



- **Spring Boot必知必会**
 - **什么是Spring Boot**
 - **Features**
 - **Spring Boot环境安装及初体验**
 - **系统要求**
 - **初体验**
 - **通过Web界面构建**
 - **通过IntelliJ IDEA使用**
 - **项目结构**
 - **pom依赖**
 - **应用入口类**
 - **启动项目**
 - **Spring Boot常用注解**
- **Spring Boot原理**
 - **核心注解**
 - @SpringBootApplication**
 - **@Configuration**
 - **@ComponentScan**

- **@EnableAutoConfiguration**
- **自动配置幕后英雄：SpringFactoriesLoader详解**
- **Spring Boot启动原理**
- **Spring Boot如何配置静态资源的地址与访问路径**
 - **spring.mvc.static-path-pattern**
 - **spring.resources.static-locations**
 - **静态资源的Bean配置**
- **Spring Boot 实战**
 - **自定义starter**
 - **为什么要自定义starter**
 - **自定义starter的命名规则**
 - **示例走起**
 - **Spring Boot整合Quartz实现定时任务**
 - **附录 cron表达式详解**
 - **配置实例：**
- **Q&A答疑**

Spring Boot必知必会

什么是Spring Boot

```
/**
 * Spring Boot makes it easy to create stand-alone,
 * production-grade Spring based Applications that you can "just run".
 *
 * We take an opinionated view of the Spring platform and third-party libraries
 * so you can get started with minimum fuss. Most Spring Boot applications need
 * very little Spring configuration.
 */
```

它使用“习惯优于配置”（项目中存在大量的配置，此外还内置了一个习惯性的配置，让你无需手动配置）的理念让你的项目快速运行起来。

它并不是什么新的框架，而是默认配置了很多框架的使用方式，就像Maven整合了所有的jar包一样，Spring Boot整合了所有框架。

Features

- 创建独立的Spring应用程序
- 直接嵌入Tomcat, Jetty或Undertow（无需部署WAR文件）
- 提供“初始”的POM文件内容，以简化Maven配置
- 尽可能自动配置Spring
- 提供生产就绪的功能，如指标，健康检查和外部化配置
- 绝对无代码生成，也不需要XML配置

Spring Boot环境安装及初体验

目前Spring Boot正式版为2.3.2 CURRENT GA。简单介绍下版本区别：

- GA:General Availability, 正式发布的版本，官方推荐使用此版本。在国外都是用GA来说明release版本的。
- PRE:预览版，内部测试版。主要是给开发人员和测试人员测试和找BUG用的，不建议使用。
- SNAPSHOT: 快照版，可以稳定使用，且仍在继续改进版本。

系统要求

- JDK 1.8 or later
- Gradle 4+ or Maven 3.2+
- A favorite text editor or IDE（推荐IntelliJ IDEA）

初体验

构建一个Spring Boot的Maven项目，强烈推荐Spring Initializr，它从本质上来说就是一个Web应用程序，它能为我们生成Spring Boot项目结构。

Spring Initializr有几种用法：

通过Web界面构建

1. 访问：<http://start.spring.io/>
2. 选择构建工具Maven Project、Spring Boot版本2.3.2以及一些工程基本信息
3. 点击Generate Project下载项目压缩包
4. 导入到你的工程，如果是IDEA，则需要：
 - a. 菜单中选择File->New->Project from Existing Sources...
 - b. 选择解压后的项目文件夹，点击OK
 - c. 点击Import project from external model并选择Maven，点击Next到底为止。
 - d. 若你的环境有多个版本的JDK，注意到选择Java SDK的时候请选择Java 8以上的版本

通过IntelliJ IDEA使用

创建Spring Boot操作步骤如下：

1. 在File菜单里面选择 New > Project，然后选择Spring Initializr，接着如下图一步步操作即可。
2. ...

项目结构

如你所见，项目里面基本没有代码，除了几个空目录外，还包含如下几样东西。

- pom.xml：Maven构建说明文件。
- SpringdemoApplication.java：一个带有main()方法的类，用于启动应用程序（关键）。
- SpringdemoApplicationTests.java：一个空的JUnit测试类，它加载了一个使用Spring Boot字典配置功能的Spring应用程序上下文。
- application.properties：一个空的properties文件，你可以根据需要添加配置属性。

pom依赖

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.2.4.RELEASE</version>
    <relativePath/> <!-- Lookup parent from repository -->
  </parent>
```

```

<groupId>com.zhuawa</groupId>
<artifactId>springdemo</artifactId>
<version>0.0.1-SNAPSHOT</version>
<name>springdemo</name>
<description>Demo project for Spring Boot</description>

<properties>
  <java.version>1.8</java.version>
</properties>

<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>

  <dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <optional>true</optional>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
    <exclusions>
      <exclusion>
        <groupId>org.junit.vintage</groupId>
        <artifactId>junit-vintage-engine</artifactId>
      </exclusion>
    </exclusions>
  </dependency>
</dependencies>

<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
  </plugins>
</build>

</project>

```

- Spring Boot父级依赖

```

<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.2.4.RELEASE</version>
  <relativePath/> <!-- Lookup parent from repository -->
</parent>

```

有了Spring Boot父级依赖，表示当前的项目就是Spring Boot项目了，spring-boot-starter-parent是一个特殊的starter，它用来提供相关的Maven默认依赖，使用它之后，常用的包依赖可以省去version标签。

- 起步依赖 spring-boot-starter-xx

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

Spring Boot提供了很多“开箱即用”的依赖模块，都是以spring-boot-starter-xx作为命名的。

Spring Boot通过提供众多起步依赖降低项目依赖的复杂度。起步依赖本质上是一个Maven项目对象模型（Project Object Model, POM），定义了对其他库的传递依赖，这些东西加在一起即支持某项功能。很多起步依赖的命名都暗示了它们提供的某种或者某类功能。

- Spring Boot Maven插件

```
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
  </plugins>
</build>
```

Spring Boot Maven插件提供了许多方便的功能：

- 把项目打包成一个可执行的超级JAR（uber-JAR），包括把应用程序的所有依赖打入JAR文件内，并为JAR添加一个描述文件，其中的内容能让你用java -jar来运行应用程序。
- 搜索public static void main()方法来标记为可运行类。

应用入口类

SpringdemoApplication是一个很关键的启动类，程序的入口就是这里，为了演示简单，我们不再新建控制类，而是直接在这个入口类中编写，添加@RestController以及index方法

```
package com.zhuawa.springdemo;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@SpringBootApplication
public class SpringdemoApplication {
```

```

@RequestMapping("/")
public String index() {
    return "Hello Spring Boot";
}

public static void main(String[] args) {
    SpringApplication.run(SpringdemoApplication.class, args);
}
}

```

- @SpringBootApplication是Spring Boot项目的核心注解，主要目的是开启自动配置。后续讲解原理的时候再深入介绍。
- main方法这是一个标准的Java应用的main的方法，主要作用是作为项目启动的入口。
- @RestController注解等价于@Controller+@ResponseBody的结合，使用这个注解的类里面的方法都以json格式输出。

启动项目

打开浏览器访问 <http://localhost:8080>, 你就能看到页面显示Hello Spring Boot效果。

Spring Boot常用注解

1. @RequestMapping: 是一个用来处理请求地址映射的注解，它提供路由信息，可用于类或方法上。用于类上，表示类中的所有响应请求的方法都是以该地址作为父路径。它告诉Spring任何来自"/"路径的HTTP请求都应该被映射到index方法。@RestController注解告诉Spring以字符串的形式渲染结果，并直接返回给调用者。
2. @Profiles: Spring Profiles提供了一种隔离应用程序配置的方式，并让这些配置只能在特定环境下生效。任何@Component或@Configuration都能被@Profiles标记，从而限制加载它时机。
3. @ResponseBody: 表示该方法的返回结果直接写入 HTTP response body中，一般在异步获取数据时使用，在使用@RequestMapping后，返回值通常解析为跳转路径，加上@ResponseBody后返回结果不会被解析为跳转路径，而是直接写入HTTP response body中。比如异步获取json数据，加上@ResponseBody后，会直接返回json数据。
4. @Component: 泛指组件，当组件不好归类的时候，我们可以使用这个注解进行标注。一般公共的方法我会用上这个注解。
5. @Autowired: byType方式，把配置好的Bean拿来用，完成属性、方法的组装，它可以对类成员变量、方法及构造函数进行标注，完成自动装配的工作。当加上(required=false)时，就算找不到bean也不会报错。
6. @RequestParam: 用在方法的参数前面。 @RequestParam String a
=request.getParameter("a");
7. @PathVariable: 路径变量。 RequestMapping("user/get/mac/{macAddress}") public String
getByMacAddress(@PathVariable String macAddress){ //do something; }
8. 全局处理异常的: @ControllerAdvice: 包含@Component。可以被扫描到，统一处理异常。
@ExceptionHandler (Exception.class) : 用在方法上面表示遇到这个异常就执行以下方法。

```

/**
 * 全局异常处理
 */
@ControllerAdvice
class GlobalDefaultExceptionHandler {
    public static final String DEFAULT_ERROR_VIEW = "error";

    @ExceptionHandler({TypeMismatchException.class, NumberFormatException.class})
    public ModelAndView formatErrorHandler(HttpServletRequest req, Exception e) throws
Exception {
        ModelAndView mav = new ModelAndView();
        mav.addObject("error", "参数类型错误");
        mav.addObject("exception", e);
        mav.addObject("url", RequestUtils.getCompleteRequestUrl(req));
        mav.addObject("timestamp", new Date());
        mav.setViewName(DEFAULT_ERROR_VIEW);
        return mav;
    }
}

```

9. 通过@value注解来读取application.properties里面的配置
 10. @Configuration标注在类上，相当于把该类作为spring的xml配置文件中的<beans>，作用为：配置spring容器(应用上下文)
 11. @Bean标注在方法上(返回某个实例的方法)，等价于spring的xml配置文件中的<bean>，作用为：注册bean对象
- 注：
 - (1)、@Bean注解在返回实例的方法上，如果未通过@Bean指定bean的名称，则默认与标注的方法名相同；
 - (2)、@Bean注解默认作用域为单例singleton作用域，可通过@Scope("prototype")设置为原型作用域；

```

@Configuration
public class TestConfiguration {
    public TestConfiguration(){
        System.out.println("spring容器启动初始化。。。");
    }

    // @Bean注解注册bean, 同时可以指定初始化和销毁方法
    // @Bean(name="testBean", initMethod="start", destroyMethod="cleanUp")
    @Bean
    @Scope("prototype")
    public TestBean testBean() {
        return new TestBean();
    }
}

```

```

public class TestMain {
    public static void main(String[] args) {
        ApplicationContext context = new
        AnnotationConfigApplicationContext(TestConfiguration.class);
    }
}

```



```

        //获取bean
        TestBean tb = context.getBean("testBean");
        tb.sayHello();
    }
}

```

Spring Boot原理

前面讲的例子我们看到，开发任何一个Spring Boot项目，都会用到下面的启动类

```

@SpringBootApplication
public class SpringdemoApplication {
    public static void main(String[] args) {
        SpringApplication.run(SpringdemoApplication.class, args);
    }
}

```

可以看到，Annotation定义（@SpringBootApplication）和类定义（SpringApplication.run）最为瞩目，所以要揭开Spring Boot的神秘面纱，我们要从这两位开始就可以了。

核心注解@SpringBootApplication

```

@Target(ElementType.TYPE)           // 注解的适用范围，其中TYPE用于描述类、接口（包括包注解类型）或enum声明
@Retention(RetentionPolicy.RUNTIME) // 注解的生命周期，保留到class文件中（三个生命周期）
@Documented                         // 表明这个注解应该被javadoc记录
@Inherited                          // 子类可以继承该注解
@SpringBootConfiguration            // 继承了Configuration，表示当前是注解类
@EnableAutoConfiguration              // 开启springboot的注解功能，springboot的四大神器之一，其借助@import的帮助
@ComponentScan(excludeFilters = {    // 扫描路径设置（具体使用待确认）
    @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 SpringdemoApplication {
    public static void main(String[] args) {
        SpringApplication.run(SpringdemoApplication.class, args);
    }
}

```

每次写这3个比较繁琐，所以写一个@SpringBootApplication方便点。接下来分别介绍这3个Annotation。

@Configuration

这里的@Configuration对我们来说不陌生，它就是JavaConfig形式的Spring IoC容器的配置类使用的那个@Configuration，SpringBoot社区推荐使用基于JavaConfig的配置形式，所以，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。举几个简单例子回顾下，XML跟config配置方式的区别：

表达形式层面

- 基于XML配置的方式是这样：

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd"
       default-lazy-init="true">
    <!--bean定义-->
</beans>

```

- 而基于JavaConfig的配置方式是这样：

```

@Configuration
public class MockConfiguration{
    //bean定义
}

```

任何一个标注了@Configuration的Java类定义都是一个JavaConfig配置类。

- 注册bean定义层面 基于XML的配置形式是这样：

```

<bean id="mockService" class="..MockServiceImpl">
    ...
</bean>

```

- 而基于JavaConfig的配置形式是这样的：

```
@Configuration
public class MockConfiguration{
    @Bean
    public MockService mockService(){
        return new MockServiceImpl();
    }
}
```

任何一个标注了@Bean的方法，其返回值将作为一个bean定义注册到Spring的IoC容器，方法名将默认成该bean定义的id。

- 表达依赖注入关系层面 为了表达bean与bean之间的依赖关系，在XML形式中一般是这样：

```
<bean id="mockService" class="..MockServiceImpl">
    <property name="dependencyService" ref="dependencyService" />
</bean>

<bean id="dependencyService" class="DependencyServiceImpl"></bean>
```

- 而基于JavaConfig的配置形式是这样的：

```
@Configuration
public class MockConfiguration{
    @Bean
    public MockService mockService(){
        return new MockServiceImpl(dependencyService());
    }

    @Bean
    public DependencyService dependencyService(){
        return new DependencyServiceImpl();
    }
}
```

如果一个bean的定义依赖其他bean,则直接调用对应的JavaConfig类中依赖bean的创建方法就可以了。

@ComponentScan

@ComponentScan这个注解在Spring中很重要，它对应XML配置中的元素，**@ComponentScan的功能其实就是自动扫描并加载符合条件的组件（比如@Component和@Repository等）或者bean定义，最终将这些bean定义加载到IoC容器中。**

我们可以通过**basePackages**等属性来细粒度的定制@ComponentScan自动扫描的范围，如果不指定，则默认Spring框架实现会**从声明@ComponentScan所在类的package进行扫描。**

注：所以SpringBoot的启动类最好是放在root package下，因为默认不指定basePackages。

@EnableAutoConfiguration

个人认为@EnableAutoConfiguration这个Annotation最为重要，所以放在最后来解读，大家是否还记得Spring框架提供的各种名字为@Enable开头的Annotation定义？比如@EnableScheduling、@EnableCaching、@EnableMBeanExport等，@EnableAutoConfiguration的理念和做事方式其实一脉相承，简单概括一下就是，借助@Import的支持，收集和注册特定场景相关的bean定义。

- @EnableScheduling是通过@Import将Spring调度框架相关的bean定义都加载到IoC容器。
- @EnableMBeanExport是通过@Import将JMX相关的bean定义加载到IoC容器。

而@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，仅此而已！

@EnableAutoConfiguration作为一个复合Annotation,其自身定义关键信息如下：

```
@SuppressWarnings("deprecation")
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@AutoConfigurationPackage
@Import(EnableAutoConfigurationImportSelector.class)
public @interface EnableAutoConfiguration {
    ...
}
```

两个比较重要的注解：

- @AutoConfigurationPackage：自动配置包
- @Import：导入自动配置的组件

AutoConfigurationPackage注解：

```
static class Registrar implements ImportBeanDefinitionRegistrar, DeterminableImports {

    @Override
    public void registerBeanDefinitions(AnnotationMetadata metadata,
        BeanDefinitionRegistry registry) {
        register(registry, new PackageImport(metadata).getPackageName());
    }
}
```

它其实是注册了一个Bean的定义。

new PackageImport(metadata).getPackageName()，它其实返回了当前主程序类的 同级以及子级 的包组件。

这也就是为什么，我们要把DemoApplication放在项目的最高级中。

Import(AutoConfigurationImportSelector.class)注解：

AutoConfigurationImportSelector 继承了 DeferredImportSelector 继承了 ImportSelector

ImportSelector有一个方法为：selectImports。

```
@Override
public String[] selectImports(AnnotationMetadata annotationMetadata) {
    if (!isEnabled(annotationMetadata)) {
        return NO_IMPORTS;
    }
    AutoConfigurationMetadata autoConfigurationMetadata =
AutoConfigurationMetadataLoader
        .loadMetadata(this.beanClassLoader);
    AutoConfigurationEntry autoConfigurationEntry =
getAutoConfigurationEntry(autoConfigurationMetadata,
        annotationMetadata);
    return StringUtils.toStringArray(autoConfigurationEntry.getConfigurations());
}
```

```
/**
 * Return the {@link AutoConfigurationEntry} based on the {@link AnnotationMetadata}
 * of the importing {@link Configuration @Configuration} class.
 * @param autoConfigurationMetadata the auto-configuration metadata
 * @param annotationMetadata the annotation metadata of the configuration class
 * @return the auto-configurations that should be imported
 */
protected AutoConfigurationEntry getAutoConfigurationEntry(AutoConfigurationMetadata
autoConfigurationMetadata,
    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 = filter(configurations, autoConfigurationMetadata);
    fireAutoConfigurationImportEvents(configurations, exclusions);
    return new AutoConfigurationEntry(configurations, exclusions);
}
```

```
protected List<String> getCandidateConfigurations(AnnotationMetadata metadata,
AnnotationAttributes 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;
    }

```

```

/**
 * The location to look for factories.
 * <p>Can be present in multiple JAR files.
 */
public static final String FACTORIES_RESOURCE_LOCATION = "META-INF/spring.factories";

private static Map<String, List<String>> loadSpringFactories(@Nullable ClassLoader
classLoader) {
    MultiValueMap<String, String> result = cache.get(classLoader);
    if (result != null) {
        return result;
    }

    try {
        Enumeration<URL> urls = (classLoader != null ?
            classLoader.getResources(FACTORIES_RESOURCE_LOCATION) :
            ClassLoader.getSystemResources(FACTORIES_RESOURCE_LOCATION));
        result = new LinkedMultiValueMap<>();
        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();
                for (String factoryImplementationName :
StringUtils.commaDelimitedListToStringArray((String) entry.getValue())) {
                    result.add(factoryTypeName, factoryImplementationName.trim());
                }
            }
        }
        cache.put(classLoader, result);
        return result;
    }
    catch (IOException ex) {
        throw new IllegalArgumentException("Unable to load factories from location ["
+
            FACTORIES_RESOURCE_LOCATION + "]", ex);
    }
}

```

它其实是去加载 **public static final String FACTORIES_RESOURCE_LOCATION = "META-INF/spring.factories";** 外部文件，这个外部文件，有很多自动配置的类。

其中，最关键的要属 **@Import(EnableAutoConfigurationImportSelector.class)**，借助 **EnableAutoConfigurationImportSelector**，**@EnableAutoConfiguration** 可以帮助 Spring Boot 应用将所有符合条件的 **@Configuration** 配置都加载到当前 Spring Boot 创建并使用的 IoC 容器。就像一只“八爪鱼”一样。

自动配置幕后英雄：SpringFactoriesLoader详解

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

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。

```
public abstract class SpringFactoriesLoader {  
    //...  
    public static <T> List<T> loadFactories(Class<T> factoryClass, ClassLoader  
    classLoader) {  
        ...  
    }  
  
    public static List<String> loadFactoryNames(Class<?> factoryClass, ClassLoader  
    classLoader) {  
        ....  
    }  
}
```

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.springframework.boot.autoconfigure.EnableAutoConfiguration作为查找的Key,获取对应的一
组@Configuration类

所以，@EnableAutoConfiguration自动配置的过程就变成了：从classpath中搜寻所有的META-
INF/spring.factories配置文件，并将其中
org.springframework.boot.autoconfigure.EnableAutoConfiguration对应的配置项通过反射
(Java Reflection) 实例化为对应的标注了@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。

Spring Boot启动原理

SpringApplication的run方法的实现是启动原理探寻的起点，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，这个方法里面首先要创建一个
SpringApplication对象实例，然后调用这个创建好的SpringApplication的实例方法。在
SpringApplication实例初始化的时候，它会提前做几件事情：

```
public static ConfigurableApplicationContext run(Class<?>[] primarySources, String[]  
args) {  
    return new SpringApplication(primarySources).run(args);  
}
```

- 根据classpath里面是否存在某个特征类
(org.springframework.web.context.ConfigurableWebApplicationContext) 来决定是否应
该创建一个为Web应用使用的ApplicationContext类型。

- 使用SpringFactoriesLoader在应用的classpath中查找并加载所有可用的ApplicationContextInitializer。
- 使用SpringFactoriesLoader在应用的classpath中查找并加载所有可用的ApplicationListener。
- 推断并设置main方法的定义类。

```
@SuppressWarnings({ "unchecked", "rawtypes" })
public SpringApplication(ResourceLoader resourceLoader, Class<?>... primarySources) {
    this.resourceLoader = resourceLoader;
    Assert.notNull(primarySources, "PrimarySources must not be null");
    //把SpringdemoApplication.class设置为属性存储起来
    this.primarySources = new LinkedHashSet<>(Arrays.asList(primarySources));
    //设置应用类型为Standard还是Web
    this.webApplicationType = WebApplicationType.deduceFromClasspath();
    //设置初始化器 (Initializer), 最后会调用这些初始化器
    setInitializers((Collection)
getSpringFactoriesInstances(ApplicationContextInitializer.class));
    //设置监听器 (Listener)
    setListeners((Collection)
getSpringFactoriesInstances(ApplicationListener.class));
    this.mainApplicationClass = deduceMainApplicationClass();
}
```

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!”。

```
/**
 * Run the Spring application, creating and refreshing a new
 * {@link ApplicationContext}.
 * @param args the application arguments (usually passed from a Java main method)
 * @return a running {@link ApplicationContext}
 */
public ConfigurableApplicationContext run(String... args) {
    //计时工具
    Stopwatch stopwatch = new Stopwatch();
    stopwatch.start();
    ConfigurableApplicationContext context = null;
    Collection<SpringBootExceptionHandler> exceptionReporters = new ArrayList<>();
    configureHeadlessProperty();
    //第一步: 获取并启动监听器
    SpringApplicationRunListeners listeners = getRunListeners(args);
    listeners.starting();
    try {
        ApplicationArguments applicationArguments = new
DefaultApplicationArguments(args);
        //第二步: 根据SpringApplicationRunListeners以及参数来准备环境
        ConfigurableEnvironment environment = prepareEnvironment(listeners,
applicationArguments);
        configureIgnoreBeanInfo(environment);
        //准备Banner打印机-就是启动Spring Boot的时候在console上的ASCII艺术字体
        Banner printedBanner = printBanner(environment);
    }
```



```

        // 第三步: 创建Spring容器
        context = createApplicationContext();
        exceptionReporters =
getSpringFactoriesInstances(SpringBootExceptionReporter.class,
        new Class[] { ConfigurableApplicationContext.class }, context);
        // 第四步: Spring容器前置处理
        prepareContext(context, environment, listeners, applicationArguments,
printedBanner);
        // 第五步: 刷新容器
        refreshContext(context);
        // 第六步: Spring容器后置处理
        afterRefresh(context, applicationArguments);
        stopWatch.stop();
        if (this.logStartupInfo) {
            new
StartupInfoLogger(this.mainApplicationClass).logStarted(getApplicationLog(), stopWatch);
        }
        // 第七步: 发出结束执行的事件
        listeners.started(context);
        // 第八步: 执行Runners
        callRunners(context, applicationArguments);
    }
    catch (Throwable ex) {
        handleRunFailure(context, ex, exceptionReporters, listeners);
        throw new IllegalStateException(ex);
    }

    try {
        listeners.running(context);
    }
    catch (Throwable ex) {
        handleRunFailure(context, ex, exceptionReporters, null);
        throw new IllegalStateException(ex);
    }
    // 返回容器
    return context;
}

```

3) 创建并配置当前Spring Boot应用将要使用的Environment（包括配置要使用的PropertySource以及Profile）。

```

private ConfigurableEnvironment prepareEnvironment(SpringApplicationRunListeners
listeners,
        ApplicationArguments applicationArguments) {
    // Create and configure the environment
    ConfigurableEnvironment environment = getOrCreateEnvironment();
    configureEnvironment(environment, applicationArguments.getSourceArgs());
    ConfigurationPropertySources.attach(environment);
    listeners.environmentPrepared(environment);
    bindToSpringApplication(environment);
    if (!this.isCustomEnvironment) {
        environment = new
EnvironmentConverter(getClassLoader()).convertEnvironmentIfNecessary(environment,
        deduceEnvironmentClass());
    }
}

```

```

        ConfigurationPropertySources.attach(environment);
        return environment;
    }

```

4) 遍历调用所有SpringApplicationRunListener的environmentPrepared()的方法，告诉他们：“当前SpringBoot应用使用的Environment准备好了咯！”。

```

void environmentPrepared(ConfigurableEnvironment environment) {
    for (SpringApplicationRunListener listener : this.listeners) {
        listener.environmentPrepared(environment);
    }
}

```

5) 如果SpringApplication的showBanner属性被设置为true，则打印banner。

```

private Banner printBanner(ConfigurableEnvironment environment) {
    if (this.bannerMode == Banner.Mode.OFF) {
        return null;
    }
    ResourceLoader resourceLoader = (this.resourceLoader != null) ?
this.resourceLoader
        : new DefaultResourceLoader(getClassLoader());
    SpringApplicationBannerPrinter bannerPrinter = new
SpringApplicationBannerPrinter(resourceLoader, this.banner);
    if (this.bannerMode == Mode.LOG) {
        return bannerPrinter.print(environment, this.mainApplicationClass, logger);
    }
    return bannerPrinter.print(environment, this.mainApplicationClass, System.out);
}

```

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ApplicationContext-Initializer，然后遍历调用这些ApplicationContextInitializer的initialize (applicationContext) 方法来对已经创建好的ApplicationContext进行进一步的处理。

```

@SuppressWarnings({ "rawtypes", "unchecked" })
protected void applyInitializers(ConfigurableApplicationContext context) {
    for (ApplicationContextInitializer initializer : getInitializers()) {
        Class<?> requiredType =
GenericResolver.resolveTypeArgument(initializer.getClass(),
        ApplicationContextInitializer.class);
        Assert.isInstanceOf(requiredType, context, "Unable to call initializer.");
        initializer.initialize(context);
    }
}

```

```

    }
}

```

8) 遍历调用所有SpringApplicationRunListener的contextPrepared()方法。

```

private void prepareContext(ConfigurableApplicationContext context,
ConfigurableEnvironment environment,
    SpringApplicationRunListeners listeners, ApplicationArguments
applicationArguments, Banner printedBanner) {
    context.setEnvironment(environment);
    postProcessApplicationContext(context);
    applyInitializers(context);
    listeners.contextPrepared(context);
    if (this.logStartupInfo) {
        logStartupInfo(context.getParent() == null);
        logStartupProfileInfo(context);
    }
    // Add boot specific singleton beans
    ConfigurableListableBeanFactory beanFactory = context.getBeanFactory();
    beanFactory.registerSingleton("springApplicationArguments",
applicationArguments);
    if (printedBanner != null) {
        beanFactory.registerSingleton("springBootBanner", printedBanner);
    }
    if (beanFactory instanceof DefaultListableBeanFactory) {
        ((DefaultListableBeanFactory) beanFactory)
.setAllowBeanDefinitionOverriding(this.allowBeanDefinitionOverriding);
    }
    if (this.lazyInitialization) {
        context.addBeanFactoryPostProcessor(new
LazyInitializationBeanFactoryPostProcessor());
    }
    // Load the sources
    Set<Object> sources = getAllSources();
    Assert.notEmpty(sources, "Sources must not be empty");
    load(context, sources.toArray(new Object[0]));
    listeners.contextLoaded(context);
}

```

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。

```

private void prepareAnalyzer(ConfigurableApplicationContext context, FailureAnalyzer
analyzer) {
    if (analyzer instanceof BeanFactoryAware) {
        ((BeanFactoryAware) analyzer).setBeanFactory(context.getBeanFactory());
    }
    if (analyzer instanceof EnvironmentAware) {
        ((EnvironmentAware) analyzer).setEnvironment(context.getEnvironment());
    }
}

```

10) 遍历调用所有SpringApplicationRunListener的contextLoaded()方法。

```
void contextLoaded(ConfigurableApplicationContext context) {
    for (SpringApplicationRunListener listener : this.listeners) {
        listener.contextLoaded(context);
    }
}
```

11) 调用ApplicationContext的refresh()方法，完成IoC容器可用的最后一道工序。

```
private void refreshContext(ConfigurableApplicationContext context) {
    refresh(context);
    if (this.registerShutdownHook) {
        try {
            context.registerShutdownHook();
        }
        catch (AccessControlException ex) {
            // Not allowed in some environments.
        }
    }
}
```

12) 查找当前ApplicationContext中是否注册有CommandLineRunner，如果有，则遍历执行它们。

```
private void callRunners(ApplicationContext context, ApplicationArguments args) {
    List<Object> runners = new ArrayList<>();
    runners.addAll(context.getBeansOfType(ApplicationRunner.class).values());
    runners.addAll(context.getBeansOfType(CommandLineRunner.class).values());
    AnnotationAwareOrderComparator.sort(runners);
    for (Object runner : new LinkedHashSet<>(runners)) {
        if (runner instanceof ApplicationRunner) {
            callRunner((ApplicationRunner) runner, args);
        }
        if (runner instanceof CommandLineRunner) {
            callRunner((CommandLineRunner) runner, args);
        }
    }
}
```

13) 正常情况下，遍历执行SpringApplicationRunListener的finished()方法、（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）去除事件通知点后，整个流程如下：

```
void failed(ConfigurableApplicationContext context, Throwable exception) {
    for (SpringApplicationRunListener listener : this.listeners) {
        callFailedListener(listener, context, exception);
    }
}
```

Spring Boot如何配置静态资源的地址与访问路径

静态资源，例如HTML文件、JS文件，设计到的Spring Boot配置有两项，一是“spring.mvc.static-path-pattern”，一是“spring.resources.static-locations”，很多人都难以分辨它们之间的差异，所以经常出现的结果就是404错误，无法找到静态资源。

spring.mvc.static-path-pattern

spring.mvc.static-path-pattern代表的含义是我们应该以什么样的路径来访问静态资源，换句话说，只有静态资源满足什么样的匹配条件，Spring Boot才会处理静态资源请求，以官方配置为例：

```
# 这表示只有静态资源的访问路径为/resources/**时，才会处理请求
spring.mvc.static-path-pattern=/resources/**
```

假定采用默认的配置端口，那么只有请求地址类似于“<http://localhost:8080/resources/jquery.js>”时，Spring Boot才会处理此请求，处理方式是根据模式匹配后的文件名查找本地文件，那么应该在什么地方查找本地文件呢？这就是“spring.resources.static-locations”的作用了。

spring.resources.static-locations

spring.resources.static-locations”用于告诉Spring Boot应该在何处查找静态资源文件，这是一个列表性的配置，查找文件时会依赖于配置的先后顺序依次进行，默认的官方配置如下：

```
spring.resources.static-
locations=classpath:/static,classpath:/public,classpath:/resources,classpath:/META-
INF/resources
```

继续以上面的请求地址为例，“<http://localhost:8080/resources/jquery.js>”就会在上述的四个路径中依次查找是否存在“jquery.js”文件，如果找到了，则返回此文件，否则返回404错误。

静态资源的Bean配置

从上面可以看出，“spring.mvc.static-path-pattern”与“spring.resources.static-locations”组合起来演绎了nginx的映射配置，如果熟悉Spring MVC，那么理解起来更加简单，它们的作用可以用Bean配置表示，如下：

```
@Configuration
@EnableWebMvc
public class WebConfig extends WebMvcConfigurerAdapter {

    @Override
    public void addResourceHandlers(ResourceHandlerRegistry registry) {
        registry.addResourceHandler("/resources/**")
            .addResourceLocations("/public-resources/")
            .setCacheControl(CacheControl.maxAge(1, TimeUnit.HOURS).cachePublic());
    }
}
```

```
}
```

或者等同与以下的XML。

```
<mvc:resources mapping="/resources/**" location="/public-resources/">
  <mvc:cache-control max-age="3600" cache-public="true"/>
</mvc:resources>
```

综上，“spring.mvc.static-path-pattern”用于阐述HTTP请求地址，而“spring.resources.static-locations”则用于描述静态资源的存放位置。

Spring Boot 实战

自定义starter

SpringBoot中的starter是一种非常重要的机制，能够抛弃以前繁杂的配置，将其统一集成进starter，应用者只需要在maven中引入starter依赖，SpringBoot就能自动扫描到要加载的信息并启动相应的默认配置。starter让我们摆脱了各种依赖库的处理，需要配置各种信息的困扰。SpringBoot会自动通过classpath路径下的类发现需要的Bean，并注册进IOC容器。SpringBoot提供了针对日常企业应用研发各种场景的spring-boot-starter依赖模块。所有这些依赖模块都遵循着约定成俗的默认配置，并允许我们调整这些配置，即遵循“约定大于配置”的理念。

为什么要自定义starter

在我们的日常开发工作中，经常会有一些独立于业务之外的配置模块，我们经常将其放到一个特定的包下，然后如果另一个工程需要复用这块功能的时候，需要将代码硬拷贝到另一个工程，重新集成一遍，麻烦至极。如果我们将这些可独立于业务代码之外的配置模块封装成一个个starter，复用的时候只需要将其在pom中引用依赖即可，SpringBoot为我们完成自动装配，简直不要太爽。

自定义starter的命名规则

SpringBoot提供的starter以spring-boot-starter-xxx的方式命名的。官方建议自定义的starter使用xxx-spring-boot-starter命名规则。以区分SpringBoot生态提供的starter。

示例走起

1. 新建工程，命名为 demo-spring-boot-starter
2. 配置pom依赖

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
```

```

<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.2.4.RELEASE</version>
  <relativePath/> <!-- Lookup parent from repository -->
</parent>
<groupId>com.zhuawa</groupId>
<artifactId>demo-spring-boot-starter</artifactId>
<version>0.0.1-SNAPSHOT</version>
<name>demo-spring-boot-starter</name>
<description>Demo project for Spring Boot</description>

<properties>
  <java.version>1.8</java.version>
</properties>

<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter</artifactId>
  </dependency>

  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-configuration-processor</artifactId>
    <optional>true</optional>
  </dependency>
  <dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <optional>true</optional>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
    <exclusions>
      <exclusion>
        <groupId>org.junit.vintage</groupId>
        <artifactId>junit-vintage-engine</artifactId>
      </exclusion>
    </exclusions>
  </dependency>
</dependencies>
</project>

```

3. 定义一个实体类映射配置信息 @ConfigurationProperties(prefix = "demo") 它可以把相同前缀的配置信息通过配置项名称映射成实体类，比如我们这里指定 prefix = "demo" 这样，我们就能将以demo为前缀的配置项拿到了。 ps: 其实这个注解很强大，它不但能映射成String或基本类型的变量。还可以映射为List， Map等数据结构。

```

/**
 * 配置信息 实体
 * @author zhibai
 * @Date 2020/2/16 00:34

```

```

*/
@ConfigurationProperties(prefix = "demo")
public class DemoProperties {
    private String sayWhat;
    private String toWho;

    public String getSayWhat() {
        return sayWhat;
    }

    public void setSayWhat(String sayWhat) {
        this.sayWhat = sayWhat;
    }

    public String getToWho() {
        return toWho;
    }

    public void setToWho(String toWho) {
        this.toWho = toWho;
    }
}

```

4. 定义一个Service

```

/**
 * 随便定义一个Service
 * @author zhibai
 */
public class DemoService {
    public String sayWhat;
    public String toWho;
    public DemoService(String sayWhat, String toWho){
        this.sayWhat = sayWhat;
        this.toWho = toWho;
    }
    public String say(){
        return this.sayWhat + "! " + toWho;
    }
}

```

5. 定义一个配置类

这里，我们将DemoService类定义为一个Bean，交给ioc容器。

▲ @Configuration 注解就不多说了。

▲ @EnableConfigurationProperties 注解。该注解是用来开启对3步骤中 @ConfigurationProperties 注解配置Bean的支持。也就是@EnableConfigurationProperties注解告诉Spring Boot 能支持@ConfigurationProperties。

当然了，也可以在 @ConfigurationProperties 注解的类上添加 @Configuration 或者 @Component 注解

▲ @ConditionalOnProperty 注解控制 @Configuration 是否生效。简单来说也就是我们可以通过在yml配置文件中控制 @Configuration 注解的配置类是否生效。

```
/**
 * 配置类
 * @author zhibai
 */
@Configuration
@EnableConfigurationProperties(DemoProperties.class)
@ConditionalOnProperty(
    prefix = "demo",
    name = "isopen",
    havingValue = "true"
)
public class DemoConfig {

    @Autowired
    private DemoProperties demoProperties;

    @Bean(name = "demo")
    public DemoService demoService(){
        return new DemoService(demoProperties.getSayWhat(), demoProperties.getToWho());
    }
}
```

6. 新建META-INF文件夹，然后创建spring.factories文件 在该文件中加入如下配置，该配置指定上步骤中定义的配置类为自动装配的配置

```
#-----starter自动装配-----
org.springframework.boot.autoconfigure.EnableAutoConfiguration=com.zhuawa.demo.config.DemoConfig
```

7. 打包验证 在demo-spring-boot-starter工程中执行mvn clean install 一个自定义的starter新鲜出炉

前面创建的测试项目中，引入starter依赖

```
<dependency>
    <groupId>com.zhuawa</groupId>
    <artifactId>demo-spring-boot-starter</artifactId>
    <version>0.0.1-SNAPSHOT</version>
</dependency>
```

修改配置文件，增加如下配置：

```
demo.isopen=true
demo.say-what=hello
demo.to-who=Spring Boot
```

写个测试类

```
/**
 * demo测试
 */
@RestController
public class DemoController {
    @Resource(name = "demo")
    private DemoService demoService;

    @GetMapping("/say")
    public String sayWhat() {
        return demoService.say();
    }
}
```

启动服务，打开浏览器，<http://localhost:8080/say> 可以看到：**hello! Spring Boot**

Spring Boot整合Quartz实现定时任务

1. 添加依赖

如果SpringBoot版本是2.0.0以后的，则在spring-boot-starter中已经包含了quartz的依赖，则可以直接使用spring-boot-starter-quartz依赖：

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

2. 创建任务类TestQuartz，该类主要是继承了QuartzJobBean

```
/**
 * Quartz定时调度测试类
 * @author zhibai
 */
public class TestQuartz extends QuartzJobBean {

    /**
     * 执行定时任务
     * @param jobExecutionContext
     * @throws JobExecutionException
     */
    @Override
    protected void executeInternal(JobExecutionContext jobExecutionContext) throws JobExecutionException {
        System.out.println("quartz task "+new Date())@Configuration
    }

    public class QuartzConfig {
        @Bean
        public JobDetail testQuartzDetail(){
            return
        }
    }
}
```

```

JobBuilder.newJob(TestQuartz.class).withIdentity("testQuartz").storeDurably().build();
    }
    @Bean
    public Trigger testQuartzTrigger(){
        SimpleScheduleBuilder scheduleBuilder = SimpleScheduleBuilder.simpleSchedule()
            .withIntervalInSeconds(10) // 设置时间周期单位秒
            .repeatForever();
        return TriggerBuilder.newTrigger().forJob(teatQuartzDetail())
            .withIdentity("testQuartz")
            .withSchedule(scheduleBuilder)
            .build();
    }
}
);
}
}

```

3. 创建配置类QuartzConfig

```

@Configuration
public class QuartzConfig {
    @Bean
    public JobDetail teatQuartzDetail(){
        return
JobBuilder.newJob(TestQuartz.class).withIdentity("testQuartz").storeDurably().build();
    }
    @Bean
    public Trigger testQuartzTrigger(){
        SimpleScheduleBuilder scheduleBuilder = SimpleScheduleBuilder.simpleSchedule()
            .withIntervalInSeconds(10) // 设置时间周期单位秒
            .repeatForever();
        return TriggerBuilder.newTrigger().forJob(teatQuartzDetail())
            .withIdentity("testQuartz")
            .withSchedule(scheduleBuilder)
            .build();
    }
}
}

```

4. 启动项目，可以看到

```

quartz task Sun Feb 16 15:43:19 CST 2020
quartz task Sun Feb 16 15:43:29 CST 2020
quartz task Sun Feb 16 15:43:39 CST 2020
quartz task Sun Feb 16 15:43:49 CST 2020

```

附录 cron表达式详解

一个cron表达式有至少6个（也可能7个）有空格分隔的时间元素。按顺序依次为：

- 秒 (0~59)
- 分钟 (0~59)

- 小时 (0~23)
- 天 (0~31)
- 月 (0~11)
- 星期 (1~7 1=SUN 或 SUN, MON, TUE, WED, THU, FRI, SAT)
- 年份 (1970 - 2099)

其中每个元素可以是一个值(如6),一个连续区间(9-12),一个间隔时间(8-18/4)(/表示每隔4小时),一个列表(1,3,5),通配符。由于“月份中的日期”和“星期中的日期”这两个元素互斥的,必须要对其中一个设置。

配置实例:

- (1) 0 0 2 1 * ? * 表示在每月的1日的凌晨2点调整任务
- (2) 0 15 10 ? * MON-FRI 表示周一到周五每天上午10:15执行作业
- (3) 0 15 10 ? 6L 2002-2006 表示2002-2006年的每个月的最后一个星期五上午10:15执行作
- (4) 0 0 10,14,16 * * ? 每天上午10点, 下午2点, 4点
- (5) 0 0/30 9-17 * * ? 朝九晚五工作时间内每半小时
- (6) 0 0 12 ? * WED 表示每个星期三中午12点
- (7) 0 0 12 * * ? 每天中午12点触发
- (8) 0 15 10 ? * * 每天上午10:15触发
- (9) 0 15 10 * * ? 每天上午10:15触发

Q&A答疑
