1. Springboot简介

springboot是spring在2014年提出的框架，他采用一站式的开发环境，很少的配置就可以开始开发，大大降低了开发成本，Spring Boot来简化Spring应用开发，整个Spring技术栈的一个大整合。

1. Springboot的优点

快速创建独立运行的Spring项目以及与主流框架集成

使用嵌入式的Servlet容器，应用无需打成WAR包

starters自动依赖与版本控制

大量的自动配置，简化开发，也可修改默认值

无需配置XML，无代码生成，开箱即用

准生产环境的运行时应用监控 – 与云计算的天然集成

1. starts 启动器

一个start是一个功能的集合，导入一个starts相当于导入了一系列的jar包，而且springboot会做好版本控制，那么就可以很好的进行这个功能的开发，无需再做配置

1. 单体应用，

All in one ，传统的引用模式，所有的模块都在一个地方，因为全部的东西都在一个地方，开发测试维护都比较方便，部署的的时候将应用直接放在服务器上面就可以了，当访问量变大的时候，将相同的应用赋值多份，放在不同的服务器上运行，再通过负载均衡，就可以提高并发能力，

但是问题也很严重

1牵一发而动全身，他们之间的耦合度过高，导致很难维护，

2当软件大到一定程度地时候，all in one 大得离谱，合作分工也很难进行

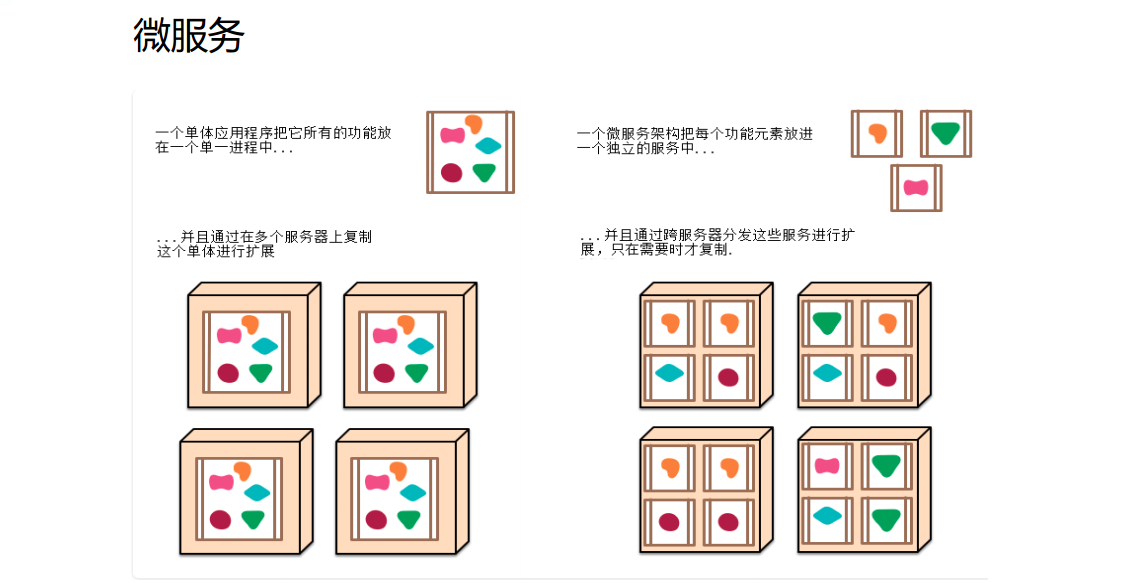
1. 微服务

2014年martin fowler，首先提出，是一种架构风格，一个程序由多个小型的服务构成，他们运行在自己的独立的环境中，他们相互之间可以通过http来进行轻量级的沟通或者通过rpc进行调用，

微服务将每一个功能都抽出成模块，可独立运行，在服务器上部署的时候，可以根据需求来部署不同的模块的多少，也可独立运维某些模块，灵活性更高，每个功能单元有独立的功能，是一个分部式的系统

6、微服务原理

Springboot可以帮助我们快速地创建功能单元，springcloud可以帮我们快速地做功能单元之间的交互，互联互调，springdate做数据交互





7、第一个helloword程序

编写一个主程序；启动Spring Boot应用

/\*\*

\* @SpringBootApplication 来标注一个主程序类，说明这是一个Spring Boot应用

\*/

@SpringBootApplication

public class HelloWorldMainApplication {

public static void main(String[] args) {

// Spring应用启动起来

SpringApplication.run(HelloWorldMainApplication.class,args);

}

}

编写相关的Controller、Service

@Controller

public class HelloController {

@ResponseBody

@RequestMapping("/hello")

public String hello(){

return "Hello World!";

}

}

```

运行主程序测试

简化部署

可以不用打war包，直接将springboot打包成jar包，然后就可以直接在服务器上运行

<!-- 这个插件，可以将应用打包成一个可执行的jar包；-->

<build>

<plugins>

<plugin>

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

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

</build>

将这个应用打成jar包，直接使用java -jar的命令进行执行；

8、spring boot 原理分析1（jar包的版本谁来控制？）

在pom.xml文件里面有

<**parent**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-parent</**artifactId**>  
 <**version**>2.0.3.RELEASE</**version**>  
 <**relativePath**/> *<!-- lookup parent from repository -->*</**parent**>

这段话的意思是导入,springboot的父项目，这个父项目里面有所有jar包的版本信息

他来真正管理Spring Boot应用里面的所有依赖版本；**是Spring Boot的版本仲裁中心**；

以后我们导入依赖默认是不需要写版本；（没有在dependencies里面管理的依赖自然需要声明版本号）

spring-boot-starter-parent只是一个版本仲裁，并不介入导入jar包，知识控制jar包的版本。

9、springboot原理2（jar包时谁导入的？，启动器是什么？）

Springboot将jar包进行以功能进行分类，如将做缓存的分为一类，叫做cache，做web分为一类，叫做web，这样子每一个类别称作一个启动器，启动器就是一系列jar包的组合，spring的导入包的方式以启动器的方式导入jar包，要用某些功能的时候，直接导入相应的启动器（starts），相当于导入了一系列的jar包

如导入web相关的启动器

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

相当于导入了

Json 、validlater、web 、mvc等一系列的jar包

10、springboot原理3（为何不用书写配置文件就可以自动配置好？Springboot如何完成自动配置？）

在运行springboot 的时候无需书写配置文件，只需要在一个主程序上写一个主方法，传入一个@SpringBootApplication标注的类作为参数，

就可以让程序跑起来，就可以让所有接口生效，所有注解和方法都加入到映射

如

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

@SpringBootApplication这个注解有什么神奇之处

1. Spring Boot应用标注在某个类上说明这个类是SpringBoot的主配置类，SpringBoot就应该运行这个类的main方法来启动SpringBoot应用；

2.@SpringBootApplication是一个组合注解，他包含下面的注解

@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) })

@SpringBootConfiguration，与@configuration相似

这个标注标在类上，表示这个类是一个配置类（等于配置文件），配置类会被当作spring容器的一个组件，包含进来

@EnableAutoConfiguration 开启自动配置

只要加上这个注解，springboot就会自动配置，自动扫描包，自动导mvc组件，如dispatchservlet，conversionsever等等等等，那么他就可以实现自动运行、

@EnableAutoConfiguration

由 @AutoConfigurationPackage和@Import(EnableAutoConfigurationImportSelector.class)组成，

其中@AutoConfigurationPackage

会通过方法得到主配置类所在的包名，然后将这个包的所有包和类扫描进入，因此springboot程序有一个大的包，然后在下面键子包，打包相当于是根目录，脱离了大包的类不能被扫描

@Import(EnableAutoConfigurationImportSelector.class)

导入自动配置类，在springboot里面，有一个包里面编写了很多配置类（相当于配置文件，可以向容器导入配置和注入相应的对象），而这个注解就是向容器中导入很多的配置类的对象，相当于为容器书写配置文件，于是就不用手动书写配置文件，以前用ssm需要配置的东西springboot配置类全部帮我们做好了

而左右的自动配置类在

J2EE的整体整合解决方案和自动配置都在spring-boot-autoconfigure-1.5.9.RELEASE.jar；

总结@SpringBootApplication的功能

1. 配置将一个类标注为主类，成为springboot的启动程序入口
2. 扫描该类对应的包下的所有子包
3. 将大量的自动配置类加载进入程序，实现自动配置

11、springboot快速创建项目

可以用Spring Initializer快速创建springboot项目

主程序已经生成好了，我们只需要我们自己的逻辑

- resources文件夹中目录结构

- static：保存所有的静态资源； js css images；

- templates：保存所有的模板页面；（Spring Boot默认jar包使用嵌入式的Tomcat，默人不支持JSP页面）；可以使用模板引擎（freemarker、thymeleaf）；

- application.properties：Spring Boot应用的配置文件，。Property，或yaml可以修改一些默认设置；

12、配置文件（SpringBoot将所有东西配置好了，配置文件用于修改配置）

SpringBoot使用一个全局的配置文件，配置文件名是固定的；

SpringBoot只有两个全局的配置文件，就是application.properties和application.yml

•application.properties

在里面写server.port=”8081”就能改变tomcat的端口号

•application.yml

​ 以前的配置文件；大多都使用的是 \*\*xxxx.xml\*\*文件；

​ YAML：\*\*以数据为中心\*\*，比json、xml等更适合做配置文件；

13、yml的使用（yml是将空格玩到极致的配置文件，要注意空格）

YAML语法：

基本语法

k:(空格)v：表示一对键值对（空格必须有）；

以空格的缩进来控制层级关系；只要是左对齐的一列数据，都是同一个层级的

例：

server:

port: 8081（：到8之间要加空格）

path: /hello

属性和值也是大小写敏感；

2、值的写法

字面量：普通的值（数字，字符串，布尔）

​ k: v：字面直接来写；

​ 字符串默认不用加上单引号或者双引号；

​ ""：双引号；不会转义字符串里面的特殊字符；特殊字符会作为本身想表示的意思

​ name: "zhangsan \n lisi"：输出；zhangsan 换行 lisi

​ ''：单引号；会转义特殊字符，特殊字符最终只是一个普通的字符串数据

​ name: ‘zhangsan \n lisi’：输出；zhangsan \n lisi

对象、Map（属性和值）（键值对）：

​ k: v：在下一行来写对象的属性和值的关系；注意缩进

对象还是k: v的方式

friends:

lastName: zhangsan

age: 20

行内写法：

friends: {lastName: zhangsan,age: 18}

数组（List、Set）：

用- 值表示数组中的一个元素

pets:

- cat

- dog

- pig

行内写法

pets: [cat,dog,pig]

14、获取配置文件的值，注入到对象用两个注解

@Component

@ConfigurationProperties(prefix = "person")

configuratiomProperties;默认从全局配置文件中获取值

配置文件

person:

lastName: hello

age: 18

boss: false

birth: 2017/12/12

maps: {k1: v1,k2: 12}

lists:

- lisi

- zhaoliu

dog:

name: 小狗

age: 12

javaBean：

\* 将配置文件中配置的每一个属性的值，映射到这个组件中

\* @ConfigurationProperties：告诉SpringBoot将本类中的所有属性和配置文件中相关的配置进行绑定；

\* prefix = "person"：配置文件中哪个下面的所有属性进行一一映射

\*

\* 只有这个组件是容器中的组件，才能容器提供的@ConfigurationProperties功能；

@Component

@ConfigurationProperties(prefix = "person")

public class Person {

private String lastName;

private Integer age;

private Boolean boss;

private Date birth;

private Map<String,Object> maps;

private List<Object> lists;

private Dog dog;

我们可以导入配置文件处理器，以后编写配置就有提示了

<!--导入配置文件处理器，配置文件进行绑定就会有提示-->

<dependency>

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

<artifactId>spring-boot-configuration-processor</artifactId>

<optional>true</optional>

</dependency>

15、在proprietary文件里面书写，也可以实现注入

*#idea默认使用的是utf-8，而property默认使用的是ascii，会乱码，要到file enconding里设置一下就可以*preson.age=12  
preson.birth=2017/12/12  
preson.boss=false  
preson.lastname=张三  
preson.dog.color=red  
preson.dog.sge=18  
preson.maps.k1=aa  
preson.maps.k2=bb  
preson.list=a,b,c

16、@value也可以将配置文件中的值注入

@Component

public class Person {

@value($({lastName}))

private String lastName;

@value(“#{11\*2”})

private Integer age;

private Boolean boss;

private Date birth;

private Map<String,Object> maps;

private List<Object> lists;

private Dog dog;

17、@value和@configuratiomProperties的区别



配置文件yml还是properties他们都能获取到值；

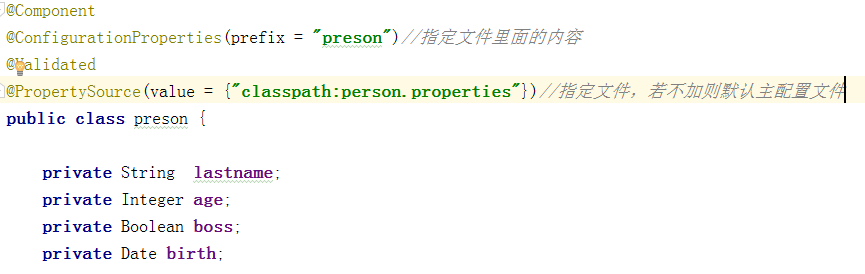
如果说，我们只是在某个业务逻辑中需要获取一下配置文件中的某项值，使用@Value；

如果说，我们专门编写了一个javaBean来和配置文件进行映射，我们就直接使用@ConfigurationProperties；

18、@PropertySource&@ImportResource&@Bean

@PropertySource：（必须和@ConfigurationProperties一起使用）

加载指定配置文件的内容，可以加载非主配置文件的内容，我们知@ConfigurationProperties；只能加载主配置文件的内容，而这个注解可以加载指定配置文件的内容，但是PropertySource须和ConfigurationProperties一同使用，PropertySource指定非默认文件，而是自己写的配置文件ConfigurationProperties指定是文件里的什么内容



在配置文件的内容注入里面@propertySources为@value和@ConfigurationProperties的父配置

@propertySources指定具体使用的配置文件，如果没使用这个配置，则默认是主配置文件

@value：指定具体注入的一个值

@ConfigurationProperties：批量指定注入的内容，如一个对象

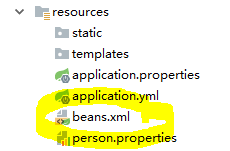
@ImportResource：

导入Spring的配置文件,如自己编写xml，让配置文件里面的内容生效；

Spring Boot里面没有Spring的xml配置文件，我们自己编写的xml配置文件，也不能自动识别；想让Spring的配置文件生效，加载进来；@ImportResource标注在一个任意的配置类上，最常见的的便是放在主配置文件上面

如

@SpringBootApplication  
@ImportResource(locations = {"classpath:beans.xml"})  
public class ZzxHellowordApplication {  
  
 public static void main(String[] args) {  
 SpringApplication.run(ZzxHellowordApplication.class, args);  
 }  
}



19、自己编写配置类来实现xml文件相同的功能，springboot建议使用配置类代替配置文件

*/\*   
\* @Configuration,指明这是一个配置类，配置类相当于配置文件\* \*/*@Configuration  
**public class** testConfigClass {  
 */\*  
 \* @Bean 放在方法上面，能够将这个方法的返回值放入到容器中，  
 \* 而放进去的东西的key就是这个方法名的小写  
 \* \*/* @Bean  
 **public** testConfigClass hellobean(){  
 **return new** testConfigClass();  
 }  
}

@Configuration标注的是一个配置类，相当于配置文件，上面的配置类相当于是一个xml文件，向里面注入了一盒对象，对象名为hellobean，值为**new** testConfigClass()，springboot里面一般用配置类代替配置文件，而且springboot的自动配置也是通过自动配置类来实现的

20、配置文件占位符

在property文件里面可以用${}使用随机数，以及使用上面存在的值来配置下面的东西

1、随机数

${random.value}、${random.int}、${random.long}

${random.int(10)}、${random.int[1024,65536]}

如

*#idea默认使用的是utf-8，而property默认使用的是ascii，会乱码，要到file enconding里设置一下就可以  
#为age配置一个随机数***preson.age**=**${random.int}  
preson.birth**=**2017/12/12  
preson.boss**=**false  
preson.lastname**=**${random.uuid}张四***#用上面的preson.lastname来配置下面***preson.dog.color**=**${preson.lastname}\_red  
preson.dog.sge**=**15***#为k1配置一个值，不存在则等于aaaaa***preson.maps.k1**=**${preson.lastname:aaaaa}  
preson.maps.k2**=**bb  
preson.list**=**a,b,c**

1. 多profile未见，配置多个生产环境，方便生成，测试，开发
2. 主配置文件为Properties文件

可以书写

application.properties //主配置文件

application-dev.properties//生产环境

application-produce.properties//发布环境

三个配置文件使用哪一个，可以在application.proerties里面去指定

如指定使用dev环境有用

Spring.profiles.avtive=dev

1. 如果使用yaml作为配置环境，则可以使用文档快的方式来编写多环境，然后在第一个文档快里面指定使用的环境

server:

port: 8081

spring:

profiles:

active: prod //指定使用的环境

--- //-- 为文档快分隔符

server:

port: 8083

spring:

profiles: dev

---

server:

port: 8084

spring:

profiles: prod #指定属于哪个环境

3、在配置好文件后，也可以在命令行指定运行环境，而且命令行权限最高：

​ java -jar spring-boot-02-config-0.0.1-SNAPSHOT.jar --spring.profiles.active=dev；

​ 可以直接在测试的时候，配置传入命令行参数

1. springboot能够自动加载配置文件的几个路径

springboot 启动会扫描以下位置的application.properties或者application.yml文件作为Spring boot的默认配置文件

–file:./config/ 在直接在项目下键一个与config，与src同级的目录然后再下面放文件

–file:./ 直接在目录下，与pom.xml同级

–classpath:/config/ 相当于resources/config/

–classpath:/ 相当于 resources/

优先级由高到底，对于相同的配置，高优先级的配置会覆盖低优先级的配置；

对于不相同的配置，SpringBoot会从这四个位置全部加载主配置文件；互补配置；

除此之外

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

项目打包好以后，我们可以使用命令行参数的形式，启动项目的时候来指定配置文件的新位置；指定配置文件和默认加载的这些配置文件共同起作用形成互补配置；

如达成jar后再重新指定配置文件

java -jar spring-boot-02-config-02-0.0.1-SNAPSHOT.jar --spring.config.location=G:/application.properties

23、外部配置加载顺序，就是项目配置文件的加载顺序

\*\*==SpringBoot也可以从以下位置加载配置； 优先级从高到低；高优先级的配置覆盖低优先级的配置，所有的配置会形成互补配置==\*\*

\*\*1.命令行参数\*\*

所有的配置都可以在命令行上进行指定

java -jar spring-boot-02-config-02-0.0.1-SNAPSHOT.jar --server.port=8087 --server.context-path=/abc

多个配置用空格分开； --配置项=值

2.来自java:comp/env的JNDI属性

3.Java系统属性（System.getProperties()）

4.操作系统环境变量

5.RandomValuePropertySource配置的random.\*属性值

由jar包外向jar包内进行寻找；

优先加载带profil

\*\*6.jar包外部的application-{profile}.properties或application.yml(带spring.profile)配置文件\*\*

//直接放在打好的jar包的外面，同级目录，而且指定了active

\*\*7.jar包内部的application-{profile}.properties或application.yml(带spring.profile)配置文件\*\*

==\*\*再来加载不带profile\*\*==

\*\*8.jar包外部的application.properties或application.yml(不带spring.profile)配置文件\*\*

//直接放在打好的jar包的外面，同级目录

\*\*9.jar包内部的application.properties或application.yml(不带spring.profile)配置文件\*\*

10.@Configuration注解类上的@PropertySource

11.通过SpringApplication.setDefaultProperties指定的默认属性

所有支持的配置加载来源；

[参考官方文档](https://docs.spring.io/spring-boot/docs/1.5.9.RELEASE/reference/htmlsingle/#boot-features-external-config)

1. springboot自动配置原理？（springboot的精髓）

@import（@EnableAutoConfigurationSecletor）

他会扫描spring-boot-AutoConfiguration下的META-INF/spring.Factories拿到Enableconfiguration下面所有的自动配置类的类名，然后放入到缓存（因为自动装配会不间断注入类，所以需要不断读取Factories文件，因此直接放缓存节约性能）。

1. 通过全限定类名找配置类，通过配置类上面标注的判断条件，判定是否启用这个配置类（即是否注入这个类，有109个）

3、生成bean放入到spring容器

4、注：所有配置类springboot已经写好放在了spring-boot-AutoConfiguratio下面

@EnableAutoConfiguration

她是一系列注解的集合注解，他的功能主要是扫描和主配置文件同一个包下的所有文件，不是在这个包下面则扫描不到

@configuration，

向springboot表明这是一个配置类，那么springboot就会将他作为一个组件，生成一个bean放到容器里面

@autoconfigurationPackage

这个配置的作用是将和主配置类同一个包下面的所有配置包扫描进来

@springBootApplication

24、上面说到@EnableAutoConfigurationSecletor会导入自动配置类的bean，那么springBoot自动配置类如何具体工作？

以HttpEncodingAutoConfiguration（Http编码自动配置）为例解释自动配置原理；

@Configuration //表示这是一个配置类，以前编写的配置文件一样，也可以给容器中添加组件

@EnableConfigurationProperties(HttpEncodingProperties.class) //启动指定类的ConfigurationProperties功能；将配置文件中对应的值和HttpEncodingProperties绑定起来；并把HttpEncodingProperties加入到ioc容器中

@ConditionalOnWebApplication //Spring底层@Conditional注解（Spring注解版），根据不同的条件，如果满足指定的条件，整个配置类里面的配置就会生效； 判断当前应用是否是web应用，如果是，当前配置类生效

@ConditionalOnClass(CharacterEncodingFilter.class) //判断当前项目有没有这个类CharacterEncodingFilter；SpringMVC中进行乱码解决的过滤器；

@ConditionalOnProperty(prefix = "spring.http.encoding", value = "enabled", matchIfMissing = true) //判断配置文件中是否存在某个配置 spring.http.encoding.enabled；如果不存在，判断也是成立的

//即使我们配置文件中不配置pring.http.encoding.enabled=true，也是默认生效的；

public class HttpEncodingAutoConfiguration {

//他已经和SpringBoot的配置文件映射了

private final HttpEncodingProperties properties;

//只有一个有参构造器的情况下，参数的值就会从容器中拿

public HttpEncodingAutoConfiguration(HttpEncodingProperties properties) {

this.properties = properties;

}

@Bean //给容器中添加一个组件，这个组件的某些值需要从properties中获取

@ConditionalOnMissingBean(CharacterEncodingFilter.class) //判断容器没有这个组件？

public CharacterEncodingFilter characterEncodingFilter() {

CharacterEncodingFilter filter = new OrderedCharacterEncodingFilter();

filter.setEncoding(this.properties.getCharset().name());

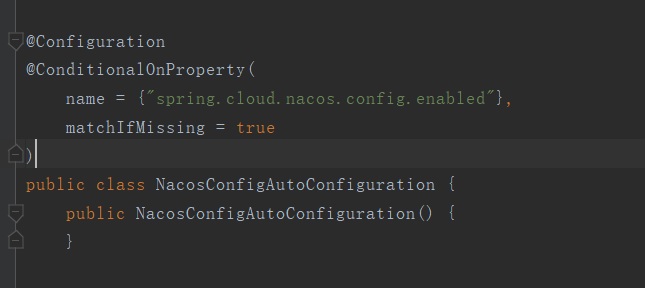
filter.setForceRequestEncoding(this.properties.shouldForce(Type.REQUEST));

filter.setForceResponseEncoding(this.properties.shouldForce(Type.RESPONSE));

return filter;

}

再如nacos



总结

1、每个自动配置类上面一定有三种注解

1.@Configuration：表明这是一个配置类

2、@EnableConfigurationProperties(xxxxProperties.class)，启动指定的自动注解（如上面的是HttpEncoding），并且将springboot对应的一个

xxxxProperties（这个类里面书写了各种配置，如编码采用什么，这些就是springboot的默认配置）与他关联

1. @Conditional+xxxx，这个是一些条件判断配置，用于判断是否需要加入这个自动配置类
2. 工作流程如下图

通过@contional。。。。判断是否需要加入这个自动配置

需要

创建对应的配置类的bean，为bean中配置赋值（如上面的为编码赋值）

配置文件里面（用户书写的，如application。Properties,application.yml）,有没有对这个配置进行赋值

从对应的xxxxproperties里面读取配置（这些配置相当于默认的配置）为bean中的配置属性赋值

无

有

从配置文件里面得到配置，进行配，为bean中的配置属性赋值

Bean中配置属性赋值完成，生成bean（组件），加入到ioc容器中，完成自动配置

1. 通过上面可以知道，其实springboot里面什么都在xxxxproperties里面配置好了，书写配置文件的目的就是为了改变默认的配置，同时配置

里面可以写什么配置，也可以从xxxxproperties里面知道

4、因此

​ \*\*1）、SpringBoot启动会加载大量的自动配置类\*\*

​ \*\*2）、我们看我们需要的功能有没有SpringBoot默认写好的自动配置类；\*\*

​ \*\*3）、我们再来看这个自动配置类中到底配置了哪些组件；（只要我们要用的组件有，我们就不需要再来配置了）\*\*

​ \*\*4）、给容器中自动配置类添加组件的时候，会从properties类中获取某些属性。我们就可以在配置文件中指定这些属性的值；\*\*

xxxxAutoConfigurartion：自动配置类；

给容器中添加组件

xxxxProperties:封装配置文件中相关属性；

25、关于@Conditiona和如何查看自己的项目里面到底springboot自动配置了哪些东西？

#### 1、@Conditional派生注解（Spring注解版原生的@Conditional作用）

作用：必须是@Conditional指定的条件成立，才给容器中添加组件，配置配里面的所有内容才生效；

| @Conditional扩展注解 | 作用（判断是否满足当前指定条件） |

| ------------------------------- | ------------------------------ |

| @ConditionalOnJava | 系统的java版本是否符合要求 |

| @ConditionalOnBean | 容器中存在指定Bean； |

| @ConditionalOnMissingBean | 容器中不存在指定Bean； |

| @ConditionalOnExpression | 满足SpEL表达式指定 |

| @ConditionalOnClass | 系统中有指定的类 |

| @ConditionalOnMissingClass | 系统中没有指定的类 |

| @ConditionalOnSingleCandidate | 容器中只有一个指定的Bean，或者这个Bean是首选Bean |

| @ConditionalOnProperty | 系统中指定的属性是否有指定的值 |

| @ConditionalOnResource | 类路径下是否存在指定资源文件 |

| @ConditionalOnWebApplication | 当前是web环境 |

| @ConditionalOnNotWebApplication | 当前不是web环境 |

| @ConditionalOnJndi | JNDI存在指定项 |

\*\*自动配置类必须在一定的条件下才能生效；\*\*

我们怎么知道哪些自动配置类生效；

\*\*==我们可以通过启用 debug=true属性（写在配置文件里，或者在yml文件里面书写**debug: true**）；来让控制台打印自动配置报告==\*\*，这样我们就可以很方便的知道哪些自动配置类生效；

如

=========================

AUTO-CONFIGURATION REPORT

=========================

Positive matches:（自动配置类启用的）

-----------------

DispatcherServletAutoConfiguration matched:

- @ConditionalOnClass found required class 'org.springframework.web.servlet.DispatcherServlet'; @ConditionalOnMissingClass did not find unwanted class (OnClassCondition)

- @ConditionalOnWebApplication (required) found StandardServletEnvironment (OnWebApplicationCondition)

Negative matches:（没有启动，没有匹配成功的自动配置类）

-----------------

ActiveMQAutoConfiguration:

Did not match:

- @ConditionalOnClass did not find required classes 'javax.jms.ConnectionFactory', 'org.apache.activemq.ActiveMQConnectionFactory' (OnClassCondition)

AopAutoConfiguration:

Did not match:

- @ConditionalOnClass did not find required classes 'org.aspectj.lang.annotation.Aspect', 'org.aspectj.lang.reflect.Advice' (OnClassCondition)

```

1. 日志框架

市面上的日志都由接口和实现类组成，接口交日志门面，实现类叫做具体实现

常见的如下

|  |  |
| --- | --- |
| 日志门面（日志接口） | 具体实现 |
| JCL（Jakarta Commons Logging） SLF4j（Simple Logging Facade for Java） boss-logging | Log4j JUL（java.util.logging） Log4j2 Logback |

Springboot选用了

SLF4j+Logback来实现日志

而spring用的是JCL+Log4j；‘

27、SLF4j使用

### 1、如何在系统中使用SLF4j https://www.slf4j.org

以后开发的时候，日志记录方法的调用，不应该来直接调用日志的实现类，而是调用日志抽象层里面的方法；

给系统里面导入slf4j的jar和 logback的实现jar

```java

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

public class HelloWorld {

public static void main(String[] args) {

Logger logger = LoggerFactory.getLogger(HelloWorld.class);

logger.info("Hello World");

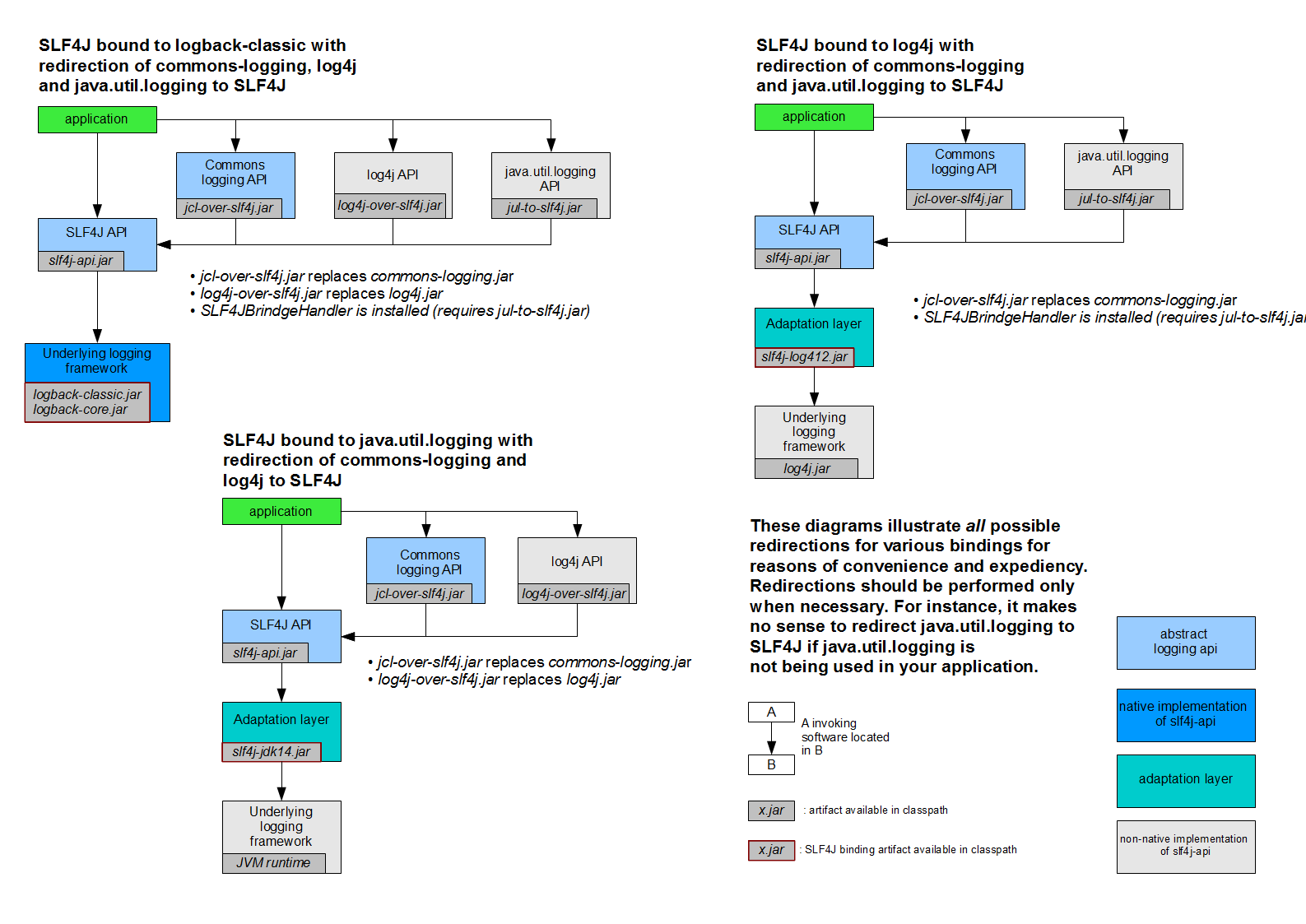
}

}

1. 历史遗留问题

（slf4j+logback）: Spring（commons-logging）、Hibernate（jboss-logging）、MyBatis、xxxx

统一日志记录，即使是别的框架和我一起统一使用slf4j进行输出？

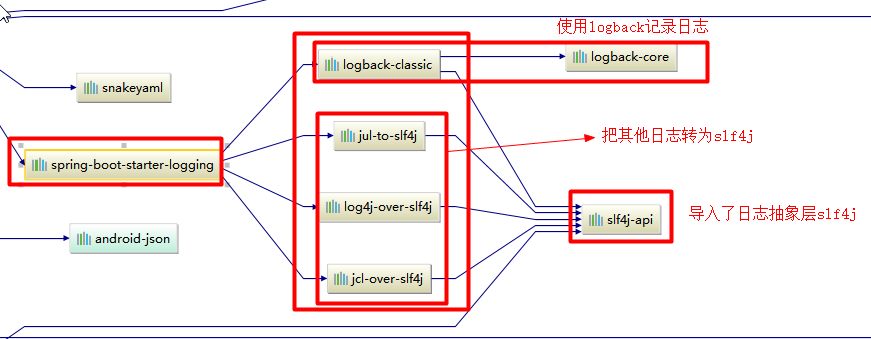


其实就是加入一个中间的适配器，然后将别的日志也转化成slf4j

1. springboot日志之间的关系图，如下
2. springboot使用的是slf4j+logback，他的父包是spring-boot-start-logging
3. 有jul-to-slf4j，log4j-to-slf4j，jcl-to-slf4j，这些包用的接口和原来的框架一样，所以可以兼容原来的框架，但是里面实际生成的内容却是slf4j，

这样就能上下兼容

3、引入其他框架的时候，只需要把这个框架依赖的日志框架排除掉即可



4、Spring框架用的是commons-logging，springboot自己排除了他的默认的日志commons-logging，然后用的是log4j-to-slf4j来转化

```xml

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-core</artifactId>

<exclusions>

<exclusion>

<groupId>commons-logging</groupId>

<artifactId>commons-logging</artifactId>

</exclusion>

</exclusions>

</dependency>

```

1. SpringBoot默认帮我们配置好了日志；

日志就像一个输出，这个输出可以到文件，也可以到控制台，在自己写日志的时候要先生产一个日志记录器，日志级别有五个

Trace

Debug

Info（root） 日志默认的级别

Warn

Error

级别由低到高，只记录比默认日志级别高的日志

```java

//记录器

Logger logger = LoggerFactory.getLogger(getClass());

@Test

public void contextLoads() {

//System.out.println();

//日志的级别；

//由低到高 trace<debug<info<warn<error

//可以调整输出的日志级别；日志就只会在这个级别以以后的高级别生效

logger.trace("这是trace日志...");

logger.debug("这是debug日志...");

//SpringBoot默认给我们使用的是info级别的，没有指定级别的就用SpringBoot默认规定的级别；root级别

logger.info("这是info日志...");

logger.warn("这是warn日志...");

logger.error("这是error日志...");

}

可以在日志文件里面设置日志的一些东西

# 在控制台输出的日志的格式

logging.pattern.console=%d{yyyy-MM-dd} [%thread] %-5level %logger{50} - %msg%n

# 指定文件中日志输出的格式

logging.pattern.file=%d{yyyy-MM-dd} === [%thread] === %-5level === %logger{50} ==== %msg%n

# 不指定路径在当前项目下生成springboot.log日志文件

# 可以指定完整的路径生成日志文件；

#logging.file=G:/springboot.log

# 在当前磁盘的根路径下创建spring文件夹和里面的log文件夹；使用 spring.log 作为默认文件，和logging.file是冲突设置

logging.path=/spring/log

1. springboot自己书写配置文件

给类路径下放上每个日志框架自己的配置文件即可；SpringBoot就不使用他默认配置的了

怎么加见下表

| Logging System | Customization |

| ----------------------- | ---------------------------------------- |

| Logback | `logback-spring.xml`, `logback-spring.groovy`, `logback.xml` or `logback.groovy` |

| Log4j2 | `log4j2-spring.xml` or `log4j2.xml` |

| JDK (Java Util Logging) | `logging.properties` |

如在springboot下把logback.xml直接放在根目录下：直接就被日志框架识别了；

\*\*logback-spring.xml\*\*：日志框架就不直接加载日志的配置项，由SpringBoot解析日志配置，可以使用SpringBoot的高级Profile功能

logback.xml是直接就被系统识别，未经过springboot，所以不能在里面配置在什么情况下啊生效

```xml

<springProfile name="staging">

<!-- configuration to be enabled when the "staging" profile is active -->

可以指定某段配置只在某个环境下生效

</springProfile>

```

如：

```xml

<appender name="stdout" class="ch.qos.logback.core.ConsoleAppender">

<!--

日志输出格式：

%d表示日期时间，

%thread表示线程名，

%-5level：级别从左显示5个字符宽度

%logger{50} 表示logger名字最长50个字符，否则按照句点分割。

%msg：日志消息，

%n是换行符

-->

<layout class="ch.qos.logback.classic.PatternLayout">

<springProfile name="dev">

<pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} ----> [%thread] ---> %-5level %logger{50} - %msg%n</pattern>

</springProfile>

<springProfile name="!dev"> <pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} ==== [%thread] ==== %-5level %logger{50} - %msg%n</pattern>

</springProfile>

</layout>

</appender>

```

如果使用logback.xml作为日志配置文件，还要使用profile功能，会有以下错误

`no applicable action for [springProfile]`

31、Web开发

1、简介

使用SpringBoot；

\*\*1）、创建SpringBoot应用，选中我们需要的模块；\*\*

\*\*2）、SpringBoot已经默认将这些场景配置好了，只需要在配置文件中指定少量配置就可以运行起来\*\*

\*\*3）、自己编写业务代码；\*\*

\*\*自动配置原理？\*\*

这个场景SpringBoot帮我们配置了什么？能不能修改？能修改哪些配置？能不能扩展？xxx

```

xxxxAutoConfiguration：帮我们给容器中自动配置组件；

xxxxProperties:配置类来封装配置文件的内容；

```

2、springboot如何引用前端框架，如引用jquery

在webmvcconfiguration里面规定了，前端的框架也可以向启动器一样引用，而且自动加载进入的juuery可以在根目录的webjar下访问到

如： localhost:8080/webjars/jquery/3.3.1/jquery.js

3、springboot的自己编写的文件怎么放

"classpath:/META-INF/resources/",

"classpath:/resources/",

"classpath:/static/",

"classpath:/public/"

"/"：当前项目的根路径

```

放在上面的文件夹下面就可以，对于一个访问，当他发现没有映射，就会到这些文件夹下面找有没有对应的静态文件

localhost:8080/abc === 去静态资源文件夹里面找abc

4、欢迎页面默认为index.html，直接放在静态文件夹里面就可以

5、图标也是在静态文件里面找，名字叫做favicon.ico 都是在静态资源文件下找

1. springboot不支持jsp，对于页面的渲染，默认使用Thymeleaf，他能够在web环境下获得后端传来的数据然后渲染
2. thymelesf的使用可以见另一个笔记

32、springboot对springmvc的自动配置（所有自动配置都在一个jar包里）

以下是SpringBoot对SpringMVC的默认配置:\*\*==（WebMvcAutoConfiguration）==\*\*

- Inclusion of `ContentNegotiatingViewResolver` and `BeanNameViewResolver` beans.

- 自动配置了ViewResolver（视图解析器：根据方法的返回值得到视图对象（View），视图对象决定如何渲染（转发？重定向？））

- ContentNegotiatingViewResolver：组合所有的视图解析器的；

- ==如何定制：我们可以自己给容器中添加一个视图解析器；自动的将其组合进来；==

- Support for serving static resources, including support for WebJars (see below).静态资源文件夹路径,webjars

- Static `index.html` support. 静态首页访问

- Custom `Favicon` support (see below). favicon.ico

​

- 自动注册了 of `Converter`, `GenericConverter`, `Formatter` beans.

- Converter：转换器； public String hello(User user)：类型转换使用Converter

- `Formatter` 格式化器； 2017.12.17===Date；

```java

@Bean

@ConditionalOnProperty(prefix = "spring.mvc", name = "date-format")//在文件中配置日期格式化的规则

public Formatter<Date> dateFormatter() {

return new DateFormatter(this.mvcProperties.getDateFormat());//日期格式化组件

}

```

​ ==自己添加的格式化器转换器，我们只需要放在容器中即可==

- Support for `HttpMessageConverters` (see below).

- HttpMessageConverter：SpringMVC用来转换Http请求和响应的；User---Json；

- `HttpMessageConverters` 是从容器中确定；获取所有的HttpMessageConverter；

==自己给容器中添加HttpMessageConverter，只需要将自己的组件注册容器中（@Bean,@Component）==

​

- Automatic registration of `MessageCodesResolver` (see below).定义错误代码生成规则

- Automatic use of a `ConfigurableWebBindingInitializer` bean (see below).

==我们可以配置一个ConfigurableWebBindingInitializer来替换默认的；（添加到容器）==

```

初始化WebDataBinder；

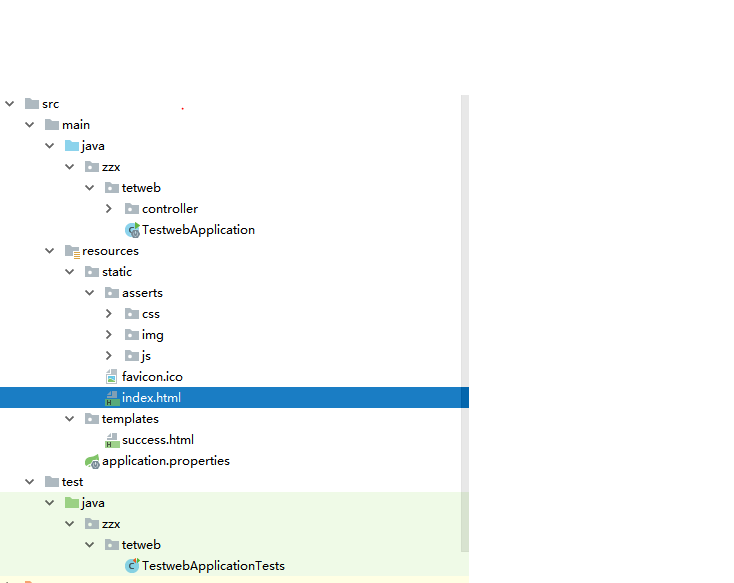
请求数据=====JavaBean；

```

\*\*org.springframework.boot.autoconfigure.web：web的所有自动场景；\*\*

总结：springboot里面整合了基本上springmvc的全部东西，当我我们需要往容器里添加自己的东西，如转换器，视图解析器，只需要自己写一个相应的类，然后利用配置类将其放到容器里（@configuration和@bean），就可以了

1. template里面放被映射的页面，如success.Jsp



Templates里面放被映射的页面,这个文件夹被thymeleaf接管，放在这里的页面才可以映射

Static可以放静态资源，首页和图标，也可以按照springboot的规定建立相应的文件夹来防止静态资源

Resources为sprinboot的classpath

与主配置文件同级的所有包springboot均能自动扫描

1. 扩展与全面接管springmvc，

扩展springmvc可以通过自己编写配置类，继承WebMvcConfigurerAdapter，然后实现里面的方法，来向容器里添加组件

组拦截器，自动跳转器

//使用WebMvcConfigurerAdapter可以来扩展SpringMVC的功能

@Configuration

public class MyMvcConfig extends WebMvcConfigurerAdapter {

@Override

public void addViewControllers(ViewControllerRegistry registry) {

// super.addViewControllers(registry);

//浏览器发送 /atguigu 请求来到 success

registry.addViewController("/atguigu").setViewName("success");

}

}

1）、WebMvcAutoConfiguration是SpringMVC的自动配置类

​ 2）、在做其他自动配置时会导入；@Import(\*\*EnableWebMvcConfiguration\*\*.class)

```java

@Configuration

public static class EnableWebMvcConfiguration extends DelegatingWebMvcConfiguration {

private final WebMvcConfigurerComposite configurers = new WebMvcConfigurerComposite();

//从容器中获取所有的WebMvcConfigurer

@Autowired(required = false)

public void setConfigurers(List<WebMvcConfigurer> configurers) {

if (!CollectionUtils.isEmpty(configurers)) {

this.configurers.addWebMvcConfigurers(configurers);

//一个参考实现；将所有的WebMvcConfigurer相关配置都来一起调用；

@Override

// public void addViewControllers(ViewControllerRegistry registry) {

// for (WebMvcConfigurer delegate : this.delegates) {

// delegate.addViewControllers(registry);

// }

}

}

}

```

​ 3）、容器中所有的WebMvcConfigurer都会一起起作用；

​ 4）、我们的配置类也会被调用；

全面接管SpringMVC；

SpringBoot对SpringMVC的自动配置不需要了，所有都是我们自己配置；所有的SpringMVC的自动配置都失效了

\*\*我们需要在配置类中添加@EnableWebMvc即可；\*\*

因为WebMvcAutoConfiguration上面有个@condition判断容器里有没有WebMvcConfigurationSupport，如果有这个自动配置类就无效，而@EnableWebMvc的功能就是往容器里面加入@EnableWebMvc。

1. 国际化

步骤：

1）、编写国际化配置文件，抽取页面需要显示的国际化消息

1. 、SpringBoot自动配置好了管理国际化资源文件的组件；(springboot默认交messages.properties,如果叫其他名字，要到配置文件里面设置，如

**spring.messages.basename**=**i18n.login**)

```java

@ConfigurationProperties(prefix = "spring.messages")

public class MessageSourceAutoConfiguration {

/\*\*

\* Comma-separated list of basenames (essentially a fully-qualified classpath

\* location), each following the ResourceBundle convention with relaxed support for

\* slash based locations. If it doesn't contain a package qualifier (such as

\* "org.mypackage"), it will be resolved from the classpath root.

\*/

private String basename = "messages";

//我们的配置文件可以直接放在类路径下叫messages.properties；

@Bean

public MessageSource messageSource() {

ResourceBundleMessageSource messageSource = new ResourceBundleMessageSource();

if (StringUtils.hasText(this.basename)) {

//设置国际化资源文件的基础名（去掉语言国家代码的）

messageSource.setBasenames(StringUtils.commaDelimitedListToStringArray(

StringUtils.trimAllWhitespace(this.basename)));

}

if (this.encoding != null) {

messageSource.setDefaultEncoding(this.encoding.name());

}

messageSource.setFallbackToSystemLocale(this.fallbackToSystemLocale);

messageSource.setCacheSeconds(this.cacheSeconds);

messageSource.setAlwaysUseMessageFormat(this.alwaysUseMessageFormat);

return messageSource;

}

```

1. 、去页面获取国际化的值；
2. 效果：根据浏览器语言设置的信息切换了国际化；
3. 开发的小技巧

禁用模板引擎thymeleaf，然后修改了静态页面之后能够刷（ctrl+F9）新到前台页面

# 禁用缓存

spring.thymeleaf.cache=false

#将更改的内容刷新到前台页面

Ctrl+f9

1. 添加组件---》拦截器

Springboot已经自动配置好了很多的东西，但是还是不够用的，当需要添加组件的时候用下面的过程

1. 编写相关组件的类
2. 将组建通过配置类添加到容器里，或者生成一个bean放到容器里面
3. 原理在于springboot添加组件的时候，如果该组件容器可存在多个，springboot会把程序员编写的组件和本身的一起添加进入容器

如果该组件只能一个，如视图解析器，他会先看程序员有没有编写，如果有，则添加程序员写的，如果没有，就添加默认的，

也就是他会扫描所有的webMvcConfigurerAdapter

以添加拦截器为例

1编写拦截器类

*/\*\*  
 \* 登陆检查，  
 \*/***public class** LoginHandlerInterceptor **implements** HandlerInterceptor {  
 *//目标方法执行之前* @Override  
 **public boolean** preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) **throws** Exception {  
 Object user = request.getSession().getAttribute(**"loginUser"**);  
 **if**(user == **null**){  
 *//未登陆，返回登陆页面* request.setAttribute(**"msg"**,**"没有权限请先登陆"**);  
 request.getRequestDispatcher(**"/index.html"**).forward(request,response);  
 **return false**;  
 }**else**{  
 *//已登陆，放行请求* **return true**;  
 }  
  
 }

2将拦截器加入到容器（可以在一个继承可WebMvcConfigurerAdapter的配置类里面添加）

@Configuration  
**public class** myMvcConfig **extends** WebMvcConfigurerAdapter {  
  
 @Override

//添加拦截器  
 **public void** addInterceptors(InterceptorRegistry registry) {  
 *//静态资源； \*.css , \*.js  
 //SpringBoot已经做好了静态资源映射  
 //addPathPatterns要拦截的  
 //excludePathPatterns放行的* registry.addInterceptor(**new** loginIntersector()).addPathPatterns(**"/\*\*"**)  
 .excludePathPatterns( **"/"**, **"/user/login"**);  
 }

1. 错误处理（定制错误页面）

错误页面有两种，（springboot会通过请求头信息来判断）

1浏览器浏览时错误，这时会响应一个错误页面

2客户端错误，这时候返回的是一个json数据

如何定制错误的页面

有模板引擎（thymeleaf）的情况下；springboot会到error/下寻找错误状态码对应的页面，如发生404错误，则找404.html，如果找不到，则找4xx.Html，再没有则用系统默认的

​ 页面能获取的信息；

​ timestamp：时间戳

​ status：状态码

​ error：错误提示

​ exception：异常对象

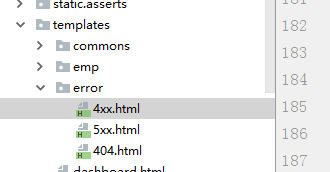
​ message：异常消息

​ errors：JSR303数据校验的错误都在这里

​ 2）、没有模板引擎（模板引擎找不到这个错误页面），静态资源文件夹下找；

​ 3）、以上都没有错误页面，就是默认来到SpringBoot默认的错误提示页面；

总结，对于自定义错误，对于浏览器端只需要在templates下夹一个error，然后加上错误页面，如404.Html（专门处理404错误）或者4xx.html(处理4开头的错误)，5xx（处理状态码5开头的错误）



如果需要错误信息，可以在页面上加上，来得到错误信息

<**h1**>状态码:[[${status}]]</**h1**>  
 <**h2**>时间戳:[[${timestamp}]]</**h2**>  
 <**h2**>错误提示:[[${error}]]</**h2**>  
 <**h2**>异常对象:[[${exception}]]</**h2**>  
 <**h2**>异常消息:[[${message}]]</**h2**>  
 <**h2**>JSR303数据校验的错误都在这里:[[${errors}]]</**h2**>

1. 客户端的错误如何定制json数据

只要一个类继承与DefaultErrorAttributes ，那么他可以自定义错误信息，如加上一些额外的信息，然偶将这个类扫描进入容器就可以

//给容器中加入我们自己定义的ErrorAttributes

@Component

public class MyErrorAttributes extends DefaultErrorAttributes {

@Override

public Map<String, Object> getErrorAttributes(RequestAttributes requestAttributes, boolean includeStackTrace) {

Map<String, Object> map = super.getErrorAttributes(requestAttributes, includeStackTrace);

map.put("company","atguigu");//定义自己的错误json

return map;

}

}

```

最终的效果：响应是自适应的，可以通过定制ErrorAttributes改变需要返回的内容，

1. 嵌入式的容器

Springboot默认以tomcat作为嵌入式的容器，也可修改为undertow和jetty（长连接）

1（如何修改容器的配置）

1. 在配置文件里面编写“sever.”（也是一个EmbeddedServletContainerCustomizer）的属性，进行修改

2、编写一个\*\*EmbeddedServletContainerCustomizer\*\*：嵌入式的Servlet容器的定制器；来修改Servlet容器的配置

```java

@Bean //一定要将这个定制器加入到容器中

public EmbeddedServletContainerCustomizer embeddedServletContainerCustomizer(){

return new EmbeddedServletContainerCustomizer() {

//定制嵌入式的Servlet容器相关的规则

@Override

public void customize(ConfigurableEmbeddedServletContainer container) {

container.setPort(8083);

}

};

2（如何将tomcat切换为jetty或者undertow）

打开项目结构图，去除tomcat部分，

引入别的容器的启动器，springboot会自动切换

3嵌入式tomcat如何完成创建

1. sprinboot根据容器里面的添加组建的情况，添加相应的tomcat、jetty、或者undertow（通过EmbeddedServletContainerFactory

【TomcatEmbeddedServletContainerFactory】 嵌入式容工厂类）

1. 一旦容器中创建了嵌入式容器类，马上就会触发专为该类编写的（EmbeddedServletContainerCustomizerBeanPostProcessor；嵌入式容器定制器后置处理器），然后利用这个后置处理器，就会去读取所有的EmbeddedServletContainerCustomizer，相当于获取在配置文件里所有sever开头的以及读取自己编写的所有的EmbeddedServletContainerCustomizer实现类，实现对嵌入式容器的配置（如端口号）

4嵌入式容器如何自启动

1. 、SpringBoot应用启动运行run方法
2. 、refreshContext(context);SpringBoot刷新IOC容器【创建IOC容器对象，并初始化容器，创建容器中的每一 个组件】；如果是web应用创建**AnnotationConfigEmbeddedWebApplicationContext**，否则： AnnotationConfigApplicationContext
3. refresh(context);刷新刚才创建好的ioc容器；并创建嵌入式容器（第一个被创建然后放入ioc的对象）
4. 从定制里面获取配置，完善容器
5. 嵌入式的Servlet容器创建对象并启动Servlet容器； 先启动嵌入式的Servlet容器，再将ioc容器中剩下没有创建出的

对象获取出来；

40使用外置容器启动（如自己装的tomcat）

1在嵌入式的容器中我们采用创建jar包就可以运行其优点实：简单、便携； 缺点：默认不支持JSP、优化定制比较复杂（使用定制器【ServerProperties、自定义 EmbeddedServletContainerCustomizer】，自己编写嵌入式Servlet容器的创建工厂 【EmbeddedServletContainerFactory】），而如果要使用外置处理器，则需要使用创建war包的方式（利用idea创建好目录结构）

2将嵌入式tomcat指定为provided

<dependency>

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

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

<scope>provided</scope>

</dependency>

```

3）、必须编写一个\*\*SpringBootServletInitializer\*\*的子类，并调用configure方法

```java

public class ServletInitializer extends SpringBootServletInitializer {

@Override

protected SpringApplicationBuilder configure(SpringApplicationBuilder application) {

//传入SpringBoot应用的主程序

return application.sources(SpringBoot04WebJspApplication.class);

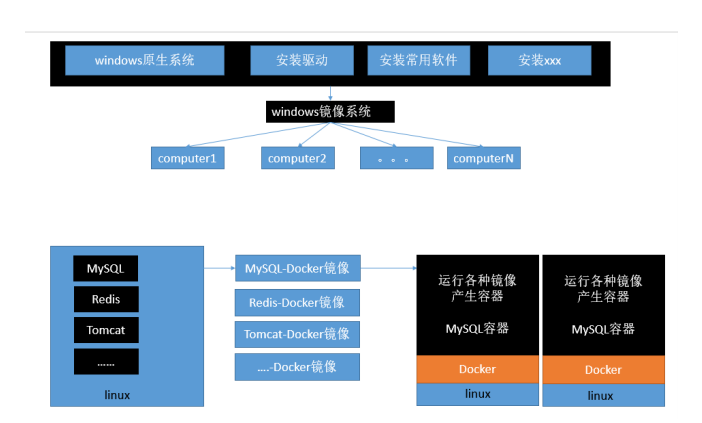
}

}

```

1. 、启动服务器，他会通过ServletInitializer 方法将整个项目带起来就可以使用；

41docker和；



主要运用是在服务器上安装docker镜像，然后就相当于安装（如mysql，redis。。。）好了很多软件，对于配置服务器有很大的便利，而且软件之间还能实现沙箱隔离。类似于window的ghost，一安装，什么都有了

42docker的核心概念

Docker主机:安装了docker的机器

Docker客户端：安装在本机的一个客户端，用于连接docker进行操作

Docker仓库：用于保存打包好的各种软件的镜像

Docker镜像：docker将软件打包，就是镜像，然后放在镜像仓库里

Docker容器：docker启动后的实例为一个容器，每一个镜像启动后都是一个容器，容器间沙箱隔离，因此可以同时运行多

相同的软件，如tomcat

Docker的使用步骤

1. 安装dockker
2. 去docker仓库找到软件对应的镜像
3. 使用docker运行镜像，docker会生成一个镜像容器，相当于这个软件也运行起来
4. 对容器的启动停止，就是对软件的启动停止

43linux的一些命令

Ip addr 查看ip地址

service network restart 重启服务器

Uname -r 查看内核版本

yum install xxxx：安装软件，还需输入y来确认，如yum install docker，yum install -y docker

Systemctl start docker ：启动docker ，Systemctl start xxxx表示启动某个软件

Docker -v 查看docker的版本号，xxxx -v表示查看某个已经启动的软件的版本号

Systemctl enable docker 开机自启动docker

Systemctl stop docker 停止docker

Ls 查看一个文件夹下面的所有文件

Vi ：编辑某个文件（q：不保存退出，wq：保存并退出）

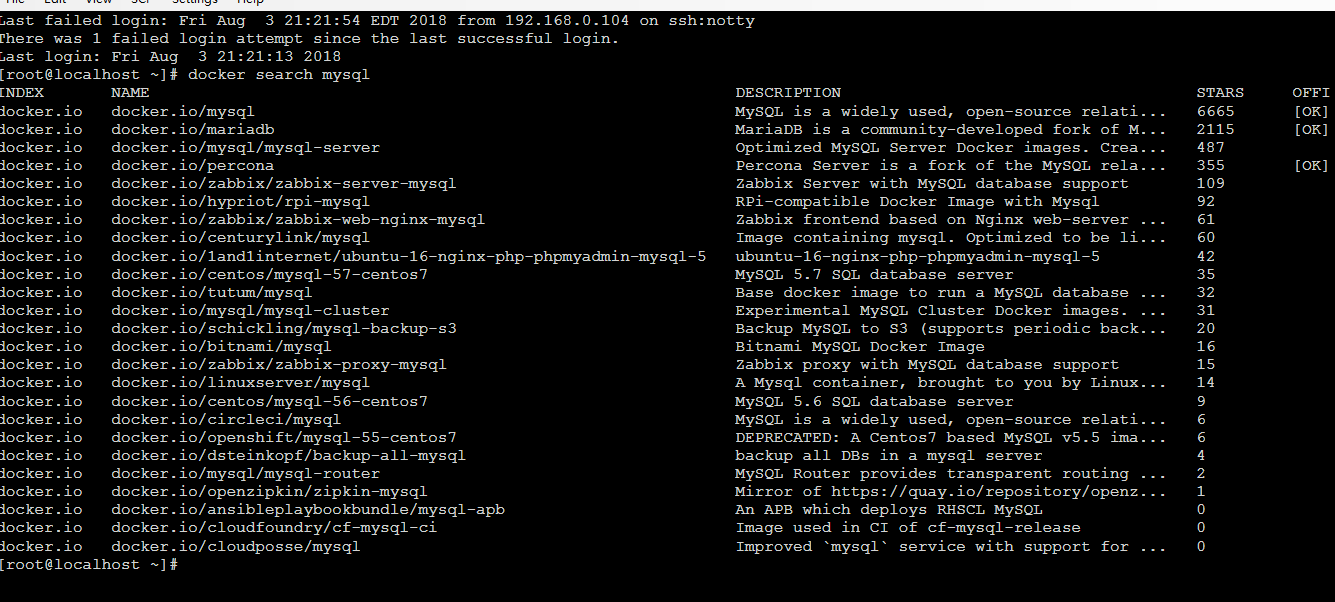
Mkdir 创键文件夹

Cd 打开某个文件夹

44 docker的一些命令

1、查找镜像文件

Docker search mysql :在docker仓库里面找mysql，其实就是在docker官网的仓库里面找mysql，



2、下载镜像

Docker pull + 镜像名+版本号（可不加，默认为least版）：如docker pull mysql：5.5

3、查看本地镜像

Docker images

4、删除本地镜像

Docker rmi image-id(镜像id)

1. 启动一个tomcat容器，（相当于运行了一个软件，并且多个容器间沙箱隔离）

docker run -d -p 8888:8080 --name tomcat01 tomcat

-d 表示后台运行

-p指定端口映射，将8888给它

--name 起别名，可要可不要

最后为镜像名称

1. 一个镜像可以启动多次，相当于运行了多个相同的软件（如再运行一个tomcat容器）

Docker run -d -p 8889:8080 --name tomcat02 tomcat

1. 启动mysql镜像，生成mysql容器

docker run -p 3306:3306 --name mysql01 -e MYSQL\_ROOT\_PASSWORD=123456 -d mysql

其中-e 表示指定参数 MYSQL\_ROOT\_PASSWORD=123456，表示指定密码为123456

8、查看docker 容器情况 docker ps -a

1. 进入容器(重要)

sudo docker exec -it 775c7c9ee1e1 /bin/bash

1. 进入mysql

mysql -uroot -p123456

45连接虚拟机数据库并使用最基本的jdbc来操作数据库

1连接数据库之需要在配置文件里面书写配置项，如在application.yml

**spring:  
 datasource:  
 username:** root  
 **password:** root  
 **url:** jdbc:mysql://192.168.0.102:3306/jdbc  
 **driver‐class‐name:** com.mysql.jdbc.Driver

springboot会自动读取配置为我们创建数据连接，并返回，除此之外，他还会往ioc容器里面注入jdbcTemplate

这是用最基本的jdbc的情况下

1. 参考DataSourceConfiguration，根据配置创建数据源，默认使用Tomcat连接池；
2. 可以使用 spring.datasource.type指定自定义的数据源类型；

3、SpringBoot默认可以支持；

org.apache.tomcat.jdbc.pool.DataSource、HikariDataSource、BasicDataSource、

4、springboot还可以自己读数据库文件进行创建数据库和插入数据

5 、DataSourceInitializer：ApplicationListener； 作用： 1）、runSchemaScripts();运行建表语句

1. 使用alibaba的druid作为数据源（能够监控sql、url、恶意访问、等）
2. 引入阿里巴巴的数据源

*<!--引入duird-->*<**dependency**>  
 <**groupId**>com.alibaba</**groupId**>  
 <**artifactId**>druid</**artifactId**>  
 <**version**>1.0.18</**version**>  
</**dependency**>

1. 在yml或properties中配置数据源，并且指定使用什么druid，和一些参数，建议赋值粘贴

**spring:  
 datasource:  
 username:** root  
 **password:** root  
 **url:** jdbc:mysql://192.168.0.102:3306/jdbc  
 **driver‐class‐name:** com.mysql.jdbc.Driver  
 **type:** com.alibaba.druid.pool.DruidDataSource//指定数据源使用druid  
  
 **initialSize:** 5  
 **minIdle:** 5  
 **maxActive:** 20  
 **maxWait:** 60000  
 **timeBetweenEvictionRunsMillis:** 60000  
 **minEvictableIdleTimeMillis:** 300000  
 **validationQuery:** SELECT 1 FROM DUAL  
 **testWhileIdle:** true  
 **testOnBorrow:** false  
 **testOnReturn:** false  
 **poolPreparedStatements:** true  
 *# 配置监控统计拦截的filters，去掉后监控界面sql无法统计，'wall'用于防火墙* **filters:** stat,wall  
 **maxPoolPreparedStatementPerConnectionSize:** 20  
 **useGlobalDataSourceStat:** true  
 **connectionProperties:** druid.stat.mergeSql=true;druid.stat.slowSqlMillis=500

1. 用配置类更换数据源，并且配置用来监控的servlet和filter

@Configuration  
**public class** datasourcesconfig {  
 @ConfigurationProperties(prefix = **"spring.datasource"**)  
 @Bean  
 **public** DataSource druid(){  
 **return new** DruidDataSource();  
 }  
  
 *//配置Druid的监控  
 //1、配置一个管理后台的Servlet* @Bean  
 **public** ServletRegistrationBean statViewServlet(){  
 ServletRegistrationBean bean = **new** ServletRegistrationBean(**new** StatViewServlet(), **"/druid/\*"**);  
 Map<String,String> initParams = **new** HashMap<>();  
  
 initParams.put(**"loginUsername"**,**"admin"**);  
 initParams.put(**"loginPassword"**,**"123456"**);  
 initParams.put(**"allow"**,**""**);*//默认就是允许所有访问* initParams.put(**"deny"**,**"192.168.15.21"**);  
  
 bean.setInitParameters(initParams);  
 **return** bean;  
 }  
  
  
 *//2、配置一个web监控的filter* @Bean  
 **public** FilterRegistrationBean webStatFilter(){  
 FilterRegistrationBean bean = **new** FilterRegistrationBean();  
 bean.setFilter(**new** WebStatFilter());  
  
 Map<String,String> initParams = **new** HashMap<>();  
 */\*静态资源文件不监控\*/* initParams.put(**"exclusions"**,**"\*.js,\*.css,/druid/\*"**);  
  
 bean.setInitParameters(initParams);  
 */\*设置监控的属性\*/* bean.setUrlPatterns(Arrays.*asList*(**"/\*"**));  
  
 **return** bean;  
 }  
}

1. 然后就可以从localhost：8080/druid访问监控程序
2. 整合mybatis

1无配置注解法

1. 在快捷创建项目的时候记得引入mybatis
2. 配置数据源
3. 创建实体类
4. 创建接口（实现类有mybatis自己做好），并将数据库语句直接写在方法上面（）

**public interface** DepartmentMapper {  
//他的流程是

//1由ctl获得数据传给方法，并完成实体类入参

//2方法穿给mybatis帮我们实现的方法（所以在这里打断点没用）

//3将参数从对象中取出给现在的参数，所以#{}里面的都是从对象里面去取数据  
 @Select(**"select \* from department where id=#{id}"**)  
 **public** Department getDeptById(Integer id);  
  
 @Delete(**"delete from department where id=#{id}"**)  
 **public int** deleteDeptById(Integer id);  
  
 @Options(useGeneratedKeys = **true**,keyProperty = **"id"**)  
 @Insert(**"insert into department(department\_name) values(#{departmentName})"**)  
 **public int** insertDept(Department department);  
  
 @Update(**"update department set department\_name=#{departmentName} where id=#{id}"**)  
 **public int** updateDept(Department department);  
}

1. 加上大小写转换，实现驼峰命名的配置类（将数据库中xx\_xx，与实体类的向匹配，相当于原来的resoultmap）

@org.springframework.context.annotation.Configuration  
**public class** MyBatisConfig {  
  
 @Bean  
 **public** ConfigurationCustomizer configurationCustomizer(){  
 **return new** ConfigurationCustomizer(){  
  
 @Override  
 **public void** customize(Configuration configuration) {  
 configuration.setMapUnderscoreToCamelCase(**true**);  
 }  
 };  
 }  
}

6、扫描所有mapper类，在主配置类上面加上扫描所有的mapper

@MapperScan(value = **"com.atguigu.springboot.mapper"**)  
@SpringBootApplication  
**public class** SpringBoot06DataMybatisApplication {  
  
 **public static void** main(String[] args) {  
 SpringApplication.*run*(SpringBoot06DataMybatisApplication.**class**, args);  
 }  
}

2使用配置文件整合没有mbatis

1. 配置数据源
2. 配置实体类
3. 写接口，

**public interface** EmployeeMapper {  
  
 **public** Employee getEmpById(Integer id);  
  
 **public void** insertEmp(Employee employee);  
}

1. 书写配置文件

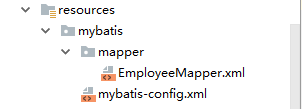
*<?***xml version="1.0" encoding="UTF-8"** *?>***<!DOCTYPE mapper  
 PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"  
 "http://mybatis.org/dtd/mybatis-3-mapper.dtd"*>***<**mapper namespace="com.atguigu.springboot.mapper.EmployeeMapper"**>  
 *<!-- public Employee getEmpById(Integer id);  
  
 public void insertEmp(Employee employee);-->* <**select id="getEmpById" resultType="com.atguigu.springboot.bean.Employee"**>  
 SELECT *\** FROM employee WHERE id=#{id}  
 </**select**>  
  
 <**insert id="insertEmp"**>  
 INSERT INTO employee(lastName,email,gender,d\_id) VALUES (#{lastName},#{email},#{gender},#{dId})  
 </**insert**>  
</**mapper**>

1. 在主配置文件里面加入驼峰命名转换

*<?***xml version="1.0" encoding="UTF-8"** *?>***<!DOCTYPE configuration  
 PUBLIC "-//mybatis.org//DTD Config 3.0//EN"  
 "http://mybatis.org/dtd/mybatis-3-config.dtd"*>***<**configuration**>  
  
 <**settings**>  
 <**setting name="mapUnderscoreToCamelCase" value="true"**/>  
 </**settings**>  
</**configuration**>

1. 在配置文件里面告诉springboot主配置文件和子配置文件的位置，如在yml文件中

**mybatis:** *# 指定全局配置文件位置* **config-location:** classpath:mybatis/mybatis-config.xml  
 *# 指定sql映射文件位置* **mapper-locations:** classpath:mybatis/mapper/\*.xml



1. 整合jpa
2. 创建项目的时候引入jpa
3. 配置数据源
4. 配置jpa规定（复合jpa标准）的实体类

@Entity  
@Table( name = **"user"**)  
@JsonIgnoreProperties(value={**"hibernateLazyInitializer"**,**"handler"**,**"fieldHandler"**})  
**public class** user {  
  
 @Id  
 @GeneratedValue(strategy = GenerationType.***IDENTITY***)*//自增策略* **private** Integer **id**;  
  
 @Column(length = 50)  
 **private** String **username**;  
  
 @Column(length = 50)  
 **private** String **password**;  
  
 **public** Integer getId() {  
 **return id**;  
 }  
  
 **public void** setId(Integer id) {  
 **this**.**id** = id;  
 }  
  
 **public** String getUsername() {  
 **return username**;  
 }  
  
 **public void** setUsername(String username) {  
 **this**.**username** = username;  
 }  
  
 **public** String getPassword() {  
 **return password**;  
 }  
  
 **public void** setPassword(String password) {  
 **this**.**password** = password;  
 }  
}

1. 书写一个接口继承于Repository<实体类，主键类型>就可以使用repository的方法，（不用实现，这样就可以了）  
   **public interface** userRespostry **extends** JpaRepository<user,Integer> {  
   }
2. 使用jpa(jpa已经帮我实现了很多的方法，直接使用就可以了)

@Controller  
**public class** userctl {  
 @Autowired  
 userRespostry **userrespostry**;*//可直接注入，因为本来就在容器里面* @ResponseBody  
 @RequestMapping(**"/getuser/{id}"**)  
 **public** user getuserbyid(@PathVariable(**"id"**) Integer id){  
 **return userrespostry**.getOne(id);  
 }  
  
 @RequestMapping(**"/deleteuserbyid/{id}"**)  
 @ResponseBody  
 **public int** deleteuserbyid(@PathVariable(**"id"**) Integer id){  
 **userrespostry**.deleteById(id);  
 **return** id;  
 }  
  
  
 @RequestMapping(**"/saveuser"**)  
 @ResponseBody  
 **public** user saveuser(user user){  
 user save = **userrespostry**.save(user);  
 **return** user;  
 }  
  
  
}

50springboot运行流程

Springboot运行流程为运行主方法，由创建SpringApplication对象和*run方法组成*我们从这里入手

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

1. 创建Springapplication对象

initialize(sources);

private void initialize(Object[] sources) {

//保存主配置类

if (sources != null && sources.length > 0)

{ this.sources.addAll(Arrays.asList(sources)); }

//判断当前是否一个web应用

this.webEnvironment = deduceWebEnvironment();

//从类路径下找到META‐INF/spring.factories配置的所有ApplicationContextInitializer；然后保存起 来 setInitializers((Collection) getSpringFactoriesInstances( ApplicationContextInitializer.class));

//从类路径下找到ETA‐INF/spring.factories配置的所有ApplicationListener

setListeners((Collection) getSpringFactoriesInstances(ApplicationListener.class));

//从多个配置类中找到有main方法的主配置类

this.mainApplicationClass = deduceMainApplicationClass();

public ConfigurableApplicationContext run(String... args)

{ StopWatch stopWatch = new StopWatch(); stopWatch.start(); }

主要是两步

1从所有的jar包META‐INF/spring.factories读取所有的ApplicationContextInitializer存起来，为后来创建容器做准备

2从类路径下找到ETA‐INF/spring.factories配置的所有ApplicationListener存起来，用来监控容器创建情况

1. 执行run方法

public ConfigurableApplicationContext run(String... args) {

StopWatch stopWatch = new StopWatch();

stopWatch.start();

ConfigurableApplicationContext context = null;

FailureAnalyzers analyzers = null; configureHeadlessProperty();

//获取SpringApplicationRunListeners；从类路径下META‐INF/spring.factories SpringApplicationRunListeners listeners = getRunListeners(args);

//回调所有的获取SpringApplicationRunListener.starting()方法 listeners.starting();

try {

//封装命令行参数

ApplicationArguments applicationArguments = new DefaultApplicationArguments( args);

//准备环境

ConfigurableEnvironment environment = prepareEnvironment(listeners, applicationArguments);

//创建环境完成后回调SpringApplicationRunListener.environmentPrepared()；表示环境准 备完成

Banner printedBanner = printBanner(environment);

//创建ApplicationContext；决定创建web的ioc还是普通的ioc

context = createApplicationContext(); analyzers = new FailureAnalyzers(context);

//准备上下文环境;将environment保存到ioc中；而且applyInitializers()；

//applyInitializers()：回调之前保存的所有的ApplicationContextInitializer的initialize方法

//回调所有的SpringApplicationRunListener的contextPrepared()；

prepareContext(context, environment, listeners, applicationArguments, printedBanner);

//prepareContext运行完成以后回调所有的SpringApplicationRunListener的contextLoaded（）；

//s刷新容器；ioc容器初始化（如果是web应用还会创建嵌入式的Tomcat）；Spring注解版

//扫描，创建，加载所有组件的地方；（配置类，组件，自动配置）

refreshContext(context);

//从ioc容器中获取所有的ApplicationRunner和CommandLineRunner进行回调

//ApplicationRunner先回调，CommandLineRunner再回调 afterRefresh(context, applicationArguments);

//所有的SpringApplicationRunListener回调finished方法

listeners.finished(context, null); stopWatch.stop();

if (this.logStartupInfo) {

new StartupInfoLogger(this.mainApplicationClass) .logStarted(getApplicationLog(), stopWatch);

}

//整个SpringBoot应用启动完成以后返回启动的ioc容器；

return context;

} catch (Throwable ex) {

handleRunFailure(context, listeners, analyzers, ex); throw new IllegalStateException(ex);

}

}