

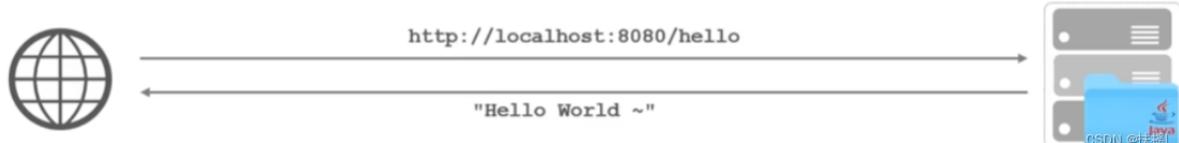
所有的 Spring 项目都基于 Spring Framework

Spring Framework 配置繁琐，入门难度大

@[TOC]

1. SpringBootWeb入门

一个简单的应用：浏览器发起 /hello 请求后，给浏览器返回一个字符串 Hello world~



步骤：

1. 创建springboot工程，并勾选web开发相关依赖。
2. 定义 `HelloController` 类，添加方法 `hello`，并添加注解。
3. 运行测试

简单的构成分析

- 一个自动生成的类，和模块名一致：启动类，用来启动springboot工程，想启动工程只需要运行这个 `main` 函数

```
1 package com.itheima;
2
3 import org.springframework.boot.SpringApplication;
4 import org.springframework.boot.autoconfigure.SpringBootApplication;
5
6 // 启动类 --- 启动springboot工程
7 @SpringBootApplication
8 public class SpringBootQuickStartApplication {
9
10    public static void main(String[] args) {
11        SpringApplication.run(SpringBootQuickStartApplication.class,
12        args);
13    }
14 }
```

- `resources/static` 和 `/templates` 暂时用不上
- `.properties` 是默认配置文件，也支持一些其他的后缀格式
- 创建的 `HelloController` 类是一个普通的Java类，需要添加注解 `@RestController` 把他变成请求处理类。还需要指定当前方法处理的是那个请求，使用注解 `@RequestMapping('/hello')` 指定处理的请求路径，浏览器将来请求 `/hello` 这个地址时，最终就会调用这个方法

```
1 package com.itheima.controller;
2
3 import org.springframework.web.bind.annotation.RequestMapping;
4 import org.springframework.web.bind.annotation.RestController;
5
6 // 请求处理类，加上注解后，这就是一个请求处理类，不加注解就是普通类
7 @RestController
8 public class HelloController {
```

```
9 // 指定当前方法处理的是哪个请求, /hello请求。浏览器请求/hello地址时, 最终就会  
10 调用这个hello方法  
11 @RequestMapping("/hello")  
12 public String hello() {  
13     System.out.println("Hello world~");  
14     return "Hello world~";  
15 }
```

工程启动后

工程如果成功启动，在浏览器输入 localhost:8080 就可以看到函数return的 Hello world，也可以在控制台看到输出的 Hello world，说明入门程序没有问题。

2. HTTP协议

超文本传输协议，规定了浏览器与服务器间数据传输的规则，实际上就是规定了数据格式

F12 查看 网络 里的 hello 请求

request header是请求的数据

response header是响应的数据

特点

1. 基于TCP协议：面向连接，安全
2. 基于请求-响应模型：一次请求对应一次响应
3. HTTP协议是**无状态协议**：对事物没有记忆力，**每次请求-响应都是独立的**
 - 缺点：多次请求间不能共享数据
 - 优点：速度快

请求协议

就是请求数据的格式，分为三个部分

1. 请求行，数据第一行， [请求方式+资源路径+协议]
2. 请求头，第二行开始，是 key: value 的格式
3. 请求体，POST请求，存放请求参数
 - GET：请求参数在请求行中，没有请求体，请求大小有限制
 - POST：请求参数在请求体中，请求大小没有限制

相应协议

同样分为三个部分

1. 响应行，数据第一行， [协议+状态码+描述]
2. 响应头，第二行开始，是 key: value 的格式
3. 响应体，最后一部分，存放响应数据

注意状态码 200、404、500

协议解析

Tomcat

3. Tomcat

最流行的HTTP协议解析程序，使程序员不必直接对协议进行操作。主要功能是“提供网上信息浏览服务”，又称Servlet容器

安装：解压安装包

卸载：删除文件夹

基本使用

`bin/startup.bat` 运行

`bin/shutdown.bat` 关闭

常见问题：

- `startup.bat` 闪退，检查环境变量是否有 `JAVA_HOME`（不带bin）
- 端口号冲突，找到对应程序将其关闭（任务管理器、详细信息、找对应端口号的程序）或修改 Tomcat使用的端口号 `conf/server.xml`

注意：HTTP协议默认端口号为80，若将Tomcat端口号改为80，则访问Tomcat时，不用输入端口号

部署项目

将项目放置到 `webapps` 下就部署完成了

4. SpringBootWeb入门程序解析

起步依赖

- `spring-boot-starter-web`：包含了web应用开发所需要的常见依赖
- `spring-boot-starter-test`：包含了单元测试所需要的常见依赖

借助依赖传递特性将某个功能所需的常见依赖聚合到一起。

在 `pom` 中会看到起步依赖没有写 `<version>`，这是因为SpringBoot项目都有一个父工程，在 `pom` 中表现为 `<parent>` 标签里的内容，所有起步依赖的版本都在**父工程中进行了统一管理，会自动引入**和 SpringBoot版本对应的起步依赖版本

可以看到 `spring-boot-starter-web` 中有一个 `tomcat` 相关的依赖，所以启动项目时会将内置的Tomcat 启动起来并占用Tomcat默认端口号8080，**这个Tomcat和外部安装的Tomcat不是同一个**，外部的 Tomcat会很少使用，多数都是使用内置的Tomcat

5. 请求响应

Tomcat不能识别我们写的如 `HelloController` 的 `controller` 程序，但Tomcat可以识别Servlet规范。SpringBoot底层提供了一个非常核心的Servlet程序 `DispatcherServlet`，它实现了Servlet规范中的 Servlet接口（可看继承体系）。

浏览器发起的请求都会先经过 `DispatcherServlet`，由 `DispatcherServlet` 将请求转给 `controller` 程序进行处理，处理完的结果返回给 `DispatcherServlet`，`DispatcherServlet` 再给浏览器响应数据

前端控制器

1 | `DispatcherServlet`

如何在Servlet中获取请求数据？浏览器发请求会携带HTTP请求数据，web服务器负责对请求协议的解析。Tomcat就会对数据进行解析，并将解析后所有请求信息封装到一个对象 `HttpServletRequest` 中，也叫请求对象

请求对象：获取请求数据

应用程序就可以从 `HttpServletRequest` 对象中获取请求数据了，然后对请求进行处理。然后Tomcat服务器需要根据响应数据的格式给浏览器响应数据。借助另一个对象设置响应数据，`HttpServletResponse`，Tomcat会根据在这个对象中设置的响应信息来响应数据给浏览器。

响应对象：设置响应数据

`HttpServletResponse`

B/S架构：浏览器/服务器架构模式。客户端只需要浏览器，应用程序和数据都存放在服务端。

C/S架构：客户端/服务器架构模式。

主要关注controller程序，最重要的是获取请求参数和设置响应数据

工具

`postman 或 apifox`

简单参数

`get post` 方式的处理方法相同

发送的是普通数据如 `name=zhangsan`，创建 `controller/RequestController` 类进行操作

- 原始方式获取请求参数：通过 `HttpServletRequest` 对象手动获取

请求地址：`http://localhost:8080/simpleParam?name=tom&age=10`

实现请求处理方法，处理对 `/simpleParam` 的请求

```
1 package com.itheima.controller;
2
3 import jakarta.servlet.http.HttpServletRequest;
4 import org.springframework.web.bind.annotation.RequestMapping;
5 import org.springframework.web.bind.annotation.RestController;
6
7 @RestController
8 public class RequestController {
9     // 原始方式获取简单参数
10    @RequestMapping("/simpleparam")
11    public String simpleParam(HttpServletRequest request) {
12        // 获取请求参数
13        String name = request.getParameter("name");
14        String ageStr = request.getParameter("age");
15
16        int age = Integer.parseInt(ageStr);
17        System.out.println(name + ":" + age);
18        return "OK";
19    }
20}
```

缺点：繁琐，需要手动类型转换

- SpringBoot方式

简单参数：参数名与形参变量名相同，即可自动接收参数

请求地址：<http://localhost:8080/simpleParam>，参数为 name=Tom 和 age=20

示例如下：

修改 RequestController 中的方法

```
1 package com.itheima.controller;
2
3 import jakarta.servlet.http.HttpServletRequest;
4 import org.springframework.web.bind.annotation.RequestMapping;
5 import org.springframework.web.bind.annotation.RestController;
6
7 @RestController
8 public class RequestController {
9     // SpringBoot方式获取参数
10    @RequestMapping("/simpleParam")
11    public String simpleParam(String name, Integer age) {
12        System.out.println(name + ":" + age);
13        return "OK";
14    }
15 }
```

如果参数名和形参名不同，默认情况下会接收不到，变量存储值为 null。可以通过注解 `@RequestParam` 完成映射

```
1 @RequestMapping("/simpleParam")
2     public String simpleParam(@RequestParam(name= "name") String username,
3                                Integer age) {
4         System.out.println(username + ":" + age);
5         return "OK";
6     }
```

注意：`@RequestParam` 中的 `required` 默认为 `true`，表示该参数必须传递，如果不传递则报错，若参数是可选的，可修改 `required` 为 `false`。

实体参数

- 简单实体对象：请求参数名与形参对象属性名相同，定义一个POJO类接收即可。

注意：定义的类中属性名必须和请求参数名相同才能成功封装

示例如下：

请求地址：<http://localhost:8080/simplePojo?name=ITCAST&age=16>

定义一个 `User` 类，其中的属性名必须和请求参数名相同，这里为 `name`、`age`

```
1 package com.itheima.pojo;
2
3 public class User {
4     public String name;
5     public Integer age;
6
7
8     public String getName() {
9         return name;
10    }
```

```

11
12     public void setName(String name) {
13         this.name = name;
14     }
15
16     public Integer getAge() {
17         return age;
18     }
19
20     public void setAge(Integer age) {
21         this.age = age;
22     }
23
24     @Override
25     public String toString() {
26         return "User{" +
27             "name='" + name + '\'' +
28             ", age=" + age +
29             '}';
30     }
31 }
```

修改 RequestController 中的方法，处理对 /simple/Pojo 的请求

```

1 package com.itheima.controller;
2
3 import com.itheima.pojo.User;
4 import org.springframework.web.bind.annotation.RequestMapping;
5 import org.springframework.web.bind.annotation.RequestParam;
6 import org.springframework.web.bind.annotation.RestController;
7
8 @RestController
9 public class RequestController {
10     @RequestMapping("/simplePojo")
11     public String simplePojo(User user) {
12         System.out.println(user);
13         return "OK";
14     }
15 }
```

- 复杂实体对象：按照对象层次关系接收嵌套 POJO 属性参数

示例如下：

请求地址：<http://localhost:8080/complexPojo?name=ITCAST&age=16&address.province=北京&address.city=北京>

写一个 Address 类

```

1 package com.itheima.pojo;
2
3 /**
4  * Created with IntelliJ IDEA.
5  *
6  * @author : wu_qing
7  * @version : 1.0
8  * @Project : LearnSpringBoot
9  * @Package : com.itheima.pojo
10 * @ClassName : .java
```

```

11 * @createTime : 2024/2/10 21:58
12 * @Email : 1553232108@qq.com
13 * @Description :
14 */
15 public class Address {
16     public String province;
17     public String city;
18
19     public String getProvince() {
20         return province;
21     }
22
23     public void setProvince(String province) {
24         this.province = province;
25     }
26
27     public String getCity() {
28         return city;
29     }
30
31     public void setCity(String city) {
32         this.city = city;
33     }
34
35     @Override
36     public String toString() {
37         return "Address{" +
38                 "province='" + province + '\'' +
39                 ", city='" + city + '\'' +
40                 '}';
41     }
42 }

```

将 `Address` 添加到 `User` 类的属性中

修改请求处理方法

```

1 package com.itheima.controller;
2
3 import com.itheima.pojo.User;
4 import org.springframework.web.bind.annotation.RequestMapping;
5 import org.springframework.web.bind.annotation.RequestParam;
6 import org.springframework.web.bind.annotation.RestController;
7
8 @RestController
9 public class RequestController {
10     @RequestMapping("/complexPojo")
11     public String complexPojo(User user) {
12         System.out.println(user);
13         return "OK";
14     }
15 }

```

- 数组集合参数/多同名参数:

数组方法: 请求参数名与数组名相同且请求参数为多个, 定义数组类型形参即可

集合方法: 使用 `@RequestParam` 绑定参数关系, 将多个请求参数的值封装到集合
示例如下:

请求地址: `http://localhost:8080/arrayParam?hobby=game&hobby=java&hobby=sing`

`http://localhost:8080/listParam?hobby=game&hobby=java&hobby=sing`

修改请求处理方法

```
1 package com.itheima.controller;
2
3 import com.itheima.pojo.User;
4 import org.springframework.web.bind.annotation.RequestMapping;
5 import org.springframework.web.bind.annotation.RequestParam;
6 import org.springframework.web.bind.annotation.RestController;
7
8 @RestController
9 public class RequestController {
10     // 数组方法
11     @RequestMapping("/arrayParam")
12     public String arrayParam(String[] hobby) {
13         System.out.println(Arrays.toString(hobby));
14         return "OK";
15     }
16
17     // 集合方法获取同名参数
18     @RequestMapping("/listParam")
19     public String listParam(@RequestParam List<String> hobby) {
20         System.out.println(hobby);
21         return "OK";
22     }
23 }
```

- 日期参数: 使用 `@DateTimeFormat` 注解完成日期参数格式转换

需要指定传来的日期参数格式, 如 `@DateTimeFormat(pattern = "yyyy-MM-dd HH:mm:ss")`

注意: 这里写的是yyyy-MM-dd HH:mm:ss的格式, 那么请求参数里也必须是4位-2位-2位 2位:2位

请求地址: `http://localhost:8080/dateParam?updateTime=2024-12-10 22:17:39`, 如这里的月份如果为1月必须写为01补全位数

```
1 package com.itheima.controller;
2
3 import com.itheima.pojo.User;
4 import org.springframework.web.bind.annotation.RequestMapping;
5 import org.springframework.web.bind.annotation.RequestParam;
6 import org.springframework.web.bind.annotation.RestController;
7
8 @RestController
9 public class RequestController {
10     // 日期参数
11     @RequestMapping("/dateParam")
12     public String dateParam(@DateTimeFormat(pattern = "yyyy-MM-dd
HH:mm:ss") LocalDateTime updateTime) {
13         System.out.println(updateTime);
14         return "OK";
15     }
16 }
```

- JSON参数：需要 POST 方式发送，要求JSON数据键名与对象属性名相同，定义POJO类型即可接收参数，需要使用 @RequestBody 标识，将JSON数据封装到实体对象中

示例如下：

请求地址：<http://localhost:8080/jsonParam>

数据：

```
1 "name": "zhangsan",
2 "age": 16,
3 "address": {
4     "province": "北京",
5     "city": "北京"
6 }
7 }
```

实现请求处理方法

```
1 package com.itheima.controller;
2
3 import com.itheima.pojo.User;
4 import org.springframework.web.bind.annotation.RequestMapping;
5 import org.springframework.web.bind.annotation.RequestParam;
6 import org.springframework.web.bind.annotation.RestController;
7
8 @RestController
9 public class RequestController {
10     // JSON参数
11     @RequestMapping("/jsonParam")
12     public String jsonParam(@RequestBody User user) {
13         System.out.println(user);
14         return "OK";
15     }
16 }
```

- 路径参数

通过请求 URL 直接传递参数，如 <http://localhost:8080/path/1>，如果注解写

@RequestMapping("/path/1")，将来 /1 变成 /2、/3、/100 那就不能再处理这个请求，所以这个参数应该是动态的。使用 {...} 来标识该路径参数，就可以写

@RequestMapping("/path/{id}") 代表这地方不是固定值而是路径参数，参数名叫 id，这样就可以在 controller 中声明一个形参叫 id，使用 @PathVariable 来指定获取路径参数并将路径参数的 id 绑定给方法参数的 id，路径参数参数名需要与方法形参参数名保持一致

示例如下：

请求地址：<http://localhost:8080/path/1>

实现请求处理方法

```
1 package com.itheima.controller;
2
3 import com.itheima.pojo.User;
4 import org.springframework.web.bind.annotation.RequestMapping;
5 import org.springframework.web.bind.annotation.RequestParam;
6 import org.springframework.web.bind.annotation.RestController;
7
8 @RestController
9 public class RequestController {
10     // 路径参数
11     @RequestMapping("/path/{id}")
12 }
```

```
12     public String pathParam(@PathVariable Integer id) {  
13         System.out.println(id);  
14         return "OK";  
15     }  
16 }
```

多路径参数：在请求路径中使用 / 分隔，再写其他的参数，请求路径也要相应改变

```
@RequestMapping("/path/{id}/{name}")
```

注意：每个参数都需要使用 @pathVariable 来绑定

示例如下：

请求地址：<http://localhost:8080/path/1/Tom>

实现请求处理方法

```
1 package com.itheima.controller;  
2  
3 @RestController  
4 public class RequestController {  
5 // 多路径参数  
6     @RequestMapping("/path/{id}/{name}")  
7     public String pathParam2(@PathVariable Integer id, @PathVariable  
String name) {  
8         System.out.println(id);  
9         System.out.println(name);  
10        return "OK";  
11    }  
12 }
```

响应数据

controller程序的 `return` 值就是响应（返回给浏览器）的值。所有的响应数据都需要依赖核心的 `@ResponseBody` 注解，需要写在controller方法上或类上。他的作用就是将方法返回值直接作为响应数据给客户端浏览器，如果返回值类型是实体对象/集合，会转为JSON格式再响应，但是似乎从来没见过？事实上是因为 `@RestController = @Controller + @ResponseBody`，已经包含了 `@ResponseBody` 注解。写 `@RestController` 等价于写 `@Controller + @ResponseBody`。在类上加了 `@ResponseBody` 注解，代表当前类下所有返回值都会作为响应数据，如果是对象或集合会先转JSON再来响应

示例如下：

请求地址分别为：

```
1 http://localhost:8080/hello  
2 http://localhost:8080/getAddr  
3 http://localhost:8080/listAddr
```

新建一个 `ResponseController` 类

```
1 package com.itheima.controller;  
2  
3 import com.itheima.pojo.Address;  
4 import org.springframework.web.bind.annotation.RequestMapping;  
5 import org.springframework.web.bind.annotation.RestController;  
6  
7 import java.util.ArrayList;  
8 import java.util.List;
```

```

9
10 @RestController
11 public class ResponseController {
12
13     @RequestMapping("/hello")
14     public String hello() {
15         System.out.println("Hello world!");
16         return "Hello world~";
17     }
18
19     @RequestMapping("/getAddr")
20     public Address getAddr() {
21         Address addr = new Address();
22         addr.setProvince("广东");
23         addr.setCity("深圳");
24         return addr;
25     }
26
27     @RequestMapping("/listAddr")
28     public List<Address> listAddr() {
29         List<Address> list = new ArrayList<>();
30
31         Address addr = new Address();
32         addr.setProvince("北京");
33         addr.setCity("北京");
34
35         Address addr2 = new Address();
36         addr2.setProvince("广东");
37         addr2.setCity("深圳");
38
39         list.add(addr);
40         list.add(addr2);
41         return list;
42     }
43 }

```

对这三个请求处理方法分别发出请求，可以看到第一个返回的就是一个字符串，第二个返回的是JSON格式的数据，第三个返回的是JSON数组的数据

说明：每个对外暴露的方法都称为功能接口，注解中写的路径为接口的访问路径。开发文档就是描述功能接口的请求路径，请求参数以及响应数据的。

可以发现每个接口响应的数据很随意，没有任何规范，前端很难解析响应回去的数据，开发成本会增加，项目不便管理且很难维护。

一般会给所有的功能接口设置统一的响应结果，可以考虑使用一个对象 `result` 来接收，主要有3个属性：

1. int code, 响应码，可以和前端约定：1表示成功，0表示失败
2. string msg, 提示信息
3. object data, 返回的数据

返回的`result`对象经过 `ResponseBody` 的处理后，就会响应一个JSON格式的数据，前端只会收到一种格式的数据，只需要根据这一种格式来解析。项目管理和维护就会更加方便。

示例如下：

实现一个 `Result` 类

```

1 package com.itheima.pojo;
2

```

```
3  public class Result {  
4      private Integer code; // 验证码, 1成功, 0失败  
5      private String msg; // 提示信息  
6      private Object data; // 数据  
7  
8      public Result() {  
9      }  
10  
11     public Result(Integer code, String msg, Object data) {  
12         this.code = code;  
13         this.msg = msg;  
14         this.data = data;  
15     }  
16  
17     public Integer getCode() {  
18         return code;  
19     }  
20  
21     public void setCode(Integer code) {  
22         this.code = code;  
23     }  
24  
25     public String getMsg() {  
26         return msg;  
27     }  
28  
29     public void setMsg(String msg) {  
30         this.msg = msg;  
31     }  
32  
33     public Object getData() {  
34         return data;  
35     }  
36  
37     public void setData(Object data) {  
38         this.data = data;  
39     }  
40  
41     public static Result success(Object data) {  
42         return new Result(1, "success", data);  
43     }  
44  
45     public static Result success() {  
46         return new Result(1, "success", null);  
47     }  
48  
49     public static Result error(String msg) {  
50         return new Result(0, msg, null);  
51     }  
52  
53     @Override  
54     public String toString() {  
55         return "Result{" +  
56                 "code=" + code +  
57                 ", msg='" + msg + '\'' +  
58                 ", data=" + data +  
59                 '}';  
60     }  
}
```

类中写了三个静态方法，分别是成功且返回数据，成功且不返回数据，失败
再修改 ResponseController 类中的方法，使三个功能接口具有相同的响应数据格式

```

1 package com.itheima.controller;
2
3 import com.itheima.pojo.Address;
4 import com.itheima.pojo.Result;
5 import org.springframework.web.bind.annotation.RequestMapping;
6 import org.springframework.web.bind.annotation.RestController;
7
8 import java.util.ArrayList;
9 import java.util.List;
10
11 @RestController
12 public class ResponseController {
13
14     /*@RequestMapping("/hello")
15     public String hello() {
16         System.out.println("Hello world!");
17         return "Hello world~";
18     }
19
20     @RequestMapping("/getAddr")
21     public Address getAddr() {
22         Address addr = new Address();
23         addr.setProvince("广东");
24         addr.setCity("深圳");
25         return addr;
26     }
27
28     @RequestMapping("/listAddr")
29     public List<Address> listAddr() {
30         List<Address> list = new ArrayList<>();
31
32         Address addr = new Address();
33         addr.setProvince("北京");
34         addr.setCity("北京");
35
36         Address addr2 = new Address();
37         addr2.setProvince("广东");
38         addr2.setCity("深圳");
39
40         list.add(addr);
41         list.add(addr2);
42         return list;
43     }*/
44
45     @RequestMapping("/hello")
46     public Result hello() {
47         System.out.println("Hello world!");
48         // return new Result(1, "success", "Hello world!");
49         return Result.success("Hello World!"); // 和上边的效果一样，一个用构造,
一个用静态方法，都是返回一个result对象
50     }
51

```

```

52     @RequestMapping("/getAddr")
53     public Result getAddr() {
54         Address addr = new Address();
55         addr.setProvince("广东");
56         addr.setCity("深圳");
57         return Result.success(addr);
58     }
59
60     @RequestMapping("/listAddr")
61     public Result listAddr() {
62         List<Address> list = new ArrayList<>();
63
64         Address addr = new Address();
65         addr.setProvince("北京");
66         addr.setCity("北京");
67
68         Address addr2 = new Address();
69         addr2.setProvince("广东");
70         addr2.setCity("深圳");
71
72         list.add(addr);
73         list.add(addr2);
74         return Result.success(list);
75     }
76 }
```

案例

- 在POM中引入dom4j，用来解析XML文件

```

1 <!-- 解析XML -->
2 <dependency>
3     <groupId>org.dom4j</groupId>
4     <artifactId>dom4j</artifactId>
5     <version>2.1.3</version>
6 </dependency>
```

- 引入工具类XMLParserUtils，实体类Emp，XML文件emp.xml
- 引入静态页面，放在resources/static目录下
- 写controller程序

```

1 package com.itheima.controller;
2
3 import com.itheima.pojo.Emp;
4 import com.itheima.pojo.Result;
5 import com.itheima.utils.XmlParserUtils;
6 import org.springframework.web.bind.annotation.RequestMapping;
7 import org.springframework.web.bind.annotation.RestController;
8
9 import java.util.List;
10
11 @RestController
12 public class EmpController {
13
14     @RequestMapping("/listEmp") // 和前端页面发送请求的路径相同
15     public Result list() {
```

```

16     // 1. 加载解析XML文件
17     String file =
18         this.getClass().getClassLoader().getResource("emp.xml").getFile();
19         System.out.println(file);
20         List<Emp> empList = XmlParserUtils.parse(file, Emp.class);
21
22         // 2. 对数据进行转换处理 gender和job
23         empList.stream().forEach(emp -> {
24             // 处理gender 1男 2女
25             String gender = emp.getGender();
26             if ("1".equals(gender)) {
27                 emp.setGender("男");
28             } else if ("2".equals(gender)) {
29                 emp.setGender("女");
30             }
31
32             // 处理job 1: 讲师, 2: 班主任 , 3: 就业指导
33             String job = emp.getJob();
34             if ("1".equals(job)) {
35                 emp.setJob("讲师");
36             } else if ("2".equals(job)) {
37                 emp.setJob("班主任");
38             } else if ("3".equals(job)) {
39                 emp.setJob("就业指导");
40             }
41         });
42
43         // 3. 响应数据
44         return Result.success(empList);
45     }
46 }
```

说明：SpringBoot项目的静态资源默认存放目录为：classpath:/static、classpath:/public、classpath:/resources，classpath是类路径，对Maven项目来说resources目录就是类路径，一般就使用static目录

6. 分层解耦

EmpController中的代码包含数据访问，逻辑处理，接受请求和响应数据。需要尽量让每个类接口方法的职责更加单一，这是**单一职责原则**。能够使类接口方法的可读性 可维护性 拓展性更好

三层架构

- Controller：表示层/控制层，就是编写的Controller程序，负责接收请求、进行处理、响应数据。
- Service：逻辑层，处理具体的业务逻辑。
- Dao：数据访问层（Data Access Object）/持久层，负责数据访问操作，包括增删改查。

前端发起请求先到达Controller，调用Service进行逻辑处理，处理的前提是拿到数据，所以再调用Dao层去操作文件中的数据，拿到数据再返回给Service，处理之后的结果返回给Controller，再响应数据给前端。

数据访问Dao层的实现方式可能有很多，如访问文件的数据、数据库的数据、别人提供接口的数据，要灵活的切换各种实现，可以通过面向接口的方式进行编程

需要先定一个Dao的接口来增强灵活性和拓展性

- 先来一个dao包下的EmpDao接口

```

1 package com.itheima.dao;
2
3
4 import com.itheima.pojo.Emp;
5
6 import java.util.List;
7
8 public interface EmpDao {
9     // 获取员工列表
10    public List<Emp> listEmp();
11 }
12

```

- 再来一个impl/EmpDaoA (方式很多所以叫A来区分) 实现类来实现接口的方法，逻辑就是Controller中解析XML文件获取数据部分

```

1 package com.itheima.dao.impl;
2
3
4 import com.itheima.dao.EmpDao;
5 import com.itheima.pojo.Emp;
6 import com.itheima.utils.XmlParserUtils;
7
8 import java.util.List;
9
10 public class EmpDaoA implements EmpDao {
11     @Override
12     public List<Emp> listEmp() {
13         // 1. 加载解析XML文件
14         String file =
15             this.getClass().getClassLoader().getResource("emp.xml").getFile();
16         System.out.println(file);
17         List<Emp> empList = XmlParserUtils.parse(file, Emp.class);
18
19         return empList;
20     }
21 }

```

- 然后是Service层，先来一个接口增加灵活性EmpService

```

1 package com.itheima.service;
2
3
4 import com.itheima.pojo.Emp;
5
6 import java.util.List;
7
8 public interface EmpService {
9     // 获取员工列表
10    public List<Emp> listEmp();
11 }
12

```

- 再来impl/EmpServiceA实现类来实现接口方法，逻辑就是Controller中数据处理部分，但是要从 Dao中获取数据

```

1 package com.itheima.service.impl;
2
3 import com.itheima.dao.EmpDao;
4 import com.itheima.dao.impl.EmpDaoA;
5 import com.itheima.pojo.Emp;
6 import com.itheima.service.EmpService;
7
8 import java.util.List;
9
10 public class EmpServiceA implements EmpService {
11
12     private EmpDao empDao = new EmpDaoA();
13
14     @Override
15     public List<Emp> listEmp() {
16         // 1. 调用Dao获取数据
17         List<Emp> empList = empDao.listEmp();
18
19         // 2. 对数据进行转换处理 gender和job
20         empList.stream().forEach(emp -> {
21             // 处理gender 1男 2女
22             String gender = emp.getGender();
23             if ("1".equals(gender)) {
24                 emp.setGender("男");
25             } else if ("2".equals(gender)) {
26                 emp.setGender("女");
27             }
28
29             // 处理job 1: 讲师, 2: 班主任 , 3: 就业指导
30             String job = emp.getJob();
31             if ("1".equals(job)) {
32                 emp.setJob("讲师");
33             } else if ("2".equals(job)) {
34                 emp.setJob("班主任");
35             } else if ("3".equals(job)) {
36                 emp.setJob("就业指导");
37             }
38         });
39
40         return empList;
41     }
42 }
```

- 最后修改Controller程序，从Service中接收处理完的数据并响应给前端

```

1 package com.itheima.controller;
2
3 import com.itheima.pojo.Emp;
4 import com.itheima.pojo.Result;
5 import com.itheima.service.EmpService;
6 import com.itheima.service.impl.EmpServiceA;
7 import com.itheima.utils.XmlParserUtils;
8 import org.springframework.web.bind.annotation.RequestMapping;
```

```

9 import org.springframework.web.bind.annotation.RestController;
10
11 import java.util.List;
12
13 @RestController
14 public class EmpController {
15     private EmpService service = new EmpServiceA();
16
17     @RequestMapping("/listEmp") // 和前端页面发送请求的路径相同
18     public Result list() {
19         // 调用Service获取数据，然后响应
20         List<Emp> empList = service.listEmp();
21         // 3. 响应数据
22         return Result.success(empList);
23     }
24
25     // @RequestMapping("/listEmp") // 和前端页面发送请求的路径相同
26     // public Result list() {
27     //     // 1. 加载解析XML文件
28     //     String file =
29     //         this.getClass().getClassLoader().getResource("emp.xml").getFile();
30     //     System.out.println(file);
31     //     List<Emp> empList = XmlParserUtils.parse(file, Emp.class);
32     //
33     //     // 2. 对数据进行转换处理 gender和job
34     //     empList.stream().forEach(emp -> {
35     //         // 处理gender 1男 2女
36     //         String gender = emp.getGender();
37     //         if ("1".equals(gender)) {
38     //             emp.setGender("男");
39     //         } else if ("2".equals(gender)) {
40     //             emp.setGender("女");
41     //         }
42     //         // 处理job 1: 讲师, 2: 班主任 , 3: 就业指导
43     //         String job = emp.getJob();
44     //         if ("1".equals(job)) {
45     //             emp.setJob("讲师");
46     //         } else if ("2".equals(job)) {
47     //             emp.setJob("班主任");
48     //         } else if ("3".equals(job)) {
49     //             emp.setJob("就业指导");
50     //         }
51     //     });
52     //
53     //     // 3. 响应数据
54     //     return Result.success(empList);
55     // }
56 }

```

整体过程：前端发起请求之后，先到达Controller程序，他只负责接受请求响应数据，所以直接调用Service层中的方法，Service层只负责逻辑处理，所以直接调用Dao中的方法获取数据，由Dao层来负责数据访问操作，将查询的数据返回给Service，Service处理完后返回给Controller，Controller拿到结果再响应给前端页面。

IOC&DI引入

- 内聚：各个功能模块内部的功能联系

员工管理的Service中只会编写与员工相关的逻辑处理，与员工无关的逻辑处理不会放在这个类中。

- 耦合：衡量软件中各个层/模块之间的依赖、关联的程度

Controller中new了一个Service的实现类，**如果Service层类名发生变化，Controller的代码也需要修改**。Service与Dao也有这样的耦合关系。

- 软件设计原则：高内聚低耦合

解耦：不能直接new Service对象

提供一个容器来存储一些对象，如果想用EmpServiceA，只需要将其创建的对象A放在容器当中。

Controller在运行时需要依赖于EmpService，就可以去容器中查找EmpService这个类型的对象，看到A就是这个类型，就可以从容器中找到对象然后将其赋值给Controller中的empService。

如果要切换实现类，从A切换为B，只需要将B创建的对象放到容器中，Controller运行时也只需要在容器中查找对象，找到对象后赋值给empService。这样即使Service发生变化，Controller代码也不需要修改。

两个问题：

- 对象怎么交给容器管理
- 容器怎么为程序提供它所依赖的资源

涉及到Spring中两个概念：

- 控制反转IoC (Inversion Of Control)，Spring框架第一大核心，对象创建的控制权由应用程序转移到了外部容器。反转前由程序自身控制对象创建，反转后由容器控制。容器也叫IoC容器或Spring容器
- 依赖注入DI (Dependency Injection)，容器为程序提供运行时依赖的资源，如Controller运行时依赖EmpService，就可以让容器为它提供。
- bean对象，IoC容器中创建管理的对象，称之为bean对象

IoC-DI入门

解耦Controller与Service，Service与Dao

- 将Service层及Dao层的实现类，交给IoC容器管理，为类加注解 @Component
- 为Controller和Service注入运行时依赖的对象，为属性加注解 @Autowired

```

1 package com.itheima.dao.impl;
2
3
4 import com.itheima.dao.EmpDao;
5 import com.itheima.pojo.Emp;
6 import com.itheima.utils.XmlParserUtils;
7 import org.springframework.stereotype.Component;
8
9 import java.util.List;
10
11 @Component // 将当前类交给IoC容器管理，成为IoC容器中的bean
12 public class EmpDaoA implements EmpDao {
13     @Override
14     public List<Emp> listEmp() {
15         // 1. 加载解析XML文件
16         String file =
this.getClass().getClassLoader().getResource("emp.xml").getFile();

```

```
17     System.out.println(file);
18     List<Emp> empList = xmlParserUtils.parse(file, Emp.class);
19
20     return empList;
21 }
22 }
23 }
```

```
1 package com.itheima.service.impl;
2
3 import com.itheima.dao.EmpDao;
4 import com.itheima.dao.impl.EmpDaoA;
5 import com.itheima.pojo.Emp;
6 import com.itheima.service.EmpService;
7 import org.springframework.beans.factory.annotation.Autowired;
8 import org.springframework.stereotype.Component;
9
10 import java.util.List;
11
12 @Component
13 public class EmpServiceA implements EmpService {
14     @Autowired
15     private EmpDao empDao;
16
17     @Override
18     public List<Emp> listEmp() {
19         // 1. 调用Dao获取数据
20         List<Emp> empList = empDao.listEmp();
21
22         // 2. 对数据进行转换处理 gender和job
23         empList.stream().forEach(emp -> {
24             // 处理gender 1男 2女
25             String gender = emp.getGender();
26             if ("1".equals(gender)) {
27                 emp.setGender("男");
28             } else if ("2".equals(gender)) {
29                 emp.setGender("女");
30             }
31
32             // 处理job 1: 讲师, 2: 班主任 , 3: 就业指导
33             String job = emp.getJob();
34             if ("1".equals(job)) {
35                 emp.setJob("讲师");
36             } else if ("2".equals(job)) {
37                 emp.setJob("班主任");
38             } else if ("3".equals(job)) {
39                 emp.setJob("就业指导");
40             }
41         });
42
43         return empList;
44     }
45 }
46 }
```

```
1 package com.itheima.controller;
```

```

2
3 import com.itheima.pojo.Emp;
4 import com.itheima.pojo.Result;
5 import com.itheima.service.EmpService;
6 import com.itheima.service.impl.EmpServiceA;
7 import com.itheima.utils.XmlParserUtils;
8 import org.springframework.beans.factory.annotation.Autowired;
9 import org.springframework.context.annotation.Bean;
10 import org.springframework.web.bind.annotation.RequestMapping;
11 import org.springframework.web.bind.annotation.RestController;
12
13 import java.util.List;
14
15
16 @RestController
17 public class EmpController {
18
19     @Autowired // 运行时，IoC容器会提供该类型的bean对象，并赋值给该变量
20     private EmpService service;
21
22     @RequestMapping("/listEmp") // 和前端页面发送请求的路径相同
23     public Result list() {
24         // 调用Service获取数据，然后响应
25         List<Emp> empList = service.listEmp();
26         // 3. 响应数据
27         return Result.success(empList);
28     }
29 }

```

这样就完成了控制反转和依赖注入，也完成了层与层之间的解耦，若要切换到EmpServiceB，只需要将A的@Component注释掉，给B添加上@Component注解和自动注入，Dao层和Controller层的代码都不需要改动。

IoC详解

除了@Component外，还提供了三个衍生注解@Controller @Service @Repository来表示bean对象到底归属于哪一层。实际上由于与Mybatis整合`@Repository用的较少。

某一个类不能归到这三层，又想交给IoC容器管理，就可以使用@Component注解，典型的就是一些工具类。

另外，Controller程序也不需要使用@Controller注解，因为@RestController注解已经包括了@Controller注解。

bean对象的默认名字为类名首字母小写，可以手动指定名字@Repository(value = "daoA")，value =可以省略写作@Repository("daoA")，一般不用指定。

注解声明的bean不一定会生效

组件扫描

四个注解要想生效，还需要被组件扫描注解@ComponentScan扫描。@ComponentScan虽然没有显式配置，但实际上已经包含在了启动类中的注解@SpringBootApplication中，默认扫描方位是启动类所在包及其子包。

可以在启动类上手动设置@ComponentScan指定扫描哪个包，但这样会覆盖掉默认的扫描操作（不推荐）

推荐的是按照SpringBoot规范将所写代码放在启动类所在包及其子包下。

DI详解

`@Autowired`默认是按照类型进行注入，如果有多个相同类型的bean，就会报错

解决方案：`@Primary`，设置bean的优先级，想让哪个bean生效，在类上加上`@Primary`

`@Qualifier`，为变量指定注入的是哪个bean，使用`@Autowired + @Qualifier(bean名)`

```
1  @RestController
2  public class EmpController {
3
4      @Qualifier("empServiceA")
5      @Autowired // 运行时，IoC容器会提供该类型的bean对象，并赋值给该变量
6      private EmpService service;
7
8      @RequestMapping("/listEmp") // 和前端页面发送请求的路径相同
9      public Result list() {
10         // 调用Service获取数据，然后响应
11         List<Emp> empList = service.listEmp();
12         // 3. 响应数据
13         return Result.success(empList);
14     }
15 }
```

`@Resource`，不再使用`autowired`，使用`resource`指定注入的是哪个bean，指定bean的名字，

`@Resource(name = bean名)`

```
1  @RestController
2  public class EmpController {
3
4      // @Qualifier("empServiceA")
5      // @Autowired // 运行时，IoC容器会提供该类型的bean对象，并赋值给该变量
6      // private EmpService service;
7
8      @Resource(name = "empServiceB")
9      private EmpService service;
10
11     @RequestMapping("/listEmp") // 和前端页面发送请求的路径相同
12     public Result list() {
13         // 调用Service获取数据，然后响应
14         List<Emp> empList = service.listEmp();
15         // 3. 响应数据
16         return Result.success(empList);
17     }
18 }
```

`@Resource`与`@Autowired`的区别

1. `Autowired`是Spring框架提供的注解，`Resource`是JDK提供的注解
2. `Autowired`默认按照类型注入，`Resource`默认按照名称注入

7. SQL

语句分类

1. DDL, 数据定义语言
2. DML, 数据操作语言
3. DQL, 数据查询语言
4. DCL, 数据控制语言

DDL

`show databases;`, 查询所有数据库

`select database();`, 查询当前数据库

`create database [if not exists] 数据库名;`, 新建数据库

`use 数据库名;`, 使用数据库

`drop database [if exists] 数据库名;`, 删除数据库

`database` 也可以替换成 `schema` (了解)

事务

一组操作的集合，是不可分割的工作单位。事务会把所有的操作作为一个整体提交给系统，这些操作要么同时成功，要么同时失败。

开启事务：`start transaction;/begin;`

提交事务：`commit`

回滚事务：`rollback`

四大特性

1. 原子性，事务是不可分割的最小单元，要么全部成功，要么全部失败
2. 一致性，事务完成时，必须使所有数据保持一致状态
3. 隔离性，隔离机制，保证事务不受外部并发操作影响的独立环境下运行
4. 持久性，事务一旦提交或回滚，对数据库中的数据的改变就是永久的

索引

帮助数据库搞笑获取数据的数据结构。可以优化查询速度，几百倍几千倍的提升

无索引的情况下执行 `select` 会执行全表扫描，速度非常慢。

索引是一种数据结构，比如二叉搜索树，可以减半查询量。

索引在增加查询效率的同时，降低了增删改的效率，因为要不断维护数据结构

创建索引：`create index 索引名 on 表名(字段)`

索引结构

MySQL支持的索引类型较多，若没有特别指明，都是指默认的B+树索引

B+树：多路平衡搜索树，并将叶节点用链表链起来

索引操作语法

- 创建索引：`create [unique] index 索引名 on 表名(字段名...);`

- 查看索引: `show index from 表名;`
- 删除索引: `drop index 索引名 on 表名;`

注意:

1. 主键字段在建表时会自动创建主键索引
2. 添加唯一约束时, 数据库实际上会添加唯一索引

8. Mybatis

持久层 (dao层) 框架, 用于简化JDBC开发

快速入门

使用Mybatis查询所有用户数据

1. 准备工作, 创建SpringBoot工程, 数据库表user, 实体类User

建议让表中的字段名和类中属性名一致, 框架可以完成自动封装

2. 引入Mybatis相关依赖, 配置Mybatis

`MyBatis framework 和 MySQL Driver`

数据库连接四要素: 驱动类名、url、username、password, 需要配置在SpringBoot默认的配置文件 `application.properties` 中, 形式为键=值

3. 编写SQL语句 (注解/XML)

入门阶段推荐使用注解, 需要定义持久层结构 `UserMapper` 并加上 `@Mapper` 注解表示当前就是 MyBatis中的一个持久层接口, 也可以叫 `UserDao`, 不过在MyBatis规范中一般都叫 `mapper`, 定义一个方法加上注解 `@select(SQL语句)` 指定当前是一个查询操作。最终要想执行SQL语句, 只需要调用这个方法, 框架会自动执行SQL语句, 并自动将结果封装到返回值中。

另外要注意, 只需要定义 `mapper` 接口就行, 不需要定义实现类。框架底层会自动生成这个接口的实现类对象。

- 先创建了一个模块 `springBootMyBatisQuickstart`
- 再创建 `pojo/User` 实体类, 其中的属性名和数据库中的字段名相同

```
1 package com.itheima.pojo;
2
3 public class User {
4     private Integer id;
5     private String name;
6     private Short age;
7     private Short gender;
8     private String phone;
9
10    public User() {
11    }
12
13    public User(Integer id, String name, Short age, Short gender, String
14        phone) {
15        this.id = id;
16        this.name = name;
17        this.age = age;
18        this.gender = gender;
19        this.phone = phone;
20    }
21}
```

```

20
21     public Integer getId() {
22         return id;
23     }
24
25     public void setId(Integer id) {
26         this.id = id;
27     }
28
29     public String getName() {
30         return name;
31     }
32
33     public void setName(String name) {
34         this.name = name;
35     }
36
37     public Short getAge() {
38         return age;
39     }
40
41     public void setAge(Short age) {
42         this.age = age;
43     }
44
45     public Short getGender() {
46         return gender;
47     }
48
49     public void setGender(Short gender) {
50         this.gender = gender;
51     }
52
53     public String getPhone() {
54         return phone;
55     }
56
57     public void setPhone(String phone) {
58         this.phone = phone;
59     }
60
61     @Override
62     public String toString() {
63         return "User{" +
64             "id=" + id +
65             ", name='" + name + '\'' +
66             ", age=" + age +
67             ", gender=" + gender +
68             ", phone='" + phone + '\'' +
69             '}';
70     }
71 }
```

- 再创建 `mapper.UserMapper` 持久层接口，给类添加注解 `@Mapper`，框架会自动生成接口的实现类对象（实际就是一个代理对象）并放到IoC容器中，给方法添加注解 `@Select(select * from user)` 表示是查询方法，查询的是所有用户信息

```
1 package com.itheima.mapper;
2
3 import com.itheima.pojo.User;
4 import org.apache.ibatis.annotations.Mapper;
5 import org.apache.ibatis.annotations.Select;
6
7 import java.util.List;
8
9 @Mapper // 在运行时自动生成该接口的实现类对象（代理对象），并将该对象交给IoC容器管理
10 public interface UserMapper {
11
12     // 查询全部用户信息
13     @Select("select * from user")
14     public List<User> list();
15 }
```

- 在配置文件 `application.properties` 中添加数据库连接需要的信息

```
1 # 配置数据库连接信息
2 spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
3 spring.datasource.url=jdbc:mysql://localhost:3306/mybatis
4 spring.datasource.username=root
5 spring.datasource.password=L=104-02
```

单元测试

在 `test/java/com.itheima/SpringBootMyBatisQuickStart` 中创建一个测试函数来测试 mapper 类，但 mapper 类是一个接口，不能通过 new 来创建一个接口对象，但 `@Mapper` 注解会自动生成实现类对象并放入 IoC 容器中，我们就可以通过依赖注入来获取接口

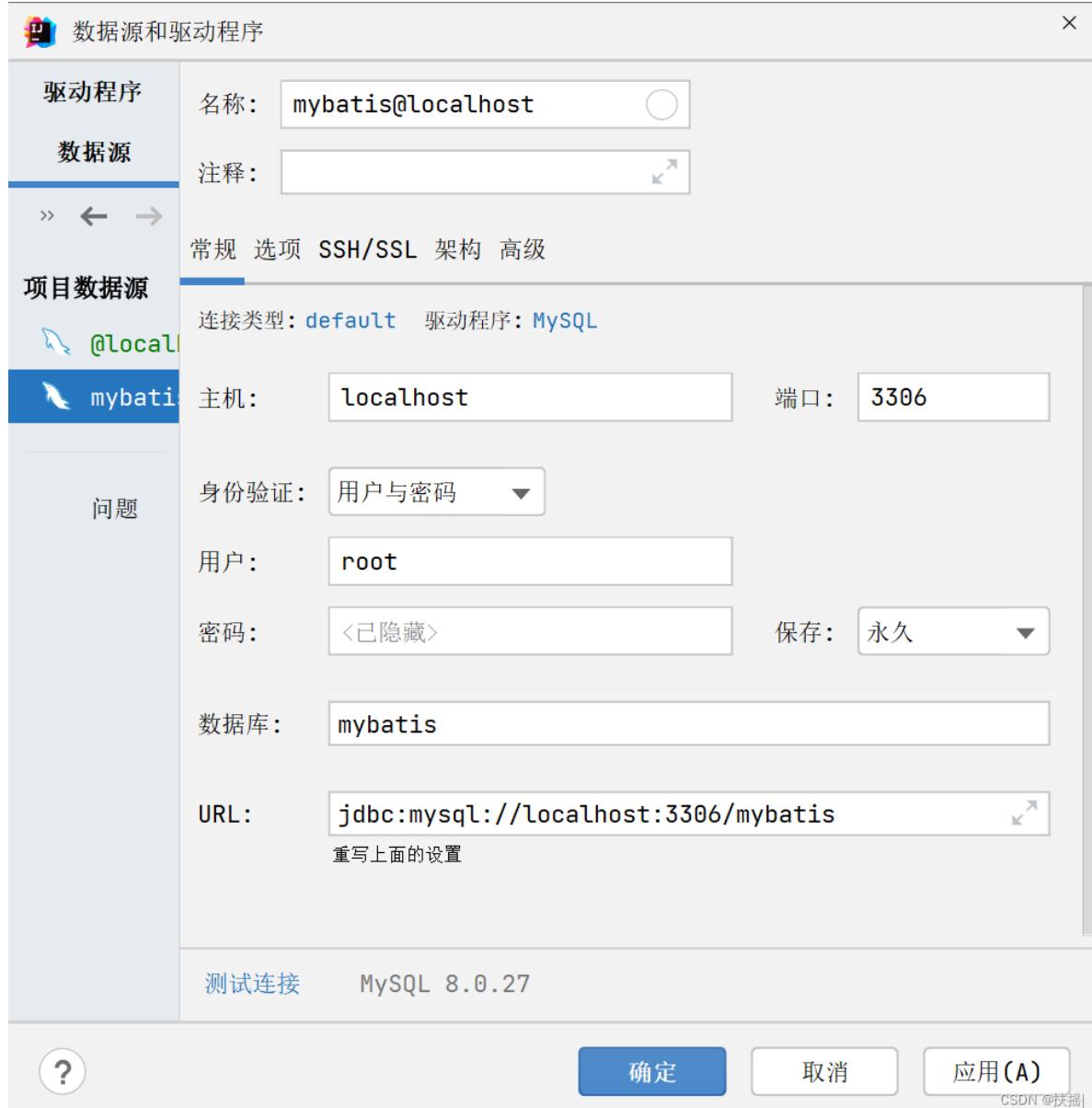
```
1 package com.itheima;
2
3 import com.itheima.mapper.UserMapper;
4 import com.itheima.pojo.User;
5 import org.junit.jupiter.api.Test;
6 import org.springframework.beans.factory.annotation.Autowired;
7 import org.springframework.boot.test.context.SpringBootTest;
8
9 import java.util.List;
10
11 @SpringBootTest // SpringBoot整合单元测试的注解
12 class SpringBootMyBatisQuickStartApplicationTests {
13
14     @Autowired
15     private UserMapper userMapper;
16
17     @Test
18     public void testListUser() {
19         List<User> userList = userMapper.list();
20         for (int i = 0; i < userList.size(); i++) {
21             System.out.println(userList.get(i));
22         }
23     }
24 }
```

配置注解SQL提示 (23版IDEA自带，无需配置)

在注解的括号内右键或按 `alt + enter`，选择注入语言或引用，选择MySQL。

此时可以提示MySQL的语句，但不会提示表名，需要在IDEA中配置MySQL数据库连接

在数据库侧边栏新建一个数据库连接，指定要连接的数据库为 mybatis



JDBC介绍

使用Java操作关系型数据库的一套API，代码不重要

1. 注册驱动
2. 获取连接对象
3. 获取执行SQL对象的statement，执行SQL，返回结果
4. 封装结果数据

缺点：硬编码，封装繁琐，反复连接与释放使性能降低

Mybatis中配置信息时所有前缀都是 `spring.datasource`，SpringBoot底层自动使用数据库连接池技术统一管理和分配连接，每次执行SQL只需要从连接池获取一个连接然后执行SQL，执行完毕后将连接归还给连接池，做到连接复用，避免反复连接的资源浪费

主要关注配置文件中Mybatis的配置和mapper接口

数据库连接池

是个容器，负责分配、管理数据库连接（Connection），允许应用重复使用一个现有的连接，而不是重新建立新连接。

怎样实现一个数据库连接池？

官方提供的数据库连接池接口DataSource，由第三方组织实现此接口。

常见产品：C3P0 DBCP Druid Hikari（SpringBoot默认自带的连接池）

- 切换SpringBoot的连接池

1. 引入Druid的依赖

2. 在配置文件中配置连接信息，就是连接四要素。之前已经配置过了，只需要导入依赖就可以
也可以在 `spring.datasource` 后全部加上一层 `.druid` 来说明选择使用的是德鲁伊连接池

注：SpringBoot3.0以上有些问题，可以在配置文件中写

```
spring.datasource.type=com.alibaba.druid.pool.DruidDataSource
```

Lombok工具包

通过注解来自动生成构造函数、get、set方法，可以自动化生成日志变量，提高开发效率

- `@Getter/@Setter`，为所有属性提供get/set方法
- `@ToString`，自动生成易阅读的toString方法
- `@EqualsAndHashCode`，自动重写equals方法和hashCode方法
- `@Data`，前面几种的总和，`@Getter + @Setter + @ToString + @EqualsAndHashCode`
- `@NoArgsConstructor`，生成无参构造器
- `@AllArgsConstructor`，生成除static字段的全参构造

需要引入Lombok的依赖，不需要指定版本，父工程中已经集成了Lombok进行了统一管理

注意：想要使用Lombok，IDEA需要安装Lombok插件（一般都自带）

```
1 package com.itheima.pojo;
2
3
4 import lombok.*;
5
6 //{@Getter
7 //{@Setter
8 //{@ToString
9 //{@EqualsAndHashCode
10
11 @Data
12 @NoArgsConstructor
13 @AllArgsConstructor
14 public class User {
15     private Integer id;
16     private String name;
17     private Short age;
18     private Short gender;
19     private String phone;
20
21     //    public User() {
22     //    }
23     //
```

```
24 //     public User(Integer id, String name, Short age, short gender, String
25 //                     phone) {
26 //         this.id = id;
27 //         this.name = name;
28 //         this.age = age;
29 //         this.gender = gender;
30 //         this.phone = phone;
31 //
32 //     public Integer getId() {
33 //         return id;
34 //     }
35 //
36 //     public void setId(Integer id) {
37 //         this.id = id;
38 //     }
39 //
40 //     public String getName() {
41 //         return name;
42 //     }
43 //
44 //     public void setName(String name) {
45 //         this.name = name;
46 //     }
47 //
48 //     public short getAge() {
49 //         return age;
50 //     }
51 //
52 //     public void setAge(Short age) {
53 //         this.age = age;
54 //     }
55 //
56 //     public short getGender() {
57 //         return gender;
58 //     }
59 //
60 //     public void setGender(short gender) {
61 //         this.gender = gender;
62 //     }
63 //
64 //     public String getPhone() {
65 //         return phone;
66 //     }
67 //
68 //     public void setPhone(String phone) {
69 //         this.phone = phone;
70 //     }
71 //
72 //     @Override
73 //     public String toString() {
74 //         return "User{" +
75 //                 "id=" + id +
76 //                 ", name='" + name + '\'' +
77 //                 ", age=" + age +
78 //                 ", gender=" + gender +
79 //                 ", phone=''" + phone + '\'' +
80 //                 '}';
}
```

```
81 // }  
82 }
```

MyBatis基本操作案例，准备工作

- 数据库表emp

```
1 -- 部门管理  
2 create table dept(  
3     id int unsigned primary key auto_increment comment  
'主键ID',  
4     name varchar(10) not null unique comment '部门名称',  
5     create_time datetime not null comment '创建时间',  
6     update_time datetime not null comment '修改时间'  
7 ) comment '部门表';  
8  
9 insert into dept (id, name, create_time, update_time) values(1, '学工  
部', now(), now()), (2, '教研部', now(), now()), (3, '咨询部', now(), now()), (4, '就  
业部', now(), now()), (5, '人事部', now(), now());  
10  
11  
12  
13 -- 员工管理  
14 create table emp (  
15     id int unsigned primary key auto_increment comment  
'ID',  
16     username varchar(20) not null unique comment '用户  
名',  
17     password varchar(32) default '123456' comment '密  
码',  
18     name varchar(10) not null comment '姓名',  
19     gender tinyint unsigned not null comment '性别, 说  
明: 1 男, 2 女',  
20     image varchar(300) comment '图像',  
21     job tinyint unsigned comment '职位, 说明: 1 班主任, 2  
讲师, 3 学工主管, 4 教研主管, 5 咨询师',  
22     entrydate date comment '入职时间',  
23     dept_id int unsigned comment '部门ID',  
24     create_time datetime not null comment '创建时间',  
25     update_time datetime not null comment '修改时间'  
26 ) comment '员工表';  
27  
28 INSERT INTO emp  
29 (id, username, password, name, gender, image, job, entrydate, dept_id,  
create_time, update_time) VALUES  
30  
    (1, 'jinyong', '123456', '金  
庸', 1, '1.jpg', 4, '2000-01-01', 2, now(), now()),  
31  
    (2, 'zhangwuji', '123456', '张无  
忌', 1, '2.jpg', 2, '2015-01-01', 2, now(), now()),  
32  
    (3, 'yangxiao', '123456', '杨  
逍', 1, '3.jpg', 2, '2008-05-01', 2, now(), now()),  
33  
    (4, 'weiyixiao', '123456', '韦一  
笑', 1, '4.jpg', 2, '2007-01-01', 2, now(), now()),
```

```

34                               (5, 'changyuchun', '123456', '常遇
35                               春', 1, '5.jpg', 2, '2012-12-05', 2, now(), now()),
36                               (6, 'xiaozhao', '123456', '小
37                               昭', 2, '6.jpg', 3, '2013-09-05', 1, now(), now()),
38                               (7, 'jixiaofu', '123456', '纪晓
39                               芙', 2, '7.jpg', 1, '2005-08-01', 1, now(), now()),
40                               (8, 'zhouzhiruo', '123456', '周芷
41                               若', 2, '8.jpg', 1, '2014-11-09', 1, now(), now()),
42                               (9, 'dingminjun', '123456', '丁敏
43                               君', 2, '9.jpg', 1, '2011-03-11', 1, now(), now()),
44                               (10, 'zhaomin', '123456', '赵
45                               敏', 2, '10.jpg', 1, '2013-09-05', 1, now(), now()),
46                               (11, 'luzhangke', '123456', '鹿杖
47                               客', 1, '11.jpg', 5, '2007-02-01', 3, now(), now()),
48                               (12, 'hebiweng', '123456', '鹤笔
49                               翁', 1, '12.jpg', 5, '2008-08-18', 3, now(), now()),
50                               (13, 'fangdongbai', '123456', '方东
51                               白', 1, '13.jpg', 5, '2012-11-01', 3, now(), now()),
52                               (14, 'zhangsanfeng', '123456', '张三
53                               丰', 1, '14.jpg', 2, '2002-08-01', 2, now(), now()),
54                               (15, 'yulianzhou', '123456', '俞莲
55                               舟', 1, '15.jpg', 2, '2011-05-01', 2, now(), now()),
56                               (16, 'songyuqiao', '123456', '宋远
57                               桥', 1, '16.jpg', 2, '2010-01-01', 2, now(), now()),
58                               (17, 'chenyouliang', '123456', '陈友
59                               谅', 1, '17.jpg', NULL, '2015-03-21', NULL, now(), now());

```

- 创建新的模块，引入 MyBatis MySQL Lombok
- 配置文件中引入数据库连接信息

```

1 # 配置数据库连接信息
2 spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
3 spring.datasource.url=jdbc:mysql://localhost:3306/mybatis
4 spring.datasource.username=root
5 spring.datasource.password=L=104-02

```

- 创建实体类Emp (采用驼峰命名)

```

1 package com.itheima.pojo;
2
3 import lombok.Data;

```

```
4  
5 import java.time.LocalDate;  
6 import java.time.LocalDateTime;  
7  
8 @Data  
9 public class Emp {  
10     private Integer id;  
11     private String username;  
12     private String password;  
13     private String name;  
14     private Short gender;  
15     private String image;  
16     private Short job;  
17     private LocalDate entrydate;  
18     private Integer dept_id;  
19     private LocalDateTime createTime;  
20     private LocalDateTime updateTime;  
21 }  
22 }
```

- 准备mapper接口EmpMapper

```
1 package com.itheima.mapper;  
2  
3 import org.apache.ibatis.annotations.Mapper;  
4  
5 @Mapper  
6 public interface EmpMapper {  
7 }
```

删除

根据ID删除数据，先写好固定数据的SQL语句

```
1 | delete from emp where id = 17;
```

但是在mapper接口中不能把id写死了，需要根据传入的id来删除，这样这个方法才能反复的使用，这里用到Mybatis中的占位符操作#{变量名}

```
1 package com.itheima.mapper;  
2  
3 import org.apache.ibatis.annotations.Delete;  
4 import org.apache.ibatis.annotations.Mapper;  
5  
6 @Mapper  
7 public interface EmpMapper {  
8  
9     // 根据id删除数据  
10    @Delete("delete from emp where id = #{id};")  
11    public void delete(Integer id);  
12 }
```

注意：**delete实际上有返回值，返回值为影响的行数**，不过这里不需要这个返回值所以用void。

如果mapper接口方法形参只有一个普通类型的参数，那么`#{} 中的属性名可以随便写。（建议始终保持一致）`

删除（预编译SQL）

MyBatis框架底层到底执行了什么样的SQL语句，结果什么样，并没有直观地看到。实际上可以借助MyBatis的日志来看到这些信息，日志信息默认关闭。需要在配置文件中打开日志并指定要输出日志到控制台。

```
1 # 配置Mybatis的日志，指定将其输出到控制台  
2 mybatis.configuration.log-impl=org.apache.ibatis.logging.stdout.StdoutImpl
```

输出的日志信息如下

```
--> Preparing: delete from emp where id = ?;  
--> Parameters: 16(Integer)  
<== Updates: 0
```

? 为参数占位符，会使用Parameters的参数替换掉占位符？，然后删除id为16的员工信息，updates为0代表这句话影响的记录数为0，没有真删除掉数据

前两行的SQL语句称为预编译SQL，`#{} 会被问号替代，生成预编译的SQL，最终执行时会把SQL语句和参数都发给数据库，数据库执行时会用16替换掉问号。`

预编译SQL的优点

- 性能更高

SQL执行流程：Java中编写了一句SQL，要想执行就需要连接数据库将语句发给数据库服务器，服务器需要进行一系列操作，如语法解析，优化器优化SQL，编译SQL，最后执行SQL。为提高效率，数据库会将优化编译后的SQL缓存起来，下一次再执行SQL时会先检查缓存是否有编译好的SQL语句，有就不用再解析、优化、编译了，可以直接执行。如果没有，就需要执行这些操作，再将语句缓存起来。

无预编译：当执行删除id=1 2 3的三条语句时，仅因为id值不同就算作不同语句，**需要编译三次**。

预编译：不会把字段值直接拼接在SQL语句中，而是使用问号占位符，当删除id=1的数据时，会将SQL语句和参数值1发给数据库，数据库先检查缓存中有无数据，发现没有会进行检查、优化、编译并将加过加入到缓存中，再执行SQL语句，会使用参数值将问号替换掉。删除id=2的数据，缓存中已经缓存了带占位符的这个语句，就不用再检查、优化、编译了，直接将参数替换掉问号后执行。id=3时同理。不管需要执行多少次，**只需要编译一次**。

- 更安全（防SQL注入）

通过操作输入的数据修改定义好的SQL语句，达到攻击服务器的方法。

- 两种占位符
 - `#{}，会被问号替代，生成的是预编译SQL，用于参数传递。`
 - `${}，拼接SQL，存在注入问题，对表名、字段名进行动态设置时使用`

新增

新增员工操作，先写好固定数据的SQL语句

```
1 | insert into emp(username, name, gender, image, job, entrydate, dept_id,
2 | create_time, update_time) values ('Tom', '汤姆', 1, '1.jpg', 1, '2005-01-01',
3 | 1, now(), now())
```

同样需要动态传入数据，但这里数据很多，虽然也可以一个一个的传，但**更推荐使用对象封装**，在使用#{}占位符时里边的**变量名就写对象的属性名**

```
1 | package com.itheima.mapper;
2 |
3 | import com.itheima.pojo.Emp;
4 | import org.apache.ibatis.annotations.Delete;
5 | import org.apache.ibatis.annotations.Insert;
6 | import org.apache.ibatis.annotations.Mapper;
7 |
8 | @Mapper
9 | public interface EmpMapper {
10 |     // 新增员工
11 |     @Insert("insert into emp(username, name, gender, image, job, entrydate,
12 |     dept_id, create_time, update_time) " +
13 |             "values (#{username}, #{name}, #{gender}, #{image}, #{job}, #
14 |             {entrydate}, #{deptId}, #{createTime}, #{updateTime})")
15 |     void insert(Emp emp);
16 | }
```

新增（主键返回）

数据添加成功后，需要获取插入数据库数据的主键。如添加套餐数据时还需要维护套餐菜品关系表数据。

默认情况下，插入操作不会把主键值返回。要想拿到主键值，可以在接口方法上再加上一个注解@Options(useGeneratedKeys = true, keyProperty = "id")，useGeneratedKeys = true表示需要拿到生成的主键值，keyProperty = "id"表示获得的主键最终会封装到Emp对象的id属性中。

```
1 | package com.itheima.mapper;
2 |
3 | import com.itheima.pojo.Emp;
4 | import org.apache.ibatis.annotations.Delete;
5 | import org.apache.ibatis.annotations.Insert;
6 | import org.apache.ibatis.annotations.Mapper;
7 | import org.apache.ibatis.annotations.Options;
8 |
9 | @Mapper
10 | public interface EmpMapper {
11 |     // 新增员工
12 |     @Options(useGeneratedKeys = true, keyProperty = "id")
13 |     @Insert("insert into emp(username, name, gender, image, job, entrydate,
14 |     dept_id, create_time, update_time) " +
15 |             "values (#{username}, #{name}, #{gender}, #{image}, #{job}, #
16 |             {entrydate}, #{deptId}, #{createTime}, #{updateTime})")
17 |     void insert(Emp emp);
18 | }
```

更新

先写好固定数据的SQL语句

```
1 | update emp set username = 'Tom1', name = '汤姆1', gender = 1, image = '1.jpg',
  | job = 1, entrydate = '2000-01-01', dept_id = 1, update_time = now() where id
  | = 18
```

同样可以把参数值封装到一个对象中，并使用占位符操作

```
1 | package com.itheima.mapper;
2 |
3 | import com.itheima.pojo.Emp;
4 | import org.apache.ibatis.annotations.*;
5 |
6 | @Mapper
7 | public interface EmpMapper {
8 |
9 |     // 根据id删除数据
10 |     @Delete("delete from emp where id = #{id};")
11 |     void delete(Integer id);
12 |     // int delete(Integer id);
13 |
14 |
15 |     // 新增员工
16 |     @Options(useGeneratedKeys = true, keyProperty = "id")
17 |     @Insert("insert into emp(username, name, gender, image, job, entrydate,
18 |     dept_id, create_time, update_time) +
19 |             values (#{username}, #{name}, #{gender}, #{image}, #{job}, #
20 |             {entrydate}, #{deptId}, #{createTime}, #{updateTime})")
21 |     void insert(Emp emp);
22 |
23 |     // 更新员工
24 |     @Update("update emp set username = '', name = '', gender = '',
25 |             image = '', job = '', entrycode = '', dept_id = '',
26 |             update_time = '' where id = 1")
27 |     void update(Emp emp);
28 }
```

查询（根据id查询）

需要传入一个参数id

```
1 | package com.itheima.mapper;
2 |
3 | import com.itheima.pojo.Emp;
4 | import org.apache.ibatis.annotations.*;
5 |
6 | @Mapper
7 | public interface EmpMapper {
8 |
9 |     // 根据id删除数据
10 |     @Delete("delete from emp where id = #{id};")
11 |     void delete(Integer id);
12 |     // int delete(Integer id);
13 |
14 |
15 |     // 新增员工
16 |     @Options(useGeneratedKeys = true, keyProperty = "id")
```

```

17     @Insert("insert into emp(username, name, gender, image, job, entrydate,
18         dept_id, create_time, update_time) " +
19             "values (#{username}, #{name}, #{gender}, #{image}, #{job}, #
{entrydate}, #{deptId}, #{createTime}, #{updateTime})")
20     void insert(Emp emp);
21
22     // 更新员工
23     @Update("update emp set username = #{username}, name = #{name}, gender =
#{gender}, image = #{image}, job = #{job}, entrydate = #{entrydate}, dept_id
= #{deptId}, update_time = #{updateTime} where id = #{id}")
24     void update(Emp emp);
25
26     // 根据id查询员工
27     @Select("select * from emp where id = #{id}")
28     Emp getById(Integer id);
29 }
```

注意：在测试输出的结果中会发现 `deptId createTime updateTime` 三个值都是null。

- 这是因为实体类属性名和数据库表查询返回的字段名一致，Mybatis会自动封装。
- 如果实体类属性名和数据库表查询返回的字段名不一致，不能自动封装。

解决办法：

- 方案1：在使用SQL语句查询时给字段起别名

```

1  @Mapper
2  public interface EmpMapper {
3      // 方案1：给字段起别名，和实体类属性名一致
4      @Select("select id, username, password, name, gender, image, job,
5          entrydate, " +
6              "dept_id as deptId, create_time as createTime,
7              update_time as updateTime from emp where id = #{id}")
8      Emp getById(Integer id);
9  }
```

- 方案2：通过注解 `@Results` 和 `@Result` 手动映射封装

```

1  @Mapper
2  public interface EmpMapper {
3      // 方案2：通过@Results, @Result注解手动映射封装
4      @Results({
5          @Result(column = "dept_id", property = "deptId"),
6          @Result(column = "create_time", property =
7              "createTime"),
8          @Result(column = "update_time", property =
9              "updateTime")
10     })
11     @Select("select id, username, password, name, gender, image,
12         job, entrydate, dept_id as deptId, create_time as createTime,
13         update_time as updateTime from emp where id = #{id}")
14     Emp getById(Integer id);
15 }
```

实际上两种解决方法都比较繁琐臃肿，使用的都较少

- 方案3：开启MyBatis驼峰命名自动映射开关，使MyBatis自动将如 `a_column` 的字段封装到实体类 `aColumn` 的属性当中。注意：使用的前提是严格遵守命名规则

需要在配置文件中配置

```
1 # 配置Mybatis的驼峰命名自动映射开关 a_column -> aColumn
2 mybatis.configuration.map-underscore-to-camel-case=true
```

查询（条件查询）

根据姓名、性别、入职时间进行查询，要求姓名支持模糊匹配，性别进行精确查询，入职时间进行范围查询。

暂不考虑分页查询。根据最后修改时间进行倒序排序

```
1 select * from emp where name like '%张%' and gender = 1 and entrydate between
'2010-01-01' and '2020-01-01' order by update_time desc;
```

发现这里不好用对象封装这些数据，干脆直接将四个参数直接传进去

注意：字符串中写了`'%%'`来进行模糊查询，而`#{}不能出现在引号之内`，可以使用`{}来直接进行拼接而不是预编译`

也可以使用SQL的字符串拼接函数`concat(参数1, 参数2, ...)`（推荐使用）

```
1 package com.itheima.mapper;
2
3 import com.itheima.pojo.Emp;
4 import org.apache.ibatis.annotations.*;
5
6 import java.time.LocalDate;
7 import java.util.List;
8
9 @Mapper
10 public interface EmpMapper {
11     // 条件查询员工
12     // 注意：字符串中写了%%表示模糊查询，而#{}不能出现在引号之内，可以使用${}来拼接
13     // @Select("select * from emp where name like '%${name}%' and gender = #"
14     // {gender} " +
15     //         "and entrydate between #{begin} and #{end} order by update_time
16     // desc;")
17     // 也可使用SQL的字符串拼接函数concat(串1, 串2, ...)
18     @Select("select * from emp where name like concat('%', #{name}, '%') and
19     gender = #{gender} " +
20     "and entrydate between #{begin} and #{end} order by update_time
desc;")
21     List<Emp> list(String name, Short gender, LocalDate begin, LocalDate
end);
22 }
```

注：在SpringBoot1.0版本或者单独使用Mybatis时，需要在函数参数列表中对每个参数使用`@Param(SQL#{}中的名字)`来指定每个参数在SQL语句中的名字。这是因为在对mapper接口编译时并不会保留形参名称，参数名会变为`var1 var2 var3...`。SpringBoot2.0以上内置了编译插件，编译时会将方法形参名称保留下，编译后参数名不变，这样`#{}就可以找到对应的参数，也就不再需要添加注解。`

XML映射文件（也叫配置文件）

三条规范

- 映射文件的名称要和mapper接口名称一致，并将映射文件和mapper接口放在相同包下（同包同名），也就是需要在 resources 目录下创建一模一样的包结构
- 映射文件的 namespace 属性和mapper接口全限定类名一致，全限定类名就是 包名+类名
- 映射文件中SQL语句的id要和mapper接口中的方法名一致，并保持返回类型一致

即，映射文件和接口有同样的包路径，配置文件要和接口关联起来（通过namespace属性），配置文件中的SQL要和接口中的方法关联起来（通过id属性）

The diagram illustrates the relationship between a Java Mapper interface and its corresponding XML mapping file. On the left, a Java code snippet shows a public interface named EmpMapper with a single method list that takes a String name, a Short gender, and two LocalDate parameters begin and end, and returns a List of Emp objects. This is labeled 'Mapper接口'. On the right, a corresponding XML mapping file is shown, starting with a <mapper> tag that specifies the namespace as com.itheima.mapper.EmpMapper. Inside this tag is a <select> tag with an id of "list" and a resultType of com.itheima.pojo.Emp. The SQL query within the <select> tag performs a search based on name (like concat('%',#{name},'%')) and gender (#{gender}), and filters by entrydate between #{begin} and #{end}, ordering by update_time desc. This is labeled 'XML映射文件'.

```
@Mapper
public interface EmpMapper {
    public List<Emp> list(String name, Short gender, LocalDate begin, LocalDate end);
}
```



```
<mapper namespace="com.itheima.mapper.EmpMapper">
    <select id="list" resultType="com.itheima.pojo.Emp">
        select * from emp where name like concat('%',#{name},'%') and gender = #{gender}
        and entrydate between #{begin} and #{end} order by update_time desc
    </select>
</mapper>
```

定义一个XML映射文件，文件头直接上[官网](#)复制一份，文件中有一个根标签mapper， mapper有一个唯一的属性**namespace**，属性值是**mapper接口的全限定名**。以条件查询员工为例，直接将注解中的SQL语句复制过来放到 <select> 标签中，并按照规范为 <select> 标签设置 **id**， **id** 要和**mapper接口中的方法名一致**。如果有返回值，则为标签添加 **resultType** 属性，属性值为封装的对象类型的全限定类名。

```
1 <?xml version="1.0" encoding="UTF-8" ?>
2 <!DOCTYPE mapper
3     PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
4     "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
5 <mapper namespace="com.itheima.mapper.EmpMapper">
6     <!--resultType: 单条数据所封装的类型，同样要写全限定类名-->
7     <select id="list" resultType="com.itheima.pojo.Emp">
8         select * from emp where name like concat('%', #{name}, '%') and
9             gender = #{gender} and
10            entrydate between #{begin} and #{end} order
11            by update_time desc;
    </select>
</mapper>
```

mapper接口中的方法就不再需要注解

```
1 package com.itheima.mapper;
2
3 import com.itheima.pojo.Emp;
4 import org.apache.ibatis.annotations.*;
5
6 import java.time.LocalDate;
```

```

7 import java.util.List;
8
9 @Mapper
10 public interface EmpMapper {
11     // 条件查询员工
12     // 注意：字符串中写了%%表示模糊查询，而#{ }不能出现在引号之内，可以使用${ }来拼接
13     // 也可使用SQL的字符串拼接函数concat(串1, 串2, ...)
14     //     @Select("select * from emp where name like '%${name}%' and gender = #{gender} " +
15     //             "and entrydate between #{begin} and #{end} order by
16     //             update_time desc;")
17     //     @Select("select * from emp where name like concat('%', #{name}, '%') "
18     //             "and gender = #{gender} " +
19     //             "and entrydate between #{begin} and #{end} order by
20     //             update_time desc;")
21     //     List<Emp> list(String name, Short gender, LocalDate begin, LocalDate
22     // end);
23 }

```

为什么要遵守三个规范？

通过MyBatis框架操作数据库，最终只需要调用mapper接口中的方法就可以完成数据库的操作。但**最终操作数据库不是靠这个方法，而是这个方法对应的SQL语句**。关键点就是通过接口中的list方法怎么找到相关联的SQL语句，注解方式就是直接执行方法上边的SQL语句，而XML配置文件的形式，方法和SQL语句时分开的，现在就需要根据方法找到对应的SQL语句。按照三个规范定义了一个XML映射文件，当调用接口方法的时候Mybatis就会自动查找 namespace 属性值域接口全类名相同的XML映射文件，并在文件中找到 id 属性值域方法名相同的SQL语句，最终来运行找到的语句完成数据库操作。

如何选择注解和XML文件？简单操作使用注解，比较复杂的操作最好使用XML配置文件，**不要局限于一种形式**

- MyBatisX插件

会出现很多的鸟，点击行号边的鸟会在接口和XML文件之间进行跳转

(功能待补充)

注：插件对mapper接口的list方法有个错误误报，原因是这个类和 requestAndResponse 模块中的类完全重名，不用关心。而且这也只会在学习阶段出现

9. MyBatis动态SQL

随着用户的输入或外部条件的变化而变化的SQL语句称为动态SQL

之前写的条件查询的SQL语句要求三个条件都要传入，如果不全传则查不到任何数据。

<if>

可以使用 if 标签改造SQL语句，用来判断条件是否成立，使用 test 属性进行条件判断，条件为true则拼接SQL。

修改XML文件：

```

1  <?xml version="1.0" encoding="UTF-8" ?>
2  <!DOCTYPE mapper
3      PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
4      "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
5  <mapper namespace="com.itheima.mapper.EmpMapper">
6
7      <!--
8          - 映射文件的名称要和mapper接口名称一致，并将映射文件和mapper接口放在相同包下（同包同名），也就是需要在`resources`目录下创建一模一样的包结构
9          - 映射文件的`namespace`属性和mapper接口全限定类名一致，全限定类名就是`包名+类名`、
10         - 映射文件中SQL语句的id要和mapper接口中的方法名一致，并保持返回类型一致
11     -->
12     <!--resultType: 单条数据所封装的类型，同样要写全限定类名-->
13     <select id="list" resultType="com.itheima.pojo.Emp">
14         select *
15         from emp
16         where
17             <if test="name != null">
18                 name like concat('%', #{name}, '%')
19             </if>
20             <if test="gender != null">
21                 and gender = #{gender}
22             </if>
23             <if test="begin != null and end != null">
24                 and entrydate between #{begin} and #{end}
25             </if>
26             order by update_time desc;
27     </select>
28 </mapper>

```

注意：在XML文件中进行格式化，IDEA会自动将SQL语句按字段分行

测试时发现，如果 name 为null，那么SQL语句就出现了语法错误：`select * from emp where and gender = ? order by update_time desc;`，此时的where与gender之间出现了一个and。

<where>

动态SQL中提供了一个标签来替代 `where`，就是 `<where>` 标签。用 `<where>` 标签将所有查询条件包裹起来，标签有两个作用，第一个作用是**动态生成 where 关键字**，`<where>` 会根据子标签判断是否有条件成立，如果所有条件都不成立，SQL语句中的 `where` 关键字也不会被生成，如果有任何一个成立就会生成 `where` 关键字。第二个作用就是会**自动去除条件前多余的 and/or**。使用 `<where>` 标签后，测试就不会报错了。

if案例

通过更新员工的方法，将id为18的员工username改为Tom111，name改为汤姆111，gender改为2

测试代码如下，把不需要更新的属性都去掉：

```

1  @Test
2  public void testUpdate2() {
3      Emp emp = new Emp();
4      emp.setId(18);
5      emp.setUsername("Tom111");
6      emp.setName("汤姆111");
7      emp.setGender((short)2);
8      emp.setUpdateTime(LocalDateTime.now());
9
10     // 执行更新员工操作
11     empMapper.update(emp);
12 }

```

查看数据库表会发现后边四个字段都变成了null

	id	username	password	name	gender	image	job	entrydate	dept_id
0	Xiaozidu	123456	小暗		2	0.jpg	3	2013-07-03	1
7	jixiaofu	123456	纪晓芙		2	7.jpg	1	2005-08-01	1
8	zhouzhiruo	123456	周芷若		2	8.jpg	1	2014-11-09	1
9	dingminjun	123456	丁敏君		2	9.jpg	1	2011-03-11	1
10	zhaomin	123456	赵敏		2	10.jpg	1	2013-09-05	1
11	luzhangke	123456	鹿杖客		1	11.jpg	5	2007-02-01	3
12	hebiweng	123456	鹤笔翁		1	12.jpg	5	2008-08-18	3
13	fangdongbai	123456	方东白		1	13.jpg	5	2012-11-01	3
14	zhangsanfeng	123456	张三丰		1	14.jpg	2	2002-08-01	2
15	yulianzhou	123456	俞莲舟		1	15.jpg	2	2011-05-01	2
18	Tom111	123456	汤姆111		2	<null>	<null>	<null>	<null>

调用update方法时后几个字段都没有赋值，会被更新为null。

将更新员工信息改为动态更新，有值就更新，没有值就不更新。

修改mapper接口中的方法，此处直接定义一个新方法，不在原有代码上修改

```

1 | void update2(Emp emp);

```

添加XML配置文件中的SQL语句，如果有MybatisX插件，则可以 alt+enter 来自动生成一对标签，插件会读取方法名来判断生成哪个标签，如有 update 关键字就会生成 <update> 标签，有 select get query 关键字就会生成 <select> 标签。如果方法名中没有类似的关键字， alt+enter 时插件会弹出一个选择框来选择要生成的标签。

使用IDEA自带的格式化，将字段自动分行

```

1 <update id="update2">
2   update emp
3     set username    = #{username},
4     name          = #{name},
5     gender        = #{gender},
6     image         = #{image},
7     job           = #{job},
8     entrydate    = #{entrydate},
9     dept_id       = #{deptId},
10    update_time   = #{updateTime}
11    where id = #{id}
12 </update>

```

修改成动态SQL语句

```

1 <update id="update2">
2   update emp
3   set
4     <if test="username != null">username = #{username},</if>
5     <if test="name != null">name = #{name},</if>
6     <if test="gender != null">gender = #{gender},</if>
7     <if test="image != null">image = #{image},</if>
8     <if test="job != null">job = #{job},</if>
9     <if test="entrydate != null">entrydate = #{entrydate},</if>
10    <if test="deptId != null">dept_id = #{deptId},</if>
11    <if test="updateTime != null">update_time = #{updateTime}</if>
12    where id = #{id}
13 </update>

```

测试时发现，如果只更新username，那么SQL语句出现了语法错误：`update emp set username = ? , where id = ?`，此时的where前多了一个逗号

<set>

Mybatis又提供了一个`<set>`标签，用`<set>`标签可以将所有更新字段进行包裹用来动态生成`set`关键字，也会去除字段之后多余的逗号。

修改XML中的标签

```

1 <update id="update2">
2   update emp
3   <set>
4     <if test="username != null">username = #{username},</if>
5     <if test="name != null">name = #{name},</if>
6     <if test="gender != null">gender = #{gender},</if>
7     <if test="image != null">image = #{image},</if>
8     <if test="job != null">job = #{job},</if>
9     <if test="entrydate != null">entrydate = #{entrydate},</if>
10    <if test="deptId != null">dept_id = #{deptId},</if>
11    <if test="updateTime != null">update_time = #{updateTime}</if>
12  </set>
13  where id = #{id}
14 </update>

```

此时就可以正常执行更新操作了

<foreach>

- 标签属性值
 1. `collection`：要遍历的集合，和接口方法的参数名保持一致
 2. `item`：遍历出来的元素，就是写在SQL中的变量名
 3. `separator`：进行拼接时的分隔符
 4. `open`：遍历开始前拼接的片段
 5. `close`：遍历结束后拼接的片段

批量删除员工

先在mapper中定义方法

```

1 | void deleteByids(List<Integer> ids);

```

在XML中写SQL语句

```
1 <delete id="deleteByIds">
2   delete
3     from emp
4     where id in
5       <foreach collection="ids" item="id" separator="," open="(" close="")">
6         #{id}
7       </foreach>
8 </delete>
```

<sql>和<include>

目前XML映射文件的问题：可能存在大量重复的SQL片段。

```
<select id="list" resultType="com.itheima.pojo.Emp">
  select id, username, password, name, gender, image, job, entrydate, dept_id, create_time, update_time from emp
  where
    <if test="name != null">
      name like concat('%',#{name},'%')
    </if>
    <if test="gender != null">
      and gender = #{gender}
    </if>
    <if test="begin != null and end != null">
      and entrydate between #{begin} and #{end}
    </if>
  order by update_time desc
</select>

<select id="getById" resultType="com.itheima.pojo.Emp">
  select id, username, password, name, gender, image, job, entrydate, dept_id, create_time, update_time from emp
  where id = #{id}
</select>
```

可以和Java一样，将重复的代码进行抽取，在抽取的地方再引入进来，这就要使用 `<sql>` 和 `<include>` 标签。`<sql>` 标签负责抽取，抽取的时候需要给SQL片段定一个唯一标识也就是 `id` 属性，`<include>` 标签负责在原来抽取的位置将 `sql` 片段引用进来，通过 `refid` 属性来指定要引入的SQL片段。

修改动态查询员工

注：在SQL中不建议使用 `*` 来选中所有的字段，而是将字段名都列出来

```
1 <sql id="commonSelect">
2   select id, username, password, name, gender, image, job, entrydate,
3     dept_id, create_time, update_time from emp
4 </sql>
5 <select id="list" resultType="com.itheima.pojo.Emp">
6   <include refid="commonSelect"/>
7   <where>
8     <if test="name != null">
9       name like concat('%', #{name}, '%')
10      </if>
11      <if test="gender != null">
12        and gender = #{gender}
13      </if>
```

```
13     <if test="begin != null and end != null">
14         and entrydate between #{begin} and #{end}
15     </if>
16     </where>
17     order by update_time desc;
18 </select>
```

10. 部门管理员工管理案例

部门新增、删除、编辑

员工条件查询、新增、编辑、删除、分页查询

环境搭建

后端工程和数据库环境准备

- 准备数据库表 (dept、emp)
- 创建工程引入依赖 (Web、MyBatis、MySQL驱动、Lombok)
- 配置MyBatis，准备实体类
- 准备Mapper、Service (接口+实体类)、Controller结构，分别加上注解

开发规范restful

REST (REpresentational State Transfer)，表达性状态转换，是一种软件架构风格（说的什么b话）

描述网络资源有两种方式

- 传统风格

`http://localhost:8080/user/getById?id=1`

`http://localhost:8080/user/saveUser`

URL的定义完全看个人喜好，没有统一标准和统一规范

- REST风格

`http://localhost:8080/user/1`，知道访问id为1的用户信息，具体是查询删除还是更新，通过URL是不知道的。

两个要点：通过URL定位资源，通过HTTP动词描述操作，动词指的是HTTP请求方式，操作指增删改查。

如get方法请求表示查询id为1的用户，delete方法请求表示删除id为1的用户，post表示新增，put表示修改

注意：

1. 是约定方式，可以不遵守
2. 描述模块功能常使用复数，表示此类资源而非单个资源。如：users、emps

开发规范统一响应结果

- 前后端交互统一响应结果Result

开发流程

- 首先要查看页面原型明确需求，根据原型和需求定义表结构和接口文档。
- 目前来说第二步是阅读接口文档，思路分析（一般接口文档都由后端来写）
- 接口开发，测试（工具：postman、apifox）

- 前后端联调

部门管理部分

- 部门列表查询

先看接口文档，确定请求路径 /depts、请求方式 GET、请求参数 无，和最后的响应数据格式 json 和参数。

确定三层架构每层都干什么和具体流程

- DeptController：接受请求，调用Service查询部门，响应结果
- DeptService：调用Mapper进行查询
- DeptMapper：执行SQL进行查询

先实现DeptController的方法

注意：在实际开发中要尽量不使用 `System.out.println`，而是使用日志对象进行打印输出。而且也不需要手动定义日志对象，Lombok提供了一个注解 `@Slf4j`，这样就可以直接调用叫 `log` 的对象中的 `info` 方法来记录日志。

```

1 package com.itheima.controller;
2
3 import com.itheima.pojo.Result;
4 import lombok.extern.slf4j.Slf4j;
5 import org.springframework.web.bind.annotation.RequestMapping;
6 import org.springframework.web.bind.annotation.RestController;
7
8
9 @Slf4j
10 @RestController
11 public class DeptController {
12
13     // private static Logger log =
14     // LoggerFactory.getLogger(DeptController.class);
15
16     @RequestMapping("/depts")
17     public Result list() {
18         log.info("查询全部部门数据");
19         return Result.success();
20     }
21 }
```

现在的这个接口通过各种请求方式访问都是可以的，但接口文档中要求当前接口的请求方式得是 get，如何限定接口的请求方式？实际上在 `@RequestMapping` 注解中有一个属性 `method` 就是用来指定请求方法的。

注意：如果有多个属性，就必须指定每个属性的属性名

```

1 package com.itheima.controller;
2
3 import com.itheima.pojo.Result;
4 import lombok.extern.slf4j.Slf4j;
5 import org.springframework.web.bind.annotation.RequestMapping;
6 import org.springframework.web.bind.annotation.RequestMethod;
7 import org.springframework.web.bind.annotation.RestController;
8
9 
```

```
10 @Slf4j
11 @RestController
12 public class DeptController {
13
14     // private static Logger log =
15     LoggerFactory.getLogger(DeptController.class);
16
17     @RequestMapping(value = "/depts", method = RequestMethod.GET) // 指
18     定请求方式为GET
19     public Result list() {
20         log.info("查询全部部门数据");
21         return Result.success();
22     }
23 }
```

RequestMethod是一个枚举类型，如果要指定 post，就直接调用 RequestMethod.POST 就可以。不过现在这样写还是有些繁琐，Spring提供了一个衍生注解 @GetMapping()，然后在里面再指定请求路径为 /depts。

```
1 public class DeptController {
2
3     // private static Logger log =
4     LoggerFactory.getLogger(DeptController.class);
5
6     // @RequestMapping(value = "/depts", method = RequestMethod.GET) // 指
7     定请求方式为GET
8     @GetMapping("/depts")
9     public Result list() {
10        log.info("查询全部部门数据");
11        return Result.success();
12    }
13 }
```

如果想指定其他的请求方式，也可以用类似的注解，如 @PostMapping @DeleteMapping。

修改完注解后使用 apifox 进行测试会发现，在使用非GET方式请求时，会返回一个405错误，代表请求方式不允许，只有用GET方式来请求才会返回正常的结果。

```
{
  "timestamp": "2024-02-16T14:14:31.346+00:00",
  "status": 405,
  "error": "Method Not Allowed",
  "path": "/depts"
}
```

继续代码实现，在Controller中注入Service对象，使用自动注入，在DeptService接口中定义返回查询结果的方法 list，在DeptServiceImpl中实现接口的方法并注入 Mapper 对象（同样用自动注入），最后在 Mapper 接口中声明查询的方法也叫 list，这个SQL语句比较简单，可以使用注解来写。

```
1 @Slf4j
2 @RestController
3 public class DeptController {
4
5     @Autowired
6     private DeptService deptService = new DeptServiceImpl();
```

```
7     // private static Logger log =
8     LoggerFactory.getLogger(DeptController.class);
9
10    //    @RequestMapping(value = "/depts", method = RequestMethod.GET) // 指定请求方式为GET
11    @GetMapping("/depts")
12    public Result list() {
13        log.info("查询全部部门数据");
14        List<Dept> deptList = deptService.list();
15        return Result.success(deptList);
16    }
17}
18
```

定义返回查询结果的方法

```
1 public interface DeptService {
2
3     /**
4      * 查询全部部门数据
5      * @return
6      */
7     List<Dept> list();
8 }
```

在实体类中实现方法，调用Mapper接口来查询数据

```
1 @Service
2 public class DeptServiceImpl implements DeptService {
3
4     @Autowired
5     private DeptMapper deptMapper;
6
7     @Override
8     public List<Dept> list() {
9         return deptMapper.list();
10    }
11 }
```

定义Mapper接口中的查询方法，使用注解来写SQL语句

```
1 @Mapper
2 public interface DeptMapper {
3
4     /**
5      * 查询全部部门
6      * @return
7      */
8     @Select("select * from dept")
9     List<Dept> list();
10 }
```

具体的请求流程：前端发送的请求会请求到Controller中的 `list` 方法，方法中调用 `service` 来获取数据，`Service` 中调用了 `Mapper` 接口中的方法，`Mapper` 接口会向数据库中发送SQL语句来查询全部的部门，并将部门信息封装在 `List` 集合中，返回给 `service`，`service` 又返回给 `Controller`，Controller拿到数据后通过静态方法 `success` 将数据返回回去。

使用apifox进行测试，apifox请求获得的返回结果如下：

```
{  
    "code": 1,  
    "msg": "success",  
    "data": [  
        {  
            "id": 1,  
            "name": "学工部",  
            "createTime": "2024-02-16T12:31:03",  
            "updateTime": "2024-02-16T12:31:03"  
        },  
        {  
            "id": 2,  
            "name": "教研部",  
            "createTime": "2024-02-16T12:31:03",  
            "updateTime": "2024-02-16T12:31:03"  
        },  
        {  
            "id": 3,  
            "name": "咨询部",  
            "createTime": "2024-02-16T12:31:03",  
            "updateTime": "2024-02-16T12:31:03"  
        },  
        {  
            "id": 4,  
            "name": "就业部",  
            "createTime": "2024-02-16T12:31:03",  
            "updateTime": "2024-02-16T12:31:03"  
        }  
    ]  
}
```

- 前后端联调

启动前端资料，将压缩包解压运行 `nginx.exe`，在任务管理器的详细信息页中可以查看nginx是否运行成功。

成功后访问 `localhost:90` 即可打开页面，打开部门管理页面，F12查看网络中的depts请求，nginx最终会将请求转给后端的8080端口的Tomcat，实际上是Tomcat来处理的这次请求。响应数据就是JSON格式的数据，经过前端解析之后渲染展示在了表格当中。



序号	部门名称	最后操作时间	操作	
1	学工部	2024-02-16 12:31:03	编辑	删除
2	教研部	2024-02-16 12:31:03	编辑	删除
3	咨询部	2024-02-16 12:31:03	编辑	删除
4	就业部	2024-02-16 12:31:03	编辑	删除
5	人事部	2024-02-16 12:31:03	编辑	删除

- 删除部门

1. 先看页面原型中的需求，发现这是给前端看的，实际上后端的任务是做一个接口，根据id来删除当前部门的信息

3. 删除部门

弹出确认框，提示“您确定要删除该部门的信息吗？”如果选择确定，则删除该部门，删除成功后，重新刷新列表页面。如果选择了取消，则不执行任何操作。

2. 看接口文档

1. 请求路径: /depts/{id}
2. 请求方式: DELETE
3. 请求参数: 部门id
4. 响应数据: 格式为JSON，返回数据为非必须

3. 修改DeptController，接受请求，调用Service删除部门，响应结果

接收路径参数的注解: `@PathVariable`

```
1  @Slf4j
2  @RestController
3  public class DeptController {
4
5      @Autowired
6      private DeptService deptService = new DeptServiceImpl();
7
8      // private static Logger log =
9      // LoggerFactory.getLogger(DeptController.class);
10
11     // @RequestMapping(value = "/depts", method = RequestMethod.GET)
12     // 指定请求方式为GET
13     @GetMapping("/depts")
14     public Result list() {
15         log.info("查询全部部门数据");
16         List<Dept> deptList = deptService.list();
17         return Result.success(deptList);
18     }
19
20     @DeleteMapping("/depts/{id}")
21     public Result delete(@PathVariable Integer id) {
22         log.info("根据id删除部门: {}", id);
23         deptService.delete(id);
24         return Result.success();
25     }
26 }
```

4. 修改DeptService接口，添加delete方法

```
1  public interface DeptService {
2
3      /**
4      * 查询全部部门数据
5      * @return
6      */
7      List<Dept> list();
8
9      /**
10      * 根据id删除部门
11      * @param id
12      */
13      void delete(Integer id);
14 }
```

5. 修改DeptServiceImpl实体类，实现接口的方法，调用Mapper进行删除

```
1  @Service
2  public class DeptServiceImpl implements DeptService {
3
4      @Autowired
5      private DeptMapper deptMapper;
6
7      @Override
8      public List<Dept> list() {
9          return deptMapper.list();
10     }
11
12     @Override
13     public void delete(Integer id) {
14         deptMapper.delete(id);
15     }
16 }
```

6. 在Mapper中定义删除方法，用注解来写SQL语句

```
1  @Mapper
2  public interface DeptMapper {
3
4      /**
5       * 查询全部部门
6       * @return
7       */
8      @Select("select * from dept")
9      List<Dept> list();
10
11     @Delete("delete from dept where id = #{id};")
12     void delete(Integer id);
13 }
```

7. 使用apifox进行调试

8. 前后端联调

注：遇到了一点小问题，nginx的目录路径中不能有中文和空格，出错的原因可以查看
`logs/error.log`，现放置在D盘根目录下。

在网页中点击删除序号为4的部门，先点一下取消，查看数据库中数据没有变化；再点确定删掉这个部门，查看数据库中也没有了id为4的部门，联调即为成功。

- 新增部门

1. 看需求

2. 看文档

请求路径：/depts

请求方式：POST

请求参数：格式JSON，部门名称name

响应数据：统一响应结果Result

3. 修改Controller，JSON格式的参数通过实体类接收并封装，需要用注解`@RequestBody`，调用Service

```

1  @Slf4j
2  @RestController
3  public class DeptController {
4
5      @Autowired
6      private DeptService deptService = new DeptServiceImpl();
7
8      // private static Logger log =
9      // LoggerFactory.getLogger(DeptController.class);
10
11     // @RequestMapping(value = "/depts", method = RequestMethod.GET)
12     // 指定请求方式为GET
13     @GetMapping("/depts")
14     public Result list() {
15         log.info("查询全部部门数据");
16         List<Dept> deptList = deptService.list();
17         return Result.success(deptList);
18     }
19
20     @DeleteMapping("/depts/{id}")
21     public Result delete(@PathVariable Integer id) {
22         log.info("根据id删除部门: {}", id);
23         deptService.delete(id);
24         return Result.success();
25     }
26
27     @PostMapping("/depts")
28     public Result add(@RequestBody Dept dept) {
29         log.info("新增部门: {}", dept);
30         deptService.add(dept);
31
32     }

```

4. 修改DeptService和实现类，在将参数传递给Mapper接口前，需要先补全其他的属性

```

1  public interface DeptService {
2
3      /**
4      * 查询全部部门数据
5      * @return
6      */
7      List<Dept> list();
8
9      /**
10     * 根据id删除部门
11     * @param id
12     */
13     void delete(Integer id);
14
15     /**
16     * 添加部门
17     * @param dept
18     */
19     void add(Dept dept);
20 }

```

```

1  @Service
2  public class DeptServiceImpl implements DeptService {
3
4      @Autowired
5      private DeptMapper deptMapper;
6
7      @Override
8      public List<Dept> list() {
9          return deptMapper.list();
10     }
11
12     @Override
13     public void delete(Integer id) {
14         deptMapper.delete(id);
15     }
16
17     @Override
18     public void add(Dept dept) {
19         // 请求发来的参数只有部门名称，需要补全其他的属性
20         dept.setCreateTime(LocalDateTime.now());
21         dept.setUpdateTime(LocalDateTime.now());
22
23         deptMapper.insert(dept);
24     }
25 }
```

5. 修改Mapper接口，用注解写SQL语句

```

1  @Mapper
2  public interface DeptMapper {
3
4      /**
5       * 查询全部部门
6       * @return
7       */
8      @Select("select * from dept")
9      List<Dept> list();
10
11     @Delete("delete from dept where id = #{id};")
12     void delete(Integer id);
13
14
15     @Insert("insert into dept (name, create_time, update_time)
16     values (#{name}, #{createTime}, #{updateTime});")
17     void insert(Dept dept);
18 }
```

6. 使用apifox进行调试

7. 前后端联调

8. 优化Controller层的代码，现在DeptController中有三个方法，三个注解中请求的路径都以 /depts 开头，在Spring中为了简化请求路径的定义，可以将公共的请求路径直接抽取到类上，在类上写注解 @RequestMapping() 并将公共的路径抽取上去

`@RequestMapping("/depts")`, `@GetMapping("/depts")` 变成 `@GetMapping` 不再需要写

路径, `@DeleteMapping("/depts/{id}")`变成`@DeleteMapping("/{id}")`,
`@PostMapping("/depts")`变成`@PostMapping`

```
1  @Slf4j
2  @RestController
3  @RequestMapping("/depts")
4  public class DeptController {
5
6      @Autowired
7      private DeptService deptService = new DeptServiceImpl();
8
9      // private static Logger log =
10     LoggerFactory.getLogger(DeptController.class);
11
12     // @RequestMapping(value = "/depts", method = RequestMethod.GET)
13     // 指定请求方式为GET
14     // @GetMapping("/depts")
15     @GetMapping
16     public Result list() {
17         log.info("查询全部部门数据");
18         List<Dept> deptList = deptService.list();
19         return Result.success(deptList);
20     }
21
22     // @DeleteMapping("/depts/{id}")
23     // @DeleteMapping("/{id}")
24     public Result delete(@PathVariable Integer id) {
25         log.info("根据id删除部门: {}", id);
26         deptService.delete(id);
27         return Result.success();
28     }
29
30     // @PostMapping("/depts")
31     @PostMapping
32     public Result add(@RequestBody Dept dept) {
33         log.info("新增部门: {}", dept);
34         deptService.add(dept);
35
36     }
```

- 修改部门的前置操作：根据id查询数据

- 看原型

一个前置操作有个jb的原型

- 看文档

- 请求路径: /depts/{id}
- 请求方式: GET
- 请求参数: 部门名称id
- 响应数据: 统一返回结果Result, 可返回查询到的部门信息

3. Controller, 路径参数记得加注解！！！

```
1  @GetMapping("/{id}")
2  public Result getById(@PathVariable Integer id) {
3      log.info("根据id查询部门: {}", id);
4      Dept dept = deptService.getById(id);
5      return Result.success(dept);
6  }
```

4. DeptService和DeptServiceImpl

```
1 /**
2  * 根据id查询部门
3  * @param id
4  * @return
5  */
6 Dept getById(Integer id);
```

```
1 @Override
2 public Dept getById(Integer id) {
3     return deptMapper.getById(id);
4 }
```

5. Mapper接口

```
1 @Select("select * from dept where id = #{id};")
2 Dept getById(Integer id);
```

6. apifox接口调试

7. 前后端联调，因为是一个前置操作所以不能进行前后端联调

- 修改部门

1. 看原型

弹窗里输入部门名称

2. 看文档

请求路径: /depts

请求方式: PUT

请求参数: JSON格式, 部门id, 部门名称name

响应数据: JSON格式, 统一返回结果Result

3. Controller, **JSON参数记得加注解！！！**

```
1 @PutMapping
2 public Result updateDept(@RequestBody Dept dept) {
3     log.info("修改的部门: {}", dept);
4     deptService.updateDept(dept);
5     return Result.success();
6 }
```

4. Service, 需要补全updateTime

```
1 void updateDept(Dept dept);
```

```
1     @Override
2     public void updateDept(Dept dept) {
3         // 每次修改要记得修改update_time! !
4         dept.setUpdateTime(LocalDateTime.now());
5         deptMapper.updateDept(dept);
6     }
```

5. Mapper, 要更新记录的更新时间

```
1 @Update("update dept set name = #{name}, update_time = #{updateTime}
2 where id = #{id};")
3 void updateDept(Dept dept);
```

6. 接口测试

7. 前后端联调

员工管理部分

- 条件分页查询前置操作，无条件分页查询

选择每页要展示的记录数，点击页码后查询返回对应的数据。MySQL中通过limit实现分页操作

```
1 -- 分页查询语法
2 -- 参数1: 起始索引, 参数2: 查询返回的记录数
3 select * from emp limit 0, 5;
4
5 -- 查询第1页的数据, 每页展示五条记录
6 select * from emp limit 0, 5;
7
8 -- 查询第2页的数据, 每页展示五条记录
9 select * from emp limit 5, 5;
10
11 -- 查询第3页的数据, 每页展示五条记录
12 select * from emp limit 10, 5;
```

起始索引和页码、每页记录数的关系：起始索引 = (页码 - 1) \$\times\$ 每页记录数

前端发给后端的数据：当前页码，每页要展示的记录数

后端要返回给前端的数据：员工信息列表，总记录数。要给前端返回两项数据，可以考虑使用map集合或使用实体类封装（建议使用实体类）

0. 再声明一个实体类，用来封装返回的两种数据

```
1 package com.itheima.pojo;
2
3
4 import lombok.AllArgsConstructor;
5 import lombok.Data;
6 import lombok.NoArgsConstructor;
7
8 import java.util.List;
9
10 /**
11  * 分页查询结果的封装类
12 */
13 @Data
```

```
14     @NoArgsConstructor  
15     @AllArgsConstructor  
16     public class PageBean {  
17  
18         private Long total; // 总记录数  
19         private List rows; // 结果列表  
20     }
```

1. 看原型

2. 看文档

请求路径: /emps

请求方式: GET

请求参数: 查询页码page (default=1) , 每页记录数pageSize (default=10) (这里做的是无条件查询, 所以不考虑其它参数)

响应数据: 统一响应结果Result封装总记录数和数据列表

注意: 设置默认值可以使用 @RequestParam 注解, 里面有一个属性为 defaultValue 专门用来设置默认值。

3. Controller

```
1  @RestController  
2  public class EmpController {  
3  
4      @Autowired  
5      private EmpService empService;  
6  
7      @GetMapping("/emps")  
8      public Result page(@RequestParam(defaultValue = "1") Integer  
page,  
9                          @RequestParam(defaultValue = "10") Integer  
pageSize) {  
10         log.info("分页查询, 参数: {} {}", page, pageSize);  
11  
12         // 调用Service  
13         PageBean pageBean = empService.page(page, pageSize);  
14  
15         return Result.success(pageBean);  
16     }  
17 }
```

4. Service实现类

```
1  @Service  
2  public class EmpServiceImpl implements EmpService {  
3  
4      @Autowired  
5      private EmpMapper empMapper;  
6  
7      @Override  
8      public PageBean page(Integer page, Integer pageSize) {  
9          Long count = empMapper.count();  
10  
11          // 起始索引= (页码-1) *每页记录数  
12          Integer start = (page - 1) * pageSize;
```

```
13     List<Emp> pageList = empMapper.page(start, pageSize);
14     return new PageBean(count, pageList);
15 }
16 }
```

5. Mapper

```
1 @Mapper
2 public interface EmpMapper {
3
4     /**
5      * 查询总记录数
6      */
7     @Select("select count(*) from emp;")
8     Long count();
9
10    /**
11     * 获取列表数据
12     */
13    @Select("select * from emp limit #{begin}, #{pageSize}")
14    List<Emp> page(Integer begin, Integer pageSize);
15 }
```

6. 接口测试

7. 前后端联调

- 分页插件PageHelper

上述的分页查询代码步骤很固定，代码又显得略多。有组织专门开发了实现分页功能的插件。

统计总记录数的SQL语句、limit指定分页参数完成分页操作，插件都已经完成了

只需要正常的查询所有数据，调用PageHelper.startPage(page, pageSize)方法设置分页参数，再调用Mapper接口的方法正常查询就可以，返回的List集合真正的类型是Page类型，是插件提供的分页结果的封装类，不需要自己来定义。调用对应的方法来获取结果，如调用getTotal()方法获取分页查询结果的总记录数，getResult()获取分页结果的数据列表。

插件会对`select * from emp;`进行改造，先改成`select count(*) from emp;`然后自动执行，再改成`select * from emp limit ?, ?;`进行分页查询，两个占位符会根据startPage(page, pageSize)的两个参数自动计算起始索引和每页展示的记录数。

1. 使用前需引入依赖

2. Mapper中定义一个方法执行正常查询

```
1 @Select("select * from emp;")
2 List<Emp> list();
```

3. 修改Service，使用PageHelper插件的功能

```
1     @Override
2     public PageBean page(Integer page, Integer pageSize) {
3         //     Long count = empMapper.count();
4         //
5         //     // 起始索引= (页码-1) *每页记录数
6         //     Integer start = (page - 1) * pageSize;
7         //     List<Emp> pageList = empMapper.page(start, pageSize);
8         //     return new PageBean(count, pageList);
```

```

9
10
11     // 设置分页参数
12     PageHelper.startPage(page, pageSize);
13
14     // 执行查询
15     List<Emp> list = empMapper.list();
16     Page<Emp> p = (Page<Emp>) list;
17
18     // 封装PageBean对象
19     PageBean pageBean = new PageBean(p.getTotal(),
20     p.getResult());
21     return pageBean;
22 }
```

4. Controller不需要修改，他只负责和前端交互，后端的迭代不会影响到他

5. 接口测试

注：遇到一点小问题，看报错信息会发现SQL报错了，执行的SQL为`select * from emp; LIMIT ?`，发现是因为SQL语句中不能写分号，把分号删了就行了，只有一个占位符是因为查询的是第1页，起始索引为0被省略了

把Mapper接口注解SQL语句的分号删掉，测试就没有问题了

6. 前后端联调

- 条件分页查询

PageHelper只能处理普通的分页查询，其他的条件还需要写到注解的SQL中。

在无条件分页查询中进行改造，主要是对Mapper接口的SQL进行修改，参数可以传递也可以不传递所以需要用动态SQL。

1. 看原型

2. 看文档

请求路径：/emps

请求方式：GET

请求参数：name, gender, begin, end, 查询页码page (default=1) , 每页记录数pageSize (default=10) (这里做的是无条件查询，所以不考虑其它参数)

响应数据：统一响应结果Result封装总记录数和数据列表

3. 修改Controller

注意：接收日期参数记得加注解！！！

```

1     @GetMapping("/emps")
2     public Result page(@RequestParam(defaultValue = "1") Integer
page,
3                         @RequestParam(defaultValue = "10") Integer
pageSize,
4                         String name, Short gender,
5                         @DateTimeFormat(pattern = "yyyy-MM-dd")
LocalDate begin,
6                         @DateTimeFormat(pattern = "yyyy-MM-dd")
LocalDate end) {
7         log.info("分页查询, 参数: {} {}", page, pageSize);
8
9         // 调用Service
10        PageBean pageBean = empService.page(page, pageSize);
11
12        return Result.success(pageBean);
13    }

```

4. 修改Service, 把参数都传进去

```

1     @Override
2     public PageBean page(Integer page, Integer pageSize, String
name, Short gender, LocalDate begin, LocalDate end) {
3         // 设置分页参数
4         PageHelper.startPage(page, pageSize);
5
6         // 执行查询
7         List<Emp> empList = empMapper.list(name, gender, begin,
end);
8         Page<Emp> p = (Page<Emp>) empList;
9
10        // 封装PageBean对象
11        PageBean pageBean = new PageBean(p.getTotal(),
p.getResult());
12        return pageBean;
13    }

```

5. 修改Mapper, 把注解的SQL改到XML, 并使用动态SQL来判断条件

```

1 | List<Emp> list(String name, Short gender, LocalDate begin, LocalDate
end);

```

```

1 <?xml version="1.0" encoding="UTF-8" ?>
2 <!DOCTYPE mapper
3     PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
4     "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
5 <mapper namespace="com.itheima.mapper.EmpMapper">
6     <!--条件查询-->
7     <select id="list" resultType="com.itheima.pojo.Emp">
8         select *
9         from emp
10        <where>
11            <if test="name != null">name like concat('%', #{name},
'%')</if>
12            <if test="gender != null">and gender = #{gender}</if>

```

```

13         <if test="begin != null and end != null">and entrydate
14             between #{begin} and #{end}</if>
15         </where>
16         order by update_time desc
17     </select>
</mapper>

```

6. 接口测试

7. 前后端联调

发现在不输入任何条件的情况下，SQL语句仍会把name的查询条件拼接上，此时的name为空字符串 ''，要想不让SQL进行拼接，只需要修改动态SQL的判断条件

```

1      <select id="list" resultType="com.itheima.pojo.Emp">
2          select *
3          from emp
4          <where>
5              <if test="name != null and name != ''">name like
concat('%', #{name}, '%')</if>
6                  <if test="gender != null">and gender = #{gender}</if>
7                      <if test="begin != null and end != null">and entrydate
between #{begin} and #{end}</if>
8                  </where>
9                  order by update_time desc
10             </select>

```

- **删除员工**

1. 看原型

批量删除和单个删除只需要完成批量删除即可。

2. 看文档

请求路径: /emps/{ids}

请求方式: DELETE

请求参数: 路径参数, 员工id数组ids, /emps/1,2,3

响应数据: 统一响应结果Result

3. Controller, 将公共路径提升到类上

```

1  @Slf4j
2  @RestController
3  @RequestMapping("/emps")
4  public class EmpController {
5
6      @Autowired
7      private EmpService empService;
8
9      @GetMapping
10     public Result page(@RequestParam(defaultValue = "1") Integer
page,
11                         @RequestParam(defaultValue = "10") Integer
pagesize,
12                         String name, Short gender,
13                         @DateTimeFormat(pattern = "yyyy-MM-dd")
LocalDate begin,

```

```

14         @DateTimeFormat(pattern = "yyyy-MM-dd")
15     LocalDate end) {
16         log.info("分页查询, 参数: {} {} {} {} {}", page, pageSize,
17         name, gender, begin, end);
18
19         // 调用Service
20         PageBean pageBean = empService.page(page, pageSize, name,
21         gender, begin, end);
22
23         return Result.success(pageBean);
24     }
25
26     @DeleteMapping("/{ids}")
27     public Result delete(@PathVariable List<Integer> ids) {
28         log.info("批量删除操作: {}", ids);
29         empService.delete(ids);
30
31     }

```

4. Service

```

1  @Override
2  public void delete(List<Integer> ids) {
3      empMapper.delete(ids);
4  }

```

5. Mapper

```

1 <!--批量删除-->
2 <delete id="delete">
3     delete
4     from emp
5     where id in
6         <foreach collection="ids" item="id" separator="," open="("
7             close="")">
8             #{id}
9         </foreach>
</delete>

```

6. 接口测试

7. 前后端联调

- 新增员工

1. 看原型

给员工添加默认密码123456，数据库表中已经设置了默认值，不需要再额外操作。

2. 看文档

请求路径: /emps

请求方式: POST

请求参数: JSON格式, username,name,gender,image,deptId,entrydate,job

响应数据: 统一响应结果Result

3. Controller, 添加一个方法, 添加注解来接收JSON参数

```
1  @PostMapping
2  public Result add(@RequestBody Emp emp) {
3      log.info("新增员工: {}", emp);
4      empService.add(emp);
5
6      return Result.success();
7 }
```

4. Service, 添加方法, 补全基础属性

```
1  @Override
2  public void add(Emp emp) {
3      emp.setCreateTime(LocalDateTime.now());
4      emp.setUpdateTime(LocalDateTime.now());
5      empMapper.add(emp);
6 }
```

5. Mapper, 添加方法定义, 写SQL语句, 可以使用注解开发

```
1  <insert id="add">
2      insert into emp (username, name, gender, image, job,
3                      entrydate, dept_id, create_time,
4                      update_time)
5          values (#{username}, #{name}, #{gender}, #{image}, #{job},
6                  #{entrydate}, #{deptId}, #{createTime}, #
7                  {updateTime})
8  </insert>
```

插播：文件上传

- 前端程序上传三要素：

- 表单项 `input type=file`
- 表单提交方式 `post`
- 表单编码格式 `multipart/form-data`, 如果不是这个格式, 那发出的请求只有一个用户名, 不会包含文件的内容。

- 后端程序

在Spring中提供了一个API `MultipartFile` 类型来接收提交的文件, 表单项和形参的名称要保持一致, 不一致的花也可以使用 `@RequestParam` 来绑定参数和表单项数据。

新建一个UploadController类

```
1 @RestController
2 @Slf4j
3 public class UpdateController {
4
5     @PostMapping("/upload")
6     public Result upload(String username, String age, MultipartFile image) {
7         log.info("文件上传: {} {} {}", username, age, image);
8
9         return Result.success();
10    }
11 }
```

通过IDEA断点调试，发现文件有一个`part.location`属性存放了一个路径，里面有3个临时文件，分别对应传来的三个参数的值，请求响应提交执行结束后，3个临时文件会自动删除。因此除了接收文件，还要保存文件。

本地存储

在服务端接收到上传上来的文件之后，将文件存储在本地服务器磁盘中。

`MultipartFile`已经提供了现成的方法`transferTo`来转存到磁盘文件中，`MultipartFile`也提供了方法`getOriginalFilename`来获取原始的文件名

```
1 @RestController
2 @Slf4j
3 public class UpdateController {
4
5     @PostMapping("/upload")
6     public Result upload(String username, String age, MultipartFile image)
7     throws Exception {
8         log.info("文件上传: {} {} {}", username, age, image);
9
10        // 获取原始文件名
11        String originalFilename = image.getOriginalFilename();
12        // String fileName = image.getName(); // 用来获取表单项的名字
13
14        // 将文件存储在服务器磁盘目录中 C:\Users\无情\下载\day11-SpringBootWeb案例
15        // \images
16        image.transferTo(new File("C:\\\\Users\\\\无情\\\\下载\\\\day11-SpringBootWeb
17        案例\\\\images\\\\" + originalFilename));
18        return Result.success();
19    }
20 }
```

断点调试运行到`return`时就可以在磁盘中看到上传来的文件了。

如果采用原始文件名存储，可能有重名覆盖的情况，这就需要保证每个文件的名字是唯一的。可以想到用毫秒值来作为文件名，但也有重名的风险。采用`uuid`来命名。

`uuid`：通用唯一识别码，长度固定的字符串，字母长度为32位，用四个-分割，总长36位。在原始文件名中获取文件的后缀名来明确文件类型，调用`UUID.randomUUID().toString()`来生成`UUID`对象并转换为字符串，将字符串作为文件名与原始文件名中截取的后缀名进行拼接作为新的文件名。

```
1 @PostMapping("/upload")
```

```

2     public Result upload(String username, String age, MultipartFile image)
3         throws Exception {
4             log.info("文件上传: {} {} {}", username, age, image);
5
6             // 获取原始文件名
7             String originalFilename = image.getOriginalFilename();
8             // String fileName = image.getName(); // 用来获取表单项的名字
9
10            // 构造唯一的文件名(不能重复 - uuid(通用唯一识别码)) b4da284e-8e8a-4f79-
11            bfb6-c45fd6476ce9
12            int index = originalFilename.lastIndexOf(".");
13            String extname = originalFilename.substring(index);
14            String newFileName = UUID.randomUUID().toString() + extname;
15            log.info("新文件名: {}", newFileName);
16
17            // 将文件存储在服务器磁盘目录中 C:\Users\无情\下载\day11-SpringBootWeb案例
18            \images
19            image.transferTo(new File("C:\\\\Users\\\\无情\\\\下载\\\\day11-SpringBootWeb
案例\\\\images\\\\" + newFileName));
20            return Result.success();
21        }

```

仍存在的问题：上传的文件大小有可能超出限制，规定的最大文件大小为\$1048576B\$即\$1MB\$。

如果要修改大小限制，只需要在配置文件中做两个配置

```

1 # 配置单个文件最大上传大小
2 spring.servlet.multipart.max-file-size=10MB
3
4 # 配置单次请求最大上传大小(一次请求可以上传多个文件)
5 spring.servlet.multipart.max-request-size=100MB

```

- 实际上本地存储很少使用
 1. 上传的文件在前端页面中没有办法直接访问
 2. 服务器磁盘存储容量有限，不方便扩容
 3. 磁盘易损坏

插播结束：阿里云OSS

- 使用第三方服务的通用思路
 1. 注册、登录、做设置
 2. 参照官方SDK(软件开发工具包，包括依赖、代码示例)编写入门程序
 3. 集成使用
- OSS
注册、登录、实名认证后搜索OSS服务并开通，在OSS管理控制台中创建bucket，开启公共读，其他都不修改。

创建accesskey

1 | 此处应保密

在官方文档中找到上传文件流的[示例](#)，直接复制到test的软件包中，**只需要修改ossClient实例前的各个参数即可**，核心代码都不需要改动

```
1 package com.itheima;
2
3 import com.aliyun.oss.ClientException;
4 import com.aliyun.oss.OSS;
5 import com.aliyun.oss.common.auth.*;
6 import com.aliyun.oss.OSSClientBuilder;
7 import com.aliyun.oss.OSSException;
8 import com.aliyun.oss.model.PutObjectRequest;
9 import com.aliyun.oss.model.PutObjectResult;
10 import java.io.FileInputStream;
11 import java.io.InputStream;
12
13 public class Demo {
14
15     public static void main(String[] args) throws Exception {
16         // Endpoint以华东1(杭州)为例，其它Region请按实际情况填写。
17         String endpoint = "https://oss-cn-hangzhou.aliyuncs.com";
18
19         // 从环境变量中获取访问凭证。运行本代码示例之前，请确保已设置环境变量
20         // OSS_ACCESS_KEY_ID和OSS_ACCESS_KEY_SECRET。
21         // EnvironmentVariableCredentialsProvider credentialsProvider =
22         // CredentialsProviderFactory.newEnvironmentVariableCredentialsProvider();
23         String accessKeyId = "此处应保密";
24         String accessKeySecret = "此处应保密";
25
26         // 填写Bucket名称，例如examplebucket。
27         // String bucketName = "examplebucket";
28         String bucketName = "wuqing-web-tlias";
29
30         // 填写Object完整路径，完整路径中不能包含Bucket名称，例如
31         // exampledir/exampleobject.txt。
32         // 上传文件在阿里云OSS中的名字
33         // String objectName = "exampledir/exampleobject.txt";
34         String objectName = "1.gif";
35
36         // 填写本地文件的完整路径，例如D:\\localpath\\examplefile.txt。
37         // 如果未指定本地路径，则默认从示例程序所属项目对应本地路径中上传文件流。
38         // String filePath= "D:\\localpath\\examplefile.txt";
39         String filePath= "D:\\working\\WEB\\03.JS\\dream.gif";
40
41         // 创建OSSClient实例。
42         // OSS ossClient = new OSSClientBuilder().build(endpoint,
43         // credentialsProvider);
44         OSS ossClient = new OSSClientBuilder().build(endpoint,
45         accessKeyId, accessKeySecret);
46
47         try {
48             InputStream inputStream = new FileInputStream(filePath);
49             // 创建PutObjectRequest对象。
50             PutObjectRequest putObjectRequest = new
51             PutObjectRequest(bucketName, objectName, inputStream);
52             // 创建PutObject请求。
53             PutObjectResult result =
54             ossClient.putObject(putObjectRequest);
```

```
48     } catch (OSSEException oe) {
49         System.out.println("Caught an OSSEException, which means your
50         request made it to OSS, "
51             + "but was rejected with an error response for some
52             reason.");
53         System.out.println("Error Message:" + oe.getErrorMessage());
54         System.out.println("Error Code:" + oe.getErrorCode());
55         System.out.println("Request ID:" + oe.getRequestId());
56         System.out.println("Host ID:" + oe.getHostId());
57     } catch (ClientException ce) {
58         System.out.println("Caught an ClientException, which means
59         the client encountered "
60             + "a serious internal problem while trying to
61             communicate with OSS, "
62             + "such as not being able to access the network.");
63         System.out.println("Error Message:" + ce.getMessage());
64     } finally {
65         if (ossClient != null) {
66             ossClient.shutdown();
67         }
68     }
69 }
```

打开bucket的后台就可以看见上传上去的文件，OSS会为每个图片分配一个访问的url。

仿佛梦魂归帝所。
闻天语，
殷勤问我归何处。

文件名	1.gif	复制
ETag	[REDACTED]	
使用 HTTPS	<input checked="" type="checkbox"/>	
URL ?	https://[REDACTED].as.oss-cn-hangzhou.aliyuncs.com/1.gif	

访问这个URL时就会把这个图片下载下来

- OSS集成

新增员工功能基本都已经实现，只需要开发文件上传的接口。

1. 看文档

请求路径: /upload

请求方式: POST

请求参数: multipart/form-data格式, image

响应数据: 统一返回对象Result封装图片的访问路径

2. 把改好的工具类复制过来，和三层架构的解耦类似，工具类同样不建议手动new，而是交给IoC容器进行管理，使用自动注入来使用。

```
1 package com.itheima.utils;
2
3 @Component
4 public class Aliossutils {
5
6     private String endpoint = "https://oss-cn-
7         hangzhou.aliyuncs.com";
8     private String accessKeyId = "此处应保密";
9     private String accessKeySecret = "此处应保密";
10    private String bucketName = "wuqing-web-tlias";
11
12    /**
13     * 实现上传图片到OSS
14     */
15    public String upload(MultipartFile file) throws IOException {
16        // 获取上传的文件的输入流
17        InputStream inputStream = file.getInputStream();
18
19        // 避免文件覆盖
20        String originalFilename = file.getOriginalFilename();
21        String fileName = UUID.randomUUID().toString() +
22            originalFilename.substring(originalFilename.lastIndexOf(".")));
23
24        // 上传文件到 OSS
25        OSS ossClient = new OSSClientBuilder().build(endpoint,
26            accessKeyId, accessKeySecret);
27        ossClient.putObject(bucketName, fileName, inputStream);
28
29        // 文件访问路径
30        String url = endpoint.split("//")[0] + "//" + bucketName +
31            "." + endpoint.split("//")[1] + "/" + fileName;
32        // 关闭OSSClient
33        ossClient.shutdown();
34        return url; // 把上传到OSS的路径返回
35    }
36}
```

3. UploadController中使用自动注入并调用upload方法

```

1      @Autowired
2      private Aliossutils aliossutils;
3
4      @PostMapping("/upload")
5      public Result upload(MultipartFile image) throws IOException {
6          log.info("文件上传, 文件名: {}", image.getOriginalFilename());
7
8          // 调用工具类进行上传
9          String url = aliossutils.upload(image);
10         log.info("文件上传完成, 访问url为: {}", url);
11
12         return Result.success(url);
13     }

```

4. 接口测试

5. 前后端联调

修改员工

- 查询回显，点击编辑时将现在的信息显示在表单中。点击保存后保存新的数据。

1. 看文档，和根据id查询一样

/emps/{id}, GET, id, Result

2. Controller、Service、Mapper都很简单，不再写

3. 接口测试

4. 前后端联调

- 修改员工

1. 看文档

/emps, PUT, JSON格式id, username, name, gender, image, deptId, entrydate, job, 统一响应结果Result

2. Service中记得补全属性，三层的修改很简单，注意SQL语句中要对字符串做空串判断

```

1      <!--更新员工-->
2      <update id="update">
3          update emp
4          <set>
5              <if test="username != null and username != ''">username =
6                  #{username},</if>
7              <if test="name != null and name != ''">name = #{name},
8                  </if>
9              <if test="gender != null">gender = #{gender},</if>
10             <if test="image != null and image != null">image = #
11                 {image},</if>
12             <if test="deptId != null">dept_id = #{deptId},</if>
13             <if test="entrydate != null">entrydate = #{entrydate},
14                 </if>
15             <if test="job != null">job = #{job},</if>
16             <if test="updateTime != null">update_time = #
17                 {updateTime}</if>
18             </set>
19             where id = #{id};
20         </update>

```

3. 接口调试

4. 前后端联调

参数配置化

- 硬编码问题：阿里云工具类中的参数都硬编码在类中，要进行修改的话需要重新编译再运行。一个真实的项目中有很多类，要修改一个类中的参数值就需要定位到类再修改+编译+运行。不便于维护和管理

代码会有很多，但配置文件只有一个！就像Maven的配置文件 `pom.xml`一样，SpringBoot的配置文件也只有一个，叫 `application.properties`。

可以将工具类中的参数定义在配置文件中，形式为 键=值。key可以自己定义，但要尽量有一定的含义。

然后需要将配置文件中的值赋给类中的各个属性，SpringBoot的配置文件当然要用SpringBoot的方式来处理，一个注解 `@value("${配置文件中的key}")`。为需要这些值的成员变量添加 `@value` 注解并指定配置文件中的键，SpringBoot在运行时就会为变量自动赋值。

```
1 # 自定义OSS配置信息，键的名字可以自定义
2 aliyun.oss.endpoint=https://oss-cn-hangzhou.aliyuncs.com
3 aliyun.oss.accessKeyId=此处应保密
4 aliyun.oss.accessKeySecret=此处应保密
5 aliyun.oss.bucketName=wuqing-web-tlias"
```

为工具类中的变量添加注解

```
1 @Component
2 public class Aliossutils {
3
4     @Value("${aliyun.oss.endpoint}")
5     private String endpoint;
6     @Value("${aliyun.oss.accessKeyId}")
7     private String accessKeyId;
8     @Value("${aliyun.oss.accessKeySecret}")
9     private String accessKeySecret;
10    @Value("${aliyun.oss.bucketName}")
11    private String bucketName;
12
13    /**
14     * 实现上传图片到OSS
15     */
16    public String upload(MultipartFile file) throws IOException {
17        // 获取上传的文件的输入流
18        InputStream inputStream = file.getInputStream();
19
20        // 避免文件覆盖
21        String originalFilename = file.getOriginalFilename();
22        String fileName = UUID.randomUUID().toString() +
23            originalFilename.substring(originalFilename.lastIndexOf("."));
24
25        //上传文件到 OSS
26        OSS ossClient = new OSSClientBuilder().build(endpoint, accessKeyId,
27            accessKeySecret);
28        ossClient.putObject(bucketName, fileName, inputStream);
29
30        //文件访问路径
31    }
32}
```

```
29         String url = endpoint.split("//")[0] + "//" + bucketName + "." +
30         endpoint.split("//")[1] + "/" + fileName;
31         // 关闭ossClient
32         ossClient.shutdown();
33         return url;// 把上传到oss的路径返回
34     }
35 }
```

yml配置文件

SpringBoot提供了多种配置文件类型，自带的`.properties`，还有`.yml .yaml`，这两种的配置形式是相同的，只是后缀名不同。但是三类配置文件的名字都是`application`。`yml`的配置格式为`key: value`，且`key`按照树形层级来写，如下：

```
1 server:
2     port: 9000
```

- 常见格式对比

1. XML：标签太多，过于臃肿
2. properties：层级结构不清晰
3. yml/yaml：简洁、以数据为中心

- 主流的写法都用`.yml`文件

yml 属性值前边必须有空格，作为分隔符

使用缩进表示层级关系，缩进时不允许使用tab，只能用空格（IDEA会自动将tab转为空格）

缩进数不重要，只要同级对齐即可

注释用`#`

- 常见数据格式

- 对象/Map集合

```
1 user:
2     name: zhangsan
3     age: 18
4     password: 123456
```

- 数组/List/Set集合

```
1 hobby:
2     - java
3     - game
4     - sport
```

- 将`properties`的配置信息都写到`yml`中

```
1 spring:
2     # 数据库连接信息
3     datasource:
4         driver-class-name: com.mysql.cj.jdbc.Driver
5         url: jdbc:mysql://localhost:3306/tlias
6         username: root
```

```

7     password: L=104-02
8     servlet:
9       multipart:
10      # 配置单个文件最大上传大小
11      max-file-size: 10MB
12      # 配置单次请求最大上传大小 (一次请求可以上传多个文件)
13      max-request-size: 100MB
14
15
16 # MyBatis配置
17 mybatis:
18   configuration:
19     log-impl: org.apache.ibatis.logging.stdout.StdoutImpl
20     map-underscore-to-camel-case: true
21
22
23 # 自定义OSS配置信息
24 aliyun:
25   oss:
26     endpoint: https://oss-cn-hangzhou.aliyuncs.com
27     accessKeyId: 此处应保密
28     accessKeySecret: 此处应保密
29     bucketName: wuqing-web-tlias
30

```

@ConfigurationProperties

之前已经将参数写到了配置文件并通过注解对成员变量进行了注入，如果每个成员变量都需要注入，会很繁琐。

是否可以不用在每个成员变量上都加注解也能将配置文件的值注入到属性中？

条件：

1. 配置文件的key和属性名必须一致，因为声明的是private，需要再加上Lombok注解 @Data
2. 将类交给IoC容器管理，@Component
3. 属性名和配置项的后缀保持一致，配置项还有一个前缀 aliyun:oss:，所以在类上添加注解 @ConfigurationProperties(prefix = "aliyun.oss")，此时就代表要将aliyun.oss下的四个子配置对应的值自动注入到对象中的属性中。

将四个成员属性值拿出来单独写一个类

```

1 @Data
2 @Component
3 @ConfigurationProperties(prefix = "aliyun.oss")
4 public class AliOSSProperties {
5   private String endpoint;
6   private String accessKeyId;
7   private String accessKeySecret;
8   private String bucketName;
9 }
10

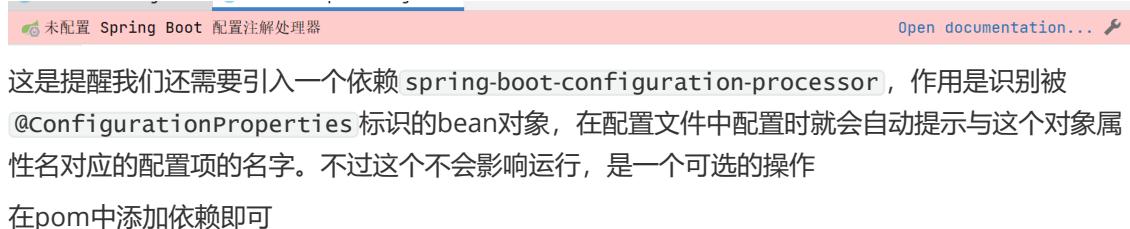
```

在工具类中通过自动注入使用这个类，在upload方法中获取四个属性的值

```
1 package com.itheima.utils;
2
3 import com.aliyun.oss.OSS;
4 import com.aliyun.oss.OSSClientBuilder;
5 import lombok.Data;
6 import org.springframework.beans.factory.annotation.Autowired;
7 import org.springframework.beans.factory.annotation.Value;
8 import org.springframework.stereotype.Component;
9 import org.springframework.web.multipart.MultipartFile;
10 import java.io.*;
11 import java.util.UUID;
12
13 /**
14 * 阿里云 oss 工具类
15 */
16 @Component
17 @Data
18
19 public class Aliossutils {
20
21     @Autowired
22     private AliOSSProperties alioSSProperties;
23
24     // @Value("${aliyun.oss.endpoint}")
25     // private String endpoint;
26     // @Value("${aliyun.oss.accessKeyId}")
27     // private String accessKeyId;
28     // @Value("${aliyun.oss.accessKeySecret}")
29     // private String accessKeySecret;
30     // @Value("${aliyun.oss.bucketName}")
31     // private String bucketName;
32
33     /**
34      * 实现上传图片到oss
35      */
36     public String upload(MultipartFile file) throws IOException {
37         // 获取阿里云OSS参数
38         String endpoint = alioSSProperties.getEndpoint();
39         String accessKeyId = alioSSProperties.getAccessKeyId();
40         String accessKeySecret = alioSSProperties.getAccessKeySecret();
41         String bucketName = alioSSProperties.getBucketName();
42
43         // 获取上传的文件的输入流
44         InputStream inputStream = file.getInputStream();
45
46         // 避免文件覆盖
47         String originalFilename = file.getOriginalFilename();
48         String fileName = UUID.randomUUID().toString() +
originalFilename.substring(originalFilename.lastIndexOf(".")));
49
50         //上传文件到 oss
51         OSS ossClient = new OSSClientBuilder().build(endpoint,
accessKeyId, accessKeySecret);
52         ossClient.putObject(bucketName, fileName, inputStream);
53
54         //文件访问路径
```

```
55     String url = endpoint.split("//")[0] + "//" + bucketName + "." +
56     endpoint.split("//")[1] + "/" + fileName;
57     // 关闭OSSClient
58     ossClient.shutdown();
59     return url; // 把上传到OSS的路径返回
60 }
61 }
62 }
```

4. 在 AliOSSProperties 属性类中加上注解后出现了一个警告



```
1 <dependency>
2   <groupId>org.springframework.boot</groupId>
3   <artifactId>spring-boot-configuration-processor</artifactId>
4 </dependency>
```

此时在配置文件的提示中驼峰命名已经转化为了横杠分隔，两种都可以正常运行。

登录功能

根据username和password在表中查询，查不到说明没有用户或输入错误。

1. 看文档

/login POST JSON格式 username password 统一响应结果Result封装JWT令牌（暂不考虑）

2. 需要定义新的Controller，叫LoginController，Service和Mapper都使用emp的即可，接收参数的实体类用Emp。

```
1 @Slf4j
2 @RestController
3 public class LoginController {
4
5     @Autowired
6     private EmpService empService;
7
8     @PostMapping("/login")
9     public Result login(@RequestBody Emp emp) {
10         log.info("员工登录: {}", emp);
11         Emp e = empService.login(emp);
12
13         return e != null ? Result.success() : Result.error("用户名或密码错
误");
14     }
15 }
```

3. ServiceImpl

```
1  @Override  
2  public Emp Login(Emp emp) {  
3      return empMapper.getByUsernameAndPassword(emp);  
4 }
```

4. Mapper, Mapper中的方法名不建议使用login, login是业务的方法, Mapper接口用业务方法名并不合适。

```
1  @Select("select * from emp where username = #{username} and password  
= #{password};")  
2  Emp getByUsernameAndPassword(Emp emp);
```

5. 接口测试

6. 前后端联调

登录校验

在未登录情况下也可以直接访问部门管理、员工管理的数据。所以要完成登录校验功能。

服务器端接收到浏览器发送的请求后, 先进行校验, 校验一下是否用户登录, 如果登录了那就可以直接执行业务操作。校验时发现没有登录, 就不允许他执行相关的业务操作, 直接给前端响应错误并跳转到登录页面。

HTTP协议是无状态协议, 每次请求都是独立的。下一次请求不会携带上一次请求的数据。这意味着我们访问了登录接口, 接下来在执行其他业务操作时, 服务器也不知道这个员工到底登录了没有。要想判断员工是否已经登录, 要在登录之后存储一个标记, 登录成功后存储一个登陆成功的标记。接下来在每个方法之前做个判断, 如果登录了就正常执行业务操作, 如果没登录就直接返回错误信息, 前端拿到信息后会自动跳转到登录页面。所有的方法都需要进行判断, 那代码重复性就会很高而且很繁琐。为了简化操作, 可以使用统一拦截技术, 拦截浏览器发送过来的所有请求, 拦截到请求后就可以进行校验员工是否登录, 此时就可以获取之前存入的登录标记, 如果获取到了标记且标记没有问题说明员工已经登录, 就放行请求正常访问业务接口。如果没有获取到标记或者获取到的标记有问题, 直接给前端响应错误信息。

完成校验操作主要有两部分:

1. 登录标记, 登陆成功后每一次请求中, 都可以获取到该标记。会话技术
2. 统一拦截, Servlet规范中的过滤器Filter, Spring规范的拦截器Interceptor

11. 会话技术

会话: 浏览器与服务器之间的一次连接。用户第一次访问服务器时这个会话就建立了。直到有任何一方断开连接, 会话才结束。**一次会话可以包含多次请求和响应。**

会话跟踪: 维护浏览器状态的方法, 服务器要识别多次请求是否来自于同一浏览器, 以便在**同一次会话的多次请求间共享数据。**

如何实现会话跟踪? HTTP是无状态的, 没有办法仅通过HTTP协议来区分两次请求是否来自同一浏览器和同一会话。

- 会话跟踪方案

1. 客户端会话跟踪技术: Cookie
2. 服务端会话跟踪技术: Session
3. 令牌技术

客户端会话跟踪技术：Cookie

Cookie存储在客户端。使用Cookie跟踪会话，在第一次请求服务器时设置一个Cookie，如请求登录接口，接口执行完成后可以设置一个Cookie，在Cookie中存储相关数据信息，如username id，服务器端响应数据时会**自动**将Cookie响应给浏览器，浏览器接收到Cookie后**自动**将Cookie的值存储在本地，接下来的每次请求都会将本地存储的Cookie**自动**携带到服务端，在服务端可以获取到Cookie的值，判断这个Cookie值是否存在。如果不存在这个Cookie，说明客户端以前没有访问登录接口。如果存在，说明客户端之前已经登录完成了。这样就可以基于Cookie在同一会话的不同请求之间共享数据。

为什么都是自动进行的？Cookie是HTTP协议中支持的技术，各大浏览器厂商都支持了这一标准，HTTP中提供了**响应头** `set-Cookie` 和**请求头** `Cookie`。服务器给前端响应数据时，直接通过响应头携带数据。浏览器会自动解析响应头，将数据存储在浏览器本地。在后续请求中，将存储的Cookie值在请求头当中携带到服务器。在Tomcat这类服务器中，也提供了Cookie操作的API，可以很方便的设置Cookie和获取Cookie。

缺点：

- 移动端无法使用Cookie
- 不安全，用户可以禁用Cookie
- Cookie不能跨域

跨域：前后端分离开发模式中，前端和后端都是分开部署的。前端在一台服务器，后端在另一台服务器。浏览器访问前端的页面，执行登录操作时又要访问后端的服务器。访问前端和后端的两次请求IP地址不同，端口也不一样，就称为跨域请求。从三个维度区分跨域请求：**协议、IP、端口，只要有任何一个不同就是跨域。**

服务器会话跟踪技术：Session

Session存储在服务器端，底层基于Cookie实现。浏览器第一次请求服务器时，可以直接在服务器中获取到会话对象Session，第一次请求时Session不存在，服务器会自动创建一个会话对象Session。每个Session都有id，服务器给浏览器响应数据时，会将Session的id通过Cookie响应给浏览器。就是在响应头中增加了 `set-Cookie` 响应头，这个响应头对应的值就是Cookie，名字固定为 `JSESSIONID` 代表服务器端Session对象的id，浏览器接收到后会**自动**将值存储在本地，后续请求中都会将Cookie获取出来携带到服务端。服务器拿到Session的id就会从众多的Session中找到当前请求对应的Session，这样就可以通过Session在同一会话的不同请求之间共享数据

缺点：

1. 现在的项目都是集群部署，通过前置服务器进行请求分发，若第一次请求了服务器1，浏览器存储了Sessionid=1，第二次请求了服务器2，但服务器2中找不到Session对象。集群环境下Session也就废掉了
2. Cookie的缺点

令牌技术

令牌就是用户身份的标识，一个字符串，存储在客户端。在请求登录接口时，如果登陆成功就可以生成一个令牌作为用户的合法身份凭证。响应数据时将令牌响应给前端，前端接收到令牌后就需要存储起来。可以在Cookie中，也可以在其他位置。每次请求都需要将令牌携带到服务端。服务端需要校验令牌的有效性，若有效说明已经登录，无效说明没有登录。共享数据存储在令牌中就可以了。

缺点

- 需要自己实现

JWT令牌

JSON Web Token，定义了一种简洁的、自包含的格式，用来以JSON格式安全的传输信息。因为数字签名的存在，这些信息是可靠的。

简洁：就是一个字符串

自包含：可以在令牌中存储自定义内容

整个令牌由三个部分组成，由两个`.`来分割。

1. 第一部分Header，记录令牌类型、签名算法、签名密钥。如`{"alg": "HS256", "type": "JWT"}`，生成令牌时要对JSON进行base64编码。
2. 第二部分Payload（有效载荷），携带自定义信息、默认信息等。如`{"id": "1", "username": "Tom"}`
3. 第三部分Signature（签名），防止Token被篡改、确保安全。基于指定的签名算法融入header, payload并加入指定密钥（第一部分的密钥），通过签名算法计算得到。

典型应用：登录认证，两步：登录成功生成令牌响应到前端，后续每次请求验证令牌是否有效。

JWT生成和校验

需要引入依赖`jjwt`，生成和校验都需要用到依赖提供的工具类`jwts`

```
1 <dependency>
2   <groupId>io.jsonwebtoken</groupId>
3   <artifactId>jjwt</artifactId>
4   <version>0.9.1</version>
5 </dependency>
```

定义令牌时要设置签名算法（可看官网），设置自定义内容，设置令牌有效期。

```
1 @Test
2 public void testGenJwt() {
3     Map<String, Object> claims = new HashMap<>();
4     claims.put("id", 1);
5     claims.put("name", "Tom");
6
7     String jwt = Jwts.builder()
8         .signWith(SignatureAlgorithm.HS256, "itheima") // 设置签名算法
9         .setClaims(claims) // 设置自定义内容
10        .setExpiration(new Date(System.currentTimeMillis() + 3600 *
11000)) // 设置有效期为1h
12        .compact();
13
14 }
```

解析JWT令牌，使用`Jwts`中的`parser()`方法，指定签名密钥`.setSigningKey()`，调用`.parseClaimsJws`解析令牌，最后通过`getBody()`拿到自定义内容。

```
1  @Test
2  public void testParseJwt() {
3      Claims claims = Jwts.parser()
4          .setSigningKey("itheima")
5
6      .parseClaimsJws("eyJhbGciOiJIUzI1NiJ9eyJubW1lIjoivG9tIiwiaQiojEsImV4CCl6MTc
7      wODE4MTA3NX0.UZXL9w6itfdkBv61GkVdmDw38DUvot1m31sg5ScVqRU")
8          .getBody();
9
10     System.out.println(claims);
11 }
```

**登录后下发令牌 (注意：令牌存储由前端负责，案例中存放
在本地存储空间)**

- 生成：登录成功后生成令牌并返回给前端。
 - 校验：请求到达服务器端后，对令牌进行统一拦截、校验。

1. 引入工具类，登录完成后调用工具类生成令牌并返回

```
1 package com.itheima.utils;
2
3
4 import io.jsonwebtoken.Claims;
5 import io.jsonwebtoken.JwtBuilder;
6 import io.jsonwebtoken.SignatureAlgorithm;
7
8 import java.util.Date;
9 import java.util.HashMap;
10 import java.util.Map;
11
12 public class JwtUtils {
13
14     private static String signKey = "itheima";
15     private static Long expire = 43200000L;
16
17     /**
18      * 生成JWT令牌
19      *
20      * @param claims JWT第二部分存储的内容
21      * @return JWT字符串
22      */
23     public static String generateJwt(Map<String, Object> claims) {
24         String jwt = JwtBuilder.  
            .addClaims(claims) // 设置自定义内容
25             .signWith(SignatureAlgorithm.HS256, signKey) // 设置签名
26             算法和第三部分进行签名时的密钥
27             .setExpiration(new Date(System.currentTimeMillis() +
28             expire)) // 设置有效期为1h
29             .compact();
30
31         return jwt;
32     }
33
34     /**
35      * 解析JWT令牌
36      *
```

```

35     * @param jwt
36     * @return
37     */
38     public static Claims parseJWT(String jwt) {
39         Claims claims = Jwts.parser()
40             .setSigningKey(signKey)
41             .parseClaimsJws(jwt)
42             .getBody();
43
44         return claims;
45     }
46 }
```

2. 修改LoginController，对令牌进行下发

```

1  @Slf4j
2  @RestController
3  public class LoginController {
4
5      @Autowired
6      private EmpService empService;
7
8      @PostMapping("/login")
9      public Result login(@RequestBody Emp emp) {
10         log.info("员工登录: {}", emp);
11         Emp e = empService.login(emp);
12
13         // 登录成功，生成令牌并下发
14         if (e != null) {
15             Map<String, Object> claims = new HashMap<>();
16             claims.put("id", e.getId());
17             claims.put("name", e.getName());
18             claims.put("username", e.getUsername());
19
20             String jwt = JwtUtils.generateJwt(claims); // 包含了当前登录的
21             员工信息
22             log.info(jwt);
23             return Result.success(jwt);
24         }
25
26         // 登录失败，返回错误信息
27         return Result.error("用户名或密码错误");
28     }
}
```

3. 接口测试

4. 前后端联调，案例中将令牌存到了本地存储空间

登录后再随意发出一个请求，查看请求的请求头中的Token。

12. 统一拦截，过滤器Filter

JavaWeb三大组件 (Servlet、Filter、Listener) 之一

可以把对资源的请求拦截下来，实现特殊的功能

一般完成一些通用操作，如登录校验、统一编码处理、敏感字符处理等。

过滤器Filter快速入门

定义Filter：定义一个类，实现Filter接口，重写所有方法

配置Filter：Filter类上加 @WebFilter(urlPatterns = ?) 注解，配置拦截资源的路径。引导类上加 @ServletComponentScan 开启Servlet组件支持。Filter是JavaWeb三大组件，不是SpringBoot提供的，要想在SpringBoot中使用JavaWeb三大组件，必须加注解。

先定义一个Filter实现类，类上加注解 @WebFilter(urlPatterns = "/*")，/*表示拦截所有请求

注意：实现接口方法时实际上只需要实现 doFilter 方法， init destroy 方法都有默认实现且使用较少。

```
1 package com.itheima.filter;
2
3 import jakarta.servlet.*;
4 import jakarta.servlet.annotation.WebFilter;
5 import org.springframework.context.annotation.ComponentScan;
6
7 import java.io.IOException;
8
9 @WebFilter(urlPatterns = "/*") // 拦截所有请求
10 public class DemoFilter implements Filter {
11     @Override // 初始化方法，只调用一次
12     public void init(FilterConfig filterConfig) throws ServletException {
13         System.out.println("init初始化方法执行了");
14         Filter.super.init(filterConfig);
15     }
16
17     @Override // 拦截到请求就调用，会调用多次
18     public void doFilter(ServletRequest servletRequest, ServletResponse
servletResponse, FilterChain filterChain) throws IOException,
ServletException {
19         System.out.println("拦截到了请求");
20     }
21
22     @Override // 销毁方法，只调用一次
23     public void destroy() {
24         System.out.println("destroy初始化方法执行了");
25         Filter.super.destroy();
26     }
27 }
28 }
```

在启动类上加注解 @ServletComponentScan，开启Servlet组件支持

```
1 @ServletComponentScan
2 @SpringBootApplication
3 public class TliaswebManagementApplication {
4
5     public static void main(String[] args) {
6         SpringApplication.run(TliaswebManagementApplication.class, args);
7     }
8
9 }
```

此时打开前端页面发出请求，控制台就会打印拦截到请求，而且页面中也不会有数据。这是因为Filter中拦截到请求后只输出了一句话，并没有把请求放行，所以资源不会被访问到。所以还需要最重要的操作，就是放行。

打开放行

```
1  @Override // 拦截到请求就调用，会调用多次
2  public void doFilter(ServletRequest servletRequest, ServletResponse
servletResponse, FilterChain filterChain) throws IOException,
ServletException {
3      System.out.println("拦截到了请求");
4
5      // 放行
6      filterChain.doFilter(servletRequest, servletResponse);
7 }
```

Filter执行流程

没有放行操作，是访问不到后面的web资源的。放行之前可以执行一段放行前的逻辑，放行就是让其访问对应的web资源，访问完web资源后还会回到过滤器，有需要还可以执行放行后的逻辑，写在doFilter之后。

Filter拦截路径

可以根据需求，配置不同的拦截资源路径

- 拦截具体路径，如 /login，只有访问/login路径时，才会被拦截
- 目录拦截，如 /emps/*，访问/emps下的所有资源，都会被拦截
- 拦截所有， /*，访问所有资源，都会被拦截

过滤器链

一个web应用中，可以配置多个过滤器，形成了过滤器链。

第一个放行，再执行第二个，...，最后一个放行后才会访问web资源。按照执行流程，访问完资源后还会回到过滤器中执行放行后逻辑，**执行放行后逻辑的顺序是反着的！！！**

filterChain 的 doFilter 方法实际上就是在过滤器链上放行，最后一个过滤器放行后会访问对应的web资源。

注意放行前和放行后的执行顺序：

ABC 拦截到了请求...放行前逻辑

Demo 拦截到了请求...放行前逻辑

Demo 拦截到了请求...放行后逻辑

ABC 拦截到了请求...放行后逻辑

为什么是ABC先执行？和类名有关，注解配置的Filter执行顺序是类名的字典序确定的。

过滤器链执行流程：过滤器1放行前逻辑，放行到过滤器2，过滤器2放行前逻辑...过滤器n放行前逻辑 过滤器n放行，访问web资源，过滤器n放行后逻辑，过滤器n-1放行后逻辑...过滤器2放行后逻辑，过滤器1放行后逻辑

案例实现登录校验过滤器

所有的请求是否都需要校验令牌？登录操作不需要校验令牌。

什么情况才可以放行，执行业务操作？有令牌且令牌合法，否则返回未登录错误结果

- 流程：通过url路径判断当前请求是否是登录请求。是登录请求就执行放行执行登录操作，否则获取请求头Token，判断令牌是否存在，令牌合法则放行，令牌不存在或不合法，返回未登录错误。

创建一个LoginCheckFilter类作为登录校验的过滤器，将形参属性的接口类型强转为子级接口类型来使用，方便使用`getRequestURL`方法获取请求路径，使用`getHeader("token")`获取请求头中的参数（**token是文档中指定的名称**）。令牌不合法时需要响应给前端一个`Result.error`，最终要响应回去的是JSON格式的数据，之前在Controller中操作的，而`@RestController`会自动将方法返回值转为JSON，现在需要手动将对象转为JSON，可以使用阿里的工具包`fastjson`，使用工具包提供的`JSONObject.toJSONString(s)`将对象转为JSON格式字符串。获取请求数据是通过`request`获取的，响应数据就可以通过`response`来响应。解析令牌不报错说明令牌解析成功，只要报错说明解析失败，可以使用`try...catch`，解析失败和上面一样返回错误结果`Result.error`。

```
1 package com.itheima.filter;
2
3 import com.alibaba.fastjson.JSONObject;
4 import com.itheima.pojo.Result;
5 import com.itheima.utils.JwtUtils;
6 import io.jsonwebtoken.Claims;
7 import jakarta.servlet.*;
8 import jakarta.servlet.annotation.WebFilter;
9 import jakarta.servlet.http.HttpServletRequest;
10 import jakarta.servlet.http.HttpServletResponse;
11 import lombok.extern.slf4j.Slf4j;
12 import org.springframework.util.StringUtils;
13
14 import java.io.IOException;
15
16 @Slf4j
17 @WebFilter(urlPatterns = "/*")
18 public class LoginCheckFilter implements Filter {
19     @Override
20     public void doFilter(ServletRequest servletRequest, ServletResponse
21     servletResponse, FilterChain filterChain) throws IOException,
22     ServletException {
23         // 1. 获取URL
24         HttpServletRequest request = (HttpServletRequest) servletRequest;
25         HttpServletResponse response = (HttpServletResponse)
26         servletResponse;
27         String requestURL = request.getRequestURL().toString();
28         log.info("请求的url: {}", requestURL);
29
30         // 2. 判断url是否含有login，是登录则放行
31         if (requestURL.contains("login")) {
32             log.info("是登录操作: 放行-----");
33             filterChain.doFilter(servletRequest, servletResponse);
34             return;
35         }
36
37         // 3. 获取Token(令牌)
38         String jwt = request.getHeader("token");
```

```

37     // 4. 判断令牌合法性
38     if (!StringUtil.hasLength(jwt)) {
39         log.info("请求头Token为空, 返回未登录信息");
40         Result error = Result.error("NOT_LOGIN");
41         // 手动转成JSON格式, 使用阿里fastjson
42         String notLogin = JSONObject.toJSONString(error);
43         // 响应未登录结果给浏览器
44         response.getWriter().write(notLogin);
45         return;
46     }
47
48     // 5. 解析Token, 失败返回结果
49     try {
50         JwtUtils.parseJWT(jwt);
51     } catch (Exception e) {
52         e.printStackTrace();
53         // 解析失败
54         log.info("解析令牌失败, 返回未登录的错误信息");
55         Result error = Result.error("NOT_LOGIN");
56         // 手动转成JSON格式, 使用阿里fastjson
57         String notLogin = JSONObject.toJSONString(error);
58         // 响应未登录结果给浏览器
59         response.getWriter().write(notLogin);
60         return;
61     }
62     // 6. 放行
63     log.info("令牌合法, 放行");
64     filterChain.doFilter(servletRequest, servletResponse);
65 }
66
67 }
```

接口测试，发一个查询部门请求，要注意在 Header 中添加参数，参数名叫token，参数值为jwt令牌。

前后端联调

13. 拦截器Interceptor快速入门

动态拦截方法调用的机制，类似于过滤器。Spring框架提供，用来动态拦截控制器方法的执行。

拦截请求，在指定方法调用前后根据需要执行设定的代码。

定义拦截器，实现HandlerInterceptor接口，重写所有方法。**注意：三个方法都有默认实现**

```

1 package com.itheima.interceptor;
2
3 import jakarta.servlet.http.HttpServletRequest;
4 import jakarta.servlet.http.HttpServletResponse;
5 import org.springframework.stereotype.Component;
6 import org.springframework.web.servlet.HandlerInterceptor;
7 import org.springframework.web.servlet.ModelAndView;
8
9
10 @Component
11 public class LoginCheckInterceptor implements HandlerInterceptor {
12     @Override // 目标资源方法运行前运行, 返回true表示放行, false表示拦截
```

```

13     public boolean preHandle(HttpServletRequest request, HttpServletResponse
14         response, Object handler) throws Exception {
15         System.out.println("preHandle 运行了-----");
16         return true;
17     }
18
19     @Override // 目标资源方法运行后运行
20     public void postHandle(HttpServletRequest request, HttpServletResponse
21         response, Object handler, ModelAndView modelAndView) throws Exception {
22         System.out.println("postHandle 运行了-----");
23         HandlerInterceptor.super.postHandle(request, response, handler,
24         modelAndView);
25     }
26
27     @Override // 视图渲染完毕后运行, 最后运行这个方法
28     public void afterCompletion(HttpServletRequest request,
29         HttpServletResponse response, Object handler, Exception ex) throws Exception
30     {
31         System.out.println("afterCompletion 运行了-----");
32         HandlerInterceptor.super.afterCompletion(request, response, handler,
33         ex);
34     }
35 }
```

注册拦截器，需要定义一个配置类，实现接口WebMvcConfigurer，在类上加上注解@Configuration 标识当前类是Spring的配置类，重写方法addInterceptors并指定拦截器的拦截路径。

注意：过滤器Filter中拦截全部路径使用的通配符为/*，拦截器Interceptor中拦截全部路径要使用/**

```

1 package com.itheima.config;
2
3 import com.itheima.interceptor.LoginCheckInterceptor;
4 import org.springframework.beans.factory.annotation.Autowired;
5 import org.springframework.context.annotation.Configuration;
6 import
7 org.springframework.web.servlet.config.annotation.InterceptorRegistry;
8 import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;
9
10 @Configuration // 配置类
11 public class webConfig implements WebMvcConfigurer {
12
13     @Autowired
14     LoginCheckInterceptor loginCheckInterceptor;
15
16     @Override
17     public void addInterceptors(InterceptorRegistry registry) {
18
19         registry.addInterceptor(loginCheckInterceptor).addPathPatterns("/**");
20     }
21 }
```

通过接口调试，preHandle中一直返回的是true，会直接放行，将返回值改为false，控制台中可以发现只执行了preHandle并没有放行到Controller。

拦截器拦截路径

使用 `addPathPatterns` 设置要拦截的资源，使用 `excludePathPatterns` 设置不需要拦截的资源
如设置拦截所有但不拦截登录：

```
1  @Override
2  public void addInterceptors(InterceptorRegistry registry) {
3
4      registry.addInterceptor(loginCheckInterceptor).addPathPatterns("/**").excludePathPatterns("/login");
    }
```

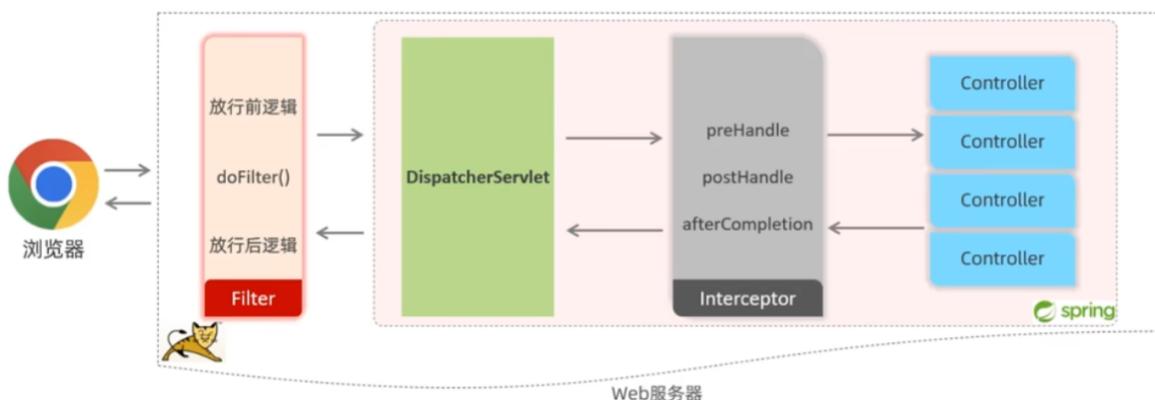
- 常见拦截路径

- /*, 一级路径, 能匹配/depts /emps /login, 不能匹配/depts/1
- /**, 任意级路径, 能匹配/depts /depts/1 /depts/1/2
- /depts/*, /depts下的一级路径, 能匹配/depts/1, 不能匹配/depts/1/2 /depts
- /depts/**, /depts下的任意级路径, 能匹配/depts /depts/1 /depts/1/2, 不能匹配/emps/1

使用接口进行调试，此时就不会拦截登录请求了。再发送一些别的请求，可以在控制台中看到 `preHandle` `postHandle` `afterCompletion`都执行了，说明请求都会走拦截器，不过现在`preHandle`返回值一直是true，所以都被放行了。

拦截器执行流程

假设过滤器和拦截器都存在，浏览器访问web应用时，过滤器会拦截请求，执行放行前逻辑+放行，放行后进入Spring环境中想要访问Controller接口方法。但是之前讲过Tomcat不识别Controller程序但是识别Servlet程序，而在Spring环境中提供了一个核心的前端控制器 `DispatcherServlet`，请求会先进入 `DispatcherServlet`，由 `DispatcherServlet` 将请求转给Controller执行对应的接口方法，但现在又定义了拦截器所以会先被拦截器拦截住，拦截器拦住请求进行一些处理，在进入Controller前会执行 `preHandle` 方法，如果返回值为true代表放行允许进入Controller，返回值为false代表不会放行进入 Controller，Controller中的方法执行完毕后会回来执行 `postHandle` 方法和 `afterCompletion` 方法，然后返回给 `DispatcherServlet`，最后执行过滤器的放行后逻辑，最终给浏览器响应数据。



检验一下：打开一个过滤器DemoFilter，使用apifox发一个查询部门请求，查看控制台信息

Demo 拦截到了请求 ... 放行前逻辑
`preHandle` 运行了——

```

⇒ Preparing: select * from dept
⇒ Parameters:
⇐   Columns: id, name, create_time, update_time
⇐     Row: 1, 学业部, 2024-02-16 12:31:03, 2024-02-17 00:08:52
⇐     Row: 2, 教研部, 2024-02-16 12:31:03, 2024-02-16 12:31:03
⇐     Row: 3, 咨询部, 2024-02-16 12:31:03, 2024-02-16 12:31:03
⇐     Row: 6, 就业部, 2024-02-16 23:26:52, 2024-02-16 23:26:52
⇐     Row: 8, 销售部, 2024-02-16 23:34:47, 2024-02-16 23:34:47
⇐   Total: 5
Closing non transactional SqlSession [org.apache.ibatis.session.defaultsession]
postHandle 运行了——
afterCompletion 运行了——
Demo 拦截到了请求 ... 放行后逻辑

```

可以看到是过滤器先执行，然后是拦截器，然后执行业务操作，再执行拦截器中的剩下两个方法，最后执行过滤器的放行后逻辑。

- Filter和Interceptor的不同
 - 接口规范不同，过滤器要实现Filter接口，拦截器要实现HandlerInterceptor接口
 - 拦截范围不同，过滤器会拦截所有资源，Interceptor只会拦截Spring环境中的资源

案例实现登录校验拦截器

逻辑和步骤和过滤器完全一致，只需要把方案由过滤器换成拦截器就可以了。

把过滤器中的复制过来改一改

```

1 package com.itheima.interceptor;
2
3 import com.alibaba.fastjson.JSONObject;
4 import com.itheima.pojo.Result;
5 import com.itheima.utils.JwtUtils;
6 import jakarta.servlet.http.HttpServletRequest;
7 import jakarta.servlet.http.HttpServletResponse;
8 import lombok.extern.slf4j.Slf4j;
9 import org.springframework.stereotype.Component;
10 import org.springframework.util.StringUtils;
11 import org.springframework.web.servlet.HandlerInterceptor;
12 import org.springframework.web.servlet.ModelAndView;
13
14 @Slf4j
15 @Component
16 public class LoginCheckInterceptor implements HandlerInterceptor {
17     @Override // 目标资源方法运行前运行，返回true表示放行，false表示拦截
18     public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {
19         String requestURL = request.getRequestURL().toString();
20         log.info("请求的url: {}", requestURL);
21
22         // 2. 判断url是否含有login，是登录则放行
23         if (requestURL.contains("login")) {
24             log.info("是登录操作：放行-----");
25             return true; // 返回true就是放行
26         }
27     }
28 }

```

```

27     // 3. 获取Token (令牌)
28     String jwt = request.getHeader("token");
29
30
31     // 4. 判断令牌合法性
32     if (!StringUtil.hasLength(jwt)) {
33         log.info("请求头Token为空, 返回未登录信息");
34         Result error = Result.error("NOT_LOGIN");
35         // 手动转成JSON格式, 使用阿里fastjson
36         String notLogin = JSONObject.toJSONString(error);
37         // 响应未登录结果给浏览器
38         response.getWriter().write(notLogin);
39         return false; // 返回false就是不放行
40     }
41
42     // 5. 解析Token, 失败返回结果
43     try {
44         JwtUtils.parseJWT(jwt);
45     } catch (Exception e) {
46         e.printStackTrace();
47         // 解析失败
48         log.info("解析令牌失败, 返回未登录的错误信息");
49         Result error = Result.error("NOT_LOGIN");
50         // 手动转成JSON格式, 使用阿里fastjson
51         String notLogin = JSONObject.toJSONString(error);
52         // 响应未登录结果给浏览器
53         response.getWriter().write(notLogin);
54         return false; // 同上 不放行
55     }
56     // 6. 放行
57     log.info("令牌合法, 放行");
58     return true;
59 }
60
61     @Override // 目标资源方法运行后运行
62     public void postHandle(HttpServletRequest request, HttpServletResponse
response, Object handler, ModelAndView modelAndView) throws Exception {
63         System.out.println("postHandle 运行了-----");
64         HandlerInterceptor.super.postHandle(request, response, handler,
modelAndView);
65     }
66
67     @Override // 视图渲染完毕后运行, 最后运行这个方法
68     public void afterCompletion(HttpServletRequest request,
HttpServletResponse response, Object handler, Exception ex) throws Exception
{
69         System.out.println("afterCompletion 运行了-----");
70         HandlerInterceptor.super.afterCompletion(request, response, handler,
ex);
71     }
72 }
```

配置类中, 可以选择不拦截 /login, 也可以不管, 因为已经在过滤器中对 /login 的请求进行了放行

```

1  @Configuration // 配置类
2  public class WebConfig implements WebMvcConfigurer {
3
4      @Autowired
5      LoginCheckInterceptor loginCheckInterceptor;
6
7      @Override
8      public void addInterceptors(InterceptorRegistry registry) {
9
10         registry.addInterceptor(loginCheckInterceptor).addPathPatterns("/**").exclu
11         dePathPatterns("/login");
12     }

```

接口测试，先发一个登录请求，获取到JWT令牌，在不携带令牌的情况下，发一个查询部门请求，控制台中会输出

请求的url: <http://localhost:8080/depts>

请求头Token为空，返回未登录信息

加上令牌再发送，控制台会输出，并执行业务操作，最后执行拦截器的 postHandle afterCompletion

请求的url: <http://localhost:8080/depts>

令牌合法，放行

查询全部部门数据

前后端联调，先将本地存储空间中的token删掉再重新登录。

复制一个系统内部的地址如 <http://localhost:90/#/system/emp>，退出登录后输入网址看能否直接进入，发现会跳到登录页面，只有登录后才能继续访问。

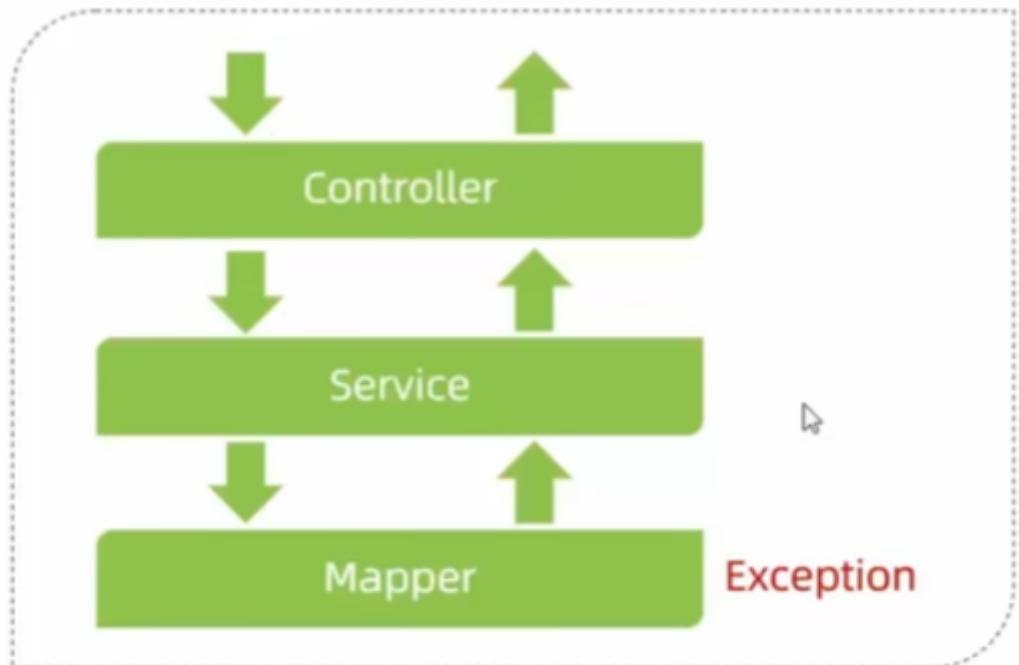
14. 异常处理

先来个异常，在部门管理中新增部门，写一个重名的部门，F12查看提交的请求，请求的错误码为500说明是服务端异常。查看IDEA控制台的报错，原因是数据库对deptname设置了唯一约束不允许重名。

前端收到的响应数据也不是统一响应结果Result，而是一个错误信息，不符合开发规范所以前端不能解析这个数据。

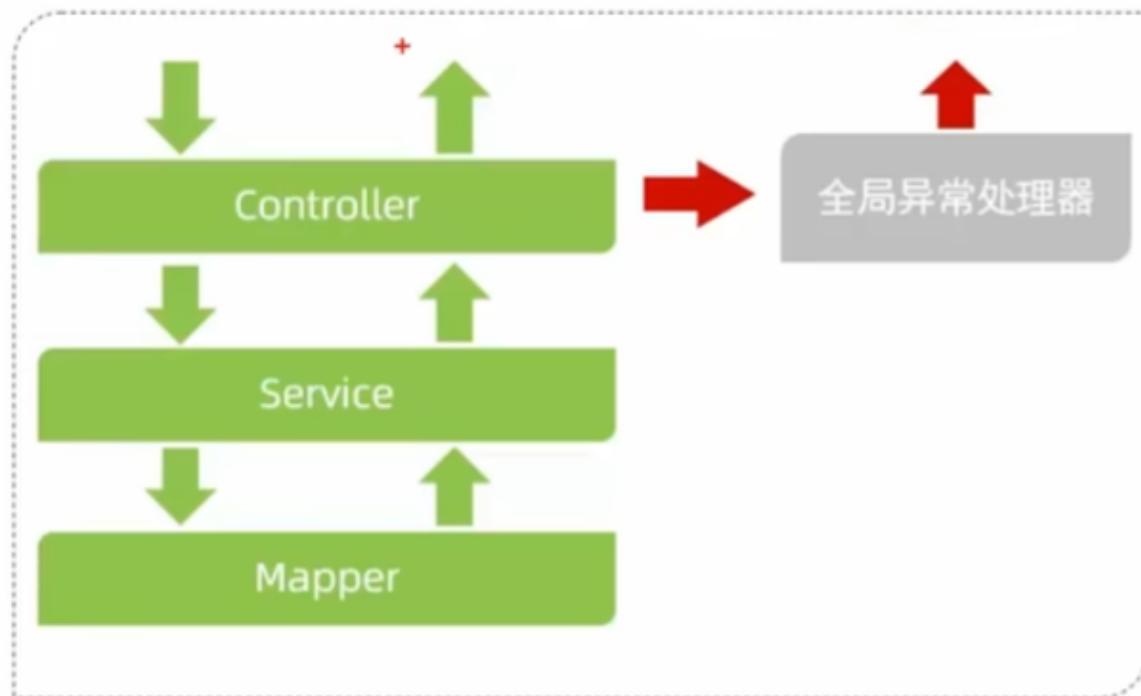
X	标头	载荷	预览	响应	启动器	时间
1				<pre>{ "timestamp": "2024-02-18T03:57:22.685+00:00", "status": 500, "error": "Internal Server Error", "path": "/depts" }</pre>		

现在要考虑的是出现异常后该怎么处理异常。先考虑当前案例中怎么处理的，发现居然什么没有处理。如果Mapper层在操作数据库时出现了异常，谁调用就抛给谁，就会抛给Service层，Service再抛给Controller，因为三层中都没有处理异常，所以Controller继续往上抛，抛给框架，框架就会返回这个JSON格式的错误信息，但这个错误信息并不符合开发规范。



异常处理方案

1. 在Controller的每个方法都进行 `try...catch` 处理，操作简单，但代码臃肿，不推荐
2. 定义一个全局异常处理器，捕获项目所有的异常，简单，推荐



当错误被抛到Controller中后，Controller将错误抛给处理器，处理器处理完异常后给前端返回统一响应结果Result，封装错误信息。

定义全局异常处理器，在类上添加注解 `@RestControllerAdvice` 代表定义了一个全局异常处理器，在处理器中定义一个方法来捕获异常，给方法添加注解 `@ExceptionHandler(Exception.class)`，通过 `value` 属性指定要捕获的是哪一类异常，`Exception.class` 代表要捕获所有的异常，最后方法返回一个标准的 `Result`，封装提示信息响应给前端。这个 `Result` 因为类上有注解 `@RestControllerAdvice` 等价于 `@Controller + @ResponseBody`，`@ResponseBody` 会将方法返回值转化为JSON响应回去。

```

1 /**
2  * 全局异常处理器
3 */
4 @RestControllerAdvice
5 public class GlobalExceptionHandler {
6
7     @ExceptionHandler(Exception.class)
8     public Result ex(Exception ex) {
9         ex.printStackTrace();
10        return Result.error("操作失败, 请联系管理员");
11    }
12}

```

15. 事务管理

- 简单回顾

事务: 一组操作集合，一个不可分割的工作单位，操作只能同时成功或同时失败。

开事务: start transaction/begin

提交事务: commit

回滚事务: rollback

完善删除部门的功能，现在的删除部门只是根据id把部门删掉，并没有把该部门的员工一起删掉，这会导致数据的不一致。因此需要添加一个删除部门员工的接口。

现在DeptServiceImpl中的删除部门:

```

1     @Override
2     public void delete(Integer id) {
3         deptMapper.delete(id);
4     }

```

因为删除员工是对员工操作，需要在 EmpMapper 中写SQL处理

```

1 /**
2  * 根据部门id删除员工
3 */
4 @Delete("delete from emp where dept_id = #{id};")
5 void deleteByDeptId(Integer id);

```

再在DeptServiceImpl的删除方法中添加员工删除:

```

1     @Override
2     public void delete(Integer id) {
3         deptMapper.delete(id); // 根据id删除部门数据
4
5         empMapper.deleteByDeptId(id); // 根据部门id删除部门员工
6     }

```

打开前端页面删除3号部门，发现对应的员工也被删掉了，这是正常运行的情况。

测试一下异常情况，在两个删除接口中间造一个异常

```
1  @Override
2  public void delete(Integer id) {
3      deptMapper.delete(id); // 根据id删除部门数据
4
5      int i = 1 / 0;
6
7      empMapper.deleteByDeptId(id); // 根据部门id删除部门员工
8  }
```

打开页面删除2号部门，会弹出操作失败的弹窗，打开数据库表可以看到部门被删除了，但是员工没有被删除。数据库当中的表结构就不完整了。

在执行 `int i = 1 / 0` 时出现了异常，一旦出现异常后边的代码都不再执行，意味着员工删除就不会执行，造成数据不一致。要保证前后操作的一致性，需要让两步操作要么同时成功，要么同时失败。就需要让两个操作处于同一个事务中。



而且无论哪个方法要想进行事务控制都是这样的模式，所以在Spring框架中已经把事务控制代码都封装好了，不需要手动实现。

Spring事务管理

只需要一个简单的注解 `@Transactional`，作用是在当前方法执行之前开启事务，执行完毕后提交事务，如果执行过程中出现了异常，Spring会自动进行事务回滚操作。

注解一般用在业务层Service中，因为一个业务可能有多个访问数据的操作，在业务层控制事务可以将多个访问操作控制在一个事务范围内。

注意：`@Transactional` 不仅可以写在方法上，也可以写在类或接口上。作用在方法上表示将当前方法交给Spring进行事务管理，作用在类上代表将类中所有方法交给Spring进行事务管理，作用在接口上代表将接口的所有实现类的所有方法交给Spring进行事务管理。

实际上只在执行多次增删改的操作方法上加注解，不会在类、接口上写注解，因为查操作不会变更数据，单独的增删改自动提交，不需要事务来进行管理。

可以在yml文件中配置一个日志开关，是Spring进行事务管理的日志开关，写法固定

```
1  # 开启事务管理日志
2  logging:
3    level:
4      org.springframework.jdbc.support.JdbcTransactionManager: debug
```

先在数据库中新增一个教研部，再在页面中删除这个部门，查看控制台的信息

```
DEBUG 18072 — [nio-8080-exec-1] o.s.jdbc.support.JdbcTransactionManager : Initiating transaction rollback
DEBUG 18072 — [nio-8080-exec-1] o.s.jdbc.support.JdbcTransactionManager : Rolling back JDBC transaction on C
DEBUG 18072 — [nio-8080-exec-1] o.s.jdbc.support.JdbcTransactionManager : Releasing JDBC Connection [HikariP
```

再看数据库中的数据并没有被删掉，说明事务运行失败自动回滚了

16. 事务进阶

在两个删除调用中间手动抛出一个异常

```
1  @Transactional
2  @Override
3  public void delete(Integer id) throws Exception {
4      deptMapper.delete(id); // 根据id删除部门数据
5
6      //      int i = 1 / 0;
7
8      if (true) { // 骗一下编译器，否则底下的删除语句会报错
9          throw new Exception("出错了");
10     }
11
12     empMapper.deleteByDeptId(id); // 根据部门id删除部门员工
13 }
```

还是在页面中删除教研部，会弹出操作失败的提示框。此时打开控制台查看debug的日志信息，会发现第一行不是 rollback 了，而是 commit，说明事务提交了。打开数据库会发现部门被删除了，但相关员工没有被删除。

```
DEBUG 19140 — [nio-8080-exec-1] o.s.jdbc.support.JdbcTransactionManager : Initiating transaction commit
DEBUG 19140 — [nio-8080-exec-1] o.s.jdbc.support.JdbcTransactionManager : Committing JDBC transaction on C
DEBUG 19140 — [nio-8080-exec-1] o.s.jdbc.support.JdbcTransactionManager : Releasing JDBC Connection [HikariP
```

rollbackFor属性

默认情况下，只有出现运行时异常 `RuntimeException`，才会回滚异常。

若想让所有的异常都回滚，可以配置 `@Transactional` 的 `rollbackFor` 属性来控制出现何种异常时才回滚事务。

```
1 | @Transactional(rollbackFor = Exception.class)
```

再把教研部添加回来，在网页中尝试删除，网页会提出操作错误提示框，查看控制台中的debug信息，会发现此时事务进行了回滚操作。

```
DEBUG 5728 — [nio-8080-exec-2] o.s.jdbc.support.JdbcTransactionManager : Initiating transaction rollback
DEBUG 5728 — [nio-8080-exec-2] o.s.jdbc.support.JdbcTransactionManager : Rolling back JDBC transaction on C
DEBUG 5728 — [nio-8080-exec-2] o.s.jdbc.support.JdbcTransactionManager : Releasing JDBC Connection [HikariP
```

propagation属性

配置事务传播行为，事务传播行为指的是一个事务方法被另一个事务方法调用时，这个事务方法该如何进行事务控制。

如a方法是个事务，b方法也是个事务，a方法中调用了b方法，a方法运行时会开启一个事务，调用b方法时因为b方法也具有事务，那到底是加入a方法还是新建一个事务？

可以通过 `propagation` 属性为b方法指定传播行为

- 常见的事物传播行为

- o REQUIRED, 【默认值】，代表b方法运行时需要事务，如果有则加入，没有则创建新事务。
- o REQUIRES_NEW，调用b方法时，不论a有无事务，b都会在新事务中运行。
- o SUPPORTS，支持事务，调用b时发现有事务那就加入，没有事务就在无事务状态中运行
- o NOT_SUPPORTED，不支持事务，调用时发现存在事务，先把事务挂起，执行b方法，执行完毕后再来执行当前事务
- o MANDATORY：必须有事务，否则抛出异常
- o NEVER：必须没有事务，否则抛出异常

删除部门及员工，记录日志到数据库表中，**要求操作失败、成功都要记录日志到数据库**

先定义一个 DeptLog 的pojo类

```

1  @Data
2  @NoArgsConstructor
3  @AllArgsConstructor
4  public class DeptLog {
5
6      private Integer id;
7      private LocalDateTime createTime;
8      private String description;
9  }

```

定义 DeptLogService 接口和实现类，加上一个 @Transactional

```

1  @Service
2  public class DeptLogServiceImpl implements DeptLogService {
3
4      @Autowired
5      private DeptLogMapper deptLogMapper;
6
7      @Transactional
8      @Override
9      public void insert(DeptLog deptLog) {
10         deptLogMapper.insert(deptLog);
11     }
12 }

```

定义 DeptLogMapper 接口

```

1  @Mapper
2  public interface DeptLogMapper {
3
4      @Insert("insert into dept_log (create_time, description) values (#{createTime}, #{description})")
5      void insert(DeptLog deptLog);
6  }

```

修改 DeptServiceImp1 中的删除方法，这里还有一个 @Transactional 注解，这样就构成了事务传播

```

1  //    @Transactional(rollbackFor = Exception.class)
2  @Transactional
3  @Override
4  public void delete(Integer id) throws Exception {
5      deptMapper.delete(id); // 根据id删除部门数据
6

```

```

7     int i = 1 / 0;
8
9 //     if (true) {
10 //         throw new Exception("出错了");
11 //     }
12
13     empMapper.deleteByDeptId(id); // 根据部门id删除部门员工
14
15     DeptLog deptLog = new DeptLog();
16     deptLog.setCreateTime(LocalDateTime.now());
17     deptLog.setDescription("执行了解散部门操作，解散的是" + id + "号部门");
18     deptLogService.insert(deptLog);
19 }
```

由于要求失败、成功都要记录，所以要优化一下，将日志信息的生成和存储放到 `finally` 代码块中

使用 `try finally` 包裹

```

1 // @Transactional(rollbackFor = Exception.class)
2 @Transactional
3 @Override
4 public void delete(Integer id) throws Exception {
5     try {
6         deptMapper.delete(id); // 根据id删除部门数据
7
8         int i = 1 / 0;
9
10 //     if (true) {
11 //         throw new Exception("出错了");
12 //     }
13
14         empMapper.deleteByDeptId(id); // 根据部门id删除部门员工
15
16     } finally {
17         DeptLog deptLog = new DeptLog();
18         deptLog.setCreateTime(LocalDateTime.now());
19         deptLog.setDescription("执行了解散部门操作，解散的是" + id + "号部
门");
20         deptLogService.insert(deptLog);
21     }
22 }
23 }
```

再再来删除教研部，不出意外还是失败，打开表结构，**看看日志是否记录进来，发现日志没有写入到数据库中**。在控制台中重点看事务管理的日志信息，执行删除部门操作后影响了1条数据，然后抛出异常，进入 `finally` 代码块中，调用了 `deptLogService.insert`，看见控制台中有这么一句

```
o.s.jdbc.support.JdbcTransactionManager : Participating in existing transaction
```

说明参与到了一个已存在的事务，`delete` 和日志 `insert` 共用一个事务，但**由于方法执行过程中抛出了除0的异常，所以整个事务都得回滚**，因此日志插入也会被回滚。

因为 `@Transactional` 的 `propagation` 属性默认值是 `REQUIRED`，有事务就加入到已存在的事务，没有事务再开一个新的事物。要满足当前需求就不能使用 `REQUIRED`，日志插入要使用一个新事物，配置属性为 `REQUIRES_NEW`，总是开启新事务。

```

1  @Service
2  public class DeptLogServiceImpl implements DeptLogService {
3
4      @Autowired
5      private DeptLogMapper deptLogMapper;
6
7      @Transactional(propagation = Propagation.REQUIRES_NEW)
8      @Override
9      public void insert(DeptLog deptLog) {
10         deptLogMapper.insert(deptLog);
11     }
12 }

```

在调用insert时，会挂起当前事务，创建一个新的事务

```
o.s.jdbc.support.JdbcTransactionManager : Suspending current transaction, creating new transaction with name
```

对insert事务的提交

```
o.s.jdbc.support.JdbcTransactionManager : Initiating transaction commit
o.s.jdbc.support.JdbcTransactionManager : Committing JDBC transaction on Connection [HikariProxyConnection@9310282 wrapping
o.s.jdbc.support.JdbcTransactionManager : Releasing JDBC Connection [HikariProxyConnection@9310282 wrapping
o.s.jdbc.support.JdbcTransactionManager : Resuming suspended transaction after completion of inner transact
```

最后一行意思是内部事务结束以后，继续运行刚才挂起的事务，即delete方法的事务，delete报错之后会正常的进行回滚操作

```
o.s.jdbc.support.JdbcTransactionManager : Initiating transaction rollback
o.s.jdbc.support.JdbcTransactionManager : Rolling back JDBC transaction on Connection [HikariProxyConnection@9310282 wrapping
o.s.jdbc.support.JdbcTransactionManager : Releasing JDBC Connection [HikariProxyConnec
```

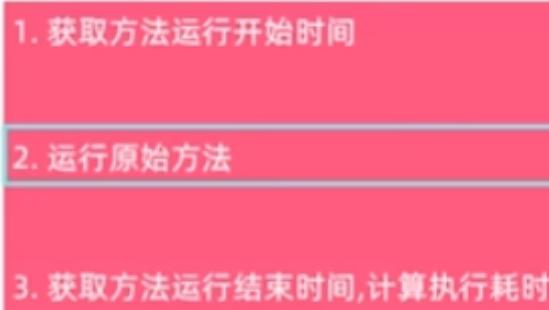
注意：大部分情况下使用REQUIRED即可，当不希望事务之间相互影响时，可以使用REQUIRES_NEW

17. AOP

Spring第二大核心（第一大IoC），Aspect Oriented Programming，面向切面/方面编程，就是面向特定方法编程。

比如要统计案例中的所有业务方法的耗时，一般的想法是在每个方法开始时记录时间，结束时记录时间，相减得到执行耗时。但要修改每个方法就很笨，很繁琐。

如果是面向方法编程，就可以在不改动原始方法的基础上，来针对原始方法进行编程，可以是对原有方法的增强，也可以是改变原始方法的功能。此时要想完成统计各个方法耗时的需求，**只需要定义一个模板方法**，将记录方法耗时这部分公共的代码定义在模板方法中。再记录开始时间、结束时间，中间运行原始方法（需要统计耗时的业务方法）。



实际上和**动态代理（SE进阶）**类似，模板方法中定义的逻辑，就是创建出来的代理对象方法的逻辑，在代理对象方法当中，可以根据需要在原始方法运行前后做些事情。

动态代理是面向切面编程最主流的实现，SpringAOP是Spring框架的高级技术，目的在于管理bean对象的过程中，通过底层动态代理机制对特定方法进行编程。

AOP快速入门

统计各个业务层方法的执行耗时，定位出耗时较长的方法

- 导入依赖，在pom中导入起步依赖
- 编写AOP程序，针对特定方法根据需要进行编程

```
1 @Component // 交给IoC容器管理
2 @Aspect // 代表当前类不是普通类而是AOP类
3 public class TimeAspect {
4     定义方法完成耗时统计
5 }
```

新建一个模块，勾选导入四个依赖，把案例中的部门相关都复制过来，手动添加AOP依赖

```
1 <dependency>
2     <groupId>org.springframework.boot</groupId>
3     <artifactId>spring-boot-starter-aop</artifactId>
4 </dependency>
```

新建AOP类 TimeAspect，添加注解 @Component 将类交给IoC容器管理，成为bean对象。还要标识这是个AOP类所以要加注解 @Aspect。在类中定义方法，完成三步逻辑（记录开始时间、运行原始方法、记录结束时间+计算耗时）

调用原始方法需要使用AOP中提供的API，就叫 ProceedingJoinPoint，接着需要调用 ProceedingJoinPoint 中的一个方法 proceed()，这句一运行就代表要运行原始方法。原始方法运行时可能会有返回值，所以proceed方法调用完成后也有返回值，返回一个Object代表原始方法的返回值，等这个方法运行结束时将原始方法的返回值返回回去。

注：此处先不关注 @Around() 注解和内部的内容

```
1 package com.itheima.aop;
2
3 import lombok.extern.slf4j.Slf4j;
4 import org.aspectj.lang.ProceedingJoinPoint;
5 import org.aspectj.lang.annotation.Around;
6 import org.aspectj.lang.annotation.Aspect;
7 import org.springframework.stereotype.Component;
8
9 import java.time.LocalDateTime;
10
11 @Component
12 @Aspect
13 @Slf4j
14 public class TimeAspect {
15
16
17     @Around("execution(* com.itheima.service.*.*(..))") // 切入点表达式
18     public Object recordTime(ProceedingJoinPoint joinPoint) throws Throwable
19     {
20         // 1. 记录开始时间
21         long begin = System.currentTimeMillis();
```

```

22     // 2. 调用原始方法运行
23     Object result = joinPoint.proceed(); // 原始方法的返回值
24
25     // 3. 记录结束时间，计算耗时
26     long end = System.currentTimeMillis();
27
28     log.info("{}方法执行时间: {}", joinPoint.getSignature(), end - begin);
29     // getSignature获取方法名
30     return result; // 将原始方法的返回值返回回去
31 }
32

```

打开页面访问部门管理页面，就会请求部门列表数据，控制台中就会输出方法的耗时

List com.itheima.service.impl.DeptServiceImpl.list()方法执行时间: 298ms

点击编辑部门按钮，就会触发根据id查询部门数据操作，可以在控制台中查看方法的耗时

Dept com.itheima.service.impl.DeptServiceImpl.getById(Integer)方法执行时间: 4ms

这样就完成了业务层耗时统计的操作，可以发现**没有修改任何原始方法就添加了统计方法耗时的功能**

AOP还能完成很多功能，如记录操作日志，完成事务管理（Spring底层就是通过AOP实现的事务管理）。

不用修改原始方法就已经对原始方法进行了功能的增强或改变（代码无侵入），减少了重复代码，提高效率，维护方便。

AOP核心概念

- 连接点：JoinPoint，可以被AOP控制的方法。如所有的部门相关方法都是可以被AOP控制的方法，所以这些方法都是连接点。SpringAOP提供的JoinPoint中提供了连接点方法执行时的相关信息。
- 通知：Advice，重复的逻辑，也就是共性功能，最终体现为一个方法。



共性功能到底要应用在哪些方法上？

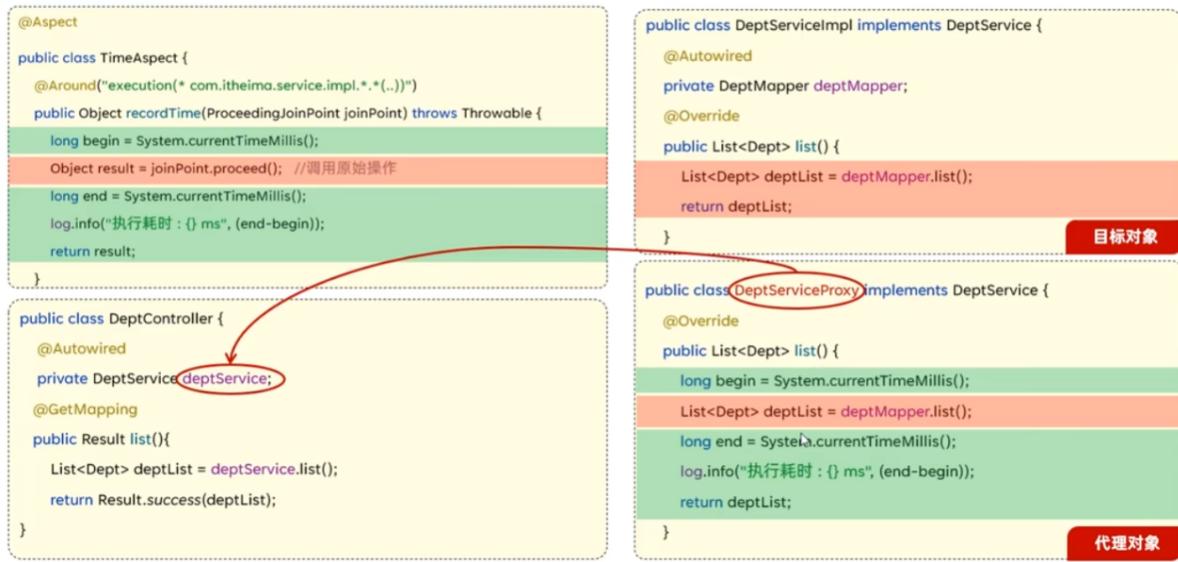
- 切入点：PointCut，匹配（感觉说查找更合适）连接点（就是被AOP调用的方法）的条件，就是查找到被AOP调用的方法然后在执行这个方法前后执行AOP类中定义的方法，通知只会在切入点方法执行时被应用。实际上切入点就是实际被AOP控制的方法，在**AOP开发中常通过切入点表达式来描述切入点**。
- 切面：Aspect，描述通知与切入点的对应关系（通知+切入点）。通过切面就能描述当前AOP程序需要针对哪个原始方法在什么时候执行什么操作。切面所在的类（被 @Aspect 注解标识的类）一般称为切面类。

- 目标对象：Target，通知应用的对象，就是可以被切入点调用的方法所在的类（接口）

通知如何与目标对象结合在一起？

AOP执行流程

前边提到SpringAOP底层是基于动态代理实现的。在运行时会自动基于动态代理技术为目标对象生成一个代理对象。在代理对象中会对目标对象的原始方法进行功能增强。增强的逻辑就是AOP类的通知（也就是AOP类的方法）。最终程序运行时自动注入使用的对象就不再是目标对象，而是代理对象，调用方法时就是调用代理对象中的增强之后的方法。



打开页面点击部门管理，通过断点调试查看信息

```

15 ① @Autowired
16   private DeptService deptService; deptService: "com.itheima.service.impl.DeptServiceProxy@2337d2b8"
17
+ {DeptServiceImpl$$SpringCGLIB$$0@7235} "com.itheima.service.impl.DeptServiceImpl@2337d2b8"

```

注入的对象是 `DeptServiceProxy`，后边一串。这里用到了Spring底层另一种动态代理技术 `CGLIB`，现在注入的就是代理对象，调用的是代理对象的 `list` 方法。代理对象的方法要进行功能增强所以不会直接运行目标对象 (`DeptServiceImpl`) 的 `list` 方法，而是先进入到AOP类中进行方法增强。

```

// 1. 记录开始时间
long begin = System.currentTimeMillis(); begin: 1708256158668

// 2. 调用原始方法运行
Object result = joinPoint.proceed(); // 原始方法的返回值 joinPoint: "execution(* com.itheima.service.impl.*(..))"

// 3. 记录结束时间，计算耗时
long end = System.currentTimeMillis();

```

当前的断点放行之后就会调用原始的目标对象 (`DeptServiceImpl`) 的 `list` 方法

```
@Service
public class DeptServiceImpl implements DeptService {
    @Autowired
    private DeptMapper deptMapper;  deptMapper: $Proxy72@7282

    @Override
    public List<Dept> list() {
        List<Dept> deptList = deptMapper.list();  deptMapper: $Proxy72@7282
        return deptList;
    }
}
```

继续断点放行，会回到AOP类中继续执行剩下的逻辑，最后返回结果

```
// 3. 记录结束时间，计算耗时
long end = System.currentTimeMillis();

log.info("{}方法执行时间: {}ms", joinPoint.getSignature(), end - begin); // getSignat
return result; // 将原始方法的返回值返回回去
```

AOP通知类型

1. `@Around`：环绕通知，在目标方法执行之前和之后都会执行通知方法的逻辑。
2. `@Before`：前置，在目标方法执行之前执行。
3. `@After`：后置，也可称为最终通知，在目标方法执行之后执行，**有无异常都会执行**。
4. `@AfterReturning`：返回后，在目标方法正常执行完毕正常回到通知后执行，**有异常时不会执行**。
5. `@AfterThrowing`：异常后，目标方法发生异常后执行。

通知执行顺序为：`around before afterReturning after around`

如果有异常：`around before afterThrowing after`（两个运行顺序见下方截图）

新建一个AOP类，加上这些注解

```
1 package com.itheima.aop;
2
3 import lombok.extern.slf4j.Slf4j;
4 import org.aspectj.lang.ProceedingJoinPoint;
5 import org.aspectj.lang.annotation.*;
6 import org.springframework.stereotype.Component;
7
8 @Component
9 @Aspect
10 @Slf4j
11 public class MyAspect1 {
12
13     @Around("execution(* com.itheima.service.impl.DeptServiceImpl.*(..))")
14     public Object Around(ProceedingJoinPoint joinPoint) throws Throwable {
15         log.info("around before...");
16
17         // 调用目标对象原始方法运行
18         Object proceed = joinPoint.proceed();
19
20         log.info("around after...");
21
22         return proceed;
23     }
24 }
```

```

25     @Before("execution(* com.itheima.service.impl.DeptServiceImpl.*(..))")
26     public void before() {
27         log.info("before...");
28     }
29
30     @After("execution(* com.itheima.service.impl.DeptServiceImpl.*(..))")
31     public void after() {
32         log.info("after...");
33     }
34
35     @AfterReturning("execution(* com.itheima.service.impl.DeptServiceImpl.*(..))")
36     public void afterReturning() {
37         log.info("after return...");
38     }
39
40     @AfterThrowing("execution(* com.itheima.service.impl.DeptServiceImpl.*(..))")
41     public void afterThrowing() {
42         log.info("after throw...");
43     }
44 }
```

打开页面访问一次部门管理，看控制台中的信息，在原始方法运行之前运行了两个通知

<code>com.itheima.aop.MyAspect1</code>	: around before ...
<code>com.itheima.aop.MyAspect1</code>	: before ...

原始方法执行结束后运行了三个通知

<code>com.itheima.aop.MyAspect1</code>	: after return ...
<code>com.itheima.aop.MyAspect1</code>	: after ...
<code>com.itheima.aop.MyAspect1</code>	: around after ...

afterThrowing没有执行，现在测试一下异常情况，在随便一个方法中加一句`int i = 1 / 0;`在页面中点一下编辑，看控制台的信息

<code>com.itheima.aop.MyAspect1</code>	: around before ...
<code>com.itheima.aop.MyAspect1</code>	: before ...
<code>com.itheima.aop.MyAspect1</code>	: after throw ...
<code>com.itheima.aop.MyAspect1</code>	: after ...

环绕前和前置通知都执行了，此时调用原始方法报错，就运行了异常后通知和后置通知。但afterReturning没有执行，因为只有目标方法正常执行返回他才会运行。

环绕后通知也没有运行，显然是因为原始方法出现了异常所以下一句就不会再运行。

可以发现AfterReturning和AfterThrowing是互斥的。

注意：@Around需要自己调用 ProceedJoinPoint.proceed() 让原始方法执行，其他通知类型不需要考虑目标方法执行。@Around的返回值必须为Object

此时发现代码中切入点表达式大量的重复了，需要将其抽取出来，只需要在AOP类中定义一个空方法，使用注解`@Pointcut(切入点表达式)`就可以重复调用这个表达式了，将各个通知括号内的表达式都替换为这个函数的调用（就是写函数名和`()`，不要只写函数名）

```

1 package com.itheima.aop;
2
3 import lombok.extern.slf4j.Slf4j;
4 import org.aspectj.lang.ProceedingJoinPoint;
5 import org.aspectj.lang.annotation.*;
6 import org.springframework.stereotype.Component;
7
8 @Component
9 @Aspect
10 @Slf4j
11 public class MyAspect1 {
12
13     @Pointcut("execution(* com.itheima.service.impl.DeptServiceImpl.*(..))")
14     public void defCutPoint() {}
15
16     @Around("defCutPoint()")
17     public Object Around(ProceedingJoinPoint joinPoint) throws Throwable {
18         log.info("around before...");
19
20         // 调用目标对象原始方法运行
21         Object proceed = joinPoint.proceed();
22
23         log.info("around after...");
24
25         return proceed;
26     }
27
28     @Before("defCutPoint()")
29     public void before() {
30         log.info("before...");
31     }
32
33     @After("defCutPoint()")
34     public void after() {
35         log.info("after...");
36     }
37
38     @AfterReturning("defCutPoint()")
39     public void afterReturning() {
40         log.info("after return...");
41     }
42
43     @AfterThrowing("defCutPoint()")
44     public void afterThrowing() {
45         log.info("after throw...");
46     }
47 }

```

注意：抽取的表达式还可以在其他切面类中使用，不过要注意方法的访问限定修饰符不能是私有。

```

1 @Component
2 @Aspect
3 @Slf4j
4 public class TimeAspect {
5
6
7     @Around("com.itheima.aop.MyAspect1.defCutPoint()") // 切入点表达式

```

```

8     public Object recordTime(ProceedingJoinPoint joinPoint) throws Throwable
9     {
10         // 1. 记录开始时间
11         long begin = System.currentTimeMillis();
12
13         // 2. 调用原始方法运行
14         Object result = joinPoint.proceed(); // 原始方法的返回值
15
16         // 3. 记录结束时间，计算耗时
17         long end = System.currentTimeMillis();
18
19         log.info("{}方法执行时间: {}ms", joinPoint.getSignature(), end - begin); // getSignature获取方法名
20         return result; // 将原始方法的返回值返回回去
21     }

```

AOP通知顺序

有多个切面的切入点都匹配到了目标方法，目标方法运行时多个通知方法都会运行。

定义三个切面类，每个类中都写上before和after通知

```

1 package com.itheima.aop;
2
3 import lombok.extern.slf4j.Slf4j;
4 import org.aspectj.lang.ProceedingJoinPoint;
5 import org.aspectj.lang.annotation.*;
6 import org.springframework.stereotype.Component;
7
8 @Component
9 @Aspect
10 @Slf4j
11 public class MyAspect2 {
12
13     @Before("com.itheima.aop.MyAspect1.defCutPoint()")
14     public void before() {
15         log.info("before...2");
16     }
17
18     @After("com.itheima.aop.MyAspect1.defCutPoint()")
19     public void after() {
20         log.info("after...2");
21     }
22
23 }

```

从页面发一个请求，看控制台信息

<code>com.itheima.aop.MyAspect2</code>	: before ... 2
<code>com.itheima.aop.MyAspect3</code>	: before ... 3
<code>com.itheima.aop.MyAspect4</code>	: before ... 4

```
com.itheima.aop.MyAspect4          : after ... 4  
com.itheima.aop.MyAspect3          : after ... 3  
com.itheima.aop.MyAspect2          : after ... 2
```

执行顺序和切面类类名有关，和过滤器的顺序定义方式一样。

这种方式很繁琐，不便管理。Spring提供了第二种方式，可以使用注解 @Order(序号) 加在切面类上来指定执行的顺序，在目标方法执行前的通知，序号小的先执行，目标方法执行后的通知，序号大的先执行。

现在让执行顺序为4 2 3，那就需要给 MyAspect4 加序号为1，给2加序号为2，给3加序号为3

示例一下：

```
1 package com.itheima.aop;  
2  
3 import lombok.extern.slf4j.Slf4j;  
4 import org.aspectj.lang.annotation.After;  
5 import org.aspectj.lang.annotation.Aspect;  
6 import org.aspectj.lang.annotation.Before;  
7 import org.springframework.core.annotation.Order;  
8 import org.springframework.stereotype.Component;  
9  
10 @Component  
11 @Aspect  
12 @Slf4j  
13 @Order(1)  
14 public class MyAspect4 {  
15  
16     @Before("com.itheima.aop.MyAspect1.defCutPoint()")  
17     public void before() {  
18         log.info("before...4");  
19     }  
20  
21     @After("com.itheima.aop.MyAspect1.defCutPoint()")  
22     public void after() {  
23         log.info("after...4");  
24     }  
25  
26 }
```

注意：点开 @Order 注解可以看到 value 属性值是 int 类型，也就是说可以有负数序号，那就知道最优先的序号为 -2147483648（估计没人会写负数序号吧）

切入点表达式

用来决定项目中哪些方法需要使用定义的通知。

常见形式

1. execution(...): 根据方法签名来匹配
2. @annotation(...): 根据注解匹配

execution

根据方法的返回值、包名、类名、方法名、方法参数来匹配

语法格式：execution([访问修饰符] 返回值 [包名.类名.]方法名(参数类型) [throws 异常])

简写一下：execution(返回值 包名.类名.方法名(参数类型))

访问修饰符可省略，包名.类名可省略（不推荐），throws 异常可省略，是方法名上声明抛出的异常，不是运行时产生的异常

写个切面类，把切面表达式抽取出来

注意：切面表达式中的方法参数类型，可能需要写全类名？这里没写全类名也是可以运行的

```
1 package com.itheima.aop;
2
3 import lombok.extern.slf4j.Slf4j;
4 import org.aspectj.lang.annotation.After;
5 import org.aspectj.lang.annotation.Aspect;
6 import org.aspectj.lang.annotation.Before;
7 import org.aspectj.lang.annotation.Pointcut;
8 import org.springframework.core.annotation.Order;
9 import org.springframework.stereotype.Component;
10
11 @Component
12 @Aspect
13 @Slf4j
14 @Order(1)
15 public class MyAspect6 {
16
17     // 为delete方法加入通知
18     @Pointcut("execution(public void
com.itheima.service.impl.DeptServiceImpl.delete(Integer))")
19     public void defcutPoint() {}
20
21     @Before("defcutPoint()")
22     public void before() {
23         log.info("MyAspect6 ... before...");
24     }
25 }
```

在测试类中写几个方法

```
1 @SpringBootTest
2 class SpringBootAopQuickStartApplicationTests {
3     @Autowired
4     private DeptService deptService;
5
6     @Test
7     void testAopDelete() {
8         deptService.delete(10);
9     }
10
11     @Test
12     void testAopList() {
13         System.out.println(deptService.list());
14     }
15
16     @Test
17     void testAopGetById() {
18         System.out.println(deptService.getById(1));
```

```
19     }
20 }
```

切入点表达式的通配符号

- `*`: 单个独立的任意符号, 可以通配任意返回值、包名、类名、方法名、任意类型的一个参数, 也可以通配包、类、方法名的一部分

`execution(* com.*.service.*.update*())`, update*表示匹配以update开头的方法

`*+单词 单词+*`分别表示匹配以单词结尾和开头的词

- `..`: 多个连续任意符号, 可以通配任意多个包名, 任意类型、任意个数的参数

`execution(* com.itheima..DeptService.*(..))`, 中间的..表示匹配任意的包

注意: `@Pointcut`注解中的`execution()`可以使用逻辑表达式来组合成较复杂的表达式

示例一下, 匹配list方法和delete方法

```
1 @Pointcut("execution(* com.itheima.service.DeptService.list()) || execution(*
2     com.itheima.service.DeptService.delete())")
3     public void defcutPoint() {}
```

- **书写建议**

- 所有业务方法名在**命名时尽量规范**方便表达式快速匹配。如: 查询类都是find开头, 更新类都是update开头
- 描述切入点方法通常**基于接口描述**, 而不是实现类, 增强拓展性
- **尽量缩小切入点的匹配范围**

annotation

同时匹配多个无规则的方法通过`execution`来描述就太繁琐了, 可以借助`@annotation(注解全类名)`来简化

annotation切入点表达式用于匹配有特定注解的方法

示例一下: 先定义一个注解, 加上两个元注解来描述当前注解。`@Retention`来描述这个自定义注解什么时候生效, `@Target`来指定这个自定义注解可以作用在哪些地方。设置当前注解运行时有效和作用在方法上

```
1 package com.itheima.aop;
2
3 import java.lang.annotation.ElementType;
4 import java.lang.annotation.Retention;
5 import java.lang.annotation.RetentionPolicy;
6 import java.lang.annotation.Target;
7
8
9 @Retention(RetentionPolicy.RUNTIME)
10 @Target(ElementType.METHOD)
11 public @interface MyLog {
12 }
```

在DeptServiceImpl的list方法和delete方法加上我们定义的注解 MyLog

```
1 @Slf4j
```

```

2  @Service
3  public class DeptServiceImpl implements DeptService {
4      @Autowired
5      private DeptMapper deptMapper;
6
7      @MyLog
8      @Override
9      public List<Dept> list() {
10         List<Dept> deptList = deptMapper.list();
11         return deptList;
12     }
13
14      @MyLog
15      @Override
16      public void delete(Integer id) {
17          //1. 删除部门
18          deptMapper.delete(id);
19      }
20
21      @Override
22      public void save(Dept dept) {
23          dept.setCreateTime(LocalDateTime.now());
24          dept.setUpdateTime(LocalDateTime.now());
25          deptMapper.save(dept);
26      }
27
28      @Override
29      public Dept getById(Integer id) {
30          int i = 1 / 0;
31          return deptMapper.getById(id);
32      }
33
34      @Override
35      public void update(Dept dept) {
36          dept.setUpdateTime(LocalDateTime.now());
37          deptMapper.update(dept);
38      }
39  }

```

修改AOP类，替换掉之前的execution表达式，

`@Pointcut("@annotation(com.itheima.aop.MyLog)")` 表示匹配方法上有`@MyLog`注解的方法

```

1  @Component
2  @Aspect
3  @Slf4j
4  public class MyAspect7 {
5
6      // 为delete方法加入通知
7      //  @Pointcut("execution(* com.itheima.service.DeptService.list()) ||"
8      //  execution(* com.itheima.service.DeptService.delete())")
9      @Pointcut("@annotation(com.itheima.aop.MyLog)")
10     public void defcutPoint() {}
11
12     @Before("defcutPoint()")
13     public void before() {
14         log.info("MyAspect7 ... before...");
```

运行之前的单元测试delete方法和list方法，看通知是否执行

```
com.itheima.aop.MyAspect7          : MyAspect7 ... before ...
com.itheima.aop.MyAspect7          : MyAspect7 ... before ...
```

两个通知都可以正常执行，把注解掉通知方法就不会再执行

连接点

可以被AOP控制的方法，在SpringAOP中连接点特指方法的执行。

Spring中通过JoinPoint对连接点进行了抽象，通过JoinPoint这个连接点对象获取目标执行时的相关信息，如类名、方法名、方法参数等

对 @Around 通知，获取连接点只能使用 ProceedingJoinPoint

对其他四个通知，获取连接点只能使用 JoinPoint，这是 ProceedingJoinPoint 的父类型

```
1 package com.itheima.aop;
2
3 import lombok.extern.slf4j.Slf4j;
4 import org.aspectj.lang.JoinPoint;
5 import org.aspectj.lang.ProceedingJoinPoint;
6 import org.aspectj.lang.Signature;
7 import org.aspectj.lang.annotation.Around;
8 import org.aspectj.lang.annotation.Aspect;
9 import org.aspectj.lang.annotation.Before;
10 import org.aspectj.lang.annotation.Pointcut;
11 import org.springframework.stereotype.Component;
12
13 import java.util.Arrays;
14
15 @Component
16 @Aspect
17 @Slf4j
18 public class MyAspect8 {
19
20     // 为delete方法加入通知
21     // @Pointcut("execution(* com.itheima.service.DeptService.list() || execution(* com.itheima.service.DeptService.delete())")
22     // @Pointcut("@annotation(com.itheima.aop.MyLog)")
23     public void defcutPoint() {}
24
25     @Before("defcutPoint()")
26     public void before(JoinPoint joinPoint) {
27         log.info("MyAspect8 ... before...\"");
28     }
29
30     @Around("defcutPoint()")
31     public Object around(ProceedingJoinPoint joinPoint) throws Throwable {
32         log.info("MyAspect8 around before...\"");
33
34         // 获取目标对象类名
35         String name = joinPoint.getClass().getName();
36         log.info("类名: {}", name);
```

```

37     // 获取目标对象方法名
38     Signature signature = joinPoint.getSignature();
39     log.info("方法名: {}", signature);
40     // 获取目标对象方法参数
41     Object[] args = joinPoint.getArgs();
42     log.info("参数: {}", Arrays.toString(args));
43     // 放行
44     // 获取目标对象返回值
45     Object proceed = joinPoint.proceed();
46     log.info("返回值: {}", proceed);
47
48     log.info("MyAspect8 around after... ");
49
50     return proceed;
51 }
52 }
```

AOP案例

将案例中增删改相关接口的操作日志记录到数据库表中

日志信息包含：操作人、操作时间、执行方法全类名、执行方法名、方法运行时参数、**返回值**、方法执行时长

通过AOP类来操作，要记录返回值和执行时长，那就需要用@Around类通知

切入点表达式：annotation

还是在案例模块中进行操作

引入AOP依赖，建立数据库表，引入实体类，自定义注解@Log，定义AOP类完成记录操作日志

要获取操作人的id，需要从请求头中的令牌中解析数据获得当前登录的员工，而现在是在AOP类中，不像过滤器和拦截器那样有参数 HttpServletRequest，这里可以直接声明一个 HttpServletRequest 类型的变量使用自动注入（为什么可以自动注入后边会讲），从请求头中解析员工的id作为操作人id

```

1  @Aspect
2  @Slf4j
3  @Component
4  public class LogAspect {
5
6      @Autowired
7      private HttpServletRequest servletRequest;
8
9      @Autowired
10     private OperateLogMapper operateLogMapper;
11
12     @Around("@annotation(com.itheima.anno.Log)")
13     public Object recordLog(ProceedingJoinPoint joinPoint) throws Throwable
14     {
15         // 环绕通知要手动调用方法运行
16
17         // 操作人ID - 登录员工的id，从jwt令牌中获取
18         // 获取请求头中的令牌，解析令牌
19         String jwt = servletRequest.getHeader("token");
20         Claims claims = JwtUtils.parseJWT(jwt);
21         Integer operateUser = (Integer) claims.get("id");
22         // 操作时间
```

```

22     LocalDateTime operateTime = LocalDateTime.now();
23     // 操作类名
24     String className = joinPoint.getTarget().getClass().getName();
25     // 操作方法名
26     String methodName = joinPoint.getSignature().getName();
27     // 操作方法参数
28     String methodParams = Arrays.toString(joinPoint.getArgs());
29
30     long begin = System.currentTimeMillis();
31     Object result = joinPoint.proceed();
32     long end = System.currentTimeMillis();
33     // 方法返回值
34     String returnValue = JSONObject.toJSONString(result);
35     // 执行耗时
36     Long costTime = end - begin;
37
38
39     // 记录操作日志
40     OperateLog operateLog = new OperateLog(null, operateUser,
41         operateTime, className,
42             methodName, methodParams, returnValue, costTime);
43     operateLogMapper.insert(operateLog);
44
45     log.info("记录操作日志: {}", operateLog);
46
47     return result;
48 }

```

打开页面删除一个部门，再添加一个部门，可以正常执行。

终章：原理篇

配置优先级

三种配置文件 properties yaml yaml 的优先级

声明三个配置文件，每个文件中写一个端口号，运行查看控制台信息

```
properties > yaml > yaml
```

SpringBoot除了支持配置文件属性配置，还支持Java系统属性和命令行参数的方式进行属性配置。

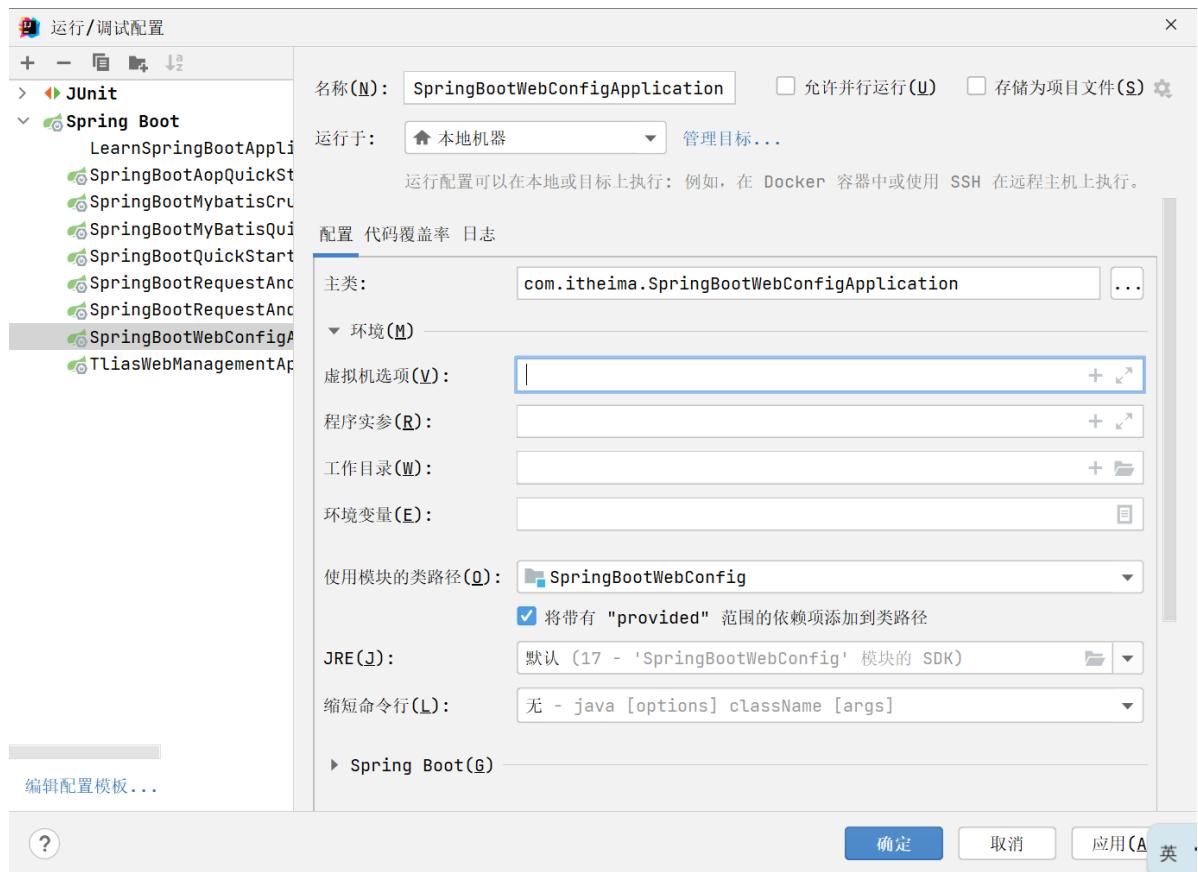
- Java系统属性

```
-Dserver.port=9000
```

- 命令行参数

```
--server.port=10010
```

IDEA中只需要编辑配置就可以使用这两种方式



运行可以知道命令行参数优先级比Java系统属性要高，且这两种都高于配置文件优先级

项目打包上线的话如何使用两个属性参数？通过命令行方式运行jar包，Java属性写在前边，命令行属性写在结尾

```
1 | java -Dserver.port=9000 -jar SpringBootWebConfig-0.0.1-SNAPSHOT.jar --  
  |   server.port=10010
```

获取bean

默认情况下SpringBoot启动时会自动创建IoC容器，在启动过程中会将bean对象创建好放在IoC容器中，程序依赖什么对象直接进行依赖注入就可以了。也可以通过Spring提供的方法主动从IoC容器获取bean对象

1. 根据name获取bean
2. 根据类型获取bean
3. 根据name和类型获取bean

以获取DeptController为例，首先要拿到IoC容器，SpringBoot环境直接使用自动注入获取IoC容器对象。

```
1 | @Autowired  
2 | ApplicationContext applicationContext; // IoC容器对象
```

三种获取方法：

```
1 | @Autowired  
2 | ApplicationContext applicationContext; // IoC容器对象  
3 |  
4 | //获取bean对象  
5 | @Test
```

```

6  public void testGetBean() {
7      //根据bean的名称获取
8      DeptController bean1 = (DeptController)
9      applicationContext.getBean("deptController");// bean对象的名称，没有指定名称时默认认为类名首字母小写
10     System.out.println(bean1);
11
12     //根据bean的类型获取
13     DeptController bean2 =
14     applicationContext.getBean(DeptController.class);
15     System.out.println(bean2);
16
17     //根据bean的名称 及 类型获取
18     DeptController bean3 = applicationContext.getBean("deptController",
19     DeptController.class);
20     System.out.println(bean3);
21 }

```

查看控制台发现三种方式获取出来的是同一个对象，说明IoC容器中bean对象只有1个（单例）

```

com.itheima.controller.DeptController@16ac4d3d
com.itheima.controller.DeptController@16ac4d3d
com.itheima.controller.DeptController@16ac4d3d

```

可以通过其他的注解将bean对象改成多例

注：上述在启动过程中会将bean对象创建好放在IOC容器中，仅针对默认情况下单例及非延迟加载的bean来说的。bean什么时候创建实际上还收到作用域和延迟初始化的影响。

bean作用域

Spring支持五种作用域，后三种在web环境中生效

- singleton：默认值，容器内同类型的bean只有一个实例（单例）
- prototype：每次使用bean时会创建新实例（非单例）
- request：每个请求范围内会创建新的实例
- session：每个会话范围内会创建新的实例
- application：每个应用范围内会创建新的实例

通过Spring提供的注解 @Scope 来配置作用域

先用个测试方法测试一下默认情况

```

1  @Test
2  public void testScope() {
3      for (int i = 0; i < 10; i++) {
4          DeptController bean =
5          applicationContext.getBean(DeptController.class);
6          System.out.println(bean);
7      }
8  }

```

查看控制台信息

```
2024-02-19 08:42:19.623 INFO 2424 — [           main] c.i.SpringbootWebConfig2ApplicationTests : Starti
2024-02-19 08:42:19.625 INFO 2424 — [           main] c.i.SpringbootWebConfig2ApplicationTests : No act
DeptController constructor ....
Logging initialized using 'class org.apache.ibatis.logging.stdout.StdoutImpl' adapter.
Property 'mapperLocations' was not specified.
com.itheima.service.impl.DeptServiceImpl@5d96bdf8

2024-02-19 08:42:24.229 INFO 2424 — [           main] c.i.SpringbootWebConfig2ApplicationTests
com.itheima.controller.DeptController@53e93fb7
```

可以发现单例bean对象在容器启动的时候就已经调用构造函数创建好并放入了IoC容器中。

可以第一次使用时在实例化，在类上使用注解 @Lazy

```
1  @Lazy
2  @RestController
3  @RequestMapping("/depts")
4  public class DeptController {
5      ...
6 }
```

再运行测试方法，看控制台信息

```
DeptController constructor ....
com.itheima.controller.DeptController@45cd8607
```

此时的构造函数调用就不在容器创建时了，而是在第一次调用前。而且只会实例化一次，后面直接从IoC容器中取单例对象。

修改一下DeptController的作用域，使用 prototype 设置为非单例的

```
1 @Scope("prototype")
2 @RestController
3 @RequestMapping("/depts")
4 public class DeptController {
5     ...
6 }
```

再运行测试方法，发现每次调用getBean方法都会实例化一个新的对象

```
DeptController constructor ....
com.itheima.controller.DeptController@4d67d5a4
DeptController constructor ....
com.itheima.controller.DeptController@a3b858f
DeptController constructor ....
com.itheima.controller.DeptController@600bbf9e
DeptController constructor ....
com.itheima.controller.DeptController@18d30e7
DeptController constructor ....
com.itheima.controller.DeptController@72b40f87
DeptController constructor ....
com.itheima.controller.DentController@2475fba3
```

注意：实际开发当中，大多数的bean是单例的，绝大部分bean不需要配置scope属性

第三方bean

之前配置的bean都是项目中自己定义的，声明bean只需要添加 @Component 及三个衍生注解就可以。还有一种情况是引入的第三方依赖当中提供的。

比如解析XML文件的依赖dom4j，引入对象SAXReader，使用他的方法来读取XML文件并解析。如果每次解析XML文件都创建一个SAXReader对象，是比较浪费资源的。现在完全可以将这个对象交给IoC容器管理。需要用到时直接注入就可以了。

需要注意的是，如果要管理的bean来自第三方依赖，是没办法用 @Component 及衍生注解声明bean的，需要用到 @Bean 注解。

使用非常简单，只需要定义一个方法，返回值是第三方的实例化对象，在方法上加 @Bean 注解，Spring 框架就知道需要将这个方法执行的返回值对象交给IoC容器管理，成为bean对象，要想在其他地方使用，直接注入就可以了。

在启动类中定义一个方法，加上 @Bean 注解

```
1 @Bean
2     public SAXReader saxReader() {
3         return new SAXReader();
4     }
```

在测试类中

```

1 //第三方bean的管理
2 @Test
3 public void testThirdBean() throws Exception {
4     SAXReader saxReader = new SAXReader();
5
6     Document document =
7         saxReader.read(this.getClass().getClassLoader().getResource("1.xml"));
8     Element rootElement = document.getRootElement();
9     String name = rootElement.element("name").getText();
10    String age = rootElement.element("age").getText();
11
12    System.out.println(name + " : " + age);
13 }

```

注意：第三方bean不建议定义在启动类中，建议单独定义一个配置类，在配置类中对第三方bean进行统一管理，配置类要使用 @Configuration 注解（配置拦截器也需要一个配置类，差不多的操作）

```

1 package com.itheima.config;
2
3 import org.dom4j.io.SAXReader;
4 import org.springframework.context.annotation.Bean;
5 import org.springframework.context.annotation.Configuration;
6
7
8 @Configuration
9 public class CommonConfig {
10
11     // 声明第三方bean
12     @Bean // 将当前方法的返回值交给IoC容器管理，成为IoC容器的bean对象
13     public SAXReader saxReader() {
14         return new SAXReader();
15     }
16 }

```

在试运行单元测试，正常运行说明第三方bean没问题。

自己声明的bean对象，直接加注解 @Component @Controller @Service @Repository，也可以通过这些注解给bean指定名字，如果没指定默认是类名首字母小写，那第三方bean的名称如何指定？

@Bean 注解中有两个属性 value name 都可以指定bean的名称，因为两个属性都有别名

@AliasFor("name") @AliasFor("value")，这两个属性实际上是一样的。

注意：实际上第三方的bean我们也不会去指定名称，如果不指定默认是方法名。

在测试类中测试一下，通过方法名（也就是bean名）来尝试从IoC容器中获取bean

```

1 @Test
2 public void testGetBean2() {
3     Object saxReader = applicationContext.getBean("saxReader");
4     System.out.println(saxReader);
5 }

```

可以看到已经拿到了bean对象

org.dom4j.io.SAXReader@559d19c

改成reader再试一下

```
1  @Test
2  public void testGetBean2() {
3      Object saxReader = applicationContext.getBean("reader");
4      System.out.println(saxReader);
5
6  }
```

此时就会报错，容器中并没有一个叫 reader 的对象

把方法名也改成 reader

```
1  @Bean // 将当前方法的返回值交给IoC容器管理，成为IoC容器的bean对象
2  // 可以通过bean注解的value/name属性指定名称，如果未指定默认是方法名
3  public SAXReader reader() {
4      return new SAXReader();
5  }
```

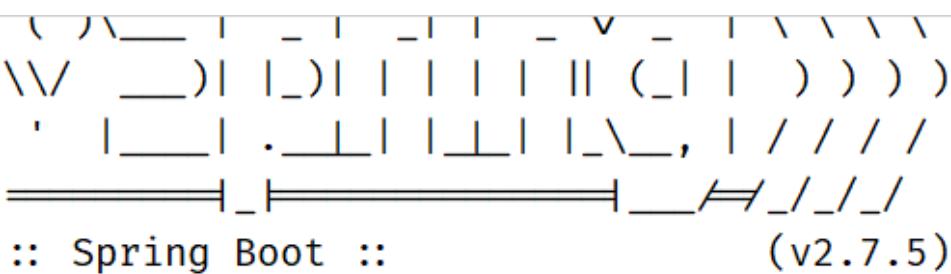
再运行测试方法，此时就可以获得一个叫 reader 的bean对象，也就是说IoC容器中有一个叫 reader 的 bean对象。

如果声明第三方bean时需要依赖注入该怎么做？自定义的bean对象是直接使用 `@Autowired` 进行注入，第三方bean直接在定义bean的方法中定义形参就可以，Spring框架就会自动装配，根据类型找到对象完成注入操作。

测试一下，在bean方法中定义形参，然后运行任意一个测试方法，测试方法运行时会加载整个Spring环境，此时就会调用这个方法并完成注入操作。

```
1  @Bean // 将当前方法的返回值交给IoC容器管理，成为IoC容器的bean对象
2  // 可以通过bean注解的value/name属性指定名称，如果未指定默认是方法名
3  public SAXReader reader(DeptService deptService) {
4      System.out.println(deptService);
5      return new SAXReader();
6  }
```

在控制台中看加载Spring时输出的日志信息，这个输出的对象就是我们在定义第三方bean方法时传入的对象



```
2024-02-19 11:47:02.923  INFO 18872 — [  
2024-02-19 11:47:02.925  INFO 18872 — [  
Logging initialized using 'class org.apache.ibatis.Property 'mapperLocations' was not specified.  
com.itheima.service.impl.DeptServiceImpl@7f37b6d9
```

SpringBoot原理

Spring家族

所有的框架都基于基础的框架：`Spring Framework`，如果基于Spring框架开发会很繁琐，依赖配置很繁琐，配置文件繁琐。Spring4.0后推出SpringBoot来**简化**Spring框架的开发，Spring官方现在推荐基于SpringBoot构建java项目。

SpringBoot更简单快捷是因为底层提供了两个非常重要的功能：起步依赖、自动配置。通过起步依赖可以大大简化pom文件中的依赖配置，可以解决Spring框架中依赖配置繁琐的问题。通过自动配置可以简化bean的声明和bean的配置。只需要引入所需的起步依赖，那开发时所有的配置都已经有了，直接使用就可以。

SpringBoot的原理实际上就是起步依赖和自动配置的原理。

起步依赖原理

如果用的是Spring框架进行的开发，就需要引入web开发需要的依赖。如 `spring-webmvc servlet-api jackson aspectjweaver`，而且引入的依赖版本还需要匹配，否则可能出现**版本冲突**的问题。

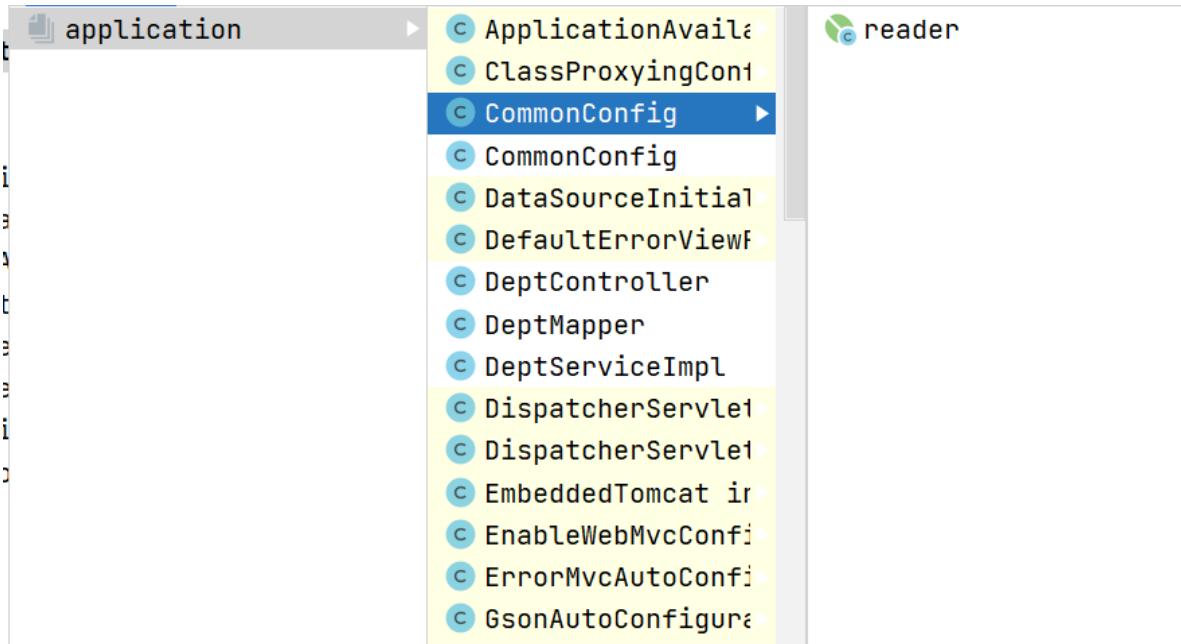
如果使用SpringBoot，只需要引入一个依赖，就是web开发的起步依赖 `spring-boot-starter-web`，如果想进行AOP开发也需要一个依赖 `spring-boot-starter-aop`。实际上就是因为**Maven的依赖传递**，我们只需要引入一个起步依赖，就会将所有需要的依赖都加入进来。比如web起步依赖中就集成了所有web开发的常见依赖，只需要引入这一个依赖，其他的依赖都会通过依赖传递传递进来。（a引入b，b引入c，c引入d，引入a之后bcd三项也就引入进来了）

起步依赖的原理就是Maven的依赖传递。

自动配置概述

自动配置是指SpringBoot启动时，除了自定义的Controller Service dao这些bean对象外，SpringBoot还会将一些内置的配置类和bean对象都创建好放入IoC容器中。使用框架一些功能时，就不需要手动声明bean对象，直接使用就可以。

把项目运行起来后看控制台中的 `Actuator` 里的 bean 里面列出了 SpringBoot 的所有 bean 对象，比如这里有两个 `CommonConfig` 其中一个是我们注入的第三方 bean 对象 `SAXReader`，还有一个是 `CommonConfig` 自身，这是因为 `@Configuration` 注解中包含了 `@Component` 注解，所以他也会被作为 bean 对象。



黄底的配置类都是SpringBoot启动时加载进来的配置类，来尝试使用一下谷歌提供的 gson 对象

```
1     @Autowired
2     private Gson gson;
3
4     @Test
5     void testJson() {
6         String s = gson.toJson(Result.success());
7         System.out.println(s);
8     }

```

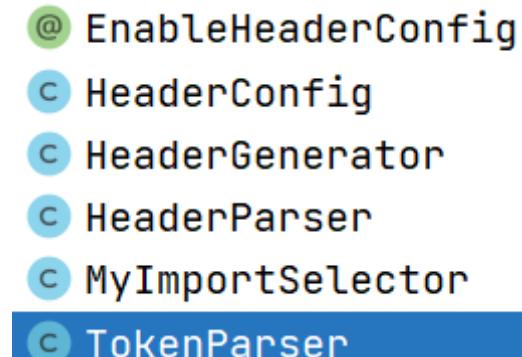
发现IDEA提示报错：无法自动装配gson，找不到 Gson 类型的bean。但运行测试发现可以正常执行，也可以正常输出结果。

这个bean对象是哪来的？SpringBoot项目启动的时候自动配置好的

自动配置原理

就是分析SpringBoot项目中引入依赖后，如何将依赖jar包中的bean和配置类直接加载到当前项目的Spring容器中。

复制资料里的 `itheima-utils` 模块，模拟第三方提供的依赖，模块内的类如下



在 webConfig2 中引入依赖的坐标

```
1      <!--引入第三方提供的依赖-->
2      <dependency>
3          <groupId>com.example</groupId>
4          <artifactId>iitheima-utils</artifactId>
5          <version>0.0.1-SNAPSHOT</version>
6      </dependency>
```

Spring的IoC容器中是否存在这几个引入的bean?

写三个测试方法试一下

```
1  @Autowired
2  private ApplicationContext applicationContext;
3
4  // 获取依赖中的bean对象
5  @Test
6  void testTokenParser() {
7      System.out.println(applicationContext.getBean(TokenParser.class));
8  }
9
10 @Test
11 void testHeaderParser() {
12     System.out.println(applicationContext.getBean(HeaderParser.class));
13 }
14
15 @Test
16 void testHeaderGenerator() {
17
18     System.out.println(applicationContext.getBean(HeaderGenerator.class));
19 }
```

运行发现三个方法都报错了，错误提示为找不到bean定义

说明引入进来的第三方依赖的bean没有生效。这是因为IoC容器的包扫描机制

以前讲过 @Component 注解想要生效需要被Spring组件扫描到，也提到 @SpringBootApplication 具有包扫描作用，但是扫描范围是当前包 com.iitheima 及其子包，这样是扫描不到第三方的 com.example 包的。

如何解决？

- 方案1：为启动类通过 @ComponentScan 手动指定扫描哪些地方的包，因为手动指定会覆盖掉默认的扫描范围，所以还需要把当前包也添加到扫描范围内

```
1  @ComponentScan({"com.iitheima", "com.example"})
2  @SpringBootApplication
3  public class SpringbootwebConfig2Application {
4
5      public static void main(String[] args) {
6          SpringApplication.run(SpringbootwebConfig2Application.class,
7          args);
8      }
9  }
```

此时再运行测试方法就可以拿到bean对象了，但很显然这样做是很笨的。之前用SpringBoot进行案例开发，也引入了很多第三方依赖，**SpringBoot并没有让我们配置组件扫描去扫描第三方依赖所在的包。**

- 方案2：通过给启动类加`@Import`注解导入。使用import导入的类会被Spring加载到IoC容器中，主要有三种方式

1. 导入普通类，不需要加任何注解，通过`@Import`将普通类导入进来后，就可以直接给IoC容器管理。
2. 导入配置类，配置类中所有的bean对象都会加载到IoC容器当中
3. 导入`ImportSelector`接口的实现类，

注意：`@Import`注解的参数类型是数组，所以需要用大括号括起来

先导入一下普通类`TokenParser`试一试

```
1 @SpringBootApplication
2 @Import({TokenParser.class}) // 导入普通类，交给IOC容器管理
3 public class SpringbootwebConfig2Application {
4
5     public static void main(String[] args) {
6         SpringApplication.run(SpringbootwebConfig2Application.class,
7             args);
8     }
9 }
```

运行获取`TokenParser`的测试方法，可以成功获取到`TokenParser`类型的bean对象，说明Spring容器中已经有了这个类型的bean对象

导入配置类`HeaderConfig`试一下

```
1 @Import({HeaderConfig.class}) // 导入配置类，交给IOC容器管理
2 @SpringBootApplication
3 public class SpringbootwebConfig2Application {
4
5     public static void main(String[] args) {
6         SpringApplication.run(SpringbootwebConfig2Application.class,
7             args);
8     }
9 }
```

这个配置类中声明的bean是`headerparser`和`headergenerator`，运行两个测试方法，两个bean对象都可以获取到，都没有问题

导入一个`ImportSelector`接口的实现类，先查看一下他的源码

```
public interface ImportSelector {
}

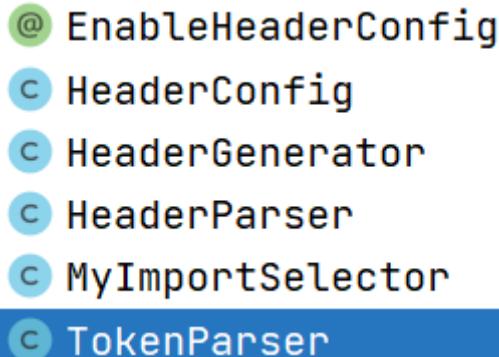
Select and return the names of which class(es) should be
imported based on the AnnotationMetadata of the importing
@Configuration class.

返回值: the class names, or an empty array if none

String[] selectImports(AnnotationMetadata importingClassMetadata);
```

里面有一个方法`selectImports`返回值是一个字符串数组，封装的信息是全类名，其实就是我们需要将哪些类交给IoC容器管理，可以将全类名封装到数组中直接返回。

模拟第三方依赖中已经有了一个`ImportSelector`的实现类，已经实现了`selectImports`方法



```

1 public class MyImportSelector implements ImportSelector {
2     public String[] selectImports(AnnotationMetadata
3         importingClassMetadata) {
4         return new String[]{"com.example.HeaderConfig"};
5     }

```

这里要的是**全类名字符串**，将来就可以把要加载的类定义在一份文件中，最终只需要读取文件，将文件中的字符串读取出来封装到数组当中就可以了。

这里指定的是HeaderConfig，相当于将这个类交给IoC容器管理。

在启动类中指定 ImportSelector 接口的实现类

```

1 @Import({MyImportSelector.class}) // 导入ImportSelector接口的实现类
2 @SpringBootApplication
3 public class SpringbootwebConfig2Application {
4
5     public static void main(String[] args) {
6         SpringApplication.run(SpringbootwebConfig2Application.class,
7             args);
8     }
9 }

```

运行headerparser和headergenerator的测试方法，都可以获得bean对象。

如果基于这三种方式来完成自动配置，将来引入一个第三方依赖还得知道要导入哪些配置类和哪些bean对象，还是很笨很繁琐。要使用第三方依赖，到底要导入哪些配置类和哪些bean，显示是第三方依赖自身最清楚。基于这种想法，我们就不用再启动类上自己来指定要导入哪些bean对象和配置类了，让第三方依赖自己来指定。常见方案是第三方依赖给我们提供一个注解，一般都是 @Enablexxx 注解封装 @Import 注解，在import注解中指定要导入哪些bean或者是哪些配置类。

4. 终极方案： @Enablexxx 注解，封装 @Import注解

模拟的第三方依赖中提供了这样一个注解

```

1 @Retention(RetentionPolicy.RUNTIME)
2 @Target(ElementType.TYPE)
3 @Import(MyImportSelector.class)
4 public @interface EnableHeaderConfig {
5 }

```

import中指定的时接口选择器的实现类，在实现类中指定了要导入的配置类和bean。想开启第三方依赖的自动配置功能，只需要在启动类中添加定义的注解 @EnableHeaderConfig，加上这个注解就相当于我们加上了import注解，只不过import中不需要我们自己来指定要导入的配置类和bean对象。

```
1 | @EnableHeaderConfig
2 | @SpringBootApplication
3 | public class SpringbootwebConfig2Application {
4 |
5 |     public static void main(String[] args) {
6 |         SpringApplication.run(SpringbootwebConfig2Application.class, args);
7 |     }
8 | }
```

这种方式也是SpringBoot采用的方式

源码跟踪分析

源码是非常巨大的，应该如何入手分析自动配置原理？从启动类上的注解入手，`@SpringBootApplication`是SpringBoot框架中最重要的一个注解。

直接从这个注解入手

- 源码跟踪技巧

抓住关键点，找到核心流程，不要一句一句的看，一个方法一个方法的研究。先对整个流程有个认识，再去认识细节。

打开启动类注解的源码

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan(excludeFilters = { @Filter(type = FilterType.CUSTOM, classes = TypeExcludeFilter.class),
    @Filter(type = FilterType.CUSTOM, classes = AutoConfigurationExcludeFilter.class) })
public @interface SpringBootApplication {
```

四个元注解不需要关注，
只需要看下边三个注解

- `SpringBootConfiguration`，没有做什么事，只是封装了`@Configuration`注解，`@Indexed`用来加速启动（不用管）。那整个注解的作用就是声明当前类是个配置类。所以我们之前才可以在启动类中声明第三方的bean对象。

```
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Configuration
@Indexed
public @interface SpringBootConfiguration {
    @AliasFor(
        annotation = Configuration.class
    )
    boolean proxyBeanMethods() default true;
}
```

- `@ComponentScan`, 进行组件扫描的注解
- `@EnableAutoConfiguration`, 自动配置的核心注解。之前还提到过enable开头的注解一般封装的是Import注解, 通过Import来导入指定的bean或者是配置类。这个注解中果然有一个Import注解, 注解中导入的是 ImportSelector 接口的实现类

```

@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@AutoConfigurationPackage
@Import(AutoConfigurationImportSelector.class)
public @interface EnableAutoConfiguration {

```

打开这个实现类看一下, 这个类实现了 `DeferredImportSelector`, 这是一个接口, 父接口就是 `ImportSelector`, 上一节提到 `ImportSelector` 中有一个重要的方法, `selectImports` 方法, 方法封装的是要导入到IoC容器中的类的全类名。

```
public class AutoConfigurationImportSelector implements DeferredImportSelector, BeanClassLoaderAware,
    ResourceLoaderAware, BeanFactoryAware, EnvironmentAware, Ordered {
```

在实现类中找一下这个方法, 关注这个返回值到底封装了哪些类的全类名

```

@Override
public String[] selectImports(AnnotationMetadata annotationMetadata) {
    if (!isEnabled(annotationMetadata)) {
        return NO_IMPORTS;
    }
    AutoConfigurationEntry autoConfigurationEntry = getAutoConfigurationEntry(annotationMetadata);
    return StringUtils.toStringArray(autoConfigurationEntry.getConfigurations());
}

```

返回值是调用的是entry对象的getConfigurations方法, entry由 `getAutoConfigurationEntry` 获取到, 跟着返回值继续往上层走, 跳到 `getAutoConfigurationEntry` 这个方法中, 看这个方法的返回值封装了什么信息

```

protected AutoConfigurationEntry getAutoConfigurationEntry(AnnotationMetadata annotationMetadata) {
    if (!isEnabled(annotationMetadata)) {
        return EMPTY_ENTRY;
    }
    AnnotationAttributes attributes = getAttributes(annotationMetadata);
    List<String> configurations = getCandidateConfigurations(annotationMetadata, attributes);
    configurations = removeDuplicates(configurations);
    Set<String> exclusions = getExclusions(annotationMetadata, attributes);
    checkExcludedClasses(configurations, exclusions);
    configurations.removeAll(exclusions);
    configurations = getConfigurationClassFilter().filter(configurations);
    fireAutoConfigurationImportEvents(configurations, exclusions);
    return new AutoConfigurationEntry(configurations, exclusions);
}

```

方法直接返回了一个entry对象, 里面有configurations和exclusions两个参数。

此时我们知道 `selectImports` 方法最终调用的是entry对象中的 `getConfigurations` 方法, 所以现在的关键点在于这个configurations, 可以看到configurations是个List集合, 通过调用方法 `getCandidateConfigurations` 得到的。现在进入这个方法看一下返回值封装了哪些数据

```

protected List<String> getCandidateConfigurations(AnnotationMetadata metadata, AnnotationAttributes attributes) {
    List<String> configurations = new ArrayList<>(
        SpringFactoriesLoader.loadFactoryNames(getSpringFactoriesLoaderFactoryClass(), getBeanClassLoader()))
    ImportCandidates.load(AutoConfiguration.class, getBeanClassLoader()).forEach(configurations::add);
    Assert.notEmpty(configurations,
        message: "No auto configuration classes found in META-INF/spring.factories nor in META-INF/spring/org."
        + "are using a custom packaging, make sure that file is correct.");
    return configurations;
}

```

这个集合调用了一个方法 `loadFactoryNames`，方法中做了什么我们也不知道，但这里还有一个 `Assert` 断言，用来判断 `configurations` 集合是否为空的，如果是空的就会提示后边这个信息，意思是没有任何一个自动配置的类在文件中被发现，这里有两个文件第一个是 `META-INF/spring.factories`，另一个文件是 `META-INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports` 也就是说 SpringBoot 在启动时会自动加载这两个文件中所配置的信息，加载出来后就会封装到 `List` 集合中，最终将集合中的内容封装到字符串数组中，这些内容就是要加载到 IoC 容器中的 bean 和配置类。现在的关键就是要找到这两份文件。

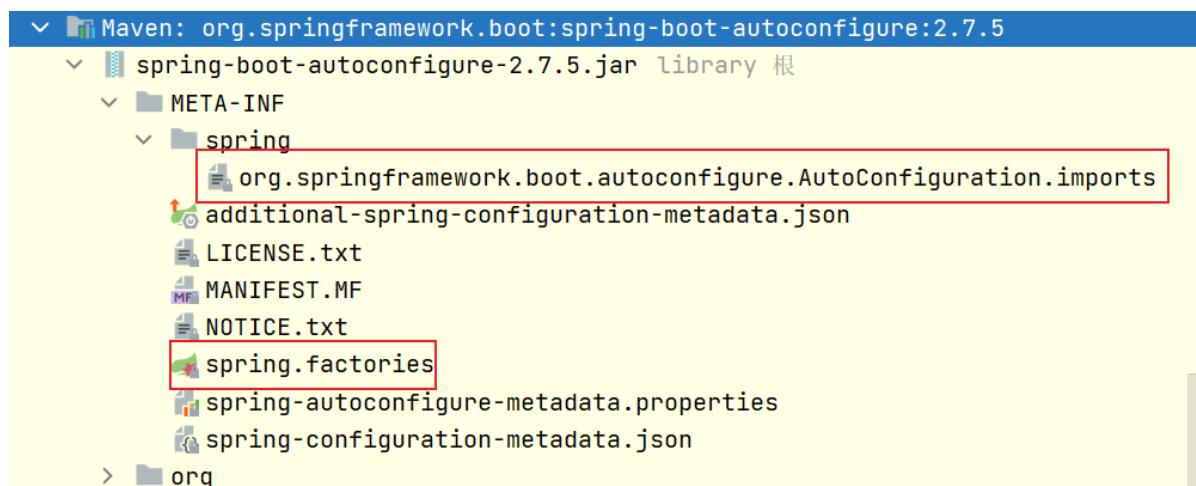
其实这一类文件在起步依赖中基本都有，从 Maven 侧边栏中选一个项目，点开依赖项，可以看到依赖中都有这样一项依赖

```

{
    "org.springframework.boot:spring-boot-starter-web:3.2.2",
    "org.springframework.boot:spring-boot-starter:3.2.2",
        "org.springframework.boot:spring-boot:3.2.2",
        "org.springframework.boot:spring-boot-autoconfigure:3.2.2" // 盒选
        "org.springframework.boot:spring-boot-starter-logging:3.2.2",
            "jakarta.annotation:jakarta.annotation-api:2.1.1",
            "org.springframework:spring-core:6.1.3 (omitted for duplicate)",
            "org.yaml:snakeyaml:2.2",
        "org.springframework.boot:spring-boot-starter-json:3.2.2",
        "org.springframework.boot:spring-boot-starter-tomcat:3.2.2",
        "org.springframework:spring-web:6.1.3",
        "org.springframework:spring-webmvc:6.1.3",
    "org.mybatis.spring.boot:mybatis-spring-boot-starter:3.0.3",
        "org.springframework.boot:spring-boot-starter:3.2.2 (omitted for duplicate)" // 盒选
        "org.springframework.boot:spring-boot-starter-jdbc:3.2.2" // 盒选
        "org.mybatis.spring.boot:mybatis-spring-boot-autoconfigure:3.0.3" // 盒选
            "org.mybatis:mybatis:3.5.14",
            "org.mybatis:mybatis-spring:3.0.3"
}

```

现在找到这两个 jar 包看看里边都有什么信息，在项目侧边栏中点开外部库，找到 SpringBoot 官方的 `autoconfigure`



这就是我们要找的两个文件

打开 `spring.factories` 看看，里面有很多信息，都是类的全类名

`autoconfigure.AutoConfiguration.imports` 里面也都是类的全类名

配置文件最终会被读取出来，通过`Import`注解加载到IoC容器当中，交给容器管理。这些类的类名后缀都是`AutoConfiguration`，所以这些类也称为自动配置类。

打开Gson的自动配置类，看看是如何实现的

```
@AutoConfiguration  
@ConditionalOnClass(Gson.class)  
@EnableConfigurationProperties(GsonProperties.class)  
public class GsonAutoConfiguration {
```

打开`@AutoConfiguration`注解，看到底层封装了一个注解`@Configuration`

回到Gson的配置类，看到一个`gson`方法，返回值类型是`Gson`，并且加了`@Bean`注解，此时这个类被加载到IoC容器中，这个bean对象自动就有了。所以我们才能使用`Autowired`自动注入`Gson`对象直接使用。

`@Bean`

```
@ConditionalOnMissingBean  
public Gson gson(GsonBuilder gsonBuilder) {  
    return gsonBuilder.create();  
}
```

注意：`spring.factories`是早期SpringBoot配置所加载的文件，在SpringBoot2.7.0之后就转为使用`AutoConfiguration.imports`。2.7.x版本还会兼容`.factories`文件，到SpringBoot3.0+就会彻底移除

最后一个问题是：配置文件中有很多的自动配置类，每个配置类中可能会有很多的bean，所有的bean都会注册到IoC容器中吗？NO！可以看到方法上有个注解`@ConditionalOnMissingBean`，除此之外还有很多`Conditional`开头的注解，只有满足一定条件之后才会将bean注册到容器中。

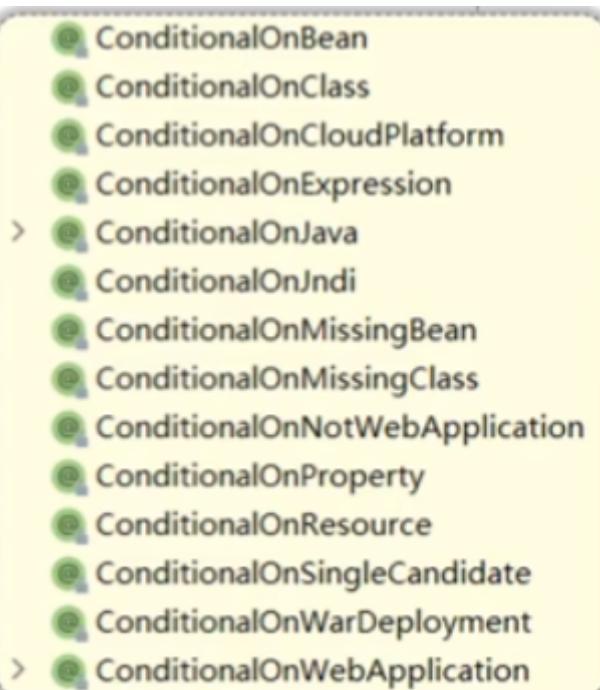
@Conditional

条件装配注解，按照一定条件进行判断，在满足条件后才会注册对应bean对象到IoC容器中。

可以加在方法或类上

`@Conditional`是一个父注解，衍生出大量子注解，主要记录这三个：

- `@ConditionalOnClass`: 判断是否有对应字节码文件，有则注册到IoC容器中。
- `@ConditionalOnMissingBean`: 判断有无对应的bean（类型或名称），才注册bean到IoC容器
- `@ConditionalOnProperty`: 判断配置文件中有无对应属性和值，才注册到IoC容器



回到 webConfig2 模块，在HeaderConfig中声明HeaderParser类型的bean时，加上条件判断注解

先来一个 @ConditionalOnClass，作用是判断当前环境当中是否有对应的字节码文件，存在则注册对象到IoC容器中，两种方式指定字节码文件 value, name, 这里使用name+全类名的方式。

```
1 package com.example;
2
3 import org.springframework.boot.autoconfigure.condition.ConditionalOnClass;
4 import org.springframework.context.annotation.Bean;
5 import org.springframework.context.annotation.Conditional;
6 import org.springframework.context.annotation.Configuration;
7
8 @Configuration
9 public class HeaderConfig {
10
11     @Bean
12     @ConditionalOnClass(name = "io.jsonwebtoken.Jwts")
13     public HeaderParser headerParser(){
14         return new HeaderParser();
15     }
16
17     @Bean
18     public HeaderGenerator headerGenerator(){
19         return new HeaderGenerator();
20     }
21 }
22 }
```

写好后代表容器启动时会自动判断当前环境中是否存在 Jwts 这个类，如果存在他才会将这个 bean 对象注册到 IoC 容器当中来。pom 文件中引入了 jwt 的依赖，代表当前环境中有 Jwts 这个类，那么就会声明 bean 对象。

运行测试方法，可以正常获取到 HeaderParser 这个 bean 对象。现在将 pom 文件中引入的 jwt 依赖注释掉并重新加载，现在环境中就没有 Jwts 这个类了，那 HeaderParser 也应该不会创建。再来运行测试方法。

```
<!-- JWT 令牌 -->
<!-- <dependency> -->
<!--     <groupId>io.jsonwebtoken</groupId> -->
<!--     <artifactId>jjwt</artifactId> -->
<!--     <version>0.9.1</version> -->
<!-- </dependency> -->
```

可以看到果然报错了，提示找不到HeaderParser这样一个bean

```
NoSuchBeanDefinitionException: No qualifying bean of type 'com.example.HeaderParser'
```

基于这个注解可以轻松实现自动配置功能，只要引入了依赖，bean自动就配置好了，不需要任何的操作。没有引入依赖，bean就不会配置。

再来`@ConditionalOnMissingBean`，如果什么属性都没有指定代表如果不存在该类型的bean，才会将bean加入到IoC容器中。该类型就是我们声明的HeaderParser这个类型，Spring自动判断容器中有没有这个bean，如果没有就声明出来，如果有就不在声明这个bean。

```
1  @Bean
2 //  @ConditionalOnClass(name = "io.jsonwebtoken.Jwts") // 环境中存在指定类，才会将bean加入到IoC容器中
3  @ConditionalOnMissingBean // 不存在该类型的bean，才会将bean加入到IoC容器中
4  public HeaderParser headerParser(){
5      return new HeaderParser();
6 }
```

运行测试方法，可以从容器中获取到bean

也可以根据指定的类型或名称来判断，此时就需要声明注解中的属性。**指定类型：value 指定名称：name**

通常用来设置一个默认的bean对象，如果用户引入这个依赖后自己定义了这个类型的bean，此时就用他自定义的，默认的不会生效。如果没有自定义还想使用，就使用默认提供的bean。

最后`@ConditionalOnProperty`，和配置文件中的属性有关。在注解中指定两个属性`name`和`havingValue`，`name`指定配置项的名字，`havingValue`指定配置项的值

```
1  @Bean
2 //  @ConditionalOnClass(name = "io.jsonwebtoken.Jwts") // 环境中存在指定类，才会将bean加入到IoC容器中
3 //  @ConditionalOnMissingBean // 不存在该类型的bean，才会将bean加入到IoC容器中 指定类型：value 指定名称：name
4  @ConditionalOnProperty(name = "name", havingValue = "itheima")
5  public HeaderParser headerParser(){
6      return new HeaderParser();
7 }
```

配置了`name`和`havingValue`，就会自动去判断当前环境的配置文件中是否存在指定属性和属性值。在yml配置文件中添加`name: itheima`，运行测试方法就可以获得bean。把值改成`itheima2`再运行单元测试，此时就没有HeaderParser这个类型的bean了。

常用于整合第三方插件的bean声明

自定义starter

不是所有的第三方技术都提供了与SpringBoot整合的起步依赖，没有提供起步依赖的技术就需要我们自己来一步步的操作，而这些技术又很通用，在各个项目中都可能使用到，不能每次都从头开始一步一步操作。实际开发中经常会定义一些公共组件，给各团队使用。在SpringBoot中会将公共组件封装为starter。

SpringBoot官方提供的起步依赖：`spring-boot-starter-xxx`

第三方提供的起步依赖：`xxx-spring-boot-starter`

打开Mybatis的起步依赖会发现，起步依赖中一个代码都没有，起步依赖只做了一个事情就是将Mybatis开发所需要的依赖配置在了pom文件中。在这些依赖中有一个比较特殊：`mybatis-spring-boot-autoconfigure`，在这项依赖中提供了一些自动配置类。

自定义起步依赖需要两个模块：

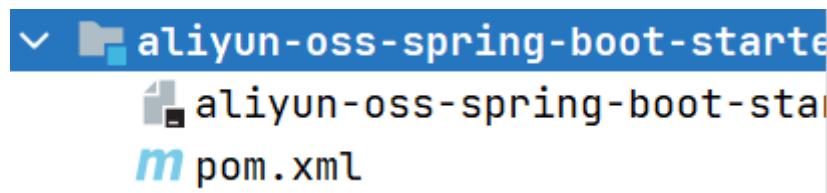
1. 依赖管理，将所有需要的依赖定义在起步依赖中
2. 自动配置，最终在starter中将autoconfigure引入进来，将来在项目中进行功能开发只需要引入一个起步依赖就可以。他会将自动配置的依赖传递下来。

自定义aliyun-oss-spring-boot-starter，完成阿里云OSS工具类的自动配置

步骤

1. 创建starter模块，将依赖管理起来
2. 创建自动配置模块，在starter中引入自动配置
3. 定义自动配置功能，定义自动配置文件xxx.imports

创建 `aliyun-oss-spring-boot-starter` 模块，只需要留下pom文件管理依赖，其他的都可以删掉



pom文件中只需要留下SpringBoot的起步依赖

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4     xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
5       https://maven.apache.org/xsd/maven-4.0.0.xsd">
6   <modelVersion>4.0.0</modelVersion>
7   <parent>
8     <groupId>org.springframework.boot</groupId>
9     <artifactId>spring-boot-starter-parent</artifactId>
10    <version>3.2.2</version>
11    <relativePath/> <!-- lookup parent from repository -->
12  </parent>
13  <groupId>com.aliyun.oss</groupId>
14  <artifactId>aliyun-oss-spring-boot-starter</artifactId>
15  <version>0.0.1-SNAPSHOT</version>
16
17  <properties>
18    <java.version>17</java.version>
19  </properties>
```

```

19 <dependencies>
20   <dependency>
21     <groupId>org.springframework.boot</groupId>
22     <artifactId>spring-boot-starter</artifactId>
23   </dependency>
24 </dependencies>
25
26 </project>

```

创建 aliyun-oss-spring-boot-autoconfigure 模块，因为是第三方模块，是要在其他项目中依赖（使用）这个模块的，所以启动类、配置文件、测试类都用不上，直接都删掉。

在start的pom中引入autoconfigure，接下来在autoconfigure中完成自动配置操作。目标是依赖引入后直接注入就可以使用，所以需要在自动配置模块中要将工具类配置好。

在案例模块中把阿里云OSS相关的工具类先复制过来，然后将所需的依赖也复制过来。接下来根据类中的报错一个一个的解决。

首先是MultipartFile报错，这是Spring中提供的web开发的类，需要引入web开发的起步依赖

然后是 aliossProperties 中方法报错，可以引入Lombok并添加注解，或者生成一下get/set方法。

现在类上的Component注解已经没用了，因为将来在SpringBoot项目中并不会让用户去扫描这个包，这些注解就都没用了，直接删掉。Autowired也删掉。

现在Utils已经不是Spring容器中的bean了，但我们的需求是引入依赖之后注入Utils就可以直接使用。就是说最终还需要将AliOSSUtils交给IoC容器管理，此时就需要定义自动配置类。既然是配置类，就需要注解 @Configuration，在配置类中定义一个方法来配置AliOSSUtils。因为Utils里使用到了Properties，如果不给Properties赋值在使用时就会报空指针异常，所以需要给成员变量Properties赋值，先设置好get/set方法，在定义对象时为变量赋值。值从哪来？这个对象是Spring加载配置文件当中 aliyun.oss 前缀的配置项最终封装成一个Properties的bean

```

@ConfigurationProperties(prefix = "aliyun.oss")
public class AliOSSProperties {
    private String endpoint;
    private String accessKeyId;
    private String accessKeySecret;
    private String bucketName;

    public String getEndpoint() {
        return endpoint;
    }

    public void setEndpoint(String endpoint) {
        this.endpoint = endpoint;
    }
}

```

此时这里还在报错，信息如下：

未通过 `@EnableConfigurationProperties` 注册、标记为 `Spring` 组件或通过 `@ConfigurationPropertiesScan` 扫描

报错是因为现在这个类已经不是一个bean了，（Component注解已经被去掉了）

可以直接使用 `@EnableConfigurationProperties` 将这个类导入到IoC容器中成为bean对象，`Enable` 开头的注解可以想到底层封装了Import注解。注意：这个注解只能加载一个声明 `@Bean` 的方法上或一个配置类上面。

现在需要这个bean怎么办？前面讲到声明第三方bean的时候需要注入某个对象，只需要在方法形参上指定参数就可以。

```
1 @Configuration
2 @EnableConfigurationProperties(AliOSSProperties.class) // 这样就相当于把这个类导
   入到IoC容器成为IoC容器中的bean
3 public class AliOSSAutoconfiguration {
4
5     @Bean
6     public Aliossutils aliossutils(AliOSSProperties aliOSSProperties) {
7         Aliossutils aliossutils = new Aliossutils();
8         aliossutils.setAliOSSProperties(aliOSSProperties);
9
10        return aliossutils;
11    }
12 }
```

这样自动配置类就配置好了

最后还需要写配置文件，找个SpringBoot官方包复制一下 `.imports` 的文件名，把自动配置类的全类名写进去即可。

导入资料里的测试工程，目标是引入刚写好的依赖，使用依赖中的类上传文件到阿里云OSS。

引入依赖，修改yml配置文件使Properties可以读取到阿里云OSS的配置信息

Web总结

三层架构

Controller Service Dao

业务处理之前的通用业务处理：借助JavaWeb过滤器或Spring拦截器

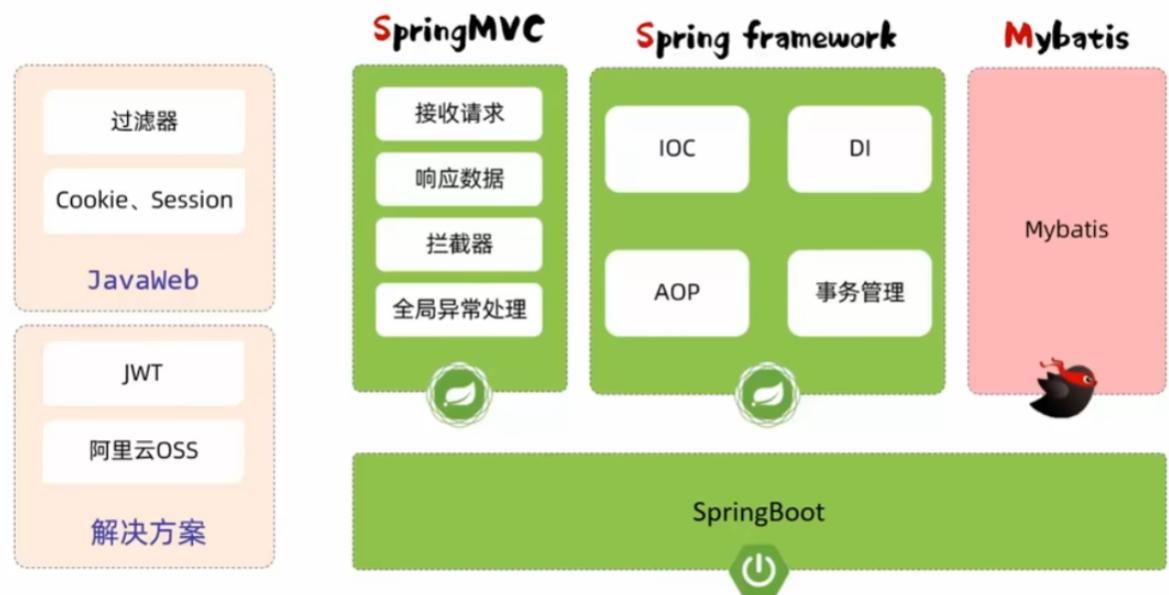
层与层解耦：IoC与DI

AOP、事务管理、全局异常处理、Cookie、Session、JWT令牌、阿里云OSS、MyBatis



在Spring Framework中对web程序开发提供了很好的支持，如拦截器，全局异常处理器。Spring框架中的web开发模块也称为springMVC框架。

SpringMVC实际上就是Spring框架的一部分，是Spring框架当中提供的web开发模块，是用来简化原始的Servlet程序开发的。Controller中接受请求，拦截数据都是springMVC提供的功能



直接基于springMVC+Spring Framework+MyBatis=SSM，基于SSM的开发实际上是很繁琐的。

至此，web后端开发的所有内容就结束了。