

注解

引入依赖

```
<dependency>
  <groupId>org.projectlombok</groupId>
  <artifactId>lombok</artifactId>
  <version>1.18.4</version>
  <scope>provided</scope>
</dependency>
```

常用的几个注解：

@Data：注在类上，提供类的get、set、equals、hashCode、canEqual、toString方法

@AllArgsConstructor：注在类上，提供类的全参构造

@NoArgsConstructor：注在类上，提供类的无参构造

@Setter：注在属性上，提供 set 方法

@Getter：注在属性上，提供 get 方法

@EqualsAndHashCode：注在类上，提供对应的 equals 和 hashCode 方法

@Log4j/@Slf4j：注在类上，提供对应的 Logger 对象，变量名为 log

@Autowired自动装配详解

(1) @Autowired自动装配，默认优先按照类型取IOC容器中寻找对应的组件

(2) 如果有多个相同类型的组件，再将属性的名称作为组件的id去容器中查找

(4) @Autowired配合@Qualifier使用，使用注解@Qualifier可以指定需要装配组件的id

当标注的属性是接口时，其实注入的是这个接口的实现类，如果这个接口有多个实现类，只使用 @Autowired就会报错，因为它默认是根据类型找，然后就会找到多个实现类bean，所有就不知道要注入哪个。然后它会根据属性名去找。所以如果有多个实现类可以配合@Qualifier(value="类名")来使用（是根据名称来进行注入的）可以参考 spring@Autowired注解的注入规则

@ControllerAdvice

```
package com.example.hungry.common;

import lombok.extern.slf4j.Slf4j;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.bind.annotation.RestController;

import java.sql.SQLIntegrityConstraintViolationException;

/*
全局异常处理
*/
```

```
/*首先, @ControllerAdvice本质上是一个@Component, 因此也会被当成组建扫描。为那些声明了
(@ExceptionHandler、@InitBinder 或 @ModelAttribute注解修饰的) 方法的类而提供的
专业化的@Component, 以供多个 Controller类所共享。*/
@ControllerAdvice(annotations = {RestController.class, Controller.class})
@Slf4j
@ResponseBody
public class GlobalExceptionHandler {
    //异常处理
    @ExceptionHandler(SQLIntegrityConstraintViolationException.class)
    public R<String> exceptionHandler(SQLIntegrityConstraintViolationException
e)
    {
        log.info(e.getMessage());
        return R.error("创建失败, 已有该员工");
    }
}.
```

文档

官方英文文档

<https://docs.spring.io/spring-boot/docs/current/reference/html/index.html>

很全的中文

<https://www.yuque.com/atguigu/springboot>

maven整合了所有jar包, springboot整合了所有框架

开发原则--程序要求高内聚低耦合

SpringBoot

整合Spring技术栈的一站式框架

简化Spring技术栈的快速开发脚手架

SpringBoot的特点:

- ①为基于Spring的开发提供更快的入门体验
- ②开箱即用, 没有代码生成, 也无需XML配置。同时也可以修改默认值来满足特定的需求。
- ③提供了一些大型的项目中常见的非功能性特性, 如嵌入式服务器、安全、指标、健康检测、外部配置等
- ④SpringBoot不是对Spring功能上的增强, 而是提供了一种快速使用Spring的方式。
- ⑤不需要导入太多外部jar, 帮我们打包好了, 导入一个就好了

什么是微服务

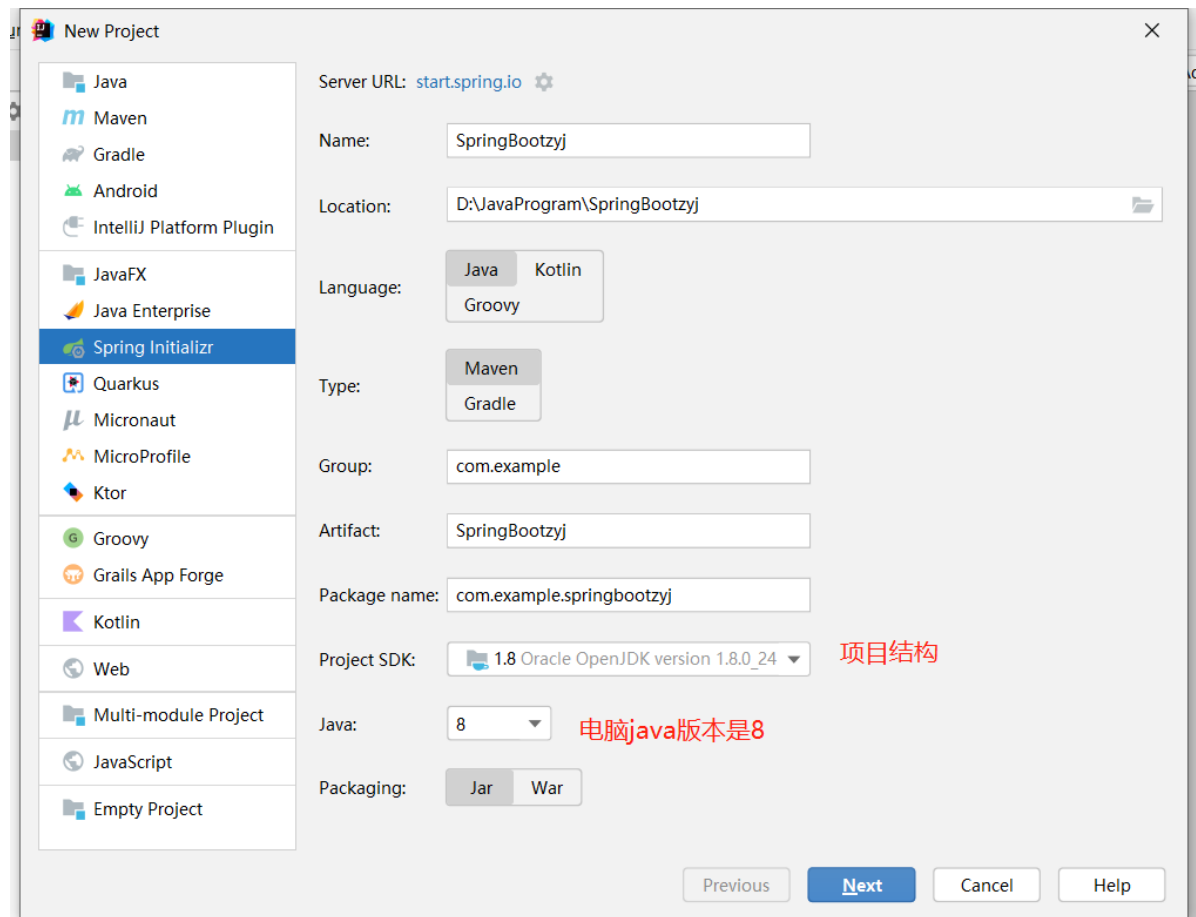
一个应用拆分成一组小型服务

创建springboot

官网下载导入idea

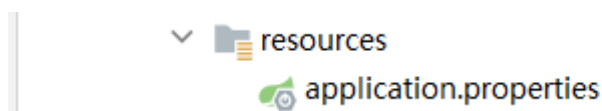
idea创建springboot项目

比maven依赖parent还有依赖环境好，因为项目框架帮我们构建好了



统一的配置文件

所有的配置都可以在这方便实现



项目流程

创建maven导入场景依赖

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.7.1</version>
</parent>
```

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

编写主程序

```
package com.atguigu.boot;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
/*
注解是为了告诉这是一个SpringBoot应用
*/
@SpringBootApplication
public class MainApplication {
    public static void main(String[] args) {
        SpringApplication.run(MainApplication.class,args);
    }
}
```

编写业务

测试

直接运行main方法

简化配置

写在resources包中application.properties

简化部署

```
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
      <version>2.4.3</version>
    </plugin>
  </plugins>
</build>
```

把项目打成jar包，直接在目标服务器执行即可

注意取消cmd的快速编辑模式

依赖管理

父项目做依赖管理，几乎声明了所有开发中常用的依赖的jar包

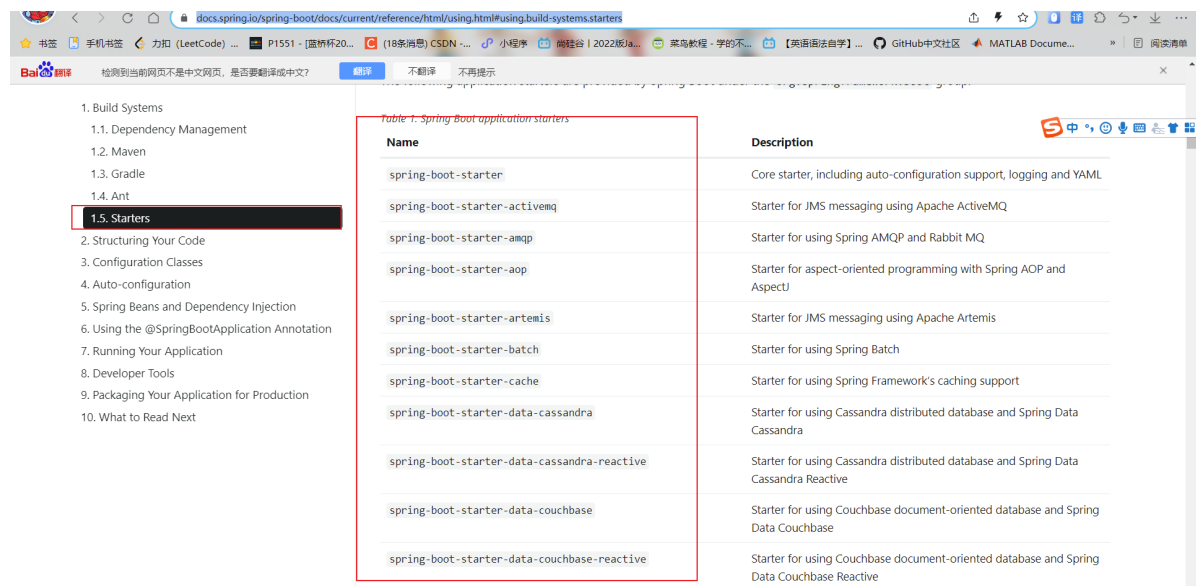
```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.7.1</version>
</parent>
```

无需关注版本号，自动版本仲裁，可以修改版本号

引入场景

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

<https://docs.spring.io/spring-boot/docs/current/reference/html/using.html#using.build-systems.starters>




The screenshot shows the Spring Boot documentation page for starters. The left sidebar lists the table of contents, with '1.5. Starters' highlighted. The main content area displays 'Table 1. Spring Boot application starters' with a table listing various starters and their descriptions.

Name	Description
spring-boot-starter	Core starter, including auto-configuration support, logging and YAML
spring-boot-starter-activemq	Starter for JMS messaging using Apache ActiveMQ
spring-boot-starter-amqp	Starter for using Spring AMQP and Rabbit MQ
spring-boot-starter-aop	Starter for aspect-oriented programming with Spring AOP and AspectJ
spring-boot-starter-artemis	Starter for JMS messaging using Apache Artemis
spring-boot-starter-batch	Starter for using Spring Batch
spring-boot-starter-cache	Starter for using Spring Framework's caching support
spring-boot-starter-data-cassandra	Starter for using Cassandra distributed database and Spring Data Cassandra
spring-boot-starter-data-cassandra-reactive	Starter for using Cassandra distributed database and Spring Data Cassandra Reactive
spring-boot-starter-data-couchbase	Starter for using Couchbase document-oriented database and Spring Data Couchbase
spring-boot-starter-data-couchbase-reactive	Starter for using Couchbase document-oriented database and Spring Data Couchbase Reactive

父项目无需关注版本号，自动版本仲裁，可以修改版本号

需点击

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.7.1</version>
</parent>
```



```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <project xmlns="http://maven.apache.org/POM/4.0.0" xsi:schemaLocat
3      <modelVersion>4.0.0</modelVersion>
4      <parent>
5          <groupId>org.springframework.boot</groupId>
6          <artifactId>spring-boot-dependencies</artifactId>
7          <version>2.7.1</version>

```

查看所修改的东西的key如

```

- boot) x spring-boot-starter-parent-2.7.1.pom x spring-boot-dependencies-2.7.1.pom x
161 <maven-surefire-plugin.version>2.22.2</maven-surefir
162 <maven-war-plugin.version>3.3.2</maven-war-plugin.version>
163 <micrometer.version>1.9.1</micrometer.version>
164 <mockito.version>4.5.1</mockito.version>
165 <mongodb.version>4.6.1</mongodb.version>
166 <mssql-jdbc.version>10.2.1.jre8</mssql-jdbc.version>
167 <mysql.version>8.0.29</mysql.version>
168 <nekohtml.version>1.9.22</nekohtml.version>
169 <neo4j-java-driver.version>4.4.6</neo4j-java-driver.version>

```

在当前项目重写配置

```

<properties>
<mysql.version>5.1.43</mysql.version>
</properties>

```

自动配置

自动配好tomcat

依赖管理中引入场景就引入tomcat依赖

```

m pom.xml (SpringBoot) x spring-boot-starter-web-2.7.1.pom x
Q tomcat
52 <artifactId>spring-boot-starter-json</artifactId>
53 <version>2.7.1</version>
54 <scope>compile</scope>
55 </dependency>
56 <dependency>
57 <groupId>org.springframework.boot</groupId>
58 <artifactId>spring-boot-starter-tomcat</artifactId>
59 <version>2.7.1</version>
60 <scope>compile</scope>
61 </dependency>

```

自动配好SpringMVC常用组件（功能）

自动配好Web常见功能如字符编码问题

默认的包规则：主程序所在的包及其下面子包里面的组件都会被默认扫描，如果想要改变扫描路径，主程序修改注解的属性

```
@SpringBootApplication(scanBasePackages = "?")
public class MainApplication {
    public static void main(String[] args) { SpringApplication.run(MainApplication.class, args); }
}
```

也可以用

```
@ComponentScan(basePackages = "?")
```

各种配置拥有默认值

默认配置最终都映射到MultipartProperties

配置文件的值最终会绑定每一个类上，这个类会在容器中创建对象

获取容器

```
//获取容器
ConfigurableApplicationContext run = SpringApplication.run(MainApplication.class, args);
```

Bean

配置类里面默认@Bean标注在方法上给容器注册组件，默认单实例，方法名作为组件的id，返回类型就是组件类型。

配置类本身也是组件，我们调用配置类获取的是代理对象！！

@Configuration(proxyBeanMethods=false) //配置类中方法获取的bean的对象是不是代理对象

Full(proxyBeanMethods = true):proxyBeanMethods参数设置为true时即为：Full 全模式。该模式下注入容器中的同一个组件无论被取出多少次都是同一个bean实例，即单实例对象，在该模式下SpringBoot每次启动都会判断检查容器中是否存在该组件

Lite(proxyBeanMethods = false):proxyBeanMethods参数设置为false时即为：Lite 轻量级模式。该模式下注入容器中的同一个组件无论被取出多少次都是不同的bean实例，即多实例对象，在该模式下SpringBoot每次启动会跳过检查容器中是否存在该组件

这关乎依赖关系

当在你的同一个Configuration配置类中，注入到容器中的bean实例之间有依赖关系时，建议使用Full全模式

当在你的同一个Configuration配置类中，注入到容器中的bean实例之间没有依赖关系时，建议使用Lite轻量级模式，以提高springboot的启动速度和性能

意思就是配置类中有获取a对象和获取b对象，a对象中有属性b对象。若希望配置类中b对象和a对象中属性b对象毫无关系则要false，若是指向同一个对象则用true。本质就是是不是单实例

组件的层次

@Component业务特殊组件层

@Service用于服务层，处理业务逻辑

@Controller用于呈现层

@Repository用于持久层，数据库访问层

四个实现功能一样，用于不同开发程序层次

@Import

带有@Configuration的配置类(4.2 版本之前只可以导入配置类，4.2版本之后 也可以导入 普通类!!!)，还有其它，不拓展

等同于完全注解开发的@Import({?.class,...}), 默认组件的名字是全类名!!!

@Conditional有前提的

范围:用在类上面或者@Bean

针对前提判断是否创建bean,用在类则判断要不要创建@Bean

```
@Conditional org.springframework.context.annotation
@ConditionalOnBean org.springframework.boot.autoconfigure...
@ConditionalOnClass org.springframework.boot.autoconfigure...
@ConditionalOnCloudPlatform org.springframework.boot.au...
@ConditionalOnDefaultWebSecurity org.springframework.bo...
@ConditionalOnEnabledResourceChain org.springframework...
@ConditionalOnExpression org.springframework.boot.autoc...
@ConditionalOnGraphQLSchema org.springframework.boot.au...
@ConditionalOnJava org.springframework.boot.autoconfigure...
@ConditionalOnJndi org.springframework.boot.autoconfigure...
@ConditionalOnMissingBean org.springframework.boot.auto...
@ConditionalOnMissingClass org.springframework.boot.out...
```

如果有bean是getB组件就创建getA组件，

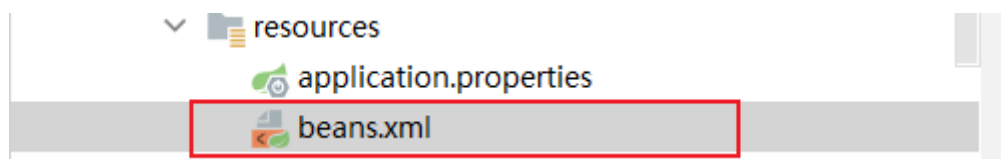

```

@Configuration
@ConditionalOnBean(name = "getB")
public class Conf {
    @ConditionalOnBean(name = "getB")
    @Bean
    public testConfClassA getA()
    {
        return new testConfClassA();
    }
    @Bean
    public testConfClassB getB()
    {
        return new testConfClassB();
    }
}

```

还有@ConditionalOnMissingBean没有上面bean才干吗，等等一些conditional

@ImportResource引入xml配置



springboot环境识别不了xml文件，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，要用到@ImportResource("classpath:beans.xml")导入spring的配置文件，在配置类写上就好了

```

@Configuration
//导入配置文件bean.xml
@ImportResource("classpath:bean.xml")
public class configuration {
}

```

配置绑定

场景一：@Component,@Service,@Controller,@Repository //将类定义为一个bean的注解

场景二：@Configuration//配置类的@Bean

```

@Configuration
public class DataSourceConfig {
    @Bean(name = "primaryDataSource")
    @ConfigurationProperties(prefix="spring.datasource.primary")
    public DataSource primaryDataSource() {
    }
}

```

场景三：@ConfigurationProperties //注解到普通类

```
@ConfigurationProperties(prefix = "user1")
public class User {
    private String name;
    // 省略getter/setter方法!!!!!!! 前面两种都需要set方法
}
```

然后再通过@EnableConfigurationProperties定义为Bean。

```
@SpringBootApplication
@EnableConfigurationProperties({User.class})
public class Application {
    public static void main(String[] args) throws Exception {
        SpringApplication.run(Application.class, args);
    }
}
```

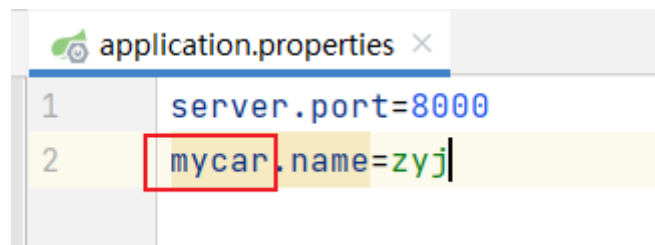
prefix: 前缀代码

ConfigurationProperties配置属性

@ConfigurationProperties

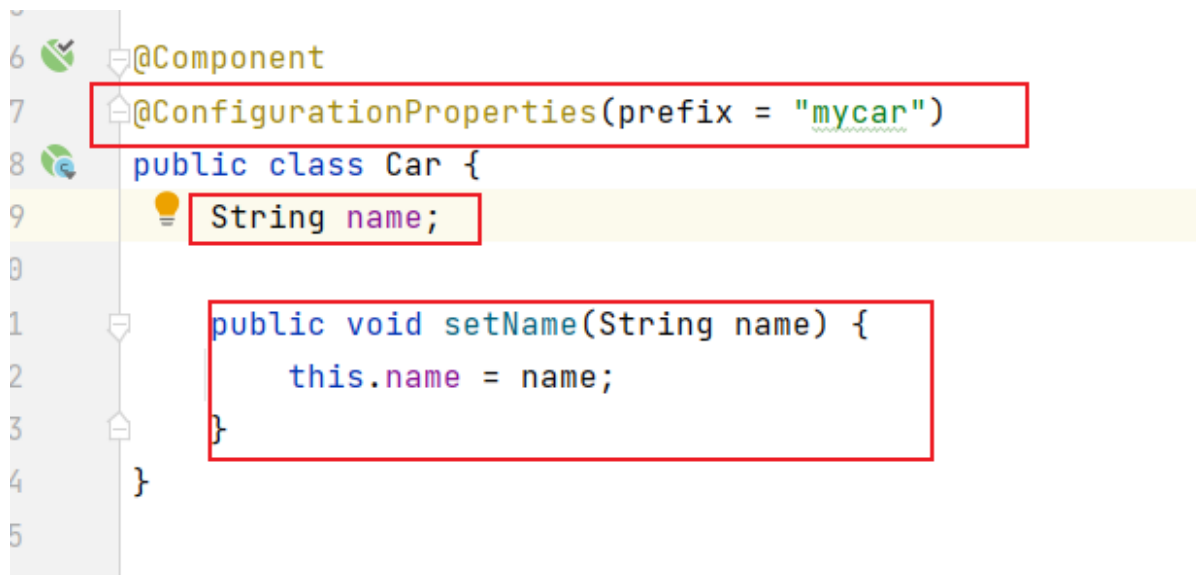
即给类的属性初始化值

首先application.properties文件设置初始值，前置代号.属性



The screenshot shows a text editor window titled 'application.properties'. It contains two lines of configuration: '1 server.port=8000' and '2 mycar.name=zyj'. The second line is highlighted with a yellow background, and 'mycar.name' is enclosed in a red box.

类中要有注解@ConfigurationProperties(prefix = "mycar"), 属性prefix的属性值要是标签id, 以及要有set方法



The screenshot shows the code for the 'Car' class. It is annotated with '@Component' and '@ConfigurationProperties(prefix = "mycar")'. The 'name' attribute is declared as 'String name;'. A 'setName' method is implemented: 'public void setName(String name) { this.name = name; }'. The '@ConfigurationProperties' annotation, the 'name' attribute declaration, and the 'setName' method are all highlighted with red boxes.

第二种方法是在配置类开启配置绑定功能，主要是针对没有@Component

```
@EnableConfigurationProperties(类名.class)
```

自动装配原理

@SpringBootApplication中有三个注解

```
@SpringBootApplication
@EnableAutoConfiguration
@ComponentScan(
    excludeFilters = {@Filter(
        type = FilterType.CUSTOM,
        classes = {TypeExcludeFilter.class}
    ), @Filter(
        type = FilterType.CUSTOM,
        classes = {AutoConfigurationExcludeFilter.class}
    )}
)
public @interface SpringBootApplication
{
    @SpringBootApplication@EnableAutoConfiguration@ComponentScan(
        excludeFilters = {@Filter( type = FilterType.CUSTOM, classes =
        {TypeExcludeFilter.class}), @Filter( type = FilterType.CUSTOM, classes =
        {AutoConfigurationExcludeFilter.class})}
    )
}
```

@SpringBootApplication中有@Configuration代表当前是一个配置类

```
@Configuration
public @interface SpringBootApplication {
}
```

@ComponentScan指定扫描

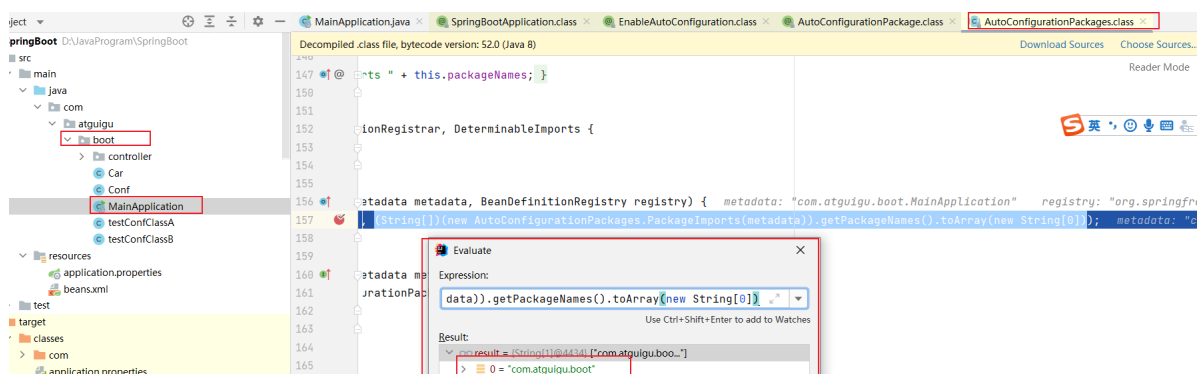
@AutoConfigurationPackage

@Import({AutoConfigurationImportSelector.class})

```
public @interface EnableAutoConfiguration {
}
```

@EnableAutoConfiguration中有@AutoConfigurationPackage（这个注解有

@Import({Registrar.class}) 利用Registrar给容器中导入一系列组件，这里标红地方体现为什么之扫描主程序所在的包！！！！



@Import({AutoConfigurationImportSelector.class})

AutoConfigurationImportSelector类中有方法getAutoConfigurationEntry的

```
List configurations = this.getCandidateConfigurations(annotationMetadata, attributes)
```

获取配置给容器中批量导入一些组件

加载所有场景的自动装配，但最终按照条件装配@Conditional规则，最终会按需配置

总结：

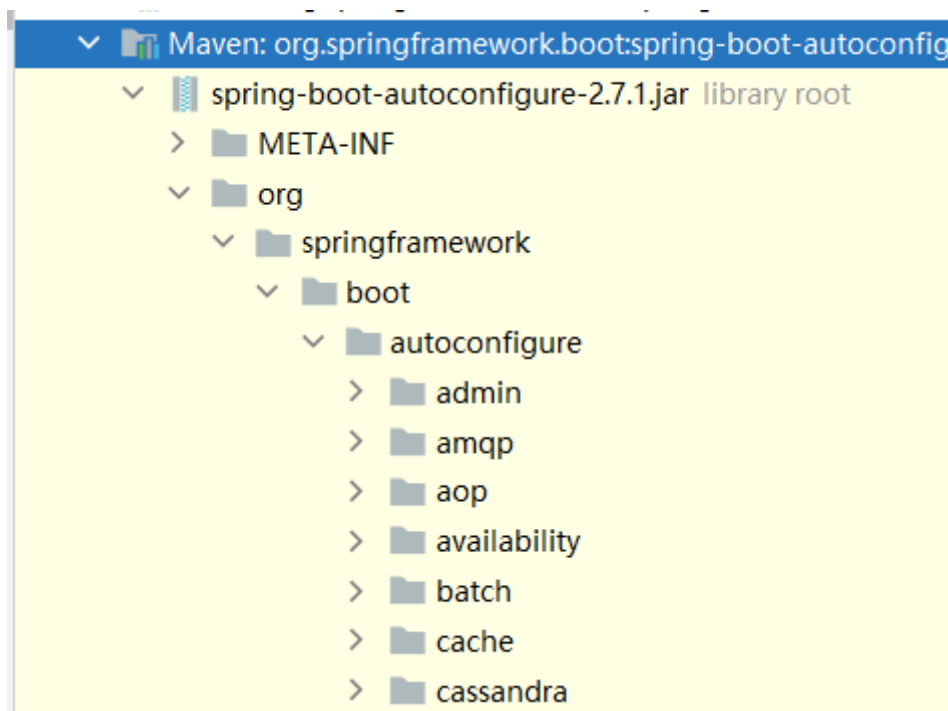
springboot先加载所有的自动配置类xxxxAutoConfiguration

每个自动：配置类按照条件进行生效，默认都会绑定配置文件指定的值，xxxxProperties里面拿，xxxxProperties和配置文件进行绑定

生效的配置类就会给容器中装配很多组件

只要容器中有这些组件，相当于这些功能就有了

定制化配置：用户直接自己@Bean替换底层组件，用户去看组件获取的配置文件什么值去修改



开发流程

1引入场景依赖

2查看自动配置：

自己分析或者引入场景对应的自动配置一般都生效

配置文件中debug=true开启自动配置报告，运行后看，negative（不生效）和postive（生效）

3是否需要修改

参照文档修改配置项

自定义加入或者替换组件@Bean@Component

lombok插件

这个知识可以帮我们在编译代码时候自动帮我们写set，get，无参构造等方法。想偷懒的话可以学一下这个插件

小技巧

devtools 引入develop tools开发者工具，修改代码后ctrl+f9则可以看修改的效果这是自动重启（对修改过的东西进行重启）不是热部署

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

核心技术

配置文件名是固定的： Application.[properties](#)、Application.yml yaml/yml

优先级是application.prope > application.yaml，两个同时生效

基础语法

yaml有更简洁的语法，更少的外存

大小写敏感

缩进表示层级关系，缩进空格数不重要，相同层次元素左对齐即可

#注释

字符串无需加引号，双引号不会转义，即\n会生效，单引号会转义即\n不会换行，而是变成字符串

字面量

key-value是k: v（中间有空格！！！！）

对象

集合形式是k: {k1: v1,...}或者

k:

 k1: v1

 k2: v2 注意对齐！！

数组

k: [v1,v2,v3]

k:

-- v1

-- v2

注意是一个横杆，这里写一个横杆会变成其它东西，然后要空格！！！！

例子

```
package com.atguigu.boot;

import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.stereotype.Component;

import java.util.*;
//只要这里要配置属性！！
@ConfigurationProperties(prefix = "person")
@Component
public class Person {
    String userName;
    Date date;
    Car car;
    Boolean boss;
    Integer age;
    String[] interests;
    List<String> animal;
    Map<String, Object> score;
    Set<Double> salarys;
    Map<String, List<Car>> allCars;

    Person() {}

    public Person(String userName, Car car, Boolean boss, Integer age, String[]
interests, List<String> animal, Map<String, Object> score, Set<Double> salarys,
Map<String, List<Car>> allCars) {
        this.userName = userName;
        this.car = car;
        this.boss = boss;
        this.age = age;
        this.interests = interests;
        this.animal = animal;
        this.score = score;
        this.salarys = salarys;
        this.allCars = allCars;
    }

    public String getUserName() {
        return userName;
    }

    public void setUserName(String userName) {
        this.userName = userName;
    }

    public Car getCar() {
        return car;
    }

    public void setCar(Car car) {
        this.car = car;
    }
}
```

```

public Boolean getBoss() {
    return boss;
}

public void setBoss(Boolean boss) {
    this.boss = boss;
}

public Integer getAge() {
    return age;
}

public void setAge(Integer age) {
    this.age = age;
}

public String[] getInterests() {
    return interests;
}

public void setInterests(String[] interests) {
    this.interests = interests;
}

public List<String> getAnimal() {
    return animal;
}

public void setAnimal(List<String> animal) {
    this.animal = animal;
}

public Map<String, Object> getScore() {
    return score;
}

public void setScore(Map<String, Object> score) {
    this.score = score;
}

public Set<Double> getSalarys() {
    return salarys;
}

public void setSalarys(Set<Double> salarys) {
    this.salarys = salarys;
}

public Map<String, List<Car>> getAllCars() {
    return allCars;
}

public void setAllCars(Map<String, List<Car>> allCars) {
    this.allCars = allCars;
}

@Override
public String toString() {

```

```

        return "Person{" +
            "userName='" + userName + '\'' +
            ", car=" + car +
            ", boss=" + boss +
            ", age=" + age +
            ", interests=" + Arrays.toString(interests) +
            ", animal=" + animal +
            ", score=" + score +
            ", salarys=" + salarys +
            ", allCars=" + allCars +
            '}';
    }
}

```

```

package com.atguigu.boot;

import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.stereotype.Component;

@Component
public class Car {
    String name;
    Integer age;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Integer getAge() {
        return age;
    }

    public void setAge(Integer age) {
        this.age = age;
    }

    Car(){}
    public Car(String name, Integer age) {
        this.name = name;
        this.age = age;
    }

    @Override
    public String toString() {
        return "Car{" +
            "name='" + name + '\'' +
            ", age=" + age +
            '}';
    }
}

```

application.yaml

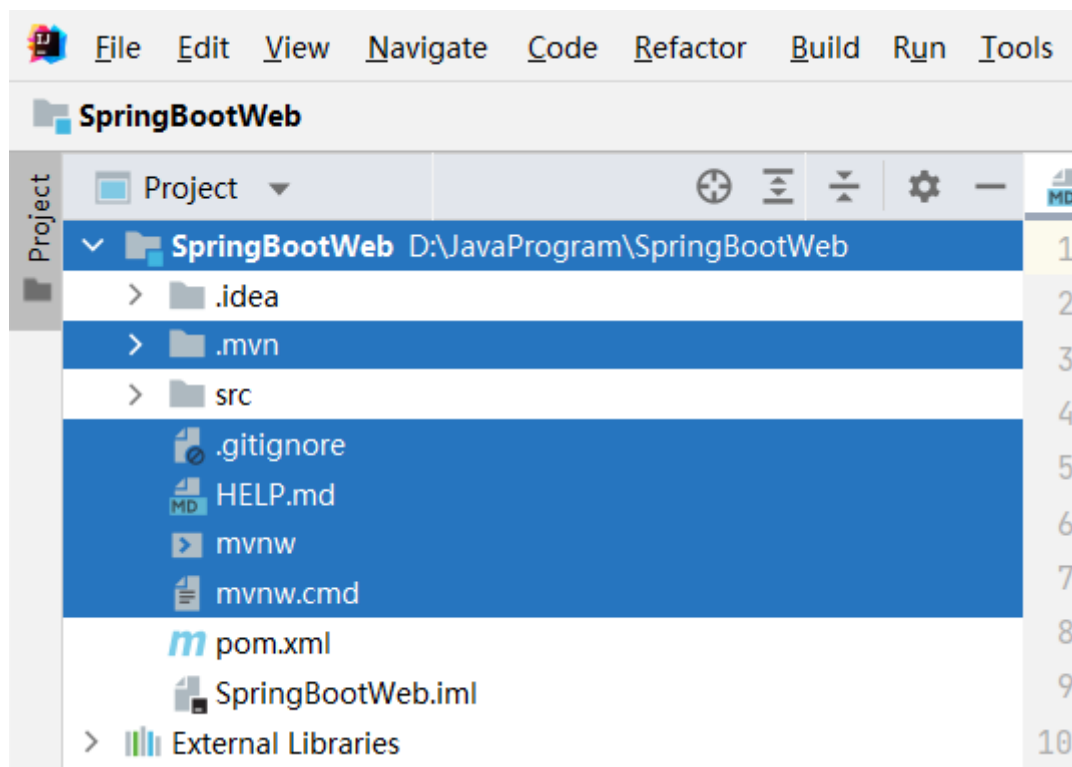

```
person:
  userName: zhangsan
  boss: true
  birth: 2019/12/9
  age: 18
# interests: [篮球, 足球]
interests:
  - 篮球
  - 足球
  - 18
animal: [阿猫, 阿狗]
# score:
#   english: 80
#   math: 90
score: {english: 80, math: 90}
salarys:
  - 999
  - 888
Car:
  name: 阿猫
  age: 12
allCars:
  sick:
    - {name: 猫1, age: 20}
    - name: 猫2
      age: 18
```

为了yaml编写时候有提示要引入依赖

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

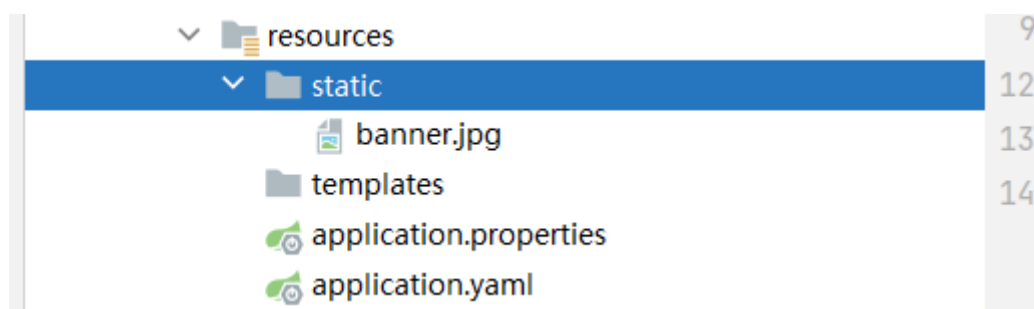
web开发

没用的东西删掉



静态资源访问

类路径src/resources下的名叫static | | public | | resources | | META-INF/resources都可以访问到静态资源



原理：静态映射

请求进来，先去找Controller能不能处理，不能处理所有请求都交给静态资源处理器，再找不到就404

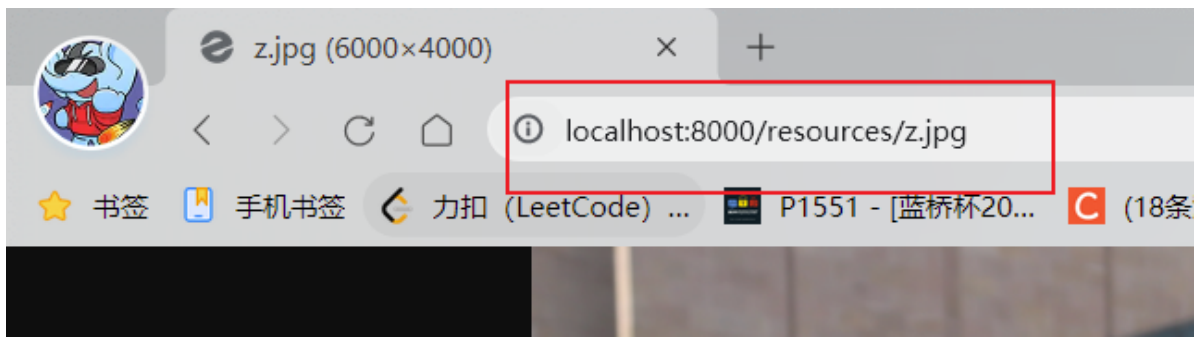
前缀

静态资源默认无前缀，即/**

若想加前缀，如前缀/resources

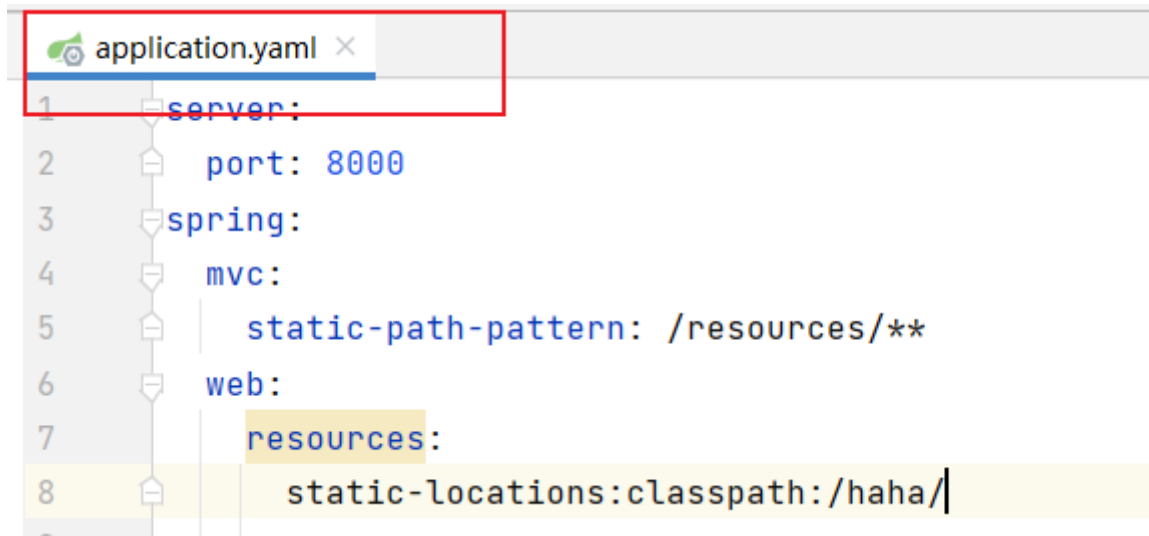
可以下操作，则/resources以下所有0或者更多的目录都可以作为静态资源





访问路径

修改访问路径



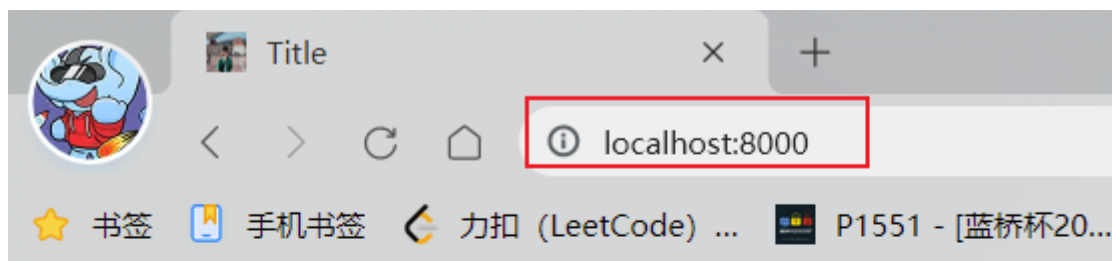
这种情况则静态资源要放在resources/haha/..... 即前缀/resources再加访问路径

多个路径的话则用数组[classpath: /haha/, classpath: /papa/, , ,]

欢迎页index.html，固定的，是在static下

不可以配置静态资源访问前缀，否则导致不能被默认访问

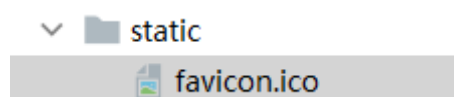




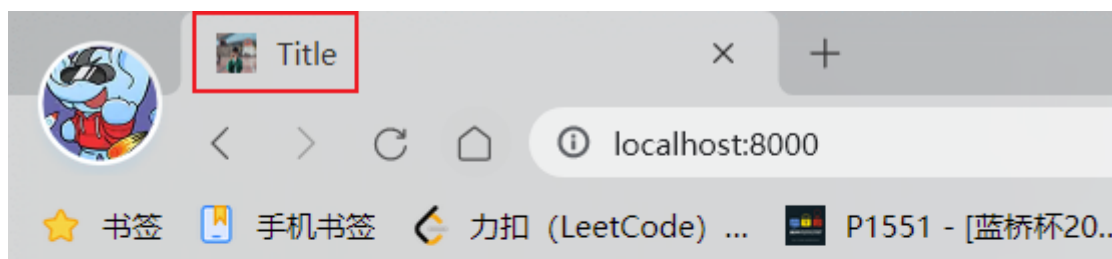
atguigu:欢迎

默认图标

不可以配置静态资源访问前缀，否则导致图标不能被默认访问



命名为favicon.ico



atguigu:欢迎

请求方式利用rest风格

原先

原先springmvc需要利用到HiddenHttpMethodFilter隐藏http方法过滤器

在web.xml中注册HiddenHttpMethodFilter

```
<filter>
  <filter-name>HiddenHttpMethodFilter</filter-name>
  <filter-class>org.springframework.web.filter.HiddenHttpMethodFilter</filter-
class>
</filter>
<filter-mapping>
  <filter-name>HiddenHttpMethodFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

- 注:

目前为止，SpringMVC中提供了两个过滤器：CharacterEncodingFilter和HiddenHttpMethodFilter

在web.xml中注册时，必须先注册CharacterEncodingFilter，再注册HiddenHttpMethodFilter

原因:

在 CharacterEncodingFilter 中通过 request.setCharacterEncoding(encoding) 方法设置字符集的

request.setCharacterEncoding(encoding) 方法要求前面不能有任何获取请求参数的操作

而 HiddenHttpMethodFilter 恰恰有一个获取请求方式的操作:

```
String paramValue = request.getParameter(this.methodParam); paramValue =
request.getParameter(this.methodParam);
```

通俗易懂编码肯定得先获取!!!

HiddenHttpMethodFilter 处理put和delete请求的条件:

a>当前请求的请求方式必须为post!!!

b>当前请求必须传输请求参数_method!!!!

满足以上条件，**HiddenHttpMethodFilter** 过滤器就会将当前请求的请求方式转换为请求参数method的值，因此请求参数method的值才是最终的请求方式

现在

application.yml中

```
spring:
  mvc:
    hiddenmethod:
      filter:
        enabled: true
        #默认enabled是false
```

注意前端value值大小写都可以，但注意需要请求方式中name一定要_method否则后端无法识别

```
<input type="hidden" name="_method" value="post">
```

@MatrixVariable注解 矩阵变量

该知识为rest风格关于路径的知识

```
//      /cars/ss;low=23;brand=byd,ad,bc
@RequestMapping("/cars/{sales}")
@ResponseBody
public String cars(
    @MatrixVariable("low") int low,      值为23
    @MatrixVariable("brand") String brand,  值为byd, ad, bc
    @PathParam("sales")String sales,      值为null
    @PathVariable("sales")String sales1)  值为ss
{
    return null;
}
```

thymeleaf

原先

springmvc的配置文件中

配置Thymeleaf视图解析器bean

```
<!--      配置Thymeleaf视图解析器-->
    <bean id="viewResolver"
class="org.thymeleaf.spring5.view.ThymeleafViewResolver">
<!--      视图解析器优先级-->
    <property name="order" value="1"/>
<!--      编码-->
    <property name="characterEncoding" value="UTF-8"/>
<!--      模板-->
    <property name="templateEngine">
        <bean class="org.thymeleaf.spring5.SpringTemplateEngine">
            <property name="templateResolver">
                <bean
class="org.thymeleaf.spring5.templateresolver.SpringResourceTemplateResolver">
<!--      视图前缀-->
                    <property name="prefix" value="/WEB-INF/templates/" />
<!--      视图后缀-->
                    <property name="suffix" value=".html"></property>
<!--      模板模型-->
                    <property name="templateMode" value="HTML5"></property>
<!--      编码-->
                    <property name="characterEncoding" value="UTF-8"/>
                </bean>
            </property>
        </bean>
    </property>
</bean>
```

现在

导入依赖就能用

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

自动渲染resource/templates下的html文件，静态资源存放在resource/static中

上下文

```
server:
  servlet:
    context-path: /Summer
```

则<http://localhost:8080/Summer/>

原先是直接<http://localhost:8080>

或者<http://localhost:8080/> 加个斜杆

拦截器

原先

SpringMVC中的拦截器用于拦截控制器方法的执行

SpringMVC中的拦截器需要实现HandlerInterceptor，必须在SpringMVC的配置文件中配置。

```
//ctrl+o重写三个方法
public class Interceptor implements HandlerInterceptor {
    @Override
    //preHandle: 控制器方法执行之前执行preHandle(), 其boolean类型的返回值表示是否拦截或放行, 返回true为放行, 即调用控制器方法; 返回false表示拦截, 即不调用控制器方法!!!!
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {
        return HandlerInterceptor.super.preHandle(request, response, handler);
    }

    @Override
    //postHandle: 控制器方法执行之后执行postHandle()
    public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler, ModelAndView modelAndView) throws Exception {
        HandlerInterceptor.super.postHandle(request, response, handler, modelAndView);
    }

    @Override
    //afterCompletion: 处理完视图和模型数据, 渲染视图完毕之后执行afterCompletion()
    public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex) throws Exception {
        HandlerInterceptor.super.afterCompletion(request, response, handler, ex);
    }
}
```

```

<bean class="com.Interceptor"></bean>
<ref bean="firstInterceptor"></ref>
<!-- 以上两种配置方式都是对DispatcherServlet所处理的所有的请求进行拦截 -->
<mvc:interceptor>
    <mvc:mapping path="/*"/>
    <mvc:exclude-mapping path="/testRequestEntity"/>
    <ref bean="firstInterceptor"></ref>
</mvc:interceptor>
<!--
    以上配置方式可以通过ref或bean标签设置拦截器，通过mvc:mapping设置需要拦截的请求，通过
    mvc:exclude-mapping设置需要排除的请求，即不需要拦截的请求
-->

```

多个拦截器的执行顺序

a>若每个拦截器的preHandle()都返回true

此时多个拦截器的执行顺序和拦截器在SpringMVC的配置文件的配置顺序有关：

preHandle()会按照配置的顺序执行，而postHandle()和afterComplation()会按照配置的反序执行

b>若某个拦截器的preHandle()返回了false

preHandle()返回false和它之前的拦截器的preHandle()都会执行，postHandle()都不执行，返回false的拦截器之前的拦截器的afterComplation()会执行

现在

先写好拦截器

```

//ctrl+o重写三个方法
public class Interceptor implements HandlerInterceptor {
    @Override
    //preHandle: 控制器方法执行之前执行preHandle(), 其boolean类型的返回值表示是否拦截或放行, 返回true为放行, 即调用控制器方法; 返回false表示拦截, 即不调用控制器方法!!!!
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {
        return HandlerInterceptor.super.preHandle(request, response, handler);
    }

    @Override
    //postHandle: 控制器方法执行之后执行postHandle()
    public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler, ModelAndView modelAndView) throws Exception {
        HandlerInterceptor.super.postHandle(request, response, handler, modelAndView);
    }

    @Override
    //afterComplation: 处理完视图和模型数据, 渲染视图完毕之后执行afterComplation()
    public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex) throws Exception {
        HandlerInterceptor.super.afterCompletion(request, response, handler, ex);
    }
}

```

然后


```
//配置类
@Configuration                                //实现WebMvc配置
public class AdmitConfig implements WebMvcConfigurer {
    @Override
    //重写拦截器方法
    public void addInterceptors(InterceptorRegistry registry) {
        //拦截器注册对象添加拦截器                //拦截器
        registry.addInterceptor(new testInterceptor())
            .addPathPatterns("/**") //拦截路径
            .excludePathPatterns("/", "/login"); //不拦截的路径
    }
}
```

注意如果拦截所有路径则静态资源路径页会被拦截

如果觉得用excludePathPatterns("/static/**")

放行静态资源路径就错了，因为在url中是不会通过/public请求的 而是直接通过/css 或者是 /js 访问 因此我们需要做的就是将这些拦截排除掉 ,和/static是没有关系的!

所以只能static下的文件css, js等采用

excludePathPatterns("/css/**")

excludePathPatterns("/js/**")

不需要/statics开头

解决拦截器拦截静态资源的问题：

https://blog.csdn.net/qq_40436854/article/details/91576009?ops_request_misc=&request_id=&biz_id=102&utm_term=springboot%E6%8B%A6%E6%88%AA%E5%99%A8%E6%8B%A6%E6%88%AA%E9%9D%99%E6%80%81%E8%B5%84%E6%BA%90%E9%97%AE%E9%A2%98&utm_medium=distrib ute.pc_search_result.none-task-blog-2~all~sobaiduweb~default-3-91576009.142^v44^new_blog_pos_by_title&spm=1018.2226.3001.4187

文件上传

原先

a>添加依赖：

```
<!-- https://mvnrepository.com/artifact/commons-fileupload/commons-fileupload -->
<dependency>
    <groupId>commons-fileupload</groupId>
    <artifactId>commons-fileupload</artifactId>
    <version>1.3.1</version>
</dependency>
```

b>在SpringMVC的配置文件中添加配置：

```
<!--必须通过文件解析器的解析才能将文件转换为MultipartFile对象-->
<bean id="multipartResolver"
    class="org.springframework.web.multipart.commons.CommonsMultipartResolver">
</bean>
```

注意MultipartFile photo

也可以有数组MultipartFile[] photo

现在

直接用，要diy的就在配置文件里面

```
spring.servlet.multipart.      开头
```

```
spring:
  servlet:
    multipart:
      enabled: true
```

异常处理机制

springboot默认处理异常机制

默认浏览器，返回一个默认的错误页面（通过浏览器的请求头来返回页面）

非浏览器则响应一个json

也可以自定义页面

会返回数据

timestamp

status

error

message

path

1. 有模板引擎的情况下

error/状态码 -- 将错误页面命名为 错误状态码.html 放在模板引擎文件夹里面的error文件夹下 (/templates/error/xxx.html)，发生此状态码的错误就会来到 对应的页面。使用4xx和5xx作为错误页面的文件名来匹配这种类型的所有错误，精确状态码优先。

2. 没有模板引擎（模板引擎找不到这个错误页面）

SpringBoot会去静态资源目录（static/error/状态码.html）下面找错误代码命名的页面。

原生的三大组件

Javaweb的三大组件分别是：Servlet、Filter（过滤器）、Listener（监听器）。

要在主类使用原生的组件需要用@ServletComponentScan注解扫描

```

@ServletComponentScan(basePackages = "com/example/junior/servlet")
@SpringBootApplication
public class JuniorApplication {

    public static void main(String[] args) {
        ConfigurableApplicationContext run =
SpringApplication.run(JuniorApplication.class, args);
    }

}

```

```

package com.example.junior.servlet;

import javax.servlet.annotation.WebServlet;

@WebServlet("/my")
public class servletTest {
}

```

原生的过滤器也需要在主类使用@ServletComponentScan注解扫描

过滤器

```

package com.example.junior.servlet;

import javax.servlet.*;
import javax.servlet.annotation.WebFilter;
import java.io.IOException;

@WebFilter
public class MyFilter implements Filter {
    @Override
    public void init(FilterConfig filterConfig) throws ServletException {
        Filter.super.init(filterConfig);
    }

    @Override
    public void doFilter(ServletRequest servletRequest, ServletResponse
servletResponse, FilterChain filterChain) throws IOException, ServletException {

    }

    @Override
    public void destroy() {
        Filter.super.destroy();
    }
}

```

注意

使用这些原生组件时，SpringBoot的拦截器不会对其进行拦截。

定制

- 编写自定义的配置类 `xxxConfiguration`; + `@Bean`替换、增加容器中默认组件; 视图解析器 (因为源码有`@Conditional`注解)
- 修改配置文件;
- Web应用 编写一个配置类实现** `WebMvcConfigurer` 即可定制化web功能; + `@Bean`给容器中再扩展一些组件

@EnableWebMvc, 慎用

```
//配置类
@EnableWebMvc
@Configuration //实现WebMvc配置
public class AdmitConfig implements WebMvcConfigurer {
    //定义静态资源
    @Override
    public void addResourceHandlers(ResourceHandlerRegistry registry) {
        //访问aa路径下的所有请求，都去classpath:/static/进行匹配

        registry.addResourceHandler("/aa/**").addResourceLocations("classpath:/static/")
        ;
    }
}
```

自动配置全部失效 (静态资源, 视图解析器, 欢迎页), 要自己处理

数据访问

原先

引入依赖

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

引入properties文件

```
druid.properties文件
url=jdbc:mysql://localhost:3306/library?useUnicode=true&characterEncoding=UTF-8&serverTimezone=Asia/Shanghai
username=root
password=mdjfbzyj515
driverClassName=com.mysql.jdbc.Driver
initialSize=10
maxActive=10
```

spring配置文件中配置连接池

```

<!-- 引入外部属性文件-->
<context:property-placeholder location="classpath:druid.properties">
</context:property-placeholder>
<!-- 利用key-value机制-->
<bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource">
    <property name="driverClassName" value="${driverClassName}"></property>
    <property name="url" value="${url}"></property>
    <property name="username" value="${username}"></property>
    <property name="password" value="${password}"></property>
    <property name="initialSize" value="${initialSize}"></property>
    <property name="maxActive" value="${maxActive}"></property>
</bean>

```

spring配置文件中配置JdbcTemplate对象，注入DataSource

```

<!-- JdbcTemplate对象-->
<bean id="JdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
    <!-- 注入dataSource-->
    <property name="dataSource" ref="dataSource"></property>
</bean>

```

现在

导入JDBC场景

```

<!-- JDBC场景-->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jdbc</artifactId>
</dependency>

```

为什么导入JDBC场景，官方不导入驱动？官方不知道我们接下来要操作什么数据库。

可通过pom.xml

```

<parent>
中<artifactId>spring-boot-starter-parent</artifactId>
的
<parent>
中<artifactId>spring-boot-dependencies</artifactId>
的
得知<mysql.version>8.0.30</mysql.version>
可能不适合我们的数据库

```

修改方式

直接依赖引入具体版本（maven的就近依赖原则）

```

<!-- mysql-->
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.49</version>
</dependency>

```

重新声明版本（maven的属性的就近优先原则）

```
<!--      mysql-->
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
</dependency>
<properties>
  <java.version>1.8</java.version>
  <mysql.version>5.1.49</mysql.version>
</properties>
```

配置文件

```
spring:
  datasource:
    url: jdbc:mysql://localhost:3306/library?
    useUnicode=true&characterEncoding=UTF-8&serverTimezone=Asia/Shanghai
    username: root
    password: mdjfbzyj515
    #驱动
    driver-class-name: com.mysql.jdbc.Driver
    #默认使用HikariDataSource数据库连接池
    #type: com.zaxxer.hikari.HikariDataSource
    #jdbcTemplate对数据库连接池的操作，如果查询超时为3秒
  jdbc:
    template:
      query-timeout: 3
```

什么是JdbcTemplate（Template译为模板）？

Spring 框架对 JDBC 进行封装，使用JdbcTemplate 方便实现对数据库操作

```
@SpringBootTest
class JuniorApplicationTests {

    @Resource
    JdbcTemplate jdbcTemplate;

    @Test
    void contextLoads() throws SQLException {
        System.out.println(jdbcTemplate.getDataSource().getConnection());
    }
}
```

打印出来

HikariProxyConnection@805184575 wrapping com.mysql.jdbc.JDBC4Connection@58ae402b

druid

原先

默认是连接池技术是用HikariDataSource，但我们习惯用druid

引入依赖

```
<!--      druid-->
<dependency>
  <groupId>com.alibaba</groupId>
  <artifactId>druid</artifactId>
  <version>1.2.11</version>
</dependency>
```

配置类

```
@Configuration
public class DataBaseConfig {
    //默认自动配置判断容器没有才会配@ConditionalOnMissingBean (DataSource.class)
    @Bean
    //配置绑定
    @ConfigurationProperties("spring.datasource")
    /*对应application.yml
    spring:
      datasource:
    */
    public DataSource dataSource()
    {
        return new DruidDataSource();
    }
}
```

现在

引入官方的druid-spring依赖

```
<!--      druid-->
<dependency>
  <groupId>com.alibaba</groupId>
  <artifactId>druid-spring-boot-starter</artifactId>
  <version>1.2.11</version>
```

则可自动配置

还有druid的监控功能

```
druid:
  filter: stat,wall
  #开启网页监控视图
  stat-view-servlet:
    #开启监控功能,默认是false
    enabled: true
    #admit
    login-username: admit
    #password
    login-password: 123456
```

```
#开启路径监控
web-stat-filter:
  #开启
  enabled: true
  #所有路径
  url-pattern: /*
  #不包括的路径
  exclusions: '*.js,*.gif,*.jpg,*.png,*.css,*.ico,/druid/*'
```

```
spring:
  datasource:
    url: jdbc:mysql://localhost:3306/library?
    useUnicode=true&characterEncoding=UTF-8&serverTimezone=Asia/Shanghai
    username: root
    password: mdjfbzyj515
    driver-class-name: com.mysql.jdbc.Driver
  druid:
    web-stat-filter:
      exclusions: '*.js,*.gif,*.jpg,*.png,*.css,*.ico,/druid/*'
      url-pattern: /*
      enabled: true
    stat-view-servlet:
      login-username: admit
      login-password: 123456
      enabled: true
```

用

<http://localhost:8080/druid>

或者

<http://localhost:8080/druid/>

开启

mybatis

以前

pom.xml

```
<!-- 引入mybatis-->
<dependency>
  <groupId>org.mybatis</groupId>
  <artifactId>mybatis</artifactId>
  <version>3.5.9</version>
</dependency>
<!-- 引入mybatis和spring联系-->
<dependency>
  <groupId>org.mybatis</groupId>
  <artifactId>mybatis-spring</artifactId>
  <version>1.3.0</version>
</dependency>
```


mybatis.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration
    PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
    <!-- 驼峰命名-->
    <settings>
        <setting name="mapUnderscoreToCamelCase" value="true"/>
    </settings>
    <!-- 设置类的别名为类名-->
    <typeAliases>
        <package name="zyj.pojo"/>
    </typeAliases>
    <!-- 配置连接数据库的环境-->
    <!-- environments:配置多个连接数据库的环境
        default: 设置默认使用环境-->
</configuration>
```

applicationContext.xml

```
<!-- 引入外部属性获取数据库连接信息-->
    <context:property-placeholder location="classpath:jdbc.properties">
</context:property-placeholder>
<!-- 连接数据库-->
    <bean id="dataSource"
class="org.apache.ibatis.datasource.pooled.PooledDataSource">
        <!-- 设置连接数据库的驱动-->
        <property name="driver" value="${jdbc.driver}"/>
        <!-- 设置连接数据库的连接地址-->
        <property name="url" value="${jdbc.url}"/>
        <!-- 设置连接数据库的用户名-->
        <property name="username" value="${jdbc.username}"/>
        <!-- 设置连接数据库的密码-->
        <property name="password" value="${jdbc.password}"/>
    </bean>
<!-- 配置sqlSessionFactoryBean, 可以在ioc获取sqlSessionFactory, spring和mybatis的
整合-->
    <bean class="org.mybatis.spring.SqlSessionFactoryBean">
<!-- 获取数据库-->
        <property name="dataSource" ref="dataSource"></property>
<!-- 加载mybatis的全局配置文件-->
        <property name="configLocation" value="classpath:mybatis.xml">
</property>
        <!-- 引入映射文件-->
        <property name="mapperLocations" value="classpath*:mappers/*Mapper.xml">
</property>
    </bean>
<!-- mapper扫描, 扫描mapper接口, 会实现mapper接口的代理实现类对象, 也就是直接new了
mappers包的对象注入IOC-->
    <bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
        <property name="basePackage" value="zyj.mappers"></property>
    </bean>
```

现在

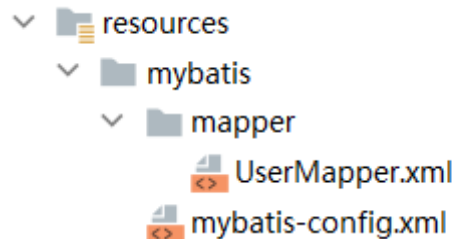
```
<!--      mybatis-->
<dependency>
  <groupId>org.mybatis.spring.boot</groupId>
  <artifactId>mybatis-spring-boot-starter</artifactId>
  <version>2.1.3</version>
</dependency>
```

配置模式

全局配置文件

SqlSessionFactory自动配置好了

SqlSession自动配置了, SqlSessionTemplate组合了SqlSession



```
mybatis:
#mybatis全局配置文件
config-location: classpath:mybatis/mybatis-config.xml
#mybatis映射文件
mapper-locations: classpath:mybatis/mapper/*.xml
#开启驼峰命名
configuration:
  map-underscore-to-camel-case: true
```

关于mybatis配置文件的值也可以在application.yml配置!!!! 但不建议!!!!

但注意这个时候就不能用

```
#mybatis全局配置文件
config-location: classpath:mybatis/mybatis-config.xml
```

会报错

java.lang.IllegalStateException: Property 'configuration' and 'configLocation' can not specified with together

注意接口要有注解@Mapper

```
package com.example.junior.mybatis;

import com.example.junior.pojo.User;
import org.apache.ibatis.annotations.Mapper;
import org.apache.ibatis.annotations.Param;
import java.time.LocalDateTime;

@Mapper
public interface UserMapper {
  //通过电话号码登录
  public User loginByPhone(@Param("phone") String phone, @Param("password")
String password);
  //注册账号
```

```

    public void register(@Param("phone") String phone, @Param("password") String
password, @Param("createTime") LocalDateTime createTime);
    //通过号码查询账号
    public User findByPhone(@Param("phone")String phone);
    //通过id查询账号
    public User findById(@Param("id")String id);
    //修改数据
    public void update(User user);
}

```

否则服务层注入不了接口对象

```

package com.example.junior.Service;

import com.example.junior.mybatis.UserMapper;
import com.example.junior.pojo.User;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class UserService {
    @Autowired
    UserMapper userMapper;
    //通过id查询获取网名和电话
    public User getPersonal(String id)
    {
        return userMapper.findById(id);
    }
}

```

也可以用@Select写sql语句，不推荐不灵活

```

@Mapper
public interface UserMapper {
    @Select("se")
    public User testSelect();
    @Insert("DASDAS")
    public void testInsert();
}

```

每一个接口都要用@Mapper扫描很麻烦，也可以

```

@MapperScan(basePackages = "com.example.junior.mapper")
@SpringBootApplication
public class JuniorApplication {
    public static void main(String[] args) {
        ConfigurableApplicationContext run =
SpringApplication.run(JuniorApplication.class, args);
        run.containsBean("SqlSessionFactoryBean");
    }
}

```

service层出现 Could not autowire. No beans of 'UserMapper' type found. 也没事

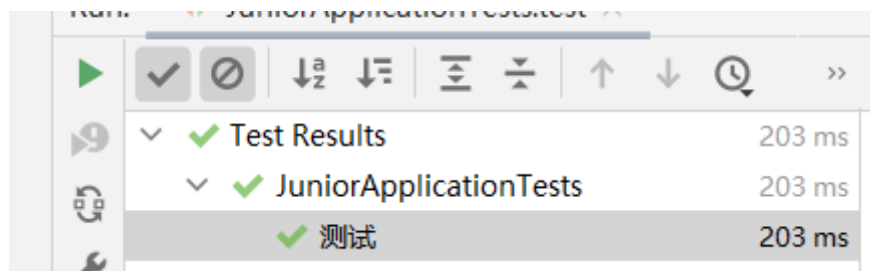
```
@Service
public class UserService {
    @Autowired
    UserMapper userMapper;
```

测试

@Test用来测试

```
@Transactional
@Test
可以回滚
```

@DisplayName("测试")会显示



@BeforeEach

每一个测试方法前都运行

@AfterEach

每一个测试方法后都运行

方法必须是static的

@BeforeAll

所有测试方法前执行

@AfterAll

所有测试方法后运行

@Disabled

执行类中所有方法可以禁用掉该方法

@Timeout(value = 500, unit = TimeUnit.MILLISECONDS)

超过500毫秒就不执行，抛出超时异常

@SpringBootTest

加了该注解在类可用容器的Bean

@RepeatedTest(5)

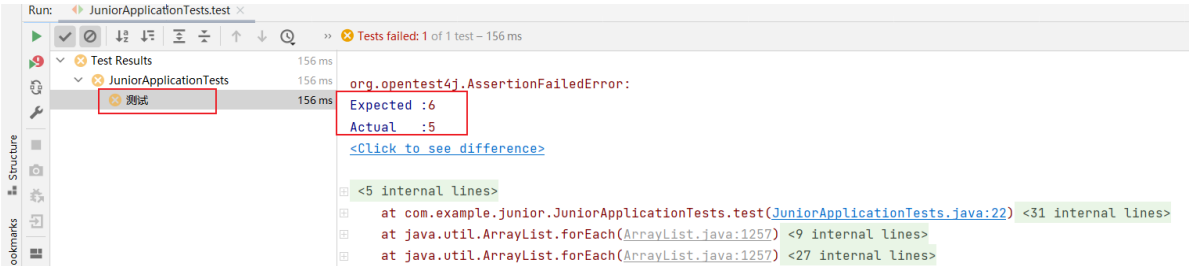
重复测试5次

断言

可以是`assertEquals(6,5);`
也可以是`Assertions.assertEquals(6,5);`

```
@DisplayName("测试")
@Test
public void test()
{
    Assertions.assertEquals(6,5);
}
```

不符合则会提示



断言	功能
<code>assertEquals([String message], expected value, actual value)</code>	检查int, short, long, byte, char或Object等类型的 值是否相等 ，第一个参数是一个可选的字符串消息。
<code>assertEquals(double expected value, double actual value, double delta)</code>	检查指定误差范围内的 double类型值是否相等 ，需要第三个参数delta表示可接受的误差范围。
<code>assertNotEquals([String message], expected value, actual value)</code>	检查 值是否不相等 ，第一个参数是一个可选的字符串消息。
<code>assertArrayEquals([String message], expectedArray, resultArray)</code>	检查 两数组内容是否相同
<code>assertTrue(boolean condition)</code>	检查 条件是否为真
<code>assertFalse(boolean condition)</code>	检查 条件是否为假
<code>assertNull(Object object)</code>	检查 对象是否为空
<code>assertNotNull(Object object)</code>	检查 对象是否不空
<code>assertSame(Object expected, Object actual)</code>	检查两个变量是否 引用同一对象
<code>assertNotSame(Object expected, Object actual)</code>	检查两个变量是否 不引用同一对象

还有组合断言，`assertAll`，所有断言成功才成功，需要再了解。

异常断言

一定要抛出异常，不异常就报错

```
@DisplayName("测试")
@Test
public void test()
{
    assertThrows(ArithmeticException.class, ()->{
        int i=10/4;
    }, "数学运算居然成功!!");
}
```

报错为

org.opentest4j.AssertionFailedError: 数学运算居然成功!! ==> Expected java.lang.ArithmeticException to be thrown, but nothing was thrown.

快速失败

```
fail("失败");
```

一定是org.opentest4j.AssertionFailedError: 失败

前置条件

前置条件类似断言，不同之处在于使用断言会看到错误提示，但是前置条件不会，具体可以看下图的示例

```
@DisplayName("测试")
@Test
public void test()
{
    System.out.println(000000);
    Assumptions.assumeTrue(false, "结果不是正确的");
    System.out.println(1111);
}
```

嵌套测试

```
@Nested
class a{
    @BeforeAll
    public void before(){};
}
```

参数化测试

```
@ParameterizedTest
@DisplayName("参数化测试")
@ValueSource(ints = {1,2,3,4,5})
void test(int i)
{
    System.out.println("测试");
}
```

✓ Test Results	188 ms	测试
✓ JuniorApplicationTests	188 ms	
✓ 参数化测试	188 ms	
✓ [1] i=1	178 ms	
✓ [2] i=2	4 ms	
✓ [3] i=3	2 ms	
✓ [4] i=4	2 ms	
✓ [5] i=5	2 ms	

```
@ParameterizedTest
@DisplayName("参数化测试")
@MethodSource("aadsadas")    //调用方法aadsadas的返回值
void test(int i)
{
    System.out.println("测试");
}
```

监控功能

就是来监控东西的

依赖

```
<!--      监控功能-->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

这是springboot程序的监控系统，可以实现健康检查，info信息等。在使用之前需要引入 `spring-boot-starter-actuator`，并做简单的配置即可。

配置

```
#management是所有actuator的配置
management:
  endpoints:
    enabled-by-default: true    #暴露所有端点信息
  web:
    exposure:
      include: '*'    #以web方式暴露
    endpoint: #端点名.xxx    具体的端点
  health:
    show-details: always
    enabled: true    #某个端点开不开
```

通过

localhost:8080/actuator

或者

localhost:8080/actuator/

访问

端点

localhost:8080/actuator/ID名

最常用

Health: 监控状况

Metrics: 运行时指标

Loggers: 日志记录

ID	描述
<code>auditevents</code>	暴露当前应用程序的审核事件信息。需要一个 <code>AuditEventRepository</code> 组件。
<code>beans</code>	显示应用程序中所有Spring Bean的完整列表。
<code>caches</code>	暴露可用的缓存。
<code>conditions</code>	显示自动配置的所有条件信息，包括匹配或不匹配的原因。
<code>configprops</code>	显示所有 <code>@ConfigurationProperties</code> 。
<code>env</code>	暴露Spring的属性 <code>ConfigurableEnvironment</code>
<code>flyway</code>	显示已应用的所有Flyway数据库迁移。需要一个或多个 <code>Flyway</code> 组件。
<code>health</code>	显示应用程序运行状况信息。
<code>httptrace</code>	显示HTTP跟踪信息（默认情况下，最近100个HTTP请求-响应）。需要一个 <code>HttpTraceRepository</code> 组件。
<code>info</code>	显示应用程序信息。
<code>integrationgraph</code>	显示Spring <code>integrationgraph</code> 。需要依赖 <code>spring-integration-core</code> 。
<code>loggers</code>	显示和修改应用程序中日志的配置。
<code>liquibase</code>	显示已应用的所有Liquibase数据库迁移。需要一个或多个 <code>Liquibase</code> 组件。
<code>metrics</code>	显示当前应用程序的“指标”信息。
<code>mappings</code>	显示所有 <code>@RequestMapping</code> 路径列表。
<code>scheduledtasks</code>	显示应用程序中的计划任务。
<code>sessions</code>	允许从Spring Session支持的会话存储中检索和删除用户会话。需要使用Spring Session的基于Servlet的Web应用程序。
<code>shutdown</code>	使应用程序正常关闭。默认禁用。
<code>startup</code>	显示由 <code>ApplicationStartup</code> 收集的启动步骤数据。需要使用 <code>SpringApplication</code> 进行配置 <code>BufferingApplicationStartup</code> 。
<code>threaddump</code>	执行线程转储。

自定义端点/需要再了解

Profile功能

为了方便多环境适配，springboot简化了profile功能。

- 默认配置文件 `application.yaml`；任何时候都会加载
- 指定环境配置文件 `application-{env}.yaml`
- 激活指定环境

例如有application-test.yml

在application.properties中配置spring.profiles.active=test则会激活指定环境

```
spring.profiles.active=test
也可以有多个配置文件一起用，不过挺鸡肋
spring.profiles.active=test
spring.profiles.group.test[0]=test0
spring.profiles.group.test[1]=test1
```

优先级

application.properties>指定环境配置文件>application.yml

@Profile

指定组件在哪个环境的情况下才能被注册到容器中，不指定，任何环境下都能注册这个组件

1. 加了环境标识的bean，只有这个环境被激活的时候才能注册到容器中。默认是default环境
2. 写在配置类上，只有是指定的环境的时候，整个配置类里面的所有配置才能开始生效

例如有application-test.yml

则

```
@Profile("test")
public class testInterceptor implements HandlerInterceptor {
}
```

命令行激活

- 命令行激活：java -jar xxx.jar --spring.profiles.active=prod --person.name=haha
- 修改配置文件的任意值，命令行优先

配置文件查找位置

- (1) classpath 根路径
- (2) classpath 根路径下config目录
- (3) jar包当前目录
- (4) jar包当前目录的config目录
- (5) /config子目录的直接子目录

越往下优先级越高，因为覆盖，也就是越往外越优先

stater的理解/干货满满

https://www.bilibili.com/video/BV19K4y1L7MT?p=83&vd_source=003e9f54d7ed05b77aee32ac0f194b0a