

内容概要

- 一、Spring Boot入门
- 二、Spring Boot配置
- 三、Spring Boot与日志
- 四、Spring Boot与Web开发
- 五、Spring Boot与Docker
- 六、Spring Boot与数据访问
- 七、Spring Boot启动配置原理
- 八、Spring Boot自定义starters
- 九、Spring Boot与缓存
- 十、Spring Boot与消息
- 十一、Spring Boot与检索
- 十二、Spring Boot与任务
- 十三、Spring Boot与安全
- 十四、Spring Boot与分布式
- 十五、Spring Boot与开发热部署
- 十六、Spring Boot与监控管理

一、Spring Boot 入门

- 1、Spring Boot 简介
- 2、微服务
- 3、环境准备
 - 1、MAVEN设置；
 - 2、IDEA设置
- 4、Spring Boot HelloWorld
 - 1、创建一个maven工程； (jar)
 - 2、导入spring boot相关的依赖
 - 3、编写一个主程序；启动Spring Boot应用
 - 4、编写相关的Controller、Service
 - 5、运行主程序测试
 - 6、简化部署
- 5、Hello World探究
 - 1、POM文件
 - 1、父项目
 - 2、启动器
 - 2、主程序类，主入口类
- 6、使用Spring Initializer快速创建Spring Boot项目
 - 1、IDEA：使用 Spring Initializer快速创建项目
 - 2、STS使用 Spring Starter Project快速创建项目(另一种idea)

二、配置文件

- 1、配置文件
- 2、YAML语法：
 - 1、基本语法
 - 2、值的写法

字面量：普通的值（数字，字符串，布尔）

对象、Map（属性和值）（键值对）：

数组（List、Set）：

3、配置文件值注入

- 1、properties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
- 3、配置文件注入值数据校验
- 4、@PropertySource & @ImportResource & @Bean

4、配置文件占位符

- 1、随机数
- 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

5、Profile

- 1、多Profile文件
- 2、yml支持多文档块方式
- 3、激活指定profile

6、配置文件加载位置

7、外部配置加载顺序

8、自动配置原理

- 1、**自动配置原理：**
- 2、细节

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

三、日志

1、日志框架

2、SLF4J使用

- 1、如何在系统中使用SLF4J <https://www.slf4j.org>
- 2、遗留问题

3、SpringBoot日志关系

4、日志使用；

- 1、默认配置
- 2、指定配置

5、切换日志框架

四、Web开发

1、简介

2、SpringBoot对静态资源的映射规则；

3、模板引擎

- 1、引入thymeleaf；
- 2、Thymeleaf使用
- 3、语法规则

4、SpringMVC自动配置

1. Spring MVC auto-configuration
2. 扩展SpringMVC
3. 全面接管SpringMVC；

5、如何修改SpringBoot的默认配置

6、RestfulCRUD

- 1) 、默认访问首页
- 2) 、国际化
- 3) 、登陆
- 4) 、拦截器进行登陆检查
- 5) 、CRUD-员工列表
thymeleaf公共页面元素抽取
- 6) 、CRUD-员工添加
- 7) 、CRUD-员工修改
- 8) 、CRUD-员工删除

7、错误处理机制

- 1) 、SpringBoot默认的错误处理机制
- 2) 、定制错误响应：
 - 1) 、**如何定制错误的页面；**
 - 2) 、如何定制错误的json数据；

3) 、将我们的定制数据携带出去;

8、配置嵌入式Servlet容器

- 1) 、如何定制和修改Servlet容器的相关配置;
- 2) 、注册Servlet三大组件【Servlet、Filter、Listener】
- 3) 、替换为其他嵌入式Servlet容器
- 4) 、嵌入式Servlet容器自动配置原理;
- 5) 、嵌入式Servlet容器启动原理;

9、使用外置的Servlet容器

步骤

原理

五、Docker

1、简介

2、核心概念

3、安装Docker

- 1) 、安装linux虚拟机
- 2) 、在linux虚拟机上安装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

2、整合Druid数据源

3、整合MyBatis

- 4) 、注解版
- 5) 、配置文件版

4、整合SpringData JPA

- 1) 、SpringData简介
- 2) 、整合SpringData JPA

七、启动配置原理

1、创建SpringApplication对象

2、运行run方法

3、事件监听机制

八、自定义starter

更多SpringBoot整合示例

一、SpringBoot与缓存

(一) JSR-107、

(二) Spring缓存抽象

- 1、
- 2、Cache接口
- 3、每次调用需要缓存功能的方法时，
- 4、使用Spring缓存抽象时我们需要关注以下两点

(三) 几个重要概念&缓存注解

- 1、
- 2、@Cacheable、@CachePut、@CacheEvict 主要的参数
- 3、Cache SpEL available metadata

(四) 缓存使用

(五) 整合Redis

二、SpringBoot与消息

(一) 概述

- 1、
- 2、消息服务中两个重要概念：
- 3、消息队列主要有两种形式的目的地
- 4、点对点式：
- 5、发布订阅式：
- 6、JMS (Java Message Service)
- 7、AMQP (Advanced Message Queuing Protocol)
- 8、

9、Spring支持
10、Spring Boot自动配置

(二)、RabbitMQ简介

0、开启RabbitMQ：
1、RabbitMQ简介：
2、核心概念

(三)、RabbitMQ运行机制

(四)、RabbitMQ整合

三、SpringBoot与检索

(一)、检索
(二)、概念
(三)、整合ElasticSearch测试
(四)、具体使用

四、SpringBoot与任务

(一)、异步任务
(二)、定时任务
(三)、邮件任务

五、SpringBoot与安全

(一)、安全
(二)、Web&安全
1、登陆/注销
2、Thymeleaf提供的SpringSecurity标签支持
3、remember me
4、CSRF (Cross-site request forgery) 跨站请求伪造

六、SpringBoot与分布式

(一)、分布式应用
(二)、Zookeeper和Dubbo
(三)、Spring Boot和Spring Cloud

七、SpringBoot与热部署

八、SpringBoot与监控管理

(一)、监控管理
(二)、定制端点信息



一、Spring Boot 入门

1、Spring Boot 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；

Spring Boot是Spring的简化

约定大于配置

“Spring全家桶”时代

Spring Boot: J2EE一站式解决方案

Spring Cloud : 分布式整体解决方案

使用嵌入式的Servlet容器，应用无需打成WAR包

大量的自动配置，简化开发，也可修改默认值



2、微服务

2014, martin fowler

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；

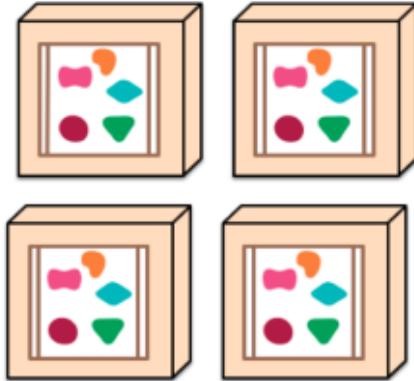
[详细参照微服务文档](#)

微服务

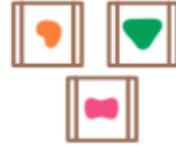
一个单体应用程序把它所有的功能放在一个单一进程中...



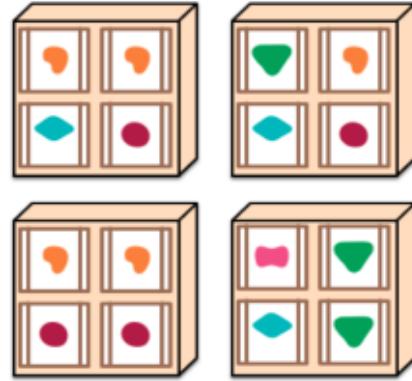
...并且通过在多个服务器上复制这个单体进行扩展



一个微服务架构把每个功能元素放进一个独立的服务中...



...并且通过跨服务器分发这些服务进行扩展，只在需要时才复制。



3、环境准备

<http://www.gulixueyuan.com/> 谷粒学院

环境约束

-jdk1.8: Spring Boot 推荐jdk1.7及以上; java version "1.8.0_112"

-maven3.x: maven 3.3以上版本; Apache Maven 3.3.9

-IntelliJIDEA2017: IntelliJ IDEA 2017.2.2 x64、STS

-SpringBoot 1.5.9.RELEASE: 1.5.9;

统一环境;

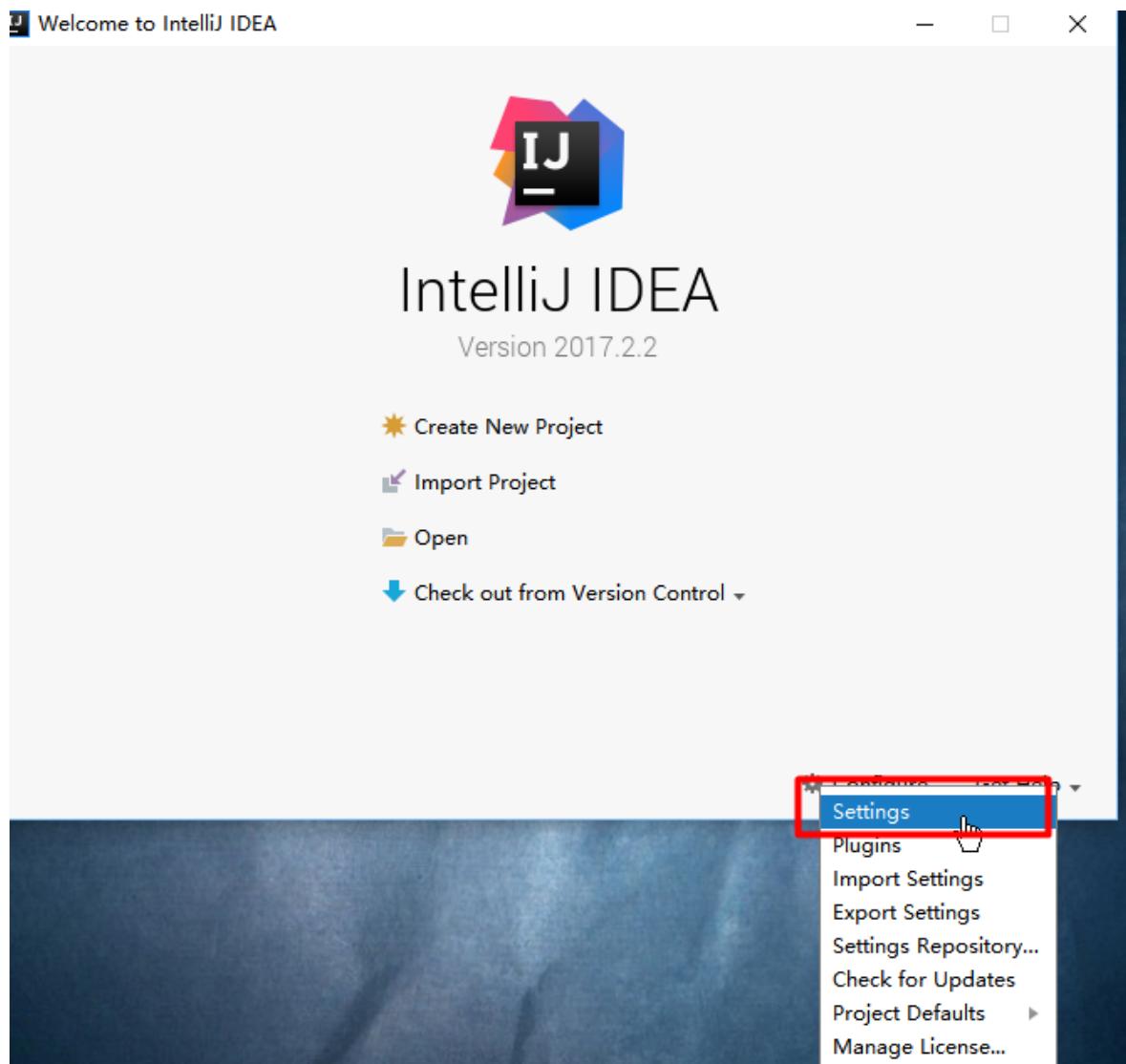
1、MAVEN设置：

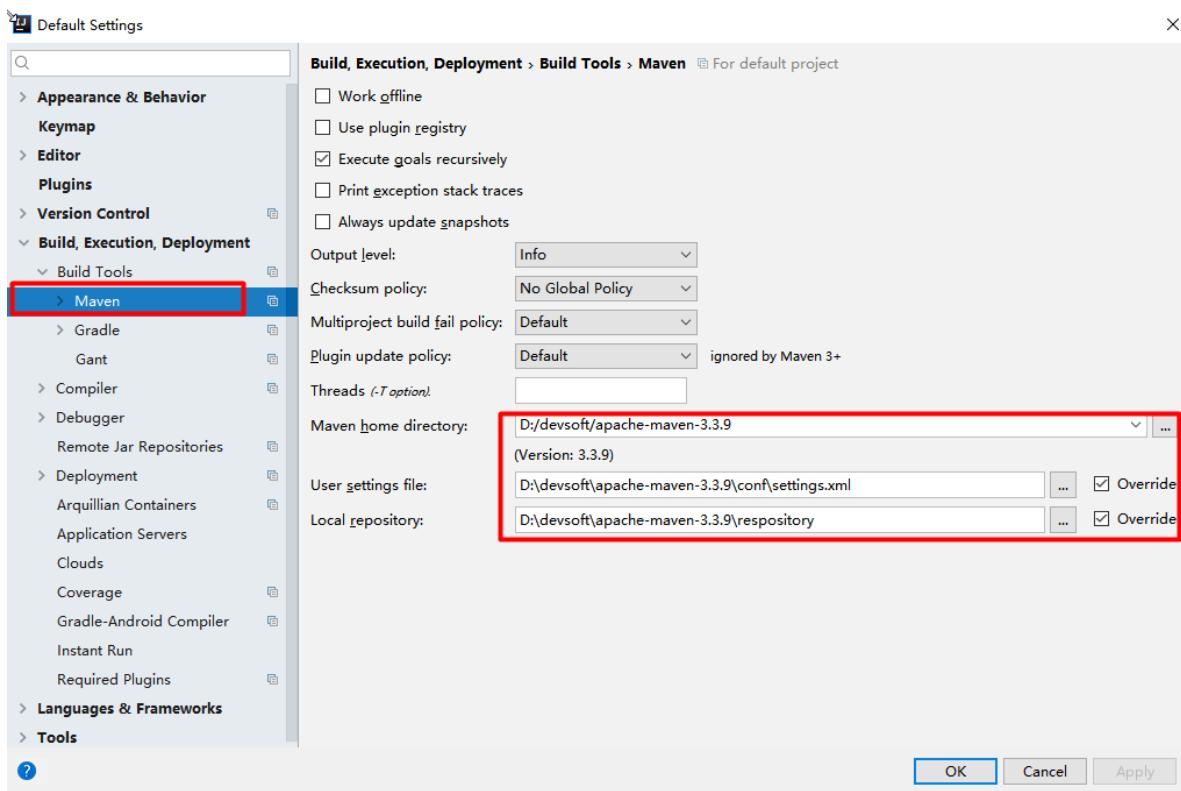
给maven 的settings.xml配置文件的profiles标签添加

```
<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>
```

2、IDEA设置

整合maven进来；





4、Spring Boot HelloWorld

一个功能：

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

1、创建一个maven工程；（jar）

2、导入spring boot相关的依赖

```
<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Spring Boot应用

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

4、编写相关的Controller、Service

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

5、运行主程序测试

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>
```

将这个应用打成jar包，直接使用java -jar的命令进行执行；

5、Hello World探究

1、POM文件

1、父项目

```
<parent>
```

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

他的父项目是

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

他来真正管理Spring Boot应用里面的所有依赖版本；

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）

2、启动器

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

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；

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

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Spring Boot应用
 */
@SpringBootApplication
public class HelloWorldMainApplication {

    public static void main(String[] args) {

        // Spring应用启动起来
        SpringApplication.run(HelloWorldMainApplication.class, args);
    }
}
```

@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；

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

@SpringBootConfiguration: Spring Boot的配置类；

标注在某个类上，表示这是一个Spring Boot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，Spring Boot帮我们自动配置；**@EnableAutoConfiguration**告诉SpringBoot开启自动配置功能；这样自动配置才能生效；

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

@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）的所在包及下面所有子包里面的所有组件扫描到Spring容器；

@Import(EnableAutoConfigurationImportSelector.class);

给容器中导入组件？

EnableAutoConfigurationImportSelector: 导入哪些组件的选择器；

将所有需要导入的组件以全类名的方式返回；这些组件就会被添加到容器中；

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；

```
> 0 = "org.springframework.boot.autoconfigure.admin.SpringApplicationAdminJmxAutoConfiguration"
onClassPostProcessor (org.springframework.context.annotation) AopAutoConfiguration"
> 2 = "org.springframework.boot.autoconfigure.amqp.RabbitAutoConfiguration"
> 3 = "org.springframework.boot.autoconfigure.batch.BatchAutoConfiguration"
> 4 = "org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration"
> 5 = "org.springframework.boot.autoconfigure.cassandra.CassandraAutoConfiguration"
> 6 = "org.springframework.boot.autoconfigure.cloud.CloudAutoConfiguration"
> 7 = "org.springframework.boot.autoconfigure.context.ConfigurationPropertiesAutoConfiguration"
> 8 = "org.springframework.boot.autoconfigure.context.MessageSourceAutoConfiguration"
> 9 = "org.springframework.boot.autoconfigure.context.PropertyPlaceholderAutoConfiguration"
> 10 = "org.springframework.boot.autoconfigure.couchbase.CouchbaseAutoConfiguration"
> 11 = "org.springframework.boot.autoconfigure.dao.PersistenceExceptionTranslationAutoConfiguration"
> 12 = "org.springframework.boot.autoconfigure.data.redis.RedisAutoConfiguration"
```

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工作；

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

Spring Boot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，自动配置类都帮我们；

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;

Spring注解版 (谷粒学院)

6、使用Spring Initializer快速创建Spring Boot项目

1、IDEA：使用 Spring Initializer 快速创建项目

IDE都支持使用Spring的项目创建向导快速创建一个Spring Boot项目；

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；

2、STS使用 Spring Starter Project快速创建项目(另一种idea)

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

```
server:  
  port: 8081
```

XML：

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

- 缩进时不允许使用Tab键，只允许使用空格。

- 缩进的空格数目不重要，只要相同层级的元素左侧对齐即可

2、YAML语法：

1、基本语法

k:(空格)v : 表示一对键值对（空格必须有）；

以空格的缩进来控制层级关系；只要是左对齐的一列数据，都是同一个层级的

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

属性和值也是大小写敏感；

2、值的写法

字面量：普通的值（数字，字符串，布尔）

k: v: 字面直接来写；

字符串默认不用加上单引号或者双引号；

": 双引号；不会转义字符串里面的特殊字符；特殊字符会作为本身想表示的意思

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

name: 'zhangsan \n lisi': 输出；zhangsan \n lisi

多行：可以写成多行，从第二行开始，必须有一个单空格缩进，换行符会被转为空格

对象、Map（属性和值）（键值对）：

k:

v: w

在下一行来写对象的属性和值的关系；注意缩进

对象还是k: v的方式

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

行内写法：

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

数组（List、Set）：

用|值表示数组中的一个元素

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

行内写法

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

3、配置文件值注入

配置文件

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

javaBean:

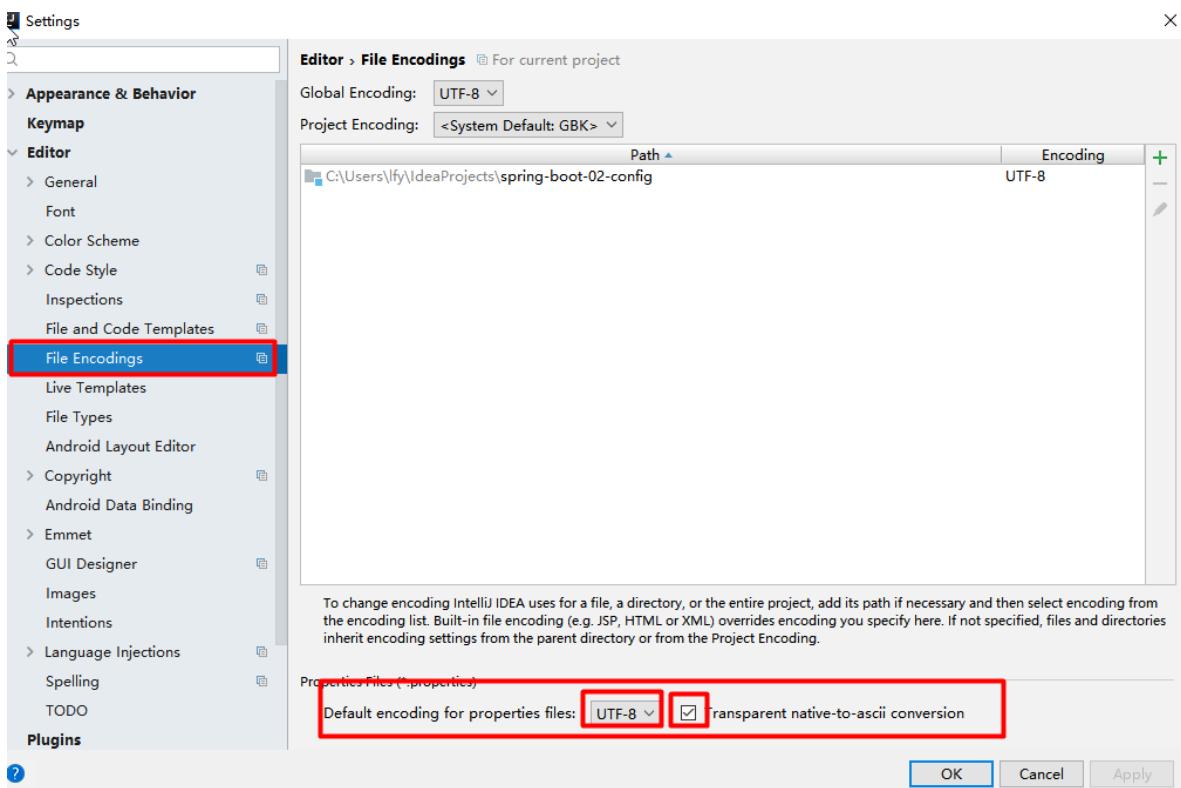
```
/**  
 * 将配置文件中配置的每一个属性的值，映射到这个组件中  
 *  
 * @ConfigurationProperties: 告诉SpringBoot将本类中的所有属性和配置文件中相关的配置进行绑定；  
 *      prefix = "person": 配置文件中哪个下面的所有属性进行一一映射  
 *  
 * 只有这个组件是容器中的组件，才能容器提供的@ConfigurationProperties功能；  
 *  
 */  
// 说明是一个组件  
@Component  
@ConfigurationProperties(prefix = "person")  
public class Person {  
  
    private String lastName;  
    private Integer age;  
    private Boolean boss;  
    private Date birth;  
  
    private Map<String, Object> maps;  
    private List<Object> lists;  
    private Dog dog;
```

我们可以导入配置文件处理器，以后编写配置就有提示了

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

1、properties配置文件在idea中默认utf-8可能会乱码

调整



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；

属性名匹配规则（Relaxed binding）（松散绑定）

- person.firstName: 使用标准方式
- person.first-name: 推荐在properties或yml文件中使用
- person.first_name:
- PERSON_FIRST_NAME: 推荐系统属性使用这种写法

3、配置文件注入值数据校验

```
@Component
@ConfigurationProperties(prefix = "person")

// 有效校验
@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

    // ${取值}
    //@value("${person.lastName}")
    private String lastName;

    // #{计算}
    //@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;

```
// 加载指定的配置文件
@PropertySource(value = {"classpath:person.properties"})

@Component
@ConfigurationProperties(prefix = "person")
//@Validated

public class Person {

    //@value("${person.lastName}")
    private String lastName;
    //@value("#{11*2}")
    private Integer age;
```

```
//@value("true")
private Boolean boss;
```

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

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；

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
```

不编写Spring的配置文件

```
<?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.atguigu.springboot.service.HelloService">
    </bean>

</beans>
```

SpringBoot推荐给容器中添加组件的方式；推荐使用全注解的方式

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 AppConfig {

    //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

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

```
# UUID 是指Universally Unique Identifier, 翻译为中文是通用唯一识别码  
person.last-name=张三${random.uuid}  
  
# 取random类的值  
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  
  
# ${person.未定义变量:缺省变量}  
person.dog.name=${person.hello:hello}_dog  
person.dog.age=15
```

5、Profile

1、多Profile文件

我们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；

2、yml支持多文档块方式

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

```
spring:  
  profiles: dev  
  
---  
server:  
  port: 8084  
  
spring:  
  profiles: prod
```

3、激活指定profile

1、在配置文件中指定 `spring.profiles.active=dev`

2、命令行：

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

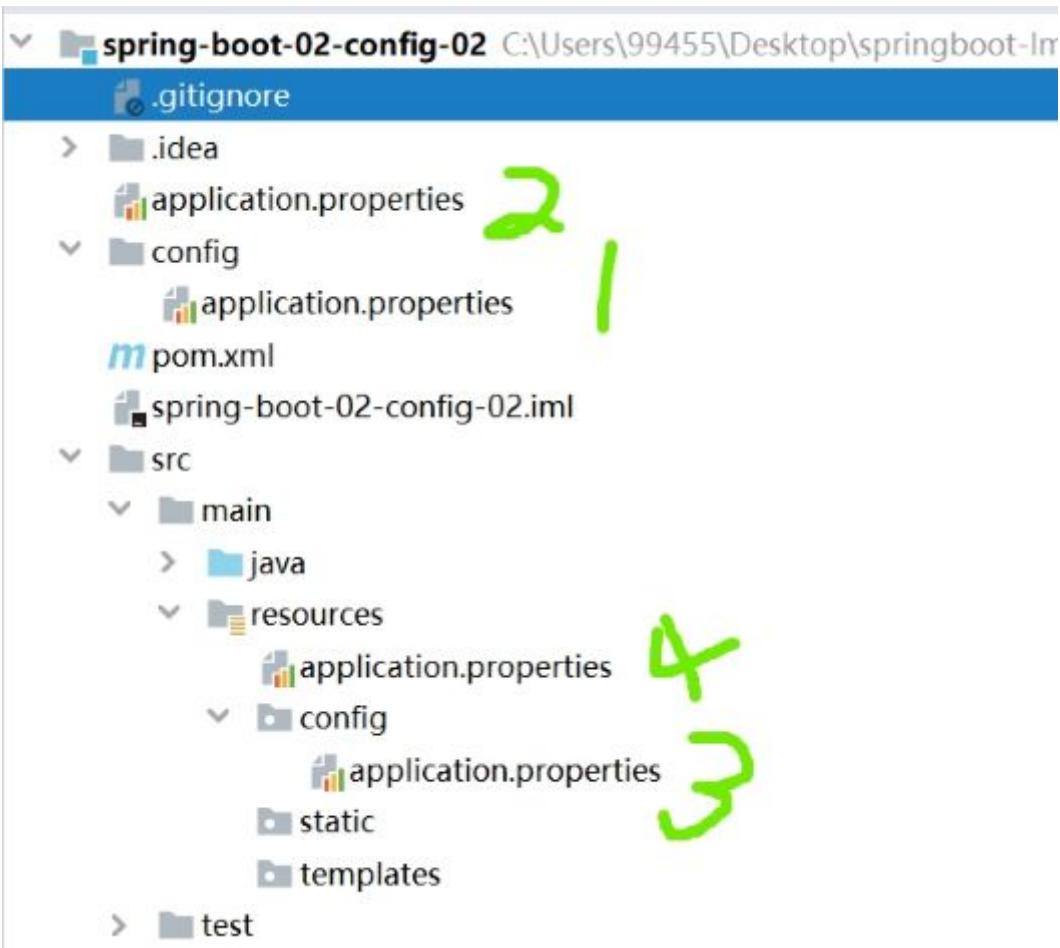
可以直接在测试的时候，配置传入命令行参数

3、虚拟机参数；

```
-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Spring boot的默认配置文件



-file:./config/

-file:./

-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**；

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

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

```
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

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

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

所有支持的配置加载来源；

[参考官方文档](#)

8、自动配置原理

配置文件到底能写什么？怎么写？自动配置原理；

[配置文件能配置的属性参照](#)

可以查看HttpEncodingAutoConfiguration

通用模式

- xxxAutoConfiguration: 自动配置类
- xxxProperties: 属性配置类
- yml/properties文件中能配置的值就来源于[属性配置类]

1、自动配置原理：

1) 、SpringBoot启动的时候加载主配置类，开启了自动配置功能 `@EnableAutoConfiguration`

2) 、`@EnableAutoConfiguration` 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<String> configurations = getCandidateConfigurations(annotationMetadata, attributes);` 获取候选的配置
 - `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，然后把它们添加在容器中

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；

```
# Auto Configure
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\

org.springframework.boot.autoconfigure.admin.SpringApplicationAdminJmxAutoConfiguration,\
org.springframework.boot.autoconfigure.aop.AopAutoConfiguration,\
org.springframework.boot.autoconfigure.amqp.RabbitAutoConfiguration,\
org.springframework.boot.autoconfigure.batch.BatchAutoConfiguration,\
org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration,\
org.springframework.boot.autoconfigure.cassandra.CassandraAutoConfiguration,\
org.springframework.boot.autoconfigure.cloud.CloudAutoConfiguration,\
org.springframework.boot.autoconfigure.context.ConfigurationPropertiesAutoConfiguration,\
org.springframework.boot.autoconfigure.context.MessageSourceAutoConfiguration,\
org.springframework.boot.autoconfigure.context.PropertyPlaceholderAutoConfiguration,\
org.springframework.boot.autoconfigure.couchbase.CouchbaseAutoConfiguration,\
org.springframework.boot.autoconfigure.dao.PersistenceExceptionTranslationAutoConfiguration,\
```

```
org.springframework.boot.autoconfigure.data.cassandra.CassandraDataAutoConfiguration,\  
org.springframework.boot.autoconfigure.data.cassandra.CassandraRepositoriesAutoConfiguration,\  
org.springframework.boot.autoconfigure.data.couchbase.CouchbaseDataAutoConfiguration,\  
org.springframework.boot.autoconfigure.data.couchbase.CouchbaseRepositoriesAutoConfiguration,\  
org.springframework.boot.autoconfigure.data.elasticsearch.ElasticsearchAutoConfiguration,\  
org.springframework.boot.autoconfigure.data.elasticsearch.ElasticsearchDataAutoConfiguration,\  
org.springframework.boot.autoconfigure.data.elasticsearch.ElasticsearchRepositoriesAutoConfiguration,\  
org.springframework.boot.autoconfigure.data.jpa.JpaRepositoriesAutoConfiguration,\  
org.springframework.boot.autoconfigure.data.ldap.LdapDataAutoConfiguration,\  
org.springframework.boot.autoconfigure.data.ldap.LdapRepositoriesAutoConfiguration,\  
org.springframework.boot.autoconfigure.data.mongo.MongoDataAutoConfiguration,\  
org.springframework.boot.autoconfigure.data.mongo.MongoRepositoriesAutoConfiguration,\  
org.springframework.boot.autoconfigure.data.neo4j.Neo4jDataAutoConfiguration,\  
org.springframework.boot.autoconfigure.data.neo4j.Neo4jRepositoriesAutoConfiguration,\  
org.springframework.boot.autoconfigure.data.solr.SolrRepositoriesAutoConfiguration,\  
org.springframework.boot.autoconfigure.data.redis.RedisAutoConfiguration,\  
org.springframework.boot.autoconfigure.data.redis.RedisRepositoriesAutoConfiguration,\  
org.springframework.boot.autoconfigure.data.rest.RepositoryRestMvcAutoConfiguration,\  
org.springframework.boot.autoconfigure.data.web.SpringDataWebAutoConfiguration,\  
org.springframework.boot.autoconfigure.elasticsearch.jest.JestAutoConfiguration,\  
org.springframework.boot.autoconfigure.freemarker.FreeMarkerAutoConfiguration,\  
org.springframework.boot.autoconfigure.gson.GsonAutoConfiguration,\  
org.springframework.boot.autoconfigure.h2.H2ConsoleAutoConfiguration,\  
org.springframework.boot.autoconfigure.hateoas.HypermediaAutoConfiguration,\  
org.springframework.boot.autoconfigure.hazelcast.HazelcastAutoConfiguration,\  
org.springframework.boot.autoconfigure.hazelcast.HazelcastJpaDependencyAutoConfiguration,\  
org.springframework.boot.autoconfigure.info.ProjectInfoAutoConfiguration,\  
org.springframework.boot.autoconfigure.integration.IntegrationAutoConfiguration,\  
org.springframework.boot.autoconfigure.jackson.JacksonAutoConfiguration,\  
org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration,\  
org.springframework.boot.autoconfigure.jdbc.JdbcTemplateAutoConfiguration,\  
org.springframework.boot.autoconfigure.jdbc.JndiDataSourceAutoConfiguration,\  
org.springframework.boot.autoconfigure.jdbc.XADataSourceAutoConfiguration,\  
org.springframework.boot.autoconfigure.jdbc.DataSourceTransactionManagerAutoConfiguration,\  
org.springframework.boot.autoconfigure.jms.JmsAutoConfiguration,\  
org.springframework.boot.autoconfigure.jmx.JmxAutoConfiguration,\  
org.springframework.boot.autoconfigure.jms.JndiConnectionFactoryAutoConfiguration,\  
org.springframework.boot.autoconfigure.jms.activemq.ActiveMQAutoConfiguration,\  
org.springframework.boot.autoconfigure.jms.artemis.ArtemisAutoConfiguration,\
```

```
org.springframework.boot.autoconfigure.flyway.FlywayAutoConfiguration,\  
org.springframework.boot.autoconfigure.groovy.template.GroovyTemplateAutoConfiguration,\  
org.springframework.boot.autoconfigure.jersey.JerseyAutoConfiguration,\  
org.springframework.boot.autoconfigure.jooq.JooqAutoConfiguration,\  
org.springframework.boot.autoconfigure.kafka.KafkaAutoConfiguration,\  
org.springframework.boot.autoconfigure.ldap.embedded.EmbeddedLdapAutoConfiguration,  
on,\  
org.springframework.boot.autoconfigure.ldap.LdapAutoConfiguration,\  
org.springframework.boot.autoconfigure.liquibase.LiquibaseAutoConfiguration,\  
org.springframework.boot.autoconfigure.mail.MailSenderAutoConfiguration,\  
org.springframework.boot.autoconfigure.mail.MailSenderValidatorAutoConfiguration  
,\  
org.springframework.boot.autoconfigure.mobile.DeviceResolverAutoConfiguration,\  
org.springframework.boot.autoconfigure.mobile.DeviceDelegatingViewResolverAutoCo  
nfiguration,\  
org.springframework.boot.autoconfigure.mobile.SitePreferenceAutoConfiguration,\  
org.springframework.boot.autoconfigure.mongo.embedded.EmbeddedMongoAutoConfigura  
tion,\  
org.springframework.boot.autoconfigure.mongo.MongoAutoConfiguration,\  
org.springframework.boot.autoconfigure.mustache.MustacheAutoConfiguration,\  
org.springframework.boot.autoconfigure.orm.jpa.HibernateJpaAutoConfiguration,\  
org.springframework.boot.autoconfigure.reactor.ReactorAutoConfiguration,\  
org.springframework.boot.autoconfigure.security.SecurityAutoConfiguration,\  
org.springframework.boot.autoconfigure.security.SecurityFilterAutoConfiguration,  
\  
org.springframework.boot.autoconfigure.securityFallbackWebSecurityAutoConfigura  
tion,\  
org.springframework.boot.autoconfigure.security.oauth2.OAuth2AutoConfiguration,\  
org.springframework.boot.autoconfigure.sendgrid.SendGridAutoConfiguration,\  
org.springframework.boot.autoconfigure.session.SessionAutoConfiguration,\  
org.springframework.boot.autoconfigure.social.SocialWebAutoConfiguration,\  
org.springframework.boot.autoconfigure.social.FacebookAutoConfiguration,\  
org.springframework.boot.autoconfigure.social.LinkedInAutoConfiguration,\  
org.springframework.boot.autoconfigure.social.TwitterAutoConfiguration,\  
org.springframework.boot.autoconfigure.solr.SolrAutoConfiguration,\  
org.springframework.boot.autoconfigure.thymeleaf.ThymeleafAutoConfiguration,\  
org.springframework.boot.autoconfigure.transaction.TransactionAutoConfiguration,  
\  
org.springframework.boot.autoconfigure.transaction.jta.JtaAutoConfiguration,\  
org.springframework.boot.autoconfigure.validation.ValidationAutoConfiguration,\  
org.springframework.boot.autoconfigure.web.DispatcherServletAutoConfiguration,\  
org.springframework.boot.autoconfigure.web.EmbeddedServletContainerAutoConfigura  
tion,\  
org.springframework.boot.autoconfigure.web.ErrorMvcAutoConfiguration,\  
org.springframework.boot.autoconfigure.web.HttpEncodingAutoConfiguration,\  
org.springframework.boot.autoconfigure.web.HttpMessageConvertersAutoConfiguratio  
n,\  
org.springframework.boot.autoconfigure.web.MultipartAutoConfiguration,\  
org.springframework.boot.autoconfigure.web.ServerPropertiesAutoConfiguration,\  
org.springframework.boot.autoconfigure.web.webClientAutoConfiguration,\  
org.springframework.boot.autoconfigure.web.WebMvcAutoConfiguration,\  
org.springframework.boot.autoconfigure.websocket.WebSocketAutoConfiguration,\  
org.springframework.boot.autoconfigure.websocket.WebSocketMessagingAutoConfigura  
tion,\  
org.springframework.boot.autoconfigure.webservices.WebServicesAutoConfiguration
```

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；用他们来做自动配置；

3)、每一个自动配置类进行自动配置功能；

4)、以**HttpEncodingAutoConfiguration (Http编码自动配置)** 为例，解释自动配置原理；

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

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

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

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

@ConditionalOnProperty(prefix = "spring.http.encoding", value = "enabled",
matchIfMissing = true) //判断配置文件中是否存在某个配置
spring.http.encoding.enabled; 如果不存在，判断也是成立的
//即使我们配置文件中不配置pring.http.encoding.enabled=true，也是默认生效的；

public class HttpEncodingAutoConfiguration {

    //在上面已经和SpringBoot的配置文件映射了
    private final HttpEncodingProperties properties;

    //只有一个有参构造器的情况下，参数的值就会从容器中拿
    public HttpEncodingAutoConfiguration(HttpEncodingProperties properties) {
        this.properties = properties;
    }

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

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

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

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

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

根据当前不同的条件判断，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；

5)、所有在配置文件中能配置的属性都是在xxxxProperties类中封装者；配置文件能配置什么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");
```

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属性的值；

xxxxAutoConfigurartion：自动配置类；

给容器中添加组件

xxxxProperties:封装配置文件中相关属性；

2、细节

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

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

@Conditional扩展注解	作用（判断是否满足当前指定条件）
@ConditionalOnJava	系统的java版本是否符合要求
@ConditionalOnBean	容器中存在指定Bean；
@ConditionalOnMissingBean	容器中不存在指定Bean；
@ConditionalOnExpression	满足SpEL表达式指定
@ConditionalOnClass	系统中有指定的类
@ConditionalOnMissingClass	系统中没有指定的类
@ConditionalOnSingleCandidate	容器中只有一个指定的Bean，或者这个Bean是首选Bean
@ConditionalOnProperty	该注释中的属性且不在类中的属性

<code>@ConditionalOnProperty</code>	示例：判断当前是否满足指定属性值 作用（判断是否满足当前指定条件）
<code>@ConditionalOnResource</code>	类路径下是否存在指定资源文件
<code>@ConditionalOnWebApplication</code>	当前是web环境
<code>@ConditionalOnNotWebApplication</code>	当前不是web环境
<code>@ConditionalOnJndi</code>	JNDI存在指定项 (Java Naming and Directory Interface, Java命名和目录接口)

自动配置类必须在一定的条件下才能生效；

我们怎么知道哪些自动配置类生效；

我们可以通过启用 `debug=true` 属性；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；

```
=====
AUTO-CONFIGURATION REPORT
=====
```

Positive matches: (自动配置类启用的)

```
-----
DispatcherServletAutoConfiguration matched:
- @ConditionalOnClass found required class
'org.springframework.web.servlet.DispatcherServlet'; @ConditionalOnMissingClass
did not find unwanted class (onClassCondition)
- @ConditionalOnWebApplication (required) found StandardServletEnvironment
(OnWebApplicationCondition)
```

Negative matches: (没有启动，没有匹配成功的自动配置类)

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

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

三、日志

1、日志框架

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) SLF4j (Simple Logging Facade for Java) Jboss-logging	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 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

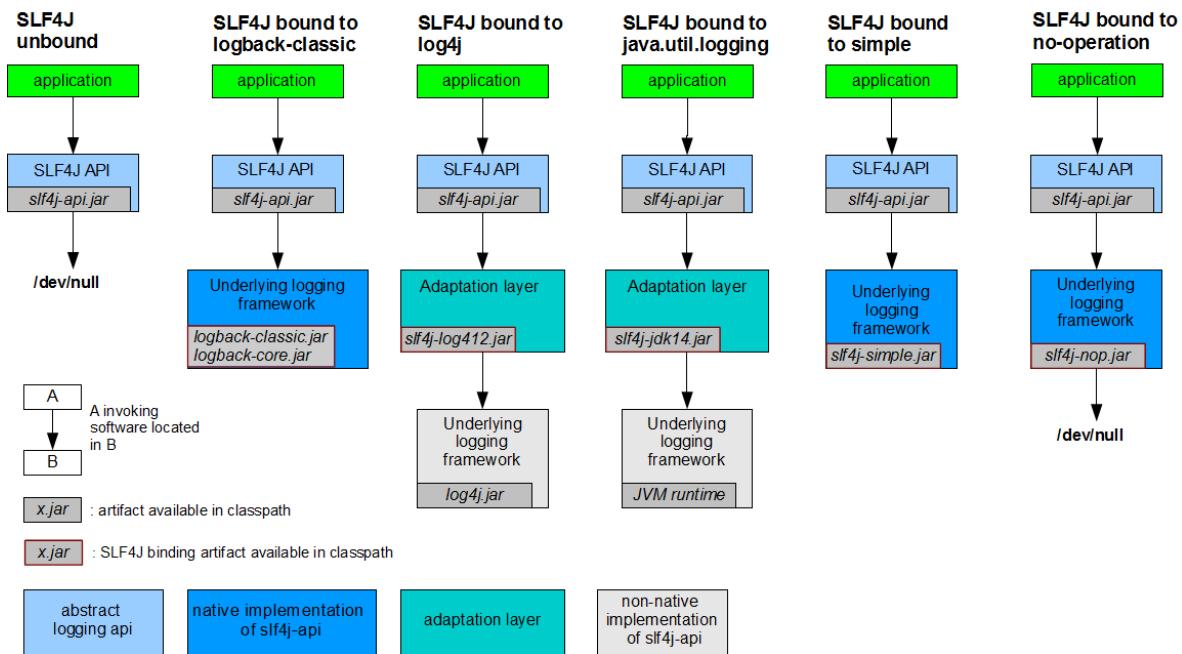
```

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");
    }
}

```

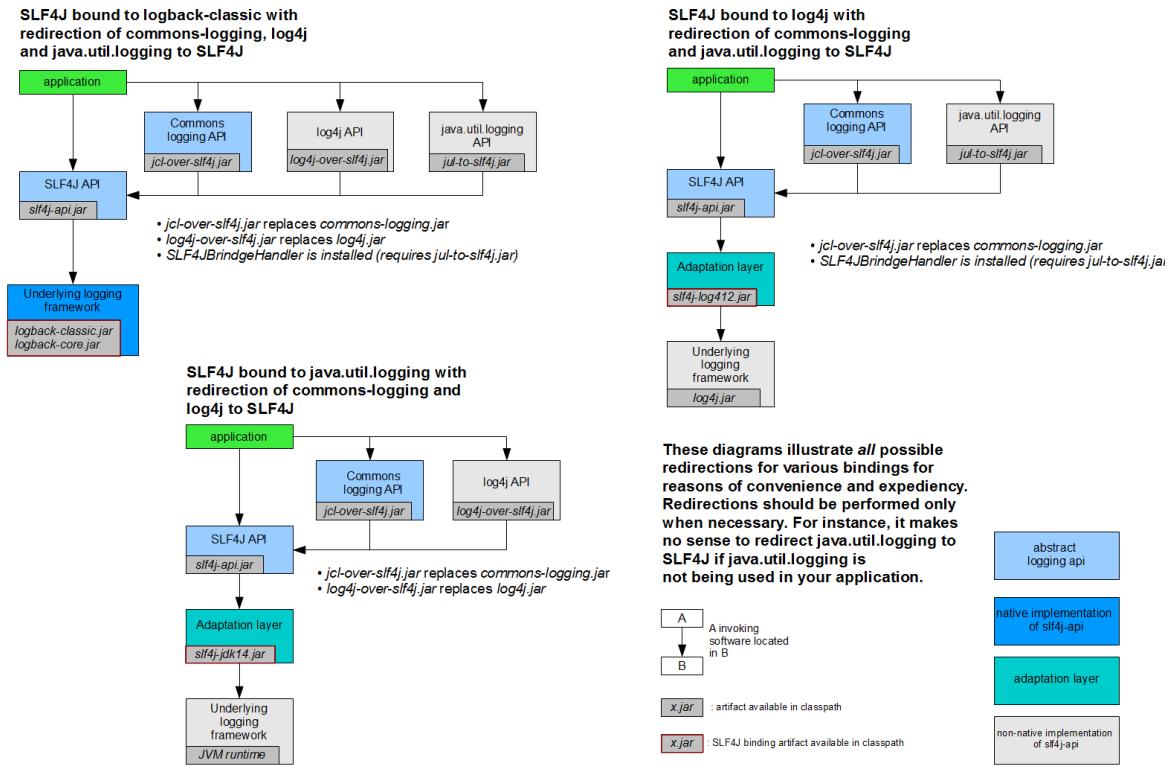
图示：



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

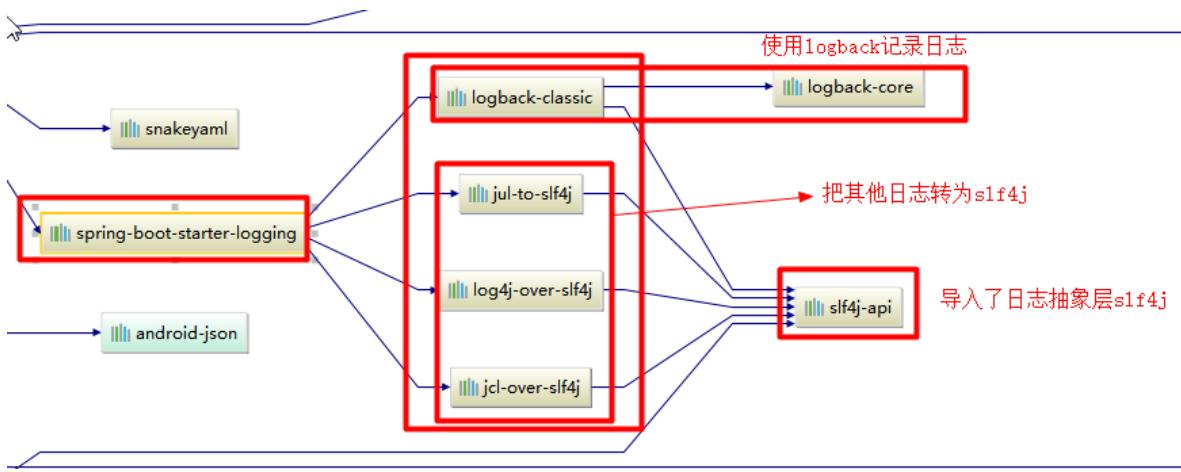
3、SpringBoot日志关系

```
<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了成了slf4j；
- 3)、中间替换包？

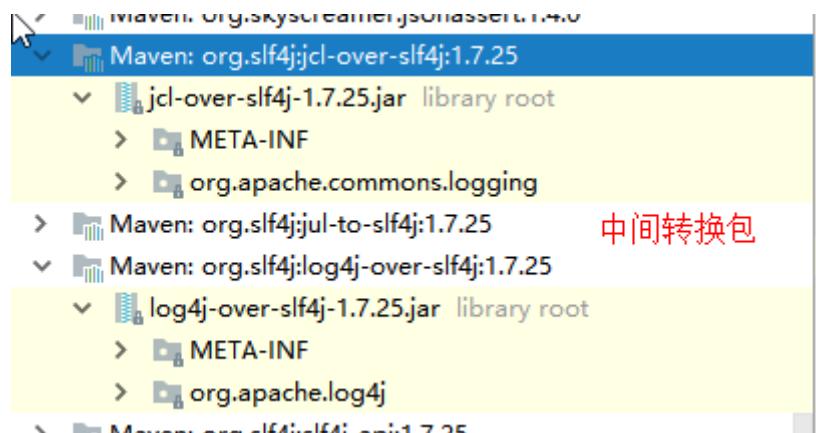
```

@SuppressWarnings("rawtypes")
public abstract class LogFactory {

    static String UNSUPPORTED_OPERATION_IN_JCL_OVER_SLF4J =
"http://www.slf4j.org/codes.html#unsupported_operation_in_jcl_over_slf4j";

    static LogFactory logFactory = new SLF4JLogFactory();
}

```



- 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；

```
<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默认帮我们配置好了日志；

```
//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
    //级别(root级别
    logger.info("这是info日志... ");
    logger.warn("这是warn日志... ");
    logger.error("这是error日志... ");
}
```

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

# 不指定路径在当前项目下生成springboot.log日志
logging.path=

# 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

给类路径下放上每个日志框架自己的配置文件即可；SpringBoot就不使用它的默认配置了

Logging System	Customization
Logback	<code>logback-spring.xml</code> , <code>logback-spring.groovy</code> , <code>logback.xml</code> or <code>logback.groovy</code>
Log4j2	<code>log4j2-spring.xml</code> or <code>log4j2.xml</code>
JDK (Java Util Logging)	<code>logging.properties</code>

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

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

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

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；

用slf4j+log4j的方式：

```
先排除logback、log4j-over-slf4j
<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>
```

再引入slf4j-log4j12的依赖

```
<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

顺序：

spring-boot-04-web-jsp
spring-boot-04-web-restfulcrud
spring-boot-04-web-jsp

1、简介

使用SpringBoot；

- 1) 、创建SpringBoot应用，选中我们需要的模块；
- 2) 、SpringBoot已经默认将这些场景配置好了，只需要在配置文件中指定少量配置就可以运行起来
- 3) 、自己编写业务代码；

自动配置原理？

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

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
```

```
WebMvcAuotConfiguration:

@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.addHandle...
```

```
        registry.addResourceHandler("/webjars/**")
            .addResourceLocations(
                "classpath:/META-
INF/resources/webjars/")
            .setCachePeriod(cachePeriod));
    }

    String staticPathPattern =
this.mvcProperties.getStaticPathPattern();

// 静态资源文件夹映射
if (!registry.hasMappingForPattern(staticPathPattern)) {
    customizeResourceHandlerRegistration(
        registry.addResourceHandler(staticPathPattern)
            .addResourceLocations(
                this.resourceProperties.getStaticLocations())
            .setCachePeriod(cachePeriod));
}
}

// 配置欢迎页映射
@Bean
public WelcomePageHandlerMapping welcomePageHandlerMapping(
    ResourceProperties resourceProperties) {

    return new
WelcomePageHandlerMapping(resourceProperties.getwelcomePage(),
    this.mvcProperties.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 **/favicon.ico
        mapping.setUrlMap(Collections.singletonMap("**/favicon.ico",
            faviconRequestHandler()));
        return mapping;
    }

    @Bean
    public ResourceHttpRequestHandler faviconRequestHandler() {
```

```

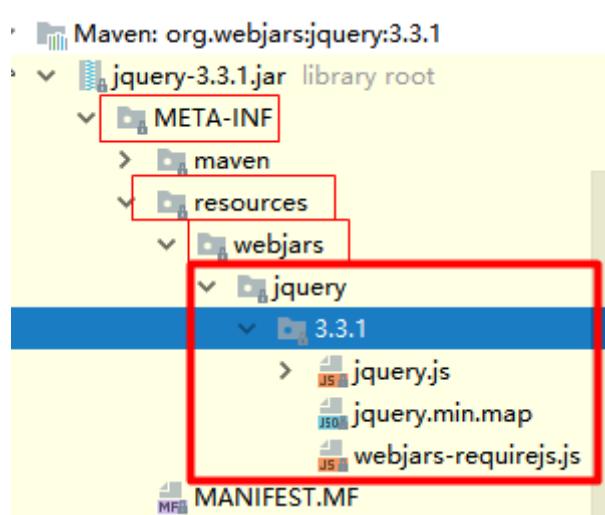
        ResourceHttpRequestHandler requestHandler = new
ResourceHttpRequestHandler();
            requestHandler
.setLocations(this.resourceProperties.getFaviconLocations());
            return requestHandler;
        }
    }
}

```

1)、所有 /webjars/** , 都去 classpath:/META-INF/resources/webjars/ 找资源;

webjars: 以jar包的方式引入静态资源;

<http://www.webjars.org/>



<localhost:8080/webjars/jquery/3.3.1/jquery.js>

```

<!--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) 找映射

```

"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

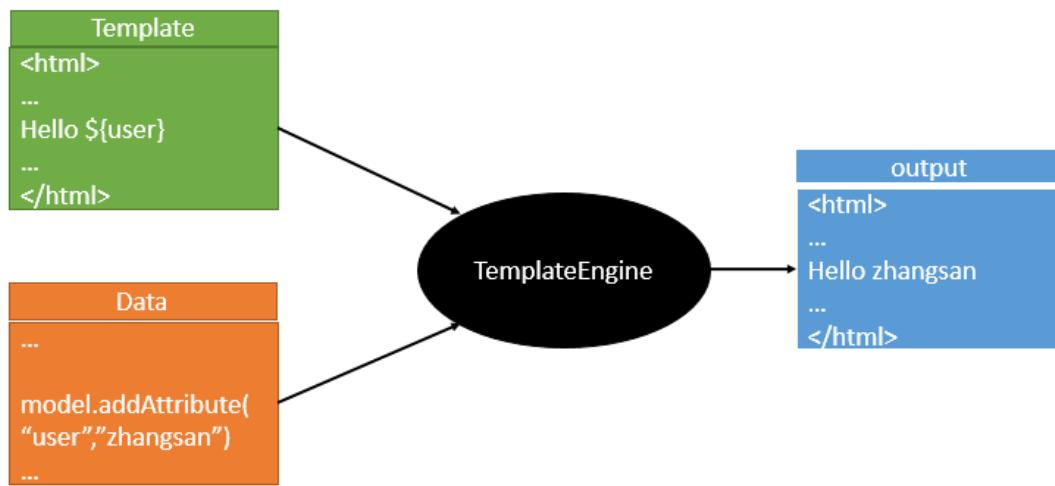
3)、欢迎页； 静态资源文件夹下的所有index.html页面； 被“/**”映射；

localhost:8080/ 找index页面

4)、所有的 “/favicon.ico” 都是在静态资源文件夹下找；

3、模板引擎

JSP、Velocity、Freemarker、Thymeleaf



SpringBoot推荐的Thymeleaf；

语法更简单，功能更强大；

Spring Boot推荐使用Thymeleaf、Freemarker等后现代的模板引擎技术；

一旦导入相关依赖，会自动配置ThymeleafAutoConfiguration、FreeMarkerAutoConfiguration。

基本语法

• 表达式：

- #{}：当前对象/变量取值，国际化消息

- \${}：变量取值

- *{}：当前对象/变量取值

- @{}：url表达式

- ~{...}: 片段引用

- 内置对象/共用对象:

• 判断/遍历:

- th:if
- th:unless
- th:each
- th:switch、th:case

• th:属性

1、引入thymeleaf;

```
<!-- 先引入thymeleaf-->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-thymeleaf</artifactId>
</dependency>

<!-- 然后切换thymeleaf版本-->
<properties>

    <!-- 用下面这个会有版本冲突-->
    <!-- <thymeleaf.version>3.0.9.RELEASE</thymeleaf.version>-->
    <thymeleaf-spring5.version>3.0.9.RELEASE</thymeleaf-spring5.version>

    <!-- 布局功能的支持程序 -->
    <!-- thymeleaf3主程序，要layout2以上版本 -->
    <thymeleaf-layout-dialect.version>2.2.2</thymeleaf-layout-
dialect.version>

</properties>
```

2、Thymeleaf使用

```
@ConfigurationProperties(prefix = "spring.thymeleaf")
public class ThymeleafProperties {

    private static final Charset DEFAULT_ENCODING = Charset.forName("UTF-8");

    private static final MediaType DEFAULT_CONTENT_TYPE =
    MediaType.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就能自动渲染;

使用:

1、在html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

`th:` 任意html属性来替换原生属性的值

Order	Feature		Attributes
1	Fragment inclusion	片段包含: jsp:include	th:insert th:replace
2	Fragment iteration	遍历: c:forEach	th:each
3	Conditional evaluation	条件判断: c:if	th:if th:unless th:switch th:case
4	Local variable definition	声明变量: c:set	th:object th:with
5	General attribute modification	任意属性修改 支持prepend , append	th:attr th:attrprepend th:attrappend
6	Specific attribute modification	修改指定属性默认值	th:value th:href th:src ... th:text th:utext
7	Text (tag body modification)	修改标签体内容	转义特殊字符 不转义特殊字符
8	Fragment specification	声明片段	th:fragment
9	Fragment removal		th:remove

2) 表达式?

Simple expressions: (表达式语法)

`${...}` : 强调值, 相当于静态变量
 `# {...}` : 强调对象, 相当于变量

Variable Expressions: `${...}` : 计算variable值;

对象导航图语言 (Object Graph Navigation Language), 简称OGNL, 是应用于Java中的一个开源的表达式语言 (Expression Language)

1)、获取对象的属性、调用方法

2)、使用内置的基本对象: `${session.foo}`

```

#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.

```

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: 在功能上和\${}是一样的;

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variable

Link URL Expressions: @{...}: 连接URL
@{/order/process(execId=\${execId}, execType='FAST')}

Fragment Expressions: ~{...}: 引用片段
<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>

1. 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`: SpringMVC用来转换Http请求和响应的；User---Json；
 - `HttpMessageConverters` 是从容器中确定；获取所有的`HttpMessageConverter`；
自己给容器中添加`HttpMessageConverter`, 只需要将自己的组件注册容器中
(@Bean,@Component)
- Automatic registration of `MessageCodesResolver` (see below). 定义错误代码生成规则
- Automatic use of a `ConfigurableWebBindingInitializer` bean (see below).
我们可以配置一个`ConfigurableWebBindingInitializer`来替换默认的；（添加到容器）

初始化`WebDataBinder`；
请求数据=====JavaBean；

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

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 `ExceptionHandlerExceptionResolver` 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`.

2、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；

```

// 使用WebMvcConfigurerAdapter (springboot2.0:webMvcConfigurer) 可以来扩展SpringMVC
的功能

@Configuration // 这是一个配置类

/*springboot1.0:public class MyMvcConfig extends WebMvcConfigurerAdapter {*/
public class MyMvcConfig implements WebMvcConfigurer {

    // 添加视图控制器，用ViewControllerRegistry
    @Override
    public void addViewControllers(ViewControllerRegistry registry) {
        // super.addViewControllers(registry);

        // 浏览器，发送/feuoy请求，那么是来到success
        // 访问http://localhost:8080/crud/feuoy，URL不变但用的是success的页面
        registry.addViewController("/feuoy").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；

```
@EnableWebMvc

@Configuration
public class MyMvcConfig implements WebMvcConfigurer {
    @Override
    public void addViewControllers(ViewControllerRegistry registry) {
        registry.addViewController("/feuoy").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
public class DelegatingWebMvcConfiguration extends WebMvcConfigurationSupport {
```

3)、

```
@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 {
```

- 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

6、RestfulCRUD

1)、默认访问首页

```
// 使用WebMvcConfigurerAdapter (springboot2.0:webMvcConfigurer) 可以来扩展SpringMVC  
的功能

// @EnableWebMvc    注释掉，我们不要接管SpringMVC

@Configuration // 这是一个配置类

/*springboot1.0:public class MyMvcConfig extends WebMvcConfigurerAdapter {*/
public class MyMvcConfig implements WebMvcConfigurer {

    // 添加视图控制器，用ViewControllerRegistry
    @Override
    public void addViewControllers(ViewControllerRegistry registry) {
        // super.addViewControllers(registry);

        // 浏览器，发送/feuoy请求，那么是来到success
        // 访问http://localhost:8080/crud/feuoy，URL不变但用的是success的页面
        registry.addViewController("/feuoy").setViewName("success");
    }

    // 所有的WebMvcConfigurerAdapter (springboot2.0:WebMvcConfigurer) 组件都会一起起
    // 作用

    //一定要，将组件注册在容器
    @Bean

    /*springboot1.0:public WebMvcConfigurerAdapter webMvcConfigurerAdapter() {
        WebMvcConfigurerAdapter adapter = new WebMvcConfigurerAdapter() {*/

```

```

public WebMvcConfigurer webMvcConfigurer() {
    WebMvcConfigurer adapter = new WebMvcConfigurer() {

        @Override
        public void addViewControllers(ViewControllerRegistry registry) {

            // 访问http://localhost:8080/crud/, 页面为login
            registry.addViewController("/").setViewName("login");

            // 访问http://localhost:8080/crud/, 页面为login
            registry.addViewController("/index.html").setViewName("login");

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

    return adapter;
}

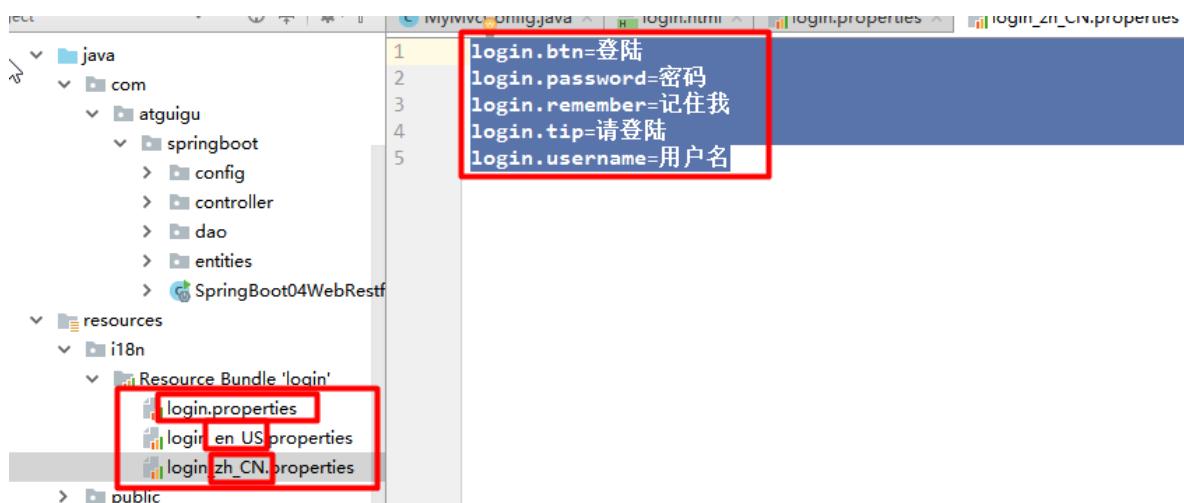
```

2) 、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



- 2) 、SpringBoot自动配置好了管理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.setBasename(StringUtils.commaDelimitedListToStringArray(
        StringUtils.trimAllWhitespace(this.basename)));
    }

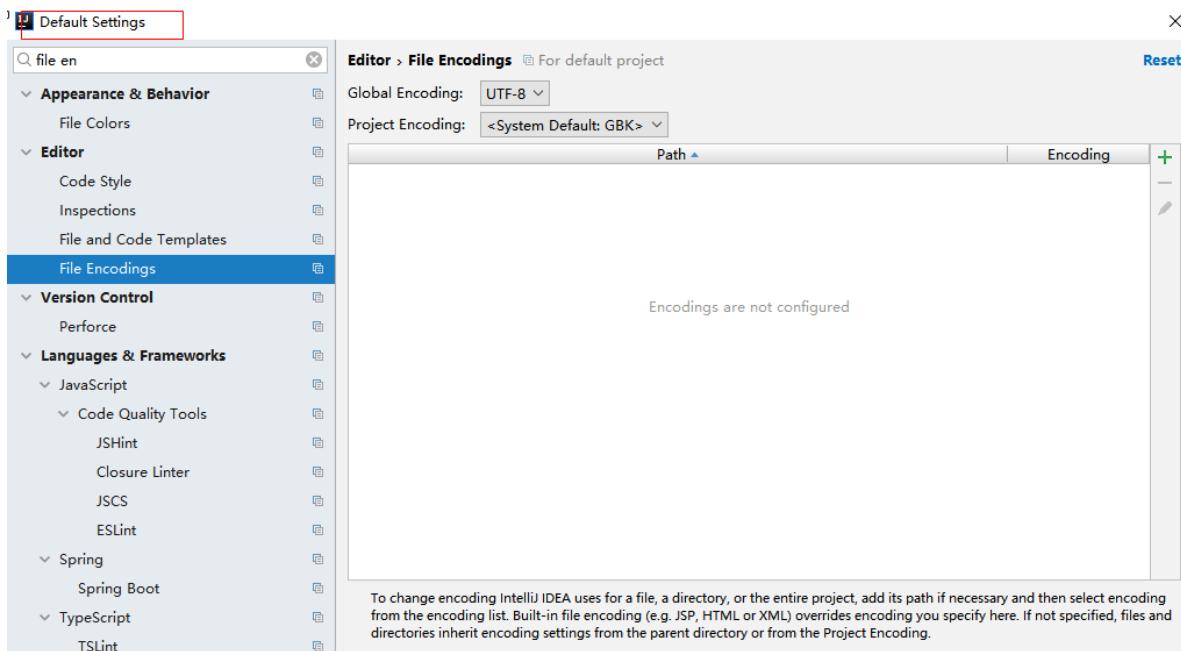
    if (this.encoding != null) {
        messageSource.setDefaultEncoding(this.encoding.name());
    }

    messageSource.setFallbackToSystemLocale(this.fallbackToSystemLocale);
    messageSource.setCacheSeconds(this.cacheSeconds);
    messageSource.setAlwaysUseMessageFormat(this.alwaysUseMessageFormat);

    return messageSource;
}

```

3)、去页面获取国际化的值；



```
<!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>

        <!--项目下/assets的bootstrap-->
        <!--<link href="assets/css/bootstrap.min.css" rel="stylesheet">-->
        <!--不知道为什么之前不能用配置包下的。现在又可以了 -->
        <!--这个是配置包下的: org\webjars\bootstrap\4.0.0\bootstrap-4.0.0.jar!\META-INF\resources\webjars\bootstrap\4.0.0\css\bootstrap.css-->
        <!--模板引擎跳转th:href=      取路径
@{/webjars/bootstrap/4.0.0/css/bootstrap.css}-->
        <!-- css用href取外部css -->
        <link href="assets/css/bootstrap.min.css"
th:href="@{/webjars/bootstrap/4.0.0/css/bootstrap.css}" rel="stylesheet">

        <!-- Custom styles for this template -->
        <!--为开发者可以自定义的css路径-->
        <link href="assets/css/signin.css" th:href="@{/assets/css/signin.css}"
rel="stylesheet">

    </head>

    <body class="text-center">

        <!--th:action=到@{/user/login}来验证登录-->
        <!--method="post"方式-->
        <form class="form-signin" action="dashboard.html" th:action="@{/user/login}"
method="post">

            <!--th:src=""来取图片更方便,      @{/assets/img/bootstrap-solid.svg}根路径
是/static等-->
            <!--alt=""用空来代替图片-->
            
```

```

<!--th:text=来写内容取代<h1></h1>里的内容    #{login.tip}来取值-->
<h1 class="h3 mb-3 font-weight-normal" th:text="#{login.tip}">Please sign
in</h1>

<!--${msg}提示语直接用强调“值”-->
<!--th:if=判断显示与否      ${not #strings.isEmpty(msg)}来: 算{非 取strings.(是
空? )}-->
<p style="color: red" th:text="${msg}" th:if="${not #strings.isEmpty(msg)}">
</p>

<!--#${login.username}有变量的用强调“对象”-->
<label class="sr-only" th:text="#{login.username}">Username</label>

<!--th:placeholder="#{login.username}"输入提示-->
<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="">

<!--[[#{login.remember}]], 直接写的行内写法要加两层中括号-->
<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>

<!--跳转, 代表首页, 设置(l='zh_CN')-->
<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>

```

效果：根据浏览器语言设置的信息切换了国际化；

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() ==  
    WebMvcProperties.LocaleResolver.FIXED) {  
        return new FixedLocaleResolver(this.mvcProperties.getLocale());  
    }  
  
    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

```
// 通过实现LocaleResolver，可以在链接上携带区域信息  
  
public class MyLocaleResolver implements LocaleResolver {  
  
    // 处理Locale问题  
    @Override  
    public Locale resolveLocale(HttpServletRequest request) {  
  
        // 从request里面拿，语言选项参数l  
        String l = request.getParameter("l");  
  
        /* 按照源码中的参数需要，填进去就可以了  
        public Locale(String language, String country) {  
            this(language, country, "");  
        }  
        */  
  
        // 先拿到默认的区域信息  
        Locale locale = Locale.getDefault();  
  
        // request的l不为空  
        if (!StringUtils.isEmpty(l)) {  
  
            // 会用"_"来做分割  
            // String[] 来存  
            String[] split = l.split("_");  
  
            // 填进去
```

```

        locale = new Locale(split[0], split[1]);
    }

    return locale;
}

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

// 构造函数
// 无参返回一个这个类对象
@Bean
public LocaleResolver localeResolver(){
    return new MyLocaleResolver();
}

}

```

3) 、登陆

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

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

4) 、拦截器进行登陆检查

拦截器

```
// 拦截器，登陆检查，通过实现HandlerInterceptor

public class LoginHandlerInterceptor implements HandlerInterceptor {

    // 目标方法执行之前
    // request, response, handler
    @Override
    public boolean preHandle(
        HttpServletRequest request,
        HttpServletResponse response,
        Object handler)
```

```

    throws Exception {

        // Object一个user
        // 是request请求, session里面设置的loginUser
        Object user = request.getSession().getAttribute("loginUser");

        // 如果null
        if (user == null) {

            // 用request, 放上消息提示msg
            request.setAttribute("msg", "没有权限, 请先登录");

            // 获取请求转发器, 转发器的指向通过getRequestDispatcher()的参数
            // (暂时这样理解) 因为带了数据, 那么用转发
            // 转发到/index.html (这个请求会转到login.html), 带上request和response
            request.getRequestDispatcher("/index.html").forward(request,
response);

            // 返回false, 这个目标方法执行不能放行
            return false;
        } else {

            // 已经登录, 放行请求
            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

```

// 所有的WebMvcConfigurerAdapter (springboot2.0:WebMvcConfigurer) 组件都会一起起作用

//一定要, 将组件注册在容器
@Bean

/*springboot1.0:public WebMvcConfigurerAdapter webMvcConfigurerAdapter() {
    WebMvcConfigurerAdapter adapter = new WebMvcConfigurerAdapter() {*/
public WebMvcConfigurer webMvcConfigurer() {
    WebMvcConfigurer adapter = new WebMvcConfigurer() {

        @Override
        public void addViewControllers(ViewControllerRegistry registry) {

            // 访问http://localhost:8080/crud/, 页面为login

```

```

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

        // 访问http://localhost:8080/crud/，页面为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);

        // 关于静态资源: *.css , *.js.... SpringBoot已经做好了静态资源映射

        // addPathPatterns("/**"): 添加任意路径下的所有请求
        // excludePathPatterns("/index.html","/","/user/login"): 除了index
        和/user/login

        // 在这里注释掉拦截器
        registry.addInterceptor(new
LoginHandlerInterceptor()).addPathPatterns("/**")
            .excludePathPatterns("/index.html","/","/user/login");
    }

    return adapter;
}

```

5) 、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

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：

```

commons/bar.html:
<!--sidebar-->
<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>
            ...
        </ul>
    </div>
</nav>

list.html:
<!--引入侧边栏commons/bar::#sidebar      传入参数(activeUri='emps')-->
<div th:replace="commons/bar::#sidebar(activeUri='emps')"></div>

```

6) 、CRUD-员工添加

添加页面

```
<!--这是一个单纯添加表单-->

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

    <!--email-->
    <div class="form-group">
        <label>Email</label>
        <input type="email" class="form-control"
placeholder="zhangsan@atguigu.com">
    </div>

    <!--gender-->
    <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="form-check form-check-inline">
            <input class="form-check-input" type="radio" name="gender"
value="0">
            <label class="form-check-label">女</label>
        </div>
    </div>

    <!--Department-->
    <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>

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

    <!--添加按钮-->
    <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；

7) 、CRUD-员工修改

修改添加二合一表单（原来的添加页面）

```
<!--这是一个员工添加和修改的二合一表单-->
<!--需要区分是员工修改还是添加！-->
<form th:action="@{/emp}" method="post">

    <!--
    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"
    4、value="put"值就是我们指定的请求方式
    -->

    <!--_method-->
    <!--如果emp存在，那么是修改，value是put方式-->
    <!--如果不存在，这行不起作用-->
    <input type="hidden" name="_method" value="put" th:if="${emp!=null}"/>

    <!--id-->
    <!--如果emp存在，那么是修改，值是这个员工的ID: ${emp.id}-->
    <!--如果不存在，这行不起作用，数据库id主键设置了自增-->
    <!--而且这是一个不用填的隐藏字段-->
    <input type="hidden" name="id" th:value="${emp.id}" th:if="${emp!=null}"/>

    <!--lastName-->
    <!--简化写法，三目运算符， th:value="${emp!=null}?${emp.lastName}"-->
    <!--如果emp存在，那么值是这个员工的lastName: ${emp.lastName}-->
    <!--如果不存在，值是输入的，不用在这里写value-->
    <div class="form-group">
        <label>LastName</label>
        <input name="lastName" type="text" class="form-control"
placeholder="zhangan"
            th:value="${emp!=null}?${emp.lastName}">
    </div>
```

```

<!--email-->


<label>Email</label>
    <input name="email" type="email" class="form-control"
placeholder="zhangsan@atguigu.com"
        th:value="${emp!=null}?${emp.email}">



<!--gender-->
<!--1代表男， 0代表女-->
<!--如果emp存在，用${emp.gender==1}来判断布尔值，如果是true，那么
th:checked="true"这个选项也就被选上-->
<!--如果不存在，值是让用户选择的，不用在这里写value-->


<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>



<!--department.id-->
<!--提交的是部门的id-->
<!--th:each="dept:${depts}"      遍历传入的depts，每个的名字是dept-->
<!--th:value="${dept.id}"        意义值是每个dept.id-->
<!--th:text="${dept.departmentName}"    显示值是每个dept.departmentName-->
<!--th:selected="${emp!=null}?${dept.id == emp.department.id}"      如果emp存
在，那么是修改的情况，判断这个员工的部门id，是不是这个dept.id选项的，是的话这个选项被选上-->


<label>department</label>
    <select class="form-control" name="department.id">
        <option th:each="dept:${depts}"
            th:value="${dept.id}"
            th:text="${dept.departmentName}"
            th:selected="${emp!=null}?${dept.id == emp.department.id}">
            1 (这个1不显示，被上面的th:text覆盖)
        </option>
    </select>



<!--birth-->
<!--${#dates.format(emp.birth, 'yyyy-MM-dd HH:mm')}格式化-->


<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')}">


```

```

<!--如果emp存在，那么显示修改-->
<button type="submit" class="btn btn-primary" th:text="${emp!=null?'修改':'添加'}>添加</button>
</form>

```

8) 、CRUD-员工删除

```

<!--表格内容-->
<tbody>
    <!--th:each= 遍历-->
    <!--"emp:${emps}" emp做emps的取值-->
    <tr th:each="emp:${emps}">
        <!--标签内-->
        <!--${emp.id}来取-->
        <td th:text="${emp.id}"></td>
        <!--行内-->
        <td>[${emp.lastName}]</td>
        <td th:text="${emp.email}"></td>
        <!--${}这里面只用来取值，要用来做逻辑判断或者运算的话要在外面-->
        <td th:text="${emp.gender}==0?'女':'男'"></td>
        <!--取两层-->
        <td th:text="${emp.department.departmentName}"></td>
        <!--${}内，再取#-->
        <!--#dates 可以拿到Dates的一个对象（类）-->
        <!--format(emp.birth, 'yyyy-MM-dd HH:mm')方法 来格式化-->
        <td th:text="${#dates.format(emp.birth, 'yyyy-MM-dd HH:mm')}"></td>
        <!--td里面再放标签-->
        <td>
            <!--跳转性质的按钮，可以直接用a标签，做样式-->
            <!--th:href="@{/emp/}+${emp.id}" @{}取路径]加${}取这个员工的id]，拼给
th:href-->
            <a class="btn btn-sm btn-primary" th:href="@{/emp/}+${emp.id}">编辑
</a>

            <!--删除有三步：-->
            <!--第一步-->
            <!--th:attr 把del_uri=@{/emp/}+${emp.id}作为属性存起来-->
            <!--行为性质的按钮，用button标签-->
            <button th:attr="del_uri=@{/emp/}+${emp.id}" class="btn btn-sm btn-
danger deleteBtn">删除
            </button>
        </td>

    </tr>
</tbody>

<script>
    // 第二步
    $(".deleteBtn").click(function () {
        // 给id为deleteEmpForm的元素设置action属性和del_uri值，提交动作submit()也传过去
    })
</script>

```

```

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

<!--第三步-->
<!--用一个隐藏的表单，来做删除-->
<!--action="@{/emp/}+${emp.id}"已经收到了-->
<!--注意method="post"      name="_method"      value="delete"-->
<form id="deleteEmpForm" method="post">
    <input type="hidden" name="_method" value="delete"/>
</form>

```

7、错误处理机制

1)、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

Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

Mon Feb 26 17:33:50 GMT+08:00 2018
 There was an unexpected error (type=Not Found, status=404).
 No message available

浏览器发送请求的请求头：

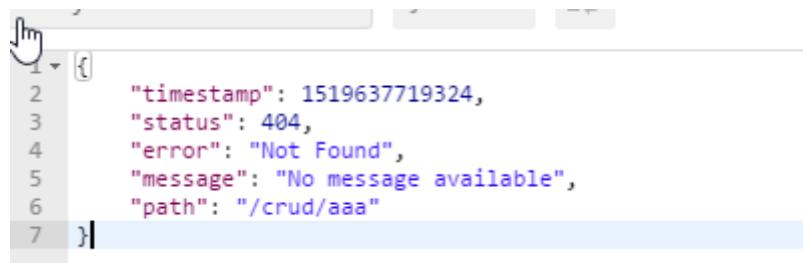
Request Headers view source

```

Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US,zh-CN;q=0.8,zh;q=0.6,en;q=0.4
Cache-Control: no-cache
Connection: keep-alive

```

- 2)、如果是其他客户端，默认响应一个json数据



```

1
2     "timestamp": 1519637719324,
3     "status": 404,
4     "error": "Not Found",
5     "message": "No message available",
6     "path": "/crud/aaa"
7 }

```

Request Headers:

```
cache-control: "no-cache"
postman-token: "b34bebc4-07a5-4c20-8f3f-952f3daec38f"
user-agent: "PostmanRuntime/7.1.1"
accept: "*/*" *
host: "localhost:8080"
cookie: "JSESSIONID=DDB37833549894367D63323D1F21957C; JSESSIONID=1BBFE9718FD60
accept-encoding: "gzip, deflate"
```

原理:

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:

```
// 帮我们在页面共享信息;

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

    Map<String, Object> errorAttributes = new LinkedHashMap<String, Object>
();
    errorAttributes.put("timestamp", new Date());
    addStatus(errorAttributes, requestAttributes);

    addErrorDetails(errorAttributes, requestAttributes, includeStackTrace);
    addPath(errorAttributes, requestAttributes);

    return errorAttributes;
}
```

2、BasicErrorController: 处理默认/error请求

```
@Controller

@RequestMapping("${server.error.path:${error.path:/error}}")
public class BasicErrorController extends AbstractErrorController {

    @RequestMapping(produces = "text/html") //产生html类型的数据; 浏览器发送的请求来到
    这个方法处理
    public ModelAndView errorHtml(HttpServletRequest request,
        HttpServletResponse response) {

        HttpStatus status = getStatus(request);
        Map<String, Object> model =
        Collections.unmodifiableMap(getErrorAttributes(
            request, isIncludeStackTrace(request, MediaType.TEXT_HTML)));
        response.setStatus(status.value());
    }
}
```

```

    // 去哪个页面作为错误页面; 包含页面地址和页面内容
    ModelAndView modelAndView = resolveErrorView(request, response, status,
model);

    return (modelAndView == null ? new ModelAndView("error", model) :
modelAndView);
}

@RequestMapping
@ResponseBody //产生json数据, 其他客户端来到这个方法处理;
public ResponseEntity<Map<String, Object>> error(HttpServletRequest request)
{

    Map<String, Object> body = getErrorAttributes(request,
        isIncludeStackTrace(request, MediaType.ALL));
    HttpStatus status = getStatus(request);

    return new ResponseEntity<Map<String, Object>>(body, status);
}

```

3、ErrorPageCustomizer:

```

@Value("${error.path:/error}")
private String path = "/error";
// 系统出现错误以来到error请求进行处理; (web.xml注册的错误页面规则)

```

4、DefaultErrorViewResolver:

```

@Override
public ModelAndView resolveErrorView(HttpServletRequest request, HttpStatus
status,
        Map<String, Object> model) {

    ModelAndView modelAndView = resolve(String.valueOf(status), model);

    if (modelAndView == null && SERIES_VIEWS.containsKey(status.series())) {
        modelAndView = resolve(SERIES_VIEWS.get(status.series()), model);
    }

    return modelAndView;
}

private ModelAndView resolve(String viewName, Map<String, Object> model) {

    //默认SpringBoot可以去找到一个页面?   error/404
    String errorViewName = "error/" + viewName;

    //模板引擎可以解析这个页面地址就用模板引擎解析
    TemplateAvailabilityProvider provider =
this.templateAvailabilityProviders
        .getProvider(errorViewName, this.applicationContext);

    if (provider != null) {

```

```

        //模板引擎可用的情况下返回到errorViewName指定的视图地址
        return new ModelAndView(errorViewName, model);
    }

    //模板引擎不可用，就在静态资源文件夹下找errorViewName对应的页面 error/404.html
    return resolveResource(errorViewName, model);
}

```

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**BasicErrorHandler**处理；
 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) 、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：

templates/error/状态码.html;

精确优先（优先寻找精确的状态码.html）（404.html）；

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，

页面能获取的信息：

timestamp: 时间戳
status: 状态码
error: 错误提示
exception: 异常对象
message: 异常消息
errors: JSR303数据校验的错误都在这里

- 2)、没有模板引擎（模板引擎找不到这个错误页面），静态资源文件夹下找；
- 3)、以上都没有错误页面，就是默认来到SpringBoot默认的错误提示页面；

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；

```
// 控制器建议注解
@ControllerAdvice

public class MyExceptionHandler {

    /*1、
    浏览器、客户端返回的都是json
    没有自适应效果*/

    @ResponseBody
    @ExceptionHandler(UserNotExistException.class)

    public Map<String, Object> handleException(Exception e){

        Map<String, Object> map = new HashMap<>();

        map.put("code", "user.notexist");
        map.put("message", e.getMessage());

        return map;
    }
}
```

- 2)、转发到/error进行自适应响应效果处理

```
/*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){

    // 用HashMap放信息
    Map<String, Object> map = new HashMap<>();
```

```

// request中带的状态码
// Integer statusCode = (Integer)
request.getAttribute("javax.servlet.error.status_code");

// 传入我们自己的错误状态码 4xx 5xx
request.setAttribute("javax.servlet.error.status_code", 500);

// 其它信息
map.put("code", "user.notexist");
map.put("message", "用户出错啦");

// 放进request
request.setAttribute("ext", map);

// 因为携带了消息，用转发
// 转发到/error页面
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

```

// 给容器中加入我们自己定义的ErrorAttributes组件
@Component

public class MyErrorAttributes extends DefaultErrorAttributes {

    /*springboot1.5.10@Override
    public Map<String, Object> getErrorAttributes(RequestAttributes
requestAttributes, boolean includeStackTrace) {
        Map<String, Object> map = super.getErrorAttributes(requestAttributes,
includeStackTrace);
        map.put("company", "atguigu");
        Map<String, Object> ext = (Map<String, Object>)
requestAttributes.getAttribute("ext", 0);
        map.put("ext", ext);
        return map;
    }*/

```

```

/*springboot2.1.8*/
@Override
public Map<String, Object> getErrorAttributes(WebRequest webRequest, boolean
includeStackTrace) {

    /* 先放继承的DefaultErrorAttributes的信息，包括
     * <ul>
     *   <li>timestamp - The time that the errors were extracted</li>
     *   <li>status - The status code</li>
     *   <li>error - The error reason</li>
     *   <li>exception - The class name of the root exception (if configured)
    </li>
     *   <li>message - The exception message</li>
     *   <li>errors - Any {@link ObjectError}s from a {@link BindingResult}
exception
     *   <li>trace - The exception stack trace</li>
     *   <li>path - The URL path when the exception was raised</li>
     * </ul>
     */
    Map<String, Object> map = super.getErrorAttributes(webRequest,
includeStackTrace);

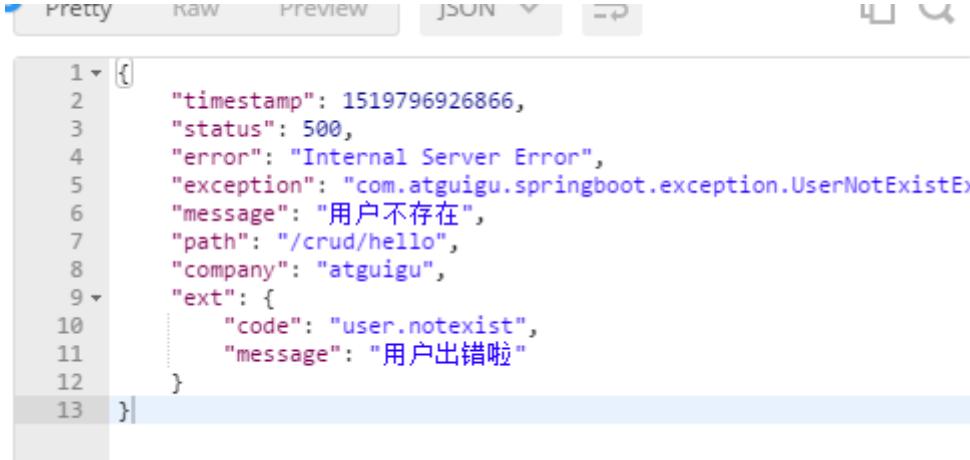
    // 再加入我们自己的map信息
    // 自定义公司信息company
    map.put("company", "feuoy");

    // 我们的异常处理器携带的数据ext
    Map<String, Object> ext = (Map<String, Object>)
webRequest.getAttribute("ext", 0);
    map.put("ext", ext);

    // 返回值的map就是浏览器页面和json能获取的所有字段
    return map;
}
}

```

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



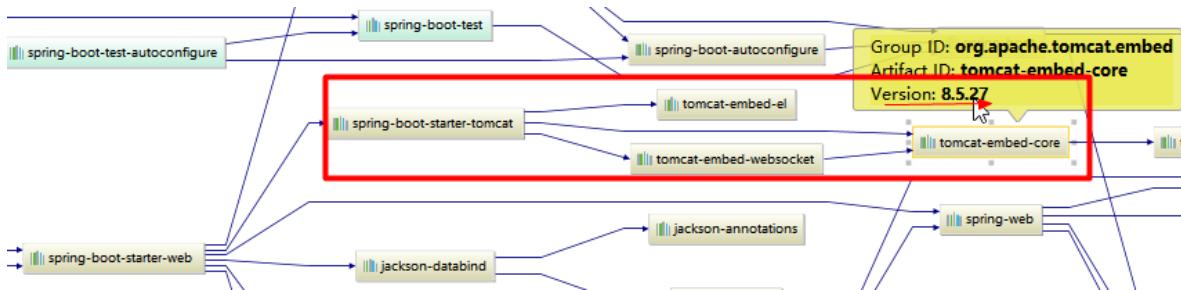
```

1  {
2      "timestamp": 1519796926866,
3      "status": 500,
4      "error": "Internal Server Error",
5      "exception": "com.atguigu.springboot.exception.UserNotExistEx
6      "message": "用户不存在",
7      "path": "/crud/hello",
8      "company": "atguigu",
9      "ext": {
10          ...
11          "code": "user.notexist",
12          "message": "用户出错啦"
13      }
14  }

```

8、配置嵌入式Servlet容器

SpringBoot默认使用Tomcat作为嵌入式的Servlet容器；



问题？

1)、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有关的配置
#server.port=8081

#springboot1.0
#server.context-path=/crud
#springboot2.0
server.servlet.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

```
/*springboot1.5.10
@Bean
public EmbeddedServletContainerCustomizer
embeddedServletContainerCustomizer(){
    return new EmbeddedServletContainerCustomizer() {
        @Override
        public void customize(ConfigurableEmbeddedServletContainer
container) {
            container.setPort(8083);
        }
    };
}*/
```

```

// springboot2.1.8
// 配置嵌入式的Servlet容器
@Bean //一定要将这个定制器加入到容器中
public WebServerFactoryCustomizer<ConfigurableWebServerFactory>
embeddedServletContainerCustomizer() {

    return new WebServerFactoryCustomizer<ConfigurableWebServerFactory>() {

        // 定制嵌入式的Servlet容器相关的规则
        @Override
        public void customize(ConfigurableWebServerFactory factory) {
            factory.setPort(8080);
        }

    };
}

```

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

MyServerConfig.java

```

@Configuration
public class MyServerConfig {

    // 注册三大组件
    // myServlet
    @Bean
    public ServletRegistrationBean myServlet() {

        ServletRegistrationBean registrationBean = new
ServletRegistrationBean(new MyServlet(), "/myServlet");
        registrationBean.setLoadOnStartup(1);

        return registrationBean;
    }

    // myFilter
    @Bean
    public FilterRegistrationBean myFilter() {

        FilterRegistrationBean registrationBean = new FilterRegistrationBean();

        registrationBean.setFilter(new MyFilter());
        registrationBean.setUrlPatterns(Arrays.asList("/hello", "/myServlet"));

    }
}

```

```

        return registrationBean;
    }

// myListener
@Bean
public ServletListenerRegistrationBean myListener() {

    ServletListenerRegistrationBean<MyListener> registrationBean = new
ServletListenerRegistrationBean<>(new MyListener());
}

return registrationBean;
}

```

ServletRegistrationBean

```

public class MyServlet extends HttpServlet {

    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
        doPost(req,resp);
    }

    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
        resp.getWriter().write("Hello MyServlet");
    }
}

```

FilterRegistrationBean

```

public class MyFilter implements Filter {

    @Override
    public void init(FilterConfig filterConfig) throws ServletException {
    }

    @Override
    public void doFilter(ServletRequest request, ServletResponse response,
FilterChain chain) throws IOException, ServletException {

        System.out.println("MyFilter process...");

        chain.doFilter(request,response);
    }

    @Override
    public void destroy() {
    }
}

```

```
}
```

ServletListenerRegistrationBean

```
public class MyListener implements ServletContextListener {  
  
    @Override  
    public void contextInitialized(ServletContextEvent sce) {  
        System.out.println("contextInitialized...web应用启动");  
    }  
  
    @Override  
    public void contextDestroyed(ServletContextEvent sce) {  
        System.out.println("contextDestroyed...当前web项目销毁");  
    }  
}
```

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

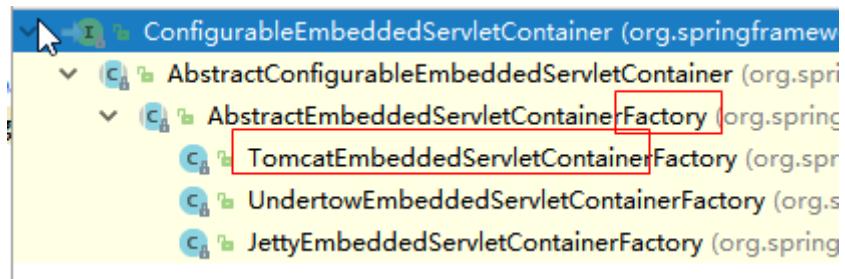
DispatcherServlet；

DispatcherServletAutoConfiguration中：

```
@Bean(name = 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;  
}
```

2)、SpringBoot能不能支持其他的Servlet容器；

3) 替换为其他嵌入式Servlet容器



默认支持：

Tomcat (默认使用)

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

```

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)

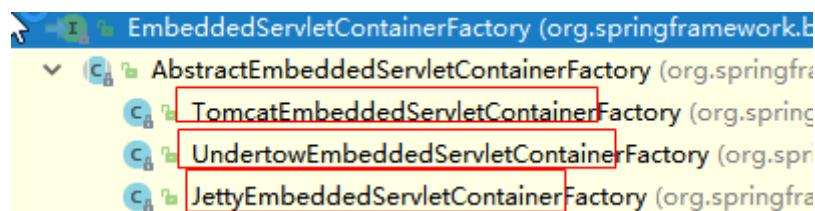
```

public interface EmbeddedServletContainerFactory {

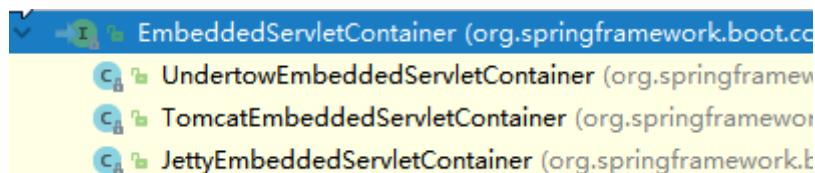
    //获取嵌入式的servlet容器
    EmbeddedServletContainer getEmbeddedServletContainer(
        ServletContextInitializer... initializers);

}

```



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);  
}
```

4) 、我们对嵌入式容器的配置修改是怎么生效?

ServerProperties、EmbeddedServletContainerCustomizer

EmbeddedServletContainerCustomizer: 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EmbeddedServletContainerCustomizerBeanPostProcessor

```
//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葛类型的组件： 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
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；

```
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

9、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应用的主程序
        return application.sources(SpringBoot04WebJspApplication.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:

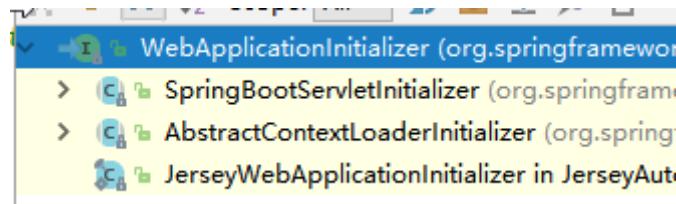
- 1) 、启动Tomcat
- 2) 、org\springframework\spring-web\4.3.14.RELEASE\spring-web-4.3.14.RELEASE.jar!\META-INF\services\javax.servlet.ServletContainerInitializer:

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；



- 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

五、Docker

1、简介

Docker

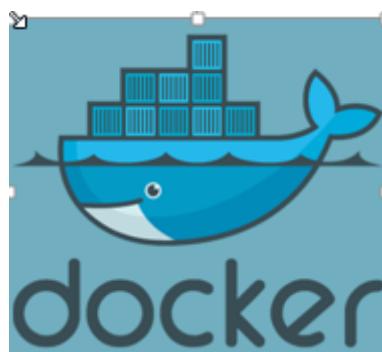
是一个开源的应用容器引擎；

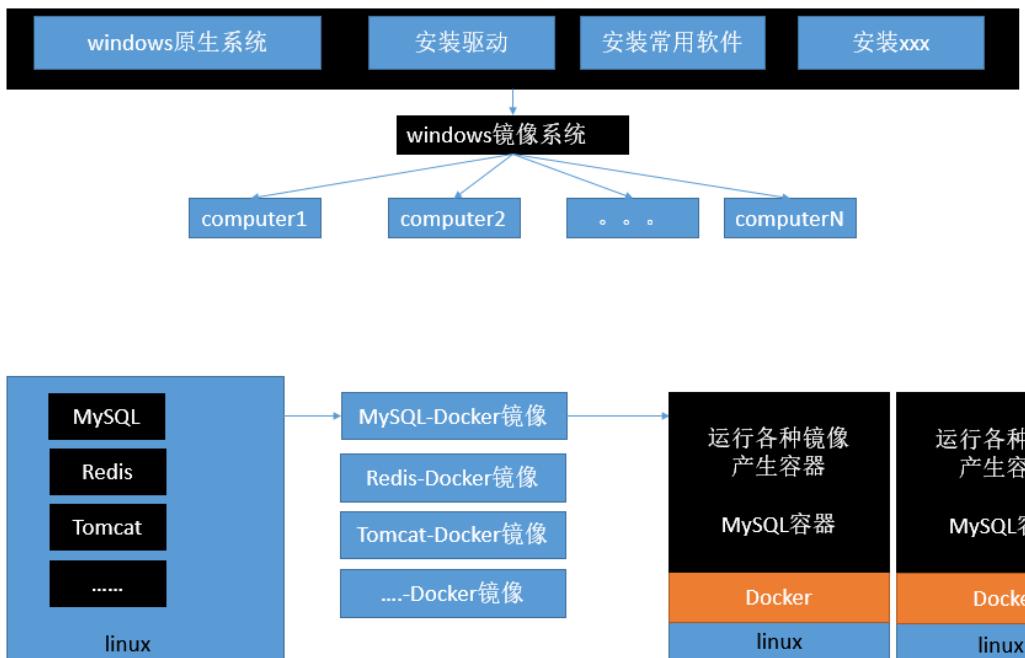
是一个轻量级容器技术；

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

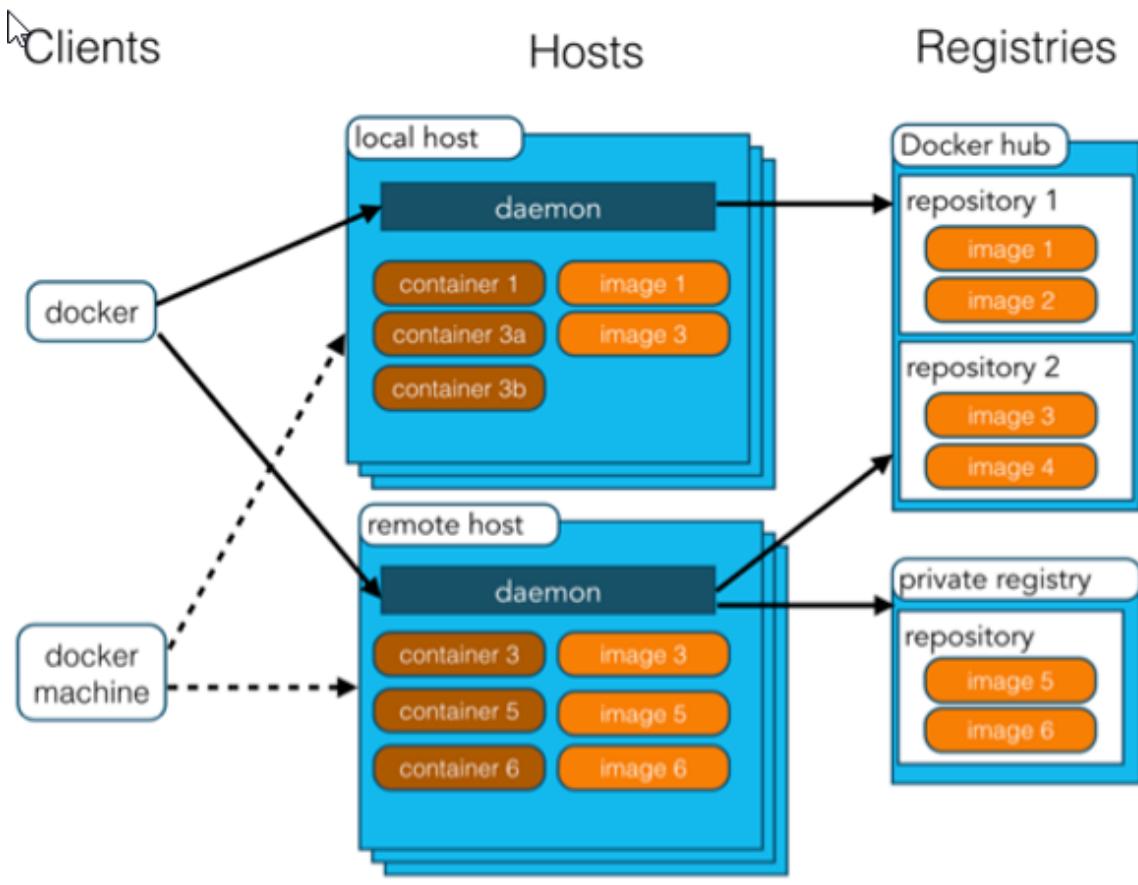
docker主机(Host): 安装了Docker程序的机器 (Docker直接安装在操作系统之上)；

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；

3、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

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
```

更多命令参看

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

可以参考每一个镜像的文档

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)	33 seconds ago		mysql01
538bde63e500	tomcat	"catalina.sh run"	About an hour
ago	Exited (143)	About an hour ago	compassionate_goldstine
c4f1ac60b3fc	tomcat	"catalina.sh run"	About an hour
ago	Exited (143)	About an hour ago	Tonely_fermi

```
81ec743a5271      tomcat          "catalina.sh run"      About an hour
ago    Exited (143) About an hour ago                                sick_ramanujan
```

```
//错误日志
//这个三个参数必须指定一个
[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;
```

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
STATUS              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            STATUS              PORTS
STATUS              NAMES
ad10e4bc5c6a        mysql              "docker-entrypoint.sh"   4 seconds ago
Up 2 seconds        0.0.0.0:3306->3306/tcp     mysql02
```

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 mysql docker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
```

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 #com.mysql.cj.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

```
/***
 * Generic DataSource configuration.
```

```

*/
@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;

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`;

可以使用

`schema:`
 `- classpath:department.sql`
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
@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"); // 视频计算机的ip
    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;
}
}

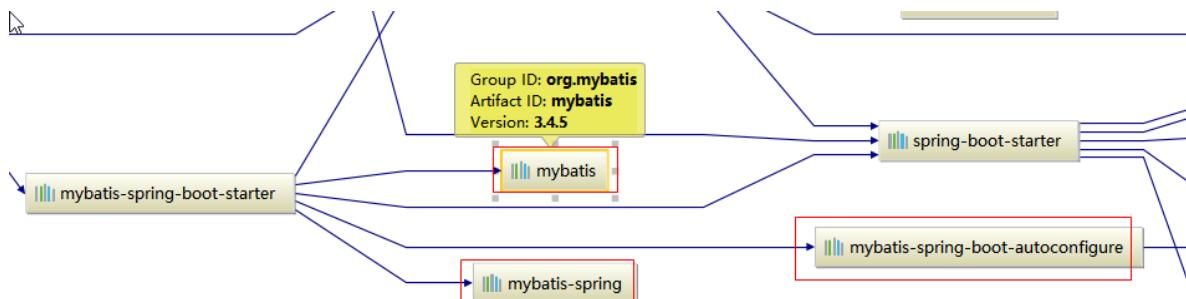
```

3、整合MyBatis

```

<dependency>
    <groupId>org.mybatis.spring.boot</groupId>
    <artifactId>mybatis-spring-boot-starter</artifactId>
    <version>1.3.1</version>
</dependency>

```



步骤：

- 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

4)、注解版

```
// 指定这是一个操作数据库的mapper
// @Mapper或@MapperScan将接口扫描装配到容器中
@Mapper

public interface DepartmentMapper {

    // #{id}
    // Integer id
    @Select("select * from department where id=#{id}")
    public Department getDeptById(Integer id);

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

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

    // 使用自动生成的键
    @Options(useGeneratedKeys = true, keyProperty = "id")

    @Insert("insert into department(department_name) values=#{departmentName}")
    public int insertDept(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.feuoy.springboot.mapper")

@SpringBootApplication
public class SpringBoot06DataMybatisApplication {
    public static void main(String[] args) {
        SpringApplication.run(SpringBoot06DataMybatisApplication.class, args);
    }
}
```

5)、配置文件版

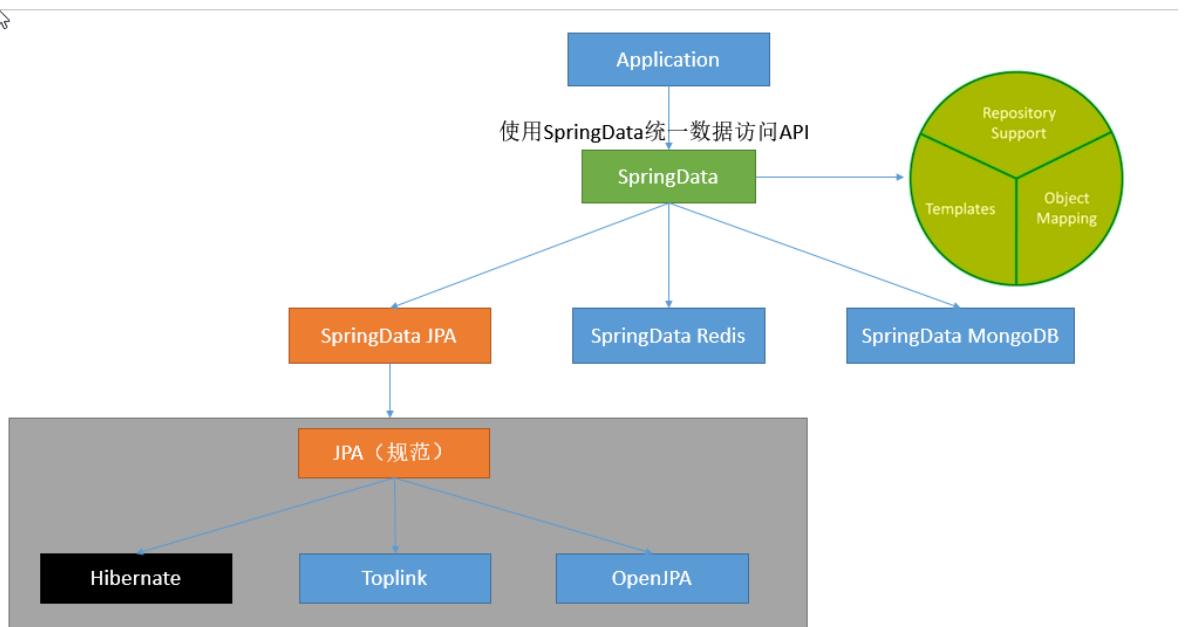
```
# 配置的方式
mybatis:
  # 指定全局配置文件位置
  config-location: classpath:mybatis/mybatis-config.xml
  # 指定sql映射文件位置
  mapper-locations: classpath:mybatis/mapper/*.xml
```

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



1、SpringData特点

SpringData为我们提供使用统一的API来对数据访问层进行操作；这主要是Spring Data Commons项目来实现的。Spring Data Commons让我们在使用关系型或者非关系型数据访问技术时都基于Spring提供的统一标准，标准包含了CRUD（创建、获取、更新、删除）、查询、排序和分页的相关操作。

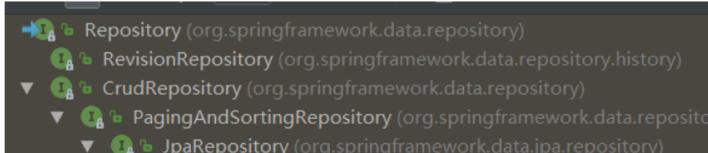
2、统一的Repository接口

`Repository<T, ID extends Serializable>` : 统一接口

`RevisionRepository<T, ID extends Serializable, N extends Number & Comparable<N>>` : 基于乐观锁机制

`CrudRepository<T, ID extends Serializable>` : 基本CRUD操作

`PagingAndSortingRepository<T, ID extends Serializable>` : 基本CRUD及分页



3、提供数据访问模板类 xxxTemplate；
如：MongoTemplate、RedisTemplate等

4、JPA与Spring Data

1) 、 JpaRepository基本功能

编写接口继承JpaRepository既有crud及分页等基本功能

2) 、 定义符合规范的方法命名

在接口中只需要声明符合规范的方法，即拥有对应的功能

Keyword	Sample	JPQL snippet
And	findByLastnameAndFirstname	where x.lastname = ?1 and x.firstname = ?2
Or	findByLastnameOrFirstname	where x.lastname = ?1 or x.firstname = ?2
Between	findByStartDateBetween	... where x.startDate between ?1 and ?2

3) 、 @Query自定义查询，定制查询SQL

4) 、 Specifications查询（ Spring Data JPA支持JPA2.0的Criteria查询）

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（和数据表映射的类）
@Table(name = "tbl_user") // @Table来指定和哪个数据表对应；如果省略默认表名就是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 // 省略默认列名就是属性名
    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
// 写上对象类型<User, Integer>
public interface UserRepository
    extends JpaRepository<User, Integer> {
}
```

3)、基本的配置JpaProperties

```
spring:
  jpa:
    hibernate:
      #       更新或创建数据表结构
      ddl-auto: update

      #       控制台显示SQL
      show-sql: true
```

七、启动配置原理

一、启动原理

- SpringApplication.run(主程序类)
 - **new SpringApplication(主程序类)**
 - 判断是否web应用
 - 加载并保存所有ApplicationContextInitializer (META-INF/spring.factories) ,
 - 加载并保存所有ApplicationListener
 - 获取到主程序类
 - **run()**
 - 回调所有的SpringApplicationRunListener (META-INF/spring.factories) 的starting
 - 获取ApplicationArguments
 - 准备环境&回调所有监听器 (SpringApplicationRunListener) 的environmentPrepared
 - 打印banner信息
 - 创建ioc容器对象 (
 - AnnotationConfigEmbeddedWebApplicationContext (web环境容器)
 - AnnotationConfigApplicationContext (普通环境容器))

- run()

- 准备环境
 - 执行**ApplicationContextInitializer**. initialize()
 - 监听器**SpringApplicationRunListener**回调**contextPrepared**
 - 加载主配置类定义信息
 - 监听器**SpringApplicationRunListener**回调**contextLoaded**
- 刷新启动IOC容器；
 - 扫描加载所有容器中的组件
 - 包括从**META-INF/spring.factories**中获取的所有**EnableAutoConfiguration**组件
- 回调容器中所有的**ApplicationRunner**、**CommandLineRunner**的run方法
- 监听器**SpringApplicationRunListener**回调**finished**

二、自动配置

- Spring Boot启动扫描所有jar包的**META-INF/spring.factories**中配置的**EnableAutoConfiguration**组件
- **spring-boot-autoconfigure.jar****META-INF\spring.factories**有启动时需要加载的**EnableAutoConfiguration**组件配置
- 配置文件中使用**debug=true**可以观看到当前启用的自动配置的信息
- 自动配置会为容器中添加大量组件
- Spring Boot在做任何功能都需要从容器中获取这个功能的组件
- Spring Boot 总是遵循一个标准；容器中有我们自己配置的组件就用我们配置的，没有就用自动配置默认注册进来的组件；

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

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));
}

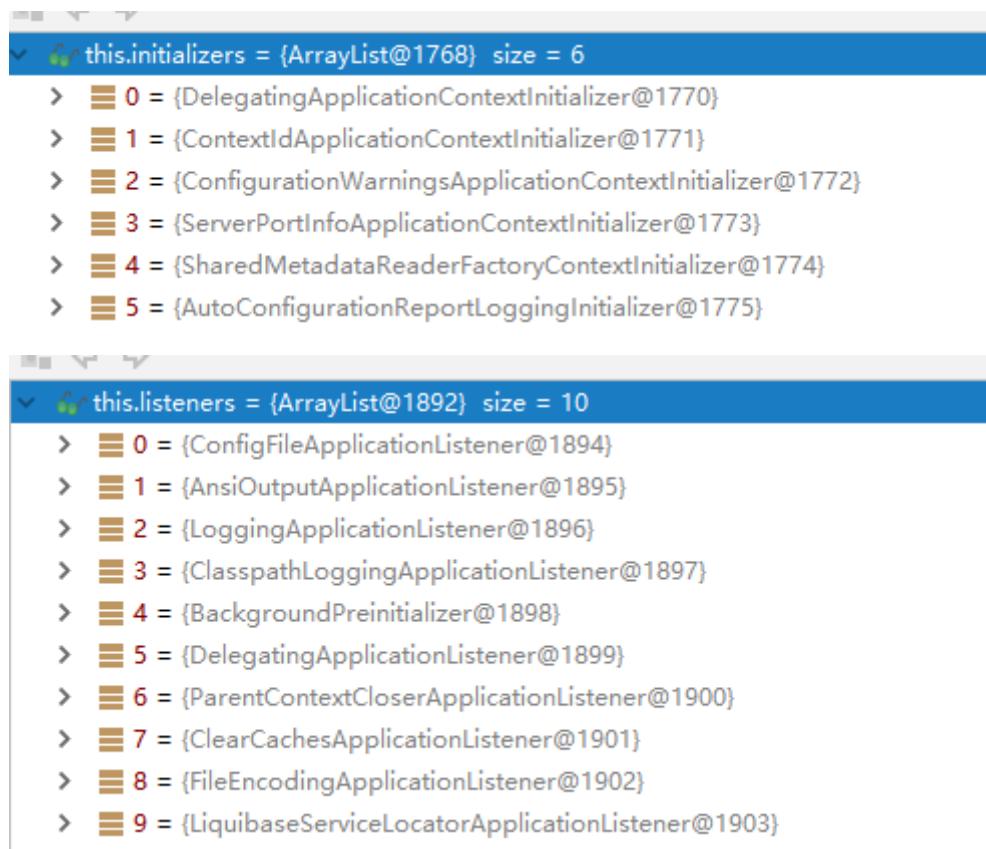
//判断当前是否一个web应用
this.webEnvironment = deduceWebEnvironment();

//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 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

```
public class HelloApplicationContextInitializer implements
ApplicationContextInitializer<ConfigurableApplicationContext> {

    @Override
    public void initialize(ConfigurableApplicationContext applicationContext) {

        System.out.println("ApplicationContextInitializer...initialize..."+applicationC
ontext);
    }
}
```

SpringApplicationRunListener

```
public class HelloSpringApplicationRunListener implements
SpringApplicationRunListener {

    //必须有的构造器
    public HelloSpringApplicationRunListener(SpringApplication application,
String[] args){}

    @Override
    public void starting() {
        System.out.println("SpringApplicationRunListener...starting...");
    }

    @Override
    public void environmentPrepared(ConfigurableEnvironment environment) {
        Object o = environment.getSystemProperties().get("os.name");

        System.out.println("SpringApplicationRunListener...environmentPrepared.."+o);
    }

    @Override
    public void contextPrepared(ConfigurableApplicationContext context) {
        System.out.println("SpringApplicationRunListener...contextPrepared...");
    }
}
```

```
}

@Override
public void contextLoaded(ConfigurableApplicationContext context) {
    System.out.println("SpringApplicationRunListener...contextLoaded..."); 
}

@Override
public void finished(ConfigurableApplicationContext context, Throwable exception) {
    System.out.println("SpringApplicationRunListener...finished..."); 
}
}
```

配置 (META-INF/spring.factories)

```
org.springframework.context.ApplicationContextInitializer=\
com.atguigu.springboot.listener.HelloApplicationContextInitializer

org.springframework.boot.SpringApplicationRunListener=\
com.atguigu.springboot.listener.HelloSpringApplicationRunListener
```

只需要放在ioc容器中

ApplicationRunner

```
@Component
public class HelloApplicationRunner implements ApplicationRunner {

    @Override
    public void run(ApplicationArguments args) throws Exception {
        System.out.println("ApplicationRunner...run....");
    }
}
```

CommandLineRunner

```
@Component
public class HelloCommandLineRunner implements CommandLineRunner {

    @Override
    public void run(String... args) throws Exception {
        System.out.println("CommandLineRunner...run..."+ Arrays.asList(args));
    }
}
```

八、自定义starter

一、自定义starters

- 自动装配Bean；
 - 自动装配使用配置类（@Configuration）结合Spring4 提供的条件判断注解 @Conditional及Spring Boot的派生注解如@ConditionOnClass完成；
- 配置自动装配Bean；
 - 将标注@Configuration的自动配置类，放在classpath下META-INF/spring.factories文件中，如：

```
# Auto Configure
org.springframework.boot.autoconfigure.EnableAutoConfiguration=
org.springframework.boot.autoconfigure.admin.SpringApplicationAdminJmxAutoConfiguration,
org.springframework.boot.autoconfigure.aop.AopAutoConfiguration,
org.springframework.boot.autoconfigure.amqp.RabbitAutoConfiguration,
org.springframework.boot.autoconfigure.batch.BatchAutoConfiguration,
org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration,
org.springframework.boot.autoconfigure.cassandra.CassandraAutoConfiguration,
org.springframework.boot.autoconfigure.cloud.CloudAutoConfiguration,
org.springframework.boot.autoconfigure.context.ConfigurationPropertiesAutoConfiguration,
org.springframework.boot.autoconfigure.context.MessageSourceAutoConfiguration,
```

- **自动装配顺序**
 - 在特定自动装配Class之前
 - @AutoConfigureBefore
 - 在特定自动装配Class之后
 - @AutoConfigureAfter
 - 指定顺序
 - @AutoConfigureOrder

- 启动器 (starter)

- 启动器模块是一个空 JAR 文件，仅提供辅助性依赖管理，这些依赖可能用于自动装配或者其他类库

- 命名规约：

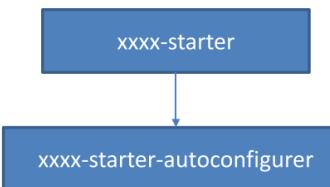
- 推荐使用以下命名规约；

- 官方命名空间

- 前缀：“spring-boot-starter-”
 - 模式：spring-boot-starter-模块名
 - 举例：spring-boot-starter-web、spring-boot-starter-actuator、spring-boot-starter-jdbc

- 自定义命名空间

- 后缀：“-spring-boot-starter”
 - 模式：模块-spring-boot-starter
 - 举例：mybatis-spring-boot-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  
 @ConfigurationProperties //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  
 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)

自定义启动器名-spring-boot-starter；

```
mybatis-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 sayHelloAtguigu(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>

Spring Boot高级

一、SpringBoot与缓存

JSR-107、Spring缓存抽象、整合Redis

(一) JSR-107、

Java Caching定义了5个核心接口，

CachingProvider

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

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

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

一个存储在Cache中的key-value对。

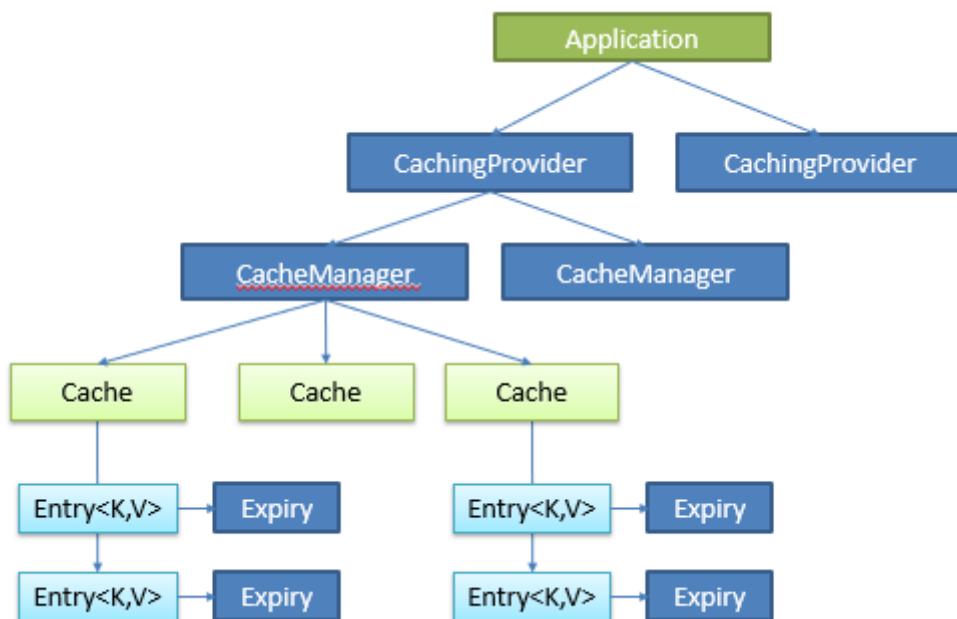
Expiry

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。



(二) Spring缓存抽象

1、

从Spring从3.1开始，定义了

`org.springframework.cache.Cache`,

`org.springframework.cache.CacheManager`

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；

2、Cache接口

它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实现，如RedisCache，EhCacheCache，ConcurrentMapCache等；

3、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；

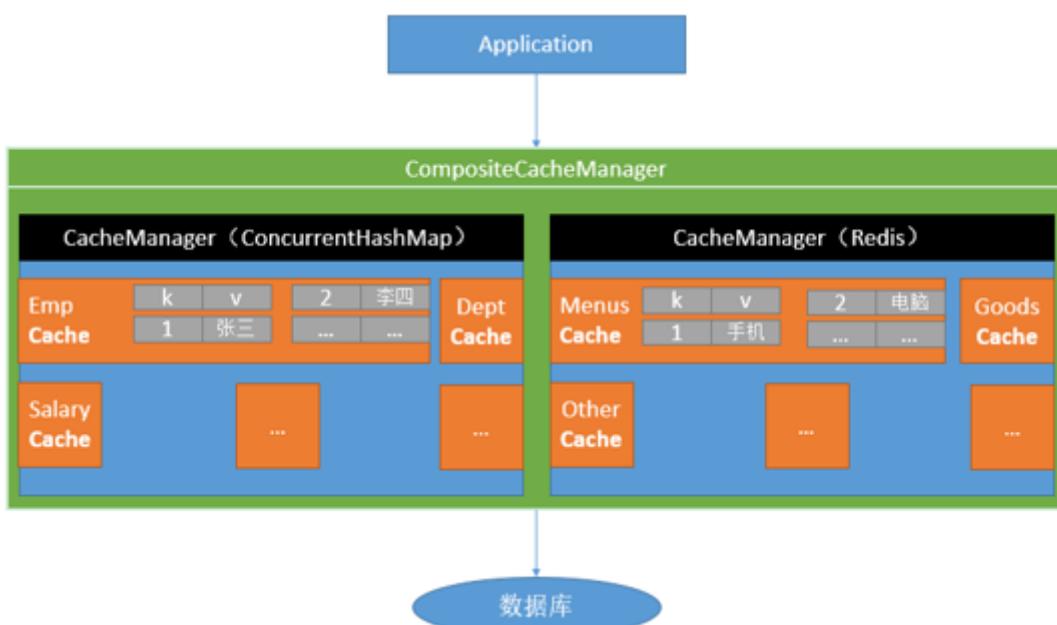
如果有就直接从缓存中获取方法调用后的结果，

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。

4、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



(三) 几个重要概念&缓存注解

1、

Cache	缓存接口，定义缓存操作。实现有： <code>RedisCache</code> 、 <code>EhCacheCache</code> 、 <code>ConcurrentMapCache</code> 等
CacheManager	缓存管理器，管理各种缓存（Cache）组件
<code>@Cacheable</code>	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
<code>@CacheEvict</code>	清空缓存
<code>@CachePut</code>	保证方法被调用，又希望结果被缓存。
<code>@EnableCaching</code>	启基于注解的缓存
<code>keyGenerator</code>	缓存数据时key生成策略
<code>serialize</code>	缓存数据时value序列化策略

2、`@Cacheable`、`@CachePut`、`@CacheEvict` 主要的参数

`cacheNames / value`

缓存的名称，在spring 配置文件中定义，必须指定至少一个

```
@Cacheable(value="mycache")
```

```
@Cacheable(value={"cache1","cache2"})
```

`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

```
@Cacheable(value="testcache",key="#id")
```

condition

缓存的条件，可以为空，在调用方法之前之后都能判断

使用SpEL 编写，返回 true 或 false，只有为 true 才进行缓存/清除缓存，

```
@Cacheable(value="testcache",condition="#userName.length()>2")
```

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

```
@CacheEvict(value="testcache",allEntries=true)
```

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

```
@CacheEvict(value="testcache", beforeInvocation=true)
```

unless (@CachePut) (@Cacheable)

用于否决缓存的，不像condition，该表达式只在方法执行之后判断，

此时可以拿到返回值result进行判断。条件为true不会缓存，fasle才缓存

```
@Cacheable(value="testcache",unless="#result== null")
```

3、Cache SpEL available metadata

名字	位置	描述	示例
methodName	root object	当前被调用的方法名	#root.methodName
method	root object	当前被调用的方法	#root.method.name
target	root object	当前被调用的目标对象	#root.target
targetClass	root object	当前被调用的目标对象类	#root.targetClass
args	root object	当前被调用的方法的参数列表	#root.args[0]
caches	root object	当前方法调用使用的缓存列表， (如@Cacheable(value={"cache1", "cache2"}))有两个 cache)	#root.caches[0].name
argument name	evaluation context	方法参数的名字，可以直接 #参数名，也可以使用 #p0或#a0的形式，0代表参数的索引	#iban、#a0、#p0
result	evaluation context	方法执行后的返回值 (仅当方法执行之后的判断有效， 如'unless'， 'cache put'的表达式， 'cache evict'的表达式 beforeInvocation=false)	#result

(四) 缓存使用

- 1、引入spring-boot-starter-cache模块
- 2、@EnableCaching开启缓存
- 3、使用缓存注解
- 4、切换为其他缓存

(五) 整合Redis

- 1、引入spring-boot-starter-data-redis
- 2、application.yml配置redis连接地址

- 3、使用RestTemplate操作redis
 - redisTemplate.opsForValue(); //操作字符串
 - redisTemplate.opsForHash(); //操作hash
 - redisTemplate.opsForList(); //操作list
 - redisTemplate.opsForSet(); //操作set
 - redisTemplate.opsForZSet(); //操作有序set

- 4、配置缓存、CacheManagerCustomizers
- 5、测试使用缓存、切换缓存、CompositeCacheManager

二、SpringBoot与消息

JMS、AMQP、RabbitMQ

(一)、概述

1、

大多应用中，可通过消息服务中间件来提升系统异步通信、扩展解耦能力

2、消息服务中两个重要概念：

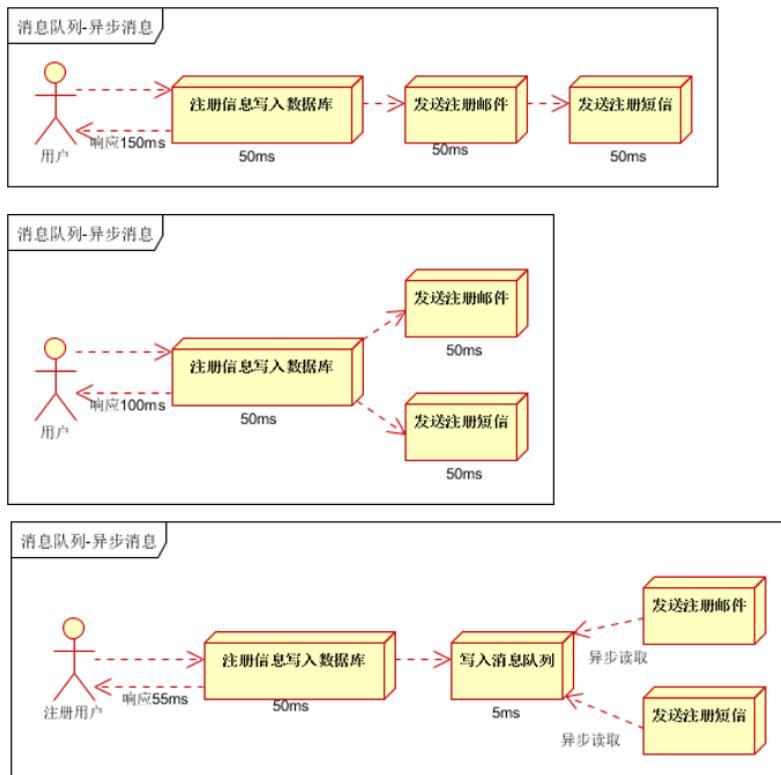
消息代理 (message broker)、目的地 (destination)

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。

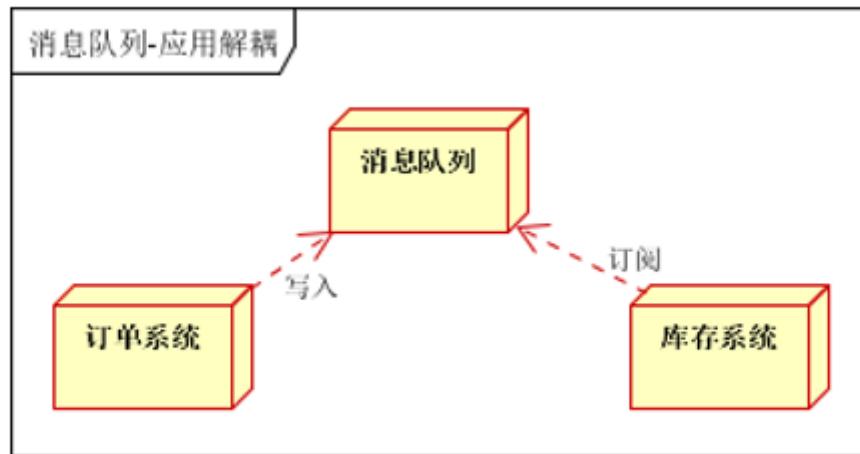
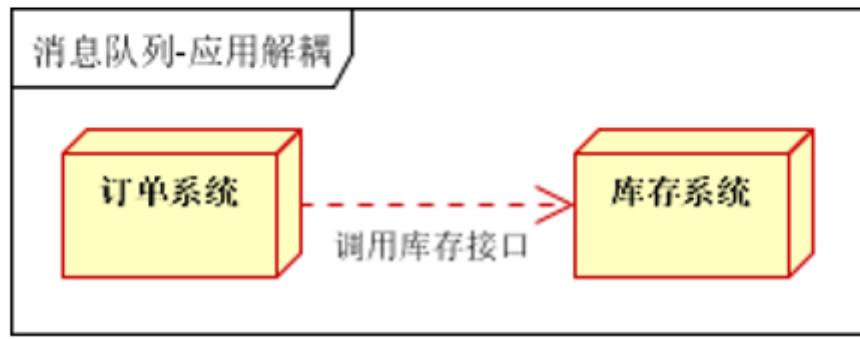
3、消息队列主要有两种形式的目的地

- 1) 队列 (queue) : (point-to-point) 点对点消息通信
- 2) 主题 (topic) : (publish/subscribe) 发布订阅消息通信

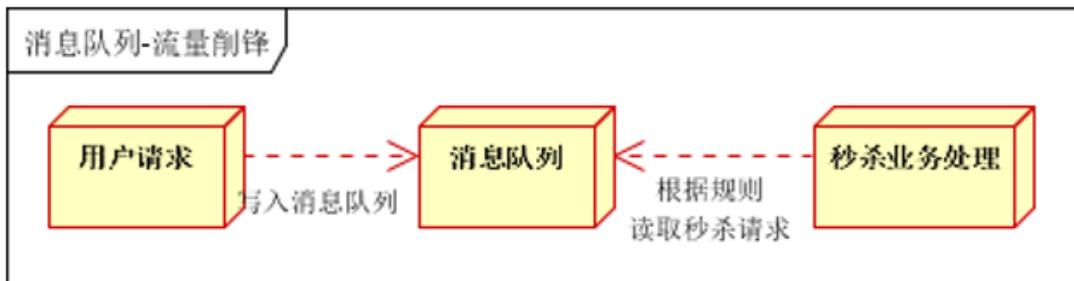
异步处理



应用解耦



流量削峰



4、点对点式：

消息发送者发送消息，消息代理将其放入一个队列，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；

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：

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，基于JVM消息代理的规范

ActiveMQ、HornetMQ是JMS实现

7、AMQP (Advanced Message Queuing Protocol)

高级消息队列协议，也是一个消息代理的规范，兼容JMS

RabbitMQ是AMQP的实现

8、

	JMS	AMQP
定义	Java api	网络线级协议
跨语言	否	是
跨平台	否	是
Model	提供两种消息模型： (1) Peer-2-Peer (2) Pub/sub	提供了五种消息模型： (1) direct exchange (2) fanout exchange (3) topic change (4) headers exchange (5) system exchange 本质来讲，后四种和JMS的pub/sub模型没有太大差别，仅是在路由机制上做了更详细的划分；
支持消息类型	TextMessage MapMessage BytesMessage StreamMessage	byte[]，当实际应用时，有复杂的消息，可以将消息

息类型	JMS ObjectMessage Message (只有消息头和属性)	序列化后发送 AMQP
综合评价	定义了JAVA API层面的标准；在java体系中，多个client均可以通过JMS进行交互，不需要应用修改代码，但是其对跨平台的支持较差	定义了wire-level层的协议标准；天然具有跨平台、跨语言特性

9、Spring支持

spring-jms提供了对JMS的支持

spring-rabbit提供了对AMQP的支持

需要ConnectionFactory的实现来连接消息代理

提供JmsTemplate (JMS) 、 RabbitTemplate (AMQP) 来发送消息

@JmsListener (JMS) 、 @RabbitListener (AMQP) 注解在方法上监听消息代理发布的消息

@EnableJms (JMS) 、 @EnableRabbit (AMQP) 开启支持

10、Spring Boot自动配置

JmsAutoConfiguration

RabbitAutoConfiguration

(二) 、 RabbitMQ简介

0、开启RabbitMQ：

方式1：

RabbitMQ Server\rabbitmq_server-3.8.0\sbin\rabbitmq-server.bat

方式2（推荐）：

services.msc——RabbitMQ

地址：

<http://localhost:15672/>

默认的账号密码：

双guest

1、RabbitMQ简介：

RabbitMQ是一个由erlang开发的AMQP的开源实现

2、核心概念

Message

消息，消息是不具名的，它由消息头和消息体组成。

消息体是不透明的，而消息头则由一系列的可选属性组成，

这些属性包括

routing-key（路由键）、

priority（相对于其他消息的优先权）、

delivery-mode（指出该消息可能需要持久性存储）等

Publisher

消息的生产者，也是一个向交换器发布消息的客户端应用程序

Exchange

交换器，用来接收生产者发送的消息并将这些消息路由给服务器中的队列

Exchange有4种类型：direct（默认），fanout，topic，headers，

不同类型的Exchange转发消息的策略有所区别

Queue

消息队列，用来保存消息直到发送给消费者。

它是消息的容器，也是消息的终点。

一个消息可投入一个或多个队列。

消息一直在队列里面，等待消费者连接到这个队列将其取走。

Binding

绑定，用于消息队列和交换器之间的关联。

一个绑定就是基于路由键将交换器和消息队列连接起来的路由规则，所以可以将交换器理解成一个由绑定构成的路由表。

Exchange 和Queue的绑定可以是多对多的关系。

Connection

网络连接，比如一个TCP连接。

Channel

信道，多路复用连接中的一条独立的双向数据流通道，建立在真实的TCP连接内的虚拟连接，

AMQP 命令都是通过信道发出去的，不管是发布消息、订阅队列还是接收消息，这些动作都是通过信道完成。

因为对于操作系统来说建立和销毁 TCP 都是非常昂贵的开销，所以引入了信道的概念，以复用一条 TCP 连接

Consumer

消息的消费者，表示一个从消息队列中取得消息的客户端应用程序。

Virtual Host

虚拟主机，表示一批交换器、消息队列和相关对象。

虚拟主机是共享相同的身份认证和加密环境的独立服务器域。

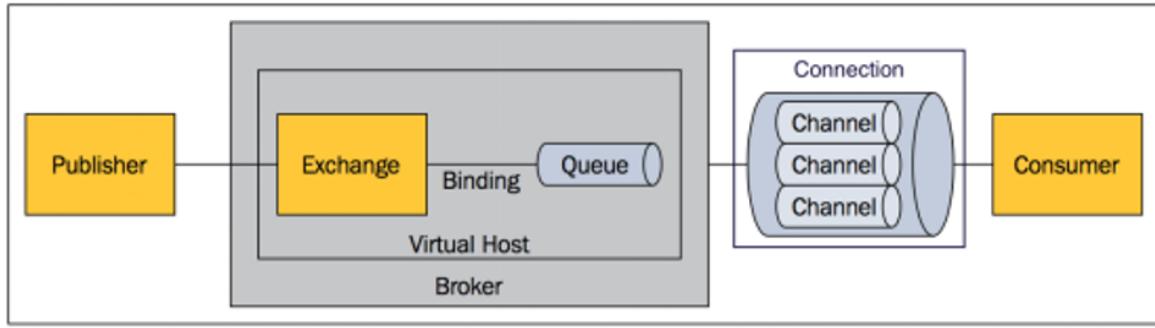
每个 vhost 本质上就是一个 mini 版的 RabbitMQ 服务器，拥有自己的队列、交换器、绑定和权限机制。

vhost 是 AMQP 概念的基础，必须在连接时指定，

RabbitMQ 默认的 vhost 是 /

Broker

表示消息队列服务器实体



(三) 、 RabbitMQ运行机制

AMQP 中的消息路由

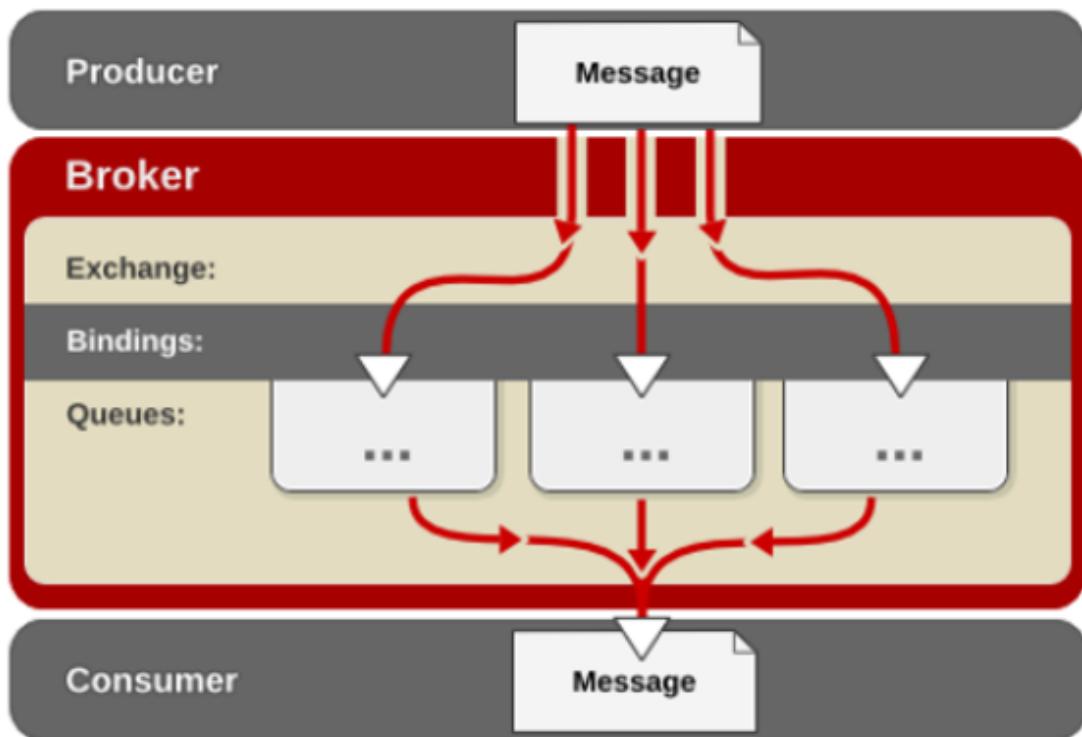
AMQP 中消息的路由过程和 Java 开发者熟悉的 JMS 存在一些差别，

增加了 Exchange 和 Binding 的角色。

生产者把消息发布到 Exchange 上，消息最终到达队列并被消费者接收，

而 Binding 决定交换器的消息应该发送到那个队列

Producer Consumer



Exchange类型

Exchange分发消息时根据类型的不同分发策略有区别，

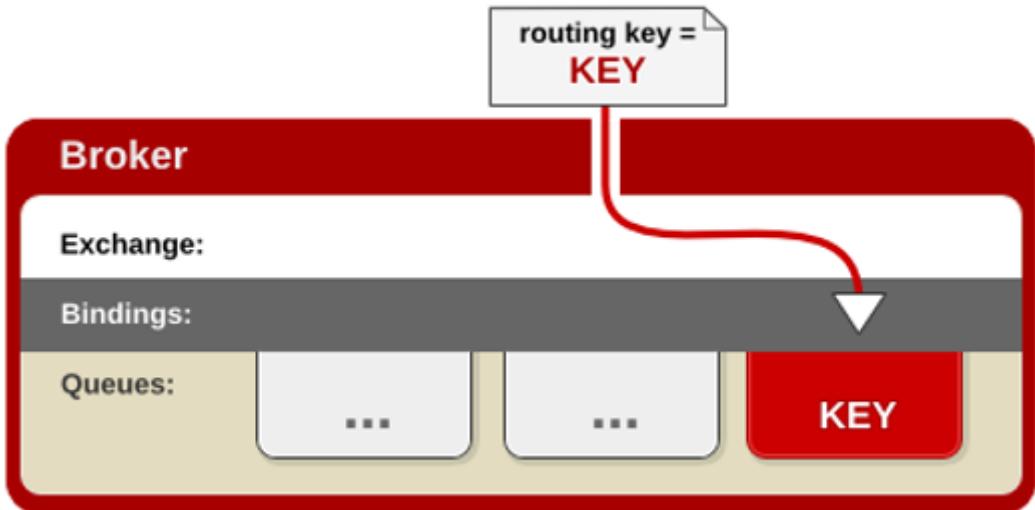
目前共四种类型：direct、fanout、topic、headers

headers 匹配 AMQP 消息的 header 而不是路由键，它和 direct 完全一致，但性能差很多，目前几乎用不到了，

所以直接看另外三种类型。

(1) direct

Direct Exchange



如果消息中的routing key和 Binding 中的 binding key 一致,

那么交换器就将消息发到对应的队列中,

routing key与队列名完全匹配

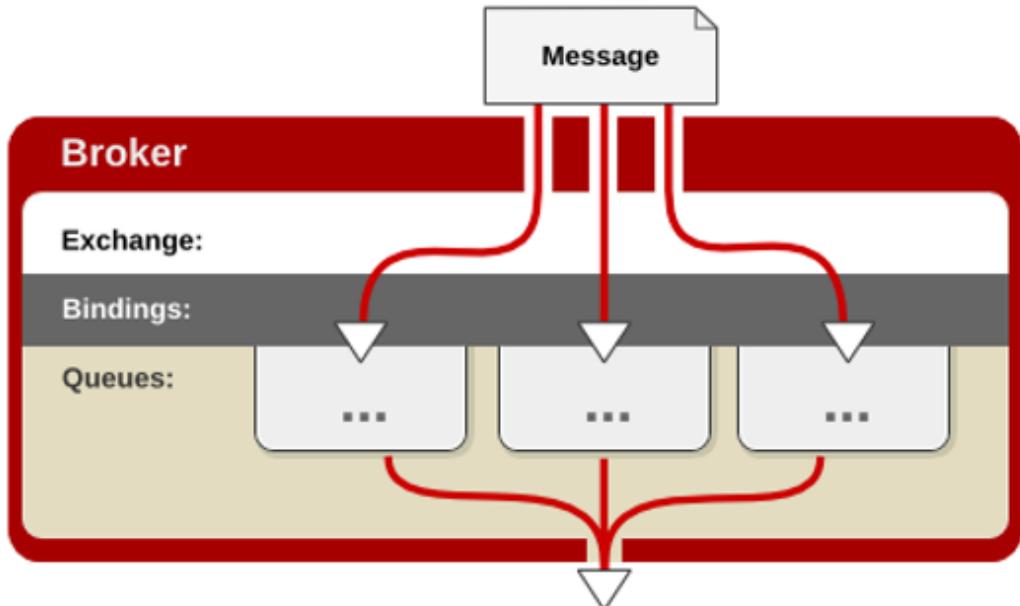
比如，一个队列绑定到交换机要求路由键为“dog”，

则只转发 routing key 标记为“dog”的消息，不会转发“dog.puppy”，也不会转发“dog.guard”等等，

它是完全匹配、单播的模式

(2) fanout

Fanout Exchange



每个发到 fanout 类型交换器的消息都会分到所有绑定的队列上去。

fanout 交换器不处理路由键，只是简单的将队列绑定到交换器上，

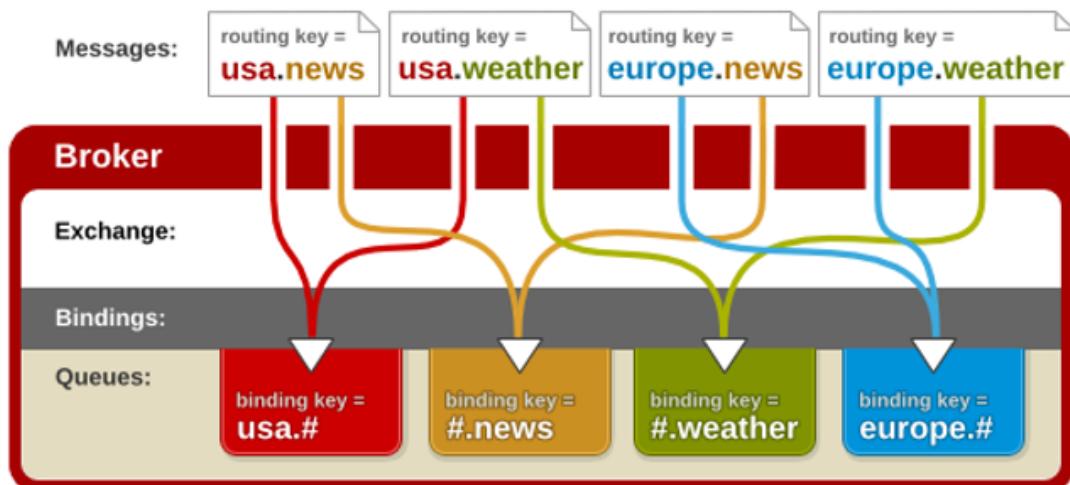
每个发送到交换器的消息都会被转发到与该交换器绑定的所有队列上。

很像子网广播，每台子网内的主机都获得了一份复制的消息。

fanout 类型转发消息是最快的。

(3) topic

Topic Exchange



topic 交换器通过模式匹配分配消息的路由键属性，
将路由键和某个模式进行匹配，此时队列需要绑定到一个模式上
它将路由键和绑定键的字符串切分成单词，这些单词之间用点隔开。
它同样也会识别两个通配符：#匹配0个或多个单词、*匹配一个单词

(四) 、 RabbitMQ整合

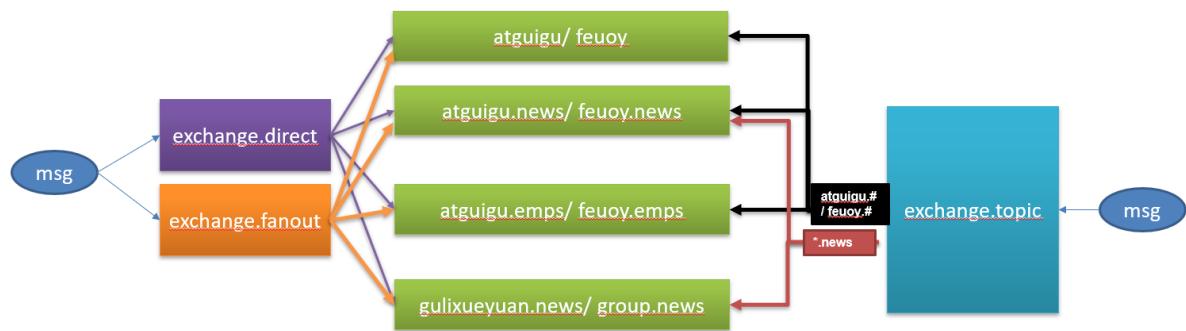
1、引入 spring-boot-starter-amqp

2、application.yml配置

3、测试RabbitMQ

AmqpAdmin：管理组件

RabbitTemplate：消息发送处理组件



三、SpringBoot与检索

(一) 、检索

我们的应用经常需要添加检索功能，开源的 **ElasticSearch** 是全文搜索引擎的首选。

它是一个分布式搜索服务，可以快速的存储、搜索和分析海量数据，

提供**Restful API**，底层基于Lucene，采用多shard（分片）的方式保证数据安全，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。

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；

(二) 、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**。

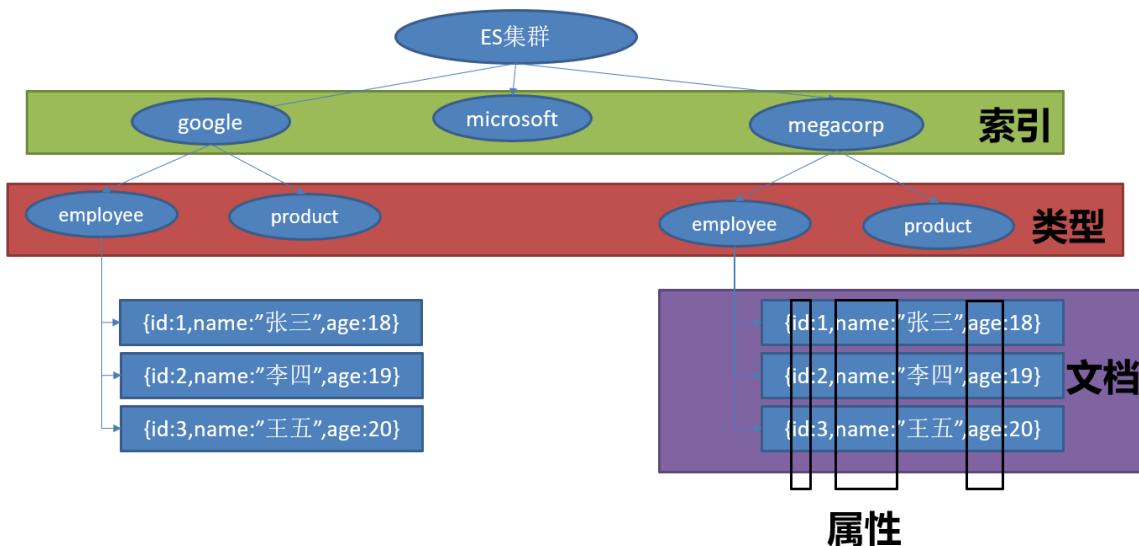
类似关系：

索引-数据库

类型-表

文档-表中的记录

属性-列



(三) 、整合ElasticSearch测试

- 1、引入spring-boot-starter-data-elasticsearch
- 2、安装Spring Data 对应版本的ElasticSearch
- 3、application.yml配置
- 4、Spring Boot自动配置的ElasticsearchRepository、ElasticsearchTemplate、Jest
- 5、测试ElasticSearch

(四) 、具体使用

开启ElasticSearch:

elasticsearch-5.6.9\bin\elasticsearch.bat

<http://127.0.0.1:9200/>

文档:

<https://www.elastic.co/guide/cn/elasticsearch/guide/current/index.html>

https://www.elastic.co/guide/cn/elasticsearch/guide/current/_document_oriented.html

一个put请求:

```
PUT /megacorp/employee/1
{
    "first_name" : "John",
    "last_name" : "Smith",
    "age" : 25,
    "about" : "I love to go rock climbing",
    "interests": [ "sports", "music" ]
}
```

Postman操作:

PUT

<http://localhost:9200/megacorp/employee/1>

(row) JSON

```
{
    "first_name" : "John",
    "last_name" : "Smith",
    "age" : 25,
    "about" : "I love to go rock climbing",
    "interests": [ "sports", "music" ]
}
```

一个get请求:

```
GET /megacorp/employee/1
```

Postman操作:

GET

<http://localhost:9200/megacorp/employee/1>

找所有文档:

```
GET /megacorp/employee/_search
```

尝试搜索姓氏为 Smith 的雇员:

```
GET /megacorp/employee/_search?q=last_name:Smith
```

使用查询表达式:

```
GET /megacorp/employee/_search
{
  "query" : {
    "match" : {
      "last_name" : "Smith"
    }
  }
}
```

Postman操作:

要放在请求体，所以灵活要用: POST

http://localhost:9200/megacorp/employee/_search

(row) JSON

```
{
  "query" : {
    "match" : {
      "last_name" : "Smith"
    }
  }
}
```

DELETE:

删除文档

HEAD:

检查文档是否存在，有的话200，否则404

[更多查看文档；](#)

主要查看文档，查字典一样而已

四、SpringBoot与任务

异步任务、定时任务、邮件任务

(一) 、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` (SpringbootApplication)

`@Aysnc` (方法的前面)

(二) 、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

cron表达式：

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	, - * /
星期	0-7或SUN-SAT 0,7是SUN	, - * ? / L C #

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个星期四

(三)、邮件任务

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



五、SpringBoot与安全

安全、Spring Security

(一)、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。

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

访问控制

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 (Authentication)

建立一个它声明的主体的过程；

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 (Authorization)

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；

(二) 、 Web&安全

1、登陆/注销

HttpSecurity配置登陆、注销功能

2、Thymeleaf提供的SpringSecurity标签支持

需要引入thymeleaf-extras-springsecurity5

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

3、remember me

表单添加remember-me的checkbox

配置启用remember-me功能

4、CSRF (Cross-site request forgery) 跨站请求伪造

HttpSecurity启用csrf功能，会为表单添加 csrf 的值，提交携带来预防CSRF

六、SpringBoot与分布式

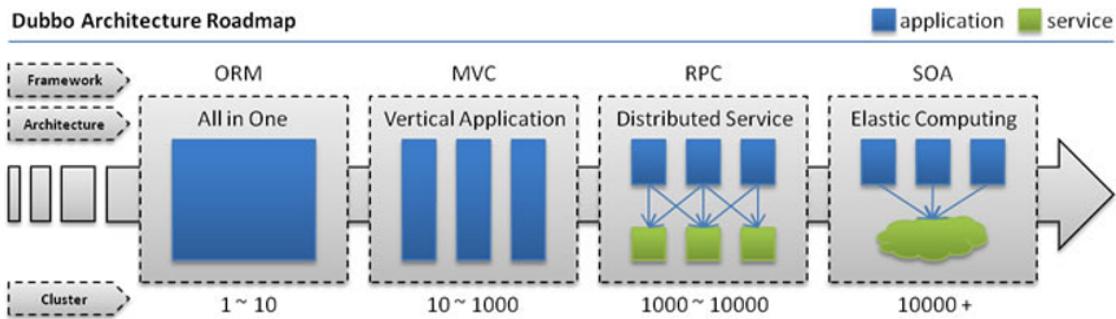
分布式、Dubbo/Zookeeper、Spring Boot/Cloud

(一)、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。

分布式系统：



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，

对象关系映射 (Object Relational Mapping, 简称ORM) 。

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，

MVC, Model View Controller, 模型(model) - 视图(view) - 控制器(controller)

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，

RPC (Remote Procedure Call Protocol) 远程过程调用协议。

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。

面向服务的架构 (SOA, Service-Oriented Architecture) 。

(二) 、Zookeeper和Dubbo

ZooKeeper

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

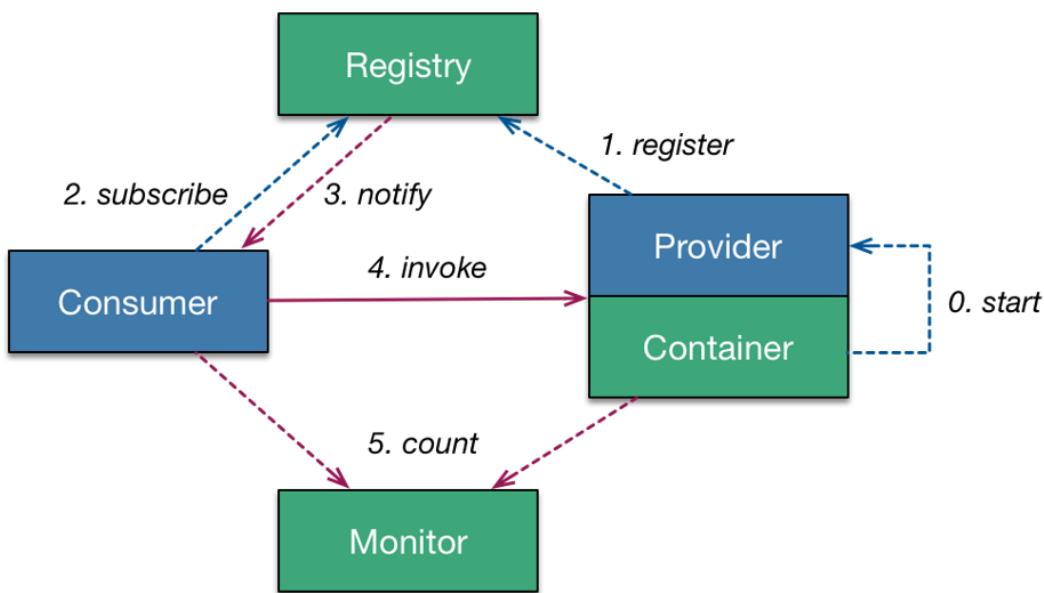
Dubbo是Alibaba开源的分布式服务框架，它最大的特点是按照分层的方式来架构，

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。

**step**

- 1、开启zookeeper作为注册中心 (zookeeper-3.4.14\bin\zkServer.cmd)
- 2、编写服务提供者
- 3、编写服务消费者
- 4、整合dubbo

(三) 、Spring Boot和Spring Cloud**Spring Cloud**

Spring Cloud是一个分布式的整体解决方案；

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；

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

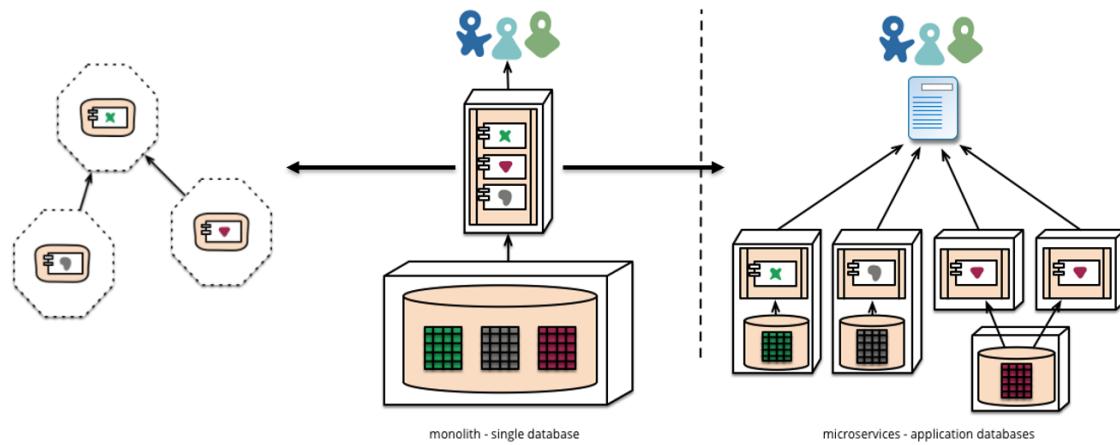
客服端负载均衡——Netflix Ribbon

断路器——Netflix Hystrix

服务网关——Netflix Zuul

分布式配置——Spring Cloud Config

微服务



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

eureka的服务地址

<http://localhost:8761/eureka/>

查看eureka中有哪些服务实例的地址

<http://localhost:8761/>

七、SpringBoot与热部署

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，如何能实现热部署——

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

Spring Loaded: <https://github.com/spring-projects/spring-loaded>

添加运行时参数：**javaagent:C:/springloaded-1.2.5.RELEASE.jar -noverify**

3、JRebel

JRebel是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**spring-boot-devtools**依赖
- IDEA使用**ctrl+F9**

【或做一些小调整】

1) 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

- 2) 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）
- 3) idea设置自动编译: settings-compiler-make project automatically, **ctrl+shift+alt+/ (maintenance)** , 勾选**compiler.automake.allow.when.app.running**
- 4) **ctrl+shift+alt+/ (maintenance)**
- 5) 勾选**compiler.automake.allow.when.app.running**

八、SpringBoot与监控管理

(一) 、监控管理

通过引入**spring-boot-starter-actuator**, 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:

- 1、引入**spring-boot-starter-actuator**
- 2、通过**http**方式访问监控端点
- 3、可进行**shutdown** (POST 提交, 此端点默认关闭)

端点地址:

<http://localhost:8080/actuator>

监控和管理端点——

端点名	描述
<u>autoconfig</u>	所有自动配置信息
<u>auditevents</u>	审计事件
<u>beans</u>	所有Bean的信息
configprops	所有配置属性
<u>dump</u>	线程状态信息
<u>env</u>	当前环境信息
<u>health</u>	应用健康状况
<u>info</u>	当前应用信息
<u>metrics</u>	应用的各项指标
<u>mappings</u>	应用@RequestMapping映射路径
shutdown	关闭当前应用（默认关闭）
<u>trace</u>	追踪信息（最新的http请求）

(二)、定制端点信息

spring2.x

定制端点一般通过management.endpoints+端点名+属性名来设置

修改端点id

management.endpoints.beans.id=mybeans

开启远程应用关闭功能

management.endpoints.shutdown.enabled=true

关闭端点

management.endpoints.beans.enabled=false

开启所需端点

```
management.endpoints.enabled=false  
management.endpoints.beans.enabled=true
```

定制端点访问根路径

```
management.context-path=/manage
```

关闭http端点

```
management.port=-1
```