1. **SpringBoot入门**
   1. **sprig boot简介**
      1. **简介:**

> 简化Spring应用开发的一个框架；

> 整个Spring技术栈的一个大整合；

> J2EE开发的一站式解决方案；

* + 1. **背景:**

J2EE笨重的开发、繁多的配置、低下的开发效率、复杂的部署流程、第三方技术集成难度大。

* + 1. **解决:**

“Spring全家桶” 时代

Spring Boot → J2EE开发一站式解决方案

Spring Could → 分布式整体解决方案

* + 1. **优缺点:**

快速创建独立运行的Spring项目以及与主流框架集成

使用嵌入式的servlet容器,应用无需打成WAR包

starters自动依赖与版本控制

大量的自动配置,简化开发也可修改默认值

无需配置XML,无代码生成,开箱即用

准生产环境的运行时应用监控

与云计算的天然集成

缺 : 入门容易精通难,需掌握spring的API

* 1. **微服务**

一种架构风格(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;

每个功能元素最终都是一个可独立替换和独立升级的软件单位;

* 1. **SpringBoot HelloWorld**

**目标功能:**

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

* + - 1. **创建一个maven工程；（jar）**
      2. **导入spring boot相关的依赖;**

<parent>

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

<artifactId>spring-boot-starter-parent</artifactId>

<version>1.5.9.RELEASE</version>

</parent>

<dependencies>

<dependency>

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

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

</dependencies>

* + - 1. **编写一个主程序；启动Spring Boot应用**

@SpringBootApplication // 来标注一个主程序类，说明这是一个Spring Boot应用

public class HelloWorldMainApplication {

public static void main(String[] args) {

// 应用启动Spring

SpringApplication.run(HelloWorldMainApplication.class,args);

}

}

* + - 1. **编写相关的Controller、Service**

@Controller

public class HelloController {

@ResponseBody

@RequestMapping("/hello")

public String hello(){

return "Hello World!";

}

**}**

* + - 1. **运行主程序测试**
      2. **简化部署**

<!-- 这个插件，可以将应用打包成一个可执行的jar包；-->

<build>

<plugins>

<plugin>

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

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

</build>

将这个应用打成jar包，直接使用java -jar的命令进行执行；

* 1. **Hello World探究**
     1. **POM文件**
        1. **父项目**

<parent>

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

<artifactId>spring-boot-starter-parent</artifactId>

<version>1.5.9.RELEASE</version>

</parent>

parent的父项目是dependencies

<parent>

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

<artifactId>spring-boot-dependencies</artifactId>

<version>1.5.9.RELEASE</version>

<relativePath>../../spring-boot-dependencies</relativePath>

</parent>

dependencies来真正管理Spring Boot应用里面的所有依赖版本；

Spring Boot的版本仲裁中心；

以后我们导入依赖默认是不需要写版本；（没有在dependencies里面管理的依赖自然需要声明版本号）

* + - 1. **启动器**

<dependency>

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

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

**spring-boot-starter-web：**

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

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

* + 1. **主程序类，主入口类(@SpringBootApplication)**

**定义:**

标注在Spring Boot应用某个类上,说明这个类是SpringBoot的主配置类，SpringBoot就应该运行这个类的main方法来启动SpringBoot应用；

**具体内部内容**  
@Target(ElementType.TYPE)

@Retention(RetentionPolicy.RUNTIME)

@Documented

@Inherited

**@SpringBootConfiguration** //Spring Boot的配置类；标注在某个类上，表示这是一个Spring Boot的配置类；

//具体内部内容:@\*\*Configuration\*\*:配置类上来标注这个注解；配置类指的就是配置文件；配置类也是容器 //中的一个组件→@Component

**@EnableAutoConfiguration** //开启自动配置功能;以前我们需要配置的东西，现在Spring Boot帮我们自动配置；

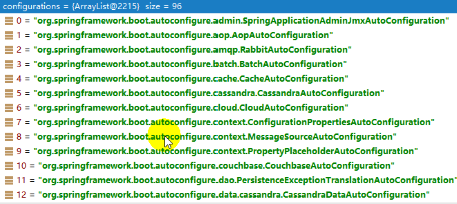
//告诉SpringBoot开启自动配置功能；这样自动配置才能生效

//具体内部内容: **@AutoConfigurationPackage**  //自动配置包

**@Import(EnableAutoConfigurationImportSelector.class)** /\*Spring的底层注解@Import，给容器中导入一个组件导入的组件 由AutoConfigurationPackages.Registrar.class将主配置类（@SpringBootApplication标注的类）的所在包及下面所有子包里面 的所有组件扫描到Spring容器句号。\*/

/\***EnableAutoConfigurationImportSelector**：导入哪些组件的选择器；使用父类的方法**selectImports**将所有需要导入的组件以 全类名的方式返回；这些组件就会被添加到容器中会给容器中导入非常多的自动配置类（xxxAutoConfiguration）；就是 给容器中导入这个场景需要的所有组件，并配置好这些组件；有了自动配置类，免去了我们手动编写配置注入功能组件 等的工作；

/\***SpringFactoriesLoader.loadFactoryNames(EnableAutoConfiguration.class,classLoader)**；Spring Boot在启动的时候从类路径下 的META-INF/spring.factories中获取EnableAutoConfiguration指定的值，将这些值作为自动配置类导入到容器中，自动配置 类就生效，帮我们进行自动配置工作;J2EE的整合解决方案和自动配置都在spring-boot-autoconfigure-1.5.9.RELEASE.jar；\*/



public @interface EnableAutoConfiguration {...}

@ComponentScan(excludeFilters = {

@Filter(type = FilterType.CUSTOM, classes = TypeExcludeFilter.class),

@Filter(type = FilterType.CUSTOM, classes = AutoConfigurationExcludeFilter.class) })

public @interface SpringBootApplication {

* 1. **使用Spring Initializer快速创建Spring Boot项目**

IDEA：使用 Spring Initializer快速创建项目

IDE都支持使用Spring的项目创建向导快速创建一个Spring Boot项目；选择我们需要的模块；向导会联网创建Spring Boot项目；默认生成的Spring Boot项目,主程序已经生成好了，我们只需要我们自己的逻辑

- resources文件夹中目录结构

- static：保存所有的静态资源； js css images；

- templates：保存所有的模板页面；（Spring Boot默认jar包使用嵌入式的Tomcat，默认不支持JSP页面）；可以使用模板引擎（freemarker、thymeleaf）；

- application.properties：Spring Boot应用的配置文件；可以修改一些默认设置；

1. **配置文件**
   1. **配置文件**

SpringBoot使用一个全局的配置文件,配置文件名称是固定的;

application.properties

application.yml

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

YAML（YAML Ain't Markup Language）

YAML A Markup Language：是一个标记语言

YAML isn’t Markup Language: 不是一个标记语言

YAML：配置例子

server:

port: 8081

* 1. **YAML的语法:**
     1. 基本语法:

k:(空格) v: 表示一对键值对(空格必须有)

以\*\*空格\*\*的缩进方式来控制层级关系,只要是左对齐的一列数据,都是同一个级别的。

server:

port: 8081

path: /hello

\*大小写敏感

* + 1. 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表达的意思

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字符串数据

name: ‘zhangsan \n lisi’：输出；zhangsan \n lisi

对象、map(属性和值)(键值对):

k: v: 在下一行写对象的属性和值的关系;注意缩进;

friends:

lastName: zhangsan

age: 20

行内写法：

friends: {lastName: zhangsan,age: 18}

数组(list,set):

用- 值表示数组中的一个元素

pets:

- cat

- dog

- pig

行内写法

pets: [cat,dog,pig]

* 1. **配置文件值注入**

**application.yml**

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

**application.properties (乱码问题到file encoding配置中去解决)**

**person.age**=**21  
person.birth**=**2013/01/01  
person.boss**=**true  
person.last-name**=**gag  
person.maps.k1**=**v1  
person.maps.k2**=**v2  
person.lists**=**1,2,3,4  
person.dog.name**=**xiaohua  
person.dog.age**=**2**

**javaBean：**

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

**测试类:**

@RunWith(SpringRunner.**class**)  
@SpringBootTest  
**public class** Springboot01QuickApplicationTests {  
  
 @Autowired  
 Person **person**;  
 @Test  
 **public void** contextLoads() {  
 System.***out***.println(**person**.toString());  
 }  
}

**导入配置文件处理器，以后编写配置就有提示了**

*<!--导入配置文件处理器，配置文件进行绑定就会有提示-->*

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

* + 1. **@Value获取值和@ConfigurationProperties获取值比较**



配置文件无论是yml还是properties他们都能获取到值；

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

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

* + 1. **配置文件注入值数据校验(JSR303数据校验:)**

@Component *//只有这个组件是容器中的组件，才能容器提供的@ConfigurationProperties功能；*@ConfigurationProperties(prefix = **"person"**)  
@Validated  
**public class** Person {  
 @Email *////用于数据校验标注的必须为Email格式(搭配@Validated使用)* **private** String **lastName**;

* + 1. **@PropertySource&@ImportResource&@Bean**

**@PropertySource ：加载指定的配置文件(用于配置文件的分类)；**

**person.properties:**

**server.port**=**8080 #这里的配置的端口也会生效  
person.last-name**=**李四  
person.age**=**18  
person.birth**=**2017/12/15  
person.boss**=**false  
person.maps.k1**=**v1  
person.maps.k2**=**14  
person.lists**=**a,b,c  
person.dog.name**=**dog  
person.dog.age**=**15**

**Person.java**

@ConfigurationProperties(prefix = **"person"**)  
@Component *//只有这个组件是容器中的组件，才能容器提供的@ConfigurationProperties功能；*@PropertySource(**"classpath:person.properties"**)*//将***person.properties加入到spring容器中**  
**public class** Person {**private** String **lastName**;

**测试类**

@Test  
**public void** contextLoads() {  
 System.***out***.println(**person**.toString());  
}

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

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

**想让Spring的配置文件生效，加载进来；@ImportResource标注在一个配置类上(spring推荐全注解的方式见@bean)**

**beans.xml**

<**bean id="helloService" class="com.walkwind.springboot\_01\_quick.service.HelloService"**></**bean**>

**Springboot01QuickApplication.java spring启动入口**

@SpringBootApplication  
@ImportResource(**"classpath:beans.xml"**) *//导入spring的配置文件***public class** Springboot01QuickApplication {  
  
 **public static void** main(String[] args) {  
 SpringApplication.run(Springboot01QuickApplication.**class**, args);  
 }

**测试类**

@RunWith(SpringRunner.**class**)  
@SpringBootTest  
**public class** Springboot01QuickApplicationTests {  
 @Autowired  
 ApplicationContext **ioc**;  
 @Test  
 **public void** contextLoads() {**boolean** helloService = **ioc**.containsBean(**"helloService"**);  
 System.***out***.println(helloService);  
 }

**@bean与@Configuration**

**Springboot推荐给容器中添加组件的方式:使用全注解的方式**

**自定义配置类**

*/\*\*  
 \** ***@Configuration指定当前类是一个配置类,为容器中添加组件*** *\* \*/*@Configuration  
**public class** Myconfig {  
 @Bean *//将方法的返回值添加到容器中；容器中这个组件默认的id就是方法名* **public** HelloService helloService(){  
 System.***out***.println(**"配置类@Bean给容器中添加组件"**);  
 **return new** HelloService();  
 }

* 1. **配置文件占位符**

**随机数、占位符获取之前配置的值，如果没有可以是用:指定默认值**

**person.last-name**=**张三${random.value}  
person.age**=**${random.int}  
person.birth**=**2017/12/15  
person.boss**=**false  
person.maps.k1**=**${random.int[1024,65536]}  
person.maps.k2**=**${random.int(10)}  
person.maps.k3**=**${random.long}  
person.lists**=**a,b,c  
person.dog.name**=**${person.who:不知是谁的}\_dog  
person.dog.age**=**${person.age}**

**创建出的person对象**

**Person{lastName='张三**

**1d6be0e27c486c9096786ddea9f74a4a',**

**age=2132294672,**

**boss=false,**

**birth=Fri Dec 15 00:00:00 CST 2017,**

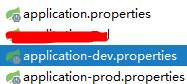
**maps={k1=8303, k2=9, k3=-4686376938916000878},**

**lists=[a, b, c],**

**dog=Dog{name='不知是谁的\_dog', age=1802414262}}**

* 1. **Profile多环境支持**
     1. **多profile**
        1. **properties中的多profile配置**

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



**默认使用application.properties的配置可以在application.propertise中指定使用那个配置文件运行spring；**

**spring.profiles.active**=**dev**

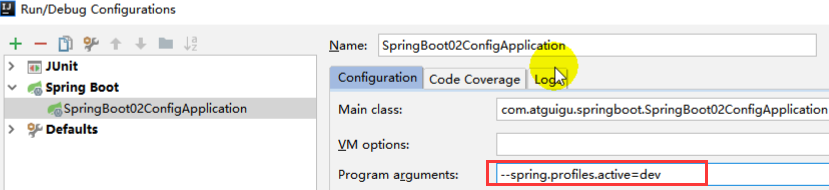
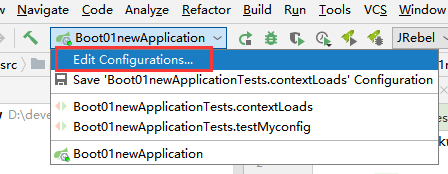
* + - 1. **yaml中的多profile**

**server:  
 port:** 8081  
**spring:  
 profiles:  
 active:** prod *#激活指定配置*---  
**server:  
 port:** 8083  
**spring:  
 profiles:** dev  
  
--- *#分割线分割多喝profile区域***server:  
 port:** 8084  
**spring:  
 profiles:** prod *#指定属于哪个环境*

* + 1. **激活指定profile**
       1. **在配置文件中指定 spring.profiles.active=dev**
       2. **命令行：**

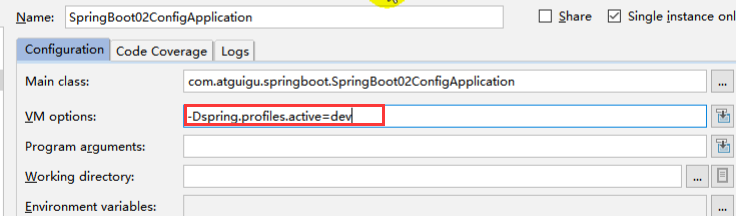
**打包后使用命令行:java -jar spring-boot-02-config-0.0.1-SNAPSHOT.jar --spring.profiles.active=dev；；**

**在开发过程中可以，配置传入命令行参数 --spring.profiles.active=dev**



* + - 1. **虚拟机参数；**

**-Dspring.profiles.active=dev**



* 1. **配置文件的加载顺序**

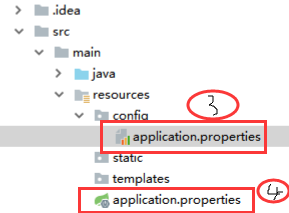
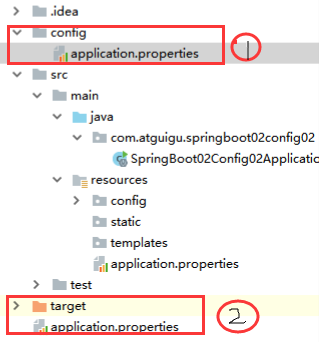
**springboot 启动会扫描以下位置的application.properties或者application.yml文件作为Spring boot的默认配置文件**

–file:./config/

–file:./

–classpath:/config/

–classpath:/



优先级由高到底，高优先级的配置会覆盖低优先级的配置；

SpringBoot会从这四个位置全部加载主配置文件；**互补配置(多个配置文件共同提供配置,高优先级覆盖低优先级)**；

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

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

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

* 1. **外部配置加载顺序**

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

* + 1. **命令行参数**

所有的配置都可以在命令行上进行指定

java -jar spring-boot-02-config-02-0.0.1-SNAPSHOT.jar --server.port=8087 --server.context-path=/abc

多个配置用空格分开； --配置项=值

* + 1. **来自java:comp/env的JNDI属性**
    2. **Java系统属性（System.getProperties()）**
    3. **操作系统环境变量**
    4. **RandomValuePropertySource配置的random.\*属性值**

**由jar包外向jar包内进行寻找；**

**优先加载带profiles**

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

**再来加载不带profile**

* + 1. **jar包外部的application.properties或application.yml(不带spring.profile)配置文件**
    2. **jar包内部的application.properties或application.yml(不带spring.profile)配置文件**
    3. **@Configuration注解类上的@PropertySource**
    4. **通过SpringApplication.setDefaultProperties指定的默认属性**

**所有支持的配置加载来源；**

**[参考官方文档](https://docs.spring.io/spring-boot/docs/1.5.9.RELEASE/reference/htmlsingle/#boot-features-external-config)**

* 1. **自动配置原理**

**配置原理:**

* + 1. **SpringBoot启动的时候加载主配置类，开启了自动配置功能 ==@EnableAutoConfiguration==**

@SpringBootApplication *//用来标注一个主程序类,说明这是一个Spring Boot应用*

@EnableAutoConfiguration

* + 1. **@EnableAutoConfiguration 作用：**

@Import(AutoConfigurationImportSelector.**class**)*//利用Selector给容器中导入一些组件？*

**可以查看selectImports()方法的内容；**

**public** String[] selectImports(AnnotationMetadata annotationMetadata) {  
 AutoConfigurationImportSelector.AutoConfigurationEntry autoConfigurationEntry = **this**.getAutoConfigurationEntry(autoConfigurationMetadata, annotationMetadata);   
 **return** StringUtils.toStringArray(autoConfigurationEntry.getConfigurations());  
 }  
}

**List<String> configurations = getCandidateConfigurations(annotationMetadata,attributes);获取候选的配置**

**protected** AutoConfigurationImportSelector.AutoConfigurationEntry getAutoConfigurationEntry(AutoConfigurationMetadata autoConfigurationMetadata, AnnotationMetadata annotationMetadata) {  
 List<String> configurations = **this**.getCandidateConfigurations(annotationMetadata, attributes);  
 **return new** AutoConfigurationImportSelector.AutoConfigurationEntry(configurations, exclusions);  
 }  
}

SpringFactoriesLoader.loadFactoryNames()

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

把扫描到的这些文件的内容包装成properties对象

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

**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;  
}

**org.springframework.boot.autoconfigure.web.HttpEncodingAutoConfiguration,\**

**org.springframework.boot.autoconfigure.web.HttpMessageConvertersAutoConfiguration,\**

**org.springframework.boot.autoconfigure.web.MultipartAutoConfiguration,\**

**org.springframework.boot.autoconfigure.web.ServerPropertiesAutoConfiguration,\**

**将类路径下META-INF/spring.factories 里面配置的所有EnableAutoConfiguration的值加入到了容器中,用他们来做自动配置；**

* + 1. **每一个自动配置类进行自动配置功能；**
    2. **以HttpEncodingAutoConfiguration（Http编码自动配置）为例解释自动配置原理；**

@Configuration *//表示这是一个配置类，以前编写的配置文件一样，也可以给容器中添加组件*@EnableConfigurationProperties(HttpProperties.**class**)*/\*启动指定类的ConfigurationProperties功能；*

*将配置文件中对应的值和HttpEncodingProperties绑定起来；并把HttpEncodingProperties加入到ioc容器中\*/*  
@ConditionalOnWebApplication(type = ConditionalOnWebApplication.Type.***SERVLET***)*/\*Spring底层@Conditional注解（Spring注解版），根据不同的条件，如果满足指定的条件，整个配置类里面的配置就会生效；判断当前应用是否是web应用，如果是，当前配置类生效\*/*@ConditionalOnClass(CharacterEncodingFilter.**class**) */\*判断当前项目有没有这个类CharacterEncodingFilter；SpringMVC中进行乱码解决的过滤器\*/*@ConditionalOnProperty(prefix = **"spring.http.encoding"**, value = **"enabled"**,  
 matchIfMissing = **true**)*/\*判断配置文件中是否存在某个配置 spring.http.encoding.enabled；如果不存在，判断也是成立的/即使我们配置文件中不配置pring.http.encoding.enabled=true，也是默认生效的；\*/***public class** HttpEncodingAutoConfiguration {  
  *//他已经和SpringBoot的配置文件映射了* **private final** HttpProperties.Encoding **properties**;  
  *//只有一个有参构造器的情况下，参数的值就会从容器中拿* **public** HttpEncodingAutoConfiguration(HttpProperties properties) {  
 **this**.**properties** = properties.getEncoding();  
 }  
  
 @Bean *//给容器中添加一个组件，这个组件的某些值需要从properties中获取* @ConditionalOnMissingBean *//判断容器没有这个组件？* **public** CharacterEncodingFilter characterEncodingFilter() {  
 CharacterEncodingFilter filter = **new** OrderedCharacterEncodingFilter();  
 filter.setEncoding(**this**.**properties**.getCharset().name());  
 filter.setForceRequestEncoding(**this**.**properties**.shouldForce(Type.***REQUEST***));  
 filter.setForceResponseEncoding(**this**.**properties**.shouldForce(Type.***RESPONSE***));  
 **return** filter;  
 }

根据当前不同的条件判断，决定这个配置类是否生效？

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

* + 1. **所有在配置文件中能配置的属性都是在xxxxProperties类中封装者‘；配置文件能配置什么就可以参照某个功能对应的这个属性类**

@ConfigurationProperties(prefix = **"spring.http"**)  
**public class** HttpProperties {**private final** Encoding **encoding** = **new** Encoding();  
 **public** Encoding getEncoding() {  
 **return this**.**encoding**;  
 }**public static class** Encoding {  
 **public static final** Charset ***DEFAULT\_CHARSET*** = StandardCharsets.***UTF\_8***;**private** Charset **charset** = ***DEFAULT\_CHARSET***;**private** Boolean **force**;**private** Boolean **forceRequest**;**private** Boolean **forceResponse**;

**精髓：**

**​ 1）、SpringBoot启动会加载大量的自动配置类**

**​ 2）、看我们需要的功能有没有SpringBoot默认写好的自动配置类；**

**​ 3）、再来看这个自动配置类中到底配置了哪些组件；（如果要用的组件有，就不需要再配置了）**

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

xxxxAutoConfigurartion：自动配置类；

给容器中添加组件

xxxxProperties:封装配置文件中相关属性；

**细节:**

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

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



自动配置类必须在一定的条件下才能生效；

我们怎么知道哪些自动配置类生效；

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

Positive matches:(自动配置启动的)

-----------------

DispatcherServletAutoConfiguration matched:

- @ConditionalOnClass found required class 'org.springframework.web.servlet.DispatcherServlet' (OnClassCondition)

- found 'session' scope (OnWebApplicationCondition)

-----------------

Negative matches:(没有启动,没有匹配成功的自动配置类)

-----------------

ActiveMQAutoConfiguration:

Did not match:

- @ConditionalOnClass did not find required class 'javax.jms.ConnectionFactory' (OnClassCondition)

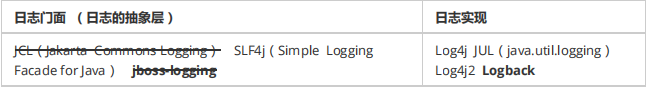
AopAutoConfiguration:

Did not match:

- @ConditionalOnClass did not find required class 'org.aspectj.lang.annotation.Aspect' (OnClassCondition)

1. **SpringBoot与日志(日志框架日志配置)**
   1. **日志框架**

市面上的日志框架: JUL、JCL、Jboss-logging、logback、log4j、log4j2、slf4j....



左边选一个门面（抽象层）、右边来选一个实现日志框架的搭建；

日志门面： SLF4J；

日志实现：Logback(log4j的升级版与slf4j出自同一人之手)；

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

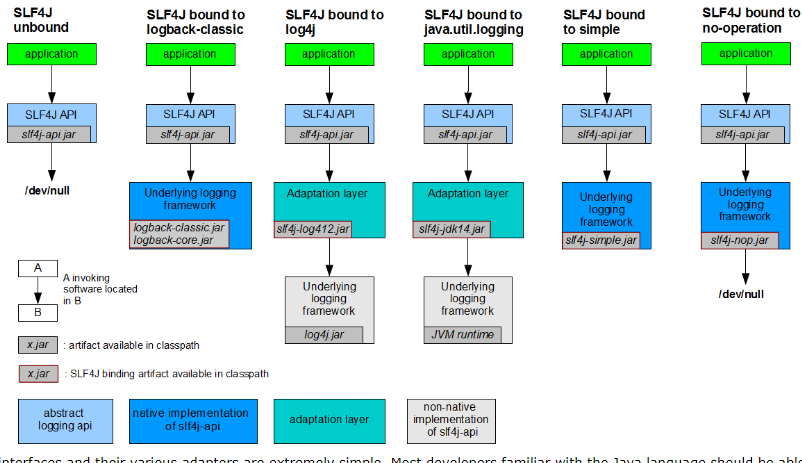
SpringBoot选用 SLF4j和logback；

* 1. **SLF4j使用**
     1. **如何在系统中使用SLF4j <https://www.slf4j.org>**

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(抽象层去调用具体实现)；

给系统里面导入slf4j的jar和 logback的实现jar

**import** org.slf4j.Logger;  
**import** org.slf4j.LoggerFactory;  
 **public class** HelloWorld {  
 **public static void** main(String[] args) {  
 Logger logger = LoggerFactory.*getLogger*(HelloWorld.**class**);  
 logger.info(**"Hello World"**);  
 }  
 }



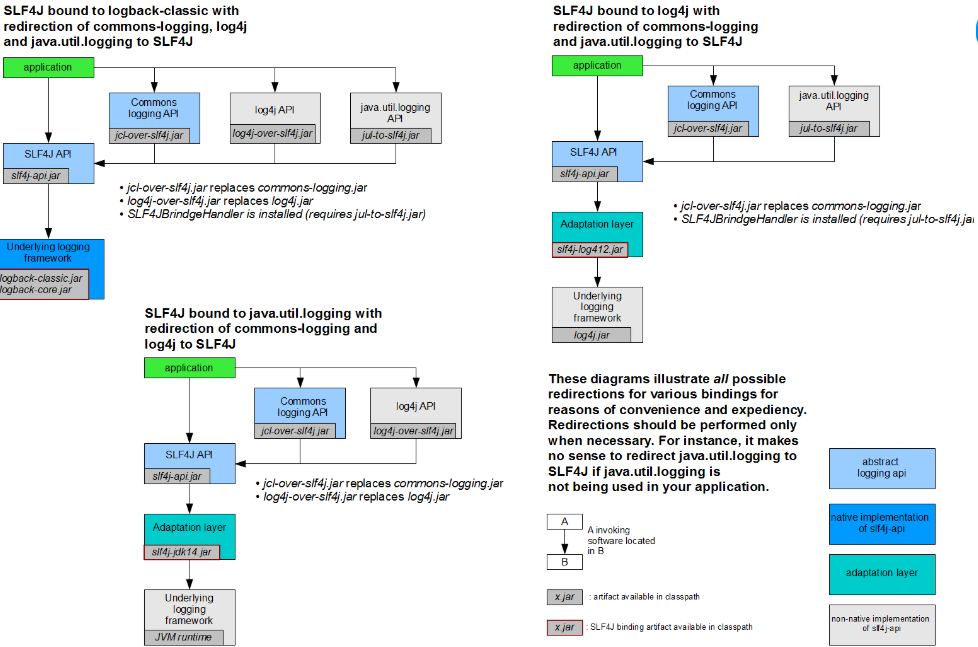
深蓝色框代表日志实现层(日志操作具体实现的类); 浅蓝色代表适配层(起到承上启下的作用适配抽象层和实现层);

每一个日志的实现框架都有自己的配置文件。使用slf4j以后，**配置文件还是使用日志 实现框架(名词) 本身的配置文件**；

* + 1. **遗留问题(其他日志框架同义转换为slf4j)**

a系统中使用（slf4j+logback）: 系统中的Spring（使用commons-logging日志框架）、Hibernate（使用log4j）、MyBatis(使用..)、xxxx

如何统一日志记录，让别的框架和我一起统一使用slf4j进行输出？



如何让系统中所有的日志都统一到slf4j；

1、将系统中其他日志框架先排除出去；

2、用中间包来替换原有的日志框架；

3、我们导入slf4j其他的实现

* + 1. **SpringBoot日志关系**

<dependency>

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

<artifactId>spring-boot-starter</artifactId>

</dependency>

SpringBoot使用它来做日志功能；

<dependency>

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

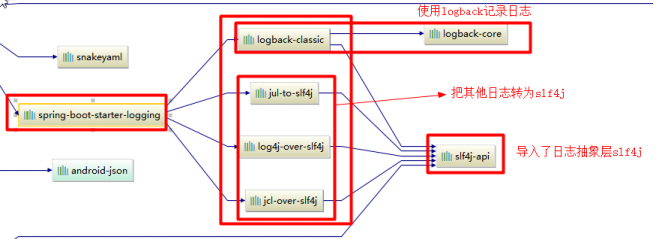
<artifactId>spring-boot-starter-logging</artifactId>

</dependency>

**idea查看底层一览关系图:**



**底层依赖关系:**



**总结：**

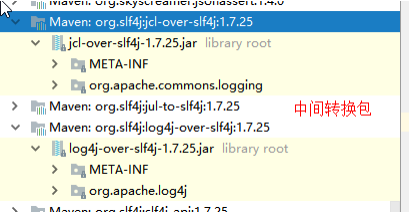
1）、SpringBoot底层也是使用slf4j+logback的方式进行日志记录

2）、SpringBoot也把其他的日志都替换成了slf4j；

3）、中间替换包？



这里在新建日志工厂的时候new的时slf4j的日志工厂也是中间包的精髓所在



4）、如果我们要引入其他框架？一定要把这个框架的默认日志依赖移除掉？

​ Spring框架用的是commons-logging；



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

* + 1. **日志使用**
       1. **默认配置**

SpringBoot默认帮我们配置好了日志；

**public class** Springboot01QuickApplicationTests {  
 Logger **logger** = LoggerFactory.*getLogger*(getClass());  
 @Test  
 **public void** contextLoads() {  
 *//日志的级别；  
 //由低到高 trace<debug<info<warn<error  
 //可以调整输出的日志级别；日志就只会在这个级别以以后的高级别生效* **logger**.trace(**"这是trace日志..."**);  
 **logger**.debug(**"这是debug日志..."**);  
 *//SpringBoot默认给我们使用的是info级别的，没有指定级别的就用SpringBoot默认规定的级别；root级别* **logger**.info(**"这是info日志..."**);  
 **logger**.warn(**"这是warn日志..."**);  
 **logger**.error(**"这是error日志..."**);}

日志输出格式：

%d表示日期时间，

%thread表示线程名，

%-5level：级别从左显示5个字符宽度

%logger{50} 表示logger名字最长50个字符，否则按照句点分割。

%msg：日志消息，

%n是换行符

-->

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

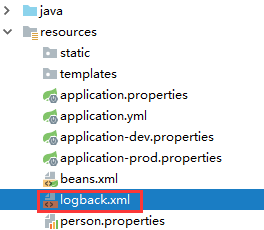
SpringBoot修改日志的默认配置

*#spring.profiles.active=dev***logging.level.com.atguigu**=**trace***# 可以指定完整的路径；不指定路径在当前项目下生成springboot.log日志***logging.file**=**G:/springboot.log***# 在当前磁盘的根路径下创建spring文件夹和里面的log文件夹；使用 spring.log 作为默认文件***logging.path**=**/spring/log***# 在控制台输出的日志的格式***logging.pattern.console**=**%d{yyyy-MM-dd} [%thread] %-5level %logger{50} - %msg%n***# 指定文件中日志输出的格式***logging.pattern.file**=**%d{yyyy-MM-dd} === [%thread] === %-5level === %logger{50} ==== %msg%n**

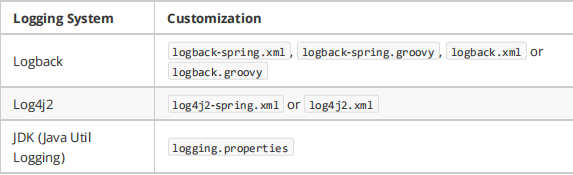


* + - 1. **指定配置**

给类路径下放上每个日志框架自己的配置文件即可；SpringBoot就不使用他默认配置的了



日志框架对应的配置文件名



logback.xml：直接就被日志框架识别了；

logback-spring.xml：日志框架就不直接加载日志的配置项，由SpringBoot解析日志配置，可以使用SpringBoot的高级Profile功能

<**appender name="stdout" class="ch.qos.logback.core.ConsoleAppender"**>  
<**layout class="ch.qos.logback.classic.PatternLayout"**>  
 <**springProfile name="dev"**> *<!--使用dev环境的配置文件生效-->*  
 <**pattern**>%d{yyyy-MM-dd HH:mm:ss.SSS} [%thread]==== %-5level %logger{50} - %msg%n</**pattern**>  
 </**springProfile**>  
 <**springProfile name="!dev"**> *<!--使用不是dev环境的配置文件生效-->*  
 <**pattern**>%d{yyyy-MM-dd HH:mm:ss.SSS} [%thread]---->%-5level %logger{50} - %msg%n</**pattern**>  
 </**springProfile**>  
 </**layout**>  
 </**appender**>

如果使用logback.xml作为日志配置文件，还要使用profile功能，会有以下错误no applicable action for [springProfile]

* + 1. **切换日志框架**

可以按照slf4j的日志适配图，进行相关的切换；

**slf4j+log4j的方式；**

<dependency>

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

<artifactId>spring-boot-starter-web</artifactId>

<exclusions>

<exclusion>

<artifactId>logback-classic</artifactId>

<groupId>ch.qos.logback</groupId>

</exclusion>

<exclusion>

<artifactId>log4j-over-slf4j</artifactId>

<groupId>org.slf4j</groupId>

</exclusion>

</exclusions>

</dependency>

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-log4j12</artifactId>

</dependency>

**切换为log4j2**

<dependency>

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

<artifactId>spring-boot-starter-web</artifactId>

<exclusions>

<exclusion>

<artifactId>spring-boot-starter-logging</artifactId>

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

</exclusion>

</exclusions>

</dependency>

<dependency>

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

<artifactId>spring-boot-starter-log4j2</artifactId>

</dependency>