用户登录扩展

用户会话 session

session 对象主要存在服务器端，可以用于保存服务器的临时数据对象，所保存的数据可以在整个项目中都可以访问来获取，把 session 的数据看做一个共享的数据，首次登录的时候所获取的用户的数据，转移到 session 对象即可。获取 session 数据：session.getAttribute("key")可以获取 session 中的数据这种行为进行封装，封装在 BaseController 类中。

1、封装 session 对象数据的获取（封装父类中）、数据的设置（当用户登录成功后进行数据的设置，设置到全局的 session 对象）。

2、在父类中封装两个数据：获取 uid 和获取 username 对应的两个方法。用户头像暂不考虑，将来封装在 cookie 中使用。

```
/**
 * 获取 session 对象的 uid
 * @param session session 对象
 * @return 当前登录的用户的 uid 值
 */

protected final Integer geUidFromSession(HttpSession session){
    return Integer.valueOf(session.getAttribute("uid").toString());
}

/**
 * 获取 session 对象的 username
 * @param session session
 * @return 当前登录用户的用户名
 */
protected final String getUsernameFromSession(HttpSession session){
    return session.getAttribute("username").toString();
}
```

3、在登录的方法中将数据封装在 session 对象中，服务器本身会自动创建有 session 对象，已经是一个全局的 session 对象，SpringBoot 直接使用 session 对象，直接将 HttpSession 类型的对象作为请求处理方法的参数，会自动将全局的 session 对象注入到请求处理的方法 session 形参中。

在 UserController 中修改 login（）

```
@RequestMapping("login")
public JsonResult<User> login(String username, String password, HttpSession session){
    //调用业务对象的方法执行登录，并获取返回值
    User data = iUserService.login(username, password);
    //向 session 对象中完成用户数据的绑定（session 是全局的）
    session.setAttribute("uid",data.getUid());
    session.setAttribute("username",data.getUsername());
    //将以上返回值和状态码 ok 封装到响应结果中并返回
    return new JsonResult<User>(ok,data);
```

```
}
```

拦截器

拦截器：将所有的请求统一拦截到拦截器中，可以在拦截器中定义过滤的规则，如果不满足系统设置的过滤规则，统一的处理是重新打开 login.html 页面（重定向和转发），推荐使用重定向。在 Spring MVC 中拦截请求是通过处理器拦截器 HandlerInterceptor 来实现的，它拦截的目标是请求的地址。在 Spring MVC 中定义一个拦截器，需要实现 HandlerInterceptor 接口。

1 HandlerInterceptor

1.1 preHandle()方法

该方法将在请求处理之前被调用。SpringMVC 中的 Interceptor 是链式的调用，在一个应用或一个请求中可以同时存在多个 Interceptor。每个 Interceptor 的调用会依据它的声明顺序依次执行，而且最先执行的都是 Interceptor 中的 preHandle()方法，所以可以在这个方法中进行一些前置初始化操作或者是对当前请求的一个预处理，也可以在这个方法中进行一些判断来决定请求是否要继续进行下去。该方法的返回值是布尔值类型，当返回 false 时，表示请求结束，后续的 Interceptor 和 Controller 都不会再执行；当返回值 true 时，就会继续调用下一个 Interceptor 的 preHandle 方法，如果已经是最后一个 Interceptor 的时，就会调用当前请求的 Controller 方法。

1.2 postHandle()方法

该方法将在当前请求进行处理之后，也就是 Controller 方法调用之后执行，但是它会在 DispatcherServlet 进行视图返回渲染之前被调用，所以我们可以在这个方法中对 Controller 处理之后的 ModelAndView 对象进行操作。postHandle 方法被调用的方向跟 preHandle 是相反的，也就是说先声明的 Interceptor 的 postHandle 方法反而会后执行。如果当前 Interceptor 的 preHandle()方法返回值为 false，则此方法不会被调用。

1.3 afterCompletion()方法

该方法将在整个当前请求结束之后，也就是在 DispatcherServlet 渲染了对应的视图之后执行。这个方法的主要作用是用于进行资源清理工作。如果当前 Interceptor 的 preHandle()方法返回值为 false，则此方法不会被调用。

2 WebMvcConfigurer

在 SpringBoot 项目中，如果想要自定义一些 Interceptor、ViewResolver、MessageConverter，该如何实现呢？在 SpringBoot 1.5 版本都是靠重写 WebMvcConfigurerAdapter 类中的方法来添加自定义拦器、视图解析器、消息转换器等。而在 SpringBoot 2.0 版本之后，该类被标记为 @Deprecated。因此我们只能靠实现 WebMvcConfigurer 接口来实现。

WebMvcConfigurer 接口中的核心方法之一 addInterceptors(InterceptorRegistry registry)方法表示添加拦截器。主要用于进行用户登录状态的拦截，日志的拦截等。

addInterceptor：将自定义拦截器进行注册，需要一个实现 HandlerInterceptor 接口的拦截器实例

addPathPatterns：用于设置拦截器的过滤路径规则；addPathPatterns("/**")对所有请求都拦截

excludePathPatterns：用于设置不需要拦截的过滤规则

3 项目添加拦截器功能

1.分析：项目中很多操作都是需要先登录才可以执行的，如果在每个请求处理之前都编写代码检查 Session 中有没有登录信息，是不现实的。所以应使用拦截器解决该问题。
2.创建拦截器类 com.cy.store.interceptor.LoginInterceptor，并实现 org.springframework.web.servlet.HandlerInterceptor 接口。

```
package com.fx.store.interceptor;

import org.springframework.web.servlet.HandlerInterceptor;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

/**
 *
 */
public class LoginInterceptor implements HandlerInterceptor {
    /**
     * 检测全局 session 对象是否有 uid 数据，如果有则执行，如果没有重定向到登录
     * @param request 请求对象
     * @param response 响应对象
     * @param handler 处理器（url+Controller：映射）
     * @return 如果返回值为 true 表示放行当前的请求，如果为 false 则表示拦截当前的请求
     * @throws Exception
     */
    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {
        //HttpServletRequest 对象来获取 session 对象
        Object obj = request.getSession().getAttribute("uid");
        if (obj == null){
            //说明用户没有登录过系统，则重定向到 login.html 页面
            response.sendRedirect("/web/login.html");
            //结束后续的调用
            return false;
        }
        //请求放行
        return true;
    }
}
```

注册过滤器：添加白名单（哪些资源可以在不登陆的情况下访问：

login.html,register.html,index.html,product.html)，添加黑名单(在用户登录的状态下才可以访问的资源)。

注册过滤器的技术：借助 WebMvcConfigurer 接口，可以将用户定义的拦截器进行注册，才可以保证拦截器能生效和使用。定义一个类，然后让这个类实习 WebMvcConfigurer 接口。配置信息，建议存放在项目的 config 包下。

3.创建 LoginInterceptorConfigurer 拦截器的配置类并实现 org.springframework.web.servlet.config.annotation.WebMvcConfigurer 接口，配置类需要添加 @Configruation 注解修饰。

```
package com.fx.store.config;

import com.fx.store.interceptor.LoginInterceptor;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.HandlerInterceptor;
import org.springframework.web.servlet.config.annotation.InterceptorRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

import java.util.ArrayList;
import java.util.List;

/**
 * 处理拦截器的注册
 */
@Configuration//用于加载当前的拦截器
public class LoginIntercrepterConfigurer implements WebMvcConfigurer{

    /**
     * 配置拦截器
     * @param registry
     */
    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        //创建自定义拦截器的对象
        HandlerInterceptor interceptor = new LoginInterceptor();
        //配置白名单，存在一个 list 集合中
        List<String> patterns = new ArrayList<>();
        patterns.add("/bootstrap3/**");
        patterns.add("/css/**");
        patterns.add("/images/**");
        patterns.add("/js/**");
        patterns.add("/web/register.html");
        patterns.add("/web/login.html");
        patterns.add("/web/index.html");
        patterns.add("/web/product.html");
        patterns.add("/users/reg");
```

```
            patterns.add("/users/login");

            //完成拦截器的注册
            registry.addInterceptor(interceptor)
                    .addPathPatterns("/**")
                    .excludePathPatterns(patterns);//表示要拦截的 url 是什么
    }
}
```

运行项目启动类，分别输入白名单和黑名单地址查看是否拦截。
若浏览器提示重定向次数过多，login.html 页面无法打开，解决方式：将浏览器 cookie 请求清除，将浏览器初始化。

Cookie 和 session

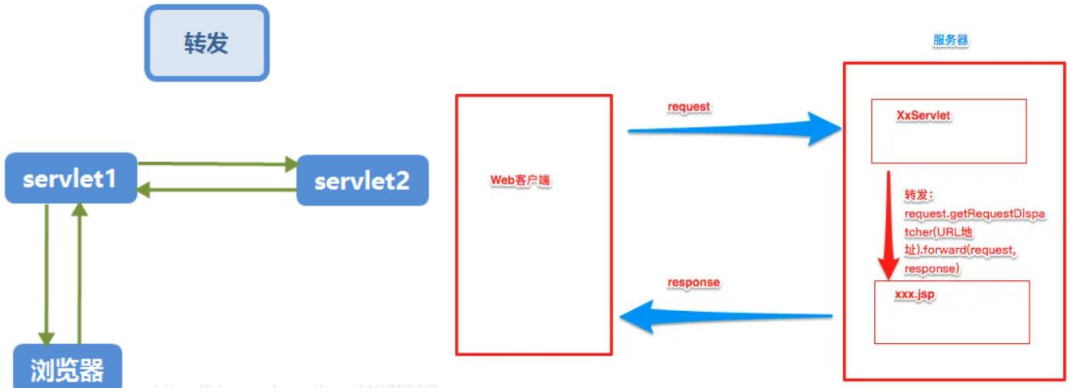
为什么要用 cookie：由于 http 协议是一种无状态的协议（客户端和服务端互相不认识）Cookies 是一些存储在用户电脑上的小文件。它是被设计用来保存一些站点的用户数据，这样能够让服务器为这样的用户定制内容。页面代码能够获取到 Cookie 值然后发送给服务器，比如 Cookie 中存储了所在地理位置，以后每次进入地图就可以默认定位到该地点。
cookie 的执行原理：就是当客户端访问服务器的时候（服务器运用了 cookie），服务器会生成一份 cookie 传输给客户端，客户端会自动把 cookie 保存起来，以后客户端每次访问服务器，都会自动的携带着这份 cookie。

Session 是另一种记录客户状态的机制，不同的是 Cookie 保存在客户端浏览器中，而 Session 保存在服务器上。客户端浏览器访问服务器的时候，服务器把客户端信息以某种形式记录在服务器上。这就是 Session。客户端浏览器再次访问时只需要从该 Session 中查找该客户的状态就可以了 session 是一种特殊的 cookie。cookie 是保存在客户端的，而 session 是保存在服务端。
session 原理：当客户端第一次请求服务器的时候，服务器生成一份 session 保存在服务端，将该数据(session)的 id 以 cookie 的形式传递给客户端；以后的每次请求，浏览器都会自动的携带 cookie 来访问服务器(session 数据 id)。

重定向和转发

转发：是指浏览器发送请求到 servlet1 之后，servlet1 需要访问 servlet2，因此在服务器内部跳转到的 servlet2，转发有时也称为服务器内跳转。整个过程浏览器只发出一次请求，服务器只发出一次响应。所以，无论是 servlet1 还是 servlet2，整个过程中，只存在一次请求，即用户所提交的请求。因此 servlet1 和 servlet2 均可从这个请求中获取到用户提交请求时所携带的相关数据。

举例：你在专卖店选中一双篮球鞋，店员告诉你这双鞋现在没有了，让你稍等下，他去另外一个专卖店去取，店员取到鞋之后把鞋放到你手中。



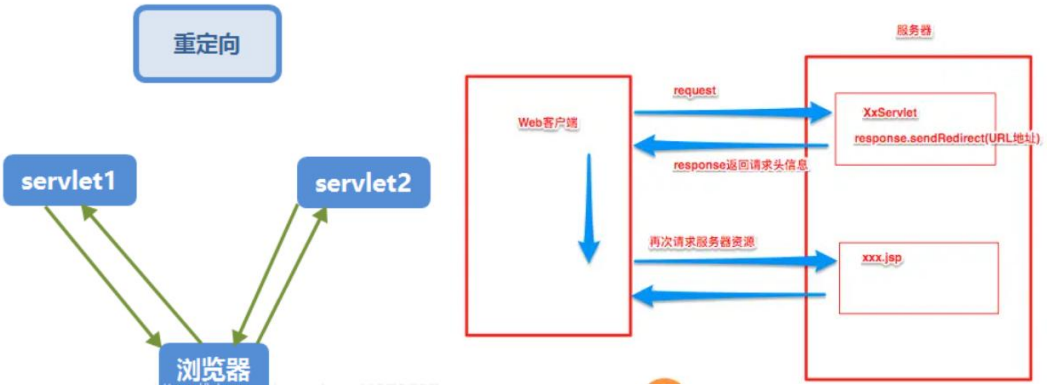
重定向：是浏览器发送请求到 servlet1 之后，servlet1 需要访问 servlet2，但并未在服务器内直接访问，而是由服务器自动向浏览器发送一个响应，浏览器再自动提交一个新的请求，这个请求就是对 servlet2 的请求。

对于 servlet2 的访问，是先由服务器响应客户端浏览器，再由客户端浏览器向服务器发送对 servlet2 的请求，所以重定向有时又称为服务器外跳转。

整个过程中，浏览器共提交了两次请求，服务器共发送了两次响应。只不过，第一次响应与第二次请求，对于用户来说是透明的，是感知不到的。用户认为，自己只提交了一次请求，且只收到了一次响应。

这样的话，就会有一个问题：servlet2 中是无法获取到用户手动提交请求中的数据的，它只能获取到第二次请求中所携带的数据。

举例：你在专卖店选中一双篮球鞋，店员告诉你这双鞋现在没有了，附近的另一个专卖店中有这双鞋，他把那个专卖店的地址告诉你，你得到地址之后，自己到达另外一个专卖店买到了这双鞋。


重定向
servlet1
servlet2
浏览器
服务器
Web客户端
request
XxServlet
response.sendRedirect(URL地址)
response返回请求头信息
再次请求服务器资源
xxx.jsp