springboot知识点

一.spring和springboot的关系

springboot的底层就是spring(4.0x),所以这么说，对spring有多精通，springboot就有多了解

二.springboot的特点

- springboot中内嵌了tomcat（spring的条件装配应用）
- 部署简单（只需要依赖jdk环境即可）
- springboot对依赖进行集成管理（解决在spring中整合遇到的依赖冲突问题）
- 快速整合第三方组件（导包---->配置）
- 是微服务组件的支撑（所有的微服务组件都需要依赖springboot）

springboot依赖管理

Java | 复制代码

```java
//springboot自定义的版本
<properties>
    <activemq.version>5.16.2</activemq.version>
    <antlr2.version>2.7.7</antlr2.version>
    <appengine-sdk.version>1.9.90</appengine-sdk.version>
    <artemis.version>2.17.0</artemis.version>
    <aspectj.version>1.9.7</aspectj.version>
    <assertj.version>3.19.0</assertj.version>
</properties>
//自己自定义版本pom.xml
<properties>
    <java.version>1.8</java.version>
</properties>
//定义引入版本
<dependencyManagement>
</dependencyManagement>
```

三.springboot底层注解

configuration	代表此类是一个配置类,注意新特性属性：proxyBeanMethods
Import	spring注解 提供了三种导入方式： .class { , , } ImportBeanDefinitionRegister
conditionalOnBean/Class	如果项目中有这个类，配置生效
conditionalMissingBean	如果项目没有这个类，配置生效
ConfigurationProperties	指定yml文件中的配置（prefix="myCar"）元数据 可以配合component
EnableConfigurationProperties	配合configurationProperties
Value	spring使用的填充值的注解 #{} ${} ""
ImportResource	导入一些其他的xml配置

四.自动装配原理

```java
//主类
@SpringBootApplication
@EnableScheduling
public class QuartzDemoApplication {
}

//springbootApplication充当的三个注解
@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan(excludeFilters = { @Filter(type = FilterType.CUSTOM, classes = TypeExcludeFilter.class),
        @Filter(type = FilterType.CUSTOM, classes = AutoConfigurationExcludeFilter.class) })
//注意这个是扫描bean的包  mapperScan扫描的是xml的包
public @interface SpringBootApplication {
}

//EnableAutoConfiguration
@Import(AutoConfigurationImportSelector.class)  自动装配的类
@Import(AutoConfigurationPackages.Registrar.class)  application 同级目录或子目录的引入
```

//重点：解释一下AutoConfigurationImportSelector

引入的AutoConfigurationImportSelector 实现DeferredImportSelector(变种的)接口实现方getImportGroup,判断是否返回一个DeferedImportSelector.Group这个类,如果返回：则是一个变种的selector,否则就是spring 中一个普通的selectImports(返回一个数组注入到IOC中),变种自动配置为一下几步

1. public class AutoConfigurationImportSelector implements DeferredImportSelector
2. @Override
   public Class<? extends Group> getImportGroup() {
       return AutoConfigurationGroup.class;
   }

3.process方法

Java | 复制代码

```java
@Override
public void process(AnnotationMetadata annotationMetadata,DeferredImportSelector,deferredImportSelector) {
            Assert.state(deferredImportSelector instanceof AutoConfigurationImportSelector,
                    () -> String.format("Only %s implementations are supported, got %s",
AutoConfigurationImportSelector.class.getSimpleName(),
deferredImportSelector.getClass().getName()));
            AutoConfigurationEntry autoConfigurationEntry = ((AutoConfigurationImportSelector) deferredImportSelector)
                    .getAutoConfigurationEntry(annotationMetadata);
            this.autoConfigurationEntries.add(autoConfigurationEntry);
            for (String importClassName : autoConfigurationEntry.getConfigurations()) {
                this.entries.putIfAbsent(importClassName, annotationMetadata);
            }
        }
```

4.getAutoConfigurationEntry获取所有的自动配置类

```
protected AutoConfigurationEntry getAutoConfigurationEntry(AnnotationMetadata annotationMetadata) {  annotationMetadata:
    if (!isEnabled(annotationMetadata)) {
        return EMPTY_ENTRY;
    }
    AnnotationAttributes attributes = getAttributes(annotationMetadata);  attributes:  size = 2
    List<String> configurations = getCandidateConfigurations(annotationMetadata, attributes);  configurations:  size = 1
    configurations = removeDuplicates(configurations);  configurations:  size = 131
    Set<String> exclusions = getExclusions(annotationMetadata, attributes);
    checkExcludedClasses(configurations, exclusions);
    configurations.removeAll(exclusions);
    configurations = getConfigurationClassFilter().filter(configurations);
    fireAutoConfigurationImportEvents(configurations, exclusions);
    return new AutoConfigurationEntry(configurations, exclusions);
}
```

```
dpoints
Variables
this = {AutoConfigurationImportSelector@3438}
annotationMetadata = {StandardAnnotationMetadata@3440} "com.qf.quartzdemo.QuartzDemoApplication"
attributes = {AnnotationAttributes@3468} size = 2
configurations = {Collections$UnmodifiableRandomAccessList@3472} size = 131
0 = "org.springframework.boot.autoconfigure.admin.SpringApplicationAdminJmxAutoConfiguration"
1 = "org.springframework.boot.autoconfigure.aop.AopAutoConfiguration"
2 = "org.springframework.boot.autoconfigure.amqp.RabbitAutoConfiguration"
3 = "org.springframework.boot.autoconfigure.batch.BatchAutoConfiguration"
4 = "org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration"
5 = "org.springframework.boot.autoconfigure.cassandra.CassandraAutoConfiguration"
6 = "org.springframework.boot.autoconfigure.context.ConfigurationPropertiesAutoConfiguration"
```

5.getCandidateConfigurations获取条件装配的配置类

```
*/
protected List<String> getCandidateConfigurations(AnnotationMetadata metadata, AnnotationAttributes attributes) {
    List<String> configurations = SpringFactoriesLoader.loadFactoryNames(getSpringFactoriesLoaderFactoryClass(),
            getBeanClassLoader());
    Assert.notEmpty(configurations, message: "No auto configuration classes found in META-INF/spring.factories. If you "
            + "are using a custom packaging, make sure that file is correct.");
    return configurations;
}
```

6.loadFactoryNames

```
*/
public static List<String> loadFactoryNames(Class<?> factoryType, @Nullable ClassLoader classLoader) {
    ClassLoader classLoaderToUse = classLoader;
    if (classLoaderToUse == null) {
        classLoaderToUse = SpringFactoriesLoader.class.getClassLoader();
    }
    String factoryTypeName = factoryType.getName();
    return loadSpringFactories(classLoaderToUse).getOrDefault(factoryTypeName, Collections.emptyList());
}
```

7.loadSpringFactories读取springFactories文件中所有的类

```
private static Map<String, List<String>> loadSpringFactories(ClassLoader classLoader) {
    Map<String, List<String>> result = cache.get(classLoader);
    if (result != null) {
        return result;
    }

    result = new HashMap<>();
    try {
        Enumeration<URL> urls = classLoader.getResources(FACTORIES_RESOURCE_LOCATION);
        while (urls.hasMoreElements()) {
            URL url = urls.nextElement();
            UrlResource resource = new UrlResource(url);
            Properties properties = PropertiesLoaderUtils.loadProperties(resource);
            for (Map.Entry<?, ?> entry : properties.entrySet()) {
                String factoryTypeName = ((String) entry.getKey()).trim();
                String[] factoryImplementationNames =
                        StringUtils.commaDelimitedListToStringArray((String) entry.getValue());
                for (String factoryImplementationName : factoryImplementationNames) {
                    result.computeIfAbsent(factoryTypeName, key -> new ArrayList<>())
```

五.springboot元数据配置怎么写

@ConfigurationProperties(perfix="person") @EnableConfigurationProperties

@Componet @ConfigurationProperties(perfix="person")

public class person(){

}

配置文件xml:基本语法

1.字符串："

2.对象:

pet:

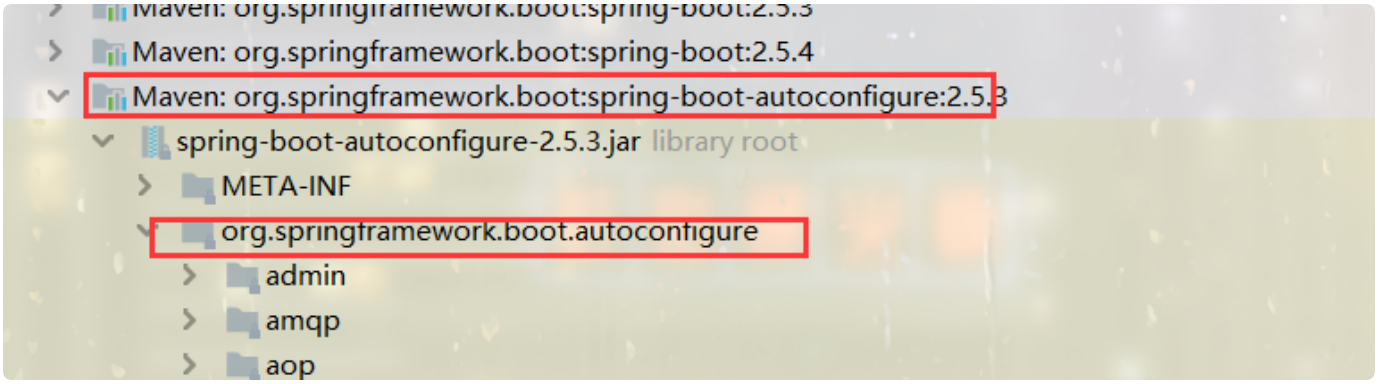
　属性1：2

　属性2： 2

3.数组： 【1，2】

4.map:{k1,v1},{k2,v2}

六.实用技巧：自动配置类我们在哪里可以找到



请求处理映

@RequestMapping和Rest风格 ----->HiddenHttpMethodFilter

ERROR_EXCEPTION_ATTRIBUTE=_method

表单请求不可以处理put delete请求,单springboot支持，我们必须按照springboot的约定找

```java
    /** Default method parameter: {@code _method}. */
    public static final String DEFAULT_METHOD_PARAM = "_method";

    private String methodParam = DEFAULT_METHOD_PARAM;
//方法
protected void doFilterInternal(HttpServletRequest request,
HttpServletResponse response, FilterChain filterChain)
            throws ServletException, IOException {

        HttpServletRequest requestToUse = request;

        if ("POST".equals(request.getMethod()) &&
request.getAttribute(WebUtils.ERROR_EXCEPTION_ATTRIBUTE) == null) {
            String paramValue = request.getParameter(this.methodParam);
            if (StringUtils.hasLength(paramValue)) {
                String method = paramValue.toUpperCase(Locale.ENGLISH);
                if (ALLOWED_METHODS.contains(method)) {
                    requestToUse = new HttpMethodRequestWrapper(request,
method);
                }
            }
        }

        filterChain.doFilter(requestToUse, response);
    }
//当然根据springboot条件装配原理可以自定义
//第一种方式
@Bean
public HiddenHttpMethodFilter hiddenHttpMethodFilter(){
    HiddenHttpMethodFilter hiddenHttpMethodFilter=new
HiddenHttpMethodFilter();
    hiddenHttpMethodFilter.setMethodParam("_m");
}
//yml配置方式
    @ConditionalOnMissingBean({HiddenHttpMethodFilter.class})
    @ConditionalOnProperty(
        prefix = "spring.mvc.hiddenmethod.filter",
        name = {"enabled"}
    )
```

参数映射

@PathVariable　@RequestHeader @RequestParam Map<String,String>接收

@RequestBody //表单 json数据
@cookieValue
@RequestAttribute("msg") String msg 参数赋值

七.弄清springboot混乱的日志体系

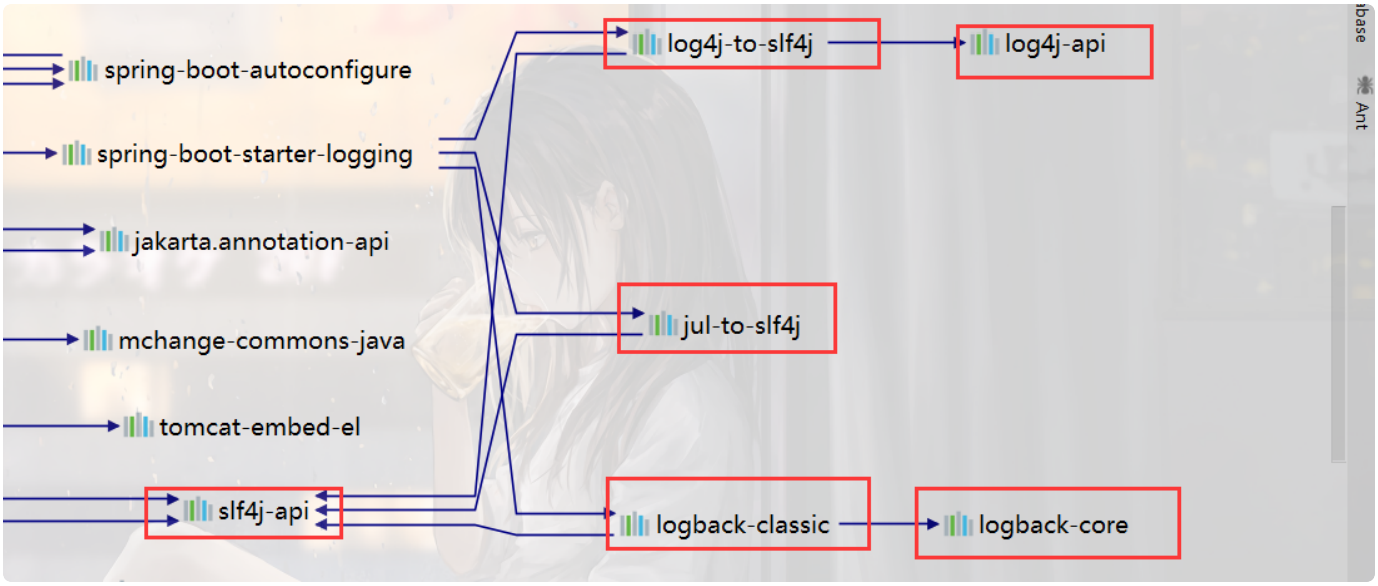
1.说个故事：日志发展史

- 从前有一个程序员叫张三，老程序员了，jdk1.3----->system.out.print()----->记录日志
- 一天，他就自己封装了一个工具类logutil 可以顺利追踪了
- 某一天，发现请求量太大，文件太大，分块存储，分日期存储 按天迭代，按物理迭代 loginfo
- 用户出现异常 loginfo能不能及时给我发送邮件,能不能按照等级划分，能不能自由控制格式
- 张三刻苦研发，黄天不负有心人，开发了一个叫log4j的日志框架，并且很大方开源
- 结果市场上备受欢迎，许多人也协同他开发 出了很多版本log4jSimple,log4jnop 边冲日志
- 张三觉得自己很有成就感，觉得自己可以与sun公司合作了，结果被sun公司拒绝了 ,最后去了apatch基金会
- sun公司毕竟是业界大佬，就自己研发了一款jul的日志框架，不过log4j已经占领了市场，所以两类持平
- sun公司很聪明，发现市面日志框架很多就开发了一款门面jcl 可以整合多个日志框架
- 张三也意识到了这一点，独自开发日志门面slf4j非常强大可以整合日志框架还可以整合sun公司的日志门面
- 当然slf4j分为适配器和桥接器两部分，apache觉得log4j性能很差，就优化了变成了log4j2，张三有意识到自己错误，有开发了一款优良的logback日志框架。

2.说说日志框架和日志门面

日志框架	日志门面
log4j log4jsimple log4jnop	
logback	slf4j
log4j2	
jul	jcl

最常用的就是蓝色部分的
springboot底层默认使用了logback



怎么更换logback为log4j系列

1. 要将logback的桥接器排除 logback-classic
2. 添加log4j的桥接器 slf4j-log4j12
3. 配置文件log4j

怎么更换logback为log4j2系列

1. 排除其他桥接器 spring-boot-starter-logging
2. 引入spring-boot-starter-log4j2

桥接器	适配器（to）
logback-classic logback	
slf4j-log4j12 slf4j	log4j-to-slf4j
jdk自带 jul	jul-to-slf4j

八.说说start-web自动配置类

目标：webMvcConfigurer mvcAutoConfiguration

- 扫包不用说@ComponemtScan
- 视图解析器 mvcAutoConfiguration--viewResolver

Java 复制代码

```java
        //原生的视图解析器
        @Bean
        @ConditionalOnMissingBean
        public InternalResourceViewResolver defaultViewResolver() {
            InternalResourceViewResolver resolver = new InternalResourceViewResolver();
            resolver.setPrefix(this.mvcProperties.getView().getPrefix());
            resolver.setSuffix(this.mvcProperties.getView().getSuffix());
            return resolver;
        }
        //自定义驶入解析器  会根据hander方法返回的视图名称去ioc容器找一个对应返回值的bean
        @Bean
        @ConditionalOnBean({View.class})
        @ConditionalOnMissingBean
        public BeanNameViewResolver beanNameViewResolver() {
            BeanNameViewResolver resolver = new BeanNameViewResolver();
            resolver.setOrder(2147483637);
            return resolver;
        }
      //不做解析，让其他解析器解析
        @Bean
        @ConditionalOnBean({ViewResolver.class})
        @ConditionalOnMissingBean(
            name = {"viewResolver"},
            value = {ContentNegotiatingViewResolver.class}
        )
        public ContentNegotiatingViewResolver viewResolver(BeanFactory beanFactory) {
            ContentNegotiatingViewResolver resolver = new ContentNegotiatingViewResolver();

resolver.setContentNegotiationManager((ContentNegotiationManager)beanFactory.getBean(ContentNegotiationManager.class));
            resolver.setOrder(-2147483648);
            return resolver;
        }
```

重点

@EnableConfigurationProperties({WebMvcProperties.**class**, ResourceProperties.**class**, WebProperties.**class**})

这个注解导入的配置类都可在yml中做好配置

- 静态资源位置

Java | 复制代码

```java
        private static final String[] CLASSPATH_RESOURCE_LOCATIONS = new String[]                          {"classpath:/META-INF/resources/",
                                                "classpath:/resources/", "classpath:/static/",
                                        "classpath:/public/"};
        private String[] staticLocations;
//静态资源的位置
        public void addResourceHandlers(ResourceHandlerRegistry registry) {
            if (!this.resourceProperties.isAddMappings()) {
                logger.debug("Default resource handling disabled");
            } else {
                this.addResourceHandler(registry, "/webjars/**", "classpath:/META-INF/resources/webjars/");
                this.addResourceHandler(registry, this.mvcProperties.getStaticPathPattern(), (registration) -> {
 registration.addResourceLocations(this.resourceProperties.getStaticLocations());
                    if (this.servletContext != null) {
                        ServletContextResource resource = new ServletContextResource(this.servletContext, "/");
                        registration.addResourceLocations(new Resource[] {resource});
                    }

                });
            }
        }
```

- 欢迎页：搜索getwelcomepage方法
- fomartting日期转换个是默认为yyyy-mm-dd springmvc中为yyyy/mm/dd 搜索fomartt
- httpMessageConverters 负责请求和响应报文处理
- hiddenHttpmethodFitter 表单过滤器
- 拦截器

1. 定义拦截器 implements HandlerInterceptor
2. 配置拦截器

```java
webMvcConfigurer
default void addInterceptors(InterceptorRegistry registry) {
}
```

- 跨域请求

```java
    default void addCorsMappings(CorsRegistry registry) {
    }
```

- springmvc-json
- httpMessageConverters ---->import(gson,jackson,json-8)---按照条件装配
  JsonIgnore,JsonFormat jsonInclude,jsonproperty

总结 mvcAutoConfiguration引入了 webMvcConfigurer　既保持了springboot的自动配置，又保证了springmvc的扩展性