1 异步调用

1.1 发送异步请求（回顾）

```
<a href="javascript:void(0);" id="testAjax">访问controller</a>
<script type="text/javascript" src="/js/jquery-3.3.1.min.js"></script>
<script type="text/javascript">
    $(function(){
    $("#testAjax").click(function(){ //为id="testAjax"的组件绑定点击事件
        $.ajax({ //发送异步调用
            type:"POST", //请求方式： POST请求
            url:"ajaxController", //请求参数（也就是请求内容）
            data:'ajax message', //请求参数（也就是请求内容）
            dataType:"text", //响应正文类型
            contentType:"application/text", //请求正文的MIME类型
        });
    });
});
</script>
```

1.2 接受异步请求参数

- 名称： @RequestBody
- 类型： 形参注解
- 位置：处理器类中的方法形参前方
- 作用：将异步提交数据组织成标准请求参数格式，并赋值给形参
- 范例：

```
@RequestMapping("/ajaxController")
public String ajaxController(@RequestBody String message){
    System.out.println(message);
    return "page.jsp";
}
```

- 注解添加到Pojo参数前方时，封装的异步提交数据按照Pojo的属性格式进行关系映射
- 注解添加到集合参数前方时，封装的异步提交数据按照集合的存储结构进行关系映射

```
@RequestMapping("/ajaxPojoToController")
//如果处理参数是POJO，且页面发送的请求数据格式与POJO中的属性对应，@RequestBody注解可以自动映射对应请求数据到POJO中
//注意：POJO中的属性如果请求数据中没有，属性值为null，POJO中没有的属性如果请求数据中有，不进行映射
public String  ajaxPojoToController(@RequestBody User user){
    System.out.println("controller pojo :"+user);
    return "page.jsp";
}

@RequestMapping("/ajaxListToController")
//如果处理参数是List集合且封装了POJO，且页面发送的数据是JSON格式的对象数组，数据将自动映射到集合参数中
```

```
public String  ajaxListToController(@RequestBody List<User> userList){
    System.out.println("controller list :"+userList);
    return "page.jsp";
}
```

1.3 异步请求接受响应数据

- 方法返回值为Pojo时，自动封装数据成json对象数据

```
@RequestMapping("/ajaxReturnJson")
@ResponseBody
public User ajaxReturnJson(){
    System.out.println("controller return json pojo...");
    User user = new User();
    user.setName("Jockme");
    user.setAge(40);
    return user;
}
```

- 方法返回值为List时，自动封装数据成json对象数组数据

```
@RequestMapping("/ajaxReturnJsonList")
@ResponseBody
//基于jackon技术，使用@ResponseBody注解可以将返回的保存POJO对象的集合转成json数组格式数据
public List ajaxReturnJsonList(){
    System.out.println("controller return json list...");
    User user1 = new User();
    user1.setName("Tom");
    user1.setAge(3);

    User user2 = new User();
    user2.setName("Jerry");
    user2.setAge(5);

    ArrayList al = new ArrayList();
    al.add(user1);
    al.add(user2);

    return al;
}
```

2 异步请求-跨域访问

2.1 跨域访问介绍

- 当通过域名A下的操作访问域名B下的资源时，称为跨域访问
- 跨域访问时，会出现无法访问的现象

2.2 跨域环境搭建

- 为当前主机添加备用域名
  - 修改windows安装目录中的host文件
  - 格式： ip 域名
- 动态刷新DNS
  - 命令： ipconfig /displaydns
  - 命令： ipconfig /flushdns

2.3 跨域访问支持

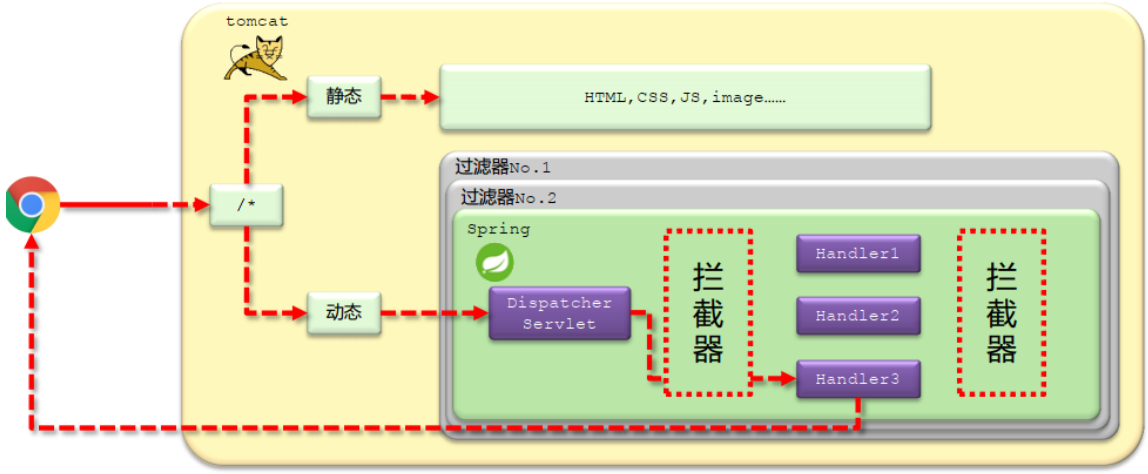
□ 名称： @CrossOrigin
□ 类型： 方法注解 、 类注解
□ 位置：处理器类中的方法上方 或 类上方
□ 作用：设置当前处理器方法/处理器类中所有方法支持跨域访问
□ 范例：

```
@RequestMapping("/cross")
@ResponseBody
//使用@CrossOrigin开启跨域访问
//标注在处理器方法上方表示该方法支持跨域访问
//标注在处理器类上方表示该处理器类中的所有处理器方法均支持跨域访问
@CrossOrigin
public User cross(HttpServletRequest request){
    System.out.println("controller cross..."+request.getRequestURL());
    User user = new User();
    user.setName("Jockme");
    user.setAge(39);
    return user;
}
```

3 拦截器

3.1 拦截器概念

- 请求处理过程解析

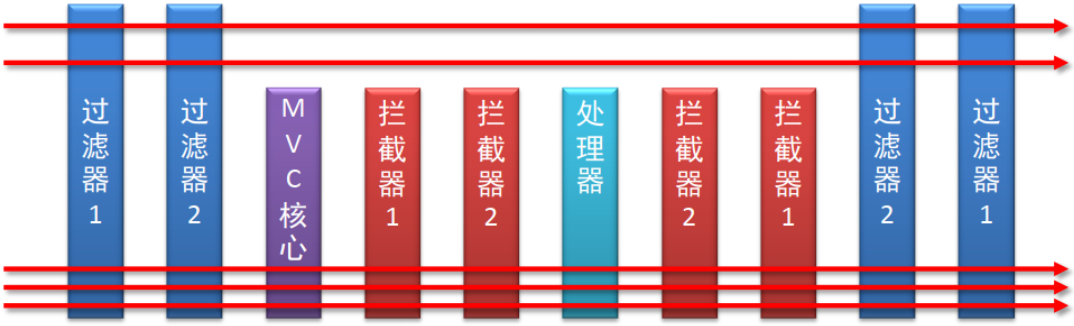


□ 拦截器（Interceptor）是一种动态拦截方法调用的机制
□ 作用：

1. 在指定的方法调用前后执行预先设定后的的代码
2. 阻止原始方法的执行

□ 核心原理： AOP思想
□ 拦截器链：多个拦截器按照一定的顺序，对原始被调用功能进行增强

- **拦截器VS过滤器**
  □ 归属不同： Filter属于Servlet技术， Interceptor属于SpringMVC技术
  □ 拦截内容不同： Filter对所有访问进行增强， Interceptor仅针对SpringMVC的访问进行增强



3.2 自定义拦截器开发过程

- 实现HandlerInterceptor接口

```
//自定义拦截器需要实现HandleInterceptor接口
public class MyInterceptor implements HandlerInterceptor {
    //处理器运行之前执行
    @Override
    public boolean preHandle(HttpServletRequest request,
                             HttpServletResponse response,
                             Object handler) throws Exception {
        System.out.println("前置运行----a1");
        //返回值为false将拦截原始处理器的运行
        //如果配置多拦截器，返回值为false将终止当前拦截器后面配置的拦截器的运行
        return true;
    }

    //处理器运行之后执行
```

```
    @Override
    public void postHandle(HttpServletRequest request,
                           HttpServletResponse response,
                           Object handler,
                           ModelAndView modelAndView) throws Exception {
        System.out.println("后置运行----b1");
    }

    //所有拦截器的后置执行全部结束后，执行该操作
    @Override
    public void afterCompletion(HttpServletRequest request,
                                HttpServletResponse response,
                                Object handler,
                                Exception ex) throws Exception {
        System.out.println("完成运行----c1");
    }

    //三个方法的运行顺序为     preHandle -> postHandle -> afterCompletion
    //如果preHandle返回值为false，三个方法仅运行preHandle
}
```

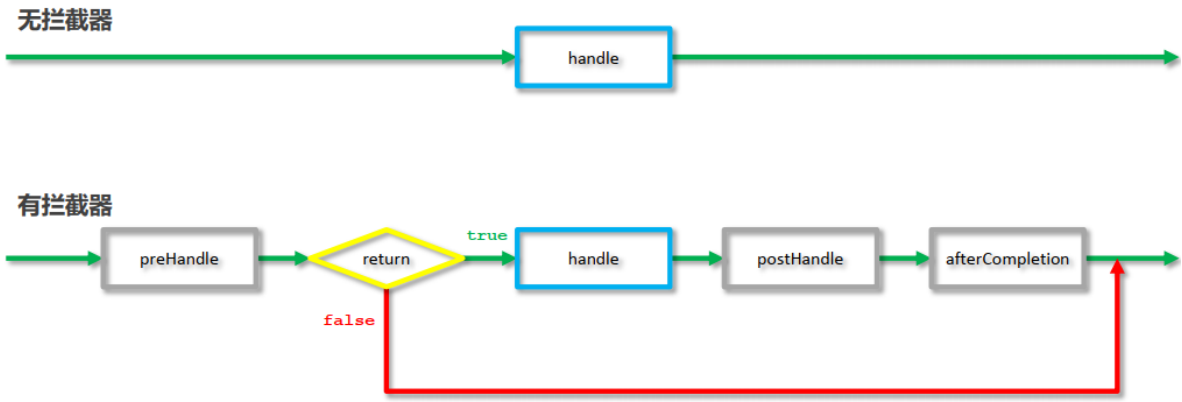
- 配置拦截器

  配置拦截器

```
<mvc:interceptors>
    <mvc:interceptor>
        <mvc:mapping path="/showPage"/>
        <bean class="com.itheima.interceptor.MyInterceptor"/>
    </mvc:interceptor>
</mvc:interceptors>
```

注意：配置顺序为先配置执行位置，后配置执行类

3.3 拦截器执行流程



3.4 拦截器配置与方法参数

3.4.1 前置处理方法

原始方法之前运行

```
public boolean preHandle(HttpServletRequest request,
                         HttpServletResponse response,
                         Object handler) throws Exception {
    System.out.println("preHandle");
    return true;
}
```

- 参数
  - request:请求对象
  - response:响应对象
  - handler:被调用的处理器对象，本质上是一个方法对象，对反射中的Method对象进行了再包装
- 返回值
  - 返回值为false，被拦截的处理器将不执行

3.4.2 后置处理方法

原始方法运行后运行，如果原始方法被拦截，则不执行

```
public void postHandle(HttpServletRequest request,
                       HttpServletResponse response,
                       Object handler,
                       ModelAndView modelAndView) throws Exception {
    System.out.println("postHandle");
}
```

- 参数
- modelAndView:如果处理器执行完成具有返回结果，可以读取到对应数据与页面信息，并进行调整

3.4.3 完成处理方法

拦截器最后执行的方法，无论原始方法是否执行

```
public void afterCompletion(HttpServletRequest request,
                            HttpServletResponse response,
                            Object handler,
                            Exception ex) throws Exception {
    System.out.println("afterCompletion");
}
```

- 参数
- ex:如果处理器执行过程中出现异常对象，可以针对异常情况进行单独处理

3.5 拦截器配置项

```
<mvc:interceptors>
    <!--开启具体的拦截器的使用，可以配置多个-->
    <mvc:interceptor>
        <!--设置拦截器的拦截路径，支持*通配-->
        <!--/**          表示拦截所有映射-->
        <!--/*           表示拦截所有/开头的映射-->
        <!--/user/*      表示拦截所有/user/开头的映射-->
        <!--/user/add*   表示拦截所有/user/开头，且具体映射名称以add开头的映射-->
```

```
        <!--/user/*All  表示拦截所有/user/开头，且具体映射名称以All结尾的映射-->
        <mvc:mapping path="/*"/>
        <mvc:mapping path="/**"/>
        <mvc:mapping path="/handleRun*"/>
        <!--设置拦截排除的路径，配置/**或/*，达到快速配置的目的-->
        <mvc:exclude-mapping path="/b*"/>
        <!--指定具体的拦截器类-->
        <bean class="MyInterceptor"/>
    </mvc:interceptor>
</mvc:interceptors>
```

3.6 多拦截器配置

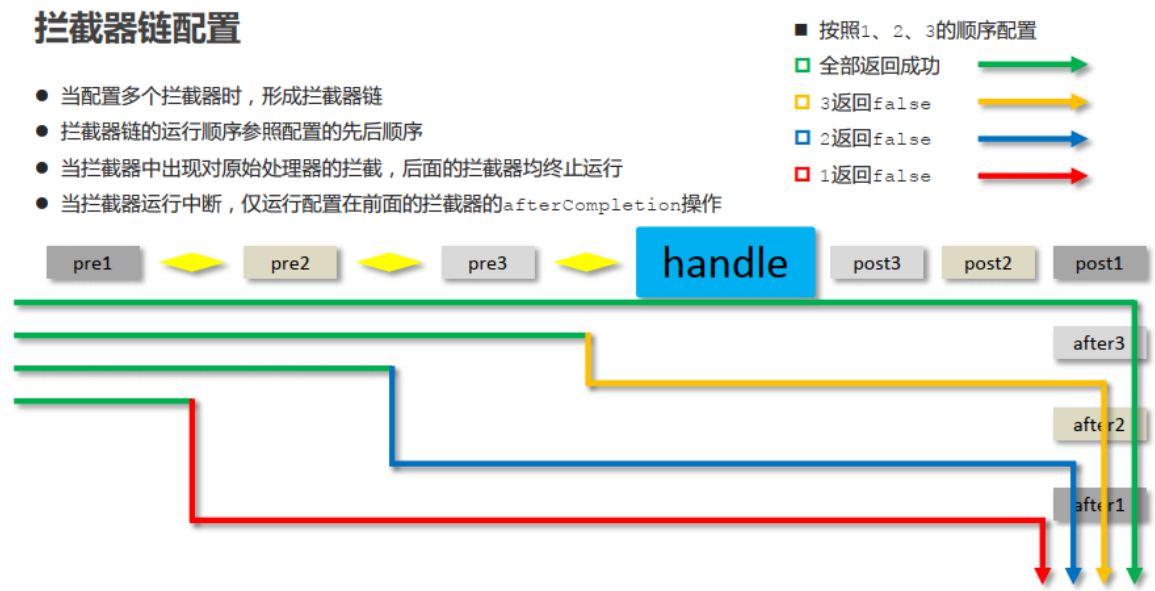


image-20210719223216050第三个拦截器前置返回false时,前两个执行情况

image-20210719223332607第二个拦截器前置返回false

image-20210719223401024第一个拦截器前置返回false,后面两个拦截器没有执行

责任链模式

□ 责任链模式是一种行为模式

□ 特征：

沿着一条预先设定的任务链顺序执行，每个节点具有独立的工作任务

□ 优势：

独立性：只关注当前节点的任务，对其他任务直接放行到下一节点

隔离性：具备链式传递特征，无需知晓整体链路结构，只需等待请求到达后进行处理即可

灵活性：可以任意修改链路结构动态新增或删减整体链路责任

解耦：将动态任务与原始任务解耦

□ 弊端：

链路过长时，处理效率低下

可能存在节点上的循环引用现象，造成死循环，导致系统崩溃

4 异常处理

4.1 异常处理器

HandlerExceptionResolver接口（异常处理器）

```
@Component
public class ExceptionResolver implements HandlerExceptionResolver {
    public ModelAndView resolveException(HttpServletRequest request,
                                         HttpServletResponse response,
                                         Object handler,
                                         Exception ex) {
        System.out.println("异常处理器正在执行中");
        ModelAndView modelAndView = new ModelAndView();
        //定义异常现象出现后，反馈给用户查看的信息
        modelAndView.addObject("msg","出错啦！ ");
        //定义异常现象出现后，反馈给用户查看的页面
        modelAndView.setViewName("error.jsp");
        return modelAndView;
    }
}
```

根据异常的种类不同，进行分门别类的管理，返回不同的信息

```
public class ExceptionResolver implements HandlerExceptionResolver {
    @Override
    public ModelAndView resolveException(HttpServletRequest request,
                                         HttpServletResponse response,
                                         Object handler,
                                         Exception ex) {
        System.out.println("my exception is running ...."+ex);
        ModelAndView modelAndView = new ModelAndView();
        if( ex instanceof NullPointerException){
            modelAndView.addObject("msg","空指针异常");
        }else if ( ex instanceof  ArithmeticException){
            modelAndView.addObject("msg","算数运算异常");
        }else{
            modelAndView.addObject("msg","未知的异常");
        }
        modelAndView.setViewName("error.jsp");
        return modelAndView;
    }
}
```

4.2 注解开发异常处理器

- 使用注解实现异常分类管理
  - 名称： @ControllerAdvice
  - 类型： 类注解
  - 位置：异常处理器类上方
  - 作用：设置当前类为异常处理器类
  - 范例：

```
@Component
@ControllerAdvice
public class ExceptionAdvice {
}
```

- 使用注解实现异常分类管理
  - □ 名称： @ExceptionHandler
  - □ 类型： 方法注解
  - □ 位置：异常处理器类中针对指定异常进行处理的方法上方
  - □ 作用：设置指定异常的处理方式
  - □ 范例：
  - □ 说明：处理器方法可以设定多个

```
@ExceptionHandler(Exception.class)
@ResponseBody
public String doOtherException(Exception ex){
    return "出错啦，请联系管理员！ ";
}
```

4.3 异常处理解决方案

- 异常处理方案
  - 业务异常：
    - □ 发送对应消息传递给用户，提醒规范操作
  - 系统异常：
    - □ 发送固定消息传递给用户，安抚用户
    - □ 发送特定消息给运维人员，提醒维护
    - □ 记录日志
  - 其他异常：
    - □ 发送固定消息传递给用户，安抚用户
    - □ 发送特定消息给编程人员，提醒维护
    - □ 纳入预期范围内
    - □ 记录日志

4.4 自定义异常

- 异常定义格式

```
//自定义异常继承RuntimeException，覆盖父类所有的构造方法
public class BusinessException extends RuntimeException {
    public BusinessException() {
    }

    public BusinessException(String message) {
        super(message);
    }

    public BusinessException(String message, Throwable cause) {
        super(message, cause);
    }

    public BusinessException(Throwable cause) {
```

```
        super(cause);
    }

    public BusinessException(String message, Throwable cause, boolean
enableSuppression, boolean writableStackTrace) {
        super(message, cause, enableSuppression, writableStackTrace);
    }
}
```

- 异常触发方式

```
if(user.getName().trim().length()<4) {
    throw new BusinessException("用户名长度必须在2-4位之间，请重新输入！ ");
}
```

- 通过自定义异常将所有的异常现象进行分类管理，以统一的格式对外呈现异常消息

5 实用技术

5.1 文件上传下载

- 上传文件过程分析

- MultipartResolver接口
  - MultipartResolver接口定义了文件上传过程中的相关操作，并对通用性操作进行了封装
  - MultipartResolver接口底层实现类CommonsMultipartResovler
  - CommonsMultipartResovler并未自主实现文件上传下载对应的功能，而是调用了apache的文件上传下载组件

```
<dependency>
    <groupId>commons-fileupload</groupId>
    <artifactId>commons-fileupload</artifactId>
    <version>1.4</version>
</dependency>
```

- 文件上传下载实现
  - 页面表单

```
<form action="/fileupload" method="post" enctype="multipart/form-data">
    上传LOGO:  <input type="file" name="file"/><br/>
    <input type="submit" value="上传"/>
</form>
```

- SpringMVC配置 id必须取**multipartResolver**名称,

```
<bean id="multipartResolver"

 class="org.springframework.web.multipart.commons.CommonsMultipartResol
ver">
</bean>
```

- 控制器

```
@RequestMapping(value = "/fileupload")
public void fileupload(MultipartFile file){
    file.transferTo(new File("file.png"));
}
```

5.2 文件上传注意事项

1. 文件命名问题， 获取上传文件名，并解析文件名与扩展名
2. 文件名过长问题
3. 文件保存路径
4. 重名问题

```
@RequestMapping(value = "/fileupload")
//参数中定义MultipartFile参数，用于接收页面提交的type=file类型的表单，要求表单名称与参数名相
同
public String fileupload(MultipartFile file,MultipartFile file1,MultipartFile
file2, HttpServletRequest request) throws IOException {
    System.out.println("file upload is running ..."+file);
    //        MultipartFile参数中封装了上传的文件的相关信息
    //        System.out.println(file.getSize());
    //        System.out.println(file.getBytes().length);
    //        System.out.println(file.getContentType());
    //        System.out.println(file.getName());
    //        System.out.println(file.getOriginalFilename());
    //        System.out.println(file.isEmpty());
    //首先判断是否是空文件，也就是存储空间占用为0的文件
    if(!file.isEmpty()){
        //如果大小在范围要求内正常处理，否则抛出自定义异常告知用户（未实现）
        //获取原始上传的文件名，可以作为当前文件的真实名称保存到数据库中备用
        String fileName = file.getOriginalFilename();
        //设置保存的路径
        String realPath = request.getServletContext().getRealPath("/images");
        //保存文件的方法，指定保存的位置和文件名即可，通常文件名使用随机生成策略产生，避免文件
名冲突问题
        file.transferTo(new File(realPath,file.getOriginalFilename()));
    }
    //测试一次性上传多个文件
    if(!file1.isEmpty()){
        String fileName = file1.getOriginalFilename();
```

```
        //可以根据需要，对不同种类的文件做不同的存储路径的区分，修改对应的保存位置即可
        String realPath = request.getServletContext().getRealPath("/images");
        file1.transferTo(new File(realPath,file1.getOriginalFilename()));
    }
    if(!file2.isEmpty()){
        String fileName = file2.getOriginalFilename();
        String realPath = request.getServletContext().getRealPath("/images");
        file2.transferTo(new File(realPath,file2.getOriginalFilename()));
    }
    return "page.jsp";
}
```

5.4 Restful风格配置

5.4.1 Rest

- Rest（ REpresentational State Transfer） 一种网络资源的访问风格，定义了网络资源的访问方式
  - 传统风格访问路径
    - http://localhost/user/get?id=1
    - http://localhost/deleteUser?id=1
  - Rest风格访问路径
    - http://localhost/user/1
- Restful是按照Rest风格访问网络资源
- 优点
  - 隐藏资源的访问行为，通过地址无法得知做的是何种操作
  - 书写简化

5.4.2 Rest行为约定方式

- GET（查询） http://localhost/user/1 GET
- POST（保存） http://localhost/user POST
- PUT（更新） http://localhost/user PUT
- DELETE（删除） http://localhost/user DELETE

注意：上述行为是约定方式，约定不是规范，可以打破，所以称Rest风格，而不是Rest规范

5.4.3 Restful开发入门

```
//设置rest风格的控制器
@RestController
//设置公共访问路径，配合下方访问路径使用
@RequestMapping("/user/")
public class UserController {

    //rest风格访问路径完整书写方式
    @RequestMapping("/user/{id}")
    //使用@PathVariable注解获取路径上配置的具名变量，该配置可以使用多次
    public String restLocation(@PathVariable Integer id){
        System.out.println("restful is running ....");
        return "success.jsp";
    }

    //rest风格访问路径简化书写方式，配合类注解@RequestMapping使用
    @RequestMapping("{id}")
```

```
    public String restLocation2(@PathVariable Integer id){
        System.out.println("restful is running ....get:"+id);
        return "success.jsp";
    }

    //接收GET请求配置方式
    @RequestMapping(value = "{id}",method = RequestMethod.GET)
    //接收GET请求简化配置方式
    @GetMapping("{id}")
    public String get(@PathVariable Integer id){
        System.out.println("restful is running ....get:"+id);
        return "success.jsp";
    }

    //接收POST请求配置方式
    @RequestMapping(value = "{id}",method = RequestMethod.POST)
    //接收POST请求简化配置方式
    @PostMapping("{id}")
    public String post(@PathVariable Integer id){
        System.out.println("restful is running ....post:"+id);
        return "success.jsp";
    }

    //接收PUT请求简化配置方式
    @RequestMapping(value = "{id}",method = RequestMethod.PUT)
    //接收PUT请求简化配置方式
    @PutMapping("{id}")
    public String put(@PathVariable Integer id){
        System.out.println("restful is running ....put:"+id);
        return "success.jsp";
    }

    //接收DELETE请求简化配置方式
    @RequestMapping(value = "{id}",method = RequestMethod.DELETE)
    //接收DELETE请求简化配置方式
    @DeleteMapping("{id}")
    public String delete(@PathVariable Integer id){
        System.out.println("restful is running ....delete:"+id);
        return "success.jsp";
    }
}
```

```
<!--配置拦截器，解析请求中的参数_method，否则无法发起PUT请求与DELETE请求，配合页面表单使用-->
<filter>
    <filter-name>HiddenHttpMethodFilter</filter-name>
    <filter-class>org.springframework.web.filter.HiddenHttpMethodFilter</filter-class>
</filter>
<filter-mapping>
    <filter-name>HiddenHttpMethodFilter</filter-name>
    <servlet-name>DispatcherServlet</servlet-name>
</filter-mapping>
```

□ 开启SpringMVC对Restful风格的访问支持过滤器，即可通过页面表单提交PUT与DELETE请求
□ 页面表单使用隐藏域提交请求类型，参数名称固定为_method，必须配合提交类型method=post使用

```
<form action="/user/1" method="post">
    <input type="hidden" name="_method" value="PUT"/>
    <input type="submit"/>
</form>
```

- Restful请求路径简化配置方式

```
@RestController
public class UserController {
    @RequestMapping(value = "/user/{id}",method = RequestMethod.DELETE)
    public String restDelete(@PathVariable String id){
        System.out.println("restful is running ....delete:"+id);
        return "success.jsp";
    }
}
```

5.5 postman工具安装与使用

postman 是 一款可以发送Restful风格请求的工具，方便开发调试。首次运行需要联网注册