

深入学习Spring Boot自动装配

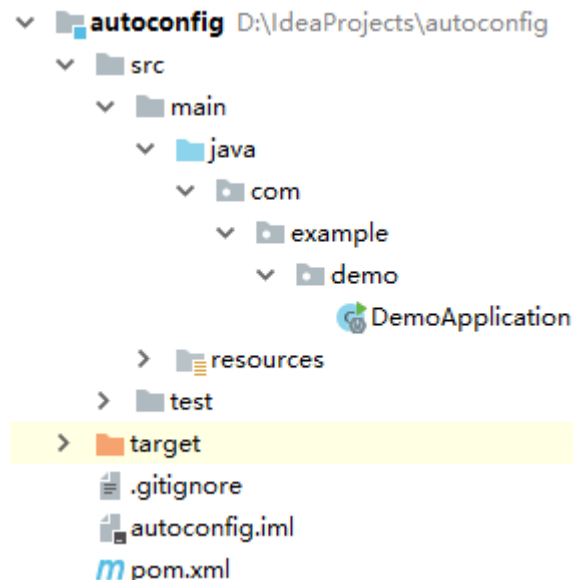
📅 2018-09-02 | 🖨 Visit count 604647

模式注解

Stereotype Annotation俗称为模式注解，Spring中常见的模式注解有 `@Service`，`@Repository`，`@Controller` 等，它们都“派生”自 `@Component` 注解。我们都知道，凡是被 `@Component` 标注的类都会被Spring扫描并纳入到IOC容器中，那么由 `@Component` 派生的注解所标注的类也会被扫描到IOC容器中。下面我们主要通过自定义模式注解来了解 `@Component` 的“派生性”和“层次性”。

📌 @Component “派生性”

新建一个Spring Boot工程，Spring Boot版本为2.1.0.RELEASE，`artifactId`为`autoconfig`，并引入`spring-boot-starter-web`依赖。项目结构如下所示：



在 `com.example.demo` 下新建 `annotation` 包，然后创建一个 `FirstLevelService` 注解：

```
1  @Target({ElementType.TYPE})
2  @Retention(RetentionPolicy.RUNTIME)
3  @Documented
4  @Service
5  public @interface FirstLevelService {
6      String value() default "";
7  }
```

这个注解定义由 `@Service` 标注，查看 `@Service` 的源码会发现其被 `@Component` 注解标注，所以它们的层次关系为：

```
1  └─@Component
```

```

2      └─@Service
3      └─@FirstLevelService

```

即 `@FirstLevelService` 为 `@Component` 派生出来的模式注解，我们来测试一下被它标注的类是否能够被扫描到IOC容器中：

在 `com.example.demo` 下新建 `service` 包，然后创建一个 `TestService` 类：

```

1  @FirstLevelService
2  public class TestService {
3  }

```

在 `com.example.demo` 下新建 `bootstrap` 包，然后创建一个 `ServiceBootstrap` 类，用于测试注册 `TestService` 并从IOC容器中获取它：

```

1  @ComponentScan("com.example.demo.service")
2  public class ServiceBootstrap {
3
4      public static void main(String[] args) {
5          ConfigurableApplicationContext context = new SpringApplicationBuilder(ServiceBootstrap.class)
6              .web(WebApplicationType.NONE)
7              .run(args);
8          TestService testService = context.getBean("testService", TestService.class);
9          System.out.println("TestService Bean: " + testService);
10         context.close();
11     }
12 }

```

运行该类的main方法，控制台输出如下：

```
TestService Bean: com.example.demo.service.TestService@4a668b6e
```

📌 @Component “层次性”

我们在 `com.example.demo.annotation` 路径下再创建一个 `SecondLevelService` 注解定义，该注解由上面的 `@FirstLevelService` 标注：

```

1  @Target({ElementType.TYPE})
2  @Retention(RetentionPolicy.RUNTIME)
3  @Documented
4  @FirstLevelService
5  public @interface SecondLevelService {
6      String value() default "";
7  }

```

这时候层次关系为：

```

1    └─@Component
2        └─@Service
3            └─@FirstLevelService
4                └─@SecondLevelService

```

我们将 `TestService` 上的注解换成 `@SecondLevelService`，然后再次运行 `ServiceBootstrap` 的 `main` 方法，输出如下：

TestService Bean: com.example.demo.service.TestService@4a668b6e

可见结果也是成功的。

这里有一点需要注意的是：`@Component` 注解只包含一个 `value` 属性定义，所以其“派生”的注解也只能包含一个 `value` 属性定义。

@Enable模块驱动

`@Enable` 模块驱动在Spring Framework 3.1后开始支持。这里的模块通俗的来说就是一些为了实现某个功能的组件的集合。通过 `@Enable` 模块驱动，我们可以开启相应的模块功能。

`@Enable` 模块驱动可以分为“注解驱动”和“接口编程”两种实现方式，下面逐一进行演示：

■ 注解驱动

Spring中，基于注解驱动的示例可以查看 `@EnableWebMvc` 源码：

```

1  @Retention(RetentionPolicy.RUNTIME)
2  @Target({ElementType.TYPE})
3  @Documented
4  @Import({DelegatingWebMvcConfiguration.class})
5  public @interface EnableWebMvc {
6  }

```

该注解通过 `@Import` 导入一个配置类 `DelegatingWebMvcConfiguration`：

```

@Configuration
public class DelegatingWebMvcConfiguration extends WebMvcConfigurationSupport {
    private final WebMvcConfigurerComposite configurers = new WebMvcConfigurerComposite();
}

public DelegatingWebMvcConfiguration() {
}

```

该配置类又继承自 `WebMvcConfigurationSupport`，里面定义了一些Bean的声明。

✓ 所以，基于注解驱动的 `@Enable` 模块驱动其实就是通过 `@Import` 来导入一个配置类，以此实现相应模块的组件注册，当这些组件注册到IOC容器中，这个模块对应的功能也就可以使用了。

我们来定义一个基于注解驱动的 `@Enable` 模块驱动。

在 `com.example.demo` 下新建 `configuration` 包，然后创建一个 `HelloWorldConfiguration` 配置类：

```
1  @Configuration
2  public class HelloWorldConfiguration {
3
4      @Bean
5      public String hello() {
6          return "hello world";
7      }
8  }
```

这个配置类里定义了一个名为 `hello` 的Bean，内容为 `hello world`。

在 `com.example.demo.annotation` 下创建一个 `EnableHelloWorld` 注解定义：

```
1  @Target({ElementType.TYPE})
2  @Retention(RetentionPolicy.RUNTIME)
3  @Documented
4  @Import(HelloWorldConfiguration.class)
5  public @interface EnableHelloWorld {
6  }
```

我们在该注解类上通过 `@Import` 导入了刚刚创建的配置类。

接着在 `com.example.demo.bootstrap` 下创建一个 `TestEnableBootstap` 启动类来测试

`@EnableHelloWorld` 注解是否生效：

```
1  @EnableHelloWorld
2  public class TestEnableBootstap {
3      public static void main(String[] args) {
4          ConfigurableApplicationContext context = new SpringApplicationBuilder(TestEnable
5              .web(WebApplicationType.NONE)
6              .run(args);
7          String hello = context.getBean("hello", String.class);
8          System.out.println("hello Bean: " + hello);
9          context.close();
10     }
11 }
```

运行该类的main方法，控制台输出如下：

hello Bean: hello world

说明我们自定义的基于注解驱动的 `@EnableHelloWorld` 是可行的。

📌 接口编程

除了使用上面这个方式外，我们还可以通过接口编程的方式来实现 `@Enable` 模块驱动。Spring中，基于接口编程方式的有 `@EnableCaching` 注解，查看其源码：

```
1  @Target({ElementType.TYPE})
2  @Retention(RetentionPolicy.RUNTIME)
3  @Documented
4  @Import({CachingConfigurationSelector.class})
5  public @interface EnableCaching {
6      boolean proxyTargetClass() default false;
7
8      AdviceMode mode() default AdviceMode.PROXY;
9
10     int order() default 2147483647;
11 }
```

`EnableCaching` 注解通过 `@Import` 导入了 `CachingConfigurationSelector` 类，该类间接实现了 `ImportSelector` 接口，在 [深入学习Spring组件注册](#) 中，我们曾介绍了可以通过 `ImportSelector` 来实现组件注册。

✓ 所以通过接口编程实现 `@Enable` 模块驱动的本质是：通过 `@Import` 来导入接口 `ImportSelector` 实现类，该实现类里可以定义需要注册到IOC容器中的组件，以此实现相应模块对应组件的注册。

接下来我们根据这个思路来自个实现一遍：

在 `com.example.demo` 下新建 `selector` 包，然后在该路径下新建一个 `HelloWorldImportSelector` 实现 `ImportSelector` 接口：

```
1  public class HelloWorldImportSelector implements ImportSelector {
2      @Override
3      public String[] selectImports(AnnotationMetadata importingClassMetadata) {
4          return new String[]{HelloWorldConfiguration.class.getName()};
5      }
6  }
```

如果看不懂上面这段代码含义的朋友可以阅读[深入学习Spring组件注册一文](#)。

接着我们修改 `EnableHelloWorld`：

```
1  @Target({ElementType.TYPE})
2  @Retention(RetentionPolicy.RUNTIME)
3  @Documented
4  @Import(HelloWorldImportSelector.class)
5  public @interface EnableHelloWorld {
6  }
```

上面导入的是 *HelloWorldImportSelector* , 而非 *HelloWorldConfiguration* 。

再次运行 *TestEnableBootstap* 的main方法, 你会发现输出是一样的。

自动装配

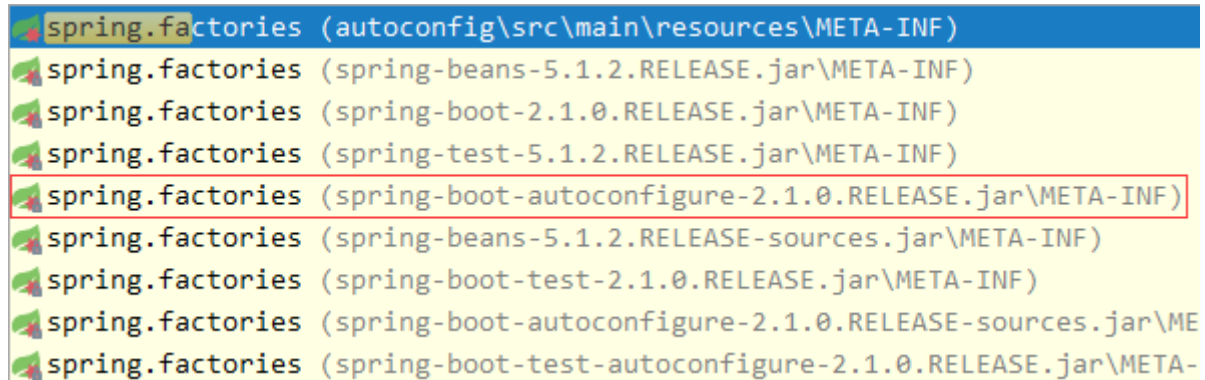
Spring Boot中的自动装配技术底层主要用到了下面这些技术:

1. Spring 模式注解装配
2. Spring @Enable 模块装配
3. Spring 条件装配装 (深入学习Spring组件注册中有介绍)
4. Spring 工厂加载机制

Spring 工厂加载机制的实现类为 *SpringFactoriesLoader* , 查看其源码:

```
public final class SpringFactoriesLoader {  
    public static final String FACTORIES_RESOURCE_LOCATION = "META-INF/spring.factories";  
    private static final Log logger = LoggerFactory.getLog(SpringFactoriesLoader.class);  
    private static final Map<ClassLoader, MultiValueMap<String, String>> cache = new ConcurrentReferenceHashMap();  
  
    private SpringFactoriesLoader() {  
    }  
}
```

该类的方法会读取META-INF目录下的spring.factories配置文件, 我们查看spring-boot-autoconfigure-2.1.0.RELEASE.jar下的该文件:



```
spring.factories (autoconfig\src\main\resources\META-INF)  
spring.factories (spring-beans-5.1.2.RELEASE.jar\META-INF)  
spring.factories (spring-boot-2.1.0.RELEASE.jar\META-INF)  
spring.factories (spring-test-5.1.2.RELEASE.jar\META-INF)  
spring.factories (spring-boot-autoconfigure-2.1.0.RELEASE.jar\META-INF)  
spring.factories (spring-beans-5.1.2.RELEASE-sources.jar\META-INF)  
spring.factories (spring-boot-test-2.1.0.RELEASE.jar\META-INF)  
spring.factories (spring-boot-autoconfigure-2.1.0.RELEASE-sources.jar\ME  
spring.factories (spring-boot-test-autoconfigure-2.1.0.RELEASE.jar\META-
```

```
# 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.batch.BatchAutoConfiguration,\
org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration,\
org.springframework.boot.autoconfigure.cassandra.CassandraAutoConfiguration,\
org.springframework.boot.autoconfigure.cloud.CloudServiceConnectorsAutoConfiguration,\
org.springframework.boot.autoconfigure.context.ConfigurationPropertiesAutoConfiguration,\
org.springframework.boot.autoconfigure.context.MessageSourceAutoConfiguration,\
org.springframework.boot.autoconfigure.context.PropertyPlaceholderAutoConfiguration,\
org.springframework.boot.autoconfigure.couchbase.CouchbaseAutoConfiguration,\
org.springframework.boot.autoconfigure.dao.PersistenceExceptionTranslationAutoConfiguration,\
org.springframework.boot.autoconfigure.data.cassandra.CassandraDataAutoConfiguration,\
org.springframework.boot.autoconfigure.data.cassandra.CassandraReactiveDataAutoConfiguration,\
org.springframework.boot.autoconfigure.data.cassandra.CassandraReactiveRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.cassandra.CassandraRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.couchbase.CouchbaseDataAutoConfiguration,\
org.springframework.boot.autoconfigure.data.couchbase.CouchbaseReactiveDataAutoConfiguration,\
org.springframework.boot.autoconfigure.data.couchbase.CouchbaseReactiveRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.couchbase.CouchbaseRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.elasticsearch.ElasticsearchAutoConfiguration,\
org.springframework.boot.autoconfigure.data.elasticsearch.ElasticsearchDataAutoConfiguration,\
org.springframework.boot.autoconfigure.data.elasticsearch.ElasticsearchRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.jdbc.JdbcRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.jpa.JpaRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.ldap.LdapRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.mongo.MongoDataAutoConfiguration,\
org.springframework.boot.autoconfigure.data.mongo.MongoReactiveDataAutoConfiguration,\
org.springframework.boot.autoconfigure.data.mongo.MongoReactiveRepositoriesAutoConfiguration,\
```

当启动类被 `@EnableAutoConfiguration` 标注后，上面截图中的所有类Spring都会去扫描，看是否可以纳入到IOC容器中进行管理。

比如我们查看 `org.springframework.boot.autoconfigure.data.redis.RedisAutoConfiguration` 的源码：

```
@Configuration
@ConditionalOnClass(RedisOperations.class)
@EnableConfigurationProperties(RedisProperties.class)
@Import({ LettuceConnectionConfiguration.class, JedisConnectionConfiguration.class })
public class RedisAutoConfiguration {
```

可看到该类上标注了一些注解，其中 `@Configuration` 为模式注解，`@EnableConfigurationProperties` 为模块装配技术，`ConditionalOnClass` 为条件装配技术。这和我们上面列出的Spring Boot自动装配底层主要技术一致，所以我们可以根据这个思路来自定义一个自动装配实现。

新建一个配置类 `HelloWorldAutoConfiguration`：

```
1  @Configuration
2  @EnableHelloWorld
3  @ConditionalOnProperty(name = "helloworld", havingValue = "true")
4  public class HelloWorldAutoConfiguration {
5  }
```

然后在resources目录下新建META-INF目录，并创建spring.factories文件：

```
1  # Auto Configure
2  org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
3  com.example.demo.configuration.HelloWorldAutoConfiguration
```


接着在配置文件`application.properties`中添加 `helloworld=true` 配置

```
1 helloworld=true
```

最后创建 `EnableAutoConfigurationBootstrap`，测试下 `HelloWorldAutoConfiguration` 是否生效：

```
1 @EnableAutoConfiguration
2 public class EnableAutoConfigurationBootstrap {
3
4     public static void main(String[] args) {
5         ConfigurableApplicationContext context = new SpringApplicationBuilder(EnableAutoConfiguration)
6             .web(WebApplicationType.NONE)
7             .run(args);
8         String hello = context.getBean("hello", String.class);
9         System.out.println("hello Bean: " + hello);
10        context.close();
11    }
12 }
```

运行该`main`方法，控制台输出如下：

```
hello Bean: hello world
```

说明我们自定义的自动装配已经成功了。

下面简要分析下代码的运行逻辑：

1. Spring 的工厂加载机制会自动读取META-INF目录下`spring.factories`文件内容；
2. 我们在`spring.factories`定义了：

```
1 org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
2 com.example.demo.configuration.HelloWorldAutoConfiguration
```

我们在测试类上使用了 `@EnableAutoConfiguration` 注解标注，那么 `HelloWorldAutoConfiguration` 就会被Spring扫描，看是否符合要求，如果符合则纳入到IOC容器中；

3. `HelloWorldAutoConfiguration` 上的 `@ConditionalOnProperty` 的注解作用为：当配置文件中配置了 `helloworld=true`（我们确实添加了这个配置，所以符合要求）则这个类符合扫描规则；
`@EnableHelloWorld` 注解是我们前面例子中自定义的模块驱动注解，其引入了`hello`这个Bean，所以IOC容器中便会存在`hello`这个Bean了；
4. 通过上面的步骤，我们就可以通过上下文获取到`hello`这个Bean了。

源码链接：<https://github.com/wuyouzhuguli/SpringAll/tree/master/44.Spring-Boot-Autoconfiguration>