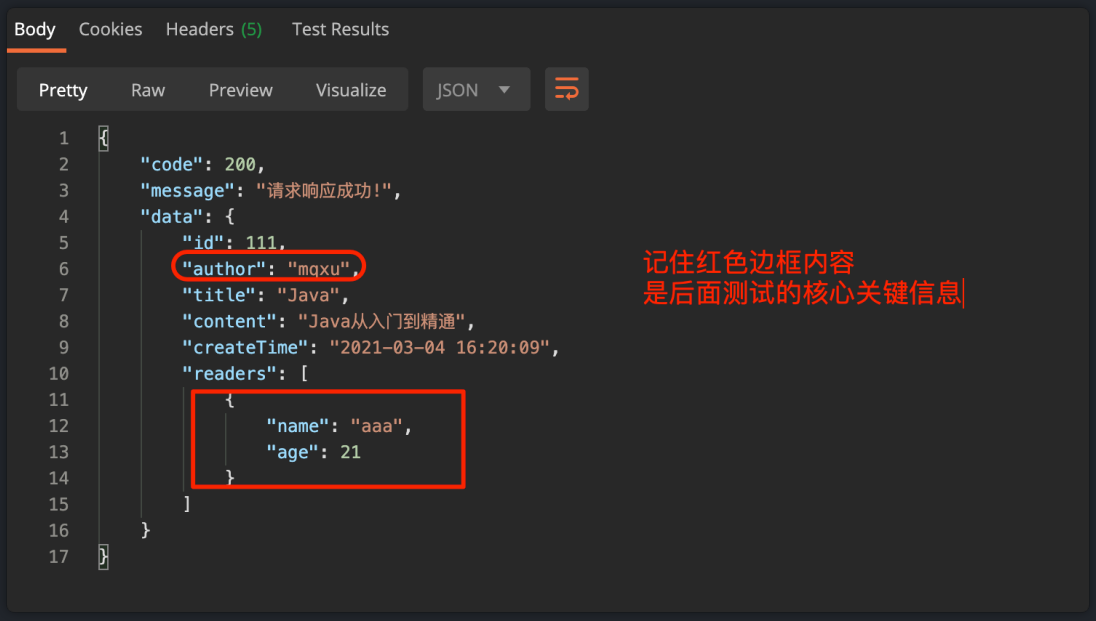
2.4 使用Mockito编码完成接口测试

之前我们是使用postman来进行接口测试的，本节使用编码的方式来进行接口测试。

使用Postman的测试新增接口返回结果如下：



一、编码实现接口测试

1.1.为什么要写代码做测试

> **使用接口测试工具Postman很方便啊，为什么还要用代码做测试？因为在做系统的自动化持续集成的时候，会要求自动的做单元测试，只有所有的单元测试都跑通了，才能打包构建。比如：使用maven在打包之前将所有的测试用例执行一遍。这里重点是自动化，所以postman这种工具很难插入到持续集成的自动化流程中去。**

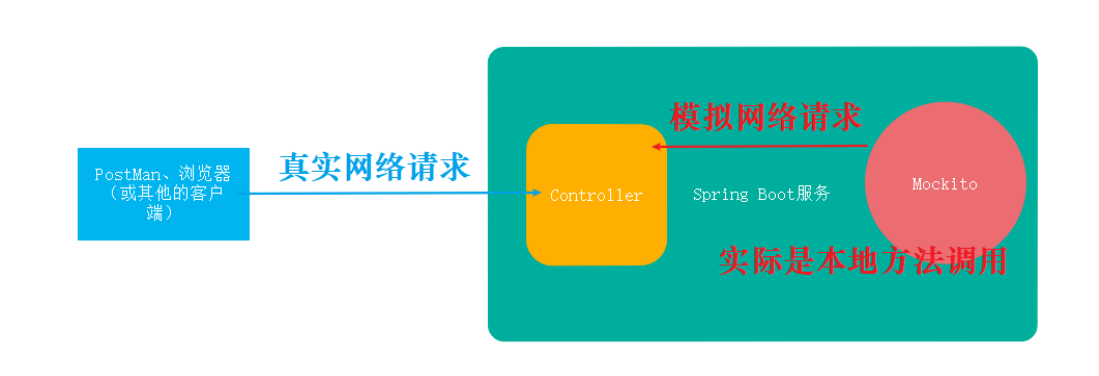
1.2.Junit测试框架

在开始编写测试代码之前，我们先回顾一下JUnit常用的测试注解。在junit4和junit5中，注解的写法有些许变化。

junit4	junit5	特点
@Test	@Test	声明一个测试方法
@BeforeClass	@BeforeAll	在当前类的所有测试方法之前执行。注解在【静态方法】上
@AfterClass	@AfterAll	在当前类中的所有测试方法之后执行。注解在【静态方法】上
@Before	@BeforeEach	在每个测试方法之前执行。注解在【非静态方法】上
@After	@AfterEach	在每个测试方法之后执行。注解在【非静态方法】
@RunWith(SpringRunner.class)	@ExtendWith(SpringExtension.class)	类class定义上

1.3.Mockito测试框架

Mockito是GitHub上使用最广泛的Mock框架,并与JUnit结合使用.Mockito框架可以创建和配置mock对象.使用Mockito简化了具有外部依赖的类的测试开发。Mockito测试框架可以帮助我们模拟HTTP请求，从而达到在服务端测试目的。因为其不会真的去发送HTTP请求，而是模拟HTTP请求内容，从而节省了HTTP请求的网络传输，测试速度更快。




```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
    <exclusions>
        <exclusion>
            <groupId>org.junit.vintage</groupId>
            <artifactId>junit-vintage-engine</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

> **spring-boot-starter-test自动包含Junit 5 和Mockito框架，以下测试代码是基于Junit5。**

```
package top.mqxu.boot.basic.controller;

import lombok.extern.slf4j.Slf4j;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.Test;
import org.springframework.http.HttpMethod;
import org.springframework.test.web.servlet.MockMvc;
import org.springframework.test.web.servlet.MvcResult;
import
org.springframework.test.web.servlet.request.MockMvcReques
tBuilders;
import
org.springframework.test.web.servlet.result.MockMvcResultM
atchers;
import
org.springframework.test.web.servlet.setup.MockMvcBuilders
;

import static
org.springframework.test.web.servlet.result.MockMvcResultH
andlers.print;

@Slf4j
class ArticleControllerTest {

    //mock对象
    private static MockMvc mockMvc;

    //在所有测试方法执行之前进行mock对象初始化
    @BeforeAll
    static void setUp() {
        mockMvc = MockMvcBuilders.standaloneSetup(new
ArticleController()).build();
    }

    @Test
    void saveArticle() throws Exception {
        String article = "{\n" +
```

```
                "    \"id\": 1,\n" +
                "    \"author\": \"mqxu\",\n" +
                "    \"title\": \"Spring Boot从入门到精通
\",\n" +
                "    \"content\": \"Spring Boot从入门到精通
\",\n" +
                "    \"createTime\": \"2021-03-06
15:23:34\",\n" +
                "    \"readers\":
[{\"name\":\"aaa\",\"age\":21},
{\"name\":\"bbb\",\"age\":20}]\n" +
                "}";
        MvcResult result = mockMvc.perform(
                MockMvcRequestBuilders
                        .request(HttpMethod.POST,
"/api/v1/articles/body")
                        .contentType("application/json")
                        .content(article)
        )

.andExpect(MockMvcResultMatchers.status().isOk())

.andExpect(MockMvcResultMatchers.jsonPath("$.data.author")
.value("mqxu"))

.andExpect(MockMvcResultMatchers.jsonPath("$.data.readers[
0].age").value(21))
                .andDo(print())
                .andReturn();
        result.getResponse().setCharacterEncoding("UTF-
8");

 log.info(result.getResponse().getContentAsString());
    }
}
```

注意：实体类头部都加上四个注解，不然会报错

```
@Data
@Builder
@AllArgsConstructor
@NoArgsConstructor
```

MockMvc对象有以下几个基本的方法:

- perform : 模拟执行一个RequestBuilder构建的HTTP请求，会执行SpringMVC的流程并映射到相应的控制器Controller执行。
- contentType：发送请求内容的序列化的格式，"application/json"表示JSON数据格式
- andExpect: 添加RequsetMatcher验证规则，验证控制器执行完成后结果是否正确，或者说是结果是否与我们期望（Expect）的一致。
- andDo: 添加ResultHandler结果处理器，比如调试时打印结果到控制台
- andReturn: 最后返回相应的MvcResult,然后进行自定义验证/进行下一步的异步处理

> **上面的整个过程，我们都没有使用到Spring Context依赖注入、也没有启动tomcat web容器。整个测试的过程十分的轻量级，速度很快。**

二、真实servlet容器环境下的测试

上面的测试执行速度非常快，但是有一个问题：它没有启动servlet容器和Spring 上下文，自然也就无法实现依赖注入（不支持@Resource和@AutoWired注解）。这就导致它在从控制层到持久层全流程测试中有很大的局限性。

换一种写法：看看有没有什么区别。在测试类上面额外加上这样两个注解，并且mockMvc对象使用@Resource自动注入，删掉Before注解及setUp函数。

```
@AutoConfigureMockMvc
@SpringBootTest
@ExtendWith(SpringExtension.class)
```

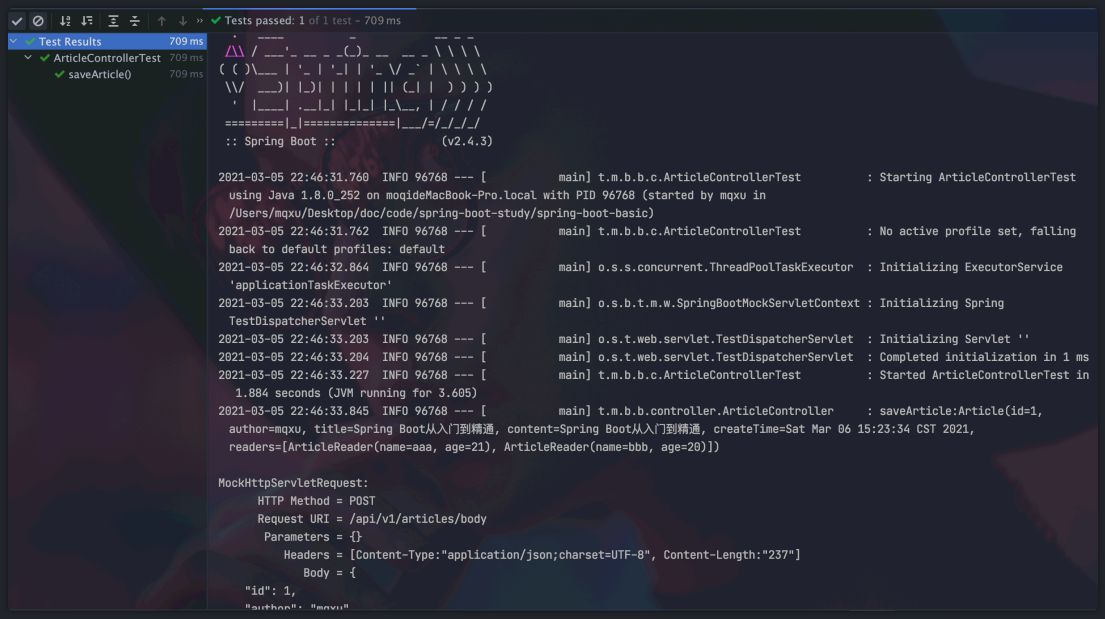
```
@Slf4j
@AutoConfigureMockMvc
@SpringBootTest
@ExtendWith(SpringExtension.class)
class ArticleControllerTest {

    //mock对象
    @Resource
    private MockMvc mockMvc;

    //在所有测试方法执行之前进行mock对象初始化
    //@BeforeAll
    //static void setUp() {
    //    mockMvc = MockMvcBuilders.standaloneSetup(new ArticleController()).build();
    //}
```

添加注解

注入MockMvc实例

删掉

启动测试一下，看看和之前有没有什么区别



看到上面这个截图，是不是已经明白了！该测试方法真实的启动了一个tomcat容器、以及Spring 上下文，所以我们可以进行依赖注入（@Resource）。实现的效果和使用MockMvcBuilders构建MockMVC对象的效果是一样的，但是有一个非常明显的缺点：每次做一个接口测试，都会真实的启动一次servlet容器，Spring上下文加载项目里面定义的所有的Bean，导致执行过程很缓慢。

2.1 @SpringBootTest 注解

是用来创建Spring的上下文ApplicationContext，保证测试在上下文环境里运行。单独使用@SpringBootTest不会启动servlet容器。所以**只是使用SpringBootTest 注解，不可以使用@Resource和@Autowired等注解进行bean的依赖注入**。（准确的说是可以使用，但被注解的bean为null）。

2.2 @ExtendWith(@RunWith注解)

- RunWith方法为我们构造了一个的Servlet容器运行运行环境，并在此环境下测试。然而为什么要构建servlet容器？因为使用了依赖注入，注入了MockMvc对象，而在上一个例子里面是我们自己new的。
- 而@AutoConfigureMockMvc注解，该注解表示mockMvc对象由spring 依赖注入构建，你只负责使用就可以了。这种写法是为了让测试在servlet容器环境下执行。

简单的说：**如果你单元测试代码使用了“依赖注入@Resource”就必须加上@ExtendWith，如果你不是手动new MockMvc对象就加上@AutoConfigureMockMvc**

> **实际上@SpringBootTest 注解注解已经包含了 @ExtendWith注解，如果使用了前者，可以忽略后者！**

2.3 @Transactional

该注解加在方法上可以使单元测试进行事务回滚，以保证数据库表中没有因测试造成的垃圾数据，因此保证单元测试可以反复执行；
但是不建议这么做，使用该注解会破坏测试真实性。请参考这篇文章详细理解：
不要在 Spring Boot 集成测试中使用 @Transactional

2.5 使用Swagger2构建API文档

一、为什么要发布API接口文档

当下很多公司都采取前后端分离的开发模式，前端和后端的工作由不同的工程师完成。在这种开发模式下，维护一份及时更新且完整的API 文档将会极大的提高我们的工作效率。传统意义上的文档都是后端开发人员使用word编写的，相信大家也都知道这种方式很难保证文档的及时性，这种文档久而久之也就会失去其参考意义，反而还会加大我们的沟通成本。而Swagger 给我们提供了一个全新的维护 API 文档的方式，下面我们就来了解一下它的优点：

- 代码变，文档变。只需要少量的注解，Swagger 就可以根据代码自动生成 API 文档，很好的保证了文档的时效性。
- 跨语言性，支持 40 多种语言。
- Swagger UI 呈现出来的是一份可交互式的 API 文档，我们可以直接在文档页面尝试 API 的调用，省去了准备复杂的调用参数的过程。
- 还可以将文档规范导入相关的工具（例如 SoapUI），这些工具将会为我们自动地创建自动化测试。

二、整合swagger2生成文档

最近 SpringFox 3.0.0 发布了，距离上一次大版本2.9.2足足有2年多时间了。

当我们在使用Spring MVC写接口的时候，为了生成API文档，为了方便整合Swagger，都是用这个SpringFox的这套封装。但是，自从2.9.2版本更新之后，就一直没有什么动静，也没有更上Spring Boot的大潮流。

现在SpringFox出了一个starter，看了一下功能，虽然还不完美，但相较于之前我们自己的轮子来说还是好蛮多的。来看看这个版本有些什么亮点：

- Spring 5，Webflux 支持（仅请求映射支持，尚不支持功能端点）
- Spring Integration 支持
- Spring Boot 支持 springfox-boot-starter 依赖性（零配置，自动配置支持）
- 具有自动完成功能的文档化配置属性
- 更好的规范兼容性
- 支持 OpenApi 3.0.3
- 几乎零依赖性（唯一需要的库是 spring-plugin、pswagger-core）
- 现有的 swagger2 注释将继续有效，并丰富 open API 3.0 规范

对于这次的更新，我觉得比较突出的几点：Webflux的支持，目前的轮子就没有做到；对OpenApi 3的支持；以及对Swagger 2的兼容（可以比较方便的做升级了）。

说那么多，不如来一发程序实验下更直接！

```
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-boot-starter</artifactId>
    <version>3.0.0</version>
<dependency>
```

然后通过启动主类加上注解

```
@SpringBootApplication
@EnableOpenApi
public class SpringBootBasicApplication {
    public static void main(String[] args) {

 SpringApplication.run(SpringBootBasicApplication.class,
args);
    }
}
```

三、书写swagger注解

通常情况下Controller类及方法书写了swagger注解，就不需要写java注释了。因为一个成熟的团队，前端人员根据英文方法的名称和参数名称就能知道方法的作用，前提是代码开发者认真的为接口及参数起英文名。通过团队内推广RESTful接口的设计原则和良好的统一的交互规范，就能知道响应结果的含义。这也是一种“约定大于配置”的体现。

controller注解

```
package top.mqxu.boot.basic.controller;

import com.fasterxml.jackson.core.JsonProcessingException;
import com.fasterxml.jackson.databind.ObjectMapper;
import io.swagger.annotations.Api;
import io.swagger.annotations.ApiOperation;
```

```java
import io.swagger.annotations.ApiParam;
import lombok.extern.slf4j.Slf4j;
import
org.springframework.format.annotation.DateTimeFormat;
import org.springframework.web.bind.annotation.*;
import top.mqxu.boot.basic.controller.dto.AjaxResponse;
import top.mqxu.boot.basic.entity.Article;
import top.mqxu.boot.basic.entity.ArticleReader;

import java.util.Arrays;
import java.util.Date;
import java.util.List;

/**
 * @description: ArticleController
 * @author: mqxu
 * @since: 2021-03-05
 **/
@Slf4j
@RestController
@RequestMapping(value = "/api/v1/articles")
@Api(tags = "文章管理接口")
public class ArticleController {
    /**
     * 查询文章，id为URL查询参数
     *
     * @param id 文章id
     * @return AjaxResponse
     */
    @ApiOperation("URL传参，根据id获取文章")
    @GetMapping()
    public AjaxResponse getArticleByParam(@ApiParam("文章
id") @RequestParam("id") int id) {
        ArticleReader[] readers = {

 ArticleReader.builder().name("aaa").age(20).build(),

 ArticleReader.builder().name("bbb").age(19).build()};
        List<ArticleReader> readerList =
Arrays.asList(readers);
        Article article = Article.builder()
                .id(id)
```

```
                .author("mqxu")
                .title("Spring Boot从入门到精通")
                .content("Spring Boot从入门到精通")
                .createTime(new Date())
                .readers(readerList)
                .build();
        log.info("article: " + article);
        return AjaxResponse.success(article);
    }


    /**
     * 增加一篇Article , @RequestParam接收参数
     *
     * @param id         id
     * @param author     作者
     * @param title      标题
     * @param content    内容
     * @param createTime 创建时间
     * @return AjaxResponse
     */
    @ApiOperation("URL传参新增文章")
    @PostMapping("param")
    public AjaxResponse saveArticle(
            @ApiParam("文章id")
            @RequestParam(value = "id", defaultValue =
"111", required = false) int id,
            @ApiParam("作者")
            @RequestParam(value = "author", defaultValue =
"mqxu", required = false) String author,
            @ApiParam("标题")
            @RequestParam String title,
            @ApiParam("内容")
            @RequestParam String content,
            @ApiParam("创建时间")
            @DateTimeFormat(pattern = "yyyy-MM-dd
HH:mm:ss")
            @RequestParam(value = "createTime",
defaultValue = "2021-03-06 12:12:12", required = false)
Date createTime) {
        Article article = Article.builder()
                .id(id)
```

```
                .title(title)
                .content(content)
                .author(author)
                .createTime(createTime)
                .build();
        log.info("saveArticle:" + article);
        return AjaxResponse.success(article);
    }

    /**
     * 增加一篇Article @RequestParam取得表单对象序列化的字符串
     *
     * @param formData 表单对象序列化的字符串
     * @return AjaxResponse
     */
    @ApiOperation("表单请求体新增文章")
    @PostMapping("form")
    public AjaxResponse saveArticleByFormData(@ApiParam("表
单字符串") @RequestParam("formData") String formData) {
        //表单传递的数据为字符串
        log.info("formData:" + formData);
        //用Jackson的反序列化将字符串转为Java对象
        ObjectMapper objectMapper = new ObjectMapper();
        Article article = null;
        try {
            article = objectMapper.readValue(formData,
Article.class);
            log.info("article:" + article);
        } catch (
                JsonProcessingException e) {
            e.printStackTrace();
        }
        return AjaxResponse.success(article);
    }

}
```

实体类注解

```java
package top.mqxu.boot.basic.controller.dto;

import io.swagger.annotations.ApiModel;
import io.swagger.annotations.ApiModelProperty;
import lombok.Data;

/**
 * @description:
 * @author: mqxu
 * @since: 2021-03-05
 **/
@Data
@ApiModel("统一响应结果")
public class AjaxResponse {
    /**
     * 请求响应状态码（200、400、500）
     */
    @ApiModelProperty("请求响应状态码")
    private int code;
    /**
     * 请求结果描述信息
     */
    @ApiModelProperty("请求结果描述信息")
    private String message;
    /**
     * 请求返回数据
     */
    @ApiModelProperty("请求返回数据")
    private Object data;

    private AjaxResponse() {
    }

    /**
     * 请求成功的响应，不带查询数据（用于删除、修改、新增接口）
     *
     * @return AjaxResponse
     */
    public static AjaxResponse success() {
        AjaxResponse ajaxResponse = new AjaxResponse();
        ajaxResponse.setCode(200);
        ajaxResponse.setMessage("请求响应成功!");
```

```
        return ajaxResponse;
    }

    /**
     * 请求成功的响应，带有查询数据（用于数据查询接口）
     *
     * @param obj obj
     * @return AjaxResponse
     */
    public static AjaxResponse success(Object obj) {
        AjaxResponse ajaxResponse = new AjaxResponse();
        ajaxResponse.setCode(200);
        ajaxResponse.setMessage("请求响应成功!");
        ajaxResponse.setData(obj);
        return ajaxResponse;
    }

    /**
     * 请求成功的响应，带有查询数据（用于数据查询接口）
     *
     * @param obj     obj
     * @param message message
     * @return AjaxResponse
     */
    public static AjaxResponse success(Object obj, String
message) {
        AjaxResponse ajaxResponse = new AjaxResponse();
        ajaxResponse.setCode(200);
        ajaxResponse.setMessage(message);
        ajaxResponse.setData(obj);
        return ajaxResponse;
    }
}
```

```
package top.mqxu.boot.basic.entity;

import com.fasterxml.jackson.annotation.JsonFormat;
import com.fasterxml.jackson.annotation.JsonInclude;
import io.swagger.annotations.ApiModel;
```

```java
import io.swagger.annotations.ApiModelProperty;
import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Data;
import lombok.NoArgsConstructor;

import java.util.Date;
import java.util.List;

/**
 * @description: Article
 * @author: mqxu
 * @since: 2021-03-05
 **/
@Data
@Builder
@AllArgsConstructor
@NoArgsConstructor
//@JsonPropertyOrder(value = {"content", "title"})
@ApiModel("文章基本信息")
public class Article {
    //@JsonIgnore
    @ApiModelProperty("id")
    private Integer id;

    //@JsonProperty("name")
    @ApiModelProperty("作者")
    private String author;

    @ApiModelProperty("标题")
    private String title;

    @ApiModelProperty("内容")
    private String content;

    @ApiModelProperty("创建时间")
    @JsonInclude(JsonInclude.Include.NON_NULL)
    @JsonFormat(pattern = "yyyy-MM-dd HH:mm:ss", timezone
= "GMT+8")
    private Date createTime;

    @ApiModelProperty("读者列表")
```

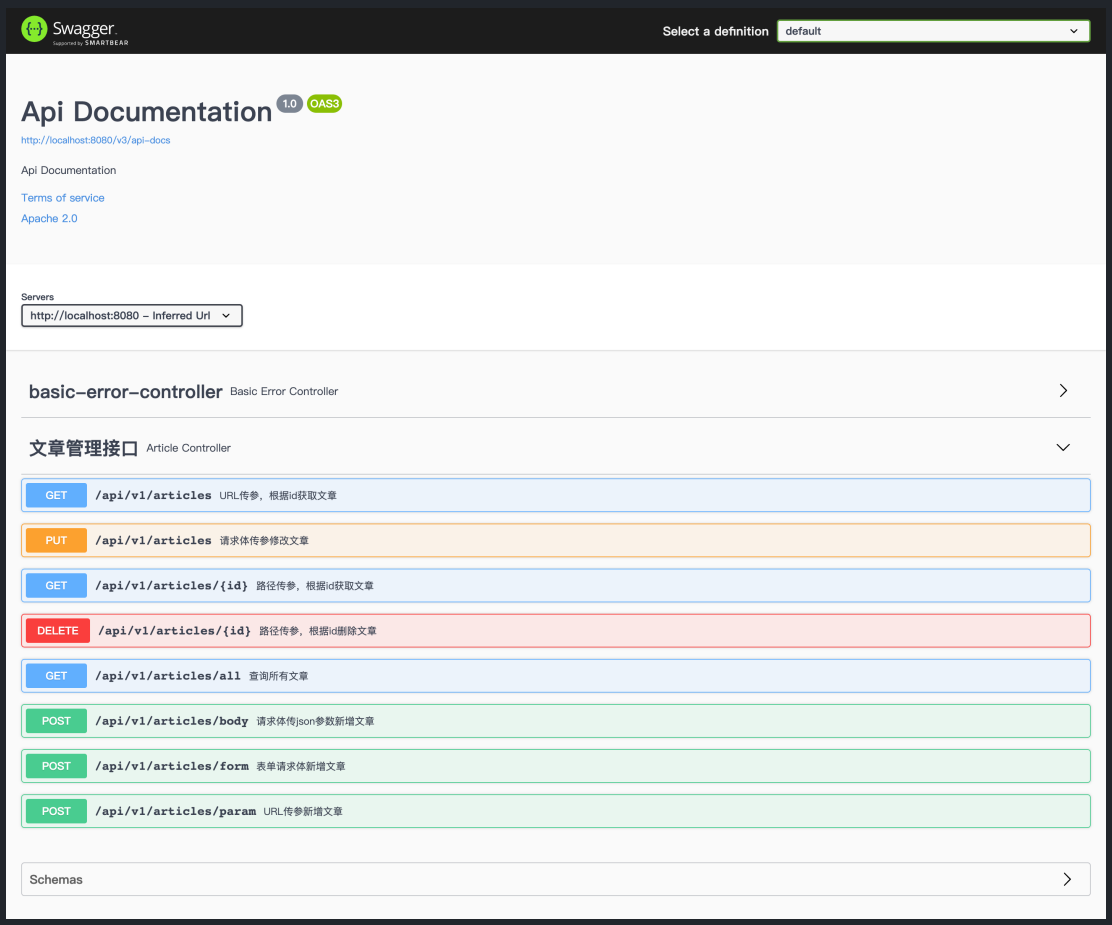
```
    private List<ArticleReader> readers;
}
```

四、生产环境下如何禁用swagger2

启动服务，浏览器输入

http://localhost:8080/swagger-ui/index.html

效果如图



每个接口都可以点开在线测试