Spring Boot

本笔记为笔者学习 Spring Boot 时写的笔记，适合没有接触过此框架的新手。其中的代码都能运行，笔者都一一做了验证。本笔记还在更新中，最新版可以关注微信公众号：Java后端，关注后后台回复 「笔记」获取最新版和**此笔记配套源码**。如果你想加入到笔记更新的队伍中来，欢迎联系我。此笔记会不断更新扩充，后台回复「微信」获取小编的联系方式。

同时，扫描下方二维码回复「技术博文」可以获取更多图文教程，还有每月的技术书籍赠送活动、福利多多~

如果学习过程中有任何疑问或者发现错误，请通过上方二维码联系我。本教程是简单笔记记录，从零到一学习 Spring Boot。本教程涉及的知识如下：

- Hell World
    - 注解的使用
    - 依赖讲解
- 配置文件
    - YAML
    - 配置文件讲解
    - Profile模式
    - 配置文件优先级

    - 外部配置加载顺序
    - 自动配置原理
- 日志处理
    - 日志框架介绍
    - SLF4j的使用
    - 日志关系
    - 日志的使用
- Web实战
    - 创建项目
    - 静态资源映射规则
    - 模板引擎
    - Thymeleaf使用
    - Thymeleaf语法
    - Spring MVC自动配置
    - 扩展 Spring MVC
    - 错误处理机制
    - 定制错误页面
- Servlet容器
    - 三大组件介绍
    - 修改容器
- Docker
- 数据访问
- ...

Spring Boot 学习记录

一、Hello World

1. 起步

1. 点击此链接点击Quick start，选择对应版本，下载Demo。
2. IDEA导入项目
3. 建立controller包，以及类

```
package com.wrq.boot.controller;

@Controller
public class HelloController {

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

4. 在于controller包同级有一个主程序类

```
package com.wrq.boot;

@SpringBootApplication
public class BootApplication {

    public static void main(String[] args) {

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

}
```

- @SpringBootApplication 来标注一个主程序，说明这是一个Sping Boot项目

5. 运行这个main方法，控制台打印启动成功。

```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::  (v1.5.19.BUILD-SNAPSHOT)
```

```
2019-01-14 00:35:00.020  INFO 18196 --- [           main]
com.wrq.boot.BootApplication             : Starting BootApplication
on DESKTOP-IJ41H0K with PID 18196 (D:\Java\Project\Spring-Boot-
Notes\boot\target\classes started by wangqian in
D:\Java\Project\Spring-Boot-Notes\boot)
2019-01-14 00:35:00.026  INFO 18196 --- [           main]
com.wrq.boot.BootApplication             : No active profile set,
falling back to default profiles: default
2019-01-14 00:35:00.203  INFO 18196 --- [           main]
ationConfigEmbeddedWebApplicationContext : Refreshing
org.springframework.boot.context.embedded.AnnotationConfigEmbeddedWeb
ApplicationContext@e50a6f6: startup date [Mon Jan 14 00:35:00 CST
2019]; root of context hierarchy
2019-01-14 00:35:04.984  INFO 18196 --- [           main]
s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat initialized with
port(s): 8081 (http)
2019-01-14 00:35:05.048  INFO 18196 --- [           main]
o.apache.catalina.core.StandardService   : Starting service [Tomcat]
2019-01-14 00:35:05.048  INFO 18196 --- [           main]
org.apache.catalina.core.StandardEngine  : Starting Servlet Engine:
Apache Tomcat/8.5.37
2019-01-14 00:35:05.396  INFO 18196 --- [ost-startStop-1] o.a.c.c.C.
[Tomcat].[localhost].[/]       : Initializing Spring embedded
WebApplicationContext
2019-01-14 00:35:05.396  INFO 18196 --- [ost-startStop-1]
o.s.web.context.ContextLoader            : Root
WebApplicationContext: initialization completed in 5205 ms
2019-01-14 00:35:05.880  INFO 18196 --- [ost-startStop-1]
o.s.b.w.servlet.ServletRegistrationBean  : Mapping servlet:
'dispatcherServlet' to [/]
2019-01-14 00:35:05.890  INFO 18196 --- [ost-startStop-1]
o.s.b.w.servlet.FilterRegistrationBean   : Mapping filter:
'characterEncodingFilter' to: [/*]
2019-01-14 00:35:05.891  INFO 18196 --- [ost-startStop-1]
o.s.b.w.servlet.FilterRegistrationBean   : Mapping filter:
'hiddenHttpMethodFilter' to: [/*]
2019-01-14 00:35:05.891  INFO 18196 --- [ost-startStop-1]
o.s.b.w.servlet.FilterRegistrationBean   : Mapping filter:
'httpPutFormContentFilter' to: [/*]
```

```
2019-01-14 00:35:05.892  INFO 18196 --- [ost-startStop-1]
o.s.b.w.servlet.FilterRegistrationBean   : Mapping filter:
'requestContextFilter' to: [/*]
2019-01-14 00:35:06.666  INFO 18196 --- [           main]
s.w.s.m.m.a.RequestMappingHandlerAdapter : Looking for
@ControllerAdvice:
org.springframework.boot.context.embedded.AnnotationConfigEmbeddedWeb
ApplicationContext@e50a6f6: startup date [Mon Jan 14 00:35:00 CST
2019]; root of context hierarchy
2019-01-14 00:35:06.854  INFO 18196 --- [           main]
s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "{[/hello]}" onto
public java.lang.String
com.wrq.boot.controller.HelloController.hello()
2019-01-14 00:35:06.863  INFO 18196 --- [           main]
s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "{[/error]}" onto
public
org.springframework.http.ResponseEntity<java.util.Map<java.lang.Strin
g, java.lang.Object>>
org.springframework.boot.autoconfigure.web.BasicErrorController.error
(javax.servlet.http.HttpServletRequest)
2019-01-14 00:35:06.865  INFO 18196 --- [           main]
s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "
{[/error],produces=[text/html]}" onto public
org.springframework.web.servlet.ModelAndView
org.springframework.boot.autoconfigure.web.BasicErrorController.error
Html(javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpSer
vletResponse)
2019-01-14 00:35:06.946  INFO 18196 --- [           main]
o.s.w.s.handler.SimpleUrlHandlerMapping  : Mapped URL path
[/webjars/**] onto handler of type [class
org.springframework.web.servlet.resource.ResourceHttpRequestHandler]
2019-01-14 00:35:06.946  INFO 18196 --- [           main]
o.s.w.s.handler.SimpleUrlHandlerMapping  : Mapped URL path [/**] onto
handler of type [class
org.springframework.web.servlet.resource.ResourceHttpRequestHandler]
2019-01-14 00:35:07.009  INFO 18196 --- [           main]
o.s.w.s.handler.SimpleUrlHandlerMapping  : Mapped URL path
[/**/favicon.ico] onto handler of type [class
org.springframework.web.servlet.resource.ResourceHttpRequestHandler]
```

```
2019-01-14 00:35:07.278  INFO 18196 --- [           main]
o.s.j.e.a.AnnotationMBeanExporter        : Registering beans for JMX
exposure on startup
2019-01-14 00:35:07.347  INFO 18196 --- [           main]
s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s):
8080(http)
2019-01-14 00:35:07.357  INFO 18196 --- [           main]
com.wrq.boot.BootApplication             : Started BootApplication in
8.343 seconds (JVM running for 9.312)
```

启动成功：访问 http://localhost:8081/hello 即可打印 hello world!

上方日志显示服务器端口等信息，默认是8080，可以在application.properties配置文件中修改默认端口号：

```
#修改端口
server.port=8081
```

可以修改HelloController:

```
@RestController
public class HelloController {


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

上方Controller等于下方的：

```
@ResponseBody
@Controller
public class HelloController {


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

2. 注解区别?

@Controller和@RestController的区别

@RestController注解相当于@ResponseBody + @Controller合在一起的作用。

1. 如果只是使用@RestController注解Controller，则Controller中的方法无法返回jsp页面，或者html，配置的视图解析器 InternalResourceViewResolver不起作用，返回的内容就是Return 里的内容。
2. 如果需要返回到指定页面，则需要用 @Controller配合视图解析器InternalResourceViewResolver才行。 如果需要返回JSON，XML或自定义mediaType内容到页面，则需要在对应的方法上加上@ResponseBody注解。

例如：

1.使用@Controller 注解

- 在对应的方法上，视图解析器可以解析return 的jsp,html页面，并且跳转到相应页面
- 若返回json等内容到页面，则需要加@ResponseBody注解

2.@RestController注解

- 相当于@Controller+@ResponseBody两个注解的结合。
- 返回json数据不需要在方法前面加@ResponseBody注解了
- 使用@RestController这个注解，就不能返回jsp,html页面，视图解析器无法解析jsp,html页面

3. 依赖讲解

1. 父项目（版本仲裁）

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.5.19.BUILD-SNAPSHOT</version>
    <relativePath/> <!-- lookup parent from repository -->
</parent>
```

- 你可以 spring-boot-starter-parent 点进去，就会发现它还有父项目。
- 它可以看成一个版本的仲裁中心，我们所配置的依赖不需要说明版本，因为仲裁中心已经说明了
- 如果中心没有声明的，必须说明版本

2. 基础依赖

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

- spring-boot-starter：spring boot场景启动器
- Spring把每个场景都抽取出来，做成了一个个starts
- 如我们需要那个场景只需要把对应的启动器来导入进来就可以，不用担心版本。
- 用什么功能(Web、缓存、kafka等等)导入相关启动器即可

3. 、Web模块相关依赖

```
<!--添加这个依赖，@ResponseBody @RequestMapping-->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

4. 配置文件处理器

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

5. 打包插件

```
<!-- 将应用打包成一个可执行Jar包，直接使用java -jar xxxx的命令来执行  -->
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>
```

二、配置文件

GitHub对应项目：boot-config

1. 配置文件

Spring Boot使用一个全局的配置文件

- application.properties
- application.yml

配置文件的作用：修改Spring Boot自动配置的默认值，SpringBoot在底层都给我们自动配置好。有什么配置项，可以移步官方文档

配置文件一般放在src/main/resources目录或者类路径/config下，当然还有很多位置可以放，它们会有不同优先级，后面会讲到。

YAML （YAML Ain't Markup Language）

简单介绍

- 以前的配置文件：大多是xml
- .yml是YAML语言的文件，以数据为中 心，比json、xml等更适合做配置文件
- 全局配置文件的可以对一些默认配置值进行修改

配置实例

xml：

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

yml:

```
server:
  port: 8081
```

2. YAML语法

基本语法

- K:(空格)V 标识一对键值对
- 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
- 属性和值也是大小写敏感

实例：

```
server:
  port: 8081
  path: /hello // 冒号后面的空格不要拉下
```

值的写法

普通的值

k: v 字面量直接来写，字符串默认不用添加单引号

- " " 双引号 不会转义字符串里面的特殊字符；

```
name: "wang \n qian" // 输出：wang 换行 qian
```

- ''单引号 会转义字符，特殊字符最终是一个普通的字符串

对象

普通写法：

```
frends:
    lastName: zhang
    age: 20
```

行内写法

```
frends:{ lastName: zhang,age: 18 }
```

Map

示例：

```
maps: {k1: v1,k2: v2}
```

数组

普通写法：

```
pets: // var onj = {pets: ['cat','pig','dog']}
 - cat
 - pig
 - dog
```

行内写法

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

配置文件获取

将配置文件中的每一个值映射到此组件中

1. Persion

```
package com.wrq.boot.bean;

@Component
@ConfigurationProperties(prefix = "persion")
public class Persion {

    private String name;
    private int age;
    private double weight;
    private boolean boss;
    private Date birth;

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

    此处，这个bean的getter、setter和tostring方法已经省略，千万不能忽略！

}
```

- @ConfigurationProperties 意思是:我们类里面的属性和配置文件中的属性做绑定
    - 不使用此注解，可以在bean的属性添加@value()注解，如下：

```
@Component
// @ConfigurationProperties(prefix = "persion")
public class Persion {

    @value("${persion.name}") // $()读取配置文件、环境变量中的值
    private String name;
    @value("#{11*2}") // #{SpEL} 采用表达式
    private int age;
    @value("true") // 直接赋值
    private boolean boos;
}
```

此处采用@ConfigurationProperties的方式，@value()和@ConfigurationProperties的区别见下方表格。

- prefix = "persion" 配置文件中那个下面的属性来一一映射
- @Component 如果想要这个注解起作用，必须放到容器里面

2. Dog

```
package com.wrq.boot.bean;

public class Dog { // 用作Persion中的属性

    private String name;
    private int age;

    此处，这个bean的getter、setter和tostring方法已经省略，千万不能忽略!
}
```

3. 配置文件

- 方式一: application.yml

```
persion:
    name: 王大锤
    age: 18
    weight: 125
    boss: false
    birth: 2018/5/5
    maps: {k1: v1,k2: v2}
    list:
      - wangli
      - wangbai
    dog:
      name: xiaogou
      age: 2
```

- 方式二: application.properties

```
persion.name = 王大锤
persion.age = 18
persion.weight = 125
persion.boss = false
persion.birth = 2018/5/5
persion.maps.k1 = v1
persion.maps.k2 = v2
persion.dog.name = xiaogou
persion.dog.age = 15
```

4. 测试类：BootApplicationTests

```
package com.wrq.boot;

@RunWith(SpringRunner.class)
@SpringBootTest
public class BootApplicationTests {

    @Autowired
    Persion persion;

    @Test
    public void contextLoads() {
        System.out.print(persion);
    }

}
```

5. 运行 BootApplicationTests方法

```
控制台打印:

application.yml的结果:
Persion{name='王大锤', age=18, weight=125.0, boss=false, birth=Sat May
05 00:00:00 CST 2018, maps={k1=v1, k2=v2}, list=[wangli, wangbai],
dog=Dog{name='xiaogou', age=2}}

application.properties的结果:
Persion{name='ï¿½ï¿½Ç¬', age=18, weight=125.0, boss=false, birth=Sat
May 05 00:00:00 CST 2018, maps={k2=v2, k1=v1}, list=[wangli, wangbai],
dog=Dog{name='xiaogou', age=15}}
```

把Bean中的属性和配置文件绑定，通过yml文件和properties都可以做到,但是properties文件出现乱码。

properties中文读取乱码：File->Settings->File Encodings最底部选utf-8、Tranparent打上勾

注解比较

@value和@ConfigurationProperties获取值比较

header 1	@ConfigurationProperties	@value
功能	批量注入配置文件中的属性	一个个指定
松散绑定	支持	不支持
JSR303数据校验	支持	不支持
SpEL	支持	不支持
复杂类型封装	支持	不支持

名词解释:

- 松散绑定

last-name和lastName都可以获取导致，则代表支持松散绑定

- JSR303

```
@Component
@ConfigurationProperties(prefix = "persion") // 如果使用的是@value注入值时，无法使用校验
@Validated // 添加此注解
public class Persion {

    @Email // 配置文件书写的属性必须是邮箱格式，不符合报错!
    private String name;

}
```

- 复杂类型封装

如果获取配置文件中map的值时,@value是获取不到值的

```
@value("${persion.maps}") // 由于使用的是@value，无法获取配置文件中的map
private Map<String,Object> maps;
```

@PropertySource

@PropertySource：加载指定配置文件

@ConfigurationProperties()默认是从全局配置文件中获取值，也就是application.properties这个文件中获取值。

如果做的配置很多，全局的配置文件就会特别大，为了方便管理。我会创建不同的配置文件定向管理不同的配置。

如创建persion.properties文件单独存放persion需要的配置

@PropertySource就是用来导入创建的配置文件

示例：

1. persion.properties

同时把两个全局的配置中关于Persion的配置都注释掉

```
persion.name = 王弟弟
persion.age = 18
persion.weight = 125
persion.boss = false
persion.birth = 2018/5/5
persion.maps.k1 = v1
persion.maps.k2 = v2
persion.dog.name = xiaogou
persion.dog.age = 15
```

2. Persion

```
package com.wrq.boot.bean;

@Component
@PropertySource(value = {"classpath:persion.properties"})
@ConfigurationProperties(prefix = "persion")
public class Persion {

    private String name;
    private int age;
    private double weight;
    private boolean boss;
    private Date birth;

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

    此处，这个bean的getter、setter和tostring方法已经省略，千万不能忽略!
}
```

这样运行测试类，控制台就可以打印persion.properties中的数据。

通过下面的注解，把类路径下的persion.properties加载进来。并且把persion开头的数据进行绑定。

- @PropertySource(value = {"classpath:persion.properties"})

- @ConfigurationProperties(prefix = "persion")

@ImportResource

@ImportResource：导入Spring的配置文件，让配置文件生效。

示例：

1. com.wrq.boot.service

```
package com.wrq.boot.service;

/**
 * Created by wangqian on 2019/1/12.
 */
public class HelloService {
}
```

2. resources目录手动建立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.wrq.boot.service.HelloService">
</bean>
</beans>
```

3. 测试类

```
package com.wrq.boot;

@RunWith(SpringRunner.class)
@SpringBootTest
public class BootApplicationTests {

    @Autowired
    ApplicationContext ioc;
```

```
    @Test
    public void testConfig() {
        boolean b = ioc.containsBean("helloService");
        System.out.print(b);
    }

}
```

试图通过添加一个Spring的配置文件bean.xml来把HelloService注入进去。

运行测试类结果：false

结果表明IoC容器中并不包含HelloService，即：配置文件bean.xml没有生效

解决方式

方式一： 主程序中进行配置@ImportResouece注解

```
package com.wrq.boot;

@ImportResource(locations = {"classpath:bean.xml"}) // 通过此配置是
bean.xml生效
@SpringBootApplication
public class BootApplication {

    public static void main(String[] args) {

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

}
```

方法二：通过配置类实现，这种方式也是Spring Boot推荐的

1. com.wrq.boot.config

```
package com.wrq.boot.config;

/**
 * Created by wangqian on 2019/1/12.
 */
@Configuration
public class MyConfig {

    // 将方法的返回值添加到容器之中,并且容器中这个组件的id就是方法名
    @Bean
    public HelloService helloService(){
        System.out.print("通过@Bean给容器添加组件了..");
        return new HelloService();
    }
}
```

- @Configuration标注这是一个配置类
- 通过@Bean注解，将方法的返回值添加到容器之中,并且容器中这个组件的id就是方法名

2. 把主程序类中@ImportResource()配置注释掉
3. 测试成功，添加了HelloService()组件

3. 配置文件占位符

随机数

RandomValuePropertySource：配置文件中可以使用随机数

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

属性配置占位符

- 可以在配置文件中引用前面配置过的属性（优先级前面配置过的这里都能用）
- ${app.name:默认值}来指定找不到属性时的默认值

```
persion.name = 王弟弟${random.uuid}
persion.age = ${random.int}
persion.dog.name = ${persion.name}_dog
```

4. Profile 多环境支持

Profile是Spring对不同环境提供不同配置功能的支持，可以通过激活、 指定参数等方式快速切换环境

1. 多Profile的方式

格式：application-{profile}.properties/yml

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

默认采用application.properties配置文件，如果使用别的，需要激活：

1. application.properties中配置：

```
# 激活application-dev.properties配置文件
spring.profiles.active=dev
```

2. application-dev.properties:

```
server.port=8082
```

3. 运行BootApplication主程序:

```
2019-01-12 20:46:09.345  INFO 14404 --- [main]
s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s):
8082 (http)
```

2. 多文档块的方式

除了上方多Profile的方式来切换环境，也可以通过YAML多文档块的方式。

示例:

application.yml:

```
server:
  port: 8081
spring:
  profiles:
    active: dev
---
spring:
  profiles: dev
server:
  port: 8083
---
spring:
  profiles: prod
server:
  port: 8084
```

3. 激活指定Profile

1. application.properties中配置:

```
# 激活application-dev.properties配置文件
spring.profiles.active=dev
```

2. application.yml中配置

```
server:
  port: 8081
spring:
  profiles:
    active: dev
---
spring:
  profiles: dev
server:
  port: 8083
```

3. 启动配置-参数

在IDE中，类似于配置tomcat的地方，按下方配置：

```
Program arguments:--spring.profiles.active=dev
```

4. 启动配置-虚拟机

在IDE中，类似于配置tomcat的地方，按下方配置：

```
VM options:-Dspring-profiles-active=dev
```

5. 命令行 使用Maven的package命令打包，移动到jar的目录。

```
java -jar spring-boot-project-config.jar --spring.profiles.active=dev
```

5. 配置文件优先级

GitHub对应项目：boot-config-position

优先级

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

- file:./config/ (项目根目录config文件夹下的配置文件)
- file:./ (项目根目下的配置文件)
- classpath:/config/ (resources目录config文件夹下的配置文件)
- classpath:/ (resources目下的配置文件)

以上是按照优先级从高到低的顺序，**所有位置**的文件都会被加载，高优先级配置内容会覆盖低优先级配置内容,形成**互补配置**。

默认配置

我们也可以通过配置spring.config.location来改变默认配置。

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

1. Maven->package对项目打包

2. 把待使用的配置文件放在本地文件夹中，如：D:/application.properties
3. 命令行执行命令

```
java -jar boot-config-position-xxxxxx.jar --
spring.config.location=D:/application.properties
```

这样即使项目上线了，我们也可以通过修改本地的配置文件，使用一行命令即可，极大方便了运维人员。

6. 外部配置加载顺序

Spring Boot 支持多种外部配置方式

可以从以下位置加载配置，优先级从高到低，高优先级配置覆盖低优先级的，所以配置形成互补配置。

1. 命令行参数

```
java -jar boot-config-position-xxxxxx.jar --server.port // 多个配置用空格隔开
```

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
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(不带spring.profile)配置文件
9. jar包内部的application.properties或application.yml(不带spring.profile)配置文件
10. @Configuration注解类上的@PropertySource
11. 通过SpringApplication.setDefaultProperties指定的默认属性

注意：从jar包外向jar包内寻找，优先加载profile最后加载不带profile，更多参考官方文档

7. 自动配置原理

GitHub对应项目：boot-config-autoconfig

1. 配置文件写什么?

- 配置文件可配置属性查阅

2. 什么是注解，如何实现一个注解?

- 关于注解的机制和相关原理可以移步此篇博客

3. 配置原理解析

我们运行Spring Boot应用是从main方法启动，在主程序类上有一个@SpringBootApplication注解。

@SpringBootApplication是一个复合注解，包括@ComponentScan，和@SpringBootConfiguration，@EnableAutoConfiguration。

- @SpringBootConfiguration继承自@Configuration，二者功能也一致，标注当前类是配置类，并会将当前类内声明的一个或多个以@Bean注解标记的方法的实例纳入到srping容器中，并且实例名就是方法名。
- @EnableAutoConfiguration的作用启动自动的配置，@EnableAutoConfiguration注解的意思就是SpringBoot根据你添加的jar包来配置你项目的默认配置，比如根据spring-boot-starter-web ，来判断你的项目是否需要添加了webmvc和tomcat，就会自动的帮你配置web项目中所需要的默认配置
- @ComponentScan，扫描当前包及其子包下被@Component，@Controller，@Service，@Repository注解标记的类并纳入到spring容器中进行管理。是以前的context:component-scan（以前使用在xml中使用的标签，用来扫描包配置的平行支持）。

@SpringBootApplication注解分析

配置原理视频讲解

4. 自动配置类判断

在配置文件properties中设置：debug=true 来让控制台打印自动配置报告，方便的得知那些配置类生效。

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

```
Positive matches:
-----------------

   DispatcherServletAutoConfiguration matched:
      - @ConditionalOnClass found required class
'org.springframework.web.servlet.DispatcherServlet';
@ConditionalOnMissingClass did not find unwanted class
(OnClassCondition)
      - @ConditionalOnWebApplication (required) found 'session' scope
(OnWebApplicationCondition)
Negative matches:
-----------------

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

三、日志处理

GitHub对应项目：boot-logging

1. 日志框架

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

分类

日志门面	日志实现
JCL SLF4J Jboos-logging	Log4j JUL Log4j2 Logback

左边选一个门面（抽象层）、右边选一个实现。

日志门面：SLF4J

日志实现：Logback

Spring Boot底层是Spring框架，Spring框架默认是使用的 Commons Logging，而Spring Boot选用的是SLF4J和Logback.

spring-boot-starter-logging采用了 slf4j+logback的形式，Spring Boot也能自动适配（jul、log4j2、logback） 并 简化配置

2. SLF4j的使用

框架使用

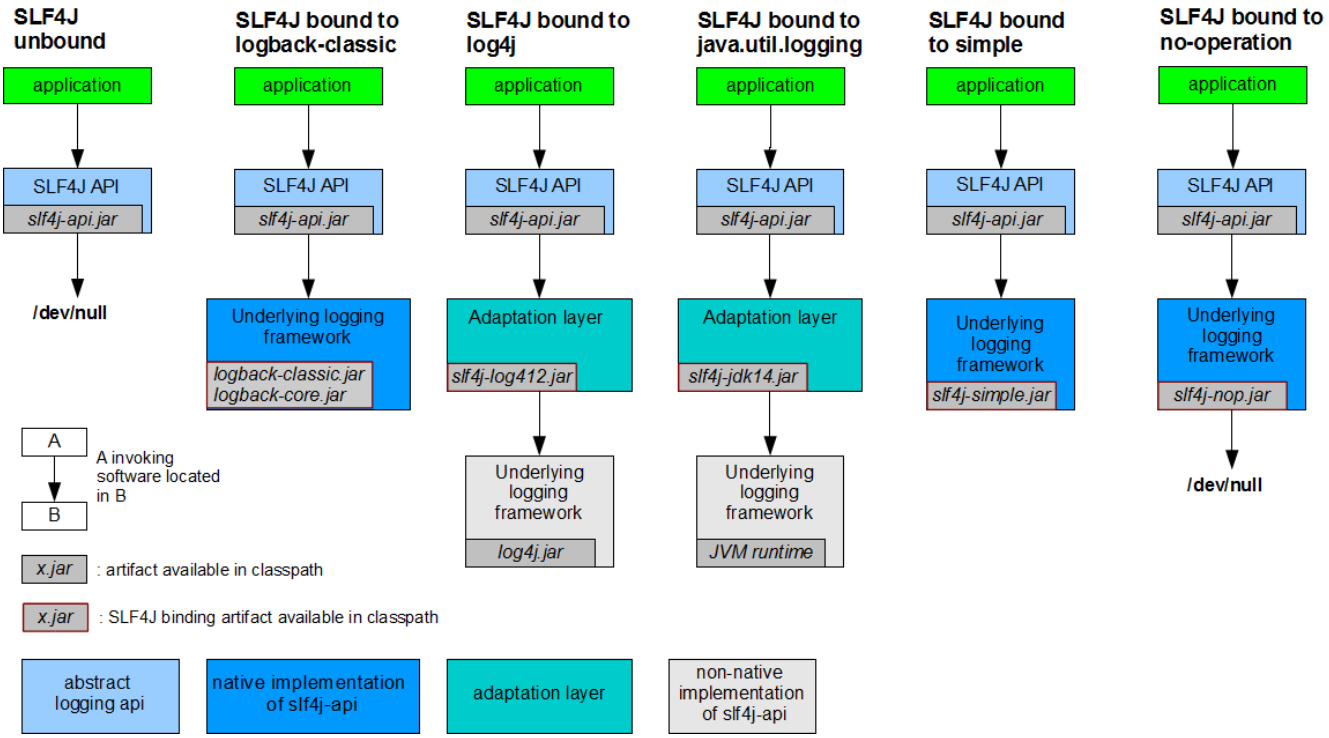
1. 导入SLF4j和Logback的Jar(Spring Boot的基础依赖已经包含)
2. 安装下面方式调用

```
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");
  }
}
```

我们调用方法的时候需要使用日志来记录方法的调用，但是我们不应该调用实现了logback的接口，应该是日志门面（抽象层）SLF4j的接口。

SLF4j是一个抽象层，实现层选用什么都可以，我们此处选用Logback。我们应该面向SLF4j的接口编程，调用SLF4j的接口来做日志记录，但是最后实现日志功能的是Logback。具体SLF4j和其他日志实现层的关系查看此图片：



当然log4j的实现和SLF4j一起使用时候，中间会有一层Adaptation layer适配层，是因为log4j出现的比较早，而SLF4j比较晚。为了和log4j适配，SLF4j开发提供的Adaptation layer层。

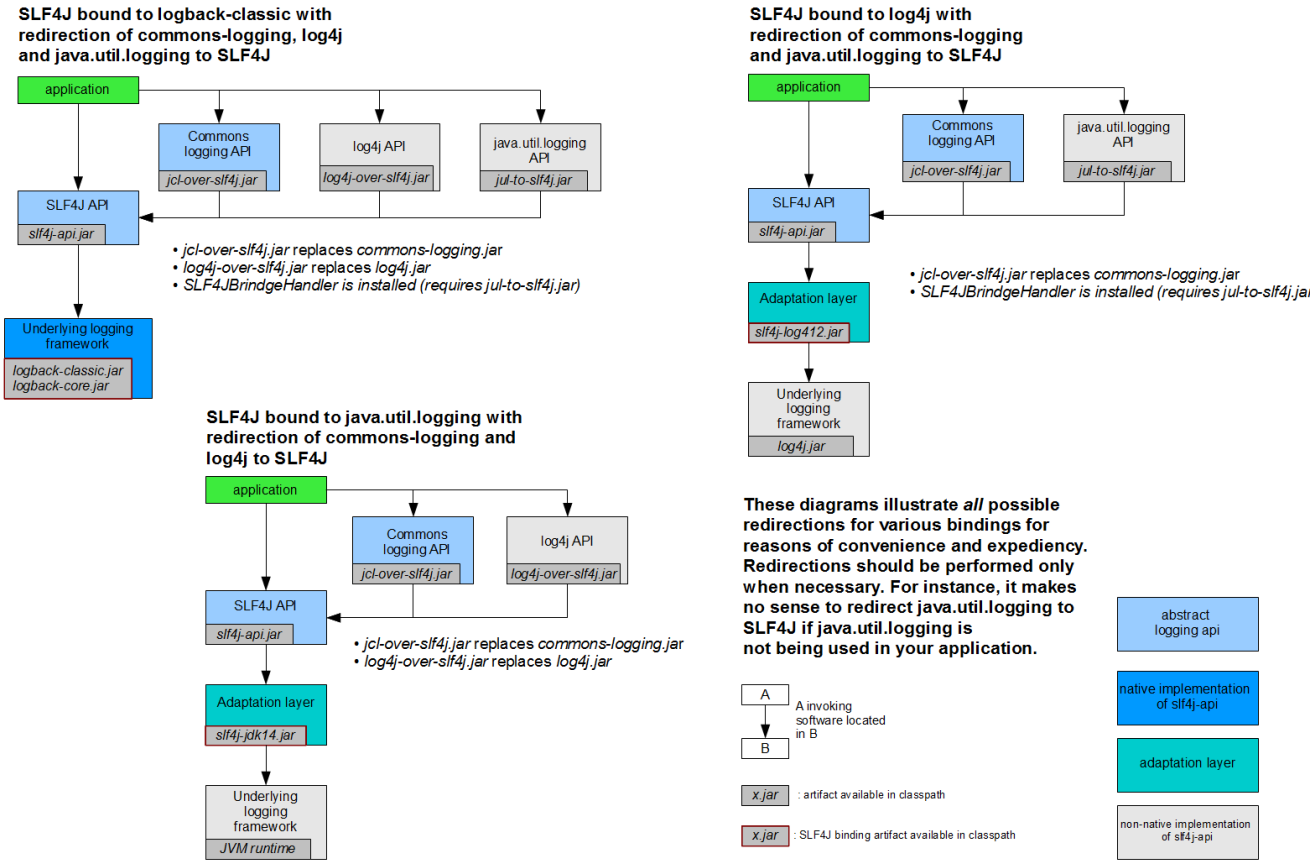
每一个日志的实现框架都有自己的配置文件，使用了slf4j以后，配置文件还是做成日志**实现框架**的配置文件，slf4j只是抽象层，你使用什么框架实现的就写哪个实现框架的配置文件。

遗留问题

如果我们在开发A系统的时候使用SLF4j和Logback框架来做日志处理，同时此系统使用了Spring、Hibernate、MyBatis等框架，Sping框架是使用commons-logging做日志处理，Hibernate使用Jboss-logging做日志处理。

如何统一做日志记录，即使是别的框架和我一起统一使用SLF4j进行输出？

查看此图：



注意：如果你使用Commons logging的接口，可以使用jcl-over-slf4j.jar替换commons-logging.jar实现日志的统一处理。

理解了图片，如何让系统中所有日志统一到SLF4j?

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以及相应实现。

3. Sping Boot日志关系

查看依赖关系: IDEA->pom.xml->右键->Diagrams->show Dependencies

基本依赖:

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

Spring Boot使用下方依赖做日志功能

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

我们观察日志依赖的关系图：

```
graph LR
spring-boot-starter-logging-->logback-classic
logback-classic-->logback-core
spring-boot-starter-logging-->jul-to-slf4j
jul-to-slf4j-->slf4j-api
spring-boot-starter-logging-->log4j-over-slf4j
log4j-over-slf4j-->slf4j-api
spring-boot-starter-logging-->jcl-over-slf4j
jcl-over-slf4j-->slf4j-api
```

其中logback是日志的实现，jul-to-slf4j、log4j-over-slf4j、jcl-over-slf4j的作用是将其他日志框架转为slf4j的。最后他们都指向了slf4j的抽象层。

总结：

1. Spring Boot选用的是SLF4J和Logback进行日志记录
2. Spring Boot也把其他的的日志都替换成了slf4j。可以查看替换包的源码，偷梁换柱！
3. 如果我们要引入其他框架，一定要把这个框架的日志包移除掉，如下Spring Boot框架依赖中就把Sping框架依赖的commons-logging依赖排除掉了。

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

4. 日志使用

默认配置

其实Spring Boot默认帮我们配置了日志，比如启动主程序时打印的日志。

```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::  (v1.5.19.BUILD-SNAPSHOT)

2019-01-13 22:50:22.392  INFO 2364 --- [           main]
com.wrq.boot.BootApplication             : Starting BootApplication on
DESKTOP-IJ41HOK with PID 2364 (D:\Java\Project\Spring-Boot-Notes\boot-
logging\target\classes started by wangqian in D:\Java\Project\Spring-
Boot-Notes\boot-logging)
2019-01-13 22:50:22.393  INFO 2364 --- [           main]
com.wrq.boot.BootApplication             : No active profile set,
falling back to default profiles: default
2019-01-13 22:50:22.394 DEBUG 2364 --- [           main]
o.s.boot.SpringApplication               : Loading source class
com.wrq.boot.BootApplication
```

使用与级别

测试类中使用日志：

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

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

    @Test
    public void contextLoads() {

        logger.trace("trace追踪信息");
```

```
        logger.debug("debug调试信息");
        logger.info("info日志信息");
        logger.warn("warn警告信息");
        logger.error("error报错信息");

    }

}
```

日志的级别：trace < debug < info < warn < error

Spring Boot默认的日志级别是：info，故运行结果：

```
2019-01-13 23:17:56.731  INFO 2624 --- [           main]
c.wrq.boot.BootLoggingApplicationTests   : info日志信息
2019-01-13 23:17:56.732  WARN 2624 --- [           main]
c.wrq.boot.BootLoggingApplicationTests   : warn警告信息
2019-01-13 23:17:56.745 ERROR 2624 --- [           main]
c.wrq.boot.BootLoggingApplicationTests   : error报错信息
```

可以调整输出日志的级别，项目上线了我们不想要debug、trace的信息，我们可以调高日志级别，只会打印调整后级别及高级别生效。

修改级别

application.properties配置文件:

```
logging.level.com.wrq=trace // 把com.wrq包下面类的日志都设成trace级别
```

运行结果：

```
2019-01-13 23:24:06.449 TRACE 18136 --- [           main]
c.wrq.boot.BootLoggingApplicationTests   : trace追踪信息
2019-01-13 23:24:06.449 DEBUG 18136 --- [           main]
c.wrq.boot.BootLoggingApplicationTests   : debug调试信息
2019-01-13 23:24:06.449  INFO 18136 --- [           main]
c.wrq.boot.BootLoggingApplicationTests   : info日志信息
2019-01-13 23:24:06.749  WARN 18136 --- [           main]
c.wrq.boot.BootLoggingApplicationTests   : warn警告信息
2019-01-13 23:24:06.749 ERROR 18136 --- [           main]
c.wrq.boot.BootLoggingApplicationTests   : error报错信息
```

修改日志输出方式

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

配置文件中：

```
logging.level.com.wrq = trace

# 不指定路径，当前项目下生成springboot.log日志
#logging.file=springboot.log

# 当D盘下生成springboot.log日志
#logging.file=D:/springboot.log

# 在当前磁盘的根路径下创建spring文件夹和里面的log文件夹，使用spring.log作为默认的日志文件名
logging.path=/spring/log

# 在控制台输出的日志的格式
logging.pattern.console=%d{yyyy-MM-dd HH:mm:ss.SSS} [%thread] %-5level %logger{50} - %msg%n

# 在指定文件中输出的日志的格式
```

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

# 日志输出格式： %d表示日期时间， %thread表示线程名， %-5level：级别从左显示5个字符宽度 %logger{50} 表示logger名字最长50个字符，否则按照句点分割。 %msg：日志消息， %n是换行符
```

指定日志文件

具体方式参考官方文档日志章节

给类路径下放上每个日志框架自己的配置文件即可，加载配置时会判断，如我们使用自己的配置文件，Spring Boot就不会使用它自己的默认配置：

日志框架	命名
Logback	logback-spring.xml, logback-spring.groovy, logback.xml or logback.groovy
Log4j2	log4j2-spring.xml or log4j2.xml
Logback	logging.properties

官方推荐 logback-spring.xml的命名配置文件。

- logback-spring.xml 可以使用springProfile的高级特性

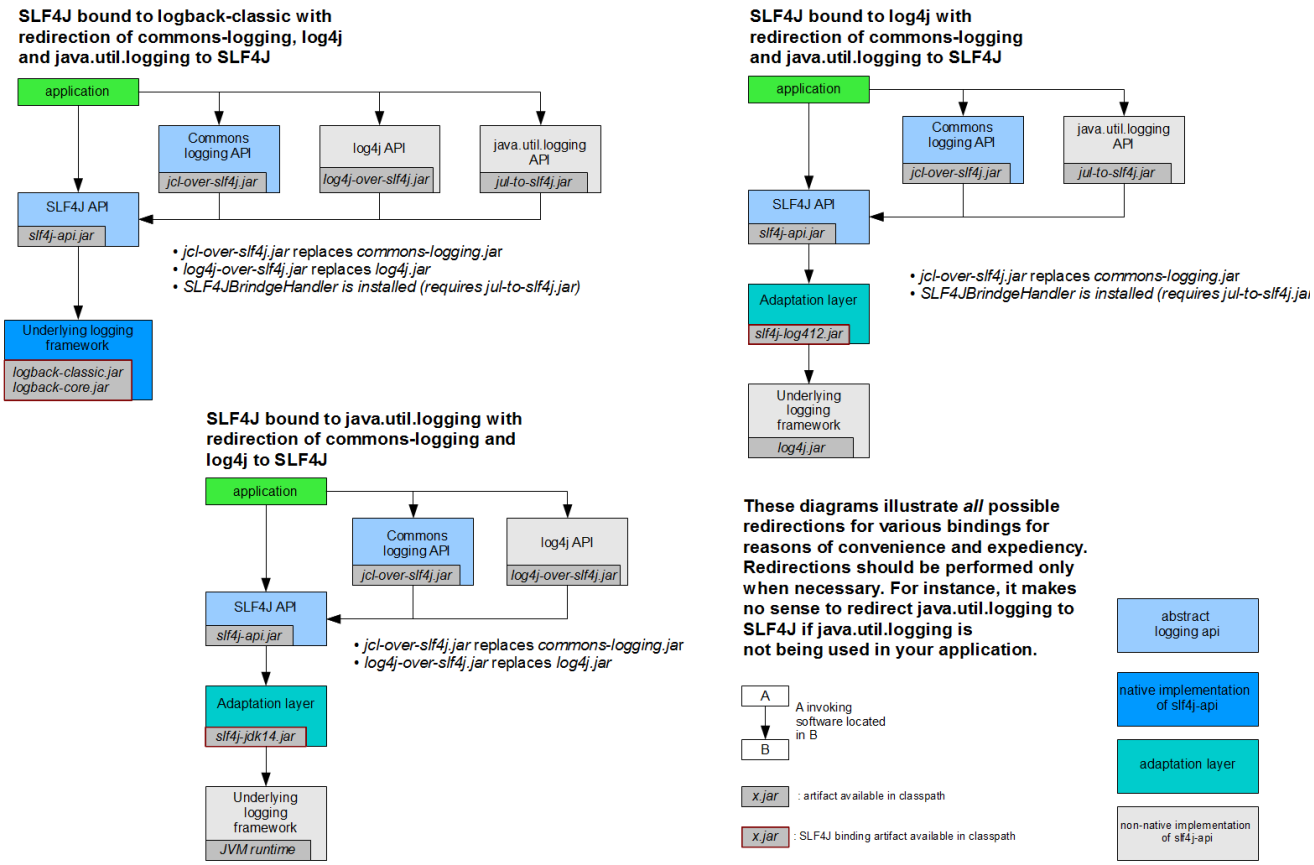
```
<!-- logback-spring.xml配置文件中： -->

<layout class="ch.qos.logback.classic.PatternLayout">
    <springProfile name="dev"> <!-- dev环境下采用下方输出方式 -->
        <pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} --- [%thread] %-5level
%logger{50} - %msg%n</pattern>
    </springProfile>
    <springProfile name="!dev"> <!-- 非dev环境下采用下方输出方式 -->
        <pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} ---- [%thread] %-5level
%logger{50} - %msg%n</pattern>
    </springProfile>
</layout>
```

- logback.xml：直接被识别为日志配置文件

切换日志

根据此图进行移除包和导入包即可，可以通过IDEA的依赖图方便的移除依赖。



四、Web实战

1. 创建项目

步骤

1. 创建Spring Boot项目
2. 导入相关的模块，比如Web模块
3. 编写自己的业务逻辑

思考

当我们每使用一个场景的时候需要思考Sping Boot帮我们配置了什么，我们可以不可修改，能修改哪些配置？

在org.springframework.boot.autoconfig的Jar里面都是自动配置的组件。

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. 静态资源映射规则

1) 第一种规则

由于我们把Web项目最后会打成Jar包，发布线上。引入Bootstrap，jQuery等静态资源文件就不能放在Webapp文件夹下(也没有Webapp文件夹)，我们必须通过把静态资源打成Jar包，添加至pom.xml，依赖查找移步WebJars官网，WebJars：使用Jar包的方式引入静态资源。

查看自动配置的Jar，Web模块中的WebMvcAutoConfiguration类的源码：

```
public void addResourceHandlers(ResourceHandlerRegistry registry) {
    if(!this.resourceProperties.isAddMappings()) {
        logger.debug("Default resource handling disabled");
    } else {
        Integer cachePeriod = this.resourceProperties.getCachePeriod();
        if(!registry.hasMappingForPattern("/webjars/**")) {

this.customizeResourceHandlerRegistration(registry.addResourceHandler(n
ew String[]{"/webjars/**"}).addResourceLocations(new String[]
{"classpath:/META-
INF/resources/webjars/"}).setCachePeriod(cachePeriod));
        }

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

this.customizeResourceHandlerRegistration(registry.addResourceHandler(n
ew String[]
{staticPathPattern}).addResourceLocations(this.resourceProperties.getSt
aticLocations()).setCachePeriod(cachePeriod));
        }
    }
 }
```

上方部分源码表明：所有的 /webjars/** 映射请求都会去 classpath:/META-INF/resources/webjars/ 路径下去找资源。

如：引入jQuery资源文件

1. 登陆WebJars官网，复制依赖：

```
<dependency>
    <groupId>org.webjars</groupId>
    <artifactId>jquery</artifactId>
    <version>2.1.1</version>
</dependency>
```

2. 导入后，查看org.webjars:jquery的目录文件：

```
org.webjars:jquery:2.1.1
 ->jquery-2.1.1.jar
   ->META-INF
   ->maven
   ->resources
     ->webjars
       ->jquery
         ->2.1.1
           ->jquery.js
           ->jquery.min.js
```

3. 对应上方映射规则访问：localhost:8080/webjars/jquery/2.1.1/jquery.js，访问后显示jquery.js的源码。
4. 故使用webjars的方式只需要引入依赖配置，访问时写webjars下面的资源名称即可。

2) 第二种规则

访问当前项目的任何资源的时候，他回去一些文件夹下寻找：

- classpath: /META-INF/resources/
- classpath: /resources/ (这是resources文件夹下面的resources文件夹)
- classpath: /static/
- classpath: /public/
- / 当前项目根路径

localhost:8080/adc/abc.js ---> 去上面资源文件夹中去找/abc/abc.js

3) 第三种规则

第三种规则是欢迎页的是设置，欢迎页：静态资源文件夹下所有的index.html页面。

localhost:8080/ ---> 去资源文件夹找找index.html

4) 第四种规则

第四种规则是网站图标。

所有的 **/favicon.ico 都是在资源文件夹下找。

5) 自定义静态文件夹路径

Spring Boot默认配置是在类路径下的resources、public、static文件夹为静态资源文件夹。我们通过下方配置修改路径：

```
spring.resources.static.locations=classpath:/hell,classpath:/wrq/
```

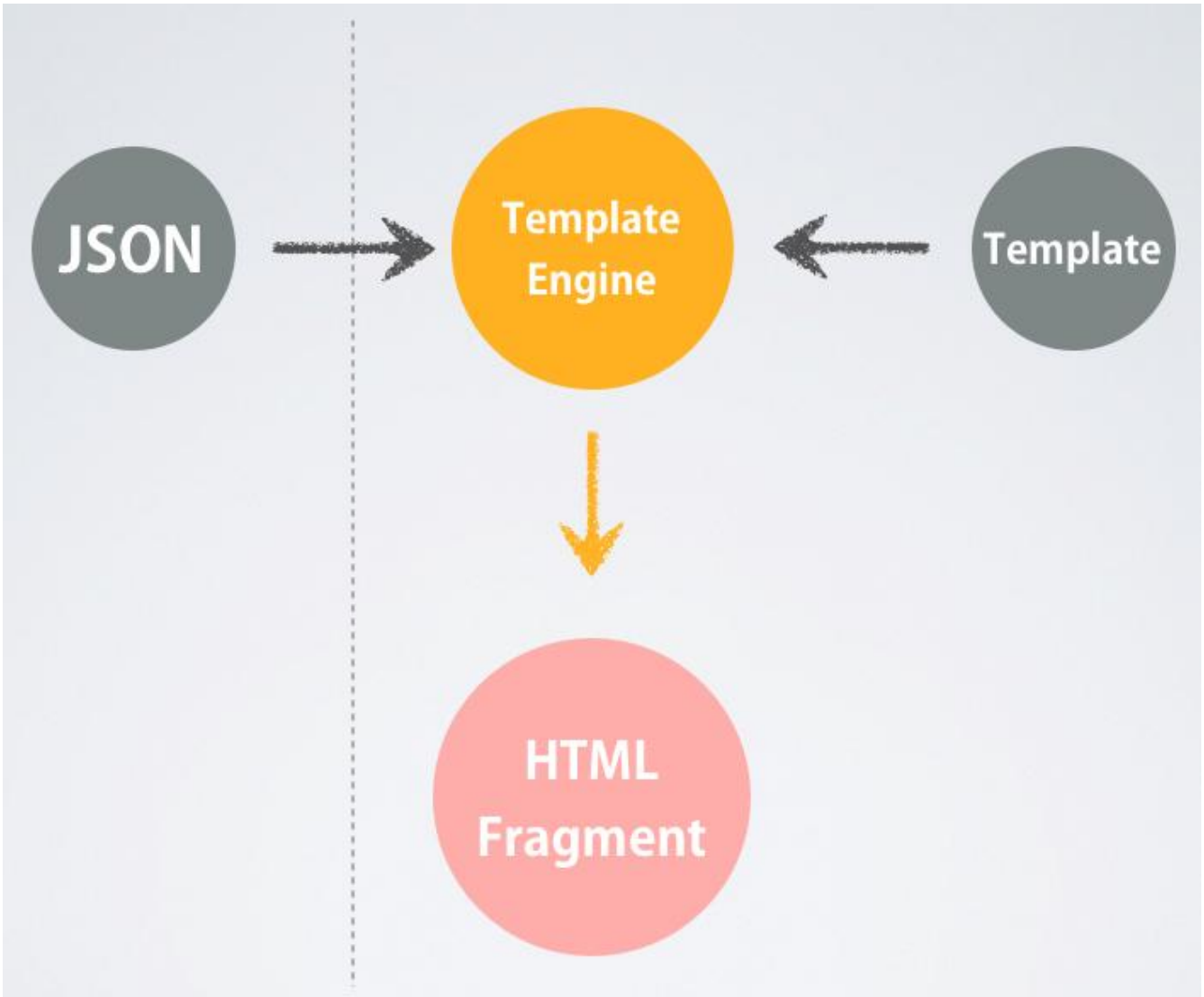
3. 模板引擎

1) 常见的模板引擎

JSP、Velocity、Freemarker、Thymeleaf

Spring Boot推荐使用Thymeleaf，语法简单、强大。

模板引擎的作用：把数据和静态模板进行绑定，生成我们想要的HTML。



2) 使用模板引擎

1. 引入依赖

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

2. 默认版本为Thymeleaf 2，如果想使用Thymeleaf 3，移步官网，添加配置。

```
<properties>
    <thymeleaf.version>3.0.2.RELEASE</thymeleaf.version>
    <thymeleaf-layout-dialect.version>2.1.1</thymeleaf-layout-
dialect.version>
</properties>
```

4. Thymeleaf使用

打开Spring Boot的关于自动配置的Jar，找到Thymeleaf相关的文件夹，有一个类：ThymeleafProperties:

```
@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";

    // 只需要把HTML放在classpath:/templates/，Thymeleaf就会自动解析渲染
```

例如：

```
package com.wrq.boot.controller;

@Controller
public class HelloController {
    // 此处并没有@ResponseBody注解
    @RequestMapping("/success")
    public String hello () {
        return "success";
    }
}
```

访问http://localhost:8080/success 就会打开 resources/templates/success.html文件，这是由thymeleaf完成的。

简单使用：

1. 导入thymeleaf的命名空间，以获得更好的提示。

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

2. 使用thymeleaf

- 编写controller

```
package com.wrq.boot.controller;

@Controller
public class HelloController {
    @RequestMapping("/success")
    public String hello (Map<String ,Object> map) {
        map.put("Hello","World"); // 传递过去一个值
        return "success";
    }
}
```

- success.html

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
<div th:text="${Hello}">模板引擎获取不到值</div>
</body>
</html>
```

- 请求 localhost:8080/success 会打印 World

5. Thymeleaf语法

常用的语法实例-中文版

1) 语法由、片段包含、遍历、条件判断、声明变量等等

- thymeleaf语法官方文档 2) 表达式
- 表达式语法

6. Sping MVC自动配置

Spring Boot 自动配置好了SpringMVC，更详细移步 Web开发相关文档

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

- Inclusion of ContentNegotiatingViewResolver and BeanNameViewResolver beans.
    - 自动配置了ViewResolver（视图解析器：根据方法的返回值得到视图对象（View），视图对象决定如何 渲染））
    - ContentNegotiatingViewResolver：组合所有的视图解析器的
    - 如何定制：我们可以自己给容器中添加一个视图解析器；自动的将其组合进来
- Support for serving static resources, including support for WebJars.静态资源文件夹路 径,webjars
- Static index.html support. 静态首页访问
- Custom Favicon support. 网站图标
- 自动注册了 of Converter , GenericConverter , Formatter beans.
    - Converter：转换器， public String hello(User user)接受前端是文本，转化成Inteager等类型..
    - Converter Formatter 格式化器，2017.12.17===Date
    - 自己添加的格式化器转换器，我们只需要放在容器中即可
- Support for HttpMessageConverters.
    - HttpMessageConverter：SpringMVC用来转换Http请求和响应的,User---Json.
    - HttpMessageConverters 是从容器中确定,获取所有的HttpMessageConverter
    - 自己给容器中添加HttpMessageConverter，只需要将自己的组件注册容器中（@Bean,@Component）
- Automatic registration of MessageCodesResolver.定义错误代码生成规则
- Automatic use of a ConfigurableWebBindingInitializer bean
    - 我们可以配置一个ConfigurableWebBindingInitializer来替换默认的（添加到容器）
    - 初始化数据绑定器

7. 如何修改Sping Boot的默认配置

我们阅读Spring Boot自动配置的源码，如：
org.springframework.boot.autoconfigure.web：web的所有自动场景

我们发现，Spring Boot都会默认注册一些组件供我们使用，但是注入的时候会判断有没有定义这个组件，如果定义则使用你定义的，没有就使用默认的。如下：

```
@Bean
@ConditionalOnMissingBean
public HttpMessageConverters messageConverters() {}
```

1. Spring Boot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），如果没有才自动配置
2. 如果有些组件有多个，比如视图解析器。用户的配置和默认配置组合起来
3. 在SpringBoot中会有非常多的xxxConfigurer帮助我们进行扩展配置
4. 在SpringBoot中会有很多的xxxCustomizer帮助我们进行定制配置

8. 扩展Spring MVC

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.

原本配置Spring MVC：

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

如果你想添加额外的MVC配置，可以添加一个WebMvcConfigurerAdapter类型的@Configuration配置类，并且不可以添加@EnableWebMvc标记。

自定义配置：

```
package com.wrq.boot.config;

@Configuration
public class ConfigController extends WebMvcConfigurerAdapter {
```

```
    /**
     * 访问 http://localhost:8080/helloMan 跳转到success.html
     * @param registry
     */
    @Override
    public void addViewControllers(ViewControllerRegistry registry) {
        registry.addViewController("/helloMan").setViewName("success");
    }
}
```

原理：

1. WebMvcAutoConfiguration是SpringMVC的自动配置类
2. 在做其他自动配置时会导入；@Import(EnableWebMvcConfiguration.class)
3. 容器中所有的WebMvcConfigurer都会一起起作用
4. 我们的配置类也会被调用

效果：SpringMVC的自动配置和我们的扩展配置都会起作用

9. 完全掌控Spring MVC

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

```
package com.wrq.boot.config;

@EnableWebMvc  // 添加此注解将会全面掌控Spring MVC
@Configuration
public class ConfigController extends WebMvcConfigurerAdapter {

    /**
     * 访问 http://localhost:8080/helloMan 跳转到success.html
     * @param registry
     */
    @Override
    public void addViewControllers(ViewControllerRegistry registry) {
        registry.addViewController("/helloMan").setViewName("success");
    }
}
```

Spring Boot对Spring MVC已经配置好了，我们可以扩展。同样我们可以完全掌控Spring MVC，就如SSM框架开发的时候，我们需要什么功能从头开始配置，Spring Boot的默认配置不会生效。

原理： 为什么添加@EnableWebMvc就使Spring Boot的配置失效?

1. @EnableWebMvc的核心

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

2. WebMvcConfigurationSupport类

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

3. WebMvcAutoConfiguration类

```
@Configuration
@ConditionalOnWebApplication @ConditionalOnClass({ Servlet.class,
DispatcherServlet.class, WebMvcConfigurerAdapter.class})
//容器中没有这个组件的时候，这个自动配置类才生效
@ConditionalOnMissingBean(WebMvcConfigurationSupport.class)
@AutoConfigureOrder(Ordered.HIGHEST_PRECEDENCE+10)
@AutoConfigureAfter({DispatcherServletAutoConfiguration.class,
ValidationAutoConfiguration.class})
public class WebMvcAutoConfiguration {
```

4. @EnableWebMvc将WebMvcConfigurationSupport组件导入进来
5. 导入的WebMvcConfigurationSupport只是SpringMVC基本的功能

10. 增删改查Demo

GitHub项目：boot-web-crud

1. 引入Bootstrap，jQuery的webjars

2. 修改静态资源的路径名，th:href="@{}"

- 使用@{}表达式的好处是：全局配置文件添加server.context-path=/project配置后，他会在添加了表达式字符串之前自动加上/project

3. 国际化的引入

- 以前使用Spring MVC
  - 编写国际化配置文件
  - 使用ResourcesBundleMessageSource管理国际化资源文件
  - 在页面使用fmt:message取出国际化内容
- 现在使用Spring Boot
  - 编写国际化配置文件(login_zn_CN.properties)，抽取页面需要显示的国际化消息
  - SpringBoot自动配置好了管理国际化资源文件的组件
  - 默认采用类路径下message.properties配置文件中的国际化配置
  - 若采用自己写的配置文件的配置，需要做下方配置后，去页面使用#{}获取配置的值即可。

```
// resources->i18n->login_en_US.properties(login_zn_CN.properties)

spring.messages.basename=i18n.login // 设置国际化资源文件的基础名（去掉语言国家代码的）
```

使用Spring Boot做国际化管理的时候，Spring Boot自动配置好了管理国际化资源文件的相关组件，此逐渐在类MessageSourceAuoConfiguration中被添加到容器中的，这个组件默认去类路径下message.properties中取配置项，但是在主配置文件中可以通过spring.messages.basename的方式修改默认的位置。

点击 中文->中文，English->英文功能

原理：

1) 国际化中非常重要的是Locale对象，它是区域信息对象，根据这个对象中的信息来确定是渲染英文还是中文，其中LocaleResolver国际化语言解析器 就是来获得Locale对象的。

2) Sping Boot默认在容器中添加了关于国际化语言解析器的组件

```
@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;
}
```

3) 默认的区域信息解析器：它根据请求头带来的区域信息获取Locale进行国际化。它添加了@ConditionalOnMissingBean组件，只要我们自定义解析器默认就不会生效，使用我们的组件。前提是我们写一个localeResolver，并且把他加入到容器中。

4. 登陆

开发期间模板引擎页面修改以后，需要实时生效： 1) 禁用模板引擎的缓存

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

2) 页面修改完成后Ctrl + F9,重新编译。 3) 拦截登陆请求

自定义拦截器：

```
public class LoginHandlerIntercepter implements HandlerInterceptor {

    @Override
    public boolean preHandle(HttpServletRequest request,
HttpServletResponse response, Object handler) throws Exception {
        Object username =
request.getSession().getAttribute("LoginUser");
        if(username != null){
            return true;
        } else  {
            request.setAttribute("msg", "没有权限，请登录");
```

```
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 {

    }
}
```

注册拦截器:

```
@Configuration
public class ConfigController extends WebMvcConfigurerAdapter {
    @Bean
    public WebMvcConfigurerAdapter webMvcConfigurerAdapter(){
        WebMvcConfigurerAdapter adapter = new WebMvcConfigurerAdapter()
{

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

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

        @Override
```

```
        public void addInterceptors(InterceptorRegistry registry) {
            // 静态资源：Spring Boot已经做好了资源映射，我们用管。
            registry.addInterceptor(new
LoginHandlerIntercepter()).addPathPatterns("/**")

.excludePathPatterns("/","/index.html","/user/login");
        }
    };
    return adapter;
 }
}
```

5. CRUD-员工列表

实验要求：

1) RestfulCRUD：CRUD满足Rest风格；

URI： /资源名称/资源标识 HTTP请求方式区分对资源CRUD操作

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

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

3) 员工列表

thymeleaf公共页面元素抽取

```
1、抽取公共片段
<div th:fragment="copy">
  &copy; 2011 The Good Thymes Virtual Grocery
</div>

2、引入公共片段
<div th:insert="~{footer :: copy}">
</div>
~{templatename::selector}：模板名::选择器 ~{templatename::fragmentname}:模板名::片段名

3、默认效果：

insert的公共片段在div标签中

如果使用th:insert等属性进行引入，可以不用写~{}
行内写法可以加上：[[~{}]];[(~{})];
```

三种引入公共片段的th属性：

th:insert：将公共片段整个插入到声明引入的元素中

th:replace：将声明引入的元素替换为公共片段

th:include：将被引入的片段的内容包含进这个标签中

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

引入方式:
<body>
  <div th:insert="footer :: copy"></div>
  <div th:replace="footer :: copy"></div>
  <div th:include="footer :: copy"></div>
</body>

引入效果:
<body>
  ...
  <div>
    <footer>
      &copy; 2011 The Good Thymes Virtual Grocery
    </footer>
  </div>

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

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

CRUD查看boot-web-crud。

11. 错误处理机制

效果

当我们访问一个没有的页面就会报404错误，抛出一个丑陋的页面。如果是移动端请求就会返回Json数据。这个错误页面可以定制，不过首先得了解错误处理机制：

之所以浏览器返回的是页面，其他客户端响应的是一个Json数据是因为发送请求的时候，浏览器发送的请求头的 Accept：text/html ，而其他客户端的 accept:"/" ,如果出现错误为何就出现这个404页面呢?

原理：

可以参照ErrorMvcAutoConfiguration这个类，错误处理的自动配置。自动给容器中添加了以下组件：

1. ErrorPageCustomizer

```
@Value("${error.path:/error}")
private String path = "/error"; // 系统出现错误发送 /error 请求
```

2. BasicErrorController：处理默认/error请求

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

@RequestMapping(produces = "text/html") // 浏览器发送的请求到这个方法处理，返回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) ? modelAndView : new
ModelAndView("error", model);
    }

    @RequestMapping
```

```
    @ResponseBody // JSON，其他客户端来到这个方法处理
    public ResponseEntity<Map<String, Object>> error(HttpServletRequest
request) {
        Map<String, Object> body = getErrorAttributes(request,
                isIncludeStackTrace(request, MediaType.ALL));
        HttpStatus status = getStatus(request);
        return new ResponseEntity<Map<String, Object>>(body, status);
    }
```

3. DefaultErrorViewResolver

```
private ModelAndView resolve(String viewName, Map<String, Object>
model) {
        //默认SpringBoot可以去找到一个页面？  error/404
        String errorViewName = "error/" + viewName;
        //模板引擎可以解析这个页面地址就用模板引擎解析
        TemplateAvailabilityProvider provider =
this.templateAvailabilityProviders
                .getProvider(errorViewName, this.applicationContext);
        if (provider != null) {
        //模板引擎可用的情况下返回到errorViewName指定的视图地址
            return new ModelAndView(errorViewName, model);
        }
         //模板引擎不可用，就在静态资源文件夹下找errorViewName对应的页面
error/404.html
        return resolveResource(errorViewName, model);
    }
```

4. DefaultErrorAttributes：共享页面信息

一旦系统出现了404或者500等错误的时候，ErrorPageCustomizer就会生效，来定制错误的响应规则，这是就相当于请求了 /error 。当发送了/error的请求，这时BasicErrorController就会起作用来处理这个 /error请求。这个BasicErrorControlle会有两种响应的方法,如果是浏览器访问就会返回一个返回html，如果是其他客户端访问就会返回一个Json数据。这样就会响应页面，而去那个页面是DefaultErrorAttributes解析到的，如果模板引擎有 error/4.4.html，就去访问它，如果没有就去找静态资源文件夹下面找errorViewName对应的页面。

12. 定制错误页面

自定义错误调整页面

1. 有模板引擎的情况下：error/状态码.html

将错误页面命名为 错误状态码.html 放在模板引擎文件夹里面的error文件夹下，发生此状态码的错误就会来到 对应的页面.

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

- 页面能获取的信息
    - timestamp：时间戳
    - status：状态码
    - error：错误提示
    - exception：异常对象
    - message：异常消息
    - errors：JSR303数据校验的错误都在这里

2. 没有模板引擎（模板引擎找不到这个错误页面），静态资源文件夹下找，无法动态获取值。
3. 以上都没有错误页面，就是默认来到SpringBoot默认的错误提示页

自定义返回Json数据

通过下面的代码实现自定义返回Json数据，不过不仅仅是其他客户端，浏览器返回的也是Json数据。

```
@ControllerAdvice
public class MyExceptionHandler {
    // 浏览器和其他客户端返回的都是Json
    @ResponseBody
    @ExceptionHandler(UserIsNoExist.class) // 自定义异常UserIsNoExist的处理方法
    public Map<String, Object> handlerException (Exception e) {
        HashMap<String, Object> map = new HashMap<>();
        map.put("code", "用户不存在");
        map.put("message", e.getMessage());
        return map;
    }
}
```

转发 /error 实现自适应错误处理

自适应就是：当出现错误的时候，会自行判断是浏览器还是其他客户端。如果是浏览器则返回404页面，如果是其他客户端则返回Json数据。

```
@ControllerAdvice
public class MyExceptionHandler {
    /**
     * 请求 /error  这个时候就会调用Spring Boot默认配置的组件处理
     */
    @ExceptionHandler(UserIsNoExist.class)
    public String  handlerException (Exception e) {
        HashMap<String, Object> map = new HashMap<>();
         request.setAttribute("javax.servlet.error.status_code", 500);
        map.put("code", "用户不存在");
        map.put("message", e.getMessage());
        return "forward:/error";
    }
}
```

出现异常的时候，请求 /error 。当发送了 /error 的请求，这时BasicErrorController就会起作用来处理这个 /error 请求。这个BasicErrorController会有两种响应的方法,如果是浏览器访问就会返回一个返回html，如果是其他客户端访问就会返回一个Json数据。

我们在template/error文件夹定义了4xx和5xx的处理页面，Spring Boot默认配置的BasicErrorController同样也是对状态码为4xx和5xx的才会处理。但是我们自定义的UserIsNoExist异常，状态码是：200，所以在处理异常的时候需要设置成：4xx或者5xx。只有这样才会进入错误页面的解析流程，会通过DefaultErrorAttributes进行解析，它会返回一个map，而这个map就是页面和Json能获取的所有字段。

将定制的数据携带出去

当发送了 /error 的请求，这时BasicErrorController就会起作用来处理这个 /error 请求。这个BasicErrorController会有两种响应的方法,如果是浏览器访问就会返回一个返回html，如果是其他客户端访问就会返回一个Json数据。响应出去可以获取的数据是由getErrorAttributes得到的，getErrorAttributes是AbstractErrorController（ErrorController）规定的方法。

如果想实现将定制的数据携带出去，方法有两个：

1. 完全来编写一个ErrorController的实现类[或者是编写AbstractErrorController的子类]，放在容器中
2. 页面上能用的数据，或者是json返回能用的数据都是通过errorAttributes.getErrorAttributes得到,容器中DefaultErrorAttributes.getErrorAttributes()默认进行数据处理的。 自定义ErrorAttributes。

```
@ControllerAdvice
public class MyExceptionHandler {
    @ExceptionHandler(UserIsNoExist.class)
    public String  handlerException (Exception e, HttpServletRequest
request) {
        HashMap<String, Object> map = new HashMap<>();
        /**
         * 必须自定义状态码，如何不自定义无法跳转到我们自己创建的5xx和4xx的错误页
面
         */
        request.setAttribute("javax.servlet.error.status_code", 500);
        map.put("code", "用户不存在");
        map.put("message", e.getMessage());
        // 把自定义的map放到request域里面，然后再去请求 /error ，接下来会去
DefaultErrorAttributes
        request.setAttribute("ext", map);
        return "forward:/error";
    }
}
```

补充阅读：转发和重定向的区别

```
@Component
public class MyErrorAttributes  extends DefaultErrorAttributes{

    // 返回值的map就是页面和json能获取所有的字段
    @Override
    public Map<String, Object> getErrorAttributes(RequestAttributes
requestAttributes, boolean includeStackTrace) {

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

```
        Map<String, Object> ext = (Map<String, Object>)
        // 刚刚传递过来的值
        requestAttributes.getAttribute("ext", 0);
        map.put("ext", ext);
        return map;
    }
}
```

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

自定义一个异常拦截器，拦截指定异常后把自定义的信息放在request中，这时候在重定向 /error ，然后通过BasicErrorController进行两种响应，紧接着通过DefaultErrorAttributes进行处理，他有一个方法getErrorAttributes，这个方法返回的map就是页面和json能获取的所有字段。所有我们自定义DefaultErrorAttributes，来获取request中放的数据，放到map中返回。

五、嵌入式Servlet容器

没有使用Spring Boot开发时，部署需要安装tomcat环境，项目打成war包后进行部署。而Spring Boot默认的使用tomcat作为嵌入的Servlet容器。

1. 修改Servlet容器配置

之前开发的时候如果想修改tomcat的配置，只需要找到配置文件修改即可。现在时嵌入式的Servlet容器，我们如何修改配置，有两种方式：

1) 在application.properties配置文件中进行配置，server相关配置与ServerProperties类绑定

```
#通用的servlet配置
server.port=8081
server.context-path=/crud

#tomcat的设置
servlet.tomcat.xxx
```

2) 编写一个嵌入式的Servlet容器的定制器：EmbeddedServletContainerCustomizer

```
package com.wrq.boot.config;
@Configuration
public class ConfigController extends WebMvcConfigurerAdapter {
    /**
     * 通过下面的bean实现自定义嵌入式容器配置
     * @return
     */
    @Bean EmbeddedServletContainerCustomizer
embeddedServletContainerCustomizer() {
        return new EmbeddedServletContainerCustomizer() {
            @Override
            public void customize(ConfigurableEmbeddedServletContainer
container) {
                container.setPort(8086);
            }
        };
    }

}
```

2. 注册三大组件：Servlet、Filter、Listener

由于Spring Boot默认是Jar包的方式启动嵌入式的Servlet容器来启动Spring Boot的应用，没有web.xml.

对三大组件使用下面的方式

- ServletRegistrationBean

```
@Configuration
public class MyConfigController {
    @Bean
    public ServletRegistrationBean testServlet() {
        // 访问 /servlet 的时候就会调用 TestServlet
        ServletRegistrationBean servletRegistrationBean = new
ServletRegistrationBean(new TestServlet(), "/servlet");
        return servletRegistrationBean;
    }
}
```

- FilterRegistrationBean

```
 @Bean
public FilterRegistrationBean filterRegistrationBean () {
    FilterRegistrationBean filterRegistrationBean = new
FilterRegistrationBean();
    filterRegistrationBean.setFilter(new TestFilter());

filterRegistrationBean.setUrlPatterns(Arrays.asList("/helloMan","/filte
r"));
    return filterRegistrationBean;
}
```

- ServletListenerRegistrationBean

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

SpringBoot帮我们自动配置的SpringMVC的时候，自动的注册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；包静态资源，但是不拦截jsp请求 /*会拦截jsp
//可以通过server.servletPath来修改SpringMVC前端控制器默认拦截的请求路径
    registration.setName(DEFAULT_DISPATCHER_SERVLET_BEAN_NAME);
    registration.setLoadOnStartup(
            this.webMvcProperties.getServlet().getLoadOnStartup());
    if (this.multipartConfig != null) {
        registration.setMultipartConfig(this.multipartConfig);
```

```
    }
    return registration;
}
```

3. 切换Servlet容器

Spring Boot默认支持的时tomcat服务器，但是它支不支持其他的Servlet容器呢?

- tomcat 默认容器配置
- Jetty 长连接，适合聊天应用
- Undertow 适合高并发，不支持JSP

Spring Boot默认tomcat作为容器是因为:web模块的start的依赖是tomcat的:

```
graph LR
spring-boot-starter-web-->sping-boot-starter-tomcat
sping-boot-starter-tomcat-->tomcat-embed-core
tomcat-embed-core-->tomcat-annotations-api
sping-boot-starter-tomcat-->tomcat-embed-el
sping-boot-starter-tomcat-->tomcat-embed-el-websocket
```

修改容器为：Jetty

```
<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>
```

修改容器为：Undertow

```
<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-undertow</artifactId>
</dependency>
```

4. 自动配置原理以及启动原理

视频教程

5. 使用外部的Servlet容器

嵌入式Servlet容器优点:

- 应用打成可执行的jar
- 简单
- 便携

嵌入式Servlet容器缺点:

- 默认不支持JSP
- 优化定制比较复杂

1. 创建一个War包的项目
2. 自己创建webapp和web.xml页面
3. 将嵌入式的Tomcat指定为provided

```
<!-- 意思是已经提供了tomcat的环境，打包不需要携带 -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-tomcat</artifactId>
    <scope>provided</scope>
</dependency>
```

4. 必须编写一个SpringBootServletInitializer的子类，并调用configure方法

```
public class ServletInitializer extends SpringBootServletInitializer {
    @Override
    protected SpringApplicationBuilder
configure(SpringApplicationBuilder application) {
        return
application.sources(BootExternalContainerApplication.class);
    }
}
```

5. 启动服务器就可以使用

6. 外部容器的相关原理

视频教程

五、Docker

1. 什么是Docker?

Docker是一个开源的应用容器引擎，基于Go语言并遵从Apache2.0协议开源。Docker可以让开发者打包他们的应用以及依赖包到一个轻量级、可移植的容器中， 然后发布到任何流行的 Linux 机器上，也可以实现虚拟化。

容器是完全使用沙箱机制，相互之间不会有任何接口,更重要的是容器性能开销极低。Docker支持将软件编译成一个镜像。然后 在镜像中各种软件做好配置，将镜像发布出去，其他使用者可以直接使用这个镜像。

如何理解呢，我们举个例子：

相信在刚刚接触JaveWeb的时候，都做过单体应用。而在Linux服务器部署一个单体JavaWeb应用，一般会在服务器安装Tomcat、MySql、Redis、JDK等相关环境或软件，安装完软件之后需要进行相关配置，最后把项目打成War包，放在服务器进行部署。这样有几个缺点，那就是面对黑糊糊的命令行，如果想部署成功需要一定的Linux知识储备，再者就是如果我们想在另一台服务器上部署，也需要重复刚刚的下载软件、配置环境、部署，极为繁琐。而Docker作为一门容器技术，很好的解决这一问题。

我们只需要在一台Linux机器上完成软件的安装和配置，然后把他们做成镜像，MySQL做成MySQL-Docker镜像，Tomcat做成Tomcat-Docker镜像。当我们在另一台Linux服务器安装的时候只需要安装Docker这个软件，然后把镜像拿过来运行即可，这个镜像就成了一个容器。容器启动是非常快速的。类似windows里面的ghost操作系统，安装好后什么都有了，这样就降低了对linux操作的难度。

2. Docker的核心概念

- docker镜像(Images)：Docker 镜像是用于创建Docker 容器的 模板。
- docker容器(Container)：容器是独立运行的一个或一组应用。
- docker客户端(Client)：客户端通过命令行或者其他工具使用 Docker API 与Docker的守护进程通信
- docker主机(Host)：一个物理或者虚拟的机器用于执行 Docker 守护进程和容器
- docker仓库(Registry)：Docker 仓库用来保存镜像，可以理解 为代码控制中的代码仓库，Docker Hub(https://hub.docker.com) 提供了庞大的镜像集合供使用

3. Docker的使用

安装Linux虚拟机：VirtualBox官方下载

六、数据访问

1. 简介

对于数据访问层，无论是SQL还是NOSQL，Spring Boot默认采用整合 Spring Data的方式进行统一处理，添加大量自动配置，屏蔽了很多设置。引入 各种xxxTemplate，xxxRepository来简化我们对数据访问层的操作。对我们来 说只需要进行简单的设置即可。

- JDBC

- MyBatis
- JPA

2. JDBC

1. 初始化项目，导入数据访问依赖

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

2. 编写配置

```
spring:
  datasource:
    username: root
    password: admin
    url: jdbc:mysql://localhost:3306/jdbc?characterEncoding=utf-8
    driver-class-name: com.mysql.jdbc.Driver
```

3. JDBC默认采用 org.apache.tomcat.jdbc.pool.DataSource 数据源，数据源的相关配置参考：DataSourceProperties。

原理

参考：org.springframework.boot.autoconfigure.jdbc

- 参考DataSourceConfiguration，根据配置创建数据源，默认使用Tomcat连接池；可以使用 spring.datasource.type指定自定义的数据源类型；
- SpringBoot默认可以支持：
    - org.apache.tomcat.jdbc.pool.DataSource
    - HikariDataSource
    - BasicDataSource

- 下面代码表明我们可以自定义数据源类型

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

    // 使用DataSourceBuilder创建数据源，利用反射创建相关的数据源，并绑定相关属性。
    @Bean
    public DataSource dataSource(DataSourceProperties properties) {
        return properties.initializeDataSourceBuilder().build();
    }
}
```

自动执行建表语句

DataSourceInitializer：ApplicationListener

作用：

1. runSchemaScripts() 运行建表语句
2. runDataScripts() 运行插入数据的sql语句 默认只需要将文件命名为：

```
schema-*.sql、data-*.sql

1. 默认规则：schema.sql，schema-all.sql，放在类路径下，启动就会执行文件中的语句

2. 在yml中可以使用下面配置指定位置
schema:
    - classpath:department.sql
```

操作数据库

操作数据库：自动配置了JdbcTemplate操作数据库

在JdbcTemplateAutoConfigration中：

```
@Bean // /容器中注入了 JdbcTemplate 的bean
@Primary
@ConditionalOnMissingBean(JdbcOperations.class)
public JdbcTemplate jdbcTemplate() {
    return new JdbcTemplate(this.dataSource);
}
```

由于Spring Boot自动配置了JdbcTemplate，我们可以从容器中拿来使用：

```
@Controller
public class HelloController {

    @Autowired
    JdbcTemplate jdbcTemplate;

    @ResponseBody
    @RequestMapping("/hello")
    public List hello () {
        List<Map<String, Object>> maps =
jdbcTemplate.queryForList("select * from department");
        return  maps;
    }
}
```

3. 配置数据源

默认采用 org.apache.tomcat.jdbc.pool.DataSource ， 但是实际开发中很少使用这个数据源。

整合**Druid**数据源：

1. 导入依赖

```
<!-- https://mvnrepository.com/artifact/com.alibaba/druid -->
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid</artifactId>
    <version>1.1.10</version>
</dependency>
```

2. 修改配置

```
spring:
  datasource:
    username: root
    password: admin
    url: jdbc:mysql://localhost:3306/jdbc?characterEncoding=utf-8
    driver-class-name: com.mysql.jdbc.Driver
    type: com.alibaba.druid.pool.DruidDataSource
#   下面的配置如果想生效需要自己配置
    initialSize: 5
    minIdle: 5
    maxActive: 20
    maxWait: 60000
    timeBetweenEvictionRunsMillis: 60000
    minEvictableIdleTimeMillis: 300000
    validationQuery: SELECT 1 FROM DUAL
    testWhileIdle: true
    testOnBorrow: false
    testOnReturn: false
    poolPreparedStatements: true
```

我们需要配置 type： com.alibaba.druid.pool.DruidDataSource 来切换数据源，但是我们下面在配置文件中的配置是不会生效的。

```
initialSize: 5
minIdle: 5
maxActive: 20
maxWait: 60000
timeBetweenEvictionRunsMillis: 60000
minEvictableIdleTimeMillis: 300000
validationQuery: SELECT 1 FROM DUAL
testWhileIdle: true
testOnBorrow: false
testOnReturn: false
poolPreparedStatements: true
```

是因为username、password属性和DatasourceProperties中的属性进行绑定了，但是initialSize、minIdle等属性没法与DatasourceProperties中的属性绑定，它需要和DruidDataSource进行绑定。

我们需要自定义一个Druid数据源来进行绑定：

```
@Configuration
public class Config {

    // 把配置文件中spring.datasource.initialSize等属性和DruidDataSource中属性进行绑定
    @ConfigurationProperties(prefix = "spring.datasource")
    @Bean
    public DruidDataSource druidDataSource () {
        return new DruidDataSource();
    }
}
```

3. 配置Druid监控、过滤器

```
@Configuration
public class DruidConfig {

    /**
     * spring.datasource中的配置和DruidDataSource中的属性绑定
     * @return
     */
    @ConfigurationProperties(prefix = "spring.datasource")
    @Bean
    public DruidDataSource druidDataSource () {
        return new DruidDataSource();
    }

    /**
     * 配置Druid的监控
     * 1、配置一个管理后台的Servlet，拦截登陆
     * @return
     */
    @Bean
    public ServletRegistrationBean statViewServlet(){
```

```
        ServletRegistrationBean servletRegistrationBean = new
ServletRegistrationBean(new StatViewServlet(),"/druid/*");
        Map<String,String> initParams = new HashMap<>();
        initParams.put("loginUsername","admin");
        initParams.put("loginPassword","123456");
        initParams.put("allow",""); // 允许所有访问
        servletRegistrationBean.setInitParameters(initParams);
        return servletRegistrationBean;
    }

    // 配置一个web监控的filter，哪些请求会被监控，哪些排除。
    @Bean
    public FilterRegistrationBean webStatFilter() {
        FilterRegistrationBean bean = new FilterRegistrationBean(new
WebStatFilter());
        Map<String,String> initParams = new HashMap<>();
        initParams.put("exclusions","*.js,*.css,/druid/*");
        bean.setInitParameters(initParams);
        bean.setUrlPatterns(Arrays.asList("/*"));
        return bean;
    }
}
```

当我们访问：http://localhost:8080/druid 就会进入监控页面，可以查看执行哪些SQL，已经时间等性能。

4. 整合MyBatis-注解版

1. 导入依赖,mybatis-spring-boot-starter是MyBatis官方提供的。

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

```
    <artifactId>mybatis-spring-boot-starter</artifactId>
    <version>1.3.2</version>
</dependency>

<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <scope>runtime</scope>
</dependency>
```

2. 配置Druid数据源

```
<!-- https://mvnrepository.com/artifact/com.alibaba/druid -->
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid</artifactId>
    <version>1.1.10</version>
</dependency>
```

3. 编写数据源配置

```
spring:
  datasource:
#   数据源基本配置
    username: root
    password: 123456
    driver-class-name: com.mysql.jdbc.Driver
    url: jdbc:mysql://localhost:3306/mybatis
    type: com.alibaba.druid.pool.DruidDataSource
#   数据源其他配置
    initialSize: 5
    minIdle: 5
    maxActive: 20
    maxWait: 60000
    timeBetweenEvictionRunsMillis: 60000
    minEvictableIdleTimeMillis: 300000
    validationQuery: SELECT 1 FROM DUAL
    testWhileIdle: true
    testOnBorrow: false
    testOnReturn: false
```

```
    poolPreparedStatements: true
#   配置监控统计拦截的filters，去掉后监控界面sql无法统计，'wall'用于防火墙
    filters: stat,wall,log4j
    maxPoolPreparedStatementPerConnectionSize: 20
    useGlobalDataSourceStat: true
    connectionProperties:
druid.stat.mergeSql=true;druid.stat.slowSqlMillis=500
```

4. Druid监控以及属性绑定

```
@Configuration
public class DruidConfig {

    /**
     * spring.datasource中的配置和DruidDataSource中的属性绑定
     * @return
     */
    @ConfigurationProperties(prefix = "spring.datasource")
    @Bean
    public DruidDataSource druidDataSource () {
        return new DruidDataSource();
    }

    /**
     * 配置Druid的监控
     * 1、配置一个管理后台的Servlet，拦截登陆
     * @return
     */
    @Bean
    public ServletRegistrationBean statViewServlet(){
        ServletRegistrationBean servletRegistrationBean = new
ServletRegistrationBean(new StatViewServlet(),"/druid/*");
        Map<String,String> initParams = new HashMap<>();
        initParams.put("loginUsername","admin");
        initParams.put("loginPassword","123456");
        initParams.put("allow",""); // 允许所有访问
        servletRegistrationBean.setInitParameters(initParams);
        return servletRegistrationBean;
    }
```

```
    // 配置一个web监控的filter，哪些请求会被监控，哪些排除。
    @Bean
    public FilterRegistrationBean webStatFilter() {
        FilterRegistrationBean bean = new FilterRegistrationBean(new
WebStatFilter());
        Map<String,String> initParams = new HashMap<>();
        initParams.put("exclusions","*.js,*.css,/druid/*");
        bean.setInitParameters(initParams);
        bean.setUrlPatterns(Arrays.asList("/*"));
        return bean;
    }
}
```

5. 编写Bean

```
public class Department {
    private Integer id;
    private String departmentName;
    // getter、setter方法已经省略
}
```

6. 编写Mapper接口 @Mapper注解是必须的，如果我们的Mapper特别的，每个都加太麻烦，可以在**主程序类**上面加上注解@MapperScan(value = "com.xxxx")扫描即可。

```
@Mapper
public interface DepartmentMapper {

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

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

    /**
     * Options注解：当插入一条数据后生成id，这个新生成的id会再次封装进来。
     * 加了Options注解，把插入的数据再返回的时候会把刚刚生成的id封装进去
     * @param department
     * @return
     */
```

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

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

7. 编写Controller

```
@Controller
public class DepartmentController {

    @Autowired
    private DepartmentMapper departmentMapper;

    @ResponseBody
    @RequestMapping(value = "/dept/{id}",method = RequestMethod.GET)
    public Department getDepartment(@PathVariable("id") Integer id){
        return departmentMapper.getDeptById(id);
    }

    /**
     * mapper中不添加Options注解，返回的json的id为空，加了Options注解，把插入
的数据再返回的时候会把刚刚生成的id封装进去
     * @param department
     * @return
     */
    @ResponseBody
    @RequestMapping(value = "/dept", method = RequestMethod.GET)
    public Department insertDepartment(Department department){
        departmentMapper.insertDept(department);
        return department;
    }
}
```

8. 开启驼峰命名，自定义配置

若数据库的自动为：department_name,我们的Bean中定义的属性为departmentName。按照以前需要开启驼峰命名法的配置，使用Sping Boot自定义MyBatis的配置：

```
@org.springframework.context.annotation.Configuration
public class MyBatisConfig {

    @Bean
    public ConfigurationCustomizer configurationCustomizer(){
        return new ConfigurationCustomizer() {
            @Override
            public void customize(Configuration configuration) {
                // 开启驼峰命名
                configuration.setMapUnderscoreToCamelCase(true);
            }
        };
    }
}
```

4. 整合MyBatis-配置版

1. 导入依赖,mybatis-spring-boot-starter是MyBatis官方提供的。

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jdbc</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>org.mybatis.spring.boot</groupId>
    <artifactId>mybatis-spring-boot-starter</artifactId>
    <version>1.3.2</version>
</dependency>

<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
```

```
    <scope>runtime</scope>
</dependency>
```

2. 配置Druid数据源

```
<!-- https://mvnrepository.com/artifact/com.alibaba/druid -->
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid</artifactId>
    <version>1.1.10</version>
</dependency>
```

3. 编写数据源配置

```
spring:
  datasource:
#   数据源基本配置
    username: root
    password: 123456
    driver-class-name: com.mysql.jdbc.Driver
    url: jdbc:mysql://localhost:3306/mybatis
    type: com.alibaba.druid.pool.DruidDataSource
#   数据源其他配置
    initialSize: 5
    minIdle: 5
    maxActive: 20
    maxWait: 60000
    timeBetweenEvictionRunsMillis: 60000
    minEvictableIdleTimeMillis: 300000
    validationQuery: SELECT 1 FROM DUAL
    testWhileIdle: true
    testOnBorrow: false
    testOnReturn: false
    poolPreparedStatements: true
#   配置监控统计拦截的filters，去掉后监控界面sql无法统计，'wall'用于防火墙
    filters: stat,wall,log4j
    maxPoolPreparedStatementPerConnectionSize: 20
    useGlobalDataSourceStat: true
    connectionProperties:
druid.stat.mergeSql=true;druid.stat.slowSqlMillis=500
```

4. Druid监控以及属性绑定

```
@Configuration
public class DruidConfig {

    /**
     * spring.datasource中的配置和DruidDataSource中的属性绑定
     * @return
     */
    @ConfigurationProperties(prefix = "spring.datasource")
    @Bean
    public DruidDataSource druidDataSource () {
        return new DruidDataSource();
    }

    /**
     * 配置Druid的监控
     * 1、配置一个管理后台的Servlet，拦截登陆
     * @return
     */
    @Bean
    public ServletRegistrationBean statViewServlet(){
        ServletRegistrationBean servletRegistrationBean = new
ServletRegistrationBean(new StatViewServlet(),"/druid/*");
        Map<String,String> initParams = new HashMap<>();
        initParams.put("loginUsername","admin");
        initParams.put("loginPassword","123456");
        initParams.put("allow",""); // 允许所有访问
        servletRegistrationBean.setInitParameters(initParams);
        return servletRegistrationBean;
    }

    // 配置一个web监控的filter，哪些请求会被监控，哪些排除。
    @Bean
    public FilterRegistrationBean webStatFilter() {
        FilterRegistrationBean bean = new FilterRegistrationBean(new
WebStatFilter());
        Map<String,String> initParams = new HashMap<>();
        initParams.put("exclusions","*.js,*.css,/druid/*");
        bean.setInitParameters(initParams);
```

```
        bean.setUrlPatterns(Arrays.asList("/*"));
        return bean;
    }
}
```

5. 编写Bean

```
package com.wrq.boot.bean;
public class Employee {
    private Integer id;
    private String lastName;
    private Integer gender;
    private String email;
    private Integer dId;
    ...
}
```

6. 编写mapper

```
@Mapper
public interface EmployeeMapper {

    public Employee getEmpById(Integer id);

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

7. 主配值文件:resources->mybatis->mybatis-config.xml

```
<?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>
```

8. mapper配置文件:resources->mybatis->mapper->*.xml

```
<?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.wrq.boot.mapper.EmployeeMapper">
    <select id="getEmpById" resultType="com.wrq.boot.bean.Employee">
      SELECT * FROM employee WHERE id=#{id}
    </select>

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

9. 主配置文件添加以下配置

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

10. 编写Controller

```
@RestController
public class EmployeeController {

    @Autowired
    private EmployeeMapper employeeMapper;

    @GetMapping("/emp/{id}")
    public Employee getEmployee(@PathVariable("id") Integer id){
        return employeeMapper.getEmpById(id);
    }
}
```

4. 整合JPA

1. 导入依赖

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

<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <scope>runtime</scope>
</dependency>
```

2. 编写配置文件

```
spring:
  datasource:
    username: root
    password: admin
    url: jdbc:mysql://localhost:3306/jpa?characterEncoding=utf-8
    driver-class-name: com.mysql.jdbc.Driver
  jpa:
    hibernate:
      ddl-auto: update
    show-sql: true
```

3. 编写实体类

```
//使用JPA注解配置映射关系
@Entity //告诉JPA这是一个实体类（和数据表映射的类）
@Table(name = "tbl_user") //@Table来指定和哪个数据表对应;如果省略默认表名就是
user;
public class User {
```

```
    @Id //这是一个主键
    @GeneratedValue(strategy = GenerationType.IDENTITY)//自增主键
    private Integer id;

    @Column(name = "last_name", length = 50) //这是和数据表对应的一个列
    private String lastName;
    @Column //省略默认列名就是属性名
    private String email;
    ...
}
```

4. 编写repository

```
package com.wrq.boot.repository;

public interface UserRepository extends JpaRepository<User,Integer> {
}
```

5. 编写controller

```
package com.wrq.boot.controller;
@RestController
public class UserController {

    @Autowired
    UserRepository repository;

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

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

参考文献：

尚硅谷Spring Boot视频

注解机制及其原理

@SpringBootApplication注解分析