一、Spring Boot入门

1、Spring Boot简介

Spring Boot是由Pivotal团队提供的全新框架，其设计目的是用来简化新Spring应用的初始搭建以及开发过程。该框架使用了特定的方式来进行配置，从而使开发人员不再需要定义样板化的配置。

2、微服务

微服务：架构风格（服务微化）

一个应用应该是一组小型服务，可以通过HTTP的方式进行互通

单体应用：ALL IN ONE

微服务：每个功能元素最终都是一个可以独立替换和升级的软件单元

3、环境准备

环境约束

- jdk1.8
- maven 3.x :maven3.3以上
- IDEA2017
- SpringBoot 1.5.9RELEASE

1、MAVEN设置

```
<!-- 配置JDK版本 -->
<profile>
    <id>jdk18</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>
   <!-- 当 nexus-aliyun 下不了的包，或许这个镜像能下，
        才开放它，这个实在太慢，而且要把它放在首位，即 nexus-aliyun 之前，做过测试。
        所以它的用途只有那么一瞬间，就是或许它能下载，可以通过 url 去查找确定一下
    -->
    <!-- <mirror>
        <id>spring-libs-milestone</id>
        <mirrorOf>central</mirrorOf>
```

```
        <name>Spring Milestones</name>
        <url>http://repo.spring.io/libs-milestone</url>
    </mirror> -->

    <!-- nexus-aliyun 首选，放第一位,有不能下载的包，再去做其他镜像的选择  -->
    <mirror>
        <id>nexus-aliyun</id>
        <mirrorOf>central</mirrorOf>
        <name>Nexus aliyun</name>
        <url>http://maven.aliyun.com/nexus/content/groups/public</url>
    </mirror>

    <!-- 备选镜像，也是可以通过 url 去查找确定一下，
        该镜像是否含有你想要的包，它比 spring-libs-milestone 快  -->
    <mirror>
        <id>central-repository</id>
        <mirrorOf>*</mirrorOf>
        typor<name>Central Repository</name>
        <url>http://central.maven.org/maven2/</url>
    </mirror>
```

2、IDEA设置

配置IDEA的Maven，指定Setting的Maven目录和MAVEN的setting.xml文件

快捷键：

Ctrl+D 复制一行

Ctrl+Y 删除一行

Ctrl+P 参数提示

Ctrl+Alt+V 自动补齐方法

Ctrl+N 查找类方法

Alt+Ins 构造器、getter/setter toString

Ctrl+O 重载方法提示

Alt+Enter 提示导入类etc

Shift+F6 :文件重命名

4、Spring Boot的Hello World

1、创建一个Maven工程

2、导入Spring Boot的相关依赖

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

```
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.0.1.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>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

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

3、编写个主程序

```
@SpringBootApplication
public class SpringBoot01HelloQuickApplication {

    public static void main(String[] args) {
        SpringApplication.run(SpringBoot01HelloQuickApplication.class, args);
    }
}
```

4、编写相应的Controller和Service

```
@Controller
public class HelloController {

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

5、运行主程序测试

访问 localhost:8080/hello

6、简化部署

在pom.xml文件中，导入build插件

```
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>
```

5、HelloWorld深度理解

1.POM.xml文件

1、父项目

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.0.1.RELEASE</version>
    <relativePath/> <!-- lookup parent from repository -->
</parent>
```

这个父项目**spring-boot-starter-parent**又依赖一个父项目

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-dependencies</artifactId>
    <version>2.0.1.RELEASE</version>
    <relativePath>../../spring-boot-dependencies</relativePath>
</parent>
```

下面有个属性，定义了对应的版本号

```
<properties>
    <activemq.version>5.15.3</activemq.version>
    <antlr2.version>2.7.7</antlr2.version>
    <appengine-sdk.version>1.9.63</appengine-sdk.version>
    <artemis.version>2.4.0</artemis.version>
    <aspectj.version>1.8.13</aspectj.version>
    <assertj.version>3.9.1</assertj.version>
    <atomikos.version>4.0.6</atomikos.version>
    <bitronix.version>2.1.4</bitronix.version>
    <build-helper-maven-plugin.version>3.0.0</build-helper-maven-plugin.version>
    。。。。。。。
```

Spring Boot的版本仲裁中心 会自动导入对应的版本，不需要我们自己导入依赖，没有dependencies里面管理的依赖自己声明

2、启动器

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

spring-boot-starter-web:帮我们导入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starter(启动器)，只需要在项目里引入这些starter相关场景的所有依赖都会被导入进来，要用什么功能就导入什么场景的启动器。

2、主程序入口

```
@SpringBootApplication
public class SpringBoot01HelloQuickApplication {

    public static void main(String[] args) {
        SpringApplication.run(SpringBoot01HelloQuickApplication.class, args);
    }
}
```

@SpringBootApplication: 说明这个类是SpringBoot的主配置类，SpringBoot就应该运行这个类的main方法来启动应用

进入SpringBootApplication注解

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

@SpringBootConfiguration：SpringBoot的配置类： 标准在某个类上，表示这是一个SpringBoot的配置类

@Configuration:配置类上，来标注这个注解； 配置类 ---- 配置文件，也是容器中的一个组件（@Component）
@EnableAutoConfiguration:开启自动配置功能 以前需要自动配置的东西，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({AutoConfigurationImportSelector.class})
public @interface EnableAutoConfiguration {
```

@AutoConfigurationPackage:自动配置包 **@Import({Registrar.class})**：底层注解，给容器导入组件； 将主配置类（@SpringBootApplication标注的类）的所在包及下面所有的子包里面的所有组件扫描到Spring容器；

@Import({AutoConfigurationImportSelector.class})： 给容器导入组件?

AutoConfigurationImportSelector：导入组件选择器

将所有需要导入的组件以及全类名的方式返回；这些组件将以字符串数组 String[] 添加到容器中；

会给容器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。

 1.configuration

```
protected List<String> getCandidateConfigurations(AnnotationMetadata metadata,
AnnotationAttributes attributes) {
    List<String> configurations =
SpringFactoriesLoader.loadFactoryNames(this.getSpringFactoriesLoaderFactoryClass(),
this.getBeanClassLoader());
    Assert.notEmpty(configurations, "No auto configuration classes found in META-
INF/spring.factories. If you are using a custom packaging, make sure that file is
correct.");
    return configurations;
}
```

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

Spring Boot在启动的时候从类路径下的META-INF/spring.factorys中获取的EnableAutoConfiguration指定的值；

将这些值作为自动配置类导入到容器中，自动配置就生效了。
 2.factories

J2EE的整体解决方案

org\springframework\boot\spring-boot-autoconfigure\2.0.1.RELEASE\spring-boot-autoconfigure-2.0.1.RELEASE.jar

6、使用Spring Initializer创建一个快速向导

1.IDE支持使用Spring Initializer

自己选择需要的组件:例如web

默认生成的SpringBoot项目

- 主程序已经生成好了，我们只需要完成我们的逻辑
- resources文件夹目录结构
  - static:保存所有的静态文件；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）；可以使用模板引擎（freemarker.thymeleaf）；
  - application.properties:Spring Boot的默认配置，例如 server.port=9000

二、配置文件

1、配置文件

Spring Boot使用全局配置文件，配置文件名是固定的；

- application.properties
- application.yml

配置文件作用：修改Spring Boot在底层封装好的默认值；

YAML（YAML AIN'T Markup Language）

是一个标记语言

又不是一个标记语言

标记语言：

以前的配置文件；大多数使用的是 xxx.xml文件；

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: 9000
```

XML:

```
<server>
    <port>9000</port>
</server>
```

2、YAML语法

1、基本语法

k:(空格)v:表示一堆键值对（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，都是同一层级的

```
server:
    port: 9000
    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** 不会转义字符串里的特殊字符；特殊字符会作为本身想要表示的意思

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

'':**单引号** 会转义特殊字符，特殊字符最终只是一个普通的字符串数据

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

对象、Map（属性和值）键值对

k :v ： 在下一行来写对象的属性和值的关系；注意空格控制缩进

对象还是k:v的方式

```
frends:
    lastName: zhangsan
    age: 20
```

行内写法

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

数组（List、Set）：用-表示数组中的一个元素

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

行内写法

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

组合变量

多个组合到一起

3、配置文件值注入

1、@ConfigurationProperties

1、application.yml 配置文件

```
person:
  age: 18
  boss: false
  birth: 2017/12/12
  maps: {k1: v1,k2: 12}
  lists:
   - lisi
   - zhaoliu
  dog:
    name: wangwang
    age: 2
  last-name: wanghuahua
```

`application.properties` 配置文件（二选一）

```
idea配置文件utf-8
properties 默认GBK
person.age=12
person.boss=false
person.last-name=张三
person.maps.k1=v1
person.maps.k2=v2
person.lists=a,b,c
person.dog.name=wanghuahu
person.dog.age=15
```

所以中文输出乱码，改进settings-->file encoding -->[property-->utf-8 ,勾选转成ascii]

javaBean

```
/**
* 将配置文件的配置每个属性的值，映射到组件中
* @ConfigurationProperties:告诉SpringBoot将文本的所有属性和配置文件中的相关配置进行绑定;
* prefix = "person" 配置文件爱你的那个属性进行一一映射
* *
只有这个组件是容器中的组件，才能提供到容器中
*/
@Component
@ConfigurationProperties(prefix = "person")
public class Person {
```

```
    private String lastName;
    private Integer age;
    private Boolean boss;
    private Map<String,Object> maps;
    private List<Object> lists;
    private Dog dog;
```

导入配置文件处理器，以后编写配置就有提示了

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-configuration-processor</artifactId>
    <optional>true</optional>
</dependency>
```

2、@Value注解

更改javaBean中的注解

```
@Component
public class Person {
    /**
     * <bean class="Person">
     *     <property name="lastName" value="字面量/${key}从环境变量/#{spEL}"></property>
     * </bean>
     */
    @Value("${person.last-name}")
    private String lastName;
    @Value("#{11*2}")
    private Integer age;
    @Value("true")
    private Boolean boss;
```

	@ConfigurationProperties	@Value
功能	批量注入配置文件属性	单个指定
松散绑定(语法)	支持	不支持
spEL	不支持	支持
JSR303校验	支持	不支持
复杂类型	支持	不支持

> 松散语法：javaBean中last-name(或者lastName) -->application.properties中的last-name;
>
> spEL语法：#{11*2}

> JSR303：@Value会直接忽略，校验规则

JSR303校验：

```
@Component
@ConfigurationProperties(prefix = "person")
@Validated
public class Person {
    @Email
    private String lastName;
```

复杂类型栗子：

```
@Component
public class Person {
    /**
     * <bean class="Person">
     *     <property name="lastName" value="字面量/${key}从环境变量/#{spEL}"></property>
     * </bean>
     */
    private String lastName;
    private Integer age;
    private Boolean boss;
   // @Value("${person.maps}")
    private Map<String,Object> maps;
```

以上会报错，不支持复杂类型

使用场景分析

如果说，我们只是在某个业务逻辑中获取一下配置文件的某一项值，使用@Value；

如果专门编写了一个javaBean和配置文件进行映射，我们直接使用@ConfigurationProperties

举栗子：

1、编写新的Controller文件

```
@RestController
public class HelloController {

    @Value("${person.last-name}")
    private String name;
    @RequestMapping("/hello")
    public  String sayHello(){
        return "Hello"+ name;
    }
}
```

2、配置文件

```
person.age=12
person.boss=false
person.last-name=李四
person.maps.k1=v1
person.maps.k2=v2
person.lists=a,b,c
person.dog.name=wanghuahu
person.dog.age=15
```

3、测试运行

访问 localhost:9000/hello

结果为`Hello 李四`

3、其他注解

@PropertySource

作用：加载指定的properties配置文件

1、新建一个person.properties文件

```
person.age=12
person.boss=false
person.last-name=李四
person.maps.k1=v1
person.maps.k2=v2
person.lists=a,b,c
person.dog.name=wanghuahu
person.dog.age=15
```

2、在javaBean中加入@PropertySource注解

```
@PropertySource(value = {"classpath:person.properties"})
@Component
@ConfigurationProperties(prefix = "person")
public class Person {
    private String lastName;
```

@ImportResource

作用：导入Spring配置文件，并且让这个配置文件生效

1、新建一个Spring的配置文件，bean.xml

```
<?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.wdjr.springboot.service.HelloService"></bean>
</beans>
```

2、编写测试类，检查容器是否加载Spring配置文件写的bean

```
@Autowired
ApplicationContext ioc;

@Test
public void testHelloService(){
    boolean b = ioc.containsBean("HelloService");
    System.out.println(b);
}
```

> import org.springframework.context.ApplicationContext;

3、运行检测

结果为false，没有加载配置的内容

4、使用@ImportResource注解

将@ImportResource标注在主配置类上

```
@ImportResource(locations={"classpath:beans.xml"})
@SpringBootApplication
public class SpringBoot02ConfigApplication {

    public static void main(String[] args) {
        SpringApplication.run(SpringBoot02ConfigApplication.class, args);
    }
}
```

5、再次运行检测

结果为true

缺点：每次指定xml文件太麻烦

SpringBoot推荐给容器添加组件的方式：

1、配置类=====Spring的xml配置文件（old）

2、全注解方式@Configuration+@Bean（new）

```
/**
 * @Configuration：指明当前类是一个配置类；就是来代替之前的Spring配置文件
 *
 * 在配置文件中用<bean></bean>标签添加组件
 */

@Configuration
public class MyAppConfig {

    //将方法的返回值添加到容器中；容器这个组件id就是方法名
    @Bean
    public HelloService helloService01(){
        System.out.println("配置类给容器添加了HelloService组件");
        return new HelloService();
    }
}
```

```
@Autowired
ApplicationContext ioc;

@Test
public void testHelloService(){
    boolean b = ioc.containsBean("helloService01");
    System.out.println(b);
}
```

容器这个组件id就是方法名

4、配置文件占位符

1、随机数

```
${random.value} 、${random.int}、${random.long}
${random.int(10)}、${random.int[100,200]}
```

2、获取配置值

```
person.age=${random.int}
person.boss=false
person.last-name=张三${random.uuid}
person.maps.k1=v1
person.maps.k2=v2
person.lists=a,b,c
person.dog.name=${person.last-name}'s wanghuahu
person.dog.age=15
```

存在以下两种情况

没有声明`person.last-name`会报错，新声明的需要加默认值

```
person.age=${random.int}
person.boss=false
person.last-name=张三${random.uuid}
person.maps.k1=v1
person.maps.k2=v2
person.lists=a,b,c
person.dog.name=${person.hello:hello}'s wanghuahu
person.dog.age=15
```

结果：输出`hello's wanghuahua`

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

- application.properties
- application-dev.properties
- application-prod.properties

默认使用application.properties

application.properties配置文件指定

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

2、YAML文档块

```
server:
  port: 8081
spring:
  profiles:
    active: dev
```

```
---

server:
  port: 9000
spring:
  profiles: dev

---
server:
  port: 80
spring:
  profiles: prod
```

3、激活指定profile

1、在配置文件中激活

2、命令行：

--spring.profiles.active=dev

 5.comandLine

优先级大于配置文件

打包 成jar后

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

虚拟机参数

`-Dspring.profiles.active=dev`

6、加载配置文件位置

SpringBoot启动扫描以下位置的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:/

优先级从高到低顺序，高优先级会覆盖低优先级的相同配置；互补配置

也可以通过spring.config.location来改变默认配置

> ```
> server.servlet.context-path=/boot03
> ```

注：spring boot1x 是server.context.path=/boot02

 7.priority

还可以通过spring.config.location来改变配置文件的位置

项目打包好了以后，可以使用命令行参数的形式，启动项目的时候来指定配置文件的新位置；指定配置文件和默认的配置文件会共同起作用，互补配置

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

运维比较有用，从外部加载，不用修改别的文件

7.引入外部配置

SpringBoot也可以从以下位置加载配置；优先级从高到低；高优先级覆盖低优先级，可以互补

1. 命令行参数

   java -jar spring-boot-config-02-0.0.1-SNAPSHOT.jar --server.port=9005 --server.context-path=/abc

   中间一个空格

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

   优先加载profile, 由jar包外到jar包内

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**
8. **jar包外部的application.properties或application.yml(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注解类的@PropertySource
11. 通过SpringApplication.setDefaultProperties指定的默认属性

官方文档

8、自动配置

配置文件到底怎么写?

Spring的所有配置参数

自动配置原理很关键

1、自动配置原理

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

2）、@EnableAutoConfiguration 作用:

- 利用AutoConfigurationImportSelector给容器中导入一些组件?

- 可以查看selectImports()方法的内容
- 获取候选的配置

```
List<String> configurations = this.getCandidateConfigurations(annotationMetadata,
attributes);
```

- 扫描类路径下的

```
  SpringFactoriesLoader.loadFactoryNames()
  扫描所有jar包类路径下的 MATA-INF/spring.factories
  把扫描到的这些文件的内容包装成properties对象
  从properties中获取到EnableAutoConfiguration.class类（类名）对应的值，然后把他们添加到容器中
```

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

3）、每一个自动配置类进行自动配置功能；

4）、以**HttpEncodingAutoConfiguration** 为例

```
@Configuration //表示是一个配置类，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；将配置文件中的值和HttpEncodingProperties绑定起来了；并把
HttpEncodingProperties加入ioc容器中
@ConditionalOnWebApplication//根据不同的条件，进行判断，如果满足条件，整个配置类里面的配置就会失效，判断
是否为web应用；
(
    type = Type.SERVLET
)
@ConditionalOnClass({CharacterEncodingFilter.class})//判断当前项目有没有这个类，解决乱码的过滤器
@ConditionalOnProperty(
    prefix = "spring.http.encoding",
    value = {"enabled"},
    matchIfMissing = true
)//判断配置文件是否存在某个配置 spring.http.encoding，matchIfMissing = true如果不存在也是成立，即使不
配置也生效
public class HttpEncodingAutoConfiguration {
   //给容器添加组件，这个组件的值需要从properties属性中获取
    private final HttpEncodingProperties properties;
    //只有一个有参数构造器情况下，参数的值就会从容器中拿
    public HttpEncodingAutoConfiguration(HttpEncodingProperties properties) {
        this.properties = properties;
    }

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

filter.setForceRequestEncoding(this.properties.shouldForce(org.springframework.boot.autoconf
```

```
igure.http.HttpEncodingProperties.Type.REQUEST));

filter.setForceResponseEncoding(this.properties.shouldForce(org.springframework.boot.autocon
figure.http.HttpEncodingProperties.Type.RESPONSE));
        return filter;
    }
```

5）、所有在配置文件中能配置的属性都是在xxxProperties类中封装着；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")//从配置文件中的值进行绑定和bean属性进行绑定
public class HttpEncodingProperties {
```

根据当前不同条件判断，决定这个配置类是否生效?

一旦这个配置类生效；这个配置类会给容器添加各种组件；这些组件的属性是从对应的properties中获取的，这些类里面的每个属性又是和配置文件绑定的

2、所有的自动配置组件

每一个xxxAutoConfiguration这样的类都是容器中的一个组件，都加入到容器中；

作用：用他们做自动配置

```
# 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.CassandraReactiveDataAutoConfiguration
,\
org.springframework.boot.autoconfigure.data.cassandra.CassandraReactiveRepositoriesAutoConfi
guration,\
org.springframework.boot.autoconfigure.data.cassandra.CassandraRepositoriesAutoConfiguration
,\
org.springframework.boot.autoconfigure.data.couchbase.CouchbaseDataAutoConfiguration,\
```

```
org.springframework.boot.autoconfigure.data.couchbase.CouchbaseReactiveDataAutoConfiguration
,\
org.springframework.boot.autoconfigure.data.couchbase.CouchbaseReactiveRepositoriesAutoConfi
guration,\
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.ElasticsearchRepositoriesAutoConfi
guration,\
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.MongoReactiveDataAutoConfiguration,\
org.springframework.boot.autoconfigure.data.mongo.MongoReactiveRepositoriesAutoConfiguration
,\
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.RedisReactiveAutoConfiguration,\
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.flyway.FlywayAutoConfiguration,\
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.http.HttpMessageConvertersAutoConfiguration,\
org.springframework.boot.autoconfigure.http.codec.CodecsAutoConfiguration,\
org.springframework.boot.autoconfigure.influx.InfluxDbAutoConfiguration,\
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.groovy.template.GroovyTemplateAutoConfiguration,\
org.springframework.boot.autoconfigure.jersey.JerseyAutoConfiguration,\
```

```
org.springframework.boot.autoconfigure.jooq.JooqAutoConfiguration,\
org.springframework.boot.autoconfigure.jsonb.JsonbAutoConfiguration,\
org.springframework.boot.autoconfigure.kafka.KafkaAutoConfiguration,\
org.springframework.boot.autoconfigure.ldap.embedded.EmbeddedLdapAutoConfiguration,\
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.mongo.embedded.EmbeddedMongoAutoConfiguration,\
org.springframework.boot.autoconfigure.mongo.MongoAutoConfiguration,\
org.springframework.boot.autoconfigure.mongo.MongoReactiveAutoConfiguration,\
org.springframework.boot.autoconfigure.mustache.MustacheAutoConfiguration,\
org.springframework.boot.autoconfigure.orm.jpa.HibernateJpaAutoConfiguration,\
org.springframework.boot.autoconfigure.quartz.QuartzAutoConfiguration,\
org.springframework.boot.autoconfigure.reactor.core.ReactorCoreAutoConfiguration,\
org.springframework.boot.autoconfigure.security.servlet.SecurityAutoConfiguration,\
org.springframework.boot.autoconfigure.security.servlet.UserDetailsServiceAutoConfiguration,
\
org.springframework.boot.autoconfigure.security.servlet.SecurityFilterAutoConfiguration,\
org.springframework.boot.autoconfigure.security.reactive.ReactiveSecurityAutoConfiguration,\
org.springframework.boot.autoconfigure.security.reactive.ReactiveUserDetailsServiceAutoConfi
guration,\
org.springframework.boot.autoconfigure.sendgrid.SendGridAutoConfiguration,\
org.springframework.boot.autoconfigure.session.SessionAutoConfiguration,\
org.springframework.boot.autoconfigure.security.oauth2.client.OAuth2ClientAutoConfiguration,
\
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.client.RestTemplateAutoConfiguration,\
org.springframework.boot.autoconfigure.web.embedded.EmbeddedWebServerFactoryCustomizerAutoCo
nfiguration,\
org.springframework.boot.autoconfigure.web.reactive.HttpHandlerAutoConfiguration,\
org.springframework.boot.autoconfigure.web.reactive.ReactiveWebServerFactoryAutoConfiguratio
n,\
org.springframework.boot.autoconfigure.web.reactive.WebFluxAutoConfiguration,\
org.springframework.boot.autoconfigure.web.reactive.error.ErrorWebFluxAutoConfiguration,\
org.springframework.boot.autoconfigure.web.reactive.function.client.WebClientAutoConfigurati
on,\
org.springframework.boot.autoconfigure.web.servlet.DispatcherServletAutoConfiguration,\
org.springframework.boot.autoconfigure.web.servlet.ServletWebServerFactoryAutoConfiguration,
\
org.springframework.boot.autoconfigure.web.servlet.error.ErrorMvcAutoConfiguration,\
org.springframework.boot.autoconfigure.web.servlet.HttpEncodingAutoConfiguration,\
org.springframework.boot.autoconfigure.web.servlet.MultipartAutoConfiguration,\
org.springframework.boot.autoconfigure.web.servlet.WebMvcAutoConfiguration,\
org.springframework.boot.autoconfigure.websocket.reactive.WebSocketReactiveAutoConfiguration
,\
org.springframework.boot.autoconfigure.websocket.servlet.WebSocketServletAutoConfiguration,\
org.springframework.boot.autoconfigure.websocket.servlet.WebSocketMessagingAutoConfiguration
,\
```

```
org.springframework.boot.autoconfigure.webservices.WebServicesAutoConfiguration
```

3、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默认写好的默认配置类；

3）、如果有在看这个自动配置类中配置了哪些组件；（只要我们要用的组件有，我们需要再来配置）

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

xxxAutoConfiguration:自动配置类；

给容器中添加组件

xxxProperties:封装配置文件中的属性；

跟之前的Person类一样，配置文件中值加入bean中

4、细节

1、@Conditional派生注解

> 利用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派生注解	**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
@ConditionalOnMissBean	容器中不存在指定Bean
@ConditionalOnExpression	满足spEL表达式
@ConditionalOnClass	系统中有指定的类
@ConditionalOnMissClass	系统中没有指定的类
@ConditionalOnSingleCandidate	容器中只有一个指定的Bean,或者这个Bean是首选Bean
@ConditionalOnProperty	系统中指定的属性是否有指定的值
@ConditionalOnResource	类路径下是否存在指定的资源文件
@ConditionalOnWebApplication	当前是web环境
@ConditionalOnNotWebApplication	当前不是web环境
@ConditionalOnJndi	JNDI存在指定项

2、自动配置报告

自动配置类必须在一定条件下生效

我们可以通过启用debug=true属性，配置文件，打印自动配合报告，这样就可以知道自动配置类生效

```
debug=true
```

自动配置报告

```
=========================

CONDITIONS EVALUATION REPORT
=========================


Positive matches:（启动的，匹配成功的）
-----------------

   CodecsAutoConfiguration matched:
      - @ConditionalOnClass found required class
'org.springframework.http.codec.CodecConfigurer'; @ConditionalOnMissingClass did not find
unwanted class (OnClassCondition)
        ......

 Negative matches:（没有启动的，没有匹配成功的）
-----------------

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

三、日志

> Spring Boot2对日志有更改

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？卸载文件中

2、框架记录系统的一些运行信息；日志框架zhanglog.jar

3、高大上功能，异步模式？自动归档？xxx?zhanglog-good.jar?

4、将以前的框架卸下来？换上新的框架，重新修改之前的相关API;zhanglog-perfect.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；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

日志抽象层	日志实现
~~JCL(Jakarta Commons Logging)~~ SLF4j(Simple Logging Facade for Java) ~~jboss-logging~~	Log4j ~~JUL(java.util.logging)~~ Log4j2 Logback

左边的抽象，右边的实现

SLF4J -- Logback

Spring Boot:底层是Spring框架，Spring默认框架是JCL；

SpringBoot选用SLF4j和logback

2、SLF4J使用

1、如何在系统中使用SLF4j

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；

应该给系统里面导入slf4j的jar包和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");
  }
}
```


8.slf4j

每个日志框架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本身的配置文件**；

2、遗留问题

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

统一日志框架，即使是别的框架和我一起统一使用slf4j进行输出；

核心：

1、将系统中其他日志框架排除出去；

2、用中间包来替换原有的日志框架/

3、导入slf4j的其他实现

3、SpingBoot日志框架解析

打开IDEA，打开pom文件的依赖图形化显示

 9.IDEAdependencies

SpringBoot的基础框架

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

SpringBoot的日志功能

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-logging</artifactId>
    <version>2.0.1.RELEASE</version>
    <scope>compile</scope>
</dependency>
```

 10.slf4jandlogback

总结：

1. SpringBoot底层也是使用SLF4J+log4jback
2. SpringBoot也把其他日志替换成了slf4j
3. 起着commons.loggings的名字其实new的SLF4J替换中间包

   SpringBoot2中改成了bridge
4. 如果要引入其他框架？一定要把这个框架的日志依赖移除掉，而且底层

4、日志的使用

1、默认配置

trace-debug-info-warn-error

可以调整需要的日志级别进行输出，不用注释语句。

```
//记录器
Logger logger = LoggerFactory.getLogger(getClass());
@Test
public void contextLoads() {

    //日志的级别
```

```
    //从低到高
    //可以调整输出的日志级别；日志就只会在这个级别以后的高级别生效
    logger.trace("这是trace日志");
    logger.debug("这是debug信息");
    //SpringBoot默认给的是info级别，如果没指定就是默认的root级别
    logger.info("这是info日志");
    logger.warn("这是warn信息");
    logger.error("这是Error信息");
}
```

调整指定包的日志级别在配置文件中进行配置

```
logging.level.com.wdjr=trace
```

日志输出格式

```
#控制台输出的日志格式
#%d：日期
#%thread：线程号
#%-5level：靠左 级别
#%logger{50}：全类名50字符限制,否则按照句号分割
#%msg：消息+换行
#%n：换行
logging.pattern.console=%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.wdjr=trace
#不指定path就是当前目录下生成springboot.log
#logging.file=springboot.log
#当前磁盘下根路径创建spring文件中log文件夹，使用spring.log作为默认
logging.path=/spring/log
#控制台输出的日志格式 日期 + 线程号 + 靠左 级别 +全类名50字符限制+消息+换行
logging.pattern.console=%d{yyyy-MM-dd HH:mm:ss.SSS} [%thread] %-5level %logger{50} - %msg%n
#指定文件中日志输出的格式
logging.pattern.file=xxx
```

2、指定配置

给类路径下放上每个日志框架自己的配置框架；SpringBoot就不会使用自己默认的配置

logging System	Customization
Logback	logback-spring.xml ,logback-spring.groovy,logback.xml or logback.groovy
Log4J2	log4j2-spring.xml or log4j2.xml
JDK(Java Util Logging)	logging.properties

logback.xml直接被日志框架识别， logback-spring.xml日志框架就不直接加载日志配置项，由SpringBoot加载

```
<springProfile name="dev">
    <!-- 可以指定某段配置只在某个环境下生效 -->
</springProfile>
<springProfile name!="dev">
    <!-- 可以指定某段配置只在某个环境下生效 -->
</springProfile>
```

如何调试开发环境,输入命令行参数

--spring.profiles.active=dev

如果不带后面的xx-spring.xml就会报错

3、切换日志框架

可以根据slf4j的日志适配图，进行相关切换；

1、log4j

slf4j+log4j的方式；


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

<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-log4j12</artifactId>
</dependency>
```

不推荐使用仅作为演示

2、log4j2

切换为log4j2

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

四、web开发

1、简介

使用SpringBoot;

1)、创建SpringBoot应用，选中我们需要的模块；

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?

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

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

2、静态资源文件映射规则

```
@ConfigurationProperties(prefix = "spring.resources", ignoreUnknownFields = false)
public class ResourceProperties implements ResourceLoaderAware, InitializingBean {
    //可以设置和静态资源相关的参数，缓存时间等
```

```
@Override
public void addResourceHandlers(ResourceHandlerRegistry registry) {
   if (!this.resourceProperties.isAddMappings()) {
```

```
            logger.debug("Default resource handling disabled");
            return;
        }
        Integer cachePeriod = this.resourceProperties.getCachePeriod();
        if (!registry.hasMappingForPattern("/webjars/**")) {
            customizeResourceHandlerRegistration(registry
                    .addResourceHandler("/webjars/**")
                    .addResourceLocations("classpath:/META-INF/resources/webjars/")
                    .setCachePeriod(cachePeriod));
        }
        String staticPathPattern = this.mvcProperties.getStaticPathPattern();
        if (!registry.hasMappingForPattern(staticPathPattern)) {
            customizeResourceHandlerRegistration(
                    registry.addResourceHandler(staticPathPattern)
                            .addResourceLocations(
                                    this.resourceProperties.getStaticLocations())
                            .setCachePeriod(cachePeriod));
        }
    }
```

1、webjar

1)、所有的/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/

 12.jquery

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

2、本地资源

```
private String staticPathPattern = "/**";
```

访问任何资源

2、会在这几文件夹下去找静态路径（静态资源文件夹）

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

 13.static

localhost:8080/abc ==>去静态资源文件夹中找abc

 14.static-css

3、index页面欢迎页，静态资源文件夹下所有的index.html页面；被“/**”映射；

localhost:8080/ -->index页面

```
@Bean
public WelcomePageHandlerMapping welcomePageHandlerMapping(
      ResourceProperties resourceProperties) {
   return new WelcomePageHandlerMapping(resourceProperties.getWelcomePage(),
         this.mvcProperties.getStaticPathPattern());
}
```

4、喜欢的图标，即网站title的图标favicon

```
@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);
       //把任何favicon的图标都在静态文件夹下找
      mapping.setUrlMap(Collections.singletonMap("**/favicon.ico",
            faviconRequestHandler()));
      return mapping;
   }

   @Bean
   public ResourceHttpRequestHandler faviconRequestHandler() {
      ResourceHttpRequestHandler requestHandler = new ResourceHttpRequestHandler();
      requestHandler
            .setLocations(this.resourceProperties.getFaviconLocations());
      return requestHandler;
   }

}
```

可以在配置文件配置静态资源文件夹

```
spring.resources.static-locations=classpath:xxxx
```

3、模板引擎

将html和数据 结合到一起 输出组装处理好的新文件

SpringBoot推荐Thymeleaf;语法简单，功能强大

1、引入thymeleaf 3

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

默认导入thymeleaf2，版本太低 所以使用thymeleaf3.

官方导入办法

```
<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
    <java.version>1.8</java.version>
    <!--thymeleaf 3的导入-->
    <thymeleaf.version>3.0.9.RELEASE</thymeleaf.version>
    <!--布局功能支持 同时支持thymeleaf3主程序 layout2.0以上版本  -->
    <!--布局功能支持 同时支持thymeleaf2主程序 layout1.0以上版本  -->
    <thymeleaf-layout-dialect.version>2.2.2</thymeleaf-layout-dialect.version>
</properties>
```

2、Thymeleaf使用和语法

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

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

   private static final MimeType DEFAULT_CONTENT_TYPE = MimeType.valueOf("text/html");

   public static final String DEFAULT_PREFIX = "classpath:/templates/";

   public static final String DEFAULT_SUFFIX = ".html";
   //只要把HTML文件方法类路径下的template文件夹下，就会自动导入
```

只要把HTML页面放到classpath:/templates/,thymeleaf就能自动渲染;

使用:

1、导入thymeleaf的名称空间

```
<html 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>success</title>
</head>
<body>
<h1>success</h1>
<!--th:text 将div里面的文本内容设置为-->
<div th:text="${Lion}">
前端数据
</div>
</body>
</html>
```

3、语法规则

1）、th:text="${hello}"可以使用任意标签 替换原生的任何属性

在SpringBoot的环境下

```
<div id="testid" class="testcalss" th:id="${Lion}" th:class="${Lion}" th:text="${Lion}">
    前端数据
</div>
```

 15.thtmeleaf-th01

直接访问HTML页面

 15.thtmeleaf-th02

2)、内联写法注意需要在body上加上 th:inline="text"敲黑板

不然不起作用

```
<body class="text-center" th:inline="text"></body>
```

th标签的访问优先级

Order Feature Attributes

3、语法规则

	功能	标签	功能和jsp对比
1	Fragment inclusion	th:insert th:replace	include(片段包含)
2	Fragment iteration	th:each	c:forEach(遍历)
3	Conditional evaluation	th:if th:unless th:switch th:case	c:if(条件判断)
4	Local variable definition	th:object th:with	c:set(声明变量)
5	General attribute modification	th:attr th:attrprepend th:attrappend	属性修改支持前面和后面追加内容
6	Specific attribute modification	th:value th:href th:src ...	修改任意属性值
7	Text (tag body modification)	th:text th:utext	修改标签体内容utext：不转义字符 # 大标题
8	Fragment specification	th:fragment	声明片段
9	Fragment removal	th:remove	

```
Simple expressions:(表达式语法)
    Variable Expressions: ${...}
        1、获取对象属性、调用方法
        2、使用内置基本对象:
            #ctx : the context object.
            #vars: the context variables.
            #locale : the context locale.
            #request : (only in Web Contexts) the HttpServletRequest object.
            #response : (only in Web Contexts) the HttpServletResponse object.
            #session : (only in Web Contexts) the HttpSession object.
            #servletContext : (only in Web Contexts) the ServletContext object.
         3、内置一些工具对象
            #execInfo : information about the template being processed.
            #messages : methods for obtaining externalized messages inside variables
expressions, in the same way as they
            would be obtained using #{…} syntax.
            #uris : methods for escaping parts of URLs/URIs
            #conversions : methods for executing the configured conversion service (if any).
            #dates : methods for java.util.Date objects: formatting, component extraction,
etc.
            #calendars : analogous to #dates , but for java.util.Calendar objects.
            #numbers : methods for formatting numeric objects.
            #strings : methods for String objects: contains, startsWith,
prepending/appending, etc.
            #objects : methods for objects in general.
            #bools : methods for boolean evaluation.
```

```
            #arrays : methods for arrays.
            #lists : methods for lists.
            #sets : methods for sets.
            #maps : methods for maps.
            #aggregates : methods for creating aggregates on arrays or collections.
            #ids : methods for dealing with id attributes that might be repeated (for
example, as a result of an iteration).
    Selection Variable Expressions: *{...} //选择表达式：和${}功能一样，补充功能
   # 配合th:object使用，object=${object} 以后获取就可以使用*{a}  相当于${object.a}
        <div th:object="${session.user}">
            <p>Name: <span th:text="*{firstName}">Sebastian</span>.</p>
            <p>Surname: <span th:text="*{lastName}">Pepper</span>.</p>
            <p>Nationality: <span th:text="*{nationality}">Saturn</span>.</p>
        </div>
    Message Expressions: #{...} //获取国际化内容
    Link URL Expressions: @{...} //定义URL链接
        #<a href="details.html" th:href="@{/order/details(orderId=${o.id})}">view</a>
    Fragment Expressions: ~{...}//片段文档

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

inline写法

```
[[]] -->th:text
[()] -->th:utext
```

4、SpringMVC自动配置

1、SpringMVC的自动导入

Spring框架

自动配置好了mvc：

以下是SpringBoot对SpringMVC的默认

Spring Boot provides auto-configuration for Spring MVC that works well with most applications.

The auto-configuration adds the following features on top of Spring's defaults:

- 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`组合所有视图解析器
  - 如何定制：我们可以自己给容器中添加一个视图解析器；自动将其整合进来
- Support for serving static resources, including support for WebJars (see below).静态资源
- Static `index.html` support.
- Custom `Favicon` support (see below).
- 自动注册 了`Converter`, `GenericConverter`, `Formatter` beans.
  - `Converter`：类型转换 文本转为字面量
  - `Formatter` ： 格式化器 转换后格式转换

    ```
    @Bean
    @ConditionalOnProperty(prefix = "spring.mvc", name = "date-format")//在文件配置入职格式化的规则
    public Formatter<Date> dateFormatter() {
       return new DateFormatter(this.mvcProperties.getDateFormat());//日期格式化组件
    }
    ```

    自己添加的格式化转换器，只需要放在容器中即可
- Support for `HttpMessageConverters` (see below).
  - `HttpMessageConverters` ： 转换HTTP转换和响应：User - json
  - `HttpMessageConverters` ： 是从容器中确定；获取所有的`HttpMessageConverters` ， 将自己的组件注册在容器中@Bean
  - If you need to add or customize converters you can use Spring Boot's `HttpMessageConverters` class:

    ```
    import org.springframework.boot.autoconfigure.web.HttpMessageConverters;
    import org.springframework.context.annotation.*;
    import org.springframework.http.converter.*;

    @Configuration
    ```

```
public class MyConfiguration {

    @Bean
    public HttpMessageConverters customConverters() {
        HttpMessageConverter<?> additional = ...
        HttpMessageConverter<?> another = ...
        return new HttpMessageConverters(additional, another);
    }

}
```

- Automatic registration of `MessageCodesResolver` (see below).
  - 定义错误代码生成规则
- Automatic use of a `ConfigurableWebBindingInitializer` bean (see below).
  - ```
    @Override
    protected ConfigurableWebBindingInitializer getConfigurableWebBindingInitializer() {
       try {
          return this.beanFactory.getBean(ConfigurableWebBindingInitializer.class);
       }
       catch (NoSuchBeanDefinitionException ex) {
          return super.getConfigurableWebBindingInitializer();
       }
    }
    ```

    在beanFactory：中可以自己创建一个，初始化webDataBinder

    请求数据 ==》javaBean

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`.

思想：修改默认配置

2、扩展SpringMVC

编写一个配置类，类型是WebMvcConfigurerAdapter(继承)，使用WebMvcConfigurerAdapter可以扩展，不能标注@EnableWebMvc;既保留了配置，也能拓展我们自己的应用
    ```

```
@Configuration
public class MyMvcConfig extends WebMvcConfigurerAdapter {

    @Override
    public void addViewControllers(ViewControllerRegistry registry) {
//        super.addViewControllers(registry);
        //浏览器发送wdjr请求，也来到success页面
        registry.addViewController("/wdjr").setViewName("success");
    }
}
```

原理：

1）、WebMvcAutoConfiguration是SpringMVC的自动配置

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webMVCconfigurer
    @Autowired(required = false)
    public void setConfigurers(List<WebMvcConfigurer> configurers) {
        if (!CollectionUtils.isEmpty(configurers)) {
            this.configurers.addWebMvcConfigurers(configurers);

                @Override
                protected void addViewControllers(ViewControllerRegistry registry) {
                    this.configurers.addViewControllers(registry);
                }
            //一个参考实现,将所有的webMVCconfigurer相关配置一起调用（包括自己的配置类）
                @Override
               // public void addViewControllers(ViewControllerRegistry registry) {
                   // for (WebMvcConfigurer delegate : this.delegates) {
                  //delegate.addViewControllers(registry);
                    //}
                }
        }
    }
```

3）、自己的配置被调用

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mvc

不需要SpringBoot对SpringMVC的自动配置。

```
@EnableWebMvc
@Configuration
public class MyMvcConfig extends WebMvcConfigurerAdapter {

@Override
public void addViewControllers(ViewControllerRegistry registry) {


//        super.addViewControllers(registry);
        //浏览器发送wdjr请求，也来到success页面
        registry.addViewController("/wdjr").setViewName("success");
    }
}
```

例如静态资源访问，不推荐全面接管

原理：

为什么@EnableWebMvc注解，SpringBoot对SpringMVC的控制就失效了

1）、核心配置

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

2）、DelegatingWebMvcConfiguration

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

3）、WebMvcAutoConfiguration

```
@Configuration
@ConditionalOnWebApplication
@ConditionalOnClass({ Servlet.class, DispatcherServlet.class,
      WebMvcConfigurerAdapter.class })
//容器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导入进来了；

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

5、修改SpringMVC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 xxxConfigurer帮助我们扩展配置。

6、RestfulCRUD

1、默认访问首页

在config/MyConfig.java中编写配置类

```
//所有的webMvcConfigurerAdapter组件会一起起作用
@Bean //註冊到容器去
public WebMvcConfigurerAdapter webMvcConfigurerAdapter(){
    WebMvcConfigurerAdapter adapter = new WebMvcConfigurerAdapter() {
        @Override
        public void addViewControllers(ViewControllerRegistry registry) {
            registry.addViewController("/").setViewName("login");
            registry.addViewController("/login.html").setViewName("login");
        }
    };
    return adapter;
}
```

静态资源引用

```
<link href="#" th:href="@{/css/signin.css}" rel="stylesheet" />
```

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、在页面中使用fmt:message，取出国际化内容

1、浏览器切换国际化

步骤

1、编写国际化配置文件，抽取页面需要的显示的国际化消息

 16.national

2、SpringBoot自动配置好了国际化配置的资源文件

```
@ConfigurationProperties(prefix = "spring.messages")
public class MessageSourceAutoConfiguration {
    //我们的配置文件可以直接放在类路径下叫messages.properties
    private String basename = "messages";
    @Bean
    public MessageSource messageSource() {
        ResourceBundleMessageSource messageSource = new ResourceBundleMessageSource();
        if (StringUtils.hasText(this.basename)) {
            //设置国际化文件的基础名，去掉语言国家代码
            messageSource.setBasenames(StringUtils.commaDelimitedListToStringArray(
                    StringUtils.trimAllWhitespace(this.basename)));
        }
        if (this.encoding != null) {
            messageSource.setDefaultEncoding(this.encoding.name());
        }
        messageSource.setFallbackToSystemLocale(this.fallbackToSystemLocale);
        messageSource.setCacheSeconds(this.cacheSeconds);
        messageSource.setAlwaysUseMessageFormat(this.alwaysUseMessageFormat);
        return messageSource;
    }
```

3、对IDEA的编码进行设置

 17.encoding

4、login进行标签插入

```
<!DOCTYPE html>
<!-- saved from url=(0051)https://getbootstrap.com/docs/4.1/examples/sign-in/ -->
<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="" />
    <link rel="icon" href="https://getbootstrap.com/favicon.ico" />

    <title>登录页面</title>

    <!-- Bootstrap core CSS -->
    <link href="#" th:href="@{/css/bootstrap.min.css}" rel="stylesheet" />

    <!-- Custom styles for this template -->
    <link href="./login_files/signin.css" th:href="@{/css/signin.css}" rel="stylesheet" />
  </head>

  <body class="text-center">
    <form class="form-signin">
      <img class="mb-4" src="./login_files/bootstrap-solid.svg" th:src="@{/img/bootstrap-
```

```
solid.svg}" alt="" width="72" height="72" />
      <h1 class="h3 mb-3 font-weight-normal" th:text="#{login.tip}">Please sign in</h1>
      <label  class="sr-only" th:text="#{login.username}">Username</label>
      <input type="text"  name="username" class="form-control" placeholder="Username"
th:placeholder="#{login.username}" required="" autofocus=""/>
      <label for="inputPassword" class="sr-only" th:text="#
{login.password}">Password</label>
      <input type="password" name="password" id="inputPassword" class="form-control"
placeholder="Password" th:placeholder="#{login.password}" required="" />
      <div class="checkbox mb-3">
        <label>
          <input type="checkbox" value="remember-me" /> [[#{login.remember}]]
        </label>
      </div>
      <button class="btn btn-lg btn-primary btn-block" type="submit" th:text="#
{login.btn}">Sign in</button>
      <p class="mt-5 mb-3 text-muted">© 2017-2018</p>
    </form>


</body></html>
```

效果根据浏览器语言的信息切换国际化

原理：

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

```
@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local国际化信息（截图就是这么犀利）

 18.accept-language

2、点击链接切换国际化

自己编写localResolver，加到容器中

1、更改HTML代码

```
<p class="mt-5 mb-3 text-muted">© 2017-2018</p>
  <a href="#" class="btn btn-sm" th:href="@{/index.html?lg=zh_CN}">中文</a>
  <a href="#" class="btn btn-sm" th:href="@{/index.html?lg=en_US}">English</a>
```

2、新建一个MyLocaleResolver.class

```
public class MyLocaleResolver implements LocaleResolver {

    //解析区域信息
    @Override
    public Locale resolveLocale(HttpServletRequest request) {
        String l = request.getParameter("lg");
        Locale locale = Locale.getDefault();
        if(!StringUtils.isEmpty(l)){
            String[] split = l.split("_");
            locale = new Locale(split[0], split[1]);
        }
        return locale;
    }

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

    }
}
```

3、将MyLocaleResolver加入到容器中

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

4、启动演示

3、登录拦截器

1、登录

开发技巧

1、清除模板缓存

2、Ctrl+F9刷新

1、新建一个LoginController

```
@Controller
public class LoginController {

    @PostMapping(value ="/user/login")
    public String login(@RequestParam("username")String username,
                        @RequestParam("password")String password,
                        Map<String,Object> map){
        if(!StringUtils.isEmpty(username) && "123456".equals(password)){
            //登录成功
            return "list";
        }else{
            map.put("msg", "用户名密码错误");
            return "login";
        }

    }
}
```

2、登录错误消息显示

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

3、表单重复提交

表单重复提交事件 --》重定向来到成功页面--》模板引擎解析

```
if(!StringUtils.isEmpty(username) && "123456".equals(password)){
    //登录成功,防止重复提交
    return "redirect:/main.html";
}else{
    map.put("msg", "用户名密码错误");
    return "login";
}
```

模板引擎解析

```
@Override
public void addViewControllers(ViewControllerRegistry registry) {
    registry.addViewController("/").setViewName("login");
    registry.addViewController("/index.html").setViewName("login");
    registry.addViewController("/main.html").setViewName("Dashboard");
}
```

4、拦截器

作用：实现权限控制，每个页面请求前中后，都会进入到拦截器进行处理（登录权限）

1、在component下新建一个LoginHandlerInterceptor拦截器

```
public class LoginHandlerInterceptor implements HandlerInterceptor {

    //目标方法执行之前
    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response,
Object handler) throws Exception {
        Object user = request.getSession().getAttribute("loginUser");
        if(user!=null){
            //已经登录
            return true;
        }
        //未经过验证
        request.setAttribute("msg", "没权限请先登录");
        request.getRequestDispatcher("/index.html").forward(request, response);

        return false;
    }

    @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 {

    }
}
```

2、在MyMvcConfig配置中重写拦截器方法，加入到容器中

```
//所有的webMvcConfigurerAdapter组件会一起起作用
@Bean //註冊到容器去
public WebMvcConfigurerAdapter webMvcConfigurerAdapter(){
    WebMvcConfigurerAdapter adapter = new WebMvcConfigurerAdapter() {
        @Override
        public void addViewControllers(ViewControllerRegistry registry) {
            registry.addViewController("/").setViewName("login");
            registry.addViewController("/index.html").setViewName("login");
            registry.addViewController("/main.html").setViewName("Dashboard");
        }
        //注册拦截器
        @Override
        public void addInterceptors(InterceptorRegistry registry) {
            //静态资源 css js img 已经做好了静态资源映射
            registry.addInterceptor(new LoginHandlerInterceptor()).addPathPatterns("/**").
                    excludePathPatterns("/index.html","/","/user/login");
```

```
        }
    };
    return adapter;
}
```

3、在LoginHandler中添加登录成功写入session

```
@Controller
public class LoginController {

    @PostMapping(value ="/user/login")
    public String login(@RequestParam("username")String username,
                        @RequestParam("password")String password,
                        Map<String,Object> map,
                        HttpSession session){
        if(!StringUtils.isEmpty(username) && "123456".equals(password)){
            //登录成功,防止重复提交
            session.setAttribute("loginUser", username);
            return "redirect:/main.html";
        }else{
            map.put("msg", "用户名密码错误");
            return "login";
        }

    }
}
```

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操作

	普通CRUD	**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

	请求URI	请求方式
查询所有员工	emps	GET
查询某个员工	emp/{id}	GET
添加页面	emp	GET
添加员工	emp	POST
修改页面(回显)	emp/{id}	GET
修改员工	emp/{id}	PUT
删除员工	emp/{id}	DELETE

3、员工列表

1、公共页面抽取

使用方法

```
1、抽取公共片段
<!--footer.html-->
<div id="footid" th:fragment="copy">xxx</div>
2、引入公共片段
<!--test.html-->
<div th:insert=~{footer::copy}></div>
~{templatename::selector} 模板名::选择器  footer::#footid
~{templatename::fragmentname} 模板名::片段名称 footer::copy
行内写法可以加~{xx::xx} 标签体可以 xx::xx
```

三种引用方式

th:insert :加个外层标签 +1

th:replace :完全替换 1

th:include：就替换里面的内容 -1

公共页面

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

结果

```
<body>
...
    <!-- th:insert -->
    <div>
        <footer>
            &copy; 2011 The Good Thymes Virtual Grocery
        </footer>
    </div>
    <!--th:replace-->
    <footer>
        &copy; 2011 The Good Thymes Virtual Grocery
    </footer>
    <!--th:include-->
    <div>
        &copy; 2011 The Good Thymes Virtual Grocery
    </div>
</body>
```

用此种方法将公共页面引入

2、列表高亮

引入片段的时候传入参数，新建一个commons文件夹存储公共页面bar.html

模板引入变量名

dashboard

```
<a class="nav-link active"
   th:class="${activeUri}=='main.html'?'nav-link active':'nav-link'"
   href="https://getbootstrap.com/docs/4.1/examples/dashboard/#" 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>
```

员工管理

```
<li class="nav-item">
    <a class="nav-link"
       th:class="${activeUri}=='emps'?'nav-link active':'nav-link'"
       href="https://getbootstrap.com/docs/4.1/examples/dashboard/#" th:href="@{/emps}">
        <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-users"><path d="M17 21v-2a4 4 0 0 0-4-4H5a4 4 0 0 0-
4 4v2"></path><circle cx="9" cy="7" r="4"></circle><path d="M23 21v-2a4 4 0 0 0-3-3.87">
</path><path d="M16 3.13a4 4 0 0 1 0 7.75"></path></svg>
        员工管理
    </a>
```

引入模板的时候传入参数

dashboard.html引入

```
<!--引入侧边栏-->
   <div th:replace="commons/bar :: sidebar(activeUri='main.html')"></div>
```

list.html引入

```
<!--引入侧边栏-->
<div th:replace="commons/bar::sidebar(activeUri='emps')"></div>
```

6、列表数据显示（查）

1、传入员工对象

EmployeeController类,传入员工对象

```
@Controller
public class EmployeeController {

    @Autowired
    EmployeeDao employeeDao;
    /**
     * 查询所有员工返回列表页面
     */
    @GetMapping(value = "/emps")
    public String list(Model model){

        Collection<Employee> employees = employeeDao.getAll();
        model.addAttribute("emps",employees);
        return "emp/list";
    }
}
```

2、 遍历对象

list.html中 使用模板的 `th:each` 方法

```
table class="table table-striped table-sm">
    <thead>
    <tr>
        <th>#</th>
        <th>lastName</th>
        <th>email</th>
        <th>gender</th>
        <th>department</th>
        <th>birth</th>
        <th>操作</th>
    </tr>
    </thead>
    <tbody>
        <tr th:each="emp:${emps}">
            <td th:text="${emp.id}">1</td>
            <td th:text="${emp.lastName}">1</td>
            <td th:text="${emp.email}">1</td>
            <td th:text="${emp.gender}">1</td>
            <td th:text="${emp.department.departmentName}">1</td>
            <td th:text="${#dates.format(emp.birth,'yyyy-MM-dd HH:mm:ss')}">1</td>
            <td>
                <button class="btn btn-sm btn-primary">编辑</button>
                <button class="btn btn-sm btn-danger">删除</button>
            </td>
        </tr>
    </tbody>
</table>
```

3、效果显示

 19.table list

7、员工添加（增）

功能：点击添加按钮，出现新增页面

1、新增页面

```
<form>
    <!-- LastName -->
    <div class="form-group">
        <label for="LastName">LastName</label>
        <input type="text" class="form-control" id="LastName"  placeholder="LastName">
    </div>
    <!-- Email -->
    <div class="form-group">
```

```
        <label for="Email">Email</label>
        <input type="email" class="form-control" id="Email"  placeholder="zhangsan@163.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 for="exampleFormControlSelect1">department</label>
        <select class="form-control" id="exampleFormControlSelect1">
            <option th:each="dept:${depts}" th:value="${dept.id}"
th:text="${dept.departmentName}"></option>
        </select>
    </div>
    <!--Birth-->
    <div class="form-group">
        <label for="birthDate">Birth</label>
        <input type="text" class="form-control" id="birthDate" placeholder="2012-12-12">
    </div>
    <button type="submit" class="btn btn-primary">添 加</button>
</form>
```

2、页面跳转

在EmployeeController中添加addEmpPage方法

```
/**
 * 添加员工
 */
@GetMapping(value = "/emp")
public String toAddPage(Model model){
    //来到添加页面,查出所有部门显示
    Collection<Department> depts = departmentDao.getDepartments();
    model.addAttribute("depts",depts);
    return "emp/add";
}
```

关键点：在添加部门页面要遍历部门信息，所以在方法中出入部门信息

3、添加功能完成

新建一个PostMapping

> ThymeleafViewResolver 查看redirect和forward,原生的sendredirect方法;

1、新建一个postMapping的方法用来接受页面的添加POST请求

```
/**
 * 员工添加
 */
@PostMapping(value = "/emp")
public String addEmp(Employee employee){

    employeeDao.save(employee);
    //来到员工列表页面、redirect:重定向到一个地址，forward转发到一个地址
    return "redirect:/emps";
}
```

2、修改添加页面，添加name属性

```
<form th:action="@{/emp}" method="post">
    <!-- LastName -->
    <div class="form-group">
        <label for="LastName">LastName</label>
        <input type="text" class="form-control" id="LastName" name="lastName"
placeholder="LastName">
    </div>
    <!-- Email -->
    <div class="form-group">
        <label for="Email">Email</label>
        <input type="email" class="form-control" id="Email"  name="email"
placeholder="zhangsan@163.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"  name="department.id">
            <option th:each="dept:${depts}" th:value="${dept.id}"
th:text="${dept.departmentName}"></option>
        </select>
    </div>
    <div class="form-group">
        <label for="birthDate">Birth</label>
```

```
        <input type="text" class="form-control" id="birthDate" placeholder="2012-12-12"
name="birth">
    </div>
    <button type="submit" class="btn btn-primary">添 加</button>
</form>
```

1、部门对象问题?

```
<select class="form-control"  name="department.id">
```

2、日期格式化?

属性中添加 date-formate 默认是 /

```
@Bean
@ConditionalOnProperty(prefix = "spring.mvc", name = "date-format")
public Formatter<Date> dateFormatter() {
   return new DateFormatter(this.mvcProperties.getDateFormat());
}

@Override
public MessageCodesResolver getMessageCodesResolver() {
   if (this.mvcProperties.getMessageCodesResolverFormat() != null) {
      DefaultMessageCodesResolver resolver = new DefaultMessageCodesResolver();
      resolver.setMessageCodeFormatter(
            this.mvcProperties.getMessageCodesResolverFormat());
      return resolver;
   }
   return null;
}
```

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

8、员工编辑（改）

思路使用add页面，并且数据回显，然后区分添加，PUT请求

1、修改按钮

在list.html的 编辑 按钮加上链接

```
<td>
    <a  href="#" th:href="@{/emp/}+${emp.id}" class="btn btn-sm btn-primary">编辑</a>
    <button class="btn btn-sm btn-danger">删除</button>
</td>
```

2、编写跳转页面

跳转到员工编辑页面的Controller

```
/**
 * 员工编辑页面
 */
@GetMapping(value = "/emp/{id}")
public String toEditPage(@PathVariable("id") Integer id ,Model model){
    Employee emp = employeeDao.getEmpById(id);
    Collection<Department> departments = departmentDao.getDepartments();
    model.addAttribute("emp",emp);
    model.addAttribute("depts",departments);
    return "emp/add";
}
```

3、对页面修改

对add页面进行修改

1）、添加回显

2）、添加判断是否emp!=null（区分add or edit)

3）、添加put请求 --两个input的hidden标签

```
 <form th:action="@{/emp}" method="post">
        <!--发送put请求-->
        <!--1.SpringMVC配置HiddenHttpMethodFilter
            2.页面创建一个post表单
            3.创建一个 input name_method 值就是我们请求的方式-->
        <input type="hidden" name="_method" value="put" th:if="${emp!=null}">

        <input type="hidden" name="id" th:value="${emp.id}" th:if="${emp!=null}">
        <!-- LastName -->
        <div class="form-group">
            <label for="LastName">LastName</label>
            <input type="text" class="form-control" id="LastName" name="lastName"
placeholder="LastName" th:value="${emp!=null}?${emp.lastName}">
        </div>
        <!-- Email -->
        <div class="form-group">
            <label for="Email">Email</label>
            <input type="email" class="form-control" id="Email"  name="email"
placeholder="zhangsan@163.com" th:value="${emp!=null}?${emp.email}">
        </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"
```

```
th:checked="${emp!=null}?${emp.gender}==1">
                <label class="form-check-label" >男</label>
            </div>
            <div class="form-check form-check-inline">
                <input class="form-check-input" type="radio" name="gender" value="0"
th:checked="${emp!=null}?${emp.gender}==0">
                <label class="form-check-label" >女</label>
            </div>
        </div>
        <!-- department -->
        <div class="form-group">
            <label >department</label>
            <select class="form-control"  name="department.id" >
                <option th:selected="${emp!=null}?${dept.id == emp.department.id}"
th:each="dept:${depts}" th:value="${dept.id}" th:text="${dept.departmentName}"></option>
            </select>
        </div>
        <div class="form-group">
            <label for="birthDate">Birth</label>
            <input type="text" class="form-control" id="birthDate" placeholder="2012-12-12"
name="birth" th:value="${emp!=null}?${#dates.format(emp.birth,'yyyy-MM-dd HH:mm:ss')}">
        </div>
        <button type="submit" class="btn btn-primary" th:text="${emp!=null}?'修改':'添加'">添
加</button>
    </form>
</main>
```

9、员工删除（删）

1、新建Contoller

```
/**
 * 员工删除
 */
@DeleteMapping(value = "/emp/{id}")
public String deleteEmp(@PathVariable("id") Integer id){
    employeeDao.deleteEmpById(id);
    return "redirect:/emps";
}
```

2、修改删除标签

```
<button th:attr="del_uri=@{/emp/}+${emp.id}"  class="btn btn-sm btn-danger deleteBtn">
    删除
</button>
```

3、写Form表单

form表单卸载外面，input 中 name="_method" value="delete" 模拟delete请求

```
                </tbody>
            </table>
        </div>
    </main>
    <form id="deleteEmpForm" method="post">
        <input type="hidden" name="_method" value="delete">
    </form>
</div>
```

4、写JS提交

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

> return false;禁用btn提交效果

7、错误机制的处理

1、默认的错误处理机制

默认错误页面

 20.error

原理参照

ErrorMvcAutoConfiguration:错误处理的自动配置

```
org\springframework\boot\spring-boot-autoconfigure\1.5.12.RELEASE\spring-boot-autoconfigure-1.5.12.RELEASE.jar!\org\springframework\boot\autoconfigure\web\ErrorMvcAutoConfiguration.class
```

- 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;
}
```

- BasicErrorController

```
@Controller
@RequestMapping("${server.error.path:${error.path:/error}}")
public class BasicErrorController extends AbstractErrorController {
    //产生HTML数据
    @RequestMapping(produces = "text/html")
    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());
        ModelAndView modelAndView = resolveErrorView(request, response, status, model);
        return (modelAndView == null ? new ModelAndView("error", model) : modelAndView);
    }
    //产生Json数据
    @RequestMapping
    @ResponseBody
    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);
    }
```

- ErrorPageCustomizer

```
@Value("${error.path:/error}")
private String path = "/error";//系统出现错误以后来到error请求进行处理，(web.xml)
```

- 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可以找到一个页面? error/状态码
   String errorViewName = "error/" + viewName;
    //如果模板引擎可以解析地址，就返回模板引擎解析
   TemplateAvailabilityProvider provider = this.templateAvailabilityProviders
         .getProvider(errorViewName, this.applicationContext);
   if (provider != null) {
       //有模板引擎就返回到errorViewName指定的视图地址
      return new ModelAndView(errorViewName, model);
   }
    //自己的文件 就在静态文件夹下找静态文件 /静态资源文件夹/404.html
   return resolveResource(errorViewName, model);
}
```

一旦系统出现4xx或者5xx错误 ErrorPageCustomizer就回来定制错误的响应规则,就会来到 /error请求,BasicErrorController处理，就是一个Controller

1.响应页面,去哪个页面是由 DefaultErrorViewResolver 拿到所有的错误视图

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

l浏览器发送请求 accpt:text/html

客户端请求：accept:/*

2、如何定制错误响应

1）、如何定制错误的页面

1.有模板引擎：静态资源/404.html,什么错误什么页面；所有以4开头的 4xx.html 5开头的5xx.html

有精确的404和4xx优先选择404

页面获得的数据

timestamp：时间戳

status：状态码

error：错误提示

exception：异常对象

message：异常信息

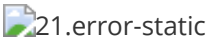
errors:JSR303有关

2.没有放在模板引擎，放在静态文件夹，也可以显示，就是没法使用模板取值

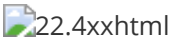
3.没有放模板引擎，没放静态，会显示默认的错误

2）、如何定义错误的数据

举例子：新建4xx和5xx文件



```
<body >
    <p>status: [[${status}]]</p>
    <p>timestamp: [[${timestamp}]]</p>
    <p>error: [[${error}]]</p>
    <p>message: [[${message}]]</p>
    <p>exception: [[${exception}]]</p>
</body>
```



3、如何定制Json数据

1、仅发送json数据

```
public class UserNotExitsException extends  RuntimeException {
    public UserNotExitsException(){
        super("用户不存在");
    }
}
```

```
/**
 * 异常处理器
 */
@ControllerAdvice
public class MyExceptionHandler {

    @ResponseBody
    @ExceptionHandler(UserNotExitsException.class)
    public Map<String ,Object> handlerException(Exception e){
```

```
        Map<String ,Object> map =new HashMap<>();
        map.put("code", "user not exist");
        map.put("message", e.getMessage());
        return map;
    }
}
```

无法自适应 都是返回的json数据

2、转发到error自适应处理

```
@ExceptionHandler(UserNotExitsException.class)
public String handlerException(Exception e, HttpServletRequest request){
    Map<String ,Object> map =new HashMap<>();
    //传入自己的状态码
    request.setAttribute("javax.servlet.error.status_code", 432);
    map.put("code", "user not exist");
    map.put("message", e.getMessage());
    //转发到error
    return "forward:/error";
}
```

程序默认获取状态码

```
protected HttpStatus getStatus(HttpServletRequest request) {
   Integer statusCode = (Integer) request
         .getAttribute("javax.servlet.error.status_code");
   if (statusCode == null) {
      return HttpStatus.INTERNAL_SERVER_ERROR;
   }
   try {
      return HttpStatus.valueOf(statusCode);
   }
   catch (Exception ex) {
      return HttpStatus.INTERNAL_SERVER_ERROR;
   }
```

没有自己写的自定义异常数据

3、自适应和定制数据传入

Spring 默认的原理，出现错误后回来到error请求，会被BasicErrorController处理,响应出去的数据是由BasicErrorController的父类AbstractErrorController(ErrorController)规定的方法getAttributes得到的;

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

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

容器中的DefaultErrorAtrributes.getErrorAtrributees();默认进行数据处理

```
public class MyErrorAttributes extends DefaultErrorAttributes {
    @Override
    public Map<String, Object> getErrorAttributes(RequestAttributes requestAttributes,
boolean includeStackTrace) {
        Map<String, Object> map = super.getErrorAttributes(requestAttributes,
includeStackTrace);
        map.put("company", "wdjr");
        return map;
    }
}
```

异常处理：把map方法请求域中

```
    @ExceptionHandler(UserNotExitsException.class)
    public String handlerException(Exception e, HttpServletRequest request){
        Map<String ,Object> map =new HashMap<>();
        //传入自己的状态码
        request.setAttribute("javax.servlet.error.status_code", 432);
        map.put("code", "user not exist");
        map.put("message", e.getMessage());
        request.setAttribute("ext", map);
        //转发到error
        return "forward:/error";
    }
}
```

在上面的MyErrorAttributes类中加上

```
//我们的异常处理器
Map<String,Object> ext = (Map<String, Object>) requestAttributes.getAttribute("ext", 0);
map.put("ext", ext);
```

8、配置嵌入式servlet容器

1、定制和修改Servlet容器

SpringBoot默认使用Tomcat作为嵌入式的Servlet容器；

 23.tomcat emd

问题?

1）、如何定制和修改Servlet容器；

1、 修改Server相关的配置文件 application.properties

```
#通用的servlet容器配置
server.xxx
#tomcat的配置
server.tomcat.xxxx
```

2、编写一个EmbeddedServletContainerCustomizer;嵌入式的Servlet容器的定制器；来修改Servlet的容器配置

```
@Bean
public EmbeddedServletContainerCustomizer embeddedServletContainerCustomizer(){
    return new EmbeddedServletContainerCustomizer() {
        //定制嵌入式Servlet的容器相关规则
        @Override
        public void customize(ConfigurableEmbeddedServletContainer container) {
            container.setPort(8999);
        }
    };
}
```

其实同理，都是实现EmbeddedServletContainerCustomizer

2、注册Servlet三大组件

三大组件 Servlet Filter Listener

由于SprringBoot默认是以jar包启动嵌入式的Servlet容器来启动SpringBoot的web应用，没有web.xml

注册三大组件

ServletRegistrationBean

```
@Bean
public ServletRegistrationBean myServlet(){
    ServletRegistrationBean servletRegistrationBean = new ServletRegistrationBean(new
MyServlet(),"/servlet");
    return servletRegistrationBean;
}
```

MyServlet

```
public class MyServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        resp.getWriter().write("Hello Servlet");
    }
}
```

FilterRegistrationBean

```
@Bean
public FilterRegistrationBean myFilter(){
    FilterRegistrationBean filterRegistrationBean = new FilterRegistrationBean();
    filterRegistrationBean.setFilter(new MyFilter());
    filterRegistrationBean.setUrlPatterns(Arrays.asList("/hello","/myServlet"));
    return filterRegistrationBean;
}
```

MyFilter

```
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

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

MyListener

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

    @Override
    public void contextDestroyed(ServletContextEvent sce) {
        System.out.println(".........web应用销毁..........");
    }
}
```

SpringBoot帮助我们自动配置SpringMVC的时候，自动注册SpringMVC的前端控制器；DispatcherServlet;

```
@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 包括静态资源 不包括jsp
       //可以通过server.servletPath来修改SpringMVC前端控制器默认拦截的请求路径
      registration.setName(DEFAULT_DISPATCHER_SERVLET_BEAN_NAME);
      registration.setLoadOnStartup(
            this.webMvcProperties.getServlet().getLoadOnStartup());
      if (this.multipartConfig != null) {
         registration.setMultipartConfig(this.multipartConfig);
      }
      return registration;
   }

}
```

3、切换其他的Servlet容器

在ServerProperties中

```
private final Tomcat tomcat = new Tomcat();

private final Jetty jetty = new Jetty();

private final Undertow undertow = new Undertow();
```

tomcat(默认支持)

jetty（长连接）

undertow（多并发）

切换容器 仅仅需要修改pom文件的依赖就可以

```
        <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>

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

4、嵌入式Servlet容器自动配置原理

```
@AutoConfigureOrder(Ordered.HIGHEST_PRECEDENCE)
@Configuration
@ConditionalOnWebApplication
@Import(BeanPostProcessorsRegistrar.class)
//给容器导入组件 后置处理器 在Bean初始化前后执行前置后置的逻辑 创建完对象还没属性赋值进行初始化工作
public class EmbeddedServletContainerAutoConfiguration {
    @Configuration
    @ConditionalOnClass({ Servlet.class, Tomcat.class })//当前是否引入tomcat依赖
    //判断当前容器没有用户自定义EmbeddedServletContainerFactory，就会创建默认的嵌入式容器
    @ConditionalOnMissingBean(value = EmbeddedServletContainerFactory.class, search =
SearchStrategy.CURRENT)
    public static class EmbeddedTomcat {

        @Bean
        public TomcatEmbeddedServletContainerFactory tomcatEmbeddedServletContainerFactory()
{
            return new TomcatEmbeddedServletContainerFactory();
        }
```

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);

}
```

继承关系

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

3）、TomcatEmbeddedServletContainerFactory为例

```
@Override
public EmbeddedServletContainer getEmbeddedServletContainer(
      ServletContextInitializer... initializers) {
   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；并且启动tomcat容器
   return getTomcatEmbeddedServletContainer(tomcat);
}
```

4）、嵌入式配置修改

```
ServerProperties、EmbeddedServletContainerCustomizer
```

EmbeddedServletContainerCustomizer:定制器帮我们修改了Servlet容器配置?

怎么修改?

5）、容器中导入了**EmbeddedServletContainerCustomizerBeanPostProcessor**

```
@Override
public void registerBeanDefinitions(AnnotationMetadata importingClassMetadata,
      BeanDefinitionRegistry registry) {
   if (this.beanFactory == null) {
      return;
   }
   registerSyntheticBeanIfMissing(registry,
         "embeddedServletContainerCustomizerBeanPostProcessor",
         EmbeddedServletContainerCustomizerBeanPostProcessor.class);
   registerSyntheticBeanIfMissing(registry,
         "errorPageRegistrarBeanPostProcessor",
         ErrorPageRegistrarBeanPostProcessor.class);
}
```

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

private void postProcessBeforeInitialization(
    ConfigurableEmbeddedServletContainer bean) {
    //获取所有的定制器，调用每个定制器的customer方法给Servlet容器进行赋值
    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
            //从容器中获取所有的这个类型的组件：EmbeddedServletContainerCustomizer
            //定制Servlet,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EmbeddedServletContainerCustomizer定制器

步骤：

1）、SpringBoot根据导入的依赖情况，给容器中添加响应的容器工厂 例：tomcat

EmbeddedServletContainerFactory【TomcatEmbeddedServletContainerFactory】

2）、容器中某个组件要创建对象就要通过后置处理器；

```
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，从容器中获取的所有的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的容器工厂？什么时候获取嵌入式的Servlet容器并启动Tomcat;

获取嵌入式的容器工厂

1）、SpringBoot应用启动Run方法

2）、刷新IOC容器对象【创建IOC容器对象，并初始化容器，创建容器的每一个组件】；如果是web环境 AnnotationConfigEmbeddedWebApplicationContext,如果不是AnnotationConfigApplicationContext

```
if (contextClass == null) {
   try {
      contextClass = Class.forName(this.webEnvironment
            ? DEFAULT_WEB_CONTEXT_CLASS : DEFAULT_CONTEXT_CLASS);
   }
```

3）、refresh(context);刷新创建好的IOC容器

```
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();
}
```

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

5）、webioc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；

```
@Bean
public TomcatEmbeddedServletContainerFactory tomcatEmbeddedServletContainerFactory() {
return new TomcatEmbeddedServletContainerFactory();
}
```

TomcatEmbeddedServletContainerFactory创建对象，后置处理器看这个对象，就来获取所有的定制器来定制Servlet容器的相关配置；

7）、使用容器工厂获取嵌入式的Servlet容器

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，在将ioc容器中剩下的没有创建出的对象获取出来

ioc启动创建Servlet容器

9、使用外置的Servlet容器

嵌入式的Servlet容器：应用达成jar包

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、自定义定制器】，自己来编写嵌入式的容器工厂）

外置的Servlet容器：外面安装Tomcat是以war包的方式打包。

1、IDEA操作外部Servlet

1、创建程序为war程序

 26.tomcat1

2、选择版本

 27.tomcat2

3、添加tomcat

 28.tomcat3

4、选择tomcat

 30.tomcat4

5、选择本地的Tomcat

 31.tomcat5

6、配置tomcat路径

 32.tomcat6

7、添加服务器

 33.tomcat7

8、添加exploded的war配置，应用OK tomcat配置完成

 34.tomcat8

二、配置webapp文件夹

1、点击配置

 35.tomcat9

2、添加webapp目录

 36.tomcat10

3、默认配置就可以

 37.tomcat11

4、配置web.xml文件

 38.tomcat12

5、文档目录结构

 39.tomcat13

2、运行一个示例

1、项目目录

 40.demo1

2、配置文件写视图解析前后缀

```
spring.mvc.view.prefix=/WEB-INF/jsp/

spring.mvc.view.suffix=.jsp
```

3、HelloController

```
@Controller
public class HelloController {
    @GetMapping("/hello")
    public String hello(Model model){
        model.addAttribute("message","这是Controller传过来的message");
        return "success";
    }
}
```

4、success.jsp

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>Success</title>
</head>
<body>
<h1>Success</h1>
message:${message}
</body>
</html>
```

5、运行结果

 41.demo2

步骤

1、必须创建一个war项目；

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方法里面的固定写法

```
public class ServletInitializer extends SpringBootServletInitializer {

    @Override
    protected SpringApplicationBuilder configure(SpringApplicationBuilder application) {
        //传入SpringBoot的主程序,
        return application.sources(SpringBoot04WebJspApplication.class);
    }

}
```

4、启动服务器就可以;

3、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的容器;

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

servlet3.0规范

8.2.4 共享库和运行时插件

规则:

1、服务器启动（web应用启动），会创建当前的web应用里面每一个jar包里面ServletContrainerInitializer的实现类的实例

2、SpringBootServletInitializer这个类的实现需要放在jar包下的META-INF/services文件夹下，有一个命名为javax.servlet.ServletContainerInitalizer的文件，内容就是ServletContainerInitializer的实现类全类名

3、还可以使用@HandlerTypes注解，在应用启动的时候可以启动我们感兴趣的类

流程:

1、启动Tomcat服务器

2、spring web模块里有这个文件

 42.servletContainerInit

```
org.springframework.web.SpringServletContainerInitializer
```

3、SpringServletContainerInitializer将handlerTypes标注的所有类型的类传入到onStartip方法的Set<Class<?>>;为这些感兴趣类创建实例

4、每个创建好的WebApplicationInitializer调用自己的onStratup

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

6、SpringBootServletInitializer执行onStartup方法会创建createRootApplicationContext

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

servletContext.setAttribute(WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE,
(Object)null);
        builder.initializers(new ApplicationContextInitializer[]{new
ParentContextApplicationContextInitializer(parent)});
    }

    builder.initializers(new ApplicationContextInitializer[]{new
ServletContextApplicationContextInitializer(servletContext)});
    builder.contextClass(AnnotationConfigEmbeddedWebApplicationContext.class);
    //调用Configure,子类重写了这个方法，将SpringBoot的主程序类传入进来
    builder = this.configure(builder);
    //创建一个spring应用
    SpringApplication application = builder.build();
    if (application.getSources().isEmpty() &&
AnnotationUtils.findAnnotation(this.getClass(), Configuration.class) != null) {
        application.getSources().add(this.getClass());
    }

    Assert.state(!application.getSources().isEmpty(), "No SpringApplication sources have
been defined. Either override the configure method or add an @Configuration annotation");
    if (this.registerErrorPageFilter) {
        application.getSources().add(ErrorPageFilterConfiguration.class);
    }
    //最后启动Spring容器
    return this.run(application);
}
```

7、Spring的应用就启动完了并且创建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);
      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);
   }
}
```

五、Docker

1、简介

Docker是一个开源的应用容器引擎

将软件编译成一个镜像；然后在镜像里各种软件做好配置，将镜像发布出去，其他的使用这就可以直接使用这个镜像。运行中的这个镜像叫做容器，容器启动速度快，类似ghost操作系统，安装好了什么都有了；

2、Docker的核心概念

docker主机（HOST）:安装了Docker程序的机器（Docker直接安装在操作系统上的）

docker客户端（Client）:操作docker主机

docker仓库（Registry）：用来保存打包好的软件镜像

docker镜像（Image）:软件打好包的镜像，放到docker的仓库中

docker容器（Container）:镜像启动后的实例（5个容器启动5次镜像）

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运行的这个镜像，镜像就会生成一个容器

4、对容器的启动停止，就是对软件的启动和停止

3、安装Docker

1、安装Linux

安装vxbox并且安装ubuntu

2、在linux上安装docker

```
1、查看centos版本
# uname -r
3.10.0-693.el7.x86_64
要求：大于3.10
如果小于的话升级*（选做）
# yum update
2、安装docker
# yum install docker
3、启动docker
# systemctl start docker
# docker -v
4、开机启动docker
# systemctl enable docker
5、停止docker
# systemctl stop docker
```

4、docker的常用操作

1、镜像操作

1、搜索

```
docker search mysql
```

默认去docker hub网站查找
 44.docker1

2、拉取

```
默认最新版本
# docekr pull mysql
安装指定版本
# docker pull mysql:5.5
```

3、查看

```
docker images
```

4、删除

```
docker rmi imageid
```

2、容器操作

软件的镜像（qq.exe） -- 运行镜像 -- 产生一个容器（正在运行的软件）

```
1、搜索镜像
# docker search tomcat
2、拉取镜像
# docker pull tomcat
3、根据镜像启动容器
[root@lion ~]# docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
docker.io/tomcat    latest              d3d38d61e402        35 hours ago        549 MB
[root@lion ~]# docker run --name mytomcat -d tomcat:latest
2f0348702f5f2a2777082198795d8059d83e5ee38f430d2d44199939cc63e249
4、查看那个进程正在进行
[root@lion ~]# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
PORTS               NAMES
2f0348702f5f        tomcat:latest       "catalina.sh run"   41 seconds ago      Up 39
seconds       8080/tcp            mytomcat
5、停止运行中容器
[root@lion ~]# docker stop 2f0348702f5f
2f0348702f5f
6、查看所有容器
[root@lion ~]# docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
PORTS               NAMES
2f0348702f5f        tomcat:latest       "catalina.sh run"   52 minutes ago      Exited (143)
2 minutes ago                       mytomcat
7、启动容器
[root@lion ~]# docker start 2f0348702f5f
8、删除docker容器
[root@lion ~]# docker rm 2f0348702f5f
2f0348702f5f
9、端口映射
[root@lion ~]# docker run --name mytomcat -d -p 8888:8080 tomcat
692c408c220128014df32ecb6324fb388427d1ecd0ec56325580135c58f63b29
虚拟机:8888
容器的:8080
-d:后台运行
-p:主机端口映射到容器端口
浏览器：192.168.179.129:8888
10、docker的日志
```

```
[root@lion ~]# docker logs 692c408c2201
11、多个启动
[root@lion ~]# docker run -d -p 9000:8080 --name mytomcat2 tomcat
浏览器：192.168.179.129:9000
```

更多命令参考docker镜像文档

3、安装Mysql

```
docker pull mysql
docker run --name mysql001 -e MYSQL_ROOT_PASSWORD -d -p 3307:3306 mysql
```

六、数据访问

1、整合JDBC数据源

1、新建项目 spring-boot-06-data-jdbc

- WEB
- Mysql
- JDBC
- SpringBoot1.5

2、编写配置文件appliction.yml

```
spring:
  datasource:
    username: root
    password: Welcome_1
    url: jdbc:mysql://192.168.179.131:3306/jdbc
    driver-class-name: com.mysql.jdbc.Driver
```

3、编写测试类测试

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class SpringBoot06DataJdbcApplicationTests {

    @Autowired
    DataSource dataSource;

    @Test
    public void contextLoads() throws SQLException {
        System.out.println(dataSource.getClass());

        Connection connection = dataSource.getConnection();
        System.out.println(connection);
```

```
        connection.close();
    }

}
```

4、测试结果

```
class org.apache.tomcat.jdbc.pool.DataSource
ProxyConnection[PooledConnection[com.mysql.jdbc.JDBC4Connection@c35af2a]]
```

数据源相关配置都在DataSourceProperties属性里

自动配置原理

E:\Develop\Maven_Repo\org\springframework\boot\spring-boot-autoconfigure\1.5.13.RELEASE\spring-boot-autoconfigure-1.5.13.RELEASE.jar!\org\springframework\boot\autoconfigure\jdbc

1、DataSource

参考DataSourceConfiguration,根据配置创建数据源，默认是使用tomcat连接池，可以使用spring.datasource.type指定自定义的数据源

2、SpringBoot默认支持

```
Tomcat数据源
HikariDataSource
dbcp.BasicDataSource
dbcp2.BasicDataSource
```

3、自定义数据源

```
 */
@ConditionalOnMissingBean(DataSource.class)
@ConditionalOnProperty(name = "spring.datasource.type")
static class Generic {

   @Bean
   public DataSource dataSource(DataSourceProperties properties) {
       //使用builder创建数据源，利用反射创建相应的type数据源，并绑定数据源
      return properties.initializeDataSourceBuilder().build();
   }

}
```

4、运行sql建表

在DataSourceAutoConfiguration中**DataSourceInitializer**类

监听器

作用：

1）、postConstruct -》runSchemaScript 运行建表sql文件

2）、runDataScript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;
```

举个栗子

创建department表

1、department.sql

```
/*
Navicat MySQL Data Transfer

Source Server         : 192.168.179.131
Source Server Version : 50719
Source Host           : 192.168.179.131:3306
Source Database       : jdbc

Target Server Type    : MYSQL
Target Server Version : 50719
File Encoding         : 65001

Date: 2018-05-14 14:28:52
*/

SET FOREIGN_KEY_CHECKS=0;

-- ----------------------------
-- Table structure for department
-- ----------------------------
DROP TABLE IF EXISTS `department`;
CREATE TABLE `department` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `departmentName` varchar(255) DEFAULT '',
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

2、将department.sql命名为schema-all.sql

 45.schema-all

3、运行测试类

自定义sql的文件名，department.sql在配置文件中

```
schema:
  - classpath:department.sql
```

5、操作JdbcTemplate

FBI warning:将department.sql删除或者改名，因为运行文件会将表中数据清除

1、新建一个Controller

```
@Controller
public class HelloController {

    @Autowired
    JdbcTemplate jdbcTemplate;

    @ResponseBody
    @GetMapping("/hello")
    public Map<String ,Object> hello(){

        List<Map<String, Object>> list = jdbcTemplate.queryForList("select * from
department");
        return list.get(0);
    }
}
```

2、表中添加数据

 46.department

3、访问请求查询数据

 47.hello

2、自定义数据源

1、导入Druid的依赖

```
<!-- https://mvnrepository.com/artifact/com.alibaba/druid -->
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid</artifactId>
    <version>1.1.9</version>
</dependency>
```

2、修改配置文件

```
spring:
  datasource:
    username: root
    password: Welcome_1
    url: jdbc:mysql://192.168.179.131:3306/jdbc
    driver-class-name: com.mysql.jdbc.Driver
    type: com.alibaba.druid.pool.DruidDataSource
#    schema:
#      - classpath:department.sql
server:
  port: 9000
```

已经替换了原来的tomcat数据源

3、配置Druid数据源配置

```
spring:
  datasource:
    username: root
    password: Welcome_1
    url: jdbc:mysql://192.168.179.131:3306/jdbc
    driver-class-name: com.mysql.jdbc.Driver
    type: com.alibaba.druid.pool.DruidDataSource
    # 初始化大小，最小，最大
    initialSize: 5
    minIdle: 5
    maxActive: 20
    # 配置获取连接等待超时的时间
    maxWait: 60000
    # 配置间隔多久才进行一次检测，检测需要关闭的空闲连接，单位是毫秒
    timeBetweenEvictionRunsMillis: 60000
    # 配置一个连接在池中最小生存的时间，单位是毫秒
    minEvictableIdleTimeMillis: 300000
    validationQuery: SELECT 1 FROM DUAL
    testWhileIdle: true
    testOnBorrow: false
    testOnReturn: false
    poolPreparedStatements: true
    # 配置监控统计拦截的filters,去掉监控界面sql无法统计，‘wall’用于防火墙
    filters: stat,wall,log4j
```

```
    maxPoolPreparedStatementPerConnectionSize: 20
    userGlobalDataSourceStat: true
    # 通过connectProperties属性来打开mergeSql功能；慢SQL记录
    connectionProperties: druid.stat.mergeSql=true;druid.stat.slowSqlMillis=500
#    schema:
#      - classpath:department.sql
server:
  port: 9000
```

4、Druid配置监控

```
@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
    @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");
        bean.setInitParameters(initParams);
        return bean;
    }
    //2、配置监控的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;
    }

}
```

5、运行测试，访问 localhost:9000/druid

 48.druid

输入刚才调好的用户名密码即可访问

3、整合Mybatis

1、新建工程，SpringBoot1.5+web+JDBC+Mysql

导入依赖

```
<dependency>
    <groupId>org.mybatis.spring.boot</groupId>
    <artifactId>mybatis-spring-boot-starter</artifactId>
    <version>1.3.2</version>
</dependency>
<!-- https://mvnrepository.com/artifact/com.alibaba/druid -->
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid</artifactId>
    <version>1.1.9</version>
</dependency>
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <scope>runtime</scope>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jdbc</artifactId>
</dependency>
```

2、导入配置文件中关于Druid的配置

2.1、导入依赖

2.2、配置文件application.yml（指定用户名密码...配置Druid的配置参数，修改sql文件加载的默认名）

2.3、将Druid组件加入到容器中（监控）重点

具体同上

3、创建数据表department和employee表

3.1、根据sql文件，新建两张表

3.2、修改加载的sql名（默认为schema.sql和schema-all.sql）

```
spring:
  datasource:
    schema:
      - classpath:sql/department.sql
      - classpath:sql/employeee.sql
```

3.3、运行程序检查数据库是否创建成功

4、创建数据库对应的JavaBean （驼峰命名，getter/setter toString/注释掉schema防止重复创建）

在配置文件中修改驼峰命名开启 ,不写配置文件就写配置类

```
mybatis:
  configuration:
    map-underscore-to-camel-case: true
```

```
//类名冲突所以全类名
@org.springframework.context.annotation.Configuration
public class MyBatisConfig {

    @Bean
    public ConfigurationCustomizer configurationCustomizer(){

        return new ConfigurationCustomizer() {
            @Override
            public void customize(Configuration configuration) {
                configuration.setMapUnderscoreToCamelCase(true);
            }
        };
    }
}
```

注解方式

5、新建mapper

```
//指定是一个mapper
@Mapper
public interface DepartmentMapper {

    @Insert("insert into department(dept_name) value(#{deptName})")
    public int insertDept(Department department);

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

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

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

}
```

6、编写controller测试

```
@RestController
public class DeptController {

    @Autowired
    DepartmentMapper departmentMapper;

    @RequestMapping("/getDept/{id}")
    public Department getDepartment(@PathVariable("id") Integer id){
        return departmentMapper.getDeptById(id);
    }

    @RequestMapping("/delDept/{id}")
    public int delDept(@PathVariable("id") Integer id){
        return departmentMapper.deleteDeptById(id);
    }

    @RequestMapping("/update/{id}")
    public int updateDept(@PathVariable("id") Integer id){
        return departmentMapper.updateDept(new Department(id, "开发部"));
    }

    @GetMapping("/insert")
    public int insertDept(Department department){
        return departmentMapper.insertDept(department);
    }
}
```

问题：

mapper文件夹下有多个mapper文件，加麻烦，可以直接扫描整个mapper文

件夹下的mapper

```
//主配置类或者mybatis配置类
@MapperScan(value = "com.wdjr.springboot.mapper")
```

配置文件方式

1、新建文件

 50.mybatisxml

2、新建mybatis的配置文件

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

3、新建Employee的接口方法

```
public interface EmployeeMapper {

    public Employee getEmpById(Integer id);

    public void insetEmp(Employee employee);
}
```

4、新建Employee的mapper.xml的映射文件

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
        PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.wdjr.springboot.mapper.EmployeeMapper">
    <select id="getEmpById" resultType="com.wdjr.springboot.bean.Employee">
      select * from employee where id=#{id}
   </select>

    <insert id="insetEmp">
        INSERT  INTO employee(last_name,email,gender,d_id) VALUES (#{lastName},#{email},#
{gender},#{dId})
    </insert>
</mapper>
```

5、修改application.yml配置文件

```
mybatis:
  config-location: classpath:mybatis/mybatis-config.xml
  mapper-locations: classpath:mybatis/mapper/*.xml
```

6、新建一个Controller访问方法

```
@RestController
public class EmployeeController {
```

```
    @Autowired
    EmployeeMapper employeeMapper;

    @RequestMapping("/getEmp/{id}")
    public Employee getEmp(@PathVariable("id") Integer id){
        return employeeMapper.getEmpById(id);
    }

    @GetMapping("/insertEmp")
    public Employee insertEmp(Employee employee){
        employeeMapper.insetEmp(employee);
        return employee;
    }
}
```

4、JPA数据访问

新建工程 springBoot1.5+Web+JPA+MYSQL+JDBC

目录结构

1、新建一个实体类User

```
//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来指定和那个数据表对应，如果省略默认表明就是user;

public class User {

    @Id //这是一个主键
    @GeneratedValue(strategy = GenerationType.IDENTITY)//自增组件
    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;
    @Column
    public Integer getId() {
        return id;
    }

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

    public String getLastName() {
        return lastName;
```

```
    }

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

    public String getEmail() {
        return email;
    }

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

2、新建一个UserRepository来继承jpa的绝大多数功能

```
//继承jpaRepository
public interface UserRepository extends JpaRepository<User,Integer> {

}
```

3、编写配置文件application.yml

```
spring:
  datasource:
    url: jdbc:mysql://192.168.179.131/jpa
    username: root
    password: Welcome_1
    driver-class-name: com.mysql.jdbc.Driver
  jpa:
    hibernate:
    #更新或创建
      ddl-auto: update
    show-sql: true
```

4、编写Controller测试

```
@RestController
public class UserController {
    @Autowired
    UserRepository userRepository;

    @GetMapping("/user/{id}")
    public User getUser(@PathVariable("id") Integer id){
        User user = userRepository.findOne(id);
        return user;
    }
```

```
    @GetMapping("/insert")
    public User insertUser(User user){
        User user1 = userRepository.save(user);
        return  user1;
    }
}
```

七、启动配置原理

几个重要的事件回调机制

加载配置文件META-INF/spring.factories

ApplicationContextInitializer

SpringApplicationRunListener

ioc容器中

ApplicationRunner

CommandLineRunner

启动流程

1、创建SpringApplicaiotn对象

```
private void initialize(Object[] sources) {
    //保存主配置类
   if (sources != null && sources.length > 0) {
      this.sources.addAll(Arrays.asList(sources));
   }
    //判断当前是否是个web应用
   this.webEnvironment = deduceWebEnvironment();
    //从类路径下找到META-INF/spring.factories配置中的所有ApplicationInitializer 然后保存起来
   setInitializers((Collection) getSpringFactoriesInstances(
         ApplicationContextInitializer.class));
    //从类路径下找到META-INF/spring.factories配置中的所有ApplicationListener 然后保存起来
   setListeners((Collection) getSpringFactoriesInstances(ApplicationListener.class));
    //决定哪一个是主程序
   this.mainApplicationClass = deduceMainApplicationClass();
}
```

ApplicationInitializer

ApplicationListener

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.factory
   SpringApplicationRunListeners listeners = getRunListeners(args);
    //回调所有的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环境准备完成
       //打印SpringBoot图标
      Banner printedBanner = printBanner(environment);
       //创建ApplicationContext，决定创建web的ioc容器还是普通的ioc
      context = createApplicationContext();
       //异常分析
      analyzers = new FailureAnalyzers(context);
       //重点：将environment保存的ioc中，applyInitializers初始化器上面那6个的获取，并且回调
ApplicationContextInitializer.initialize方法

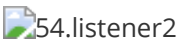
       //回调所有的SpringApplicationRunListener的contextPrepare()
       //告诉prepareContext运行完成以后回调所有的SpringApplicationRunListener的contextLoaded
      prepareContext(context, environment, listeners, applicationArguments,
            printedBanner);
       //重要：刷新所有组件 ioc容器初始化，如果是web应用还会创建嵌入式的tomcat
       //扫描 创建加载所有组件的地方
      refreshContext(context);
       //从ioc中获取所有的ApplicationRunner和CommandLineRunner
       //ApplicationRunner先回调
      afterRefresh(context, applicationArguments);
       //所有的SpringApplicationRunListener回调finished方法
      listeners.finished(context, null);
       //保存应用状态
      stopWatch.stop();
      if (this.logStartupInfo) {
         new StartupInfoLogger(this.mainApplicationClass)
               .logStarted(getApplicationLog(), stopWatch);
      }
       //整个springboot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

新建listener监听

文件目录



1、HelloApplicationContextInitializer

```
//泛型监听ioc容器
public class HelloApplicationContextInitializer implements
ApplicationContextInitializer<ConfigurableApplicationContext> {
    @Override
    public void initialize(ConfigurableApplicationContext applicationContext) {
        System.out.println("ApplicationContextInitializer...跑起来
了....."+applicationContext);
    }
}
```

2、HelloSpringApplicationRunListener

加构造器

```
public class HelloSpringApplicationRunListener implements SpringApplicationRunListener {

    public HelloSpringApplicationRunListener(SpringApplication application, String[] args){

    }

    @Override
    public void starting() {
        System.out.println("监听容器开始......");
    }

    @Override
    public void environmentPrepared(ConfigurableEnvironment environment) {
        System.out.println("环境准备好
了......"+environment.getSystemProperties().get("os.name"));
    }

    @Override
    public void contextPrepared(ConfigurableApplicationContext context) {
        System.out.println("ioc容器准备好了......");
    }
```

```
    @Override
    public void contextLoaded(ConfigurableApplicationContext context) {
        System.out.println("容器环境已经加载完成......");
    }

    @Override
    public void finished(ConfigurableApplicationContext context, Throwable exception) {
        System.out.println("全部加载完成......");
    }
}
```

3、HelloApplicationRunner

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

4、HelloCommandLineRunner

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

事件运行方法

HelloApplicationContextInitializer和HelloSpringApplicationRunListener文件META-INF/spring.factories中加入

```
# Initializers
org.springframework.context.ApplicationContextInitializer=\
com.wdjr.springboot.listener.HelloApplicationContextInitializer

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

HelloApplicationRunner和HelloCommandLineRunner ioc加入

@Component

八、SpringBoot的自定义starter

starter：场景启动器

1、场景需要使用什么依赖?

2、如何编写自动配置

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

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

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
# Auto Configure
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
org.springframework.boot.autoconfigure.admin.SpringApplicationAdminJmxAutoConfiguration,\
org.springframework.boot.autoconfigure.aop.AopAutoConfiguration,\
```

3、模式

启动器空的jar只需要做依赖管理导入；

专门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starter

xxx-spring-boot-starter

新建一个starter

> 绕的你怀疑人生

1、新建一个空项目工程

 56.starter01

2、项目命名

 57.starter02

3、导入module

 58.starter03

4、新建一个Maven工程

 59.starter04

5、项目命名

 60.starter05

 61.starter06

6、在新建一个autoconfiguration类的spring

 62.starter07

7、项目命名

 63.starter08

8、无需导入依赖

 64.starter09

9、next

 65.starter10

最后配置完成

2、编写starter

autoconfigurer

对**lxy-spring-boot-starter-autoconfigurer**进行删减

目录

 66.starter-build01

2、pom文件修改

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

    </dependencies>


</project>
```

3、编写相关的类

 67.starter-build02

4、HelloProperties

```
package com.lxy.starter;
```

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

@ConfigurationProperties(prefix = "lxy.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;
    }
}
```

5、HelloService

```
package com.lxy.starter;

public class HelloService {

    HelloProperties helloProperties;

    public HelloProperties getHelloProperties() {
        return helloProperties;
    }

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

    public String sayHello(String name){
        return helloProperties.getPrefix()+name+helloProperties.getSuffix();
    }
}
```

6、HelloServiceAutoConfiguration

```
package com.lxy.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
@EnableConfigurationProperties(HelloProperties.class)
public class HelloServiceAutoConfiguration {

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

}
```

7、配置文件

```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
com.lxy.starter.HelloServiceAutoConfiguration
```

8、修改lxy-spring-boot-starter 也就是之前的Maven项目，修改pom文件引入autoconfiguration依赖

```
<dependencies>
    <dependency>
        <groupId>com.lxy.starter</groupId>
        <artifactId>lxy-spring-boot-starter-autoconfigurer</artifactId>
        <version>0.0.1-SNAPSHOT</version>
    </dependency>
</dependencies>
```

9、install生成

 68.starter-build03

3、测试

新建一个springboot 1.5+web

1、引入starter

```
    <dependency>
        <groupId>com.lxy.starter</groupId>
        <artifactId>lxy-spring-boot-starter</artifactId>
        <version>1.0-SNAPSHOT</version>
    </dependency>

</dependencies>
```

2、新建一个Controller用来测试

```
@RestController
public class HelloController {

    @Autowired
    HelloService helloService;

    @GetMapping
    public  String hello(){
        return helloService.sayHello("test");
    }
}
```

3、编写配置文件制定前缀和后缀名

```
lxy.hello.prefix=Starter-
lxy.hello.suffix=-Success
```

4、运行访问http://localhost:8080/hello

 70.starter-build05

成功爽啊

九、SpringBoot缓存

缓存的场景

- 临时性数据存储【校验码】
- 避免频繁因为相同的内容查询数据库【查询的信息】

1、JSR107缓存规范

> 用的比较少

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间访问多个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CacheManage的上下文中，一个CacheManage只被一个CachingProvider拥有

- Cache

  类似于Map的数据结构并临时储存以key为索引的值，一个Cache仅仅被一个CacheManage所拥有

- Entry

  存储在Cache中的key-value对

- Expiry

  存储在Cache的条目有一个定义的有效期，一旦超过这个时间，就会设置过期的状态，过期无法被访问，更新，删除。缓存的有效期可以通过ExpiryPolicy设置。

   35.cache

2、Spring的缓存抽象

包括一些JSR107的注解

CahceManager

Cache

1、基本概念

重要的概念&缓存注解

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

2、整合项目

1、新建一个SpringBoot1.5+web+mysql+mybatis+cache

2、编写配置文件，连接Mysql

```
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
spring.datasource.url=jdbc:mysql://192.168.179.131:3306/mybatis01
spring.datasource.username=root
spring.datasource.password=Welcome_1
mybatis.configuration.map-underscore-to-camel-case=true
server.port=9000
```

3、创建一个bean实例

Department

```
package com.wdjr.cache.bean;

public class Department {
    private Integer id;
    private String deptName;

    public Department(){

    }

    public Department(Integer id, String deptName) {
        this.id = id;
        this.deptName = deptName;
    }

    public Integer getId() {
        return id;
    }

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

    public String getDeptName() {
        return deptName;
    }

    public void setDeptName(String deptName) {
        this.deptName = deptName;
    }

    @Override
    public String toString() {
        return "Department{" +
                "id=" + id +
                ", deptName='" + deptName + '\'' +
                '}';
    }
}
```

Employee

```
package com.wdjr.cache.bean;

public class Employee {
    private Integer id;
    private String lastName;
    private String gender;
    private String email;
    private Integer dId;

    public Integer getId() {
        return id;
    }

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

    public String getLastName() {
        return lastName;
    }

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

    public String getGender() {
        return gender;
    }

    public void setGender(String gender) {
        this.gender = gender;
    }

    public String getEmail() {
        return email;
    }

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

    public Integer getdId() {
        return dId;
    }

    public void setdId(Integer dId) {
        this.dId = dId;
    }

    @Override
    public String toString() {
```

```
        return "Employee{" +
                "id=" + id +
                ", lastName='" + lastName + '\'' +
                ", gender='" + gender + '\'' +
                ", email='" + email + '\'' +
                ", dId=" + dId +
                '}';
    }
}
```

4、创建mapper接口映射数据库，并访问数据库中的数据

```
package com.wdjr.cache.mapper;


import com.wdjr.cache.bean.Employee;
import org.apache.ibatis.annotations.Mapper;
import org.apache.ibatis.annotations.Select;
import org.apache.ibatis.annotations.Update;

@Mapper
public interface EmployeeMapper {

    @Select("SELECT * FROM employee WHERE id = #{id}")
    public Employee getEmpById(Integer id);
    @Update("UPDATE employee SET lastName=#{lastName},email=#{email},gender=#{gender},d_id=#
{dId} WHERE id=#{id}")
    public void updateEmp(Employee employee);
}
```

5、主程序添加注解MapperScan，并且使用@EnableCaching开启缓存

```
package com.wdjr.cache;

import org.mybatis.spring.annotation.MapperScan;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cache.annotation.EnableCaching;

@EnableCaching
@MapperScan("com.wdjr.cache.mapper")
@SpringBootApplication
public class Springboot01CacheApplication {

    public static void main(String[] args) {
        SpringApplication.run(Springboot01CacheApplication.class, args);
    }
}
```

6、编写service，来具体实现mapper中的方法

```
package com.wdjr.cache.service;

import com.wdjr.cache.bean.Employee;
import com.wdjr.cache.mapper.EmployeeMapper;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.cache.annotation.Cacheable;
import org.springframework.stereotype.Service;

@Service
public class EmployeeService {

    @Autowired
    EmployeeMapper employeeMapper;

    /**
     * 将方法的运行结果进行缓存，以后要是再有相同的数据，直接从缓存中获取，不用调用方法
     * CacheManager中管理多个Cache组件，对缓存的真正CRUD操作在Cache组件中，每个缓存组件都有自己的唯一名
字;
     *
     * 属性:
     *  CacheName/value:指定存储缓存组件的名字
     *  key:缓存数据使用的key,可以使用它来指定。默认是使用方法参数的值，1-方法的返回值
     *  编写Spel表达式: #id 参数id的值， #a0/#p0 #root.args[0]
     *  keyGenerator:key的生成器，自己可以指定key的生成器的组件id
     *  key/keyGendertor二选一使用
     *
     *  cacheManager指定Cache管理器，或者cacheReslover指定获取解析器
     *  condition:指定符合条件的情况下，才缓存;
     *  unless: 否定缓存，unless指定的条件为true，方法的返回值就不会被缓存，可以获取到结果进行判断
     *  sync:是否使用异步模式，unless不支持
     *
     *
     * @param id
     * @return
     */
    @Cacheable(cacheNames = {"emp"},key = "#id",condition = "#id>0",unless =
"#result==null")
    public Employee getEmp(Integer id){
        System.out.println("查询id= "+id+"的员工");
        return employeeMapper.getEmpById(id);
    }
}
```

7、编写controller测试

```
@RestController
public class EmployeeController {
    @Autowired
    EmployeeService employeeService;

    @GetMapping("/emp/{id}")
    public Employee getEmp(@PathVariable("id")Integer id){
        return employeeService.getEmp(id);
    }
}
```

8、测试结果

 35.cache

继续访问，就不会执行方法，因为直接在缓存中取值

3、缓存原理

原理：

1、CacheAutoConfiguration

2、导入缓存组件

 36.importcache

3、查看哪个缓存配置生效

```
SimpleCacheConfiguration生效
```

4、给容器注册一个CacheManager:ConcurrentMapCacheManager

5、可以获取和创建ConcurrentMapCache,作用是将数据保存在ConcurrentMap中

运行流程

1、方法运行之前，先查Cache(缓存组件），按照cacheName的指定名字获取；

（CacheManager先获取相应的缓存），第一次获取缓存如果没有cache组件会自己创建

2、去Cache中查找缓存的内容，使用一个key，默认就是方法的参数；

key是按照某种策略生成的，默认是使用keyGenerator生成的，默认使用SimpleKeyGenerator生成key

没有参数 key=new SimpleKey()

如果有一个参数 key=参数值

如果多个参数 key=new SimpleKey(params);

3、没有查到缓存就调用目标方法

4、将目标方法返回的结果，放回缓存中

方法执行之前，@Cacheable先来检查缓存中是否有数据，按照参数的值作为key去查询缓存，如果没有，就运行方法，存入缓存，如果有数据，就取出map的值。

4、Cache的注解

1、@Cacheput

修改数据库的某个数据，同时更新缓存

运行时机

先运行方法，再将目标结果缓存起来

> cacheable的key是不能使用result的参数的

1、编写更新方法

```
@CachePut(value = {"emp"},key = "#result.id")
public Employee updateEmp(Employee employee){
    System.out.println("updateEmp"+employee);
    employeeMapper.updateEmp(employee);
    return employee;
}
```

2、编写Controller方法

```
@GetMapping("/emp")
public Employee updateEmp(Employee employee){
    employeeService.updateEmp(employee);
    return employee;
}
```

测试

测试步骤

1、先查询1号员工

2、更新1号员工数据

3、查询1号员工

可能并没有更新，

是因为查询和更新的key不同

效果：

- 第一次查询：查询mysql
- 第二次更新：更新mysql
- 第三次查询：调用内存

2、CacheEvict

清除缓存

编写测试方法

```
@CacheEvict(value = "emp",key = "#id")
public  void  deleteEmp(Integer id){
    System.out.println("delete的id"+id);
}
```

allEntries = true,代表不论清除那个key，都重新刷新缓存

beforeInvocation=true.方法执行前，清空缓存，默认是false,如果程序异常，就不会清除缓存

3、Caching

组合

- Cacheable
- CachePut
- CacheEvict

CacheConfig抽取缓存的公共配置

```
@CacheConfig(cacheNames = "emp")
@Service
public class EmployeeService {
```

然后下面的value=emp就不用写了

```
@Caching(
        cacheable = {
                @Cacheable(value = "emp",key = "#lastName")
        },
        put = {
                @CachePut(value = "emp",key = "#result.id"),
                @CachePut(value = "emp",key = "#result.gender")
        }
)
public Employee getEmpByLastName(String lastName){
    return employeeMapper.getEmpByLastName(lastName);
}
```

如果查完lastName,再查的id是刚才的值，就会直接从缓存中获取数据

5、Redis

默认的缓存是在内存中定义HashMap，生产中使用Redis的缓存中间件

Redis 是一个开源（BSD许可）的，内存中的数据结构存储系统，它可以用作数据库、缓存和消息中间件

1、安装Docker

安装redis在docker上

```
#拉取redis镜像
docker pull redis
#启动redis[bfcb1f6df2db]docker images的id
 docker run -d -p 6379:6379 --name redis01 bfcb1f6df2db
```

2、Redis的Template

Redis的常用五大数据类型

String【字符串】、List【列表】、Set【集合】、Hash【散列】、ZSet【有序集合】

分为两种一种是**StringRedisTemplate**，另一种是**RedisTemplate**

根据不同的数据类型，大致的操作也分为这5种，以StringRedisTemplate为例

```
stringRedisTemplate.opsForValue()  --String
stringRedisTemplate.opsForList()  --List
stringRedisTemplate.opsForSet()  --Set
stringRedisTemplate.opsForHash()  --Hash
stringRedisTemplate.opsForZset()  -Zset
```

1、导入依赖

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

2、修改配置文件

```
spring.redis.host=192.168.179.131
```

3、添加测试类

```
    @Autowired
    StringRedisTemplate stringRedisTemplate;//操作字符串【常用】

    @Autowired
    RedisTemplate redisTemplate;//操作k-v都是对象
    @Test
    public void test01(){
//        stringRedisTemplate.opsForValue().append("msg", "hello");
        String msg = stringRedisTemplate.opsForValue().get("msg");
        System.out.println(msg);
    }
```

写入数据

 37.redis

读取数据

 38.redis02

3、测试保存对象

> 对象需要序列化

1、序列化bean对象

```
public class Employee implements Serializable {
```

2、将对象存储到Redis

```
@Test
public  void test02(){
    Employee emp = employeeMapper.getEmpById(2);
    redisTemplate.opsForValue().set("emp-01", emp);
}
```

3、效果演示

 38.redis03

4、以json方式传输对象

1、新建一个Redis的配置类MyRedisConfig,

```
@Configuration
public class MyRedisConfig {
    @Bean
    public RedisTemplate<Object, Employee> empRedisTemplate(
            RedisConnectionFactory redisConnectionFactory)
            throws UnknownHostException {
        RedisTemplate<Object, Employee> template = new RedisTemplate<Object, Employee>();
        template.setConnectionFactory(redisConnectionFactory);
        Jackson2JsonRedisSerializer<Employee> jsonRedisSerializer = new
Jackson2JsonRedisSerializer<Employee>(Employee.class);
        template.setDefaultSerializer(jsonRedisSerializer);
        return template;
    }
```

2、编写测试类

```
 @Autowired
 RedisTemplate<Object,Employee> empRedisTemplate;
@Test
public  void test02(){
    Employee emp = employeeMapper.getEmpById(2);
    empRedisTemplate.opsForValue().set("emp-01", emp);

}
```

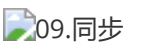
3、测试效果

十、SpringBoot的消息中间件

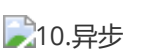
1、JMS&AMQP简介

1、异步处理

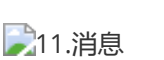
同步机制



并发机制



消息队列机制



2、应用解耦

使用中间件，将两个服务解耦，一个写入，一个订阅

3、流量削锋

例如消息队列的FIFO，限定元素的长度，防止出现多次请求导致的误操作

概述

1、大多数应用，可以通过消息服务中间件来提升系统的异步通信、拓展解耦能力

2、消息服务中的两个重要概念：

消息代理（message broker）和目的地（destination）,当消息发送者发送消息以后，将由消息代理接管，消息代理保证消息传递到指定的目的地。

3、消息队列主要的两种形式的目的地

1）、队列（queue）：点对点消息通信【point-to-point】，取出一个没一个，一个发布，多个消费

2）、主题（topic）:发布（publish）/订阅（subscribe）消息通信，多人【订阅者】可以同时接到消息

4、JMS(Java Message Service) Java消息服务：

- 基于JVM消息规范的代理。ActiveMQ/HornetMQ是JMS的实现

5、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

	JMS	**AMQP**
定义	Java API	网络线级协议
跨平台	否	是
跨语言	否	是
Model	(1)、Peer-2-Peer (2)、Pub/Sub	(1)、direct exchange (2)、fanout exchange (3)、topic change (4)、headers exchange (5)、system exchange 后四种都是pub/sub ,差别路由机制做了更详细的划分
支持消息类型	TextMessage MapMessage ByteMessage StreamMessage ObjectMessage Message	byte[]通常需要序列化

6、SpringBoot的支持

spring-jms提供了对JMS的支持

spring-rabbit提供了对AMQP的支持

需要创建ConnectionFactory的实现来连接消息代理

提供JmsTemplate,RabbitTemplate来发送消息

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

@EnableJms,@EnableRabbit开启支持

7、SpringBoot的自动配置

- JmsAutoConfiguration
- RabbitAutoConfiguration

2、RabbitMQ简介

> AMQP的实现

1、核心概念

Message:消息头和消息体组成，消息体是不透明的，而消息头上则是由一系列的可选属性组成，属性：路由键【routing-key】,优先级【priority】,指出消息可能需要持久性存储【delivery-mode】

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的4中类型：direct【默认】点对点，fanout,topic和headers, 发布订阅，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数据取走。

Binding:绑定，队列和交换机之间的关联，多对多关系

Connection:网络连接，例如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，信道是建立在真是的TCP链接之内的虚拟连接AMQP命令都是通过信道发送出去的。不管是发布消息，订阅队列还是接受消息，都是信道，减少TCP的开销，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的 应用程序

VirtualHost:小型的rabbitMQ,相互隔离

Broker:表示消息队列 服务实体

 13.RabbitMQ结构

2、RabbitMQ的运行机制

Exchange的三种方式

direct：根据路由键直接匹配，一对一

 14.RabbitMQDirect

fanout:不经过路由键，直接发送到每一个队列

 14.RabbitMQfaout

topic:类似模糊匹配的根据路由键，来分配绑定的队列

 14.RabbitMQtopic

3、RabbitMQ安装测试

1、打开虚拟机，在docker中安装RabbitMQ

```
#1.安装rabbitmq，使用镜像加速
docker pull registry.docker-cn.com/library/rabbitmq:3-management
[root@node1 ~]# docker images
REPOSITORY                                      TAG                 IMAGE ID
CREATED             SIZE
registry.docker-cn.com/library/rabbitmq         3-management        c51d1c73d028        11
days ago         149 MB
#2.运行rabbitmq
##### 端口：5672 客户端和rabbitmq通信 15672：管理界面的web页面

docker run -d -p 5672:5672 -p 15672:15672 --name myrabbitmq c51d1c73d028

#3.查看运行
docker ps
```

2、打开网页客户端并登陆，账号【guest】,密码【guest】，登陆

 13.rabbitmq

3、添加 【direct】【faout】【topic】的绑定关系等

1）、添加Exchange,分别添加**exchange.direct**、**exchange.fanout**、**exchange.topic**

 15.exchanges

2）、添加 Queues,分别添加**lxy.news、wdjr、wdjr.emps、wdjr.news**

 16.queues

3）、点击【exchange.direct】添加绑定规则

 17.bind

4）、点击【exchange.fanout】添加绑定规则

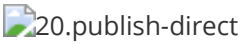
 18,bindfanout

5）、点击【exchange.topic】添加绑定规则

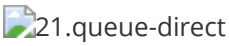
> /*: 代表匹配1个单词
>
> /#：代表匹配0个或者多个单词

4、发布信息测试

【direct】发布命令，点击 Publish message

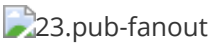


查看队列的数量

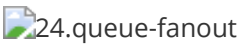


点击查看发送的信息

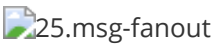
【fanout】的发布消息



队列信息

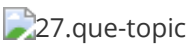


随意一个数据信息例如：wdjr.emp



【topic】发布信息测试

队列的值



信息查看

4、创建工程整合

```
1、RabbitAutoConfiguration
2、自动配置了连接工厂 ConnectionFactory
3、RabbitProperties封装了 RabbitMQ
4、RabbitTemplate:给RabbitMQ发送和接受消息的
5、AmqpAdmin：RabbitMQ的系统管理功能组件
```

1、RabbitTemplate

1、新建SpringBoot工程，SpringBoot1.5+Integeration/RabbitMQ+Web

2、RabbitAutoConfiguration文件

3、编写配置文件application.yml

```
spring:
  rabbitmq:
    host: 192.168.179.131
    port: 5672
    username: guest
    password: guest
```

4、编写测试类,将HashMap写入Queue

```
 @Autowired
    RabbitTemplate rabbitTemplate;

    @Test
    public void contextLoads() {
        //Message需要自己构建一个；定义消息体内容和消息头
        // rabbitTemplate.send(exchange, routingKey, message);
        //Object 默认当成消息体，只需要传入要发送的对象，自动化序列发送给rabbitmq；
        Map<String,Object> map = new HashMap<>();
        map.put("msg", "这是第一个信息");
        map.put("data", Arrays.asList("helloWorld",123,true));
        //对象被默认序列以后发送出去
        rabbitTemplate.convertAndSend("exchange.direct","wdjr.news",map);
    }
```

5、查看网页的信息

 29.dir-idea

6、取出队列的值

> 取出队列中数据就没了

```
@Test
public void reciverAndConvert(){

    Object o = rabbitTemplate.receiveAndConvert("wdjr.news");
    System.out.println(o.getClass());
    System.out.println(o);

}
```

结果

```
class java.util.HashMap
{msg=这是第一个信息, data=[helloWorld, 123, true]}
```

7、使用Json方式传递，并传入对象Book

1）、MyAMQPConfig

```
@Configuration
public class MyAMQPConfig  {

    @Bean
    public MessageConverter messageConverter(){
        return new Jackson2JsonMessageConverter();
    }
}
```

2）、编写Book实体类

```
package com.wdjr.amqp.bean;

public class Book {
    private String  bookName;
    private String author;

    public Book(){

    }

    public Book(String bookName, String author) {
        this.bookName = bookName;
        this.author = author;
    }

    public String getBookName() {
        return bookName;
    }

    public void setBookName(String bookName) {
        this.bookName = bookName;
    }

    public String getAuthor() {
        return author;
    }

    public void setAuthor(String author) {
        this.author = author;
    }

    @Override
```

```
    public String toString() {
        return "Book{" +
                "bookName='" + bookName + '\'' +
                ", author='" + author + '\'' +
                '}';
    }
}
```

3）、测试类

```
@Test
public void contextLoads() {
    //对象被默认序列以后发送出去
    rabbitTemplate.convertAndSend("exchange.direct","wdjr.news",new Book("百年孤独", "季羡林"));
}
```

4）、查看wdjr.news

 30.dir-idea-json

5）、取出数据

```
@Test
public void reciverAndConvert(){
    Object o = rabbitTemplate.receiveAndConvert("wdjr.news");
    System.out.println(o.getClass());
    System.out.println(o);
}
```

6）、结果演示

```
class com.wdjr.amqp.bean.Book
Book{bookName='百年孤独', author='季羡林'}
```

2、开启基于注解的方式

1、新建一个BookService

```
@Service
public class BookService {
    @RabbitListener(queues = "wdjr.news")
    public void receive(Book book){
        System.out.println(book);
    }

    @RabbitListener(queues = "wdjr")
    public void receive02(Message message){
```

```
        System.out.println(message.getBody());
        System.out.println(message.getMessageProperties());
    }
}
```

2、主程序开启RabbitMQ的注解

```
@EnableRabbit //开启基于注解的rabbitmq
@SpringBootApplication
public class AmqpApplication {

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

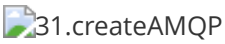
3、AmqpAdmin

> 创建和删除 Exchange 、Queue、Bind

1）、创建Exchange

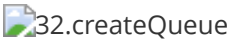
```
@Test
public void createExchange(){
    amqpAdmin.declareExchange(new DirectExchange("amqpadmin.direct"));
    System.out.println("Create Finish");
}
```

效果演示



2）、创建Queue

```
@Test
public void createQueue(){
    amqpAdmin.declareQueue(new Queue("amqpadmin.queue",true));
    System.out.println("Create Queue Finish");
}
```



3）、创建Bind规则

```
@Test
public void createBind(){
    amqpAdmin.declareBinding(new Binding("amqpadmin.queue",Binding.DestinationType.QUEUE ,
"amqpadmin.direct", "amqp.haha", null));
}
```


33.createBinding

删除类似

```
@Test
public void deleteExchange(){
    amqpAdmin.deleteExchange("amqpadmin.direct");
    System.out.println("delete Finish");
}
```

十一、SpringBoot的检索

1、ElasticSearch简介

ElasticSearch是一个基于Lucene的搜索服务器。它提供了一个分布式多用户能力的全文搜索引擎，基于RESTful web接口。Elasticsearch是用Java开发的，并作为Apache许可条款下的开放源码发布，是当前流行的企业级搜索引擎。设计用于云计算中，能够达到实时搜索，稳定，可靠，快速，安装使用方便。

2、ElasticSearch的安装

1、安装java最新版本

- 下载linux的.tar.gz
- 解压到指定目录
- 配置环境变量

2、安装Docker(非必须这是是在Docker中安装)

```
1、查看centos版本
# uname -r
3.10.0-693.el7.x86_64
要求：大于3.10
如果小于的话升级*（选做）
# yum update
2、安装docker
# yum install docker
3、启动docker
# systemctl start docker
# docker -v
4、开机启动docker
# systemctl enable docker
5、停止docker
```

```
# systemctl stop docker
```

3、安装ElasticSearch的镜像

```
docker pull registry.docker-cn.com/library/elasticsearch
```

4、运行ElasticSearch

> -e ES_JAVA_OPTS="-Xms256m -Xmx256m" 表示占用的最大内存为256m,默认是2G

```
[root@node1 ~]# docker images
REPOSITORY                                    TAG                 IMAGE ID
CREATED             SIZE
registry.docker-cn.com/library/elasticsearch   latest              671bb2d7da44
32 hours ago        486 MB
[root@node1 ~]#
[root@node1 ~]# docker run -e ES_JAVA_OPTS="-Xms256m -Xmx256m" -d -p 9200:9200 -p 9300:9300
--name ES01 671bb2d7da44
```

5、测试是否启动成功

访问9200端口：http://192.168.179.131:9200/ 查看是否返回json数据

```
{
  "name" : "onB-EUU",
  "cluster_name" : "elasticsearch",
  "cluster_uuid" : "j3SXX6tdThWUomW3tAvDFg",
  "version" : {
    "number" : "5.6.9",
    "build_hash" : "877a590",
    "build_date" : "2018-04-12T16:25:14.838Z",
    "build_snapshot" : false,
    "lucene_version" : "6.6.1"
  },
  "tagline" : "You Know, for Search"
}
```

3、Elastic的快速入门

> 最好的工具就是官方文档，以下操作都在文档中进行操作。

1、基础概念

面向文档，JSON作为序列化格式，ElasticSearch的基本概念

索引（名词）：

如前所述，一个 *索引* 类似于传统关系数据库中的一个 *数据库* ，是一个存储关系型文档的地方。 *索引* (*index*) 的复数词为 *indices* 或 *indexes* 。

索引（动词）：

索引一个文档 就是存储一个文档到一个 *索引* （名词）中以便它可以被检索和查询到。这非常类似于 SQL 语句中的 `INSERT` 关键词，除了文档已存在时新文档会替换旧文档情况之外。

类型：相当于数据库中的表

文档：相当于数据库中的行，即每条数据都叫一个文档

属性：相当于数据库中的列，即文档的属性

2、测试

下载POSTMAN，并使用POSTMAN测试

1、插入数据

具体信息查看官方示例

重点：PUT请求+请求体

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

 01.postman

2、检索文档

官方示例

重点：GET请求+URI+index+type+ID

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

 02.postmanget

3、轻量检索

官方文档

重点：GET请求+index+type+_search+条件（非必须）

搜索所有雇员： `_search`

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

高亮搜索：URL参数

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

4、使用查询表达式

官方文档

重点：GET+URI+index+type+_search+请求体【match】

Query-string 搜索通过命令非常方便地进行临时性的即席搜索，但它有自身的局限性（参见 *轻量搜索*）。Elasticsearch 提供一个丰富灵活的查询语言叫做 *查询表达式*，它支持构建更加复杂和健壮的查询。

领域特定语言（DSL），指定了使用一个 JSON 请求。我们可以像这样重写之前的查询所有 Smith 的搜索：

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

返回结果与之前的查询一样，但还是可以看到有一些变化。其中之一是，不再使用 *query-string* 参数，而是一个请求体替代。这个请求使用 JSON 构造，并使用了一个 `match` 查询（属于查询类型之一，后续将会了解）。

5、更加复杂的查询

官方文档

重点：GET+URI+index+type+_search + 请求体【match+filter】

现在尝试下更复杂的搜索。 同样搜索姓氏为 Smith 的雇员，但这次我们只需要年龄大于 30 的。查询需要稍作调整，使用过滤器 *filter*，它支持高效地执行一个结构化查询。

```
GET /megacorp/employee/_search
{
    "query" : {
        "bool": {
            "must": {
                "match" : {
                    "last_name" : "smith"
                }
            },
            "filter": {
                "range" : {
                    "age" : { "gt" : 30 }
```

```
                }
            }
        }
    }
}
```

❶	**这部分与我们之前使用的 `match` *查询* 一样。**
❷	这部分是一个 `range` *过滤器*， 它能找到年龄大于 30 的文档，其中 `gt` 表示*大于*(great than)。

目前无需太多担心语法问题，后续会更详细地介绍。只需明确我们添加了一个 *过滤器* 用于执行一个范围查询，并复用之前的 `match` 查询。现在结果只返回了一个雇员，叫 Jane Smith，32 岁。

6、全文搜索

官方文档

重点：GET+index+type+_search+请求体【match】 ==》看相关性得分

截止目前的搜索相对都很简单：单个姓名，通过年龄过滤。现在尝试下稍微高级点儿的全文搜索——一项传统数据库确实很难搞定的任务。

搜索下所有喜欢攀岩（rock climbing）的雇员：

```
GET /megacorp/employee/_search
{
    "query" : {
        "match" : {
            "about" : "rock climbing"
        }
    }
}
```

显然我们依旧使用之前的 `match` 查询在 `about` 属性上搜索 “rock climbing” 。得到两个匹配的文档：

```
{
   ...
   "hits": {
      "total":      2,
      "max_score":  0.16273327,
      "hits": [
         {
            ...
            "_score":         0.16273327,
            "_source": {
               "first_name":  "John",
               "last_name":   "Smith",
               "age":         25,
               "about":       "I love to go rock climbing",
               "interests": [ "sports", "music" ]
```

```
            }
         },
         {
            ...
            "_score":         0.016878016,
            "_source": {
               "first_name":  "Jane",
               "last_name":   "Smith",
               "age":         32,
               "about":       "I like to collect rock albums",
               "interests": [ "music" ]
            }
         }
      ]
   }
}
```

> "_score":相关性得分

Elasticsearch 默认按照相关性得分排序，即每个文档跟查询的匹配程度。第一个最高得分的结果很明显：John Smith 的 `about` 属性清楚地写着 “rock climbing” 。

但为什么 Jane Smith 也作为结果返回了呢？原因是她的 `about` 属性里提到了 “rock” 。因为只有 “rock” 而没有 “climbing” ，所以她的相关性得分低于 John 的。

这是一个很好的案例，阐明了 Elasticsearch 如何 *在* 全文属性上搜索并返回相关性最强的结果。Elasticsearch中的 *相关性* 概念非常重要，也是完全区别于传统关系型数据库的一个概念，数据库中的一条记录要么匹配要么不匹配。

7、短语搜索

官方文档

重点：GET+index+type+_search+请求体【match_phrase 】

找出一个属性中的独立单词是没有问题的，但有时候想要精确匹配一系列单词或者 *短语* 。 比如， 我们想执行这样一个查询，仅匹配同时包含 “rock” *和* “climbing” ， *并且* 二者以短语 “rock climbing” 的形式紧挨着的雇员记录。

为此对 `match` 查询稍作调整，使用一个叫做 `match_phrase` 的查询：

```
GET /megacorp/employee/_search
{
    "query" : {
        "match_phrase" : {
            "about" : "rock climbing"
        }
    }
}
```

返回的信息

```
{
   ...
```

```
   "hits": {
      "total":      1,
      "max_score":  0.23013961,
      "hits": [
         {
            ...
            "_score":         0.23013961,
            "_source": {
               "first_name":  "John",
               "last_name":   "Smith",
               "age":         25,
               "about":       "I love to go rock climbing",
               "interests": [ "sports", "music" ]
            }
         }
      ]
   }
}
```

8、高亮搜索

官方地址

重点：GET+index+type+_search+请求体【match_phrase+highlight】==>返回关键字加了em标签

许多应用都倾向于在每个搜索结果中 *高亮* 部分文本片段，以便让用户知道为何该文档符合查询条件。在 Elasticsearch 中检索出高亮片段也很容易。

再次执行前面的查询，并增加一个新的 `highlight` 参数：

```
GET /megacorp/employee/_search
{
    "query" : {
        "match_phrase" : {
            "about" : "rock climbing"
        }
    },
    "highlight": {
        "fields" : {
            "about" : {}
        }
    }
}
```

当执行该查询时，返回结果与之前一样，与此同时结果中还多了一个叫做 `highlight` 的部分。这个部分包含了 `about` 属性匹配的文本片段，并以 HTML 标签 `<em></em>` 封装：

```
{
   ...
   "hits": {
      "total":      1,
```

```
      "max_score":  0.23013961,
      "hits": [
         {
            ...
            "_score":         0.23013961,
            "_source": {
               "first_name":  "John",
               "last_name":   "Smith",
               "age":         25,
               "about":       "I love to go rock climbing",
               "interests": [ "sports", "music" ]
            },
            "highlight": {
               "about": [
                  "I love to go <em>rock</em> <em>climbing</em>"
               ]
            }
         }
      ]
   }
}
```

9、分析

官方文档

重点：GET+index+type+_search+请求体【aggs-field】

> aggs：聚合

终于到了最后一个业务需求：支持管理者对雇员目录做分析。 Elasticsearch 有一个功能叫聚合（aggregations），允许我们基于数据生成一些精细的分析结果。聚合与 SQL 中的 `GROUP BY` 类似但更强大。

举个例子，挖掘出雇员中最受欢迎的兴趣爱好：

```
GET /megacorp/employee/_search
{
  "aggs": {
    "all_interests": {
      "terms": { "field": "interests" }
    }
  }
}
```

会报错

Fielddata is disabled on text fields by default. Set fielddata=true on [inte

默认情况下，字段数据在文本字段上禁用。设置字段数据= TRUE

首先开启数据结构

```
PUT megacorp/_mapping/employee/
{
  "properties": {
    "interests": {
      "type":     "text",
      "fielddata": true
    }
  }
}
```

然后在进行请求

```
{
   ...
   "hits": { ... },
   "aggregations": {
      "all_interests": {
         "buckets": [
            {
               "key":       "music",
               "doc_count": 2
            },
            {
               "key":       "forestry",
               "doc_count": 1
            },
            {
               "key":       "sports",
               "doc_count": 1
            }
         ]
      }
   }
}
```

可以看到，两位员工对音乐感兴趣，一位对林地感兴趣，一位对运动感兴趣。这些聚合并非预先统计，而是从匹配当前查询的文档中即时生成。

如果想知道叫 Smith 的雇员中最受欢迎的兴趣爱好，可以直接添加适当的查询来组合查询：

```
GET /megacorp/employee/_search
{
  "query": {
    "match": {
      "last_name": "smith"
    }
  },
  "aggs": {
    "all_interests": {
      "terms": {
```

```
        "field": "interests"
      }
    }
  }
}
```

`all_interests` 聚合已经变为只包含匹配查询的文档：

```
...
 "all_interests": {
    "buckets": [
       {
          "key": "music",
          "doc_count": 2
       },
       {
          "key": "sports",
          "doc_count": 1
       }
    ]
 }
```

聚合还支持分级汇总 。比如，查询特定兴趣爱好员工的平均年龄：

```
GET /megacorp/employee/_search
{
    "aggs" : {
        "all_interests" : {
            "terms" : { "field" : "interests" },
            "aggs" : {
                "avg_age" : {
                    "avg" : { "field" : "age" }
                }
            }
        }
    }
}
```

输出基本是第一次聚合的加强版。依然有一个兴趣及数量的列表，只不过每个兴趣都有了一个附加的 `avg_age` 属性，代表有这个兴趣爱好的所有员工的平均年龄。

即使现在不太理解这些语法也没有关系，依然很容易了解到复杂聚合及分组通过 Elasticsearch 特性实现得很完美。可提取的数据类型毫无限制。

4、SpringBoot+ElasticSearch

1、新建项目SpringBoot1.5+Web+Nosql-->ElasticSearch

2、springBoot默认支持两种技术和ES进行交互

1、Jest【需要导入使用】

利用JestClient和服务器的9200端口进行http通信

2、SpringData ElasticSearch【默认】

1）、客户端:Client节点信息: clusterNodes: clusterName

2）、ElasticsearchTemplate操作es

3）、编写ElasticsearchRepository子接口

1、Jest

1、注释SpringDataElasticSearch的依赖，并导入Jest【5.xx】的相关依赖

```
        <!--   SpringData管理ElasticSearch   -->
<!--        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-elasticsearch</artifactId>
        </dependency>-->

        <!-- https://mvnrepository.com/artifact/io.searchbox/jest -->
        <dependency>
            <groupId>io.searchbox</groupId>
            <artifactId>jest</artifactId>
            <version>5.3.3</version>
        </dependency>
```

2、修改配置文件application.yml

```
spring:
  elasticsearch:
    jest:
      uris: http://192.168.179.131:9200
```

3、创建 bean.Article

```
package com.wdjr.springboot.bean;

import io.searchbox.annotations.JestId;

public class Article {

    @JestId
    private Integer id;
    private String autor;
    private String title;
    private String content;

    public Integer getId() {
```

```
        return id;
    }

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

    public String getAutor() {
        return autor;
    }

    public void setAutor(String autor) {
        this.autor = autor;
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public String getContent() {
        return content;
    }

    public void setContent(String content) {
        this.content = content;
    }
}
```

4、运行程序

5、编写Jest Cilent的测试类

> 向wdjr-article中插入数据

```
@Autowired
JestClient jestClient;

@Test
public void contextLoads() {
    //1、给Es中索引（保存）一个文档
    Article article = new Article();
    article.setId(2);
    article.setTitle("好消息");
    article.setAutor("zhangsan");
    article.setContent("Hello World");
    //构建一个索引功能
    Index index = new Index.Builder(article).index("wdjr").type("article").build();
```

```
    try {
        //执行
        jestClient.execute(index);
    } catch (IOException e) {
        e.printStackTrace();
    }

}
```

> 查询数据

```
@Test
public void search(){
    //查询表达式
    String json = "{\n" +
            "    \"query\" : {\n" +
            "        \"match\" : {\n" +
            "            \"content\" : \"Hello\"\n" +
            "        }\n" +
            "    }\n" +
            "}";
    //构建搜索操作
    Search search = new Search.Builder(json).addIndex("wdjr").addType("article").build();

    //执行
    try {
        SearchResult result = jestClient.execute(search);
        System.out.println(result.getJsonString());
    } catch (IOException e) {
        e.printStackTrace();
    }

}
```

2、SpringData-Elastic

1、下载对应版本的ElasticSearch

如果版本不适配，会报错，解决方案：升级SpringBoot版本，或者安装合适的ES

spring data elasticsearch	elasticsearch
3.1.x	6.2.2
3.0.x	5.5.0
2.1.x	2.4.0
2.0.x	2.2.0
1.3.x	1.5.2

2、在Docker中安装适合版本的ES【2.4.6】

```
docker pull elasticsearch:2.4.6
docker run -e ES_JAVA_OPTS="-Xms256m -Xmx256m" -d -p 9201:9200 -p 9301:9300 --name ES02
elasticsearch:2.4.6
```

3、编写配置文件

```
spring:
  data:
    elasticsearch:
      cluster-name: elasticsearch
      cluster-nodes: 192.168.179.131:9301
```

4、运行主程序

5、操作ElasticSearch有两种方式

1）、编写一个ElasticsearchRepositry

2）、编写一个ElasticsearchTemplate

6、ElasticsearchRepositry的操作

1）、新建一个bean/Book类

```
@Document(indexName = "wdjr",type="book")
public class Book {

    private Integer id;
    private String bookName;
    private String auto;


    public Book() {
        super();
    }

    public Book(Integer id, String bookName, String auto) {
```

```
        super();
        this.id = id;
        this.bookName = bookName;
        this.auto = auto;
    }

    public Integer getId() {
        return id;
    }

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

    public String getBookName() {
        return bookName;
    }

    public void setBookName(String bookName) {
        this.bookName = bookName;
    }

    public String getAuto() {
        return auto;
    }

    public void setAuto(String auto) {
        this.auto = auto;
    }

    @Override
    public String toString() {
        return "Book{" +
                "id=" + id +
                ", bookName='" + bookName + '\'' +
                ", auto='" + auto + '\'' +
                '}';
    }
}
```

2）、新建一个repositry/BookRepositry

```
public interface BookRepositry extends ElasticsearchRepository<Book,Integer> {
    //自定义查询方法
    public List<Book> findByBookNameLike(String bookName);
}
```

3）、编写测试类

```
@Autowired
BookRepositry bookRepositry;
@Test
public void testSearch(){
    for (Book book : bookRepositry.findByBookNameLike("金")) {
        System.out.println(book);
    }

}
```

十二、SpringBoot的任务

十三、SpringBoot的安全

十四、SpringBoot的分布式

1、Dubbo简介

1. Dubbo是什么?

dubbo就是个服务框架，如果没有分布式的需求，其实是不需要用的，只有在分布式的时候，才有dubbo这样的分布式服务框架的需求，并且本质上是个服务调用的东东，**说白了就是个远程服务调用的分布式框架（告别Web Service模式中的WSdl，以服务者与消费者的方式在dubbo上注册）**

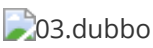
2. Dubbo能做什么?

1.透明化的远程方法调用，就像调用本地方法一样调用远程方法，只需简单配置，没有任何API侵入。

2.软负载均衡及容错机制，可在内网替代F5等硬件负载均衡器，降低成本，减少单点。

3.服务自动注册与发现，不再需要写死服务提供方地址，注册中心基于接口名查询服务提供者的IP地址，并且能够平滑添加或删除服务提供者。

3、docker的原理



调用关系说明:

0. 服务容器负责启动，加载，运行服务提供者。
1. 服务提供者在启动时，向注册中心注册自己提供的服务。
2. 服务消费者在启动时，向注册中心订阅自己所需的服务。
3. 注册中心返回服务提供者地址列表给消费者，如果有变更，注册中心将基于长连接推送变更数据给消费者。
4. 服务消费者，从提供者地址列表中，基于软负载均衡算法，选一台提供者进行调用，如果调用失败，再选另一台调用。
5. 服务消费者和提供者，在内存中累计调用次数和调用时间，定时每分钟发送一次统计数据到监控中心。

2、Zookeeper

安装Zookeeper

```
#安装zookeeper镜像
docker pull registry.docker-cn.com/library/zookeeper
#运行zookeeper
 docker run --name zk01  --restart always -d -p 2111:2181 bf5cbc9d5cac
```

3、Dubbo、Zookeeper整合

目的：完成**服务消费者**从注册中心查询调用**服务生产者**

1、将服务提供者注册到注册中心

1）、引入dubbo和zkclient的相关依赖

```
<dependency>
    <groupId>com.alibaba.boot</groupId>
    <artifactId>dubbo-spring-boot-starter</artifactId>
    <version>0.1.0</version>
</dependency>

<dependency>
    <groupId>com.github.sgroschupf</groupId>
    <artifactId>zkclient</artifactId>
    <version>0.1</version>
</dependency>
```

2）、配置service服务，新建service.TicketService 和service.TicketServiceImp

```
public interface TicketService {
    public String getTicket();
}
```

```
import com.alibaba.dubbo.config.annotation.Service;
@Component
//是dubbo包下的service
@Service
public class TicketServiceImp implements TicketService {
    @Override
    public String getTicket() {
        return "《厉害了，我的国》";
    }
}
```

3）、配置文件application.yml

```
dubbo:
  application:
    name: provider-ticket
  registry:
    address: zookeeper://192.168.179.131:2111
  scan:
    base-packages: com.wdjr.ticket.service
server:
  port: 9001
```

4）、启动服务提供者

2、启动服务消费者

1）、引入Dubbo和Zookeeper的依赖

```
<dependency>
    <groupId>com.alibaba.boot</groupId>
    <artifactId>dubbo-spring-boot-starter</artifactId>
    <version>0.1.0</version>
</dependency>

<dependency>
    <groupId>com.github.sgroschupf</groupId>
    <artifactId>zkclient</artifactId>
    <version>0.1</version>
</dependency>
```

2）、新建一个service.userService,并将TicketService的接口调用过来【全类名相同-包相同】

 03.dubbo2

```
package com.wdjr.user.service;

import com.alibaba.dubbo.config.annotation.Reference;
import com.wdjr.ticket.service.TicketService;

import org.springframework.stereotype.Service;

@Service
public class UserService {

    @Reference
    TicketService ticketService;

    public void hello(){
        String ticket = ticketService.getTicket();
        System.out.println("您已经成功买票："+ticket);
    }
```

```
}
```

3）、配置文件application.yml

```
dubbo:
  application:
    name: comsumer-user
  registry:
    address: zookeeper://192.168.179.131:2111
```

4）、编写测试类测试

```
@Autowired
UserService userService;
@Test
public void contextLoads() {
    userService.hello();
}
```

结果展示：

 04.dubbo+zk3

4、SpringCloud

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

SpringCloud分布式开发的五大常用组件

> Eureka:找到

- 服务器发现 ——Netflix Eureka
- 客服端负载均衡——Netflix Ribbon
- 断路器——Netflix Hystrix 发现不了就及时断开
- 服务网关——Netflix Zuul 过滤请求
- 分布式配置——SpringCloud Config

目的：

多个A服务调用多个B服务，负载均衡

注册中心+服务提供者+服务消费者

1、注册中心（eureka-server）

1、新建Spring项目，SpringBoot1.5+Eureka Server

2、编写application.yml

```
server:
  port: 8761
eureka:
  instance:
    hostname: eureka-server #实例的主机名
  client:
    register-with-eureka: false #不把自己注册到euraka上
    fetch-registry: false #不从euraka上来获取服务的注册信息
    service-url:
      defaultZone: http://localhost:8761/eureka/
```

3、编写主程序

```
@EnableEurekaServer
@SpringBootApplication
public class EurekaServerApplication {

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

2、服务提供者（provider-ticket）

1、新建Spring项目，SpringBoot1.5+Eureka Discovery

2、编写配置文件application.yml

```
server:
  port: 8002
spring:
  application:
    name: provider-ticket


eureka:
  instance:
    prefer-ip-address: true #注册是服务使用IP地址
  client:
```

```
    service-url:
      defaultZone: http://localhost:8761/eureka/
```

3、创建一个售票的service

```
@Service
public class TicketService {

    public String getTicket(){
        System.out.println("8001");
        return "《厉害了，我的国》";
    }
}
```

4、创建一个用于访问的controller

```
@RestController
public class TicketController {

    @Autowired
    TicketService ticketService;

    @GetMapping("/ticket")
    public String getTicket(){
        return ticketService.getTicket();
    }
}
```

5、完毕

3、服务消费者（consumer-user）

1、新建Spring项目，SpringBoot1.5+Eureka Discovery

2、编写application.yml文件

```
spring:
  application:
    name: consumer-user
server:
  port: 9001
eureka:
  instance:
    prefer-ip-address: true
  client:
    service-url:
      defaultZone: http://localhost:8761/eureka/
```

3、编写一个controller

```
@RestController
public class UserController {

    @Autowired
    RestTemplate restTemplate;
    @GetMapping("/buy")
    public String buyTicket(String name){
        String s = restTemplate.getForObject("http://PROVIDER-TICKET/ticket", String.class);
        return name+"购买了"+"  "+s;
    }
}
```

4、编写主程序

```
@EnableDiscoveryClient //开启发现服务功能
@SpringBootApplication
public class ConsumerUserApplication {

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

    }

    @LoadBalanced //使用负载均衡机制
    @Bean
    public RestTemplate restTemplate(){
        return new RestTemplate();
    }
}
```

5、完毕

4、测试

1、运行Eureka-server，provider-ticket【8002执行】(端口改为8001打成jar包，执行)，consumer-user

 06.EurekaServer

2、provider-ticket

 07.provider-ticket

 07.provider-ticket

3、consumer-user

 08.consumer

访问是以负载均衡的方式，所以每次都是 8001 。8002.轮询访问

十五、SpringBoot的监管