SpringBoot自动配置原理

Java3y
原创技术公众号：Java3y

前言

> 只有光头才能变强。
> 文本已收录至我的GitHub仓库，欢迎Star：github.com/ZhongFuCheng...

回顾前面Spring的文章(以学习的顺序排好)：

- Spring入门这一篇就够了
- Spring【依赖注入】就是这么简单
- Spring【AOP模块】就这么简单
- Spring【DAO模块】知识要点
- SpringMVC入门就这么简单
- SpringMVC【开发Controller】详解
- SpringMVC【参数绑定、数据回显、文件上传】
- SpringMVC【校验器、统一处理异常、RESTful、拦截器】
- SpringBoot就是这么简单
- SpringData JPA就是这么简单
- Spring IOC知识点一网打尽!
- Spring AOP就是这么简单啦

- 外行人都能看懂的SpringCloud，错过了血亏！

作为一名Java程序员，就不可能不了解SpringBoot，如果不了解(赶紧学！)

一、SpringBoot的自动配置原理

不知道大家第一次搭SpringBoot环境的时候，有没有觉得非常简单。无须各种的配置文件，各种繁杂的pom坐标，一个main方法，就能run起来了。与其他框架整合也贼方便，使用 `EnableXXXXX` 注解就可以搞起来了！

所以今天来讲讲SpringBoot是如何实现自动配置的~

1.1三个重要的注解

我们可以发现，在使用 `main()` 启动SpringBoot的时候，只有一个注解 `@SpringBootAppli`

```
@SpringBootApplication
public class Java3yApplication {

    public static void main(String[] args) {
        SpringApplication.run(Java3yApplication.class, args
    }
}
```

我们可以点击进去 `@SpringBootApplication` 注解中看看，可以发现有**三个注解**是比较重要

```
@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.c
public @interface SpringBootApplication {
```

- `@SpringBootConfiguration` ： 我们点进去以后可以发现底层是**Configuration**注解，说是支持**JavaConfig**的方式来进行配置(**使用Configuration配置类等同于XML文件**)
- `@EnableAutoConfiguration` ： 开启**自动配置**功能(后文详解)

- `@ComponentScan` ： 这个注解，学过Spring的同学应该对它不会陌生，就是**扫描**注解，默认扫描**当前类下**的package。将 `@Controller/@Service/@Component/@Repository` 等注解加载到IoC容器中。

所以， `Java3yApplication` 类可以被我们当做是这样的：

```
@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan
public class Java3yApplication {

    public static void main(String[] args) {
        SpringApplication.run(Java3yApplication.class, args);
    }
}
```

1.2重点EnableAutoConfiguration

我们知道SpringBoot可以帮我们减少很多的配置，也肯定听过“约定大于配置”这么一句话，SpringBoot是怎么做的呢？其实靠的就是 `@EnableAutoConfiguration` 注解。

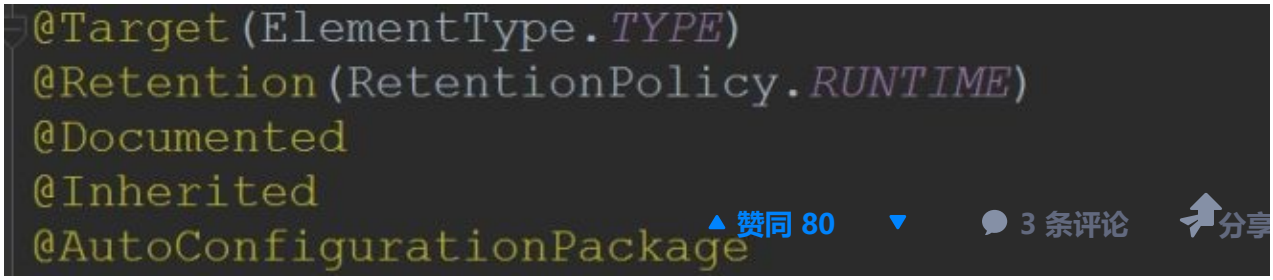
简单来说，这个注解可以帮助我们**自动载入**应用程序所需要的所有**默认配置**。

介绍有一句说：

> if you have tomcat-embedded.jar on your classpath you are likely to want a TomcatServletWebServerFactory

如果你的类路径下有 `tomcat-embedded.jar` 包，那么你很可能就需要TomcatServletWebServerFactory

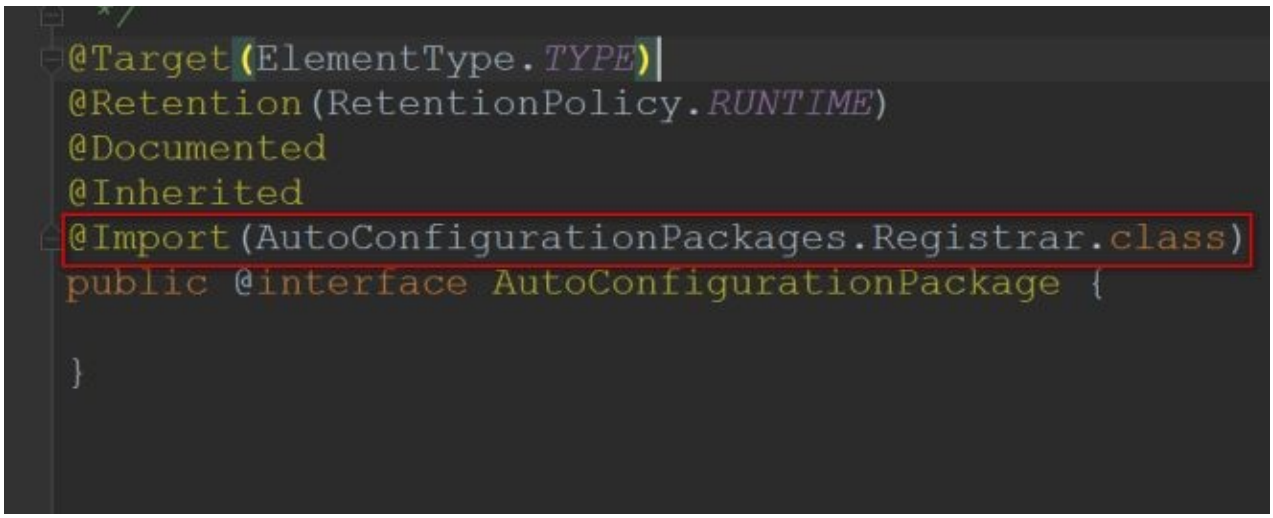
我们点进去看一下，发现有**两个**比较重要的注解：



- `@AutoConfigurationPackage` ： 自动配置包
- `@Import` ： 给IOC容器导入组件

1.2.1AutoConfigurationPackage

网上将这个 `@AutoConfigurationPackage` 注解解释成**自动配置包**，我们也看看 `@AutoConfigurationPackage` 里边有什么：



我们可以发现，依靠的还是 `@Import` 注解，再点进去查看，我们发现重要的就是以下的代码

```
@Override
public void registerBeanDefinitions(AnnotationMetadata metadata,
        BeanDefinitionRegistry registry) {
    register(registry, new PackageImport(metadata).getPackageName());
}
```

在**默认**的情况下就是将：主配置类(`@SpringBootApplication`)的所在包及其子包里边的组件到Spring容器中。

- 看完这句话，会不会觉得，这不就是ComponentScan的功能吗？这俩不就重复了吗？

我开始也有这个疑问，直到我看到文档的这句话：

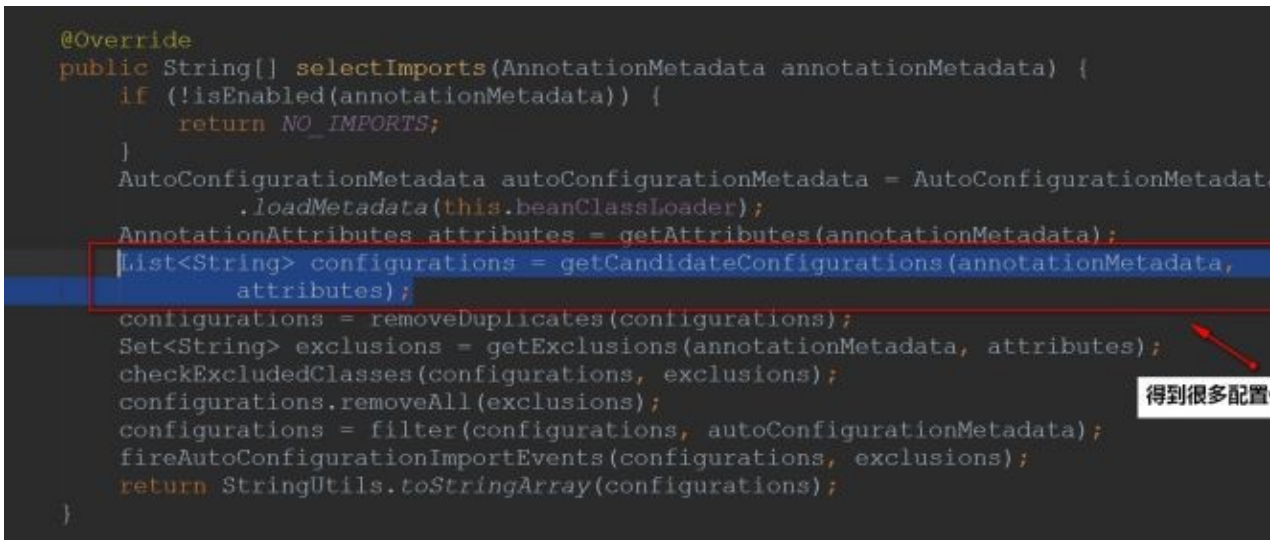
> it will be used when scanning for code @Entity classes. It is generally recommen
> that you place EnableAutoConfiguration (if you're not using @SpringBootApplic
> in a root package so that all sub-packages and classes can be searched.

比如说，你用了Spring Data JPA，可能会在实体类上写 `@Entity` 注解。这个 `@Entity` 注解 `@AutoConfigurationPackage` 扫描并加载，而我们平时开发用的 `@Controller/@Service/@Component/@Repository` 这些注解是由 `ComponentScan` 来扫描并加的。

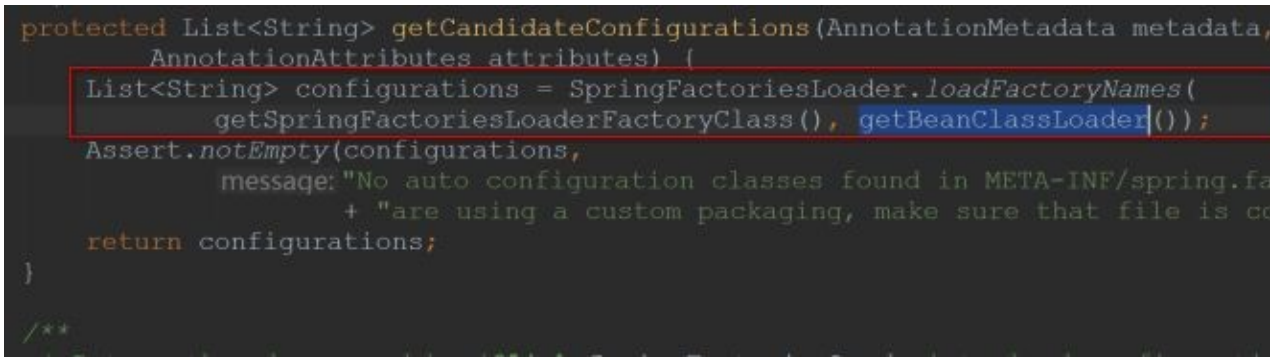
- 简单理解：这二者**扫描的对象是不一样**的。

1.2.2回到Import

我们回到 `@Import(AutoConfigurationImportSelector.class)` 这句代码上，再点进去 `AutoConfigurationImportSelector.class` 看看具体的实现是什么：



我们再进去看一下这些配置信息是从哪里来的(进去getCandidateConfigurations方法)：



这里包装了一层，我们看到的只是通过SpringFactoriesLoader来加载，还没看到关键信息进去：


```
public static List<String> loadFactoryNames(Class<?> factoryClass, @Nullable ClassLoader classLoader) {
    String factoryClassName = factoryClass.getName();
    return loadSpringFactories(classLoader).getOrDefault(factoryClassName, Collections.emptyList());
}

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

    try {
        Enumeration<URL> urls = (classLoader != null ?
                classLoader.getResources(FACTORIES_RESOURCE_LOCATION) :
                ClassLoader.getSystemResources(FACTORIES_RESOURCE_LOCATION));
        result = new LinkedMultiValueMap<>();
        while (urls.hasMoreElements()) {
            URL url = urls.nextElement();
            UrlResource resource = new UrlResource(url);
            Properties properties = PropertiesLoaderUtils.loadProperties(resource);
            for (Map.Entry<?, ?> entry : properties.entrySet()) {
                List<String> factoryClassNames = Arrays.asList(
                        StringUtils.commaDelimitedListToStringArray((String) entry.getValue()));
                result.addAll((String) entry.getKey(), factoryClassNames);
            }
        }
```

加载的是这里

简单梳理：

- FACTORIES_RESOURCE_LOCATION 的值是 META-INF/spring.factories
- Spring启动的时候会扫描所有jar路径下的 META-INF/spring.factories ，将其文件包装成Properties对象
- 从Properties对象获取到key值为 EnableAutoConfiguration 的数据，然后添加到容器里

```
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.fa
                    + "are using a custom packaging, make sure that file is co
    return configurations;
}

/**
 * Return the class used by {@link SpringFactoriesLoader} to load configuratio
 * candidates.
 * @return the factory class
 */
protected Class<?> getSpringFactoriesLoaderFactoryClass() {
    return EnableAutoConfiguration.class;
}
```

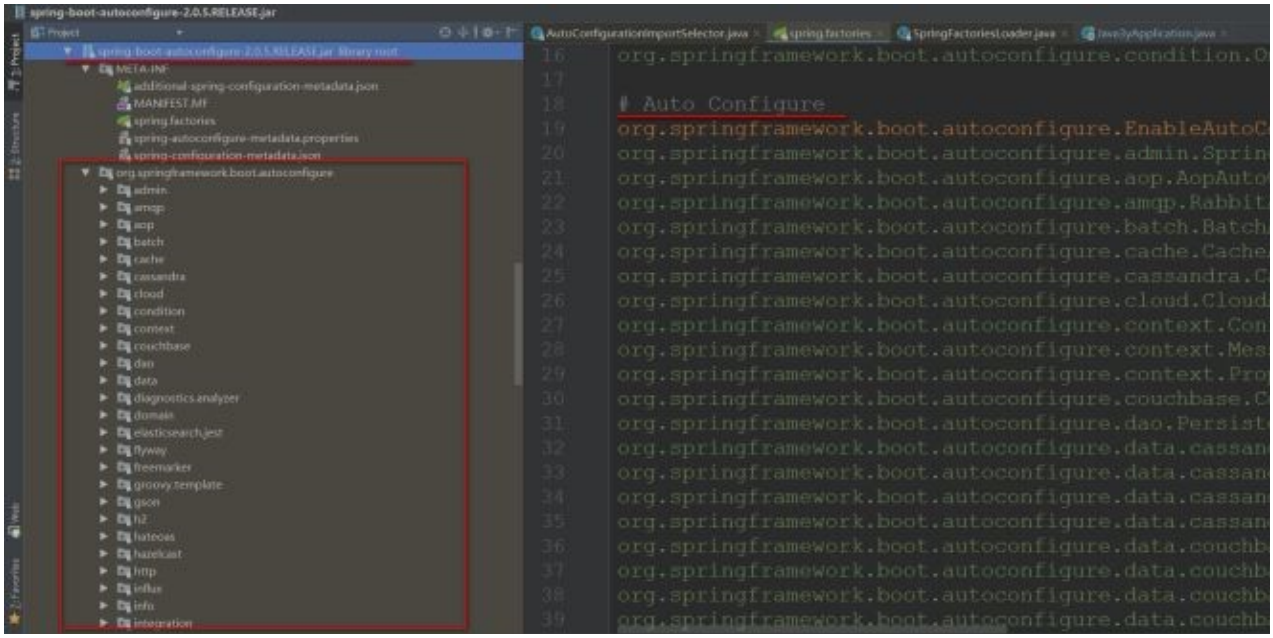
筛选出以EnableAutoConf为key的信息

最后我们会默认加载113个默认的配置类：

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

/**
 * Return the class used by {@link SpringFactoriesLoader} to load configuration
 * candidates.
 * @return the factory class
 */
protected Class<?> getSpringFactoriesLoaderFactoryClass() {
    return EnableAutoConfiguration.class;
}
```

有兴趣的同学可以去翻一下这些文件以及配置类哦：

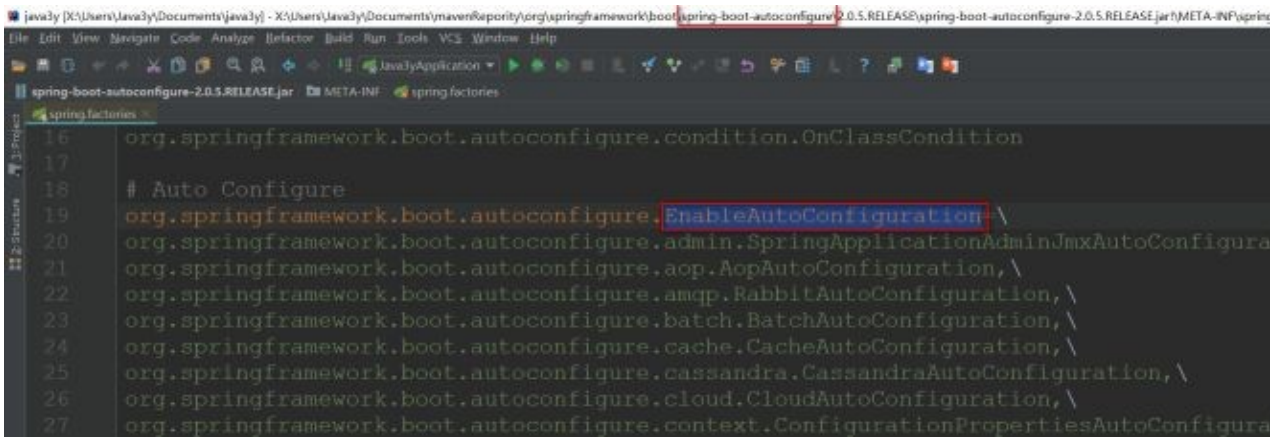


1.3总结

`@SpringBootApplication` 等同于下面三个注解：

- `@SpringBootConfiguration`
- `@EnableAutoConfiguration`
- `@ComponentScan`

其中 `@EnableAutoConfiguration` 是关键(启用自动配置)，内部实际上就去加载 `META-INF/spring.factories` 文件的信息，然后筛选出以 `EnableAutoConfiguration` 为key的数据到IOC容器中，实现自动配置功能！



官网文档参考：

- docs.spring.io/spring-b...

英语不好的同学可以像我一样，对照着来看：

最后

> 乐于输出**干货**的Java技术公众号：Java3y。公众号内有200多篇**原创**技术文章、海量视频精美脑图，不妨来**关注**一下！