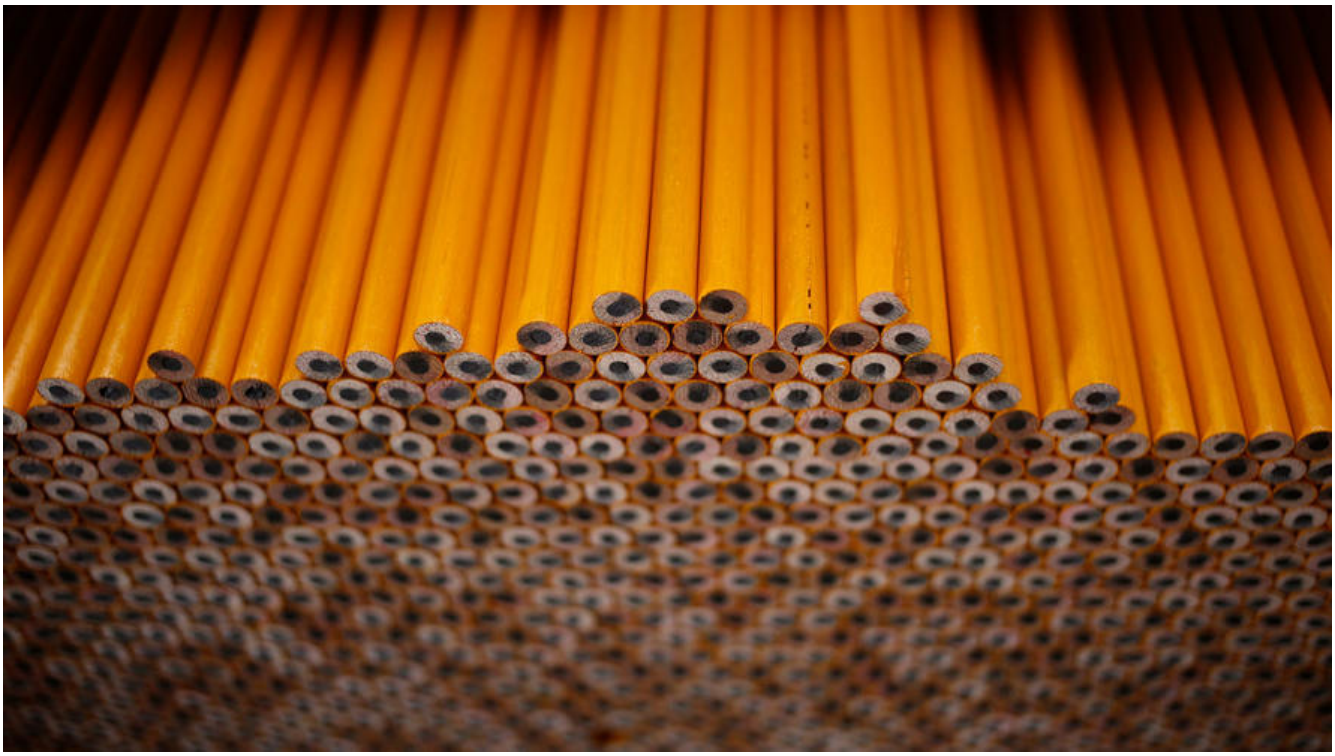
链滴

SpringBoot- 自动配置源码解析

作者：Ethan
原文链接：https://ld246.com/article/1524816262503
来源网站：链滴



接着上一篇博客《SpringBoot-快速搭建WEB工程》提出的需要分析的三个方面：我们来深入的探究SringBoot是如何在没有一个配置文件的情况下为我们启动好一个完整的WEB工程的，首先我们从@SprngBootApplication 开始这里的分析会剖出一些次要的信息沿着主干走，所以可能会有一些略过的地。以下源码截取自spring-boot-1.4.0.RELEASE

```
@Target(ElementType.TYPE) @Retention(RetentionPolicy.RUNTIME) @Documented @Inherit
d @SpringBootConfiguration @EnableAutoConfiguration @ComponentScan(excludeFilters
= @Filter(type = FilterType.CUSTOM, classes = TypeExcludeFilter.class)) public @interface Spr
ngBootApplication {}
```

可以看到这是一个复合注解：其中@Target，@Retention，@Documented，@Inherited这四个解不做过多的解释。

@ComponentScan这个是Spring很常用的注解也不做解释。

下面来看一看@SpringBootConfiguration：

```
@Target(ElementType.TYPE) @Retention(RetentionPolicy.RUNTIME) @Documented @Confi
uration public @interface SpringBootConfiguration {}
```

这个注解，有作用的也是@Configuration：这是标注当前类为：JavaConfig类

现在就来看看最后的一个注解@EnableAutoConfiguration

```
@Target(ElementType.TYPE) @Retention(RetentionPolicy.RUNTIME) @Documented @Inherit
d @AutoConfigurationPackage @Import(EnableAutoConfigurationImportSelector.class) publ
c @interface EnableAutoConfiguration {}
```

这个注解上@Import(EnableAutoConfigurationImportSelector.class)代表引入其它的Spring的JavConfig接着进入EnableAutoConfigurationImportSelector.class

关注一下以下的方法：

```
@Override

public String[] selectImports(AnnotationMetadata metadata) {
    if (!isEnabled(metadata)) {
        return NO_IMPORTS;
    }
    try {
        AnnotationAttributes attributes = getAttributes(metadata);
        List<String> configurations = getCandidateConfigurations(metadata,
                attributes);
        configurations = removeDuplicates(configurations);
        Set<String> exclusions = getExclusions(metadata, attributes);
        configurations.removeAll(exclusions);
        configurations = sort(configurations);
        recordWithConditionEvaluationReport(configurations, exclusions);
        return configurations.toArray(new String[configurations.size()]);
    }
    catch (IOException ex) {
        throw new IllegalStateException(ex);
    }
}```
```

进入：List configurations = getCandidateConfigurations(metadata,attributes);

```
protected List<String> getCandidateConfigurations(AnnotationMetadata metadata,Annotati
nAttributes attributes) {


List<String> configurations = SpringFactoriesLoader.loadFactoryNames(
        getSpringFactoriesLoaderFactoryClass(), getBeanClassLoader());
Assert.notEmpty(configurations,
        "No auto configuration classes found in META-INF/spring.factories. If you "
                + "are using a custom packaging, make sure that file is correct.");
return configurations;

}```
```

在进入：List configurations = SpringFactoriesLoader.loadFactoryNames(

getSpringFactoriesLoaderFactoryClass(), getBeanClassLoader());

代码如下:

```
public static List<String> loadFactoryNames(Class<?> factoryClass, ClassLoader classLoader) {

    String factoryClassName = factoryClass.getName();
    try {
        Enumeration<URL> urls = (classLoader != null ? classLoader.getResources(FACTORIES_R
SOURCE_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);
            result.addAll(Arrays.asList(StringUtils.commaDelimitedListToStringArray(factoryClassN
mes)));
        }
        return result;
    }
    catch (IOException ex) {
        throw new IllegalArgumentException(
        "Unable to load [" + factoryClass.getName() +
        + FACTORIES_RESOURCE_LOCATION + "]", ex);
    }
}
public static final String FACTORIES_RESOURCE_LOCATION =
"META-INF/spring.factories";```
```

在上面的代码可以看到自动配置器会跟根据传入的factoryClass.getName()到spring.factories的文中找到相应的key，从而加载里面的类但我们打开spring-boot-autoconfigure-1.4.0.RELEASE.jar里的spring.factories可以发现很多key,那么这里是怎么样的一个加载的流程呢这里只把代码贴出来不进讲解，下一篇博客会对SpringBoot启动的整个流程进入深入的分析《[SpringBoot-启动流程分析](htt://blog.csdn.net/doegoo/article/details/52471310)》.

```
public ConfigurableApplicationContext run(String... args) {


StopWatch stopWatch = new StopWatch();
stopWatch.start();
ConfigurableApplicationContext context = null;
configureHeadlessProperty();
SpringApplicationRunListeners listeners = getRunListeners(args);
listeners.started();
try {
    ApplicationArguments applicationArguments = new DefaultApplicationArguments(
            args);
    ConfigurableEnvironment environment = prepareEnvironment(listeners,
            applicationArguments);
    Banner printedBanner = printBanner(environment);
    context = createApplicationContext();
    prepareContext(context, environment, listeners, applicationArguments,
            printedBanner);
    refreshContext(context);
    afterRefresh(context, applicationArguments);
    listeners.finished(context, null);
    stopWatch.stop();
    if (this.logStartupInfo) {
        new StartupInfoLogger(this.mainApplicationClass)
                .logStarted(getApplicationLog(), stopWatch);
    }
    return context;
}
catch (Throwable ex) {
```

```
    handleRunFailure(context, listeners, ex);
    throw new IllegalStateException(ex);
}

}```
```

这篇文章只是说明自动配置功能，所以这里只指明自动配置是的加载是发生在refreshContext(context;这一句。

由于篇幅原因这里截取了一小部分，完整的请到spring-boot-autoconfigure-1.4.0.RELEASE.jar包里看spring.factories文件。

```
# Auto Configure

org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
org.springframework.boot.autoconfigure.admin.SpringApplicationAdminJmxAutoConfigurati
n,\
org.springframework.boot.autoconfigure.aop.AopAutoConfiguration,\
org.springframework.boot.autoconfigure.amqp.RabbitAutoConfiguration,\
org.springframework.boot.autoconfigure.MessageSourceAutoConfiguration,\
org.springframework.boot.autoconfigure.PropertyPlaceholderAutoConfiguration,\
org.springframework.boot.autoconfigure.batch.BatchAutoConfiguration,\
org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration,\
org.springframework.boot.autoconfigure.cassandra.CassandraAutoConfiguration,\
org.springframework.boot.autoconfigure.cloud.CloudAutoConfiguration,\
org.springframework.boot.autoconfigure.context.ConfigurationPropertiesAutoConfiguration,\
org.springframework.boot.autoconfigure.couchbase.CouchbaseAutoConfiguration,\
org.springframework.boot.autoconfigure.dao.PersistenceExceptionTranslationAutoConfigurat
on,\
org.springframework.boot.autoconfigure.data.cassandra.CassandraDataAutoConfiguration,\
org.springframework.boot.autoconfigure.data.cassandra.CassandraRepositoriesAutoConfigura
ion,\
org.springframework.boot.autoconfigure.data.couchbase.CouchbaseDataAutoConfiguration,\
org.springframework.boot.autoconfigure.data.couchbase.CouchbaseRepositoriesAutoConfigu
ation,\
org.springframework.boot.autoconfigure.data.elasticsearch.ElasticsearchAutoConfiguration,\
org.springframework.boot.autoconfigure.data.elasticsearch.ElasticsearchDataAutoConfigurati
n,\
org.springframework.boot.autoconfigure.data.elasticsearch.ElasticsearchRepositoriesAutoConf
guration,\
org.springframework.boot.autoconfigure.data.jpa.JpaRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.mongo.MongoDataAutoConfiguration,\
org.springframework.boot.autoconfigure.data.mongo.MongoRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.neo4j.Neo4jDataAutoConfiguration,\
org.springframework.boot.autoconfigure.data.neo4j.Neo4jRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.solr.SolrRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.redis.RedisAutoConfiguration,\
org.springframework.boot.autoconfigure.data.redis.RedisRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.rest.RepositoryRestMvcAutoConfiguration,\
org.springframework.boot.autoconfigure.data.web.SpringDataWebAutoConfiguration```
```

这里以org.springframework.boot.autoconfigure.data.redis.RedisAutoConfiguration为例查看代如下：

```
@Configuration
@ConditionalOnClass({ JedisConnection.class, RedisOperations.class, Jedis.class })
@EnableConfigurationProperties(RedisProperties.class)
public class RedisAutoConfiguration {
@Configuration
@ConditionalOnClass(GenericObjectPool.class)
protected static class RedisConnectionConfiguration {
@Bean
@ConditionalOnMissingBean(RedisConnectionFactory.class)
public JedisConnectionFactory redisConnectionFactory()
throws UnknownHostException {
return applyProperties(createJedisConnectionFactory());
}
}
@Configuration
protected static class RedisConfiguration {
@Bean
@ConditionalOnMissingBean(name = "redisTemplate")
public RedisTemplate<Object, Object> redisTemplate(
RedisConnectionFactory redisConnectionFactory)
throws UnknownHostException {
RedisTemplate<Object, Object> template = new RedisTemplate<Object, Object>();
template.setConnectionFactory(redisConnectionFactory);
return template;
}
@Bean
@ConditionalOnMissingBean(StringRedisTemplate.class)
public StringRedisTemplate stringRedisTemplate(
RedisConnectionFactory redisConnectionFactory)
throws UnknownHostException {
StringRedisTemplate template = new StringRedisTemplate();
template.setConnectionFactory(redisConnectionFactory);
return template;
}

}
```

} ```

把类简化一下基本上就可以看出这就是一个Spring的注解版的配置

@ConditionalOnClass({ JedisConnection.class, RedisOperations.class, Jedis.class })这个注解的思是：当存在JedisConnection.class, RedisOperations.class, Jedis.class三个类时才解析RedisAutoonfiguration配置类,否则不解析这一个配置类

@ConditionalOnMissingBean(name = “redisTemplate”)这个注解的意思是如果容器中不存在nme指定的bean则创建bean注入，否则不执行

内部代码可以看出里面又定义了两个带@Configuration注解的配置类，这两个配置类会向SpringIO容器注入可能3个bean：

首先当类路径下存在(GenericObjectPool.class)时则注入JedisConnectionFactory 的实例如果Sprin容器中不存在name = “redisTemplate”的实体，则创建RedisTemplate和StringRedisTemplate例注入容器，这样在Spring的项目中，就可以用在任意的Spring管理的bean中注册用RedisTemplat和StringRedisTemplate的实例来对redis进入操作了。

通过以上分析的过程我们可以发现只要一个基于SpringBoot项目的类路径下存在JedisConnection.cla s, RedisOperations.class, Jedis.class就可以触发自动化配置,意思说我们只要在maven的项目中依赖spring-data-redis-1.7.2.RELEASE.jar和C:jedis-2.8.2.jar就可以触发自动配置,但这样不是每集成一个能都要去分析里其自动化配置类，那就代不到开箱即用的效果了。所以Spring-boot为我提供了统一的tarter可以直接配置好相关触发自动配置的所有的类的依赖集如redis的start如下：

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>```
```

这里是截取的spring-boot-starter-data-redis的源码中pom.xml文件中所有依赖:

```
<dependencies>
   <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter</artifactId>
   </dependency>
   <dependency>
      <groupId>org.springframework.data</groupId>
      <artifactId>spring-data-redis</artifactId>
   </dependency>
   <dependency>
      <groupId>redis.clients</groupId>
      <artifactId>jedis</artifactId>
   </dependency>
</dependencies>```
```

因为maven依赖的传递性，我们只要依赖starter就可以看在类路径下配置好所有的触发自动配置的所类，实现开箱即用的功能。

这里只是大概的说明了一下SpringBoot的Starter的用法，后面在说明自制自已的starter时还会做深的说明。