SPRING BOOT 整合 MYBATIS

王瑞

2019-7

南京天汇智码软件技术有限公司——智码堂软件技术培训中心
南京江北新区星火路 15 号智芯大厦 1026

目录

1 概述

本文内容涉及 SpringBoot 与 MyBatis 框架的整合应用。使用注解方式来配置 SpringBoot 和 MyBatis。阅读本文之前，需具有 SpringBoot 和 MyBatis 相应知识。

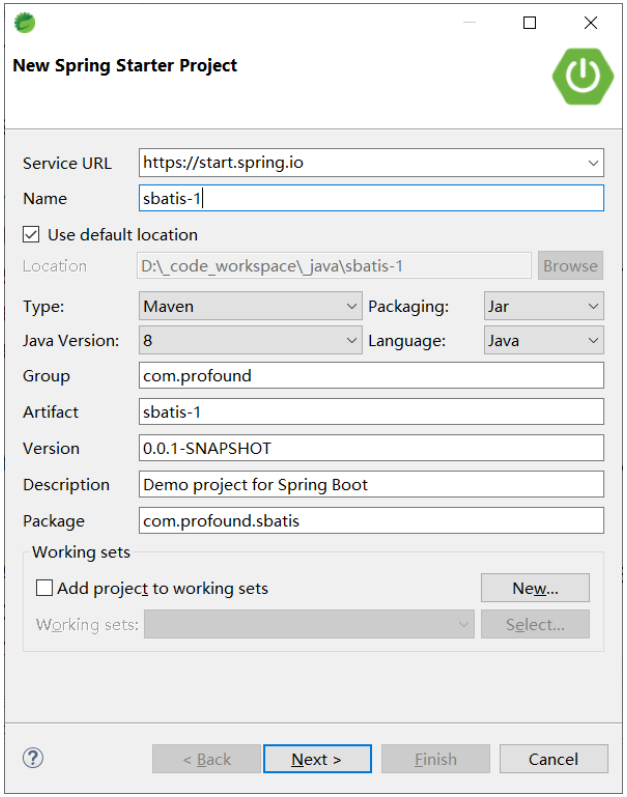
开发环境 Mariadb\STS4.4.3 整合版。

本文所使用的数据库为 Mariadb，大家可以将其它成 MySQL。它实际上就 MySQL 的兼容版本。在本示例中没有使用 MySQL 数据库驱动，而是使用了 Mariadb JDBC 驱动。

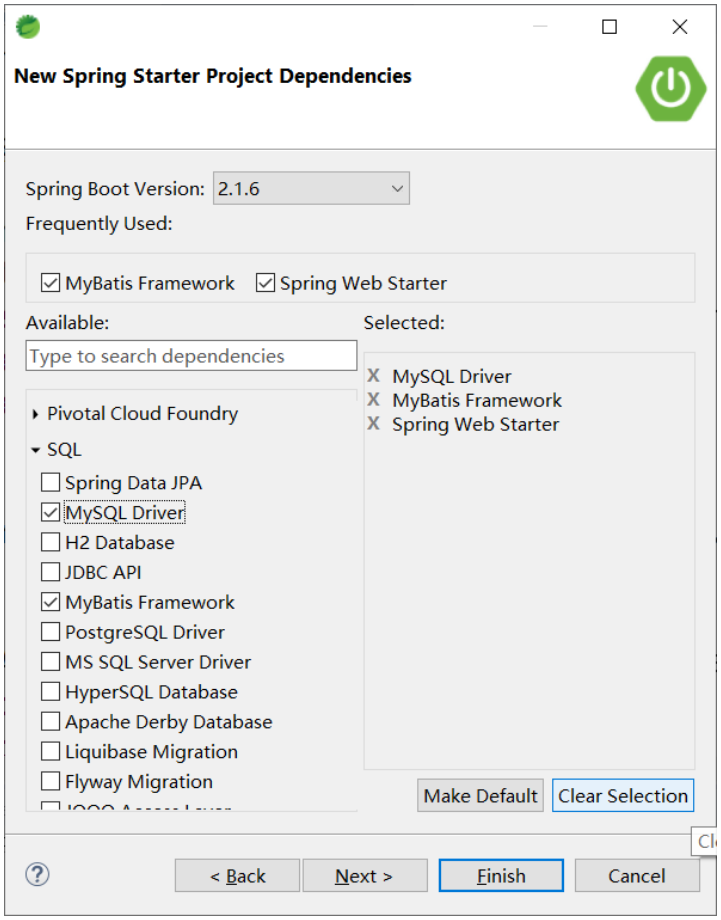
本文中实现了一个 Service Bean，在该 Bean 中，注入了 Mybatis Bean。实现了事务管理。

2 项目搭建

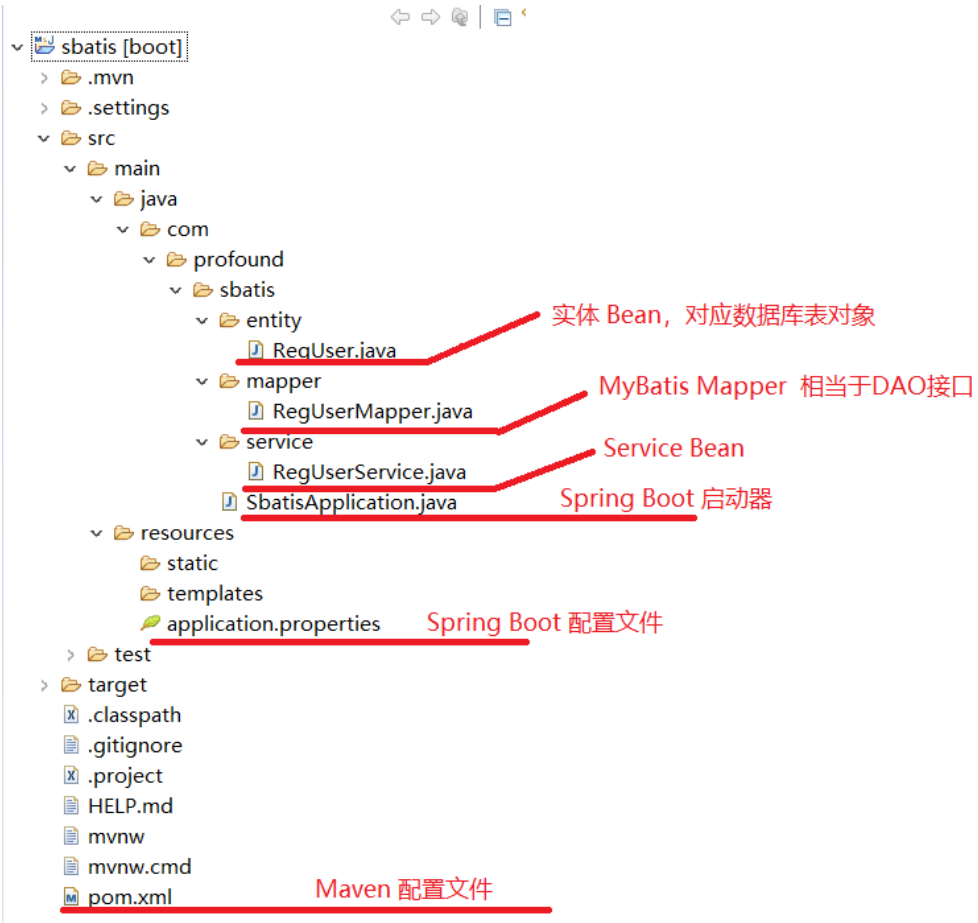
2.1创建 Spring Starter 项目。



2.2选择 Spring Web、MySQL Driver(之后修改成了 Mariadb 驱动)、MyBatis Framework。



2.3项目包的设置


sbatis [boot]
.mvn
.settings
src
main
java
com
profound
sbatis
entity
RegUser.java
实体 Bean，对应数据库表对象
mapper
RegUserMapper.java
MyBatis Mapper 相当于DAO接口
service
RegUserService.java
Service Bean
SbatisApplication.java
Spring Boot 启动器
resources
static
templates
application.properties
Spring Boot 配置文件
test
target
.classpath
.gitignore
.project
HELP.md
mvnw
mvnw.cmd
pom.xml
Maven 配置文件

2.4修改 pox.xml 文件，将 MySQ 驱动，更改为 Mariadb 驱动

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.1.6.RELEASE</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <groupId>com.profound</groupId>
    <artifactId>sbatis</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>sbatis</name>
    <description>Demo project for Spring Boot</description>

    <properties>
        <java.version>1.8</java.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
```

```
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.mybatis.spring.boot</groupId>
            <artifactId>mybatis-spring-boot-starter</artifactId>
            <version>2.0.1</version>
        </dependency>

        <!-- https://mvnrepository.com/artifact/org.mariadb.jdbc/mariadb-java-client -->
        <dependency>
            <groupId>org.mariadb.jdbc</groupId>
            <artifactId>mariadb-java-client</artifactId>
            <version>2.4.2</version>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
```

```
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>

</project>
```

3 设置数据源和驱动

修改 application.properties 文件，添加数据源所需要的相应信息。在本示例中只添加了必须的相应信息。

```
# 数据库连接 RUL
spring.datasource.url=jdbc:mariadb://127.0.0.1:3306/blog
# 数据库 JDBC 驱动
spring.datasource.driver-class-name=org.mariadb.jdbc.Driver
# 数据库用户名和密码
spring.datasource.username=blog
spring.datasource.password=blog

# 输出 SQL 语句
logging.level.com.profound.sbatis.mapper =debug
```

4 创建实体 Bean

根据数据库表中字段，创建对应的实体 Bean。在 entity 包下创建 RegUser 类。

数据库表

```
MariaDB [blog]> desc reg_user;
+----------+-------------+------+-----+---------+----------------+
| Field    | Type        | Null | Key | Default | Extra          |
+----------+-------------+------+-----+---------+----------------+
| user_id  | int(11)     | NO   | PRI | NULL    | auto_increment |
| reg_name | varchar(20) | YES  | UNI | NULL    |                |
| passwd   | varchar(12) | YES  |     | NULL    |                |
| email    | varchar(20) | YES  |     | NULL    |                |
| question | varchar(20) | YES  |     | NULL    |                |
| answer   | varchar(20) | YES  |     | NULL    |                |
| name     | varchar(20) | YES  |     | NULL    |                |
| gender   | char(1)     | YES  |     | NULL    |                |
| birthday | date        | YES  |     | NULL    |                |
+----------+-------------+------+-----+---------+----------------+
```

实例 Bean

```
package com.profound.sbatis.entity;

import java.io.Serializable;
import java.sql.Date;

public class RegUser implements Serializable{
```

```
    private Integer userID;
    private String regName;
    private String password;
    private String email;
    private String question;
    private String answer;
    private String name;
    private String gender;
    private Date birthday;

// 省略 geters \seters 构造及 toString 方法。
}
```

5 创建 MyBatis Mapper

该接口为 MyBatis ORM 映射接口。在默认情况下 SpringBoot 会使用 SqlSessionTemplate 来实现这些映射接口。

在 mapper 包下创建接口 RegUserMapper。

```
package com.profound.sbatis.mapper;

import java.util.List;

import org.apache.ibatis.annotations.Mapper;
import org.apache.ibatis.annotations.Result;
import org.apache.ibatis.annotations.ResultMap;
```

```
import org.apache.ibatis.annotations.Results;
import org.apache.ibatis.annotations.Select;
import org.apache.ibatis.annotations.Update;

import com.profound.sbatis.entity.RegUser;

/************************************************************
@Mapper 与 @MapperScan 注解。
MyBatis 所有映射接口需添加 @Mapper 注解。
如不在接口上添加@Mapper 注解，则必需在启动器上添加注解
@MapperScan(value = {"com.profound.sbatis.mapper"})来指定 Mapper 所在的包。
************************************************************/
@Mapper
public interface RegUserMapper {

    // @Insert 注解定义添加操作的语句，绑定到接口方法上。
    @Insert("insert into reg_user(reg_name, reg_name, passwd, email, question, answer, name, gender, birthday) "
            + "values (#{regName}, #{password}, #{email}, #{question}, #{answer}, #{name}, #{gender}, #{birthday})")
    public int insert(RegUser u);

    // @Delete 注解定义删除操作，绑定到接口方法上。
    @Delete("delete from reg_user where user_id=#{id}")
    public int deleteByID(int id);

    // @Select 确定该接口方法实现的查询语句
    @Select("select * from reg_user")
```

```
    // @Results 定义一个结果映射，id 属性指定这个映射的名字，以供其它的查询引用些映射。与字段是否是主键不是一个意思。
    @Results(id = "RegUserMapper",
            value = {@Result(column = "user_id", property = "userID", id = true), //@Result 定义表中的字段，对应 JavaBean 中的属性，是否是主键。
                    @Result(column = "reg_name", property = "regName"),
                    @Result(column = "passwd", property = "password"),
                    @Result(column = "email", property = "email"),
                    @Result(column = "question", property = "question"),
                    @Result(column = "answer", property = "answer"),
                    @Result(column = "name", property = "name"),
                    @Result(column = "gender", property = "gender"),
                    @Result(column = "birthday", property = "birthday")
                    })
    public List<RegUser> listAll();

    @Select("select * from reg_user where user_id=#{id}")
    @ResultMap(value = "RegUserMapper") // 指定该查询所引用的映射。
    public RegUser findByID(int id);

    // @Update 注解，来要定义接口方法，所要实现的更新操作语句。
    @Update("update reg_user set reg_name=#{regName} where user_id=#{userID}")
    public int updateRegName(String regName, int userID);

    @Update("update reg_user set passwd=#{pwd} where user_id=#{userID}")
    public int updatePWD(String pwd, int userID);
}
```

5.1定义操作方法

本文使用注解来实现 MyBatis 的所有功能。在 Mapper 接口中，定义接口方法。在每个接口方法上使用注解，确定该方法所使用的 SQL 语句。

- 添加:
  @Insert 注解，用来确定 insert 语句。
- 删除:
  @Delete 注解，用来确定 delete 语句。
- 更改
  @Update 注解，用来确定 update 语句。
- 查询
  @Select 注解，用来确定 Select 语句。

5.2实体与表映射

从数据库表中查询出记录，并将记录映射到实体对象中，是 MyBatis 的主要功能。
使用@Results 注解，定义实体与表的映射关系。
@Results 注解属性如下:

属性	说明	示例
id	该@Results 所定义的映射的 ID。 可以通过该 ID 的值，引用该映射。	@Results(id = "RegUserMapper"，……)
value	value 的值是一个@Result 数组。 每一个@Result 中定义一个字段与实体属性的映射关系。	@Results(id = "RegUserMapper", value = {@Result(column = "user_id", property = "userID", id = true),

	@Result 注解，定义映射关系。id 属性为 true 表示该字段为主键；column 为字段名；property 为实体属性名；javaType 为属性对应的数据类型； jdbcType 为字段对应的数据类型。many 和 one 为关系映射。	@Result(column = "reg_name", property = "regName"), @Result(column = "passwd", property = "password"), @Result(column = "email", property = "email"), @Result(column = "question", property = "question"), @Result(column = "answer", property = "answer"), @Result(column = "name", property = "name"), @Result(column = "gender", property = "gender"), @Result(column = "birthday", property = "birthday") })

6 动态 SQL

使用注解来实现动态 SQL，与 XML 配置方式不同。MyBatis 提供了一个 SQL 类，通过该类的相应方法，返回动态生成的 SQL 语句。
使用动态 SQL 要从两个方面入手，1 是提供动态 SQL，2 是映射到 MyBatis 接口。
MyBatis 通用@SelectProvicer 、@UpdateProvicer、 @InsertProvider、 @DeleteProvider 提供动态 SQL 实现。
MyBatis 接口：

```
@Mapper
public interface BlogMapper {

    // 本方法不是动态查询
    @Select("select * from blog where user_id = #{userID}")
    @Results(id = "BlogMapper",
            value = {@Result(column = "blog_id", property = "blogID", id = true),
```

```
                @Result(column = "user_id", property = "userID"),
                @Result(column = "blog_name" , property = "blogName"),
                @Result(column = "nick_name", property = "nickName"),
                @Result(column = "photo", property = "photo"),
                @Result(column = "description" , property = "description"),
                @Result(column = "affiche", property = "affiche")})
    public Blog findByuserID(int userID);

    // 动态查询 and 条件
    @SelectProvider(type = BlogProvider.class, method = "findByExample")
    @ResultMap(value = "BlogMapper")
    public List<Blog> findByExample(Blog b);

    // 动态更新
    @UpdateProvider(type = BlogProvider.class, method = "updateByExample")
    public int updateByExample(Blog b);

    // 动态删除
    @DeleteProvider(type = BlogProvider.class, method = "delByExample")
    public int delByExample(Blog b);

    // 动态添加
    @InsertProvider(type = BlogProvider.class, method = "insertByExample")
    public int insertByExample(Blog b);

    // 动态查询 or 条件
```

```
    @ResultMap(value = "BlogMapper")
    @SelectProvider(type = BlogProvider.class, method = "listByNames")
    public List<Blog> listByNames(String[] names);
}
```

动态 SQL 提供类

```
package com.profound.sbatis.mapper;

import java.util.List;
import java.util.Map;

import org.apache.ibatis.jdbc.SQL;

import com.profound.sbatis.entity.Blog;

public class BlogProvider {

    public String findByExample(Blog b) {
        String s = null;
        SQL sql = new SQL();

        sql.SELECT("*").FROM("blog");
        if(b.getBlogID()!=null) {
            sql.WHERE("blog_id=#{blogID}");
        }else {
            if(b.getBlogName()!= null) {
                sql.WHERE("blog_name like #{blogName}");
```

```
            }

            else if(b.getNickName() != null) {
                sql.WHERE("nick_name like #{nickName}");
            }
        }

        s = sql.toString();
        return s;
    }

    public String updateByExample(Blog b) {
        SQL sql = new SQL();
        sql.UPDATE("blog");
        if(b.getBlogName()!=null) {
            sql.SET("blog_name = #{blogName}");
        }
        if(b.getNickName() != null) {
            sql.SET("nick_name= #{blogName}");
        }
        if(b.getBlogID() != null) {
            sql.WHERE("blog_id=#{blogID}");
        }

        String s = sql.toString();
        return s;
```

```
    }

    public String delByExample(Blog b) {
        SQL sql = new SQL();
        sql.DELETE_FROM("blog");
        if(b.getBlogID()!= null) {
            sql.WHERE("blog_id=#{blogID}");
        }
        else if(b.getBlogName()!= null) {
            sql.WHERE("blog_name=#{blogName}");
        }
        return sql.toString();
    }

    public String insertByExample(Blog b) {
        SQL sql = new SQL();

        sql.INSERT_INTO("blog");
        if(b.getBlogID() != null) {
            sql.VALUES("blog_id", "#{blogID}");
        }
        if(b.getBlogName()!= null) {
            sql.VALUES("blog_name", "#{blogName}");
        }
        if(b.getNickName()!=null) {
            sql.VALUES("nick_name", "#{nickName}");
```

```
        }
        return sql.toString();
    }

    /**动态 SQL 的入参，不是 JavaBean 就是 Map。我们传入的数组或 List 被自动的装入 Map 中。
     *      从 map 中取数组参数用"array"做 KEY,取 List 参数用"list"做 KEY。
     **/
    public String listByNames(Map<String, String[]> map) {
        SQL sql = new SQL();
        sql.SELECT("*").FROM("blog");

        String[] array = map.get("array");

        if(null != array) {
            for (int i=0;i<array.length; i++) {
                sql.WHERE("blog_name=#{array["+i+"]}");
                if(i<array.length-1) {
                    sql.OR();   // or 条件
                }
            }
        }
        return sql.toString();
    }
}
```

7 关系映射

数据库之间的一对多、一对一、多对一、多对多的关系。在 java 对象中表现为对象之间的组合关系。

7.1一对一

数据库表之间的一对一关系，在对象之间表现为一个对象中包含另一个对象。如一个用户拥有一个博客。
用户对象定义如下:

```
public class RegUser implements Serializable{
    private Integer userID;
    private String regName;
    private String password;
    private String email;
    private String question;
    private String answer;
    private String name;
    private String gender;
    private Date birthday;
    private Blog blog;    // 一对一关联对象
   /* 省略 geter seter  构造方法及 toStrig 方法。*/
}
```

在 Mapper 中映射一对一关系

```
package com.profound.sbatis.mapper;
```

```
import java.util.List;

import org.apache.ibatis.annotations.Delete;
import org.apache.ibatis.annotations.Insert;
import org.apache.ibatis.annotations.Mapper;
import org.apache.ibatis.annotations.One;
import org.apache.ibatis.annotations.Options;
import org.apache.ibatis.annotations.Result;
import org.apache.ibatis.annotations.ResultMap;
import org.apache.ibatis.annotations.Results;
import org.apache.ibatis.annotations.Select;
import org.apache.ibatis.annotations.Update;
import org.apache.ibatis.mapping.FetchType;
import org.mybatis.spring.annotation.MapperScan;

import com.profound.sbatis.entity.RegUser;

@Mapper
public interface RegUserMapper {

	@Select("select * from reg_user where reg_name=#{regName} and passwd=#{password}")
	@Results(id = "RegUserBlogMapper",
			value = {@Result(column = "user_id", property = "userID", id = true),
					@Result(column = "reg_name", property = "regName"),
					@Result(column = "passwd", property = "password"),
```

```
                @Result(column = "email", property = "email"),
                @Result(column = "question", property = "question"),
                @Result(column = "answer", property = "answer"),
                @Result(column = "name", property = "name"),
                @Result(column = "gender", property = "gender"),
                @Result(column = "birthday", property = "birthday"),
                @Result(column="user_id", property = "blog",
                        one = @One(fetchType = FetchType.EAGER,
                                select = "com.profound.sbatis.mapper.BlogMapper.findByuserID" // 关联查询映射
                        ))}
        )
    public RegUser findByRegNamePWD(String regName, String password);


}
```

7.2一对多

一个博客中可以有多个文章，表现为在 Blog 对象中有一个 List<Article>对象。

对象关联关系

```
public class Blog implements Serializable {
    private Integer blogID;
    private Integer userID;
```

```
    private String blogName;
    private String nickName;
    private String photo;
    private String description;
    private String affiche;
    private List<Article> articles;     //一对多关联对象。

    /*省略 geter\seter  构造方法*/
}
```

查询映射

```
@Mapper
public interface BlogMapper {

    @Select("select * from blog where blog_id=#{blogID}")
    @Results(id="BlogArticlesMapper",
            value= {@Result(column = "blog_id", property = "blogID", id = true),
                    @Result(column = "user_id", property = "userID"),
                    @Result(column = "blog_name" , property = "blogName"),
                    @Result(column = "nick_name", property = "nickName"),
                    @Result(column = "photo", property = "photo"),
                    @Result(column = "description" , property = "description"),
                    @Result(column = "affiche", property = "affiche"),
                    @Result(column = "blog_id", property = "articles",
                            many = @Many(select = "com.profound.sbatis.mapper.ArticleMapper.listAritcleByBlogID")
                            )
```

```
                    }
                )
    public Blog findWithArticles(int blogID);
}


@Mapper
public interface ArticleMapper {

    @Select("select * from article where blog_id=#{blogID}")
    @Results(id="ArticleMapper",
            value = {@Result(column = "article_id", property = "articleID", id = true),
                    @Result(column = "blog_id", property = "blog",
                            one = @One(select ="com.profound.sbatis.mapper.BlogMapper.findWithArticles")),
                    @Result(column = "title", property = "title"),
                    @Result(column = "content", property = "content"),
                    @Result(column = "keyss", property = "keyss"),
                    @Result(column = "access_count", property = "accessCount"),
                    @Result(column = "publish_date", property = "publishDate", jdbcType = JdbcType.DATE, javaType = Date.class)
            })
    public List<Article> listAritcleByBlogID(int blogID);
}
```

7.3多对一

文章与博客之间是多对一的关系。

Article 对象中包含一个 Blog 对象：

```
public class Article implements Serializable{

    private Integer articleID;
    private Blog blog;
    private String title;
    private String content;
    private String keyss;
    private int accessCount;
    private Date publishDate;

    /* 省略 geter\seter\构造方法*/
}
```

映射接口：

```
package com.profound.sbatis.mapper;

import java.sql.Date;
import java.util.List;

import org.apache.ibatis.annotations.Mapper;
import org.apache.ibatis.annotations.One;
```

```
import org.apache.ibatis.annotations.Result;
import org.apache.ibatis.annotations.Results;
import org.apache.ibatis.annotations.Select;
import org.apache.ibatis.type.JdbcType;
import com.profound.sbatis.entity.Article;

@Mapper
public interface ArticleMapper {

    @Select("select * from article where blog_id=#{blogID}")
    @Results(id="ArticleMapper",
            value = {@Result(column = "article_id", property = "articleID", id = true),
                    @Result(column = "blog_id", property = "blog",
                            one = @One(select ="com.profound.sbatis.mapper.BlogMapper.findWithArticles")),
                    @Result(column = "title", property = "title"),
                    @Result(column = "content", property = "content"),
                    @Result(column = "keyss", property = "keyss"),
                    @Result(column = "access_count", property = "accessCount"),
                    @Result(column = "publish_date", property = "publishDate", jdbcType = JdbcType.DATE, javaType = Date.class)
            })
    public List<Article> listAritcleByBlogID(int blogID);
}
```

8 添加事务

在完成以上步骤后，MyBatis 已经可以通过注入 Mapper 来实现 Mapper 的使用。但是每一个接口方法都有自己的事务。不能在多个接口方法间共享同一个事务，从而实现各方法间的事务关联。

要想实现各方法间相互配合完成一个共同的事务，需使用两个注解。

在启动器上添加@EnableTransactionManagement 注解。

在 Service Bean 上相应方法上添加 @Transactional 注解。

示例：本示例，在上面 Mapper 基础上实现，同时更改用户的“注册名”和“密码”。这两个更改在两个接口方法中。现在要在一个 Service 方法中调用这两个接口方法，实现同一事务，如果注册名重名（在数据库中注册名有唯一约束），之前修改的密码要回滚。全部成功才提交。

Service 类：

```
package com.profound.sbatis.service;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import com.profound.sbatis.mapper.RegUserMapper;

// 要使用@Service 注解，使 Spring 管理该对象
@Service
public class RegUserService {

    // 自动注入接口的实现
    @Autowired
    RegUserMapper regUserMapper;
```

```
    // 添加申明性事务
    @Transactional
    public void update(String regName, String pwd, int userID) {
        this.regUserMapper.updatePWD(pwd, userID);
        this.regUserMapper.updateRegName(regName, userID);

    }

}
```

在启动器中测试

```
package com.profound.sbatis;

import java.util.List;

import org.mybatis.spring.annotation.MapperScan;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.transaction.annotation.EnableTransactionManagement;

import com.profound.sbatis.entity.RegUser;
import com.profound.sbatis.mapper.RegUserMapper;
import com.profound.sbatis.service.RegUserService;
```

```
//@MapperScan(value = {"com.profound.sbatis.mapper"}) // 映射接口有@Mapper，就不需要用@MapperScan，两选一。
@EnableTransactionManagement
@SpringBootApplication
public class SbatisApplication implements CommandLineRunner{
    private RegUserMapper mapper;
    private RegUserService regUserService;

    public static void main(String[] args) {
        SpringApplication.run(SbatisApplication.class, args);
    }

    SbatisApplication(RegUserMapper m, RegUserService s){
        this.mapper = m;
        this.regUserService = s;
    }

    @Override
    public void run(String... args) throws Exception {
        this.regUserService.update("xie", "123", 4);
        System.out.println(this.mapper.listAll());
    }

}
```