

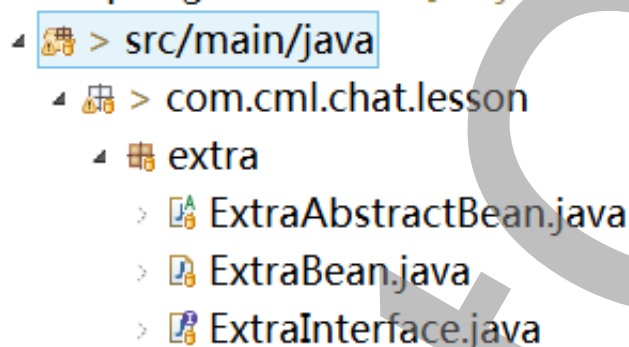
## 第06课：Spring 中 Bean 扫描实战

当需要扫描 Bean 可以使用 `@ComponentScan(basePackages="")` 对指定包下添加的 Spring 支持的注解的类。Spring Boot 是默认会扫描 `@SpringBootApplication` 注解所在包和所有子包的类。这样使用的话对于单纯的业务逻辑实现是没有问题的，但是如果想要把共通实现抽取出来作为公共项目，或者自定义拓展自己的注解，这时该如何扫描 Bean 呢？

假设你需要使用自定义注解的方式实现某一个功能，或者想用接口的方式实现某一个功能，那么这时候 `@ComponentScan` 就起不了作用了，默认情况包的扫描是不会扫描接口类的，而且自定义的注解也不会被扫描进去。这时候就需要自己对包进行扫描了，Spring 提供了几个非常好的类扫描功能，通过这几个类可以非常简单高效的完成我们需要的功能。

### PathMatchingResourcePatternResolver

在 Spring 中，类文件也是一种资源文件，那么就可以通过 `PathMatchingResourcePatternResolver` 进行扫描，当前工程的目录结构如下（根据名字可以分辨出类的类型）：



```
src/main/java
├── com.cml.chat.lesson
│   └── extra
│       ├── ExtraAbstractBean.java
│       ├── ExtraBean.java
│       └── ExtraInterface.java
```

入口类为 `Application`，在 `Application` 的 `main` 方法中扫描 `extra` 这个包下的文件，代码如下：

```
public static void main(String[] args) throws Exception {
    SpringApplication springApplication = new SpringApplication(Application.class)
    ;
    ConfigurableApplicationContext application = springApplication.run(args);

    PathMatchingResourcePatternResolver resolver = new PathMatchingResourcePattern
    Resolver();
    Resource[] resources = resolver.getResources("classpath:/com/cml/chat/lesson/e
    xtra/**/*.class");

    log.info("findResourceSize==>" + Arrays.asList(resources).toString());
}
```

使用 `PathMatchingResourcePatternResolver` 对 `extra` 包下所有的资源文件进行扫描，筛选出 `class` 文件。

这样包下所有的 class 都被扫描进来了，但是我们这里只是获取到了 class 文件而已，还得要根据扫描出的结果，对扫描进来的 class 文件进行校验，判断对应的 class 是否是我们需要的对象，这时候还得 class 加载进来，这样的繁琐操作显然不适合实际开发需求。正好 Spring 提供了更好的操作方式 `ClassPathScanningCandidateComponentProvider`，通过这个类可以扫描出需要的类文件，并且可以对扫描出的对象通过字节码的方式获取到类的类型。

## ClassPathScanningCandidateComponentProvider

`ClassPathScanningCandidateComponentProvider` 也是通过 `PathMatchingResourcePatternResolver` 进行文件扫描，但是对其封装了一层，将扫描到的类通过读取字节码的方式获取到类信息。其核心实现在方法 `findCandidateComponents` 中，通过此方法可以扫描出所有需要的对象，并且可以通过 `isCandidateComponent` 方法对扫描到的对象进行筛选。核心实现如下：

```
public Set<BeanDefinition> findCandidateComponents(String basePackage) {
    Set<BeanDefinition> candidates = new LinkedHashSet<BeanDefinition>();
    try {
        //配置扫描规则，最后生成的是：classpath*:basePackage/**/*.class
        String packageSearchPath = ResourcePatternResolver.CLASSPATH_ALL_URL_PREFIX +
            resolveBasePackage(basePackage) + '/' + this.resourcePattern;
        //resourcePatternResolver为PathMatchingResourcePatternResolver
        Resource[] resources = this.resourcePatternResolver.getResources(packageSearchPath);
        boolean traceEnabled = logger.isTraceEnabled();
        boolean debugEnabled = logger.isDebugEnabled();
        for (Resource resource : resources) {
            //打印log
            if (resource.isReadable()) {
                try {
                    MetadataReader metadataReader = this.metadataReaderFactory.getMetadataReader(resource);
                    if (isCandidateComponent(metadataReader)) {
                        ScannedGenericBeanDefinition sbd = new ScannedGenericBeanDefinition(metadataReader);
                        sbd.setResource(resource);
                        sbd.setSource(resource);
                        if (isCandidateComponent(sbd)) {
                            //打印log
                            candidates.add(sbd);
                        }
                    } else {
                        //打印log
                    }
                }
            } else {
                //打印log
            }
        }
    }
}
```

```

        }
    }
    catch (Throwable ex) {
        //抛出异常
    }
}
else {
    //打印log
}
}
}
}
catch (IOException ex) {
    //抛出异常...
}
return candidates;
}
}

```

主要流程可以归结如下：

- 通过 `resourcePatternResolver` 将需要的 class 扫描出来

`resourcePatternResolver` 为 `PathMatchingResourcePatternResolver` 实例。

- 将扫描出的类信息封装为 `MetadataReader` 对象

使用 ASM 框架读取字节码获取 class 对象信息，将 class 信息封装为 `AnnotationMetadata` 对象。[ASM 信息可以参考这里](#)。

- `isCandidateComponent` 方法中，对扫描结果进行过滤

回调通过 `addExcludeFilter`，`addIncludeFilter` 方法添加的对象过滤器，进行规则匹配，只有满足过滤条件的数据才会进入候选类，进入下一轮筛选。

- 过滤成功后使用 `isCandidateComponent` 方法校验对象是否是我们需要的类

默认是只扫描普通类和加了 `@Lookup` 注解的抽象类的，如果需要扫描接口和抽象类，就需要重写这个方法，将接口和抽象类添加到候选列表中。代码如下：

```

@Override
protected boolean isCandidateComponent(AnnotatedBeanDefinition beanDefinition) {
    AnnotationMetadata metadata = beanDefinition.getMetadata();
    return (metadata.isIndependent() && (metadata.isAbstract() || metadata.isInterface()));
}

```

通过以上的步骤最终筛选出的类就是我们需要的对象了。那么如何使用 `ClassPathScanningCandidateComponentProvider` 呢？这里举个例子：`SimpleBeanScanner` 在 `Bean` 初始

化完成之后就使用 `MyClassScanner` 对资源文件进行扫描，将扫描出的类打印出来。

```
@Component
public class SimpleBeanScanner implements EnvironmentAware, ResourceLoaderAware {
    private static Logger log = LoggerFactory.getLogger(SimpleBeanScanner.class);
    private ResourceLoader resourceLoader;
    private Environment environment;

    @PostConstruct
    public void scan() {
        log.info("=====start scan=====");
        ;
        MyClassScanner scanner = new MyClassScanner();
        scanner.setEnvironment(environment);
        scanner.setResourceLoader(resourceLoader);
        scanner.addIncludeFilter(new TypeFilter() {

            @Override
            public boolean match(MetadataReader metadataReader, MetadataReaderFactory
metadataReaderFactory) throws IOException {
                return true;
            }
        });
        log.info(scanner.findCandidateComponents("com.cml.chat.lesson.lesson6").toStri
ng());
        log.info("=====end scan=====");
    }

    @Override
    public void setResourceLoader(ResourceLoader resourceLoader) {
        this.resourceLoader = resourceLoader;
    }

    @Override
    public void setEnvironment(Environment environment) {
        this.environment = environment;
    }
}

public class MyClassScanner extends ClassPathScanningCandidateComponentProvider {

    public MyClassScanner() {
    }

    @Override
    protected boolean isCandidateComponent(AnnotatedBeanDefinition beanDefinition) {
        AnnotationMetadata metadata = beanDefinition.getMetadata();
        return (metadata.isIndependent() && (metadata.isAbstract() || metadata.isInter
```

```
face())));  
}  
}
```

通过调用 `ClassPathScanningCandidateComponentProvider.findCandidateComponents` 就可以对指定包下的类进行筛选，并且将扫描到的类转换成 Spring 中 `BeanDefinition` 集合返回。这样我们就可以非常方便的使用它进行 Bean 扫描，并将扫描到的类通过工厂 Bean 的方式添加到 Spring 上下文中了。但是添加到 Spring 上下文中还需要我们手动添加，有没有更简便的方式呢？

## ClassPathBeanDefinitionScanner

`ClassPathBeanDefinitionScanner` 继承自 `ClassPathScanningCandidateComponentProvider`，对 `ClassPathScanningCandidateComponentProvider` 提供了更高层的封装，对外开放 `scan` 方法，通过 `BeanDefinitionRegistry` 直接将扫描到的 Bean 对象添加到 Spring 上下文中。这样的实现方式对于普通类来说是非常实用的。

如果扫描到了接口对象，这时添加到 Spring 上下文中就会报错了。因为会自动将 Bean 注册到 Spring 上下文中，接口是无法实例化的，所以添加接口时需要使用工厂 Bean 的方式。

如果需要自定义扫描的话只需要继承 `ClassPathBeanDefinitionScanner` 就可以了，代码如下：

```
public class MyClassPathBeanDefinitionScanner extends ClassPathBeanDefinitionScanner {  
  
    public MyClassPathBeanDefinitionScanner(BeansDefinitionRegistry registry) {  
        super(registry);  
    }  
  
    @Override  
    protected boolean isCandidateComponent(AnnotatedBeanDefinition beanDefinition) {  
        AnnotationMetadata metadata = beanDefinition.getMetadata();  
        return (metadata.isIndependent() && (metadata.isConcrete() || metadata.isAbstract() || metadata.isInterface()));  
    }  
  
}
```

如果使用 `@Import` 导入的方式，只需要实现 `ImportBeanDefinitionRegistrar` 接口即可。代码如下：

```

public class MyClassPathBeanDefinitionScannerEntrance2 implements ImportBeanDefinitionRegistrar {

    @Override
    public void registerBeanDefinitions(AnnotationMetadata importingClassMetadata, BeanDefinitionRegistry registry) {

        MyClassPathBeanDefinitionScanner scanner = new MyClassPathBeanDefinitionScanner(registry);
        scanner.addIncludeFilter(new TypeFilter() {

            @Override
            public boolean match(MetadataReader metadataReader, MetadataReaderFactory metadataReaderFactory) throws IOException {
                return true;
            }
        });
        System.out.println("====MyClassPathBeanDefinitionScannerEntrance2=====>" + scanner.scan("com.cml.chat.lesson.extra"));
    }

}

```

SpringBoot 使用的 Bean 工厂为 DefaultListableBeanFactory，因为 DefaultListableBeanFactory 实现了 BeanDefinitionRegistry 接口，所以可以先获取到 Bean 工厂再进行扫描。当然也可以从 BeanDefinitionRegistry 注释中得知。

Spring's bean definition readers expect to work on an implementation of this interface. Known implementors within the Spring core are DefaultListableBeanFactory and GenericApplicationContext.

如果使用 @Component 注解的方式，则需要获取到 Bean 工厂，只需要实现 BeanFactoryAware 接口即可，至于 BeanFactoryAware 的原理前面的文章《Spring 各种 Aware 注入的原理与实战》已经详细说明了。

```

@Component
public class MyClassPathBeanDefinitionScannerEntrance implements BeanFactoryAware
{

    @Override
    public void setBeanFactory(BeanFactory beanFactory) throws BeansException {

        MyClassPathBeanDefinitionScanner scanner = new MyClassPathBeanDefinitionScanner(
            (BeanDefinitionRegistry) beanFactory);
        //所有的类筛选进来
        scanner.addIncludeFilter(new TypeFilter() {

            @Override
            public boolean match(MetadataReader metadataReader, MetadataReaderFactory
            metadataReaderFactory) throws IOException {
                return true;
            }
        });
        System.out.println("====MyClassPathBeanDefinitionScannerEntrance====");
        >" + scanner.scan("com.cml.chat.lesson.extra"));
    }

}

```

## 总结

这里对上文中提供扫描类功能的几个类进行总结。

- @ComponentScan

具有一定的局限性，只能识别 Spring 中内置的一些注解类，适合项目业务逻辑开发，不适合架构类的项目使用。

- PathMatchingResourcePatternResolver

基于 Spring 的强大的资源扫描器，可以对工程中的任意类型文件数据进行扫描，但是只是扫描到了对应的资源文件，并不能提供对资源文件的类型的校验。这个适合用户资源文件的扫描，比如 Properties 文件等

- ClassPathScanningCandidateComponentProvider

通过封装 PathMatchingResourcePatternResolver，筛选出 PathMatchingResourcePatternResolver 扫描出的类文件，并通过字节码的方式判断对应类的类型。将扫描到的类对象转换成 BeanDefinition，方便导入到上下文中。

- ClassPathBeanDefinitionScanner

通过封装 `ClassPathScanningCandidateComponentProvider`，将扫描出的 `BeanDefinition` 集合添加到 Spring 上下文中。对于框架类扫描功能，这个类还是非常实用的。

以上几个类都能实现资源文件的扫描功能，但是各有各的实用场景，通常来说对于类的扫描 `ClassPathBeanDefinitionScanner` 还是使用最多的，毕竟这个类封装了一层，提供更方便的方式。