简介

MyBatis是持久层框架（和数据库进行交互的框架）

MyBatis 不像 Hibernete 等这些全自动框架，它把关键的SQL部分交给程序员自己编写，而不是自动生成

HelloWorld

步骤：

- 导入mybatis依赖
- 配置数据源
- 编写javabean对应数据库一个表模型
- 以前：Dao接口-->Dao实现 。-->标注@Repository注解

  现在：Mapper接口-->Mapper.xml实现 ,-->标注@Mapper注解

  （安装mybatisx插件自动为mapper类生成mapper文件，我们只需要在mapper文件中配置方法的sql语句）
- 告诉mybatis去哪里找mapper文件：`mybatis.mapper-locations=classpath:mapper/**.xml`

mapper接口：

```
@Mapper //告诉spring，这是MyBatis操作数据库用的接口
public interface EmpMapper {
    Emp getEmpById(Integer id);
}
```

mapper.xml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" >
<mapper namespace="com.atguigu.mybatis.dao.EmpMapper" >
<!-- namespace ：编写mapper接口的全类名，代表这个xml文件和这个mapper接口进行绑定  -->

    <!--
    select 标签代表一次查询
    id：绑定的方法名
    resultType:返回值类型
    -->
    <select id="getEmpById" resultType="com.atguigu.mybatis.bean.Emp">
    select id,emp_name empName,age,emp_salary salary from t_emp where id=#{id}
    </select>

</mapper>
```

application.properties:

```
spring.application.name=mybatis-01-helloworld

spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/mybatis-example
spring.datasource.username=root
spring.datasource.password=abc123

# 告诉MyBatis，xml文件（Mapper文件）在哪里
mybatis.mapper-locations=classpath:mapper/**.xml
```

测试类：

```
@SpringBootTest
class Mybatis01HelloworldApplicationTests {
    @Autowired//容器中是mybatis为每个mapper接口创建的代理对象
    EmpMapper empMapper;

    @Test
    void contextLoads() {
        Emp emp = empMapper.getEmpById(1);
        System.out.println("emp:"+emp);
    }
}
```

细节

1. 每个Dao 接口 对应一个 XML 实现文件
2. Dao 实现类 是一个由 MyBatis 自动创建出来的代理对象
3. XML 中 namespace 需要绑定 Dao 接口 的全类名
4. XML 中使用 select、update、insert、delete 标签来代表增删改查
5. 每个 CRUD 标签 的 id 必须为Dao接口的方法名
6. 每个 CRUD标签的 resultType 是Dao接口的返回值类型全类名
7. 未来遇到复杂的返回结果封装，需要指定 resultMap 规则
8. 以后 xxxDao 我们将按照习惯命名为 xxxMapper，这样更明显的表示出 持久层是用 MyBatis 实现的

完成CRUD

增删改：

```
<insert id="addEmp">
    insert into t_emp(emp_name,age,emp_salary) values (#{empName},#{age},#
{salary})
</insert>

<update id="updateEmp">
    update t_emp set emp_name=#{empName},
                     age=#{age},
                     emp_salary=#{salary}
                where id=#{id}
</update>
```

```
<delete id="deleteEmp">
    delete from t_emp where id=#{id}
</delete>
```

在application.properties中写入`logging.level.com.atguigu.mybatis.mapper=debug`来生成日志。（这里的mapper1包就是原来的dao包）。

查全部：

```
<!--返回的是集合，返回值类型还写集合中的元素类型-->
<select id="getAll" resultType="com.atguigu.mybatis.bean.Emp">
    select * from t_emp
</select>
```

在配置文件中配置以下，就不用起别名的方式进行封装了

```
#开启驼峰命名自动封装:将数据库中形如emp_name封装至empName
mybatis.configuration.map-underscore-to-camel-case=true
```

细节

xml中标签中的参数有一个useGeneratedKeys="true"表示使用自动生成的id 。还有一个keyProperty="id"表示把自动生成的id封装到emp对象的id属性中。

这样，在使用了addEmp方法后，用`Integger id=emp.getId`就可以获取上次自增的id

参数处理

#{}和${}

#{}和${}都可以实现动态传参，但二者的机制不同

#{}在底层用的是预编译方式，如下：

```
@Autowired
DataSource dataSource;
@Test
void testCRUD2() throws SQLException {
    Connection connection=dataSource.getConnection();
    PreparedStatement ps=connection.prepareStatement("select * from e_emp where id=?");
    ps.setInt(1,5);
}
```

${}在底层则采用拼接方式，如下：

```
Statement statement=connection.createStatement();
statement.execute("select * from e_emp where id="+2);
```

所以，采用${}会发生sql注入的问题

但是有一种情况必须使用${} ,即sql语句中表名、字段位置不支持预编译，不能用#{}占位。这种情况用${}，但要进行sql防注入校验

结论

新版mybatis支持多个参数下，直接用#{参数名}

老版mybatis不支持以上操作，需要用@Param

```
Emp getEmployByIdAndName(@Param("id")Long id, @Param("empName")String name);
```

```
<select id="getEmployByIdAndName" resultType="com.atguigu.mybatis.bean.Emp">
    select * from t_emp where id=#{id} and emp_name=#{empName}
</select>
```

以后，用@Param(参数名)指定参数名，#{参数名}就可以取值。

在方法名后面按`alt+回车`自动生成@Param

返回值

- 如果要返回基本类型或对象，returnType就写全类名。当然，java也有对基本类型的别名，写别名也可以（但一般用全类名）。
- 如果返回list，returntype写集合中元素的类型
- 如果返回map，也写全类名，但为了让spring知道返回的map中的key是什么，我们给方法上加上@MapKey("id")表示用emp的id作为key。但此时返回的value不是Emp类型的，而是又一个hashmap。

  如果要Emp，那么returntype不能写map的全类名，而是Emp类的全类名

自定义结果集

解决问题

在上面，我们面对“和数据库名称对不上的字段查询结果封装为空”这一问题，有两种解决方案：

1.使用列别名

2.使用驼峰命名自动映射

接下来，介绍一种新的解决方案：使用ResultMap（自定义结果集）

从数据库查询数据封装至对象

默认规则是：

javabean中的属性名去数据库表中找对应列名的值（映射封装）

自定义规则（ResultMap）：我们来告诉mybatis如何把结果封装到bean。

将getEmpById方法的配置修改为：

```
<resultMap id="EmpRM" type="com.atguigu.mybatis.bean.Emp">
    <!--id:声名主键映射规则-->
    <id column="id" property="id"></id>
    <!--result:声名普通列映射规则-->
    <result column="emp_name" property="empName"></result>
    <result column="age" property="age"></result>
    <result column="emp_salary" property="empSalary"></result>
</resultMap>
<select id="getEmpById" resultMap="EmpRM">
    select * from t_emp where id=#{id}
</select>
```

所以以后遇到这个问题时：
1.开启驼峰命名

2.1搞不定的，用ResultMap

关联查询

association标签

【关联查询】案例：按照id查询订单以及下单的客户信息

```
@Data
public class Order {
    private Long id;
    private String address;
    private BigDecimal amount;
    private Long customerId;
    //订单对应的客户
    private Customer customer;

}
```

```
@Data
public class Customer {
    private Long id;
    private String customerName;
    private String phone;
    //保存所有订单
    private List<Order> orders;


}
```

```
<select id="getOrderByIdWithCustomer"
resultType="com.atguigu.mybatis.bean.Order">
    select o.*,
           c.id c_id,
           c.customer_name,
           c.phone
    from t_order o
    left join t_customer c on o.customer_id=c.id
    where o.id=#{id}

</select>
```

以上的查询我们发现spring无法将查询到的customer的字段自动封装成customer对象。

所以我们使用ResultMap

```
<resultMap id="OrderRM" type="com.atguigu.mybatis.bean.Order">
    <id column="id" property="id"></id>
    <result column="address" property="address"></result>
    <result column="amount" property="amount"></result>
    <result column="customer_id" property="customerId"></result>
    <!--一对一关联封装-->
    <association property="customer"
javaType="com.atguigu.mybatis.bean.Customer">
        <id column="c_id" property="id"></id>
        <result column="customer_name" property="customerName"></result>
        <result column="phone" property="phone"></result>
    </association>
</resultMap>
<select id="getOrderByIdWithCustomer" resultMap="OrderRM">
    select o.*,
           c.id c_id,
           c.customer_name,
           c.phone
    from t_order o
    left join t_customer c on o.customer_id=c.id
    where o.id=#{id}

</select>
```

association标签用于将查询到的customer的字段封装成customer对象。

collection标签

【关联查询】案例：按照id查询订单以及下单的客户信息

```
<resultMap id="CustomerRM" type="com.atguigu.mybatis.bean.Customer">
    <id column="c_id" property="id"></id>
    <result column="customer_name" property="customerName"></result>
    <result column="phone" property="phone"></result>
    <!--collection说明一对多的封装规则
    ofType 说明集合中的元素的类型-->
    <collection property="orders" ofType="com.atguigu.mybatis.bean.Order">
        <id column="id" property="id"></id>
```

```
        <result column="address" property="address"></result>
        <result column="amount" property="amount"></result>
        <result column="customer_id" property="customerId"></result>
    </collection>
</resultMap>
<select id="selectCustomerByIdWithOrders" resultMap="CustomerRM">
    select
        c.id c_id,
        c.customer_name,
        c.phone,
        o.*
    from t_customer c
    left join t_order o on c.id=o.customer_id
    where c.id=#{id}
</select>
```

分步查询

【分步查询】案例1：按照id查询客户 以及 他下的所有订单

原生分步写法

```
@Mapper
public interface OrderCustomerStepMapper {

    //1.查询客户
    Customer getCustomerById(Long id);

    //2.查询订单
    List<Order> getOrdersByCustomerId(Long cId);

}
```

```
<!--按照id查询客户-->
<select id="getCustomerById" resultType="com.atguigu.mybatis.bean.Customer">
    select * from t_customer where id=#{id}
</select>
<!--按照客户id查询它的所有订单-->
<select id="getOrdersByCustomerId" resultType="com.atguigu.mybatis.bean.Order">
    select * from t_order where customer_id=#{cId}
</select>
```

test:

```
@SpringBootTest
public class StepTest {
    @Autowired
    OrderCustomerStepMapper orderCustomerStepMapper;
    @Test
    void test01(){
        //按照id查询客户
```

```
        Customer customerById = orderCustomerStepMapper.getCustomerById(1L);
        //他下的所有订单
        List<Order> list =
orderCustomerStepMapper.getOrdersByCustomerId(customerById.getId());
        //组合一起
        customerById.setOrders(list);
        System.out.println(customerById);
    }

}
```

自动分步查询

collection

```
<!--分布查询的自定义结果集-->
<resultMap id="CustomerOrderStepRM" type="com.atguigu.mybatis.bean.Customer">
    <id column="id" property="id"></id>
    <result column="customer_name" property="customerName"></result>
    <result column="phone" property="phone"></result>
    <!-- 告诉mybatis，封装orders属性的时候，
    是一个集合，但是这个集合需要调用另一个方法进行查询
    select 指定我们要调用的另一个方法
    column 来指定我们要调用第二个方法时，把哪一列的值作为传递的参数
             如果只需要传递一个参数，直接如下写就行
             如果多参数，column="{cid=id,name=customer_name}"
             其中cid、name是@param给方法的形参起的名-->
    <collection property="orders"
                ofType="com.atguigu.mybatis.bean.Order"

 select="com.atguigu.mybatis.mapper.OrderCustomerStepMapper.getOrdersByCustomerI
d"
                column="id">
    </collection>
</resultMap>
<select id="getCustomerByIdAndOrderStep" resultMap="CustomerOrderStepRM">
    select * from t_customer where id=#{id}
</select>
```

association

格式同collection

延迟加载

分步查询 有时候并不需要立即运行，我们希望在用到的时候再去查询，可以开启延迟加载的功能。

配置：

```
#开启懒加载（延迟加载）
mybatis.configuration.lazy-loading-enabled=true
mybatis.configuration.aggressive-lazy-loading=false
```

动态SQL

if标签

```
   <!--if标签：判断
    test：判断条件；java代码怎么写，它就怎么写
    -->
    <select id="queryEmpByNameAndSalary"
resultType="com.atguigu.mybatis.bean.Emp">
        select * from t_emp where
        <if test="name!=null"> emp_name=#{name}</if>
        <if test="salary!=null">and emp_salary=#{salary}</if>
    </select>
</mapper>
```

where标签

我们发现在上面的xml代码中，如果name=null，会出现sql语法错误,于是：

```
<select id="queryEmpByNameAndSalary" resultType="com.atguigu.mybatis.bean.Emp">
    select * from t_emp
    <where>
        <if test="name!=null"> emp_name=#{name}</if>
        <if test="salary!=null">and emp_salary=#{salary}</if>
    </where>
</select>
```

会自动解决以上问题

where标签的作用就是解决语法错误

set标签

```
<update id="updateEmp">
    update t_emp set emp_name=#{empName},
                     emp_salary=#{empSalary},
                     age=#{age}
    where id=#{id}
</update>
```

```
@Test
void test02(){
    Emp emp = new Emp();
    emp.setEmpSalary(10.0D);
    emp.setEmpName("哈哈");
    emp.setId(4);
    emp.setAge(13);
    empDynamicSQLMapper.updateEmp(emp);
}
```

上面是能够正常操作的，但是如果测试中某一个参数不赋值，那么更新后的数据对应字段就会为null，如何解决

```
<update id="updateEmp">
    update t_emp set
    <if test="empName!=null"> emp_name=#{empName},</if>
    <if test="empSalary !=null">emp_salary=#{empSalary},</if>
    <if test="age!=null">age=#{age}</if>

    where id=#{id}
</update>
```

但是这种方法，如果不传age，sql语句会出现多一个","的错误。

```
<update id="updateEmp">
    update t_emp set
     <set> <if test="empName!=null"> emp_name=#{empName},</if>
            <if test="empSalary !=null">emp_salary=#{empSalary},</if>
            <if test="age!=null">age=#{age}</if>
     </set>
    where id=#{id}
</update>
```

set和where一样，解决语法错误问题

分支选择

```
select * from t_emp
<where>
    <choose>
        <when test="name!=null">
            emp_name=#{name}
        </when>
        <when test="salary!=null">
            emp_salary=#{salary}
        </when>
        <otherwise>
            id=1
        </otherwise>
```

```
    </choose>
</where>
```

三个里只选择一个，有多个满足选择前面的。

foreach批量操作

- 查询指定id集合中所有的员工

```
List<Emp> getEmpByIdIn(List<Integer> ids);
```

```
<!--
foreach:遍历集合、set、map、数组
    collection：指定要遍历的集合名
    item：将当前遍历出的元素赋值给指定的变量
    seperator：指定在每次遍历时，元素之间拼接的分隔符
    open:遍历开始的前缀，遍历不开始就不会拼接
    close遍历结束的后缀
-->
<select id="getEmpByIdIn" resultType="com.atguigu.mybatis.bean.Emp">
    select * from t_emp
        <if test="ids!=null">
            <foreach collection="ids" item="id" separator="," open="where id in
(" close=")">
                #{id}
            </foreach>
        </if>
</select>
```

- 批量添加员工

```
void addEmps(List<Emp> emps);
```

```
<insert id="addEmps">
    insert into t_emp(emp_name,age,emp_salary)
    values
        <foreach collection="emps" item="emp" separator=",">
            (#{emp.empName},#{emp.age},#{emp.empSalary})
        </foreach>
</insert>
```

- 批量修改

```
void updateEmps(List<Emp> emps);
```

```
<update id="updateEmps">
    <foreach collection="emps" item="emp" separator=";">
        update t_emp
```

```
            <set>
                <if test="emp.empName!=null">
                    emp_name=#{emp.empName},
                </if>
                <if test="emp.empSalary">
                    emp_salary=#{emp.empSalary},
                </if>
                <if test="emp.age!=null">
                    age=#{emp.age}
                </if>
            </set>
            where id=#{emp.id}
    </foreach>
</update>
```

并且需要在配置中设置allowMultiQueries=true来允许多个SQL用;隔开，批量发送给数据库执行

抽取可复用的sql片段

```
<sql id="column_names">
    id,emp_name empName,age,emp_salary empSalary
</sql>
```

```
<select ...>
    select
        <include refid="column_names"><include>
    from t_emp
</select>
```

xml转义字符

以后在xml中，以下字符需要用转义字符，不能直接写

原始字符	转义字符
&	&amp
<	&lt
>	&gt
"	&quot
'	&apos

缓存机制

- 一级缓存：

默认事务期间，会开启事务级别缓存；

1.同一个事务期间，前面查询的数据，后面如果再要执行相同查询，会从一级缓存中获取数据，不会给数据库发送sql

2.若两次查询的东西不一样或者两次查询期间进行了增删改，缓存会失效（缓存不命中）

3.事务结束缓存清空

- 二级缓存

事务结束后，会把数据共享到二级缓存。

其他事务开启，先在二级缓存中看有没有，再看一级缓存。如果都没有再去数据库查询

开启二级缓存需要在Mapper的xml文件的标签中写上。并且缓存的对象要实现序列化接口

PageHelper分页插件

基本使用

PageHelper 是可以用在 MyBatis 中的一个强大的分页插件

分页插件就是利用MyBatis 插件机制，在底层编写了 分页Interceptor，每次SQL查询之前会自动拼装分页数据

使用:

1. 在 pom.xml 中添加如下依赖：

```
<dependency>
    <groupId>com.github.pagehelper</groupId>
    <artifactId>pagehelper</artifactId>
    <version>最新版本</version>
</dependency>
```

2. 创建插件对象

```
@Configuration
public class MyBatisConfig {
    @Bean
    PageInterceptor pageInterceptor(){
        //创建分页插件对象
        return new PageInterceptor();
    }

}
```

3. 使用

在查询方法前调用静态方法：`PageHelper.startPage(要查询的页数，每一页的数量)`即可

原理:拦截器

1. 统计这个表的总数量
2. 给原业务底层SQL动态拼接上limit

为什么只有紧跟的第一个查询方法执行分页操作?

1. 第一个查询从ThredLocal中中获取到共享数据，执行分页
2. 第一个执行完会把ThreadLocal分页数据删除
3. 以后的查询，从ThreadLocal中拿不到分页的数据

前后端配合分页

需求:

后端收到前端传来的代码

响应前端需要的数据

1. 总页码或总记录数
2. 当前页码
3. 本页数据

将执行查询方法得到的集合list进行如下封装:

```
PageInfo<Emp> info=new PageInfo<>(list);
```

给前端返回此info, 可以用此info获取信息

逆向生成

选择数据库，自动生成bean、mapper、mapper.xml等

步骤:

- 配置数据源
- 在右侧栏中选择数据库，使用MyBatisX-Generator

- 使用@MapperScan(mapper的包名)给所有mapper类添加@mapper注解

补充

全局异常处理的每一个方法加上`e.printStackTrace()`