SpringBoot自动化配置的注解开关原理

2016-11-13 | springboot

在之前我们分析SpringBoot的自动化配置原理的时候，分析了freemarker的自动化配置类FreeMarkerAutoConfiguration，这个自动化配置类需要classloader中的一些类需要存在并且在其他的一些配置类之后进行加载。

但是还存在一些自动化配置类，它们需要在使用一些注解开关的情况下才会生效。比如spring-boot-starter-batch里的@EnableBatchProcessing注解、@EnableCaching等。

一个需求

在分析这些开关的原理之前，我们来看一个需求：

> 定义一个Annotation，让使用了这个Annotaion的应用程序自动化地注入一些类或者做一些底层的事情。

我们会使用Spring提供的@Import注解配合一个配置类来完成。

我们以一个最简单的例子来完成这个需求：定义一个注解EnableContentService，使用了这个注解的程序会自动注入ContentService这个bean。

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@Import(ContentConfiguration.class)
public @interface EnableContentService {}

public interface ContentService {
    void doSomething();
}

public class SimpleContentService implements ContentService {
    @Override
    public void doSomething() {
        System.out.println("do some simple things");
    }
}
```

然后在应用程序的入口加上@EnableContentService注解。

这样的话，ContentService就被注入进来了。SpringBoot也就是用这个完成的。只不过它用了更加高级点的ImportSelector。

ImportSelector的使用

用了ImportSelector之后，我们可以在Annotation上添加一些属性，然后根据属性的不同加载不同的bean。

我们在@EnableContentService注解添加属性policy，同时Import一个Selector。

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
```

```
@Import(ContentImportSelector.class)
public @interface EnableContentService {
    String policy() default "simple";
}
```

这个ContentImportSelector根据EnableContentService注解里的policy加载不同的bean。

```
public class ContentImportSelector implements ImportSelector {

    @Override
    public String[] selectImports(AnnotationMetadata importingClassMetadata) {
        Class<?> annotationType = EnableContentService.class;
        AnnotationAttributes attributes = AnnotationAttributes.fromMap(importingClassMetadata.getAnnotationAttr
                annotationType.getName(), false));
        String policy = attributes.getString("policy");
        if ("core".equals(policy)) {
            return new String[] { CoreContentConfiguration.class.getName() };
        } else {
            return new String[] { SimpleContentConfiguration.class.getName() };
        }
    }

}
```

CoreContentService和CoreContentConfiguration如下：

```
public class CoreContentService implements ContentService {
    @Override
    public void doSomething() {
        System.out.println("do some import things");
    }
}

public class CoreContentConfiguration {
    @Bean
    public ContentService contentService() {
        return new CoreContentService();
    }
}
```

这样的话，如果在@EnableContentService注解的policy中使用core的话，应用程序会自动加载CoreContentService，否则会加载SimpleContentService。

ImportSelector在SpringBoot中的使用

SpringBoot里的ImportSelector是通过SpringBoot提供的@EnableAutoConfiguration这个注解里完成的。

这个@EnableAutoConfiguration注解可以显式地调用，否则它会在@SpringBootApplication注解中隐式地被调用。

@EnableAutoConfiguration注解中使用了EnableAutoConfigurationImportSelector作为ImportSelector。下面这段代码就是EnableAutoConfigurationImportSelector中进行选择的具体代码：

```
@Override
public String[] selectImports(AnnotationMetadata metadata) {
    try {
        AnnotationAttributes attributes = getAttributes(metadata);
        List<String> configurations = getCandidateConfigurations(metadata,
                attributes);
        configurations = removeDuplicates(configurations); // 删除重复的配置
        Set<String> exclusions = getExclusions(metadata, attributes); // 去掉需要exclude的配置
        configurations.removeAll(exclusions);
```

```
            configurations = sort(configurations); // 排序
            recordWithConditionEvaluationReport(configurations, exclusions);
            return configurations.toArray(new String[configurations.size()]);
        }
        catch (IOException ex) {
            throw new IllegalStateException(ex);
        }
    }
```

其中getCandidateConfigurations方法将获取配置类：

```
    protected List<String> getCandidateConfigurations(AnnotationMetadata metadata,
            AnnotationAttributes attributes) {
        return SpringFactoriesLoader.loadFactoryNames(
                getSpringFactoriesLoaderFactoryClass(), getBeanClassLoader());
    }
```

SpringFactoriesLoader.loadFactoryNames方法会根据FACTORIES_RESOURCE_LOCATION这个静态变量从所有的jar包中读取META-INF/spring.factories文件信息：

```
    public static List<String> loadFactoryNames(Class<?> factoryClass, ClassLoader classLoader) {
        String factoryClassName = factoryClass.getName();
        try {
            Enumeration<URL> urls = (classLoader != null ? classLoader.getResources(FACTORIES_RESOURCE_LOCATION) :
                    ClassLoader.getSystemResources(FACTORIES_RESOURCE_LOCATION));
            List<String> result = new ArrayList<String>();
            while (urls.hasMoreElements()) {
                URL url = urls.nextElement();
                Properties properties = PropertiesLoaderUtils.loadProperties(new UrlResource(url));
                String factoryClassNames = properties.getProperty(factoryClassName); // 只会过滤出key为factoryClassNa
                result.addAll(Arrays.asList(StringUtils.commaDelimitedListToStringArray(factoryClassNames)));
            }
            return result;
        }
        catch (IOException ex) {
            throw new IllegalArgumentException("Unable to load [" + factoryClass.getName() +
                    "] factories from location [" + FACTORIES_RESOURCE_LOCATION + "]", ex);
        }
    }
```

getCandidateConfigurations方法中的getSpringFactoriesLoaderFactoryClass方法返回的是EnableAutoConfiguration.class，所以会过滤出key为org.springframework.boot.autoconfigure.EnableAutoConfiguration的值。

下面这段配置代码就是autoconfigure这个jar包里的spring.factories文件的一部分内容（有个key为org.springframework.boot.autoconfigure.EnableAutoConfiguration，所以会得到这些AutoConfiguration）：

```
# Initializers
org.springframework.context.ApplicationContextInitializer=\
org.springframework.boot.autoconfigure.logging.AutoConfigurationReportLoggingInitializer

# Application Listeners
org.springframework.context.ApplicationListener=\
org.springframework.boot.autoconfigure.BackgroundPreinitializer

# Auto Configure
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
org.springframework.boot.autoconfigure.admin.SpringApplicationAdminJmxAutoConfiguration,\
org.springframework.boot.autoconfigure.aop.AopAutoConfiguration,\
org.springframework.boot.autoconfigure.amqp.RabbitAutoConfiguration,\
org.springframework.boot.autoconfigure.MessageSourceAutoConfiguration,\
```

当然了，这些AutoConfiguration不是所有都会加载的，会根据AutoConfiguration上的@ConditionalOnClass等条件判断是否加载。

上面这个例子说的读取properties文件的时候只会过滤出key为org.springframework.boot.autoconfigure.EnableAutoConfiguration的值。

SpringBoot内部还有一些其他的key用于过滤得到需要加载的类：

- org.springframework.test.context.TestExecutionListener
- org.springframework.beans.BeanInfoFactory
- org.springframework.context.ApplicationContextInitializer
- org.springframework.context.ApplicationListener
- org.springframework.boot.SpringApplicationRunListener
- org.springframework.boot.env.EnvironmentPostProcessor
- org.springframework.boot.env.PropertySourceLoader

#spring #springboot

SpringBatch中的retry和skip机制实现分析

SpringBoot编写自定义的starter