二种，在mybatis.xml配置中添加

```
<bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">  
    <property name="dataSource" ref="dataSource" />  
    <!-- 扫描mapping.xml文件 -->  
    <property name="mapperLocations" value="classpath:mapper/*.xml"></property>  
    <!-- 配置分页插件 -->  
    <property name="plugins">  
        <array>  
            <bean class="com.github.pagehelper.PageHelper">  
                <property name="properties">  
                    <value>  
                        helperDialect=mysql  
                        offsetAspageNum=true  
                        rowBoundsWithCount=true  
                        reasonable=false  
                    </value>  
                </property>  
            </bean>  
        </array>  
    </property>  
</bean>
```

第三种，在代码中添加，使用@Bean注解在启动程序的时候初始化。

```
@Bean  
public PageHelper pageHelper(){  
    PageHelper pageHelper = new PageHelper();  
    Properties properties = new Properties();  
    //数据库  
    properties.setProperty("helperDialect", "mysql");  
    //是否将参数offset作为pageNum使用  
    properties.setProperty("offsetAspageNum", "true");  
    //是否进行count查询  
    properties.setProperty("rowBoundsWithCount", "true");  
    //是否分页合理化  
    properties.setProperty("reasonable", "false");  
    pageHelper.setProperties(properties);  
}
```

因为这里我们使用的是多数据源，所以这里的配置稍微有些不同。我们需要在sessionFactory这里配置。这里就对MasterDataSourceConfig.java进行相应的修改。

在masterSqlSessionFactory方法中，添加如下代码。

```

@Bean(name = "masterSqlSessionFactory")
@Primary
public SqlSessionFactory masterSqlSessionFactory(@Qualifier("masterDataSource") DataSource masterDataSource)
    throws Exception {
    final SqlSessionFactoryBean sessionFactory = new SqlSessionFactoryBean();
    sessionFactory.setDataSource(masterDataSource);
    sessionFactory.setMapperLocations(new PathMatchingResourcePatternResolver()
        .getResources(MasterDataSourceConfig.MAPPER_LOCATION));
    //分页插件
    Interceptor interceptor = new PageInterceptor();
    Properties properties = new Properties();
    //数据库
    properties.setProperty("helperDialect", "mysql");
    //是否将参数offset作为PageNum使用
    properties.setProperty("offsetAsPageNum", "true");
    //是否进行count查询
    properties.setProperty("rowBoundsWithCount", "true");
    //是否分页合理化
    properties.setProperty("reasonable", "false");
    interceptor.setProperties(properties);
    sessionFactory.setPlugins(new Interceptor[] {interceptor});

    return sessionFactory.getObject();
}

```

注：其它的数据源也想进行分页的时候，参照上面的代码即可。

这里需要注意的是reasonable参数，表示分页合理化，默认值为false。如果该参数设置为true时，pageNum<=0时会查询第一页，pageNum>pages(超过总数时)，会查询最后一页。默认false时，直接根据参数进行查询。

设置完PageHelper之后，使用的话，只需要在查询的sql前面添加PageHelper.startPage(pageNum,pageSize)；如果是想知道总数的话，在查询的sql语句后买呢添加page.getTotal()就可以了。

#### 代码示例：

```

public List<T> findByListEntity(T entity) {
    List<T> list = null;
    try {
        Page<?> page = PageHelper.startPage(1,2);
        System.out.println(getClass().getName()+"设置第一页两条数据!");
        list = getMapper().findByListEntity(entity);
        System.out.println("总共有:"+page.getTotal()+"条数据,实际返回:"+list.size()+"两条数据!");
    } catch (Exception e) {
        logger.error("查询"+getClass().getName()+"失败!原因是:",e);
    }
    return list;
}

```

代码编写完毕之后，开始进行最后的测试。

查询t\_user表的所有数据，并进行分页。

#### 请求：

```
GET http://localhost:8084/api/user
```

#### 返回：

```
[  
 {  
   "id": 1,  
   "name": "张三",  
   "age": 25  
 },  
 {  
   "id": 2,  
   "name": "李四",  
   "age": 25  
 }  
]
```

#### 控制台打印:

```
开始查询...  
User设置第一页两条数据!  
2018-04-27 19:55:50.769 DEBUG 6152 --- [io-8084-exec-10] c.p.d.m.UserDao.findByListEntity_COUNT :==> Preparing: SELECT cou  
2018-04-27 19:55:50.770 DEBUG 6152 --- [io-8084-exec-10] c.p.d.m.UserDao.findByListEntity_COUNT :==> Parameters:  
2018-04-27 19:55:50.771 DEBUG 6152 --- [io-8084-exec-10] c.p.d.m.UserDao.findByListEntity_COUNT :<= Total: 1  
2018-04-27 19:55:50.772 DEBUG 6152 --- [io-8084-exec-10] c.p.dao.master.UserDao.findByListEntity :==> Preparing: select id, nar  
2018-04-27 19:55:50.773 DEBUG 6152 --- [io-8084-exec-10] c.p.dao.master.UserDao.findByListEntity :==> Parameters: 2(Integer)  
2018-04-27 19:55:50.774 DEBUG 6152 --- [io-8084-exec-10] c.p.dao.master.UserDao.findByListEntity :<= Total: 2  
总共有:3条数据,实际返回:2两条数据!
```

查询t\_student表的所有的数据，并进行分页。

#### 请求:

```
GET http://localhost:8084/api/student
```

#### 返回:

```
[  
 {  
   "id": 1,  
   "name": "学生A",  
   "age": 16  
 },  
 {  
   "id": 2,  
   "name": "学生B",  
   "age": 17  
 }  
]
```

#### 控制台打印:

```
开始查询...  
Studnet设置第一页两条数据!  
2018-04-27 19:54:56.155 DEBUG 6152 --- [nio-8084-exec-8] c.p.d.c.S.findByListEntity_COUNT :==> Preparing: SELECT count(0) FR  
2018-04-27 19:54:56.155 DEBUG 6152 --- [nio-8084-exec-8] c.p.d.c.S.findByListEntity_COUNT :==> Parameters:  
2018-04-27 19:54:56.156 DEBUG 6152 --- [nio-8084-exec-8] c.p.d.c.S.findByListEntity_COUNT :<= Total: 1  
2018-04-27 19:54:56.157 DEBUG 6152 --- [nio-8084-exec-8] c.p.d.c.StudentDao.findByListEntity :==> Preparing: select id, name, a  
2018-04-27 19:54:56.157 DEBUG 6152 --- [nio-8084-exec-8] c.p.d.c.StudentDao.findByListEntity :==> Parameters: 2(Integer)  
2018-04-27 19:54:56.157 DEBUG 6152 --- [nio-8084-exec-8] c.p.d.c.StudentDao.findByListEntity :<= Total: 2  
总共有:3条数据,实际返回:2两条数据!
```

查询完毕之后，我们再来看Druid 的监控界面。

在浏览器输入:<http://127.0.0.1:8084/druid/index.html>

The screenshot shows the 'SQL Stat View JSON API' section of the Druid Monitor. It displays a table of executed SQL statements with the following columns:

N	SQL	执行数	执行时间	慢慢	事务执行	错误数	更新行数	试做行数	执行中	最大并发	执行时间分布	执行+RS时间分布	读取行分布	更新行分布
1	SELECT id, name, ag...	1					2		1	[1.0.0.0.0.0.0]	[1.0.0.0.0.0.0]	[0,1.0.0.0]	[1.0.0.0.0]	
2	select id, name, ag...	2					4		1	[2.0.0.0.0.0.0]	[2.0.0.0.0.0.0]	[0,2.0.0.0]	[2.0.0.0.0]	
3	SELECT count(0) FROM t_us...	1					1		1	[1.0.0.0.0.0.0]	[1.0.0.0.0.0.0]	[0,1.0.0.0]	[1.0.0.0.0]	
4	SELECT count(0) FROM t_st...	2	1	1			2		1	[1.1.0.0.0.0.0]	[2.0.0.0.0.0.0]	[0,2.0.0.0]	[2.0.0.0.0]	

可以很清晰的看到操作记录！

如果想知道更多的Druid相关知识，可以查看官方文档！

## 结语

这篇终于写完了，在进行代码编写的时候，碰到过很多问题，然后慢慢的尝试和找资料解决了。本篇文章只是很浅的介绍了这些相关的使用，在实际的应用可能会更复杂。如果有有更好的想法和建议，欢迎留言进行讨论！

参考文章：

<https://www.bysocket.com/?p=1712>

Durid官方地址：

<https://github.com/alibaba/druid>

PageHelper官方地址：

<https://github.com/pagehelper/Mybatis-PageHelper>

文中源码：

<https://github.com/xuwujing/springBoot>

- END -



喜欢文章, 点个在看 

声明：pdf仅供学习使用，一切版权归原创公众号所有；建议持续关注原创公众号获取最新文章，学习愉快！

# Spring Boot + MyBatis 多模块项目搭建教程

枫本非凡 Java后端 2019-09-30

点击上方Java后端, 选择“[设为星标](#)”

优质文章, 及时送达

作者 | 枫本非凡

链接 | [cnblogs.com/orzlin/p/9717399.html](http://cnblogs.com/orzlin/p/9717399.html)

上篇 | IDEA 远程一键部署 Spring Boot 到 Docker

## 一、前言

最近公司项目准备开始重构，框架选定为SpringBoot+Mybatis，本篇主要记录了在IDEA中搭建SpringBoot多模块项目的过程。

## 1、开发工具及系统环境

- IDE:

IntelliJ IDEA 2018.2

- 系统环境:

mac OSX

## 2、项目目录结构

- biz层:

业务逻辑层

- dao层:

数据持久层

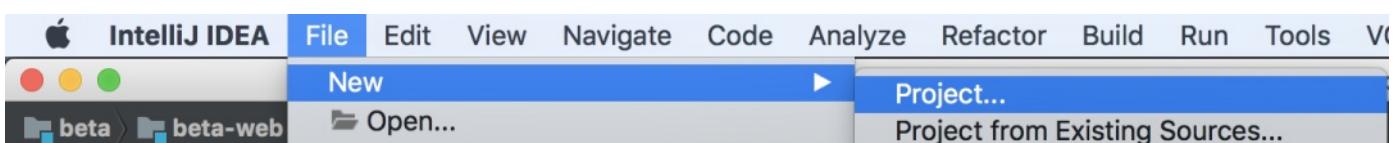
- web层:

请求处理层

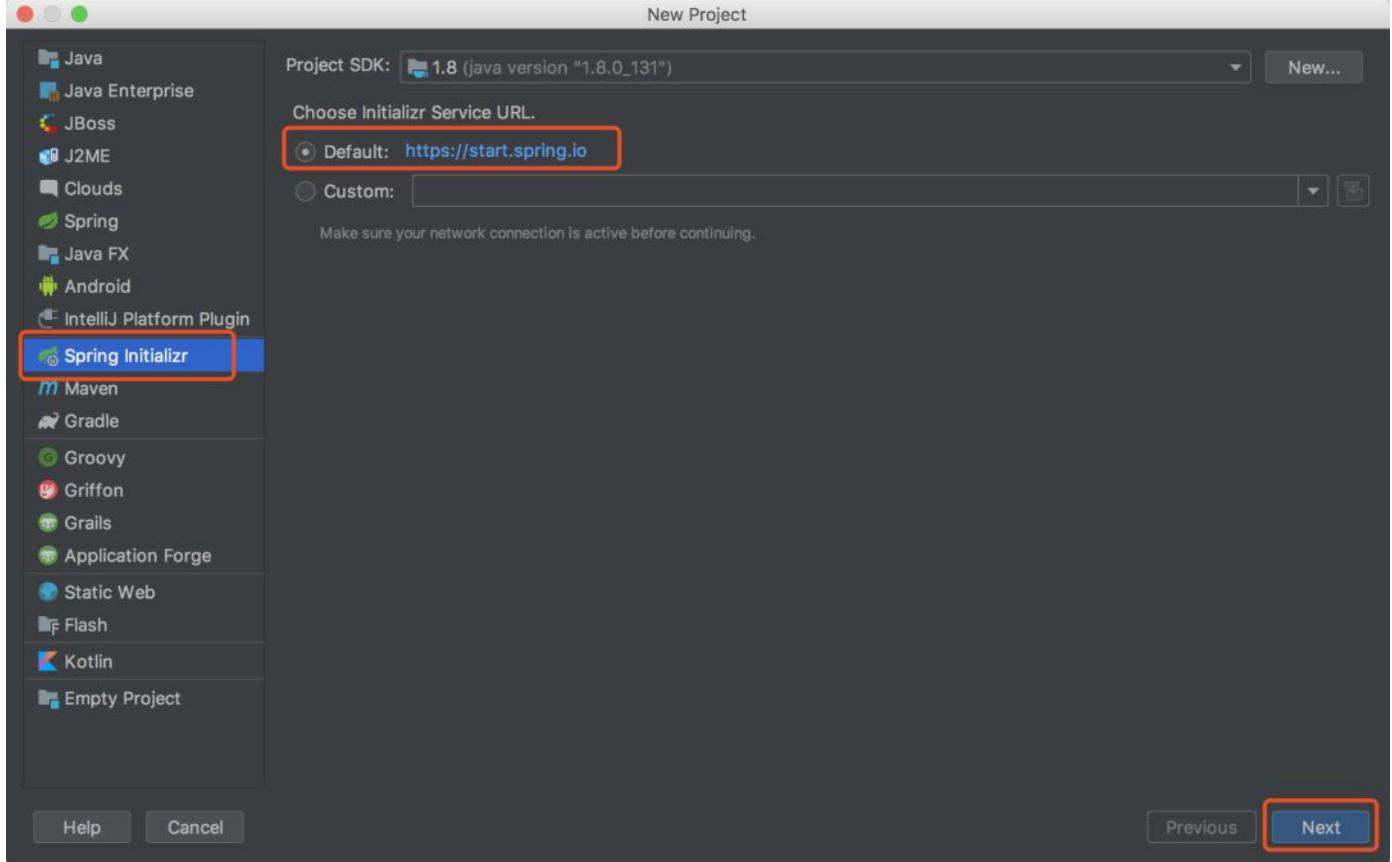
## 二、搭建步骤

### 1、创建父工程

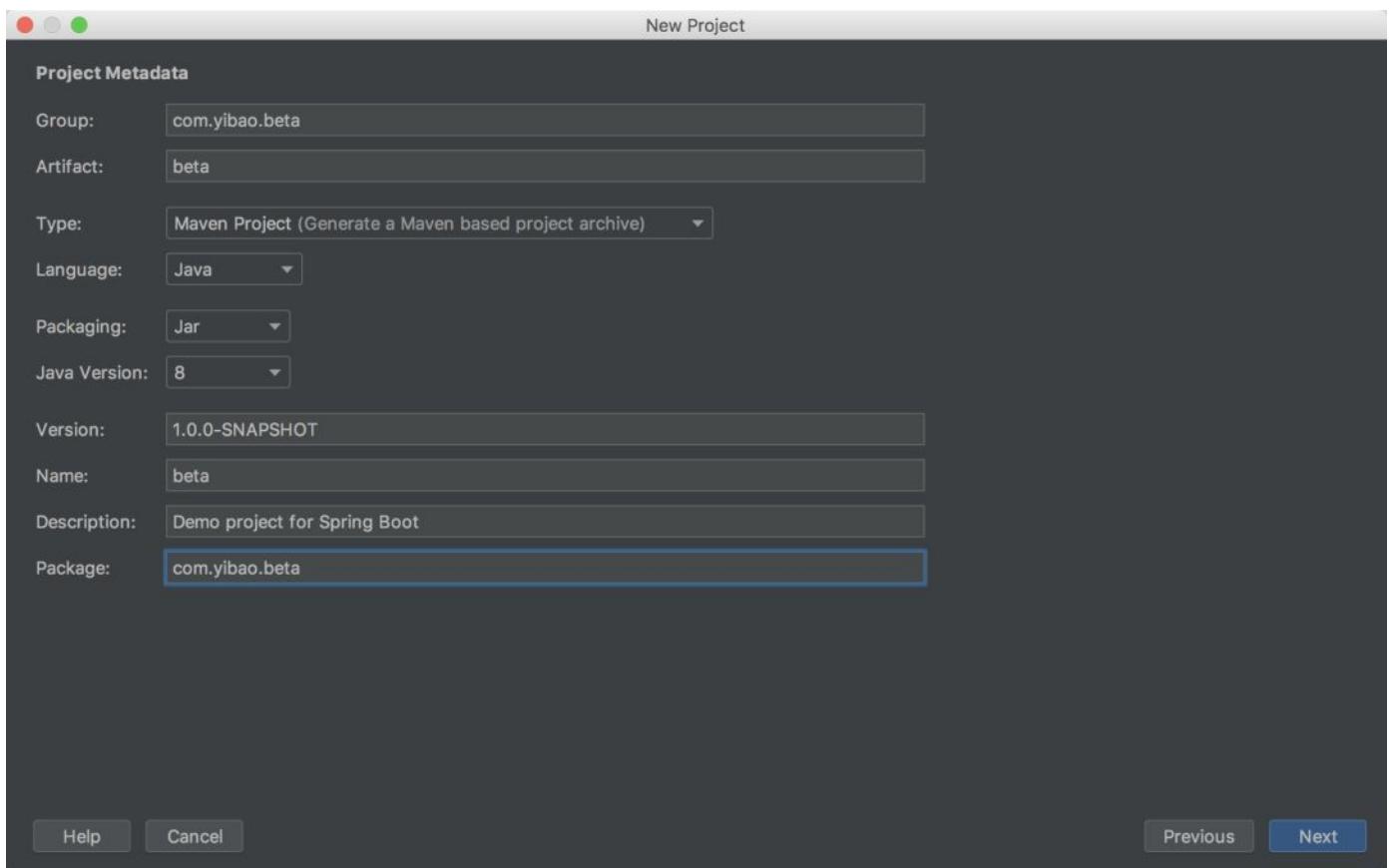
IDEA 工具栏选择菜单 File -> New -> Project...



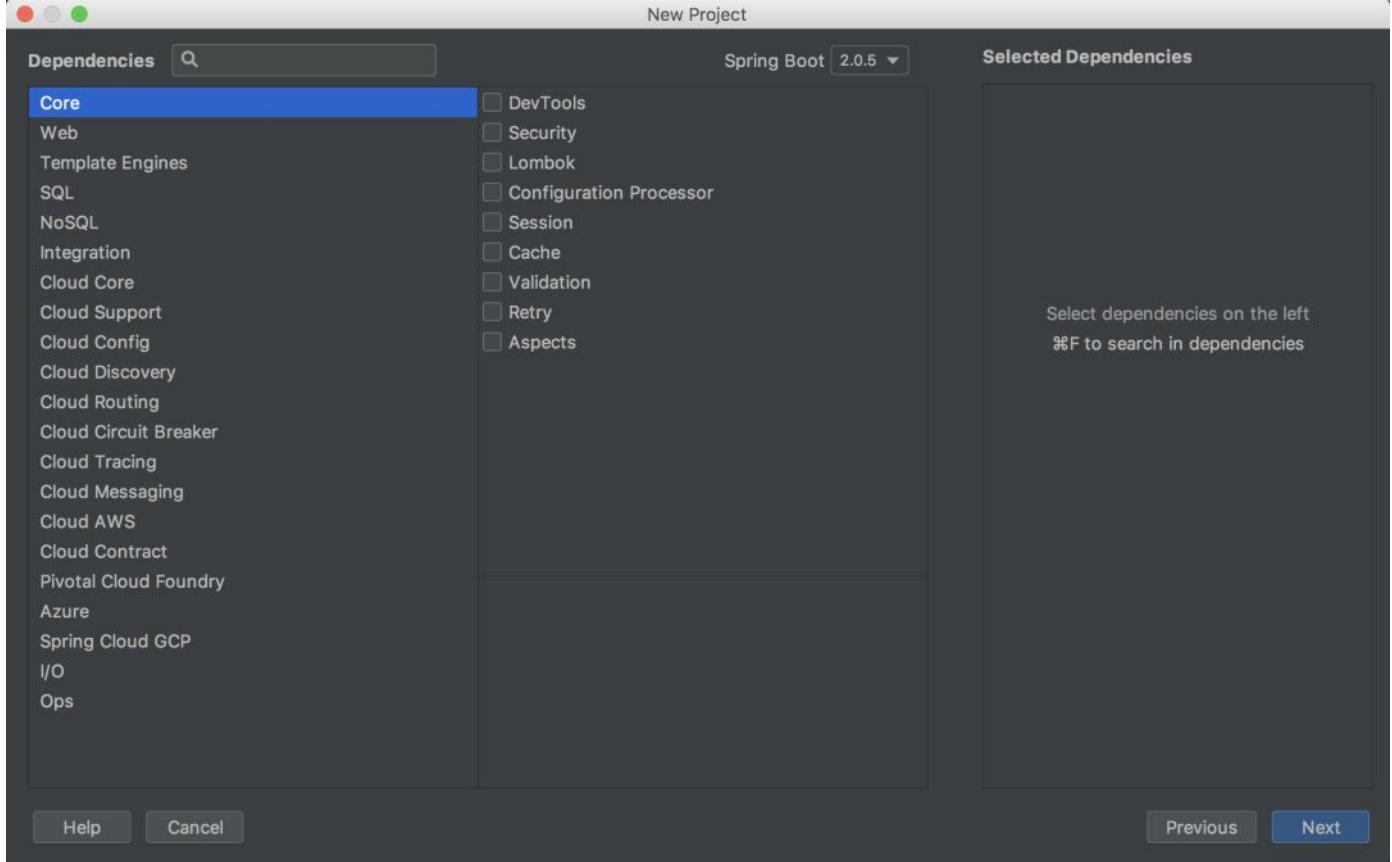
选择Spring Initializr, Initializr默认选择Default, 点击Next



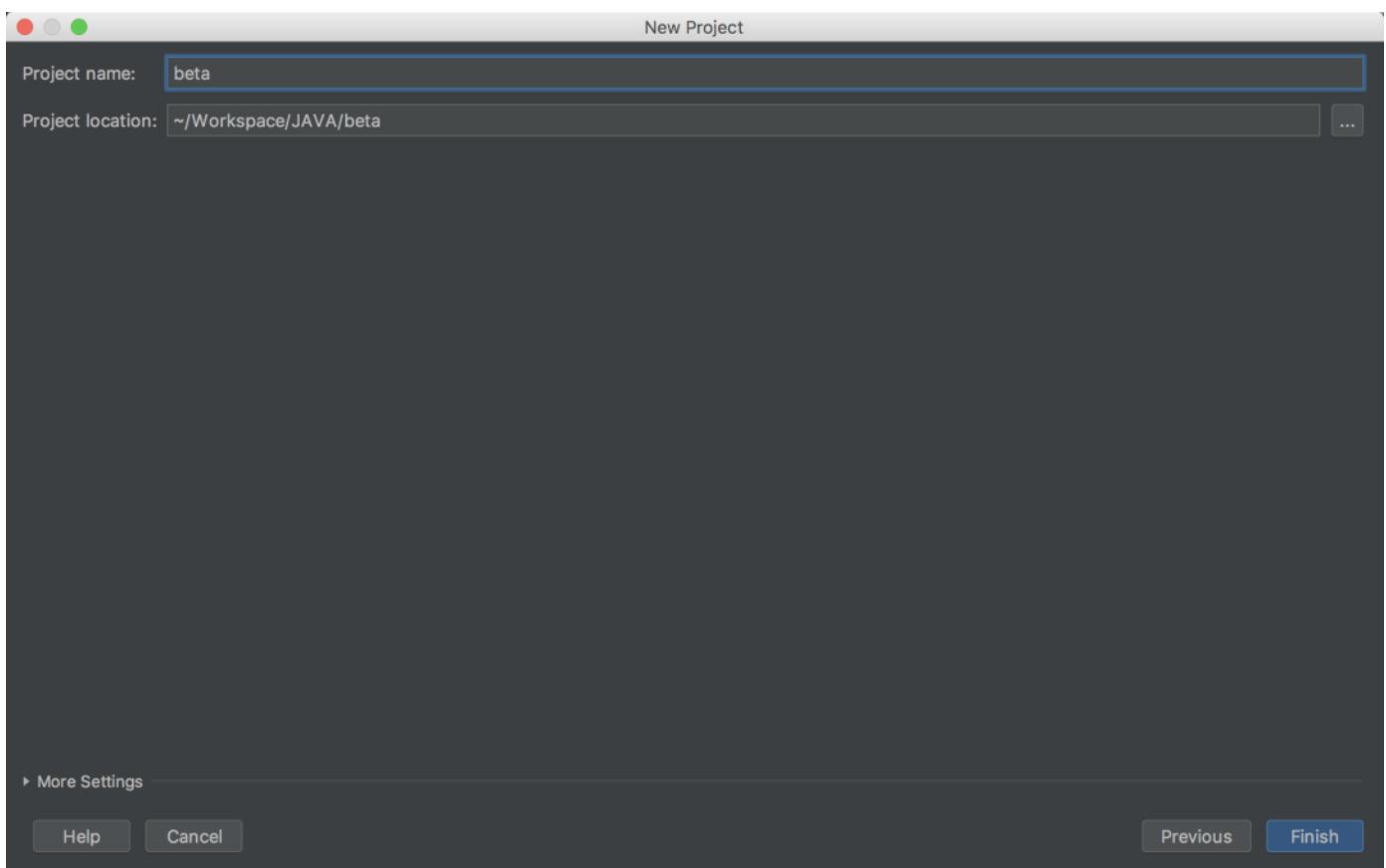
填写输入框，点击Next



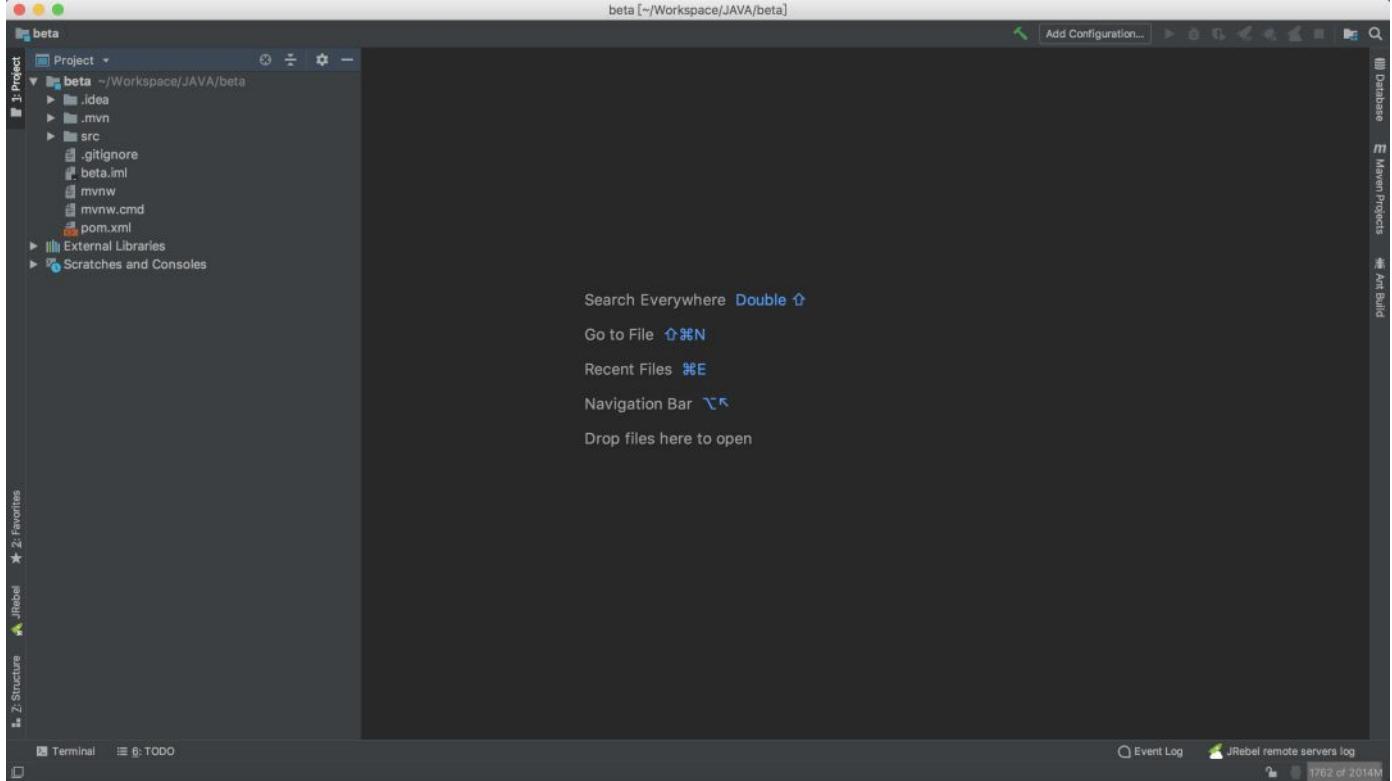
这步不需要选择直接点Next



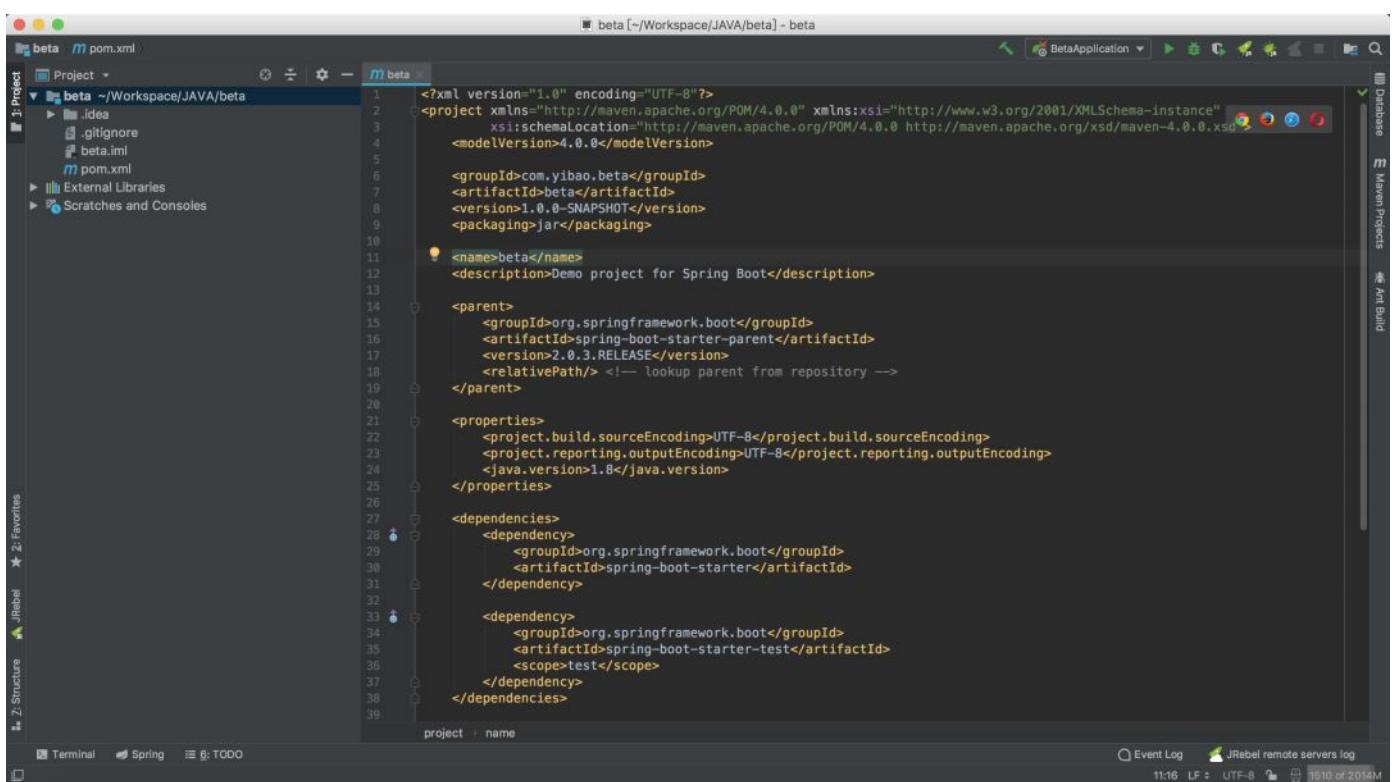
点击Finish创建项目



最终得到的项目目录结构如下

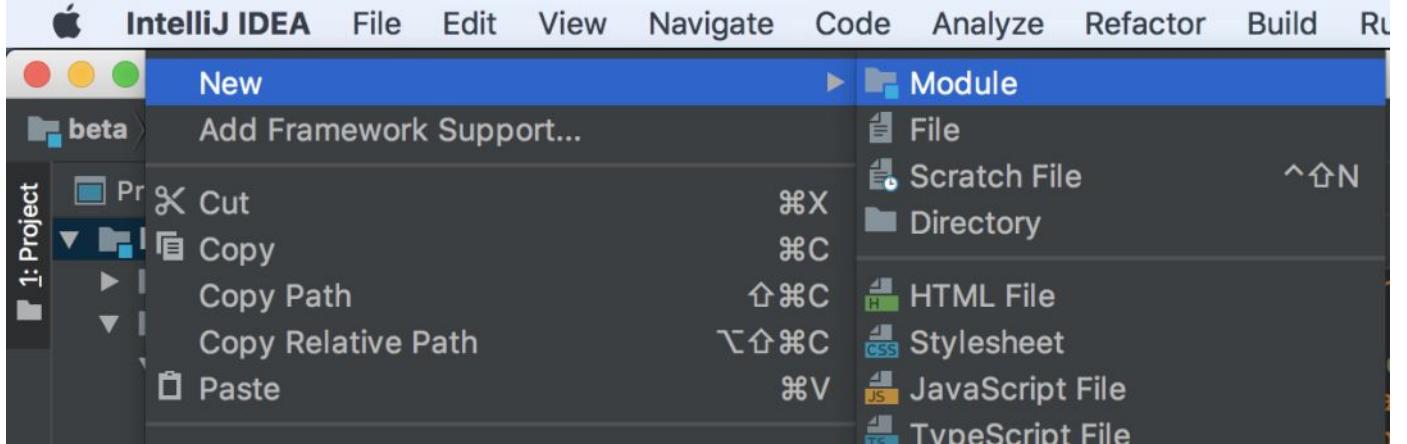


删除无用的.mvn目录、src目录、mvnw及mvnw.cmd文件，最终只留.gitignore和pom.xml

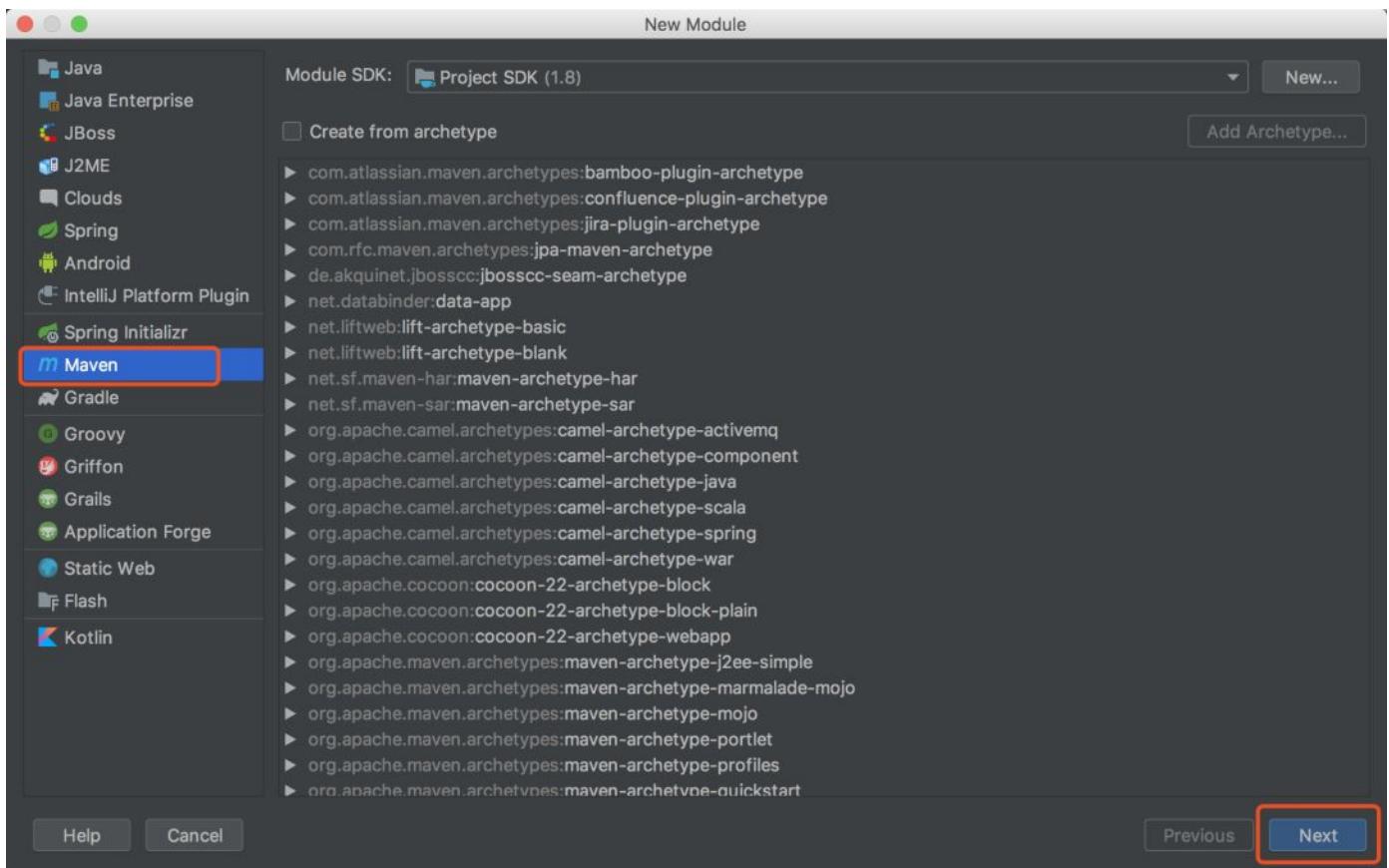


## 2、创建子模块

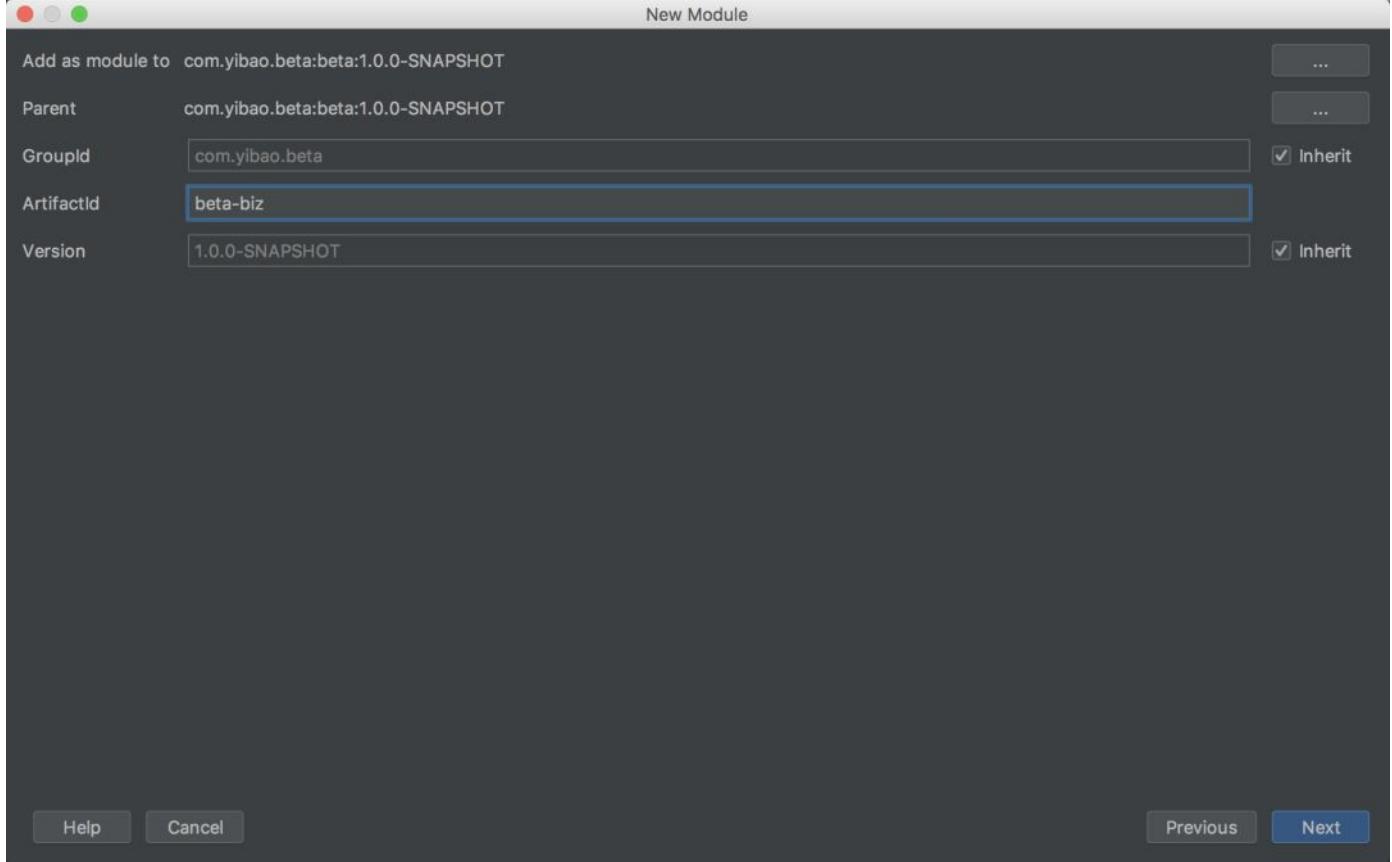
选择项目根目录beta右键呼出菜单，选择New -> Module



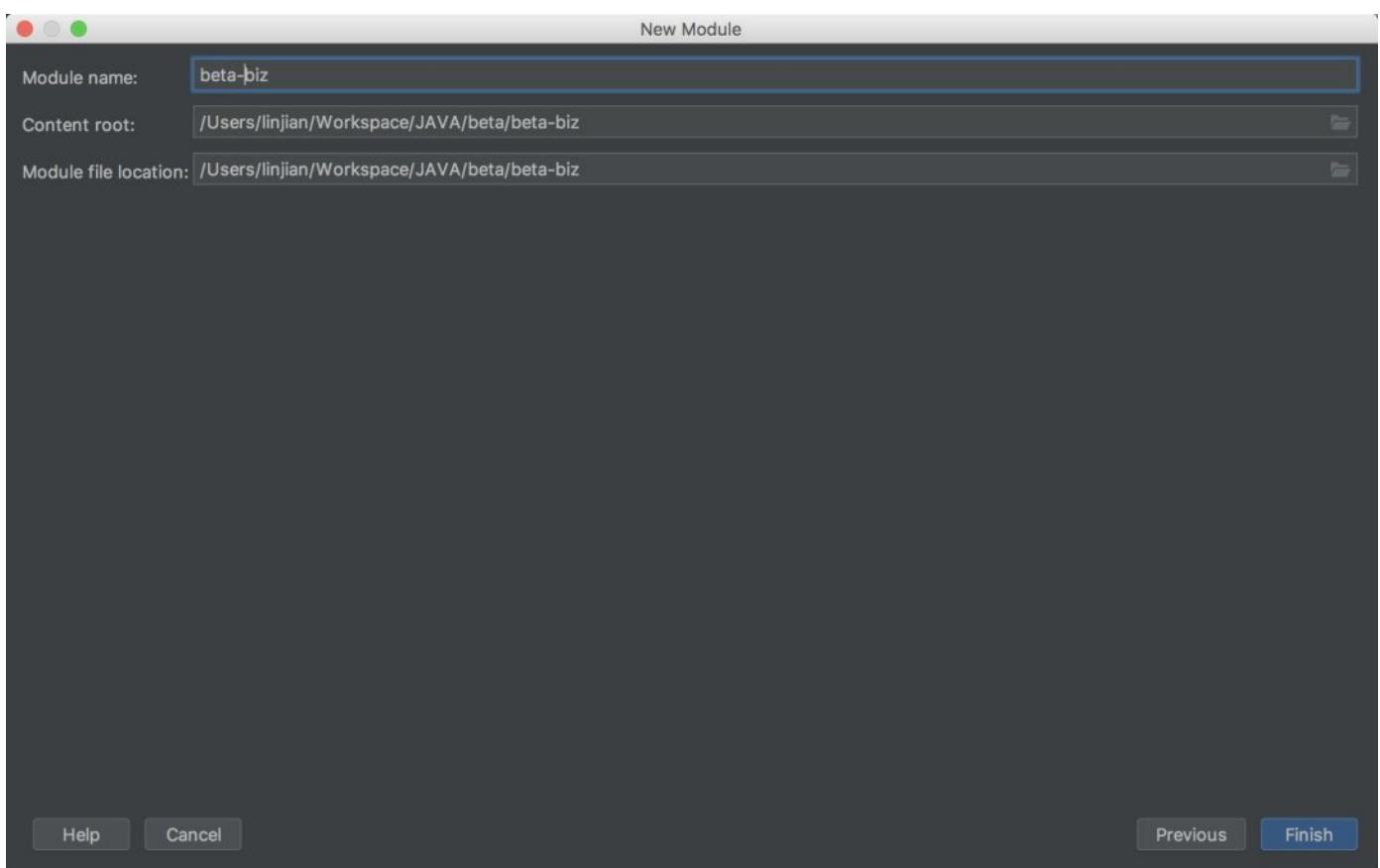
选择Maven，点击Next



填写ArtifactId，点击Next



修改Module name增加横杠提升可读性，点击Finish



同理添加beta-dao、beta-web子模块，最终得到项目目录结构如下图

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.yibao.beta</groupId>
  <artifactId>beta</artifactId>
  <version>1.0.0-SNAPSHOT</version>
  <modules>
    <module>beta-biz</module>
    <module>beta-dao</module>
    <module>beta-web</module>
  </modules>
  <packaging>pom</packaging>
  <name>beta</name>
  <description>Demo project for Spring Boot</description>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.0.3.RELEASE</version>
    <relativePath/> 
  </parent>
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
    <java.version>1.8</java.version>
  </properties>
  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
```

### 3、运行项目

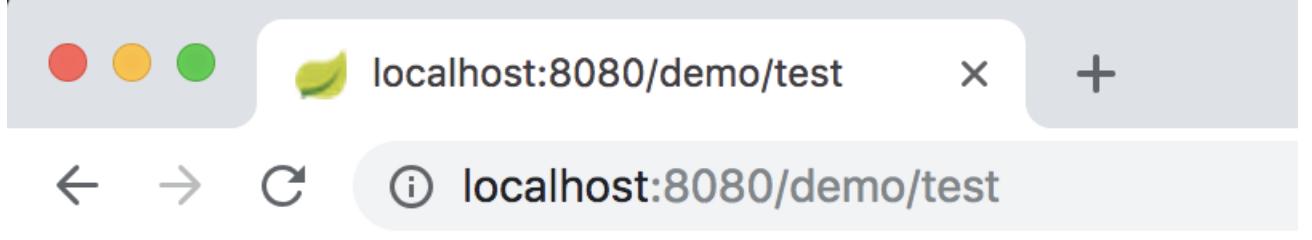
在beta-web层创建com.yibao.beta.web包(注意:这是多层目录结构并非单个目录名,com >> yibao >> beta >> web)并添加入口类BetaWebApplication.java

```
1 @SpringBootApplication
2 public class BetaWebApplication {
3
4     public static void main(String[] args)
5     {
6         SpringApplication.run(BetaWebApplication.class, args);
7     }
8 }
```

在com.yibao.beta.web包中添加controller目录并新建一个controller，添加test方法测试接口是否可以正常访问

```
1 @RestController
2 @RequestMapping("demo")
3 public class DemoController {
4
5     @GetMapping("test")
6     public String test() {
7         return "Hello World!";
8     }
9 }
```

运行BetaWebApplication类中的main方法启动项目，默认端口为8080，访问<http://localhost:8080/demo/test>得到如下效果



# Hello World!

以上虽然项目能正常启动，但是模块间的依赖关系却还未添加，下面继续完善。微信搜索 `web_resource` 获取更多推送

## 4、配置模块间的依赖关系

各个子模块的依赖关系：biz层依赖dao层，web层依赖biz层

父pom文件中声明所有子模块依赖（dependencyManagement及dependencies的区别自行查阅文档）

```
1 <dependencyManagement>
2   <dependencies>
3   >
4     <dependency>
5   >
6       <groupId>com.yibao.beta</groupId>
7   >
8       <artifactId>beta-biz</artifactId>
9   >
10      <version>${beta.version}</version>
11  >
12      </dependency>
13  >
14      <dependency>
15  >
16        <groupId>com.yibao.beta</groupId>
17  >
18        <artifactId>beta-dao</artifactId>
19  >
20        <version>${beta.version}</version>
21  >
22      </dependency>
23  >
24      <dependency>
25  >
26        <groupId>com.yibao.beta</groupId>
27  >
28        <artifactId>beta-web</artifactId>
29  >
30        <version>${beta.version}</version>
31  >
32      </dependency>
33  >
34  </dependencies>
```

```
</dependencyManagement>
```

其中\${beta.version}定义在properties标签中

在beta-web层中的pom文件中添加beta-biz依赖

```
1 <dependencies>
2     <dependency>
3     >
4         <groupId>com.yibao.beta</groupId>
5     >
6         <artifactId>beta-biz</artifactId>
7     >
8     </dependency>
9 >
</dependencies>
```

在beta-biz层中的pom文件中添加beta-dao依赖

```
1 <dependencies>
2     <dependency>
3     >
4         <groupId>com.yibao.beta</groupId>
5     >
6         <artifactId>beta-dao</artifactId>
7     >
8     </dependency>
9 >
</dependencies>
```

#### 4. web层调用biz层接口测试

在beta-biz层创建com.yibao.beta.biz包，添加service目录并在其中创建DemoService接口类，[微信搜索 web\\_resource 获取更多推送](#)

```
1 public interface DemoService {
2     String test()
3 }
4 }
```

```
1 @Service
2 public class DemoServiceImpl implements DemoService {
3
4     @Override
5     public String test()
6     {
7         return "test"
8     }
9 }
```

```
8     }
9 }
```

DemoController通过@.Autowired注解注入DemoService，修改DemoController的test方法使之调用DemoService的test方法，最终如下所示：

```
1 package com.yibao.beta.web.controller;@RestController
2 @RequestMapping("demo")
3 public class DemoController {
4
5     @Autowired
6     private DemoService demoService;
7
8     @GetMapping("test"
9     )
10    public String test() {
11        return demoService.test();
12    }
13}
```

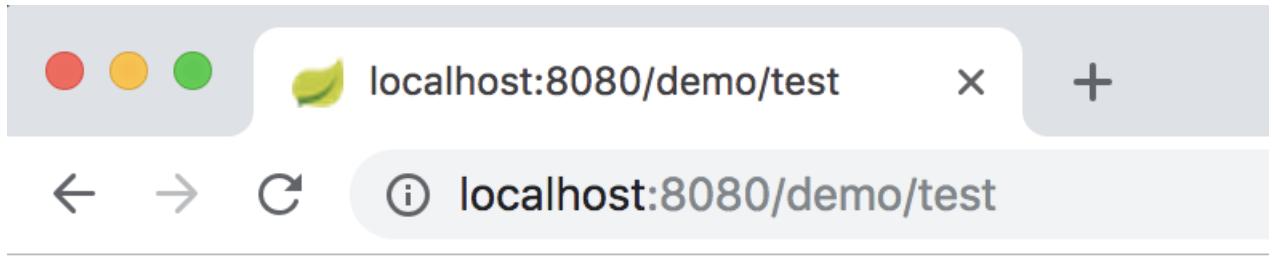
再次运行BetaWebApplication类中的main方法启动项目，发现如下报错

```
1 ****
2 APPLICATION FAILED TO START
3 ****
4
5 Description:
6 Field demoService in com.yibao.beta.web.controller.DemoController required a bean of type 'com.yibao.beta.biz.service.DemoService' which could not be found.
7
8 Action:
9 Consider defining a bean of type 'com.yibao.beta.biz.service.DemoService' in your configuration.
```

原因是找不到DemoService类，此时需要在BetaWebApplication入口类中增加包扫描，设置@SpringBootApplication注解中的scanBasePackages值为com.yibao.beta，最终如下所示

```
1 package com.yibao.beta.web;
2
3 @SpringBootApplication(scanBasePackages = "com.yibao.beta"
4 )
5 @MapperScan("com.yibao.beta.dao.mapper")
6 public class BetaWebApplication {
7
8     public static void main(String[] args) {
9         SpringApplication.run(BetaWebApplication.class, args)
10    ;
11    }
12}
```

设置完后重新运行main方法，项目正常启动，访问http://localhost:8080/demo/test得到如下效果



## 6. 集成Mybatis

父pom文件中声明mybatis-spring-boot-starter及lombok依赖

```
1 <dependencyManagement>
2   <dependencies>
3   >
4     <dependency>
5     >
6       <groupId>org.mybatis.spring.boot</groupId>
7     >
8       <artifactId>mybatis-spring-boot-starter</artifactId>
9     >
10      <version>1.3.2</version>
11    >
12    </dependency>
13  >
14  <dependency>
15  >
16    <groupId>org.projectlombok</groupId>
17  >
18    <artifactId>lombok</artifactId>
19  >
20    <version>1.16.22</version>
21  >
22    </dependency>
23  >
24  </dependencies>
25 >
26 </dependencyManagement>
```

在beta-dao层中的pom文件中添加上述依赖

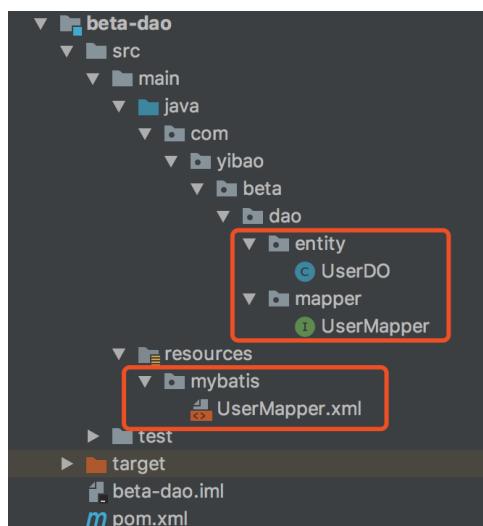
```
1 <dependencies>
2   <dependency>
3   >
4     <groupId>org.mybatis.spring.boot</groupId>
5   >
```

```

6   <artifactId>mybatis-spring-boot-starter</artifactId>
7   >
8     </dependency>
9   >
10    <dependency>
11      >
12        <groupId>mysql</groupId>
13      >
14        <artifactId>mysql-connector-java</artifactId>
15      >
16        </dependency>
17      >
18        <dependency>
19          >
20            <groupId>org.projectlombok</groupId>
21          >
22            <artifactId>lombok</artifactId>
23          >
24            </dependency>
25          >
26        </dependencies>

```

在beta-dao层创建com.yibao.beta.dao包，通过mybatis-generatoor工具生成dao层相关文件（DO、Mapper、xml），存放目录如下



application.properties文件添加jdbc及mybatis相应配置项

```

1 spring.datasource.driverClassName = com.mysql.jdbc.Driver
2 spring.datasource.url = jdbc:mysql://192.168.1.1/test?useUnicode=true&characterEncoding=utf-8
3 spring.datasource.username = test
4 spring.datasource.password = 123456
5
6 mybatis.mapper-locations = classpath:mybatis/*.xml
7 mybatis.type-aliases-package = com.yibao.beta.dao.entity
8

```

DemoService通过@Autowired注解注入UserMapper，修改DemoService的test方法使之调用UserMapper的

selectByPrimaryKey方法，最终如下所示

```
1 package com.yibao.beta.biz.service.impl;
2
3 @Service
4 public class DemoServiceImpl implements DemoService {
5
6     @Autowired
7     private UserMapper userMapper;
8
9     @Override
10    public String test()
11    {
12        UserDO user = userMapper.selectByPrimaryKey(1)
13    ;
14        return user.toString();
15    }
16}
```

再次运行BetaWebApplication类中的main方法启动项目，发现如下报错

```
1 APPLICATION FAILED TO START
*****
2
3
4 Description:
5 Field userMapper in com.yibao.beta.biz.service.impl.DemoServiceImpl required a bean of type 'com.y
6 .
7
8
9 Action:
10 Consider defining a bean of type 'com.yibao.beta.dao.mapper.UserMapper' in your configuration.
```

原因是找不到UserMapper类，此时需要在BetaWebApplication入口类中增加dao层包扫描，添加@MapperScan注解并设置其值为com.yibao.beta.dao.mapper，最终如下所示

```
1 package com.yibao.beta.web;
2
3 @SpringBootApplication(scanBasePackages = "com.yibao.beta"
4 )
5 @MapperScan("com.yibao.beta.dao.mapper")
6 public class BetaWebApplication {
7
8     public static void main(String[] args) {
9         SpringApplication.run(BetaWebApplication.class, args)
10    ;
11        }
12 }
```

设置完后重新运行main方法，项目正常启动，访问http://localhost:8080/demo/test得到如下效果



localhost:8080/demo/test

## UserDO(id=1, userName=linjian)

至此，一个简单的SpringBoot+Mybatis多模块项目已经搭建完毕，我们也通过启动项目调用接口验证其正确性。

### 四、总结

一个层次分明的多模块工程结构不仅方便维护，而且有利于后续微服务化。在此结构的基础上还可以扩展common层（公共组件）、server层（如dubbo对外提供的服务）[微信搜索 web\\_resource 获取更多推送](#)

此为项目重构的第一步，后续还会在框架中集成logback、disconf、redis、dubbo等组件

### 五、未提到的坑

在搭建过程中还遇到一个maven私服的问题，原因是公司内部的maven私服配置的中央仓库为阿里的远程仓库，它与maven自带的远程仓库相比有些jar包版本并不全，导致在搭建过程中好几次因为没拉到相应jar包导致项目启动不了。

-END-

如果看到这里，说明你喜欢这篇文章，请[转发、点赞](#)。微信搜索「web\_resource」，关注后回复「进群」或者扫描下方二维码即可进入无广告交流群。

↓ 扫描二维码进群 ↓



### 推荐阅读

1. Java后端优质文章整理
2. IDEA 远程一键部署 Spring Boot 到 Docker
3. 这 26 条，你赞同几个？
4. 7 个开源的 Spring Boot 前后端分离项目

**5. 如何设计 API 接口, 实现统一格式返回?**



Java后端

长按识别二维码, 关注我的公众号

喜欢文章, 点个在看 

[阅读原文](#)

声明: pdf仅供学习使用, 一切版权归原创公众号所有; 建议持续关注原创公众号获取最新文章, 学习愉快!

# Spring Boot + Mybatis 配合 AOP 和注解实现动态数据源切换配置

韩数 Java后端 2019-10-25

点击上方 Java后端, 选择 [设为星标](#)

技术博文, 及时送达

来自 | 韩数

链接 | [juejin.im/post/5d830944f265da03963bd153](https://juejin.im/post/5d830944f265da03963bd153)

## 0、前言

随着应用用户数量的增加，相应的并发请求的数量也会跟着不断增加，慢慢地，单个数据库已经没有办法满足我们频繁的数据库操作请求了。

在某些场景下，我们可能会需要配置多个数据源，使用多个数据源(例如实现数据库的读写分离)来缓解系统的压力等，同样的，Springboot官方提供了相应的实现来帮助开发者们配置多数据源，一般分为两种方式(目前我所了解到的)，分包和AOP。

而在Springboot +Mybatis实现多数据源配置中，我们实现了静态多数据源的配置，但是这种方式怎么说呢，在实际的使用中不够灵活，为了解决这个问题，我们可以使用上文提到的第二种方法,即使用AOP面向切面编程的方式配合我们的自定义注解来实现不同数据源之间动态切换的目的。

## 1、数据库准备

数据库准备仍然和之前的例子相同，具体建表sql语句则不再详细说明，表格如下：

数据库	testdatasource1	testdatasource2
数据表	sys_user	sys_user2
字段	user_id(int), user_name(varchar) user_age (int)	同

并分别插入两条记录，为了方便对比，其中testdatasource1为芳年25岁的张三， testdatasource2为芳年30岁的李四。

## 2、环境准备

首先新建一个Springboot项目，我这里版本是2.1.7.RELEASE，并在pom文件中引入相关依赖，和上次相比，这次主要额外新增了aop相关的依赖，如下：

```
1 <dependency>
2   <groupId>mysql</groupId>
3   <artifactId>mysql-connector-java</artifactId>
4 </dependency>
5 <dependency>
6   <groupId>org.springframework.boot</groupId>
7   <artifactId>spring-boot-starter-jdbc</artifactId>
8 </dependency>
9 <dependency>
10  <groupId>org.springframework</groupId>
```

```
11 <artifactId>spring-aop</artifactId>
12 <version>5.1.5.RELEASE</version>
13 </dependency>
14 <dependency>
15   <groupId>junit</groupId>
16   <artifactId>junit</artifactId>
17 </dependency>
18 <dependency>
19   <groupId>org.aspectj</groupId>
20   <artifactId>aspectjweaver</artifactId>
21   <version>1.9.2</version>
22 </dependency>
```

### 3、代码部分

首先呢，在我们Springboot的配置文件中配置我们的datasource，和以往不一样的是，因为我们有两个数据源，所以要指定相关数据库的名称，其中主数据源为primary，次数据源为secondary如下：

```
1 #配置主数据库
2 spring.datasource.primary.jdbc-url=jdbc:mysql://localhost:3306/testdatasource1?useUnicode=true&cha
3 spring.datasource.primary.username=root
4 spring.datasource.primary.password=root
5 spring.datasource.primary.driver-class-name=com.mysql.cj.jdbc.Driver
6
7 ##配置次数据库
8 spring.datasource.secondary.jdbc-url=jdbc:mysql://localhost:3306/testdatasource2?useUnicode=true&c
9 spring.datasource.secondary.username=root
10 spring.datasource.secondary.password=root
11 spring.datasource.secondary.driver-class-name=com.mysql.cj.jdbc.Driver
12
13
14 spring.http.encoding.charset=UTF-8
15 spring.http.encoding.enabled=true
16 spring.http.encoding.force=true
```

需要我们注意的是，Springboot2.0 在配置数据库连接的时候需要使用jdbc-url，如果只使用url的话会报 **jdbcUrl is required with driverClassName.** 错误。

新建一个配置文件，DynamicDataSourceConfig 用来配置我们相关的bean,代码如下

```
1 @Configuration
2 @MapperScan(basePackages = "com.mzd.multipledatasources.mapper", sqlSessionFactoryRef = "SqlSession"
3 public class DynamicDataSourceConfig {
4
5     // 将这个对象放入Spring容器中
6     @Bean(name = "PrimaryDataSource")
7     // 表示这个数据源是默认数据源
8     @Primary
9     // 读取application.properties中的配置参数映射成为一个对象
10    // prefix表示参数的前缀
11    @ConfigurationProperties(prefix = "spring.datasource.primary")
```

```

12     public DataSource getDateSource1() {
13         return DataSourceBuilder.create().build();
14     }
15
16
17     @Bean(name = "SecondaryDataSource")
18     @ConfigurationProperties(prefix = "spring.datasource.secondary")
19     public DataSource getDateSource2() {
20         return DataSourceBuilder.create().build();
21     }
22
23
24     @Bean(name = "dynamicDataSource")
25     public DynamicDataSource DataSource(@Qualifier("PrimaryDataSource") DataSource primaryDataSour
26                                         @Qualifier("SecondaryDataSource") DataSource secondaryDataS
27
28         //这个地方是比较核心的targetDataSource 集合是我们数据库和名字之间的映射
29         Map<Object, Object> targetDataSource = new HashMap<>();
30         targetDataSource.put(DataSourceType.DataBaseType.Primary, primaryDataSource);
31         targetDataSource.put(DataSourceType.DataBaseType.Secondary, secondaryDataSource);
32         DynamicDataSource dataSource = new DynamicDataSource();
33         dataSource.setTargetDataSources(targetDataSource);
34         dataSource.setDefaultTargetDataSource(primaryDataSource); //设置默认对象
35         return dataSource;
36     }
37
38
39     @Bean(name = "SqlSessionFactory")
40     public SqlSessionFactory SqlSessionFactory(@Qualifier("dynamicDataSource") DataSource dynamicD
41             throws Exception {
42         SqlSessionFactoryBean bean = new SqlSessionFactoryBean();
43         bean.setDataSource(dynamicDataSource);
44         bean.setMapperLocations(
45             new PathMatchingResourcePatternResolver().getResources("classpath*:mapping/*/*.xml"));
46         return bean.getObject();
47     }
48 }
```

而在这所有的配置中，最核心的地方就是DynamicDataSource这个类了，DynamicDataSource是我们自定义的动态切换数据源的类，该类继承了AbstractRoutingDataSource 类并重写了它的determineCurrentLookupKey()方法。

AbstractRoutingDataSource 类内部维护了一个名为targetDataSources的Map，并提供的setter方法用于设置数据源关键字与数据源的关系，实现类被要求实现其determineCurrentLookupKey()方法，由此方法的返回值决定具体从哪个数据源中获取连接。同时AbstractRoutingDataSource类提供了程序运行时动态切换数据源的方法，在dao类或方法上标注需要访问数据源的关键字，路由到指定数据源，获取连接。

DynamicDataSource代码如下：

```

1  public class DynamicDataSource extends AbstractRoutingDataSource {
2
3     @Override
4     protected Object determineCurrentLookupKey() {
5         DataBaseType DataBaseType = DataBaseType.getDataBaseType();
```

```
5     return DataBaseType;
6 }
7
8 }
9
```

DataSourceType类的代码如下：

```
1 public class DataSourceType {
2
3     //内部枚举类，用于选择特定的数据类型
4     public enum DataBaseType {
5         Primary, Secondary
6     }
7
8     // 使用ThreadLocal保证线程安全
9     private static final ThreadLocal<DataBaseType> TYPE = new ThreadLocal<DataBaseType>();
10
11    // 往当前线程里设置数据源类型
12    public static void set DataBaseType(DataBaseType DataBaseType) {
13        if (dataBaseType == null) {
14            throw new NullPointerException();
15        }
16        TYPE.set(dataBaseType);
17    }
18
19    // 获取数据源类型
20    public static DataBaseType get DataBaseType() {
21        DataBaseType DataBaseType = TYPE.get() == null ? DataBaseType.Primary : TYPE.get();
22        return DataBaseType;
23    }
24
25    // 清空数据类型
26    public static void clear DataBaseType() {
27        TYPE.remove();
28    }
29
30 }
```

接下来编写我们相关的Mapper和xml文件，代码如下：

```
1 @Component
2 @Mapper
3 public interface PrimaryUserMapper {
4
5     List<User> findAll();
6 }
7
8 @Component
9 @Mapper
10 public interface SecondaryUserMapper {
11
12     List<User> findAll();
13 }
```

```

1 <?xml version="1.0" encoding="UTF-8" ?>
2 <!DOCTYPE mapper
3     PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
4     "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
5 <mapper namespace="com.jdkcb.mybatisstuday.mapper.one.PrimaryUserMapper">
6
7     <select id="findAll" resultType="com.jdkcb.mybatisstuday.pojo.User">
8         select * from sys_user;
9     </select>
10
11 </mapper>
12
13
14 <?xml version="1.0" encoding="UTF-8" ?>
15 <!DOCTYPE mapper
16     PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
17     "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
18 <mapper namespace="com.jdkcb.mybatisstuday.mapper.two.SecondaryUserMapper">
19
20     <select id="findAll" resultType="com.jdkcb.mybatisstuday.pojo.User">
21         select * from sys_user2;
22     </select>
23
24
25 </mapper>

```

相关接口文件编写好之后，就可以编写我们的aop代码了：

```

1 @Aspect
2 @Component
3 public class DataSourceAop {
4     //在primary方法前执行
5     @Before("execution(* com.jdkcb.mybatisstuday.controller.UserController.primary(..))")
6     public void setDataSource2test01() {
7         System.out.println("Primary业务");
8         DataSourceType.set DataBaseType(DataSourceType.DataBaseType.Primary);
9     }
10
11     //在secondary方法前执行
12     @Before("execution(* com.jdkcb.mybatisstuday.controller.UserController.secondary(..))")
13     public void setDataSource2test02() {
14         System.out.println("Secondary业务");
15         DataSourceType.set DataBaseType(DataSourceType.DataBaseType.Secondary);
16     }
17 }

```

编写我们的测试 UserController， 代码如下：

```

1 @RestController

```

```
2 public class UserController {  
3  
4     @Autowired  
5     private PrimaryUserMapper primaryUserMapper;  
6     @Autowired  
7     private SecondaryUserMapper secondaryUserMapper;  
8  
9  
10    @RequestMapping("primary")  
11    public Object primary(){  
12        List<User> list = primaryUserMapper.findAll();  
13        return list;  
14    }  
15    @RequestMapping("secondary")  
16    public Object secondary(){  
17        List<User> list = secondaryUserMapper.findAll();  
18        return list;  
19    }  
20  
21 }
```

## 4、测试

启动项目，在浏览器中分别输入http://127.0.0.1:8080/primary 和http://127.0.0.1:8080/secondary，结果如下：

```
1 [{"user_id":1,"user_name":"张三","user_age":25}]  
2 [{"user_id":1,"user_name":"李四","user_age":30}]
```

## 5、等等

等等，啧啧啧，我看你这不行啊，还不够灵活，几个菜啊，喝成这样，这就算灵活了？

那肯定不能的，aop也有aop的好处，比如两个包下的代码分别用两个不同的数据源，就可以直接用aop表达式就可以完成了，但是，如果想本例中方法级别的拦截，就显得优点不太灵活了，这个适合就需要我们的注解上场了。

## 6、配合注解实现

首先自定义我们的注解 @DataSource

```
1 /**  
2  * 切换数据注解 可以用于类或者方法级别 方法级别优先级 > 类级别  
3  */  
4 @Target({ElementType.METHOD, ElementType.TYPE, ElementType.PARAMETER})  
5 @Retention(RetentionPolicy.RUNTIME)  
6 @Documented  
7 public @interface DataSource {  
8     String value() default "primary"; //该值即key值， 默认使用默认数据库  
9 }
```

通过使用aop拦截，获取注解的属性value的值。如果value的值并没有在我们DataBaseType里面，则使用我们默认的数据源，如果有的话，则切换为相应的数据源。

```
1 @Aspect
2 @Component
3 public class DynamicDataSourceAspect {
4     private static final Logger logger = LoggerFactory.getLogger(DynamicDataSourceAspect.class);
5
6     @Before("@annotation(dataSource)");//拦截我们的注解
7     public void changeDataSource(JoinPoint point, DataSource dataSource) throws Throwable {
8         String value = dataSource.value();
9         if (value.equals("primary")){
10             DataSourceType.setDatabaseType(DataSourceType.DatabaseType.Primary);
11         }else if (value.equals("secondary")){
12             DataSourceType.setDatabaseType(DataSourceType.DatabaseType.Secondary);
13         }else {
14             DataSourceType.setDatabaseType(DataSourceType.DatabaseType.Primary);//默认使用主数据库
15         }
16
17     }
18
19     @After("@annotation(dataSource) ") //清除数据源的配置
20     public void restoreDataSource(JoinPoint point, DataSource dataSource) {
21         DataSourceType.clearDatabaseType();
22
23
24     }
25 }
```

## 7、测试

修改我们的mapper，添加我们的自定义的@DataSouse注解，并注解掉我们DataSourceAop类里面的内容。如下：

```
1 @Component
2 @Mapper
3 public interface PrimaryUserMapper {
4
5     @DataSource
6     List<User> findAll();
7 }
8
9 @Component
10 @Mapper
11 public interface SecondaryUserMapper {
12
13     @DataSource("secondary")//指定数据源为：secondary
14     List<User> findAll();
15 }
```

启动项目，在浏览器中分别输入http://127.0.0.1:8080/primary 和http://127.0.0.1:8080/secondary，结果如下：

```
1 [{"user_id":1,"user_name":"张三", "user_age":25}]\n2 [{"user_id":1,"user_name":"李四", "user_age":30}]
```

到此，就算真正的大功告成啦。

## 8、源码

[github.com/hanshuaiKang/HanShu-Note](https://github.com/hanshuaiKang/HanShu-Note)

学Java，请关注公众号：Java后端



Java后端

长按识别二维码，关注我的公众号

声明：pdf仅供学习使用，一切版权归原创公众号所有；建议持续关注原创公众号获取最新文章，学习愉快！

# Spring MVC+Spring+MyBatis实现支付宝扫码支付功能（图文详解）

纪莫 Java后端 2019-11-13

点击上方 Java后端, 选择 [设为星标](#)

优质文章, 及时送达

作者 | 思海(公众号:好好学java)

链接 | [blog.csdn.net/sihai12345](http://blog.csdn.net/sihai12345)

## 前言

本教程详细介绍了如何使用 SSM 框架实现支付宝支付功能。本文章分为两大部分，分别是「支付宝测试环境代码测试」和「将支付宝支付整合到 SSM 框架」，详细的代码和图文解释，自己实践的时候一定仔细阅读相关文档，话不多说我们开始。

## 支付宝测试环境代码测试

### 1. 下载电脑网站的官方demo：

下载：<https://docs.open.alipay.com/270/106291/>

The screenshot shows the Alipay Developer Documentation website. On the left, there's a sidebar with links like '全部文档', '产品介绍', '快速接入', '支付结果异步通知', 'SDK&Demo' (which is highlighted), 'API列表', and '联调问题排查'. The main content area has a header '电脑网站支付SDK&Demo' with a timestamp '更新时间：2017-10-27'. Below it is a section titled '开放平台服务端SDK' with a note about the SDK for Java, PHP, and .NET. It also includes a link to '下载和使用教程'. Another section below is titled '电脑网站支付demo', with a '下载demo' button. A table lists two demo versions: 'JAVA版' and 'PHP版', each with its respective runtime environment requirements and a '点击下载' button.

Demo版本	运行环境	下载链接
JAVA版	Eclipse+JDK1.6及以上+Tomcat6.0及以上	<a href="#">点击下载</a>
PHP版	PHP5.5及以上+Tomcat6.0及以上	<a href="#">点击下载</a>

### 2. 下载解压导入eclipse

readme.txt请好好看一下。

只有一个Java配置类，其余都是JSP。

### 3. 配置AlipayConfig

(1) 注册蚂蚁金服开发者账号（免费，不像苹果会收取费用）

注册地址: <https://open.alipay.com> , 用你的支付宝账号扫码登录, 完善个人信息, 选择服务类型 (我选的是自研)。



## (2) 设置app\_id和gatewayUrl

安全 | <https://openhome.alipay.com/platform/developerIndex.htm>

应用 前端 学习路线 公司 杂家 项目 java me 架构 大数据 OS 插件 web ORM 认证与鉴权

蚂蚁金服开放平台 首页 服务商中心 开发者中心 服务市场

网页&移动应用 快速接入支付/行业

生活号 为商户提供服务解决方案

开发者中心 总览 >

研发服务 沙箱/凤蝶/验收/数据实验室

安全中心 检测/安骑士/防火墙/DDoS

监控中心 核心数据指标/健康度/告警



心 / 研发服务 / 沙箱环境 / 沙箱应用

沙箱长期稳定，每周日中午12点至每周一中午12点沙箱环境进行维护，期间可能出现不可用情况。

应用 (1)

配置

必看部分

APPID	2016090900468822
支付宝网关	<a href="https://openapi.alipaydev.com/gateway.do">https://openapi.alipaydev.com/gateway.do</a>
RSA2(SHA256)密钥(推荐)	<a href="#">查看应用公钥</a>   <a href="#">查看支付宝公钥</a>

AlipayConfig.java

```

17 // 应用 ID,您的APPID。收款账号既是您的APPID对应支付宝账号
18 public static String app_id = "2016090900468822";
19
20 // 商户私钥,您的PKCS8格式RSA2私钥
21 public static String merchant_private_key = "MIIEvQIBADANBgkqhkiG9w0B
22
23 // 支付宝公钥,查看地址: https://openhome.alipay.com/platform/keyManage
24 public static String alipay_public_key = "MIIBIjANBgkqhkiG9w0BAQEFAAC
25
26 // 服务器异步通知页面路径,需http://格式的完整路径,不能加?id=123这类自定
27 public static String notify_url = "http://localhost:8080/alipay.trade
28
29 // 页面跳转同步通知页面路径,需http://格式的完整路径,不能加?id=123这类自定
30 public static String return_url = "http://localhost:8080/alipay.trade
31
32 public static String sign_type = "RSA2";
33 // 字符编码格式
34 public static String charset = "utf-8";
35 // 支付宝网关
36 public static String gatewayUrl = "https://openapi.alipaydev.com/gate
37
38 public static String log_path = "d:\\test\\pay\\";

```

小溪\_宁静而致远© 2017

其中密钥需要自己生成，appID和支付宝网关是已经给好的，网关有dev字样，表明是用于开发测试。

### (3) 设置密钥

APPID	2016090900468822	有技术问题点我
支付宝网关	<a href="https://openapi.alipaydev.com/gateway.do">https://openapi.alipaydev.com/gateway.do</a>	开发者要保证接口中使用的私钥与此处的公钥匹配，否则无法调用接口。可参考密钥的 <a href="#">生成方法</a> 。沙箱支付宝公钥与线上不同，请更换代码中配置；
RSA2(SHA256)密钥(推荐)	<a href="#">查看应用公钥</a>   <a href="#">查看支付宝公钥</a>	小溪_宁静而致远© 2017

点击“生成方法”，打开界面如下：

搜索

**快速入门**

**教程**  
[生成RSA密钥](#)

[上传应用公钥并获取支付...](#)

[使用应用私钥生成请求签名](#)

[使用支付宝公钥验签](#)

**自助排查**

**工具**

**FAQ**

**补充说明**

## 生成RSA密钥

更新时间：2017/03/29 访问次数：134207

**阅读角色：技术同学**

支付宝提供一键生成工具便于开发者生成一对RSA密钥，可通过下方链接下载密钥生成工具：

**WINDOWS**

**MAC OSX**

下载该工具后，解压打开文件夹，运行“RSA签名验签工具.bat”（WINDOWS）或“RSA签名验签工具.command”（MAC OSX）。

界面示例：



小溪\_宁静而致远© 2017

下周密钥生成工具，解压打开后，选择2048位生成密钥：

**RSA签名验签工具(V1.3)**

**生成密钥 签名 验签 格式转换 密钥匹配**

密钥格式： PKCS8(JAVA适用)  PKCS1(非JAVA适用)

密钥长度： 1024  2048

**生成密钥** **打开密钥文件路径**

商户应用私钥：  
MIIEvQIBADANBgkqhkiG9w0BAQEFAASCBKcwggSjAgEAAoIBAC...  
hWQZF3ojVGHUbw8YiFP8DmY  
+ySzao06yBid1I1wx3X4HIMLE0UGS4EEFOX6Y0gZ/suhjEV7vito...  
VrgCNb13EX69KOsJmeSVn2DlomkM2QVW1GOzDO8r05umpJ8...  
zQk9lZcnfolghpmucp1XKnqC9aLJwzBi5woI51bisGzTlSkKzjQNL5...  
vgJmEMy8mFZiHalsSposYM2HOHzYgAmK89xoOvHCSnElK6r...  
GIDdIGAnAgMBAAECggEBAJOXpL4xL90zlxQkLUq2oJUHwlQK/k...  
+JyQxMUv/uO6LlBpiq8UoZlyAwRCsbjXmerJkp9FV4KPe2yLP/n...  
J5iT Rmp3VuVlNgGfo63NQWa6WLO  
+F7sqTIK9H4cn6aFYyHGD62djLQg5H4x3bbNAQNjX/XkcYtCr9t...  
rgfW0UAwpJu29A1ipYRydg7a4e8pjISzyOs1FivE0l2Syfs9+e1/m...  
tP8YSwpVmfbzvV2oOlkj5y795BdP3rGRtFEjvU9gUFfu8vECgYE...  
+mNIGf7LoYxFe74raNx4k2WTIT4tVqws3IPCWqf1a4AjW9dxF...  
+vSjrCZnkIz9g5aJJjNkFvtRjszvK9ObpqSfMgi2zwPPaA6/4uHh...  
nKdVvn6avWivcMwYucKledW4rRs05!Jrcs40f...  
小溪\_宁静而致远© 2017

商户应用公钥：  
+mNIGf7LoYxFe74raNx4k2WTIT4tVqws3IPCWqf1a4AjW9dxF...  
+vSjrCZnkIz9g5aJJjNkFvtRjszvK9ObpqSfMgi2zwPPaA6/4uHh...  
nKdVvn6avWivcMwYucKledW4rRs05!Jrcs40f...  
小溪\_宁静而致远© 2017

如果没有设置过，此时显示文本是“设置应用公钥”，我这里是已经设置过得。

APPID <a href="#">i</a>	2016090900468822
支付宝网关 <a href="#">i</a>	<a href="https://openapi.alipaydev.com/gateway.do">https://openapi.alipaydev.com/gateway.do</a>
RSA2(SHA256)密钥(推荐) <a href="#">i</a>	<a href="#">查看应用公钥</a>   <a href="#">查看支付宝公钥</a>

小溪\_宁静而致远© 2017

设置方法,"打开密钥文件路径":

RSA签名验签工具(V1.3)

生成密钥 签名 验签 格式转换 密钥匹配

密钥格式 :  PKCS8(JAVA适用)  PKCS1(非JAVA适用)

密钥长度 :  1024  2048

[生成密钥](#) [打开密钥文件路径](#)

**商户 应用私钥 :**  
MIIEvQIBADANBgkqhkiG9w0BAQEFAASCBKcwggsjAgEAAoIBAQCs  
hWQZF3ojVGHUbw8YiFP8DmY  
+ySzaoo6yBid1I1wx3X4HIMLE0UGS4EEFOX6Y0gZ/suhjEV7vito3HiT  
VrgCNb13EX69KOsjmeSVn2DlokmM2QVV1GOzDO8r05umpJ8yCPI  
zQk9IZcnfolghpmucp1XKnqC9aLJwzBi5woI51bisGzTlSkKzjQNL5bu4  
vgJmEMy8mFZiHalnSqosYM2HOHbzYgAmK89xoOvHCSnElK6rAOy·  
GIDdIGAnAgMBAECCggEBAJOXpL4xL90zlxQkLUq2oJUhwIQK/KWyl  
+JyQxMUv/uO6LlBpiq8UoZlyAwRCsbjXmerJKp9FV4KPe2yLP/n8jKh  
J5iT Rmp3VuVlNgGfo63NQWa6WLO  
+F7sqTIK9H4cn6aFYyHD62djLOg5H4x3bbNAQNjX/XkcYtCr9tJr0t  
rgfW0UAwpJu29A1ipYRydg7a4e8pjISzyOs1FivE0l2Syfs9+e1/mQojA  
tP8YSwpVmfbzvV2oOlkj5y795BdP3rGRtFEjvU9gUFFu8vECgYEAE3Nlk

**商户 应用公钥 :**  
+mNIGf7LoYxFe74raNx4k2WTIT4tVqws3lPCWqf1a4AjW9dxF  
+vSjrCZnkIz9g5aJJjNkFVtRjswzvK9ObpqSfMgj2zwPPA6/4uHh5ccd  
nKdVyp6gvWiycMwYucKJedW4rBs05UpCs40DS 小溪\_宁静而致远© 2017

验签步骤.txt  
 应用公钥2048.txt  
 应用私钥2048.txt 小溪\_宁静而致远© 2017

复制应用公钥2048.txt中的内容到点击"设置应用公钥"的弹出框中, 保存:

## 应用公钥(SHA256withRsa)

- ⑤ 使用SHA256withRsa，支付宝会用SHA256withRsa算法进行接口调用时的方法

```
MIIBljANBgkqhkiG9w0BAQEFAOCAQ8AMIIIBCgKCAQEAn7IQghEEJMS0.  
Pv8ks2qKOsgYndSNcMd1+ByDCxNFBkuBBBTI+mNIGf7LoYxFe74raNx4k2  
CZnklZ9g5aJJjNkFVtRjswzvK9ObpqSfMgj2zwPPaA6/4uHh5ccdbHRM0JPS  
edW4rBs05UpCs40DS+W7uA33Dg5KwJMS5EgliPUJI74CZhDMvJhWYh2iJ  
wDsulBRKKsU6mAcAOI+M2RiA3SBgJwIDAQAB
```

[验证公钥正确性>>](#)

[保存](#)  
小溪\_宁静而致远© 2017

- 商户私钥 (merchant\_private\_key)

复制 应用私钥2048.txt 中的内容到merchant\_private\_key中。

- 支付宝公钥 (alipay\_public\_key)

APPID <a href="#">i</a>	2016090900468822
支付宝网关 <a href="#">i</a>	<a href="https://openapi.alipaydev.com/gateway.do">https://openapi.alipaydev.com/gateway.do</a>
RSA2(SHA256)密钥(推荐) <a href="#">i</a>	<a href="#">查看应用公钥</a> <a href="#">查看支付宝公钥</a>

小溪\_宁静而致远© 2017

点击如上图链接，复制弹出框里面的内容到alipay\_public\_key。

如果这个设置不对，结果是：支付成功，但是验签失败。

如果是正式环境，需要上传到对应的应用中：

快速入门

教程

生成RSA密钥

上传应用公钥并获取支付...

使用应用私钥生成请求签名

使用支付宝公钥验签

自助排查

工具

FAQ

补充说明

## 上传应用公钥并获取支付宝公钥

更新时间：2017/01/04 访问次数：44807

阅读角色：支付宝账号管理者

1. 点击签名验签工具右下角的“上传公钥”会打开支付宝开放平台网页，输入账号登录。（建议使用IE或Chrome浏览器。）

2. 在“我的应用”中，选择要配置密钥的应用，点击“查看”。记录对应的APPID（下图红框处），在代码中使用。

应用名称	APPID	状态	操作
TestCenter	2016112403189534	已上线	创建服务 查看
服务窗2015051105784167	2015051100069126	已上线	创建服务 查看

小溪\_宁静而致远© 2017

### (4) 服务器异步通知页面路径 (notify\_url)

如果没有改名，修改IP和端口号就可以了，我自己的如下：

[http://localhost:8080/alipay.trade.page.pay-JAVA-UTF-8/notify\\_url.jsp](http://localhost:8080/alipay.trade.page.pay-JAVA-UTF-8/notify_url.jsp)

### (5) 页面跳转同步通知页面的路径 (return\_url)

[http://localhost:8080/alipay.trade.page.pay-JAVA-UTF-8/return\\_url.jsp](http://localhost:8080/alipay.trade.page.pay-JAVA-UTF-8/return_url.jsp)

## 4. 测试运行

支付宝电脑网站支付体验入口页

付款 交易查询 退款 退款查询 交易关闭

商户订单号：2018620154714403

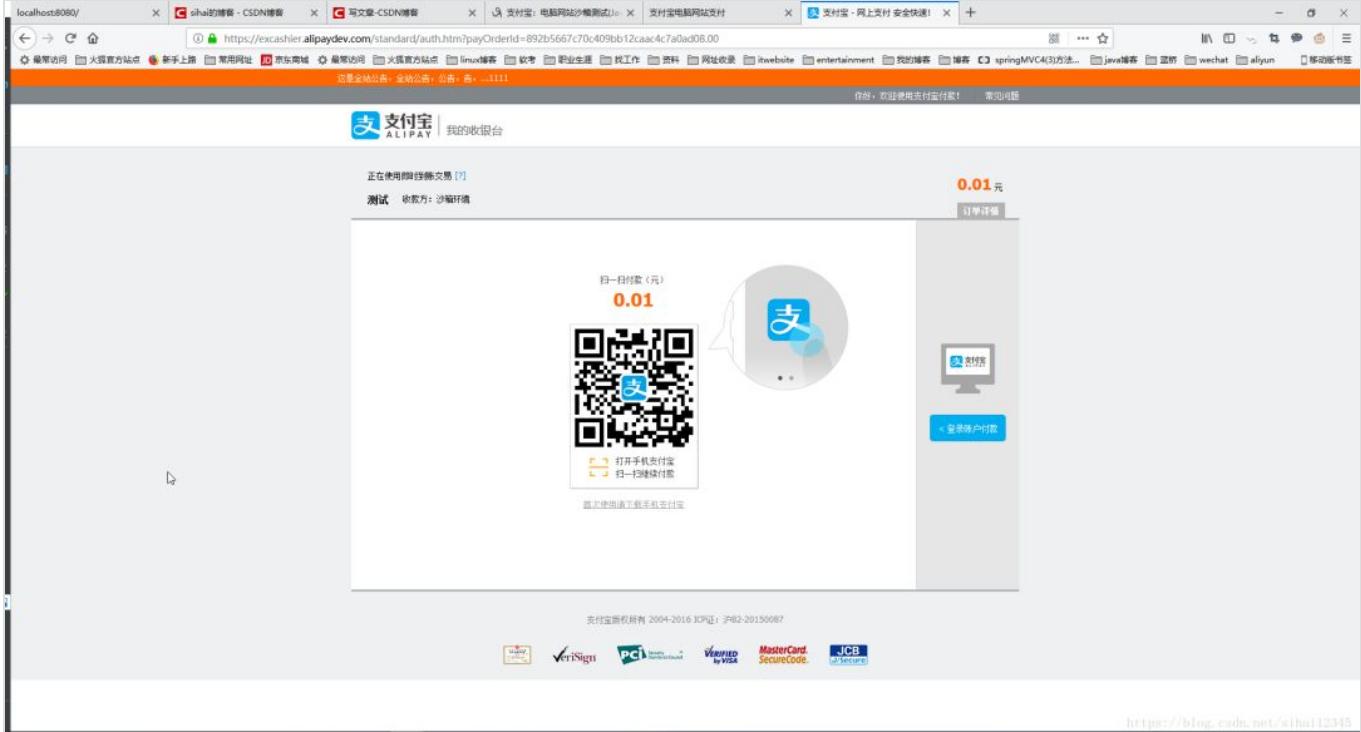
订单名称：测试

付款金额：0.01

商品描述：

付款  
如果点击付款按钮，即表示您同意该次的执行操作。

<https://blog.csdn.net/sihai12345>



测试用的支付宝买家账户可以在"沙箱账"这个页面可以找到：

开发者中心 / 研发服务 / 沙箱环境 / 沙箱应用

为保证沙箱长期稳定，每周日中午12点至每周一中午12点沙箱环境进行维护，

**沙箱应用**

接入说明

1、蚂蚁沙箱环境(Beta)是协助开发者进行接口功能开发及主要功能联调的桥逻辑以正式环境为准。

支付成功后，验签结果：

trade\_no:2017110421001004530200221733  
out\_trade\_no:2017111415342964  
total\_amount:0.01

## 问题解决

由于我们使用的是沙箱测试环境，测试环境和正式上线的环境的网关是不一样的，如果配置错误，会出现，appid错误的问题。

配置如下：

```

1 package com.sihai.utils;
2
3 public class AlipayConfig {
4
5     public static String app_id = "2016091400505963";
6
7     public static String merchant_private_key = "M1EvaIBADANBgkqhkiG9w0BAQEFAASCBKYwggSiAgEAAoIBAQCny91HUPfwEx70VI8FbtWz4YWPY2XGKrT29+1";
8
9     public static String alipay_public_key = "MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMITBCgKCAQEAcjcdR1BX8BMe9FSPBW7Vs+GFhctlxig02ffpQmtJmLpVfs";
10
11    public static String notify_url = "http://localhost:8080/alipay/alipayNotifyNotice.action";
12
13    public static String return_url = "http://localhost:8080/alipay/alipayReturnNotice.action";
14
15    public static String sign_type = "RSA2";
16
17    public static String charset = "utf-8";
18
19    public static String gatewayUrl = "https://openapi.alipaydev.com/gateway.do"; // 阿里的配置是这个地址，而不是正式的上线的 https://openapi.alipay.com/gateway.do
20 }

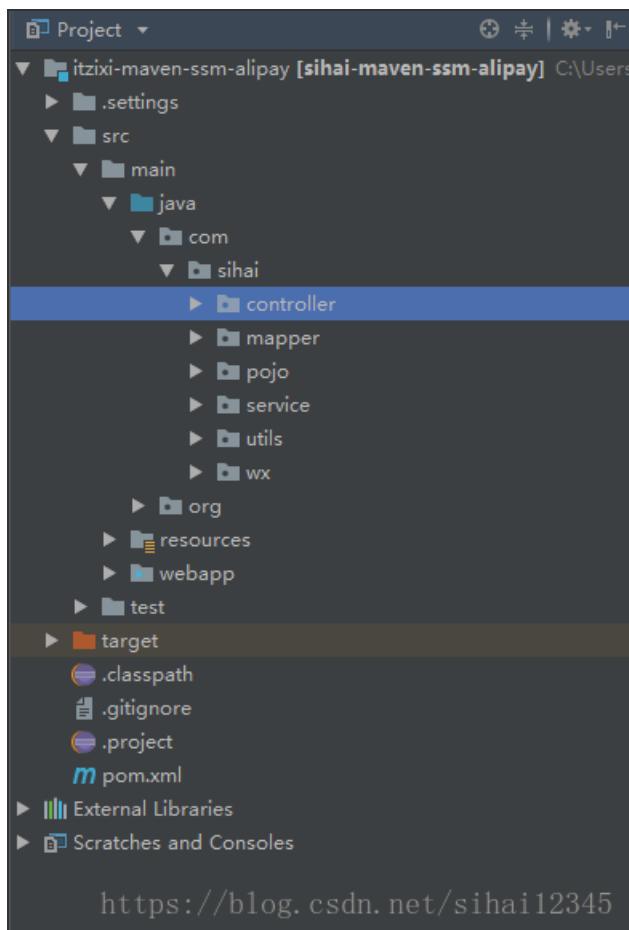
```

<https://blog.csdn.net/sihai12345>

源代码下载，可以关注微信公众号 Java后端 回复扫码即可获取。

## 将支付宝支付整合到ssm框架

### 1、项目架构



- 项目架构：spring+springmvc+mybatis
- 数据库：mysql
- 部署环境：tomcat9.0
- 开发环境：jdk9、idea
- 支付：支付宝、微信

整合到ssm一样，我们需要像沙箱测试环境一样，需要修改支付的配置信息

```
package com.sihai.utils;

public class AlipayConfig {

    public static String app_id = "2016091400505963";

    public static String merchant_private_key = "MIIEvAIBADANBgkqhkiG9w0BAQEFAASCBKYwggSiAgEAAoIBAQCNy91HUFwEx70V18F6tWz4YWPY2XGKrT29+1Ca6...";

    public static String alipay_public_key = "MIIBIjANBkghkiG9w0BAQEFAOCg8AMIIBCgKCAQEAjcdR1BXSBMe9FSFBW7Vs+GFhctlxigO2ffpQmtJmLpVf...";

    public static String notify_url = "http://localhost:8080/alipay/alipayNotifyNotice.action";

    public static String return_url = "http://localhost:8080/alipay/alipayReturnNotice.action";

    public static String sign_type = "RSA2";

    public static String charset = "utf-8";

    public static String gatewayUrl = "https://openapi.alipaydev.com/gateway.do";
}
```

对讲机的聊天地址，alipaydev  
是测试环境，alipay，没有用。如果不注意会去到appid错误的问题

<https://blog.csdn.net/sihai12345>

## 2、数据库代码

主要包括以下的数据库表：

- user：用户表
- order：支付产生的订单
- flow：流水账
- product：商品表：用于模拟购买商品。

```

drop table if exists user;

/*=====
/* Table: user */
=====*/
create table user
(
    id      varchar(20) not null,
    username varchar(128),
    sex varchar(20),
    primary key (id)
);

alter table user comment '用户表';

CREATE TABLE `flow` (
    `id` varchar(20) NOT NULL,
    `flow_num` varchar(20) DEFAULT NULL COMMENT '流水号',
    `order_num` varchar(20) DEFAULT NULL COMMENT '订单号',
    `product_id` varchar(20) DEFAULT NULL COMMENT '产品主键ID',
    `paid_amount` varchar(11) DEFAULT NULL COMMENT '支付金额',
    `paid_method` int(11) DEFAULT NULL COMMENT '支付方式\r\n1: 支付宝\r\n2: 微信',
    `buy_counts` int(11) DEFAULT NULL COMMENT '购买个数',
    `create_time` datetime DEFAULT NULL COMMENT '创建时间',
    PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COMMENT='流水表';

CREATE TABLE `orders` (
    `id` varchar(20) NOT NULL,
    `order_num` varchar(20) DEFAULT NULL COMMENT '订单号',
    `order_status` varchar(20) DEFAULT NULL COMMENT '订单状态\r\n10: 待付款\r\n20: 已付款',
    `order_amount` varchar(11) DEFAULT NULL COMMENT '订单金额',
    `paid_amount` varchar(11) DEFAULT NULL COMMENT '实际支付金额',
    `product_id` varchar(20) DEFAULT NULL COMMENT '产品表外键ID',
    `buy_counts` int(11) DEFAULT NULL COMMENT '产品购买的个数',
    `create_time` datetime DEFAULT NULL COMMENT '订单创建时间',
    `paid_time` datetime DEFAULT NULL COMMENT '支付时间',
    PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COMMENT='订单表';

CREATE TABLE `product` (
    `id` varchar(20) NOT NULL,
    `name` varchar(20) DEFAULT NULL COMMENT '产品名称',
    `price` varchar(11) DEFAULT NULL COMMENT '价格',
    PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COMMENT='产品表';

```

### 3、dao数据接口层

这里就不介绍了，这个只包括简单的curd，可以使用`通用mapper`，或者`逆向工程`就行。以订单order为例给出：

```

public interface OrdersMapper {
    int countByExample(OrdersExample example);

    int deleteByExample(OrdersExample example);

    int deleteByPrimaryKey(String id);

    int insert(Orders record);

    int insertSelective(Orders record);

    List<Orders> selectByExample(OrdersExample example);

    Orders selectByPrimaryKey(String id);

    int updateByExampleSelective(@Param("record") Orders record, @Param("example") OrdersExample example);

    int updateByExample(@Param("record") Orders record, @Param("example") OrdersExample example);

    int updateByPrimaryKeySelective(Orders record);

    int updateByPrimaryKey(Orders record);
}

```

#### 4、service层

同上，最后在项目源代码里可见。以订单order为例给出：

```

/**
 * 订单操作 service
 * @author ibm
 *
 */
public interface OrdersService {

    /**
     * 新增订单
     * @param order
     */
    public void saveOrder(Orders order);

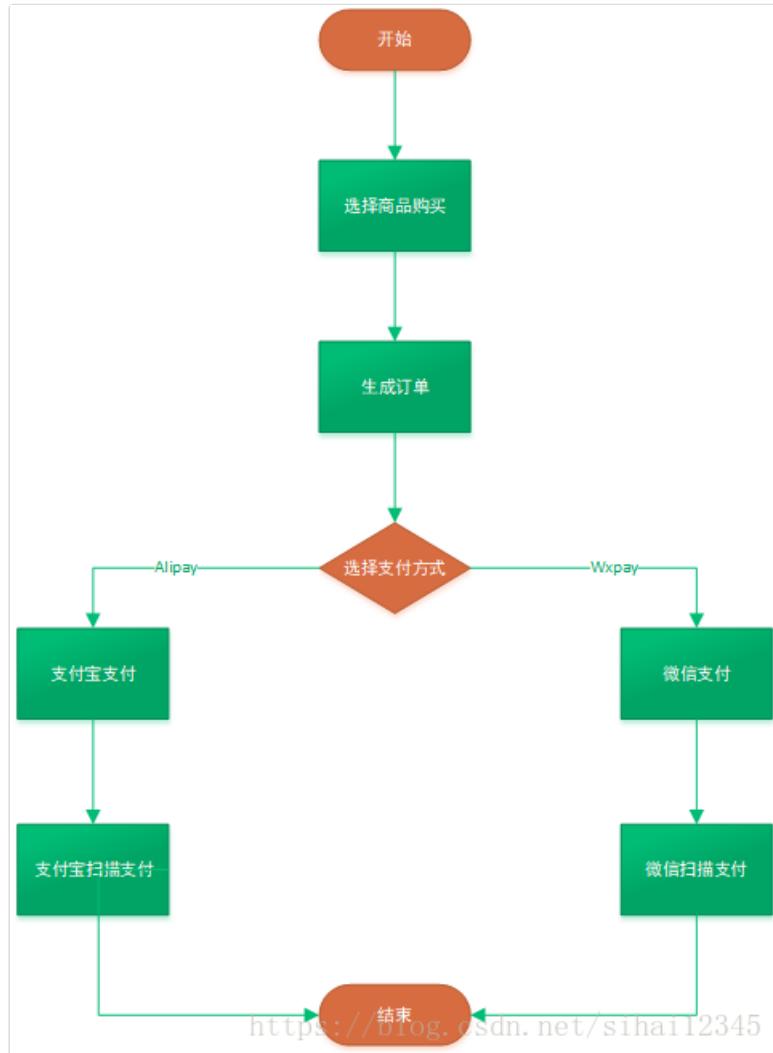
    /**
     *
     * @Title: OrdersService.java
     * @Package com.sihai.service
     * @Description: 修改叮当状态，改为 支付成功，已付款;同时新增支付流水
     * Copyright: Copyright (c) 2017
     * Company:FURUIBOKE.SCIENCE.AND.TECHNOLOGY
     *
     * @author sihai
     * @date 2017年8月23日 下午9:04:35
     * @version V1.0
     */
    public void updateOrderStatus(String orderId, String alipayFlowNum, String paidAmount);

    /**
     * 获取订单
     * @param orderId
     * @return
     */
    public Orders getOrderById(String orderId);
}

```

## 4、支付宝支付controller（支付流程）

支付流程图



首先，启动项目后，输入http://localhost:8080/，会进入到商品页面，如下：

A screenshot of a web browser showing the product list page. The URL in the address bar is 'localhost:8080'. The page displays a table with two products:

产品编号	产品名称	产品价格	操作
1	辣条	0.01	<a href="#">购买</a>
2	面包	0.02	<a href="#">购买</a>

At the bottom right of the page, there is a link: 'https://blog.csdn.net/sihai12345'.

下面是页面代码

商品页面 (products.jsp)

A screenshot of a file explorer showing the project structure. The 'WEB-INF' folder contains a 'jsp' folder which includes several JSP files: alipaySuccess.jsp, goConfirm.jsp, goPay.jsp, payQrCode.jsp, paySuccess.jsp, products.jsp (highlighted in blue), and user.jsp. Below the 'WEB-INF' folder is a 'web.xml' file. At the bottom left, the URL 'https://blog.csdn.net/sihai12345' is shown.

代码实现：

```

<html>
<body>
<table>
<tr>
<td>产品编号</td>
<td>产品名称</td>
<td>产品价格</td>
<td>操作</td>
</tr>
<c:forEach items="${pList }" var="p">
<tr>
<td>${p.id }</td>
<td>${p.name }</td>
<td>${p.price } </td>
<td>
<a href="<%request.getContextPath()%>/alipay/goConfirm.action?productId=${p.id }">购买</a>
</td>
</tr>
</c:forEach>
</table>
<input type="hidden" id="hdnContextPath" name="hdnContextPath" value="<%request.getContextPath()%>" />
</body>

</html>

<script type="text/javascript">

$(document).ready(function() {
    var hdnContextPath = $("#hdnContextPath").val();
});

</script>

```

点击上面的购买，进入到订单页面



填写个数，然后点击生成订单，调用如下代码

```

/**
 * 分段提交
 * 第一段：保存订单
 * @param order
 * @return
 * @throws Exception
 */
@RequestMapping(value = "/createOrder")
@ResponseBody
public LeeJSONResult createOrder(Orders order) throws Exception {
    Product p = productService.getProductById(order.getProductId());
    String orderId = sid.nextShort();
    order.setId(orderId);
    order.setOrderNum(orderId);
    order.setCreateTime(new Date());
    order.setOrderAmount(String.valueOf(Float.valueOf(p.getPrice()) * order.getBuyCounts()));
    order.setOrderStatus(OrderStatusEnum.WAIT_PAY.key());
    orderService.saveOrder(order);

    return LeeJSONResult.ok(orderId);
}

```

https://blog.csdn.net/sihai12345

根据 **SID** (生成id的工具) 等信息生成订单，保存到数据库。

Tips：可以关注微信公众号：Java后端，获取更多类似技术博文推送。

### 进入到选择支付页面



调用了如下代码：

```

/*
 * 分段提交
 * 第二段
 * @param orderId
 * @return
 * @throws Exception
 */

@RequestMapping(value = "/goPay")
public ModelAndView goPay(String orderId) throws Exception {

    Orders order = orderService.getOrderByid(orderId);

    Product p = productService.getProductById(order.getId());

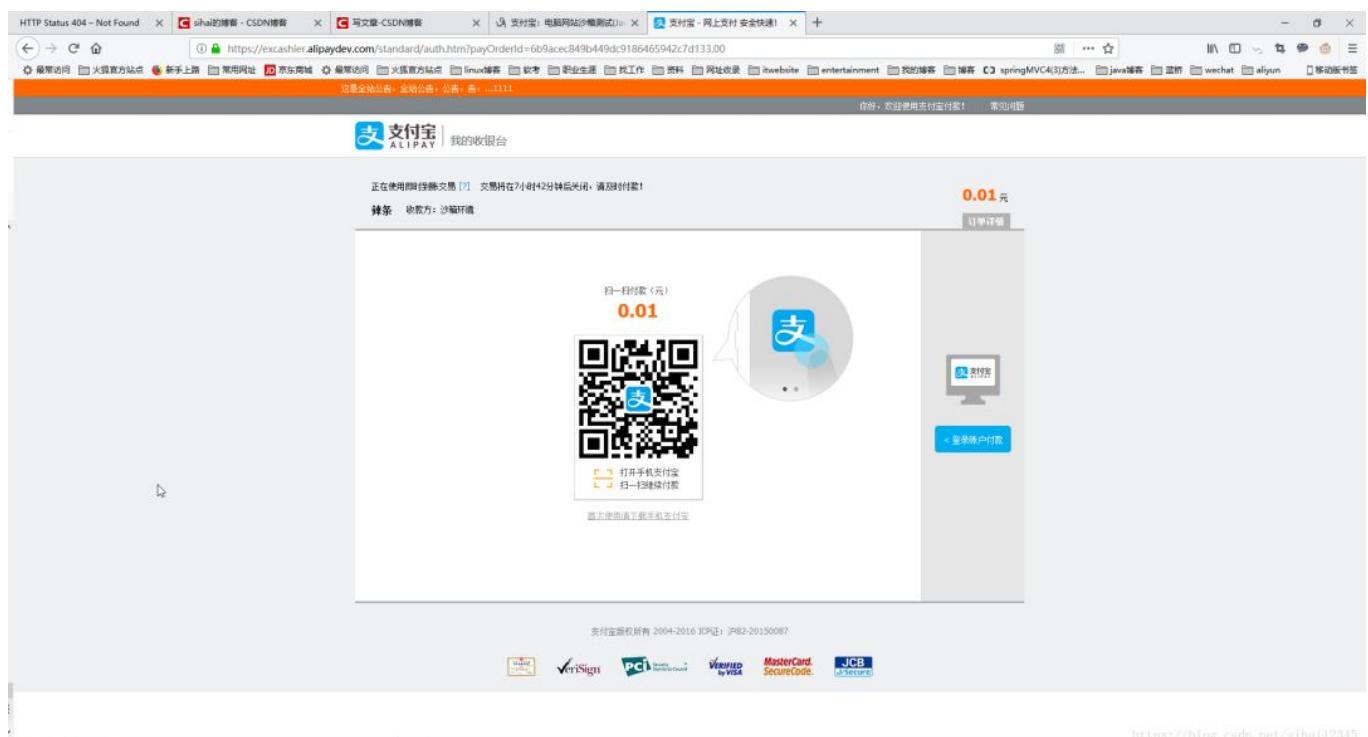
    ModelAndView mv = new ModelAndView( viewName: "goPay");
    mv.addObject( attributeName: "order", order);
    mv.addObject( attributeName: "p", p);

    return mv;
}

```

<https://blog.csdn.net/sihai12345>

然后，我们选择支付宝支付，进入到了我们支付的页面了，大功告成！



调用了如下代码：

```

@RequestMapping(value = "/goAlipay", produces = "text/html; charset=UTF-8")
@ResponseBody
public String goAlipay(String orderId, HttpServletRequest request, HttpServletResponse response) throws Exception {

    Orders order = orderService.getOrderByid(orderId);

    Product product = productService.getProductById(order.getProductId());

    //获得初始化的AlipayClient
    AlipayClient alipayClient =new DefaultAlipayClient(AlipayConfig.gatewayUrl, AlipayConfig.app_id, AlipayConfig.merchant_private_key, "json");

    //设置请求参数
    AlipayTradePagePayRequest alipayRequest = new AlipayTradePagePayRequest();
    alipayRequest.setReturnUrl(AlipayConfig.return_url);
    alipayRequest.setNotifyUrl(AlipayConfig.notify_url);

    //商户订单号，商户网站订单系统中唯一订单号，必填
    String out_trade_no = orderId;
    //付款金额，必填
    String total_amount = order.getOrderAmount();
    //订单名称，必填
    String subject = product.getName();
    //商品描述，可空
    String body = "用户订购商品个数：" + order.getBuyCounts();

    //该笔订单允许的最晚付款时间，逾期将关闭交易。取值范围：1m~15d。m-分钟，h-小时，d-天，1c-当天（1c-当天的情况下，无论交易何时到达支付宝，都在1分钟内关闭），t-默认值，即当天23：59：59点自动关闭。
    String timeout_express = "1c";

    alipayRequest.setBizContent("{\"out_trade_no\":\""+ out_trade_no +"\",
        + "\total_amount\":\""+ total_amount +"\",
        + "\subject\":\""+ subject +"\",
        + "\body\":\""+ body +"\",
        + "\timeout_express\":\""+ timeout_express +"\",
        + "\product_code\":\"FAST_INSTANT_TRADE_PAY\"}");

    //请求
    String result = alipayClient.pageExecute(alipayRequest).getBody();

    return result;
}

```

这段代码都可以在阿里支付的demo里面找到的，只需要复制过来，然后改改，整合到ssm环境即可。

上面就是将支付宝支付整合到ssm的全过程了。本文参考了博主「小溪\_宁静而致远」的支付教程文章，欢迎搜索博主关注。

---

- END -

如果看到这里，说明你喜欢这篇文章，请[转发](#)、[点赞](#)。微信搜索「web\_resource」，关注后回复「进群」或者扫描下方二维码即可进入无广告交流群。

[↓ 扫描二维码进群 ↓](#)



扫一扫上面的二维码图案，加我微信

#### 推荐阅读

1. Spring Boot + PageHelper 实现多数据源并分页
2. Spring Boot 微信点餐系统
3. Spring MVC 到 Spring Boot 的简化之路
4. 12306 的架构到底有多牛逼？
5. 团队开发中 Git 最佳实践



[阅读原文](#)

声明：pdf仅供学习使用，一切版权归原创公众号所有；建议持续关注原创公众号获取最新文章，学习愉快！

# 【MyBatis】写了 10 年的代码，我最怕写 MyBatis 这些配置，现在有详解了

Java后端 2月28日



作者 | 阿进的写字台

编辑 | 公众号 (Java后端)

链接 | [cnblogs.com/homejim/p/9782403.html](http://cnblogs.com/homejim/p/9782403.html)

在使用 **mybatis** 过程中，当手写 **JavaBean** 和 **XML** 写的越来越多的时候，就越来越容易出错。这种重复性的工作，我们当然不希望做那么多。

还好，**mybatis** 为我们提供了强大的代码生成--**MybatisGenerator**。

通过简单的配置，我们就可以生成各种类型的实体类，Mapper接口，MapperXML文件，Example对象等。通过这些生成的文件，我们就可以方便的进行单表进行增删改查的操作。

Tips: 关注微信公众号：Java后端，获取每日推送。

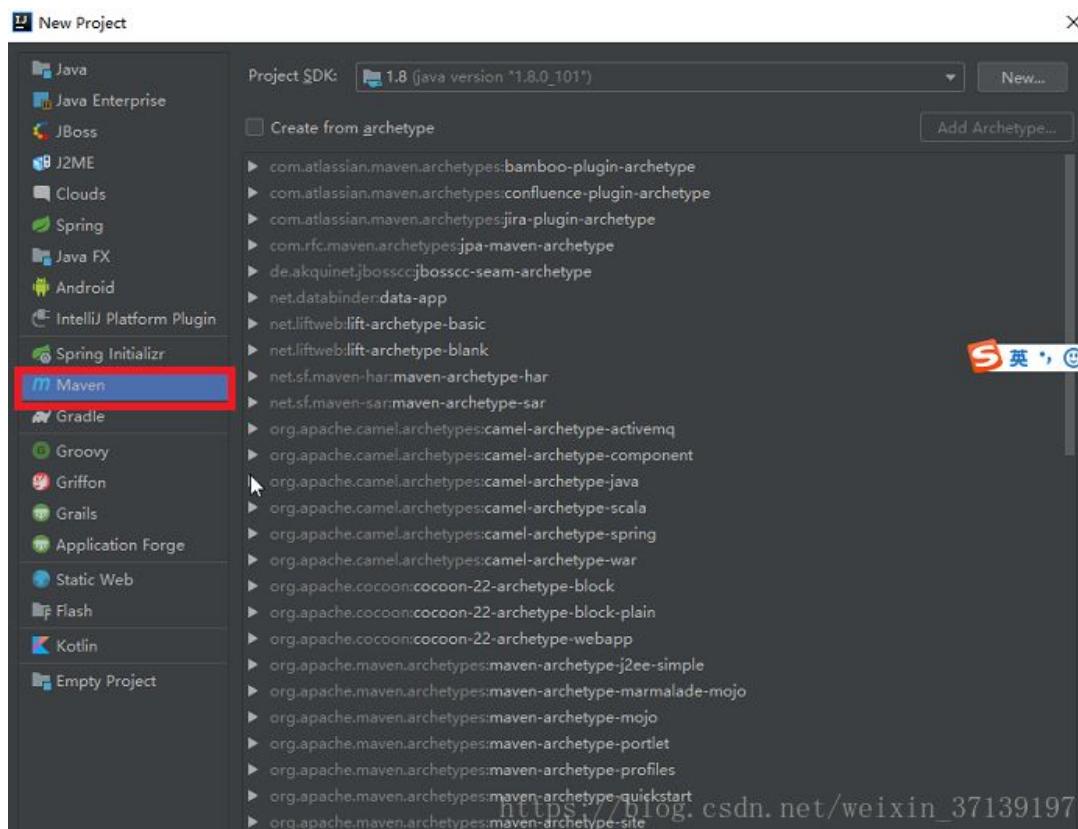
以下的工具使用的都是 **IDEA**

## 1.1 创建Maven项目

### 1.1.1 菜单上选择新建项目



### 1.1.2 选择左侧的Maven



由于我们只是创建一个普通的项目，此处点击 Next即可。

### 1.1.3 输入GroupId和ArtifactId

- 在我的项目中，

GroupId 填 com.homejim.mybatisplus

ArtifactId 填 mybatis-generator

点击 Next。

### 1.1.4 Finish

通过以上步骤，一个普通的Maven项目就创建好了。

## 1.2 配置 generator.xml

其实名字无所谓，只要跟下面的 **pom.xml** 文件中的对应上就好了。

```

<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE generatorConfiguration PUBLIC
"-//mybatis.org//DTD MyBatis Generator Configuration 1.0//EN"
"http://mybatis.org/dtd/mybatis-generator-config_1_0.dtd">
<generatorConfiguration>

<classPathEntry location="C:\Users\Administrator\.m2\repository\mysql\mysql-connector-java\8.0.12\mysql-connector-java-8.0.12.jar" />
<context id="context" targetRuntime="MyBatis3">
    <commentGenerator>
        <property name="suppressAllComments" value="false"/>
        <property name="suppressDate" value="true"/>
    </commentGenerator>

    <jdbcConnection
        driverClass="com.mysql.jdbc.Driver"
        connectionURL="jdbc:mysql://localhost:3306/mybatis"
        userId="root"
        password="jim777"/>

    <javaTypeResolver>
        <property name="forceBigDecimals" value="false"/>
    </javaTypeResolver>

    <javaModelGenerator
        targetPackage="com.homejim.mybatis.entity"
        targetProject=".src\main\java">
        <property name="enableSubPackages" value="false"/>
        <property name="trimStrings" value="true"/>
    </javaModelGenerator>

    <sqlMapGenerator
        targetPackage="mybatis/mapper"
        targetProject=".src\main\resources">
        <property name="enableSubPackages" value="false"/>
    </sqlMapGenerator>

    <javaClientGenerator type="XMLMAPPER"
        targetPackage="com.homejim.mybatis.mapper"
        targetProject=".src\main\java">
        <property name="enableSubPackages" value="false"/>
    </javaClientGenerator>

    <table tableName="blog" />
</context>
</generatorConfiguration>

```

需要改一些内容：

1. 本地数据库驱动程序jar包的全路径（**必须要改**）。
2. 数据库的相关配置（**必须要改**）
3. 相关表的配置（**必须要改**）
4. 实体类生成存放的位置。
5. MapperXML 生成文件存放的位置。
6. Mapper 接口存放的位置。

如果不知道怎么改，请看后面的[配置详解](#)。

## 1.3 配置 pom.xml

在原基础上添加一些内容。

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
<modelVersion>4.0.0</modelVersion>

<groupId>com.homejim.mybatisplus</groupId>
<artifactId>mybatis-generator</artifactId>
<version>1.0-SNAPSHOT</version>

<build>
  <finalName>mybatis-generator</finalName>
  <plugins>
    <plugin>
      <groupId>org.mybatis.generator</groupId>
      <artifactId>mybatis-generator-maven-plugin</artifactId>
      <version>1.3.7</version>
      <configuration>

        <configurationFile>src/main/resources/generator.xml</configurationFile>
        <verbose>true</verbose>
        <overwrite>true</overwrite>
      </configuration>
    <executions>
      <execution>
        <id>Generate MyBatis Artifacts</id>
        <goals>
          <goal>generate</goal>
        </goals>
      </execution>
    </executions>
    <dependencies>
      <dependency>
        <groupId>org.mybatis.generator</groupId>
        <artifactId>mybatis-generator-core</artifactId>
        <version>1.3.7</version>
      </dependency>
    </dependencies>
  </plugin>
</plugins>
</build>

</project>
```

需要注意的是 **configurationFile** 中的文件指的是 **generator.xml**。因此路径写的是该文件的相对路径，名称也跟该文件相同。

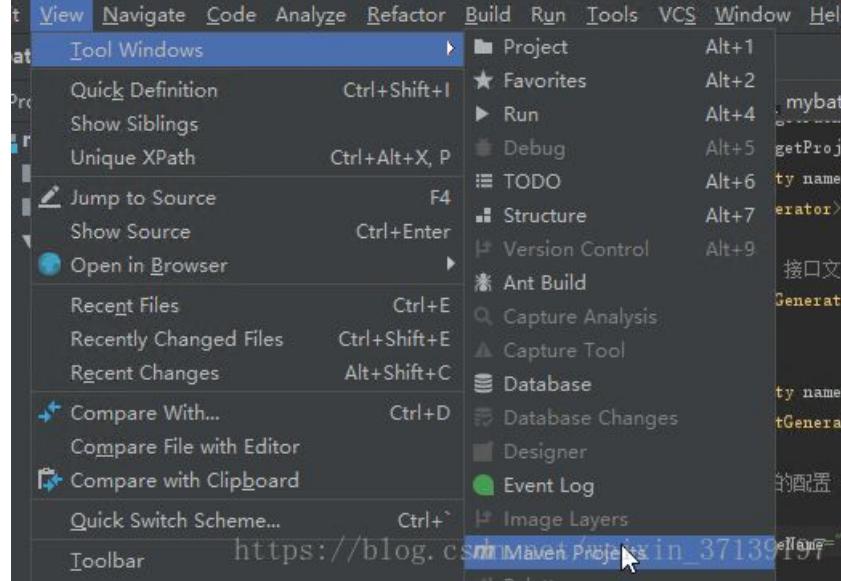
到此， **mybatis-generator** 就可以使用啦。

## 1.4 使用及测试

### 1.4.1 打开 Maven Projects 视图

在 IDEA 上，打开：

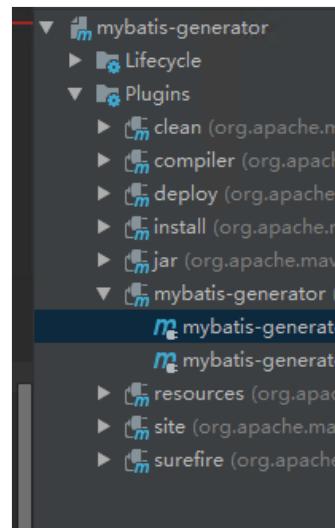
View | Tools | Windwos | Maven Projects



#### 1.4.2 Maven Projects 中双击 mybatis-generator

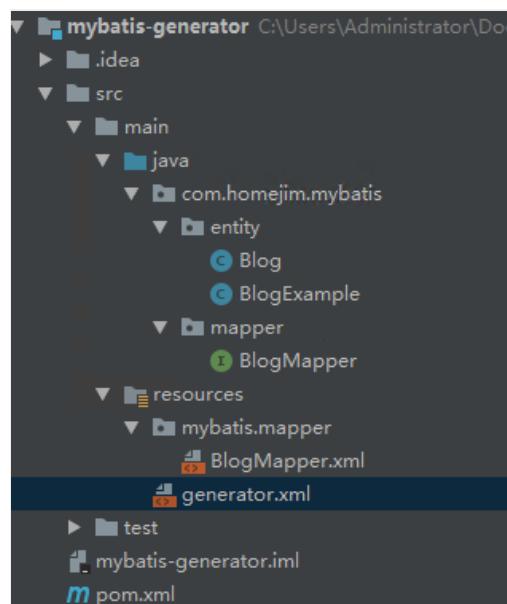
在右侧此时可以看到 Maven Projects 了。找到 mybatis-generator 插件。

mybatis-generator | Plugins | mybatis-generator | mybatis-generator



#### 1.4.3 双击运行

运行正确后，生成代码，得到如下的结构



仅仅是上面那么简单的使用还不够爽。那么我们就可以通过更改 generator.xml 配置文件的方式进行生成的配置。

## 2.1 文档

推荐查看官方的文档。

英文不错的：<http://www.mybatis.org/generator/configreference/xmlconfig.html>

中文翻译版：<http://mbg.cndocs.ml/index.html>

## 2.2 官网没有的

### 2.2.1 property 标签

该标签在官网中只是说用来指定元素的属性，至于怎么用没有详细的讲解。

#### 2.2.1.1 分隔符相关

```
<property name="autoDelimitKeywords" value="true"/>
<property name="beginningDelimiter" value="`"/>
<property name="endingDelimiter" value="`"/>
```

以上的配置对应的是 mysql，当数据库中的字段和数据库的关键字一样时，就会使用分隔符。

比如我们的数据列是 delete，按以上的配置后，在它出现的地方，就变成 `delete`。

#### 2.2.1.2 编码

默认是使用当前的系统环境的编码，可以配置为 GBK 或 UTF-8。

```
<property name="javaFileEncoding" value="UTF-8"/>
```

我想项目为 UTF-8，如果指定生成 GBK，则自动生成的中文就是乱码。

#### 2.2.1.3 格式化

```
<property name="javaFormatter" value="org.mybatis.generator.api.dom.DefaultJavaFormatter"/>
<property name="xmlFormatter" value="org.mybatis.generator.api.dom.DefaultXmlFormatter"/>
```

这些显然都是可以自定义实现的。

### 2.2.2 plugins 标签

plugins 标签用来扩展或修改代码生成器生成的代码。

在生成的 XML 中，是没有 **<cache>** 这个标签的。该标签是配置缓存的。

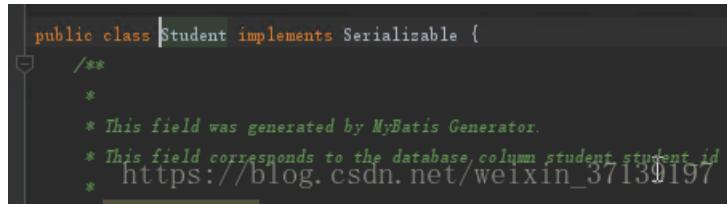
如果我们想生成这个标签，那么可以 **plugins** 中进行配置。

```
<plugin type="org.mybatis.generator.plugins.CachePlugin" >
    <property name="cache_eviction" value="LRU"/>
</plugin>
```

```
<cache eviction="LRU">
<!--
  WARNING - @mbg generated
  This element is automatically generated by MyBatis Generator, do not modify.
-->
https://blog.csdn.net/weixin_37139197
</cache>
```

比如你想生成的 **JavaBean** 中自行实现 **Serializable** 接口。

```
<plugin type="org.mybatis.generator.plugins.SerializablePlugin" />
```



还能自定义插件。

这些插件都蛮有用的，感觉后续可以专门开一篇文章来讲解。

看名称，就知道是用来生成注释用的。

默认配置：

```
<commentGenerator>
<property name="suppressAllComments" value="false"/>
<property name="suppressDate" value="false"/>
<property name="addRemarkComments" value="false"/>
</commentGenerator>
```

suppressAllComments：阻止生成注释， 默认值是false。

suppressDate: 阻止生成的注释包含时间戳， 默认为false。

addRemarkComments: 注释中添加数据库的注释， 默认为 false。

还有一个就是我们可以通过 **type** 属性指定我们自定义的注解实现类，生成我们自己想要的注解。自定义的实现类需要实现 **org.mybatis.generator.api.CommentGenerator**。

## 2.2.4 源码

<https://github.com/homejim/mybatis-cn>

homejim / mybatis-cn

Code Issues 0 Pull requests 0 Projects 0 Wiki Insights Settings

mybatis mybatis-sources mybatis-chinese Manage topics

83 commits 1 branch 0 releases 1 contributor View license

Create new file Upload files Find file Clone or download

本文作者：阿进的写字台，原文链接：[cnblogs.com/homejim/p/9782403.html](http://cnblogs.com/homejim/p/9782403.html)

由公众号（Java后端）编辑，转载请著名本句话及原文出处。

如果看到这里，说明你喜欢这篇文章，请[转发、点赞](#)。微信搜索「web\_resource」，欢迎添加小编微信「focusoncode」，每日朋友圈更新一篇高质量技术博文(无广告)。

↓ 扫描二维码添加小编 ↓



#### 推荐阅读

1. 浅谈 Web 网站架构演变过程
2. 一个非常实用的特性，很多人没用过
3. Spring Boot 2.x 操作缓存的新姿势
4. MyBatis 中 SQL 写法技巧小总结



微信搜一搜

Java后端

声明：pdf仅供学习使用，一切版权归原创公众号所有；建议持续关注原创公众号获取最新文章，学习愉快！

# 【小技巧】MyBatis 中 SQL 写法技巧小总结

koala Java后端 2月27日



最近有个兄弟在搞mybatis，问我怎么写sql，说简单一点mybatis就是写原生sql，官方都说了 mybatis 的动态sql语句是基于OGNL表达式的。可以方便的在 sql 语句中实现某些逻辑。总体说来mybatis 动态SQL 语句主要有以下几类：

1. if 语句 (简单的条件判断)
2. choose (when,otherwise) ,相当于java 语言中的 switch ,与 jstl 中的choose 很类似.
3. trim (对包含的内容加上 prefix,或者 suffix 等, 前缀, 后缀)
4. where (主要是用来简化sql语句中where条件判断的, 能智能的处理 and or ,不必担心多余导致语法错误)
5. set (主要用于更新时)
6. foreach (在实现 mybatis in 语句查询时特别有用)

我说一下：if when 都仅仅对于Map类型才能进行判断，Integer, String 那些都不能进行判断，虽然说mybatis最后都是把参数封装为一个Map集合再通过占位符接入的。另一个就是trim很强大，可以说where和set能干的他都能干。choose在进行”单个“模糊查询时候很方便。

往期优质技术文章可以关注微信公众号：Java后端，后台回复 技术博文 获取。

下面分别介绍这几种处理方式

## 1. mybatis if 语句处理

```
<select id="dynamicIfTest" parameterType="Blog" resultType="Blog">
    select * from t_blog where 1 = 1
    <if test="title != null">
        and title = #{title}
    </if>
    <if test="content != null">
        and content = #{content}
    </if>
    <if test="owner != null">
        and owner = #{owner}
    </if>
</select>
```

这条语句的意思非常简单，如果你提供了title参数，那么就要满足title=#{title}，同样如果你提供了Content和Owner的时候，它们也需要满足相应的条件，之后就是返回满足这些条件的所有Blog，这是非常有用的一个功能，以往我们使用其他类型框架或者直接使用JDBC的时候，如果我们要达到同样的选择效果的时候，我们就需要拼SQL语句，这是极其麻烦的，比起来，上述的动态SQL就要简单多了。

## 2. choose (when,otherwise) ,相当于java 语言中的 switch ,与 jstl 中的choose 很类似

```

<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>
            and owner = "owner1"
        </otherwise>
    </choose>
</select>

```

when元素表示当when中的条件满足的时候就输出其中的内容，跟JAVA中的switch效果差不多的是按照条件的顺序，当when中有条件满足的时候，就会跳出choose，即所有的when和otherwise条件中，只有一个会输出，当所有的我很条件都不满足的时候就输出otherwise中的内容。所以上述语句的意思非常简单，当title!=null的时候就输出and title = #{title}，不再往下判断条件，当title为空且content!=null的时候就输出and content = #{content}，当所有条件都不满足的时候就输出otherwise中的内容。

### 3.trim (对包含的内容加上 prefix,或者 suffix 等，前缀，后缀)

```

<select id="dynamicTrimTest" parameterType="Blog" resultType="Blog">
    select * from t_blog
    <trim prefix="where" prefixOverrides="and |or">
        <if test="title != null">
            title = #{title}
        </if>
        <if test="content != null">
            and content = #{content}
        </if>
        <if test="owner != null">
            or owner = #{owner}
        </if>
    </trim>
</select>

```

后缀，与之对应的属性是prefix和suffix；可以把包含内容的首部某些内容覆盖，即忽略，也可以把尾部的某些内容覆盖，对应的属性是prefixOverrides和suffixOverrides；正因为trim有这样的功能，所以我们也可以非常的简单的利用trim来代替where元素的功能。

### 4. where (主要是用来简化sql语句中where条件判断的，能智能的处理 and or 条件)

```

<select id="dynamicWhereTest" parameterType="Blog" resultType="Blog">
    select * from t_blog
    <where>
        <if test="title != null">
            title = #{title}
        </if>
        <if test="content != null">
            and content = #{content}
        </if>
        <if test="owner != null">
            and owner = #{owner}
        </if>
    </where>
</select>

```

where元素的作用是会在写入where元素的地方输出一个where，另外一个好处是你不需要考虑where元素里面的条件输出是什么样子的，MyBatis会智能的帮你处理，如果所有的条件都不满足那么MyBatis就会查出所有的记录，如果输出后是and 开头的，MyBatis会把第一个and忽略，当然如果是or开头的，MyBatis也会把它忽略；此外，在where元素中你不需要考虑空格的问题，MyBatis会智能的帮你加上。像上述例子中，如果title=null，而content != null，那么输出的整个语句会是select \* from t\_blog where content = #{content}，而不是select \* from t\_blog where and content = #{content}，因为MyBatis会智能的把首个and 或 or 给忽略。

## 5.set (主要用于更新时)

```

<update id="dynamicSetTest" parameterType="Blog">
    update t_blog
    <set>
        <if test="title != null">
            title = #{title},
        </if>
        <if test="content != null">
            content = #{content},
        </if>
        <if test="owner != null">
            owner = #{owner}
        </if>
    </set>
    where id = #{id}
</update>

```

set元素主要是用在更新操作的时候，它的主要功能和where元素其实是差不多的，主要是在包含的语句前输出一个set，然后如果包含的语句是以逗号结束的话将会把该逗号忽略，如果set包含的内容为空的话则会出错。有了set元素我们就可以动态的更新那些修改了的字段。

## 6. foreach (在实现 mybatis in 语句查询时特别有用)

foreach的主要用在构建in条件中，它可以在SQL语句中进行迭代一个集合。foreach元素的属性主要有item, index, collection, open, separator, close。item表示集合中每一个元素进行迭代时的别名，index指定一个名字，用于表示在迭代过程中，每次迭代到的位置，open表示该语句以什么开始，separator表示在每次进行迭代之间以什么符号作为分隔符，close表示以什么结束，在使用foreach的时候最关键的也是最容易出错的就是collection属性，该属性是必须指定的，但是在不同情况下，该属性的值是不一样的，主要有一下3种情况：

- 如果传入的是单参数且参数类型是一个List的时候，collection属性值为list。
- 如果传入的是单参数且参数类型是一个array数组的时候，collection的属性值为array。
- 如果传入的参数是多个的时候，我们就需要把它们封装成一个Map了，当然单参数也可以封装成map，实际上如果你在传入参数的时候，在MyBatis里面也是会把它封装成一个Map的，map的key就是参数名，所以这个时候collection属性值就是传入的List或array对象在自己封装的map里面的key。

作者：牧羊人影视

原文链接：<https://blog.csdn.net/tengxing007/article/details/54864897>

-END-

如果看到这里，说明你喜欢这篇文章，请[转发、点赞](#)。微信搜索「web\_resource」，欢迎添加小编微信「focusoncode」，每日朋友圈更新一篇高质量技术博文（无广告）。

↓ 扫描二维码添加小编 ↓



#### 推荐阅读

1. 狠！删库跑路！
2. 分布式与集群的区别究竟是什么？
3. 看完这篇 HTTP，面试官就难不倒你了
4. 快速建站利器！



微信搜一搜

Q Java后端

声明：pdf仅供学习使用，一切版权归原创公众号所有；建议持续关注原创公众号获取最新文章，学习愉快！

# 从 0 开始手写一个 Mybatis 框架，三步搞定！

我叫刘半仙 Java后端 1月9日

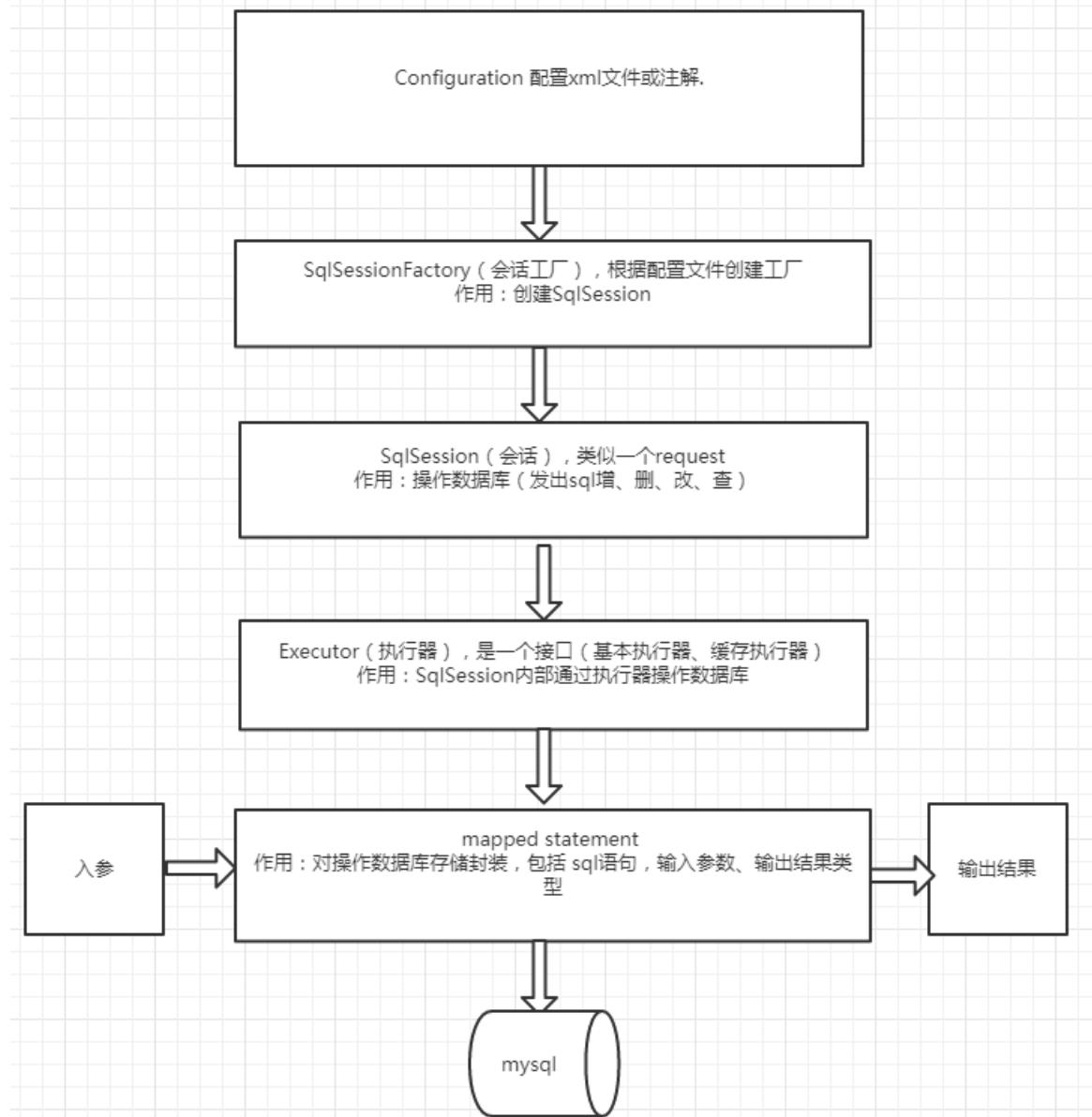


来自：开源中国，作者：我叫刘半仙

链接：[my.oschina.net/liughDevelop/blog/1631006](http://my.oschina.net/liughDevelop/blog/1631006)

继上一篇手写SpringMVC之后，我最近趁热打铁，研究了一下Mybatis。MyBatis框架的核心功能其实不难，无非就是动态代理和jdbc的操作，难的是写出来可扩展，高内聚，低耦合的规范的代码。本文完成的Mybatis功能比较简单，代码还有许多需要改进的地方，大家可以结合Mybatis源码去动手完善。

## 一、Mybatis框架流程简介



在手写自己的Mybatis框架之前，我们先来了解一下Mybatis，它的源码中使用了大量的设计模式，阅读源码并观察设计模式在其中的应用，才能够更深入的理解源码（ref: Mybatis源码解读-设计模式总结 <http://www.crazyant.net/2022.html>）。我们对上图进行分析总结：

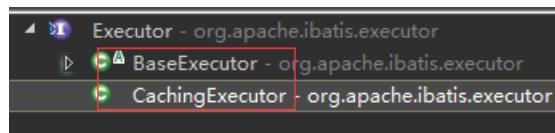
### 1、mybatis的配置文件有2类

- mybatisconfig.xml，配置文件的名称不是固定的，配置了全局的参数的配置，全局只能有一个配置文件。
- Mapper.xml 配置多个statement，也就是多个sql，整个mybatis框架中可以有多个Mapper.xml配置文件。

### 2、通过mybatis配置文件得到SqlSessionFactory

3、通过SqlSessionFactory得到SqlSession，用SqlSession就可以操作数据了。

### 4、SqlSession通过底层的Executor（执行器），执行器有2类实现：



- 基本实现
- 带有缓存功能的实现

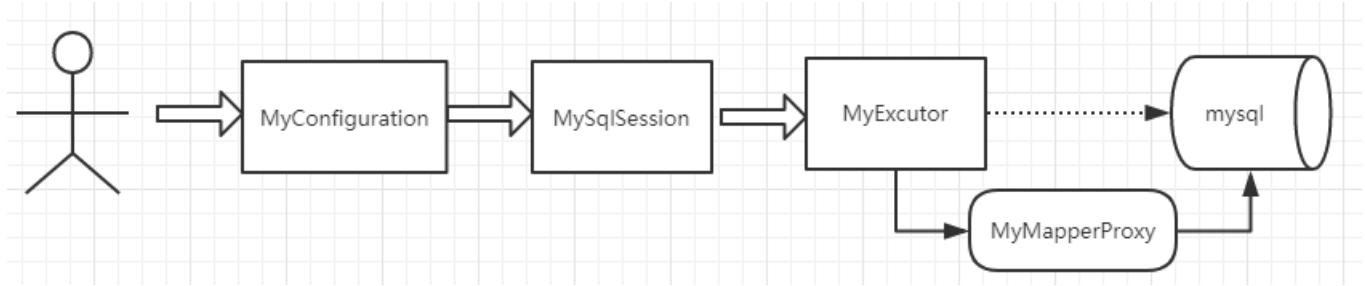
5、MappedStatement是通过Mapper.xml中定义statement生成的对象。

6、参数输入执行并输出结果集，无需手动判断参数类型和参数下标位置，且自动将结果集映射为Java对象

- HashMap, KV格式的数据类型
- Java的基本数据类型
- POJO, java的对象

## 二、梳理自己的Mybatis的设计思路

根据上文Mybatis流程，我简化了下，分为以下步骤：



### 1、读取xml文件，建立连接

从图中可以看出，MyConfiguration负责与人交互。待读取xml后，将属性和连接数据库的操作封装在MyConfiguration对象中供后面的组件调用。本文将使用dom4j来读取xml文件，它具有性能优异和非常方便使用的特点。

### 2、创建SqlSession，搭建Configuration和Executor之间的桥梁

我们经常在使用框架时看到Session，Session到底是什么呢？一个Session仅拥有一个对应的数据库连接。类似于一个前段请求Request，它可以直接调用exec(SQL)来执行SQL语句。从流程图中的箭头可以看出，MySqlSession的成员变量中必须得有MyExecutor和MyConfiguration去集中做调配，箭头就像是一种关联关系。我们自己的MySqlSession将有一个getMapper方法，然后使用动态代理生成对象后，就可以做数据库的操作了。

### 3、创建Executor，封装JDBC操作数据库

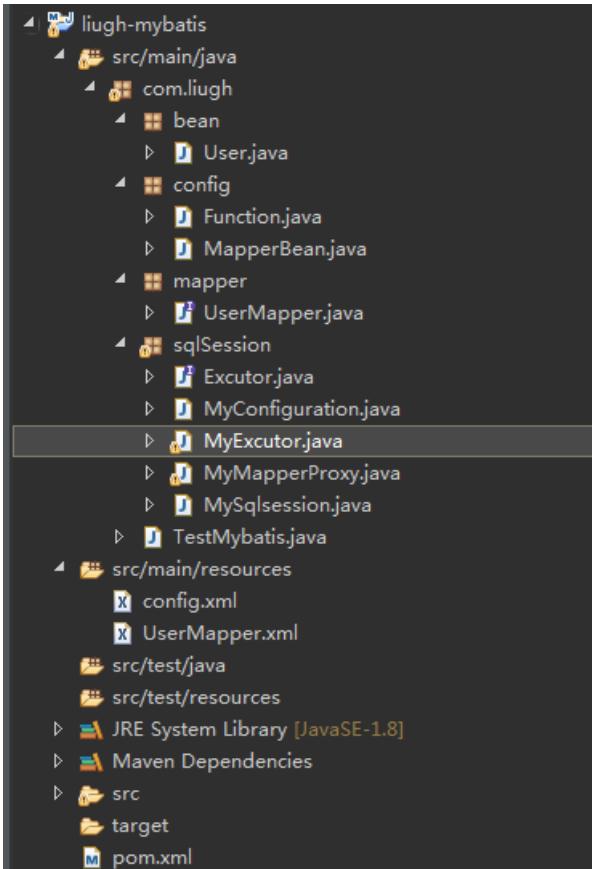
Executor是一个执行器，负责SQL语句的生成和查询缓存（缓存还没完成）的维护，也就是jdbc的代码将在这里完成，不过本文只实现了单表，有兴趣的同学可以尝试完成多表。

### 4、创建MapperProxy，使用动态代理生成Mapper对象

我们只是希望对指定的接口生成一个对象，使得执行它的时候能运行一句sql罢了，而接口无法直接调用方法，所以这里使用动态代理生成对象，在执行时还是回到MySqlSession中调用查询，最终由MyExecutor做JDBC查询。这样设计是为了单一职责，可扩展性更强。

### 三、实现自己的Mybatis

工程文件及目录：



首先，新建一个maven项目，在pom.xml中导入以下依赖：

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
<modelVersion>4.0.0</modelVersion>
<groupId>com.liugh</groupId>
<artifactId>liugh-mybatis</artifactId>
<version>0.0.1-SNAPSHOT</version>
<packaging>jar</packaging>

<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
    <java.version>1.8</java.version>
</properties>

<dependencies>
    <!-- 读取xml文件 -->
    <dependency>
        <groupId>dom4j</groupId>
        <artifactId>dom4j</artifactId>
        <version>1.6.1</version>
    </dependency>

    <!-- MySQL -->
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>5.1.29</version>
    </dependency>
</dependencies>
</project>
```

创建我们的数据库xml配置文件：

```
<?xml version="1.0" encoding="UTF-8"?>
<database>
    <property name="driverClassName">com.mysql.jdbc.Driver</property>
    <property name="url">jdbc:mysql://localhost:3306/test?useUnicode=true&characterEncoding=utf8</property>
    <property name="username">root</property>
    <property name="password">123456</property>
</database>
```

然后在数据库创建test库，执行如下SQL语句：

```
CREATE TABLE `user` (
    `id` varchar(64) NOT NULL,
    `password` varchar(255) DEFAULT NULL,
    `username` varchar(255) DEFAULT NULL,
    PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=2 DEFAULT CHARSET=utf8;
INSERT INTO `test`.`user`(`id`, `password`, `username`) VALUES ('1', '123456', 'liugh');
```

创建User实体类，和UserMapper接口和对应的xml文件：

```
package com.liugh.bean;

public class User {
    private String id;
    private String username;
    private String password;
    //省略get set toString方法...
}
```

```
package com.liugh.mapper;
import com.liugh.bean.User;
public interface UserMapper {
    public User getUserId(String id);
}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<mapper nameSpace="com.liugh.mapper.UserMapper">
    <select id="getUserById" resultType="com.liugh.bean.User">
        select * from user where id = ?
    </select>
</mapper>
```

基本操作配置完成，接下来我们开始实现MyConfiguration：

```
/**
 * 读取与解析配置信息，并返回处理后的Environment
 */
public class MyConfiguration {
    private static ClassLoader loader = ClassLoader.getSystemClassLoader();

    /**
     * 读取xml信息并处理
     */
    public Connection build(String resource){
        try {
            InputStream stream = loader.getResourceAsStream(resource);
            SAXReader reader = new SAXReader();

```

```

SAXReader reader = new SAXReader();
Document document = reader.read(stream);
Element root = document.getRootElement();
return evalDataSource(root);
} catch (Exception e) {
    throw new RuntimeException("error occurred while evaling xml " + resource);
}
}

private Connection evalDataSource(Element node) throws ClassNotFoundException {
if (!node.getName().equals("database")) {
    throw new RuntimeException("root should be <database>");
}
String driverClassName = null;
String url = null;
String username = null;
String password = null;
//获取属性节点
for (Object item : node.elements("property")) {
    Element i = (Element) item;
    String value = getValue(i);
    String name = i.getAttributeValue("name");
    if (name == null || value == null) {
        throw new RuntimeException("[database]: <property> should contain name and value");
    }
    //赋值
    switch (name) {
        case "url" : url = value; break;
        case "username" : username = value; break;
        case "password" : password = value; break;
        case "driverClassName" : driverClassName = value; break;
        default : throw new RuntimeException("[database]: <property> unknown name");
    }
}
}

Class.forName(driverClassName);
Connection connection = null;
try {
    //建立数据库链接
    connection = DriverManager.getConnection(url, username, password);
} catch (SQLException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}
return connection;
}

//获取property属性的值,如果有value值,则读取没有设置value,则读取内容
private String getValue(Element node) {
    return node.hasContent() ? node.getText() : node.getAttributeValue("value");
}

}

@SuppressWarnings("rawtypes")
public MapperBean readMapper(String path){
MapperBean mapper = new MapperBean();
try{
    InputStream stream = loader.getResourceAsStream(path);
    SAXReader reader = new SAXReader();
    Document document = reader.read(stream);
    Element root = document.getRootElement();
    mapper.setInterfaceName(root.getAttributeValue("nameSpace").trim()); //把mapper节点的nameSpace值存为接口名
    List<Function> list = new ArrayList<Function>(); //用来存储方法的List
    for (Object item : root.elements("function")) {
        Function function = new Function();
        function.setName(item.getAttributeValue("name"));
        function.setParam(item.getAttributeValue("param"));
        function.setReturnType(item.getAttributeValue("returnType"));
        list.add(function);
    }
    mapper.setFunctions(list);
}
}

```

```

for(iterator rootIter = root.elementIterator();rootIter.hasNext();) {//遍历根节点下所有子节点
    Function fun = new Function(); //用来存储一条方法的信息
    Element e = (Element) rootIter.next();
    String sqltype = e.getName().trim();
    String funcName = e.attributeValue("id").trim();
    String sql = e.getText().trim();
    String resultType = e.attributeValue("resultType").trim();
    fun.setSqltype(sqltype);
    fun.setFuncName(funcName);
    Object newInstance=null;
    try {
        newInstance = Class.forName(resultType).newInstance();
    } catch (InstantiationException e1) {
        e1.printStackTrace();
    } catch (IllegalAccessException e1) {
        e1.printStackTrace();
    } catch (ClassNotFoundException e1) {
        e1.printStackTrace();
    }
    fun.setResultType(newInstance);
    fun.setSql(sql);
    list.add(fun);
}
mapper.setList(list);

} catch (DocumentException e) {
    e.printStackTrace();
}
return mapper;
}
}

```

用面向对象的思想设计读取xml配置后：

```

package com.liugh.config;

import java.util.List;
public class MapperBean {
    private String interfaceName; //接口名
    private List<Function> list; //接口下所有方法
    //省略get set方法...
}

```

Function对象包括sql的类型、方法名、sql语句、返回类型和参数类型。

```

package com.liugh.config;

public class Function {
    private String sqltype;
    private String funcName;
    private String sql;
    private Object resultType;
    private String parameterType;
    //省略get set方法
}

```

接下来实现我们的MySqlSession,首先的成员变量里得有Excutor和MyConfiguration，代码的精髓就在getMapper的方法里

```
package com.liugh.sqlSession;

import java.lang.reflect.Proxy;

public class MySqlSession {

    private Executor excutor= new MyExecutor();

    private MyConfiguration myConfiguration = new MyConfiguration();

    public <T> T selectOne(String statement, Object parameter){
        return excutor.query(statement, parameter);
    }

    @SuppressWarnings("unchecked")
    public <T> T getMapper(Class<T> clas){
        //动态代理调用
        return (T)Proxy.newProxyInstance(clas.getClassLoader(),new Class[]{clas},
            new MyMapperProxy(myConfiguration,this));
    }

}
```

紧接着创建Executor和实现类：

```
package com.liugh.sqlSession;

public interface Executor {
    public <T> T query(String statement, Object parameter);
}
```

MyExecutor中封装了JDBC的操作：

```

public class MyExcutor implements Excutor{

    private MyConfiguration xmlConfiguration = new MyConfiguration();

    @Override
    public <T> T query(String sql, Object parameter) {
        Connection connection=getConnection();
        ResultSet set=null;
        PreparedStatement pre=null;
        try {
            pre = connection.prepareStatement(sql);
            //设置参数
            pre.setString(1, parameter.toString());
            set = pre.executeQuery();
            User u=new User();
            //遍历结果集
            while(set.next()){
                u.setId(set.getString(1));
                u.setUsername(set.getString(2));
                u.setPassword(set.getString(3));
            }
            return (T) u;
        } catch (SQLException e) {
            e.printStackTrace();
        } finally{
            try{
                if(set!=null){
                    set.close();
                }if(pre!=null){
                    pre.close();
                }if(connection!=null){
                    connection.close();
                }
            }catch(Exception e2){
                e2.printStackTrace();
            }
        }
        return null;
    }

    private Connection getConnection() {
        try {
            Connection connection =xmlConfiguration.build("config.xml");
            return connection;
        } catch (Exception e) {
            e.printStackTrace();
        }
        return null;
    }
}

```

MyMapperProxy代理类完成xml方法和真实方法对应，执行查询：

```

public class MyMapperProxy implements InvocationHandler{

    private MySqlSession mySqlSession;

    private MyConfiguration myConfiguration;

    public MyMapperProxy(MyConfiguration myConfiguration, MySqlSession mySqlSession) {
        this.myConfiguration = myConfiguration;
        this.mySqlSession = mySqlSession;
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        MapperBean readMapper = myConfiguration.readMapper("UserMapper.xml");
        //是否是xml文件对应的接口
        if (!method.getDeclaringClass().getName().equals(readMapper.getInterfaceName())){
            return null;
        }
        List<Function> list = readMapper.getList();
        if(null != list || 0 != list.size()){
            for (Function function : list) {
                //id是否和接口方法名一样
                if(method.getName().equals(function.getFuncName())){
                    return mySqlSession.selectOne(function.getSql(), String.valueOf(args[0]));
                }
            }
        }
        return null;
    }
}

```

到这里，就完成了自己的Mybatis框架，我们测试一下：

```

public class TestMybatis {

    public static void main(String[] args) {
        MySqlSession sqlSession = new MySqlSession();
        UserMapper mapper = sqlSession.getMapper(UserMapper.class);
        User user = mapper.getUserById("1");
        System.out.println(user);
    }
}

```

执行结果：

```

19  public static void main(String[] args) {
20      MySqlSession sqlSession = new MySqlSession();
21      UserMapper mapper = sqlSession.getMapper(UserMapper.class);
22      User user = mapper.getUserById("1"); |
23      System.out.println(user);
24  }
25
26 }
27

```

Problems Javadoc Declaration Search Console X Progress JUnit History Servers Debug Git

<terminated> TestMybatis [Java Application] D:\javaSoft\neon\jdk1.8.0\_101\bin\javaw.exe (2018年3月7日 下午12:54:15)

User [id=1, username=123456, password=liugh]

查询一个不存在的用户试试：

```
19 public static void main(String[] args) {  
20     MySqlSession sqlsession=new MySqlSession();  
21     UserMapper mapper = sqlsession.getMapper(UserMapper.class);  
22     User user = mapper.getUserById("2");  
23     System.out.println(user);  
24 } |  
25  
26 }  
27
```

Problems Javadoc Declaration Search Console X Progress JUnit History Servers Debug G

```
<terminated> TestMybatis [Java Application] D:\javaSoft\eclipse-neon\jdk1.8.0_101\bin\javaw.exe (2018年3月7日 下午12:54:29)  
User [id=null, username=null, password=null]
```

到这里我们就大功告成了！

我是个普通的程序猿，水平有限，文章难免有错误，欢迎牺牲自己宝贵时间的读者，就本文内容直抒己见，我的目的仅仅是希望对读者有所帮助。源码地址：<https://github.com/qq53182347/liugh-mybatis>

- END -

#### 推荐阅读

1. 警告！你的隐私正在被上亿网友围观偷看！
2. 全面了解 Nginx 主要应用场景
3. 一场近乎完美基于 Dubbo 的微服务改造实践
4. 什么是一致性 Hash 算法？
5. 团队开发中 Git 最佳实践



喜欢文章，点个在看

阅读原文

声明：pdf仅供学习使用，一切版权归原创公众号所有；建议持续关注原创公众号获取最新文章，学习愉快！

# 从零搭建一个 Spring Boot 开发环境！Spring Boot+Mybatis+Swagger2 环境搭建

calebman Java后端 2019-11-15

点击上方 Java后端, 选择 [设为星标](#)

优质文章, 及时送达

作者 | calebman

链接 | [www.jianshu.com/p/95946d6b0c7d](http://www.jianshu.com/p/95946d6b0c7d)

## 本文简介

- 为什么使用Spring Boot
- 搭建怎样一个环境
- 开发环境
- 导入快速启动项目
- 集成前准备
- 集成Mybatis
- 集成Swagger2
- 多环境配置
- 多环境下的日志配置
- 常用配置

## 为什么使用Spring Boot

Spring Boot 相对于传统的SSM框架的优点是提供了默认的样板化配置，简化了Spring应用的初始搭建过程，如果你不想被众多的xml配置文件困扰，可以考虑使用Spring Boot替代

## 搭建怎样一个环境

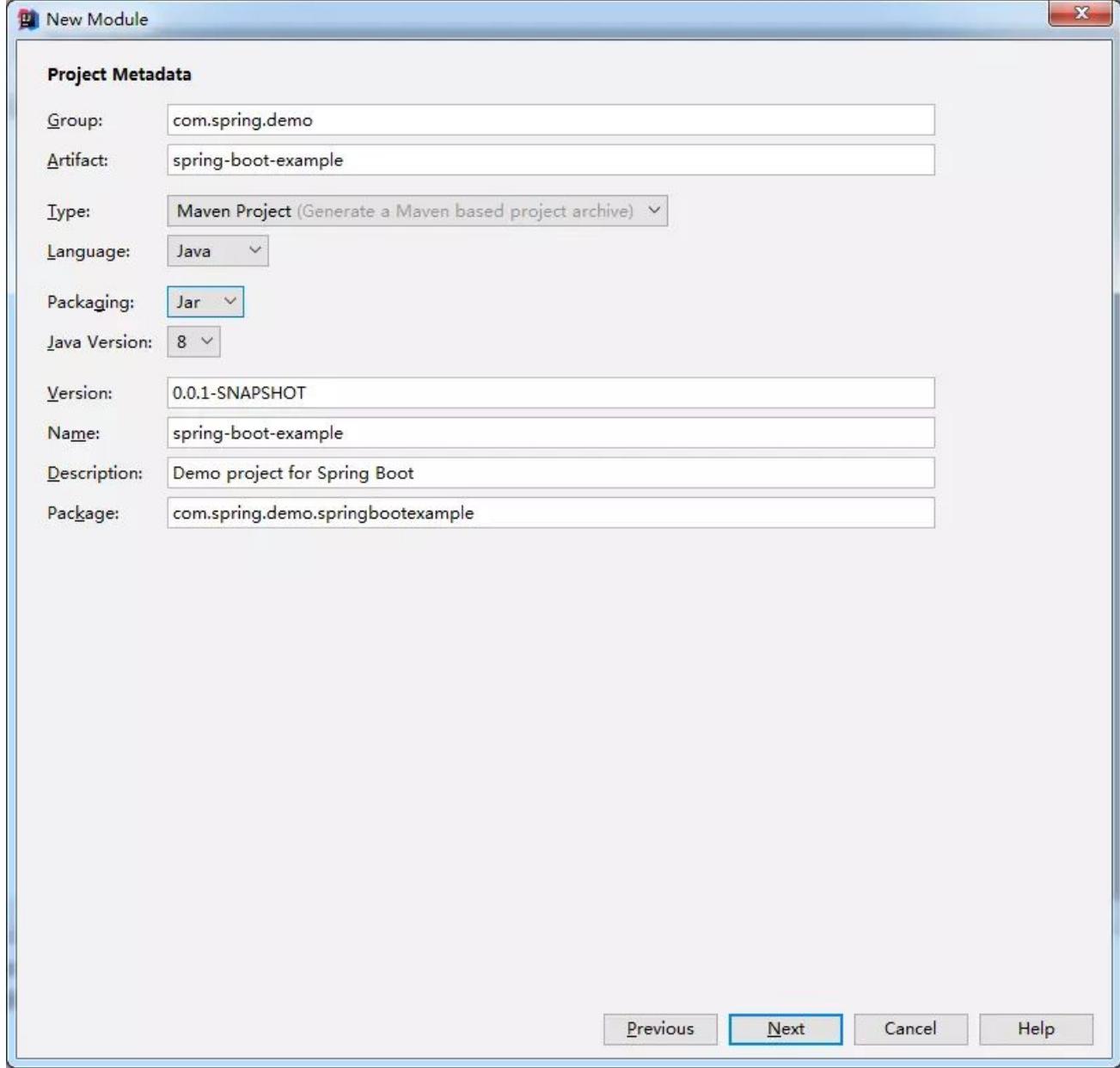
本文将基于Spring官方提供的快速启动项目模板集成Mybatis、Swagger2框架，并讲解mybatis generator一键生成代码插件、logback、一键生成文档以及多环境的配置方法，最后再介绍一下自定义配置的注解获取、全局异常处理等经常用到的东西。

## 开发环境

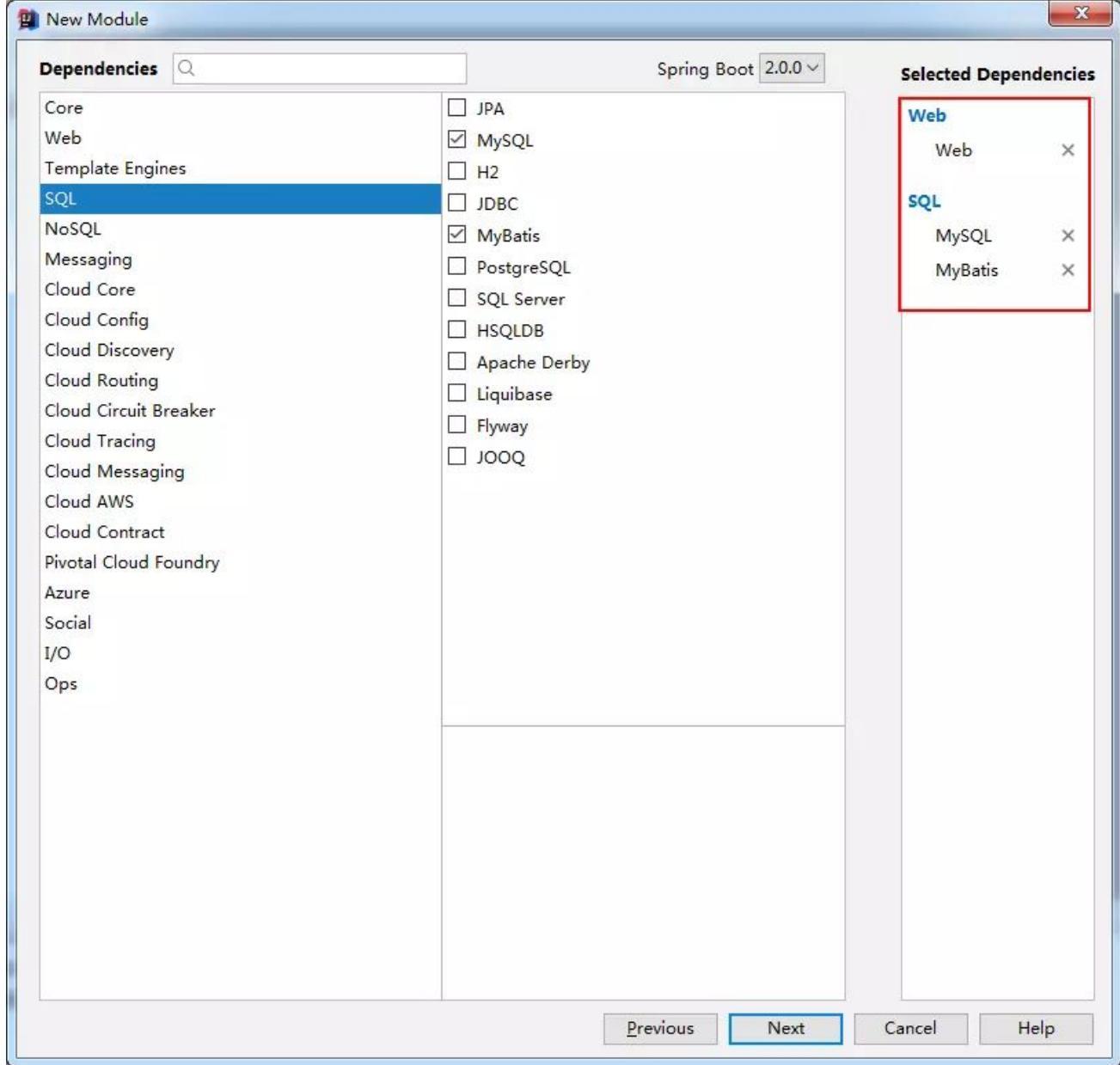
本人使用IDEA作为开发工具，IDEA下载时默认集成了SpringBoot的快速启动项目可以直接创建，如果使用Eclipse的同学可以考虑安装SpringBoot插件或者直接从这里配置并下载SpringBoot快速启动项目，需要注意的是本次环境搭建选择的是SpringBoot2.0的快速启动框架，Spring Boot2.0要求jdk版本必须要在1.8及以上。

## 导入快速启动项目

不管是通过IDEA导入还是通过命令行下载模板工程都需要初始化快速启动工程的配置，如果使用IDEA，在新建项目时选择Spring Initializr，主要配置如下图



IDEA新建SpringBoot项目-填写项目/包名

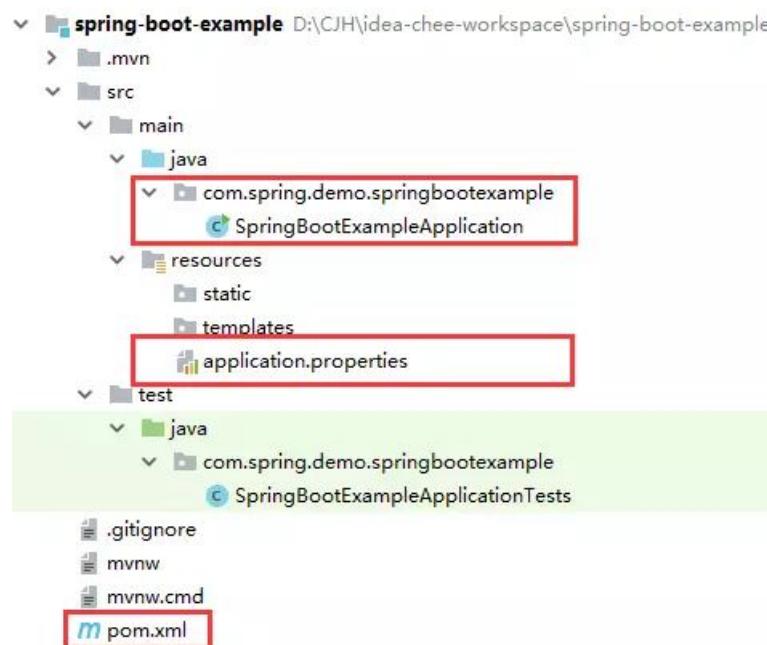


IDEA新建SpringBoot项目-选择依赖包

点击next之后finish之后IDEA显示正在下载模板工程，下载完成后会根据pom.xml下载包依赖，依赖下载完毕后模板项目就算创建成功了，如果是直接从官方网站配置下载快速启动项目可参考下图配置

The screenshot shows the Spring Initializr website interface. At the top, it says 'SPRING INITIALIZR bootstrap your application now'. Below that, there's a search bar with 'Generate a [Maven Project] with [Java] and Spring Boot [2.0.0]'. The 'Project Metadata' section has 'Artifact coordinates' fields for 'Group' (set to 'com.spring.demo') and 'Artifact' (set to 'spring-boot-example'). The 'Dependencies' section has a 'Search for dependencies' input field containing 'Web, Security, JPA, Actuator, Devtools...'. Under 'Selected Dependencies', 'Web', 'MySQL', and 'MyBatis' are highlighted in green. At the bottom, there's a 'Generate Project' button and a note: 'Don't know what to look for? Want more options? Switch to the full version.'

从Search for dependencies 框中输入并选择Web、 Mysql、 Mybatis加入依赖，点击Generate Project下载快速启动项目，然后在IDE中选择导入Maven项目，项目导入完成后可见其目录结构如下图



快速启动项目-项目结构

需要关注红色方框圈起来的部分，由上往下第一个java类是用来启动项目的入口函数，第二个properties后缀的文件是项目的配置文件，第三个是项目的依赖包以及执行插件的配置

## 集成前准备

修改.properties为.yml

yml相对于properties更加精简而且很多官方给出的Demo都是yml的配置形式，在这里我们采用yml的形式代替properties，相对于properties形式主要有以下两点不同

1. 对于键的描述由原有的 "." 分割变成了树的形状
2. 对于所有的键的后面一个要跟一个空格，不然启动项目会报配置解析错误

```
# properties式语法描述
spring.datasource.name = mysql
spring.datasource.url = jdbc:mysql://localhost:3306/db?characterEncoding=utf-8
spring.datasource.username = root
spring.datasource.password = 123
# yml式语法描述
spring:
  datasource:
    name: mysql
    url: jdbc:mysql://localhost:3306/db?characterEncoding=utf-8
    username: root
    password: 123
```

## 配置所需依赖

快速启动项目创建成功后我们观察其pom.xml文件中的依赖如下图，包含了我们选择的Web、 Mybatis以及Mysql

```

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>

<dependency>
    <groupId>org.mybatis.spring.boot</groupId>
    <artifactId>mybatis-spring-boot-starter</artifactId>
    <version>1.3.1</version>
</dependency>

<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <scope>runtime</scope>
</dependency>

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
</dependency>

```

但是我们使用ORM框架一般还会配合数据库连接池以及分页插件来使用，在这里我选择了阿里的druid以及pagehelper这个分页插件，再加上我们还需要整合swagger2文档自动化构建框架，所以增加了以下四个依赖项

```

<dependency>
    <groupId>com.github.pagehelper</groupId>
    <artifactId>pagehelper-spring-boot-starter</artifactId>
    <version>1.2.3</version>
</dependency>

<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid-spring-boot-starter</artifactId>
    <version>1.1.1</version>
</dependency>

<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>fastjson</artifactId>
    <version>1.2.31</version>
</dependency>

<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger2</artifactId>
    <version>2.5.0</version>
</dependency>

```

## 集成Mybatis

Mybatis的配置主要包括了druid数据库连接池、pagehelper分页插件、mybatis-generator代码逆向生成插件以及mapper、pojo扫描配置

### 配置druid数据库连接池

添加以下配置至application.yml文件中

```

spring:
  datasource:
    # 如果存在多个数据源，监控的时候可以通过名字来区分开来
    name: mysql
    # 连接数据库的url
    url: jdbc:mysql://localhost:3306/db?characterEncoding=utf-8
    # 连接数据库的账号
    username: root
    # 连接数据库的密码
    password: 123
    # 使用druid数据源
    type: com.alibaba.druid.pool.DruidDataSource
    # 扩展插件
    # 监控统计用的filter:stat 日志用的filter:log4j 防御sql注入的filter:wall
    filters: stat
    # 最大连接池数量
    maxActive: 20
    # 初始化时建立物理连接的个数。初始化发生在显示调用init方法，或者第一次getConnection时
    initialSize: 1
    # 获取连接时最大等待时间，单位毫秒
    maxWait: 60000
    # 最小连接池数量
    minIdle: 1
    timeBetweenEvictionRunsMillis: 60000
    # 连接保持空闲而不被驱逐的最长时间
    minEvictableIdleTimeMillis: 300000
    # 用来检测连接是否有效的sql，要求是一个查询语句
    # 如果validationQuery为null，testOnBorrow、testOnReturn、testWhileIdle都不会起作用
    validationQuery: select count(1) from 'table'
    # 申请连接的时候检测，如果空闲时间大于timeBetweenEvictionRunsMillis，执行validationQuery检测连接是否有效
    testWhileIdle: true
    # 申请连接时执行validationQuery检测连接是否有效，做了这个配置会降低性能
    testOnBorrow: false
    # 归还连接时执行validationQuery检测连接是否有效，做了这个配置会降低性能
    testOnReturn: false
    # 是否缓存PreparedStatement，即PSCache
    poolPreparedStatements: false
    # 要启用PSCache，必须配置大于0，当大于0时，poolPreparedStatements自动触发修改为true
    maxOpenPreparedStatements: -1

```

## 配置pagehelper分页插件

```

# pagehelper分页插件
pagehelper:
  # 数据库的方言
  helperDialect: mysql
  # 启用合理化，如果pageNum < 1会查询第一页，如果pageNum > pages会查询最后一页
  reasonable: true

```

## 代码逆向生成插件mybatis-generator的配置及运行

mybatis-generator插件的使用主要分为以下三步

1. pom.xml中添加mybatis-generator插件

```
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>

    <plugin>
      <groupId>org.mybatis.generator</groupId>
      <artifactId>mybatis-generator-maven-plugin</artifactId>
      <version>1.3.2</version>
      <configuration>
        <configurationFile>
          ${basedir}/src/main/resources/generator/generatorConfig.xml
        </configurationFile>
        <overwrite>true</overwrite>
        <verbose>true</verbose>
      </configuration>
    </plugin>
  </plugins>
</build>
```

## 2. 创建逆向代码生成配置文件generatorConfig.xml

参照pom.xml插件配置中的扫描位置，在resources目录下创建generator文件夹，在新建的文件夹中创建generatorConfig.xml配置文件，文件的详细配置信息如下

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE generatorConfiguration
    PUBLIC "-//mybatis.org//DTD MyBatis Generator Configuration 1.0//EN"
    "http://mybatis.org/dtd/mybatis-generator-config_1_0.dtd">
<generatorConfiguration>

    <properties resource="generator/generator.properties"/>
    <classPathEntry location="${classPathEntry}"/>
    <context id="DB2Tables" targetRuntime="MyBatis3">

        <jdbcConnection
            driverClass="com.mysql.jdbc.Driver"
            connectionURL="jdbc:mysql://localhost:3306/${db}?characterEncoding=utf-8"
            userId="${userId}"
            password="${password}">
        </jdbcConnection>
        <javaTypeResolver>
            <property name="forceBigDecimals" value="false"/>
        </javaTypeResolver>
        <javaModelGenerator targetPackage="${pojoTargetPackage}" targetProject="src/main/java">
            <property name="enableSubPackages" value="true"/>
            <property name="trimStrings" value="true"/>
        </javaModelGenerator>

        <sqlMapGenerator targetPackage="${mapperTargetPackage}" targetProject="src/main/resources">
            <property name="enableSubPackages" value="true"/>
        </sqlMapGenerator>

        <javaClientGenerator type="XMLMAPPER" targetPackage="${daoTargetPackage}" targetProject="src/main/java">
            <property name="enableSubPackages" value="true"/>
        </javaClientGenerator>

        <table tableName "%" schema="${db}"/>
    </context>
</generatorConfiguration>

```

为了将generatorConfig.xml配置模板化，在这里将变动性较大的配置项单独提取出来作为一个generatorConfig.xml的配置文件，然后通过properties标签读取此文件的配置，这样做的好处是当需要多处复用此xml时只需要关注少量的配置项。  
在generatorConfig.xml同级创建generator.properties文件，现只需要配置generator.properties文件即可，配置内容如下

```

# 请手动配置以下选项
# 数据库驱动:选择你的本地硬盘上面的数据库驱动包
classPathEntry = D:/CJH/maven-repository/mysql/mysql-connector-java/5.1.30/mysql-connector-java-5.1.30.jar
# 数据库名称、用户名、密码
db = db
userId = root
password = 123
# 生成pojo的包名位置 在src/main/java目录下
pojoTargetPackage = com.spring.demo.springbootexample.mybatis.po
# 生成DAO的包名位置 在src/main/java目录下
daoTargetPackage = com.spring.demo.springbootexample.mybatis.mapper
# 生成Mapper的包名位置 位于src/main/resources目录下
mapperTargetPackage = mapper

```

### 3. 运行mybatis-generator插件生成Dao、 Model、 Mapping

```

# 打开命令行cd到项目pom.xml同级目录运行以下命令
mvn mybatis-generator:generate -e

```

## mybatis扫描包配置

至此已经生成了指定数据库对应的实体、映射类，但是还不能直接使用，需要配置mybatis扫描地址后才能正常调用

### 1. 在application.yml配置mapper.xml以及pojo的包地址

```
mybatis:  
  # mapper.xml包地址  
  mapper-locations: classpath:mapper/*.xml  
  # pojo生成包地址  
  type-aliases-package: com.spring.demo.springbootexample.mybatis.po
```

### 2. 在SpringBootApplication.java中开启Mapper扫描注解

```
@SpringBootApplication  
 @MapperScan("com.spring.demo.springbootexample.mybatis.mapper")  
 public class SpringBootExampleApplication {  
  
  public static void main(String[] args) {  
    SpringApplication.run(SpringBootExampleApplication.class, args);  
  }  
}
```

### 测试mapper的有效性

```
@Controller  
public class TestController {  
  
  @Autowired  
  UserMapper userMapper;  
  
  @RequestMapping("/test")  
  @ResponseBody  
  public Object test(){  
  
    return userMapper.selectByExample(null);  
  }  
}
```

启动SpringBootApplication.java的main函数，如果没有在application.yml特意配置server.port那么springboot会采用默认的8080端口运行，运行成功将打印如下日志

```
Tomcat started on port(s): 8080 (http) with context path ''
```

在浏览器输入地址如果返回表格中的所有数据代表mybatis集成成功

```
http://localhost:8080/test
```

## 集成Swagger2

Swagger2是一个文档快速构建工具，能够通过注解自动生成一个Restful风格json形式的接口文档，并可以通过如swagger-ui等工具生成html网页形式的接口文档，swagger2的集成比较简单，使用需要稍微熟悉一下，集成、注解与使用分如下四步

### 1. 建立SwaggerConfig文件

```
@Configuration
public class SwaggerConfig {

    private final String version = "1.0";

    private final String title = "SpringBoot示例工程";

    private final String description = "API文档自动生成示例";

    private final String termsOfServiceUrl = "http://www.kingeid.com";

    private final String license = "MIT";

    private final String licenseUrl = "https://mit-license.org/";

    private final Contact contact = new Contact("calebman", "https://github.com/calebman", "chenjianhui0428@gmail.com");

    @Bean
    public Docket buildDocket() {
        return new Docket(DocumentationType.SWAGGER_2).apiInfo(buildApiInf())
            .select().build();
    }

    private ApiInfo buildApiInf() {
        return new ApiInfoBuilder().title(title).termsOfServiceUrl(termsOfServiceUrl).description(description)
            .version(version).license(license).licenseUrl(licenseUrl).contact(contact).build();
    }
}
```

## 2. 在SpringBootExampleApplication.java中启用Swagger2注解

在@SpringBootApplication注解下面加上@EnableSwagger2注解

## 3. 常用注解示例

```

@Controller
@RequestMapping("/v1/product")

@Api(value = "DocController", tags = {"restful api示例"})
public class DocController extends BaseController {

    @RequestMapping(value = "/{id}", method = RequestMethod.PUT)
    @ResponseBody

    @ApiOperation(value = "修改指定产品", httpMethod = "PUT", produces = "application/json")

    @ApiImplicitParams({@ApiImplicitParam(name = "id", value = "产品ID", required = true, paramType = "path")})
    public WebResult update(@PathVariable("id") Integer id, @ModelAttribute Product product) {
        logger.debug("修改指定产品接收产品id与产品信息=>%d,{}", id, product);
        if (id == null || "" .equals(id)) {
            logger.debug("产品id不能为空");
            return WebResult.error(ERRORDetail.RC_0101001);
        }
        return WebResult.success();
    }
}

```

```

@ApiModel(value = "产品信息")
public class Product {

    @ApiModelProperty(required = true, name = "name", value = "产品名称", dataType = "query")
    private String name;
    @ApiModelProperty(name = "type", value = "产品类型", dataType = "query")
    private String type;
}

```

#### 4. 生成json形式的文档

集成成功后启动项目控制台会打印级别为INFO的日志，截取部分如下，表明可通过访问应用的v2/api-docs接口得到文档api的json格式数据，可在浏览器输入指定地址验证集成是否成功

```

Mapped "[/v2/api-docs],methods=[GET],produces=[application/json || application/hal+json]]"
http://localhost:8080/v2/api-docs

```

## 多环境配置

应用研发过程中多环境是不可避免的，假设我们现在有开发、演示、生产三个不同的环境其配置也不同，如果每次都在打包环节来进行配置难免出错，SpringBoot支持通过命令启动不同的环境，但是配置文件需要满足application-{profile}.properties的格式，profile代表对应环境的标识，加载时可通过不同命令加载不同环境。

```

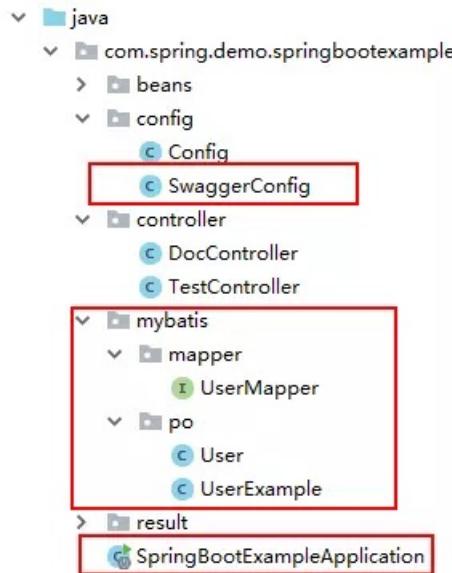
application-dev.properties: 开发环境
application-test.properties: 演示环境
application-prod.properties: 生产环境
# 运行演示环境命令
java -jar spring-boot-example-0.0.1-SNAPSHOT --spring.profiles.active=test

```

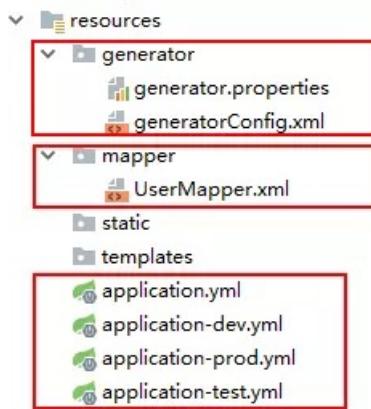
基于现在的项目实现多环境我们需要在application.yml同级目录新建application-dev.yml、application-test.yml、application-prod.yml三个不同环境的配置文件，将不变的公有配置如druid的大部分、pagehelper分页插件以及mybatis包扫描配置放置于application.yml中，并在application.yml中配置默认采用开发环境，那么如果不带--spring.profiles.active启动应用就默认为开发环境启动，变动较大的配置如数据库的账号密码分别写入不同环境的配置文件中

```
spring:  
  profiles:  
    # 默认使用开发环境  
    active: dev
```

配置到这里我们的项目目录结构如下图所示



src/main/java目录结构



src/main/resources目录结构

至此我们分别完成了Mybatis、Swagger2以及多环境的集成，接下来我们配置多环境下的logger。对于logger我们总是希望在项目研发过程中越多越好，能够给予足够的信息定位bug，项目处于演示或者上线状态时为了不让日志打印影响程序性能我们只需要警告或者错误的日志，并且需要写入文件，那么接下来就基于logback实现多环境下的日志配置

## 多环境下的日志配置

创建logback-spring.xml在application.yml的同级目录，springboot推荐使用logback-spring.xml而不是logback.xml文件，logback-spring.xml的配置内容如下所示

```
<?xml version="1.0" encoding="UTF-8"?>  
<configuration scan="true" scanPeriod="60 seconds" debug="false">  
  
<property name="maxsize" value="30MB" />  
  
<property name="maxdays" value="90" />  
  
<springProperty scope="context" name="logdir" source="resources.logdir"/>
```

```

<springProperty scope="context" name="appname" source="resources.appname"/>

<springProperty scope="context" name="basepackage" source="resources.basepackage"/>

<appender name="consoleLog" class="ch.qos.logback.core.ConsoleAppender">

<layout class="ch.qos.logback.classic.PatternLayout">
  <pattern>
    <pattern>%d{HH:mm:ss.SSS} [%-5level] %logger{36} - %msg%n</pattern>
  </pattern>
</layout>
</appender>

<appender name="fileLog" class="ch.qos.logback.core.rolling.RollingFileAppender">

<File>${logdir}/${appname}.log</File>

<rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">

  <FileNamePattern>${logdir}/${appname}.%d{yyyy-MM-dd}.log</FileNamePattern>
  <maxHistory>${maxdays}</maxHistory>
  <totalSizeCap>${maxsize}</totalSizeCap>
</rollingPolicy>

<encoder>
  <charset>UTF-8</charset>
  <pattern>%d{HH:mm:ss.SSS} [%-5level] %logger{36} - %msg%n</pattern>
</encoder>
</appender>

<springProfile name="dev">
  <root level="INFO">
    <appender-ref ref="consoleLog"/>
  </root>

  <logger name="${basepackage}" level="DEBUG" additivity="false">
    <appender-ref ref="consoleLog"/>
  </logger>
</springProfile>

<springProfile name="test">
  <root level="WARN">
    <appender-ref ref="consoleLog"/>
    <appender-ref ref="fileLog"/>
  </root>
  <logger name="${basepackage}" level="INFO" additivity="false">
    <appender-ref ref="consoleLog"/>
    <appender-ref ref="fileLog"/>
  </logger>
</springProfile>

<springProfile name="prod">
  <root level="ERROR">
    <appender-ref ref="consoleLog"/>
    <appender-ref ref="fileLog"/>
  </root>
</springProfile>
</configuration>

```

日志配置中引用了application.yml的配置信息，主要有logdir、appname、basepackage三项，logdir是日志文件的写入地址，可以传入相对路径，appname是应用名称，引入这项是为了通过日志文件名称区分是哪个应该输出的，basepackage是包

过滤配置，比如开发环境中需要打印debug级别以上的日志，但是又想使除我写的logger之外的DEBUG不打印，可过滤到本项目的包名才用DEBUG打印，此外包名使用INFO级别打印，在application.yml中新建这三项配置，也可在不同环境配置不同属性

```
#应用配置
resources:
  # log文件写入地址
  logdir: logs/
  # 应用名称
  appname: spring-boot-example
  # 日志打印的基础扫描包
  basepackage: com.spring.demo.springbootexample
```

使用不同环境启动测试logger配置是否生效，在开发环境下将打印DEBUG级别以上的四条logger记录，在演示环境下降打印INFO级别以上的三条记录并写入文件，在生产环境下只打印ERROR级别以上的一条记录并写入文件

```
@RequestMapping("/logger")
@ResponseBody
public WebResult logger() {
    logger.trace("日志输出 {}", "trace");
    logger.debug("日志输出 {}", "debug");
    logger.info("日志输出 {}", "info");
    logger.warn("日志输出 {}", "warn");
    logger.error("日志输出 {}", "error");
    return "00";
}
```

## 常用配置

加载自定义配置

```
@Component
@PropertySource(value = {"classpath:application.yml"}, encoding = "utf-8")
public class Config {

    @Value("${resources.midpHost}")
    private String midpHost;

    public String getMidpHost() {
        return midpHost;
    }
}
```

全局异常处理器

```
@ControllerAdvice
public class GlobalExceptionResolver {

    Logger logger = LoggerFactory.getLogger(GlobalExceptionResolver.class);

    @ExceptionHandler(value = Exception.class)
    @ResponseBody
    public WebResult exceptionHandle(HttpServletRequest req, Exception ex) {
        ex.printStackTrace();
        logger.error("未知异常", ex);
        return WebResult.error(ERRORDetail.RC_0401001);
    }
}
```

[示例工程开源地址](#)

github:https://link.jianshu.com/?t=https%3A%2F%2Fgithub.com%2Fcalebman%2Fspring-boot-example

[阅读原文](#)

声明：pdf仅供学习使用，一切版权归原创公众号所有；建议持续关注原创公众号获取最新文章，学习愉快！

# 关于 MyBatis 我总结了 10 种通用的写法

Java后端 2019-12-06

点击上方 Java后端, 选择 [设为星标](#)

优质文章, 及时送达

作者 | smile\_lg

链接 | [blog.csdn.net/smile\\_lg/article/details/71215619](http://blog.csdn.net/smile_lg/article/details/71215619)

## 用来循环容器的标签forEach,查看例子

foreach元素的属性主要有item, index, collection, open, separator, close。

- item: 集合中元素迭代时的别名,
- index: 集合中元素迭代时的索引
- open: 常用语where语句中, 表示以什么开始, 比如以'('开始
- separator: 表示在每次进行迭代时的分隔符,
- close 常用语where语句中, 表示以什么结束,

在使用foreach的时候最关键的也是最容易出错的就是collection属性, 该属性是必须指定的, 但是在不同情况下, 该属性的值是不一样的, 主要有以下3种情况:

- 如果传入的是单参数且参数类型是一个List的时候, collection属性值为list .
- 如果传入的是单参数且参数类型是一个array数组的时候, collection的属性值为array .
- 如果传入的参数是多个的时候, 我们就需要把它们封装成一个Map了, 当然单参数也可以封装成map, 实际上如果你在传入参数的时候, 在MyBatis里面也是会把它封装成一个Map的, map的key就是参数名, 所以这个时候collection属性值就是传入的List或array对象在自己封装的map里面的key.

针对最后一条, 我们来看一下官方说法:

注意 你可以将一个 List 实例或者数组作为参数对象传给 MyBatis, 当你这么做的时候, MyBatis 会自动将它包装在一个 Map 中并以名称为键。List 实例将会以 “list” 作为键, 而数组实例的键将是 “array” 。

所以, 不管是多参数还是单参数的list,array类型, 都可以封装为map进行传递。如果传递的是一个List, 则mybatis会封装为一个list为key, list值为object的map, 如果是array, 则封装成一个array为key, array的值为object的map, 如果自己封装呢, 则colloection里放的是自己封装的map里的key值

```
//mapper中我们要为这个方法传递的是一个容器,将容器中的元素一个一个的
```

```
//拼接到xml的方法中就要使用这个forEach这个标签了
```

```
public List<Entity> queryById(List<String> userids);
```

```
//对应的xml中如下
```

```
<select id="queryById" resultMap="BaseResultMap" >  
    select * FROM entity  
    where id in  
        <foreach collection="userids" item="userid" index="index" open="(" separator="," close=")">  
            #{userid}  
        </foreach>  
</select>
```

## concat模糊查询

```
//比如说我们想要进行条件查询,但是几个条件不是每次都要使用,那么我们就可以
```

```
//通过判断是否拼接到sql中
```

```
<select id="queryById" resultMap="BaseResultMap" parameterType="entity">  
    SELECT * from entity  
    <where>  
        <if test="name!=null">  
            name like concat('%',concat(#{name},'%'))  
        </if>  
    </where>  
</select>
```

## choose (when, otherwise)标签

choose标签是按顺序判断其内部when标签中的test条件出否成立，如果有任何一个成立，则 choose 结束。当 choose 中所有 when 的条件都不满足时，则执行 otherwise 中的sql。类似于Java 的 switch 语句，choose 为 switch，when 为 case，otherwise 则为 default。

例如下面例子，同样把所有可以限制的条件都写上，方便使用。choose会从上到下选择一个when标签的test为true的sql执行。

安全考虑，我们使用where将choose包起来，放置关键字多于错误。

```
<!-- choose(判断参数) - 按顺序将实体类 User 第一个不为空的属性作为：where条件 -->  
<select id="getUserList_choose" resultMap="resultMap_user" parameterType="com.yiibai.pojo.User">  
    SELECT *  
    FROM User u  
    <where>  
        <choose>  
            <when test="username != null ">  
                u.username LIKE CONCAT(CONCAT('%', #{username, jdbcType=VARCHAR}), '%')  
            </when >  
            <when test="sex != null and sex != '' ">  
                AND u.sex = #{sex, jdbcType=INTEGER}  
            </when >  
            <when test="birthday != null ">  
                AND u.birthday = #{birthday, jdbcType=DATE}  
            </when >  
        <otherwise>  
        </otherwise>  
    </choose>  
    </where>  
</select>
```

## selectKey 标签

在insert语句中，在Oracle经常使用序列、在MySQL中使用函数来自动生成插入表的主键，而且需要方法能返回这个生成主键。使用myBatis的selectKey标签可以实现这个效果。

下面例子，使用mysql数据库自定义函数nextval('student')，用来生成一个key，并把他设置到传入的实体类中的studentId属性上。所以在执行完此方法后，边可以通过这个实体类获取生成的key。

```
<!-- 插入学生 自动主键-->
<insert id="createStudentAutoKey" parameterType="liming.student.manager.data.model.StudentEntity" keyProperty="studentId">
    <selectKey keyProperty="studentId" resultType="String" order="BEFORE">
        select nextval('student')
    </selectKey>
    INSERT INTO STUDENT_TBL(STUDENT_ID,
        STUDENT_NAME,
        STUDENT_SEX,
        STUDENT_BIRTHDAY,
        STUDENT_PHOTO,
        CLASS_ID,
        PLACE_ID)
    VALUES (#{studentId},
        #{studentName},
        #{studentSex},
        #{studentBirthday},
        #{studentPhoto, javaType=byte[], jdbcType=BLOB, typeHandler=org.apache.ibatis.type.BlobTypeHandler},
        #{classId},
        #{placeId})
</insert>
```

调用接口方法，和获取自动生成key

```
StudentEntity entity = new StudentEntity();
entity.setStudentName("黎明你好");
entity.setStudentSex(1);
entity.setStudentBirthday(DateUtil.parse("1985-05-28"));
entity.setClassId("20000001");
entity.setPlaceId("70000001");
this.dynamicSqlMapper.createStudentAutoKey(entity);
System.out.println("新增学生ID: " + entity.getStudentId());
```

## if标签

if标签可用在许多类型的sql语句中，我们以查询为例。首先看一个很普通的查询：

```
<!-- 查询学生list, like姓名 -->
<select id="getStudentListLikeName" parameterType="StudentEntity" resultMap="studentResultMap">
    SELECT * from STUDENT_TBL ST
    WHERE ST.STUDENT_NAME LIKE CONCAT(CONCAT('%', #{studentName}), '%')
</select>
```

但是此时如果studentName为null，此语句很可能报错或查询结果为空。此时我们使用if动态sql语句先进行判断，如果值为null或等于空字符串，我们就不进行此条件的判断，增加灵活性。

参数为实体类StudentEntity。将实体类中所有的属性均进行判断，如果不为空则执行判断条件。

```

<!-- 2 if(判断参数) - 将实体类不为空的属性作为 where 条件 -->
<select id="getStudentList_if" resultMap="resultMap_studentEntity" parameterType="liming.student.manager.data.model.StudentEntity">
    SELECT ST.STUDENT_ID,
        ST.STUDENT_NAME,
        ST.STUDENT_SEX,
        ST.STUDENT_BIRTHDAY,
        ST.STUDENT_PHOTO,
        ST.CLASS_ID,
        ST.PLACE_ID
    FROM STUDENT_TBL ST
    WHERE
        <if test="studentName !=null ">
            ST.STUDENT_NAME LIKE CONCAT(CONCAT('%', #{studentName, jdbcType=VARCHAR}), '%')
        </if>
        <if test="studentSex != null and studentSex != '' ">
            AND ST.STUDENT_SEX = #{studentSex, jdbcType=INTEGER}
        </if>
        <if test="studentBirthday != null ">
            AND ST.STUDENT_BIRTHDAY = #{studentBirthday, jdbcType=DATE}
        </if>
        <if test="classId != null and classId!= '' ">
            AND ST.CLASS_ID = #{classId, jdbcType=VARCHAR}
        </if>
        <if test="classEntity != null and classEntity.classId !=null and classEntity.classId !=' ' ">
            AND ST.CLASS_ID = #{classEntity.classId, jdbcType=VARCHAR}
        </if>
        <if test="placeId != null and placeId != '' ">
            AND ST.PLACE_ID = #{placeId, jdbcType=VARCHAR}
        </if>
        <if test="placeEntity != null and placeEntity.placeId != null and placeEntity.placeId != '' ">
            AND ST.PLACE_ID = #{placeEntity.placeId, jdbcType=VARCHAR}
        </if>
        <if test="studentId != null and studentId != '' ">
            AND ST.STUDENT_ID = #{studentId, jdbcType=VARCHAR}
        </if>
    </select>

```

使用时比较灵活，new一个这样的实体类，我们需要限制那个条件，只需要附上相应的值就会where这个条件，相反不去赋值就可以不在where中判断。

```

public void select_test_2_1() {
    StudentEntity entity = new StudentEntity();
    entity.setStudentName("");
    entity.setStudentSex(1);
    entity.setStudentBirthday(DateUtil.parse("1985-05-28"));
    entity.setClassId("20000001");
    //entity.setPlaceId("70000001");
    List<StudentEntity> list = this.dynamicSqlMapper.getStudentList_if(entity);
    for (StudentEntity e : list) {
        System.out.println(e.toString());
    }
}

```

## if + where 的条件判断

当where中的条件使用的if标签较多时，这样的组合可能会导致错误。我们以在3.1中的查询语句为例子，当java代码按如下方法调用时：

```

@Test
public void select_test_2_1() {
    StudentEntity entity = new StudentEntity();
    entity.setStudentName(null);
    entity.setStudentSex(1);
    List<StudentEntity> list = this.dynamicSqlMapper.getStudentList_if(entity);
    for (StudentEntity e : list) {
        System.out.println(e.toString());
    }
}

```

如果上面例子，参数studentName为null，将不会进行STUDENT\_NAME列的判断，则会直接导“WHERE AND”关键字多余的错误SQL。

这时我们可以使用where动态语句来解决。这个“where”标签会知道如果它包含的标签中有返回值的话，它就插入一个‘where’。此外，如果标签返回的内容是以AND 或OR 开头的，则它会剔除掉。

上面例子修改为：

```

<!-- 3 select - where/if(判断参数) - 将实体类不为空的属性作为 where条件 -->
<select id="getStudentList_wherelf" resultMap="resultMap_studentEntity" parameterType="liming.student.manager.data.model.Studen
    SELECT ST.STUDENT_ID,
        ST.STUDENT_NAME,
        ST.STUDENT_SEX,
        ST.STUDENT_BIRTHDAY,
        ST.STUDENT_PHOTO,
        ST.CLASS_ID,
        ST.PLACE_ID
    FROM STUDENT_TBL ST
<where>
    <if test="studentName !=null ">
        ST.STUDENT_NAME LIKE CONCAT(CONCAT('%', #{studentName,jdbcType=VARCHAR}), '%')
    </if>
    <if test="studentSex != null and studentSex != '' ">
        AND ST.STUDENT_SEX = #{studentSex,jdbcType=INTEGER}
    </if>
    <if test="studentBirthday != null ">
        AND ST.STUDENT_BIRTHDAY = #{studentBirthday,jdbcType=DATE}
    </if>
    <if test="classId != null and classId!= '' ">
        AND ST.CLASS_ID = #{classId,jdbcType=VARCHAR}
    </if>
    <if test="classEntity != null and classEntity.classId !=null and classEntity.classId !=' ' ">
        AND ST.CLASS_ID = #{classEntity.classId,jdbcType=VARCHAR}
    </if>
    <if test="placeId != null and placeId != '' ">
        AND ST.PLACE_ID = #{placeId,jdbcType=VARCHAR}
    </if>
    <if test="placeEntity != null and placeEntity.placeId != null and placeEntity.placeId !='' ">
        AND ST.PLACE_ID = #{placeEntity.placeId,jdbcType=VARCHAR}
    </if>
    <if test="studentId != null and studentId != '' ">
        AND ST.STUDENT_ID = #{studentId,jdbcType=VARCHAR}
    </if>
</where>
</select>

```

当update语句中没有使用if标签时，如果有一个参数为null，都会导致错误。

当在update语句中使用if标签时，如果前面的if没有执行，则或导致逗号多余错误。使用set标签可以将动态的配置SET关键字，和剔除追加到条件末尾的任何不相关的逗号。使用if+set标签修改后，如果某项为null则不进行更新，而是保持数据库原值。

如下示例：

```
<!-- 4 if/set(判断参数) - 将实体类不为空的属性更新 -->
<update id="updateStudent_if_set" parameterType="liming.student.manager.data.model.StudentEntity">
    UPDATE STUDENT_TBL
    <set>
        <if test="studentName != null and studentName != ''">
            STUDENT_TBL.STUDENT_NAME = #{studentName},
        </if>
        <if test="studentSex != null and studentSex != ''">
            STUDENT_TBL.STUDENT_SEX = #{studentSex},
        </if>
        <if test="studentBirthday != null ">
            STUDENT_TBL.STUDENT_BIRTHDAY = #{studentBirthday},
        </if>
        <if test="studentPhoto != null ">
            STUDENT_TBL.STUDENT_PHOTO = #[studentPhoto, javaType=byte[], jdbcType=BLOB, typeHandler=org.apache.ibatis.type.BlobTyp
        </if>
        <if test="classId != ''">
            STUDENT_TBL.CLASS_ID = #{classId}
        </if>
        <if test="placeId != ''">
            STUDENT_TBL.PLACE_ID = #{placeId}
        </if>
    </set>
    WHERE STUDENT_TBL.STUDENT_ID = #{studentId};
</update>
```

### if + trim代替where/set标签

trim是更灵活的去处多余关键字的标签，他可以实践where和set的效果。

#### trim代替where

```

<!-- 5.1if/trim代替where(判断参数) -将实体类不为空的属性作为where条件-->
<select id="getStudentList_if_trim" resultMap="resultMap_studentEntity">
    SELECT ST.STUDENT_ID,
        ST.STUDENT_NAME,
        ST.STUDENT_SEX,
        ST.STUDENT_BIRTHDAY,
        ST.STUDENT_PHOTO,
        ST.CLASS_ID,
        ST.PLACE_ID
    FROM STUDENT_TBL ST
    <trim prefix="WHERE" prefixOverrides="AND|OR">
        <if test="studentName !=null ">
            ST.STUDENT_NAME LIKE CONCAT(CONCAT('%', #{studentName, jdbcType=VARCHAR}), '%')
        </if>
        <if test="studentSex != null and studentSex != '' ">
            AND ST.STUDENT_SEX = #{studentSex, jdbcType=INTEGER}
        </if>
        <if test="studentBirthday != null ">
            AND ST.STUDENT_BIRTHDAY = #{studentBirthday, jdbcType=DATE}
        </if>
        <if test="classId != null and classId!= '' ">
            AND ST.CLASS_ID = #{classId, jdbcType=VARCHAR}
        </if>
        <if test="classEntity != null and classEntity.classId !=null and classEntity.classId !=' ' ">
            AND ST.CLASS_ID = #{classEntity.classId, jdbcType=VARCHAR}
        </if>
        <if test="placeId != null and placeId != '' ">
            AND ST.PLACE_ID = #{placeId, jdbcType=VARCHAR}
        </if>
        <if test="placeEntity != null and placeEntity.placeId != null and placeEntity.placeId !='' ">
            AND ST.PLACE_ID = #{placeEntity.placeId, jdbcType=VARCHAR}
        </if>
        <if test="studentId != null and studentId != '' ">
            AND ST.STUDENT_ID = #{studentId, jdbcType=VARCHAR}
        </if>
    </trim>
</select>

```

trim代替set

```

<!-- 5.2 if/trim代替set(判断参数) - 将实体类不为空的属性更新 -->
<update id="updateStudent_if_trim" parameterType="liming.student.manager.data.model.StudentEntity">
    UPDATE STUDENT_TBL
    <trim prefix="SET" suffixOverrides=",">
        <if test="studentName != null and studentName != '' ">
            STUDENT_TBL.STUDENT_NAME = #{studentName},
        </if>
        <if test="studentSex != null and studentSex != '' ">
            STUDENT_TBL.STUDENT_SEX = #{studentSex},
        </if>
        <if test="studentBirthday != null ">
            STUDENT_TBL.STUDENT_BIRTHDAY = #{studentBirthday},
        </if>
        <if test="studentPhoto != null ">
            STUDENT_TBL.STUDENT_PHOTO = #{studentPhoto, javaType=byte[], jdbcType=BLOB, typeHandler=org.apache.ibatis.type.BlobTyp
        </if>
        <if test="classId != '' ">
            STUDENT_TBL.CLASS_ID = #{classId},
        </if>
        <if test="placeId != '' ">
            STUDENT_TBL.PLACE_ID = #{placeId}
        </if>
    </trim>
    WHERE STUDENT_TBL.STUDENT_ID = #{studentId}
</update>

```

## foreach

对于动态SQL非常必须的，主要是要迭代一个集合，通常是用于IN条件。List实例将使用“list”做为键，数组实例以“array”做为键。

foreach元素是非常强大的，它允许你指定一个集合，声明集合项和索引变量，它们可以用在元素体内。它也允许你指定开放和关闭的字符串，在迭代之间放置分隔符。这个元素是很智能的，它不会偶然地附加多余的分隔符。

注意：你可以传递一个List实例或者数组作为参数对象传给MyBatis。当你这么做的时候，MyBatis会自动将它包装在一个Map中，用名称作为键。List实例将会以“list”作为键，而数组实例将会以“array”作为键。

这个部分是对关于XML配置文件和XML映射文件的而讨论的。下一部分将详细讨论Java API，所以你可以得到你已经创建的最有效的映射。

### 1.参数为array示例的写法

接口的方法声明：

```
public List<StudentEntity> getStudentListByClassIds_foreach_array(String[] classIds);
```

动态SQL语句：

```

<!-- 7.1 foreach(循环array参数) - 作为where中in的条件 -->
<select id="getStudentListByClassIds_foreach_array" resultMap="resultMap_studentEntity">
    SELECT ST.STUDENT_ID,
        ST.STUDENT_NAME,
        ST.STUDENT_SEX,
        ST.STUDENT_BIRTHDAY,
        ST.STUDENT_PHOTO,
        ST.CLASS_ID,
        ST.PLACE_ID
    FROM STUDENT_TBL ST
    WHERE ST.CLASS_ID IN
        <foreach collection="array" item="classIds" open="(" separator="," close=")">
            #{classIds}
        </foreach>
    </select>

```

测试代码，查询学生中，在20000001、20000002这两个班级的学生：

```

@Test
public void test7_foreach() {
    String[] classIds = { "20000001", "20000002" };
    List<StudentEntity> list = this.dynamicSqlMapper.getStudentListByClassIds_foreach_array(classIds);
    for (StudentEntity e : list) {
        System.out.println(e.toString());
    }
}

```

## 2.参数为list示例的写法

接口的方法声明：

```
public List<StudentEntity> getStudentListByClassIds_foreach_list(List<String> classIdList);
```

动态SQL语句：

```

<!-- 7.2 foreach(循环List<String>参数) - 作为where中in的条件 -->
<select id="getStudentListByClassIds_foreach_list" resultMap="resultMap_studentEntity">
    SELECT ST.STUDENT_ID,
        ST.STUDENT_NAME,
        ST.STUDENT_SEX,
        ST.STUDENT_BIRTHDAY,
        ST.STUDENT_PHOTO,
        ST.CLASS_ID,
        ST.PLACE_ID
    FROM STUDENT_TBL ST
    WHERE ST.CLASS_ID IN
        <foreach collection="list" item="classIdList" open="(" separator="," close=")">
            #{classIdList}
        </foreach>
    </select>

```

测试代码，查询学生中，在20000001、20000002这两个班级的学生：

```
@Test  
public void test7_2_foreach() {  
    ArrayList<String> classIdList = new ArrayList<String>();  
    classIdList.add("20000001");  
    classIdList.add("20000002");  
    List<StudentEntity> list = this.dynamicSqlMapper.getStudentListByClassIds_foreach_list(classIdList);  
    for (StudentEntity e : list) {  
        System.out.println(e.toString());  
    }  
}
```

sql片段标签 `<sql>`：通过该标签可定义能复用的sql语句片段，在执行sql语句标签中直接引用即可。这样既可以提高编码效率，还能有效简化代码，提高可读性

需要配置的属性：`id="" >>>`表示需要改sql语句片段的唯一标识

引用：通过 `<include refid="" />` 标签引用，`refid=""` 中的值指向需要引用的 `<sql>` 中的`id=“”` 属性

```
<!--定义sql片段-->  
<sql id="orderAndItem">  
    o.order_id,o.cid,o.address,o.create_date,o.orderitem_id,i.orderitem_id,i.product_id,i.count  
</sql>  
  
<select id="findOrderAndItemsByOid" parameterType="java.lang.String" resultMap="BaseResultMap">  
    select  
<!--引用sql片段-->  
    <include refid="orderAndItem" />  
    from ordertable o  
    join orderitem i on o.orderitem_id = i.orderitem_id  
    where o.order_id = #{orderId}  
</select>
```

【END】

如果看到这里，说明你喜欢这篇文章，请[转发、点赞](#)。微信搜索「web\_resource」，关注后回复「进群」或者扫描下方二维码即可进入无广告交流群。

↓ 扫描二维码进群 ↓



1. Spring Boot:启动原理解析
2. 一键下载 Pornhub 视频!
3. Spring Boot 多模块项目实践(附打包方法)
4. 一个女生不主动联系你还有机会吗?
5. 团队开发中 Git 最佳实践

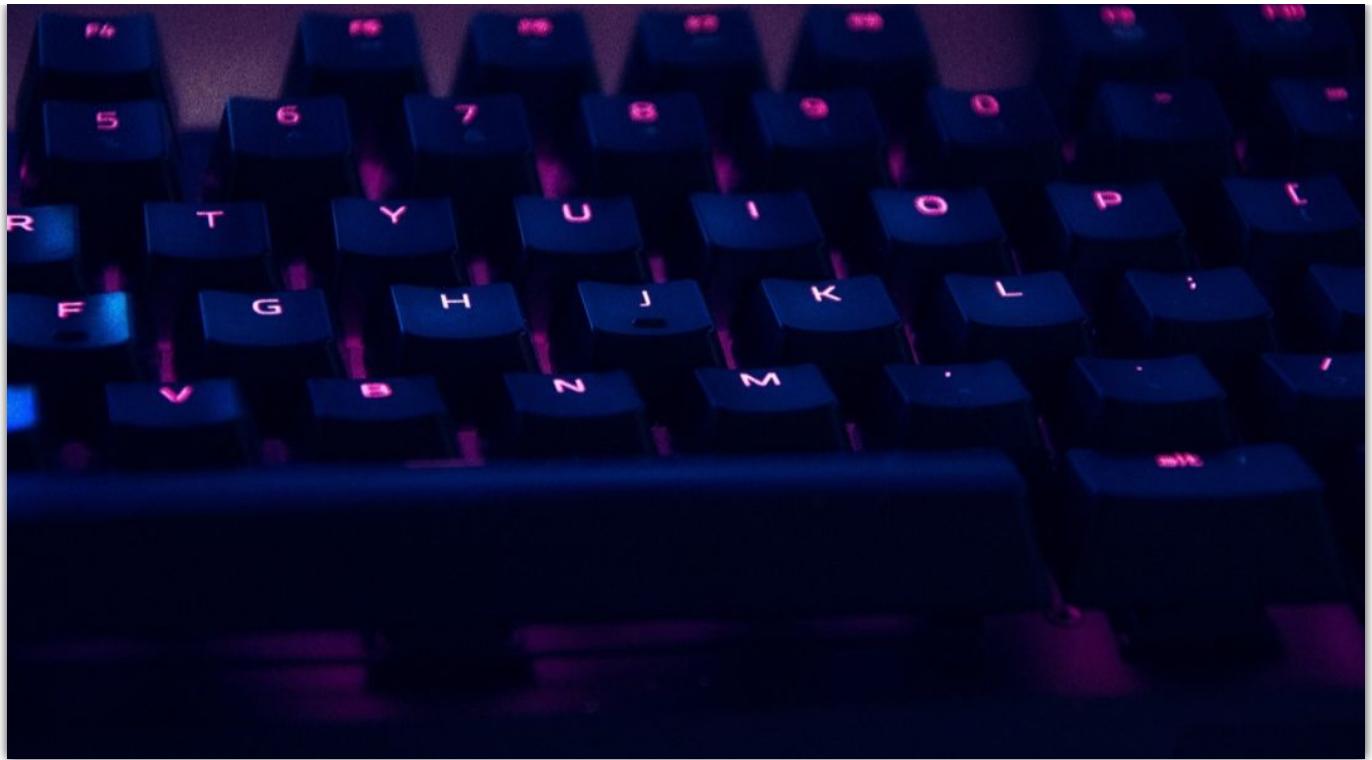


喜欢文章, 点个在看 

声明: pdf仅供学习使用, 一切版权归原创公众号所有; 建议持续关注原创公众号获取最新文章, 学习愉快!

# 腊月二十八，聊聊 MyBatis 中的设计模式

crazyant Java后端 1月22日



作者 | crazyant

链接 | [www.crazyant.net/2022.html](http://www.crazyant.net/2022.html)

虽然我们都知道有26个设计模式，但是大多停留在概念层面，真实开发中很少遇到，Mybatis源码中使用了大量的设计模式，阅读源码并观察设计模式在其中的应用，能够更深入的理解设计模式。

Mybatis至少遇到了以下的设计模式的使用：

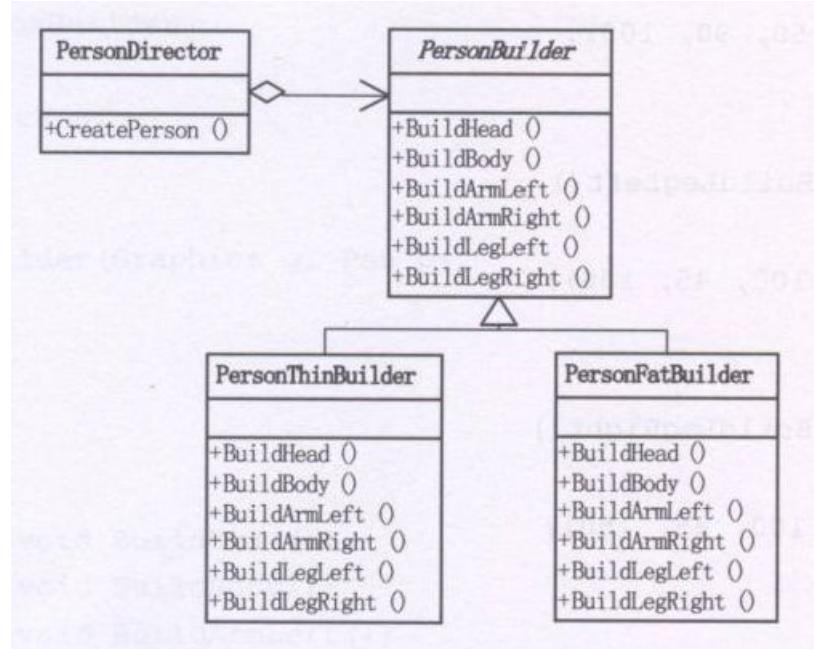
1. Builder模式，例如SqlSessionFactoryBuilder、XMLConfigBuilder、XMLMapperBuilder、XMLStatementBuilder、CacheBuilder；
2. 工厂模式，例如SqlSessionFactory、ObjectFactory、MapperProxyFactory；
3. 单例模式，例如ErrorContext和LogFactory；
4. 代理模式，Mybatis实现的核心，比如MapperProxy、ConnectionLogger，用的jdk的动态代理；还有executor.loader包使用了cglib或者javassist达到延迟加载的效果；
5. 组合模式，例如SqlNode和各个子类ChooseSqlNode等；
6. 模板方法模式，例如BaseExecutor和SimpleExecutor，还有BaseTypeHandler和所有的子类例如IntegerTypeHandler；
7. 适配器模式，例如Log的Mybatis接口和它对jdbc、log4j等各种日志框架的适配实现；
8. 装饰者模式，例如Cache包中的cache.decorators子包中等各个装饰者的实现；
9. 迭代器模式，例如迭代器模式PropertyTokenizer；

接下来挨个模式进行解读，先介绍模式自身的知识，然后解读在Mybatis中怎样应用了该模式。

## 1、Builder模式

Builder模式的定义是“将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。”，它属于创建类模

式，一般来说，如果一个对象的构建比较复杂，超出了构造函数所能包含的范围，就可以使用工厂模式和Builder模式，相对于工厂模式会产出一个完整的产品，Builder应用于更加复杂的对象的构建，甚至只会构建产品的一个部分。



在Mybatis环境的初始化过程中，`SqlSessionFactoryBuilder`会调用`XMLConfigBuilder`读取所有的`MybatisMapConfig.xml`和所有的`*Mapper.xml`文件，构建Mybatis运行的核心对象`Configuration`对象，然后将该`Configuration`对象作为参数构建一个`SqlSessionFactory`对象。

其中`XMLConfigBuilder`在构建`Configuration`对象时，也会调用`XMLMapperBuilder`用于读取`*Mapper`文件，而`XMLMapperBuilder`会使用`XMLStatementBuilder`来读取和build所有的SQL语句。

在这个过程中，有一个相似的特点，就是这些Builder会读取文件或者配置，然后做大量的XpathParser解析、配置或语法的解析、反射生成对象、存入结果缓存等步骤，这么多的工作都不是一个构造函数所能包括的，因此大量采用了Builder模式来解决。

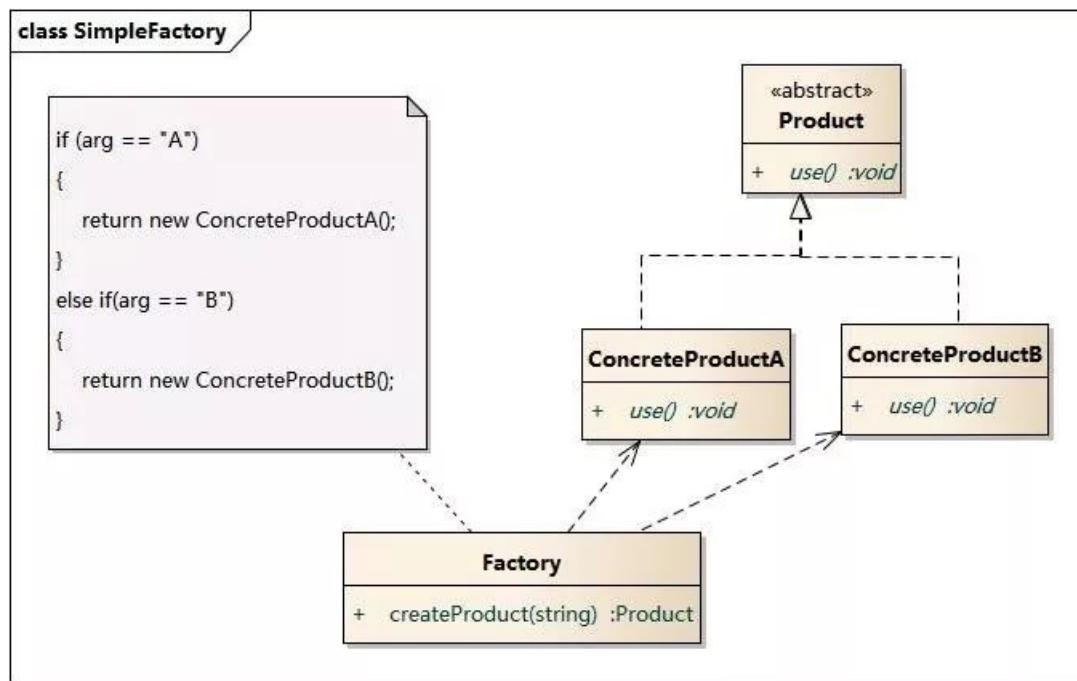
对于builder的具体类，方法都大都用`build*`开头，比如`SqlSessionFactoryBuilder`为例，它包含以下方法：



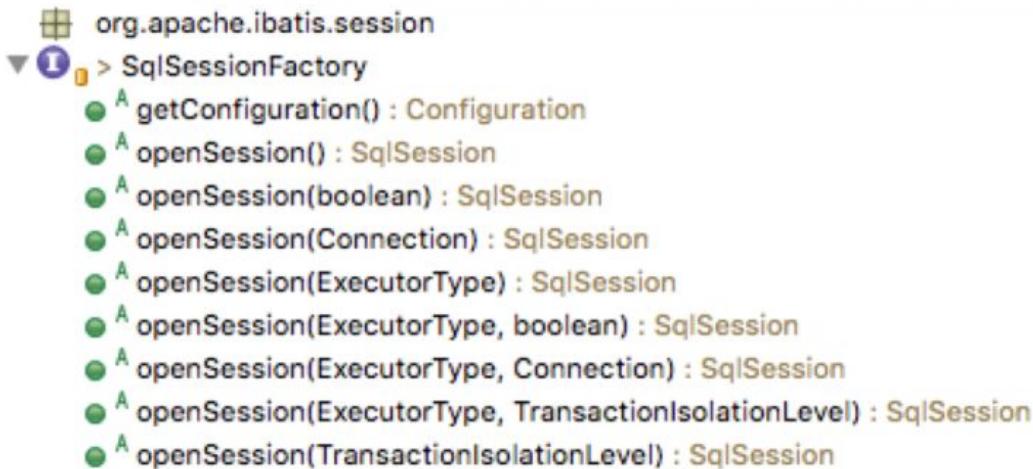
即根据不同的输入参数来构建`SqlSessionFactory`这个工厂对象。

在Mybatis中比如SqlSessionFactory使用的是工厂模式，该工厂没有那么复杂的逻辑，是一个简单工厂模式。

简单工厂模式(Simple Factory Pattern): 又称为静态工厂方法(Static Factory Method)模式，它属于类创建型模式。在简单工厂模式中，可以根据参数的不同返回不同类的实例。简单工厂模式专门定义一个类来负责创建其他类的实例，被创建的实例通常都具有共同的父类。



SqlSession可以认为是一个Mybatis工作的核心的接口，通过这个接口可以执行执行SQL语句、获取Mappers、管理事务。类似于连接MySQL的Connection对象。



可以看到，该Factory的`openSession`方法重载了很多个，分别支持`autoCommit`、`Executor`、`Transaction`等参数的输入，来构建核心的`SqlSession`对象。

在`DefaultSqlSessionFactory`的默认工厂实现里，有一个方法可以看出工厂怎么产出一个产品：

```

private SqlSession openSessionFromDataSource(ExecutorType execType, TransactionIsolationLevel level,
    boolean autoCommit) {
    Transaction tx = null;
    try {
        final Environment environment = configuration.getEnvironment();
        final TransactionFactory transactionFactory = getTransactionFactoryFromEnvironment(environment);
        tx = transactionFactory.newTransaction(environment.getDataSource(), level, autoCommit);
        final Executor executor = configuration.newExecutor(tx, execType);
        return new DefaultSqlSession(configuration, executor, autoCommit);
    } catch (Exception e) {
        closeTransaction(tx); // may have fetched a connection so lets call
        // close()
        throw ExceptionFactory.wrapException("Error opening session. Cause: " + e, e);
    } finally {
        ErrorContext.instance().reset();
    }
}

```

这是一个openSession调用的底层方法，该方法先从configuration读取对应的环境配置，然后初始化TransactionFactory获得一个Transaction对象，然后通过Transaction获取一个Executor对象，最后通过configuration、Executor、是否autoCommit三个参数构建了SqlSession。

在这里其实也可以看到端倪，SqlSession的执行，其实是委托给对应的Executor来进行的。

而对于LogFactory，它的实现代码：

```

public final class LogFactory {
    private static Constructor<? extends Log> logConstructor;

    private LogFactory() {
        // disable construction
    }

    public static Log getLog(Class<?> aClass) {
        return getLog(aClass.getName());
    }
}

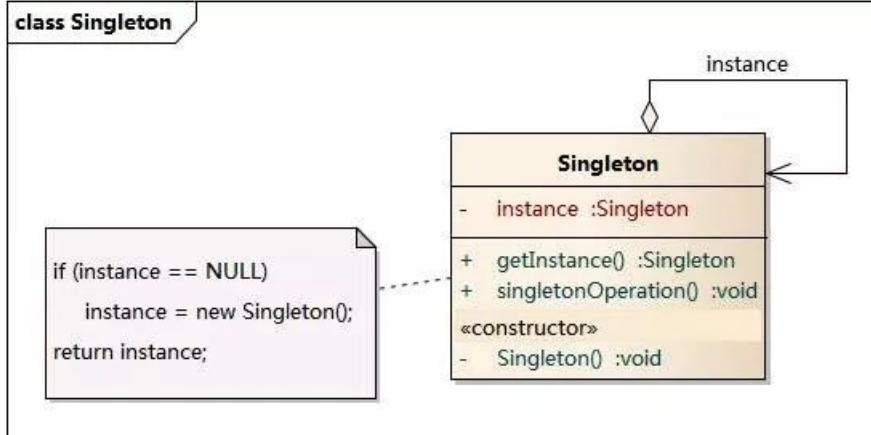
```

这里有个特别的地方，是Log变量的类型是Constructor<?extends Log>，也就是说该工厂生产的不只是一个产品，而是具有Log公共接口的一系列产品，比如Log4jImpl、Slf4jImpl等很多具体的Log。

### 3、单例模式

单例模式(Singleton Pattern)：单例模式确保某一个类只有一个实例，而且自行实例化并向整个系统提供这个实例，这个类称为单例类，它提供全局访问的方法。

单例模式的要点有三个：一是某个类只能有一个实例；二是它必须自行创建这个实例；三是它必须自行向整个系统提供这个实例。单例模式是一种对象创建型模式。单例模式又名单件模式或单态模式。



在Mybatis中有两个地方用到单例模式，`ErrorContext`和`LogFactory`，其中`ErrorContext`是用在每个线程范围内的单例，用于记录该线程的执行环境错误信息，而`LogFactory`则是提供给整个Mybatis使用的日志工厂，用于获得针对项目配置好的日志对象。

`ErrorContext`的单例实现代码：

```

public class ErrorContext {

    private static final ThreadLocal<ErrorContext> LOCAL = new ThreadLocal<ErrorContext>();

    private ErrorContext() {
    }

    public static ErrorContext instance() {
        ErrorContext context = LOCAL.get();
        if (context == null) {
            context = new ErrorContext();
            LOCAL.set(context);
        }
        return context;
    }
}

```

构造函数是`private`修饰，具有一个`static`的局部`instance`变量和一个获取`instance`变量的方法，在获取实例的方法中，先判断是否为空如果是的话就先创建，然后返回构造好的对象。

只是这里有个有趣的地方是，`LOCAL`的静态实例变量使用了`ThreadLocal`修饰，也就是说它属于每个线程各自的数据，而在`instance()`方法中，先获取本线程的该实例，如果没有就创建该线程独有的`ErrorContext`。

#### 4、代理模式

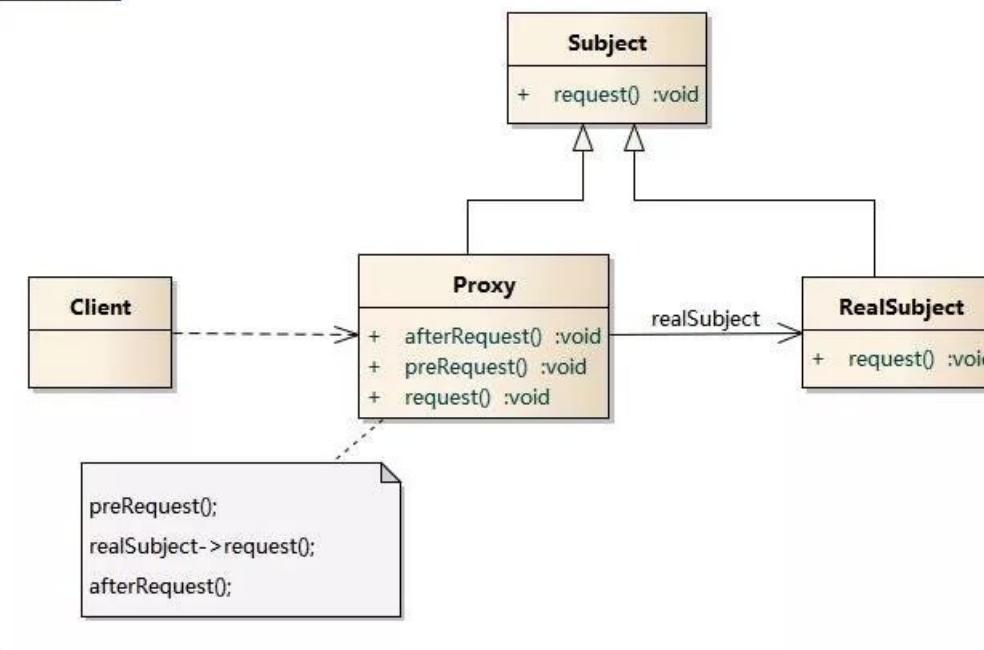
代理模式可以认为是Mybatis的核心使用的模式，正是由于这个模式，我们只需要编写`Mapper.java`接口，不需要实现，由Mybatis后台帮我们完成具体SQL的执行。

代理模式(Proxy Pattern)：给某一个对象提供一个代理，并由代理对象控制对原对象的引用。代理模式的英文叫做Proxy或Surrogate，它是一种对象结构型模式。

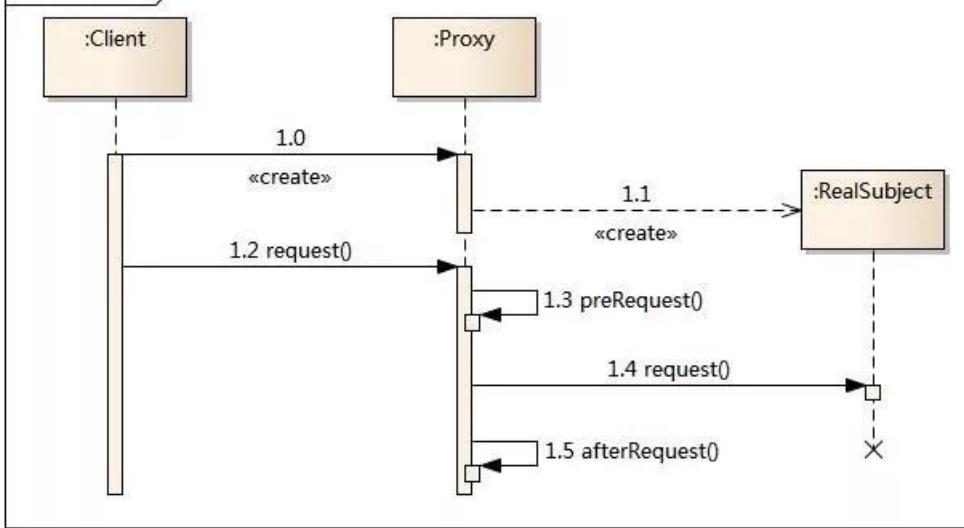
代理模式包含如下角色：

- Subject: 抽象主题角色
- Proxy: 代理主题角色
- RealSubject: 真实主题角色

class Proxy



sd seq\_Proxy



这里有两个步骤，第一个是提前创建一个Proxy，第二个是使用的时候会自动请求Proxy，然后由Proxy来执行具体事务；

当我们使用Configuration的getMapper方法时，会调用mapperRegistry.getMapper方法，而该方法又会调用 mapperProxyFactory.newInstance(sqlSession)来生成一个具体的代理：

```

/**
 * @author Lasse Voss
 */
public class MapperProxyFactory<T> {

    private final Class<T> mapperInterface;
    private final Map<Method, MapperMethod> methodCache = new ConcurrentHashMap<Method, MapperMethod>();

    public MapperProxyFactory(Class<T> mapperInterface) {
        this.mapperInterface = mapperInterface;
    }

    public Class<T> getMapperInterface() {
        return mapperInterface;
    }

    public Map<Method, MapperMethod> getMethodCache() {
        return methodCache;
    }

    @SuppressWarnings("unchecked")
    protected T newInstance(MapperProxy<T> mapperProxy) {
        return (T) Proxy.newProxyInstance(mapperInterface.getClassLoader(), new Class[] { mapperInterface },
            mapperProxy);
    }

    public T newInstance(SqlSession sqlSession) {
        final MapperProxy<T> mapperProxy = new MapperProxy<T>(sqlSession, mapperInterface, methodCache);
        return newInstance(mapperProxy);
    }
}

```

在这里，先通过T newInstance(SqlSession sqlSession)方法会得到一个MapperProxy对象，然后调用T newInstance(MapperProxy<T> mapperProxy)生成代理对象然后返回。

而查看MapperProxy的代码，可以看到如下内容：

```

public class MapperProxy<T> implements InvocationHandler, Serializable {

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

```

非常典型的，该MapperProxy类实现了InvocationHandler接口，并且实现了该接口的invoke方法。

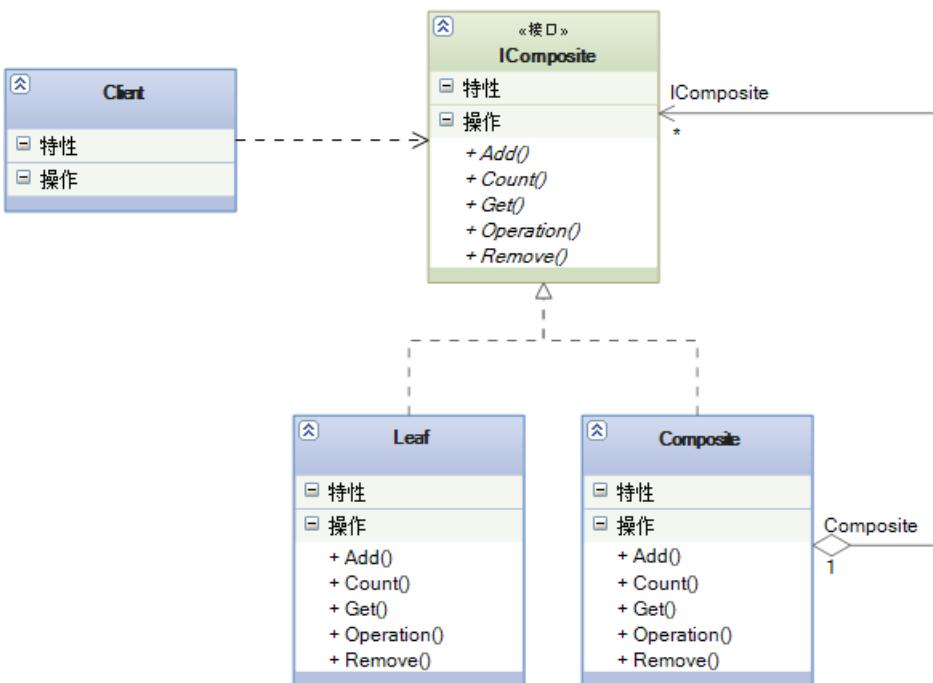
通过这种方式，我们只需要编写Mapper.java接口类，当真正执行一个Mapper接口的时候，就会转发给MapperProxy.invoke方法，而该方法则会调用后续的sqlSession.createExecutor().execute().prepareStatement等一系列方法，完成SQL的执行和返回。

## 5. 组合模式

组合模式组合多个对象形成树形结构以表示“整体-部分”的结构层次。

组合模式对单个对象(叶子对象)和组合对象(组合对象)具有一致性，它将对象组织到树结构中，可以用来描述整体与部分的关系。同时它也模糊了简单元素(叶子对象)和复杂元素(容器对象)的概念，使得客户能够像处理简单元素一样来处理复杂元素，从而使客户程序能够与复杂元素的内部结构解耦。

在使用组合模式中需要注意一点也是组合模式最关键的地方：叶子对象和组合对象实现相同的接口。这就是组合模式能够将叶子节点和对象节点进行一致处理的原因。



Mybatis支持动态SQL的强大功能，比如下面的这个SQL：

```
<update id="update" parameterType="org.format.dynamicproxy.mybatis.bean.User">
    UPDATE users
    <trim prefix="SET" prefixOverrides=",">
        <if test="name != null and name != "">
            name = #{name}
        </if>
        <if test="age != null and age != "">
            , age = #{age}
        </if>
        <if test="birthday != null and birthday != "">
            , birthday = #{birthday}
        </if>
    </trim>
    where id = ${id}
</update>
```

在这里面使用到了trim、if等动态元素，可以根据条件来生成不同情况下的SQL；

在DynamicSqlSource.getBoundSql方法里，调用了rootSqlNode.apply(context)方法，apply方法是所有的动态节点都实现的接口：

```
public interface SqlNode {  
    boolean apply(DynamicContext context);  
}
```

对于实现该SqlSource接口的所有节点，就是整个组合模式树的各个节点：

Type hierarchy of 'org.apache.ibatis.scripting.xmltags.SqlNode':

```
SqlNode - org.apache.ibatis.scripting.xmltags  
└─ ChooseSqlNode - org.apache.ibatis.scripting.xmltags  
└─ ForEachSqlNode - org.apache.ibatis.scripting.xmltags  
└─ IfSqlNode - org.apache.ibatis.scripting.xmltags  
└─ MixedSqlNode - org.apache.ibatis.scripting.xmltags  
└─ StaticTextSqlNode - org.apache.ibatis.scripting.xmltags  
└─ TextSqlNode - org.apache.ibatis.scripting.xmltags  
└─ TrimSqlNode - org.apache.ibatis.scripting.xmltags  
    └─ SetSqlNode - org.apache.ibatis.scripting.xmltags  
    └─ WhereSqlNode - org.apache.ibatis.scripting.xmltags  
└─ VarDeclSqlNode - org.apache.ibatis.scripting.xmltags
```

Press 'Shift+T' to see the supertype hierarchy

组合模式的简单之处在在于，所有的子节点都是同一类节点，可以递归的向下执行，比如对于TextSqlNode，因为它是最底层的叶子节点，所以直接将对应的内容append到SQL语句中：

```
@Override  
public boolean apply(DynamicContext context) {  
    GenericTokenParser parser = createParser(new BindingTokenParser(context, injectionFilter));  
    context.appendSql(parser.parse(text));  
    return true;  
}
```

但是对于IfSqlNode，就需要先做判断，如果判断通过，仍然会调用子元素的SqlNode，即contents.apply方法，实现递归的解析。

```
@Override  
public boolean apply(DynamicContext context) {  
    if (evaluator.evaluateBoolean(test, context.getBindings())) {  
        contents.apply(context);  
        return true;  
    }  
    return false;  
}
```

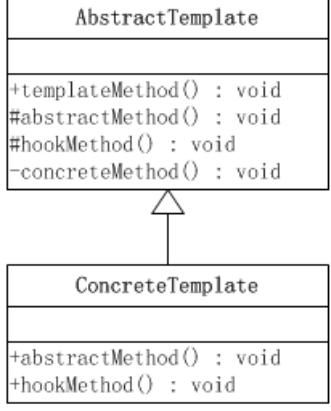
## 6、模板方法模式

模板方法模式是所有模式中最为常见的几个模式之一，是基于继承的代码复用的基本技术。

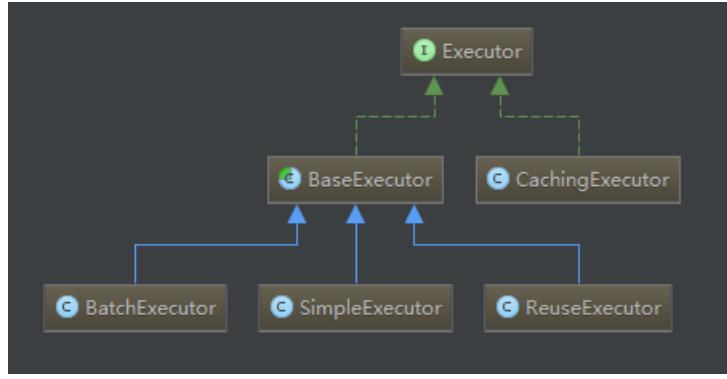
模板方法模式需要开发抽象类和具体子类的设计师之间的协作。一个设计师负责给出一个算法的轮廓和骨架，另一些设计师则负

责给出这个算法的各个逻辑步骤。代表这些具体逻辑步骤的方法称做基本方法(primitive method);而将这些基本方法汇总起来的方法叫做模板方法(template method)，这个设计模式的名字就是从此而来。

模板类定义一个操作中的算法的骨架，而将一些步骤延迟到子类中。使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤。



在Mybatis中，sqlSession的SQL执行，都是委托给Executor实现的，Executor包含以下结构：



其中的BaseExecutor就采用了模板方法模式，它实现了大部分的SQL执行逻辑，然后把以下几个方法交给子类定制化完成：

```
protected abstract int doUpdate(MappedStatement ms, Object parameter) throws SQLException;
protected abstract List<BatchResult> doFlushStatements(boolean isRollback) throws SQLException;
protected abstract <E> List<E> doQuery(MappedStatement ms, Object parameter, RowBounds rowBounds,
                                         ResultHandler resultHandler, BoundSql boundSql) throws SQLException;
```

该模板方法类有几个子类的具体实现，使用了不同的策略：

- 简单SimpleExecutor：每执行一次update或select，就开启一个Statement对象，用完立刻关闭Statement对象。（可以是Statement或PreparedStatement对象）
- 重用ReuseExecutor：执行update或select，以sql作为key查找Statement对象，存在就使用，不存在就创建，用完后，不关闭Statement对象，而是放置于Map<String, Statement>内，供下一次使用。（可以是Statement或PreparedStatement对象）
- 批量BatchExecutor：执行update（没有select，JDBC批处理不支持select），将所有sql都添加到批处理中(addBatch())，等待统一执行(executeBatch())，它缓存了多个Statement对象，每个Statement对象都是

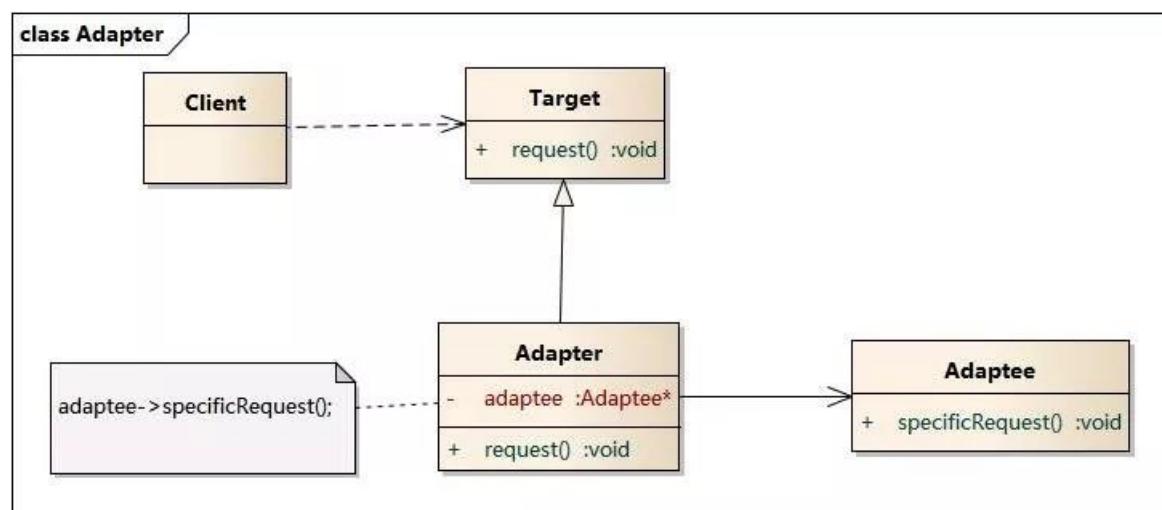
addBatch()完毕后，等待逐一执行executeBatch()批处理的；BatchExecutor相当于维护了多个桶，每个桶里都装了很多属于自己的SQL，就像苹果蓝里装了很多苹果，番茄蓝里装了很多番茄，最后，再统一倒进仓库。（可以是Statement或PreparedStatement对象）

比如在SimpleExecutor中这样实现update方法：

```
@Override  
public int doUpdate(MappedStatement ms, Object parameter) throws SQLException {  
    Statement stmt = null;  
    try {  
        Configuration configuration = ms.getConfiguration();  
        StatementHandler handler = configuration.newStatementHandler(this, ms, parameter, RowBounds.DEFAULT, null,  
            null);  
        stmt = prepareStatement(handler, ms.createStatementLog());  
        return handler.update(stmt);  
    } finally {  
        closeStatement(stmt);  
    }  
}
```

## 7、适配器模式

适配器模式(Adapter Pattern)：将一个接口转换成客户希望的另一个接口，适配器模式使接口不兼容的那些类可以一起工作，其别名为包装器(Wrapper)。适配器模式既可以作为类结构型模式，也可以作为对象结构型模式。



在Mybatis的logging包中，有一个Log接口：

```
/**  
 * @author Clinton Begin  
 */  
  
public interface Log {  
  
    boolean isDebugEnabled();  
  
    boolean isTraceEnabled();  
  
    void error(String s, Throwable e);  
  
    void error(String s);  
  
    void debug(String s);  
  
    void trace(String s);  
  
    void warn(String s);  
  
}
```

该接口定义了Mybatis直接使用的日志方法，而Log接口具体由谁来实现呢？Mybatis提供了多种日志框架的实现，这些实现都匹配这个Log接口所定义的接口方法，最终实现了所有外部日志框架到Mybatis日志包的适配：

Type hierarchy of 'org.apache.ibatis.logging.Log':

- ▼ 1 Log - org.apache.ibatis.logging
- C JakartaCommonsLoggingImpl - org.apache.ibatis.logging.commons
- C Jdk14LoggingImpl - org.apache.ibatis.logging.jdk14
- C Log4j2AbstractLoggerImpl - org.apache.ibatis.logging.log4j2
- C Log4j2Impl - org.apache.ibatis.logging.log4j2
- C Log4j2LoggerImpl - org.apache.ibatis.logging.log4j2
- C Log4jImpl - org.apache.ibatis.logging.log4j
- C NoLoggingImpl - org.apache.ibatis.logging.nologging
- C Slf4jImpl - org.apache.ibatis.logging.slf4j
- C Slf4jLocationAwareLoggerImpl - org.apache.ibatis.logging.slf4j
- C Slf4jLoggerImpl - org.apache.ibatis.logging.slf4j
- C StdOutImpl - org.apache.ibatis.logging.stdout

比如对于Log4jImpl的实现来说，该实现持有了org.apache.log4j.Logger的实例，然后所有的日志方法，均委托该实例来实现。

```

public class Log4jImpl implements Log {

private static final String FQCN = Log4jImpl.class.getName();

private Logger log;

public Log4jImpl(String clazz) {
    log = Logger.getLogger(clazz);
}

@Override
public boolean isDebugEnabled() {
    return log.isDebugEnabled();
}

@Override
public boolean isTraceEnabled() {
    return log.isTraceEnabled();
}

@Override
public void error(String s, Throwable e) {
    log.log(FQCN, Level.ERROR, s, e);
}

@Override
public void error(String s) {
    log.log(FQCN, Level.ERROR, s, null);
}

@Override
public void debug(String s) {
    log.log(FQCN, Level.DEBUG, s, null);
}

@Override
public void trace(String s) {
    log.log(FQCN, Level.TRACE, s, null);
}

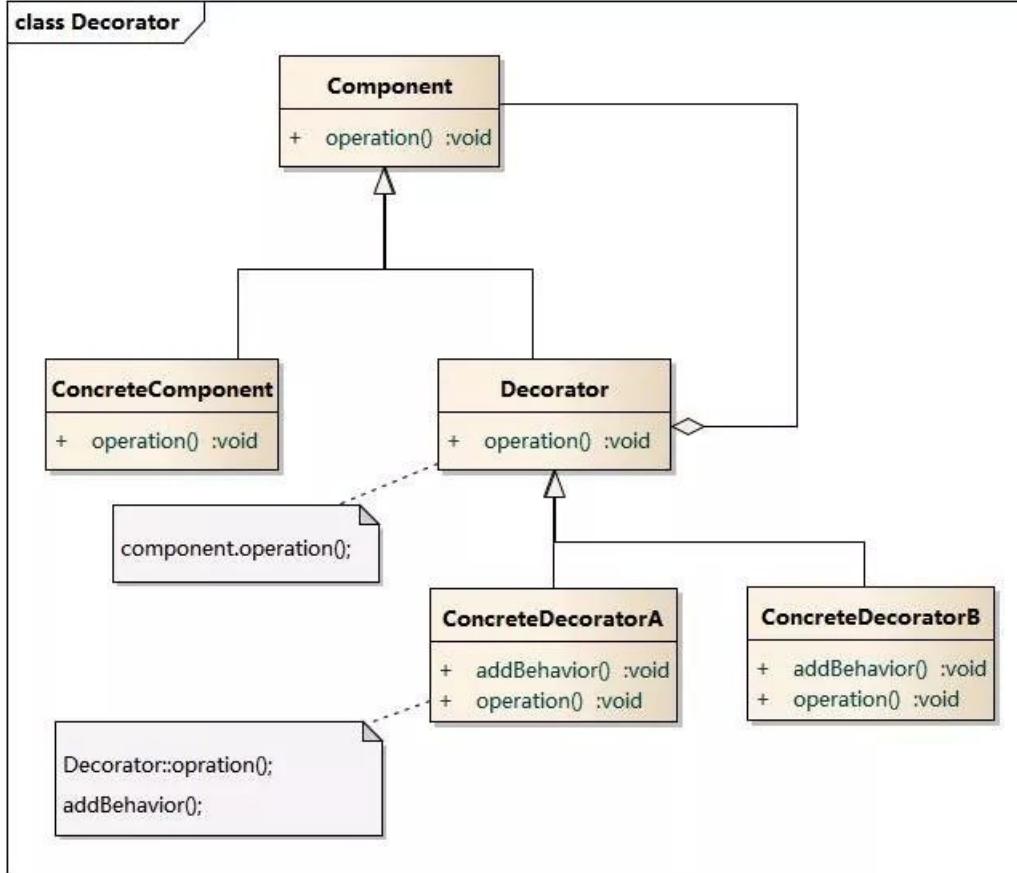
@Override
public void warn(String s) {
    log.log(FQCN, Level.WARN, s, null);
}

}

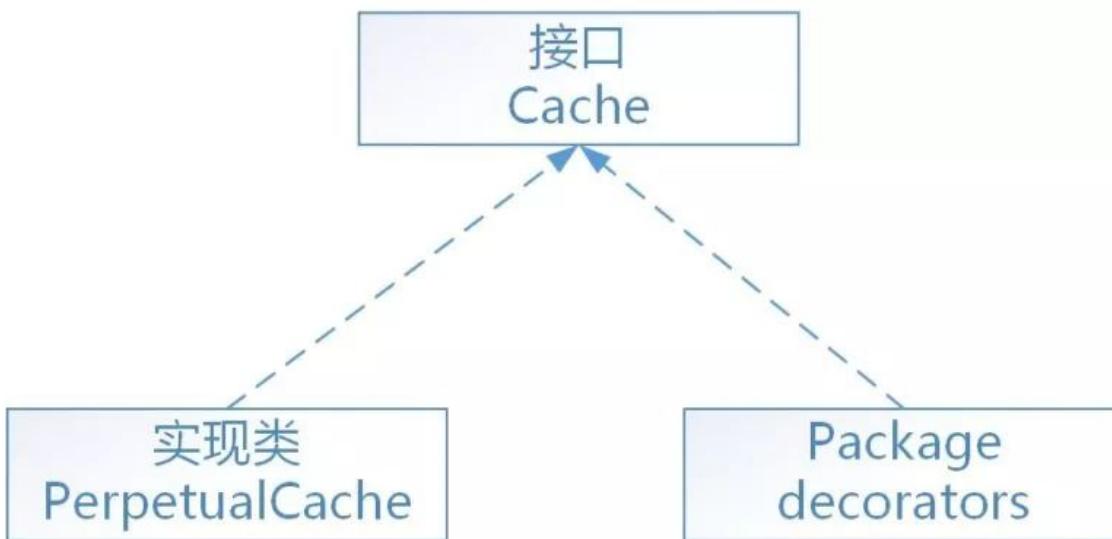
```

## 8、装饰者模式

装饰模式(Decorator Pattern) :动态地给一个对象增加一些额外的职责(Responsibility),就增加对象功能来说,装饰模式比生成子类实现更为灵活。其别名也可以称为包装器(Wrapper),与适配器模式的别名相同,但它们适用于不同的场合。根据翻译的不同,装饰模式也有人称之为“油漆工模式”,它是一种对象结构型模式。



在mybatis中，缓存的功能由根接口Cache (`org.apache.ibatis.cache.Cache`) 定义。整个体系采用装饰器设计模式，数据存储和缓存的基本功能由PerpetualCache (`org.apache.ibatis.cache.impl.PerpetualCache`) 永久缓存实现，然后通过一系列的装饰器来对PerpetualCache永久缓存进行缓存策略等方便的控制。如下图：



用于装饰PerpetualCache的标准装饰器共有8个（全部在`org.apache.ibatis.cache.decorators`包中）：

1. FifoCache：先进先出算法，缓存回收策略
2. LoggingCache：输出缓存命中的日志信息
3. LruCache：最近最少使用算法，缓存回收策略
4. ScheduledCache：调度缓存，负责定时清空缓存
5. SerializedCache：缓存序列化和反序列化存储
6. SoftCache：基于软引用实现的缓存管理策略
7. SynchronizedCache：同步的缓存装饰器，用于防止多线程并发访问
8. WeakCache：基于弱引用实现的缓存管理策略

另外，还有一个特殊的装饰器TransactionalCache：事务性的缓存

正如大多数持久层框架一样，mybatis缓存同样分为一级缓存和二级缓存

- 一级缓存，又叫本地缓存，是PerpetualCache类型的永久缓存，保存在执行器中（BaseExecutor），而执行器又在SqlSession（DefaultSqlSession）中，所以一级缓存的生命周期与SqlSession是相同的。
- 二级缓存，又叫自定义缓存，实现了Cache接口的类都可以作为二级缓存，所以可配置如encache等的第三方缓存。二级缓存以namespace名称空间为其唯一标识，被保存在Configuration核心配置对象中。

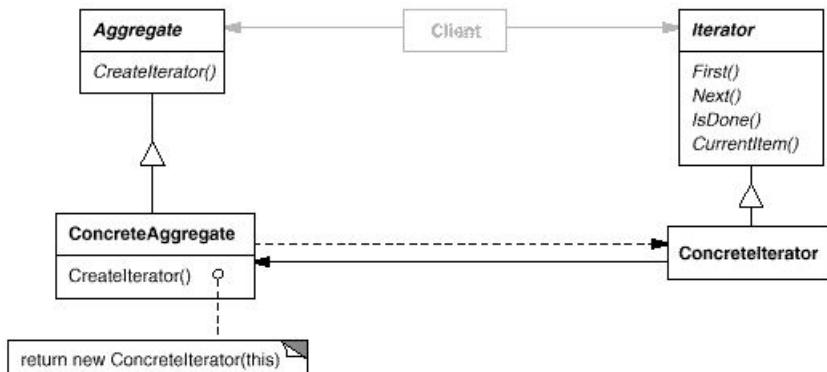
二级缓存对象的默认类型为PerpetualCache，如果配置的缓存是默认类型，则mybatis会根据配置自动追加一系列装饰器。

Cache对象之间的引用顺序为：

SynchronizedCache->LoggingCache->SerializedCache->ScheduledCache->LruCache->PerpetualCache

## 9、迭代器模式

迭代器（Iterator）模式，又叫做游标（Cursor）模式。GOF给出的定义为：提供一种方法访问一个容器（container）对象中各个元素，而又不需暴露该对象的内部细节。



Java的Iterator就是迭代器模式的接口，只要实现了该接口，就相当于应用了迭代器模式：

### ▼ ① Iterator<E>

- `D forEachRemaining(Consumer<? super E>) : void`
- `A hasNext() : boolean`
- `A next() : E`
- `D remove() : void`

比如Mybatis的PropertyTokenizer是property包中的重量级类，该类会被reflection包中其他的类频繁的引用到。这个类实现了Iterator接口，在使用时经常被用到的是Iterator接口中的hasNext这个函数。

```

public class PropertyTokenizer implements Iterator<PropertyTokenizer> {
    private String name;
    private String indexedName;
    private String index;
    private String children;

    public PropertyTokenizer(String fullname) {
        int delim = fullname.indexOf('.');
        if (delim > -1) {
            name = fullname.substring(0, delim);
            children = fullname.substring(delim + 1);
        } else {
            name = fullname;
            children = null;
        }
        indexedName = name;
        delim = name.indexOf('[');
        if (delim > -1) {
            index = name.substring(delim + 1, name.length() - 1);
            name = name.substring(0, delim);
        }
    }

    public String getName() {
        return name;
    }

    public String getIndex() {
        return index;
    }

    public String getIndexedName() {
        return indexedName;
    }

    public String getChildren() {
        return children;
    }

    @Override
    public boolean hasNext() {
        return children != null;
    }

    @Override
    public PropertyTokenizer next() {
        return new PropertyTokenizer(children);
    }

    @Override
    public void remove() {
        throw new UnsupportedOperationException(
            "Remove is not supported, as it has no meaning in the context of properties.");
    }
}

```

可以看到，这个类传入一个字符串到构造函数，然后提供了iterator方法对解析后的子串进行遍历，是一个很常用的方法类。

参考资料：

- 图说设计模式：[http://design-patterns.readthedocs.io/zh\\_CN/latest/index.html](http://design-patterns.readthedocs.io/zh_CN/latest/index.html)

- 深入浅出Mybatis系列（十）—SQL执行流程分析（源码篇）：<http://www.cnblogs.com/dongying/p/4142476.html>
- 设计模式读书笔记—组合模式<http://www.cnblogs.com/chenssy/p/3299719.html>
- Mybatis3.3.x技术内幕（四）：五鼠闹东京之执行器Executor设计原  
本<http://blog.csdn.net/wagcy/article/details/32963235>
- mybatis缓存机制详解（一）——Cache <https://my.oschina.net/lixin91/blog/620068>

本文地址：<http://crazyant.net/2022.html>，转载请注明来源。



微信扫描二维码，关注我的公众号

声明：pdf仅供学习使用，一切版权归原创公众号所有；建议持续关注原创公众号获取最新文章，学习愉快！

# 面试官：讲一下 Mybatis 初始化原理

亦山 Java后端 2019-11-23

点击上方 Java后端, 选择 [设为星标](#)

优质文章, 及时送达

作者 | 亦山

原文 | <https://urlify.cn/zaYRJv>

对于任何框架而言，在使用前都要进行一系列的初始化，MyBatis也不例外。本章将通过以下几点详细介绍MyBatis的初始化过程。

1. MyBatis的初始化做了什么
2. MyBatis基于XML配置文件创建Configuration对象的过程
3. 手动加载XML配置文件创建Configuration对象完成初始化，创建并使用SqlSessionFactory对象
4. 涉及到的设计模式

## 一、MyBatis的初始化做了什么

任何框架的初始化，无非是加载自己运行时所需要的配置信息。MyBatis的配置信息，大概包含以下信息，其高级结构如下：

### × configuration 配置

- × properties 属性
- × settings 设置
- × typeAliases 类型命名
- × typeHandlers 类型处理器
- × objectFactory 对象工厂
- × plugins 插件
- × environments 环境
- × environment 环境变量
- × transactionManager 事务管理器
- × dataSource 数据源

### × 映射器

MyBatis的上述配置信息会配置在XML配置文件中，那么，这些信息被加载进入MyBatis内部，MyBatis是怎样维护的呢？

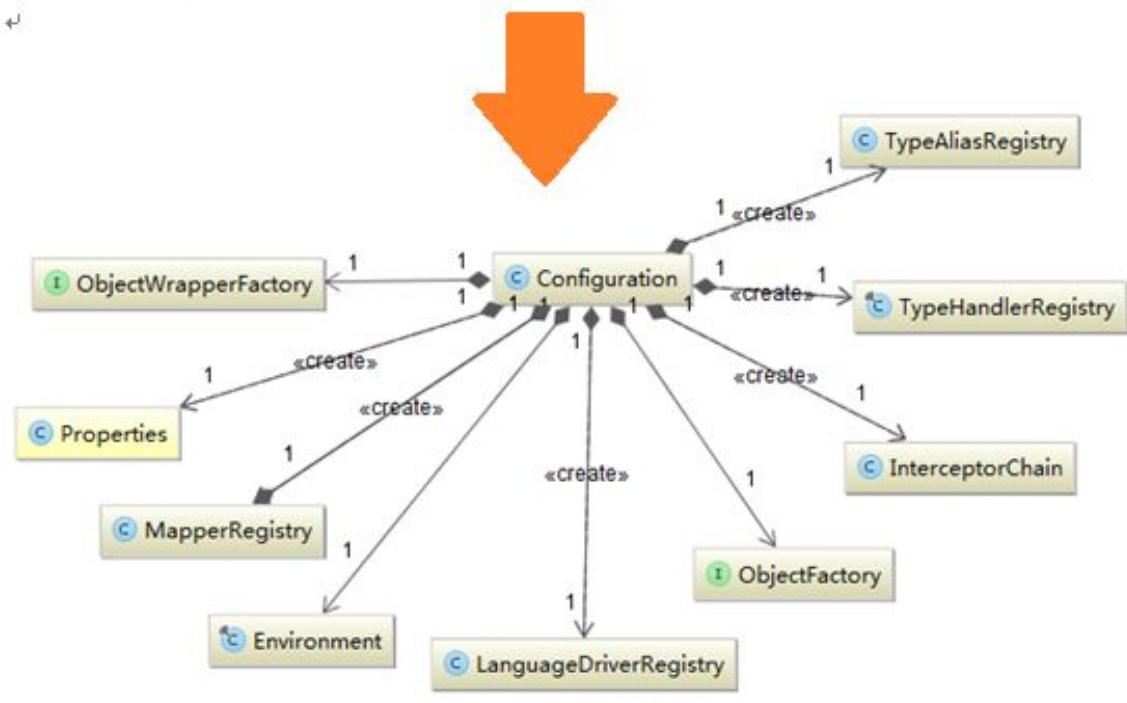
MyBatis采用了一个非常直白和简单的方式---使用 `org.apache.ibatis.session.Configuration` 对象作为一个所有配置信息的容器，`Configuration`对象的组织结构和XML配置文件的组织结构几乎完全一样（当然，`Configuration`对象的功能并不限于此，它还负责创建一些MyBatis内部使用的对象，如`Executor`等，这将在后续的文章中讨论）。如下图所示：

```

mybatis-config.xml
  ▼ <configuration>
    ▼ <environments default="development">
      ▼ <environment name="development">
        <dataSource type="UNPOOLED"/>
        <transactionManager type="JDBC"/>
      </environment>
    </environments>
    <mappers>
      <mapper resource="org/apache/ibatis/submitted/foreach/Mapper.x
    </mappers>

```

Narrow down on typing



Designed by LouLuan  
<http://blog.csdn.net/luanlouis>

MyBatis根据初始化好Configuration信息，这时候用户就可以使用MyBatis进行数据库操作了。

可以说，MyBatis初始化的过程，就是创建 Configuration对象的过程。

MyBatis的初始化可以有两种方式：

- 基于XML配置文件：基于XML配置文件的方式是将MyBatis的所有配置信息放在XML文件中，MyBatis通过加载并解析XML配置文件，将配置文信息组装成内部的Configuration对象
- 基于Java API：这种方式不使用XML配置文件，需要MyBatis使用者在Java代码中，手动创建Configuration对象，然后将配置参数set 进入Configuration对象中

(PS： MyBatis具体配置信息有哪些，又分别表示什么意思，不在本文的叙述范围，读者可以参考我的《Java Persistence withMyBatis 3 (中文版)》的第二章 引导MyBatis中有详细的描述)

接下来我们将通过 基于XML配置文件方式的MyBatis初始化，深入探讨MyBatis是如何通过配置文件构建Configuration对象，并使用它的。

## 二、MyBatis基于XML配置文件创建Configuration对象的过程

现在就从使用MyBatis的简单例子入手，深入分析一下MyBatis是怎样完成初始化的，都初始化了什么。看以下代码：

```
String resource = "mybatis-config.xml";
InputStream inputStream = Resources.getResourceAsStream(resource);
SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);
SqlSession sqlSession = sqlSessionFactory.openSession();
List list = sqlSession.selectList("com.foo.bean.BlogMapper.queryAllBlogInfo");
```

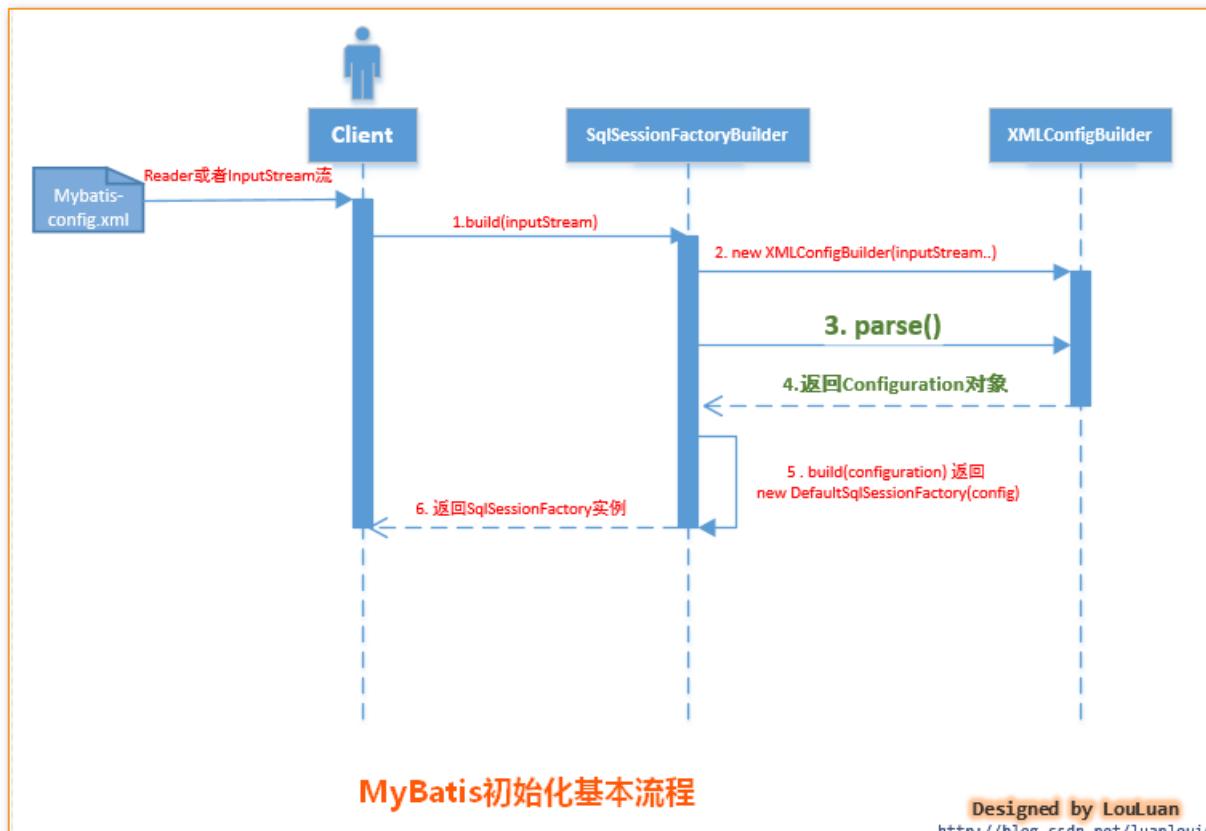
有过MyBatis使用经验的读者会知道，上述语句的作用是执行com.foo.bean.BlogMapper.queryAllBlogInfo 定义的SQL语句，返回一个List结果集。总的来说，上述代码经历了mybatis初始化 --> 创建SqlSession --> 执行SQL语句 返回结果三个过程。

上述代码的功能是根据配置文件mybatis-config.xml 配置文件，创建SqlSessionFactory对象，然后产生SqlSession，执行SQL语句。而mybatis的初始化就发生在第三句：SqlSessionFactory       sqlSessionFactory       =       new SqlSessionFactoryBuilder().build(inputStream); 现在就让我们看看第三句到底发生了什么。

### MyBatis初始化基本过程：

SqlSessionFactoryBuilder根据传入的数据流生成Configuration对象，然后根据Configuration对象创建默认的SqlSessionFactory实例。

初始化的基本过程如下序列图所示：



由上图所示，mybatis初始化要经过简单的以下几步：

1. 调用SqlSessionFactoryBuilder对象的build(inputStream)方法；
2. SqlSessionFactoryBuilder会根据输入流inputStream等信息创建XMLConfigBuilder对象；
3. SqlSessionFactoryBuilder调用XMLConfigBuilder对象的parse()方法；

4. XMLConfigBuilder对象返回Configuration对象；
5. SqlSessionFactoryBuilder根据Configuration对象创建一个DefaultSessionFactory对象；
6. SqlSessionFactoryBuilder返回 DefaultSessionFactory对象给Client，供Client使用。

SqlSessionFactoryBuilder相关的代码如下所示：

```

public SqlSessionFactory build(InputStream inputStream)
{
    return build(inputStream, null, null);
}
public SqlSessionFactory build(InputStream inputStream, String environment,
Properties properties)
{
    try
    {
        //2. 创建XMLConfigBuilder对象用来解析XML配置文件，生成Configuration对象
        XMLConfigBuilder parser = new XMLConfigBuilder(inputStream, environment,
properties);
        //3. 将XML配置文件内的信息解析成Java对象Configuration对象
        Configuration config = parser.parse();
        //4. 根据Configuration对象创建出SqlSessionFactory对象
        return build(config);
    }
    catch (Exception e)
    {
        throw ExceptionFactory.wrapException("Error building SqlSession.", e);
    }
    finally
    {

```

```

        ErrorContext.instance().reset();
        try
        {
            inputStream.close();
        }
        catch (IOException e)
        {
            // Intentionally ignore. Prefer previous error.
        }
    }
}
//从此处可以看出，MyBatis内部通过Configuration对象来创建SqlSessionFactory，用户也可以
//自己通过API构造好Configuration对象，调用此方法创建SqlSessionFactory
public SqlSessionFactory build(Configuration config)
{
    return new DefaultSqlSessionFactory(config);
}
```

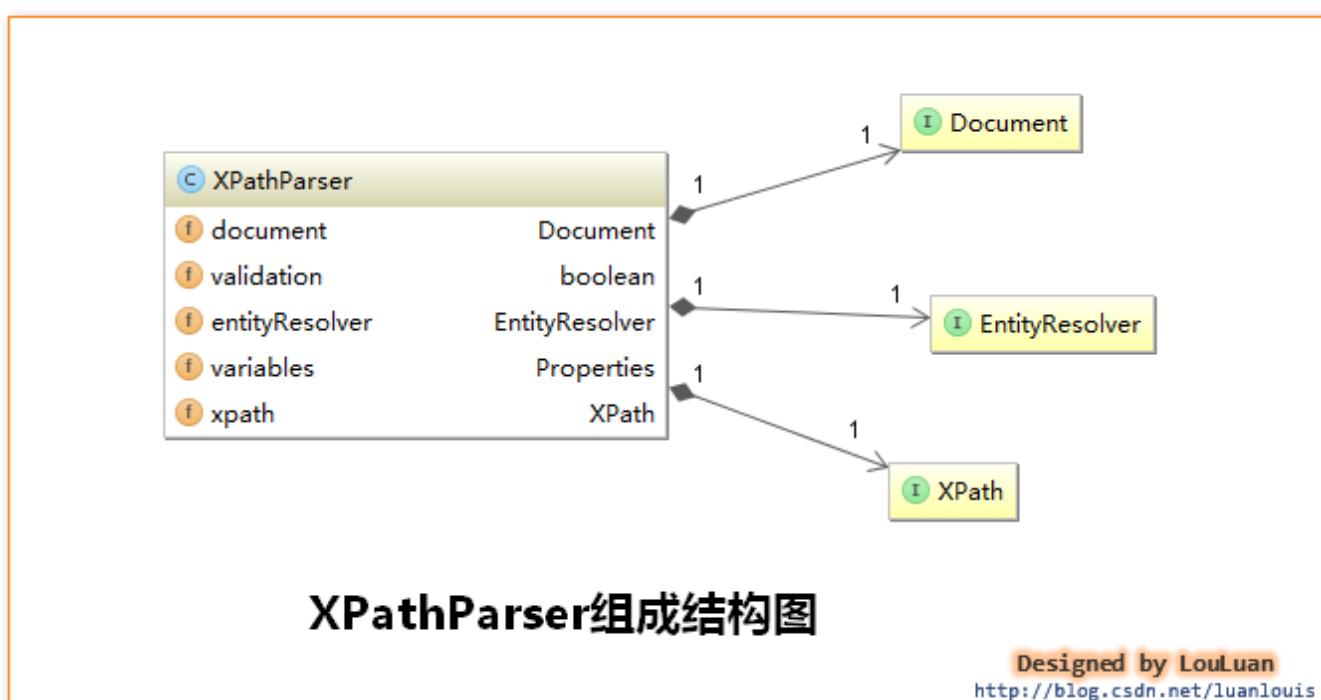
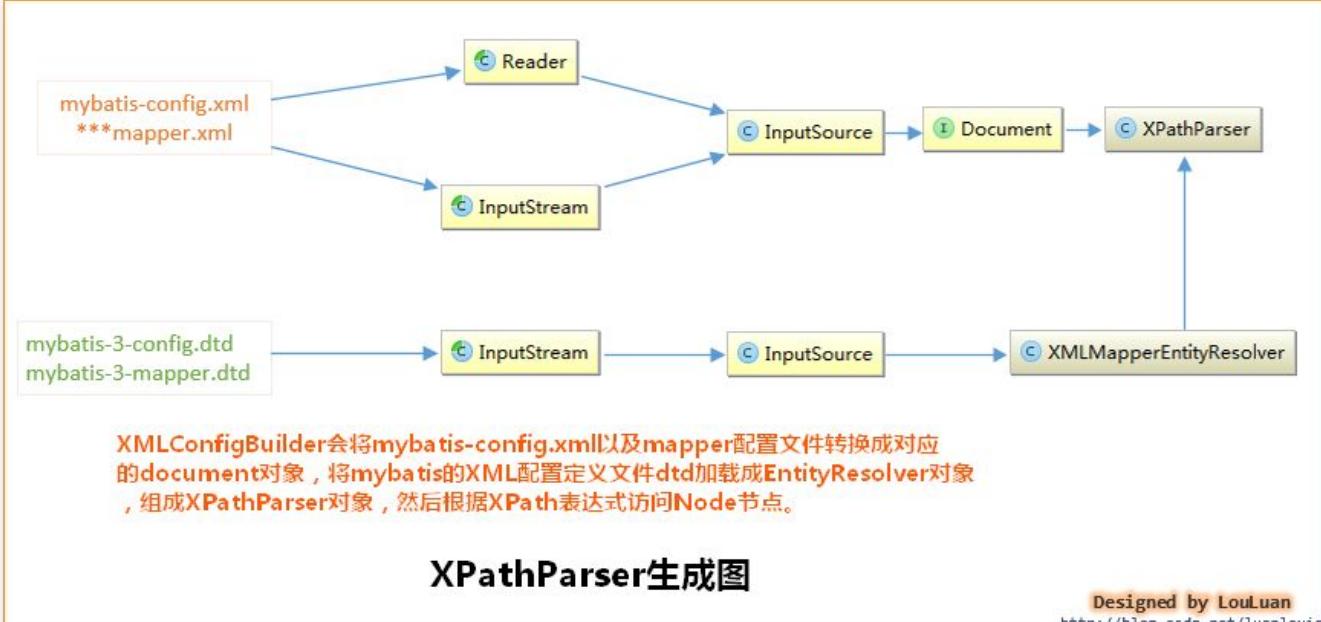
上述的初始化过程中，涉及到了以下几个对象：

- **SqlSessionFactoryBuilder** :SqlSessionFactory的构造器，用于创建SqlSessionFactory，采用了Builder设计模式
- **Configuration** :该对象是mybatis-config.xml文件中所有mybatis配置信息
- **SqlSessionFactory**:SqlSession工厂类，以工厂形式创建SqlSession对象，采用了Factory工厂设计模式
- **XmlConfigParser** :负责将mybatis-config.xml配置文件解析成Configuration对象，共SqlSessionFactoryBuilder使用，创建SqlSessionFactory

## 创建Configuration对象的过程

接着上述的 MyBatis初始化基本过程讨论,当SqlSessionFactoryBuilder执行build()方法,调用了XMLConfigBuilder的parse()方法,然后返回了Configuration对象。那么parse()方法是如何处理XML文件,生成Configuration对象的呢?

1. XMLConfigBuilder会将XML配置文件的信息转换为Document对象,而XML配置定义文件DTD转换成XMLMapperEntityResolver对象,然后将二者封装到XpathParser对象中,XpathParser的作用是提供根据XPath表达式获取基本的DOM节点Node信息的操作。如下图所示:



2. 之后XMLConfigBuilder调用parse()方法:会从XPathParser中取出 <configuration>节点对应的Node对象,然后解析此Node节点的子Node:properties, settings, typeAliases,typeHandlers, objectFactory, objectWrapperFactory, plugins, environments,databaseIdProvider, mappers

```

public Configuration parse()
{
    if (parsed)
    {
        throw new BuilderException("Each XMLConfigBuilder can only be used
once.");
    }
    parsed = true;
    //源码中没有这一句，只有
parseConfiguration(parser.evalNode("/configuration"));
    //为了让读者看得更明晰，源码拆分为以下两句
    XNode configurationNode = parser.evalNode("/configuration");
    parseConfiguration(configurationNode);
    return configuration;
}
/*
解析 "/configuration"节点下的子节点信息，然后将解析的结果设置到Configuration对象中
*/
private void parseConfiguration(XNode root) {
    try {
        //1.首先处理properties 节点
        propertiesElement(root.evalNode("properties")); //issue #117 read properties
first
        //2.处理typeAliases
        typeAliasesElement(root.evalNode("typeAliases"));
        //3.处理插件
        pluginElement(root.evalNode("plugins"));

        //4.处理objectFactory
        objectFactoryElement(root.evalNode("objectFactory"));
        //5.objectWrapperFactory
        objectWrapperFactoryElement(root.evalNode("objectWrapperFactory"));
        //6.settings
        settingsElement(root.evalNode("settings"));
        //7.处理environments
        environmentsElement(root.evalNode("environments")); // read it after
objectFactory and objectWrapperFactory issue #631
        //8.database
        databaseIdProviderElement(root.evalNode("databaseIdProvider"));
        //9. typeHandlers
        typeHandlerElement(root.evalNode("typeHandlers"));
        //10 mappers
        mapperElement(root.evalNode("mappers"));
    } catch (Exception e) {
        throw new BuilderException("Error parsing SQL Mapper Configuration. Cause: "
+ e, e);
    }
}

```

注意：在上述代码中，还有一个非常重要的地方，就是解析XML配置文件子节点<mappers>的方法

`mapperElements(root.evalNode("mappers"))`，它将解析我们配置的Mapper.xml配置文件，Mapper配置文件可以说是MyBatis的核心，MyBatis的特性和理念都体现在此Mapper的配置和设计上，我们将在后续的文章中讨论它，敬请期待～

3. 然后将这些值解析出来设置到Configuration对象中。

解析子节点的过程这里就不一一介绍了，用户可以参照MyBatis源码仔细揣摩，我们就看上述的`environmentsElement(root.evalNode("environments"))` 方法是如何将environments的信息解析出来，设置到 Configuration对象中的：

```
/*
 * 解析environments节点，并将结果设置到Configuration对象中
 * 注意：创建environment时，如果SqlSessionFactoryBuilder指定了特定的环境（即数据源）；
 * 则返回指定环境（数据源）的Environment对象，否则返回默认的Environment对象；
 * 这种方式实现了MyBatis可以连接多数据源
 */
```

```
private void environmentsElement(XNode context) throws Exception
{
    if (context != null)
    {
        if (environment == null)
        {
            environment = context.getStringAttribute("default");
        }
        for (XNode child : context.getChildren())
        {
            String id = child.getStringAttribute("id");
            if (isSpecifiedEnvironment(id))
            {
                //1. 创建事务工厂 TransactionFactory
                TransactionFactory txFactory =
transactionManagerElement(child.evalNode("transactionManager"));
                DataSourceFactory dsFactory =
dataSourceElement(child.evalNode("dataSource"));
                //2. 创建数据源DataSource
                DataSource dataSource = dsFactory.getDataSource();
            }
        }
    }
}
```

```
//3. 构造Environment对象
Environment.Builder environmentBuilder = new
Environment.Builder(id)
    .transactionFactory(txFactory)
    .dataSource(dataSource);
//4. 将创建的Environment对象设置到configuration 对象中
configuration.setEnvironment(environmentBuilder.build());
}

}

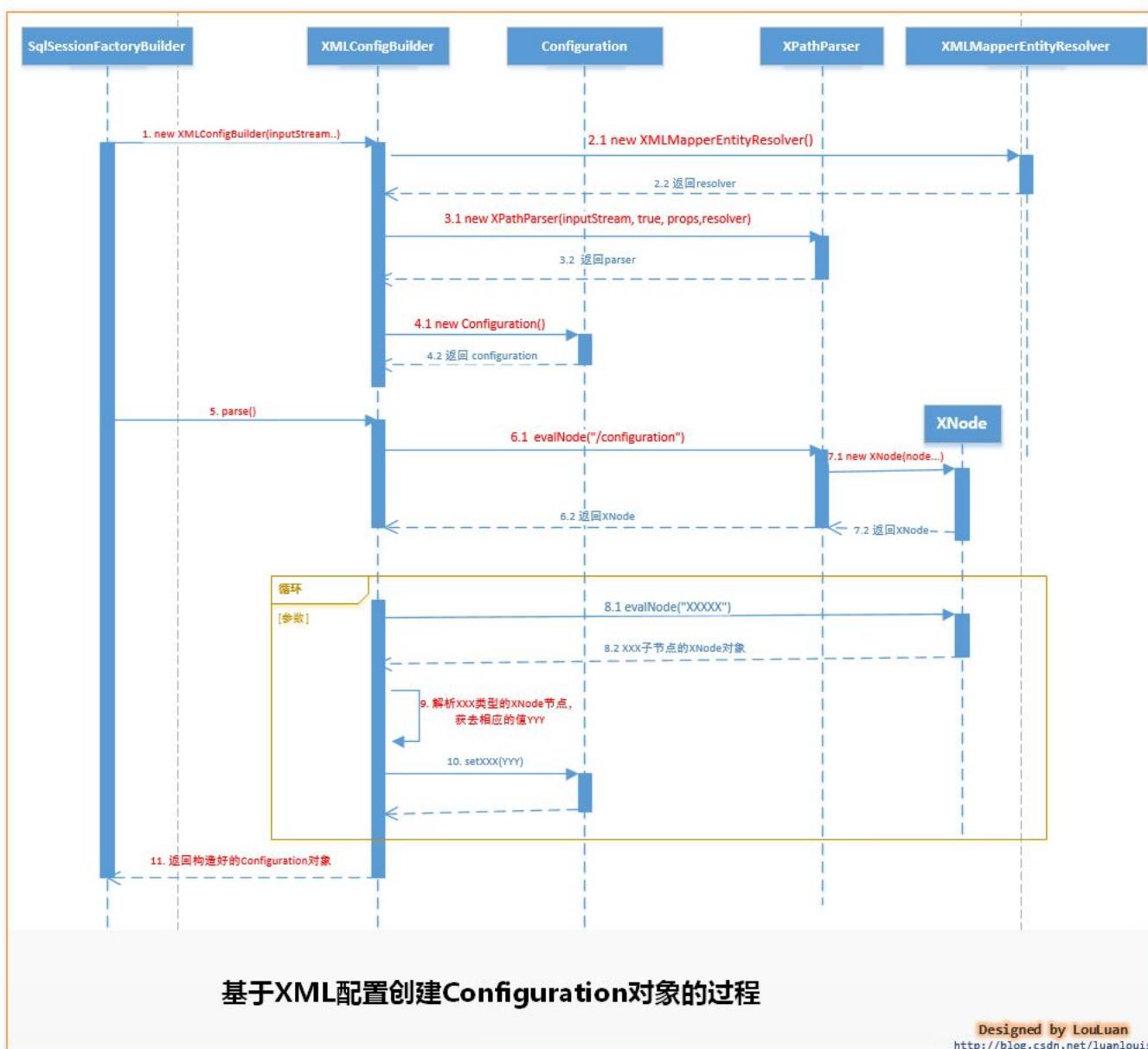
}

}

private boolean isSpecifiedEnvironment(String id)
{
    if (environment == null)
    {
        throw new BuilderException("No environment specified.");
    }
    else if (id == null)
    {
        throw new BuilderException("Environment requires an id attribute.");
    }
    else if (environment.equals(id))
    {
        return true;
    }
    return false;
}
```

#### 4. 返回Configuration对象

我们将上述的MyBatis初始化基本过程的序列图细化。



### 基于XML配置创建Configuration对象的过程

Designed by LouLuan  
http://blog.csdn.net/luanlouis

## 三、手动加载XML配置文件创建Configuration对象完成初始化, 创建并使用SqlSessionFactory对象

我们可以使用XMLConfigBuilder手动解析XML配置文件来创建Configuration对象, 代码如下:

```
String resource = "mybatis-config.xml";
InputStream inputStream = Resources.getResourceAsStream(resource);
//手动创建XMLConfigBuilder，并解析创建Configuration对象
XMLConfigBuilder parser = new XMLConfigBuilder(inputStream, null, null);
Configuration configuration=parse();
//使用Configuration对象创建SqlSessionFactory
SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(config
uration);
```

## 四、涉及到的设计模式

初始化的过程涉及到创建各种对象, 所以会使用一些创建型的设计模式。在初始化的过程中, Builder模式运用的比较多。

Builder模式应用1: SqlSessionFactory的创建

对于创建SqlSessionFactory时, 会根据情况提供不同的参数, 其参数组合可以有以下几种:

(Reader)

(Reader, String)

(Reader, Properties)

(Reader, String, Properties)

(InputStream)

(InputStream, String)

(InputStream, Properties)

(InputStream, String, Properties)

(Configuration)

Designed by LouLuan

<http://blog.csdn.net/luanlouis>

由于构造时参数不定，可以为其创建一个构造器Builder，将SqlSessionFactory的构建过程和表示分开：

C SqlSessionFactoryBuilder	
m build(Reader)	SqlSessionFactory
m build(Reader, String)	SqlSessionFactory
m build(Reader, Properties)	SqlSessionFactory
m build(Reader, String, Properties)	SqlSessionFactory
m build(InputStream)	SqlSessionFactory
m build(InputStream, String)	SqlSessionFactory
m build(InputStream, Properties)	SqlSessionFactory
m build(InputStream, String, Properties)	SqlSessionFactory
m build(Configuration)	SqlSessionFactory

《create》



Designed by LouLuan  
<http://blog.csdn.net/luanlouis>

MyBatis将SqlSessionFactoryBuilder和SqlSessionFactory相互独立。

在构建Configuration对象的过程中, XMLConfigParser解析 mybatis XML配置文件节点<environment>节点时, 会有以下相应的代码:

```
private void environmentsElement(XNode context) throws Exception {
    if (context != null) {
        if (environment == null) {
            environment = context.getStringAttribute("default");
        }
        for (XNode child : context.getChildren()) {
            String id = child.getStringAttribute("id");
            //是和默认的环境相同时, 解析之
            if (isSpecifiedEnvironment(id)) {
                TransactionFactory txFactory =
transactionManagerElement(child.evalNode("transactionManager"));
                DataSourceFactory dsFactory =
dataSourceElement(child.evalNode("dataSource"));
                DataSource dataSource = dsFactory.getDataSource();

                //使用了Environment内置的构造器Builder, 传递id 事务工厂和数据源
                Environment.Builder environmentBuilder = new Environment.Builder(id)
                    .transactionFactory(txFactory)
                    .dataSource(dataSource);
                configuration.setEnvironment(environmentBuilder.build());
            }
        }
    }
}
```

在Environment内部, 定义了静态内部Builder类:

```
public final class Environment {
    private final String id;
    private final TransactionFactory transactionFactory;
    private final DataSource dataSource;

    public Environment(String id, TransactionFactory transactionFactory, DataSource
dataSource) {
        if (id == null) {
            throw new IllegalArgumentException("Parameter 'id' must not be null");
        }
        if (transactionFactory == null) {
            throw new IllegalArgumentException("Parameter 'transactionFactory' must n
ot be null");
        }
        this.id = id;
        if (dataSource == null) {
            throw new IllegalArgumentException("Parameter 'dataSource' must not be null
");
        }
        this.transactionFactory = transactionFactory;
        this.dataSource = dataSource;
    }

    public static class Builder {
        private String id;
        private TransactionFactory transactionFactory;
        private DataSource dataSource;

        public Builder(String id) {
```

```
    this.id = id;
}

public Builder transactionFactory(TransactionFactory transactionFactory) {
    this.transactionFactory = transactionFactory;
    return this;
}

public Builder dataSource(DataSource dataSource) {
    this.dataSource = dataSource;
    return this;
}

public String id() {
    return this.id;
}

public Environment build() {
    return new Environment(this.id, this.transactionFactory, this.dataSource);
}

}

public String getId() {
    return this.id;
}

public TransactionFactory getTransactionFactory() {

    return this.transactionFactory;
}

public DataSource getDataSource() {
    return this.dataSource;
}
}
```

以上就是本文《深入理解mybatis原理》Mybatis初始化机制详解的全部内容，希望对大家有所帮助！上述内容如有不妥之处，还请读者指出，共同探讨，共同进步！

---

【END】

## 推荐阅读

1. [我采访了一位 Pornhub 工程师，聊了这些纯纯的话题](#)
2. [常见排序算法总结 - Java 实现](#)
3. [Java：如何更优雅的处理空值？](#)
4. [MySQL：数据库优化，可以看看这篇文章](#)
5. [团队开发中 Git 最佳实践](#)



喜欢文章, 点个在看 

声明：pdf仅供学习使用，一切版权归原创公众号所有；建议持续关注原创公众号获取最新文章，学习愉快！