

# Springboot-Notes

## 1. Springboot

可查看此文章[spring和springboot](#)

重点要理解微服务和分布式的概念

- 微服务
- 分布式

## 2. Springboot的官方文档

- 文档地址: <https://spring.io/projects/spring-boot#learn>

可自行选择版本, 建议选择稳定版

- 文档架构

The reference documentation consists of the following sections:

|   |   |
|---|---|
| <a href="#">Legal</a>                                 | Legal information.  |
| <a href="#">Documentation Overview</a>                | About the Documentation, Getting Help, First Steps, and more.   |
| <a href="#">Getting Started</a> 入门                    | Introducing Spring Boot, System Requirements, Servlet Containers, Installing Spring Boot, Developing Your First Spring Boot Application |
| <a href="#">Using Spring Boot</a> 进阶                  | Build Systems, Structuring Your Code, Configuration, Spring Beans and Dependency Injection, DevTools, and more.                         |
| <a href="#">Spring Boot Features</a> 高级特性             | Profiles, Logging, Security, Caching, Spring Integration, Testing, and more.  |
| <a href="#">Spring Boot Actuator</a> 监控               | Monitoring, Metrics, Auditing, and more.  |
| <a href="#">Deploying Spring Boot Applications</a> 部署 | Deploying to the Cloud, Installing as a Unix application.   |
| <a href="#">Spring Boot CLI</a>                       | Installing the CLI, Using the CLI, Configuring the CLI, and more.   |
| <a href="#">Build Tool Plugins</a>                    | Maven Plugin, Gradle Plugin, Antlib, and more.  |
| <a href="#">"How-to" Guides</a> 小技巧                   | Application Development, Configuration, Embedded Servers, Data Access, and many more.   |

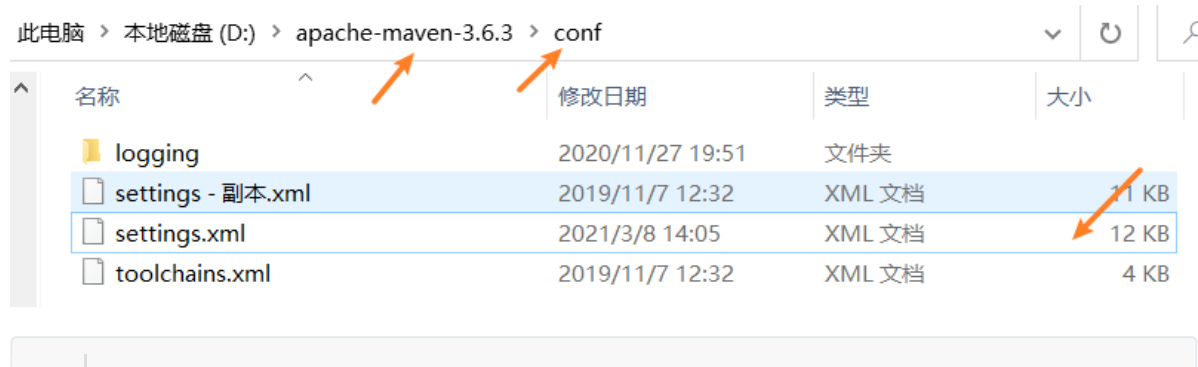
The reference documentation has the following appendices:

|  |   |
|--|---|
| <a href="#">Application Properties</a> 所有配置概览              | Common application properties that can be used to configure your application. |
| <a href="#">Configuration Metadata</a>                     | Metadata used to describe configuration properties.                           |
| <a href="#">Auto-configuration Classes</a> 所有自动配置          | Auto-configuration classes provided by Spring Boot.                           |
| <a href="#">Test Auto-configuration Annotations</a> 常见测试注解 | Test-autoconfiguration annotations used to test slices of your application.   |
| <a href="#">Executable Jars</a> 可执行jar                     | Spring Boot's executable jars, their launchers, and their format.             |
| <a href="#">Dependency Versions</a> 所有场景依赖版本               | Details of the dependencies that are managed by Spring Boot.                  |

## 3. springboot2系统要求

- jdk1.8+
- maven3.3+

需要在maven的conf目录下配置settings.xml文件。添加以下两个项, 避免错误



```

1 <mirrors>
2     <mirror>
3         <id>nexus-aliyun</id>
4         <mirrorOf>central</mirrorOf>
5         <name>Nexus aliyun</name>
6         <url>http://maven.aliyun.com/nexus/content/groups/public</url>
7     </mirror>
8 </mirrors>
9
10 <profiles>
11     <profile>
12         <id>jdk-1.8</id>
13         <activation>
14             <activeByDefault>true</activeByDefault>
15             <jdk>1.8</jdk>
16         </activation>
17         <properties>
18             <maven.compiler.source>1.8</maven.compiler.source>
19             <maven.compiler.target>1.8</maven.compiler.target>
20
21             <maven.compiler.compilerVersion>1.8</maven.compiler.compilerVersion>
22         </properties>
23     </profile>
24 </profiles>

```

## 4. Springboot快速入门

- 创建普通maven工程，引入springboot依赖

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0"
3         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
5         http://maven.apache.org/xsd/maven-4.0.0.xsd">
6     <modelVersion>4.0.0</modelVersion>
7
8     <groupId>com.hhj</groupId>
9     <artifactId>Springboot-learnNote</artifactId>
10    <version>1.0-SNAPSHOT</version>
11    <packaging>jar</packaging>
12
13    <!-- springboot项目的根依赖，包含了许多核心基础的包
14         使用parent标签，表示它的父依赖是这个，可以直接使用父亲的jar包-->
15    <parent>
16        <groupId>org.springframework.boot</groupId>
17        <artifactId>spring-boot-starter-parent</artifactId>
18        <version>2.3.4.RELEASE</version>
19    </parent>
20
21    <!-- 如果要创建web应用，需要引入此依赖-->
22    <!-- 该依赖中集成了tomcat等-->
23    <dependencies>
24        <dependency>
25            <groupId>org.springframework.boot</groupId>
26            <artifactId>spring-boot-starter-web</artifactId>
27        </dependency>

```

```

28
29     </dependencies>
30
31
32 </project>

```

- 创建主程序/入口

```

1 package com.hhj;
2
3 import org.springframework.boot.SpringApplication;
4 import org.springframework.boot.autoconfigure.SpringBootApplication;
5
6 // 标记这是一个springboot应用
7 @SpringBootApplication
8 public class MainApplication {
9     public static void main(String[] args) {
10         SpringApplication.run(MainApplication.class,args);
11     }
12 }

```

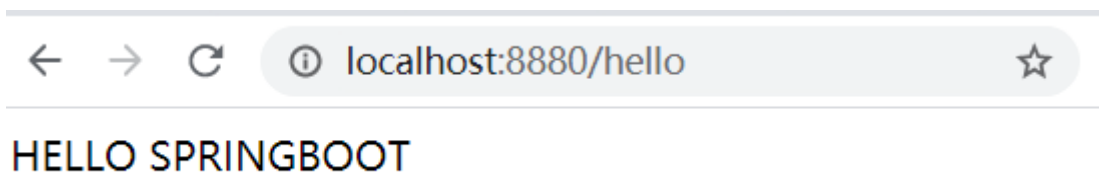
- 编写控制器

```

1 package com.hhj.controller;
2
3 import org.springframework.stereotype.Controller;
4 import org.springframework.web.bind.annotation.RequestMapping;
5 import org.springframework.web.bind.annotation.ResponseBody;
6 import org.springframework.web.bind.annotation.RestController;
7
8 @RestController
9 //@RestController=@ResponseBody+@Controller,表示当前类是一个控制器并且返回的
  数据直接输出给浏览器，不是页面跳转
10 public class HelloController {
11
12     @RequestMapping("/hello")
13     public String hello(){
14         return "HELLO SPRINGBOOT";
15     }
16 }

```

- 运行主程序



此外，Springboot还可以帮我们简化配置和部署

- 简化配置

创建application.properties文件，配置应用程序的公用属性。

上面提到的springboot的官方文档里面有详细说明，配置文件名字也是从此而来，不能改变。

[application.properties说明](#)

The reference documentation has the following appendices:

配置应用程序的公用属性

| Application Properties              | Common application properties that can be used to configure your application. |
|-------------------------------------|---|
| Configuration Metadata              | Metadata used to describe configuration properties.                           |
| Auto-configuration Classes          | Auto-configuration classes provided by Spring Boot.                           |
| Test Auto-configuration Annotations | Test-autoconfiguration annotations used to test slices of your application.   |
| Executable Jars                     | Spring Boot's executable jars, their launchers, and their format.             |
| Dependency Versions                 | Details of the dependencies that are managed by Spring Boot.                  |

```
1 #配置tomcat服务器端口号
2 server.port=8880
```

- 简化部署

以前ssm部署项目的原理是把项目打成war包，放到tomcat服务器上。springboot可以直接打成jar包，jar包中集成了tomcat服务器，因此可以在服务器本地使用 `java -jar` 命令直接运行

先添加maven插件

```
1 <build>
2   <plugins>
3     <plugin>
4       <groupId>org.springframework.boot</groupId>
5       <artifactId>spring-boot-maven-plugin</artifactId>
6     </plugin>
7   </plugins>
8 </build>
```

maven install打成jar包

命令行运行

```
PS D:\ideaProjects\Springboot-learnNote\target> java -jar .\Springboot-learnNote-1.0-SNAPSHOT.jar

Spring
:: Spring Boot :: (v2.3.4.RELEASE)

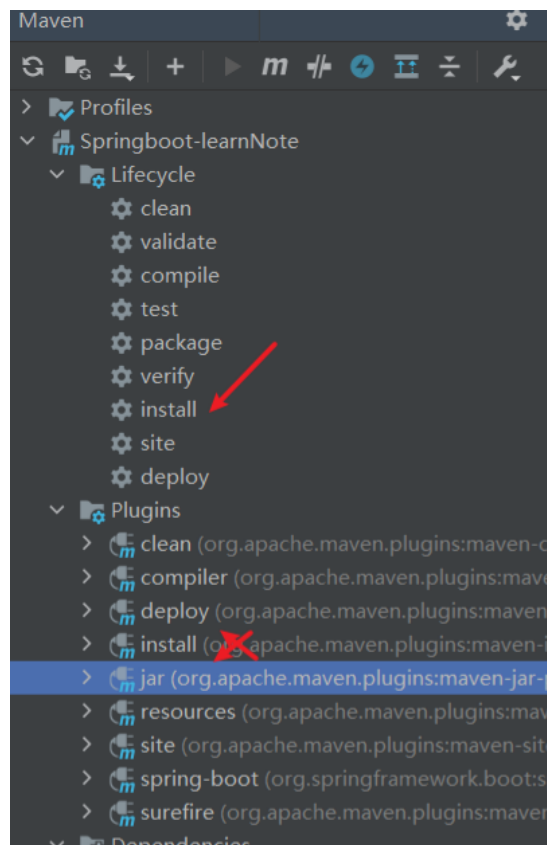
2021-03-08 15:30:16.217 INFO 17204 --- [main] com.hhj.MainApplication : Starting MainApplication v1.0-SNAPSHOT on DESKTOP-9FR1FRS with PID 17204 (D:\ideaProjects\Springboot-learnNote\target\Springboot-learnNote-1.0-SNAPSHOT.jar started by sang_Ma in D:\ideaProjects\Springboot-learnNote\target)
2021-03-08 15:30:16.219 INFO 17204 --- [main] com.hhj.MainApplication : No active profile set, falling back to default profiles: default
2021-03-08 15:30:17.557 INFO 17204 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 8880 (http)
2021-03-08 15:30:17.600 INFO 17204 --- [main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2021-03-08 15:30:17.601 INFO 17204 --- [main] org.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/9.0.38]
2021-03-08 15:30:17.671 INFO 17204 --- [main] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
2021-03-08 15:30:17.672 INFO 17204 --- [main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 1384 ms
2021-03-08 15:30:17.859 INFO 17204 --- [main] o.s.s.concurrent.ThreadPoolTaskExecutor : Initializing ExecutorService 'applicationTaskExecutor'
2021-03-08 15:30:18.060 INFO 17204 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8880 (http) with context path '/'
2021-03-08 15:30:18.075 INFO 17204 --- [main] com.hhj.MainApplication : Started MainApplication in 2.318 seconds (JVM running for 2.718)
2021-03-08 15:30:32.250 INFO 17204 --- [nio-8880-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring DispatcherServlet 'dispatcherServlet'
2021-03-08 15:30:32.250 INFO 17204 --- [nio-8880-exec-1] o.s.web.servlet.DispatcherServlet : Initializing Servlet 'dispatcherServlet'
2021-03-08 15:30:32.263 INFO 17204 --- [nio-8880-exec-1] o.s.web.servlet.DispatcherServlet : Completed initialization in 11 ms
```

出现的问题:

```
PS D:\ideaProjects\Springboot-learnNote\target> java -jar .\Springboot-learnNote-1.0-SNAPSHOT.jar
.\Springboot-learnNote-1.0-SNAPSHOT.jar中没有主清单属性
```

运行jar包部署服务器时出现此错误

原因是安装完插件的第一次打包要用上面的install，不能用下面的install



## 5. Springboot自动装配的原理

### ① Springboot是如何管理依赖的

前面在学习ssm的时候，要导入一大堆jar包，但是springboot只需要引入一个父依赖以及场景启动器就可以构建web程序，它是如何实现的呢？

我们先来看看springboot的父依赖

```
1 <parent>
2   <groupId>org.springframework.boot</groupId>
3   <artifactId>spring-boot-starter-parent</artifactId>
4   <version>2.3.4.RELEASE</version>
5 </parent>
```

1. 点击spring-boot-starter-parent，我们可以看到他也有父依赖

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-dependencies</artifactId>
  <version>2.3.4.RELEASE</version>
</parent>
```

2. 再点进去它的父依赖

```

<properties>
  <activemq.version>5.15.13</activemq.version>
  <antlr2.version>2.7.7</antlr2.version>
  <appengine-sdk.version>1.9.82</appengine-sdk.version>
  <artemis.version>2.12.0</artemis.version>
  <aspectj.version>1.9.6</aspectj.version>
  <assertj.version>3.16.1</assertj.version>
  <atomikos.version>4.0.6</atomikos.version>
  <awaitility.version>4.0.3</awaitility.version>
  <bitronix.version>2.1.4</bitronix.version>
  <build-helper-maven-plugin.version>3.1.0</build-helper-maven-plugin.version>
  <byte-buddy.version>1.10.14</byte-buddy.version>
  <caffeine.version>2.8.5</caffeine.version>
  <cassandra-driver.version>4.6.1</cassandra-driver.version>
  <classmate.version>1.5.1</classmate.version>
  <commons-codec.version>1.14</commons-codec.version>
  <commons-dbcp2.version>2.7.0</commons-dbcp2.version>
  <commons-lang3.version>3.10</commons-lang3.version>
  <commons-pool.version>1.6</commons-pool.version>
  <commons-pool2.version>2.8.1</commons-pool2.version>
  <couchbase-client.version>3.0.8</couchbase-client.version>
  <db2-jdbc.version>11.5.4.0</db2-jdbc.version>
  <dependency-management-plugin.version>1.0.10.RELEASE</dependency-management-plugin.version>
  <derby.version>10.14.2.0</derby.version>

```

我们可以看到这个坐标定义了很多属性，这些属性基本涵盖了我们需要使用的依赖的版本号。所以后面我们再导入其他场景启动器时不需要写版本号，因为版本号都定义在了这里

3. 所以说，spring-boot-starter-parent依赖并没有实际的jar包可以导入，它的作用是定义各组件的版本号

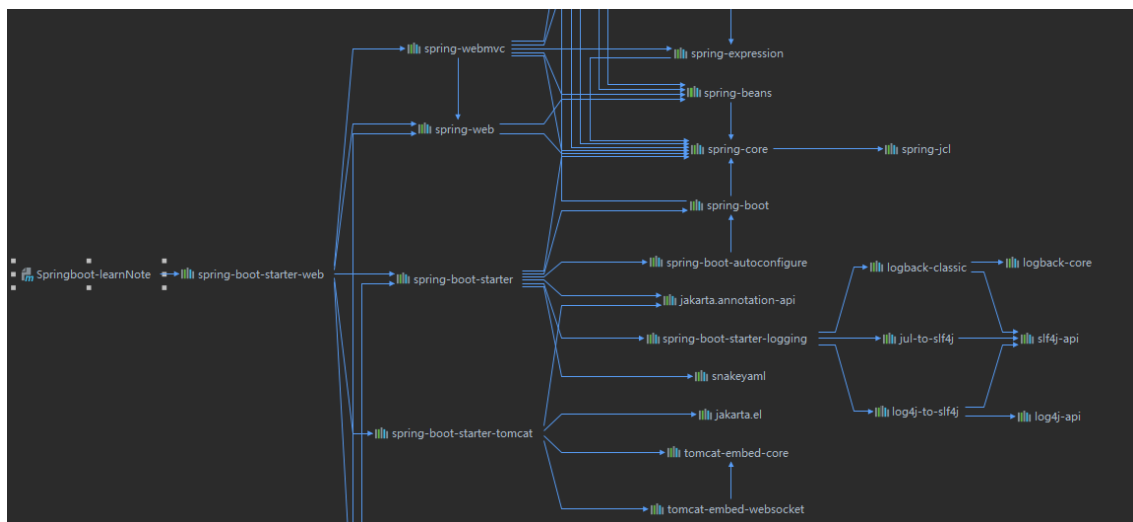
```

<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
  <version>2.3.4.RELEASE</version>
</dependency>

```

web场景启动器的版本号就定义在了这里

4. 看完spring-boot-starter-parent，我们接来看spring-boot-starter-web，首先来看我们这个项目的maven依赖图



这个图中没有关于spring-boot-starter-parent的信息，也侧面印证了我们上面的分析

5. 来看spring-boot-starter-web的代码

```

<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter</artifactId>
<version>2.3.4.RELEASE</version>
<scope>compile</scope>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-json</artifactId>
  <version>2.3.4.RELEASE</version>
  <scope>compile</scope>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-tomcat</artifactId>
  <version>2.3.4.RELEASE</version>
  <scope>compile</scope>
</dependency>
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-web</artifactId>
  <version>5.2.9.RELEASE</version>
  <scope>compile</scope>
</dependency>
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-webmvc</artifactId>
  <version>5.2.9.RELEASE</version>
  <scope>compile</scope>
</dependency>

```

可以看到web启动器集成了springmvc、tomcat、springcontext等

只要引入starter启动器，这个场景的所有常规需要的依赖我们都自动引入

SpringBoot所有支持的场景

<https://docs.spring.io/spring-boot/docs/current/reference/html/using-spring-boot.html#using-boot-starter>

见到的 `*-spring-boot-starter`： 第三方为我们提供的简化开发的场景启动器。

## 6. 如何修改默认的版本号

在当前项目的pom.xml文件中重写相应属性

```

1 1、查看spring-boot-dependencies里面规定当前依赖的版本 用的 key。
2 2、在当前项目里面重写配置
3 比如修改mysql的版本号
4 <properties>
5     <mysql.version>5.1.43</mysql.version>
6 </properties>

```

## ② 自动装配原理

1. Springboot程序是这样启动的...

```

1 @SpringBootApplication
2 public class MainApplication {
3     public static void main(String[] args) {
4         SpringApplication.run(MainApplication.class, args);
5     }
6 }

```

这个方法返回的实际上是一个IOC容器ApplicationContext，springboot根据我们配置的依赖pom.xml动态的为我们创建并填充IOC容器

我们先来看看这个IOC容器中装配了哪些bean

```

1 @SpringBootApplication
2 public class MainApplication {
3     public static void main(String[] args) {
4         ConfigurableApplicationContext application =
5 SpringApplication.run(MainApplication.class, args);
6         String[] beanDefinitionNames =
7 application.getBeanDefinitionNames();
8         for(String s:beanDefinitionNames) {
9             System.out.println(s);
10        }
11    }
12 }

```

```

1 org.springframework.context.annotation.internalConfigurationAnnotationPr
2 ocessor
3 org.springframework.context.annotation.internalAutowiredAnnotationProces
4 sor
5 org.springframework.context.annotation.internalCommonAnnotationProcessor
6 org.springframework.context.event.internalEventListenerProcessor
7 org.springframework.context.event.internalEventListenerFactory
8 mainApplication
9 org.springframework.boot.autoconfigure.internalCachingMetadataReaderFact
10 ory
11 helloController
12 org.springframework.boot.autoconfigure.AutoConfigurationPackages
13 org.springframework.boot.autoconfigure.context.PropertyPlaceholderAutoCo
14 nfiguration
15 propertySourcesPlaceholderConfigurer
16 org.springframework.boot.autoconfigure.websocket.servlet.WebSocketServle
17 tAutoConfiguration$TomcatWebSocketConfiguration
18 websocketServletWebServerCustomizer
19 org.springframework.boot.autoconfigure.websocket.servlet.WebSocketServle
20 tAutoConfiguration

```



```

15 org.springframework.boot.autoconfigure.web.servlet.ServletWebServerFacto
    ryConfiguration$EmbeddedTomcat
16 tomcatServletWebServerFactory
17 org.springframework.boot.autoconfigure.web.servlet.ServletWebServerFacto
    ryAutoConfiguration
18 servletWebServerFactoryCustomizer
19 tomcatServletWebServerFactoryCustomizer
20 org.springframework.boot.context.properties.ConfigurationPropertiesBindi
    ngPostProcessor
21 .....等等

```

可以看到我们在springmvc中用到的dispathServlet、viewResolver、multipartResolver以及我们的helloController都被添加到该容器供springboot调用，这也就是为什么我们什么都没有配置，也可以使用mvc的组件。springboot都帮我们自动装配好了所有web开发常用的组件。

## 2. 还有一个问题：springboot是如何知道扫描哪些包的

前面学习spring以及springmvc的时候，都要使用一个 `<context:component-scan base-package="com.hhj"/>` 来决定扫描哪些包，springboot又是如何实现的呢

springboot有默认包扫描规则，默认扫描主程序所在包及其下面的所有子包

也可以修改默认扫描包

- 通过 `@SpringBootApplication(scanBasePackages = "com.hhj")`
- 通过 `@ComponentScan(basePackages = "com.hhj")` 加上另外两个注解替代 `@SpringBootApplication`

```

1 @SpringBootApplication
2 等同于
3 @SpringBootConfiguration
4 @EnableAutoConfiguration
5 @ComponentScan("com.atguigu.boot")

```

实际上，配置文件中的值最终都是映射到IOC容器中的某一个对象上的，springboot按需加载对象

## 6. 底层常用注解

### ① @Configuration

- 标记当前类为配置类，用法和spring中的该注解一样
- 有一个重要属性 `@Configuration(proxyBeanMethods = true)`，不写默认为true

proxyBeanMethods表示@Bean注解下调用该方法的方法是

- Full模式(proxyBeanMethods = true)【保证每个@Bean方法被调用多少次返回的组件都是单实例的】
- Lite模式(proxyBeanMethods = false)【每个@Bean方法被调用多少次返回的组件都是新创建的】

一般情况下要调成false，因为如果是true的话，spring每次向IOC中添加该bean前都会检查以下IOC容器中有没有该bean，如何有，返回容器中的bean，不重新创建。这样虽然保证了单例，但是会降低Springboot的性能。

但是如果组件内部有依赖则必须使用Full模式，可以保证是同一个bean。其他默认是否Lite模式。

类似的添加bean的注解还有@Bean、@Compone、@Controller、@Service、@Respository

## ② @Import

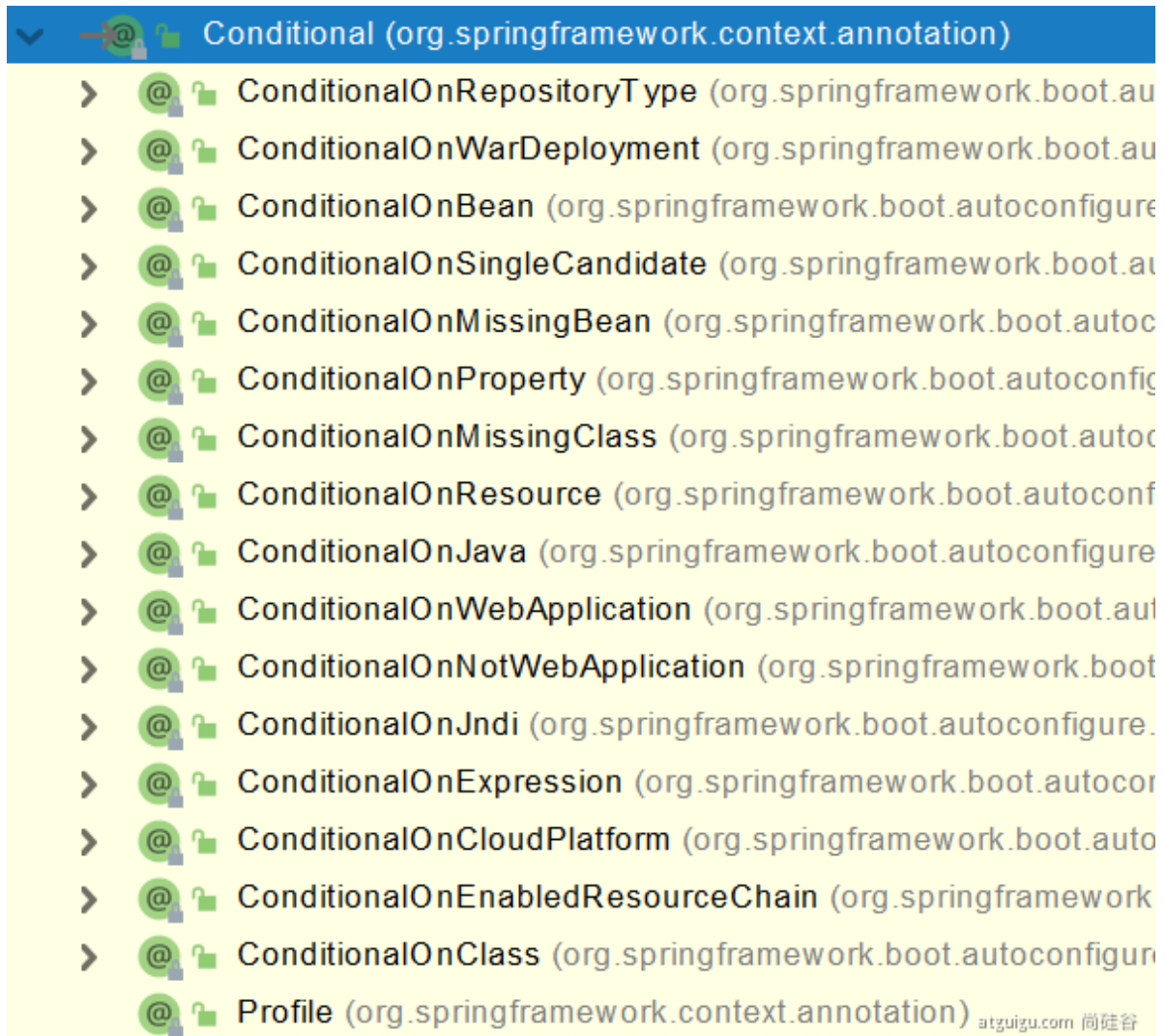
- 只能再类上进行注解 `@Import({User.class,DBHelper.class})`

作用是使用无参构造函数在IOC上创建这两个类的bean，bean的默认名字就是全类名

## ③ @Conditional

这是条件装配注解，只能满足Conditional指定的条件，采用注入IOC容器

它的实现注解：



- 可以用在类上，表示当前类的所有@Bean注解下的方法都有此条件
- 也可以用在@Bean方法上，仅表示但钱方法有限制

```
1 @Bean
2 @ConditionalOnBean(name = "user")
3 public Dog dog(){
4     return new Dog("tom",5);
5 }
```

## ④ @ImportResource

这个注解的作用和xml注解中的component scan一样，都是实现注解配置和xml配置的结合

使用该注解可以将路径下的spring配置文件中的bean添加到同一个IOC容器

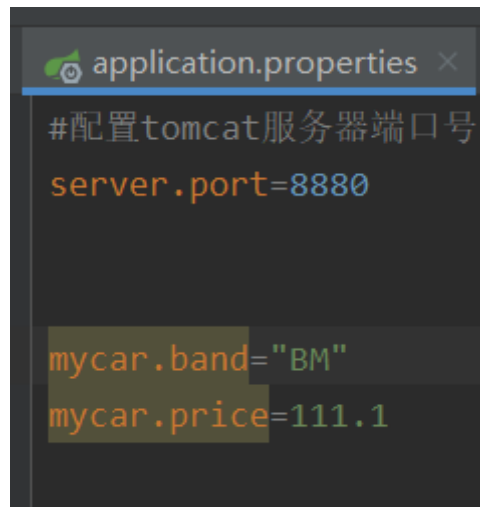
`@ImportResource("classpath:spring.xml")`，一般用于配置类

## ⑤ 关于绑定根配置文件的注解

把springboot的properties配置文件的信息绑定到JavaBean实体类

有两种种实现方式

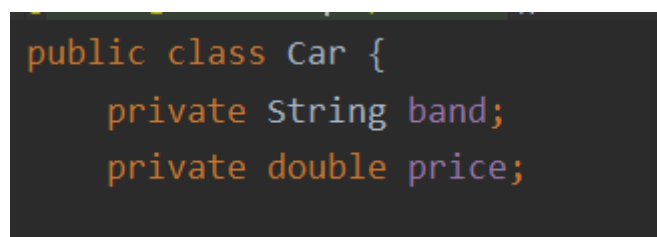
配置文件：



```
application.properties ×
#配置tomcat服务器端口号
server.port=8880

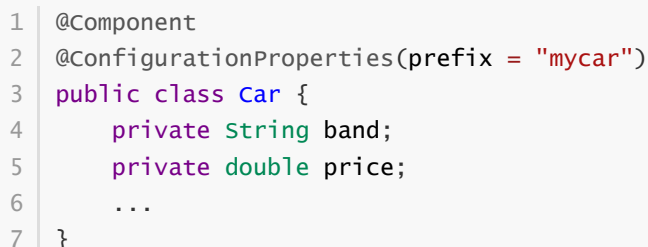
mycar.band="BM"
mycar.price=111.1
```

实体类：



```
public class Car {
    private String band;
    private double price;
```

- 在要绑定的实体类上使用@ConfigurationProperties



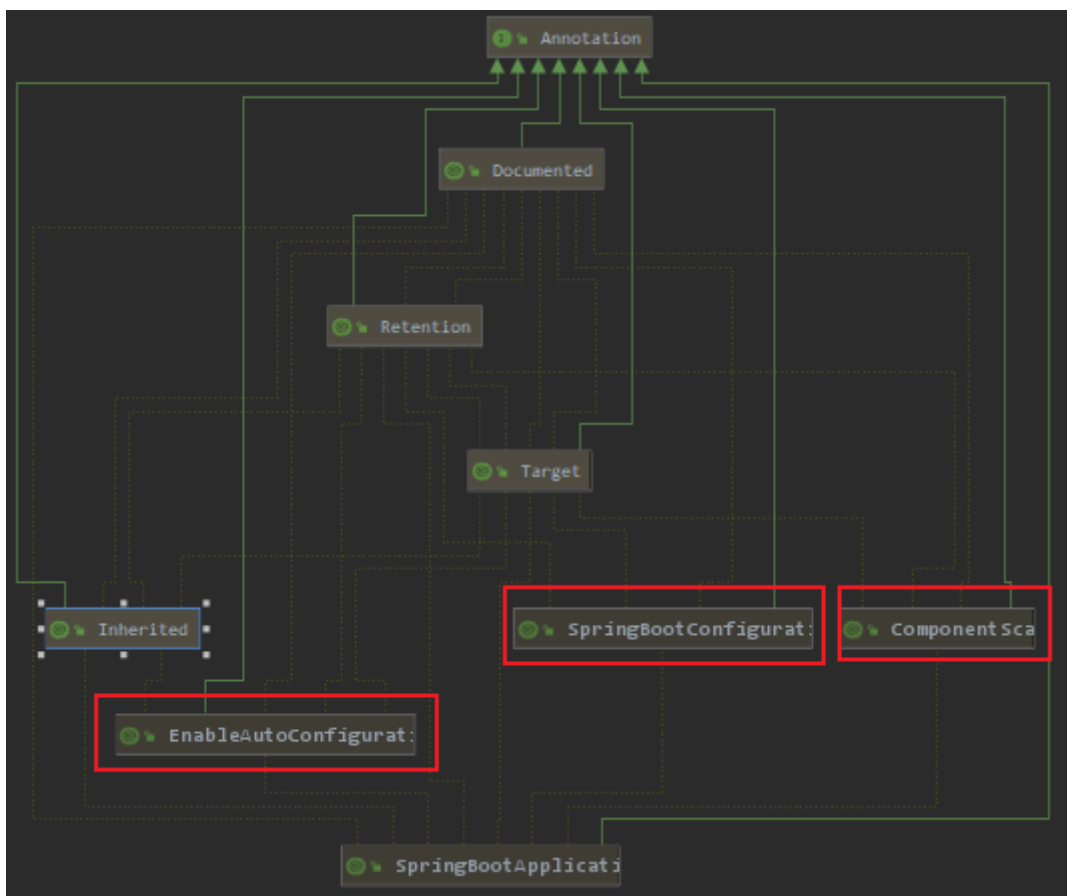
```
1 @Component
2 @ConfigurationProperties(prefix = "mycar")
3 public class Car {
4     private String band;
5     private double price;
6     ...
7 }
```

- 配置类@EnableConfigurationProperties+实体类@ConfigurationProperties

## 7. 深入了解Springboot自动装配的原理

springboot应用的入口就是@SpringBootApplication所在的类，所以说这个注解到底在底层为我们做了什么？

- 首先看他的架构图



核心就是其他三个注解

```

1  @SpringBootConfiguration
2  @EnableAutoConfiguration
3  @ComponentScan(
4      excludeFilters = {@Filter(
5          type = FilterType.CUSTOM,
6          classes = {TypeExcludeFilter.class}
7      )}, @Filter(
8          type = FilterType.CUSTOM,
9          classes = {AutoConfigurationExcludeFilter.class}
10 )}
11 )

```

- 分析@SpringBootConfiguration

```

1  @Configuration
2  public @interface SpringBootConfiguration {
3      @AliasFor(
4          annotation = Configuration.class
5      )
6      boolean proxyBeanMethods() default true;
7  }

```

表示当前类是一个spring配置类，没有涉及自动装配

- 分析@ComponentScan

这个说过很多次了，指定扫描的包

- 分析@EnableAutoConfiguration

同样是一个合成注解

```

1 @AutoConfigurationPackage
2 @Import({AutoConfigurationImportSelector.class})
3 public @interface EnableAutoConfiguration {
4     String ENABLED_OVERRIDE_PROPERTY =
5         "spring.boot.enableautoconfiguration";
6
7     Class<?>[] exclude() default {};
8
9     String[] excludeName() default {};
10 }

```

- 分析@AutoConfigurationPackage

```

@Import({Registrar.class})
public @interface AutoConfigurationPackage {
    String[] basePackages() default {};

    Class<?>[] basePackageClasses() default {};
}

```

看来这个Registrar类就是关键了，看看这个类的源码并调试

```

public void registerBeanDefinitions(AnnotationMetadata metadata, BeanDefinitionRegistry registry) {
    metadata: StandardAnnotationMetadata@3584 registry: "org.springframework.boot.autoconfigure"
    AutoConfigurationPackages.register(registry, (String[])(new AutoConfigurationPackages.PackageImports(metadata)).getPackageNames().toArray(new String[0]));
}

```

使用debug可以看到metadata就是我们的mainapplication

Expression: `new AutoConfigurationPackages.PackageImports(metadata)`

Result:

- result = {Collections\$UnmodifiableRandomAccessList@3999} size = 1
- 0 = "com.hhj"
- value = {char[7]@4004}
- hash = 0

springboot通过metadata信息得到主类所在的包名，封装到数组后进行传参。这也是自动扫描默认包的底层实现

- 分析 @Import({AutoConfigurationImportSelector.class})

```

public String[] selectImports(AnnotationMetadata annotationMetadata) {
    if (!this.isEnabled(annotationMetadata)) {
        return NO_IMPORTS;
    } else {
        AutoConfigurationImportSelector.AutoConfigurationEntry autoConfigurationEntry = this.getAutoConfigurationEntry(annotationMetadata);
        return StringUtils.toStringArray(autoConfigurationEntry.getConfigurations());
    }
}

```

selectImports方法调用getAutoConfigurationEntry方法

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

通过该方法给IOC容器批量导入一批自动配置的组件。

总结: `@EnableAutoConfiguration` 是关键(启用自动配置), 内部实际上就去加载 `META-INF/spring.factories` 文件的信息, 然后筛选出以 `EnableAutoConfiguration` 为key的数据, 加载到IOC容器中, 实现自动配置功能!

Springboot自动配置的总结:

- SpringBoot先加载所有的自动配置类 `xxxxxAutoConfiguration`
- 每个自动配置类按照条件进行生效, 默认都会绑定配置文件指定的值。`xxxxProperties` 里面拿。`xxxProperties`和配置文件进行了绑定
- 生效的配置类就会给容器中装配很多组件
- 只要容器中有这些组件, 相当于这些功能就有了
- 定制化配置
  - 用户直接自己@Bean替换底层的组件
  - 用户去看这个组件是获取的配置文件什么值就去修改。

`xxxxxAutoConfiguration` ---> 组件 ---> `xxxxProperties`里面拿值 ----> `application.properties`

我的总结: 总的来说就是通过配置类: 先加载全部配置类, 但是并不全部生效, 按照条件生效。各个配置类的信息都是从`xxxproperties`里面拿的, 而`xxxProperties`又和我们的根配置文件绑定。因此, 添加或者修改注解可以通过自己编写的@Bean添加, 也可通过修改配置文件。一般是第二个, 方便又快捷。

## 8. springboot开发web应用的思路

- 引入场景启动器

<https://docs.spring.io/spring-boot/docs/current/reference/html/using-spring-boot.html#using-boot-starter>

- 查看自动配置了什么组件
  - 自己分析, 引入场景对应的自动配置一般都生效了, 组件都注入了
  - 配置文件中 `debug=true` 开启自动配置报告, 可查看自动配置了什么组件
- 配置是否需要修改
  - 参考文件修改

<https://docs.spring.io/spring-boot/docs/current/reference/html/appendix-application-properties.html#common-application-properties>

- 自己分析。`xxxxProperties`绑定了配置文件的哪些。
- 自定义加入或者替换组件@Bean....

## 9. Lombok插件

```
1 <dependency>
2     <groupId>org.projectlombok</groupId>
3     <artifactId>lombok</artifactId>
4 </dependency>
```

```
1 @Data
2 @ToString
3 @AllArgsConstructor
4 @NoArgsConstructor
5 @EqualsAndHashCode
6 public class Dog {
7     private String name;
8     private int age;
9 }
```

```
1 package com.hhj.controller;
2
3 @RestController
4 @Slf4j
5 public class HelloController {
6     @RequestMapping("/hello")
7     public String hello(){
8         log.info("简化日志开发");
9         return "HELLO SPRINGBOOT";
10    }
11 }
12
```

## 10. 热部署工具devtools

```
1 <dependency>
2     <groupId>org.springframework.boot</groupId>
3     <artifactId>spring-boot-devtools</artifactId>
4     <optional>true</optional>
5 </dependency>
```

spring-boot-devtools 是一个为开发者服务的一个模块，其中最重要的功能就是热部署。

当我们修改了classpath下的文件（包括类文件、属性文件、页面等）时，会重新启动应用（由于其采用的双类加载器机制，这个启动会非常快）。

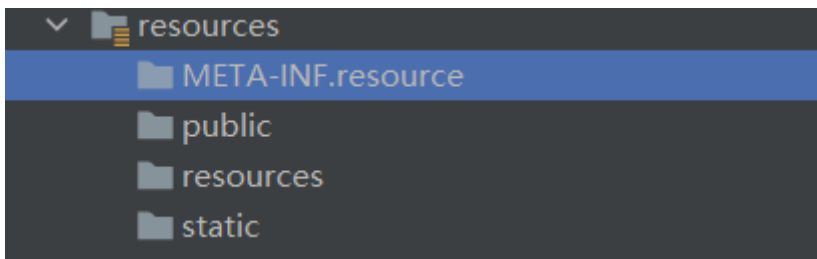
###

## 11. web开发

### ① 访问静态资源

Springboot约定，静态资源放在类路径下，包名可以是 `/static` `/public` `/resources` `/META-INF/resource`

访问时直接当前项目根路径+静态资源名即可，不需要具体的目录



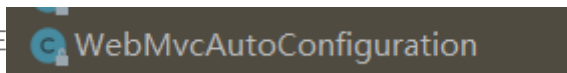
- 原理：静态资源的映射url是/\*\*，请求进来的时候，先去找controller看看有没有匹配的，不能处理的请求在交给静态资源处理器
- 关于静态资源的配置

```
1  spring:
2    mvc:
3      static-path-pattern: /res/**
4      # 访问静态资源的前缀必须是这个，即使resources下没有这个目录
5      # 但是如何配置了这个，欢迎页面和logo图无法正常显示，和源码有关，被写死
6    web:
7      resources:
8        static-locations: [classpath:/public2]
9        # 配置这个的话静态资源就必须放在这个目录下，原先的默认目录就不能用了，被替代了
10       # 这是一个数组，可以重新配置多个静态资源目录
```

- 欢迎页支持  
静态资源路径下的 index.html 就是欢迎页，
- 自定义Favicon  
favicon.ico 放在静态资源目录下即可
- 静态资源配置自动装配的源码解析

以后补

具体实现在



## ② Springboot与Rest风格

Rest风格我们都知道，url改变了，并且可以根据不同的请求方式(POST GET PUSH DELETE) 执行不同的控制器。

- 以前：/getUser 获取用户 /deleteUser 删除用户 /editUser 修改用户 /saveUser 保存用户
- 现在：/user GET获取用户 DELETE删除用户 PUT修改用户 POST保存用户

但是因为浏览器只能发送post和get请求，因此我们要配置一个核心的Filter：  
HiddenHttpMethodFilter，将POST/GET请求改成隐藏的请求方式

### ◦ 配置

```
1  spring:
2    mvc:
3      hiddenmethod:
4        filter:
5          enabled: true
```

### ◦ 用法

```
1  测试REST风格；
```



```

2 <form action="/user" method="get">
3     <input value="REST-GET 提交" type="submit"/>
4 </form>
5 <form action="/user" method="post">
6     <input value="REST-POST 提交" type="submit"/>
7 </form>
8 <form action="/user" method="post">
9     <input name="_method" type="hidden" value="delete"/>
10    <input name="_m" type="hidden" value="delete"/>
11    <input value="REST-DELETE 提交" type="submit"/>
12 </form>
13 <form action="/user" method="post">
14    <input name="_method" type="hidden" value="PUT"/>
15    <input value="REST-PUT 提交" type="submit"/>
16 </form>

```

表单提交要这么写，表现上是post，实际上还有一个type='hidden'，真正的提交方式是value

```

1 package com.hhj.controller;
2
3 @Controller
4 @ResponseBody
5 public class RestController {
6     @RequestMapping(value = "/user", method = RequestMethod.GET)
7     public String getUser(){
8         return "GET";
9     }
10
11     // 上面那样写有些麻烦，因此SPRING给我们提供了几个新注解
12     @PostMapping("/user")
13     public String saveUser(){
14         return "post";
15     }
16
17     @PutMapping("/user")
18     public String putUser(){
19         return "put";
20     }
21
22     @DeleteMapping("/user")
23     public String deleteUser(){
24         return "delete";
25     }
26 }

```

- 怎么修改固定的 \_method

```

1  @Configuration
2  public class MyConfig {
3
4      //自定义filter
5      @Bean
6      public HiddenHttpMethodFilter hiddenHttpMethodFilter(){
7          HiddenHttpMethodFilter methodFilter = new
HiddenHttpMethodFilter();
8          methodFilter.setMethodParam("_m");
9          return methodFilter;
10     }
11 }

```

### ③ springboot接收请求常用注解

- @PathVariable: 绑定Restful风格url的占位符
- @RequestHeader: 绑定请求头信息
- @RequestParam: 绑定具体key的请求参数
- @CookieValue: 绑定具体key的cookie
- @RequestBody: 绑定post方法的所有参数
- @RequestAttribute: 绑定request请求域中的参数

Restful的请求路径"car/3/owner/lisi?age=18&inters=basketball&inters=game"

怎么提取这个的请求参数呢？首先我们要知道哪些是请求参数

控制器地址映射 @GetMapping("/car/{id}/owner/{username}")

因此我们可以知道传递过来的参数有: 3 & lisi & age=18 & inters=basketball & inters=game

也可以写成这样: id=3 & username=lisi & age=18 & inters=basketball & inters=game

因为Rest风格的url把参数的key值隐藏起来了，这也是它的一个优点

```

1  <a href="car/3/owner/lisi?
    age=18&inters=basketball&inters=game">car/{id}/owner/{username}</a>

```

```

1  @GetMapping("/car/{id}/owner/{username}")
2  public Map<String,Object> getCar(@PathVariable("id") Integer id,
3                                  @PathVariable("username") String name,
4                                  @PathVariable Map<String,String> pv,
5                                  @RequestHeader("User-Agent") String
userAgent,
6                                  @RequestHeader Map<String,String> header,
7                                  @RequestParam("age") Integer age,
8                                  @RequestParam("inters") List<String>
inters,
9                                  @RequestParam Map<String,String> params,
10                                 @CookieValue("_ga") String _ga,
11                                 @CookieValue("_ga") Cookie cookie){
12     Map<String,Object> map = new HashMap<>();
13
14     map.put("id",id);
15     map.put("name",name);
16     map.put("pv",pv);
17     map.put("userAgent",userAgent);

```

```

18     map.put("headers", header);
19     map.put("age", age);
20     map.put("inters", inters);
21     map.put("params", params);
22     map.put("_ga", _ga);
23     System.out.println(cookie.getName()+"==>"+cookie.getValue());
24     return map;
25 }

```

## 测试@RequestBody

```

1 <form action="/save" method="post">
2     测试@RequestBody获取数据 <br/>
3     用户名: <input name="userName"/> <br>
4     邮箱: <input name="email"/>
5     <input type="submit" value="提交"/>
6 </form>

```

```

1 @PostMapping("/save")
2 public Map postMethod(@RequestBody String content){
3     Map<String, Object> map = new HashMap<>();
4     map.put("content", content);
5     return map;
6 }

```

## 总结

springboot接收参数其实只有三个方式：基本类型、实体类、map

- 关于request域中存值取值的三种方法

```

1 map<String, String>
2 Model model
3 HttpServletRequest req

```

## ④ thymeleaf模板引擎

## ⑤ 拦截器

类似于Servlet开发的过滤器，要想自定义拦截器，必须实现 `HandlerInterceptor` 接口。

创建一个拦截器

```

1 public class MyInterceptor implements HandlerInterceptor {
2
3     // 执行控制器方法之前执行该方法
4     // return true 放行 否则不调用控制器方法
5     @Override
6     public boolean preHandle(HttpServletRequest request, HttpServletResponse
7     response, Object handler) throws Exception {
8         return false;
9     }
10
11     // 控制器方法执行后执行该方法
12     @Override

```

```

12     public void postHandle(HttpServletRequest request, HttpServletResponse
response, Object handler, ModelAndView modelAndView) throws Exception {
13
14     }
15
16     // 页面渲染后执行该方法
17     @Override
18     public void afterCompletion(HttpServletRequest request,
HttpServletResponse response, Object handler, Exception ex) throws Exception
19     {
20     }
21 }

```

注册拦截器(实现接口, 重写方法)

```

1  @Configuration
2  public class InterceptorConfig implements WebMvcConfigurer {
3
4      // 实现该接口 重写该方法
5      @Override
6      public void addInterceptors(InterceptorRegistry registry) {
7          registry.addInterceptor(new MyInterceptor())
8              .addPathPatterns("/**") //所有请求都被拦截包括静态资源
9
10         .excludePathPatterns("/", "/login", "/css/**", "/fonts/**", "/images/**",
11             "/js/**", "/aa/**"); //放行的请求
12     }
13 }

```

拦截器和过滤器的区别:

- 实现的功能都差不多
- 过滤器是Servlet API原生的东西, 可以任意使用, 但是它不由spring容器管理, 也就是不需要添加到spring容器中
- 拦截器是springmvc的东西, 必须添加到容器中才能生效
- 它们的实现都是创建自己的拦截器并进行注册
  - 注册可以在xml中配置
  - 也可以使用配置类的方式
- 拦截器的原理是拦截器链

## ⑥ 文件上传

- 表单要添加enctype属性

```

1  测试文件上传
2  <form action="fileLoad" method="POST" enctype="multipart/form-data">
3      单个文件上传: <input type="file" name="file"/>
4      多个文件上传:<input type="file" name="files"/>
5      <input type="submit" value="提交"/>
6  </form>

```

```

1  @RestController

```

```

2 // 文件上传
3 @Slf4j
4 public class FileLoadController {
5
6     @PostMapping("/fileLoad")
7     public String fileLoad(@RequestPart("file")MultipartFile
file,@RequestPart("files") MultipartFile[] files) throws IOException {
8         log.info("单个文件: {} \n 多个文件:
{} ",file.getOriginalFilename(),files.length);
9
10        if(!file.isEmpty()){
11            // 保存到本地或者服务器
12            String filename = file.getOriginalFilename();
13            file.transferTo(new File("D:\\"+filename));
14        }
15
16        if(files.length>0){
17            for(MultipartFile a:files){
18                String filename = a.getOriginalFilename();
19                a.transferTo(new File("D:\\"+filename));
20            }
21        }
22
23        return "上传成功";
24    }
25
26 }

```

## ⑦ 错误处理

springboot会自动识别当前类目录下的/template/error文件夹，出错时根据文件名返回特定的页面

- 比如404, 返回404.html
- 500,返回500.html

## ⑧ WEB原生API: Servlet、Filter以及Listen的嵌入

- 在配置类使用注解@ComponentScan, 把原生servlet组件放在那个包下
- 使用RegistrationBean

```

1 @Configuration
2 public class MyRegistConfig {
3
4     @Bean
5     public ServletRegistrationBean myServlet(){
6         MyServlet myServlet = new MyServlet();
7
8         return new ServletRegistrationBean(myServlet,"/my","/my02");
9     }
10
11
12     @Bean
13     public FilterRegistrationBean myFilter(){
14
15         MyFilter myFilter = new MyFilter();
16         // return new FilterRegistrationBean(myFilter,myServlet());

```

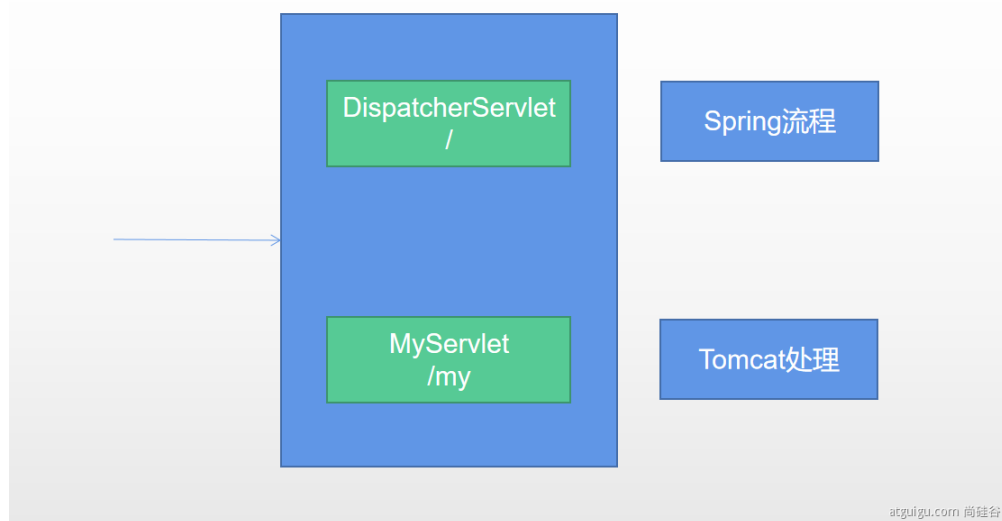
```

17     FilterRegistrationBean filterRegistrationBean = new
FilterRegistrationBean(myFilter);
18
19     filterRegistrationBean.setUrlPatterns(Arrays.asList("/my", "/css/*"));
20     return filterRegistrationBean;
21 }
22
23 @Bean
24 public ServletListenerRegistrationBean myListener(){
25     MySwervletContextListener mySwervletContextListener = new
MySwervletContextListener();
26     return new
ServletListenerRegistrationBean(mySwervletContextListener);
27 }

```

- 注意点:

- DispatcherServlet的控制器方法和原生Servlet不是一个东西，一般的springmvc只用到一个Servlet，就是DispatcherServlet
- 因为上面那点，所以springIOC的拦截器对原生Servlet不起作用



- DispatcherServlet对应的配置项时 `spring.mvc`

## ⑨ 配置springboot服务器

Springboot默认支持的服务器有Tomcat, Jetty, or Undertow

加载服务器的原理是:

- springboot发现当前应用是web应用
- 创建一个web版本的IOC容器，查找匹配对应的webserver工厂TomcatServletWebServerFactory, JettyServletWebServerFactory, or UndertowServletWebServerFactory
- 判断系统导入了那个web服务器的包，springboot-starter默认导入tomcat包
- 创建web服务器，启动
- 配置服务器的配置项时 `server.`

如何切换服务器？更改默认导入的包即可

```

1 <dependency>
2   <groupId>org.springframework.boot</groupId>
3   <artifactId>spring-boot-starter-web</artifactId>
4   <exclusions>
5     <exclusion>
6       <groupId>org.springframework.boot</groupId>
7       <artifactId>spring-boot-starter-tomcat</artifactId>
8     </exclusion>
9   </exclusions>
10 </dependency>

```

导入其他服务器的starter依赖

```

1 <groupId>org.springframework.boot</groupId>
2 <artifactId>spring-boot-starter-undertow</artifactId>

```

## ⑩ 如何定制Springboot

定制化的套路一般是：**场景starter** - xxxxAutoConfiguration - 导入xxx组件 - 绑定xxxProperties -- **绑定配置文件项**

但是springboot已经帮我们作了大部分的工作，所以我们一般只需要修改配置文件。但是也有一些定制化不能靠修改配置文件实现，需要像容器中添加bean，比如拦截器。

常见的定制化操作：

- 修改配置文件
- 编写自定义的配置类 xxxConfiguration+@Bean替换、增加容器中默认组件
- web开发时我们一般编写一个配置类实现 WebMvcConfigurer 接口即可定制化web功能  
使用@Bean给容器中再扩展一些组件

## 12. 数据访问

### ① 快速入门

- 需要引入场景选择器

```

1 <dependency>
2   <groupId>org.springframework.boot</groupId>
3   <artifactId>spring-boot-starter-data-jdbc</artifactId>
4 </dependency>

```

这个是spring提供的，里面集成了jdbcTemplate、HikariDataSource等，但是没有数据库驱动，因为springboot不知道我们使用的是哪个数据库，所以还需要引入mysql驱动

```

1 <dependency>
2   <groupId>mysql</groupId>
3   <artifactId>mysql-connector-java</artifactId>
4 </dependency>

```

- 配置默认的HikariDataSource数据连接池

如果没有自己添加数据库连接池，springboot默认使用这个

```

1  spring:
2    datasource:
3      url: jdbc:mysql://localhost:3306/sbtest
4      username: root
5      password: 123abc
6      driver-class-name: com.mysql.cj.jdbc.Driver

```

- 使用JdbcTemplate进行简单测试

```

1  public class SpringbootApplicationMain {
2    public static void main(String[] args) {
3      ConfigurableApplicationContext run =
SpringApplication.run(SpringbootApplicationMain.class, args);
4
5      JdbcTemplate bean = run.getBean(JdbcTemplate.class);
6      Long aLong = bean.queryForObject("select count(*) from student",
Long.class);
7      System.out.println(aLong);
8
9    }

```

## ② 使用Druid数据连接池

```

1  <dependency>
2    <groupId>com.alibaba</groupId>
3    <artifactId>druid</artifactId>
4    <version>1.1.17</version>
5  </dependency>

```

切换数据库连接池有两个方式：一是自己定义连接池并导入容器，二是找场景启动器，让它帮我们自动配置

- 自定义

```

1  @Configuration
2  public class DruidConfig {
3
4      // 默认自动配置是判断容器中没有才会配
5      @ConditionalOnMissingBean(DataSource.class)
6      // @ConfigurationProperties("spring.datasource")
7      @Bean
8      @ConfigurationProperties(prefix = "spring.datasource")
9      public DataSource dataSource(){
10         DruidDataSource druidDataSource = new DruidDataSource();
11         // druidDataSource.setUrl();
12         // ...可以直接和配置文件绑定，不用这样写
13         // 加入监控功能
14         druidDataSource.setFilters("stat,wall");
15         return druidDataSource;
16     }
17 }

```



|  |   |
|--|---|
| ← → ↻ localhost:8060/druid/index.html  |   |
| Druid Monitor 首页 数据源 SQL监控 SQL防火墙 Web应用 URI监控 Session监控 Spring监控 JSON API 重置 |   |
| Stat Index <a href="#">查看JSON API</a>  |   |
| 版本   | 1.1.17  |
| 驱动   | com.mysql.cj.jdbc.Driver<br>com.alibaba.druid.mock.MockDriver<br>com.alibaba.druid.proxy.DruidDriver  |
| 是否允许重置   | true  |
| 重置次数   | 0   |
| Java版本   | 1.8.0_221   |
| JVM名称  | Java HotSpot(TM) 64-Bit Server VM   |
| classpath 路径   | C<br>I:\Program Files\Java\jdk1.8.0_221\jre\lib\charsets.jar<br>C<br>I:\Program Files\Java\jdk1.8.0_221\jre\lib\deploy.jar<br>C<br>I:\Program Files\Java\jdk1.8.0_221\jre\lib\ext\access-bridge-64.jar<br>C<br>I:\Program Files\Java\jdk1.8.0_221\jre\lib\ext\cldrdata.jar<br>C<br>I:\Program Files\Java\jdk1.8.0_221\jre\lib\ext\dnsns.jar |

添加内置监控页面，该页面是一个Servlet，因此要使用之间那种方式为spring容器添加一个servlet

```

1      @Bean
2      public ServletRegistrationBean addStatViewServlet(){
3          ServletRegistrationBean<StatViewServlet> registrationBean = new
ServletRegistrationBean<>(new StatViewServlet(), "/druid/*");
4          // 可以设置登录用户民和密码
5          //      registrationBean.setUrlMappings(Arrays.asList("/"));
6          registrationBean.addInitParameter("loginUsername", "root");
7          registrationBean.addInitParameter("loginPassword", "root");
8
9          return registrationBean;
10
11     }
```

添加和Spring关联的监控，是一个Filter过滤器

```

1      //      @Bean
2      public FilterRegistrationBean webStatFilter(){
3          webStatFilter webStatFilter = new webStatFilter();
4
5          FilterRegistrationBean<WebStatFilter> filterRegistrationBean =
new FilterRegistrationBean<>(webStatFilter);
6          filterRegistrationBean.setUrlPatterns(Arrays.asList("/"));
7
8          filterRegistrationBean.addInitParameter("exclusions", "*.js,*.gif,*.jpg,
*.png,*.css,*.ico,/druid/*");
9
10         return filterRegistrationBean;
11     }
```

### ③ 整合Mybatis

导入start

```

1 <dependency>
2   <groupId>org.mybatis.spring.boot</groupId>
3   <artifactId>mybatis-spring-boot-starter</artifactId>
4   <version>2.1.3</version>
5 </dependency>

```

配置springboot配置文件,只需要配置全局配置文件和mapper映射文件的位置即可

```

1 # 配置mybatis
2 mybatis:
3   config-location: classpath:mybatis/mybatis-config.xml
4   mapper-locations: classpath:mybatis/mapper/*

```

- 使用xml配置查询数据库
  - 编写dao层mapper接口

```

1 @Mapper
2 // 只有带有@Mapper注解的接口才会使用sqlSessionFactory创建对应的mapper,调用方法
3 // 而且该mapper才会被注入到IOC容器中
4 public interface StudentMapper {
5
6     public Student getStudentById(int id);
7 }

```

底层接口添加mapper注解,然后该mapper会被自动注入到IOC容器中,不需要自己  
`sqlSessionFactory.getMapper(Student.class)`

- 编写mapper接口映射的xml文件

```

1 <?xml version="1.0" encoding="UTF-8" ?>
2 <!DOCTYPE mapper
3     PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
4     "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
5 <mapper namespace="com.hhj.mapper.StudentMapper">
6     <select id="getStudentById" resultType="student">
7         select * from student where id = #{id}
8     </select>
9 </mapper>

```

- 编写service层接口和实现类

```

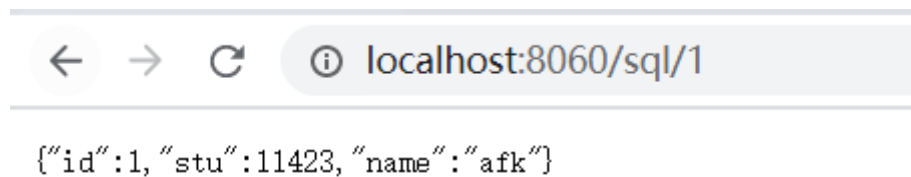
1 @Service
2 public class StudentServiceImpl implements StudentService {
3
4     @Autowired
5     StudentMapper studentMapper;
6
7     @Override
8     public Student getStudentById(int id) {
9         Student student = studentMapper.getStudentById(id);
10        return student;
11    }
12 }

```

- controller层调用

```
1 @RestController
2 public class SqlController {
3
4     @Autowired
5     StudentService studentService;
6
7     @GetMapping("/sql/{id}")
8     public Student getUserById(@PathVariable("id") int id){
9         Student studentById = studentService.getStudentById(id);
10        return studentById;
11    }
12 }
```

- 结果展示



- 使用注解查询数据库

注解查询和xml查询不同的是有可能不需要写映射的mapper文件，在接口方法上直接使用 `@Select` `@Insert` 等查询数据库即可，不过最佳实战是简单方法使用注解方式查询，复杂方法建议还是使用mapper映射

```
1 @Insert("insert into student(stu,name) values ({stu},{name})")
2 public int insertStudent(Student student);
```