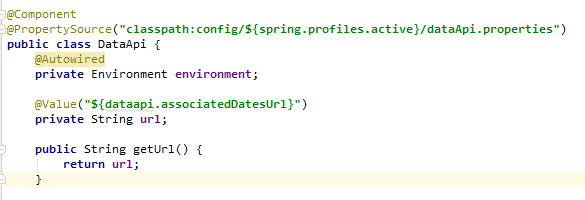
spring实例：

<http://git.oschina.net/didispace/SpringBoot-Learning>



**spring.profiles.active是环境变量，我们可以通过这种方式获得某个环境下的配置信息。**

1.

@Configuration 声明当前类是一个配置类,相当于一个Spring配置的xml文件 .

@ComponentScan 自动扫描包名下所有使用@Service,@[Component, @Repository](mailto:Component,@Repository) 和@Controller的类，并注册到Bean.

使用AnnotationConfigApplicationContext作为spring容器，接受输入一个配置类作为参数；构造函数传入一个@Configuration的配置类云构建IOC容器。

|  |
| --- |
| @Configuration  @ComponentScan("com....")  public class DiConfig{...} |
| public static void main(String[] args){         AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(DiConfig.class);  ....................  } |

JAVA配置是通过@Configuration 和 @Bean 来实现的。

@Bean 注解在方法上，声明当前方法的返回值为一个Bean.

可以不使用@Service ，可以在@Configuration的类中加了有返回值的@Bean标签来代替。所有的Bean都用这种方式的话，基本上就不需要@ComponentScan 了。

Spring AOP存在的目的就是为了解藕。AOP可以让一组类共享相同的行为。在OOP中只能通过继承类和实现接口，来使代码的耦合度增强，且类继承只能为单继承，阻碍更多行为添加到一组类上，AOP可以弥补了OOP的不足。

Spring支持AspectJ的注解式切面编程：

1.使用@Aspect声明是一个切面。

2.使用@After,@Before,@Around定义建言(advice)，可直接将拦截规则（切点）作为参数。

3.其中@After,@Before,@Around参数的拦截规则为切点(PointCut),为了使切点复用，可使用@PointCut专门定义拦截规则，然后在@After,@Before,@Around的参数中调用。

4.其中符合条件的第一个被拦截处为连接点(JoinPont)

在有@Configuration中的配置类上加入@EnableAspectJAutoProxy 注解开启Spring 对AspectJ的支持。

Scope描述的是Spring容器如何新建Bean的实例的。Spring的Scope有以下几种，通过@Scope注解来实现。

1.Singleton: 一个Spring容器中只有一个Bean的实例，此为Spring的默认配置，全容器共享一个实例。

2.Prototype: 每次调用新建一个Bean的实例。

3.Request： Web项目中，给每一个http request新建一个Bean实例。

4.Session: WEB项目中，给每个httpsession新建一个Bean实例。

5.GlobalSession:这个只在portal应用中有用，给每一个global http session新建一个Bean实例。

默认为Singleton,相当于@Scope("singleton")

Spring EL 和资源调用：

Spring EL-Spring表达式语文，支持在xml和注解中使用表达式，类似于jsp的EL表达式语言。

Spring 开发中经常涉及调用各种资源的情况，包含普通文件，网址，配置文件，系统环境变量等，我们可以使用spring的表达式语言实现资源的注入。Spring主要在注解@Value的参数中使用表达式。

 @Value("普通字符串")

private String another;

|  |
| --- |
| @Configuration  @ComponentScan("...")  @PropertySource("classpath:com/abc/test.properties") //注入配置文件  public  class ElConfig{     @Value("I love you") //注入普通字符串     private String normal;     @Value("#{systemProperties['os.name']}") //注入操作系统属性     private String osName;      @Value("#{ T(java.lang.Math).random() \* 100.0 }") //注入表达式结果     private double randomNumber;      @Value("classpath:com/xxx/xx.txt")   //注入文件资源     private Resource resource;      @Value("http://www.baidu.com") //注入网址资源      private Resource urlSource;      @Value("${book.name}");   //注入配置文件      private String bookName;  } |

注入配置配件需使用@PropertySource指定文件地址，若使用@Value注入，则要配置一个PropertySourcesPlaceholderConfigurer的Bean.

Bean的初始化和销毁：

1.Java配置文件：使用@Bean的initMethod和destroyMethod(相当于XML配置的init-method和destroy-method).

2.注解方式：利用jsr-250的@PostConstrut和@PreDestroy.

@PostConstruct 在构造函数执行完之后执行。

@PreDestroy 在Bean销毁之前执行。

|  |
| --- |
| @Bean(init-method="init",destroyMethod="destroy")  //构建初始化  BeanWayService beanWayService() {        return new BeanWayService();  }  initMethod 和 destroyMethod指定BeanWayService类的init和destroy方法在构造之后，Bean销毁之前执行。 |

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

Profile为在不同环境下使用不同的配置提供了支持，如开发环境和测试环境。

1.通过设定Environment的ActiveProfiles来设定当前context需要使用的配置环境。在开发中使用@Profile注解类或者方法，达到在不同情况下选择实例化不同的Bean.

2.通过设定jvm的spring.profiles.active参数来设置配置环境。

3.Web项目设置在Servlet的context parameter中。

|  |
| --- |
| //servlet 2.5  <Servlet>      <servlet-name>dispatcher</servlet-name>      <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>      <init-param>          <param-name>spring.profiles.active</param-name>         <param-value>production</param-value>       </init-param> |

|  |
| --- |
| Servlet 3.0+  public class WebInit implements WebApplicationInitializer {       public void onStartup(ServletContext context) throws ServletException {              context.setInitParameter("spring.profiles.default","dev");       }  } |

|  |
| --- |
| @Configuration  public class ProfileConfig {        @Bean        @Profile("dev")        public DemoBean devDemoBean(){             return new DemoBean();         }         @Bean         @Profile("qa")          public DemoBean qaDemoBean(){             return new DemoBean();         }  } |

Profile为dev时实例化devDemoBean. 为qa时实例化qaDemoBean.

|  |
| --- |
| 运行：  AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext();  context.getEnvironment().setActiveProfiles("dev"); //先将活动的Profile设置为dev  context.register(ProfileConfig.class); //后置注册Bean配置类，不然会报Bean未定义的错  context.refresh(); //刷新容器 |

事件：

Spring的事件(Application Event)为Bean与Bean之间的消息通信提供了支持，当一个Bean处理完一个任务之后，希望另一个Bean知道并能做相应的处理，这时我们就需要让另外一个Bean监听当前Bean所发送的事件。

Spring的事件需要遵循如下流程：

1.自定义事件，集成ApplicationEvent.

2.定义事件监听器，实现ApplicationListener.

3.使用容器发布事件。

|  |
| --- |
| 什么是ApplicationContext?  它是Spring的核心，Context我们通常解释为上下文环境，但是理解成容器会更好些。  ApplicationContext则是应用的容器，用来发布事件。  Spring把Bean（object）放在容器中，需要用就通过get方法取出来。  ApplicationEvent  是个抽象类，里面只有一个构造函数和一个长整型的timestamp。  ApplicationListener  是一个接口，并指定监听的事件类型，里面只有一个onApplicationEvent方法对消息进行接受处理。  所以自己的类在实现该接口的时候，要实装该方法。  如果在上下文中部署一个实现了ApplicationListener接口的bean,  那么每当在一个ApplicationEvent发布到 ApplicationContext时，  这个bean得到通知。其实这就是标准的Oberver设计模式。  下面给出例子：  首先创建一个ApplicationEvent实现类：  **[java]** [view plain](http://blog.csdn.net/u012635819/article/details/50131695#) [copy](http://blog.csdn.net/u012635819/article/details/50131695#)  **import** org.springframework.context.ApplicationEvent;    **public** **class** EmailEvent **extends** ApplicationEvent {      /\*\*       \* <p>Description：</p>       \*/  **private** **static** **final** **long** serialVersionUID = 1L;  **public** String address;  **public** String text;    **public** EmailEvent(Object source) {  **super**(source);      }    **public** EmailEvent(Object source, String address, String text) {  **super**(source);  **this**.address = address;  **this**.text = text;      }    **public** **void** print(){          System.out.println("hello spring event!");      }    }  给出监听器：  **[java]** [view plain](http://blog.csdn.net/u012635819/article/details/50131695#) [copy](http://blog.csdn.net/u012635819/article/details/50131695#)  **import** org.springframework.context.ApplicationEvent;  **import** org.springframework.context.ApplicationListener;    **public** **class** EmailListener **implements** ApplicationListener {    **public** **void** onApplicationEvent(ApplicationEvent  event) {  **if**(event **instanceof** EmailEvent){              EmailEvent emailEvent = (EmailEvent)event;              emailEvent.print();              System.out.println("the source is:"+emailEvent.getSource());              System.out.println("the address is:"+emailEvent.address);              System.out.println("the email's context is:"+emailEvent.text);          }        }    }  **[java]** [view plain](http://blog.csdn.net/u012635819/article/details/50131695#) [copy](http://blog.csdn.net/u012635819/article/details/50131695#)  applicationContext.xml文件配置：  <bean id="emailListener" **class**="com.spring.event.EmailListener"></bean>  测试类：  **import** org.springframework.context.ApplicationContext;  **import** org.springframework.context.support.ClassPathXmlApplicationContext;  **public** **class** Test {  **public** **static** **void** main(String[] args) {          ApplicationContext context = **new** ClassPathXmlApplicationContext("classpath:applicationContext.xml");            //HelloBean hello = (HelloBean) context.getBean("helloBean");          //hello.setApplicationContext(context);          EmailEvent event = **new** EmailEvent("hello","boylmx@163.com","this is a email text!");          context.publishEvent(event);          //System.out.println();      }  }  测试结果：  hello spring event!  the source is:hello  the address is:boylmx@163.com  the email's context is:this is a email text! |

Spring Aware

spring提供的Aware接口

|  |  |
| --- | --- |
| BeanNameAware | 获得到容器中的Bean的名称 |
| BeanFactoryAware | 获得当前bean factory，这样可以调用容器的服务 |
| ApplicationContextAware | 当前的application context,这样可以调用容器的服务 |
| MessageSourceAware | 获得message source,这样可以获得广本信息 |
| ApplicationEventPublisherAware | 应用时间发布器，可以发布事件 |
| ResourceLoaderAware | 获得资源加载器，可以获得外部资源文件 |

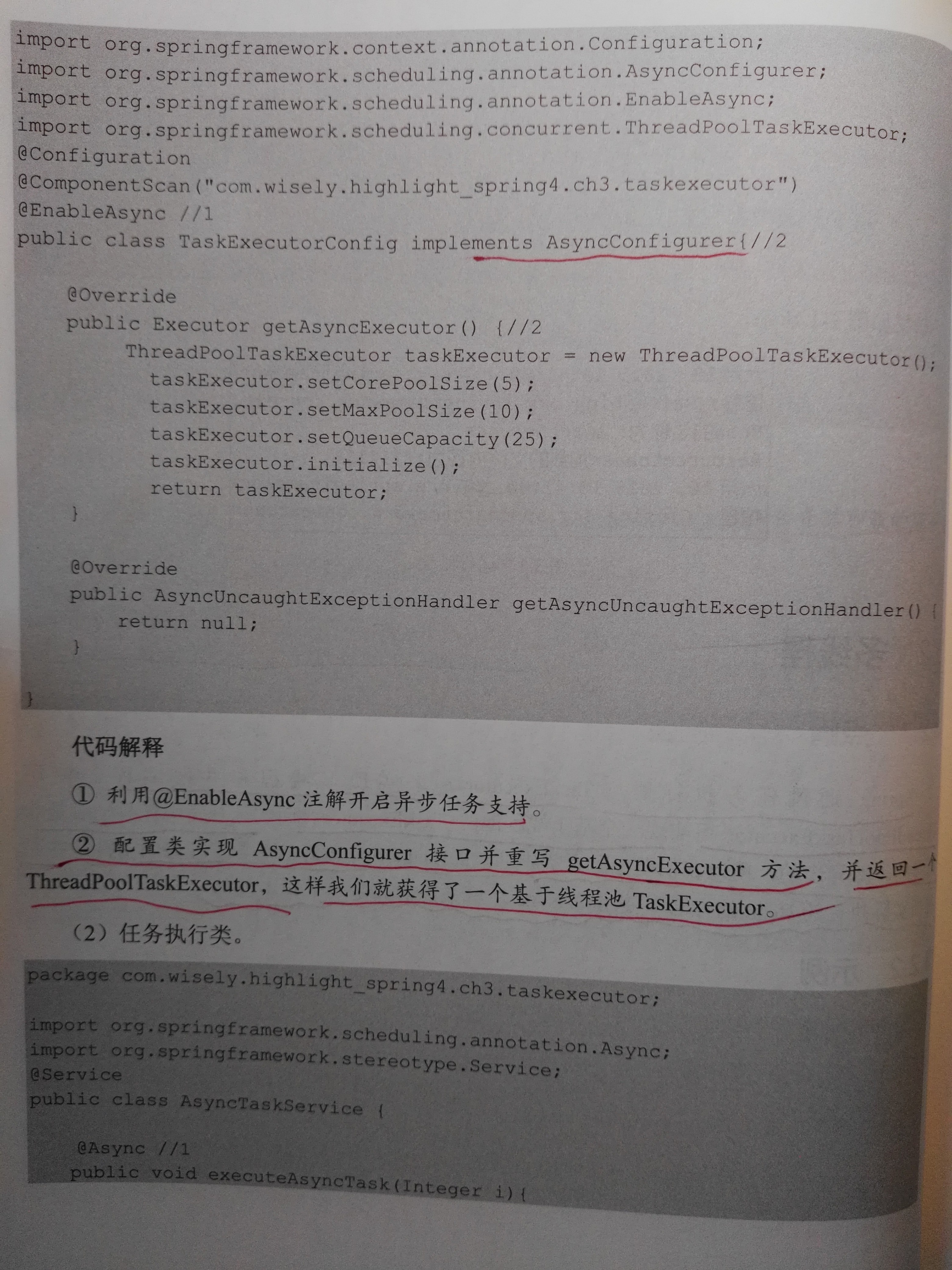
Spring Aware的目的是为了让Bean获得Spring容器的服务。因为ApplicationContext接口集成了MessageSource接口，ApplicationEventPublisher接口和ResourceLoader接口，所以Bean继承ApplicationContextAware可以获得Spring容器的所有服务，但原则上我们用到什么 接口就用什么 接口。

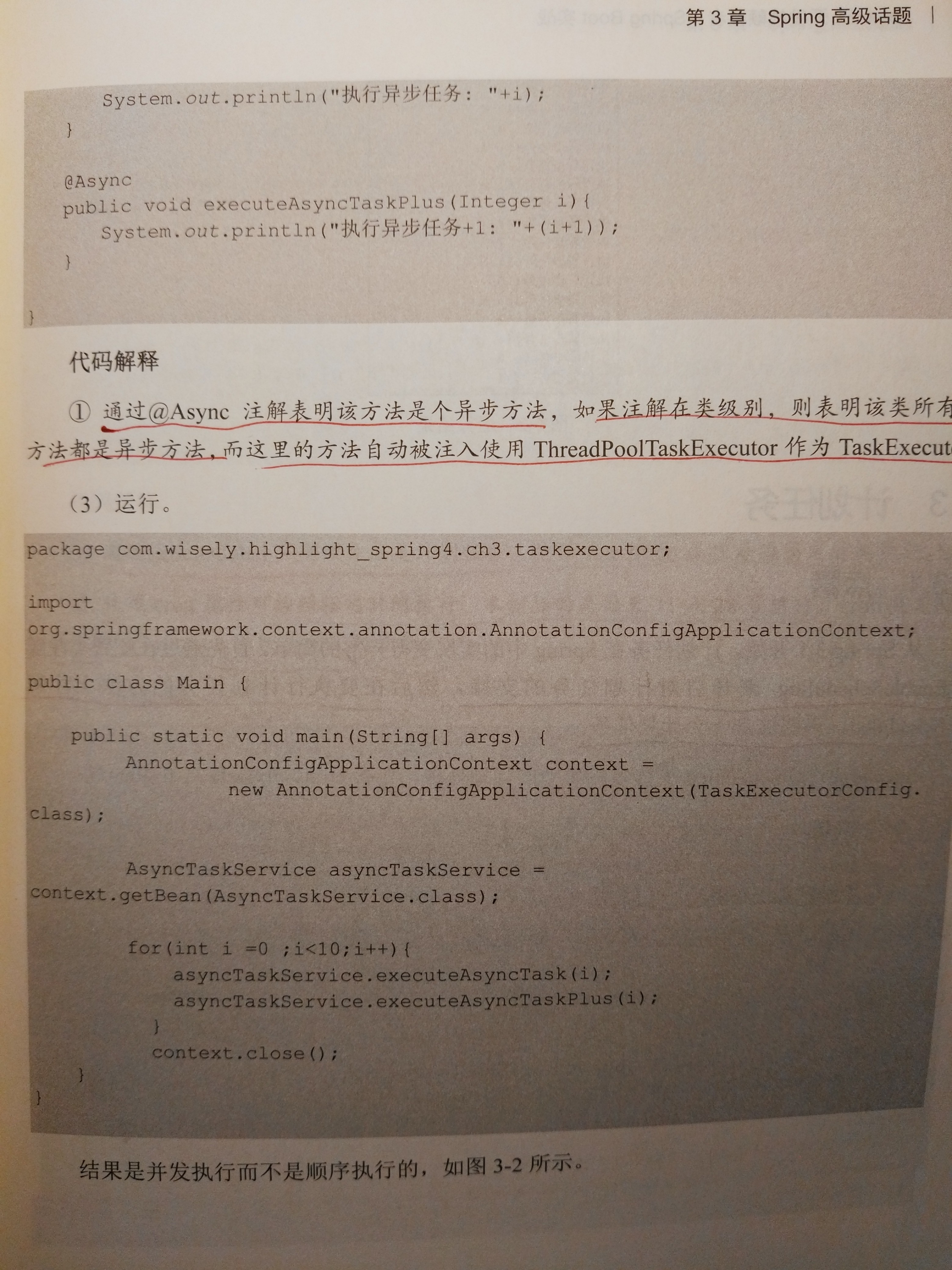
实现ResourceLoaderAware需要重写setResourceLoader;

实现BeanNameAware需要重写setBeanName方法。

多线程：

 Spring 通过任务执行器（TaskExecutor)来实现多线程和并发编程。使用ThreadPoolTaskExecutor可实现一个基于线程池的TaskExecutor.而实际开发中任务一般是非阻碍的，即异步的，所以我们要在配置类中通过@EnableAsync开启对异步任务的技持，并通过在实际执行的Bean的方法中使用@Async注解来声明其是一个异步任务 .





计划任务：

可以通过配置类注解@EnableScheduling来开启对计划任务的支持，然后在执行计划任务的方法上注解@Scheduled，声明这是一个计划任务。

|  |
| --- |
| @Service  public class ScheduledTaskService {        @Scheduled(fixedRate = 5000)        public void reportCurrentTime() {              System.out.println("每隔5秒执行一次");        }        @Scheduled(cron = "0 28 11 ? \* \* ") //使用corn属性可按照指定时间执行，每天11点28分执行，cron是UNIX和LINUX下的定时任务         public void fixTimeExecution(){.....}  }  配置类：  @Configuration  @ComponentScan("...")  @EnableScheduling //通过@EnableScheduling注解开启计划任务的支持  public class Xx(){....} |

条件注解：

基于条件的Bean的创建，即使用@Conditional注解。

@Conditional根据满足某一个特定的条件创建一个特定的Bean。比方说，当某一个jar包在一个类路径下的时候，自动配置一个或多个Bean；或者只有某个Bean被创建才会创建另外一个Bean，总的来说，就是根据特定条件来控制Bean的创建行为，这样我们可以利用这个特征进行一些自动的配置。

|  |
| --- |
| public class WindowsCondition implements Condition{         public boolean matches(ConditionContext context){           context.getEnvironment().getProperty("os.name").contains("Windows");         }  }  public class LinuxCondition implements Condition{}  public interface ListService{}  public class WindowListService implements ListService {}  public class LinuxListService implements ListService {}  @Configuration  public class ConditionConfig{       @Bean       @Conditional(WindowsCondition.class)       public ListService windowsListService() {                return new WindowsListService();       }       @Bean       @Conditional(LinuxCondition.class)       public ListService linuxListService() {                return new LinuxListService();       }  } |

组合注解：

有时一些常用的注解需要多个，现在我们用一个注解把这个都合并在一个注解里，这样以后使用的时候就只需要使用一个注解就可以了。

|  |
| --- |
| @Target(ElementType.TYPE)  @Retention(RetentionPolicy.RUNTIME)  @Documented  @Configuration     //组合@Configuration元注解  @ComponentScan   //组合@ComponentScan元注解  public @interface WiselyConfiguration {        String[] value() default {} ;// 覆盖value参数  } |

使用@WiselyConfiguration 组合注解替代@Configuration和@ComponentScan.

@Enable\*注解的工作原理

@EnableAspectJAutoProxy开启对AspectJ自动代理的支持

@EnableAsync开启异步方法的支持

@EnableScheduling开启计划任务的支持

@EnableWebMvc开启web mvc的配置支持

@EnableConfigurationProperties 开启对@ConfigurationProperties注解配置Bean的支持

@EnableJpaRepositories开启对Spring Data JPA Repository的支持

@EnableTransactionManagement开启注解式事务的支持

@EnableCaching开启注解式的缓存支持。

使用@Enable\*来开启一项功能的支持，从而避免自己配置大量的代码，大大降低使用难度。通过观察这些@Enable\* 注解的源码，我们发现所有的注解都有一个@Import注解，@Import是用来导入配置类的，这也就浊意味着这些自动开启的实现其实是导入了一些自动配置的Bean.

Spring Mvc里实现WebApplicationInitializer接口便可实现等同于web.xml的配置。

@EnableWebMvc注解会开启一些默认配置，如一些ViewResolver或者MessageConverter等。

@ResponseBody 支持将返回值放在response体中，而不是返回一个页面。一般用于ajax上。

@RequestBody允许request的参数在request体中，而不是直接链接在地址后面。此注解放置在参数前。

@RestController 是一个组合注解，组合了@Controller和@ResponseBody,这就意味着当你只开发一个和页面交互数据的控制的时候，需要使用此注解。若没有此注解，要想实现上述功能，则需自己在代码中加@Controller和＠ResponseBody两个注解。

Spring MVC的定制配置需要我们的配置类继承一个WebMvcConfigureAdapter类，并在此类使用@EnableWebMvc注解，来开启对Spring Mvc的配置支持，这样我们就可以重写这个类的方法，完成我们的常用配置。

程序的静态文件(js,css,图片)等需要直接访问，这时我们可以在配置里重写addResourceHandlers方法来实现。

|  |
| --- |
| @Configuration  @EnableWebMvc   //开启Spring mvc支持，若无此句，重写WebMvcConfigurerAdapter  @ComponentScan("com....")  public class MyMvcConfig extends WebMvcConfigureAdapter {   //继承WebMvcConfigurerAdapter类，重写其方法可对Spring MVC 进行配置    @Bean    public InternalResourceViewResolver viewResolver(){          InternalResourceViewResolver viewResolver = new InternalResourceViewResolver();          viewResolver.setPrefix("/WEB-INF/classes/views/");          viewResolver.setSuffix(".jsp");          viewResolver.setViewClass(JstlView.class);          return viewResolver;     }    public void addResourceHandlers(ResourceHandlerRegistry registry) {        registry.addResourceHandler("/assets/\*\*").addResourceLocations("classpath:/assets/");      //addResourceLocations 指的是文件放置的目录，addResourceHandler指的是对外暴露的访问路径     }  } |

拦截器配置：

拦截器Interceptor实现对每一个请求处理前后进行相关的业务处理，类似于Servlet的Filter.

可以让普通的Bean实现HanlderInterceptor接口或者继承HandlerInterceptorAdapter类来实现自定义拦截器。

通过重写WebMvcConfigureAdapter的addInterceptors方法来注册自定义的拦截器。

重写preHandle方法，在请求发生前执行；重写postHandle方法，在请求完成后执行。

@ControllerAdvice

通过@ControllerAdvice　我们可以将对于控制器的全局配置放置在同一个位置，注解了@Controller的类的方法可使用@ExceptionHandler,@InitBinder,@ModelAttribute注解到方法上，这对所有注解了@RequestMapping的控制器内的方法有效。

@ExceptionHnadler：用于全局处理控制器里的异常。

@InitBinder：用来设置WebDataBinder,WebDataBinder用来自动绑定前台请求参数到Model中。

@ModelAttribute:@ModelAttribute本来的作用是绑定键值对到Model里，此处是让全局的@RequestMapping都能获得在此处设置的键值对。

|  |
| --- |
| @ControllerAdvice  public class ExceptionHandlerAdvice{      @ExceptionHandler(value = Exception.class)      public ModalAndView exception(Exception exception, WebRequest request) {...}         @ModelAttribute       public void addAttributes(Model model){           model.addAttribute("msg",".....");       }       @InitBinder       public void initBinder(WebDataBinder webDataBinder) {              webDataBinder.setDisallowedFields("id"); //忽略id,控制台id为变成null       }  }  //使用  @RequestMapping("advices") //从model中读取msg  public String getSomething(@ModelAttribute("msg") String msg) {  } |

对于无业务逻辑的页面转向，我们可以在配置时使用：

|  |
| --- |
| public void addViewControllers(ViewControllerRegistry registry) {      registry.addViewController("/index").setViewName("/index");  }  //在spring mvc中,路径参数如果带"."的话，"."后面的值将被忽略,如<http://..../xx.yy>  通过重写configurePathMatch方法可以不忽略"."后面的参数，代码如下  @Override  public void configurePathMatch(PathMatchConfigurer configurer){      configurer.setUseSuffixPatternMatch(false);  }  更多配置请查看WebMvcConfigurerAdapter类的Api; |

自定义HttpMessageConverter

HttpMessageConverter是用来处理request和response里的数据的。Spring为我们内置了大量的HttpMessageConverter,如MappingJackson2HttpMessageConverter,StringHttpMessageConverter等，我们可以实现自定义的HttpMessageConverter,并注册(**extendMessageConverters**)这个HttpMessageConverter到Spring MVC.

|  |
| --- |
| public class MyMessageConverter extends AbstractHttpMessageConverter<DemoObj> {    } |

服务端推送技术方案：

当客户端向服务端发送请求，服务端会抓住这个请求不放，等有数据更新的时候才返回给客户端，当客户端接收到消息后，再向服务端发送请求，周而复始。这种方式的好处是减少了服务器的请求数量，大减少了服务器的压力。推送的媒体类型为：text/event-stream;

我们也可以使用websocket

在Spring mvc里实现WebApplicationInitializer接口便可实现等于同于web.xml的配置 。

Spring mvc增加静态资源，

|  |
| --- |
| @Configuration  @EnableWebMvc  @ComponentScan("")  public class MyMvcConfig extends WebMvcConfigurerAdapter {        @Bean        public InternalResourceViewResolver viewResolver(){           ......         }         public void addResourceHandlers(ResourceHandlerRegistry registry){               registry.addResourceHandler("/assets/\*\*").addResourceLocations("classpath:/assets/");//addResourceLocation指是的文件放置的目录，addResourceHandler指的是对外暴露的访问路径。         }  } |

spring mvc拦截器需要继承HandlerInterceptorAdapter 实现preHandle和postHandle方法 。preHandle发生在请求前，postHandle发生在请求后。

配置拦截器Bean,重写addInterceptors方法，注册拦截器。

配置拦截器到容器：

@Bean

public DemoInterceptor demoInterceptor(){

     return new DemoInterceptor();

}

public void addInterceptors(InterceptorRegistry registry){

     registry.addInterceptor(demoInterceptor());

}

Spring Boot会自动扫描@SpringBootApplication所在类的同级包以及下级包里的Bean,（若为JPA项目还可以扫描标注@Entity的实体类）。建议入口类放置的位置在groupid + artifactID组合的包名下。

Main执行：

SpringApplication app = new SpringApplication(xxx.class);

app.run(args);

spring boot 使用一个全局的配置文件application.properties或application.yml,放置在src/main/resources目录或者类路径的/config 下。

在application.properties中:

server.port=9090 #设置端口

server.context-path=/hello #设置默认的访问路径

Spring Boot提倡零配置，既无xml配置,但是在实际项目中，可能有一些特殊要求你必须使用xml配置，这时我们可以通过Spring提供的@ImportResource来加载xml配置如：

@ImportResource({"classpath:xxx.xml","classpath:aaa.xml"})

可以将spring boot项目打成jar:

#java -jar xx.jar

#java -jar xx.jar --server.port=9090

在Spring环境下,注入properties文件里的值的方式，通过@PropertySource指明properties文件的位置，然后通过@Value注入值。在Spring boot里，我们只需要在application.properties定义属性，直接使用@Value注入即可。

类安全的配置：

Spring boot还提供了基于类型安全的配置方式，通过@ConfigurationProperties将properties属性和一个

Bean及其属性关联，从而实现类型安全的配置 。

|  |
| --- |
| xxx.properties:  author.name=ccc  author.age=33 |

|  |
| --- |
| @Component  @ConfigurationProperties(locations="classpath:xxx.properties", prefix="author")  public class Author{      private String name;      private int age;  }  通过@ConfigurationProperties加载properties文件内的配置，通过prefix属性指定properties的配置的前缀，通过locations指定properties文件的位置。 |

Profile配置 ：

profile是spring用来针对不同环境对不同的配置提供支持的，全局Profile配置 使用application-{profile}.properties.通过在application.properties中设置spring.profiles.active=prod来指定活动的Profile.

可以为不同的环境设置不的端口。可以在application.properties中指定spring.profiles.active=dev.

如果想知道spring为我们做了哪些自动配置 ，可以加入debug.

1.运行jar时增加--debug参数： java -jar xx.jar --debug

2.在application.properties中设置属性： debug=true

SpringBoot原理，其实它会扫描jar包里META-INF/spring.factories文件里的jar包,里面有很多自动配置类的路径。

自己写一个spring boot插件：

|  |
| --- |
| @ConfigurationProperties(prefix="hello")  public class HelloServiceProperties{      .....  }  public class HelloService {     ....  }  //自动配置类  @Configuration  @EnableConfigurationProperties(HelloServiceProperties.class)  @ConditionalOnClass(HelloService.class)  @ConditionalOnProperty(perfix="hello",value="enabled",matchIfMissing=true)  public class HelloServiceAutoConfiguration{     @Autowired     private HelloServiceProperties helloServiceProperties;        @Bean      @ConditionOnMissingBean(HelloService.class)       public HelloService helloService() {           .......       }  }  //根据HelloServiceProperties提供的参数，并通过@ConditionalOnClass判断HelloService这个类在类路径中是否存在，且当容器中没有这个Bean的情况下自动配置这个Bean.  //注册配置：若想自动配置生效，需要注册自动配置类。在src/main/resources下新建META-INF/spring.factories.在里面填写注册内容：  org.springframework.boot.autoconfigure.EnableAutoConfiguration=\  com.xxx.xxx.HelloServiceAutoConfiguration  若有多个自动配置，则用","隔开，些处"\"是为了换行后仍然能读到属性。 |

自动配置静态资源：

在自动配置类的addResourceHandlers方法中定义了以下静态资源的自动配置。

1.类路径文件

/static,/public,/resources和/META-INF/resources文件夹下的静态文件直接映射为/\*\*，可以通过http://localhost:8080/\*\*来访问。

2.webjar

把webjar的/META-INF/resources/webjars/下的静态文件映射为/webjar/\*\*,可能通过http://localhost:8080/webjar/\*\*访问。

当使用嵌入式的Servlet容器(Tomcat,jetty)时，我们通过将servlet,filter和listener声明为Spring bean而达到注册的效果；或者注册ServletRegistrationBean,FilterRegistrationBean 和ServletListenerRegistrationBean的Bean.

public XServlet extend HttpServlet{}

@Bean

public XServlet xServlet(){

       return new XServlet();

}

Spring boot 配置Tomcat

如果要自定义Tomcat的配置，那么就要实现EmbeddedServletContainerCustomizer

或者使用TomcatEmbededServletContainerFactory

Spring Boot 配置SSL

1.生成证书

使用Jre里的工具keytool用来生成证书，#keytool -genkey -alias tomcat ,会生成一个.keystore文件，这就是证书文件。将.keystore文件复制到项目的根目录，然后在application.properties中做如下ssl配置：

server.port = 8443

server.ssl.key-store = .keystore

server.ssl.key-store-password = 111111

server.ssl.keyStoreType = JKS

server.ssl.keyAlias: tomcat

<http://localhost:8443>

要实现这个功能，需要配置TomcatEmbeddedServletContainerFactory

<http://www.jianshu.com/p/c3879add87ec>67

install apache and configure ssl

<https://www.digitalocean.com/community/tutorials/how-to-create-an-ssl-certificate-on-apache-for-centos-7>

<https://www.server-world.info/en/note?os=CentOS_7&p=httpd&f=6>

spring boot mvc 配置InternalviewResolver 的配置可以在application.properties中配置了：

spring.mvc.view.prefix: /WEB-INF/jsp/
spring.mvc.view.suffix: .jsp
welcome.message: Hello Mkyong

<https://www.mkyong.com/spring-boot/spring-boot-hello-world-example-jsp/>

如果想看数组中是否包含某一个值，可以先将数组转换成List,再用List中的conatins()方法查看。

java8中也可以这么做：

// Convert to stream and test it ， alphabet is a array

boolean result = Arrays.stream(alphabet).anyMatch("A"::equals);

if (result) {

 System.out.println("Hello A");

}

查看数字数组里是否包含4：

int[] number ={1,2,3,4,5,6,7,8,9,10};

boolean result = IntStream.of(number).anyMatch(x -> x ==4);

java覆盖equals和hashCode可以使用Objects.equals()和Objects.hash()方法，还可以使用HashCodeBuilder

<http://www.kode12.com/>

Spring boot 压缩 response

在application.properties中 我们可以设置一些配置去压缩配置文件和response

server.compression.enabled=true | false #压缩response

server.compression.enabled=true

server.compression.mime-types = application/json

spring boot版本1.4以上用以上2个配置

server.compression.excluded-user-agents= #哪些user-agent不压缩，比如**Example Values:** Mozilla/5.0 (Windows NT 6.3; WOW64; rv:49.0) Gecko/20100101 Firefox/49.0

server.compression.mime-types=

#哪些mime-type需要压缩，用逗号隔开，如**Example values:**application/json,application/xml,text/html,text/xml,text/plain,text/css,application/javascript,application/json

server.compression.min-response-size=2048 #如果大于2048就压缩

通过这些配置你会在header里发现Content-Encoding-gzip 通过这些配置，你根本不需要在maven,gardle里去配置一些压缩的插件了。

spring boot 配置多数据库，有可能是多个不同的数据库mysql, sqlserver,postgresdb

在application.properties 中配置数据库的配置：

|  |
| --- |
| 其它类型的数据库请查看application.properties中属性 |

|  |  |
| --- | --- |
| 3  4  5  6  7  8  9  10  11 | server.port=6060    spring.ds\_post.url=jdbc:postgresql://localhost:5432/kode12  spring.ds\_post.username=postgres  spring.ds\_post.password=root  spring.ds\_post.driverClassName=org.postgresql.Driver    spring.ds\_mysql.url=jdbc:mysql://localhost:3306/kode12  spring.ds\_mysql.username=root  spring.ds\_mysql.password=root  spring.ds\_mysql.driverClassName=com.mysql.jdbc.Driver |

增加Bean配置

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38 | packagecom.kode12.config;    importjavax.sql.DataSource;    importorg.springframework.beans.factory.annotation.Qualifier;  importorg.springframework.boot.autoconfigure.jdbc.DataSourceBuilder;  importorg.springframework.boot.context.properties.ConfigurationProperties;  importorg.springframework.context.annotation.Bean;  importorg.springframework.context.annotation.Configuration;  importorg.springframework.context.annotation.Primary;  importorg.springframework.jdbc.core.JdbcTemplate;    *@Configuration*  publicclassMultipleDBConfig{    *@Bean*(name="mysqlDb")  *@ConfigurationProperties*(prefix="spring.ds\_mysql")  publicDataSource oracleDataSource(){  returnDataSourceBuilder.create().build();  }    *@Bean*(name="mysqlJdbcTemplete")  publicJdbcTemplate jdbcTemplate(*@Qualifier*("mysqlDb")DataSource dsMySQL){  returnnewJdbcTemplate(dsMySQL);  }  *@Bean*(name="postgresDb")  *@ConfigurationProperties*(prefix="spring.ds\_post")  publicDataSource postgresDataSource(){  return  DataSourceBuilder.create().build();  }    *@Bean*(name="postgresJdbcTemplete")  publicJdbcTemplate postgresJdbcTemplate(*@Qualifier*("postgresDb")DataSource dsPostgres){  returnnewJdbcTemplate(dsPostgres);  }  } |

然后在Controller中使用

*@Autowired*

*@Qualifier*("postgresJdbcTemplete")

privateJdbcTemplate postgresTemplate;

*@Autowired*

*@Qualifier*("mysqlJdbcTemplete")

privateJdbcTemplate mysqlTemplate;

|  |
| --- |
|  |

spring boot 单元测试

**如何做单元测试**

打开的src/test/下的测试入口，编写简单的http请求来测试；使用mockmvc进行，利用MockMvcResultHandlers.print()打印出执行结果。

@RunWith(SpringJUnit4ClassRunner.class)@SpringApplicationConfiguration(classes=MockServletContext.class)@WebAppConfigurationpublicclassHelloWorldControlerTests{privateMockMvcmvc;@BeforepublicvoidsetUp()throwsException{mvc=MockMvcBuilders.standaloneSetup(newHelloWorldController()).build();}@TestpublicvoidgetHello()throwsException{mvc.perform(MockMvcRequestBuilders.get("/hello").accept(MediaType.APPLICATION\_JSON)).andExpect(MockMvcResultMatchers.status().isOk()).andDo(MockMvcResultHandlers.print()).andReturn();}}

**log配置**

配置输出的地址和输出级别

logging.path=/user/local/log
logging.level.com.favorites=DEBUG
logging.level.org.springframework.web=INFO
logging.level.org.hibernate=ERROR

path为本机的log地址，logging.level 后面可以根据包路径配置不同资源的log级别

**共享Session-spring-session-data-redis**

分布式系统中，sessiong共享有很多的解决方案，其中托管到缓存中应该是最常用的方案之一，

**Spring Session官方说明**

Spring Session provides an API and implementations for managing a user’s session information.

**如何使用**

1、引入依赖

2、Session配置：

@Configuration
@EnableRedisHttpSession(maxInactiveIntervalInSeconds = 86400\*30)
public class SessionConfig {
}

maxInactiveIntervalInSeconds: 设置Session失效时间，使用Redis Session之后，原Boot的server.session.timeout属性不再生效

好了，这样就配置好了，我们来测试一下

3、测试

添加测试方法获取sessionid

@RequestMapping("/uid")
String uid(HttpSession session) {
UUID uid = (UUID) session.getAttribute("uid");
if (uid == null) {
uid = UUID.randomUUID();
}
session.setAttribute("uid", uid);
return session.getId();
}

登录redis 输入 keys '\*sessions\*' xml t<spring:session:sessions:db031986-8ecc-48d6-b471-b137a3ed6bc4 t(spring:session:expirations:1472976480000 其中 1472976480000为失效时间，意思是这个时间后session失效，db031986-8ecc-48d6-b471-b137a3ed6bc4 为sessionId,登录[http://localhost:8080/uid](https://link.zhihu.com/?target=http%3A//localhost%3A8080/uid) 发现会一致，就说明session 已经在redis里面进行有效的管理了。

**如何在两台或者多台中共享session**

其实就是按照上面的步骤在另一个项目中再次配置一次，启动后自动就进行了session共享。

[**示例代码地址**](https://link.zhihu.com/?target=https%3A//github.com/ityouknow/spring-boot-starter/tree/master/spring-boot-web)