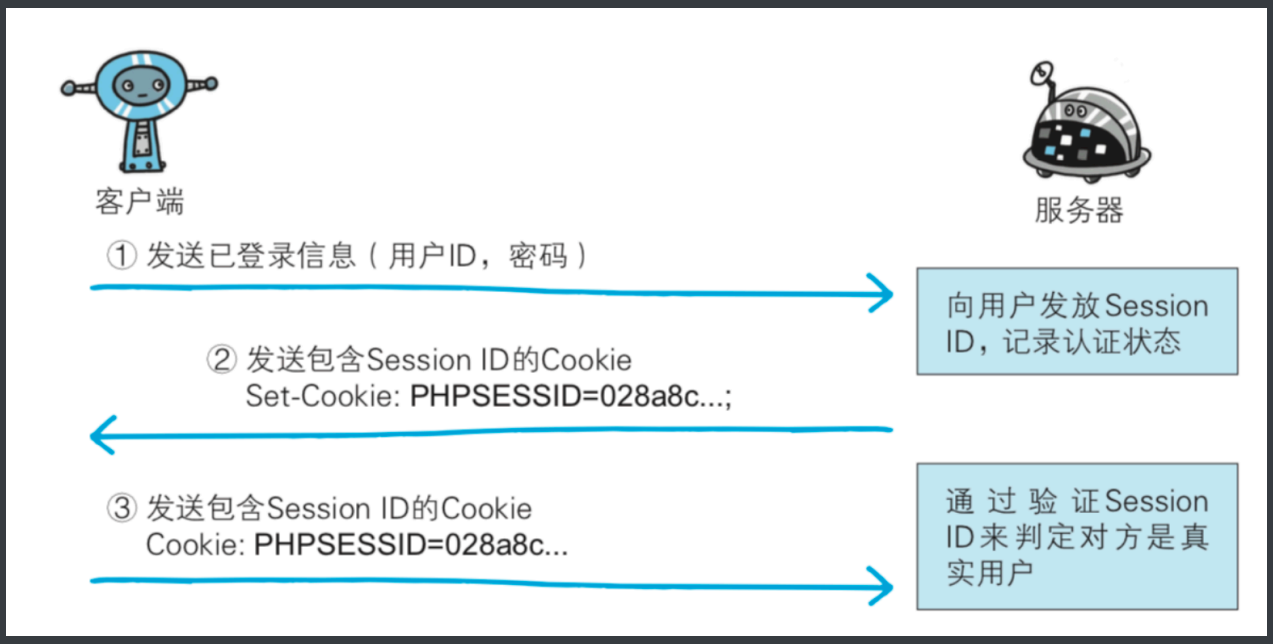
1 二次“登陆”导致的权限提升

1.1 “登陆”的实现

登陆功能应该是Web项目里见到的比较多的业务模块的，通常会跟session会话结合，完成对应的操作，常见的实现过程如下：



session本身是一个容器，里面可以存放类似用户身份等权限控制所需的元素，在对应的接口完成对应的鉴权功能。最简单的例如如果在当前会话session中没有找到对应的登陆凭证，说明用户没有登录或者登录失效，如果找到了证明用户已经登录可执行后面操作，防止接口的未授权访问。或者是权限细粒度覆盖业务接口，将业务相关的关键参数（例如userid、fileid等）与当前用户的身份凭证（一般是session）进行绑定，防止越权操作。

正常业务场景下，用户在进行身份认证后，便使用当前会话进行业务操作了，例如查询个人信息，进行下订单等。此时若**尝试使用当前会话，继续访问登陆认证接口进行二次“登陆”**，又可能会发生什么事情呢？重复登录又可能对session中的内容造成什么样的影响呢？

1.2 二次"登陆"带来的权限提升

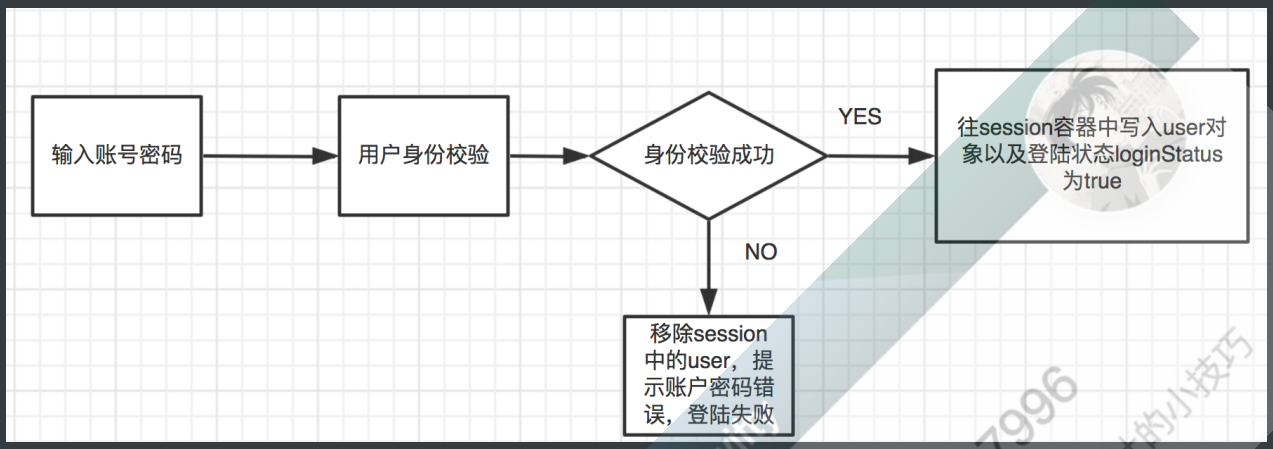
在某次审计过程中，发现了这样的一个有趣的安全问题，在进行身份认证后，**尝试再使用当前会话，继续访问登陆认证接口使用错误的账号密码进行二次“登陆”**，然后发现当前会话的角色权限提升了，部分管理员接口可以进行访问并进行业务操作了。以下是相关过程。

首先是登陆认证的接口：

```
@RequestMapping(value={"auth"}, method=RequestMethod.POST)
```

```
    @ResponseBody
    public JsonResponse LoginInterface(String username,String
    password,HttpServletRequest request,HttpSession session,Model model){
        if(username==null||username.toString().equals("")){
            return JsonResponse.fail("请输入用户名！");
        }
        if(password==null||password.toString().equals("")){
             return JsonResponse.fail("请输入密码！");
        }
        //查询用户
        SysUser user = null;
        try{
            user = userservice.find(username,password);
        }catch(Exception e){
            user = null;
        }
        if(user==null||user.isEmpty()){
            session.removeAttribute("user");
            return JsonResponse.fail("用户名密码错误！");
        }
        session.setAttribute("user",user);
        session.setAttribute("loginStatus",true);
        return JsonResponse.succ("success");
    }
```

首先检查用户关键输入（用户名密码）是否为null，然后通过service层方法进行用户密码的有效性检查，返回对应的user对象。如果返回的对象不为null那么在session中存入当前user对象，并且设置登录状态loginStatus为true。否则清空当前session中的user，提示用户名密码错误。简单的流程如下：



再来看一下相关权限控制的安全防护：

通过拦截器对于接口的访问进行控制，结合登陆成功后的loginStatus内容，防止在非登陆情况下进行未授权访问：

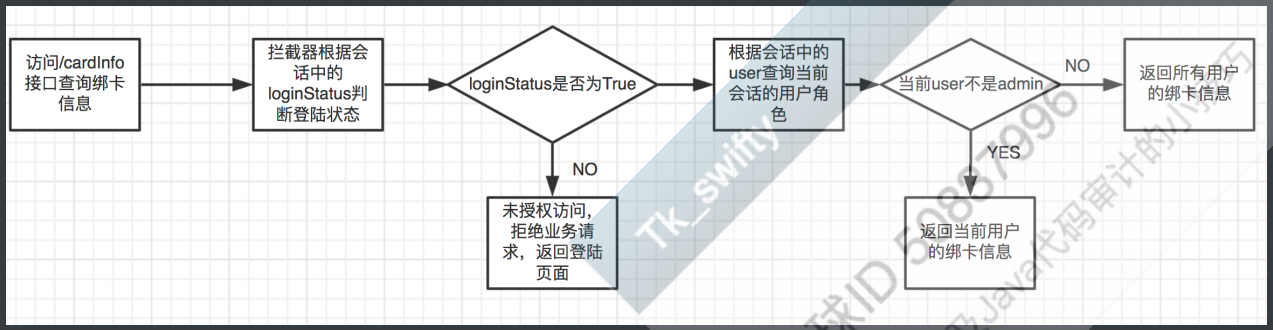
```
public class LoginedInterceptor extends HandlerInterceptorAdapter{
    @Override
    public boolean preHandle(HttpServletRequest request,HttpServletResponse
response,Object handler) throws Exception{
      HttpSession session = request.getSession();
      boolean loginStatus =
session.getAttribute(SystemBaseConstant.LOGIN_STATUS);
      if(loginStatus==null||"".equals(loginStatus)||!loginStatus){
            //如果登陆态loginStatus不为true，同样也返回登陆页面

 request.getRequestDispathcer("/login").forward(request,response);
            return false;
      }
    }
    return true;
}
```

查看下查询用户绑卡信息的接口：

```
@RequestMapping({"/cardInfo"})
public JsonResponse UserQuery(HttpSession session){
    User user = session.getAttribute("user");
    if(!user.getUserName.equals("admin")&&user!=null){
        CardInfo info = cardservice.findByUser(user);
    }else{
        CardInfo info = cardservice.findAll();
    }
    return JsonResponse.res(info);
}
```

可以看到这里跟管理员admin的查询接口是复用的。如果当前登陆用户不是admin，则调用service层的findByUser方法，返回当前用户绑定的卡号信息。否则返回所有用户的绑卡信息（admin管理员查询）。以查看下查询用户绑卡信息业务为例，相关流程如下：



这么梳理下来乍一看还是合理的，业务接口在非登陆状态下不可未授权访问，同时获取业务数据时候与当前会话的用户角色进行了绑定，防止通过类似userid=xxx的方式越权查看别人的卡号信息。

有一个关键点，在查询用户绑卡信息的接口，这里默认认为当前是可以从当前会话中取得到user，当user=null的时候，不满足if条件，此时直接查询返回所有用户的绑卡信息：

```
if(!user.getUserName.equals("admin")&&user!=null){
    CardInfo info = cardservice.findByUser(user);
}else{
    CardInfo info = cardservice.findAll();
}
```

那么追溯到啊user的初始化，是在登陆成功后存储到session容器中的：

```
    //查询用户
    SysUser user = null;
    try{
        user = userservice.find(username,password);
    }catch(Exception e){
        user = null;
    }
    if(user==null||user.isEmpty()){
        session.removeAttribute("user");
        return JsonResponse.fail("用户名密码错误！");
    }
    session.setAttribute("user",user);
    session.setAttribute("loginStatus",true);
```

因为登陆成功后，才会把user存储到session容器中，同时虽然非登陆状态时user为null，也满足直接查询返回所有用户的绑卡信息（admin权限）的条件，但是由于此时loginStatus为非登陆状态，在拦截器的作用下并不能满足接口业务的访问。这么一看逻辑好像没啥大问题。

这里整个过程都是跟登陆以及会话相关的，**上述的所有场景都是建立在一次普通用户登陆的场景下去讨论的**。那么如果二次“登陆”的情况下，会是怎么样一样场景呢。

首先假设以tkswifty用户进行登陆，当前会话cookie为：

```
JSESSIONID=19997B1355BFFF12CAD862232C273505
```

登陆成功后，此时session容器的存储内容为：

```
loginStatus=true
user=封装tkswifty用户信息的Bean
```

此时访问/cardInfo接口，应该是只能查询到tkswifty本身的卡号绑定信息的。

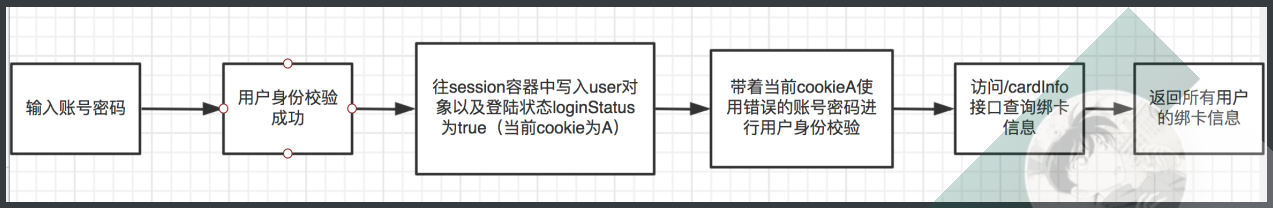
此时做如下操作，继续使用刚刚记录的会话cookie：JSESSIONID=19997B1355BFFF12CAD862232C273505，使用不存在的用户sec-in以及随意密码进行登录：

```
if(user==null||user.isEmpty()){
        session.removeAttribute("user");
        return JsonResponse.fail("用户名密码错误！");
    }
```

很明显此时调用userservice.find(username,password)返回的结果应该为null（没有sec-in这个账户信息），那么根据登陆逻辑，会默认把当前session中的user清除。那么此时session容器的存储内容为：

```
loginStatus=true
user=null
```

这里根据前面的分析，已经满足了直接查询返回所有用户的绑卡信息（admin权限）的条件，user为null，并且此时我们的账户状态loginStatus为true，拦截器认为这是一个登陆的合法请求。那么此时会话里的用户就处于一种游离态了，并且其已经达到了一个权限提升的效果，再次访问/cardInfo接口，此时应该会返回所有用户的绑卡信息了。大致的攻击利用流程如下：



1.3 修复及思考

整个过程还是比较“诡异”，在开发过程中并没有对上述的二次“登陆”场景考虑，结合种种条件，导致了越权问题。在业务开发过程中还是比较值得注意的：

- 非空判断的时机

上述问题很多地方均存在非空判断的逻辑，尤其是在查询接口/cardInfo中：

```
if(!user.getUserName.equals("admin")&&user!=null){
    CardInfo info = cardservice.findByUser(user);
}else{
    CardInfo info = cardservice.findAll();
}
```

这里非空判断的位置是值得考究的，如果是如下的逻辑，那么就不会存在上述的缺陷了：

```
if(user!=null){
    //业务处理
}else{
    //无法绑定对话属性，抛出异常
}
```

- session容器存储属性

session作为一个容器辅助进行权限校验是很常见的一种使用方式，但是属性是否冗余，不同情景下如何创建销毁同样也值得考量。

例如上述缺陷，loginStatus是否可以由user代替。如果每次登陆校验成功后均会有一个user对象一一对应，那么拦截其的代码修改如下，便可以达到防护效果：

```
User users = session.getAttribute("user");
if(user==null||user.isEmpty()){
      //如果登陆态loginStatus不为true，同样也返回登陆页面
      request.getRequestDispathcer("/login").forward(request,response);
      return false;
}
```

同时也避免了上述的游离态的用户，解决了对应的安全缺陷。

再者loginStatus可能业务需要，例如涉及到跨平台架构等无法由user代替，那么在多账号同一会话登陆时，若登陆失败也需及时修改当前会话的登陆认证状态。而不是仅仅清除用户user即可了：

```
if(user==null||user.isEmpty()){
        session.removeAttribute("user");
        session.removeAttribute("loginStatus");
        return JsonResponse.fail("用户名密码错误！");
    }
```

最后，在进行身份认证后，**尝试再使用当前会话，继续访问登陆认证接口使用错误的账号密码进行二次“登陆”**，这种业务场景的确很多时候在黑盒白盒测试中经常会遗漏，也是个挖掘和审计的思路，毕竟越复杂的设计、越多的参数，往往可能暗藏不少业务逻辑问题。