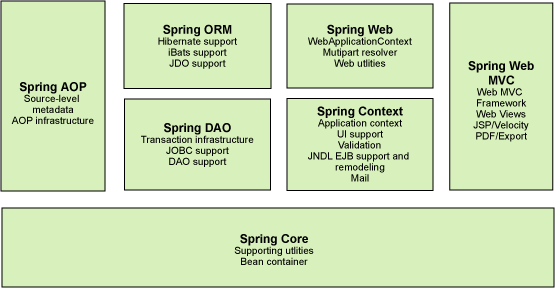
## Spring Boot介绍

Spring Framework

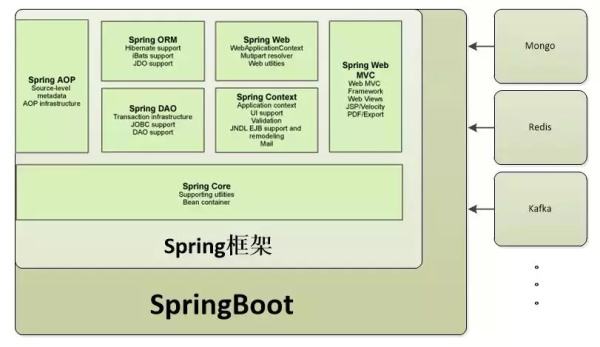


Spring 框架的每个模块（或组件）都可以单独存在，或者与其他一个或多个模块联合实现。每个模块的功能如下：

* **Spring Core**：核心容器提供 Spring 框架的基本功能。核心容器的主要组件是 BeanFactory，它是工厂模式的实现。BeanFactory 使用控制反转 （IOC） 模式将应用程序的配置和依赖性规范与实际的应用程序代码分开。
* **Spring Context**：Spring 上下文是一个配置文件，向 Spring 框架提供上下文信息。Spring 上下文包括企业服务，例如 JNDI、EJB、电子邮件、国际化、校验和调度功能。
* **Spring AOP**：通过配置管理特性，Spring AOP 模块直接将面向方面的编程功能集成到了 Spring 框架中。所以，可以很容易地使 Spring 框架管理的任何对象支持 AOP。Spring AOP 模块为基于 Spring 的应用程序中的对象提供了事务管理服务。通过使用 Spring AOP，不用依赖 EJB 组件，就可以将声明性事务管理集成到应用程序中。
* **Spring DAO**：JDBC DAO 抽象层提供了有意义的异常层次结构，可用该结构来管理异常处理和不同数据库供应商抛出的错误消息。异常层次结构简化了错误处理，并且极大地降低了需要编写的异常代码数量（例如打开和关闭连接）。Spring DAO 的面向 JDBC 的异常遵从通用的 DAO 异常层次结构。
* **Spring ORM**：Spring 框架插入了若干个 ORM 框架，从而提供了 ORM 的对象关系工具，其中包括 JDO、Hibernate 和 iBatis SQL Map。所有这些都遵从 Spring 的通用事务和 DAO 异常层次结构。
* **Spring Web**：Web 上下文模块建立在应用程序上下文模块之上，为基于 Web 的应用程序提供了上下文。所以，Spring 框架支持与 Jakarta Struts 的集成。Web 模块还简化了处理多部分请求以及将请求参数绑定到域对象的工作。
* **Spring MVC**：MVC 框架是一个全功能的构建 Web 应用程序的 MVC 实现。通过策略接口，MVC 框架变成为高度可配置的，MVC 容纳了大量视图技术，其中包括 JSP、Velocity、Tiles、iText 和 POI。

Spring 框架的功能可以用在任何 J2EE 服务器中，大多数功能也适用于不受管理的环境。Spring 的核心要点是：支持不绑定到特定 J2EE 服务的可重用业务和数据访问对象。毫无疑问，这样的对象可以在不同 J2EE 环境 （Web 或 EJB）、独立应用程序、测试环境之间重用。

Spring Boot



Spring Boot是由Pivotal团队提供的全新框架，其设计目的是用来简化新Spring应用的初始搭建以及开发过程。该框架使用了特定的方式来进行配置，从而使开发人员不再需要定义样板化的配置。通过这种方式，Spring Boot致力于在蓬勃发展的快速应用开发领域(rapid application development)成为领导者。

Spring boot并不是一个全新的框架，它不是spring解决方案的一个替代品，而是spring的一个封装。所以，你以前可以用spring做的事情，现在用spring boot都可以做。现在流行微服务与分布式系统，springboot就是一个非常好的微服务开发框架，你可以使用它快速的搭建起一个系统。同时，你也可以使用Spring Cloud（Spring Cloud是一个基于Spring Boot实现的云应用开发工具）来搭建一个分布式的网站。

Spring Boot使创建独立的、基于Spring的生产级应用程序变得很容易，您可以运行这些应用程序。我们对Spring平台和第三方库有自己的看法，这样您就可以轻松入门了。大多数Spring启动应用程序只需要很少的Spring配置。您可以使用Spring Boot创建可以通过使用java -jar或更传统的war部署启动的Java应用程序。我们还提供了一个运行“spring scripts” ”的命令行工具。

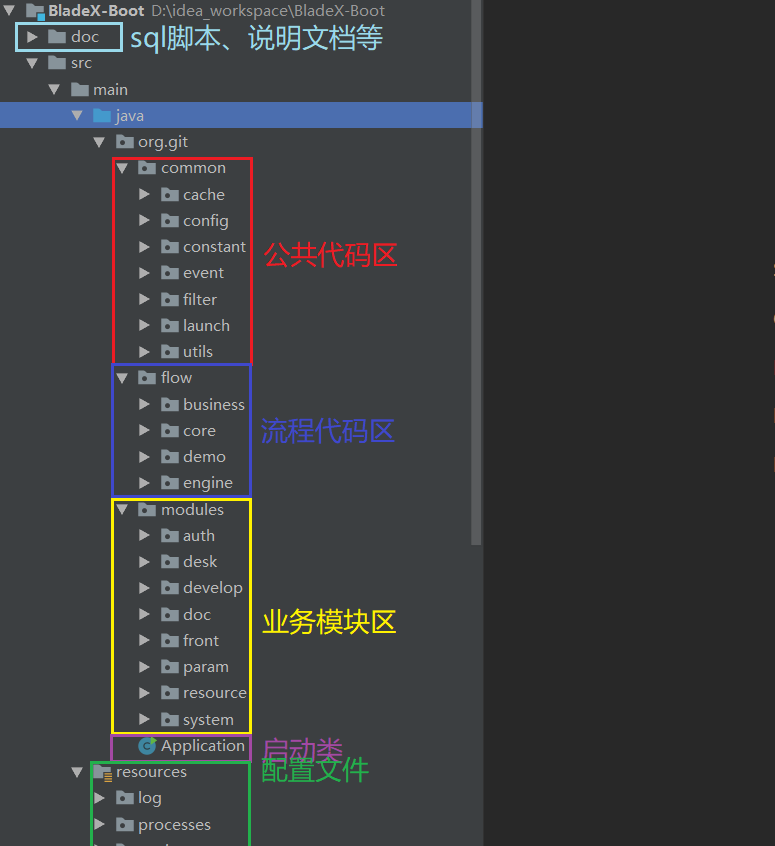
**特性：**

* 创建独立的Spring应用程序
* 直接嵌入Tomcat、Jetty或Undertow，不需要部署WAR文件
* 提供自有的“starter”依赖项来简化构建配置
* 尽可能自动配置Spring和第三方库
* 提供可用于生产的特性，如度量标准、健康状况检查和外部化配置
* 完全不需要代码生成，也不需要XML配置

集成框架一览表

|  |  |  |  |
| --- | --- | --- | --- |
| 组织机构 | 名称 | 版本 | 官网 |
| lmax | Disruptor | 3.4.2 | <http://lmax-exchange.github.io/disruptor/> |
| SmartBear | Swagger | 2.9.2 | [https://swagger.io](https://swagger.io/) |
| baomidou | MyBatis-Plus | 3.1.2 | <https://mybatis.plus/> |
| MyBatis | MyBatis | 3.5.1 | <https://mybatis.org/mybatis-3/zh/index.html> |
| Apache | Shiro | 1.4.1 | <http://shiro.apache.org/> |
| Google | Protobuf | 3.1.0 | <https://github.com/protocolbuffers/protobuf/releases> |
| David Yu | Protostuff | 1.6.0 | <https://github.com/protostuff/protostuff> |
| codecentric | Spring Boot Admin | 2.1.5 | <https://github.com/codecentric/spring-boot-admin> |
| pivotal | Micrometer | 1.3.0 | <http://micrometer.io/> |
| pivotal | Spring Boot Actuator | 2.1.7 | <https://docs.spring.io/spring-boot/docs/current/actuator-api/html/> |
| pivotal | Spring Boot | 2.1.7 | <https://spring.io/projects/spring-boot> |
| pivotal | Spring IO Platform | Cairo-SR7 | <https://spring.io/projects/platform> |
| pivotal | Spring Framework | 5.2.0 | <https://spring.io/projects/spring-framework> |
| Apache | Maven Plugin | 3.8.1 | <http://maven.apache.org/plugin-tools/maven-plugin-plugin/> |
| Apache | JWT | 0.9.1 | <https://jwt.io/> |
| Apache | Druid | 1.1.18 | <https://github.com/alibaba/druid> |
| QOS.ch | Slf4j |  | <http://www.slf4j.org/> |
| Apache | Log4j2 | 2.1.7 | <http://logging.apache.org/log4j/2.x/> |
| Terracotta | EhCache | 2.10.5 | <https://github.com/ehcache> |
| Apache | Velocity | 1.7 | <http://velocity.apache.org/> |
| Square | OkHttp | 3.11.0 | <https://square.github.io/okhttp/> |
| Netflix | Feign | 10.1 | <https://github.com/OpenFeign/feign> |
| Netflix | Hystrix | 1.5.11 | <https://github.com/Netflix/Hystrix> |
| Alibaba | Nacos | 1.1.3 | <https://nacos.io/zh-cn/> |
| Alibaba | Sentinel | 1.6.3 | <https://github.com/alibaba/Sentinel> |
| Google | Guava | 26.0 | <https://github.com/google/guava> |
| Netflix | Curator | 4.0.1 | <http://curator.apache.org/> |
| minio | minio-java | 6.0.7 | [https://github.com/minio/minio-java https://docs.min.io/cn/](https://github.com/minio/minio-java) |
| 七牛云 | QiNiu | 7.2.23 | <https://developer.qiniu.com/sdk#official-sdk> |
| RedisLabs | Redis | 5.0 | <https://redis.io/> |
| Flowable | Flowable | 6.4.2 | <https://www.flowable.org/> |
| projectlombok | lombok | 1.16.20 | <https://projectlombok.org/> |
| Igor Sysoev | Nginx | 1.14.2 | <http://nginx.org/en/download.html> |
| Apache | Maven | 3.6.1 | <http://maven.apache.org/> |
| XStream | xstream | 1.4.11.1 | <http://x-stream.github.io/download.html> |
| hibernate | hibernate-validator | 6.0.17.Final | [https://github.com/hibernate/hibernate-validator http://hibernate.org/validator/](https://github.com/hibernate/hibernate-validator) |

目录结构说明



核心工具类

框架已集成各类通用工具，可根据业务开发需要，通过类名.方法名直接进行静态调用。

1. org.springblade.core.tool.utils.AesUtil
2. org.springblade.core.tool.utils.AntPathFilter
3. org.springblade.core.tool.utils.Base64Util
4. org.springblade.core.tool.utils.BeanUtil
5. org.springblade.core.tool.utils.CharPool
6. org.springblade.core.tool.utils.Charsets
7. org.springblade.core.tool.utils.ClassUtil
8. org.springblade.core.tool.utils.CollectionUtil
9. org.springblade.core.tool.utils.ConcurrentDateFormat
10. org.springblade.core.tool.utils.ConvertUtil
11. org.springblade.core.tool.utils.DatatypeConverterUtil
12. org.springblade.core.tool.utils.DateTimeUtil
13. org.springblade.core.tool.utils.DateUtil
14. org.springblade.core.tool.utils.DigestUtil
15. org.springblade.core.tool.utils.Exceptions
16. org.springblade.core.tool.utils.FileUtil
17. org.springblade.core.tool.utils.Func
18. org.springblade.core.tool.utils.ImageUtil
19. org.springblade.core.tool.utils.IntegerPool
20. org.springblade.core.tool.utils.IoUtil
21. org.springblade.core.tool.utils.Lazy
22. org.springblade.core.tool.utils.NumberUtil
23. org.springblade.core.tool.utils.ObjectUtil
24. org.springblade.core.tool.utils.OkHttpUtil
25. org.springblade.core.tool.utils.PathUtil
26. org.springblade.core.tool.utils.PlaceholderUtil
27. org.springblade.core.tool.utils.ProtostuffUtil
28. org.springblade.core.tool.utils.RandomType
29. org.springblade.core.tool.utils.ReflectUtil
30. org.springblade.core.tool.utils.RegexUtil
31. org.springblade.core.tool.utils.ResourceUtil
32. org.springblade.core.tool.utils.RuntimeUtil
33. org.springblade.core.tool.utils.SpringUtil
34. org.springblade.core.tool.utils.StringPool
35. org.springblade.core.tool.utils.StringUtil
36. org.springblade.core.tool.utils.SuffixFileFilter
37. org.springblade.core.tool.utils.ThreadUtil
38. org.springblade.core.tool.utils.UrlUtil
39. org.springblade.core.tool.utils.Version
40. org.springblade.core.tool.utils.WebUtil
41. org.springblade.core.tool.utils.XmlUtil

## 自定义注解

@PreAuth注解解析

• 查询oauth授权用户，并关联查询用户基本信息（blade-auth/oauth/token）

• 用户信息中的角色名role\_name通过redis查询，如果查询失败或错误，导致Blade-Auth解析失败。

• Blade-Auth头报文中，通过oauth加密封装了token，用户Id、角色id等用户等授权信息。后台OAuth通过jwt框架解析。

• Blade-Auth头属性是 由 token\_type + refresh\_token 组成，通过接口（blade-auth/oauth/token）获取。

• org.springblade.auth.service.BladeUserDetailsServiceImpl loadUserByUsername

• org.springblade.core.secure.auth.AuthFun hasAnyRole

• org.springblade.core.secure.aspect.AuthAspect preAuth

• 授权校验拦截器 org.springblade.gateway.filter.AuthFilter

• org.springblade.core.secure.utils.SecureUtil

/\*\*

\*解析jsonWebToken

\*

\*@paramjsonWebTokenjsonWebToken

\*@returnClaims

\*/

Public static Claim sparseJWT(StringjsonWebToken){

returnJwtUtil.parseJWT(jsonWebToken);

}

• 解析工具

org.springblade.core.jwt.JwtUtil

## Spring Boot注解

使用注解的优势：

* 采用纯java代码，不在需要配置繁杂的xml文件
* 在配置中也可享受面向对象带来的好处
* 类型安全对重构可以提供良好的支持
* 减少复杂配置文件的同时亦能享受到springIoC容器提供的功能

@Configuration

@Configuration：相当于传统的xml配置文件，如果有些第三方库需要用到xml文件，建议仍然通过@Configuration类作为项目的配置主类——可以使用@ImportResource注解加载xml配置文件。

@Import：用来导入其他配置类。

@ImportResource：用来加载xml配置文件。

@Configuration并不是一个新的注解，在Spring3.0时已经存在了。@Configuration用于定义配置类，可替换xml配置文件，被注解的类内部包含有一个或多个被@Bean注解的方法，这些方法将会被AnnotationConfigApplicationContext或AnnotationConfigWebApplicationContext类进行扫描，并用于构建bean定义，初始化Spring容器。

注意：@Configuration注解的配置类有如下要求：

* @Configuration不可以是final类型；
* @Configuration不可以是匿名类；
* 嵌套的configuration必须是静态类。

@Configuration用法：

用@Configuration加载spring@Configuration配置spring并启动spring容器

@Configuration启动容器+@Bean注册Bean@Configuration启动容器+@Component注册Bean

使用 AnnotationConfigApplicationContext 注册 AppContext 类的两种方法配置Web应用程序(web.xml中配置AnnotationConfigApplicationContext)

2.组合多个配置类

在@configuration中引入spring的xml配置文件在@configuration中引入其它注解配置@configuration嵌套（嵌套的Configuration必须是静态类）

3.@EnableXXX注解

4.@Profile逻辑组配置

5.使用外部变量

@Conditional

@Conditional是spring4就提供出来的基于条件的注解。该注解可以根据是否满足某一个特定条件来决定是否创建某个特定的Bean。

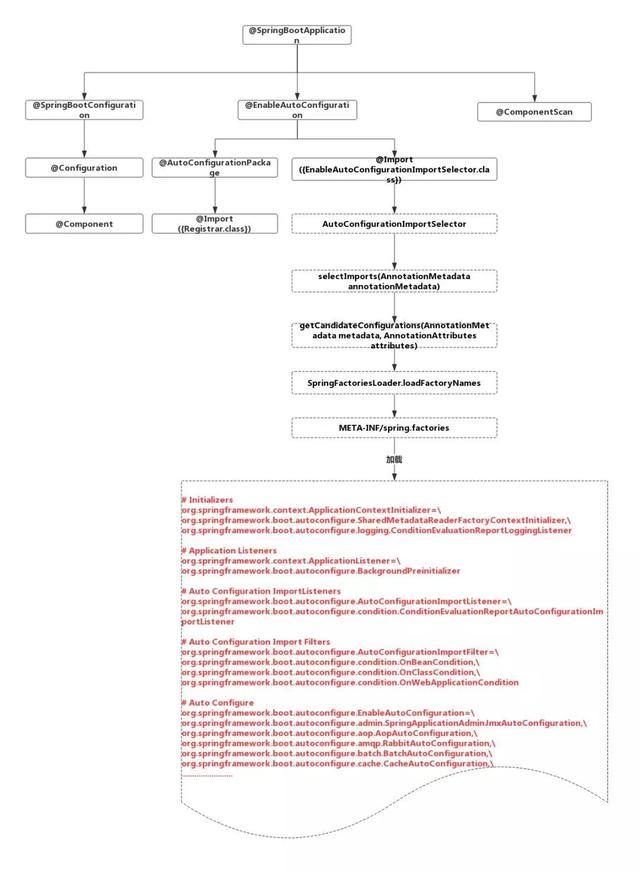
比如可以通过@Conditional注解来实现只有当某个Bean被已创建时（存在时）才会创建另外一个Bean，这样就可以依据这一特定的条件来控制Bean的创建行为，这样的话我们就可以利用这样一个特性来实现一些自动的配置。

@Conditional是springboot实现自动配置的关键基础能力。在此基础上，springboot又创建了多个适用于不同场景的组合条件注解。

@ConditionalOnBean：当上下文中有指定Bean的条件下进行实例化。@ConditionalOnMissingBean：当上下文没有指定Bean的条件下进行实例化。@ConditionalOnClass：当classpath类路径下有指定类的条件下进行实例化。@ConditionalOnMissingClass：当类路径下没有指定类的条件下进行实例化。@ConditionalOnWebApplication：当项目本身是一个Web项目时进行实例化。@ConditionalOnNotWebApplication：当项目本身不是一个Web项目时进行实例化。@ConditionalOnProperty：当指定的属性有指定的值时进行实例化。@ConditionalOnExpression：基于SpEL表达式的条件判断。@ConditionalOnJava：当JVM版本为指定的版本范围时进行实例化。@ConditionalOnResource：当类路径下有指定的资源时进行实例化。@ConditionalOnJndi：在JNDI存在时触发实例化。@ConditionalOnSingleCandidate：当指定的Bean在容器中只有一个，或者有多个但是指定了首选的Bean时触发实例化。具体用法还要具体查看文档或源码

@SpringBootApplication

这是SpringBoot最核心的注解。当然也是个组合注解。申明让spring boot自动给程序进行必要的配置，这个配置等同于：@Configuration ，@EnableAutoConfiguration 和 @ComponentScan 三个配置。



@Target(ElementType.TYPE)

@Retention(RetentionPolicy.RUNTIME)

@Documented

@Inherited

@SpringBootConfiguration

@EnableAutoConfiguration

@ComponentScan(excludeFilters = { @Filter(type = FilterType.CUSTOM, classes = TypeExcludeFilter.class), @Filter(type = FilterType.CUSTOM, classes = AutoConfigurationExcludeFilter.class) })

public @interface SpringBootApplication {

@ComponentScan

表示将该类自动发现扫描组件。个人理解相当于，如果扫描到有@Component、@Controller、@Service等这些注解的类，并注册为Bean，可以自动收集所有的Spring组件，包括@Configuration类。我们经常使用@ComponentScan注解搜索beans，并结合@Autowired注解导入。可以自动收集所有的Spring组件，包括@Configuration类。我们经常使用@ComponentScan注解搜索beans，并结合@Autowired注解导入。如果没有配置的话，Spring Boot会扫描启动类所在包下以及子包下的使用了@Service,@Repository等注解的类。

@EnableAutoConfiguration

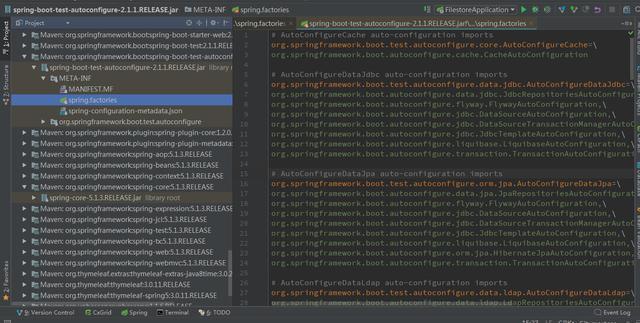
SpringBoot自动配置（auto-configuration）：尝试根据你添加的jar依赖自动配置你的Spring应用。例如，如果你的classpath下存在HSQLDB，并且你没有手动配置任何数据库连接beans，那么我们将自动配置一个内存型（in-memory）数据库”。你可以将@EnableAutoConfiguration或者@SpringBootApplication注解添加到一个@Configuration类上来选择自动配置。如果发现应用了你不想要的特定自动配置类，你可以使用@EnableAutoConfiguration注解的排除属性来禁用它们。

具体的实现，通过查看源码可知，是通过导入AutoConfigurationImportSelector类。

@Import(AutoConfigurationImportSelector.class)AutoConfigurationImportSelector最终是要加载所在jar下的META-INF/spring.factories文件。



该文件则声明了有哪些自动配置需要被Spring容器加载



配置的是注解和它对应的类。

从而Spring Boot应用程序就能自动加载Spring核心容器配置，以及其他依赖的项目组件配置，从而最终完成应用的自动初始化，通过这种方法就向开发者屏蔽了启动加载的过程。

@ResponseBody

表示该方法的返回结果直接写入HTTP response body中，一般在异步获取数据时使用，用于构建RESTful的api。在使用@RequestMapping后，返回值通常解析为跳转路径，加上@responsebody后返回结果不会被解析为跳转路径，而是直接写入HTTP response body中。比如异步获取json数据，加上@Responsebody后，会直接返回json数据。该注解一般会配合@RequestMapping一起使用。

@Controller

用于定义控制器类，在spring项目中由控制器负责将用户发来的URL请求转发到对应的服务接口（service层），一般这个注解在类中，通常方法需要配合注解@RequestMapping。

@RestController

用于标注控制层组件(如struts中的action)，@ResponseBody和@Controller的合集。

@RequestMapping

提供路由信息，负责URL到Controller中的具体函数的映射。用于类上，表示类中的所有响应请求的方法都是以该地址作为父路径。该注解有六个属性：

* params：指定request中必须包含某些参数值是，才让该方法处理。
* headers：指定request中必须包含某些指定的header值，才能让该方法处理请求。
* value：指定请求的实际地址，指定的地址可以是URI Template 模式
* method：指定请求的method类型， GET、POST、PUT、DELETE等
* consumes：指定处理请求的提交内容类型（Content-Type），如application/json,text/html;
* produces：指定返回的内容类型，仅当request请求头中的(Accept)类型中包含该指定类型才返回

@RequestParam

@RequestParam：用在方法的参数前面。

String a =request.getParameter(“a”)。

@PathVariable

路径变量

@Autowired

自动导入依赖的bean。byType方式。把配置好的Bean拿来用，完成属性、方法的组装，它可以对类成员变量、方法及构造函数进行标注，完成自动装配的工作。当加上（required=false）时，就算找不到bean也不报错。

@Service

一般用于修饰service层的组件

@Repository

使用@Repository注解可以确保DAO或者repositories提供异常转译，这个注解修饰的DAO或者repositories类会被ComponetScan发现并配置，同时也不需要为它们提供XML配置项。

@Component

泛指组件，当组件不好归类的时候，我们可以使用这个注解进行标注。

@Resource

@Resource(name=”name”,type=”type”)：没有括号内内容的话，默认byName。与@Autowired干类似的事。

@Bean：用@Bean标注方法等价于XML中配置的bean。

@Value：注入Spring boot application.properties配置的属性的值。示例代码：

@Inject：等价于默认的@Autowired，只是没有required属性；

@Bean：相当于XML中的,放在方法的上面，而不是类，意思是产生一个bean,并交给spring管理。

@Qualifier：当有多个同一类型的Bean时，可以用@Qualifier(“name”)来指定。与@Autowired配合使用。@Qualifier限定描述符除了能根据名字进行注入，但能进行更细粒度的控制如何选择候选者，具体使用方式如下：

## 异常注解

@ControllerAdvice

包含@Component。可以被扫描到。统一处理异常。@ControllerAdvice是组件注解，他使得其实现类能够被classpath扫描自动发现，如果应用是通过MVC命令空间或MVC Java编程方式配置，那么该特性默认是自动开启的。注解@ControllerAdvice的类可以拥有@ExceptionHandler, @InitBinder或 @ModelAttribute注解的方法，并且这些方法会被应用到控制器类层次的所有@RequestMapping方法上。

@ExceptionHandler

@ExceptionHandler（Exception.class）：用在方法上面表示遇到这个异常就执行以下方法。

@RestControllerAdvice

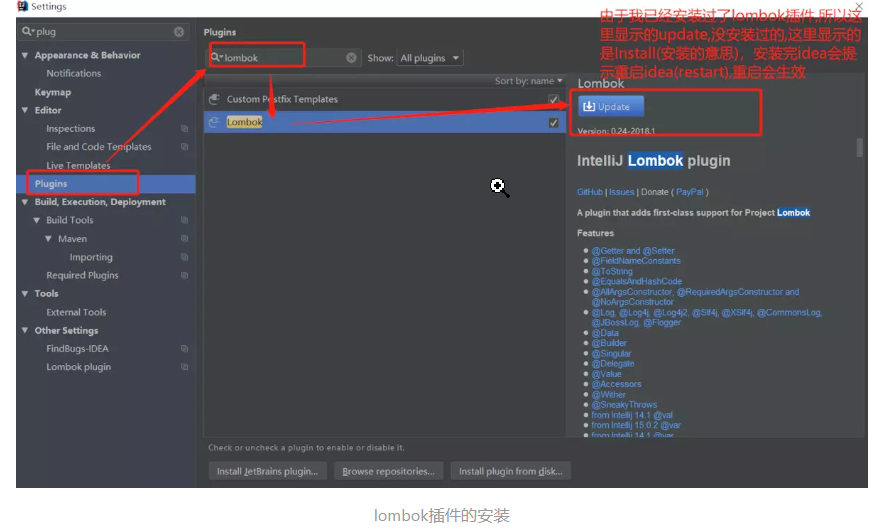
如果全部异常处理返回json，那么可以使用 @RestControllerAdvice 代替 @ControllerAdvice ，这样在方法上就可以不需要添加 @ResponseBody。。类似于 @RestController 与 @Controller的区别

## 日志注解

@Slf4j

声明：如果不想每次都写private  final Logger logger = LoggerFactory.getLogger(当前类名.class); 可以用注解@Slf4j;

1.使用idea首先需要安装Lombok插件;



2.在pom文件加入lombok的依赖

<dependency>

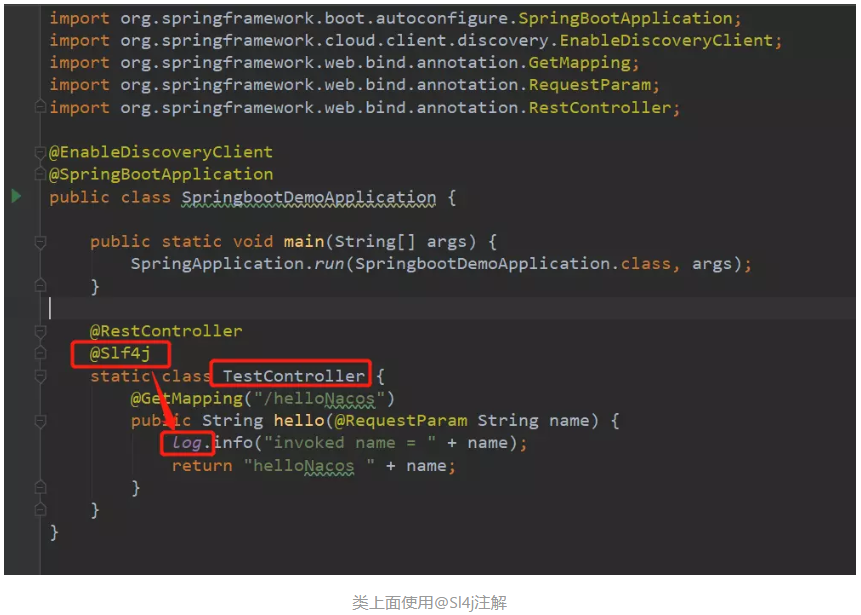
   <groupId>org.projectlombok</groupId>

   <artifactId>lombok</artifactId>

    <version>1.16.16</version><!--版本号自己选一个就行-->

</dependency>

3.类上面添加@Sl4j注解,然后使用log打印日志;



springboot的核心就是注解。springboot通过各种组合注解，极大地简化了spring项目的搭建和开发。

在springboot中有一些注解是其中的关键，必须掌握。

## Swagger注解

@Api

用于类；表示标识这个类是swagger的资源

tags="说明该类的作用，可以在UI界面上看到的注解"

value="该参数没什么意义，在UI界面上也看到，所以不需要配置"

@ApiOperation

用于方法；用在请求的方法上，说明方法的用途、作用，表示一个http请求的操作

value="说明方法的用途、作用"

notes="方法的备注说明"

@ApiParam

用于方法，参数，字段说明；表示对参数的添加元数据（说明或是否必填等）

@ApiModel

用于类；表示对类进行说明，用于参数用实体类接收

@ApiModelProperty

用于方法，字段；表示对model属性的说明或者数据操作更改

@ApiIgnore

用于类，方法，方法参数；表示这个方法或者类被忽略

@ApiImplicitParams

@ApiImplicitParams用在请求的方法上，表示一组参数说明

@ApiImplicitParam

用于方法；表示单独的请求参数

用在@ApiImplicitParams注解中，指定一个请求参数的各个方面

name：参数名

value：参数的汉字说明、解释

required：参数是否必须传

paramType：参数放在哪个地方

· header --> 请求参数的获取：@RequestHeader

· query --> 请求参数的获取：@RequestParam

· path（用于restful接口）--> 请求参数的获取：@PathVariable

· body（不常用）

· form（不常用）

dataType：参数类型，默认String，其它值dataType="Integer"

defaultValue：参数的默认值

## Spring缓存注解

从3.1开始，Spring引入了对Cache的支持。其使用方法和原理都类似于Spring对事务管理的支持。Spring Cache是作用在方法上的，其核心思想是这样的：当我们在调用一个缓存方法时会把该方法参数和返回结果作为一个键值对存放在缓存中，等到下次利用同样的参数来调用该方法时将不再执行该方法，而是直接从缓存中获取结果进行返回。所以在使用Spring Cache的时候我们要保证我们缓存的方法对于相同的方法参数要有相同的返回结果。

Spring为我们提供了几个注解来支持Spring Cache。其核心主要是@Cacheable和@CacheEvict。使用@Cacheable标记的方法在执行后Spring Cache将缓存其返回结果，而使用@CacheEvict标记的方法会在方法执行前或者执行后移除Spring Cache中的某些元素。下面我们将来详细介绍一下Spring基于注解对Cache的支持所提供的几个注解。

使用Spring Cache需要我们做两方面的事：

* 声明某些方法使用缓存
* 配置Spring对Cache的支持

和Spring对事务管理的支持一样，Spring对Cache的支持也有基于注解和基于XML配置两种方式。下面我们先来看看基于注解的方式。

@Cacheable

@Cacheable可以标记在一个方法上，也可以标记在一个类上。当标记在一个方法上时表示该方法是支持缓存的，当标记在一个类上时则表示该类所有的方法都是支持缓存的。对于一个支持缓存的方法，Spring会在其被调用后将其返回值缓存起来，以保证下次利用同样的参数来执行该方法时可以直接从缓存中获取结果，而不需要再次执行该方法。Spring在缓存方法的返回值时是以键值对进行缓存的，值就是方法的返回结果，至于键的话，Spring又支持两种策略，默认策略和自定义策略，这个稍后会进行说明。需要注意的是当一个支持缓存的方法在对象内部被调用时是不会触发缓存功能的。@Cacheable可以指定三个属性，value、key和condition。

* **value属性指定Cache名称**

       value属性是必须指定的，其表示当前方法的返回值是会被缓存在哪个Cache上的，对应Cache的名称。其可以是一个Cache也可以是多个Cache，当需要指定多个Cache时其是一个数组。

   @Cacheable("cache1")//Cache是发生在cache1上的

   public User find(Integer id) {

      returnnull;

   }

   @Cacheable({"cache1", "cache2"})//Cache是发生在cache1和cache2上的

   public User find(Integer id) {

      returnnull;

   }

* **使用key属性自定义key**

       key属性是用来指定Spring缓存方法的返回结果时对应的key的。该属性支持SpringEL表达式。当我们没有指定该属性时，Spring将使用默认策略生成key。我们这里先来看看自定义策略，至于默认策略会在后文单独介绍。

       自定义策略是指我们可以通过Spring的EL表达式来指定我们的key。这里的EL表达式可以使用方法参数及它们对应的属性。使用方法参数时我们可以直接使用“#参数名”或者“#p参数index”。下面是几个使用参数作为key的示例。

   @Cacheable(value="users", key="#id")

   public User find(Integer id) {

      returnnull;

   }

   @Cacheable(value="users", key="#p0")

   public User find(Integer id) {

      returnnull;

   }

   @Cacheable(value="users", key="#user.id")

   public User find(User user) {

      returnnull;

   }

   @Cacheable(value="users", key="#p0.id")

   public User find(User user) {

      returnnull;

   }

       除了上述使用方法参数作为key之外，Spring还为我们提供了一个root对象可以用来生成key。通过该root对象我们可以获取到以下信息。

|  |  |  |
| --- | --- | --- |
| 属性名称 | 描述 | 示例 |
| methodName | 当前方法名 | #root.methodName |
| method | 当前方法 | #root.method.name |
| target | 当前被调用的对象 | #root.target |
| targetClass | 当前被调用的对象的class | #root.targetClass |
| args | 当前方法参数组成的数组 | #root.args[0] |
| caches | 当前被调用的方法使用的Cache | #root.caches[0].name |

       当我们要使用root对象的属性作为key时我们也可以将“#root”省略，因为Spring默认使用的就是root对象的属性。如：

   @Cacheable(value={"users", "xxx"}, key="caches[1].name")

   public User find(User user) {

      returnnull;

   }

@CacheEvict

@CacheEvict是用来标注在需要清除缓存元素的方法或类上的。当标记在一个类上时表示其中所有的方法的执行都会触发缓存的清除操作。@CacheEvict可以指定的属性有value、key、condition、allEntries和beforeInvocation。其中value、key和condition的语义与@Cacheable对应的属性类似。即value表示清除操作是发生在哪些Cache上的（对应Cache的名称）；key表示需要清除的是哪个key，如未指定则会使用默认策略生成的key；condition表示清除操作发生的条件。下面我们来介绍一下新出现的两个属性allEntries和beforeInvocation。

**allEntries属性**

allEntries是boolean类型，表示是否需要清除缓存中的所有元素。默认为false，表示不需要。当指定了allEntries为true时，Spring Cache将忽略指定的key。有的时候我们需要Cache一下清除所有的元素，这比一个一个清除元素更有效率。

*@CacheEvict(value="users", allEntries=true)*

*public void delete(Integer id) {*

*System.out.println("delete user by id: " + id);*

*}*

**beforeInvocation属性**

清除操作默认是在对应方法成功执行之后触发的，即方法如果因为抛出异常而未能成功返回时也不会触发清除操作。使用beforeInvocation可以改变触发清除操作的时间，当我们指定该属性值为true时，Spring会在调用该方法之前清除缓存中的指定元素。

*@CacheEvict(value="users", beforeInvocation=true)*

*public void delete(Integer id) {*

*System.out.println("delete user by id: " + id);*

*}*

其实除了使用@CacheEvict清除缓存元素外，当我们使用Ehcache作为实现时，我们也可以配置Ehcache自身的驱除策略，其是通过Ehcache的配置文件来指定的。由于Ehcache不是本文描述的重点，这里就不多赘述了，想了解更多关于Ehcache的信息，请查看我关于Ehcache的专栏。

@CachePut

在支持Spring Cache的环境下，对于使用@Cacheable标注的方法，Spring在每次执行前都会检查Cache中是否存在相同key的缓存元素，如果存在就不再执行该方法，而是直接从缓存中获取结果进行返回，否则才会执行并将返回结果存入指定的缓存中。@CachePut也可以声明一个方法支持缓存功能。与@Cacheable不同的是使用@CachePut标注的方法在执行前不会去检查缓存中是否存在之前执行过的结果，而是每次都会执行该方法，并将执行结果以键值对的形式存入指定的缓存中。

@CachePut也可以标注在类上和方法上。使用@CachePut时我们可以指定的属性跟@Cacheable是一样的。

*@CachePut("users")//每次都会执行方法，并将结果存入指定的缓存中*

*public User find(Integer id) {*

*returnnull;*

*}*

@Caching

@Caching注解可以让我们在一个方法或者类上同时指定多个Spring Cache相关的注解。其拥有三个属性：cacheable、put和evict，分别用于指定@Cacheable、@CachePut和@CacheEvict。

*@Caching(cacheable = @Cacheable("users"), evict = { @CacheEvict("cache2"),*

*@CacheEvict(value = "cache3", allEntries = true) })*

*public User find(Integer id) {*

*returnnull;*

*}*

## lombok注解

@Data

使用这个注解，就不用再去手写Getter,Setter,equals,canEqual,hasCode,toString等方法了，注解后在编译时会自动加进去。

@AllArgsConstructor

使用后添加一个构造函数，该构造函数含有所有已声明字段属性参数

@NoArgsConstructor

使用后创建一个无参构造函数

@Builder

关于Builder较为复杂一些，Builder的作用之一是为了解决在某个类有很多构造函数的情况，也省去写很多构造函数的麻烦，在设计模式中的思想是：用一个内部类去实例化一个对象，避免一个类出现过多构造函数，

@RequiredArgsConstructor

会生成一个包含常量，和标识了NotNull的变量的构造方法。生成的构造方法是私有的private。

@ToString

自动生成toString()方法，默认情况，按顺序（以“,”分隔）打印你的类名称以及每个字段。也可以设置不包含哪些字段/@ToString(exclude = {“id”,”name”})



@Synchronized

给方法加上同步锁，建议直接写在代码中Synchronized

## JPA注解

@Entity

@Table(name=”“)：表明这是一个实体类。一般用于jpa这两个注解一般一块使用，但是如果表名和实体类名相同的话，@Table可以省略

@MappedSuperClass

用在确定是父类的entity上。父类的属性子类可以继承。

@NoRepositoryBean

一般用作父类的repository，有这个注解，spring不会去实例化该repository。

@Column

如果字段名与列名相同，则可以省略。

@Id

表示该属性为主键。

@GeneratedValue

@GeneratedValue(strategy = GenerationType.SEQUENCE,generator = “repair\_seq”)：表示主键生成策略是sequence（可以为Auto、IDENTITY、native等，Auto表示可在多个数据库间切换），指定sequence的名字是repair\_seq。

@SequenceGeneretor

@SequenceGeneretor(name = “repair\_seq”, sequenceName = “seq\_repair”, allocationSize = 1)：name为sequence的名称，以便使用，sequenceName为数据库的sequence名称，两个名称可以一致。

@Transient

表示该属性并非一个到数据库表的字段的映射,ORM框架将忽略该属性。如果一个属性并非数据库表的字段映射,就务必将其标示为@Transient,否则,ORM框架默认其注解为@Basic。@Basic(fetch=FetchType.LAZY)：标记可以指定实体属性的加载方式

@JsonIgnore

作用是json序列化时将Java bean中的一些属性忽略掉,序列化和反序列化都受影响。

@JoinColumn

@JoinColumn（name=”loginId”）：一对一：本表中指向另一个表的外键。一对多：另一个表指向本表的外键。

@OneToOne

对应hibernate配置文件中的一对一

@OneToMany

对应hibernate配置文件中的一对多

@ManyToOne

对应hibernate配置文件中的多对一

持久事件

当你在执行各种持久化方法的时候，实体的状态会随之改变，状态的改变会引发不同的生命周期事件。这些事件可以使用不同的注释符来指示发生时的回调函数。

@javax.persistence.PostLoad：加载后。

@javax.persistence.PrePersist：持久化前。

@javax.persistence.PostPersist：持久化后。

@javax.persistence.PreUpdate：更新前。

@javax.persistence.PostUpdate：更新后。

@javax.persistence.PreRemove：删除前。

@javax.persistence.PostRemove：删除后。

**1）数据库查询**

@PostLoad事件在下列情况下触发：

执行EntityManager.find()或getreference()方法载入一个实体后。

执行JPQL查询后。

EntityManager.refresh()方法被调用后。

**2）数据库插入**

@PrePersist和@PostPersist事件在实体对象插入到数据库的过程中发生：

@PrePersist事件在调用persist()方法后立刻发生，此时的数据还没有真正插入进数据库。

@PostPersist事件在数据已经插入进数据库后发生。

**3）数据库更新**

@PreUpdate和@PostUpdate事件的触发由更新实体引起：

@PreUpdate事件在实体的状态同步到数据库之前触发，此时的数据还没有真正更新到数据库。

@PostUpdate事件在实体的状态同步到数据库之后触发，同步在事务提交时发生。

**4）数据库删除**

@PreRemove和@PostRemove事件的触发由删除实体引起：

@PreRemove事件在实体从数据库删除之前触发，即在调用remove()方法删除时发生，此时的数据还没有真正从数据库中删除。

@PostRemove事件在实体从数据库中删除后触发。