https://www.jianshu.com/p/d5ffdccc4f5d

# Spring 的类扫描器分析 - ClassPathBeanDefinitionScanner

[](https://www.jianshu.com/u/b79a3348a7ad)

[白袜子先生](https://www.jianshu.com/u/b79a3348a7ad) 关注

2018.08.03 10:37\* 字数 776 阅读 717评论 0喜欢 2

#### 1. 简介

SpringBoot项目中或者 Spring项目中配置<context:component-scan base-package="com.example.demo" />  
，那么在IOC 容器初始化阶段（调用beanFactoryPostProcessor阶段） 就会采用ClassPathBeanDefinitionScanner进行扫描包下 所有类，并将符合过滤条件的类注册到IOC 容器内。Mybatis 的Mapper注册器(ClassPathMapperScanner) 是同过继承ClassPathBeanDefinitionScanner,并且自定义了过滤器规则来实现的。具体的 调用过程并不会在这里说明，只是想在这里描述ClassPathBeanDefinitionScanner是如何 扫描 和 注册BeanDefinition的。

#### 2. 作用

ClassPathBeanDefinitionScanner作用就是将指定包下的类通过一定规则过滤后 将Class 信息包装成 BeanDefinition 的形式注册到IOC容器中。

1. 根据指定扫描报名 生成匹配规则。

例如：classpath\*:com.example.demo/\*\*/\*.class

1. resourcePatternResolver（资源加载器）根据匹配规则 获取 Resource[] 。
   * Resource数组中每一个 对象 都是对应一个 Class 文件，Spring 用Resource定位资源， 封装了资源的IO操作。
   * 这里的 Resource 实际类型是 FileSystemResource.
   * 资源加载器 其实就是 容器 本身。
2. meteDataFactory根据 Resouce 获取到 MetadataReader 对象
   * MetadataReader 提供了 获取 一个Class 文件的 ClassMetadata 和 AnnotationMetadata 的 操作。
3. 根据过滤器规则 匹配 MetadataReader中的类 进行过滤，比如 是否是Componet 注解标注的类。
4. 转换 MetadataReader 为 BeanDefinition.
5. 将BeanDefinition 注册到 BeanFactory.

#### 3. 默认的过滤器注册

过滤器用来过滤 从指定包下面查找到的 Class ,如果能通过过滤器，那么这个class 就会被转换成BeanDefinition 注册到容器。

如果在实例化ClassPathBeanDefinitionScanner时，没有说明要使用用户自定义的过滤器的话，那么就会采用下面的默认的过滤器规则。

注册了@Component 过滤器到 includeFiters ,相当于 同时注册了所有被@Component注释的注解，包括@Service ，@Repository,@Controller，同时也支持java EE6 的javax.annotation.ManagedBean 和 JSR-330 的 @Named 注解。

protected void registerDefaultFilters() {

// 添加Component 注解过滤器

//这就是为什么 @Service @Controller @Repostory @Component 能够起作用的原因。

this.includeFilters.add(new AnnotationTypeFilter(Component.class));

ClassLoader cl = ClassPathScanningCandidateComponentProvider.class.getClassLoader();

try {

// 添加ManagedBean 注解过滤器

this.includeFilters.add(new AnnotationTypeFilter(

((Class<? extends Annotation>) ClassUtils.forName("javax.annotation.ManagedBean", cl)), false));

logger.debug("JSR-250 'javax.annotation.ManagedBean' found and supported for component scanning");

}

catch (ClassNotFoundException ex) {

// JSR-250 1.1 API (as included in Java EE 6) not available - simply skip.

}

try {

// 添加Named 注解过滤器

this.includeFilters.add(new AnnotationTypeFilter(

((Class<? extends Annotation>) ClassUtils.forName("javax.inject.Named", cl)), false));

logger.debug("JSR-330 'javax.inject.Named' annotation found and supported for component scanning");

}

catch (ClassNotFoundException ex) {

// JSR-330 API not available - simply skip.

}

}

#### 4. 执行扫描（doScan）

实际执行包扫描,进行封装的函数是findCandidateComponents,findCandidateComponents定义在父类中。ClassPathBeanDefinitionScanner的主要功能实现都在这个函数中。

doScan流程.png

public Set<BeanDefinition> findCandidateComponents(String basePackage) {

Set<BeanDefinition> candidates = new LinkedHashSet<BeanDefinition>();

try {

// 1.根据指定包名 生成包搜索路径

String packageSearchPath = ResourcePatternResolver.CLASSPATH\_ALL\_URL\_PREFIX +

resolveBasePackage(basePackage) + '/' + this.resourcePattern;

//2. 资源加载器 加载搜索路径下的 所有class 转换为 Resource[]

Resource[] resources = this.resourcePatternResolver.getResources(packageSearchPath);

// 3. 循环 处理每一个 resource

for (Resource resource : resources) {

if (resource.isReadable()) {

try {

// 读取类的 注解信息 和 类信息 ，信息储存到 MetadataReader

//

MetadataReader metadataReader = this.metadataReaderFactory.getMetadataReader(resource);

// 执行判断是否符合 过滤器规则，函数内部用过滤器 对metadataReader 过滤

if (isCandidateComponent(metadataReader)) {

//把符合条件的 类转换成 BeanDefinition

ScannedGenericBeanDefinition sbd = new ScannedGenericBeanDefinition(metadataReader);

sbd.setResource(resource);

sbd.setSource(resource);

// 再次判断 如果是实体类 返回true,如果是抽象类，但是抽象方法 被 @Lookup 注解注释返回true

if (isCandidateComponent(sbd)) {

if (debugEnabled) {

logger.debug("Identified candidate component class: " + resource);

}

candidates.add(sbd);

}

//省略了 部分代码

}

catch (IOException ex) {

throw new BeanDefinitionStoreException("I/O failure during classpath scanning", ex);

}

return candidates;

}

#### 5. 自定义扫描器

通过自定义的扫描器,扫描指定包下所有被@MyBean 注释的类。

##### 5.1 定义一个注解，并注释一个类

@Target({ElementType.TYPE})

@Retention(RetentionPolicy.RUNTIME)

@Documented

public @interface MyBean {

}

@MyBean

public class TestScannerBean {

}

##### 5.2 编写扫描器

class MyClassPathDefinitonScanner extends ClassPathBeanDefinitionScanner{

private Class type;

public MyClassPathDefinitonScanner(BeanDefinitionRegistry registry,Class<? extends Annotation> type){

super(registry,false);

this.type = type;

}

/\*\*

\* 注册 过滤器

\*/

public void registerTypeFilter(){

addIncludeFilter(new AnnotationTypeFilter(type));

}

}

##### 5.3 测试自定义扫描器

* 测试代码

@Test

public void testSimpleScan() {

String BASE\_PACKAGE = "com.example.demo";

GenericApplicationContext context = new GenericApplicationContext();

MyClassPathDefinitonScanner myClassPathDefinitonScanner = new MyClassPathDefinitonScanner(context, MyBean.class);

// 注册过滤器

myClassPathDefinitonScanner.registerTypeFilter();

int beanCount = myClassPathDefinitonScanner.scan(BASE\_PACKAGE);

context.refresh();

String[] beanDefinitionNames = context.getBeanDefinitionNames();

System.out.println(beanCount);

for (String beanDefinitionName : beanDefinitionNames) {

System.out.println(beanDefinitionName);

}

}

* 测试结果

7

//这个就是我们扫描到的bean

testScannerBean

//下面这些 是 父类扫描器 注册的 beanFactory后置处理器

org.springframework.context.annotation.internalConfigurationAnnotationProcessor

org.springframework.context.annotation.internalAutowiredAnnotationProcessor

org.springframework.context.annotation.internalRequiredAnnotationProcessor

org.springframework.context.annotation.internalCommonAnnotationProcessor

org.springframework.context.event.internalEventListenerProcessor

org.springframework.context.event.internalEventListenerFactory

#### 6. 总结

通过对ClassPathBeanDefinitionScanner的分析,终于揭开了Spring 的类扫描的神秘面纱，其实，就是对指定路径下的 所有class 文件进行逐一排查，对符合条件的 class ,封装成 BeanDefinition注册到IOC 容器。

理解ClassPathBeanDefinitionScanner的工作原理，可以帮助理解Spring　IOC 容器的初始化过程。

同时对理解MyBatis 的 Mapper 扫描 也是有很大的帮助。  
因为 MyBatis 的MapperScannerConfigurer的底层实现也是一个ClassPathBeanDefinitionScanner的子类。就像我们自定义扫描器那样，自定定义了 过滤器的过滤规则。