# Fluorite原理概述

## 一、Fluorite启动与初始化

1).由标注了@RunnerAs注解的根启动类调用静态的run方法而启动。在@RunnerAs中可指定”debug”和”debugFormAop”属性控制Fluorite运行期间是否显示IOC和AOP相关日志信息。

2).Environment，运行当中的实现为StandardServletEnvironment，由ConfigurableEnvironment接口的customizePropertySources()方法读取配置文件和属性填充。

3).Banner，FluoriteBootBanner为此接口的唯一实现，再来打印横幅信息。横幅信息可像SpringBoot那样自定义更改，同时也可在配置文件中指定路径位置。配置文件中指定的Banner文件优先于项目根目录下的Banner.txt文件，若此两处均为读取到指定的Banner文件中默认使用fluorite-boot包下的META-INF文件夹下的banner-stant.txt文件。

4).ApplicationContext,运行当中的实现为AnnotationConfigServletWebServerApplicationContext。

5).对于要使用的ApplicationContextInitializer与ApplicationListener实现，则通过MetaFileLoader类来读取META-INF文件夹下的fluorite.factories文件获得。以下为文件key定义对象。

|  |  |
| --- | --- |
| **public** **final** **class** FactoriesProperty { // fluorite.factoroes文件的key定义  **public** **static** **final** String ***APPLICATION\_CNTEXT\_INITIALIZER*** = "org.zy.fluorite.context.interfaces.ApplicationContextInitializer";  **public** **static** **final** String ***APPLICATION\_LISTENER*** = "org.zy.fluorite.context.event.interfaces.ApplicationListener";  **public** **static** **final** String ***ENABLE\_AUTOCONFIGURATION*** = "org.zy.fluorite.boot.annotation.EnableAutoConfiguration";  } | |
| -fluorite-boot包下的fluorite.factories文件内容  # ApplicationContext 初始化器  org.zy.fluorite.context.interfaces.ApplicationContextInitializer=\  org.zy.fluorite.context.support.initializer.ContextIdApplicationContextInitializer,\  org.zy.fluorite.context.support.initializer.DelegatingApplicationContextInitializer,\  org.zy.fluorite.context.support.initializer.ConfigurationWarningsApplicationContextInitializer | # Application 监听器  org.zy.fluorite.context.event.interfaces.ApplicationListener=\  org.zy.fluorite.boot.context.event.listener.DelegatingApplicationListener,\  org.zy.fluorite.boot.context.event.listener.ClasspathLoggingApplicationListener,\  org.zy.fluorite.boot.context.event.listener.LoggingApplicationListener |
| **public** **final** **class** Property { // 配置文件定义属性Key  /\*\* 应用程序名称-fluorite.application.name \*/  **public** **static** **final** String ***APPLICATION\_NAME*** = "fluorite.application.name";  /\*\* 内镶容器端口号-server.port \*/  **public** **static** **final** String ***SERVER\_PORT*** = "server.port";    /\*\* 激活的配置文件后缀-fluorite.profiles.active \*/  **public** **static** **final** String ***PROFILES\_ACTIVVE*** = "fluorite.profiles.active";    /\*\* 默认的配置文件后缀-fluorite.profiles.default \*/  **public** **static** **final** String ***PROFILES\_DEFAULT*** = "fluorite.profiles.default";    /\*\* 配置不建议扫描的包路径-fluorite.scanner.problematic.packages \*/  **public** **static** **final** String ***PROBLEM\_PACKAGES*** = "fluorite.scanner.problematic.packages"; | /\*\* 配置组件索引文件的存储位置（相对与项目根目录的路径），默认存储于项目根目录下。此路径需以“/”结尾 \*/  **public** **static** **final** String ***COMPONENT\_INDEX*** = "fluorite.scanner.component.index";    /\*\* 配置组件索引文件的名称，默认为 'component.index' \*/  **public** **static** **final** String ***COMPONENT\_INDEX\_NAME*** = "fluorite.scanner.component.indexName";    /\*\* 配置要使用的ApplicationContextInitializer接口实现-context.initializer.classes \*/  **public** **static** **final** String ***INITIALIZER\_CLASSES*** = "context.initializer.classes";    /\*\* 配置要使用的ApplicationListener接口实现-context.listener.classes \*/  **public** **static** **final** String ***LISTENER\_CLASSES*** = "context.listener.classes";    /\*\* 设置非存储于在classpath路径下的'banner.txt'文件的路径. 路径格式为java的File对象支持的格式 \*/  **public** **static** **final** String ***BANNER\_LOCATION*** = "fluorite.banner.location";  } |

### 1.包扫描

1).包扫描路径不建议以[org、org.zy、org.zy.fluorite]开头。在根启动类上若指定了包扫描注解则根据获得的包路径求交集。即根据文件路径的包含与被包含关系获得最靠近项目根目录的路径集合。比如说指定了三个包扫描路径[“com.zy.controller”,”com.zy.service”,”com.zy”]，那么最终的候选路径集为[“com.zy”]。

2).组件扫描结果的过滤，根据@ComponentScan注解的Filter属性来配置同一扫描周期所使用的TypeFilter实现。同一扫描周期是指：从解析一个类获得@ComponentScan集合开始，并通过注解属性进行的包扫描，直到本次包扫描结束为止，此为一个扫描周期。比如说现有两个类A和B，其中此两个类都标注了@ComponentScan注解，且指定”com.zy.ccontroller”和”com.zy.service”分别作为其扫描路径，那么此时解析这两个类就有两个扫描周期。

3).本实现与SpringBoot相比，同样具有@indexed注解。此注解仅标注于@Compont注解上。在包扫描结果通过TypeFilter实现筛选之后，判断其是否标注了@indexed注解，标注了则为其保存组件索引。但此处与Spring不同的是保存组件索引这一动作只有在项目开始运行解析组件类和没有组件索引文件时才会保保存。即开始进行包扫描时首先判断是否有组件索引文件存在于项目中，没有则创建然后记录标注@indexed注解的组件。若存在则直接读取组件索引文件内容，根据其内容创建BeanDefinition实例。

### 2.自动配置

1).在@RunnerAs注解上标注了@EnableAutoConfiguration注解，其中使用@Import注解导入了AutoConfigurationImportSelector类。由此而实现动态的导入fluorite-autoconfigure包下的fluorite.factories中预制的配置类。

2).自动配置注解@Conditional及其扩展注解。通过此类注解来控制配置的生效与否。因为本人对SpringBoot的条件注解的ConfigurationPhase枚举作用还为完全理解，所以Fluorite中的条件注解解析逻辑和使用限制与SpringBoot不同：

对于Condition实现类来说，一般是由诸如 ConditionalOnBean 或 ConditionalOnClass 这样的条件注解所引入。在ConditionEvaluator中，同一个处理周期就意味着可能要处理多个 Conditional 注解。

而由其引入的 Condition 实现类又不可控的可能出现重复。所以特规定：同一个处理周期中每个Condition 实现只会被调用一次matcher(ConditionContext, AnnotationAttributes)方法。

所谓一个处理周期是对于一个配置类或者是一个@Bean方法来说的。分为配置类解析阶段(对应着ConfigurationPhase的PARSE\_CONFIGURATION属性)和Bean注册阶段（对应着ConfigurationPhase的REGISTER\_BEAN属性）。而划分这两阶段的标志就是程序是否进入到ConfigurationClassParser类和ConfigurationClassBeanDefinitionReader。前者代表配置类解析阶段，只会执行这一阶段的Condition实现类，如：OnClassCondition、OnPropertyCondition、OnResourceCondition。而后者则代表Bean注册阶段，执行的是此一阶段的Condition实现类，如OnBeanCondition。

但Condition实现也不强制要求其站队，也可以直接实现Condition接口成为一个”骑墙派”，这样它就拥有了两次执行matcher(ConditionContext, AnnotationAttributes)方法的机会，且在整个程序运行期间所有的Condition实例都将是单例的存在于ReflectionUtils类的缓存中。

所以”骑墙派”的Condition可以进行前置检查和后置检查这样的逻辑任务。但Fluorite中并未提供此样式的实现，使用者可以自行实现和创建对应的注解应用到配置类中。

## 二、IOC与AOP

1).Bean实例化和初始化与Spring相同，且实现生命周期Bean的处理逻辑。

2).提供AutowiredAnnotationBeanPostProcessor和CommonAnnotationBeanPostProcessor进行属性注入

## 三、AOP

1).对于AOP来说，实现了Cglib动态代理和JDK动态代理，但Spring的JDK动态代理本人没有任何使用经验，所以当前只是照着Spring的逻辑提供了JdkDynamicAopProxy类，能否正常使用暂时无法确定，而CglibAopProxy当前可正常使用。

2).不依赖于AspectJ来做切点匹配，而由PointcutExpression和ClassFilter来代替。其中切面Advice使用PointcutExpression，DeclareParentsAdvisor使用ClassFilter进行切点匹配。在PointcutExpression中，持有一个PointcutExpressionParse接口的实现DefaultPointcutExpressionParse类。其为切点表达式解析器，负 责具体的切点表达式预解析匹配事务，其parse(…)方法返回一个永不为null的PointcutMatcher实例来描述匹配结果。使用者可在项目根目录下的fluorite.factories文件中指定” org.zy.fluorite.aop.aspectj.interfaces.PointcutExpressionParse”的属性值来设置自定义的PointcutExpressionParse实现。

3).切点表达式定义与AspectJ相似，但为了避免进行大量的字符串分割所以此切点表达式并不完全容纳于注解的value属性中，而是将AspectJ式的切点表达式进行拆分而分布于注解的各项属性中，以下为@Pointcut注解的定义内容：

|  |  |  |
| --- | --- | --- |
| @Retention(RetentionPolicy.***RUNTIME***)  @Target(ElementType.***METHOD***)  @Documented  **public** **@interface** Pointcut {  /\*\*  \* 切点的全限定名称，但方法参数无论切点方法是否有都不需在此指定。  \* 而在args()方法中指定。  \* ps：  \* <ol>语法示例：  \* <li>org.zy.fluorite.aop.aspectj.annotation.Pointcut.\*：将Pointcut类中定义的方法全部作为切点匹配。</li>  \* <li>org.zy.fluorite.aop.aspectj.annotation.Pointcut.value()：将Pointcut类中定义的value方法作为切点匹配。</li>  \* <li>org.zy.fluorite.aop.aspectj.annotation..：将此包名下的所有类中的所有方法作为切点匹配</li>  \* <li>org.zy.fluorite.aop.aspectj.annotation..value()：将此包名下的所有类中的value方法作为切点匹配</li>  \* </ol>  \*/  String value() **default** "";  /\*\*  \* 为了应对在编译之后方法参数名丢失但在程序运行时需要此信息的情况， 可在此指定参数名列表，并使用逗号分隔。  \* 但此方法返回值的使用与否以切点表达式而定，若切点表达式中使用到了参数名但此处却未指定对应的参数名则会抛出异常  \*/  @Deprecated  String ~~argNames~~() **default** "";    /\*\* | \* 切点方法所拥有的全部参数类型，若未指定则默认匹配无参的切点方法，未找到则抛出异常</br>  \* ps：此数组内容会作为方法参数匹配的依据，所以若切点方法有参数则必须指定  \*/  Class<?>[] args() **default** {};    /\*\*  \* 切点表达式的语义补正（此功能暂未实现）  \* <ol>支持的语法前缀：  \* <li>execution</li>  \* <li>within</li>  \* <li>args</li>  \* <li>this</li>  \* <li>target</li>  \* </ol>  \*/  String prefix() **default** "execution";  /\*\*  \* 访问修饰符约束  \* <ol>支持的修饰符约束：  \* <li>public</li>  \* <li>protected</li>  \* <li>default</li> | \* <li>private</li>  \* <li>abstract</li>  \* </ol>  \*/  **int** accessModifier() **default** Modifier.***PUBLIC*** ;    /\*\*  \* 方法的返回值，若返回值有泛型则可连带着泛型信息。</br>  \* <p>  \* 如方法返回一个Object对象，那么此注解属性值就应该被定义为 {@code "Object" }</br>  \* 如方法返回一个List对象，且其内元素类型为String类型，那么此注解属性值就应该被定义为 {@code "LIst<String>"}</br>  \* 如方法没有返回值，那么此注解属性值就应该被定义为 {@code "void"}</br>  \* </p>  \* <ul>补充：  \* <li>此返回值信息会作为方法返回值匹配的依据，所以若切点方法有返回值则必须指定，未指定则匹配失败。</li>  \* <li>使用“\*”可匹配所有的返回值包括void。</li>  \* <li>此属性大小写不敏感。</li>  \* <li>当前不支持泛型嵌套，只支持一对“<>”之中的泛型比对</li>  \* <li>倘若方法返回值有泛型信息但此属性未携带泛型信息则视为泛型擦除。<br/>  \* 此时只比对方法的返回值，返回值相同或满足继承实现关系则视为返回值匹配。</li>  \* </ui>  \*/  String returnType() **default** "\*"; |

4).切点表达式的定义有两种方式：自身使用的切点表达式和引用连接点方法式的切点表达式。以下为选择说明：

(1). 当一个切面通知注解引用了连接点方法，那么此注解除了value属性之外的其他属性都将被忽略，而以引用的连接点方法上标注的@Pointcut注解值描述其目标方法。

(2). 若一个切面类中若存多个通知方法，且多个通知方法都指向了一个切点，那么就宜采用引用连接点方法的方式来组织切面通知，而不是为它们单独的设置切面表达式。

(3). 本实现鼓励使用统一的连接点来对切面中定义的通知方法进行分组，通过连接点的第一次匹配结果来描述这组通知方法是否适配给定Bean。

(4). 若一个通知方法没有引用连接点方法，那么就视之为切面类中唯一的通知方法，在PointcutExpression也就不会缓存其切点相关信息。

## 四、提交记录

### AOP切面匹配逻辑新增（7907f84）

1).新增AspectJ中的三中切面匹配逻辑支持。包括：@within、@annotation、@args。此三者不支持使用通配符，即：在@Pointcut的value属性中不能使用“\*”和“..”

2).关于AOP功能使用的限制：

(1). 在匹配多个方法是不能在切面方法中使用额外参数，即除了JoinPoint及其子类之外的其他类型参数，否则在进行切面方法参数绑定时会报参数异常。而匹配单个或重载方法时可以使用额外的参数类型以获取目标切点的参数值。

(2). 在匹配单个方法时可以在任意切面方法中获得参数。

(3). 在适配多个方法时使用@DeclareParents注解引入的接口方法也会被设置aop环绕。

(4). 此次更新后使用除 “execution”之外的其他切面匹配逻辑需指定与之对应的prefix值。

## 附录

联系方式：15969413461@163.com