

MyBatis 完全自学手册

——CH.CHAN 2017.03.18

一、快速入门

二、MyBatis的增删改查

三、MyBatis框架的运行原理

四、MyBatis开发中的两种方式

原始dao的开发方式

Mapper代理的开发方式

五、SqlMapConfig配置文件说明

六、Mapper文件的输入输出映射

七、动态sql

八、关联查询

九、延迟加载

十、MyBatis缓存

十一、整合spring

十二、MyBatis逆向工程

一快速入门

1、MyBatis简介

MyBatis是什么？

我们来看看百度百科的介绍：

MyBatis 本是apache的一个开源项目*iBatis*，2010年这个项目由apache software foundation 迁移到了google code，并且改名为MyBatis。2013年11月迁移到Github。

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（Plain Old Java Objects，普通的Java对象）映射成数据库中的记录。

2、快速入门

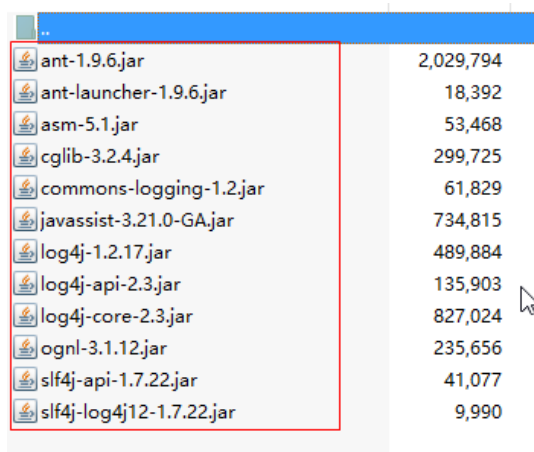
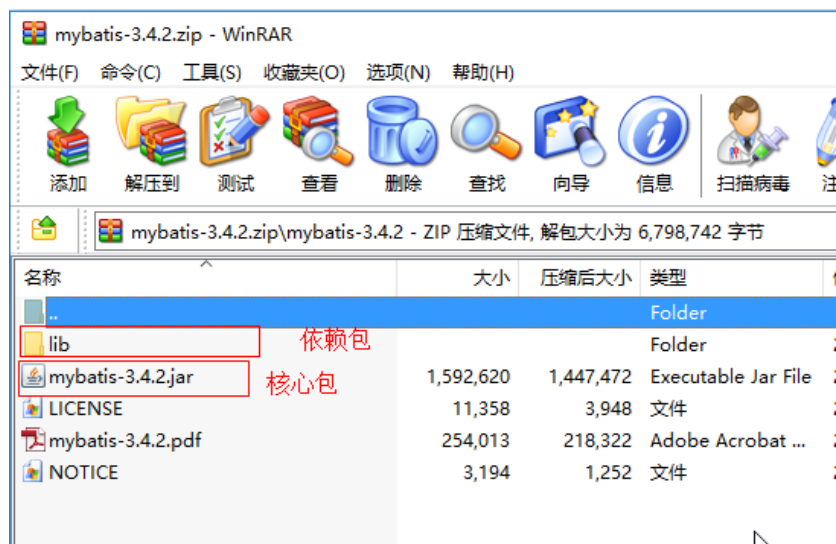
通过快速入门案例，了解使用mybatis开发的整个流程。

1)、下载 我下载3.4.6

mybatis 最新版本是3.4.2，可以从<https://github.com/mybatis/mybatis-3/releases/tag/mybatis-3.4.2> 链接进入下载。

打开压缩包可以看到，里面结构很简单，核心包，以及mybatis的以来包，mybatis的说明文档。

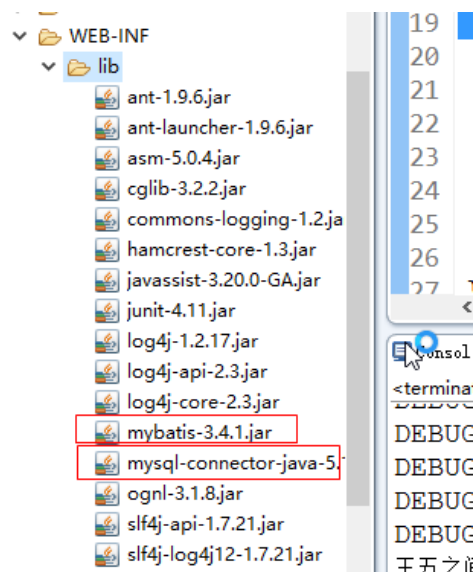
依赖包



2)、创建工程

2.1 创建一个普通的web工程(java工程也可以，这里为了方便，直接用web工程)，将mybatis核心包及依赖包拷贝到 lib目录中

2.2 将驱动包拷贝到lib目录(数据库选择mysql)



3)、创建数据库表

根据下面建表语句，通过mysql命令行窗口或者navicat工具，创建数据库表。直接运行即可，当然，前提是要创建好数据库。

/* 由于我有表格，所以此处省略

Navicat MySQL Data Transfer

尝试阅读并注解

Date: 2017-03-13 06:42:04

*/

SET FOREIGN_KEY_CHECKS=0;

-- Table structure for items

```
DROP TABLE IF EXISTS `items`;
CREATE TABLE `items` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `name` varchar(32) NOT NULL COMMENT '商品名称',
  `price` float(10,1) NOT NULL COMMENT '商品定价',
  `detail` text COMMENT '商品描述',
  `pic` varchar(64) DEFAULT NULL COMMENT '商品图片',
  `createtime` datetime NOT NULL COMMENT '生产日期',
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=4 DEFAULT CHARSET=utf8;
```

为什么不直接给出表格。。。
这里使用其他表代替可否？

-- Table structure for orderdetail

```
DROP TABLE IF EXISTS `orderdetail`;
CREATE TABLE `orderdetail` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `orders_id` int(11) NOT NULL COMMENT '订单id',
  `items_id` int(11) NOT NULL COMMENT '商品id',
  `items_num` int(11) DEFAULT NULL COMMENT '商品购买数量',
  PRIMARY KEY (`id`),
  KEY `FK_orderdetail_1` (`orders_id`),
  KEY `FK_orderdetail_2` (`items_id`),
  CONSTRAINT `FK_orderdetail_1` FOREIGN KEY (`orders_id`) REFERENCES `orders` (`id`) ON DELETE NO ACTION ON UPDATE NO ACTION,
  CONSTRAINT `FK_orderdetail_2` FOREIGN KEY (`items_id`) REFERENCES `items` (`id`) ON DELETE NO ACTION ON UPDATE NO ACTION
) ENGINE=InnoDB AUTO_INCREMENT=3 DEFAULT CHARSET=utf8;
```

-- Table structure for orders

```
DROP TABLE IF EXISTS `orders`;
CREATE TABLE `orders` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `user_id` int(11) NOT NULL COMMENT '下单用户id',
  `number` varchar(32) NOT NULL COMMENT '订单号',
  `createtime` datetime NOT NULL COMMENT '创建订单时间',
```

```
`note` varchar(100) DEFAULT NULL COMMENT '备注',
PRIMARY KEY (`id`),
KEY `FK_orders_1` (`user_id`),
CONSTRAINT `FK_orders_id` FOREIGN KEY (`user_id`) REFERENCES `user` (`id`) ON DELETE NO ACTION ON UPDATE NO ACTION
) ENGINE=InnoDB AUTO_INCREMENT=2 DEFAULT CHARSET=utf8;
```

-- Table structure for user

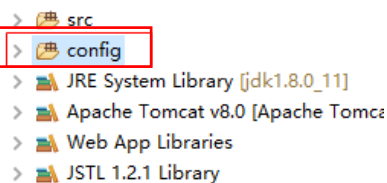
```
DROP TABLE IF EXISTS `user`;
CREATE TABLE `user` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `username` varchar(32) NOT NULL COMMENT '用户名称',
  `birthday` date DEFAULT NULL COMMENT '生日',
  `sex` char(1) DEFAULT NULL COMMENT '性别',
  `address` varchar(256) DEFAULT NULL COMMENT '地址',
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=34 DEFAULT CHARSET=utf8;
```

插入测试数据

```
INSERT INTO `items` VALUES ('1', '台式机', '3000.0', '该电脑质量非常好！！！！', null, '2015-02-03 13:22:53');
INSERT INTO `items` VALUES ('2', '笔记本', '6000.0', '笔记本性能好，质量好！！！！', null, '2015-02-09 13:22:57');
INSERT INTO `items` VALUES ('3', '背包', '200.0', '名牌背包，容量大质量好！！！！', null, '2015-02-06 13:23:02');
INSERT INTO `user` VALUES ('10', '张三', '2014-07-10', '1', '北京市');
INSERT INTO `user` VALUES ('16', '张小明', null, '1', '河南郑州');
INSERT INTO `user` VALUES ('22', '陈小明', null, '1', '河南郑州');
INSERT INTO `user` VALUES ('24', '张三丰', null, '1', '河南郑州');
INSERT INTO `user` VALUES ('25', '陈小明', '2016-12-27', '1', '河南郑州');
```

4)、配置文件

4.1 在工程下创建一个config源码包，里面存放项目的配置文件。



4.2 在config源码包下创建log4j的日志配置文件，这里使用简单的配置信息，直接拷贝内容到文件中即可。

log4j.properties

这个配置文件没问题吧？

```
# Global logging configuration
```

这里照抄原文

```
log4j.rootLogger=DEBUG, stdout
```

```
# Console output...
```

```
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
```

```
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
```

```
log4j.appender.stdout.layout.ConversionPattern=%5p [%t] - %m%n
```

这个配置文件没问题吧？

这个配置文件没问题吧？

4.2 创建db.properties文件

在config源码包下创建一个存放数据连接信息的db.properties文件

```
db.driver=com.mysql.jdbc.Driver      修改
db.url=jdbc:mysql://localhost:3306/mybatis?
useUnicode=true&characterEncoding=utf8
db.username=root                      ?
db.password=123
```

这部分代码完全照抄

4.3 mybatis配置文件

在config源码包下创建mybatis配置文件，里面存放所有mybatis全局的配置信息

SqlMapConfig.xml

打开下载的压缩包里面的pdf用户手册，在Getting Started中可以找到配置文件的模板。

Table of Contents
Introduction
Getting Started
Configuration XML
Mapper XML Files
Dynamic SQL
Java API
Statement Builders
Logging

```
<?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"/>
      <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>
  <mappers>
    <mapper resource="org/mybatis/example/BlogMapper.xml"/>
  </mappers>
</configuration>
```

数据的连接不采用模板中的方式，我们将数据库信息抽出来放到db.properties中。

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration
  PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
  "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
```

```
<!-- 加载java的配置文件或者声明属性信息 -->
<properties resource="db.properties"></properties>
```

```
<!-- 配置mybatis的环境信息-->
```

```
<environments default="development">
```

```
<environment id="development">
```

```
<!-- 配置JDBC事务控制，由mybatis进行管理 -->
```

```
<transactionManager type="JDBC"></transactionManager>
```

```
<!-- 配置数据源，采用mybatis连接池 -->
```

多了一个这个

稍有变化

```

<dataSource type="POOLED">
  <property name="driver" value="${db.driver}" />
  <property name="url" value="${db.url}" />
  <property name="username" value="${db.username}" />
  <property name="password" value="${db.password}" />
</dataSource>
</environment>
</environments>

```

这一部分抄写完毕

少了一样东西Mappers

</configuration>

首先是加载了db.properties文件。然后是mybatis的数据库配置。环境准备已经ok，接下来开始项目代码开发

? 哪一个地方?
?

5)、创建po类

根据数据库表中的user，创建User类

```

private int id;

private String username; // 用户姓名

private String sex; // 性别

private Date birthday; // 生日

private String address; // 地址

//省略getter\setter方法

```

这里使用Employee表和post表，上次的项目pojo类直接拷贝进来---算已完成

Pojo类

问题一：有些数据类型如何转换？对应？

有没有快速建立的方法？

转化为word拷贝？或者pdf如何拷贝？

6)、创建mapper文件

在config包下面，创建一个user.xml的mapper文件。文件头同样可以从mybatis的用户手册中拷贝。

这个配置文件没问题吧？

书签

- Table of Contents
- Introduction
- Getting Started
- Configuration XML
- Mapper XML Files
- Dynamic SQL
- Java API
- Statement Builders
- Logging

realized by using the XML based mapping language that has made MyBatis popular over the years. If you've used MyBatis before, the concept will be familiar to you, but there have been numerous improvements to the XML mapping documents that will become clear later. Here is an example of an XML based mapped statement that would satisfy the above SqlSession calls.

```

<?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="org.mybatis.example.BlogMapper">
  <select id="selectBlog" resultType="Blog">
    select * from Blog where id = #{id}
  </select>
</mapper>

```

While this looks like a lot of overhead for this simple example, it is actually very light. You can define as many mapped statements in a single mapper XML file as you like, so you get

下面根据用户ID查询用户信息作为案例，编写mapper文件

```

<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<!-- namespace: 命名空间，对statement的信息进行分类管理 -->
<!-- 注意: 在mapper代理时，它具有特殊及重要的作用 -->
<mapper namespace="test" >

```

这里如何拷贝啊

??

```

<!-- 根据用户ID查询用户信息 -->
<!-- select: 表示一个MappedStatement对象 -->
<!-- id: statement的唯一标示, mybatis的sqlSession的方法调用时需要用到 -->
<!-- #{}: 表示一个占位符? -->
<!-- #{id}: 里面的id表示输入参数的参数名称, 如果该参数是简单类型, 那么#{ }里面的参数名称可以任意 -->
<!-- parameterType: 输入参数的java类型 -->
<!-- resultType: 输出结果的所映射的java类型 (单条结果所对应的java类型) -->
<select id="findUserById" parameterType="int"
        resultType="org.cychan.mybatis.po.User" >
    SELECT * FROM USER WHERE id =#{id}
</select>
</mapper>

```

此处不会

7)、将mapper信息添加到mybatis的配置文件SqlMapperConfig.xml中, 注意这里要放在环境配置的后面

```

<!-- 加载映射文件 -->

```

```

<mappers>
    <mapper resource="User.xml" />
</mappers>

```

这就是上面少的那一部分吧?

8)、编写测试类

通过SqlSession的selectOne方法, 可以查询单个对象。

@Test

```

public void testSelect() throws IOException{
    //为什么没有main函数

```

这一部分也编写完毕

//加载mybatis的配置文件到输入流中

InputStream is = Resources.getResourceAsStream("SqlMapConfig.xml");

//通过SqlSessionFactoryBuilder()创建sqlSessionFactory

SqlSessionFactory sf = new SqlSessionFactoryBuilder().build(is);

//创建sqlSession

SqlSession session = sf.openSession();

//调用查询方法selectOne表示只查询一条记录

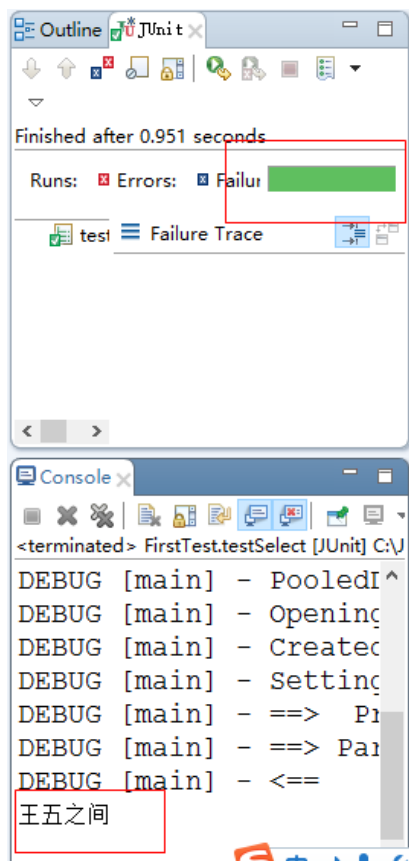
User user = session.selectOne("test.findUserById", 1);

System.out.println(user.getUsername()+ user.getAddress());

}

9)、运行测试

这个测试类没问题吧?



入门分割线

二、MyBatis的增删改查

在入门案例的基础上，继续学习mybatis的增删改查。

1、查询方面，**单个查询**上面案例已经说明，使用**selectOne**方法即可，这次使用**模糊查询**，返回多个查询结果，**mybatis**中通过**selectList**可以返回**List**结果。如果返回结果是单个，采用**selectList**不会出现问题，而返回结果是多个，但代码采用**selectOne**，编译时不会出错，但是运行时会报错。

错误提示: Expected one result (or null) to be returned by selectOne(), but found: 3

1)、复制上面入门案例的项目

2)、在user.xml中加入多个查询的statement语句

根据用户名称模糊查询用户列表，

<!-- \${}: 表示一个sql的连接符 -->

<!-- \${value}: 里面的value表示输入参数的参数名称，如果该参数是简单类型，那么\${}里面的参数名称必须是value -->

<!-- \${}这种写法存在sql注入的风险，所以要慎用！！但是在一些场景下，必须使用\${}，比如排序时，动态传入排序的列名，\${}会原样输出，不加解释 -->

```
<select id="findUsersByName" parameterType="java.lang.String"
        resultType="org.cychan.mybatis.po.User">
```



```
SELECT * FROM USER WHERE username LIKE '%${value}%'
```

```
</select>
```

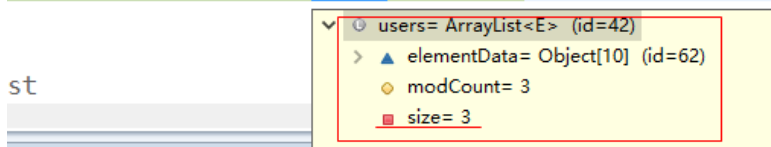
为了方便测试，在测试类中加入变量，通过setUp初始化SqlSessionFactory变量。

```
private SqlSessionFactory sqlSessionFactory;

@Before
public void setUp() throws IOException{
    String resource = "sqlMapConfig.xml";
    InputStream inputStream = Resources.getResourceAsStream(resource);
    sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);
}
```

测试方法：

```
List<User> users = session.selectList("test.findUs
System.out.println(users.size());
```



```
/**
 * 返回多条记录查询
 * @throws IOException
 */
@Test
public void testSelectByName() throws IOException{
    SqlSession session = sqlSessionFactory.openSession();

    List<User> users = session.selectList("test.findUsersByName", "小明");
    System.out.println(users.size());
}
```

2、添加记录

1)、在User.xml 文件中添加插入语句的statement，采用自增主键，添加的statement语句采用insert标签

```
<!-- 添加用户 -->
```

```
<!-- selectKey: 查询主键，在标签内需要输入查询主键的sql -->
```

```
<!-- order: 指定查询主键的sql和insert语句的执行顺序，相对于insert语句来说 -->
```

```
<!-- LAST_INSERT_ID: 该函数是mysql的函数，获取自增主键的ID，它必须配合insert语句一起使用 -->
```

```
<insert id="insertUser" parameterType="org.cychan.mybatis.po.User">

    <selectKey keyProperty="id" resultType="int" order="AFTER">
```

```

        SELECT LAST_INSERT_ID()

</selectKey>

INSERT INTO USER

(username,birthday,sex,address)

VALUES (#{username},#{birthday},#{sex},#{address})

</insert>

```

如果自增主键是uuid方式的，可以采用下面的主键生成策略，注意order必须是before，如果换成after，则会导致id为null

```

<selectKey keyProperty="id" resultType="string" order="BEFORE">

        SELECT UUID()

</selectKey>

<!-- 插入一条记录到用户表
parameterType: 输入 用户信息，指定用户的po类型
#{ }: 接收输入 参数，使用OGNL解析对象 中的属性，表示方式就是对象.属性.属性...
#{username}: 表示解析到输入 参数user对象的username属性值
-->
<insert id="insertUser" parameterType="cn.itcast.mybatis.po.User">
    <!--
    order:指定select last_insert_id()执行顺序，相对于insert into...这个语句来说
    resultType: select last_insert_id()返回类型
    keyProperty: 将select last insert id()返回值设置到user对象的哪个属性中
    -->
    <selectKey order="AFTER" resultType="int" keyProperty="id" >
        select last_insert_id()
    </selectKey>
    insert into user(username,birthday,sex,address) values(#{username},#{birthday},#{sex},#{address})
</insert>

```

2、测试代码

```

/**
 *插入对象
 */

@Test
public void insertTest(){

    SqlSession session = sqlSessionFactory.openSession();

    //创建用户对象

    User u = new User();

    u.setAddress("深南中路");

    u.setBirthday(new Date());

    u.setSex("0");

    u.setUsername("拖鞋大叔");

    session.insert("test.insertUser", u);

```

```
session.commit();
```

```
}
```

3、运行结果

```
--F--
- ==> Parameters: 拖鞋大叔(String), 2017-03-18 20:25:41.585(
- <== Updates: 1
- Committing JDBC Connection [com.mysql.jdbc.JDBC4Connect
```

查询数据库:

```
2 SELECT * from `user`
3 WHERE username like '%拖鞋%'
```

信息

结果1

概况

状态

id	username	birthday	sex	address
112	拖鞋大叔	2017-03-18	0	深南中路

证明数据已经插入数据库

3、更新记录

1)、需求: 根据id将地址深南中路加上深圳市福田区。

2)、在user.xml文件中, 加入更新语句的statement, 更新语句的statement语句采用update标签

<!-- 更新用户 -->

```
<update id="updateUser" parameterType="org.cychan.mybatis.po.User">
```

```
    UPDATE USER SET  username=#{username}, birthday=#{birthday}, sex=#
```

```
{sex}, address=#{address}
```

```
    WHERE id = #{id}
```

```
</update>
```

3)、测试

@Test

```
public void updateUser(){
```

```
    SqlSession session = sqlSessionFactory.openSession();
```

```
    User user = session.selectOne("test.findUserById", 112);
```

```
    user.setAddress("深圳市福田区深南中路绿景纪元NEO大厦");
```

```
    session.update("test.updateUser", user);
```

```
    session.commit();
```

```

        session.close()
    }
}

```

查看数据库：

```

SELECT * from `user`
WHERE username like '%拖鞋%'

```

结果1	概况	状态	
username	birthday	sex	address
112 拖鞋大叔	2017-03-18	0	深圳市福田区深南中路绿景纪元NEO大厦

4、删除记录

1)、删除操作类似更新，差别的地方在于statement的编写

```

<!-- 删除用户 -->
<delete id="deleteUser" parameterType="int">
    DELETE FROM USER WHERE ID=#{id}
</delete>

```

删除前数据库：

id	username	birthday	sex	address
1	王五	(Null)	2	之间
10	张三	2014-07-11		北京市
16	张小明	(Null)	1	河南郑州
22	陈小明	(Null)	1	河南郑州
24	张三丰	(Null)	1	河南郑州
25	陈小明	2016-12-21		河南郑州
35	更新用户	2017-03-10		d

2)、测试代码

```

/**
 * 删除
 */
@Test
public void deleteUser(){
    SqlSession session = sqlSessionFactory.openSession();
    session.delete("test.deleteUser",35);
    session.commit();
}

```

```
        session.close();  
    }  
}
```

3)、运行结果

```
DEBUG [main] - ==> Parameters: 35(Integer)  
DEBUG [main] - <== Updates: 1  
DEBUG [main] - Committing JDBC Connection [co  
DEBUG [main] - Resetting autocommit to true o
```

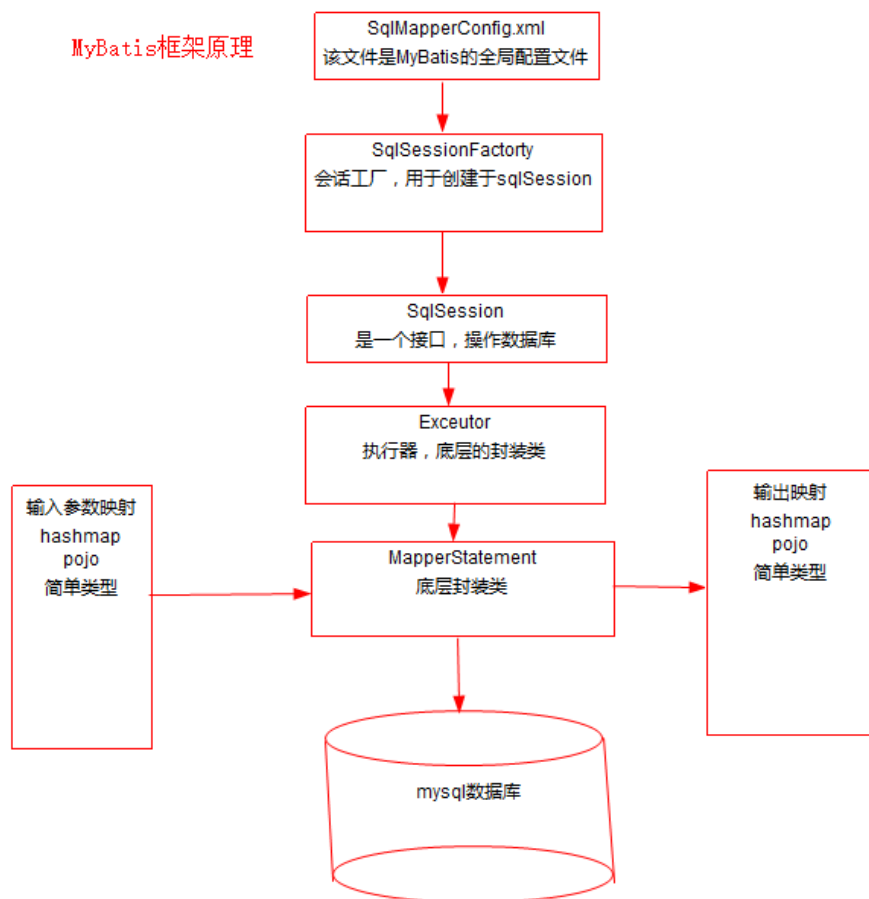
id	username	birthday	sex	address
1	王五	(Null)	2	之间
10	张三	2014-07-11	1	北京市
16	张小明	(Null)	1	河南郑州
22	陈小明	(Null)	1	河南郑州
24	张三丰	(Null)	1	河南郑州
25	陈小明	2016-12-21		河南郑州

三、MyBatis框架的运行原理

上面从使用的角度学习了mybatis的crud操作，对mybatis的使用方式有了基本的了解。接下来将从实际开发的角度，来学习mybatis在开发中的应用。

1、mybatis的运行原理：

MyBatis框架原理



1)、mybatis配置

SqlMapConfig.xml，此文件作为mybatis的全局配置文件，配置mybatis的运行环境等信息。

mapper.xml文件即**sql**映射文件，文件中配置了操作数据库的**sql**语句。此文件需要在**SqlMapConfig.xml**中加载。

2)、通过mybatis环境等配置信息构造**SqlSessionFactory**即会话工厂

3)、由会话工厂创建**sqlSession**即会话，操作数据库需要通过**sqlSession**进行。

4)、mybatis底层自定义了**Executor**执行器接口操作数据库，**Executor**接口有两个实现，一个是基本执行器、一个是缓存执行器。

5)、**Mapped Statement**也是mybatis一个底层封装对象，它包装了mybatis配置信息及**sql**映射信息等。**mapper.xml**文件中一个**sql**对应一个**Mapped Statement**对象，**sql**的**id**即是**Mapped statement**的**id**。

6)、**Mapped Statement**对**sql**执行输入参数进行定义，包括**HashMap**、基本类型、**pojo**，**Executor**通过**Mapped Statement**在执行**sql**前将输入的**java**对象映射至**sql**中，输入参数映射就是**jdbc**编程中对**preparedStatement**设置参数。

7)、**Mapped Statement**对**sql**执行输出结果进行定义，包括**HashMap**、基本类型、**pojo**，**Executor**通过**Mapped Statement**在执行**sql**后将输出结果映射至**java**对象中，输出结果映射过程相当于**jdbc**编程中对结果的解析处理过程。

2、与hibernate的区别

Mybatis和**hibernate**不同，它不完全是一个ORM框架，因为**Mybatis**需要程序员自己编写**Sql**语句，不过**mybatis**可以通过XML或注解方式灵活配置要运行的**sql**语句，并将**java**对象和**sql**语句映射生成最终执行的**sql**，最后将**sql**执行的结果再映射生成**java**对象。

Mybatis学习门槛低，简单易学，程序员直接编写原生态**sql**，可严格控制**sql**执行性能，灵活度高，非常适合对关系数据模型要求不高的软件开发，例如互联网软件、企业运营类软件等，因为这类软件需求变化频繁，一旦需求变化要求成果输出迅速。但是灵活的前提是**mybatis**无法做到数据库无关性，如果实现支持多种数据库的软件则需要自定义多套**sql**映射文件，工作量大。

Hibernate对象/关系映射能力强，数据库无关性好，对于关系模型要求高的软件（例如需求固定的定制化软件）如果用**hibernate**开发可以节省很多代码，提高效率。但是**Hibernate**的学习门槛高，要精通门槛更高，而且怎么设计O/R映射，在性能和对象模型之间如何权衡，以及怎样用好**Hibernate**需要具有很强的经验和能力才行。

四、MyBatis开发中的两种方式

再回忆下入门案例中用到的几个类：

SqlSessionFactoryBuilder: SqlSessionFactoryBuilder用于创建SqlSessionFactory，SqlSessionFactory一旦创建完成就不需要

SqlSessionFactoryBuilder了，因为SqlSession是通过SqlSessionFactory生产，所以可以将SqlSessionFactoryBuilder当成一个工具类使用，最佳使用范围是方法范围即方法体内局部变量。与spring整合后，SqlSessionFactory就不需要手动创建了，由spring容器来产生。

SqlSessionFactory: SqlSessionFactory是一个接口，接口中定义了openSession的不同重载方法，SqlSessionFactory的最佳使用范围是整个应用运行期间，一旦创建后可以重复使用，通常以单例模式管理SqlSessionFactory。

SqlSession: SqlSession是一个面向用户的接口，sqlSession中定义了数据库操作，默认使用DefaultSqlSession实现类。sqlSession是线程不安全的，在sqlSession实现类中，除了接口中的方法，还有数据域属性，所以sqlSession的最佳使用范围，是在方法体内，即当做局部变量来使用。

明确了这几个类的使用范围，再来看看实际开发中MyBatis是如何使用的。

1、原始dao开发方式

原始的dao开发方式，需要创建dao与实现类。

需求：采用原始dao的方式，根据id查询用户信息

1) 创建UsrDao

在入门案例的基础上，创建UserDao接口

```
package org.cychan.mybatis.dao;

import org.cychan.mybatis.po.User;

public interface UserDao {
    /**
     * 根据id查询用户
     * @param id
     * @return
     * @throws Exception
     */
    public User findUserById(int id) throws Exception;
}
```

2) 创建UserDao的实现类UserDaoImpl

```
public class UserDaoImpl implements UserDao {

    private SqlSessionFactory factory;
    public UserDaoImpl(SqlSessionFactory factory){
        this.factory = factory;
    }

    @Override
    public User findUserById(int id) throws Exception {
        SqlSession session = factory.openSession();
        User u = session.selectOne("test.findUserById", id);
        return u;
    }
}
```

经过前面的分析，SqlSessionFactory与SqlSession的作用域，将SqlSessionFactory作为实现类的变

量，通过构造方法进行注入。而SqlSession则在实现类的方法体内，避免因为线程安全导致出现问题。

3) 测试类

```
public class UserDaoTest {

    private SqlSessionFactory sqlSessionFactory;

    @Before

    public void setUp() throws IOException{

        String resource = "sqlMapConfig.xml";

        InputStream inputStream = Resources.getResourceAsStream(resource);

        sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);

    }

    /**
     * 原始dao开发方式，根据用户id查询用户信息
     * @throws Exception
     */

    @Test

    public void testFindUserById() throws Exception {

        //通过构造方法传入sqlSessionFactory实例

        UserDao userDao = new UserDaoImpl(sqlSessionFactory);

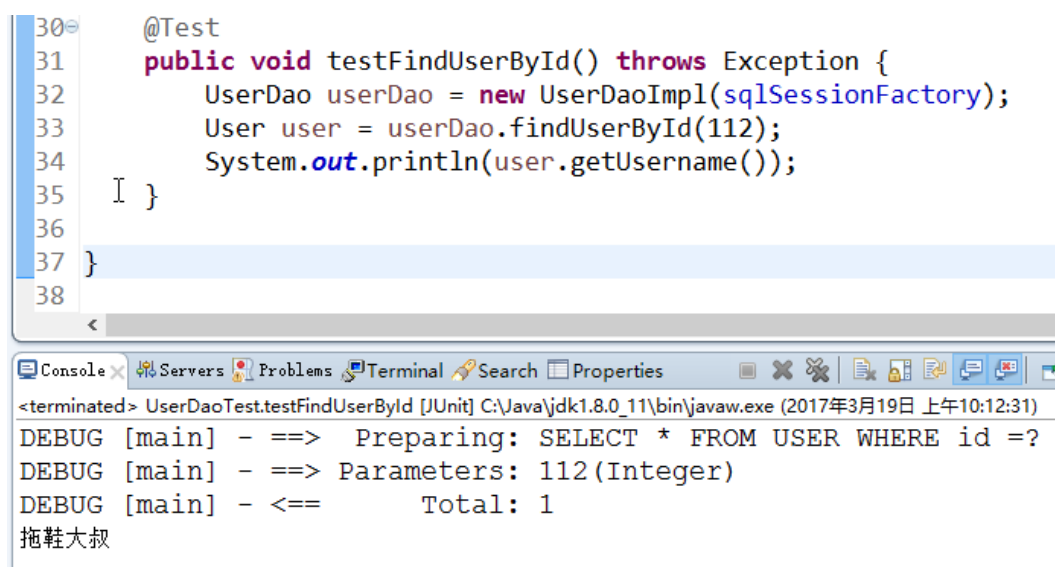
        User user = userDao.findUserById(112);

        System.out.println(user.getUsername());

    }

}
```

运行结果：



The screenshot shows an IDE with a code editor and a console window. The code editor displays the test method `testFindUserById()` from the `UDaoTest` class. The console window shows the execution output, including the SQL query `SELECT * FROM USER WHERE id =?` and the result `Total: 1`. The console also shows the path to the Java executable and the date/time of the run.

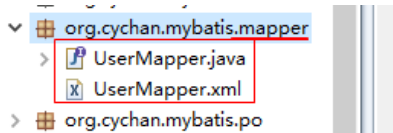
```
@Test
public void testFindUserById() throws Exception {
    UserDao userDao = new UserDaoImpl(sqlSessionFactory);
    User user = userDao.findUserById(112);
    System.out.println(user.getUsername());
}

<terminated> UserDaoTest.testFindUserById [JUnit] C:\Java\jdk1.8.0_11\bin\javaw.exe (2017年3月19日 上午10:12:31)
DEBUG [main] - ==> Preparing: SELECT * FROM USER WHERE id =?
DEBUG [main] - ==> Parameters: 112(Integer)
DEBUG [main] - <==      Total: 1
拖鞋大叔
```


2、Mapper代理的开发方式

Mapper接口开发方法只需要程序员编写Mapper接口（相当于Dao接口），由Mybatis框架根据接口定义创建接口的动态代理对象，代理对象的方法体同上边Dao接口实现类方法。

1）、在项目中创建一个mapper包，把上面案例的user.xml中复制一份，改名为UserMapper.xml放到mapper包中，最后创建一个UserMapper.java的接口，注意：UserMapper.xml与UserMapper.java 必须在同一个包下。



2）、打开UserMapper.xml文件，修改namespace中的值为UserMapper接口的全限定类名：

```
namespace="org.cychan.mybatis.mapper.UserMapper"
```

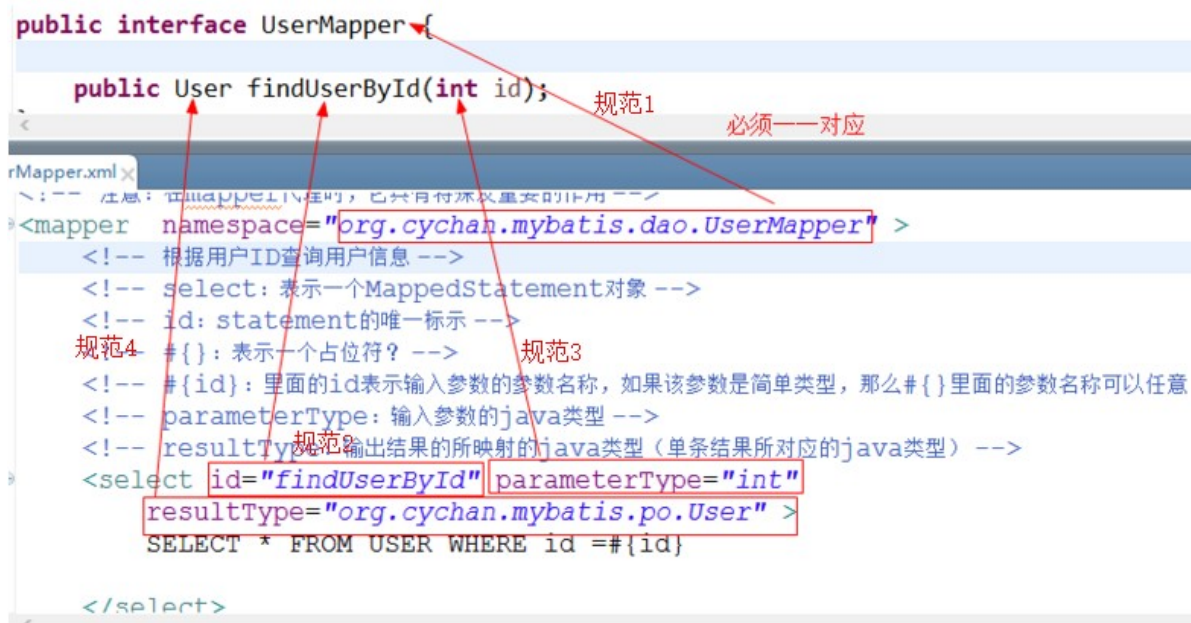
Mapper代理方式的要求规范是：

规范1：使用`mapper`代理，`namespace`接口的全限定类名

规范2：接口中的方法名与`statement`的`id`一致

规范3：接口的输入参数与`parameterType`指定的类型一致

规范4：接口的返回值与`statement`的`resultType`一致



3）、把UserMapper.xml文件加入SqlMapConfig.xml配置文件中。

```
<mapper resource="org/cychan/mybatis/mapper/UserMapper.xml" />
```

问题：如果每一个mapper文件都要手动配置加载，不但工作量大，而且会造成sqlmapConfig文件过于臃肿。

解决方案：使用批量扫描，通过package标签批量加载mapper文件。

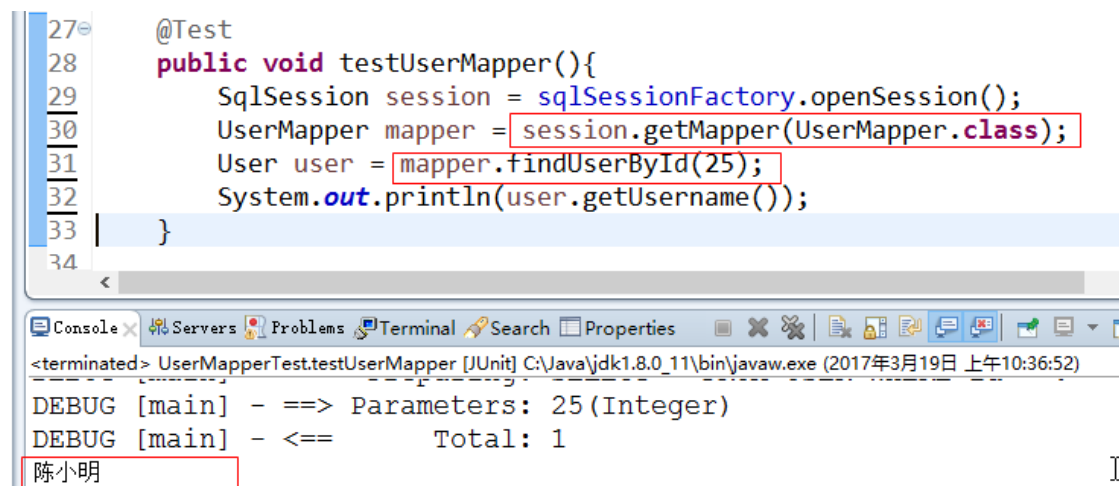
```
<package name="org.cychan.mybatis.mapper" />
```

4)、创建测试方法

@Test

```
public void testUserMapper(){  
    SqlSession session = sqlSessionFactory.openSession();  
    //直接通过session获取mapper代理对象  
    UserMapper mapper = session.getMapper(UserMapper.class);  
    //通过mapper代理对象操作数据库  
    User user = mapper.findUserById(25);  
    System.out.println(user.getUsername());  
}
```

5)、运行结果:



The screenshot shows an IDE with the following code in the editor:

```
27 @Test  
28 public void testUserMapper(){  
29     SqlSession session = sqlSessionFactory.openSession();  
30     UserMapper mapper = session.getMapper(UserMapper.class);  
31     User user = mapper.findUserById(25);  
32     System.out.println(user.getUsername());  
33 }  
34
```

The console output at the bottom shows the execution of the test:

```
<terminated> UserMapperTest.testUserMapper [JUnit] C:\Java\jdk1.8.0_11\bin\javaw.exe (2017年3月19日 上午10:36:52)  
DEBUG [main] - ==> Parameters: 25(Integer)  
DEBUG [main] - <== Total: 1  
陈小明
```

3、两种开发方式的优缺点

原始dao方式: 需要程序员自己写实现类; 在方法体内硬编码statement id ;

mapper代理方式: 需要遵循MyBatis的开发规范。

MyBatis官方推荐使用mapper代理的方式进行开发。企业开发中, 较多的是使用mapper代理方式。

五、SqlMapConfig.xml配置文件说明

1、properties属性

可以配置属性值或者加载properties属性文件。对于properties的属性加载顺序:

- I 首先读取properties中元素内配置的属性,
- II 然后读取resource 或者url中加载的属性, 会覆盖properties体内的值
- III 读取parameterType传递的属性, 会覆盖已经存在的属性值。

2、setting 属性

全局参数配置:

全局参数将会影响mybatis的运行行为。

Setting (设置)	Description (描述)	Valid Values (验证值组)	Default (默认值)
cacheEnabled	在全局范围内启用或禁用缓存配置任何映射器在此配置下。	true false	TRUE
lazyLoadingEnabled	在全局范围内启用或禁用延迟加载。禁用时，所有协会将热加载。	true false	TRUE
aggressiveLazyLoading	启用时，有延迟加载属性的对象将被完全加载后调用懒惰的任何属性。否则，每一个属性是按需加载。	true false	TRUE
multipleResultSetsEnabled	允许或不允许从一个单独的语句（需要兼容的驱动程序）要返回多个结果集。	true false	TRUE
useColumnLabel	使用列标签，而不是列名。在这方面，不同的驱动有不同的行为。参考驱动文档或测试两种方法来决定你的驱动程序的行为如何。	true false	TRUE
useGeneratedKeys	允许JDBC支持生成的密钥。兼容的驱动程序是必需的。此设置强制生成的键被使用，如果设置为true，一些驱动会不兼容性，但仍然可以工作。	true false	FALSE
autoMappingBehavior	指定MyBatis的应如何自动映射列到字段/属性。NONE自动映射。PARTIAL只会自动映射结果没有嵌套结果映射定义里面。FULL会自动映射的结果映射任何复杂的（包含嵌套或其他）。	NONE, PARTIAL, FULL	PARTIAL
defaultExecutorType	配置默认执行人。SIMPLE执行人确实没有什么特别的。REUSE执行器重用准备好的语句。BATCH执行器重用语句和批处理更新。	SIMPLE REUSE BATCH	SIMPLE
defaultStatementTimeout	设置驱动程序等待一个数据库响应的秒数。	Any positive integer	Not Set (null)
safeRowBoundsEnabled	允许使用嵌套的语句RowBounds。	true false	FALSE
mapUnderscoreToCamelCase	从经典的数据库列名A_COLUMN启用自动映射到骆驼标识的经典的Java属性名aColumn。	true false	FALSE
localCacheScope	MyBatis的使用本地缓存，以防止循环引用，并加快反复嵌套查询。默认情况下（SESSION）会话期间执行的所有查询缓存。如果localCacheScope=STATMENT本地会话将被用于语句的执行，只是没有将数据共享之间的两个不同的调用相同的SqlSession。	SESSION STATEMENT	SESSION
jdbcTypeForNull	指定为空值时，没有特定的JDBC类型的参数的JDBC类型。有些驱动需要指定列的JDBC类型，但其他像NULL，VARCHAR或OTHER的工作与通用值。	JdbcType enumeration. Most common are: NULL, VARCHAR and OTHER	OTHER
lazyLoadTriggerMethods	指定触发延迟加载的对象的方法。	A method name list separated by commas	equals, clone, hash Code, toString
defaultScriptingLanguage	指定所使用的语言默认为动态SQL生成。	A type alias or fully qualified class name.	org.apache.ibatis.scripting.xmltag s.XMLDynamicLanguageDriver
callSettersOnNulls	指定如果setter方法或地图的put方法时，将调用检索到的值是null。它是有用的，当你依靠Map.keySet（）或null初始化。注意原语（如整型，布尔等）不会被设置为null。	true false	FALSE
logPrefix	指定的前缀字符串，MyBatis将会增加记录器的名称。	Any String	Not set
logImpl	指定MyBatis的日志实现使用。如果此设置是不存在的记录的实施将自动查找。	SLF4J LOG4J LOG4J2 JDK_LOGGING COMMONS_LOGGING STDOUT_LOGGING	Not set
proxyFactory	指定代理工具，MyBatis将会使用创建懒加载能力的对象。	CGLIB JAVASSIST	

3、typeAliases别名定义

statement 中的输出输入参数，需要用到全限定类名，不方便开发，可以采用别名的方式，简化开发。

别名的方式：

1) 单个别名定义方式

<typeAliases type="全限定类名" alias="别名 ">

2) 批量定义别名：别名是类名，首字母大写或者小写

<package name="包名_com.itheima.mybatis.po" />

测试：

在sqlMapConfig.xml文件中加入自定义别名，两种采用其中一种方式即可

```

<!-- 自定义别名 -->
<typeAliases>
  <!-- 单个别名定义 -->
  <!-- <typeAlias type="com.itheima.mybatis.po.User" alias="user"/> -->

  <!-- 批量别名定义（推荐） -->
  <!-- package: 指定包名称来为该包下的po类声明别名，默认的别名就是类名（首字母大小写都可） -->
  <package name="com.itheima.mybatis.po" />
</typeAliases>

```

使用别名：

```

<select id="findUsersByName" parameterType="java.lang.String"
  resultType="user">
  SELECT * FROM USER WHERE username LIKE '%${value}%'
</select>

```

4、typeHandlers 类型处理器

mysql 的类型与jdbc类型的转换

类型处理器用于java类型和jdbc类型映射，如下：

```

<select id="findUserById" parameterType="int" resultType="user">
  select * from user where id = #{id}
</select>

```

mybatis自带的类型处理器基本上满足日常需求，不需要单独定义。

mybatis支持类型处理器：

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

不是索引)。

5、mapper的映射配置

1) 单个加载方式:

```
<mapper resource="xxx.xml" />
```

```
<mapper class="xxx.xml" />
```

class方式要求规范: 需要将mapper接口类名与mapper.xml映射文件的名称保持一致, 且在一个目录中

2) 批量加载的方式

要求规范: 需要将mapper接口类名与mapper.xml映射文件的名称保持一致, 且在一个目录中

```
<package name="com.itheima.mybatis.mapper" />
```

六、Mapper文件的输入输出映射

1、输入映射

在mapper文件中, 输入映射通过parameterType属性, 可以使简单类型、包装类型与pojo
建议通过扩展po的方式。

1) 简单类型

根据id查询用户案例中, id int是简单类型, 直接写int即可。

```
<select id="findUserById" parameterType="int"
        resultType="org.cychan.mybatis.po.User" >
    SELECT * FROM USER WHERE id =#{id}

</select>
```

2) pojo类型

如插入与更新的statement中, parameterType是User类型

<!-- 更新用户 -->

```
<update id="updateUser" parameterType="org.cychan.mybatis.po.User">
    UPDATE USER SET  username=#{username},birthday=#{birthday},sex=#{
sex},address=#{address}
    WHERE id = #{id}

</update>
```

3) 输入类型为hashmap

Sql映射文件定义如下:

```
<!-- 传递hashmap综合查询用户信息 -->
<select id="findUserByHashmap" parameterType="hashmap" resultType="user">
    select * from user where id=#{id} and username like '%${username}%'

</select>
```

注意红色标注的是hashmap的key。

4) 包装类型与pojo类似, 后面案例用到时会详细介绍。

2、输出映射

MyBatis的输出映射比输入映射要复杂的多。对象关联关系就是通过高级映射来实现的。

输出映射有三种, resultType, resultMap

2.1 resultType结果类型

1) resultType映射到简单类型

```
<!-- 获取用户列表总数 -->
<select id="findUserCount" parameterType="user" resultType="int">
    select count(1) from user

</select>
```

注意: 查询结果只有一行, 且只有一列, 可以映射输出简单类型, 使用selectOne方法即可查询。

2) resultMap映射pojo

使用resultType进行输出映射，只有当列名与pojo的属性名一致，才能映射成功；只要有一个属性值与列名一致，就会创建pojo对象。sql执行结果不管是单个还是多个，类型都填pojo的类型

3) resultMap映射到hashmap

输出pojo对象可以改用hashmap输出类型，将输出的字段名称作为map的key，value为字段值。与输入hashmap类似，这里不做重复举例了。

2.2 resultMap结果类型

1) 将pojo映射到resultMap

resultMap的定义采用标签resultMap，属性type指的是需要映射到的最终对象。

id标签，表示与po类中id属性一一对应唯一标识。非标识属性采用result标签进行映射。

```
<resultMap id="user_rm" type="org.cychan.mybatis.po.User" >
    <id column="id" property="id"/> //唯一标识 表--po属性
    <result column="username" property="username"/>
    <result column="sex" property="sex"/>
    <result column="birthday" property="birthday"/>
    <result column="address" property="address"/>
</resultMap>
```

将statement修改为：

```
<select id="findUserById" parameterType="int"
    resultMap="user_rm"> //如果resultMap在其他的mapper文件中，需要加入namespace
    SELECT * FROM USER where id = #{id}
</select>
```

在Mapper代理的实例中，按上面的方式，将resultType 改为resultMap="user_rm"，其他不变，再运行测试程序，程序可正常运行。

3、#{ } 与\${ } 的差别

#{ }实现的是向prepareStatement中的预处理语句中设置参数值，sql语句中#{ }表示一个占位符即?。

```
<!-- 根据id查询用户信息 -->
<select id="findUserById" parameterType="int" resultType="user">
    select * from user where id = #{id}
</select>
```

使用占位符#{ }可以有效防止sql注入，在使用时不需要关心参数值的类型，mybatis会自动进行java类型和jdbc类型的转换。#{ }可以接收简单类型值或pojo属性值，如果parameterType传输单个简单类型值，#{ }括号中可以是value或其它名称。

\${ }和#{ }不同，通过\${ }可以将parameterType 传入的内容拼接在sql中且不进行jdbc类型转换，\${ }可以接收简单类型值或pojo属性值，如果parameterType传输单个简单类型值，\${ }括号中只能是value。使用\${ }不能防止sql注入，但是有时用\${ }会非常方便，如下的例子：

```
<!-- 根据名称模糊查询用户信息 -->
<select id="selectUserByName" parameterType="string" resultType="user">
    select * from user where username like '%${value}%' --注入 小明% and sex like '%'
</select>
```

如果本例子使用#{ }则传入的字符串中必须有%号，而%是人为拼接在参数中，显然有点麻烦，如果采用\${ }在sql中拼接为%的方式则在调用mapper接口传递参数就方便很多。

```
//如果使用占位符号则必须人为在传参数中加%
List<User> list = userMapper.selectUserByName("%管理员%");
```


//如果使用\${}原始符号则不用人为在参数中加%

```
List<User>list = userMapper.selectUserByName("管理员");
```

再比如order by排序，如果将列名通过参数传入sql，根据传的列名进行排序，应该写为：

```
ORDER BY ${columnName}
```

如果使用#{ }将无法实现此功能。

七、动态sql

1、 where、 if

```
<where>
```

```
<if test="user.sex != null and user.sex != ''">
```

```
    and user.sex = #{user.sex}
```

```
</if>
```

```
</where>
```

如果有多个条件，在后面直接添加if语句即可。注意语气的前面要加"and"，where标签可以自动去除第一个and

注意要做不等于空字符串校验。

```
<!--  
where可以自动去掉条件中的第一个and  
-->  
<where>  
    <if test="userCustom!=null">  
        <if test="userCustom.sex!=null and userCustom.sex!=''">  
            and user.sex = #{userCustom.sex}  
        </if>  
        <if test="userCustom.username!=null and userCustom.username!=''">  
            and user.username LIKE '%${userCustom.username}%'  
        </if>  
    </if>  
</where>
```

2、 sql 片段

mapper文件中的Sql可将重复的sql提取出来，使用时用include引用即可，最终达到sql重用的目的。出于复用的角度考虑，一般建议不要包含where

```
<sql id="query_user">  
    id = #{value}  
</sql>
```

使用：

```
<select id="findUserById" parameterType="int"  
    resultMap="user_rm">  
    SELECT * FROM USER  
    <where>  
        <include refid="query_user"/>  
    </where>  
  
</select>
```

3、 foreach

向sql传递数组或者list，可以使用foreach解析

```
<foreach
```

```

        collection="指定输入对象中的集合属性"
        separator=", 两个对象中间拼接的串 "
        index=""
        open="and ( 开始遍历时的拼接串 " close=") "
        item="每次遍历生成的对象id">
            每次遍历需要拼接的串
            id = #{id}
    </foreach>

```

1)、传递pojo中的list

需求：传入多个id查询用户信息，用下边两个sql实现：

```
SELECT * FROM USERS WHERE username LIKE '%张%' AND (id =10 OR id =89 OR id=16)
```

```
SELECT * FROM USERS WHERE username LIKE '%张%' id IN (10,89,16)
```

◆ 在pojo的包装类QueryVO中定义list属性ids存储多个用户id，并添加getter/setter方法

```

public class QueryVo {

    private User user;

    //自定义用户扩展类
    private UserCustom userCustom;

    //传递多个用户id
    private List<Integer> ids;
}

```

◆ mapper.xml

```

<if test="ids!=null and ids.size>0">
    <foreach collection="ids" open="and id in(" close=")" item="id" separator="," >
        #{id}
    </foreach>
</if>

```

◆ 测试代码：

```

List<Integer> ids = new ArrayList<Integer>();
ids.add(1); //查询id为1的用户
ids.add(10); //查询id为10的用户
queryVo.setIds(ids);
List<User> list = userMapper.findUserList(queryVo);

```

2) 传递单个list

传递List类型在编写mapper.xml没有区别，唯一不同的是只有一个List参数时它的参数名为list。

如下：

◆ Mapper.xml

```

<select id="selectUserByList" parameterType="java.util.List" resultType="user">
    select * from user
    <where>
    <!-- 传递List, List中是pojo -->
    <if test="list!=null">
    <foreach collection="list" item="item" open="and id in(" separator="," close=")">

```



```

        #{item.id}
    </foreach>
</if>
</where>
</select>

```

◆ Mapper接口

```
public List<User> selectUserByList(List userList) throws Exception;
```

◆ 测试:

```

Public void testselectUserByList()throws Exception{
    //获取session
    SqlSession session = sqlSessionFactory.openSession();
    //获取mapper接口实例
    UserMapper userMapper = session.getMapper(UserMapper.class);
    //构造查询条件List
    List<User> userList = new ArrayList<User>();
    User user = new User();
    user.setId(1);
    userList.add(user);
    user = new User();
    user.setId(2);
    userList.add(user);
    //传递userlist列表查询用户列表
    List<User> list = userMapper.selectUserByList(userlist);
    //关闭session
    session.close();
}

```

3)、传递数组

传递数组，需要用到index属性

```

<!-- 传递数组综合查询用户信息 -->
<select id="selectUserByArray" parameterType="Object[]" resultType="user">
    select * from user
    <where>
        <!-- 传递数组 -->
        <if test="array!=null">
            <foreach collection="array" index="index" item="item" open="and id in("separator=","close=")">
                #{item.id}
            </foreach>
        </if>
    </where>
</select>

```

sql只接收一个数组参数，这时sql解析参数的名称mybatis固定为array，如果数组是通过一个pojo传递到sql则参数的名称为pojo中的属性名。

index: 为数组的下标。

item: 为数组每个元素的名称，名称随意定义

open: 循环开始

close: 循环结束

separator: 中间分隔输出

```

<if test="ids!=null">
<!-- 使用 foreach遍历传入ids
collection: 指定输入 对象中集合属性
item: 每个遍历生成对象中
open: 开始遍历时拼接的串
close: 结束遍历时拼接的串
separator: 遍历的两个对象中需要拼接的串
-->
<!-- 使用实现下边的sql拼接:
    AND (id=1 OR id=10 OR id=16)
-->
<foreach collection="ids" item="user_id" open="AND (" close=")" separator="or">
    <!-- 每个遍历需要拼接的串 -->
    id=#{user_id}
</foreach>
</if>

```

◆ Mapper接口:

```
public List<User> selectUserByArray(Object[] userlist) throws Exception;
```

◆ 测试:

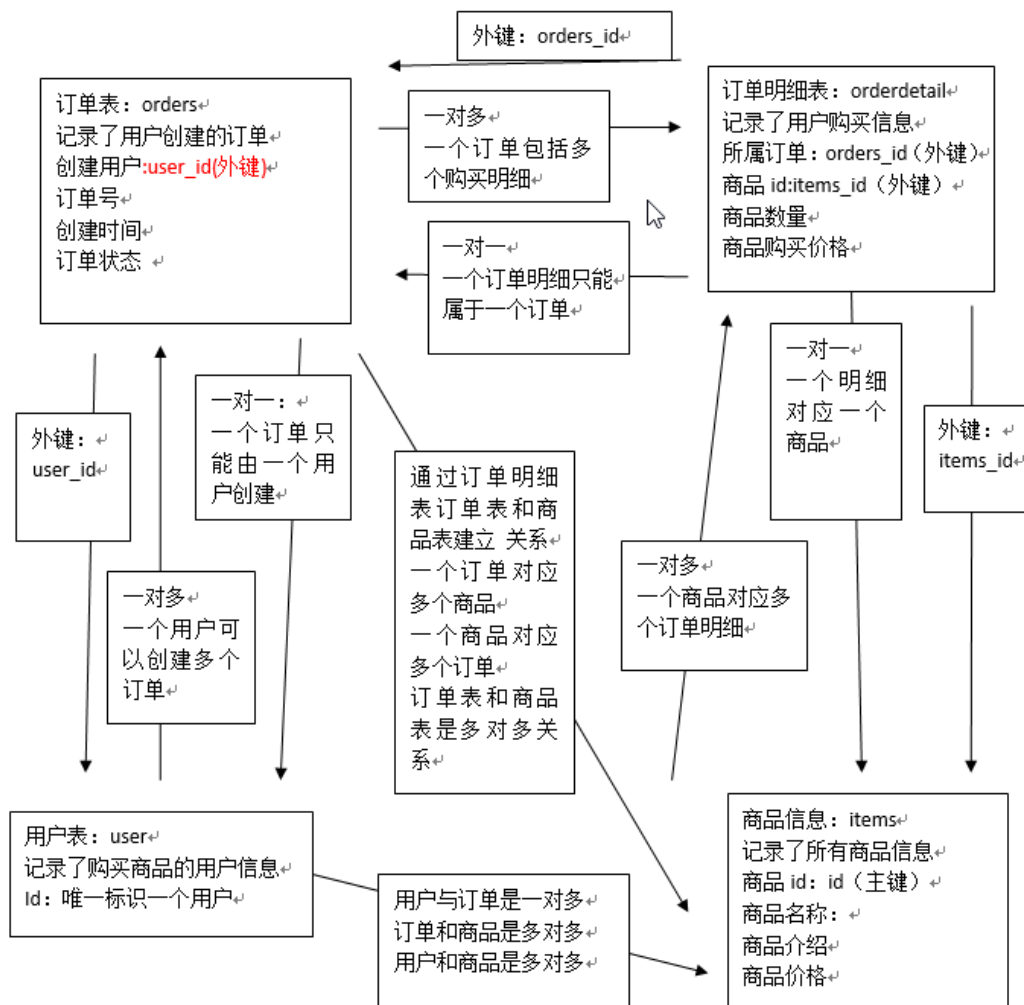
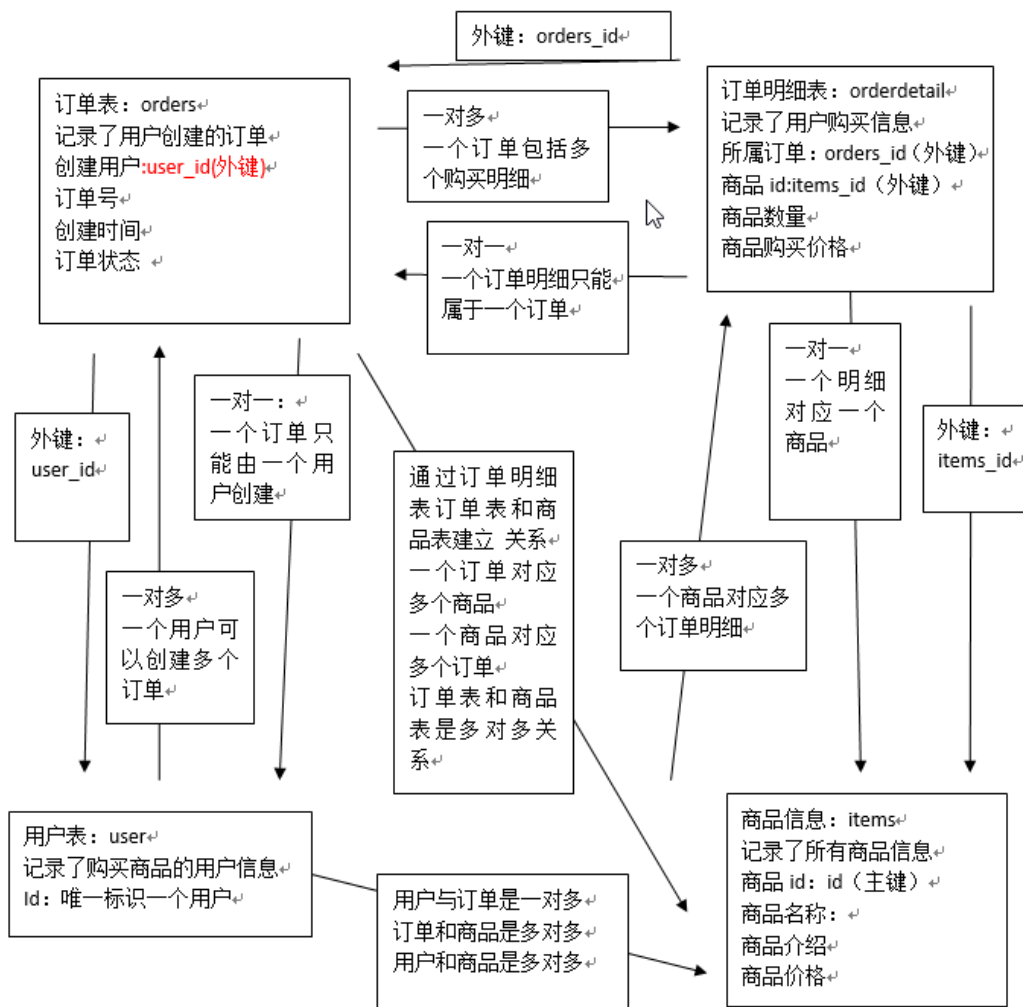
```

Public void testselectUserByArray()throws Exception{
    //获取session
    SqlSession session = sqlSessionSessionFactory.openSession();
    //获取mapper接口实例
    UserMapper userMapper = session.getMapper(UserMapper.class);
    //构造查询条件List
    Object[] userlist = new Object[2];
    User user = new User();
    user.setId(1);
    userlist[0]=user;
    user = new User();
    user.setId(2);
    userlist[1]=user;
    //传递user对象查询用户列表
    List<User>list = userMapper.selectUserByArray(userlist);
    //关闭session
    session.close();
}

```

八、关联查询

商品订单数据模型



1、一对一查询

案例：查询所有订单信息，关联查询下单用户信息。

注意：因为一个订单信息只会是一个用户下的订单，所以从查询订单信息出发关联查询用户信息为一对一查询。如果从用户信息出发查询用户下的订单信息则为一对多查询，因为一个用户可以下多个订单。

1.1、方法一：

使用resultType，定义订单信息po类，此po类中包括了订单信息和用户信息：

1) Sql语句：

```
SELECT
orders.*,
    user.username,
userss.address
FROM
orders,
    user
WHERE orders.user_id = user.id
```

2) 定义po类

Po类中应该包括上边sql查询出来的所有字段，如下：

```
public class OrdersCustom extends Orders {

    private String username;// 用户名称
    private String address;// 用户地址
    get/set。。。。
}
```

OrdersCustom类继承Orders类后OrdersCustom类包括了Orders类的所有字段，只需要定义用户的信息字段即可。

3) Mapper.xml

```
<!-- 查询所有订单信息 -->
<select id="findOrdersList" resultType="cn.itcast.mybatis.po.OrdersCustom">
    SELECT
    orders.*,
    user.username,
    user.address
    FROM
    orders, user
    WHERE orders.user_id = user.id
</select>
```

4) Mapper接口：

```
public List<OrdersCustom> findOrdersList() throws Exception;
```

5) 测试：

```
Public void testfindOrdersList()throws Exception{
    //获取session
    SqlSession session = sqlSessionSessionFactory.openSession();
}
```

```

//获取mapper接口实例
UserMapper userMapper = session.getMapper(UserMapper.class);
//查询订单信息
List<OrdersCustom> list = userMapper.findOrdersList();
System.out.println(list);
//关闭session
session.close();
}

```

6) 总结:

定义专门的po类作为输出类型，其中定义了sql查询结果集所有的字段。此方法较为简单，企业中使用普遍。

1.2、方法二:

使用resultMap，定义专门的resultMap用于映射一对一查询结果。

1) Sql语句:

```

SELECT
orders.*,
user.username,
user.address
FROM
orders,
user
WHERE orders.user_id = user.id

```

3) 定义po类

在Orders类中加入User属性，user属性中用于存储关联查询的用户信息，因为订单关联查询用户是一对一关系，所以这里使用单个User对象存储关联查询的用户信息。

3) Mapper.xml

```

<select id="findOrdersListResultMap" resultMap="userordermap">
    SELECT
    orders.*,
    user.username,
    user.address
    FROM
    orders, user
    WHERE orders.user_id = user.id
</select>

```

这里resultMap指定userordermap。

4) 定义resultMap

需要关联查询映射的是用户信息，使用association将用户信息映射到订单对象的用户属性中。

```

<!-- 订单信息resultmap -->
<resultMap type="cn.itcast.mybatis.po.Orders" id="userordermap">
    <!-- 这里的id，是mybatis在进行一对一查询时将user字段映射为用户对象时要使用，必须写 -->
    <id property="id" column="id"/>
    <result property="user_id" column="user_id"/>
    <result property="number" column="number"/>

```

```

<association property="user" javaType="cn.itcast.mybatis.po.User">
<!-- 这里的id为用户的id, 如果写上表示给user的id属性赋值 -->
<id property="id" column="user_id"/>
<result property="username" column="username"/>
<result property="address" column="address"/>
</association>
</resultMap>

```

association: 表示进行关联查询单条记录

property: 表示关联查询的结果存储在cn.itcast.mybatis.po.Orders的user属性中

javaType: 表示关联查询的结果类型

<id property="id" column="user_id"/>: 查询结果的user_id列对应关联对象的id属性, 这里是<id />表示user_id是关联查询对象的唯一标识。

<result property="username" column="username"/>: 查询结果的username列对应关联对象的username属性。

5) Mapper接口:

```
public List<Orders> findOrdersListResultMap() throws Exception;
```

6) 测试:

```

Public void testfindOrdersListResultMap()throws Exception{
    //获取session
    SqlSession session = sqlSessionFactory.openSession();
    //获取mapper接口实例
    UserMapper userMapper = session.getMapper(UserMapper.class);
    //查询订单信息
    List<Orders> list = userMapper.findOrdersList2();
    System.out.println(list);
    //关闭session
    session.close();
}

```

7) 小结:

使用association完成关联查询, 将关联查询信息映射到pojo对象中。

2、一对多查询

案例: 查询所有订单信息及订单下的订单明细信息。

订单信息与订单明细为一对多关系。

使用resultMap实现如下:

1) Sql语句:

```

SELECT
orders.*,
user.username,
user.address,
orderdetail.id orderdetail_id,
orderdetail.items_id,

```

```
orderdetail.items_num
FROM
orders,user,orderdetail
```

```
WHERE orders.user_id = user.id
```

```
AND orders.id = orderdetail.orders_id
```

```
<!-- 查询订单关联查询用户及订单明细，使用resultmap -->
```

```
<select id="findOrdersAndOrderDetailResultMap" resultMap="OrdersAndOrderDetailResultMap">
```

```
SELECT
```

```
orders.*,
```

```
USER.username,
```

```
USER.sex,
```

```
USER.address,
```

```
orderdetail.id orderdetail_id,
```

使用关联时需要有唯一标识

```
orderdetail.items_id,
```

```
orderdetail.items_num,
```

```
orderdetail.orders_id
```

```
FROM
```

```
orders,
```

```
USER,
```

```
orderdetail
```

```
WHERE orders.user_id = user.id AND orderdetail.orders_id=orders.id
```

```
</select>
```

2) 定义po类

在Orders类中加入User属性。

在Orders类中加入List<Orderdetail> orderdetails属性

3) Mapper.xml

```
<select id="findOrdersDetailList" resultMap="userorderdetailmap">
```

```
SELECT
```

```
orders.*,
```

```
user.username,
```

```
user.address,
```

```
orderdetail.id orderdetail_id,
```

```
orderdetail.items_id,
```

```
orderdetail.items_num
```

```
FROM orders,user,orderdetail
```

```
WHERE orders.user_id = user.id
```

```
AND orders.id = orderdetail.orders_id
```

```
</select>
```

4) 定义resultMap

```
<!-- 订单信息resultmap -->
```

```
<resultMap type="cn.itcast.mybatis.po.Orders" id="userorderdetailmap">
```

```
<id property="id" column="id"/>
```

```
<result property="user_id" column="user_id"/>
```

```
<result property="number" column="number"/>
```

```
<association property="user" javaType="cn.itcast.mybatis.po.User">
```

```
<id property="id" column="user_id"/>
```

```
<result property="username" column="username"/>
```

```
<result property="address" column="address"/>
```

```
</association>
```

```
<collection property="orderdetails" ofType="cn.itcast.mybatis.po.Orderdetail">
```

```

<id property="id" column="orderdetail_id"/>
<result property="items_id" column="items_id"/>
<result property="items_num" column="items_num"/>
</collection>
</resultMap>

```

黄色部分和上边一对一查询订单及用户信息定义的resultMap相同，

collection部分定义了查询订单明细信息。

collection: 表示关联查询结果集

property="orderdetails": 关联查询的结果集存储在cn.itcast.mybatis.po.Orders上哪个属性。

ofType="cn.itcast.mybatis.po.Orderdetail": 指定关联查询的结果集中的对象类型即List中的对象类型。

<id />及<result/>的意义同一对一查询。

5) resultMap使用继承

上边定义的resultMap中黄色部分和一对一查询订单信息的resultMap相同，这里使用继承可以不再填写重复的内容，如下：

```

<resultMap type="cn.itcast.mybatis.po.Orders" id="userorderdetailmap" extends="userordermap">
<collection property="orderdetails" ofType="cn.itcast.mybatis.po.Orderdetail">
  <id property="id" column="orderdetail_id"/>
  <result property="items_id" column="items_id"/>
  <result property="items_num" column="items_num"/>
</collection>
</resultMap>

```

一对多查询配置

使用extends继承订单信息userordermap。

<!-- 订单及订单明细的resultMap

使用extends继承，不用在中配置订单信息和用户信息的映射

-->

```

<resultMap type="cn.itcast.mybatis.po.Orders" id="OrdersAndOrderDetailResultMap" extends="">
  <!-- 订单信息 -->

```

```

<id column="id" property="id"/>
<result column="user_id" property="userId"/>
<result column="number" property="number"/>
<result column="createtime" property="createtime"/>
<result column="note" property="note"/>

```

通过继承，可以删除这部分

<!-- 配置映射的关联的用户信息 -->

<!-- association: 用于映射关联查询单个对象的信息

property: 要将关联查询的用户信息映射到Orders中哪个属性

-->

<!-- 用户信息 -->

```

<association property="user" javaType="cn.itcast.mybatis.po.User">
  <!-- id: 关联查询用户的唯一标识
  column: 指定唯一标识用户信息的列
  javaType: 映射到user的哪个属性
  -->
  <id column="user_id" property="id"/>
  <result column="username" property="username"/>
  <result column="sex" property="sex"/>
  <result column="address" property="address"/>
</association>

```



```

<!-- 订单明细信息
一个订单关联查询出了多条明细，要使用collection进行映射
collection: 对关联查询到多条记录映射到集合对象中
property: 将关联查询到多条记录映射到cn.itcast.mybatis.po.Orders哪个属性
ofType: 指定映射到list集合属性中pojo的类型
-->
<collection property="orderdetails" ofType="cn.itcast.mybatis.po.Orderdetail">
  <!-- id: 订单明细唯一标识
  property: 要将订单明细的唯一标识映射到cn.itcast.mybatis.po.Orderdetail的哪个属性
  -->
  <id column="orderdetail_id" property="id"/>
  <result column="items_id" property="itemsId"/>
  <result column="items_num" property="itemsNum"/>
  <result column="orders_id" property="ordersId"/>
</collection>

```

6) Mapper接口:

```
public List<Orders>findOrdersDetaillist () throws Exception;
```

7) 测试:

```

Public void testfindOrdersDetaillist()throws Exception{
    //获取session
    SqlSession session = sqlSessionFactory.openSession();
    //获取mapper接口实例
    UserMapper userMapper = session.getMapper(UserMapper.class);
    //查询订单信息
    List<Orders> list = userMapper.findOrdersDetaillist();
    System.out.println(list);
    //关闭session
    session.close();
}

```

8) 小结

使用collection完成关联查询，将关联查询信息映射到集合对象。

3、多对多查询

查询用户购买的商品信息

1) 需求

查询用户购买的商品信息。

2) sql

需要查询所有用户信息，关联查询订单及订单明细信息，订单明细信息中关联查询商品信息

```

SELECT
    orders.*,

```

```

USER .username,
USER .address,
orderdetail.id orderdetail_id,
orderdetail.items_id,
orderdetail.items_num,
items.name items_name,
items.detail items_detail
FROM
    orders,
    USER,
    orderdetail,
    items
WHERE
    orders.user_id = USER .id
AND orders.id = orderdetail.orders_id
AND orderdetail.items_id = items.id

```

```
<select id="findUserAndItemsResultMap" resultMap="UserAndItemsResultMap">
```

```

SELECT
    orders.*,
    USER.username,
    USER.sex,
    USER.address,
    orderdetail.id orderdetail_id,
    orderdetail.items_id,
    orderdetail.items_num,
    orderdetail.orders_id,
    items.name items_name,
    items.detail items_detail,
    items.price items_price|
FROM
    orders,
    USER,
    orderdetail,
    items
WHERE orders.user_id = user.id AND orderdetail.orders_id=orders.id AND orderdetail.items_id = items.id

```

SQL

```
</select>
```

3) po定义

在 `User` 中添加 `List<Orders> orders` 属性，在 `Orders` 类中加入 `List<Orderdetail> orderdetails` 属性

映射思路

将用户信息映射到 user 中

在 user 类中添加订单列表属性 `List<Orders> orderslist`，将用户创建的订单映射到 `orderslist`

在 Orders 中添加订单明细列表属性 `List<OrderDetail> orderdetails`，将订单的明细映射到 `orderdetails`

在 `OrderDetail` 中添加 `Items` 属性，将订单明细所对应的商品映射到 `Items`

4) resultMap

需要关联查询映射的信息是：订单、订单明细、商品信息

订单：一个用户对应多个订单，使用 `collection` 映射到用户对象的订单列表属性中

订单明细：一个订单对应多个明细，使用 `collection` 映射到订单对象中的明细属性中

商品信息：一个订单明细对应一个商品，使用 `association` 映射到订单明细对象的商品属性中。

```
<!-- 一对多查询
```

```
    查询用户信息、关联查询订单、订单明细信息、商品信息
```

```
-->
```

```
<resultMap type="cn.itcast.mybatis.po.User" id="userOrderListResultMap">
```

```

<id column="user_id" property="id"/>
<result column="username" property="username"/>
<collection property="orders" ofType="cn.itcast.mybatis.po.Orders">
  <id column="id" property="id"/>
  <result property="number" column="number"/>
  <collection property="orderdetails" ofType="cn.itcast.mybatis.po.Orderdetail">
    <id column="orderdetail_id" property="id"/>
    <result property="ordersId" column="id"/>
    <result property="itemsId" column="items_id"/>
    <result property="itemsNum" column="items_num"/>
    <association property="items" javaType="cn.itcast.mybatis.po.Items">
      <id column="items_id" property="id"/>
      <result column="items_name" property="name"/>
      <result column="items_detail" property="detail"/>
    </association>
  </collection>
</collection>
</resultMap>

```

<!-- 查询用户及购买的商品 -->

<resultMap type="cn.itcast.mybatis.po.User" id="UserAndItemsResultMap">

<!-- 用户信息 -->

```

<id column="user_id" property="id"/>
<result column="username" property="username"/>
<result column="sex" property="sex"/>
<result column="address" property="address"/>

```

第一步：映射用户信息

<!-- 订单信息

一个用户对应多个订单，使用collection映射

-->

```

<collection property="ordersList" ofType="cn.itcast.mybatis.po.Orders">
  <id column="id" property="id"/>
  <result column="user_id" property="userId"/>
  <result column="number" property="number"/>
  <result column="createtime" property="createtime"/>
  <result column="note" property="note"/>

```

第二步：映射订单信息

<!-- 订单明细

一个订单包括多个明细

-->

```

<collection property="orderdetails" ofType="cn.itcast.mybatis.po.Orderdetail">
  <id column="orderdetail_id" property="id"/>
  <result column="items_id" property="itemsId"/>
  <result column="items_num" property="itemsNum"/>
  <result column="orders_id" property="ordersId"/>

```

<!-- 商品信息

一个订单明细对应一个商品

-->

```

<association property="items" javaType="cn.itcast.mybatis.po.Items">
  <id column="items_id" property="id"/>
  <result column="items_name" property="name"/>
  <result column="items_detail" property="detail"/>
  <result column="items_price" property="price"/>
</association>

```

</collection>

</collection>

</resultMap>

一对多是多对多的特例，如下需求：

查询用户购买的商品信息，用户和商品的关系是多对多关系。

需求1：

查询字段：用户账号、用户名称、用户性别、商品名称、商品价格(最常见)

企业开发中常见明细列表，用户购买商品明细列表，

使用resultType将上边查询列映射到pojo输出。

需求2：

查询字段：用户账号、用户名称、购买商品数量、商品明细（鼠标移上显示明细）

使用resultMap将用户购买商品明细列表映射到user对象中。

6) resultMap小结

resultType:

作用：

将查询结果按照sql列名pojo属性名一致性映射到pojo中。

场合：

常见一些明细记录的展示，比如用户购买商品明细，将关联查询信息全部展示在页面时，此时可直接使用resultType将每一条记录映射到pojo中，在前端页面遍历list（list中是pojo）即可。

resultMap:

使用association和collection完成一对一和一对多高级映射（对结果有特殊的映射要求）。

association:

作用：

将关联查询信息映射到一个pojo对象中。

场合：

为了方便查询关联信息可以使用association将关联订单信息映射为用户对象的pojo属性中，比如：查询订单及关联用户信息。

使用resultType无法将查询结果映射到pojo对象的pojo属性中，根据对结果集查询遍历的需要选择使用resultType还是resultMap。

collection:

作用：

将关联查询信息映射到一个list集合中。

场合：

为了方便查询遍历关联信息可以使用collection将关联信息映射到list集合中，比如：查询用户权限范围模块及模块下的菜单，可使用collection将模块映射到模块list中，将菜单列表映射到模块对象的菜单list属性中，这样的作的目的也是方便对查询结果集进行遍历查询。

如果使用resultType无法将查询结果映射到list集合中。

7) 延迟加载

需要查询关联信息时，使用mybatis延迟加载特性可有效的减少数据库压力，首次查询只查询主要信息，关联信息等用户获取时再加载。

```
<!-- 实现对用户信息进行延迟加载
select: 指定延迟加载需要执行的statement的id (是根据user_id查询用户信息的statement)
要使用userMapper.xml中findUserById完成根据用户id(user_id)用户信息的查询, 如果findUserById不在本mapper中需要前边加namespace
column: 订单信息中关联用户信息查询的列, 是user_id
关联查询的sql理解为:
SELECT orders.*,
(SELECT username FROM USER WHERE orders.user_id = user.id)username,
(SELECT sex FROM USER WHERE orders.user_id = user.id)sex
FROM orders
-->
<association property="user" javaType="cn.itcast.mybatis.po.User"
select="cn.itcast.mybatis.mapper.UserMapper.findUserById" column="user_id">
<!-- 实现对用户信息进行延迟加载 -->

</association>
```

5.6.1 打开延迟加载开关

在mybatis核心配置文件中配置：
lazyLoadingEnabled、aggressiveLazyLoading

设置项	描述	允许值	默认值
lazyLoadingEnabled	全局性设置懒加载。如果设为‘false’，则所有相关联的都会被初始化加载。	true false	false
aggressiveLazyLoading	当设置为‘true’的时候，懒加载的对象可能任何懒属性全部加载。否则，每个属性都按需加载。	true false	true

```
<settings>
    <setting name="lazyLoadingEnabled" value="true"/>
    <setting name="aggressiveLazyLoading" value="false"/>
</settings>
```

5.6.2 一对一查询延迟加载

5.6.2.1 需求

查询订单信息，关联查询用户信息。
默认只查询订单信息，当需要查询用户信息时再去查询用户信息。

需要定义两个 mapper 的方法对应的 statement。
1、只查询订单信息
SELECT * FROM orders
在查询订单的 statement 中使用 association 去延迟加载（执行）下边的 statement(关联查询用户信息)

```
<!-- 查询订单关联查询用户，用户信息需要延迟加载 -->
<select id="findOrdersUserLazyLoading" resultMap="OrdersUserLazyLoadingResultMap">
    SELECT * FROM orders
</select>
```

2、关联查询用户信息
通过上边查询到的订单信息中 user_id 去关联查询用户信息
使用 UserMapper.xml 中的 findUserById
<select id="findUserById" parameterType="int" resultType="user">
 SELECT * FROM USER WHERE id=#{value}
</select>

SELECT orders.*,
 (SELECT username FROM USER WHERE orders.user_id = user.id)username,
 (SELECT sex FROM USER WHERE orders.user_id = user.id)sex,
 FROM orders

5.6.2.2 Sql语句:

SELECT
orders.*

```
FROM
orders
```

5.6.2.3 定义po类

在Orders类中加入User属性。

5.6.2.4 Mapper.xml

```
<select id="findOrdersList3" resultMap="userordermap2">
    SELECT
    orders.*
    FROM
    orders
</select>
```

5.6.2.5 定义resultMap

```
<!-- 订单信息resultmap -->
<resultMap type="cn.itcast.mybatis.po.Orders" id="userordermap2">
<id property="id" column="id"/>
<result property="user_id" column="user_id"/>
<result property="number" column="number"/>
<association property="user" javaType="cn.itcast.mybatis.po.User" select="findUserById" column="user_id"/>
</resultMap>
```

association:

select="findUserById": 指定关联查询sql为findUserById

column="user_id": 关联查询时将users_id列的值传入findUserById

最后将关联查询结果映射至cn.itcast.mybatis.po.User。

5.6.2.6 Mapper接口:

```
public List<Orders> findOrdersList3() throws Exception;
```

5.6.2.7 测试:

```
Public void testfindOrdersList3()throws Exception{
    //获取session
    SqlSession session = sqlSessionSessionFactory.openSession();
    //获取mapper接口实例
    UserMapper userMapper = session.getMapper(UserMapper.class);
    //查询订单信息
    List<Orders> list = userMapper.findOrdersList3();
    System.out.println(list);
    //开始加载,通过orders.getUser方法进行加载
    for(Orders orders:list){
        System.out.println(orders.getUser());
    }
    //关闭session
    session.close();
}
```

5.6.2.8 延迟加载的思考

不使用mybatis提供的延迟加载功能是否可以实现延迟加载？

实现方法：

针对订单和用户两个表定义两个mapper方法。

- 1、订单查询mapper方法
- 2、根据用户id查询用户信息mapper方法

默认使用订单查询mapper方法只查询订单信息。

当需要关联查询用户信息时再调用根据用户id查询用户信息mapper方法查询用户信息。

5.6.3 一对多延迟加载

一对多延迟加载的方法同一对一延迟加载，在collection标签中配置select内容。

本部分内容自学。

5.6.4 延迟加载小结

作用：

当需要查询关联信息时再去数据库查询，默认不去关联查询，提高数据库性能。

只有使用resultMap支持延迟加载设置。

场合：

当只有部分记录需要关联查询其它信息时，此时可按需延迟加载，需要关联查询时再向数据库发出sql，以提高数据库性能。

当全部需要关联查询信息时，此时不用延迟加载，直接将关联查询信息全部返回即可，可使用resultType或resultMap完成映射。

6 查询缓存

6.1 mybatis缓存介绍

如下图，是mybatis一级缓存和二级缓存的区别图解：



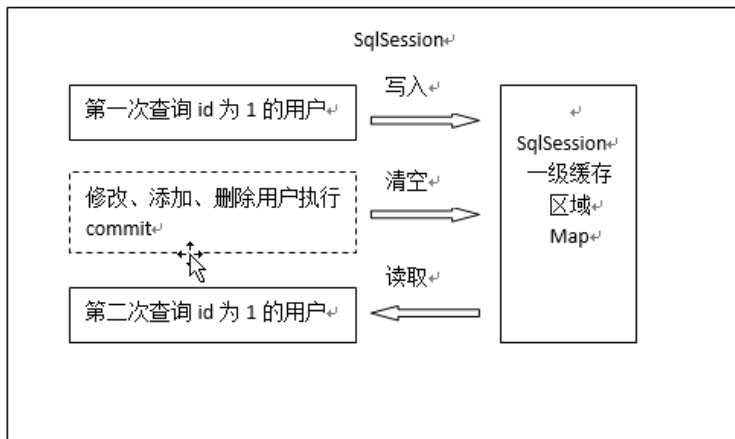
Mybatis一级缓存的作用域是同一个SqlSession，在同一个sqlSession中两次执行相同的sql语句，第一次执行完毕会将数据库中查询的数据写到缓存（内存），第二次会从缓存中获取数据将不再从数据库查询，从而提高查询效率。当一个sqlSession结束后该sqlSession中的一级缓存也就不存在了。Mybatis默认开启一级缓存。

Mybatis二级缓存是多个SqlSession共享的，其作用域是mapper的同一个namespace，不同的sqlSession两次执行相同namespace下的sql语句且向sql中传递参数也相同即最终执行相同的sql语句，第一次执行完毕会将数据库中查询的数据写到缓存（内存），第二次会从缓存中获取数据将不再从数据库查询，从而提高查询效率。Mybatis默认没有开启二级缓存需要在setting全局参数中配置开启二级缓存。

6.2 一级缓存

6.2.1 原理

下图是根据id查询用户的一级缓存图解：



一级缓存区域是根据SqlSession为单位划分的。

每次查询会先从缓存区域找，如果找不到从数据库查询，查询到数据将数据写入缓存。

Mybatis内部存储缓存使用一个HashMap，key为hashCode+sqlId+Sql语句。value为从查询出来映射生成的java对象
sqlSession执行insert、update、delete等操作commit提交后会清空缓存区域。

6.2.2 测试1

```
//获取session
SqlSession session = sqlSessionFactory.openSession();
//获取mapper接口实例
UserMapper userMapper = session.getMapper(UserMapper.class);
//第一次查询
User user1 = userMapper.findUserById(1);
System.out.println(user1);
//第二次查询，由于是同一个session则不再向数据库发出语句直接从缓存取出
User user2 = userMapper.findUserById(1);
System.out.println(user2);
//关闭session
session.close();
```

6.2.3 测试2

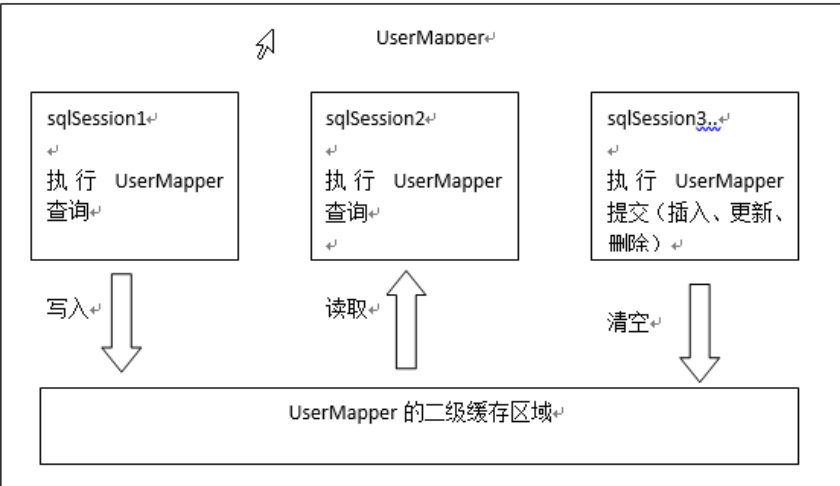
```
//获取session
SqlSession session = sqlSessionFactory.openSession();
//获取mapper接口实例
UserMapper userMapper = session.getMapper(UserMapper.class);
//第一次查询
User user1 = userMapper.findUserById(1);
System.out.println(user1);
//在同一个session执行更新
User user_update = new User();
user_update.setId(1);
user_update.setUsername("李奎");
userMapper.updateUser(user_update);
session.commit();
//第二次查询，虽然是同一个session但是由于执行了更新操作session的缓存被清空，这里重新发出sql操作
User user2 = userMapper.findUserById(1);
System.out.println(user2);
```


九、MyBatis缓存

6.3 二级缓存

6.3.1 原理

下图是多个 sqlSession 请求 UserMapper 的二级缓存图解。



二级缓存区域是根据mapper的namespace划分的，相同namespace的mapper查询数据放在同一个区域，如果使用mapper代理方法每个mapper的namespace都不同，此时可以理解为二级缓存区域是根据mapper划分。

每次查询会先从缓存区域找，如果找不到从数据库查询，查询到数据将数据写入缓存。

Mybatis内部存储缓存使用一个HashMap，key为hashCode+sqlId+Sql语句。value为从查询出来映射生成的java对象
sqlSession执行insert、update、delete等操作commit提交后会清空缓存区域。

首先开启 `mybatis` 的二级缓存。

`sqlSession1` 去查询用户 id 为 1 的用户信息，查询到用户信息会将查询数据存储到二级缓存中。

`sqlSession2` 去查询用户 id 为 1 的用户信息，去缓存中找是否存在数据，如果存在直接从缓存中取出数据。

二级缓存与一级缓存区别，二级缓存的范围更大，多个 `sqlSession` 可以共享一个 `UserMapper` 的二级缓存区域。
`UserMapper` 有一个二级缓存区域（按 namespace 分），其它 `mapper` 也有自己的二级缓存区域（按 namespace 分）。

每一个 namespace 的 `mapper` 都有一个二缓存区域，两个 `mapper` 的 namespace 如果相同，这两个 `mapper` 执行 `sql` 查询到数据将存在相同的二级缓存区域

6.3.2 开启二级缓存：

1、在核心配置文件SqlMapConfig.xml中加入

```
<setting name="cacheEnabled" value="true"/>
```

	描述	允许值	默认值
cacheEnabled	对在此配置文件下的所有cache 进行全局性开/关设置。	true false	true

2、要在你的Mapper映射文件中添加一行：`<cache />`，表示此mapper开启二级缓存。

6.3.3 实现序列化

二级缓存需要查询结果映射的pojo对象实现java.io.Serializable接口实现序列化和反序列化操作，注意如果存在父类、成员pojo都需要实现序列化接口。

```
public class Orders implements Serializable
public class User implements Serializable
....
```

6.3.4 测试

```
//获取session1
SqlSession session1 = sqlSessionFactory.openSession();
UserMapper userMapper = session1.getMapper(UserMapper.class);
//使用session1执行第一次查询
User user1 = userMapper.findUserById(1);
System.out.println(user1);
//关闭session1
session1.close();
//获取session2
SqlSession session2 = sqlSessionFactory.openSession();
UserMapper userMapper2 = session2.getMapper(UserMapper.class);
//使用session2执行第二次查询，由于开启了二级缓存这里从缓存中获取数据不再向数据库发出sql
User user2 = userMapper2.findUserById(1);
System.out.println(user2);
//关闭session2
session2.close();

// 创建代理对象
UserMapper userMapper1 = sqlSession1.getMapper(UserMapper.class);
// 第一次发起请求，查询id为1的用户
User user1 = userMapper1.findUserById(1);
System.out.println(user1);
//这里执行关闭操作，将sqlsession中的数据写到二级缓存区域
sqlSession1.close();
```

6.3.5 禁用二级缓存

在statement中设置useCache=false可以禁用当前select语句的二级缓存，即每次查询都会发出sql去查询，默认情况是true，即该sql使用二级缓存。

```
<select id="findOrderListResultMap" resultMap="ordersUserMap" useCache="false">
```

6.3.6 刷新缓存

在mapper的同一个namespace中，如果有其它insert、update、delete操作数据后需要刷新缓存，如果不执行刷新缓存会出现脏读。

设置statement配置中的flushCache="true" 属性，默认情况下为true即刷新缓存，如果改成false则不会刷新。使用缓存时如果手动修改数据库表中的查询数据会出现脏读。

如下：

```
<insert id="insertUser" parameterType="cn.itcast.mybatis.po.User" flushCache="true">
```

6.3.7 Mybatis Cache参数

`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`。

如下例子：

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

这个更高级的配置创建了一个 FIFO 缓存,并每隔 60 秒刷新,存数结果对象或列表的 512 个引用,而且返回的对象被认为是只读的,因此在不同线程中的调用者之间修改它们会导致冲突。可用的回收策略有,默认的是 LRU:

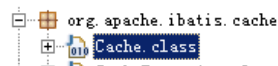
- 1.LRU - 最近最少使用的:移除最长时间不被使用的对象。
- 2.FIFO - 先进先出:按对象进入缓存的顺序来移除它们。
- 3.SOFT - 软引用:移除基于垃圾回收器状态和软引用规则的对象。
- 4.WEAK - 弱引用:更积极地移除基于垃圾收集器状态和弱引用规则的对象。

6.3.8 mybatis整合ehcache

EhCache 是一个纯Java的进程内缓存框架，是一种广泛使用的开源Java分布式缓存，具有快速、精干等特点，是Hibernate中默认的CacheProvider。

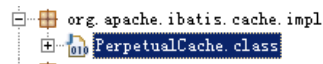
6.3.8.1 mybatis整合ehcache原理

mybatis提供二级缓存Cache接口，如下：



```
org.apache.ibatis.cache
├── Cache.class
```

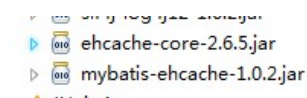
它的默认实现类：



```
org.apache.ibatis.cache.impl
├── PerpetualCache.class
```

通过实现Cache接口可以实现mybatis缓存数据通过其它缓存数据库整合，mybatis的特长是sql操作，缓存数据的管理不是mybatis的特长，为了提高缓存的性能将mybatis和第三方的缓存数据库整合，比如ehcache、memcache、redis等。

6.3.8.2 第一步：引入缓存的依赖包



```
org.mybatis.caches
├── ehcache-core-2.6.5.jar
└── mybatis-ehcache-1.0.2.jar
```

maven坐标：

```
<dependency>
  <groupId>org.mybatis.caches</groupId>
  <artifactId>mybatis-ehcache</artifactId>
  <version>1.0.2</version>
</dependency>
```

6.3.8.3 第二步：引入缓存配置文件

classpath下添加：ehcache.xml

内容如下：

```
<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="../config/ehcache.xsd">
```

```

<diskStore path="F:\develop\ehcache"/>
<defaultCache
    maxElementsInMemory="1000"
    maxElementsOnDisk="10000000"
    eternal="false"
    overflowToDisk="false"
    timeToIdleSeconds="120"
    timeToLiveSeconds="120"
    diskExpiryThreadIntervalSeconds="120"
    memoryStoreEvictionPolicy="LRU">
</defaultCache>
</ehcache>

```

属性说明:

- **diskStore**: 指定数据在磁盘中的存储位置。
- **defaultCache**: 当借助CacheManager.add("demoCache")创建Cache时, EhCache便会采用<defaultCache/>指定的管理策略

以下属性是必须的:

- **maxElementsInMemory** - 在内存中缓存的element的最大数目
- **maxElementsOnDisk** - 在磁盘上缓存的element的最大数目, 若是0表示无穷大
- **eternal** - 设定缓存的elements是否永远不过期。如果为true, 则缓存的数据始终有效, 如果为false那么还要根据timeToIdleSeconds, timeToLiveSeconds判断
- **overflowToDisk** - 设定当内存缓存溢出的时候是否将过期的element缓存到磁盘上

以下属性是可选的:

- **timeToIdleSeconds** - 当缓存在EhCache中的数据前后两次访问的时间超过timeToIdleSeconds的属性取值时, 这些数据便会删除, 默认值是0, 也就是可闲置时间无穷大
- **timeToLiveSeconds** - 缓存element的有效生命期, 默认是0, 也就是element存活时间无穷大
- **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都应该有一个缓冲区。
- **diskPersistent** - 在VM重启的时候是否启用磁盘保存EhCache中的数据, 默认是false。
- **diskExpiryThreadIntervalSeconds** - 磁盘缓存的清理线程运行间隔, 默认是120秒。每个120s, 相应的线程会进行一次EhCache中数据的清理工作
- **memoryStoreEvictionPolicy** - 当内存缓存达到最大, 有新的element加入的时候, 移除缓存中element的策略。默认是LRU (最近最少使用), 可选的有LFU (最不常使用) 和FIFO (先进先出)

6.3.8.4 第三步: 开启ehcache缓存

EhcacheCache是ehcache对Cache接口的实现:

```

mybatis-ehcache-1.0.2.jar
├── org.mybatis.caches.ehcache
│   └── EhcacheCache.class

```

修改mapper.xml文件, 在cache中指定EhcacheCache。

```

<cache type="org.mybatis.caches.ehcache.EhcacheCache"/>

<!-- 开启本mapper的namespace下的二缓存
type: 指定cache接口的实现类的类型, mybatis默认使用PerpetualCache
要和ehcache整合, 需要配置type为ehcache实现cache接口的类型
-->
<cache type="org.mybatis.caches.ehcache.EhcacheCache"/>

```

根据需求调整缓存参数:

```

<cache type="org.mybatis.caches.ehcache.EhcacheCache">
    <property name="timeToIdleSeconds" value="3600"/>
    <property name="timeToLiveSeconds" value="3600"/>
    <!-- 同ehcache参数maxElementsInMemory -->
    <property name="maxEntriesLocalHeap" value="1000"/>
    <!-- 同ehcache参数maxElementsOnDisk -->
    <property name="maxEntriesLocalDisk" value="10000000"/>
    <property name="memoryStoreEvictionPolicy" value="LRU"/>
</cache>

```

6.3.9 应用场景

对于访问多的查询请求且用户对查询结果实时性要求不高，此时可采用mybatis二级缓存技术降低数据库访问量，提高访问速度，业务场景比如：耗时较高的统计分析sql、电话账单查询sql等。

实现方法如下：通过设置刷新间隔时间，由mybatis每隔一段时间自动清空缓存，根据数据变化频率设置缓存刷新间隔flushInterval，比如设置为30分钟、60分钟、24小时等，根据需求而定。

6.3.10 局限性

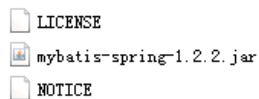
mybatis二级缓存对细粒度的数据级别的缓存实现不好，比如如下需求：对商品信息进行缓存，由于商品信息查询访问量大，但是要求用户每次都能查询最新的商品信息，此时如果使用mybatis的二级缓存就无法实现当一个商品变化时只刷新该商品的缓存信息而不刷新其它商品的信息，因为mybatis的二级缓存区域以mapper为单位划分，当一个商品信息变化会将所有商品信息的缓存数据全部清空。解决此类问题需要在业务层根据需求对数据有针对性缓存。

7与spring整合

实现mybatis与spring进行整合，通过spring管理SqlSessionFactory、mapper接口。

7.1mybatis与spring整合jar

mybatis官方提供与mybatis与spring整合jar包：



还包括其它jar：

spring3.2.0

mybatis3.2.7

dbcp连接池

数据库驱动

参考：



7.2Mybatis配置文件

在classpath下创建mybatis/SqlMapConfig.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>

<!--使用自动扫描器时，mapper.xml文件如果和mapper.java接口在一个目录则此处不用定义mappers -->
<mappers>
```

```
<package name="cn.itcast.mybatis.mapper" />
</mappers>
</configuration>
```

7.3 Spring配置文件:

在classpath下创建applicationContext.xml, 定义数据库链接池、SqlSessionFactory。

```
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xmlns:mvc="http://www.springframework.org/schema/mvc"
        xmlns:context="http://www.springframework.org/schema/context"
        xmlns:aop="http://www.springframework.org/schema/aop" xmlns:tx="http://www.springframework.org/schema/tx"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.2.xsd
        http://www.springframework.org/schema/mvc
        http://www.springframework.org/schema/mvc/spring-mvc-3.2.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-3.2.xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop-3.2.xsd
        http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/tx/spring-tx-3.2.xsd ">
<!-- 加载配置文件 -->
<context:property-placeholder location="classpath:db.properties"/>
<!-- 数据库连接池 -->
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
    <property name="driverClassName" value="${jdbc.driver}"/>
    <property name="url" value="${jdbc.url}"/>
    <property name="username" value="${jdbc.username}"/>
    <property name="password" value="${jdbc.password}"/>
    <property name="maxActive" value="10"/>
    <property name="maxIdle" value="5"/>
</bean>
<!-- mapper配置 -->
<!-- 让spring管理sqlSessionFactory 使用mybatis和spring整合包中的 -->
<bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
    <!-- 数据库连接池 -->
    <property name="dataSource" ref="dataSource" />
    <!-- 加载mybatis的全局配置文件 -->
    <property name="configLocation" value="classpath:mybatis/SqlMapConfig.xml" />
</bean>
</beans>
```

注意: 在定义sqlSessionFactory时指定数据源dataSource和mybatis的配置文件。

7.4 Mapper编写的三种方法

7.4.1 Dao接口实现类继承SqlSessionDaoSupport

使用此种方法即原始dao开发方法, 需要编写dao接口, dao接口实现类、映射文件。

1、在sqlMapConfig.xml中配置映射文件的位置

```
<mappers>
  <mapper resource="mapper.xml文件的地址" />
  <mapper resource="mapper.xml文件的地址" />
</mappers>
```

2、定义dao接口

3、dao接口实现类集成SqlSessionDaoSupport

dao接口实现类方法中可以this.getSqlSession()进行数据增删改查。

4、spring 配置

```
<bean id=" " class="mapper接口的实现">
  <property name="sqlSessionFactory" ref="sqlSessionFactory"></property>
</bean>
```

7.4.2 使用 org.mybatis.spring.mapper.MapperFactoryBean

此方法即mapper接口开发方法，只需定义mapper接口，不用编写mapper接口实现类。每个mapper接口都需要在spring配置文件中定义。

1、在sqlMapConfig.xml中配置mapper.xml的位置

如果mapper.xml和mappre接口的名称相同且在同一个目录，这里可以不用配置

```
<mappers>
  <mapper resource="mapper.xml文件的地址" />
  <mapper resource="mapper.xml文件的地址" />
</mappers>
```

2、定义mapper接口

3、Spring中定义

```
<bean id="" class="org.mybatis.spring.mapper.MapperFactoryBean">
  <property name="mapperInterface" value="mapper接口地址"/>
  <property name="sqlSessionFactory" ref="sqlSessionFactory"/>
</bean>
```

7.4.3 使用mapper扫描器

此方法即mapper接口开发方法，只需定义mapper接口，不用编写mapper接口实现类。只需要在spring配置文件中定义一个mapper扫描器，自动扫描包中的mapper接口生成代理对象。

1、mapper.xml文件编写，

2、定义mapper接口

注意mapper.xml的文件名和mapper的接口名称保持一致，且放在同一个目录

3、配置mapper扫描器

```
<bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
  <property name="basePackage" value="mapper接口包地址"></property>
  <property name="sqlSessionFactoryBeanName" value="sqlSessionFactory"/>
</bean>
```

```

<!-- mapper批量扫描，从mapper包中扫描出mapper接口，自动创建代理对象并且在spring容器中注册
遵循规范：将mapper.java和mapper.xml映射文件名称保持一致，且在一个目录中
-->
<bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
    <!-- 指定扫描的包名 -->
    <property name="basePackage" value="cn.itcast.ssm.mapper"/>
    <property name="sqlSessionFactoryBeanName" value="sqlSessionFactory"/>
    必须使用SQLSessionFactoryBeanName，否则会因为先扫描mapper文件，后加载db的properties文件
    而导致出错
</bean>

```

这里 自动扫描出来的mapper的bean的id为mapper类名（首字母小写）

basePackage: 扫描包路径，中间可以用逗号或分号分隔定义多个包

4、使用扫描器后从spring容器中获取mapper的实现对象

如果将**mapper.xml**和**mapper**接口的名称保持一致且放在一个目录 则不用在**sqlMapConfig.xml**中进行配置

```

<!-- 批量加载mapper
指定mapper接口的包名，mybatis自动扫描包下边所有mapper接口进行加载
遵循一些规范：需要将mapper接口类名和mapper.xml映射文件名称保持一致，且在一个目录中
上边规范的前提是：使用的是mapper代理方法

```

和spring整合后，使用mapper扫描器，这里不需要配置了

```

-->
<!-- <package name="cn.itcast.ssm.mapper"/> -->

```

疑问：

- 1、如果直接把数据库连接信息写在spring配置文件中，问题是否不会发生。
- 2、改用sqlSessionFactoryBeanName后，对其他有没有什么影响？

```

<!-- 使用MapperScannerConfigurer扫描mapper

```

扫描器将**mapper**扫描出来自动 注册到spring容器，bean的id是类名（首字母小写）

```

-->
<bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
    <!-- 指定扫描的包

```

如果扫描多个包中间使用半角逗号分隔

如果使用扫描器，不用在sqlmapconfig.xml中去配置**mapper**的扫描了，如果使用**mapper**代理的开发，

在SqlMapConfig.xml中不用配置**mapper**项了

```

-->
<property name="basePackage" value="cn.itcast.mybatis.mapper"/>
<!-- 使用sqlSessionFactoryBeanName注入sqlSessionFactory -->
<property name="sqlSessionFactoryBeanName" value="sqlSessionFactory"/>
<!--

```

注意： 这里使用sqlSessionFactoryBeanName而不使用sqlSessionFactory原因如下：

MapperScannerConfigurer在扫描**mapper**时需要注入 sqlSessionFactory，如果使用

```

<property name="sqlSessionFactory" ref="sqlSessionFactory"/>

```

会存在PropertyPlaceholderConfigurer还没来得及替换dataSource定义中的\${jdbc.driver}等数据源变量就注入到了MapperScannerConfigurer中，将导致数据库连接不上，如果改为如下方式可以解决问题：

来源：<https://d.docs.live.net/c1e04d8163240660/01-技术类/mybatis/mybatis第二天课堂笔记20150629.docx>

8 Mybatis逆向工程

使用官方网站的mapper自动生成工具mybatis-generator-core-1.3.2来生成po类和mapper映射文件。

8.1 第一步：mapper生成配置文件：

在generatorConfig.xml中配置mapper生成的详细信息，注意改下几点：

- 1、添加要生成的数据库表
- 2、po文件所在包路径
- 3、mapper文件所在包路径

配置文件如下：

详见generatorSqlmapCustom工程

8.2 第二步：使用java类生成mapper文件：

```
Public void generator() throws Exception{
    List<String> warnings = new ArrayList<String>();
    boolean overwrite = true;
    File configFile = new File("generatorConfig.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);
}
Public static void main(String[] args) throws Exception {
    try {
        GeneratorSqlmap generatorSqlmap = new GeneratorSqlmap();
        generatorSqlmap.generator();
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

8.3 第三步：拷贝生成的mapper文件到工程中指定的目录中

8.3.1 Mapper.xml

Mapper.xml的文件拷贝至mapper目录内

8.3.2 Mapper.java

Mapper.java的文件拷贝至mapper 目录内

注意：mapper.xml文件和mapper.java文件在一个目录内且文件名相同。

8.3.3 第四步Mapper接口测试

学会使用mapper自动生成的增、删、改、查方法。

```
//删除符合条件的记录
int deleteByExample(UserExample example);
//根据主键删除
int deleteByPrimaryKey(String id);
```

```

//插入对象所有字段
int insert(User record);
//插入对象不为空的字段
int insertSelective(User record);
//自定义查询条件查询结果集
List<User> selectByExample(UserExample example);
//根据主键查询
UserselectByPrimaryKey(String id);
//根据主键将对象中不为空的值更新至数据库
int updateByPrimaryKeySelective(User record);
//根据主键将对象中所有字段的值更新至数据库
int updateByPrimaryKey(User record);

```

8.4 逆向工程注意事项

8.4.1 Mapper文件内容不覆盖而是追加

XXXMapper.xml文件已经存在时，如果进行重新生成则mapper.xml文件内容不被覆盖而是进行内容追加，结果导致mybatis解析失败。

解决方法：删除原来已经生成的mapper.xml文件再进行生成。

Mybatis自动生成的po及mapper.java文件不是内容而是直接覆盖没有此问题。

8.4.2 Table schema问题

下边是关于针对oracle数据库表生成代码的schema问题：

Schema即数据库模式，oracle中一个用户对应一个schema，可以理解为用户就是schema。

当Oracle数据库存在多个schema可以访问相同的表名时，使用mybatis生成该表的mapper.xml将会出现mapper.xml内容重复的问题，结果导致mybatis解析错误。

解决方法：在table中填写schema，如下：

```
<table schema="XXXX" tableName=" " >
```

XXXX即为一个schema的名称，生成后将mapper.xml的schema前缀批量去掉，如果不去掉当oracle用户变更了sql语句将查询失败。

快捷操作方式：mapper.xml文件中批量替换：“from XXXX.” 为空

Oracle查询对象的schema可从dba_objects中查询，如下：

```
select * from dba_objects
```