### 1、@SpringBootApplication背后的秘密

@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 {

...

}

虽然定义使用了多个Annotation进行了原信息标注，但实际上重要的只有三个Annotation：

* @Configuration（@SpringBootConfiguration点开查看发现里面还是应用了@Configuration）
* @EnableAutoConfiguration
* @ComponentScan

所以，如果我们使用如下的SpringBoot启动类，整个SpringBoot应用依然可以与之前的启动类功能对等：

@Configuration

@EnableAutoConfiguration

@ComponentScan

public class Application {

public static void main(String[] args) {

SpringApplication.run(Application.class, args);

}

}

但每次都写这三个Annotation显然过于繁琐，所以写一个@SpringBoot-Application这样的一站式符合Annotation显然更方便些。

### 2、@Configuration创世纪

这里的@Configuration就是JavaConfig形式的Spring IoC容器的配置类使用的那个@Configuration，既然SpringBoot应用骨子里就是一个Spring应用，那么，自然也需要加载某个IoC容器的配置，而SpringBoot社区推荐使用基于JavaConfig的配置形式，所以，很明显，这里的启动类标注了@Configuration之后，本身其实也是一个IoC容器的配置类！

很多SpringBoot的代码示例都喜欢在启动类上直接标注@Configuration或者@SpringBootApplication，对于初接触SpringBoot的开发者来说，其实这种做法不便于理解，如果将上面的SpringBoot启动类拆分为两个独立的Java类，整个形势就明朗了：

@Configuration

@EnableAutoConfiguration

@ComponentScan

public class DemoConfiguration {

@Bean

public Controller controller() {

return new Controller();

}

}

public class DemoApplication {

public static void main(String[] args) {

SpringApplication.run(DemoConfiguration.class, args);

}

}

启动类其实就是一个标准的独立（Standalone）类型Java程序的main函数启动类，没有什么特殊的。而@Configuration标注的DemoConfiguration定义其实也是一个普通的JavaConfig形式的IoC容器配置类，全是Spring框架里的概念。

Spring容器中为了简化XML配置，允许使用JavaConfig注册一个Bean。就是使用的是@Configuration，每个拥有注解@Bean的函数的返回值，都将会在Spring启动时候注册到容器中，可以使用自动装配，如下一个JavaConfig的注册Bean：

@Configuration

public class Configs {

@Value("classpath:data.json")

protected File configFile;

@Bean

public PersonCfg readServerConfig() throws IOException {

return new ObjectMapper().readValue(configFile, PersonCfg.class);

}

}

### 3、@EnableAutoConfiguration的功效

借助@Import的支持，收集和注册特定场景相关的bean的定义：

* @Enable Scheduling是通过@Import将Spring调度框架相关的bean定义都加载到IoC容器。
* @Enable M Bean Export是通过@Import将JMX相关的bean定义加载到IoC容器。

@EnableAutoConfiguration也是借助@Import的帮助，将所有符合自动配置条件的bean定义加载到IoC容器！

@Target(ElementType.TYPE)

@Retention(RetentionPolicy.RUNTIME)

@Documented

@Inherited

@AutoConfigurationPackage

@Import(EnableAutoConfigurationImportSelector.class)

public @interface EnableAutoConfiguration {

...

}

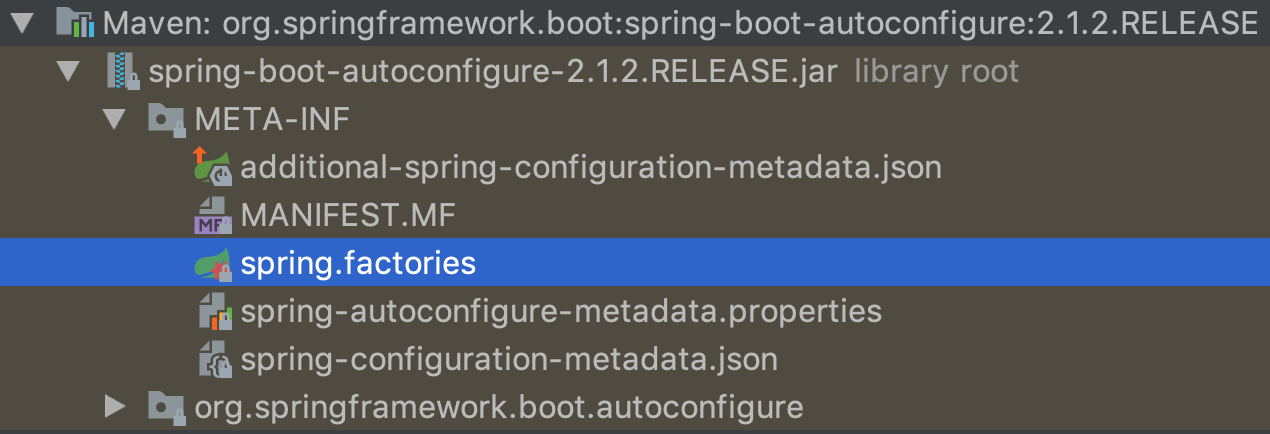
其中最关键的要素@Import(EnableAutoConfigurationImportSelector.class)，借助EnableAutoConfigurationImportSelector，@EnableAutoConfiguration可以帮助SpringBoot应用将所有符合条件的@Configuration配置都加载到当前SpringBoot创建并使用的IoC容器。

借助于Spring框架原有的一个工具类：SpringFactoriesLoader的支持，@EnableAutoConfiguration可以“智能”地自动配置功效才得以大功告成！

### 4、自动配置幕后英雄：SpringFactoriesLoader详解

SpringFactoriesLoader属于Spring框架私有的一种扩展方案，其主要功能就是从指定的配置文件META-INF/spring.factories加载配置。

配合@EnableAutoConfiguration使用的话，它更多是提供一种配置查找的功能支持，即根据@EnableAutoConfiguration的完整类名org.springframewoek.boot.EnableAutoConfiguration作为查找的key，获取对应的一组@Configuration类：



# PropertySource Loaders

org.springframework.boot.env.PropertySourceLoader=\

org.springframework.boot.env.PropertiesPropertySourceLoader,\

org.springframework.boot.env.YamlPropertySourceLoader

# Run Listeners

org.springframework.boot.SpringApplicationRunListener=\

org.springframework.boot.context.event.EventPublishingRunListener

# Application Context Initializers

org.springframework.context.ApplicationContextInitializer=\

org.springframework.boot.context.ConfigurationWarningsApplicationContextInitializer,\

org.springframework.boot.context.ContextIdApplicationContextInitializer,\

org.springframework.boot.context.config.DelegatingApplicationContextInitializer,\

org.springframework.boot.context.embedded.ServerPortInfoApplicationContextInitializer

# Application Listeners

org.springframework.context.ApplicationListener=\

org.springframework.boot.ClearCachesApplicationListener,\

org.springframework.boot.builder.ParentContextCloserApplicationListener,\

org.springframework.boot.context.FileEncodingApplicationListener,\

org.springframework.boot.context.config.AnsiOutputApplicationListener,\

org.springframework.boot.context.config.ConfigFileApplicationListener,\

org.springframework.boot.context.config.DelegatingApplicationListener,\

org.springframework.boot.liquibase.LiquibaseServiceLocatorApplicationListener,\

org.springframework.boot.logging.ClasspathLoggingApplicationListener,\

org.springframework.boot.logging.LoggingApplicationListener

# Environment Post Processors

org.springframework.boot.env.EnvironmentPostProcessor=\

org.springframework.boot.cloud.CloudFoundryVcapEnvironmentPostProcessor,\

org.springframework.boot.env.SpringApplicationJsonEnvironmentPostProcessor

# Failure Analyzers

org.springframework.boot.diagnostics.FailureAnalyzer=\

org.springframework.boot.diagnostics.analyzer.BeanCurrentlyInCreationFailureAnalyzer,\

org.springframework.boot.diagnostics.analyzer.BeanNotOfRequiredTypeFailureAnalyzer,\

org.springframework.boot.diagnostics.analyzer.BindFailureAnalyzer,\

org.springframework.boot.diagnostics.analyzer.ConnectorStartFailureAnalyzer,\

org.springframework.boot.diagnostics.analyzer.NoUniqueBeanDefinitionFailureAnalyzer,\

org.springframework.boot.diagnostics.analyzer.PortInUseFailureAnalyzer,\

org.springframework.boot.diagnostics.analyzer.ValidationExceptionFailureAnalyzer

# FailureAnalysisReporters

org.springframework.boot.diagnostics.FailureAnalysisReporter=\

org.springframework.boot.diagnostics.LoggingFailureAnalysisReporter

以上是从Springboot的autoconfiguration依赖包中的META-INF/spring.factories配置文件中摘录的一段内容。以@EnableAutoConfiguration自动配置的魔法其实就变成了：

* 从classpath中搜寻META-INF/spring.factories配置文件
* 将org.springframework.boot.autoconfiguration.EnableAutoConfiguration对应的配置项通过反射（Java Reflection）实例化对应的标注了@Configuration的JavaConfig形式的IoC容器配置类
* 然后汇总为一个并加载到IoC容器。

### 5、可有可无的@Configuration

@ComponentScan的功能其实就是自动扫描并加载符合条件的组件或bean定义，最终将这些bean定义加载到容器中。加载bean定义到Spring的IoC容器，我们可以手工单个注册，不一定非要通过批量的自动扫描完成，所以说@ComponentScan是可有可无的。

### 6、深入探索SpringApplication执行流程

SpringApplication的run方法的实现是我们本次旅程的主要线路，该方法的主要流程大体可以归纳如下：

1. 如果我们使用的是SpringApplication的静态run方法，那么这个方法里面首先需要创建一个SpringApplication对象实例，然后调用这个创建好的SpringApplication的实例run方法。在SpringApplication实例初始化的时候，它会提前做几件事：
   1. 根据classpath里面是否存在某个特征类来决定是否应该创建一个为Web应用使用的ApplicationContext类型，还是应该创建一个标准Standalone应用使用的ApplicationContext类型。
   2. 使用SpringFactoriesLoader在应用的classpath中查找并加载所有可用的ApplicationContextInitializer。
   3. 使用SpringFactoriesLoader在应用的classpath中查找并加载所有可用的ApplicationListener。
   4. 推断并设置main方法的定义类。
2. SpringApplication实例初始化完成并且完成设置后，就开始执行run方法的逻辑了，方法执行伊始，首先遍历执行所有通过SpringFactoriesLoader可以查找到并加载的SpringApplicationRunListener，调用它们的started()方法，告诉这些SpringApplicationRunListener，“嘿，SpringBoot应用要开始执行咯！”。
3. 创建并配置当前SpringBoot应用将要使用的Environment（包括配置要使的PropertySource以及Profile）。
4. 便利调用所有SpringApplicationRunListener的environmentPrepared()的方法，告诉它们：“当前SpringBoot应用使用的Environment准备好了！”
5. 如果SpringApplication的showBanner属性被设置为true，则打印banner（SpringBoot 1.3x版本，这里应该是基于Banner。Mode决定banner的打印行为）。这一步的逻辑可以不用关心。
6. 根据用户是否明确设置了applicationContextClass类型以及初始化阶段的推断结果，决定该为当前SpringBoot应用创建什么类型的ApplicationContext并创建完成，然后根据条件决定是否添加ShutdownHook，决定是否使用自定义的BeanNameGenerator，决定是否使用自定义的ResourceLoader，当然，最重要的，将之前准备好的Environment设置给创建好的ApplicationContext使用。
7. ApplicationContext创建好之后，SpringApplication会再次借助Spring-FactoriesLoader，查找并加载classpath中所有可用的ApplicationContextInitializer的initialize（applicationContext）方法来对已经创建好的ApplicationContext进行进一步处理。
8. 遍历调用所有SpringApplicationRunListener的contextPrepared()方法，通知它们：“SpringBoot应用使用的ApplicationContext准备好啦！”
9. 最核心的一步，将之前通过@EnableAutoConfiguration获取的所有配置以及其他形式的IoC容器配置加载到已经准备完毕的ApplicationContext。
10. 遍历调用所有SpringApplicationRunListener的contextLoaded()方法，告知所有SpringApplicationRunListener，ApplicationContext“装填完毕”！
11. 调用ApplicationContext的refresh()方法，完成IoC容器可用的最后一道工序。
12. 查找当前ApplicationContext中是否注册有CommondLineRunner，如果有，则遍历执行它们。
13. 正常情况下，遍历执行SpringApplicationRunListener的finished()方法，告知它们“搞定！”。（如果整个过程出现异常，则依然调用所有SpringApplicationRunListener的finished()方法，只不过这种情况下会将异常信息一并传入处理）。

至此，一个完整的SpringBoot应用启动完毕！

整个过程看起来冗长无比，其实很多都是一些事情通知的扩展点，如果我们将这些逻辑暂时忽略，那么，整个SpringBoot应用启动的逻辑就可压缩到及其精简的几步。

