springboot 是怎么做到简化配置的?

作者：jianzh5
原文链接：https://ld246.com/article/1582807434220
来源网站：链滴

00 前言

惭愧地狠，前几天的一个面试问到springboot是怎么做到简化配置的，我就说了个事先约定，内部实没有答上来。用springboot也用了一年多，从来没想着去看看springboot是怎么实现简化配置，让家爱用这个玩意儿的。

然后搜了下，说是加载jar包下的META-INF/spring.factories文件，但是又有个面试官问我，这里面配置代表了什么意思呢?

我又瞎说了一通。

今天就找了个下资料，学习了下，然后自己点开源码看了下，发现主脉络写的很清晰，并不是很难懂就此写一篇文章，加深下自己的记忆吧。

也为我的两次面试哀悼。

可能我不是属于考试型的吧。唉。

面试还是要懂些原理性的东西。

有一个面试官说的好，人有两种能力，一种是面试时表现出的能力，一种是工作中解决实际问题的能。

我的第一种能力太弱了，都是靠第二种能力撑着的，但是第二种能力面试时没法面试出来，所以我每换工作时都挺痛苦的。

废话不多说了，开启今天的springboot自动化配置之旅吧。

01 EnableAutoConfiguration

springboot的启动类上有个注解SpringBootApplication，点开这个注解，可以看到它的内部实现，由多个注解组成的。

```
@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.class}
)}
)
public @interface SpringBootApplication {

}
```

在这其中，可以看到有个EnableAutoConfiguration，就是负责开启我们的自动配置。

```
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@AutoConfigurationPackage
@Import({AutoConfigurationImportSelector.class})
public @interface EnableAutoConfiguration {
    String ENABLED_OVERRIDE_PROPERTY = "spring.boot.enableautoconfiguration";

    Class<?>[] exclude() default {};

    String[] excludeName() default {};
}
```

打开EnableAutoConfiguration，可以看到里面是import了AutoConfigurationImportSelector这一个类。

继续追踪这个类，AutoConfigurationImportSelector类里主要看一个selectImports方法，这个方的调用链如下：

```
selectImports
    ->getAutoConfigurationEntry
        -> getCandidateConfigurations
            ->SpringFactoriesLoader.loadFactoryNames
```

SpringFactoriesLoader.loadFactoryNames方法如下：

```
protected List<String> getCandidateConfigurations(AnnotationMetadata metadata, AnnotatinAttributes attributes) {
    List<String> configurations = SpringFactoriesLoader.loadFactoryNames(this.getSpringFactriesLoaderFactoryClass(), this.getBeanClassLoader());
    Assert.notEmpty(configurations, "No auto configuration classes found in META-INF/spring.actories. If you are using a custom packaging, make sure that file is correct.");
    return configurations;
}
```

可以看到断言中说到是去META-INF/spring.factories这个文件下去寻找有没有自动配置类。

我们再点开SpringFactoriesLoader.loadFactoryNames这个方法确认下。

```
public static List<String> loadFactoryNames(Class<?> factoryType, @Nullable ClassLoader clssLoader) {
    String factoryTypeName = factoryType.getName();
    return (List)loadSpringFactories(classLoader).getOrDefault(factoryTypeName, Collections.eptyList());
}

private static Map<String, List<String>> loadSpringFactories(@Nullable ClassLoader classLoder) {
    MultiValueMap<String, String> result = (MultiValueMap)cache.get(classLoader);
    if (result != null) {
        return result;
    } else {
        try {
            Enumeration<URL> urls = classLoader != null ? classLoader.getResources("META-INF/
```

```
pring.factories") : ClassLoader.getSystemResources("META-INF/spring.factories");
            LinkedMultiValueMap result = new LinkedMultiValueMap();

            while(urls.hasMoreElements()) {
                URL url = (URL)urls.nextElement();
                UrlResource resource = new UrlResource(url);
                Properties properties = PropertiesLoaderUtils.loadProperties(resource);
                Iterator var6 = properties.entrySet().iterator();

                while(var6.hasNext()) {
                    Entry<?, ?> entry = (Entry)var6.next();
                    String factoryTypeName = ((String)entry.getKey()).trim();
                    String[] var9 = StringUtils.commaDelimitedListToStringArray((String)entry.getVal
e());
                    int var10 = var9.length;

                    for(int var11 = 0; var11 < var10; ++var11) {
                        String factoryImplementationName = var9[var11];
                        result.add(factoryTypeName, factoryImplementationName.trim());
                    }
                }
            }

            cache.put(classLoader, result);
            return result;
        } catch (IOException var13) {
            throw new IllegalArgumentException("Unable to load factories from location [META-I
F/spring.factories]", var13);
        }
    }
}
```

可以看到确实调用了loadSpringFactories这个方法，loadSpringFactories方法中开头的这段代码

```
Enumeration<URL> urls = classLoader != null ? classLoader.getResources("META-INF/spring.f
ctories") : ClassLoader.getSystemResources("META-INF/spring.factories");
```

即告诉我们，它是要找到所有META-INF/spring.factories下的文件，然后加载进来，使用Properties
oaderUtils.loadProperties方法读取其中的文件配置。

02 spring.factories

哪些jar包下有META-INF/spring.factories文件呢？一般是在springboot的核心类及XXX-spring-boo
-autoconfigure或spring-boot-autoconfigure-XXX这样的jar包中。

本次我们只说EnableAutoConfiguration，打开spring-boot-autoconfigure-2.2.1.RELEASE.jar，
个就是springboot中最重要的自动配置包。

打开下面的META-INF/spring.factories 文件，可以发现这个文件也是也是一组一组的key=value的
式,与我们常用的properties文件没有什么区别。

截取其中部分内容如下：

```
# 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,\
.....
```

这里EnableAutoConfiguration这个key对应的value很长，以逗号分割。

可以看到这里的value其实都是一个个类，那么它们是在哪里呢?

再返回上一层看看spring-boot-autoconfigure-2.2.1.RELEASE.jar下的org.springframework.boot.autoconfigure代码包。

这里，有各种各样事先写好的配置类。

我们思考下，为什么在application.properties中写上诸如spring.datasource.url=XXX的配置就能加jdbc了?

03 application.properties

下面是我们在application.properties常用的加载jdbc的方式:

```
spring.datasource.url=jdbc:oracle:thin:@192.168.1.7:1521:orcl
spring.datasource.username=yaomaomao
spring.datasource.password=lyaoshen369
spring.datasource.type=com.alibaba.druid.pool.DruidDataSource
spring.datasource.driverClassName=oracle.jdbc.driver.OracleDriver
```

这里配置的意思大家都应该明白，但是为什么它能起作用呢?

我们先从spring.factories文件中 org.springframework.boot.autoconfigure.EnableAutoConfiguration 这个key中找到 jdbc相关的value，如下，找到了这些:

```
.....
org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration,\
org.springframework.boot.autoconfigure.jdbc.JdbcTemplateAutoConfiguration,\
org.springframework.boot.autoconfigure.jdbc.JndiDataSourceAutoConfiguration,\
org.springframework.boot.autoconfigure.jdbc.XADataSourceAutoConfiguration,\
org.springframework.boot.autoconfigure.jdbc.DataSourceTransactionManagerAutoConfiguration,\
.....
```

打开 org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration 这个类，发这样的代码

```
@Configuration(
    proxyBeanMethods = false
)
@ConditionalOnClass({DataSource.class, EmbeddedDatabaseType.class})
@EnableConfigurationProperties({DataSourceProperties.class})
@Import({DataSourcePoolMetadataProvidersConfiguration.class, DataSourceInitializationConf
```

```
guration.class})
public class DataSourceAutoConfiguration {
    public DataSourceAutoConfiguration() {
    }
}
```

主要看这段注解

```
@EnableConfigurationProperties({DataSourceProperties.class})
```

打开DataSourceProperties这个类，发现如下内容

```
@ConfigurationProperties(
    prefix = "spring.datasource"
)
public class DataSourceProperties implements BeanClassLoaderAware, InitializingBean {
    private ClassLoader classLoader;
    private String name;
    private boolean generateUniqueName;
    private Class<? extends DataSource> type;
    private String driverClassName;
    private String url;
    private String username;
    private String password;
    private String jndiName;
    ...
}
```

至此，我们基本上是明白了@EnableAutoConfiguration是怎么起作用的，又是怎么与application.properties关联的。

04 总结

最后，在面试时，我们能不能一句话总结下，springboot是怎么做到简化配置的?

答：主要是@EnableAutoConfiguration这个注解起的作用，这个注解是间接隐藏在springboot的动类注解@SpringBootApplication中。

通过这个注解，SpringApplication.run(...)的内部就会执行selectImports()方法，寻找 META-INF/spring.factories文件，读取里面的文件配置，将事先已经写好的自动配置类有选择地加载到Spring容器，并且能按照约定的写法在application.properties中配置参数或开关。