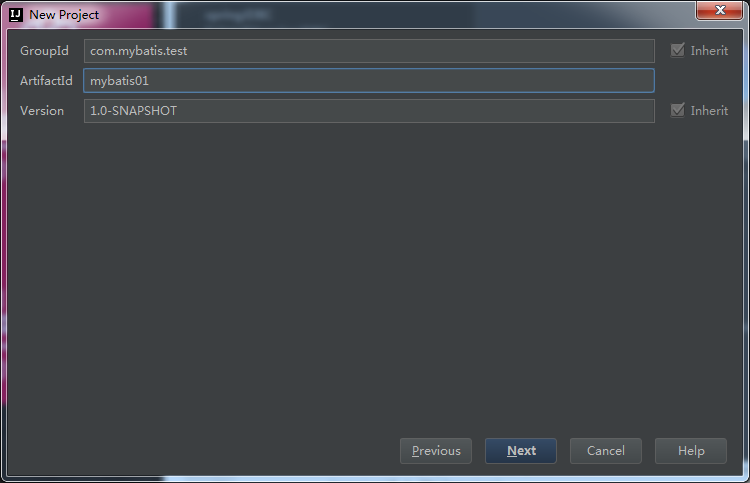
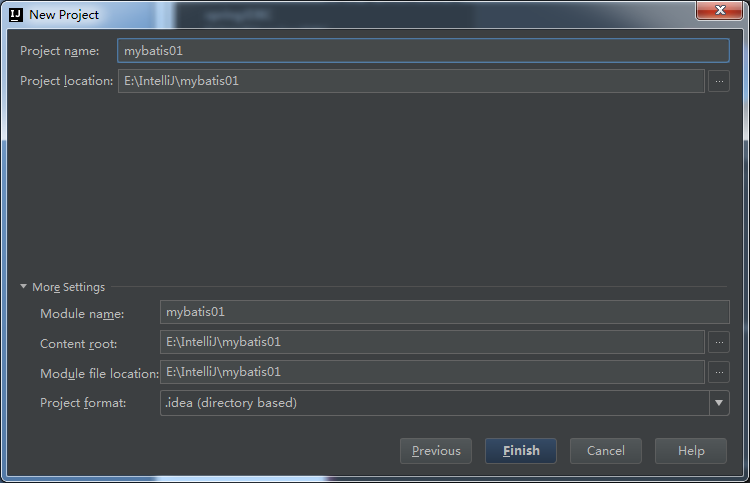
# Mybatis

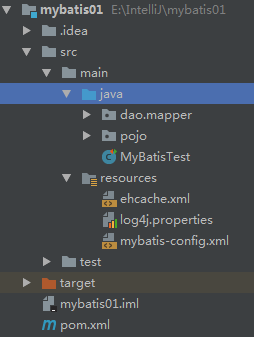
## 环境搭建

### 创建工程





本次项目示例工程完整的目录结构如下：



### 导入jar包

pom.xml中增加如下配置：

<dependencies>  
 <!-- mybatis核心包 -->  
 <dependency>  
 <groupId>org.mybatis</groupId>  
 <artifactId>mybatis</artifactId>  
 <version>3.4.1</version>  
 </dependency>  
 <!-- mysql驱动包 -->  
 <dependency>  
 <groupId>mysql</groupId>  
 <artifactId>mysql-connector-java</artifactId>  
 <version>5.1.29</version>  
 </dependency>  
 <!-- junit测试包 -->  
 <dependency>  
 <groupId>junit</groupId>  
 <artifactId>junit</artifactId>  
 <version>4.11</version>  
 </dependency>  
 <!--日志包，方便看sql语句-->  
 <dependency>  
 <groupId>org.slf4j</groupId>  
 <artifactId>slf4j-log4j12</artifactId>  
 <version>1.6.1</version>  
 </dependency>  
 <dependency>  
 <groupId>org.slf4j</groupId>  
 <artifactId>slf4j-api</artifactId>  
 <version>1.6.1</version>  
 </dependency>  
</dependencies>  
<build>  
 <resources>  
 <resource>  
 <directory>src/main/java</directory>  
 <includes>  
 <include>\*\*/\*.xml</include>  
 </includes>  
 <filtering>true</filtering>  
 </resource>  
 <resource>  
 <directory>src/main/resources</directory>  
 <includes>  
 <include>\*\*/\*.xml</include>  
 <include>\*\*/\*.properties</include>  
 </includes>  
 </resource>  
 </resources>  
</build>

### 创建实体类

pojo包下创建两个实体类：

package pojo;  
  
import java.util.List;  
  
public class Dept {  
 private Integer id;  
 private String name;  
 private List<Employee> employees;  
 // getter/setter方法略

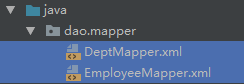
}

package pojo;  
  
import java.util.Date;  
  
public class Employee {  
 private Integer id;  
 private String name;  
 private Dept dept;  
 private String job;  
 private Float salary;  
 private Date hireDate;  
 //getter/setter方法略

}

### 创建Mapper映射文件

在dao.mapper包下创建映射文件：



DeptMapper.xml和EmployeeMapper.xml是一样的，只是mapper节点namespace的值不同。一个工程中，namespace的值要唯一。

通常一个Mapper文件对应一张表的增删改查。namespace的值可以设置成表名。

一个空的Mapper映射文件如下：

*<?*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="DEPT">  
 <!--<![CDATA[在这里写增删改查的语句]]>-->

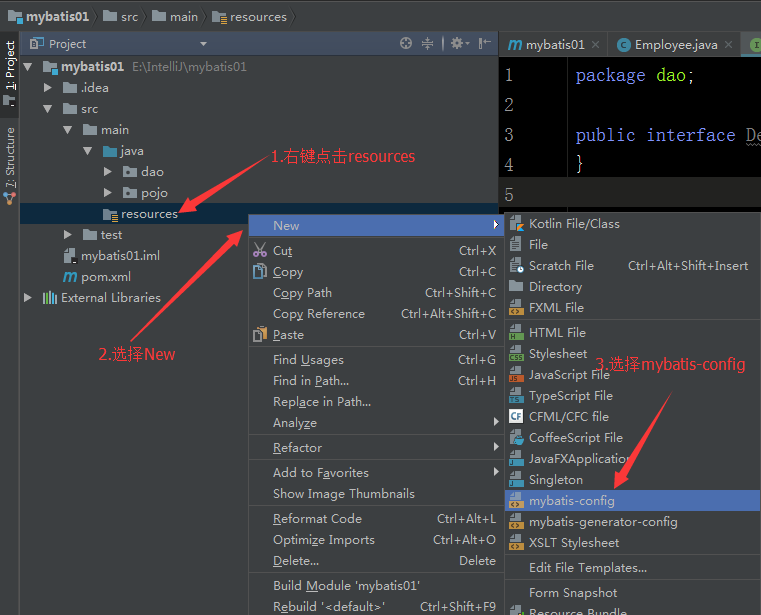
<!--<![CDATA[你的注释]]>-->  
</mapper>

注意Mapper文件中如果要写中文注释，需要用<![CDATA[]]>将中文注释包起来。否则运行时会报编码错误。或者把工程的文件编码设置成UTF-8，有两种方式:

1. File->Settings->Editor->File Encodings 这种方式修改的文件编码方式只对当前 project 起作用，每次新建了一个工程后还需要重新设置编码方式。
2. File->Other Settings->Default Settings->Editor->File Encodings ，这儿设置的是默认的文件编码方式，所有新建的工程使用的都是默认的文件编码方式。

### 创建配置文件

创建mybatis的配置文件：



完整的mybatis-config.xml如下：

*<?*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"*>*<!--注意configuration中节点的位置是有先后顺序的-->  
<configuration>  
 <typeAliases>  
 <!-- 类型别名，表示可以使用 Dept 来代替 pojo.Dept -->  
 <!-- 类型别名就是可以给类的全路径起一个简称 -->  
 <typeAlias type="pojo.Dept" alias="Dept"/>  
 <typeAlias type="pojo.Employee" alias="Employee"/>  
 </typeAliases>  
 <environments default="development">  
 <environment id="development">  
 <transactionManager type="JDBC"/>  
 <dataSource type="POOLED">  
 <!-- 数据库连接信息 -->  
 <property name="driver" value="com.mysql.jdbc.Driver"/>  
 <property name="url"

value="jdbc:mysql://localhost:3306/test?characterEncoding=UTF-8"/>  
 <property name="username" value="用户名"/>  
 <property name="password" value="密码"/>  
 </dataSource>  
 </environment>  
 </environments>  
 <!--实体类的映射文件，在dao.mapper包下创建完Mapper文件后在此追加即可-->  
 <mappers>  
 <mapper resource="dao/mapper/DeptMapper.xml"/>  
 <mapper resource="dao/mapper/EmployeeMapper.xml"/>  
 </mappers>  
</configuration>

注意配置文件中的节点是有先后顺序的,如果写错了会 看到提示:



### 添加log4j日志文件

log4j.properties用于打印查看日志，方便我们查看sql语句的执行情况

完整的log4j.properties如下：

# For all other servers: Comment out the Log4J listener in web.xml to activate Log4J.  
log4j.rootLogger=DEBUG, stdout  
  
log4j.appender.stdout=org.apache.log4j.ConsoleAppender  
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout  
log4j.appender.stdout.layout.ConversionPattern=%d %p [%c] - %m%n  
  
#begin  
#for normal test,delete when online  
log4j.logger.com.ibatis=DEBUG  
log4j.logger.com.ibatis.common.jdbc.SimpleDataSource=DEBUG  
log4j.logger.com.ibatis.common.jdbc.ScriptRunner=DEBUG  
log4j.logger.com.ibatis.sqlmap.engine.impl.SqlMapClientDelegate=DEBUG  
  
log4j.logger.java.sql.Connection=DEBUG  
log4j.logger.java.sql.Statement=DEBUG  
log4j.logger.java.sql.PreparedStatement=DEBUG  
log4j.logger.java.sql.ResultSet=DEBUG  
#end

## 使用Mybatis进行增删改

使用Mybatis操作数据库，需要将SQL写在Mapper映射文件中。sql语句封装到标签中，对应增删改查的操作，有四种标签

<insert >用于执行insert插入数据

<delete>用于执行delete删除数据

<update>用于执行update修改数据

<select>用于执行select查询数据

每个标签都有一个id属性，在同一个Mapper文件中，每条语句的id都是唯一的，在执行语句的时候，通过namespace.id来调用语句。(详见2.1 id的说明和测试代码)

sql语句中需要传入参数的时候，通过parameterType指定参数的类型(详见2.1)，可以是基本数据类型，也可以是封装好的类或集合。通过#{属性}或${属性}来得到参数值(详见5.2)。

### 添加数据

以DEPT表为例。所有的语句都写在XXMapper.xml的<mapper>节点中。

在Mybatis中，添加数据使用insert标签：

<!--<![CDATA[insert标签用于执行添加操作]]> -->  
<insert id="addDept" parameterType="Dept" keyProperty="id" useGeneratedKeys="true">  
 insert into DEPT(NAME) values(#{name})  
</insert>

此标签对应生成的语句是：

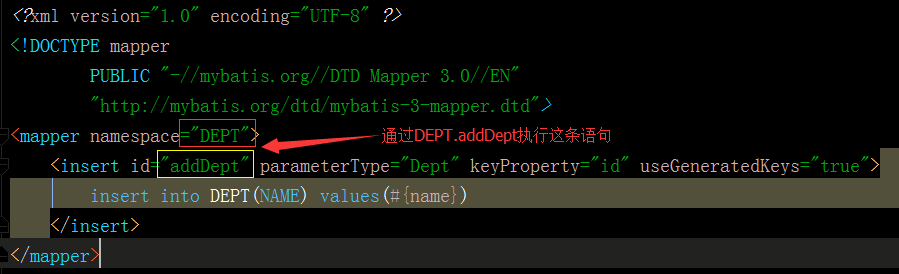
insert into DEPT (NAME) values (?)

parameterType="Dept"指定了调用语句时，传入的参数是Dept类型的，#{name}会获取到Dept对象中的name属性值，并在执行时替换掉语句中的?占位。

insert标签属性说明：

1. id

在同一个Mapper文件中，每条语句的id都是唯一的，在执行语句的时候，通过namespace.id来调用语句，如图：

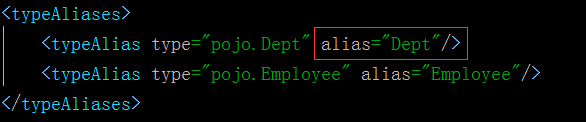


当前文件DeptMapper.xml的namespace是DEPT，语句的id是addDept。要执行这条语句的时候，需要通过DEPT.addDept来找到它

如测试代码中的session.insert("DEPT.addDept", dept);

1. parameterType

是指参数类型，sql语句执行的时候需要绑定参数，通常我们将参数封装到实体类中。parameterType的值可以是类的全路径如pojo.Dept，也可以是mybatis-config.xml文件中typeAlias中配置的别名alias，如1.5小节中我们在配置文件中给实体类指定了别名，将pojo.Dept简写成Dept：



1. keyProperty

仅对 insert有用，标记一个属性， MyBatis 会通过 getGeneratedKeys或者通过 insert 语句的 selectKey 子元素设置它的值。默认：不设置。 当添加完数据后，如果想获得到刚刚添加的这条数据的 主键，同时设置keyProperty和useGeneratedKeys即可。

1. useGeneratedKeys

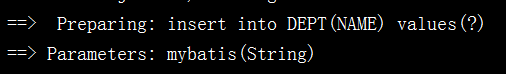
仅对insert语句有用，告 诉 MyBatis 使 用 JDBC 的getGeneratedKeys 方法来取出由数据（比如：像 MySQL 和 SQLServer 这样的数据库管理系统的自动递增字段）内部生成的主键。默认值： false。

测试代码：

@Test  
public void testInsert() {  
 String resource = "mybatis-config.xml";//配置文件名  
 Reader reader = null; //读取配置文件的工具  
 SqlSession session = null;  
 try {  
 //使用MyBatis提供的Resources类加载mybatis的配置文件  
 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  
 session = sessionFactory.openSession();  
 //创建实体对象，将它insert到数据库中  
 Dept dept = new Dept();  
 dept.setName("mybatis");  
 //使用session的insert方法执行添加操作  
 //第一个参数是要Mapper文件的namespace+.+语句的id  
 //第二个参数的类型要和语句的parameterType一致  
 session.insert("DEPT.addDept", dept);  
 session.commit();//增删改要提交事务  
 //insert标签中设置keyProperty和useGeneratedKeys后才能看到值，否则是null  
 System.***out***.println(dept.getId());  
 } catch (Exception ex) {  
 ex.printStackTrace();

session.rollback();//有异常事务回滚  
 } finally {//关闭资源  
 if (session != null) {  
 session.close();  
 }  
 if (reader != null) {  
 try {  
 reader.close();  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
}

运行测试代码，可以看到控制台上打印的语句：



为了简化代码，我们把打开和关闭资源的代码封装到@Before和@After中：

String resource = "mybatis-config.xml";  
Reader reader = null;  
SqlSession session = null;  
SqlSessionFactory sessionFactory = null;  
  
@Before  
public void openSession() throws Exception {  
 reader = Resources.getResourceAsReader(resource);  
 sessionFactory = new SqlSessionFactoryBuilder().build(reader);  
 session = sessionFactory.openSession();  
}  
  
@After  
public void closeSession() {  
 if (session != null) {  
 session.close();  
 }  
 if (reader != null) {  
 try {  
 reader.close();  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 }  
}

简化后的testInsert方法如下：

@Test  
public void testInsert() {  
 try {  
 Dept dept = new Dept();  
 dept.setName("mybatis");  
 session.insert("DEPT.addDept", dept);  
 session.commit();  
 } catch (Exception ex) {  
 ex.printStackTrace();  
 session.rollback();  
 }  
}

### 修改数据

修改数据使用的是update标签：

<update id="updateDept" parameterType="Dept">  
 update DEPT  
 set NAME=#{name}  
 where ID=#{id}  
</update>

对应生成的语句是：

update DEPT set NAME=? where ID=?

测试代码：

@Test  
public void testUpdate() {  
 try {  
 Dept dept = new Dept();  
 dept.setId(26);  
 dept.setName("update");  
 //session的update方法用于更新数据  
 session.update("DEPT.updateDept", dept);  
 session.commit();//提交事务  
 } catch (Exception e) {  
 e.printStackTrace();  
 session.rollback();//事务回滚  
 }  
}

### 删除数据

<delete id="deleteDept" parameterType="int">  
 delete from DEPT where ID=#{id}  
</delete>

对应生成的语句是：

delete from DEPT where ID=?

delete根据主键查询，Dept主键是Integer的。上面的语句中parameterType=”int”，其中int是Integer的别名。Mybatis默认的对基本数据类型都有起了别名。基本数据类型的参数，直接写类型或者类名小写即可。比如java.lang.Integer别名有Integer、integer、int虽然不是对象类型，但是int可以自动转化成Integer。

语句中使用了#{id}获取具体的参数值，但是int类型中并没有一个叫id的属性。可是测试发现语句仍然能执行成功。所以在parameterType=”基本数据类型”的时候，可以通过#{任意名称}获得参数值。这里将#{id}写成#{abc}也可以。

测试代码：

@Test  
public void testDelete() {  
 try {  
 //session的delete用于删除数据  
 session.delete("DEPT.deleteDept", 26);  
 session.commit();  
 } catch (Exception ex) {  
 session.rollback();  
 ex.printStackTrace();  
 }  
}

## 使用Mybatis进行查询

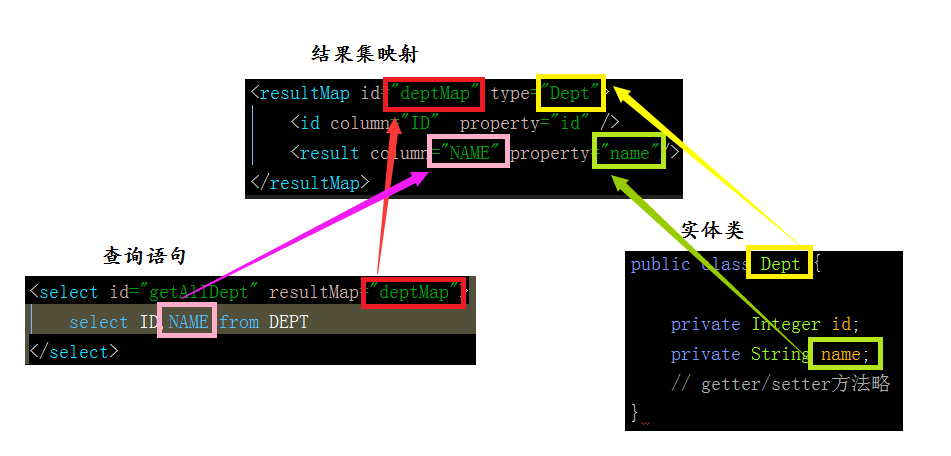
### 查询结果的映射

Mybatis通过<select>进行查询。通过<resultMap>将查询结果封装成实体类：

<resultMap id="deptMap" type="Dept">  
 <!--<![CDATA[<id>是主键标签]]> -->  
 <!--<![CDATA[<property是类中的属性名，column是sql语句中查询出来的字段>]]> -->  
 <id property="id" column="ID"/>  
 <!--<![CDATA[<result>是主键标签]]> -->  
 <result property="name" column="NAME"/>  
</resultMap>  
  
<select id="getAllDept" resultMap="deptMap">  
 select ID,NAME from DEPT  
</select>

<resultMap>用于封装结果集。一个Mapper中可以有多个<resultMap>，每个<resultMap>的id是唯一的。type用于指定结果集对应的实体类类型。type="Dept"即将<resultMap>中所有的属性都对应到Dept类中。

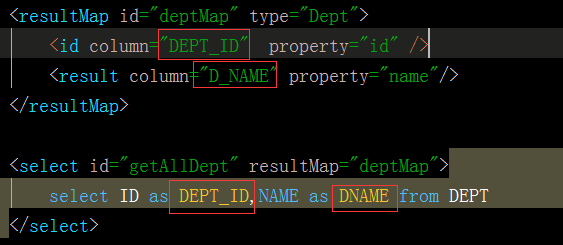
对应关系如下图所示：



我们可以这样理解，一个查询语句，从查询出结果，到封装成实体类，经历如下步骤：

1. 当使用<select>查询出结果后，会根据<select>中resultMap=”deptMap”找到id=”deptMap”的<resultMap>标签
2. 根据<resultMap>的type=”Dept”找到Dept对应的类，这里Dept是别名，也可以写成全路径pojo.Dept。
3. <resultMap>中的<result>标签，将查询结果中的字段与实体类中的属性对应起来，property=”name”是找到Dept类中的name属性，column=”NAME”中NAME是SQL语句中NAME字段对应的值，通过setName方法将结果集绑定到name属性上。

需要注意的是column=”NAME”中的NAME不是数据库中的字段名，而是select语句中查询出来的字段别名，因为我们可以用as去给字段起别名。如上面改变上面的语句，为查询结果起别名，对应的column也要改变：



测试代码：

@Test  
public void testSelectAll() {  
 //当查询结果有多个的时候，使用selectList  
 List<Dept> deptList = session.selectList("DEPT.getAllDept");  
 for (Dept dept : deptList) {  
 System.***out***.println(dept.getName());  
 }  
}

session.selectList适用于查询结果是一个集合的情况，如果实现按主键查询，则使用session.selectOne方法。如按主键查询：

<select id="getDeptById" parameterType="int" resultMap="deptMap">  
 select ID,NAME from DEPT where ID=#{id}  
</select>

测试代码：

@Test  
public void testSelectOne() {  
 Dept dept = session.selectOne("DEPT.getDeptById", 21);  
 System.***out***.println(dept.getName());  
}

resultMap还支持继承：

<resultMap id="deptMap" type="Dept">  
 <id column="ID" property="id" />  
 <result column="NAME" property="name"/>  
</resultMap>  
  
<resultMap id="deptMapChild" type="Dept" extends="deptMap">  
 <!--这里写新的<result>-->  
</resultMap>

当我们又多个<resultMap>的时候，有些属性<result>的配置可以复用，就可以使用extends进行继承，上面的示例中，deptMapChild可以使用deptMap中的配置，具体的示例，请参照3.2.2中的deptMapWithEmployee

### 关联查询

#### 一对一、多对一

本节的操作使用EmployeeMapper.xml，请提前创建文件并将EmployeeMapper.xml的namespace设置为”EMPLOYEE”。

Mybatis中一对一和多对一的配置一样。以员工EMPLOYEE为例。多个员工同属于一个部门，所以员工对部门是多对一。

员工实体类：

package pojo;  
  
import java.util.Date;

public class Employee {  
 private Integer id;  
 private String name;  
 private Dept dept;  
 private String job;  
 private Float salary;  
 private Date hireDate;  
 //getter/setter方法略  
}

如果查询出员工的时候要查询出部门名，可以编写如下语句：

<select id="getEmployeeWithDept" resultMap="employeeMap">  
 select E.ID, E.NAME ,E.JOB, E.SALARY, E.HIRE\_DATE, E.DEPT\_ID, D.NAME DEPT\_NAME  
 from EMPLOYEE E  
 inner join DEPT D ON E.DEPT\_ID=D.ID  
</select>

对应的resultMap配置如下：

<resultMap id="employeeMap" type="Employee">  
 <id property="id" column="ID"/>  
 <result property="name" column="NAME"/>  
 <result property="job" column="JOB"/>  
 <result property="salary" column="SALARY"/>  
 <result property="hireDate" column="HIRE\_DATE"/>  
 <result property="dept.id" column="DEPT\_ID"/>  
 <result property="dept.name" column="DEPT\_NAME"/>  
</resultMap>

通过dept.id和dept.name这种级联的方式给Employee类中dept这个属性的id和name赋值。

测试代码：

@Test  
public void testManyToOne() {  
 List<Employee> empList = session.selectList("EMPLOYEE.getEmployeeWithDept");  
 for (Employee employee : empList) {  
 System.***out***.println(employee.getName() + "的部门："

+ employee.getDept().getName());  
 }  
}

结果集employeeMap也可以使用association进行优化：

<resultMap id="employeeMap2" type="Employee">  
 <id property="id" column="ID"/>  
 <result property="name" column="NAME"/>  
 <result property="job" column="JOB"/>  
 <result property="salary" column="SALARY"/>  
 <result property="hireDate" column="HIRE\_DATE"/>  
 <!--使用association将属于Dept的属性进行封装，不再使用dept.id-->  
 <association property="dept" javaType="Dept">  
 <result property="id" column="DEPT\_ID"/>  
 <result property="name" column="DEPT\_NAME"/>  
 </association>  
</resultMap>

因为我们在DeptMapper.xml已经为Dept配置了结果集，所以上面的<association>可以通过resultMap引用其他的结果集：

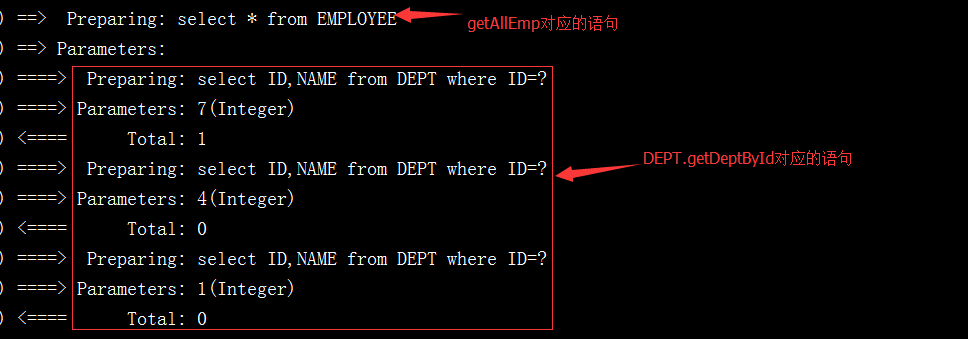
<!--引用DEPT.deptMap-->  
<association property="dept" resultMap="DEPT.deptMap">  
</association>

<association>标签还有一个select属性，可以通过一个查询语句得到结果集并进行映射。如：

<resultMap id="employeeMap3" type="Employee">  
 <id property="id" column="ID"/>  
 <result property="name" column="NAME"/>  
 <result property="job" column="JOB"/>  
 <result property="salary" column="SALARY"/>  
 <result property="hireDate" column="HIRE\_DATE"/>  
 <!--通过select调用DEPT.getDeptById-->  
 <association property="dept" column="DEPT\_ID" select="DEPT.getDeptById" >  
 </association>  
</resultMap>

<select id="getAllEmp" resultMap="employeeMap3">  
 select *\** from EMPLOYEE  
</select>

执行getAllEmp这个语句，我们只查询了EMPLOYEE表，没有查询DEPT表，在结果映射到employeeMap3的时候，associstion通过select指定的语句，去调用了DEPT.getDeptById进行查询，并将结果集封装到dept中。控制台打印sql如下：



可以看到，执行的流程是先查询出所有的EMPLOYEE，然后遍历结果集，根据每条查询结果的DEPT\_ID去查询了DEPT表。直接使用join会执行一条语句，而在<associstion>中使用select，会执行n+1条。n是主表的查询结果集的条数，效率较低。

#### 一对多、多对多

一对多、多对多的配置都是通过<collection>进行的。以一对多为例，一个部门DEPT有多个员工EMPLOYEE。

Dept实体类中添加Employee集合：

package pojo;  
  
import java.util.List;  
  
public class Dept {  
  
 private Integer id;  
 private String name;  
 private List<Employee> employees;  
 // getter/setter方法略

}

如果查询部门的时候要查询出部门所有的员工，sql语句为：

<select id="getDeptWith" resultMap="deptMapWithEmployee">  
 select D.ID, D.NAME, E.ID EMP\_ID, E.NAME EMP\_NAME  
 from DEPT D  
 left join EMPLOYEE E ON E.DEPT\_ID=D.ID  
</select>

为了简化操作，我们只联合查询EMPLOYEE的ID和NAME两个属性。

对应的resultMap映射为：

<!--extends实现继承id=继承deptMap的resultMap-->  
<resultMap id="deptMapWithEmployee" type="Dept" extends="deptMap">  
 <!--collection用于映射一个集合-->  
 <collection property="employees" ofType="Employee" >  
 <id property="id" column="EMP\_ID"/>  
 <result property="name" column="EMP\_NAME"/>  
 </collection>  
</resultMap>

<collection>也支持select属性：

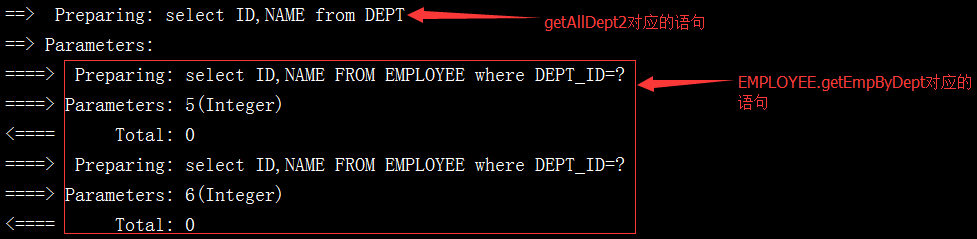
<resultMap id="deptMapWithEmployee2" type="Dept" extends="deptMap">  
 <!--collection用于映射一个集合-->  
 <collection property="employees" ofType="Employee"

select="EMPLOYEE.getEmpByDept" column="ID">  
 </collection>  
</resultMap>  
<select id="getAllDept2" resultMap="deptMapWithEmployee2">  
 select ID,NAME from DEPT  
</select>

<collection>的select调用了EMPLOYEE.getEmpByDept，需要提前在EmployeeMapper.xml中写好对应的语句：

<select id="getEmpByDept" resultMap="employeeMap" parameterType="int">  
 select ID,NAME FROM EMPLOYEE where DEPT\_ID=#{deptid}  
</select>

执行getAllDept2，可以看到控制台上的输出：



和<associstion>一样，直接使用join会执行一条语句，而在<collection>中使用select，会执行n+1条。n是主表的查询结果集的条数，效率较低

## 动态SQL及SQL片段

### if

<if test="条件">拼接的sql</if>

Mybatis中的if标签用于拼接sql。例如：

<select id="getEmployees" parameterType="Employee" resultMap="employeeMap">  
 select *\** from EMPLOYEE  
 where ID=#{id} and NAME like CONCAT('%', #{name},'%')  
</select>

如果id和name两个条件要动态组合，即有时需要根据id查询，有时需要根据name查询，有时要id和name一起插查询。如果写三个语句会大大降低开发效率，增加代码冗余。如果需要动态组合的条件更多，那么开发人员将要写无数种情况的语句。

这时，可以使用if标签进行动态sql的生成：

<select id="getEmployees" parameterType="Employee" resultMap="employeeMap">  
 select *\** from EMPLOYEE  
 where  
 <if test="id!=null"> ID=#{id}</if>  
 <if test="name!=null and name !=''"> and NAME like CONCAT('%', #{name},'%')</if>  
</select>

当test的条件满足的时候，<if>标签中包含的语句会拼接到前面的sql中。

·判断数值类型、日期类型不为空，可以直接使用“属性!=null”来判断。

·判断字符串类型不为空，除了判断不为null意外，还要判断不等于空字符串。

上面的例子中，如果id和name都不为空，则生成的语句是：

select \* from EMPLOYEE where ID=? and NAME like CONCAT('%', #{name},'%')

如果id不为空，name为空，则生成的语句是：

select \* from EMPLOYEE where ID=?

但如果id和name都为空，生成的语句就是：

select \* from EMPLOYEE where

这是一个不完整的sql。这时就需要用到where或trim标签。

### where

where标签可以根据条件，决定是否拼接where关键字,并且能动态处理近邻where后的and或or关键字：

<select id="getEmployees" parameterType="Employee" resultMap="employeeMap">  
 select *\** from EMPLOYEE  
 <where>  
 <if test="id!=null">ID=#{id}</if>  
 <if test="name!=null and name !=''">and NAME like CONCAT('%', #{name},'%')</if>  
 </where>  
</select>

如果id和name都为空，生成语句是：

select \* from EMPLOYEE

如果id为空，name有值，where会自动把and NAME like CONCAT('%', #{name},'%')前面的and去掉，生成的语句是：

select \* from EMPLOYEE where NAME like CONCAT('%', ?,'%')

### set

set标签用于update语句中。

<update id="updateEmployee" parameterType="Employee">  
 update EMPLOYEE  
 set  
 <if test="name!=null and name !=''">NAME =#{name},</if>  
 <if test="salary!=null">SALARY =#{salary}</if>  
 where ID=#{id}  
</update>

上面的语句，当name!=null而salary==null的时候，生成的语句是：

update EMPLOYEE set name=?, where ID=?

执行的时候会报错，原因是where前面多了一个逗号。

使用set标签：

<update id="updateEmployee" parameterType="Employee">  
 update EMPLOYEE  
 <set>  
 <if test="name!=null and name !=''">NAME =#{name},</if>  
 <if test="salary!=null">SALARY =#{salary},</if>  
 </set>  
 where ID=#{id}  
</update>

当name为空而salary不为空的时候，生成的语句是：

update EMPLOYEE set salary=? where ID=?

set标签可以动态设置set关键字，并且智能消除无关的逗号。

但是当set中所有的if条件都不满足的时候(name和salary都为空)，生成的语句是：

update EMPLOYEE set where ID=?

这是个错误的语句，程序会报错。

从业务逻辑上讲，需要执行update更新的时候却一个属性都不赋值，本身就是不对的。所以可以在程序中捕捉到异常并提示用户更新失败。如果不想捕捉异常也不想判断所有的属性，可以加上ID字段：

<update id="updateEmployee" parameterType="Employee">  
 update EMPLOYEE  
 <set>  
 <if test="name!=null and name !=''">NAME =#{name},</if>  
 <if test="salary!=null">SALARY =#{salary},</if>  
 </set>  
 ID=#{id}  
 where ID=#{id}  
</update>

### trim

trim标签可以用来替代where和set标签。

使用trim代替where：

<select id="getEmployees2" parameterType="Employee" resultMap="employeeMap">  
 select *\** from EMPLOYEE  
 <trim prefix="WHERE" prefixOverrides="AND |OR ">  
 <if test="id!=null">ID=#{id}</if>  
 <if test="name!=null and name !=''">and NAME like CONCAT('%', #{name},'%')</if>  
 </trim>  
</select>

prefix=”WHERE”是拼接where关键字，prefixOverrides=”AND | OR ”是如果where后紧邻and或or的时候智能的去掉。注意AND和OR后面都有个空格。

使用trim代替set：

<update id="updateEmployee2" parameterType="Employee">  
 update EMPLOYEE  
 <trim prefix="SET" suffixOverrides=",">  
 <if test="name!=null and name !=''">NAME =#{name},</if>  
 <if test="salary!=null">SALARY =#{salary},</if>  
 </trim>  
 where ID=#{id}  
</update>

### choose,when,otherwise

choose，when，otherwise三个标签要结合到一起使用：

<select id="getEmployees3" parameterType="Employee" resultMap="employeeMap">  
 select *\** from EMPLOYEE  
 <trim prefix="WHERE" prefixOverrides="AND |OR ">  
 <choose>  
 <when test="id!=null">ID=#{id}</when >  
 <when test="name!=null and name !=''"> NAME =#{name}</when >

<when test="salary!=null">SALARY=#{salary}</when >  
 <otherwise > DEPT\_ID=1</otherwise>  
 </choose>  
 </trim>  
</select>

choose-when和if标签的区别是，一个choose中有多个when，从上至下，如果第一个when条件满足，则只拼接这个when中的语句，后面的when和otherwise都跳过不判断了也不拼接。如果第一个when不满足，就判断第二个，依次类推。如果所有的when都不满足，就拼接otherwise中的语句。

即多个if标签，彼此之间都是独立的，互不影响。而choose-when标签中，多个when只能满足一个，从上至下判断，找到第一个匹配的条件后，剩余的when都不再进行匹配。

### foreach

foreach通常用在构建IN的查询条件：

<select id="getEmployees4" parameterType="List" resultMap="employeeMap">  
 select *\** from EMPLOYEE  
 where ID IN  
 <foreach collection="list" index="index" item="item" open="(" separator="," close=")">  
 #{item}  
 </foreach>  
</select>

foreach用于遍历collection中指定的集合open和close指定前后拼接的字符，separator是集合元素间插入的符号。

上面的语句如果传入一个包含3个元素的List，生成的语句是：

select \* from EMPLOYEE where ID IN ( ? , ? , ? )

·如果parameterType类型是List，则collection=list

·如果parameterType类型是ArrayList（数组）, 则collection=array

·如果parameterType是自定义的实体类，而类中有个集合属性, 则collection=集合属性名

### bind

bind可以创建一个变量，并绑定到上下文，如

<select id="getEmployees5" parameterType="Employee" resultMap="employeeMap">  
 <bind name="pattern" value="'%' + name+ '%'" />  
 select *\** from EMPLOYEE  
 where  
 NAME like #{pattern}  
</select>

上面的代码，bind标签将name的前后拼接了%并赋值给了pattern

如果传入的name=张，生成的语句为：

select \* from EMPLOYEE where NAME like ?

?的值会替换为“%张%”

### sql,include

sql标签可以把可复用的语句提取出来，通过include语句引用它。

比如前面的示例中，很多地方都用到了select \* from EMPLOYEE，我们就可以把它提取出来：

<sql id="selectEmployee">  
 select *\** from EMPLOYEE  
</sql>  
<select id="getEmployees6" parameterType="int" resultMap="employeeMap">  
 <include refid="selectEmployee"/>  
 where ID =#{id}  
</select>

## 补充

### \_parameter

<select id="getDeptById" parameterType="int" resultMap="deptMap">  
 select ID,NAME from DEPT where ID=#{id}  
</select>

在<select>标签中，parameterType传递基本数据类型的时候，可以给参数取任意的名字，如此处可以用#{id}，#{abc}，#{xxx}都可以。

但是当使用<if>判断test条件的时候，就需要使用\_parameter作为参数名，如：

<select id="getDeptById" parameterType="int" resultMap="deptMap">  
 select ID,NAME from DEPT  
 <if test="\_parameter!=null"> where ID=#{id}</if>  
</select>

如果写成下面的形式，就会报错：

<select id="getDeptById" parameterType="int" resultMap="deptMap">  
 select ID,NAME from DEPT  
 <if test="id!=null"> where ID=#{id}</if>  
</select>

org.apache.ibatis.exceptions.PersistenceException:

### Error querying database. Cause: org.apache.ibatis.reflection.ReflectionException: There is no getter for property named 'id' in 'class java.lang.Integer'

### Cause: org.apache.ibatis.reflection.ReflectionException: There is no getter for property named 'id' in 'class java.lang.Integer'

### #{}和${}

#{}是将传入的值当做字符串的形式，并且有预编译功能。

${}是将传入的数据直接显示生成sql语句。假设现在参数name的值是abc

select \* from DEPT where name=#{name} 会预编译成select \* from DEPT where ID=?。执行的时候给?赋值，实际执行的语句是：select \* from DEPT where name=’abc’。注意’abc’外面有个引号，它是字符串形式。

select \* from DEPT where name=${name}没有预编译功能，直接生成：

select \* from DEPT where name=abc。

#方式能够很大程度防止sql注入，$方式无法防止Sql注入。$方式一般用于传入数据库对象，例如传入表名、列名。 一般能用#的就别用$。

如果MyBatis排序时需要动态改变order by 的排序字段，使用order by 动态参数时需要注意，用$而不是#。ORDER BY ${columnName}。

## 分页插件PageHelper



mybatis中首先要在配置文件中配置一些东西，然后根据这些配置去创建一个会话工厂，再根据会话工厂创建会话，会话发出操作数据库的sql语句，然后通过执行器操作数据，再使用mappedStatement对数据进行封装，这就是整个mybatis框架的执行情况。mybatis的插件主要作用在Executor执行器与mappedeStatement之间，也就是说mybatis可以在插件中获得要执行的sql语句，在sql语句中添加limit语句，然后再去对sql进行封装，从而可以实现分页处理

使用PageHelper需要引入jar包。

<dependency>  
 <groupId>com.github.pagehelper</groupId>  
 <artifactId>pagehelper</artifactId>  
 <version>4.2.1</version>  
</dependency>

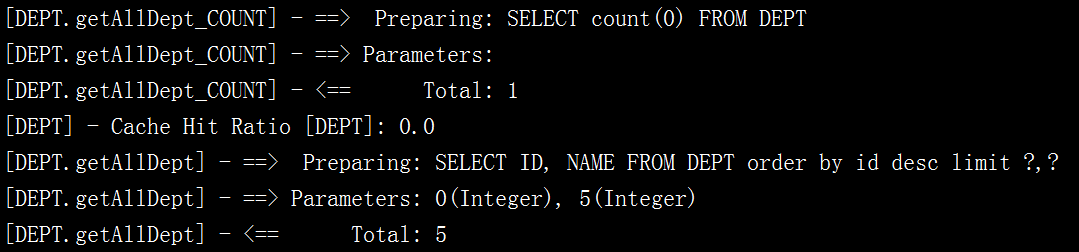
在mybatis-config.xml中添加插件的配置：

<!--plugins在settings之后，environments之前-->  
<plugins>  
 <!-- PageHelper4.1.1 -->  
 <plugin interceptor="com.github.pagehelper.PageHelper">  
 <property name="dialect" value="mysql"/><!--数据库方言-->  
 <!-- 3.3.0版本可用 - 分页参数合理化，默认false禁用 -->  
 <!-- 启用合理化时，如果pageNum<1会查询第一页，如果pageNum>pages会查询最后一页 -->  
 <!-- 禁用合理化时，如果pageNum<1或pageNum>pages会返回空数据 -->  
 <property name="reasonable" value="false"/>  
 </plugin>  
</plugins>

测试代码：

**@Test  
public void** testPageHelper() **{  
 //这句一定要写在sql执行之前,只对紧跟的第一条语句起作用  
 PageHelper**.startPage(**1**, **5**,**"id desc"**);  
 List**<Dept> deptList = session**.**selectList**(**"DEPT.getAllDept"**);  
 **PageInfo<Dept> page = new** PageInfo**<>**(**deptList**);  
 **System**.***out***.println(**"总页数：" + page**.getPages());  
 **System**.***out***.println(**"页码：" + page**.getPageNum());  
 **System**.***out***.println(**"总条数："+page**.getTotal());  
 **System**.***out***.println(**deptList**.**size**());  
 **for** (**Dept dept : deptList**) **{  
 System**.***out***.println(**dept**.getName());  
 **}  
}**

控制台打印语句：



## Mybatis注解(选学)

注解只能实现简单的功能，复杂的语句还要在xml中进行。

**package dao.mapper**;  
  
**import org.apache.ibatis.annotations.\***;  
**import pojo.Dept**;  
  
**public interface** DeptMapper **{  
 @Insert**(**"insert into DEPT(NAME) values(#{name})"**)  
 **@Options**(**useGeneratedKeys=true**, **keyProperty="id"**, **keyColumn="ID"**)  
 **void** addDept(**Dept dept**);  
  
 **@Update**(**"update DEPT set NAME=#{name} where ID=#{id}"**)  
 **void** updateDept(**Dept dept**);  
  
 **@Delete**(**"delete from DEPT where ID=#{id}"**)  
 **void** deleteDept(**int id**);  
  
 **@Select**(**"select ID,NAME from DEPT where ID=#{id}"**)  
 **Dept** getDeptById(**Integer id**);  
**}**

在mybatis-config.xml中<mappers>节点加入：

<mapper class="dao.mapper.DeptMapper"/>

测试代码：

**@Test  
public void** testAnnotations() **{** DeptMapper **mapper = session**.**getMapper**(DeptMapper.**class**);  
 **Dept dept1 = new** Dept();  
 **dept1**.setName(**"anno"**);  
 **mapper**.**addDept**(**dept1**);  
 **session**.**commit**();

}

## 缓存(选学，工作后进阶)

### 一级缓存

在对数据库的一次会话中，我们有可能会反复地执行完全相同的查询语句，如果不采取一些措施的话，每一次查询都会查询一次数据库，而我们在极短的时间内做了完全相同的查询，那么它们的结果极有可能完全相同，由于查询一次数据库的代价很大，这有可能造成很大的资源浪费。

MyBatis会在一次会话的表示一个SqlSession对象中创建一个本地缓存(local cache)，对于每一次查询，都会尝试根据查询的条件去本地缓存中查找是否在缓存中，如果在缓存中，就直接从缓存中取出，然后返回给用户；否则，从数据库读取数据，将查询结果存入缓存并返回给用户。

一级缓存是Mybatis自带的缓存。

如下面的测试代码：

@Test  
public void testFirstCache() {  
 //连续查了两次id=21的数据  
 //dept1从数据库中查询，控制台会打印sql语句  
 Dept dept1 = session.selectOne("DEPT.getDeptById", 21);  
 //dept2从缓存中查询，控制台上没有sql输出  
 Dept dept2 = session.selectOne("DEPT.getDeptById", 21);  
 System.***out***.println(dept1.getName());  
 System.***out***.println(dept2.getName());  
}

这段代码执行一次查询。

同一个SqlSession中，执行了任何一个update()、insert()、delete()或commit()后都会清空一级缓存。下面的测试代码会查询两次：

@Test  
public void testFirstCache() {  
 //连续查了两次id=22的数据  
 //dept1从数据库中查询，控制台会打印sql语句,放入缓存中  
 Dept dept1 = session.selectOne("DEPT.getDeptById", 22);  
 //使用进行增删改任何操作，或comimit()后都会清空缓存  
 session.delete("DEPT.deleteDept", 27);  
 session.commit();  
 //因为前面进行了删除,所以缓存清空了，这里又会重新查询数据库  
 Dept dept2 = session.selectOne("DEPT.getDeptById", 22);  
 System.***out***.println(dept1.getName());  
 System.***out***.println(dept2.getName());  
}

### 二级缓存

二级缓存需要用户去设置。二级缓存比一级缓存范围更大，一级缓存是一个SqlSession范围内的，不同的SqlSession之间不共享。而二级缓存可以实现多个SqlSession之间共享对同一个Mapper的缓存。

二级缓存需要在mybatis-config.xml中设置开启：

<settings>  
 <!--开启二级缓存-->  
 <setting name="cacheEnabled" value="true"/>  
</settings>

注意<settings>节点要写在typeAliases之前，否则会报错。

在需要开始二级缓存的Mapper中加入<cache>节点开启缓存。如在DeptMapper.xml中开启缓存：

<mapper namespace="DEPT">  
 <!--开启本Mapper的二级缓存-->  
 <cache></cache>

二级缓存要求放入缓存的pojo对象都必须实现java.io.Serializable接口：

public class Dept implements Serializable{

测试代码：

@Test  
public void testSecondCache() {  
 //使用session1查询dept1从数据库中查询，控制台会打印sql语句  
 SqlSession session1=sessionFactory.openSession();  
 Dept dept1 = session1.selectOne("DEPT.getDeptById", 28);  
 session1.close();//关闭session1，此时将dept1写入二级缓存  
 //开启一个新的session  
 SqlSession session2=sessionFactory.openSession();  
 //因为session1已经将id=28的对象放到二级缓存中了，所以这里从缓存中取  
 Dept dept2 = session2.selectOne("DEPT.getDeptById", 28);  
 System.***out***.println(dept1.getName());  
 System.***out***.println(dept2.getName());  
 session2.close();  
}

<select>有useCache属性，默认是true，使用二级缓存。设置成false禁用二级缓存。

增删改查的statement标签都有一个**flushCache属性，默认是true，即执行后刷新缓存。一般不去改它。避免数据出现脏读。**

<cache  eviction='FIFO' flushInterval='60000'  size='512' readOnly='true'/>

flushInterval（刷新间隔）可以被设置为任意的正整数，而且它们代表一个合理的毫秒形式的时间段。默认情况是不设置，也就是没有刷新间隔，缓存仅仅调用语句时刷新。

size（引用数目）可以被设置为任意正整数，要记住你缓存的对象数目和你运行环境的可用内存资源数目。默认值是1024。

readOnly（只读）属性可以被设置为true或false。只读的缓存会给所有调用者返回缓存对象的相同实例。因此这些对象不能被修改。这提供了很重要的性能优势。可读写的缓存会返回缓存对象的拷贝（通过序列化）。这会慢一些，但是安全，因此默认是false。

1.      LRU – 最近最少使用的:移除最长时间不被使用的对象。

2.      FIFO – 先进先出:按对象进入缓存的顺序来移除它们。

3.      SOFT – 软引用:移除基于垃圾回收器状态和软引用规则的对象。

4.      WEAK – 弱引用:更积极地移除基于垃圾收集器状态和弱引用规则的对

### 整合第三方缓存ehcache

ehcache缓存数据可以在内存和磁盘上缓存数据，无需担心容量问题。

Mybatis与ehcache结合使用，可以实现多个SqlSession对同一个Mapper的缓存共享。

#### 导入jar包

使用ehcache需要导入jar包，在pom.xml中加入如下配置：

<dependency>  
 <groupId>net.sf.ehcache</groupId>  
 <artifactId>ehcache-core</artifactId>  
 <version>2.4.4</version>  
</dependency>  
<dependency>  
 <groupId>org.mybatis.caches</groupId>  
 <artifactId>mybatis-ehcache</artifactId>  
 <version>1.1.0</version>  
</dependency>

#### 创建ehcache配置文件

在resources目录下创建ehcache.xml文件：

*<?*xml version="1.0" encoding="UTF-8"*?>*<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:noNamespaceSchemaLocation="../bin/ehcache.xsd">  
 <defaultCache overflowToDisk="true" eternal="false"/>  
 <!--硬盘上存放数据的位置-->  
 <diskStore path="E:/tmp"/>  
 <!--缓存的配置，与Mybatis整合时，name等于Mapper的namespace-->  
 <cache name="DEPT" overflowToDisk="true" eternal="false"  
 timeToIdleSeconds="300" timeToLiveSeconds="600" maxElementsInMemory="2"  
 maxElementsOnDisk="10" diskPersistent="true"

diskExpiryThreadIntervalSeconds="300" diskSpoolBufferSizeMB="100"

memoryStoreEvictionPolicy="LRU"/>  
</ehcache>

参数说明：

name：Cache的唯一标识，与Mybatis结合使用时，name是Mapper的namespace

maxElementsInMemory：内存中最大缓存对象数

maxElementsOnDisk：磁盘中最大缓存对象数，若是0表示无穷大

eternal：Element是否永久有效，一但设置了，timeout将不起作用

overflowToDisk：配置此属性，当内存中Element数量达到maxElementsInMemory时，Ehcache将会Element写到磁盘中

timeToIdleSeconds：设置Element在失效前的允许闲置时间。仅当element不是永久有效时使用，可选属性，默认值是0，也就是可闲置时间无穷大

timeToLiveSeconds：设置Element在失效前允许存活时间。最大时间介于创建时间和失效时间之间。仅当element不是永久有效时使用，默认是0.，也就是element存活时间无穷大

diskPersistent：是否缓存虚拟机重启期数据

diskExpiryThreadIntervalSeconds：磁盘失效线程运行时间间隔，默认是120秒

diskSpoolBufferSizeMB：这个参数设置DiskStore（磁盘缓存）的缓存区大小。默认是30MB。每个Cache都应该有自己的一个缓冲区

memoryStoreEvictionPolicy：当达到maxElementsInMemory限制时，Ehcache将会根据指定的策略去清理内存。默认策略是LRU（最近最少使用）。你可以设置为FIFO（先进先出）或是LFU（较少使用）。

#### 开启Mybatis二级缓存

在mybatis-config.xml中开启二级缓存：

<settings>  
 <!--开启二级缓存-->  
 <setting name="cacheEnabled" value="true"/>  
</settings>

#### 在Mapper中设置缓存

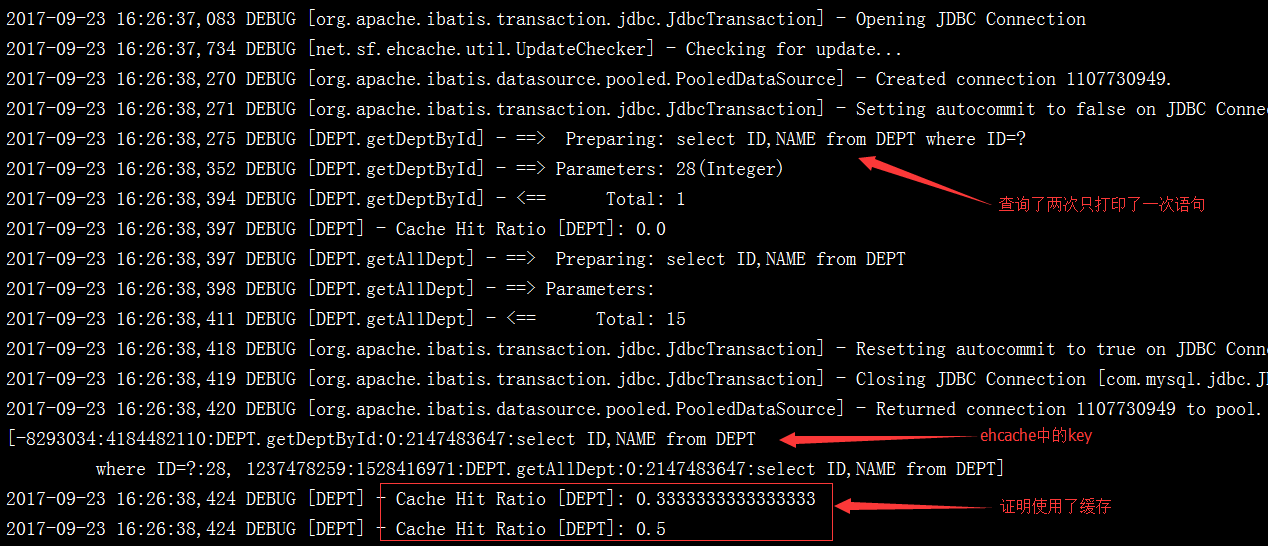
在DeptMapper.xml中开启缓存：

<mapper namespace="DEPT">  
 <!--开启本Mapper的二级缓存-->  
 <!--<cache ></cache>-->  
 <cache type="org.mybatis.caches.ehcache.EhcacheCache"/>

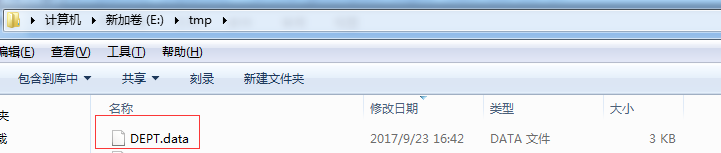
测试代码：

@Test  
public void testEhCache() throws InterruptedException {  
 //使用session1查询dept1从数据库中查询，控制台会打印sql语句  
 SqlSession session1 = sessionFactory.openSession();  
 Dept dept1 = session1.selectOne("DEPT.getDeptById", 28);  
 List<Dept> list1 = session1.selectList("DEPT.getAllDept");  
 session1.close();//关闭session1，此时将dept1写入缓存  
 //查看缓存是否生效  
 CacheManager cacheManager = CacheManager.getInstance();  
 if (cacheManager.cacheExists("DEPT")) {  
 Cache cache = cacheManager.getCache("DEPT");  
 //可以看到缓存中写入了数据  
 System.***out***.println(cache.getKeys());  
 }  
 //开启一个新的session  
 SqlSession session2 = sessionFactory.openSession();  
 //因为session1已经将id=28的对象放到二级缓存中了，所以这里从缓存中取  
 //使用session2执行和session1一样的语句时，不会打印sql  
 Dept dept2 = session2.selectOne("DEPT.getDeptById", 28);  
 List<Dept> list2 = session2.selectList("DEPT.getAllDept");  
 session2.close();  
  
}

运行结果：



去磁盘上查看，可以看到缓存文件：



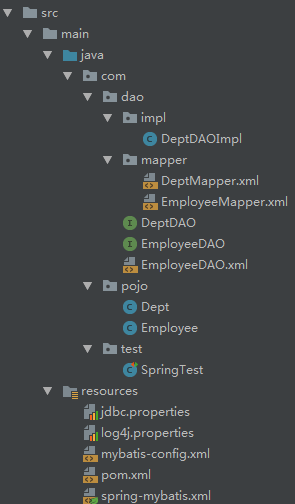
## Spring和Mybatis整合

### 创建工程

新建工程，导入所需jar包：

<dependencies>  
 <!-- mybatis核心包 -->  
 <dependency>  
 <groupId>org.mybatis</groupId>  
 <artifactId>mybatis</artifactId>  
 <version>3.4.1</version>  
 </dependency>  
 <!-- mysql驱动包 -->  
 <dependency>  
 <groupId>mysql</groupId>  
 <artifactId>mysql-connector-java</artifactId>  
 <version>5.1.29</version>  
 </dependency>  
 <!-- junit测试包 -->  
 <dependency>  
 <groupId>junit</groupId>  
 <artifactId>junit</artifactId>  
 <version>4.12</version>  
 </dependency>  
 <!--日志包-->  
 <dependency>  
 <groupId>org.slf4j</groupId>  
 <artifactId>slf4j-log4j12</artifactId>  
 <version>1.6.1</version>  
 </dependency>  
 <dependency>  
 <groupId>org.slf4j</groupId>  
 <artifactId>slf4j-api</artifactId>  
 <version>1.6.1</version>  
 </dependency>  
 <!--spring数据库-->  
 <dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-jdbc</artifactId>  
 <version>4.3.11.RELEASE</version>  
 </dependency>  
 <!--aop-->  
 <dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-aspects</artifactId>  
 <version>4.3.11.RELEASE</version>  
 </dependency>  
 <dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-context</artifactId>  
 <version>4.3.11.RELEASE</version>  
 </dependency>  
 <!--spring-mybatis整合包-->  
 <dependency>  
 <groupId>org.mybatis</groupId>  
 <artifactId>mybatis-spring</artifactId>  
 <version>1.3.1</version>  
 </dependency>  
</dependencies>  
<!--配置资源文件扫描，否则Mapper-->  
<build>  
 <resources>  
 <resource>  
 <directory>src/main/java</directory>  
 <includes>  
 <include>\*\*/\*.xml</include>  
 </includes>  
 <filtering>true</filtering>  
 </resource>  
 <resource>  
 <directory>src/main/resources</directory>  
 <includes>  
 <include>\*\*/\*.xml</include>  
 <include>\*\*/\*.properties</include>  
 </includes>  
 </resource>  
 </resources>  
</build>

工程完整目录如下：



jdbc.properties文件存放数据库连接信息：

jdbc.driver=com.mysql.jdbc.Driver  
jdbc.url=jdbc:mysql://localhost:3306/数据库?characterEncoding=utf-8  
jdbc.username=用户名  
jdbc.password=密码

log4j日志文件见1.6节

实体类Dept和Employee见1.3

映射文件DeptMapper.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"*>*  
<mapper namespace="com.dao.DeptDAO">  
 <insert id="addDept" parameterType="Dept" keyProperty="id" useGeneratedKeys="true" >  
 insert into DEPT(NAME) values(#{name})  
 </insert>  
</mapper>

接口DeptDAO

**package com.dao**;  
  
**import com.pojo.Dept**;  
  
**public interface** DeptDAO **{  
 void** addDept(**Dept dept**);  
**}**

### spring-mybatis整合

在spring与mybatis整合示例中，spring负责管理数据源、加载mybatis配置文件、mybatis的session和事务。

*<?*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:context="http://www.springframework.org/schema/context"  
 xsi:schemaLocation="  
 http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-4.2.xsd  
 http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-4.2.xsd">  
 <!--spring管理数据源-->  
 <context:property-placeholder location="classpath:jdbc.properties" ignore-unresolvable="true"/>  
 <bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">  
 <property name="driverClassName" value="${jdbc.driver}"/>  
 <property name="url" value="${jdbc.url}"/>  
 <property name="username" value="${jdbc.username}"/>  
 <property name="password" value="${jdbc.password}"/>  
 </bean>

</beans>

mybatis的配置文件中不再写Mapper映射文件的配置：

*<?*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>  
 <typeAliases>  
 <typeAlias type="com.pojo.Dept" alias="Dept"/>  
 <typeAlias type="com.pojo.Employee" alias="Employee"/>  
 </typeAliases>  
</configuration>

spring中加载mybatis的配置文件

<!--spring管理session工厂-->  
<bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">  
 <property name="dataSource" ref="dataSource"/>  
 <property name="mapperLocations" value="classpath:com/dao/mapper/\*.xml"/>  
 <property name="configLocation" value="classpath:mybatis-config.xml"/>  
</bean>

省略mybatis配置文件的写法：

<bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">  
 <property name="dataSource" ref="dataSource"/>  
 <property name="mapperLocations" value="classpath:dao/\*.xml"/>  
 <property name="typeAliasesPackage" value="pojo"/>  
 <!--<property name="configLocation" value="classpath:mybatis-config.xml"/>-->  
 <property name="plugins">  
 <array>  
 <bean class="com.github.pagehelper.PageHelper">  
 <property name="properties">  
 <value>  
 dialect=mysql  
 reasonable=true  
 </value>  
 </property>  
 </bean>  
 </array>  
 </property>  
</bean>

#### dao的基本整合

编写DeptDAO的实现类：

**package com.dao.impl**;  
  
**import com.dao.**DeptDAO;  
**import com.pojo.Dept**;  
**import org.mybatis.spring.support.**SqlSessionDaoSupport;  
  
**public class DeptDAOImpl extends** SqlSessionDaoSupport **implements** DeptDAO**{  
 @Override  
 public void** addDept(**Dept dept**) **{  
 super**.getSqlSession().**insert**(**"com.dao.DeptDAO.addDept"**, **dept**);  
 **}  
}**

DAO继承了spring的SqlSessionDaoSupport，SqlSessionDaoSupport中注入sessionFactory，负责管理session。

声明DeptDAO的bean

<bean id="deptDAO" class="com.dao.impl.DeptDAOImpl">  
 <property name="sqlSessionFactory" ref="sqlSessionFactory"/>  
</bean>

测试代码：

**public class SpringTest {** ApplicationContext **ac = new** ClassPathXmlApplicationContext(**"spring-mybatis.xml"**);  
  
 **@Test  
 public void** testDeptDAO() **{** DeptDAO **deptDAO =** (DeptDAO) **ac**.**getBean**(**"deptDAO"**);  
 **Dept dept = new** Dept();  
 **dept**.setName(**"abcdefg"**);  
 **deptDAO**.**addDept**(**dept**);  
 **}**

**}**

#### 面向接口的整合

mybatis的dao可以没有实现类，只写接口，要求对应的Mapper映射文件的namespace是接口的全路径。接口中方法名与sql语句的id一致。

接口EmployeeDAO

**package com.dao**;  
  
**import com.pojo.Employee**;  
  
**import java.util.**List;  
  
**/\*\*  
 \* 本接口没有实现类  
 \*/  
public interface** EmployeeDAO **{** List**<Employee>** getAllDept();  
**}**

映射文件：

映射文件EmployeeMapper.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"*>*<!--注意namespace和接口全路径一致-->

<mapper namespace="com.dao.EmployeeDAO">  
 <resultMap id="employeeMap" type="Employee">  
 <id property="id" column="ID"/>  
 <result property="name" column="NAME"/>  
 <result property="job" column="JOB"/>  
 <result property="salary" column="SALARY"/>  
 <result property="hireDate" column="HIRE\_DATE"/>  
 <result property="dept.id" column="DEPT\_ID"/>  
 <result property="dept.name" column="DEPT\_NAME"/>  
 </resultMap>  
<!--注意语句的id与接口中的方法名一致-->  
 <select id="getAllDept" resultMap="employeeMap">  
 select ID,NAME FROM EMPLOYEE  
 </select>  
</mapper>

spring中配置bean:

<!--面向接口的整合，EmployeeDAO接口没有实现类-->  
<bean id="employeeDAO" class="org.mybatis.spring.mapper.MapperFactoryBean">  
 <property name="mapperInterface" value="com.dao.EmployeeDAO"/>  
 <property name="sqlSessionFactory" ref="sqlSessionFactory"/>  
</bean>

测试代码：

**@Test  
public void** testEmpDAO() **{** EmployeeDAO **dao =** (EmployeeDAO) **ac**.**getBean**(**"employeeDAO"**);  
 List**<Employee> list = dao**.**getAllDept**();  
 **for** (**Employee employee : list**) **{  
 System**.***out***.println(**employee**.getName());  
 **}  
}**

#### 简化配置

Mybatis的接口可以没有实现类，当接口很多的时候，可以统一扫描，不需要每个接口都声明bean:

<!—dao接口包扫描器 -->  
<bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">  
 <property name="basePackage" value="com.dao"/>  
</bean>

注释掉7.2.2中employeeDAO的bean，运行测试代码一样可以成功。

简化：

当DAO的接口和Mapper映射文件在一个文件夹下，且命名一致的时候：



映射文件和接口的命名一样，只是 文件拓展名不同，配置了上面的MapperScannerConfigurer后，可以省略sqlSessionFactory中配置文件的路径：

<!--spring管理session工厂-->  
<bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">  
 <property name="dataSource" ref="dataSource"/>  
 <!--如果映射文件和接口在一个包下，并且命名一致，可以不写这句-->  
 <!--<property name="mapperLocations" value="classpath:com/dao/mapper/\*.xml"/>-->  
 <property name="configLocation" value="classpath:mybatis-config.xml"/>

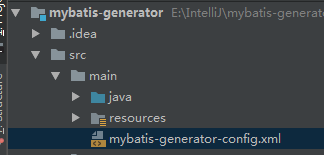
<property name="plugins">  
 <array>  
 <bean class="com.github.pagehelper.PageHelper">  
 <property name="properties">  
 <value>  
 dialect=mysql  
 reasonable=true  
 </value>  
 </property>  
 </bean>  
 </array>  
 </property>  
</bean>

## Mybatis逆向工程

新建工程，添加jar包：

<dependency>  
 <groupId>org.mybatis.generator</groupId>  
 <artifactId>mybatis-generator-core</artifactId>  
 <version>1.3.5</version>  
</dependency>  
<dependency>  
 <groupId>org.mybatis</groupId>  
 <artifactId>mybatis</artifactId>  
 <version>3.4.1</version>  
</dependency>  
<!-- mysql驱动包 -->  
<dependency>  
 <groupId>mysql</groupId>  
 <artifactId>mysql-connector-java</artifactId>  
 <version>5.1.29</version>  
</dependency>

创建逆向工程配置文件：mybatis-generator-config.xml，路径与resources和java同级，如图：



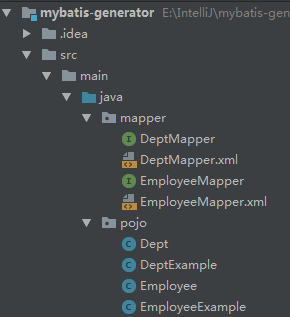
配置文件内容如下：

*<?*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>  
  
  
 <context id="context" targetRuntime="MyBatis3">  
 <commentGenerator>  
 <property name="suppressAllComments" value="false"/>  
 <property name="suppressDate" value="true"/>  
 </commentGenerator>  
  
 <!-- 数据库的配置信息 -->  
 <jdbcConnection driverClass="com.mysql.jdbc.Driver"  
 connectionURL="jdbc:mysql://localhost:3306/test?characterEncoding=utf-8"  
 userId="root"  
 password=""/>  
 <!-- 默认false，把JDBC DECIMAL 和 NUMERIC 类型解析为 Integer，为 true时把JDBC DECIMAL 和  
 NUMERIC 类型解析为java.math.BigDecimal -->  
 <javaTypeResolver>  
 <property name="forceBigDecimals" value="false"/>  
 </javaTypeResolver>  
  
 <!-- 生成实体类存放的包-->  
 <javaModelGenerator targetPackage="pojo" targetProject=".\src\main\java">  
 <property name="enableSubPackages" value="false"/>  
 <property name="trimStrings" value="true"/>  
 </javaModelGenerator>  
  
 <!-- 生成xml存放的包 -->  
 <sqlMapGenerator targetPackage="mapper" targetProject=".\src\main\java">  
 <property name="enableSubPackages" value="false"/>  
 </sqlMapGenerator>  
  
 <!-- mapper接口生成的位置-->  
 <javaClientGenerator targetPackage="mapper" targetProject=".\src\main\java" type="XMLMAPPER">  
 <property name="enableSubPackages" value="false"/>  
 </javaClientGenerator>  
  
 <!-- 需要逆向生成哪些表 -->  
 <table tableName="DEPT"/>  
 <table tableName="EMPLOYEE"/>  
 </context>  
</generatorConfiguration>

逆向生成工具类：

**import org.mybatis.generator.api.MyBatisGenerator**;  
**import org.mybatis.generator.config.Configuration**;  
**import org.mybatis.generator.config.xml.ConfigurationParser**;  
**import org.mybatis.generator.internal.DefaultShellCallback**;  
  
**import java.io.File**;  
**import java.util.ArrayList**;  
**import java.util.**List;  
  
**public class Generator {  
 public static void** main(**String**[] **args**) **throws Exception{** List**<String> warnings = new** ArrayList**<>**();  
 **boolean overwrite = true**;  
 **//指定逆向工程配置文件  
 File configFile = new** File(**"src/main/mybatis-generator-config.xml"**);  
 **ConfigurationParser cp = new** ConfigurationParser(**warnings**);  
 **Configuration config = cp**.parseConfiguration(**configFile**);  
 **DefaultShellCallback callback = new** DefaultShellCallback(**overwrite**);  
 **MyBatisGenerator myBatisGenerator = new** MyBatisGenerator(**config**,  
 **callback**, **warnings**);  
 **myBatisGenerator**.generate(**null**);  
 **System**.***out***.println(**"结束"**);  
 **}  
}**

生成的结果如下：



XXExample是条件查询的工具类，可以动态添加where查询条件

EmployeeMapper **mapper=session**.**getMapper**(EmployeeMapper.**class**);  
**EmployeeExample example=new** EmployeeExample();  
**EmployeeExample**.**Criteria criteria=example**.createCriteria();  
**criteria**.andIdGreaterThan(**8**);  
**criteria**.andNameLike(**"%" + "李" + "%"**);  
List**<Employee> employees=mapper**.**selectByExample**(**example**);  
**for** (**Employee employee : employees**) **{  
 System**.***out***.println(**employee**.getName());  
**}**

逆向生成的实体类复制到自己要开发的工程中就可以使用了。