

Springboot 运行起来

2017年7月31日 15:22

SpringBoot是SpringMVC的进阶改进版，其实两者没什么必然联系，现在SpringBoot是流行趋势，[可以直接学习SpringBoot而跳过SpringMVC](#).

前置知识

- 1. 利用maven构建项目
- 2. Spring注解
- 3. RESTful API

《项目管理利器maven》
<http://www.imooc.com/learn/443>

《Spring入门篇》
<http://www.imooc.com/learn/196>

注意！

- 1. 具备必要的前置知识
- 2. 不需要去学SpringMVC
- 3. Java、Maven等版本保持一致

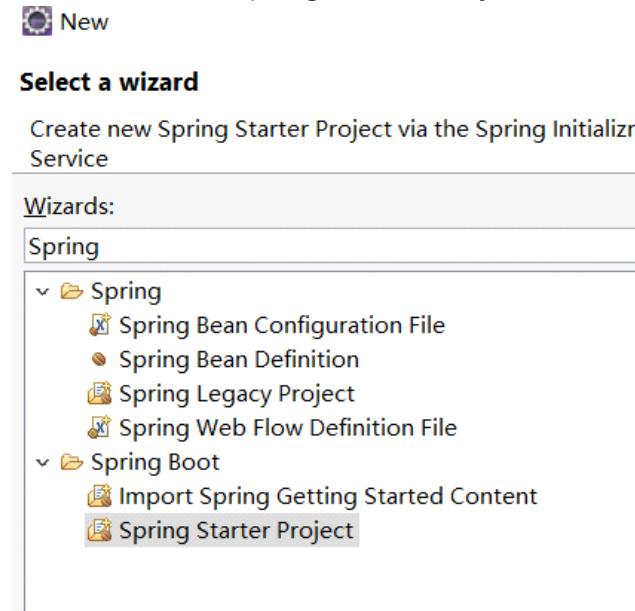
SpringBoot的特点

- 1. 化繁为简，简化配置
- 2. 备受关注，是下一代框架
- 3. 微服务的入门级微框架

[Eclipse中新建SpringBoot项目](#)

我用的最新版eclipse Oxygen已经自带Spring Boot了，

1、新建项目选择Spring Starter Project



2、给项目命名，Group改成自己想要的包名，Type默认是Maven，也就是会自动变成

Maven项目（建好后等一分钟就变成Maven格式了）

New Spring Starter Project

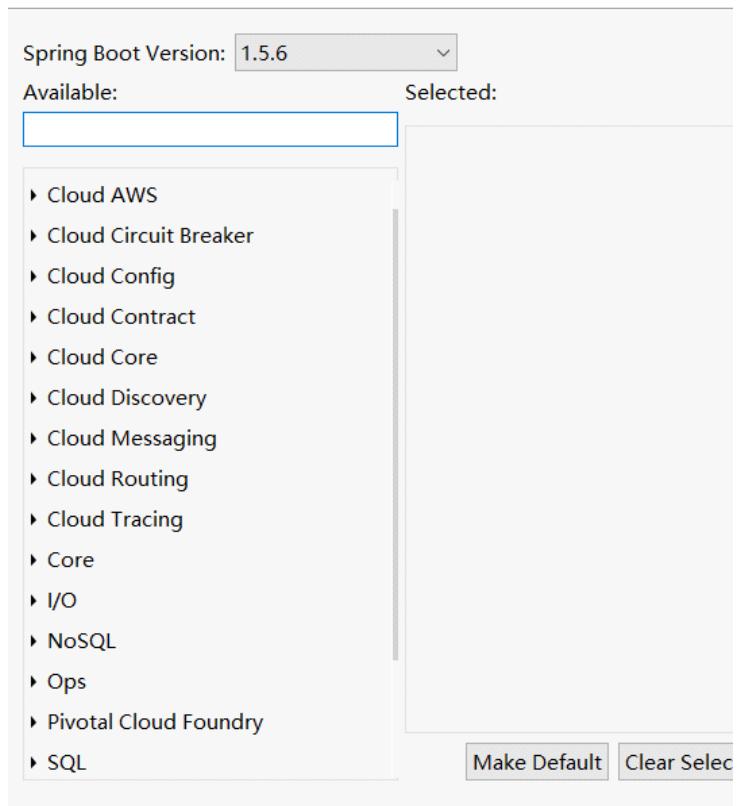
This screenshot shows the 'New Spring Starter Project' configuration dialog. It has several fields:

- Service URL: http://start.spring.io
- Name: mySpringBootTest
- Use default location
- Location: C:\Users\18894\Desktop\Design\JAVA\mySpringBootTest
- Type: Maven
- Packaging: (dropdown menu)
- Java Version: 1.8
- Language: (dropdown menu)
- Group: com.jinsong
- Artifact: mySpringBootTest
- Version: 0.0.1-SNAPSHOT
- Description: Demo project for Spring Boot
- Package: com.example.demo
- Working sets:
 - Add project to working sets
 - Working sets: (dropdown menu)

3、选择SpringBoot版本和勾选各种想要的组件

作为测试，我们只勾选了web，还能勾选mysql、mybatis

New Spring Starter Project Dependencies



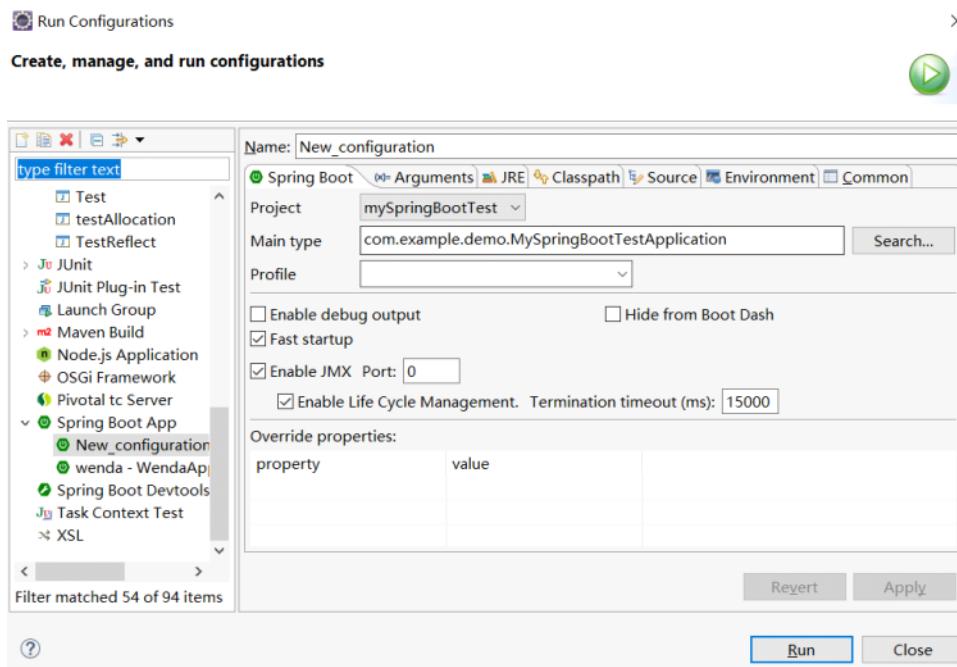
4、建好后测试项目是否能运行，右键main方法

```
package com.example.demo;
import org.springframework.boot.SpringApplication;
@SpringBootApplication
public class MySpringBootTestApplication {
    public static void main(String[] args) {
        SpringApplication.run(MySpringBootTestApplication.class, args);
    }
}
```

5、第一次运行要设置Run Configuration

选择Name、Project和Main type

注意不要额外开启Tomcat



```

on      : Starting MySpringBootTestApplication on JsPC with PID 1
on      : No active profile set, falling back to default profiles
context : Refreshing org.springframework.boot.context.embedded.AnnotationConfigEmbeddedWebApplicationContext@633333: startup date [2017-07-31 17:20:09 CST]; root of context hierarchy
container : Tomcat initialized with port(s): 8080 (http)
vice    : Starting service [Tomcat]
ngine   : Starting Servlet Engine: Apache Tomcat/8.5.16
          : Initializing Spring embedded WebApplicationContext
          : Root WebApplicationContext: initialization completed in 1 ms
nBean  : Mapping servlet: 'dispatcherServlet' to [/]
Bean    : Mapping filter: 'characterEncodingFilter' to: [/*]
Bean    : Mapping filter: 'hiddenHttpMethodFilter' to: [/*]
Bean    : Mapping filter: 'httpPutFormContentFilter' to: [/*]
Bean    : Mapping filter: 'requestContextFilter' to: [/*]
dapter : Looking for @ControllerAdvice: org.springframework.boot.autoconfigure.web.servlet.error.ErrorMvcAutoConfiguration$DefaultErrorController
apping  : Mapped "[/*/error]" onto public org.springframework.http.ResponseEntity<org.springframework.util.MultiValueMap<java.lang.String, java.lang.Object>> org.springframework.boot.autoconfigure.web.servlet.error.ErrorMvcAutoConfiguration$DefaultErrorController.handleError(javax.servlet.http.HttpServletRequest)
apping  : Mapped "[/*/error],produces=[text/html]" onto public org.springframework.http.ResponseEntity<java.lang.String> org.springframework.boot.autoconfigure.web.servlet.error.ErrorMvcAutoConfiguration$DefaultErrorController.error(javax.servlet.http.HttpServletRequest)
apping  : Mapped URL path [/webjars/**] onto handler of type [class org.springframework.web.servlet.handler.BeanNameUrlHandler]
apping  : Mapped URL path [/**] onto handler of type [class org.springframework.boot.autoconfigure.web.servlet.error.ErrorMvcAutoConfiguration$DefaultErrorController]
apping  : Mapped URL path [/**/favicon.ico] onto handler of type [class org.springframework.boot.autoconfigure.web.servlet.error.ErrorMvcAutoConfiguration$DefaultErrorController]
tainer : Tomcat started on port(s): 8080 (http)
on      : Started MySpringBootTestApplication in 1.726 seconds (JVM running for 2.002)

```

Started

127.0.0.1:8080

Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback

Mon Jul 31 17:20:09 CST 2017

There was an unexpected error (type=Not Found, status=404).

No message available

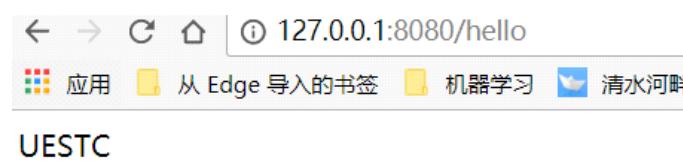
第二种启动方式：cmd中启动springboot项目

首先进入项目目录，然后mvn spring-boot:run

如果eclipse中也启动了该项目，端口被占用则会启动失败。

```
C:\Users\18894\Desktop\Design\JAVA\mySpringBootTest>mvn spring-boot:run
```

```
Tomcat started on port(s): 8080 (http)
Started MySpringBootApplication in 3.621 seconds (JVM running for 7.323)
Initializing Spring FrameworkServlet 'dispatcherServlet'
FrameworkServlet 'dispatcherServlet': initialization started
FrameworkServlet 'dispatcherServlet': initialization completed in 24 ms
```



启动成功！

第三种启动方式：cmd中编译项目成jar包再启动项目

首先进入项目目录，然后mvn install对项目进行编译

```
C:\Users\18894\Desktop\Design\JAVA\mySpringBootTest>mvn install
```

显示编译成功

```
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 01:49 min
[INFO] Finished at: 2017-08-01T10:52:42+08:00
[INFO] Final Memory: 26M/304M
[INFO] -----
```

编译成功后target目录下会有编译好的jar包

名称	修改日期
classes	2017/8/1 9:50
generated-sources	2017/8/1 10:45
generated-test-sources	2017/8/1 10:45
maven-archiver	2017/8/1 10:52
maven-status	2017/8/1 10:45
surefire-reports	2017/8/1 10:52
test-classes	2017/8/1 9:13
mySpringBootTest-0.0.1-SNAPSHOT.jar	2017/8/1 10:52

进入target目录，用java -jar 命令运行这个jar包来启动项目

```
i\JAVA\mySpringBootTest\target>java -jar mySpringBootTest-0.0.1-SNAPSHOT.jar
```

启动成功！！！

```
Tomcat started on port(s): 8080 (http)
Started MySpringBootApplication in 4.089 seconds (JVM running for 4.67)
```

SpringBoot配置文件application.properties

```
application.properties > mySpringBootTest\p... MySpringBo
1server.port=8081
2server.context-path=/mySpringBootTest
```

```
application.properties
1server.port=8081
2server.context-path=/mySpringBootTest
```

Port	更改端口号
Context-path	在访问路径前加上/mySpringBootTest, 比如原来访问路径是 http://127.0.0.1:8081/hello, 现在是 http://127.0.0.1:8081/mySpringBootTest/hello

```
application.yml
server:
  port: 8082
  context-path: /girly
```

更推荐使用.yml文件的这种格式进行配置，注意port:后面有一个空格，然后就可以把上面的application.properties删掉了。

使用application.yml中的配置变量值

方法一：

```
application.yml
Server:
  port: 8080
  cupSize: B
  age: 18
  content: "cupSize:${cupSize},age:${age}"
```

```
@RestController
public class HelloController {

    //cupSize是在配置文件application.yml中设置的变量, cupSize=8
    //这里使用Value注解, 调用配置文件里的变量值
    @Value("${cupSize}")
    private String cupSize;

    @Value("${age}")
    private int age;

    @Value("${content}")
    private String content;

    @RequestMapping(value="/hello",method=RequestMethod.GET)
    public String say() {
        return age+cupSize+"...."+content;
    }
}
```



18B.....cupSize:B,age:18

方法二：

application.yml

```
1 Server:
2   port: 8080
3 cupSize: B
4 age: 18
5 content: "cupSize:${cupSize},age:$
6 boy:
7   school: UESTC
8   city: ChengDu
9
```

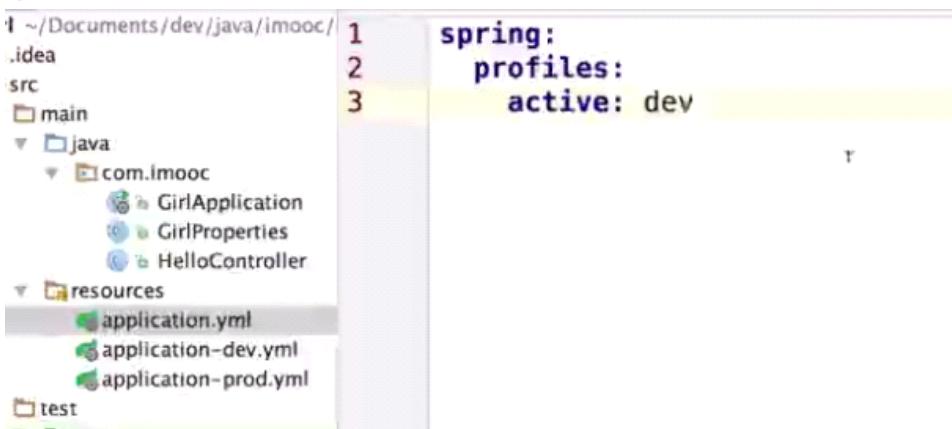
BoyProperties.java

```
1 package com.example.demo;
2
3 import org.springframework.boot.context.properties.ConfigurationProperties;
4 import org.springframework.stereotype.Component;
5
6 /**
7 * 使用application.yml中配置的属性值
8 *
9 * 记得构造get/set方法以方便其他类调用
10 * @author 188949420@qq.com
11 *
12 */
13 @ConfigurationProperties(prefix="boy") //此注解获得application.yml中boy
14 @Component //声明为组件，加此注解才能在其他类通过注解调用
15 public class BoyProperties {
16
17     private String school;
18     private String city;
19
20     public String getSchool() {
21         return school;
22     }
23     public void setSchool(String school) {
24         this.school = school;
25     }
26     public String getCity() {
27         return city;
28     }
29     public void setCity(String city) {
30         this.city = city;
31     }
32 }
```

```
@Autowired  
private BoyProperties boyProperties;  
  
 @RequestMapping(value = "/hello", method = RequestMethod.GET)  
 public String say() {  
     return boyProperties.getSchool();  
 }
```



不同配置的切换



在默认配置文件中application.yml中照上面写，表示采用application-dev.yml配置。
如果启动两个同样的项目但是采用不同的配置，就能这样写。

Controller的使用

@Controller	处理http请求
@RestController	Spring4之后新加的注解，原来返回json 需要@RequestBody配合@Controller
@RequestMapping	配置url映射

Controller的使用

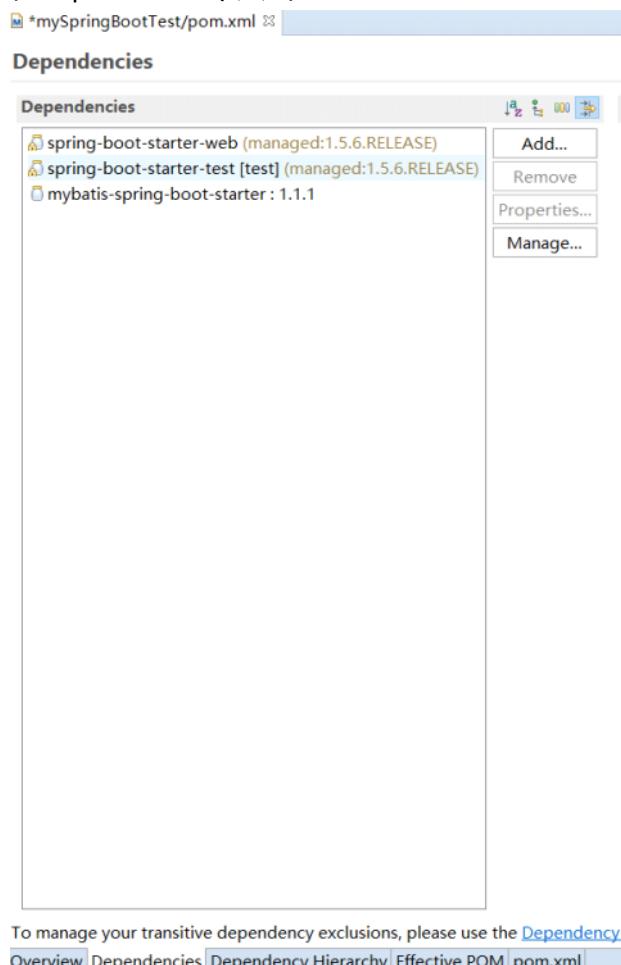
@PathVariable	获取url中的数据
@RequestParam	获取请求参数的值
@GetMapping	组合注解

@GetMapping就是@RequestMapping+method=RequestMethod.GET

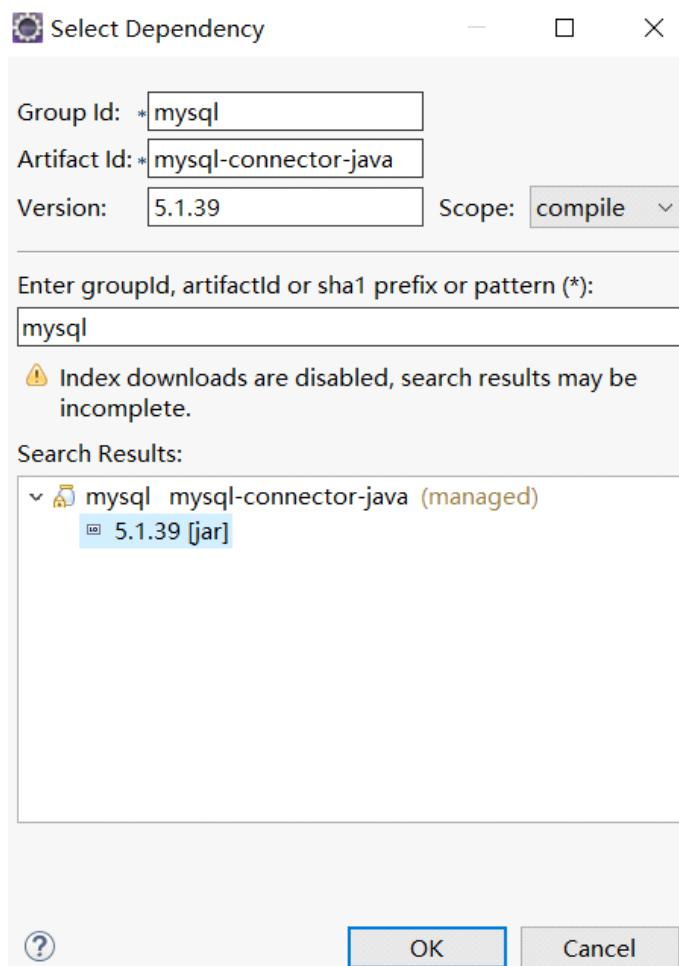
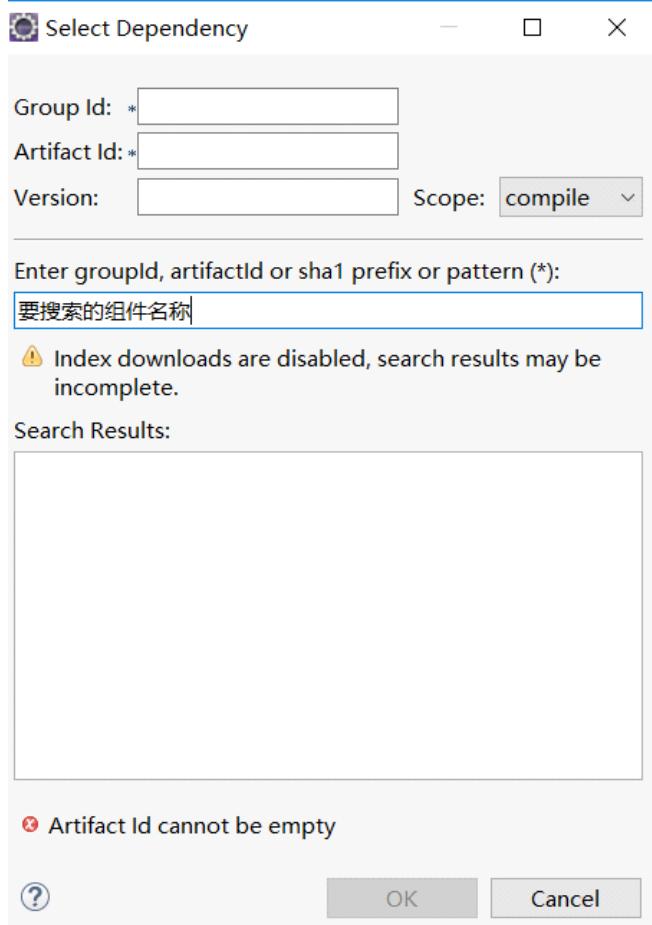
@PostMapping同理

Pom.xml添加组件

不一定非要手写，因为常常不知道GroupId\Version信息怎么写，
在Dependencies中点击Add



在这里输入要搜索的组件名称



Spring Boot 揭秘与实战

来自 <<https://github.com/lianggzone/springboot-action>>

Spring Boot配置

Spring Boot会自动扫描@`SpringBootApplication` (`springboot`项目启动类的注解) 所在类的同级包以及下级包里的所有bean (所有注解组件@)，所以这个入口类要放在最外层包名下。所以Spring Boot 不需要XML配置包扫描，直接使用Annotation注解就行啦
Spring Boot的默认属性配置文件是在resources下的`application.properties`，也可以使用`application.yaml`来配置默认参数，配置即使用，无需再单独声明

```
@SpringBootApplication
public class WendaApplication {

    public static void main(String[] args) {
        SpringApplication.run(WendaApplication.class, args)
    }
}
```

Spring Boot事务

Spring Boot 默认集成事务，所以无须手动开启使用

`@EnableTransactionManagement` 注解

在需要使用事务的方法上添加注解 `@Transactional`

Spring 拦截器会在这个方法调用时，开启一个新的事务，当方法运行结束且无异常的情况下，提交这个事务。

我们如果使用到 `spring-boot-starter-jdbc` 或 `spring-boot-starter-data-jpa`，Spring Boot 会自动默认分别注入

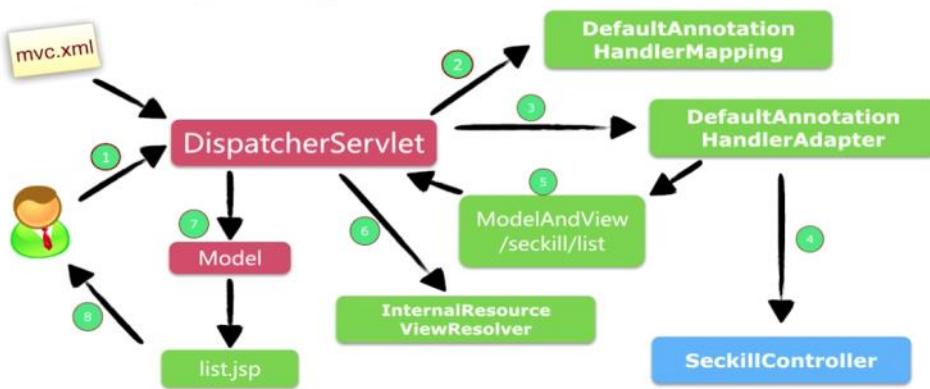
`DataSourceTransactionManager` 或 `JpaTransactionManager`这两个事务管理器

`@Transactional (value= “指定事务管理器”)`

Mysql只有发生`RuntimeException`时事务才会回滚

来自 <http://blog.720ui.com/2017/springboot_02_data_transactional/>

SpringMVC运行流程



Springboot 项目架构

2018年2月5日 16:38

梳理一下springboot项目架构

- 1、建立新的springboot项目等同于Maven项目，详细步骤看[Spring Boot运行起来](#)
- 2、pom依赖导入好之后，application.properties设置数据库信息和mybatis信息并在mybatis-config.xml中配置mybatis
- 3、分层写代码，先在model层中把模型写好（采用MBG [Mybatis Generator](#)），然后从下往上写，依次是mapper文件夹下的mybatis的SQL（查看[Mybatis](#)）、dao层、service层（service接口和服务实现层）、controller层（[fastJSON工具](#)）。
- 4、注意controller层写请求转发、参数验证和简单的处理代码，具体、复杂的服务处理写到service层去。至少形成controller-service-dao一对一关系，实际中常常是一对多关系，即一个service中调用多个dao层，毕竟dao层相当于纯粹的数据库数据获取，根据业务可以进行组合。

5、后台接受POST数据存到数据库里

MoemilAfterSales项目中有演示

采用@ModelAttribute注解，可以把表单里的参数直接传到后面Model里的对应属性值里，这里是直接传到Repair里，不需要其他任何操作。但注意请求头必须用"Content-Type": "application/x-www-form-urlencoded"

```
* @ModelAttribute 的请求头必须用"Content-Type": "application/x-www-form-urlencoded"
*/
@PostMapping("/repair")
public String insertRepair(@ModelAttribute Repair repair) {
```

6、前端展示，正常开发用 (3) @Controller，(5)@RestController这种，(2)也常用

(1) 直接返回String

按理说返回的不是HTML标签页面应该加@ResponseBody注解

```
@GetMapping("/test")
public String test() {
    return "Test ! Moemil AfterSales";
}
```

(2) 添加@ResponseBody返回对象，好用！！！！！！

这里我们new了一个对象，然后直接返回这个对象，添加了@ResponseBody注解后，会把这个对象转换成JSON格式String，前端看到的就是JSON格式的String

```
@GetMapping("/test2")
@ResponseBody
public Engineer test2() {
```

```

@GetMapping("/test2")
@ResponseBody
public Engineer test2() {
    Engineer engineer = new Engineer();
    engineer.setEngineer("Jinsong");
    engineer.setTel("18571686931");

    return engineer;
}

```



集合对象一样能正确转换成JSON格式的String

```

@GetMapping("/test3")
@ResponseBody
public List<Engineer> test3() {
    List<Engineer> engineers = testService.listEngineer();

    return engineers;
}

```



(3) 传入model返回String 好用！！！

参数中传入Model则Spring会自动传一个model进来，model添加的key都能在前端拿到
这里要显示list.html页面的话，必须用@Controller注解，不能用@RestController注解，因为@RestController注解中含有@ResponseBody注解，结果index没有经过视图解析器解析直接变成json字符串返回给浏览器了

采用@RestController注解返回页面的话，采用(5)方法

如果用Thymeleaf的话，要引入依赖，才能正常返回页面，否则就是返回字符串，如果用velocity，在application.properties中要添加spring.velocity.suffix=.html

```

@RequestMapping(value = "/list", method = RequestMethod.GET)
public String list(Model model) {
    List<Seckill> seckillList = seckillService.getSeckillList();

    // model里的内容作为JSON串会自动的给到前端页面里去
    model.addAttribute("list", seckillList);

    return "list";
}

```

(4) 返回JSON格式的String

这里要善用阿里的JSON工具 [fastJSON工具](#)

```
@GetMapping("/engineers")
@ResponseBody
public String listEngineer(Model model) {
    List<Engineer> engineers =testService.listEngineer();

    //阿里巴巴的fastjson工具，直接把对象、list\map等转换成JSON格式，非常方便！！！
    //比如json.toJSONString(engineers)
    JSONObject json =new JSONObject();

    model.addAttribute("result",engineers);
    return json.toJSONString(model);
}
```



(5)返回 ModelAndView

用@RestController注解的话，必须用 ModelAndView 才能返回页面

new ModelAndView的时候传入的参数就是前端html页面的名字，前端接参数就用 JSP、Thymeleaf（详见 [Spring Boot静态资源目录](#)）、velocity等各种渲染语言了。

ModelAndView.addObject()的key 在前端页面中都可以拿到

```
@GetMapping("pagetest")
public ModelAndView page() {
    ModelAndView result = new ModelAndView("page");

    PageHelper.startPage(0, 3); //PageHelper.startPage紧接着的一句代码会被分页
    List<Seckill> list =seckillService.selectAll();

    PageInfo<Seckill> pageInfo =new PageInfo<Seckill>(list);
    result.addObject("pageInfo", pageInfo);

    return result;
}
```

Springboot 静态资源目录

2017年8月4日 10:40

SpringBoot有默认的静态资源存放目录，不需要任何配置

来自 <http://www.cnblogs.com/chry/p/5877979.html>

在Maven 工程目录下，所有静态资源都放在src/main/resource目录下，结构如下

src/main/resource

```
|____ static
|   |____ js
|   |____ images
|   |____ css
|____ templates(页面模板)
....
```

例如：

```
✓ M S > wenda [boot] [devtools] [wenda master]
  > 📂 > src/main/java
  ✓ 📂 src/main/resources
    > 📂 com
    ✓ 📂 static
      > 📂 images
      > 📂 scripts
      > 📂 styles
    ✓ 📂 templates
      > 📂 mails
        📄 detail.html
        📄 feeds.html
        📄 followees.html
        📄 followers.html
        📄 footer.html
        📄 header.html
```

```
application.properties index.html ✘
1 #parse("header.html")
2 <link rel="stylesheet" href="../styles/index.css">
3 <link rel="stylesheet" href="../styles/detail.css">
```

```
application.properties ✘ index.html js.html ✘
1 
2 <script type="text/javascript" src="/scripts/main/jquery.js"></script>
3 <script type="text/javascript" src="/scripts/main/base/base.js"></script>
4 <script type="text/javascript" src="/scripts/main/base/util.js"></script>
5 <script type="text/javascript" src="/scripts/main/base/event.js"></script>
```

控制器中

```
return "home"; // 指定返回的模板也就是html文件
```

```
*application.properties ✘ index.html js.html
1spring.velocity.suffix=.html
```

返回的home就是
src/main/resource/templates/home.html
不需要其他任何配置了

Thymeleaf与Spring Boot

<http://blog.csdn.net/u012558400/article/details/53321558>

<http://blog.csdn.net/u014695188/article/details/52347318>

```
<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
    <java.version>1.8</java.version>
    <spring-cloud.version>Dalston.SR2</spring-cloud.version>
    <!-- springboot默认集成thymeleaf 2.0，添加以下两行升级为3.0 -->
    <thymeleaf.version>3.0.2.RELEASE</thymeleaf.version>
    <thymeleaf-layout-dialect.version>2.0.4</thymeleaf-layout-dialect.version>
</properties>

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

Thymeleaf基本语法

来自 <http://blog.csdn.net/u012706811/article/details/52185345>

```
<!-- thymeleaf标签全部写在标签里面，标签外面也能写静态的数据，真正实现一个页面里，前后端数据分离！！！ -->
<tr th:each="sk:${list}">   <!-- list是后端传过来的Model里的变量名 -->
    <td th:text="${sk.name}">名称</td>
    <td th:text="${sk.number}">库存</td>
    <td th:text="${#dates.format(sk.startTime,'yyyy-MM-dd HH:mm:ss')}">秒杀开启时间</td>
    <td th:text="${#dates.format(sk.endTime,'yyyy-MM-dd HH:mm:ss')}">秒杀结束时间</td>
    <td th:text="${#dates.format(sk.createTime,'yyyy-MM-dd HH:mm:ss')}">秒杀创建时间</td>
    <td><a class="btn btn-info"
        th:href="@{'/'+'${sk.seckillId}'+ '/detail'}" target="_blank">详情页</a></td>
    </td>
</tr>
</c.forEach>
```

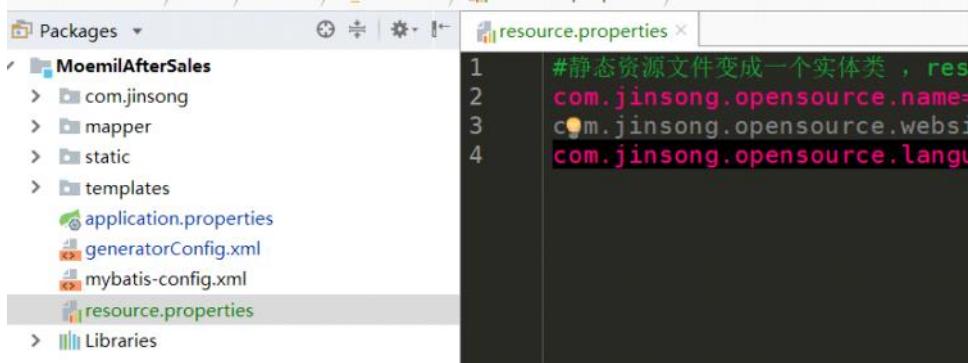
Springboot 静态资源转成类

2018年5月11日 15:21

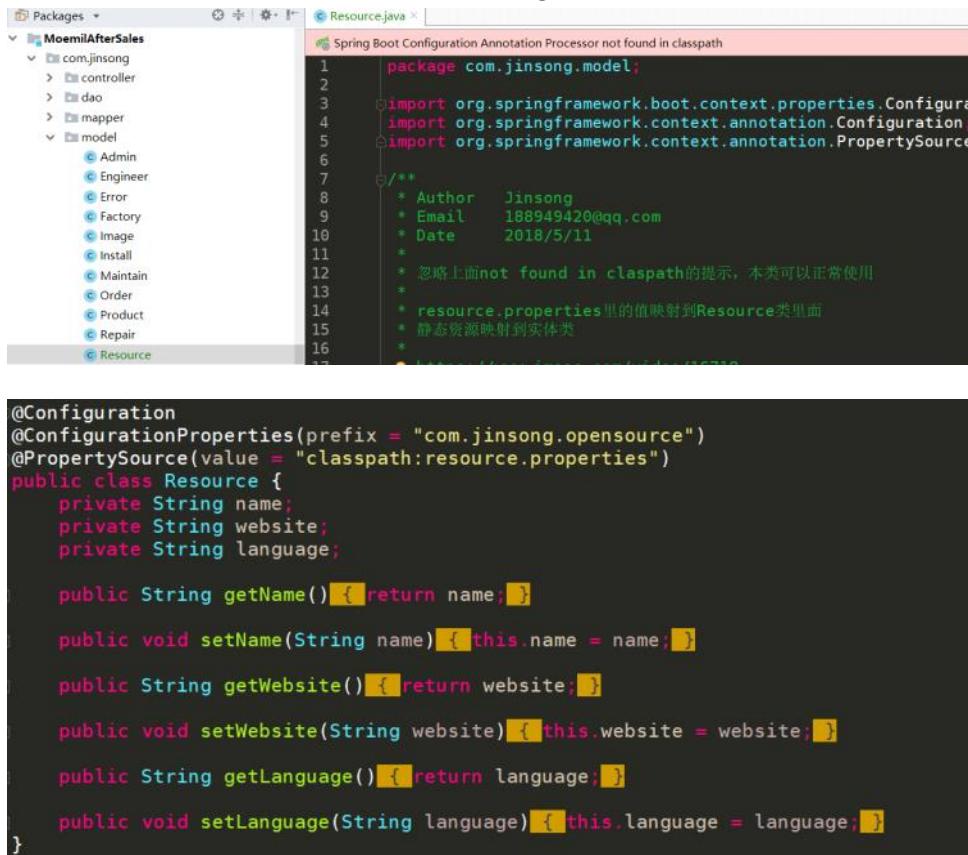
1、先引入依赖

```
<!-- 静态资源文件变成一个实体类，resource.properties 对应Resource类-->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-configuration-processor</artifactId>
    <optional>true</optional>
</dependency>
```

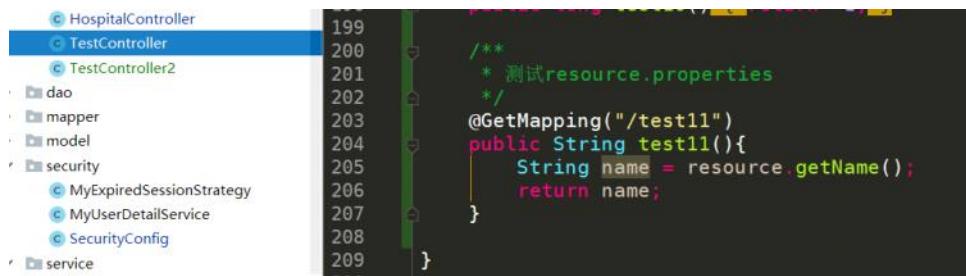
2、写resource.properties（这就是我们的静态资源，名字可以随意改）



3、写资源对应的类，对应的注解，成员变量，get/set方法



4、测试是OK的



The screenshot shows a Java code editor with a file tree on the left and a code editor window on the right. The file tree includes packages like HospitalController, TestController, TestController2, dao, mapper, model, security, and service. The code editor displays the following Java code:

```
199 /**
200  * 测试resource.properties
201 */
202 @GetMapping("/test11")
203 public String test11(){
204     String name = resource.getName();
205     return name;
206 }
207
208 }
209 }
```

Springboot 定时任务

2018年5月11日 15:21

[部署一个定时任务](#)

[定时任务cron表达式](#)

部署一个定时任务

1、启动类添加@EnableScheduling注解

```
//开启定时任务  
@EnableScheduling  
public class ImoocApplication {  
  
    public static void main(String[] args) {  
        SpringApplication.run(ImoocApplication.class, args);  
    }  
}
```

2、功能类中，添加@Component注解，方法体添加@Scheduled注解，注解中的fixedRate是定时任务执行的间隔时间

```
@Component  
public class TestTask {  
  
    private static final SimpleDateFormat dateFormat = new SimpleDateFormat("HH:mm:ss");  
  
    // 定义每过3秒执行任务  
    @Scheduled(fixedRate = 3000)  
    public void reportCurrentTime() {  
        System.out.println("现在时间：" + dateFormat.format(new Date()));  
    }  
}
```

3、直接启动项目，定时任务就会执行了

```
2017-12-08 20:34:20.247  INFO 11692 --- [ rest  
现在时间： 20:34:20  
2017-12-08 20:34:20.293  INFO 11692 --- [ rest  
2017-12-08 20:34:20.298  INFO 11692 --- [ rest  
现在时间： 20:34:23  
现在时间： 20:34:26
```

定时任务cron表达式

<https://www.imooc.com/video/16792>

```
@Scheduled(cron = "4-40 * * * * ?")  
public void reportCurrentTime() {  
    System.out.println("现在时间：" + dateFormat.format(new Date()));  
}
```

cron表达式在线生成地址 (Springboot不支持“年”)

<http://cron.qqe2.com/>

Springboot 热部署

2018年5月12日 12:50

[Tomcat热部署](#)

[Springboot热部署](#)

一句话解释就是修改代码、发布不需要重新启动服务器

热部署的使用场景

场景：

- ◆ 本地调试
- ◆ 线上发布

优点：

- 1、无论本地还是线上，都适用
- 2、无需重启服务器
 - 提高开发、调试效率
 - 提升发布、运维效率，降低运维成本

热部署与热加载的区别

部署方式

- ◆ 热部署在服务器运行时重新部署项目
- ◆ 热加载在运行时重新加载class

热加载是只重新加载修改的那个类，应用在开发环境中
而热部署是重新加载整个应用，应用在生产环境中

[Tomcat热部署](#)

1. Tomcat自带热部署功能，我们把war包直接扔进Tomcat的webapp下就行了。
2. 另外还有一种修改Tomcat配置文件的方法，具体看视频
<https://www.imooc.com/video/16057>

Springboot热部署

1、引入依赖

```
<!-- 热部署 -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
    <!-- optional=true, 依赖不会传递, 该项目依赖devtools; -->
    <optional>true</optional>
</dependency>
```

2、更改idea配置

- “File” -> “Settings” -> “Build,Execution,Deployment” -> “Compiler”，选中打勾 “Build project automatically”。
- 组合键：“Shift+Ctrl+Alt+/"，选择 “Registry”，选中打勾 “compiler.automake.allow.when.app.running”。

3、Chrome禁用缓存

F12或者 “Ctrl+Shift+I”，打开开发者工具，“Network” 选项卡下 选中打勾 “Disable Cache(while DevTools is open)”

4、Thymeleaf禁用缓存

```
#thymeleaf配置
#Spring-boot使用thymeleaf时默认是有缓存的
#你必须重新运行spring-boot的main()方法才能生效
#我们可以把thymeleaf的缓存关掉, 添加以下配置
spring.thymeleaf.cache=false
```

TIPS: 禁用devtools热部署

```
#devtools热部署, 设置为false就禁用, 默认是true
spring.devtools.restart.enabled=false
```

Mybatis 与Springboot集成

2017年8月1日 17:04

SQL写在哪？

XML提供SQL

注解提供SQL

实际应用中更多的把SQL写在XML中

- 1、因为注解其实还是写在源码里，写完了仍然要编译，写在XML中更方便SQL的调试
- 2、并且XML里写复杂的SQL语句时思路更清晰

XML版

以我自己的项目mySecondKill为例

目录结构

```
我的 mySecondKill [boot]
  +-- src/main/java
    +-- com.jinsong
    |   +-- com.jinsong.controller
    |   +-- com.jinsong.dao
    |       +-- SeckillDAO.java
    |       +-- SuccessKilledDAO.java
    |   +-- com.jinsong.model
    |       +-- Seckill.java
    |       +-- SuccessKilled.java
  +-- src/main/resources
    +-- mapper
        +-- SeckillDAO.xml
        +-- SuccessKilledDAO.xml
    +-- sql
        +-- static
        +-- templates
        +-- application.properties
        +-- mybatis-config.xml
```

- 1、Maven项目的pom.xml添加mybatis和mysql依赖

mybatis是mybatis-spring-boot-starter

The screenshot shows the IntelliJ IDEA interface with the 'Dependencies' tab selected. The list of dependencies includes:

- spring-cloud-starter (managed:1.2.3.RELEASE)
- mybatis-spring-boot-starter : 1.3.0
- spring-boot-starter-web (managed:1.5.6.RELEASE)
- mysql-connector-java [runtime] (managed:5.1.43)
- spring-boot-starter-test [test] (managed:1.5.6.RELEASE)

2、application.properties中配置

分别是数据库配置、mybatis基本配置、mybatis XML方式配置

注意：

mybatis.type-aliases-package=com.jinsong.model指定DataSource (数据库模型)

mybatis.mapper-locations=classpath:mapper/*.xml指定我们的mybatis XML文件位置

```
#数据库配置
spring.datasource.url=jdbc:mysql://localhost:3306/mysecondkill?useUnicode=true&characterEncoding=utf-8
spring.datasource.username=root
spring.datasource.password=83862973

#mybatis全局基本配置
mybatis.config-location=classpath:mybatis-config.xml

#使用mybatis XML 方式写SQL 就要添加以下两行参数
#type-aliases-package指定Datasource
#使用type-aliases-package，需要配合自动扫描Mappers使用，需要在Mapper接口上标注@Mapper，否则失败
mybatis.type-aliases-package=com.jinsong.model

#mapper-locations这个配置参数当mapper xml与mapper class不在同一个目录时添加，指定与mapper对应的xml文件在哪里
mybatis.mapper-locations=classpath:mapper/*.xml
```

3、DAO层 (Mapper层)

DAO层一定要加上@Mapper

```

1 package com.jinsong.dao;
2
3 import java.util.Date;
4
5 @Mapper
6 public interface SeckillDAO {
7
8     /**
9      * 减库存
10     * @param seckillId
11     * @param killTime
12     * @return 如果返回值>1, 表示更新库存的记录行数, 返回值=0, 表示更新失败
13     */
14     int reduceNumber(@Param("seckillId") long seckillId,@Param("killTime")
15
16     /**
17      * 根据id查询秒杀的商品信息
18      * @param seckillId
19      * @return
20      */
21     Seckill queryById(long seckillId);
22
23 }

```

4、 mapper文件夹下写XML

注意mapper的目录

```

v src/main/resources
  v mapper
    x SeckillDAO.xml
    x SuccessKilledDAO.xml
  > sql
  static
  templates
  application.properties
  mybatis-config.xml

```

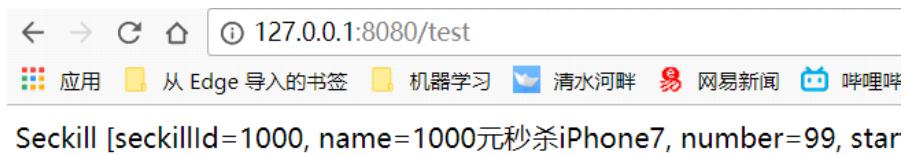
```

1 <?xml version="1.0" encoding="UTF-8" ?>
2 <!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
3   "http://mybatis.org/dtd/mybatis-3-mapper.dtd" >
4
5 <mapper namespace="com.jinsong.dao.SeckillDAO" > <!-- 这里标注了对应方法的具体位置
6 <!--目的:为dao接口方法提供sql语句配置 即针对dao接口中的方法编写我们的sql语句-->
7
8 <!-- 下面依次对应DAO层对应类的每个接口方法 -->
9     <update id="reduceNumber">
10       UPDATE seckill
11       SET number = number - 1
12       WHERE seckill_id=#{seckillId}
13       AND start_time <![CDATA[ <= ]]> #{killTime} <!-- <=符号与SQL语句有歧义
14       AND end_time >= #{killTime}
15       AND number > 0
16     </update>
17
18     <select id="queryById" resultType="Seckill" parameterType="long">
19       SELECT *
20       FROM seckill
21       WHERE seckill_id=#{seckillId}
22     </select>

```

5、 控制器中写逻辑然后测试

```
1 package com.jinsong.controller;
2
3 import org.springframework.beans.factory.annotation.Autowired;
4
5 @RestController
6 public class TestController {
7
8     @Autowired
9     private SeckillDAO seckillDAO;
10
11     @GetMapping(value="/test")
12     public String myTest() {
13         Seckill seckill = seckillDAO.queryById(1000);
14         return seckill.toString();
15     }
16 }
17
18 }
```



OK!!!

注解版

来自 <<http://blog.csdn.net/gebitan505/article/details/54929287>>

我也用的注解版，参考这篇文章

1、maven项目的pom.xml添加mybatis和mysql依赖

⚠ Dependencies Overriding managed version 5.1

Dependencies

- spring-boot-starter-web (managed:1.5.6.RELEASE)
- spring-boot-starter-test [test] (managed:1.5.6)
- mybatis-spring-boot-starter : 1.1.1
- mysql-connector-java : 5.1.39 (managed:5.1.43)

2、application.yml中配置文件

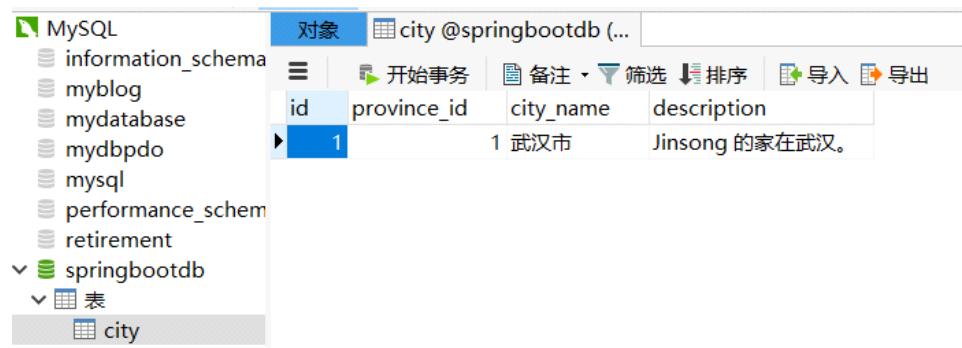
注意更改**数据库名称**和账户、密码

```
##数据库配置
spring.datasource.driverClassName: com.mysql.jdbc.Driver
spring.datasource.url: jdbc:mysql://localhost:3306/springbootdb?useUnicode=true&characterEncoding=utf8
spring.datasource.username: root
spring.datasource.password: 83862973
```

以上配置就OK拉！！！！

应用mybatis

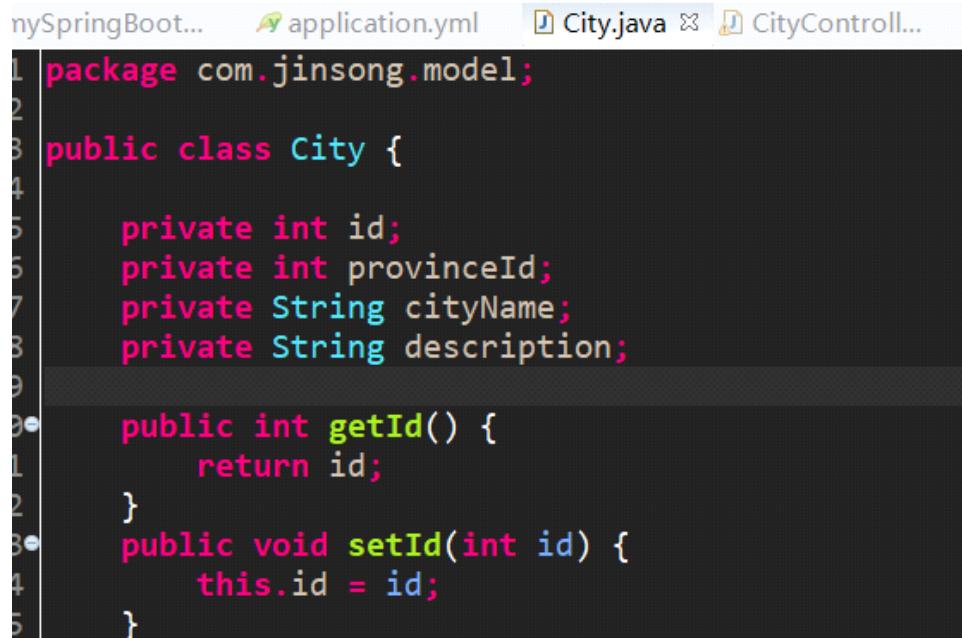
1、数据库



	对象	city @springbootdb (1)	
MySQL	信息模式		
	information_schema		
	myblog		
	mydatabase		
	mydbpdo		
	mysql		
	performance_schema		
	retirement		
springbootdb	表		
	city		

	开始事务	备注	筛选
	排序	导入	导出
id	1	1 武汉市	Jinsong 的家在武汉。

2、Model层



```
1 package com.jinsong.model;
2
3 public class City {
4
5     private int id;
6     private int provinceId;
7     private String cityName;
8     private String description;
9
10    public int getId() {
11        return id;
12    }
13    public void setId(int id) {
14        this.id = id;
15    }
16}
```

3、Dao层也就是Mapper层

注意：

在 Application 应用启动类添加注解 `MapperScan`（“包名”）

可以扫描整个Mapper层，这样Mapper中的类就不用每个在开头添加`@Mapper`了

```
mySpringBoot... application.yml CityMapper.java City.java CityControll...  
1 package com.jinsong.Dao;  
2  
3 import java.util.List;  
4  
5 @Mapper //如果在启动类MySpringBootTestAppllicatin.java中添加了@MapperScan  
6 public interface CityMapper {  
7  
8     @Select("SELECT * FROM city") //查询语句  
9     @Results({  
10         @Result(property="provinceId",column="province_id"),  
11         @Result(property="cityName",column="city_name")  
12     }) //@Result返回修饰的结果集，数据库中字段名是city_name，这里修饰为cityName  
13     List<City> getAll();  
14 }  
15  
16
```

4、控制器

```
mySpringBoot... CityControll... application.yml CityMapper.java City.java  
1 package mySpringBootTest/pom.xml er;  
2  
3 import java.util.List;  
4  
5 @RestController  
6 public class CityController {  
7  
8     @Autowired  
9     private CityMapper cityMapper;  
10  
11     @GetMapping(value="/city")  
12     public String testCity() {  
13         List<City> list= cityMapper.getAll();  
14         City city =list.get(0);  
15         return city.getCityName()+city.getDescription();  
16     }  
17 }
```

5、页面显示



武汉市Jinsong 的家在武汉。

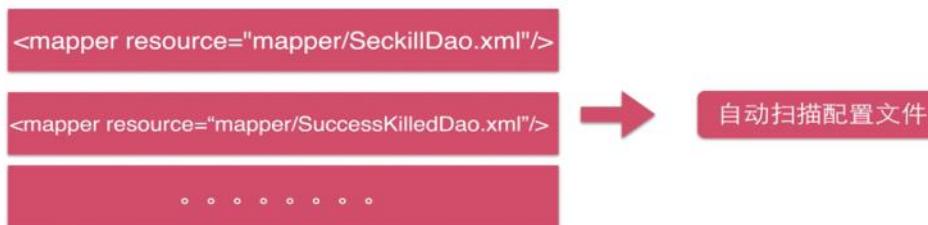
MyBatis与Spring整合之后的优点

<http://www.imooc.com/video/11713>

更少的配置-dao实现



更少的配置-配置扫描



更少的配置-别名



足够的灵活性



Mybatis

2017年8月1日 17:04

SQL写在哪？

XML提供SQL

注解提供SQL

实际应用中更多的把SQL写在XML中

- 1、因为注解其实还是写在源码里，写完了仍然要编译，写在XML中更方便SQL的调试
- 2、并且XML里写复杂的SQL语句时思路更清晰

XML版

以我自己的项目mySecondKill为例

目录结构

```
└─ mySecondKill [boot]
    ├─ src/main/java
    │   ├─ com.jinsong
    │   ├─ com.jinsong.controller
    │   └─ com.jinsong.dao
    │       ├─ SeckillDAO.java
    │       ├─ SuccessKilledDAO.java
    │       └─ com.jinsong.model
    │           ├─ Seckill.java
    │           └─ SuccessKilled.java
    └─ src/main/resources
        ├─ mapper
        │   └─ SeckillDAO.xml
        │   └─ SuccessKilledDAO.xml
        ├─ sql
        │   ├─ static
        │   └─ templates
        └─ application.properties
            └─ mybatis-config.xml
```

- 1、Maven项目的pom.xml添加mybatis和mysql依赖

mybatis是mybatis-spring-boot-starter

The screenshot shows the 'Dependencies' tab in an IDE. It lists the following dependencies:

- spring-cloud-starter (managed:1.2.3.RELEASE)
- mybatis-spring-boot-starter : 1.3.0
- spring-boot-starter-web (managed:1.5.6.RELEASE)
- mysql-connector-java [runtime] (managed:5.1.43)
- spring-boot-starter-test [test] (managed:1.5.6.RELEASE)

2、application.properties中配置

分别是数据库配置、 mybatis基本配置、 mybatis XML方式配置

注意：

mybatis.type-aliases-package=com.jinsong.model指定DataSource (数据库模型)

mybatis.mapper-locations=classpath:mapper/*.xml指定我们的mybatis XML文件位置

```
#数据库配置
spring.datasource.url=jdbc:mysql://localhost:3306/mysecondkill?useUnicode=true&characterEncoding=utf-8
spring.datasource.username=root
spring.datasource.password=83862973

#mybatis全局基本配置
mybatis.config-location=classpath:mybatis-config.xml

#使用mybatis XML 方式写SQL 就要添加以下两行参数
#type-aliases-package指定Datasource
#使用type-aliases-package，需要配合自动扫描Mappers使用，需要在Mapper接口上标注@Mapper，否则失败
mybatis.type-aliases-package=com.jinsong.model

#mapper-locations这个配置参数当mapper xml与mapper class不在同一个目录时添加，指定与mapper对应的xml文件在哪里
mybatis.mapper-locations=classpath:mapper/*.xml
```

3、DAO层 (Mapper层)

DAO层一定要加上@Mapper

```
1 package com.jinsong.dao;
2
3 import java.util.Date;
4
5 @Mapper
6 public interface SeckillDAO {
7
8     /**
9      * 减库存
10     * @param seckillId
11     * @param killTime
12     * @return 如果返回值>1, 表示更新库存的记录行数, 返回值=0, 表示更新失败
13     */
14     int reduceNumber(@Param("seckillId") long seckillId,@Param("killTime")
15
16     /**
17      * 根据id查询秒杀的商品信息
18      * @param seckillId
19      * @return
20      */
21     Seckill queryById(long seckillId);
22
23 }
```

4、 mapper文件夹下写XML

注意mapper的目录

```
✓ src/main/resources
  ↘ mapper
    ✘ SeckillDAO.xml
    ✘ SuccessKilledDAO.xml
  > ↗ sql
  ↗ static
  ↗ templates
  🌟 application.properties
  ✘ mybatis-config.xml
```

```
1 <?xml version="1.0" encoding="UTF-8" ?>
2 <!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
3   "http://mybatis.org/dtd/mybatis-3-mapper.dtd" >
4
5 <mapper namespace="com.jinsong.dao.SeckillDAO" <!-- 这里标注了对应方法的具体位置
6 <!--目的:为dao接口方法提供sql语句配置 即针对dao接口中的方法编写我们的sql语句-->
7
8 <!-- 下面依次对应DAO层对应类的每个接口方法 -->
9     <update id="reduceNumber">
10       UPDATE seckill
11       SET number = number - 1
12       WHERE seckill_id=#{seckillId}
13       AND start_time <![CDATA[ <= ]]> #{killTime} <!-- <=符号与SQL语句有歧义
14       AND end_time >= #{killTime}
15       AND number > 0
16     </update>
17
18     <select id="queryById" resultType="Seckill" parameterType="long">
19       SELECT *
20       FROM seckill
21       WHERE seckill_id=#{seckillId}
22     </select>
```

5、 控制器中写逻辑然后测试

```
1 package com.jinsong.controller;
2
3 import org.springframework.beans.factory.annotation.Autowired;
4
5 @RestController
6 public class TestController {
7
8     @Autowired
9     private SeckillDAO seckillDAO;
10
11     @GetMapping(value="/test")
12     public String myTest() {
13         Seckill seckill = seckillDAO.queryById(1000);
14         return seckill.toString();
15     }
16 }
17
18 }
```



OK!!!

注解版

来自 <<http://blog.csdn.net/gebitan505/article/details/54929287>>

我也用的注解版，参考这篇文章

1、maven项目的pom.xml添加mybatis和mysql依赖

⚠ Dependencies Overriding managed version 5.1

Dependencies
spring-boot-starter-web (managed:1.5.6.RELEASE)
spring-boot-starter-test [test] (managed:1.5.6)
mybatis-spring-boot-starter : 1.1.1
mysql-connector-java : 5.1.39 (managed:5.1.43)

2、application.yml中配置文件

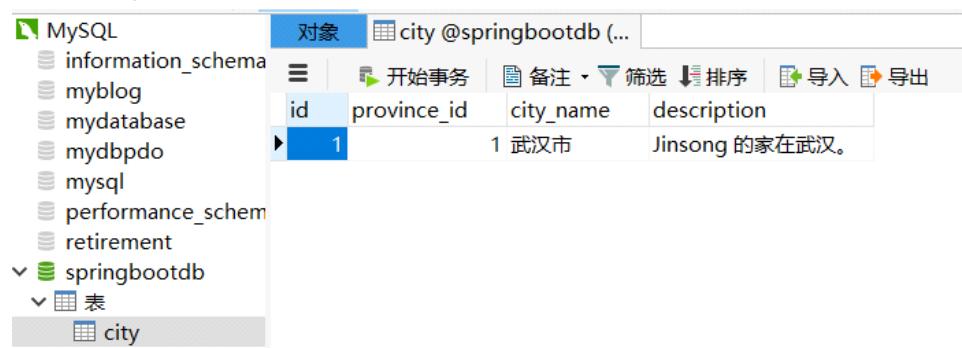
注意更改**数据库名称**和账户、密码

```
##数据库配置
spring.datasource.driverClassName: com.mysql.jdbc.Driver
spring.datasource.url: jdbc:mysql://localhost:3306/springbootdb?useUnicode=true&characterEncoding=utf8
spring.datasource.username: root
spring.datasource.password: 83862973
```

以上配置就OK拉！！！！

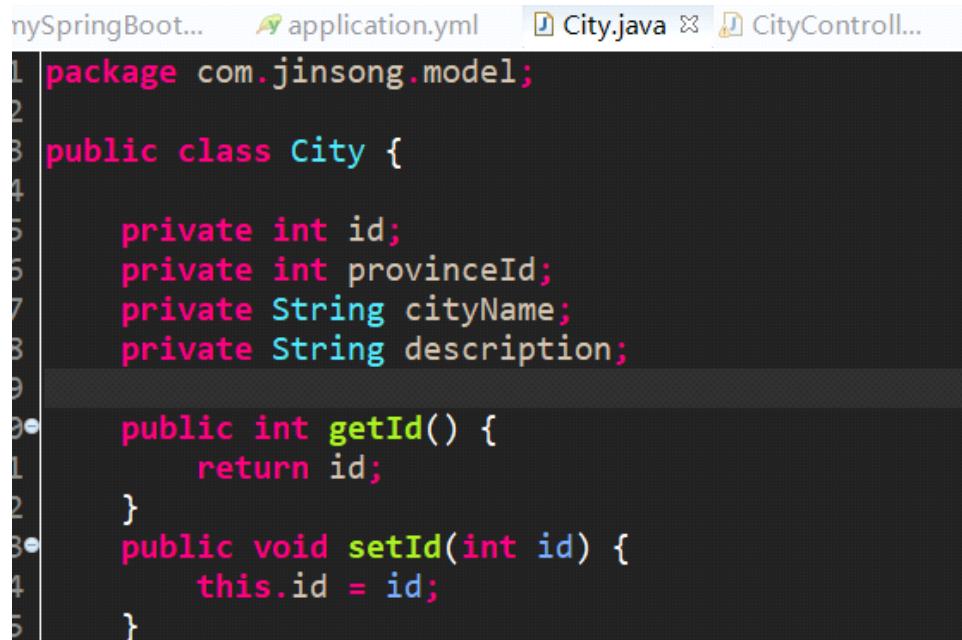
应用mybatis

1、数据库



		id	province_id	city_name	description
		1	1	武汉市	Jinsong 的家在武汉。

2、Model层



```
1 package com.jinsong.model;
2
3 public class City {
4
5     private int id;
6     private int provinceId;
7     private String cityName;
8     private String description;
9
10    public int getId() {
11        return id;
12    }
13    public void setId(int id) {
14        this.id = id;
15    }
16}
```

3、Dao层也就是Mapper层

注意：

在 Application 应用启动类添加注解 `MapperScan`（“包名”）

可以扫描整个Mapper层，这样Mapper中的类就不用每个在开头添加`@Mapper`了

```
mySpringBoot... application.yml CityMapper.java City.java CityControll...  
1 package com.jinsong.Dao;  
2  
3 import java.util.List;  
4  
5 @Mapper //如果在启动类MySpringBootTestAppllicatin.java中添加了@MapperScan  
6 public interface CityMapper {  
7  
8     @Select("SELECT * FROM city") //查询语句  
9     @Results({  
10         @Result(property="provinceId",column="province_id"),  
11         @Result(property="cityName",column="city_name")  
12     }) //@Result返回修饰的结果集，数据库中字段名是city_name，这里修饰为cityName  
13     List<City> getAll();  
14 }  
15  
16
```

4、控制器

```
mySpringBoot... CityControll... application.yml CityMapper.java City.java  
1 package mySpringBootTest/pom.xml er;  
2  
3 import java.util.List;  
4  
5 @RestController  
6 public class CityController {  
7  
8     @Autowired  
9     private CityMapper cityMapper;  
10  
11     @GetMapping(value="/city")  
12     public String testCity() {  
13         List<City> list= cityMapper.getAll();  
14         City city =list.get(0);  
15         return city.getCityName()+city.getDescription();  
16     }  
17 }
```

5、页面显示



武汉市Jinsong 的家在武汉。

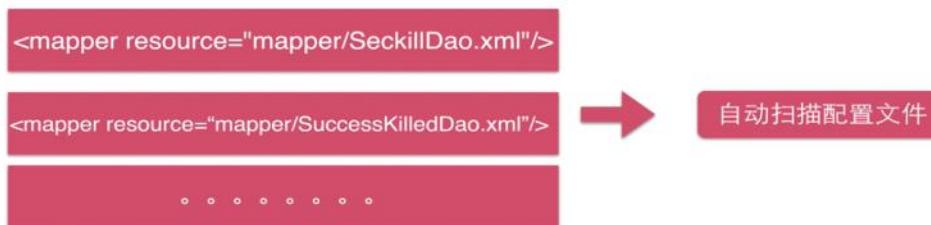
MyBatis与Spring整合之后的优点

<http://www.imooc.com/video/11713>

更少的配置-dao实现



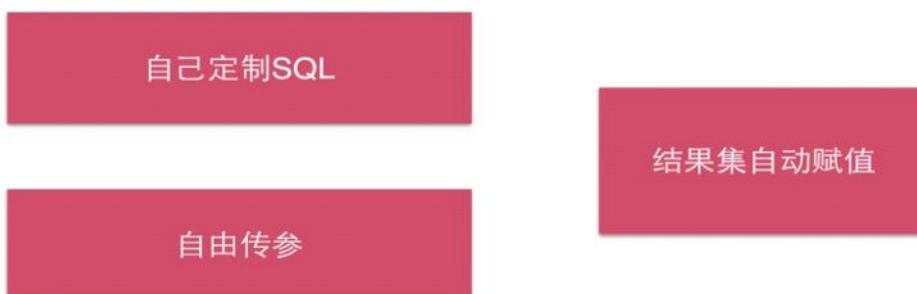
更少的配置-配置扫描



更少的配置-别名



足够的灵活性



Mybatis Generator

2018年2月6日 10:49

使用Mybatis-Generator可以帮助我们自动生成Model、Mapper，不用自己手写，非常方便。但如果涉及联表查询，还是得自己手动修改。MBG自动生成的文件可以与我们手写的文件混用。

注意：每次数据库表结构变化后，都要重新运行mvn mybatis-generator:generate来自动生成，重新运行后记得添加@Mapper注解，只会根据主键来生成select\delete语句，并且一个数据库表一般建议是只有一个主键。

一、先在pom.xml中添加依赖，注意配置的参数

MoemilAfterSales项目中有代码示例

```
</dependencies>

<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
        <!-- mybatis generator插件 -->
        <plugin>
            <groupId>org.mybatis.generator</groupId>
            <artifactId>mybatis-generator-maven-plugin</artifactId>
            <version>1.3.2</version>
            <configuration>
                <!-- configurationFile定义配置文件的位置 -->
                <configurationFile>${basedir}/src/main/resources/generatorConfig.xml</configurationFile>
                <!-- 执行过程会输出到控制台 -->
                <verbose>true</verbose>
                <!-- 如果生成的Java文件存在已经同名的文件，新生成的文件会覆盖原有的文件 -->
                <overwrite>true</overwrite>
            </configuration>
        </plugin>
    </plugins>
</build>
```

二、写generatorConfig.xml配置文件

具体看项目，这里说几个要修改的地方

- (1) <classPathEntry> 在Maven Dependencies依赖包中找到mysql-connector-java的路径填到这里
- (2) <jdbcConnection> 注意connectionURL属性中各参数用&分隔
- (3) <javaModelGenerator> 配置自动生成Model存放的包名
- (4) <sqlMapGenerator> 配置自动生成SQL语句XML存放的文件名
- (5) <javaClientGenerator> 配置自动生成Mapper存放的包名
- (6) <table> 配置要生成的表信息

其他详细配置查看下面文档

来自 http://blog.csdn.net/testcs_dn/article/details/77881776

三、项目根目录下命令行运行 mvn mybatis-generator:generate



```
yMapper.java was overwritten
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 3.673 s
[INFO] Finished at: 2018-02-06T11:03:05+08:00
[INFO] Final Memory: 14M/165M
[INFO] -----
```

四、**BUILD SUCCESS**后自动生成了以下文件，不用手写Model和sql语句，非常方便

```
✓ com.jinsong.mapper
  > AdminMapper.java
  > EngineerMapper.java
  > ErrorMapper.java
  > FactoryMapper.java
  > InstallMapper.java
  > MaintainMapper.java
  > ProductMapper.java
  > RepairMapper.java
✓ com.jinsong.model
  > Admin.java
  > Engineer.java
  > Error.java
  > Factory.java
  > Install.java
  > Maintain.java
  > Order.java
  > Product.java
  > Repair.java

✓ src/main/resources
  ✓ mapper
    AdminMapper.xml
    EngineerMapper.xml
    ErrorMapper.xml
    FactoryMapper.xml
    InstallMapper.xml
    MaintainMapper.xml
    ProductMapper.xml
    RepairMapper.xml
    TestDAO.xml
```

五、生成的Mapper文件就是DAO层，xml就是DAO层对应的SQL xml，具体调用示例看

MoemilAfterSales的TestController中的test6方法。

Spring AOP

2017年2月20日 18:40

[注解的方式配置SpringAOP](#)

[xml文件的方式配置SpringAOP](#)

AOP是面向切面编程，是一种**编程范式**，下面列出了所有编程范式

编程范式概览

面向过程编程

面向对象编程

函数式编程

事件驱动编程

面向切面编程

面向切面编程是面向对象编程的互补，并不冲突

AOP的好处：AOP主要是提供另外一种编程思路，可以把类似的行为抽离出来统一处理，比如权限控制，不用AOP，我得在每个方法里写权限验证，用AOP，就抽离出来了，减少了代码的侵入，当然，用过滤器、拦截器也可以啦~~

AOP的应用场景

权限控制

缓存控制

事务控制

审计日志

性能监控

分布式追踪

异常处理

AOP 前奏

- WHY AOP ?



如果实现上面加减乘除方法的需求1-日志，如下面的代码

```
calculator.java | CalculatorImpl.java | Main.java
package com.js.aop;

public class CalculatorImpl implements Calculator {

    @Override
    public int add(int i, int j) {
        //这些sysout输出相当于在输出日志内容
        System.out.println("The method add begins with ["+i+","+j+"]");
        int result = i + j;
        System.out.println("The method add ends with"+result);
        return result;
    }

    @Override
    public int sub(int i, int j) {
        System.out.println("The method sub begins with ["+i+","+j+"]");
        int result = i - j;
        System.out.println("The method sub ends with"+result);
        return result;
    }

    @Override
    public int mul(int i, int j) {
        System.out.println("The method mul begins with ["+i+","+j+"]");
        int result = i * j;
        System.out.println("The method mul ends with"+result);
        return result;
    }

    @Override
    public int div(int i, int j) {
        System.out.println("The method div begins with ["+i+","+j+"]");
        int result = i / j;
        System.out.println("The method div ends with"+result);
        return result;
    }
}
```

这样写就会造成下面的问题

什么问题呢？就是加减乘除4个方法里，都写了日志输出语句，而且这些日志输出语句的形式还是一样的，如果我想修改日志输出语句，就得一个一个改，如果是40个呢？改到明年去了

问题

代码混乱：越来越多的非业务需求(日志和验证等)加入后，原有的业务方法急剧膨胀。每个方法在处理核心逻辑的同时还必须兼顾其他多个关注点。

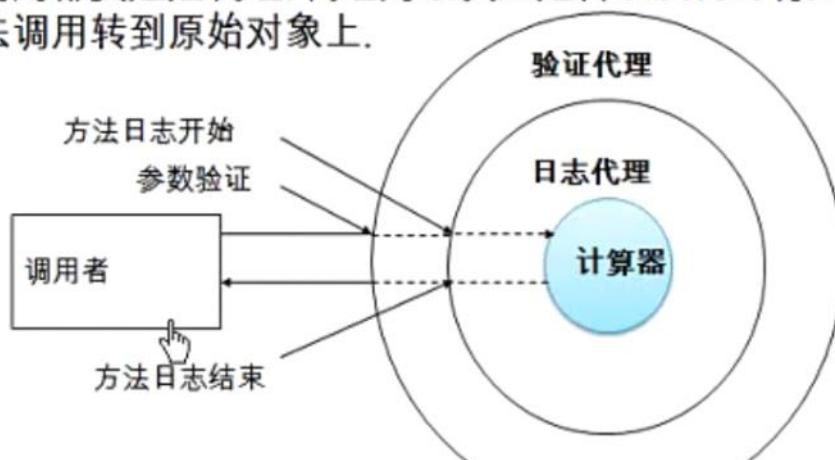
代码分散：以日志需求为例，只是为了满足这个单一需求，就不得不在多个模块（方法）里多次重复相同的日子代码。如果日志需求发生变化，必须修改所有模块。

那么这个问题如何解决呢？

使用动态代理

使用动态代理解决上述问题

代理设计模式的原理：使用一个代理将对象包装起来，然后用该代理对象取代原始对象。任何对原始对象的调用都要通过代理。代理对象决定是否以及何时将方法调用转到原始对象上。



代码如下：

```
calculator.java | CalculatorLoggingProxy.java | CalculatorImpl_Log.java | Main.java | CalculatorImpl.java  
pac SpringStudyAOP/src/com/jx/aop/Calculator.java  
  
import java.lang.reflect.InvocationHandler;  
import java.lang.reflect.Method;  
import java.lang.reflect.Proxy;  
import java.util.Arrays;  
  
//动态代理  
public class CalculatorLoggingProxy {  
  
    //要代理的对象  
    private Calculator target;  
  
    public CalculatorLoggingProxy(Calculator target){  
        this.target=target;  
    }  
  
    public Calculator getLoggingProxy(){  
        Calculator proxy =null;  
  
        //代理对象由哪一个类加载器负责加载  
        ClassLoader loader=target.getClass().getClassLoader();  
  
        //代理对象的类型，即其中有哪些方法  
        Class[] interfaces=new Class[]{Calculator.class},
```

```

//当调用代理对象其中的方法时，执行该代码
InvocationHandler h=new InvocationHandler() {

    /**
     * proxy:正在返回的那个代理对象，一般情况下，在invoke方法中都不使用该对象
     * method:正在被调用的方法
     * args:调用方法时，传入的参数
     */
    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        String methodName=method.getName();
        //日志
        System.out.println("The method "+methodName+" begins with"+Arrays.asList(args));
        //执行方法
        Object result=method.invoke(target, args);
        //日志
        System.out.println("The method "+methodName+" begins with"+result);
        return result;
    }
}
proxy=(Calculator) Proxy.newProxyInstance(loader, interfaces, h);

return proxy;
}

```

```

calculator.java  CalculatorLoggingProxy.java  CalculatorImpl_Log.java  Main.java  CalculatorImpl.java
package com.js.aop;

public class Main {

    public static void main(String[] args) {

        //Calculator是接口，想要新建对象，必须new他的实现类
        //Calculator c=new Calculator_Log(); //这是会报错的
        //Calculator c=new CalculatorImpl_Log(); //这才是对的

        // System.out.println("---->"+c.add(1, 2));
        // System.out.println("---->"+c.div(4,2));

        //使用代理
        Calculator target=new CalculatorImpl();

        Calculator proxy=new CalculatorLoggingProxy(target).getLoggingProxy();

        System.out.println("---->"+proxy.add(1, 2));
        System.out.println("---->"+proxy.div(4,2));
    }
}

```

总而言之，实现动态代理是很麻烦的！！！

然而我们Spring的AOP就能简化这一过程！！！

AOP 简介

AOP(Aspect-Oriented Programming, 面向切面编程): 是一种新的方法论, 是对传统 OOP(Object-Oriented Programming, 面向对象编程) 的补充.

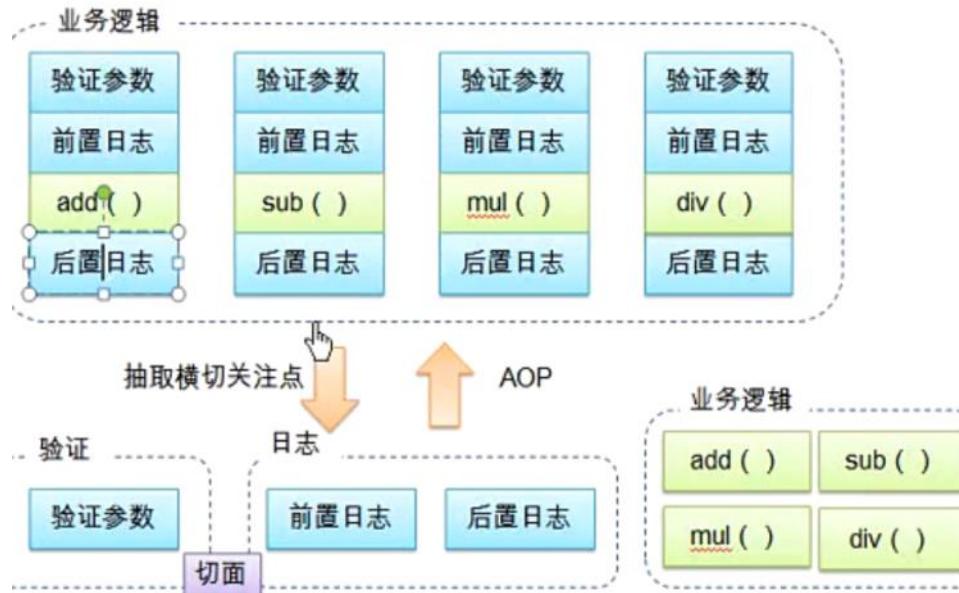
AOP 的主要编程对象是 **切面**(aspect), 而 **切面模块化横切关注点**.

在应用 AOP 编程时, 仍然需要**定义公共功能**, 但可以明确的**定义这个功能在哪里, 以什么方式应用, 并且不必修改受影响的类**. 这样一来**横切关注点就被模块化到特殊的对象(切面)里**.

AOP 的好处:

- 每个事物逻辑位于一个位置, 代码不分散, 便于维护和升级
- 业务模块更简洁, 只包含核心业务代码.

AOP



AOP 术语

切面(Aspect): 横切关注点(跨越应用程序多个模块的功能)被模块化的特殊对象

通知(Advice): 切面必须要完成的工作

目标(Target): 被通知的对象

代理(Proxy): 向目标对象应用通知之后创建的对象

连接点 (Joinpoint) : 程序执行的某个特定位置 : 如类某个方法调用前、调用后, 方法抛出异常后等。连接点由两个信息确定 : 方法表示的程序执行点相对点表示的方位。例如 `ArithmetricCalculator#add()` 方法执行前的连接点执行点为 `ArithmetricCalculator#add()` ; 方位为该方法执行前的位置

切点 (Pointcut) : 每个类都拥有多个连接点 : 例如 `ArithmetricCalculator` 的所有方法实际上都是连接点, 即连接点是程序类中客观存在的事务。AOP 通过切点定位到特定的连接点。类比 : 连接点相当于数据库中的记录, 切点相当于查询条件。切点和连接点不是一对一的关系, 一个切点匹配多个连接点, 切点通过 `org.springframework.aop.Pointcut` 接口进行描述, 它使用类和方法作为连接点的查询条件。

Spring AOP

AspectJ : Java 社区里最完整最流行的 AOP 框架.

在 Spring2.0 以上版本中, 可以使用基于 `AspectJ` 注解或基于 XML 配置的 AOP

AspectJ

在 Spring 中启用 AspectJ 注解支持

要在 Spring 应用中使用 AspectJ 注解, 必须在 classpath 下包含 AspectJ 类库: aopalliance.jar、aspectj.weaver.jar 和 spring-aspects.jar

将 aop Schema 添加到 <beans> 根元素中.

要在 Spring IOC 容器中启用 AspectJ 注解支持, 只要在 Bean 配置文件中定义一个空的 XML 元素 <aop:aspectj-autoproxy>

当 Spring IOC 容器侦测到 Bean 配置文件中的 <aop:aspectj-autoproxy> 元素时, 会自动为与 AspectJ 切面匹配的 Bean 创建代理.

用 AspectJ 注解声明切面

- 要在 Spring 中声明 AspectJ 切面, 只需要在 IOC 容器中将切面声明为 Bean 实例. 当在 Spring IOC 容器中初始化 AspectJ 切面之后, Spring IOC 容器就会为那些与 AspectJ 切面相匹配的 Bean 创建代理.
- 在 AspectJ 注解中, 切面只是一个带有 @Aspect 注解的 Java 类.
- 通知是标注有某种注解的简单的 Java 方法.
- AspectJ 支持 5 种类型的通知注解:
 - @Before: 前置通知, 在方法执行之前执行
 - @After: 后置通知, 在方法执行之后执行
 - @AfterReturning: 返回通知, 在方法返回结果之后执行
 - @AfterThrowing: 异常通知, 在方法抛出异常之后
 - @Around: 环绕通知, 围绕着方法执行

利用方法签名编写 AspectJ 切入点表达式

最典型的切入点表达式时根据方法的签名来匹配各种方法:

- execution * com.atguigu.spring.ArithmeticCalculator.*(..): 匹配 ArithmeticCalculator 中声明的所有方法, 第一个 * 代表任意修饰符及任意返回值. 第二个 * 代表任意方法... 匹配任意数量的参数. 若目标类与接口与该切面在同一个包中, 可以省略包名.
- execution public* ArithmeticCalculator.*(..): 匹配 ArithmeticCalculator 接口的 **所有公有方法**.
- execution publicdouble ArithmeticCalculator.*(..): 匹配 ArithmeticCalculator 中 **返回 double 类型数值的方法**
- execution publicdouble ArithmeticCalculator.*(double,..): 匹配第一个参数为 double 类型的方法, .. 匹配任意数量任意类型的参数
- execution publicdouble ArithmeticCalculator.*(double, double): 匹配参数类型为 double, double 类型的方法.

让通知访问当前连接点的细节

- 可以在通知方法中声明一个类型为 JoinPoint 的参数. 然后就能访问链接细节. 如方法名称和参数值.

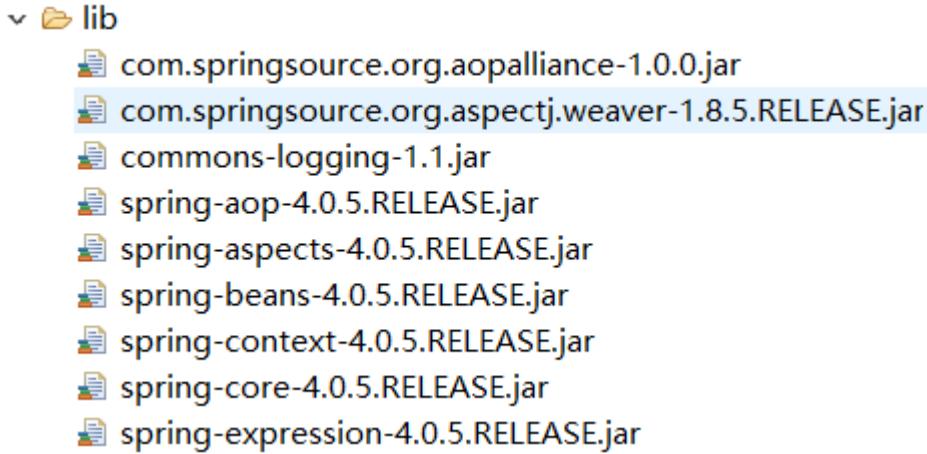
```
@Aspect
public class CalculatorLoggingAspect {
    private Log log = LoggerFactory.getLog(this.getClass());

    @Before("execution(* *.{..})")
    public void logBefore(JoinPoint joinPoint){
        log.info("The method " + joinPoint.getSignature().getName()
            + "() begins with " + Arrays.toString(joinPoint.getArgs()));
    }
}
```

标识这个方法是个前置通知, 切点表达式表示执行任意类的任意方法. 第一个 * 代表匹配任意修饰符及任意返回值, 第二个 * 代表任意类的对象, 第三个 * 代表任意方法, 参数列表中的 .. 匹配任意数量的参数

用注解的方式配置SpringAOP

1. 需要加入的jar包:



2. 在配置文件中加入aop的命名空间

Configure Namespaces

Select XSD namespaces to use in the configuration file

- aop - http://www.springframework.org/schema/aop
- beans - http://www.springframework.org/schema/beans
- c - http://www.springframework.org/schema/c
- cache - http://www.springframework.org/schema/cache
- context - http://www.springframework.org/schema/context
- jee - http://www.springframework.org/schema/jee
- lang - http://www.springframework.org/schema/lang
- p - http://www.springframework.org/schema/p
- task - http://www.springframework.org/schema/task
- util - http://www.springframework.org/schema/util

3. 基于注解的方式，在配置文件中加入如下配置：

```

Main.java applicationContext.xml LoggingAspect.java CalculatorImpl.java Calculator.java
1<?xml version="1.0" encoding="UTF-8"?>
2<beans xmlns="http://www.springframework.org/schema/beans"
3    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4    xmlns:aop="http://www.springframework.org/schema/aop"
5    xmlns:context="http://www.springframework.org/schema/context"
6    xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.spring
7        http://www.springframework.org/schema/context http://www.springframework.org/
8        http://www.springframework.org/schema/aop http://www.springframework.org/sch
9
10<!-- 配置自动扫描的包 -->
11<context:component-scan base-package="com.js.aop.impl"></context:component-scan>
12
13<!-- 使AspectJ注解起作用：自动为匹配的类生成代理对象 -->
14<aop:aspectj-autoproxy></aop:aspectj-autoproxy>
```

4. 把横切关注点的代码抽象到切面的类中，新建一个java文件如下的LoggingAspect.java 专门负责日志的输出

- a. 切面首先是一个IOC中的bean，即加入@Component注解
- b. 切面还需要加入@Aspect注解

```
1 package com.js.aop.impl;
2
3 import java.util.Arrays;
4 import java.util.List;
5
6 import org.aopalliance.intercept.Joinpoint;
7 import org.aspectj.lang.JoinPoint;
8 import org.aspectj.lang.annotation.Aspect;
9 import org.aspectj.lang.annotation.Before;
10 import org.springframework.stereotype.Component;
11
12 //把这个类声明为一个切面：1.需要把该类放入到IOC容器中
13 @Component
14 // 2.再声明为一个切面
15 @Aspect
16 public class LoggingAspect {
```

5. 在类中声明各种通知

- a. 声明一个方法
- b. 在方法前加入各种注解，比如@Before注解表示前置通知，在方法执行前执行，@After方法表示后置通知，在方法执行后执行，无论是否出现异常，并且在后置通知中无法访问目标方法的执行结果，要访问方法的执行结果得用返回通知@AfterReturning，返回通知是在方法返回结果之后执行，还有其他的。
- c. 可以在通知方法中声明一个类型为JoinPoint ([org.aspectj.lang.JoinPoint](#)) 的参数，然后就能访问链接细节，如方法名称和参数值
- d. 注意 List是java.util.List
- e. 注意JoinPoint是org.aspectj.lang.JoinPoint
- f. @Order(1),如果有多个切面执行同一个方法，那么可以用@Order来定义优先级，值越小，优先级越高
- g. 使用@Pointcut来声明切入点的表达式，后面直接加切入点的具体位置

```

cationContext.xml  Main.java  Calculator.java  CalculatorImpl.java  LoggingAspect.java
package com.js.aop.impl;

import java.util.ArrayList;□

//{@Order(1)} ,如果有多个切面执行同一个方法，那么可以用@Order来定义优先级，值越小，优先级越高
//把这个类声明为一个切面：1.需要把该类放入到IOC容器中
@Component
// 2.再声明为一个切面
@Aspect
public class LoggingAspect {

    /**
     * 声明切入点的表达式，以在@Before@After这些注解中表明切入点时重用
     *
     *一般的，该方法中不需要添入其他代码
     *
     *使用@Pointcut来声明切入点的表达式，后面直接加切入点的具体位置
     *
     *也可以在其他类中引用，加上本类名，如LoggingAspect.declareJoinPointExpress()其他包中引用还要加上包名
     */
    @Pointcut("execution(public int com.js.aop.impl.Calculator.*(..))")
    public void declareJoinPointExpress(){}

}

/**
 * 前置通知
 *
 * 下面是一个日志输出，那么我们想要这个日志输出在具体哪个位置执行呢？用到@Before这个注解
 *
 * 声明该方法是一个前置通知：在目标方法开始之前执行，execution()中的内容即方法的具体位置
 *
 * execution()中public是方法的修饰符，int是方法的返回值类型，.*表示Calculator这个类中的所有方法！！
 * 方法中的参数只需要类型，不需要形参
 *
 * 还能进行省略写成@Before("execution(* com.js.aop.impl.Calculator.*(int, int))") 前面的*表示任意
 *
 * @param joinPoint
 */
@Before("execution(* com.js.aop.impl.Calculator.*(int, int))")
public void beforeMethod(JoinPoint joinPoint){
    //获得方法名字
    String methodName=joinPoint.getSignature().getName();
    //获得方法的参数值 注意List是java.util.List
    List<Object> args=Arrays.asList(joinPoint.getArgs());
    //打印
    System.out.println("前置通知：The method "+methodName+" begins with"+args);
}

/**
 * 后置通知
 * 在方法执行之后，在方法返回结果之前执行
 *
 * 后置通知不能获得方法的返回结果，如果要获得方法的返回结果用返回通知@AfterReturning
 *
 */
@After("execution(* com.js.aop.impl.Calculator.*(int, int))")
public void afterMethod(JoinPoint joinPoint){
    //获得方法名字
    String methodName=joinPoint.getSignature().getName();
    //打印
    System.out.println("The method "+methodName+" ends");
}

```

```

/**
 * 返回通知
 *
 * 在方法正常结束后执行的代码，因此返回通知可以访问到方法的返回值
 * 如果方法有异常，则不会执行返回通知
 *
 * @param joinPoint
 * @param result      这个参数就是返回结果
 */
@AfterReturning(value="execution(* com.js.aop.impl.Calculator.*(int, int))",
               returning="result")
public void afterReturningMethod(JoinPoint joinPoint, Object result){
    //获得方法名字
    String methodName=joinPoint.getSignature().getName();
    //打印
    System.out.println("返回通知: The method "+methodName+" ends : "+result);
}

/**
 * 异常通知
 *
 * 在目标方法出现异常的时候会执行
 *
 * @param joinPoint
 * @param ex        这个参数就是异常，并可指定异常从类型，比如参数类型Exception换成空指针异常NullPointerException
 */
@AfterThrowing(value="execution(* com.js.aop.impl.Calculator.*(int, int))",
               throwing="ex")
public void afterThrowing(JoinPoint joinPoint, Exception ex){
    String methodName=joinPoint.getSignature().getName();
    System.out.println("异常通知: The method "+methodName+" occurs exception : "+ex);
}

/**
 * 环绕通知（不常用）
 *
 * 环绕通知需要携带ProceedingJoinPoint类型的参数
 * 环绕通知类似于动态代理的全过程，ProceedingJoinPoint类型的参数可以决定是否执行目标方法
 * 切环绕通知必须有返回值，返回值即为目标方法的返回值
 *
 * @param pjp
 */
@Around("execution(* com.js.aop.impl.Calculator.*(int, int))")
public Object aroundMethod(ProceedingJoinPoint pjp){

    Object result=null;
    String methodName=pjp.getSignature().getName();
    List<Object> args=Arrays.asList(pjp.getArgs());
    //执行目标方法
    try {
        //前置通知
        System.out.println("环绕通知中的‘前置通知’过程：方法名是"+methodName+"参数是"+args);
        result=pjp.proceed();
        //返回通知
        System.out.println("环绕通知中的‘返回通知’过程：结果是"+result);
    } catch (Throwable e) {
        //异常通知
        System.out.println("环绕通知中的‘异常通知’过程：异常是"+e);
        throw new RuntimeException(e); //这一句话不太懂
    }
    //后置通知
    System.out.println("环绕通知中的‘后置通知’过程：");

    return result;
}

```

上面这都是用注解的方式配置SpringAOP

接下来用xml配置文件的方式配置SpringAOP

首先复制上文的包到另一个包里，用的名字是com.js.aop.impl.xml，然后删掉文件中所有的注解，即带@的内容

然后新建一个xml叫applicationContext_xml.xml如下

```
SpringStudyAOP
  src
    com.js.aop
    com.js.aop.impl
    com.js.aop.impl.xml
      Calculator.java
      CalculatorImpl.java
      LoggingAspect.java
      Main.java
    applicationContext.xml.xml
```

这里面的LoggingAspect.java如下

```
applicationContext.xml.xml  Main.java  Calculator.java  LoggingAspect.java
package com.js.aop.impl.xml;

import java.util.ArrayList;

public class LoggingAspect {

    public void beforeMethod(JoinPoint joinPoint){
        //获得方法名字
        String methodName=joinPoint.getSignature().getName();
        //获得方法的参数值 注意 List是java.util.List
        List<Object> args=Arrays.asList(joinPoint.getArgs());
        //打印
        System.out.println("前置通知: THe method "+methodName+" begins with"+args);
    }

    public void afterMethod(JoinPoint joinPoint){
        //获得方法名字
        String methodName=joinPoint.getSignature().getName();
        //打印
        System.out.println("后置通知: THe method "+methodName+" ends");
    }

    public void afterReturningMethod(JoinPoint joinPoint,Object result){
        //获得方法名字
        String methodName=joinPoint.getSignature().getName();
        //打印
        System.out.println("返回通知: THe method "+methodName+" ends : "+result);
    }

    public void afterThrowing(JoinPoint joinPoint,Exception ex){
        String methodName=joinPoint.getSignature().getName();
        System.out.println("异常通知: THe method "+methodName+" occurs exception : "+ex);
    }
}
```

重点是xml的配置如下

注意分三步，先配置bean，然后配置切面的bean，最后配置AOP

```

applicationContext.xml.xml ☐ Main.java ☐ Calculator.java ☐ LoggingAspect.java
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xmlns:aop="http://www.springframework.org/schema/aop"
5   xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.spring
6       http://www.springframework.org/schema/aop http://www.springframework.org/sch
7
8
9 <!-- 配置bean -->
10 <bean id="calculator" class="com.js.aop.impl.xml.CalculatorImpl"></bean>
11
12 <!-- 配置切面的bean -->
13 <bean id="loggingAspect" class="com.js.aop.impl.xml.LoggingAspect"></bean>
14
15 <!-- 配置AOP -->
16 <aop:config>
17   <!-- 配置切点表达式 -->
18   <aop:pointcut expression="execution(* com.js.aop.impl.xml.Calculator.*(..))" id="pointcut"/>
19   <!-- 配置切面及通知 ref中是切面所在的类名 -->
20   <aop:aspect ref="loggingAspect" order="1">
21     <!-- 配置具体的通知，这个before是前置通知 -->
22     <aop:before method="beforeMethod" pointcut-ref="pointcut"/>
23     <aop:after method="afterMethod" pointcut-ref="pointcut"/>
24   </aop:aspect>
25 </aop:config>
26
applicationContext.xml.xml ☐ Main.java ☐ Calculator.java ☐ LoggingAspect.java
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xmlns:aop="http://www.springframework.org/schema/aop"
5   xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.spring
6       http://www.springframework.org/schema/aop http://www.springframework.org/sch
7
8
9 <!-- 配置bean -->
10 <bean id="calculator" class="com.js.aop.impl.xml.CalculatorImpl"></bean>
11
12 <!-- 配置切面的bean -->
13 <bean id="loggingAspect" class="com.js.aop.impl.xml.LoggingAspect"></bean>
14
15 <!-- 配置AOP -->
16 <aop:config>
17   <!-- 配置切点表达式 -->
18   <aop:pointcut expression="execution(* com.js.aop.impl.xml.Calculator.*(..))" id="pointcut"/>
19   <!-- 配置切面及通知 ref中是切面所在的类名 -->
20   <aop:aspect ref="loggingAspect" order="1">
21     <!-- 配置具体的通知，这个before是前置通知 -->
22     <aop:before method="beforeMethod" pointcut-ref="pointcut"/>
23     </aop:aspect>
24 </aop:config>
25

```

运行Main

```
applicationContext.xml.xml Main.java Calculator.java LoggingAspect.java
package com.js.aop.impl.xml;

import org.springframework.context.ApplicationContext;

public class Main {

    public static void main(String[] args) {
        //1.创建Spring的IOC容器
        ApplicationContext ctx=new ClassPathXmlApplicationContext("applicationContext.xml.xml");

        //2.从IOC容器中获取Bean的实例，注意getBean里的bean名称
        Calculator c=ctx.getBean(Calculator.class);

        //3.使用bean
        int result=c.add(3, 3);
        System.out.println("result:"+result);

        int result2=c.div(15, 3);
        System.out.println("result:"+result2);
    }
}
```

结果是OK的

```
Problems @ Javadoc Declaration Console
<terminated> Main (10) [Java Application] C:\Java\jre\bin\javaw.exe (2017年
二月 22, 2017 4:33:33 下午 org.springframework.context
信息: Refreshing org.springframework.context.support.
二月 22, 2017 4:33:33 下午 org.springframework.beans.f
信息: Loading XML bean definitions from class path
前置通知: THe method add begins with[3, 3]
后置通知: THe method add ends
result:6
前置通知: THe method div begins with[15, 3]
后置通知: THe method div ends
result:5
```

```
Problems @ Javadoc Declaration Console
<terminated> Main (10) [Java Application] C:\Java\jre\bin\javaw.exe
二月 22, 2017 4:31:05 下午 org.springframework.co
信息: Refreshing org.springframework.context.s
二月 22, 2017 4:31:05 下午 org.springframework.be
信息: Loading XML bean definitions from class
前置通知: THe method add begins with[3, 3]
result:6
前置通知: THe method div begins with[15, 3]
result:5
```