最近因为项目原因在挖洞。

其实已经有段时间没挖洞了，java 都有点忘记了。

这里想整体复习一下，顺便记录下学习流程。

学习的中心思想就是最短时间学完最多的东西，就是要快。

慢 != 扎实

学了很多遍+做了练习 = 扎实

如果学一个东西感觉很吃力不是自己的原因，是因为很多前置知识没有掌握，这个时候应该先把简单的前置知识掌握了，学起来会好很多。

线性思维我实操下来真的不好使，学到后面，前面的东西又忘记了。

环形思维好用很多。

不求一次完美，用一次次 looper 来加深理解，最后效果很好。

我这个快速学习，经过我自己实践证明，是很好用的。

因为我大学看高数考前突击也就是三天时间（基本没去上过课），其他简单的课也就是一晚上的时间（基本没上过课），最后分数虽然没有很高，但是都过了。

但快速学习也不是瞎学，有以下要点

1、针对实战培养感觉：

也就是说，先直面最终需要解决的实际问题，然后再反哺回来，看自己需要哪些知识，切忌一上来就对着基础知识一个个啃，效率非常之低下，而且没有感觉，因为学了不知道有啥用。

就像大学老师划重点，我就看他画的重点，所以才能学的那么快，而且也能过关。

28 法则在哪里都适用，有些东西用的少，就少看，用的多，就多看，眉毛胡子一把抓，势必分散精力，最后效果不好。

2、不会的问题不纠结

不会不是我智商有问题，而是我的方法有问题。

记住这句话，问题先解决 99%。

如果世界上的问题都可以用死磕来解决，那么也不需要什么方法了。

我很不喜欢那些误导人的文章和言论，说一些什么鬼话，什么天天努力到吐啥的，什么遇到难题坚持不懈，什么头昏脑胀还坚持学习。

都他妈放屁的，努力到吐，说明过载了。

机器过载会影响使用寿命，人也是一样，可持续发展才是硬道理。

坚持不懈，中等，简单难度的东西还行，遇到很难的问题你试试，你再怎么坚持也不可能解出来，因为少了前置的东西。

训练大脑就像训练肌肉一样，讲究循序渐进。

比如举重运动，就算是奥运会冠军，都是从 30kg，40kg，50kg 这样一点点累加上来的，没有人没经过训练一上来就是 100kg，200kg，冠军也不行，也要遵守基本方法。

大脑也是一样，只能学习稍微超出当前范围一点点的知识，这个范围称为学习区，超出了，大脑就当机，当机的表现就是昏昏欲睡，注意力分散，这是基本规律，不信可以试试。

3、注意劳逸结合

劳逸结合不是一句空话，而是一句实在话。

就像健身一样，跑步累了，就要休息，睡一觉，然后第二天再继续训练，或者第三天。

学习也是一样，只不过跑步锻炼的是身体的肌肉，学习锻炼的是大脑的肌肉。

学累了，就要休息，有时候不一定是睡觉，就像跑步累了不一定是睡觉一样。

可以打打游戏，玩玩手机之类的，放松一下，然后过一会其实感觉又可以了，那就继续学。

以上三点，道理都很简单，但是世界上偏偏充满了很多不希望别人学好的人，于是放出了大量的毒鸡汤，过分强调意志力还有一些主观层面的东西的重要性，是真的影响别人学习。

意志力是什么？比如一个人举重，平常只能举起 100kg，然后现在参加比赛，面对这么多观众，心中激荡，于是决定挑战自我，举 110kg，最后成功了。

意志力的作用，更多是面对极限环境的一种自我激素调动，然后让自己的能力能够比平时超出一些范围，但不多。

而且超出极限，还有副作用，需要更多的休息。

超出那一点点极限，然后浪费我更多的时间来休息，我认为不划算。

因此我除了比赛的时候，平时基本用不到意志力，就是遵循客观规律，规划时间，学习工作生活也没耽误，还有时间打游戏。

学习能力是贯穿人的一生的，是我认为最值得培养的一种能力，没有之一。

而且一通百通，这套方法其实放在任何学科都适用。

不要和身体产生的事实进行抗辩，身体饿了，就去吃饭，身体困了，就睡觉，身体注意力分散，就去玩玩游戏或者看看视频，要遵循身体基本的物理规律，因为身体的底层代码不是我们自己写的，我们没法改，只能遵循他写好的规律，我们去用，就像我们使用任何工具一样，遵循当前工具本身的使用规律，就能取得事半功倍的效果。

遵循规律，利用规律，最后组合创新规律，最后拥有体检更佳的人生之旅，真的不试一试嘛。

以下是我自己的 java 学习实例，仅作为参考。

---java 部分学习记录---

我 java 不是零基础，但我自己感觉跟零基础也差不多>_<，所以感觉这个流程，大家初入门，或者如果很久没看 java 的人，应该也可以用。

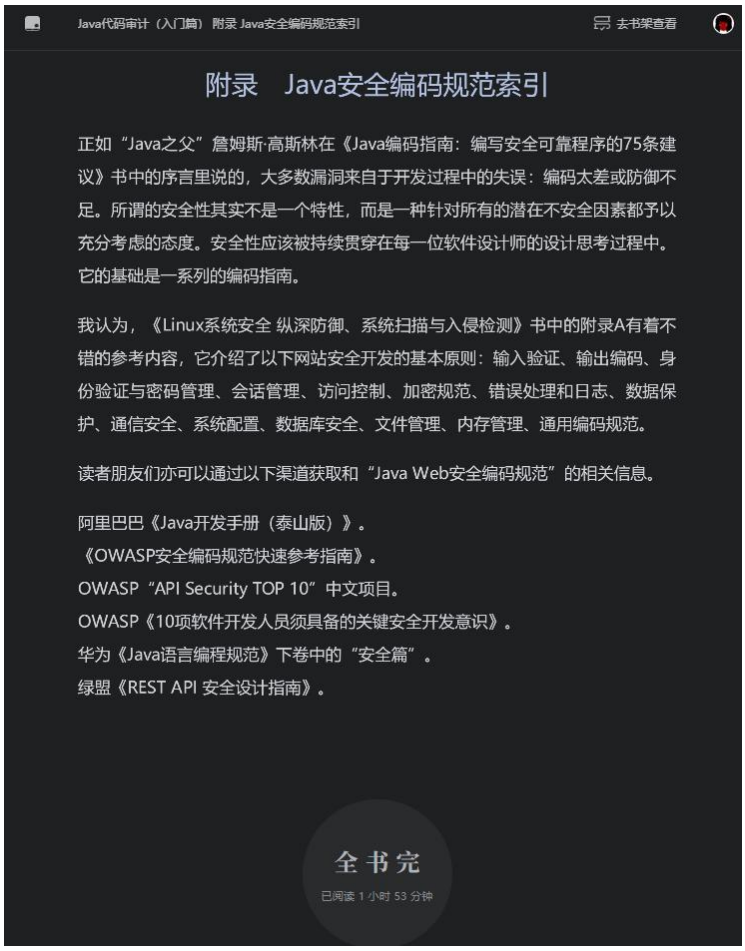
预计用 5-7 天时间，想把整个 java 的基础+web+框架整个过一遍。

下面边写会边列出时间线

---20220505 12:00---

1、全局观建立

很快的时间过了一遍 java 代码审计这本书



我用的是微信读书看的

东西不用都细看，反正我就看 rce 相关的东西。
比如

5.6 安全配置错误

- 5.6.1 安全配置错误漏洞简介
- 5.6.2 Tomcat任意文件写入（CVE-2017-12615）
- 5.6.3 Tomcat AJP 文件包含漏洞（CVE-2020-1938）
- 5.6.4 Spring Boot远程命令执行
- 5.6.5 小结

5.8 不安全的反序列化

- 5.8.1 不安全的反序列化漏洞简介
- 5.8.2 反序列化基础
- 5.8.3 漏洞产生的必要条件
- 5.8.4 反序列化拓展
- 5.8.5 Apache Commons Collections反序列化漏洞
- 5.8.6 FastJson反序列化漏洞
- 5.8.7 小结

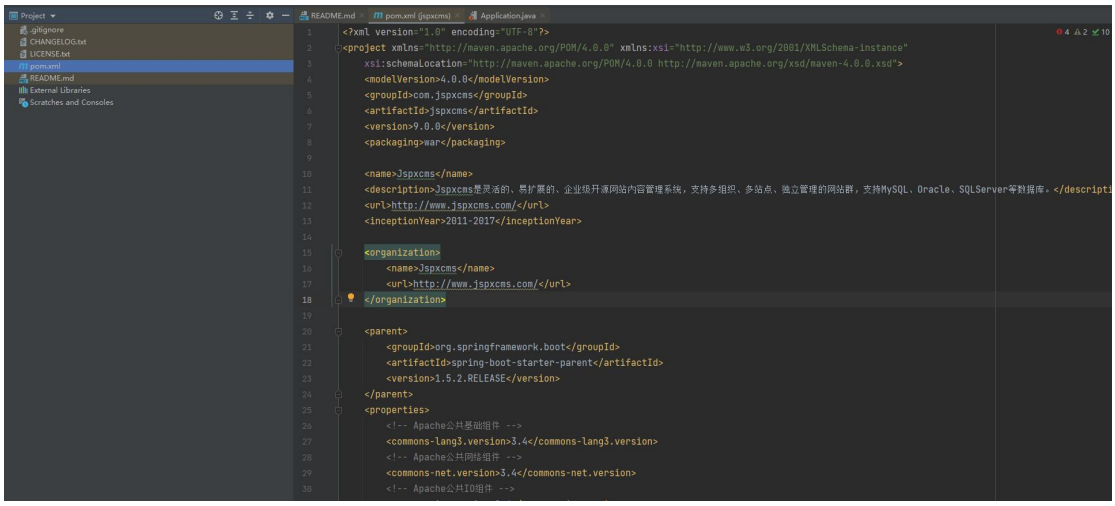
5.9 使用含有已知漏洞的组件

- 5.9.1 组件漏洞简介
- 5.9.2 Weblogic中组件的漏洞
- 5.9.3 富文本编辑器漏洞
- 5.9.4 小结

然后心中大概有数了
不要求完全记下来，但是需要了解大概 java 代码审计是要干什么

---20220505 14:44---

2、做题
先做一套题练练手，那么这里选择书中举例的 jspxcms
初期做题一定要选有答案的题，不然很容易打击信心。
先知上也有答案可以参考
//https://xz.aliyun.com/t/10891
然后发现一个问题
比如第一步，环境搭建



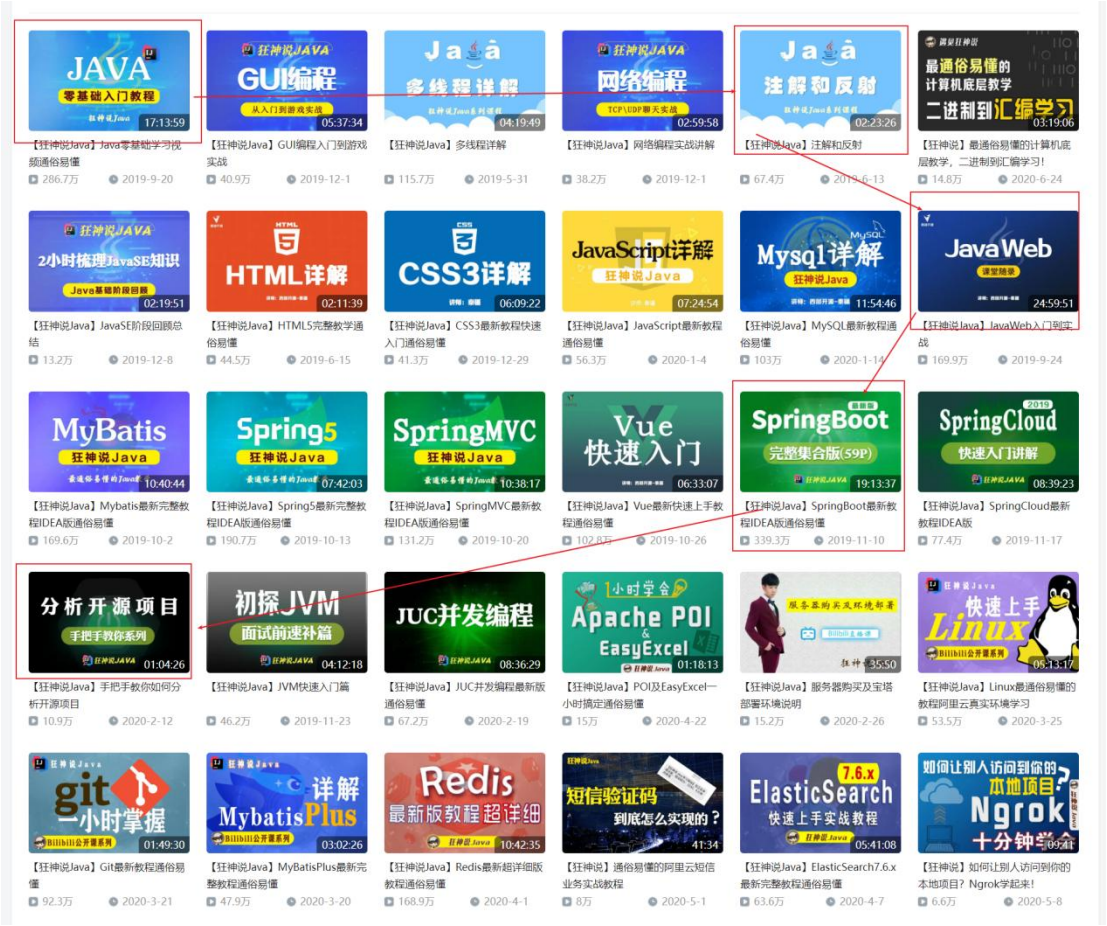
东西我都搞好了，但是 springboot 跑不起来

这里排错出现了问题，得去看 java 开发基础。

推荐 b 站看视频+看文字教程结合

视频的好处在于他的东西你都能复现出来

文字教程的好处在于速度快，但是他写的东西你不一定能够都复现出来，因为时代的原因或者因为作者省略了一些步骤。



打算按照上面红线的顺序过一遍

---20220505 20:02---

睡了一觉，然后吃了个饭，回来就七点了。

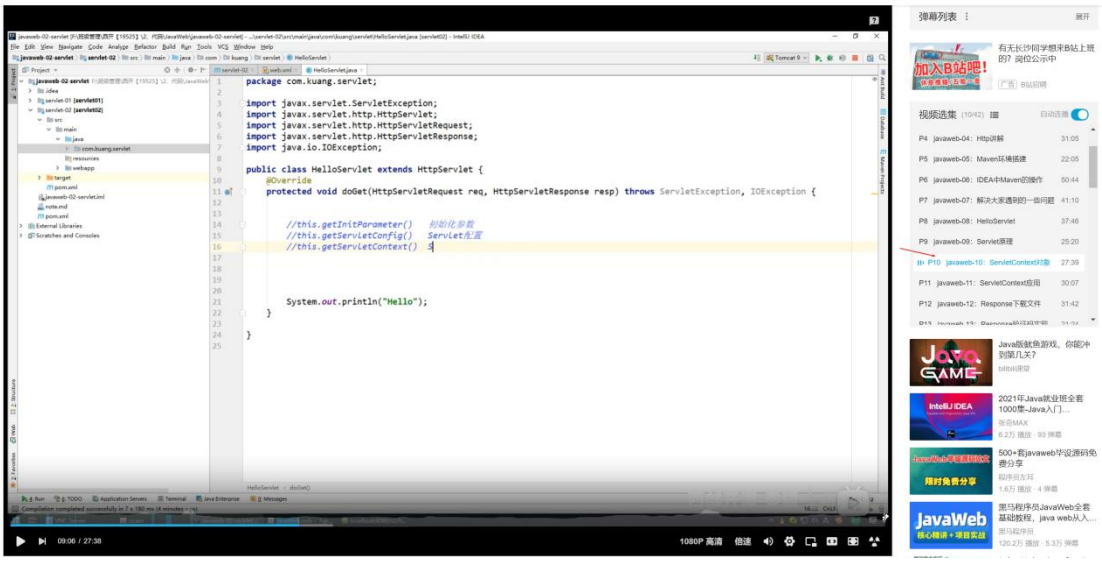
Springboot 的东西看了七小节

看不下去了，因为没应用场景，而且少了前置知识

跳到 javaweb 那边去

会的我就不看了

直接查漏补缺，看不会的



从 servletContext 相关开始看起

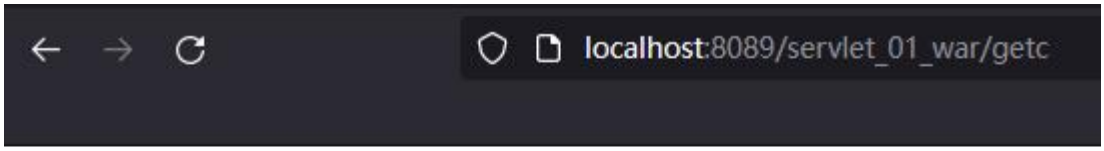
配置 context

```java
package com.fuck.servet;

import ...

public class HelloServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
        ServletContext context = this.getServletContext();
        String username = "fucku";
        context.setAttribute( name: "username", username);
    }

    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
        super.doPost(req, resp);
    }
}
```

Web.xml 定义

```
<servlet>
  <servlet-name>fuck</servlet-name>
  <servlet-class>com.fuck.servet.GetServlet</servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name>fuck</servlet-name>
  <url-pattern>/fuck</url-pattern>
</servlet-mapping>

</web-app>
```

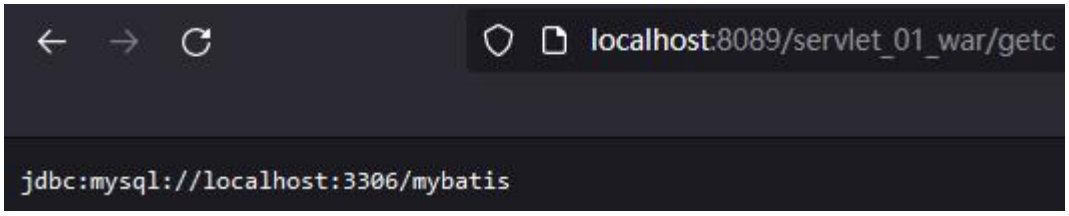
取出 context



看的时候多用>键来快进，会的就不看了，别耽误时间。
应用 1
获取 param
配置好 web.xml

```
<context-param>
  <param-name>url</param-name>
  <param-value>jdbc:mysql://localhost:3306/mybatis</param-value>
</context-param>
```

获取



Context 理解
就是一个应用中的一个公共文件夹，可以往这里面放东西，然后大家都可以用。

Context 转发：

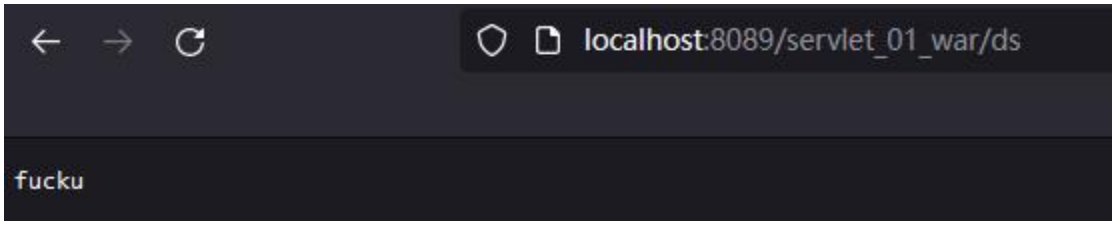
转发器

```java
public class DispatchServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
        ServletContext context  = this.getServletContext();
        RequestDispatcher requestDispatcher = context.getRequestDispatcher( path: "/fuck");
        requestDispatcher.forward(req, resp);
    }

    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
        super.doPost(req, resp);
    }
}
```

被转发的输出单元

```java
public class FuckServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
        resp.getWriter().print("fucku");
    }

    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
        super.doPost(req, resp);
    }
}
```

结果



转发和重定向

转发 a-b-c

重定向 a-b b-a a-c

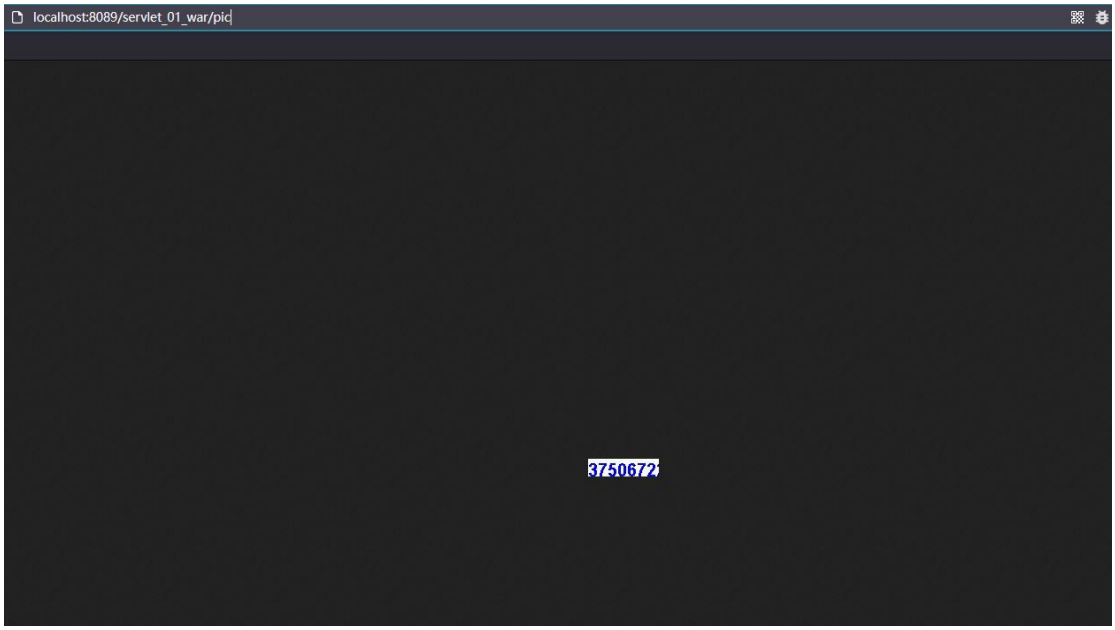
文件下载

```java
public class FileDownload extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
        //获取文件下载路径
        String realPath =                                   \java\\projectnormal\\servlet-01\\target\\classes\\fuck.png";
        System.out.println("path:" + realPath);
        //下载的文件name是啥
        String filename = realPath.substring(realPath.lastIndexOf( str: "\\") + 1);
        //想办法让浏览器支持我们需要下载的东西
        resp.setHeader( name: "Content-Disposition", value: "attachment:filename=" + filename);
        //获取下载输入
        FileInputStream in = new FileInputStream(realPath);
        //创建缓冲区
        int len = 0;
        byte[] buffer = new byte[1024];
        //获取outputstream
        ServletOutputStream out = resp.getOutputStream();
        //将fileoutputstream写入buffer
        while((len = in.read(buffer)) > 0){
            out.write(buffer, off: 0,len);
        }
```

结果

验证码

```java
@Override
protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
    //浏览器3s刷新一次
    resp.setHeader( name: "refresh",  value: "3");
    //定义一个图片
    BufferedImage image = new BufferedImage( width: 80, height: 20,BufferedImage.TYPE_INT_RGB);
    //笔
    Graphics2D g = (Graphics2D) image.getGraphics();
    //设置背景颜色
    g.setColor(Color.white);
    g.fillRect( x: 0, y: 0, width: 80, height: 20);
    //写入

    g.setColor(Color.BLUE);
    g.setFont(new Font( name: null,Font.BOLD, size: 20));
    g.drawString(makeNum(), x: 0 , y: 20);

    //告诉浏览器这个请求是图片
    resp.setContentType("image/jpeg");
    //清除浏览器缓存
    resp.setDateHeader( name: "expires",  date: -1);
    resp.setHeader( name: "Cache-Control",  value: "no-cache");
    resp.setHeader( name: "Pragma",  value: "no-cache");

    //传给浏览器
    boolean write = ImageIO.write(image,  formatName: "jpg",resp.getOutputStream());
}

private String makeNum(){
    Random random = new Random();
    String num = random.nextInt( bound: 99999999) + "";
    StringBuffer sb = new StringBuffer();
    for(int i = 0;i < 7 - num.length();i++){
        sb.append("0");
    }
    num = sb.toString() + num;
```



这里觉得他写的这个验证码算法有点意思，做了个小实验

```java
public class Test11 {

    public static void main(String[] args) {
        Random random = new Random();
        String num = random.nextInt( bound: 9999) + "";

        System.out.println(num);
        System.out.println(num.length());
        StringBuffer sb = new StringBuffer();
        for(int i = 0;i < 7 - num.length();i++){
            sb.append("0");//相当于位数不够就在前面补0
        }
        num = sb.toString() + num;
        System.out.println(num);
    }

}
```

结果

```
"C:\Program Files\Java\jdk1.8.0_181\bin\java.exe" ...
6689
4
0006689
```

重定向

B一个web资源收到客户端A请求后，B他会通知A客户端去访问另外一个web资源C，这个过程叫重定向

和转发有区别

很简单

代码：

```java
public class RedictServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
        resp.sendRedirect( location: "/servlet_01_war/fuck");
    }

    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
        super.doPost(req, resp);
    }
```

抓包：

Request

Pretty Raw \n Actions

```
GET /servlet_01_war/redict HTTP/1.1
Host: localhost:8089
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:99.0) Gecko/20100101 Firefox/99.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
Accept-Language: zh-CN,zh;q=0.8,zh-TW;q=0.7,zh-HK;q=0.5,en-US;q=0.3,en;q=0.2
Accept-Encoding: gzip, deflate
Connection: close
Cookie: JSESSIONID=D715A742D22EAD30360C8E74C0D7DB6D; Idea-7f7ed34b=37b0a42b-b86c-4f53-88b6-097493e07446; JSESSIONID=54D6F39E5BE50F54B845954D7C2C0DED
Upgrade-Insecure-Requests: 1
Sec-Fetch-Dest: document
Sec-Fetch-Mode: navigate
Sec-Fetch-Site: none
Sec-Fetch-User: ?1
```

Response

Pretty Raw Render \n Actions

```
HTTP/1.1 302
Location: /servlet_01_war/fuck
Content-Length: 0
Date: Thu, 05 May 2022 14:27:56 GMT
Connection: close
```

结果：

Request

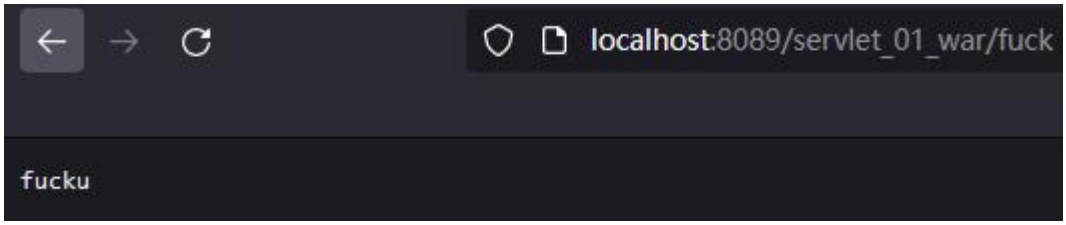
Pretty Raw \n Actions

```
GET /servlet_01_war/fuck HTTP/1.1
Host: localhost:8089
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:99.0) Gecko/20100101 Firefox/99.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
Accept-Language: zh-CN,zh;q=0.8,zh-TW;q=0.7,zh-HK;q=0.5,en-US;q=0.3,en;q=0.2
Accept-Encoding: gzip, deflate
Connection: close
Cookie: JSESSIONID=D715A742D22EAD30360C8E74C0D7DB6D; Idea-7f7ed34b=37b0a42b-b86c-4f53-88b6-097493e07446; JSESSIONID=54D6F39E5BE50F54B845954D7C2C0DED
Upgrade-Insecure-Requests: 1
Sec-Fetch-Dest: document
Sec-Fetch-Mode: navigate
Sec-Fetch-Site: none
Sec-Fetch-User: ?1
```

Response

Pretty Raw Render \n Actions

```
HTTP/1.1 200
Content-Length: 5
Date: Thu, 05 May 2022 14:28:13 GMT
Connection: close

fucku
```

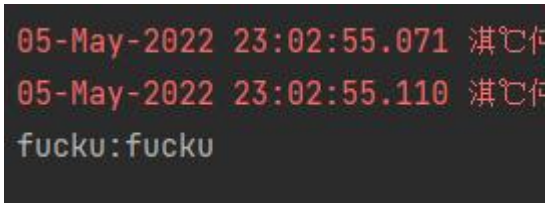


HttpRequest

写了两行

```java
public class RequestTest extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {

        req.setCharacterEncoding("utf-8");
        String username = req.getParameter( name: "username");
        String password = req.getParameter( name: "password");

        System.out.println(username + ":" + password);

    }
```

结果：



简单

XD

Cookie

就是 ticket

很多都是票据逻辑

电影票凭票入场等等

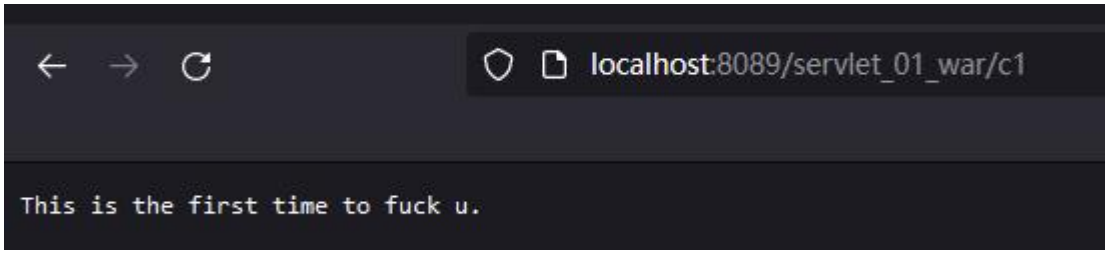
kerberos 也是

代码

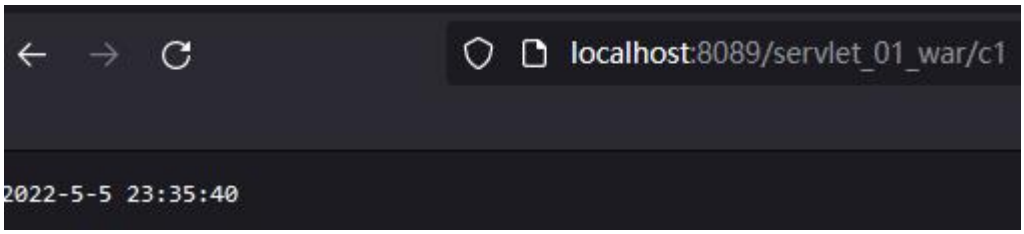
```java
protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
    req.setCharacterEncoding("utf-8");
    resp.setCharacterEncoding("utf-8");

    PrintWriter out = resp.getWriter();
    Cookie[] cookies = req.getCookies();

    if (cookies != null){
        //取出cookie
        for(int i = 0;i < cookies.length; i++){
            Cookie cookie = cookies[i];
            if(cookie.getName().equals("LastLoginTime")){
                long lastLoginTime = Long.parseLong(cookie.getValue());
                Date date = new Date(lastLoginTime);
                out.write(date.toLocaleString());
            }
        }

    }else{
        out.write( s: "This is the first time to fuck u.");
    }
    Cookie cookie = new Cookie( name: "LastLoginTime",  value: System.currentTimeMillis() + "");
    resp.addCookie(cookie);
}
```

结果

第一次访问



第二次访问



---20220506 14：41---

打了一通宵游戏，睡到刚刚才起

Session

创建 session

```
@Override
protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletExcep
    HttpSession session = req.getSession();
    session.setAttribute( name: "name",  value: "fuck");
    String sessionId = session.getId();

    if(session.isNew()){
        resp.getWriter().write( s: "session created:" + sessionId);
    }else{
        resp.getWriter().write( s: "session already exsists" + sessionId);
    }
}
```

读取 session

```
protected void doGet(HttpServletRequest req, HttpServletResponse resp) t
    req.setCharacterEncoding("UTF-8");
    resp.setCharacterEncoding("UTF-8");
    resp.setContentType("text/html;charset=utf-8");

    HttpSession session = req.getSession();


    String session1 = (String) session.getAttribute( name: "name");
    resp.getWriter().print(session1);

}
```

就理解为字符串的增删改查就行

流程
第一次请求 给 cookie
第二次请求 带上 cookie cookie 中有 sessionsid，服务端用 sessionid 来区分用户是谁

Jsp 原理剖析
Jsp 看起来是前端技术，其实是后端技术
就是在 jsp 中用 jsp 的格式来写 java

Jsp 基础语法
看看即可
到时候写的时候再查

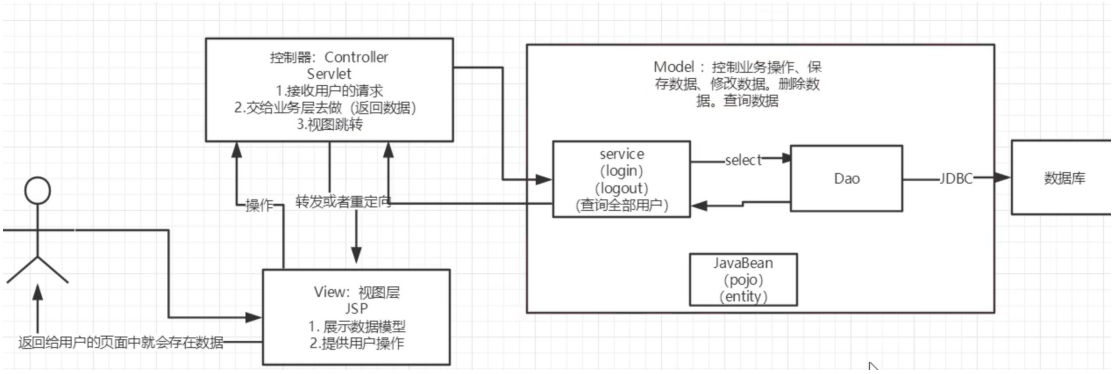
内置对象和作用域
看看即可

Jstl 标签
一种简化的 jsp 语法
写马过 waf 可能有用
先了解即可

Javabean
和 jsp 联动的一个东西
Javabean 理解为一个公共类
大家都可以用
然后就不用重复写这个类了
写在 pojo 里就行

MVC
M model
V view
C controller
早期架构：
Servlet 中写了 crud（增删改查），也就是说 servlet 中有具体逻辑，高耦合，不便维护

演进架构：



他画的很清楚了

相当于业务逻辑 数据库 实体类 各自独立
然后 javabean 作为公共实体类单独分出来
Service 写具体的业务逻辑
Dao 层写 select 之类的查询语句去跟数据库交互

低耦合

在 springcloud 之前都是这个架构

Filter //内存马可用

定义一个 filter 类

```
@Override
public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain) throws IOException, Serv
    request.setCharacterEncoding("utf-8");
    response.setCharacterEncoding("utf-8");
    response.setContentType("text/html;charset=UTF-8");

    System.out.printf("fucku");
    chain.doFilter(request, response);//转接请求
}
```

配置路由

```
<filter>
  <filter-name>Filter111</filter-name>
  <filter-class>com.fuck.servet.FilterDemo1</filter-class>
</filter>

<filter-mapping>
  <filter-name>Filter111</filter-name>
  <url-pattern>/aaa</url-pattern>
</filter-mapping>
```

然后访问

回显

```
fuckufuckufuckufuckufucku
```

Listener 监听器//内存马可用

像个地雷一样，一碰开关就会爆炸

比如这个 sessionCreated

如果创建了 session 就会触发这个机制

```
public void sessionCreated(HttpSessionEvent se) {
```

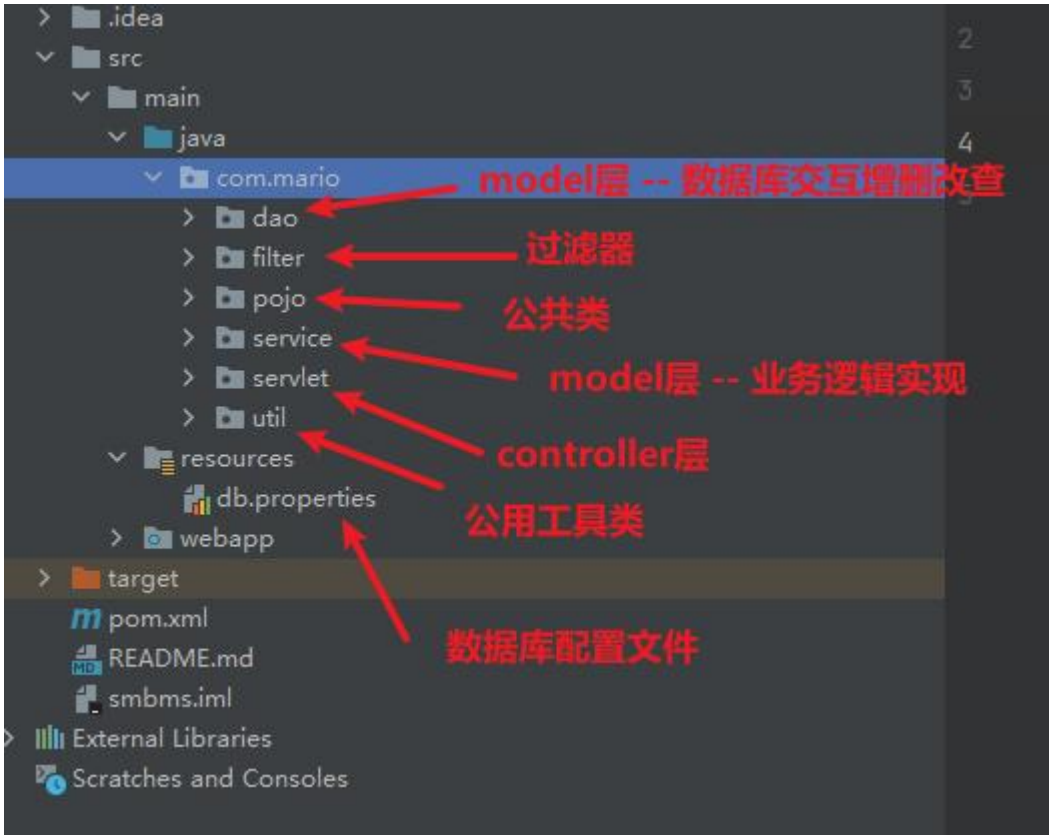
代码懒得全写了

利用 filter 来实现权限拦截
放一个 filter 放到/servlet/*下面
然后判断 session
如果 session 为空
直接跳转到别的页面
Session

Jdbc 事务
要么都成功
要么都失败
ACID 原则

---20220507 0:33---
晚上十点多睡了一觉 现在醒来了

smbms 项目搭建
//通过自己做一个项目来熟悉一个项目的框架，将来方便审计
//整完这个项目 基础的 javaweb 估计就差不多了
//后续也可以根据这个架构演进使用 spring mvc springboot 等
为了节约时间 这里代码就不手打了
直接下了一套一样的
老规矩 先看架构



先从前端页面看起

死活登不进去

排错

```
java.sql.SQLException Create breakpoint : The server time zone value '?й???????' is unrecognized or represents
    at com.mysql.cj.jdbc.exceptions.SQLError.createSQLException(SQLError.java:129)
    at com.mysql.cj.jdbc.exceptions.SQLError.createSQLException(SQLError.java:97)
    at com.mysql.cj.jdbc.exceptions.SQLError.createSQLException(SQLError.java:89)
    at com.mysql.cj.jdbc.exceptions.SQLError.createSQLException(SQLError.java:63)
    at com.mysql.cj.jdbc.exceptions.SQLError.createSQLException(SQLError.java:73)
```

说老子时区有问题 妈的

Jdbc 加上一行连接

```
SL=true&serverTimezone=GMT%2B8
```

可以了

```
LoginServlet--start....
wen
123
```

一个比较简单的系统
源码从前端开始看起
一个登录页面

```jsp
<header class="loginHeader">
    <h1>超市订单管理系统</h1>
</header>
<section class="loginCont">
    <form class="loginForm" action="${pageContext.request.contextPath }/login.do"  name="actionForm" id="actionForm"  method="post" >
        <div class="info">${error }</div>
        <div class="inputbox">
            <label >用户名：</label>
            <input type="text" class="input-text" id="userCode" name="userCode" placeholder="请输入用户名" required/>
        </div>
        <div class="inputbox">
            <label >密码：</label>
            <input type="password" id="userPassword" name="userPassword" placeholder="请输入密码" required/>
        </div>
        <div class="subBtn">

            <input type="submit" value="登录"/>
            <input type="reset" value="重置"/>
        </div>
    </form>
```

传参往

```jsp
<form class="loginForm" action="${pageContext.request.contextPath }/login.do"  name="actionForm" id="actionForm"  method="post" >
```

这里传
看 web.xml 其中的 servlet

```xml
<!--    注册登录页面的Servlet-->
    <servlet>
        <servlet-name>LoginServlet</servlet-name>
        <servlet-class>com.mario.servlet.user.LoginServlet</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>LoginServlet</servlet-name>
        <url-pattern>/login.do</url-pattern>
    </servlet-mapping>
<!--    注册注销页面-->
```

定位具体的 servlet 类

```java
@Override
protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {

    System.out.println("LoginServlet--start....");
    //获取用户名和密码
    String userCode = req.getParameter( name: "userCode");
    String userPassword = req.getParameter( name: "userPassword");
    //和数据库中的密码进行对比，调用业务层；
    UserService userService = new UserServiceImpl();
    User user = userService.login(userCode, userPassword);  //这里已经把登录的人给查出来了
    System.out.println(userCode);
    System.out.println(userPassword);
    if (user!=null){ //查有此人，可以登录
        //将用户的信息放到Session中；
        req.getSession().setAttribute(Constants.USER_SESSION,user);
        //跳转到主页重定向
        resp.sendRedirect( location: "jsp/frame.jsp");
    }else {//查无此人，无法登录
        //转发回登录页面，顺带提示它，用户名或者密码错误；
        req.setAttribute( name: "error", o: "用户名或者密码不正确，傻批！");
        req.getRequestDispatcher( path: "login.jsp").forward(req,resp);
    }
```

这里由 view 层转到 controller 层

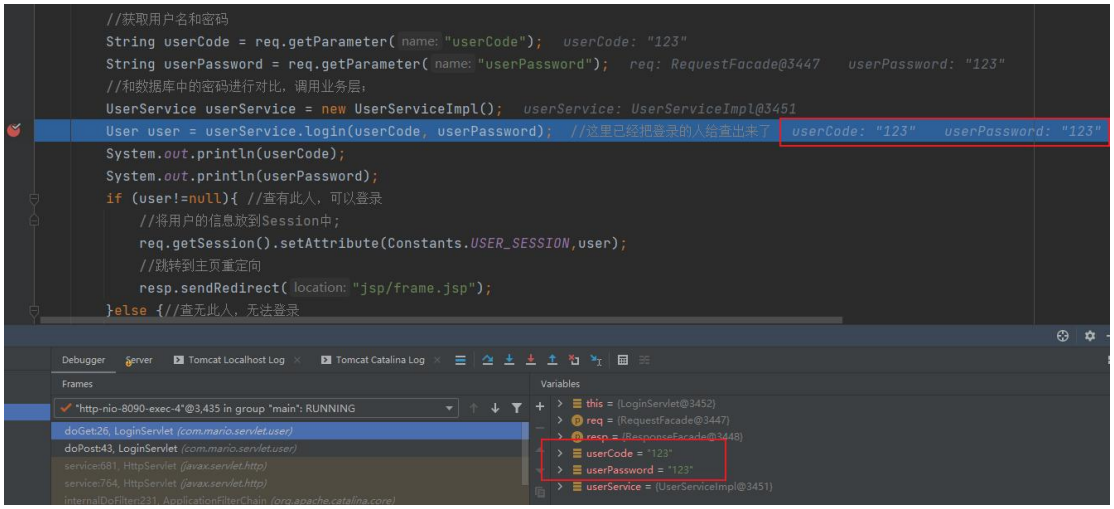
然后这里的具体查询登录的人的账号密码

Controller 没有具体的业务逻辑

业务逻辑在 service 里面完成

```java
UserService userService = new UserServiceImpl();
User user = userService.login(userCode, userPassword);  //这里已经把登录的人给查出来了
```

下个断点调试下 直观一些



Into 进来看一下

```
gin.jsp × web.xml × LoginServlet.java × UserServiceImpl.java ×

    public User login(String userCode, String password) {  userCode: "123"    password: "123"
        Connection connection = null;
        User user = null;

        try {
            connection = BaseDao.getConnection();
            //通过业务层调用对应的具体的数据库操作
            user = userDao.getLoginUser(connection, userCode);
        } catch (SQLException e) {
            e.printStackTrace();
        }finally {
            BaseDao.closeResource(connection, preparedStatement: null, resultSet: null);
        }
        //匹配密码
        if(null != user){
            if(!user.getUserPassword().equals(password))
                user = null;
        }
        return user;
    }
```

还是没有具体的数据库查询语句

只是做了逻辑匹配

再 into 进去看看

```
public User getLoginUser(Connection connection, String userCode) throws SQLException {
    PreparedStatement pstm = null;
    ResultSet rs = null;
    User user = null;

    if (connection!=null){
        String sql = "select * from smbms_user where userCode=?";
        Object[] params = {userCode};
        //System.out.println(userPassword);
        rs = BaseDao.execute(connection,sql,params,rs,pstm);
        if (rs.next()){
```

通过 username 查用户的所有数据

```
    if (connection!=null){
        String sql = "select * from smbms_user where userCode=?";  sql: "select * from smbms_user where userCode=?"
        Object[] params = {userCode};  userCode: "111"    params: Object[1]@4175
        //System.out.println(userPassword);
        rs = BaseDao.execute(connection,sql,params,rs,pstm);  connection: ConnectionImpl@4174    sql: "select * from smbms_user where
        if (rs.next()){
            user = new User();
            user.setId(rs.getInt( columnLabel: "id"));
            user.setUserCode(rs.getString( columnLabel: "userCode"));
            user.setUserName(rs.getString( columnLabel: "userName"));
            user.setUserPassword(rs.getString( columnLabel: "userPassword"));
            user.setGender(rs.getInt( columnLabel: "gender"));
            user.setBirthday(rs.getDate( columnLabel: "birthday"));
            user.setPhone(rs.getString( columnLabel: "phone"));
            user.setAddress(rs.getString( columnLabel: "address"));
            user.setUserRole(rs.getInt( columnLabel: "userRole"));
            user.setCreatedBy(rs.getInt( columnLabel: "createdBy"));
            user.setCreationDate(rs.getTimestamp( columnLabel: "creationDate"));
            user.setModifyBy(rs.getInt( columnLabel: "modifyBy"));
            user.setModifyDate(rs.getTimestamp( columnLabel: "modifyDate"));  user: null
        }
        BaseDao.closeResource( connection: null,pstm,rs);  pstm: null    rs: "com.mysql.cj.jdbc.result.ResultSetImpl@15651bf2"
```

如果查不到，走这段逻辑

```
if (user!=null){ //查有此人，可以登录
    //将用户的信息放到Session中；
    req.getSession().setAttribute(Constants.USER_SESSION,user);
    //跳转到主页重定向
    resp.sendRedirect( location: "jsp/frame.jsp");
}else {//查无此人，无法登录
    //转发回登录页面，顺带提示它，用户名或者密码错误；
    req.setAttribute( name: "error", o: "用户名或者密码不正确，憨批！ ");
    req.getRequestDispatcher( path: "login.jsp").forward(req,resp);
```

如果能查到

```
rs = BaseDao.execute(connection,sql,params,rs,pstm);   connection: ConnectionImpl@4206   pstm: null   sql: "select * fr
if (rs.next()){
    user = new User();
    user.setId(rs.getInt( columnLabel: "id"));
    user.setUserCode(rs.getString( columnLabel: "userCode"));
    user.setUserName(rs.getString( columnLabel: "userName"));
    user.setUserPassword(rs.getString( columnLabel: "userPassword"));   rs: "com.mysql.cj.jdbc.result.ResultSetImpl@1130e4
    user.setGender(rs.getInt( columnLabel: "gender"));
    user.setBirthday(rs.getDate( columnLabel: "birthday"));
    user.setPhone(rs.getString( columnLabel: "phone"));
    user.setAddress(rs.getString( columnLabel: "address"));
    user.setUserRole(rs.getInt( columnLabel: "userRole"));
    user.setCreatedBy(rs.getInt( columnLabel: "createdBy"));
    user.setCreationDate(rs.getTimestamp( columnLabel: "creationDate"));
    user.setModifyBy(rs.getInt( columnLabel: "modifyBy"));
    user.setModifyDate(rs.getTimestamp( columnLabel: "modifyDate"));
}
BaseDao.closeResource( connection: null,pstm,rs);
```

一个个设置实体类的参数

最后把这个实体类 return 回去

```
            user.setCreatedBy(rs.getInt( columnLabel: "createdBy"));
            user.setCreationDate(rs.getTimestamp( columnLabel: "creati
            user.setModifyBy(rs.getInt( columnLabel: "modifyBy"));
            user.setModifyDate(rs.getTimestamp( columnLabel: "modifyDa
        }
        BaseDao.closeResource( connection: null,pstm,rs);
    }
    return user;
}
```

然后调用实体类的基础 get 方法获取 password 进行比较

```
    }
    //匹配密码
    if(null != user){
        if(!user.getUserPassword().equals(password))   password: "123"    user: User@4212
            user = null;
    }
    return user;
```

最后又跳回 controller 层
走了登录成功的逻辑

```
    User user = userService.login(userCode, userPassword);  //这里已经把登录的人给查出来了   userService: UserServiceImpl@
    System.out.println(userCode);   userCode: "wen"
    System.out.println(userPassword);   userPassword: "123"
    if (user!=null){ //查有此人，可以登录
        //将用户的信息放到Session中；
        req.getSession().setAttribute(Constants.USER_SESSION,user);   req: RequestFacade@4154    user: User@4212
        //跳转到主页重定向
        resp.sendRedirect( location: "jsp/frame.jsp");   resp: ResponseFacade@4155
```

这里是我自己操作的，我还是跟着他的流程走一遍
关键是学习开发思路
有了开发思路之后对白盒审计很有好处

然后我这里算了一下他的流程
他还剩下的课时是九个多小时
因为这个项目是 javaweb 的基础 也是后面学框架的基础
所以必须全部看完
我这里开两倍速 算了一下时间
全部看完视频还需要 4.38 个小时
现在是凌晨 2:38
也就是早上七点多能看完
留出 coding 时间
算八点吧
学的时候建议这样估算一下 有利于流程把控

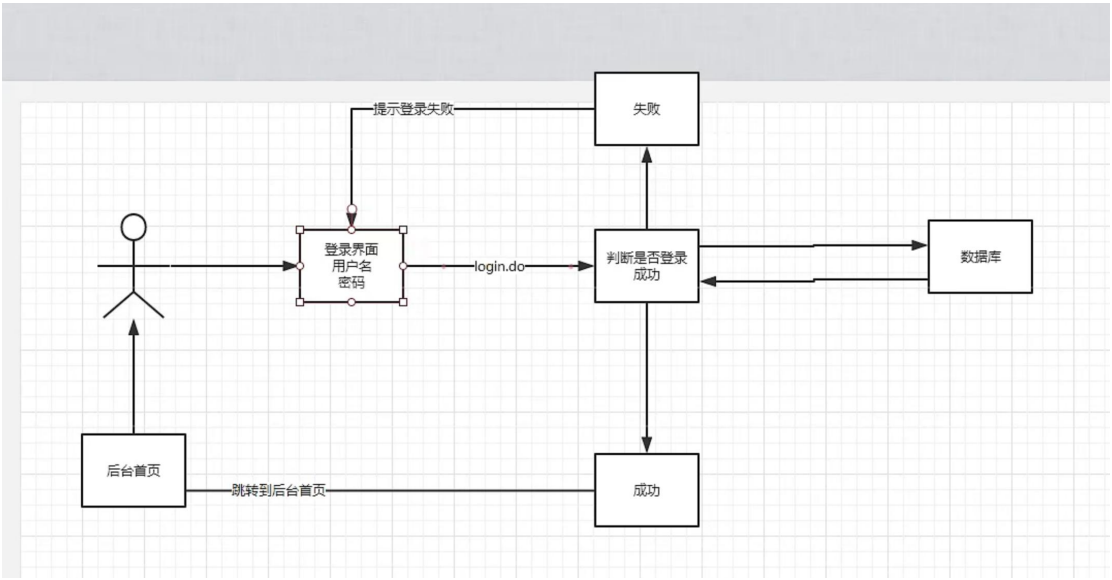
Basedao 放基础的方法
Open
Select
Update
Close

再写个过滤器
Filer 把编码问题解决掉

```java
import ...

public class CharacterEncodingFilter implements Filter {

    public void init(FilterConfig filterConfig) throws ServletException {
    }
    public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain) throws IOException, ServletException {
        request.setCharacterEncoding("utf-8");
        response.setCharacterEncoding("utf-8");
        chain.doFilter(request,response);
    }
    public void destroy() {

    }
```

再导入静态资源
直接导入即可
不赘述

登录模块



和刚刚我自己推的一样

先写 dao 层//后台查用户的数据
再写业务层//controller 层
Dao 层先写接口约束
再写 impl 实现

业务层先调 dao 层

```java
public User login(String userCode, String password) {
    Connection connection = null;
    User user = null;

    try {
        connection = BaseDao.getConnection();
        //通过业务层调用对应的具体的数据库操作
        user = userDao.getLoginUser(connection, userCode);
    } catch (SQLException e) {
        e.printStackTrace();
    }finally {
        BaseDao.closeResource(connection, preparedStatement: null, resultSet: null);
```

然后 return 一个 user

```java
        BaseDao.closeResource(connection, preparedStatement: n
    }
    //匹配密码
    if(null != user){
        if(!user.getUserPassword().equals(password))
            user = null;
    }
    return user;
}
```

整个看他写下来
先写 dao 层：实现用户接口以及查询用户的 impl 类
再写 service 层：实现查询用户的逻辑，调用 dao 层
再写 controller 层的 servlet：实现用户 password 判断，把前端传参和后端查询的 password 比对
再写 view 层的 jsp：给用户一个传参数的地方
大体是这么个逻辑

登录优化
注销&权限过滤
注销

```java
public class LogoutServlet  extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
        //移除用户的session
        req.getSession().removeAttribute(Constants.USER_SESSION);
        resp.sendRedirect( location: req.getContextPath()+"/login.jsp");//返回登录页面
    }

    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
        doGet(req, resp);
    }
}
```

就是删除服务端中的 session
还有个过滤器，判断用户是否登录，禁止直接访问后台

```
}

public void doFilter(ServletRequest req, ServletResponse resp, FilterChain chain) throws
    HttpServletRequest request = (HttpServletRequest) req;
    HttpServletResponse response = (HttpServletResponse) resp;

    //过滤器，从Session中获取用户，
    User user = (User) request.getSession().getAttribute(Constants.USER_SESSION);

    if (user==null){ //已经被移除或者注销了，或者未登录
        response.sendRedirect( location: "/smbms/error.jsp");
    }else {
        chain.doFilter(req,resp);
    }
}
```

设置 web.xml 路由

```
<filter>
    <filter-name>Sysfilter</filter-name>
    <filter-class>com.mario.filter.SysFilter</filter-class>
</filter>
<filter-mapping>
    <filter-name>Sysfilter</filter-name>
    <url-pattern>/jsp/*</url-pattern>
</filter-mapping>
```

密码修改
先写 dao

```
public int updatePwd(Connection connection, int id, String userPassword) throws SQLException {
    int updateRows=0;
    PreparedStatement pstm = null;
    if(connection!=null){
        String Sql="UPDATE `smbms_user` SET `userPassword`=? WHERE `id`=? ";
        Object []params={userPassword,id};
        updateRows=BaseDao.execute(connection,Sql,params,pstm);
    }
    BaseDao.closeResource( connection: null,pstm, resultSet: null);
    return updateRows;
```

Service 调用

```java
public boolean updatePwd(int id, String password) {
    boolean flag = false;
    Connection connection = null;
    try{
        connection = BaseDao.getConnection();
        if(userDao.updatePwd(connection,id,password) > 0)
            flag = true;
    }catch (Exception e) {
        e.printStackTrace();
    }finally{
        BaseDao.closeResource(connection, preparedStatement: null, resultSet: null);
    }
    return flag;
}
```

这里定义了一个 flag
如果修改成功
Flag 就为 true

然后作者在这里排错排了半天
哈哈哈

```java
String newpassword = req.getParameter( name: "newpassword");
boolean flag= false;
//判断是否有这个用户是否存在，以及新密码不为空
if(attribute!=null && !StringUtils.isNullOrEmpty(newpassword)){
    UserService userService = new UserServiceImpl();
    flag = userService.updatePwd(((User) attribute).getId(), newpassword);
    if (flag) {
        req.setAttribute(Constants.SYS_MESSAGE, o: "修改密码成功，请退出后重新登录");
        //密码修改成功后移除当前session
        req.getSession().removeAttribute(Constants.USER_SESSION);
    }else{
        req.setAttribute(Constants.SYS_MESSAGE, o: "密码修改失败请重新输入");
    }
}else{
    req.setAttribute(Constants.SYS_MESSAGE, o: "新密码设置错误请重新输入");
```

小伙子这里少了一个!
真的服气
哈哈哈
找了半小时
然后根据 flag 判断是否修改成功

优化密码注册
判断旧密码

```
$.ajax({
    type:"GET",
    url:path+"/jsp/user.do",
    data:{method:"pwdmodify",oldpassword:oldpassword.val()},//data就是ajax传递的参数
    /*
    上面这句话等价于path+"/jsp/user.do?method="pwdmodify"&&oldpassword=oldpassword.val()
    */
    dataType:"json",//主流开发都是用JSON实现前后端开发{}
    success:function(data){
        if(data.result == "true"){//旧密码正确
            validateTip(oldpassword.next(), css: {"color":"green"},imgYes, status: true);
        }else if(data.result == "false"){//旧密码输入不正确
            validateTip(oldpassword.next(), css: {"color":"red"}, tipString: imgNo + " 原密码输入不正确", status: false);
        }else if(data.result == "sessionerror"){//当前用户session过期，请重新登录
            validateTip(oldpassword.next(), css: {"color":"red"}, tipString: imgNo + " 当前用户session过期，请重新登录", status: false);
        }else if(data.result == "error"){//旧密码输入为空
            validateTip(oldpassword.next(), css: {"color":"red"}, tipString: imgNo + " 请输入旧密码", status: false);
        }
    },
```

这里作者用了一套 ajax 来传参实现验证老密码
这个 json 传参有点意思
果不其然 pom 文件里就有好东西

```
<!--     阿里fastjson依赖-->
<!-- https://mvnrepository.com/artifact/com.alibaba/fastjson -->
<dependency>
  <groupId>com.alibaba</groupId>
  <artifactId>fastjson</artifactId>
  <version>1.2.62</version>
</dependency>
</dependencies>
```

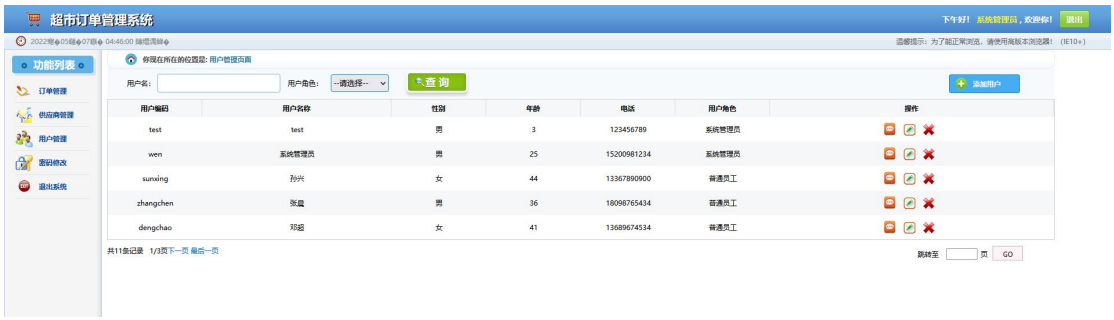
哈哈哈
他这里校验完毕

```
private void modifyPwd(HttpServletRequest req, HttpServletResponse resp){
    //从session中获得用户的旧密码,这里的attribute包括了用户的所用信息
    Object attribute = req.getSession().getAttribute(Constants.USER_SESSION);
    //从前端输入的页面中获得输入的旧密码
    String oldpassword = req.getParameter( name: "oldpassword");
    //万能的Map
    Map<String, String> resultMap = new HashMap<~>();
    if (attribute==null){//取到的session为空，意味着session过期了
        resultMap.put("result","sessionerror");
    }else if (StringUtils.isNullOrEmpty(oldpassword)){//如果输入的旧密码为空
        resultMap.put("result","sessionerror");
    }else{//session不为空，输入的旧密码也不为空，则取出当前旧密码与之比较
        String userPassword = ((User) attribute).getUserPassword();
        if(oldpassword.equals(userPassword)){
            resultMap.put("result","true");
        }else {
            resultMap.put("result","false");
        }
```

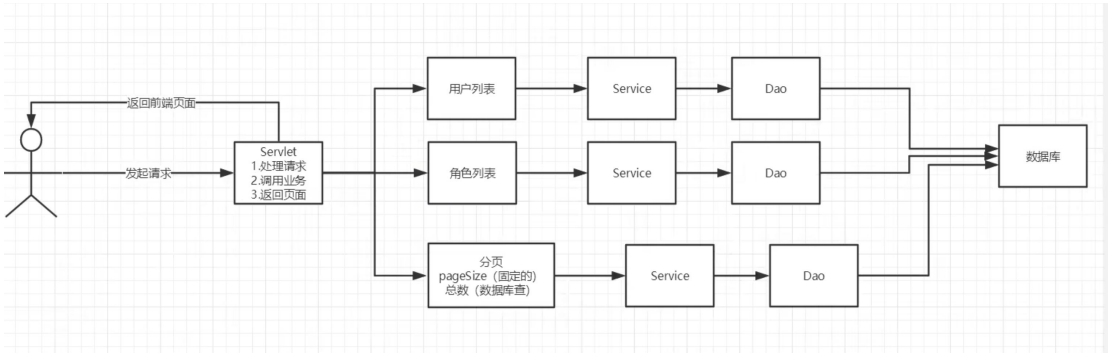
然后返回状态码给前端，前端再根据状态码显示页面
最后进行 json 转换，用 json 格式 write 过去

```
        }
        try {
            resp.setContentType("application/json");
            PrintWriter out = resp.getWriter();
            out.write(JSONArray.toJSONString(resultMap));
            out.flush();
            out.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
```

用户管理底层实现



架构



先导入分页工具类
然后导入用户前端 view 的 jsp
然后再来写 dao 层

```
public int getUserCount(Connection connection, String userName, int userRole) throws Exception {
    int count=0;
    PreparedStatement pstm = null;
    ResultSet rs=null;
    if (connection!=null) {
        StringBuffer sql=new StringBuffer();
        sql.append("SELECT COUNT(*) AS count FROM `smbms_user` u,`smbms_role` r WHERE u.`userRole`=r.`id`");
        ArrayList<Object> list = new ArrayList<>();//存放可能会放进sql里的参数，就是用来替代?的params
        if(!StringUtils.isNullOrEmpty(userName)){
            sql.append(" and u.username like ?");
            list.add("%"+userName+"%");//模糊查询，index:0
        }
        if(userRole>0){
            sql.append(" and u.userRole = ?");
            list.add(userRole);//index:1
```

他这个 dao 层 就有 sql 注入
Id 这里传参过去即可
后面这里跟了两段 sql

```
ArrayList<Object> list = new ArrayList<>();//存放可能
if(!StringUtils.isNullOrEmpty(userName)){
    sql.append(" and u.username like ?");
    list.add("%"+userName+"%");//模糊查询，index:0
}
if(userRole>0){
    sql.append(" and u.userRole = ?");
    list.add(userRole);//index:1
}
```

Append 过去的
最后执行

```
Object[] params = list.toArray();//转换成数组
System.out.println("当前的sql语句为---------->"+sql);
rs = BaseDao.execute(connection, sql.toString(), params, rs, pstm);
if(rs.next()){
    count=rs.getInt( columnLabel: "count");

}
BaseDao.closeResource( connection: null,pstm,rs);
```

用户角色列表的建立

```
public class RoleDaoImpl implements RoleDao {
    public List<Role> getRoleList(Connection connection) throws Exception {
        PreparedStatement pstm=null;
        ResultSet rs=null;
        List<Role> roleList = new ArrayList<~>();
        if(connection!=null){
            String sql="SELECT * FROM `smbms_role`";
            Object[] params={};
            rs = BaseDao.execute(connection, sql, params, rs, pstm);
            while(rs.next()){
                Role role = new Role();
                role.setId(rs.getInt( columnLabel: "id"));
                role.setRoleCode(rs.getString( columnLabel: "roleCode"));
                role.setRoleName(rs.getString( columnLabel: "roleName"));
                roleList.add(role);
            }
        }
        BaseDao.closeResource( connection: null,pstm,rs);
        return roleList;
        //细心细心细心
```

这里还要单独写一个 pojo 实体类，方便操作

```java
import java.util.Date;
public class Role {
    private Integer id;   //id
    private String roleCode; //角色编码
    private String roleName; //角色名称
    private Integer createdBy; //创建者
    private Date creationDate; //创建时间
    private Integer modifyBy; //更新者
    private Date modifyDate;//更新时间

    public Integer getId() { return id; }
    public void setId(Integer id) { this.id = id; }
    public String getRoleCode() { return roleCode; }
    public void setRoleCode(String roleCode) { this.roleCode = roleCode; }
    public String getRoleName() { return roleName; }
    public void setRoleName(String roleName) { this.roleName = roleName; }
    public Integer getCreatedBy() { return createdBy; }
    public void setCreatedBy(Integer createdBy) { this.createdBy = createdBy; }
    public Date getCreationDate() { return creationDate; }
    public void setCreationDate(Date creationDate) { this.creationDate = creationDate; }
    public Integer getModifyBy() { return modifyBy; }
    public void setModifyBy(Integer modifyBy) { this.modifyBy = modifyBy; }
    public Date getModifyDate() { return modifyDate; }
    public void setModifyDate(Date modifyDate) { this.modifyDate = modifyDate; }
```

前端基本的增删改查和后端对应

增

```java
//增加用户
private void add(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
    System.out.println("当前正在执行增加用户操作");
    //从前端得到页面的请求的参数即用户输入的值
    String userCode = req.getParameter( name: "userCode");
    String userName = req.getParameter( name: "userName");
    String userPassword = req.getParameter( name: "userPassword");
    //String ruserPassword = req.getParameter("ruserPassword");
    String gender = req.getParameter( name: "gender");
    String birthday = req.getParameter( name: "birthday");
    String phone = req.getParameter( name: "phone");
    String address = req.getParameter( name: "address");
    String userRole = req.getParameter( name: "userRole");
    //把这些值塞进一个用户属性中
```

删

```java
//删除用户，需要当前的Id，来找到这个用户然后删除
private void delUser(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException{
    String id = req.getParameter( name: "uid");
    Integer delId = 0;
    try{
        delId = Integer.parseInt(id);
    }catch (Exception e) {
        // TODO: handle exception
        delId = 0;
    }
    //需要判断是否能删除成功
    HashMap<String, String> resultMap = new HashMap<~>();
    if(delId <= 0){
        resultMap.put("delResult", "notexist");
    }else{
        UserService userService = new UserServiceImpl();
        if(userService.deleteUserById(delId)){
```

改

```java
//修改用户信息
private void modify(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException{
    //需要拿到前端传递进来的参数
    String id = req.getParameter( name: "uid");;
    String userName = req.getParameter( name: "userName");
    String gender = req.getParameter( name: "gender");
    String birthday = req.getParameter( name: "birthday");
    String phone = req.getParameter( name: "phone");
    String address = req.getParameter( name: "address");
    String userRole = req.getParameter( name: "userRole");

    //创建一个user对象接收这些参数
    User user = new User();
    user.setId(Integer.valueOf(id));
    user.setUserName(userName);
    user.setGender(Integer.valueOf(gender));
    try {
```

查

```java
    }

    //通过id得到用户信息
    private void getUserById(HttpServletRequest req, HttpServletResponse resp,String url) throws ServletException, IOExceptio

        String id = req.getParameter( name: "uid");
        if(!StringUtils.isNullOrEmpty(id)){
            //调用后台方法得到user对象
            UserService userService = new UserServiceImpl();
            User user = userService.getUserById(id);
            req.setAttribute( name: "user", user);
            req.getRequestDispatcher(url).forward(req, resp);
        }
    }
```

基本都是一套逻辑

前端传参

```java
protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
    String method = request.getParameter( name: "method");
    System.out.println("method----> " + method);
    if(method != null && method.equals("add")){
        //增加操作
        this.add(request, response);
    }else if(method != null && method.equals("query")){
        //查询用户操作
        this.query(request, response);
    }else if(method != null && method.equals("getrolelist")){
        //查询用户角色表
        this.getRoleList(request, response);
    }else if(method != null && method.equals("ucexist")){
        //查询当前用户编码是否存在
        this.userCodeExist(request, response);
    }else if(method != null && method.equals("deluser")){
        //删除用户
        this.delUser(request, response);
    }else if(method != null && method.equals("view")){
```

这里根据前端传参判断具体要执行什么操作
然后调用 this.xxx 方法
然后往 service 层走

```java
        user.setUserRole(Integer.valueOf(userRole));
        user.setCreationDate(new Date());
        //查找当前正在登陆的用户的id
        user.setCreatedBy(((User)req.getSession().getAttribute(Constants.USER_SESSION)).getId());
        UserServiceImpl userService = new UserServiceImpl();
        Boolean flag = userService.add(user);
        //如果添加成功，则页面转发，否则重新刷新，再次跳转到当前页面
        if(flag){
            resp.sendRedirect( location: req.getContextPath()+"/jsp/user.do?method=query");
        }else{
            req.getRequestDispatcher( path: "useradd.jsp").forward(req,resp);
        }
    }
```

```java
public Boolean add(User user) {
    boolean flag = false;
    Connection connection = null;
    try {
        connection = BaseDao.getConnection();//获得连接
        connection.setAutoCommit(false);//开启JDBC事务管理
        int updateRows = userDao.add(connection,user);
        connection.commit();
        if(updateRows > 0){
            flag = true;
            System.out.println("add success!");
        }else{
            System.out.println("add failed!");
        }

    } catch (Exception e) {
        // TODO Auto-generated catch block
```

走到 service 层之，很明显看到，调用了 dao 层

```
//增加用户信息
public  int add(Connection connection,User user) throws Exception;

//通过用户id删除用户信息
public int deleteUserById(Connection connection, Integer delId)throws Exception

//通过userId查看当前用户信息
public User getUserById(Connection connection, String id)throws Exception;
//修改用户信息
public int modify(Connection connection, User user)throws Exception;
```

然后然后到 impl 实现方法

```
public int add(Connection connection, User user) throws Exception {
    PreparedStatement pstm=null;
    int updateNum=0;
    if(connection!=null) {
        String sql = "insert into smbms_user (userCode,userName,userPassword," +
                "userRole,gender,birthday,phone,address,creationDate,createdBy) " +
                "values(?,?,?,?,?,?,?,?,?)";
        Object[] params = {user.getUserCode(), user.getUserName(), user.getUserPassword(),
                user.getUserRole(), user.getGender(), user.getBirthday(),
                user.getPhone(), user.getAddress(), user.getCreationDate(), user.getCreatedBy()};
        updateNum = BaseDao.execute(connection, sql, params, pstm);
        BaseDao.closeResource( connection: null, pstm,  resultSet: null);
    }
    return updateNum;
}
```

基本就是这套东西一直复用

一个路子

文件传输原理

基本 IO 操作

Input

Output

然后就是基本的上传

很简单

不赘述了

---20220507 7：45---

Javaweb 基础这块都过完一遍了

中途还打了三把游戏 出去吃了个早饭

因为有些东西本来就会 所以就跳过了 节约了一些时间

先睡一觉 醒了之后会继续跟上框架的学习过程

---20220507 17:45---

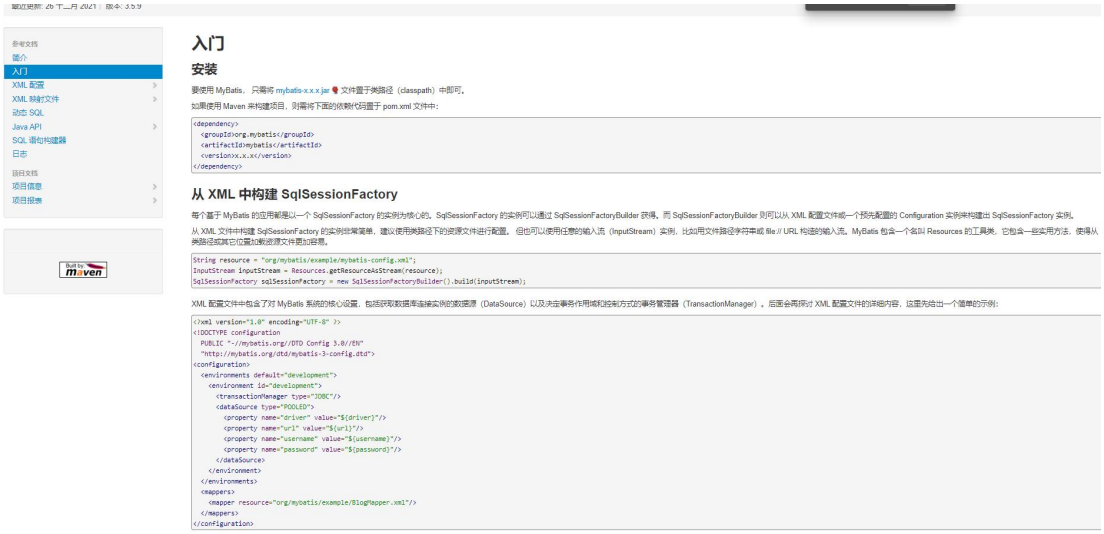
下午两点半醒来的

处理了一点项目上的事情

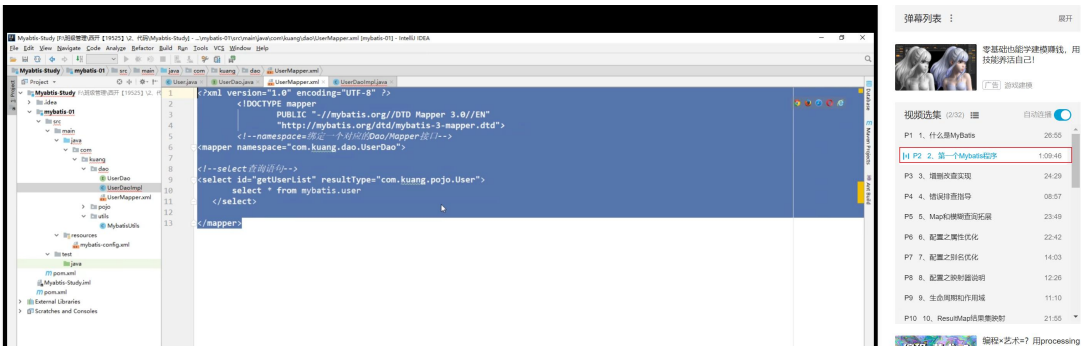
就五点了

看到 b 站推荐给我了这个视频
//java 学习路线
bilibili.com/video/BV15g41157NK/?spm_id_from=333.788.recommend_more_video.0
这个是开发路线
还是按照基础的
语法-->算法-->框架-->微服务这块来学
我们搞安全的 侧重在漏洞上 因此开发部分可以简化 主要是框架+设计模式+源码阅读上
因为要看很多框架 所以这些东西要了解

Mybatis
一个数据库框架
先过一遍官方文档
https://mybatis.org/mybatis-3/zh/index.html



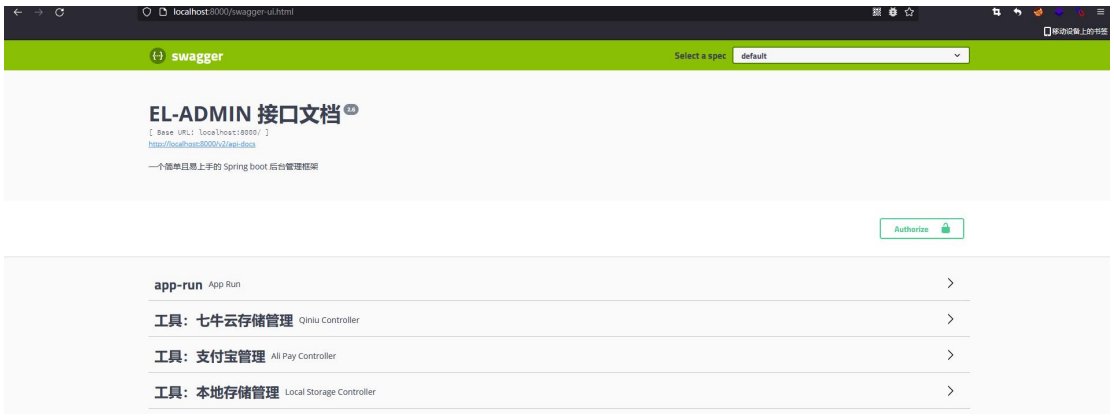
主要是先看看架构
怎么配置之类的
狂神的一个视频一个小时
太久了



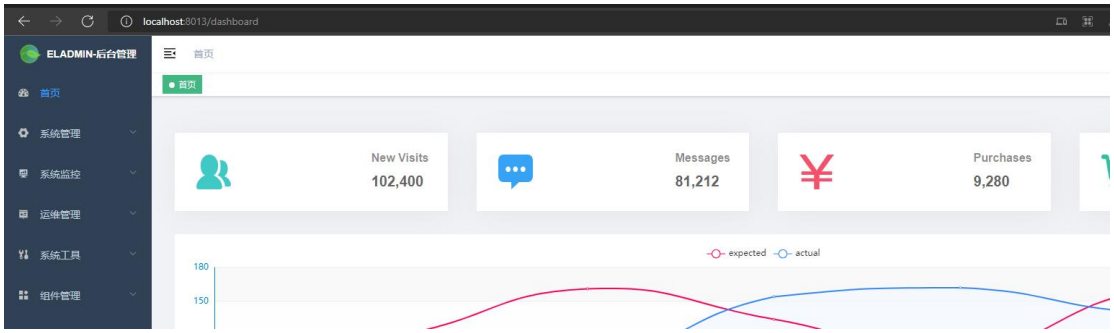
于是找到他的微信公众号狂神说 直接看文章

---20220508 2:22---
不想整了 休息一下
我发现写代码这个东西吧
有时候是真的不想手敲 就想看着 然后复制粘贴
---20220508 4:44---
划水到现在
看到了狂神一个视频
分析开源项目
https://www.bilibili.com/video/BV1T7411L74W?spm_id_from=333.999.0.0
决定研究一下
然后半途而废。。。
因为又去打游戏了。。。
---20220508 14:38---
起床了，直接跟上

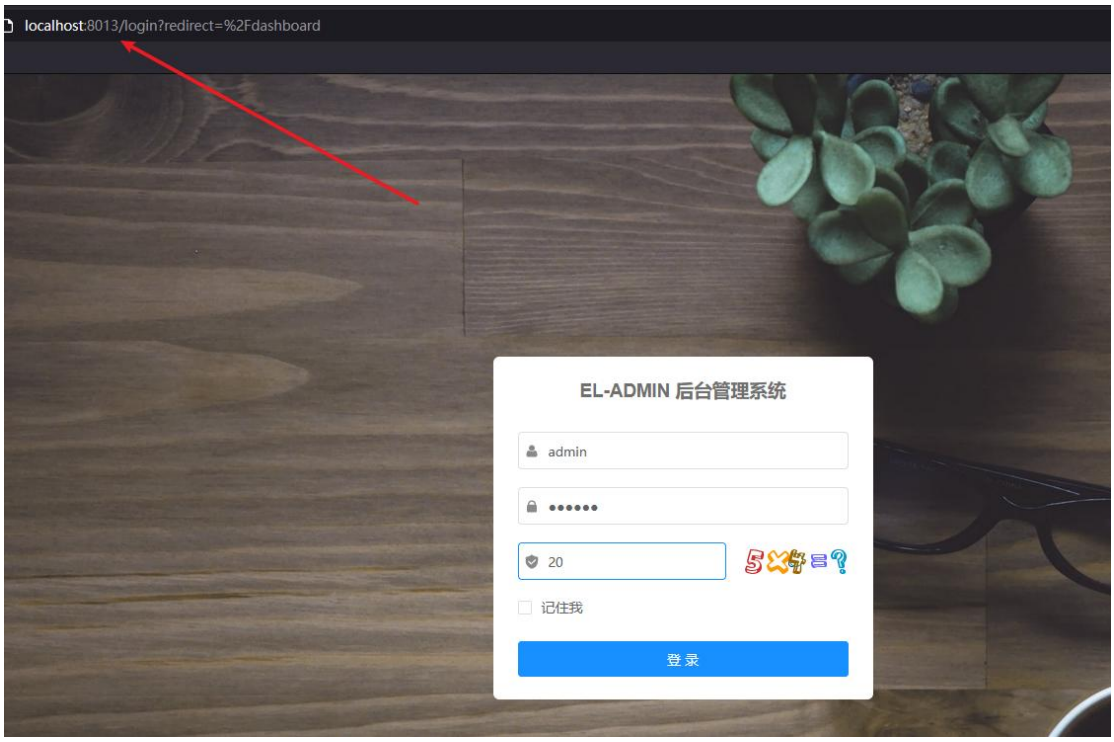
思路
先看项目架构
再看 swagger
然后再查看配置文件
swagger 路径是固定的
/swagger-ui.html



项目是前后端分离的
前端还需要增加 node.js 的环境
然后跑起来



这里存在一个问题
前后端端口调用不一致
这里前端是 8013



后端是 8000

证明前端封装了请求

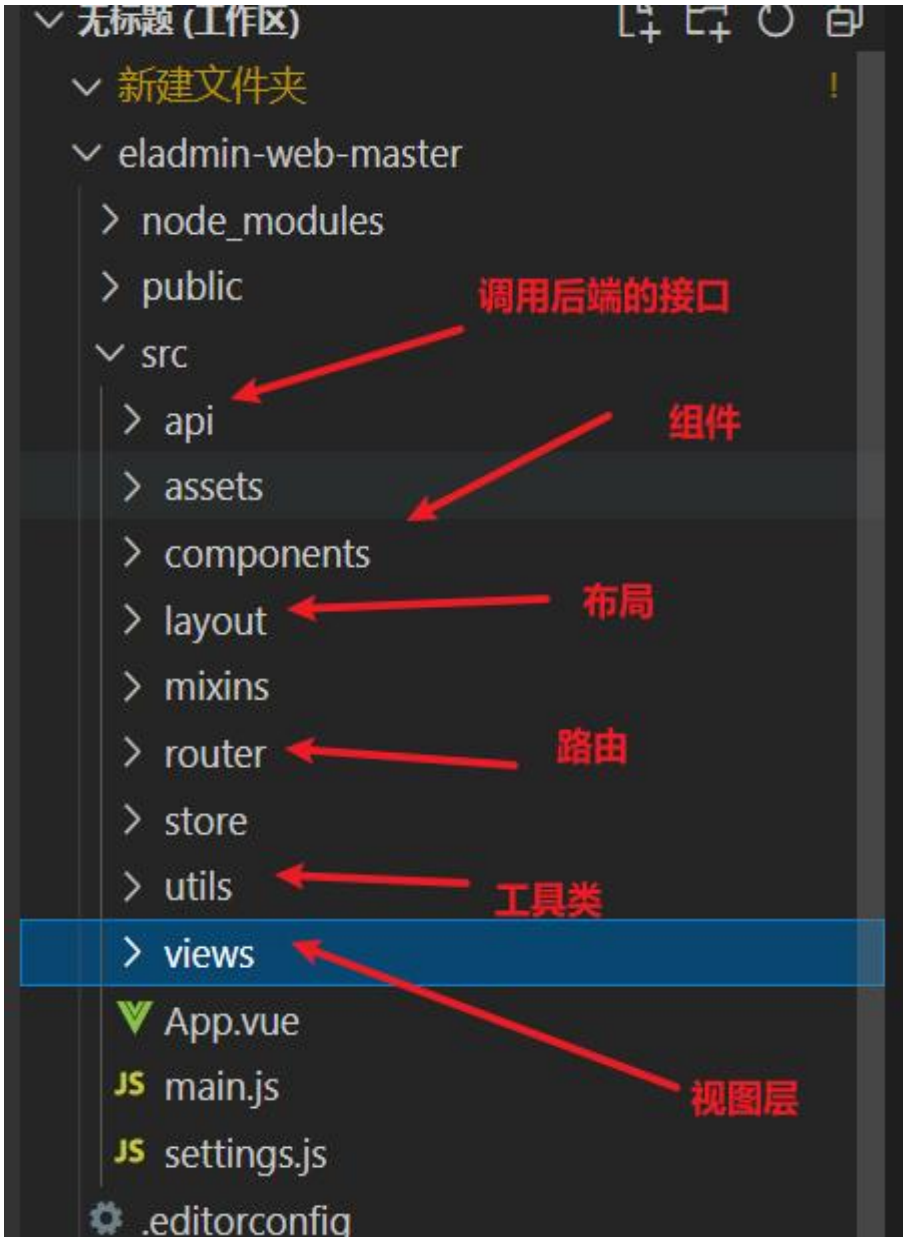
然后找到前端对应后端的地址

```
ENV = 'development'

# 接口地址
VUE_APP_BASE_API  = 'http://localhost:8000'
VUE_APP_WS_API = 'ws://localhost:8000'

# 是否启用 babel-plugin-dynamic-import-node插件
VUE_CLI_BABEL_TRANSPILE_MODULES = true
```

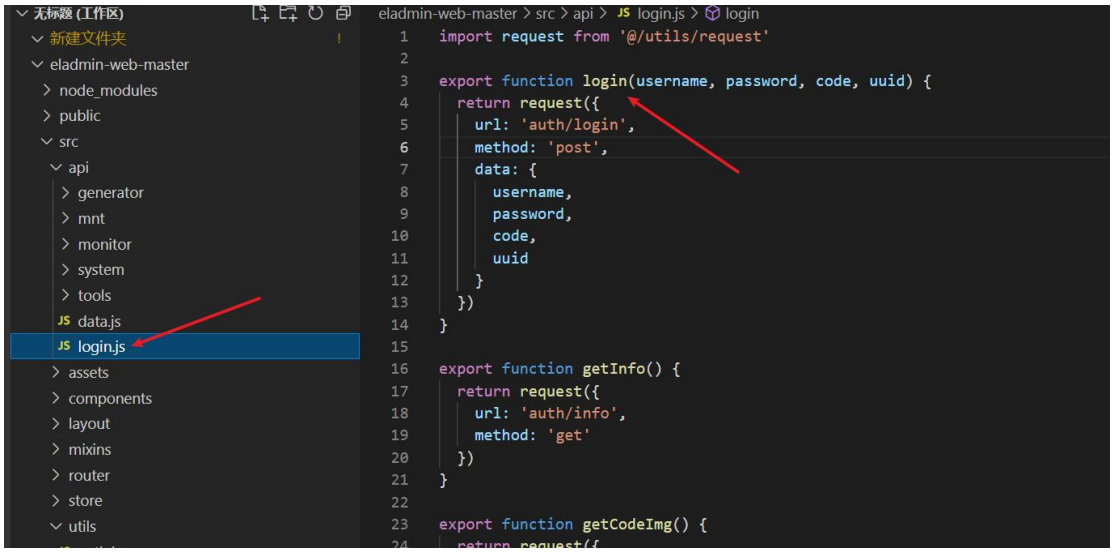
固定套路

Vue 渲染页面

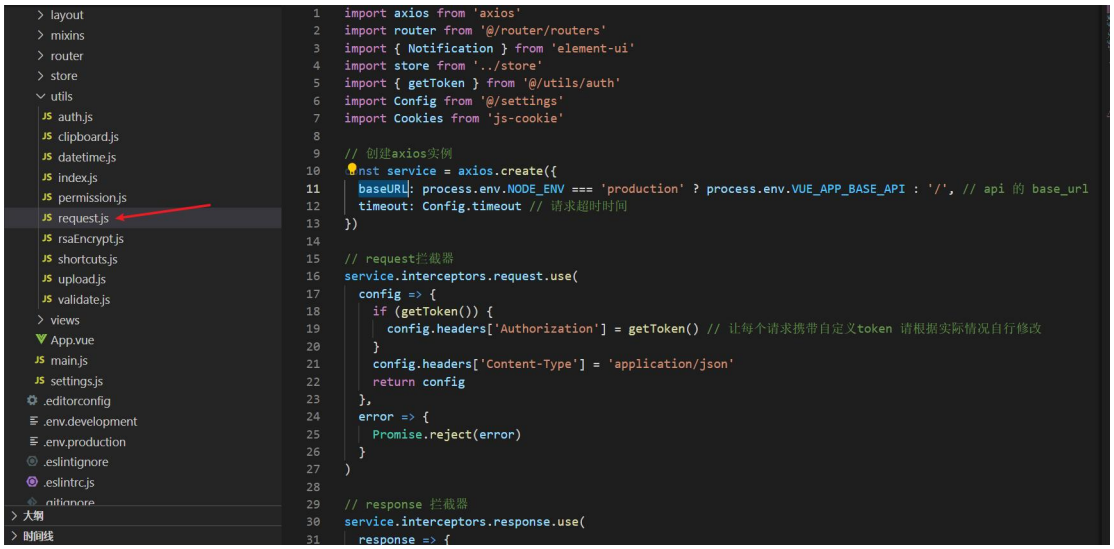
后端 springboot 提供服务



查看登录接口

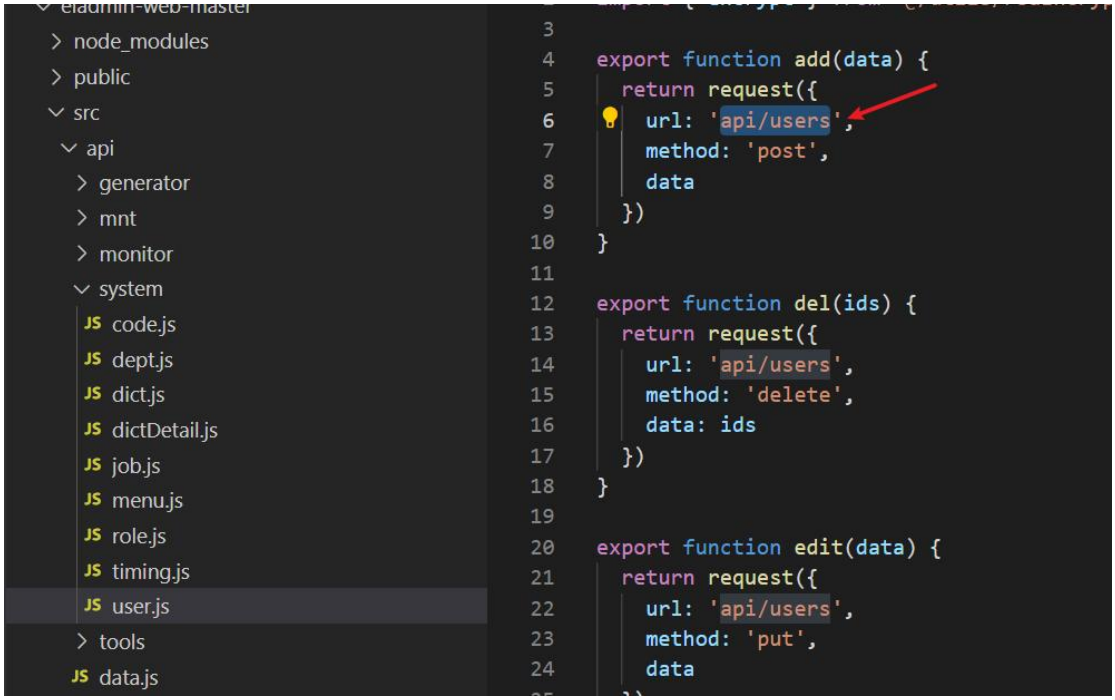


Vue 这个代码我也是第一次看
看起来也是封装的
然后跟一下上面这个 util/request

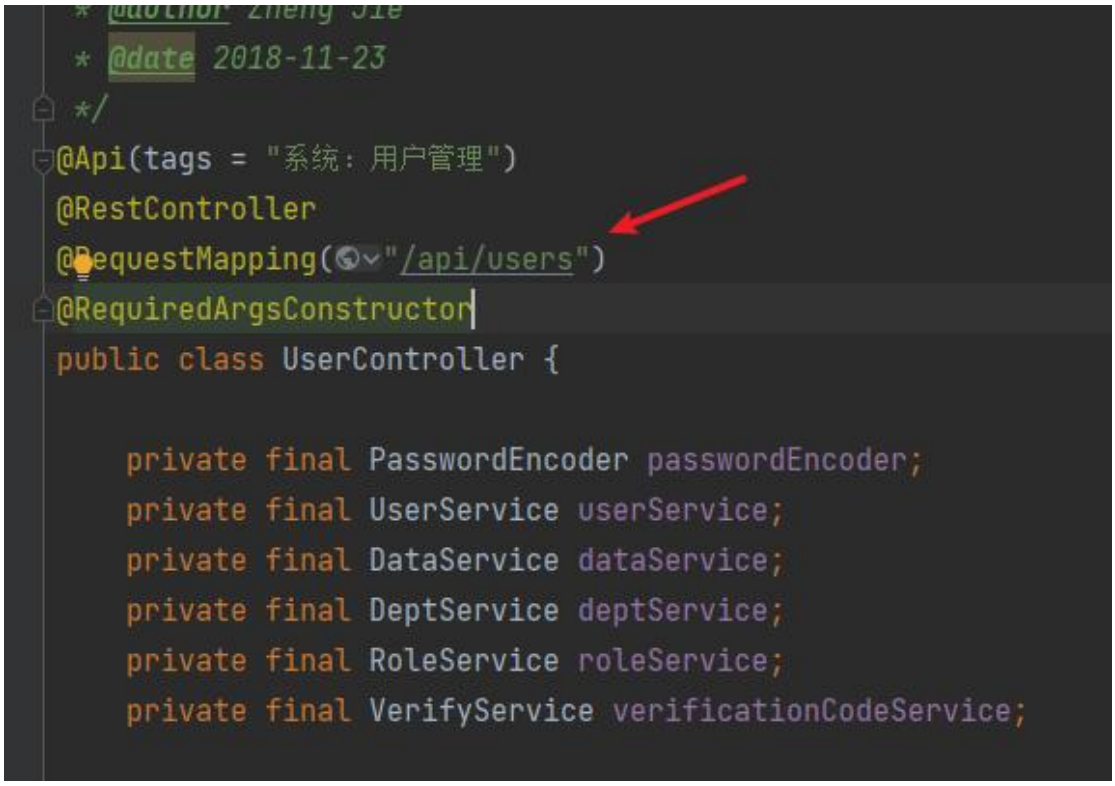


实现了前端的 request
类似于我们自己写一个 form 给提交
只是这里分开了
逻辑更加独立

找到前端的接口



后端可以找到接口的具体实现



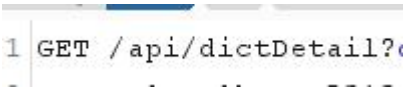
这里就是个 springboot 的 api

可以从前端来检查

Burp 抓包

```
Pretty Raw \n Actions
1 GET /api/dictDetail?dictName=job_status&page=0&size=9999 HTTP/1.1
2 Host: localhost:8013
3 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:100.0) Gecko/20100101 Firefox/100.0
4 Accept: application/json, text/plain, */*
5 Accept-Language: zh-CN,zh;q=0.8,zh-TW;q=0.7,zh-HK;q=0.5,en-US;q=0.3,en;q=0.2
6 Accept-Encoding: gzip, deflate
7 Authorization: Bearer
  eyJhbGciOiJIUzUxMiJ9.eyJqdGkiOiI0YWNmMjcxYzRjZTE0YmMxYjYyNWM4NmNmYTYyNjBlNyIsInVzZXIiOiJhZGlpbiIsInNlYiI6ImFkbWluIn0.ESoooDKs4lLliBr36PJ89_1x6dZrTw_1G6jZ
  M0YKHW2rZzHlYQH4K8IQ5F_8BlQV_DG55IBifJtgL-ztHLk3kw
8 Connection: close
9 Referer: http://localhost:8013/system/job
10 Cookie: Idea-7f7ed34b=37b0a42b-b86c-4f53-88b6-097493e07446; EL-ADMIN-TOEKN=
  Bearer%20eyJhbGciOiJIUzUxMiJ9.eyJqdGkiOiI0YWNmMjcxYzRjZTE0YmMxYjYyNWM4NmNmYTYyNjBlNyIsInVzZXIiOiJhZGlpbiIsInNlYiI6ImFkbWluIn0.ESoooDKs4lLliBr36PJ89_1x6dZ
  rTw_1G6jZM0YKHW2rZzHlYQH4K8IQ5F_8BlQV_DG55IBifJtgL-ztHLk3kw
11 Sec-Fetch-Dest: empty
12 Sec-Fetch-Mode: cors
13 Sec-Fetch-Site: same-origin
14
15
```

然后找到对应接口

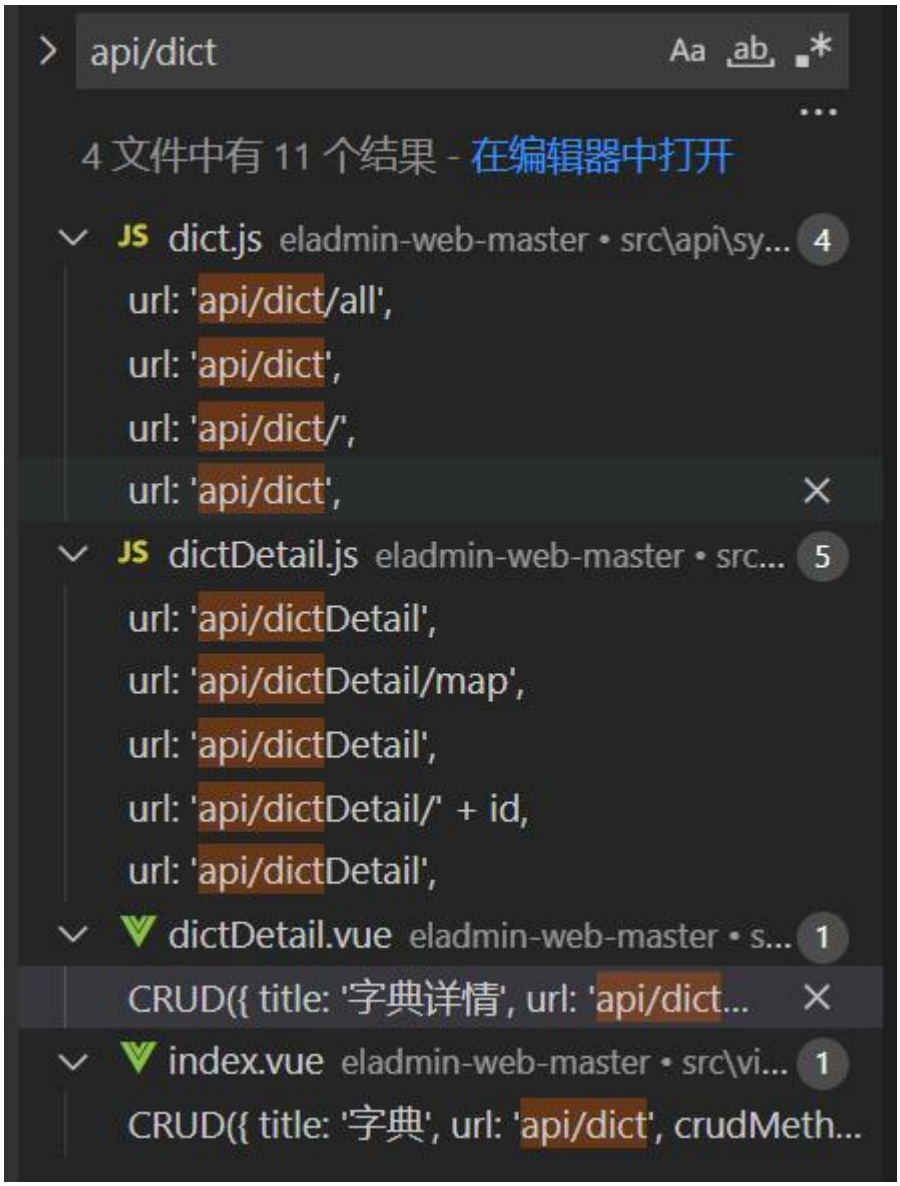


```
1 GET /api/dictDetail?
```

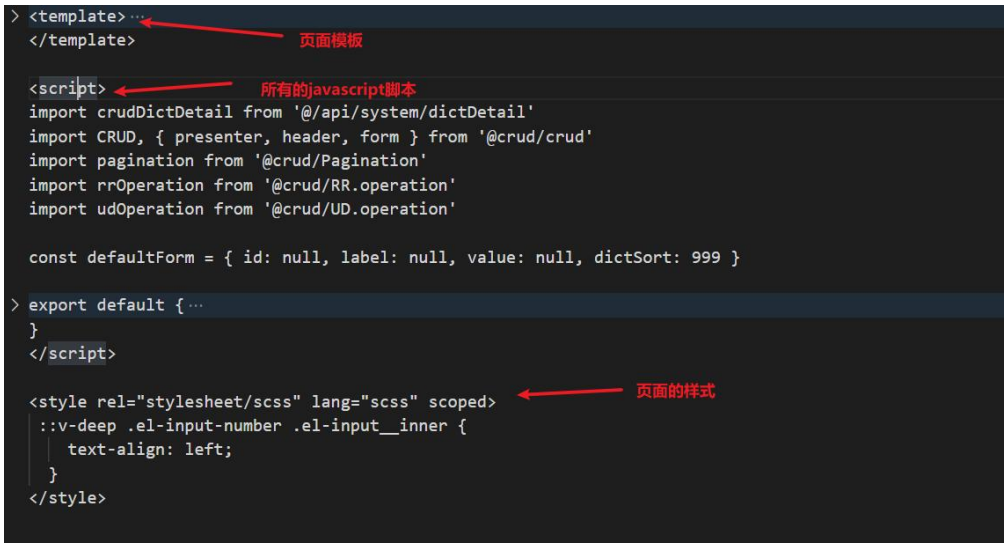
然后去后端搜索

```
@RequestMapping("/api/dictDetail")
public class DictDetailController {

    private final DictDetailService dictDetailService;
    private static final String ENTITY_NAME = "dictDetail";

    @ApiOperation("查询字典详情")
    @GetMapping
    public ResponseEntity<Object> queryDictDetail(DictDetailQueryCriteria criteria,
                                                  @PageableDefault(sort = {"dictSort"}, direction = Sort.Direction.ASC) Pageable pageable){
        return new ResponseEntity<>(dictDetailService.queryAll(criteria,pageable),HttpStatus.OK);
    }

    @ApiOperation("查询多个字典详情")
```

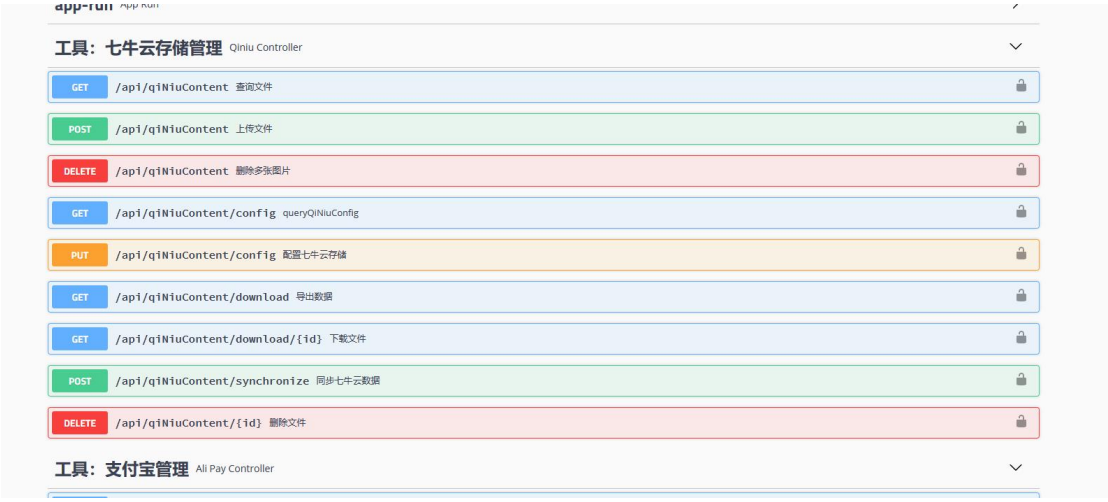
找到后端的接口地址
然后看看前端的接口地址
搜索找到


api/dict
Aa ab .*
4 文件中有 11 个结果 - 在编辑器中打开
dict.js eladmin-web-master • src\api\sy... 4
url: 'api/dict/all',
url: 'api/dict',
url: 'api/dict/',
url: 'api/dict',
dictDetail.js eladmin-web-master • src... 5
url: 'api/dictDetail',
url: 'api/dictDetail/map',
url: 'api/dictDetail',
url: 'api/dictDetail/' + id,
url: 'api/dictDetail',
dictDetail.vue eladmin-web-master • s... 1
CRUD({ title: '字典详情', url: 'api/dict...
index.vue eladmin-web-master • src\vi... 1
CRUD({ title: '字典', url: 'api/dict', crudMeth...

找到对应的 vue


<template>
</template>
页面模板
<script>
所有的javascript脚本
import crudDictDetail from '@/api/system/dictDetail'
import CRUD, { presenter, header, form } from '@crud/crud'
import pagination from '@crud/Pagination'
import rrOperation from '@crud/RR.operation'
import udOperation from '@crud/UD.operation'
const defaultForm = { id: null, label: null, value: null, dictSort: 999 }
export default {
}
</script>
<style rel="stylesheet/scss" lang="scss" scoped>
页面的样式
::v-deep .el-input-number .el-input__inner {
text-align: left;
}
</style>

再看看 swagger



这些东西真的很有用

前期渗透的时候直接找到了 swagger-ui 文档，那么接口信息就都拿到了

直接可以渗透

继续回到 mybatis

项目都写好之后

报错

排错

接口没问题

Mybatis 的 xml 配置文件没问题

```xml
<?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="dao.UserDao">
    <select id="getUserList" resultType="pojo.User">
        select * from mybatis.user
    </select>
</mapper>
```

实现的 util 工具类没问题

```java
//sqlSessionFactory -- > sqlSession
public class MybatisUtils {

    private static SqlSessionFactory sqlSessionFactory;

    static{

        try {
            String resource = "mybatis-config.xml";
            InputStream inputStream = Resources.getResourceAsStream(resource);
            SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public static SqlSession getSqlSession(){
        return sqlSessionFactory.openSession();
    }
```

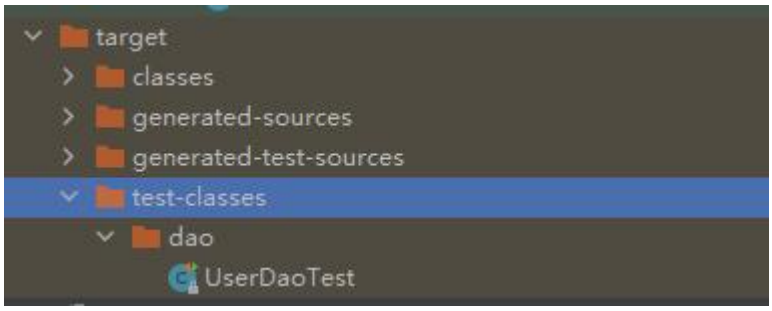
测试类也没问题

```java
import org.apache.ibatis.session.SqlSession;
import org.junit.Test;
import pojo.User;
import utils.MybatisUtils;

import java.util.List;

public class UserDaoTest {
    @Test
    public void test(){

        //1
        SqlSession sqlSession = MybatisUtils.getSqlSession();


        UserDao userDao = sqlSession.getMapper(UserDao.class);
        List<User> userList = userDao.getUserList();

        for(User user : userList){
            System.out.println(user);
        }

        sqlSession.close();
```

继续看报错

```
### Error building SqlSession.
### The error may exist in dao/UserMapper.xml
```

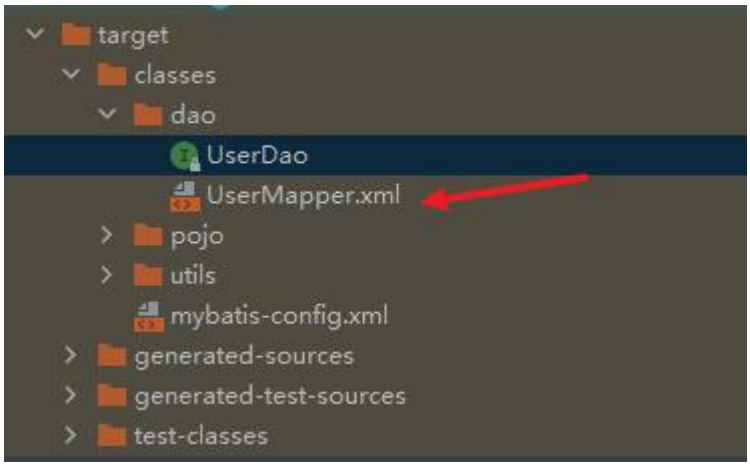
显示配置文件不在
看看 class 文件



确实，生成的 target 中的 class 中没有资源文件
怎么办呢
这里需要在 maven 中配置资源文件 build
然后配置了一下

```xml
<build>
    <resources>
        <resource>
            <directory>src/main/resources</directory>
            <includes>
                <include>**/*.properties</include>
                <include>**/*.xml</include>
            </includes>
            <filtering>true</filtering>
        </resource>

        <resource>
            <directory>src/main/java</directory>
            <includes>
                <include>**/*.properties</include>
                <include>**/*.xml</include>
            </includes>
        </resource>
    </resources>
</build>
```

这下有了



然后尝试运行

其实发现还是报错

后来我换了一套 jdbc 的 url

```
<property name="url" value="jdbc:mysql://127.0.0.1:3306/mybatis?useUnicode=true&amp;characterEncoding=utf-8&amp;useSSL=tru
```

然后可以了

```
"C:\Program Files\Java\jdk1.8.0_181\bin\java.exe" ...
Loading class `com.mysql.jdbc.Driver'. This is deprecated. The new driver class is `com.mysql.cj.jdbc.Driver'. The drive
User{id=1, name='狂神', pwd='123456'}
User{id=2, name='张三', pwd='abcdef'}
User{id=3, name='李四', pwd='987654'}
```

能查出来，但是报红

这里我换个 driver 看看

```
"C:\Program Files\Java\jdk1.8.0_181\bin\java.exe" ...
User{id=1, name='狂神', pwd='123456'}
User{id=2, name='张三', pwd='abcdef'}
User{id=3, name='李四', pwd='987654'}
```

全绿了

其实前面还有个空指针的错

问题出在这里

```
public class MybatisUtils {

    private static SqlSessionFactory sqlSessionFactory;

    static{

        try {
            String resource = "mybatis-config.xml";
            InputStream inputStream = Resources.getResourceAsStream(resource);
            sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public static SqlSession getSqlSession(){
        return sqlSessionFactory.openSession();
    }
```

静态变量 已经声明过了
如果二次声明，就会报 nullpointer 的错

```java
//sqlSessionFactory -- > sqlSession
public class MybatisUtils {

    private static SqlSessionFactory sqlSessionFactory;

    static{

        try {
            String resource = "mybatis-config.xml";
            InputStream inputStream = Resources.getResourceAsStream(resource);
            SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public static SqlSession getSqlSession(){
        return sqlSessionFactory.openSession();
```

这里代码没有变红
因为语法检查是没问题的
但是运行的时候
会有问题

```
"C:\Program Files\Java\jdk1.8.0_181\bin\java.exe" ...

java.lang.NullPointerException Create breakpoint
    at utils.MybatisUtils.getSqlSession(MybatisUtils.java:27)
    at dao.UserDaoTest.test(UserDaoTest.java:15) <22 internal lines>
```

删掉第二次声明即可

CRUD
这里再理一下这几个文件的逻辑
其实就是 javaweb 写 dao 层的那一套

只是在原有的 dao 层的基础上做了一层封装
封装如下

```xml
<?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="dao.UserMapper">
    <select id="getUserList" resultType="pojo.User">
        select * from mybatis.user
    </select>

<!--    <select id="getUserById" resultType="pojo.User" parameterType="int">-->
<!--        select * from mybatis.user where id = #{}-->
<!--    </select>-->
    <select id="getUserById" parameterType="int" resultType="pojo.User">
        select * from mybatis.user where id = #{id}
    </select>

</mapper>
```

所有的 sql 语句用规定的格式封装好了
然后在使用的时候，直接调用

```
Test
ublic void getUserById(){
    SqlSession sqlSession = MybatisUtils.getSqlSession();
    UserMapper mapper = sqlSession.getMapper(UserMapper.class);

    User user = mapper.getUserById(1);
    System.out.println(user);

    sqlSession.close();
```

实例化mybatis的sqlsession
加载sql语句实现文件
调用sql语句 传参

这个

```
SqlSession sqlSession = MybatisUtils.getSqlSession();
```

实际上就是调用公共工具类中的

```
static{
    try {
        String resource = "mybatis-config.xml";
        InputStream inputStream = Resources.getResourceAsStream(resource);
        sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);
    } catch (IOException e) {
        e.printStackTrace();
    }
}

public static SqlSession getSqlSession(){
    return sqlSessionFactory.openSession();
}

}
```

静态方法 自动运行

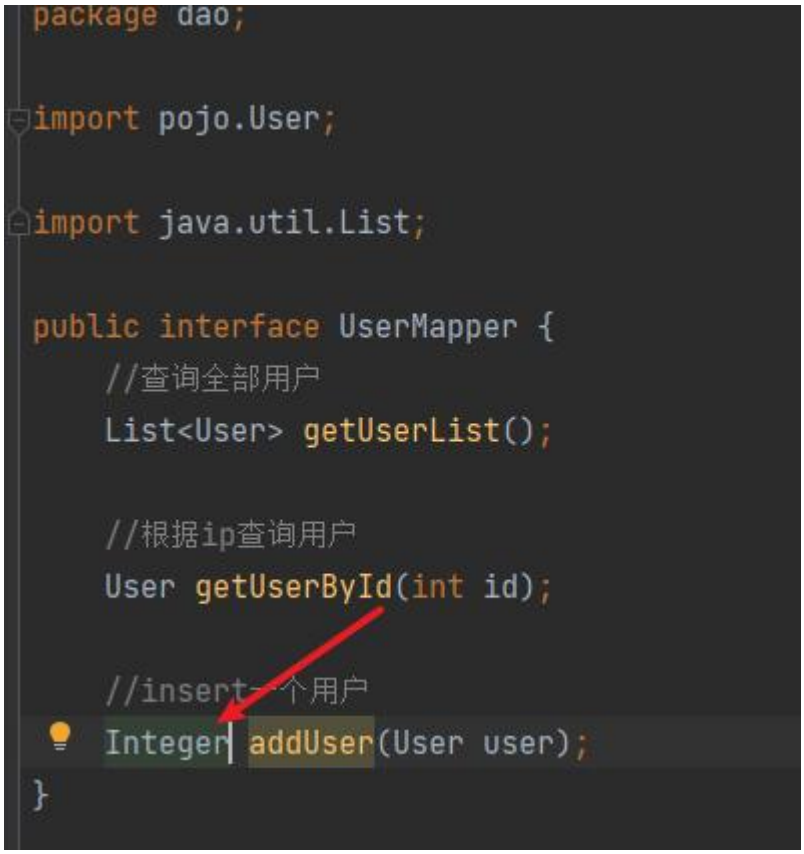
理下来，逻辑就很清晰了。

Insert

这里也是神坑

就算跟着视频都不一定对

首先 interface 这里定义的东西不能为 int



这是解决方法

下面的 insert 语句

要改为包装类型 integer

然后这里需要增加事务

```
SqlSession sqlSession = MybatisUtils.getSqlSession();
UserMapper mapper = sqlSession.getMapper(UserMapper.class);
mapper.addUser(new User( id: 4,  name: "fucku",  pwd: "fucku"));

sqlSession.commit();
//增删改需要提前插入事务
sqlSession.close();
```

视频说提交了事务才能插入成功
但是我这里没用事务也成功了
原因未知

Update
一样的套路

```
@Test
public void updateUser(){
    SqlSession sqlSession = MybatisUtils.getSqlSession();
    UserMapper mapper = sqlSession.getMapper(UserMapper.class);
    mapper.updateUser(new User( id: 3,  name: "bitch1",  pwd: "111111"));

    sqlSession.commit();
    //增删改需要提前插入事务
    sqlSession.close();
```

1	1	狂神	123456
2	2	张三	abcdef
3	3	bitch1	111111
4	4	fucku	fucku
5	5	fucku	fucku

Remove

```
@Test
public void deleteUser(){
    SqlSession sqlSession = MybatisUtils.getSqlSession();
    UserMapper mapper = sqlSession.getMapper(UserMapper.class);

    mapper.deleteUser( id: 5);

    sqlSession.commit();
    //增删改需要提前插入事务
    sqlSession.close();
}
```

WHERE ORDER BY

	id	name	pwd
1	1	狂神	123456
2	2	张三	abcdef
3	3	bitch1	111111
4	4	fucku	fucku

一个套路

Map 模糊&查询

map 可以理解为 python 中的 dict

本质就是键值对

模糊查询就是 like 然后跟通配符

配置之属性优化

MyBatis 的配置文件包含了会深深影响 MyBatis 行为的设置和属性信息

- configuration（配置）
  - properties（属性）
  - settings（设置）
  - typeAliases（类型别名）
  - typeHandlers（类型处理器）
  - objectFactory（对象工厂）
  - plugins（插件）
  - environments（环境配置）
    - environment（环境变量）
      - transactionManager（事务管理器）
      - dataSource（数据源）
  - databaseIdProvider（数据库厂商标识）
  - mappers（映射器）

根据作者的思路，这几个是重点

不要全部都学 浪费时间

数据源（dataSource）

dataSource 元素使用标准的 JDBC 数据源接口来配置 JDBC 连接对象的资源。

- 大多数 MyBatis 应用程序会按示例中的例子来配置数据源。虽然数据源配置是可选的，但如果要启用延迟加载特性，就必须配置数据源。

有三种内建的数据源类型（也就是 type="[UNPOOLED|POOLED|JNDI]"）：

这个 jndi 还是很熟悉的

然后 mybatis 默认的事务管理是 jdbc

默认的连接数据源是 pooled

```
<environments default="development">
    <environment id="development">
        <transactionManager type="JDBC"/>
        <dataSource type="POOLED">
            <property name="driver" value="com.mysql.cj.jdbc.Driver
            <property name="url" value="jdbc:mysql://127.0.0.1:3306
            <property name="username" value="root"/>
            <property name="password" value="root"/>
        </dataSource>
    </environment>
```

还有一些配置文件的引入方法 不赘述

这里有个技巧
看视频刚开始看不能开 2 倍速
因为万事开头难。。。
刚开始看 2 倍速确实容易懵逼
但是慢慢往后看
2 倍速其实还行，因为框架已经掌握了
就是填充细节 所以其实还行

类型别名
把长的名字设置成短的名字
和数据库中也是一样的
也可以使用注解来加上别名

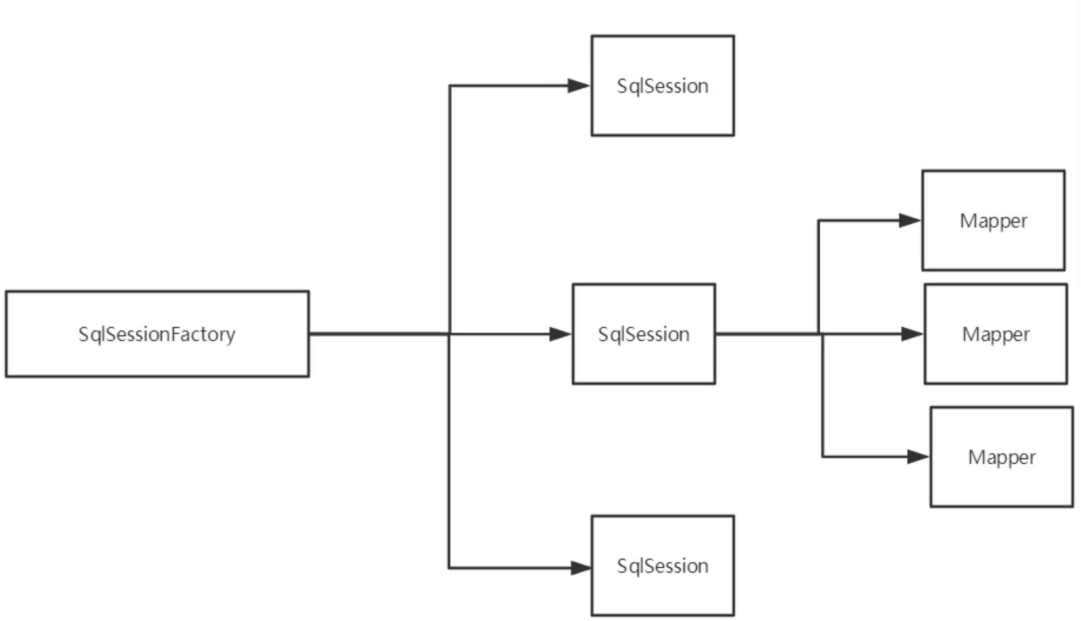
配置
就是 mapper 文件的语法写法
简单 不赘述

生命周期
这个东西是全局应用

SqlSessionFactory:

- 说白了就是可以想象为： 数据库连接池
- SqlSessionFactory 一旦被创建就应该在应用的运行期间一直存在，**没有任何理由丢弃它或重新创建另一个实例。**
- 因此 SqlSessionFactory 的最佳作用域是应用作用域。
- 最简单的就是使用**单例模式**或者静态单例模式。

下面这个图更加直观
Sqlsessionfactory 是一个全局存在的东西
然后每一个 sqlsession 是一个线程



然后每个 mapper 都代表一个具体的业务

结果集

要下面这些繁琐的配置。 但出于示范的原因，让我们来看看最后一个示例中，如果使用外部的 `resultMap` 会怎样，这也是解决列名不匹配的另外一种方式。

```
<resultMap id="userResultMap" type="User">
  <id property="id" column="user_id" />
  <result property="username" column="user_name"/>
  <result property="password" column="hashed_password"/>
</resultMap>
```

而在引用它的语句中使用 `resultMap` 属性就行了（注意我们去掉了 `resultType` 属性）。比如:

```
<select id="selectUsers" resultMap="userResultMap">
  select user_id, user_name, hashed_password
  from some_table
  where id = #{id}
</select>
```

我理解

就是别名

把不熟悉的名称用自己熟悉的别名替代

。 。 。

日志工厂

看日志来排错

	合或关联）。（新增于 3.4.2）		
logPrefix	指定 MyBatis 增加到日志名称的前缀。	任何字符串	未设置
logImpl	指定 MyBatis 所用日志的具体实现，未指定时将自动查找。	SLF4J | LOG4J | LOG4J2 | JDK_LOGGING | COMMONS_LOGGING | STDOUT_LOGGING | NO_LOGGING	未设置
proxyFactory	指定 Mybatis 创建可延迟加载对象所用到的代理工具。	CGLIB | JAVASSIST	JAVASSIST （M

又看到熟悉的东西了

实现

```
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class UserMapperTest {
    @Test
    public void test(){...}


    @Test
    public void getUserById(){...}

    @Test
```

```
Tests passed: 1 of 1 test – 2 sec 61 ms
<==        Row: 4, fucku, fucku
<==      Total: 4
User{id=1, name='狂神', pwd='123456'}
User{id=2, name='张三', pwd='abcdef'}
User{id=3, name='bitch1', pwd='111111'}
User{id=4, name='fucku', pwd='fucku'}
Resetting autocommit to true on JDBC Connection [com.mysql.cj.jdbc.ConnectionImpl@49438269]
Closing JDBC Connection [com.mysql.cj.jdbc.ConnectionImpl@49438269]
Returned connection 1229161065 to pool.
```

配置 log4j

导包

```
<dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>2.17.2</version>
    <type>pom</type>
</dependency>
```

设置

log4j.properties | UserMapper.java | UserMapperTest.java | pom.xml (mybatistest1)

```
#控制台输出的相关设置
log4j.appender.console = org.apache.log4j.ConsoleAppender
log4j.appender.console.Target = System.out
log4j.appender.console.Threshold=DEBUG
log4j.appender.console.layout = org.apache.log4j.PatternLayout
log4j.appender.console.layout.ConversionPattern=[%c]-%m%n

#文件输出的相关设置
log4j.appender.file = org.apache.log4j.RollingFileAppender
log4j.appender.file.File=./log/kuang.log
log4j.appender.file.MaxFileSize=10mb
log4j.appender.file.Threshold=DEBUG
log4j.appender.file.layout=org.apache.log4j.PatternLayout
log4j.appender.file.layout.ConversionPattern=[%p][%d{yy-MM-dd}][%c]%m%n

#日志输出级别
log4j.logger.org.mybatis=DEBUG
log4j.logger.java.sql=DEBUG
```

测试运行

```
[org.apache.ibatis.logging.LogFactory]-Logging initialized using 'class org.apache.ibatis.logging.log4j.Log4jImpl' adapter.
[org.apache.ibatis.logging.LogFactory]-Logging initialized using 'class org.apache.ibatis.logging.log4j.Log4jImpl' adapter.
[org.apache.ibatis.datasource.pooled.PooledDataSource]-PooledDataSource forcefully closed/removed all connections.
[org.apache.ibatis.datasource.pooled.PooledDataSource]-PooledDataSource forcefully closed/removed all connections.
[org.apache.ibatis.datasource.pooled.PooledDataSource]-PooledDataSource forcefully closed/removed all connections.
[org.apache.ibatis.datasource.pooled.PooledDataSource]-PooledDataSource forcefully closed/removed all connections.
[org.apache.ibatis.transaction.jdbc.JdbcTransaction]-Opening JDBC Connection
[org.apache.ibatis.datasource.pooled.PooledDataSource]-Created connection 293508253.
[org.apache.ibatis.transaction.jdbc.JdbcTransaction]-Setting autocommit to false on JDBC Connection [com.mysql.cj.jdbc.ConnectionImp
[dao.UserMapper.getUserList]-==>  Preparing: select * from mybatis.user
[dao.UserMapper.getUserList]-==> Parameters:
```

使用 log4j

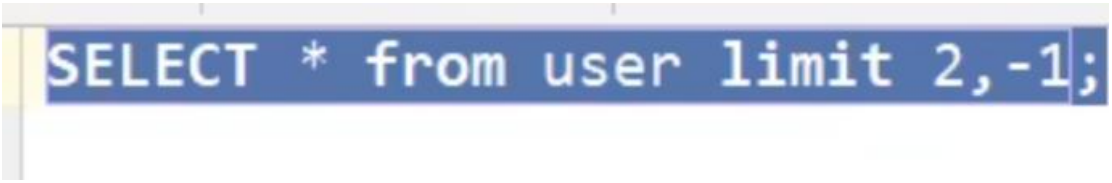
```
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import org.apache.log4j.Logger;

public class UserMapperTest {

    static Logger logger = Logger.getLogger((UserMapperTest.clas

    @Test
    public void testLog4j(){
        System.out.println();
        logger.info("info:enter testlog4j");
        logger.debug("debug:enter testlog4j");
        logger.error("error:enter testlog4j");
    }

    @Test
    public void test(){...}
```

Tests passed: 1 of 1 test – 4 ms

```
"C:\Program Files\Java\jdk1.8.0_181\bin\java.exe" ...

[dao.UserMapperTest]-info:enter testlog4j
[dao.UserMapperTest]-debug:enter testlog4j
[dao.UserMapperTest]-error:enter testlog4j
```

可以看到成功输出了

Limit 分页



就是使用 limit 限定

Sql 注入的时候用的多

Mybatis 实现分页

无非就是用 mybatis 封装一下这个 limit

这里作者是用了一个 hashmap 来做

```
@Test
public void getUserByLimit(){
    SqlSession sqlSession = MybatisUtils.getSqlSession();
    UserMapper mapper = sqlSession.getMapper(UserMapper.class);

    HashMap<String, Integer> map = new HashMap<String, Integer>();
    map.put("startIndex",0);
    map.put("pageSize",2);

    List<User> userList =  mapper.getUserByLimit(map);


    sqlSession.close();
}
```

本质还是 limit

Rowbound 实现分页

字面理解

Row 行

Bound 边界

```
@Test
public void getUserByRowBounds(){
SqlSession sqlSession = MybatisUtils.getSqlSession();

//RowBounds实现
RowBounds rowBounds = new RowBounds(1, 2);

//通过Java代码层面实现分页
List<User> userList =
sqlSession.selectList("com.kuang.dao.UserMapper.getUserByRowBounds",null,rowBounds
);
```

这个东西好理解

效果和 limit 也是一样的

截取

还可以用插件

MyBatis 分页插件 PageHelper

使用注解开发
就是面向接口编程
后期玩到 springboot 基本就是这一套东西

```java
public interface UserMapper {
    @Select("select * from user")
    List<User> getUsers();
}
```

这里是写好了一个注解
然后写一个 test

```java
    public void testfuck(){
        SqlSession sqlSession = MybatisUtils.getSqlSession();

        UserMapper mapper = sqlSession.getMapper(UserMapper.class);
        List<User> users = mapper.getUsers();
        for(User user: users){
            System.out.println(user);
        }
        sqlSession.close();
    }
```

得到结果

```
log4j:WARN See http://logging.apache.org
User{id=1, name='狂神', pwd='123456'}
User{id=2, name='张三', pwd='abcdef'}
User{id=3, name='bitch1', pwd='111111'}
User{id=4, name='fucku', pwd='fucku'}
```

简单，不赘述

Mybatis 运行流程

```
public class MybatisUtils {

    private static SqlSessionFactory sqlSessionFactory;

    static{

        try {
            String resource = "mybatis-config.xml";
            InputStream inputStream = Resources.getResourceAsStream(resource);
            sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public static SqlSession getSqlSession() { return sqlSessionFactory.openSession(); }

}
```

跟一下这个 build 方法

```
public SqlSessionFactory build(InputStream inputStream) {
  return build(inputStream, environment: null, properties: null);
}
```

Build 继续调用 build 方法

继续跟

```
public SqlSessionFactory build(InputStream inputStream, String environment, Properties properties) {
  try {
    XMLConfigBuilder parser = new XMLConfigBuilder(inputStream, environment, properties);
    return build(parser.parse());
  } catch (Exception e) {
    throw ExceptionFactory.wrapException("Error building SqlSession.", e);
  } finally {
    ErrorContext.instance().reset();
    try {
      inputStream.close();
    } catch (IOException e) {
      // Intentionally ignore. Prefer previous error.
    }
  }
}
```

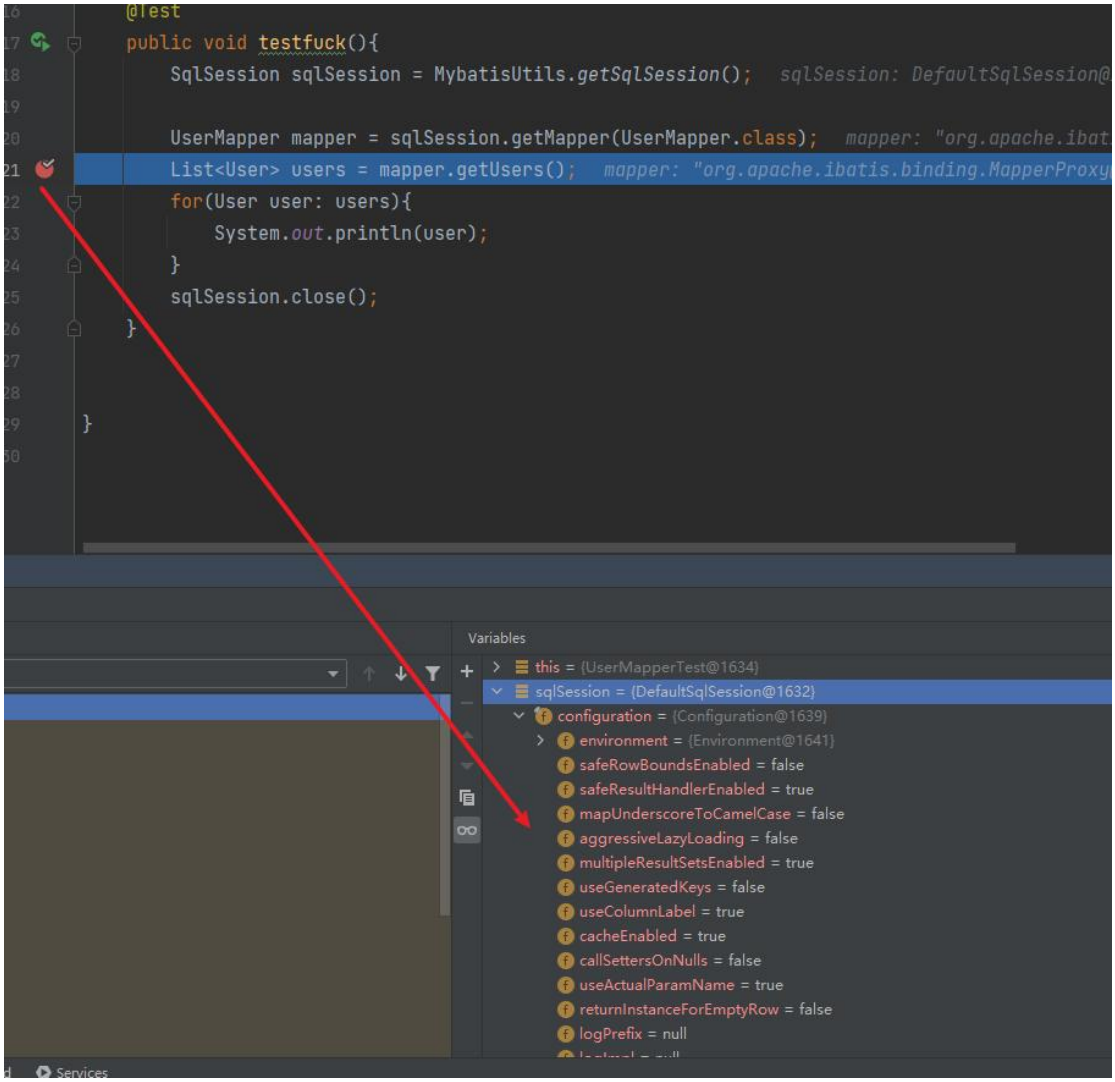
发现

这里调用了一个 xml 解析器

```
XMLConfigBuilder parser = new XMLConfigBuilder(inputStream, environment, properties);
return build(parser.parse());
```

这里解析文件，然后 return 解析过的 xml 文件

下个断点调试一下

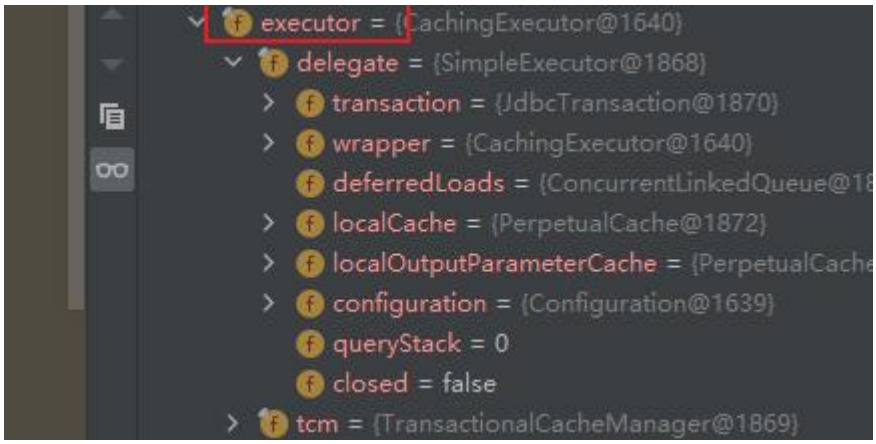


这里可以很清晰的看到

Configuration 都已经被解析出来了

然后往下走

是个 executor



是个执行器

执行事务

这里我 into 进去看了一下

```
public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {   method: "public abstract
  try {
    if (Object.class.equals(method.getDeclaringClass())) {   method: "public abstract java.util.List dao.UserM
      return method.invoke( obj: this, args);
    } else if (method.isDefault()) {
      return invokeDefaultMethod(proxy, method, args);
    }
  } catch (Throwable t) {
    throw ExceptionUtil.unwrapThrowable(t);
  }
  final MapperMethod mapperMethod = cachedMapperMethod(method);
  return mapperMethod.execute(sqlSession, args);
}
```

底层其实还是反射

跟一下这个 execute 方法

```
public Object execute(SqlSession sqlSession, Object[] args) {   sqlSession: DefaultSqlSession@1632   args: null
  Object result;
  switch (command.getType()) {   command: MapperMethod$SqlCommand@1858
    case INSERT: {
      Object param = method.convertArgsToSqlCommandParam(args);
      result = rowCountResult(sqlSession.insert(command.getName(), param));
      break;
    }
    case UPDATE: {
      Object param = method.convertArgsToSqlCommandParam(args);
      result = rowCountResult(sqlSession.update(command.getName(), param));
      break;
    }
    case DELETE: {
      Object param = method.convertArgsToSqlCommandParam(args);
      result = rowCountResult(sqlSession.delete(command.getName(), param));
      break;
```

要用什么就选什么

这里跳到了 select

```
    }
    case SELECT:
      if (method.returnsVoid() && method.hasResultHandler()) {   method: MapperMethod$MethodSignature@1859
        executeWithResultHandler(sqlSession, args);
        result = null;
      } else if (method.returnsMany()) {
        result = executeForMany(sqlSession, args);
      } else if (method.returnsMap()) {
        result = executeForMap(sqlSession, args);
      } else if (method.returnsCursor()) {
        result = executeForCursor(sqlSession, args);
      } else {
        Object param = method.convertArgsToSqlCommandParam(args);
        result = sqlSession.selectOne(command.getName(), param);
        if (method.returnsOptional()
```

跟下来

```
if (result == null && false && method.getReturnType().isPrimitive() && !method.returnsVoid()) {   result:   size = 4   method: MapperMethod$MethodSignatur
  throw new BindingException("Mapper method '" + command.getName()
      + " attempted to return null from a method with a primitive return type (" + method.getReturnType() + ").");
}
return result;
```

这里其实已经拿到结果了

然后把结果 return 回去

```
  }
  return result;   result:   size = 4
```

这里跳回上一层

```
return mapperMethod.execute(sqlSession, args);   args: null    mapperMethod: MapperMethod@1854    sqlSession: DefaultSi
```

然后直接的结果已经给到 users 了

```
@Test
public void testfuck(){
    SqlSession sqlSession = MybatisUtils.getSqlSession();   sqlSession: DefaultSqlSession@1632

    UserMapper mapper = sqlSession.getMapper(UserMapper.class);   mapper: "org.apache.ibatis.binding.Mapper
    List<User> users = mapper.getUsers();   mapper: "org.apache.ibatis.binding.MapperProxy@1d296da"
    for(User user: users){
        System.out.println(user);
    }
    sqlSession.close();
}
```

最后走 for 循环，一一输出结果

```
for(User user: users){   users:  size = 4    user: "User{id=1, name='狂神', pwd='123456'}"
    System.out.println(user);   user: "User{id=1, name='狂神', pwd='123456'}"
}
sqlSession.close();
}
```

注解增删改查

Crud

其实我懒得写注解的 crud 了。。。

还是跟着视频看一遍吧。。。

我这里其实对后面这几课

P18	18、Lombok的使用	15:56
P19	19、复杂查询环境搭建	18:49
P20	20、多对一的处理	24:06
P21	21、一对多的处理	30:37
P22	22、动态SQL环境搭建	19:21
P23	23、动态SQL之IF语句	09:43
P24	24、动态SQL常用标签	30:53
P25	25、动态SQL之Foreach	30:58
P26	26、缓存简介	15:25

这个动态 sql 比较感兴趣

因为跟注入可能有关

而且以后审到 mybatis 的框架，也有可能看到，到时候不会审就完犊子了

下面看下这个注解

Insert

```
@Insert("insert into user(id,name,pwd) values (#{id},#{name},#{password})")
int addUser(User user);
```

就是写法不一样而已
其实效果也是一样的
快速的过一遍
Update

```
@Update("update user set name=#{name},pwd=#{password} where id = #{id}")
int updateUser(User user);
```

Delete

```
@Delete("delete from user where id = #{id}")
int deleteUser(@Param("uid") int id);
```

@param 注解
百度了下

1、概述

首先明确这个注解是为 SQL语句 中参数赋值而服务的。

@Param的作用就是给参数命名，比如在mapper里面某方法A（int id），当添加注解后A（@Param("userId") int id），想要取出传入的id值，只需要取它的参数名userId就可以了。将参数值传如SQL语句中，通过#{userId}进行取值给SQL的参

2、实例:

实例一：@Param注解基本类型的参数

mapper中的方法:

```
public User selectUser(@Param("userName") String name,@Param("password") String pwd);
```

映射到xml中的<select>标签

```
1 <select id="selectUser" resultMap="User">
2    select * from user  where user_name = #{userName} and user_password=#{password}
3 </select>
```

其中where user_name = #{userName} and user_password = #{password}中的userName和password都是从注解@Pa

我理解 就是别名
还有个
#{}
${}
的区别
第一个能防止 sql 注入 是用的 pdo
第二个是拼接

Lombok

lombok的作用及注释

2021-08-16 19:35:06

一， lombok作用

作用: 简化用户创建**实体对象(POJO)**的过程,由插件自动的完成实体对象中常用方法的构建(get/set/toString/构造等)

二， Lombok链式加载

```
//注解的作用: 动态的生成get/set/toString....方法
@Data   //一般为属性赋值 get/set方法
@NoArgsConstructor  //无参构造
@AllArgsConstructor //全参构造
@Accessors(chain = true) //开启链式加载(重写set方法)
public class User {
```

简化 pojo 用的

直接过

多对一和一对多

多个学生对应一个班主任 //关联

一个学生对应多个科任老师 //集合

简单看看就行

毕竟不是开发

动态 sql

先看官方文档

直觉

这个东西能搞漏洞

因为他用了 ognl

而且后面还有个这个

script

要在带注解的映射器接口类中使用动态 SQL，可以使用 *script* 元素。比如:

```
@Update({"<script>",
  "update Author",
  "  <set>",
  "    <if test='username != null'>username=#{username},</if>",
  "    <if test='password != null'>password=#{password},</if>",
  "    <if test='email != null'>email=#{email},</if>",
  "    <if test='bio != null'>bio=#{bio}</if>",
  "  </set>",
  "where id=#{id}",
  "</script>"})
void updateAuthorValues(Author author);
```

还有这个

动态 SQL 中的插入脚本语言

MyBatis 从 3.2 版本开始支持插入脚本语言，这允许你插入一种语言驱动，并基于这种语言来编写动态 SQL 查询语句。

可以通过实现以下接口来插入一种语言：

```
public interface LanguageDriver {
  ParameterHandler createParameterHandler(MappedStatement mappedStatement, Object parameterObject, BoundSql boundS
  SqlSource createSqlSource(Configuration configuration, XNode script, Class<?> parameterType);
  SqlSource createSqlSource(Configuration configuration, String script, Class<?> parameterType);
}
```

没做实验，先放在这里，回头再来仔细研究。
然后这个动态 sql 本质上就是条件判断
在执行的时候还可以做条件匹配
比如

使用动态 SQL 最常见情景是根据条件包含 where 子句的一部分。比如：

```
<select id="findActiveBlogWithTitleLike"
     resultType="Blog">
  SELECT * FROM BLOG
  WHERE state = 'ACTIVE'
  <if test="title != null">
    AND title like #{title}
  </if>
</select>
```

这条语句提供了可选的查找文本功能。如果不传入 “title”，那么所有处于 “ACTIVE” 状态的 BLOG 都会返回；如果传入了 “title” 参数，那么就会对 “

一列进行模糊查找并返回对应的 BLOG 结果（细心的读者可能会发现，“title” 的参数值需要包含查找掩码或通配符字符）。

如果希望通过 “title” 和 “author” 两个参数进行可选搜索该怎么办呢？首先，我想先将语句名称修改成更名副其实的名称；接下来，只需要加入另一个条

相当于这里加了一个 if 判断
但是是用他的语法加的
他这里封装了一层
我估计底层也就是个逻辑判断语句

缓存

> **mybatis一级缓存和二级缓存的区别：**
>
> 1）一级缓存 Mybatis的一级缓存是指SQLSession，一级缓存的作用域是SQlSession, Mabits默认开启一级缓存。 在同一个SqlSession中，执行相同的SQL查询时；第一次会去查询数据库，并写在缓存中，第二次会直接从缓存中取。 当执行SQL时候两次查询中间发生了增删改的操作，则SQLSession的缓存会被清空。
>
> 每次查询会先去缓存中找，如果找不到，再去数据库查询，然后把结果写到缓存中。 Mybatis的内部缓存使用一个HashMap，key为hashcode+statementId+sql语句。Value为查询出来的结果集映射成的java对象。 SqlSession执行insert、update、delete等操作commit后会清空该SQLSession缓存。
>
> 2）二级缓存 二级缓存是mapper级别的，Mybatis默认是没有开启二级缓存的。 第一次调用mapper下的SQL去查询用户的信息，查询到的信息会存放代该mapper对应的二级缓存区域。 第二次调用namespace下的mapper映射文件中，相同的sql去查询用户信息，会去对应的二级缓存内取结果。

这个好理解
然后这里有个重点

> 1）一级缓存 Mybatis的一级缓存是指SQLSession，一级缓存的作用域是SQlSession, Mabits默认开启一级缓存。 在同一个SqlSession中，执行相同的SQL查询时；第一次会去查询数据库，并写在缓存中，第二次会直接从缓存中取。 当执行SQL时候两次查询中间发生了增删改的操作，则SQLSession的缓存会被清空。

所以不用担心注入 mybatis 会一直有缓存
动了库 缓存就会清空

Mybatis 总结
单体：用 mybatis 实现增删改查
更多的配置：注解 xml 配置
排错：日志 log4j
复杂的数据关系：一对多 多对一
Sql 高级特性：动态 sql
缓存：buffer 提速用的

---2020509 6:09---
mybatis 的基础基本看完了
主要中途还打了好几把游戏 不然还能再快点
这里学习我觉得还有个关键点
就是不要去记忆 要去理解
理解逻辑 之后就算忘记了也可以根据现有条件再推出来
但是如果是死记 忘了就是忘了 就相当于白学了
