SpringBoot1经典版

一、SpringBoot入门

1、SPringBoot简介

- 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

①优点

- 快速创建独立运行的Spring项目以及主流框架集成
- 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，也可以修改默认值
- 无需配置xml，代码自动生成，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

2、微服务

- 微服务：架构风格（服务微化）
    - 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
- 单体应用：ALL IN ONE
- 微服务：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

3、环境准备

- jdk1.8+
- maven 3.x
- IDEA
- SpringBoot 1.5.9.RELEASE

①Maven设置

给maven 的settings.xml配置文件的profiles标签添加

```
<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>
```

②IDEA设置

设置Maven

- 使用系统安装的Maven并且重写Setting文件和本地仓库

4、SpringBoot HelloWorld

功能

浏览器发送hello请求，服务器接受请求并处理，响应HelloWorld字符串

①创建一个Maven工程（jar）

②导入SpringBoot的相关依赖

```
<parent>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-parent</artifactId>
   <version>1.5.9.RELEASE</version>
</parent>
<dependencies>
   <dependency>
       <groupId>org.springframework.boot</groupId>
       <artifactId>spring-boot-starter-web</artifactId>
   </dependency>
</dependencies>
```

③编写一个主程序：启动SpringBoot应用

```
/*
* @SpringBootApplication 来标注一个主程序类，说明这是一个SpringBoot应用
* */
@SpringBootApplication
public class HelloWorldMainApplication {
    public static void main(String[] args) {
//        Spring应用启动
        SpringApplication.run(HelloWorldMainApplication.class);
    }
}
```

④编写相关的Controller、Service

```
//可以直接在类中使用@RestController
//@RestController中包含@Controller和@ResponseBody
@Controller
public class HelloController {
    @ResponseBody
    @RequestMapping("/hello")
    public String hello(){
        return "Hello World";
    }
}
```

⑤运行主程序测试

运行main方法

访问http://localhost:8080/hello

⑥简化部署

```
<!-- 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>

<!--在当前文件目录下使用 java -jar jar包名 进行运行部署-->
```

将这个应用打成jar包，直接使用java -jar的命令进行执行；

5、HelloWorld探究

①pom文件

父项目

```
<parent>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-parent</artifactId>
   <version>1.5.9.RELEASE</version>
</parent>

这个父项目的父项目是
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-dependencies</artifactId>
    <version>1.5.9.RELEASE</version>
    <relativePath>../../spring-boot-dependencies</relativePath>
</parent>
这个是真正管理SpringBoot应用里面的所有依赖版本
```

SpringBoot版本的仲裁中心

以后我们导入依赖默认是不需要写版本（**没有在dependencies里面管理的依赖需要写版本号**）

启动器

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

spring-boot-starter-web

- spring-boot-starter：spring-boot场景启动器，帮我们导入了web模块正常运行所依赖的组件

SpringBoot将所有的功能都抽取出来，做成一个个的starters（启动器），只需要在项目里面引入这些starters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

②主程序类，主入口类

```
/*
* @SpringBootApplication 来标注一个主程序类，说明这是一个SpringBoot应用
* */
@SpringBootApplication
public class HelloWorldMainApplication {
    public static void main(String[] args) {
//        Spring应用启动
        SpringApplication.run(HelloWorldMainApplication.class);
    }
}
```

- **@SpringBootApplication**
    - SpringBoot应用标注在某个类上说明这个类是SpringBoot的主配置类，SpringBoot就应该运行这个类的main方法来启动SpringBoot应用
- **@SpringBootConfiguration**
    - SpringBoot的配置类，表示这是一个SpringBoot的配置类
- **@Configuration**
    - 配置类上来标注这个注解
        - 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
        - 以前我们需要配置的东西，SpringBoot帮我们自动配置
- **@AutoConfigurationPackage**

    - 自动配置包
- **@Import(AutoConfigurationPackages.Registrar.class)**
    - Spring的底层注解@Import，给容器中导入一个组件，导入的组件由AutoConfigurationPackages.Registrar.class；
    - 将**主配置类（@SpringBootApplication标注的类）**的所在包以及下面的所有子包里面包含的所有组件扫描到Spring容器
- **@Import(EnableAutoConfigurationImportSelector.class)**
    - EnableAutoConfigurationImportSelector：导入哪些组件的选择器
    - 将所有需要导入的组件以及全类名的方式返回，这些组件将会被添加到容器中
    - 会给容器中导入非常多的自动配置类（xxxAutoConfiguration）就是给容器中导入这个场景需要的所有组件，并配置好这些组件
    - 有了自动配置类，免去了我们手动编写配置注入功能组件等的工作
    - 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，帮我们进行自动配置工作；以前我们需要自己配置的东西，自动配置类都会帮我们配置完成
- J2EE的整体整合解决方案和自动配置都在spring-boot-autoconfigure-1.5.9.RELEASE.jar

6、使用Spring Initializer快速创建SpringBoot项目

①IDEA：使用Spring Initializer快速创建项目

1. IDE都支持使用Spring的项目创建向导快速创建一个Spring Boot项目；
2. 选择我们需要的模块；向导会**联网创建Spring Boot项目；**
3. 默认生成的Spring Boot项目；
    1. 主程序已经生成好了，我们只需要我们自己的逻辑
    2. resources文件夹中目录结构
        1. static：保存所有的静态资源； js css images；
        2. 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）；
        3. **application.properties：Spring Boot应用的配置文件；可以修改一些默认设置；**
4. **创建步骤**
    1. 创建项目面板选择Spring Initializer

2. 输入包名，模块ID等信息，也可以指定什么样的工程（如maven工程）
3. 选择模块——根据需求选择（如web功能）
4. 选择工程名和项目位置
5. 联网自动创建

②STS使用Spring Initializer快速创建项目

二、配置文件

1、配置文件

SpringBoot使用一个全局的配置文件，**配置文件名是固定的**：

- *application.properties
- *application.yml

配置文件的作用：**修改SpringBoot自动配置的默认值**；SpringBoot在底层都给我们自动配置好

YAML（YAML Ain't Markup Language）

YAML A Markup Language：是一个标记语言

YAML isn't Markup Language：不是一个标记语言；

标记语言：

以前的配置文件；大多都使用的是 xxxx.xml文件；

YAML：**以数据为中心**，比json、xml等更适合做配置文件；

YAML：配置例子

```yaml
server:
    port: 8081
```

xml

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

2、YAML语法

①基本语法

key:(空格)value:表示一对键值对**（空格必须有）**

以空格的缩进来控制层级关系：只要是左对齐的一列数据，都是同一个层级

```
server:
 port: 8081
 path: /hello
```

属性和值也是**大小写敏感**

②值的写法【k——键 v——值】

1. **字面量：普通的值（数字，字符串，布尔值）**
    1. k、v：字面直接来写
    2. 字符串默认不添加**单引号或者双引号**
    3. ""：**双引号，不会转义字符串里面的特殊字符**，特殊字符会作为本身表示的意思
        1. name: "zhangsan \n lisi" 输出 zhansan （换行） lisi
    4. ' '：**单引号，会转义特殊字符**，特殊字符只是一个普通的字符串数据
        1. name: ‘zhangsan \n lisi’：输出；zhangsan \n lisi
2. **对象、Map（键值对）**
    1. k: v 在下一行来写对象的属性和值的关系，注意缩进
        1. 对象还是k:v 的方式

```
friends:
    lastName: zhangsan
    age: 20
```

        2. 行内写法

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

3. **数组（List，Set）**
    1. 用 -值 表示数组中的一个元素

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

2. 行内写法

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

3、配置文件值注入（获取配置文件值）

配置文件

```
person:
    lastName: hello
    age: 18
    boss: false
    birth: 2017/12/12
    maps: {k1:v1,k2:v2}
    lists:
        - lisi
        - zhaoliu
    dog:
        name: 小狗
        age: 12
```

javaBean（使用@ConfigurationProperties）

```
/**
* 将配置文件中配置的每一个属性的值，映射到这个组件中
* @ConfigurationProperties：告诉SpringBoot将本类中的所有属性和配置文件中相关的配置进行绑定；
* prefix = "person"：配置文件中哪个下面的所有属性进行一一映射
*
* 只有这个组件是容器中的组件，才能容器提供的@ConfigurationProperties功能；
*
*/
@Component
@ConfigurationProperties(prefix = "person")
public class Person {
   private String lastName;
   private Integer age;
   private Boolean boss;
   private Date birth;

   private Map<String,Object> maps;
   private List<Object> lists;
   private Dog dog;
    //构造器
    //set get方法
}
```

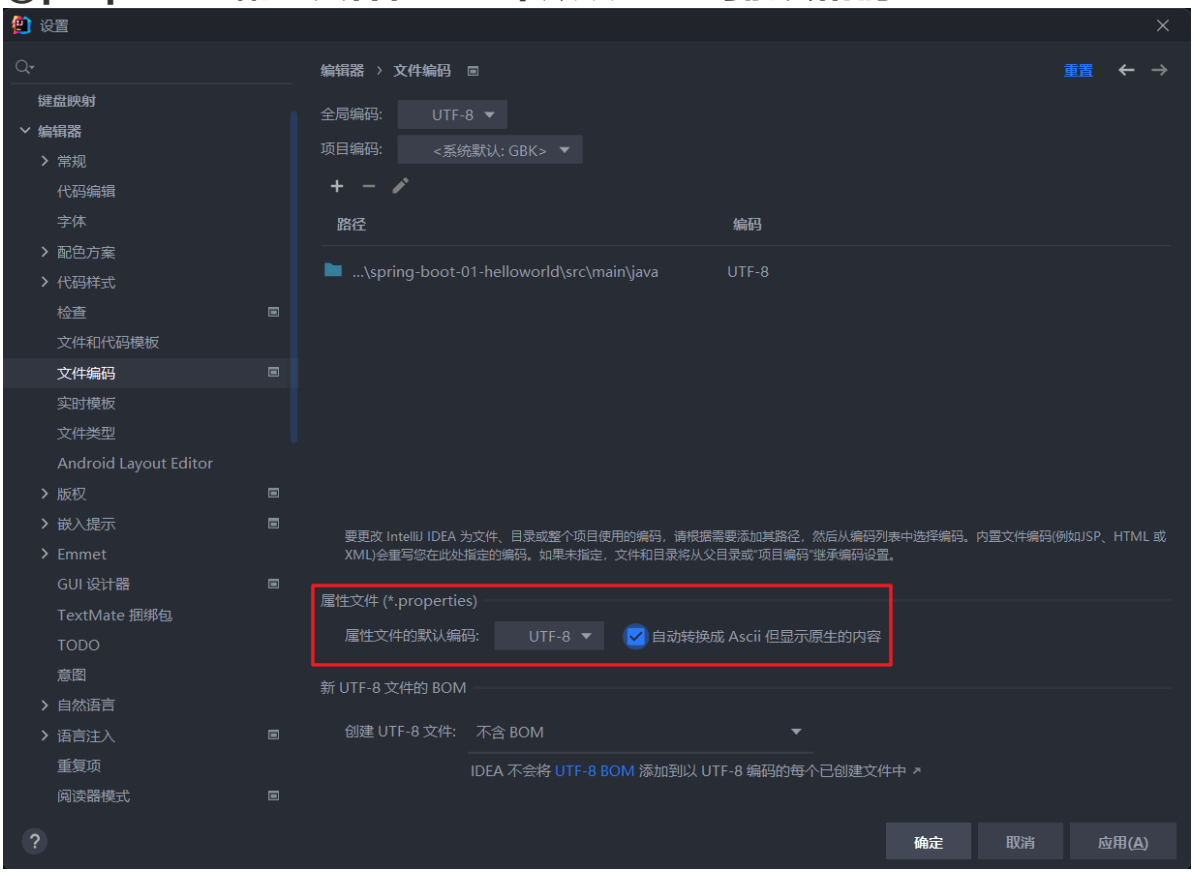
javaBean（使用@Value）

```
@Component
public class Person{
    //使用${配置文件key值} 也就是${SpEL}
    @Value("${person.last-name}")
    private String lastName;
}
```

我们可以导入配置文件处理器，以后编辑配置就有提示

```
<!--导入配置文件处理器，配置文件进行绑定就会有提示-->
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-configuration-processor</artifactId>
   <optional>true</optional>
</dependency>
```

①properties配置文件在idea中默认utf-8可能会乱码



②@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
SpEL	不支持	支持
JSR303数据校验	支持	不支持
复杂类型封装	支持	不支持

配置文件yml还是properties他们都能获取到值

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

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

③配置文件注入值数据校验

```
@Component
@ConfigurationProperties(prefix = "person")
@Validated
public class Person {
    /**
     * <bean class="Person">
     * <property name="lastName" value="字面量/${key}从环境变量、配置文件中获取值/#{SpEL}"></property>
     * <bean/>
*/
    //lastName必须是邮箱格式
    @Email
    //@Value("${person.last-name}")
    private String lastName;
    //@Value("#{11*2}")
    private Integer age;
    //@Value("true")
    private Boolean boss;
    private Date birth;
    private Map<String,Object> maps;
    private List<Object> lists;
    private Dog dog;
}
```

④@PropertySource & @ImportResource & @Bean

@PropertySource：**加载指定的配置文件**

```
@PropertySource(value = {"classpath:person.properties"})
```

@ImportResource：**导入Spring的配置文件**，让配置文件里面的内容生效

SpringBoot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**

想让Spring的配置文件生效，加载进来；@ImportResource标注在一个配置类上

```
@ImportResource(locations = {"classpath:beans.xml"})
导入Spring的配置文件让其生效
```

不写Spring的配置文件【所以不需要写@ImportResource】，SpringBoot推荐给容器中添加组件的方式：使用**全注解的方式**

1. **配置类@Configuration ——>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；就是来替代之前的Spring配置文件
* 在配置文件中用<bean><bean/>标签添加组件
*/
@Configuration
public class MyAppConfig {
    //将方法的返回值添加到容器中；容器中这个组件默认的id就是方法名
    @Bean
    public HelloService helloService02(){
        System.out.println("配置类@Bean给容器中添加组件了...");
        return new HelloService();
    }
}
```

4、配置文件占位符 ${v}【以Properties为例】

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，如果没有可以是用：指定默认值

```
person.last-name=张三${random.uuid}
person.age=${random.int}
person.birth=2017/12/15
person.boss=false
person.maps.k1=v1
person.maps.k2=14
person.lists=a,b,c
#如果last-name存在就要lastname，否则使用默认值hello
person.dog.name=${person.last-name:hello}_dog
person.dog.age=15
```

5、Profile

①多Profile文件

在主配置文件编写的时候，文件名可以是 **application-{profile}.properties/yml**

默认使用application.properties的配置

②yml支持多文档快方式

```
server:
    port: 8081
spring:
    profiles:
        active: prod
---
server:
    port: 8083
spring:
    profiles: dev
---
server:
    port: 8084
spring:
    profiles: prod #指定属于哪个环境
```

③激活指定profile

不同的方式

1. 在配置文件中指定 **spring.profiles.active=dev**
2. 命令行

1. java -jar spring-boot-02-config-0.0.1-SNAPSHOT.jar --spring.profiles.active=dev
2. 可以直接在测试的时候，配置传入命令行参数

3. 虚拟机参数
    1. -Dspring.profiles.active=dev

6、配置文件加载位置

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

【-file:当前文件的根目录，也就是项目目录】

–file:./config/

–file:./

【-classpath:为类路径，也就是resources下】

–classpath:/config/

–classpath:/

优先级由高到底，高优先级的配置会覆盖低优先级的配置

SpringBoot会从这四个位置全部加载主配置文件；**互补配置**

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

项目**打包之后**可以使用**命令行参数的形式**，启动项目的时候来指定配置文件的新位置，指定配置文件和默认加载的这些配置文件共同起作用来形成**互补配置**

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

7、外部配置加载顺序

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

1. 命令行参数
    1. 所有的配置都可以在命令行上进行指定
       java -jar spring-boot-02-config-02-0.0.1-SNAPSHOT.jar --server.port=8087 --server.context-path=/abc
       多个配置用空格分开； --配置项=值

2. 来自java:comp/env的JNDI属性
3. Java系统属性（System.getProperties()）
4. 操作系统环境变量
5. RandomValuePropertySource配置的random.*属性值

由jar包外向jar包内进行寻找；

优先加载带profile

1. jar包外部的application-{profile}.properties或application.yml(带spring.profile)配置文件
2. 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

1. jar包外部的application.properties或application.yml(不带spring.profile)配置文件
2. jar包内部的application.properties或application.yml(不带spring.profile)配置文件

1. @Configuration注解类上的@PropertySource
2. 通过SpringApplication.setDefaultProperties指定的默认属性

所有支持的配置加载来源详见**官方文档**

8、自动配置原理

配置文件属性参照 https://docs.spring.io/spring-boot/docs/1.5.9.RELEASE/reference/htmlsingle/#appendix

①自动配置原理

- SpringBoot启动的时候加载主配置类，开启了自动配置功能 **@EnableAutoConfiguration**
- @EnableAutoConfigurationImportSelector作用
    - 利用EnableAutoConfigurationImportSelector给容器中导入一些组件?
    - 可以查看selectImports()方法的内容

    - *List configurations = getCandidateConfigurations(annotationMetadata, attributes);*获取候选的配置
    - 将 类路径下 **/META-INF/spring.factories** 里面配置的所有EnableAutoConfiguration的值加入到容器中
    - 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，用他们来自动配置
- 每一个自动配置类进行自动配置功能
- 以HttpEncodingAutoConfiguration（Http编码自动配置）为例解释自动配置原理

```
//表示这是一个配置类，以前编写的配置文件一样，也可以给容器中添加组件
@Configuration
//启动指定类的ConfigurationProperties功能；将配置文件中对应的值和HttpEncodingProperties绑定起来；并把HttpEncodingProperties加入到ioc容器中
@EnableConfigurationProperties(HttpEncodingProperties.class)
//Spring底层@Conditional注解（Spring注解版），根据不同的条件，如果满足指定的条件，整个配置类里面的配置就会生效；  判断当前应用是否是web应用，如果是，当前配置类生效
@ConditionalOnWebApplication
//判断当前项目有没有这个类CharacterEncodingFilter；SpringMVC中进行乱码解决的过滤器；
@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
@ConditionalOnProperty(prefix = "spring.http.encoding", value = "enabled",matchIfMissing = true)
   //即使我们配置文件中不配置pring.http.encoding.enabled=true，也是默认生效的；

   public class HttpEncodingAutoConfiguration {
       //他已经和SpringBoot的配置文件映射了
       private final HttpEncodingProperties properties;
       //只有一个有参构造器的情况下，参数的值就会从容器中拿
       public HttpEncodingAutoConfiguration(HttpEncodingProperties properties) {
           this.properties = properties;
       }

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

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

```
            CharacterEncodingFilter filter = new
OrderedCharacterEncodingFilter();

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

filter.setForceRequestEncoding(this.properties.shouldForce(Typ
e.REQUEST));

filter.setForceResponseEncoding(this.properties.shouldForce(Ty
pe.RESPONSE));
            return filter;
        }
```

一旦这个配置类生效，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xxxProperties类中封装者；配置文件能配置扫描就可以参照某个功能对应的这个属性类

```
@ConfigurationProperties(prefix = "spring.http.encoding") //从
配置文件中获取指定的值和bean的属性进行绑定
   public class HttpEncodingProperties {
      public static final Charset DEFAULT_CHARSET =
Charset.forName("UTF-8");
   }
```

总结

1. **SpringBoot启动会加载大量的自动配置类**
2. **看需要的功能有没有SpringBoot默认写好的自动配置类**
3. **再来看这个自动配置类中到底配置了哪些组件（只要要用的组件有，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，就可以在配置文件中指定这些属性的值**

xxxxAutoConfiguration：自动配置类

xxxxProperties：封装配置文件中相关属性

②细节

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

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

@Conditional扩展注解	作用（判断是否满足当前指定条件）
@ConditionalOnJava	系统的java版本是否符合要求
@ConditionalOnBean	容器中存在指定Bean
@ConditionalOnMissingBean	容器中不存在指定Bean
@ConditionalOnExpression	满足SpEL表达式指定
@ConditionalOnClass	系统中有指定的类
@ConditionalOnMissingClass	系统中没有指定的类
@ConditionalOnSingleCandidate	容器中只有一个指定的Bean，或者这个Bean是首选Bean
@ConditionalOnProperty	系统中指定的属性是否有指定的值
@ConditionalOnResource	类路径下是否存在指定资源文件
@ConditionalOnWebApplication	当前是web环境
@ConditionalOnNotWebApplication	当前不是web环境
@ConditionalOnJndi	JNDI存在指定项

自动配置类必须在一定的情况下才能生效

我们可以通过启用 debug=true 属性；来让控制台打印自动配置报告，这样我们就可以很方便的知道哪些自动配置类生效

三、日志

1、日志框架

市面上的日子框架

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

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 SLF4j和logback；

2、SLF4j使用

①如何在系统中使用SLF4j https://www.slf4j.org/

以后开发的时候，日志记录方法的调用，不应该来直接调用日志的实现类，而是调用日志抽象层里面的方法；给系统里面导入slf4j的jar和logback的实现jar

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

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自己本身的配置文件；

②遗留问题

a（slf4j+logback）：Spring（commons-logging）、Hibernate（jboss-logging）、MyBatis、xxxx统一日志记录，即使是别的框架和我一起统一使用slf4j进行输出？

如何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、我们导入slf4j其他的实现

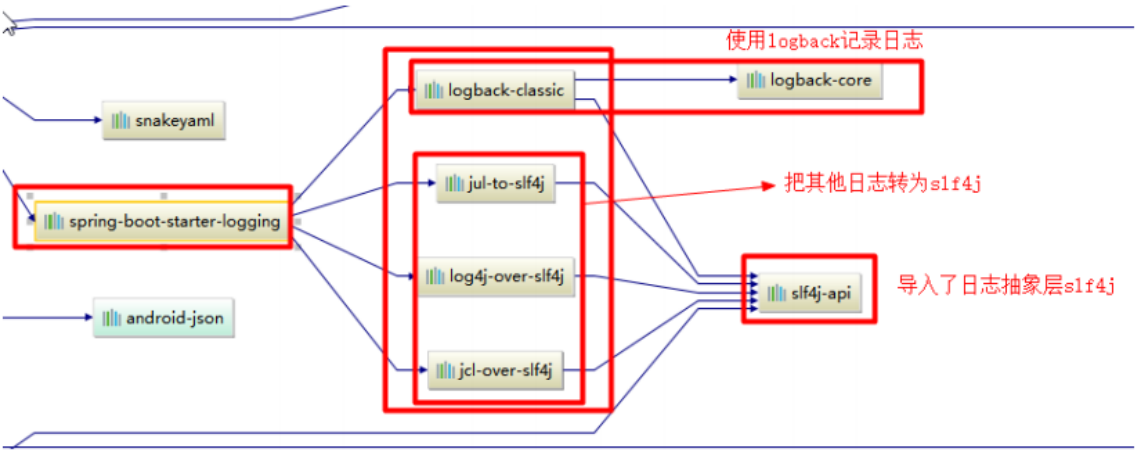
3、SpringBoot日志关系

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

SpringBoot使用它来做日志功能

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

底层依赖关系



总结：

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长了 slf4j
3. 中间替换包

```
@SuppressWarnings("rawtypes")
public abstract class LogFactory {
    static String UNSUPPORTED_OPERATION_IN_JCL_OVER_SLF4J =

 "http://www.slf4j.org/codes.html#unsupported_operation_in_jcl
_ove_slf4j";
    static LogFactory logFactory = new SLF4JLogFactory();
}
```

4. 如果要引入其他框架，一定要把这个框架的默认日志依赖**排除掉**

   Spring框架用的是commons-logging

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-core</artifactId>
    <exclusions>
        <exclusion>
            <groupId>commons-logging</groupId>
            <artifactId>commons-logging</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

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，只需要把这个框架依赖的日志框架排除掉即可

4、日志使用

①默认配置

SpringBoot默认配置好了日志

```
//记录器
Logger logger = LoggerFactory.getLogger(getClass());
@Test
public void contextLoads() {
    //System.out.println();
    //日志的级别；
    //由低到高 trace<debug<info<warn<error
    //可以调整输出的日志级别；日志就只会在这个级别以以后的高级别生效
    logger.trace("这是trace日志...");
    logger.debug("这是debug日志...");
    //SpringBoot默认给我们使用的是info级别的，没有指定级别的就用SpringBoot默认规定的别；root级别
    logger.info("这是info日志...");
    logger.warn("这是warn日志...");
    logger.error("这是error日志...");
}
```

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

SpringBoot修改日志的默认配置

```
logging.level.com.atguigu=trace

#logging.path=
# 不指定路径在当前项目下生成springboot.log日志
# 可以指定完整的路径；
#logging.file=G:/springboot.log
# 在当前磁盘的根路径下创建spring文件夹和里面的log文件夹；使用 spring.log 作为默认文件

logging.path=/spring/log
# 在控制台输出的日志的格式
logging.pattern.console=%d{yyyy-MM-dd} [%thread] %-5level %logger{50} - %msg%n
# 指定文件中日志输出的格式
logging.pattern.file=%d{yyyy-MM-dd} === [%thread] === %-5level === %logger{50} ==== %msg%n
```

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

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Profile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>
```

如：

```
<appender name="stdout" class="ch.qos.logback.core.ConsoleAppender">
    <!--
    日志输出格式：
    %d表示日期时间，
    %thread表示线程名，
    %-5level：级别从左显示5个字符宽度
    %logger{50} 表示logger名字最长50个字符，否则按照句点分割。
    %msg：日志消息，
    %n是换行符
    -->
    <layout class="ch.qos.logback.classic.PatternLayout">
        <springProfile name="dev">
            <pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} ----> [%thread] ---->
%-5leve %logger{50} - %msg%n</pattern>
        </springProfile>
        <springProfile name="!dev">
            <pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} ==== [%thread] ====
%-5leve %logger{50} - %msg%n</pattern>
        </springProfile>
    </layout>
</appender
```

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

no applicable action for [springProfile]

5、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

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <exclusions>
        <exclusion>
            <artifactId>logback-classic</artifactId>
            <groupId>ch.qos.logback</groupId>
        </exclusion>
        <exclusion>
```

```
            <artifactId>log4j-over-slf4j</artifactId>
            <groupId>org.slf4j</groupId>
        </exclusion>
    </exclusions>
</dependency>
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-log4j12</artifactId>
</dependency
```

切换为log4j2

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <exclusions>
        <exclusion>
            <artifactId>spring-boot-starter-logging</artifactId>
            <groupId>org.springframework.boot</groupId>
        </exclusion>
    </exclusions>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-log4j2</artifactId>
</dependency
```

四、Web开发

1、简介

使用SpringBoot

1. 创建SpringBoot应用，选择我们需要的模块
2. SpringBoot已经默认将这些场景配置好，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

```
xxxxAutoConfiguration：帮我们给容器中自动配置组件；
xxxxProperties:配置类来封装配置文件的内容；
```

2、SpringBoot对静态资源的映射规则

```
@ConfigurationProperties(prefix =
"spring.resources",ignoreUnknownFields = false)
public class ResourceProperties implements ResourceLoaderAware{
    //可以设置和静态资源有关的参数，缓存时间等
    WebMvcAutoConfiguration;
    @Override
    public void addResourceHandlers(ResourceHandlerRegistry registry){
        if (!this.resourceProperties.isAddMappings()){
            logger.debug("Default resource handling disabled");
            return;
        }
        Integer cachePeriod =
this.resourceProperties.getCachePeriod();
        if (!registry.hasMappingForPattern("/webjars/**")){
            customizeResourceHandlerRegistration(
                registry/addResourceHandler("/webjars/**")
                .addResourceLocations("classpath:/META-
INF/resources/webjars")
                .setCachePeriod());
        }
    }

    //配置欢迎页面映射
    @Bean
    public WelcomePageHandlerMapping
welcomePageHandlerMapping(ResourceProperties resourceProperties){
        return new
WelcomePageHandlerMapping(resourceProperties.getWelcomePage(),this.mvc
Properties.getStaticPathPattern());
    }

    //配置喜欢的图标
    @Configuration
    @ConditionalOnProperty(value = "spring.mvc.favicon.enabled",
matchIfMissing = true)
    public static class FaviconConfiguration{
        private final ResourceProperties resourceProperties;
        public FaviconConfiguration(ResourceProperties
resourceProperties){
            this.resourceProperties = resourceProperties;
        }

        @Bean
        public SimpleUrlHandlerMapping faviconHandlerMapping(){
            SimpleUrlHandlerMapping mapping = new
SimpleUrlHandlerMapping();
            mapping.setOrder(Ordered.HIGHEST_PRECEDENCE + 1);
```

```
            //所有的 **/favicon.ico

 mapping.setUrlMap(Collections.singletonMap("**/favicon.ico",
                                faviconRequestHandler()));
            return mapping;
        }

        @Bean
        public ResourceHttpRequestHandler faviconRequestHandler(){
            ResourceHttpRequestHandler requestHandler = new
ResourceHttpRequestHandler();

requestHandler.setLocations(this.resourceProperties.getFaviconLocation
s());
            return requestHandler;
        }
    }
}
```

1. 所有/webjars/**，都去classpath:/META-INF/resources/webjars/找资源
    1. webjars：以**jar包**的方式引入静态资源 https://www.webjars.org/
    2. ```
       <!--引入jquery-webjar 在访问的时候只需要写webjars下面资源的名称即可-->
       <dependency>
           <groupId>org.webjars</groupId>
           <artifactId>jquery</artifactId>
           <version>3.3.1</version>
       </dependency>
       ```
2. "/** 访问当前项目的任何资源，都去（静态资源的文件夹）找资源"
    1. ```
       "classpath:/META-INF/resources/",
       "classpath:/resources/",
       "classpath:/static/",
       "classpath:/public/"
       "/"：当前项目的根路径
       ```
3. 欢迎页；静态资源文件夹下的所有index.html页面；被"/**"映射
4. 所有的 **/favicon.ico 都是在静态资源文件下找

3、模板引擎

JSP、Velocity、Freemarker、Thymeleaf

SpringBoot推荐的**Thymeleaf**；语法更简单，功能更强大

①引入thymeleaf

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-thymeleaf</artifactId>
    <version>2.1.6</version>
</dependency>

<!--切换thymeleaf版本-->
<properties>
    <thymeleaf.version>3.0.9.RELEASE</thymeleaf.version>
    <!-- 布局功能的支持程序 thymeleaf3主程序 layout2以上版本 -->
    <!-- thymeleaf2 layout1-->
    <thymeleaf-layout-dialect.version>2.2.2</thymeleaf-layout-
diaect.version>
</properties
```

②Thymeleaf使用

```
@ConfigurationProperties(prefix = "spring.thymeleaf")
public class ThymeleafProperties {
    private static final Charset DEFAULT_ENCODING =
Charset.forName("UTF-8");
    private static final MimeType DEFAULT_CONTENT_TYPE =
MimeType.valueOf("text/html");
    public static final String DEFAULT_PREFIX =
"classpath:/templates/";
    public static final String DEFAULT_SUFFIX = ".html";
}
```

只要我们把HTML页面放在classpath:/templates/，thymeleaf就会自动渲染

使用：

1. 导入thymeleaf的名称空间

```
<html lang="en" xmlns:th="http://www.thymeleaf.org"/>
```

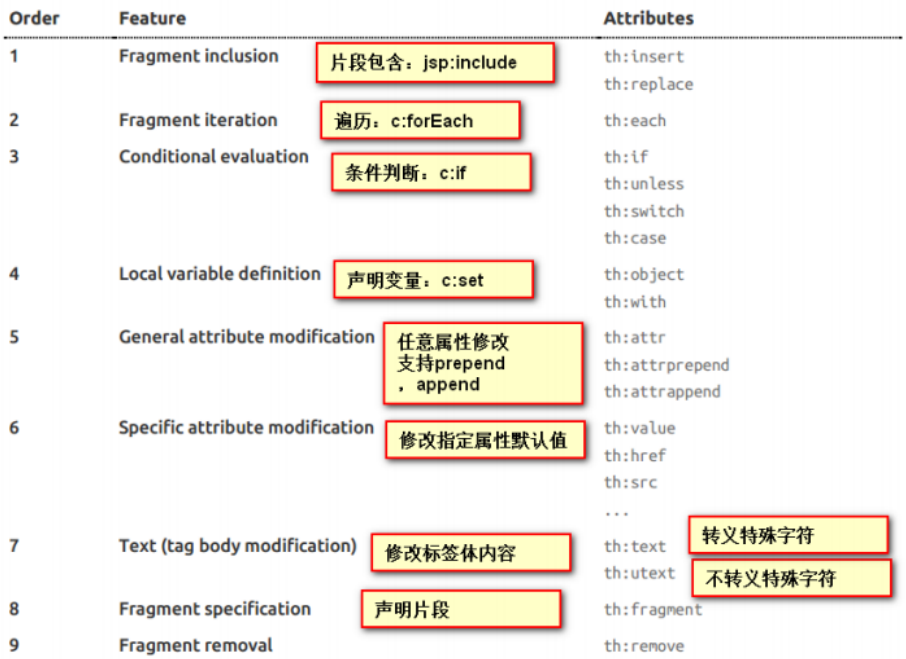
2. 使用thymeleaf语法

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
    <head>
        <meta charset="UTF-8">
        <title>Title</title>
    </head>
    <body>
        <h1>成功！</h1>
        <!--th:text 将div里面的文本内容设置为 -->
        <div th:text="${hello}">这是显示欢迎信息</div>
    </body>
</html>
```

③语法规则

1. th:text　改变当前元素里面的文本内容

    1. th:任意thml属性；来替换原生属性的值



2. 表达式

```
Simple expressions: （表达式语法）
    Variable Expressions: ${...}：获取变量值；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}

        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
            #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.
            #bools : 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：和${}
在功能上是一样；
            补充：配合 th:object="${session.user}：
            <div th:object="${session.user}">
                <p>Name: <span th:text="*
{firstName}">Sebastian</span>.</p>
                <p>Surname: <span th:text="*
{lastName}">Pepper</span>.</p>
                <p>Nationality: <span th:text="*
{nationality}">Saturn</span>.</p>
            </div>
        Message Expressions: #{...}：获取国际化内容
        Link URL Expressions: @{...}：定义URL
@{/order/process(execId=${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（三元运算符）
    If-then: (if) ? (then)
    If-then-else: (if) ? (then) : (else)
    Default: (value) ?: (defaultvalue)
    Special tokens:
    No-Operation: _
```

4、SpringMVC自动配置

https://docs.spring.io/spring-boot/docs/1.5.10.RELEASE/reference/htmlsingle/#boot-features-developing-web-applications

①SpringMVC auto-configuration

SpringBoot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：我们可以给自己容器中添加一个视图解析器；自动的将其组合起来
- 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
//在文件中配置日期格式化的规则
@ConditionalOnProperty(prefix = "spring.mvc",name = "data-
format")
public Formatter<Date> dateFormatter(){
    //日期格式化组件
    return new
DateFormatter(this.mvcProperties.getDateFormat());
}
```

自己添加的格式化转化器，我们只需要放在容器中即可

- Support for HttpMessageConverters （see below）
    - HttpMessageConverter：SpringMVC用来转换Http请求和响应的；User--Json
    - HttpMessageConverter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

②扩展SpringMVC

```
<!--使用配置文件的方式-->
<mvc:view-controller path="/hello" view-name="success" />
<mvc:interceptors>
    <mvc:interceptor>
        <mvc:mapping path="/hello"/>
    </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

```
//使用WebMvcConfigurerAdapter可以扩展SpringMVC的功能
@Configuration
public class MyMvcConfig extends WebMvcConfigurerAdapter{
    @Override
    public void addViewControllers(ViewControllerRegistry registry){
        //super.addViewControllers(registry);
        //浏览器发送 /atguigu 请求来到 success
        registry.addViewController("/atguigu").setViewName("success");
    }
}
```

原理：

- WebMvcAutoConfiguration是SpringMVC的自动配置类
- 在做其他自动配置时会导入；**@Import(EnableWebMvcConfiguration.class)**

```
@Configuration
public static class EnableWebMvcConfiguration extends
DelegatingWebConfiguration{
    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);
            // }
        }
    }
}
```

- 容器中所有的WebMvcConfigurer都会一起起作用
- 我们的配置类也会被调用
    - **效果：SpringMVC的自动配置类和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@EnbaleWebMvc即可

```
//使用@WebMvcConfigurerAdapter可以扩展SpringMVC的功能
@EnableWebMvc
@Configuration
public class MyMvcConfig extends WebMvcConfigurerAdapter{
    @Override
    public void addViewControllers(ViewControllerRegistery registry){
        //supper.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？

- @EnableWebMvc的核心

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

- ```
  @Configuration
  public class DelegatingWebMvcConfiguration extends
  WebMvcConfigurationSupport {}
  ```
  ```

- ```
  @Configuration
  @ConditionalOnWebApplication
  @ConditionalOnClass({ Servlet.class, DispatcherServlet.class,
                        WebMvcConfigurerAdapter.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(Ordered.HIGHEST_PRECEDENCE + 10)
  @AutoConfigureAfter({
  DispatcherServletAutoConfiguration.class,
                        ValidationAutoConfiguration.class })
  public class WebMvcAutoConfiguration {}
  ```

- @EnableWebMvc将WebMvcConfigurationSupport组件导入进来
- 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：

- 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；
- 在SpringBoot中会有非常多的xxxConfigurer帮助我们进行**扩展配置**
- 在SpringBoot中会有很多的xxxCustomizer帮助我们进行**定制配置**

6、RestfulCRUD

①默认访问首页

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

    //所有的WebMvcConfigurerAdapter组件都会一起起作用，就可以不用再控制器中写
```
```

```
    @Bean //将组件注册在容器
    public WebMvcConfigurerAdapter webMvcConfigurerAdapter(){
        WebMvcConfigurerAdapter adapter = new
WebMvcConfigurerAdapter() {
            @Override
            public void addViewControllers(ViewControllerRegistry
registry) {
                registry.addViewController("/").setViewName("login");

 registry.addViewController("/index.html").setViewName("login");
            }
        };
        return adapter;
    }
}
```

或者在HelloController中

```
@RequestMapping("/")
public String index(){
    return "index";
}
```

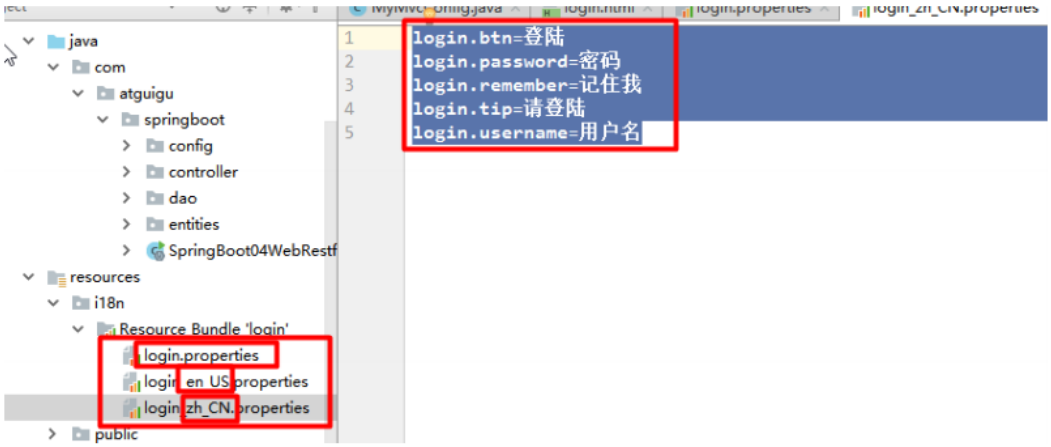
②国际化

SpringMVC实现步骤

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

SpringBoot实现步骤

- 编写国家化配置文件，抽取页面需要显示的国际化消息

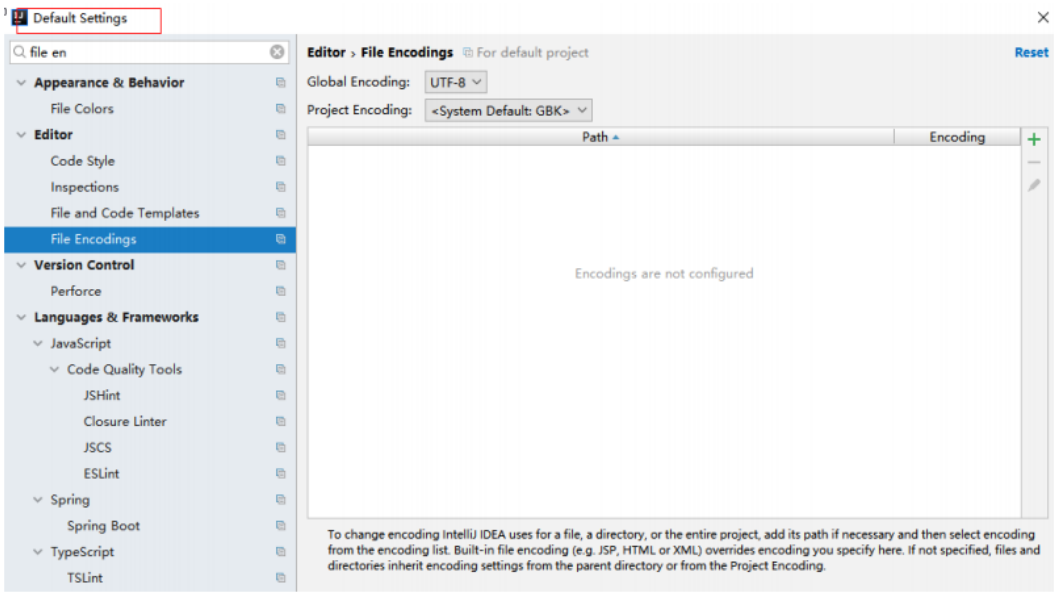


注意文件命名！

- SpringBoot自动配置好了国际化资源文件的组件

```
@ConfigurationProperties(prefix = "spring.messages")
public class MessageSourceAutoConfiguration {
    /**
    * Comma-separated list of basenames (essentially a fully-
qualified classpath
    * location), each following the ResourceBundle convention
with relaxed support for
    * slash based locations. If it doesn't contain a package
qualifier (such as
    * "org.mypackage"), it will be resolved from the classpath
root.
    */

    private String basename = "messages";
    //我们的配置文件可以直接放在类路径下叫messages.properties;
    @Bean
    public MessageSource messageSource() {
        ResourceBundleMessageSource messageSource = new
ResourceBundleMessageSource();
        if (StringUtils.hasText(this.basename)) {
            //设置国际化资源文件的基础名（去掉语言国家代码的）

 messageSource.setBasenames(StringUtils.commaDelimitedListToSt
ringArray(

 StringUtils.trimAllWhitespace(this.basename)));
        }
        if (this.encoding != null) {

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

 messageSource.setFallbackToSystemLocale(this.fallbackToSystem
Locale);
        messageSource.setCacheSeconds(this.cacheSeconds);

 messageSource.setAlwaysUseMessageFormat(this.alwaysUseMessage
Format);
        return messageSource;
    }
```

- 去页面获取国际化的值，设置文件编码为UTF-8并且自动转为ASCII码




```
<!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/4.0.0/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">
            <img class="mb-4" th:src="@{/asserts/img/bootstrap-
solid.svg}"
                 src="asserts/img/bootstrap-solid.svg" alt=""
width="72" height="72">
            <h1 class="h3 mb-3 font-weight-normal" th:text="#
{login.tip}">Please sign
                in</h1>
            <label class="sr-only" th:text="#
{login.username}">Username</label>
            <input type="text" class="form-control"
placeholder="Username"
```

```
                    th:placeholder="#{login.username}" required=""
autofocus="">
            <label class="sr-only" th:text="#
{login.password}">Password</label>
            <input type="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">中文</a>
            <a class="btn btn-sm">English</a>
        </form>
    </body>
</html
```

效果：根据浏览器语言设置的信息切换了国际化

原理：

国家化Locale（区域信息对象）；LocalResolver（获取区域信息对象）

```
@Bean
@ConditionalOnMissingBean
@ConditionalOnProperty(prefix = "spring.mvc", name = "locale")
public LocaleResolver localeResolver() {
    if (this.mvcProperties.getLocaleResolver() ==
WebMvcProperties.LocaleResolver.FIXED) {
        return new
FixedLocaleResolver(this.mvcProperties.getLocale());
    }
    AcceptHeaderLocaleResolver localeResolver = new
AcceptHeaderLocaleResolver();
    localeResolver.setDefaultLocale(this.mvcProperties.getLocale());
    return localeResolver;
}
```

- 点击链接切换国际化

```
/**
* 可以在连接上携带区域信息
```

```
*/
public class MyLocaleResolver implements LocaleResolver{
    @Override
    public Locale resolveLocale(HttpServletRequest request){
        String l = request.getParameter("l");
        Locale locale = Locale.getDefault();
        if(!StringUtils.isEmpty(l)){
            String[] split = l.split("_");
            locale = new Locale(split[0],split[1]);
        }
        return locale;
    }

    @Override
    public void setLocale(HttpServletRequest request,
HttpServletResponse response, Locale locale) {}

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

③登录

开发期间模板引擎页面修改以后，要实时生效

- 禁用模板引擎的缓存

```
spring.thymeleaf.cache=false
```

- 页面修改完成以后ctrl+F9，重新编译；

登录错误信息的显示

```
<p style="color:red" th:text="${msg}" th:if="${not
#strings.isEmpty(msg)}"></p>
```

④拦截器进行登录检查

拦截器

```
/**
* 登陆检查，
```

```
*/
public class LoginHandlerInterceptor implements HandlerInterceptor {
    //目标方法执行之前
    @Override
    public boolean preHandle(HttpServletRequest request,
HttServletResponse response,
                             Object handler) throws Exception {
        Object user = request.getSession().getAttribute("loginUser");
        if(user == null){
            //未登陆，返回登陆页面
            request.setAttribute("msg","没有权限请先登陆");

 request.getRequestDispatcher("/index.html").forward(request,response)
;
            return false;
        }else{
            //已登陆，放行请求
            return true;
        }
    }

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

    @Override
    public void afterCompletion(HttpServletRequest request,
HttServletResponse response,
                                Object handler, Exception ex) throws
Exception
    }
}
```

注册拦截器

```
//所有的WebMvcConfigurerAdapter组件都会一起起作用
@Bean //将组件注册在容器
public WebMvcConfigurerAdapter webMvcConfigurerAdapter(){
    WebMvcConfigurerAdapter adapter = new WebMvcConfigurerAdapter() {
        @Override
        public void addViewControllers(ViewControllerRegistry
registry) {
            registry.addViewController("/").setViewName("login");

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

```
 registry.addViewController("/main.html").setViewName("dashboard");
        }
        //注册拦截器
        @Override
        public void addInterceptors(InterceptorRegistry registry) {
            //super.addInterceptors(registry);
            //静态资源；  *.css , *.js
            //SpringBoot已经做好了静态资源映射，不用处理静态资源
            //addPathPatterns 添加拦截路径
            //excludePathPatterns 排除拦截路径
            registry.addInterceptor(new
LoginHandlerInterceptor()).addPathPatterns("/**")
                    .excludePathPatterns("/index.html","/","/user/login");
        }
    };
    return adapter;
}
```

⑤CRUD-员工列表

实验要求：

- 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=xxx	emp/{id}——DELETE

- 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

- 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（同CSS）
~{templatename::fragmentname}:模板名::片段名

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

```
<nav class="col-md-2 d-none d-md-block bg-light sidebar"
id="sidebar">
    <div class="sidebar-sticky">
        <ul class="nav flex-column">
            <li class="nav-item">
                <a class="nav-link active"
                   th:class="${activeUri=='main.html'?'nav-
link active':'nav-link'}"
                   href="#" th:href="@{/main.html}">
                    <svg xmlns="http://www.w3.org/2000/svg"
width="24" height="24"
                         viewBox="0 0 24 24" fill="none"
stroke="currentColor" stroke-width="2" stroke-linecap="round"
stroke-linejoin="round" class="feather feather-home">
                        <path d="M3 9l9-7 9 7v11a2 2 0 0 1-2
2H5a2 2 0 0 1-2-2z"></path>
                        <polyline points="9 22 9 12 15 12 15
22"></polyline>
                    </svg>
                    Dashboard <span class="sr-only">(current)
</span>
                </a>
            </li>
            <!--引入侧边栏;传入参数-->
            <div
th:replace="commons/bar::#sidebar(activeUri='emps')"></div>
```

⑥CRUD-员工添加

添加页面

```
<form>
    <div class="form-group">
        <label>LastName</label>
        <input type="text" class="form-control"
placeholder="zhangsan">
    </div>

    <div class="form-group">
        <label>Email</label>
        <input type="email" class="form-control"
placeholder="zhangsan@atguigu.com">
    </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">
            <label class="form-check-label">男</label>
        </div>
        <div class="fo rm-check form-check-inline">
            <input class="form-check-input" type="radio" name="gender"
value="0">
            <label class="form-check-label">女</label>
        </div>
    </div>

    <div class="form-group">
        <label>department</label>
        <select class="form-control">
            <option>1</option>
            <option>2</option>
            <option>3</option>
            <option>4</option>
            <option>5</option>
        </select>
    </div>

    <div class="form-group">
        <label>Birth</label>
        <input type="text" class="form-control"
placeholder="zhangsan">
    </div>
```

```
    <button type="submit" class="btn btn-primary">添加</button>
</form>
```

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；

配置日期格式

```
spring.mvc.date-format=yyyy-MM-dd
```

⑦CRUD-员工修改

修改添加二合一表单

```
<!--需要区分是员工修改还是添加；-->
<form th:action="@{/emp}" method="post">
    <!--发送put请求修改员工数据-->
    <!--
        1、SpringMVC中配置HiddenHttpMethodFilter;（SpringBoot自动配置好的）
        2、页面创建一个post表单
        3、创建一个input项，name="_method";值就是我们指定的请求方式
    -->
    <input type="hidden" name="_method" value="put" th:if="${emp!=null}"/>
    <input type="hidden" name="id" th:if="${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>
        <input name="birth" type="text" class="form-control"
placeholder="zhangsan"
               th:value="${emp!=null}?${#dates.format(emp.birth,
'yyyy-MM-dd HH:mm')}">
    </div>
    <button type="submit" class="btn btn-primary"
th:text="${emp!=null}?'修改':'添加'">添加
    </button>
</form>
```

⑧CRUD-员工删除

```
<tr th:each="emp:${emps}">
    <td th:text="${emp.id}"></td>
    <td>[[${emp.lastName}]]</td>
    <td th:text="${emp.email}"></td>
    <td th:text="${emp.gender}==0?'女':'男'"></td>
    <td th:text="${emp.department.departmentName}"></td>
```

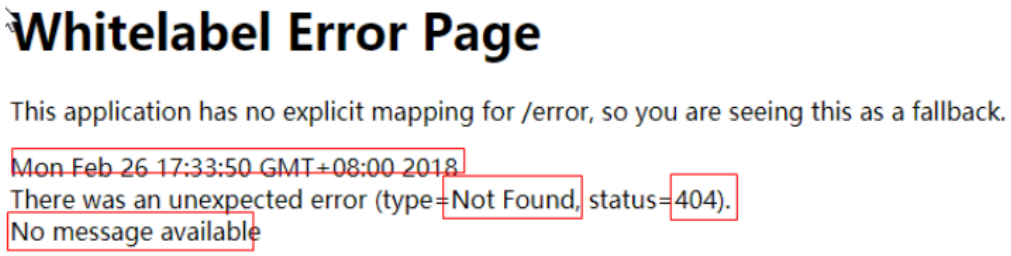
```
    <td th:text="${#dates.format(emp.birth, 'yyyy-MM-dd HH:mm')}">
</td>
    <td>
        <a class="btn btn-sm btn-primary"
th:href="@{/emp/}+${emp.id}">编辑</a>
        <button th:attr="del_uri=@{/emp/}+${emp.id}" class="btn btn-sm
btn-danger

 deleteBtn">删除</button>
    </td>
</tr>
<script>
    $(".deleteBtn").click(function(){
        //删除当前员工的

 $("#deleteEmpForm").attr("action",$(this).attr("del_uri")).submit();
        return false;
    });
</script>
```

7、错误处理机制

①SpringBoot默认的错误处理机制

默认效果

- 浏览器，返回一个默认的错误页面



- 如果是其他客户端，默认响应一个json数据

原理：

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

给容器中添加了一下组件

1. DefaultErrorAttributes
2. BasicErrorController：处理默认/error请求
3. ErrorPageCustomizer
4. DefaultErrorViewResolver

步骤

一旦系统中初心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

```
protected ModelAndView resolveErrorView(HttpServletRequest request,
                                        HttpServletResponse response,
HttpStatus status,
                                        Map<String, Object> model) {
    //所有的ErrorViewResolver得到ModelAndView
    for (ErrorViewResolver resolver : this.errorViewResolvers) {
        ModelAndView modelAndView = resolver.resolveErrorView(request,
status, model);
        if (modelAndView != null) {
            return modelAndView;
        }
    }
    return null;
}
```

②如何定制错误响应

1. **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】，发生状态码错误就会来到对应的页面

       页面能获取的信息**【使用行内写法获取】**

       timestamp：时间戳
       status：状态码
       error：错误提示
       exception：异常对象
       message：异常消息
       errors：JSR303数据校验的错误都在这里
    2. 没有模板引擎（模板引擎找不到这个错误页面），静态资源文件夹下找
    3. 以上都是错误页面，就是默认来到SpringBoot默认的错误提示页面
2. **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

```
@ControllerAdvice
public class MyExceptionHandler{
    @ResponseBody
    @ExceptonHandler(UserNotExitException.class)
    public Map<String,Object> handleException(Exception
e){
        Map<String,Object> map = new HashMap<>();
        map.put("code","user.notexit");
        map.put("message",e.getMessage());
        return map;
    }
}
//没有自适应效果
```

2. 转发到/error进行自适应响应效果处理

```
@ExceptionHandler(UserNotExistException.class)
public String handleException(Exception e,
HttpServletRequest request){
    Map<String,Object> map = new HashMap<>();
    //传入我们自己的错误状态码 4xx 5xx，否则就不会进入定制错误
页面的解析流程
    /**
        * Integer statusCode = (Integer) request

.getAttribute("javax.servlet.error.status_code");
    */

 request.setAttribute("javax.servlet.error.status_code"
,500);
    map.put("code","user.notexist");
    map.put("message",e.getMessage());
    //转发到/error
    return "forward:/error";
}
```

3. 将定制数据携带出去

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AttributeErrorController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

```
//给容器中加入我们自己定义的ErrorAttributes
@Component
public class MyErrorAttributes extends
DefaultErrorAttributes {
    @Override
    public Map<String, Object>
getErrorAttributes(RequestAttributes
requestAttributes,boolean includeStackTrace) {
        Map<String, Object> map =
super.getErrorAttributes(requestAttributes,

    includeStackTrace);
        map.put("company","atguigu");
        return map;
    }
}
```

8、配置嵌入式Servlet容器

*SpringBoot*默认使用的是*Tomcat*作为嵌入式的*Servlet*容器

①如何定制和修改Servlet容器的相关配置

两种方式实现

1. 修改和Servlet有关的配置【ServletProperties 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

   写在MyMvcConfig.java中

```
@Bean //一定要将这个定制器加入到容器中
public EmbeddedServletContainerCustomizer
embeddedServletContainerCustomizer(){
    return new EmbeddedServletContainerCustomizer() {
        //定制嵌入式的Servlet容器相关的规则
        @Override
        public void
customize(ConfigurableEmbeddedServletContainer container) {
            container.setPort(8083);
        }
    };
}
```

②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【写在MyMvcConfig.java】

ServletRegistrationBean

```
//注册Servlet
@Bean
public ServletRegistrationBean myServlet(){
    ServletRegistrationBean registrationBean = new
ServletRegistrationBean(new MyServlet(),"/myServlet");
    return registrationBean;
}
```

FilterRegistrationBean

```
//注册Filter
@Bean
public FilterRegistrationBean myFilter(){
    FilterRegistrationBean registrationBean = new
FilterRegistrationBean();
    registrationBean.setFilter(new MyFilter());

 registrationBean.setUrlPatterns(Arrays.asList("/hello","/myServlet"))
;
    return registrationBean;
}
```

ServletListenerRegistrationBean

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

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

DispatcherServletAutoConfiguration中

```
@Bean(name = DEFAULT_DISPATCHER_SERVLET_REGISTRATION_BEAN_NAME)
@ConditionalOnBean(value = DispatcherServlet.class, name =
                   DEFAULT_DISPATCHER_SERVLET_BEAN_NAME)
public ServletRegistrationBean dispatcherServletRegistration(
    DispatcherServlet dispatcherServlet) {
    ServletRegistrationBean registration = new
ServletRegistrationBean(
        dispatcherServlet, this.serverProperties.getServletMapping());
    //默认拦截： / 所有请求；包静态资源，但是不拦截jsp请求； /*会拦截jsp
    //可以通过server.servletPath来修改SpringMVC前端控制器默认拦截的请求路径
    registration.setName(DEFAULT_DISPATCHER_SERVLET_BEAN_NAME);
    registration.setLoadOnStartup(
        this.webMvcProperties.getServlet().getLoadOnStartup());
    if (this.multipartConfig != null) {
        registration.setMultipartConfig(this.multipartConfig);
    }
    return registration;
}
```

③替换为其他嵌入式Servlet容器

Tomcat【默认】

```
<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【长链接】

```
<!--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【不支持JSP】

```
<!-- 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
```

④嵌入式Servlet容器自动配置原理

EmbeddedServletContainerAutoConfiguration：嵌入式的Servlet容器自动配置?

```
@AutoConfigureOrder(Ordered.HIGHEST_PRECEDENCE)
@Configuration
@ConditionalOnWebApplication
@Import(BeanPostProcessorsRegistrar.class)
//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
public class EmbeddedServletContainerAutoConfiguration {
```

```
    @Configuration
    @ConditionalOnClass({ Servlet.class, Tomcat.class })//判断当前是否引入了Tomcat依赖；
    @ConditionalOnMissingBean(value =
EmbeddedServletContainerFactory.class, search =
                                SearchStrategy.CURRENT)//判断当前容器没有用户自己定义EmbeddedServletContainerFactory：嵌入式的 Servlet容器工厂；作用：创建嵌入式的Servlet容器
        public static class EmbeddedTomcat {
            @Bean
            public TomcatEmbeddedServletContainerFactory
tomcatEmbeddedServletContainerFactory(){
                return new TomcatEmbeddedServletContainerFactory();
            }
        }
    /**
        * Nested configuration if Jetty is being used.
    */
    @Configuration
    @ConditionalOnClass({ Servlet.class, Server.class, Loader.class,
                        WebAppContext.class })
    @ConditionalOnMissingBean(value =
EmbeddedServletContainerFactory.class, search =
                                SearchStrategy.CURRENT)
    public static class EmbeddedJetty {
        @Bean
        public JettyEmbeddedServletContainerFactory
jettyEmbeddedServletContainerFactory() {
            return new JettyEmbeddedServletContainerFactory();
        }
    }
    /**
        * Nested configuration if Undertow is being used.
    */
    @Configuration
    @ConditionalOnClass({ Servlet.class, Undertow.class,
SslClientAuthMode.class })
    @ConditionalOnMissingBean(value =
EmbeddedServletContainerFactory.class, search =
                                SearchStrategy.CURRENT)
    public static class EmbeddedUndertow {
        @Bean
        public UndertowEmbeddedServletContainerFactory
            undertowEmbeddedServletContainerFactory() {
            return new UndertowEmbeddedServletContainerFactory();
        }
    }
}
```

1. EmbeddedServletContainerFactory（嵌入式Servlet容器工厂）
2. EmbeddedServletContainer：（嵌入式的Servlet容器）
3. 以TomcatEmbeddedServletContainerFactory为例

```
@Override
public EmbeddedServletContainer
getEmbeddedServletContainer(ServletContextInitializer...
initializers) {
    //创建一个Tomcat
    Tomcat tomcat = new Tomcat();
    //配置Tomcat的基本环节
    File baseDir = (this.baseDirectory != null ?
this.baseDirectory:
                    createTempDir("tomcat"));
    tomcat.setBaseDir(baseDir.getAbsolutePath());
    Connector connector = new Connector(this.protocol);
    tomcat.getService().addConnector(connector);
    customizeConnector(connector);
    tomcat.setConnector(connector);
    tomcat.getHost().setAutoDeploy(false);
    configureEngine(tomcat.getEngine());
    for (Connector additionalConnector :
this.additionalTomcatConnectors) {
        tomcat.getService().addConnector(additionalConnector);
    }
    prepareContext(tomcat.getHost(), initializers);
    //将配置好的Tomcat传入进去，返回一个EmbeddedServletContainer；
并且启动Tomcat服务器
    return getTomcatEmbeddedServletContainer(tomcat);
}
```

怎么修改的原理

1. 容器中导入了EmbeddedServletContainerCustomizerBeanPostProcessor

步骤：

1）、SpringBoot根据导入的依赖情况，给容器中添加相应的EmbeddedServletContainerFactory【TomcatEmbeddedServletContainerFactory】

2）、容器中某个组件要创建对象就会惊动后置处理器；EmbeddedServletContainerCustomizerBeanPostProcessor；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

⑤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；
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：
    1. EmbeddedServletContainerFactory containerFactory =
       getEmbeddedServletContainerFactory();
    2. 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

9、使用外置的Servlet容器

嵌入式Servlet容器：应用打包成可执行的jar

优点：简单、便携

缺点：默认不支持JSP、优先定制比较复杂（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；

①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Tomact指定为provided

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-tomcat</artifactId>
    <scope>provided</scope>
</dependency>
```

3. 必须便携一个SpringBootServletInitializer的子类，并调用config方法

```
public class ServletInitializer extends
SpringBootServletInitializer {
    @Override
    protected SpringApplicationBuilder
configure(SpringApplicationBuilder application) {
        //传入SpringBoot应用的主程序
        return
application.sources(SpringBoot04WebJspApplication.class);
    }
}
```

4. 启动服务器就可以使用

②原理

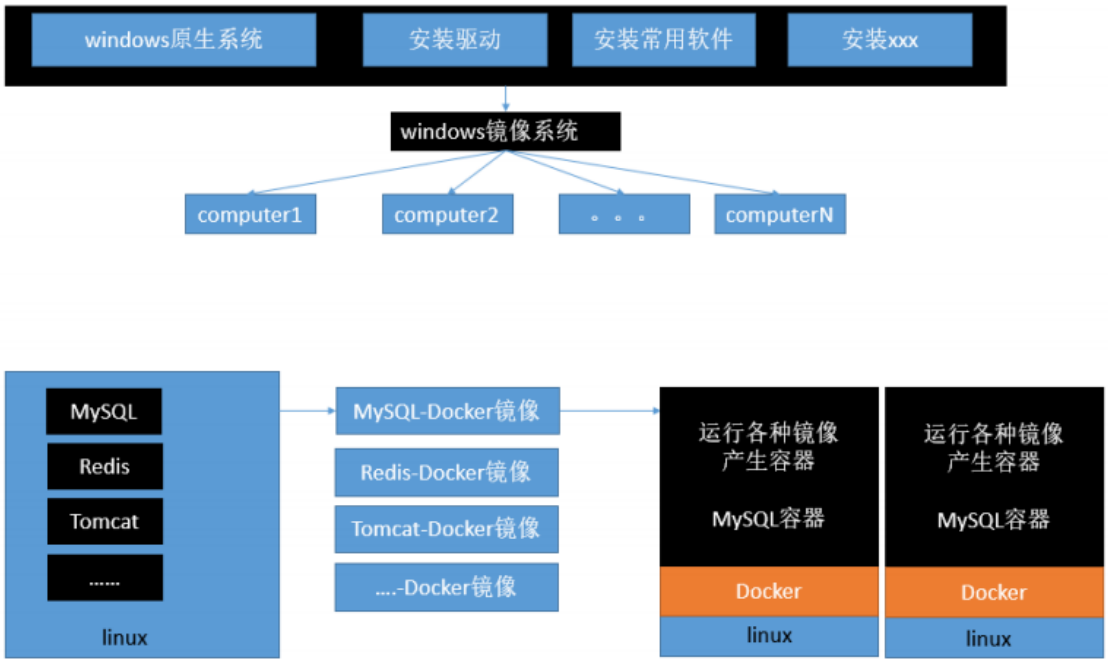
五、Docker

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，容器启动时非常快速的



2、核心概念

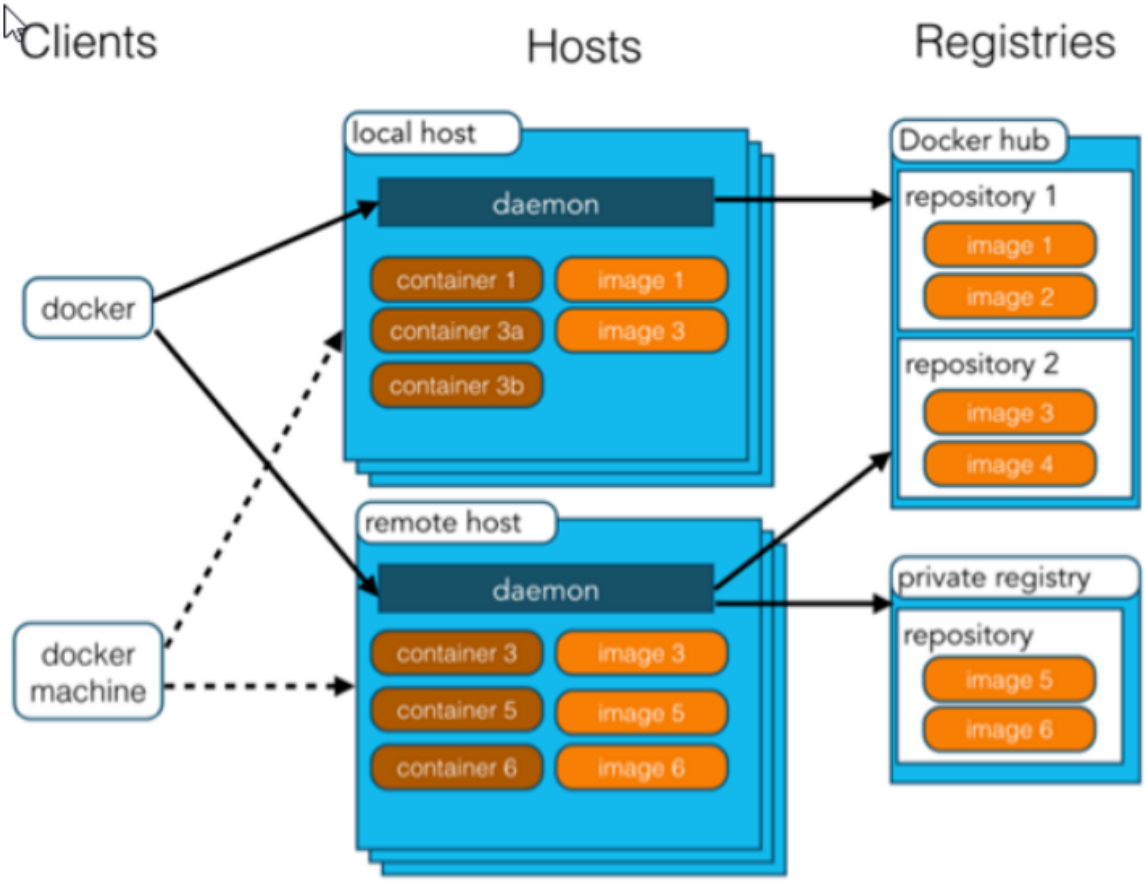
docker主机（Host）：安装了Docker程序的机器（Docker直接安装在操作系统之上）

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

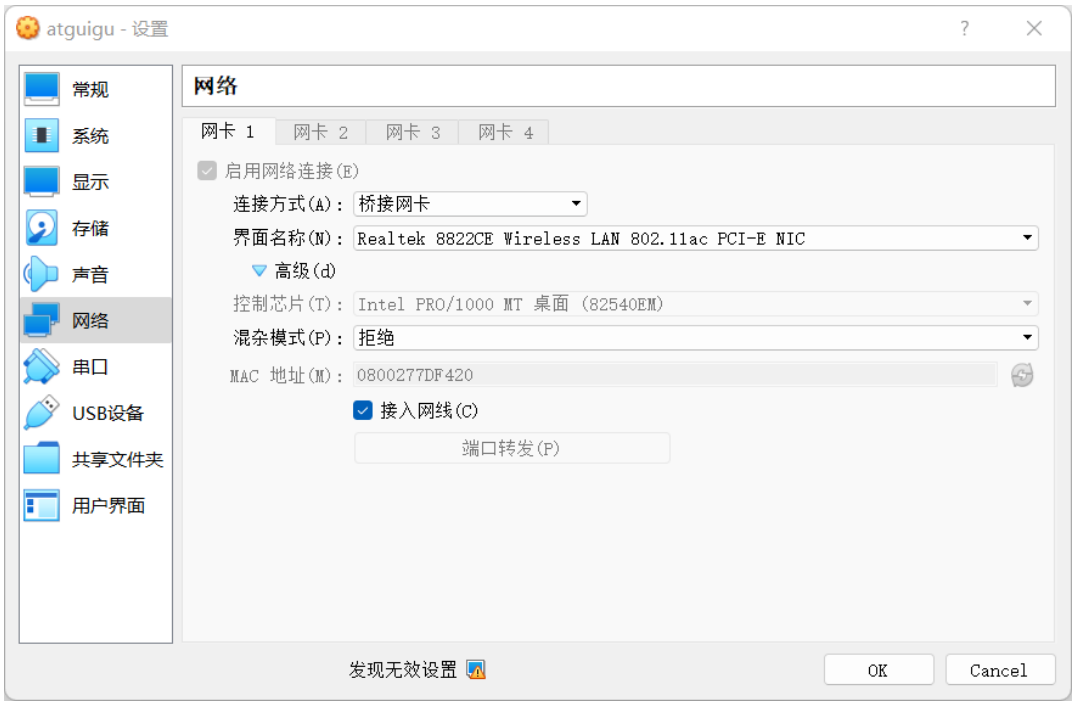
1. 安装docker
2. 去docker仓库找到这个软件对应的镜像
3. 使用docker运行这个镜像，这个镜像就会生成一个Docker容器
4. 对容器的启动停止就是对软件的启动停止

3、安装Docker

①安装Linux虚拟机

1. VMWare、VirtualBox（安装） https://www.virtualbox.org/wiki/Downloads
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； SmarTTY

5. 设置虚拟机网络——桥接网==选好网卡==加入网线 【无线网络为例 选择含有Wireless字段】



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【建立新的SSH，使用linux的ip和root 123456进行连接】

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

②容器操作

③安装MySQL示例

六、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org.apache.tomcat.jdbc.pool.DataSource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Tomcat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

```
@ConditionalOnMissingBean(DataSource.class)
@ConditionalOnProperty(name = "spring.datasource.type")
static class Generic {
    @Bean
    public 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，并且绑定相关属性
        return
properties.initializeDataSourceBuilder().build();
    }
}
```

4. DataSourceInitializer：ApplicationListener；

    1. 作用
    2. runSchemaScripts();运行建表语句；
    3. 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；
可以使用
    schema:
        - classpath:department.sql
        指定位置
```

5. 操作数据库：自动配置了JDBCTemplate

2、整合Druid数据源

```
//导入druid数据源
@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");
        //默认访问所有
        initParams.put("allow","");
        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 WebStaFilter());
        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>1.3.1</version>
</dependency
```

步骤

1. 配置数据源相关属性
2. 给数据库建表
3. 创建javabean

①注解版

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

    @Select("delete from department where id = #{id}")
    public int deleteDeptById(Integer id);

    @Options(useGeneratedKeys = true,keyProperty = "id")
    @Insert("insert into department(departmentName) values(#
{departmentName})")
    public int insertDept(Department department);

    @Update("update department set departmentName=#{departmentName}
where id=#{id}")
    public int updateDept(Department department);
}
```

问题：

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

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

```
//使用MapperScan批量扫描所有的Mapper接口
@MapperScan(value = "com.atguigu.springboot.mapper")
@SpringBootApplication
public class SpringBoot06DataMybatisApplication{
    public static void main(String[] args){

 SpringApplication.run(SpringBoot06DataMybatisApplication.class,args);
    }
}
```

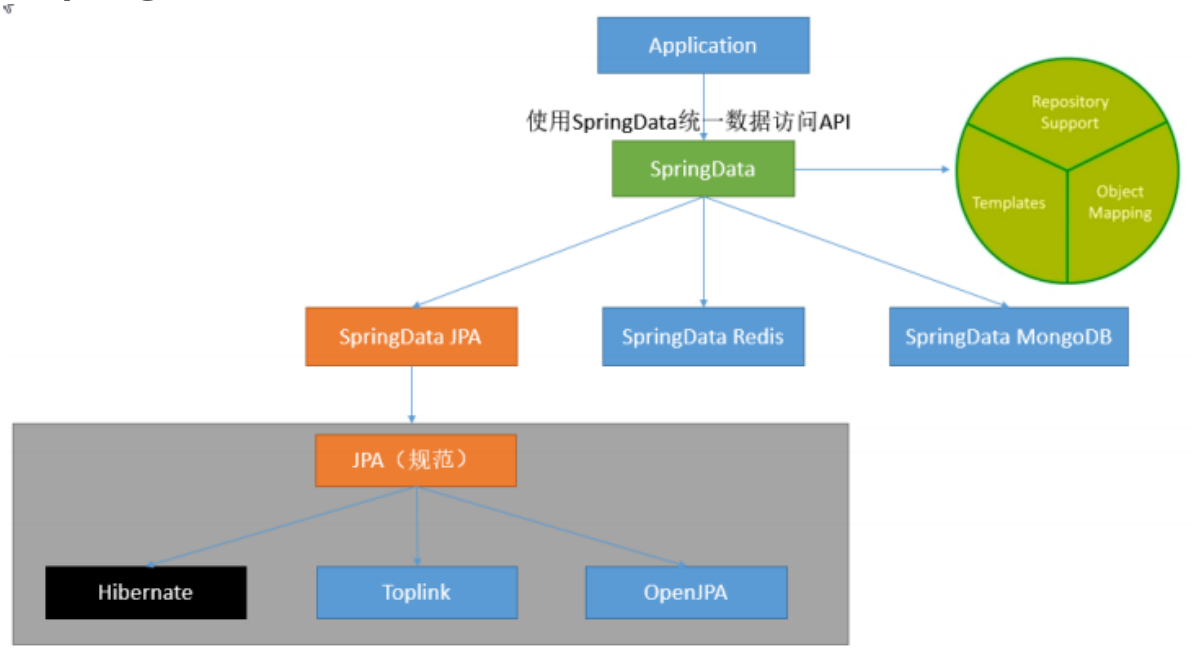
②配置文件版

```
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
```

参照 http://mybatis.org/spring-boot-starter/mybatis-spring-boot-autoconfigure/

4、整合SpringData JPA

①SpringData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

```
//使用JPA注解配置映射关系
@Entity   //告诉JPA这是一个实体类（和数据表映射关系）
@Table(name = "tbl_user")
public class User{
    @Id  //这是一个主键
    @GeneratedValue(strategy = GenerationType.IDENTITY)  //自增主键
    private Integer id;
```

```
    @Column(name = "last_name" ,length = 50)  //这是和数据表对应的一个列
    private String lastName;

    @Column  //省略列名就是属性名
    private String 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来完成对数据库的操作
public interface UserRepository extends
JpaRepository<User,Integer>{}
```

3. 基本的配置jdbcProperties

```
spring:
    jpa:
        hibernate:
        #更新或者创建数据表结构
        ddl-auto: update
        #控制台显示SQL
        show-sql: true
```

七、启动配置原理

几个重要的事件回溯机制

配置在META-INF/spring.factories

ApplicationContextInitializer

SpringApplicationRunListener

只需要放在ioc容器中

ApplicationRunner

CommandLineRunner

1、创建SpringApplication

```
private void initialize(Object[] sources) {
    //保存主配置类
    if (sources != null && sources.length > 0) {
        this.sources.addAll(Arrays.asList(sources));
```

```
    }
    //判断当前是否一个web应用
    this.webEnvironment = deduceWebEnvironment();
    //从类路径下找到META-INF/spring.factories配置的所有
ApplicationContextInitializer；然后保存起来
    setInitializers((Collection) getSpringFactoriesInstances(
        ApplicationContextInitializer.class));
    //从类路径下找到ETA-INF/spring.factories配置的所有ApplicationListener
    setListeners((Collection)
getSpringFactoriesInstances(ApplicationListener.class));
    //从多个配置类中找到有main方法的主配置类
    this.mainApplicationClass = deduceMainApplicationClass();
}
```

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（）；
            //s刷新容器；ioc容器初始化（如果是web应用还会创建嵌入式的Tomcat）；
Spring注解版
            //扫描，创建，加载所有组件的地方；（配置类，组件，自动配置）
            refreshContext(context);
            //从ioc容器中获取所有的ApplicationRunner和CommandLineRunner进行回
调
            //ApplicationRunner先回调，CommandLineRunner再回调
            afterRefresh(context, applicationArguments);
            //所有的SpringApplicationRunListener回调finished方法
            listeners.finished(context, null);
            stopWatch.stop();
            if (this.logStartupInfo) {
                new StartupInfoLogger(this.mainApplicationClass)
                    .logStarted(getApplicationLog(), stopWatch);
            }
            //整个SpringBoot应用启动完成以后返回启动的ioc容器；
            return context;
        }
        catch (Throwable ex) {
            handleRunFailure(context, listeners, analyzers, ex);
            throw new IllegalStateException(ex);
        }
    }
```

3、事件监听器机制

八、自定义starter

附录

1、官方文档

https://docs.spring.io/spring-boot/docs/1.5.9.RELEASE/