



Spring Boot

核心编程



• KimmKing •

目录 | Contents

1. 为什么需要Spring Boot
2. 使用 Spring Boot 快速构建项目
3. 注解驱动与约定大于配置开发模式
4. Spring Boot 项目属性配置
5. 自动配置机制和装配方法
6. 自定义Spring Boot Starter
7. Spring Boot条件化自动装配
8. Spring Boot 数据库基本操作与事务
9. Spring Boot 与常用框架技术整合
10. Spring Boot 拓展进阶
11. 实战：基于Spring的工程在Spring Boot下的重构



一、为什么需要Spring Boot

Spring臃肿以后的必然选择。

一切都是为了简化。

- 让开发变简单：
- 让配置变简单：
- 让运行变简单：

怎么变简单？关键词：整合

就像是SSH、SSM，国产的SpringSide

基于什么变简单：约定大于配置

一、为什么需要Spring Boot

Spring Boot 的历史

2012 年 10 月，Mike Youngstrom 在 Spring jira 中创建了一个功能请求，要求在 Spring 框架中支持无容器 Web 应用程序体系结构。他谈到了在主容器引导 spring 容器内配置 Web 容器服务：

我认为 Spring 的 Web 应用体系结构可以大大简化，如果它提供了从上到下利用 Spring 组件和配置模型的工具和参考体系结构。在简单的 main 方法引导的 Spring 容器内嵌入和统一这些常用 Web 容器服务的配置。

这一要求促使了 2013 年初开始的 Spring Boot 项目的研发。2014 年 4 月，Spring Boot 1.0.0 发布。从那以后，一些 Spring Boot 版本出来了：

Spring boot 1.1 (2014 年 6 月) - 改进的模板支持，gemfire 支持，elasticsearch 和 apache solr 的自动配置。

Spring Boot 1.2 (2015 年 3 月) - 升级到 servlet 3.1 / tomcat 8 / jetty 9, spring 4.1 升级，支持 banner / jms / SpringApplication 注解。

Spring Boot 1.3 (2016 年 12 月) - Spring 4.2 升级，新的 spring-boot-devtools，用于缓存技术 (ehcache, hazelcast, redis 和 infinispan) 的自动配置以及完全可执行的 jar 支持。

一、为什么需要Spring Boot

Spring Boot 的历史

Spring boot 1.4 (2017年1月) - spring 4.3 升级, 支持 couchbase / neo4j, 分析启动失败和 RestTemplateBuilder。

Spring boot 1.5 (2017年2月) - 支持 kafka / ldap, 第三方库升级, 弃用 CRaSH 支持和执行器记录器端点以动态修改应用程序日志级别。

Spring boot 2.0 (2018 年 03 月) -基于 Java 8, 支持 Java 9, 支持 Quartz , 调度程序大大简化了安全自动配置, 支持嵌入式 Netty

Spring Boot 简单性使 java 开发人员能够快速大规模地采用该项目。Spring Boot 可以说是在 Java 中开发基于 REST 的微服务 Web 应用程序的最快方法之一。。

一、为什么需要Spring Boot

为什么能做到简化：

- 1、Spring本身技术的成熟与完善，各方面第三方组件的成熟集成
- 2、Spring团队在去web容器化等方面的努力
- 3、基于MAVEN与POM的Java生态体系，整合POM模板成为可能
- 4、避免大量maven导入和各种版本冲突

Spring Boot 是 Spring 的一套快速配置脚手架

一、为什么需要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 minimal Spring configuration.

Features

Create stand-alone Spring applications

Embed Tomcat, Jetty or Undertow directly (no need to deploy WAR files)


Provide opinionated 'starter' dependencies to simplify your build configuration

Automatically configure Spring and 3rd party libraries whenever possible

Provide production-ready features such as metrics, health checks, and externalized configuration

Absolutely no code generation and no requirement for XML configuration

二、使用 Spring Boot 快速构建项目

 **spring** initializr

Project

- ☒ Maven Project
- ☐ Gradle Project

Language

- ☒ Java ☐ Kotlin
- ☐ Groovy

Spring Boot

- ☐ 2.4.0 (SNAPSHOT) ☐ 2.4.0 (M4) ☐ 2.3.5 (SNAPSHOT)
- ☒ 2.3.4 ☐ 2.2.11 (SNAPSHOT) ☐ 2.2.10 ☐ 2.1.18 (SNAPSHOT)
- ☐ 2.1.17

Project Metadata

Group

Artifact

Name



Description

Package name

Dependencies

ADD DEPENDENCIES... CTRL + B

No dependency selected



GENERATE CTRL + G

EXPLORE CTRL + SPACE

SHARE...

二、使用 Spring Boot 快速构建项目

展示一个最简单的Spring Boot实例

Hello, Spring Boot!

三、注解驱动与约定大于配置开发模式

Spring MVC

```
<!-- 注解驱动 -->
```

```
<context:component-scan base-package="io.kimmking"/>
```

```
<mvc:annotation-driven/>
```

Spring Boot: 只有注解是不够的, 还需要有条件判断, why?

- spring.factories
- @KKConfiguration
- @KKEnableConfiguration

@SpringBootApplication

@EnableAutoConfiguration

@ComponentScan

三、注解驱动与约定大于配置开发模式

为什么要约定大于配置？

举例来说，JVM有1000多个参数，但是我们不需要一个参数，就能java Hello。

优势在于，开箱即用：

- 一、Maven的目录结构：默认有resources文件夹存放配置文件。默认打包方式为jar。
- 二、默认的配置文件的 application.properties 或 application.yml 文件
- 三、默认通过 spring.profiles.active 属性来决定运行环境时的配置文件。
- 四、EnableAutoConfiguration 默认对于依赖的 starter 进行自动装载。
- 五、spring-boot-start-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。

三、注解驱动与约定大于配置开发模式

```
/**
 * @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 {
```

约定大于配置 (Convention Over Configuration) 也称作按约定编程是一种软件设计范式。

核心入口为: SpringBootApplication

三、注解驱动与约定大于配置开发模式

Project ▾

> Maven: org.jboss.spec.javax.resource:jboss-connector-

> Maven: org.jboss.spec.javax.transaction:jboss-transacti

> Maven: org.jboss:jboss-transaction-spi:7.6.0.Final

> Maven: org.mariadb.jdbc:mariadb-java-client:2.4.2

> Maven: org.mockito:mockito-core:2.7.21

> Maven: org.mockito:mockito-inline:2.7.21

> Maven: org.objenesis:objenesis:2.5

> Maven: org.ow2.asm:asm:4.2

> Maven: org.ow2.asm:asm:5.0.3

> Maven: org.postgresql:postgresql:42.2.5

> Maven: org.projectlombok:lombok:1.16.20

> Maven: org.skyscreamer:jsonassert:1.4.0

> Maven: org.slf4j:slf4j-over-slf4j:1.7.7

> Maven: org.slf4j:slf4j-to-slf4j:1.7.7

> Maven: org.slf4j:log4j-over-slf4j:1.7.26

> Maven: org.slf4j:slf4j-api:1.7.7

> Maven: org.springframework.boot:spring-boot:1.5.22

▼ Maven: org.springframework.boot:spring-boot-autococ

▼ spring-boot-autoconfigure-1.5.22.RELEASE.jar libra

▼ META-INF

▼ maven.org.springframework.boot.spring-booc

▼ pom.properties

▼ pom.xml

▼ additional-spring-configuration-metadata.js

▼ MANIFEST.MF

▼ spring.factories

▼ spring-autoconfigure-metadata.properties

▼ spring-configuration-metadata.json

▼ org.springframework.boot.autoconfigure

> admin

> amqp

> aop

> batch

> cache

> cassandra

> cloud

> condition

SpringBootApplication.java x spring.factories x

```
1 # Initializers
2 org.springframework.context.ApplicationContextInitializer=\
3 org.springframework.boot.autoconfigure.SharedMetadataReaderFactoryContextInitializer,\
4 org.springframework.boot.autoconfigure.logging.AutoConfigurationReportLoggingInitializer
5
6 # Application Listeners
7 org.springframework.context.ApplicationListener=\
8 org.springframework.boot.autoconfigure.BackgroundPreinitializer
9
10 # Auto Configuration Import Listeners
11 org.springframework.boot.autoconfigure.AutoConfigurationImportListener=\
12 org.springframework.boot.autoconfigure.condition.ConditionEvaluationReportAutoConfigurationImportListener
13
14 # Auto Configuration Import Filters
15 org.springframework.boot.autoconfigure.AutoConfigurationImportFilter=\
16 org.springframework.boot.autoconfigure.condition.OnClassCondition
17
18 # Auto Configure
19 org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
20 org.springframework.boot.autoconfigure.admin.SpringApplicationAdminJmxAutoConfiguration,\
21 org.springframework.boot.autoconfigure.aop.AopAutoConfiguration,\
22 org.springframework.boot.autoconfigure.amqp.RabbitAutoConfiguration,\
23 org.springframework.boot.autoconfigure.batch.BatchAutoConfiguration,\
24 org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration,\
25 org.springframework.boot.autoconfigure.cassandra.CassandraAutoConfiguration,\
26 org.springframework.boot.autoconfigure.cloud.CloudAutoConfiguration,\
27 org.springframework.boot.autoconfigure.context.ConfigurationPropertiesAutoConfiguration,\
28 org.springframework.boot.autoconfigure.context.MessageSourceAutoConfiguration,\
29 org.springframework.boot.autoconfigure.context.PropertyPlaceholderAutoConfiguration,\
30 org.springframework.boot.autoconfigure.couchbase.CouchbaseAutoConfiguration,\
31 org.springframework.boot.autoconfigure.dao.PersistenceExceptionTranslationAutoConfiguration,\
32 org.springframework.boot.autoconfigure.data.cassandra.CassandraDataAutoConfiguration,\
33 org.springframework.boot.autoconfigure.data.cassandra.CassandraRepositoriesAutoConfiguration,\
34 org.springframework.boot.autoconfigure.data.couchbase.CouchbaseDataAutoConfiguration,\
35 org.springframework.boot.autoconfigure.data.couchbase.CouchbaseRepositoriesAutoConfiguration,\
36 org.springframework.boot.autoconfigure.data.elasticsearch.ElasticsearchAutoConfiguration,\
37 org.springframework.boot.autoconfigure.data.elasticsearch.ElasticsearchDataAutoConfiguration,\
38 org.springframework.boot.autoconfigure.data.elasticsearch.ElasticsearchRepositoriesAutoConfiguration,\
39 org.springframework.boot.autoconfigure.data.jpa.JpaRepositoriesAutoConfiguration,\
40 org.springframework.boot.autoconfigure.data.ldap.LdapDataAutoConfiguration,\
41 org.springframework.boot.autoconfigure.data.ldap.LdapRepositoriesAutoConfiguration,\
42 org.springframework.boot.autoconfigure.data.mongo.MongoDataAutoConfiguration,\
43 org.springframework.boot.autoconfigure.data.mongo.MongoRepositoriesAutoConfiguration,\
44 org.springframework.boot.autoconfigure.data.neo4j.Neo4jDataAutoConfiguration,\
45 org.springframework.boot.autoconfigure.data.neo4j.Neo4jRepositoriesAutoConfiguration,\
46 org.springframework.boot.autoconfigure.data.solr.SolrRepositoriesAutoConfiguration,\
```

三、注解驱动与约定大于配置开发模式

展示一个Configuration配置的Spring Boot实例

四、Spring Boot 项目属性配置

#application.yml:

spring:

profiles:

active: dev

#application-dev.yml:

server:

port: 8080

size: B

content: "Size: \${size}"

girl:

size: C

#application-test.yml:

#application-prod.yml:

@Component//表示为组件

@ConfigurationProperties(prefix = "girl")

//对应的就是配置文件中的girl

public class GirlProperties {

private String size;

}

四、Spring Boot 项目属性配置

总结:

基于profile分组做环境隔离

基于前缀做组内覆盖

与Spring properties相同, 与maven profile类似

与maven profile的区别?

运行期间确定 vs 打包期间确定

可以结合Spring Cloud的配置中心使用

四、Spring Boot 项目属性配置

演示项目属性配置的实例。

以三个环境演示

dev

test

prod

五、自动配置机制和装配方法

Spring Boot 自动配置注解

- 自动装配与@SpringBootApplication
- @EnableAutoConfiguration
- 自动装配原理、创建自动配置类
- 自动装配Class命名的潜规则
- 自动装配package命名的潜规则

Spring Boot 启动过程

- 启动涉及的类
- 启动步骤

五、自动配置机制和装配方法

Spring Boot 自动配置注解

- `@SpringBootApplication`
- SpringBoot应用标注在某个类上说明这个类是SpringBoot的主配置类，SpringBoot就会运行这个类的main方法来启动SpringBoot项目。
- `@SpringBootConfiguration`
- `@EnableAutoConfiguration`
- `@AutoConfigurationPackage`
- `@Import({AutoConfigurationImportSelector.class})`
- 加载所有META-INF/spring.factories中存在的配置类（类似SpringMVC中加载所有converter）

五、自动配置机制和装配方法

自动配置机制

```
1  @Target(ElementType.TYPE)
2  @Retention(RetentionPolicy.RUNTIME)
3  @Documented
4  @Inherited
5  @SpringBootConfiguration
6  @EnableAutoConfiguration
7  @ComponentScan(excludeFilters = {
8      @Filter(type = FilterType.CUSTOM, classes = TypeExcludeFilter.class),
9      @Filter(type = FilterType.CUSTOM, classes = AutoConfigurationExcludeFilter.class) })
10 public @interface SpringBootApplication {
11
12     @AliasFor(annotation = EnableAutoConfiguration.class)
13     Class<?>[] exclude() default {};
14     @AliasFor(annotation = EnableAutoConfiguration.class)
15     String[] excludeName() default {};
16     //根据包路径扫描
17     @AliasFor(annotation = ComponentScan.class, attribute = "basePackages")
18     String[] scanBasePackages() default {};
19     //直接根据class类扫描
20     @AliasFor(annotation = ComponentScan.class, attribute = "basePackageClasses")
21     Class<?>[] scanBasePackageClasses() default {};
22
23 }
```

五、自动配置机制和装配方法

自动配置机制

```
1 //从这里可以看出该类实现很多的xxxAware和DeferredImportSelector, 所有的aware都优先于selectImports
2 //方法执行, 也就是说selectImports方法最后执行, 那么在它执行的时候所有需要的资源都已经获取到了
3 public class AutoConfigurationImportSelector implements DeferredImportSelector, BeanClassLoaderAware {
4     ...
5     public String[] selectImports(AnnotationMetadata annotationMetadata) {
6         if (!this.isEnabled(annotationMetadata)) {
7             return NO_IMPORTS;
8         } else {
9             //1加载META-INF/spring-autoconfigure-metadata.properties文件
10             AutoConfigurationMetadata autoConfigurationMetadata = AutoConfigurationMetadataLoader.loadMetadata();
11             //2获取注解的属性及其值 (PS: 注解指的是@EnableAutoConfiguration注解)
12             AnnotationAttributes attributes = this.getAttributes(annotationMetadata);
13             //3.在classpath下所有的META-INF/spring.factories文件中查找org.springframework.boot.autoconfigure
14             List<String> configurations = this.getCandidateConfigurations(annotationMetadata, attributes);
15             //4.对上一步返回的List中的元素去重、排序
16             configurations = this.removeDuplicates(configurations);
17             //5.依据第2步中获取的属性值排除一些特定的类
18             Set<String> exclusions = this.getExclusions(annotationMetadata, attributes);
19             //6对上一步中所得到的List进行过滤, 过滤的依据是条件匹配。这里用到的过滤器是
20             //org.springframework.boot.autoconfigure.condition.OnClassCondition最终返回的是一个ConditionOutcome
21             //数组。(PS: 很多类都是依赖于其它的类的, 当有某个类时才会装配, 所以这次过滤的就是根据是否有某个
22             //class进而决定是否装配的。这些类所依赖的类都写在META-INF/spring-autoconfigure-metadata.properties
23             this.checkExcludedClasses(configurations, exclusions);
24             configurations.removeAll(exclusions);
25             configurations = this.filter(configurations, autoConfigurationMetadata);
26             this.fireAutoConfigurationImportEvents(configurations, exclusions);
27             return StringUtils.toStringArray(configurations);
28         }
29     }
30     protected List<String> getCandidateConfigurations(AnnotationMetadata metadata, AnnotationAttributes attributes) {
31         List<String> configurations = SpringFactoriesLoader.loadFactoryNames(this.getSpringFactoriesLoader(), this.getClassLoader());
32         Assert.notEmpty(configurations, "No auto configuration classes found in META-INF/spring.factories");
33         return configurations;
34     }
35 }
36
```

五、自动配置机制和装配方法

自动配置机制

```
1 public abstract class SpringFactoriesLoader {
2     public static final String FACTORIES_RESOURCE_LOCATION = "META-INF/spring.factories";
3     private static Map<String, List<String>> loadSpringFactories(@Nullable ClassLoader classLoader)
4         MultiValueMap<String, String> result = cache.get(classLoader);
5         if (result != null)
6             return result;
7         try {
8             Enumeration<URL> urls = (classLoader != null ?
9                 classLoader.getResources(FACTORIES_RESOURCE_LOCATION) :
10                 ClassLoader.getSystemResources(FACTORIES_RESOURCE_LOCATION));
11             result = new LinkedMultiValueMap<>();
12             while (urls.hasMoreElements()) {
13                 URL url = urls.nextElement();
14                 UrlResource resource = new UrlResource(url);
15                 Properties properties = PropertiesLoaderUtils.loadProperties(resource);
16                 for (Map.Entry<?, ?> entry : properties.entrySet()) {
17                     List<String> factoryClassNames = Arrays.asList(
18                         StringUtils.commaDelimitedListToStringArray((String) entry.getValue()));
19                     result.addAll((String) entry.getKey(), factoryClassNames);
20                 }
21             }
22             cache.put(classLoader, result);
23             return result;
24         }
25         catch (IOException ex) {
26             throw new IllegalArgumentException("Unable to load factories from location [" +
27                 FACTORIES_RESOURCE_LOCATION + "]", ex);
28         }
29     }
30     ...
}
```

五、自动配置机制和装配方法

创建自动配置类

1、将App标记为@EnableAutoConfiguration，并作为App.run方法的首参。

@EnableAutoConfiguration

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

五、自动配置机制和装配方法

创建自动配置类

2、WebConfiguration

@Configuration

```
public class WebConfiguration {
```

```
...
```

```
}
```


五、自动配置机制和装配方法

创建自动配置类

3、创建自动装配类WebAutoConfiguration, 并使用@Import导入WebConfiguration

```
import org.springframework.context.annotation.Configuration;  
import org.springframework.context.annotation.Import;
```

```
@Configuration
```

```
@Import(WebConfiguration.class)
```

```
public class WebAutoConfiguration {  
}
```

五、自动配置机制和装配方法

创建自动配置类

4、在项目src/main/resources目录下新建META-INF/spring.factories资源，并配置WebAutoConfiguration类：

#自动装配

```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\ncom.ebc.WebAutoConfiguration
```

五、自动配置机制和装配方法

- 自动装配命名的潜规则

自动装配Class的命名规则

SpringBoot内建的自动装配的class的名称模式均为*AutoConfiguration，按照该模式，我们自定义的自动装配Class均沿用该命名方式

自动装配Package的命名规则

`${group}.autoconfigure`

自定义Starter的命名规则

`${module}-spring-boot-starter`

五、自动配置机制和装配方法

总结

- 1.通过各种注解实现了类与类之间的依赖关系，容器在启动的时候Application.run，会调用EnableAutoConfigurationImportSelector.class的selectImports方法（其实是其父类的方法）
- 2.selectImports方法最终会调用SpringFactoriesLoader.loadFactoryNames方法来获取一个全面的常用BeanConfiguration列表
- 3.loadFactoryNames方法会读取FACTORIES_RESOURCE_LOCATION（也就是spring-boot-autoconfigure.jar 下面的spring.factories），获取到所有的Spring相关的Bean的全限定名ClassName，大概120多个
- 4.selectImports方法继续调用filter(configurations, autoConfigurationMetadata);这个时候会根据这些BeanConfiguration里面的条件，来一一筛选，最关键的是@ConditionalOnClass，这个条件注解会去classpath下查找，jar包里面是否有这个条件依赖类，所以必须有了相应的jar包，才有这些依赖类，才会生成IOC环境需要的一些默认配置Bean
- 5.最后把符合条件的BeanConfiguration注入默认的EnableConfigurationProperties类里面的属性值，并且注入到IOC环境当中

六、自定义Spring Boot Starter

一个完整的Spring Boot Starter可能包含以下组件：

autoconfigure模块：包含自动配置的代码

starter模块：提供对autoconfigure模块的依赖，以及一些其它的依赖

1. 命名

模块名称不能以spring-boot开头

如果你的starter提供了配置keys，那么请确保它们有唯一的命名空间。而且，不要用Spring Boot用到的命名空间（比如：server， management， spring 等等）

举个例子，假设你为“acme”创建了一个starter，那么你的auto-configure模块可以命名为acme-spring-boot-autoconfigure， starter模块可以命名为acme-spring-boot-starter。如果你只有一个模块包含这两部分，那么你可以命名为acme-spring-boot-starter。

六、自定义Spring Boot Starter

2. autoconfigure模块

建议在autoconfigure模块中包含下列依赖：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-autoconfigure-processor</artifactId>
    <optional>true</optional>
</dependency>
```

3. starter模块

事实上，starter是一个空jar。它唯一的目的是提供这个库所必须的依赖。

你的starter必须直接或间接引用核心的Spring Boot starter（spring-boot-starter）

六、自定义Spring Boot Starter

实例演示一个Hello-Starter

七、Spring Boot条件化自动装配

条件化自动装配

使用`conditional`注解可自由定制自动化装配

`@ConditionalOnBean`

`@ConditionalOnClass`

`@ConditionalOnMissingBean`

`@ConditionalOnProperty`

`@ConditionalOnResource`

`@ConditionalOnSingleCandidate`

`@ConditionalOnWebApplication`

七、Spring Boot条件化自动装配

自动化装配条件详解：

1. Class条件注解
2. Bean条件注解
3. 属性条件注解
4. Resource条件注解
5. Web应用条件注解

自动化装配设计有什么好处？

七、Spring Boot条件化自动装配

演示自动化装配的例子

七、Spring Boot 条件化自动装配

Spring Boot 最核心的 25 个注解

1、@SpringBootApplication

这是 Spring Boot 最最最核心的注解，用在 Spring Boot 主类上，标识这是一个 Spring Boot 应用，用来开启 Spring Boot 的各项能力。

其实这个注解就是 @SpringBootConfiguration、@EnableAutoConfiguration、@ComponentScan 这三个注解的组合，也可以用这三个注解来代替 @SpringBootApplication 注解。

2、@EnableAutoConfiguration

允许 Spring Boot 自动配置注解，开启这个注解之后，Spring Boot 就能根据当前类路径下的包或者类来配置 Spring Bean。

如：当前类路径下有 Mybatis 这个 JAR 包，MybatisAutoConfiguration 注解就能根据相关参数来配置 Mybatis 的各个 Spring Bean。

七、Spring Boot条件化自动装配

3、@Configuration

这是 Spring 3.0 添加的一个注解，用来代替 applicationContext.xml 配置文件，所有这个配置文件里面能做到的事情都可以通过这个注解所在类来进行注册。

4、@SpringBootConfiguration

这个注解就是 @Configuration 注解的变体，只是用来修饰是 Spring Boot 配置而已，或者可利于 Spring Boot 后续的扩展。

5、@ComponentScan

这是 Spring 3.1 添加的一个注解，用来代替配置文件中的 component-scan 配置，开启组件扫描，即自动扫描包路径下的 @Component 注解进行注册 bean 实例到 context 中。

七、Spring Boot条件化自动装配

6、@Conditional

这是 Spring 4.0 添加的新注解，用来标识一个 Spring Bean 或者 Configuration 配置文件，当满足指定的条件才开启配置。

7、@ConditionalOnBean

组合 @Conditional 注解，当容器中有指定的 Bean 才开启配置。

8、@ConditionalOnMissingBean

组合 @Conditional 注解，和 @ConditionalOnBean 注解相反，当容器中没有指定的 Bean 才开启配置。

9、@ConditionalOnClass

组合 @Conditional 注解，当容器中有指定的 Class 才开启配置。

七、Spring Boot 条件化自动装配

10、@ConditionalOnMissingClass

组合 @Conditional 注解，和 @ConditionalOnClass 注解相反，当容器中没有指定的 Class 才开启配置。

11、@ConditionalOnWebApplication

组合 @Conditional 注解，当前项目类型是 WEB 项目才开启配置。

当前项目有以下 3 种类型。

```
enum Type {    /**      * Any web application will match.      */    ANY,
    /**      * Only servlet-based web application will match.      */    SERVLET,
    /**      * Only reactive-based web application will match.      */    REACTIVE}
```

12、@ConditionalOnNotWebApplication

组合 @Conditional 注解，和 @ConditionalOnWebApplication 注解相反，当前项目类型不是 WEB 项目才开启配置。

七、Spring Boot条件化自动装配

13、@ConditionalOnProperty

组合 @Conditional 注解，当指定的属性有指定的值时才开启配置。

14、@ConditionalOnExpression

组合 @Conditional 注解，当 SpEL 表达式为 true 时才开启配置。

15、@ConditionalOnJava

组合 @Conditional 注解，当运行的 Java JVM 在指定的版本范围时才开启配置。

16、@ConditionalOnResource

组合 @Conditional 注解，当类路径下有指定的资源才开启配置。

17、@ConditionalOnJndi

组合 @Conditional 注解，当指定的 JNDI 存在时才开启配置。

七、Spring Boot条件化自动装配

18、@ConditionalOnCloudPlatform

组合 @Conditional 注解，当指定的云平台激活时才开启配置。

19、@ConditionalOnSingleCandidate

组合 @Conditional 注解，当指定的 class 在容器中只有一个 Bean，或者同时有多个但为首选时才开启配置。

20、@ConfigurationProperties

用来加载额外的配置（如 .properties 文件），可用在 @Configuration 注解类，或者@Bean 注解方法上面。

关于这个注解的用法可以参考《Spring Boot读取配置的几种方式》这篇文章。

21、@EnableConfigurationProperties

一般要配合 @ConfigurationProperties 注解使用，用来开启对@ConfigurationProperties 注解配置 Bean 的支持。

七、Spring Boot条件化自动装配

22、@AutoConfigureAfter

用在自动配置类上面，表示该自动配置类需要在另外指定的自动配置类配置完之后。

如 Mybatis 的自动配置类，需要在数据源自动配置类之后。

23、@AutoConfigureBefore

这个和 @AutoConfigureAfter 注解使用相反，表示该自动配置类需要在另外指定的自动配置类配置之前。

24、@Import

这是 Spring 3.0 添加的新注解，用来导入一个或者多个 @Configuration 注解修饰的类。

25、@ImportResource

这是 Spring 3.0 添加的新注解，用来导入一个或者多个 Spring 配置文件，这对 Spring Boot 兼容老项目非常有用，因为有些配置无法通过 Java Config 的形式来配置就只能用这个注解来导入。

八、Spring Boot 数据库操作与事务

演示操作Spring Boot与JPA/Hibernate:

准备环境与配置依赖

Configuration与配置文件

实体类与服务类

启动类

访问测试

八、Spring Boot 数据库操作与事务

演示操作Spring Boot与MyBatis:

准备环境与配置依赖

Configuration与配置文件

Pojo、Mapper与服务类

启动类

访问测试

九、Spring Boot 与常用框架技术整合

spring boot与activemq整合

九、Spring Boot 与常用框架技术整合

更多参见Spring Cloud

十、Spring Boot 拓展进阶

启动Spring Boot项目时传递参数，有三种参数形式：

选项参数

非选项参数

系统参数

ApplicationArguments解析

十、Spring Boot 拓展进阶

Spring boot Actuator监控功能

ID	Description	Enabled by default
<code>auditevents</code>	Exposes audit events information for the current application.	Yes
<code>beans</code>	Displays a complete list of all the Spring beans in your application.	Yes
<code>conditions</code>	Shows the conditions that were evaluated on configuration and auto-configuration classes and the reasons why they did or did not match.	Yes
<code>configprops</code>	Displays a collated list of all <code>@ConfigurationProperties</code> .	Yes
<code>env</code>	Exposes properties from Spring's <code>ConfigurableEnvironment</code> .	Yes
<code>flyway</code>	Shows any Flyway database migrations that have been applied.	Yes
<code>health</code>	Shows application health information.	Yes
<code>httptrace</code>	Displays HTTP trace information (by default, the last 100 HTTP request-response exchanges).	Yes
<code>info</code>	Displays arbitrary application info.	Yes
<code>loggers</code>	Shows and modifies the configuration of loggers in the application.	Yes
<code>liquibase</code>	Shows any Liquibase database migrations that have been applied.	Yes
<code>metrics</code>	Shows 'metrics' information for the current application.	Yes
<code>mappings</code>	Displays a collated list of all <code>@RequestMapping</code> paths.	Yes
<code>scheduledtasks</code>	Displays the scheduled tasks in your application.	Yes
<code>sessions</code>	Allows retrieval and deletion of user sessions from a Spring Session-backed session store. Not available when using Spring Session's support for reactive web applications.	Yes
<code>shutdown</code>	Lets the application be gracefully shutdown.	No
<code>threaddump</code>	Performs a thread dump.	Yes

<https://blog.csdn.net/laravelshao>

十、Spring Boot 拓展进阶

Spring Boot注意事项

2.0配置方式的变化

配置冲突，以DataSource为例

十、Spring Boot 拓展进阶

Spring Boot最佳实践

1. 代码结构:

避免使用默认包

如果创建的类没有声明包信息，则类会默认使用默认包，默认包使用在使用诸如@ComponentScan, @EntityScan, 及@SpringBootApplication时可能会引发特殊问题。

官方建议遵循java既有的命名约定规则，使用反转域名的方式命名包。例如，com.example.project.

2. 应用主类位置:

通常我们建议将主类放置于根路径下，注解@SpringBootApplication 通常放置于主类上，并且作为某些扫描的根路径。如JPA配置的Entity扫描等。

@SpringBootApplication注解包含 @EnableAutoConfiguration 和 @ComponentScan，可以单独配置，或者直接使用@SpringBootApplication 简化配置。

十、Spring Boot 拓展进阶

3. 配置类@Configuration:

Spring boot倾向使用基于java配置类的配置方式，建议使用主类作为主要的配置位置@Configuration。

4. 引入额外的配置类：不需要将所有的配置放到一个配置类中，可以通过使用@Import注解引入额外的配置类信息。当然@ComponentScan注解会扫描包含@Configuration注解的配置类。

5. 引入xml配置：如果存在不许使用xml配置的情况，则可以通过@ImportResource注解来进行加载。

6. 自动配置@EnableAutoConfiguration

Spring boot基于添加的相应的功能jar进行自动配置。例如，类路径中有HSQLDB jar包的情况下，如果没有主动定义相应的数据源连接bean，则spring boot会自动配置内存数据库。自动配置需添加相应的@EnableAutoConfiguration或者@SpringBootApplication来启用。通常放置其一于主类即可。

十、Spring Boot 拓展进阶

7. 自动配置的覆盖：

自动配置是非侵入性的，可以通过定义相应的自定义配置类进行覆盖，如果需要知道工程目前使用了那些自动配置，可以通过在启动时添加—debug选项，来进行输出。

8. 禁用某些自动配置

如果发现输出的日志中包含一些不需要应用的自动配置可以通过在注解@EnableAutoConfiguration上添加exclude附加选项来禁用。

9. maven打包后为fat jar：

```
mvn clean package
```

使用maven插件运行：

```
$ mvn spring-boot:run
```

十一、Spring->Spring Boot

实战：基于Spring的工程在Spring Boot下的重构

将交易系统改造成Spring Boot项目，

1. 向Spring Boot的演进，
2. 使用Spring Boot构建项目、属性配置，
3. 数据库操作和事务等。

课程总结

1. 为什么需要Spring Boot
2. 使用 Spring Boot 快速构建项目
3. 注解驱动与约定大于配置开发模式
4. Spring Boot 项目属性配置
5. 自动配置机制和装配方法
6. 自定义Spring Boot Starter
7. Spring Boot条件化自动装配
8. Spring Boot 数据库基本操作与事务
9. Spring Boot 与常用框架技术整合
10. Spring Boot 拓展进阶
11. 实战：基于Spring的工程在Spring Boot下的重构



THANKS! |  极客大学