简介

- SpringMVC是spring的web模块，用来开发Web应用
- SpringMVC应用最终作为B/S、C/S模式下的Server端
- Web应用的核心就是处理Http请求响应

两种开发模式

- 前后不分离开发：

  浏览器/客户端发送请求，访问指定路径资源-->服务端进行业务处理并用数据组装页面（服务端渲染）-->服务端返回带数据的完整页面给浏览器/客户端

- 前后分离开发：

  前端和后端作为独立的部分进行开发，前端专注于用户界面的设计和交互，后端则专注于提供API接口和服务。前端通过AJAX或其他方式调用后端提供的RESTful API获取数据，动态更新页面内容。

  前后端分离其实是数据和页面分离，数据由后端负责，页面由前端负责

Helloworld

```
@Controller//告诉spring这是一个控制器（处理请求的组件）
public class HelloController {

    @ResponseBody//把返回值放到响应体中
    @RequestMapping("/hello")
    public String hello() {
        return "hello,springmvc! 你好!";
        //如果只用@RequestMapping注解，springmvc默认认为返回值是页面的地址去跳转
    }

}
```

注：如果类里的所有方法都想要将返回值放进响应体而不是当作地址，可以在类上标注@ResponseBody。而@Controller+@ResponseBody又可以用@RestController代替，所以以后只要标注@RestController就行了。

在启动ioc容器后，在浏览器访问localhost:8080/hello就能得到`hello,springmvc! 你好!`

想更改端口，在配置文件里写`server.port=`

优点：
1.tomcat不用整合

2.servlet开发变得简单，不用实现任何接口

3.自动解决了乱码等问题

@RequestMapping

路径映射

@RequestMapping的参数可以使用通配符

*:匹配任意多个字符

**：匹配任意多层路径

？：匹配任意单个字符

通配符存在的情况下，有多个方法能匹配请求，精确优先，完全匹配>?>*>**

```
    @ResponseBody
    @RequestMapping("/hello")
    public String hello() {
        return "hello,springmvc! 你好!";

    }
    @ResponseBody
    @RequestMapping("/hell?")
    public String hello1(){
        return "hello1,springmvc!";
    }
```

比如如上两个方法，在请求localhost:8080/hello时，执行的方法是第一个。

注意：不能写两个相同的路径映射，不然会报错

请求限定

@RequestMapping的参数：

- value： 路径映射
- method：限定请求方法（get，post，put等）
- params：限定请求参数 `params="username"` 表示请求必须包含username参数, `params="age=18"` 请求必须包含age=18参数,!username表示不能带username参数
- headers： `headers="haha"` 请求中必须包含名为haha的请求头。 `headers="hehe!=1"` 表示hehe请求头不为1（不带也可以）
- consumes： `consumes = "application/json"` 表示浏览器必须携带json格式的数据
- produces： `produces = "text/plain;charset=utf-8"` 表示返回数据的类型是纯文本，此时返回 `"<h1>你好</h1>"` 在浏览器中的到的就是 `"<h1>你好</h1>"` ，如果设置为 `produces = "text/html;charset=utf-8"` ,则返回h1大小的"你好"

```
@RestController
public class RequestMappingLimit {
    @RequestMapping(value="/test01",method= RequestMethod.POST)
    public String test01(){
        return "helloworld";
    }
    @RequestMapping(value = "/test02",params = "username")
    public String test02(){
```

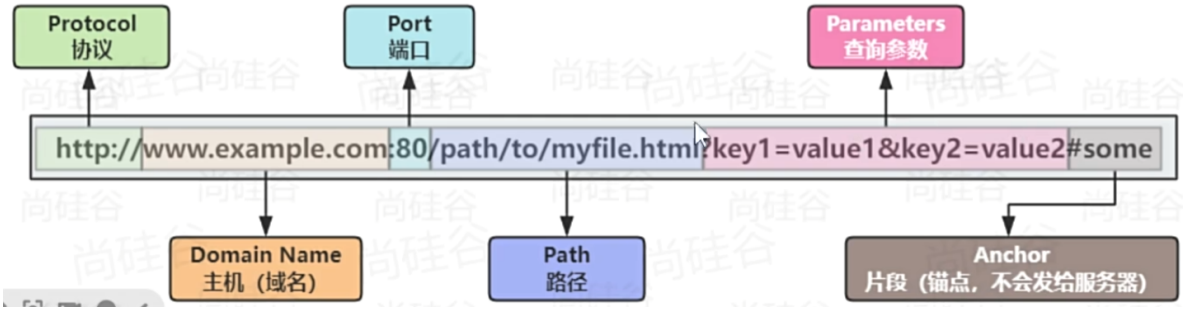
```
        return "test02";
    }
    @RequestMapping(value = "/test03",headers = "haha")
    public String test03(){
        return "test03";
    }
    @RequestMapping(value = "/test04",consumes = "application/json")
    public String test04(){
        return "test04";
    }
    @RequestMapping(value = "/test05",produces = "text/plain;charset=utf-8")
    public String test05(){
        return "<h1>你好</h1>";
    }
}
```

回顾HTTP

HTTP请求格式

- 请求首行：请求方式、请求路径、请求协议
- 请求头：k:v \n k:v
- 请求体：此次请求携带的其他数据（get放在请求首行的请求路径后，post放在负载中）

url格式：



JSON数据格式

- JavaScript Object Notation（JavaScript对象表示法）
- JSON用于将结构化数据(key-value)表示为JavaScript对象的标准格式，通常用于在网站上表示和传输数据
- JSON可以作为一个**对象**或**字符串**存在
  - 前者用于解读JSON中的数据，后者用于通过网络传输JSON数据
  - JavaScript提供一个全局的可访问的JSON对象来对这两种数据进行转换
- JSON是一种纯数据格式，它只包含属性，没有方法

请求处理

用普通变量，封装请求参数

```
请求参数:username=zhangsan&password=123456&cellphone=12345456&agreement=on
```

```
@RequestMapping("/handle01")
public String handle01(String username,
                       String password,
                       String cellphone,
                       boolean agreement){
    //注意：变量名要和参数名保持一致
    System.out.println(username);
    System.out.println(password);
    System.out.println(cellphone);
    System.out.println(agreement);
    return "ok";
}
```

输出：

```
zhangsan
123456
12345456
true
```

- 如果没有携带此参数，在获取时：包装类型自动封装为null，基本类型封装为默认值

*@RequestParam，封装多个参数

```
@RequestMapping("/handle02")
public String handle02(@RequestParam("username") String name,
                       @RequestParam("password") String pwd,
                       @RequestParam("cellphone") String phone,
                       @RequestParam("agreement") boolean ok){
    System.out.println(name);
    System.out.println(pwd);
    System.out.println(phone);
    System.out.println(ok);
    return "ok";
}
```

如上这种写法要求请求中一定要有这个参数，否则报错

我们可以用这种写法来取消必须携带

```
@RequestParam(value="agreement",required = false)
```

还可以：

```
@RequestParam(value="agreement",defaultValue="false")
```

表示不但可以不携带，并且如果没有携带那么获取的时这个默认值

*用POJO，封装多个参数

```
@RequestMapping("/handle03")
public String handle03(Person person){
    System.out.println(person);
    return "ok";
}
```

```
@Data
public class Person {
    private String username;
    private String password;
    private String cellphone;
    private boolean agreement;
}
```

（lombok的@Data注解不生效就将pom.xml中的lombok插件加上`<version>${lombok.version}</version>`）

输出：

```
Person{username = zhangsan, password = 123456, cellphone = 12345456, agreement = true}
```

@RequestHeader获取请求头

```
@RequestMapping("/handle04")
public String handle04(@RequestHeader("host") String host){
    System.out.println(host);
    return "ok";
}
```

与@RequestParam一样，同样可以加defaultValue参数、required参数

@CookieValue获取Cookie数据

同@RequestHeader

用POJO，级联封装复杂对象

```
username=zhangsan&password=123456&cellphone=12345456&agreement=on

address.province=陕西&address.city=西安市&address.area=雁塔区&sex=男&hobby=足球
&hobby=篮球&grade=二年级
```

发现address有三个属性,不能定义为String，而是一个对象

```
@Data
public class Person {
    private String username;
    private String password;
    private String cellphone;
    private boolean agreement;
    private Address address;
    private String sex;
    private String[] hobby;
    private String grade;
}
@Data
class Address{
    private String province;
    private String city;
    private String area;
}
```

*@RequestBody，封装JSON对象

```
{
    "username":"张三",
    "password":"123456",
    "cellphone":"1111111",
    "agreement":"on",
    "sex":"男",
    "hobby":["篮球","足球"],
    "grade":"二年级",
    "address":{
        "province":"陕西省",
        "city":"西安市",
        "area":"雁塔区"
 }
```

普通pojo拿不了json的数据,所以用@RequestBody使spring给我们反序列化对象：

```
@RequestMapping("/handle07")
public String handle07(@RequestBody Person person){
    System.out.println(person);
    return "ok";
}
```

输出：

```
Person{username = 张三, password = 123456, cellphone = 1111111, agreement = true, address = Address{province = 陕西省, city = 西安市, area = 雁塔区}, sex = 男, hobby = [篮球, 足球], grade = 二年级}
```

也可以：

```
@RequestMapping("/handle07")
public String handle07(@RequestBody String abc){
    System.out.println(abc);
    return "ok";
}
```

输出：

```
{
    "username":"张三",
    "password":"123456",
    "cellphone":"1111111",
    "agreement":"on",
    "sex":"男",
    "hobby":["篮球","足球"],
    "grade":"二年级",
    "address":{
        "province":"陕西省",
        "city":"西安市",
        "area":"雁塔区"
 }
```

所以，@RequestBody执行有两步

1.拿到请求体中的JSON字符串

2.把json字符串转为person对象

*@RequestPart/@RequestParam，封装文件，测试文件上传

MultipartFile ：专门封装文件项

```
@RequestMapping("/handle08")
public String handle08( Person person,
                        @RequestParam("headerImg") MultipartFile headerImgFile,
                        @RequestParam("lifeImg" ) MultipartFile[] lifeImgFiles)
throws IOException {
    //获取原始文件名
    String originalFilename = headerImgFile.getOriginalFilename();
    System.out.println(originalFilename);

    //文件大小
    long size = headerImgFile.getSize();
    //获取文件流,从而能获取文件的内容(但一般使用transferto)
```

```
        InputStream inputStream = headerImgFile.getInputStream();

        //文件保存
        headerImgFile.transferTo(new File("D:\\无用\\"+"1.png"));
        System.out.println("====以上处理了头像===========");

        if(lifeImgFiles.length>0){
            for (MultipartFile lifeImgFile : lifeImgFiles) {
                lifeImgFile.transferTo(new File("D:\\无用\\"+"2.png"));
            }
            System.out.println("========生活照保存结束=======");
        }
        System.out.println(person);
        return "ok";
    }
```

另外，springmvc限制了文件上传大小，单个1mb，总10mb，可以通过以下配置改变

```
spring.servlet.multipart.max-file-size=1GB
spring.servlet.multipart.max-request-size=10GB
```

使用HttpEntity，封装请求原始数据

HttpEntity：封装请求头、请求体； 把整个请求拿过来

泛型<>：请求体类型

```
    @RequestMapping("/handle09")
    public String handle09(HttpEntity<String> entity){
        //拿到所有请求头
        HttpHeaders headers = entity.getHeaders();
        System.out.println("请求头："+headers);
        //拿到请求体
        String body = entity.getBody();
        System.out.println("请求体："+body);

        return "ok";
    }

}
```

同样也可以直接封装成Person:

```
    @RequestMapping("/handle09")
    public String handle09(HttpEntity<Person> entity){
        //拿到所有请求头
        HttpHeaders headers = entity.getHeaders();
        System.out.println("请求头："+headers);
        //拿到请求体
        Person body = entity.getBody();
        System.out.println("请求体："+body);

        return "ok";
    }

}
```

使用原生Servlet API，获取原生请求对象

```
@RequestMapping("/handle10")
public void handle10(HttpServletRequest request,
                     HttpServletResponse response) throws IOException {
    String username = request.getParameter("username");
    System.out.println(username);

    response.getWriter().write("ok!"+username);
}
```

总结

重点：

- @RequestParam、pojo封装
- @RequestBody
- 文件上传

响应处理

*返回json

```
@RestController
public class ResponseTestController {

    @RequestMapping("/resp01")
    public Person resp01(){
        Person person = new Person();
        person.setUsername("张三");
        person.setPassword("123456");
        person.setCellphone("123456789");
        person.setAgreement(false);
        person.setSex("男");
        person.setHobby(new String[]{"篮球，足球"});
```

```
        person.setGrade("三年级");

        return person;

    }
}
```

在浏览器收到：

```
{
  "username": "张三",
  "password": "123456",
  "cellphone": "123456789",
  "agreement": false,
  "address": null,
  "sex": "男",
  "hobby": [
    "篮球，足球"
  ],
  "grade": "三年级"
}
```

发现，springmvc自动把返回的对象转为json

其实这是使用了@ResponseBody注解，将返回的内容写道响应体中（如果返回的是对象，那自然写入的就是json）。由于我们在整个类上加了@RestController注解，就可以不用使用@ResponseBody了

*文件下载

需要文件下载，自然要在响应头中告诉浏览器要进行文件下载，那么就要拿到整个响应数据，就要用到ResponseEntity（可以拿到响应头、响应体、状态码）

```
@RequestMapping("/download")
public ResponseEntity<InputStreamResource> download() throws IOException {

    FileInputStream fileInputStream = new
FileInputStream("C:\\Users\\14693\\Pictures\\Screenshots\\屏幕截图 2024-10-14
183901.png");
    //一次全部读取会溢出
    //byte[] bytes = fileInputStream.readAllBytes();

    //两个问题：
    //1、文件名中文会乱码:解决：
    String encode= URLEncoder.encode("哈哈.jpg","UTF-8");
    //2、文件太大会oom（内存溢出）：
    InputStreamResource resource = new InputStreamResource(fileInputStream);

    return ResponseEntity.ok()
            //内容类型：流
            .contentType(MediaType.APPLICATION_OCTET_STREAM)
            //内容大小
            .contentLength(fileInputStream.available())
            //ContentDisposition：内容处理方式
            .header("Content-Disposition", "attachment;filename="+encode)
```

```
                .body(resource);

    }
```

这以后就是文件下载的模板，只需要改文件路径和文件名，

总结

重点：

- 方法返回值类型String：返回json
- 返回值类型HttpEntity<>、ResponseEntity<>：请求头&请求体

RESTful

简介

基本理解

REST:Representational State Transfer表现层状态转移。是一种软件架构风格

完整理解：

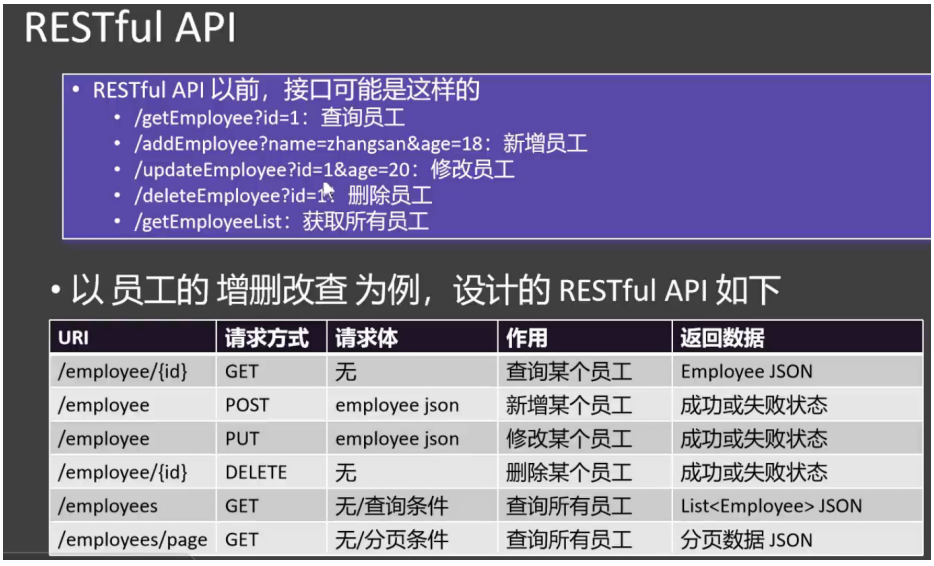
Resource Representational State Transfer

- Resource:资源
- Representational：表现形式，比如JSON、XML、JPEG等
- State Transfer：状态变化，通过HTTP的动词（GET、POST、PUT、DELETE）实现

一句话：使用资源名作为URI，使用HTTP的请求方式表示对资源的操作

满足REST风格的系统，我们称作RESTful系统

举例



URI	请求方式	请求体	作用	返回数据
/employee/{id}	GET	无	查询某个员工	Employee JSON
/employee	POST	employee json	新增某个员工	成功或失败状态
/employee	PUT	employee json	修改某个员工	成功或失败状态
/employee/{id}	DELETE	无	删除某个员工	成功或失败状态
/employees	GET	无/查询条件	查询所有员工	List<Employee> JSON
/employees/page	GET	无/分页条件	查询所有员工	分页数据 JSON

比如GET/user/1/order/7/produce/3 。表示获取1号用户的7号订单的3号产品信息

API理解

可以理解为接口（不是java中的interface），即Web应用暴露出来的让别人访问的请求路径

所以写web应用主要就是在写API接口，使前端调用对应的接口可以得到想要的数据

测试接口就是和前端交互，规定前端请求需要带的数据，以及后端响应带的数据，看能不能实现

调用别人的功能的方式

1.API：给第三方发请求，获取响应数据

2.SDK：导入jar包

CRUD案例

要求

controller只能调service，service调dao。只能上层调下层，不能反调（也可以同层）

bean

```
@Data
public class Employee {
    private Long id;
    private String name;
    private Integer age;
    private String email;
    private String gender;
    private String address;
    private BigDecimal salary;
}
```

DAO层

```
public interface EmployeeDao {
    //根据id查询员工信息
    Employee getEmpById(long id);
    //新增员工
    void addEmp(Employee employee);
    //修改
    void updateEmp(Employee employee);
    //删除
    void deleteEmpById(long id);
}
```

```
@Component
```

```
public class EmployeeDaoImpl implements EmployeeDao {
    @Autowired
    private JdbcTemplate jdbcTemplate;


    @Override
    public Employee getEmpById(long id) {
        String sql = "select * from employee where id = ?";
        Employee employee = jdbcTemplate.queryForObject(sql, new
BeanPropertyRowMapper<>(Employee.class), id);
        return employee;
    }

    @Override
    public void addEmp(Employee employee) {
        String sql = "insert into employee
(name,age,email,gender,address,salary) values (?,?,?,?,?,?)";
        int update=jdbcTemplate.update(sql,
                employee.getName(),
                employee.getAge(),
                employee.getEmail(),
                employee.getGender(),
                employee.getAddress(),
                employee.getSalary());
        System.out.println("新增成功，影响"+update+"行");
    }

    @Override
    public void updateEmp(Employee employee) {
        String sql = "update employee set
name=?,age=?,email=?,gender=?,address=?,salary=? where id = ?";
        int update=jdbcTemplate.update(sql,
                employee.getName(),
                employee.getAge(),
                employee.getEmail(),
                employee.getGender(),
                employee.getAddress(),
                employee.getSalary(),
                employee.getId());
        System.out.println("更新成功，影响"+update+"行");

    }

    @Override
    public void deleteEmpById(long id) {
        String sql = "delete from employee where id = ?";
        jdbcTemplate.update(sql, id);
    }
}
```

Service层

其实service是对dao层进行了一些包装(类似于静态代理)，比如防null处理。

```java
public interface EmployeeService {

    Employee getEmp(long id);

    void updateEmp(Employee emp);

    void saveEmp(Employee emp);
    void deleteEmp(long id);

}
```

```java
@Service
public class EmployeeServiceImpl implements EmployeeService {
    @Autowired
    EmployeeDao employeeDao;


    @Override
    public Employee getEmp(long id) {
        Employee empById = employeeDao.getEmpById(id);
        return empById;
    }

    @Override
    public void updateEmp(Employee emp) {
        //防null处理。考虑到service是被controller调用的
        //controller层传过来的employee的某些属性可能为null，所以先处理一下
        //怎么处理？
        Long id = emp.getId();
        if(id==null){
            return;
        }
        //1、到数据库查询到employee原来的值
        Employee empById = employeeDao.getEmpById(id);
        //2、把页面带来的值覆盖原来的值，页面没带的自然保持原值
        if(StringUtils.hasText(emp.getName())){
            empById.setName(emp.getName());
        }
        if(StringUtils.hasText(emp.getEmail())){
            empById.setEmail(emp.getEmail());
        }
        if(StringUtils.hasText(emp.getAddress())){
            empById.setAddress(emp.getAddress());
        }
        if(StringUtils.hasText(emp.getGender())){
            empById.setGender(emp.getGender());
        }
        if(emp.getAge()!=null){
            empById.setAge(emp.getAge());
        }
        if(emp.getSalary()!=null){
            empById.setSalary(emp.getSalary());
        }

        employeeDao.updateEmp(empById);
```

```
    }

    @Override
    public void saveEmp(Employee emp) {
        employeeDao.addEmp(emp);

    }

    @Override
    public void deleteEmp(long id) {
        employeeDao.deleteEmpById(id);
    }
}
```

Controller层

```
@RequestMapping("/employee/{id}")
public Employee get(@PathVariable("id") Long id){}
{id}相当于?
以上代码相当于访问localhost:8080/employee/id时将id拿出来赋值给方法的形参
```

@XxxMapping()：Rest映射注解,相当于@RequestMapping(value = "",method = RequestMethod.Xxx)

```
@GetMapping("/employee/{id}")
@RequestMapping(value = "/employee/{id}",method = RequestMethod.GET)
这两种写法相当
```

```
@RestController
public class EmployeeRestController {
    @Autowired
    EmployeeService employeeService;
    //按照id查询员工
    @GetMapping("/employee/{id}")
    //@RequestMapping(value = "/employee/{id}",method = RequestMethod.GET)
    public Employee get(@PathVariable("id") Long id){
        Employee emp=employeeService.getEmp(id);
        return emp;
    }

    //新增员工:
    // 要求前端发送请求把员工的json放在请求体中
    //要求json中必须携带id
    @PostMapping("/employee")
    public String add(@RequestBody Employee emp){

        employeeService.saveEmp(emp);
        return "ok";
    }
```

```
    //修改员工：
    // 要求前端发送请求把员工的json放在请求体中
    @PutMapping("/employee")
    public String update(@RequestBody Employee emp){
        employeeService.updateEmp(emp);
        return "ok";
    }

    @DeleteMapping("/employee/{id}")
    //@RequestMapping(value="/employee/{id}",method = RequestMethod.DELETE)
    public String Delete(@PathVariable("id") Long id){
        employeeService.deleteEmp(id);
        return "ok";
    }
}
```

统一返回R对象

我们发现，我们在上述代码中，返回的数据类型不一致，如何返回一个统一类型的对象？

如果我们规定对象的格式如下：

```
code:业务状态码，200是成功，剩下都是失败；前后端将来会一起商定不同的状态码前端要显示的效果
msg:服务端返给前端的提示信息
data:服务端返回给前端的数据。可以是多个对象，多个对象用数组
{
    "code":
    "msg":
    "data":

}
```

R类：

```
@Data
public class R {
    private Integer code;
    private String msg;
    private Object data;
    //方便返回对象
    public static R ok(Object data) {
        R r = new R();
        r.setCode(200);
        r.setData(data);
        r.setMsg("ok");
        return r;
    }
    //返回无data的对象
    public static R ok() {
        R r = new R();
        r.setCode(200);
        r.setMsg("ok");
        return r;
    }
    //失败的情况
```

```
    public static R error() {
        R r = new R();
        r.setCode(500);
        r.setMsg("error");
        return r;
    }
    //失败且返回code和msg
    public static R error(Integer code,String msg) {
        R r = new R();
        r.setCode(code);
        r.setMsg(msg);
        return r;
    }
    //失败且返回code和msg，并携带信息
    public static R error(String msg,Integer code,Object data) {
        R r = new R();
        r.setCode(code);
        r.setMsg(msg);
        r.setData(data);
        return r;
    }

}
```

修改后的Controller层:

```
@RestController
public class EmployeeRestController {
    @Autowired
    EmployeeService employeeService;
    //按照id查询员工
    @GetMapping("/employee/{id}")
    //@RequestMapping(value = "/employee/{id}",method = RequestMethod.GET)
    public R get(@PathVariable("id") Long id){
        Employee emp=employeeService.getEmp(id);
        return R.ok(emp);
    }

    //新增员工:
    // 要求前端发送请求把员工的json放在请求体中
    //要求json中必须携带id
    @PostMapping("/employee")
    public R add(@RequestBody Employee emp){
        employeeService.saveEmp(emp);
        return R.ok();
    }

    //修改员工:
    // 要求前端发送请求把员工的json放在请求体中
    @PutMapping("/employee")
    public R update(@RequestBody Employee emp){
        employeeService.updateEmp(emp);
        return R.ok();
    }

    @DeleteMapping("/employee/{id}")
```

```
    //@RequestMapping(value="/employee/{id}",method = RequestMethod.DELETE)
    public R Delete(@PathVariable("id") Long id){
        employeeService.deleteEmp(id);
        return R.ok();
    }
}
```

查询所有

dao:

```
@Override
public List<Employee> getList() {
    String sql = "select * from employee";
    List<Employee> list = jdbcTemplate.query(sql, new BeanPropertyRowMapper<>
(Employee.class));
    return list;
}
```

service:

```
@Override
public List<Employee> getList() {
    List<Employee> list = employeeDao.getList();
    return list;
}
```

controller:

```
@GetMapping("/employees")
public R all(){
    List<Employee> employees= employeeService.getList();
    return R.ok(employees);
}
```

跨域

补充:

如果想要访问路径在localhost:8080后面不直接写/employee...。可以在controller类上写:

```
@RequestMapping("/api/v1")
```

以后使用原本`localhost:8080/employees`可以访问的接口现在就要`localhost:8080/api/v1/employees`

我们在浏览器直接输入`localhost:8080/api/v1/employees`是可行的，但是如果我在`localhost:8080`用某种方法偷偷访问`localhost:8080/api/v1/employees`而不使用浏览器的地址框，会发现访问不了，且被提示违背了CORS policy，引发了跨域问题。这是啥?

CORS (Cross-Origin Resource Sharing) :跨源资源共享:

浏览器为了安全，默认会遵循同源策略（请求要去的服务器和当前项目所在服务器必须是同一个源【同一个服务器】），如果不是,请求就会被拦截。

复杂的跨域请求会发送两次

1.options请求：预检请求。浏览器先发送oprions请求，询问服务器是否允许当前域名进行跨域访问

2.真正的请求：POST、DELETE、PUT等

（协议、ip、端口有一个不一样，就不同源）

解决方案：

1.前端自己解决

2.后端解决： 允许前端跨域即可。

原理：服务器给浏览器的响应头中添加字段：Access-Control-Allow-Origin=(表示所有)

方法：用@CrossOrigin注释（类上或方法上）

拦截器

HandlerInterceptor

要让spring知道我们要拦截什么，必须对springMVC底层做一些配置，那么我们定义一个MySpringMVCConfig。并且要求容器中有WebMvcConfigurer组件

```
@Component
public class MyHandlerInterceptor implements HandlerInterceptor {
    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse
response, Object handler) throws Exception {
        System.out.println("MyHandlerInterceptor .preHandle");
        return true;//如果return false，则不执行目标方法
    }
    @Override
    public void afterCompletion(HttpServletRequest request, HttpServletResponse
response, Object handler, Exception ex) throws Exception {
        System.out.println("MyHandlerInterceptor afterCompletion");
    }

    @Override
    public void postHandle(HttpServletRequest request, HttpServletResponse
response, Object handler, @Nullable ModelAndView modelAndView) throws Exception
{
        System.out.println("MyHandlerInterceptor postHandle");
    }

}
```

```
@Configuration//专门对springmvc底层做一些配置
```

```
public class MySpringMVCConfig implements WebMvcConfigurer {
    @Autowired
    MyHandlerInterceptor myHandlerInterceptor;

    //添加拦截器
    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(myHandlerInterceptor)
                .addPathPatterns("/**");//拦截所有请求
    }


}
```

执行顺序: preHandle-->目标方法-->postHandle-->afterCompletion

如果有多个拦截器，可以加@order设置顺序

多拦截器执行顺序

拦截器执行顺序：顺序prehandle=>目标方法=>倒序posthandle=>渲染=>倒序aftercompletion。举例：

```
MyHandlerInterceptor0 .preHandle
MyHandlerInterceptor1 .preHandle
MyHandlerInterceptor2 .preHandle
MyHandlerInterceptor2 postHandle
MyHandlerInterceptor1 postHandle
MyHandlerInterceptor0 postHandle
MyHandlerInterceptor2 afterCompletion
MyHandlerInterceptor1 afterCompletion
MyHandlerInterceptor0 afterCompletion
```

只有执行成功的prehandle会倒序执行aftercompletion

posthandle、aftercompletion从哪里异常，此倒序链路就从哪里结束

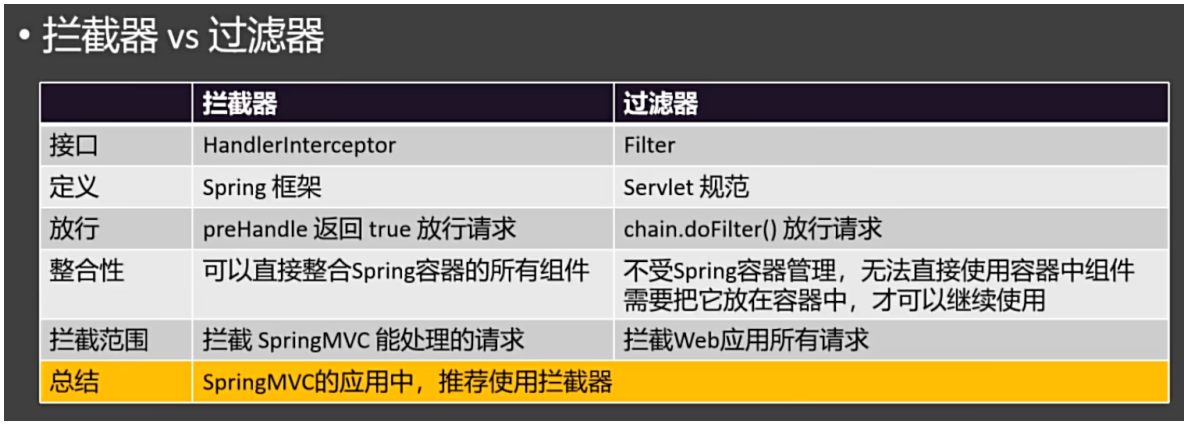
posthandle失败不会影响aftercompletion执行

比如，在上面三个拦截器中，将拦截器1的prehandle返回false，输出如下:

```
MyHandlerInterceptor0 .preHandle
MyHandlerInterceptor1 .preHandle
MyHandlerInterceptor0 afterCompletion
```

不执行目标方法就不会有posthandle

拦截器和过滤器的区别



异常处理

try-catch和throw之类的编程式异常处理在代码量增多时会很繁琐

所以我们考虑使用声明式异常处理

@ExceptionHandler指定异常处理方法

当我们在处理数学运算异常时，采用如下方法：

```
@RequestMapping("/hello")
public R hello(){
    try{
        int i=10/0;
        return R.ok();
    }
    catch (Exception e){
        return R.error(100,"执行异常");
    }
}
```

如果采用声明式异常，将会是如下代码:

```
//测试声名式异常处理
@RestController
public class HelloController {
    @RequestMapping("/hello")
    public R hello(@RequestParam(value = "i",defaultValue = "0") Integer i) {
        int j=10/i;
        return R.ok(j);
    }

    //如果此Controller类出现异常，会自动在本类中找有没有@ExceptionHandler标注的方法
    //如果有，执行这个方法，它的返回值就是客户端收到的结果
    //如果多个异常处理方法可以处理同一个异常，精确优先
    @ExceptionHandler(ArithmeticException.class)
    public R handleArithmeticException(ArithmeticException e) {
        return R.error(100,"执行异常"+e.getMessage());
    }

    //写一个大的，处理其他异常
```

```
    @ExceptionHandler(Throwable.class)
    public R handleException(Throwable e) {
        return R.error(500,"其他异常"+e.getMessage());
    }
```

@ControllerAdvice全局异常处理

@ExceptionHandler只能负责本类的异常，怎么处理全局异常呢?

在类上定义@ControllerAdvice,专门告诉springmvc，这个组件是专门负责全局异常处理的。

又因为我们要返回json格式，每个方法都要@ResponseBody,所以我们把他干脆写在类上。然后@ResponseBody和@ControllerAdvice可以合成一个注解：@RestControllerAdvice

```
//全局异常处理器
/*@ResponseBody
@ControllerAdvice*///专门告诉springmvc，这个组件是专门负责全局异常处理的
@RestControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(Throwable.class)
    public R error(Throwable e) {
        return R.error(500, e.getMessage());
    }

    @ExceptionHandler(ArithmeticException.class)
    public R error(ArithmeticException e){
        return R.error(500, e.getMessage());
    }
}
```

异常处理执行优先级：本类>全局, 然后精确优先

如果出现了异常，本类和全局都不能处理，springboot底层对springmvc有兜底处理机制。但我们不应该让他兜底，而应该编写全局异常处理器，处理所有异常

异常处理的最终方式

异常处理的推荐方式:

后端只编写正确的业务逻辑，如果出现业务问题，后端通过抛异常的方式提前中断业务逻辑(抛异常也可以让上层及以上的链路知道中断原因)，前端感知异常。

实现:

1.必须有业务异常类；举例：BizException

2.必须有异常枚举类；举例：BizExceptionEnume　专门列举项目中每个模块将会出现的所有异常情况

3.编写业务代码时，只需编写正确逻辑，如果出现预期问题，需要以抛异常的方式中断逻辑并通知上层

4.全局异常处理器；举例：GlobalExceptionHandler　处理所有异常，返回给前端约定的json数据和错误码

业务异常类:

```
//业务异常类
import lombok.Getter;
public class BizException extends RuntimeException {
    @Getter
    private Integer code;
    @Getter
    private String msg;
    public BizException(BizExceptionEnume exceptionEnume) {
        super(exceptionEnume.getMsg());
        this.code = exceptionEnume.getCode();
        this.msg = exceptionEnume.getMsg();
    }
}
```

异常枚举类：

```
public enum BizExceptionEnume {

    //枚举的异常将会在写业务时动态扩充
    //订单模块相关异常
    ORDER_CLOSED(10001,"订单已关闭"),
    ORDER_NOT_EXIST(10002,"订单不存在"),
    ORDER_TIMEOUT(10003,"订单超时"),
    //商品模块异常
    PRODUCT_STOCK_NOT_ENOUGH(20003,"库存不足")
    //.....
    ;
    @Getter
    private Integer code;
    @Getter
    private String msg;
    private BizExceptionEnume(Integer code, String msg) {
        this.code = code;
        this.msg = msg;
    }
}
```

全局异常处理器：

```
@RestControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(Throwable.class)
    public R error(Throwable e) {
        return R.error(500, e.getMessage());
    }
    @ExceptionHandler(BizException.class)
    public R handleBizException(BizException e){
        Integer code=e.getCode();
        String msg=e.getMsg();
        return R.error(code,msg);
    }
}
```

使用示例：

```
if(id==null){
   throw new BizException(BizExceptionEnume.ORDER_CLOSED);
}
```

数据校验

JSR303

步骤:

- 导入校验包
- 编写校验注解
- 使用@Valid告诉springmvc 进行校验
- 在@Valid修饰的参数后面加一个BindingResult bindingResult参数 （这一步以后会被全局异常处理替代）

导入依赖:

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

编写注解：

```
@Data
public class Employee {
    private Long id;
    @NotBlank(message = "姓名不能为空")
    private String name;
    @NotNull(message = "年龄不能为空")
    @Max(value=150,message = "年龄不能超过150岁")
    @Min(value=0,message = "年龄不能小于0")
    private Integer age;
    @Email(message="邮箱格式不正确")
    private String email;
    private String gender;
    private String address;
    private BigDecimal salary;
}
//（）中的message是返回给前端的默认消息，在第四步中用到
```

第三步：在Controller层中

```
@PostMapping("/employee")
public R add(@RequestBody @Valid Employee emp){
    employeeService.saveEmp(emp);
    return R.ok();
}
```

完成，再来加一点要求：

如果校验出错，返回前端如下格式的数据：

```
{
    "code":500,
    "msg":"校验失败",
    "data":{
        "name":"姓名不能为空",
        "age":“年龄不能超过150"
    }
}
```

所以，第四步：

```
@PostMapping("/employee")
public R add(@RequestBody @Valid Employee emp, BindingResult bindingResult){

    if(!bindingResult.hasErrors()){
        employeeService.saveEmp(emp);
        return R.ok();
    }
    //拿到所有属性错误的信息
    Map<String,String> map=new HashMap<>();
    for (FieldError fieldError : bindingResult.getFieldErrors()) {
        //获取属性名
        String field = fieldError.getField();
        //获取错误消息
        String message = fieldError.getDefaultMessage();
        map.put(field,message);
    }
    return R.error("校验失败",500,map);
}
```

全局异常处理

发现如果每个方法采用如上方式，非常繁琐，所以考虑采用全局异常处理

因为校验失败会报MethodArgumentNotValidException异常，所以在全局异常类中加上：

```
@ExceptionHandler(MethodArgumentNotValidException.class)
public R handleMethodArgumentNotValidException(MethodArgumentNotValidException
e){
    BindingResult bindingResult = e.getBindingResult();
    //拿到所有属性错误的信息
    Map<String,String> map=new HashMap<>();
    for (FieldError fieldError : bindingResult.getFieldErrors()) {
        //获取属性名
        String field = fieldError.getField();
        //获取错误消息
        String message = fieldError.getDefaultMessage();
        map.put(field,message);
    }
    return R.error("校验失败",500,map);
}
```

自定义校验器

流程：

写一个校验注解并绑定校验器

校验注解：

```
@Documented
@Constraint(validatedBy = {GenderValidator.class})//校验器完成真正的校验功能
@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Gender {
    String message() default "{jakarta.validation.constraints.NotNull.message}";
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};//三个属性都没啥用
}
```

校验器:

```
//第一个泛型表示注解，第二个泛型表示要校验的属性的类型
public class GenderValidator implements ConstraintValidator<Gender,String> {

    //vlaue:前端提交来的让我们进行校验的属性值
    //context:校验上下文
    @Override
    public boolean isValid(String value, ConstraintValidatorContext
constraintValidatorContext) {
        return "男".equals(value)||"女".equals(value);//表示校验是否通过
    }
}
```

错误消息提示国际化

为了让客户在不同的地区访问得到的错误信息语言其所在地区一致，我们通常不将注解中的message信息写死，而是写成占位符。

@Gender(message="{gender.message}")。

然后在配置文件夹resources下创建messages_zh_CN.properties。在其中写上`gender.message=性别必须是男或女`。（如果是美国就是messages_en_US.properties）

此时会发现报出的错误信息是乱码，那么将设置中File Encodings中的设置修改成UTF-8。

接下来，就会根据请求中的请求头`accept-language`设置的语言返回错误信息。

各种O的分层模型

如果前端的对象和数据库的对象使用一个javabean，会发生一些问题，比如前端的注解和数据库的注解都作用在一个对象上了，这不符合设计模式中的单一职责。再比如如果返回密码等敏感信息给前端，不安全。

所以出现了各种O的模型：

Pojo：普通java类

Dao：数据库访问对象

TO：transfer object 用于传输数据的对象

VO：View/Value Object 视图对象/值对象 ： 专门用来封装前端数据的对象。而且在给前端返回数据时要进行脱敏（消除敏感信息如密码等），所以javabean也要分层

项目中的VO用法

add方法

在vo/req包下创建EmployeeAddVo,表示在添加员工时对接前端的object。它不需要id这个字段

内容如下:

```
@Data
public class EmployeeAddVo {
    @NotBlank(message = "姓名不能为空")
    private String name;
    @NotNull(message = "年龄不能为空")
    @Max(value=150,message = "年龄不能超过150岁")
    @Min(value=0,message = "年龄不能小于0")
    private Integer age;
    @Email(message="邮箱格式不正确")
    private String email;
    @Gender(message = "性别只能为男/女")
    private String gender;
    private String address;
    private BigDecimal salary;

}
```

这样的话Employee类中的所有数据校验注解都不需要了。

并且我们把EmployeeRestController层中的add方法的形参改成EmployeeAddVo类型。但又发现调用的Service层的方法需要Employee类的对象，于是我们需要把传入的EmployeeAddVo类型的参数转化成Employee类型的对象。

spring给我们提供了`BeanUtils.copyProperties(要拷贝的对象，目标对象)`方法。

修改后的add方法如下：

```
@PostMapping("/employee")
public R add(@RequestBody @Valid EmployeeAddVo vo){
    //把vo转为do
    Employee emp=new Employee();
    //属性拷贝
    BeanUtils.copyProperties(vo,emp);
    employeeService.saveEmp(emp);
    return R.ok();
}
```

update方法

创建EmployeeUpdateVo类，它需要的字段与Employee类相同，但是id不能为空。

```
@Data
public class EmployeeUpdateVo {
    @NotNull(message = "id不能为空")
    private Long id;
    private String name;
    private Integer age;
    private String email;
    private String gender;
    private String address;
    private BigDecimal salary;
}
```

修改后的方法：

```
@PutMapping("/employee")
public R update(@RequestBody @Valid EmployeeUpdateVo vo){

    Employee emp=new Employee();
    BeanUtils.copyProperties(vo,emp);
    employeeService.updateEmp(emp);
    return R.ok();
}
```

*接口文档

swagger可以快速生成实时接口文档，方便前后开发人员进行协调沟通。遵循OpenAPI规范。

knife4j是基于swagger之上的增强套件

在pom.xml中导入依赖

```
<dependency>
    <groupId>com.github.xiaoymin</groupId>
    <artifactId>knife4j-openapi3-jakarta-spring-boot-starter</artifactId>
    <version>4.5.0</version>
</dependency>
```

启动项目后访问

localhost:8080/doc.html 即可生成文档

注解：

- @Tag(name="")：描述controller类的作用
- @Operation(summary="")：描述方法的作用
- @Schema(description="")：描述vo等类的作用，并且类中属性也用此注解表明是什么
- @Parameters({@Parameter(name="形参名",description="") ,@Parameter(name="形参名",description="")..... }) ：写在controller层的方法上，用于描述形参的含义。 其中@Parameter中的参数还有 in（表示在请求中参数所在的位置）、required（是否一定需要此参数）

日期相关

前端在提交日期相关的数据时，如果后端将数据库的日期记录的类型设置为datetime，那么前端在提交请求时的日期格式要求很严格，格式不对就会显示无法反序列化。如何解决：

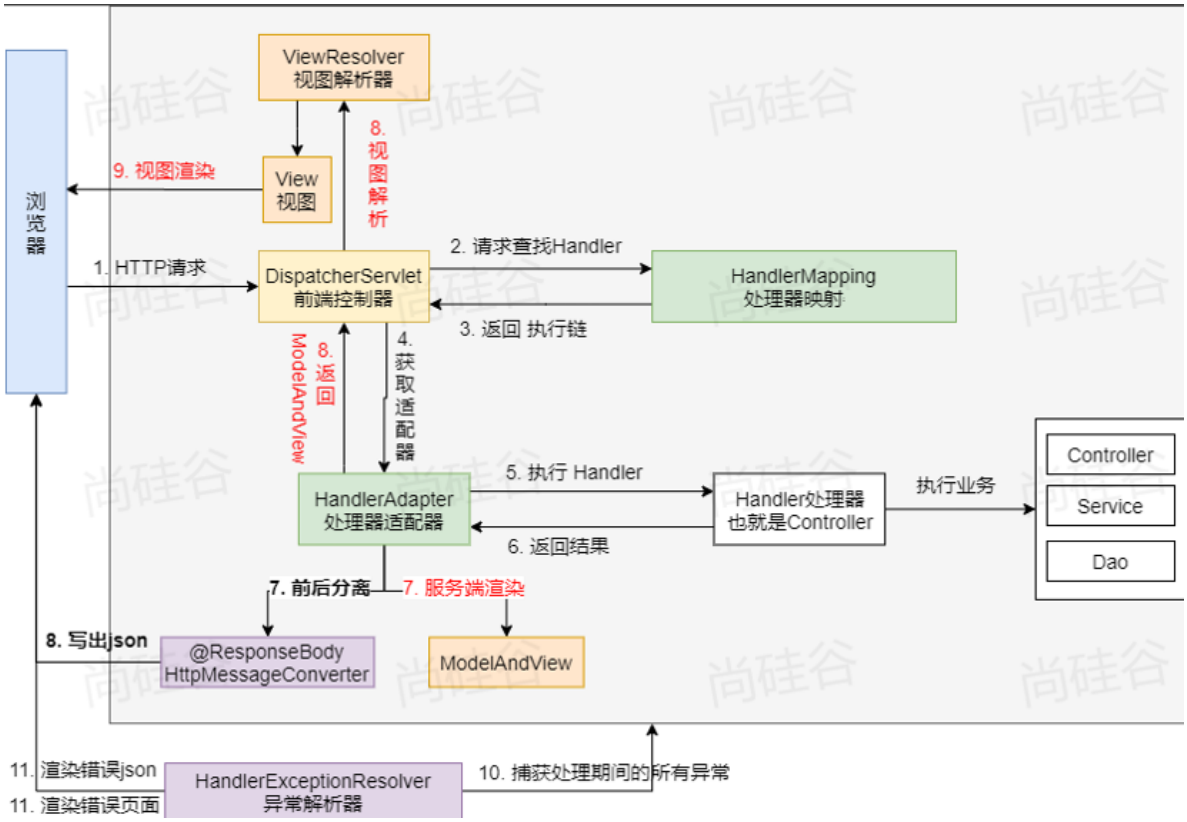
在addvo和respvo的birth属性上标注@JsonFormat(pattern="格式")。然后将获取数据库对象的方法改为如下格式:

```
@GetMapping("/employee/{id}")
//@RequestMapping(value = "/employee/{id}",method = RequestMethod.GET)
public R get(@PathVariable("id") Long id){
    Employee emp=employeeService.getEmp(id);
    EmployeeRespVo respVo=new EmployeeRespVo();
    BeanUtils.copyProperties(emp,respVo);
    return R.ok(respVo);
}
```

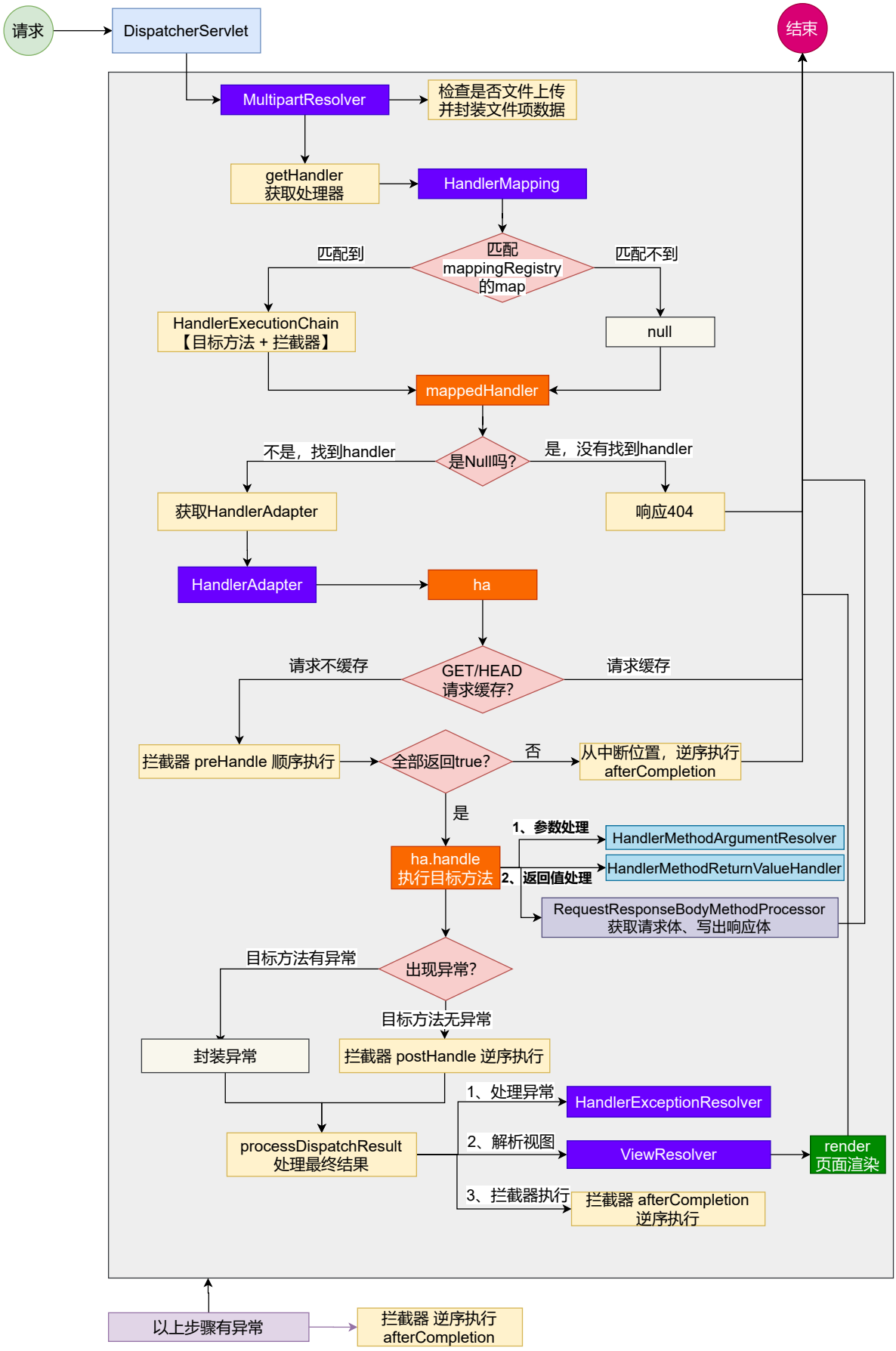
即将返回的对象类型改为EmployeeRespVo类型，这样就可以做到以指定格式发送请求&获取数据库对象

Dispatcherservlet运行流程

简要版：


ViewResolver
视图解析器
8. 视图解析
9. 视图渲染
View
视图
浏览器
1. HTTP请求
DispatcherServlet
前端控制器
2. 请求查找Handler
HandlerMapping
处理器映射
3. 返回 执行链
8. 返回ModelAndView
4. 获取适配器
HandlerAdapter
处理器适配器
5. 执行 Handler
Handler处理器
也就是Controller
6. 返回结果
执行业务
Controller
Service
Dao
7. 前后分离
7. 服务端渲染
8. 写出json
@ResponseBody
HttpMessageConverter
ModelAndView
11. 渲染错误json
11. 渲染错误页面
HandlerExceptionResolver
异常解析器
10. 捕获处理期间的所有异常

详细版：


请求
DispatcherServlet
结束
MultipartResolver
检查是否文件上传
并封装文件项数据
getHandler
获取处理器
HandlerMapping
匹配
mappingRegistry
的map
匹配到
匹配不到
HandlerExecutionChain
【目标方法 + 拦截器】
null
mappedHandler
是Null吗?
不是，找到handler
是，没有找到handler
获取HandlerAdapter
响应404
HandlerAdapter
ha
GET/HEAD
请求缓存?
请求不缓存
请求缓存
拦截器 preHandle 顺序执行
全部返回true?
否
从中断位置，逆序执行
afterCompletion
是
1、参数处理
HandlerMethodArgumentResolver
ha.handle
执行目标方法
2、返回值处理
HandlerMethodReturnValueHandler
RequestResponseBodyMethodProcessor
获取请求体、写出响应体
出现异常?
目标方法有异常
目标方法无异常
封装异常
拦截器 postHandle 逆序执行
1、处理异常
HandlerExceptionResolver
processDispatchResult
处理最终结果
2、解析视图
ViewResolver
render
页面渲染
3、拦截器执行
拦截器 afterCompletion
逆序执行
以上步骤有异常
拦截器 逆序执行
afterCompletion