**SpringBoot**

**目录**

[概述 1](#_Toc3063)

[微服务架构 2](#_Toc22383)

[依赖管理 3](#_Toc866)

[父项目管理依赖 3](#_Toc19497)

[场景与场景启动器（starter） 4](#_Toc25823)

[SpringBoot自动配置 5](#_Toc30412)

[默认的包结构 5](#_Toc19789)

[自动配置类 6](#_Toc928)

[配置绑定 6](#_Toc22287)

[实现配置绑定的两种方法 6](#_Toc19902)

[按需开启场景的自动配置 7](#_Toc23090)

[场景组件都有默认的配置 8](#_Toc22783)

[了解的一些注解 8](#_Toc20232)

[自动配置原理 10](#_Toc14545)

[SpringBoot配置文件的加载位置 22](#_Toc28220)

[多环境下切换SpringBoot的配置文件 22](#_Toc12884)

[Themeleaf模板引擎 23](#_Toc17136)

[SpringSecurity 24](#_Toc7330)

[Shiro 25](#_Toc18022)

[shiro架构 25](#_Toc22590)

[shiro组件 29](#_Toc6713)

[Swagger 29](#_Toc16114)

[RPC 31](#_Toc23870)

[zookeeper和dubbo 31](#_Toc14245)

**概述**

SpringBoot是基于Spring开发出来的框架，简化了Spring中繁琐的配置。Spirng Boot 本身并不提供 Spring 框架的核心特性以及扩展功能，只是用于快速、敏捷地开发新一代基于 Spring 框架的应用程序。也就是说，它并不是用来替代 Spring 的解决方案，而是和 Spring 框架紧密结合用于提升 Spring 开发效率的工具。Spring Boot 以约定大于配置的核心思想，默认帮我们进行了很多设置，多数 Spring Boot 应用只需要很少的 Spring 配置。同时它集成了大量常用的第三方库配置（例如 Redis、MongoDB、Jpa、RabbitMQ、Quartz 等等），Spring Boot 应用中这些第三方库几乎可以零配置的开箱即用。简单来说就是SpringBoot其实不是什么新的框架，它默认配置了很多框架的使用方式。

总结：

1. SpringBoot是用来简化Spring的开发流程，快速整合第三方框架，搭建微服务的一个框架。
2. SpringBoot是整合Spring技术栈的一站式框架
3. SpringBoot是简化Spring技术栈的快速开发脚手架

SpringBoot框架帮助文档：

<https://docs.spring.io/spring-boot/docs/2.0.3.RELEASE/reference/htmlsingle/#using-boot-starter>

SpringBoot语雀笔记：

https://www.yuque.com/atguigu/springboot/rmxq85

**微服务架构**

我们知道一个项目会包含很多功能模块，以电商项目为例，它可能会包含的登录模块，注册模块，购物车模块，我的订单模块等等。可想而知，采用传统架构的项目，这么多的功能模块都是整合在一个项目里（all in one），维护起来是非常不便的，假如其中一个模块需要更新升级，则整合项目必须先下线，待模块更新完毕，再重新上线。但是，一旦公司规模做大，有了大量的用户，频繁的下线项目，会对用户造成不好的体验，因此微服务架构应运而生。

微服务架构的思想是：将一个应用程序中包含的功能模块都拆分出来，形成一个个微小的功能模块，每一个微小的功能模块称为一个微服务。这样一个微小的功能模块就是一个独立的项目，可以进行独立部署（独立部署到不同的服务器）和升级，也就是说当其中一个模块需要更新时，只下线它即可，其它部署在其它服务器上的模块（微服务）照常提供服务。最重要的是这些微服务之间可以按需求相互组合，提供服务。

我们将一个大型的项目（应用），拆分成一个个微小的模块并部署到不同的服务器上时，就会涉及到分布式应用的问题，SpringCloud为我们提供了分布式的解决方案。

**依赖管理**

**父项目管理依赖**

Maven构建的springboot项目的pom文件中，必定要继承一个父项目，如下

所示:

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>springboot的版本号</version>

</parent>

spring-boot-starter-parent又继承了一个父项目

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-dependencies</artifactId>

<version>springboot的版本号</version>

</parent>

注：在spring-boot-dependencies中，几乎声明了所有开发中常用的依赖的

版本号（自动版本仲裁机制）。因此，父项目（spring-boot-dependencies）

用来管理子项目（我们创建的springboot项目）中用到的依赖。自动版本仲

裁机制是指：子项目中用到的依赖的版本号，都由父项目管理，子项目无需再

指定其版本号。注意：如果在子项目中引入的是未被父项目管理的依赖，则需

要指定其版本。

**场景与场景启动器（starter）**

SpringBoot支持常见的开发场景，比如web开发场景。在2.6.0版本中共支

持131个场景，不同的版本支持场景数量可能不同。在这个链接中可以查看

SpringBoot到底支持哪些场景，以及查看场景的启动器，链接如下：

启动器（starter）是一组依赖描述符的集合。在pom文件中导入某个场景

的启动器，就会自动导入该场景需要的所有依赖。官方提供的启动器以

spring-boot-starter-\*的形式命名，\*号代表某个场景，比如spring-boot-starter-web，

这是web开发场景的启动器。第三方提供的场景启动器以\*-spring-boot-starter

命名。spring-boot-starter是所有场景启动器最底层的依赖，如下所示：

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter</artifactId>

<version>2.6.0-M1</version>

<scope>compile</scope>

</dependency>

总结：SpringBoot支持常见的开发场景，只要导入了该场景的启动器，就会自

动导入该场景需要的所有依赖，并且SpringBoot还会自动将该场景的需要用

到的组件注入到容器中。这也正是SpringBoot为什么能简化Spring中繁琐配

置的原因。

**SpringBoot自动配置**

**默认的包结构**

SpringBoot默认的包结构如下:



之前，我们在编写controller类时，需要开启组件扫描后，指定包下的组件

才会生效。而SpringBoot应用在启动时，会默认扫描主程序类所在包及其子

包中的所有组件，并将这些组件注入到容器中。因此，无需再配置组件扫

描。但是，如果你声明的组件在主程序所在包的外面，就需要人为指定扫描

哪些包下的组件。可以使用以下方式，改变默认扫描的路径。

@SpringBootApplication(scanBasePackages="扫描的路径")

**自动配置类**

前面提到过Springboot支持一些常见开发场景。这些场景都有各自的启动

器，只有导入启动器后，才会对场景进行自动配置。每个场景除了有各自的启

动器外，还有各自的自动配置类，自动配置类一般以xxxAutoConfiguration命

名，在场景的自动配置类中定义了该场景需要用到的组件。所以对场景进行自

动配置也就是将该场景需要用到的组件导入到容器中。

**配置绑定**

配置绑定是指：使用java读取properties/yml配置文件中指定“前缀”下的属

性，并将这些属性与某个类中的属性，进行一一绑定。这样的类，一般以

xxxProperties命名。这里需要注意的是xxxProperties类中的属性名一定要和

springboot配置文件中指定“前缀”下的属性名相同，才能绑定成功。

简单讲，就是将SpringBoot配置文件中指定“前缀”下的属性绑定到一个

javabean（类）上，以供随时使用，这样的javabean建议以xxxProperties命

名。

**实现配置绑定的两种方法**

方法一：使用@component + @ConfigurationProperties

采用此方法必须保证此javabean已作为组件被注入到容器，才能进行配置

绑定。

方法二：使用@EnableConfigurationProperties + @ConfigurationProperties

例如：在配置类上标注@EnableConfigurationProperties(Person.class) //为

Person类开启自动配置绑定，并将这个Person类作为组件注入到容器。采

用此方法在Person类上必须使用@ConfigurationProperties标注，简单理解

就是这个Persion类必须先保证自己支持配置绑定，才能再为其开启配置绑

定。

**按需开启场景的自动配置**

SpringBoot支持的所有场景的自动配置类都被写死在 “META-INF/spring.factories”

文件中（该文件位于spring-boot-autoconfigure jar中）。SpringBoot应用

在启动时，虽然会默认加载spring.factories文件中所有场景的自动配置

类，但会按照条件装配规则，按需对场景进行自动配置。

也就是说，虽然Springboot应用在启动时，会加载所有场景的自动配

置类，但并不是所有的自动配置类都会生效。我们知道自动配置类上频繁

的使用了@Conditional的衍生注解，这类注解的作用就是指定条件装配规则。

所以说，只有符合了条件装配规则的自动配置类才会生效，生效就意味着该自

动配置类中定义的组件才会被注入到容器，这样也就完成了对应场景的自动配

置（场景的自动配置类中定义了该场景需要用到的组件，对场景进行自动配置

也就是将该场景需要用到的组件注入到容器）。

只有已导入了启动器（starter）的场景，此场景的自动配置类才会符合

条件装配规则，才会对其进行自动配置。

注："META-INF/spring.factories"不是唯一的，可以在多个jar中存在。

**场景组件都有默认的配置**

前面提到过场景的自动配置类中，定义了该场景需要用到的组件，不仅如此

这些的组件都有着默认的配置。自动配置类中的组件的默认配置都是映射到某

个类上（准确的说应该是类的属性上），这样的类一般以xxxProperties命名。

最重要的是xxxProperties类中会与springboot配置文件中指定”前缀”下的属性

进行绑定。

修改自动配置类中组件的默认配置：自动配置类中定义了场景需要用到的组

件 -> 组件有默认配置，会从xxxProperties类中获取默认配置的值 ->

xxxProperties类与SpringBoot配置文件中指定”前缀”下的属性进行绑定。这相

当于给开发者提供了一个修改组件默认配置的入口 -> 开发者想要组件使用

自定义的配置，只需修改SpringBoot配置文件中指定“前缀”下的属性值，即可

实现用自定义的配置取代组件默认的配置。

定制化组件

自动配置类中定义的组件有着默认的配置，我们可以修改组件默认的配置，

也就是定制化组件，对组件的配置实现自定义。

定制化组件常用的两种方式：

一、修改配置文件中与xxxProproties绑定的前缀下的属性。

二、在自配置类中通过@Bean注解定义的组件上，会频繁使用到

@ConditionalOnMissingBean注解，表示如果容器中没有这个组件，

springboot就会帮我们注册一个。所以我们可以通过自定义配置类，往容器

中添加组件来替换默认的组件。

**了解的一些注解**

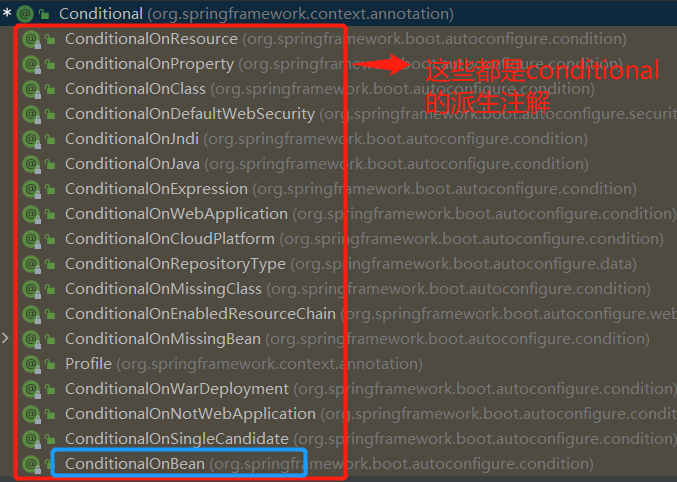
- @Conditional

作用：条件装配，即满足@Conditional注解指定的条件后，才能进行组件

注入。该注解标注在配置类上时，该配置类中使用@Bean注解定义的所有

的组件，只有在满足指定条件后，才会被注入到容器中。另外该注解存在很

多派生注解，如图所示：



注：@Conditional注解及其衍生注解，可以用来定义条件装配规则。在场

景的自动配置类上就大量的使用了@Conditional注解及其衍生注解，通过

判断是否符合这些注解定义的条件装配规则，来决定是否对场景进行自动配

置。

- @import

作用：向容器中快速导入指定类型的组件，以该方式导入的组件名为全类名。

@import注解的另外一种用法: @import(xxxImportSelector.class)。

xxxImportSelector类实现了ImportSelector接口，返回指定组件的全类名，

返回值是一个字符串数组。@import + xxxImportSelector.class 可以实现选择

导入一些指定的组件。

- @ImportResource

SpringBoot虽然已经摈弃了原生的spring配置文件，但任然提供了

@ImportResource注解，使用该注解可以导入原生的spring配置文件，例如：

@ImportResource("classpath:beans.xml")

- @ConfigurationProperties

作用：实现配置绑定，也就是读取properties/yml配置文件中自定义的属性值，

将其封装到一个javabean（实体类）中，前提是这个javabean已作为组件被

注入到容器。所以该注解需要配合使用@component注解才能实现配置绑定。

一句话：先注册为容器中的组件，在进行配置绑定。

**自动配置原理**

- @SpringBootApplication //该注解标记的类，是一个主配置类/主程序类，并

表示这是一个SpringBoot应用。

该注解是由以下三个注解合成而来，分别是：

@SpringBootConfiguration //SpringBoot配置

@EnableAutoConfiguration //开启自动配置

@ComponentScan。 //组件扫描

- @SpringBootConfiguration //该注解标注在某个类上，表示这是类是主配置

类。该注解中合成了@Configuration注解，然后@Configuration注解中又合

成了@Component注解。看到这里，可以得出主配置类本身也是容器中的一

个组件，它的作用是用来启动SpringBoot应用。

- @EnableAutoConfiguration //该注解表示开启自动配置功能，SpringBoot之

所以能够简化Spring中的繁琐的配置，就是因为SpringBoot为常见的开发场

景提供了自动配置功能，该注解由以下两个注解合成而来，分别是：

@AutoConfigurationPackage //通过该注解，springboot主程序类所在包及子

包中的组件，都会被扫描并注入到容器中。

@Import(AutoConfigurationImportSelector.class) //通过该注解的，可以选择将

哪些场景的自动配置类，导入容器。我们知道配置类本身就是一个组件。

**注：这两个注解就是实现springboot自动配置的核心注解。**

**- @AutoConfigurationPackage** //该注解表示自动配置包，该注解中又合成了

@Import(AutoConfigurationPackages.Registrar.class)。

AutoConfigurationPackages是一个抽象类

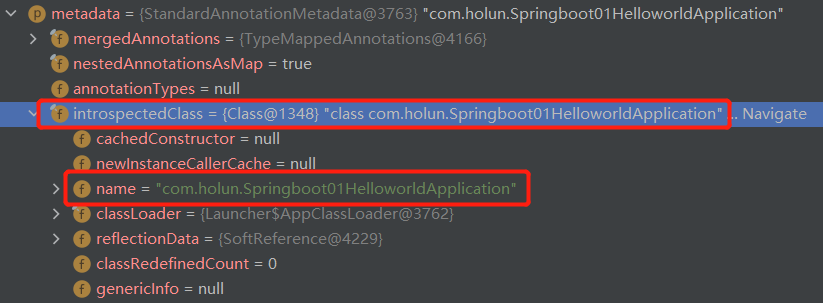
Registrar是抽象类AutoConfigurationPackages中的静态内部类（注册者）

所以@Import(AutoConfigurationPackages.Registrar.class)就是将Registrar这个静态内部类作为组件导入到容器中。（注意：因为Registrar是AutoConfigurationPackages的内部类，将Registrar类作为组件导入容器，实际导入的是AutoConfigurationPackages类，所以在容器中只有名为AutoConfigurationPackages的组件，并没有名为Registrar的组件）。那为什么要将这么一个奇怪的类作组件导入容器呢？那肯定是因为这个类提供给了一个很重要的功能。查看一下这个静态内部类的源码：

static class Registrar implements ImportBeanDefinitionRegistrar, DeterminableImports {  
 @Override  
 public void registerBeanDefinitions(AnnotationMetadata metadata, BeanDefinitionRegistry registry) {  
 *register*(registry, new PackageImports(metadata).getPackageNames().toArray(new String[0]));  
 }  
  
 @Override  
 public Set<Object> determineImports(AnnotationMetadata metadata) {  
 return Collections.*singleton*(new PackageImports(metadata));  
 }

分析：

Registrar类中提供了一个registerBeanDefinitions方法，该方法中有一个形参：AnnotationMetadata metadata，metadata是@AutoConfigurationPackage注解的元数据（元信息），我们查看一下@AutoConfigurationPackage注解的元信息包含哪些内容：

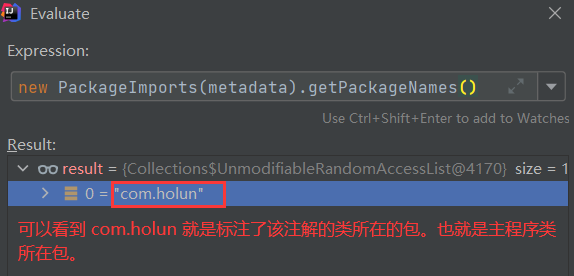


从元信息中不仅可以看出@AutoConfigurationPackag注解的一些属性值，最重要的是可以通过introspectedClass（被标注的类）看出@AutoConfigurationPackage注解被标注在哪个类上，从图中可以看出该注解被标注在主程序类上（com.holun.Springboot01HelloworldApplication就是主程序类）。

在registerBeanDefinitions方法中又调用register方法，register方法并不是Registrar类中的方法，而是AutoConfigurationPackages抽象类中的方法。register方法作用是：将指定包及其子包中的组件注册到容器中。所以在调用register方法时，需要提供这么一个参数：

new PackageImports(metadata).getPackageNames().toArray(new String[0])

这条语句的作用是：通过@AutoConfigurationPackag注解的元信息，获取被该注解标注的类所在的包的名字。因为该注解可能会被标注在多个类上，不一定是主程序类，并且这些类又可能存在不同的包中，所以getPackageNames()返回的是一个List<String>集合，获取的包名都被存储到这个集合中。我们可以看一下标注了@AutoConfigurationPackag注解的类所在的包，如图：



然后在通过toArray方法，将这个List集合转化成字符串数组，这样字符串数组中就包含了所有的包名，最后将这个字符串数组作为实参，传递给register方法的形参（String... packageNames)，register方法的中的形参packageNames是一个字符串类型的可变长参数，可以获取字符串数组中所有的包名，这样register方法就可以将指定包下的组件注入到容器了。

register方法如下所示：

public static void register(BeanDefinitionRegistry registry, String... packageNames) {  
 if (registry.containsBeanDefinition(*BEAN*)) {  
 BasePackagesBeanDefinition beanDefinition = (BasePackagesBeanDefinition) registry.getBeanDefinition(*BEAN*);  
 beanDefinition.addBasePackages(packageNames);  
 }  
 else {  
 registry.registerBeanDefinition(*BEAN*, new BasePackagesBeanDefinition(packageNames));  
 }  
}

总结：

绕了一圈子，其实静态内部类Registrar的作用就是将主程序类/主配置类所在的包及其子包中的组件，全部注入到容器中。

所以先通过@Import(AutoConfigurationPackages.Registrar.class)注解，将Registrar类作为组件导入到容器（实际导入的是AutoConfigurationPackages类，因为Registrar类是该类的内部类，包含在该类的里面），然后这个组件会将主程序类所在的包及其子包中的组件，全部注入到容器中。

看到这里就应该理解了为什么SpringBoot应用在启动时，会默认扫描主程序类所在包及其子包中的所有组件，并将这些组件注入到容器。

**- @Import(AutoConfigurationImportSelector.class)** //通过该注解，可以选择将哪些场景的自动配置类导入到容器。我们知道场景的自动配置类定义了该场景需要用到的组件，所以准确说，该注解就是将选定的场景需要用到的组件，批量导入到容器中。

AutoConfigurationImportSelector类的作用就是为我们找到需要被导入容器的自动配置类。

分析AutoConfigurationImportSelector类中几个中重要方法

selectImports方法，源码如下：

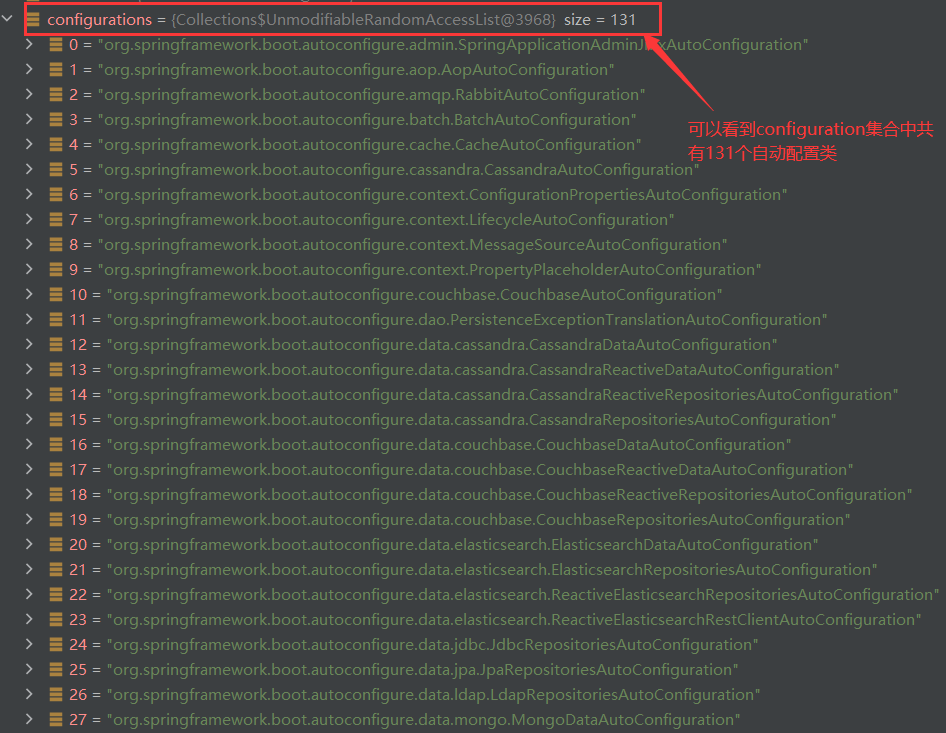
@Override  
public String[] selectImports(AnnotationMetadata annotationMetadata) {  
 if (!isEnabled(annotationMetadata)) {  
 return *NO\_IMPORTS*;  
 }  
 AutoConfigurationEntry autoConfigurationEntry = getAutoConfigurationEntry(annotationMetadata);  
 return StringUtils.*toStringArray*(autoConfigurationEntry.getConfigurations());  
}

该方法的作用是返回需要被导入容器的自动配置类的类名，所以该方法的返回值是一个字符串数组，存于存放自动配置类的类名。在该方法中通过调用getAutoConfigurationEntry方法来获取自动配置条目/自动配置集合。

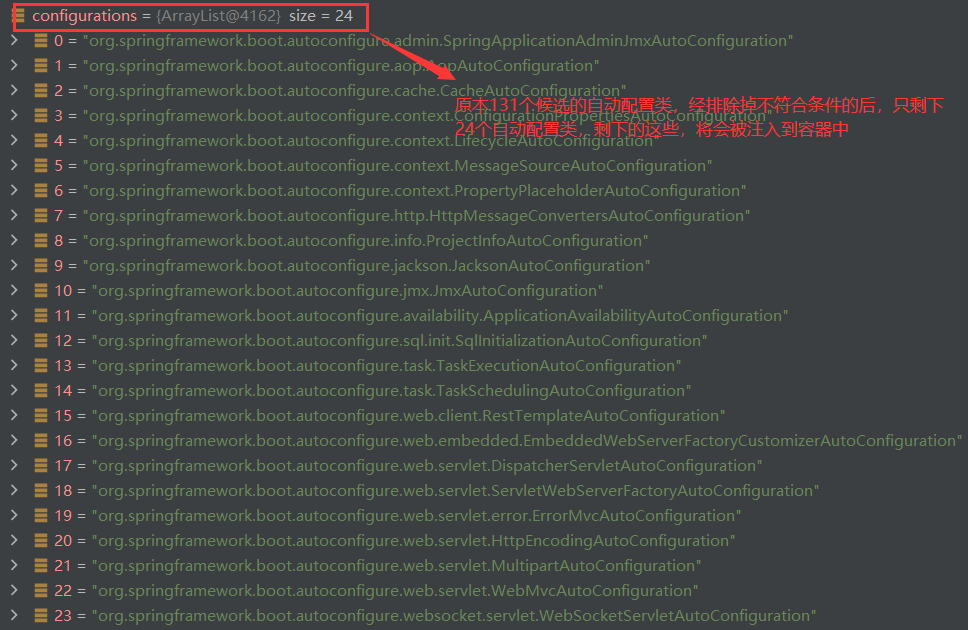
getAutoConfigurationEntry方法，源码如下：

protected AutoConfigurationEntry getAutoConfigurationEntry(AnnotationMetadata annotationMetadata) {  
 if (!isEnabled(annotationMetadata)) {  
 return *EMPTY\_ENTRY*;  
 }  
 AnnotationAttributes attributes = getAttributes(annotationMetadata);  
 List<String> configurations = getCandidateConfigurations(annotationMetadata, attributes);  
 configurations = removeDuplicates(configurations);  
 Set<String> exclusions = getExclusions(annotationMetadata, attributes);  
 checkExcludedClasses(configurations, exclusions);  
 configurations.removeAll(exclusions);  
 configurations = getConfigurationClassFilter().filter(configurations);  
 fireAutoConfigurationImportEvents(configurations, exclusions);  
 return new AutoConfigurationEntry(configurations, exclusions);  
}

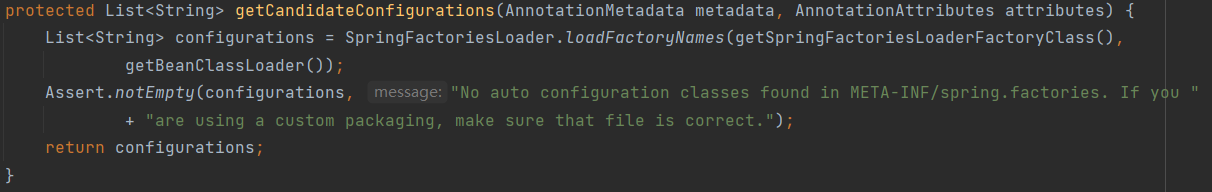
在该方法中调用了getCandidateConfigurations方法来获取候选的自动配置类的类名，这些类名被存放到 configurations中，configurations是List<String>集合。我们可以看些以下configurations中存放了哪些候选自动配置类。注：由于configurations集合中存放的候选自动配合类过多，这里只截取了一部分。如图所示：



该方法在返回自动配置条目（AutoConfigurationEntry）前，又调用了fireAutoConfigurationImportEvents方法，从候选的自动配置类中排除掉不符合条件的自定配置类。此时，configurations集合中剩下的自动配置类，就是要导入到容器中的自动配置类。我们可以看一下，需要被导入容器的自动配置类有哪些，如图：



getCandidateConfigurations方法，源码如下：



在该方法中，实际是通过调用SpringFactoriesLoader类中的静态方法loadFactoryNames来获取候选自动配置类的类名。SpringFactoriesLoader（Spring工厂加载器），是Spring框提供的一种加载方式，SpringFactoriesLoader用来加载 ”META-INF/spring.factories” 中的内容。

loadFactoryNames方法，源码如下：

public static List<String> loadFactoryNames(Class<?> factoryType, @Nullable ClassLoader classLoader) {  
 ClassLoader classLoaderToUse = classLoader;  
 if (classLoaderToUse == null) {  
 classLoaderToUse = SpringFactoriesLoader.class.getClassLoader();  
 }  
 String factoryTypeName = factoryType.getName();  
 return *loadSpringFactories*(classLoaderToUse).getOrDefault(factoryTypeName, Collections.*emptyList*());  
}

在该方法中先调用loadSpringFactories方法，返回一个Map<String, List<String>>，再获取该Map中的的value（这里的value就是自动配置类的类名），并将其存储的List<String>集合中返回。

loadSpringFactories方法，源码如下：

private static Map<String, List<String>> loadSpringFactories(ClassLoader classLoader) {  
 Map<String, List<String>> result = *cache*.get(classLoader);  
 if (result != null) {  
 return result;  
 }  
  
 result = new HashMap<>();  
 try {  
 Enumeration<URL> urls = classLoader.getResources(*FACTORIES\_RESOURCE\_LOCATION*);  
 while (urls.hasMoreElements()) {  
 URL url = urls.nextElement();  
 UrlResource resource = new UrlResource(url);  
 Properties properties = PropertiesLoaderUtils.*loadProperties*(resource);  
 for (Map.Entry<?, ?> entry : properties.entrySet()) {  
 String factoryTypeName = ((String) entry.getKey()).trim();  
 String[] factoryImplementationNames =  
 StringUtils.*commaDelimitedListToStringArray*((String) entry.getValue());  
 for (String factoryImplementationName : factoryImplementationNames) {  
 result.computeIfAbsent(factoryTypeName, key -> new ArrayList<>())  
 .add(factoryImplementationName.trim());  
 }  
 }  
 }  
  
 // Replace all lists with unmodifiable lists containing unique elements  
 result.replaceAll((factoryType, implementations) -> implementations.stream().distinct()  
 .collect(Collectors.*collectingAndThen*(Collectors.*toList*(), Collections::*unmodifiableList*)));  
 *cache*.put(classLoader, result);  
 }  
 catch (IOException ex) {  
 throw new IllegalArgumentException("Unable to load factories from location [" +  
 *FACTORIES\_RESOURCE\_LOCATION* + "]", ex);  
 }  
 return result;  
}

该方法用于将 ”META-INF/spring.factories” 文件中的key-value对加载到Map<String, List<String>>中，并把这个Map返回给loadFactoryNames方法.

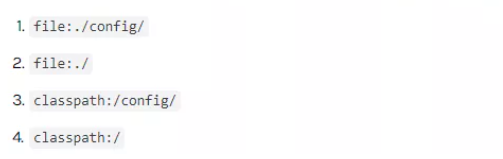
- @ComponentScan



//使用@ComponentScan注解，定义过滤规则，排除不需要导入到容器的组件。

**SpringBoot配置文件的加载位置**

SpringBoot中的配置文件（application.properties/application.yaml），可以放到以下4个位置：



1. 放到项目路径下的config文件夹中（项目名/config）
2. 直接放到项目路径下
3. 放到类路径下的config文件夹中（resources/config）
4. 直接放到类路径下

注：放在这四个路径下的SpringBoot配置文件涉及优先级的问题。优先级：4<3<2<1。SpringBoot应用在启动时，会按优先级扫描加载这4个路径下的配置文件作为默认配置文件，高优先级的配置文件会覆盖优先级低的配置文件。SpringBoot会按优先级加载这四个位置下的配置文件，并且各配置文件之间会互补配置，如果存在配置相同的情况，优先高的配置文件会覆盖低的。还有一点需要注意的是，在相同目录下，.properties结尾的配置文件优先级高于.yaml结尾的配置文件。

**多环境下切换SpringBoot的配置文件**

我们在实际工作中，会涉及到多个环境，比如开发环境，测试环境，生产环境。每个环境都会有自己一套配置，因此一个SpringBoot应用会包含多个配置文件。我们在创建主配置文件时，文件名可以是 application-{profile}.properties/yml , 用来指定多个环境下的配置。比如：

application-test.properties/yml 代表测试环境配置

application-dev.properties/yal 代表开发环境配置

application-prod.properties/yal 代表生产/发布环境配置

但是Springboot并不会直接使用这些配置文件，它默认使用名字为application.properties/yml的配置文件作为主配置文件。我们要使用的话，必须通过配置来激活不同环境下的配置。例如

spring.profiles.active=dev //将application-dev.properties/yml 作为主配置文件。

注1：如果yaml和properties同时都配置了端口，并且没有激活其他环境下的配置， 默认会使用以.properties结尾的配置文件作为主配置文件。

注2：SpringBoot约定多环境下可以有多个配置文件，但一定要有一个名为application.properties/yml的配置文件。

注3：除了在主配置文件中使用spring.profiles.active来激活指定环境外。还可以通过命令行的形式来指定使用哪个环境下的配置，如下：

java -jar xxx.jar --spring.profiles.active=prod

拓展：运维小技巧

加载指定位置配置文件

我们还可以通过spring.config.location来改变默认的配置文件位置

项目打包好以后，在启动项目时，我们可以使用命令行参数的形式，来指定配置文件的新位置；这种情况，一般是后期运维做的多，相同配置，外部指定的配置文件优先级最高。例如：

java -jar spring-boot-config.jar --spring.config.location=F:/application.properties

**Themeleaf模板引擎**

- Thymeleaf 是 Web 和独立环境的现代服务器端 Java 模板引擎，能够处

理HTML，XML，JavaScript，CSS 甚至纯文本。

- Thymeleaf 的主要目标是提供一种优雅和高度可维护的创建模板的方式。

为了实现这一点，它建立在自然模板的概念上，将其逻辑注入到模板文件中，

不会影响模板被用作设计原型。这改善了设计的沟通，弥补了设计和开发团

队之间的差距。

- Thymeleaf 也从一开始就设计了Web标准 - 特别是 HTML5 - 允许您创

建完全验证的模板，Spring Boot 官方推荐使用 thymeleaf 而不是 JSP。

- 市面上主流的 Java 模板引擎有：JSP、Velocity、Freemarker、Thymeleaf

- Thymeleaf 官网：https://www.thymeleaf.org/

- Thymeleaf 在 Github 的主页：<https://github.com/thymeleaf/thymeleaf>

**SpringSecurity**

Spring Security是一个功能强大且高度可定制的身份验证和访问控制框架。

Spring Security 是针对Spring项目的安全框架，也是Spring Boot底层安全模块默认的技术选型，他可以实现强大的Web安全控制，对于安全控制，我们仅需要引入 spring-boot-starter-security 模块，进行少量的配置，即可实现强大的安全管理！

记住几个类：

- WebSecurityConfigurerAdapter：自定义Security策略

- AuthenticationManagerBuilder：自定义认证策略

- @EnableWebSecurity：开启WebSecurity模式

Spring Security的两个主要目标是 “认证” 和 “授权”（访问控制）。

“认证”（Authentication）

身份验证是关于验证您的凭据，如用户名/用户ID和密码，以验证您的身份。

身份验证通常通过用户名和密码完成，有时与身份验证因素结合使用。

“授权” （Authorization）

授权发生在系统成功验证您的身份后，最终会授予您访问资源（如信息，文件，

数据库，资金，位置，几乎任何内容）的完全权限。

这个概念是通用的，而不是只在Spring Security 中存在。

SpringSecurity帮助文档:

<https://docs.spring.io/spring-security/site/docs/5.3.0.RELEASE/reference/html5>

**Shiro**

Apache Shiro是一个强大且易用的Java安全框架,可以使用该框架执行身份验证、授权、加密数据和会话管理。使用Shiro的易于理解的API,您可以快速、轻松地获得任何应用程序,从最小的移动应用程序到最大的网络和企业应用程序。

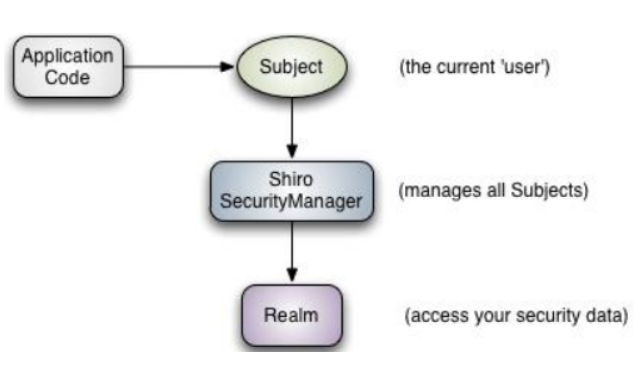
shiro帮助文档：<https://shiro.apache.org/reference.html#lend-a-hand-with-documentation>

shiro框架下载地址：<https://github.com/apache/shiro.git>

**shiro架构**

我们从外部来看 Shiro ，即从应用程序角度的来观察如何使用 Shiro 完成

工作。如下图：



可以看到：应用代码直接交互的对象是 Subject，也就是说 Shiro 的对外

API 核心就是 Subject；其每个 API 的含义：

Subject：主体，代表了当前 “用户”，这个用户不一定是一个具体的人，与

当前应用交互的任何东西都是Subject，如网络爬虫，机器人等；即一个抽

象概念；所有 Subject 都绑定到 SecurityManager，与 Subject 的所有交互

都会委托给 SecurityManager；可以把 Subject 认为是一个门面；

SecurityManager 才是实际的执行者；

SecurityManager：安全管理器；即所有与安全有关的操作都会与

SecurityManager 交互；且它管理着所有 Subject；可以看出它是 Shiro 的

核心，它负责与后边介绍的其他组件进行交互，如果学习过 SpringMVC，

你可以把它看成 DispatcherServlet 前端控制器；

Realm：领域，Shiro 从 Realm 获取安全数据（如用户、角色、权限），就

是说 SecurityManager 要验证用户身份，那么它需要从 Realm 获取相应的

用户进行比较以确定用户身份是否合法；也需要从 Realm 得到用户相应的

角色 / 权限进行验证用户是否能进行操作；可以把 Realm 看成

DataSource，即安全数据源。

也就是说对于我们而言，最简单的一个 Shiro 应用：

应用代码通过 Subject 来进行认证和授权，而 Subject 又委托给

SecurityManager；

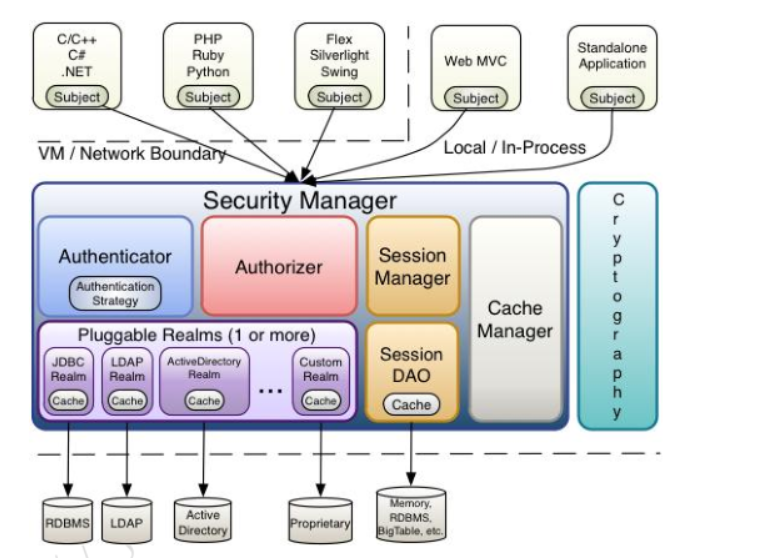
我们需要给 Shiro 的 SecurityManager 注入 Realm，从而让

SecurityManager 能得到合法的用户及其权限进行判断。

从以上也可以看出，Shiro 不提供维护用户 / 权限，而是通过 Realm 让开

发人员自己注入。

接下来我们来从 Shiro 内部来看下 Shiro 的架构，如下图所示：



Subject：主体，可以看到主体可以是任何可以与应用交互的 “用户”；

SecurityManager：相当于 SpringMVC 中的 DispatcherServlet 或者 Struts2

中的 FilterDispatcher；是 Shiro 的心脏；所有具体的交互都通过

SecurityManager 进行控制；它管理着所有 Subject、且负责进行认证和授

权、及会话、缓存的管理。

Authenticator：认证器，负责主体认证的，这是一个扩展点，如果用户觉得

Shiro 默认的不好，可以自定义实现；其需要认证策略（Authentication

Strategy），即什么情况下算用户认证通过了；

Authrizer：授权器，或者访问控制器，用来决定主体是否有权限进行相应的

操作；即控制着用户能访问应用中的哪些功能；

Realm：可以有 1 个或多个 Realm，可以认为是安全实体数据源，即用于

获取安全实体的；可以是 JDBC 实现，也可以是 LDAP 实现，或者内存实

现等等；由用户提供；注意：Shiro 不知道你的用户 / 权限存储在哪及以何

种格式存储；所以我们一般在应用中都需要实现自己的 Realm；

SessionManager：如果写过 Servlet 就应该知道 Session 的概念，Session

呢需要有人去管理它的生命周期，这个组件就是 SessionManager；而 Shiro

并不仅仅可以用在 Web 环境，也可以用在如普通的 JavaSE 环境、EJB 等

环境；所有呢，Shiro 就抽象了一个自己的 Session 来管理主体与应用之间

交互的数据；这样的话，比如我们在 Web 环境用，刚开始是一台 Web 服

务器；接着又上了台 EJB 服务器；这时想把两台服务器的会话数据放到一

个地方，这个时候就可以实现自己的分布式会话（如把数据放到

Memcached 服务器）；

SessionDAO：DAO 大家都用过，数据访问对象，用于会话的 CRUD，比如

我们想把 Session 保存到数据库，那么可以实现自己的 SessionDAO，通过

如 JDBC 写到数据库；比如想把 Session 放到 Memcached 中，可以实现

自己的 Memcached SessionDAO；另外 SessionDAO 中可以使用 Cache

进行缓存，以提高性能；

CacheManager：缓存控制器，来管理如用户、角色、权限等的缓存的；因

为这些数据基本上很少去改变，放到缓存中后可以提高访问的性能

Cryptography：密码模块，Shiro 提高了一些常见的加密组件用于如密码加

密 / 解密的。

**shiro组件**

身份验证，即在应用中谁能证明他就是他本人。一般提供如他们的身份 ID

一些标识信息来表明他就是他本人，如提供身份证，用户名 / 密码来证明。

在 shiro 中，用户需要提供 principals （身份）和 credentials（证明）给

shiro，从而应用能验证用户身份。

principals：身份，即主体的标识属性，可以是任何东西，如用户名、邮箱等，

唯一即可。一个主体可以有多个 principals，但只有一个 Primary principals，

一般是用户名 / 密码 / 手机号。

credentials：证明件/ 凭证，即只有主体知道的安全值，如密码 / 数字证书

等。

最常见的 principals 和 credentials 组合就是用户名 / 密码了。

另外两个相关的概念是之前提到的 Subject 及 Realm，分别是主体及验证

主体的数据源。

**Swagger**

Swagger 是一套基于 OpenAPI 规范（OpenAPI Specification，OAS）构建的开源工具，可以帮助我们设计、构建、记录以及使用 Rest API。

对于 Rest API 来说很重要的一部分内容就是文档，Swagger 为我们提供了一套通过代码和注解自动生成文档的方法，这一点对于保证 API 文档的及时性将有很大的帮助。

Swagger 主要包含了以下三个部分：

Swagger Editor：基于浏览器的编辑器，我们可以使用它编写我们 OpenAPI 规

范。

Swagger UI：它会将我们编写的 OpenAPI 规范呈现为交互式的 API 文档，

后文我将使用浏览器来查看并且操作我们的 Rest API。

Swagger Codegen：它可以通过为 OpenAPI（以前称为 Swagger）规范定义

的任何 API 生成服务器存根和客户端 SDK 来简化构建过程。

由于Spring的流行，Marty Pitt编写了一个基于Spring的组件

swagger-springmvc，用于将swagger集成到springmvc中来，而springfox

则是从这个组件发展而来。

通常SpringBoot项目整合swagger需要用到两个依赖：springfox-swagger2

和springfox-swagger-ui，用于自动生成swagger文档。

springfox-swagger2：这个组件的功能用于帮助我们自动生成描述API的json

文件。

springfox-swagger-ui：就是将描述API的json文件解析出来，用一种更友好

的方式呈现出来。

Springfox3.0.0发布后，直接导入springfox-boot-starter依赖即可。

**RPC**

RPC【Remote Procedure Call】是指远程过程调用，是一种进程间通信方式，他是一种技术的思想，而不是规范。它允许程序调用另一个地址空间（通常是共享网络的另一台机器上）的过程或函数，而不用程序员显式编码这个远程调用的细节。即程序员无论是调用本地的还是远程的函数，本质上编写的调用代码基本相同。

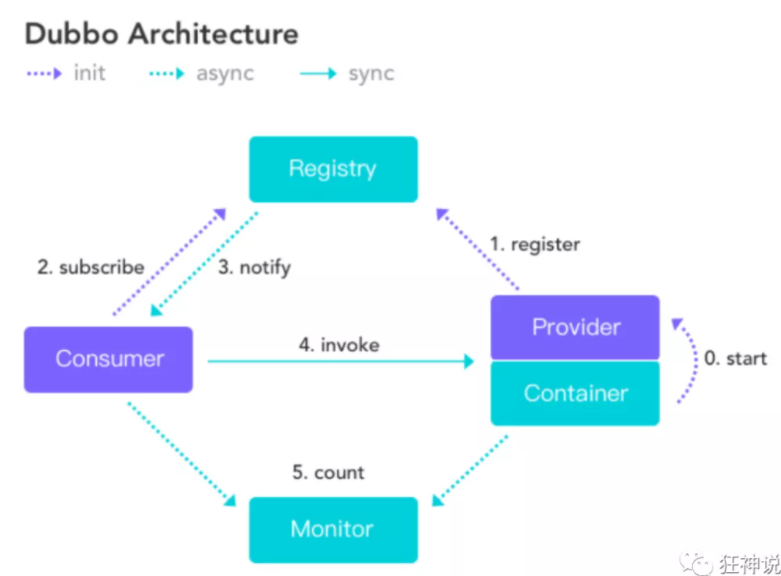
也就是说两台服务器A，B，一个应用部署在A服务器上，想要调用B服务器上应用提供的函数/方法，由于不在一个内存空间，不能直接调用，需要通过网络来表达调用的语义和传达调用的数据。为什么要用RPC呢？就是无法在一个进程内，甚至一个计算机内通过本地调用的方式完成的需求，比如不同的系统间的通讯，甚至不同的组织间的通讯，由于计算能力需要横向扩展，需要在多台机器组成的集群上部署应用。RPC就是要像调用本地的函数一样去调远程函数。

推荐阅读文章：https://www.jianshu.com/p/2accc2840a1b

**zookeeper和dubbo**

Apache Dubbo是一款高性能、轻量级的开源Java RPC框架，它提供了三大核心能力：面向接口的远程方法调用，智能容错和负载均衡，以及服务自动注册和发现。

Dubbo基本概念



服务提供者（Provider）：暴露服务的服务提供方，服务提供者在启动时，向注册中心注册自己提供的服务。

服务消费者（Consumer）：调用远程服务的服务提供方，服务消费者在启动时，向注册中心订阅自己所需的服务，服务消费者，从提供者地址列表中，基于软负载均衡算法，选一台提供者进行调用，如果调用失败，再选另一台调用。

注册中心（Registry）：注册中心返回服务提供者地址列表给消费者，如果有变更，注册中心将基于长连接推送变更数据给消费者。dubbo推荐使用Apache zookeeper作为注册中心。

监控中心（Monitor）：服务消费者和提供者，在内存中累计调用次数和调用时间，定时每分钟发送一次统计数据到监控中心

调用关系说明

服务容器负责启动，加载，运行服务提供者。

服务提供者在启动时，向注册中心注册自己提供的服务。

服务消费者在启动时，向注册中心订阅自己所需的服务。

注册中心返回服务提供者地址列表给消费者，如果有变更，注册中心将基于长连接推送变更数据给消费者。

服务消费者，从提供者地址列表中，基于软负载均衡算法，选一台提供者进行调用，如果调用失败，再选另一台调用。

服务消费者和提供者，在内存中累计调用次数和调用时间，定时每分钟发送一次统计数据到监控中心。

dubbo下载地址:

https://github.com/apache/dubbo

zookeeper，它是一个分布式服务框架，是Apache Hadoop 的一个子项目，它主要是用来解决分布式应用中经常遇到的一些数据管理问题，如：统一命名服务、状态同步服务、集群管理、分布式应用配置项的管理等。

zookeeper下载地址:

https://zookeeper.apache.org/releases.html

