

## 九、SpringBoot整合MyBatis

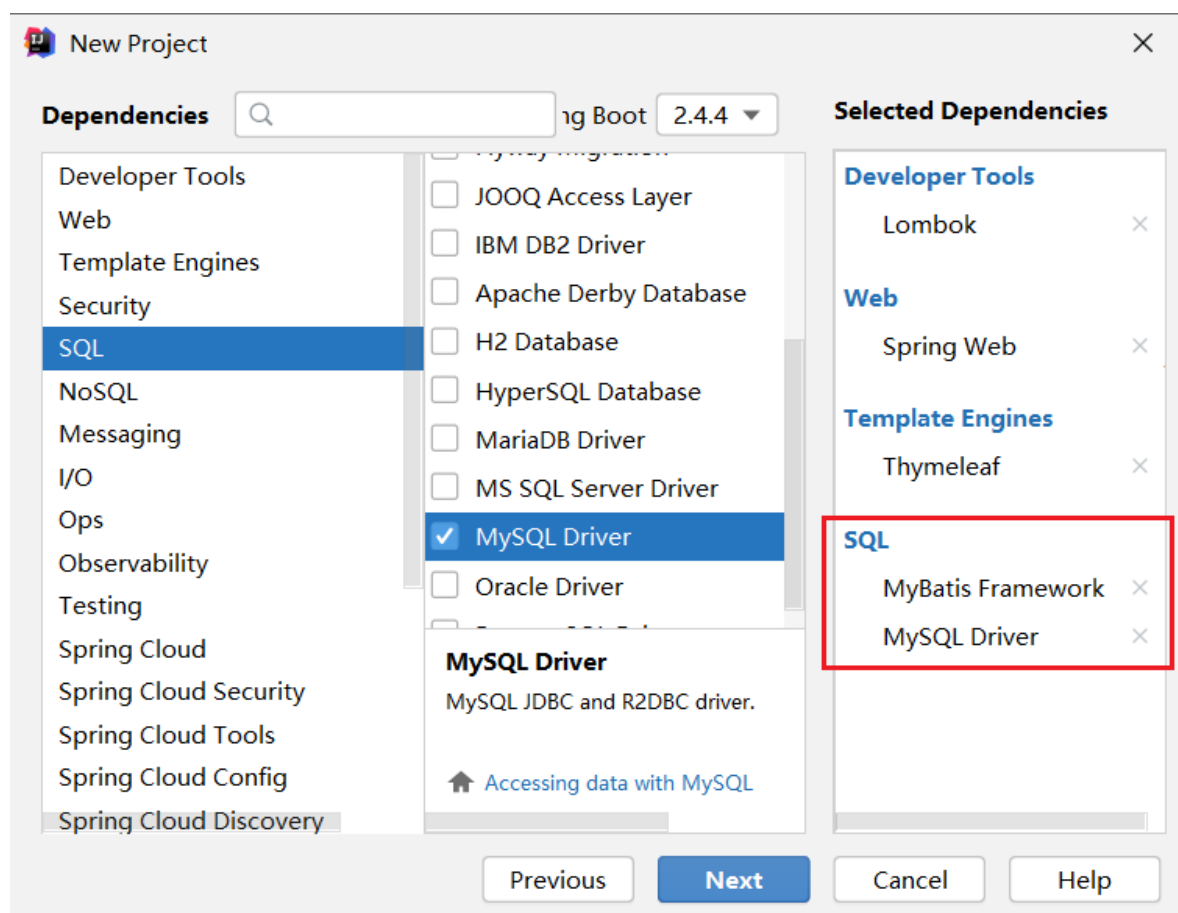
### 9.1 SpringBoot下MyBatis框架开发过程

#### 9.1.1 创建项目

通过一个用户注册和登录的案例来讲解SpringBoot中如何整合MyBatis以及事务管理，简单的功能需求描述就是可以进行用户的注册和登录，并在登录后跳转到首页显示出所有用户信息。

首先来创建项目，使用IDEA中的Spring Initializr创建项目：

要注意MyBatis的起步依赖中并没有包含对应的数据库驱动坐标，因为它并不知道我们当前使用的数据库是MySQL还是Oracle，所以需要自己来添加，在IDEA的初始器中勾选MyBatis框架和MySQL驱动：

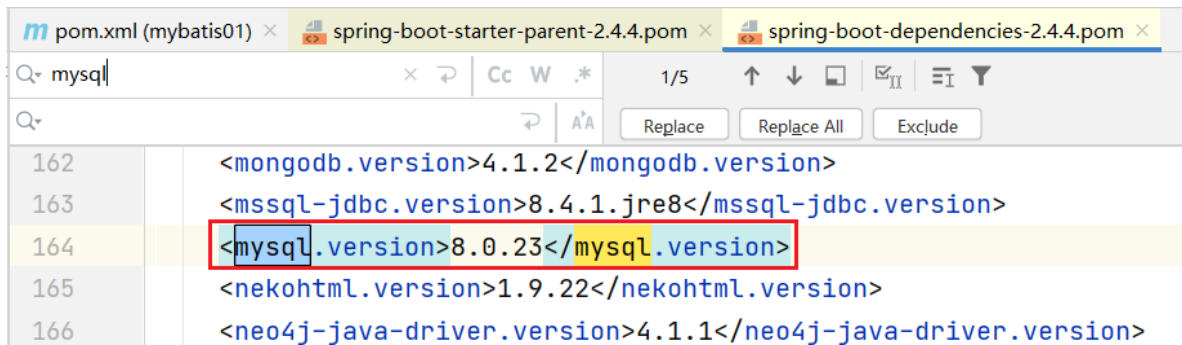


如果是常规Maven工程创建，则手动输入依赖部分，MySQL连接驱动的版本可以不指定（使用SpringBoot的默认指定版本）或者手动指定：

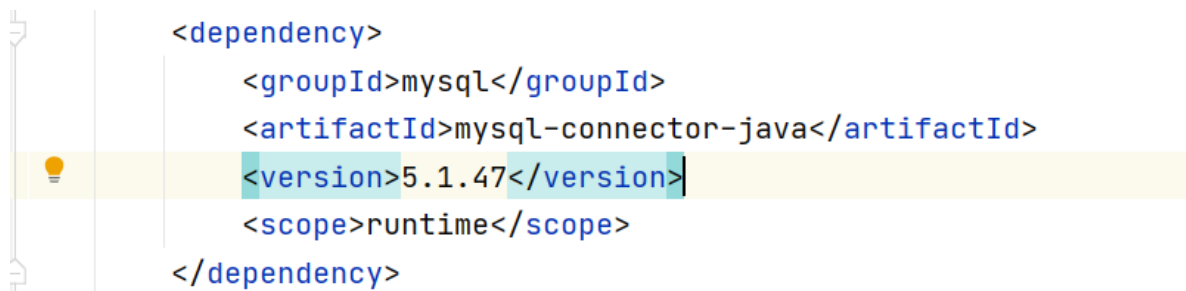
```

<!--MyBatis起步依赖-->
<dependency>
    <groupId>org.mybatis.spring.boot</groupId>
    <artifactId>mybatis-spring-boot-starter</artifactId> <!--mybatis开头，是MyBatis提供的-->
    <version>1.3.2</version>
</dependency>
<!--MySQL连接驱动-->
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <!--这里的版本在parent中已经有默认版本，可以使用默认指定的版本，也可以手动指定-->
    <version>5.1.47</version>
</dependency>

```

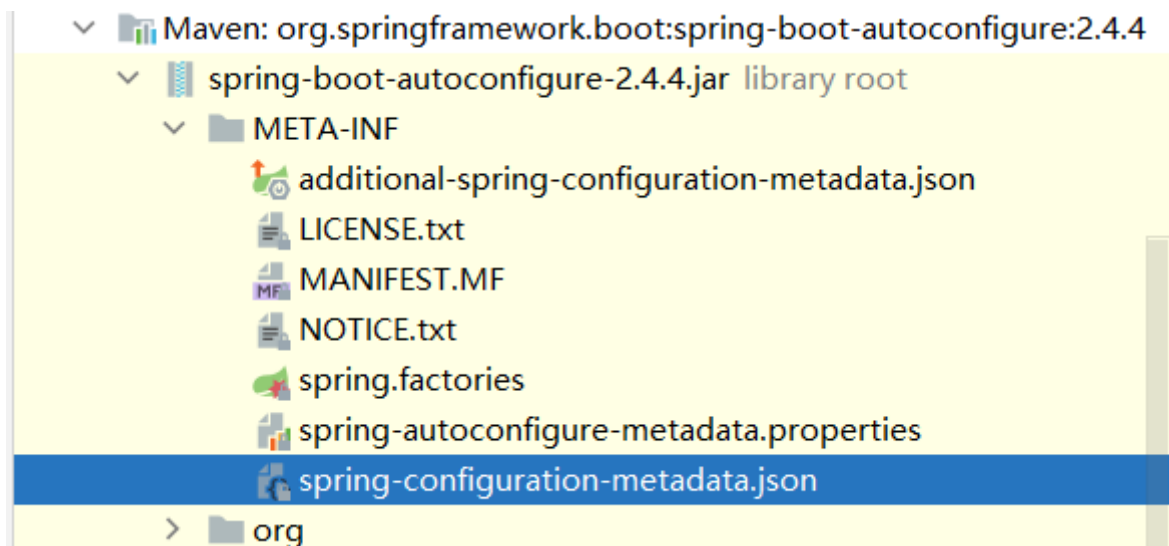


手动指定:



## 9.1.2 配置数据库连接

在springboot的自动配置包中，有一个默认配置的元数据文件，里面可以搜索到数据源配置相关的信息：



```

{
    "name": "spring.datasource.url",
    "type": "java.lang.String",
    "description": "JDBC URL of the database.",
    "sourceType": "org.springframework.boot.autoconfigure.jdbc.DataSourceProperties"
},
{
    "name": "spring.datasource.username",
    "type": "java.lang.String",
    "description": "Login username of the database.",
    "sourceType": "org.springframework.boot.autoconfigure.jdbc.DataSourceProperties"
},

```

配置的属性来源于自动配置包下的 `jdbc` 包中的 `DataSourceProperties` 类

按照以前的方式进行数据源配置（注意MySQL Connector 5.x和8.x版本中配置的区别）：

```

spring:
  datasource:
    driver-class-name: com.mysql.cj.jdbc.Driver
    url: jdbc:mysql://localhost:3306/18Java1班?useUnicode=true&characterEncoding=utf-8
    username: root
    password: root

```

# 8.x版本中需要修改驱动类名称，并补充连接url的参数配置信息：

```

spring:
  datasource:
    driver-class-name: com.mysql.cj.jdbc.Driver
    url: jdbc:mysql://localhost:3306/18Java1班?
useUnicode=true&characterEncoding=utf8&useSSL=false&serverTimezone=UTC&rewriteBatchedS
tatements=true
    username: root
    password: root

```

在 `DataSourceProperties` 类中还有一个 `type` 属性，用来指定使用的连接池实现类的全限定名称，默认情况下是根据类路径自动检测的。当前SpringBoot里默认使用的连接池是 `HikariCP`。

### 9.1.3 数据库、表和实体类创建

可以直接使用前面项目中创建过的用户表，或者自己再创建一个数据库和表也可以

列名	数据类型	长度	默认	主键?	非空?	Unsigned	自增?	Zerofill?	字符集	核对	更新	注释
id	int	11		<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>			<input type="checkbox"/>	
loginname	varchar	100		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	utf8	utf8 general ci	<input type="checkbox"/>	
password	varchar	100		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	utf8	utf8 general ci	<input type="checkbox"/>	
username	varchar	50		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	utf8	utf8 general ci	<input type="checkbox"/>	
status	int	11		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>			<input type="checkbox"/>	
salt	varchar	100		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	utf8	utf8 general ci	<input type="checkbox"/>	

```

@Data
public class User {
    private int id;
    private String loginname;
    private String password;
    private String username;
    private int status;
    private String salt;
}

```

### 9.1.4 控制器和页面

```
@Controller
public class LoginController {

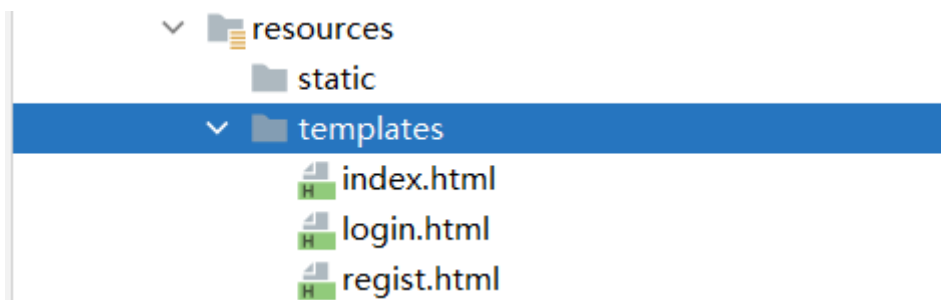
    @GetMapping("/login")
    public String getLogin(){
        return "login";
    }

    @PostMapping("/login")
    public ModelAndView doLogin(String loginname, String password){
        ModelAndView mav = new ModelAndView();
        return mav;
    }

    @GetMapping("/regist")
    public String getRegist(){
        return "regist";
    }

    @PostMapping("/regist")
    public ModelAndView doRegist(User user) {
        ModelAndView mav = new ModelAndView();
        return mav;
    }

    @GetMapping("/index")
    public String getIndex(){
        return "index";
    }
}
```



### 9.1.5 Service

IUserService.java:

```
public interface IUserService {
    // 添加用户操作
    int insert(User user);
    // 根据登录名和密码查询用户操作
    User query(String loginname, String password);
    // 获取所有用户信息操作
    List<User> queryAll();
}
```

UserServiceImpl.java:

```

@Service
public class UserServiceImpl implements IUserService {

    @Autowired
    private IUserDao userDao; // 注意，此处会报错，是因为在当前编码阶段找不到对应dao类型的Bean
    对象，是因为还没有使用MyBatis框架对Mapper生成相应的代理，可以使用@Resource注解替代，或者在DAO接口
    上加@Repository注解

    @Override
    public int insert(User user) {
        user.setSalt(user.getLoginname());
        user.setStatus(0);
        return userDao.insert(user);
    }

    @Override
    public User query(String loginname, String password) {
        return userDao.query(loginname, password);
    }

    @Override
    public List<User> queryAll() {
        return userDao.queryAll();
    }
}

```

## 9.1.6 Dao

编写dao接口

IUserDao.java:

```

public interface IUserDao {
    // 添加用户操作
    int insert(User user);
    // 根据登录名和密码查询用户操作
    User query(String loginname, String password);
    // 获取所有用户信息操作
    List<User> queryAll();
}

```

这个Dao接口的作用就是建立数据库中表的数据到项目中实体类之间的联系，因此需要实现这个接口进行数据库操作，这里是交给 **MyBatis** 框架来完成。

**MyBatis** 中对Dao接口的实现有两种方式：

1. 通过创建对应的 **mapper.xml** 文件进行映射
2. 通过在Dao接口的方法上加对应的注解实现

### 9.1.7.1 注解方式Dao开发

```

@Mapper
public interface IUserDao {
    // 添加用户操作
    @Insert("insert into user(loginname,password,username,status,salt) value(#{loginname},#{password},#{username},#{status},#{salt})")
    int insert(User user);
}

```

```

// 根据登录名和密码查询用户操作
@Select("select * from user where loginname=#{loginname} and password=#{password}")
User query(@Param("loginname") String loginname, @Param("password") String password);

// 获取所有用户信息操作
@Select("select * from user")
List<User> queryAll();
}

```

**@Mapper** 注解标记这个类是一个MyBatis的Mapper接口，可以被SpringBoot自动扫描到Spring上下文中。

如果不使用 **@Mapper** 注解而使用以前学习的 **@Repository** 注解，则需要在启动类上添加 **@MapperScan** 注解才能进行Mapper的扫描：

```

@Repository
public interface IUserDao {

@SpringBootApplication
@MapperScan(basePackages = "com.iflytek.mybatis01.dao")
public class Mybatis01Application {

```

### 9.1.7.2 XML配置方式Dao开发

在 **resources** 目录下创建一个 **mapper** 文件夹，专门用来存储各个mapper的xml文件：



**IUserDao.xml**：其中以查询所有用户方法为例：

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.iflytek.mybatis01.dao.IUserDao">
    <select id="queryAll" resultType="User">
        select * from user
    </select>
</mapper>

```

为了能够让MyBatis发现并自动绑定mapper的xml文件，以及在mapper文件中使用别名，需要进行以下配置：

```

mybatis:
# POJO别名包
type-aliases-package: com.iflytek.mybatis01.pojo
# Mapper映射文件加载路径
mapper-locations: classpath:mapper/*.xml

```

## 9.1.7 页面测试

### 9.1.7.1 登录测试

```

@PostMapping("/login")
public ModelAndView doLogin(String loginname, String password){
    ModelAndView mav = new ModelAndView();
    // 这里不考虑加密的问题，直接以登录名和密码进行SQL查询操作
    User user = userService.query(loginname, password);
    if (user != null) {
        mav.setViewName("redirect:/index");
    }else{
        mav.setViewName("/login");
        mav.addObject("msg", "用户名或密码错误! ");
    }
    return mav;
}

```

如果要观察MyBatis执行的中间过SQL语句过程，可以指定Dao层的日志输出级别为debug：

```

logging:
  level:
    com:
      iflytek:
        mybatis01:
          dao: debug

```

### 9.1.7.2 首页数据测试

```

@GetMapping("/index")
public ModelAndView getIndex(){
    ModelAndView mav = new ModelAndView("index");
    List<User> userList = userService.queryAll();
    mav.addObject("users", userList);
    return mav;
}

```

```

<html xmlns:th="http://www.thymeleaf.org">
. . .
<body>
<table border="1" cellpadding="0" cellspacing="0">
  <tr><th>序号</th><th>id</th><th>登录名</th><th>用户名</th><th>状态</th></tr>
  <tr th:each="user, stat : ${users}">
    <td>[[${stat.count}]]</td>
    <td>[[${user.id}]]</td>
    <td>[[${user.loginname}]]</td>
    <td>[[${user.username}]]</td>
    <td>[[${user.status}]]</td>
  </tr>
</table>
. . .

```

### 9.1.7.3 注册测试

```

@PostMapping("/regist")
public ModelAndView doRegist(User user) {
    ModelAndView mav = new ModelAndView("redirect:/login");
    int rlt = userService.insert(user);
    mav.addObject("rlt", rlt);
    return mav;
}

```

## 注册用户信息

登录名:

登录密码:

用户名:

[已有账号? 立刻登录](#)

完成注册后，使用新注册的账号进行登录测试。

## 9.2 事务管理

### 9.2.1 事务相关概念回顾

- 分类
  - 声明式事务管理：基于AOP，自动扫描包，指定范围开启事务
  - 编程式事务管理：手动开启事务：提交、回滚等
- 原理
  - 通过AOP技术，使用环绕通知进行拦截
- 注意
  - 不能使用 `try-catch` 代码块处理事务的异常
    - 因为要将异常抛出给上一层，外层对操作行为进行回滚操作。

### 9.3.2 SpringBoot中事务管理

SpringBoot中自动配置了 `DataSourceTransactionManager`，当在方法上（或者类上）添加 `@Transactional` 注解后，就自动纳入了Spring的事务管理中。

示例：

```
@Transactional
@Override
public int insert(User user) {
    user.setSalt(user.getLoginname());
    user.setStatus(0);
    int rlt = userDao.insert(user);
    LoggerFactory.getLogger(this.getClass()).info("数据库insert操作完成...");
    if (user.getLoginname().length() < 6 || user.getLoginname().length() > 10 ){
        // 模拟产生异常：假设用户名长度不能超出6-10的范围
        throw new RuntimeException("用户的登录名长度超出范围！");
    }
    LoggerFactory.getLogger(this.getClass()).info("即将返回操作结果");
    return rlt;
}
```

对比测试加和不加 `@Transactional` 注解时的表现。

## 十、MyBatis-Plus增强框架



## 10.1 MP增强框架入门

MyBatis已经足够我们使用了，但是我們希望能够更快、更方便，而MyBatis-Plus就是这样一个框架，它不能替代MyBatis，它的作用就是用来简化开发的。

之前我们学习MyBatis就是为了简化复杂的DAO层的JDBC操作，而MyBatis-Plus则是来更进一步简化MyBatis的。

### 10.1.1 是什么

官网地址：<https://mp.baomidou.com>

MyBatisPlus（简称MP）是国内人员开发的MyBatis增强工具,在MyBatis的基础上只做增强不做改变,为简化开发、提高效率而生。

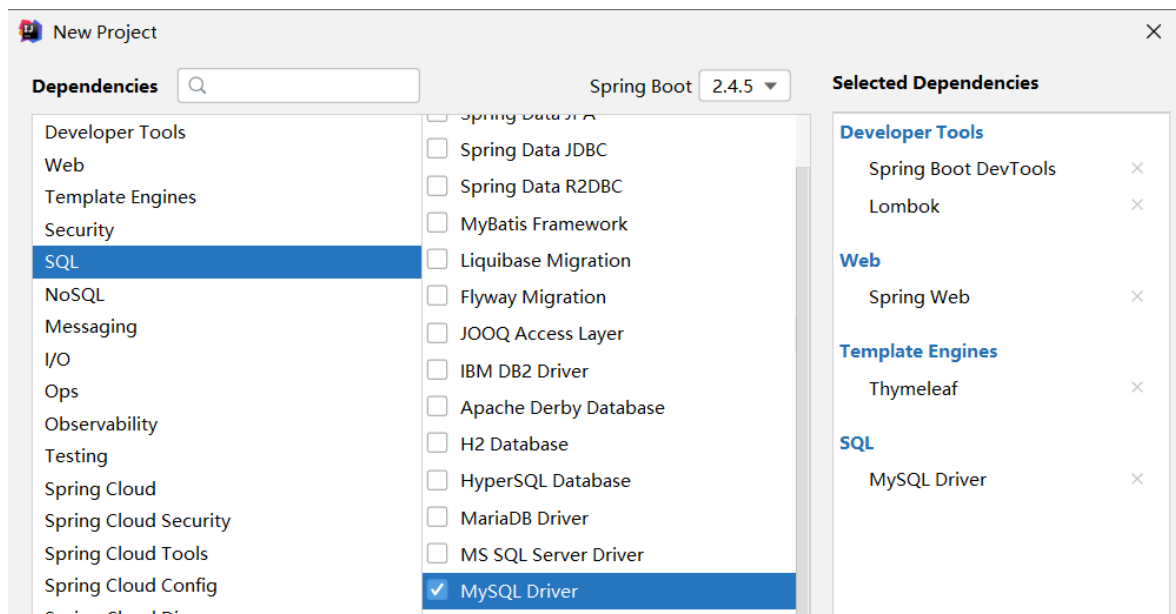
特性：<https://mp.baomidou.com/guide/#特性>

### 10.1.2 入门案例

#### 1) 创建项目

创建SpringBoot项目，使用IDEA的SpringBoot Initializr完成初始化。

注意，由于在 **MP** 中已经包含了MyBatis的依赖，所以创建项目时不需要勾选MyBatis依赖，避免版本差异导致问题。



#### 2) 导入依赖

官方手册中使用H2数据库作为默认数据库，注意区分依赖：

```
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <scope>runtime</scope>
  <version>5.1.47</version>
</dependency>
<dependency>
  <groupId>org.projectlombok</groupId>
  <artifactId>lombok</artifactId>
  <optional>true</optional>
```

```

</dependency>
<dependency>
    <groupId>com.baomidou</groupId>
    <artifactId>mybatis-plus-boot-starter</artifactId>
    <version>3.4.2</version>
</dependency>

```

### 3) 配置数据源

```

spring:
  datasource:
    driver-class-name: com.mysql.jdbc.Driver
    url: jdbc:mysql://localhost:3306/18java1班?useUnicode=true&characterEncoding=utf-8
    username: root
    password: root

```

### 4) 创建实体类

以图书表为例

```

@Data
public class Book {
    private int id;
    private String name;
    private String author;
    private float price;
    private String category;
    private String publisher;
    private Date publishdate;
    private int count;
    private String cover;
}

```

MP中提供了关于实体类及属性的几个注解：

- `@TableName`，修饰实体类，常用属性值：
  - `value`：实体类对应的数据库表名，相同时注解可以省略
- `@TableId`，修饰实体类中代表主键的属性，常用属性值：
  - `value`：对应数据库中的主键字段名称
  - `type`：主键生成策略
- `@TableField`，修饰实体类中非主键的其他属性，常用属性值：
  - `value`：对应数据库中的非主键字段名称
  - `exist`：默认为true，指定该属性在数据库中是否存在对应字段
  - `select`：默认为true，查询时是否查询该属性对应数据

### 5) 创建Service接口

```

public interface IBookService {
    Book queryById(int id);
}

```

```

@Service
public class BookServiceImpl implements IBookService {
    @Autowired
    private BookMapper bookMapper;
    @Override
    public Book queryById(int id) {
        return null;
    }
}

```

## 6) 创建Mapper接口

就是之前一直编写的Dao接口，不过在MP中的Dao接口需要继承 `BaseMapper<T>` 接口，从其作用上来说也是起到映射SQL的作用，因此命名习惯使用了 `mapper`。

```

@Repository
@Mapper
public interface BookMapper extends BaseMapper<Book> {
}

```




注意，此时只需要完成接口的继承即可。

当继承了指定的接口后，在Service方法中可以直接调用其中继承的方法了。

```

@Override
public Book queryById(int id) {
    return bookMapper.;
}

```

	<code>selectById(Serializable id)</code>	<code>Book</code>
	<code>selectOne(Wrapper&lt;Book&gt; queryWrapper)</code>	<code>Book</code>
	<code>selectMaps(Page&lt;Book&gt; page, Wrapper&lt;Book&gt; queryWrapper)</code>	<code>List&lt;Book&gt;</code>

```

@Override
public Book queryById(int id) {
    return bookMapper.selectById(id);
}

```

## 7) 启动测试

创建控制器类和访问方法，完成按照id查询图书信息的操作：

```

@Controller
@Slf4j
public class TestController {

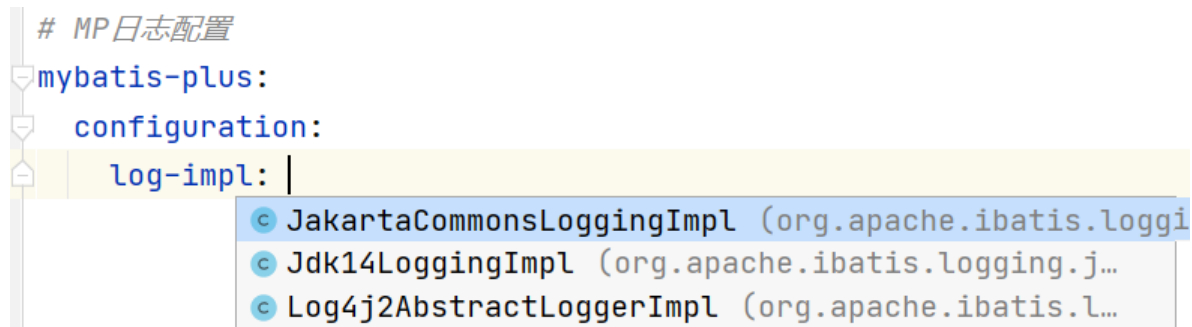
    @Autowired
    private IBookService bookService;

    @GetMapping("/book/{id}")
    @ResponseBody
    public Book findBook(@PathVariable("id") int id){
        log.info("do find book:" + id);
        return bookService.queryById(id);
    }
}

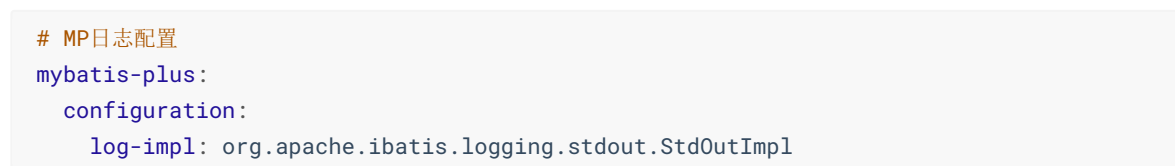
```

## 10.1.3 配置日志输出

这里我们会比较好奇，它到底生成了什么样的SQL语句呢？现在是看不到的，而且现有的信息里也没有看到相关的输出，所以如果想要看它具体怎么执行的，可以通过配置日志输出来看。



MP里的日志实现类有很多，其中部分使用时需要导入依赖，这里可以直接选择StdOutImpl，也就是直接输出在标准输出控制台中：

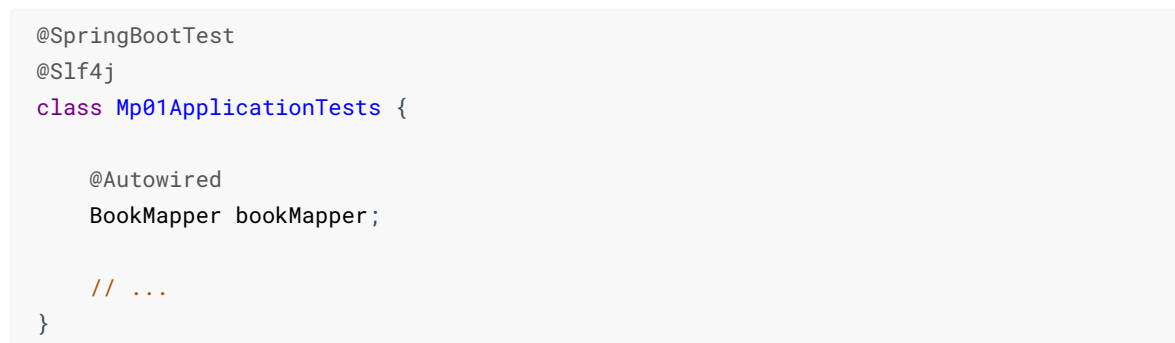
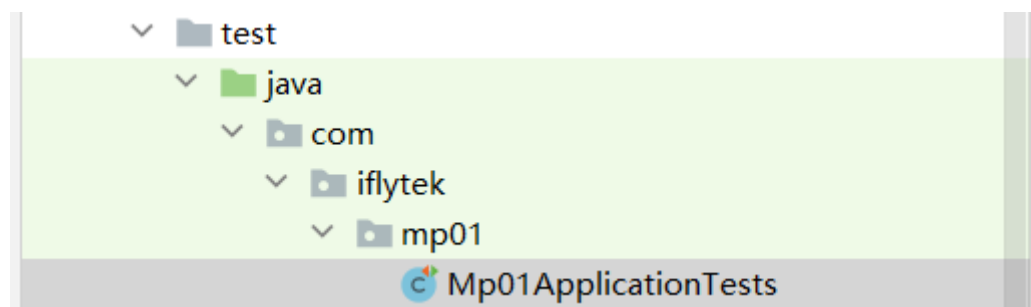


再次运行可以观察到日志输出：



## 10.2 MP增强框架进行CRUD操作

为了方便进行MP提供的方法测试，下面操作都在单元测试类中直接对Mapper进行调用，不通过控制器调用Service了：



单元测试里的日志输出默认黑白，要使用彩色输出，配置：

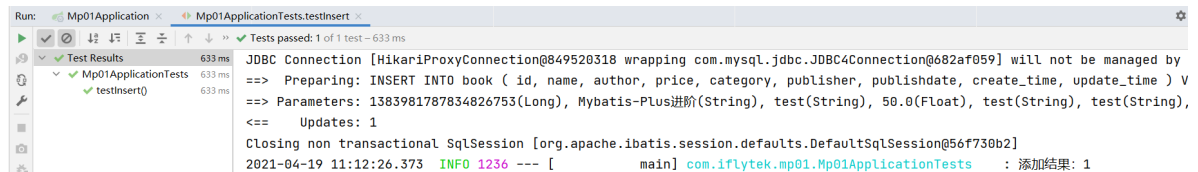
```
spring:
  output:
    ansi:
      enabled: always
```

## 10.2.1 添加操作

单元测试方法：

```
// 测试添加
@Test
void testInsert(){
    // 测试数据
    Book book = new Book();
    book.setName("Mybatis-Plus进阶");
    book.setAuthor("test");
    book.setCategory("test");
    book.setPrice(50.0f);
    book.setPublisher("test");
    book.setPublishdate(new Date());
    int rlt = bookMapper.insert(book);
    log.info("添加结果: " + rlt);
}
```

观察控制台可以看到输出结果以及执行的中间语句：



The screenshot shows the IDE's Run console. At the top, it says 'Tests passed: 1 of 1 test - 633 ms'. Below that, a 'Test Results' section shows 'testInsert()' passed in 633 ms. The main part of the console displays the SQL execution logs: 'JDBC Connection [HikariProxyConnection@849520318 wrapping com.mysql.jdbc.JDBC4Connection@682af059] will not be managed by ...', 'Preparing: INSERT INTO book ( id, name, author, price, category, publisher, publishdate, create\_time, update\_time ) V', 'Parameters: 1383981787834826753(Long), Mybatis-Plus进阶(String), test(String), 50.0(Float), test(String), test(String),', 'Updates: 1', and 'Closing non transactional SqlSession [org.apache.ibatis.session.defaults.DefaultSqlSession@56f730b2]'. At the bottom, a log line shows '2021-04-19 11:12:26.373 INFO 1236 --- [ main] com.iflytek.mp01.Mp01ApplicationTests : 添加结果: 1'.

## 10.2.2 主键生成策略

如果将实体类中的主键对应的id的类型从 `int` 改为对应的 `包装类型`（这也是比较推荐的，实体类中应当没有基本类型，基本类型使用其包装类型，修改价格`price`和库存`count`类型为对应的包装类型），将会出现这样的异常：

```
java.lang.IllegalArgumentException: Create breakpoint : argument type mismatch <4 internal calls>
```

具体的原因在页面异常信息中可以看到：

```
There was an unexpected error (type=Internal Server Error, status=500).
nested exception is org.apache.ibatis.reflection.ReflectionException: Could not set property 'id' of 'class com.iflytek.mp01.pojo.Book' with value '1383792215662305282' Cause:
java.lang.IllegalArgumentException: argument type mismatch
org.mybatis.spring.MyBatisSystemException: nested exception is org.apache.ibatis.reflection.ReflectionException: Could not set property 'id' of 'class com.iflytek.mp01.pojo.Book' with value
'1383792215662305282' Cause: java.lang.IllegalArgumentException: argument type mismatch
```

这里会发现，MP生成了一个很长的数字值 `138379xxxxxx`，然后想要赋值给Book类的id属性，然而id属性是一个Integer类型，数值范围超出了，因此无法完成赋值。

为什么会出现一个这么长的数字格式的值呢？但如果将id的类型设置为 `Long` 或者 `String`，并指定 `@TableId` 注解：

```
private @TableId Long id;
```

然后将数据库中的字段的类型设置为 `bigint` 或者 `varchar`，并指定 `足够的长度`，那么将会得到一个没有出错的结果：

```
==> Preparing: INSERT INTO book ( id, name, author, price, category, publisher, publishdate ) VALUES ( ?, ?, ?, ?, ?, ?, ? )
==> Parameters: 1383796975501070337(Long) Mybatis-Plus进阶(String), test(String), 50.0(Float), test(String), test(String), 2021-1
<== Updates: 1
Closing non transactional SqlSession [org.apache.ibatis.session.defaults.DefaultSqlSession@52aaf2c4]
添加结果: 1
```

这是由MP的主键生成策略来实现的。

MP的主键生成策略有哪些呢？可以从MP的一个注解入手：在主键字段上，可以添加一个注解 `@TableID` 来指定主键生成策略。这个注解的定义是：

```
public @interface TableId {
    String value() default "";

    IdType type() default IdType.NONE;
}
```

参数 `value` 表示当实体类中的主键属性名称和数据库字段名称不一致时指定对应的字段名称

另外一个参数 `type` 表示主键生成类型，取值为枚举类型 `IdType`，从这里的定义中可以看到其默认值为 `IdType.NONE`

具体取值可以参考官网说明(<https://mp.baomidou.com/guide/annotation.html#tableid>)

点击 `IdType` 链接，可以跳转到其源码（没有源码注释，可以点击右上角download resources），其中的注释说明了很多：

- AUTO
  - 数据库ID自增，使用这个值要求数据库中的主键也需要设置为自增长才行
- NONE
  - 该类型为未设置主键类型(注解里等于跟随全局,全局里约等于 INPUT)
- INPUT
  - 该类型要求insert前自行set设置值，可以通过自己注册自动填充插件进行填充，或者可以自己直接设定，例如通过user.setId设置指定值，但是如果没有指定呢？往下看，只有当插入对象ID为空时，就会用下面的策略自动填充
- ASSIGN\_ID
  - 分配ID (主键类型为Number (Long或者Integer) 或String) ,使用接口IdentifierGenerator的方法nextId(默认实现类为DefaultIdentifierGenerator雪花算法)
- ASSIGN\_UUID
  - 分配UUID (主键类型为 String)从3.3.0版本新增 使用接口IdentifierGenerator的方法nextUUID(默认default方法)

其他几种均已被弃用（后期版本种可能会直接删除）

### 10.2.3 更新操作

添加单元测试方法，设置要修改的测试数据：

```
// 测试更新
@Test
void testUpdate(){
    // 测试数据-只修改书名
    Book book = new Book();
    book.setId(1383796975501070337L);
    book.setName("Mybatis-Plus进阶操作");
    int rlt = bookMapper.updateById(book);
    log.info("修改结果: " + rlt);
}
```

运行测试:

```
Tests passed: 1 of 1 test - 851 ms
Test Results 851 ms
  ✓ Mp01ApplicationTests 851 ms
    ✓ testUpdate() 851 ms
2021-04-19 11:20:01.136 INFO 15000 --- [main] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Starti
2021-04-19 11:20:01.518 INFO 15000 --- [main] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Start
JDBC Connection [HikariProxyConnection@815590838 wrapping com.mysql.jdbc.JDBC4Connection@3b95a6db] will not be managed by
==> Preparing: UPDATE book SET name=? WHERE id=?
==> Parameters: Mybatis-Plus进阶操作(String), 1383796975501070337(Long)
<== Updates: 1
Closing non transactional SqlSession [org.apache.ibatis.session.defaults.DefaultSqlSession@7ce85af2]
2021-04-19 11:20:01.622 INFO 15000 --- [main] com.iflytek.mp01.Mp01ApplicationTests : 修改结果: 1
```

继续添加一个修改的参数:

```
// 测试更新
@Test
void testUpdate(){
    // 测试数据-只修改书名
    Book book = new Book();
    book.setId(1383796975501070337L);
    book.setName("Mybatis-Plus进阶操作");
    book.setPrice(40.0f); // 添加一个修改数据
    int rlt = bookMapper.updateById(book);
    log.info("修改结果: " + rlt);
}
```

运行测试:

```
Tests passed: 1 of 1 test - 632 ms
Test Results 632 ms
  ✓ Mp01ApplicationTests 632 ms
    ✓ testUpdate() 632 ms
2021-04-19 11:22:08.012 INFO 10660 --- [main] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Starti
2021-04-19 11:22:08.281 INFO 10660 --- [main] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Start
JDBC Connection [HikariProxyConnection@312629339 wrapping com.mysql.jdbc.JDBC4Connection@21b6c9c2] will not be managed by
==> Preparing: UPDATE book SET name=?, price=? WHERE id=?
==> Parameters: Mybatis-Plus进阶操作(String), 40.0(Float), 1383796975501070337(Long)
<== Updates: 1
Closing non transactional SqlSession [org.apache.ibatis.session.defaults.DefaultSqlSession@5807efad]
2021-04-19 11:22:08.375 INFO 10660 --- [main] com.iflytek.mp01.Mp01ApplicationTests : 修改结果: 1
```

一句话就是: MP可以自动完成动态SQL语句的配置

## 10.2.4 查询操作

开始时, 已经用过根据Id进行查询的操作了, MP还提供了一些其他的查询方法:

```
// 测试其他查询操作
@Test
void testQuery(){
    // 根据多个id批量查询
    List<Book> books = bookMapper.selectBatchIds(Arrays.asList(1,2,3));
    books.forEach(System.out::println); // Lambda表达式遍历List集合输出
    System.out.println("-----");

    // 条件查询--使用Map集合做参数 (简单条件使用Map, 更复杂的后面讲Wrapper)
    Map<String, Object> map = new HashMap<>();
    // 指定查询条件
```

```
map.put("author", "明日科技");
map.put("publisher", "清华大学出版社");
List<Book> books2 = bookMapper.selectByMap(map);
books2.forEach(System.out::println);
System.out.println("-----");
}
```

## 10.2.5 删除操作

`bookMapper.delete`

<code>delete(Wrapper&lt;Book&gt; queryWrapper)</code>	<code>int</code>
<code>deleteById(Serializable id)</code>	<code>int</code>
<code>deleteBatchIds(Collection&lt;? extends Serial...</code>	<code>int</code>
<code>deleteByMap(Map&lt;String, Object&gt; columnMap)</code>	<code>int</code>

Ctrl+向下箭头 and Ctrl+向上箭头 will move caret down and up in the editor [Next Tip](#)

删除方法可以有多个条件：

```
// 测试删除
@Test
void testDelete(){
    // 根据id删除
    //bookMapper.deleteById(1383796975501070337L);

    // 根据id批量删除
    //bookMapper.deleteBatchIds(Arrays.asList(1384033698094112770L,
    1384033740007763969L));

    // 通过map删除
    Map<String, Object> map = new HashMap<>();
    map.put("name", "MyBatis-Plus进阶");
    bookMapper.deleteByMap(map);
}
```

## 10.3 条件构造器

### 10.3.1 AbstractWrapper

MP中的多个查询方法里有一个 `Wrapper` 类型参数，是MyBatis-Plus中的一个抽象类，并在子类 `AbstractWrapper` 进行了部分实现，之后又分别派生了 `QueryWrapper` 和 `UpdateWrapper` 对应处理查询和增删改。

## AbstractWrapper

### 说明：

`QueryWrapper(LambdaQueryWrapper)` 和 `UpdateWrapper(LambdaUpdateWrapper)` 的父类用于生成 sql 的 where 条件, entity 属性也用于生成 sql 的 where 条件

注意: entity 生成的 where 条件与 使用各个 api 生成的 where 条件**没有任何关联行为**

要进行查询操作就需要实例化一个 `QueryWrapper` 对象即可。



### 10.3.2 查询一个：selectOne

查询数据库里书名叫三体的图书，并且只需要查询出一条记录出来：

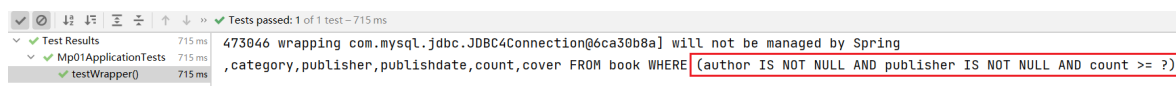
```
// 查询书名等于“三体”的图书
QueryWrapper<Book> wrapper1 = new QueryWrapper<>();
wrapper1.eq("name", "三体");
Book book1 = bookMapper.selectOne(wrapper1);
System.out.println(book1);
```

可以看到这个条件函数挺简单的，`eq` 就是表示这里哪个字段等于哪个值：equals

### 10.3.2 查询多个：selectList、selectMap

查询作者和出版社都不为空，且在库存量不少于50本的图书：

```
// 查询作者和出版社都不为空，且在库存量不少于50本的图书
QueryWrapper<Book> wrapper2 = new QueryWrapper<>();
wrapper2.isNull("author").isNull("publisher").ge("count", 50);
bookMapper.selectList(wrapper2).forEach(System.out::println);
```

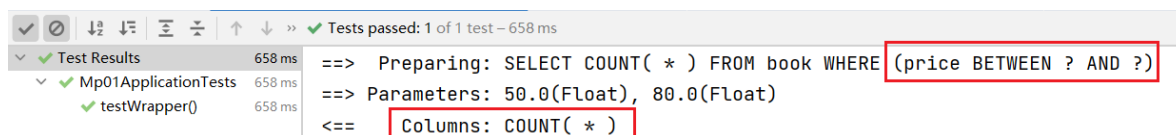


### 10.3.3 获取查询结果数量：selectCount

通常还有一些需求就是要获取指定条件的记录条数，例如查询开发部有多少员工等

现在查询价格在50到80之间的图书数据：

```
// 查询价格在50到80之间的图书数据
QueryWrapper<Book> wrapper3 = new QueryWrapper<>();
wrapper3.between("price", 50.0f, 80.0f);
Integer count = bookMapper.selectCount(wrapper3);
System.out.println(count);
```



### 10.3.4 模糊查询：like、notLike、likeLeft、likeRight

查询出版社中不包含“大学”，且书名是以“Java”开头的图书信息：

```
// 查询出版社中不包含“大学”，且书名是以“Java”开头的图书信息：
QueryWrapper<Book> wrapper4 = new QueryWrapper<>();
wrapper4.notLike("publisher", "大学")
    .likeRight("name", "Java");
bookMapper.selectList(wrapper4).forEach(System.out::println);
```

```
==> Preparing: SELECT id,name,author,price,category,publisher,publishdate,count,cover FROM book WHERE (publisher NOT LIKE ? AND name LIKE ?)
==> Parameters: %大学%(String), Java%(String)
```

除了 `likeRight` 还有 `likeLeft`，这里的右和左对应就是 `%` 号的位置是右还是左边。

### 10.3.5 其他

- 子查询：in、notIn、inSql、notInSql

- 排序：orderByAsc、orderByDesc、OrderBy
- 条件：and、or
- ...

自行了解学习。

# 十一、SpringBoot整合Spring Data JPA

## 1. JPA介绍

---

JPA是通过注解或者XML描述【对象-关系表】之间的映射关系，并将实体对象持久化到数据库中。

1) ORM映射元数据：JPA支持XML和注解两种元数据的形式，元数据描述对象和表之间的映射关系，框架据此将实体对象持久化到数据库表中；

如：@Entity、@Table、@Column、@Transient等注解。

2) JPA 的API：用来操作实体对象，执行CRUD操作，框架在后台替我们完成所有的事情，开发者从繁琐的JDBC和SQL代码中解脱出来。

如：entityManager.merge(T t);

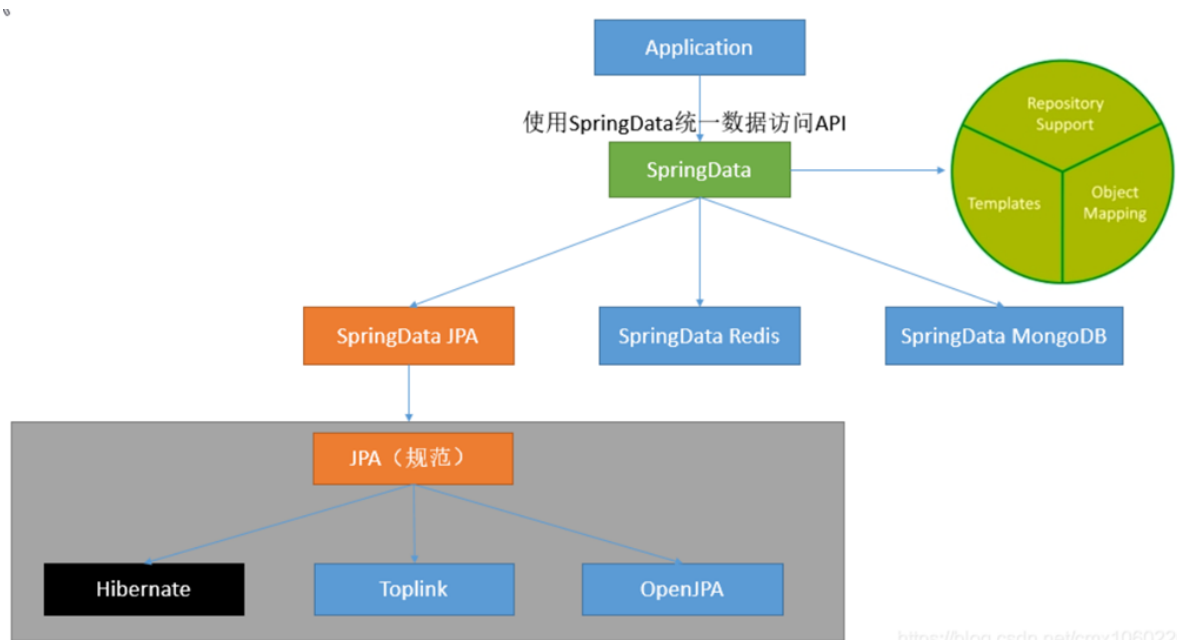
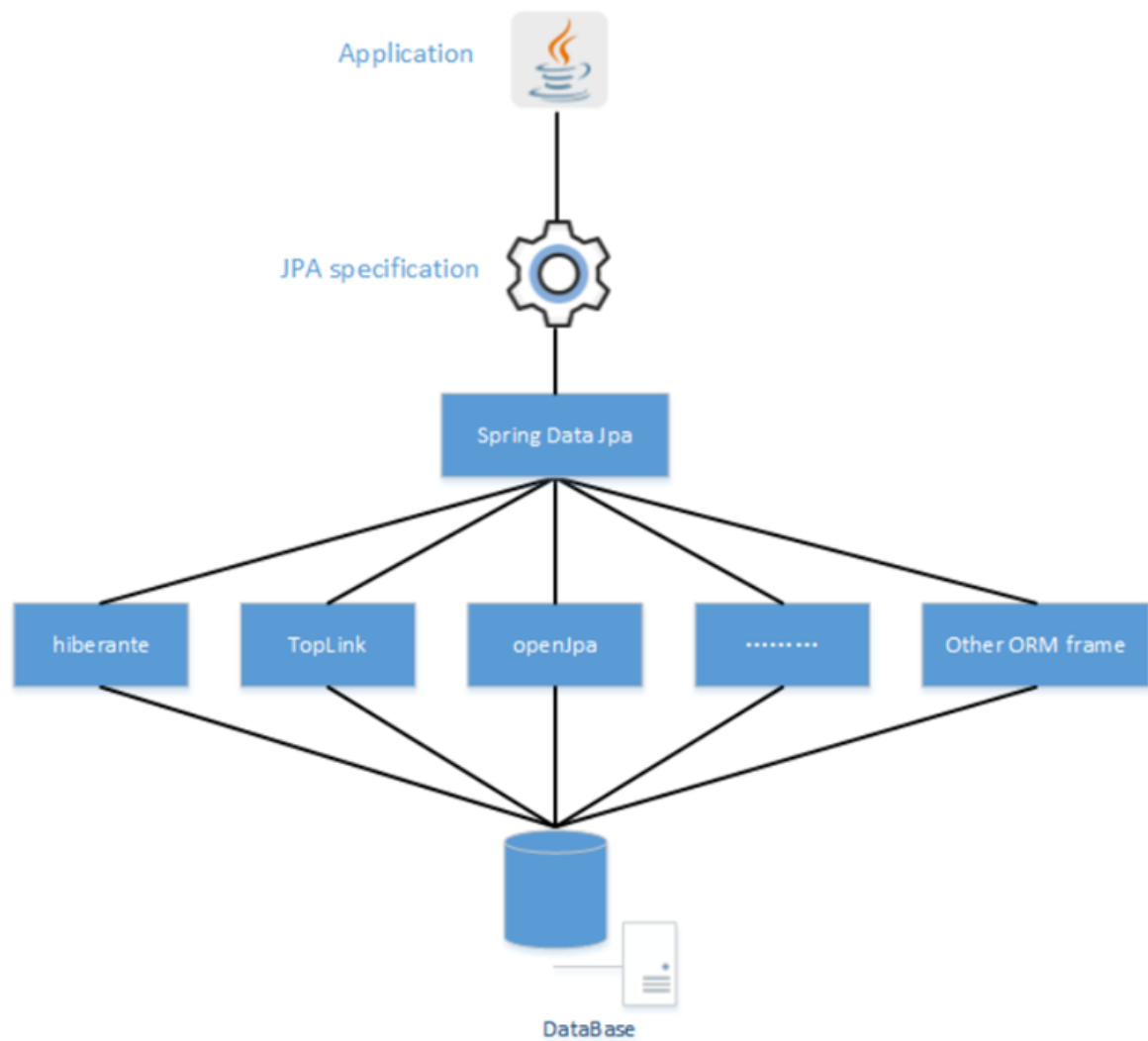
3) JPQL查询语言：通过面向对象而非面向数据库的查询语言查询数据，避免程序的SQL语句紧密耦合。

如：from Student s where s.name = ?

JPA仅仅是一种规范，也就是说JPA仅仅定义了一些接口，而接口是需要实现才能工作的。所以底层需要某种实现，而Hibernate就是实现了JPA接口的ORM框架。

spring data jpa是spring提供的一套简化JPA开发的框架，按照约定好的【方法命名规则】写dao层接口，就可以在不写接口实现的情况下，实现对数据库的访问和操作。同时提供了很多除了CRUD之外的功能，如分页、排序、复杂查询等等。

Spring Data JPA 可以理解为 JPA 规范的再次封装抽象，底层还是使用了 Hibernate 的 JPA 技术实现。



<https://blog.csdn.net/cmx1060226>

## 2.入门案例

1) 、引入POM包;

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

2)、编写一个实体类 (bean) 和数据表进行映射, 并且配置好映射关系;

```
@Table(name="t_stus")
@Entity
public class Stus {
    // 自增长主键
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    @Column
    private String name;
    @Column
    private String account;
    @Column
    private String psw;
```

3)、编写一个Dao接口来操作实体类对应的数据表 (Repository), 并指定泛型, 默认支持简单的CRUD操作。

```
public interface StusDao extends JpaRepository<Stus,Integer>
{
}
```

4)、基本的配置JpaProperties

```
#配置数据库连接
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/jpa01
spring.datasource.username=root
spring.datasource.password=123456

spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
```

- create: 每次加载Hibernate时都会删除上一次生成的表 (包括数据), 然后重新生成新表, 即使两次没有任何修改也会这样执行。适用于每次执行单测前清空数据库的场景。

- create-drop: 每次加载Hibernate时都会生成表, 但当SessionFactory关闭时, 所生成的表将自动删除。

- update: 最常用的属性值, 第一次加载Hibernate时创建数据表 (前提是需要先有数据库), 以后加载Hibernate时不会删除上一次生成的表, 会根据实体更新, 只新增字段, 不会删除字段 (即使实体中已经删除)。

- validate: 每次加载Hibernate时都会验证数据表结构, 只会和已经存在的数据表进行比较, 根据model修改表结构, 但不会创建新表。

## 5)、测试

```
@SpringBootTest
class Jpa01ApplicationTests {
    @Autowired
    private StusDao sd;
    @Test
    void contextLoads() {
        Stus s = new Stus();
        s.setName("张三");
        s.setAccount("test001");
        s.setPsw("001");
        s.setBrith(new Date());
        sd.save(s);
    }
}
```

## 3. 分页查询

```
// 分页查询并排序
@Test
public void selectUsers(){
    int page = 1;
    int size = 2;
    // 分页查询
    Pageable p = PageRequest.of( page: page-1, size);
    Page<Users> page1 = ud.findAll(p);
    Iterator<Users> i = page1.iterator();
    while(i.hasNext()){
        System.out.println(i.next());
    }
    // 排序查询
    Sort s = Sort.by(Sort.Direction.ASC, ...properties: "brith");
    List<Users> lst = ud.findAll(s);
    System.out.println(lst);
}
```

## 4. 自定义简单查询

•自定义的简单查询就是根据方法名来自动生成SQL，主要的语法是  
findXXBy, readAXXBy, queryXXBy, countXXBy, getXXBy后面跟属性名称

```

public interface UsersDao extends JpaRepository<Users,Integer> {

    public Users findByAccountAndPsw(String account,String psw);

    public List<Users> findAllByNameLike(String name);

}

@Test
public void selectUsers01(){
    List<Users> lst = ud.findAllByNameLike("李%");
    System.out.println(lst);
    System.out.println(ud.findByAccountAndPsw( account: "test001", psw: "001"));
}

```

## 5.自定义SQL

```

@Transactional
@Query(value = "delete from t_users where id = ?1",nativeQuery = true)
@Modifying
public int deleteUsersByUserId(int id);

@Transactional
@Modifying
@Query(value = "select u.id as id,u.name as name,u.account as account," +
    "u.psw as psw,u.brith as brith from t_users u " +
    "where name like ?1",nativeQuery = true)
public List<UsersInterface> getAllUsersByName(String name);

```

注意：@Query 与 @ Modifying 这两个 annotation 一起声明，可定义个性化更新操作

@Transactional声明为事务

nativeQuery=true使用原生态SQL

查询出来的数据必须自定义一个接口接收

```

public interface UsersInterface {
    int getId();
    String getName();
    String getAccount();
    String getPsw();
    Date getBrith();
}

```

```

@Test
public void selectUsers02(){
    // ud.deleteUsersByUserId(4);
    // System.out.println(ud.findAllByNameLike("李%"));
    List<UsersInterface> lst = ud.getAllUsersByAccount("test%");
    for(int i=0;i<lst.size();i++){
        UsersInterface u = lst.get(i);
        System.out.println(u.getAccount());
    }
}

// 告诉JPA这是一个实体类
@Table(name="t_stus")
@Entity
public class Stus {
    // 自增长主键
    @Id
    private int id;
    @Column
    private String name;
    @Column
    private String account;
    @Column
    private String psw;
    @Column
    private Date brith;
    @ManyToOne(cascade = CascadeType.ALL)
    @JoinColumn(name = "mid")
    private Major m;
}

@Table(name="t_majors")
@Entity
public class Major {

    @GeneratedValue(strategy= GenerationType.IDENTITY)
    @Id
    private int id;
    @Column
    private String name;
}

```



## 十二、SpringBoot整合Shiro

### 12.1 回顾Shiro安全框架基本内容

#### 12.1.1 是什么

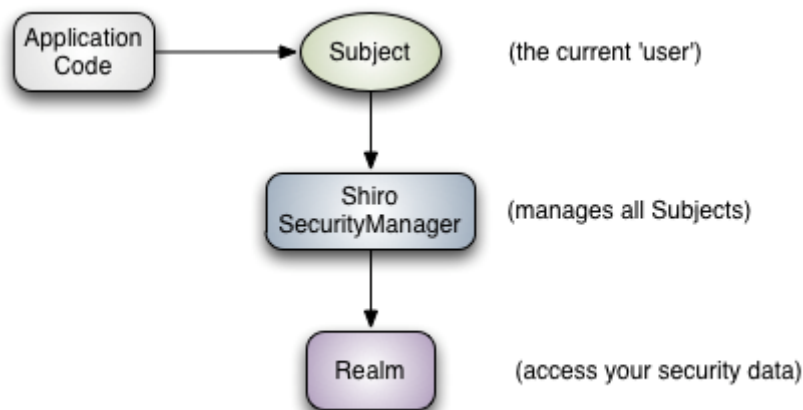
Apache Shiro是一个Java的安全（权限）框架，可以轻松地完成：身份认证、授权、加密、会话管理：

- 身份认证，说白了就是登录，涉及到身份权限识别、密码管理等
- 授权，根据身份找到对应的权限，权限不同，访问某些功能时就会不一样，可能继续访问，可能就提示权限不足
- 加密，密码是经过加密存储到数据库而不是明文存储的
- 会话管理，例如在web应用中我们登录后把登录用户存储在session中，做身份识别也就需要把身份信息、登录状态存储到会话中，所以做安全、访问控制的框架自然需要能够管理会话 基本不需要我们做什么控制，shiro框架在后台完成一些定制，开发者还是按session来使用

在SpringBoot中集成shiro，涉及到的主要业务逻辑是 身份认证 还有 权限校验，而这些shiro是不会自动维护的，因此要做的事情是自己来设计和提供这部分的功能代码，然后通过一些特定的接口注入到shiro框架里。

#### 12.1.2 核心组件

Shiro运行流程中有3个核心组件：Subject、SecurityManager、Realm



- **Subject**：主体，代表了当前“用户”，这个用户不一定是一个具体的人，与当前应用交互的任何东西都是Subject，如网络爬虫，机器人等；即一个抽象概念；所有 Subject 都绑定到 SecurityManager，与 Subject 的所有交互都会委托给 SecurityManager；可以把 Subject 认为是一个门面；SecurityManager 才是实际的执行者；
- **SecurityManager**：安全管理器；即所有与安全有关的操作都会与 SecurityManager 交互；且它管理着所有 Subject；可以看出它是 Shiro 的核心，它负责与后边介绍的其他组件进行交互，如果学习过 SpringMVC，可以把它看成 DispatcherServlet 前端控制器；
- **Realm**：域，Shiro 从 Realm 获取安全数据（如用户、角色、权限），就是说 SecurityManager 要验证用户身份，那么它需要从 Realm 获取相应的用户进行比较以确定用户身份是否合法；也需要从 Realm 得到用户相应的角色 / 权限进行验证用户是否能进行操作；可以把 Realm 看成 DataSource，即安全数据源。

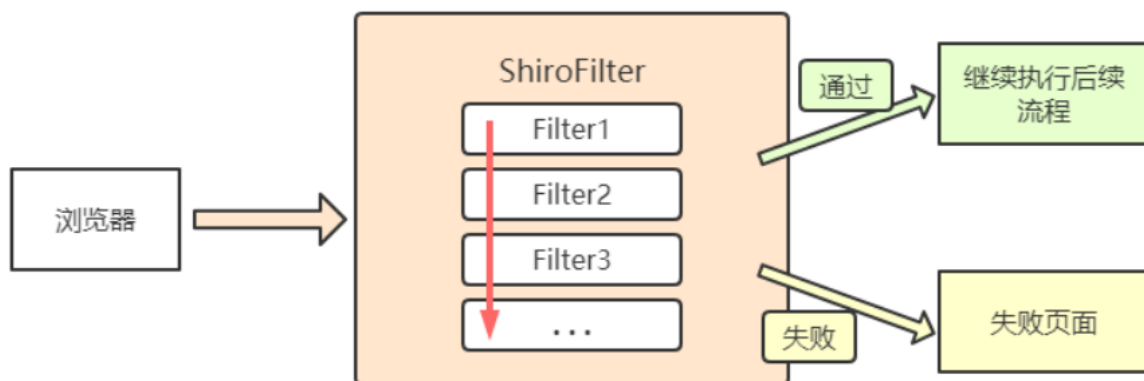
最简单的一个 Shiro 应用是这样的：



- 应用代码通过 Subject 来进行认证和授权，而 Subject 又委托给 SecurityManager；
- 需要给 Shiro 的 SecurityManager 注入 Realm，从而让 SecurityManager 能得到合法的用户及其权限进行判断。

从以上也可以看出，Shiro 不提供维护用户 / 权限，而是通过 Realm 让开发人员自己注入。

### 12.1.3 Web中的工作模式



Web项目中会有很多的Controller，然后在浏览器里访问一些路径，由Shiro对这些路径进行过滤，可以认为shiro是集成web项目后的一个入口，ShiroFilter拦截所有请求，对请求做访问控制。

例如：请求对应的功能是否需要认证的身份、是否需要某种角色、是否需要某种权限。如果没有做身份认证，则将请求强制跳转到登录页面；如果没有充分的角色或权限，则将请求跳转到权限不足的页面；如果校验成功，则执行请求的业务逻辑。

### 12.1.4 Shiro的开发基本流程

首先创建项目这些肯定是要做好的，导入依赖、建立需要的页面和Controller；

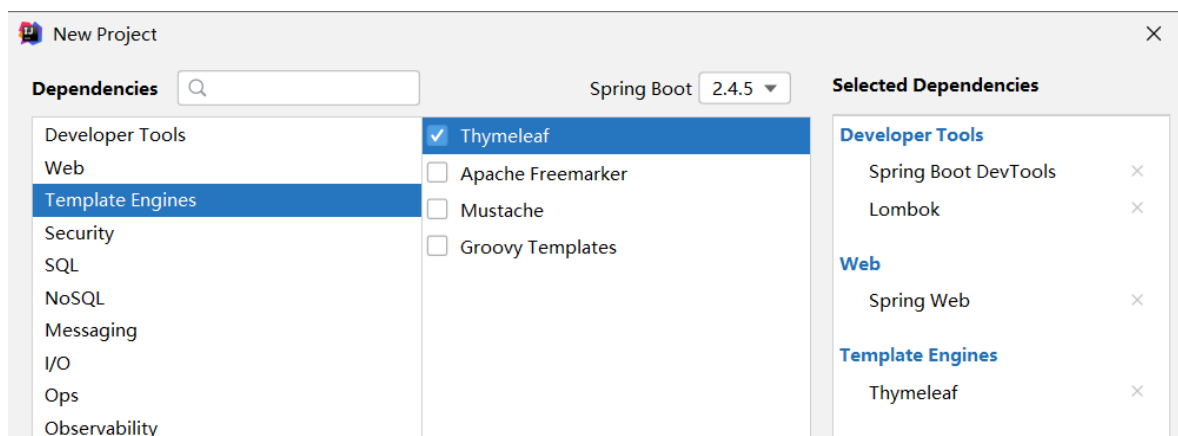
然后要写配置，包括用户、角色、权限这些信息，还要配置Shiro的过滤器，可以建立一个配置类来完成；

另外还需要配置Web中Shiro的入口，就是ShiroFilter，ShiroFilter里面配置一些过滤器，这个是Web中的核心组件；

## 12.2 SpringBoot中集成Shiro进行身份认证和权限校验

### 12.2.1 准备工作

创建项目，项目重点关注在Shiro框架上，所以此处的底层数据采用模拟数据，不使用数据库：



导入Shiro依赖：

```
<dependency>
  <groupId>org.apache.shiro</groupId>
  <artifactId>shiro-spring-boot-starter</artifactId>
  <version>1.7.1</version>
</dependency>
```

创建控制器和页面：

**UserController.java:**

```
@Controller
public class UserController {

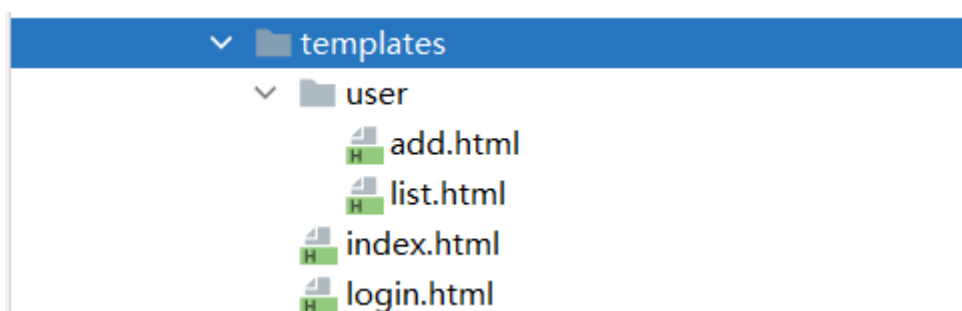
    @GetMapping("/login")
    public String getLogin(){
        return "login";
    }

    @GetMapping("/unauthor")
    @ResponseBody
    public String unauthor(){
        return "未授权！请联系管理员！"; // 简单返回，不设计页面了
    }

    @GetMapping("/index")
    public String index(){
        return "index";
    }

    @GetMapping("/user/add")
    public String add(){
        return "user/add";
    }

    @GetMapping("/user/list")
    public String list(){
        return "user/list";
    }
}
```



**index.html:**

```
<h1>首页</h1>
<a href="/logout">登出</a> <br>
<a th:href="@{/user/add}">添加用户</a> <br>
<a th:href="@{/user/list}">用户列表</a> <br>
</body>
```

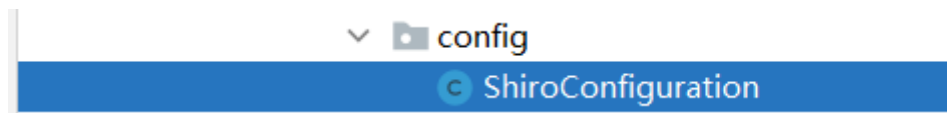
**login.html:**

```
<form action="/login" method="post">
    <input type="text" name="loginname" placeholder="用户名"> <br>
    <input type="password" name="password" placeholder="密码"> <br>
    <input type="submit" value="登录"> <br>
</form>
```

## 12.2.2 整合Shiro

### 12.2.2.1 配置类

创建Shiro的配置类 `ShiroConfiguration`：



```
@Configuration
public class ShiroConfiguration {
    // 过滤器: ShiroFilterFactoryBean

    // 安全管理器: DefaultWebSecurityManager

    // 数据域: Realm

}
```

### 12.2.2.2 自定义Realm

在config包下创建一个Java类，命名为 `MyRealm`，继承自 `AuthorizingRealm` 类，可以实现身份信息和权限信息获取：

```
public class MyRealm extends AuthorizingRealm {
    // 获取权限信息--授权，当访问的资源被ShiroFilter拦截时调用
    @Override
    protected AuthorizationInfo doGetAuthorizationInfo(PrincipalCollection
principalCollection) {
        return null;
    }

    // 获取身份信息--认证，执行subject.login时调用
    @Override
    protected AuthenticationInfo doGetAuthenticationInfo(AuthenticationToken
authenticationToken) throws AuthenticationException {
        return null;
    }
}
```

其中的功能暂时先放一下，在Shiro配置类中生成自定义Realm的对象并放到容器中：

```
// 数据域: Realm
@Bean(name = "myRealm") // Bean注解表示这个方法的返回值将创建一个Bean对象并交给Spring容器管理
                        // 默认这个Bean对象的名字为方法名，可以通过参数设置
public MyRealm getMyRealm(){
    return new MyRealm();
}
```

### 12.2.2.3 SecurityManager

在Shiro配置类中完成SecurityManager的构建，传入自定义的Realm

```
// 安全管理器: DefaultWebSecurityManager
// 因为SecurityManager需要一个Realm数据，而前面已经用Bean注解创建了Realm的对象
// 所以这里应该直接从Spring容器中去寻找这个Bean对象，因此用参数的形式来注入，并用注解获取它
// 这里使用Qualifier注解，作用：在按照类型注入的基础之上，再按照名称注入。
// 它在给类成员注入时不能单独使用，但是在给方法参数注入时可以单独使用。
// 最终这里的SecurityManager也要作为Bean对象进入Spring容器，让ShiroFilter去使用
@Bean("securityManager")
public DefaultWebSecurityManager defaultWebSecurityManager(@Qualifier("myRealm")
MyRealm myRealm) {
    DefaultWebSecurityManager securityManager = new DefaultWebSecurityManager();
    securityManager.setRealm(myRealm);
    return securityManager;
}
```

#### 12.2.2.4 ShiroFilter

```
// 过滤器: ShiroFilterFactoryBean
@Bean
public ShiroFilterFactoryBean shiroFilterFactoryBean(@Qualifier("securityManager")
DefaultWebSecurityManager securityManager){
    ShiroFilterFactoryBean bean = new ShiroFilterFactoryBean();
    // 设置安全管理器
    bean.setSecurityManager(securityManager);
    return bean;
}
```

到此核心组件基本配置完成，但是ShiroFilter的功能还不完善，例如哪些用户能访问哪些页面，哪些页面需要什么规则，这些都是要在ShiroFilter中来指定过滤器的规则。

回顾常用的内置过滤器：

- anon：匿名访问
- authc：需要认证，即登录
- roles、perms：需要指定的角色和权限

```
// 过滤器: ShiroFilterFactoryBean
@Bean
public ShiroFilterFactoryBean shiroFilterFactoryBean(@Qualifier("securityManager")
DefaultWebSecurityManager securityManager){
    ShiroFilterFactoryBean bean = new ShiroFilterFactoryBean();
    // 设置安全管理器
    bean.setSecurityManager(securityManager);

    // 获取shiro的内置过滤器链
    Map<String, String> filterChain = bean.getFilterChainDefinitionMap();
    // 添加指定规则
    // filterChain.put("/index", "anon"); // 不指定即为anon，index若允许匿名访问，可以不配置
    filterChain.put("/user/add", "authc,perms[user:add]"); // 需要登录，并具有user:add权限
    filterChain.put("/user/list", "authc,roles[admin]"); // 需要登录，并具有admin角色
    filterChain.put("/logout", "logout"); // 访问登出地址，不需要控制器映射，内置过滤器直接实现了登出

    // 设置登录规则（需要认证，但是未登录时将会跳转的页面）
    bean.setLoginUrl("/login");
}
```

```
// 设置授权规则（需要授权，但是权限不足时将会跳转的页面）
bean.setUnauthorizedUrl("/unauthor");

return bean;
}
```

## 12.2.3 数据及功能实现

### 12.2.3.1 用户、角色、权限类



User.java:

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class User {
    private Integer id;
    private String loginname;
    private String password;
    private String username;

    private Set<Role> roles; // 当前用户具备的角色
}
```

Role.java:

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class Role {
    private Integer id;
    private String rolename;

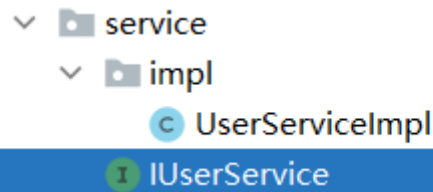
    private Set<Permission> perms; // 当前角色具备的权限
}
```

Permission.java:

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class Permission {
    private Integer id;
    private String permissionname;
}
```

### 12.2.3.2 数据

本案例中重点不在持久层，因此此处模拟实现，同学们自行扩展连接底层数据库查询：



IUserService.java:

```
public interface IUserService {
    // 根据登录名查询用户
    User getByLoginname(String loginname);
}
```

UserServiceImpl.java:

```
@Service
public class UserServiceImpl implements IUserService {
    // 模拟数据
    private static Map<String, User> map = new HashMap<>();
    static {
        // 模拟多组数据:
        // 权限数据
        Permission perm1 = new Permission(1, "user:add");
        Permission perm2 = new Permission(1, "user:query");

        // 角色及用户数据
        // zhangsan: admin: 具有所有权限
        Set<Permission> permSet1 = new HashSet<>();
        permSet1.add(perm1);
        permSet1.add(perm2);
        Role role1 = new Role(1, "admin", permSet1);
        Set<Role> roleSet1 = new HashSet<>();
        roleSet1.add(role1);
        User user1 = new User(1, "zhangsan", "12345", "张三", roleSet1);
        // lisi: employee: 可以查询
        Set<Permission> permSet2 = new HashSet<>();
        permSet2.add(perm2);
        Role role2 = new Role(2, "employee", permSet2);
        Set<Role> roleSet2 = new HashSet<>();
        roleSet2.add(role2);
        User user2 = new User(2, "lisi", "123", "李四", roleSet2);

        map.put(user1.getLoginname(), user1);
        map.put(user2.getLoginname(), user2);
    }

    @Override
    public User getByLoginname(String loginname) {
        return map.get(loginname); // 根据登录名查找对应用户
    }
}
```

### 12.2.3.3 自定义Realm实现

```
public class MyRealm extends AuthorizingRealm {

    @Autowired
    private IUserService userService;
```

```

// 获取权限信息--授权, 当访问的资源被ShiroFilter拦截时调用
@Override
protected AuthorizationInfo doGetAuthorizationInfo(PrincipalCollection
principalCollection) {
    String loginname = principalCollection.getPrimaryPrincipal().toString();
    User user = userService.getByLoginname(loginname);
    if (user == null){
        return null;
    }

    SimpleAuthorizationInfo info = new SimpleAuthorizationInfo();
    for (Role role : user.getRoles()){
        info.addRole(role.getRolename());
        for (Permission perm : role.getPerms()){
            info.addStringPermission(perm.getPermissionname());
        }
    }
    return info;
}

// 获取身份信息--认证, 执行subject.login时调用
@Override
protected AuthenticationInfo doGetAuthenticationInfo(AuthenticationToken
authenticationToken) throws AuthenticationException {
    String loginname = authenticationToken.getPrincipal().toString();
    System.out.println("loginname:" + loginname);

    User user = userService.getByLoginname(loginname);
    if (user != null){
        return new SimpleAuthenticationInfo(user.getLoginname(),
user.getPassword(), this.getName());
    }

    return null;
}
}

```

#### 12.2.3.4 控制器及页面实现

**UserController.java:**

```

@PostMapping("/login")
public String doLogin(User user){
    Subject subject = SecurityUtils.getSubject();
    AuthenticationToken token = new UsernamePasswordToken(user.getLoginname(),
user.getPassword());
    subject.login(token);
    return "redirect:index";
}

```

登录前、登录后访问测试。。。。

#### 12.2.3.5 角色过滤问题

如果想要以角色控制, 并实现多个角色都可以访问指定的地址时, 可能会配置成:

```
filterChain.put("/user/list", "authc,roles[admin,employee]"); // 需要登录，并具有admin或者employee角色
```

但是在Shiro中对于这种配置实际的逻辑是要求：必须具备admin和employee角色，是 **AND** 的关系。

需要重写角色过滤器。

创建类 `MyRolesAuthorizationFilter`，并重写检测方法：

```
public class MyRolesAuthorizationFilter extends RolesAuthorizationFilter {
    @Override
    public boolean isAccessAllowed(ServletRequest request, ServletResponse response,
    Object mappedValue) throws IOException {
        Subject subject = this.getSubject(request, response);
        String[] rolesArray = (String[]) mappedValue;
        if (rolesArray != null && rolesArray.length != 0) {
            Set<String> roles = CollectionUtils.asSet(rolesArray);
            //return subject.hasAllRoles(roles); // 默认方法是hasAllRoles，要求所有角色，
            重写此处
            for (String roleName : rolesArray) {
                if (subject.hasRole(roleName)) {
                    return true; // 只要具备角色之一就返回true
                }
            }
            return false; // 未在角色数组中检测到目标角色
        } else {
            return true;
        }
    }
}
```

之后将其配置到Shiro的安全管理器组件中：

```
// 配置自定义的角色过滤器
bean.getFilters().put("roles", new MyRolesAuthorizationFilter());
```

完成后再次测试。