

SpringBoot

[项目打包为war包](#)

手动打war包：cd 项目目录

```
jar -cvf 项目名称.war *
```

解压war： **jar -xvf xxx.war**

[项目打包为jar包](#)

运行jar包：有配置可以看：激活指定profile节；和后面三节

```
java -jar springboot_config-0.0.1-SNAPSHOT.jar
```

SpringBoot简介

优点：

通俗的所：

简化Spring应用开发的一个框架

整个Spring技术栈的一个大整合

J2EE开发的一站式解决方案

- 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

微服务

2014年，martin fowler发表关于“微服务”的博客，生动的介绍了其设计思想和理念

微服务：架构风格

一个应用应该是一组小型服务；可以通过http的方式进行互通

每一个功能元素最终都是一个可独立替换和独立升级的软件单元；

SpringBoot-HelloWorld

Maven设置:

在maven内config中的setting配置文件中；添加以下标签，可管理jdk编译版本

```
<profile>
  <id>jdk-1.8</id>
  <activation>
    <activeByDefault>true</activeByDefault>
    <jdk>1.8</jdk>
  </activation>
  <properties>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
    <maven.compiler.compilerVersion>1.8</maven.compiler.compilerVersion>
  </properties>
</profile>
```

导入对应依赖:

```
<parent><!--父项目，管理版本-->
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.3.1.RELEASE</version>
</parent>
<dependencies>
  <dependency><!--springboot启动器-->
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
</dependencies>
```

编写主程序：启动SpringBoot应用

在类上标注，说明这是一个SpringBoot应用：@SpringBootApplication

编写主方法：main，启动Spring应用

```
SpringApplication.run(HelloWorldMainApplication.class, args);
```

编写相关Controller、Service、Dao....

```
@Controller
public class HelloController {
    @RequestMapping("/hello")
    @ResponseBody
    public String hello(){
        return "hello world";
    }
}
```

简化部署：把SpringBoot项目，打包为jar包

```
<!-- 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>
```

在右侧的maven工具中，点击package；即可打包一个jar包

可直接运行(window下): `java -jar 项目名.jar`

剖析Hello World项目

1、pom文件

父项目:

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.3.1.RELEASE</version>
</parent>
```

其仍有个父项目

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-dependencies</artifactId>
  <version>2.3.1.RELEASE</version>
</parent>
```

其内部的`properties`定义了几乎大部分依赖的版本

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)

springboot启动器

导入了web模块正常运行所依赖的组件 (版本由父项目掌控)

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

Spring Boot将所有的功能场景都抽取出来,做成一个个的**starters** (启动器),只需要在项目里面引入这些starter相关场景的所有依赖都会导入进来。要用什么功能就导入什么场景的启动器

主程序，主入口

@SpringBootApplication: 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;

组合注解:

```
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@SpringBootApplication
@EnableAutoConfiguration
@ComponentScan(
    excludeFilters = {@Filter(
        type = FilterType.CUSTOM,
        classes = {TypeExcludeFilter.class}
    )}, @Filter(
        type = FilterType.CUSTOM,
        classes = {AutoConfigurationExcludeFilter.class}
    )}
)
```

@SpringBootApplication:SpringBoot的配置类

标注在某个类上，表示这是一个SpringBoot的配置类

@Configuration：配置类上标注这个注解

配置类——>配置文件；配置类也是容器中的一个组件@Component

@EnableAutoConfiguration：开启自动配置功能

之前需要配置的东西，SpringBoot会自动配置；需要在类上标注这个注解上，自动配置才会生效

```
@AutoConfigurationPackage
@Import({AutoConfigurationImportSelector.class})
public @interface EnableAutoConfiguration {
```

@AutoConfigurationPackage：自动配置包

@Import({Registrar.class}):

spring底层注解@Import，给容器中导入一个组件；导入的组件由Registrar.class指定，应导入那个组件

将主配置类(@SpringBootApplication标注的类)的所在包，及其子包里面的所有组件扫描到Spring容器

image-20200704181053095

image-20201015180008192

@Import({AutoConfigurationImportSelector.class}): 给容器中导入组件

将所有需要导入的组件以全类名的方式返回；这些组件会被添加到容器中

会给容器中导入大量的自动配置类(xxxAutoConfiguration)，就是给容器中导入这个给场景需要的所有组件，并配置好这些组件

有了自动配置类，则免去了手动编写注入功能组件等...

配置都存在于以下配置文件

image-20200704194757931

```
==SpringFactoriesLoader.loadFactoryNames(EnableAutoConfiguration.class, classLoader);==
```

SpringBoot在启动的时候，从类路径下的META-INF/spring.factories中获取EnableAutoConfiguration指定的值；将这些值作为自动配置类导入到容器中，自动配置类就生效；之前需要手动配置的东西，自动配置类都会解决

J2EE的整合解决方案和自动配置都在spring-boot-autoconfigure-2.3.1.RELEASE.jar

使用Spring Initializr快速创建SpringBoot项目

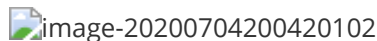
IDE都支持使用Spring的项目创建向导快速创建一个Spring Boot项目; (需要联网)



其中Artifact不能有大小写



选择模块web、sql、NoSQL、微服务等模块...



选择我们需要的模块;向导会联网创建Spring Boot项目;

默认生成的Spring Boot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 ,默认不支持JSP页面);可以使用模板引擎(freemarker. thymeleaf);
 - application.properties : Spring Boot应用的配置文件;可以修改一些默认设置

SpringBoot的配置文件

Spring Boot使用一个全局的配置文件 (共三种格式:properties、yml、yaml) ,配置文件名是固定的

- application.properties
- application.yml

.yml是YAML (YAML Ain't Markup Language) 语言的文件,以**数据为中心**,比json、xml等更适合做配置文件

YAML A Markup Language :是一个标记语言

YAML. isn't Markup Language :不是一个标记语言;

[参考语法规范](#)

标记语言:

以前的配置文件;大多都使用的是xxxx.xml文件;

配置文件放在src/main/resources目录或者类路径/config下

全局配置文件的可以对一些默认配置值进行修改(SpringBoot在底层都给我们自动配置好了)

yml配置案例:

```
server:
  port: 8082
```

XMI

```
<server>
  <port>8081</port>
</server>
```

YML语法:

基本语法:

K:(空格)V : 表示一对键值对 (空格不能少)

以空格的缩进来控制层级关系; 只要是左对齐的一系列数据, 都是同一级的

属性和值也是大小写敏感

```
server:
  port: 8082
  path: /hello
```

值的写法:

字面值: 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: 在下一行写对象的属性和值的关系; 注意缩进

对象仍然是k: v的方式

```
friends:
  astName: zhangsan
  age: 20
```

行内写法:

```
friends:{lastName: zhangsan,age: 18}
```

数组 (List、Set)

用- (空格)值表示数组中的一个元素

```
pets:
- cat
- dog
- pig
```

行内写法:

```
pets: [cat,dog,pig]
```

配置文件值的注入:

配置文件

```
server:
  port: 8082

person:
  lastname: zhangsan
  age: 18
  boss: false
  birth: 2017/12/12

#行内写法
maps: {k1: v1, k2: v2}
lists:
  - lisi
  - zhao1iu

dog:
  name: 小狗
```



```
age: 2
```

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j进行一一映射
 */
@Component //声明将这个组件添加至容器中
@ConfigurationProperties(prefix = "person") //只有这个组件是容器中的组件，才能使用容器提
供的@ConfigurationProperties功能
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;
```

```
@RunWith(SpringRunner.class) //使用spring的启动器，而不是junit
@SpringBootTest //springboot的单元测试
class SpringbootConfigApplicationTests {

    @Autowired
    Person person;

    @Autowired
    ApplicationContext ioc;

    @Test
    public void testHelloService(){//测试容器中，是否存在helloService类
        boolean f = ioc.containsBean("helloService");//对应MyAppConfig类的方法名
        //没在主方法的类上写@ImportResource注解，则是false----写了此注解，且写了配置文件的
        路径，则为true
        System.out.println(f);
        //config包下的MyAppConfig类中添加的@Configuration和@Bean注解；结果返回也是true
    }

    @Test
    public void contextLoads() {
        System.out.println(person);
        /**
        在配置文件中，使用了占位符的结果
        Person{lastname='张三6a07b752-7ccf-4baa-9ba8-2932685969d1',
        age=-584104987, boss=false, birth=Fri Dec 15 00:00:00 CST 2017, maps={k1=v1,
        k2=v2}, lists=[a, b, c], dog=Dog{name='张三ac3ee9bd-c52a-4999-9c60-
        522ece5aceb2_dog', age=5}}
        */
    }

}
```

可以导入配置文件处理器，以后编写配置就有提示；所需依赖

```
<!--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><!--确保
ConfigurationProperties注解所在类，不爆红-->
    <artifactId>spring-boot-configuration-processor</artifactId>
    <optional>true</optional>
</dependency>
```

idea中获取yml文件的值，乱码解决方案

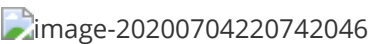
当yml文件出现乱码问题时（idea默认使用的是utf-8；需要在File Encodings设置utf-8，在把后面的勾选中），使用以下设置



@Value获取值和@ConfigurationProperties获取值比较

	@ConfigurationProperties	@Value
功能	批量注入配置文件中的属性	一个个指定
松散绑定(松散语法)	支持	不支持
(#{表达式})	不支持	支持
JSR303数据校验(@Validated+@Email)	支持	不支持
复杂类型封装(集合或数组...)	支持	不支持

松散绑定



配置文件yml还是properties他们都能获取到值;

如果说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

配置文件注入值数据校验

```
@Component
//声明将这个组件添加至容器中
```

```

@ConfigurationProperties(prefix = "person") //只有这个组件是容器中的组件，才能使用容器提供的@ConfigurationProperties功能
@Validated
public class Person {

    /**
     *类似spring里面的
     * <bean class= "Person " )
     * <property name= "lastname" value= "字面量/${key}从环境变量、配置文件中获取值/#
{SpEL}"></property>
     * <bean/>
     */
    //value注解: Person{lastname='张三', age=22, boss=true, birth=null, maps=null,
lists=null, dog=null}
    //@Value("${person.lastname}")
    @Email //lastname必须是邮箱格式
    private String lastname;
    @Value("#{11*2}")//Person{lastname='null', age=22, boss=null, birth=null,
maps=null, lists=null, dog=null}
    private Integer age;
    //@Value("true")
    private Boolean boss;
    private Date birth;

    private Map<String, Object> maps;
    private List<Object> lists;
    private Dog dog;

```

@PropertySource&@ImportResource&@Bean

@PropertySource(value = "classpath: 配置文件路径(person.properties)": 放置在实体类上，读取指定的配置文件

```

@PropertySource(value = {"classpath:person.properties"}) //后缀只能是
properties，且冒号后面不能加空格
@Component //声明将这个组件添加至容器中
@ConfigurationProperties(prefix = "person") //只有这个组件是容器中的组件，才能使用容器提供的@ConfigurationProperties功能
//@Validated //JSR303数据校验，不是本类需要校验
public class Person {

    /**
     *类似spring里面的
     * <bean class= "Person " )
     * <property name= "lastname" value= "字面量/${key}从环境变量、配置文件中获取值/#
{SpEL}"></property>
     * <bean/>
     */
    //value注解: Person{lastname='张三', age=22, boss=true, birth=null, maps=null,
lists=null, dog=null}

```

```

    //@Value("${person.lastname}")
    //@Email    //lastname必须是邮箱格式
    (org.hibernate.validator.constraints.Email;中)
    private String lastname;
    //@Value("#{11*2}")//Person{lastname='null', age=22, boss=null, birth=null,
    maps=null, lists=null, dog=null}
    private Integer age;
    //@Value("true")
    private Boolean boss;
    private Date birth;

    //@Value("${person.maps}")
    private Map<String, Object> maps;
    private List<Object> lists;
    private Dog dog;

```

@ImportResource（不推荐）：导入spring的配置文件，让配置文件的内容生效

Spring Boot里面没有Spring的配置文件,我们自己编写的配置文件,也不能自动识别;

要让spring的配置文件生效，加载进来；

@ImportResource(locations = {"classpath:beans.xml"}): 标注在配置类上，location写数组(配置文件所在路径)

```

@ImportResource(locations = {"classpath:beans.xml"})
@SpringBootApplication
public class SpringbootConfigApplication {

```

不推荐编写spring配置文件

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="helloService" class="com.fisir.service.HelloService"></bean>

</beans>

```

@Bean注解：代替@ImportResource注解，可以放在方法上，也可以放在类上；作为配置文件使用

springboot推荐给容器添加组件的方式；推荐使用**全注解**的方式：

1、配置类————spring配置文件

2、**使用@Bean给容器中添加组件**

```

/**
 * @Configuration指明当前类是一个配置类，用来替代之前的spring配置文件(beans.xml)
 *
 * 在配置文件中，使用<bean></bean>添加组件；配置类中使用@Bean注解
 */

```

```
@Configuration
public class MyAppConfig {

    @Bean          //将方法的返回值，添加到容器中；容器中组件的默认id就是方法名
    public HelloService helloService(){
        System.out.println("配置类@Bean给容器中添加组件了");
        return new HelloService();
    }
}
```

```
@Test
public void testHelloService(){//测试容器中，是否存在helloService类
    boolean f = ioc.containsBean("helloService");//对应MyAppConfig类的方法名
    //没在主方法的类上写@ImportResource注解，则是false----写了此注解，且写了配置文件的路径，则为true
    System.out.println(f);
    //config包下的MyAppConfig类中添加的@Configuration和@Bean注解；结果返回也是true
}
```

配置文件占位符

1、随机数

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

2、占位符获取之前配置的值，如果没有，可以用":"指定默认值

```
#名字是：张三+uuid
person.lastname=张三${random.uuid}

#person.age=#{11*2}      ConfigurationProperties注解，不支持SpEL表达式
#年龄是：18+int随机数
person.age=${random.int}
person.birth=2017/12/15
person.boss=false
person.maps.k1=v1
person.maps.k2=v2
person.lists=a,b,c
#配置的名字+狗的名字
#person.dog.name=${person.lastname}_dog

#配置的名字不存在的情况，那么就原样输出占位符中的内容
#person.dog.name=${person.hello}_dog
#Person{lastname='张三e4d1be70-aa2b-456c-a325-c468cecf2af4', age=-1054022999,
boss=false, birth=Fri Dec 15 00:00:00 CST 2017, maps={k1=v1, k2=v2}, lists=[a, b,
c], dog=Dog{name='${person.hello}_dog', age=5}}

#配置的名字不存在的情况，给出指定的值
person.dog.name=${person.hello:hello}_dog
```

```
#Person{lastname='张三e4d1be70-aa2b-456c-a325-c468cecf2af4', age=-1054022999,
boss=false, birth=Fri Dec 15 00:00:00 CST 2017, maps={k1=v1, k2=v2}, lists=[a, b,
c], dog=Dog{'hello_dog', age=5}}
```

```
person.dog.age=5
```

profile

1、多profile文件(测试/开发/生产环境)

在配置文件编写时，文件名可以是 application-{profile}.properties

默认使用application.properties的配置

 image-20200708215048295

激活指定环境：在application.properties文件中

```
spring.profiles.active=dev
```

2、yml支持多文档块方式

```
#修改springboot内置tomcat的端口号
server:
  port: 8082

person:
  lastname: zhangsan
  age: 18
  boss: false
  birth: 2017/12/12

#行内写法
maps: {k1: v1, k2: v2}
lists:
  - lisi
  - zhaoliu

dog:
  name: 小狗
  age: 2
spring:
  profiles:
    active: prod      #激活指定环境
---
#三个-，隔离一个文档块,开发环境
server:
  port: 8083
spring:
```


```
profiles: dev
---
#生成环境
server:
  port: 8084
spring:
  profiles: prod
```

3、激活指定profile

注意:

优先级: 命令行优先级>虚拟机>application配置文件配置

1. 在配置文件(yml或properties文件)中指定: spring.profiles.active=dev
2. 在命令行中指定: --spring.profiles.active=dev

image-20200708131325506

也可以打包后, 在命令行内, 指定

```
cd D:\java_project\VM\shangguigu\springboot_config\target
D:

java -jar springboot_config-0.0.1-SNAPSHOT.jar --spring.profiles.active=dev
```

注意:

编译环境时的jdk, 要和电脑中的jdk环境版本一致; 否则会报异常

```
at java.lang.ClassLoader.defineClass1(Native Method)
at java.lang.ClassLoader.defineClass(Unknown Source)
at java.security.SecureClassLoader.defineClass(Unknown Source)
at java.net.URLClassLoader.defineClass(Unknown Source)
at java.net.URLClassLoader.access$100(Unknown Source)
at java.net.URLClassLoader$1.run(Unknown Source)
at java.net.URLClassLoader$1.run(Unknown Source)
at java.security.AccessController.doPrivileged(Native Method)
at java.net.URLClassLoader.findClass(Unknown Source)
at java.lang.ClassLoader.loadClass(Unknown Source)
at
org.springframework.boot.loader.LaunchedURLClassLoader.loadClass(LaunchedURLClass
Loader.java:151)
at java.lang.ClassLoader.loadClass(Unknown Source)
at java.lang.Class.forName0(Native Method)
at java.lang.Class.forName(Unknown Source)
at
org.springframework.boot.loader.MainMethodRunner.run(MainMethodRunner.java:46)
at org.springframework.boot.loader.Launcher.launch(Launcher.java:109)
at org.springframework.boot.loader.Launcher.launch(Launcher.java:58)
```

2. jvm参数: -Dspring.profiles.active=dev

 image-20200708214607349

配置文件加载位置

spring boot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/(项目根目录下的config下的配置文件)
- file: ./(项目根目录下的配置文件)
- classpath: /config(resources目录下的config目录的配置文件)
- classpath: /(resources目录下的配置文件)

以上是按照**优先级从高到低**的顺序,所有位置的文件都会被加载,**高优先级配置内容会覆盖低优先级配置内容**。会**互补位置**(没有覆盖的,则会用低优先级的配置; parent版本太高,则不行); 目测2.3.1想不通,而1.5.9可以

可以通过**配置spring.config.location来改变默认配置**

但是,得项目打包后; 使用命令行参数的形式; 启动项目时, 指定配置文件的新位置; 指定配置文件和默认加载的配置文件, 会一起启动, 形成互补配置

```
java -jar springboot_config_02-0.0.1-SNAPSHOT.jar --  
spring.config.location=D:\java_project\vm\shangguigu\test-spring-  
boot_config_02\application.properties
```

=后面是配置文件所在路径, 及文件本身的名字

注意:

前提得和编写时的jdk版本一致, 否则报异常

```
Exception in thread "main" java.lang.UnsupportedClassVersionError:  
com/fsir/SpringbootConfig02Application has been compiled by a more recent versi  
on of the Java Runtime (class file version 55.0), this version of the Java  
Runtime only recognizes class file versions up to 52.0
```

```
at java.lang.ClassLoader.defineClass1(Native Method)
```



```
at java.lang.ClassLoader.defineClass(Unknown Source)
at java.security.SecureClassLoader.defineClass(Unknown Source)
at java.net.URLClassLoader.defineClass(Unknown Source)
at java.net.URLClassLoader.access$100(Unknown Source)
at java.net.URLClassLoader$1.run(Unknown Source)
at java.net.URLClassLoader$1.run(Unknown Source)
at java.security.AccessController.doPrivileged(Native Method)
at java.net.URLClassLoader.findClass(Unknown Source)
at java.lang.ClassLoader.loadClass(Unknown Source)
at
org.springframework.boot.loader.LaunchedURLClassLoader.loadClass(LaunchedURLClass
sLoader.java:151)
at java.lang.ClassLoader.loadClass(Unknown Source)
at java.lang.Class.forName0(Native Method)
at java.lang.Class.forName(Unknown Source)
at
org.springframework.boot.loader.MainMethodRunner.run(MainMethodRunner.java:46)
at org.springframework.boot.loader.Launcher.launch(Launcher.java:109)
at org.springframework.boot.loader.Launcher.launch(Launcher.java:58)
at org.springframework.boot.loader.JarLauncher.main(JarLauncher.java:88)
```

外部配置加载顺序

[优先级参考](#)

SpringBoot也可以从以下位置加载配置;按照优先级, 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
```

可以写一个配置, 也可以像上面那样, 写多个; 多个以空格隔开, 加--配置信息
springboot2.0之后, 使用server.servlet.context-path=/boot02; 指定虚拟路径

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profile

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

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

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

自动配置原理

[配置文件能配置的属性参照\(第10条\)](#)

[视频地址](#)

自动配置原理:

1、springboot启动时，加载主配置类，开启了自动配置注解@EnableAutoConfiguration

2、@EnableAutoConfiguration作用:

利用AutoConfigurationImportSelector给容器中，导入一些组件

可以查看其内的selectImports方法的内容

```
SpringFactoriesLoader.loadFactoryNames()
```

扫描所有jar包类路径下 META-INF/spring.factories

把扫描到的这些文件的内容包装成properties对象

从properties中获取到EnableAutoConfiguration.class类（类名）对应的值，然后把他们添加在容器中

```
AutoConfigurationImportSelector.AutoConfigurationEntry getAutoConfigurationEntry  
方法下
```

```
Set<String> exclusions = this.getExclusions(annotationMetadata, attributes);
```

```
返回: new AutoConfigurationImportSelector.AutoConfigurationEntry(configurations,  
exclusions);
```

其内的getCandidateConfigurations方法的

```
SpringFactoriesLoader.loadFactoryNames(this.getSpringFactoriesLoaderFactoryClasses(),  
this.getBeanClassLoader());
```

从类路径下获得

loadSpringFactories方法内

```
classLoader.getResources("META-INF/spring.factories") :  
ClassLoader.getSystemResources("META-INF/spring.factories");  
扫描所有jar包，类路径下的META-INF/spring.factories
```

把扫描到的这些文件的内容包装成properties对象

传入的类名，在AutoConfigurationImportSelector.class的getCandidateConfigurations方法，再调用其内的loadFactoryNames方法，其内的形参getSpringFactoriesLoaderFactoryClass方法得到，对于的值

从properties中获取到EnableAutoConfiguration.class类(类名)对应的值，然后把他们添加在容器中

将类路径下META-INF/spring.factories 里面配置的所有EnableAutoConfiguration的值加入到了容器中

其内容在spring-boot-autoconfigure-2.3.1.RELEASE.jar中META-INF的spring.factories中Auto Configure

每一个这样的xxxAutoConfiguration类都是容器中的一个组件,都加入到容器中；用来做自动配置

3、每一个自动配置类进行自动配置功能

例如：HttpEncodingAutoConfiguration：Http编码配置，为解决乱码问题

```
//类上的注解  
@Configuration(      //表示这是一个配置类，类似把Java文件编写为配置文件一样，可以给容器中添加  
组件  
    proxyBeanMethods = false  
)  
//启用ConfigurationProperties功能  
//ConfigurationProperties功能;将配置文件中对应的值和HttpEncodingProperties绑定起来;并把  
ServerProperties加入到ioc容器中  
@EnableConfigurationProperties({ServerProperties.class})  
//spring底层的@Conditional注解;作用：根据不同条件，可以根据不同条件，可以自己条件判断；如果满  
足指定的条件，那么整个配置类里面的配置，才会生效；判断当前应用，是否是web应用，是，则生效  
@ConditionalOnWebApplication(  
    type = Type.SERVLET  
)  
//判断当前项目，有没有CharacterEncodingFilter类；springmvc中进行乱码解决的过滤器；判断有没有  
整个过滤器  
@ConditionalOnClass({CharacterEncodingFilter.class})  
//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  
//即使配置文件中，不配置spring.http.encoding.enabled=true，也是默认生效  
@ConditionalOnProperty(  

```

```

        prefix = "server.servlet.encoding",
        value = {"enabled"},
        matchIfMissing = true
    )

    public class HttpEncodingAutoConfiguration {
        //已经和springboot的配置文件映射了
        private final Encoding 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(ServerProperties properties) {
            this.properties = properties.getServlet().getEncoding();
        }

        @Bean          //给容器中，添加一个组件(CharacterEncodingFilter),这个组件的某些值，需要从properties中获取
        @ConditionalOnMissingBean
        public CharacterEncodingFilter characterEncodingFilter() {

            CharacterEncodingFilter filter = new OrderedCharacterEncodingFilter();
            filter.setEncoding(this.properties.getCharset().name());

            filter.setForceRequestEncoding(this.properties.shouldForce(org.springframework.boot.web.servlet.server.Encoding.Type.REQUEST));

            filter.setForceResponseEncoding(this.properties.shouldForce(org.springframework.boot.web.servlet.server.Encoding.Type.RESPONSE));

            return filter;
        }
    }

```

根据不同的条件判断，决定这个配置类，是否生效

一但这个配置类生效;这个配置类就会给容器中添加各种组件;这些组件的属性是从对应的properties类中获取的,这些类里面的每一个属性又是和配置文件绑定的;

所有在配置文件中能配置的属性都是在xxxxProperties类中封装着；配置文件能配置什么就可以参照某个功能对于的属性类

```

@ConfigurationProperties(//从配置文件中获取指定的值和bean的属性进行绑定
    prefix = "server",
    ignoreUnknownFields = true
)
public class ServerProperties {

```

精髓：

- 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的属性

@Conditional派生注解

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;

底层都是@Conditional原理

@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;

如果需要知道, 哪些配置类生效: 可以在配置文件启用debug=true属性; 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

Positive matches: 已生效的自动配置类

Negative matches: 没有匹配的自动配置类

Spring Boot与日志

小张；开发一个大型系统；

- 1、System.out.println(""); 将关键数据打印在控制台；去掉？ 写在一个文件？
- 2、框架来记录系统的一些运行时信息；日志框架； zhanglogging.jar；
- 3、高大上的几个功能？异步模式？自动归档？xxxx？ zhanglogging-good.jar？
- 4、将以前框架卸下来？换上新的框架，重新修改之前相关的API； zhanglogging-prefect.jar；
- 5、JDBC---数据库驱动；

写了一个统一的接口层；日志门面（日志的一个抽象层）； logging-abstract.jar；

给项目中导入具体的日志实现就行了；我们之前的日志框架都是实现的抽象层；

市面上的日志框架；

JUL、JCL、Jboss-logging、logback、log4j、log4j2、slf4j....

日志门面（日志的抽象层）	日志实现
JCL (Jakarta Commons Logging) (最后2014年更新的) SLF4j (Simple Logging Facade for Java) jboss-logging (用的很少)	Log4j JUL (java.util.logging) Log4j2 Logback

左边选一个门面（抽象层）、右边来选一个实现；

日志门面： SLF4j；

日志实现： Logback；

SpringBoot :底层是Spring框架, Spring框架默认是用JCL；

SpringBoot选用SL F4j和logback；

SLF4j使用

1、在系统中使用SLF4j

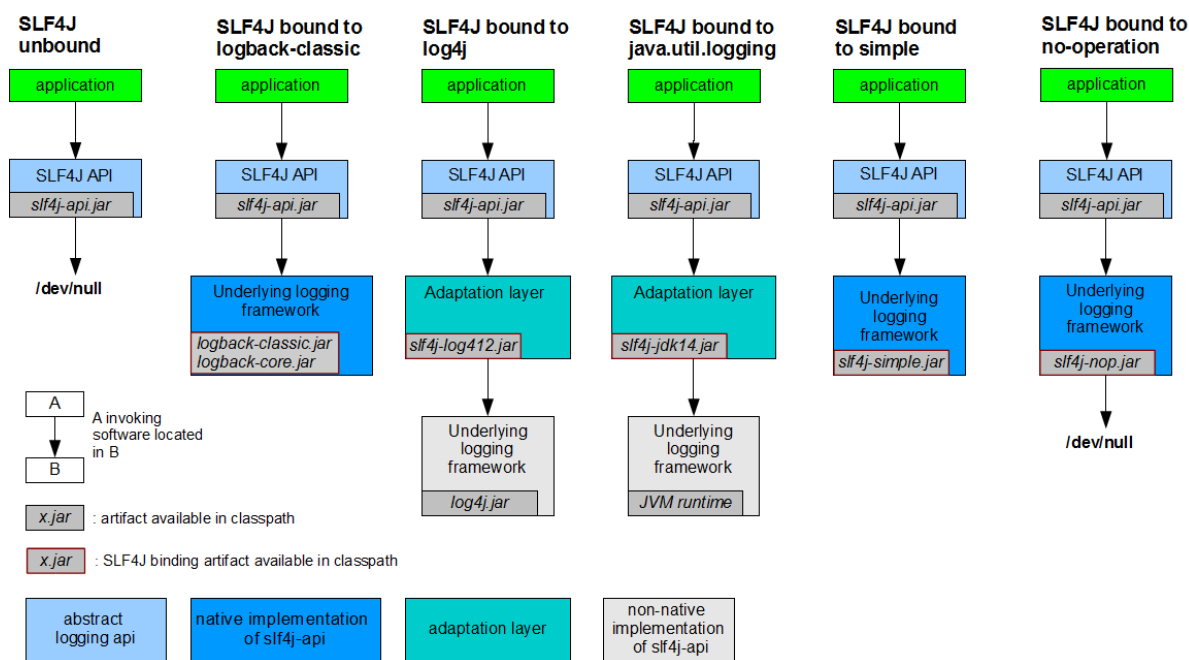
开发的时候,日志记录方法的调用,不应该来直接调用日志的实现类,而是调用日志抽象层里面的方法;

给系统里面导入slf4j的jar和logback的实现jar

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

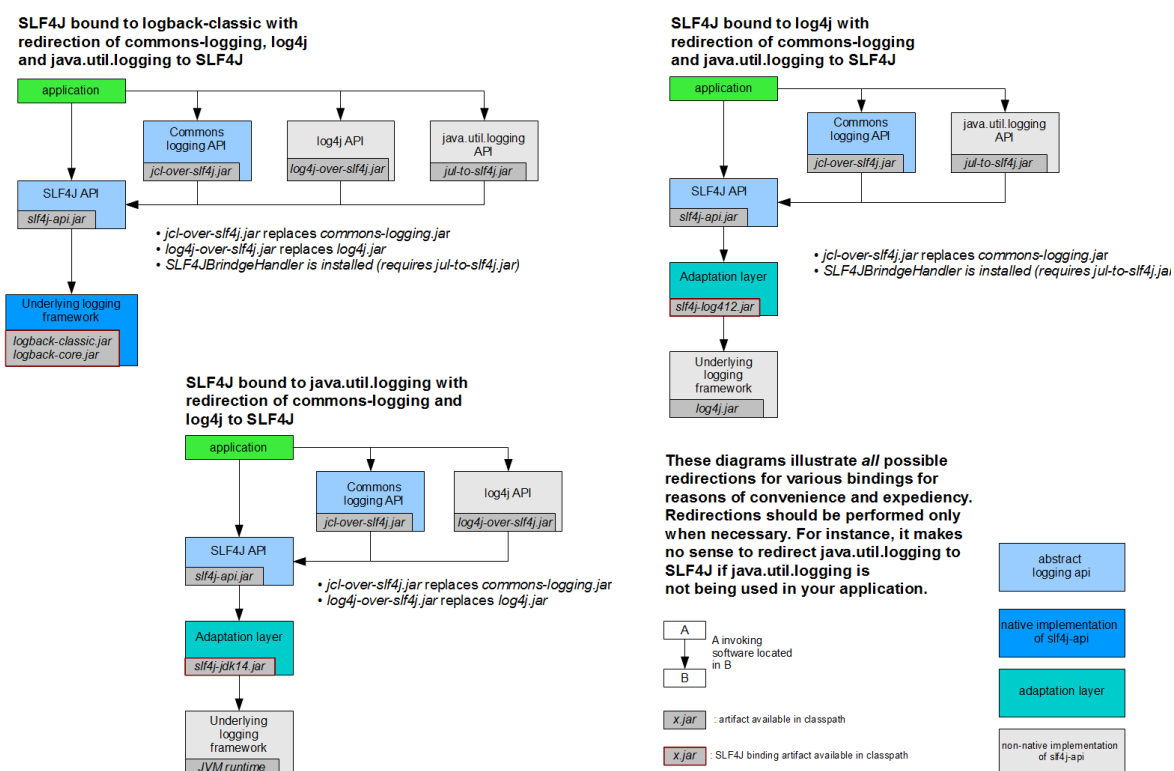
public class HelloWorld {
    public static void main(String[] args) {
        Logger logger = LoggerFactory.getLogger(HelloWorld.class);
        logger.info("Hello world");
    }
}
```

图示：



每一个日志的实现框架都有自己的配置文件。使用slf4j以后,配置文件还是做成日志实现框架的配置文件;

a 系统(slf4j+logback):Spring (commons-logging)、Hibernate (jbosslogging)、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



让系统中所有的日志都统一到slf4j ;

- 1、将系统中其他日志框架先排除出去;
- 2、用中间包来替换原有的日志框架;
- 3、导入slf4j其他的实现

SpringBoot日志关系

最基本的依赖：启动器

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter</artifactId>
    <version>2.3.1.RELEASE</version>
    <scope>compile</scope>
</dependency>
```


springboot的日志，由以下依赖实现日志功能

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-logging</artifactId>
    <version>2.3.1.RELEASE</version>
    <scope>compile</scope>
</dependency>
```

1、SpringBoot底层也是使用slf4j+logback的方式进行日志记录

2、SpringBoot也把其他的日志都替换成了slf4j；

3、中间替换包（应该是上一个jar包）

 image-20200713125705780

4、如果我们要引入其他框架?一定要把这个框架的默认日志依赖移除掉

Spring框架用的是commons-logging；

SpringBoot能自动适配所有的日志,而且底层使用slf4j+logback的方式记录日志，引入其他框架的时候，只需要把这个框加依赖的日志框架排除掉即可

日志使用

[官方文档链接](#)

1、默认配置

springboot默认已经配置好了日志

```
package com.fsir;

import org.junit.jupiter.api.Test;
import org.junit.runner.RunWith;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;
```



```
//等同于@RunWith(SpringJUnit4ClassRunner.class):让测试运行于Spring测试环境，以便在测试
开始的时候自动创建Spring的应用上下文
@RunWith(SpringRunner.class)
@SpringBootTest
class SpringbootLoggingApplicationTests {

    //记录器
    Logger logger = LoggerFactory.getLogger(getClass());

    @Test
    void contextLoads() {

        //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及以后的高级别生效
        logger.trace("这是trace日志...");//跟踪
        logger.debug("这是debug日志...");//debug调试

        //springboot默认使用的是infoj级别，没有指定级别就使用springbootm默认规定的级别：
        root级别
        logger.info("这是info日志...");//信息
        logger.warn("这是warn日志...");//警告
        logger.error("这是error日志...");//报错
    }

}
```

基本配置

```
#设置日志的级别
logging.level.com.fsir=trace

#如果path和file同时指定，那么起作用的是file；一般指定path
#没指定在控制台输出；指定了就输出到指定目录的spring.log文件内
#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
#logging.file.path=/spring/log

#没指定文件在控制台输出；指定了就在当前项目下输出到指定文件内，
# 前面加上路径就在指定路径下的指定输出到文件内；此版本直接写：logging.file没效果
#logging.file.name=D:/springboot.log

#在控制台输出的日志的格式      日期                线程名        日志等级        启动类所在的包名.启动
类名称  消息及换行
#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

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
Process ID  进程id
-->
%d{yyyy-MM-dd HH:mm:ss.SSS} [%thread] %-5level %logger{50} - %msg%n
```

logging.file	logging.path	Example	Description
(none)	(none)		只在控制台输出
指定文件名	(none)	my.log	输出日志到my.log文件
(none)	指定目录	/var/log	输出到指定目录的 spring.log 文件中

2、指定配置

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

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

logback.xml: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profiles功能

```
<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>
```

```
<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"><!--开发环境下执行此(例:logback.xml)-->
```

```

        <pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} ---> [%thread] --->
%-5level %logger{50} - %msg%n</pattern>
    </springProfile>
    <springProfile name="!dev"><!--非开发环境下执行此(例:logback.xml)-->
        <pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} === [%thread] === %-5level
%logger{50} - %msg%n</pattern>
    </springProfile>
</layout>
</appender>

```

配置文件指向的dev则输出--->的语句；反之则输出===的语句；(2.3.1版本)就算是默认的logback配置，也是=

当然低版本可能会报异常

```
no applicable action for [springProfile]
```

切换日志框架

可以按照slf4j的日志适配图,进行相关的切换;

slf4j+log4j的方式;

需要注意的是

Spring Boot 只有1.3.x和1.3.x以下版本才支持log4j的日志配置，1.3.x以上版本只支持log4j2。

```

<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
  <exclusions><!-- 需要排除的默认日志logback和把log4j转换为slf4j的jar包-->
    <exclusion>
      <groupId>logback-classic</groupId>
      <artifactId>ch.qos.logback</artifactId>
    </exclusion>
    <exclusion>
      <groupId>log4j-over-slf4j</groupId>
      <artifactId>org.slf4j</artifactId>
    </exclusion>
  </exclusions>
</dependency>

<dependency>
  <groupId>org.slf4j</groupId><!--导入slf4j-log4j框架-->
  <artifactId>slf4j-log4j12</artifactId>
</dependency>

```

切换为log4j2日志框架

```
<dependencies>
```

```

    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
      <exclusions><!--使用log4j2,就需要排除掉以下logging的jar包-->
        <exclusion>
          <artifactId>spring-boot-starter-logging</artifactId>
          <groupId>org.springframework.boot</groupId>
        </exclusion>
      </exclusions>
    </dependency>
  </dependency>

  <dependency><!--添加log4j2的jar包-->
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-log4j2</artifactId>
  </dependency>

```

web开发

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;

springboot对静态页面的映射规则

```

public void addResourceHandlers(ResourceHandlerRegistry registry) {
    if (!this.resourceProperties.isAddMappings()) {
        logger.debug("Default resource handling disabled");
    } else {
        Duration cachePeriod =
this.resourceProperties.getCache().getPeriod();

```

```

        CacheControl cacheControl =
this.resourceProperties.getCache().getCacheControl().toHttpCacheControl();
        if (!registry.hasMappingForPattern("/webjars/**")) {

            this.customizeResourceHandlerRegistration(registry.addHandler(new
String[] {"/webjars/**"}).addResourceLocations(new String[] {"classpath:/META-
INF/resources/webjars/"}).setCachePeriod(this.getSeconds(cachePeriod)).setCacheC
ontrol(cacheControl));
        }

        String staticPathPattern =
this.mvcProperties.getStaticPathPattern();
        if (!registry.hasMappingForPattern(staticPathPattern)) {

            this.customizeResourceHandlerRegistration(registry.addHandler(new
String[]
{staticPathPattern}).addResourceLocations(WebMvcAutoConfiguration.getResourceLoc
ations(this.resourceProperties.getStaticLocations())).setCachePeriod(this.getSec
onds(cachePeriod)).setCacheControl(cacheControl));
        }

    }
}

//配置欢迎页面映射
@Bean
    public welcomePageHandlerMapping
welcomePageHandlerMapping(ApplicationContext applicationContext,
FormattingConversionService mvcConversionService, ResourceUrlProvider
mvcResourceUrlProvider) {
        welcomePageHandlerMapping welcomePageHandlerMapping = new
welcomePageHandlerMapping(new TemplateAvailabilityProviders(applicationContext),
applicationContext, this.getWelcomePage(),
this.mvcProperties.getStaticPathPattern());

        welcomePageHandlerMapping.setInterceptors(this.getInterceptors(mvcConversionSer
vice, mvcResourceUrlProvider));

        welcomePageHandlerMapping.setCorsConfigurations(this.getCorsConfigurations());
        return welcomePageHandlerMapping;
    }
}

```

1、所有/webjars/; 都在classpath:/META-INF/resources/webjars/此处找**

[webjars](#) :以jar包的方式引入静态资源;

 image-20200817111005617

```

<dependency><!--导入jquery的jar包-----在访问的时候只需要写webjars下面资源的名称即可-->
    <groupId>org.webjars</groupId>
    <artifactId>jquery</artifactId>
    <version>1.12.4</version>
</dependency>

```

2、"/"访问当前项目的任何资源（静态资源的文件夹）**

在: org\springframework\boot\spring-boot-autoconfigure\2.3.3.RELEASE\spring-boot-autoconfigure-2.3.3.RELEASE.jar!org.springframework.boot.autoconfigure.web;包下ResourceProperties类

```
private static final String[] CLASSPATH_RESOURCE_LOCATIONS = new String[]
{"classpath:/META-INF/resources/", "classpath:/resources/", "classpath:/static/",
"classpath:/public/"};

classpath:/META-INF/resources/,
classpath:/resources/,
classpath:/static/,
classpath:/public/
"/": 当前项目的根路径
```

localhost:8080/abc===去静态资源文件夹里面找abc

3、欢迎页，静态页面文件夹下的所有index.html页面；被"/"映射**

localhost:8080/ 找index页面

4、所有的/favicon.ico都是在静态资源文件下找;**

模板引擎

jsp、Velocity、Freemarker、Thymeleaf

SpringBoot推荐的Thymeleaf： 语法简单，功能强大

引入Thymeleaf

```
<dependency><!--引入thymeleaf依赖(2.3.3版本的springboot下载的: 3.0.11版本)-->
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-thymeleaf</artifactId>
</dependency>

<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <project.reporting.outputEncoding>UTF-
8</project.reporting.outputEncoding>
  <thymeleaf.version>3.0.11.RELEASE</thymeleaf.version><!--切换thymeleaf版本-->
  <!--布局功能的支持程序 thymeleaf3主程序 Layout2以上版本-->
  <thymeleaf-layout-dialect.version>2.5.1</thymeleaf-layout-
dialect.version>
  <java.version>1.8</java.version>
</properties>
```

```
@ConfigurationProperties(
    prefix = "spring.thymeleaf"
)
public class ThymeleafProperties {
    private static final Charset DEFAULT_ENCODING;
    public static final String DEFAULT_PREFIX = "classpath:/templates/";
    public static final String DEFAULT_SUFFIX = ".html";
    private boolean checkTemplate = true;
    private boolean checkTemplateLocation = true;
    //只要把html页面，放在/templates/下；thymeleaf就能自动渲染
    private String prefix = "classpath:/templates/";
    private String suffix = ".html";
    private String mode = "HTML";
    private Charset encoding;
    private boolean cache;
    private Integer templateResolverOrder;
    private String[] viewNames;
    private String[] excludedViewNames;
    private boolean enableSpringElCompiler;
    private boolean renderHiddenMarkersBeforeCheckboxes;
    private boolean enabled;
    private final ThymeleafProperties.Servlet servlet;
    private final ThymeleafProperties.Reactive reactive;
```

只要我们把HTML页面放在classpath:/templates/，thymeleaf就能自动渲染；

使用：

1、导入thymeleaf的名称空间

```
<html lang="en" xmlns:th="http://www.thymeleaf.org"><!--thymeleaf的语法提示-->
```

2、使用thymeleaf语法：

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org"><!--thymeleaf的语法提示-->
<head>
    <meta charset="UTF-8">
    <title>success页面</title>
</head>
<body>
    <h2>成功</h2>

    <!--将div里面的文本内容设置为指定的值-->
    <div th:text="${hello}">这是显示欢迎信息</div>
</body>
</html>
```

3、语法规则

3、1: th:text ;改变当前元素里面的文本内容;

th: 任意html属性; 来替换原生属性的值



image-20200817130650187

4、表达式:

Simple expressions: (表达式语法)

Variable Expressions: `${...}`: 获取变量值; 底层是OGNL

1、获取对象的属性, 调用方法

2、使用内置的基本对象

#ctx : the context object.

#vars: the context variables.

#locale : the context locale.

#request : (only in web contexts) the HttpServletRequest object.

#response : (only in web contexts) the HttpServletResponse object.

#session : (only in web contexts) the HttpSession object.

#servletContext : (only in web contexts) the ServletContext object.

`${session.foo}` // Retrieves the session attribute 'foo'

`${session.size()}`

`${session.isEmpty()}`

`${session.containsKey('foo')}`

...

3、内置的一些工具对象

#execInfo : information about the template being processed.

#messages : methods for obtaining externalized messages inside variables expressions, in the same way as they would be obtained using `#{...}` syntax.

#uris : methods for escaping parts of URLs/URIs

Page 20 of 106

#conversions : methods for executing the configured conversion service (if any).

#dates : methods for java.util.Date objects: formatting, component extraction, etc.

#calendars : analogous to #dates , but for java.util.Calendar objects.

#numbers : methods for formatting numeric objects.

#strings : methods for String objects: contains, startsWith, prepending/appending, etc.

#objects : methods for objects in general.

#booleans : methods for boolean evaluation.

#arrays : methods for arrays.

#lists : methods for lists.

#sets : methods for sets.

#maps : methods for maps.

#aggregates : methods for creating aggregates on arrays or collections.

#ids : methods for dealing with id attributes that might be repeated (for example, as a result of an iteration).

Selection Variable Expressions: `*{...}`: 选择表达式 (配合 `th:object=${session.user}`)

Message Expressions: `#{...}`: 获取国际化内容

Link URL Expressions: `@{...}`: 定义URL

`@{/order/process(execId=${execId},execType='FAST')}`

Fragment Expressions: `~{...}`: 片段引用类表达式


```

<div th:insert=~{commons :: main}>...</div>
Literals (字面量)
  Text literals: 'one text' , 'Another one!' , ...
  Number literals: 0 , 34 , 3.0 , 12.3 , ...
  Boolean literals: true , false
  Null literal: null
  Literal tokens: one , sometext , main , ...
Text operations: (文本操作)
  String concatenation: +
  Literal substitutions: |The name is ${name}|
Arithmetic operations: (数学运算)
  Binary operators: + , - , * , / , %
  Minus sign (unary operator): -
Boolean operations: (布尔运算)
  Binary operators: and , or
  Boolean negation (unary operator): ! , not
Comparisons and equality: (比较运算)
  Comparators: > , < , >= , <= ( gt , lt , ge , le )
  Equality operators: == , != ( eq , ne )
Conditional operators: (条件运算)
  If-then: (if) ? (then)
  If-then-else: (if) ? (then) : (else)
  Default: (value) ?: (defaultvalue)
Special tokens: (特殊操作)
  Page 17 of 106
  No-Operation: _

```

 image-20200817132002621

4、SpringMVC自动配置

Spring MVC auto-configuration

Spring Boot 自动配置好了SpringMVC

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`;
 -

```

@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`: Spring MVC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;

If you want to keep Spring Boot MVC features, and you just want to add additional [MVC configuration](#) (interceptors, formatters, view controllers etc.) you can add your own `@Configuration` class of type `WebMvcConfigurerAdapter`, but **without** `@EnableWebMvc`. If you wish to provide custom instances of `RequestMappingHandlerMapping`, `RequestMappingHandlerAdapter` or `ExceptionHandlerResolver` you can declare a `WebMvcRegistrationsAdapter` instance providing such components.

If you want to take complete control of Spring MVC, you can add your own `@Configuration` annotated with `@EnableWebMvc`.

扩展SpringMVC

```

<mvc:view-controller path="/hello" view-name="success"/>
<mvc:interceptors>
    <mvc:interceptor>
        <mvc:mapping path="/hello"/>
        <bean></bean>
    </mvc:interceptor>
</mvc:interceptors>

```

编写一个配置类 (`@Configuration`) , 是`WebMvcConfigurerAdapter`类型; 不能标注 `@EnableWebMvc`==**;

既保留了所有的自动配置，也能用我们扩展的配置；

```
@Configuration
public class MyMvcConfig implements WebMvcConfigurer {
    //SpringBoot2.0+版本继承WebMvcConfigurerAdapter已经过时，而是通过实现
    WebMvcConfigurer接口来拓展配置

    @Override
    public void addViewControllers(ViewControllerRegistry registry) {
        //访问atguigu请求，会跳到模板中的success页面
        registry.addViewController("/atguigu").setViewName("success");
    }
}
```

原理：

- 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)

```
@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；

效果：SpringMVC的自动配置和我们的扩展配置都会起作用；

3、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；

```
//使用WebMvcConfigurerAdapter可以来扩展SpringMVC的功能
@EnableWebMvc
@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");
    }
}
```

原理:

为什么@EnableWebMvc自动配置就失效了;

1) @EnableWebMvc的核心

```
@Import(DelegatingWebMvcConfiguration.class)
public @interface EnableWebMvc {
```

2) 、

```
@Configuration(
    proxyBeanMethods = false
)
public class DelegatingWebMvcConfiguration extends
WebMvcConfigurationSupport {
```

3) 、

```
@Configuration(
    proxyBeanMethods = false
)
@ConditionalOnWebApplication(
    type = Type.SERVLET
)
@ConditionalOnClass({Servlet.class, DispatcherServlet.class,
WebMvcConfigurer.class})
//容器中没有这个组件的时候，这个自动配置类才生效
@ConditionalOnMissingBean({WebMvcConfigurationSupport.class})
@AutoConfigureOrder(-2147483638)
@AutoConfigureAfter({DispatcherServletAutoConfiguration.class,
TaskExecutionAutoConfiguration.class, ValidationAutoConfiguration.class})
public class WebMvcAutoConfiguration {
```

4) 、@EnableWebMvc将WebMvcConfigurationSupport组件导入进来;

5) 、导入的WebMvcConfigurationSupport只是SpringMVC最基本的功能;

5、如何修改SpringBoot的默认配置

模式：

- 1)、SpringBoot在自动配置很多组件的时候，先看容器中有没有用户自己配置的（@Bean、@Component）如果有就用用户配置的，如果没有，才自动配置；如果有些组件可以有多个（ViewResolver）将用户配置的和自己默认的组合起来；
- 2)、在SpringBoot中会有非常多的xxxConfigurer帮助我们进行扩展配置
- 3)、在SpringBoot中会有很多的xxxCustomizer帮助我们进行定制配置

RestfulCRUD

默认访问首页

```
@Configuration
public class MyMvcConfig implements WebMvcConfigurer {
    //WebMvcConfigurerAdapter失效可以直接用WebMvcConfigurer，WebMvcConfigurer已经重写了

    @Override
    public void addViewControllers(ViewControllerRegistry registry) {
        //访问atguigu请求，会跳到模板中的success页面
        registry.addViewController("/atguigu").setViewName("success");
    }

    //所有的WebMvcConfigurer组件都会一起起作用
    @Bean //将组件注册在spring容器
    public WebMvcConfigurer webMvcConfigurer() {
        WebMvcConfigurer webMvcConfigurer = new WebMvcConfigurer() {
            @Override
            public void addViewControllers(ViewControllerRegistry registry) {
                registry.addViewController("/").setViewName("login");
                registry.addViewController("/index.html").setViewName("login");
            }
        };
        return webMvcConfigurer;
    }
}
```

国际化

- 1、编写国际化配置文件；
- 2、使用ResourceBundleMessageSource管理国际化资源文件
- 3、在页面使用fmt:message取出国际化内容

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

 image-20201119183215755

2、SpringBoot自动配置好了管理国际化资源文件的组件;

```
public class MessageSourceProperties {
    private String basename = "messages";

    @Configuration(
        proxyBeanMethods = false
    )
    @ConditionalOnMissingBean(
        name = {"messageSource"},
        search = SearchStrategy.CURRENT
    )
    @AutoConfigureOrder(-2147483648)
    @Conditional({MessageSourceAutoConfiguration.ResourceBundleCondition.class})
    @EnableConfigurationProperties
    public class MessageSourceAutoConfiguration {

        @Bean
        public MessageSource messageSource(MessageSourceProperties properties) {
            ResourceBundleMessageSource messageSource = new
            ResourceBundleMessageSource();
            if (StringUtils.hasText(properties.getBasename())) {
                messageSource.setBasenames(StringUtils.commaDelimitedListToStringArray(StringUtils.trimAllWhitespace(properties.getBasename())));
            }

            if (properties.getEncoding() != null) {
                messageSource.setDefaultEncoding(properties.getEncoding().name());
            }

            messageSource.setFallbackToSystemLocale(properties.isFallbackToSystemLocale());
            Duration cacheDuration = properties.getCacheDuration();
            if (cacheDuration != null) {
                messageSource.setCacheMillis(cacheDuration.toMillis());
            }

            messageSource.setAlwaysUseMessageFormat(properties.isAlwaysUseMessageFormat());

            messageSource.setUseCodeAsDefaultMessage(properties.isUseCodeAsDefaultMessage());
            return messageSource;
        }
    }
}
```

3、去页面获取国际化的值;

```
# 绑定国际化资源文件的基础名
spring.messages.basename=i18n.login
```

```
//标签体内：需要相互转换的地方：#{login.tip}...
//标签体中：需要相互转换的地方：[[#{login.remember}]]...
```

效果：根据浏览器语言设置的信息切换了国际化；

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1,
shrink-to-fit=no">
    <meta name="description" content="">
    <meta name="author" content="">
    <title>Signin Template for Bootstrap</title>
    <!-- Bootstrap core CSS -->
    <link href="asserts/css/bootstrap.min.css"
th:href="@{/webjars/bootstrap/4.5.3/css/bootstrap.css}" rel="stylesheet">
    <!-- Custom styles for this template -->
    <link href="asserts/css/signin.css" th:href="@{/asserts/css/signin.css}"
rel="stylesheet">
  </head>
  <body class="text-center">
    <form class="form-signin" action="dashboard.html"
th:action="@{/user/login}" method="post">
      
      <h1 class="h3 mb-3 font-weight-normal" th:text="#{login.tip}">Please
sign in</h1>
      <!--判断-->
      <p style="color: red" th:text="${msg}" th:if="${not
#strings.isEmpty(msg)}"></p>
      <label class="sr-only" th:text="#{login.username}">Username</label>
      <input type="text" name="username" class="form-control"
placeholder="Username" th:placeholder="#{login.username}" required=""
autofocus="">
      <label class="sr-only" th:text="#{login.password}">Password</label>
      <input type="password" name="password" class="form-control"
placeholder="Password" th:placeholder="#{login.password}" required="">
      <div class="checkbox mb-3">
        <label>
          <input type="checkbox" value="remember-me"/> [[#{
login.remember}]]
        </label>
      </div>
      <button class="btn btn-lg btn-primary btn-block" type="submit"
th:text="#{login.btn}">Sign in</button>
      <p class="mt-5 mb-3 text-muted">© 2017-2018</p>
      <a class="btn btn-sm" th:href="@{/index.html(l='zh_CN')}">中文</a>
      <a class="btn btn-sm"
th:href="@{/index.html(l='en_US')}">English</a>
    </form>
  </body>
</html>
```

原理：

国际化Locale（区域信息对象）；LocaleResolver（获取区域信息对象）；

```
@Bean
@ConditionalOnMissingBean
@ConditionalOnProperty(
    prefix = "spring.mvc",
    name = {"locale"}
)
public LocaleResolver localeResolver() {
    if (this.mvcProperties.getLocaleResolver() ==
org.springframework.boot.autoconfigure.web.servlet.WebMvcProperties.LocaleResolv
er.FIXED) {
        return new FixedLocaleResolver(this.mvcProperties.getLocale());
    } else {
        AcceptHeaderLocaleResolver localeResolver = new
AcceptHeaderLocaleResolver();
        localeResolver.setDefaultLocale(this.mvcProperties.getLocale());
        return localeResolver;
    }
}

//默认的就是根据请求头带来的区域信息获取Locale进行国际化
```

4、点击链接切换国际化

```
public class MyLocalResolver implements LocaleResolver {

    @Override
    public Locale resolveLocale(HttpServletRequest httpServletRequest) {
        Locale locale = Locale.getDefault();//默认使用操作系统的默认语言
        String l = httpServletRequest.getParameter("l");
        if (!Strings.isEmpty(l)) {
            String[] split = l.split("_");
            locale = new Locale(split[0], split[1]);
        }

        return locale;
    }

    @Override
    public void setLocale(HttpServletRequest httpServletRequest,
        HttpServletResponse httpServletResponse, Locale locale) {

    }
}

@Configuration
public class MyMvcConfig implements WebMvcConfigurer {
    //WebMvcConfigurerAdapter失效可以直接用WebMvcConfigurer，WebMvcConfigurer已经重写了

    @Override
```



```

public void addViewControllers(ViewControllerRegistry registry) {
    //访问atguigu请求，会跳到模板中的success页面
    registry.addViewController("/atguigu").setViewName("success");
}

//所有的webMvcConfigurer组件都会一起起作用
@Bean    //将组件注册在spring容器
public WebMvcConfigurer webMvcConfigurer() {
    WebMvcConfigurer webMvcConfigurer = new WebMvcConfigurer() {
        @Override
        public void addViewControllers(ViewControllerRegistry registry) {
            registry.addViewController("/").setViewName("login");
            registry.addViewController("/index.html").setViewName("login");
        }
    };
    return webMvcConfigurer;
}

@Bean
public LocaleResolver localeResolver() {
    return new MyLocaleResolver();
}
}

```

登录

开发期间模板引擎页面修改以后，要实时生效

1)、禁用模板引擎的缓存

```

# 禁用缓存
spring.thymeleaf.cache=false

```

2)、页面修改完成以后ctrl+f9：重新编译；

登陆错误消息的显示

```

<!--判断：msg为空就显示错误信息-->
<p style="color: red" th:text="${msg}" th:if="${not
#strings.isEmpty(msg)}"></p>

```

拦截器进行登陆检查

拦截器

```

public class loginHandlerInterceptor implements HandlerInterceptor {
    //目标方法执行前
    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse
response, Object handler) throws Exception {
        Object user = request.getSession().getAttribute("loginUser");
    }
}

```

```

        if (null == user) {
            //未登录，返回登录页面
            //转发到index页面
            request.setAttribute("msg", "没有权限，请先登录");
            request.getRequestDispatcher("/index.html").forward(request,
response);
            return false;
        } else {
            //已登录
            return true;
        }
    }

    @Override
    public void postHandle(HttpServletRequest request, HttpServletResponse
response, Object handler, ModelAndView modelAndView) throws Exception {

    }

    @Override
    public void afterCompletion(HttpServletRequest request, HttpServletResponse
response, Object handler, Exception ex) throws Exception {

    }
}

```

注册拦截器

```

@Configuration
public class MyMvcConfig implements WebMvcConfigurer {
    //WebMvcConfigurerAdapter失效可以直接用WebMvcConfigurer，WebMvcConfigurer已经重写
    了

    @Override
    public void addViewControllers(ViewControllerRegistry registry) {
        //访问atguigu请求，会跳到模板中的success页面
        registry.addViewController("/atguigu").setViewName("success");
    }

    @Override
    public void addInterceptors(InterceptorRegistry registry) { //添加拦截器
        //拦截所有请求；排除指定请求：首页，登录页
        //SpringBoot 2.x前已经做好了静态资源映射；2.0后又需要释放静态资源了
        registry.addInterceptor(new
loginHandlerInterceptor()).addPathPatterns("/**")
            .excludePathPatterns("/index.html", "/", "/user/login",
"/assets/**", "/webjars/**");
    }

    //所有的WebMvcConfigurer组件都会一起起作用
    @Bean    //将组件注册在spring容器
    public WebMvcConfigurer webMvcConfigurer() {
        WebMvcConfigurer webMvcConfigurer = new WebMvcConfigurer() {
            @Override
            public void addViewControllers(ViewControllerRegistry registry) {

```

```

        registry.addViewController("/").setViewName("login");
        registry.addViewController("/index.html").setViewName("login");

        registry.addViewController("/main.html").setViewName("dashboard");
    }
};

    return webMvcConfigurer;
}

@Bean
public LocaleResolver localeResolver() {
    return new MyLocalResolver();
}
}

```

CRUD-员工列表

实验要求：

1、RestfulCRUD：CRUD满足Rest风格；

URI： /资源名称/资源标识 HTTP请求方式区分对资源CRUD操作

	普通CRUD (uri来区分操作)	RestfulCRUD
查询	getEmp	emp---GET
添加	addEmp?xxx	emp---POST
修改	updateEmp?id=xxx&xxx=xx	emp/{id}---PUT
删除	deleteEmp?id=1	emp/{id}---DELETE

2、实验的请求架构;

实验功能	请求URI	请求方式
查询所有员工	emps	GET
查询某个员工(来到修改页面)	emp/1	GET
来到添加页面	emp	GET
添加员工	emp	POST
来到修改页面（查出员工进行信息回显）	emp/1	GET
修改员工	emp	PUT
删除员工	emp/1	DELETE

3、员工列表：

thymeleaf公共页面元素抽取

1、抽取公共片段

```
<div th:fragment="copy">
&copy; 2011 The Good Thymes Virtual Grocery
</div>
```

2、引入公共片段

```
<div th:insert=~{footer :: copy}"></div>
~{templatename::selector}: 模板名::选择器
~{templatename::fragmentname}:模板名::片段名(fragment的名称)
```

3、默认效果:

insert的公共片段在div标签中

如果使用th:insert等属性进行引入，可以不用写~{}:

行内写法可以加上: [[~{}]] ; [(~{})];

三种引入公共片段的th属性:

th:insert: 将公共片段整个插入到声明引入的元素中

th:replace: 将声明引入的元素替换为公共片段

th:include: 将被引入的片段的内容包含进这个标签中

```
<footer th:fragment="copy">
&copy; 2011 The Good Thymes Virtual Grocery
</footer>
```

引入方式

```
<div th:insert="footer :: copy"></div>
<div th:replace="footer :: copy"></div>
<div th:include="footer :: copy"></div>
```

效果

```
<div>
  <footer>
    &copy; 2011 The Good Thymes Virtual Grocery
  </footer>
</div>
```

```
<footer>
&copy; 2011 The Good Thymes Virtual Grocery
</footer>
```

```
<div>
&copy; 2011 The Good Thymes Virtual Grocery
</div>
```

引入片段的时候传入参数

CRUD-员工添加

```
/**
 * 查询出所有部门
 * @param model
 * @return
 */
@GetMapping("/addAmp")
public String toAddPage(Model model) {
    //查询出所有部门，显示在页面的下拉框
    Collection<Department> departments = departmentDao.getDepartments();
    model.addAttribute("depts", departments);

    return "emp/add"; //跳转到添加页面
}

/**
 * 添加用户
 * springmvc自动将请求参数和入参对象的属性进行一一绑定;要求了请求参数的名字和javaBean入参
的对象里面的属性名是一样的
 * @param employee
 * @return
 */
@PostMapping("/addAmp")
public String addEmp(Employee employee) {
    //拿到传递的对象数据
    System.out.println("employee:\t" + employee);
    employeeDao.save(employee); //保存信息

    //redirect: 重定向到某个地址(代表当前项目路径)
    //forward: 转发到某个地址
    return "redirect:/emp/emp";
}

@Repository
public class EmployeeDao {
    public void save(Employee employee){
        if(employee.getId() == null){
            employee.setId(initId++);
        }

        employee.setDepartment(departmentDao.getDepartment(employee.getDepartment().getId()));
        employees.put(employee.getId(), employee);
    }
}
```

```
<!DOCTYPE html>
<!-- saved from url=(0052)http://getbootstrap.com/docs/4.0/examples/dashboard/ -->
<html lang="en" xmlns:th="http://www.thymeleaf.org">
```

```

<head>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1,
shrink-to-fit=no">
  <meta name="description" content="">
  <meta name="author" content="">

  <title>Dashboard Template for Bootstrap</title>
  <!-- Bootstrap core CSS -->
  <link href="assets/css/bootstrap.min.css"
th:href="@{/webjars/bootstrap/4.5.3/css/bootstrap.css}" rel="stylesheet">
  <link href="https://cdn.bootcss.com/bootstrap-
datetimepicker/4.17.47/css/bootstrap-datetimepicker.min.css" rel="stylesheet">

  <!-- Custom styles for this template -->
  <link href="assets/css/dashboard.css"
th:href="@{/assets/css/dashboard.css}" rel="stylesheet">
  <style type="text/css">
    /* Chart.js */

    @-webkit-keyframes chartjs-render-animation {
      from {
        opacity: 0.99
      }
      to {
        opacity: 1
      }
    }

    @keyframes chartjs-render-animation {
      from {
        opacity: 0.99
      }
      to {
        opacity: 1
      }
    }

    .chartjs-render-monitor {
      -webkit-animation: chartjs-render-animation 0.001s;
      animation: chartjs-render-animation 0.001s;
    }
  </style>
</head>

<body>
  <!--引入抽取的topbar-->
  <!--模板名：会使用thymeleaf的前后缀配置规则进行解析-->
  <div th:replace="commons/bar::topbar"></div>

  <div class="container-fluid">
    <div class="row">
      <!--引入侧边栏-->
      <div th:replace="commons/bar::#sidebar(activeUri='emps')"></div>

      <main role="main" class="col-md-9 ml-sm-auto col-lg-10 pt-3 px-
4">

```

置好的)

方式

```
<!--需要区分是员工修改还是添加; -->
<form th:action="@{/emp/emp}" method="post">
    <!--发送put请求修改员工数据-->
    <!--
    1、SpringMVC中配置HiddenHttpMethodFilter; (SpringBoot自动配
    2、页面创建一个post表单
    3、创建一个input项, name="_method";值(value)就是我们指定的请求
    方式
    -->
    <input type="hidden" name="_method" value="put"
th:if="${emp!=null}"/>
    <!-- emp不为空, 也就是修改的时候, 才有隐藏的input的id标签 -->
    <input type="hidden" name="id" th:if="${emp!=null}"
th:value="${emp.id}">
    <div class="form-group">
        <label>LastName</label>
        <input name="lastName" type="text" class="form-
control" placeholder="zhangsan" th:value="${emp!=null}?${emp.lastName}">
    </div>
    <div class="form-group">
        <label>Email</label>
        <input name="email" type="email" class="form-
control" placeholder="zhangsan@atguigu.com" th:value="${emp!=null}?
${emp.email}">
    </div>
    <div class="form-group">
        <label>Gender</label><br/>
        <div class="form-check form-check-inline">
            <input class="form-check-input" type="radio"
name="gender" value="1" th:checked="${emp!=null}?${emp.gender==1}">
            <label class="form-check-label">男</label>
        </div>
        <div class="form-check form-check-inline">
            <input class="form-check-input" type="radio"
name="gender" value="0" th:checked="${emp!=null}?${emp.gender==0}">
            <label class="form-check-label">女</label>
        </div>
    </div>
    <div class="form-group">
        <label>department</label>
        <!--提交的是部门的id-->
        <select class="form-control" name="department.id">
            <option th:selected="${emp != null} ? ${dept.id
== emp.department.id}" th:value="${dept.id}" th:each="dept:${depts}"
th:text="${dept.departmentName}">1</option>
        </select>
    </div>
    <div class="form-group">
        <label>Birth</label><!--
th:value="${#dates.format(emp.birth, 'yyyy-MM-
dd')}"+'T'+${#dates.format(emp.birth, 'HH:mm:ss')}" -->
        <input type='datetime-local' name="birth"
th:value="${emp != null} ? (${#dates.format(emp.birth, 'yyyy-MM-
dd')}"+'T'+${#dates.format(emp.birth, 'HH:mm:ss')})" />
    </div>
    <button type="submit" class="btn btn-primary"
th:text="${emp!=null}?'修改':'添加'">添加</button>
```

```

        </form>
    </main>
</div>
</div>

<!-- Bootstrap core JavaScript
===== -->
<!-- Placed at the end of the document so the pages load faster -->
<script type="text/javascript" src="asserts/js/jquery-3.2.1.slim.min.js"
th:src="@{/webjars/jquery/1.12.4/jquery.js}"></script>
<script type="text/javascript" src="asserts/js/popper.min.js"
th:src="@{/webjars/popper.js/1.11.1/dist/popper.js}"></script>
<script type="text/javascript" src="asserts/js/bootstrap.min.js"
th:src="@{/webjars/bootstrap/4.5.3/js/bootstrap.js}"></script>

<!-- Icons -->
<script type="text/javascript" src="asserts/js/feather.min.js"
th:src="@{/asserts/js/feather.min.js}"></script>
<script>
    feather.replace();

</script>

</body>

</html>

```

提交的数据格式不对：生日：日期；

2017-12-12； 2017/12/12； 2017.12.12；

日期的格式化；SpringMVC将页面提交的值需要转换为指定的类型；

2017-12-12---Date； 类型转换，格式化；

默认日期是按照/的方式；

```

# 格式化mvc的日期格式
spring.mvc.format.date=yyyy-MM-dd

```

设置后就只能使用这种方式了

CRUD-员工修改

修改添加二合一表单

```

/**
 * 跳转至修改页面，查出当前员工，回显到表单中
 * @param id
 * @return
 */
@GetMapping("/editAmp/{id}")
public String toEditPage(@PathVariable("id") Integer id, Model model) {
    //查询出所有员工和部门，显示在页面的下拉框
    Employee employee = employeeDao.get(id);
    model.addAttribute("emp", employee);
}

```



```

        Collection<Department> departments = departmentDao.getDepartments();
        model.addAttribute("depts", departments);

        return "emp/add"; //跳转到修改页面（add和修改通用）
    }

    /**
     * 修改员工信息
     * @param employee
     * @return
     */
    @PutMapping("/emp")
    public String updateEmployee(Employee employee) {
        System.out.println("修改employee:\t" + employee);
        employeeDao.save(employee);

        return "redirect:/emp/emps";
    }
}

```

CRUD-员工删除

```

    /**
     * 删除指定员工
     * @param id
     * @return
     */
    @DeleteMapping("/emp/{id}")
    public String deleteEmployee(@PathVariable("id") Integer id) {
        System.out.println("删除id:\t" + id);
        employeeDao.delete(id);
        return "redirect:/emp/emps";
    }
}

```

错误处理机制

SpringBoot默认的错误处理机制

默认效果：

- 1、浏览器，返回一个默认的错误页面

浏览器发送请求的请求头：

 image-20201125135216693

浏览器的请求头

 image-20201125141622394

- 2、如果是其他客户端，默认响应一个json数据

其他测试工具的请求头

 image-20201125141736664

```
{"timestamp": "2020-11-25T05:51:27.559+00:00", "status": 404, "error": "Not Found", "message": "", "path": "/crud/a"}
```

原理:

```
package org.springframework.boot.autoconfigure.web.servlet.error;
```

可以参照ErrorMvcAutoConfiguration; 错误处理的自动配置;

给容器中添加了以下组件

1、DefaultErrorAttributes:

```
@Bean
@ConditionalOnMissingBean(
    value = {ErrorAttributes.class},
    search = SearchStrategy.CURRENT
)
public DefaultErrorAttributes errorAttributes() {
    return new DefaultErrorAttributes();
}
```

2、BasicErrorController: 处理默认/error请求

```
@Bean
@ConditionalOnMissingBean(
    value = {ErrorController.class},
    search = SearchStrategy.CURRENT
)
public BasicErrorController basicErrorController(ErrorAttributes
errorAttributes, ObjectProvider<ErrorViewResolver> errorViewResolvers) {
    return new BasicErrorController(errorAttributes,
this.serverProperties.getError(),
(List)errorViewResolvers.orderedStream().collect(Collectors.toList()));
}

/*
最终返回
@Controller
    取出端口后的路径，如果取不出就是要error.path，error.path也没有就用error
@RequestMapping("${server.error.path:${error.path:/error}}")
public class BasicErrorController extends AbstractErrorController {
    */

//有以下两种处理方式
    @RequestMapping(
        produces = {"text/html"} //产生html类型数据
    )
    public ModelAndView errorHtml(HttpServletRequest request,
        HttpServletResponse response) {
        HttpStatus status = this.getStatus(request);
```

```

        Map<String, Object> model =
Collections.unmodifiableMap(this.getErrorAttributes(request,
this.getErrorAttributeOptions(request, MediaType.TEXT_HTML)));
        response.setStatus(status.value());
        //错误页面, 包含页面地址和页面内容
        ModelAndView modelAndView = this.resolveErrorView(request, response,
status, model);
        return modelAndView != null ? modelAndView : new ModelAndView("error",
model);
    }

    @RequestMapping//产生json数据
    public ResponseEntity<Map<String, Object>> error(HttpServletRequest request)
    {
        HttpStatus status = this.getStatus(request);
        if (status == HttpStatus.NO_CONTENT) {
            return new ResponseEntity(status);
        } else {
            Map<String, Object> body = this.getErrorAttributes(request,
this.getErrorAttributeOptions(request, MediaType.ALL));
            return new ResponseEntity(body, status);
        }
    }
}

```

3、ErrorPageCustomizer:

```

    static class ErrorPageCustomizer implements ErrorPageRegistrar, Ordered {
        private final ServerProperties properties;
        private final DispatcherServletPath dispatcherServletPath;

        protected ErrorPageCustomizer(ServerProperties properties,
DispatcherServletPath dispatcherServletPath) {
            this.properties = properties;
            this.dispatcherServletPath = dispatcherServletPath;
        }

        public void registerErrorPages(ErrorPageRegistry errorPageRegistry) {
            ErrorPage errorPage = new
ErrorPage(this.dispatcherServletPath.getRelativePath(this.properties.getError().
getPath()));//根据getPath方法得到ErrorProperties中的path:
            /*
            @value("${error.path:/error}")
            private String path = "/error";
            */
            errorPageRegistry.addErrorPages(new ErrorPage[]{errorPage});
        }

        public int getOrder() {
            return 0;
        }
    }
}

```

4、DefaultErrorViewResolver:

```

@Configuration(
    proxyBeanMethods = false
)
static class DefaultErrorViewResolverConfiguration {
    private final ApplicationContext applicationContext;
    private final ResourceProperties resourceProperties;

    DefaultErrorViewResolverConfiguration(ApplicationContext
applicationContext, ResourceProperties resourceProperties) {
        this.applicationContext = applicationContext;
        this.resourceProperties = resourceProperties;
    }

    @Bean
    @ConditionalOnBean({DispatcherServlet.class})
    @ConditionalOnMissingBean({ErrorViewResolver.class})
    DefaultErrorViewResolver conventionErrorViewResolver() {
        return new DefaultErrorViewResolver(this.applicationContext,
this.resourceProperties);
    }
}

```

步骤：

一旦系统出现4xx或者5xx之类的错误；ErrorPageCustomizer就会生效（定制错误的响应规则）；就会来到/error请求；就会被**BasicErrorController**处理；

1、响应页面；去哪个页面是由**DefaultErrorViewResolver**解析得到的；

```

//根据BasicErrorController下的errorHtml方法resolveErrorView得到
protected ModelAndView resolveErrorView(HttpServletRequest request,
HttpServletResponse response, HttpStatus status, Map<String, Object> model) {
    Iterator var5 = this.errorViewResolvers.iterator();

    ModelAndView modelAndView;
    do {
        if (!var5.hasNext()) {
            return null;
        }

        ErrorViewResolver resolver = (ErrorViewResolver)var5.next();
        modelAndView = resolver.resolveErrorView(request, status, model);
    } while(modelAndView == null);

    return modelAndView;
}

```

```

//根据DefaultErrorViewResolverConfiguration内的DefaultErrorViewResolver得到
//package org.springframework.boot.autoconfigure.web.servlet.error;
public class DefaultErrorViewResolver implements ErrorViewResolver, Ordered {
    static {
        Map<Series, String> views = new EnumMap(Series.class);
        views.put(Series.CLIENT_ERROR, "4xx");
        views.put(Series.SERVER_ERROR, "5xx");
        SERIES_VIEWS = Collections.unmodifiableMap(views);
    }
}

```

```

    public ModelAndView resolveErrorView(HttpServletRequest request, HttpStatus
status, Map<String, Object> model) {
        ModelAndView modelAndView = this.resolve(String.valueOf(status.value()),
model);
        if (modelAndView == null && SERIES_VIEWS.containsKey(status.series())) {
            modelAndView =
this.resolve((String)SERIES_VIEWS.get(status.series()), model);
        }

        return modelAndView;
    }

    private ModelAndView resolve(String viewName, Map<String, Object> model) {
        //springboot可以找到error/404页面
        String errorViewName = "error/" + viewName;

        //模板引擎可以解析这个地址
        TemplateAvailabilityProvider provider =
this.templateAvailabilityProviders.getProvider(errorViewName,
this.applicationContext);
        //模板引擎可用返回到errorViewName指定的视图地址
        return provider != null ? new ModelAndView(errorViewName, model) :
this.resolveResource(errorViewName, model);
        //模板引擎不可用，就在静态资源文件下找errorViewName对应的页面：error/404.html
    }
}

```

如何定制错误响应：

如何定制错误的页面：

1、有模板引擎的情况下；error/状态码；【将错误页面命名为 错误状态码.html 放在模板引擎文件夹里面的 error文件夹下】，发生此状态码的错误就会来到 对应的页面；

我们可以使用4xx和5xx作为错误页面的文件名来匹配这种类型的所有错误，精确优先（优先寻找精确的状态码.html）；

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(static)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；

如何定制错误的json数据;

1、自定义异常处理&返回定制json数据;

```
#手动配置了资源文件夹，那么默认的配置就不再生效了
#spring.resources.static-locations=classpath:/hello/,classpath:/atguigu

# 设置项目虚拟路径
server.servlet.context-path=/crud

# 绑定国际化资源文件的基础名
spring.messages.basename=i18n.login

# 禁用缓存
spring.thymeleaf.cache=false

# 格式化mvc的日期格式
spring.mvc.format.date=yyyy-MM-dd

# boot2以上默认不配置HiddenHttpMethodFilter,需要手动配置
spring.mvc.hiddenmethod.filter.enabled=true
spring.mvc.hiddenmethod.filter.enabled=true

#获取SpringBoot的异常对象exception: 以下两个不配做，那么就不会显示exception和message
server.error.include-exception=true
server.error.include-message=always
```

```
//抛出指定异常
public class UserNotExistException extends RuntimeException {
    public UserNotExistException () {
        super("用户不存在");
    }
}

//异常处理器（浏览器客户端返回的都是json）
@ControllerAdvice
public class MyExceptionHandler {
    @ResponseBody
    @ExceptionHandler(UserNotExistException.class)//处理指定异常
    public Map<String, Object> handleException (Exception e) {
        Map<String, Object> map = new HashMap<>();
        map.put("code", "notexist");//标识码
        map.put("message", e.getMessage());//异常错误信息
        return map;
    }
}
```

2、转发到/error进行自适应响应效果处理

```

public String handleException (Exception e, HttpServletRequest request) {
    Map<String, Object> map = new HashMap<>();
    //传入自定义的错误状态码:4xx、5xx，否则不会进入定制错误页面解析；不写默认是200
    /*
        BasicErrorController中errhtml方法中getStatus的内容

        Integer statusCode = (Integer)request
            .getAttribute("javax.servlet.error.status_code");
        */
    request.setAttribute("javax.servlet.error.status_code", 500);
    map.put("code", "notexist");//标识码
    map.put("message", e.getMessage());//异常错误信息
    //转发到/error
    return "forward:/error";
}

```

3、将我们的定制数据携带出去；

出现错误以后，会来到/error请求，会被BasicErrorController处理，响应出去可以获取的数据是由getErrorAttributes得到的（是AbstractErrorController（ErrorController）规定的方法）；

1、完全来编写一个ErrorController的实现类【或者是编写AbstractErrorController的子类】，放在容器中；

2、页面上能用的数据，或者是json返回能用的数据都是通过errorAttributes.getErrorAttributes得到；

容器中DefaultErrorAttributes.getErrorAttributes(); 默认进行数据处理的；

自定义ErrorAttributes

```

@Component //给容器加入自定义的错误属性ErrorAttrbuytes
public class MyErrorAttrbuytes extends DefaultErrorAttributes {
    @Override
    public Map<String, Object> getErrorAttributes(WebRequest webRequest,
        ErrorAttributeOptions options) {
        Map<String, Object> map = super.getErrorAttributes(webRequest, options);
        map.put("company", "fsir");
        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

 image-20201125185659313

配置嵌入式Servlet容器

SpringBoot默认使用Tomcat作为嵌入式的Servlet容器

 image-20201125204254435

如何定制和修改Servlet容器的相关配置：

1、修改和server有关的配置（ServerProperties【也是EmbeddedServletContainerCustomizer】）；

```
server.port=8081
server.context-path=/crud

server.tomcat.uri-encoding=UTF-8

//通用的Servlet容器设置
server.xxx
//Tomcat的设置
server.tomcat.xxx
```

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

boot2.0后使用的是：WebServerFactoryCustomizer类

```
@Configuration
public class MyMvcConfig implements WebMvcConfigurer {
    //WebMvcConfigurerAdapter失效可以直接用WebMvcConfigurer，WebMvcConfigurer已经重写了

    @Override
    public void addViewControllers(ViewControllerRegistry registry) {
        //访问atguigu请求，会跳到模板中的success页面
        registry.addViewController("/atguigu").setViewName("success");
    }

    /**
     * 添加拦截器
     * @param registry
     */
    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        //拦截所有请求；排除指定请求：首页，登录页
        //SpringBoot 2.x前已经做好了静态资源映射；2.0后又需要释放静态资源了
        registry.addInterceptor(new
        LoginHandlerInterceptor()).addPathPatterns("/**")
            .excludePathPatterns("/index.html", "/", "/user/login",
            "/assets/**", "/webjars/**");
    }

    //所有的WebMvcConfigurer组件都会一起起作用
    @Bean //将组件注册在spring容器
    public WebMvcConfigurer webMvcConfigurer() {
        WebMvcConfigurer webMvcConfigurer = new WebMvcConfigurer() {
            @Override
            public void addViewControllers(ViewControllerRegistry registry) {
                /* 将指定视图名，映射到另一个指定视图名 */
                registry.addViewController("/").setViewName("login");
                registry.addViewController("/index.html").setViewName("login");
                registry.addViewController("/main").setViewName("dashboard");

                registry.addViewController("/main.html").setViewName("dashboard");
            }
        }
    }
}
```



```

};

    return webMvcConfigurer;
}

@Bean
public LocaleResolver localeResolver() {
    return new MyLocalResolver();
}

@Bean    //也可以和三大组件配置一起
public WebServerFactoryCustomizer<ConfigurableServletWebServerFactory>
webServerFactoryCustomizer () {
    return new
WebServerFactoryCustomizer<ConfigurableServletWebServerFactory>() {
        //定制嵌入式的servlet容器相关规则
        @Override
        public void customize(ConfigurableServletWebServerFactory factory) {
            factory.setPort(8083);
        }
    };
}
}
}

```

注册Servlet三大组件【Servlet、Filter、Listener】

由于SpringBoot默认是以jar包的方式启动嵌入式的Servlet容器来启动SpringBoot的web应用，没有web.xml文件。

注册三大组件用以下方式

ServletRegistrationBean

```

@Bean    //注册到spring容器
public ServletRegistrationBean myServlet() {
    ServletRegistrationBean registrationBean = new ServletRegistrationBean<>
(new MyServlet(), "/myServlet");
    return registrationBean;
}

//servlet类
public class MyServlet extends HttpServlet {
    //处理get请求
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
        //super.doGet(req, resp);
        doPost(req, resp);
    }
}

```

```

//处理post请求
@Override
protected void doPost(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
//    super.doPost(req, resp);
    resp.getWriter().write("Hello Servlet");
}
}

```

FilterRegistrationBean

```

/**
 * 注册过滤器
 * @return
 */
@Bean
public FilterRegistrationBean myFilter () {
    FilterRegistrationBean<Filter> registrationBean = new
FilterRegistrationBean<>();
    registrationBean.setFilter(new MyFilter());
    registrationBean.setUrlPatterns(Arrays.asList("/hello", "/myServlet"));
    return registrationBean;
}

//filter类
public class MyFilter implements Filter {
    /**
     * 初始发
     * @param filterConfig
     * @throws ServletException
     */
    @Override
    public void init(FilterConfig filterConfig) throws ServletException {

    }

    /**
     * 过滤
     * @param servletRequest
     * @param servletResponse
     * @param filterChain
     * @throws IOException
     * @throws ServletException
     */
    @Override
    public void doFilter(ServletRequest servletRequest, ServletResponse
servletResponse, FilterChain filterChain) throws IOException, ServletException {
        System.out.println("MyFilter process.....");
        filterChain.doFilter(servletRequest, servletResponse);
    }

    /**
     * 销毁
     */
}

```

```

@Override
public void destroy() {

}
}

```

ServletListenerRegistrationBean

```

/**
 * 注册listener
 * @return
 */
@Bean
public ServletListenerRegistrationBean myListener () {
    ServletListenerRegistrationBean<MyListener> registrationBean = new
ServletListenerRegistrationBean<MyListener>(new MyListener());
    return registrationBean;
}

//listener类
public class MyListener implements ServletContextListener {
    /**
     * servlet初始发
     * @param sce
     */
    @Override
    public void contextInitialized(ServletContextEvent sce) {
        System.out.println("contextInitialized start.....");
    }

    /**
     * 销毁
     * @param sce
     */
    @Override
    public void contextDestroyed(ServletContextEvent sce) {
        System.out.println("contextDestroyed 当前web项目关闭了.....");
    }
}

```

SpringBoot帮我们自动SpringMVC的时候，自动的注册SpringMVC的前端控制器；
DispatcherServlet;

DispatcherServletAutoConfiguration中：

```

@Bean(
    name = {"dispatcherServletRegistration"}
)
@ConditionalOnBean(
    value = {DispatcherServlet.class},
    name = {"dispatcherServlet"}
)

```

```

    public DispatcherServletRegistrationBean
    dispatcherServletRegistration(DispatcherServlet dispatcherServlet,
    webMvcProperties webMvcProperties, ObjectProvider<MultipartConfigElement>
    multipartConfig) {
        DispatcherServletRegistrationBean registration = new
        DispatcherServletRegistrationBean(dispatcherServlet,
        webMvcProperties.getServlet().getPath());

        //默认拦截: / (所有请求; 包括静态资源, 但是不拦截jsp请求) /*会拦截jsp
        //可以通过servlet.servletPath来修改springmvc前端控制器默认拦截的请求路径
        registration.setName("dispatcherServlet");

        registration.setLoadOnStartup(webMvcProperties.getServlet().getLoadOnStartup());
        multipartConfig.ifAvailable(registration::setMultipartConfig);
        return registration;
    }

```

替换为其他嵌入式Servlet容器



默认支持:

Tomcat (默认使用)

```

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <!-- 引入web模块默认就是使用嵌入式的Tomcat作为Servlet容器; -->
</dependency>

```

Jetty

```

<!-- 引入web模块 -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <exclusions>
        <exclusion>
            <artifactId>spring-boot-starter-tomcat</artifactId>
            <groupId>org.springframework.boot</groupId>
        </exclusion>
    </exclusions>
</dependency>

<!--引入其他的Servlet容器-->
<dependency>
    <artifactId>spring-boot-starter-jetty</artifactId>
    <groupId>org.springframework.boot</groupId>
</dependency>

```

Undertow

```
<!-- 引入web模块 -->
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
  <exclusions>
    <exclusion>
      <artifactId>spring-boot-starter-tomcat</artifactId>
      <groupId>org.springframework.boot</groupId>
    </exclusion>
  </exclusions>
</dependency>

<!--引入其他的servlet容器-->
<dependency>
  <artifactId>spring-boot-starter-undertow</artifactId>
  <groupId>org.springframework.boot</groupId>
</dependency>
```

嵌入式Servlet容器自动配置原理；

EmbeddedWebServerFactoryCustomizerAutoConfiguration：嵌入式的Servlet容器自动配置？
servlet包下的的ServletWebServerFactoryConfiguration

```
@Configuration(                //声明为配置类
    proxyBeanMethods = false
)
@ConditionalOnWebApplication    //web应用才生效
@EnableConfigurationProperties({ServerProperties.class})
// 2.0前
//导入BeanPostProcessorsRegistrar: Spring注解版：给容器中导入一些组件
//导入了EmbeddedServletContainerCustomizerBeanPostProcessor:
//后置处理器：bean初始化前后（创建完对象，还没赋值赋值）执行初始化工作

//2.0后和serverProperties绑定了
public class EmbeddedWebServerFactoryCustomizerAutoConfiguration {

    //此方法在org.springframework.boot.autoconfigure.web.servlet的
    ServletWebServerFactoryConfiguration中
    @Configuration(
        proxyBeanMethods = false
    )
    @ConditionalOnClass({Servlet.class, Tomcat.class, UpgradeProtocol.class})
    //判断当前是否引入了tomcat依赖
    @ConditionalOnMissingBean(
        value = {ServletWebServerFactory.class},
        search = SearchStrategy.CURRENT
    )                //判断当前容器中没有用户自定义的ServletWebServerFactory: 嵌入式的
    servlet容器工厂；作用：创建嵌入式的servlet容器
    static class EmbeddedTomcat {
        EmbeddedTomcat() {
        }

        //导入了tomcat依赖。就会创建tomcatservlet的嵌入式工厂
        @Bean
```

```

        TomcatServletWebServerFactory
tomcatServletWebServerFactory(ObjectProvider<TomcatConnectorCustomizer>
connectorCustomizers, ObjectProvider<TomcatContextCustomizer>
contextCustomizers, ObjectProvider<TomcatProtocolHandlerCustomizer<?>>
protocolHandlerCustomizers) {
    TomcatServletWebServerFactory factory = new
TomcatServletWebServerFactory();

    factory.getTomcatConnectorCustomizers().addAll((Collection)connectorCustomizers
.orderedStream().collect(Collectors.toList()));

    factory.getTomcatContextCustomizers().addAll((Collection)contextCustomizers.ord
eredStream().collect(Collectors.toList()));

    factory.getTomcatProtocolHandlerCustomizers().addAll((Collection)protocolHandle
rCustomizers.orderedStream().collect(Collectors.toList()));
    return factory;
}
}

/**
 * Nested configuration if Jetty is being used.
 */
@Configuration(
    proxyBeanMethods = false
)
//server是jetty包下的依赖
@ConditionalOnClass({Servlet.class, Server.class, Loader.class,
webAppContext.class})
@ConditionalOnMissingBean(
    value = {ServletWebServerFactory.class},
    search = SearchStrategy.CURRENT
)
static class EmbeddedJetty {
    EmbeddedJetty() {
    }

    //导入jetty依赖，就创建jetty的工厂
    @Bean
    JettyServletWebServerFactory
JettyServletWebServerFactory(ObjectProvider<JettyServerCustomizer>
serverCustomizers) {
        JettyServletWebServerFactory factory = new
JettyServletWebServerFactory();

        factory.getServerCustomizers().addAll((Collection)serverCustomizers.orderedStre
am().collect(Collectors.toList()));
        return factory;
    }
}

/**
 * Nested configuration if Undertow is being used.
 */
@Configuration(
    proxyBeanMethods = false
)
@ConditionalOnClass({Servlet.class, Undertow.class,
sslClientAuthMode.class})

```

```

@ConditionalOnMissingBean(
    value = {ServletWebServerFactory.class},
    search = SearchStrategy.CURRENT
)
static class EmbeddedUndertow {
    EmbeddedUndertow() {

        //导入的Undertow依赖，就创建Undertow工厂
        @Bean
        UndertowServletWebServerFactory
        undertowServletWebServerFactory(ObjectProvider<UndertowDeploymentInfoCustomizer>
        deploymentInfoCustomizers, ObjectProvider<UndertowBuilderCustomizer>
        builderCustomizers) {
            UndertowServletWebServerFactory factory = new
            UndertowServletWebServerFactory();

            factory.getDeploymentInfoCustomizers().addAll((Collection)deploymentInfoCustomi
            zers.orderedStream().collect(Collectors.toList()));

            factory.getBuilderCustomizers().addAll((Collection)builderCustomizers.orderedSt
            ream().collect(Collectors.toList()));
            return factory;
        }
    }
}

```

1)、EmbeddedServletContainerFactory (嵌入式Servlet容器工厂)

```

@FunctionalInterface
public interface ServletWebServerFactory {
    //获取嵌入式的servlet容器
    WebServer getWebServer(ServletContextInitializer... initializers);
}

```



2)、EmbeddedServletContainer: (嵌入式的Servlet容器)



3)、以TomcatEmbeddedServletContainerFactory为例

```

//org.springframework.boot.web.embedded.tomcat下TomcatServletWebServerFactory类

public WebServer getWebServer(ServletContextInitializer... initializers) {
    if (this.disableMBeanRegistry) {
        Registry.disableRegistry();
    }

    //创建一个tomcat
    Tomcat tomcat = new Tomcat();

    //配置tomcat的基本环境
    File baseDir = this.baseDirectory != null ? this.baseDirectory :
    this.createTempDir("tomcat");
}

```

```

tomcat.setBaseDir(baseDir.getAbsolutePath());
Connector connector = new Connector(this.protocol);
connector.setThrowOnFailure(true);
tomcat.getService().addConnector(connector);
this.customizeConnector(connector);
tomcat.setConnector(connector);
tomcat.getHost().setAutoDeploy(false);
this.configureEngine(tomcat.getEngine());
Iterator var5 = this.additionalTomcatConnectors.iterator();

while(var5.hasNext()) {
    Connector additionalConnector = (Connector)var5.next();
    tomcat.getService().addConnector(additionalConnector);
}

this.prepareContext(tomcat.getHost(), initializers);

//将配置好的tomcat传入进去，返回一个WebServer嵌入式容器，并启动tomcat容器
//在org.springframework.boot.web.embedded.tomcat类的TomcatWebServer类下
TomcatWebServer方法
return this.getTomcatWebServer(tomcat);
}

```

4)、我们对嵌入式容器的配置修改是怎么生效?

ServerProperties、WebServerFactoryCustomizer<ConfigurableServletWebServerFactory>

WebServerFactoryCustomizer：定制器帮我们修改了Servlet容器的配置?

怎么修改的原理?

5)、容器中导入了WebServerFactoryCustomizerBeanPostProcessor

```

//这些方法在org.springframework.boot.web.server下
WebServerFactoryCustomizerBeanPostProcessor类

//初始化之前
@Override
public Object postProcessBeforeInitialization(Object bean, String beanName)
    throws BeansException {
    //如果当前初始化的是一个ConfigurableEmbeddedServletContainer类型的组件
    if (bean instanceof ConfigurableEmbeddedServletContainer) {
        //
        postProcessBeforeInitialization((ConfigurableEmbeddedServletContainer)
bean);
    }
    return bean;
}

private void postProcessBeforeInitialization(
    ConfigurableEmbeddedServletContainer bean) {
    //获取所有的定制器，调用每一个定制器的customize方法来给Servlet容器进行属性赋值；
    for (EmbeddedServletContainerCustomizer customizer : getCustomizers()) {
        customizer.customize(bean);
    }
}

```



```

}

private Collection<EmbeddedServletContainerCustomizer> getCustomizers() {
    if (this.customizers == null) {
        // Look up does not include the parent context
        this.customizers = new ArrayList<EmbeddedServletContainerCustomizer>(
            this.beanFactory
            //从容器中获取所有这个类型的组件: EmbeddedServletContainerCustomizer
            //定制Servlet容器, 给容器中可以添加一个EmbeddedServletContainerCustomizer
            类型的组件
            .getBeansOfType(EmbeddedServletContainerCustomizer.class,
                false, false)
            .values());
        Collections.sort(this.customizers,
            AnnotationAwareOrderComparator.INSTANCE);
        this.customizers = Collections.unmodifiableList(this.customizers);
    }
    return this.customizers;
}

ServerProperties也是定制器

```

步骤:

- 1)、SpringBoot根据导入的依赖情况, 给容器中添加相应的WebServerFactoryCustomizerConfiguration 【TomcatWebServerFactoryCustomizerConfiguration】
- 2)、容器中某个组件要创建对象就会惊动后置处理器;
WebServerFactoryCustomizerBeanPostProcessor;
只要是嵌入式的Servlet容器工厂, 后置处理器就工作;
- 3)、后置处理器, 从容器中获取所有的**EmbeddedServletContainerCustomizer**, 调用定制器的定制方法

嵌入式Servlet容器启动原理;

什么时候创建嵌入式的Servlet容器工厂? 什么时候获取嵌入式的Servlet容器并启动Tomcat;

获取嵌入式的Servlet容器工厂:

- 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;

```

public void refresh() throws BeansException, IllegalStateException {
    synchronized (this.startupShutdownMonitor) {
        // Prepare this context for refreshing.
        prepareRefresh();

        // Tell the subclass to refresh the internal bean factory.
        ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory();
    }
}

```

```

// Prepare the bean factory for use in this context.
prepareBeanFactory(beanFactory);

try {
    // Allows post-processing of the bean factory in context subclasses.
    postProcessBeanFactory(beanFactory);

    // Invoke factory processors registered as beans in the context.
    invokeBeanFactoryPostProcessors(beanFactory);

    // Register bean processors that intercept bean creation.
    registerBeanPostProcessors(beanFactory);

    // Initialize message source for this context.
    initMessageSource();

    // Initialize event multicaster for this context.
    initApplicationEventMulticaster();

    // Initialize other special beans in specific context subclasses.
    onRefresh();

    // Check for listener beans and register them.
    registerListeners();

    // Instantiate all remaining (non-lazy-init) singletons.
    finishBeanFactoryInitialization(beanFactory);

    // Last step: publish corresponding event.
    finishRefresh();
}

catch (BeansException ex) {
    if (logger.isWarnEnabled()) {
        logger.warn("Exception encountered during context initialization - "
+
            "cancelling refresh attempt: " + ex);
    }

    // Destroy already created singletons to avoid dangling resources.
    destroyBeans();

    // Reset 'active' flag.
    cancelRefresh(ex);

    // Propagate exception to caller.
    throw ex;
}

finally {
    // Reset common introspection caches in Spring's core, since we
    // might not ever need metadata for singleton beans anymore...
    resetCommonCaches();
}
}
}

```

4)、onRefresh(); web的ioc容器重写了onRefresh方法

5)、webioc容器会创建嵌入式的Servlet容器; **createEmbeddedServletContainer()**;

6)、获取嵌入式的Servlet容器工厂:

EmbeddedServletContainerFactory containerFactory = getEmbeddedServletContainerFactory();

从ioc容器中获取EmbeddedServletContainerFactory 组件;

TomcatEmbeddedServletContainerFactory创建对象, 后置处理器一看是这个对象, 就获取所有的定制器来先定制Servlet容器的相关配置;

7)、**使用容器工厂获取嵌入式的Servlet容器**: this.embeddedServletContainer = containerFactory.getEmbeddedServletContainer(getSelfInitializer());

8)、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;

==IOC容器启动创建嵌入式的Servlet容器==

使用外置的Servlet容器

嵌入式Servlet容器: 应用打成可执行的jar

优点: 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】);

外置的Servlet容器: 外面安装Tomcat---应用war包的方式打包;

步骤

1)、必须创建一个war项目; (利用idea创建好目录结构)

2)、将嵌入式的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

```
public class ServletInitializer extends SpringBootServletInitializer {

    @Override
    protected SpringApplicationBuilder configure(SpringApplicationBuilder
application) {
        //传入springboot应用主程序
        return application.sources(SpringBootWebJspApplication.class);
    }

}
```

4)、启动服务器就可以使用;

原理

jar包: 执行SpringBoot主类的main方法, 启动ioc容器, 创建嵌入式的Servlet容器;

war包: 启动服务器, **服务器启动SpringBoot应用【SpringBootServletInitializer】**, 启动ioc容器;

servlet3.0 (Spring注解版):

8.2.4 Shared libraries / runtimes pluggability:

规则:

1)、服务器启动 (web应用启动) 会创建当前web应用里面每一个jar包里面ServletContainerInitializer实例:

2)、ServletContainerInitializer的实现放在jar包的META-INF/services文件夹下, 有一个名为javax.servlet.ServletContainerInitializer的文件, 内容就是ServletContainerInitializer的实现类的全类名

3)、还可以使用@HandlesTypes, 在应用启动的时候加载我们感兴趣的类;

流程:

1)、启动Tomcat

2)、org\springframework\spring-web\4.3.14.RELEASE\spring-web-4.3.14.RELEASE.jar\META-INF\services\javax.servlet.ServletContainerInitializer:

Spring的web模块里面有这个文件: **org.springframework.web.SpringServletContainerInitializer**

3)、SpringServletContainerInitializer将@HandlesTypes(WebApplicationInitializer.class)标注的所有这个类型的类都传入到onStartup方法的Set<Class<?>>; 为这些WebApplicationInitializer类型的类创建实例;

4)、每一个WebApplicationInitializer都调用自己的onStartup;

 image-20201126174005166

5)、相当于我们的SpringBootServletInitializer的类会被创建对象, 并执行onStartup方法

6)、SpringBootServletInitializer实例执行onStartup的时候会createRootApplicationContext; 创建容器

```
protected WebApplicationContext createRootApplicationContext(  
    ServletContext servletContext) {  
    //1、创建SpringApplicationBuilder  
    SpringApplicationBuilder builder = createSpringApplicationBuilder();  
    StandardServletEnvironment environment = new StandardServletEnvironment();  
    environment.initPropertySources(servletContext, null);  
    builder.environment(environment);  
    builder.main(getClass());  
    ApplicationContext parent =  
    getExistingRootWebApplicationContext(servletContext);  
    if (parent != null) {  
        this.logger.info("Root context already created (using as parent).");  
        servletContext.setAttribute(  

```

```

        WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE, null);
        builder.initializers(new
ParentContextApplicationContextInitializer(parent));
    }
    builder.initializers(
        new ServletContextApplicationContextInitializer(servletContext));
    builder.contextClass(AnnotationConfigEmbeddedWebApplicationContext.class);

    //调用configure方法，子类重写了这个方法，将SpringBoot的主程序类传入了进来
    builder = configure(builder);

    //使用builder创建一个Spring应用
    SpringApplication application = builder.build();
    if (application.getSources().isEmpty() && AnnotationUtils
        .findAnnotation(getClass(), Configuration.class) != null) {
        application.getSources().add(getClass());
    }
    Assert.state(!application.getSources().isEmpty(),
        "No SpringApplication sources have been defined. Either override the "
        + "configure method or add an @Configuration annotation");
    // Ensure error pages are registered
    if (this.registerErrorPageFilter) {
        application.getSources().add(ErrorPageFilterConfiguration.class);
    }
    //启动Spring应用
    return run(application);
}

```

7) 、Spring的应用就启动并且创建IOC容器

```

public ConfigurableApplicationContext run(String... args) {
    Stopwatch stopwatch = new Stopwatch();
    stopwatch.start();
    ConfigurableApplicationContext context = null;
    FailureAnalyzers analyzers = null;
    configureHeadlessProperty();
    SpringApplicationRunListeners listeners = getRunListeners(args);
    listeners.starting();
    try {
        ApplicationArguments applicationArguments = new
DefaultApplicationArguments(
            args);
        ConfigurableEnvironment environment = prepareEnvironment(listeners,
            applicationArguments);
        Banner printedBanner = printBanner(environment);
        context = createApplicationContext();
        analyzers = new FailureAnalyzers(context);
        prepareContext(context, environment, listeners, applicationArguments,
            printedBanner);

        //刷新IOC容器
        refreshContext(context);
        afterRefresh(context, applicationArguments);
        listeners.finished(context, null);
        stopwatch.stop();
        if (this.logStartupInfo) {
            new StartupInfoLogger(this.mainApplicationClass)

```

```
        .logStarted(getApplicationLog(), stopwatch);
    }
    return context;
}
catch (Throwable ex) {
    handleRunFailure(context, listeners, analyzers, ex);
    throw new IllegalStateException(ex);
}
}
```

==启动Servlet容器，再启动SpringBoot应用==

Docker

1、简介

Docker是一个开源的应用容器引擎；是一个轻量级容器技术；

Docker支持将软件编译成一个镜像；然后在镜像中各种软件做好配置，将镜像发布出去，其他使用者可以直接使用这个镜像；

运行中的这个镜像称为容器，容器启动是非常快速的。



2、核心概念

docker主机(Host): 安装了Docker程序的机器 (Docker直接安装在操作系统之上) ；

docker客户端(Client): 连接docker主机进行操作；

docker仓库(Registry): 用来保存各种打包好的软件镜像；

docker镜像(Images): 软件打包好的镜像；放在docker仓库中；

docker容器(Container): 镜像启动后的实例称为一个容器；容器是独立运行的一个或一组应用



使用Docker的步骤：

- 1)、安装Docker
- 2)、去Docker仓库找到这个软件对应的镜像；
- 3)、使用Docker运行这个镜像，这个镜像就会生成一个Docker容器；
- 4)、对容器的启动停止就是对软件的启动停止；

安装Docker

1)、安装linux虚拟机

- 1)、VMWare、VirtualBox (安装) ；
- 2)、导入虚拟机文件centos7-atguigu.ova；
- 3)、双击启动linux虚拟机;使用 root/ 123456登陆
- 4)、使用客户端连接linux服务器进行命令操作；

5)、设置虚拟机网络;

桥接网络===选好网卡===接入网线;

6)、设置好网络以后使用命令重启虚拟机的网络

```
service network restart
```

7)、查看linux的ip地址

```
ip addr
```

8)、使用客户端连接linux;

2)、在linux虚拟机上安装docker

步骤:

1、检查内核版本, 必须是3.10及以上

```
uname -r
```

2、安装docker

```
yum install docker
```

3、输入y确认安装

4、启动docker

```
[root@localhost ~]# systemctl start docker
```

```
[root@localhost ~]# docker -v
```

```
Docker version 1.12.6, build 3e8e77d/1.12.6
```

5、开机启动docker

```
[root@localhost ~]# systemctl enable docker
```

```
Created symlink from /etc/systemd/system/multi-user.target.wants/docker.service  
to /usr/lib/systemd/system/docker.service.
```

6、停止docker

```
systemctl stop docker
```

4、Docker常用命令&操作

1)、镜像操作

操作	命令	说明
检索	docker search 关键字 eg: docker search redis	我们经常去docker hub上检索镜像的详细信息, 如镜像的TAG。
拉取	docker pull 镜像名:tag	:tag是可选的, tag表示标签, 多为软件的版本, 默认是latest
列表	docker images	查看所有本地镜像
删除	docker rmi image-id	删除指定的本地镜像

<https://hub.docker.com/>

2)、容器操作

[centos7开放指定端口](#)

软件镜像（QQ安装程序）----运行镜像----产生一个容器（正在运行的软件，运行的QQ）；

步骤：

```
1、搜索镜像
[root@localhost ~]# docker search tomcat

2、拉取镜像
[root@localhost ~]# docker pull tomcat

3、根据镜像启动容器
docker run --name mytomcat -d tomcat:latest

4、docker ps
查看运行中的容器

5、停止运行中的容器
docker stop 容器的id

6、查看所有的容器
docker ps -a

7、启动容器
docker start 容器id

8、删除一个容器
docker rm 容器id

9、启动一个做了端口映射的tomcat
[root@localhost ~]# docker run -d -p 8888:8080 tomcat
-d: 后台运行
-p: 将主机的端口映射到容器的一个端口    主机端口:容器内部的端口

10、为了演示简单关闭了linux的防火墙
service firewalld status : 查看防火墙状态
service firewalld stop: 关闭防火墙

11、查看容器的日志
docker logs container-name/container-id
```

更多命令参考

<https://docs.docker.com/engine/reference/commandline/docker/>

可以参考每一个镜像的文档

开放端口后，却仍然访问不了，执行以下操作

当Tomcat版本过高时，根据IP地址和端口号访问可能会出现下面问题



出现404是因为webapps文件夹下内容为空，内容都在webapps.dist 目录下

1. 进入Tomcat容器

```
docker exec -it 容器id /bin/bash
```

2.ls 查看文件夹内容，可以发现下面有webapps文件夹和webapps.dist文件夹，将webapps.dist下的内容全部复制到webapps中


```
cp -r webapps.dist/* webapps
```

3)、安装MySQL示例

```
docker pull mysql
```

错误的启动

```
[root@localhost ~]# docker run --name mysql01 -d mysql
42f09819908bb72dd99ae19e792e0a5d03c48638421fa64cce5f8ba0f40f5846

mysql退出了
[root@localhost ~]# docker ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED
STATUS		PORTS	NAMES
42f09819908b	mysql	"docker-entrypoint.sh"	34 seconds ago
Exited (1)			mysql01
538bde63e500	tomcat	"catalina.sh run"	About an hour ago
Exited (143)			compassionate_
goldstine			
c4f1ac60b3fc	tomcat	"catalina.sh run"	About an hour ago
Exited (143)			lonely_fermi
81ec743a5271	tomcat	"catalina.sh run"	About an hour ago
Exited (143)			sick_ramanujan

```
//错误日志
[root@localhost ~]# docker logs 42f09819908b
error: database is uninitialized and password option is not specified
  You need to specify one of MYSQL_ROOT_PASSWORD, MYSQL_ALLOW_EMPTY_PASSWORD and
  MYSQL_RANDOM_ROOT_PASSWORD; 这个三个参数必须指定一个
```

正确的启动

```
[root@localhost ~]# docker run --name mysql01 -e MYSQL_ROOT_PASSWORD=123456 -d
mysql
b874c56bec49fb43024b3805ab51e9097da779f2f572c22c695305dedd684c5f
[root@localhost ~]# docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED
STATUS	PORTS	NAMES	
b874c56bec49	mysql	"docker-entrypoint.sh"	4 seconds ago
Up 3 seconds	3306/tcp	mysql01	

做了端口映射

```
[root@localhost ~]# docker run -p 3306:3306 --name mysql02 -e
MYSQL_ROOT_PASSWORD=123456 -d mysql
ad10e4bc5c6a0f61cbad43898de71d366117d120e39db651844c0e73863b9434
[root@localhost ~]# docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED
ad10e4bc5c6a	mysql	"docker-entrypoint.sh"	4 seconds ago
Up 2 seconds	0.0.0.0:3306->3306/tcp	mysql02	

再次启动: `docker restart` 镜像名

```
# 进入指定docker的mysql
docker exec -it id /bin/bash
# 在执行mysql登录命令
```

几个其他的高级操作

```
docker run --name mysql03 -v /conf/mysql:/etc/mysql/conf.d -e
MYSQL_ROOT_PASSWORD=my-secret-pw -d mysql:tag
```

把主机的/`conf/mysql`文件夹挂载到 `mysql03`容器的/`etc/mysql/conf.d`文件夹里面
改mysql的配置文件就只需要把mysql配置文件放在自定义的文件夹下（/`conf/mysql`）

```
docker run --name some-mysql -e MYSQL_ROOT_PASSWORD=my-secret-pw -d mysql:tag --
character-set-server=utf8mb4 --collation-server=utf8mb4_unicode_ci
```

指定mysql的一些配置参数

命令行和sqlyog登录不了执行以下操作
解决方案（在centos7环境下）：

1. 进入mysql 容器

```
docker exec -it mysql02 /bin/bash
```

2. 进入mysql

```
mysql -uroot -proot
```

3. 修改密码

```
ALTER USER 'root'@'%' IDENTIFIED WITH mysql_native_password BY 'root';
```

[最后一句的理解](#)

SpringBoot与数据访问

1、JDBC

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jdbc</artifactId>
</dependency>
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <scope>runtime</scope>
</dependency>
```

```
spring:
  datasource:
    username: root
    password: 123456
    url: jdbc:mysql://192.168.15.22:3306/jdbc
    driver-class-name: com.mysql.jdbc.Driver
```

效果：

默认是用com.zaxxer.hikari.HikariDataSource作为数据源；

数据源的相关配置都在DataSourceProperties里面；

自动配置原理：

org.springframework.boot.autoconfigure.jdbc:

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

2、SpringBoot默认可以支持；

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

3、自定义数据源类型

```
/**
 * Generic DataSource configuration.
 */
static class Generic {
    Generic() {
    }

    @Bean
    DataSource dataSource(DataSourceProperties properties) {
        //使用DataSourceBuilder创建数据源，利用反射创建响应type的数据源，并且绑定相关
        return properties.initializeDataSourceBuilder().build();
    }
}
```

4、DataSourceInitializer: ApplicationListener;

作用：

- 1) 、runSchemaScripts();运行建表语句;
- 2) 、runDataScripts();运行插入数据的sql语句;

默认只需要将文件命名为：

```
schema-*.sql、data-*.sql
默认规则: schema.sql, schema-all.sql;
不过boot2.0后配置文件要加:
置spring.datasource.initialization-mode:always; 才能执行resource下schema前缀的sql文件
可以使用
    schema:
        - classpath:department.sql    # 这里没空格
          指定位置
```

5、操作数据库：自动配置了JdbcTemplate操作数据库

2、整合Druid数据源

```
//导入druid数据源
/*
    <!-- 配置log4j, 否则DruidConfig配置类报错 -->
    <dependency>
        <groupId>log4j</groupId>
        <artifactId>log4j</artifactId>
        <version>1.2.17</version>
    </dependency>
*/
@Configuration
public class DruidConfig {

    @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<>();
    initParams.put("exclusions","*.js,*.css,/druid/*");

    bean.setInitParameters(initParams);

    bean.setUrlPatterns(Arrays.asList("/*"));

    return bean;
}
}
```

3、整合MyBatis

```
<dependency>
    <groupId>org.mybatis.spring.boot</groupId>
    <artifactId>mybatis-spring-boot-starter</artifactId>
    <version>2.1.4</version>
</dependency>
```



步骤：（和jdbc一样，只是换一个依赖）

- 1)、配置数据源相关属性（见上一节Druid）
- 2)、给数据库建表
- 3)、创建JavaBean（对应表的对象）

4)、注解版

```
//指定这是一个操作数据库的mapper
@Mapper //指定这是操作数据库的mapper类
public interface DepartmentMapper {
    /**
     * 根据id查部门
     * @param id
     * @return
     */
    @Select("select * from department where id = #{id}")
    public Department getDeptById(Integer id);

    /**
     * 插入部门
     * @param department
     * @return
     */
    @Insert("insert into department (departmentName) values (#{departmentName})")
    public Integer insertDept(Department department);
}
```

```

/**
 * 根据id删除指定部门
 * @param id
 * @return
 */
@Delete("delete from department where id = #{id} ")
public Integer deleteDeptById(Integer id);

/**
 * 更新部门信息
 * @param department
 * @return
 */
@Update("update department set departmentName = #{departmentName} where id =
#{id}")
public Integer updateDept(Department department);
}

```

问题:

自定义MyBatis的配置规则; 给容器中添加一个ConfigurationCustomizer;

```

@Configuration
public class MybatisConfig {
    /**
     * 开启驼峰命名规则
     * 也可以在配置文件中mybatis: configuration: map-underscore-to-camel-case: true
     * @return
     */
    @Bean
    public ConfigurationCustomizer configurationCustomizer() {
        return new ConfigurationCustomizer() {
            @Override
            public void customize(org.apache.ibatis.session.Configuration
configuration) {
                //开启驼峰命名规则
                configuration.setMapUnderscoreToCamelCase(true);
            }
        };
    }
}

```

使用MapperScan批量扫描所有的Mapper接口; mapper接口中的类就不用写@Mapper注解了

```

@SpringBootApplication
@MapperScan("com.fsir.springboot.mapper")
public class SpringbootDataMybatisApplication {

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

}

```

5)、配置文件版

```
mybatis:
  config-location: classpath:mybatis/mybatis-config.xml 指定全局配置文件的位置
  mapper-locations: classpath:mybatis/mapper/*.xml 指定sql映射文件的位置
```

更多使用参照

<http://www.mybatis.org/spring-boot-starter/mybatis-spring-boot-autoconfigure/>

4、整合SpringData JPA

1)、SpringData简介



2)、整合SpringData JPA

JPA:ORM (Object Relational Mapping) ;

1)、编写一个实体类 (bean) 和数据表进行映射, 并且配置好映射关系;

```
@Entity      //告诉jpa这是一个实体类（和数据库表映射的类）
@Table(name = "tbl_user")    //@Table指定和哪个数据库表对应；如果省略默认是表名(小写)user
public class User {
    @Id        //表示这是主键
    @GeneratedValue(strategy = GenerationType.IDENTITY) //自增主键
    private Integer id;

    @Column(name = "last_name", length = 50)    //指定和数据库表对应的列,长度
    private String lastName;

    @Column    //省略不写默认是属性名
    private String email;

    public User() {
    }

    public User(Integer id, String lastName, String email) {
        this.id = id;
        this.lastName = lastName;
        this.email = email;
    }

    @Override
    public String toString() {
        return "User{" +
            "id=" + id +
            ", lastName='" + lastName + '\'' +
            ", email='" + email + '\'' +
            '}';
    }

    public Integer getId() {
        return id;
    }
}
```

```

    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }
}

```

2)、编写一个Dao接口来操作实体类对应的数据表 (Repository)

```

/**
 * 继承JpaRepository完成对数据库的操作
 * 继承的接口的泛型第一个是要操作的实体类， 第二个是主键的包装类
 */
public interface UserRepository extends JpaRepository<UserRepository, Integer> {

}

```

3)、基本的配置JpaProperties

```

spring:
  datasource:
    password: root
    username: root
    url: jdbc:mysql://192.168.3.67:3307/jpa
    driver-class-name: com.mysql.cj.jdbc.Driver
  jpa:
    hibernate:
      ddl-auto: update      # 数据表达策略；更新创建数据表；没有操作的表就会创建，有就会进行
      # 修改表结构
      show-sql: true        # 表示操作sql都显示在控制台

```

启动配置原理

几个重要的事件回调机制

配置在META-INF/spring.factories

ApplicationContextInitializer

SpringApplicationRunListener

只需要放在ioc容器中

ApplicationRunner

CommandLineRunner

启动流程:

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META-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();
}
```



2、运行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 ();

    //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);
}
}

```

3、事件监听机制

配置在META-INF/spring.factories

ApplicationContextInitializer

```

/**
 * 应用程序上下文初始化器
 * 需要配置
 */

```

```

public class HelloApplicationContextInitializer implements
ApplicationContextInitializer<ConfigurableApplicationContext> {

    /**
     * 重写初始发方法
     * @param configurableApplicationContext
     */
    @Override
    public void initialize(ConfigurableApplicationContext
configurableApplicationContext) {
        System.out.println("22行
ApplicationContextInitializer...initialize():\t" +
configurableApplicationContext);
    }
}

```

SpringApplicationRunListener

```

/**
 * Spring应用程序运行监听器
 * 需要配置
 */
public class HelloSpringApplicationRunListener implements
SpringApplicationRunListener {
    /**
     * 必须有这个有参构造器，否则报错：参考SpringApplicationRunListener的
EventPublishingRunListener实现类
     * @param application
     * @param args
     */
    public HelloSpringApplicationRunListener(SpringApplication application,
String[] args) {

    }

    /**
     * 开始监听容器
     * @param bootstrapContext
     */
    @Override
    public void starting(ConfigurableBootstrapContext bootstrapContext) {
        System.out.println("34 SpringApplicationRunListener...starting:\t");
    }

    /**
     * 准备环境
     * @param bootstrapContext
     * @param environment
     */
    @Override
    public void environmentPrepared(ConfigurableBootstrapContext
bootstrapContext, ConfigurableEnvironment environment) {
        Object o = environment.getSystemProperties().get("os.name");
        System.out.println("45
SpringApplicationRunListener...environmentPrepared:\t" + o);
    }
}

```

```

    }

    /**
     * 容器准备完成
     * @param context
     */
    @Override
    public void contextPrepared(ConfigurableApplicationContext context) {
        System.out.println("54
SpringApplicationRunListener...contextPrepared:\t");
    }

    /**
     * 容器加载完成
     * @param context
     */
    @Override
    public void contextLoaded(ConfigurableApplicationContext context) {
        System.out.println("63
SpringApplicationRunListener...contextLoaded:\t");
    }

    /**
     * 启动完成
     * @param context
     */
    @Override
    public void started(ConfigurableApplicationContext context) {
        System.out.println("72  SpringApplicationRunListener...started:\t");
    }
}

```

配置 (META-INF/spring.factories)

```

# 可参考spring-boot-autoconfigure的META-INF的spring.factories
# 自定义的ApplicationContextInitializer和SpringApplicationRunListener所在包路径
# Initializers
org.springframework.context.ApplicationContextInitializer=\
com.fsir.springboot.listener.HelloApplicationContextInitializer

org.springframework.boot.SpringApplicationRunListener=\
com.fsir.springboot.listener.HelloSpringApplicationRunListener

```

只需要放在ioc容器中

ApplicationRunner

```

@Component          //加入到spring容器中
public class HelloApplicationRunner implements ApplicationRunner {
    @Override
    public void run(ApplicationArguments args) throws Exception {
        System.out.println("16 ApplicationRunner...run...\t");
    }
}

```

CommandLineRunner

```

@Component          //加入到spring容器中
public class HelloCommandLineRunner implements CommandLineRunner {
    @Override
    public void run(String... args) throws Exception {
        System.out.println("17 CommandLineRunner...run...\t" +
Arrays.asList(args));
    }
}

```

自定义starter

starter:

- 1、这个场景需要使用到的依赖是什么？
- 2、如何编写自动配置

```

@Configuration    //指定这个类是一个配置类
@ConditionalOnXXX //在指定条件成立的情况下自动配置类生效
@AutoConfigureAfter //指定自动配置类的顺序
@Bean             //给容器中添加组件

@ConfigurationProperty //结合相关xxxProperties类来绑定相关的配置
@EnableConfigurationProperties //让xxxProperties生效加入到容器中

//自动配置类要能加载
//将需要启动就加载的自动配置类，配置在META-INF/spring.factories(可参考spring-boot-
autoconfigure的META-INF的spring.factories)

# Auto Configure
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
org.springframework.boot.autoconfigure.admin.SpringApplicationAdminJmxAutoConfig
uration,\
org.springframework.boot.autoconfigure.aop.AopAutoConfiguration,\

```

- 3、模式:

启动器只用来做依赖导入;

专门来写一个自动配置模块;

启动器依赖自动配置; 别人只需要引入启动器 (starter)

mybatis-spring-boot-starter; 自定义启动器名-spring-boot-starter

步骤:

1)、启动器模块

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

    <groupId>com.atguigu.starter</groupId>
    <artifactId>atguigu-spring-boot-starter</artifactId>
    <version>1.0-SNAPSHOT</version>

    <!--启动器-->
    <dependencies>

        <!--引入自动配置模块-->
        <dependency>
            <groupId>com.atguigu.starter</groupId>
            <artifactId>atguigu-spring-boot-starter-autoconfigurer</artifactId>
            <version>0.0.1-SNAPSHOT</version>
        </dependency>
    </dependencies>

</project>
```

2)、自动配置模块

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

    <groupId>com.atguigu.starter</groupId>
    <artifactId>atguigu-spring-boot-starter-autoconfigurer</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>

    <name>atguigu-spring-boot-starter-autoconfigurer</name>
    <description>Demo project for Spring Boot</description>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.5.10.RELEASE</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>

    <properties>
```

```

    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
    <java.version>1.8</java.version>
</properties>

<dependencies>

    <!--引入spring-boot-starter; 所有starter的基本配置-->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter</artifactId>
    </dependency>

</dependencies>

</project>

```

```

package com.atguigu.starter;

import org.springframework.boot.context.properties.ConfigurationProperties;

@ConfigurationProperties(prefix = "atguigu.hello")
public class HelloProperties {

    private String prefix;
    private String suffix;

    public String getPrefix() {
        return prefix;
    }

    public void setPrefix(String prefix) {
        this.prefix = prefix;
    }

    public String getSuffix() {
        return suffix;
    }

    public void setSuffix(String suffix) {
        this.suffix = suffix;
    }
}

```

```

package com.atguigu.starter;

public class HelloService {

    HelloProperties helloProperties;
}

```

```

    public HelloProperties getHelloProperties() {
        return helloProperties;
    }

    public void setHelloProperties(HelloProperties helloProperties) {
        this.helloProperties = helloProperties;
    }

    public String sayHellAtguigu(String name){
        return helloProperties.getPrefix()+"-" +name +
helloProperties.getSuffix();
    }
}

```

```

package com.atguigu.starter;

import org.springframework.beans.factory.annotation.Autowired;
import
org.springframework.boot.autoconfigure.condition.ConditionalOnWebApplication;
import
org.springframework.boot.context.properties.EnableConfigurationProperties;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
@ConditionalOnWebApplication //web应用才生效
@EnableConfigurationProperties(HelloProperties.class)
public class HelloServiceAutoConfiguration {

    @Autowired
    HelloProperties helloProperties;
    @Bean
    public HelloService helloService(){
        HelloService service = new HelloService();
        service.setHelloProperties(helloProperties);
        return service;
    }
}

```

更多SpringBoot整合示例

<https://github.com/spring-projects/spring-boot/tree/master/spring-boot-samples>

springboot高级

JSR107

JSR-107（缓存规范）、Spring缓存抽象、整合Redis

可参考springboot-01-cache项目

Spring从3.1开始定义了org.springframework.cache.**Cache** 和 org.springframework.cache.**CacheManager** (**缓存管理器**) 接口来统一不同的缓存技术； 并支持使用**Cache (JSR-107)** 注解简化我们开发；

- Cache接口为缓存的组件规范定义，包含缓存的各种操作集合；
- Cache接口下Spring提供了各种xxxCache的实现；如RedisCache， EhCacheCache， ConcurrentMapCache等；
- 每次调用需要缓存功能的方法时， Spring会检查指定参数的指定的目标方法是否 已经被调用过；如果有就直接从缓存中获取方法调用后的结果，如果没有就调用方法 并缓存结果后返回给用户。下次调用直接从缓存中获取。
- 使用Spring缓存抽象时我们需要关注以下两点；
 - 1、确定方法需要被缓存以及他们的缓存策略
 - 2、从缓存中读取之前缓存存储的数据

JSR-107规范

Java Caching定义了5个核心接口，分别是CachingProvider(缓存提供者)、CacheManager(缓存管理器)、Cache(缓存)、Entry(缓存键值对)和Expiry(缓存时效)。

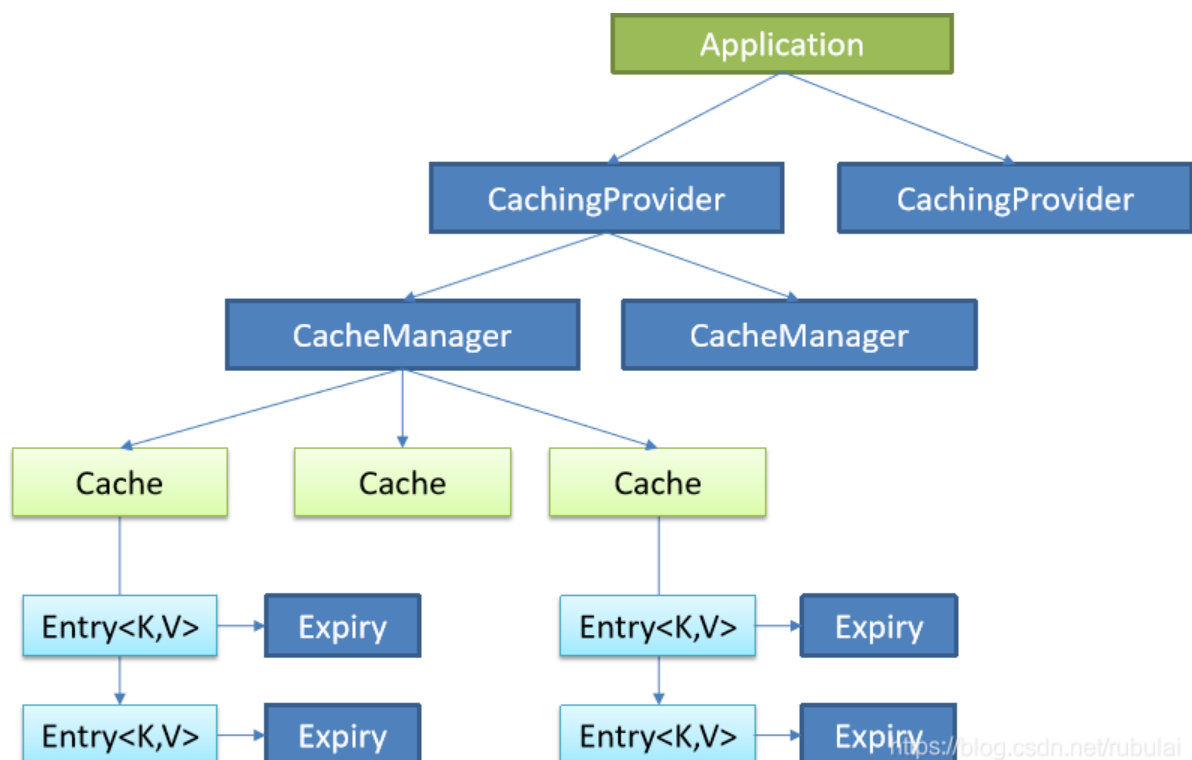
==CachingProvider(缓存提供者)==定义了创建、配置、获取、管理和控制多个**CacheManager**。一个应用可以在运行期访问多个CachingProvider。

==CacheManager(缓存管理器)==定义了创建、配置、获取、管理和控制多个唯一命名的Cache，这些Cache存在于CacheManager的上下文中。一个CacheManager仅被一个CachingProvider所拥有。

==Cache==是一个类似Map的数据结构并临时存储以Key为索引的值。一个**Cache**仅被一个CacheManager所拥有。

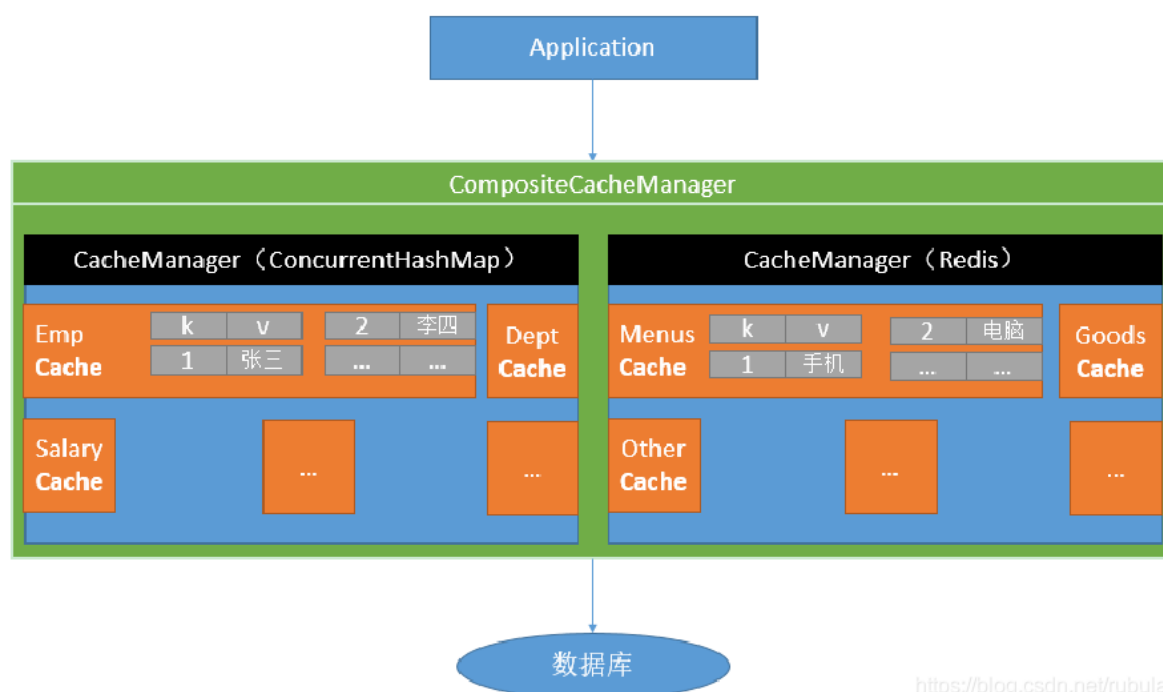
==Entry==是一个存储在Cache中的key-value对。

==Expiry== 每一个存储在Cache中的条目有一个定义的有效期。一旦超过这个时间，条目为过期的状态。一旦过期，条目将不可访问、更新和删除。缓存有效期可以通过ExpiryPolicy设置



几个重要概念&缓存注解

Cache	缓存接口，定义缓存操作。实现有：RedisCache、EhCacheCache、ConcurrentMapCache等
CacheManager	缓存管理器，管理各种缓存（Cache）组件
@Cacheable	主要针对方法配置，能够根据方法的请求参数对其结果进行缓存
@CacheEvict	清空缓存
@CachePut	保证方法被调用，又希望结果被缓存。
@EnableCaching	开启基于注解的缓存
keyGenerator	缓存数据时key生成策略
serialize	缓存数据时value序列化策略



@Cacheable/@CachePut/@CacheEvict 主要的参数

value	缓存的名称，在 spring 配置文件中定义，必须 指定至少一个	例如： <code>@Cacheable(value="mycache")</code> 或者 <code>@Cacheable(value={"cache1","cache2"})</code>
key	缓存的 key，可以为空，如果指定要按照 SpEL 表达式编写，如果不指定，则缺省按照方法的所有参数进行组合	例如： <code>@Cacheable(value="testcache",key="#userName")</code>
condition	缓存的条件，可以为空，使用 SpEL 编写，返回 true 或者 false，只有为 true 才进行缓存/清除缓存	例如： <code>@Cacheable(value="testcache",condition="#userName.length()>2")</code>
allEntries(@CacheEvict)	是否清空所有缓存内容，缺省为 false，如果指定为 true，则方法调用后将立即清空所有缓存	例如： <code>@CacheEvict(value="testcache",allEntries=true)</code>
beforeInvocation(@CacheEvict)	是否在方法执行前就清空，缺省为 false，如果指定为 true，则在方法还没有执行的时候就清空缓存，缺省情况下，如果方法执行抛出异常，则不会清空缓存	例如： <code>@CacheEvict(value="testcache",beforeInvocation=true)</code>

key可取的参数

 image-20201129221901691

整合redis实现缓存

docker安装redis: `docker pull redis`

docker启动redis: `-d`: 后台方式启动; `-p`: 暴露端口

自定义此redis的名称 启动docker的程序(redis)

`docker run -d -p6379:6379 --name myredis redis`

1. 引入spring-boot-starter-data-redis
2. application.yml配置redis连接地址
3. 配置缓存
 1. @EnableCaching、
 2. CachingConfigurerSupport、
4. 试使用缓存
5. 切换为其他缓存&CompositeCacheManager

Spring Boot与消息

1. 在大多应用中，我们系统之间需要进行异步通信，即异步消息。
2. 异步消息中两个重要概念：
消息代理 (message broker) 和目的地 (destination) 当消息发送者发送消息以后，将由消息代理接管，消息代理保证消息传递到指定目的地。
3. 异步消息主要有两种形式的目的地
 1. **队列 (queue)**：点对点消息通信 (point-to-point)
 2. **主题 (topic)**：发布 (publish) / 订阅 (subscribe) 消息通信
4. 点对点式：
 1. 消息发送者发送消息，消息代理将其放入一个队列中，消息接收者从队列中获取消息内容，消息读取后被移出队列
 2. 消息只有唯一的发送者和接受者，但并不是说只能有一个接收者
5. 发布订阅式：
 1. 发送者 (发布者) 发送消息到主题，多个接收者 (订阅者) 监听 (订阅) 这个主题，那么就会在消息到达时同时收到消息
6. JMS (Java Message Service) java消息服务：
 1. 基于JVM消息代理的规范。ActiveMQ、HornetMQ是JMS实现
7. AMQP (Advanced Message Queuing Protocol)
 1. 高级消息队列协议，也是一个消息代理的规范，兼容JMS
 2. RabbitMQ是AMQP的实现
8. Spring支持
 1. **spring-jms提供了对JMS的支持**
 2. **spring-rabbit提供了对AMQP的支持**
 3. **需要ConnectionFactory的实现来连接消息代理**
 4. **提供JmsTemplate、RabbitTemplate来发送消息**
 5. **@JmsListener (JMS)、@RabbitListener (AMQP) 注解在方法上监听消息代理发布的消息**
 6. **@EnableJms、@EnableRabbit开启支持**
9. Spring Boot自动配置
 1. **JmsAutoConfiguration**
 2. **RabbitAutoConfiguration**

RabbitMQ简介

RabbitMQ简介：

RabbitMQ是一个由erlang开发的AMQP(Advanced Message Queue)的开源实现。

核心概念

==Producer&Consumer==

producer指的是消息生产者，consumer消息的消费者。

==Broker==

它提供一种传输服务,它的角色就是维护一条从生产者到消费者的路线，保证数据能按照指定的方式进行传输，

==Queue==

消息队列，提供了FIFO的处理机制，具有缓存消息的能力。rabbitmq中，队列消息可以设置为持久化，临时或者自动删除。

设置为持久化的队列，queue中的消息会在server本地硬盘存储一份，防止系统crash，数据丢失

设置为临时队列，queue中的数据在系统重启之后就会丢失

设置为自动删除的队列，当不存在用户连接到server，队列中的数据会被自动删除

==Exchange==

消息交换机，它指定消息按什么规则，路由到哪个队列。

Exchange有4种类型：direct(默认)，fanout，topic，和headers，不同类型的Exchange转发消息的策略有所区别：

==Binding==

将一个特定的Exchange和一个特定的Queue绑定起来。

Exchange和Queue的绑定可以是多对多的关系。

==virtual host (vhosts) ==

在rabbitmq server上可以创建多个虚拟的message broker，又叫做 virtual hosts (vhosts)

每一个vhost本质上是一个mini-rabbitmq server，分别管理各自的 exchange，和bindings

vhost相当于物理的server，可以为不同app提供边界隔离

producer和consumer连接rabbit server需要指定一个vhost

RabbitMQ运行机制

RabbitMQ整合

docker安装rabbitmq

management有这种后缀的，是带web管理界面的

1、docker安装rabbitmq: `docker pull rabbitmq:3.8.9-management`

-d: 后台运行；-p: 暴露端口(有两个端口，6572: 客户端和容器进行通信的端口，15672管理页面的端口)；--name 给此容器起个名称

2、使用docker启动rabbitmq: `docker run -d -p 5672:5672 -p 15672:15672 --name myrabbitmq 镜像id(image id)`

3、ip:15672进入管理界面（管理界面的账号和密码都是：guest）

1. 引入spring-boot-starter-amqp

2. application.yml配置

3. 测试RabbitMQ

整合测试项目：D:\java_project\VM\shangguigu\springboot-amqp

Spring Boot与检索

ElasticSearch

我们的应用经常需要添加检索功能，更或者是大量日志检索分析等，Spring Boot 通过整合Spring Data ElasticSearch为我们提供了非常便捷的检索功能支持；

Elasticsearch是一个分布式搜索服务，提供Restful API，底层基于Lucene，采用多shard的方式保证数据安全，并且提供自动resharding的功能，github等大型的 站点也是采用了Elasticsearch作为其搜索服务，

概念

- 以 员工文档 的形式存储为例：一个文档代表一个员工数据。存储数据到 Elasticsearch 的行为叫做索引，但在索引一个文档之前，需要确定将文档存 储在哪里。
- 一个 Elasticsearch 集群可以 包含多个 索引，相应的每个索引可以包含多 个 类型。这些不同的类型存储着多个 文档，每个文档又有 多个 属性。
- 类似关系：

索引-数据库

类型-表

文档-表中的记录

属性-列

docker安装：

`docker pull elasticsearch:版本号`

`docker pull ustc-edu-cn.mirror.aliyuncs.com/library/elasticsearch:` 加速拉取

docker启动elasticsearch：因为elasticsearch是用java写的，初始会占用2G内存，`-e`限制内存使用
初始使用256m，最大使用也是256m，`-d`后台运行，`-p`端口映射(web通信默认9200，分布式时测试各个企业间的通信用9300)

`docker run -e ES_JAVA_OPTS="-Xms256m -Xmx256m" -d -p 9200:9200 -p 9300:9300 -`
`-name ES01 镜像名或镜像id`

整合ElasticSearch测试

- 引入spring-boot-starter-data-elasticsearch
 - 安装Spring Data 对应版本的ElasticSearch
 - application.yml配置
 - Spring Boot自动配置的
 - ElasticsearchRepository、Client
 - 测试ElasticSearch
-

Spring Boot与任务

异步任务、定时任务、邮件任务

异步任务

在Java应用中，绝大多数情况下都是通过同步的方式来实现交互处理的；但是在 处理与第三方系统交互的时候，容易造成响应迟缓的情况，之前大部分都是使用 多线程来完成此类任务，其实，在Spring 3.x之后，就已经内置了@Async来完美解决这个问题。

==两个注解：==

@EnableAysnc、@Aysnc

定时任务

项目开发中经常需要执行一些定时任务，比如需要在每天凌晨时候，分析一次前 一天的日志信息。Spring为我们提供了异步执行任务调度的方式，提供 TaskExecutor 、TaskScheduler 接口。

==两个注解==： @EnableScheduling、@Scheduled

字段	允许值	允许的特殊字符
秒	0-59	, - * /
分	0-59	, - * /
小时	0-23	, - * /
日期	1-31	, - * ? / L W C
月份	1-12或JAN-DEC	, - * /
星期	1-7或SUN-SAT	, - * ? / L C #
年（可选）	空,1970-2099	, - * /

特殊字符	代表含义
,	枚举
-	区间
*	任意
/	步长
?	日/星期冲突匹配
L	最后
W	工作日
C	和calendar联系后计算过的值
#	星期, 4#2, 第2个星期三

邮件任务

- 邮件发送需要引入spring-boot-starter-mail
- Spring Boot 自动配置MailSenderAutoConfiguration
- 定义MailProperties内容，配置在application.yml中
- 自动装配JavaMailSender
- 测试邮件发送

Spring Boot与安全

安全、Spring Security

安全

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

几个类：

WebSecurityConfigurerAdapter：自定义Security策略

AuthenticationManagerBuilder：自定义认证策略

@EnableWebSecurity：开启WebSecurity模式

- 应用程序的两个主要区域是“认证”和“授权”（或者访问控制）。这两个主要区域是Spring Security的两个目标。
- “认证”，是建立一个他声明的主体的过程（一个“主体”一般是指 用户，设备或一些可以在你的应用程序中执行动作的其他系统）。
- “授权”指确定一个主体是否允许在你的应用程序执行一个动作的过程。为了抵达需要授权的店，主体的身份已经有认证过程建立。
- 这个概念是通用的而不只在Spring Security中

Web&安全

1. CSRF (Cross-site request forgery) 跨站请求伪造

HttpSecurity启用csrf功能

2. 登陆/注销

HttpSecurity配置登陆、注销功能

3. remember me

表单添加remember-me的checkbox

配置启用remember-me功能

4. Thymeleaf提供的SpringSecurity标签支持

需要引入thymeleaf-extras-springsecurity4

sec:authentication="name"获得当前用户的用户名

sec:authorize="hasRole('ADMIN')"当前用户必须拥有ADMIN权限时才会显示标签内容

Spring Boot与分布式

分步式、Spring Boot/Cloud、Dubbo/Zookeeper

分布式应用

在分布式系统中，国内常用zookeeper+dubbo组合，而Spring Boot推荐使用全栈的Spring，Spring Boot+Spring Cloud

- **单一应用架构**

当网站流量很小时，只需一个应用，将所有功能都部署在一起，以减少部署节点和成本。此时，用于简化增删改查工作量的数据访问框架(ORM)是关键。

- **垂直应用架构**

当访问量逐渐增大，单一应用增加机器带来的加速度越来越小，将应用拆成互不相干的几个应用，以提升效率。此时，用于加速前端页面开发的Web框架(MVC)是关键。

- **分布式服务架构**

当垂直应用越来越多，应用之间交互不可避免，将核心业务抽取出来，作为独立的服务，逐渐形成稳定的服务中心，使前端应用能更快速的响应多变的市场需求。此时，用于提高业务复用及整合的分布式服务框架(RPC)是关键。

- **流动计算架构**

当服务越来越多，容量的评估，小服务资源的浪费等问题逐渐显现，此时需增加一个调度中心基于访问压力实时管理集群容量，提高集群利用率。此时，用于提高机器利用率的资源调度和治理中心(SOA)是关键

Zookeeper和Dubbo

- ZooKeeper 服务注册中心

ZooKeeper 是一个分布式的，开放源码的分布式应用程序协调服务。它是一个为分布式应用提供一致性服务的软件，提供的功能包括：配置维护、域名服务、分布式同步、组服务等。

- Dubbo

Dubbo是Alibaba开源的分布式服务框架，它最大的特点是按照分层的方式来架构，使用这种方式可以使各个层之间解耦合（或者最大限度地松耦合）。从服务模型的角度来看，Dubbo采用的是一种非常简单的模型，要么是提供方提供服务，要么是消费方消费服务，所以基于这一点可以抽象出服务提供方（Provider）和服务消费方（Consumer）两个角色

整合dubbo

引入spring-boot-starter-dubbo

```
<dependency>
  <groupId>com.gitee.reger</groupId>
  <artifactId>spring-boot-starter-dubbo</artifactId>
  <version>1.0.4</version>
</dependency>
```

配置服务提供者与消费者

测试

Spring Boot和Spring Cloud

- Spring Cloud

Spring Cloud是一个分布式的整体解决方案。Spring Cloud 为开发者提供了**在分布式系统（配置管理，服务发现，熔断，路由，微代理，控制总线，一次性token，全局锁，leader选举，分布式session，集群状态）中快速构建的工具**，使用Spring Cloud的开发者可以快速的启动服务 或构建应用、同时能够快速和云平台资源进行对接。

- SpringCloud分布式开发五大常用组件

服务发现——Netflix Eureka

客户端负载均衡——Netflix Ribbon

断路器——Netflix Hystrix

服务网关——Netflix Zuul

分布式配置——Spring Cloud Config

Spring Cloud 入门

1、创建provider

- 2、创建consumer
- 3、引入Spring Cloud
- 4、引入Eureka注册中心
- 5、引入Ribbon进行客户端负载均衡
- 6、引入Feign进行声明式HTTP远程调用

Spring Boot与开发热部署

热部署

在开发中我们修改一个Java文件后想看到效果不得不重启应用，这导致大量时间 花费，我们希望不重启应用的情况下，程序可以自动部署（热部署）。有以下四 种情况，如何实现热部署

1、模板引擎

在Spring Boot中开发情况下禁用模板引擎的cache

页面模板改变ctrl+F9可以重新编译当前页面并生效

2、Spring Loaded

Spring官方提供的热部署程序，实现修改类文件的热部署

下载Spring Loaded（项目地址<https://github.com/springprojects/spring-loaded>）

添加运行时参数；

```
-javaagent:C:/springloaded-1.2.5.RELEASE.jar -noverify
```

3、JRebel

收费的一个热部署软件

安装插件使用即可

4、Spring Boot Devtools（推荐）

引入依赖

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

IDEA必须做一些小调整

IntelliJ IDEA和Eclipse不同，Eclipse设置了自动编译之后，修改类它会自动编译，而IDEA在非RUN或DEBUG情况下 才会自动编译（前提是你已经设置了Auto-Compile）。

- 设置自动编译（settings-compiler-make project automatically）
- ctrl+shift+alt+/（maintenance）
- 勾选compiler.automake.allow.when.app.running

Spring Boot与监控管理

监控管理

通过引入spring-boot-starter-actuator，可以使用Spring Boot为我们提供的准生产环境下的应用监控和管理功能。我们可以通过HTTP，JMX，SSH协议来进行操作，自动得到审计、健康及指标信息等

- 步骤

引入spring-boot-starter-actuator

通过http方式访问监控端点

可进行shutdown（POST 提交，此端点默认关闭）

- 监控和管理端点

端点名	描述
actuator	所有Endpoint端点，需加入spring HATEOAS支持
autoconfig	所有自动配置信息
beans	所有Bean的信息
configprops	所有配置属性
dump	线程状态信息
env	当前环境信息
health	应用健康状况
info	当前应用信息
metrics	应用的各项指标
mappings	应用@RequestMapping映射路径
shutdown	关闭当前应用（默认关闭）
trace	追踪信息（最新的http请求）

定制端点信息

- 定制端点一般通过endpoints+端点名+属性名来设置。
- 修改端点id（endpoints.beans.id=mybeans）
- 开启远程应用关闭功能（endpoints.shutdown.enabled=true）
- 关闭端点（endpoints.beans.enabled=false）
- 开启所需端点
 - endpoints.enabled=false
 - endpoints.beans.enabled=true
- 定制端点访问路径

- management.context-path=/manage
- 关闭http端点
 - management.port=-1

Centos7查看开放端口命令及开放端口号

查看已开放的端口

firewall-cmd --list-ports

开放端口（开放后需要要重启防火墙才生效）

firewall-cmd --zone=public --add-port=3338/tcp --permanent

重启防火墙

firewall-cmd --reload

关闭端口（关闭后需要要重启防火墙才生效）

firewall-cmd --zone=public --remove-port=3338/tcp --permanent

开机启动防火墙

systemctl enable firewalld

开启防火墙

systemctl start firewalld

禁止防火墙开机启动

systemctl disable firewalld

停止防火墙

systemctl stop firewalld