SQL注入攻击指的是通过构建特殊的输入作为参数传入Web应用程序，而这些输入大都是SQL语法里的一些组合，通过执行SQL语句进而执行攻击者所要的操作，其主要原因是程序没有细致地过滤用户输入的数据，致使非法数据侵入系统。

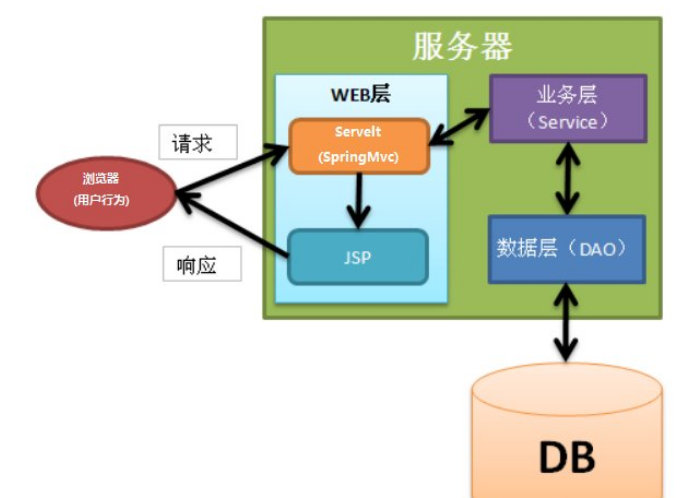
**Web程序三层架构**

三层架构(3-tier architecture) 通常意义上就是将整个业务应用划分为：

界面层（User Interface layer）

业务逻辑层（Business Logic Layer）

数据访问层（Data access layer）



当我们访问动态网页时, Web 服务器会向数据访问层发起 Sql 查询请求，如果权限验证通过就会执行 Sql 语句。 这种网站内部直接发送的Sql请求一般不会有危险，但实际情况是很多时候需要结合用户的输入数据动态构造 Sql 语句，如果用户输入的数据被构造成恶意 Sql 代码，Web 应用又未对动态构造的 Sql 语句使用的参数进行审查，则会带来意想不到的危险。

这里我们举一个比较常见的例子来简要说明一下sql注入的原理。假如我们有一个users表，里面有两个字段username和password。在我们的java代码中我们初学者都习惯用sql拼接的方式进行用户验证。比如："select id from users where username = '"+username +"' and password = '" + password +"'" 这里的username和password都是我们存取从web表单获得的数据。下面我们来看一下一种简单的注入，如果我们在表单中username的输入框中输入' or 1=1-- ，password的表单中随便输入一些东西，假如这里输入123.此时我们所要执行的sql语句就变成了select id from users where username = '' or 1=1-- and password = '123'，我们来看一下这个sql，因为1=1是true，后面 and password = '123'被注释掉了。所以这里完全跳过了sql验证。

事实上SQL注入有很多种，按数据类型可以分为数字型、字符型和搜索型，按提交方式可分为GET型，POST型，Cookie型和HTTP请求头注入，按执行效果有可以分为报错注入、联合查询注入、盲注和堆查询注入，其中盲注又可分为基于bool的和基于时间的注入。

MySQL 支持多种类型，大致可以分为三类：数值、日期/时间和字符串(字符)类型。

1.判断是否存在 Sql 注入漏洞

最为经典的**单引号判断法**：  
在参数后面加上单引号,比如:

http://xxx/abc.php?id=1'

如果页面返回错误，则存在 Sql 注入。  
原因是无论字符型还是整型都会因为单引号个数不匹配而报错。  
（如果未报错，不代表不存在 Sql 注入，因为有可能页面对单引号做了过滤，这时可以使用判断语句进行注入）

数字型判断：

当输入的参 x 为整型时，通常 abc.php 中 Sql 语句类型大致如下：  
select \* from <表名> where id = x  
这种类型可以使用经典的 and 1=1 和 and 1=2 来判断：

Url 地址中输入 http://xxx/abc.php?id= x and 1=1 页面依旧运行正常，继续进行下一步。

Url 地址中继续输入 http://xxx/abc.php?id= x and 1=2 页面运行错误，则说明此 Sql 注入为数字型注入。

原因如下：  
当输入 and 1=1时，后台执行 Sql 语句：

select \* from <表名> where id = x and 1=1 没有语法错误且逻辑判断为正确，所以返回正常。

当输入 and 1=2时，后台执行 Sql 语句：

select \* from <表名> where id = x and 1=2没有语法错误但是逻辑判断为假，所以返回错误。

字符型判断：

当输入的参 x 为字符型时，通常 abc.php 中 SQL 语句类型大致如下：  
select \* from <表名> where id = 'x'  
这种类型我们同样可以使用 and '1'='1 和 and '1'='2来判断：

Url 地址中输入 http://xxx/abc.php?id= x' and '1'='1 页面运行正常，继续进行下一步。

Url 地址中继续输入 http://xxx/abc.php?id= x' and '1'='2 页面运行错误，则说明此 Sql 注入为字符型注入。

原因如下：  
当输入 and '1'='1时，后台执行 Sql 语句：select \* from <表名> where id = 'x' and '1'='1'语法正确，逻辑判断正确，所以返回正确。

当输入 and '1'='2时，后台执行 Sql 语句：select \* from <表名> where id = 'x' and '1'='2'语法正确，但逻辑判断错误，所以返回正确。

通过以上的分析，我们仍然还会有疑问：黑客并不知道我们程序代码的逻辑和SQL语句的写法，他是如何确定一个网站是否存在SQL注入漏洞呢？一般说来有以下3种途径：

根据应用程序处理数据库返回内容的不同，可以分为可显注入、报错注入和盲注。

1.可显注入：攻击者可以直接在当前界面内容中获取想要获得的内容。

2 错误提示

如果目标Web网站开启了错误显示，攻击者就可以通过反复调整发送的参数、查看页面打印的错误信息，推测出Web网站使用的数据库和开发语言等重要信息。

3 盲注

除非运维人员疏忽，否则大部分的Web运营网站应该都关闭了错误提示信息，此时攻击者一般会采用盲注的技巧来进行反复的尝试判断。 仍然以上面的数据表user为例，我们之前的查看会员详情页面的url地址为 *userinfo.php?username=plhwin*，此时黑客分别访问 *userinfo.php?username=plhwin' AND 1=1-- hack* 和 *userinfo.php?username=plhwin' AND 1=2-- hack*，如果前者访问能返回正常的信息而后者不能，就基本可以判断此网站存在SQL注入漏洞，因为后者的 *1=2* 这个表达式永远不成立，所以即使username传入了正确的参数也无法通过，由此可以推断这个页面存在SQL注入漏洞，并且可以通过username参数进行注入。

**Sql 注入带来的威胁主要有如下几点**

* 猜解后台数据库，这是利用最多的方式，盗取网站的敏感信息。
* 绕过认证，列如绕过验证登录网站后台。
* 注入可以借助数据库的存储过程进行提权等操作

如何避免

1、检查变量数据类型和格式如果你的SQL语句是类似where id={$id}这种形式，数据库里所有的id都是数字，那么就应该在SQL被执行前，检查确保变量id是int类型；如果是接受邮箱，那就应该检查并严格确保变量一定是邮箱的格式，其他的类型比如日期、时间等也是一个道理。总结起来：只要是有固定格式的变量，在SQL语句执行前，应该严格按照固定格式去检查，确保变量是我们预想的格式，这样很大程度上可以避免SQL注入攻击。比如，我们前面接受username参数例子中，我们的产品设计应该是在用户注册的一开始，就有一个用户名的规则，比如5-20个字符，只能由大小写字母、数字以及一些安全的符号组成，不包含特殊字符。此时我们应该有一个check\_username的函数来进行统一的检查。不过，仍然有很多例外情况并不能应用到这一准则，比如文章发布系统，评论系统等必须要允许用户提交任意字符串的场景，这就需要采用过滤等其他方案了。

2、过滤特殊符号对于无法确定固定格式的变量，一定要进行特殊符号过滤或转义处理。以PHP为例，通常是采用addslashes函数，它会在指定的预定义字符前添加反斜杠转义，这些预定义的字符是：单引号 (') 双引号 (") 反斜杠 (\) NULL。

3.绑定变量，使用预编译语句

MySQL提供了预编译语句的支持

MySQL为例，现在项目大部分都用框架实现，比如MyBatis框架，轻量级，功能强大：在MyBatis中，如果写${变量名}，则为直接把传入的值填充到SQL语句中；如果写#{变量名}，则为传入的值只能作为值，放到SQL语句中。

"#{}"是将传入的值按照字符串的形式进行处理，如下面这条语句：

select user\_id,user\_name from t\_user where user\_id = #{user\_id}

MyBaits会首先对其进行预编译，将#{user\_ids}替换成?占位符，然后在执行时替换成实际传入的user\_id值，\*\*并在两边加上单引号，以字符串方式处理。

"${}"是做简单的字符串替换，即将传入的值直接拼接到SQL语句中，且不会自动加单引号。将上面的SQL语句改为：

select user\_id,user\_name from t\_user where user\_id = ${user\_id}

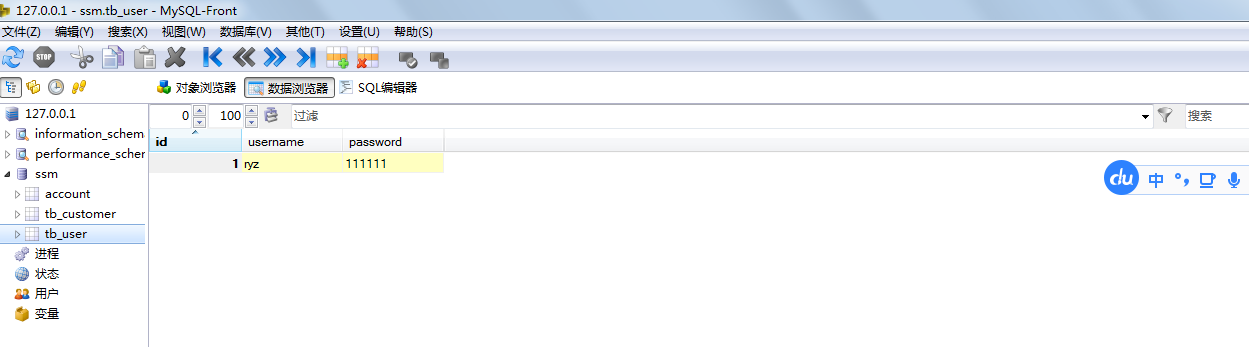
参数是直接替换的，且没有单引号处理，这样就有SQL注入的风险。

实验环境

基于Spring+Springmvc+Mybatis框架搭建的信息管理网站，采用的是Mysql数据库。

**登录模块**

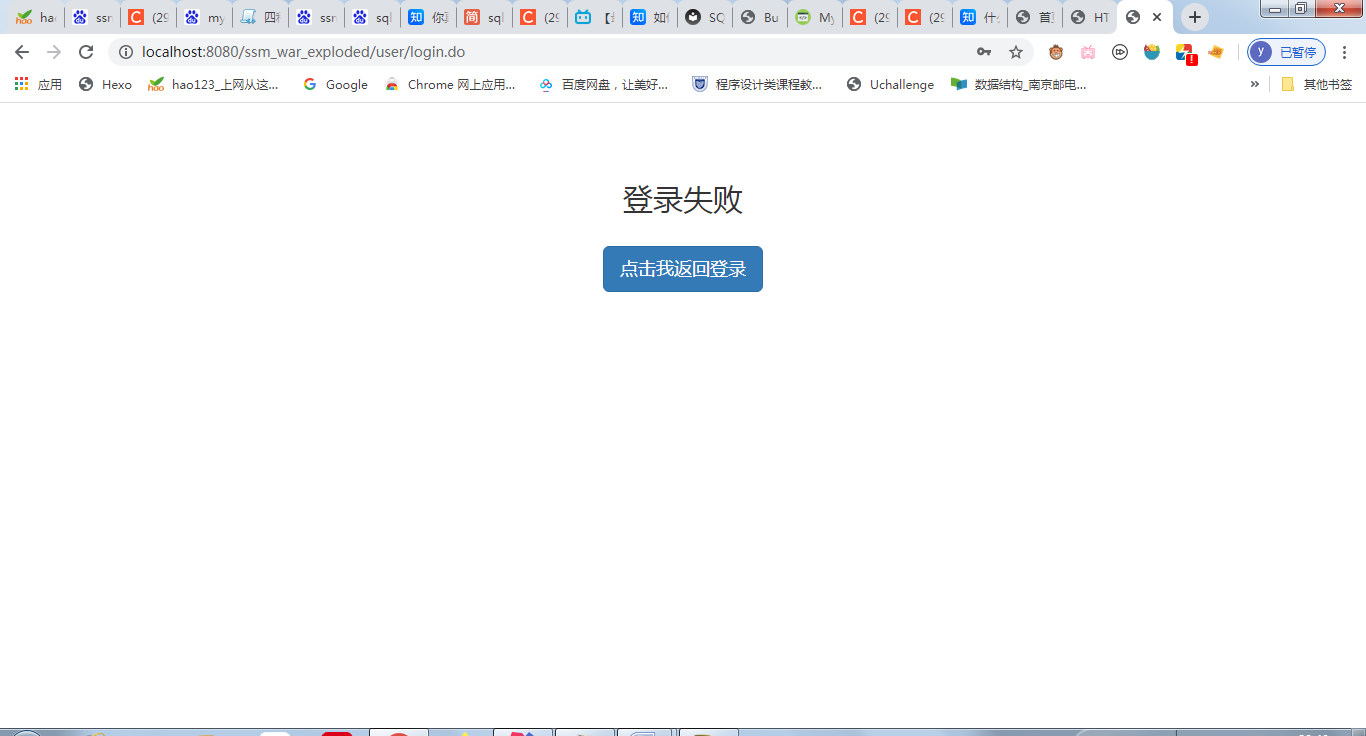
**我的注册信息**

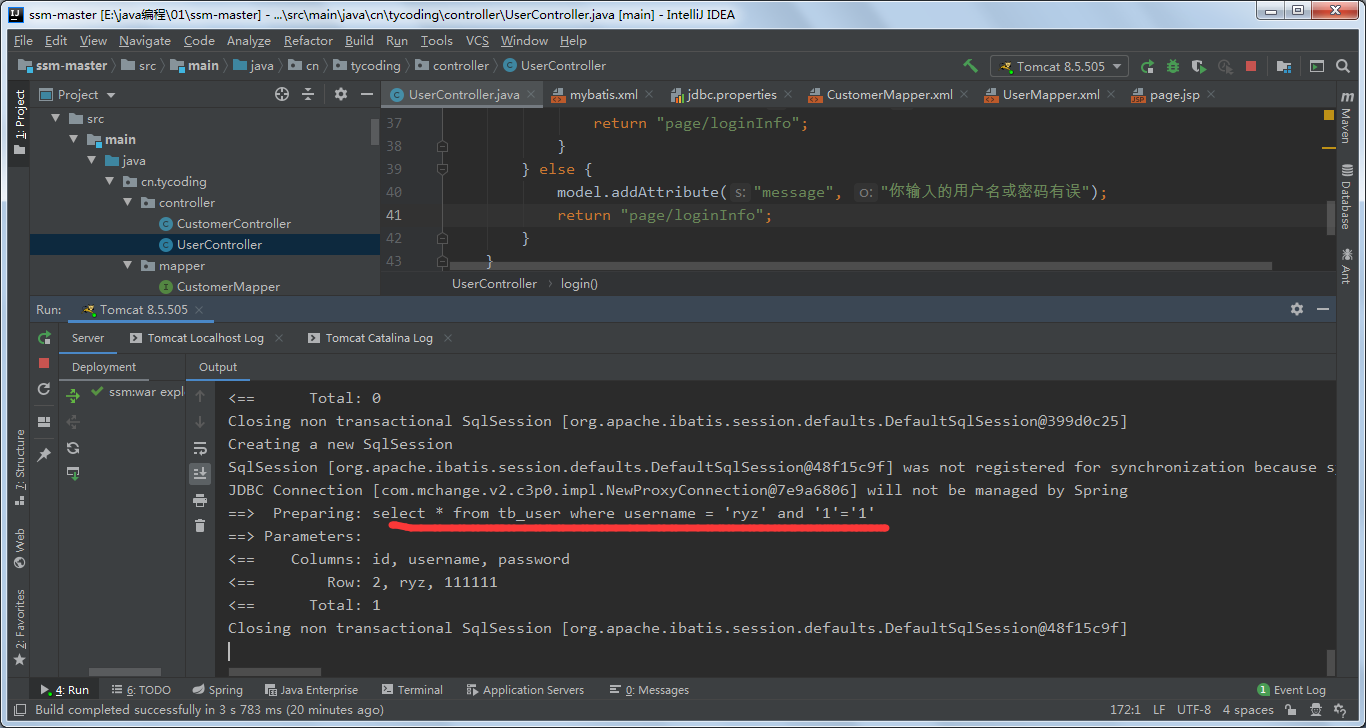
****

**ryz' and '1'='1**

**密码乱输**



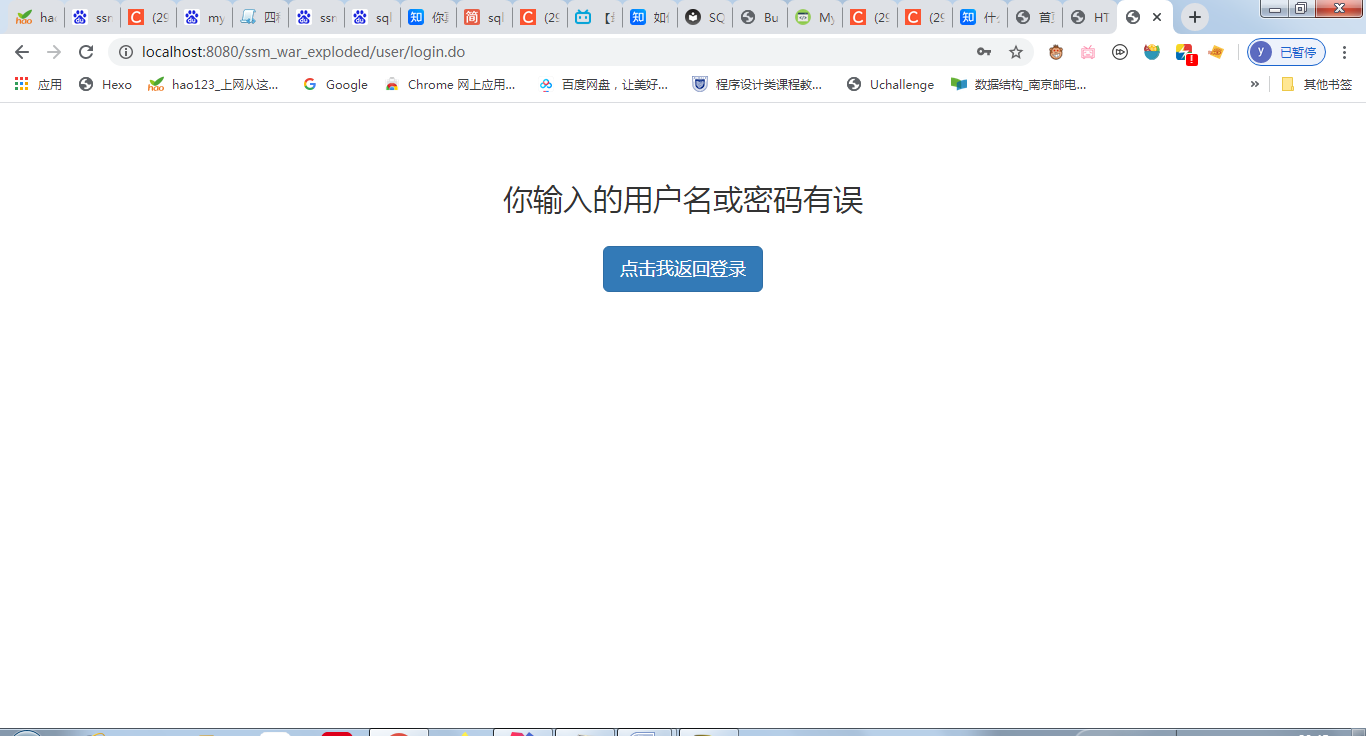


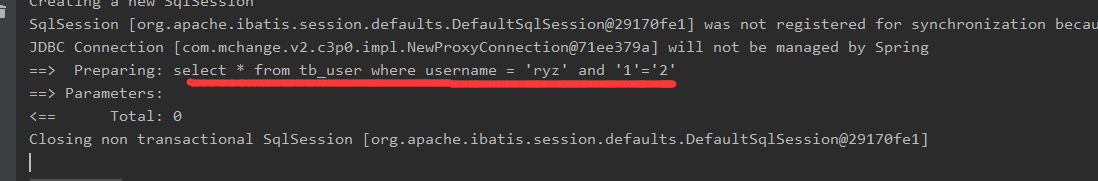


**ryz' and '1'='2**

**密码乱输**

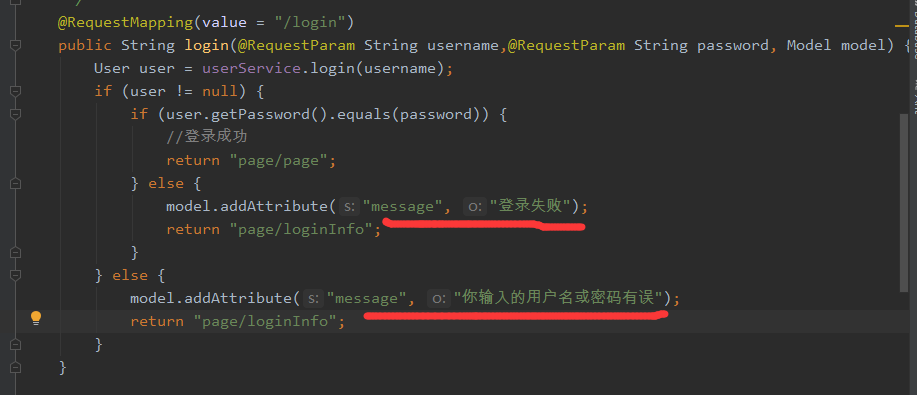






注意：两次尝试的报错页面不一样。这说明此页面很可能存在SQL注入！

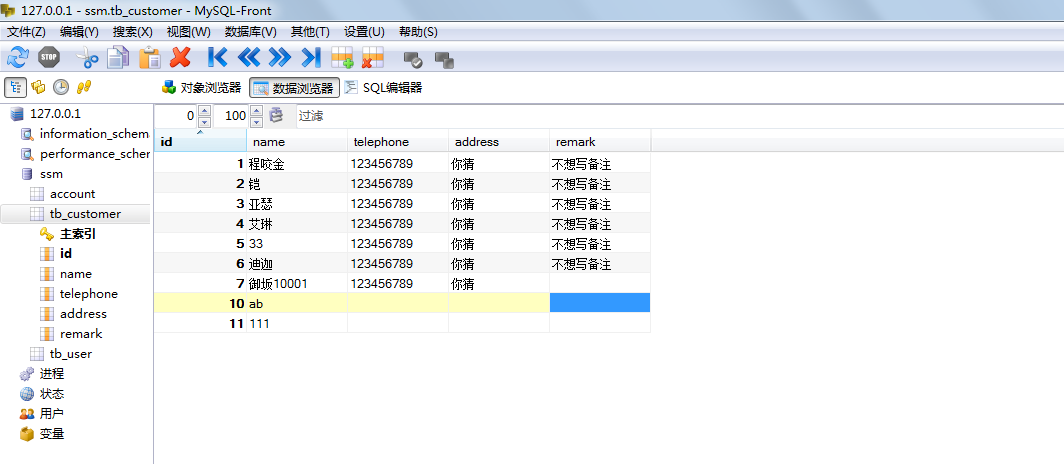
**原理：查看网站源码，我们可以发现当在数据库中查询用户名成功时，会返回登录失败提示；如果在数据库中压根查不到用户名，会返回你输入的用户名或密码有误**



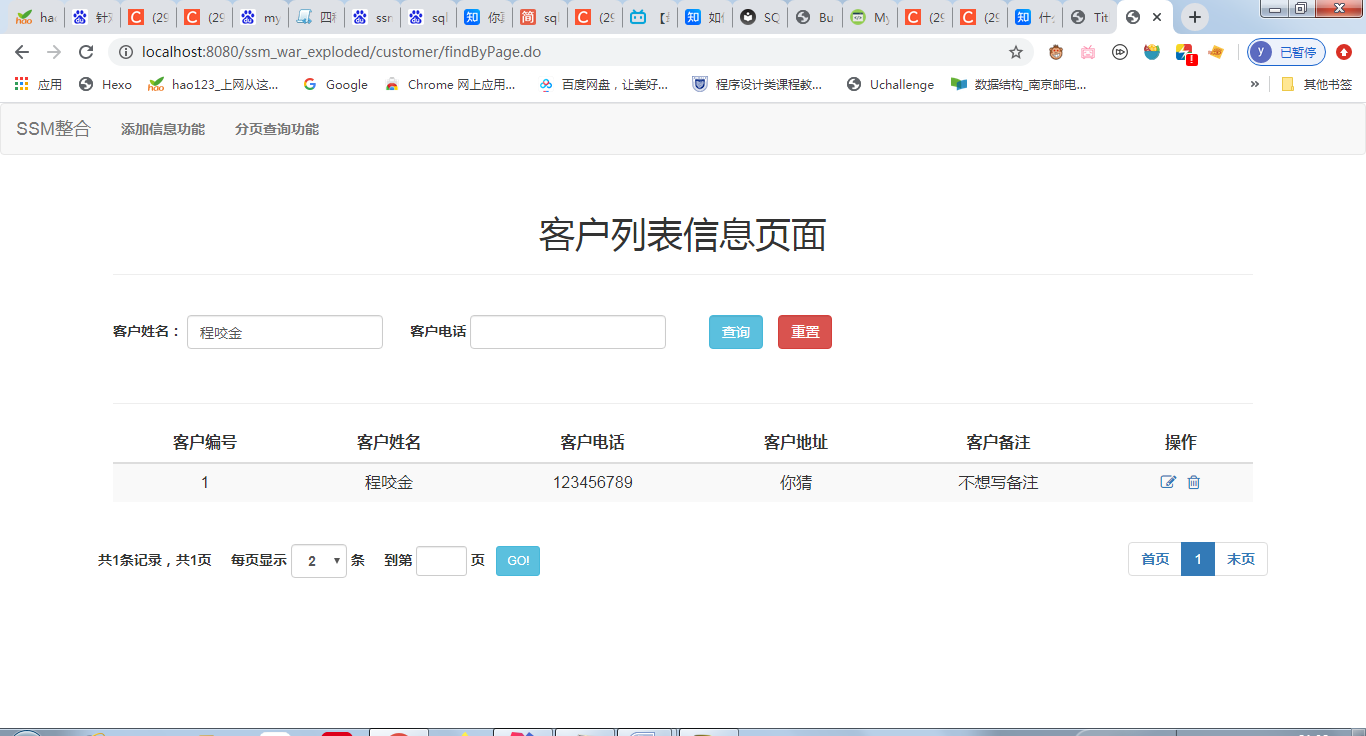
接下来，攻击者可能会尝试在用户名中输入 ryz’ or 1=1 #来尝试绕过登录，但却失败了。这是因为网站并没有采用简单的诸如select \* from user where username = '$name' and password = '$pwd'" 语句来判断是否放行，而是采用了多级判断的机制。先是判断数据库中是否存在传入的用户名，如果存在，再比较传入的密码和上一步中查出的密码是否一致，只有这两个都满足，才放行。可见，尽管后端开发程序员可能会犯动态拼接sql语句的问题，但只要程序登录逻辑严谨，还是可以有效避免凭借sql注入绕过登录。

**信息查询模块**

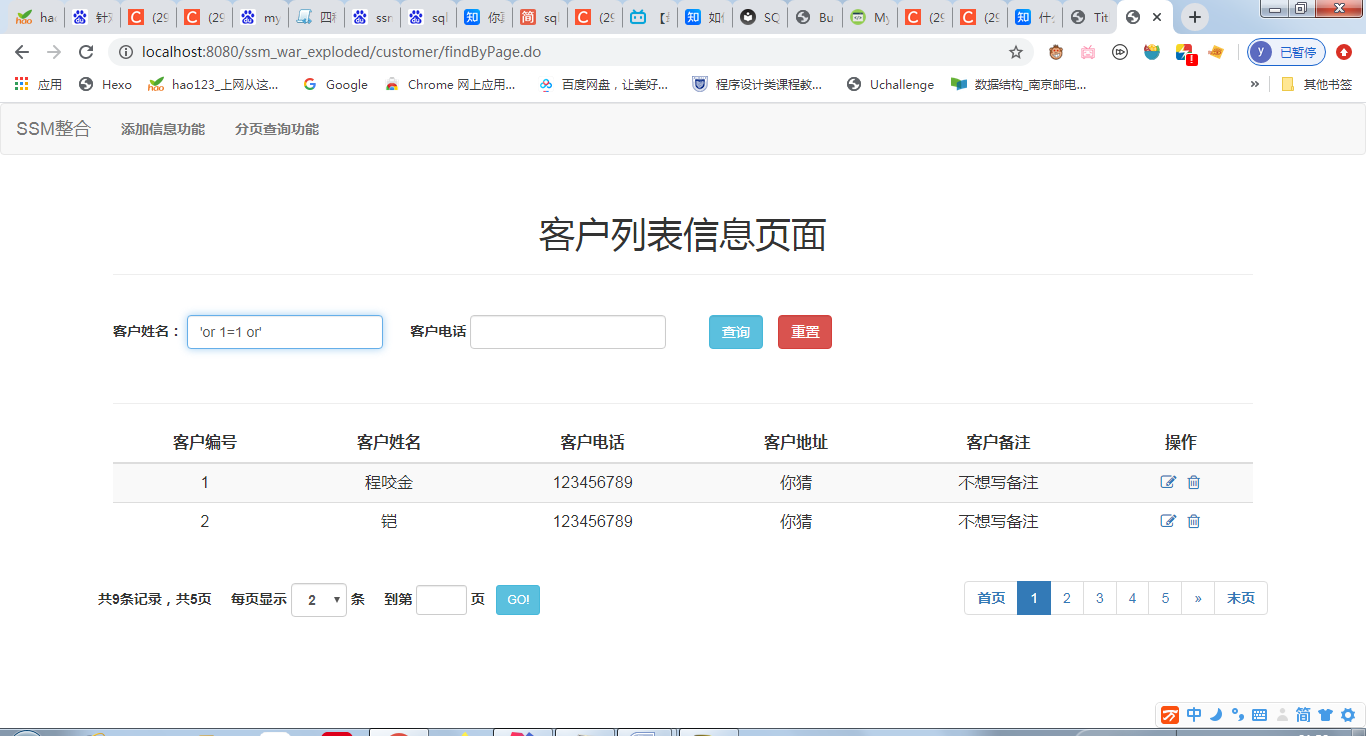
数据库中信息



现在我们只能查询我们知道的人名的信息，如何才能把所有人的信息全部查出来呢。



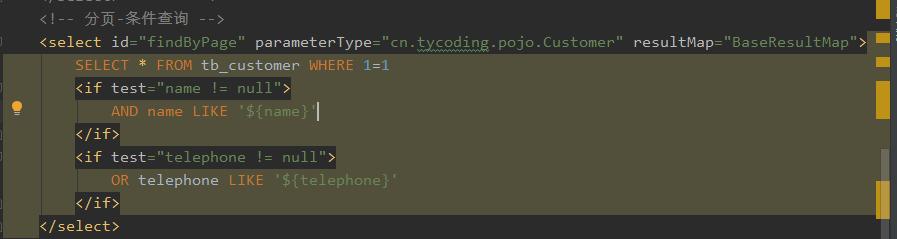
我们可以在客户姓名框中输入 'or 1=1 or'





之所以存在注入漏洞，是因为我在Mybatis框架里特意取消了预编译，而采用了sql语句拼接的方式。

这种是sql拼接方式的配置文件



这种是预编译方式的配置文件

