2019年7月3日 20:41

约定优于配置的体现：

maven的目录结构

https://docs.spring.io/spring-boot/docs/2.1.6.RELEASE/reference/html/

Spi 机制

2019年7月4日 17:54

Springboot 的基本认识

不管是 spring cloud alibaba 还是 spring cloud netflix，都是基于 springboot 这个微框架来构建的，所以我希望花一

点时间来讲一下 springboot

什么是 springboot

对于 spring 框架，我们接触得比较多的应该是 spring mvc、和 spring。而 spring 的核心在于 IOC（控制反转）和 DI（依赖注入）。而这些框架在使用的过程中会需要配置大量的 xml，或者需要做很多繁琐的配置。

springboot 框架是为了能够帮助使用 spring 框架的开发者快速高效的构建一个基于 spirng 框架以及 spring 生态

体系的应用解决方案。它是对“约定优于配置”这个理念下的一个最佳实践。因此它是一个服务于框架的框架，服务的范围是简化配置文件。

约定优于配置的体现

约定优于配置的体现主要是

1. maven 的目录结构
   a) 默认有 resources 文件夹存放配置文件
   b) 默认打包方式为 jar
2. spring-boot -starter -web 中默认包含 spring mvc 相关依赖以及内置的 tomcat 容器，使得构建一个 web 应用

更加简单

3. 默认提供 application.properties/yml 文件
4. 默认通过 spring.profiles.active 属性来决定运行环境时读取的配置文件
5. EnableAutoConfiguration 默认对于依赖的 starter 进行自动装载

从 SpringBootApplication 注解入手

为了揭开 springboot 的奥秘，我们直接从 Annotation 入手，看看@SpringBootApplication 里面，做了什么?

打开 SpringBootApplication 这个注解，可以看到它实际上是一个复合注解

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@SpringBootConfiguration - 实际上是@Configuration
@EnableAutoConfiguration
```

```
@ComponentScan(excludeFilters = {
   @Filter(type = FilterType.CUSTOM, classes = TypeExcludeFilter.class),
   @Filter(type = FilterType.CUSTOM, classes = AutoConfigurationExcludeFilter.class)
})
public @interface SpringBootApplication {
```

SpringBootApplication 本质上是由 3 个注解组成，分别是

1. @Configuration
2. @EnableAutoConfiguration
3. @ComponentScan

我们可以直接用这三个注解也可以启动 springboot 应用，只是每次配置三个注解比较繁琐，所以直接用一个复合注解更方便些。

然后仔细观察者三个注解，除了 EnableAutoConfiguration可能稍微陌生一点，其他两个注解使用得都很多

简单分析@Configuration

Configuration 这个注解大家应该有用过，它是 JavaConfig 形式的基于 Spring IOC 容器的配置类使用的一种注解。因为 SpringBoot 本质上就是一个 spring 应用，所以通过这个注解来加载 IOC 容器的配置是很正常的。所以在启动类里面标注了@Configuration，意味着它其实也是一个 IoC容器的配置类。

传统意义上的 spring 应用都是基于 xml 形式来配置 bean的依赖关系。然后通过 spring 容器在启动的时候，把 bean进行初始化并且，如果 bean 之间存在依赖关系，则分析这些已经在 IoC 容器中的 bean 根据依赖关系进行组装。

直到 Java5 中，引入了 Annotations 这个特性，Spring 框架也紧随大流并且推出了基于 Java 代码和 Annotation 元信息的依赖关系绑定描述的方式。也就是 JavaConfig。

从 spring3 开始，spring 就支持了两种 bean 的配置方式，一种是基于 xml 文件方式、另一种就是 JavaConfig

任何一个标注了@Configuration 的 Java 类定义都是一个JavaConfig 配置类。而在这个配置类中，任何标注了

@Bean 的方法，它的返回值都会作为 Bean 定义注册到Spring 的 IOC 容器，方法名默认成为这个 bean 的 id

简单分析 ComponentScan

ComponentScan 这个注解是大家接触得最多的了，相当于 xml 配置文件中的 <context:component -scan>。 它的

主要作用就是扫描指定路径下的标识了需要装配的类，自动装配到 spring 的 Ioc 容器中。标识需要装配的类的 形式主要是： @Component 、@Repository、 @Service、 @Controller 这类的注解标识的类。

ComponentScan 默认会扫描当前 package 下的的所有加了相关注解标识的类到 IoC 容器中；简单分析 EnableAutoConfiguration我们把 EnableAutoConfiguration 放在最后讲的目的并不是说它是一个新的东西，只是他对于 springboot 来说意义重大。

Enable 并不是新鲜玩意

仍然是在 spring3.1 版本中，提供了一系列的@Enable 开头的注解， Enable 主机应该是在 JavaConfig 框架上更进一步的完善，是的用户在使用 spring 相关的框架是，避免配置大量的代码从而降低使用的难度，比如常见的一些Enable 注解：EnableWebMvc， （这个注解引入了 MVC 框架在 Spring 应用中需要用到的所有bean）；

比如说@EnableScheduling，开启计划任务的支持； 找到 EnableAutoConfiguration，我们可以看到每一个涉及

到 Enable 开头的注解，都会带有一个@Import 的注解。

```
@Import(AutoConfigurationImportSelector.class)
public @interface EnableAutoConfiguration {
```

Import 注解

import 注解是什么意思呢？ 联想到 xml 形式下有一个<import resource/> 形式的注解，就明白它的作用了。

import 就是把多个分来的容器配置合并在一个配置中。在JavaConfig 中所表达的意义是一样的。

深入分析 EnableAutoConfiguration

EnableAutoConfiguration 的主要作用其实就是帮助springboot 应用把所有符合条件的@Configuration 配置都加载到当前 SpringBoot 创建并使用的 IoC 容器中。再回到 EnableAutoConfiguration 这个注解中，我们发现

它的 import 是这样

```
@Import(AutoConfigurationImportSelector.class)
public @interface EnableAutoConfiguration {
```

AutoConfigurationImportSelector 是什么?

Enable 注解不仅仅可以像前面演示的案例一样很简单的实现多个 Configuration 的整合，还可以实现一些复杂的场景，比如可以根据上下文来激活不同类型的 bean，

@Import 注解可以配置三种不同的 class

1. 第一种就是前面演示过的，基于普通 bean 或者带有@Configuration 的 bean 进行诸如
2. 实现 ImportSelector 接口进行动态注入
3. 实现 ImportBeanDefinitionRegistrar 接口进行动态注入

@EnableAutoConfiguration 注解的实现原理

了解了 ImportSelector 和 ImportBeanDefinitionRegistrar后，对于 EnableAutoConfiguration的理解就容易一些了

它会通过 import 导入第三方提供的 bean 的配置类:

```
AutoConfigurationImportSelector
@Import(AutoConfigurationImportSelector.class)
```

从名字来看，可以猜到它是基于 ImportSelector 来实现基于动态 bean 的加载功能。之前我们讲过 Springboot

@Enable*注解的工作原理 ImportSelector 接口 selectImports 返回的数组（类的全类名）都会被纳入到spring 容器中。

那么可以猜想到这里的实现原理也一定是一样的，定位到AutoConfigurationImportSelector 这个类中的selectImports 方法本质上来说，其实 EnableAutoConfiguration 会帮助springboot 应用把所有符合@Configuration 配置都加载到当前 SpringBoot 创建的 IoC 容器，而这里面借助了 Spring 框架提供的一个工具类 SpringFactoriesLoader 的支持。以及用到了 Spring 提供的条件注解@Conditional，选择性的针对需要加载的 bean 进行条件过滤

SpringFactoriesLoader

为了给大家补一下基础，我在这里简单分析一下SpringFactoriesLoader 这个工具类的使用。它其实和java 中的 SPI 机制的原理是一样的，不过它比 SPI 更好的点在于不会一次性加载所有的类，而是根据 key 进行加载。

首先， SpringFactoriesLoader 的作用是从classpath/META-INF/spring.factories 文件中，根据 key 来

加载对应的类到 spring IoC 容器中。

深入理解条件过滤

在分析 AutoConfigurationImportSelector 的源码时，会先扫描 spring-autoconfiguration-metadata.properties

文件，最后在扫描 spring.factories 对应的类时，会结合前面的元数据进行过滤，为什么要过滤呢？ 原因是很多

的@Configuration 其实是依托于其他的框架来加载的，如果当前的 classpath 环境下没有相关联的依赖，则意味

着这些类没必要进行加载，所以，通过这种条件过滤可以有效的减少@configuration 类的数量从而降低

SpringBoot 的启动时间。

Conditional 中的其他注解

Conditions	描述
@ConditionalOnBean	在存在某个 bean 的时候
@ConditionalOnMissingBean	不存在某个 bean 的时候
@ConditionalOnClass	当前 classpath 可以找到某个类型的类时
@ConditionalOnMissingClass	当前 classpath 不可以找到某个类型的类时
@ConditionalOnResource	当前 classpath 是否存在某个资源文件
@ConditionalOnProperty	当前 jvm 是否包含某个系统属性为某个值
@ConditionalOnWebApplication	当前 spring context 是否是 web 应用程序

2019年7月4日 21:28

1.基于你对springboot的理解描述一下什么是springboot

springBoot 是一个基于框架的框架,为了帮助使用spring框架的开发者快速高效的构建一个spring框架以及spring体系的应用的解决方案 ,它是基于约定优于配置的最佳实践.

2.约定优于配置指的是什么?

1:maven的目录结构

默认resource文件存放配置文件

默认打包方式为jar

2:spring-boot-starter-web 中默认装载spring mvc 相关依赖 以及内置一个tomcat ,以便快速简单构建一个web应用

3:默认提供application.yml 和 application.properties

4:通过spring.active 制定启用哪个配置文件

5:EnableAutoConfiguration 默认对依赖的starter 进行加载

3.@SpringBootApplication由哪几个注解组成，这几个注解分别表示什么作用

1:@Configuration

任何标注了@Configuration的类都是一个javaConfig类,这个类中标注了@Bean 的方法的返回值都会作为bean注入到ioc容器中,方法名作bean为id.

2:@ConponentScan

默认将标注这个注解的类所在的包下面 带有(@Respository @Service @Component @Controller)注解的类自动注册到ioc容器中

3:@AutoEnableConfiguration

1:通过@import导入带有@configuration 的类

2. 实现 importSelector接口进行动态注入

3. 实现 ImportBeanDefinitionRegister接口进行动态注入

4.springboot自动装配的实现原理

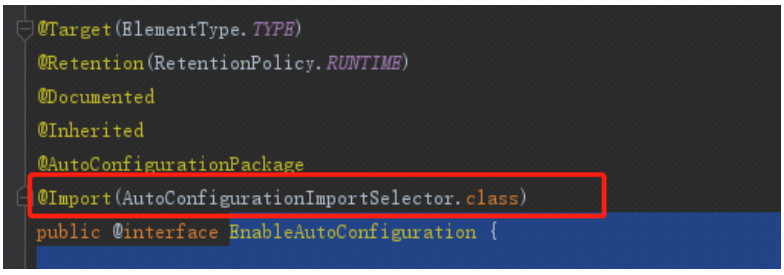
importSelector解析:

1:入口:

```
@SpringBootApplication
//@EnableDefineService
public class EnableDemoMain {

    public static void main(String[] args) {
        ConfigurableApplicationContext ca=SpringApplication.run(EnableDemoMain.class,args);

        System.out.println(ca.getBean(CacheService.class));
        System.out.println(ca.getBean(LoggerService.class));
    }
```



selectImport:

```
@Override
publicString[]selectImports(
   AnnotationMetadataannotationMetadata) {
   if(!isEnabled(annotationMetadata)) {
      returnNO_IMPORTS;
   }
//获取元数据（@springBootApplication(exclude="")中的数据就是在这里获取）
   AutoConfigurationMetadataautoConfigurationMetadata = AutoConfigurationMetadataLoader.loadMetadata(this.beanClassLoader);
//获取自动加载的对象
   AutoConfigurationEntryautoConfigurationEntry = getAutoConfigurationEntry(autoConfigurationMetadata,annotationMetadata);
   returnStringUtils.toStringArray(autoConfigurationEntry.getConfigurations());
}
```

loadMetadata获取源数据:

```
protected static final String PATH = "META-INF/" + "spring-autoconfigure-
metadata.properties";

private AutoConfigurationMetadataLoader() {
}

public static AutoConfigurationMetadata
//回加载类路径下所有META-INF/spring-autoconfigure-metadata.properties 文件中的信息作为元数据
loadMetadata(ClassLoader classLoader) {
   return loadMetadata(classLoader, PATH);
}
```

spring-autoconfigure-metadata.properties示例:

```
org.springframework.boot.autoconfigure.reactor.core.ReactorCoreAutoConfiguration.Configuration=
org.springframework.boot.autoconfigure.data.cassandra.CassandraReactiveDataAutoConfiguration.AutoConfigureAfter=org.springframework.boot.autoconfigure.data.cassandra.CassandraDataAutoConfiguration
org.springframework.boot.autoconfigure.jdbc.JndiDataSourceAutoConfiguration.ConditionalOnClass=javax.sql.DataSource,org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseType
org.springframework.boot.autoconfigure.admin.SpringApplicationAdminJmxAutoConfiguration=
```

getAutoConfigurationEntry 获取加载的对象:

```
protected AutoConfigurationEntry getAutoConfigurationEntry( AutoConfigurationMetadata
autoConfigurationMetadata,
   AnnotationMetadata annotationMetadata) {
   if (!isEnabled(annotationMetadata)) {
      return EMPTY_ENTRY;
```

```
    }
    AnnotationAttributes attributes = getAttributes(annotationMetadata);
    List<String> configurations = getCandidateConfigurations(annotationMetadata,attributes);
//去重
    configurations = removeDuplicates(configurations);
    Set<String> exclusions = getExclusions( annotationMetadata, attributes);
    checkExcludedClasses(configurations, exclusions);
//删除所有不满足条件的
    configurations.removeAll(exclusions);
    configurations = filter(configurations, autoConfigurationMetadata);
    fireAutoConfigurationImportEvents(configurations, exclusions);
    return new AutoConfigurationEntry(configurations, exclusions);
}
```

getCandidateConfigurations:

```
protected List<String> getCandidateConfigurations( AnnotationMetadata metadata,
AnnotationAttributes attributes) {
    List<String> configurations = SpringFactoriesLoader.loadFactoryNames(
            getSpringFactoriesLoaderFactoryClass(),
            getBeanClassLoader());
//获取配置文件下META-INF/spring.factories中对应注解key下面的数据
    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;
}
```

spring.factories示例:

```
#AutoConfigurationImportFilters
org.springframework.boot.autoconfigure.AutoConfigurationImportFilter=\
org.springframework.boot.autoconfigure.condition.OnBeanCondition,\
org.springframework.boot.autoconfigure.condition.OnClassCondition,\
org.springframework.boot.autoconfigure.condition.OnWebApplicationCondition

#AutoConfigure
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
org.springframework.boot.autoconfigure.admin.SpringApplicationAdminJmxAutoConfigurati
on,\
org.springframework.boot.autoconfigure.aop.AopAutoConfiguration,\
org.springframework.boot.autoconfigure.amqp.RabbitAutoConfiguration,\
org.springframework.boot.autoconfigure.batch.BatchAutoConfiguration,\
org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration,\
```

5.spring中的spi机制的原理是什么?

将classpath下面 `META-INF/spring.factories` 中的z配置文件中的类自动加载至ioc容器中

提交地址

https://gper.club/homework/subjects/7e7e7f7ff0g5dgc5g6d