# Mybatis

## 简介

Mybatis的强大在动态sql，

## Jar包

mysql-connector-java-5.1.25-bin.jar：数据库驱动包

:mybatis包

：

：动态代理包

：日志包

：log4j日志包

## 基本开发步骤

### 建表

|  |
| --- |
| CREATE TABLE `user` (  `id` int(10) unsigned NOT NULL AUTO\_INCREMENT,  `name` varchar(64) NOT NULL DEFAULT '',  `dept` varchar(254) NOT NULL DEFAULT '',  `website` varchar(254) DEFAULT '',  `phone` varchar(16) NOT NULL DEFAULT '',  PRIMARY KEY (`id`)  ) ENGINE=InnoDB AUTO\_INCREMENT=2 DEFAULT CHARSET=latin1;  -- ----------------------------  -- Records of user  -- ----------------------------  INSERT INTO `user` VALUES ('1', 'yiibai', 'Tech', 'http://www.yiibai.com', '13800009988'); |

### 主配置文件

Configure.xml

|  |
| --- |
| <?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>  <!—mybatis中configuration的子标签主要有：  configuration  |--- properties：  |--- settings  |--- typeAliases  |--- typeHandlers  |--- objectFactory  |--- plugins  |--- environments  |--- |--- environment  |--- |--- |--- transactionManager  |--- |--- |\_\_ dataSource  |\_\_ mappers  -->    <!—获取mysql.properties文件的属性值 -->  <properties resource="mysql.properties">  <property name="jdbc.driverClassName" value="com.mysql.jdbc.Driver"/>  <property name="jdbc.url" value="jdbc:mysql://localhost:3306/student\_manager"/>  <property name="username" value="root"/>  <property name="password" value="limingnihao"/>  </properties>  <!—这是MyBatis 修改操作运行过程细节的重要的步骤。下方这个表格描述了这些设置项、含义和默认值。 -->  <settings>  <!—对在此配置文件下的所有cache 进行全局性开/关设置，默认true -->  <setting name="cacheEnabled" value="true" />  <!—全局性设置懒加载。如果设为‘false’，则所有相关联的都会被初始化加载，默认true。 -->  <setting name="lazyLoadingEnabled" value="true" />  <!—当设置为‘true’的时候，懒加载的对象可能被任何懒属性全部加载。否则，每个属性都按需加载，默认true。-->  <setting name="aggressiveLazyLoading" value="true" />  <!—允许和不允许单条语句返回多个数据集（取决于驱动需求），默认true -->  <setting name="multipleResultSetsEnabled" value="true" />  <!—使用列标签代替列名称。不同的驱动器有不同的作法。参考一下驱动器文档，或者用这两个不同的选项进行测试一下。默认true-->  <setting name="useColumnLabel" value="true" />  <!—允许JDBC 生成主键。需要驱动器支持。如果设为了true，这个设置将强制使用被生成的主键，有一些驱动器不兼容不过仍然可以执行。默认false-->  <setting name="useGeneratedKeys" value="false" />  <!—指定MyBatis 是否并且如何来自动映射数据表字段与对象的属性。PARTIAL将只自动映射简单的，没有嵌套的结果。FULL 将自动映射所有复杂的结果。 默认PARTIAL-->  <setting name="autoMappingBehavior" value="false" />  <!—-->  <setting name="enhancementEnabled" value="false" />  <!—配置和设定执行器，SIMPLE 执行器执行其它语句。REUSE 执行器可能重复使用prepared statements 语句，BATCH执行器可以重复执行语句和批量更新。默认SIMPLE -->  <setting name="defaultExecutorType" value="SIMPLE" />  <!—设置一个时限，以决定让驱动器等待数据库回应的多长时间为超时。默认为null，正整数 -->  <setting name="defaultStatementTimeout" value="SIMPLE" />  </settings>  <!—类型别名是Java 类型的简称。它仅仅只是关联到XML 配置，简写冗长的JAVA 类名。这个别名alias对应user.xml中的resultType 。使用这个配置，“StudentEntity”就能在任何地方代替“com.manager.data.model.StudentEntity”被使用。对于普通的Java类型，有许多内建的类型别名。它们都是大小写不敏感的，由于重载的名字，要注意原生类型的特殊处理，参考11基本类型别名对照表。-->  <typeAliases>  <typeAlias alias="User" type="com.yiibai.mybatis.models.User" />  <typeAlias alias="StudentEntity" type="com.manager.data.model.StudentEntity" />  </typeAliases>    <!—无论是MyBatis在预处理语句中设置一个参数，还是从结果集中取出一个值时，类型处理器被用来将获取的值以合适的方式转换成Java类型。下面这个表格描述了默认的类型处理器，参考11.2默认的类型处理器  你可以重写类型处理器或创建你自己的类型处理器来处理不支持的或非标准的类型。要这样做的话，简单实现TypeHandler接口（org.mybatis.type），然后映射新的类型处理器类到Java类型，还有可选的一个JDBC类型。然后再typeHandlers中添加这个类型处理器。  新定义的类型处理器将会覆盖已经存在的处理Java的String类型属性和VARCHAR参数及结果的类型处理器。要注意MyBatis不会审视数据库元信息来决定使用哪种类型，所以你必须在参数和结果映射中指定那是VARCHAR类型的字段，来绑定到正确的类型处理器上。这是因为MyBatis直到语句被执行都不知道数据类型的这个现实导致的。。 -->  <typeHandlers>  <typeHandler javaType="String" jdbcType="VARCHAR" handler="liming.student.manager.type.Lim ingStringTypeHandler"/>  </typeHandlers>  <!—每次MyBatis 为结果对象创建一个新实例，都会用到ObjectFactory。默认的ObjectFactory 与使用目标类的构造函数创建一个实例毫无区别，如果有已经映射的参数，那也可能使用带参数的构造函数。  如果你重写ObjectFactory 的默认操作，你可以通过继承org.apache.ibatis.reflection.factory.DefaultObjectFactory创建一下你自己的。  ObjectFactory接口很简单。它包含两个创建用的方法，一个是处理默认构造方法的，另外一个是处理带参数构造方法的。最终，setProperties方法可以被用来配置ObjectFactory。在初始化你的ObjectFactory实例后，objectFactory元素体中定义的属性会被传递给setProperties方法。-->  <objectFactory type="liming.student.manager.configuration.LimingObjectFactory">  <property name="someProperty" value="100"/>  </objectFactory>  <!—MyBatis允许你在某一点拦截已映射语句执行的调用。默认情况下，MyBatis允许使用插件来拦截方法调用：Executor(update, query, flushStatements, commit, rollback, getTransaction, close, isClosed)  ParameterHandler(getParameterObject, setParameters)  ResultSetHandler(handleResultSets, handleOutputParameters)  StatementHandler(prepare, parameterize, batch, update, query) 这些类中方法的详情可以通过查看每个方法的签名来发现，而且它们的源代码在MyBatis的发行包中有。你应该理解你覆盖方法的行为，假设你所做的要比监视调用要多。如果你尝试修改或覆盖一个给定的方法，你可能会打破MyBatis的核心。这是低层次的类和方法，要谨慎使用插件。  使用插件是它们提供的非常简单的力量。简单实现拦截器接口，要确定你想拦截的指定签名。 -->  <!—建立sessionFactory用的数据 ，MyBatis 可以配置多个环境。这可以帮助你SQL 映射对应多种数据库等-->  <environments default="development">  <environment id="development">  <transactionManager type="JDBC" />  <dataSource type="POOLED">                  <property name="driver" value="com.mysql.jdbc.Driver" />  <property name="url" value="jdbc:mysql://127.0.0.1:3306/yiibai" />  <property name="username" value="root" />  <property name="password" value="123456" />  </dataSource>  </environment>  </environments>  <mappers>  <!—包含要映射的xml文件，这里是告诉MyBatis 去哪寻找映射SQL 的语句。可以使用类路径中的资源引用，或者使用字符，输入确切的URL 引用。-->  <mapper resource="com/yiibai/mybatis/models/User.xml" />  </mappers>  </configuration> |

### 实体类和实体类映射文件

|  |
| --- |
| package com.yiibai.mybatis.models;  public class User {  private int id;  private String name;  private String dept;  private String phone;  private String website;  public String getWebsite() {  return website;  }  public void setWebsite(String website) {  this.website = website;  }  public int getId() {  return id;  }  public void setId(int id) {  this.id = id;  }  public String getName() {  return name;  }  public void setName(String name) {  this.name = name;  }  public String getDept() {  return dept;  }  public void setDept(String dept) {  this.dept = dept;  }  public String getPhone() {  return phone;  }  public void setPhone(String phone) {  this.phone = phone;  }  } |

User.xml

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"  "http://mybatis.org/dtd/mybatis-3-mapper.dtd">  <!—定义各种sql语句，以及这些语句的参数，返回值类型等  当Java接口与XML文件在一个相对路径下时，可以不在myBatis配置文件的mappers中声明。  SQL 映射XML 文件一些初级的元素：  mapper  |-- **cache** – 配置给定模式的缓存。加入</cache>开启二级缓存，好吃如下  1. 所有在映射文件里的select 语句都将被缓存。  2. 所有在映射文件里insert,update 和delete 语句会清空缓存。  3. 缓存使用“最近很少使用”算法来回收  4. 缓存不会被设定的时间所清空。  5. 每个缓存可以存储1024 个列表或对象的引用（不管查询出来的结果是什么）。  6. 缓存将作为“读/写”缓存，意味着获取的对象不是共享的且对调用者是安全的。不会有其它的调用者或线程潜在修改。  |--|-- eviction： 缓存策略：  LRU - 最近最少使用法：移出最近较长周期内都没有被使用的对象。  FIFI- 先进先出：移出队列里较早的对象  SOFT - 软引用：基于软引用规则，使用垃圾回收机制来移出对象  WEAK - 弱引用：基于弱引用规则，使用垃圾回收机制来强制性地移出对象  默认 LRU  |--|-- flushInterval：代表一个合理的毫秒总计时间。默认是不设置，因此使用无间隔清空即只能调用语句来清空。  |--|-- size：缓存的对象的大小  |--|-- readOnly：只读缓存将对所有调用者返回同一个实例。因此都不能被修改，这可以极大的提高性能。可写的缓存将通过序列化来返回一个缓存对象的拷贝。这会比较慢，但是比较安全。所以默认值是false。  |-- **cache-ref** – 从别的模式中引用一个缓存  |-- **resultMap** – 这是最复杂而却强大的一个元素了，它描述如何从结果集中加载对象  |--|-- constructor：查询结果与构造器参数对应  |--|--|-- idArg：id参数，将结果标记为id，以便全局使用  |--|--|-- arg：普通参数  |--|-- id：ID 结果，将结果集标记为ID，以方便全局调用  |--|-- result：反射到JavaBean 属性的普通结果  |--|-- association：一对多，①在resultmap中对应sql语句②在resultMap中对应resultMap  |--|-- collection：多对一，①在resultmap中对应sql语句②在resultMap中对应resultMap  |--|-- discriminator：鉴别器，有时一个单独的数据库查询也许返回很多不同（但是希望有些关联）数据类型的结果集, 鉴别器元素就是被设计来处理这个情况的。相当于根据值确定类型，相当于switch。  |--|--|-- case:相当于switch中的case  |--|--|--|-- result：相当于switch中的执行语句。  |-- **sql** – 一个可以被其他语句复用的SQL 块  |-- **insert** – 映射INSERT 语句  |--|-- id：唯一表示，可以被其他语句引用  |--|-- parameterType：参数类型，完整类名或别名  |--|-- flushCache：true：每次调用都清空缓存，select默认false  |--|-- useCache：true：语句结果集将被缓存，select默认false  |--|-- timeout：驱动器在抛出异常前的等待时间，默认由驱动器自己决定  |--|-- fetchSize：结果集能容纳的最大记录数，到达最大值就返回，默认驱动器自己决定  |--|-- statementType：使用的statement类型：statement，preparedstatement，callablestatement，默认preparedstatement  |--|-- useGeneratedKeys： 告诉MyBatis 使用JDBC 的getGeneratedKeys 方法来获取数据库自己生成的主键（MySQL、SQLSERVER 等关系型数据库会有自动生成的字段）。默认：false。  |--|-- keyProperty：调用数据库的方法，获取返回值，设置到具体属性上，例：<selectKey keyProperty="studentId" resultType="String" order="BEFORE">select nextval('student')</selectKey> 将select nextval('student')返回的值设置到studentId上  |--|--|-- keyProperty：selectKey 语句生成结果需要设置的属性。  |--|--|-- resultType：生成结果类型，MyBatis 允许使用基本的数据类型，包括String 、int类型。  |--|--|-- order： 1：BEFORE，会先选择主键，然后设置keyProperty，再执行insert语句；  2：AFTER，就先运行insert 语句再运行selectKey 语句。  |--|--|-- statementType：MyBatis 支持STATEMENT，PREPARED和CALLABLE 的语句形式， 对应Statement ，PreparedStatement 和CallableStatement 响应  |-- **update** – 映射UPDATE 语句  |--|-- 子参数参考delete子参数  |-- **delete** – 映射DELEETE 语句  |--|-- id：唯一表示，可以被其他语句引用  |--|-- parameterType：参数类型，完整类名或别名  |--|-- flushCache：true：每次调用都清空缓存，select默认false  |--|-- useCache：true：语句结果集将被缓存，select默认false  |--|-- timeout：驱动器在抛出异常前的等待时间，默认由驱动器自己决定  |--|-- fetchSize：结果集能容纳的最大记录数，到达最大值就返回，默认驱动器自己决定  |--|-- statementType：使用的statement类型：statement，preparedstatement，callablestatement，默认preparedstatement  |-- **select** - 映射SELECT语句  |--|-- id：唯一表示，可以被其他语句引用  |--|-- parameterType：参数类型，完整类名或别名  |--|-- resultType：返回值类型，完整类名或别名，若是集合，则填写集合元素的类型，不能同resultMap并用。  |--|-- resultMap：resultMap的引用，可以是本文件，也可以其他文件，不能同resultMap并用。  |--|-- flushCache：true：每次调用都清空缓存，select默认false  |--|-- useCache：true：语句结果集将被缓存，select默认false  |--|-- timeout：驱动器在抛出异常前的等待时间，默认由驱动器自己决定  |--|-- fetchSize：结果集能容纳的最大记录数，到达最大值就返回，默认驱动器自己决定  |--|-- statementType：使用的statement类型：statement，preparedstatement，callablestatement，默认preparedstatement  |--|-- resultSetType：只转发orward\_only，滚动敏感scroll\_sensitive，不区分大小写的滚动scroll\_insensitive，由驱动器决定默认值。  传参问题：   1. 一个参数可以用基本类型，实体类，map等，用parameterType标识什么类型就行，用#{}获取 2. 多个类型可以用@para注解来传值，#{}获取 3. #家‘’，$不加‘’；   -->  <mapper namespace="com.yiibai.mybatis.models.UserMapper">  <select id="GetUserByID" parameterType="int" resultType="User">  select \* from `user` where id = #{id}  </select>  <select id="getUser" parameterType="int"  resultType="com.yiibai.mybatis.models.User">  SELECT \*  FROM USER  WHERE id = #{userId}  </select>  <insert id="insertUser" parameterType="User">  INSERT INTO USER(name,  dept, website,phone)  VALUES(#{name}, #{dept}, #{website}, #{phone})  </insert>  <select id="getUserList" resultType="com.yiibai.mybatis.models.User">  SELECT \* FROM USER  </select>  <update id="updateUser" parameterType="User">  UPDATE USER  SET  name=  #{name},  dept = #{dept},  website = #{website},  phone = #{phone}  WHERE  id =  #{id}  </update>  <delete id="deleteUser" parameterType="int">  DELETE FROM USER WHERE id = #{id}  </delete>  </mapper> |
|  |

### 接口代替user.xml

|  |
| --- |
| package org.mybatis.example;  public interface BlogMapper {  @Select("SELECT \* FROM blog WHERE id = #{id}")  Blog selectBlog(int id);  } |

## API

### 获得sessionfactory

|  |
| --- |
| //通过sqlsessionFactoryBuilder.build(reader)获取  reader = Resources.getResourceAsReader("config/Configure.xml");  sqlSessionFactory = new SqlSessionFactoryBuilder().build(reader);  //不使用xml配置文件获得  DataSource dataSource = BlogDataSourceFactory.getBlogDataSource();  TransactionFactory transactionFactory = new JdbcTransactionFactory();  Environment environment = new Environment("development", transactionFactory, dataSource);  Configuration configuration = new Configuration(environment);  configuration.addMapper(BlogMapper.class);  SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(configuration); |

### 获得session

|  |
| --- |
| SqlSession session = sqlSessionFactory.openSession() |

### 查询

|  |
| --- |
| Object selectOne(String statement, Object parameter)  List selectList(String statement, Object parameter)  int insert(String statement, Object parameter)  int update(String statement, Object parameter)  int delete(String statement, Object parameter)  Object selectOne(String statement)  List selectList(String statement)  int insert(String statement)  int update(String statement)  int delete(String statement)  List selectList(String statement, Object parameter, RowBounds rowBounds)  void select(String statement, Object parameter, ResultHandler handler)  void select(String statement, Object parameter, RowBounds rowBounds,ResultHandler handler) |

### 插入更新删除

|  |
| --- |
| session.insert(…);  session.update(…);  session.delete(…); |

### 其他

|  |
| --- |
| void commit()  void commit(boolean force)  void rollback()  void rollback(boolean force)  void clearCache()  session.close(); |

## 注解

@CacheNamespace(size = 512) ： 定义在该命名空间内允许使用内置缓存，最大值为512个对象引用，读写默认是开启的，缓存内省刷新时间为默认3600000毫秒，写策略是拷贝整个对象镜像到全新堆（如同CopyOnWriteList）因此线程安全。

@SelectProvider(type = TestSqlProvider.class, method = "getSql") ： 提供查询的SQL语句，如果你不用这个注解，你也可以直接使用@Select("select \* from ....")注解，把查询SQL抽取到一个类里面，方便管理，同时复杂的SQL也容易操作，type = TestSqlProvider.class就是存放SQL语句的类，而method = "getSql"表示get接口方法需要到TestSqlProvider类的getSql方法中获取SQL语句。

@Options(useCache = true, flushCache = false, timeout = 10000) ： 一些查询的选项开关，比如useCache = true表示本次查询结果被缓存以提高下次查询速度，flushCache = false表示下次查询时不刷新缓存，timeout = 10000表示查询结果缓存10000秒。

@Results(value = { @Result(id = true, property = "id", column = "test\_id", javaType = String.class, jdbcType = JdbcType.VARCHAR),

@Result(property = "testText", column = "test\_text", javaType = String.class, jdbcType = JdbcType.VARCHAR) }) ： 表示sql查询返回的结果集，@Results是以@Result为元素的数组，@Result表示单条属性-字段的映射关系，如：@Result(id = true, property = "id", column = "test\_id", javaType = String.class, jdbcType = JdbcType.VARCHAR)可以简写为：@Result(id = true, property = "id", column = "test\_id")，id = true表示这个test\_id字段是个PK，查询时mybatis会给予必要的优化，应该说数组中所有的@Result组成了单个记录的映射关系，而@Results则单个记录的集合。另外还有一个非常重要的注解@ResultMap也和@Results差不多，到时会讲到。

@Param("id") ：全局限定别名，定义查询参数在sql语句中的位置不再是顺序下标0,1,2,3....的形式，而是对应名称，该名称就在这里定义。

@ResultMap(value = "getByTestText") ：重要的注解，可以解决复杂的映射关系，包括resultMap嵌套，鉴别器discriminator等等。注意一旦你启用该注解，你将不得不在你的映射文件中配置你的resultMap，而value = "getByTestText"即为映射文件中的resultMap ID(注意此处的value = "getByTestText"，必须是在映射文件中指定命名空间路径)。@ResultMap在某些简单场合可以用@Results代替，但是复杂查询，比如联合、嵌套查询@ResultMap就会显得解耦方便更容易管理。

@InsertProvider(type = TestSqlProvider.class, method = "insertSql") ：用法和含义@SelectProvider一样，只不过是用来插入数据库而用的。

@Options(flushCache = true, timeout = 20000) ：对于需要更新数据库的操作，需要重新刷新缓存flushCache = true使缓存同步。

@UpdateProvider(type = TestSqlProvider.class, method = "updateSql") ：用法和含义@SelectProvider一样，只不过是用来更新数据库而用的。

@Param("testBean") ：是一个自定义的对象，指定了sql语句中的表现形式，如果要在sql中引用对象里面的属性，只要使用testBean.id，testBean.textText即可，mybatis会通过反射找到这些属性值。

@DeleteProvider(type = TestSqlProvider.class, method = "deleteSql") ：用法和含义@SelectProvider一样，只不过是用来删除数据而用的。

BEGIN();表示刷新本地线程，某些变量为了线程安全，会先在本地存放变量，此处需要刷新。

SELECT，FROM，WHERE等等都是sqlbuilder定义的公用静态方法，用来组成你的sql字符串。如果你在testMapper中调用该方法的某个接口方法已经定义了参数@Param()，那么该方法的参数Map<String, Object> parameters即组装了@Param()定义的参数，比如testMapper接口方法中定义参数为@Param("testId"),@Param("testText")，那么parameters的形态就是：[key="testId",value=object1],[key="testText",value=object2]，如果接口方法没有定义@Param()，那么parameters的key就是参数的顺序小标：[key=0,value=object1],[key=1,value=object2]，SQL()将返回最终append结束的字符串，sql语句中的形如

#{id,javaType=string,jdbcType=VARCHAR}完全可简写为#{id}，我只是为了规整如此写而已。另外，对于复杂查询还有很多标签可用，比如：JOIN，INNER\_JOIN，GROUP\_BY，ORDER\_BY等等，具体使用详情，你可以查看源码。

## 动态sql

1. selectKey
2. if：根据test值判断
3. where if：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。
4. set if：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。
5. if + trim代替where/set标签，  
   prefix：前缀覆盖并增加其内容  
   suffix：后缀覆盖并增加其内容  
   prefixOverrides：若前缀为value，则去掉  
   suffixOverrides：若后缀为value，则去掉
6. choose when：相当于java的switch语句，choose=switch，when=case
7. foreach：循环，collection：集合属性，item：接收集合每个变量，open：前缀，separator，分隔符，close：后缀。一般用在in条件中。

## 映射器Mapper

有一个映射文件：

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN" "http://mybatis.org/dtd/mybatis-3-mapper.dtd">  <mapper namespace="com.happy.data.mappers.QCallMapper">  <resultMap type="com.happy.data.pojo.MerQueue" id="QTakeResultMap">  <id property="\_id" column="\_id"/>  <result property="merchantId" column="merchant\_id"/>  <result property="cusPhone" column="cus\_phone"/>  <result property="queuePhone" column="queue\_phone"/>  <result property="queueNo" column="queue\_no"/>  <result property="queueType" column="queue\_type"/>  <result property="takeTime" column="take\_time"/>  <result property="eatType" column="eat\_type"/>  <result property="queuePeople" column="queue\_people"/>  <result property="tag" column="tag"/>  <result property="today" column="today"/>  <result property="breakTimes" column="break\_times"/>  <result property="isOut" column="is\_out"/>  <result property="remark1" column="remark1"/>  <result property="remark2" column="remark2"/>  </resultMap>    <!--查询大中小桌的队列信息 add by lxx 20130219-->  <select id="getQuInfo" resultMap="QTakeResultMap">  SELECT \*  from t\_merchant\_queue tmq  where tmq.merchant\_id=#{merchantId}  and tmq.eat\_type=#{eatType}  AND YEAR(NOW())=YEAR(tmq.today)  AND MONTH(NOW())=MONTH(tmq.today)  AND DAY(NOW()) = DAY(tmq.today)  </select>  <insert id="insertAuthor" parameterType="domain.blog.Author">  insert into Author (id,username,password,email,bio)  values (#{id},#{username},#{password},#{email},#{bio})  </insert>  <update id="updateAuthor" parameterType="domain.blog.Author">  update Author set  username = #{username},  password = #{password},  email = #{email},  bio = #{bio}  where id = #{id}  </update>  <delete id="deleteAuthor” parameterType="int">  delete from Author where id = #{id}  </delete>  </mapper> |

它对应的映射器mappers如下：绑定映射语句的接口

|  |
| --- |
| Packge com.happy.data.mappers;  public interface QCallMapper{  public List<MerQueue> getQuInfo(@Param("merchantId")String merchantId,@Param("eatType")int eatType );  } |

其中：

1. 接口全名与namespace一致
2. 方法名与sql的id一致
3. 在命名空间com.happy.data.mappers.QCallMapper中，定义了一名为“getQuInfo”的映射语句。这样就允许你使用完全限定名com.happy.data.mappers.QCallMapper.getQuInfo来调用映射语句。
4. 注意参数注释：  
   merchantId  
   这就告诉MyBatis创建一个PreparedStatement（预处理语句）参数
5. JdbcType  
   在这个表格之后的所支持的JDBC类型列表中的类型。JDBC类型是仅仅需要对插入，更新和删除操作可能为空的列进行处理。这是JDBC的需要，而不是MyBatis的。如果你直接使用JDBC编程，你需要指定这个类型-但仅仅对可能为空的值。

## 对应多个数据库

配置多个environment，建立多个SqlSessionFactory

为了指定创建哪种环境，只要将它作为可选的参数传递给 SqlSessionFactoryBuilder 即可。可以接受环境配置的两个方法签名是：

SqlSessionFactory factory = new SqlSessionFactoryBuilder().build(reader, environment);

SqlSessionFactory factory = new SqlSessionFactoryBuilder().build(reader, environment,properties);

如果忽略了环境参数，那么默认环境将会被加载，如下所示：

SqlSessionFactory factory = new SqlSessionFactoryBuilder().build(reader);

SqlSessionFactory factory = new SqlSessionFactoryBuilder().build(reader,properties);

注意这里的关键点:

默认的环境 ID（比如:default=”development”）。

每个 environment 元素定义的环境 ID（比如:id=”development”）。

事务管理器的配置（比如:type=”JDBC”）。

数据源的配置（比如:type=”POOLED”）。

默认的环境和环境 ID 是一目了然的。随你怎么命名，只要保证默认环境要匹配其中一个环境ID。

## 与Spring的整合

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <beans xmlns="http://www.springframework.org/schema/beans" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:tx="http://www.springframework.org/schema/tx" xmlns:aop="http://www.springframework.org/schema/aop" xsi:schemaLocation="  http://www.springframework.org/schema/beans  http://www.springframework.org/schema/beans/spring-beans-3.0.xsd  http://www.springframework.org/schema/tx  http://www.springframework.org/schema/tx/spring-tx-3.0.xsd  http://www.springframework.org/schema/aop  http://www.springframework.org/schema/aop/spring-aop-3.0.xsd  ">  <!-- JNDI方式配置数据源 -->  <!-- <bean id="dataSource" class="org.springframework.jndi.JndiObjectFactoryBean"> <property name="jndiName" value="${jndiName}"></property> </bean> -->  <!-- ========================================配置数据源========================================= -->  <!-- 配置数据源，使用的是alibaba的Druid(德鲁伊)数据源 -->  <bean name="dataSource" class="com.alibaba.druid.pool.DruidDataSource" init-method="init" destroy-method="close">  <property name="url" value="${jdbc\_url}" />  <property name="username" value="${jdbc\_username}" />  <property name="password" value="${jdbc\_password}" />  <!-- 初始化连接大小 -->  <property name="initialSize" value="0" />  <!-- 连接池最大使用连接数量 -->  <property name="maxActive" value="20" />  <!-- 连接池最大空闲 -->  <property name="maxIdle" value="20" />  <!-- 连接池最小空闲 -->  <property name="minIdle" value="0" />  <!-- 获取连接最大等待时间 -->  <property name="maxWait" value="60000" />  <!--  <property name="poolPreparedStatements" value="true" />  <property name="maxPoolPreparedStatementPerConnectionSize" value="33" />  -->  <property name="validationQuery" value="${validationQuery}" />  <property name="testOnBorrow" value="false" />  <property name="testOnReturn" value="false" />  <property name="testWhileIdle" value="true" />  <!-- 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 -->  <property name="timeBetweenEvictionRunsMillis" value="60000" />  <!-- 配置一个连接在池中最小生存的时间，单位是毫秒 -->  <property name="minEvictableIdleTimeMillis" value="25200000" />  <!-- 打开removeAbandoned功能 -->  <property name="removeAbandoned" value="true" />  <!-- 1800秒，也就是30分钟 -->  <property name="removeAbandonedTimeout" value="1800" />  <!-- 关闭abanded连接时输出错误日志 -->  <property name="logAbandoned" value="true" />  <!-- 监控数据库 -->  <!-- <property name="filters" value="stat" /> -->  <property name="filters" value="mergeStat" />  </bean>    <!-- ========================================分隔线========================================= -->    <!-- ========================================针对myBatis的配置项============================== -->  <!-- 配置sqlSessionFactory  DataSource：SqlSessionFactory 有一个单独的必须属性,就是 JDBC 的 DataSource。这可以是任意 的 DataSource,其配置应该和其它 Spring 数据库连接是一样的。  configLocation：一个通用的属性是 configLocation,它是用来指定 MyBatis 的 configure.XML 配置文件路径的。 如果基本的 MyBatis 配置需要改变, 那么这就是一个需要它的地方。 通常这会是<settings> 或<typeAliases>的部分。  要注意这个配置文件不需要是一个完整的 MyBatis 配置。确切地说, configure.XML中的任意环境,数据源 和 MyBatis 的事务管理器都会被忽略。SqlSessionFactoryBean 会创建它自己的,虽然定制 MyBatis 的 Environment 时需要这些值。  mapperLocations：如果 MyBatis 映射器 configure.XML文件在和映射器类相同的路径下不存在,那么另外一个需要配置文件的原因就是它了。需要配置sqlmapping.xml，有两种选择。第一是手动在 MyBatis 的 XML 配 置文件中使用<mappers>部分来指定类路径。第二是使用工厂 bean 的 mapperLocations 属 性。  mapperLocations 属性使用一个资源位置的 list。 这个属性可以用来指定 MyBatis 的 XML 映射器文件的位置。 它的值可以包含 Ant 样式来加载一个目录中所有文件, 或者从基路径下 递归搜索所有路径。  -->  <bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">  <!-- 实例化sqlSessionFactory时需要使用上述配置好的数据源以及SQL映射文件，替代Configuration.xml-->  <property name="dataSource" ref="dataSource" />  <!-- 自动扫描me/gacl/mapping/目录下的所有SQL映射的xml文件, 省掉Configuration.xml里的手工配置，因为有 了DataSource，不在需要environment，所以没mybatis的Configuration.xml文件，所以在这里引入sqlmapping文件value="classpath:me/gacl/mapping/\*.xml"指的是classpath(类路径)下me.gacl.mapping包中的所有xml文件  UserMapper.xml位于me.gacl.mapping包下，这样UserMapper.xml就可以被自动扫描 -->  <property name="mapperLocations" value="classpath:me/gacl/mapping/\*.xml" />  </bean>  <!-- 配置扫描器 -->  <bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">  <!-- 扫描me.gacl.dao这个包以及它的子包下的所有映射接口类-->  <property name="basePackage" value="me.gacl.dao" />  <property name="sqlSessionFactoryBeanName" value="sqlSessionFactory" />  </bean>    <!-- ========================================分隔线========================================= -->  <!-- 配置Spring的事务管理器 -->  <bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">  <property name="dataSource" ref="dataSource" />  </bean>  <!-- 注解方式配置事物 -->  <!-- <tx:annotation-driven transaction-manager="transactionManager" /> -->  <!-- 拦截器方式配置事物 -->  <tx:advice id="transactionAdvice" transaction-manager="transactionManager">  <tx:attributes>  <tx:method name="add\*" propagation="REQUIRED" />  <tx:method name="append\*" propagation="REQUIRED" />  <tx:method name="insert\*" propagation="REQUIRED" />  <tx:method name="save\*" propagation="REQUIRED" />  <tx:method name="update\*" propagation="REQUIRED" />  <tx:method name="modify\*" propagation="REQUIRED" />  <tx:method name="edit\*" propagation="REQUIRED" />  <tx:method name="delete\*" propagation="REQUIRED" />  <tx:method name="remove\*" propagation="REQUIRED" />  <tx:method name="repair" propagation="REQUIRED" />  <tx:method name="delAndRepair" propagation="REQUIRED" />  <tx:method name="get\*" propagation="SUPPORTS" />  <tx:method name="find\*" propagation="SUPPORTS" />  <tx:method name="load\*" propagation="SUPPORTS" />  <tx:method name="search\*" propagation="SUPPORTS" />  <tx:method name="datagrid\*" propagation="SUPPORTS" />  <tx:method name="\*" propagation="SUPPORTS" />  </tx:attributes>  </tx:advice>  <aop:config>  <aop:pointcut id="transactionPointcut" expression="execution(\* me.gacl.service..\*Impl.\*(..))" />  <aop:advisor pointcut-ref="transactionPointcut" advice-ref="transactionAdvice" />  </aop:config>  <!-- 配置druid监控spring jdbc -->  <bean id="druid-stat-interceptor" class="com.alibaba.druid.support.spring.stat.DruidStatInterceptor">  </bean>  <bean id="druid-stat-pointcut" class="org.springframework.aop.support.JdkRegexpMethodPointcut" scope="prototype">  <property name="patterns">  <list>  <value>me.gacl.service.\*</value>  </list>  </property>  </bean>  <aop:config>  <aop:advisor advice-ref="druid-stat-interceptor" pointcut-ref="druid-stat-pointcut" />  </aop:config>  </beans> |

## 参考文档

### 参考url

主配置文件详解：<http://limingnihao.iteye.com/blog/1060764>

Sql语句映射文件：<http://limingnihao.iteye.com/blog/781878>

动态sql：<http://limingnihao.iteye.com/blog/782190>

与Spring整合：<http://www.cnblogs.com/wangmingshun/p/5674633.html>

三种Spring整合：<http://www.cnblogs.com/xdp-gacl/p/4271627.html>

注解：<http://www.cnblogs.com/ibook360/archive/2012/07/16/2594056.html>

### 基本类型别名对照表

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | | 别名 |  | 映射的类型 | | \_byte |  | byte | | \_long |  | long | | \_short |  | short | | \_int |  | int | | \_integer |  | int | | \_double |  | double | | \_float |  | float | | \_boolean |  | boolean | | string |  | String | | byte |  | Byte | | long |  | Long | | short |  | Short | | int |  | Integer | | integer |  | Integer | | double |  | Double | | float |  | Float | | boolean |  | Boolean | | date |  | Date | | decimal |  | BigDecimal | | bigdecimal |  | BigDecimal | | object |  | Object | | map |  | Map | | hashmap |  | HashMap | | list |  | List | | arraylist |  | ArrayList | | collection |  | Collection | | iterator |  | Iterator | |

### 默认的类型处理器。

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | | 类型处理器 | Java类型 | JDBC类型 | | BooleanTypeHandler | Boolean，boolean | 任何兼容的布尔值 | | ByteTypeHandler | Byte，byte | 任何兼容的数字或字节类型 | | ShortTypeHandler | Short，short | 任何兼容的数字或短整型 | | IntegerTypeHandler | Integer，int | 任何兼容的数字和整型 | | LongTypeHandler | Long，long | 任何兼容的数字或长整型 | | FloatTypeHandler | Float，float | 任何兼容的数字或单精度浮点型 | | DoubleTypeHandler | Double，double | 任何兼容的数字或双精度浮点型 | | BigDecimalTypeHandler | BigDecimal | 任何兼容的数字或十进制小数类型 | | StringTypeHandler | String | CHAR和VARCHAR类型 | | ClobTypeHandler | String | CLOB和LONGVARCHAR类型 | | NStringTypeHandler | String | NVARCHAR和NCHAR类型 | | NClobTypeHandler | String | NCLOB类型 | | ByteArrayTypeHandler | byte[] | 任何兼容的字节流类型 | | BlobTypeHandler | byte[] | BLOB和LONGVARBINARY类型 | | DateTypeHandler | Date（java.util） | TIMESTAMP类型 | | DateOnlyTypeHandler | Date（java.util） | DATE类型 | | TimeOnlyTypeHandler | Date（java.util） | TIME类型 | | SqlTimestampTypeHandler | Timestamp（java.sql） | TIMESTAMP类型 | | SqlDateTypeHandler | Date（java.sql） | DATE类型 | | SqlTimeTypeHandler | Time（java.sql） | TIME类型 | | ObjectTypeHandler | Any | 其他或未指定类型 | | EnumTypeHandler | Enumeration类型 | VARCHAR-任何兼容的字符串类型，作为代码存储（而不是索引）。 | |