---

1. 动态秒杀地址

1.1 前端的改变

之前我们实现秒杀的时候是直接跳转到秒杀接口，使得我们每次的秒杀地址都是一样的，这样具有安全隐患，所以，我们将其改为动态地址，通过在前端上写一个方法进行跳转，如下所示。

- 它会先跳转到 `/miaosha/path` ， 获取秒杀地址中的 `path值` ， 将其存储在Redis中

```
function getMiaoshaPath(){
    var goodsId = $("#goodsId").val();
    g_showLoading();
    $.ajax({
        url:"/miaosha/path",
        type:"GET",
        data:{
            goodsId:goodsId,
            verifyCode:$("#verifyCode").val()
        },
        success:function(data){
            if(data.code == 0){
                var path = data.data;
                doMiaosha(path);
            }else{
                layer.msg(data.msg);
            }
        },
        error:function(){
            layer.msg("客户端请求有误");
        }
    });
}
```


- 然后携带 `path值` 去访问真正的秒杀方法，在其中将 `path` 值与Redis中的值进行比较，一致才能继续秒杀

```
function doMiaosha(path){
    $.ajax({
        url:"/miaosha/"+path+"/do_miaosha",
        type:"POST",
        data:{
            goodsId:$("#goodsId").val()
        },
        success:function(data){
            if(data.code == 0){
                //window.location.href="/order_detail.htm?orderId="+data.data.id;
                getMiaoshaResult($("#goodsId").val());
            }else{
                layer.msg(data.msg);
            }
        },
        error:function(){
            layer.msg("客户端请求有误");
        }
    });

}
```


1.2 获取路径的Java代码

```
    @ResponseBody
    @RequestMapping(value = "/path",method = RequestMethod.GET)
    public Result<String> getMiaoshaPath(MiaoShaUser user,@RequestParam("goodsId")long go
                                         @RequestParam(value = "verifyCode",defaultValue
        if(user == null)
            return Result.error(CodeMsg.SESSION_ERROR);


        String path = miaoshaService.createMiaoshaPath(user,goodsId);

        return Result.success(path);
    }
```

- 先调用createMiaoshaPath()方法，在其中会创建一串随机值，并且存储到Redis中，具体方法如下，执行完之后将路径值返回到前端

```
    public String createMiaoshaPath(MiaoShaUser user, long goodsId) {
        if(user == null || goodsId <= 0)
            return null;
```

```
4
5           String str = MD5Util.md5(UUIDUtil.getUUID());
6           redisService.set(MiaoshaKey.miaoshaPathPrefix,user.getId() + "_" + goodsId,str);
7
8           return str;
9       }
```

1.3 执行秒杀接口的修改

```
 */
@RequestMapping(value = "/{path}/do_miaosha", method = RequestMethod.POST)
@ResponseBody
public Result<Integer> doMiaosha(Model model, MiaoShaUser user,
                                 @RequestParam("goodsId")long goodsId, @PathVariable(value = "path")String path){
    //user不能为空，空了去登陆
    if(user == null){
        return Result.error(CodeMsg.SESSION_ERROR);
    }

    boolean check = miaoshaService.checkPath(user.getId(),goodsId,path);
    if( ! check)
        return Result.error(CodeMsg.REQUEST_ILLEGAL);
```


- 路径上，我们采用了RestFul风格，通过@PathVariable注解获取其中的路径值，并与redis服务器中的值进行比较，一致才能向下一步继续执行

---

2. 添加验证码验证

我们在立即秒杀按钮处添加验证码， 防止机器人对我们的系统进行多次秒杀 ， 也可以使秒杀能够 错峰访问 ， 削减并发量，我们采用的是 ScriptEngine

2.1 实现过程

1. 首先，我们在路径获取中，添加了对验证码验证的步骤

```
@ResponseBody
@RequestMapping(value = "/path",method = RequestMethod.GET)
public Result<String> getMiaoshaPath(MiaoShaUser user,@RequestParam("goodsId")long goodsId,
                                     @RequestParam(value = "verifyCode",defaultValue = "0")int verifyCode){
    if(user == null)
        return Result.error(CodeMsg.SESSION_ERROR);

    boolean check = miaoshaService.checkVerifyCode(user,goodsId,verifyCode);
    if(! check)
        return Result.error(CodeMsg.REQUEST_ILLEGAL);

    String path = miaoshaService.createMiaoshaPath(user,goodsId);

    return Result.success(path);
}
```


在该方法中，实现的是将从前端获取的验证码与Redis存储的验证码进行验证，验证完成之后，就将它从Redis中移除，方法代码如下

```
public boolean checkVerifyCode(MiaoShaUser user, long goodsId, int verifyCode) {

    if(user == null || goodsId <=0) {
        return false;
    }
    Integer codeOld = redisService.get(MiaoshaKey.miaoshaVerifyCodePrefix, key: user.getId()+","+goodsId, Integer.class);
    if(codeOld == null || codeOld - verifyCode != 0 ) {进行判断
        return false;
    }
    redisService.delete(MiaoshaKey.miaoshaVerifyCodePrefix, id: user.getId()+","+goodsId);
    return true;
}
```

从Redis中移除正确的验证码值

2. 在此之前，前端验证码会和后端有一个响应，每次刷新验证码都会将其的正确结果同步到服务器的Redis上

```java
@RequestMapping(value = "/verifyCode",method = RequestMethod.GET)
@ResponseBody
public Result<String> getMiaoshaVerifyCode(HttpServletResponse response,MiaoShaUser user,
                                           @RequestParam("goodsId")long goodsId){
    if(user == null)
        return Result.error(CodeMsg.SESSION_ERROR);

    try {
        BufferedImage image  = miaoshaService.createVerifyCode(user, goodsId);
        OutputStream out = response.getOutputStream();
        ImageIO.write(image, formatName: "JPEG", out);
        out.flush();
        out.close();
        return null;
    }catch(Exception e) {
        e.printStackTrace();
        return Result.error(CodeMsg.MIAOSHA_FAIL);
    }

}
```


---

3. 接口限流防刷

- 接口限流防刷的作用是在规定的时间内访问固定的次数。我们实现的思路是，在要限制防刷的方法上添加注解，通过拦截器进行限制访问次数

3.1 创建出这个注解

该注解中，包含了需要访问时间内的访问次数，以及判断是否需要登录

```java
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface AccessLimit {
    int seconds();
    int maxCount();
    boolean needLogin() default true;
}
```

- `@Retention(RetentionPolicy.RUNTIME)`：注解不仅被保存到class文件中，jvm加载class文件之后，仍然存在；
- `@Target(ElementType.METHOD)`： 表示注解修饰的是方法

对我们想要限流的方法进行标记

```
@AccessLimit(seconds = 5,maxCount = 5,needLogin = true)
@ResponseBody
@RequestMapping(value = "/path",method = RequestMethod.GET)
public Result<String> getMiaoshaPath(MiaoShaUser user,@RequestParam("goodsId")long goodsId,
                                     @RequestParam(value = "verifyCode",defaultValue = "0")int verifyCode){
    if(user == null)
        return Result.error(CodeMsg.SESSION_ERROR);

    boolean check = miaoshaService.checkVerifyCode(user,goodsId,verifyCode);
    if(! check)
        return Result.error(CodeMsg.YAN_ZHENG_MA_CUO_WU);

    String path = miaoshaService.createMiaoshaPath(user,goodsId);

    return Result.success(path);
}
```


3.2 创建拦截器

```
public class AccessInterceptor extends HandlerInterceptorAdapter {

    @Autowired
    MiaoShaUserService userService;
    @Autowired
    RedisService redisService;

    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Ob

        if(handler instanceof HandlerMethod){
            MiaoShaUser user  = getUser(request,response);
            UserContext.setUser(user);
            HandlerMethod hm = (HandlerMethod) handler;
            //处理方法的对象，获取的是方法的注解
            AccessLimit accessLimit = hm.getMethodAnnotation(AccessLimit.class);
            if(accessLimit == null){
                return false;
            }
            int seconds = accessLimit.seconds();
            int maxCount = accessLimit.maxCount();
            boolean needLogin = accessLimit.needLogin();
            String key = request.getRequestURI();//获取请求的地址
            if (needLogin) {
                if(user == null){
                    //user为空，递交错误信息
                    render(response, CodeMsg.SESSION_ERROR);
```

```
                    return false;
                }
                key += "_" + user.getId();
            }
            AccessKey accessKey = AccessKey.withExpireSecond(seconds);
            Integer count = redisService.get(accessKey, key, Integer.class);
            if(count == null){
                redisService.set(accessKey,key,1);
            }else if(count < maxCount){
                redisService.incr(accessKey,key);
            }else{
                render(response,CodeMsg.ACCESS_LIMIT_REACHED);
                return false;
            }
        }
        return true;
    }
        ......
}
```

- 继承 `HandlerInterceptorAdapter` ， 重写 `preHandle` 方法
- 重要的UserContext

```
@Override
public boolean preHandle(HttpServletRequest request, HttpServletResponse respons

    if(handler instanceof HandlerMethod){
        MiaoShaUser user  = getUser(request,response);
        UserContext.setUser(user);//将user存起来,存在！当前！线程中
        HandlerMethod hm = (HandlerMethod) handler;
        //处理方法的对象，获取的是方法的注解
        AccessLimit accessLimit = hm.getMethodAnnotation(AccessLimit.class);
        if(accessLimit == null){
            return false;
        }
```


我们看一下具体的实现

```
public class UserContext {

    //用ThreadLocal来装user信息，调用它的set和get方法，向其中存储值
    //ThreadLocal是为当前线程存储值，所以，在多线程下，各个线程的user并不冲突
    private static ThreadLocal<MiaoShaUser> userHolder = new ThreadLocal<>();

    public static void setUser(MiaoShaUser user){
```

```
8           userHolder.set(user);
9       }
10
11      public static MiaoShaUser getUser(){
12          return userHolder.get();
13      }
14  }
```

其中ThreadLocal()源码如下

```
*/
public void set(T value) {
    Thread t = Thread.currentThread();
    ThreadLocalMap map = getMap(t);
    if (map != null)
        map.set(this, value);
    else
        createMap(t, value);
}
```


3.3 后序步骤解释

方法后边比较简单啦

```
if(handler instanceof HandlerMethod){
    MiaoShaUser user  = getUser(request,response);
    UserContext.setUser(user);//将user存起来,存在！当前！线程中
    HandlerMethod hm = (HandlerMethod) handler;
    //处理方法的对象，获取的是方法的注解
    AccessLimit accessLimit = hm.getMethodAnnotation(AccessLimit.class);
    if(accessLimit == null){
        return false;
    }
    int seconds = accessLimit.seconds();
    int maxCount = accessLimit.maxCount();
    boolean needLogin = accessLimit.needLogin();
    String key = request.getRequestURI();//获取请求的地址
    if (needLogin) {
        if(user == null){
            //user为空，递交错误信息
            render(response, CodeMsg.SESSION_ERROR);
            return false;
        }
        key += "_" + user.getId();
    }
    AccessKey accessKey = AccessKey.withExpireSecond(seconds);
    Integer count = redisService.get(accessKey, key, Integer.class);
    if(count == null){
        redisService.set(accessKey,key, value: 1);
    }else if(count < maxCount){
        redisService.incr(accessKey,key);
    }else{
        render(response,CodeMsg.ACCESS_LIMIT_REACHED);
        return false;
    }
}
return true;
```

通过该方法来获得方法上的注解

这里应该写成true，没有被该注解标记的全部放行才对。
这里对应文章最后的配置说明

第一次访问的时候，向redis中存储值，
key为目标地址和用户id，value为访问次数，并有过期时间
每次访问都将该值与访问的限制最大值进行比对，超过规定的次数返回错误信息

3.4 切莫忘记配置，不配置约等于不加拦截器

```
@Configuration
public class WebConfig extends WebMvcConfigurerAdapter{

    @Autowired
    UserArgumentResolver userArgumentResolver;
    @Autowired
    AccessInterceptor accessInterceptor;

    @Override
    public void addArgumentResolvers(List<HandlerMethodArgumentResolver> argumentResolver
        super.addArgumentResolvers(argumentResolvers);
        argumentResolvers.add(userArgumentResolver);
    }

    @Override
```

```
    public void addInterceptors(InterceptorRegistry registry) {
        InterceptorRegistration interceptorRegistration = registry.addInterceptor(accessI
        interceptorRegistration.addPathPatterns("/miaosha/path");


    }
}
```

```
@Configuration
public class WebConfig extends WebMvcConfigurerAdapter{

    @Autowired
    UserArgumentResolver userArgumentResolver;
    @Autowired
    AccessInterceptor accessInterceptor;

    @Override
    public void addArgumentResolvers(List<HandlerMethodArgumentResolver> argumentResolvers) {
        super.addArgumentResolvers(argumentResolvers);
        argumentResolvers.add(userArgumentResolver);
    }

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        InterceptorRegistration interceptorRegistration = registry.addInterceptor(accessInterceptor);
        interceptorRegistration.addPathPatterns("/miaosha/path");

    }
}
```

在这个配置类中，我们重写的是addInterceptors方法，将拦截器注入进来，加到配置中，(指定要拦截的地址这一步可以省略掉了，因为我们使用的是注解标记，前边有一处写错，开始写的是没有注解的话，返回false，这样全局都被拦截了，应该写成true，这样才能放行），接下来就可以使用了！