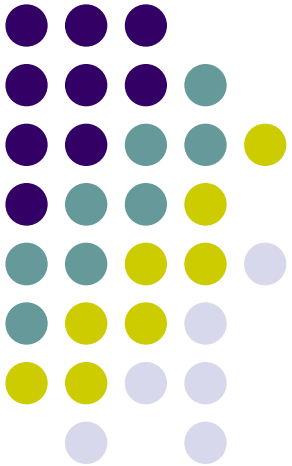
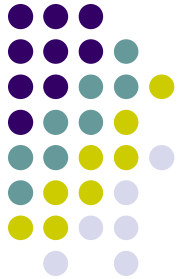
Mybatis技术简介

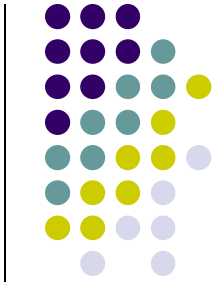
dezhaos@gmail.com





持久层

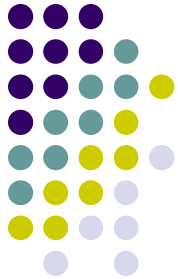
- 所谓“持久层”，是指在系统逻辑层面上，专注于实现数据持久化的一个相对独立的领域（Domain）
- 一个完善的持久化层应该能达到以下目标：
  - 代码的可重用性高，能完成所有数据库的访问操作
  - 可以支持多种数据库平台
  - 具有相对独立性，当持久层的实现发生变化时，不影响上层的实现



持久化

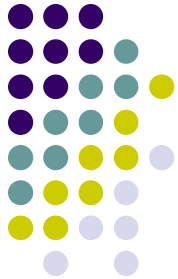
- 狭义的理解，“持久化”指将域对象永久保存至数据库中
- 广义的理解，“持久化”包括与数据库相关的各种操作
  - 保存：将域对象永久保存至数据库中
  - 更新：更新数据库中域对象的状态
  - 删除：从数据库中删除一个域对象
  - 加载：根据特定的OID，将一个域对象从数据库加载至内存
  - 查询：根据特定的查询条件，将符合查询条件的一个或多个域对象从数据库加载至内存

JDBC开发中存在的问题



- 通过编写JDBC代码，发现JDBC操作非常繁复。
  - 1.定义数据库连接参数
  - 2.打开数据库连接
  - 3.声明SQL语句
  - 4.预编译并执行SQL语句
  - 5.遍历查询结果（如果需要的话），对每一记录行进行处理
  - 6.处理事务
  - 7.关闭数据库连接
- 以上步骤每次除了3和5步骤，其他全部是重复工作。
- 如何想办法将这些重复的工作抽象出去，简化JDBC的开发工作呢？这是值得每一个开发人员思考的问题，因为软件开发者是善于分析的思想家以及问题的解决者。

MyBatis简介



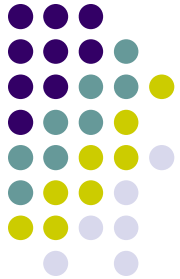
- MyBatis是支持普通SQL查询，存储过程和高级映射的优秀持久层框架。Mybatis消除了几乎所有的JDBC代码和参数的手工设置以及结果集的检索。Mybatis使用简单的XML或注解用于配置和原始映射，将接口和java的POJOs映射成数据库的记录。

  MyBatis作为持久层框架，其**主要思想**是将程序中的大量sql语句剥离出来，配置在配置文件中，实现sql的灵活配置。这样做的好处是将sql与程序代码分离，可以在不修改程序代码的情况下，直接在配置文件中修改sql。

MyBatis的前世今生

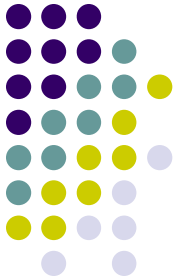
- MyBatis的前身就是iBatis,iBatis本是由Clinton Begin开发，后来捐给Apache基金会，成立了iBatis开源项目。2010年5月该项目由Apahce基金会迁移到了Google Code，并且改名为MyBatis。

MyBatis介绍



- MyBatis是一个数据持久层(ORM)框架。把实体类和SQL语句之间建立了映射关系，是一种半自动化的ORM实现。

- MyBatis的优点：
  1.基于SQL语法，简单易学。
  2.能了解底层组装过程。
  3.SQL语句封装在配置文件中，便于统一管理与维护，降低了程序的耦合度。
  4.程序调试方便。



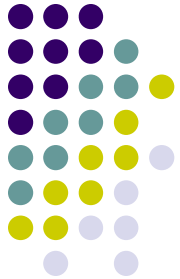
与传统JDBC的比较

- 减少了61%的代码量
- 最简单的持久化框架
- 架构级性能增强
- SQL代码从程序代码中彻底分离，可重用
- 增强了项目中的分工
- 增强了移植性

JDBC 与 MyBatis直观对比

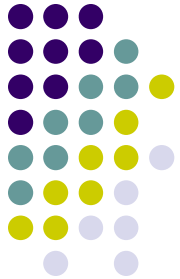
```
1    Class.forName("oracle.jdbc.driver.OracleDriver");
2    Connection conn= DriverManager.getConnection(url,user,password);
3    java.sql.PreparedStatement  st = conn.prepareStatement(sql);
4    st.setInt(0,1);
5    st.execute();
6    java.sql.ResultSet rs =  st.getResultSet();
7    while(rs.next()){
8        String result = rs.getString(colname);
9    }
10 <mapper namespace="org.mybatis.example.BlogMapper">
11         <select id="selectBlog" parameterType="int" resultType="Blog">
12                 select * from Blog where id = #{id}
13         </select>
14 </mapper>
15
```

- MyBatis 就是将上面这几行代码分解包装：
  - 前两行是对数据库的数据源的管理包括事务管理，
  - 3、4 两行MyBatis通过配置文件来管理 SQL 以及输入参数的映射，
  - 6、7、8 行MyBatis获取返回结果到 Java 对象的映射，也是通过配置文件管理。



- 用了 **MyBatis** 之后，**只需要提供 SQL 语句就好了**，其余的诸如：建立连接、操作 Statment、ResultSet，处理 JDBC 相关异常等等都可以交给 MyBatis 去处理，我们的**关注点于是可以就此集中在 SQL 语句上**，关注在增删改查这些操作层面上

与Hibernate的对比



MyBatis

- 1、是一个SQL语句映射的框架（工具）
- 2、注重POJO与SQL之间的映射关系。不会为程序员在运行期自动生成 SQL
- 3、自动化程度低、手工映射SQL,灵活程度高.
- 4、需要开发人员熟炼掌据SQL语句

Hibernate

- 1、主流的ORM框架、提供了从 POJO 到数据库表的全套映射机制
- 2、会自动生成全套SQL语句。
- 3、因为自动化程度高、映射配置复杂，api也相对复杂，灵活性低.
- 4、开发人同不必关注SQL底层语句开发

MyBatis与Hibernate的比较

- Hibernate的映射关系：

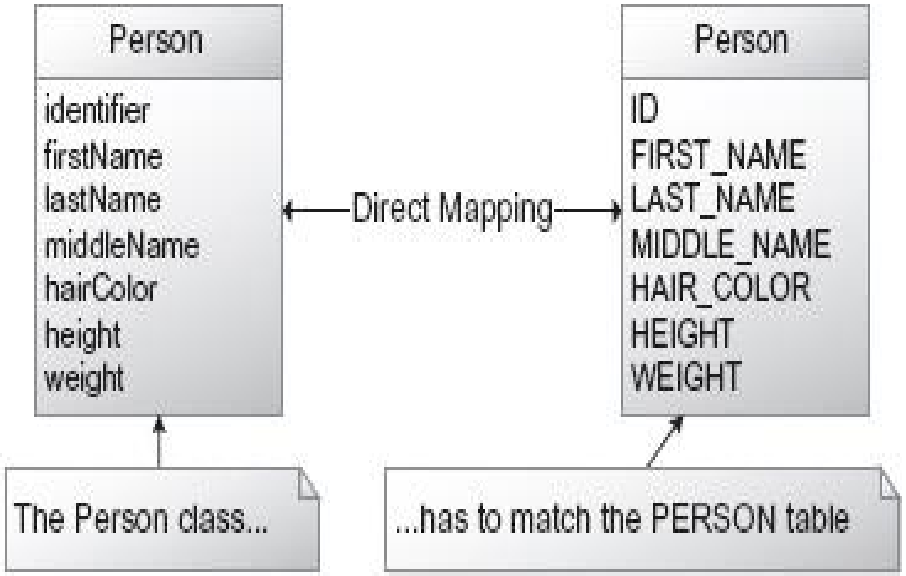


Figure 2.1
Object/relational mapping

MyBatis与Hibernate的比较

- MyBatis的映射关系：

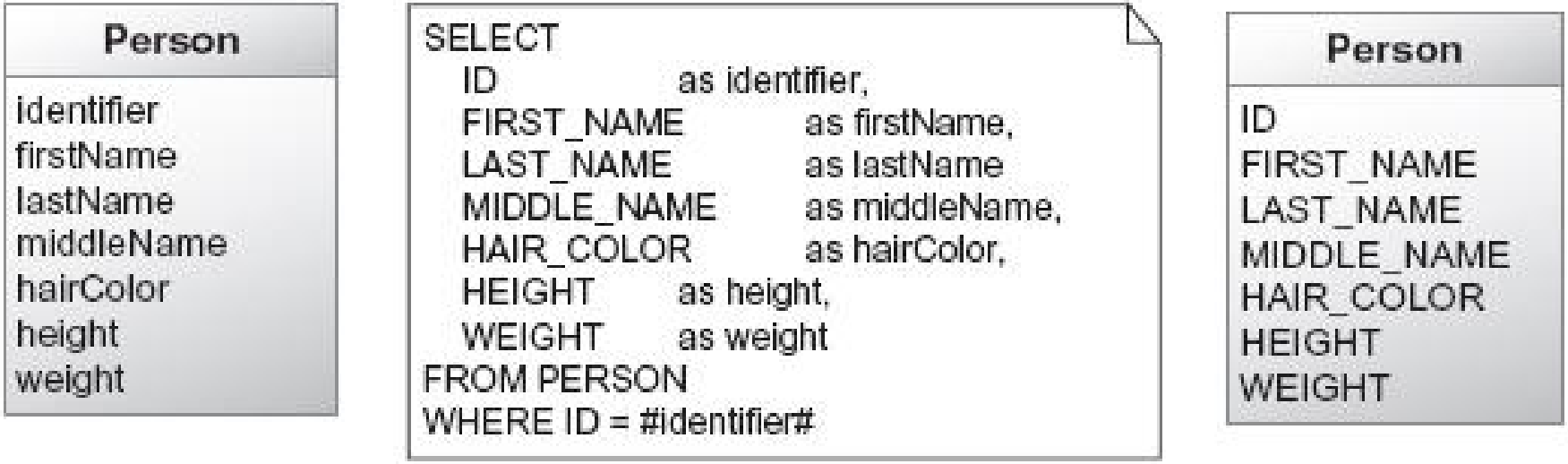
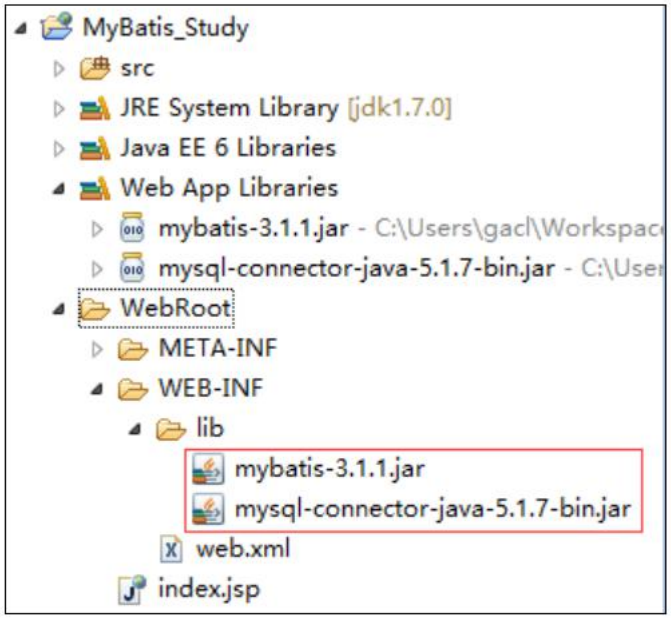


Figure 2.2 iBATIS SQL mapping

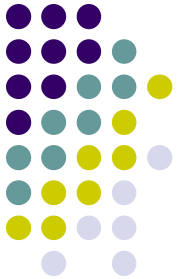
使用mybatis操作数据库

- 操作步骤
  - 创建项目-导入jar包
  - 创建mybatis配置文件
  - 创建实体类
  - 创建实体类对应的映射器及其配置文件
  - 操作数据
  - 测试操作

入门例子

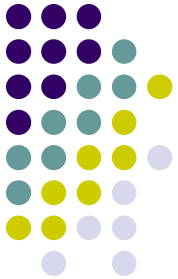

MyBatis_Study
src
JRE System Library [jdk1.7.0]
Java EE 6 Libraries
Web App Libraries
mybatis-3.1.1.jar - C:\Users\gacl\Workspac
mysql-connector-java-5.1.7-bin.jar - C:\User
WebRoot
META-INF
WEB-INF
lib
mybatis-3.1.1.jar
mysql-connector-java-5.1.7-bin.jar
web.xml
index.jsp

创建数据库和表

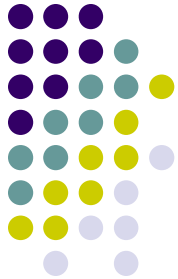


- create database mybatis;
- use mybatis;
- CREATE TABLE users(id INT PRIMARY KEY AUTO_INCREMENT, NAME VARCHAR(**20**), age INT);
- INSERT INTO users(NAME, age) VALUES('孤傲苍狼', **27**);
- INSERT INTO users(NAME, age) VALUES('白虎神皇', **27**);
- CREATE USER 'test'@'localhost' IDENTIFIED BY '123456'; -- *授予权限*
- GRANT ALL PRIVILEGES ON *.* TO 'test'@'localhost' WITH GRANT OPTION;
- -- *刷新权限使其生效*
- FLUSH PRIVILEGES;

mybatis配置文件



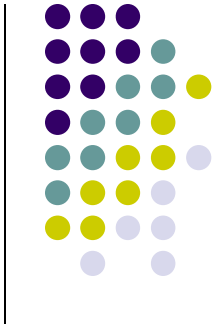
- 添加**Mybatis**的配置文件**conf.xml**
- <?xml version="1.0" encoding="UTF-8"?>
- <!DOCTYPE configuration PUBLIC "-//mybatis.org//DTD Config 3.0//EN" "http://mybatis.org/dtd/mybatis-3-config.dtd">
- <configuration>
- <environments default="development">
- <environment id="development">
- <transactionManager type="JDBC" />
- <!-- 配置数据库连接信息 -->
- <dataSource type="POOLED">
- <property name="driver" value="com.mysql.jdbc.Driver" />
- <property name="url" value="jdbc:mysql://localhost:3306/mybatis" />
- <property name="username" value="root" />
- <property name="password" value="XDP" />
- </dataSource>
- </environment>
- </environments>
-
- </configuration>



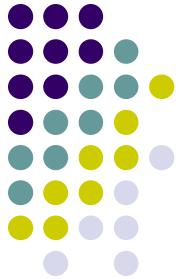
前面已经介绍了，在所有的 ORM 框架中都有一个非常重要的媒介：PO（持久化对象）。持久化对象的作用就是完成持久化操作，简单地说，就是通过该对象可对数据库执行增、删、改的操作，以面向对象的方式操作数据库。

MyBatis 中的 PO 是非常简单的，其是低侵入式的设计，完全采用普通的 Java 对象作为持久化对象使用。下面即是一个 POJO（普通的、传统的 Java 对象）类。

2、定义表所对应的实体类

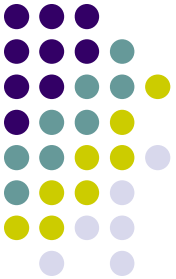


```
package me.gacl.domain;
public class User {
//实体类的属性和表的字段名称一一对应
    private int id;
    private String name;
    private int age;
    public int getId() {
        return id;
    }
public void setId(int id) {
        this.id = id;
    }
public String getName() {
        return name;
    }
public void setName(String name) {
        this.name = name;
    }
public int getAge() {
        return age;
    }
public void setAge(int age) {
        this.age = age;
    }
```



- Mybatis是通过XML文件去完成持久化类和数据库表之间的映射关系的

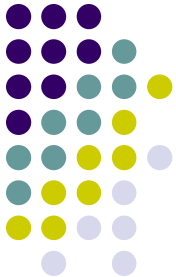
定义操作users表的sql映射文件userMapper.xml



- <?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">
- <!-- 为这个mapper指定一个唯一的namespace，namespace的值习惯上设置成包名+sql映射文件名，这样就能够保证namespace的值是唯一的
- 例如namespace="me.gacl.mapping.userMapper"就是me.gacl.mapping(包名)+userMapper(userMapper.xml文件去除后缀) -->
- <mapper namespace="me.gacl.mapping.userMapper">
- <!-- 在select标签中编写查询的SQL语句， 设置select标签的id属性为getUser，id属性值必须是唯一的，不能够重复。 使用parameterType属性指明查询时使用的参数类型，resultType属性指明查询返回的结果集类型。resultType="me.gacl.domain.User"就表示将查询结果封装成一个User类的对象返回User类就是users表所对应的实体类-->
- <!-- 根据id查询得到一个user对象-->
- <select id="getUser" parameterType="int"
- resultType="me.gacl.domain.User">
- select * from users where id=#{id}
- </select>
- </mapper>

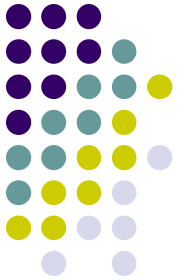
作用：根据id，查询users表

在conf.xml文件中注册userMapper.xml文件



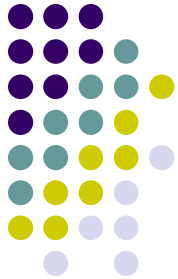
```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE configuration PUBLIC "-//mybatis.org//DTD Config 3.0//EN" "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
    <environments default="development">
        <environment id="development">
            <transactionManager type="JDBC" />
            <!-- 配置数据库连接信息 -->
            <dataSource type="POOLED">
                <property name="driver" value="com.mysql.jdbc.Driver" />
                <property name="url" value="jdbc:mysql://localhost:3306/mybatis" />
                <property name="username" value="root" />
                <property name="password" value="XDP" />
            </dataSource>
        </environment>
    </environments>    <mappers>
        <!-- 注册userMapper.xml文件，
        userMapper.xml位于me.gacl.mapping这个包下，所以resource写成me/gacl/mapping/userMapper.xml-->
        <mapper resource="me/gacl/mapping/userMapper.xml"/>
    </mappers>

</configuration>
```

测试



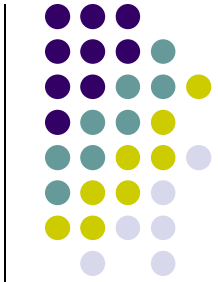
```
package me.gacl.test;
public class Test1 {
    public static void main(String[] args) throws IOException {
        //mybatis的配置文件
        String resource = "conf.xml";
        //使用类加载器加载mybatis的配置文件（它也加载关联的映射文件）
        InputStream is = Test1.class.getClassLoader().getResourceAsStream(resource);
        //构建sqlSession的工厂
        SqlSessionFactory sessionFactory = new SqlSessionFactoryBuilder().build(is);
        //使用MyBatis提供的Resources类加载mybatis的配置文件（它也加载关联的映射文件
        //Reader reader = Resources.getResourceAsReader(resource);
        //构建sqlSession的工厂
        //SqlSessionFactory sessionFactory = new SqlSessionFactoryBuilder().build(reader)
        //创建能执行映射文件中sql的sqlSession
        SqlSession session = sessionFactory.openSession();
        /**
         * 映射sql的标识字符串，
         * me.gacl.mapping.userMapper是userMapper.xml文件中mapper标签的namespace属
    ，
         * getUser是select标签的id属性值，通过select标签的id属性值就可以找到要执行的SQ
         */
        String statement = "me.gacl.mapping.userMapper.getUser";//映射sql的标识字符串
        //执行查询返回一个唯一user对象的sql
        User user = session.selectOne(statement, 1);
        System.out.println(user);
    }
}
```

操作数据



- 步骤
  - 读取并解析配置文件，创建SqlSessionFactory。
  - 从SqlSessionFactory中获取SqlSession对象。
  - 使用SqlSession对象操作数据。
  - 提交事务（回滚事务）。
  - 关闭SqlSession对象。

SqlSessionFactory

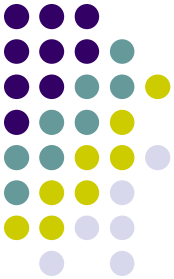


SqlSessionFactory 是 MyBatis 的关键对象，它是单个数据库映射关系经过编译后的内存镜像。SqlSessionFactory 对象的实例可以通过 SqlSessionFactoryBuilder 对象来获得，而 SqlSessionFactoryBuilder 则可以从 XML 配置文件或一个预先定制的 Configuration 的实例构建出 SqlSessionFactory 的实例。每一个 MyBatis 的应用程序都以一个 SqlSessionFactory 对象的实例为核心。其也是线程安全的，SqlSessionFactory 一旦被创建，应该在应用执行期间都存在。在应用运行期间不要重复创建多次，建议使用单例模式。SqlSessionFactory 是创建 SqlSession 的工厂。

SqlSessionFactory 的常用方法如下：

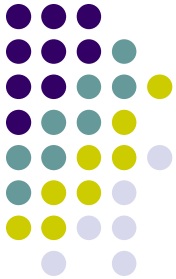
- SqlSession openSession()。创建 SqlSession 对象。

创建SqlSessionFactory



```
//配置文件的路径（在src下）
String resource = "mybatis-config.xml";
//读取配置文件
Reader reader = Resources.getResourceAsReader(resource);
//创建SqlSessionFactoryBuilder对象
SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();
//解析配置，构建SqlSessionFactory对象
SqlSessionFactory factory = builder.build(reader);
```

SqlSession



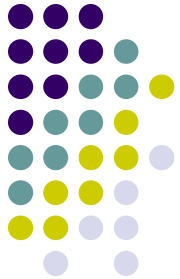
SqlSession 是 MyBatis 的关键对象，是执行持久化操作的对象，类似于 JDBC 中的 Connection。它是应用程序与持久存储层之间执行交互操作的一个单线程对象，也是 MyBatis 执行持久化操作的关键对象。SqlSession 对象完全包含以数据库为背景的所有执行 SQL 操作的方法，它的底层封装了 JDBC 连接，可以用 SqlSession 实例来直接执行已映射的 SQL 语句。每个线程都应该有它自己的 SqlSession 实例。SqlSession 的实例不能被共享，也是线程不安全的，绝对不能将 SqlSession 实例的引用放在一个类的静态字段甚至是实例字段中。也绝不能将 SqlSession 实例的引用放在任何类型的管理范围中，比如 Serlvet 当中的 HttpSession 对象中。使用完 SqlSession 之后关闭 Session 很重要，应该确保使用 finally 块来关闭它。

获取SqlSession对象

```
SqlSession sqlSession = factory.openSession();
```

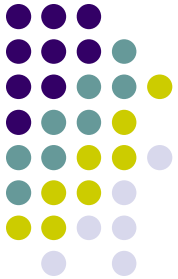
SqlSession的方法

<T> T selectOne(String statement, Object parameter)
<E> List<E> selectList(String statement, Object parameter)
<K,V> Map<K,V> selectMap(String statement, Object parameter, String mapKey)
int insert(String statement, Object parameter)
int update(String statement, Object parameter)
int delete(String statement, Object parameter)



- int insert(String statement)。插入方法，参数 statement 是在配置文件中定义的<insert.../>元素的 id，返回执行 SQL 语句所影响的行数。
- int insert(String statement,Object parameter)。插入方法，参数 statement 是在配置文件中定义的<insert.../>元素的 id，parameter 是插入所需的参数，通常是对象或者 Map，返回执行 SQL 语句所影响的行数。
- int update(String statement)。更新方法，参数 statement 是在配置文件中定义的<update.../>元素的 id，返回执行 SQL 语句所影响的行数。
- int update(String statement,Object parameter)。更新方法，参数 statement 是在配置文件中定义的<update.../>元素的 id，parameter 是插入所需的参数，通常是对象或者 Map，返回执行 SQL 语句所影响的行数。
- int delete(String statement)。删除方法，参数 statement 是在配置文件中定义的

使用SqlSession对象操作数据



- **SqlSession**对象封装了对数据的各种增加、删除、修改和查询操作

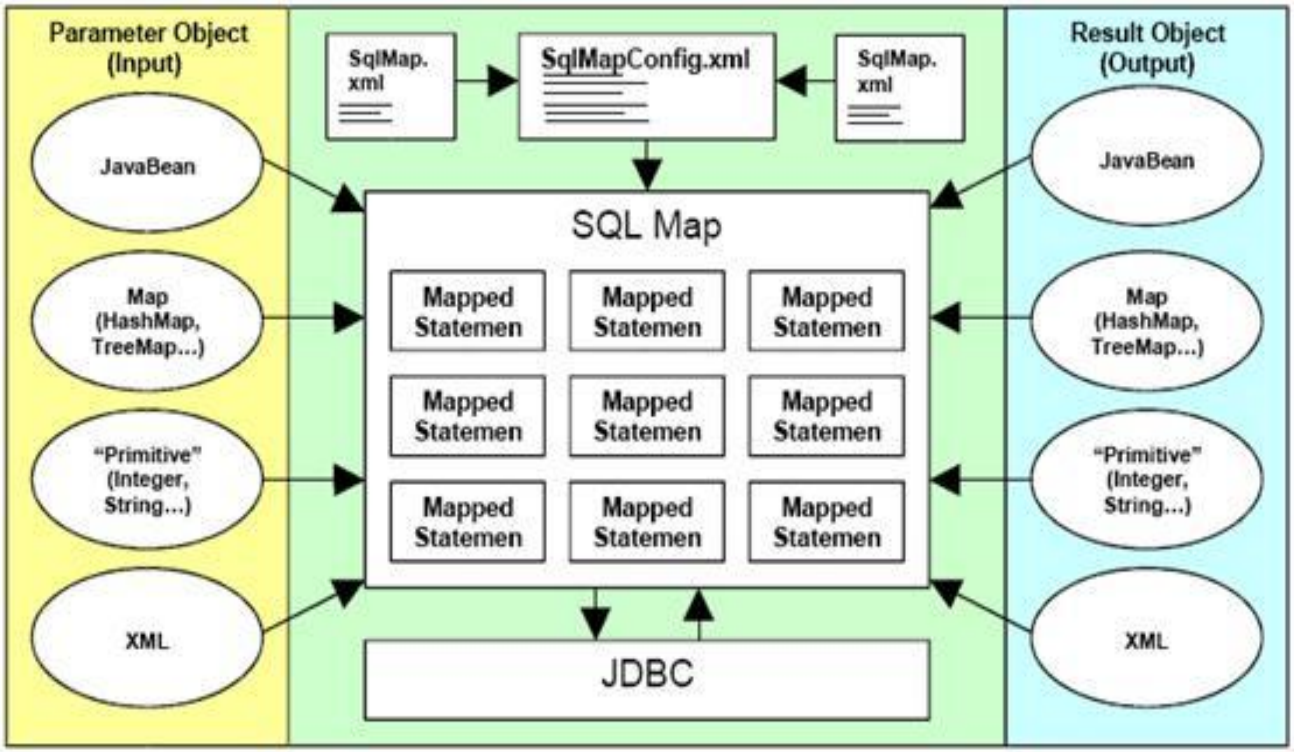
```
User user = new User("汤姆", "tom", new Date(), "湖北武汉", "18688889999", 1);
sqlSession.insert("insertUser", user);//insertUser为<insert>节点id
```

- 使用映射器接口（推荐）

```
User user = new User("汤姆", "tom", new Date(), "湖北武汉", "18688889999", 1);
UserMapper userMapper = sqlSession.getMapper(UserMapper.class);
userMapper.insertUser(user );
```

- 提交事务（回滚事务）。
  sqlSession.commit(); (sqlSession.rollback();)
- 关闭**SqlSession**对象。
  sqlSession.close();

MyBatis工作流程



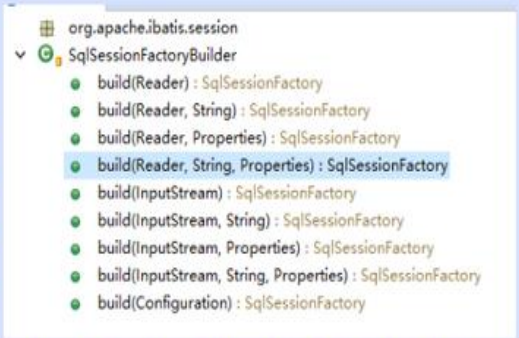
😊3、几个关键类

公众号java-mindmap

SqlSessionFactoryBuilder

概念：SqlSessionFactoryBuilder通过类名就可以看出这个类的主要作用就是创建一个SqlSessionFactory，通过输入mybatis配置文件的字节流或者字符流，生成XMLConfigBuilder，XMLConfigBuilder创建一个Configuration，Configuration这个类中包含了mybatis的配置的一切信息，mybatis进行的所有操作都需要根据Configuration中的信息来进行。

作用域（Scope）和生命周期：
- 可以重用 SqlSessionFactoryBuilder 来创建多个 SqlSessionFactory 实例，但是最好还是不要让其一直存在以保证所有的 XML 解析资源开放给更重要的事情
- 这个类可以被实例化、使用和丢弃，一旦创建了 SqlSessionFactory，就不再需要它了。因此 SqlSessionFactoryBuilder 实例的最佳作用域是方法作用域（也就是局部方法变量）



SqlSessionFactory接口

概念：sql会话工厂，用于创建SqlSession

作用域（Scope）和生命周期：
- SqlSessionFactory 一旦被创建就应该在应用的运行期间一直存在，没有任何理由对它进行清除或重建
- 最佳作用域是应用作用域。有很多方法可以做到，最简单的就是使用单例模式或者静态单例模式。

如何创建：

使用xml构建：

```
String resource = "org/mybatis/example/mybatis-config.xml";
InputStream inputStream = Resources.getResourceAsStream(resource);
SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);
```

java代码构建：

```
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);
```

SqlSession

概念：
- SqlSession是MyBatis的一个重要接口，定义了数据库的增删改查以及事务管理的常用方法。
- SqlSession还提供了查找Mapper接口的有关方法。

每个线程都应该有它自己的 SqlSession 实例

😊4、mybatis执行浅析

公众号java-mindmap

三层功能架构

http://www.jian...

JDBC执行流程

mybatis在jdbc的基础上进行了封装

1. 加载JDBC驱动；
2. 建立并获取数据库连接；
3. 创建 JDBC Statements 对象；
4. 设置SQL语句的传入参数；
5. 执行SQL语句并获得查询结果；
6. 对查询结果进行转换处理并将处理结果返回；
7. 释放相关资源（关闭Connection，关闭Statement，关闭ResultSet）；

数据处理层

- 通过传入参数构建动态SQL语句
  - MyBatis 通过传入的参数值，使用 Ognl 来动态地构造SQL语句，使得MyBatis 有很强的灵活性和扩展性
  - 参数映射指的是对于java 数据类型和jdbc数据类型之间的转换
- SQL语句的执行以及封装查询结果集成List<E>
  - 支持结果集关系一对多和多对一的转换
  - 嵌套查询语句的查询
  - 嵌套结果集的查询

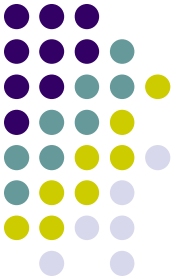
API接口层

- 使用传统的MyBatis提供的API
- 使用Mapper接口
  - 接口中声明的方法和<mapper> 节点中的<select|update|delete|insert> 节点项对应
  - 即<select|update|delete|insert> 节点的id值为Mapper 接口中的方法名称
  - parameterType 值表示Mapper 对应方法的入参类型
  - resultMap 值则对应了Mapper 接口表示的返回值类型或者返回结果集的元素类型

基础支撑层

1. 事务管理机制
2. 连接池管理机制
3. 缓存机制
4. SQL语句的配置方式

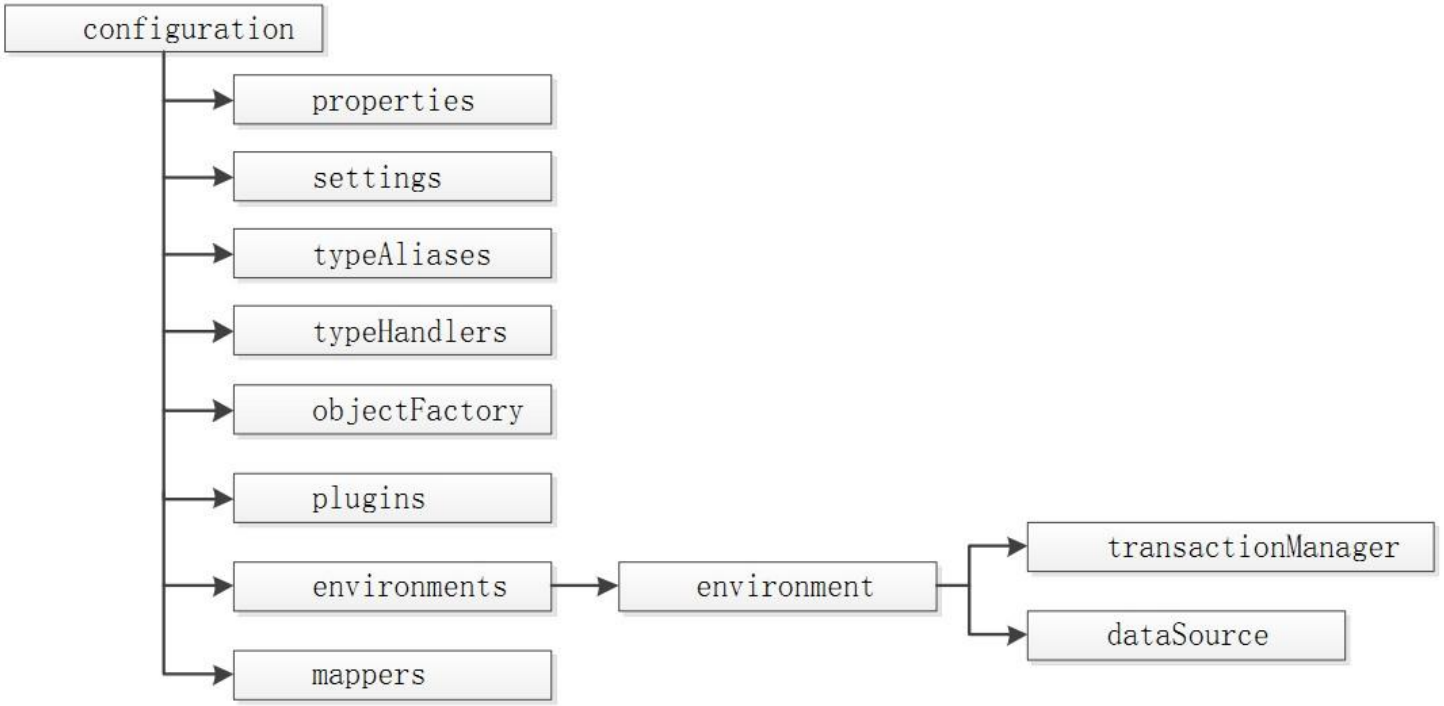
MyBatis基本要素



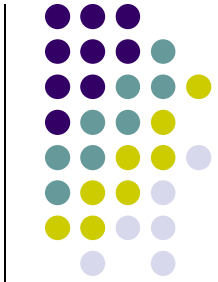
- 一、configuration.xml 全局配置文件
- 二、mapper.xml 核心映射文件
- 三、SqlSession接口

基础配置文件configuration.xml

configuration.xml是系统的核心配置文件，包含数据源和事务管理器等设置和属性信息，XML文档结构如下：



配置文件的元素properties



- 该元素是外部化的、可替代的属性，这些属性也可以配置在典型的Java属性配置文件中，或者通过**properties**元素的子元素来传递。例如：

```
<properties resource="org/mybatis/example/config.properties">
    <property name="username" value="dev_user"/>
    <property name="password" value="F2Fa3!33TYyg"/>
</properties>
```

- 其中的属性值就可以在整个配置文件中使用，使用可替换的属性来实现动态配置。例如：

```
<dataSource type="POOLED">
    <property name="driver" value="${driver}"/>
    <property name="url" value="${url}"/>
    <property name="username" value="${username}"/>
    <property name="password" value="${password}"/>
</dataSource>
```

settings

- 这些是极其重要的调整，它们会修改 MyBatis 在运行时的行为方式。

```
<!--
    这些是极其重要的调整，它们会修改 MyBatis 在运行时的行为方式。
    cacheEnabled 使全局的映射器启用或禁用 缓存
    lazyLoadingEnabled 全局启用或禁用延迟加载。当禁用时，所有关联对象都会即时加载。
 -->
<settings>
    <setting name="cacheEnabled" value="true"/>
    <setting name="lazyLoadingEnabled" value="true"/>
</settings>
```

typeAliases

- 类型别名即为**Java**类型命名一个短的名称。它仅同**XML**配置有关，只用于减少类完全限定名的多余部分。

```
<typeAliases>
    <typeAlias alias="Author" type="domain.blog.Author"/>
    <typeAlias alias="Blog" type="domain.blog.Blog"/>
    <typeAlias alias="Comment" type="domain.blog.Comment"/>
    <typeAlias alias="Post" type="domain.blog.Post"/>
    <typeAlias alias="Section" type="domain.blog.Section"/>
    <typeAlias alias="Tag" type="domain.blog.Tag"/>
</typeAliases>
```

typeHandlers

- 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
类型。

```
// ExampleTypeHandler.java
public class ExampleTypeHandler implements TypeHandler {
public void setParameter(PreparedStatement ps, int i, Object
        parameter,JdbcType jdbcType) throws SQLException {
ps.setString(i, (String) parameter);
}
public Object getResult(ResultSet rs, String columnName)
throws SQLException {
return rs.getString(columnName);
}
public Object getResult(CallableStatement cs, int columnIndex)
throws SQLException {
return cs.getString(columnIndex);
}
}
```

environments

- **MyBatis**可以配置多种环境(开发环境,测试环境等)。这便于将**SQL**映射应用于多种数据库之中。

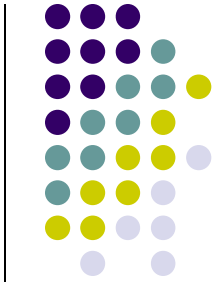
```
<environments default="development">
    <environment id="development">
    <transactionManager type="JDBC"/>
    <dataSource type="POOLED">
    <property name="driver" value="${driver}"/>
    <property name="url" value="${url}"/>
    <property name="username" value="${username}"/>
    <property name="password" value="${password}"/>
    </dataSource>
    </environment>
</environments>
```

transactionManager

- 在MyBatis中，有两种事务管理器类型（即type=”[JDBC|MANAGED]”）

（1）**JDBC**：该配置直接简单地使用了JDBC的提交和回滚设置。它依赖于从数据源得到的连接来管理事务范围。

（2）**MANAGED**：该配置几乎无用。它从来不提交或回滚一个连接，且它会让容器来管理事务的整个生命周期（如Spring或JEE应用服务器的上下文）。默认情况下，它会关闭连接。然而一些容器并不希望这样，因此,如果需要从连接中停止它，则需要将closeConnection属性设置为false。



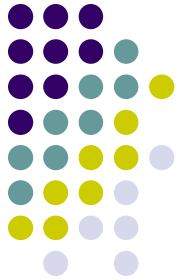
dataSource

- **dataSource元素使用基本的JDBC数据源接口来配置JDBC连接对象的资源。**
- **内建数据源类型有以下三种：**
- UNPOOLED - 这个数据源实现只是在每次请求的时候简单的打开和关闭一个连接。虽然这有点慢，但作为一些不需要性能和立即响应的简单应用来说，不失为一种好选择 。
- POOLED - 这个数据源缓存 JDBC 连接对象用于避免每次都要连接和生成连接实例而需要的验证时间 。对于并发 WEB 应用，这种方式非常流行因为它有最快的响应时间。
- JNDI - 这个数据源实现是为了准备和 Spring 或应用服务一起使用，可以在外部也可以在内部配置这个数据源，然后在 JNDI 上下文中引用它。这个数据源配置只需要两上属性：

mappers

- 既然**MyBatis**的行为已经由上述元素配置完毕，那么我们现在要定义**SQL**映射语句。这些语句简单阐述了**MyBatis**要从哪里寻找映射文件，其余的细节便在每个**SQL**映射文件中。

```
<mappers>
    <mapper resource="org/mybatis/builder/AuthorMapper.xml"/>
    <mapper resource="org/mybatis/builder/BlogMapper.xml"/>
    <mapper resource="org/mybatis/builder/PostMapper.xml"/>
</mappers>
```



MyBatis 的真正强大之处在于它的映射语句，这也是它的魔力所在。由于它的功能异常强大，映射器的 XML 文件就显得相对简单。如果拿它跟具有相同功能的 JDBC 代码进行对比，你会立即发现省掉了将近 95%的代码。MyBatis 就是针对 SQL 构建的，并且比普通的方法做得更好。

SQL映射的XML文件

- **SQL映射文件包括以下几个很少的顶级元素（按照它们应该被定义的顺序排序）：**
  - cache：配置给定命名空间的缓存。
  - cache-ref：从其他命名空间引用缓存配置。
  - resultMap：最复杂，也是最有力量的元素，用于描述如何从数据库结果集中加载对象。
  - sql：可重用的SQL块，也可以被其他语句引用。
  - insert：映射插入语句。
  - update：映射更新语句。
  - delete：映射删除语句。
  - select：映射查询语句。

select

- 查询语句是使用**MyBatis**时最常用的元素之一。当从数据库中取出数据时，会发现将数据存储在数据库中是很有价值的，所以许多应用程序的查询操作相较更改数据操作更多。

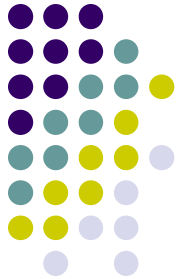
  <select id=”selectPerson” parameterType=”int”
  resultType=”hashmap”>//<resultMap>节点配置的**id**

  SELECT * FROM PERSON WHERE ID = #{id}

  </select>

- 该语句被称为**selectPerson**，使用一个**int**（或**Integer**）类型的参数，并返回一个**HashMap**类型的对象，其中的键是列名，值是列对应的值。
- ResultMap配置:
  - <resultMap type="java.util.HashMap" id="cityResult">
  - <id property="cityCode" column="city_code"/>
  - <result property="cityName" column="city_name"/>
  - <result property="stateCode" column="state_code"/>
  - </resultMap>

Select中的属性



属性	描述
id	在命名空间中唯一的标识符，可以被用于引用该语句。
parameterType	将会传入该语句的参数类的完全限定名或别名。
resultType	从该语句中返回的期望类型的类的完全限定名或别名。
resultMap	命名引用外部的resultMap。
flushCache	将其设置为true，无论语句什么时候被调用，都会导致缓存被清空。默认值为false。
useCache	将其设置为true，将会导致本条语句的结果被缓存。默认值为true。
timeout	该设置驱动程序等待数据库返回请求结果。
fetchSize	暗示驱动程序每次批量返回的结果行数。默认不设置（驱动自行处理）
statementType	STATEMENT、PREPARED或CALLABLE的一种。
resultSetType	FORWARD_ONLY|SCROLL_SENSITIVE|SCROLL_INSENSITIVE中的一种。默认不设置（驱动自行处理）。

insert、update和delete

- 数据修改语句insert、update和delete在它们

<insert

<delete
id="insertAuthor"
parameterType="domain.blog.Author"
flushCache="true"
statementType="PREPARED"
timeout="20000">

bio = #{bio}
where id = #{id}
</update>
<delete id="deleteAuthor" parameterType="int">
delete from Author where id = #{id}
</delete>

insert、update和delete的属性

属性	描述
id	在命名空间中唯一的标识符，可以被用于引用该语句。
parameterType	将会传入该语句的参数类的完全限定名或别名。
flushCache	将其设置为true，无论语句什么时候被调用，都会导致缓存被清空。默认值为false。
timeout	该设置驱动程序等待数据库返回请求结果，并抛出异常时间的最大等待值。默认不设置（驱动自行处理）。
statementType	STATEMENT、PREPARED或CALLABLE的一种。用于方便MyBatis选择使用Statement、PreparedStatement或CallableStatement。默认值为PREPARED。
useGeneratedKeys	（仅对insert有用）通知MyBatis使用JDBC的getGeneratedKeys方法来取出由数据（如MySQL和SQL Server的数据库管理系统的自动递增字段）内部生成的主键。默认值为false。
keyProperty	（仅对insert有用）标记一个属性，MyBatis会通过getGeneratedKeys或insert语句的selectKey子元素设置其值。默认不设置。

sql

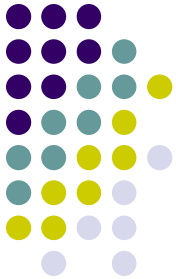
- 该元素可以被用于定义可重用的SQL代码段，可以包含在其他语句中。如：

```
<sql id=”userColumns”> id,username,password </sql>
```

上述SQL片段可以被包含在其他语句中。如：

```
<select id=”selectUsers” parameterType=”int”
    resultType=”hashmap”>
    select <include refid=”userColumns”/>
    from some_table
    where id = #{id}
</select>
```

resultMap



- **resultMap**元素是**MyBatis**中最重要、最强大的元素。
- 解决列名与属性名称不匹配的问题：

```
<resultMap id="userResultMap" type="User">
    <id property="id" column="user_id" />
    <result property="username" column="user_name"/>
    <result property="password" column="hashed_password"/>
</resultMap>
```

- 其引用语句使用**resultMap**属性即可（注意，我们去掉了**resultType**属性）。例如：

```
<select id=”selectUsers” parameterType=”int”
    resultMap=”userResultMap”>
    select user_id, user_name, hashed_password
    from some_table
    where id = #{id}
</select>
```

Mybatis的范围与生命周期

- **SqlSessionFactoryBuilder**：该类可以被实例化、使用和丢弃。
- **SqlSessionFactory**：一旦被创建，**SqlSessionFactory**实例会一直存在于应用程序执行期间。
- **Sq**
  Sq
- 映
  口
  获
  器实例的最宽范围和**SqlSession**是相同的。

```
SqlSession session = sqlSessionFactory.openSession();
try {
        mapper = session.getMapper(BlogMapper.class);
// do work
} finally {
session.close();
}
```

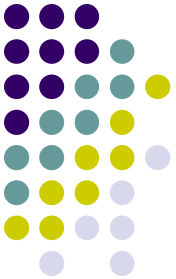
😊13、mapper的xml文件（一）

微信公众号java-mindmap

```
<select
  id="selectPerson"
  parameterType="int"
  parameterMap="deprecated"
  resultType="hashmap"
  resultMap="personResultMap"
  flushCache="false"
  useCache="true"
  timeout="10000"
  fetchSize="256"
  statementType="PREPARED"
  resultSetType="FORWARD_ONLY">
  SELECT * FROM PERSON WHERE ID = #{id}
</select>
```

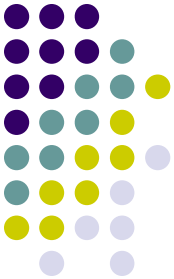
- select – 映射查询语句
  - 实例
    - 这个语句被称作 selectPerson，接受一个 int（或 Integer）类型的参数，并返回一个 HashMap 类型的对象，其中的键是列名，值便是结果行中的对应值
  - id
    - 在命名空间中唯一的标识符，可以被用来引用这条语句
    - 必选
  - parameterType
    - 将会传入这条语句的参数类的完全限定名或别名
    - 可选
  - resultType
    - 从这条语句中返回的期望类型的类的完全限定名或别名
    - 注意
      - 1 如果是集合情形，那应该是集合可以包含的类型，而不能是集合本身
      - 2 使用 resultType 或 resultMap，但不能同时使用
  - resultMap
    - 外部 resultMap 的命名引用
    - 注意 使用 resultMap 或 resultType，但不能同时使用
  - 属性
    - flushCache
      - 将其设置为 true，任何时候只要语句被调用，都会导致本地缓存和二级缓存都会被清空
      - 默认值：false
    - useCache
      - 将其设置为 true，将会导致本条语句的结果被二级缓存
      - 默认值：对 select 元素为 true
    - timeout
      - 这个设置是在抛出异常之前，驱动程序等待数据库返回请求结果的秒数
      - 默认值为 unset（依赖驱动）
    - fetchSize
      - 这是尝试影响驱动程序每次批量返回的结果行数和这个设置值相等
      - 默认值为 unset（依赖驱动）
    - statementType
      - STATEMENT，PREPARED 或 CALLABLE 的一个
      - 这会让 MyBatis 分别使用 Statement，PreparedStatement 或 CallableStatement
      - 默认值：PREPARED
    - resultSetType
      - 这个设置仅对多结果集的情况适用，它将列出语句执行后返回的结果集并每个结果集给一个名称，名称是逗号分隔的。
- delete – 映射删除语句
- insert – 映射插入语句
- update – 映射更新语句
  - useGeneratedKeys
    - （仅对 insert 和 update 有用）这会令 MyBatis 使用 JDBC 的 getGeneratedKeys 方法来取出由数据库内部生成的主键（比如：像 MySQL 和 SQL Server 这样的关系数据库管理系统的自动递增字段）

使用MyBatis对表执行CRUD操作——基于XML的实现



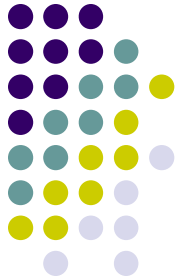
- <?xml version="1.0" encoding="UTF-8" ?>
- <!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
- "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
- <!-- 为这个mapper指定一个唯一的namespace，namespace的值习惯上设置成包名+sql映射文件名，这样就能够保证namespace的值是唯一的
- 例如namespace="me.gacl.mapping.userMapper"就是me.gacl.mapping(包名)+userMapper(userMapper.xml文件去除后缀) -->
- <mapper namespace="me.gacl.mapping.userMapper">
- <!-- 在select标签中编写查询的SQL语句， 设置select标签的id属性为getUser，id属性值必须是唯一的，不能够重复
- 使用parameterType属性指明查询时使用的参数类型，resultType属性指明查询返回的结果集类型
- resultType="me.gacl.domain.User"就表示将查询结果封装成一个User类的对象返回
- User类就是users表所对应的实体类-->
- <!-- 根据id查询得到一个user对象-->
- <select id="getUser" parameterType="int"
- resultType="me.gacl.domain.User">
- select * from users where id=#{id}
- </select>

使用MyBatis对表执行CRUD操作——基于XML的实现



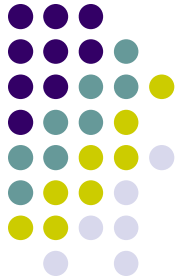
```
    <!-- 创建用户(Create) -->
    <insert id="addUser" parameterType="me.gacl.domain.User">
        insert into users(name,age) values(#{name},#{age})
    </insert>
    <!-- 删除用户(Remove) -->
    <delete id="deleteUser" parameterType="int">
        delete from users where id=#{id}
    </delete>    <!-- 修改用户(Update) -->
    <update id="updateUser" parameterType="me.gacl.domain.User">
        update users set name=#{name},age=#{age} where id=#{id}
    </update>
    <!-- 查询全部用户-->
    <select id="getAllUsers" resultType="me.gacl.domain.User">
        select * from users
    </select>
</mapper>
```

```
package me.gacl.test;

import java.util.List;
import me.gacl.domain.User;
import me.gacl.util.MyBatisUtil;
import org.apache.ibatis.session.SqlSession;
import org.junit.Test;

public class TestCRUDByXmlMapper {

    @Test
    public void testAdd(){
        //SqlSession sqlSession = MyBatisUtil.getSqlSession(false);
        SqlSession sqlSession = MyBatisUtil.getSqlSession(true);
        /**
         * 映射sql的标识字符串，
         * me.gacl.mapping.userMapper是userMapper.xml文件中mapper标签的namespace属性的值，
         * addUser是insert标签的id属性值，通过insert标签的id属性值就可以找到要执行的SQL
         */
        String statement = "me.gacl.mapping.userMapper.addUser";//映射sql的标识字符串
        User user = new User();
        user.setName("用户孤傲苍狼");
        user.setAge(20);
        //执行插入操作
        int retResult = sqlSession.insert(statement,user);
        //手动提交事务
        //sqlSession.commit();
        //使用SqlSession执行完SQL之后需要关闭SqlSession
        sqlSession.close();
        System.out.println(retResult);
    }

    @Test
    public void testUpdate(){
        SqlSession sqlSession = MyBatisUtil.getSqlSession(true);
        /**
         * 映射sql的标识字符串，
         * me.gacl.mapping.userMapper是userMapper.xml文件中mapper标签的namespace属性的值，
         * updateUser是update标签的id属性值，通过update标签的id属性值就可以找到要执行的SQL
         */
        String statement = "me.gacl.mapping.userMapper.updateUser";//映射sql的标识字符串
        User user = new User();
        user.setId(3);
        user.setName("孤傲苍狼");
        user.setAge(25);
        //执行修改操作
        int retResult = sqlSession.update(statement,user);
        //使用SqlSession执行完SQL之后需要关闭SqlSession
        sqlSession.close();
        System.out.println(retResult);
    }

```



```
public class MyBatisUtil {

    /**
     * 获取SqlSessionFactory
     * @return SqlSessionFactory
     */
    public static SqlSessionFactory getSqlSessionFactory() {
        String resource = "conf.xml";
        InputStream is = MyBatisUtil.class.getClassLoader().getResourceAsStream(resource);
        SqlSessionFactory factory = new SqlSessionFactoryBuilder().build(is);
        return factory;
    }

    /**
     * 获取SqlSession
     * @return SqlSession
     */
    public static SqlSession getSqlSession() {
        return getSqlSessionFactory().openSession();
    }

    /**
     * 获取SqlSession
     * @param isAutoCommit
     *        true 表示创建的SqlSession对象在执行完SQL之后会自动提交事务
     *        false 表示创建的SqlSession对象在执行完SQL之后不会自动提交事务，这时就需要我们手动调用sqlSession.commit()提交事务
     * @return SqlSession
     */
    public static SqlSession getSqlSession(boolean isAutoCommit) {
        return getSqlSessionFactory().openSession(isAutoCommit);
    }
}
```

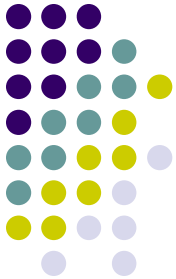
使用MyBatis对表执行CRUD操作——基于注解的实现



定义sql映射的接口

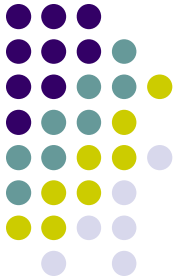
```
/**
 * @author gacl
 * 定义sql映射的接口，使用注解指明方法要执行的SQL
 */
public interface UserMapperI {

    //使用@Insert注解指明add方法要执行的SQL
    @Insert("insert into users(name, age) values(#{name}, #{age})")
    public int add(User user);

    //使用@Delete注解指明deleteById方法要执行的SQL
    @Delete("delete from users where id=#{id}")
    public int deleteById(int id);

    //使用@Update注解指明update方法要执行的SQL
    @Update("update users set name=#{name},age=#{age} where id=#{id}")
    public int update(User user);

    //使用@Select注解指明getById方法要执行的SQL
    @Select("select * from users where id=#{id}")
    public User getById(int id);

    //使用@Select注解指明getAll方法要执行的SQL
    @Select("select * from users")
    public List<User> getAll();
}
```

在conf.xml文件中注册这个映射接口

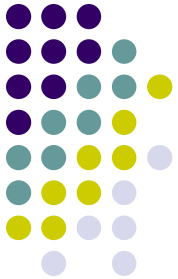


```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE configuration PUBLIC "-//mybatis.org//DTD Config 3.0//EN" "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
    <environments default="development">
        <environment id="development">
            <transactionManager type="JDBC" />
            <!-- 配置数据库连接信息 -->
            <dataSource type="POOLED">
                <property name="driver" value="com.mysql.jdbc.Driver" />
                <property name="url" value="jdbc:mysql://localhost:3306/mybatis" />
                <property name="username" value="root" />
                <property name="password" value="XDP" />
            </dataSource>
        </environment>
    </environments>

    <mappers>
        <!-- 注册userMapper.xml文件，
        userMapper.xml位于me.gacl.mapping这个包下，所以resource写成me/gacl/mapping/userMapper.xml-->
        <mapper resource="me/gacl/mapping/userMapper.xml"/>
        <!-- 注册UserMapper映射接口-->
        <mapper class="me.gacl.mapping.UserMapperI"/>
    </mappers>

</configuration>
```

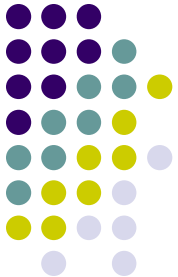
单元测试类



```
public class TestCRUDByAnnotationMapper {
    @Test
    public void testAdd(){
        SqlSession sqlSession = MyBatisUtil.getSqlSession(true);
        //得到UserMapperI接口的实现类对象，UserMapperI接口的实现类对象由sqlSession.getMapper(UserMapperI.class)动态构建出来
        UserMapperI mapper = sqlSession.getMapper(UserMapperI.class);
        User user = new User();
        user.setName("用户xdp");
        user.setAge(20);
        int add = mapper.add(user);
        //使用SqlSession执行完SQL之后需要关闭SqlSession
        sqlSession.close();
        System.out.println(add);
    }

    @Test
    public void testUpdate(){
        SqlSession sqlSession = MyBatisUtil.getSqlSession(true);
        //得到UserMapperI接口的实现类对象，UserMapperI接口的实现类对象由sqlSession.getMapper(UserMapperI.class)动态构建出来
```



```
@Test
public void testGetUser(){
    SqlSession sqlSession = MyBatisUtil.getSqlSession();
    //得到UserMapperI接口的实现类对象，UserMapperI接口的实现类对象由sqlSession.getMapper(UserMapperI.class)动态构建出来
    UserMapperI mapper = sqlSession.getMapper(UserMapperI.class);
    //执行查询操作，将查询结果自动封装成User返回
    User user = mapper.getById(8);
    //使用SqlSession执行完SQL之后需要关闭SqlSession
    sqlSession.close();
    System.out.println(user);
}

@Test
public void testGetAll(){
    SqlSession sqlSession = MyBatisUtil.getSqlSession();
    //得到UserMapperI接口的实现类对象，UserMapperI接口的实现类对象由sqlSession.getMapper(UserMapperI.class)动态构建出来
    UserMapperI mapper = sqlSession.getMapper(UserMapperI.class);
    //执行查询操作，将查询结果自动封装成List<User>返回
    List<User> lstUsers = mapper.getAll();
    //使用SqlSession执行完SQL之后需要关闭SqlSession
    sqlSession.close();
    System.out.println(lstUsers);
```



```
public class MyBatisUtil {
/**
     * 获取SqlSessionFactory
     * @return SqlSessionFactory
     */
    public static SqlSessionFactory getSqlSessionFactory() {
        String resource = "conf.xml";
        InputStream is = MyBatisUtil.class.getClassLoader().getResourceAsStream(resource);
        SqlSessionFactory factory = new SqlSessionFactoryBuilder().build(is);
        return factory;
    } /**
     * 获取SqlSession
     * @return SqlSession
     */
    public static SqlSession getSqlSession() {
        return getSqlSessionFactory().openSession();
    }
 /**
     * 获取SqlSession
     * @param isAutoCommit
     *        true 表示创建的SqlSession对象在执行完SQL之后会自动提交事务
     *        false 表示创建的SqlSession对象在执行完SQL之后不会自动提交事务，这时就需要我们手动调用
sqlSession.commit()提交事务
     * @return SqlSession
     */
    public static SqlSession getSqlSession(boolean isAutoCommit) {
        return getSqlSessionFactory().openSession(isAutoCommit);
```