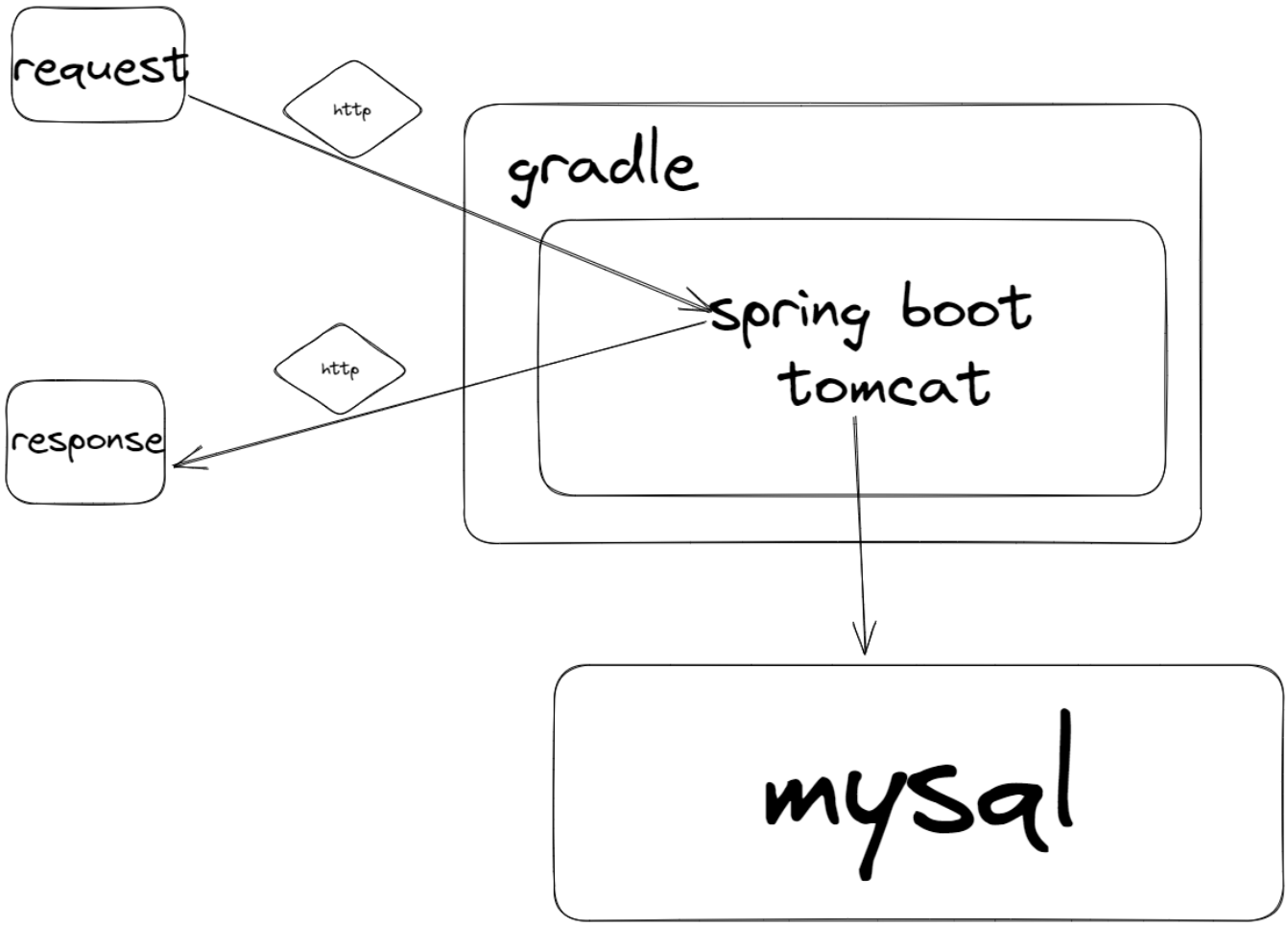
SPM开发日志

1. 架构

1.1 图书馆管理系统架构

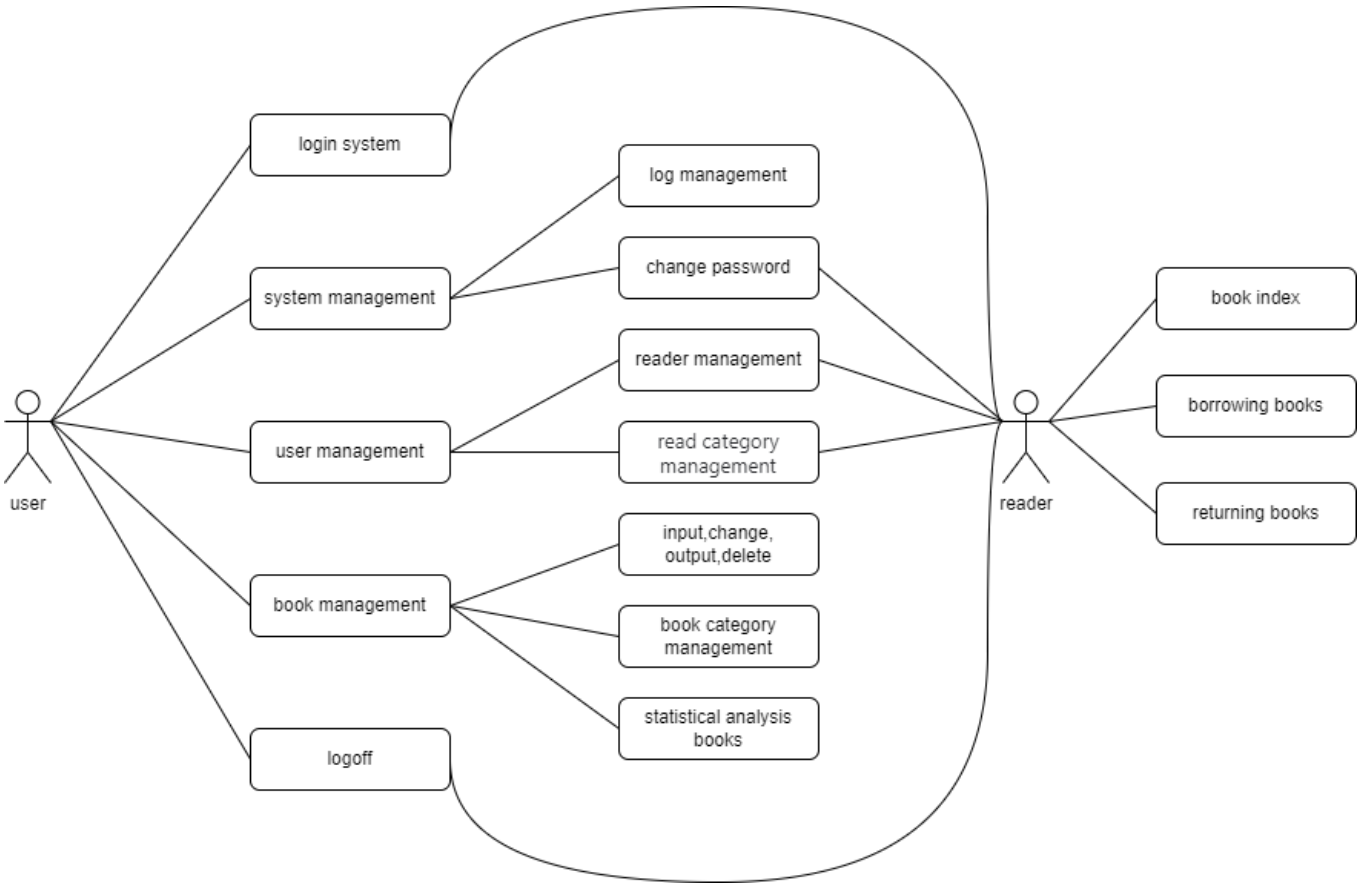


如图所示，我们通过使用gradle框架构建项目的架构。gradle中包含一个spring boot的项，spring boot内置了一个tomcat服务器。当外部发送请求时，其通过http协议传给spring boot，而spring boot将它的请求解析出来，经过spring boot和tomcat的共同处理，其最后再通过http协议将结果返回回去做出应答，同时将相应的数据存储到mysql中。

2.项目结构

2.1 项目结构导图

一个系统实现前我们应该将其的项目结构明晰，要做什么，要完成什么功能。那就应该有一个明确的思维导图。



2.2 项目结构阐述

对于一个图书馆管理系统来说，一切操作的基础都是用户，而对用户来说，其可以细分为管理图书馆的管理员和向图书馆寻求服务的客户。同时，客户和管理员的操作都是建立在图书的基础上的，所以我们还需要建立图书管理系统，用户实现操作需要登录，结束操作需要登出，因此这五个属于基本要求，之后的功能可以根据需求进行添加。

2.3 项目结构具体细分

2.3.1 系统层面

系统需要满足用户基本的登录登出，更改密码需求，同时对每个用户做的操作，例如客户借了书没，什么时候借的，还没还，什么时候还，要交多少押金，这都应该生成一条记录存入数据库中。

2.3.2 用户层面

用户层面细分管理员和客户。

- 管理员：对客户的数据进行管理，同时图书相关的所有操作应该由管理员来进行，例如书籍的增删改查等。

- 客户：查找书籍，借书，还书

2.3.3 图书层面

前面提到，管理员对书籍增删改查和客户对书籍进行查找，我们操作的是图书，图书作为被操作的对象，理应提供相应的方法。例如在现实生活中，每一本书的背后都有一个条形码标有数字，表示这本书独有的属性，那我们可以对书籍标记唯一的isbn，管理员就可以通过这个对书籍进行操作，客户也可以这样子查找，但是生活中客户一般都不知道书籍的isbn，所以我们还要提供通过书籍名查找的方法。除此之外，还需要对图书进行分类也是非常重要的，便于对图书进行管理。

3.技术点

3.1 系统层面

按照项目架构逐一实现：

- 设置了两个tomcat服务器，一个用于存放书的封皮，另一个用于存储文件。
- 使用了mybatis-puls(国人开发的，对mybatis进行了再一次封装)。
- 使用了spring-boot,配置为:
  - Project: Gradle-Groovy
  - Language: Java
  - Spring Boot: 3.0.1
  - JDK: 17
  - Package: Jar
  - DEVELOPER TOOLS: Spring Web, Spring Boot DevTools
- 加入了插件

```
plugins {
    id 'java'
    id 'org.springframework.boot' version '3.0.1'
    id 'io.spring.dependency-management' version '1.1.0'
}
```

- 加入了依赖

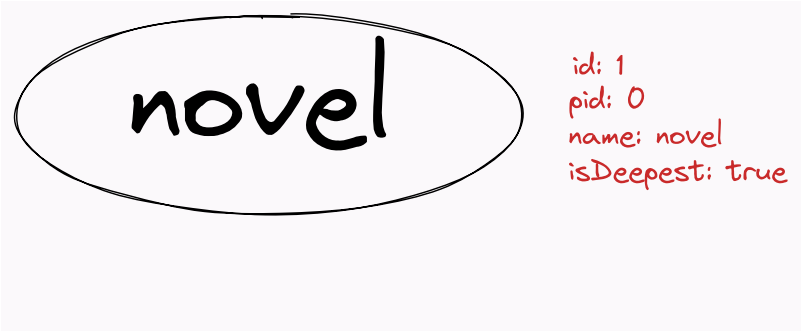
```
dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-web'
    implementation group: 'com.baomidou', name: 'mybatis-plus-boot-starter', version:
'3.5.3.1'
    implementation group: 'mysql', name: 'mysql-connector-java', version: '8.0.31'
    developmentOnly 'org.springframework.boot:spring-boot-devtools'
    testImplementation 'org.springframework.boot:spring-boot-starter-test'
}
```

- 配置mysql

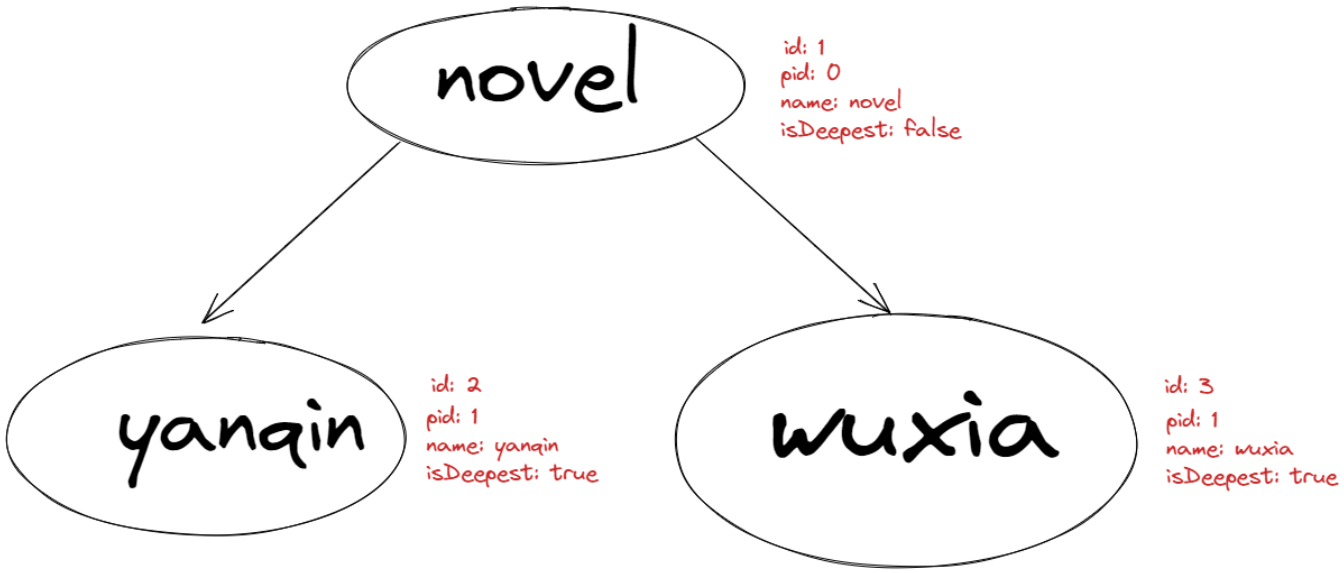
```yaml
spring:
  datasource:
    username: root
    password: 'spreadzhao'
    url: jdbc:mysql://localhost:3306/library?
serverTimezone=UTC&characterEncoding=utf8&useSSL=false
    driver-class-name: com.mysql.cj.jdbc.Driver
  main:
    banner-mode: off  # stop printing logo in terminal

mybatis-plus:
  global-config:
    banner: false  # stop printing logo in termin
```

3.2 图书层面

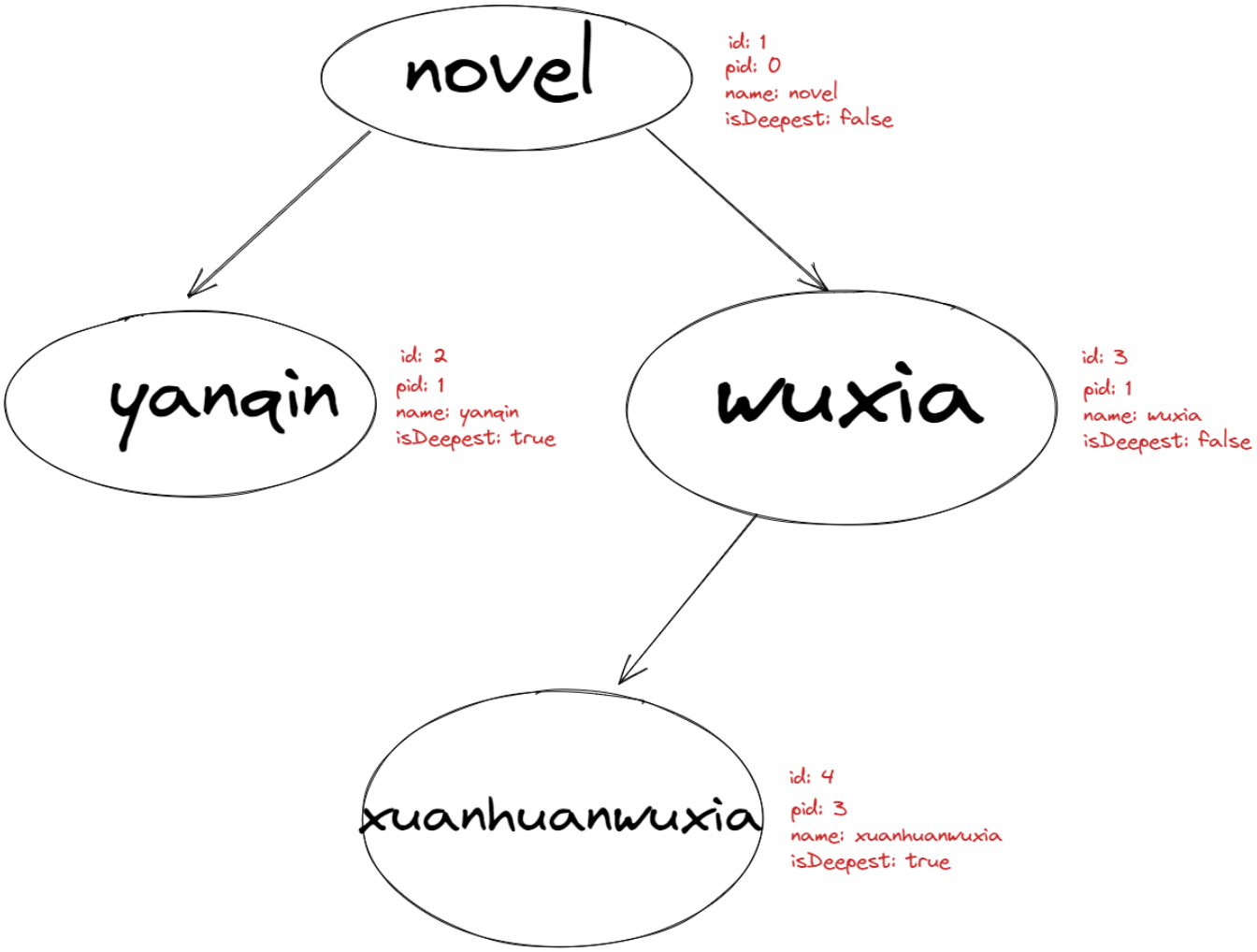
对图书的增删改查，这属于比较基本的的操作，而在图书层面的难点在于如何对图书进行分类，比如图书可以分为小说，小说又可以分为言情小说和武侠小说，武侠小说又可以细分为玄幻武侠小说等等。但是我们如何标记父类和子类，同样的，当一个父类的全部子类被删除时，那么它应该如何变化，这都是难点。



如图，我们首先创建一个小说类，归纳所有小说方面的图书。当我们创建是给它一个自身的标识，也就是id，同时因为它没有父类，所有将它的pid(parents id)置零。同时，我们不可能根据id去查找图书，所有我们还需要给它赋一个name，这里也就是novel。因为这时候只有一个类，所有它自身就是自己这个体系最深的节点，将isDeepest置为true。那它添加子类将如何变化呢？



可以看到，这里我们为小说类添加了言情，和武侠两个类。那么此时小说类拥有了子类，那么它不再是体系最深的节点，因而把isDeepest置为false，同时对新增的言情和武侠来说，它们的父类都是novel类，因此pid都置为1。由图显而易见，言情和武侠都是体系中最深的节点，故而将isDeepest置为true。



继续进行添加，如武侠生出了玄幻武侠这个类，那么武侠此时作为父类，它的isDeepest置为false。同时对玄幻武侠这个类它为体系中最深的节点之一，故而将isDeepest置为true，它的父类为武侠，因此pid置为3。

这就是图书添加以类划分的基本原理，我们可以看到。它的整个流程其实类似一颗树形，但和树状图还是存在一定的区别，例如novel类可以同时存在多个节点，武侠，言情，冒险等等。

4.具体实现

明确了项目的架构，结构，和所需要用到的技术点，我们就可以来正式开发这个项目了。

4.1 用户注册

一个系统最基本的元素就是用户嘛，我们所有的操作都是通过用户实现的，要想有用户，我们应该首先让用户注册，也就是先实现注册功能。同时用户分为客户和管理员，所以我们需要分别实现他们的功能(因为用户注册调用的都是同一个方法，因而这里只介绍管理员的注册流程）。

我们首先设计了一个管理员(客户也是一样的)，里面包括了所有管理员的操作方法，然后逐一实现。这里具体讲一下注册功能。

```
@GetMapping("managerop/register")
public Response registerManager(@RequestParam("account") String acc,
                                @RequestParam("password") String pwd){
    return Operation.register(acc, pwd, AccountType.MANAGER);
}
```

这就是一个注册方法了，@GetMapping("managerop/register") 是为了提示用户这是管理员的登录页面我们通过@RequestParam获取前端传入的accout即acc，password即pwd来将对应的值传入具体的Operation.register()方法中去，来实现管理员的注册功能。AccountType.MANAGER是为了告知这是个管理员的账户类型，因为管理员和客户用同一个方法注册账号，因而以此区分。

AccountTpye类的具体实现如下：

```
public class AccountType {
    public final static int MANAGER = 1;
    public final static int USER = 2;
}
```

我们将 MANAGER = 1， USER = 2，这样子当传入1时我们就知道是要注册管理员，传入2时是要注册客户。(PS：不用0与1是因为在java中，0通常表示FALSE，1表示TRUE，这里为了更加严谨。)

注意到这里的返回值类型是一个Response类，那么Response是用来做什么的呢？我们来看一下它的内容是什么。

```
private boolean status;
private String op;
private Message msg;
```

其他的就是一些基本的构造方法和get(，set()函数了。我们具体讲一下这三个变量是用来干嘛的。

首先是 status，我们注意到它是一个boolean类型的变量，所以它使可以作为标识符来用的，就比如我在开始操作时，我先将status置为FALSE，当操作成功时，我就把它改为TRUE，如果操作失败，那我就直接返回。这样就可以监控一个操作是否有效。

然后是op，顾名思义op也就是operation嘛，可以看到它是一个String类型的变量，它就是用来告诉我们在进行什么操作。其实这里还设计了一个OP类，这样子我们通过OP.xxxx的方式就可以调用OP类中的String类型的常量了。为什么不用OP op这样子捏，主要是因为把一个常量给弄成函数多少有点多余。这里给出OP类的部分常量。

```
public class Op {
    public final static String REGISTER_MANAGER = "Register manager";
    public final static String REGISTER_USER = "Register user";
    public final static String LOGIN = "Login";
    public final static String CHANGE_PSWD = "Change password";
    }
```

当我进行一个操作，比如登录，我就可以OP.LOGIN，它就会调用OP类中的LOGIN常量，输出"Login"了。

最后是msg了，也就是message嘛。在日常生活中，我们登录网页多少会遇到404这种报错，而msg也就是为了实现这个定义的，对每一种错误提示错误类型。注意到msg属于Message 类，那就让我们看看Message都有什么吧。

```
private String code;
private String content;
```

这是Message的两个属性，其它的就是它们的工作方法和get()函数了，为什么没有set()函数捏，因为出错类型，成功类型都已经被我们用另一个Msg类全部记录了，比如我在注册用户的时候出错，它应该是直接有对应的code和content，这些都是固定的，而Message类只是我得到出了什么错，或者没有错的途径。这里给出部分Msg类的方法，比较简单，就不细讲了。

```
private static String squareBracket(String str){
    return String.format("[%s]", str);
}

private static String bracket(String str){
    return String.format("(%s)", str);
}

public static Message Register(){
    return new Message("100", "Register successfully!");
}

public static Message Login(){
```

```
    return new Message("101", "Login successfully!");
}
public static Message ISE(){
    return new Message("000", "Internal-server error.");
}

public static Message AccountRegistered(String acc){
    return new Message("001", "Account name " + squareBracket(acc) + " has been
registered.");
}
```

可以看到code我们用了三位数字表示，第一位就是1和0嘛，标准的1为TRUE，0为FALSE，后两位就是编号了，后续如果错误类型多可以再添一位数。(PS:应该用不到那么多)

介绍完上述的前提，那么当acc, pwd, AccountType.MANAGER全部传入之后该如何操作呢?

我们知道，当要注册一个用户时，我们首先应该判断这个数据库中是否已经存在该用户，如果存在那我肯定要阻止你继续注册，要不然岂不是乱套了，不存在我才能让你注册下去。所以我们首先就应该对传入的账户与数据库中已经存在的账户进行比较。

```
Response res = new Response(false);
//就像之前所说，我们先将 res置为false，然后开始我们的操作。
LambdaQueryWrapper<Account> lqw = new LambdaQueryWrapper<>();
lqw.eq(Account::getName, acc);
```

这两句话就是mybatis-plus查询的条件，其性质就等价于

```
select xxx
from xxx
where accout=acc
```

对于传入的数据我们应该判断是管理员还是客户注册的。

```
if(type == AccountType.MANAGER) res.setOp(Op.REGISTER_MANAGER);
else if(type == AccountType.USER) res.setOp(Op.REGISTER_USER);
```

这里就用到了之前的AccountTpye类和op类，Response类，这两句话就是告诉你如果传入的是AccountType.MANAGER类型也就是1，那我就调用Response的set方法，将op的值置为Op.REGISTER_MANAGER。之前给出的OP类常量中可以看到op变成了Register manager。传入的是user也是一样的。

之后就是最核心的部分了，对传入的acc和pwd在数据库中继续查询，如果没查到，那我就应该把新的用户添加进去，然后把res的status置为TRUE，否则我就应该返回。但是这就结束了吗？如果在添加过程中出错该怎么办？因此，我们还应该判断添加是否成功。

```
if(null == accountMapper.selectOne(lqw)){
    accountMapper.insert(new Account(acc, pwd, type));
```

```
  if(null != accountMapper.selectOne(lqw)){
        res.setStatus(true);
        res.setMsg(Msg.Success.Register());
    }else{
        res.setMsg(Msg.Fail.ISE());
    }
}else{
    res.setMsg(Msg.Fail.AccountRegistered(acc));
}
return res;
```

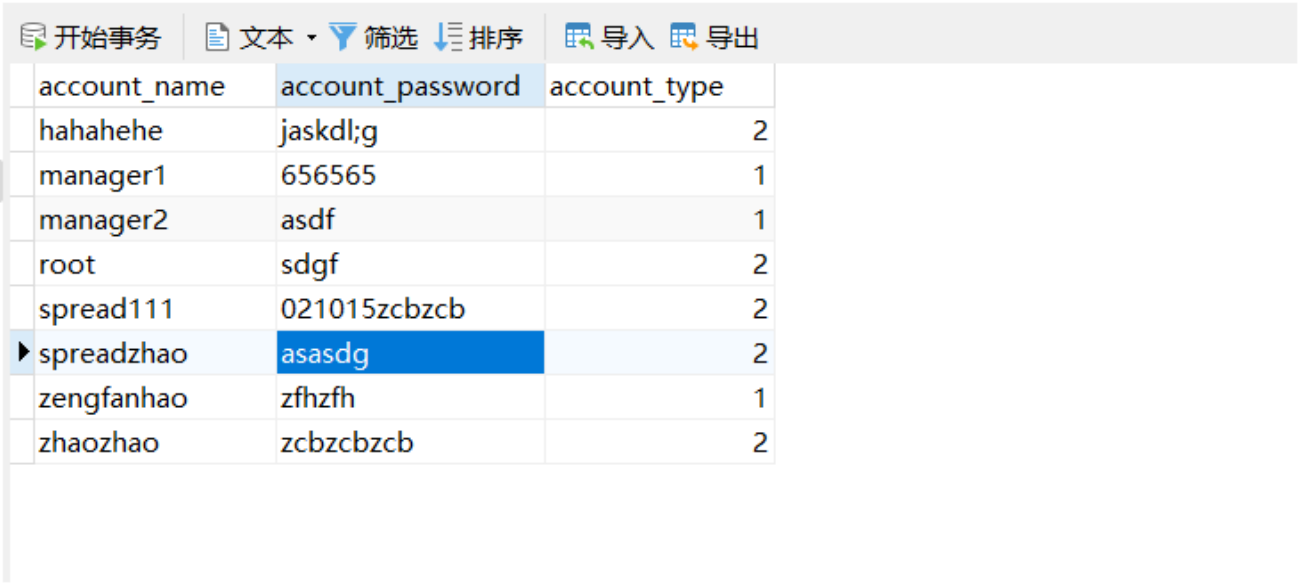
从代码可以看到，我们首先判断查询结果是否为空，为空就说明可以插入，那我就将值插入到数据库中。这里细讲一下这两句。

```
if(null == accountMapper.selectOne(lqw)){
    accountMapper.insert(new Account(acc, pwd, type));
```

注意到这里有个accountMapper，这是什么呢？其实不止有accountMapper，我们还有其他的Mapper,一个Mapper就对应一张表，我们对表的增删改查都是通过Mapper实现的。然后是Account，我们将acc, pwd, type全部插入即可。但是mysql怎么知道我要插入哪个表，哪个属性捏？让我们来看一下Accout类，这里给出了主要属性，其它都是基础的构造方法，get(),set()函数：

```
@TableId("account_name")
private String name;
@TableField("account_password")
private String password;

@TableField("account_type")
private int type;
```

可以看到我这里用了一个@TableId("account_name")，这就是告诉你数据库，我要的是这个表的这个属性。我们看一下数据库的表(PS：这里为了直观，使用了Navicat)



account_name	account_password	account_type
hahahehe	jaskdl;g	2
manager1	656565	1
manager2	asdf	1
root	sdgf	2
spread111	021015zcbzcb	2
spreadzhao	asasdg	2
zengfanhao	zfhzfh	1
zhaozhao	zcbzcbzcb	2

可以看到，我们通过@TableId("account_name")，将name与数据库的account_name进行了关联，其它类型也是一样的，例如Book之类的，后面就不再细讲了。

第一次查询成功之后，然后再进行第二次查询，看看我插入的值是否在数据库中。如果在，我就把res的Status改为true，然后把msg的值改为code："100", content："Register successfully!"。如果第二次查询结果为空，我就把msg的值改为code："000", content："Internal-server error!"，如果第一次就失败了，那我就把msg的值改为code："001", content："Account name " + squareBracket(acc) + " has been registered"，这里的squareBracket(acc)也就是账户名。然后将res的结果返回。也就是 **{xxx(操作op)+结果为true/false(status)+正确码/错误码(code)+正确内容/错误内容(content)}** 。这样子就完成了一次注册操作。

4.2 用户登录

有了用户之后，我们就应该实现用户的登录功能，毕竟只有登录了系统，你才能在系统中实现各类操作。登录功能管理员和客户调用一个方法，不同的是这次不用传入AccoutType，毕竟一旦注册，你的账户就唯一确定了。下面我们就来看看具体实现的过程。

我们这里还是看管理员的登录流程(客户是一样的)：

```
@GetMapping("managerop/login")
public Response loginManager(@RequestParam("account") String acc,
                             @RequestParam("password") String pwd){
    return Operation.login(acc, pwd);
}
```

如图所示，这里的@GetMapping("managerop/login") 和注册流程的大同小异，也就是告诉你这是管理员的登录页面，(如果是@GetMapping("userop/login")就是客服的登录页面) ，然后通过@RequestParam("account") String acc, @RequestParam("password") String pwd这两个语句获取你输入的账户和密码，通过Operation.login();方法，传入之后，在operation中判断你是否有这个账户，有的话，密码是否正确，登录是否成功之类的流程，最后返回Response类型的结果，之前已经详细写过，这里就不再赘述。

接下来我们来看一下Operation.login()方法具体是如何完成登录的。

```
Response res = new Response(false, Op.LOGIN);
```

首先，因为这个操作还没有成功，那么我就默认它失败，然后赋予op变量为Op.LOGIN，这个步骤其实就是告诉用户，这里要执行登录操作了。

```
LambdaQueryWrapper<Account> lqw = new LambdaQueryWrapper<>();
lqw.eq(Account::getName, acc);
Account a = accountMapper.selectOne(lqw);
```

和前面注册的原理是一样的，我们通过查询输入的账号，之后把查到的结果传给Accout类型的变量a，也是就账户名，密码，和账户类型嘛。（管理员AccoutType为1，用户为2）当然数据库中可能没有这个账户，这个时候我们肯定不能让用户登录，提示用户没有该用户；也有可能账户是有的，但是你输的密码错了，这个时候要告诉用户密码错了。

```
if(a != null){
    if(pwd.equals(a.getPassword())){
        res.setStatus(true);
        res.setMsg(Msg.Success.Login());
    }else{
        res.setMsg(Msg.Fail.WrongPassword(acc));
    }
}else{  // Account doesn't exist.
    res.setMsg(Msg.Fail.AccountNotExist(acc));
}
return res;
```

可以看到，我们先对输入的Accout类型a进行判断，因为如果数据库中不存在输入的acc，那么a的值一定是null，当它为null时，我们就应该将它的msg的content设置为acc账户不存在错误，并给出相应的错误码code，即 res.setMsg(Msg.Fail.AccountNotExist(acc))。如果存在我们就应该进行密码的判断，如果密码错误，那么我们要告诉用户，acc账户你输入的密码错了，也就是res.setMsg(Msg.Fail.WrongPassword(acc))。如果都没错，那么我这个登录流程就算是成功了，此时，我的res对象的status就应该置为true，即res.setStatus(true)。然后告诉用户，登录成功了！即res.setMsg(Msg.Success.Login())；**（Msg类前面已经具体介绍了，它包括Success，Fail两类，这两类下有对应的方法，其实看名字就能知道，方法内容只是输出xxx码（标记码）xxxx内容（xx成功还是失败））** 最后将res进行返回，我们就完成了一个登录的流程。

4.3 用户更改密码

更改密码也就是用户自身属性的最后一个功能了。那么我们需要如何实现这个功能捏？在生活中，我们更改密码肯定需要先登录自己的账户对吧，这个时候肯定要传账号和密码，然后更改密码就完事了。在这个系统中，我们的用户类分为管理员类和客户类，那么管理员是不是应该可以对客户的密码进行操作，就像我们忘记密码时填完一堆密保问题就能更改了，这里的场景也差不多，只不过为了简便是直接通过管理员进行更改。这里依旧只讲管理员更改密码的流程，客户和其共用一个方法，过程是一样的。

```
@GetMapping("managerop/changepassword")
public Response changePasswordManger(@RequestParam("account") String acc,
                                     @RequestParam("target") String targetAcc,
                                     @RequestParam("newpswd") String newPassword){
    return Operation.changePassword(acc, targetAcc, newPassword, AccountType.MANAGER);
}
```

可以从代码看到，我们传入的是四个数据，即acc, targetAcc, newPassword, AccountType.MANAGER，后面两个自不必多说，一个新密码，一个账户类型嘛，关键是前两个，acc, targetAcc，这里为啥要传两个账户捏，它们又有什么区别?

前面说了，我要更改密码其实是有两种方式的，一种是我作为客户自己改自己的，另一种是我忘了密码，那么你管理员就应该做到可以帮我更改密码。那此时传入的targetAcc也就是我需要更改密码的目标账户，而account就相当于是一个确认，确认啥捏，就是你的身份，是管理员还是客户。接下来我们看一下具体实现过程。

```
Response res = new Response(false, Op.CHANGE_PSWD);
LambdaQueryWrapper<Account> lTarget = new LambdaQueryWrapper<>();
LambdaQueryWrapper<Account> lSelf = new LambdaQueryWrapper<>();
lTarget.eq(Account::getName, targetAcc);
lSelf.eq(Account::getName, acc);
Account target = accountMapper.selectOne(lTarget);
Account self = accountMapper.selectOne(lSelf);
```

熟悉的步骤，先将res的status置为false，然后给op传入这是更改密码的操作。接下来就是一个查询语句，看看数据库中是否存在传入的账户和目标账户，没有就是null嘛。下面才是该功能的核心逻辑。

```
if(target == null){
    res.setMsg(Msg.Fail.TargetNotExist(targetAcc));
    return res;
}
if(self == null){
    res.setMsg(Msg.Fail.AccountNotExist(acc));
    return res;
}

switch(type){
    case AccountType.MANAGER -> {
        if(acc.equals(targetAcc)){ // change own
            performChange(newPassword, res, lTarget, target);
        }else{  // change other's
            // can't change other manager's password.            if(target.getType() ==
AccountType.MANAGER) res.setMsg(Msg.Fail.ChangeOtherManager());
            else performChange(newPassword, res, lTarget, target);
        }
    }
```

```
    case AccountType.USER -> {
        if(!acc.equals(targetAcc)){ // You can only change the password of your own.
            res.setMsg(Msg.Fail.ChangeOther());
            return res;
        }
        performChange(newPassword, res, lTarget, target);
    }
}

return res;
```

看起来有点长，我们慢慢讲，前面看到，我们把targetAcc的查询结果传给了target，acc的查询结果传给了self。

那么第一步我们得判断，你数据库中是不是有我传入的账户啊，没有我就应该报相应的错误对吧。因此两个if语句的功能就是做这个的。

账户还分管理员和客户，这两类人的操作应该是不同的，因为管理员可以操作客户的账户，而客户只能操作自己的账户。怎么办捏？这就要用到之前传入的AccoutType了。一个简单的Switch语句就能简单明了地将这两种类型分开。

先讲管理员类型，因为管理员可以更改客户账户和自身账户嘛，我们首先应该对管理员的操作目标进行一个判断。判断方法也很简单，看看你的targetAcc是不是等于acc就行了，等于的话就是操作自身账户嘛。可以看到这里用了一个新的方法 **performChange(newPassword, res, lTarget, target)**。这是啥玩意，我们来看看。

```
private static void performChange(String newPassword, Response res,
LambdaQueryWrapper<Account> lTarget, Account target) {
    target.setPassword(newPassword);
    accountMapper.updateById(target);
    if(accountMapper.selectOne(lTarget).getPassword().equals(newPassword)){
        res.setStatus(true);
        res.setMsg(Msg.Success.ChangePassword());
    }else{
        res.setMsg(Msg.Fail.ISE());
    }
}
```

这个方法的作用也很好理解，我们通过这个方法**target.setPassword(newPassword);** 来设置新的密码，然后更新目标账户 **accountMapper.updateById(target);** 这样子密码就更新了对吧，但是我是不是还应该判断我密码是不是更新成功了捏？因此我们又进行了一次判断 **if(accountMapper.selectOne(lTarget).getPassword().equals(newPassword))** 这句代码也很简单，就是我现在目标账户的密码是不是等于我传入的密码。如果等于，那么我跟改密码的操作就应该成功了，这时候我把res的status置为true，设置msg为更改密码成功。如果失败，那么msg就应该写入内部服务器错误。

说完这个方法，我们继续上面的内容，对管理员来说当targetAcc与acc不等时怎么做。我们知道账户类型分为管理员和客户，管理员可以更改客户密码，但他能不能更改其他管理员的密码捏？答案显然是不能的。因此我们还需要二次判断。**if(target.getType() == AccountType.MANAGER) res.setMsg(Msg.Fail.ChangeOtherManager())** 一个简单的逻辑，如果我的目标账户类型也是管理员账户那我就不能对整个账户进行更改，然后报个错误，不能更改其他管理员的账户密码。如果不是，那我就直接调用**performChang()** 这个方法更改密码就完事了。

讲完管理员，我们来看看客户。因为客户只能对自身进行修改，逻辑就简单很多了。

```
if(!acc.equals(targetAcc)){ // You can only change the password of your own.
            res.setMsg(Msg.Fail.ChangeOther());
            return res;
        }
        performChange(newPassword, res, lTarget, target);
    }
```

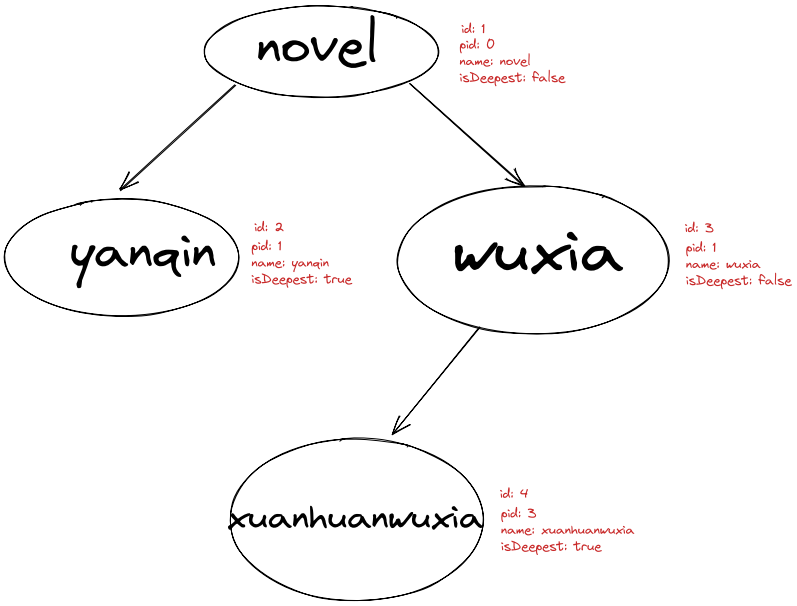
判断目标账户和自身账户是否相同，不同就报错，然后返回res就行了，相同就说明我改的是我自己的账户嘛，我就直接调用**performChang()** 这个方法更改密码。

4.4 添加书籍类(管理员)

在现实生活中，我们如果去图书馆找书肯定不是一本一本的找，那样子效率太低了，我们通常会去一个分区，比如我要找《神雕侠侣》这本书，我肯定是去小说区去找对吧。因此，我们在这个系统添加书之前，我们首先得创建书的大类，然后把它们一个一个添加到这个类之中去。接下来是这个方法的实现过程。

```
@GetMapping("managerop/addcategory")
public Response addCategory(@RequestParam("name") String name,
                            @RequestParam(value = "pname", required = false) String
pname){
    return Operation.addCategory(name, pname);
}
```

可以看到这里我们传了一个name和一个pname这两啥玩意，用来干什么的捏？其实这玩意我们早就提到过了，这里再放一下图。(不再细讲)。



提一嘴这两句话的区别，

```
@RequestParam("name") String name,
@RequestParam(value = "pname", required = false)
```

其实这个函数为 **@RequestParam(value = "xxxx", required = true)** 也就是如果默认为true，required = true也就是需求为真，看字面意思也很好理解，就是你一定要给我传值，这里设为false是因为我传的这个类不一定有父类，那我就可以不传入pname。下面我们细说一下具体方法的实现addCategory();

```
Response res = new Response(false, Op.ADD_CATEGORY);
Category c = new Category(name, true);
LambdaQueryWrapper<Category> l = new LambdaQueryWrapper<>();
l.eq(Category::getName, name);
```

将res的status置为false，然后在数据库中查一下这个类是否存在。这属于是经典套餐了，这里我们重点介绍一下新出现的类**Category** 它是个什么玩意。这里给出Category类的主要属性和比较重要的构造方法：

```
private int id;
private int pid;
private String name;
private boolean isDeepest;

public Category(String name, boolean isDeepest) {
    this.name = name;
    this.isDeepest = isDeepest;
}
```

这些属性也就是上面那个图描述的，其他也就是一些基本的构造方法和get(),set()函数了。我们传入的类属性，它的id应该是自增的，pid应该是找父亲的，不可能提前设置好。因此这里的构造方法只传了name，和isDeepest。注意到isDeepest是一个boolean类型，作用看图也很明确，就是判断是否是叶子节点。

```
if(!categoryMapper.selectList(l).isEmpty()){ // not empty, means that you shouldn't
insert
    res.setMsg(Msg.Fail.CategoryExisted(name));
    return res;
}
```

既然查找了，我们是不是还应该判断数据库中是否有这个类。如果有的话，我们肯定是要告诉你管理员数据库已经有这个类了，给你报个错，你不该再插入了。如果没有这个类，我们就可以继续我们的操作了。

```
if(null != pname){
    LambdaQueryWrapper<Category> lqw = new LambdaQueryWrapper<>();
    lqw.eq(Category::getName, pname);
    Category parent = categoryMapper.selectOne(lqw);
    if(parent != null){
        c.setPid(parent.getId());
        if(parent.isDeepest()){
            parent.setDeepest(false);
            categoryMapper.updateById(parent);
        }
    }else{
        res.setMsg(Msg.Fail.NoCategory(pname));
        return res;
    }
}
```

这段代码看起来比较复杂，其实它的作用也很简单，就是判断你传入的类是否有父亲，如果有我就在数据库中找一下你的父亲并把值赋给变量**parent** ，但是你虽然传了你的父亲，可数据库中不一定有你的父亲对吧，所以我们还得判断**parent** 是否为空，如果为空那我就应该告诉管理员数据库中找不到你这个类的父类，你弄错了什么的。如果存在**parent** 那我就应该把这个类的pid设置为父亲的id，然后判断父亲是否是最深的，如果是，我就把它的isDeepest改为false，然后把表更新一下就行了。

以上操作完成之后我们就搞完了添加一个类的前提条件，接下来就是添加这个类的过程了。

```
categoryMapper.insert(c);

if(categoryMapper.selectOne(l) != null){ // exist, insert successfully
    res.setStatus(true);
    res.setMsg(Msg.Success.AddCategory(name));
}else{
    res.setMsg(Msg.Fail.ISE());
```

```
}
return res;
```

插入类其实就这一句话categoryMapper.insert(c);但这就结束了吗，并没有，我们还需要判断类的插入是否成功。这个就相当经典了，查一下现在的表是否有这个类，有的话就把res的status置为true，然后告诉管理员类的添加成功，否则就报错，这样子一个类的添加就完成了。

4.5 删除书籍类(管理员)

有了添加类，肯定少不了删除类。和添加类一样，这是管理员才有权限进行的操作。那它又是怎么实现的捏?

```
@GetMapping("managerop/deletecategory")
public Response deleteCategory(@RequestParam("name") String name){
    return Operation.deleteCategory(name);
}
```

相比于添加类，删除类只传入了一个参数，即name。这是因为一个类的名字在数据库中是唯一的(添加类的过程可以看到)，因此，我们只要知道这个类的名字就可以找到它了。下面看一下deleteCategory()方法的具体实现。

```
Response res = new Response(false, Op.DELETE_CATEGORY);
LambdaQueryWrapper<Category> l = new LambdaQueryWrapper<>();
l.eq(Category::getName, name);
Category c = categoryMapper.selectOne(l);
```

这段话也是老一套了。作用就是根据类的名字先在数据库中查找这个类。然后把结果传给Category类型的变量c。

接下来就应该对c进行判断了。

```
if(c == null){
    res.setMsg(Msg.Fail.NoCategory(name));
    return res;
}else if(!c.isDeepest()){ // Can't delete the non-deepest category
    res.setMsg(Msg.Fail.Subcategory());
    return res;
}
```

如果我们找不到c，也就是c == null，那么你管理员肯定删错类了，我就要告诉你数据库中没有这个类，然后将res结果返回。

然后是接下来的else语句判断，这段代码什么意思捏？举个栗子，我们如果在浏览器中打开一坨窗口，然后这时候你点击右上角的关闭按钮，会弹出一个窗口提示，你是否要关闭所有窗口。这里也是类似的，但是我们应该避免在你要删除的类存在子类的情况下，对该类进行删除。因此，这里也就是判断这个类是不是父类，如果是，我就不能对其进行删除，然后报个错给管理员。

在前面的**技术点-图书层面** 的部分，我们已经讲过了，要删除一个类肯定不只是把它直接删除那么简单，我们还需要对它的父亲做出相应的变化。接下来我们就来实现这一部分。

```
Category parent = categoryMapper.selectById(c.getPid());
if(parent != null){
    LambdaQueryWrapper<Category> nos = new LambdaQueryWrapper<>();
    nos.eq(Category::getPid, parent.getId());
    nos.ne(Category::getName, c.getName());
    List<Category> sons = categoryMapper.selectList(nos);
    if(sons.isEmpty()){ // No other sons, delete and make parent the deepest.
        parent.setDeepest(true);
        categoryMapper.updateById(parent);
    }
}else if(c.getPid() == 0){ // The topmost, but also the deepest.
    // Just do nothing.
    }
    else{
    res.setMsg(Msg.Fail.ISE());
    return res
}

categoryMapper.deleteById(c.getId());//delete
```

首先我们在数据库中查找该类的pid，然后看它是否存在。
如果pid不存在，就很简单了，那你肯定错了，因为我的每个类都是存在父节点的，我就报一个服务器错误然后return res(基本不可能遇到，这里是为了补全情况)。
如果pid存在，我们就在数据库中先找到它的父亲除它以外是否还有其他的儿子。
如果有的话，这时候就比较简单了，我直接删除，因为删除一个儿子并不会影响父类是否为deepest。
但是如果只有一个儿子，我删除它肯定会影响父类的，所以我就要更改父类的isDeepest为true。然后更新父亲的数据。
还有一种特殊情况，也就是**c.getPid() == 0** 因为根节点的pid=0，这也就意味着这个点同时是根叶，那我就不需要管它了，等着删除就可以了。

```
if(categoryMapper.selectOne(l) == null){
    res.setStatus(true);
    res.setMsg(Msg.Success.DeleteCategory(name));
}else{
    res.setMsg(Msg.Fail.ISE());
}

return res;
```

同样的，删完我肯定不能就不管了，还得判断是否删除成功。我再在数据库找一下这个类是否存在，如果不存在我就认为删除成功了，把res的status置为true，然后告诉管理员类的删除成功，否则就报错，这样子一个类的删除就完成了。

4.6 查看书籍类

说完了增删，我们来看对类的查看操作，对类的查看应该是管理员和客户都能做到的事情，也就是说用户共用同一个方法，过程也十分简单。这里我们依旧以管理员为例：

```
@GetMapping("managerop/getallcat")
public CategoryResponse getAllCategoriesManager(){
    return Operation.getAllCategories();
}
```

这次的方法不用传入参数，接下来我们来看一下具体实现：

```
public static CategoryResponse getAllCategories(){
    List<Category> list = categoryMapper.selectList(null);
    CategoryResponse res = new CategoryResponse(false, Op.GET_ALL_CATEGORIES);
    if(!list.isEmpty()){
        res.setCategories(list);
        res.setStatus(true);
        res.setMsg(Msg.Success.GetAllCategories());
    }else{
        res.setMsg(Msg.Fail.NoCategory());
    }
    return res;
}
```

非常的简短，同时也非常的好理解。首先我们把查找的结果放在Category类型的集合list中。然后判断list是否为空，如果不为空，我们就把集合传入res中去，把status置true输出操作成功的消息，否则就是这个类不存在，报错，最后返回res，这个流程就算结束了。注意到这里出现了一个新的类**CategoryResponse**，让我们看一下这是干什么的。

```
public class CategoryResponse extends Response{
    private List<Category> categories;
    public CategoryResponse(boolean status, String op, Message msg) {
        super(status, op, msg);
    }
    public CategoryResponse(boolean status, String op) {
        super(status, op);
    }
    public List<Category> getCategories() {
        return categories;
    }
    public void setCategories(List<Category> categories) {
        this.categories = categories;
    }
}
```

可以看到这个类是继承Response类的。它的构造方法都是沿用父类的，那为什么要设置这个类捏？是因为我们最后肯定是把所有的类存入集合中然后展示给用户的，而Response只能反馈基本的操作信息，有没有成功啊什么的， 因此我们单独设置了一个CategoryResponse类与 Response区分开，它拥有集合的构造方法。除了这个，我们还有 BrrowerResponse(接下来遇到就不在细讲)等等，总而言之，需要将输出结果展示给用户的，我们都应该将结果存入集合便于查看。

4.7书籍操作

有了书籍类和用户，我们还得有书，得有书籍的基本的增删改查操作，接下来我们一个一个探讨。

4.7.1 添加书籍(管理员)

有了用户，实现了用户的基本功能，有了书籍类用于存放书，我们就应该开始满足用户的需求了。一个图书馆管理系统实现用户基本需要的元素不就是书嘛，要对书进行操作我们肯定得先添加书籍。因此我们先来讲一下添加书籍的实现。

```
@GetMapping("managerop/addbook")
public Response addBook(@RequestParam("isbn") String ISBN,
                        @RequestParam(value = "name", required = false) String name,
                        @RequestParam(value = "author", required = false) String author,
                        @RequestParam(value = "publisher", required = false) String
publisher,
                        @RequestParam(value = "summary", required = false) String
summary,
                        @RequestParam(value = "cover", required = false) String cover,
                        @RequestParam(value = "price", required = false) Float price,
                        @RequestParam(value = "stock", required = false) Integer stock,
                        @RequestParam(value = "category", required = false) String
category){
    return Operation.addBook(ISBN, name, author, publisher,
            summary, cover, price, stock, category);
}
```

看起来有点复杂，但其实中间这一坨都是书籍本身的属性，也就是管理员要录入一本书，要输入它的ISBN书号，书名，作者，出版商啊之类的（除了书号，其它可以不写）。接下来我们看看添加一本书的具体过程，也就是这个addBook()方法。

```
Response res = new Response(false, Op.ADD_BOOK);
LambdaQueryWrapper<Book> l = new LambdaQueryWrapper<>();
l.eq(Book::getISBN, ISBN);
```

一样的步骤，先把res的status置为false，我们要添加一本书，我们肯定得先判断这种书在数据库存不存在，如果存在我肯定不能添加这本书了，(而是对这本书进行更新，把它的数量进行变化,这属于后面的方法)如果不存在，我就可以开始添加书籍了。这里看到了一个全新的类，BOOK，我们来看一下它是来干嘛的。

```
@TableName("book")
public class Book {
    @TableId("book_id")
    private int id;

    @TableField("book_isbn")
    private String ISBN;

    @TableField("book_name")
    private String name;

    @TableField("book_author")
    private String author;

    @TableField("book_publisher")
    private String pulisher;

    @TableField("book_summary")
    private String summary;

    @TableField("book_cover")
    private String cover;

    @TableField(value = "book_price")
    private float price;

    @TableField("book_stock")
    private int stock;

    @TableField("book_category_name")
    private String category;
    }
```

它的基本属性就是这些了，其他的就是基本的有参无参构造方法和经典的get()，set()方法。这里还写了一个toString（）方法便于后面将集合中的book对象内容输出。

```
if(null != bookMapper.selectOne(1)){ // if not null, means book has existed.
    res.setMsg(Msg.Fail.BookExisted(ISBN));
    return res;
}
```

可以看到我们对书是否存在于数据库中进行了判断，如果存在我们就报告书籍已经存在的错误，然后将res进行返回。

在书籍不存在的情况，我们得先判断这本书是否属于数据库中的某个书籍类，就像武侠小说，我肯定要把它放在小说类下面。如果找的类别不存在，我肯定得报错告诉你找不到这个类别(这时就应该调用添加类的方法）。

首先判断类别是否为空，如果不为空，**if(category != null)** 那我就在数据库找一下这个类别看看它存不存在。也就是这段代码（ Category类已经讲过，这里就不细说了）：

```
LambdaQueryWrapper<Category> lc = new LambdaQueryWrapper<>();
            lc.eq(Category::getName, category);
            Category c = categoryMapper.selectOne(lc);
```

查找也就两种结果，找到了和没找到。没找到的话，我就应该报一个错即 **res.setMsg(Msg.Fail.NoCategory(category));** 也就是查找类别失败，返回res就可以了。如果找到了捏？那我就继续下去呗，因为这本书是全新的一个书种，所以我们先把它的价格和数量置零，之后通过update方法更改就行了。然后把书籍插入就行了。

```
if(price == null) price = 0F;
if(stock == null) stock = 0;
bookMapper.insert(new Book(ISBN, name, author, publisher,
        summary, cover, price, stock, category));
```

和之前一样，我们肯定不能插入就不管了，还需要判断它是真的插进去了吗，因此这里还需要在数据库中查一下现在是不是有这本书。

```
if(bookMapper.selectOne(l) != null){ // insert successfully
    res.setStatus(true);
    res.setMsg(Msg.Success.AddBook());
}else{
    res.setMsg(Msg.Fail.ISE());
}
```

如果是，我们就把res的status置true，然后告诉管理员书籍添加成功，否则就报错。最后返回res，这个流程就算正式结束了。

4.7.2 删除书籍(管理员)

有了添加书籍，我们很容易联想到还要做到删除书籍。与添加书籍一样，这也是管理员独有的方法。下面我们来看一下这是怎么做的。

```
@GetMapping("managerop/deletebook")
public Response deleteBook(@RequestParam("isbn") String ISBN){
    return Operation.deleteBook(ISBN);
}
```

可以看到，我们获取了输入的ISBN，ISBN不就是书的书号吗，这是每一本书独一无二的属性。因此我们只需要传入ISBN就可以找到对应的目标书籍了。

接下来是方法的具体实现：

```
LambdaQueryWrapper<Book> l = new LambdaQueryWrapper<>();
l.eq(Book::getISBN, ISBN);
Book book = bookMapper.selectOne(l);
Response res = new Response(false, Op.DELETE_BOOK);
```

首先在数据库中查找书籍的ISBN结果然后把它赋给book，之后对res进行基本设置。完成前置操作，我们就可以开始删书了。

这里有两种情况，第一种是书籍不存在，第二种是书籍存在。我们先来看书籍不存在的情况。

```
if(book == null){
    res.setMsg(Msg.Fail.BookNotExist(ISBN));
    return res;
}
```

如果book == null，那么数据库中就没有对应的ISBN，我们就应该把msg设为书籍不存在，然后把res返回。那么如果书籍存在捏？那我们就直接把它删除就行了。

```
bookMapper.deleteById(book);
```

除此之外，我们还需要验证书籍是否删除了。在数据库中再次查找输入的ISBN，如果没有结果，即为null，说明操作成功了，我们就把res的status置true，然后告诉管理员书籍删除成功，否则就报错。最后返回res，这个流程就算正式结束了。

```
if(bookMapper.selectOne(l) == null){ // successfully deleted.
    res.setStatus(true);
    res.setMsg(Msg.Success.DeleteBook());
}else{
    res.setMsg(Msg.Fail.ISE());
}

return res;
```

4.7.3 更新书籍(管理员)

除了添加和删除书，我们还应该做到对书的属性进行更新。比如一种书有114本，然后被借了51本。那我该书籍的数量是不是应该要做出对应的更改。同样的书籍原价25，现在降价，我是不是应该对价格进行更新。这些都是需要进行的操作，但是如果我一个一个属性的传，那效率也太低了。所以这里我们直接传入Book类的book对象。

```
@PostMapping("managerop/updatebook")
public Response updateBook(@RequestBody Book newBook){
    return Operation.updateBook(newBook);
}
```

注意到这里用了@RequestBody，这和之前的@RequestParam有什么区别捏？@RequestParam是用来传字符串的，前面说了，我这里是要传一个book，它是一个实体类，因此我们需要使用@RequestBody来获取输入的book，而使用@RequestBody需要调用post方法，这也是我们使用@PostMapping的原因。下面我们看一下updateBook()的具体实现。

```
Response res = new Response(false, Op.UPDATE_BOOK);
LambdaQueryWrapper<Book> l = new LambdaQueryWrapper<>();
l.eq(Book::getISBN, newBook.getISBN());
Book book = bookMapper.selectOne(l);
```

熟悉的代码，我们首先还是设置res，表示操作开始，并说明这是更新书籍的操作。然后在数据库中查找这个书籍，前面提到过ISBN是每种书独一无二的属性，因此我们只要对ISBN进行查询就行了。之后再把查询结果赋给book对象。

之后还是两种情况，书找到了，书没找到。

```
if(book == null){ // Book doesn't exist.
    res.setMsg(Msg.Fail.BookNotExist(newBook.getISBN()));
    return res;
}
```

如果书没找到，说明数据库中没有嘛，我就要报一个该书号的书籍不存在。然后返回res就行了。

如果找到了，我就应该用传入的(new)book属性覆盖数据库中书籍的属性(old)book。

```
LambdaUpdateWrapper<Book> luw = new LambdaUpdateWrapper<>();
luw.eq(Book::getISBN, newBook.getISBN());
bookMapper.update(newBook, luw);
res.setStatus(true);
res.setMsg(Msg.Success.UpdateBook());
return res;
```

这里我们先查找传入书籍的ISBN，对其进行更新，把newbook的属性赋给数据库中查到的book，最后把res的status置true，然后告诉管理员书籍更新成功，返回res，这个流程就算正式结束了。

4.7.4 查找书籍

接下来就是对书籍基本操作增删改查的最后一个了。书籍的查找功能是用户都要使用的，因此方法是共同的，这里以管理员为例讲一下。我们平时多少用过看书的软件找书，像QQ阅读之类的。要找一本书，我们可以通过两种方式。

第一种就是我们知道自己想看的书的名字，那我们就可以直接通过查找名字来得到这本书。

第二种就是我不知道自己具体想看什么，但是我想看小说类，那我可以在小说类中查找书籍。

因此，我们需要对这两种方法来分别实现。

4.7.4.1 通过书名查找书籍

我们先来说通过书名查找书籍。

```
@GetMapping("managerop/getbook/byname")
public BookResponse getBookByNameManager(@RequestParam("name") String name){
    return Operation.getBookByName(name);
}
```

通过书名查书那肯定得传名字，传完之后调用getBookByName()方法，就可以查到对应的书籍了。这个方法的具体实现如下：

```
public static BookResponse getBookByName(String name){
    BookResponse res = new BookResponse(false, Op.GET_BOOK_NAME);
    LambdaQueryWrapper<Book> l = new LambdaQueryWrapper<>();
    l.like(Book::getName, name); // %category%, such as "ma -> math"

    List<Book> bks = bookMapper.selectList(l);
    res.setBooks(bks);

    if(bks.isEmpty()){
        res.setMsg(Msg.Fail.NoBookName(name));
    }else{
        res.setStatus(true);
        res.setMsg(Msg.Success.GetBook());
    }
    return res;
}
```

方法还是很简单的。首先我们通过这段代码在数据库中查找相关的书籍：

```
  BookResponse res = new BookResponse(false, Op.GET_BOOK_NAME);
    LambdaQueryWrapper<Book> l = new LambdaQueryWrapper<>();
    l.like(Book::getName, name); // %category%, such as "ma -> math"
```

注意到这里有个 BookResponse类，这个其实和之前说的CategoryResponse类作用是一样的，这里就不再细讲了。第一步还是把status置为false，然后给op变量传一下我们在进行通过书名查找书的操作。

在日常生活中查书，我们如果输入一个字，它是不是会把含有这个字的所有书都显示给我们。比如我搜大学，它就应该显示出大学物理，大学英语等等含有大学的书。因此我们这里使用了 **l.like(Book::getName, name);** 和数据库中的like作用是一样的，查出包含name的所有书籍。

```
    List<Book> bks = bookMapper.selectList(l);
    res.setBooks(bks);

  if(bks.isEmpty()){
```

```
        res.setMsg(Msg.Fail.NoBookName(name));
    }else{
        res.setStatus(true);
        res.setMsg(Msg.Success.GetBook());
    }
    return res;
```

接下来我把所有查到的书存入到集合中去。这时候还没有结束，因为我这个集合还可能是空的，就是我一本书都没查到，那我肯定要给报一个输入书籍名不存在的错误。然后返回res。如果集合不是空的，那我就查到了，说明操作成功，把res的status置true，然后告诉用户根据书籍名查找成功，返回res，这个流程就算正式结束了。

4.7.4.1 通过书类查找书籍

接下来是通过书类来查找书籍。

```
@GetMapping("managerop/getbook/bycategory")
public BookResponse getBookByCategoryManager(@RequestParam("category") String category){
    return Operation.getBookByCategory(category);
}
```

因为根据类来查找，我们肯定得传入前端输入的书籍类别。这个比较简单易懂，我们看一下具体的实现过程。

```
public static BookResponse getBookByCategory(String category){
    BookResponse res = new BookResponse(false, Op.GET_BOOK_CATEGORY);
    LambdaQueryWrapper<Category> l1 = new LambdaQueryWrapper<>();
    LambdaQueryWrapper<Book> l2 = new LambdaQueryWrapper<>();
    l1.eq(Category::getName, category);
    l2.eq(Book::getCategory, category);
    if(categoryMapper.selectOne(l1) == null){ // category doesn't exist
        res.setMsg(Msg.Fail.NoCategory(category));
        return res;
    }

    List<Book> bks = bookMapper.selectList(l2);
    res.setBooks(bks);
    if(bks.isEmpty()){
        res.setMsg(Msg.Fail.NoBookCategory(category));
    }else{
        res.setStatus(true);
        res.setMsg(Msg.Success.GetBook());
    }

    return res;
}
```

这段代码看起来很长，但是逻辑还是很简单的。要通过类查找书籍，我们肯定得先判断这个类存不存在，如果类存在，那这个类里面是否有书，如果有，将查询结果返回就行了。

从代码可以看到，我们对类别和书籍分别进行了查找，把值传给l1，l2。按照之前所说，我们首先对类别进行了判断，如果为空，我们就报一个书籍类别不存在的错误，然后返回res就行了。如果不为空，那就说明类别存在。我们把类别中的所有书传给Book类型的集合bks，也就是这这两句。

```
 List<Book> bks = bookMapper.selectList(l2);
    res.setBooks(bks);
```

最后对集合进行判断，如果集合为空，那我就报错，这个类别中没有书，否则说明操作成功，把res的status置true，然后告诉用户根据书籍类查找成功，返回res，这个流程就算正式结束了。

4.8 借书(客户)

完成了系统的基本功能，在图书系统里面塞了书，接下来我们就应该来满足客户的需求了。
借了书才能还书嘛，所以我们先来讲一下借书的过程。

```
@Transactional
@GetMapping("userop/borrow")
public Response borrowBook(@RequestParam("bookid") int id,
                           @RequestParam("duration") int duration,
                           @RequestParam("fine") float fine,
                           @RequestParam("account") String account){
    return Operation.borrowBook(id, duration, fine, account);
}
```

首先来看@Transactional，这个注解是平时开发中很常用的一个注解，它能保证方法内多个数据库操作要么同时成功、要么同时失败。

在生活中，如果我们去图书馆借书，那我们肯定要选定一个账号进行操作，然后确定要借什么书，借多久，如果借书超过时间，是不是要进行罚款，罚多少钱，这些都是应该被记录的。因此，这里我们总共传入了四个参数，id，duration，fine，account。下面我们看一下borrowBook()方法的具体实现。

```
Response res = new Response(false, Op.RETURN_BOOK);
Book book = bookMapper.selectById(bookId);
Account acc = accountMapper.selectById(account);
```

第一步进行Response的设置，接下来我们对book和account分别进行了查找。因为我肯定得先对你传入的账号先进行判断对吧，如果你这账号都是错的，你借个毛的书。如果账户对了，那我接下来就得判断书籍存不存在，如果不存在，那你肯定是借不了的。

```
if(acc == null){
    res.setMsg(Msg.Fail.AccountNotExist(account));
}else if(book == null){ // Book does not exist
```

```
    res.setMsg(Msg.Fail.NoBookId(bookId));
}
```

可以看到，我们对acc与book分别进行了判断，这里还是有顺序的，得先判断账户，才能判断书籍。接下来如果这两个都存在，我还得判断书籍够不够，不如我只有3本，全都借出去了，那我肯定没办法借给你。

```
 if(book.getStock() <= 0){ // Book is out of stock.
    res.setMsg(Msg.Fail.BookOutOfStock(book.getISBN(), book.getName()));
}
```

如果书籍没了，我就告诉你ISBN的xxx书籍已经没有了。

如果上述条件都满足，那就可以开始借书了，这里分为借书成功和失败两种情况。失败就直接报错嘛，没什么好讲的，这里具体看一下成功的情况。

```
if(book.beBorrowed()){ // Successfully being borrowed.
    bookMapper.updateById(book);
    String now = sdf.format(new Date());
    Borrow borrow = new Borrow(now, book.getId(), duration, false, fine, account);
    borrowMapper.insert(borrow);
    res.setStatus(true);
    res.setMsg(Msg.Success.BorrowBook());
}else{
    res.setMsg(Msg.Fail.ISE());
}

return res;
```

这里用了一个beBorrowed()；方法，这是啥，其实就是Book类的一个stock自减方法

```
public boolean beBorrowed(){
    if(stock > 0){
        --this.stock;
        return true;
    }
    return false;
}
```

如果书籍还有余量，我就对剩余书的数量减一返回true就行了。如果没有，就是false。接着我定义了一个现在的时间now，用于标记借书的时间。这里用到了一个量sdf，用于日期格式化。

```
private static final SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd hh:mm:ss");
```

然后看到这里出现了一个新的类，叫做Borrow，那么它具体是干什么的捏？下面我们先讲一下这玩意。

```
private Integer id;
private String time;
private int bookId;
private int duration;
private boolean isOverTime;
private float fine;
private String account;
```

这里只给出了这个类的主要属性，其它都是一些构造方法和set(),get(),toString()函数。可以看到我们定义了一个id，这个id是用来干嘛的？我们对每个借书的记录都应该存储下来，这个id就是借书记录的编号。接下来就是借书开始时间time；借的书的ISBN，bookid；借书持续时间，duration；借书结束时间isovertime，因为overtime就两种情况，到了和没到，因此这里用了boolean类型；借书逾期罚款fine；借书的账号，account。

之后我就把相应的内容传给borrow变量，再将它插入到数据库中，一条借书记录就生成了。最后把res的status置true，然后告诉客户借书成功，返回res，这个流程就算正式结束了。

4.9 还书(客户)

有借书功能肯定还要有还书功能。与借书功能一样，还书功能也是客户独有的功能。

```
@Transactional
@GetMapping("userop/return")
public Response returnBook(@RequestParam("bookid")int bookId,
                           @RequestParam("damaged") boolean isDamaged,
                           @RequestParam("account") String account,
                           @RequestParam("borrowid") int borrowId){
    return Operation.returnBook(bookId, isDamaged, account, borrowId);
}
```

可以看到我们传入了bookid，damage，account，和borrowid。这些参数用来干什么捏。

传入账号自不必多说，对于还书，我们不可能是借一本书，还一本书。因此我们需要bookid与borrowid中的bookid进行匹配，判断客户还的是哪本书，所以需要这两个参数。其次，如果书籍在借阅过程中被损坏，那客户是不是要进行赔偿啊，基于此，我们还定义了一个boolean类型的变量用于记录书籍的状态。接下来是方法的具体实现：

```
Response res = new Response(false, Op.RETURN_BOOK);
Book book = bookMapper.selectById(bookId);
Account acc = accountMapper.selectById(account);
```

和借书的流程类似，第一步进行Response的设置，接下来我们对book和account分别进行了查找。如果账号为空，肯定是要报错的。接下来判断书籍是否为空，毕竟你不可能还一本不存在的书。

```
if(acc == null){
    res.setMsg(Msg.Fail.AccountNotExist(account));
```

```
    return res;
}else if(book == null){
    res.setMsg(Msg.Fail.NoBookId(bookId));
    return res;
}
```

完成了这一步，我们还需要在数据库找一下是否存在你传入的借书记录，如果不存在肯定是还不了书的，那我就要告诉你没有这条借书记录。如果存在，按照前面所说，我需要对你传入的bookid和借书记录的bookid进行匹配。毕竟你不可能拿上次借的A书来还这次的借的B书。

```
Borrow borrow = borrowMapper.selectById(borrowId);

if(borrow == null){ // Borrow record does not exist.
    res.setMsg(Msg.Fail.BorrowNotExist(borrowId));
}else if(book.getId() != borrow.getBookId()){ // Book id and borrow id does not match
each other.
    res.setMsg(Msg.Fail.BookUnMatchBorrow());
}
```

可以看到，如果不存在我就报了个借书书籍不匹配的错误。如果书籍匹配就行了吗，我们是不是还得看一下账号匹不匹配。你总不能用别人的账户给自己还书吧。因此我们还需要进行一次判断。

```
if(!account.equals(borrow.getAccount())){ // Account unmatched.
    res.setMsg(Msg.Fail.NotYourBorrow());
```

满足了还书的条件，接下来就是看你弄坏了我的书没有，如果弄坏了你小子肯定要赔钱，如果没弄坏，但是超时了，那就要罚款。

```
if(isDamaged || borrow.isOverTime()){ // Should pay the fine.
    res.setMsg(Msg.Fail.PayFine(borrow));
}
```

走完了上述流程，我就可以开始还书了。

```
 if(book.beReturned()){ // Successfully be returned.
    bookMapper.updateById(book);
    borrowMapper.deleteById(borrow);
    res.setStatus(true);
    res.setMsg(Msg.Success.ReturnBook());
}else{
    res.setMsg(Msg.Fail.ISE());
}

return res;
```

看到这里调用了一个beReturned()方法，这是干嘛的。其实和借书的beBorrowed()方法相反，它是对书籍的数量进行加一。

```
public boolean beReturned(){
    ++this.stock;
    return true;
}
```

可以看到这个方法的内容十分简单。书籍数量加一之后，我就应该更新数据库中的该书属性，然后把借书记录删除。最后把res的status置true，然后告诉客户还书成功，返回res，这个流程就算正式结束了。