**Springboot概述**

**1.Spring Framework**

**1.什么是Spring？**

1.)从微观上来说，Spring指spring-framework，是开源的轻量级的一站式JAVAEE框架。Spring在JAVAEE三层结构中分别提供了不同的解决技术，web提供了SpringMvc，Service提供了IOC和AOP, 数据访问层提供了JdbcTemplate；

2.)从宏观上来说，Spring指的是Spring构建的整个Spring生态圈。Spring的整个生态圈覆盖了web开发、数据访问、安全控制、分布式、消息服务、移动开发、批处理等领域都提供了对应的解决方案。

SpringBoot:可以快速构建Spring应用。

SpringCloud：提供了一套目前完整微服务解决框架，是分布式整体解决方案；

SpringSecurity：为应用安全问题提供了对应的解决方案。

SpringSession:为分布式场景下session共享的问题提供了对应的解决方案。

Spring Data:提供了一整套数据访问层(DAO)的解决方案。

**2.Spring能做什么？**

1.微服务开发(Microservices)、响应式编程(Reactive)、分布式云开发(Cloud)、Web开发(Web Apps)、函数式服务(Serverless)、事件驱动(Event Driven)、批处理(Batch)....

**3.Spring Framework框架有什么优缺点？**

1.提供了IOC、AOP、事务功能等功能，容易和其他主流的框架整合，方便解耦，简化开发，降低企业应用开发的复杂度。

2.传统项目中整合SSH或SSM框架时，需要编写大量的配置文件，考虑jar包的冲突,整合第三方技术集成难度大，开发效率低；

**2.SpringMVC**

**1.什么是MVC？**

MVC是一种设计模式。Model（模型） - View（视图） - Controller（控制器）三层架构的设计模式，用于实现前端页面的展现与后端业务数据处理的分离。

**2.mvc设计模式的好处**

1.分层设计，实现了业务系统各个组件之间的解耦，有利于业务系统的可扩展性，可维护性。

2.有利于系统的并行开发，提升开发效率。

**3.什么是SpringMVC？**

SpringMVC是spring-framework中的一个模块，是一个基于Java实现了MVC设计模式的请求驱动类型的轻量级Web框架，通过把Model，View，Controller分离，将web层进行职责解耦，把复杂的web应用分成逻辑清晰的几部分，简化开发，减少出错，方便组内开发人员之间的配合。

**2.Springboot**

**1.什么是springboot？**

Springboot是一个快速整合框架，它的底层是SpringFramework(SpringFramework的底层是Java），可以帮助我们简化Spring的开发，快速创建出生产级别的Spring应用，完成应用开发，是整个Spring技术栈的一个大整合，是J2EE开发的一站式解决方案。

**2.Springboot有什么优势？**

传统项目中整合SSH或SSM框架时，需要编写大量配置文件，考虑jar包的冲突,整合起来非常复杂、繁琐。部署时需要将项目买打成一个war包放入到tomacat的webapps目录下进行执行;Springboot解决了SpringFramework中的这些问题：

1.Springboot能够帮我们快速构建独立运行的Spring应用、提供了Starts自动依赖和版本控制可快速整合第三方框架（通过Maven的依赖关系）；

2.完全注解化，大量的自动配置，简化XML文件配置；

3.使用嵌入式的Servlet容器，默认嵌入了tomacat服务器，应用无需打成war包，最终以java应用程序进行执行（使用java创建tomacat且没有web.xml）。使部署更加的简单，只需将项目打成jar包，直接在目标服务器上运行；

4.提供生产级别的监控、健康检查及外部化配置。

5.无代码生成和xml配置，Spring Boot不生成代码。完全不需要任何xml配置即可实现Spring的所有配置；

6.与云计算机天然集成；

**3.Springboot和Spring Framework的关系？**

Spring Framework 是springboot的基础，Spring Boot 是 Spring Framework 的引导程序以简化其配置和使用，通过上层 Spring Boot的引导来使用Spring Framework。

**4.Springboot和Springmvc的关系？**

1.Springboot web组件集成了SpringMvc框架;

2.Springboot启动SpringMVc是没有传统的配置文件,SpringBoot如何启动SpringMvc？在SpringMVC3.0支持注解方式启动;

**5.微服务**

单体应用(传统的WEB应用架构模式)将所有的代码等统统写在一个应用里边，最后再部署到服务器上，从而运行应用。这种模式会带来牵一发而动全身的问题，一旦某一部分出现修改，则整个应用都必须重新部署运行。而且面对日益增长的需求，项目的维护成本会越来越高。

微服务是一种架构风格，提倡一个应用应该是一组小型服务的组合，每一个服务运行在自己的进程内，每一个服务可以通过HTTP的方式进行互通。每一个功能元素最终都是一个可以独立替换和独立升级的软件单元。

Springboot可以快速构建一个微服务单元，SpringBoot+SpringCloud 实现微服务开发。

相关资料： <https://martinfowler.com/microservices/>

**6.Springboot和SpringCloud区别和关系？**

1.Springboot是一个快速开发框架，可以快速帮助我们整理常用第三方框架, 完全采用注解化，简化xml文件配置，使用嵌入式的Servlet容器，最终以java应用程序进行执行，只需将项目打成jar包，直接在目标服务器上运行。

2.SpringCould是一套目前完整微服务解决框架。在务发现注册、配置中心、消息总线、负载均衡、断路器、数据监控等方面提供了一系列解决方案。

3.SpringBoot实现快速开发，可以快速构建一个微服务单元。SpringBoot+SpringCloud 实现微服务开发。SpringCloud具备微服务开发的核心技术：RPC远程调用技术，Springboot Web组件默认集成了SpringMVC，可以实现HTTP+JSON的轻量级传输，使用SpringMVC编写微服务接口，SpringCould依赖于Springboot实现微服务。

# 如何构建Springboot项目

## 1.Springboot2环境要求

1.Java8及以上

2.springboot2.7要求在Maven3.5及以上

## 2.环境准备

1.下载安装并配置JDK；

1)JAVA\_HOME:JDK根目录

2)path:JDK根目录\bin

3)classpath:.;JDK根目录\bin

2.下载安装并配置Maven

1)MAVEN\_HOME:Maven根目录

2)path:Maven根目录\bin

3)修改Maven中的Setting.xml文件  
a.配置Maven本地仓库

<localRepository>D:\\debug\\mavenRepository</localRepository>

b.修改镜像（减小下载jar包的等待时间）

<mirror>

<id>alimaven</id>

<name>aliyun maven</name>

<url>http://maven.aliyun.com/nexus/content/groups/public/</url>

<mirrorOf>central</mirrorOf>

</mirror>

3.在开发工具Eclipse+STS插件、STS或Idea中配置Maven；

windows-->prefereces--->maven--->user settings选项

## 3.Springboot项目特点

1.Spring Boot让配置变的简单，遵循依赖优于配置的原则，可以不用配置，就可快速构建Spring应用，完成功能开发。

2.提供Starter POM，高效包管理，简化maven配置。Springboot将第三方应用（框架）抽取为一个个 Starts（场景），只需要引入对应的场景，就会将该场景的依赖自动导入且管理依赖的版本号；

3.内嵌Servlet容器，默认嵌入了tomacat，省去了配置http服务器的繁琐，无需以war包形式部署项目。

4.快速与主流框架集成。

5.自动配置Spring：Spring Boot会根据项目依赖来自动配置Spring 框架，极大地减少项目要使用的配置。

6.无代码生成和xml配置：Spring Boot不生成代码。完全不需要任何xml配置即可实现Spring的所有配置。

7.让测试变的简单，内置了JUnit、Spring Boot Test等多种测试框架，方便测试。

8.方便监控，使用Spring Boot Actuator组件提供了应用的系统监控，可以查看应用配置的详细信息。

9.提供生产就绪型功能：提供可以直接在生产环境中使用的功能，如性能指标、应用信息和应用健康检查。

相关博客：

https://www.php.cn/java/base/464111.html

https://www.cnblogs.com/jxxblogs/p/14854712.html

## 4.构建项目

### I.)手动构建Maven项目的方式构建Springboot项目

#### 1.构建Maven项目，引入依赖

 <parent>

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

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

        <version>2.7.2</version>

    </parent>

  <dependencies>

    <dependency>

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

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

    </dependency>

  </dependencies>

#### 2.编写服务

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RestController;

@RestController

public class HelloController {

@RequestMapping("/hello")

public String hello() {

 return "Hello Springboot!";

}

}

#### 3.编写主程序启动类

使用@SpringBootApplication标记主配置类；

@SpringBootApplication

public class App {

public static void main(String[] args) {

 SpringApplication.run(App.class, args);

}

}

#### 4.部署

使用Maven插件将项目打包为jar包，直接在目标服务器上执行即可。

<build>

    <plugins>

      <plugin>

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

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

      </plugin>

    </plugins>

  </build>

### II.)使用Spring initializr 构建项目

1.使用Spring提供 https://start.spring.io/

2.Eclipse+STS插件、STS、Idea支持使用Spring initializ快速创建Springboot项目；

## 5.Springboot项目目录结构

1.java：java代码目录  
2.resource：  
static:静态资源(css,js,img,video,audio)  
templates:模板文件（freemarker,thymeleaf,默认不支持JSP）  
3.application.properties：Springboot应用全局配置文件，可以修改默认的配置。

## 3.相关注解

### 1.@Controller

@Controller是控制层注解，SpringMvcUrl接口映射默认情况下返回页面跳转，如果需要返回Json格式数据需要添加@ResponseBody注解；

@Target(ElementType.TYPE)

@Retention(RetentionPolicy.RUNTIME)

@Documented

@Component

public @interface Controller {

......

}

### 2.@ResponseBody

使用@ResponseBody注解表示该方法返回Json格式;

@Target({ElementType.TYPE, ElementType.METHOD})

@Retention(RetentionPolicy.RUNTIME)

@Documented

public @interface ResponseBody {

}

### 3.@RestController

1.@RestController是Springframwork提供的注解，相当于 @Controller ＋ @ResponseBody组合使用；

2.在类上加上@RestController注解表示修饰该Controller所有的方法返回JSON格式。

3.如果在@RestController中的方法想要跳转页面，需要使用ModelAndView进行封装；

@Target(ElementType.TYPE)

@Retention(RetentionPolicy.RUNTIME)

@Documented

@Controller

@ResponseBody

public @interface RestController {

......

}

### 4.@Configuration

1.@Configuration是Spring IoC 容器的配置类。

2.加入 @Configuration的类会被纳入到Spring 容器。

3.Springboot推荐使用配置类来代替xml配置文件。

@Target(ElementType.TYPE)

@Retention(RetentionPolicy.RUNTIME)

@Documented

@Component

public @interface Configuration {

......

}

### 5.@SpringBootApplication

1.@SpringBootApplication标注在某个类上表名这个类是Springboot项目的主配置类，Springboot通过运行这个类的main方法来启动Springboot引用。 2.@SpringBootApplication是@SpringBootConfiguration、@EnableAutoConfiguration、@ComponentScan注解的复合注解；

@Target(ElementType.TYPE)

@Retention(RetentionPolicy.RUNTIME)

@Documented

@Inherited

@SpringBootConfiguration

@EnableAutoConfiguration

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

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

public @interface SpringBootApplication {

······

｝

#### 5.1@SpringBootConfiguration

1.@SpringBootConfiguration是Springboot的配置类，标注在某一个类上表明这是一个Springboot的配置类，等同于@Configuration,将该类标记为配置类，纳入Spring IOC容器管理； 2.Springboot推荐使用配置类来代替xml配置文件； 3.配置类也是容器中的一个组件@Component；

@Target(ElementType.TYPE)

@Retention(RetentionPolicy.RUNTIME)

@Documented

@Configuration

@Indexed

public @interface SpringBootConfiguration {

......

}

@Configuration

@Target(ElementType.TYPE)

@Retention(RetentionPolicy.RUNTIME)

@Documented

@Component

public @interface Configuration {

......

}

#### 5.2.@EnableAutoConfiguration

1.@EnableAutoConfiguration:开启自动配置功能；

2.在传统Spring应用中需要将类所在包（控制器）手工写入scan扫描器中（加入spring容器)，Springboot帮我们自动配置，@EnableAutoConfiguration注解告诉Springboot开启自动配置的功能，这样配置才能生效；

3.@EnableAutoConfiguration包含了@AutoConfigurationPackage注解，使Springboot可以自动配置（约定优于配置）,可以找到主配置类@SpringBootApplication所在类的包，将该包和该包下所有的子包全部纳入Spring容器。

4.@Import(AutoConfigurationImportSelector.class)：@Import是Spring底层注解，给容器导入一个组件。 通过导入组件AutoConfigurationImportSelector类中selectImports()方法引入第三方依赖。Springboot启动的时候会根据META-INF/spring-autoconfigure-metadata.properties"文件找到对应的第三方依赖，并将这些依赖引入项目。

@Target(ElementType.TYPE)

@Retention(RetentionPolicy.RUNTIME)

@Documented

@Inherited

@AutoConfigurationPackage

@Import(AutoConfigurationImportSelector.class)

public @interface EnableAutoConfiguration {

......

}

##### 5.2.1 @AutoConfigurationPackage

@AutoConfigurationPackage注解，使Springboot可以自动配置（约定优于配置）,可以找到主配置类@SpringBootApplication所在类的包，将该包和该包下所有的子包全部纳入Spring容器。

@Target(ElementType.TYPE)

@Retention(RetentionPolicy.RUNTIME)

@Documented

@Inherited

@Import(AutoConfigurationPackages.Registrar.class)

public @interface AutoConfigurationPackage {

......

}

@AutoConfigurationPackage通过AutoConfigurationPackages.Registrar中的register方法将主配置类所在的包及子包下纳入Spring管理；

public abstract class AutoConfigurationPackages {

*/\*\**

  \* {@link ImportBeanDefinitionRegistrar} to store the base package from the importing

  \* configuration.

  \*/

 static class Registrar implements ImportBeanDefinitionRegistrar, DeterminableImports {

  @Override

  public void registerBeanDefinitions(AnnotationMetadata metadata, BeanDefinitionRegistry registry) {

   register(registry, new PackageImports(metadata).getPackageNames().toArray(new String[0]));

  }

  @Override

  public Set<Object> determineImports(AnnotationMetadata metadata) {

   return Collections.singleton(new PackageImports(metadata));

  }

 }

}

Springboot项目中通过AutoConfigurationImportSelector类中selectImports()方法引入第三方依赖;Spring 启动的时候会根据"MMETA-INF/spring-autoconfigure-metadata.properties"文件找到对应的第三方依赖，并将这些依赖引入项目。

public class AutoConfigurationImportSelector implements DeferredImportSelector, BeanClassLoaderAware,

  ResourceLoaderAware, BeanFactoryAware, EnvironmentAware, Ordered {

 @Override

  public Iterable<Entry> selectImports() {

   if (this.autoConfigurationEntries.isEmpty()) {

    return Collections.emptyList();

   }

   Set<String> allExclusions = this.autoConfigurationEntries.stream()

     .map(AutoConfigurationEntry::getExclusions).flatMap(Collection::stream).collect(Collectors.toSet());

   Set<String> processedConfigurations = this.autoConfigurationEntries.stream()

     .map(AutoConfigurationEntry::getConfigurations).flatMap(Collection::stream)

     .collect(Collectors.toCollection(LinkedHashSet::new));

   processedConfigurations.removeAll(allExclusions);

   return sortAutoConfigurations(processedConfigurations, getAutoConfigurationMetadata()).stream()

     .map((importClassName) -> new Entry(this.entries.get(importClassName), importClassName))

     .collect(Collectors.toList());

  }

  private AutoConfigurationMetadata getAutoConfigurationMetadata() {

   if (this.autoConfigurationMetadata == null) {

    this.autoConfigurationMetadata = AutoConfigurationMetadataLoader.loadMetadata(this.beanClassLoader);

   }

   return this.autoConfigurationMetadata;

  }

 }

AutoConfigurationMetadataLoader.loadMetadata()方法

final class AutoConfigurationMetadataLoader {

 protected static final String PATH = "META-INF/spring-autoconfigure-metadata.properties";

 private AutoConfigurationMetadataLoader() {

 }

 static AutoConfigurationMetadata loadMetadata(ClassLoader classLoader) {

  return loadMetadata(classLoader, PATH);

 }

}

#### 5.3@ComponentScan

@ComponentScan:根据定义的扫描路径，把符合扫描规则的类装配到spring容器中;

@Retention(RetentionPolicy.RUNTIME)

@Target(ElementType.TYPE)

@Documented

@Repeatable(ComponentScans.class)

public @interface ComponentScan {

……

}

#### 总结

1.在编写项目时，一般会对自己写的代码以及第三方依赖进行配置，Springboot可以自动配置：

a.自己写的代码Springboot可以通过@SpringBootConfiguration自动帮我们配置

b.第三方依赖通过spring-boot-autoconfigure-2.7.2.jar中"MMETA-INF/spring-autoconfigure-metadata.properties"文件进行声明，然后通过@EnableAutoConfiguration开启使用即可。

c.spring-boot-autoconfigure-2.7.2.jar中包含了J2EE整合体系中需要的依赖。

**Springboot是如何完成自动装配**

[Springboot](http://localhost:9093/itour/work/detail?id=141)

阅读  最后发布于 2022-08-19 19:24

**1.Springboot是如何完成自动装配的？**

Springboot会根据我们的配置，去检查自动配置类(XxAutoConfiguration)，满足对应@Conditional的拓展注解中的条件时，配置类会自动生效,将对应的组件纳入Spring管理。

**2.以HttpEncodingAutoConfiguration源码为例**

1.@AutoConfiguration注解将当前类纳入SpringIOC容器;

2.@EnableConfigurationProperties(ServerProperties.class), 通过ServerProperties将默认编码设置为UTF-8（自动装配为UTF-8）

3.@ConditionalOnWebApplication(type = ConditionalOnWebApplication.Type.SERVLET): 如果是一个servlet项目，则成立。

4.@ConditionalOnClass(CharacterEncodingFilter.class)： 如果有CharacterEncodingFilter则成立；

5.@ConditionalOnProperty(prefix = "server.servlet.encoding", value = "enabled", matchIfMissing = true)  
prefix：前缀  
value:等价于name  
matchIfMissing：如果没有匹配则满足  
当属性满足条件时,此条件成立：如果没有配置server.servlet.encoding=enabled,则条件成立;

6.当项目是一个servlet项目、存在CharacterEncodingFilter且没有配置server.servlet.encoding=enabled则会自动装配HttpEncodingAutoConfiguration；

@AutoConfiguration

@EnableConfigurationProperties(ServerProperties.class)

@ConditionalOnWebApplication(type = ConditionalOnWebApplication.Type.SERVLET)

@ConditionalOnClass(CharacterEncodingFilter.class)

@ConditionalOnProperty(prefix = "server.servlet.encoding", value = "enabled", matchIfMissing = true)

public class HttpEncodingAutoConfiguration {

 private final Encoding properties;

 public HttpEncodingAutoConfiguration(ServerProperties properties) {

  this.properties = properties.getServlet().getEncoding();

 }

 @Bean

 @ConditionalOnMissingBean

 public CharacterEncodingFilter characterEncodingFilter() {

  CharacterEncodingFilter filter = new OrderedCharacterEncodingFilter();

  filter.setEncoding(this.properties.getCharset().name());

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

  filter.setForceResponseEncoding(this.properties.shouldForce(Encoding.Type.RESPONSE));

  return filter;

 }

 @Bean

 public LocaleCharsetMappingsCustomizer localeCharsetMappingsCustomizer() {

  return new LocaleCharsetMappingsCustomizer(this.properties);

 }

 static class LocaleCharsetMappingsCustomizer

   implements WebServerFactoryCustomizer<ConfigurableServletWebServerFactory>, Ordered {

  private final Encoding properties;

  LocaleCharsetMappingsCustomizer(Encoding properties) {

   this.properties = properties;

  }

  @Override

  public void customize(ConfigurableServletWebServerFactory factory) {

   if (this.properties.getMapping() != null) {

    factory.setLocaleCharsetMappings(this.properties.getMapping());

   }

  }

  @Override

  public int getOrder() {

   return 0;

  }

 }

}

**3.如何修改默认字符集编码？**

通过HttpEncodingProperties中的prefix+属性名（servlet.encoding.charset）进行修改[在配置文件中yml/properties], 该注解提供了默认的utf-8，并提供了prefix+属性名的方式来修改编码;

server:

servlet:

encoding:

charset: ISO\_8859\_1

**4.总结**

1)每个XxAutoConfiguration都有很多@ConditionalOnXx,当这些条件满足时,则此配置自动生效;但是可以手工修改自动装配XxxProperties中的prefix+属性名=value;

2）全局配置文件中的key,来源于某个XxxProperties中的prefix+属性名=value;

**5.@Conditional的拓展注解**

|  |  |
| --- | --- |
| **@Conditional扩展注解** | **作用** |
| @ConditionalOnJava | 系统的java版本是否符合要求 |
| @ConditionalOnBean | 容器中存在指定Bean |
| @ConditionalOnMissingBean | 容器中不存在指定Bean |
| @ConditionalOnExpression | 满足SpEL表达式指定 |
| @ConditionalOnClass | 系统中有指定的类 |
| @ConditionalOnMissingClass | 系统中没有指定的类 |
| @ConditionalOnSingleCandidate | 容器中只有一个指定的Bean，或者这个Bean是首选Bean |
| @ConditionalOnProperty | 系统中指定的属性是否有指定的值 |
| @ConditionalOnResource | 路径下是否存在指定资源文件 |
| @ConditionalOnWebApplication | 当前是web环境 |
| @ConditionalOnNotWebApplication | 当前不是web环境 |
| @ConditionalOnJndi | JNDI存在指定项 |

**6.如何知道springboot开启，禁止了哪些自动装配？**

在application.properties文件中配置 debug=true, Positive matches:开启的自动配置, Negative matches:禁止的自动配置;

**Springboot配置文件**

**1.Springboot配置文件的作用？**

1.)Springboot的自动配置是基于约定的（默认端口为8080）,可以使用配置文件对默认的配置进行修改。

2.)SpringBoot支持两种配置方式,一种是properties文件,一种是yml文件。springboot默认可以读取application.properties和application.yml文件。使用yml可以减少配置文件的重复性。

3.)properties和yml中的配置，互相补充，二者可以结合使用, 如果冲突properties文件的优先级更高。

**2.Spring默认的全局配置文件**

springboot默认可以读取application.properties和application.yml文件。

**2.1application.properties**

语法：key = value

#端口号

server.port = @server.online.port@

#配置项目名词

server.servlet.context-path=/itour

#指定激活环境

spring.profiles.active = @profiles.active@

#配置应用名称

spring.application.name = itour-online-web

#eureka

eureka.client.service-url.defaultZone=http:*//@eureka.host@:@server.eureka.port@/eureka*

eureka.instance.prefer-ip-address=@eureka.instance.prefer-ip-address@

#beetl

beetl.templatesPath=/templates/

**2.2application.yml**

概念:YAML不是标记语言,YAML是一个人性化的数据序列化所有编程语言的标准。

语法：key:空格+value  
说明：  
1.通过垂直对齐,指定层次关系;

2.字符串默认可以不加引号（双引号|单引号），双引号中转义符生效，单引号和不加引号时转义符无效;

3.包含行内写法和普通写法两种方式（key：value|）  
a.两种写法可以互换;  
b.[set/List/数组],{Map,对象类型的属性}，[]可以省略，{}不可以省略

*#端口号*

server.port = @server.online.port@

*#配置项目名词*

server.servlet.context-path=/itour

*#指定激活环境*

spring.profiles.active = @profiles.active@

*#配置应用名称*

spring.application.name = itour-online-web

*#eureka*

eureka.client.service-url.defaultZone=http://@eureka.host@:@server.eureka.port@/eureka

eureka.instance.prefer-ip-address=@eureka.instance.prefer-ip-address@

*#beetl*

beetl.templatesPath=/templates/

**3.Springboot项目中配置文件的位置**

在Springboot中properties和yml中的配置，相互补充，如果冲突properties的优先级更高。Springboot默认能够读取application.properties和application.yml文件。这两个文件可以存在以下几个目录:

1)普通目录(file)：项目根目录/config

2)普通目录(file)：项目根目录

3)类路径classpath：项目根目录/config

4)类路径classpath：项目根目录

说明：如果某项配置冲突，则优先级从上往下，如果不冲突，则互补结合使用；

**Springboot项目外的配置文件**

如果需要修改的内容较多，可以使用外部的配置文件覆盖内部的配置文件配置的信息；

1.）在项目Run Configrations-->Arguments --Program Arguments-->--spring.config.location=文件路径。如果同一配置，同时存在于内部配置文件和外部配置文件，则外部的配置文件的优先级高于内部。

2.）通过命令行调用外部配置文件，java -jar 项目.jar ----spring.config.location=文件路径

如果需要修改的内容较少，可以在外部使用运行参数覆盖内部配置文件的信息；

1.）在项目Run Configrations-->Arguments --Program Arguments-->--修改参数（--server.port=9090）

2.）通过命令行调用外部参数： java -jar 项目名.jar 修改参数（--server.port=9090）

优先级：多个地方配置时，运行参数>外部配置文件> 内部配置文件

**Springboot 多环境切换**

**1.Springboot多环境切换实现方式**

1.使用properties文件方式

2.使用yml文件实现

3.使用Maven的profile标签实现

**2.使用properties文件方式**

Springboot会默认读取application.properties文件；  
1.为每个环境创建一个application-环境名.properties文件；  
开发环境：application-dev.properties  
测试环境：application-test.properties  
生成环境：application-pro.properties  
2.选择具体的环境  
1.）在application.properties文件指定具体要选择的环境

spring.profiles.active= dev|test|pro

**3.使用yml文件实现**

Springboot会默认读取application.yml文件；

**方式一**

1.为每个环境创建一个application-环境名.yml文件；  
开发环境：application-dev.yml  
测试环境：application-test.yml  
生成环境：application-pro.yml  
2.在application.properties文件指定具体要选择的环境

spring:

profiles:

active: dev|test|pro

**方式二**

通过单个yaml文件，使用"---"将各个环境隔开；

server:

  port: 9061

spring:

  profiles:

    active: test *#指定具体环境*

---

server:

  port: 9062

spring:

  config:

   activate:

    on-profile: test

---

server:

  address: 9063

spring:

  config:

   activate:

    on-profile: pro

**4.使用Maven的profile标签实现**

相关博客：https://blog.csdn.net/coding\_ss/article/details/90787692

1.在maven中使用profiles标签,写入不同环境变量

          <profile>

            <id>dev</id>

            <activation>

                <activeByDefault>true</activeByDefault>

            </activation>

            <properties>

                <profiles.active>dev</profiles.active>

                <server.eureka.port>9090</server.eureka.port>

                <server.member.port>9091</server.member.port>

                <server.travel.port>9092</server.travel.port>

                <server.work.port>9093</server.work.port>

                <server.online.port>9094</server.online.port>

                <eureka.host>192.168.1.18</eureka.host>

                <eureka.instance.prefer-ip-address>true</eureka.instance.prefer-ip-address>

                <!-- 数据库相关 -->

                <spring.datasource.url>jdbc:mysql:*//mysql-dev:3306/itour?serverTimezone=Asia/Shanghai</spring.datasource.url>*

                <spring.datasource.driverClassName>com.mysql.cj.jdbc.Driver</spring.datasource.driverClassName>

                <spring.datasource.username>root</spring.datasource.username>

                <spring.datasource.password>mysql@pass</spring.datasource.password>

            </properties>

        </profile>

  <profile>

            <id>test</id>

            <activation>

                <activeByDefault>false</activeByDefault>

            </activation>

            <properties>

                <profiles.active>test</profiles.active>

                <server.eureka.port>9090</server.eureka.port>

                <server.member.port>9091</server.member.port>

                <server.travel.port>9092</server.travel.port>

                <server.work.port>9093</server.work.port>

                <server.online.port>9094</server.online.port>

                <eureka.host>192.168.1.18</eureka.host>

                <eureka.instance.prefer-ip-address>true</eureka.instance.prefer-ip-address>

                <!-- 数据库相关 -->

                <spring.datasource.url>jdbc:mysql:*//mysql-test:3306/itour?serverTimezone=Asia/Shanghai</spring.datasource.url>*

                <spring.datasource.driverClassName>com.mysql.cj.jdbc.Driver</spring.datasource.driverClassName>

                <spring.datasource.username>root</spring.datasource.username>

                <spring.datasource.password>mysql@pass</spring.datasource.password>

            </properties>

        </profile>

  <profile>

            <id>pro</id>

            <activation>

                <activeByDefault>false</activeByDefault>

            </activation>

            <properties>

                <profiles.active>pro</profiles.active>

                <server.eureka.port>9090</server.eureka.port>

                <server.member.port>9091</server.member.port>

                <server.travel.port>9092</server.travel.port>

                <server.work.port>9093</server.work.port>

                <server.online.port>9094</server.online.port>

                <eureka.host>192.168.1.18</eureka.host>

                <eureka.instance.prefer-ip-address>true</eureka.instance.prefer-ip-address>

                <!-- 数据库相关 -->

                <spring.datasource.url>jdbc:mysql:*//mysql-pro:3306/itour?serverTimezone=Asia/Shanghai</spring.datasource.url>*

                <spring.datasource.driverClassName>com.mysql.cj.jdbc.Driver</spring.datasource.driverClassName>

                <spring.datasource.username>root</spring.datasource.username>

                <spring.datasource.password>mysql@pass</spring.datasource.password>

            </properties>

        </profile>

2.在yml或者properties文件中使用'@@'符合引入

*#端口*

server:

  port: '@server.account.port@'

*#项目名称*

spring:

  profiles:

    active: '@profiles.active@'

  application:

    name: itour-account-service

  datasource: *#mysql的配置*

    url: '@spring.datasource.url@'

    driver-class-name: '@spring.datasource.driverClassName@'

    username: '@spring.datasource.username@'

    password: '@spring.datasource.password@'

*#eureka*

eureka:

  client:

    service-url:

      defaultZone: http://@eureka.host@:@server.eureka.port@/eureka

  instance:

    prefer-ip-address: '@eureka.instance.prefer-ip-address@'

*#mybatis plus*

mybatis-plus:

  mapper-locations: classpath\*:mapping/\*.xml

  type-aliases-package: com.itour.model.account

  configuration:

    map-underscore-to-camel-case: true

3.在启动是默认activeByDefault标签中为true的Profile会被激活，可以使用maven打包时候可以指定对应的环境； run Configrations-->Maven build选择都有的项目--Goals使用clean package -P pro打包，使用-P指定对应的环境；

**5.动态指定环境**

1.）通过项目运行参数指定环境  
STS(Eclipse): Run Configurations--Argument--Program Argument-->--spring.profiles.active = 环境名  
2.)通过命令行方式指定环境  
java -jar 项目名.jar --spring.profiles.active=环境名

3.）通过VM参数指定环境  
STS(Eclipse): Run Configurations--Argument--VM Argument--->-Dspring.profiles.active = 环境名

4）在Maven打包时通过-P指定环境

STS(Eclipse): Run Configurations--Goals-->clean package -P 环境名来指定

**yml文件注入属性**

**1.通过yml文件注入属性的方式**

1）使用@ConfigurationProperties注解

2）使用@Value注解

**2.使用@ConfigurationProperties注解注入属性**

在JavBean中使用@ConfigurationProperties实现对属性的注入；需要spring-boot-configuration-processor的支持

<dependency>

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

  <artifactId>spring-boot-configuration-processor</artifactId>

  <optional>true</optional>

</dependency>

1)新建javaBean

@Component

@ConfigurationProperties(prefix = "student")

@Validated *//开启JSR303数据校验*

public class Student {

private String sname;

@Max(value=120,message="年龄最大不能查过120")

private int age;

private boolean sex;

private Date birthday;

private Map<String,Object> location;

private Map<String,Object> address;

private String[] hobby;

private String[] hobbies;

private String[] interest;

private List<String> skills;

private List<String> skill;

private Pet pet1;

private Pet pet2;

.....(省略了set|get)

｝

@Component

public class Pet {

private String nickName;

private String strain;

......(省略了set|get)

}

2)注入属性

student:

  sname: 'mike' #String

  age: 18 #int

  sex: true  #boolean

  birthday: 2020/8/10 #Date

  location: {province: 湖南,city: 长沙,zone: 岳麓区 } #map-行内写法

  address: #map-普通写法

    province: "上\t海\t市"

    city: '上\t海'

    zone: 宝\t山\t区

  hobby: [游泳,篮球] #数组-行内写法

  hobbies: #数组-普通写法

    - 游泳

    - 篮球

  interest: 美食,体育 #数组-[]可以省略

  skill: ["编程","#集合"] #集合-行内写法

  skills: #集合 -普通写法

    - 编程

    - 金融

  pet1: {nickName: "marc", strain: "hsq"} #自定义对象-行内写法  {}不能省略

  pet2: #自定义对象-普通写法

    nick-name: "Mataline"

    strain: "bg"

3）测试

*/\*\**

 \*

 \* @author wwang

 \*1.@SpringBootTest:注解告诉 SpringBoot 去寻找一个主配置类(带有 @SpringBootApplication 的配置类)，并使用它来启动 Spring 应用程序上下文。

 \*  SpringBootTest加载完整的应用程序并注入所有可能的bean。

 \*2.@Test：表示测试方法,是junit提供的注解;

 \*/

@SpringBootTest

public class SpringbootJunit5Test {

 @Autowired

 Student stu;

 @Test

void testYaml() {

 System.out.println(stu);

}

}

4)结果

Student [

    sname=mike,

 age=18,

 sex=true,

 birthday=Mon Aug 10 00:00:00 GMT+08:00 2020,

 location={province=湖南, city=长沙, zone=岳麓区},

 address={province=上 海 市, city=上\t海, zone=宝\t山\t区},

 hobby=[游泳, 篮球],

 hobbies=[游泳, 篮球],

 interest=[美食, 体育],

 skills=[编程, 金融],

 skill=[编程, #集合],

 pet1=Pet [nickName=marc, strain=hsq],

 pet2=Pet [nickName=Mataline, strain=bg]

 ]

5)总结  
1.字符串默认可以不加引号（双引号|单引号），双引号中转义符生效，单引号和不加引号时转义符无效;

2.行类写法中，[set/List/数组],{Map,对象类型的属性}，[]可以省略，{}不可以省略；

**3.使用@Value注解注入属性**

1)编写javaBean注入

@Component

*//@ConfigurationProperties(prefix = "pet")*

@Validated *//开启JSR303数据校验*

public class Pet {

 @Value("${pet.nickName}")

 @Length(max = 6,message = "品种长度不得大于6")

private String nickName;

@Value("阿拉斯加雪橇犬")

private String strain;

@Value("#{2\*3}")

private int age;

......(省略了set|get)

}

2)在yml中配置设置值

pet:

 nickName: 'Mataline'

 strain: "bg"

 age： '#{2\*2}' *#不支持SpEL表达式*

1.3)测试

@SpringBootTest

public class SpringbootJunit5Test {

 @Autowired

 Pet pet;

 @Test

void testYaml() {

    System.out.println(pet);

}

}

4)结果

Pet [nickName=Mataline, strain=阿拉斯加雪橇犬, age=6]

5)总结：  
1.支持Spring表达式（SpEl）；

2.两者混用@ConfigurationProperties的优先级比@value的优先级高，但是二者可以互补;

**4.@ConfigurationProperties和@Value的区别**

|  |  |  |
| --- | --- | --- |
| **功能** | **@ConfigurationProperties** | **@value** |
| 优先级 | 如果两者混用@ConfigurationProperties的优先级比@value的优先级高，但是二者可以互补; | |
| 注值 | 支持批量注入 | 单个注入 |
| 松散语法(nickName-->nick-name(驼峰-->-的形式)) | 支持 | 不支持 |
| Spring表达式（SpEL） | 支持 | 不支持 |
| JSR303数据校验 | 支持 | 不支持 |
| 注入复杂类型（除基本类型，String，Date以外类型） | 支持 | 不支持 |

**5.JSR303**

JSR303是一套JavaBean参数校验的标准，定义了很多常用的校验注解，可以直接将这些注解加在我们JavaBean的属性上面就可以在需要校验的时候进行校验了。 1.需要添加依赖

<dependency>

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

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

</dependency>

**6.@PropertySource注解**

1.@ConfigurationProperties注解默认会加载application.properties和application.yml文件中的数据。

2.当读取不在默认全局配置文件中的数据时，可以使用@PropertySourc(evalue="classpath:conf.properties")加载conf.properties文件的数据,但是该注解只能加载properties文件,不支持yml文件；

1.创建conf.properties文件

pet.nickName = Mataline

pet.strain = hsq

pet.age = 8

2.创建javaBean

@Component

@Validated *//开启JSR303数据校验*

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

@ConfigurationProperties(prefix = "pet")

public class Pet {

private String nickName;

@Length(max = 6,message = "品种长度不得大于6")

private String strain;

private int age;

.....(省略了set|get)

}

3.测试

@SpringBootTest

public class SpringbootJunit5Test {

 @Autowired

 Pet pet;

 @Test

void testYaml() {

    System.out.println(pet);

}

}

4.结果

Pet [nickName=Mataline, strain=hsq, age=8]

**7.@ImportResource**

1.springboot项目自动装配，默认会将Spring等配置文件自动配置好，如果自己编写的Spring等配置文件，Springboot默认不能识别。如果需要识别需要在Springboot主配置类上通过@ImportResource指定配置文件的路径；

2.Springboot推荐使用注解的方式进行配置（通过@Configration和@Bean注解实现）；

**7.1使用@ImportResource引入自己编写的配置文件**

1. 编写自己的配置文件

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:aop="http://www.springframework.org/schema/aop"

 xmlns:tx="http://www.springframework.org/schema/tx" xmlns:jdbc="http://www.springframework.org/schema/jdbc"

 xmlns:context="http://www.springframework.org/schema/context"

 xsi:schemaLocation="

      http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-4.1.xsd

      http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-4.1.xsd

      http://www.springframework.org/schema/jdbc http://www.springframework.org/schema/jdbc/spring-jdbc-4.1.xsd

      http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-4.1.xsd

      http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-4.1.xsd">

<bean id="p" class="com.tms.entity.Pet"></bean>

</beans>

2.在主配置类上使用@ImportResource

@SpringBootApplication

@ImportResource(locations = {"classpath:Bean1.xml"})

public class App {

public static void main(String[] args) {

 SpringApplication.run(App.class, args);

}

}

3.测试

*/\*\**

 \*

 \* @author wwang

 \*1.@SpringBootTest:注解告诉 SpringBoot 去寻找一个主配置类(带有 @SpringBootApplication 的配置类)，并使用它来启动 Spring 应用程序上下文。

 \*  SpringBootTest加载完整的应用程序并注入所有可能的bean。

 \*2.@Test：表示测试方法,是junit提供的注解;

 \*/

@SpringBootTest

public class SpringbootJunit5Test {

 @Test

 void testSpringXML() {

  ApplicationContext context = new ClassPathXmlApplicationContext("classpath:Bean1.xml");

  Pet bean = context.getBean("p", Pet.class);

  System.out.println(bean);

 }

}

**7.2使用注解方式实现**

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.ComponentScan;

import org.springframework.context.annotation.Configuration;

import org.springframework.context.annotation.EnableAspectJAutoProxy;

import org.springframework.jdbc.core.JdbcTemplate;

import org.springframework.jdbc.datasource.DataSourceTransactionManager;

import org.springframework.transaction.annotation.EnableTransactionManagement;

import com.alibaba.druid.pool.DruidDataSource;

@Configuration*//表示配置类*

@ComponentScan(basePackages = "com.itour")*//开启注解扫描*

@EnableTransactionManagement*//开启事务注解*

public class SpringConfig {

*//配置数据源*

@Bean *//将组件纳入Spring容器*

public DruidDataSource dataSource() {

 DruidDataSource dataSource = new DruidDataSource();

 dataSource.setUrl("jdbc:mysql://1.116.226.147:3307/itour?serverTimezone=Asia/Shanghai");

 dataSource.setDriverClassName("com.mysql.cj.jdbc.Driver");

 dataSource.setUsername("root");

 dataSource.setPassword("mysql@958958");

 return dataSource;

}

*//配置JdbcTemplate*

@Bean

public JdbcTemplate jdbcTemplate(DruidDataSource dataSource) {

 JdbcTemplate jdbcTemplate = new JdbcTemplate();

 jdbcTemplate.setDataSource(dataSource);

 return jdbcTemplate;

}

*//配置事务管理器*

@Bean

public DataSourceTransactionManager transactionManager(DruidDataSource dataSource) {

 DataSourceTransactionManager transactionManager = new DataSourceTransactionManager();

 transactionManager.setDataSource(dataSource);

 return transactionManager;

}

}

**8.Springboot全局配置文件中的占位表达式**

1.随机数  
$(random.uuid)：uuid  
$(random.value)：随机字符串  
$(random.int)：随机整型数  
$(random.long)：随机长整型数  
$(random.int（10)：随机10以内的整型数  
$(random.int(m,n))：随机m-n之间的数字  
2.引用变量值  
1.在yml中是占位符表达式

pet1:

  nickName: Mataline

pet:

 nickName: ${pet1.nickName}

 strain: "bg"

 age: '${random.int(10)}'

2.编写javabean进行注入

@Component

@ConfigurationProperties(prefix = "pet")

public class Pet {

private String nickName;

private String strain;

private int age;

}

3.测试

@SpringBootTest

public class SpringbootJunit5Test {

 @Autowired

 Pet pet;

 @Test

void testYaml() {

    System.out.println(pet);

}

}

4.结果

Pet [nickName=Mataline, strain=bg, age=7]

**Springboot如何处理静态资源**

**1.Springboot2.7.2 web项目静态资源放在哪儿？**

Springboot最终打包为一个jar,静态资源部不再存放在webapps中，静态资源存放的路径通过WebMvcAutoConfiguration类中的addResourceHandlers()指定:/webjar/；Springboot将静态资源存入jar包中，引入的时候从jar目录结构的webjars开始写（localhost:8080/webjars/jquery/3.3.1-1/jquery.js）。

@AutoConfiguration(after = { DispatcherServletAutoConfiguration.class, TaskExecutionAutoConfiguration.class,

  ValidationAutoConfiguration.class })

@ConditionalOnWebApplication(type = Type.SERVLET)

@ConditionalOnClass({ Servlet.class, DispatcherServlet.class, WebMvcConfigurer.class })

@ConditionalOnMissingBean(WebMvcConfigurationSupport.class)

@AutoConfigureOrder(Ordered.HIGHEST\_PRECEDENCE + 10)

public class WebMvcAutoConfiguration {

@Override

  public void addResourceHandlers(ResourceHandlerRegistry registry) {

   if (!this.resourceProperties.isAddMappings()) {

    logger.debug("Default resource handling disabled");

    return;

   }

   addResourceHandler(registry, "/webjars/\*\*", "classpath:/META-INF/resources/webjars/");

   addResourceHandler(registry, this.mvcProperties.getStaticPathPattern(), (registration) -> {

    registration.addResourceLocations(this.resourceProperties.getStaticLocations());

    if (this.servletContext != null) {

     ServletContextResource resource = new ServletContextResource(this.servletContext, SERVLET\_LOCATION);

     registration.addResourceLocations(resource);

    }

   });

  }

}

this.resourceProperties.getStaticLocations()

@ConfigurationProperties("spring.web")

public class WebProperties {

public static class Resources {

  private static final String[] CLASSPATH\_RESOURCE\_LOCATIONS = { "classpath:/META-INF/resources/",

    "classpath:/resources/", "classpath:/static/", "classpath:/public/" };

*/\*\**

   \* Locations of static resources. Defaults to classpath:[/META-INF/resources/,

   \* /resources/, /static/, /public/].

   \*/

  private String[] staticLocations = CLASSPATH\_RESOURCE\_LOCATIONS;

*/\*\**

   \* Whether to enable default resource handling.

   \*/

  private boolean addMappings = true;

  private boolean customized = false;

  private final Chain chain = new Chain();

  private final Cache cache = new Cache();

  public String[] getStaticLocations() {

   return this.staticLocations;

  }

  }

}

**2.webjar**

WebJars是将客户端（浏览器）资源（JavaScript，Css等）打成jar包文件，以对资源进行统一依赖管理。WebJars的jar包部署在Maven中央仓库上。关于webjars资源，有一个专门的网站<http://www.webjars.org/>

**3.Springboot2.7.2自己写的静态资源如何放入springboot中？**

1.将自己写的静态资源打成jar包使用（不推荐）。

2.Springboot将一些目录约定为静态资源存放的目录，将自己编写的静态资源直接放入这些目录即可；

private static final String[] CLASSPATH\_RESOURCE\_LOCATIONS = { "classpath:/META-INF/resources/",

    "classpath:/resources/", "classpath:/static/", "classpath:/public/" };

说明：在springboot约定的静态资源目录中存放资源文件后，访问时不需要加前缀，直接访问即可（http://localhost:8080/word.html）。

**4.Springboot2.7.2设置欢迎页**

通过WebMvcAutoConfiguration类中welcomePageHandlerMapping（）-->getWelcomePage-->getIndexHtml()-->location.createRelative("index.html")。location就是Springboot约定为静态资源的目录。

所有我们只需要将欢迎页取名index.html并放入Springboot约定的静态资源目录中即可。即任意一个静态资源目录中的index.html就是欢迎页。

@AutoConfiguration(after = { DispatcherServletAutoConfiguration.class, TaskExecutionAutoConfiguration.class,

  ValidationAutoConfiguration.class })

@ConditionalOnWebApplication(type = Type.SERVLET)

@ConditionalOnClass({ Servlet.class, DispatcherServlet.class, WebMvcConfigurer.class })

@ConditionalOnMissingBean(WebMvcConfigurationSupport.class)

@AutoConfigureOrder(Ordered.HIGHEST\_PRECEDENCE + 10)

public class WebMvcAutoConfiguration {

  @Configuration(proxyBeanMethods = false)

 @EnableConfigurationProperties(WebProperties.class)

 public static class EnableWebMvcConfiguration extends DelegatingWebMvcConfiguration implements ResourceLoaderAware {

  @Bean

  public WelcomePageHandlerMapping welcomePageHandlerMapping(ApplicationContext applicationContext,

    FormattingConversionService mvcConversionService, ResourceUrlProvider mvcResourceUrlProvider) {

   WelcomePageHandlerMapping welcomePageHandlerMapping = new WelcomePageHandlerMapping(

     new TemplateAvailabilityProviders(applicationContext), applicationContext, getWelcomePage(),

     this.mvcProperties.getStaticPathPattern());

   welcomePageHandlerMapping.setInterceptors(getInterceptors(mvcConversionService, mvcResourceUrlProvider));

   welcomePageHandlerMapping.setCorsConfigurations(getCorsConfigurations());

   return welcomePageHandlerMapping;

  }

private Resource getWelcomePage() {

   for (String location : this.resourceProperties.getStaticLocations()) {

    Resource indexHtml = getIndexHtml(location);

    if (indexHtml != null) {

     return indexHtml;

    }

   }

   ServletContext servletContext = getServletContext();

   if (servletContext != null) {

    return getIndexHtml(new ServletContextResource(servletContext, SERVLET\_LOCATION));

   }

   return null;

  }

  private Resource getIndexHtml(String location) {

   return getIndexHtml(this.resourceLoader.getResource(location));

  }

  private Resource getIndexHtml(Resource location) {

   try {

    Resource resource = location.createRelative("index.html");

    if (resource.exists() && (resource.getURL() != null)) {

     return resource;

    }

   }

   catch (Exception ex) {

   }

   return null;

  }

}

}

**5.Springboot2.7.2如何自定义静态资源目录**

通过Properties文件中的prefix+属性方式修改默认的配置； spring.web.resources.static-locations=classpath:/res/,classpath:/static/ 自定义静态资源目录后，Springboot约定的静态资源目录就不能识别了；

@ConfigurationProperties("spring.web")

public class WebProperties {

private final Resources resources = new Resources();

 public static class Resources {

  private static final String[] CLASSPATH\_RESOURCE\_LOCATIONS = { "classpath:/META-INF/resources/",

    "classpath:/resources/", "classpath:/static/", "classpath:/public/" };

*/\*\**

   \* Locations of static resources. Defaults to classpath:[/META-INF/resources/,

   \* /resources/, /static/, /public/].

   \*/

  private String[] staticLocations = CLASSPATH\_RESOURCE\_LOCATIONS;

}

}

**6.网站标签的logo设置**

网站中创建：favicon.ico，在页面中引入

<link rel="icon" href="${ctxPath}/img/favicon.ico" />

**总结**

1.通过源码发现静态资源的目录； 2.使用静态资源：只需要将静态资源放入Springboot约定的静态资源目录中； 3.欢迎页只需将对应的页面放入Springboot约定的目录中即可；

**Springboot整合Jdbc**

[Springboot](http://localhost:9093/itour/work/detail?id=155)

阅读  最后发布于 2022-08-18 17:14

**1.Springboot操作数据库**

对于数据库访问层,无论是SQL还是NOSQL，Springboot默认采用整合Spring Data的方式进行统一处理，添加大量的自动配置,屏蔽了很多的设置。 引入了各种XXXTemplate,xxxRepository来简化对数据库的操作。对我们来说只需要进行简单的设置即可。

**2.Springboot整合基本的JDBC和数据源**

**1.)引入对应的依赖**

<!-- jdbc -->

<dependency>

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

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

</dependency>

<!-- MYSQL驱动包 -->

<dependency>

 <groupId>mysql</groupId>

 <artifactId>mysql-connector-java</artifactId>

</dependency>

**2.)配置数据源**

Springboot2.7.2默认使用com.zaxxer.hikari.HikariDataSource作为数据源。数据源的相关配置都在DataSourceProperties里面，可以使用type属性指定数据源类型;

spring:

  datasource:

    url: jdbc:mysql:*//1.116.226.147:3307/itour?serverTimezone=UTC*

    driver-class-name: com.mysql.cj.jdbc.Driver

    username: root

    password: mysql@958958

    type: com.zaxxer.hikari.HikariDataSource

**3)注入JdbcTemplate对象，使用JdbcTemplate操作数据库**

@RestController

public class HelloController {

 @Autowired

 JdbcTemplate jbcTemplate;

 @Autowired

 DataSource dataSource;

@RequestMapping("/dataSource")

public Map<String,Object> testSource() throws SQLException {

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

 map.put("dataSource", dataSource.getClass());

 map.put("jbcTemplate", jbcTemplate.getClass());

 return map;

}

@RequestMapping("/testJdbc")

public Map<String,Object> testJdbc(String id) throws SQLException {

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

 String sql = "select \* from t\_a\_oauth where id=?";

 Oauth oauth = jbcTemplate.queryForObject(sql, new BeanPropertyRowMapper<Oauth>(Oauth.class),id);

 map.put("Oauth", oauth);

 return map;

}

}

# SpringBoot配置Druid数据源

[Springboot](http://localhost:9093/itour/work/detail?id=157)

阅读  最后发布于 2022-08-19 13:02

## 为什么使用Druid

Druid在功能、性能、扩展性方面表现出色。Druid内置了一个功能强大的StatFilter插件可以监控数据库访问性能，可以清楚知道连接池和SQL的工作情况。

github地址：<https://github.com/alibaba/druid/tree/master/druid-spring-boot-starter>

## 配置Druid数据源的步骤

### 1.引入druid数据源和数据库驱动包

<dependency>

   <groupId>com.alibaba</groupId>

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

   <version>1.2.11</version>

</dependency>

<dependency>

   <groupId>mysql</groupId>

   <artifactId>mysql-connector-java</artifactId>

</dependency>

### 2.配置Druid

Springboot默认HikariDataSource作为数据源，可使用type指定数据源；

#### 1）使用yml方式配置

spring:

  datasource:

    url: jdbc:mysql:*//localhost:3307/itour?serverTimezone=UTC*

    driver-class-name: com.mysql.cj.jdbc.Driver

    username: root

    password: root

    type: com.alibaba.druid.pool.DruidDataSource

    druid:

                # 不配置，无法监控SQL

      filters: stat,wall

                #配置StatViewServlet(管理后台的Servlet)

      stat-view-servlet:

                      #是否启用StatViewServlet

        enabled: true

                       #访问路径为/druid时，跳转到StatViewServlet

        url-pattern: /druid/\*

                       #是否能重置数据

        reset-enable: false

                      # 设置账号信息

        login-username: admin

        login-password: admin

                  #配置WebStatFilterm(管理监控的filter)

      web-stat-filter:

                      #是否启用WebStatFilterm

        enabled: true

                       #配置过滤规则

        url-pattern: /\*

                      #配置忽略过滤规则

        exclusions: /druid/\*,\*.js,\*.gif,\*.jpg,\*.png,\*.css,\*.ico

#### 2）使用JavaBean方式配置

1.在yml或properties文件中配置

spring:

  datasource:

    url: jdbc:mysql://1.116.226.147:3307/itour?serverTimezone=UTC

    driver-class-name: com.mysql.cj.jdbc.Driver

    username: root

    password: mysql@958958

    type: com.alibaba.druid.pool.DruidDataSource

    druid:

*# 不配置，无法监控SQL*

      filters: stat,wall

2.使用@ConfigurationProperties注解注入对应的属性

@Configuration

public class DruidCofig {

 @Bean

 @ConfigurationProperties(prefix = "spring.datasource.druid")

public DataSource druidDateSource() {

 DataSource dataSource = new DruidDataSourceWrapper();

 return dataSource;

}

*//1.配置Druid的监控,配置一个管理后台的Servlet*

 @Bean

public ServletRegistrationBean<StatViewServlet> statViewServlet() {

 ServletRegistrationBean<StatViewServlet> bean = new ServletRegistrationBean<StatViewServlet>(new StatViewServlet(), "/druid/\*");

 Map<String, String> initParams = new HashMap<String, String>();

  initParams.put("loginUsername", "admin");*//用户名*

     initParams.put("loginPassword", "top@141421");*//密码*

*/\* 默认就是允许所有访问 \*/*

     initParams.put("allow", "");*//白名单，没有配置或者为空，则允许所有访问*

*//initParams.put("deny","");//黑名单，存在共同时，deny优先于allow*

     initParams.put("resetEnable","false");*//禁止HTML页面重置数据功能*

  bean.setInitParameters(initParams);

 return bean;

}

*//2.配置一个管理监控的filter*

 @Bean

public FilterRegistrationBean<WebStatFilter> webStatfilter() throws SQLException {

 FilterRegistrationBean<WebStatFilter> bean = new FilterRegistrationBean<WebStatFilter>();

 bean.setFilter(new WebStatFilter());

 Map<String, String> initParams = new HashMap<String, String>();

 initParams.put("exclusions","\*.js,\*.css,/druid/\*,\*gif,\*.jpg,\*.png,\*.ico");

 bean.setInitParameters(initParams);

 bean.setUrlPatterns(Arrays.asList("/\*"));

 return bean;

}

}

### 3 .测试

http://localhost:9061/druid

@RestController

public class HelloController {

 @Autowired

 JdbcTemplate jbcTemplate;

 @Autowired

 DataSource dataSource;

@RequestMapping("/dataSource")

public Map<String,Object> testSource() throws SQLException {

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

 map.put("dataSource", dataSource.getClass());

 map.put("jbcTemplate", jbcTemplate.getClass());

 DruidDataSource dsource =  (DruidDataSource)dataSource;

 map.put("minIdle", dsource.getMinIdle());

 map.put("initialSize", dsource.getInitialSize());

 return map;

}

@RequestMapping("/testJdbc")

public Map<String,Object> testJdbc(String id) throws SQLException {

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

 String sql = "select \* from t\_a\_oauth where id=?";

 Oauth oauth = jbcTemplate.queryForObject(sql, new BeanPropertyRowMapper<Oauth>(Oauth.class),id);

 map.put("Oauth", oauth);

 return map;

}

}

# Springboot集成Mybatis

## 1.集成Mybatis

### 1.引入Mybatis依赖及数据库驱动包

<!-- mybatis-->

<dependency>

<groupId>org.mybatis.spring.boot</groupId>

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

<version>2.2.2</version>

</dependency>

<!-- MYSQL驱动包 -->

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>8.0.31</version>

</dependency>

### 2.配置Mybatis

spring:

datasource: #mysql的配置

url: jdbc:mysql://1.116.226.147:3307/itour?serverTimezone=UTC

driver-class-name: com.mysql.cj.jdbc.Driver

username: root

password: mysql@958958

mybatis:

mapper-locations:

- classpath\*:mapping/\*.xml

type-aliases-package: com.explore.model.member

configuration:

map-underscore-to-camel-case: true

### 使用

**1.基于Mybatis注解方式**

**1.）编写Mapper**

使用@Mapper注解标记Mapper接口

@Mapper

public interface AccountMapper{

 @Select("SELECT \* FROM t\_account c")

 public List<Account> queryAccount();

 @Update("update t\_account set username=#{userName},money=#{money} where id=#{id}")

 public int updateAccount(Account account);

 @Insert("insert into t\_account(username,money)values(#{userName},#{money})")

 public int insertAccount(Account account);

 @Delete("delete from t\_account where id=#{id}")

 public int deleteAccount(Long id);

}

**3.）测试**

@RestController

public class HelloController {

 @Autowired

 DataSource dataSource;

 @Autowired

 AccountMapper accountMapper;

@RequestMapping("/testSelet")

public Object testSelet() throws SQLException {

 List<Account> queryAccount = accountMapper.queryAccount();

 return queryAccount;

}

@RequestMapping("/testUpdate")

public Object testUpdate(Account account) throws SQLException {

 int  updateAccount = accountMapper.updateAccount(account);

 return updateAccount;

}

@RequestMapping("/testInsert")

public Object testInsert(Account account) throws SQLException {

 int  insertAccount = accountMapper.insertAccount(account);

 return insertAccount;

}

@RequestMapping("/testDelete")

public Object testDelete(Long id) throws SQLException {

 int  deleteAccount = accountMapper.deleteAccount(id);

 return deleteAccount;

}

**总结**

1.当Mapper较多的时候使用@Mapper注解相对繁琐，可以使用@MapperScan批量扫描所有的Mapper接口； 2.使用Mybatis注解方式编写SQL和配置文件无关；

@SpringBootApplication

@MapperScan("com.tms.mapper")

public class App {

public static void main(String[] args) {

SpringApplication.run(App.class, args);

}

}

**2.基于XML方式**

**2）编写配置文件**

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN" "http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<mapper namespace="com.stu.mybatis.mapper.EmpMapper">

*<!-- 通用查询映射结果 -->*

 <resultMap id="BaseResultMap" type="com.stu.mybatis.Emp">

  <id column="EMPNO" property="empno"  />

     <result column="ENAME" property="ename"  />

     <result column="JOB" property="job"  />

     <result column="E\_HOBBY" property="eHobby"  />

 </resultMap>

    <select id="selectEmp" resultType="com.stu.mybatis.Emp">

      select \* from emp  where empno = #{empno}

    </select>

    <select id="selectEmpList" resultType="com.stu.mybatis.Emp">

      select \* from emp

      <where>

       <if test="ename!=null">

         ename like concat(concat('%',#{ename}),'%')

       </if>

     </where>

    </select>

    <insert id="insertEmp" parameterType="com.stu.mybatis.Emp">

*<!--  keyProperty:查出的主键值封装到JavaBean的那个属性。*

         order="BEFORE":当前SQL在插入SQL之前运行。

         resultType:查出数据返回类型。 -->

     <selectKey keyProperty="empno" order="BEFORE" resultType="Long">

      select seq\_emp.nextval from dual

     </selectKey>

*<!--  插入时的主键从序列中拿到 -->*

    insert into emp(empno,ename,job,e\_hobby)values(#{empno},#{ename},#{job},#{eHobby})

    </insert>

    <update id="updateEmp" parameterType="com.stu.mybatis.Emp">

    update emp set ename=#{ename},job=#{job},e\_hobby=#{eHobby} where empno= #{empno}

    </update>

    <delete id="deleteEmp" parameterType="Long">

     delete emp where empno=#{empno}

    </delete>

    <select id="queryEmp" resultType="com.stu.mybatis.Emp">

     select \* from emp c

     <where>

       <if test="ename!=null">

         ename like concat(concat('%',#{ename}),'%')

       </if>

     </where>

    </select>

</mapper>

**3）编写Mapper接口**

public interface EmpMapper {

  Emp selectEmp(Long id, @Param("hobby") String eHobby);

  Long insertEmp(Emp emp);

  Long updateEmp(Emp emp);

  Long deleteEmp(Long empno);

  List<Emp> queryEmp(Emp emp);

  List<Emp> selectEmpList();

}

**4)测试**

public class MybatisTest {

 @Test

 public void testHello() {

  System.out.println("hello world!");

 }

    @Test

    public void testSelectEmp() throws IOException  {

*//1.获取SqlSessionFactory对象*

      SqlSessionFactory sqlSessionFactory = getSqlSessionFactory();

*//2.获取sqlSession对象*

   SqlSession openSession = sqlSessionFactory.openSession();

   System.out.println(openSession);

*//3.获取接口的实现对象*

   EmpMapper mapper = openSession.getMapper(EmpMapper.class);

*//4.调用增删改查方法*

   try {

*//1.查询单条*

    Emp selectOne = mapper.selectEmp(7369L,"打球");

    System.out.println(selectOne);

   }catch (Exception e) {

*// TODO: handle exception*

    e.printStackTrace();

   }finally {

    openSession.close();

   }

  }

    @Test

    public void testSelectEmpList() throws IOException  {

*//1.获取SqlSessionFactory对象*

     SqlSessionFactory sqlSessionFactory = getSqlSessionFactory();

*//2.获取sqlSession对象*

     SqlSession openSession = sqlSessionFactory.openSession();

     System.out.println(openSession);

*//3.获取接口的实现对象*

     EmpMapper mapper = openSession.getMapper(EmpMapper.class);

*//4.调用增删改查方法*

     try {

*//查询列表*

      List<Emp> selectEmpList = mapper.selectEmpList();

      System.out.println(selectEmpList);

     }catch (Exception e) {

*// TODO: handle exception*

      e.printStackTrace();

     }finally {

      openSession.close();

     }

    }

    @Test

    public void testInsertEmp() throws IOException  {

*//1.获取SqlSessionFactory对象*

     SqlSessionFactory sqlSessionFactory = getSqlSessionFactory();

*//2.获取sqlSession对象*

     SqlSession openSession = sqlSessionFactory.openSession();

     System.out.println(openSession);

*//3.获取接口的实现对象*

     EmpMapper mapper = openSession.getMapper(EmpMapper.class);

*//4.调用增删改查方法*

     try {

*//插入*

      Emp emp = new Emp();

      emp.seteHobby("摄影");

      emp.setEname("oracle");

      emp.setJob("doctor");

      Long insertEmp = mapper.insertEmp(emp);

      openSession.commit();

     }catch (Exception e) {

*// TODO: handle exception*

      e.printStackTrace();

     }finally {

      openSession.close();

     }

    }

    @Test

    public void testUpdateEmp() throws IOException  {

*//1.获取SqlSessionFactory对象*

     SqlSessionFactory sqlSessionFactory = getSqlSessionFactory();

*//2.获取sqlSession对象*

     SqlSession openSession = sqlSessionFactory.openSession();

     System.out.println(openSession);

*//3.获取接口的实现对象*

     EmpMapper mapper = openSession.getMapper(EmpMapper.class);

*//4.调用增删改查方法*

     try {

      Emp e = new Emp();

      e.seteHobby("游泳,打篮球");

      e.setEname("mike1");

      e.setJob("doctor1");

      e.setEmpno(6L);

      Long updateEmp = mapper.updateEmp(e);

      openSession.commit();

     }catch (Exception e) {

*// TODO: handle exception*

      e.printStackTrace();

     }finally {

      openSession.close();

     }

    }

    @Test

    public void testDeleteEmp() throws IOException  {

*//1.获取SqlSessionFactory对象*

     SqlSessionFactory sqlSessionFactory = getSqlSessionFactory();

*//2.获取sqlSession对象*

     SqlSession openSession = sqlSessionFactory.openSession();

     System.out.println(openSession);

*//3.获取接口的实现对象*

     EmpMapper mapper = openSession.getMapper(EmpMapper.class);

*//4.调用增删改查方法*

     try {

   mapper.deleteEmp(7L);

      openSession.commit();

     }catch (Exception e) {

*// TODO: handle exception*

      e.printStackTrace();

     }finally {

      openSession.close();

     }

    }

   @Test

 public void test11() throws IOException {

*//1.获取SqlSessionFactory对象*

     SqlSessionFactory sqlSessionFactory = getSqlSessionFactory();

*//2.获取sqlSession对象*

  SqlSession openSession = sqlSessionFactory.openSession();

  System.out.println(openSession);

*//3.获取接口的实现对象*

  HydaMapper mapper = openSession.getMapper(HydaMapper.class);

*//4.调用增删改查方法*

  try {

   Hyda hyda = new Hyda();

   hyda.setId(1720L);

   Hyda selectOne = mapper.selectHyda(1720L);

   System.out.println(selectOne);

  }finally {

   openSession.close();

  }

 }

   @Test

   public void testQueryHyda() throws IOException {

*//1.获取SqlSessionFactory对象*

    SqlSessionFactory sqlSessionFactory = getSqlSessionFactory();

*//2.获取sqlSession对象*

    SqlSession openSession = sqlSessionFactory.openSession();

    System.out.println(openSession);

*//3.获取接口的实现对象*

    HydaMapper mapper = openSession.getMapper(HydaMapper.class);

*//4.调用增删改查方法*

    try {

     Hyda hyda = new Hyda();

    hyda.setHyqc("中钢");

     List<Hyda> list = mapper.queryHyda(hyda);

     System.out.println(list);

    }finally {

     openSession.close();

    }

   }

private static SqlSessionFactory getSqlSessionFactory() throws IOException {

 String resource = "conf/mybatis-conf.xml";

 Reader reader = Resources.getResourceAsReader(resource);

 SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(reader);

 return sqlSessionFactory;

}

}

# Springboot整合Redis

1. **添加依赖**

<dependency>

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

<artifactId>spring-boot-starter-data-redis</artifactId> </dependency>

1. **配置redis**

spring:

redis:

database: 0

password: redis@958958

host: 1.116.226.147

port: 6378

timeout: 5000