JavaWeb基础培训

目录

[JavaWeb基础培训 1](#_Toc52371507)

[1. 创建web项目 2](#_Toc52371508)

[1.1 SpringInitializr 2](#_Toc52371509)

[1.1.1 选择Spring Initializr 2](#_Toc52371510)

[1.1.2 配置项目基本信息 2](#_Toc52371511)

[1.1.3 选择组件 3](#_Toc52371512)

[1.1.4 完成项目创建 4](#_Toc52371513)

[1.1.5 自动导入 4](#_Toc52371514)

[2. 项目介绍 5](#_Toc52371515)

[2.1 目录介绍 5](#_Toc52371516)

[2.2 pom文件的使用 6](#_Toc52371517)

[2.2.1 添加依赖 6](#_Toc52371518)

[2.2.2 其他标签 8](#_Toc52371519)

[2.3 项目文件 8](#_Toc52371520)

[Project2Application.java 8](#_Toc52371521)

[Application.properties 9](#_Toc52371522)

[3. 运行项目 9](#_Toc52371523)

[3.1 直接运行 9](#_Toc52371524)

[3.2 修改端口 10](#_Toc52371525)

[3.3 Controller与restful 11](#_Toc52371526)

[4. 获取ServletAPI 11](#_Toc52371527)

[5. springMVC拦截器配置 13](#_Toc52371528)

[5.1 拦截器类 13](#_Toc52371529)

[5.2 配置拦截器生效 14](#_Toc52371530)

[6. springMVC全局异常处理 15](#_Toc52371531)

[6.1 编写自定义异常类 15](#_Toc52371532)

[6.2 标准返回类 15](#_Toc52371533)

[6.3 异常处理类 18](#_Toc52371534)

[6.4 接口测试 18](#_Toc52371535)

[6.5 成果 19](#_Toc52371536)

[7. springAOP 19](#_Toc52371537)

[7.1 概念介绍 19](#_Toc52371538)

[Aop的基本概念： 19](#_Toc52371539)

[通知方法: 20](#_Toc52371540)

[7.2 代码实现 20](#_Toc52371541)

[7.2.1 引入jar包 20](#_Toc52371542)

[7.2.2 启用aop功能 20](#_Toc52371543)

[7.2.3 编写切面 21](#_Toc52371544)

[7.2.4 配置切入点 21](#_Toc52371545)

[7.2.5 配置通知 22](#_Toc52371546)

[7.2.6 效果 26](#_Toc52371547)

[7.3 aop总结 27](#_Toc52371548)

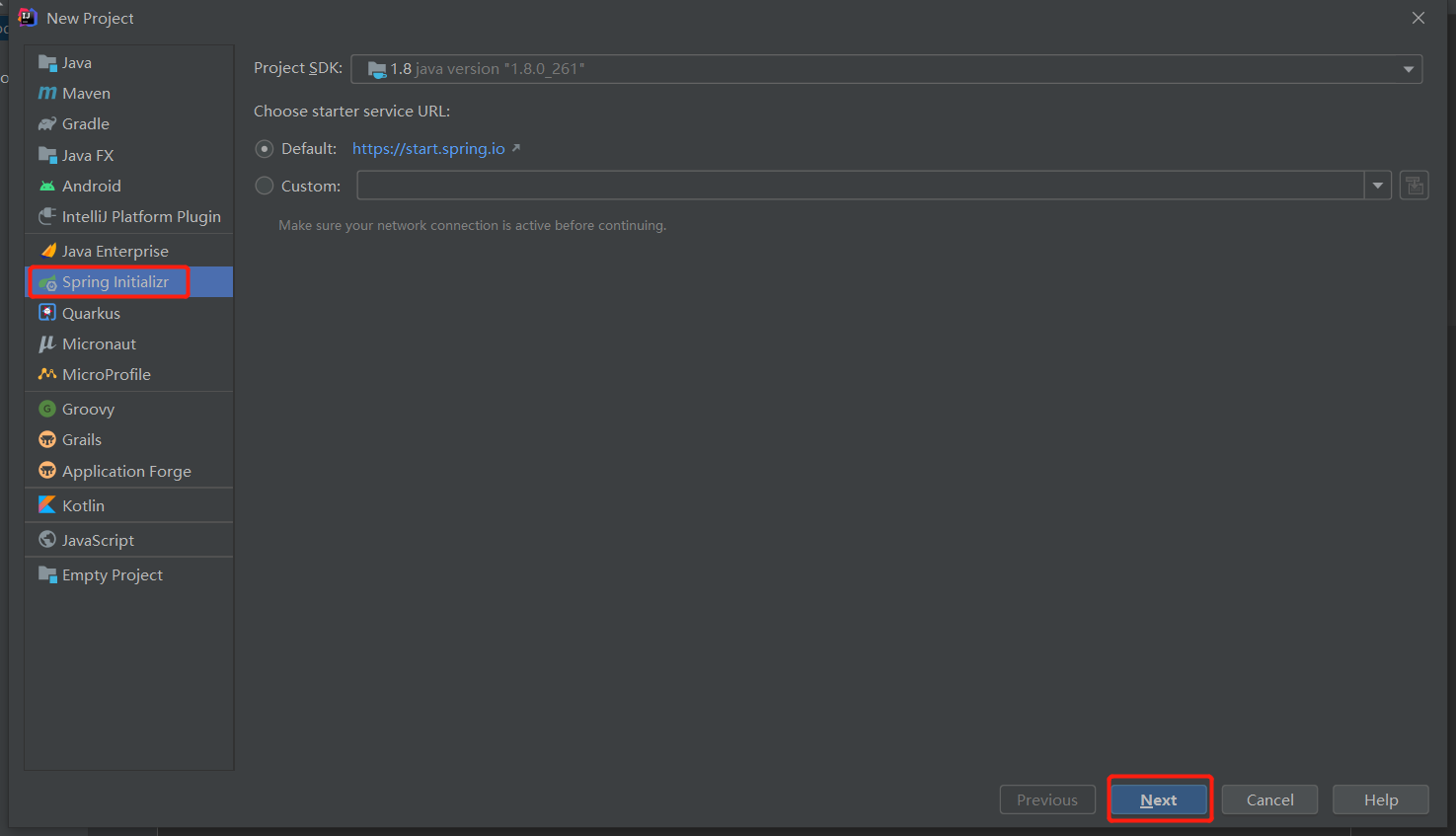
# 创建web项目

## SpringInitializr

**SpringInitializr是idea中快速搭建spring相关项目的模板，只需要简单的勾选，就可以自动导入依赖，自动配置项目所需选项。**

### 选择Spring Initializr

点击下一步后，会自动联网获取项目模板。



### 配置项目基本信息

**Group：**项目所在组，这里一般起名为项目域名倒着写，但不要写到子域名，到主域名就可以了，比如cn.com.apsp

**Artifact：**一般写项目组下的具体项目，因为一个项目组会有多个项目，所以后续会出现，group相同，Artifact不同的状况。这里我叫project2

**Type：**选择项目管理工具使用Maven还是Gradle，我们选择Maven

**Language：**选择语言，选Java

**Packaging：**选择打包方式，因为我们是用SpringBoot框架来搭建Web项目，就选择jar包就可以了

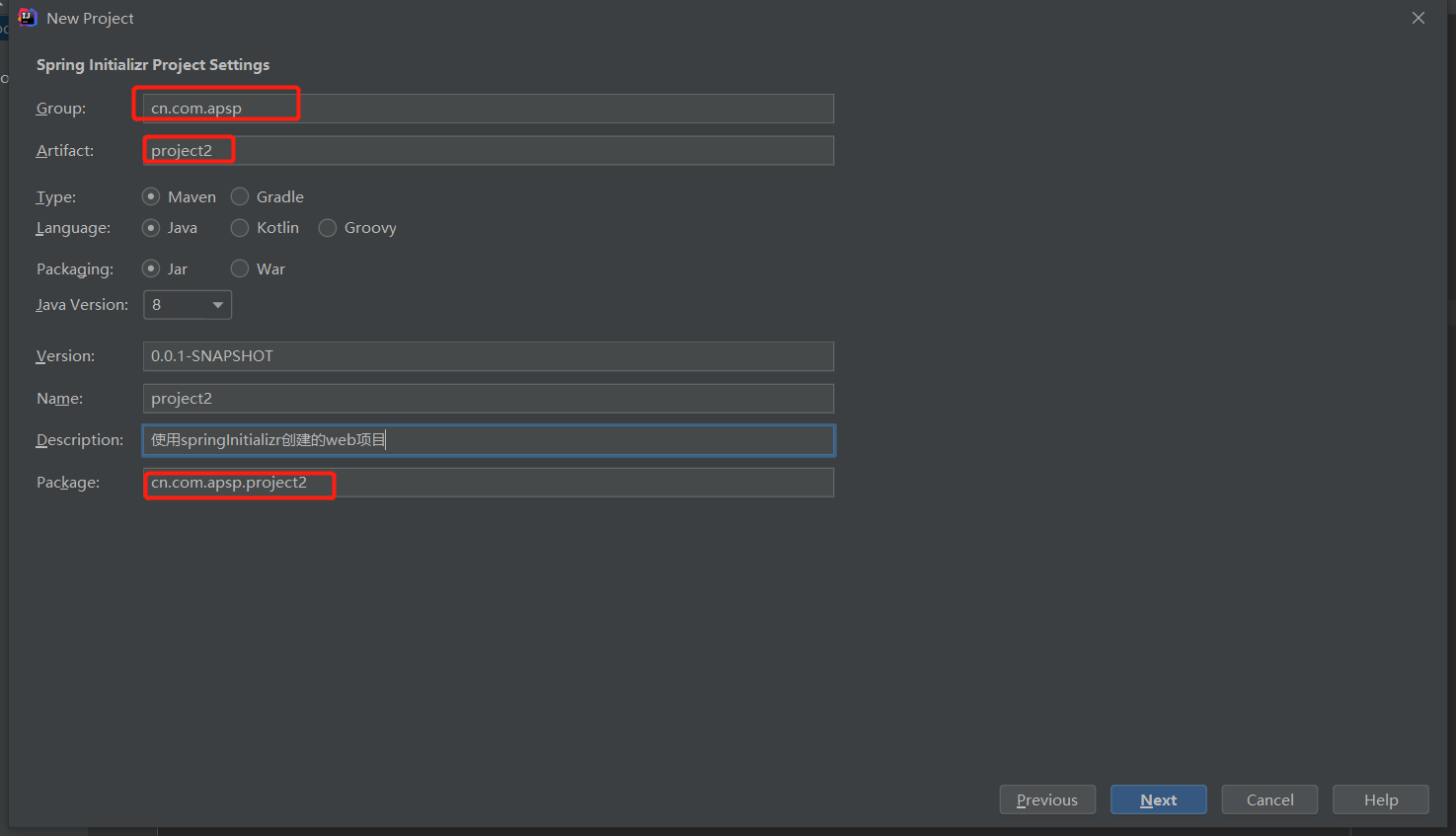
**Java Version：**选择8版本的

**Version：**指当前项目属于第几版本，默认就可以

**Name：**一般与Artifact项目

**Description：**项目备注，卸载pom文件里。

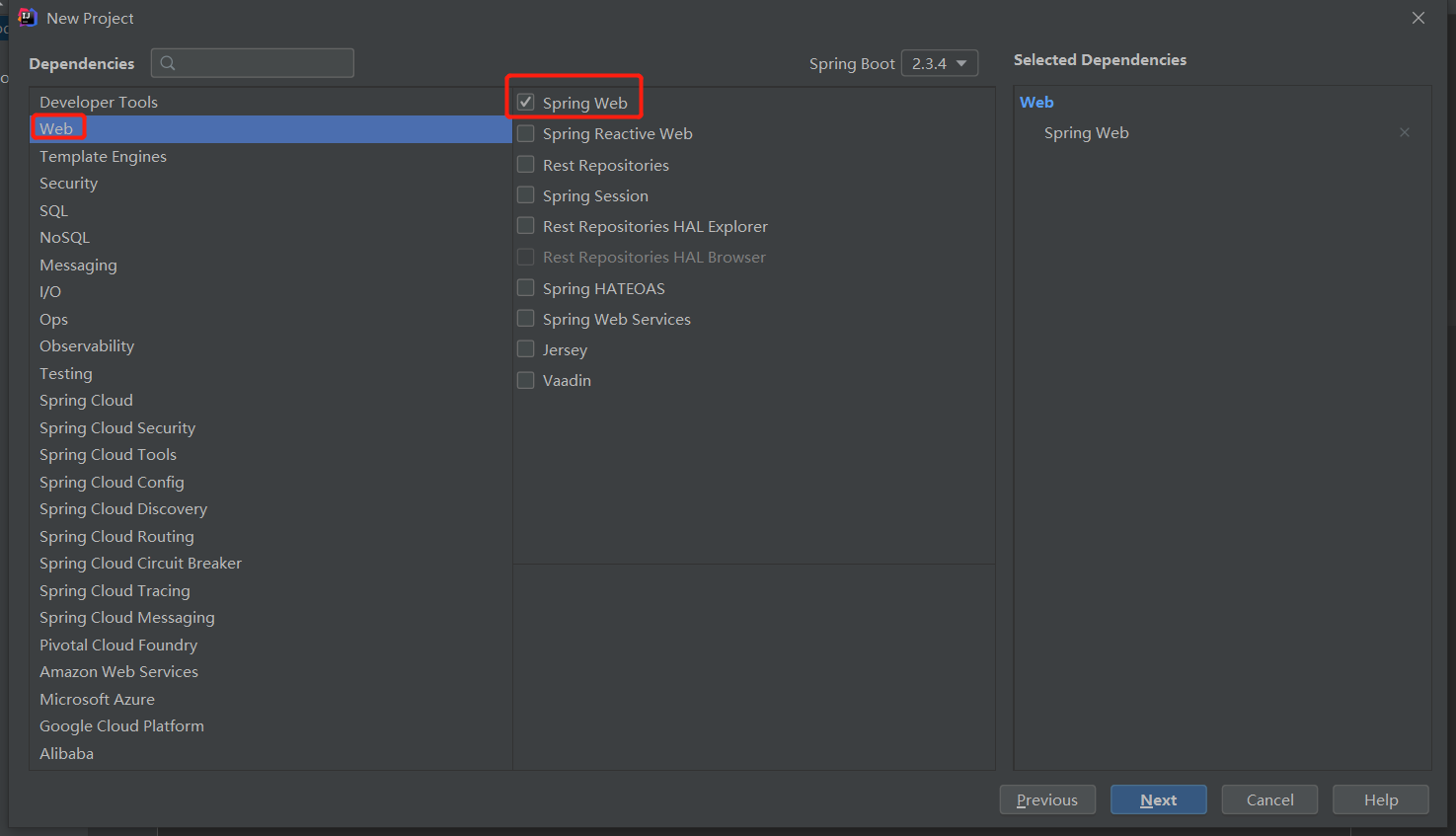
**Package：**项目包名，除非特殊要求，否则使用 group.artifact



### 选择组件

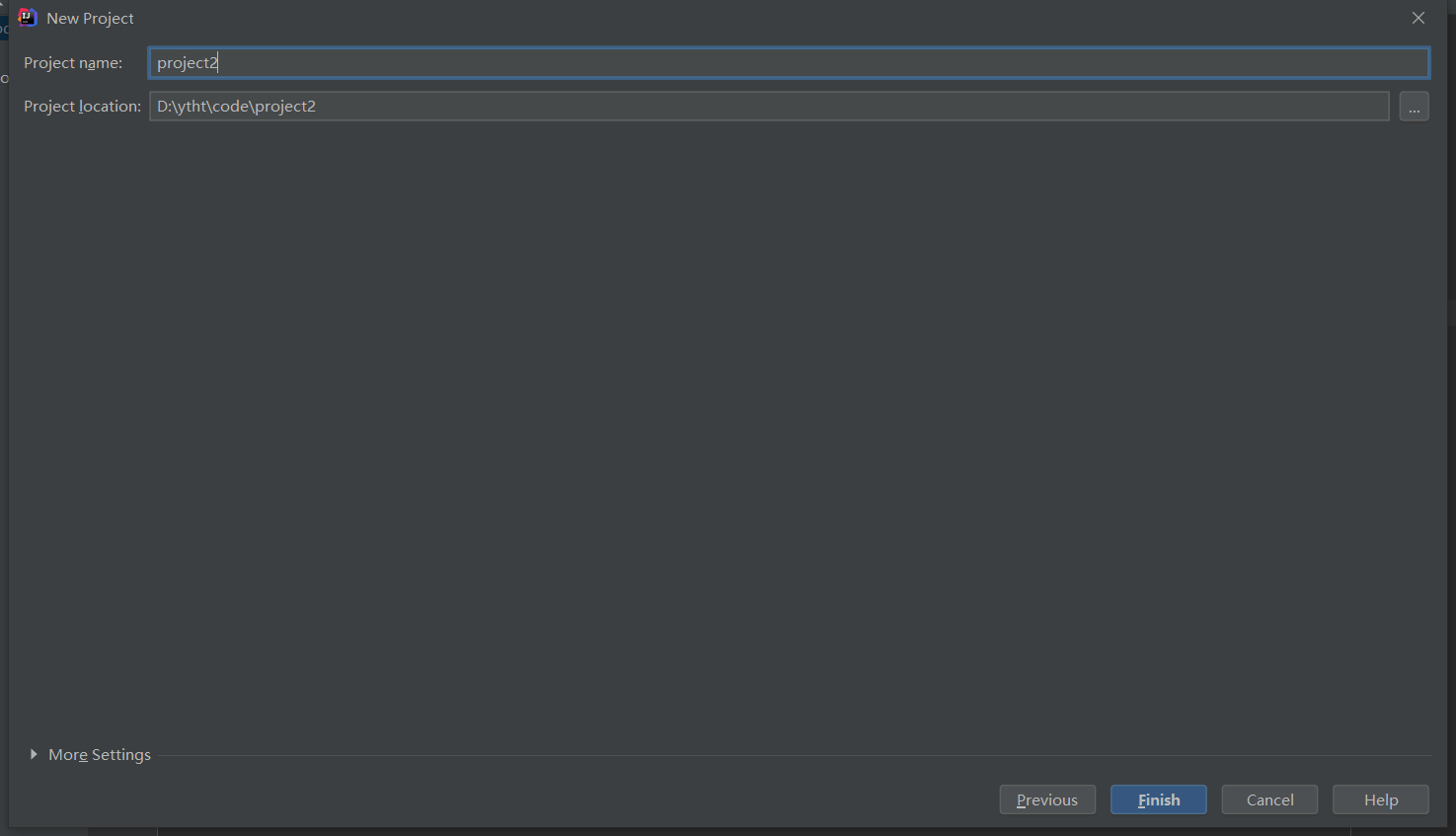
点击next来到选择组件或者说是选择依赖，来让我们的项目拥有什么功能。

**首先点击web，右侧选择 springweb** 使项目拥有springmvc和tomcat等功能。勾选后，在ide创建项目时，会自动导入springmvc相关的jar包，以及自动集成tomcat。



### 完成项目创建

点击下一步后，让我们输入项目工程名称和工程存储的位置，工程名称应和artifact值一致，不一致也不会报错。



### 自动导入

都完成后，点击finish，ide会自动帮我们创建项目，并导入jar包，网慢的话多等一会儿，加载spring的依赖需要下载几十MB的文件。

# 项目介绍

## 目录介绍

.idea：idea自己的配置文件，我们不用管

Src：此目录下的文件，是我们平常写代码的位置，一般代码写在java下，配置文件放在resources下

Test：test下也有java，这个是给我们进行单元测试的地方，有单元测试的需求，代码可以写在这里。

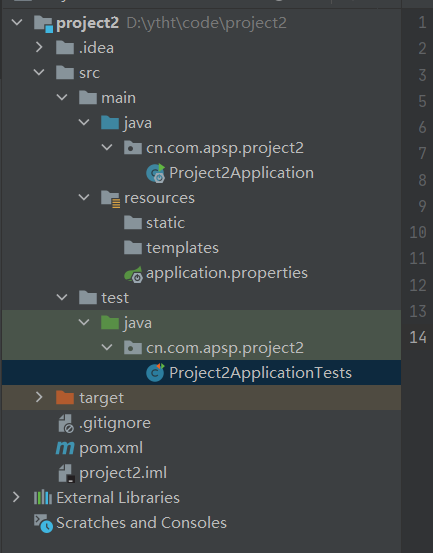
Target：这是项目编译后产生的class文件位置，以及打包文件所在文件夹

.gitignore：这是配置git忽略文件的地方

pom.xml：这里是配置依赖的地方

project2.iml：idea针对此项目生成的，不用管它

external Libraries：依赖的所有jar包都可以在这里找到。



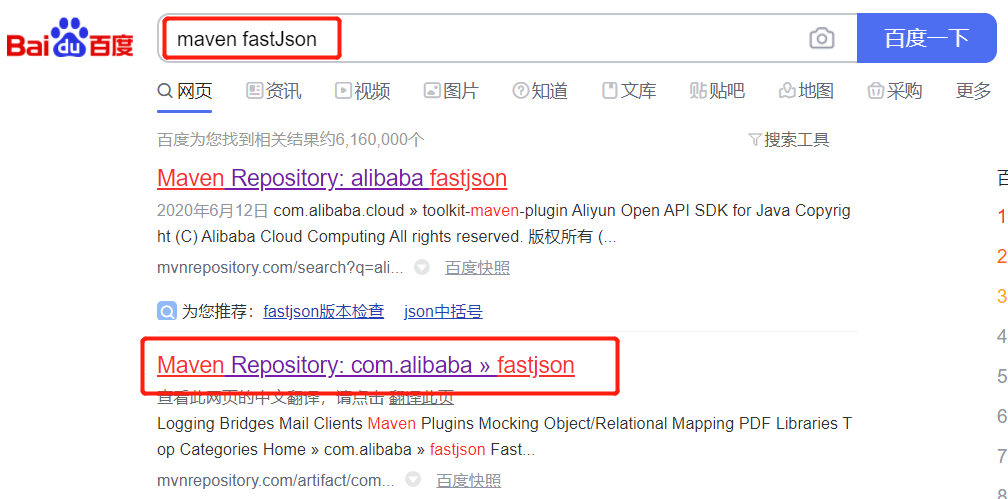
## pom文件的使用

项目中，我们经常需要添加一些jar包，现在我们通过maven来进行管理，就不应该直接导入jar包，而是应该应该在pom文件中配置上该jar包的地址，由maven来导入，接下来举个例子：

导入fastJson包，使用最新版本。

### 添加依赖

可以直接百度搜索 maven fastJson



点击进入网址：



看到最新版本是1.2.73版本，点击进去：



复制上方蓝色内容，回到idea中，找到项目的pom.xml文件，找到标签<dependencies></dependencies>，将内容黏贴进去，点击右上角刷新



Maven就开始帮我们下载jar包到本地仓库，并引用到项目中，下方进度条走完后，就可以直接使用了。

### 其他标签

properties：一些全局配置可以卸载这里

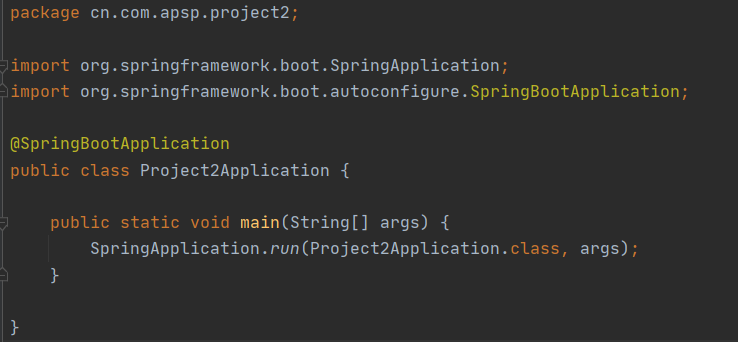
parent：指定项目的继承关系，可以继承有父项目的依赖。

build：这里可以配置一些实用的插件

## 项目文件

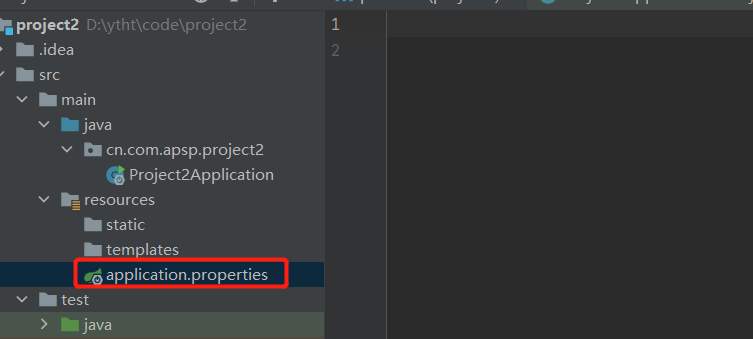
### Project2Application.java

此类是项目启动的入口，直接运行即可启动一个web项目了



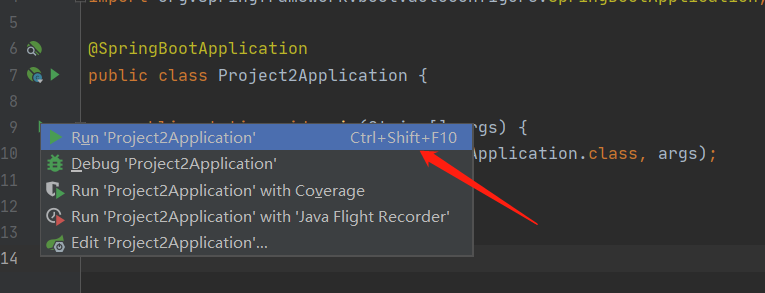
### Application.properties

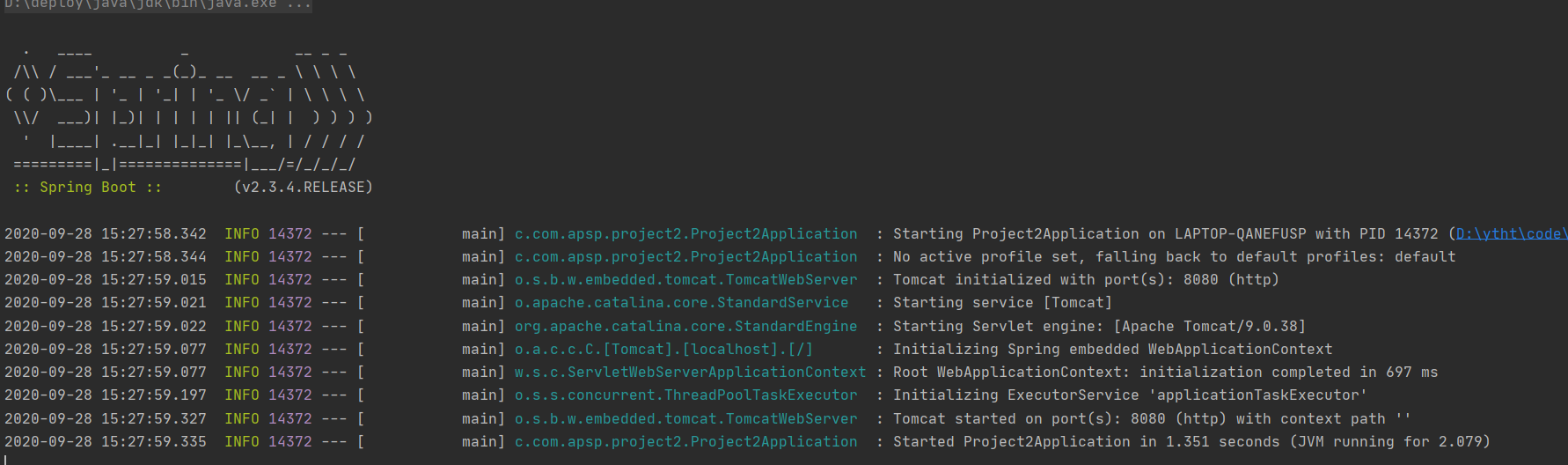
项目的配置文件，springBoot同时支持properties格式和yml格式配置文件。一般使用yml格式。



# 运行项目

## 直接运行

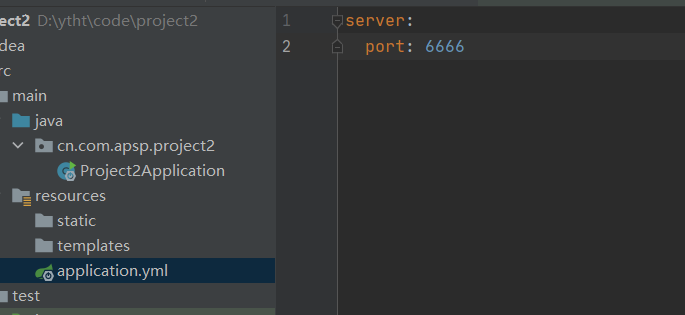




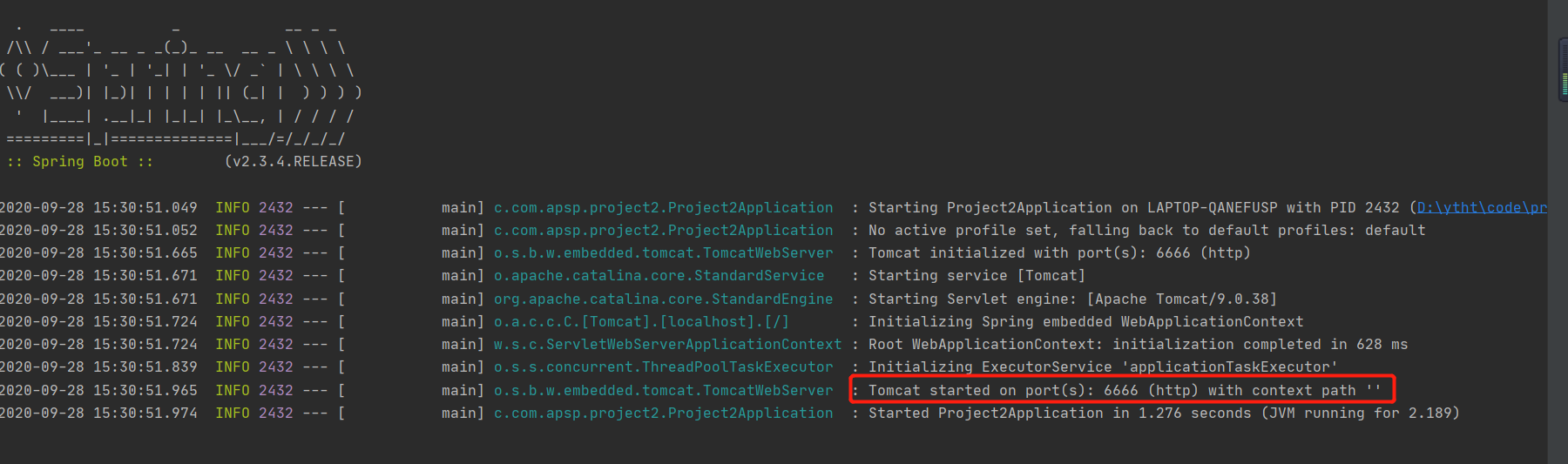
在8080端口上，已经启动一个web服务，但是由于我们什么也没有做，所以是没有页面，没有接口的。

## 修改端口

我把properties改成了yml，在里面配置：server.port=6666



再次重新启动



端口已经修改为6666

Idea有自动保存功能，所以修改后不需要保存。

## Controller与restful

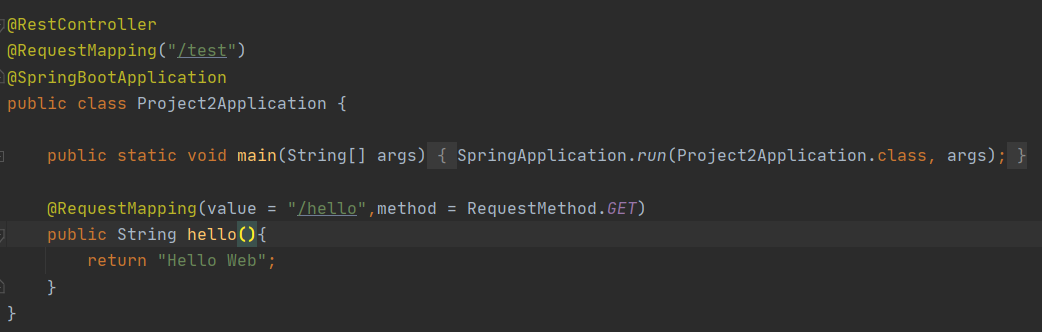
在springboot中，书写非常简单，直接在启动类中添加即可

首先在类名上添加：

@RestController 表明返回前端的是json格式数据

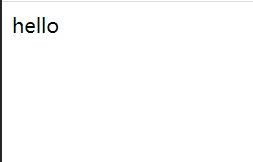
接下来写一个方法：

端口修改回8080



重新运行后

然后访问 <http://localhost:8080/test/hello> 会得到Hello Web



这样一个Web项目就完全创建起来了。

# 获取ServletAPI

在web项目中ServletApi非常重要。

Request，Repsonse，Session，Cookie等api经常被用到

|  |
| --- |
| @RequestMapping(value = "/hello", method = RequestMethod.GET)  public String hello(String userName, HttpServletRequest request, HttpServletResponse response) {  //获取session  HttpSession session = request.getSession();  //session中存入userName  session.setAttribute("user", userName);  //向响应头中存放cookie  Cookie cookie = new Cookie("id", "ABCD");  cookie.setPath("/");  response.addCookie(cookie);  System.out.println(request.getRequestURI());  return "Hello Web";  } |

此方法使用request，response，session，cookie

在浏览器访问后，可以在浏览器上看到设置的cookie信息



# springMVC拦截器配置

## 拦截器类

|  |
| --- |
| package cn.com.apsp.project2.Intercepter;  import org.springframework.stereotype.Component;  import org.springframework.web.servlet.HandlerInterceptor;  import org.springframework.web.servlet.ModelAndView;  import javax.servlet.http.HttpServletRequest;  import javax.servlet.http.HttpServletResponse;  @Component  public class FirstInterceptor implements HandlerInterceptor {  @Override  public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {  System.out.println("调用controller前");  return true;  }  @Override  public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler, ModelAndView modelAndView) throws Exception {  System.out.println("调用controller后");  }  @Override  public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex) throws Exception {  System.out.println("渲染视图后");  }  } |

1. 写一个类实现HandlerInterceptor接口
2. 实现需要使用的方法。
3. 在前置拦截中，如果返回false，将不会进入controller，返回true才会进入controller。

## 配置拦截器生效

写完拦截器类后，拦截器并不会生效，需要通过配置类，将拦截器加入springmvc的拦截器链，并配置拦截规则才可以

|  |
| --- |
| package cn.com.apsp.project2.config;  import cn.com.apsp.project2.Intercepter.FirstInterceptor; import org.springframework.beans.factory.annotation.Autowired; import org.springframework.stereotype.Component; import org.springframework.web.servlet.config.annotation.InterceptorRegistry; import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;  @Component public class InterceptorConfig implements WebMvcConfigurer {   @Autowired  private FirstInterceptor firstInterceptor;   @Override  public void addInterceptors(InterceptorRegistry registry) {  registry.addInterceptor(firstInterceptor).addPathPatterns("/\*\*");  } } |

public void addInterceptors(InterceptorRegistry registry);方法就是增加拦截器的方法，通过实现这个方法，将上部写的类的实例加入到拦截器链，并增加了拦截规则，示例规则为 拦截所有请求，包括子路径。

# springMVC全局异常处理

通过全局异常处理，可以让前端获取到指定的异常信息，并且更加人性化，可以自定义一些异常处理机制，非常好用。

## 编写自定义异常类

**随便写一个自定义异常类，等下抛出这个异常。**

|  |
| --- |
| package cn.com.apsp.project2.exception;  /\*\*  \* 自定义异常  \*/  public class MyException extends RuntimeException {  public MyException(String message) {  super(message);  }  } |

## 标准返回类

**写一个标准相应类，所有的后端返回数据，都应该按此格式进行响应，形成统一格式。**

|  |
| --- |
| package cn.com.apsp.project2.domain;  public class BaseResponse {  private boolean status;  private String msg;  private Object data;  private Integer code;  public static BaseResponse getErrorResponse(String msg) {  BaseResponse baseResponse = new BaseResponse();  baseResponse.setCode(500);  baseResponse.setData(null);  baseResponse.setStatus(false);  baseResponse.setMsg(msg);  return baseResponse;  }  public boolean isStatus() {  return status;  }  public void setStatus(boolean status) {  this.status = status;  }  public String getMsg() {  return msg;  }  public void setMsg(String msg) {  this.msg = msg;  }  public Object getData() {  return data;  }  public void setData(Object data) {  this.data = data;  }  public Integer getCode() {  return code;  }  public void setCode(Integer code) {  this.code = code;  }  @Override  public String toString() {  return "BaseResponse{" +  "status=" + status +  ", msg='" + msg + '\'' +  ", data=" + data +  ", code=" + code +  '}';  }  } |

## 异常处理类

**编写异常处理类，用来接收我们抛出的异常**

|  |
| --- |
| package cn.com.apsp.project2.exception;  import cn.com.apsp.project2.domain.BaseResponse;  import org.springframework.web.bind.annotation.ControllerAdvice;  import org.springframework.web.bind.annotation.ExceptionHandler;  import org.springframework.web.bind.annotation.ResponseBody;  @ControllerAdvice  public class MyExceptionHandler {  @ExceptionHandler(value = MyException.class)  @ResponseBody  public BaseResponse myExceptionAdvice(MyException myException) {  //这里是异常处理  return BaseResponse.getErrorResponse(myException.getMessage());  }  } |

## 接口测试

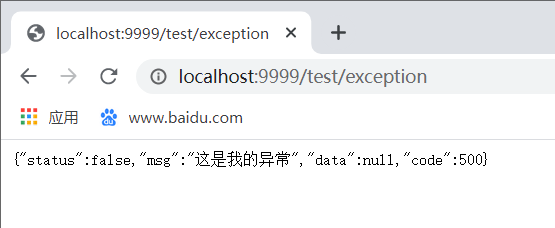
再写一个接口，由前端访问，并抛出自定义异常。

|  |
| --- |
| @RequestMapping(value = "/exception", method = RequestMethod.GET)  public String hello() {  throw new MyException("这是我的异常");  } |

## 成果

访问地址：http://localhost:9999/test/exception

返回：{"status":false,"msg":"这是我的异常","data":null,"code":500}



# springAOP

aop面向切面编程。对原有的方法进行增强。

## 概念介绍

### Aop的基本概念：

1. Aspect(切面):通常是一个类，里面可以定义切入点和通知
2. JointPoint(连接点):程序执行过程中明确的点，一般是方法的调用
3. Advice(通知):AOP在特定的切入点上执行的增强处理，有before,after,afterReturning,afterThrowing,around
4. Pointcut(切入点):就是带有通知的连接点，在程序中主要体现为书写切入点表达式
5. AOP代理：AOP框架创建的对象，代理就是目标对象的加强。Spring中的AOP代理可以使JDK动态代理，也可以是CGLIB代理，前者基于接口，后者基于子类

### 通知方法:

1. 前置通知:在我们执行目标方法之前运行(@Before)
2. 后置通知:在我们目标方法运行结束之后 ,不管有没有异常(@After)
3. 返回通知:在我们的目标方法正常返回值后运行(@AfterReturning)
4. 异常通知:在我们的目标方法出现异常后运行(@AfterThrowing)
5. 环绕通知:动态代理, 需要手动执行joinPoint.procced()(其实就是执行我们的目标方法执行之前相当于前置通知, 执行之后就相当于我们后置通知(@Around)

## 代码实现

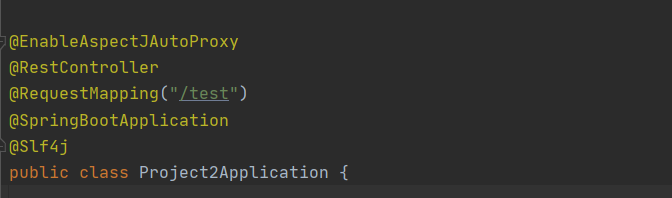
### 引入jar包

使用AOP必须引入jar包，复制下方maven地址到POM文件即可，刷新pom文件。

|  |
| --- |
| <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-aop</artifactId>  </dependency> |

### 启用aop功能

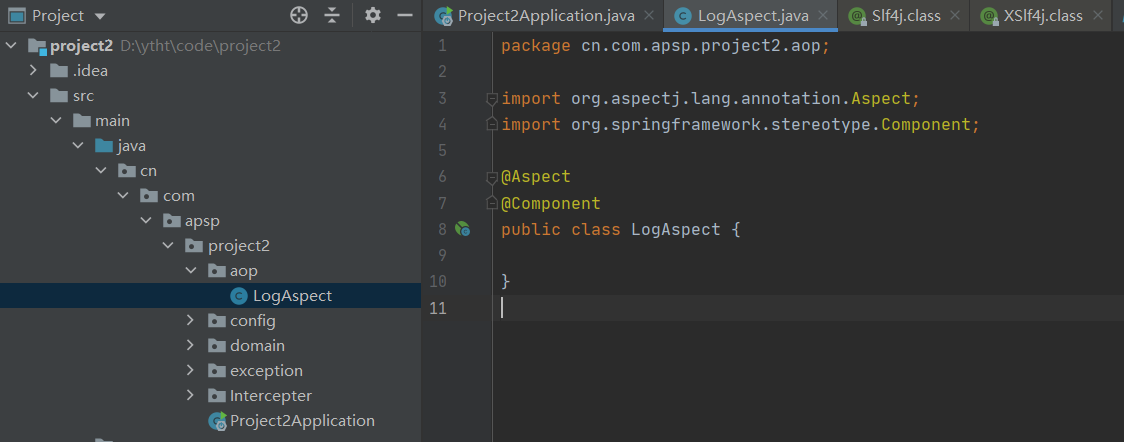
在启动类上添加注解：@EnableAspectJAutoProxy



### 编写切面

新建一个切面类

|  |
| --- |
| package cn.com.apsp.project2.aop;  import org.aspectj.lang.annotation.Aspect;  import org.springframework.stereotype.Component;  @Aspect  @Component  public class LogAspect {    } |

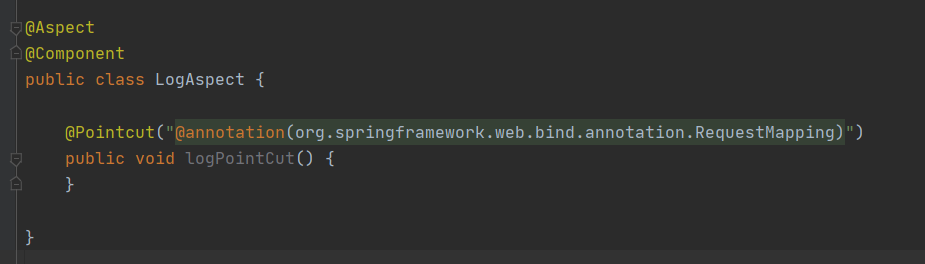


### 配置切入点

添加代码：

|  |
| --- |
| @Pointcut("@annotation(org.springframework.web.bind.annotation.RequestMapping)")  public void logPointCut() {  } |

这是通过注解的方式配置切入点，所有拥有此注解的方法，都会被切



### 配置通知

这次使用前置通知，返回通知，后置通知，异常通知

1. 编写日志类，进行管理。

**注意：这里使用了lombok，通过在类上增加注解，省去书写 get set toString 方法，使程序更简洁**

|  |
| --- |
| package cn.com.apsp.project2.domain;  import lombok.Data;  @Data  public class SystemLog {  /\*\*  \* 请求参数  \*/  private String requestParam;  /\*\*  \* 请求的类  \*/  private String className;  /\*\*  \* 请求的方法  \*/  private String method;  /\*\*  \* 远程ip地址  \*/  private String remoteIp;  /\*\*  \* 接口返回值  \*/  private Object result;  /\*\*  \* 开始时间  \*/  private Long startTime;  /\*\*  \* 结束时间  \*/  private Long endTime;  /\*\*  \* 耗时  \*/  private Long useTime;  /\*\*  \* 异常信息  \*/  private String exceptionMsg;  } |

1. 完善通知

|  |
| --- |
| package cn.com.apsp.project2.aop;  import cn.com.apsp.project2.domain.SystemLog;  import com.alibaba.fastjson.JSONObject;  import lombok.Data;  import lombok.extern.slf4j.Slf4j;  import org.aspectj.lang.JoinPoint;  import org.aspectj.lang.Signature;  import org.aspectj.lang.annotation.\*;  import org.springframework.stereotype.Component;  import org.springframework.web.context.request.RequestContextHolder;  import org.springframework.web.context.request.ServletRequestAttributes;  import javax.servlet.http.HttpServletRequest;  import java.util.Map;  @Data  @Slf4j  @Aspect  @Component  public class LogAspect {  /\*\*  \* 线程本地化，解决多线程问题  \*/  private ThreadLocal<SystemLog> logThreadLocal = new ThreadLocal<>();  @Pointcut("@annotation(org.springframework.web.bind.annotation.RequestMapping)")  public void logPointCut() {  }  @Before("logPointCut()")  public void beforeAdvice(JoinPoint joinPoint) {  //获取request  ServletRequestAttributes requestAttributes = (ServletRequestAttributes) RequestContextHolder.currentRequestAttributes();  HttpServletRequest request = requestAttributes.getRequest();  //获取请求信息  Signature signature = joinPoint.getSignature();  //请求的方法名  String methodName = signature.getName();  //请求的类的名字  String className = signature.getDeclaringTypeName();  SystemLog systemLog = new SystemLog();  systemLog.setClassName(className);  systemLog.setMethod(methodName);  systemLog.setStartTime(System.currentTimeMillis());  //获取请求参数  Map<String, String[]> parameterMap = request.getParameterMap();  String paramString = JSONObject.toJSON(parameterMap).toString();  systemLog.setRequestParam(paramString);  systemLog.setRemoteIp(request.getRemoteHost());  logThreadLocal.set(systemLog);  }  @AfterReturning(value = "logPointCut()", returning = "obj")  public void afterReturningAdvice(Object obj) {  SystemLog systemLog = logThreadLocal.get();  Long endTime = System.currentTimeMillis();  Long startTime = systemLog.getStartTime();  systemLog.setUseTime(endTime - startTime);  systemLog.setEndTime(endTime);  systemLog.setResult(obj);  }  @AfterThrowing(value = "logPointCut()", throwing = "e")  public void throwExceptionAdvice(Exception e) {  SystemLog systemLog = logThreadLocal.get();  systemLog.setExceptionMsg(e.getMessage());  }  @After("logPointCut()")  public void afterAdvice() {  SystemLog systemLog = logThreadLocal.get();  log.info(systemLog.toString());  logThreadLocal.remove();  log.info("响应时间{}，请求地址：{},请求接口：{}.{}，请求参数：{}，返回：{}",  systemLog.getUseTime(), systemLog.getRemoteIp(), systemLog.getClassName(),  systemLog.getMethod(), systemLog.getRequestParam(), systemLog.getResult());  }  } |

### 效果

使用另外一台电脑访问接口，看控制台输出：

|  |
| --- |
| 请求耗时0，请求地址：192.168.3.2,请求接口：cn.com.apsp.project2.Project2Application.hello，请求参数：{"userName":["123"]}，返回：Hello Web |



## aop总结

aop能实现很多功能，对于日志记录这种，可以极大程度减少代码量，并且很好配置，在原有的方法上看不出任何代码侵入。不影响整体的美观，而且增强程序功能。