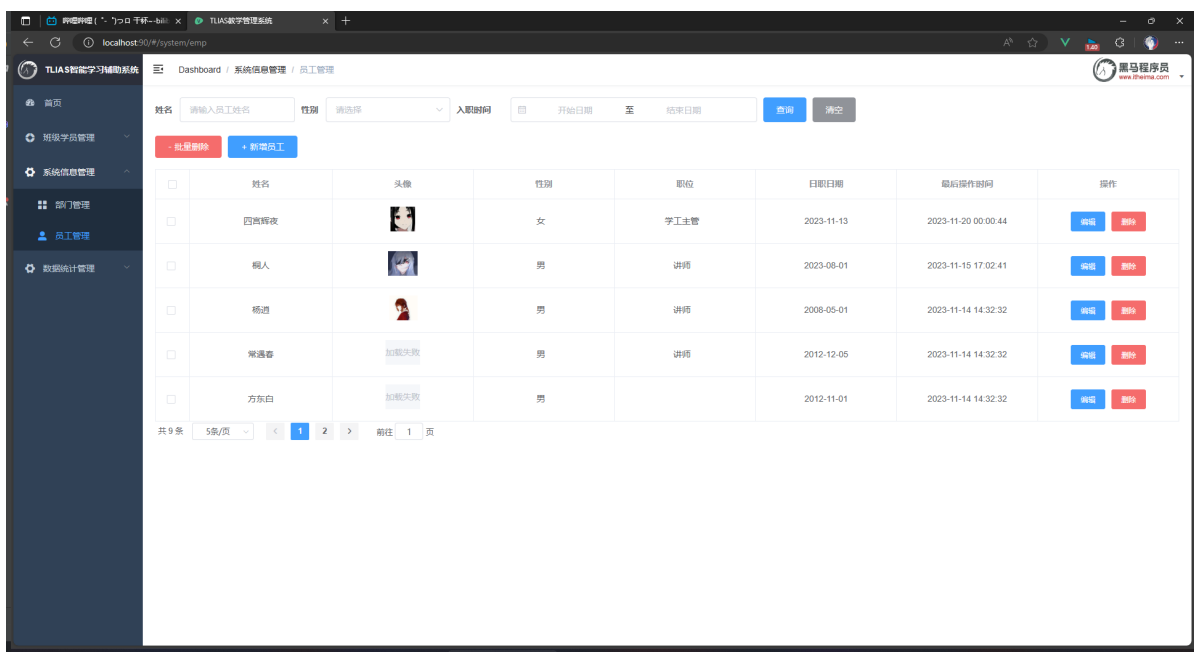
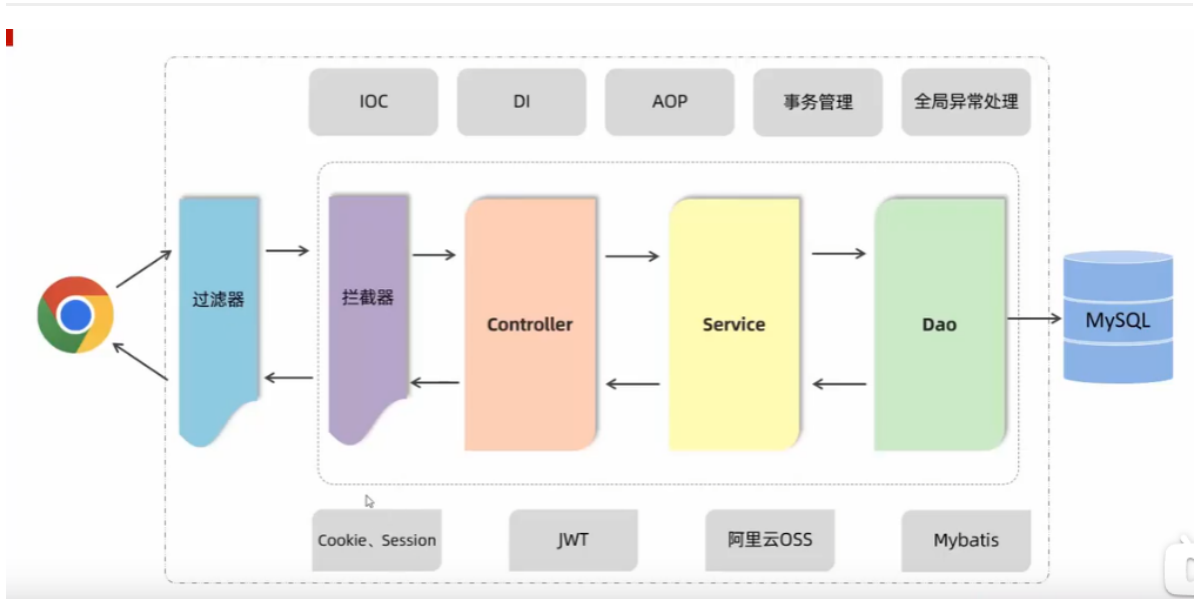


# JavaWeb后台管理项目



## 项目概况



### 1. IOC (Inversion of Control):

- 意义：IOC 是一种设计原则，它反转了传统的控制流。在传统编程中，程序员控制对象的创建和管理，而在IOC中，控制权被反转给了框架。
- 作用：Spring框架利用IOC容器，管理应用中的组件（Bean），使得组件之间的依赖关系由Spring容器动态管理。
- 业务场景：通过IOC容器，可以更方便地组织和管理各个模块，降低组件之间的耦合度。

### 2. DI (Dependency Injection):

- 意义：DI 是IOC的一种实现方式，它通过将对象的依赖关系注入到对象中，而不是由对象自己创建依赖关系。
- 作用：简化组件之间的依赖关系，降低耦合度，使得系统更易于维护和扩展。
- 业务场景：在Spring中，通过注解或XML配置，容器会负责将依赖关系注入到需要的地方。

### 3. AOP (Aspect-Oriented Programming):

- 意义：AOP是一种编程范式，通过在程序中横切关注点（cross-cutting concerns）来实现一些通用的功能，如日志、事务管理等。
- 作用：将横切关注点与主要业务逻辑分离，提高代码的模块化和可维护性。
- 业务场景：在Spring中，可以通过AOP实现**事务管理**、**日志记录**等横切关注点的功能。

### 4. 事务管理:

- 作用：确保一组操作要么全部成功执行，要么全部失败回滚，以维护数据的一致性。
- 业务场景：在涉及到多个数据库操作或需要保证数据一致性的业务中，事务管理非常重要。

### 5. 全局异常处理:

- 作用：统一处理应用程序中发生的异常，提高系统的健壮性和可维护性。
- 业务场景：通过全局异常处理，可以将异常信息记录下来，返回友好的错误信息给前端，同时避免系统因未处理异常而崩溃。

### 6. 过滤器:

- 作用：拦截请求或响应，对它们进行处理或修改。
- 业务场景：在请求到达Controller之前或响应返回给客户端之前，可以通过过滤器进行一些预处理或后处理的操作。

### 7. 拦截器:

- 作用：与过滤器类似，拦截器也可以对请求进行处理，但它更专注于对Controller的方法进行预处理和后处理。
- 业务场景：在业务逻辑处理前后执行一些通用的操作，如权限验证、日志记录等。

### 8. Controller:

- 作用：处理用户请求，调用相应的业务逻辑，并返回视图或数据给客户端。
- 业务场景：Controller是应用程序的入口点，负责接收和处理用户的请求。

### 9. Service:

- 作用：包含业务逻辑，处理业务规则，是Controller和Dao之间的中间层。
- 业务场景：业务逻辑通常封装在Service层，以保持Controller的简洁性，并使业务规则更易于维护和测试。

### 10. Dao:

- 作用：数据访问对象，负责与数据库进行交互，执行CRUD操作。
- 业务场景：Dao层将数据库操作封装起来，使得业务逻辑层不必关心数据访问的具体实现细节。

### 11. MySQL:

- 作用：关系型数据库管理系统，用于存储和检索数据。
- 业务场景：作为数据存储的一种选择，用于持久化应用程序的数据。

### 12. Cookie、Session:

- 作用：用于在Web应用中维护用户状态和跟踪用户会话。
- 业务场景：在用户登录后，通过Cookie和Session来标识和跟踪用户，实现状态的保持。

### 13. JWT (JSON Web Token):

- 作用：一种用于在网络上安全传输信息的开放标准，通常用于身份验证和信息传递。
- 业务场景：在分布式系统中，JWT可用于生成令牌，用于用户认证和授权。

### 14. 阿里云 OSS (Object Storage Service) :

- 阿里云OSS是阿里云提供的分布式对象存储服务，用于存储和检索大量的非结构化数据，例如图片、音频、视频等。在Spring Boot项目中，你可以使用阿里云OSS来存储和管理你的应用程序的静态资源。

### 15. MyBatis:

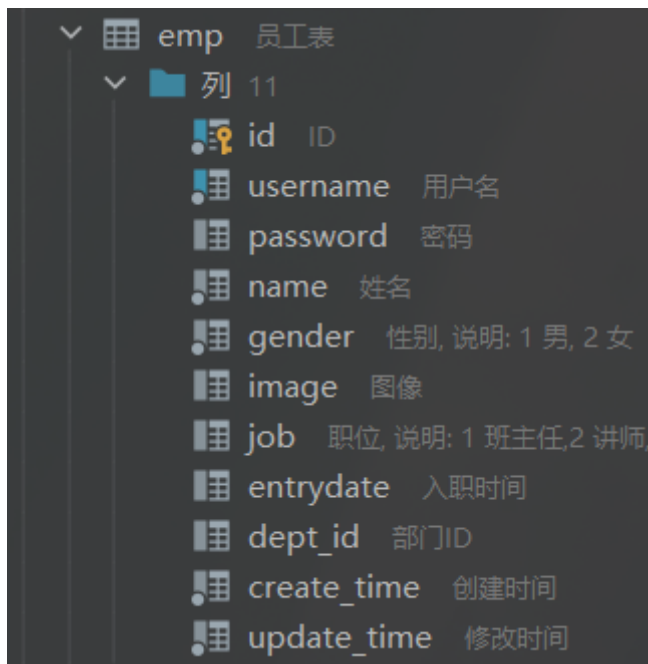
- MyBatis是一个基于Java的持久层框架，用于简化数据库访问。它通过XML或注解配置，将Java对象映射到数据库表，并提供了灵活的SQL查询语言。MyBatis使得数据库操作更直观，同时提供了对原生SQL的支持。在Spring Boot中，你可以集成MyBatis来处理数据库交互。

## 项目步骤

# 1.设计

现有项目经理给的页面原型,对各个部分都设计好了才开始开发

这里说一下数据库的设计:



是否唯一\非空\有默认值 要根据产品经理说的, 另外无论任何表都需要带创建时间和修改时间

## 2.后端准备

- 1.准备数据库表(dept、emp)
- 2.创建springboot工程, 引入对应的起步依赖(web、mybatis、mysql驱动、lombok)
- 3.配置文件application.yml中引入mybatis的配置信息, 准备对应的实体类

```
spring:
  #数据库的配置
  datasource:
    driver-class-name: com.mysql.cj.jdbc.Driver
    url: jdbc:mysql://localhost:3306/tlias
    username: root
    password: 123456

  #上传文件的限制
  servlet:
    multipart:
      max-file-size: 10MB
      max-request-size: 100MB

mybatis:
  configuration:
    #执行 SQL 语句操作时在控制台输出日志信息
    log-impl: org.apache.ibatis.logging.stdout.StdoutImpl
    #数据库的命名到类的时候转换 a_column -----> aColumn
    map-underscore-to-camel-case: true

#阿里云OSS配置后期专门创建一个类,用@ConfigurationProperties(prefix = "aliyun.oss")拿到
aliyun:
```

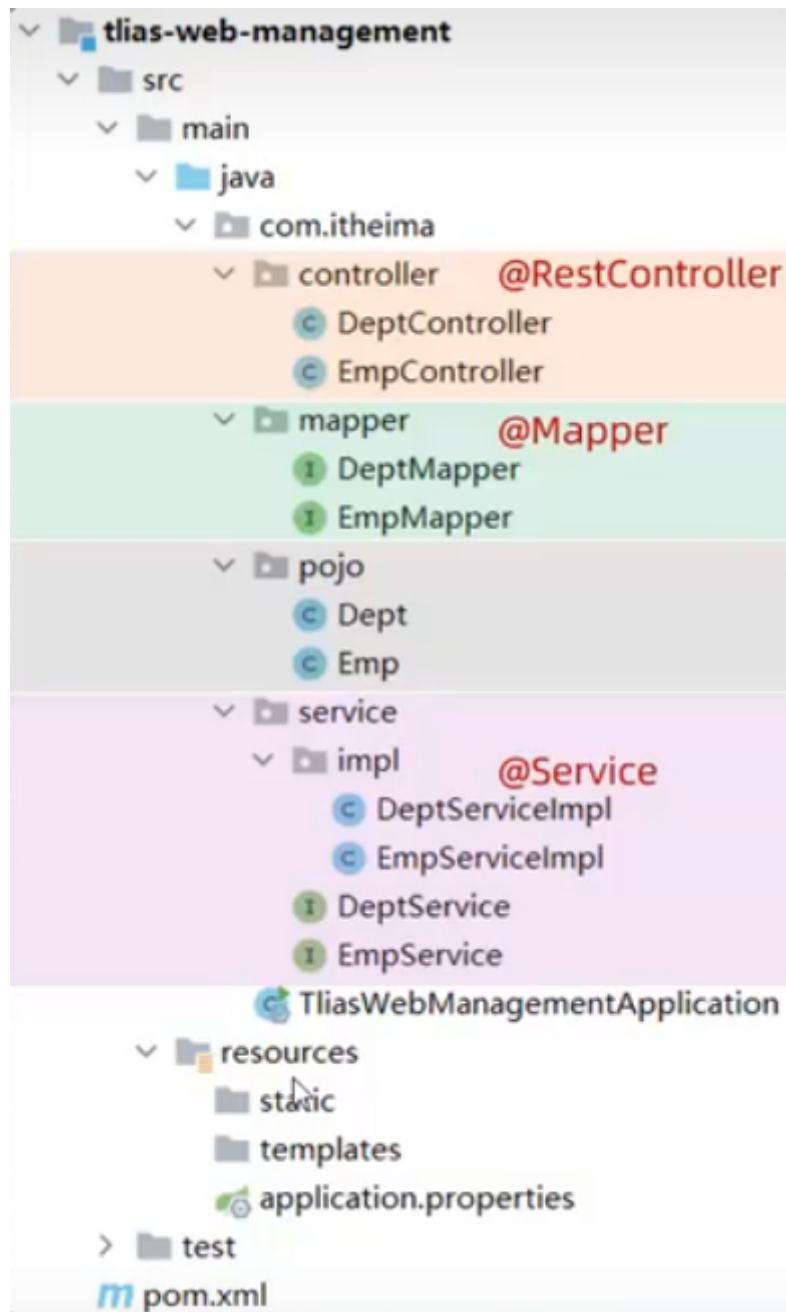
```

oss:
    endpoint: https://oss-cn-hangzhou.aliyuncs.com
    accessKeyId: ????? #填自己的
    accessKeySecret: ????? #填自己的
    bucketName: ????? #填自己的

#spring事务管理日志:输出详细的调试信息, 包括 SQL 语句的执行情况、事务管理的细节等
logging:
    level:
        org.springframework.jdbc.support.JdbcTransactionManager: debug

```

#### 4.准备对应的Mapper、Service(接口、实现类)、Controller基础结构



#### 5.在pojo包中创建Result类,用于对前端的响应

```

@Data
@NoArgsConstructor
@AllArgsConstructor
public class Result {
    private Integer code; //响应码, 1 代表成功; 0 代表失败
    private String msg; //响应信息 描述字符串
}

```

```

private Object data; //返回的数据

//增删改 成功响应
public static Result success(){
    return new Result(1,"success",null);
}
//查询 成功响应
public static Result success(Object data){
    return new Result(1,"success",data);
}
//失败响应
public static Result error(String msg){
    return new Result(0,msg,null);
}
}

```

## 3.业务实现 - 员工管理的CRUD

### 3.1.查询

Controller层代码:

```

@GetMapping
public Result pagingQuery(@RequestParam(defaultValue = "1") Integer page,
                          @RequestParam(defaultValue = "10") Integer pageSize,
                          String name,
                          Short gender,
                          @DateTimeFormat(pattern = "yyyy-MM-dd") LocalDate begin,
                          @DateTimeFormat(pattern = "yyyy-MM-dd") LocalDate end) {
    //这里是用了Param传递参数 -- 具体怎么传的看产品经理给的文档
    log.info("参数: {}, {}, {}, {}, {}, {}", page, pageSize, name, gender, begin, end);
    PageBean pageBean = empService.pagingQuery(page, pageSize, name, gender, begin,
end);
    return Result.success(pageBean);
}

```

#### 3.1.1.分页条件查询 - PageHelper

自己实现的话代码固定并且繁琐,因此使用**mybatis的分页查询插件 : PageHelper**

要用插件那么就需要pom.xml配置:

```

<!--      分页查询插件-->
<dependency>
    <groupId>com.github.pagehelper</groupId>
    <artifactId>pagehelper-spring-boot-starter</artifactId>
    <version>1.4.6</version>
</dependency>

```

service层代码:

```

@Override
public PageBean pagingQuery(Integer page, Integer pageSize, String name, Short gender,
                             LocalDate begin,
                             LocalDate end) {
    PageHelper.startPage(page, pageSize);
    List<Emp> list = empMapper.getAllData(name, gender, begin, end);
    Page<Emp> p = (Page<Emp>) list; //这条代码之后相当于顺便执行了
    //select count(*) from emp; 和 select from emp limit ?,?;
    PageBean pageBean = new PageBean(p.getTotal(), p.getResult());
    return pageBean;
}

```

说明:

1. page: 页码    pageSize: 传过来每页展示的记录数 交给PageHelp相当于这个工具的配置
2. Page类型是插件带的
3. 最后用pageBean这个自己创建的类装好**总数与需要展示的几条数据**,返回给前端

```

public class PageBean {
    private Long total; //返回的数据的总数
    private List rows;  //返回的数据
}

```

mapper层代码:

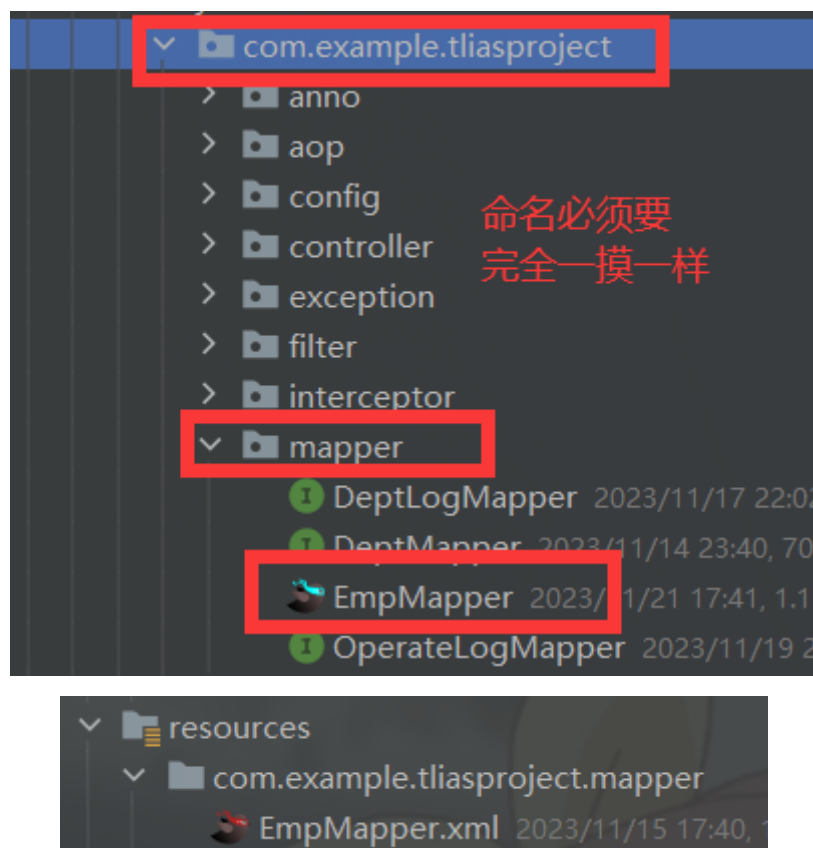
```

//这里不用注解写了,因为业务逻辑比较复杂,后面xml文件会找到并且关联
public List<Emp> getAllData(String name, Short gender, LocalDate begin, LocalDate end);

```

### 3.1.2.涉及mybatis里面与数据库进行复杂交互xml文件编写

在resources目录下创建与相关的mapper同包同名的文件,以刚刚提到的EmpMapper为例子:



```

<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
    PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.example.tliasproject.mapper.EmpMapper">
    <select id="getAllData" resultType="com.example.tliasproject.pojo.Emp">
        select *
        from emp
        <where>
            <if test="name != null and name != ''">
                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>
        </where>
        order by update_time desc
    </select>
</mapper>

```

代码先这样放着吧,后面说到增删改的再放相关的 **主要注意一下用到的标签**

### 3.2.删除

Controller层代码:

```

@DeleteMapping("/{ids}")
public Result deleteData(@PathVariable ArrayList<Integer> ids) {
    //这里是路径传递参数
    log.info("删除数据id:{", ids);
    empService.deleteData(ids);
    return Result.success();
}

```

Service层代码:

```

@Override
public void deleteData(ArrayList<Integer> ids) {
    empMapper.deleteData(ids);
}

```

Mapper层代码:

```

void deleteData(ArrayList<Integer> ids);

```

```

<delete id="deleteData">
    delete
    from emp
    where id in
    <foreach collection="ids" item="id" open="(" close=")" separator=",">
        #{id}
    </foreach>
</delete>

```

注意这里的循环标签

### 3.3. 新增

Controller层代码:

```
@PostMapping
public Result addData(@RequestBody Emp emp) {
    //这里使用Json格式的传输的数据
    log.info("新增员工:{}", emp);
    empService.addData(emp);
    return Result.success();
}
```

Service层代码:

```
@Override
public void addData(Emp emp) {
    //主要是更新一下操作时间
    emp.setCreateTime(LocalDateTime.now());
    emp.setUpdateTime(LocalDateTime.now());
    empMapper.addData(emp);
}
```

Mapper层代码:

```
void addData(Emp emp);
```

```
<insert id="addData" parameterType="com.example.tliasproject.pojo.Emp">
    insert into emp (username, name, gender, image, job, entrydate, dept_id,
create_time, update_time)
    values (#{username}, #{name}, #{gender}, #{image}, #{job}, #{entrydate}, #{deptId},
#{createTime},
        #{updateTime})
</insert>
```

### \*文件上传 - 阿里云

#### 云服务使用步骤

1. 准备工作 - 对象存储OSS, 创建bucket 拿到自己的密钥
2. 参照官方SDK编写入门程序  
SDK:Software Development Kit的缩写, 软件开发工具包, 包括辅助软件开发的依赖(jar包)、代码示例等, 都可以叫做SDK。
3. 集成使用

#### 最终的应用代码

pom.xml配置 -> 看官方文档里面会有写的

```
<!-- 阿里云OSS -->
<dependency>
    <groupId>com.aliyun.oss</groupId>
    <artifactId>aliyun-sdk-oss</artifactId>
    <version>3.15.1</version>
```



```

</dependency>

<dependency>
    <groupId>javax.xml.bind</groupId>
    <artifactId>jaxb-api</artifactId>
    <version>2.3.1</version>
</dependency>
<dependency>
    <groupId>javax.activation</groupId>
    <artifactId>activation</artifactId>
    <version>1.1.1</version>
</dependency>
<!-- no more than 2.3.3-->
<dependency>
    <groupId>org.glassfish.jaxb</groupId>
    <artifactId>jaxb-runtime</artifactId>
    <version>2.3.3</version>
</dependency>

```

Controller层代码:

```

@RestController
public class UploadController {
    @Autowired
    AliOSSUtils alioSSUtils;

    @PostMapping("/upload") //无所谓,前端会出手
    public Result upload(MultipartFile image) throws IOException {
        log.info("上传图片文件名:{}", image.getOriginalFilename());
        String url = alioSSUtils.upload(image);
        log.info("上传完毕,url为:{}", url);
        return Result.success(url);
    }
}

```

说明:MultipartFile image 这个参数

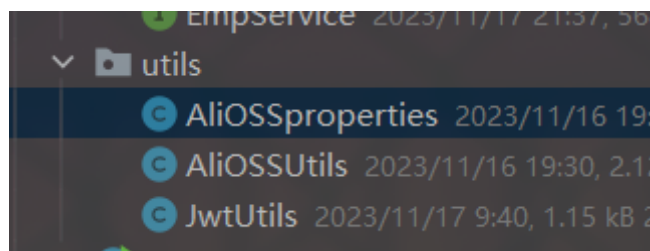
MultipartFile 是Spring框架中用于处理文件上传的接口,通常用于在Web应用中接收用户上传的文件。

1. **用途:** 主要用于接收HTTP请求中的文件数据,比如HTML表单中的文件上传部分。
2. **特性:** MultipartFile 封装了上传文件的信息,包括文件名、文件内容、文件类型等。

**与Spring Boot结合:**

- 在Spring Boot应用程序中,你无需进行过多的配置, Spring Boot会自动为文件上传提供支持。只需在Controller中使用 MultipartFile 参数即可。

-----创建utils工具包 存放从外部引进来的工具类



```

@Data
@Component
@ConfigurationProperties(prefix = "aliyun.oss") //通过这个注解和yml配置文件关联起来了
public class AliOSSProperties {
    private String endpoint;
    private String accessKeyId;
    private String accessKeySecret;
    private String bucketName;
}

```

这个工具类的话**官方文档**也有的

```

@Slf4j
@Component
public class AliOSSUtils {
    // @value("${aliyun.oss.endpoint}")
    // private String endpoint;
    // @value("${aliyun.oss.accessKeyId}")
    // private String accessKeyId;
    // @value("${aliyun.oss.accessKeySecret}")
    // private String accessKeySecret;
    // @value("${aliyun.oss.bucketName}")
    // private String bucketName;

    /**
     * 实现上传图片到OSS
     */
    @Autowired
    AliOSSProperties alioSSProperties; //一次性引入一堆的话就用这个，上面注释的情况是用注入的信息比较少的时候就直接@value
    public String upload(MultipartFile file) throws IOException {
        String accessKeyId = alioSSProperties.getAccessKeyId();
        String bucketName = alioSSProperties.getBucketName();
        String accessKeySecret = alioSSProperties.getAccessKeySecret();
        String endpoint = alioSSProperties.getEndpoint();
        // 获取上传的文件的输入流
        InputStream inputStream = file.getInputStream();

        // 避免文件覆盖 - 文件命名时用到了UUID
        String originalFilename = file.getOriginalFilename();
        String fileName = UUID.randomUUID().toString() +
        originalFilename.substring(originalFilename.lastIndexOf("."));

        //上传文件到 OSS
        OSS ossClient = new OSSClientBuilder().build(endpoint, accessKeyId,
        accessKeySecret);
        ossClient.putObject(bucketName, fileName, inputStream);

        //文件访问路径
        String url = endpoint.split("///")[0] + "///" + bucketName + "." +
        endpoint.split("///")[1] + "/" + fileName;
        // 关闭ossClient
        ossClient.shutdown();
        return url; // 把上传到oss的路径返回
    }
}

```

### 3.4.修改(更新update)

编辑员工

×

用户名

sigonghuiye

员工姓名


四宫辉夜

性别

女

▼

头像



职位

学工主管

▼

入职日期

自 2023-11-13

归属部门

学生会

▼

提交

取消

-----这里涉及到一个信息的回显,所以有一个**根据ID查询对应信息**的业务 比较简单不讲了

Controller层代码

```
@PutMapping
public Result updateData(@RequestBody Emp emp) {
    log.info("修改信息的员工id:{}", emp.getId());
    empService.updateData(emp);
    return Result.success();
}
```

Service层代码

```
public void updateData(Emp emp) {
    //主要是要注意更新时间
    emp.setUpdateTime(LocalDateTime.now());
    empMapper.updateData(emp);
}
```

Mapper层代码

```
void updateData(Emp emp);
```

```
<update id="updateData" parameterType="com.example.tliasproject.pojo.Emp">
    update emp
    <set>
        <if test="username != null and username != ''">username=#{username},</if>
        <if test="name != null and name != ''">name=#{name},</if>
        <if test="image != null and image != ''">image=#{image},</if>
        <if test="job != null and job != ''">job=#{job},</if>
        <if test="gender != null and gender != ''">gender=#{gender},</if>
        <if test="entrydate != null">entrydate=#{entrydate},</if>
        <if test="deptId != null">dept_id=#{deptId},</if>
    </set>
    where id = #{id}
</update>
```

## 4.完善系统

### 4.1.登录校验 --jwt令牌

#### 4.1.1.jwt说明

在用户登录后，生成一个包含用户信息的 JWT，将其返回给客户端。客户端在后续请求中携带该令牌，服务端通过验证 JWT 来识别用户身份。

JWT 通常由三部分组成：

1. **Header (头部)**：包含了两部分信息，令牌的类型 (JWT) 和使用的签名算法，通常使用 HS256

```
{ "alg": "HS256", "typ": "JWT" }
```

2. **Payload (负载)**：包含了声明 (claims)。声明是关于实体 (通常是用户) 和其他数据的声明。有三种类型的声明：注册声明、公共声明和私有声明。

```
jsonCopy code{
  "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516239022
}
```

3. **Signature (签名)**：使用Base64编码的头部和负载，以及一个密钥，通过指定的算法生成的签名。用于验证消息的完整性。

```
HMACSHA256(
  base64urlEncode(header) + "." +
  base64urlEncode(payload),
  secret)
```

### 4.1.2.具体应用

pom.xml配置 --看官方文档

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

JWT解析工具类 --看官方文档

```
public class JwtUtils {

    private static String signKey = "itheima"; //签名密钥 --自定义的
    private static Long expire = 43200000L; //有效时间 单位(ms)

    /**
     * 生成JWT令牌
     * @param claims JWT第二部分负载 payload 中存储的内容
     * @return 返回jwt令牌 --一串东西
     */
    public static String generateJwt(Map<String, Object> claims){
        String jwt = Jwts.builder()
            .addClaims(claims)
            .signWith(SignatureAlgorithm.HS256, signKey)
            .setExpiration(new Date(System.currentTimeMillis() + expire))
            .compact();
        return jwt;
    }

    /**
     * 解析JWT令牌
     * @param jwt JWT令牌
     * @return 返回JWT第二部分负载 payload 中存储的内容
     */
    public static Claims parseJWT(String jwt){
        Claims claims = Jwts.parser()
            .setSigningKey(signKey)
            .parseClaimsJws(jwt)
            .getBody();
        return claims;
    }
}
```

Controller层代码

```
@Slf4j
@RestController
@RequestMapping("/login")
public class LoginController {

    @Autowired
    EmpService empService;

    @PostMapping
```

```

public Result login(@RequestBody Emp emp) {
    log.info("正在实现Login,username:{},password:{}", emp.getUsername(),
emp.getPassword());
    Emp e = empService.login(emp);

    if (e != null) {
        Map<String, Object> claims = new HashMap<>();
        claims.put("id", e.getId());
        claims.put("username", e.getUsername());
        String jwt = JwtUtils.generateJwt(claims); //送进去生成jwt
        return Result.success(jwt);
    }
    return Result.error("用户名或密码错误!");
}
}

```

Service层代码

```

@Override
public Emp login(Emp emp) {
    return empMapper.login(emp);
}

```

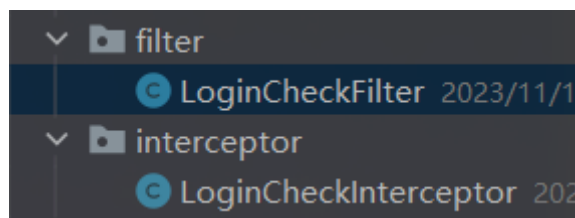
Mapper层代码

```

@Select("select * from emp where username = #{username} and password = #{password}")
Emp login(Emp emp);

```

## 4.2.登录校验 --拦截 filter和interceptor二选一



[-> 记得在启动类里面加@ServletComponentScan注解 <-]

二者区别: 作用的时间不同 --具体看项目概况有讲

两个都用到了一个插件: **fastJSON**

作用:讲结果转成json ---因为创建的类不是Controller,没有带@RestController注解(这个注解能自动把结果转成json返回)

pom.xml配置:

```

<!--      fastJSON工具类让结果转成json-->
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>fastjson</artifactId>
    <version>2.0.32</version>
</dependency>

```

## 拦截的步骤

1. 拿到请求的url
2. 根据url判断是不是**登录**的请求 是直接放行,不用拦截了(因为要登录)  
如果是其他的话就要看看有没有正常登录来决定是否拦截
3. 拿到请求头中带的token --获取jwt令牌
4. 判断token是否为空(**是否未登录**) 如果为空就结束,返回错误信息(**拦截**)
5. 判断token是否有效(**登录信息是否有误**) 如果无效就结束,返回错误信息(**拦截**)
6. 所有判断逻辑通过,放行

### 4.2.1.filter过滤器

-- 这个东西是servlet里面的所以有些参数要处理一下

对于doFilter中的参数说明:

#### 1. servletRequest:

- 类型: `ServletRequest`
- 作用: 表示客户端请求的信息, 是一个接口, 提供了访问请求头、请求参数、请求体等请求信息的方法。在 `doFilter` 方法中, 你可以通过这个参数获取关于客户端请求的各种信息。

#### 2. servletResponse:

- 类型: `ServletResponse`
- 作用: 表示响应到客户端的信息, 也是一个接口, 提供了设置响应头、响应体等响应信息的方法。在 `doFilter` 方法中, 你可以通过这个参数对响应进行处理, 例如设置响应头或者修改响应内容。

#### 3. filterChain:

- 类型: `FilterChain`
- 作用: 表示过滤器链, 通过调用其 `doFilter` 方法可以将请求传递到下一个过滤器, 或者到达目标资源。在 `doFilter` 方法中, 你可以选择是否调用 `filterChain.doFilter` 来继续执行下一个过滤器或目标资源, 或者在不调用的情况下终止请求处理。

```

@Slf4j
@WebFilter(urlPatterns = "/*") //最终的拦截器\过滤器就带上这个注解 --参数:拦截的请求路径
//需要实例化接口Filter , 并且重写doFilter方法
public class LoginCheckFilter implements Filter {
    @Override
    public void doFilter(ServletRequest servletRequest, ServletResponse
servletResponse, FilterChain filterChain) throws IOException, ServletException {

        HttpServletRequest req = (HttpServletRequest) servletRequest; //拿url 拿Header的
时候要用 所以需要强转类型
        HttpServletResponse resp = (HttpServletResponse) servletResponse; //使用强制类型转
换, 你可以将 servletResponse 转换为 HttpServletResponse 对象, 以便在后续的代码中调用
HttpServletResponse 特定的方法。

        //      1.拿到请求的Url
        String reqURL = req.getRequestURI();

```

```

log.info("请求的url:{}", reqURL);

//      2.判断是不是登录 如果是就放行,结束
if (reqURL.contains("/login")) {
    log.info("登录请求,放行");
    filterChain.doFilter(servletRequest, servletResponse);
    return;
}

//      3.拿到token
String jwt = req.getHeader("token");

//      4.token是否为空如果为空直接不给
if (!StringUtils.hasLength(jwt)) {
    log.info("Token长度为0 返回未登录的信息");
    Result error = Result.error("NOT_LOGIN");
    //      到了这一步还不行,要把这个错误结果变成JSON格式 送到Response里面前端才能收到
    //      因为这个不是Controller,不能自动变成Json格式 所以要自己写 这里用到了阿里巴巴的工具类
    fastJson
    //      Maven里面配置了就行
    String notLogin = JSONObject.toJSONString(error);//需要转为json格式才能送给前端
    resp.getWriter().write(notLogin);//向前端响应数据(返回数据) --未成功登录的错误信息
    return;
}

//      5.token是否有效
try {
    JwtUtils.parseJWT(jwt);
} catch (Exception e) {
    e.printStackTrace();
    log.info("token检测无效 返回未登录信息");
    Result error = Result.error("NOT_LOGIN");
    String notLogin = JSONObject.toJSONString(error);
    resp.getWriter().write(notLogin);//向前端响应数据(返回数据) --未成功登录的错误信息
    return;
}

//      6.放行
log.info("一切正常,放行");
filterChain.doFilter(servletRequest, servletResponse);
}
}

```

#### 4.2.2.interceptor拦截器

--这个就是spring的了,可以直接用

要定义拦截器然后注册拦截器

定义拦截器

1. 实例化接口 HandlerInterceptor



2. 里面有三个方法可供重写: preHandle--资源执行前(一般就用这个) postHandle--执行时  
afterCompletion--执行后

## 定义拦截器的代码

```
@Component
@Slf4j
public class LoginCheckInterceptor implements HandlerInterceptor {
    /**
     * @param req      作用: 表示当前的HTTP请求对象, 可以通过它获取请求的信息, 如请求头、请求参数等。
     * @param rsp      作用: 表示当前的HTTP响应对象, 可以通过它设置响应的信息, 如响应头、状态码等。
     * @param handler 作用: 表示当前请求要执行的处理器 (Controller方法)。在 preHandle 方法中,
     你可以根据需要对这个处理器进行判断或者处理。
     * @return boolean 作用: 表示是否允许请求继续执行。如果返回 true, 则继续执行后续的拦截器和处理器
     (Controller); 如果返回 false, 则中断后续的处理, 请求不会到达处理器。
     */
    @Override
    public boolean preHandle(HttpServletRequest req, HttpServletResponse rsp, Object
handler) throws Exception {
        //      1.拿到请求的url
        String reqURL = req.getRequestURI();
        log.info("请求的url:{}", reqURL);

        //      2.判断是不是登录如果是就放行,结束
        if (reqURL.contains("/login")) {
            log.info("登录请求,放行");
            return true;
        }

        //      3.拿到token
        String jwt = req.getHeader("token");

        //      4.token是否为空如果为空直接不给
        if (!StringUtils.hasLength(jwt)) {
            log.info("Token长度为0 返回未登录的信息");
            Result error = Result.error("NOT_LOGIN");
            //      到了这一步还不行,要把这个错误结果变成JSON格式 送到Response里面前端才能收到
            //      因为这个不是Controller,不能自动变成Json格式 所以要自己写 这里用到了阿里巴巴的工具类
            fastJson
            //      Maven里面配置了就行
            String notLogin = JSONObject.toJSONString(error);
            rsp.getWriter().write(notLogin);
            return false;
        }

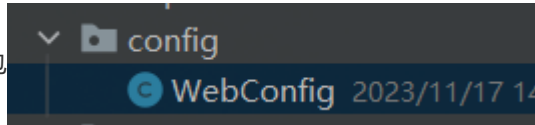
        //      5.token是否有效
        try {
            JwtUtils.parseJWT(jwt);
        } catch (Exception e) {
            e.printStackTrace();
            log.info("token检测无效 返回未登录信息");
            Result error = Result.error("NOT_LOGIN");
            String notLogin = JSONObject.toJSONString(error);
            rsp.getWriter().write(notLogin);
            return false;
        }

        //      6.放行
        log.info("一切正常,放行");
        return true;
    }
}
```

```
}  
}
```

#### 注册拦截器

1. 建包



2. 相当于在这个类中用拦截器

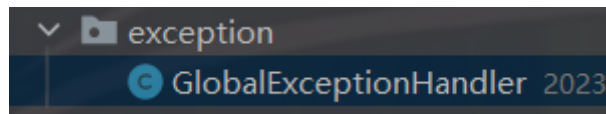
#### 注册拦截器的代码

```
@Configuration //最终的拦截器带上这个注解  
public class WebConfig implements WebMvcConfigurer {  
  
    @Autowired  
    LoginCheckInterceptor loginCheckInterceptor; //注入刚刚定义的拦截器  
  
    @Override  
    public void addInterceptors(InterceptorRegistry registry) {  
        registry.addInterceptor(loginCheckInterceptor)  
            .addPathPatterns("/**") //增加需要拦截的请求路径  
            .excludePathPatterns("/login"); //剔除不需要拦截的请求路径  
    }  
}
```

### 4.3.异常处理

像是数据库中指定唯一的地方,又新增了个重名的 那么肯定会有异常,

除此之外项目还有很多的异常 那么这里就把所有的异常扔到一个类中处理就OK(全局异常处理).



```
@RestControllerAdvice //注解的意思是:全局异常处理器  
public class GlobalExceptionHandler {  
    @ExceptionHandler(Exception.class) //指定捕获的是哪一类型的异常 --这里填了捕获全部异常  
    public Result ex(Exception ex){  
        ex.printStackTrace();  
        return Result.error("操作失败,请联系管理员");  
    }  
}
```

## 4.4.事务管理 - 删除部门时相关员工连带删除

关键:

@Transactional(rollbackFor = Exception.class)

@Transactional(propagation = Propagation.REQUIRES\_NEW)

部门删除代码:

```
@Transactional(rollbackFor = Exception.class) //事务相关的注解
//rollback属性设置的是何种异常回滚 --默认是RuntimeException.class
@Override
public void deleteById(String id) {
    try {
        deptMapper.deleteById(id);
        // Float i = 1/0; //异常
        empMapper.deleteByDeptID(Integer.valueOf(id));
    } finally {
        DeptLog log = new DeptLog();
        log.setCreateTime(LocalDateTime.now());
        log.setDescription("解散部门id:" + id);
        deptLogMapper.insert(log);
    }
}
```

Q: 为什么要事务管理?

A: 因为有可能出现deptMapper.deleteById() 与 empMapper.deleteByDeptID() 两行代码之间出现异常, 导致后面的没有执行的可能

Q: 为什么新建一个DeptLog类?以及相关Controller层 Service层 Mapper层代码?

A: 删除部门属于危险操作 需要记录操作日志 (当然,记录操作日志也会4.5提及的AOP编程,这里只是刚好讲到事务管理)

DeptLog Service层代码 --提到了事务管理的注解

```
@Service
public class DeptLogServiceImpl implements DeptLogService {
    @Autowired
    private DeptLogMapper deptLogMapper;

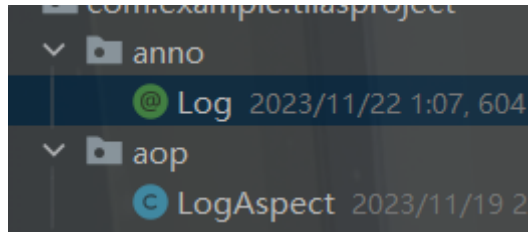
    @Transactional(propagation = Propagation.REQUIRES_NEW) //这个是开个新事物 无论成功与否都记录日志
    //REQUIRED: 大部分情况下都是用该传播行为即可。 --上一层事务回滚就回滚
    //REQUIRES_NEW: 当我们不希望事务之间相互影响时, 可以使用该传播行为。比如: 下订单前需要记录日志, 不论订单保存成功与否, 都需要保证日志记录能够记录成功。
    @Override
    public void insert(DeptLog deptLog) {
        deptLogMapper.insert(deptLog);
    }
}
```

## 4.5.记录操作日志 --AOP切面编程

可以联想为AOE技能? --可以同时作用于多个方法的统一**功能增强**,比如记录日志

AOP核心概念

**连接点:** JoinPoint,可以被AOP控制的方法 (暗含方法执行时的相关信息)  
**通知:** Advice,指哪些重复的逻辑,也就是共性功能 (最终体现为一个方法)  
**切入点:** PointCut,匹配连接点的条件,通知仅会在切入点方法执行时被应用  
**切面:** Aspect,描述通知与切入点的对应关系 (通知+切入点)  
**目标对象:** Target,通知所应用的对象



pom.xml配置

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

Log注解 --之后用到的时候@一下就行

```
/**
 * 基于注解进行切面编程
 * 这个注解相当于标记了用哪个aop的方法,要用的时候在类上面@一下这个注解就行
 */
@Retention(RetentionPolicy.RUNTIME) //何时生效? 运行时有效
@Target(ElementType.METHOD) //技能作用范围? 作用于[方法]上
public @interface Log {
    // 这个只是个标记 真正的业务交给aop切面类来写~
}
```

LogAspect切面类业务编写

```
/**
 * 切面类,业务是:记录操作日志
 * 封装到注解 @Log 里面
 */
@Slf4j
@Component
@Aspect //切面类 也就是相当于AOP的实体类
public class LogAspect {

    @Autowired
    private OperateLogMapper operateLogMapper; //得调用里面的方法操作数据库
    @Autowired
    HttpServletRequest request; //拿到http请求对象 也就是为了拿jwt令牌~

    @Around("@annotation(com.example.tliasproject.anno.Log)")//环绕通知
    //与此相关的注解还有@Before、@After (作用的时机不同)

    //需要填[形参 joinPoint]才能用到作用的方法捏
    public Object recordLog(ProceedingJoinPoint joinPoint) throws Throwable {
        // 写业务了哥们,想拿[属性]直接在[jwt令牌]那里拿,就很方便
    }
}
```

```

//      OK开始拿jwt令牌
String jwt = request.getHeader("token");
//      解析一下
Map<String, Object> claims = JwtUtils.parseJWT(jwt);
//      操作人的ID
Integer operateUser = (Integer) claims.get("id");
//      操作时间
LocalDateTime operateTime = LocalDateTime.now();

//操作类名
String className = joinPoint.getTarget().getClass().getName();
// 操作方法名
String methodName = joinPoint.getSignature().getName();
// 操作方法参数
Object[] args = joinPoint.getArgs();
String methodParams = Arrays.toString(args);
//操作返回值(必须是JSON格式才行,所以要转一下 - 用fastJSON) --这个不是Controller没有带@Rest注解
long begin = System.currentTimeMillis();
Object result = joinPoint.proceed();//必须拿到原始目标方法的对象,因为哪怕没有业务写也要
return回去啊
String returnValue = JSONObject.toJSONString(result);
// 操作耗时
long end = System.currentTimeMillis();
long costTime = end - begin;

//      最后总结:记录操作日志
OperateLog operateLog = new OperateLog(null, operateUser, operateTime,
className, methodName, methodParams, returnValue, costTime);//构造的时候填进去
operateLogMapper.insert(operateLog);
log.info("操作日志:{}", operateLog);

return result;
}
}

```

## 注意

这里我使用的方法是基于注解 @Log 的AOP编程, --

**@Around("@annotation(com.example.tliasproject.anno.Log)")**

也就是**不先指定**作用的方法,以后哪个方法需要用的时候在那个方法上面加@Log

例如做删除员工操作时Controller层代码:

```

@Log //加入了注解 将执行切面类中的方法
@DeleteMapping("/{ids}")
public Result deleteData(@PathVariable ArrayList<Integer> ids) {
    log.info("删除数据id:{}", ids);
    empService.deleteData(ids);
    return Result.success();
}

```

与此对应的是: **先指定**作用的方法

**@Around("execution(\* com.example.tliasproject.service..(..)")**

如果这样写的话就直接在切面类写完就得了

## 事务管理与AOP的联系

事务管理往往涉及到横切关注点，而AOP正是用于处理这些横切关注点的一种编程范式。

### 联系：

1. **事务管理是AOP的一种应用：** 事务管理可以看作是AOP的一种应用，因为事务往往需要在方法执行前、执行后以及发生异常时执行不同的操作，这正是AOP的通知所提供的功能。
2. **AOP用于解耦事务逻辑：** AOP可以用来将事务管理的横切关注点从业务逻辑中抽离，实现业务逻辑和事务逻辑的解耦。通过将事务管理的代码抽象到AOP切面中，业务逻辑可以更专注于核心功能。

### 区别：

1. **目的和关注点不同：** 事务管理的主要目的是确保一系列操作要么全部成功执行（提交事务），要么全部失败回滚（回滚事务）。而AOP更广泛地涉及到在应用程序执行的不同点插入横切关注点的概念，不仅仅局限于事务管理。
2. **AOP更广泛的应用场景：** AOP不仅仅用于事务管理，还可以用于其他横切关注点，如日志记录、权限控制、性能监控等。而事务管理主要关注于保障数据一致性的场景。
3. **配置和使用方式：** 事务管理通常涉及使用 `@Transactional` 注解或者XML配置来声明事务属性。AOP的配置方式可以是基于注解的，也可以是基于XML的。在Spring Boot项目中，通常使用基于注解的AOP配置更为方便，而事务管理可以通过在Service类或方法上使用 `@Transactional` 注解来实现。

## 更深入学习的知识点=====

### 1.Bean

#### bean的获取

默认情况下，Spring项目启动时，会把bean都创建好放在IOC容器中，如果想要主动获取这些bean,可以通过如下方式：

根据name获取bean: `Object getBean(String name)`

根据类型获取bean: `T getBean(ClassrequiredType)`

根据name获取bean(带类型转换): `T getBean(String name,ClassrequiredType)`

#### bean的作用域

可以通过@Scope注解来进行配置作用域：

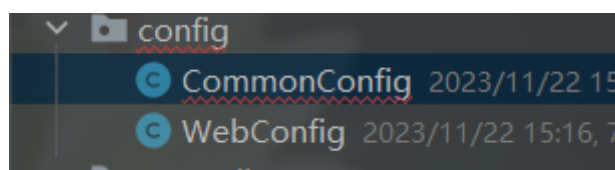
属性 singleton 容器内同名称的bean只有一个实例（单例）（默认）

prototype 每次使用该bean时会创建新的实例（非单例）

#### \*第三方bean

其实就是自己写的类要交给IOC管理直接加@Component 很方便

但是如果是第三方的类呢?它是只读的啊，所以这个时候我们就要转门建一个配置类,对第三方类进行集中管理



```

@Configuration
public class CommonConfig {
    //声明第三方bean
    @Bean//将当前方法的返回值对象交给Ioc容器管理，成为Ioc容器bean
    //通过Bean.注解的name/value属性指定bean名称，如果未指定，默认是方法名
    public SAXReader saxReader() {
        return new SAXReader;
    }
}

```

```

/**
 * 切面类,业务是:记录操作日志
 * 封装到注解 @Log 里面
 */
@Slf4j
@Component
@Aspect //切面类 也就是相当于AOP的实体类
public class LogAspect {

    @Autowired
    private OperateLogMapper operateLogMapper; //得调用里面的方法操作数据库
    @Autowired
    HttpServletRequest request; //拿到http请求对象 就是为了拿jwt令牌~

    @Around("@annotation(com.example.tliasproject.anno.Log)")//环绕通知 需要填[形参]才能用到
    作用的方法捏
    public Object recordLog(ProceedingJoinPoint joinPoint) throws Throwable {
        // 写业务了哥们,想拿[属性]直接在[jwt令牌]那里拿,就很方便
        // OK开始拿jwt令牌
        String jwt = request.getHeader("token");
        // 解析一下
        Map<String, Object> claims = JwtUtils.parseJWT(jwt);
        // 操作人的ID
        Integer operateUser = (Integer) claims.get("id");
        // 操作时间
        LocalDateTime operateTime = LocalDateTime.now();
        // 操作类名
        String className = joinPoint.getTarget().getClass().getName();
        // 操作方法名
        String methodName = joinPoint.getSignature().getName();
        // 操作方法参数
        Object[] args = joinPoint.getArgs();
        String methodParams = Arrays.toString(args);
        // 操作返回值(必须是JSON格式才行,所以要转一下 - 用fastJSON)
        long begin = System.currentTimeMillis();
        // 必须拿到原始目标方法的对象,因为哪怕没有业务写也要return回去啊
        Object result = joinPoint.proceed();
        String returnValue = JSONObject.toJSONString(result);
        // 操作耗时
        long end = System.currentTimeMillis();
        long costTime = end - begin;

        // 最后总结:记录操作日志
        OperateLog operateLog = new OperateLog(null, operateUser, operateTime,
        className, methodName, methodParams, returnValue, costTime);//pojo包里面新建一个类,构造的时
        候填进去
        operateLogMapper.insert(operateLog);
        log.info("操作日志:{}",operateLog);

        return result;
    }
}

```

```
}  
}
```

## 2.SpringBoot原理

本质上就是把开发好的功能转成一个依赖,这样别人需要用的时候直接导入就行了(就是自己写依赖)

找依赖的过程:本地仓库 -> 私服(Maven会讲) -> 中央仓库

这里以将图片上传到阿里云OSS为例子:

自定义starter

### 需求

需求: 自定义aliyun-oss-spring-boot-starter,完成阿里云OSS操作工具类AliyunOSSUtils的自动配置。

目标: 引入起步依赖引入之后,要想使用阿里云OSS,注入AliyunOSSUtils.直接使用即可。

### 步骤

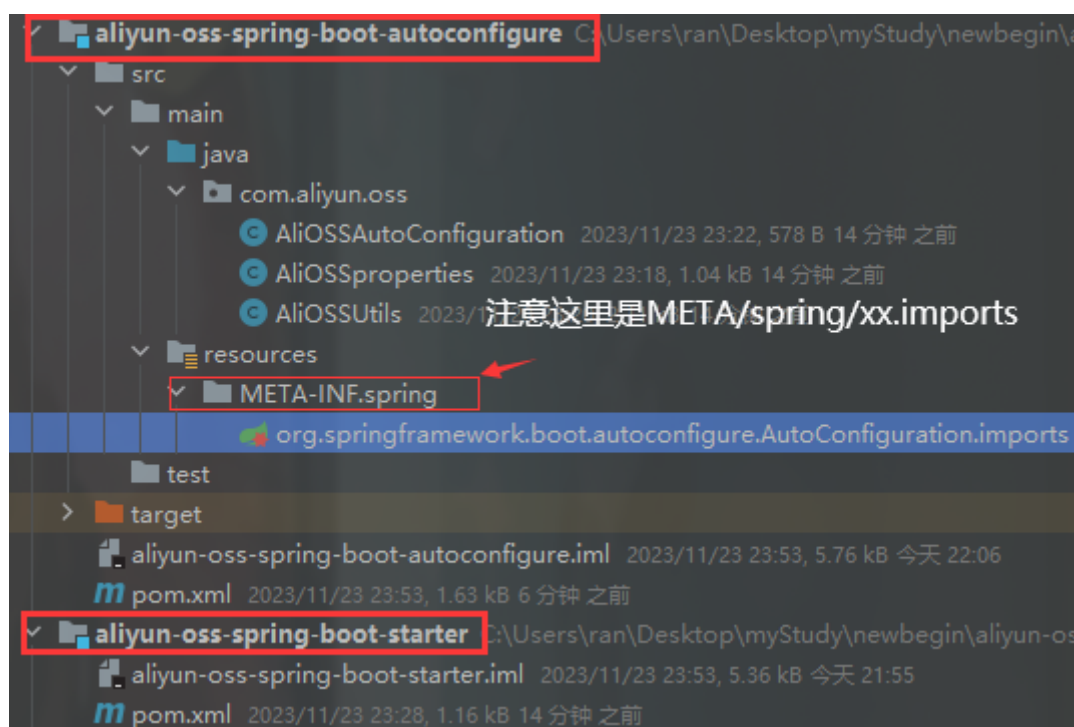
创建aliyun-oss-spring-boot-starter模块

创建aliyun-oss-spring-boot-autoconfigure模块,在starter中引入该模块

在aliyun-oss-spring-boot-autoconfigure模块中的定义自动配置功能,并定义自动配置文件META-INF/spring/xxx.imports

自动配置功能: aliyun-oss-spring-boot-autoconfigure [在这里配置写主要的代码/

依赖管理功能: aliyun-oss-spring-boot-starter [这里这是最后的标明]



### 2.1.autoconfigure配置

这里pom.xml记得导入阿里云OSS的起步依赖 - 不然用不了

```
<!--阿里云OSS 起步依赖 - 导入才能用刚刚自己写的-->  
<dependency>  
  <groupId>com.aliyun.oss</groupId>
```



```

        <artifactId>aliyun-sdk-oss</artifactId>
        <version>3.15.1</version>
    </dependency>

    <dependency>
        <groupId>javax.xml.bind</groupId>
        <artifactId>jaxb-api</artifactId>
        <version>2.3.1</version>
    </dependency>
    <dependency>
        <groupId>javax.activation</groupId>
        <artifactId>activation</artifactId>
        <version>1.1.1</version>
    </dependency>
    <!-- no more than 2.3.3-->
    <dependency>
        <groupId>org.glassfish.jaxb</groupId>
        <artifactId>jaxb-runtime</artifactId>
        <version>2.3.3</version>
    </dependency>

```

```

@Configuration //配置类
@EnableConfigurationProperties(AliOSSproperties.class)
public class AliOSSAutoConfiguration {
    @Bean //相当于变成第三方类交给IOC容器 然后别的地方就可以注入
    public AliOSSutils alioSSutils(AliOSSproperties alioSSproperties) {
        AliOSSutils alioSSutils = new AliOSSutils();
        alioSSutils.setAliOSSproperties(alioSSproperties);
        return alioSSutils;
    }
}

```

```

@ConfigurationProperties(prefix = "aliyun.oss")
public class AliOSSproperties {
    private String endpoint;
    private String accessKeyId;
    private String accessKeySecret;
    private String bucketName;
    //这里没有@Data标签啊 因为省得加lombok插件(怕和原项目冲突) 所以要自己getter/setter
    //再说一下: 要自己getter/setter !!!
}

```

```

public class AliOSSutils {
    /**
     * 实现上传图片到OSS
     */
    AliOSSproperties alioSSproperties; //自己getter/setter
    public AliOSSproperties getAliOSSproperties() {
        return alioSSproperties;
    }

    public void setAliOSSproperties(AliOSSproperties alioSSproperties) {
        this.alioSSproperties = alioSSproperties;
    }

    public String upload(MultipartFile file) throws IOException {
        String accessKeyId = alioSSproperties.getAccessKeyId();
    }
}

```

```

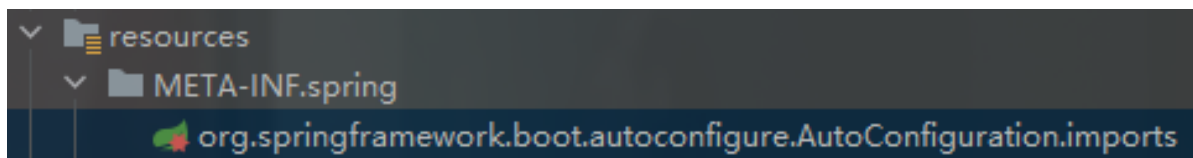
String bucketName = alioSSproperties.getBucketName();
String accessKeySecret = alioSSproperties.getAccessKeySecret();
String endpoint = alioSSproperties.getEndpoint();
// 获取上传的文件的输入流
InputStream inputStream = file.getInputStream();

// 避免文件覆盖
String originalFilename = file.getOriginalFilename();
String fileName = UUID.randomUUID().toString() +
originalFilename.substring(originalFilename.lastIndexOf("."));

//上传文件到 OSS
OSS ossClient = new OSSClientBuilder().build(endpoint, accessKeyId,
accessKeySecret);
ossClient.putObject(bucketName, fileName, inputStream);

//文件访问路径
String url = endpoint.split("///")[0] + "///" + bucketName + "." +
endpoint.split("///")[1] + "/" + fileName;
// 关闭ossClient
ossClient.shutdown();
return url;// 把上传到oss的路径返回
}
}

```



这里就放配置的全类名：com.aliyun.oss.AliOSSAutoConfiguration

## 2.2.starter

这里的话就只需要注意pom.xml的配置

```

<dependency>
    <groupId>com.aliyun.oss</groupId>
    <artifactId>aliyun-oss-spring-boot-autoconfigure</artifactId>
    <version>0.0.1-SNAPSHOT</version>
</dependency>

```

## 3.Maven高级

### 3.1.分模块设计 (有白学)

#### 1.什么是分模块设计？

将项目按照功能拆分成若干个子模块

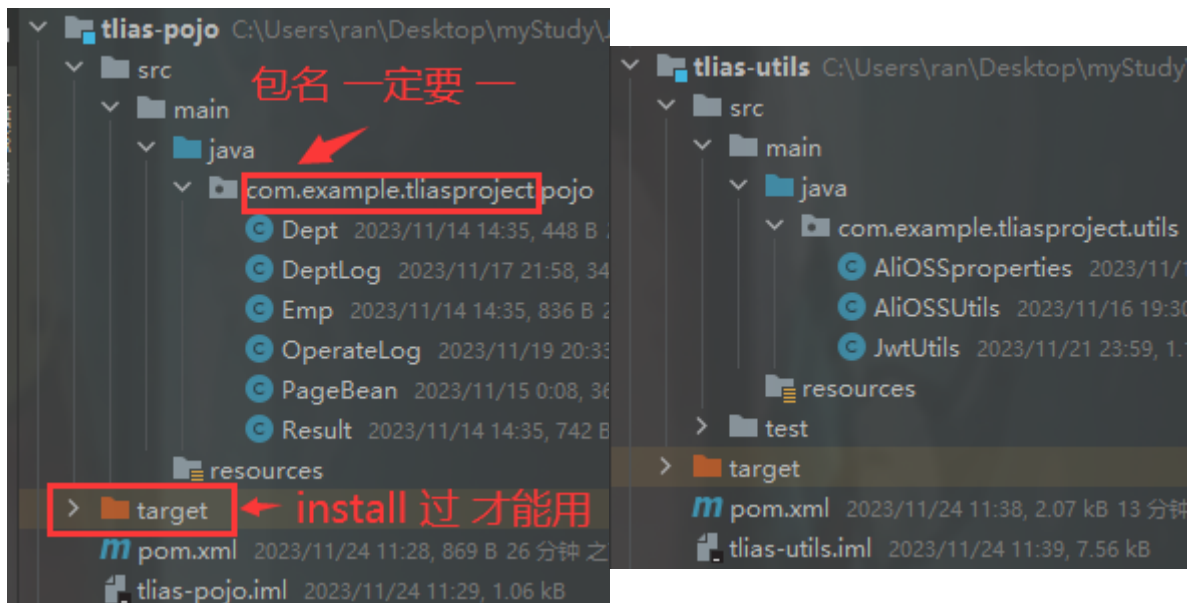
#### 2.为什么要分模块设计？

方便项目的管理维护、扩展，也方便模块间的相互调用，资源共享

#### 3.注意事项

分模块设计需要先针对模块功能进行设计，再进行编码。不会先将工程开发完毕，然后进行拆分

本质上就是为了**协同开发**, 许多人一起开发同一个大项目的时候要分出一个个模块 最后再总项目对模块进行集成



这里将项目中的pojo包和utils包分成模块了

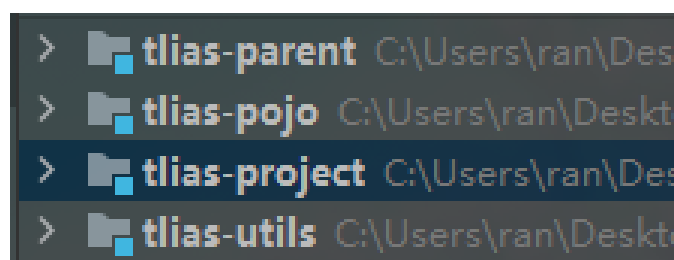
注意! 1. 所有模块的组名, 包名一定要一致!!!

2. 所有模块写完之后要在 Maven 里面用 install 插件 生成 jar 包之后才能正常被导入 [白学, 3.2.3.聚合关系 那里会有更好的解决办法]

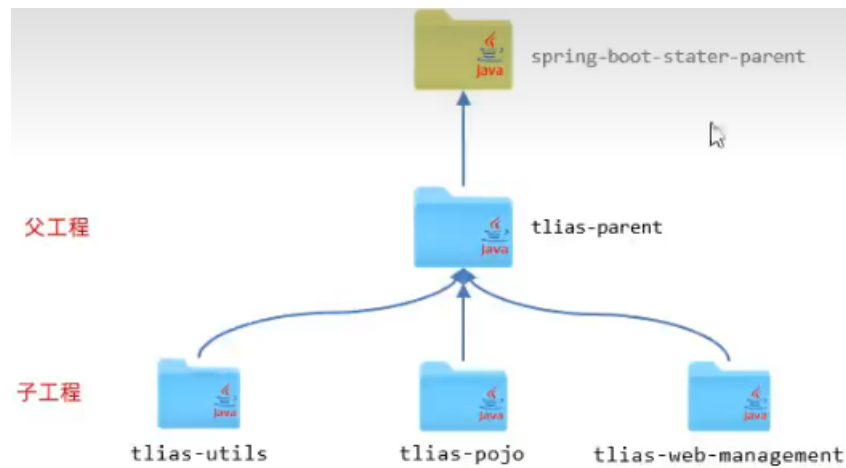
总项目想用这个模块 那么pom.xml文件需要配置:

```
<!-- 自己写的or导入模块-->
<dependency>
  <groupId>com.example</groupId>
  <artifactId>tlias-pojo</artifactId>
  <version>1.0-SNAPSHOT</version>
</dependency>
<dependency>
  <groupId>org.example</groupId>
  <artifactId>tlias-utils</artifactId>
  <version>1.0-SNAPSHOT</version>
</dependency>
```

### 3.2.继承与聚合



### 3.2.1.继承关系的实现



本质上就是对**依赖进行管理**

①.创建maven模块tlia-parent,该工程为父工程，设置打包方式pom。

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.7.17</version>
  <relativePath/> <!-- springboot启动的万恶之源父类 -->
</parent>

<groupId>com.example</groupId>
<artifactId>tlia-parent</artifactId>
<version>1.0-SNAPSHOT</version>
<packaging>pom</packaging> <!-- *打包方式 -->
```

②.在子工程的pom.xml文件中，配置继承关系。

```
<parent>
  <groupId>com.example</groupId>
  <artifactId>tlia-parent</artifactId>
  <version>1.0-SNAPSHOT</version>
  <relativePath>../tlia-parent/pom.xml</relativePath>
</parent>
```

```
<relativePath/> <!-- 不指定的话会在本地仓库/中央仓库找
```

在子工程中，配置了继承关系之后，子工程自己的是可以省略的，因为会自动继承父工程的。

③.在父工程中配置各个工程共有的依赖（子工程会自动继承父工程的依赖）

```
<dependencies>
  <dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <version>1.18.24</version>
  </dependency>
</dependencies>
```

若父子工程都配置了同一个依赖的不同版本，以子工程的为准。

### 3.2.2.版本锁定

在maven中，可以在父工程的pom文件中通过来统一管理依赖版本。

子工程引入依赖时，无需指定版本号，父工程统一管理。变更依赖版本，只需在父工程中统一变更。

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.projectlombok</groupId>
      <artifactId>lombok</artifactId>
      <version>1.18.24</version>
    </dependency>
  </dependencies>
</dependencyManagement>
```

也可以通过配置properties来指定版本号

```
<!--版本锁定-->
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.projectlombok</groupId>
      <artifactId>lombok</artifactId>
      <version>${lombok.version}</version><!--指定成下面说的版本-->
    </dependency>
  </dependencies>
</dependencyManagement>

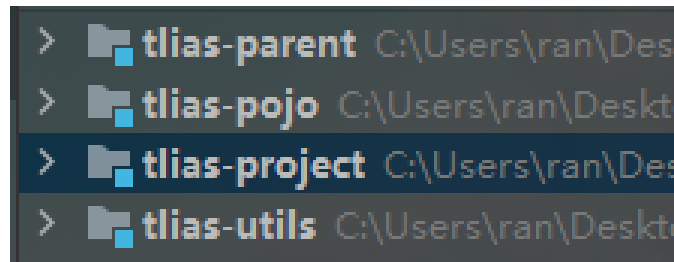
<properties>
  <maven.compiler.source>11</maven.compiler.source>
  <maven.compiler.target>11</maven.compiler.target>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

  <lombok.version>1.18.24</lombok.version><!--自己建一个标签作为版本-->
</properties>
```

是直接依赖，在父工程配置了依赖，子工程会直接继承下来。

是统一管理依赖版本，不会直接依赖，还需要在子工程中引入所需依赖（无需指定版本）

### 3.2.3.聚合关系的实现



**聚合**：将多个模块组织成一个整体，同时进行项目的构建。

**聚合工程**：一个不具有业务功能的“空”工程（有且仅有一个pom文件）

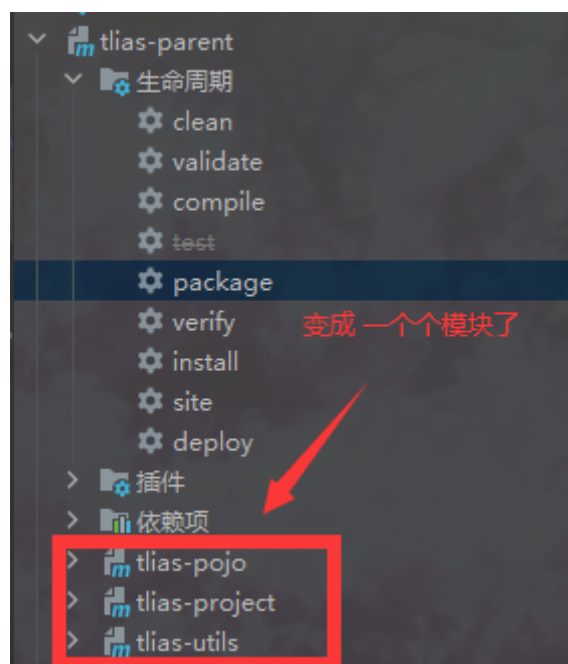
**作用**：快速构建项目（无需根据依赖关系手动构建，直接在聚合工程上构建即可）

**[这是对分模块开发时，最终构建项目的一键优化方案]**

非常简单 只需要对parent的pom.xml里面配置：

```
<!-- 聚合其他模块-->
<modules>
  <module>../tlias-pojo</module>
  <module>../tlias-project</module>
  <module>../tlias-utils</module>
</modules>
```

最终实现的效果：Maven里面



### 3.2.4.总结 - 两者的异同

继承与聚合 -> tlias-parent 既当被继承模块 又当聚合模块

**作用：**

继承：用于简化依赖配置、统一管理依赖

聚合：用于快速构建项目

#### 相同点:

聚合与继承的pom.xml文件打包方式均为pom,可以将两种关系制作到同一个pom文件中(一般都是把项目放在一个parent模块下)

聚合与继承均属于设计型模块,并无实际的模块内容

#### 不同点:

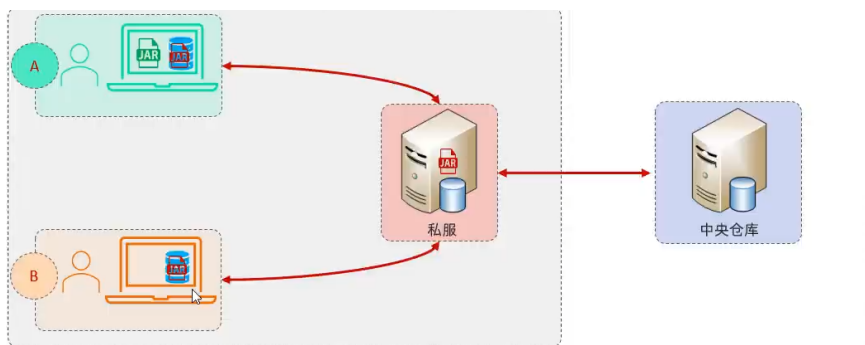
继承是在子模块中配置关系,父模块无法感知哪些子模块继承了自己

聚合是在聚合工程中配置关系,聚合可以感知到参与聚合的模块有哪些

### 3.3.私服

私服是一种特殊的远程仓库,它是架设在局域网内的仓库服务,用来代理位于外部的中央仓库,用于解决团队内部的资源共享与资源同步问题。

#### 3.3.1.个人理解与简单说明

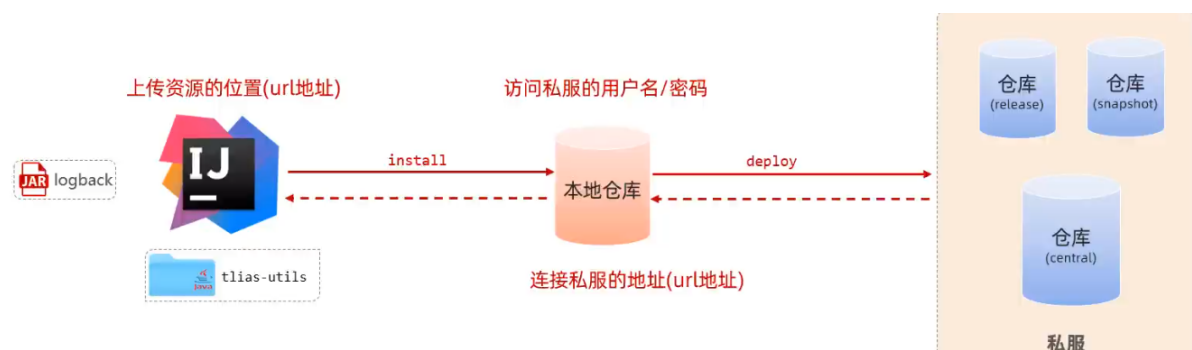


- 依赖查找顺序:
  - 本地仓库
  - 私服
  - 中央仓库

找依赖的过程:先到本地仓库找 没有的话 再到Apache公司的中央仓库里面找

但是公司之间协同开发的话需要用别人电脑里的本地依赖,但是找不到啊

因此如果需要用别人公司的依赖,就需要本地仓库和中央仓库之间再建一个仓库 -> 私服  
两家公司就可以在上面传自己做好的依赖 也可导入别人写的依赖



项目版本:

RELEASE(发行版本): 功能趋于稳定、当前更新停止,可以用于发行的版本,存储在私服中的RELEASE仓库中。

SNAPSHOT(快照版本): 功能不稳定、尚处于开发中的版本,即快照版本,存储在私服的SNAPSHOT仓库中。

### 3.3.2.私服配置说明

访问私服: <http://192.168.150.101:8081>

访问密码: admin/admin

使用私服, 需要在maven的settings.xml配置文件中, 做如下配置:

1. 需要在 **servers** 标签中, 配置访问私服的凭证(访问的用户名和密码)

```
<server>
  <id>maven-releases</id>
  <username>admin</username>
  <password>admin</password>
</server>

<server>
  <id>maven-snapshots</id>
  <username>admin</username>
  <password>admin</password>
</server>
```

2. 在 **mirrors** 中只配置我们自己私服的连接地址(如果之前配置过阿里云, 需要直接替换掉)

```
<mirror>
  <id>maven-public</id>
  <mirrorOf>*</mirrorOf>
  <url>http://192.168.150.101:8081/repository/maven-public/</url>
</mirror>
```

3. 需要在 **profiles** 中, 增加如下配置, 来指定snapshot快照版本的依赖, 依然允许使用

```
<profile>
  <id>allow-snapshots</id>
  <activation>
    <activeByDefault>true</activeByDefault>
  </activation>
  <repositories>
    <repository>
      <id>maven-public</id>
      <url>http://192.168.150.101:8081/repository/maven-public/</url>
      <releases>
        <enabled>true</enabled>
      </releases>
      <snapshots>
        <enabled>true</enabled>
      </snapshots>
    </repository>
  </repositories>
</profile>
```



```
    </repository>
  </repositories>
</profile>
```

4. 如果需要上传自己的项目到私服上，需要在项目的pom.xml文件中，增加如下配置，来配置项目发布的地址(也就是私服的地址)

```
<distributionManagement>
  <!-- release版本的发布地址 -->
  <repository>
    <id>maven-releases</id>
    <url>http://192.168.150.101:8081/repository/maven-releases/</url>
  </repository>

  <!-- snapshot版本的发布地址 -->
  <snapshotRepository>
    <id>maven-snapshots</id>
    <url>http://192.168.150.101:8081/repository/maven-snapshots/</url>
  </snapshotRepository>
</distributionManagement>
```

5. 发布项目，直接运行 deploy 生命周期即可 (发布时，建议跳过单元测试)

### 3.3.3.启动本地私服

1. 解压：apache-maven-nexus.zip
2. 进入目录：apache-maven-nexus\nexus-3.39.0-01\bin
3. 启动服务：双击 start.bat
4. 访问服务：localhost:8081
5. 私服配置说明：将上述配置私服信息的 192.168.150.101 改为 localhost

