尚硅谷JAVA
从0开始的全套资料

JAVA全端工程师系列

适合小白上手的WEB教材

JavaWeb 随堂讲义

尚硅谷随堂配套课件
尚硅谷教育© 编著

扫一扫
获取更多内部资料

尚硅谷IT教育

北京校区：北京市昌平区宏福科技园2号楼3层

深圳校区：深圳市宝安区西部硅谷大厦B座C区1层

上海校区：上海市松江区谷阳北路166号大江商厦3层

武汉校区：武汉市东湖高新区东湖网谷6号楼4层

西安校区：西安市雁塔区和发智能大厦B座3层

成都校区：成都市成华区北辰星拱青创园综合楼3层

第一章 WEB概述

一 JAVAWEB简介 1
二 JAVAWEB技术栈 2
三 JAVAWEB交互模式 2
四 JAVAWEB的CS和BS模式 2
五 JAVAWEB实现前后端分离 3

第二章 HTML&CSS

一 HTML入门 5
1.1 HTML&CSS&JavaScript的作用
1.2 什么是HTML
1.3 什么是超文本
1.4 什么是标记语言
1.5 HTML基础结构
1.6 HTML的入门程序
1.7 HTML概念词汇解释
1.8 HTML的语法规则
1.9 开发工具VsCode的安装和使用
二 HTML常见标签 11
2.1 标题标签
2.2 段落标签
2.3 换行标签
2.4 列表标签
2.5 超链接标签
2.6 多媒体标签
2.7 表格标签(重点)
2.8 表单标签(重点)
2.9 常见表单项标签(重点)
2.10 布局相关标签
2.11 特殊字符
三 CSS的使用 19
3.1 CSS引入方式
3.2 CSS选择器
3.4 CSS浮动
3.5 CSS定位
3.6 CSS盒子模型

第三章 JavaScript

一 JS简介 31
1.1 JS起源
1.2 JS 组成部分
1.3 JS的引入方式
二 JS的数据类型和运算符 33
2.1 JS的数据类型
2.2 JS的变量
2.3 JS的运算符
三 JS的流程控制和函数 35

3.1 JS分支结构
3.2 JS循环结构
3.3 JS函数声明
四 JS的对象和JSON 37
4.1 JS声明对象的语法
4.2 JSON格式
4.3 JS常见对象
4.3.1 Array
4.3.2 Boolean
4.3.3 Date
4.3.4 Math
4.3.5 Number
4.3.6 String
五 事件的绑定 41
5.1 什么是事件
5.2 常见事件
5.3 事件的绑定
5.4 事件的触发
六 BOM编程 42
6.1 什么是BOM
6.2 window对象的常见属性(了解)
6.3 window对象的常见方法(了解)
6.4 通过BOM编程控制浏览器行为演示
6.5 通过BOM编程实现会话级和持久级数据存储
七 DOM编程 46
7.1 什么是DOM编程
7.2 获取页面元素的几种方式
7.2.1 在整个文档范围内查找元素结点
7.2.2 在具体元素节点范围内查找子节点
7.2.3 查找指定子元素节点的父节点
7.2.4 查找指定元素节点的兄弟节点
7.3 操作元素属性值
7.3.1 属性操作
7.3.2 内部文本操作
7.4 增删元素
7.4.1 对页面的元素进行增删操作
八 正则表达式 51
8.1 正则表达式简介
8.2 正则表达式体验
8.2.1 验证
8.2.2 匹配
8.2.3 替换
8.2.4 全文查找
8.2.5 忽略大小写
8.2.6 元字符使用
8.2.7 字符集合的使用
8.2.8 常用正则表达式
九 案例开发-日程管理-第一期 56
9.1 登录页及校验
9.2 注册页及校验

第四章 XML_Tomcat10_HTTP

一 XML 62
1.1 常见配置文件类型
1.1.1 properties配置文件
1.1.2 xml配置文件
1.2 DOM4J进行XML解析
1.2.1 DOM4J的使用步骤
1.2.2 DOM4J的API介绍
二 Tomcat10 64
2.1 WEB服务器
2.2 Tomcat服务器
2.2.1 简介
2.2.2 安装
2.3 Tomcat目录及测试
2.4 WEB项目的标准结构
2.5 WEB项目部署的方式
2.6 IDEA中开发并部署运行WEB项目
2.6.1 IDEA关联本地Tomcat
2.6.2 IDEA创建web工程
2.6.3 IDEA部署-运行web项目
三 HTTP协议 88
3.1 HTTP简介
3.1.1 发展历程
3.1.2 HTTP协议的会话方式
3.1.3 HTTP1.0和HTTP1.1的区别
3.1.4 在浏览器中通过F12抓取请求响应报文包
3.2 请求和响应报文
3.2.1 报文的格式
3.2.2 请求报文
3.2.3 响应报文

第五章 Servlet

一 Servlet简介 98
1.1 动态资源和静态资源
1.2 Servlet简介
二 Servlet开发流程 99
2.1 目标
2.2 开发过程
三 Servlet注解方式配置 102
3.1 @WebServlet注解源码
3.2 @WebServlet注解使用
四 Servlet生命周期 103
4.1 生命周期简介
4.2 生命周期测试
4.3 生命周期总结
五 Servlet继承结构 105
5.1 Servlet 接口
5.2 GenericServlet 抽象类
5.3 HttpServlet 抽象类
5.4 自定义Servlet
六 ServletConfig和ServletContext 108
6.1 ServletConfig的使用

6.2 ServletContext的使用
6.3 ServletContext其他重要API
七 HttpServletRequest 113
7.1 HttpServletRequest简介
7.2 HttpServletRequest常见API
八 HttpServletResponse 114
8.1 HttpServletResponse简介
8.2 HttpServletResponse的常见API
九 请求转发和响应重定向 116
9.1 概述
9.2 请求转发
9.3 响应重定向
十 web乱码和路径问题总结 120
10.1 乱码问题
10.1.1 HTML乱码问题
10.1.2 Tomcat控制台乱码
10.1.3 请求乱码问题
10.1.3 响应乱码问题
10.2 路径问题
10.2.1 前端路径问题
10.2.2 重定向中的路径问题
10.2.3 请求转发中的路径问题
十一 MVC架构模式 135
十二 案例开发-日程管理-第二期 136
12.1 项目搭建
12.1.1 数据库准备
12.1.2 项目结构
12.1.3 导入依赖
12.1.4 pojo包处理
12.1.5 dao包处理
12.1.6 service包处理
12.1.7 controller包处理
12.1.8 加密工具类的使用
12.1.9 页面文件的导入
12.3 业务代码
12.3.1 注册业务处理
12.3.2 登录业务处理

第六章 会话_过滤器_监听器_AJAX

一 会话 150
1.1 会话管理概述
1.1.1 为什么需要会话管理
1.1.2 会话管理实现的手段
1.2 Cookie
1.2.1 Cookie概述
1.2.2 Cookie的使用
1.2.2 Cookie的时效性
1.2.3 Cookie的提交路径
1.3 Session
1.3.1 HttpSession概述
1.3.2 HttpSession的使用

1.3.3 HttpSession时效性
1.4 三大域对象
1.4.1 域对象概述
1.4.2 域对象的使用
二 过滤器 162
2.1 过滤器概述
2.2 过滤器使用
2.3 过滤器生命周期
2.4 过滤器链的使用
2.5 注解方式配置过滤器
三 监听器 171
3.1 监听器概述
3.2 监听器的六个主要接口
3.2.1 application域监听器
3.2.2 session域监听器
3.2.3 request域监听器
3.3 session域的两个特殊监听器
3.3.3 session绑定监听器
3.3.4 钝化活化监听器
四 案例开发-日程管理-第三期 180
4.1 过滤器控制登录校验
五 AJAX 182
5.1 什么是AJAX
5.2 如何实现AJAX请求
六 案例开发-日程管理-第四期 183
6.1 注册提交前校验用户名是否占用功能

第七章 前端工程化

一 前端工程化开篇 190
1.1 什么是前端工程化
1.2 前端工程化实现技术栈
二 ECMA6Script 190
2.1 es6的介绍
2.2 es6的变量和模板字符串
2.3 es6的解构表达式
2.4 es6的箭头函数
2.5 es6的对象创建和拷贝
2.6 es6的模块化处理
三 前端工程化环境搭建 201
3.1 Nodejs的简介和安装
3.2 npm 配置和使用
四 Vue3简介和快速体验 205
4.1 Vue3介绍
4.2 Vue3快速体验(非工程化方式)
五 Vue3通过Vite实现工程化 207
5.1 Vite的介绍
5.2 Vite创建Vue3工程化项目
六 Vue3视图渲染技术 215
6.1 模版语法
6.2 响应式基础
6.3 条件和列表渲染

6.4 双向绑定
6.5 属性计算
6.6 数据监听器
6.7 Vue生命周期
6.8 Vue组件
七 Vue3路由机制Router 238
7.1 路由简介
7.2 路由入门案例
7.3 路由重定向
7.4 编程式路由(useRouter)
7.5 路由传参(useRoute)
7.6 路由守卫
八 案例开发-日程管理-第五期 248
8.1 重构前端工程
九 Vue3数据交互Axios 257
9.0 预讲知识-Promise
9.1 Axios介绍
9.2 Axios 入门案例
9.3 Axios get和post方法
9.4 Axios 拦截器
十 案例开发-日程管理-第六期 268
10.1 前端代码处理
10.2 后端代码处理
十一 Vue3状态管理Pinia 277
11.1 Pinia介绍
11.2 Pinia基本用法
11.3 Pinia其他细节
十二 案例开发-日程管理-第七期 283
12.1 前端使用Pinia存储数据
12.2 显示所有日程数据
12.3 增加和修改日程数据
12.5 删除日程数据
十三 Element-plus组件库 301
13.1 Element-plus介绍
13.2 Element-plus入门案例
13.3 Element-plus常用组件

第八章 微头条项目开发

一 项目简介 304
1.1 微头条业务简介
1.2 技术栈介绍
1.3 功能展示
二 前端项目环境搭建 308
三 后端项目环境搭建 309
3.1 数据库准备
3.2 搭建项目
3.2.1 创建WEB项目
3.2.2 导入依赖
3.2.3 准备包结构
3.3 准备工具类
3.3.1 异步响应规　范格式类

3.3.2 MD5加密工具类
3.3.3 JDBCUtil连接池工具类
3.3.4 JwtHelper工具类
3.3.5 JSON转换的WEBUtil工具类
3.4 准备各层的接口和实现类
3.4.1 准备实体类和VO对象
3.4.2 DAO层接口和实现类
3.4.3 Service层接口和实现类
3.4.4 Controller层接口和实现类
3.5 开发跨域CORS过滤器
四 PostMan测试工具 325
4.1 什么是PostMan
4.2 怎么安装PostMan
4.3 怎么使用PostMan
五 登录注册功能 328
5.1 登录表单提交
5.2 根据token获取完整用户信息
5.3 注册时用户名占用校验
5.4 注册表单提交
六 头条首页功能 339
6.1 查询所有头条分类
6.2 分页带条件查询所有头条
6.3 查看头条详情
七 头条发布修改和删除 350
7.1 登录校验
7.2 提交发布头条
7.3 修改头条回显
7.4 保存修改
7.5 删除头条

扫描二维码，查看教材对应课程
或者访问：https://www.bilibili.com/video/BV1UN411x7xe/

第一章 WEB概述

1.1 JAVAWEB简介

> 用 Java 技术来解决相关 web互联网领域 的技术栈，使用 JAVAEE 技术体系开发企业级互联网项目。项目规模和架构模式与JAVASE阶段有着很大的差别，在互联网项目下，首先需要明白 客户端 和 服务器 的概念。

> 客户端 ： 与用户进行交互，用于接收用户的输入(操作)、展示服务器端的数据以及向服务器传递数据 。如手机App，微信小程序，PC端程序，PC浏览器，以及一些其他设备。

> 服务端：与客户端进行交互，接收客户端的数据、处理具体的业务逻辑、传递给客户端其需要的数据。

- “服务器”是一个非常宽泛的概念。 从硬件而言： 服务器是计算机的一种，它比普通计算机运行更快、负载更高、价格更贵。服务器在网络中为其它客户机（如PC机、智能手机、ATM等终端甚至是火车系统等大型设备）提供计算或者应用服务。 从软件而言：服务器其实就是安装在计算机上的一个软件，根据其作用的不同又可以分为各种不同的服务器，例如应用服务器、数据库服务器、Redis服务器、DNS服务器、ftp服务器等等。

综上所述: 用我们自己的话来总结的话，服务器其实就是一台(或者一个集群) 安装了服务器软件的高性能计算机 。

1.2 JAVAWEB技术栈

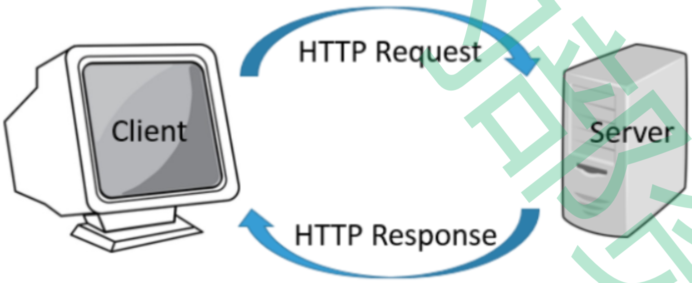
> 客户端-前端部分：

HTML CSS JavaScript ES6 Nodejs npm vite vue3 router pinia axios element-plus ...

> 服务端-后端部分：

HTTP xml Tomcat Servlet   Request   Response Cookie Sesssion Filter Listener MySQL JDBC Druid Jackson lombok jwt ...

1.3 JAVAWEB交互模式



> 请求：客户端向服务端传递数据的主要方式之一，客户端主动向服务端发送请求，交给服务端处理。请求只能是客户端向服务端。

> 响应：服务端向客户端传递数据的主要方式之一，在接收请求后开始对数据进行处理，将数据发送给客户端。响应只能是服务端向客户端

1.4 JAVAWEB的CS和BS模式

> CS模式： Client-Server模式/客户端-服务端模式，该模式特点如下。

1 程序分两部分，一部分是客户端需要安装的程序，一部分是要部署在服务器上的程序；

2 用户需要在硬件设备或者操作系统中，下载安装特定的客户端程序才可以使用；

3 程序运行的压力由客户端和服务端共同承担；

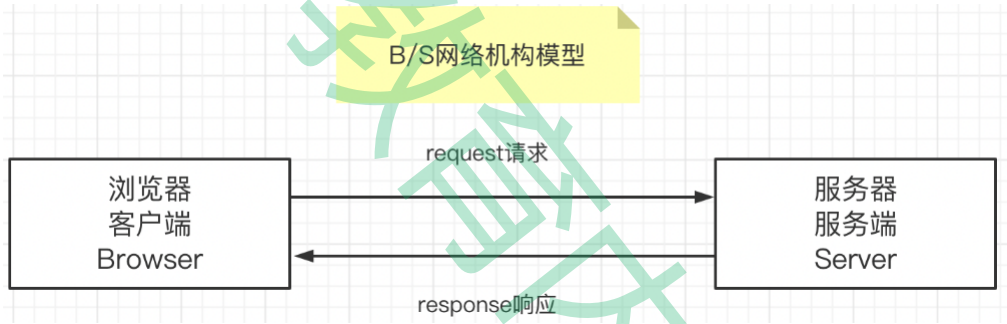
4 可以借助客户端运算资源，对数据继续处理，一般可以有较好的画质和展现效果；

5 程序更新时，往往需要同时更新客户端和服务端两部分；

6 跨平台性能一般，不同的平台未必都有对应的客户端程序；

7 开发成本较高，要开发服务端和客户端程序，要为不同的客户端开发不同客户端程序；

> BS模式 Browser-Server模式：



1 程序就一部分，只要部署在服务器上即可；

2 无论用户使用什么设备和操作系统，只要有一个安装任意一款浏览器即可；

3 程序运行的压力主要由服务端承担；

4 客户端承担的计算压力小，可以对数据进行简单的继续处理，但是不像CS模式那些可以获得较好的画质和展现效果；

5 程序更新时，只需要同时服务端部分；

6 跨平台性能优秀，只要有一个浏览器，到处都可以使用；

7 开发成本略低，不必为不同的客户端开发不同客户端程序；

> 模式选择：对于JAVA程序员来说，开发的是服务端代码，所有无论前端是何种类型的客户端，只要按照接口文档要求开发后端功能即可，前后端分离模式下，我们可以在几乎不接触前端的状态下完成开发。

1.5 JAVAWEB实现前后端分离

> 非前后端分离：

- 1 开发不分离： 程序员既要编写后端代码，又要去修改甚至编写前端代码，程序员的工作压力较大。
- 2 部署不分离：使用了后端动态页面技术(JSP,Thymeleaf等)，前端代码不能脱离后端服务器环境，必须部署在一起。

> 前后端分离：

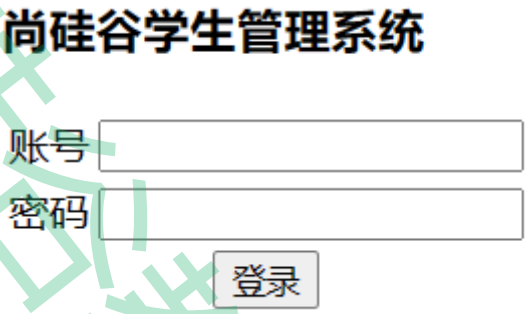
- 1 开发分离：后端程序员只要按照接口文档去编写后端代码，无需编写或者关系前端代码，前后端程序员压力都降低。
- 2 部署分离：前端使用单独的页面动态技术，通过VUE等框架，工程化项目，前端项目可以部署到独立的服务器上。

第二章 HTML&CSS

一 HTML入门

1.1 HTML&CSS&JavaScript的作用

> HTML 主要用于网页主体结构的搭建：



> CSS 主要用于页面元素美化：

> JavaScript 主要用于动态处理页面元素：

1.2 什么是HTML

HTML是Hyper Text Markup Language的缩写，意思是超文本标记语言。它的作用是搭建网页结构，在网页上展示内容。`HTML5` 在 2008 年正式发布，在 2012 年已形成了稳定的版本，2014年10月28日，W3C发布了HTML5的最终版。

1.3 什么是超文本

HTML文件本质上是文本文件，而普通的文本文件只能显示字符。通过标签把其他网页、图片、音频、视频等各种多媒体资源引入到当前网页中，让网页有了非常丰富的呈现方式，这就是超文本的含义： 本身是文本，但是呈现出来的最终效果超越了文本。

1.4 什么是标记语言

说HTML是一种 `标记语言` ，是因为它不是像Java这样的编程语言，因为它是由一系列 `标签` 组成的，没有常量、变量、流程控制、异常处理、IO等等这些功能。HTML很简单，每个标签都有它固定的含义和确定的页面显示效果。

- 双标签：标签是通过一组尖括号+标签名的方式来定义的

```
<p>HTML is a very popular fore-end technology.</p>
```

这个例子中使用了一个p标签来定义一个段落， 叫**开始标签**， 叫**结束标签**。开始标签和结束标签一起构成了一个完整的标签。开始 标签和结束标签之间的部分叫**文本标签体**，也简称：**标签体**。

- 单标签

```
<input type="text" name="username" />
```

- 属性

```
<a href="http://www.xxx.com">show detail</a>
```

href="**http://www.xxx.com**"就是属性，href是属性名，"**http://www.xxx.com**"是属性值。

1.5 HTML基础结构

1 文档声明：HTML文件中第一行的内容，用来告诉浏览器当前HTML文档的基本信息以及HTML文档遵循的语法标准。`<!DOCTYPE html>`。

2根标签：`<html></html>` 标签是整个文档的根标签，所有其他标签都必须放在这对标签里面。

3头部元素：`<head></head>` 标签是 `<html></html>` 第一个一级子标签,用于定义文档的头部，其他头部元素都放在head标签里。头部元素包括title标签、script标签、style标签、link标签、meta标签等等。

4主体元素：`<body></body>` 标签是 `<html></html>` 第二个一级子标签,用于标签定义网页的主体内容，在浏览器窗口内显示的内容都定义到body标签内。

5注释：`<!-- 注释内容 -->`。

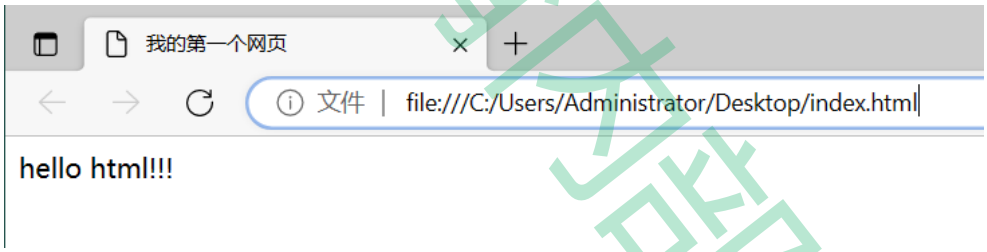
1.6 HTML的入门程序

第一步 准备一个纯文本文件，拓展名为html。

第二步 使用记事本打开网页，在网页内开发代码。

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8">
        <title>我的第一个网页</title>
    </head>
    <body>
        hello html!!!
    </body>
</html>
```

第三步 使用浏览器打开文件，查看显示的内容。



1.7 HTML概念词汇解释

标签：代码中的一个 <> 叫做一个标签，有些标签成对出现，称之为双标签，有些标签单独出现，称之为单标签。

属性：在开始标签中，用于定义标签的一些特征。

文本：双标签中间的文字，包含空格换行等结构。

元素：经过浏览器解析后，每一个完整的标签(标签+属性+文本)可以称之为一个元素。

1.8 HTML的语法规则

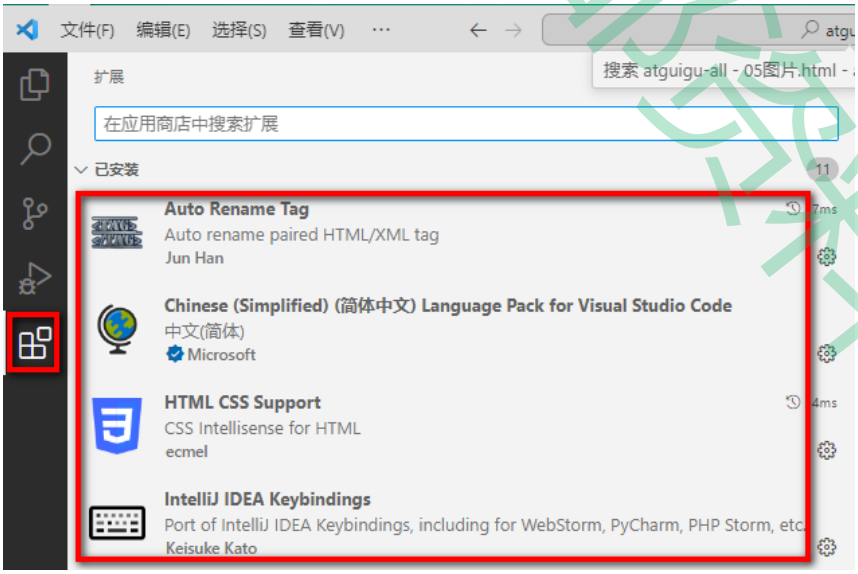
- 1 根标签是 `<html></html>` 有且只能有一个；
- 2 无论是双标签还是单标签都需要正确关闭；
- 3 标签可以嵌套但不能交叉嵌套；
- 4 注释语法为 ，注意不能嵌套；
- 5 属性必须有值，值必须加引号，H5中属性名和值相同时可以省略属性值；
- 6 HTML中不严格区分字符串使用单双引号；
- 7 HTML标签不严格区分大小写，但是不能大小写混用；
- 8 HTML中不允许自定义标签名，强行自定义则无效；

1.9 开发工具VsCode的安装和使用

- 前端工程师“Front-End-Developer”源自于美国。大约从2005年开始正式的前端工程师角色被行业所认可，到了2010年，互联网开始全面进入移动时代，前端开发的工作越来越重要。
- 前端工程师比较推崇的一款开发工具就是visual studio code，下载地址为: https://code.visualstudio.com/

> 1 安装过程：安装过程比较简单，一路next，注意安装路径不要有中文、空格和特殊符号即可。

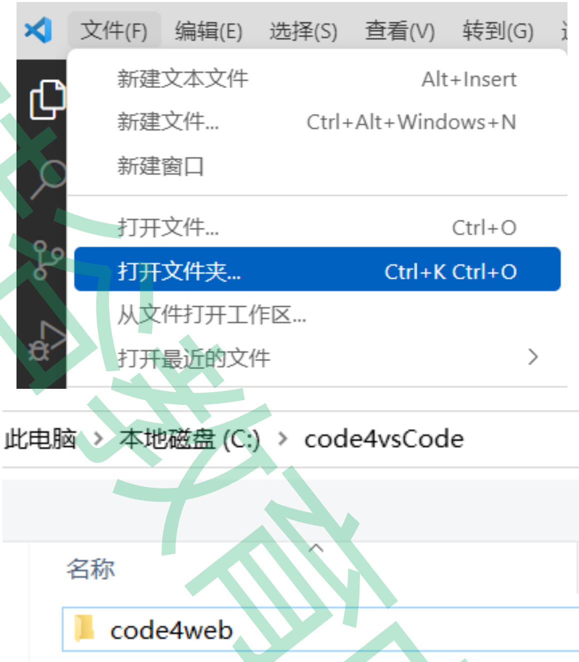
> 2 安装插件：



- Auto Rename Tag 自动修改标签对插件
- Chinese Language Pack 汉化包
- HTML CSS Support HTML CSS 支持

- Intellij IDEA Keybindings IDEA快捷键支持
- Live Server 实时加载功能的小型服务器
- open in browser 通过浏览器打开当前文件的插件
- Prettier-Code formatter 代码美化格式化插件
- Vetur   VScode中的Vue工具插件
- vscode-icons 文件显示图标插件
- Vue3 snipptes 生成VUE模板插件
- Vue language Features Vue3语言特征插件

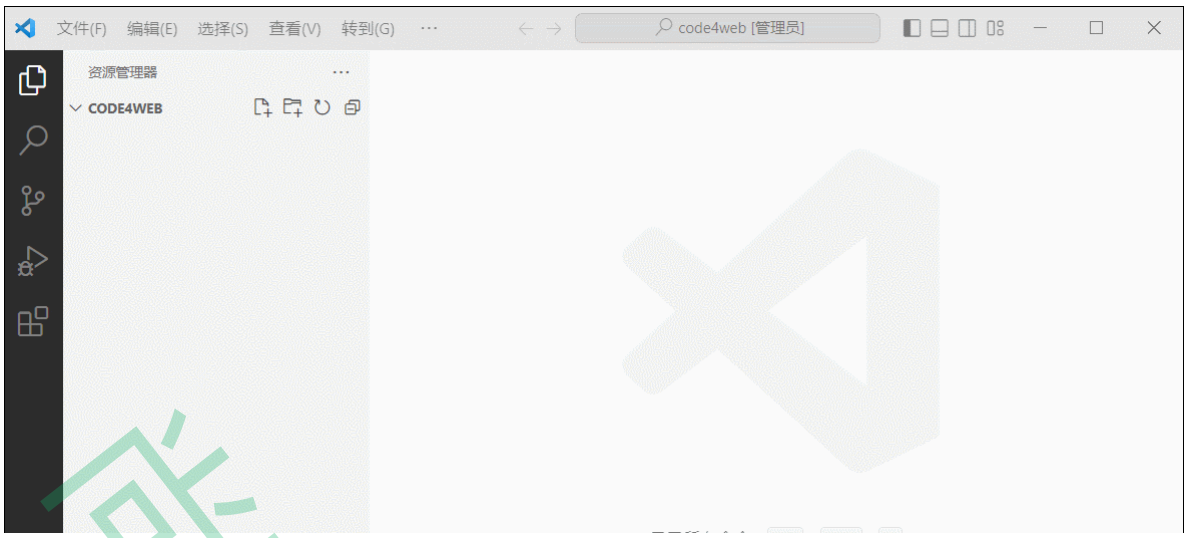
> 3 准备工作空间，提前在合适位置创建一个空目录，直接用vscode打开，即可直接将该作为项目代码存放的根目录，既工作空间：



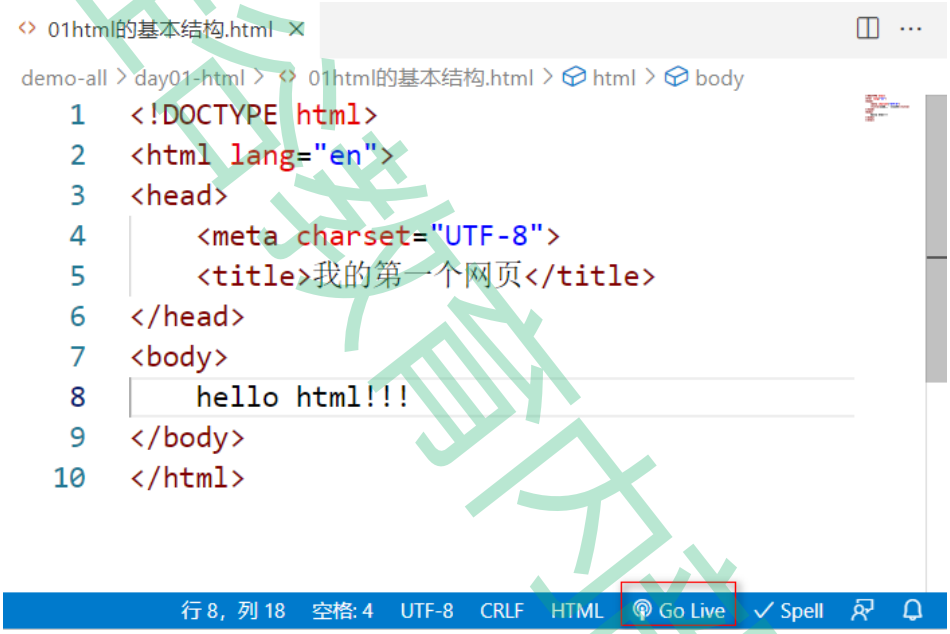
> 4 在工作空间下创建目录和文件：

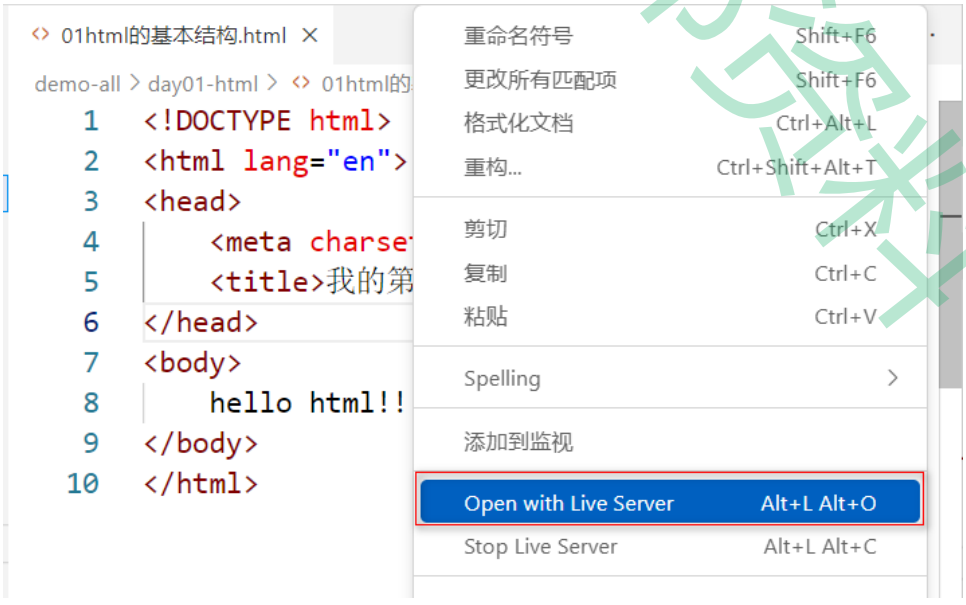
- 点击带有"+"号的按钮即可创建文件或者目录

- 在html中，输入"！"并回车即可快速出现html的基本结构



5 点击右下角Go Live，或者在html编辑视图上右击 open with live Server ，会自动启动小型服务器，并自动打开浏览器访问当前资源。





- Live Server支持实时加载更新，但是使用完毕后，要记得关闭，点击右下角 “Port:5500” 即可关闭

> 6 其他常见设置：

- 设置字体：齿轮>search>搜索 "字体大小"；
- 设置字体大小可以用滚轮控制：齿轮>设置>搜索 "Mouse Wheel Zoom"；
- 设置左侧树缩进：齿轮>设置>搜索 "树缩进"；
- 设置文件夹折叠：齿轮>设置>搜索 "compact" 取消第一个勾选；
- 设置编码自动保存：齿轮> 设置> 搜索 "Auto Save" ，选择为"afterDelay"；

二 HTML常见标签

> w3school在线帮助文档： http://www.w3school.com.cn 。

2.1 标题标签

> 标题标签一般用于在页面上定义一些标题性的内容，如新闻标题、文章标题等，有h1到h6六级标题：

```
<body>
    <h1>一级标题</h1>
    <h2>二级标题</h2>
    <h3>三级标题</h3>
    <h4>四级标题</h4>
    <h5>五级标题</h5>
    <h6>六级标题</h6>
</body>
```

2.2 段落标签

> 段落标签一般用于定义一些在页面上要显示的大段文字，多个段落标签之间实现自动分段的效果：

```
<body>
    <p>记者从工信部了解到，近年来我国算力产业规模快速增长，年增长率近30%，算力规模排名全球第二。</p>
    <p>工信部统计显示，截至去年底，我国算力总规模达到180百亿亿次浮点运算/秒，存力总规模超过1000EB（1万亿GB）。</p>
    <p>近年来，我国算力基础设施发展成效显著，梯次优化的算力供给体系初步构建，算力基础设施的综合能力显著提升。</p>
</body>
```

2.3 换行标签

> 单纯实现换行的标签是br，如果想添加分隔线，可以使用hr标签：

```
<body>
        截至去年底，我国算力总规模达到180百亿亿次浮点运算/秒，存力总规模超过1000EB（1万亿GB）。
<br>
        国家枢纽节点间的网络单向时延降低到20毫秒以内，算力核心产业规模达到1.8万亿元。<hr>
        中国信息通信研究院测算，算力每投入1元，将带动3至4元的GDP经济增长。
</body>
```

2.4 列表标签

> 有序列表，分条列项展示数据的标签，其每一项前面的符号带有顺序特征：

```
<ol>
    <li>JAVA</li>
    <li>前端</li>
    <li>大数据</li>
</ol>
```

> 无序列表，分条列项展示数据的标签，其每一项前面的符号不带有顺序特征：

```
<ul>
    <li>JAVASE</li>
    <li>JAVAEE</li>
    <li>数据库</li>
</ul>
```

> 嵌套列表，列表和列表之前可以嵌套，实现某一项内容详细展示：

```
<ol>
    <li>
        JAVA
        <ul>
            <li>JAVASE</li>
            <li>JAVAEE</li>
            <li>数据库</li>
        </ul>
    </li>
    <li>前端</li>
    <li>大数据</li>
</ol>
```

2.5 超链接标签

> 点击后跳转链接标签，也叫作a标签：

- href属性用于定义链接地址：
    - href中可以使用绝对路径，以/开头，始终以一个固定路径作为基准路径作为出发点；

    - href中也可以使用相对路径，不以/开头，以当前文件所在路径为出发点，./开头表示当前路径 ../表示上一层路径；
    - href中也可以使用完整的URL；
- target用于定义链接打开的方式：
    - _blank 在新窗口中打开目标资源；
    - _self 在当前窗口中打开目标资源；

```
<body>
   <a href="01html的基本结构.html" target="_blank">相对路径本地资源连接</a> <br>
   <a href="/day01-html/01html的基本结构.html" target="_self">绝对路径本地资源连接</a> <br>
   <a href="http://www.atguigu.com" target="_blank">外部资源链接</a> <br>
</body>
```

2.6 多媒体标签

> img(重点) 图片标签，用于在页面上引入图片：

- src属性用于定义图片的连接。
- title属性用于定义鼠标悬停时显示的文字。
- alt属性用于定义图片加载失败时显示的提示文字。

```
<img src="img/logo.png"  title="尚硅谷" alt="尚硅谷logo" />
```

> audio 用于在页面上引入一段声音，video 用于在页面上引入一段视频：

- src属性用于定义目标声音资源 。
- autoplay属性用于控制打开页面时是否自动播放 。
- controls属性用于控制是否展示控制面板 。
- loop属性用于控制是否进行循环播放 。

```
<audio src="img/music.mp3" autoplay="autoplay" controls="controls" loop="loop" />
<video src="img/movie.mp4" autoplay="autoplay" controls="controls" loop="loop"
width="400px" />
```

2.7 表格标签(重点)

> 常规表格：

- table标签代表表格；
- thead标签代表表头，可以省略不写；
- tbody标签代表表体，可以省略不写，浏览器解析DOM时会自动添加；
- tfoot标签 代表表尾，可以省略不写；
- tr标签代表一行；
- td标签代表行内的一格；
- th标签自带加粗和居中效果的td；

```
    <h3 style="text-align: center;">员工技能竞赛评分表</h3>
    <table  border="1px" style="width: 400px; margin: 0px auto;">
        <tr>
```

```
            <th>排名</th>     <th>姓名</th>       <th>分数</th>
        </tr>
        <tr>
            <td>1</td>       <td>张小明</td>     <td>100</td>
        </tr>
        <tr>
            <td>2</td>       <td>李小东</td>     <td>99</td>
        </tr>
        <tr>
            <td>3</td>       <td>王小虎</td>     <td>98</td>
        </tr>
    </table>
```

员工技能竞赛评分表

排名	姓名	分数
1	张小明	100
2	李小东	99
3	王小虎	98

> 单元格跨行：

- 通过td的rowspan属性实现上下跨行

```
    <h3 style="text-align: center;">员工技能竞赛评分表</h3>
    <table  border="1px" style="width: 400px; margin: 0px auto;">
        <tr>
            <th>排名</th>    <th>姓名</th>    <th>分数</th>     <th>备注</th>
        </tr>
        <tr>
            <td>1</td>      <td>张小明</td>  <td>100</td>     <td rowspan="3">前三名升职加
薪</td>
        </tr>
        <tr>
            <td>2</td>      <td>李小东</td>  <td>99</td>
        </tr>
        <tr>
            <td>3</td>      <td>王小虎</td>  <td>98</td>
        </tr>
    </table>
```

员工技能竞赛评分表

排名	姓名	分数	备注
1	张小明	100	前三名升职加薪
2	李小东	99	
3	王小虎	98	

> 单元格跨列：

- 通过td的colspan属性实现左右的跨列

```
    <h3 style="text-align: center;">员工技能竞赛评分表</h3>
    <table  border="1px" style="width: 400px; margin: 0px auto;">
        <tr>
            <th>排名</th>        <th>姓名</th>              <th>分数</th>       <th>备注
</th>
        </tr>
        <tr>
            <td>1</td>          <td>张小明</td>             <td>100</td>        <td
rowspan="6">前三名升职加薪</td>
        </tr>
        <tr>
            <td>2</td>          <td>李小东</td>             <td>99</td>
        </tr>
        <tr>
            <td>3</td>          <td>王小虎</td>             <td>98</td>
        </tr>
        <tr>
            <td>总人数</td>      <td colspan="2">2000</td>
        </tr>
        <tr>
            <td>平均分</td>      <td colspan="2">90</td>
        </tr>
        <tr>
            <td>及格率</td>      <td colspan="2">80%</td>
        </tr>
    </table>
```

员工技能竞赛评分表

排名	姓名	分数	备注
1	张小明	100	前三名升职加薪
2	李小东	99	
3	王小虎	98	
总人数	2000		
平均分	90		
及格率	80%		

2.8 表单标签(重点)

> 表单标签，可以实现让用户在界面上输入各种信息并提交的一种标签，是向服务端发送数据主要的方式之一：

- form标签：表单标签，其内部用于定义可以让用户输入信息的表单项标签。
  - action属性：用于定义信息提交的服务器的地址。
  - method属性：用于定义信息的提交方式。
    - get值： get方式提交，数据会缀到url后，以?作为参数开始的标识，多个参数用&隔开。

        - post值： post方式提交，数据会通过请求体发送，不会在缀到url后。
- input标签：主要的表单项标签，可以用于定义表单项。
    - name属性：用于定义提交的参数名。
    - type属性：用于定义表单项类型。
        - text 文本框
        - password 密码框
        - submit 提交按钮
        - reset 重置按钮

```
<form action="http://www.atguigu.com" method="get">
    用户名 <input type="text" name="username" /> <br>
    密　码 <input type="password" name="password" /> <br>
    <input type="submit"  value="登录" />
    <input type="reset"  value="重置" />
</form>
```

2.9 常见表单项标签(重点)

单行文本框：

```
个性签名：<input type="text" name="signal"/><br/>
```

个性签名：

密码框：

```
密码：<input type="password" name="secret"/><br/>
```

密码：

单选框：

```
你的性别是：
<input type="radio" name="sex" value="boy" />男
<input type="radio" name="sex" value="girl" checked="checked" />女
```

你的性别是： 男 女

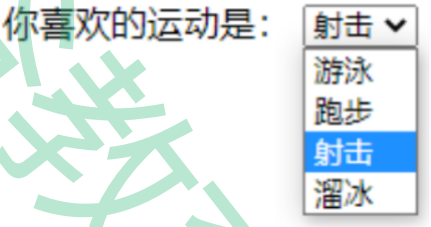
- name属性相同的radio为一组，组内互斥；
- 当用户选择了一个radio并提交表单，这个radio的name属性和value属性组成一个键值对发送给服务器；
- 设置checked="checked"属性设置默认被选中的radio，如果属性名和属性值一样的话，可以省略属性值，只写checked即可；

复选框：

```
你喜欢的球队是：
<input type="checkbox" name="team" value="Brazil"/>巴西
<input type="checkbox" name="team" value="German" checked/>德国
<input type="checkbox" name="team" value="France"/>法国
<input type="checkbox" name="team" value="China" checked="checked"/>中国
<input type="checkbox" name="team" value="Italian"/>意大利
```

下拉框：

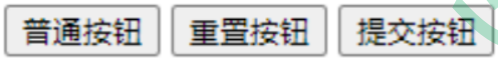
```
你喜欢的运动是：
<select name="sport">
    <option value="swimming">游泳</option>
    <option value="running">跑步</option>
    <option value="shooting" selected="selected">射击</option>
    <option value="skating">溜冰</option>
</select>
```



- 下拉列表用到了两种标签，其中select标签用来定义下拉列表，而option标签设置列表项；
- name属性在select标签中设置，value属性在option标签中设置；
- option标签的标签体是显示出来给用户看的，提交到服务器的是value属性的值；
- 通过在option标签中设置selected="selected"属性实现默认选中的效果；

按钮：

```
<button type="button">普通按钮</button>   或   <input type="button" value="普通按钮"/>
<button type="reset">重置按钮</button>    或   <input type="reset" value="重置按钮"/>
<button type="submit">提交按钮</button>   或   <input type="submit" value="提交按钮"/>
```



- 普通按钮：点击后无效果，需要通过JavaScript绑定单击响应函数；
- 重置按钮：点击后将表单内的所有表单项都恢复为默认值；
- 提交按钮： 点击后提交表单；

隐藏域：

```
<input type="hidden" name="userId" value="2233"/>
```

- 通过表单隐藏域设置的表单项不会显示到页面上，用户看不到，但是提交表单时会一起被提交。用来设置一些需要和表单一起提交但是不希望用户看到的数据，例如：用户id等等。

多行文本框：

```
自我介绍：<textarea name="desc"></textarea>
```

自我介绍：

- textarea没有value属性，如果要设置默认值需要写在开始和结束标签之间。

> 文件标签：

```
头像:<input type="file" name="file"/>
```

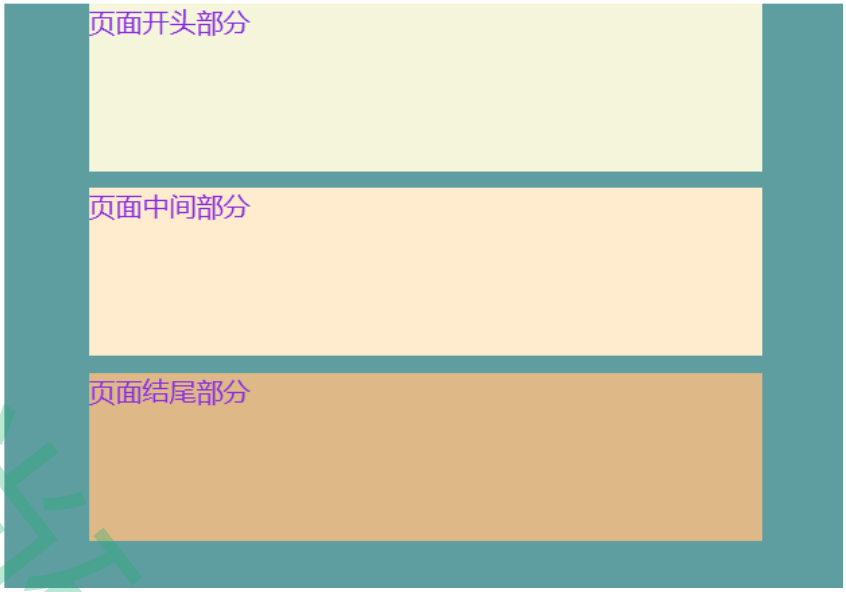
头像: 选择文件 未选择文件

2.10 布局相关标签

> div标签： 俗称"块"，主要用于划分页面结构，做页面布局。

> span标签： 俗称"层"，可以用于划分元素范围，配合CSS做页面元素样式的修饰。

```
    <div style="width: 500px; height: 400px;background-color: cadetblue;">
        <div style="width: 400px; height: 100px;background-color: beige;margin: 10px
auto;">
            <span style="color: blueviolet;">页面开头部分</span>
        </div>
        <div style="width: 400px; height: 100px;background-color:
blanchedalmond;margin: 10px auto;">
            <span style="color: blueviolet;">页面中间部分</span>
        </div>
        <div style="width: 400px; height: 100px;background-color: burlywood;margin:
10px auto;">
            <span style="color: blueviolet;">页面结尾部分</span>
        </div>
    </div>
```



2.11 特殊字符

> 对于有特殊含义的字符，需要通过转义字符来表示，详情见 https://www.w3school.com.cn/charsets/ref_html_8859.asp 。

三 CSS的使用

> CSS 层叠样式表(英文全称：(Cascading Style Sheets) ，能够对网页中元素位置的排版进行像素级精确控制，支持几乎所有的字体字号样式，拥有对网页对象和模型样式编辑的能力 ，简单来说，美化页面。

3.1 CSS引入方式

> 1、行内式：通过元素开始标签的style属性引入，语法为 style="样式名:样式值; 样式名:样式值;" ：

```
<input
    type="button"
    value="按钮"
    style="
        display: block;      width: 60px;
        height: 40px;        background-color: rgb(140, 235, 100);
        color: white;        border: 3px solid green;
        font-size: 22px;     font-family: '隶书';
        line-height: 30px;   border-radius: 5px;
"/>
```

- 缺点
  - html代码和css样式代码交织在一起，增加阅读难度和维护成本；
  - css样式代码仅对当前元素有效，代码重复量高，复用度低；

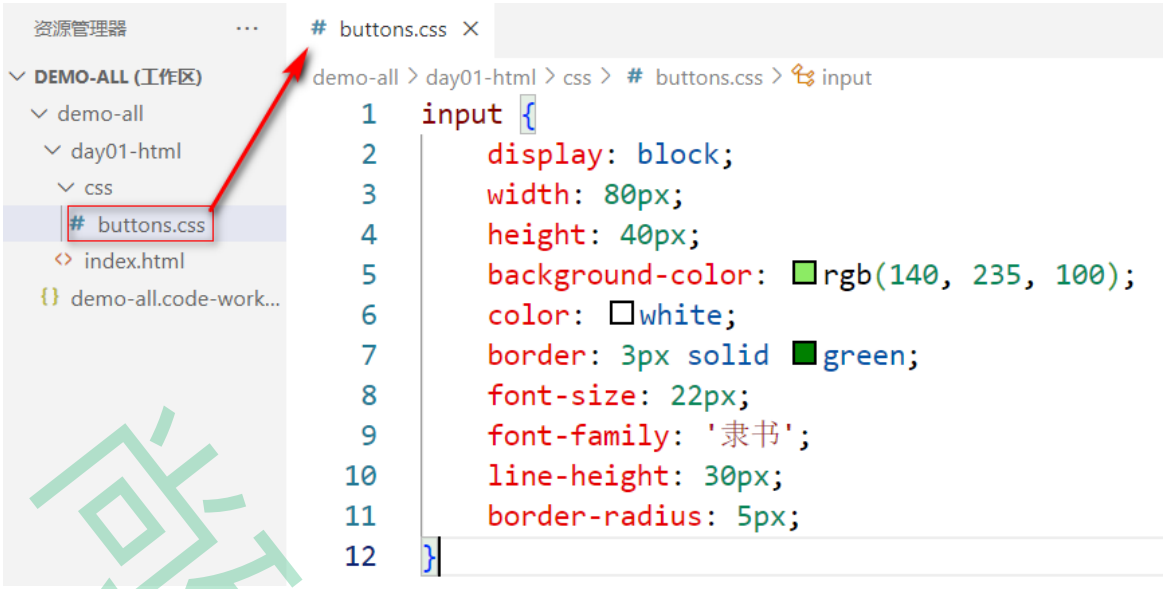
> 2、内嵌式：在head标签中通过style标签引入，这里要使用选择器确定样式的做作用位置：

```
<head>
    <meta charset="UTF-8">
    <style>
        /*选择器*/
        input {
            display: block;      width: 80px;
            height: 40px;        background-color: rgb(140, 235, 100);
            color: white;        border: 3px solid green;
            font-size: 22px;     font-family: '隶书';
            line-height: 30px;   border-radius: 5px;
        }
    </style>
</head>
<body>
    <input type="button" value="按钮1"/>
    <input type="button" value="按钮2"/>
    <input type="button" value="按钮3"/>
    <input type="button" value="按钮4"/>
</body>
```

- 内嵌式样式需要在head标签中，通过一对style标签定义CSS样式；
- CSS样式的作用范围控制要依赖选择器；
- CSS的样式代码中注释的方式为 /* */；
- 缺点：
  - 内嵌式虽然对样式代码做了抽取，但是CSS代码仍然在html文件中；
  - 内嵌样式仅仅能作用于当前文件，代码复用度还是不够，不利于网站风格统一；

> 3、连接式/外部样式表：将CSS代码单独放入css样式文件，在head标签中通过link标签引入外部样式表：

- 可以在项目单独创建css样式文件，专门用于存放CSS样式代码。



- 在head标签中，通过link标签引入外部CSS样式即可。

```
<head>
    <meta charset="UTF-8">
    <link href="css/buttons.css" rel="stylesheet" type="text/css"/>
</head>
<body>
    <input type="button" value="按钮1" />
    <input type="button" value="按钮2" />
</body>
```

- CSS样式代码从html文件中剥离，利于代码的维护。
- CSS样式文件可以被多个不同的html引入，利于网站风格统一。

3.2 CSS选择器

> 1、元素选择器：

```
<head>
    <meta charset="UTF-8">
   <style>
    input {
        display: block;     width: 80px;
        height: 40px;       background-color: rgb(140, 235, 100);
        color: white;       border: 3px solid green;
        font-size: 22px;    font-family: '隶书';
        line-height: 30px; border-radius: 5px;
    }
   </style>
</head>
<body>
    <input type="button" value="按钮1"/>
    <button>按钮5</button>
</body>
```

- 根据标签名确定样式的作用范围；

- 语法为： 元素名 {};
- 样式只能作用到同名标签上，其他标签不可用；
- 相同的标签未必需要相同的样式，会造成样式的作用范围太大；

2、id选择器：

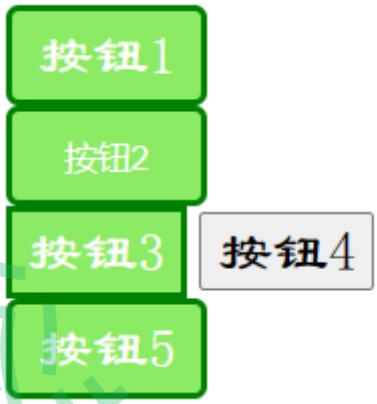
```
<head>
    <meta charset="UTF-8">
   <style>
    #btn1 {
        display: block;     width: 80px;
        height: 40px;       background-color: rgb(140, 235, 100);
        color: white;       border: 3px solid green;
        font-size: 22px;    font-family: '隶书';
        line-height: 30px;  border-radius: 5px;
    }
   </style>
</head>
<body>
    <input id="btn1" type="button" value="按钮1"/>
    <input id="btn2" type="button" value="按钮2"/>
    <input id="btn3" type="button" value="按钮3"/>
    <input id="btn4" type="button" value="按钮4"/>
    <button id="btn5">按钮5</button>
</body>
```

- 根据元素id属性的值确定样式的作用范围；
- 语法为： #id值 {};
- id属性的值在页面上具有唯一性，所有id选择器也只能影响一个元素的样式；
- 因为id属性值不够灵活，所以使用该选择器的情况较少；

3、class选择器：

```
<head>
    <meta charset="UTF-8">
   <style>
    .shapeClass {
        display: block;
        width: 80px;
        height: 40px;
        border-radius: 5px;
    }
    .colorClass{
        background-color: rgb(140, 235, 100);
        color: white;
        border: 3px solid green;
    }
    .fontClass {
        font-size: 22px;
        font-family: '隶书';
```

```
        line-height: 30px;
    }
   </style>
</head>
<body>
    <input  class ="shapeClass colorClass fontClass"type="button" value="按钮1"/>
    <input  class ="shapeClass colorClass" type="button" value="按钮2"/>
    <input  class ="colorClass fontClass" type="button" value="按钮3"/>
    <input  class ="fontClass" type="button" value="按钮4"/>
    <button class="shapeClass colorClass fontClass" >按钮5</button>
</body>
```



- 根据元素class属性的值确定样式的作用范围；
- 语法为： .class值 {}；
- class属性值可以有一个，也可以有多个，多个不同的标签也可以是使用相同的class值；
- 多个选择器的样式可以在同一个元素上进行叠加；
- 因为class选择器非常灵活，所以在CSS中，使用该选择器的情况较多；

3.4 CSS浮动

> CSS 的 Float（浮动）使元素脱离文档流，按照指定的方向（左或右发生移动），直到它的外边缘碰到包含框或另一个浮动框的边框为止。

- 浮动设计的初衷为了解决文字环绕图片问题，浮动后一定不会将文字挡住，这是设计初衷。
- 文档流是是文档中可显示对象在排列时所占用的位置/空间，而脱离文档流就是在页面中不占位置了。

> 浮动的样式float常见值：

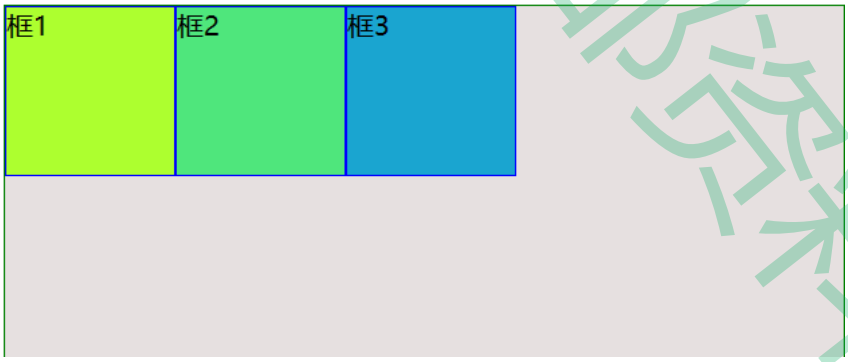
值	描述
left	元素向左浮动。
right	元素向右浮动。
none	默认值。元素不浮动，并会显示在其在文本中出现的位置。

> 通过代码感受浮动的效果：

```
<head>
```

```
    <meta charset="UTF-8">
    <style>
    .outerDiv {
        width: 500px;
        height: 300px;
        border: 1px solid green;
        background-color: rgb(230, 224, 224);
    }
    .innerDiv{
        width: 100px;
        height: 100px;
        border: 1px solid blue;
        float: left;
    }
    .d1{
        background-color: greenyellow;
       /*  float: right; */
    }
    .d2{
        background-color: rgb(79, 230, 124);
         /*  float: right; */
    }
    .d3{
        background-color: rgb(26, 165, 208);
         /*  float: right; */
    }
    </style>
</head>
<body>
    <div class="outerDiv">
        <div class="innerDiv d1">框1</div>
        <div class="innerDiv d2">框2</div>
        <div class="innerDiv d3">框3</div>
    </div>
</body>
```

- 效果



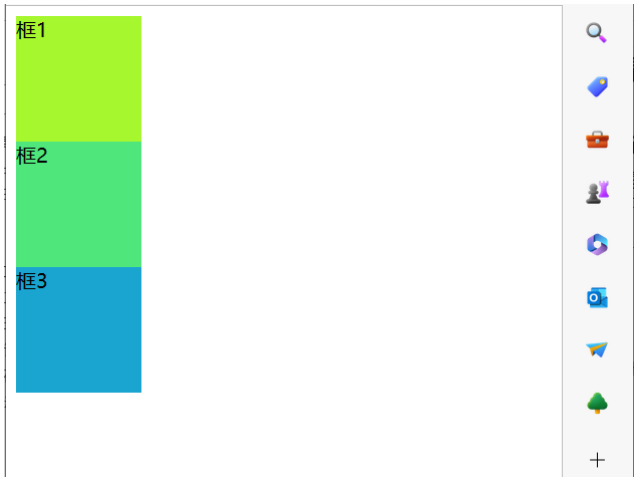
3.5 CSS定位

> position 属性指定了元素的定位类型：

值	描述
absolute	生成绝对定位的元素，相对于 static 定位以外的第一个父元素进行定位。 元素的位置通过 "left", "top", "right" 以及 "bottom" 属性进行规定。
fixed	生成绝对定位的元素，相对于浏览器窗口进行定位。 元素的位置通过 "left", "top", "right" 以及 "bottom" 属性进行规定。
relative	生成相对定位的元素，相对于其正常位置进行定位。 因此，"left:20" 会向元素的 LEFT 位置添加 20 像素。
static	默认值。没有定位，元素出现在正常的流中（忽略 top, bottom, left, right 或者 z-index 声明）。

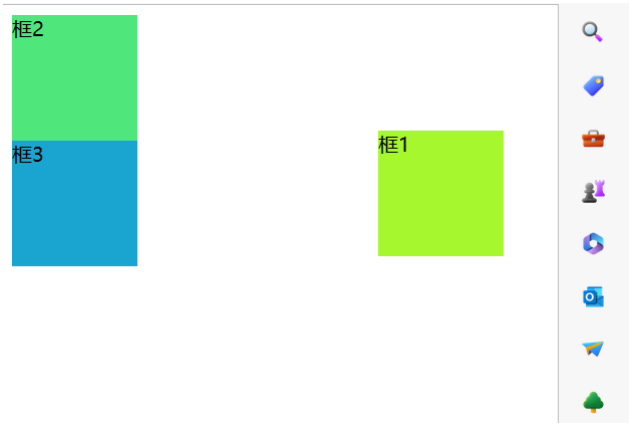
> 静态定位static，不设置的时候的默认值就是static，静态定位，元素出现在该出现的位置，块级元素垂直排列，行内元素水平排列：

```
<head>
    <meta charset="UTF-8">
    <style>
        .innerDiv{
                width: 100px;
                height: 100px;
        }
        .d1{
            background-color: rgb(166, 247, 46);
            position: static;
        }
        .d2{
            background-color: rgb(79, 230, 124);
        }
        .d3{
            background-color: rgb(26, 165, 208);
        }
    </style>
</head>
<body>
        <div class="innerDiv d1">框1</div>
        <div class="innerDiv d2">框2</div>
        <div class="innerDiv d3">框3</div>
</body>
```



> 绝对定位 absolute，通过 top left right bottom 指定元素在页面上的固定位置，定位后元素会让出原来位置，其他元素可以占用：

```
<head>
    <meta charset="UTF-8">
    <style>
        .innerDiv{
                width: 100px;
                height: 100px;
        }
        .d1{
            background-color: rgb(166, 247, 46);
            position: absolute;
            left: 300px;
            top: 100px;
        }
        .d2{
            background-color: rgb(79, 230, 124);
        }
        .d3{
            background-color: rgb(26, 165, 208);
        }
    </style>
</head>
<body>
        <div class="innerDiv d1">框1</div>
        <div class="innerDiv d2">框2</div>
        <div class="innerDiv d3">框3</div>
</body>
```



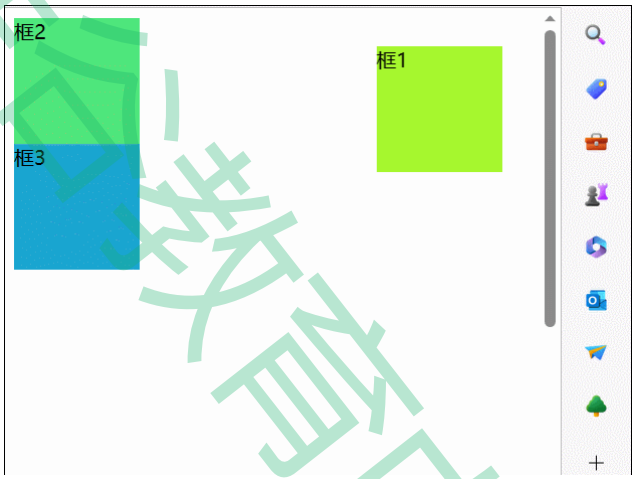
相对定位relative， 相对于自己原来的位置进行地位，定位后保留原来的站位，其他元素不会移动到该位置：

```
<head>
    <meta charset="UTF-8">
    <style>
        .innerDiv{
                width: 100px;
                height: 100px;
        }
        .d1{
            background-color: rgb(166, 247, 46);
            position: relative;
            left: 30px;
            top: 30px;
        }
        .d2{
            background-color: rgb(79, 230, 124);
        }
        .d3{
            background-color: rgb(26, 165, 208);
        }
    </style>
</head>
<body>
        <div class="innerDiv d1">框1</div>
        <div class="innerDiv d2">框2</div>
        <div class="innerDiv d3">框3</div>
</body>
```

固定定位fixed，始终在浏览器窗口固定位置，不会随着页面的上下移动而移动，元素定位后会让出原来的位置,其他元素可以占用：

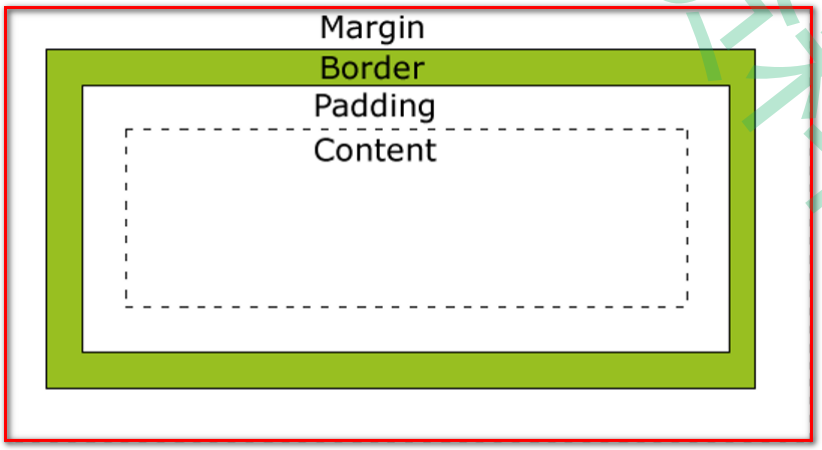
```
<head>
    <meta charset="UTF-8">
    <style>
        .innerDiv{
                width: 100px;
                height: 100px;
        }
        .d1{
            background-color: rgb(166, 247, 46);
```

```
            position: fixed;
            right: 30px;
            top: 30px;
        }
        .d2{
            background-color: rgb(79, 230, 124);
        }
        .d3{
            background-color: rgb(26, 165, 208);
        }
    </style>
</head>
<body>
    <div class="innerDiv d1">框1</div>
    <div class="innerDiv d2">框2</div>
    <div class="innerDiv d3">框3</div>
    br*100+tab
</body>
```



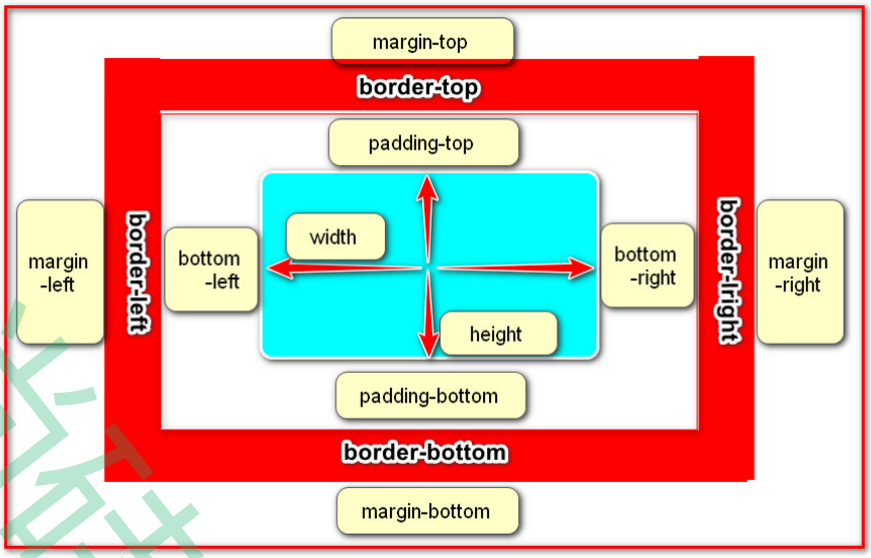
3.6 CSS盒子模型

> 所有HTML元素可以看作盒子，在CSS中，"box model"这一术语是用来设计和布局时使用。CSS盒模型本质上是一个盒子，封装周围的HTML元素，它包括：边距（margin），边框（border），填充（padding），和实际内容（content）。



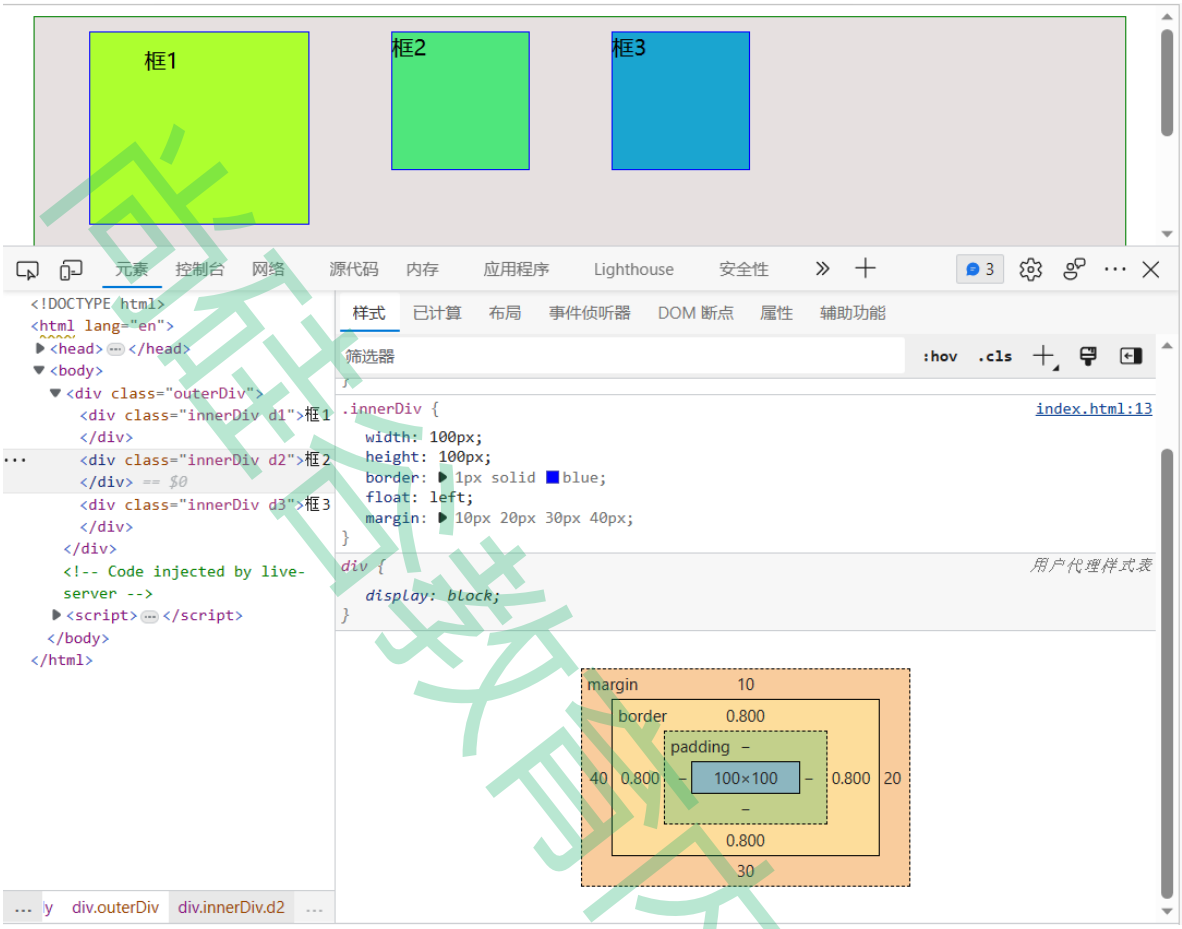
- Margin(外边距) - 清除边框外的区域，外边距是透明的；
- Border(边框) - 围绕在内边距和内容外的边框；

- Padding(内边距) - 清除内容周围的区域，内边距是透明的；
- Content(内容) - 盒子的内容，显示文本和图像；




```
<head>
   <meta charset="UTF-8">
   <style>
    .outerDiv {
        width: 800px;
        height: 300px;
        border: 1px solid green;
        background-color: rgb(230, 224, 224);
        margin: 0px auto;
    }
    .innerDiv{
        width: 100px;
        height: 100px;
        border: 1px solid blue;
        float: left;
        /* margin-top: 10px;
        margin-right: 20px;
        margin-bottom: 30px;
        margin-left: 40px; */
        margin: 10px 20px 30px 40px;

    }
    .d1{
        background-color: greenyellow;
        /* padding-top: 10px;
        padding-right: 20px;
        padding-bottom: 30px;
        padding-left: 40px; */
        padding: 10px 20px 30px 40px;
    }
    .d2{
        background-color: rgb(79, 230, 124);
    }
    .d3{
        background-color: rgb(26, 165, 208);
    }
   </style>
</head>
```

```
    <body>
        <div class="outerDiv">
            <div class="innerDiv d1">框1</div>
            <div class="innerDiv d2">框2</div>
            <div class="innerDiv d3">框3</div>
        </div>
    </body>
```



第三章 JavaScript

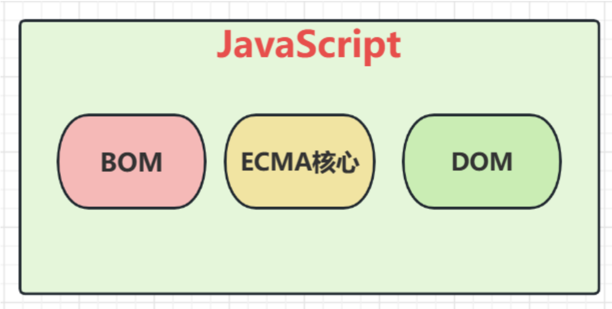
一 JS简介

1.1 JS起源

> Javascript是一种由Netscape(网景)的LiveScript发展而来的原型化继承的面向对象的动态类型的区分大小写的 客户端脚本语言，主要目的是为了解决服务器端语言，遗留的速度问题，于是Netscape的浏览器Navigator加入了Javascript，提供了数据验证的基本功能。ECMA-262 是正式的 JavaScript 标准。这个标准开发始于 1996 年，在 1997 年 7 月，ECMA 会员大会采纳了它的首个版本。这个标准由 ECMA 组织发展和维护。JavaScript 的正式名称是 "ECMAScript"。JavaScript的组成包含ECMAScript、DOM、BOM。 JS是一种运行于浏览器端上的小脚本语句,可以实现网页如文本内容动,数据动态变化和动画特效等 。JS有 如下特点：

- 脚本语言：JavaScript是一种解释型的脚本语言。不同于C、C++、Java等语言先编译后执行，JavaScript不会产生编译出来的字节码文件，而是在程序的运行过程中对源文件逐行进行解释；
- 基于对象：JavaScript是一种基于对象的脚本语言，它不仅可以创建对象，也能使用现有的对象。但是面向对象的三大特性：『封装』、『继承』、『多态』中，JavaScript能够实现封装，可以模拟继承，不支持多态，所以它不是一门面向对象的编程语言；
- 弱类型：JavaScript中也有明确的数据类型，但是声明一个变量后它可以接收任何类型的数据，并且会在程序执行过程中根据上下文自动转换类型；
- 事件驱动：JavaScript是一种采用事件驱动的脚本语言，它不需要经过Web服务器就可以对用户的输入做出响应；
- 跨平台性：JavaScript脚本语言不依赖于操作系统，仅需要浏览器的支持。因此一个JavaScript脚本在编写后可以带到任意机器上使用，前提是机器上的浏览器支持JavaScript脚本语言。目前JavaScript已被大多数的浏览器所支持；

1.2 JS 组成部分



> ECMA 及版本变化：

- 是一种由欧洲计算机制造商协会（ECMA）通过ECMA-262标准化的脚本程序语言,ECMAScript描述了语法、类型、语句、关键字、保留字、运算符和对象。它就是定义了脚本语言的所有属性、方法和对象；
- ECMA-262第1版本质上跟网景的JavaScript 1.1相同；
- ECMA-262第2版只是做了一些编校工作，并没有增减或改变任何特性；
- ECMA-262第3版第一次真正对ECMAScript进行更新；
- ECMA-262第4版是对这门语言的一次彻底修订；
- ECMA-262第5版是ECMA-262第3版的小幅改进，于2009年12月3日正式发布；
- ECMA-262第6版-ES6、于2015年6月发布。 这一版包含了大概这个规范有史以来最重要的一批增强特性。 但是并不是所有的浏览器都全面支持；
- ECMA-262第7版也称为ES7或ES2016，于2016年6月发布；
- ECMA-262第8版也称为ES8、ES2017，完成于2017年6月；
- ECMA-262第9版也称为ES9、ES2018，发布于2018年6月；
- ECMA-262第10版也称为ES10、ES2019，发布于2019年6月；
- ECMA-262第11版，也成为ES11、ES2020，发布于2020年6月；
- ... ...

> BOM编程：

- BOM是Browser Object Model的简写，即浏览器对象模型；
- BOM有一系列对象组成，是访问、控制、修改浏览器的属性和方法；
- BOM没有统一的标准(每种客户端都可以自定标准)；
- BOM编程是将浏览器窗口的各个组成部分抽象成各个对象，通过各个对象的API操作组件行为的一种编程；

> DOM编程：

- 简单来说，DOM编程就是使用document对象的API完成对网页HTML文档进行动态修改，以实现网页数据和样式动态变化效果的编程；
- document对象代表整个html文档，可用来访问页面中的所有元素，是最复杂的一个dom对象，可以说是学习好dom编程的关键所在；
- DOM编程其实就是用window对象的document属性的相关API完成对页面元素的控制的编程；

1.3 JS的引入方式

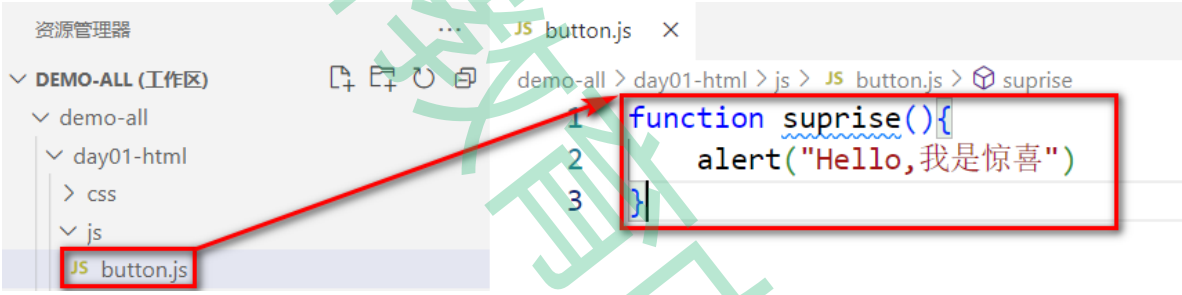
> 1、内部脚本方式引入：

- 在页面中，通过一对script标签引入JS代码；
- script代码放置位置有一定的随意性，一般放在head标签中；

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8">
        <title>小标题</title>
        <script>
            function suprise(){
                alert("Hello,我是惊喜")
            }
        </script>
    </head>
    <body>
        <button  onclick="suprise()">点我有惊喜</button>
    </body>
</html>
```

2、外部脚本方式引入：

- 内部脚本仅能在当前页面上使用，代码复用度不高；
- 可以将脚本放在独立的js文件中，通过script标签引入外部脚本文件；
- 一对script标签要么用于定义内部脚本，要么用于引入外部js文件，不能混用；
- 一个html文档中，可以有多个script标签；




```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8">
        <title>小标题</title>
        <script src="js/button.js" type="text/javascript"></script>
    </head>
    <body>
        <button  onclick="suprise()">点我有惊喜</button>
    </body>
</html>
```

二 JS的数据类型和运算符

2.1 JS的数据类型

数值类型：数值类型统一为 number，不区分整数和浮点数。

字符串类型：字符串类型为 string 和JAVA中的String相似，JS中不严格区分单双引号，都可以用于表示字符串。

布尔类型：布尔类型为boolean 和Java中的boolean相似，但是在JS的if语句中，非空字符串会被转换为'真'，非零数字也会被认为是'真'。

引用数据类型：引用数据类型对象是Object类型，各种对象和数组在JS中都是Object类型。

function类型：JS中的各种函数属于function数据类型。

命名未赋值：js为弱类型语言，统一使用 var 声明对象和变量，在赋值时才确定真正的数据类型，变量如果只声明没有赋值的话,数据类型为undefined。

赋予NULL值：在JS中，如果给一个变量赋值为null，其数据类型是Object，可以通过typeof关键字判断数据类型。

2.2 JS的变量

JS中的变量具有如下特征：

1、弱类型变量，可以统一声明成var；

2、var声明的变量可以再次声明；

3、变量可以使用不同的数据类型多次赋值；

4、JS的语句可以以; 结尾，也可以不用;结尾；

5、变量标识符严格区分大小写；

6、标识符的命名规则参照JAVA；

7、如果使用了 一个没有声明的变量，那么运行时会报：uncaught ReferenceError: *** is not defined at index.html:行号:列号；

8、如果一个变量只声明，没赋值，那么值是undefined；

2.3 JS的运算符

算数运算符： + - * / %

- 其中需要注意的是 / 和 %
  - / 在除0时，结果是Infinity，而不是报错；
  - %在模0时，结果是NaN，意思为 not a number，而不是报错；

复合算数运算符： ++ -- += -= *= /= %=

- 符合算数运算符基本和JAVA一致,同样需要注意 /=和%=
  - 在/=0时，结果是Infinity，而不是报错；
  - 在%=0时，结果是NaN，意思为 not a number，而不是报错；

> 关系运算符： > < >= <= == === !=

- 需要注意的是 == 和 === 差别
  - == 符号，如果两端的数据类型不一致，会尝试将两端的数据转换成number，再对比number大小。
    - '123' 这种字符串可以转换成数字；
    - true会被转换成1 ， false会被转换成0；
  - === 符号，如果两端数据类型不一致，直接返回false，数据类型一致再比较是否相同。

> 逻辑运算符： || &&

- 几乎和JAVA中的一样，需要注意的是，这里直接就是短路的逻辑运算符，单个的| 和& 以及 ^ 是位运算符。

> 条件运算符： 条件? 值1 : 值2

- 几乎和JAVA中的一样。

> 位运算符： | & ^ << >> >>>

- 和 java中的类似(了解)。

三 JS的流程控制和函数

3.1 JS分支结构

> if结构：

- 这里的if结构几乎和JAVA中的一样，需要注意的是：
  - if()中的非空字符串会被认为是true；
  - if()中的非零数字会被认为是true；

```
if('false'){// 非空字符串 if判断为true
    console.log(true)
}else{
    console.log(false)
}
if(''){// 长度为0字符串 if判断为false
    console.log(true)
}else{
    console.log(false)
}
if(1){// 非零数字 if判断为true
    console.log(true)
}else{
    console.log(false)
}
if(0){
    console.log(true)
```

```
}else{
    console.log(false)
}
```

switch结构：

```
var monthStr=prompt("请输入月份","例如:10 ");
var month= Number.parseInt(monthStr)
switch(month){
    case 3:
    case 4:
    case 5:
        console.log("春季");
        break;
    case 6:
    case 7:
    case 8:
        console.log("夏季");
        break;
    case 9:
    case 10:
    case 11:
        console.log("秋季");
        break;
    case 1:
    case 2:
    case 12:
        console.log("冬季");
        break;
    default :
        console.log("月份有误")
}
```

3.2 JS循环结构

while结构：

```
/* 打印99 乘法表 */
var i = 1;
while(i <= 9){
    var j = 1;
    while(j <= i){
        document.write(j+"*"+i+"="+i*j+"     ");
        j++;
    }
    document.write("<hr/>");
    i++;
}
```

for循环：

```
/* 打印99 乘法表 */
for(  var i = 1;i <= 9; i++){
    for(var j = 1;j <= i;j++){
        document.write(j+"*"+i+"="+i*j+"     ");
    }
    document.write("<hr/>");
}
```

> foreach循环：

- 迭代数组时，和java不一样。
  - 括号中的临时变量表示的是元素的索引，不是元素的值。
  - ()中也不在使用: 分隔，而是使用 in 关键字。

```
var cities =["北京","上海","深圳","武汉","西安","成都"]
document.write("<ul>")
for(var index in  cities){
    document.write("<li>"+cities[index]+"</li>")
}
document.write("</ul>")
```

3.3 JS函数声明

> JS中的方法，多称为函数，函数的声明语法和JAVA中有较大区别。

- 函数说明
  - 函数没有权限控制符。
  - 不用声明函数的返回值类型，需要返回在函数体中直接return即可，也无需void关键字。
  - 参数列表中，无需数据类型。
  - 调用函数时，实参和形参的个数可以不一致。
  - 声明函数时需要用function关键字。
  - J函数没有异常列表。
- 代码

```
/* 语法1 : function 函数名 (参数列表){函数体} */
function sum(a, b){
    return a+b;
}
var result =sum(10,20);
console.log(result)
/* 语法2 :  var 函数名 = function (参数列表){函数体}  */
var add = function(a, b){
    return a+b;
}
var result = add(1,2);
console.log(result);
```

四 JS的对象和JSON

4.1 JS声明对象的语法

> 语法1， 通过new Object()直接创建对象：

```
var person =new Object();
// 给对象添加属性并赋值
person.name="张小明";
person.age=10;
person.foods=["苹果","橘子","香蕉","葡萄"];
// 给对象添加功能函数
person.eat= function (){
    console.log(this.age+"岁的"+this.name+"喜欢吃:")
    for(var i = 0;i<this.foods.length;i++){
        console.log(this.foods[i])
    }
}
//获得对象属性值
console.log(person.name)
console.log(person.age)
//调用对象方法
person.eat();
```

> 语法2，通过 {}形式创建对象：

```
var person ={
    "name":"张小明",
    "age":10,
    "foods":["苹果","香蕉","橘子","葡萄"],
    "eat":function (){
        console.log(this.age+"岁的"+this.name+"喜欢吃:")
        for(var i = 0;i<this.foods.length;i++){
            console.log(this.foods[i])
        }
    }
}
//获得对象属性值
console.log(person.name)
console.log(person.age)
//调用对象方法
person.eat();
```

4.2 JSON格式

> JSON（JavaScript Object Notation, JS对象简谱）是一种轻量级的数据交换格式。它基于ECMAScript（European Computer Manufacturers Association, 欧洲计算机协会的一个子集，采用完全独立于编程语言的文本格式来存储和表示数据。简洁和清晰的层次结构使得 JSON 成为理想的数据交换语言。 易于人阅读和编写，同时也易于机器解析和生成，并有效地提升网络传输效率，简单来说，JSON 就是一种字符串格式，这种格式无论是在前端还是在后端，都可以很容易地和对象之间进行转换，所以常用于前后端数据传递 。

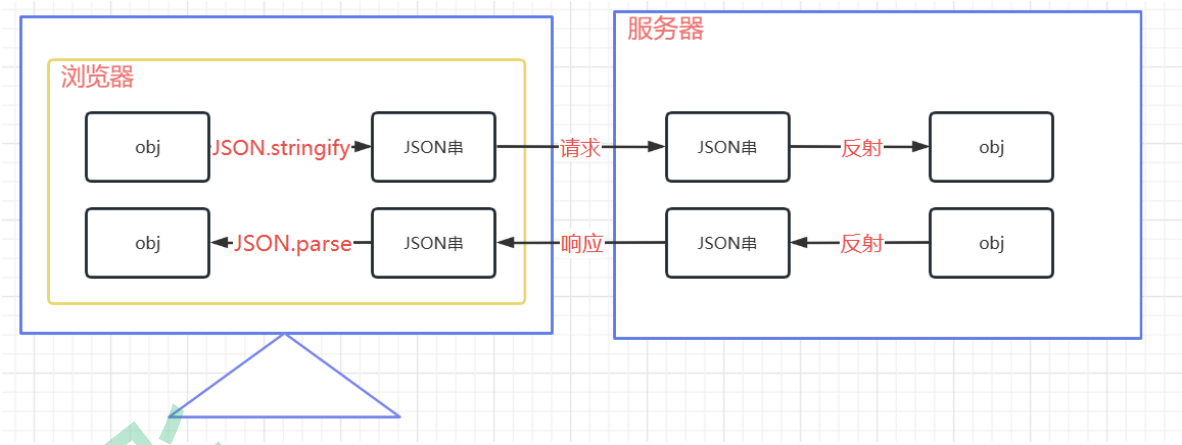
- JSON的语法： var str="{'属性名':'属性值','属性名':{'属性名':'属性值'},'属性名':['值1','值1','值3']}" ；

- JSON字符串一般用于传递数据，一般都是用对象的属性表示数据，所以在此不研究对象的函数，只看对象的属性；
- 通过JSON.parse()方法可以将一个JSON串转换成对象；
- 通过JSON.stringify()方法可以将一个对象转换成一个JSON格式的字符串；

```
/* 定义一个JSON串 */
var personStr ='{"name":"张小明","age":20,"girlFriend":{"name":"铁
铃","age":23},"foods":["苹果","香蕉","橘子","葡萄"],"pets":[{"petName":"大
黄","petType":"dog"},{"petName":"小花","petType":"cat"}]}'
console.log(personStr)
console.log(typeof personStr)
/* 将一个JSON串转换为对象 */
var person =JSON.parse(personStr);
console.log(person)
console.log(typeof person)
/* 获取对象属性值 */
console.log(person.name)
console.log(person.age)
console.log(person.girlFriend.name)
console.log(person.foods[1])
console.log(person.pets[1].petName)
console.log(person.pets[1].petType)
```

```
/* 定义一个对象 */
var person={
    'name':'张小明',
    'age':20,
    'girlFriend':{
        'name':'铁铃',
        'age':23
    },
    'foods':['苹果','香蕉','橘子','葡萄'],
    'pets':[
        {
            'petName':'大黄',
            'petType':'dog'
        },
        {
            'petName':'小花',
            'petType':'cat'
        }
    ]
}

/* 获取对象属性值 */
console.log(person.name)
console.log(person.age)
console.log(person.girlFriend.name)
console.log(person.foods[1])
console.log(person.pets[1].petName)
console.log(person.pets[1].petType)
/* 将对象转换成JSON字符串 */
var personStr =JSON.stringify(person)
console.log(personStr)
console.log(typeof personStr)
```

- 前后端传递数据



4.3 JS常见对象

4.3.1 数组

创建数组的四种方式：

- new Array()　　创建空数组
- new Array(5)　　创建数组时给定长度
- new Array(ele1,ele2,ele3,... ... ,elen);　　创建数组时指定元素值
- [ele1,ele2,ele3,... ... ,elen];　　相当于第三种语法的简写

数组的常见API见： https://www.runoob.com/jsref/jsref-obj-array.html

4.3.2 Boolean对象

boolean常见API见： https://www.runoob.com/jsref/jsref-obj-boolean.html

4.3.3 Date对象

Date常见API见： https://www.runoob.com/jsref/jsref-obj-date.html

4.3.4 Math

Math常见API见： https://www.runoob.com/jsref/jsref-obj-math.html

4.3.5 Number

Number常见API见： https://www.runoob.com/jsref/jsref-obj-number.html

4.3.6 String

String常见API见： https://www.runoob.com/jsref/jsref-obj-string.html

五 事件的绑定

5.1 什么是事件

> HTML 事件可以是浏览器行为，也可以是用户行为。 当这些一些行为发生时，可以自动触发对应的JS函数的运行。 JS的事件驱动指的就是行为触发代码运行的这种特点 。

5.2 常见事件

> 鼠标事件： https://www.runoob.com/jsref/dom-obj-event.html

> 键盘事件： https://www.runoob.com/jsref/dom-obj-event.html

> 表单事件： https://www.runoob.com/jsref/dom-obj-event.html

5.3 事件的绑定

> 通过属性绑定：

```
    <head>
        <meta charset="UTF-8">
        <title>小标题</title>
        <script>
            function testDown1(){console.log("down1")}
            function testDown2(){console.log("down2")}
            function testFocus(){console.log("获得焦点")}
            function testBlur(){ console.log("失去焦点") }
            function testChange(input){console.log("内容改
变");console.log(input.value);}
            function testMouseOver(){ console.log("鼠标悬停") }
            function testMouseLeave(){ console.log("鼠标离开")}
            function testMouseMove(){ console.log("鼠标移动")}
        </script>
    </head>

    <body>
        <input type="text"
        onkeydown="testDown1(),testDown2()"
        onfocus="testFocus()"
        onblur="testBlur()"
        onchange="testChange(this)"
        onmouseover="testMouseOver()"
        onmouseleave="testMouseLeave()"
        onmousemove="testMouseMove()"
         />
```

```
    </body>
```

- 通过事件属性绑定函数，在行为发生时会自动执行函数。
- 一个事件可以同时绑定多个函数。
- 一个元素可以同时绑定多个事件。
- 方法中可以传入 this对象，代表当前元素。

5.4 事件的触发

> 行为触发：

- 发生行为时触发，演示：略

> DOM编程触发：

- 通过DOM编程，用代码触发，执行某些代码相当于发生了某些行为。

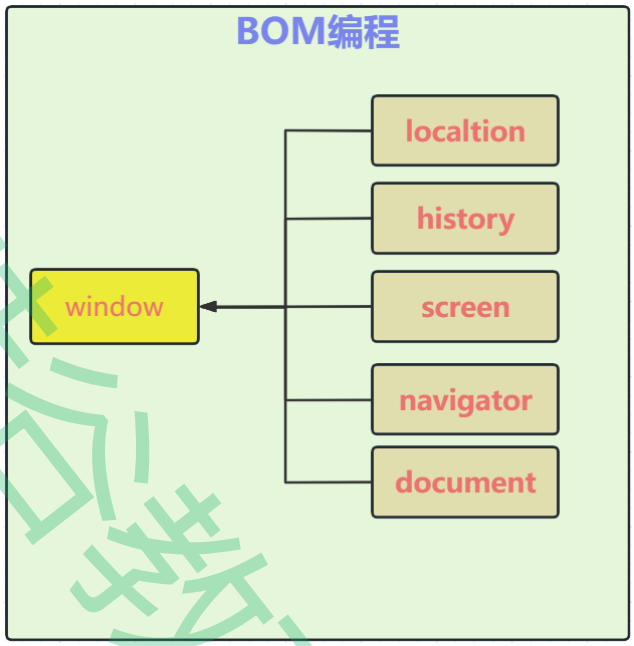
```
<head>
    <meta charset="UTF-8">
    <title>小标题</title>
    <script>
        // 页面加载完毕事件，浏览器加载完整个文档行为
        window.onload=function(){
            var in1 =document.getElementById("in1");
            // 通过DOM编程绑定事件
            in1.onchange=testChange
        }
        function testChange(){
            console.log("内容改变")
            console.log(event.target.value);
        }
    </script>
</head>
<body>
    <input id="in1" type="text" />
</body>
```

六 BOM编程

6.1 什么是BOM

- BOM是Browser Object Model的简写，即浏览器对象模型；
- BOM由一系列对象组成，是访问、控制、修改浏览器的属性和方法；
- BOM没有统一的标准(每种客户端都可以自定标准)；
- BOM编程是将浏览器窗口的各个组成部分抽象成各个对象，通过各个对象的API操作组件行为的一种编程；
- BOM编程的对象结构如下:
    - window 顶级对象，代表整个浏览器窗口；

- location属性，代表浏览器的地址栏；
- history属性，代表浏览器的访问历史；
- screen属性，代表屏幕；
- navigator属性，代表浏览器软件本身；
- document属性，代表浏览器窗口目前解析的html文档；
- console属性，代表浏览器开发者工具的控制台；
- localStorage属性，代表浏览器的本地数据持久化存储；
- sessionStorage属性，代表浏览器的本地数据会话级存储；



6.2 window对象的常见属性(了解)

- window对象常见属性见：https://www.runoob.com/jsref/obj-window.html

6.3 window对象的常见方法(了解)

- window对象常见方法见：https://www.runoob.com/jsref/obj-window.html

6.4 通过BOM编程控制浏览器行为演示

> 三种弹窗方式：

```
<head>
    <meta charset="UTF-8">
    <title>小标题</title>
    <script>
       function testAlert(){
            //普通信息提示框
            window.alert("提示信息");
       }
       function testConfirm(){
            //确认框
```

```
            var con =confirm("确定要删除吗?");
            if(con){
                alert("点击了确定")
            }else{
                alert("点击了取消")
            }
        }
        function testPrompt(){
            //信息输入对话框
            var res =prompt("请输入昵称","例如:张三");
            alert("您输入的是:"+res)
        }
    </script>
</head>

<body>
    <input type="button" value="提示框" onclick="testAlert()"/> <br>
    <input type="button" value="确认框" onclick="testConfirm()"/> <br>
    <input type="button" value="对话框" onclick="testPrompt()"/> <br>
</body>
```

> 页面跳转:

```
<head>
    <meta charset="UTF-8">
    <title>小标题</title>
    <script>
        function goAtguigu(){
            var flag =confirm("即将跳转到尚硅谷官网,本页信息即将丢失,确定吗?")
            if(flag){
                // 通过BOM编程地址栏url切换
                window.location.href="http://www.atguigu.com"
            }
        }
    </script>
</head>
<body>
    <input type="button" value="跳转到尚硅谷" onclick="goAtguigu()"/> <br>
</body>
```

6.5 通过BOM编程实现会话级和持久级数据存储

- 会话级数据： 内存型数据，是浏览器在内存上临时存储的数据，浏览器关闭后数据失去，通过window的sessionStorge实现；
- 持久级数据： 磁盘型数据，是浏览器在磁盘上持久存储的数据，浏览器关闭后数据仍在，通过window的localStorge实现；
- 在F12开发者工具的应用程序栏，可以查看数据的状态；


```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
    <script>
        function saveItem(){
            // 让浏览器存储一些会话级数据
            window.sessionStorage.setItem("sessionMsg","sessionValue")
            // 让浏览器存储一些持久级数据
            window.localStorage.setItem("localMsg","localValue")
            console.log("haha")
        }
        function removeItem(){
            // 删除数据
            sessionStorage.removeItem("sessionMsg")
            localStorage.removeItem("localMsg")
        }
        function readItem(){
            console.log("read")
            // 读取数据
            console.log("session:"+sessionStorage.getItem("sessionMsg"))
            console.log("local:"+localStorage.getItem("localMsg"))
        }
    </script>
</head>
<body>
    <button onclick="saveItem()">存储数据</button>
    <button onclick="removeItem()">删除数据</button>
    <button onclick="readItem()">读取数据</button>
</body>
</html>
```

七 DOM编程

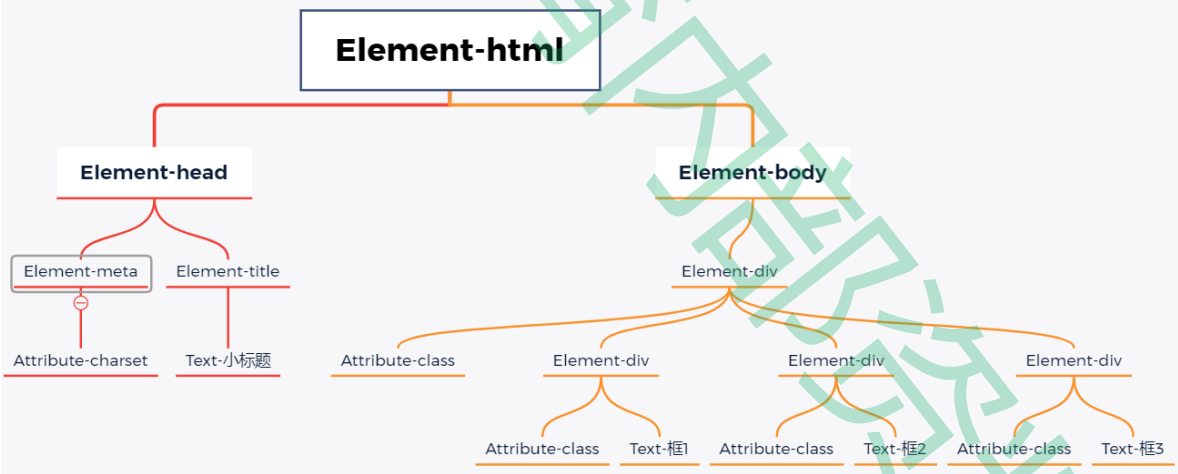
7.1 什么是DOM编程

> 简单来说：DOM(Document Object Model)编程就是使用document对象的API，完成对网页HTML文档进行动态修改，以实现网页数据和样式动态变化效果的编程。

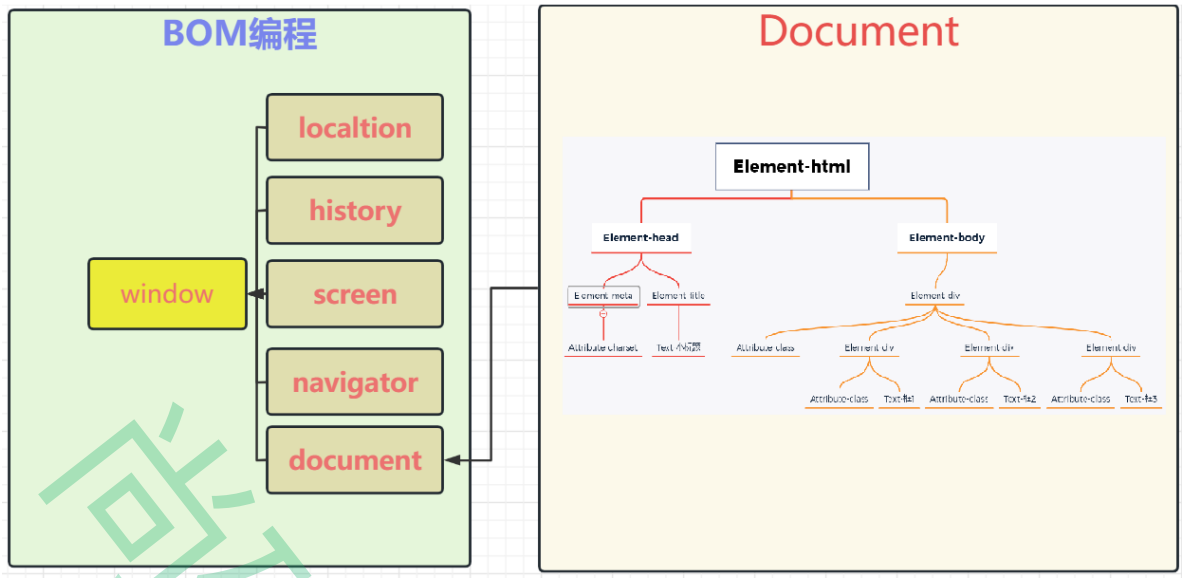
- document对象代表整个html文档，可用来访问页面中的所有元素，是最复杂的一个dom对象，可以说是学习好dom编程的关键所在；
- 根据HTML代码结构特点，document对象本身是一种树形结构的文档对象；

```
<html lang="en">
    <head>
        <meta charset="UTF-8">
        <title>小标题</title>
    </head>
    <body>
        <div class="outerDiv">
            <div class="innerDiv d1">框1</div>
            <div class="innerDiv d2">框2</div>
            <div class="innerDiv d3">框3</div>
        </div>
    </body>
</html>
```

- 上面的代码生成的树如下：



- DOM编程其实就是用window对象的document属性的相关API完成对页面元素的控制的编程：



- dom树中节点的类型：
  - node 节点，所有结点的父类型：
    - element 元素节点，node的子类型之一，代表一个完整标签；
    - attribute 属性节点，node的子类型之一，代表元素的属性；
    - text 文本节点，node的子类型之一，代表双标签中间的文本；

7.2 获取页面元素的几种方式

7.2.1 在整个文档范围内查找元素结点

功能	API	返回值
根据id值查询	document.getElementById(“id值”)	元素节点
根据标签名查询	document.getElementsByTagName(“标签名”)	元素节点数组
根据name属性值查询	document.getElementsByName(“name值”)	元素节点数组
根据类名查询	document.getElementsByClassName("类名")	元素节点数组

7.2.2 在具体元素节点范围内查找子节点

功能	API	返回值
查找子标签	element.children	返回子标签数组
查找第一个子标签	element.firstElementChild	标签对象
查找最后一个子标签	element.lastElementChild	节点对象

7.2.3 查找指定子元素节点的父节点

功能	API	返回值
查找指定元素节点的父标签	element.parentElement	标签对象

7.2.4 查找指定元素节点的兄弟节点

功能	API	返回值
查找前一个兄弟标签	node.previousElementSibling	标签对象
查找后一个兄弟标签	node.nextElementSibling	标签对象

代码见视频 p47

7.3 操作元素属性值

7.3.1 属性操作

需求	操作方式
读取属性值	元素对象.属性名
修改属性值	元素对象.属性名=新的属性值

7.3.2 内部文本操作

需求	操作方式
获取或者设置标签体的文本内容	element.innerText
获取或者设置标签体的内容	element.innerHTML

代码见视频 p48

7.4 增删元素

7.4.1 对页面的元素进行增删操作

API	功能
document.createElement(“标签名”)	创建元素节点并返回，但不会自动添加到文档中
document.createTextNode(“文本值”)	创建文本节点并返回，但不会自动添加到文档中
element.appendChild(ele)	将ele添加到element所有子节点后面
parentEle.insertBefore(newEle,targetEle)	将newEle插入到targetEle前面
parentEle.replaceChild(newEle, oldEle)	用新节点替换原有的旧子节点
element.remove()	删除某个标签

```
<!DOCTYPE html>
```

```
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
   <script>
    /*
    1 获得document  dom树
        window.document
    2 从document中获取要操作的元素
        1）直接获取
            var el1 =document.getElementById("username") // 根据元素的id值获取页面上唯一的一个元素
            var els =document.getElementsByTagName("input") // 根据元素的标签名获取多个同名元素
            var els =document.getElementsByName("aaa") // 根据元素的name属性值获得多个元素
            var els =document.getElementsByClassName("a") // 根据元素的class属性值获得多个元素
        2）间接获取
            var cs=div01.children // 通过父元素获取全部的子元素
            var firstChild =div01.firstElementChild  // 通过父元素获取第一个子元素
            var lastChild = div01.lastElementChild   // 通过父元素获取最后一个子元素
            var parent = pinput.parentElement  // 通过子元素获取父元素
            var pElement = pinput.previousElementSibling // 获取前面的第一个元素
            var nElement = pinput.nextElementSibling // 获取后面的第一个元素
    3 对元素进行操作
        1）操作元素的属性   元素名.属性名=""
        2）操作元素的样式   元素名.style.样式名=""  样式名"-" 要进行驼峰转换
        3）操作元素的文本   元素名.innerText   只识别文本
                            元素名.innerHTML   同时可以识别html代码
        4）增删元素
            var element =document.createElement("元素名") // 创建元素
            父元素.appendChild(子元素)                  // 在父元素中追加子元素
            父元素.insertBefore(新元素,参照元素)        // 在某个元素前增加元素
            父元素.replaceChild(新元素,被替换的元素) // 用新的元素替换某个子子元素
            元素.remove()                                 // 删除当前元素
    */
   function addCs(){
        // 创建一个新的元素
        // 创建元素
        var csli =document.createElement("li") // <li></li>
        // 设置子元素的属性和文本 <li id="cs">长沙</li>
        csli.id="cs"
        csli.innerText="长沙"
        // 将子元素放入父元素中
        var cityul =document.getElementById("city")
        // 在父元素中追加子元素
        cityul.appendChild(csli)
   }
   function addCsBeforeSz(){
        // 创建一个新的元素
        // 创建元素
        var csli =document.createElement("li") // <li></li>
        // 设置子元素的属性和文本 <li id="cs">长沙</li>
        csli.id="cs"
        csli.innerText="长沙"
        // 将子元素放入父元素中
        var cityul =document.getElementById("city")
```

```
        // 在父元素中追加子元素
        //cityul.insertBefore(新元素,参照元素)
        var szli =document.getElementById("sz")
        cityul.insertBefore(csli,szli)
    }
    function replaceSz(){
        // 创建一个新的元素
        // 创建元素
        var csli =document.createElement("li") // <li></li>
        // 设置子元素的属性和文本 <li id="cs">长沙</li>
        csli.id="cs"
        csli.innerText="长沙"
        // 将子元素放入父元素中
        var cityul =document.getElementById("city")
        // 在父元素中追加子元素
        //cityul.replaceChild(新元素,被替换的元素)
        var szli =document.getElementById("sz")
        cityul.replaceChild(csli,szli)
    }
    function removeSz(){
        var szli =document.getElementById("sz")
        // 哪个元素调用了remove该元素就会从dom树中移除
        szli.remove()
    }
    function clearCity(){
        var cityul =document.getElementById("city")
        /* var fc =cityul.firstChild
        while(fc != null ){
            fc.remove()
            fc =cityul.firstChild
        } */
        cityul.innerHTML=""
        //cityul.remove()
    }
    </script>
</head>
<body>
    <ul id="city">
        <li id="bj">北京</li>
        <li id="sh">上海</li>
        <li id="sz">深圳</li>
        <li id="gz">广州</li>
    </ul>
    <hr>
    <!-- 目标1 在城市列表的最后添加一个子标签  <li id="cs">长沙</li>  -->
    <button onclick="addCs()">增加长沙</button>
    <!-- 目标2 在城市列表的深圳前添加一个子标签  <li id="cs">长沙</li>  -->
    <button onclick="addCsBeforeSz()">在深圳前插入长沙</button>
    <!-- 目标3  将城市列表的深圳替换为  <li id="cs">长沙</li>  -->
    <button onclick="replaceSz()">替换深圳</button>
    <!-- 目标4  将城市列表删除深圳  -->
    <button onclick="removeSz()">删除深圳</button>
    <!-- 目标5  清空城市列表  -->
    <button onclick="clearCity()">清空</button>
</body>
</html>
```

八 正则表达式

8.1 正则表达式简介

> 正则表达式是描述字符模式的对象。正则表达式用于对字符串模式匹配及检索替换，是对字符串执行模式匹配的强大工具。

- 语法

```
var patt=new RegExp(pattern,modifiers);
或者更简单的方式：
var patt=/pattern/modifiers;
```

> 修饰符：

修饰符	描述
i	执行对大小写不敏感的匹配。
g	执行全局匹配（查找所有匹配而非在找到第一个匹配后停止）。
m	执行多行匹配。

> 方括号：

表达式	描述
[abc]	查找方括号之间的任何字符。
[^abc]	查找任何不在方括号之间的字符。
[0-9]	查找任何从 0 至 9 的数字。
[a-z]	查找任何从小写 a 到小写 z 的字符。
[A-Z]	查找任何从大写 A 到大写 Z 的字符。
[A-z]	查找任何从大写 A 到小写 z 的字符。
[adgk]	查找给定集合内的任何字符。
[^adgk]	查找给定集合外的任何字符。
(red|blue|green)	查找任何指定的选项。

> 元字符：

元字符	描述
.	查找单个字符，除了换行和行结束符。
\w	查找数字、字母及下划线。
\W	查找非单词字符。
\d	查找数字。
\D	查找非数字字符。
\s	查找空白字符。
\S	查找非空白字符。
\b	匹配单词边界。
\B	匹配非单词边界。
\0	查找 NULL 字符。
\n	查找换行符。
\f	查找换页符。
\r	查找回车符。
\t	查找制表符。
\v	查找垂直制表符。
\xxx	查找以八进制数 xxx 规定的字符。
\xdd	查找以十六进制数 dd 规定的字符。
\uxxxx	查找以十六进制数 xxxx 规定的 Unicode 字符。

> 量词：

量词	描述
n+	匹配任何包含至少一个 n 的字符串。例如，/a+/ 匹配 "candy" 中的 "a"， "caaaaaaandy" 中所有的 "a"。
n*	匹配任何包含零个或多个 n 的字符串。例如，/bo*/ 匹配 "A ghost booooed" 中的 "boooo"， "A bird warbled" 中的 "b"，但是不匹配 "A goat grunted"。
n?	匹配任何包含零个或一个 n 的字符串。例如，/e?le?/ 匹配 "angel" 中的 "el"， "angle" 中的 "le"。
n{X}	匹配包含 X 个 n 的序列的字符串。例如，/a{2}/ 不匹配 "candy," 中的 "a"，但是匹配 "caandy," 中的两个 "a"，且匹配 "caaandy." 中的前两个 "a"。
n{X,}	X 是一个正整数。前面的模式 n 连续出现至少 X 次时匹配。例如，/a{2,}/ 不匹配 "candy" 中的 "a"，但是匹配 "caandy" 和 "caaaaaaandy." 中所有的 "a"。
n{X,Y}	X 和 Y 为正整数。前面的模式 n 连续出现至少 X 次，至多 Y 次时匹配。例如，/a{1,3}/ 不匹配 "cndy"，匹配 "candy," 中的 "a"， "caandy," 中的两个 "a"，匹配 "caaaaaaandy" 中的前面三个 "a"。注意，当匹配 "caaaaaaandy" 时，即使原始字符串拥有更多的 "a"，匹配项也是 "aaa"。
n$	匹配任何结尾为 n 的字符串。
^n	匹配任何开头为 n 的字符串。
?=n	匹配任何其后紧接指定字符串 n 的字符串。
?!n	匹配任何其后没有紧接指定字符串 n 的字符串。

> RegExp对象方法：

方法	描述
compile	在 1.5 版本中已废弃。 编译正则表达式。
exec	检索字符串中指定的值。返回找到的值，并确定其位置。
test	检索字符串中指定的值。返回 true 或 false。
toString	返回正则表达式的字符串。

> 支持正则的String的方法：

方法	描述
search	检索与正则表达式相匹配的值。
match	找到一个或多个正则表达式的匹配。
replace	替换与正则表达式匹配的子串。
split	把字符串分割为字符串数组。

8.2 正则表达式体验

8.2.1 验证

注意：这里是使用正则表达式对象来调用方法。

```
// 创建一个最简单的正则表达式对象
var reg = /o/;
// 创建一个字符串对象作为目标字符串
var str = 'Hello World!';
// 调用正则表达式对象的test()方法验证目标字符串是否满足我们指定的这个模式，返回结果true
console.log("/o/.test('Hello World!')="+reg.test(str));
```

8.2.2 匹配

```
// 创建一个最简单的正则表达式对象
var reg = /o/;
// 创建一个字符串对象作为目标字符串
var str = 'Hello World!';
// 在目标字符串中查找匹配的字符，返回匹配结果组成的数组
var resultArr = str.match(reg);
// 数组长度为1
console.log("resultArr.length="+resultArr.length);
// 数组内容是o
console.log("resultArr[0]="+resultArr[0]);
```

8.2.3 替换

注意：这里是使用字符串对象来调用方法。

```
// 创建一个最简单的正则表达式对象
var reg = /o/;
// 创建一个字符串对象作为目标字符串
var str = 'Hello World!';
var newStr = str.replace(reg,'@');
// 只有第一个o被替换了，说明我们这个正则表达式只能匹配第一个满足的字符串
console.log("str.replace(reg)="+newStr);//Hell@ World!
// 原字符串并没有变化，只是返回了一个新字符串
console.log("str="+str);//str=Hello World!
```

8.2.4 全文查找

- 如果不使用g对正则表达式对象进行修饰，则使用正则表达式进行查找时，仅返回第一个匹配。使用g后，返回所有匹配。

```
// 目标字符串
var targetStr = 'Hello World!';
// 没有使用全局匹配的正则表达式
var reg = /[A-Z]/;
// 获取全部匹配
var resultArr = targetStr.match(reg);
// 数组长度为1
console.log("resultArr.length="+resultArr.length);
// 遍历数组，发现只能得到'H'
for(var i = 0; i < resultArr.length; i++){
  console.log("resultArr["+i+"]="+resultArr[i]);
}
```

- 对比

```
// 目标字符串
var targetStr = 'Hello World!';
// 使用了全局匹配的正则表达式
var reg = /[A-Z]/g;
// 获取全部匹配
var resultArr = targetStr.match(reg);
// 数组长度为2
console.log("resultArr.length="+resultArr.length);
// 遍历数组，发现可以获取到“H”和“W”
for(var i = 0; i < resultArr.length; i++){
  console.log("resultArr["+i+"]="+resultArr[i]);
}
```

8.2.5 忽略大小写

```
//目标字符串
var targetStr = 'Hello WORLD!';
//没有使用忽略大小写的正则表达式
var reg = /o/g;
//获取全部匹配
var resultArr = targetStr.match(reg);
//数组长度为1
console.log("resultArr.length="+resultArr.length);
//遍历数组，仅得到'o'
for(var i = 0; i < resultArr.length; i++){
  console.log("resultArr["+i+"]="+resultArr[i]);
}
```

- 对比

```
//目标字符串
var targetStr = 'Hello WORLD!';
//使用了忽略大小写的正则表达式
var reg = /o/gi;
//获取全部匹配
var resultArr = targetStr.match(reg);
//数组长度为2
console.log("resultArr.length="+resultArr.length);
//遍历数组，得到'o'和'O'
for(var i = 0; i < resultArr.length; i++){
  console.log("resultArr["+i+"]="+resultArr[i]);
}
```

8.2.6 元字符使用

```
var str01 = 'I love Java';
var str02 = 'Java love me';
// 匹配以Java开头
var reg = /^Java/g;
console.log('reg.test(str01)='+reg.test(str01)); // false
console.log("<br />");
console.log('reg.test(str02)='+reg.test(str02)); // true
```

```
var str01 = 'I love Java';
var str02 = 'Java love me';
// 匹配以Java结尾
var reg = /Java$/g;
console.log('reg.test(str01)='+reg.test(str01)); // true
console.log("<br />");
console.log('reg.test(str02)='+reg.test(str02)); // false
```

8.2.7 字符集合的使用

```
//n位数字的正则
var targetStr="123456789";
var reg=/^[0-9]{0,}$/;
//或者 ：  var reg=/^\d*$/;
var b = reg.test(targetStr);//true
```

```
//数字+字母+下划线，6-16位
var targetStr="HelloWorld";
var reg=/^[a-z0-9A-Z_]{6,16}$/;
var b = reg.test(targetStr);//true
```

8.2.8 常用正则表达式

需求	正则表达式
用户名	/^[a-zA-Z][a-zA-Z-0-9]{5,9}$/
密码	/^[a-zA-Z0-9 _-@#& *]{6,12}$/
前后空格	/^\s+|\s+$/g
电子邮箱	/^[a-zA-Z0-9 _.-]+@([a-zA-Z0-9-]+[.]{1})+[a-zA-Z]+$/

九 案例开发-日程管理-第一期

9.1 登录页及校验

欢迎使用日程管理系统

请登录

请输入账号	
请输入密码	
登录 重置 去注册	

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
    <style>
        .ht{
            text-align: center;
            color: cadetblue;
            font-family: 幼圆;
        }
        .tab{
            width: 500px;
            border: 5px solid cadetblue;
            margin: 0px auto;
            border-radius: 5px;
            font-family: 幼圆;
        }
        .ltr td{
            border: 1px solid  powderblue;
        }
        .ipt{
            border: 0px;
            width: 50%;
        }
        .btn1{
            border: 2px solid powderblue;
            border-radius: 4px;
            width:60px;
            background-color: antiquewhite;
        }
        #usernameMsg , #userPwdMsg {
            color: rgb(230, 87, 51);
        }
        .buttonContainer{
            text-align: center;
        }
    </style>
    <script>
        // 检验用户名格式是否合法的函数
        function checkUsername(){
            // 定义正则表示字符串的规则
            var  usernameReg= /^[a-zA-Z0-9]{5,10}$/
            // 获得用户在页面上输入的信息
            var usernameInput =document.getElementById("usernameInput")
            var username = usernameInput.value
```

```
            // 获得格式提示的框
            var usernameMsg =document.getElementById("usernameMsg")
            // 格式有误时,返回false,在页面上提示
            if(!usernameReg.test(username)){
                usernameMsg.innerText="用户名格式有误"
                return false
            }
            // 格式OK,返回true 在页面上提示OK
            usernameMsg.innerText="OK"
            return true
        }
        // 检验密码格式是否合法的函数
        function checkUserPwd(){
            // 定义正则表示字符串的规则
            var  userPwdReg= /^[0-9]{6}$/
            // 获得用户在页面上输入的信息
            var userPwdInput =document.getElementById("userPwdInput")
            var userPwd = userPwdInput.value
            // 获得格式提示的框
            var userPwdMsg =document.getElementById("userPwdMsg")
            // 格式有误时,返回false,在页面上提示
            if(!userPwdReg.test(userPwd)){
                userPwdMsg.innerText="密码必须是6位数字"
                return false
            }
            // 格式OK,返回true 在页面上提示OK
            userPwdMsg.innerText="OK"
            return true
        }
        // 表单在提交时,校验用户名和密码格式,格式OK才会提交
        function checkForm(){
            var flag1 =checkUsername()
            var flag2 =checkUserPwd()
            return flag1&&flag2
        }
    </script>
</head>
<body>
    <h1 class="ht">欢迎使用日程管理系统</h1>
    <h3 class="ht">请登录</h3>
    <form method="post" action="/user/login" onsubmit="return checkForm()">
        <table class="tab" cellspacing="0px">
            <tr class="ltr">
                <td>请输入账号</td>
                <td>
                    <input class="ipt" type="text" id="usernameInput" name="username"
onblur="checkUsername()">
                    <span id="usernameMsg"></span>
                </td>
            </tr>
            <tr class="ltr">
                <td>请输入密码</td>
                <td>
                    <input class="ipt" type="password" id="userPwdInput"
 name="userPwd" onblur="checkUserPwd()">
                    <span id="userPwdMsg"></span>
                </td>
            </tr>
```

```
            <tr class="ltr">
                <td colspan="2" class="buttonContainer">
                    <input class="btn1" type="submit" value="登录">
                    <input class="btn1" type="reset" value="重置">
                    <button class="btn1"><a href="regist.html">去注册</a></button>
                </td>
            </tr>
        </table>
    </form>
</body>
</html>
```

9.2 注册页及校验


```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
    <style>
        .ht{
            text-align: center;
            color: cadetblue;
            font-family: 幼圆;
        }
        .tab{
            width: 500px;
            border: 5px solid cadetblue;
            margin: 0px auto;
            border-radius: 5px;
            font-family: 幼圆;
        }
        .ltr td{
            border: 1px solid  powderblue;
        }
        .ipt{
            border: 0px;
            width: 50%;
        }
        .btn1{
            border: 2px solid powderblue;
            border-radius: 4px;
            width:60px;
```

```
            background-color: antiquewhite;
        }
        .msg {
            color: gold;
        }
        .buttonContainer{
            text-align: center;
        }
    </style>
    <script>
        function checkUsername(){
            var usernameReg = /^[a-zA-Z0-9]{5,10}$/
            var usernameInput = document.getElementById("usernameInput")
            var username = usernameInput.value
            var usernameMsg = document.getElementById("usernameMsg")
            if(!usernameReg.test(username)){
                usernameMsg.innerText="格式有误"
                return false
            }
            usernameMsg.innerText="OK"
            return true
        }
        function checkUserPwd(){
            var userPwdReg = /^\d{6}$/
            var userPwdInput = document.getElementById("userPwdInput")
            var userPwd = userPwdInput.value
            var userPwdMsg = document.getElementById("userPwdMsg")
            if(!userPwdReg.test(userPwd)){
                userPwdMsg.innerText="格式有误"
                return false
            }
            userPwdMsg.innerText="OK"
            return true
        }
        function checkReUserPwd(){
            var userPwdReg = /^\d{6}$/
            // 再次输入的密码的格式
            var reUserPwdInput = document.getElementById("reUserPwdInput")
            var reUserPwd = reUserPwdInput.value
            var reUserPwdMsg = document.getElementById("reUserPwdMsg")
            if(!userPwdReg.test(reUserPwd)){
                reUserPwdMsg.innerText="格式有误"
                return false
            }
            // 获得上次密码,对比两次密码是否一致
            var userPwdInput = document.getElementById("userPwdInput")
            var userPwd = userPwdInput.value
            if(reUserPwd != userPwd){
                reUserPwdMsg.innerText="两次密码不一致"
                return false
            }
            reUserPwdMsg.innerText="OK"
            return true
        }
        function checkForm(){
            var flag1 = checkUsername()
            var flag2 = checkUserPwd()
            var flag3 = checkReUserPwd()
```

```
            return flag1 && flag2 && flag3
        }
    </script>
</head>
<body>
<h1 class="ht">欢迎使用日程管理系统</h1>
<h3 class="ht">请注册</h3>
<form method="post" action="/user/regist" onsubmit="return checkForm()">
    <table class="tab" cellspacing="0px">
        <tr class="ltr">
            <td>请输入账号</td>
            <td>
                <input class="ipt" id="usernameInput" type="text" name="username"
onblur="checkUsername()">
                <span id="usernameMsg" class="msg"></span>
            </td>
        </tr>
        <tr class="ltr">
            <td>请输入密码</td>
            <td>
                <input class="ipt" id="userPwdInput" type="password" name="userPwd"
onblur="checkUserPwd()">
                <span id="userPwdMsg" class="msg"></span>
            </td>
        </tr>
        <tr class="ltr">
            <td>确认密码</td>
            <td>
                <input class="ipt" id="reUserPwdInput" type="password"
onblur="checkReUserPwd()">
                <span id="reUserPwdMsg" class="msg"></span>
            </td>
        </tr>
        <tr class="ltr">
            <td colspan="2" class="buttonContainer">
                <input class="btn1" type="submit" value="注册">
                <input class="btn1" type="reset" value="重置">
                <button class="btn1"><a  href="login.html">去登录</a></button>
            </td>
        </tr>
    </table>
</form>
</body>
</html>
```

第四章 XML_Tomcat10_HTTP

一 XML

> XML是EXtensible Markup Language的缩写，翻译过来就是可扩展标记语言。所以很明显，XML和HTML一样都是标记语言，也就是说它们的基本语法都是标签。

- **可扩展**：三个字表面上的意思是XML允许自定义格式。但这不代表你可以随便写；
- 在XML基本语法规范的基础上，你使用的那些第三方应用程序、框架会通过XML约束的方式强制规定配置文件中可以写什么和怎么写；
- XML基本语法这个知识点的定位是：我们不需要从零开始，从头到尾的一行一行编写**XML**文档，而是在第三方应用程序、框架已提供的配置文件的基础上修改。要改成什么样取决于你的需求，而怎么改取决XML基本语法和具体的XML约束；

1.1 常见配置文件类型

1. properties：例如druid连接池就是使用properties文件作为配置文件；
2. XML,：例如Tomcat就是使用XML文件作为配置文件；
3. YAML/YML：例如SpringBoot就是使用YAML作为配置文件；
4. json：通常用来做文件传输，也可以用来做前端或者移动端的配置文件；
5. ... ...

1.1.1 properties配置文件

> 示例：

```
atguigu.jdbc.url=jdbc:mysql://localhost:3306/atguigu
atguigu.jdbc.driver=com.mysql.cj.jdbc.Driver
atguigu.jdbc.username=root
atguigu.jdbc.password=root
```

> 语法规范：

- 由键值对组成；
- 键和值之间的符号是等号；
- 每一行都必须顶格写，前面不能有空格之类的其他符号；

1.1.2 xml配置文件

> 示例：

```
<?xml version="1.0" encoding="UTF-8"?>
<students>
    <student>
        <name>张三</name>
        <age>18</age>
    </student>
    <student>
        <name>李四</name>
        <age>20</age>
    </student>
</students>
```

> XML的基本语法：

- XML的基本语法和HTML的基本语法简直如出一辙。XML基本语法+HTML约束=HTML语法。在逻辑上HTML确实是XML的子集：
  1、XML文档声明：这部分基本上就是固定格式，`<?xml version="1.0" encoding="UTF-8"?>`；
  2、根标签：根标签有且只能有一个；
  3、标签关闭：开始标签和结束标签必须成对出现，单标签在标签内关闭；
  4、标签嵌套：标签可以嵌套，但是不能交叉嵌套，注释不能嵌套；
  5、标签名、属性名建议使用小写字母；
  6、属性：属性必须有值，属性值必须加引号，单双都行；

> XML的约束(稍微了解)：将来主要就是根据XML约束中的规定来编写XML配置文件，在编写XML的时候，编辑工具会根据约束提示。XML约束主要包括DTD和Schema两种。以下是一个web.xml约束的示例：

```
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd"
         version="4.0">
```

1.2 DOM4J进行XML解析

1.2.1 DOM4J的使用步骤

1. 导入jar包 dom4j.jar
2. 创建解析器对象(SAXReader)
3. 解析xml 获得Document对象
4. 获取根节点RootElement
5. 获取根节点下的子节点

1.2.2 DOM4J的API介绍

1、创建SAXReader对象。

```
SAXReader saxReader = new SAXReader();
```

2. 解析XML获取Document对象，需要传入要解析的XML文件的字节输入流。

```
Document document = reader.read(inputStream);
```

3. 获取文档的根标签。

```
Element rootElement = documen.getRootElement();
```

4. 获取标签的子标签。

```
//获取所有子标签
List<Element> sonElementList = rootElement.elements();
//获取指定标签名的子标签
List<Element> sonElementList = rootElement.elements("标签名");
```

5. 获取标签体内的文本。

```
String text = element.getText();
```

6. 获取标签的某个属性的值。

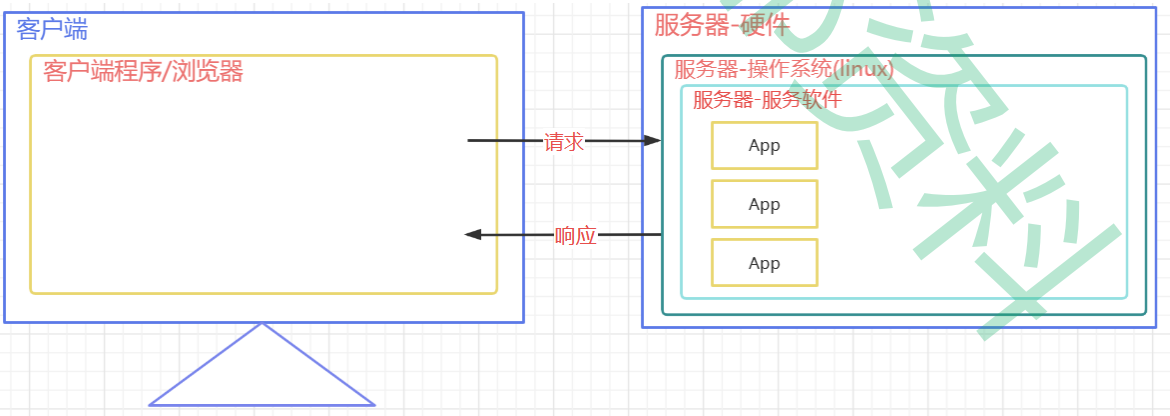
```
String value = element.attributeValue("属性名");
```

二 Tomcat10

2.1 WEB服务器

> Web服务器通常由硬件和软件共同构成。

- 硬件：电脑，提供服务供其它客户电脑访问；
- 软件：电脑上安装的服务器软件，安装后能提供服务给网络中的其他计算机，将本地文件映射成一个url地址供网络中的其他人访问；



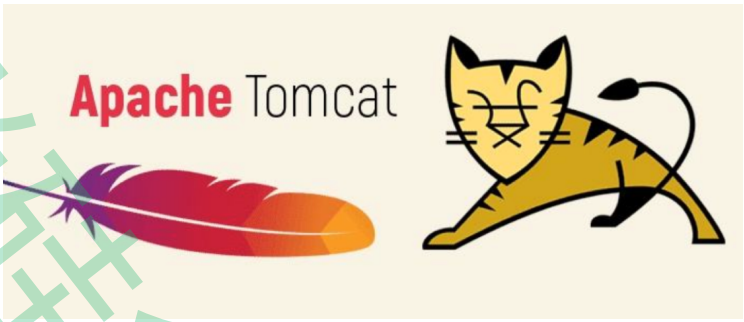
> 常见的JavaWeb服务器：

- **Tomcat（Apache）**：当前应用最广的JavaWeb服务器。
- Jetty：更轻量级、更灵活的servlet容器。
- JBoss（Redhat红帽）：支持JavaEE，应用比较广EJB容器。
- GlassFish（Orcale）：Oracle开发JavaWeb服务器。

- Resin（Caucho）：支持JavaEE，应用越来越广。
- Weblogic（Orcale）：支持JavaEE，适合大型项目。
- Websphere（IBM）：支持JavaEE，适合大型项目。

2.2 Tomcat服务器

2.2.1 简介



> Tomcat是Apache 软件基金会（Apache Software Foundation）的Jakarta 项目中的一个核心项目，由Apache、Sun 和其他一些公司及个人共同开发而成。最新的Servlet 和JSP 规范总是能在Tomcat中得到体现，因为Tomcat 技术先进、性能稳定，而且免费，因而深受Java 爱好者的喜爱并得到了部分软件开发商的认可，成为目前比较流行的Web 应用服务器。

2.2.2 安装

> 版本：

- 版本：企业用的比较广泛的是8.0和9.0，目前比较新正式发布版本是Tomcat10.0，Tomcat11仍然处于测试阶段；
- JAVAEE版本和Servlet版本号对应关系 https://jakarta.ee/release/；

Servlet Version	EE Version
6.1	Jakarta EE ?
6.0	Jakarta EE 10
5.0	Jakarta EE 9/9.1
4.0	JAVA EE 8
3.1	JAVA EE 7
3.1	JAVA EE 7
3.0	JAVAEE 6

- Tomcat 版本和Servlet版本之间的对应关系：

Servlet Version	Tomcat Version	JDK Version
6.1	11.0.x	17 and later
6.0	10.1.x	11 and later
5.0	10.0.x (superseded)	8 and later
4.0	9.0.x	8 and later
3.1	8.5.x	7 and later
3.1	8.0.x (superseded)	7 and later
3.0	7.0.x (archived)	6 and later (7 and later for WebSocket)

> 下载：

- Tomcat官方网站：http://tomcat.apache.org/；
- 安装版：需要安装，一般不考虑使用；
- 解压版：直接解压缩使用，我们使用的版本；

10.1.7

Please see the README file for packaging information. It explains what every distribution contains.

Binary Distributions

- Core:
  - zip (pgp, sha512)
  - tar.gz (pgp, sha512)
  - 32-bit Windows zip (pgp, sha512)
  - 64-bit Windows zip (pgp, sha512)
  - 32-bit/64-bit Windows Service Installer (pgp, sha512)
- Full documentation:
  - tar.gz (pgp, sha512)
- Deployer:
  - zip (pgp, sha512)
  - tar.gz (pgp, sha512)
- Embedded:
  - tar.gz (pgp, sha512)
  - zip (pgp, sha512)

程序包

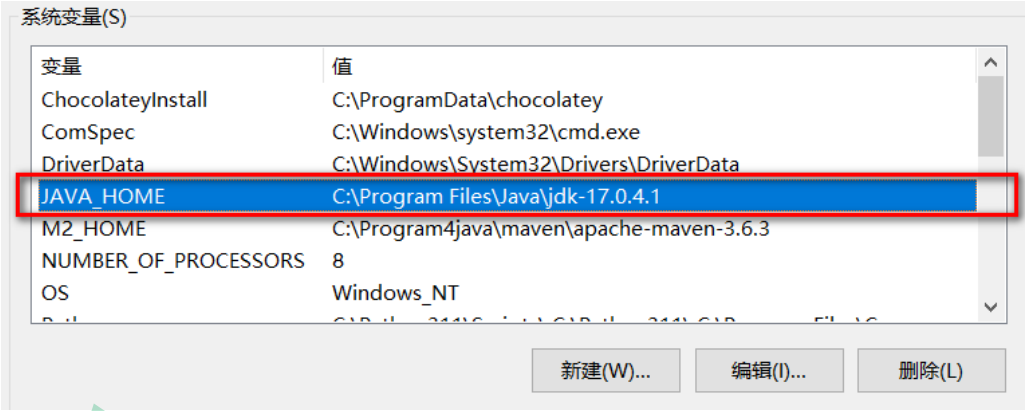
Source Code Distributions

- tar.gz (pgp, sha512)
- zip (pgp, sha512)

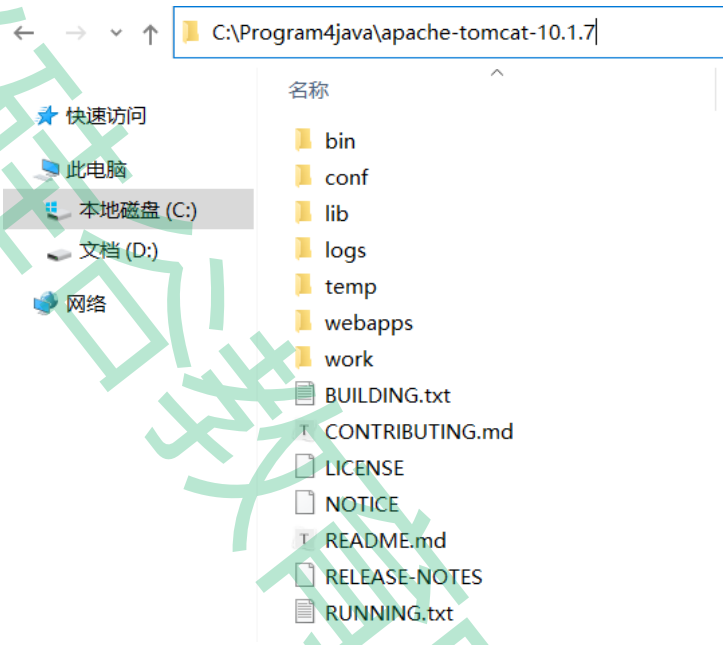
源码包

> 安装：

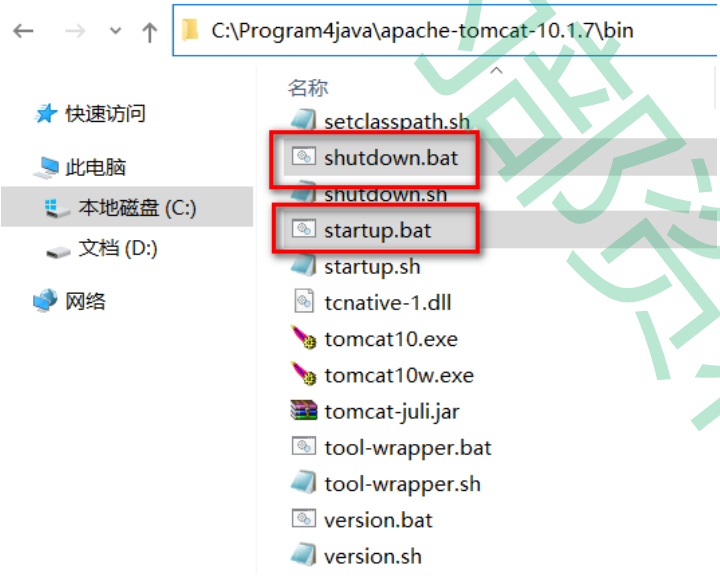
1. 正确安装JDK并配置JAVA_HOME。



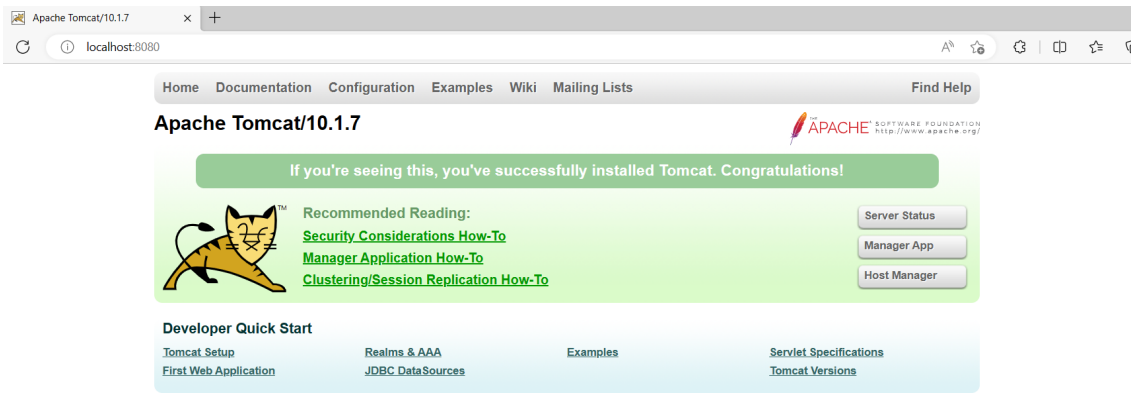
2. 解压Tomcat到非中文无空格目录。



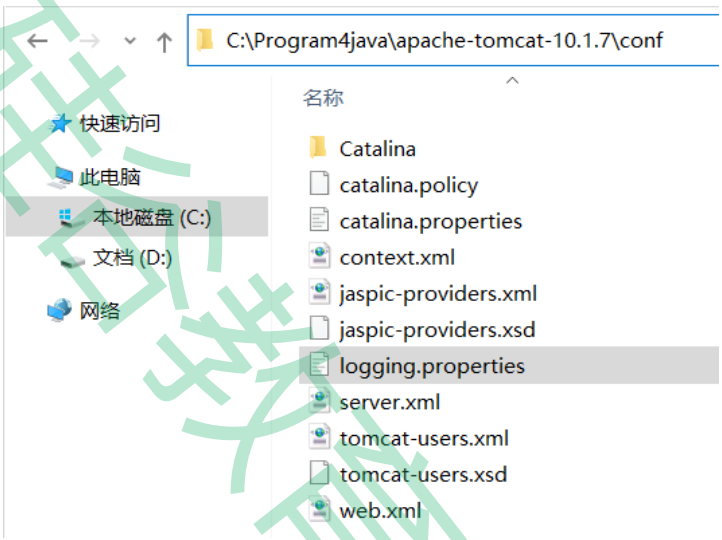
3. 点击bin/startup.bat启动。



4. 打开浏览器输入 http://localhost:8080 访问测试。



5. 直接关闭窗口或者运行 bin/shutdown.bat关闭tomcat。
6. 处理dos窗口日志中文乱码问题：修改conf/logging.properties，ConsoleHandler.encoding的UTF-8修改为GBK。



logging.properties

```
# Describes specific configuration info for Handlers.
############################################################

1catalina.org.apache.juli.AsyncFileHandler.level = FINE
1catalina.org.apache.juli.AsyncFileHandler.directory = ${catalina.base}/logs
1catalina.org.apache.juli.AsyncFileHandler.prefix = catalina.
1catalina.org.apache.juli.AsyncFileHandler.maxDays = 90
1catalina.org.apache.juli.AsyncFileHandler.encoding = UTF-8

2localhost.org.apache.juli.AsyncFileHandler.level = FINE
2localhost.org.apache.juli.AsyncFileHandler.directory = ${catalina.base}/logs
2localhost.org.apache.juli.AsyncFileHandler.prefix = localhost.
2localhost.org.apache.juli.AsyncFileHandler.maxDays = 90
2localhost.org.apache.juli.AsyncFileHandler.encoding = UTF-8

3manager.org.apache.juli.AsyncFileHandler.level = FINE
3manager.org.apache.juli.AsyncFileHandler.directory = ${catalina.base}/logs
3manager.org.apache.juli.AsyncFileHandler.prefix = manager.
3manager.org.apache.juli.AsyncFileHandler.maxDays = 90
3manager.org.apache.juli.AsyncFileHandler.encoding = UTF-8

4host-manager.org.apache.juli.AsyncFileHandler.level = FINE
4host-manager.org.apache.juli.AsyncFileHandler.directory = ${catalina.base}/logs
4host-manager.org.apache.juli.AsyncFileHandler.prefix = host-manager.
4host-manager.org.apache.juli.AsyncFileHandler.maxDays = 90
4host-manager.org.apache.juli.AsyncFileHandler.encoding = UTF-8

java.util.logging.ConsoleHandler.level = FINE
java.util.logging.ConsoleHandler.formatter = org.apache.juli.OneLineFormatter
java.util.logging.ConsoleHandler.encoding = UTF-8
```

2.3 Tomcat目录及测试

> C:\Program4java\apache-tomcat-10.1.7 这个目录下直接包含Tomcat的bin目录，conf目录等，我们称之为**Tomcat的安装目录或根目录**。

- bin：该目录下存放的是二进制可执行文件，如果是安装版，那么这个目录下会有两个exe文件：tomcat10.exe、tomcat10w.exe，前者是在控制台下启动Tomcat，后者是弹出GUI窗口启动Tomcat；如果是解压版，那么会有startup.bat和shutdown.bat文件，startup.bat用来启动Tomcat，但需要先配置JAVA_HOME环境变量才能启动，shutdown.bat用来停止Tomcat；
- conf：这是一个非常非常重要的目录，这个目录下有四个最为重要的文件：
  - **server.xml：配置整个服务器信息。例如修改端口号。默认HTTP请求的端口号是：8080**；
  - tomcat-users.xml：存储tomcat用户的文件，这里保存的是tomcat的用户名及密码，以及用户的角色信息。可以按着该文件中的注释信息添加tomcat用户，然后就可以在Tomcat主页中进入Tomcat Manager页面了；

```xml
<tomcat-users xmlns="http://tomcat.apache.org/xml"
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
              xsi:schemaLocation="http://tomcat.apache.org/xml tomcat-users.xsd"
              version="1.0">
    <role rolename="admin-gui"/>
    <role rolename="admin-script"/>
    <role rolename="manager-gui"/>
    <role rolename="manager-script"/>
    <role rolename="manager-jmx"/>
    <role rolename="manager-status"/>
    <user   username="admin"
            password="admin"
            roles="admin-gui,admin-script,manager-gui,manager-script,manager-jmx,manager-status"
    />
</tomcat-users>
```

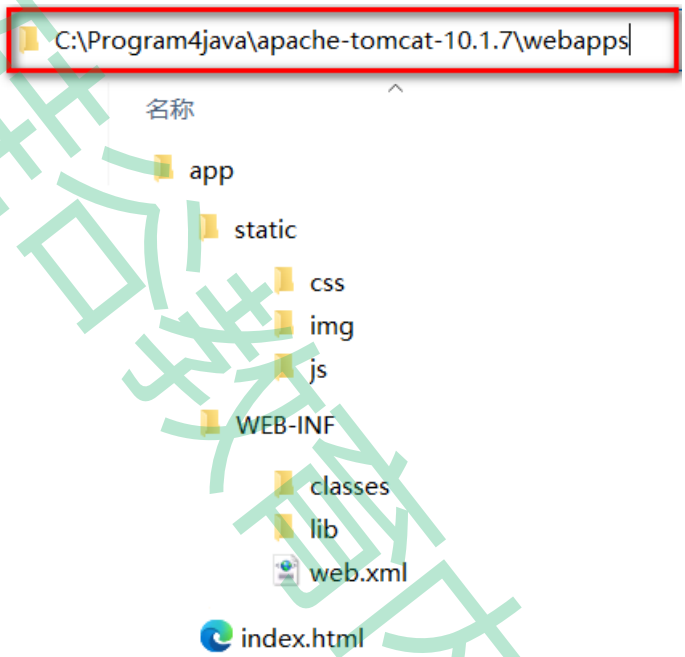
web.xml：部署描述符文件，这个文件中注册了很多MIME类型，即文档类型。这些MIME类型是客户端与服务器之间说明文档类型的，如用户请求一个html网页，那么服务器还会告诉客户端浏览器响应的文档是text/html类型的，这就是一个MIME类型。客户端浏览器通过这个MIME类型就知道如何处理它了。当然是在浏览器中显示这个html文件了。但如果服务器响应的是一个exe文件，那么浏览器就不可能显示它，而是应该弹出下载窗口才对。MIME就是用来说明文档的内容是什么类型的！

  - context.xml：对所有应用的统一配置，通常我们不会去配置它。
- lib：Tomcat的类库，里面是一大堆jar文件。如果需要添加Tomcat依赖的jar文件，可以把它放到这个目录中，当然也可以把应用依赖的jar文件放到这个目录中，这个目录中的jar所有项目都可以共享之，但这样你的应用放到其他Tomcat下时就不能再共享这个目录下的jar包了，所以建议只把Tomcat需要的jar包放到这个目录下；
- logs：这个目录中都是日志文件，记录了Tomcat启动和关闭的信息，如果启动Tomcat时有错误，那么异常也会记录在日志文件中。
- temp：存放Tomcat的临时文件，这个目录下的东西可以在停止Tomcat后删除！
- **webapps：存放web项目的目录，其中每个文件夹都是一个项目**；如果这个目录下已经存在了目录，那么都是tomcat自带的项目。其中ROOT是一个特殊的项目，在地址栏中访问：http://127.0.0.1:8080，没有给出项目目录时，对应的就是ROOT项目. http://localhost:8080/examples，进入示例项目。其中examples"就是项目名，即文件夹的名字。

- work：运行时生成的文件，最终运行的文件都在这里。通过webapps中的项目生成的！可以把这个目录下的内容删除，再次运行时会生再次生成work目录。当客户端用户访问一个JSP文件时，Tomcat会通过JSP生成Java文件，然后再编译Java文件生成class文件，生成的java和class文件都会存放到这个目录下。
- LICENSE：许可证。
- NOTICE：说明文件。

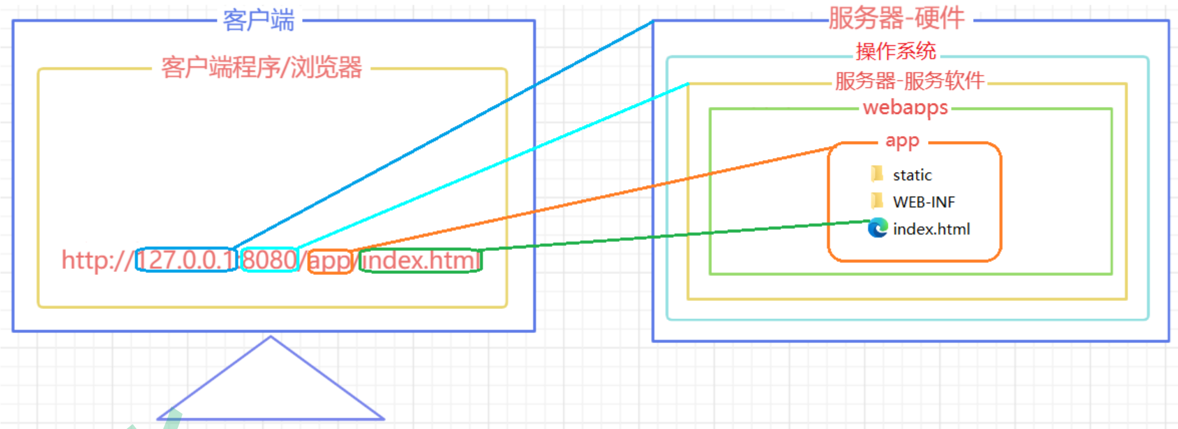
2.4 WEB项目的标准结构

> 一个标准的可以用于发布的WEB项目标准结构如下：



- app 本应用根目录
  - static 非必要目录，约定俗成的名字，一般在此处放静态资源（css js img）；
  - WEB-INF必要目录，必须叫WEB-INF。受保护的资源目录，浏览器通过url不可以直接访问的目录；
    - classes 必要目录，src下源代码、配置文件，编译后会在该目录下。web项目中如果没有Java源码，则该目录不会出现。
    - lib 必要目录，项目依赖的jar编译后会出现在该目录下，web项目要是没有依赖任何jar，则该目录不会出现。
    - web.xml 必要文件，web项目的基本配置文件.，较新的版本中可以没有该文件，但是学习过程中还是需要该文件。
  - index.html 非必要文件，index.html/index.htm/index.jsp为默认的欢迎页；

> url的组成部分和项目中资源的对应关系：



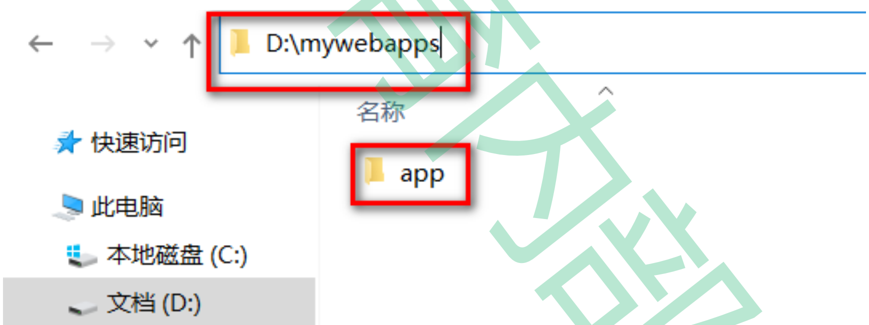
2.5 WEB项目部署的方式

> 方式1 直接将编译好的项目放在webapps目录下（已经演示）。

> 方式2 将编译好的项目打成war包放在webapps目录，tomcat启动后会自动解压war包(其实和第一种一样，后面通过maven完成)。

> 方式3 可以将项目放在非webapps的其他目录下，在Tomcat中通过配置文件指向app的实际磁盘路径。

- 在磁盘的自定义目录上准备一个app



- 在tomcat的conf下创建Catalina/localhost目录,并在该目录下准备一个app.xml文件

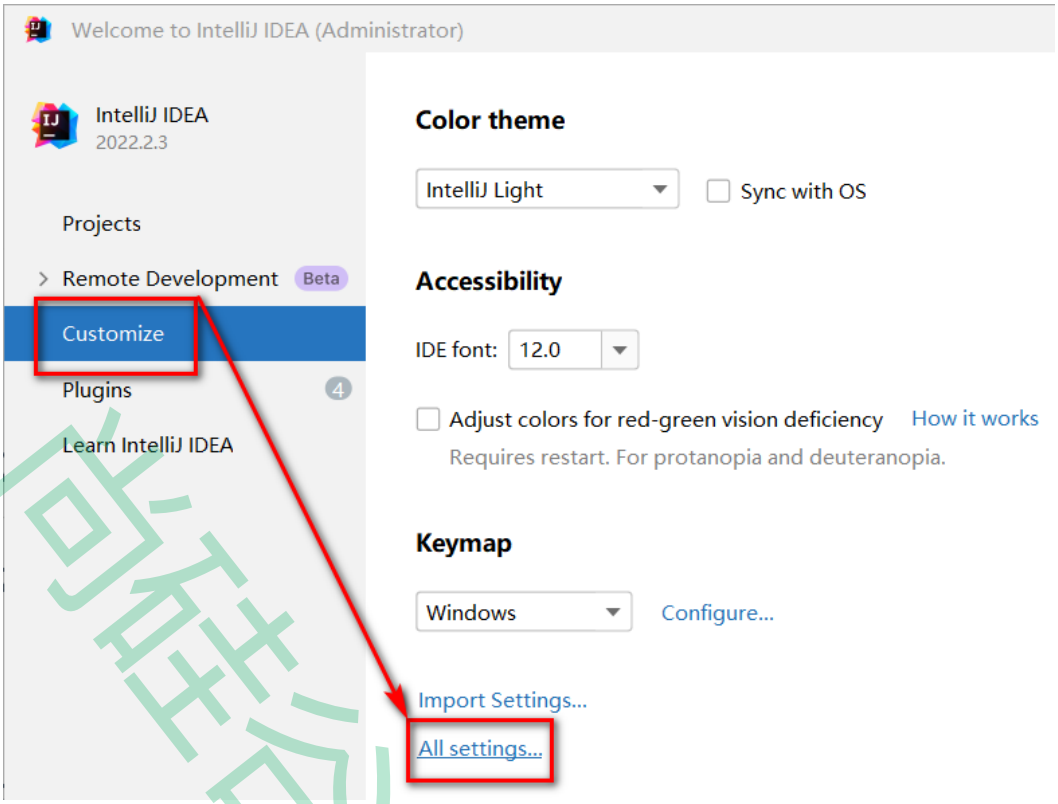
```xml
<!--
    path: 项目的访问路径,也是项目的上下文路径,就是在浏览器中,输入的项目名称
    docBase: 项目在磁盘中的实际路径
 -->
<Context path="/app" docBase="D:\mywebapps\app" />
```

- 启动Tomcat访问测试即可

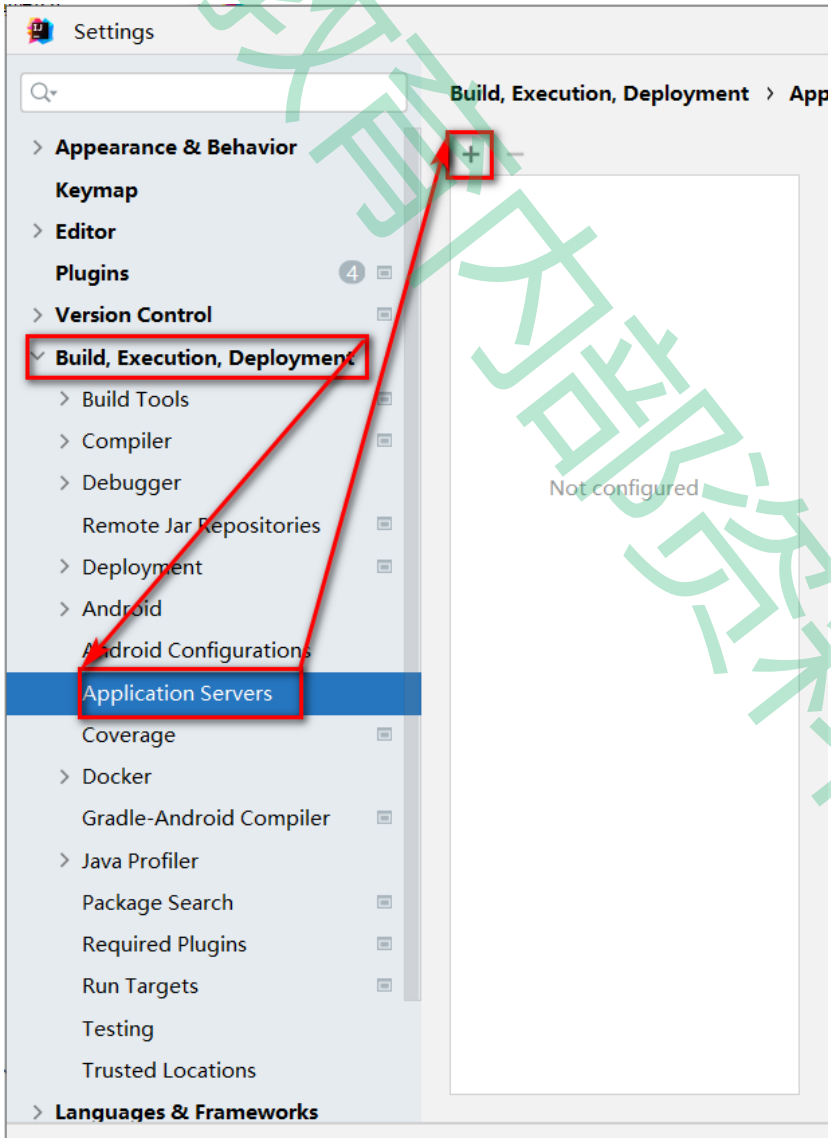
2.6 IDEA中开发并部署运行WEB项目

2.6.1 IDEA关联本地Tomcat

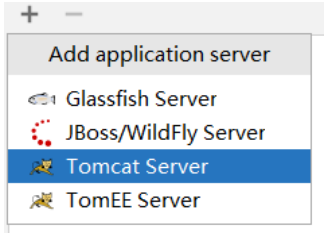
> 可以在创建项目前设置本地Tomcat，也可以在打开某个项目的状态下找到settings：



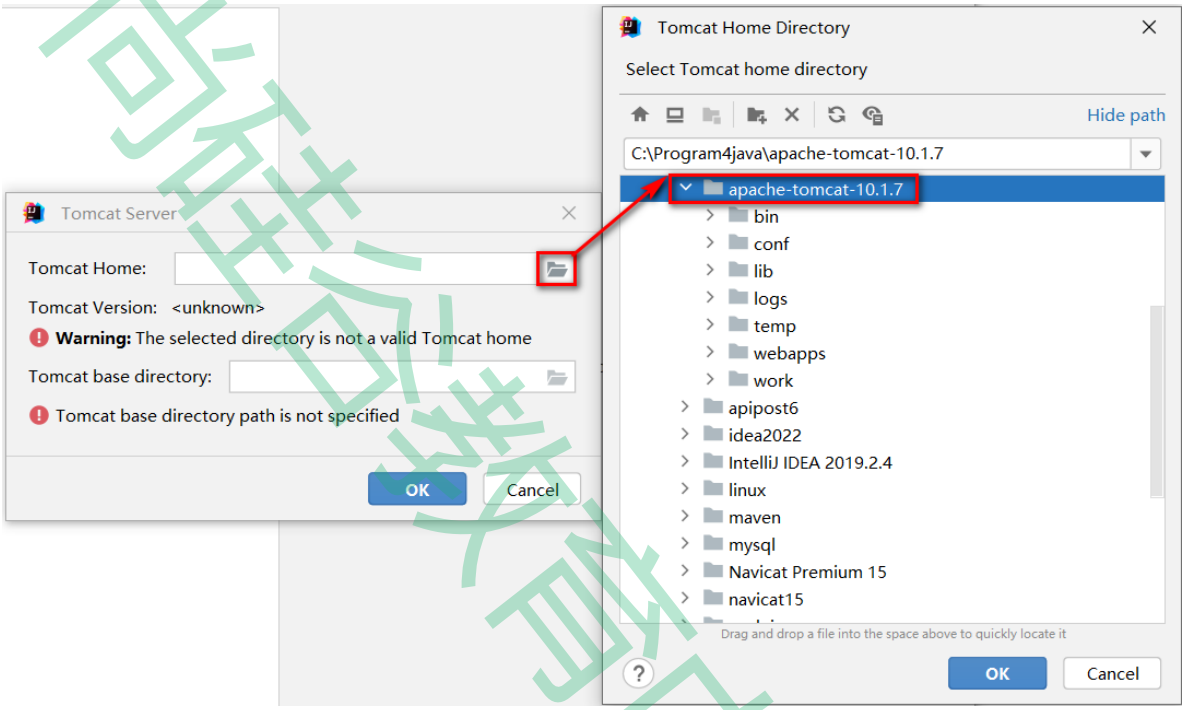
找到 Build>Execution>Eeployment下的Application Servers，找到+号：



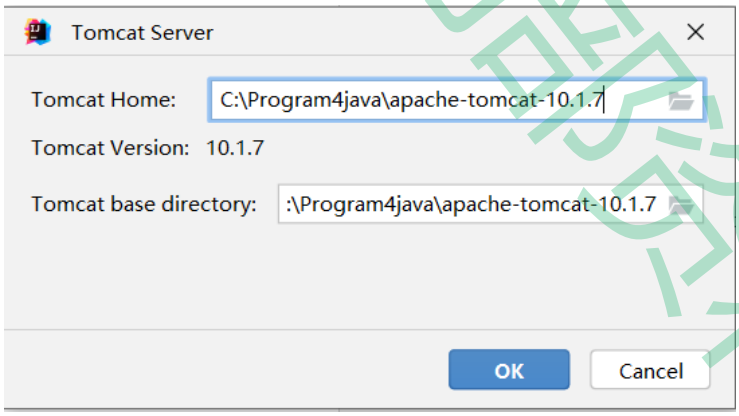
选择Tomcat Server：



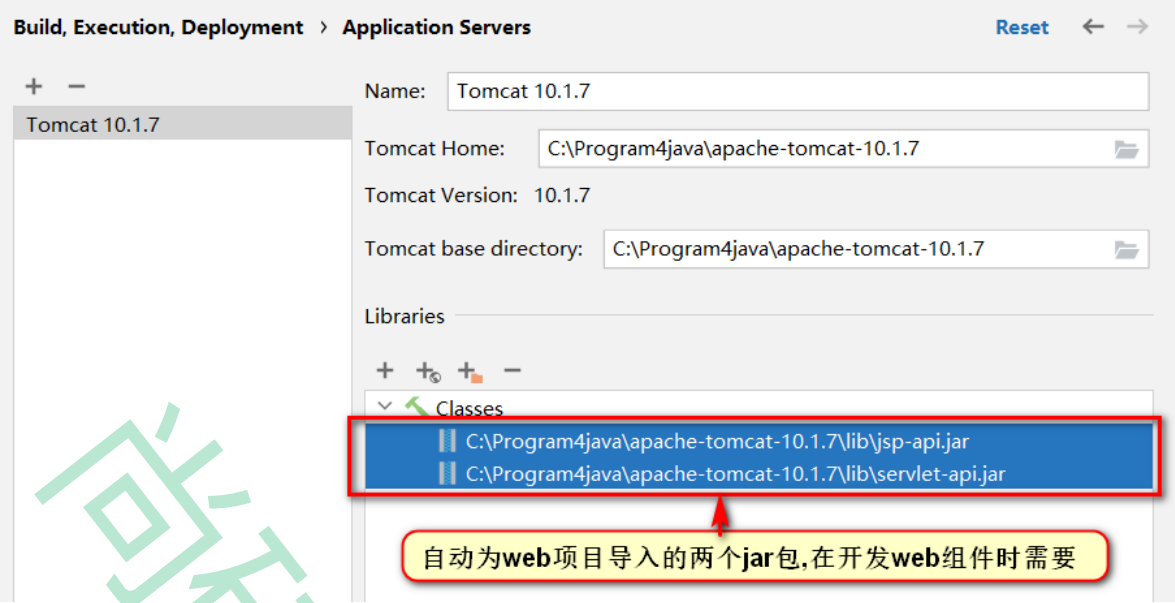
选择Tomcat的安装目录：



点击ok：

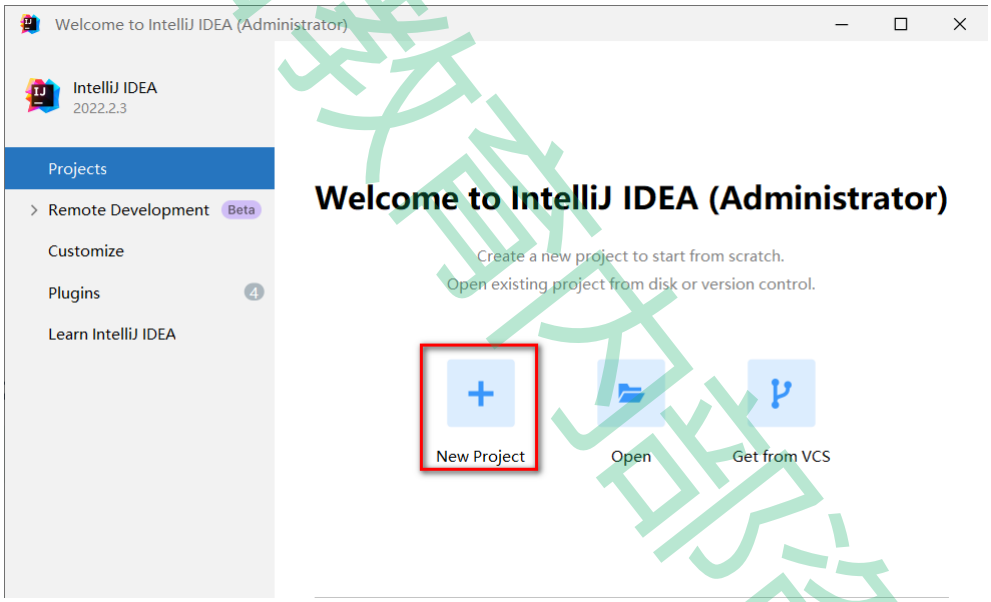


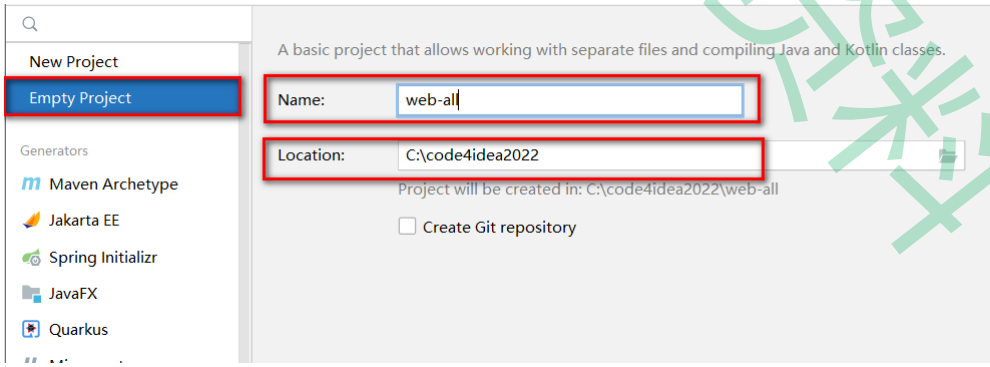
关联完毕：



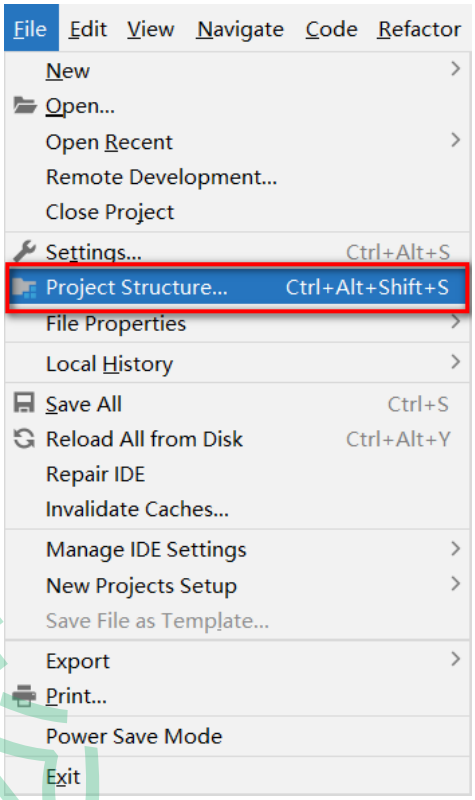
2.6.2 IDEA创建web工程

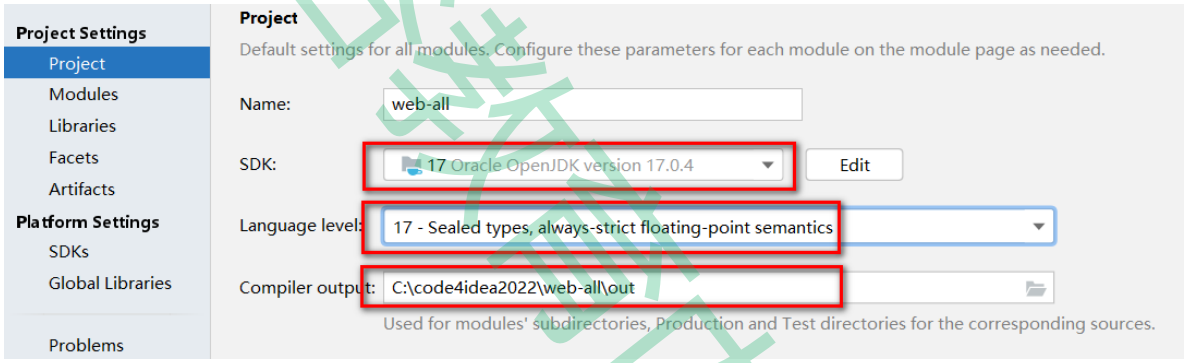
推荐先创建一个空项目，既workspace，这样可以在一个空项目下同时存在多个modules，不用后续来回切换之前的项目,当然也可以忽略此步直接创建项目：



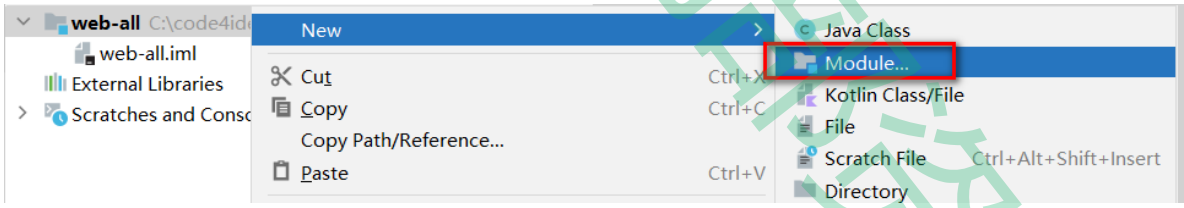


检查项目的SDK，语法版本，以及项目编译后的输出目录：

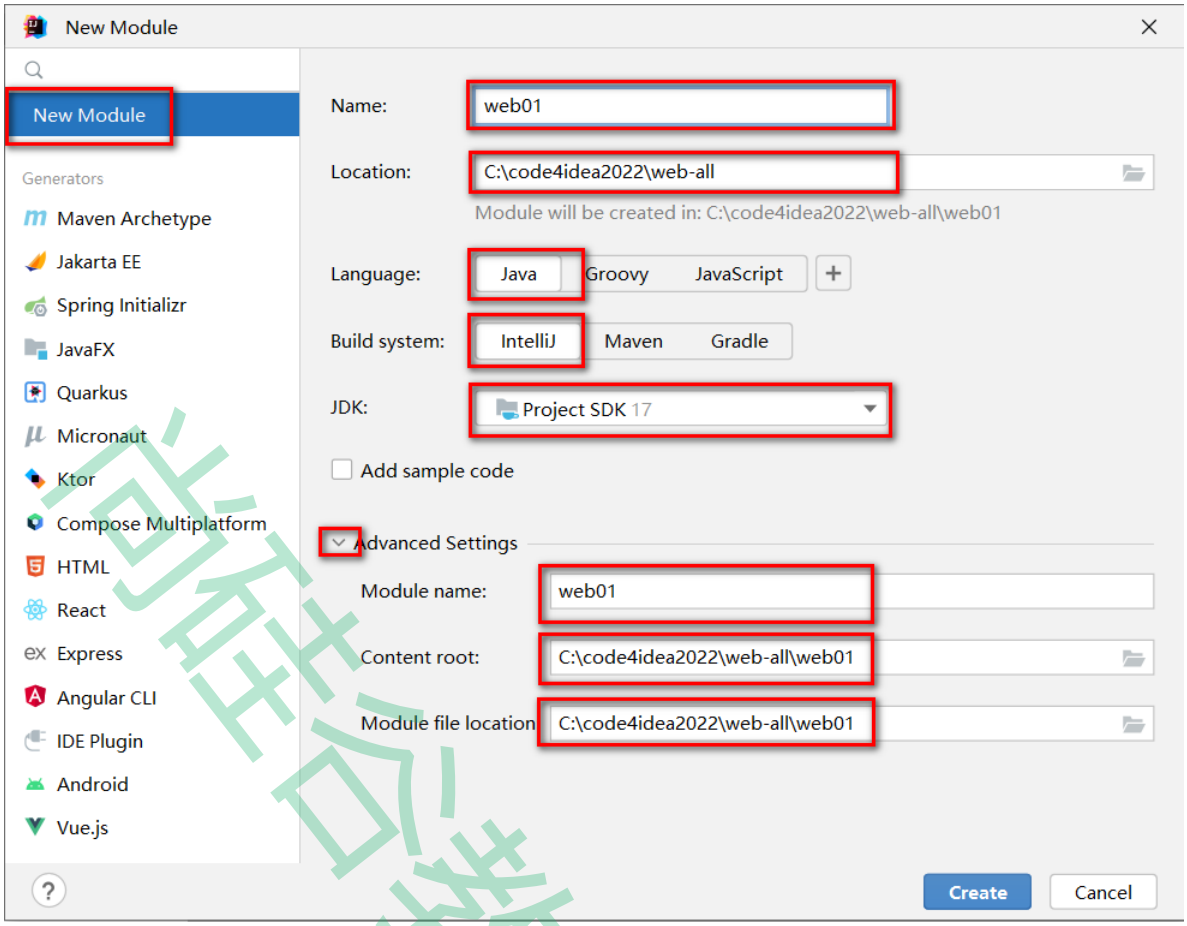




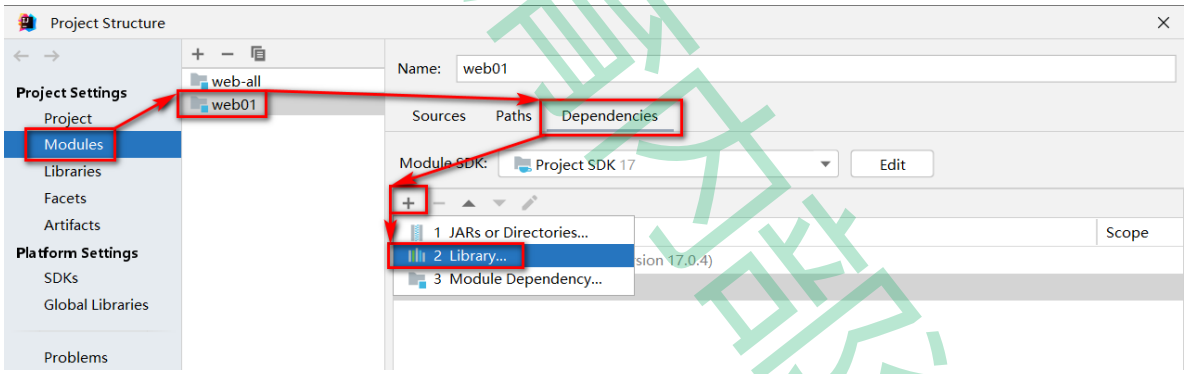
先创建一个普通的Java项目：

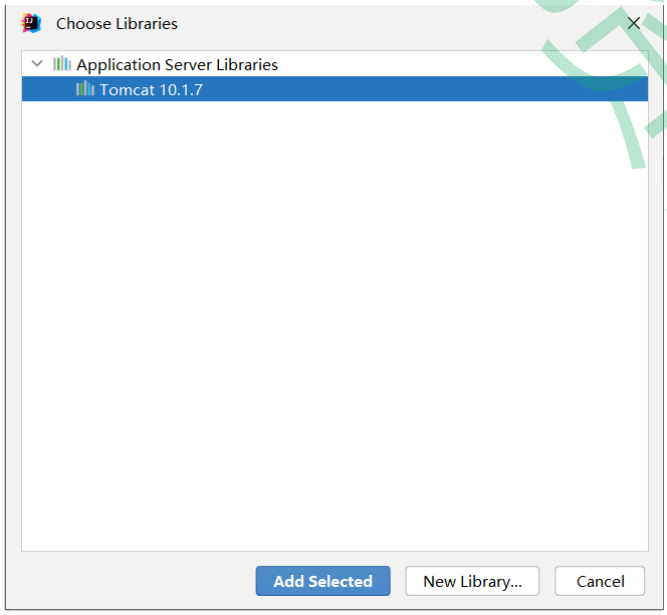


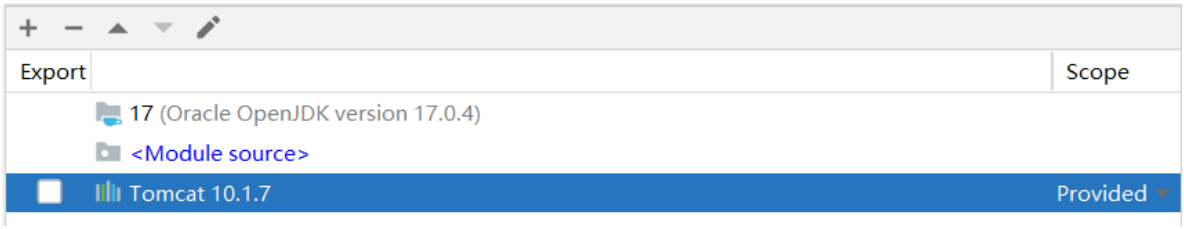
检查各项信息是否填写有误，然后点击 “create” 创建项目：



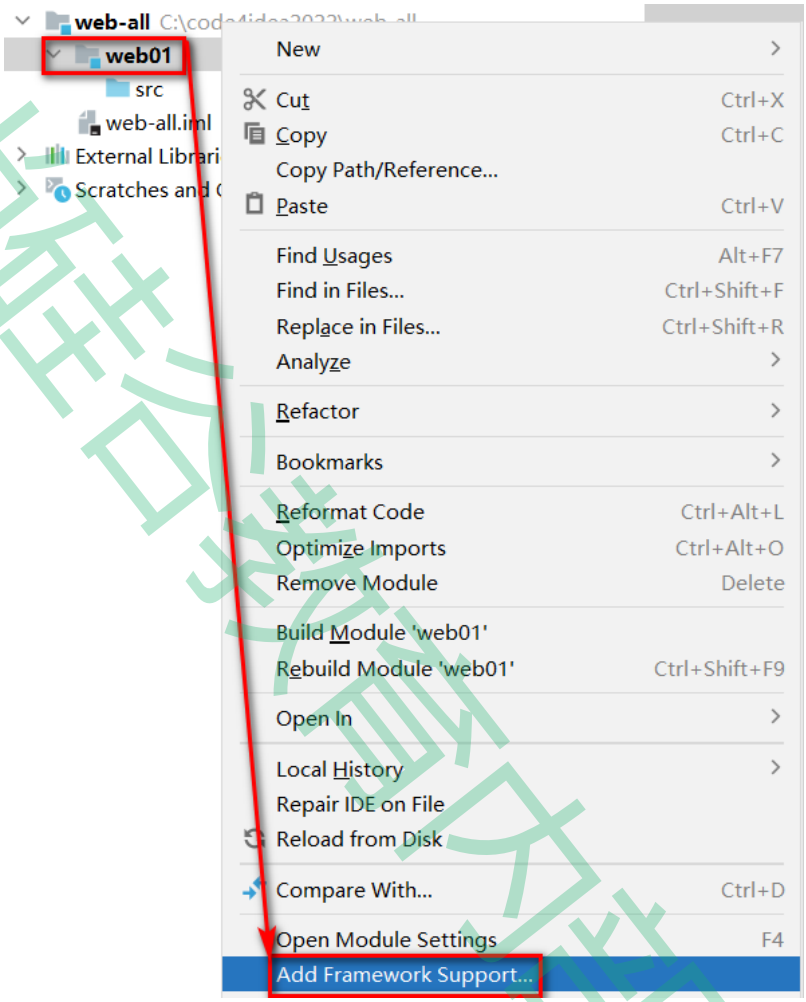
创建完毕后，为项目添加Tomcat依赖：



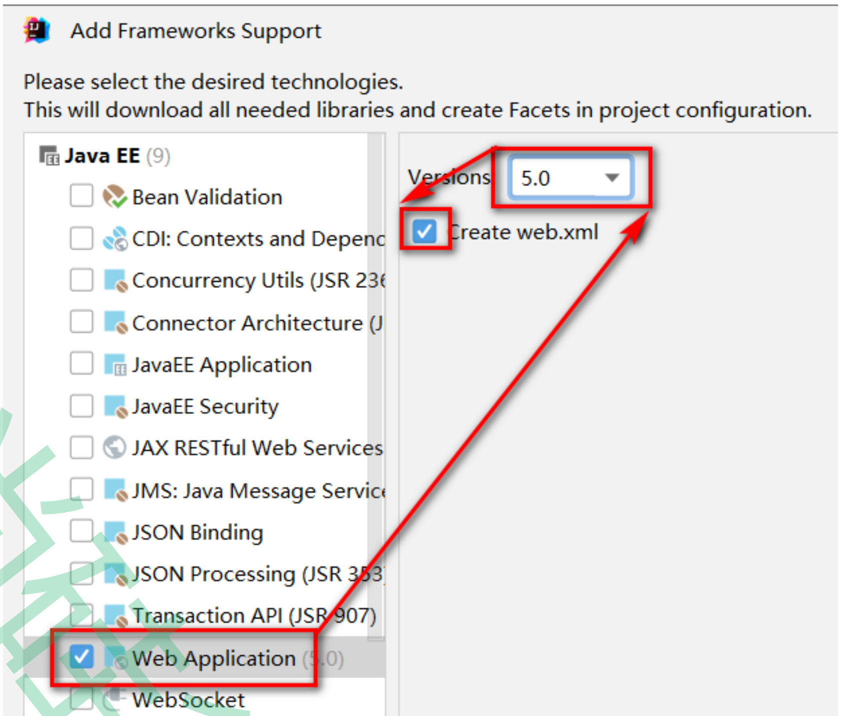




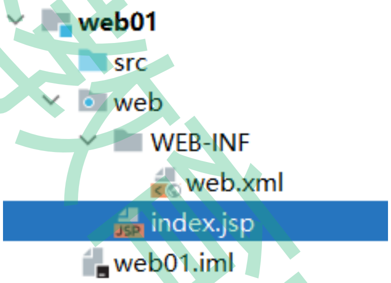
选择modules，添加 framework support：

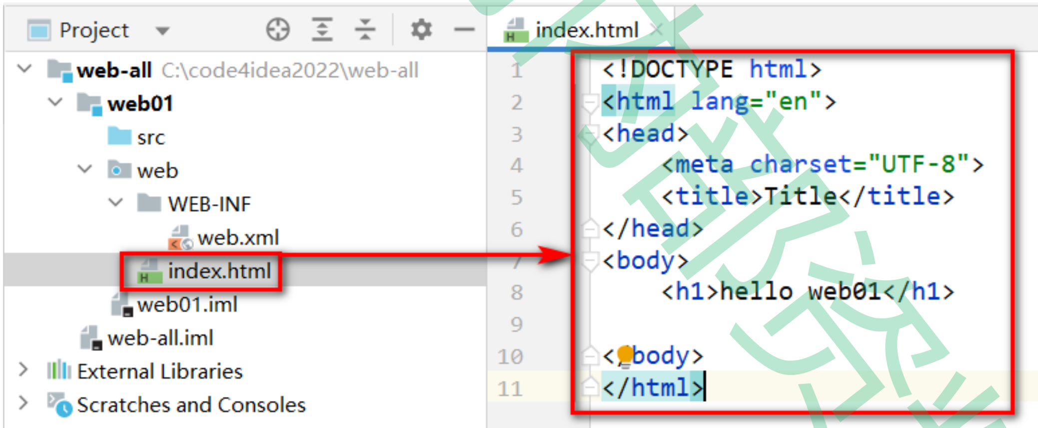


选择Web Application 注意Version，勾选 Create web.xml：



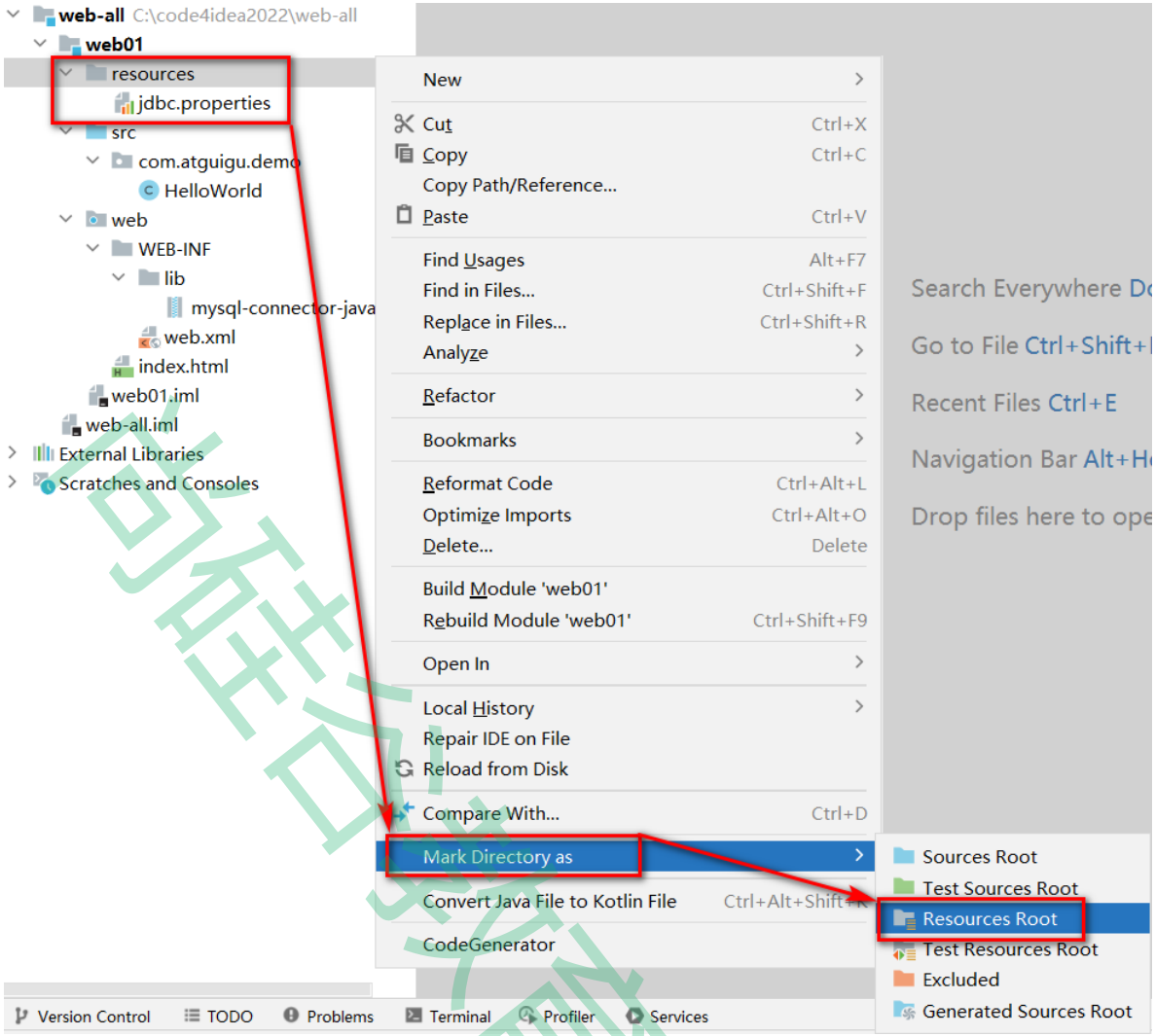
删除index.jsp， 替换为 index.html：



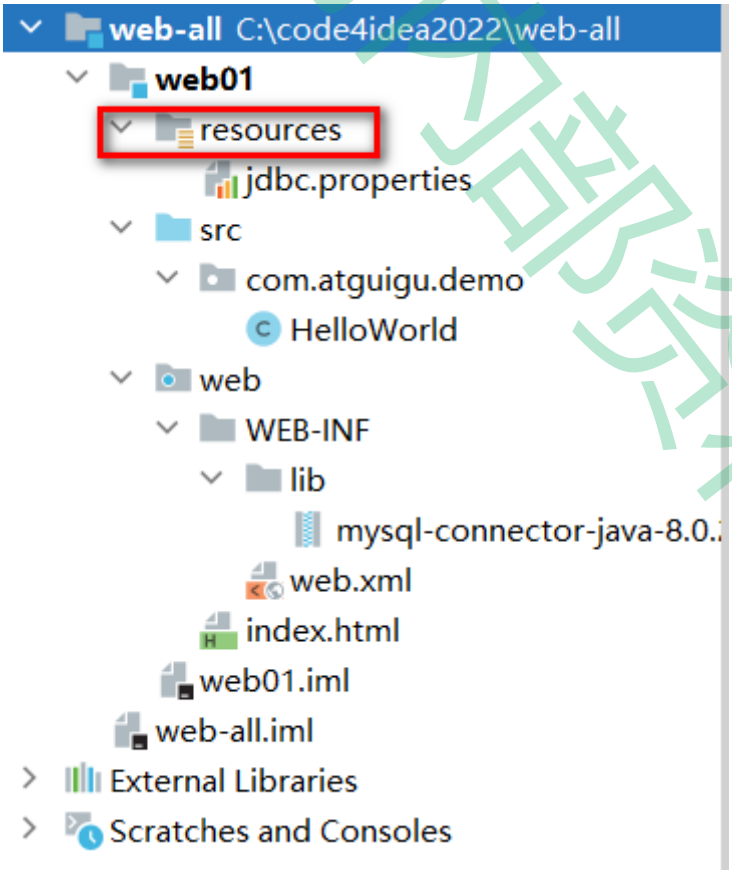


处理配置文件：

- 在工程下创建resources目录，专门用于存放配置文件(都放在src下也行,单独存放可以尽量避免文件集中存放造成的混乱)；
- 标记目录为资源目录，不标记的话则该目录不参与编译；

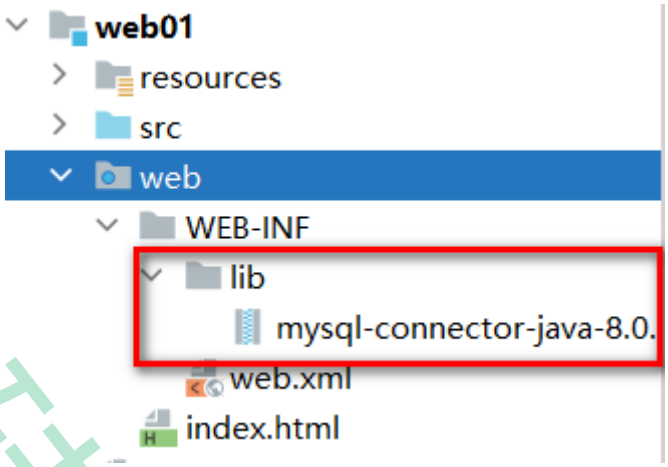


- 标记完成后,显示效果如下:

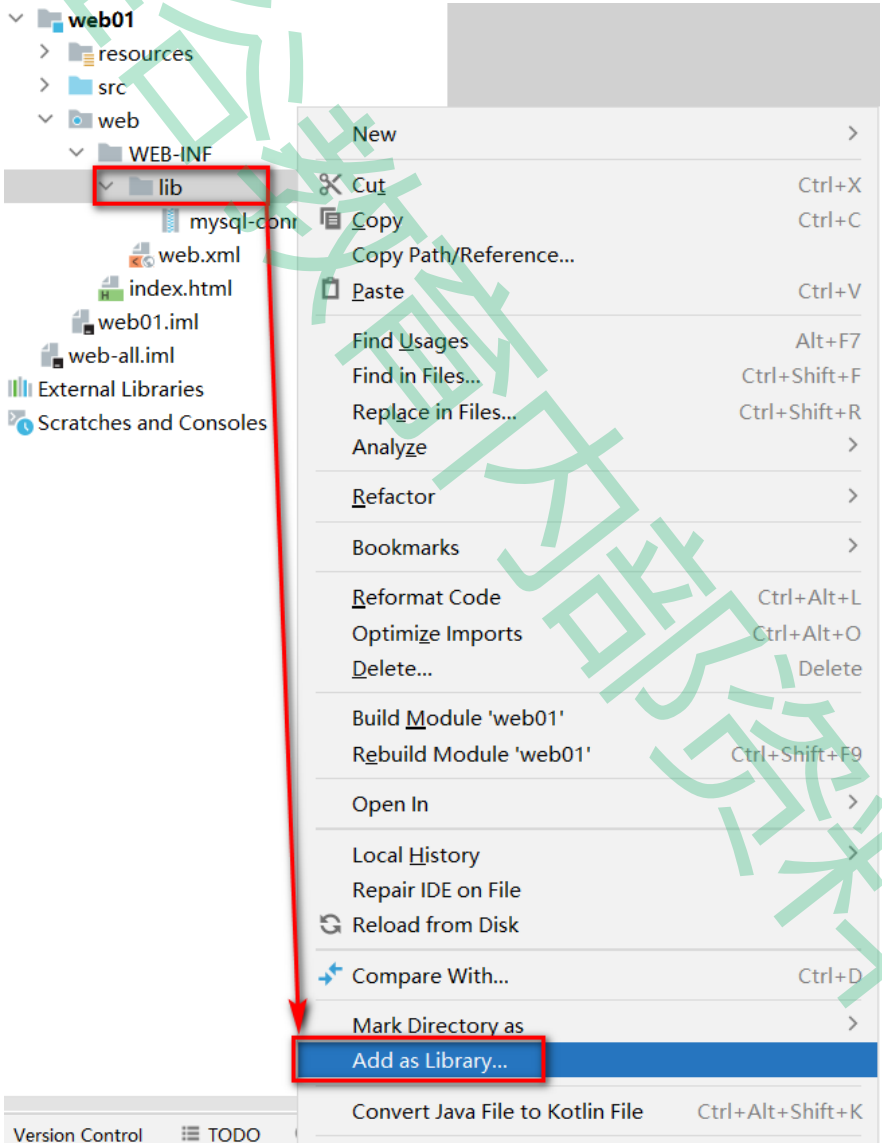


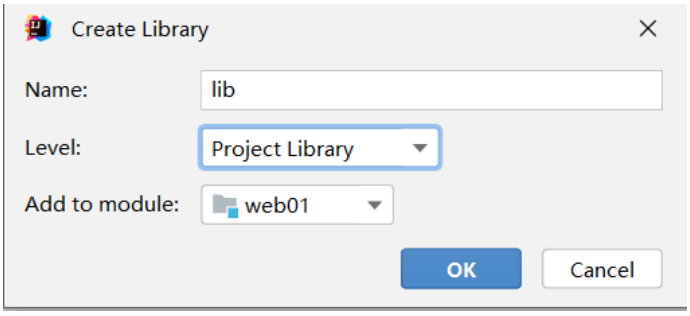
处理依赖jar包问题:

- 在WEB-INF下创建lib目录。
- 必须在WEB-INF下，且目录名必须叫lib！！！
- 复制jar文件进入lib目录。

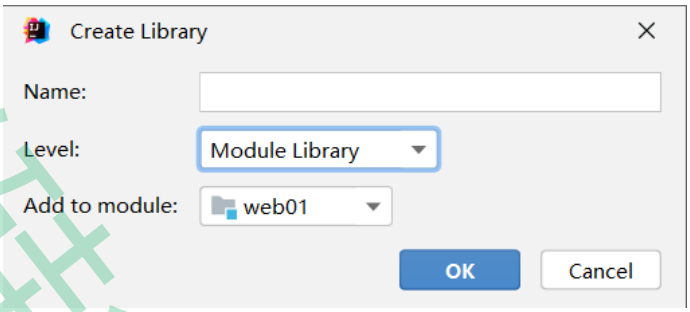


- 将lib目录添加为当前项目的依赖，后续可以用maven统一解决。

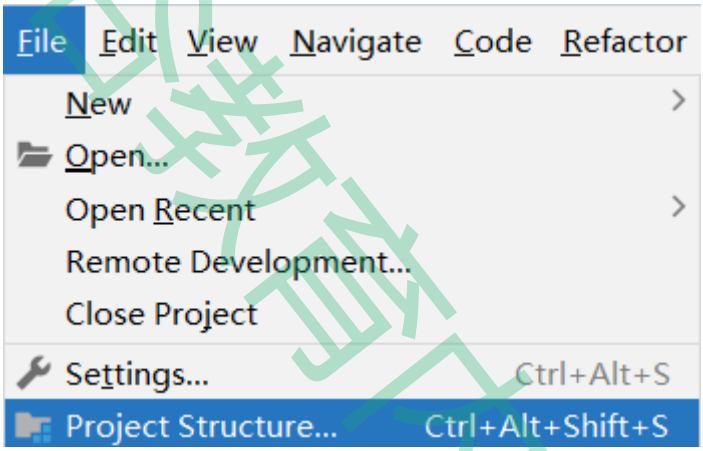


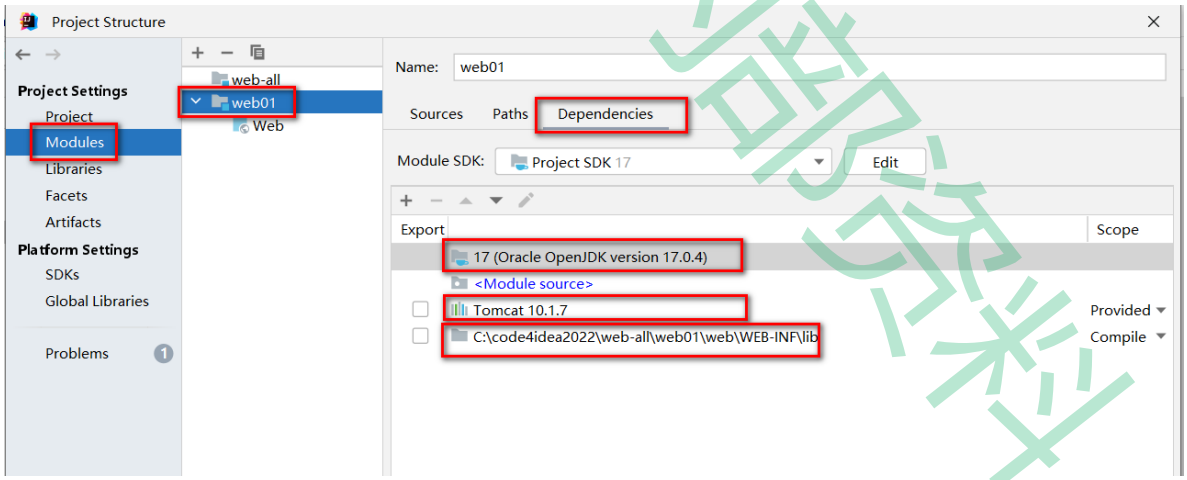


- 环境级别推荐选择module 级别，降低对其他项目的影响，name可以空着不写。

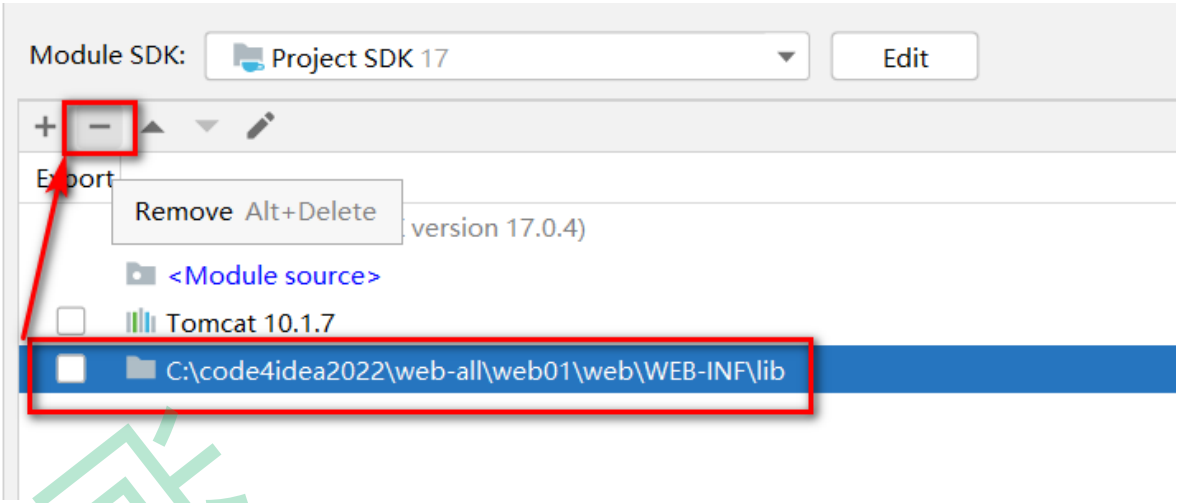


- 查看当前项目有那些环境依赖。





- 在此位置，可以通过-号解除依赖。

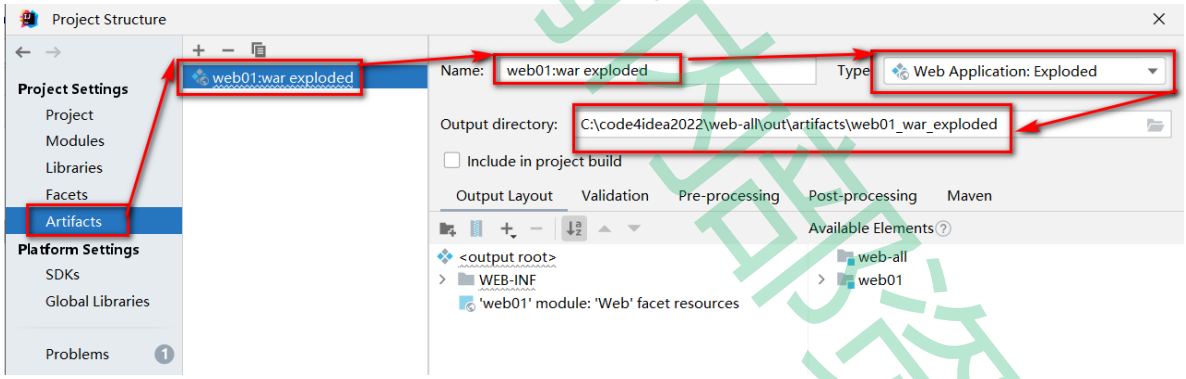


2.6.3 IDEA部署-运行web项目

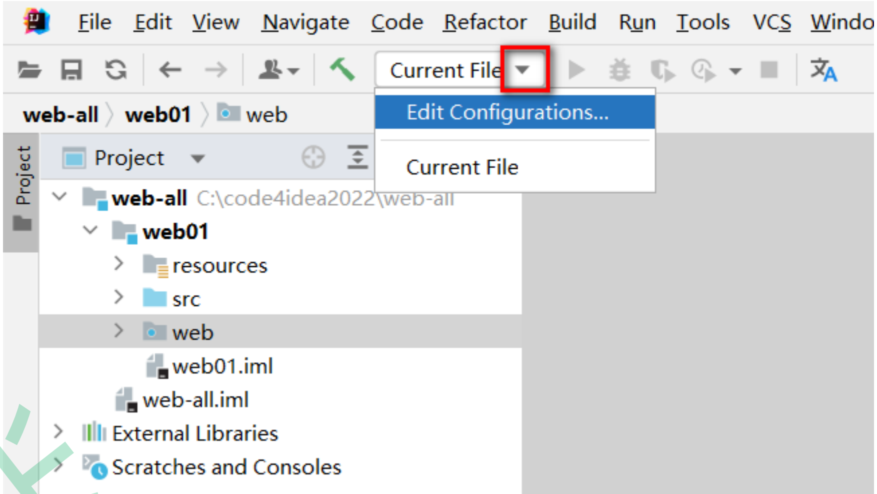
> 检查idea是否识别modules为web项目，并且存在将项目构建成发布结构的配置：

- 就是检查工程目录下，web目录有没有特殊的识别标记。

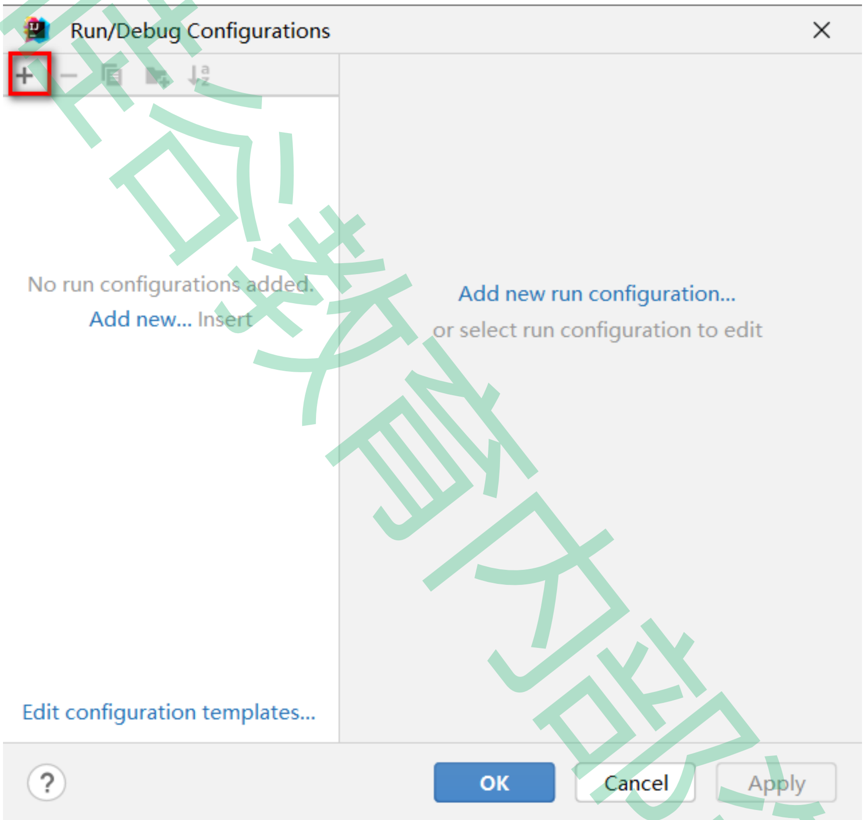
- 以及artifacts下，有没有对应 _war_exploded，如果没有，就点击+号添加。



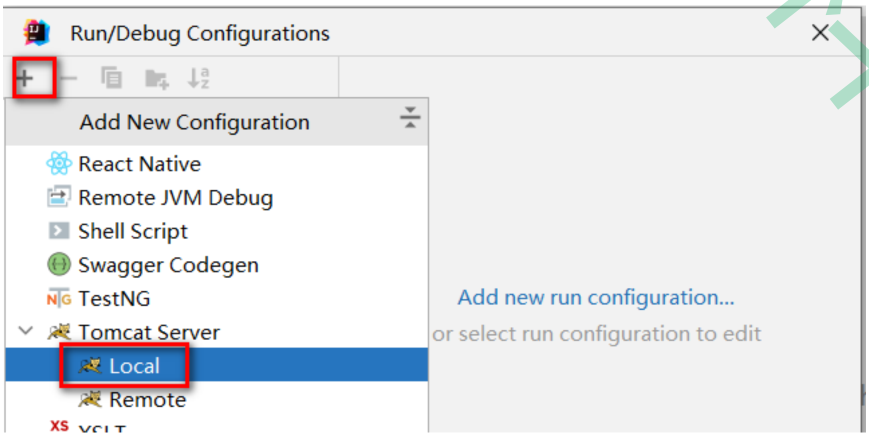
> 点击向下箭头，出现 Edit Configurations选项：



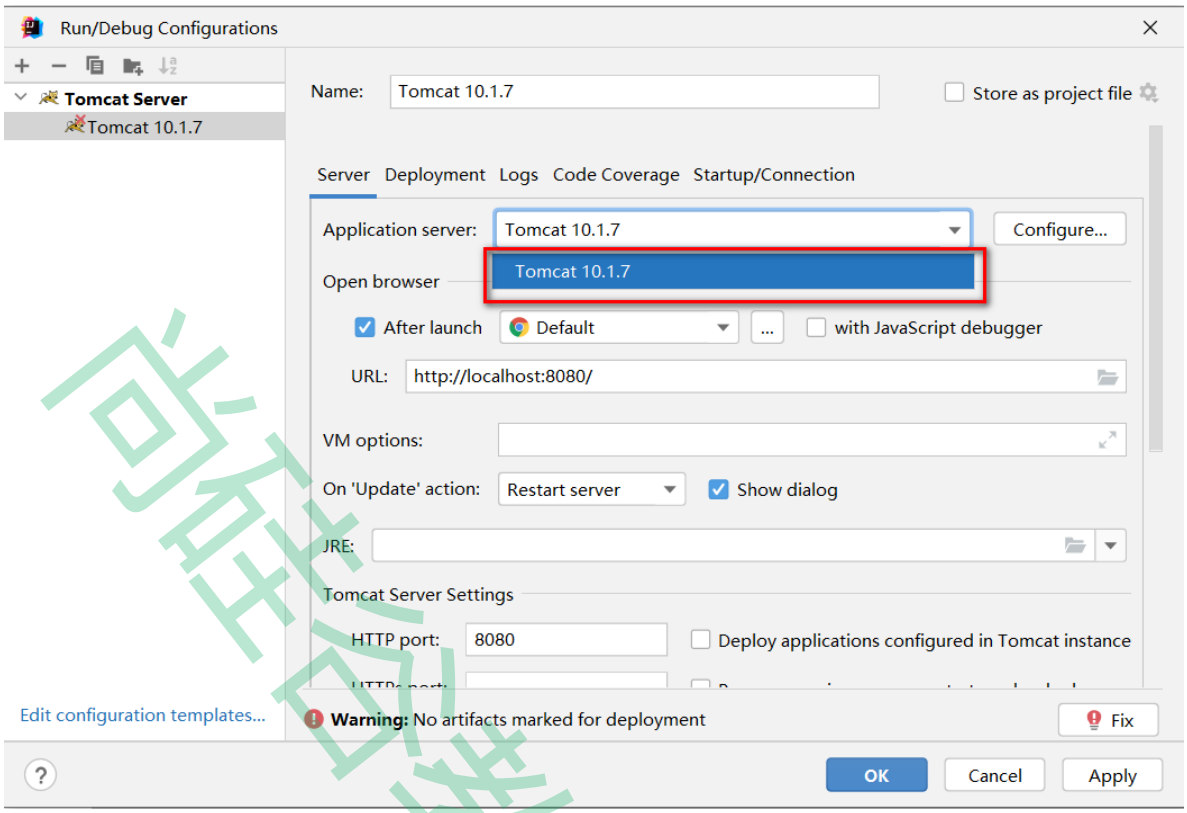
出现运行配置界面：



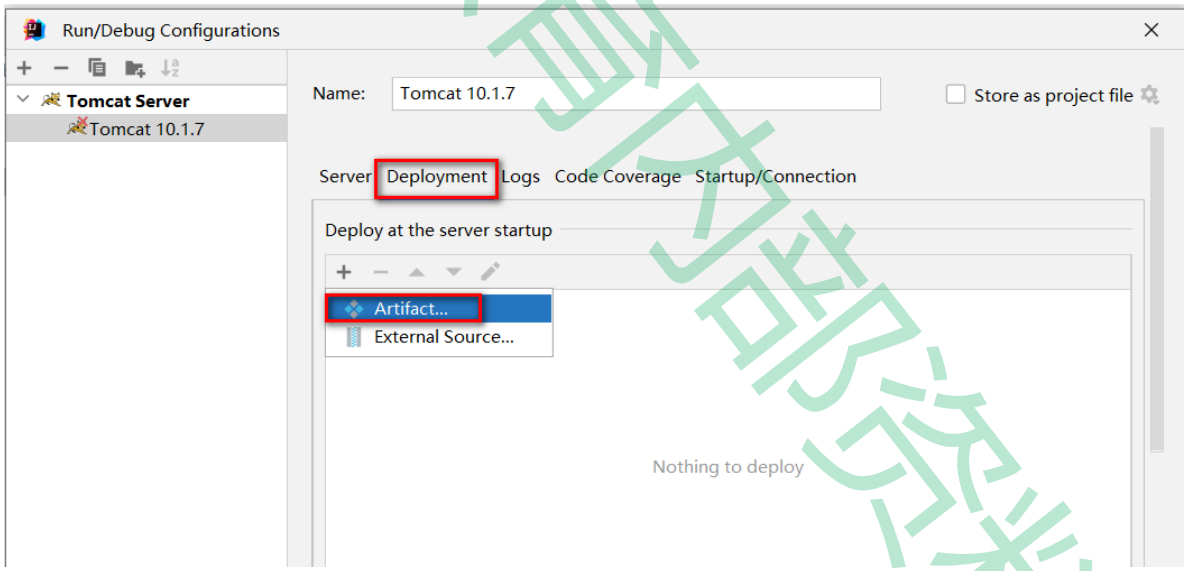
点击+号，添加本地Tomcat服务器：



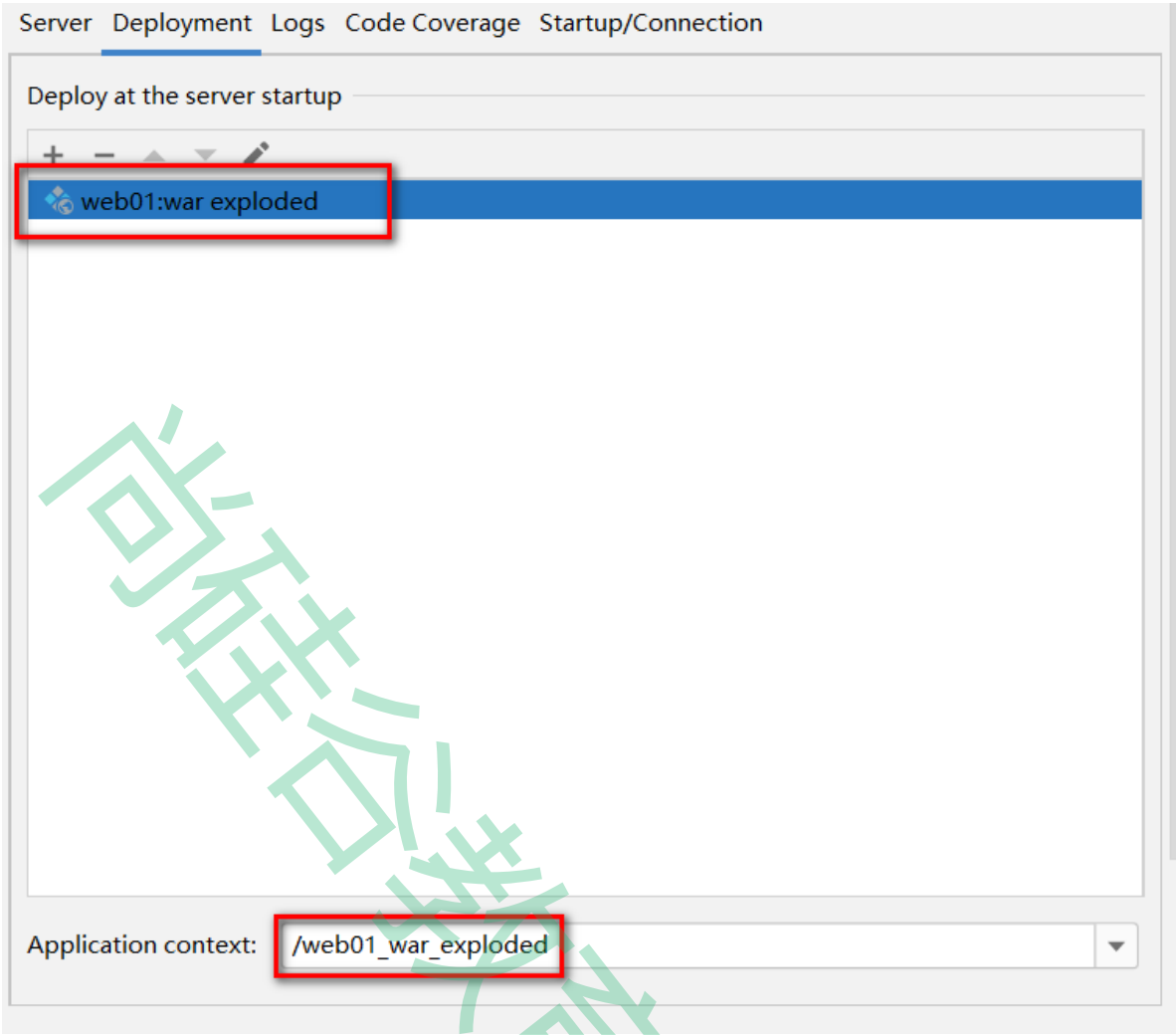
因为IDEA 只关联了一个Tomcat，红色部分就只有一个Tomcat可选：



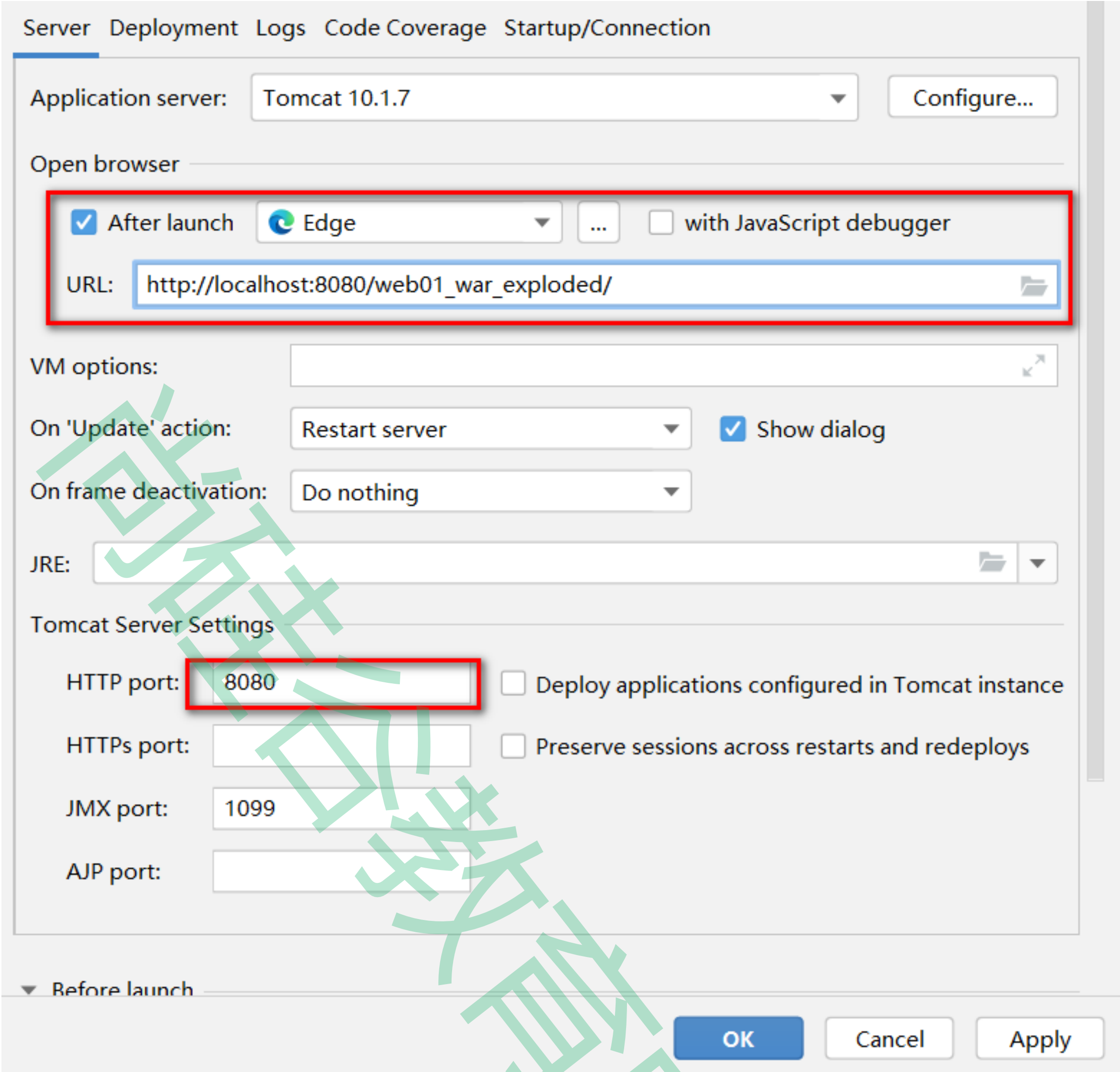
选择Deployment，通过+添加要部署到Tomcat中的artifact：



Application context中是默认的项目上下文路径，也就是url中需要输入的路径，这里可以自己定义，可以和工程名称不一样，也可以不写，但是要保留/，这里暂时就用默认的：

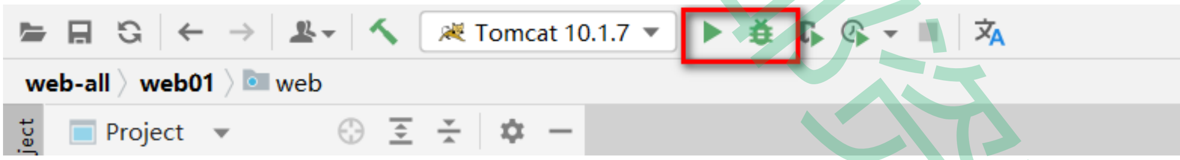


点击apply 应用后，回到Server部分。After Launch是配置启动成功后，是否默认自动打开浏览器并输入URL中的地址，HTTP port是HTTP连接器目前占用的端口号：

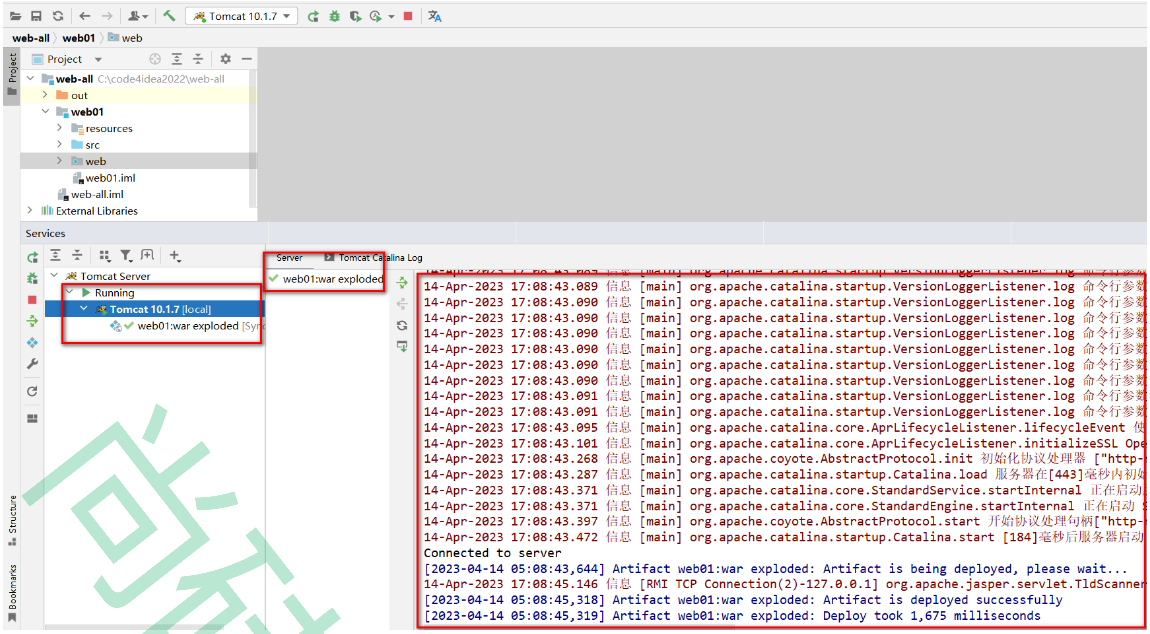


点击OK后，启动项目，访问测试：

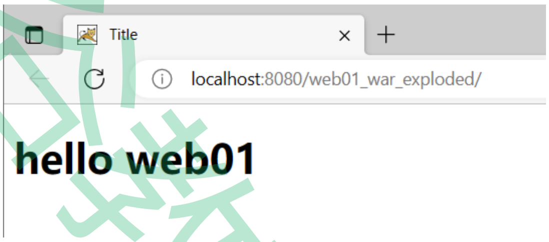
- 绿色箭头是正常运行模式。
- "小虫子"是debug运行模式。



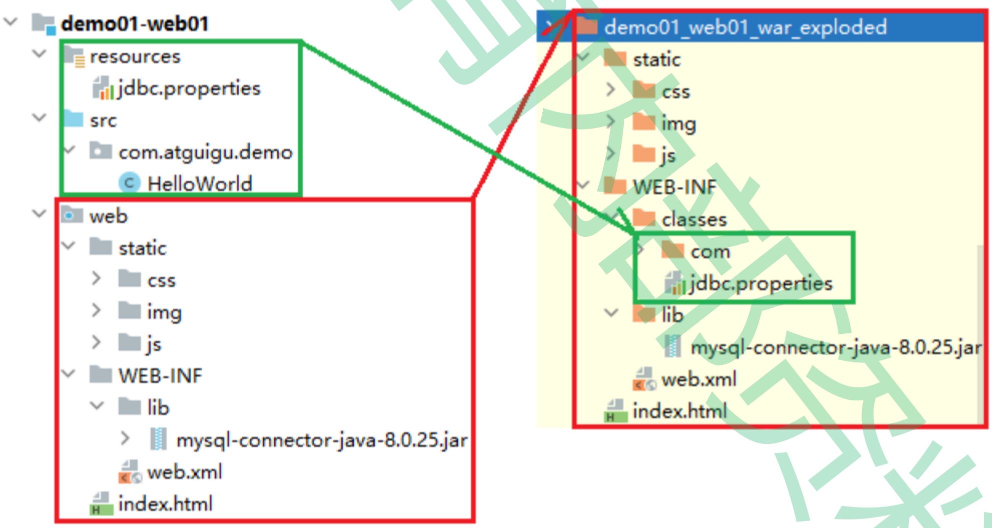
- 点击后，查看日志状态是否有异常。



- 浏览器自动打开并自动访问了index.html欢迎页。

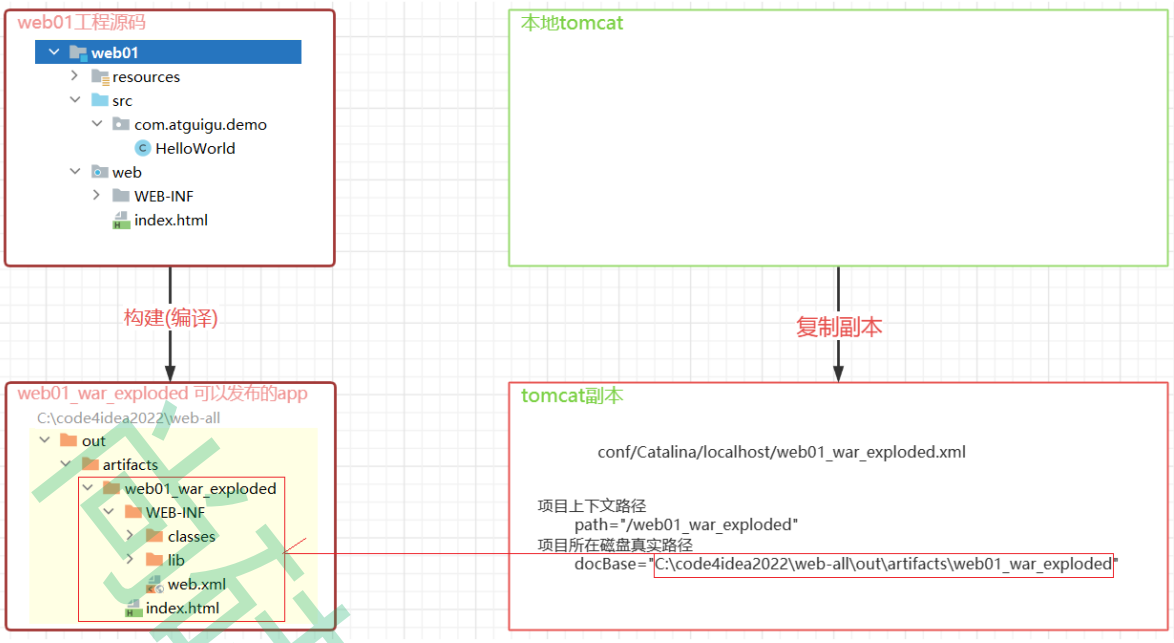


> 工程结构和可以发布的项目结构之间的目录对应关系：



> IDEA部署并运行项目的原理：

- idea并没有直接进将编译好的项目放入Tomcat的webapps中；
- idea根据关联的Tomcat，创建了一个Tomcat副本，将项目部署到了这个副本中；
- idea的Tomcat副本在C:\用户\当前用户\AppData\Local\JetBrains\IntelliJIdea2022.2\tomcat\中；
- idea的Tomcat副本并不是一个完整的tomcat，副本里只是准备了和当前项目相关的配置文件而已；
- idea启动Tomcat时，是让本地Tomcat程序按照Tomcat副本里的配置文件运行；
- idea的Tomcat副本部署项目的模式是通过conf/Catalina/localhost/*.xml配置文件的形式实现项目部署的；



三 HTTP协议

3.1 HTTP简介

> **HTTP** 超文本传输协议 (HTTP-Hyper Text transfer protocol)，是一个属于应用层的面向对象的协议，由于其简捷、快速的方式，适用于分布式超媒体信息系统。它于1990年提出，经过十几年的使用与发展，得到不断地完善和扩展。 它是一种详细规定了浏览器和万维网服务器之间互相通信的规则 ，通过因特网传送万维网文档的数据传送协议。客户端与服务端通信时传输的内容我们称之为 报文 。 HTTP协议规定了报文的格式。 HTTP就是一个通信规则，这个规则规定了客户端发送给服务器的报文格式，也规定了服务器发送给客户端的报文格式。实际我们要学习的就是这两种报文。 客户端发送给服务器的称为"请求报文" ， 服务器发送给客户端的称为"响应报文" 。

3.1.1 发展历程

> HTTP/0.9：

- 蒂姆伯纳斯李是一位英国计算机科学家，也是万维网的发明者。他在 1989 年创建了单行 HTTP 协议。它只是返回一个网页。这个协议在 1991 年被命名为 HTTP/0.9。

> HTTP/1.0：

- 1996 年，HTTP/1.0 发布。该规范是显著扩大，并且支持三种请求方式：GET，HEAD，和POST。
- HTTP/1.0 相对于 HTTP/0.9 的改进如下：
  - 每个请求都附加了 HTTP 版本；
  - 在响应开始时发送状态代码；
  - 请求和响应都包含 HTTP 报文头；
  - 内容类型能够传输 HTML 文件以外的文档；
- 但是，HTTP/1.0 不是官方标准。

> HTTP/1.1：

- HTTP 的第一个标准化版本 HTTP/1.1 (RFC 2068) 于 1997 年初发布，支持七种请求方法：OPTIONS，GET，HEAD，POST，PUT，DELETE，和TRACE 。
- HTTP/1.1 是 HTTP 1.0 的增强：
  - 虚拟主机允许从单个 IP 地址提供多个域；
  - 持久连接和流水线连接允许 Web 浏览器通过单个持久连接发送多个请求；
  - 缓存支持节省了带宽并使响应速度更快；
- HTTP/1.1 在接下来的 15 年左右将非常稳定。
- 在此期间，出现了 HTTPS（安全超文本传输协议）。它是使用 SSL/TLS 进行安全加密通信的 HTTP 的安全版本。

> HTTP/2：

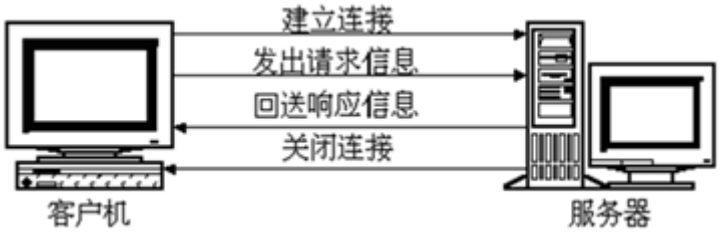
- 由IETF在2015年发布。HTTP/2旨在提高Web性能，减少延迟，增加安全性，使Web应用更加快速、高效和可靠。
- 多路复用：HTTP/2 允许同时发送多个请求和响应，而不是像 HTTP/1.1 一样只能一个一个地处理。这样可以减少延迟，提高效率，提高网络吞吐量。
- 二进制传输：HTTP/2 使用二进制协议，与 HTTP/1.1 使用的文本协议不同。二进制协议可以更快地解析，更有效地传输数据，减少了传输过程中的开销和延迟。
- 头部压缩：HTTP/2 使用 HPACK 算法对 HTTP 头部进行压缩，减少了头部传输的数据量，从而减少了网络延迟。
- 服务器推送：HTTP/2 支持服务器推送，允许服务器在客户端请求之前推送资源，以提高性能。
- 改进的安全性：HTTP/2 默认使用 TLS（Transport Layer Security）加密传输数据，提高了安全性。
- 兼容 HTTP/1.1：HTTP/2 可以与 HTTP/1.1 共存，服务器可以同时支持 HTTP/1.1 和 HTTP/2。如果客户端不支持 HTTP/2，服务器可以回退到 HTTP/1.1。

> HTTP/3：

- 于 2021 年 5 月 27 日发布 , HTTP/3 是一种新的、快速、可靠且安全的协议，适用于所有形式的设备。 HTTP/3 没有使用 TCP，而是使用谷歌在 2012 年开发的新协议 QUIC 。
- HTTP/3 是继 HTTP/1.1 和 HTTP/2之后的第三次重大修订。
- HTTP/3 带来了革命性的变化，以提高 Web 性能和安全性。设置 HTTP/3 网站需要服务器和浏览器支持。
- 目前，谷歌云、Cloudflare和Fastly支持 HTTP/3。Chrome、Firefox、Edge、Opera 和一些移动浏览器支持 HTTP/3。

3.1.2 HTTP协议的会话方式

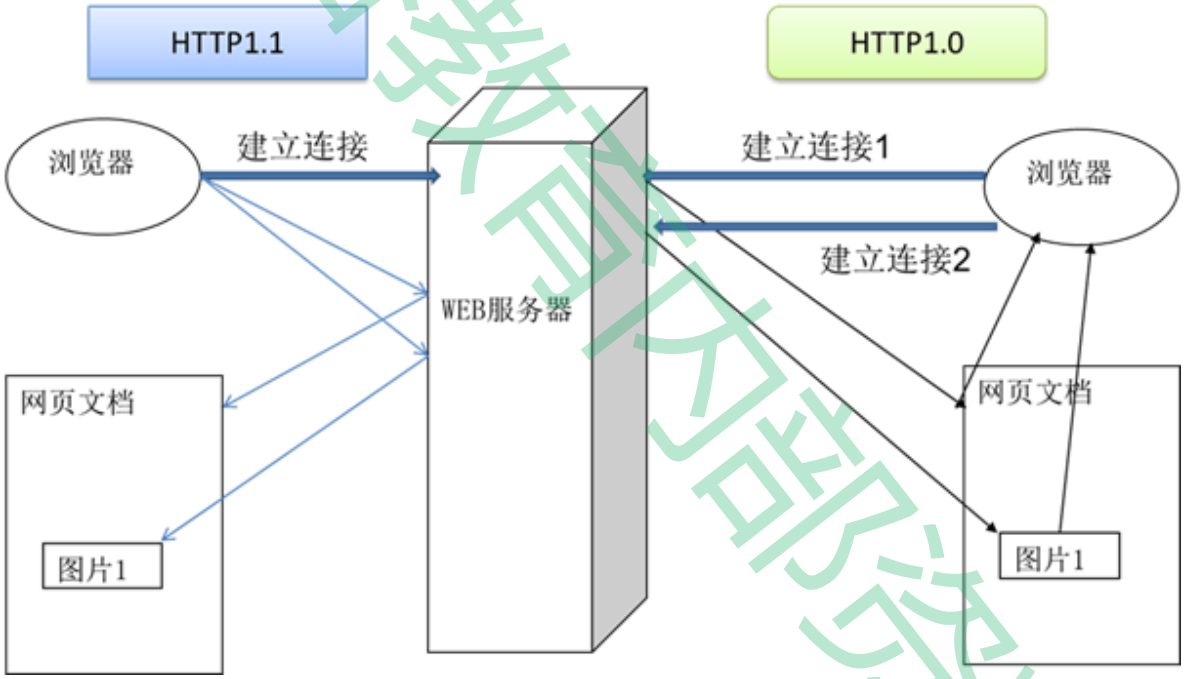
浏览器与服务器之间的通信过程要经历四个步骤。



- 浏览器与WEB服务器的连接过程是短暂的，每次连接只处理一个请求和响应。对每一个页面的访问，浏览器与WEB服务器都要建立一次单独的连接。
- 浏览器到WEB服务器之间的所有通讯都是完全独立分开的请求和响应对。

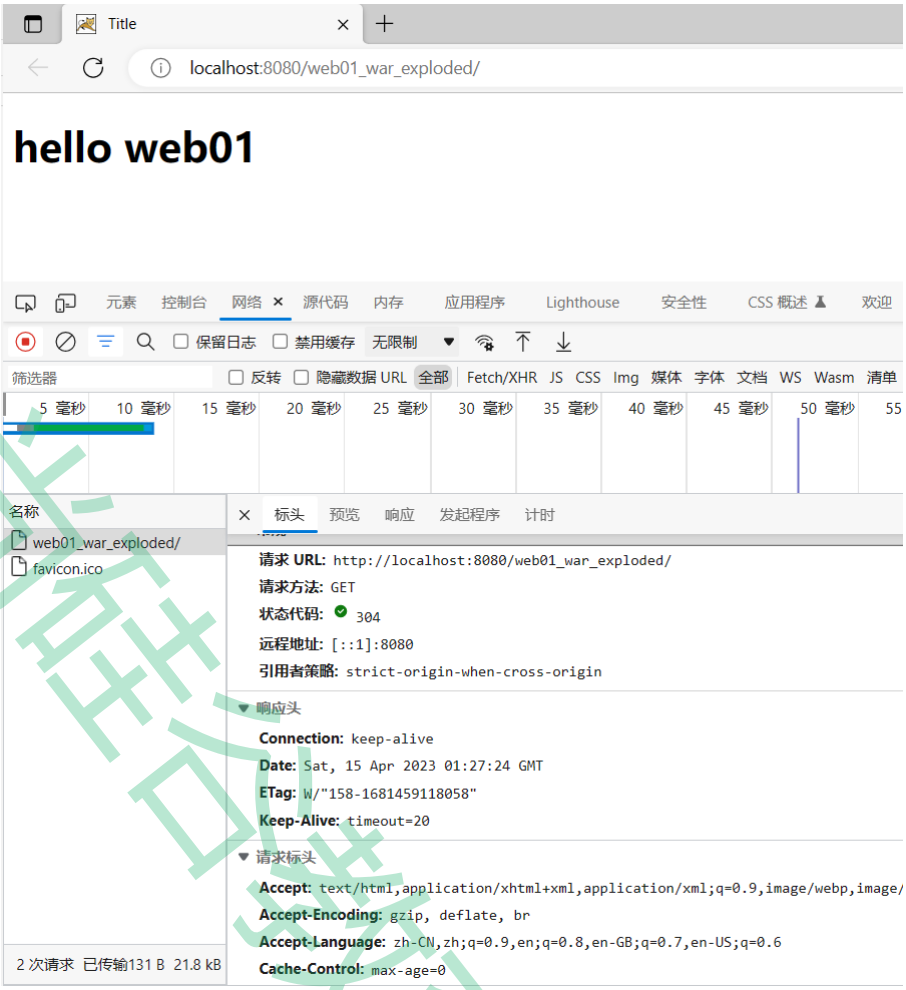
3.1.3 HTTP1.0和HTTP1.1的区别

在HTTP1.0版本中，浏览器请求一个带有图片的网页，会由于下载图片而与服务器之间开启一个新的连接；但在HTTP1.1版本中，允许浏览器在拿到当前请求对应的全部资源后再断开连接，提高了效率。



3.1.4 在浏览器中通过F12抓取请求响应报文包

几乎所有的PC端浏览器都支持了F12开发者工具，只不过不同的浏览器工具显示的窗口有差异。

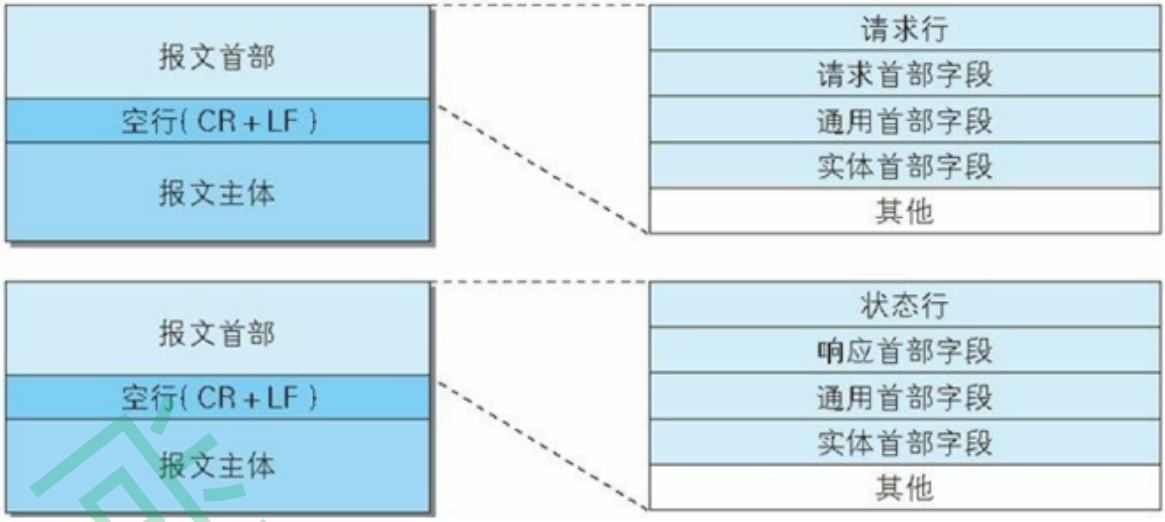


3.2 请求和响应报文

3.2.1 报文的格式

> 主体上分为报文首部和报文主体，中间空行隔开：

> 报文部首可以继续细分为 "行" 和 "头"：



3.2.2 请求报文

> 客户端发给服务端的报文：

- 请求报文格式
  - 请求首行（**请求行**）； GET/POST 资源路径?参数 HTTP/1.1
  - 请求头信息（**请求头**）；
  - 空行；
  - **请求体**；

> 请求行 ： 请求方式 、资源路径 、协议及版本。

```
GET /05_web_tomcat/login_success.html?username=admin&password=123213 HTTP/1.1
```

> 请求头：

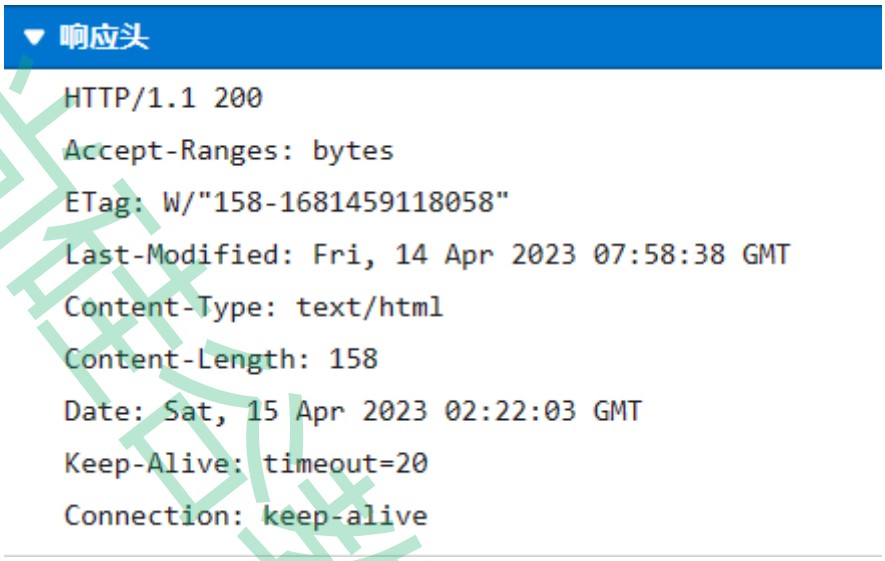
```
-主机虚拟地址
Host: localhost:8080
-长连接
Connection: keep-alive
-请求协议的自动升级[http的请求，服务器却是https的，浏览器自动会将请求协议升级为https的]
Upgrade-Insecure-Requests: 1
- 用户系统信息
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/68.0.3440.75 Safari/537.36
- 浏览器支持的文件类型
Accept:text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*
;q=0.8
- 当前页面的上一个页面的路径[当前页面通过哪个页面跳转过来的]：  可以通过此路径跳转回上一个页面， 广
告计费，防止盗链
Referer: http://localhost:8080/05_web_tomcat/login.html
- 浏览器支持的压缩格式
Accept-Encoding: gzip, deflate, br
- 浏览器支持的语言
Accept-Language: zh-CN,zh;q=0.9,en-US;q=0.8,en;q=0.7
```

> 请求体：使用form表单发送POST请求，请求体中才有数据。

3.2.3 响应报文

> 响应报文格式：

- 响应首行（**响应行**）； 协议/版本 状态码 状态码描述
- 响应头信息（**响应头**）；
- 空行；
- **响应体**；



> 响应行：协议及版本、响应状态码、状态描述。

```
HTTP/1.1 200 OK (缺省)
```

> 响应头：

```
Server: Apache-Coyote/1.1              服务器的版本信息
Accept-Ranges: bytes
ETag: W/"157-1534126125811"
Last-Modified: Mon, 13 Aug 2018 02:08:45 GMT
Content-Type: text/html                响应体数据的类型[浏览器根据类型解析响应体数据]
Content-Length: 157                    响应体内容的字节数
Date: Mon, 13 Aug 2018 02:47:57 GMT    响应的时间，这可能会有8小时的时区差
```

> 响应体：

```
<!--需要浏览器解析使用的内容[如果响应的是html页面，最终响应体内容会被浏览器显示到页面中]-->
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Insert title here</title>
  </head>
  <body>
    恭喜你，登录成功了...
  </body>
</html>
```

> 响应状态码：响应码对浏览器来说很重要，它告诉浏览器响应的结果。比较有代表性的响应码如下：

- **200：** 请求成功，浏览器会把响应体内容（通常是html）显示在浏览器中；
- **302：** 重定向，当响应码为302时，表示服务器要求浏览器重新再发一个请求，服务器会发送一个响应头Location指定新请求的URL地址；
- **304：** 使用了本地缓存；
- **404：** 请求的资源没有找到，说明客户端错误的请求了不存在的资源；
- **405：** 请求的方式不允许；
- **500：** 请求资源找到了，但服务器内部出现了错误；

> 更多的响应状态码：

状态码	英文描述	中文含义
1**		
100	Continue	继续。客户端应继续其请求
101	Switching Protocols	切换协议。服务器根据客户端的请求切换协议。只能切换到更高级的协议，例如，切换到HTTP的新版本协议
2**		
200	OK	请求成功。一般用于GET与POST请求
201	Created	已创建。成功请求并创建了新的资源
202	Accepted	已接受。已经接受请求，但未处理完成
203	Non-Authoritative Information	非授权信息。请求成功。但返回的meta信息不在原始的服务器，而是一个副本
204	No Content	无内容。服务器成功处理，但未返回内容。在未更新网页的情况下，可确保浏览器继续显示当前文档
205	Reset Content	重置内容。服务器处理成功，用户终端（例如：浏览器）应重置文档视图。可通过此返回码清除浏览器的表单域
206	Partial Content	部分内容。服务器成功处理了部分GET请求
3**		
300	Multiple Choices	多种选择。请求的资源可包括多个位置，相应可返回一个资源特征与地址的列表用于用户终端（例如：浏览器）选择
301	Moved Permanently	永久移动。请求的资源已被永久的移动到新URI，返回信息会包括新的URI，浏览器会自动定向到新URI。今后任何新的请求都应使用新的URI代替
302	Found	临时移动。与301类似。但资源只是临时被移动。客户端应继续使用原有URI
303	See Other	查看其它地址。与301类似。使用GET和POST请求查看
304	Not Modified	未修改。所请求的资源未修改，服务器返回此状态码时，不会返回任何资源。客户端通常会缓存访问过的资源，通过提供一个头信息指出客户端希望只返回在指定日期之后修改的资源
305	Use Proxy	使用代理。所请求的资源必须通过代理访问
306	Unused	已经被废弃的HTTP状态码
307	Temporary Redirect	临时重定向。与302类似。使用GET请求重定向
4**		

状态码	英文描述	中文含义
400	Bad Request	客户端请求的语法错误，服务器无法理解
401	Unauthorized	请求要求用户的身份认证
402	Payment Required	保留，将来使用
403	Forbidden	服务器理解请求客户端的请求，但是拒绝执行此请求
404	Not Found	服务器无法根据客户端的请求找到资源（网页）。通过此代码，网站设计人员可设置"您所请求的资源无法找到"的个性页面
405	Method Not Allowed	客户端请求中的方法被禁止
406	Not Acceptable	服务器无法根据客户端请求的内容特性完成请求
407	Proxy Authentication Required	请求要求代理的身份认证，与401类似，但请求者应当使用代理进行授权
408	Request Time-out	服务器等待客户端发送的请求时间过长，超时
409	Conflict	服务器完成客户端的 PUT 请求时可能返回此代码，服务器处理请求时发生了冲突
410	Gone	客户端请求的资源已经不存在。410不同于404，如果资源以前有现在被永久删除了可使用410代码，网站设计人员可通过301代码指定资源的新位置
411	Length Required	服务器无法处理客户端发送的不带Content-Length的请求信息
412	Precondition Failed	客户端请求信息的先决条件错误
413	Request Entity Too Large	由于请求的实体过大，服务器无法处理，因此拒绝请求。为防止客户端的连续请求，服务器可能会关闭连接。如果只是服务器暂时无法处理，则会包含一个Retry-After的响应信息
414	Request-URI Too Large	请求的URI过长（URI通常为网址），服务器无法处理
415	Unsupported Media Type	服务器无法处理请求附带的媒体格式
416	Requested range not satisfiable	客户端请求的范围无效

状态码	英文描述	中文含义
417	Expectation Failed	服务器无法满足Expect的请求头信息
5**		
500	Internal Server Error	服务器内部错误，无法完成请求
501	Not Implemented	服务器不支持请求的功能，无法完成请求
502	Bad Gateway	作为网关或者代理工作的服务器尝试执行请求时，从远程服务器接收到了一个无效的响应
503	Service Unavailable	由于超载或系统维护，服务器暂时的无法处理客户端的请求。延时的长度可包含在服务器的Retry-After头信息中
504	Gateway Time-out	充当网关或代理的服务器，未及时从远端服务器获取请求
505	HTTP Version not supported	服务器不支持请求的HTTP协议的版本，无法完成处理

第五章 Servlet

一 Servlet简介

1.1 动态资源和静态资源

> 静态资源：

- 无需在程序运行时通过代码运行生成的资源,在程序运行之前就写好的资源. 例如:html css js img ,音频文件和视频文件；

> 动态资源：

- 需要在程序运行时通过代码运行生成的资源,在程序运行之前无法确定的数据,运行时动态生成,例如Servlet,Thymeleaf ... ...；
- 动态资源指的不是视图上的动画效果或者是简单的人机交互效果；

> 生活举例：

- 去蛋糕店买蛋糕：
  - 直接买柜台上已经做好的：静态资源；
  - 和柜员说要求后现场制作：动态资源；

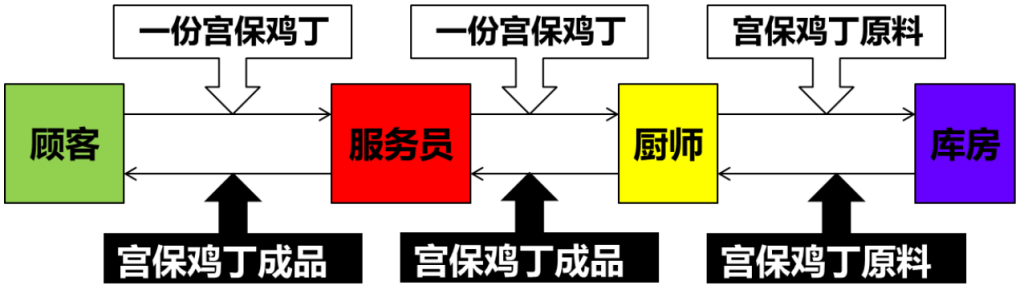
> 官方JAVAEEAPI文档下载地址： **https://www.oracle.com/java/technologies/javaee/javaeetechnologies.html#javaee8**

1.2 Servlet简介

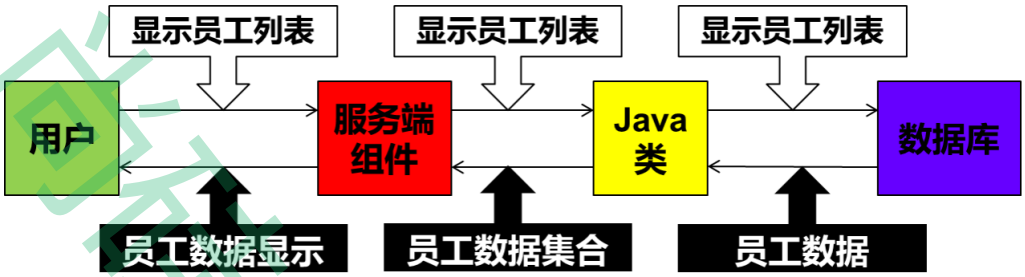
> Servlet（server applet）是运行在服务端(Tomcat)的Java小程序，是sun公司提供一套定义动态资源规范；代码层面上Servlet就是一个接口。

- 用来接收、处理客户端请求、响应给浏览器的动态资源。在整个Web应用中，Servlet主要负责接收处理请求、协同调度功能以及响应数据。我们可以把Servlet称为Web应用中的**控制器**。

类比：

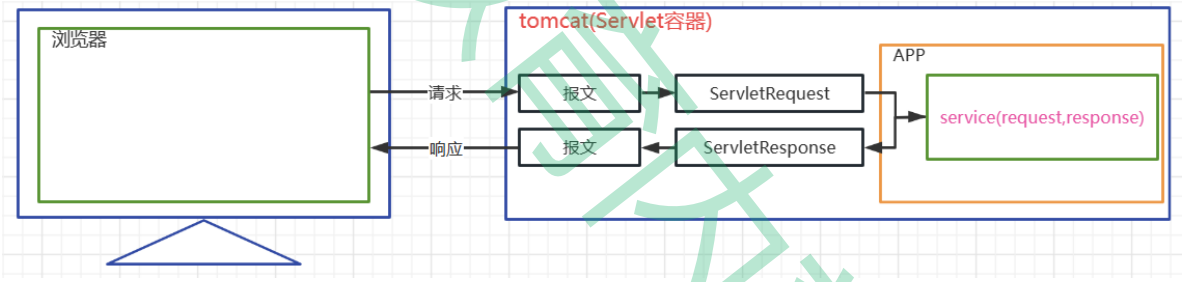


要实现的效果：



- 不是所有的JAVA类都能用于处理客户端请求，能处理客户端请求并做出响应的一套技术标准就是Servlet。
- Servlet是运行在服务端的，所以Servlet必须在WEB项目中开发且在Tomcat这样的服务容器中运行。

> 请求响应与HttpServletRequest和HttpServletResponse之间的对应关系：



二 Servlet开发流程

2.1 目标

> 校验注册时，用户名是否被占用。通过客户端向一个Servlet发送请求，携带username，如果用户名是'atguigu'，则向客户端响应 NO，如果是其他，响应YES。

2.2 开发过程

> 步骤1 开发一个web类型的module：

- 过程参照之前。

> 步骤2 开发一个UserServlet：

```
public class UserServlet  extends HttpServlet {
    @Override
    protected void service(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        // 获取请求中的键值对参数   key=value
        String username = req.getParameter("username");
        if("atguigu".equals(username)){
            //通过响应对象响应信息
            resp.getWriter().write("NO");
        }else{
            resp.getWriter().write("YES");
        }

    }
}
```

- 自定义一个类，要继承HttpServlet类；
- 重写service方法，该方法主要就是用于处理用户请求的服务方法；
- HttpServletRequest 代表请求对象，是有请求报文经过Tomcat转换而来的，通过该对象可以获取请求中的信息；
- HttpServletResponse 代表响应对象，该对象会被Tomcat转换为响应的报文，通过该对象可以设置响应中的信息；
- Servlet对象的生命周期(创建、初始化、处理服务、销毁)是由Tomcat管理的，无需自己new；
- HttpServletRequest HttpServletResponse 两个对象也是由Tomcat负责转换，在调用service方法时传入给我们用的；

> 步骤3 在web.xml为UseServlet配置请求的映射路径：

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="https://jakarta.ee/xml/ns/jakartaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="https://jakarta.ee/xml/ns/jakartaee
https://jakarta.ee/xml/ns/jakartaee/web-app_5_0.xsd"
         version="5.0">
    <servlet>
        <!--给UserServlet起一个别名-->
        <servlet-name>userServlet</servlet-name>
        <servlet-class>com.atguigu.servlet.UserServlet</servlet-class>
    </servlet>
    <servlet-mapping>
        <!--关联别名和映射路径-->
        <servlet-name>userServlet</servlet-name>
        <!--可以为一个Servlet匹配多个不同的映射路径,但是不同的Servlet不能使用相同的url-pattern-
->
        <url-pattern>/userServlet</url-pattern>
       <!-- <url-pattern>/userServlet2</url-pattern>-->
        <!--
            /         表示通配所有资源,不包括jsp文件
            /*        表示通配所有资源,包括jsp文件
            /a/*      匹配所有以a前缀的映射路径
            *.action 匹配所有以action为后缀的映射路径
        -->
       <!-- <url-pattern>/*</url-pattern>-->
    </servlet-mapping>
</web-app>
```

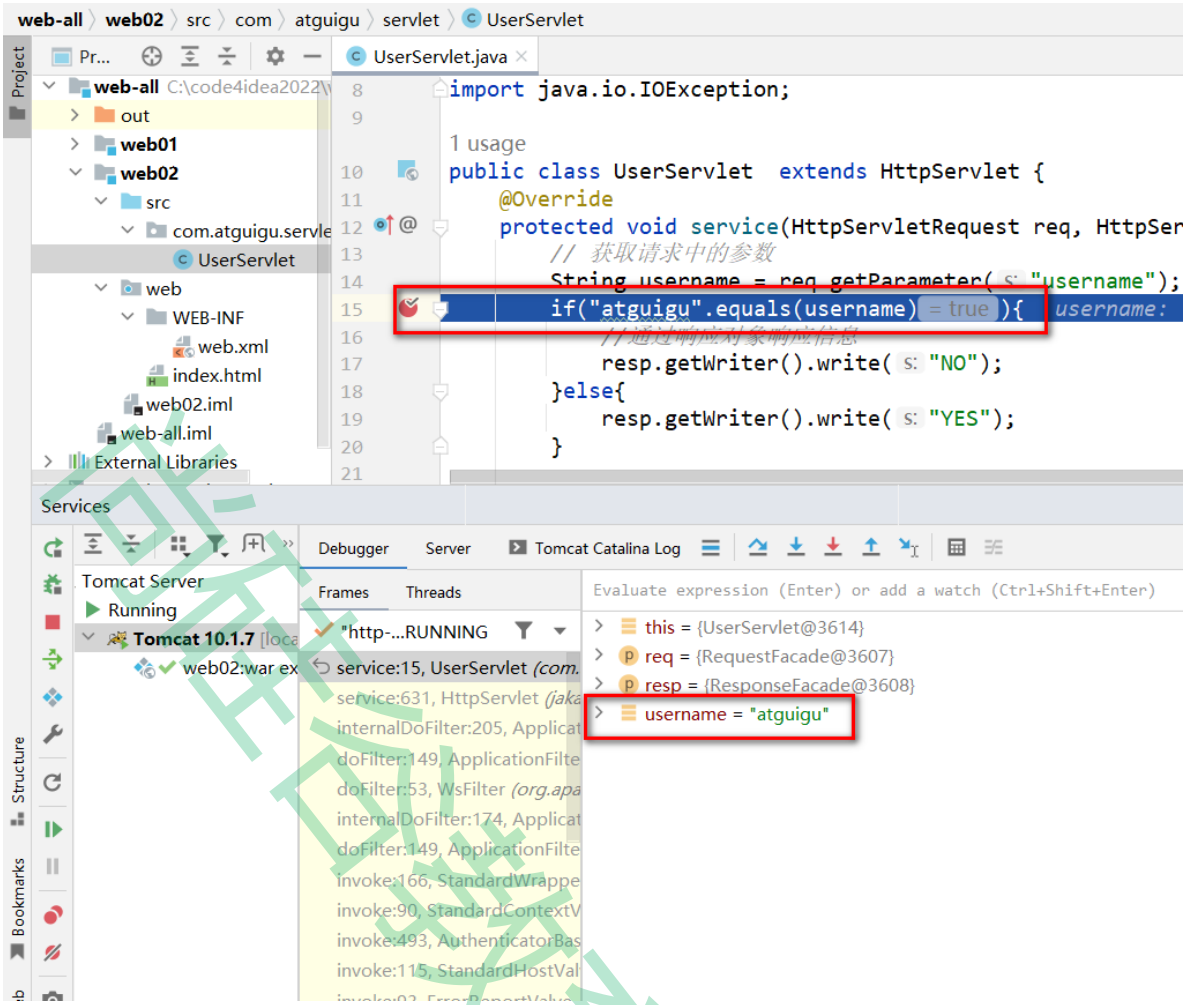
- Servlet并不是文件系统中实际存在的文件或者目录，所以为了能够请求到该资源，需要为其配置映射路径；
- Servlet的请求映射路径配置在web.xml中；
- `<servlet-name>`作为Servlet的别名，可以自己随意定义，见名知意就好；
- `<url-pattern>`标签用于定义Servlet的请求映射路径；
- 一个Servlet可以对应多个不同的`<url-pattern>`；
- 多个Servlet不能使用相同的`<url-pattern>`；
- `<url-pattern>`中可以使用一些通配写法：
  - / 表示通配所有资源，不包括jsp文件；
  - /* 表示通配所有资源，包括jsp文件；
  - /a/* 匹配所有以a前缀的映射路径；
  - *.action 匹配所有以action为后缀的映射路径；

> 步骤4 开发一个form表单，向servlet发送一个GET请求并携带username参数：

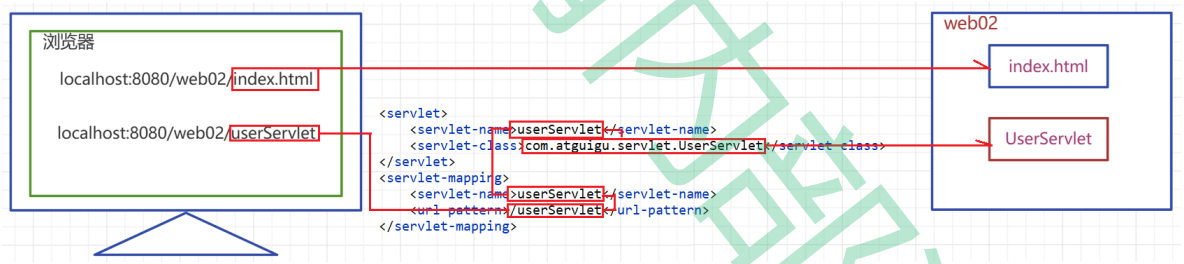
```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
    <form action="userServlet">
        请输入用户名:<input type="text" name="username" /> <br>
        <input type="submit" value="校验">
    </form>
</body>
</html>
```

> 步骤5 启动项目，访问index.html，提交表单测试：

- 使用debug模式运行测试



映射关系图：



三 Servlet注解方式配置

3.1 @WebServlet注解源码

@WebServlet注解的源码阅读：

- 源码通过idea查看，此处略。

3.2 @WebServlet注解使用

> 使用@WebServlet注解替换Servlet配置：

```
@WebServlet(
        name = "userServlet",
        //value = "/user",
        urlPatterns = {"/userServlet1","/userServlet2","/userServlet"},
        initParams = {@WebInitParam(name = "encoding",value = "UTF-8")},
        loadOnStartup = 6
)
public class UserServlet  extends HttpServlet {
    @Override
    protected void service(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        String encoding = getServletConfig().getInitParameter("encoding");
        System.out.println(encoding);
        // 获取请求中的参数
        String username = req.getParameter("username");
        if("atguigu".equals(username)){
            //通过响应对象响应信息
            resp.getWriter().write("NO");
        }else{
            resp.getWriter().write("YES");
        }
    }
}
```

四 Servlet生命周期

4.1 生命周期简介

> 什么是Servlet的生命周?

- 应用程序中的对象不仅在空间上有层次结构的关系，在时间上也会因为处于程序运行过程中的不同阶段而表现出不同状态和不同行为，这就是对象的生命周期。
- 简单的叙述生命周期，就是对象在容器中从开始创建到销毁的过程。

> Servlet容器：

- Servlet对象是Servlet容器创建的，生命周期方法都是由容器(目前我们使用的是Tomcat)调用的。这一点和我们之前所编写的代码有很大不同。在今后的学习中我们会看到，越来越多的对象交给容器或框架来创建，越来越多的方法由容器或框架来调用，开发人员要尽可能多的将精力放在业务逻辑的实现上。

> Servlet主要的生命周期执行特点：

生命周期	对应方法	执行时机	执行次数
构造对象	构造器	第一次请求或者容器启动	1
初始化	init()	构造完毕后	1
处理服务	service(HttpServletRequest req,HttpServletResponse resp)	每次请求	多次
销毁	destory()	容器关闭	1

4.2 生命周期测试

> 开发servlet代码：

```
package com.atguigu.servlet;
import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import java.io.IOException;
public class ServletLifeCycle  extends HttpServlet {
    public ServletLifeCycle(){
        System.out.println("构造器");
    }
    @Override
    public void init() throws ServletException {
        System.out.println("初始化方法");
    }
    @Override
    protected void service(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        System.out.println("service方法");
    }
    @Override
    public void destroy() {
        System.out.println("销毁方法");
    }
}
```

> 配置Servlet：

```
    <servlet>
        <servlet-name>servletLifeCycle</servlet-name>
        <servlet-class>com.atguigu.servlet.ServletLifeCycle</servlet-class>
        <!--load-on-startup 如果配置的是正整数则表示容器在启动时就要实例化Servlet，数字表示的是实例化的顺序 -->
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>servletLifeCycle</servlet-name>
        <url-pattern>/servletLiftCycle</url-pattern>
    </servlet-mapping>
```

4.3 生命周期总结

1. 通过生命周期测试发现Servlet对象在容器中是单例的；
2. 容器是可以处理并发的用户请求的，每个请求在容器中都会开启一个线程；
3. 多个线程可能会使用相同的Servlet对象，所以在Servlet中，我们不要轻易定义一些需要经常修改的成员变量；
4. load-on-startup中定义的正整数表示实例化顺序，如果数字重复了，容器会自行解决实例化顺序问题，但是应该避免重复；
5. Tomcat容器中，已经定义了一些随系统启动实例化的Servlet，自定义的Servlet的load-on-startup尽量不要占用数字1-5；

五 Servlet继承结构

5.1 Servlet 接口

> 源码及功能解释：

- 通过idea查看，此处略。

> 接口及方法说明：

- Servlet 规范接口，所有的Servlet必须实现
  - public void init(ServletConfig config) throws ServletException;
    - 初始化方法,容器在构造servlet对象后,自动调用的方法,容器负责实例化一个ServletConfig对象,并在调用该方法时传入。
    - ServletConfig对象可以为Servlet 提供初始化参数。
  - public ServletConfig getServletConfig();
    - 获取ServletConfig对象的方法,后续可以通过该对象获取Servlet初始化参数。
  - public void service(ServletRequest req, ServletResponse res) throws ServletException, IOException;
    - 处理请求并做出响应的服务方法,每次请求产生时由容器调用。
    - 容器创建一个ServletRequest对象和ServletResponse对象,容器在调用service方法时,传入这两个对象。
  - public String getServletInfo();
    - 获取ServletInfo信息的方法。

    - public void destroy();
        - Servlet实例在销毁之前调用的方法。

5.2 GenericServlet 抽象类

> 源码：

- 通过idea查看，此处略。

> 源码解释：

- GenericServlet 抽象类是对Servlet接口一些固定功能的粗糙实现,以及对service方法的再次抽象声明,并定义了一些其他相关功能方法。
    - private transient ServletConfig config;
        - 初始化配置对象作为属性。
    - public GenericServlet() { }
        - 构造器,为了满足继承而准备。
    - public void destroy() { }
        - 销毁方法的平庸实现。
    - public String getInitParameter(String name)
        - 获取初始参数的快捷方法。
    - public Enumeration getInitParameterNames()
        - 返回所有初始化参数名的方法。
    - public ServletConfig getServletConfig()
        - 获取初始Servlet初始配置对象ServletConfig的方法。
    - public ServletContext getServletContext()
        - 获取上下文对象ServletContext的方法。
    - public String getServletInfo()
        - 获取Servlet信息的平庸实现。
    - public void init(ServletConfig config) throws ServletException()
        - 初始化方法的实现,并在此调用了init的重载方法。
    - public void init() throws ServletException
        - 重载init方法,为了让我们自己定义初始化功能的方法。
    - public void log(String msg)
    - public void log(String message, Throwable t)
        - 打印日志的方法及重载。
    - public abstract void service(ServletRequest req, ServletResponse res) throws ServletException, IOException;
        - 服务方法再次声明。
    - public String getServletName()
        - 获取ServletName的方法。

5.3 HttpServlet 抽象类

> 源码：

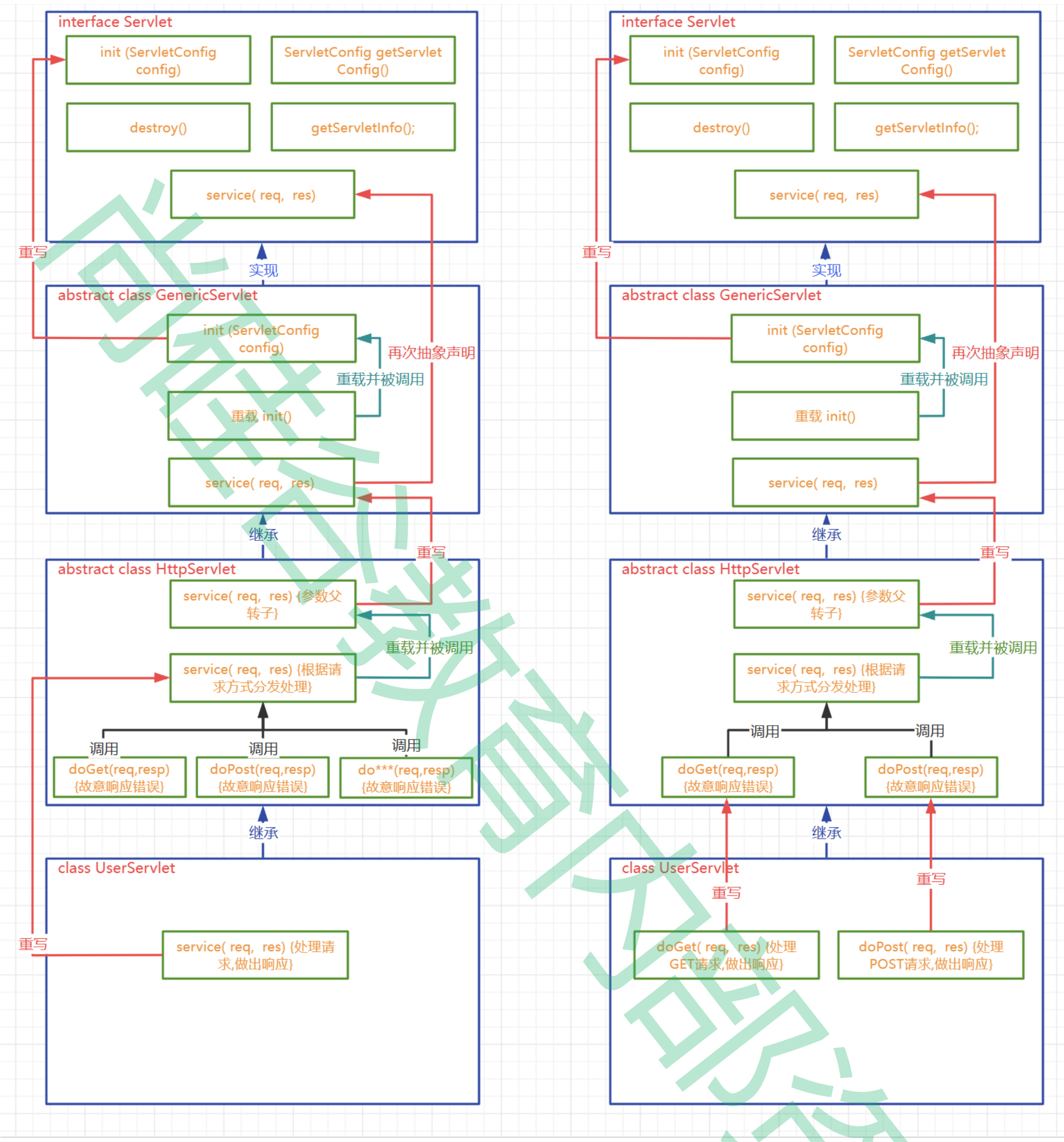
- 通过idea查看，此处略。

> 解释：

- abstract class HttpServlet extends GenericServlet HttpServlet抽象类,除了基本的实现以外,增加了更多的基础功能。
    - private static final String METHOD_DELETE = "DELETE";
    - private static final String METHOD_HEAD = "HEAD";
    - private static final String METHOD_GET = "GET";
    - private static final String METHOD_OPTIONS = "OPTIONS";
    - private static final String METHOD_POST = "POST";
    - private static final String METHOD_PUT = "PUT";
    - private static final String METHOD_TRACE = "TRACE";
        - 上述属性用于定义常见请求方式名常量值。
    - public HttpServlet() {}
        - 构造器,用于处理继承。
    - public void service(ServletRequest req, ServletResponse res) throws ServletException, IOException
        - 对服务方法的实现。
        - 在该方法中,将请求和响应对象转换成对应HTTP协议的HttpServletRequest HttpServletResponse对象。
        - 调用重载的service方法。
    - public void service(HttpServletRequest req, HttpServletResponse res) throws ServletException, IOException
        - 重载的service方法,被重写的service方法所调用。
        - 在该方法中,通过请求方式判断,调用具体的do***方法完成请求的处理。
    - protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException
    - protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException
    - protected void doHead(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException
    - protected void doPut(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException
    - protected void doDelete(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException
    - protected void doOptions(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException
    - protected void doTrace(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException
        - 对应不同请求方式的处理方法。
        - 除了doOptions和doTrace方法,其他的do*** 方法都在故意响应错误信息。

5.4 自定义Servlet

> 继承关系图解：



- 自定义Servlet中，必须要对处理请求的方法进行重写：
  - 要么重写service方法；
  - 要么重写doGet/doPost方法；

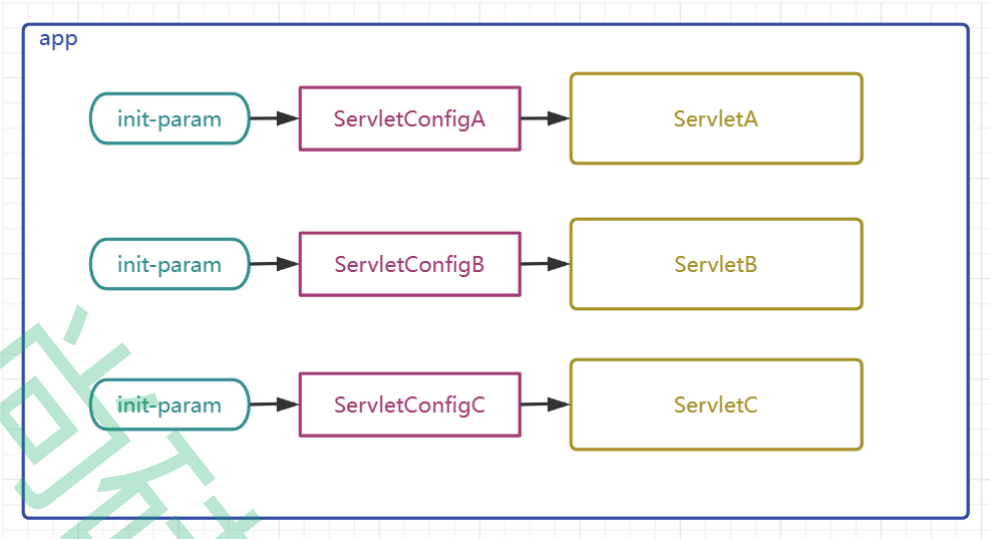
六 ServletConfig和ServletContext

6.1 ServletConfig的使用

> ServletConfig是什么？

- 为Servlet提供初始配置参数的一种对象，每个Servlet都有自己独立唯一的ServletConfig对象；

- 容器会为每个Servlet实例化一个ServletConfig对象，并通过Servlet生命周期的init方法传入给Servlet作为属性；



> ServletConfig是一个接口,定义了如下API：

```
package jakarta.servlet;
import java.util.Enumeration;
public interface ServletConfig {
    String getServletName();
    ServletContext getServletContext();
    String getInitParameter(String var1);
    Enumeration<String> getInitParameterNames();
}
```

方法名	作用
getServletName()	获取<servlet-name>HelloServlet</servlet-name>定义的Servlet名称。
getServletContext()	获取ServletContext对象。
getInitParameter()	获取配置Servlet时设置的『初始化参数』，根据名字获取值。
getInitParameterNames()	获取所有初始化参数名组成的Enumeration对象。

> ServletConfig怎么用，测试代码如下：

- 定义Servlet

```
public class ServletA extends HttpServlet {
    @Override
    protected void service(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        ServletConfig servletConfig = this.getServletConfig();
        // 根据参数名获取单个参数
        String value = servletConfig.getInitParameter("param1");
        System.out.println("param1:"+value);
        // 获取所有参数名
        Enumeration<String> parameterNames = servletConfig.getInitParameterNames();
        // 迭代并获取参数名
        while (parameterNames.hasMoreElements()) {
```

```
            String paramaterName = parameterNames.nextElement();

 System.out.println(paramaterName+":"+servletConfig.getInitParameter(paramaterName));
        }
    }
}
```

```
public class ServletB extends HttpServlet {
    @Override
    protected void service(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        ServletConfig servletConfig = this.getServletConfig();
        // 根据参数名获取单个参数
        String value = servletConfig.getInitParameter("param1");
        System.out.println("param1:"+value);
        // 获取所有参数名
        Enumeration<String> parameterNames = servletConfig.getInitParameterNames();
        // 迭代并获取参数名
        while (parameterNames.hasMoreElements()) {
            String paramaterName = parameterNames.nextElement();

 System.out.println(paramaterName+":"+servletConfig.getInitParameter(paramaterName));
        }
    }
}
```

- 配置Servlet

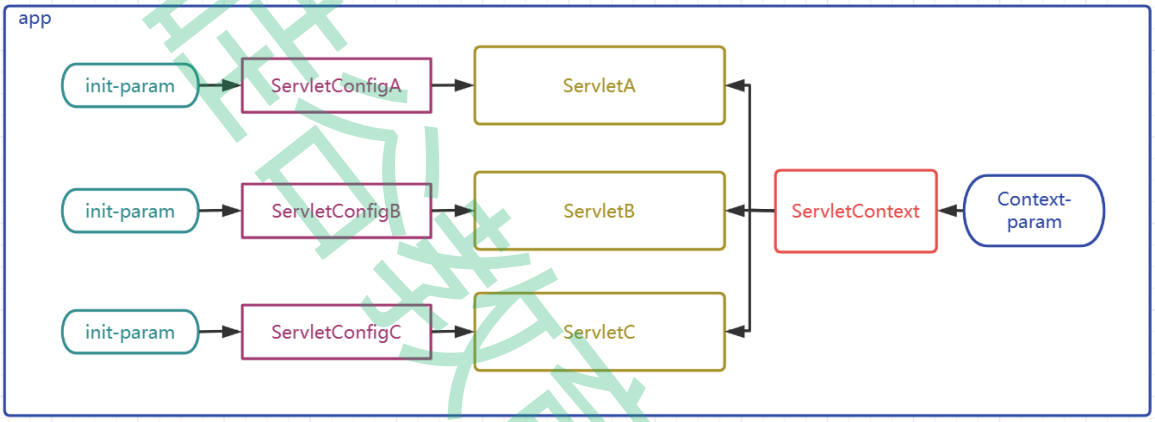
```
  <servlet>
       <servlet-name>ServletA</servlet-name>
       <servlet-class>com.atguigu.servlet.ServletA</servlet-class>
       <!--配置ServletA的初始参数-->
       <init-param>
           <param-name>param1</param-name>
           <param-value>value1</param-value>
       </init-param>
       <init-param>
           <param-name>param2</param-name>
           <param-value>value2</param-value>
       </init-param>
   </servlet>
    <servlet>
        <servlet-name>ServletB</servlet-name>
        <servlet-class>com.atguigu.servlet.ServletB</servlet-class>
        <!--配置ServletB的初始参数-->
        <init-param>
            <param-name>param3</param-name>
            <param-value>value3</param-value>
        </init-param>
        <init-param>
            <param-name>param4</param-name>
            <param-value>value4</param-value>
        </init-param>
    </servlet>
    <servlet-mapping>
        <servlet-name>ServletA</servlet-name>
        <url-pattern>/servletA</url-pattern>
    </servlet-mapping>
```

```
    <servlet-mapping>
        <servlet-name>ServletB</servlet-name>
        <url-pattern>/servletB</url-pattern>
    </servlet-mapping>
```

6.2 ServletContext的使用

> ServletContext是什么?

- ServletContext对象有称呼为上下文对象，或者叫应用域对象(后面统一讲解域对象);
- 容器会为每个app创建一个独立的唯一的ServletContext对象;
- ServletContext对象为所有的Servlet所共享;
- ServletContext可以为所有的Servlet提供初始配置参数;



> ServletContext怎么用?

- 配置ServletContext参数

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="https://jakarta.ee/xml/ns/jakartaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="https://jakarta.ee/xml/ns/jakartaee
https://jakarta.ee/xml/ns/jakartaee/web-app_5_0.xsd"
         version="5.0">
    <context-param>
        <param-name>paramA</param-name>
        <param-value>valueA</param-value>
    </context-param>
    <context-param>
        <param-name>paramB</param-name>
        <param-value>valueB</param-value>
    </context-param>
</web-app>
```

- 在Servlet中获取ServletContext并获取参数

```
public class ServletA extends HttpServlet {
    @Override
    protected void service(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        // 从ServletContext中获取为所有的Servlet准备的参数
        ServletContext servletContext = this.getServletContext();
```

```
        String valueA = servletContext.getInitParameter("paramA");
        System.out.println("paramA:"+valueA);
        // 获取所有参数名
        Enumeration<String> initParameterNames =
servletContext.getInitParameterNames();
        // 迭代并获取参数名
        while (initParameterNames.hasMoreElements()) {
            String paramaterName = initParameterNames.nextElement();

 System.out.println(paramaterName+":"+servletContext.getInitParameter(paramaterName));
        }
    }
}
```

6.3 ServletContext其他重要API

> 获取资源的磁盘路径：

```
String realPath = servletContext.getRealPath("资源在web目录中的路径");
```

- 例如我们的目标是需要获取项目中某个静态资源的路径，不是工程目录中的路径，而是**部署目录中的路径**；我们如果直接拷贝其在我们电脑中的完整路径的话其实是有问题的，因为如果该项目以后部署到公司服务器上的话，路径肯定是会发生改变的，所以我们需要使用代码动态获取资源的真实路径. 只要使用了servletContext动态获取资源的真实路径，**那么无论项目的部署路径发生什么变化，都会动态获取项目运行时候的实际磁盘路径**，所以就不会发生由于写死真实路径而导致项目部署位置改变引发的路径错误问题。

> 获取项目的上下文路径：

```
String contextPath = servletContext.getContextPath();
```

- 项目的部署名称，也叫项目的上下文路径，在部署进入tomcat时所使用的路径，该路径是可能发生变化的，通过该API动态获取项目真实的上下文路径，可以**帮助我们解决一些后端页面渲染技术或者请求转发和响应重定向中的路径问题。**

> 域对象的相关API：

- 域对象：一些用于在一些特定的范围内存储数据和传递数据的对象，不同的范围称为不同的“域”，不同的域对象代表不同的域，共享数据的范围也不同；
- ServletContext代表应用，所以ServletContext域也叫作应用域，是webapp中最大的域，可以在本应用内实现数据的共享和传递；
- webapp中的三大域对象，分别是应用域，会话域，请求域。后续我们会将三大域对象统一进行讲解和演示；
- 三大域对象都具有的API如下：

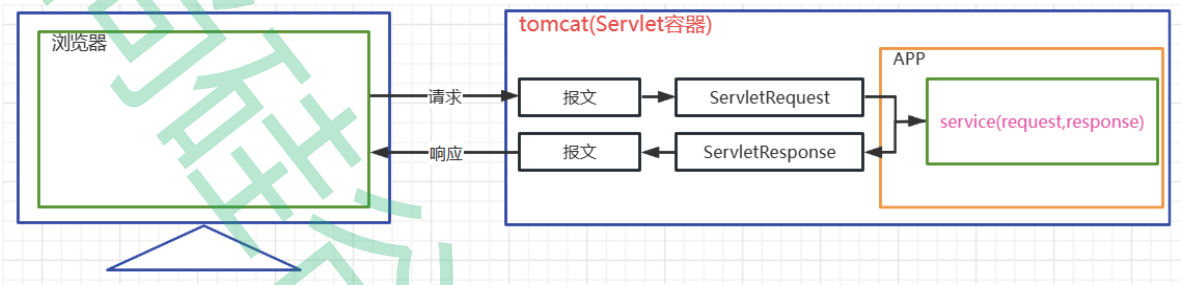
API	功能解释
void setAttribute(String key,Object value);	向域中存储/修改数据
Object getAttribute(String key);	获得域中的数据
void removeAttribute(String key);	移除域中的数据

七 HttpServletRequest

7.1 HttpServletRequest简介

> HttpServletRequest是什么?

- HttpServletRequest是一个接口，其父接口是ServletRequest；
- HttpServletRequest是Tomcat将请求报文转换封装而来的对象，在Tomcat调用service方法时传入；
- HttpServletRequest代表客户端发来的请求，请求中的所有信息都可以通过该对象获得；



7.2 HttpServletRequest常见API

> HttpServletRequest怎么用?

- 获取请求行信息相关(方式，请求的url，协议及版本)：

API	功能解释
StringBuffer getRequestURL();	获取客户端请求的url。
String getRequestURI();	获取客户端请求项目中的具体资源。
int getServerPort();	获取客户端发送请求时的端口。
int getLocalPort();	获取本应用在所在容器的端口。
int getRemotePort();	获取客户端程序的端口。
String getScheme();	获取请求协议。
String getProtocol();	获取请求协议及版本号。
String getMethod();	获取请求方式。

- 获得请求头信息相关：

API	功能解释
String getHeader(String headerName);	根据头名称获取请求头。
Enumeration getHeaderNames();	获取所有的请求头名字。
String getContentType();	获取content-type请求头。

- 获得请求参数相关：

API	功能解释
String getParameter(String parameterName);	根据请求参数名获取请求单个参数值。
String[] getParameterValues(String parameterName);	根据请求参数名获取请求多个参数值数组。
Enumeration getParameterNames();	获取所有请求参数名。
Map<String, String[]> getParameterMap();	获取所有请求参数的键值对集合。
BufferedReader getReader() throws IOException;	获取读取请求体的字符输入流。
ServletInputStream getInputStream() throws IOException;	获取读取请求体的字节输入流。
int getContentLength();	获得请求体长度的字节数。

- 其他API：

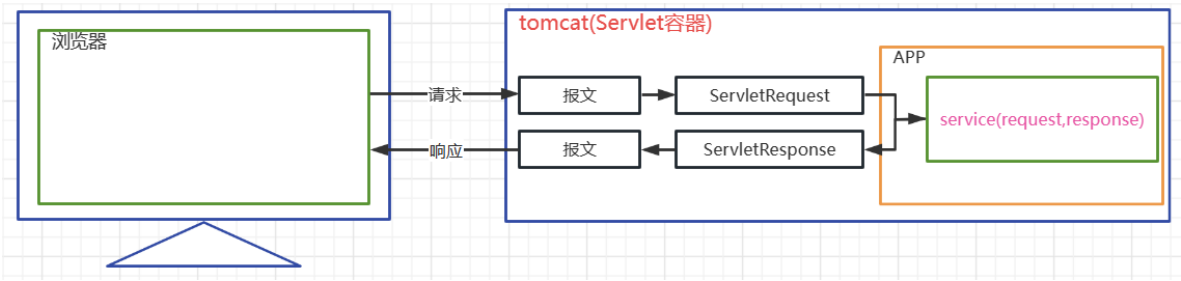
API	功能解释
String getServletPath();	获取请求的Servlet的映射路径。
ServletContext getServletContext();	获取ServletContext对象。
Cookie[] getCookies();	获取请求中的所有cookie。
HttpSession getSession();	获取Session对象。
void setCharacterEncoding(String encoding) ;	设置请求体字符集。

八 HttpServletResponse

8.1 HttpServletResponse简介

> HttpServletResponse是什么?

- HttpServletResponse是一个接口，其父接口是ServletResponse；
- HttpServletResponse是Tomcat预先创建的，在Tomcat调用service方法时传入；
- HttpServletResponse代表对客户端的响应，该对象会被转换成响应的报文发送给客户端，通过该对象我们可以设置响应信息；



8.2 HttpServletResponse的常见API

> HttpServletRequest怎么用?

- 设置响应行相关:

API	功能解释
void setStatus(int code);	设置响应状态码

- 设置响应头相关:

API	功能解释
void setHeader(String headerName, String headerValue);	设置/修改响应头键值对
void setContentType(String contentType);	设置content-type响应头及响应字符集(设置MIME类型)

- 设置响应体相关:

API	功能解释
PrintWriter getWriter() throws IOException;	获得向响应体放入信息的字符输出流
ServletOutputStream getOutputStream() throws IOException;	获得向响应体放入信息的字节输出流
void setContentLength(int length);	设置响应体的字节长度，其实就是在设置content-length响应头

- 其他API:

API	功能解释
void sendError(int code, String message) throws IOException;	向客户端响应错误信息的方法，需要指定响应码和响应信息
void addCookie(Cookie cookie);	向响应体中增加cookie
void setCharacterEncoding(String encoding);	设置响应体字符集

> MIME类型:

- MIME类型,可以理解为文档类型，用户表示传递的数据是属于什么类型的文档；
- 浏览器可以根据MIME类型决定该用什么样的方式解析接收到的响应体数据；
- 可以这样理解:：前后端交互数据时，告诉对方发给对方的是 html/css/js/图片/声音/视频/... ...；
- tomcat/conf/web.xml中配置了常见文件的拓展名和MIME类型的对应关系；
- 常见的MIME类型举例如下：

文件拓展名	MIME类型
.html	text/html
.css	text/css
.js	application/javascript
.json	application/json
.png /.jpeg/.jpg/... ...	image/jpeg
.mp3/.mpe/.mpeg/	audio/mpeg
.mp4	video/mp4
.m1v/.m1v/.m2v/.mpe/... ...	video/mpeg

九 请求转发和响应重定向

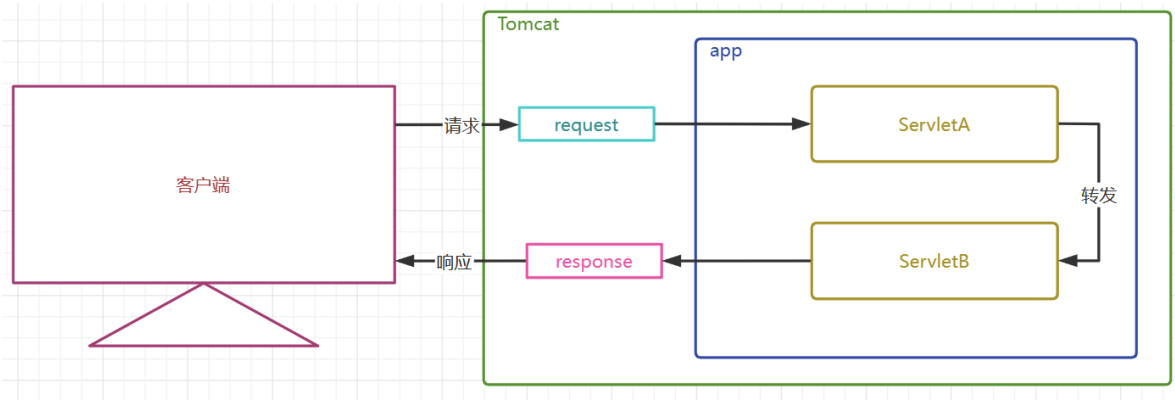
9.1 概述

> 什么是请求转发和响应重定向?

- 请求转发和响应重定向是web应用中间接访问项目资源的两种手段，也是Servlet控制页面跳转的两种手段；
- 请求转发通过HttpServletRequest实现，响应重定向通过HttpServletResponse实现；
- 请求转发生活举例： 张三找李四借钱，李四没有，李四找王五，让王五借给张三；
- 响应重定向生活举例：张三找李四借钱，李四没有，李四让张三去找王五，张三自己再去找王五借钱；

9.2 请求转发

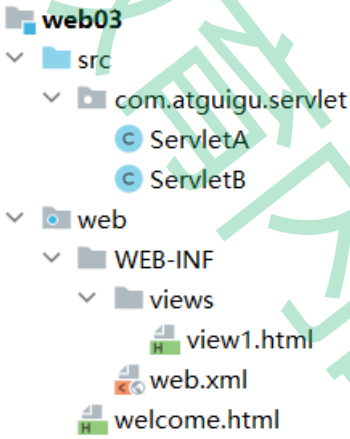
> 请求转发运行逻辑图：



请求转发特点(背诵):

- 请求转发通过HttpServletRequest对象获取请求转发器实现;
- 请求转发是服务器内部的行为，对客户端是屏蔽的;
- 客户端只发送了一次请求，客户端地址栏不变;
- 服务端只产生了一对请求和响应对象，这一对请求和响应对象会继续传递给下一个资源;
- 因为全程只有一个HttpServletRequset对象，所以请求参数可以传递,请求域中的数据也可以传递;
- 请求转发可以转发给其他Servlet动态资源，也可以转发给一些静态资源以实现页面跳转;
- 请求转发可以转发给WEB-INF下受保护的资源;
- 请求转发不能转发到本项目以外的外部资源;

请求转发测试代码:



- ServletA

```
@WebServlet("/servletA")
public class ServletA extends HttpServlet {
    @Override
    protected void service(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        //  获取请求转发器
        //  转发给servlet  ok
        RequestDispatcher  requestDispatcher = req.getRequestDispatcher("servletB");
        //  转发给一个视图资源 ok
        //RequestDispatcher requestDispatcher =
req.getRequestDispatcher("welcome.html");
        //  转发给WEB-INF下的资源  ok
        //RequestDispatcher requestDispatcher = req.getRequestDispatcher("WEB-
INF/views/view1.html");
        //  转发给外部资源   no
```

```
        //RequestDispatcher requestDispatcher =
req.getRequestDispatcher("http://www.atguigu.com");
        //  获取请求参数
        String username = req.getParameter("username");
        System.out.println(username);
        //  向请求域中添加数据
        req.setAttribute("reqKey","requestMessage");
        //  做出转发动作
        requestDispatcher.forward(req,resp);
    }
}
```

- ServletB

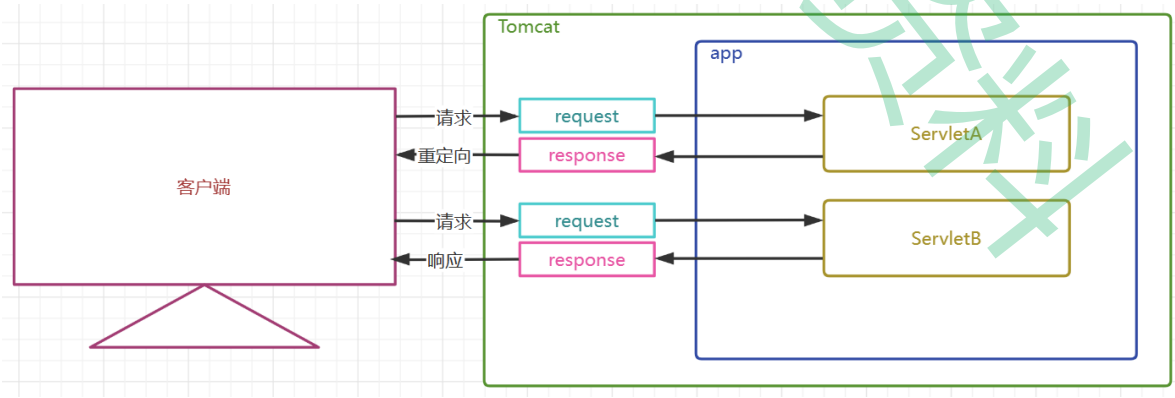
```
@WebServlet("/servletB")
public class ServletB extends HttpServlet {
    @Override
    protected void service(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        // 获取请求参数
        String username = req.getParameter("username");
        System.out.println(username);
        // 获取请求域中的数据
        String reqMessage = (String)req.getAttribute("reqKey");
        System.out.println(reqMessage);
        // 做出响应
        resp.getWriter().write("servletB response");
    }
}
```

- 打开浏览器,输入以下url测试

```
http://localhost:8080/web03_war_exploded/servletA?username=atguigu
```

9.3 响应重定向

> 响应重定向运行逻辑图:

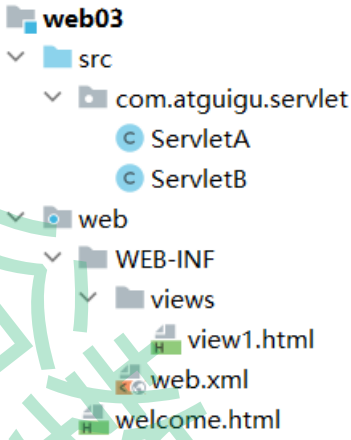


> 响应重定向特点(背诵):

- 响应重定向通过HttpServletResponse对象的sendRedirect方法实现;

- 响应重定向是服务端通过302响应码和路径，告诉客户端自己去找其他资源，是在服务端提示下的客户端的行为；
- 客户端至少发送了两次请求，客户端地址栏是要变化的；
- 服务端产生了多对请求和响应对象，且请求和响应对象不会传递给下一个资源；
- 因为全程产生了多个HttpServletRequset对象，所以请求参数不可以传递，请求域中的数据也不可以传递；
- 重定向可以是其他Servlet动态资源，也可以是一些静态资源以实现页面跳转；
- 重定向不可以到给WEB-INF下受保护的资源；
- 重定向可以到本项目以外的外部资源；

> 响应重定向测试代码：



- ServletA

```
@WebServlet("/servletA")
public class ServletA extends HttpServlet {
    @Override
    protected void service(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        //  获取请求参数
        String username = req.getParameter("username");
        System.out.println(username);
        //  向请求域中添加数据
        req.setAttribute("reqKey","requestMessage");
        //  响应重定向
        // 重定向到servlet动态资源 OK
        resp.sendRedirect("servletB");
        // 重定向到视图静态资源 OK
        //resp.sendRedirect("welcome.html");
        // 重定向到WEB-INF下的资源 NO
        //resp.sendRedirect("WEB-INF/views/view1");
        // 重定向到外部资源
        //resp.sendRedirect("http://www.atguigu.com");
    }
}
```

- ServletB

```
@WebServlet("/servletB")
public class ServletB extends HttpServlet {
    @Override
    protected void service(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        // 获取请求参数
```

```
        String username = req.getParameter("username");
        System.out.println(username);
        // 获取请求域中的数据
        String reqMessage = (String)req.getAttribute("reqKey");
        System.out.println(reqMessage);
        // 做出响应
        resp.getWriter().write("servletB response");

    }
}
```

- 打开浏览器,输入以下url测试

```
http://localhost:8080/web03_war_exploded/servletA?username=atguigu
```

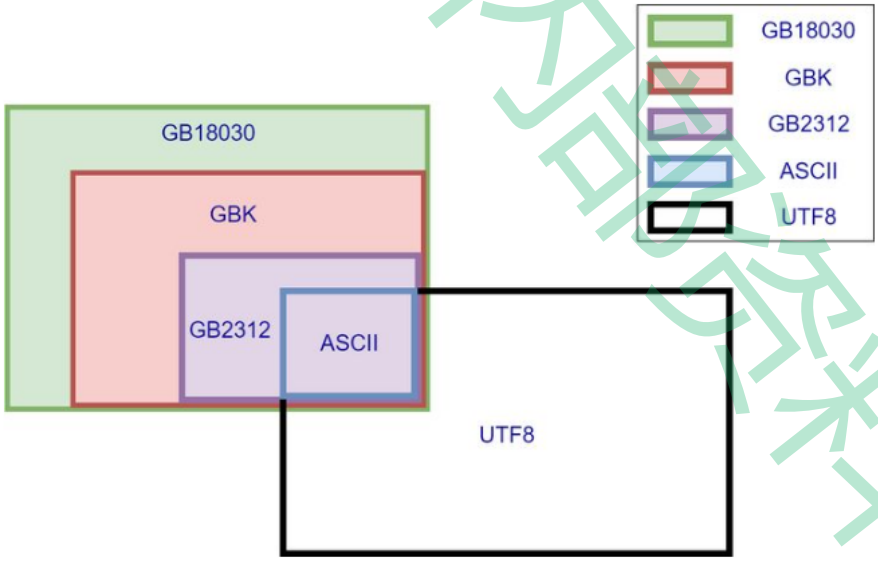
十 web乱码和路径问题总结

10.1 乱码问题

> 乱码问题产生的根本原因是什么?

1. 数据的编码和解码使用的不是同一个字符集
2. 使用了不支持某个语言文字的字符集

> 各个字符集的兼容性:

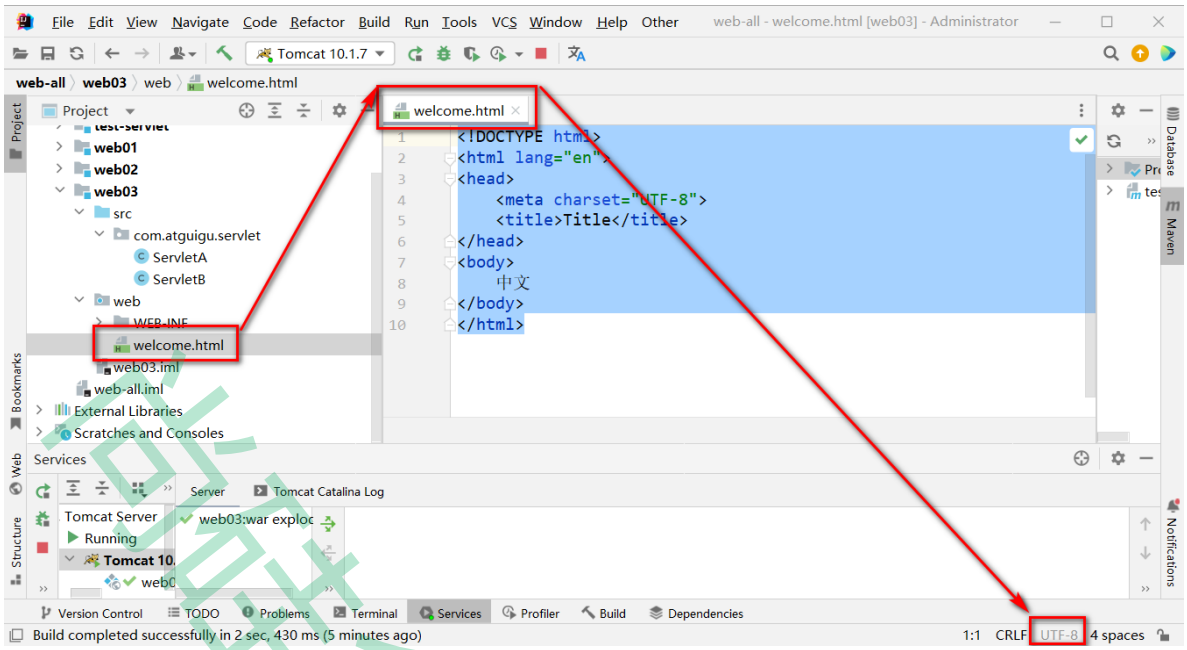


- 由上图得知，上述字符集都兼容了ASCII;
- ASCII中有什么? 英文字母和一些通常使用的符号，所以这些东西无论使用什么字符集都不会乱码;

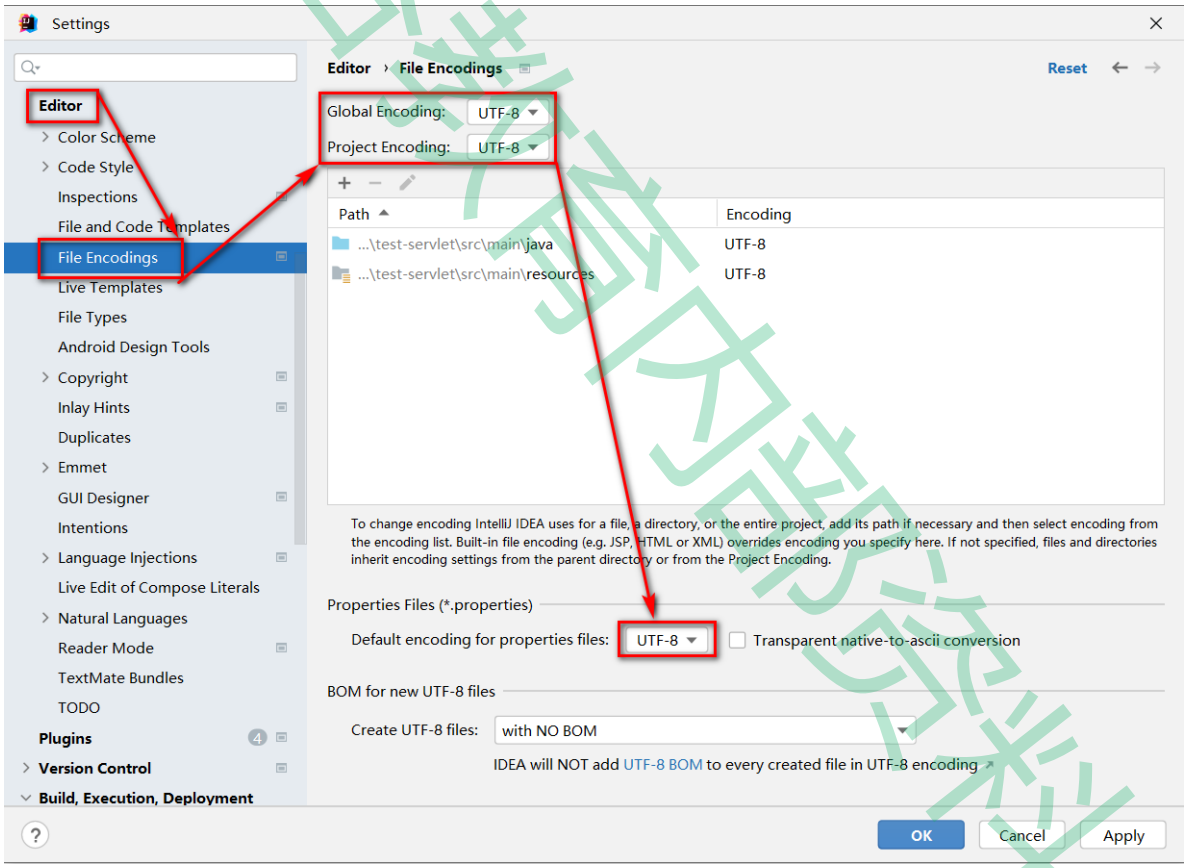
10.1.1 HTML乱码问题

> 设置项目文件的字符集要使用一个支持中文的字符集。

- 查看当前文件的字符集。



- 查看项目字符集配置，将Global Encoding 全局字符集，Project Encoding 项目字符集，Properties Files 属性配置文件字符集设置为UTF-8。



> 当前视图文件的字符集通过 `<meta charset="UTF-8">` 来告知浏览器通过什么字符集来解析当前文件：

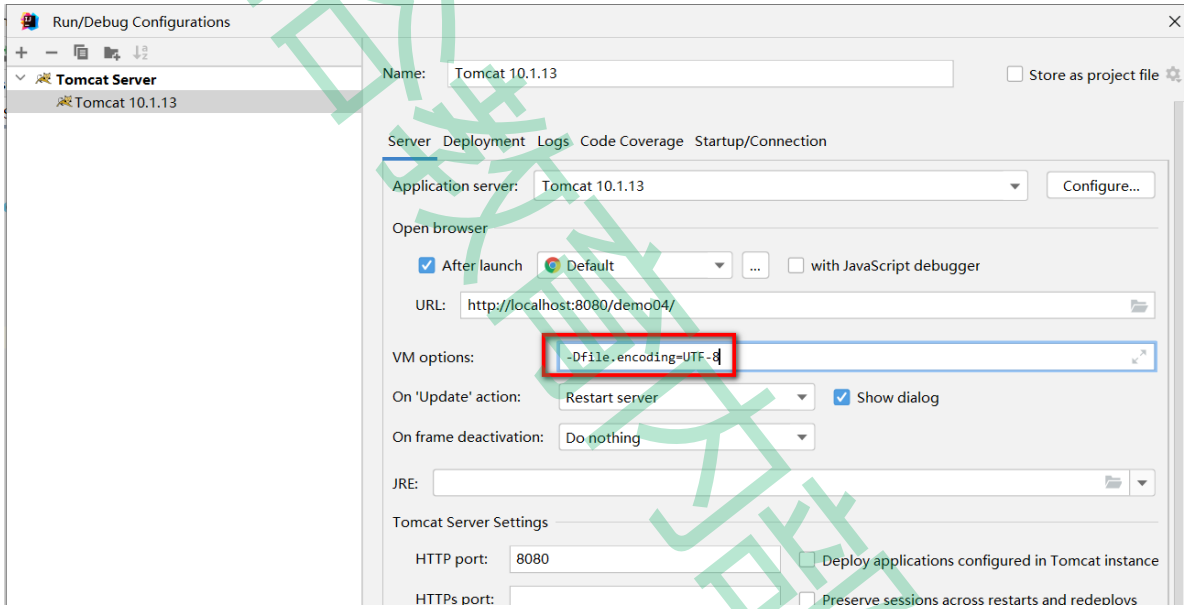
```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
    中文
</body>
</html>
```

10.1.2 Tomcat控制台乱码

> 在tomcat10.1.7这个版本中，修改 Tomcat/conf/logging.properties中，所有的UTF-8为GBK即可。

> sout乱码问题，设置JVM加载.class文件时使用UTF-8字符集。

- 设置虚拟机加载.class文件的字符集和编译时使用的字符集一致。



10.1.3 请求乱码问题

10.1.3.1 GET请求乱码

> GET请求方式乱码分析：

- GET方式提交参数的方式是将参数放到URL后面，如果使用的不是UTF-8，那么会对参数进行URL编码处理；
- HTML中的 `<meta charset='字符集'/>` 影响了GET方式提交参数的URL编码；
- Tomcat10.1.7的URI编码默认为 UTF-8；
- 当GET方式提交的参数URL编码和Tomcat10.1.7默认的URI编码不一致时，就会出现乱码；

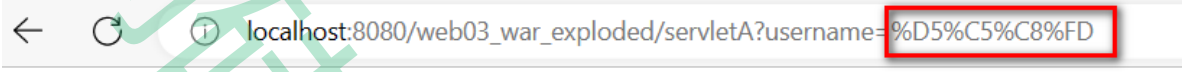
> GET请求方式乱码演示：

- 浏览器解析的文档的 `<meta charset="GBK" />` 。

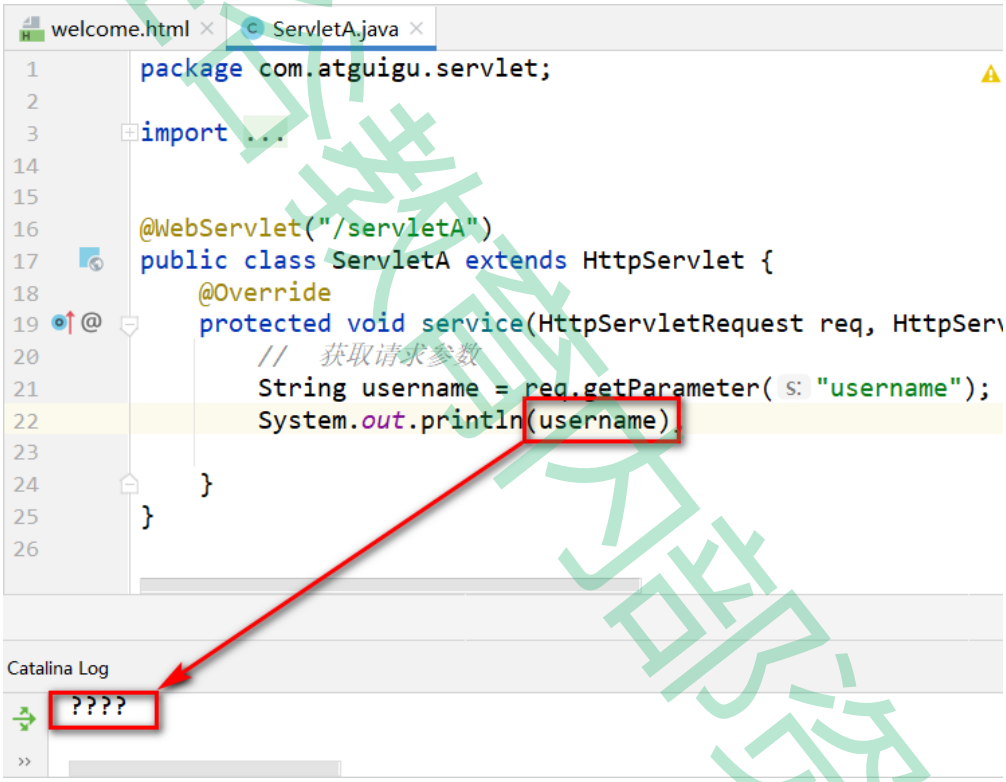
welcome.html | ServletA.java

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="GBK">
    <title>Title</title>
</head>
<body>

    <form action="servletA" method="get">
        用户名:<input type="text" name="username"><input type="submit">
    </form>
</body>
</html>
```

- GET方式提交时，会对数据进行URL编码处理，是将GBK 转码为 "百分号码"。



- Tomcat10.1.7 默认使用UTF-8对URI进行解析，造成前后端使用的字符集不一致，出现乱码。



GET请求方式乱码解决：

- 方式1： 设置GET方式提交的编码和Tomcat10.1.7的URI默认解析编码一致即可 (推荐)。

welcome.html | ServletA.java

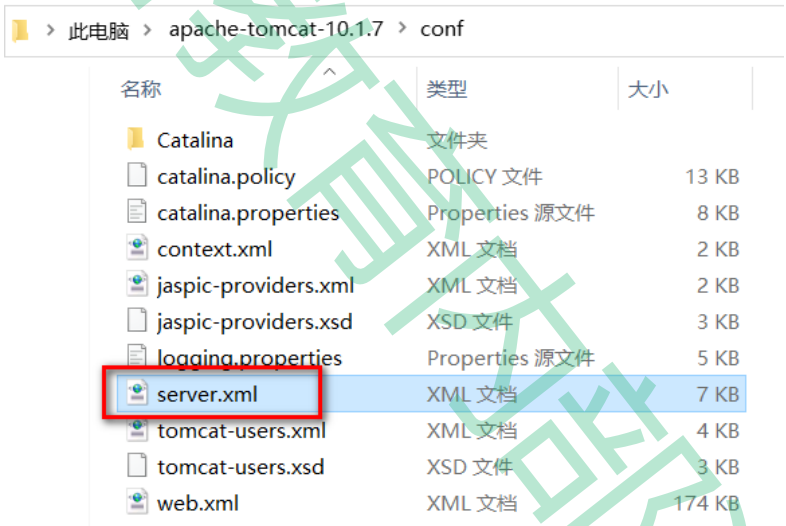
```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>

    <form action="servletA" method="get">
        用户名:<input type="text" name="username"><input type="subm
    </form>
</body>
</html>
```

html › body

Catalina Log

张三

- 方式2： 设置Tomcat10.1.7的URI解析字符集和GET请求发送时所使用URL转码时的字符集一致即可，修改conf/server.xml中 Connecter 添加 URIEncoding="GBK" (此时不推荐)。



server.xml

```
        HTTP Connector: /docs/config/http.html
        AJP  Connector: /docs/config/ajp.html
        Define a non-SSL/TLS HTTP/1.1 Connector on port 8080
    -->
    <Connector port="8080" protocol="HTTP/1.1"
               connectionTimeout="20000"
               redirectPort="8443" URIEncoding="GBK"/>
```

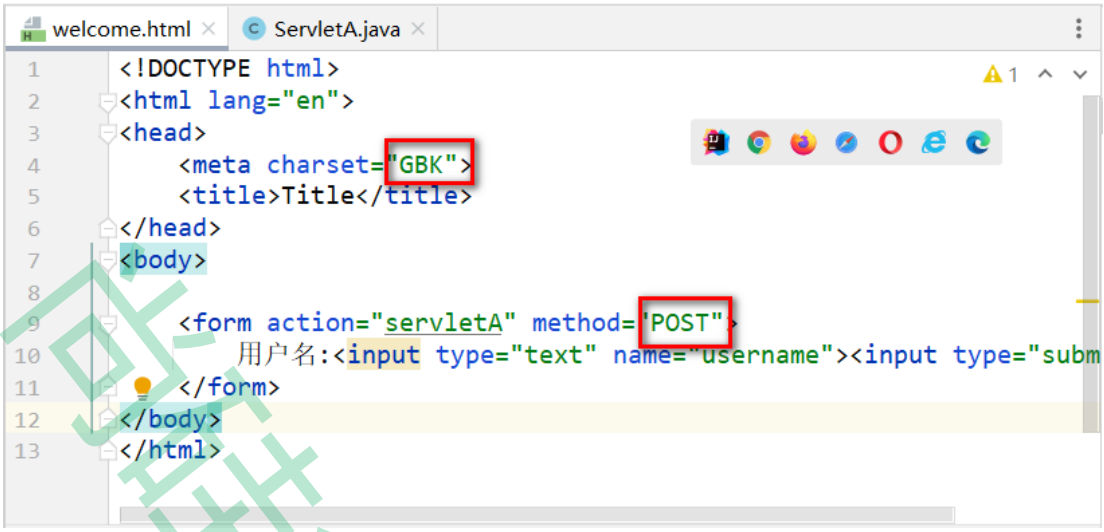
10.1.3.2 POST方式请求乱码

> POST请求方式乱码分析：

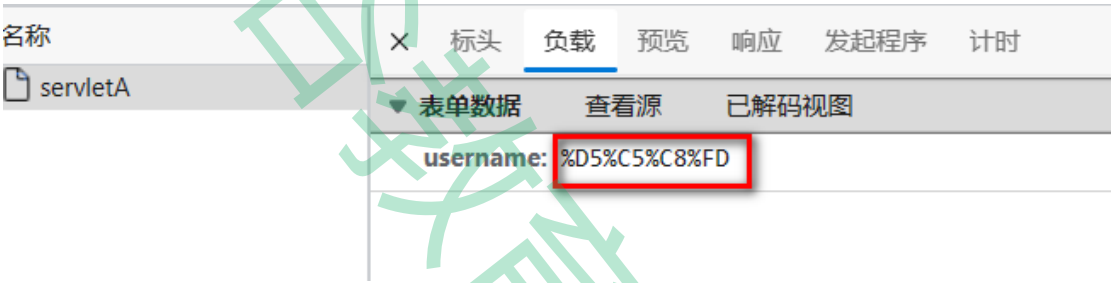
- form表单的POST请求将参数放在请求体中进行发送；
- 请求体使用的字符集受到了 `<meta charset="字符集"/>` 的影响；
- Tomcat10.1.7 默认使用UTF-8字符集对请求体进行解析；
- 如果请求体的URL转码和Tomcat的请求体解析编码不一致，就容易出现乱码；

POST方式乱码演示：

- POST请求请求体受到了 `<meta charset="字符集"/>` 的影响。



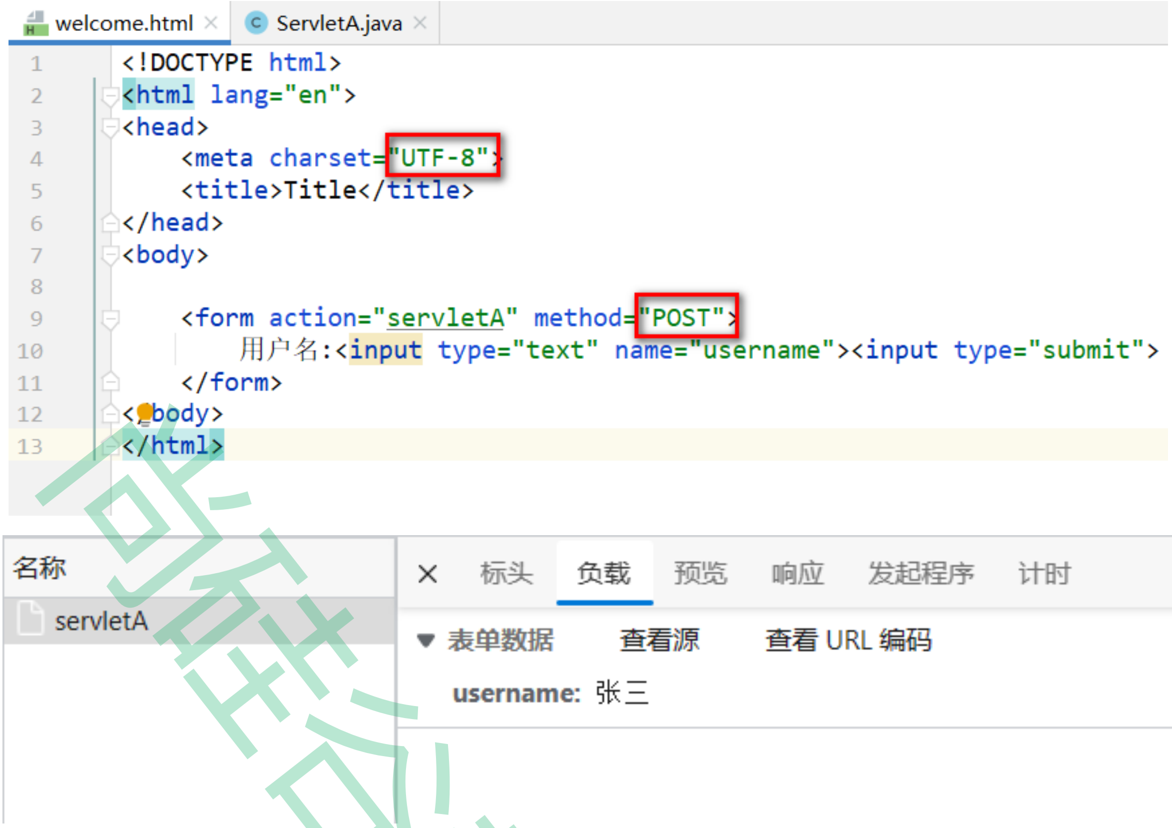
- 请求体中，将GBK数据进行URL编码。



- 后端默认使用UTF-8解析请求体，出现字符集不一致，导致乱码。

POST请求方式乱码解决：

- 方式1：请求时，使用UTF-8字符集提交请求体 (推荐)。



- 方式2：后端在获取参数前，设置解析请求体使用的字符集和请求发送时使用的字符集一致（此时不推荐）。

10.1.3 响应乱码问题

> 响应乱码分析：

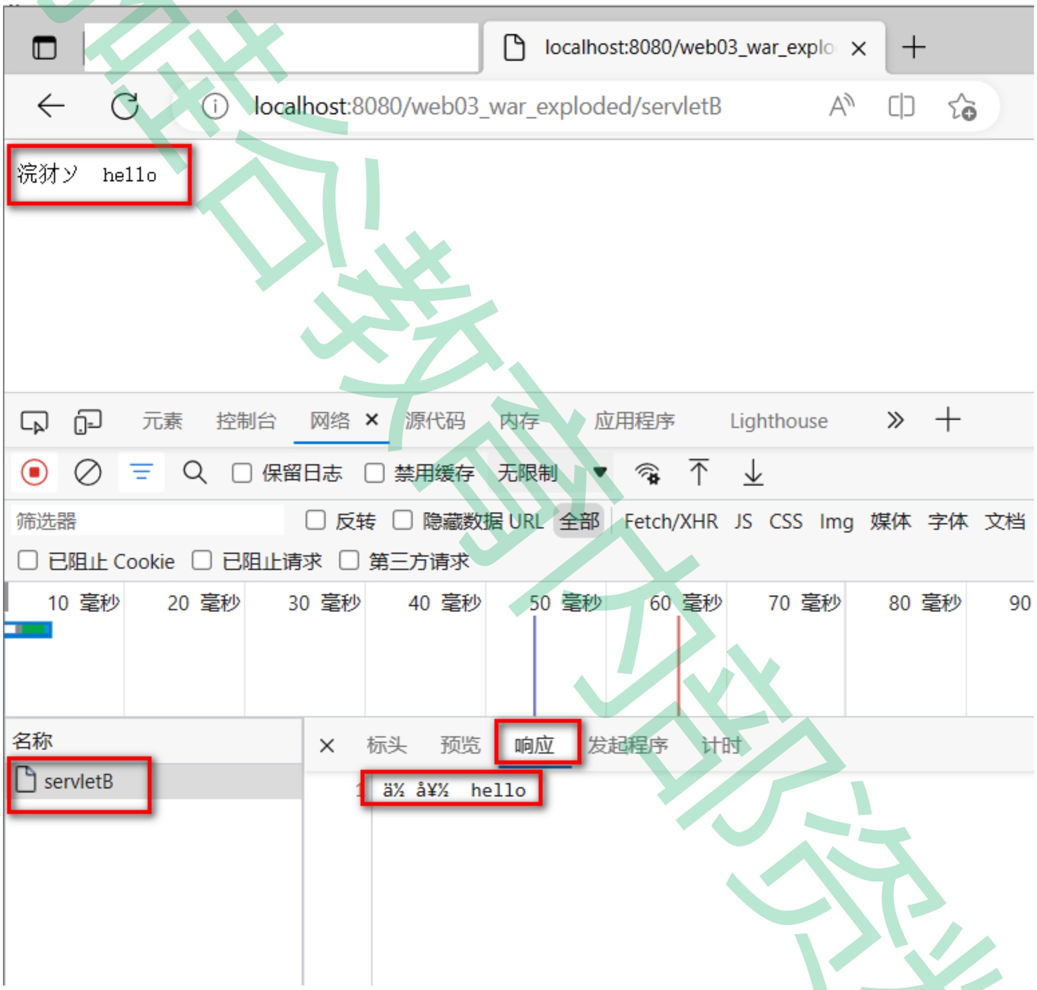
- 在Tomcat10.1.7中，向响应体中放入的数据默认使用了工程编码 UTF-8。
- 浏览器在接收响应信息时，使用了不同的字符集或者是不支持中文的字符集就会出现乱码。

> 响应乱码演示：

- 服务端通过response对象向响应体添加数据。

```java
@WebServlet("/servletB")
public class ServletB extends HttpServlet {
    @Override
    protected void service(HttpServletRequest req,
        resp.getWriter().write( s: "你好  hello");

    }
}
```

- 浏览器接收数据解析乱码。



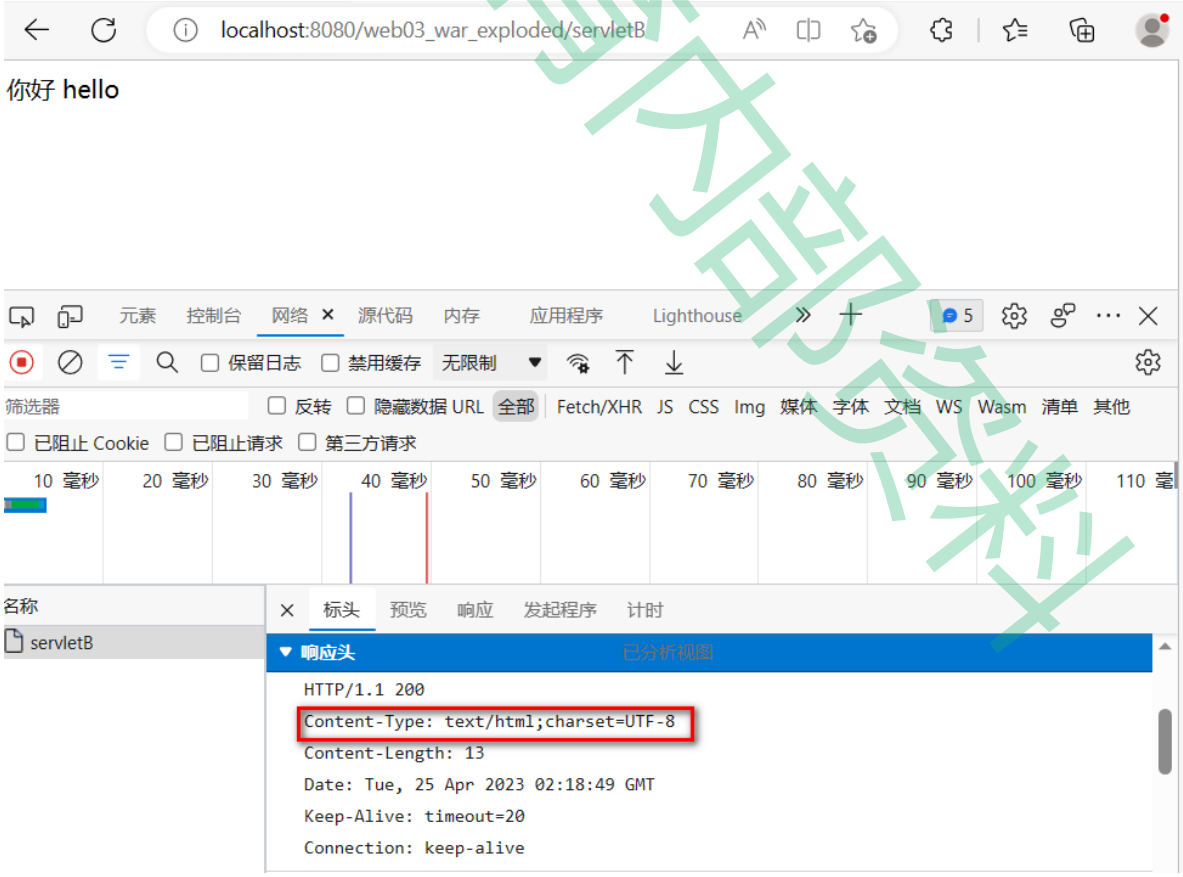
> 响应乱码解决：

- 方式1： 手动设定浏览器对本次响应体解析时使用的字符集(不推荐)。
    - edge和 chrome浏览器没有提供直接的比较方便的入口，不方便；
- 方式2：后端通过设置响应体的字符集和浏览器解析响应体的默认字符集一致(此时不推荐)。

```java
import ...

@WebServlet("/servletB")
public class ServletB extends HttpServlet {
    @Override
    protected void service(HttpServletRequest req, HttpServletRes
        // 设置响应体使用的字符集和浏览器默认的字符集一致
        resp.setCharacterEncoding("GBK");
        resp.getWriter().write( s: "你好  hello");

    }
}
```

方式3: 通过设置Content-Type响应头，告诉浏览器以指定的字符集解析响应体(推荐)。

```java
@WebServlet("/servletB")
public class ServletB extends HttpServlet {
    @Override
    protected void service(HttpServletRequest req, HttpSer
        // 设置content-type响应头
        // 告诉浏览器以指定的MIME类型和字符集解析响应体
        resp.setContentType("text/html;charset=UTF-8");
        resp.getWriter().write( s: "你好  hello");

    }
}
```



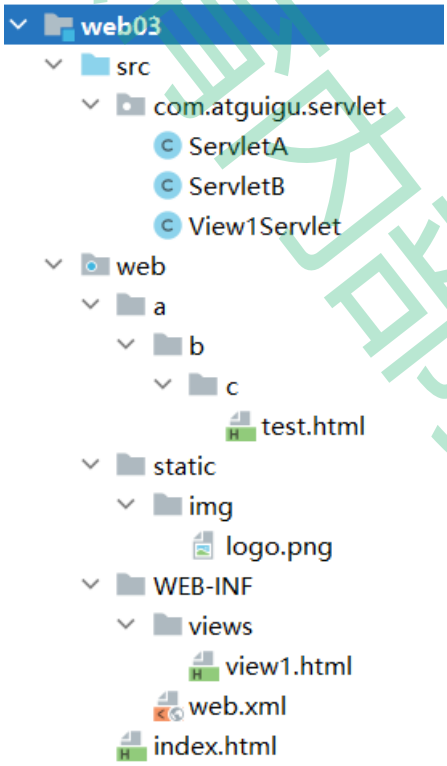
10.2 路径问题

> 相对路径和绝对路径：

- 相对路径：
  - 相对路径的规则是以当前资源所在的路径为出发点去寻找目标资源；
  - 相对路径不以 / 开头；
  - 在file协议下，使用的是磁盘路径；
  - 在http协议下，使用的是url路径；
  - 相对路径中可以使用 ./表示当前资源所在路径,可以省略不写；
  - 相对路径中可以使用../表示当前资源所在路径的上一层路径，需要时要手动添加；
- 绝对路径：
  - 绝对路径的规则是: 使用以一个固定的路径做出出发点去寻找目标资源，和当前资源所在的路径没有关系；
  - 绝对路径要以/ 开头；
  - 绝对路径的写法中，不以当前资源的所在路径为出发点,所以不会出现 ./ 和../开头；
  - 不同的项目和不同的协议下，绝对路径的基础位置可能不同,要通过测试确定；
  - 绝对路径的好处就是，无论当前资源位置在哪,寻找目标资源路径的写法都一致；
- 应用场景：
  1. 前端代码中，href、src、action 等属性；
  2. 请求转发和重定向中的路径；

10.2.1 前端路径问题

> 项目结构：



10.2.1.1 相对路径情况分析

> 相对路径情况1：web/index.html中引入web/static/img/logo.png。

- 访问index.html的url为 ： http://localhost:8080/web03_war_exploded/index.html
- 当前资源为 ：index.html

- 当前资源的所在路径为：http://localhost:8080/web03_war_exploded/
- 要获取的目标资源url为：http://localhost:8080/web03_war_exploded/static/img/logo.png
- index.html中定义的了　：`<img src="static/img/logo.png"/>`
- 寻找方式就是在当前资源所在路径(http://localhost:8080/web03_war_exploded/)后拼接src属性值(static/img/logo.png)，正好是目标资源正常获取的url(http://localhost:8080/web03_war_exploded/static/img/logo.png)

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
    <img src="static/img/logo.png">
</body>
</html>
```

> 相对路径情况2：web/a/b/c/test.html中引入web/static/img/logo.png

- 访问test.html的url为　：http://localhost:8080/web03_war_exploded/a/b/c/test.html
- 当前资源为　　　　　：test.html
- 当前资源的所在路径为：http://localhost:8080/web03_war_exploded/a/b/c/
- 要获取的目标资源url为：http://localhost:8080/web03_war_exploded/static/img/logo.png
- test.html中定义的了　：`<img src="../../../static/img/logo.png"/>`
- 寻找方式就是在当前资源所在路径(http://localhost:8080/web03_war_exploded/a/b/c/)后拼接src属性值(../../../static/img/logo.png)，其中 ../可以抵消一层路径，正好是目标资源正常获取的url(http://localhost:8080/web03_war_exploded/static/img/logo.png)

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
    <!-- ../代表上一层路径 -->
    <img src="../../../static/img/logo.png">
</body>
</html>
```

> 相对路径情况3：web/WEB-INF/views/view1.html中引入web/static/img/logo.png

- view1.html在WEB-INF下，需要通过Servlet请求转发获得

```java
@WebServlet("/view1Servlet")
public class View1Servlet extends HttpServlet {
    @Override
    protected void service(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        RequestDispatcher requestDispatcher = req.getRequestDispatcher("WEB-
INF/views/view1.html");
        requestDispatcher.forward(req,resp);
    }
}
```

- 访问view1.html的url为 ： http://localhost:8080/web03_war_exploded/view1Servlet
- 当前资源为 ：view1Servlet
- 当前资源的所在路径为：http://localhost:8080/web03_war_exploded/
- 要获取的目标资源url为：http://localhost:8080/web03_war_exploded/static/img/logo.png
- view1.html中定义的了 ：`<img src="static/img/logo.png"/>`
- 寻找方式就是在当前资源所在路径(http://localhost:8080/web03_war_exploded/)后拼接src属性值(static/img/logo.png)，正好是目标资源正常获取的url(http://localhost:8080/web03_war_exploded/static/img/logo.png)

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
    <img src="static/img/logo.png">
</body>
</html>
```

10.2.1.2 绝对路径情况分析

> 绝对路径情况1：web/index.html中引入web/static/img/logo.png

- 访问index.html的url为 ： http://localhost:8080/web03_war_exploded/index.html
- 绝对路径的基准路径为 ： http://localhost:8080
- 要获取的目标资源url为：http://localhost:8080/web03_war_exploded/static/img/logo.png
- index.html中定义的了 ：`<img src="/web03_war_exploded/static/img/logo.png"/>`
- 寻找方式就是在基准路径(http://localhost:8080)后面拼接src属性值(/web03_war_exploded/static/img/logo.png)，得到的正是目标资源访问的正确路径

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
    <!-- 绝对路径写法 -->
    <img src="/web03_war_exploded/static/img/logo.png">
</body>
</html>
```

> 绝对路径情况2：web/a/b/c/test.html中引入web/static/img/logo.png

- 访问test.html的url为 ： http://localhost:8080/web03_war_exploded/a/b/c/test.html
- 绝对路径的基准路径为 ： http://localhost:8080
- 要获取的目标资源url为：http://localhost:8080/web03_war_exploded/static/img/logo.png
- test.html中定义的了 ：`<img src="/web03_war_exploded/static/img/logo.png"/>`
- 寻找方式就是在基准路径(http://localhost:8080)后面拼接src属性值(/web03_war_exploded/static/img/logo.png)，得到的正是目标资源访问的正确路径

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
    <!-- 绝对路径写法 -->
    <img src="/web03_war_exploded/static/img/logo.png">
</body>
</html>
```

> 绝对路径情况3：web/WEB-INF/views/view1.html中引入web/static/img/logo.png

- view1.html在WEB-INF下，需要通过Servlet请求转发获得

```
@WebServlet("/view1Servlet")
public class View1Servlet extends HttpServlet {
    @Override
    protected void service(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        RequestDispatcher requestDispatcher = req.getRequestDispatcher("WEB-
INF/views/view1.html");
        requestDispatcher.forward(req,resp);
    }
}
```

- 访问view1.html的url为 ： http://localhost:8080/web03_war_exploded/view1Servlet
- 绝对路径的基准路径为 ： http://localhost:8080
- 要获取的目标资源url为：http://localhost:8080/web03_war_exploded/static/img/logo.png
- view1.html中定义的了 ：`<img src="/web03_war_exploded/static/img/logo.png"/>`
- 寻找方式就是在基准路径(http://localhost:8080)后面拼接src属性值(/static/img/logo.png)，得到的正是目标资源访问的正确路径

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
    <img src="/web03_war_exploded/static/img/logo.png">
</body>
</html>
```

10.2.1.3 base标签的使用

> base标签定义页面相对路径公共前缀：

- base 标签定义在head标签中，用于定义相对路径的公共前缀；
- base 标签定义的公共前缀只在相对路径上有效，绝对路径中无效；
- 如果相对路径开头有 ./ 或者../修饰，则base标签对该路径同样无效；

> index.html 和a/b/c/test.html 以及view1Servlet 中的路径处理：

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
    <!--定义相对路径的公共前缀,将相对路径转化成了绝对路径-->
    <base href="/web03_war_exploded/">
</head>
<body>
    <img src="static/img/logo.png">
</body>
</html>
```

10.2.1.4 缺省项目上下文路径

> 项目上下文路径变化问题：

- 通过 base标签虽然解决了相对路径转绝对路径问题，但是base中定义的是项目的上下文路径；
- 项目的上下文路径是可以随意变化的；
- 一旦项目的上下文路径发生变化，所有base标签中的路径都需要改；

> 解决方案：

- 将项目的上下文路径进行缺省设置，设置为 / ，则所有的绝对路径中就不必填写项目的上下文了，直接就是/开头即可。

10.2.2 重定向中的路径问题

> 目标： 由/x/y/z/servletA重定向到a/b/c/test.html

10.2.2.1相对路径写法

- 访问ServletA的url为 ： **http://localhost:8080/web03_war_exploded/x/y/z/servletA**
- 当前资源为 ：servletA
- 当前资源的所在路径为：**http://localhost:8080/web03_war_exploded/x/x/z/**
- 要获取的目标资源url为：**http://localhost:8080/web03_war_exploded/a/b/c/test.html**
- ServletA重定向的路径 ：../../../a/b/c/test/html
- 寻找方式就是在当前资源所在路径(**http://localhost:8080/web03_war_exploded/x/y/z/**)后拼接(../../../a/b/c/test/html)，形成(**http://localhost:8080/web03_war_exploded/x/y/z/../../../a/b/c/test/html**)每个../抵消一层目录，正好是目标资源正常获取的url(**http://localhost:8080/web03_war_exploded/a/b/c/test/html**)

```
@WebServlet("/x/y/z/servletA")
public class ServletA extends HttpServlet {
    @Override
    protected void service(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        // 相对路径重定向到test.html
        resp.sendRedirect("../../../a/b/c/test.html");
    }
}
```

10.2.2.2绝对路径写法

- 访问ServletA的url为 ： http://localhost:8080/web03_war_exploded/x/y/z/servletA
- 绝对路径的基准路径为 ： http://localhost:8080
- 要获取的目标资源url为： http://localhost:8080/web03_war_exploded/a/b/c/test.html
- ServletA重定向的路径 ：/web03_war_exploded/a/b/c/test.html
- 寻找方式就是在基准路径(http://localhost:8080)后面拼接(/web03_war_exploded/a/b/c/test.html)，得到(http://localhost:8080/web03_war_exploded/a/b/c/test.html)正是目标资源访问的正确路径
- 绝对路径中需要填写项目上下文路径，但是上下文路径是变换的
  - 可以通过 ServletContext的getContextPath()获取上下文路径
  - 可以将项目上下文路径定义为 / 缺省路径，那么路径中直接以/开头即可

```
//绝对路径中,要写项目上下文路径
//resp.sendRedirect("/web03_war_exploded/a/b/c/test.html");
// 通过ServletContext对象动态获取项目上下文路径
//resp.sendRedirect(getServletContext().getContextPath()+"/a/b/c/test.html");
// 缺省项目上下文路径时,直接以/开头即可
resp.sendRedirect("/a/b/c/test.html");
```

10.2.3 请求转发中的路径问题

> 目标： 由x/y/servletB请求转发到a/b/c/test.html

10.2.3.1 相对路径写法

- 访问ServletB的url为 ： http://localhost:8080/web03_war_exploded/x/y/servletB
- 当前资源为 ：servletB
- 当前资源的所在路径为 ：http://localhost:8080/web03_war_exploded/x/x/
- 要获取的目标资源url为： http://localhost:8080/web03_war_exploded/a/b/c/test.html
- ServletA请求转发路径 ：../../a/b/c/test/html
- 寻找方式就是在当前资源所在路径(http://localhost:8080/web03_war_exploded/x/y/)后拼接(../../a/b/c/test/html)，形成(http://localhost:8080/web03_war_exploded/x/y/../../a/b/c/test/html)每个../抵消一层目录，正好是目标资源正常获取的url(http://localhost:8080/web03_war_exploded/a/b/c/test/html)

```
@WebServlet("/x/y/servletB")
public class ServletB extends HttpServlet {
    @Override
    protected void service(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
        RequestDispatcher requestDispatcher =
req.getRequestDispatcher("../../a/b/c/test.html");
        requestDispatcher.forward(req,resp);
    }
}
```

10.2.3.2绝对路径写法

- 请求转发只能转发到项目内部的资源，其绝对路径无需添加项目上下文路径；
- 请求转发绝对路径的基准路径相当于 http://localhost:8080/web03_war_exploded ；
- 在项目上下文路径为缺省值时，也无需改变，直接以/开头即可；

```java
@WebServlet("/x/y/servletB")
public class ServletB extends HttpServlet {
    @Override
    protected void service(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
        RequestDispatcher requestDispatcher =
req.getRequestDispatcher("/a/b/c/test.html");
        requestDispatcher.forward(req,resp);
    }
}
```

10.2.3.3目标资源内相对路径处理

- 此时需要注意，请求转发是服务器行为，浏览器不知道，地址栏不变化，相当于我们访问test.html的路径为 http://localhost:8080/web03_war_exploded/x/y/servletB ；
- 那么此时 test.html资源的所在路径就是 http://localhost:8080/web03_war_exploded/x/y/ 所以test.html中相对路径要基于该路径编写，如果使用绝对路径则不用考虑；

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
    <!--
        当前资源路径是      http://localhost:8080/web03_war_exploded/x/y/servletB
        当前资源所在路径是   http://localhost:8080/web03_war_exploded/x/y/
        目标资源路径=所在资源路径+src属性值
        http://localhost:8080/web03_war_exploded/x/y/../../static/img/logo.png
        http://localhost:8080/web03_war_exploded/static/img/logo.png
        得到目标路径正是目标资源的访问路径
    -->
<img src="../../static/img/logo.png">
</body>
</html>
```

十一 MVC架构模式

> MVC（Model View Controller）是软件工程中的一种 软件架构模式 ，它把软件系统分为 模型 、 视图 和 控制器 三个基本部分。用一种业务逻辑、数据、界面显示分离的方法组织代码，将业务逻辑聚集到一个部件里面，在改进和个性化定制界面及用户交互的同时，不需要重新编写业务逻辑。

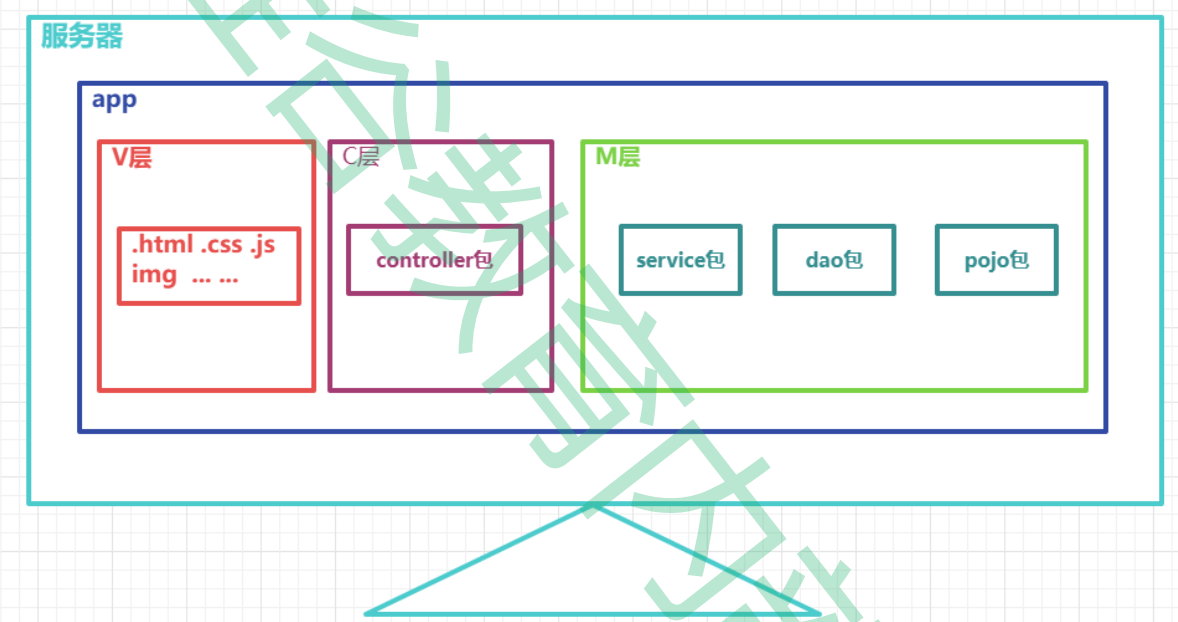
- **M**：Model 模型层，具体功能如下：
    1. 存放和数据库对应的实体类以及一些用于存储非数据库表完整相关的VO对象；
    2. 存放一些对数据进行逻辑运算操作的一些业务处理代码；
- **V**：View 视图层，具体功能如下：
    1. 存放一些视图文件相关的代码 html css js等；
    2. 在前后端分离的项目中，后端已经没有视图文件，该层次已经衍化成独立的前端项目；
- **C**：Controller 控制层,具体功能如下：
    1. 接收客户端请求，获得请求数据；

2. 将准备好的数据响应给客户端；

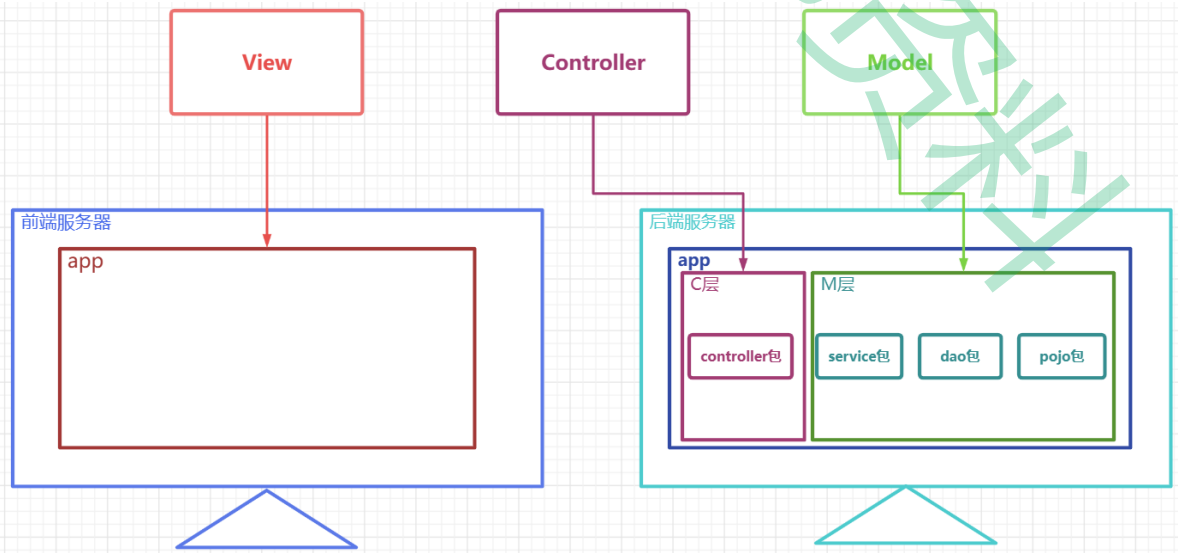
> MVC模式下项目中的常见包：

- M:
  1. 实体类包(pojo /entity /bean) 专门存放和数据库对应的实体类和一些VO对象；
  2. 数据库访问包(dao/mapper) 专门存放对数据库不同表格CURD方法封装的一些类；
  3. 服务包(service) 专门存放对数据进行业务逻辑运算的一些类；
- C:
  1. 控制层包(controller)；
- V:
  1. web目录下的视图资源 html 、css、 js 、img 等；
  2. 前端工程化后，在后端项目中已经不存在了；

> 非前后端分离的MVC：



> 前后端分离的MVC：



十二 案例开发-日程管理-第二期

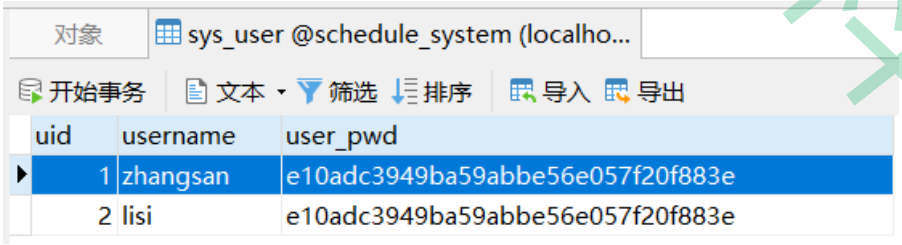
12.1 项目搭建

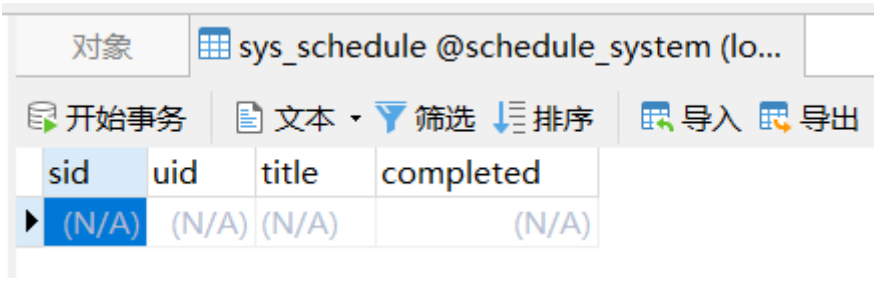
12.1.1 数据库准备

- 创建schedule_system数据库并执行如下语句。

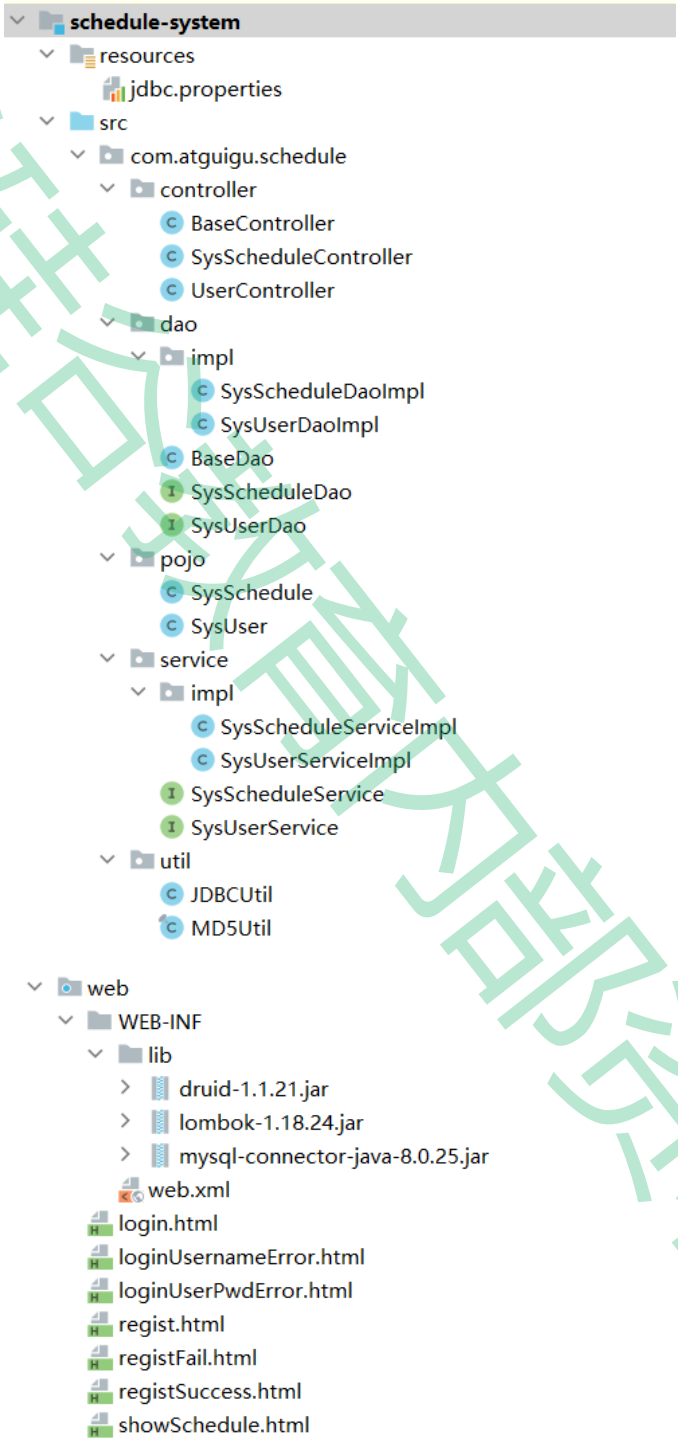
```
SET NAMES utf8mb4;
SET FOREIGN_KEY_CHECKS = 0;
-- ----------------------------
-- 创建日程表
-- ----------------------------
DROP TABLE IF EXISTS `sys_schedule`;
CREATE TABLE `sys_schedule`  (
  `sid` int NOT NULL AUTO_INCREMENT,
  `uid` int NULL DEFAULT NULL,
  `title` varchar(20) CHARACTER SET utf8mb4 COLLATE utf8mb4_0900_ai_ci NULL DEFAULT
NULL,
  `completed` int(1) NULL DEFAULT NULL,
  PRIMARY KEY (`sid`) USING BTREE
) ENGINE = InnoDB AUTO_INCREMENT = 1 CHARACTER SET = utf8mb4 COLLATE =
utf8mb4_0900_ai_ci ROW_FORMAT = Dynamic;
-- ----------------------------
-- 创建用户表
-- ----------------------------
DROP TABLE IF EXISTS `sys_user`;
CREATE TABLE `sys_user`  (
  `uid` int NOT NULL AUTO_INCREMENT,
  `username` varchar(10) CHARACTER SET utf8mb4 COLLATE utf8mb4_0900_ai_ci NULL DEFAULT
NULL,
  `user_pwd` varchar(100) CHARACTER SET utf8mb4 COLLATE utf8mb4_0900_ai_ci NULL
DEFAULT NULL,
  PRIMARY KEY (`uid`) USING BTREE,
  UNIQUE INDEX `username`(`username`) USING BTREE
) ENGINE = InnoDB CHARACTER SET = utf8mb4 COLLATE = utf8mb4_0900_ai_ci ROW_FORMAT =
Dynamic;
-- ----------------------------
-- 插入用户数据
-- ----------------------------
INSERT INTO `sys_user` VALUES (1, 'zhangsan', 'e10adc3949ba59abbe56e057f20f883e');
INSERT INTO `sys_user` VALUES (2, 'lisi', 'e10adc3949ba59abbe56e057f20f883e');
SET FOREIGN_KEY_CHECKS = 1;
```

- 获得如下表格。

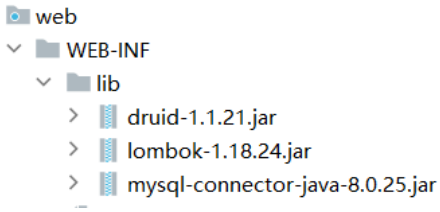




12.1.2 项目结构



12.1.3 导入依赖



12.1.4 pojo包处理

> 使用lombok处理getter、setter、equals、hashcode 构造器：

```
//-----------------------------------------------------
package com.atguigu.schedule.pojo;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import java.io.Serializable;
@AllArgsConstructor
@NoArgsConstructor
@Data
public class SysUser  implements Serializable {
    private Integer uid;
    private String username;
    private String userPwd;
}
//-----------------------------------------------------
package com.atguigu.schedule.pojo;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import java.io.Serializable;
@AllArgsConstructor
@NoArgsConstructor
@Data
public class SysSchedule implements Serializable {
    private Integer sid;
    private Integer uid;
    private String title;
    private Integer completed;
}
```

12.1.5 dao包处理

- 导入JDBCUtil连接池工具类并准备jdbc.properties配置文件：

```
package com.atguigu.schedule.util;
import com.alibaba.druid.pool.DruidDataSourceFactory;
import javax.sql.DataSource;
import java.io.IOException;
import java.io.InputStream;
import java.sql.Connection;
import java.sql.SQLException;
import java.util.Properties;
public class JDBCUtil {
    private static ThreadLocal<Connection> threadLocal =new ThreadLocal<>();
    private static DataSource dataSource;
```

```
    // 初始化连接池
    static{
        // 可以帮助我们读取.properties配置文件
        Properties properties =new Properties();
        InputStream resourceAsStream =
JDBCUtil.class.getClassLoader().getResourceAsStream("jdbc.properties");
        try {
            properties.load(resourceAsStream);
            dataSource = DruidDataSourceFactory.createDataSource(properties);
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
    /*1 向外提供连接池的方法*/
    public static DataSource getDataSource(){
        return dataSource;
    }
    /*2 向外提供连接的方法*/
    public static Connection getConnection(){
        Connection connection = threadLocal.get();
        if (null == connection) {
            try {
                connection = dataSource.getConnection();
            } catch (SQLException e) {
                throw new RuntimeException(e);
            }
            threadLocal.set(connection);
        }
        return connection;
    }
    /*定义一个归还连接的方法 (解除和ThreadLocal之间的关联关系) */
    public static void releaseConnection(){
        Connection connection = threadLocal.get();
        if (null != connection) {
            threadLocal.remove();
            // 把连接设置回自动提交的连接
            try {
                connection.setAutoCommit(true);
                // 自动归还到连接池
                connection.close();
            } catch (SQLException e) {
                throw new RuntimeException(e);
            }
        }
    }
}
```

```
driverClassName=com.mysql.cj.jdbc.Driver
url=jdbc:mysql://localhost:3306/schedule_system
username=root
password=root
initialSize=5
maxActive=10
maxWait=1000
```

- 创建BaseDao对象并复制如下代码：

```java
package com.atguigu.schedule.dao;
import com.atguigu.schedule.util.JDBCUtil;
import java.lang.reflect.Field;
import java.sql.*;
import java.time.LocalDateTime;
import java.util.ArrayList;
import java.util.List;
public class BaseDao {
    // 公共的查询方法  返回的是单个对象
    public <T> T baseQueryObject(Class<T> clazz, String sql, Object ... args) {
        T t = null;
        Connection connection = JDBCUtil.getConnection();
        PreparedStatement preparedStatement = null;
        ResultSet resultSet = null;
        int rows = 0;
        try {
            // 准备语句对象
            preparedStatement = connection.prepareStatement(sql);
            // 设置语句上的参数
            for (int i = 0; i < args.length; i++) {
                preparedStatement.setObject(i + 1, args[i]);
            }
            // 执行 查询
            resultSet = preparedStatement.executeQuery();
            if (resultSet.next()) {
                t = (T) resultSet.getObject(1);
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            if (null != resultSet) {
                try {
                    resultSet.close();
                } catch (SQLException e) {
                    e.printStackTrace();
                }
            }
            if (null != preparedStatement) {
                try {
                    preparedStatement.close();
                } catch (SQLException e) {
                    e.printStackTrace();
                }
            }
            JDBCUtil.releaseConnection();
        }
        return t;
    }
    // 公共的查询方法  返回的是对象的集合
    public <T> List<T> baseQuery(Class clazz, String sql, Object ... args){
        List<T> list =new ArrayList<>();
        Connection connection = JDBCUtil.getConnection();
        PreparedStatement preparedStatement=null;
        ResultSet resultSet =null;
        int rows = 0;
        try {
            // 准备语句对象
            preparedStatement = connection.prepareStatement(sql);
```

```
            // 设置语句上的参数
            for (int i = 0; i < args.length; i++) {
                preparedStatement.setObject(i+1,args[i]);
            }
            // 执行 查询
            resultSet = preparedStatement.executeQuery();
            ResultSetMetaData metaData = resultSet.getMetaData();
            int columnCount = metaData.getColumnCount();
            // 将结果集通过反射封装成实体类对象
            while (resultSet.next()) {
                // 使用反射实例化对象
                Object obj =clazz.getDeclaredConstructor().newInstance();
                for (int i = 1; i <= columnCount; i++) {
                    String columnName = metaData.getColumnLabel(i);
                    Object value = resultSet.getObject(columnName);
                    // 处理datetime类型字段和java.util.Data转换问题
                    if(value.getClass().equals(LocalDateTime.class)){
                        value= Timestamp.valueOf((LocalDateTime) value);
                    }
                    Field field = clazz.getDeclaredField(columnName);
                    field.setAccessible(true);
                    field.set(obj,value);
                }
                list.add((T)obj);
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            if (null !=resultSet) {
                try {
                    resultSet.close();
                } catch (SQLException e) {
                    throw new RuntimeException(e);
                }
            }
            if (null != preparedStatement) {
                try {
                    preparedStatement.close();
                } catch (SQLException e) {
                    throw new RuntimeException(e);
                }
            }
            JDBCUtil.releaseConnection();
        }
        return list;
    }
    // 通用的增删改方法
    public int baseUpdate(String sql,Object ... args) {
        // 获取连接
        Connection connection = JDBCUtil.getConnection();
        PreparedStatement preparedStatement=null;
        int rows = 0;
        try {
            // 准备语句对象
            preparedStatement = connection.prepareStatement(sql);
            // 设置语句上的参数
            for (int i = 0; i < args.length; i++) {
                preparedStatement.setObject(i+1,args[i]);
```

```
            }
            // 执行 增删改 executeUpdate
            rows = preparedStatement.executeUpdate();
            // 释放资源(可选)
        } catch (SQLException e) {
            e.printStackTrace();
        } finally {
            if (null != preparedStatement) {
                try {
                    preparedStatement.close();
                } catch (SQLException e) {
                    throw new RuntimeException(e);
                }
            }
            JDBCUtil.releaseConnection();
        }
        // 返回的是影响数据库记录数
        return rows;
    }
}
```

- dao层所有接口：

```
package com.atguigu.schedule.dao;
public interface SysUserDao {
}
```

```
package com.atguigu.schedule.dao;
public interface SysScheduleDao {
}
```

- dao层所有实现类：

```
package com.atguigu.schedule.dao.impl;
import com.atguigu.schedule.dao.BaseDao;
import com.atguigu.schedule.dao.SysUserDao;
public class SysUserDaoImpl extends BaseDao implements SysUserDao {
}
```

```
package com.atguigu.schedule.dao.impl;
import com.atguigu.schedule.dao.BaseDao;
import com.atguigu.schedule.dao.SysScheduleDao;
public class SysScheduleDaoImpl extends BaseDao implements SysScheduleDao {
}
```

12.1.6 service包处理

- 接口：

```
package com.atguigu.schedule.service;
public interface SysUserService {
}
```

```
package com.atguigu.schedule.service;
public interface SysScheduleService {
}
```

- 实现类：

```
package com.atguigu.schedule.service.impl;
import com.atguigu.schedule.service.SysUserService;
public class SysUserServiceImpl implements SysUserService {
}
```

```
package com.atguigu.schedule.service.impl;
import com.atguigu.schedule.service.SysScheduleService;
public class SysScheduleServiceImpl implements SysScheduleService {
}
```

12.1.7 controller包处理

- BaseController处理请求路径问题：

```
package com.atguigu.schedule.controller;

import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.lang.reflect.Method;

public class BaseController extends HttpServlet {
    @Override
    protected void service(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        String requestURI = req.getRequestURI();
        String[] split = requestURI.split("/");
        String methodName =split[split.length-1];
        // 通过反射获取要执行的方法
        Class clazz = this.getClass();
        try {
            Method
method=clazz.getDeclaredMethod(methodName,HttpServletRequest.class,HttpServletResponse
.class);
            // 设置方法可以访问
            method.setAccessible(true);
            // 通过反射执行代码
            method.invoke(this,req,resp);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

- 多个处理器继承BaseController：

```
package com.atguigu.schedule.controller;
import jakarta.servlet.annotation.WebServlet;
@WebServlet("/user/*")
public class UserController extends BaseController{
}
```

```
package com.atguigu.schedule.controller;
import jakarta.servlet.annotation.WebServlet;
@WebServlet("/schedule/*")
public class SysScheduleController  extends BaseController{
}
```

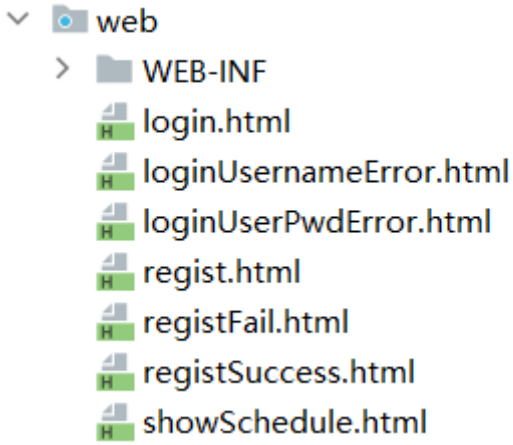
12.1.8 加密工具类的使用

- 导入MD5Util工具类：

```
package com.atguigu.schedule.util;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
public final class MD5Util {
    public static String encrypt(String strSrc) {
        try {
            char hexChars[] = { '0', '1', '2', '3', '4', '5', '6', '7', '8',
                    '9', 'a', 'b', 'c', 'd', 'e', 'f' };
            byte[] bytes = strSrc.getBytes();
            MessageDigest md = MessageDigest.getInstance("MD5");
            md.update(bytes);
            bytes = md.digest();
            int j = bytes.length;
            char[] chars = new char[j * 2];
            int k = 0;
            for (int i = 0; i < bytes.length; i++) {
                byte b = bytes[i];
                chars[k++] = hexChars[b >>> 4 & 0xf];
                chars[k++] = hexChars[b & 0xf];
            }
            return new String(chars);
        } catch (NoSuchAlgorithmException e) {
            e.printStackTrace();
            throw new RuntimeException("MD5加密出错!!!")
        }
    }
}
```

12.1.9 页面文件的导入

- 复制资源下的日程管理中的HTML到项目的web目录下即可：



12.3 业务代码

12.3.1 注册业务处理

- controller

```
package com.atguigu.schedule.controller;
import com.atguigu.schedule.pojo.SysUser;
import com.atguigu.schedule.service.SysUserService;
import com.atguigu.schedule.service.impl.SysUserServiceImpl;
import jakarta.servlet.ServletException;
import jakarta.servlet.annotation.WebServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import java.io.IOException;
@WebServlet("/user/*")
public class SysUserController  extends BaseContoller {
    private SysUserService userService =new SysUserServiceImpl();
    /**
     * 接收用户注册请求的业务接口实现( 业务接口 不是java中的interface  )
     * @param req
     * @param resp
     * @throws ServletException
     * @throws IOException
     */
    protected void regist(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        // 1 接收客户端提交的参数
        String username = req.getParameter("username");
        String userPwd = req.getParameter("userPwd");
        // 2 调用服务层方法,完成注册功能
            //将参数放入一个SysUser对象中,在调用regist方法时传入
        SysUser sysUser =new SysUser(null,username,userPwd);
        int rows =userService.regist(sysUser);
        // 3 根据注册结果(成功  失败) 做页面跳转
        if(rows>0){
            resp.sendRedirect("/registSuccess.html");
        }else{
            resp.sendRedirect("/registFail.html");
        }
    }
}
```

- service

```
package com.atguigu.schedule.service;
import com.atguigu.schedule.pojo.SysUser;
public interface SysUserService {
    /**
     * 用户完成注册的业务方法
     * @param registUser 用于保存注册用户名和密码的对象
     * @return 注册成功返回>0的整数,否则返回0
     */
    int regist(SysUser registUser);
}
```

```
package com.atguigu.schedule.service.impl;
import com.atguigu.schedule.dao.SysUserDao;
import com.atguigu.schedule.dao.impl.SysUserDaoImpl;
import com.atguigu.schedule.pojo.SysUser;
import com.atguigu.schedule.service.SysUserService;
import com.atguigu.schedule.util.MD5Util;
public class SysUserServiceImpl  implements SysUserService {
    private SysUserDao  userDao =new SysUserDaoImpl();
    @Override
    public int regist(SysUser sysUser) {
        // 将用户的明文密码转换为密文密码
        sysUser.setUserPwd(MD5Util.encrypt(sysUser.getUserPwd()));
        // 调用DAO 层的方法  将sysUser信息存入数据库
        return userDao.addSysUser(sysUser);
    }
}
```

- dao

```
package com.atguigu.schedule.dao;
import com.atguigu.schedule.pojo.SysUser;
public interface SysUserDao {
    /**
     * 向数据库中增加一条用户记录的方法
     * @param sysUser 要增加的记录的username和user_pwd字段以SysUser实体类对象的形式接收
     * @return 增加成功返回1 增加失败返回0
     */
    int addSysUser(SysUser sysUser);
}
```

```
package com.atguigu.schedule.dao.impl;
import com.atguigu.schedule.dao.BaseDao;
import com.atguigu.schedule.dao.SysUserDao;
import com.atguigu.schedule.pojo.SysUser;
public class SysUserDaoImpl extends BaseDao implements SysUserDao {
    @Override
    public int addSysUser(SysUser sysUser) {
        String sql ="insert into sys_user values(DEFAULT,?,?)";
        return baseUpdate(sql,sysUser.getUsername(),sysUser.getUserPwd());
    }
}
```

12.3.2 登录业务处理

- controller

```
package com.atguigu.schedule.controller;
import com.atguigu.schedule.pojo.SysUser;
import com.atguigu.schedule.service.SysUserService;
import com.atguigu.schedule.service.impl.SysUserServiceImpl;
import com.atguigu.schedule.util.MD5Util;
import jakarta.servlet.ServletException;
import jakarta.servlet.annotation.WebServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import java.io.IOException;
@WebServlet("/user/*")
public class SysUserController  extends BaseContoller {
    private SysUserService userService =new SysUserServiceImpl();
    /**
     * 登录业务接口实现
     * @param req
     * @param resp
     * @throws ServletException
     * @throws IOException
     */
    protected void login(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        //1 接收用户名和密码
        String username = req.getParameter("username");
        String userPwd = req.getParameter("userPwd");
        //2 调用服务层方法,根据用户名查询用户信息
        SysUser loginUser =userService.findByUsername(username);
        if(null == loginUser){
            // 跳转到用户名有误提示页
            resp.sendRedirect("/loginUsernameError.html");
        }else if(! MD5Util.encrypt(userPwd).equals(loginUser.getUserPwd())){
            //3 判断密码是否匹配
            // 跳转到密码有误提示页
            resp.sendRedirect("/loginUserPwdError.html");
        }else{
            //4 跳转到首页
            resp.sendRedirect("/showSchedule.html");
        }
    }
}
```

- service

```
package com.atguigu.schedule.service;
import com.atguigu.schedule.pojo.SysUser;
public interface SysUserService {
     /**
     * 根据用户名获得完整用户信息的方法
     * @param username 要查询的用户名
     * @return 如果找到了返回SysUser对象,找不到返回null
     */
    SysUser findByUsername(String username);
}
```

```
package com.atguigu.schedule.service.impl;
import com.atguigu.schedule.dao.SysUserDao;
```

```java
import com.atguigu.schedule.dao.impl.SysUserDaoImpl;
import com.atguigu.schedule.pojo.SysUser;
import com.atguigu.schedule.service.SysUserService;
import com.atguigu.schedule.util.MD5Util;
public class SysUserServiceImpl implements SysUserService {
    private SysUserDao userDao =new SysUserDaoImpl();
    @Override
    public SysUser findByUsername(String username) {
        // 调用服务层方法,继续查询
        return userDao.findByUsername(username);
    }
}
```

- dao

```java
package com.atguigu.schedule.dao;
import com.atguigu.schedule.pojo.SysUser;
public interface SysUserDao {
     /**
     * 根据用户名获得完整用户信息的方法
     * @param username 要查询的用户名
     * @return 如果找到了返回SysUser对象,找不到返回null
     */
    SysUser findByUsername(String username);
}
```

```java
package com.atguigu.schedule.dao.impl;
import com.atguigu.schedule.dao.BaseDao;
import com.atguigu.schedule.dao.SysUserDao;
import com.atguigu.schedule.pojo.SysUser;
import java.util.List;
public class SysUserDaoImpl extends BaseDao implements SysUserDao {
    @Override
    public SysUser findByUsername(String username) {
        String sql ="select uid,username, user_pwd userPwd from sys_user where
username = ?";
        List<SysUser> userList = baseQuery(SysUser.class, sql, username);
        return  null != userList&& userList.size()>0? userList.get(0):null;
    }
}
```

第六章 会话_过滤器_监听器_Ajax

一 会话

1.1 会话管理概述

1.1.1 为什么需要会话管理

> HTTP是无状态协议：

- 无状态就是不保存状态，即无状态协议(stateless)，HTTP协议自身不对请求和响应之间的通信状态进行保存，也就是说，在HTTP协议这个级别，协议对于发送过的请求或者响应都不做持久化处理；
- 简单理解：浏览器发送请求，服务器接收并响应，但是服务器不记录请求来自哪个浏览器，服务器没记录浏览器的特征，就是客户端的状态；

> 举例：张三去一家饭馆点了几道菜，觉得味道不错，第二天又去了，对老板说，还点上次的那几道菜。

- 无状态：老板没有记录张三是否来过，更没有记录上次他点了那些菜，张三只能重新再点一遍；
- 有状态：老板把每次来吃饭的用户都做好记录，查阅一下之前的记录，查到了张三之前的菜单，直接下单；

1.1.2 会话管理实现的手段

> Cookie和Session配合解决：

- cookie是在客户端保留少量数据的技术，主要通过响应头向客户端响应一些客户端要保留的信息；
- session是在服务端保留更多数据的技术，主要通过服务端HttpSession对象保存一些和客户端相关的信息；
- cookie和session配合记录请求状态；

> 举例：张三去银行办业务。

- 张三第一次去某个银行办业务，银行会为张三开户(Session)，并向张三发放一张银行卡(Cookie)；
- 张三后面每次去银行，就可以携带之间的银行卡(Cookie)，银行根据银行卡找到之前张三的账户(Session)；

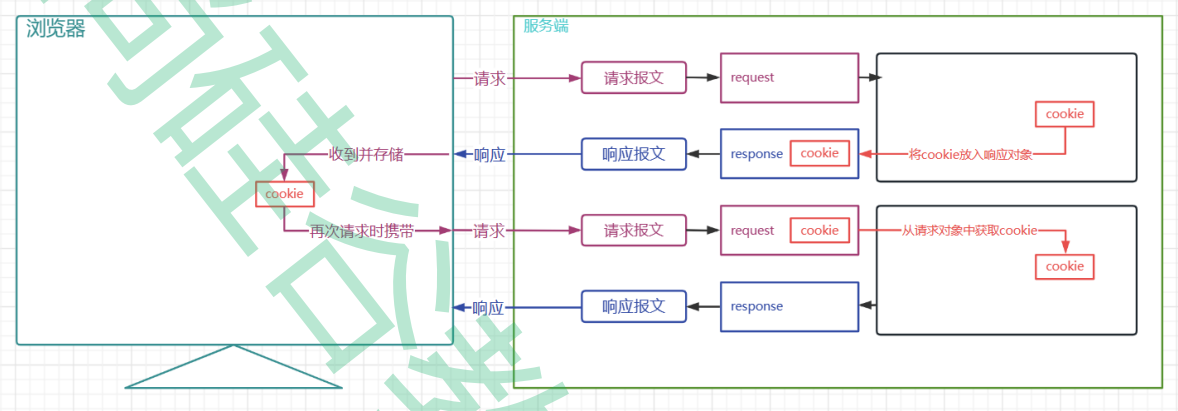
1.2 Cookie

1.2.1 Cookie概述

> Cookie是一种客户端会话技术，Cookie由服务端产生，它是服务器存放在浏览器的一小份数据，浏览器以后每次访问该服务器的时候都会将这小份数据携带到服务器去：

- 服务端创建Cookie，将Cookie放入响应对象中，Tomcat容器将Cookie转化为set-cookie响应头，响应给客户端；
- 客户端在收到Cookie的响应头时，在下次请求该服务的资源时，会以cookie请求头的形式携带之前收到的Cookie；
- Cookie是一种键值对格式的数据，从tomcat8.5开始可以保存中文，但是不推荐；
- 由于Cookie是存储于客户端的数据，比较容易暴露，一般不存储一些敏感或者影响安全的数据；

> 原理图：



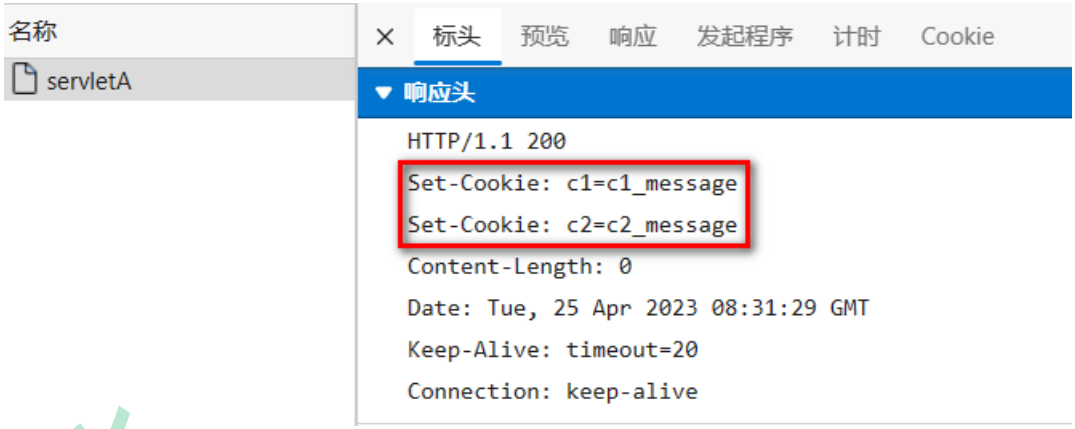
> 应用场景举例：

1. 记录用户名；
2. 保存电影播放进度；
3. ... ...

1.2.2 Cookie的使用

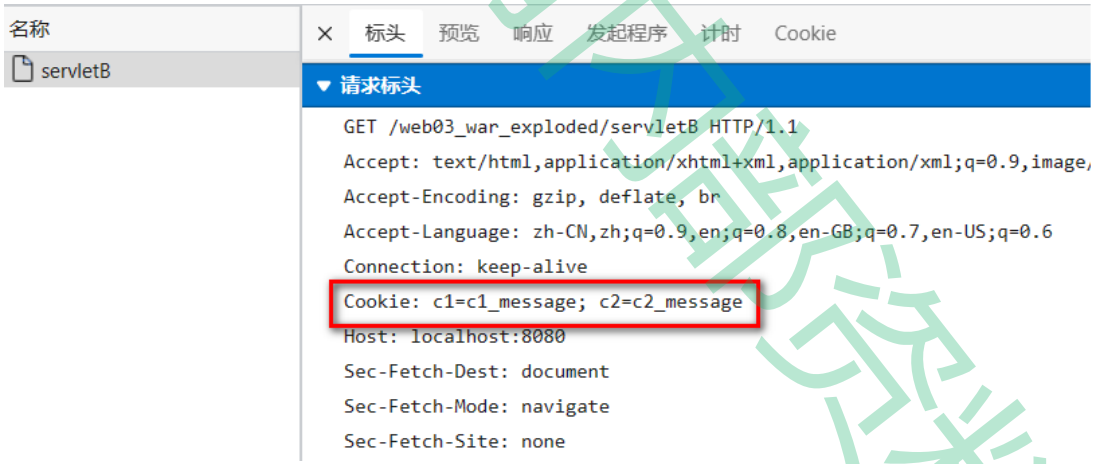
> servletA向响应中增加Cookie：

```
@WebServlet("/servletA")
public class ServletA extends HttpServlet {
    @Override
    protected void service(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        // 创建Cookie
        Cookie cookie1 =new Cookie("c1","c1_message");
        Cookie cookie2 =new Cookie("c2","c2_message");
        // 将cookie放入响应对象
        resp.addCookie(cookie1);
        resp.addCookie(cookie2);
    }
}
```



servletB从请求中读取Cookie：

```
@WebServlet("/servletB")
public class ServletB extends HttpServlet {
    @Override
    protected void service(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        //获取请求中的cookie
        Cookie[] cookies = req.getCookies();
        //迭代cookies数组
        if (null != cookies && cookies.length!= 0) {
            for (Cookie cookie : cookies) {
                System.out.println(cookie.getName()+":"+cookie.getValue());
            }
        }
    }
}
```



1.2.2 Cookie的时效性

默认情况下Cookie的有效期是一次会话范围内，我们可以通过Cookie的setMaxAge()方法让Cookie持久化保存到浏览器上。

- 会话级Cookie：
  - 服务器端并没有明确指定Cookie的存在时间；
  - 在浏览器端，Cookie数据存在于内存中；
  - 只要浏览器还开着，Cookie数据就一直都在；
  - 浏览器关闭，内存中的Cookie数据就会被释放；
- 持久化Cookie：

- 服务器端明确设置了Cookie的存在时间；
- 在浏览器端，Cookie数据会被保存到硬盘上；
- Cookie在硬盘上存在的时间根据服务器端限定的时间来管控，不受浏览器关闭的影响；
- 持久化Cookie到达了预设的时间会被释放；

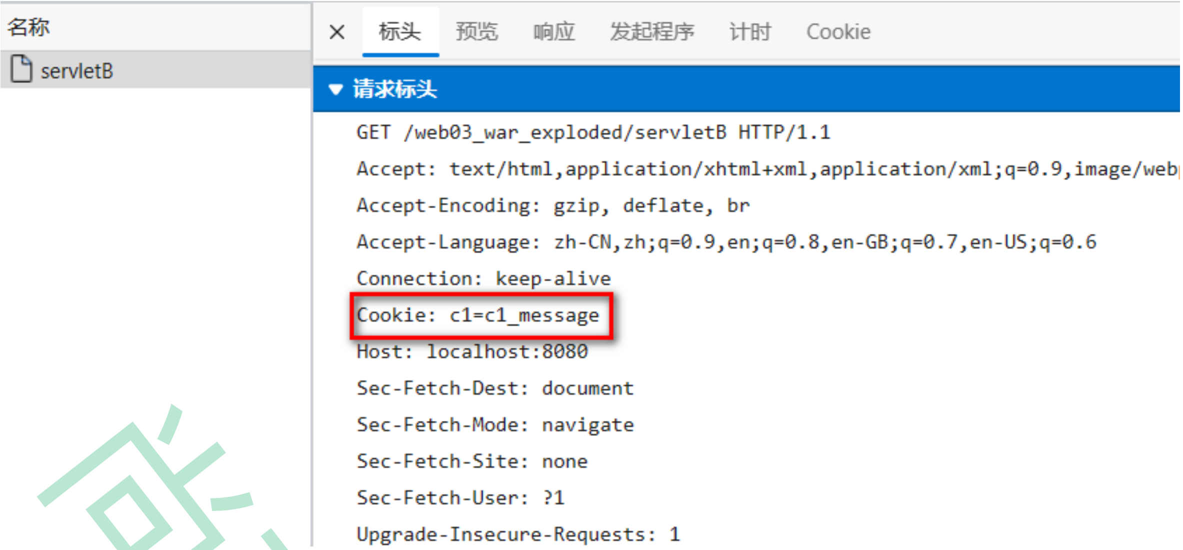
> cookie.setMaxAge(int expiry)参数单位是秒，表示cookie的持久化时间，如果设置参数为0，表示将浏览器中保存的该cookie删除。

- servletA设置一个Cookie为持久化cookie。

```
@WebServlet("/servletA")
public class ServletA extends HttpServlet {
    @Override
    protected void service(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        // 创建Cookie
        Cookie cookie1 =new Cookie("c1","c1_message");
        cookie1.setMaxAge(60);
        Cookie cookie2 =new Cookie("c2","c2_message");
        // 将cookie放入响应对象
        resp.addCookie(cookie1);
        resp.addCookie(cookie2);
    }
}
```

- servletB接收Cookie，浏览器中间发生一次重启再请求servletB测试。

```
@WebServlet("/servletB")
public class ServletB extends HttpServlet {
    @Override
    protected void service(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        //获取请求中的cookie
        Cookie[] cookies = req.getCookies();
        //迭代cookies数组
        if (null != cookies && cookies.length!= 0) {
            for (Cookie cookie : cookies) {
                System.out.println(cookie.getName()+":"+cookie.getValue());
            }
        }
    }
}
```

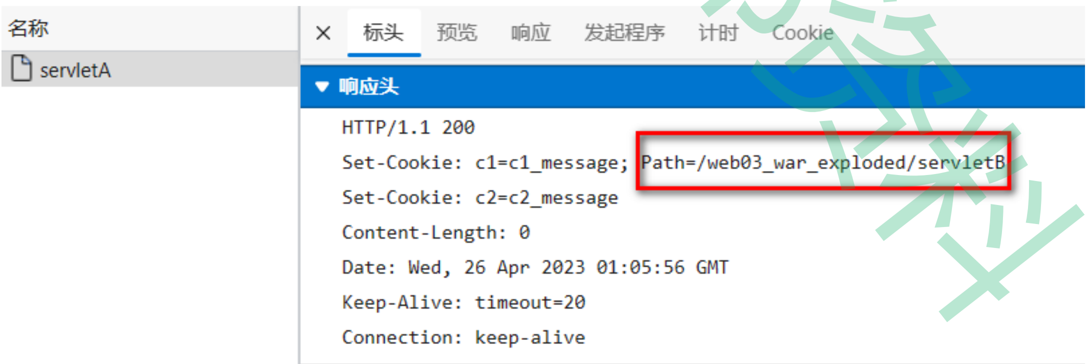


1.2.3 Cookie的提交路径

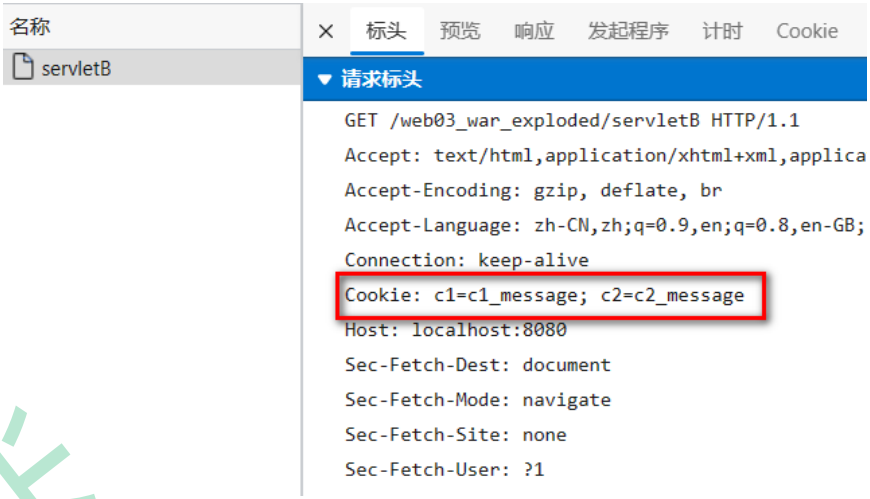
> 访问互联网资源时不需要每次都需要把所有Cookie带上。访问不同的资源时，可以携带不同的cookie，可以通过Cookie的setPath(String path) 对Cookie的路径进行设置。

- 从ServletA中获取cookie。

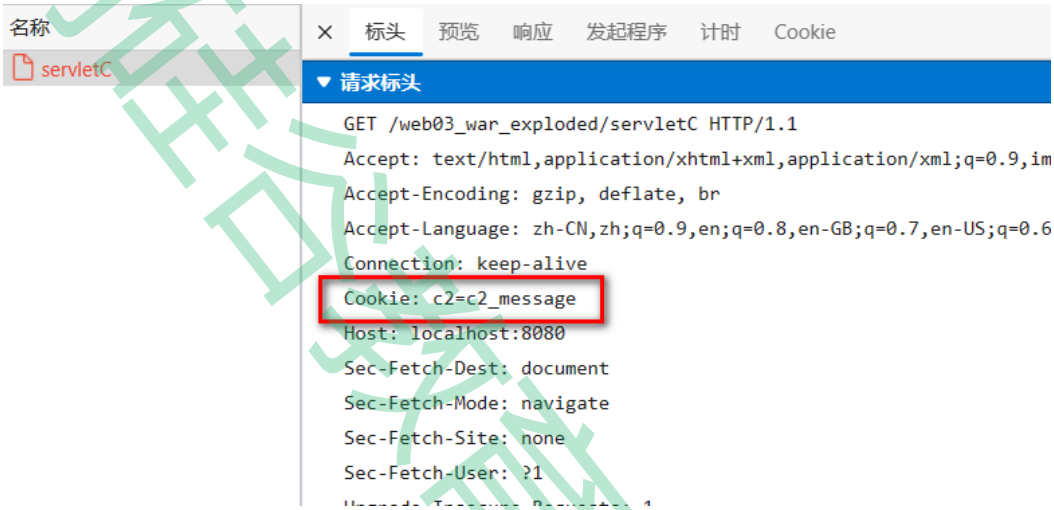
```java
public class ServletA extends HttpServlet {
    @Override
    protected void service(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
        // 创建Cookie
        Cookie cookie1 =new Cookie("c1","c1_message");
        // 设置cookie的提交路径
        cookie1.setPath("/web03_war_exploded/servletB");
        Cookie cookie2 =new Cookie("c2","c2_message");
        // 将cookie放入响应对象
        resp.addCookie(cookie1);
        resp.addCookie(cookie2);
    }
}
```



- 向ServletB请求时携带携带了 c1。



- 向其他资源请求时就不携带c1了。



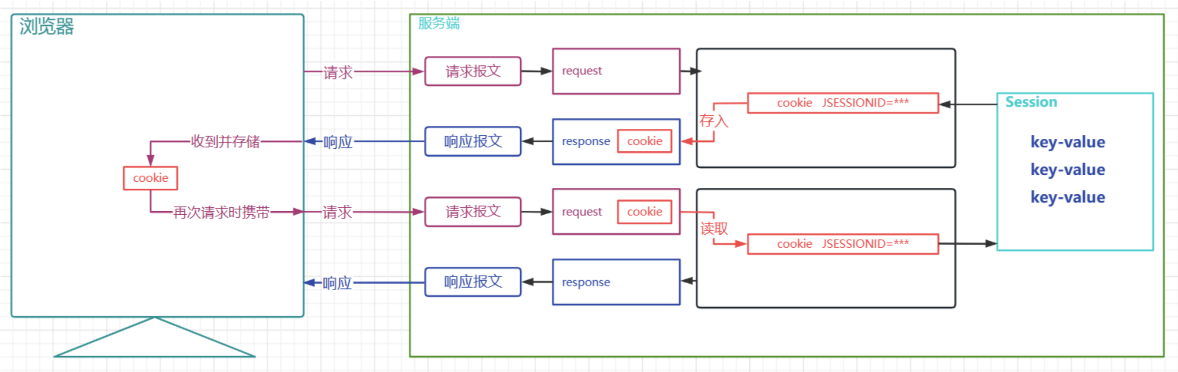
1.3 Session

1.3.1 HttpSession概述

> HttpSession是一种保留更多信息在服务端的一种技术，服务器会为每一个客户端开辟一块内存空间，即session对象.。客户端在发送请求时，都可以使用自己的session。这样服务端就可以通过session来记录某个客户端的状态了。

- 服务端在为客户端创建session时，会同时将session对象的id，即JSESSIONID以Cookie的形式放入响应对象；
- 后端创建完session后，客户端会收到一个特殊的Cookie，叫做JSESSIONID；
- 客户端下一次请求时携带JSESSIONID，后端收到后，根据JSESSIONID找到对应的session对象；
- 通过该机制，服务端通过session就可以存储一些专门针对某个客户端的信息了；
- session也是域对象(后续详细讲解)；

> 原理图如下：



> 应用场景：

1. 记录用户的登录状态：

   用户登录后，将用户的账号等敏感信息存入session；

2. 记录用户操作的历史：

   例如记录用户的访问痕迹，用户的购物车信息等临时性的信息；

1.3.2 HttpSession的使用

> 用户提交form表单到ServletA，携带用户名，ServletA获取session 将用户名存到session，用户再请求其他任意Servlet，获取之间存储的用户。

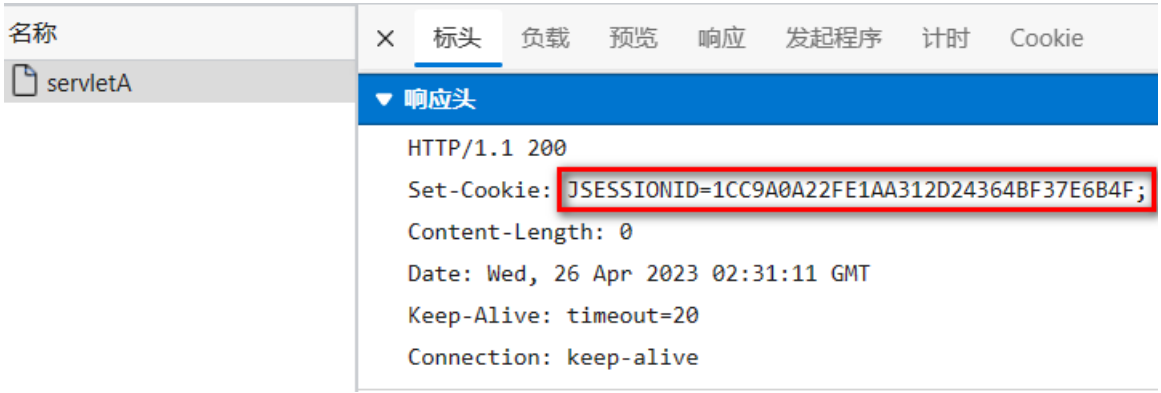
- 定义表单页，提交用户名。

```html
    <form action="servletA" method="post">
        用户名：
        <input type="text" name="username">
        <input type="submit" value="提交">
    </form>
```

- 定义ServletA，将用户名存入session。

```java
@WebServlet("/servletA")
public class ServletA extends HttpServlet {
    @Override
    protected void service(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        // 获取请求中的参数
        String username = req.getParameter("username");
        // 获取session对象
        HttpSession session = req.getSession();
         // 获取Session的ID
        String jSessionId = session.getId();
        System.out.println(jSessionId);
        // 判断session是不是新创建的session
        boolean isNew = session.isNew();
        System.out.println(isNew);
        // 向session对象中存入数据
        session.setAttribute("username",username);
    }
}
```

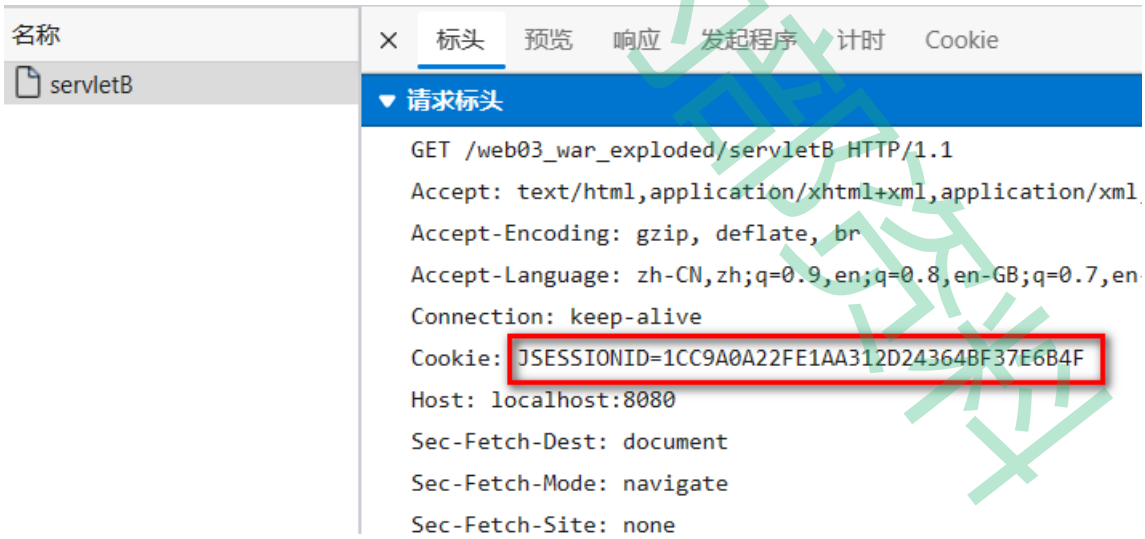
- 响应中收到了一个JSESSIONID的Cookie。



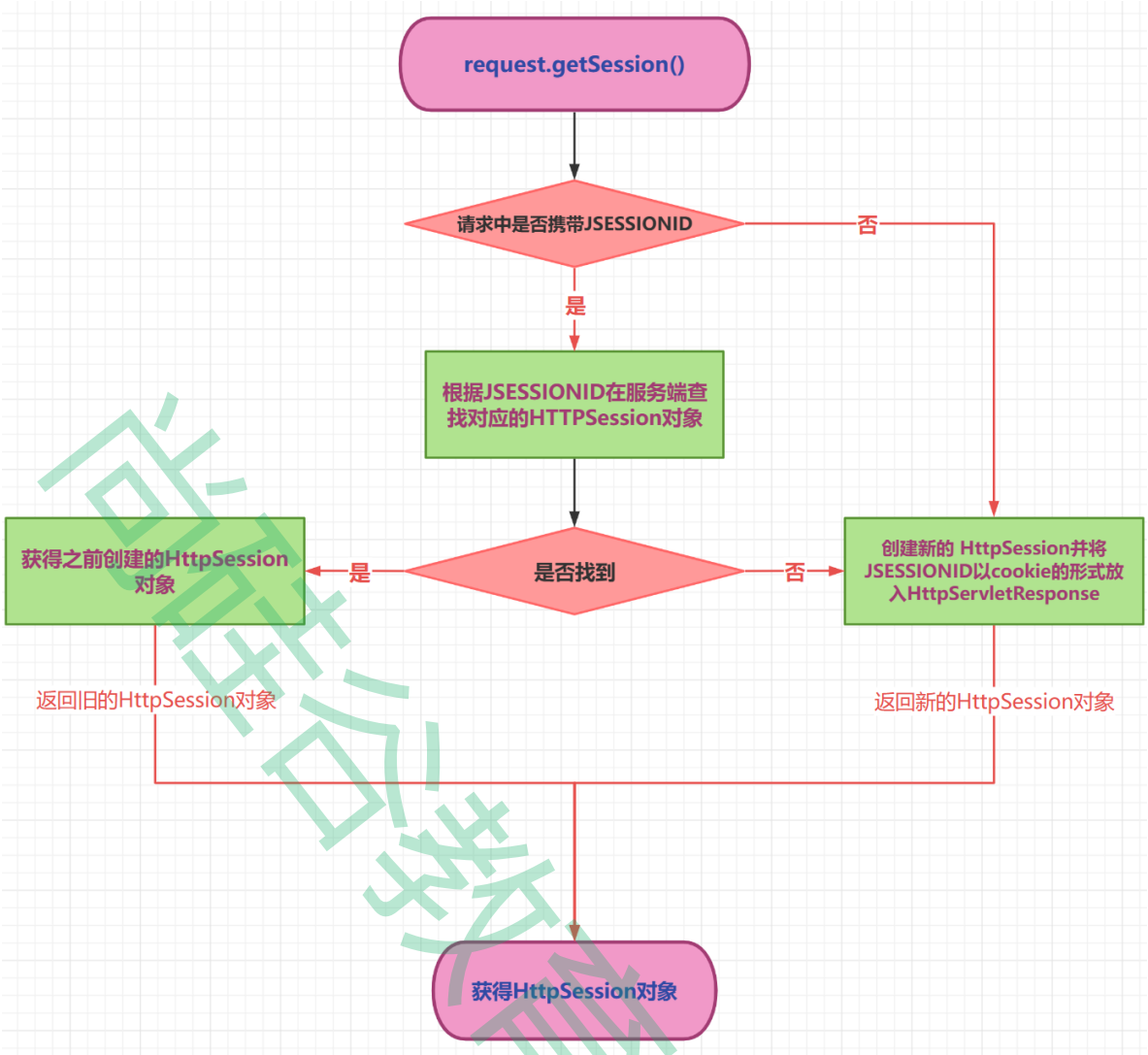
- 定义其他Servlet，从session中读取用户名。

```
@WebServlet("/servletB")
public class ServletB extends HttpServlet {
    @Override
    protected void service(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        // 获取session对象
        HttpSession session = req.getSession();
         // 获取Session的ID
        String jSessionId = session.getId();
        System.out.println(jSessionId);
        // 判断session是不是新创建的session
        boolean isNew = session.isNew();
        System.out.println(isNew);
        // 从session中取出数据
        String username = (String)session.getAttribute("username");
        System.out.println(username);
    }
}
```

- 请求中携带了一个JSESSIONID的Cookie。



> getSession方法的处理逻辑：



1.3.3 HttpSession时效性

> 为什么要设置session的时效?

- 用户量很大之后，Session对象相应的也要创建很多。如果一味创建不释放，那么服务器端的内存迟早要被耗尽；
- 客户端关闭行为无法被服务端直接侦测，或者客户端较长时间不操作也经常出现，类似这些的情况，就需要对session的时限进行设置了；

> 默认的session最大闲置时间(两次使用同一个session中的间隔时间) 在Tomcat/conf/web.xml配置为30分钟。

```
632     <!-- ==================== Default Session Configuration ================ -->
633     <!-- You can set the default session timeout (in minutes) for all newly   -->
634     <!-- created sessions by modifying the value below.                       -->
635
636     <session-config>
637         <session-timeout>30</session-timeout>
638     </session-config>
```

> 我们可以自己在当前项目的web.xml对最大闲置时间进行重新设定。

> 也可以通过HttpSession的API 对最大闲置时间进行设定。

```
// 设置最大闲置时间
session.setMaxInactiveInterval(60);
```

> 也可以直接让session失效。

```
// 直接让session失效
session.invalidate();
```

1.4 三大域对象

1.4.1 域对象概述

> 域对象：一些用于存储数据和传递数据的对象。传递数据不同的范围，我们称之为不同的域。不同的域对象代表不同的域，共享数据的范围也不同。

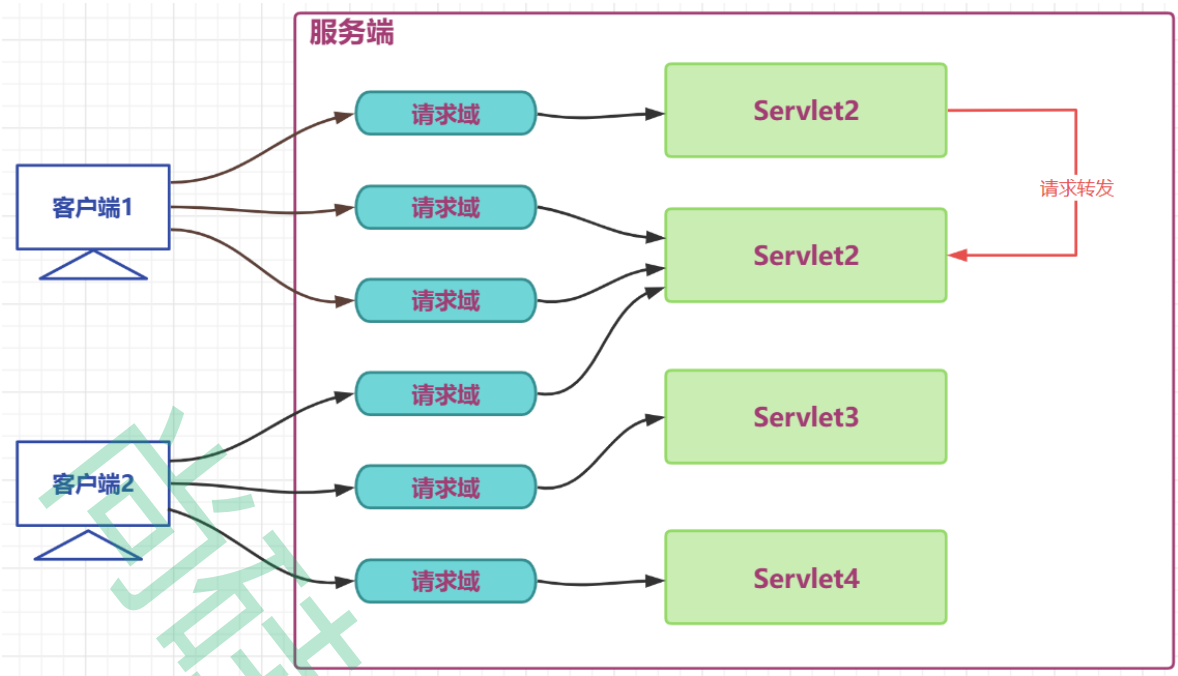
- web项目中，我们一定要熟练使用的域对象分别是：请求域、会话域、应用域；
- 请求域对象是HttpServletRequest，传递数据的范围是一次请求之内及请求转发；
- 会话域对象是HttpSession，传递数据的范围是一次会话之内，可以跨多个请求；
- 应用域对象是ServletContext，传递数据的范围是本应用之内，可以跨多个会话；

> 生活举例：热水壶摆放位置不同。使用的范围就不同。

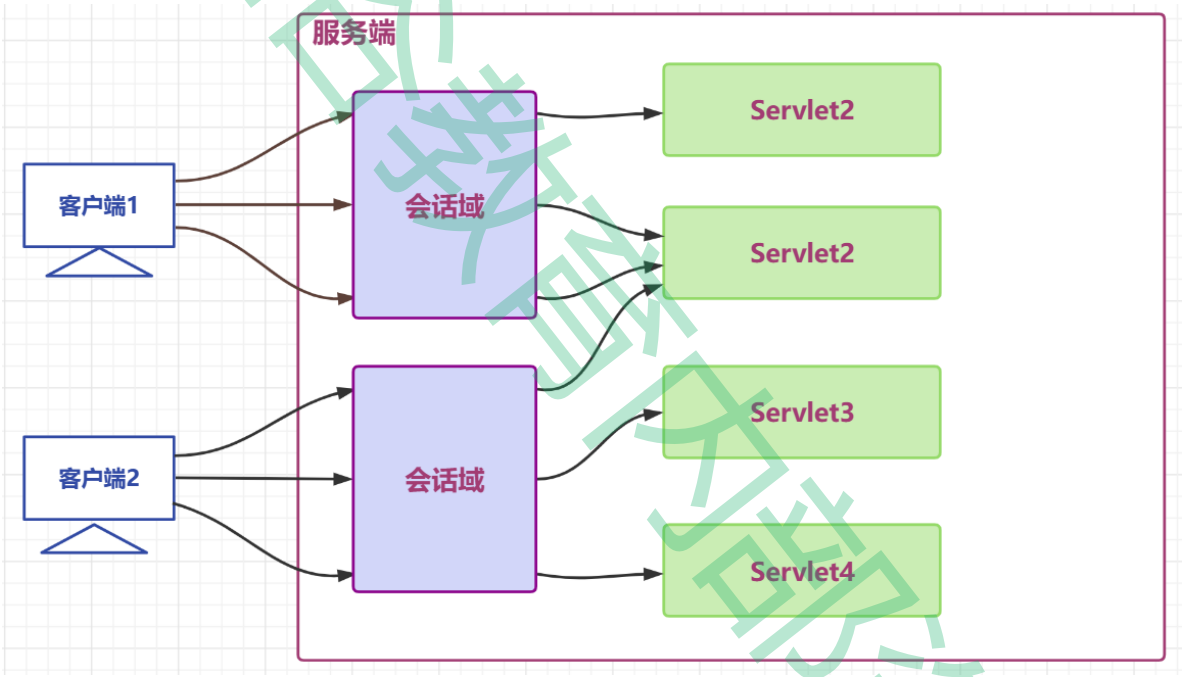
1. 摆在张三工位下，就只有张三一个人能用；
2. 摆在办公室的公共区，办公室内的所有人都可以用；
3. 摆在楼层的走廊区，该楼层的所有人都可以用；

> 三大域对象的数据作用范围图解：

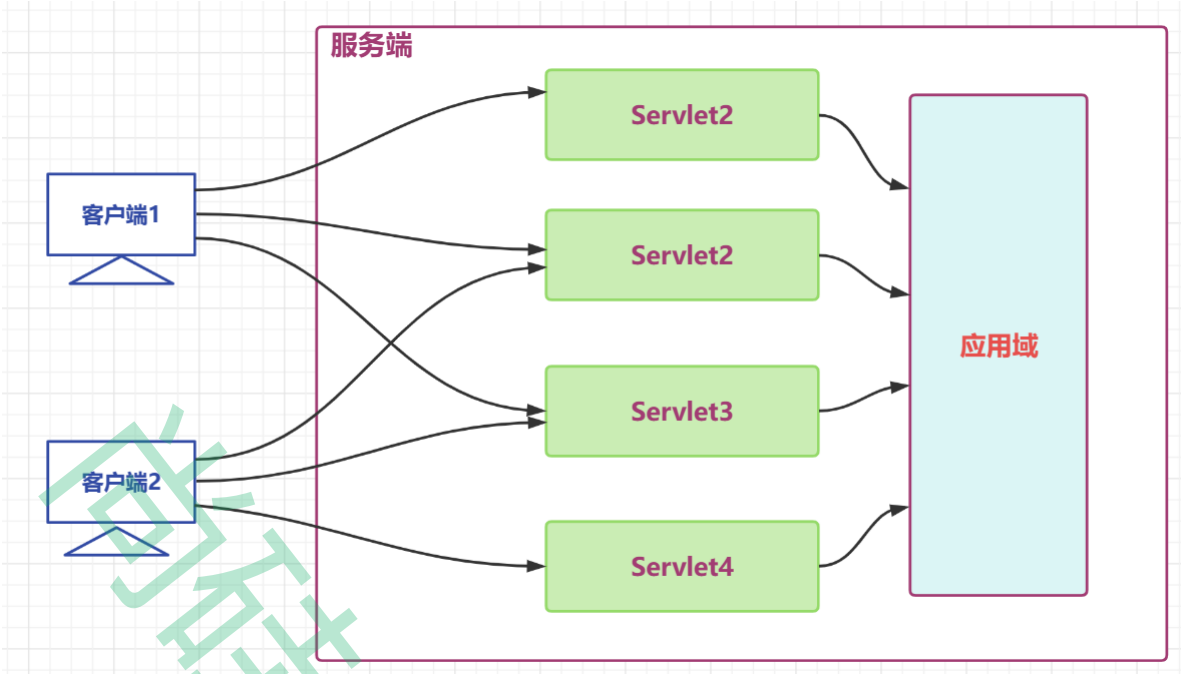
- 请求域



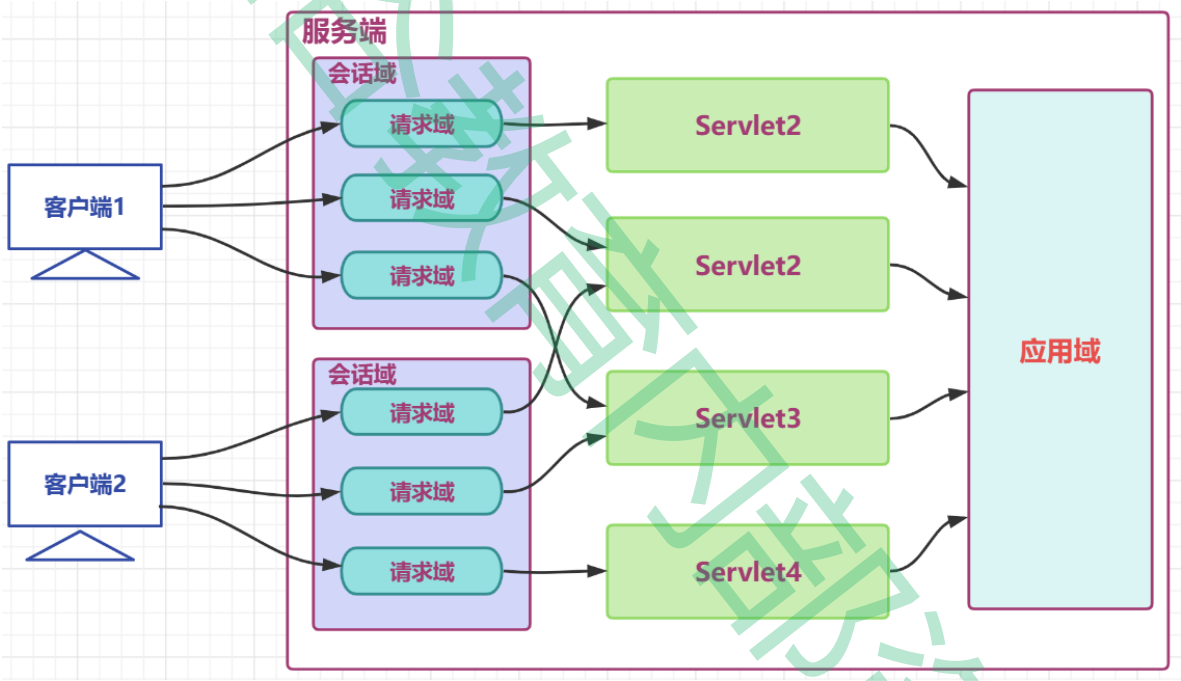
- 会话域



- 应用域



- 所有域在一起



1.4.2 域对象的使用

域对象的API：

API	功能
void setAttribute(String name,String value)	向域对象中添加/修改数据
Object getAttribute(String name);	从域对象中获取数据
void removeAttribute(String name);	移除域对象中的数据

API测试：

- ServletA向三大域中放入数据。

```
@WebServlet("/servletA")
public class ServletA extends HttpServlet {
    @Override
    protected void service(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        // 向请求域中放入数据
        req.setAttribute("request","request-message");
        //req.getRequestDispatcher("servletB").forward(req,resp);
        // 向会话域中放入数据
        HttpSession session = req.getSession();
        session.setAttribute("session","session-message");
        // 向应用域中放入数据
        ServletContext application = getServletContext();
        application.setAttribute("application","application-message");
    }
}
```

- ServletB从三大于中取出数据。

```
@WebServlet("/servletB")
public class ServletB extends HttpServlet {
    @Override
    protected void service(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        // 从请求域中获取数据
        String reqMessage =(String)req.getAttribute("request");
        System.out.println(reqMessage);
        // 从会话域中获取数据
        HttpSession session = req.getSession();
        String sessionMessage =(String)session.getAttribute("session");
        System.out.println(sessionMessage);
        // 从应用域中获取数据
        ServletContext application = getServletContext();
        String applicationMessage =(String)application.getAttribute("application");
        System.out.println(applicationMessage);
    }
}
```

- 请求转发时，请求域可以传递数据。`请求域内一般放本次请求业务有关的数据,如:查询到的所有的部门信息`；
- 同一个会话内，不用请求转发，会话域可以传递数据。`会话域内一般放本次会话的客户端有关的数据,如:当前客户端登录的用户`；
- 同一个APP内，不同的客户端，应用域可以传递数据。`应用域内一般放本程序应用有关的数据如:Spring框架的IOC容器`；

二 过滤器

2.1 过滤器概述

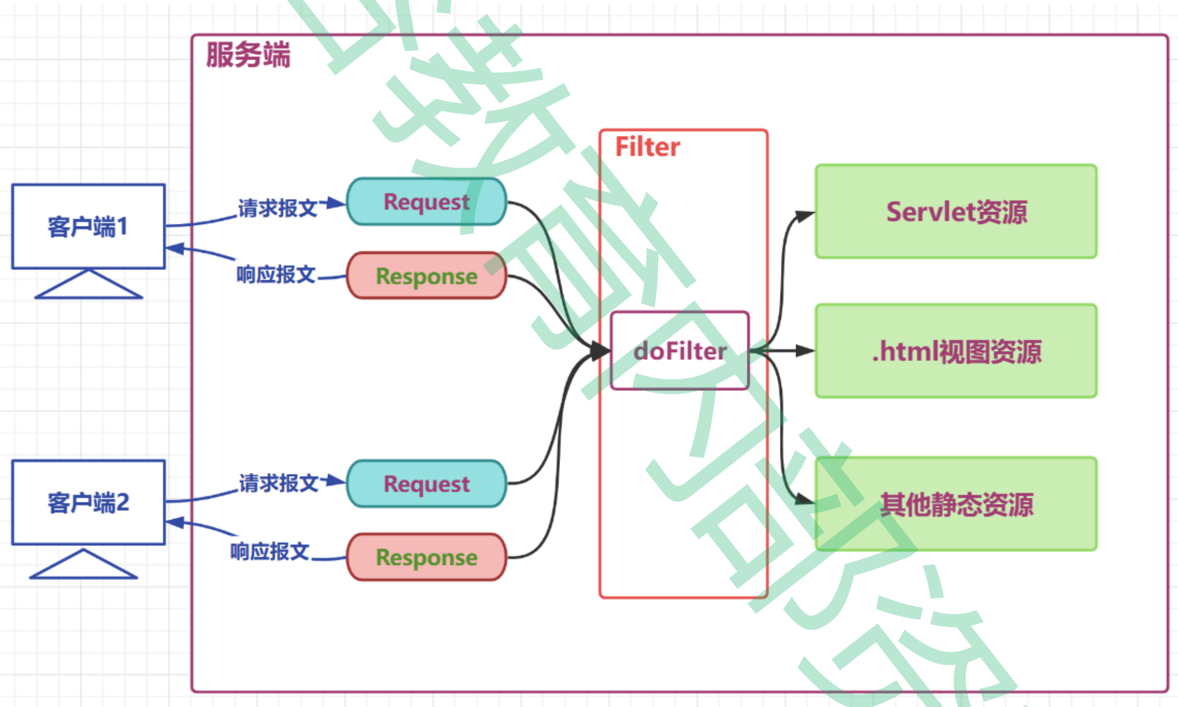
> Filter，即过滤器，是JAVAEE技术规范之一，作为目标资源的请求进行过滤的一套技术规范，是Java Web项目中`最为实用的技术之一`。

- Filter接口定义了过滤器的开发规范，所有的过滤器都要实现该接口；
- Filter的工作位置是项目中所有目标资源之前，容器在创建HttpServletRequest和HttpServletResponse对象后，会先调用Filter的doFilter方法；
- Filter的doFilter方法可以控制请求是否继续，如果放行，则请求继续，如果拒绝，则请求到此为止，由过滤器本身做出响应；
- Filter不仅可以对请求做出过滤，也可以在目标资源做出响应前，对响应再次进行处理；
- Filter是GOF中责任链模式的典型案例；
- Filter的常用应用包括但不限于：登录权限检查、解决网站乱码、过滤敏感字符、日志记录、性能分析... ...；

> 生活举例：公司前台，停车场安保，地铁验票闸机... ...

- 公司前台对来访人员进行审核，如果是游客则拒绝进入公司，如果是客户则放行。客户离开时提醒客户不要遗忘物品；
- 停车场保安对来访车辆进行控制，如果没有车位拒绝进入。如果有车位，发放停车卡并放行，车辆离开时收取请车费；
- 地铁验票闸机在人员进入之前检查票，没票拒绝进入，有票验票后放行，人员离开时同样验票；

> 过滤器工作位置图解：



> Filter接口API：

- 源码

```
package jakarta.servlet;
import java.io.IOException;
public interface Filter {
    default public void init(FilterConfig filterConfig) throws ServletException {
    }
    public void doFilter(ServletRequest request, ServletResponse response, FilterChain
chain)
            throws IOException, ServletException;
    default public void destroy() {
    }
}
```

- API目标

API	目标
default public void init(FilterConfig filterConfig)	初始化方法,由容器调用并传入初始配置信息filterConfig对象
public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)	过滤方法,核心方法,过滤请求,决定是否放行,响应之前的其他处理等都在该方法中
default public void destroy()	销毁方法,容器在回收过滤器对象之前调用的方法

2.2 过滤器使用

> 目标：开发一个日志记录过滤器。

- 用户请求到达目标资源之前，记录用户的请求资源路径；
- 响应之前记录本次请求目标资源运算的耗时；
- 可以选择将日志记录进入文件，为了方便测试，这里将日志直接在控制台打印；

> 定义一个过滤器类，编写功能代码：

```
public class LoggingFilter  implements Filter {
    private SimpleDateFormat dateFormat =new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
    @Override
    public void doFilter(ServletRequest servletRequest, ServletResponse
servletResponse, FilterChain filterChain) throws IOException, ServletException {
        // 参数父转子
        HttpServletRequest request =(HttpServletRequest)  servletRequest;
        HttpServletResponse  response =(HttpServletResponse)  servletResponse;
        // 拼接日志文本
        String requestURI = request.getRequestURI();
        String time = dateFormat.format(new Date());
        String beforeLogging =requestURI+"在"+time+"被请求了";
        // 打印日志
        System.out.println(beforeLogging);
        // 获取系统时间
        long t1 = System.currentTimeMillis();
        // 放行请求
        filterChain.doFilter(request,response);
```

```
        // 获取系统时间
        long t2 = System.currentTimeMillis();
        //  拼接日志文本
        String afterLogging =requestURI+"在"+time+"的请求耗时:"+(t2-t1)+"毫秒";
        // 打印日志
        System.out.println(afterLogging);
    }
}
```

- 说明
  - doFilter方法中的请求和响应对象是以父接口的形式声明的，实际传入的实参就是HttpServletRequest和HttpServletResponse子接口级别的，可以安全强转；
  - filterChain.doFilter(request,response);这行代码的功能是放行请求，如果没有这一行代码，则请求到此为止；
  - filterChain.doFilter(request,response);在放行时需要传入request和response，意味着请求和响应对象要继续传递给后续的资源，这里没有产生新的request和response对象；

> 定义两个Servlet作为目标资源：

- ServletA

```
@WebServlet(urlPatterns = "/servletA",name = "servletAName")
public class ServletA extends HttpServlet {
    @Override
    protected void service(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        // 处理器请求
        System.out.println("servletA处理请求的方法,耗时10毫秒");
        // 模拟处理请求耗时
        try {
            Thread.sleep(10);
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
    }
}
```

- ServletB

```
@WebServlet(urlPatterns = "/servletB", name = "servletBName")
public class ServletB extends HttpServlet {
    @Override
    protected void service(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        // 处理器请求
        System.out.println("servletB处理请求的方法,耗时15毫秒");
        // 模拟处理请求耗时
        try {
            Thread.sleep(15);
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
    }
}
```

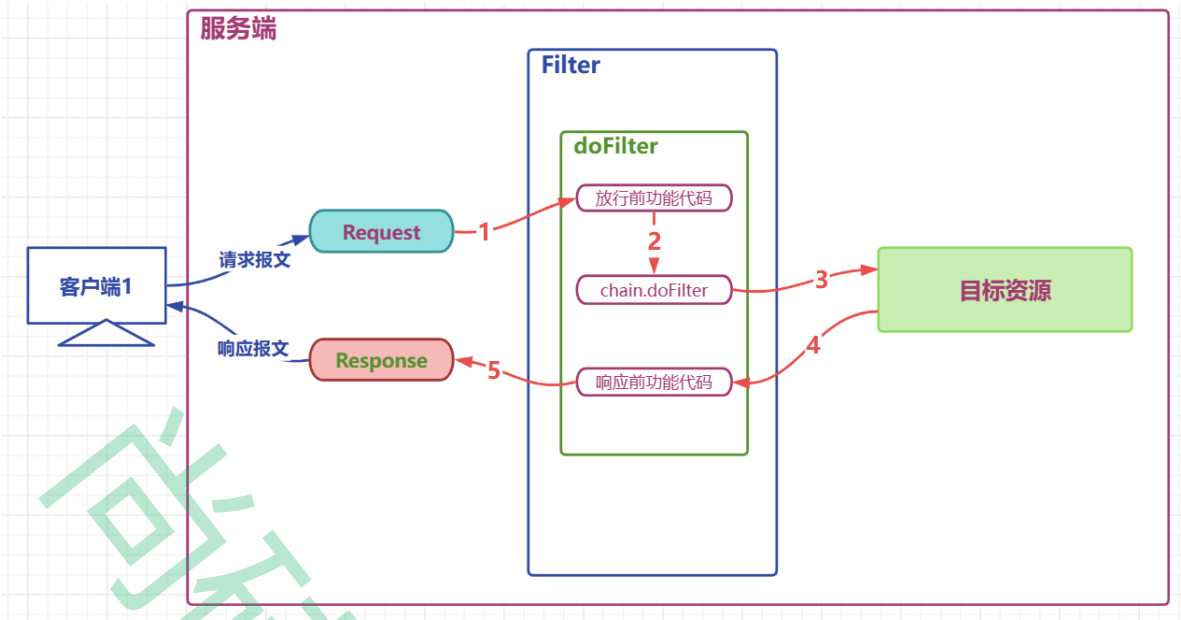
> 配置过滤器以及过滤器的过滤范围：

- web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="https://jakarta.ee/xml/ns/jakartaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="https://jakarta.ee/xml/ns/jakartaee
https://jakarta.ee/xml/ns/jakartaee/web-app_5_0.xsd"
         version="5.0">
    <!--配置filter,并为filter起别名-->
   <filter>
       <filter-name>loggingFilter</filter-name>
       <filter-class>com.atguigu.filters.LoggingFilter</filter-class>
   </filter>
    <!--为别名对应的filter配置要过滤的目标资源-->
    <filter-mapping>
        <filter-name>loggingFilter</filter-name>
        <!--通过映射路径确定过滤资源-->
        <url-pattern>/servletA</url-pattern>
        <!--通过后缀名确定过滤资源-->
        <url-pattern>*.html</url-pattern>
        <!--通过servlet别名确定过滤资源-->
        <servlet-name>servletBName</servlet-name>
    </filter-mapping>
</web-app>
```

- 说明
  - filter-mapping标签中定义了过滤器对那些资源进行过滤；
  - 子标签url-pattern通过映射路径确定过滤范围；
    - /servletA 精确匹配，表示对servletA资源的请求进行过滤；
    - *.html 表示对以.action结尾的路径进行过滤；
    - /* 表示对所有资源进行过滤；
    - 一个filter-mapping下可以配置多个url-pattern；
  - 子标签servlet-name通过servlet别名确定对那些servlet进行过滤；
    - 使用该标签确定目标资源的前提是servlet已经起了别名；
    - 一个filter-mapping下可以定义多个servlet-name；
    - 一个filter-mapping下，servlet-name和url-pattern子标签可以同时存在；

> 过滤过程图解：



2.3 过滤器生命周期

过滤器作为web项目的组件之一，和Servlet的生命周期类似，略有不同，没有servlet的load-on-startup的配置，默认就是系统启动立刻构造。

阶段	对应方法	执行时机	执行次数
创建对象	构造器	web应用启动时	1
初始化方法	void init(FilterConfig filterConfig)	构造完毕	1
过滤请求	void doFilter(ServletRequest servletRequest, ServletResponse servletResponse, FilterChain filterChain)	每次请求	多次
销毁	default void destroy()	web应用关闭时	1次

测试代码：

```
@WebServlet("/*")
public class LifeCycleFilter implements Filter {
    public LifeCycleFilter(){
        System.out.println("LifeCycleFilter constructor method invoked");
    }
    @Override
    public void init(FilterConfig filterConfig) throws ServletException {
        System.out.println("LifeCycleFilter init method invoked");
    }
    @Override
    public void doFilter(ServletRequest servletRequest, ServletResponse
servletResponse, FilterChain filterChain) throws IOException, ServletException {
```

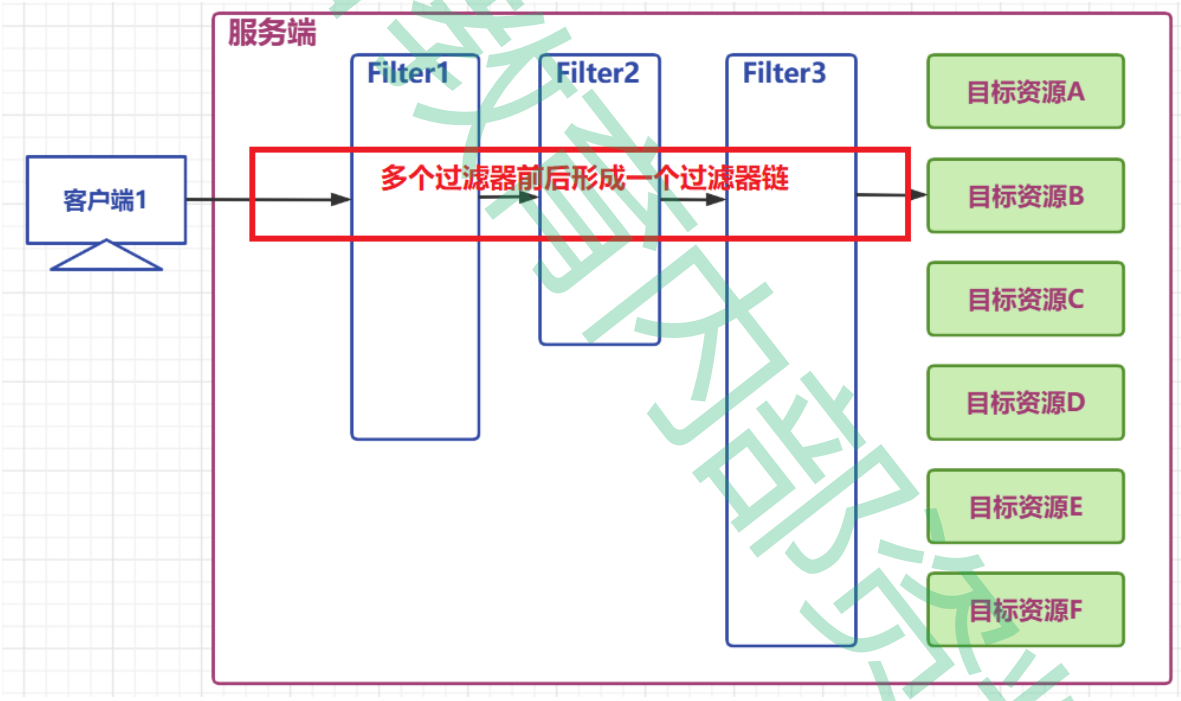
```
        System.out.println("LifeCycleFilter doFilter method invoked");
        filterChain.doFilter(servletRequest,servletResponse);
    }
    @Override
    public void destroy() {
        System.out.println("LifeCycleFilter destory method invoked");
    }
}
```

2.4 过滤器链的使用

> 一个web项目中可以同时定义多个过滤器，多个过滤器对同一个资源进行过滤时，工作位置有先后，整体形成一个工作链，称为过滤器链。

- 过滤器链中的过滤器的顺序由filter-mapping顺序决定；
- 每个过滤器过滤的范围不同，针对同一个资源来说，过滤器链中的过滤器个数可能是不同的；
- 如果某个Filter是使用ServletName进行匹配规则的配置，那么这个Filter执行的优先级要更低；

> 图解过滤器链：



> 过滤器链功能测试：

- 定义三个过滤器，对目标资源Servlet的请求进行过滤；
- 目标Servlet资源代码；

```
@WebServlet("/servletC")
public class ServletC extends HttpServlet {
    @Override
    protected void service(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        System.out.println("servletC service method  invoked");
    }
}
```

- 三个过滤器代码：

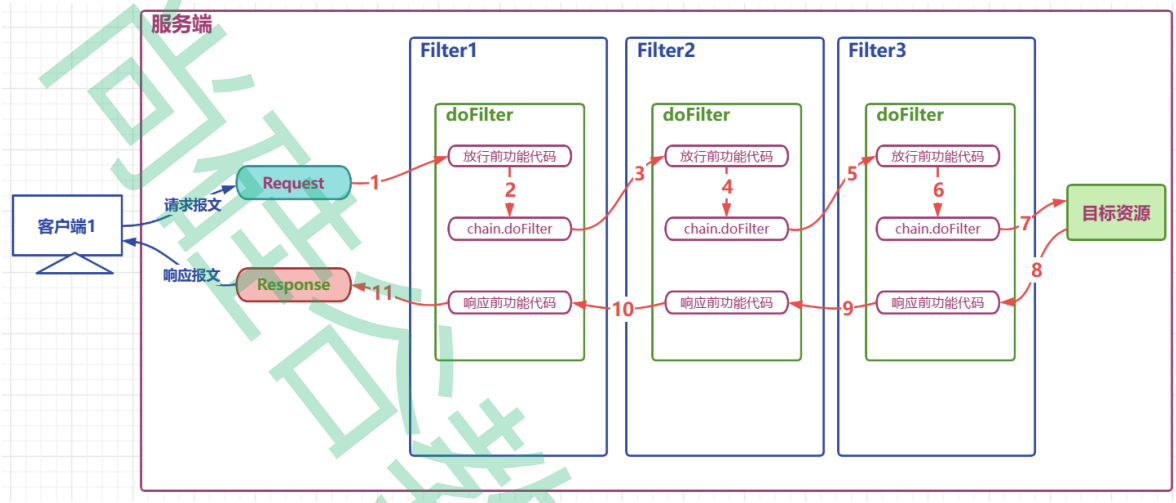
```
public class Filter1  implements Filter {
    @Override
    public void doFilter(ServletRequest servletRequest, ServletResponse
servletResponse, FilterChain filterChain) throws IOException, ServletException {
        System.out.println("filter1 before chain.doFilter code invoked");
        filterChain.doFilter(servletRequest,servletResponse);
        System.out.println("filter1 after  chain.doFilter code invoked");
    }
}

public class Filter2 implements Filter {
    @Override
    public void doFilter(ServletRequest servletRequest, ServletResponse
servletResponse, FilterChain filterChain) throws IOException, ServletException {
        System.out.println("filter2 before chain.doFilter code invoked");
        filterChain.doFilter(servletRequest,servletResponse);
        System.out.println("filter2 after  chain.doFilter code invoked");
    }
}

public class Filter3 implements Filter {
    @Override
    public void doFilter(ServletRequest servletRequest, ServletResponse
servletResponse, FilterChain filterChain) throws IOException, ServletException {
        System.out.println("filter3 before chain.doFilter code invoked");
        filterChain.doFilter(servletRequest,servletResponse);
        System.out.println("filter3 after  chain.doFilter code invoked");
    }
}
```

- 过滤器配置代码：

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="https://jakarta.ee/xml/ns/jakartaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="https://jakarta.ee/xml/ns/jakartaee
https://jakarta.ee/xml/ns/jakartaee/web-app_5_0.xsd"
         version="5.0">
    <filter>
        <filter-name>filter1</filter-name>
        <filter-class>com.atguigu.filters.Filter1</filter-class>
    </filter>
    <filter>
        <filter-name>filter2</filter-name>
        <filter-class>com.atguigu.filters.Filter2</filter-class>
    </filter>
    <filter>
        <filter-name>filter3</filter-name>
        <filter-class>com.atguigu.filters.Filter3</filter-class>
    </filter>
    <!--filter-mapping的顺序决定了过滤器的工作顺序-->
    <filter-mapping>
        <filter-name>filter1</filter-name>
        <url-pattern>/servletC</url-pattern>
    </filter-mapping>
    <filter-mapping>
```

```
        <filter-name>filter2</filter-name>
        <url-pattern>/servletC</url-pattern>
    </filter-mapping>
    <filter-mapping>
        <filter-name>filter3</filter-name>
        <url-pattern>/servletC</url-pattern>
    </filter-mapping>
</web-app>
```

> 工作流程图解：



2.5 注解方式配置过滤器

> @WebFilter注解的使用：

注解源码通过idea查看

- 一个比较完整的Filter的XML配置。

```
<!--配置filter,并为filter起别名-->
<filter>
    <filter-name>loggingFilter</filter-name>
    <filter-class>com.atguigu.filters.LoggingFilter</filter-class>
    <!--配置filter的初始参数-->
    <init-param>
        <param-name>dateTimePattern</param-name>
        <param-value>yyyy-MM-dd HH:mm:ss</param-value>
    </init-param>
</filter>
<!--为别名对应的filter配置要过滤的目标资源-->
<filter-mapping>
    <filter-name>loggingFilter</filter-name>
    <!--通过映射路径确定过滤资源-->
    <url-pattern>/servletA</url-pattern>
    <!--通过后缀名确定过滤资源-->
    <url-pattern>*.html</url-pattern>
    <!--通过servlet别名确定过滤资源-->
    <servlet-name>servletBName</servlet-name>
</filter-mapping>
```

- 将xml配置转换成注解方式实现。

```
@WebFilter(
        filterName = "loggingFilter",
        initParams = {@WebInitParam(name="dateTimePattern",value="yyyy-MM-dd
HH:mm:ss")},
        urlPatterns = {"/servletA","*.html"},
        servletNames = {"servletBName"}
)
public class LoggingFilter  implements Filter {
    /* 内部代码 略 */
}
```

三 监听器

3.1 监听器概述

> 监听器：专门用于对域对象对象身上发生的事件或状态改变进行监听和相应处理的对象；

- 监听器是GOF设计模式中，观察者模式的典型案例；
- 监听器使用的感受类似JS中的事件，被观察的对象发生某些情况时，自动触发代码的执行；
- 监听器并不监听web项目中的所有组件，仅仅是对三大域对象做相关的事件监听；

> 监听器的分类：

- web中定义八个监听器接口作为监听器的规范，这八个接口按照不同的标准可以形成不同的分类；
- 按监听的对象划分：
  - application域监听器 ServletContextListener ServletContextAttributeListener；
  - session域监听器 HttpSessionListener HttpSessionAttributeListener HttpSessionBindingListener HttpSessionActivationListener ；
  - request域监听器 ServletRequestListener ServletRequestAttributeListener；
- 按监听的事件分：
  - 域对象的创建和销毁监听器 ServletContextListener HttpSessionListener ServletRequestListener ；
  - 域对象数据增删改事件监听器 ServletContextAttributeListener HttpSessionAttributeListener ServletRequestAttributeListener；
  - 其他监听器 HttpSessionBindingListener HttpSessionActivationListener ；

3.2 监听器的六个主要接口

3.2.1 application域监听器

> ServletContextListener 监听ServletContext对象的创建与销毁。

方法名	作用
contextInitialized(ServletContextEvent sce)	ServletContext创建时调用

方法名	作用
contextDestroyed(ServletContextEvent sce)	ServletContext销毁时调用

- ServletContextEvent对象代表从ServletContext对象身上捕获到的事件，通过这个事件对象我们可以获取到ServletContext对象。

> ServletContextAttributeListener 监听ServletContext中属性的添加、移除和修改。

方法名	作用
attributeAdded(ServletContextAttributeEvent scab)	向ServletContext中添加属性时调用
attributeRemoved(ServletContextAttributeEvent scab)	从ServletContext中移除属性时调用
attributeReplaced(ServletContextAttributeEvent scab)	当ServletContext中的属性被修改时调用

- ServletContextAttributeEvent对象代表属性变化事件，它包含的方法如下：

方法名	作用
getName()	获取修改或添加的属性名
getValue()	获取被修改或添加的属性值
getServletContext()	获取ServletContext对象

> 测试代码

- 定义监听器：

```
@WebListener
public class ApplicationListener implements ServletContextListener ,
ServletContextAttributeListener {
    // 监听初始化
    @Override
    public void contextInitialized(ServletContextEvent sce) {
        ServletContext application = sce.getServletContext();
        System.out.println("application"+application.hashCode()+" initialized");
    }
    // 监听销毁
    @Override
    public void contextDestroyed(ServletContextEvent sce) {
        ServletContext application = sce.getServletContext();
        System.out.println("application"+application.hashCode()+" destroyed");
    }

    // 监听数据增加
    @Override
    public void attributeAdded(ServletContextAttributeEvent scae) {
        String name = scae.getName();
        Object value = scae.getValue();
        ServletContext application = scae.getServletContext();
        System.out.println("application"+application.hashCode()+"
add:"+name+"="+value);
    }
```

```
    // 监听数据移除
    @Override
    public void attributeRemoved(ServletContextAttributeEvent scae) {
        String name = scae.getName();
        Object value = scae.getValue();
        ServletContext application = scae.getServletContext();
        System.out.println("application"+application.hashCode()+"
remove:"+name+"="+value);
    }
    // 监听数据修改
    @Override
    public void attributeReplaced(ServletContextAttributeEvent scae) {
        String name = scae.getName();
        Object value = scae.getValue();
        ServletContext application = scae.getServletContext();
        Object newValue = application.getAttribute(name);
        System.out.println("application"+application.hashCode()+"
change:"+name+"="+value+" to "+newValue);
    }
}
```

- 定义触发监听器的代码：

```
// ServletA用于向application域中放入数据
@WebServlet(urlPatterns = "/servletA",name = "servletAName")
public class ServletA extends HttpServlet {
    @Override
    protected void service(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        // 向application域中放入数据
        ServletContext application = this.getServletContext();
        application.setAttribute("k1","v1");
        application.setAttribute("k2","v2");
    }
}
// ServletB用于向application域中修改和移除数据
@WebServlet(urlPatterns = "/servletB", name = "servletBName")
public class ServletB extends HttpServlet {
    @Override
    protected void service(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        ServletContext appliation  = getServletContext();
        //  修改application域中的数据
        appliation.setAttribute("k1","value1");
        //  删除application域中的数据
        appliation.removeAttribute("k2");
    }
}
```

3.2.2 session域监听器

> HttpSessionListener 监听HttpSession对象的创建与销毁：

方法名	作用
sessionCreated(HttpSessionEvent hse)	HttpSession对象创建时调用

方法名	作用
sessionDestroyed(HttpSessionEvent hse)	Session对象销毁时调用

- HttpSessionEvent对象代表从HttpSession对象身上捕获到的事件，通过这个事件对象我们可以获取到触发事件的HttpSession对象。

> HttpSessionAttributeListener 监听HttpSession中属性的添加、移除和修改：

方法名	作用
attributeAdded(HttpSessionBindingEvent se)	向HttpSession中添加属性时调用
attributeRemoved(HttpSessionBindingEvent se)	从HttpSession中移除属性时调用
attributeReplaced(HttpSessionBindingEvent se)	当HttpSession中的属性被修改时调用

- HttpSessionBindingEvent对象代表属性变化事件，它包含的方法如下：

方法名	作用
getName()	获取修改或添加的属性名
getValue()	获取被修改或添加的属性值
getSession()	获取触发事件的HttpSession对象

> 测试代码：

- 定义监听器：

```
@WebListener
public class SessionListener implements HttpSessionListener,
HttpSessionAttributeListener {
    // 监听session创建
    @Override
    public void sessionCreated(HttpSessionEvent se) {
        HttpSession session = se.getSession();
        System.out.println("session"+session.hashCode()+" created");
    }
    // 监听session销毁
    @Override
    public void sessionDestroyed(HttpSessionEvent se) {
        HttpSession session = se.getSession();
        System.out.println("session"+session.hashCode()+" destroyed");
    }
    // 监听数据增加
    @Override
    public void attributeAdded(HttpSessionBindingEvent se) {
        String name = se.getName();
        Object value = se.getValue();
        HttpSession session = se.getSession();
        System.out.println("session"+session.hashCode()+" add:"+name+"="+value);
    }
    // 监听数据移除
    @Override
    public void attributeRemoved(HttpSessionBindingEvent se) {
```

```
        String name = se.getName();
        Object value = se.getValue();
        HttpSession session = se.getSession();
        System.out.println("session"+session.hashCode()+" remove:"+name+"="+value);
    }
    // 监听数据修改
    @Override
    public void attributeReplaced(HttpSessionBindingEvent se) {
        String name = se.getName();
        Object value = se.getValue();
        HttpSession session = se.getSession();
        Object newValue = session.getAttribute(name);
        System.out.println("session"+session.hashCode()+" change:"+name+"="+value+" to
"+newValue);
    }
}
```

- 定义触发监听器的代码：

```
// servletA用于创建session并向session中放数据
@WebServlet(urlPatterns = "/servletA",name = "servletAName")
public class ServletA extends HttpServlet {
    @Override
    protected void service(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        // 创建session,并向session中放入数据
        HttpSession session = req.getSession();

        session.setAttribute("k1","v1");
        session.setAttribute("k2","v2");
    }
}
// servletB用于修改删除session中的数据并手动让session不可用
@WebServlet(urlPatterns = "/servletB", name = "servletBName")
public class ServletB extends HttpServlet {
    @Override
    protected void service(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        HttpSession session = req.getSession();
        //  修改session域中的数据
        session.setAttribute("k1","value1");
        //  删除session域中的数据
        session.removeAttribute("k2");
        // 手动让session不可用
        session.invalidate();
    }
}
```

3.2.3 request域监听器

> ServletRequestListener 监听ServletRequest对象的创建与销毁：

方法名	作用
requestInitialized(ServletRequestEvent sre)	ServletRequest对象创建时调用
requestDestroyed(ServletRequestEvent sre)	ServletRequest对象销毁时调用

- ServletRequestEvent对象代表从HttpServletRequest对象身上捕获到的事件，通过这个事件对象我们可以获取到触发事件的HttpServletRequest对象。另外还有一个方法可以获取到当前Web应用的ServletContext对象。

> ServletRequestAttributeListener 监听ServletRequest中属性的添加、移除和修改：

方法名	作用
attributeAdded(ServletRequestAttributeEvent srae)	向ServletRequest中添加属性时调用
attributeRemoved(ServletRequestAttributeEvent srae)	从ServletRequest中移除属性时调用
attributeReplaced(ServletRequestAttributeEvent srae)	当ServletRequest中的属性被修改时调用

- ServletRequestAttributeEvent对象代表属性变化事件，它包含的方法如下：

方法名	作用
getName()	获取修改或添加的属性名
getValue()	获取被修改或添加的属性值
getServletRequest ()	获取触发事件的ServletRequest对象

- 定义监听器：

```
@WebListener
public class RequestListener implements ServletRequestListener ,
ServletRequestAttributeListener {
    // 监听初始化
    @Override
    public void requestInitialized(ServletRequestEvent sre) {
        ServletRequest request = sre.getServletRequest();
        System.out.println("request"+request.hashCode()+" initialized");
    }
    // 监听销毁
    @Override
    public void requestDestroyed(ServletRequestEvent sre) {
        ServletRequest request = sre.getServletRequest();
        System.out.println("request"+request.hashCode()+" destoryed");
    }
    // 监听数据增加
    @Override
    public void attributeAdded(ServletRequestAttributeEvent srae) {
        String name = srae.getName();
        Object value = srae.getValue();
        ServletRequest request = srae.getServletRequest();
        System.out.println("request"+request.hashCode()+" add:"+name+"="+value);
    }
    //  监听数据移除
    @Override
    public void attributeRemoved(ServletRequestAttributeEvent srae) {
        String name = srae.getName();
        Object value = srae.getValue();
        ServletRequest request = srae.getServletRequest();
```

```
        System.out.println("request"+request.hashCode()+" remove:"+name+"="+value);
    }
    // 监听数据修改
    @Override
    public void attributeReplaced(ServletRequestAttributeEvent srae) {
        String name = srae.getName();
        Object value = srae.getValue();
        ServletRequest request = srae.getServletRequest();
        Object newValue = request.getAttribute(name);
        System.out.println("request"+request.hashCode()+" change:"+name+"="+value+" to
"+newValue);
    }
}
```

- 定义触发监听器的代码：

```
//  servletA向请求域中放数据
@WebServlet(urlPatterns = "/servletA",name = "servletAName")
public class ServletA extends HttpServlet {
    @Override
    protected void service(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        // 向request中增加数据
        req.setAttribute("k1","v1");
        req.setAttribute("k2","v2");
        // 请求转发
        req.getRequestDispatcher("servletB").forward(req,resp);
    }
}
// servletB修改删除域中的数据
@WebServlet(urlPatterns = "/servletB", name = "servletBName")
public class ServletB extends HttpServlet {
    @Override
    protected void service(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        //  修改request域中的数据
        req.setAttribute("k1","value1");
        //  删除session域中的数据
        req.removeAttribute("k2");
    }
}
```

3.3 session域的两个特殊监听器

3.3.3 session绑定监听器

> HttpSessionBindingListener 监听当前监听器对象在Session域中的增加与移除：

方法名	作用
valueBound(HttpSessionBindingEvent event)	该类的实例被放到Session域中时调用
valueUnbound(HttpSessionBindingEvent event)	该类的实例从Session中移除时调用

- HttpSessionBindingEvent对象代表属性变化事件，它包含的方法如下：

方法名	作用
getName()	获取当前事件涉及的属性名
getValue()	获取当前事件涉及的属性值
getSession()	获取触发事件的HttpSession对象

> 测试代码：

- 定义监听器：

```
public class MySessionBindingListener  implements HttpSessionBindingListener {
    // 监听绑定
    @Override
    public void valueBound(HttpSessionBindingEvent event) {
        HttpSession session = event.getSession();
        String name = event.getName();
        System.out.println("MySessionBindingListener"+this.hashCode()+" binding into
session"+session.hashCode()+" with name "+name);
    }
    // 监听解除绑定
    @Override
    public void valueUnbound(HttpSessionBindingEvent event) {
        HttpSession session = event.getSession();
        String name = event.getName();
        System.out.println("MySessionBindingListener"+this.hashCode()+" unbond outof
session"+session.hashCode()+" with name "+name);
    }
}
```

- 定义触发监听器的代码：

```
@WebServlet(urlPatterns = "/servletA",name = "servletAName")
public class ServletA extends HttpServlet {
    @Override
    protected void service(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        HttpSession session = req.getSession();
        // 绑定监听器
        session.setAttribute("bindingListener",new MySessionBindingListener());
        // 解除绑定监听器
        session.removeAttribute("bindingListener");
    }
}
```

3.3.4 钝化活化监听器

> HttpSessionActivationListener 监听某个对象在Session中的序列化与反序列化：

方法名	作用
sessionWillPassivate(HttpSessionEvent se)	该类实例和Session一起钝化到硬盘时调用

方法名	作用
sessionDidActivate(HttpSessionEvent se)	该类实例和Session一起活化到内存时调用

- HttpSessionEvent对象代表事件对象，通过getSession()方法获取事件涉及的HttpSession对象。

> 什么是钝化活化：

- session对象在服务端是以对象的形式存储于内存的，session过多，服务器的内存也是吃不消的；
- 而且一旦服务器发生重启，所有的session对象都将被清除，也就意味着session中存储的不同客户端的登录状态丢失；
- 为了分摊内存 压力并且为了保证session重启不丢失，我们可以设置将session进行钝化处理；
- 在关闭服务器前或者到达了设定时间时，对session进行序列化到磁盘，这种情况叫做session的钝化；
- 在服务器启动后或者再次获取某个session时，将磁盘上的session进行反序列化到内存，这种情况叫做session的活化；

> 如何配置钝化活化？

- 在web目录下，添加 META-INF 下创建Context.xml。

- 文件中配置钝化。

```
<?xml version="1.0" encoding="UTF-8"?>
<Context>
    <Manager className="org.apache.catalina.session.PersistentManager"
maxIdleSwap="1">
        <Store className="org.apache.catalina.session.FileStore"
directory="d:\mysession"></Store>
    </Manager>
</Context>
```

- 请求servletA，获得session，并存入数据，然后重启服务器：

```
@WebServlet(urlPatterns = "/servletA",name = "servletAName")
public class ServletA extends HttpServlet {
    @Override
    protected void service(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        HttpSession session = req.getSession();
        // 添加数据
        session.setAttribute("k1","v1");
    }
}
```

- 请求servletB获取session，获取重启前存入的数据：

```
@WebServlet(urlPatterns = "/servletB", name = "servletBName")
public class ServletB extends HttpServlet {
    @Override
    protected void service(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        HttpSession session = req.getSession();
        Object v1 = session.getAttribute("k1");
        System.out.println(v1);
    }
}
```

如何监听钝化活化?

- 定义监听器:

```
package com.atguigu.listeners;
import jakarta.servlet.http.HttpSession;
import jakarta.servlet.http.HttpSessionActivationListener;
import jakarta.servlet.http.HttpSessionEvent;
import java.io.Serializable;
public class ActivationListener  implements HttpSessionActivationListener,
Serializable {
    //  监听钝化
    @Override
    public void sessionWillPassivate(HttpSessionEvent se) {
        HttpSession session = se.getSession();
        System.out.println("session with JSESSIONID "+ session.getId()+" will
passivate");
    }
    //  监听活化
    @Override
    public void sessionDidActivate(HttpSessionEvent se) {
        HttpSession session = se.getSession();
        System.out.println("session with JSESSIONID "+ session.getId()+" did
activate");
    }
}
```

- 定义触发监听器的代码:

```
@WebServlet(urlPatterns = "/servletA",name = "servletAName")
public class ServletA extends HttpServlet {
    @Override
    protected void service(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        HttpSession session = req.getSession();
        // 添加数据
        session.setAttribute("k1","v1");
        // 添加钝化活化监听器
        session.setAttribute("activationListener",new ActivationListener());
    }
}
```

四 案例开发-日程管理-第三期

4.1 过滤器控制登录校验

> 需求说明：未登录状态下不允许访问showShedule.html和SysScheduleController相关增删改处理，重定向到login.html，登录成功后可以自由访问。

- 开发登录过滤器，对指定资源的请求进行过滤。

```
package com.atguigu.schedule.filters;
import jakarta.servlet.*;
import jakarta.servlet.annotation.WebFilter;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import jakarta.servlet.http.HttpSession;
import java.io.IOException;
@WebFilter(urlPatterns = {"/showSchedule.html","/schedule/*"})
public class LoginFilter  implements Filter {
    @Override
    public void doFilter(ServletRequest servletRequest, ServletResponse
servletResponse, FilterChain filterChain) throws IOException, ServletException {
        HttpServletRequest request =(HttpServletRequest) servletRequest;
        HttpServletResponse response =(HttpServletResponse) servletResponse;
        HttpSession session = request.getSession();
        Object sysUser = session.getAttribute("sysUser");
        if(null != sysUser){
            // session中如果存在登录的用户 代表用户登录过,则放行
            filterChain.doFilter(servletRequest,servletResponse);

        }else{
            //用户未登录,重定向到登录页
            response.sendRedirect("/login.html");
        }
    }
}
```

- 修改用户登录请求的的login方法，登录成功时，将用户信息存入session。

```
    /**
     * 用户登录的业务接口
     * @param req
     * @param resp
     * @throws ServletException
     * @throws IOException
     */
    protected void login(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        // 接收用户请求参数
        // 获取要注册的用户名密码
        String username = req.getParameter("username");
        String userPwd = req.getParameter("userPwd");
        // 调用服务层方法,根据用户名查询数据库中是否有一个用户
        SysUser loginUser =userService.findByUsername(username);
        if(null == loginUser){
            // 没有根据用户名找到用户,说明用户名有误
            resp.sendRedirect("/loginUsernameError.html");
        }else if(! loginUser.getUserPwd().equals(MD5Util.encrypt(userPwd))){
            // 用户密码有误,
```

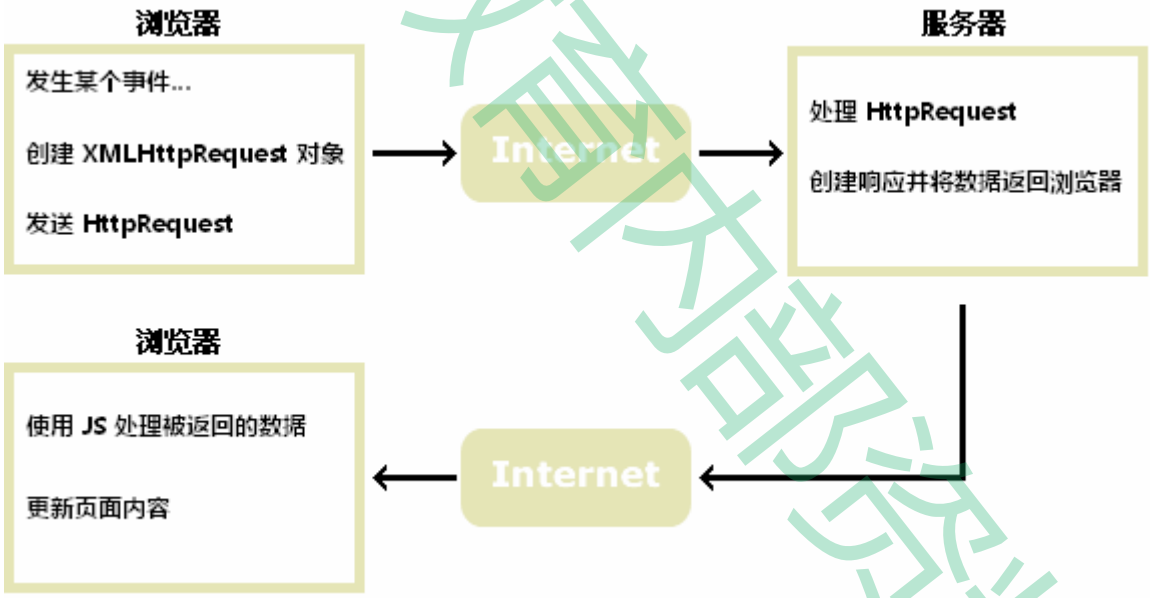
```
            resp.sendRedirect("/loginUserPwdError.html");
        }else{
            // 登录成功,将用户信息存入session
            req.getSession().setAttribute("sysUser",loginUser);
            // 登录成功,重定向到日程展示页
            resp.sendRedirect("/showSchedule.html");
        }
    }
```

五 AJAX

5.1 什么是AJAX

- AJAX = Asynchronous JavaScript and XML（异步的 JavaScript 和 XML）；
- AJAX 不是新的编程语言，而是一种使用现有标准的新方法；
- AJAX 最大的优点是在不重新加载整个页面的情况下，可以与服务器交换数据并更新部分网页内容；
- AJAX 不需要任何浏览器插件，但需要用户允许 JavaScript 在浏览器上执行；
- XMLHttpRequest 只是实现 Ajax 的一种方式；

> AJAX 工作原理：



- 简单来说，我们之前发的请求通过类似 form表单标签、a标签这种方式。现在通过运行js代码动态决定什么时候发送什么样的请求；
- 通过运行JS代码发送的请求浏览器可以不用跳转页面，我们可以在JS代码中决定是否要跳转页面；
- 通过运行JS代码发送的请求，接收到返回结果后，我们可以将结果通过dom编程渲染到页面的某些元素上，实现局部更新；

5.2 如何实现AJAX 请求

> 原生javascript方式进行ajax(了解):

```
<script>
  function loadXMLDoc(){
    var xmlhttp=new XMLHttpRequest();
      // 设置回调函数处理响应结果
    xmlhttp.onreadystatechange=function(){
      if (xmlhttp.readyState==4 && xmlhttp.status==200)
      {
        document.getElementById("myDiv").innerHTML=xmlhttp.responseText;
      }
    }
      // 设置请求方式和请求的资源路径
    xmlhttp.open("GET","/try/ajax/ajax_info.txt",true);
      // 发送请求
    xmlhttp.send();
  }
</script>
```

六 案例开发-日程管理-第四期

6.1 注册提交前校验用户名是否占用功能

> 客户端代码编写处理：

- regist.html页面代码

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
    <style>

        .ht{
            text-align: center;
            color: cadetblue;
            font-family: 幼圆;
        }
        .tab{
            width: 500px;
            border: 5px solid cadetblue;
            margin: 0px auto;
            border-radius: 5px;
            font-family: 幼圆;
        }
        .ltr td{
            border: 1px solid  powderblue;
        }
        .ipt{
            border: 0px;
            width: 50%;
        }
        .btn1{
            border: 2px solid powderblue;
```

```
            border-radius: 4px;
            width:60px;
            background-color: antiquewhite;
        }

        .msg {
            color: gold;
        }
        .buttonContainer{
            text-align: center;
        }
    </style>

    <script>
        // 校验用户名的方法
        function checkUsername(){
            // 定义正则
            var usernameReg=/^[a-zA-Z0-9]{5,10}$/
            var username =document.getElementById("usernameInput").value
            var usernameMsgSpan =document.getElementById("usernameMsg")
            if(!usernameReg.test(username)){
                usernameMsgSpan.innerText="不合法"
                return false
            }
            // 发送ajax请求校验用户名是否被占用
            var request;
            if(window.XMLHttpRequest){
                request= new XMLHttpRequest();
            }else{
                request= new ActiveXObject("Microsoft.XMLHTTP");
            }
            request.onreadystatechange= function (){
                // request.readyState == 4 代表请求结束,已经接收到响应结果
                // request.status== 200  表示后端响应状态码是200
                if(request.readyState == 4  && request.status== 200){
                    // 后端的响应的JSON字符串转换为前端的对象
                    var response =JSON.parse(request.responseText)
                    console.log(response)
                    //  判断业务码是否是200
                    if (response.code != 200){
                        usernameMsgSpan.innerText="已占用"
                        return false
                    }
                }
            }
            // 设置请求方式,请求资源路径,是否为异步请求
            request.open("GET",'/user/checkUsernameUsed?username='+username,true)
            // 发送请求
            request.send();
            // 前面校验都通过
            // usernameMsgSpan.innerText="OK"
            // return true
        }
        // 校验密码的方法
        function checkUserPwd(){
            // 定义正则
            var passwordReg=/^[0-9]{6}$/
            var userPwd =document.getElementById("userPwdInput").value
```

```
            var userPwdMsgSpan =document.getElementById("userPwdMsg")
            if(!passwordReg.test(userPwd)){
                userPwdMsgSpan.innerText="不合法"
                return false
            }
            userPwdMsgSpan.innerText="OK"
            return true
        }
        // 校验密码的方法
        function checkReUserPwd(){
            // 定义正则
            var passwordReg=/^[0-9]{6}$/
            var userPwd =document.getElementById("userPwdInput").value
            var reUserPwd =document.getElementById("reUserPwdInput").value
            var reUserPwdMsgSpan =document.getElementById("reUserPwdMsg")
            if(!passwordReg.test(userPwd)){
                reUserPwdMsgSpan.innerText="不合法"
                return false
            }
            if(userPwd != reUserPwd){
                reUserPwdMsgSpan.innerText="不一致"
                return false

            }
            reUserPwdMsgSpan.innerText="OK"
            return true
        }
        //表单提交时统一校验
        function checkForm(){
            return checkUsername() && checkUserPwd() && checkReUserPwd()
        }
    </script>
</head>
<body>
<h1 class="ht">欢迎使用日程管理系统</h1>
<h3 class="ht">请注册</h3>
<form method="post" action="/user/regist" onsubmit="return checkForm()">
    <table class="tab" cellspacing="0px">
        <tr class="ltr">
            <td>请输入账号</td>
            <td>
                <input class="ipt" id="usernameInput" type="text" name="username"
onblur="checkUsername()">
                <span id="usernameMsg" class="msg"></span>
            </td>
        </tr>
        <tr class="ltr">
            <td>请输入密码</td>
            <td>
                <input class="ipt" id="userPwdInput" type="password" name="userPwd"
onblur="checkUserPwd()">
                <span id="userPwdMsg" class="msg"></span>
            </td>
        </tr>
        <tr class="ltr">
            <td>确认密码</td>
            <td>
```

```
                <input class="ipt" id="reUserPwdInput" type="password"
onblur="checkReUserPwd()">
                <span id="reUserPwdMsg" class="msg"></span>
            </td>
        </tr>
        <tr class="ltr">
            <td colspan="2" class="buttonContainer">
                <input class="btn1" type="submit" value="注册">
                <input class="btn1" type="reset" value="重置">
                <button class="btn1"><a  href="/login.html">去登录</a></button>
            </td>
        </tr>
    </table>
</form>
</body>
</html>
```

> 服务端代码处理：

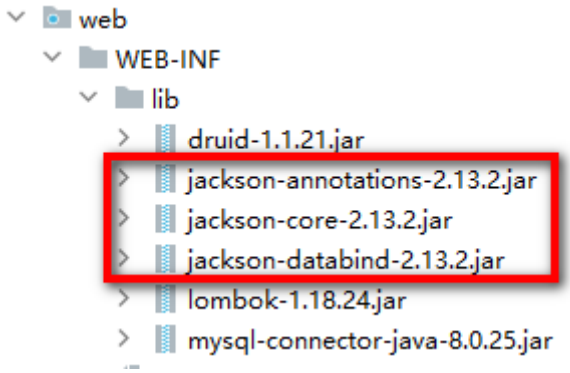
- 添加公共的JSON数据响应格式类：

```
/**
 * 业务含义和状态码对应关系的枚举
 */
public enum ResultCodeEnum {
    SUCCESS(200,"success"),
    USERNAME_ERROR(501,"usernameError"),
    PASSWORD_ERROR(503,"passwordError"),
    NOTLOGIN(504,"notLogin"),
    USERNAME_USED(505,"userNameUsed")
    ;
    private Integer code;
    private String message;
    private ResultCodeEnum(Integer code, String message) {
        this.code = code;
        this.message = message;
    }
    public Integer getCode() {
        return code;
    }
    public String getMessage() {
        return message;
    }
}
```

```
/**
 * 全局统一响应的JSON格式处理类
 */
public class Result<T> {
    // 返回码
    private Integer code;
    // 返回消息
    private String message;
    // 返回数据
    private T data;
    public Result(){}
    // 返回数据
```

```java
    protected static <T> Result<T> build(T data) {
        Result<T> result = new Result<T>();
        if (data != null)
            result.setData(data);
        return result;
    }
    public static <T> Result<T> build(T body, Integer code, String message) {
        Result<T> result = build(body);
        result.setCode(code);
        result.setMessage(message);
        return result;
    }
    public static <T> Result<T> build(T body, ResultCodeEnum resultCodeEnum) {
        Result<T> result = build(body);
        result.setCode(resultCodeEnum.getCode());
        result.setMessage(resultCodeEnum.getMessage());
        return result;
    }
    public static<T> Result<T> ok(T data){
        Result<T> result = build(data);
        return build(data, ResultCodeEnum.SUCCESS);
    }
    public Result<T> message(String msg){
        this.setMessage(msg);
        return this;
    }
    public Result<T> code(Integer code){
        this.setCode(code);
        return this;
    }
    public Integer getCode() {
        return code;
    }
    public void setCode(Integer code) {
        this.code = code;
    }
    public String getMessage() {
        return message;
    }
    public void setMessage(String message) {
        this.message = message;
    }
    public T getData() {
        return data;
    }
    public void setData(T data) {
        this.data = data;
    }
}
```


- 增加Jackson依赖：



- 添加WEBUtil工具类：

```
public class WebUtil {
    private static ObjectMapper objectMapper;
    // 初始化objectMapper
    static{
        objectMapper=new ObjectMapper();
        // 设置JSON和Object转换时的时间日期格式
        objectMapper.setDateFormat(new SimpleDateFormat("yyyy-MM-dd HH:mm:ss"));
    }
    // 从请求中获取JSON串并转换为Object
    public static <T> T readJson(HttpServletRequest request,Class<T> clazz){
        T t =null;
        BufferedReader reader = null;
        try {
            reader = request.getReader();
            StringBuffer buffer =new StringBuffer();
            String line =null;
            while((line = reader.readLine())!= null){
                buffer.append(line);
            }
            t= objectMapper.readValue(buffer.toString(),clazz);
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
        return t;
    }
    // 将Result对象转换成JSON串并放入响应对象
    public static void writeJson(HttpServletResponse response, Result result){
        response.setContentType("application/json;charset=UTF-8");
        try {
            String json = objectMapper.writeValueAsString(result);
            response.getWriter().write(json);
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
}
```

- 用户名校验业务接口代码：

```
  /**
     * SysUserController下,注册时校验用户名是否被占用的业务接口实现
     * @param req
     * @param resp
     * @throws ServletException
     * @throws IOException
```

```
     */
    protected void checkUsernameUsed(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
        String username = req.getParameter("username");
        SysUser registUser = userService.findByUsername(username);
        //封装结果对象
        Result result=null;
        if(null ==registUser){
            // 未占用,创建一个code为200的对象
            result= Result.ok(null);
        }else{
            // 占用, 创建一个结果为505的对象
            result= Result.build(null, ResultCodeEnum.USERNAME_USED);
        }
        // 将result对象转换成JSON并响应给客户端
        WebUtil.writeJson(resp,result);
    }
```

第七章 前端工程化

一、前端工程化开篇

1.1 什么是前端工程化

> 前端工程化 是使用 软件工程的方法 来 单独 解决 前端 的开发流程中 模块化、组件化、规范化、自动化 的问题，以提高效率和降低成本。

1.2 前端工程化实现技术栈

> 前端工程化实现的技术栈有很多，我们采用ES6+Nodejs+npm+Vite+VUE3+Router+Pinia+Axios+Element-plus组合来实现。

- ECMAScript6　VUE3中大量使用ES6语法；
- Nodejs　前端项目运行环境；
- npm　依赖下载工具；
- Vite　前端项目构建工具；
- VUE3　优秀的渐进式前端框架；
- Router　通过路由实现页面切换；
- Pinia　通过状态管理实现组件数据传递；
- Axios　ajax异步请求封装技术实现前后端数据交互；
- Element-plus　可以提供丰富的快速构建网页的组件仓库；

二、ECMA6Script

2.1. ES6的介绍

> ECMAScript 6，简称ES6，是**JavaScript**语言的一次重大更新。它于**2015**年发布，是原来的ECMAScript标准的第六个版本。ES6带来了大量的新特性，包括箭头函数、模板字符串、let和const关键字、解构、默认参数值、模块系统等等，大大提升了JavaScript的开发体验。 由于VUE3中大量使用了ES6的语法,所以ES6成为了学习VUE3的门槛之一 。ES6对JavaScript的改进在以下几个方面：

1. 更加简洁：ES6引入了一些新的语法，如箭头函数、类和模板字符串等，使代码更加简洁易懂；
2. 更强大的功能：ES6引入了一些新的API、解构语法和迭代器等功能，从而使得JavaScript更加强大；
3. 更好的适用性：ES6引入的模块化功能为JavaScript代码的组织和管理提供了更好的方式，不仅提高了程序的可维护性，还让JavaScript更方便地应用于大型的应用程序；

> 总的来说，ES6在提高JavaScript的核心语言特性和功能方面取得了很大的进展。由于ES6已经成为了JavaScript的标准，它的大多数新特性都已被现在浏览器所支持，因此现在可以放心地使用ES6来开发前端应用程序。

历史版本：

标准版本	发布时间	新特性
ES1	1997年	第一版 ECMAScript
ES2	1998年	引入setter和getter函数，增加了try/catch，switch语句允许字符串
ES3	1999年	引入了正则表达式和更好的字符串处理
ES4	取消	取消，部分特性被ES3.1和ES5继承
ES5	2009年	Object.defineProperty，JSON，严格模式，数组新增方法等
ES5.1	2011年	对ES5做了一些勘误和例行修订
ES6	2015年	箭头函数、模板字符串、解构、let和const关键字、类、模块系统等
ES2016	2016年	数组.includes，指数操作符（**），Array.prototype.fill等
ES2017	2017年	异步函数async/await，Object.values/Object.entries，字符串填充
ES2018	2018年	正则表达式命名捕获组，几个有用的对象方法，异步迭代器等
ES2019	2019年	Array.prototype.{flat,flatMap}，Object.fromEntries等
ES2020	2020年	BigInt、动态导入、可选链操作符、空位合并操作符
ES2021	2021年	String.prototype.replaceAll，逻辑赋值运算符，Promise.any等
……		

2.2 es6的变量和模板字符串

> ES6 新增了 `let` 和 `const` ，用来声明变量，使用的细节上也存在诸多差异。

- let 和var的差别：
  1、let 不能重复声明；
  2、let有块级作用域，非函数的花括号遇见let会有块级作用域，也就是只能在花括号里面访问；
  3、let不会预解析进行变量提升；
  4、let 定义的全局变量不会作为window的属性；
  5、let在es6中推荐优先使用；

```
<script>
    //1. let只有在当前代码块有效代码块. 代码块、函数、全局
    {
      let a = 1
      var b = 2
    }
    console.log(a);  // a is not defined   花括号外面无法访问
    console.log(b);  // 可以正常输出
```

```
    //2. 不能重复声明
    let name = '天真'
    let name = '无邪'
    //3. 不存在变量提升（先声明，在使用）
    console.log(test) //可以      但是值为undefined
    var test = 'test'
    console.log(test1) //不可以  let命令改变了语法行为，它所声明的变量一定要在声明后使用，否则报错。
    let test1 = 'test1'
    //4、不会成为window的属性
    var a = 100
    console.log(window.a) //100
    let b = 200
    console.log(window.b) //undefined
    //5. 循环中推荐使用
    for (let i = 0; i < 10; i++) {
      // ...
    }
    console.log(i);
</script>
```

- const和var的差异：

  1、新增const和let类似，只是const定义的变量不能修改；

  2、并不是变量的值不得改动，而是变量指向的那个内存地址所保存的数据不得改动；

```
<script>
    //1声明常量
    const PI = 3.1415926;
    PI=3.14 //报错
    //2.对应数组和对象元素修改，不算常量修改，修改值，不修改地址
    const TEAM = ['刘德华','张学友','郭富城'];
    TEAM.push('黎明');
    TEAM=[] // 报错
    console.log(TEAM)
</script>
```

> 模板字符串（template string）是增强版的字符串，用反引号（`）标识。

1、字符串中可以出现换行符；

2、可以使用 ${xxx} 形式输出变量和拼接变量；

```
<script>
    // 1 多行普通字符串
    let ulStr =
        '<ul>'+
        '<li>JAVA</li>'+
        '<li>html</li>'+
        '<li>VUE</li>'+
        '</ul>'
    console.log(ulStr)
    // 2 多行模板字符串
    let ulStr2 = `
        <ul>
            <li>JAVA</li>
            <li>html</li>
            <li>VUE</li>
```

```
        </ul>`
    console.log(ulStr2)
    // 3  普通字符串拼接
    let name ='张小明'
    let infoStr =name+'被评为本年级优秀学员'
    console.log(infoStr)
    // 4  模板字符串拼接
    let infoStr2 =`${name}被评为本年级优秀学员`
    console.log(infoStr2)
</script>
```

2.3 es6的解构表达式

> ES6 的解构赋值是一种方便的语法，可以快速将数组或对象中的值拆分并赋值给变量。解构赋值的语法使用花括号 `{}` 表示对象，方括号 `[]` 表示数组。通过解构赋值，函数更方便进行参数接受等！

> **数组解构赋值**：

```
let [a, b, c, d = 4] = [1, 2, 3];
console.log(a,b,c,d); // 4
```

- 该语句将数组 [1, 2, 3] 中的第一个值赋值给 a 变量，第二个值赋值给 b 变量，第三个值赋值给 c 变量。可以使用默认值为变量提供备选值，在数组中缺失对应位置的值时使用该默认值。

> **对象解构赋值**：

```
let {a, b} = {a: 1, b: 2};
//新增变量名必须和属性名相同，本质是初始化变量的值为对象中同名属性的值
//等价于 let a = 对象.a  let b = 对象.b
console.log(a,b);
```

- 该语句将对象 {a: 1, b: 2} 中的 a 属性值赋值给 a 变量，b 属性值赋值给 b 变量。可以为标识符分配不同的变量名称，使用 : 操作符指定新的变量名。例如：

```
let {a: x, b: y} = {a: 1, b: 2};
console.log(x,y);
```

> **函数参数解构赋值**：

- 解构赋值也可以用于函数参数，例如：

```
function add([x, y]) {
  return x + y;
}
add([1, 2]);
```

- 该函数接受一个数组作为参数，将其中的第一个值赋给 x，第二个值赋给 y，然后返回它们的和；
- ES6 解构赋值让变量的初始化更加简单和便捷。通过解构赋值，我们可以访问到对象中的属性，并将其赋值给对应的变量，从而提高代码的可读性和可维护性；

2.4 es6的箭头函数

ES6 允许使用"箭头" 义函数。语法类似Java中的Lambda表达式。

2.4.1 声明和特点

```
<script>
    //ES6 允许使用"箭头"(=>)定义函数。
    //1. 函数声明
    let fn1 = function(){}
    let fn2 = ()=>{} //箭头函数,此处不需要书写function关键字
    let fn3 = x =>{} //单参数可以省略(),多参数无参数不可以!
    let fn4 = x => console.log(x) //只有一行方法体可以省略{};
    let fun5 = x => x + 1 //当函数体只有一句返回值时，可以省略花括号和 return 语句
    //2. 使用特点 箭头函数this关键字
    // 在 JavaScript 中，this 关键字通常用来引用函数所在的对象，
    // 或者在函数本身作为构造函数时，来引用新对象的实例。
    // 但是在箭头函数中，this 的含义与常规函数定义中的含义不同，
    // 并且是由箭头函数定义时的上下文来决定的，而不是由函数调用时的上下文来决定的。
    // 箭头函数没有自己的this，this指向的是外层上下文环境的this
    let person ={
        name:"张三",
        showName:function (){
            console.log(this) //  这里的this是person
            console.log(this.name)
        },
        viewName: () =>{
            console.log(this) //  这里的this是window
            console.log(this.name)
        }
    }
    person.showName()
    person.viewName()
    //this应用
    function Counter() {
        this.count = 0;
        setInterval(() => {
            // 这里的 this 是上一层作用域中的 this，即 Counter实例化对象
            this.count++;
            console.log(this.count);
        }, 1000);
    }
    let counter = new Counter();
</script>
```

2.4.2 实践和应用场景

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Document</title>
    <style>
        #xdd{
            display: inline-block;
            width: 200px;
            height: 200px;
            background-color: red;
```

```
        }
    </style>
</head>
<body>
    <div id="xdd"></div>
    <script>
       let xdd = document.getElementById("xdd");
       // 方案1
       xdd.onclick = function(){
            console.log(this)
            let _this= this;  //this 是xdd
            //开启定时器
            setTimeout(function(){
                console.log(this)
                //变粉色
                _this.style.backgroundColor = 'pink';
            },2000);
        }
        // 方案2
        xdd.onclick = function(){
            console.log(this)
            //开启定时器
            setTimeout(()=>{
                console.log(this)// 使用setTimeout() 方法所在环境时的this对象
                //变粉色
                this.style.backgroundColor = 'pink';
            },2000);
        }
    </script>
</body>
</html>
```

2.4.3 rest和spread

> rest参数，在形参上使用，和JAVA中的可变参数几乎一样：

```
<script>
    // 1 参数列表中多个普通参数  普通函数和箭头函数中都支持
    let fun1 = function (a,b,c,d=10){console.log(a,b,c,d)}
    let fun2 = (a,b,c,d=10) =>{console.log(a,b,c,d)}
    fun1(1,2,3)
    fun2(1,2,3,4)
    // 2 ...作为参数列表,称之为rest参数 普通函数和箭头函数中都支持 ,因为箭头函数中无法使用
arguments,rest是一种解决方案
    let fun3 = function (...args){console.log(args)}
    let fun4 = (...args) =>{console.log(args)}
    fun3(1,2,3)
    fun4(1,2,3,4)
    // rest参数在一个参数列表中的最后一个只,这也就无形之中要求一个参数列表中只能有一个rest参数
    //let fun5 =  (...args,...args2) =>{} // 这里报错
</script>
```

> spread参数，在实参上使用：

```
<script>
    let arr =[1,2,3]
```

```
    //let arrSpread = ...arr;// 这样不可以,...arr必须在调用方法时作为实参使用
    let fun1 =(a,b,c) =>{
        console.log(a,b,c)
    }
    // 调用方法时,对arr进行转换 转换为1,2,3
    fun1(...arr)
    //应用场景1 合并数组
    let arr2=[4,5,6]
    let arr3=[...arr,...arr2]
    console.log(arr3)
    //应用场景2 合并对象属性
    let p1={name:"张三"}
    let p2={age:10}
    let p3={gender:"boy"}
    let person ={...p1,...p2,...p3}
    console.log(person)
</script>
```

2.5 es6的对象创建和拷贝

2.5.1 对象创建的语法糖

> ES6中新增了对象创建的语法糖，支持了class extends constructor等关键字，让ES6的语法和面向对象的语法更加接近。

```
<script>
    class Person{
      // 属性
      #n;
      age;
      get name(){
          return this.#n;
      }
      set name(n){
          this.#n =n;
      }
      // 实例方法
      eat(food){
          console.log(this.age+"岁的"+this.#n +"用筷子吃"+food)
      }
      // 静态方法
      static sum(a,b){
          return a+b;
      }
      // 构造器
      constructor(name,age){
          this.#n=name;
          this.age = age;
      }
  }
  let person =new Person("张三",10);
  // 访问对象属性
  // 调用对象方法
  console.log(person.name)
  console.log(person.#n ) // 私有 报错
  person.name="小明"
```

```
  console.log(person.age)
  person.eat("火锅")
  console.log(Person.sum(1,2))
  class Student extends  Person{
      grade ;
      score ;
      study(){
      }
      constructor(name,age ) {
          super(name,age);
      }
  }
  let stu =new Student("学生小李",18);
  stu.eat("面条")
</script>
```

2.5.2 对象的深拷贝和浅拷贝

> 对象的拷贝，快速获得一个和已有对象相同的对象的方式：

- 浅拷贝：

```
<script>
    let arr  =['java','c','python']
    let person ={
        name:'张三',
        language:arr
    }
    // 浅拷贝,person2和person指向相同的内存
    let person2 = person;
    person2.name="小黑"
    console.log(person.name)
</script>
```

- 深拷贝：

```
<script>
    let arr  =['java','c','python']
    let person ={
        name:'张三',
        language:arr
    }
    // 深拷贝,通过JSON和字符串的转换形成一个新的对象
    let person2 = JSON.parse(JSON.stringify(person))
    person2.name="小黑"
    console.log(person.name)
    console.log(person2.name)
</script>
```

2.6 es6的模块化处理

2.6.1模块化介绍

> 模块化是一种组织和管理前端代码的方式，将代码拆分成小的模块单元，使得代码更易于维护、扩展和复用。它包括了定义、导出、导入以及管理模块的方法和规范。前端模块化的主要优势如下：

1. 提高代码可维护性：通过将代码拆分为小的模块单元，使得代码结构更为清晰，可读性更高，便于开发者阅读和维护；
2. 提高代码可复用性：通过将重复使用的代码变成可复用的模块，减少代码重复率，降低开发成本；
3. 提高代码可扩展性：通过模块化来实现代码的松耦合，便于更改和替换模块，从而方便地扩展功能；

> 目前，前端模块化有多种规范和实现，包括 CommonJS、AMD 和 ES6 模块化。ES6 模块化是 JavaScript 语言的模块标准，使用 import 和 export 关键字来实现模块的导入和导出。现在，大部分浏览器都已经原生支持 ES6 模块化，因此它成为了最为广泛使用的前端模块化标准.。

- ES6模块化的几种暴露和导入方式：
    1. 分别导出；
    2. 统一导出；
    3. 默认导出；
- `ES6中无论以何种方式导出，导出的都是一个对象，导出的内容都可以理解为是向这个对象中添加属性或者方法`！！！

2.6.2 分别导出

- module.js 向外分别暴露成员：

```
//1.分别暴露
// 模块想对外导出,添加export关键字即可!
// 导出一个变量
export const PI = 3.14
// 导出一个函数
export function sum(a, b) {
  return a + b;
}
// 导出一个类
export class Person {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }
  sayHello() {
    console.log(`Hello, my name is ${this.name}, I'm ${this.age} years old.`);
  }
}
```

- app.js 导入module.js中的成员：

```
/*
    *代表module.js中的所有成员
    m1代表所有成员所属的对象
*/
import * as m1 from './module.js'
// 使用暴露的属性
console.log(m1.PI)
// 调用暴露的方法
let result =m1.sum(10,20)
console.log(result)
// 使用暴露的Person类
let person =new m1.Person('张三',10)
person.sayHello()
```

- index.html作为程序启动的入口，导入app.js：

```
<!-- 导入JS文件 添加type='module' 属性,否则不支持ES6的模块化 -->
<script src="./app.js" type="module" />
```

2.6.3 统一导出

- module.js向外统一导出成员：

```
//2.统一暴露
// 模块想对外导出,export统一暴露想暴露的内容!
// 定义一个常量
const PI = 3.14
// 定义一个函数
function sum(a, b) {
  return a + b;
}
// 定义一个类
class Person {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }
  sayHello() {
    console.log(`Hello, my name is ${this.name}, I'm ${this.age} years old.`);
  }
}
// 统一对外导出(暴露)
export {
    PI,
    sum,
    Person
}
```

- app.js导入module.js中的成员：

```
/*
```

```
    {}中导入要使用的来自于module.js中的成员
    {}中导入的名称要和module.js中导出的一致,也可以在此处起别名
    {}中如果定义了别名,那么在当前模块中就只能使用别名
    {}中导入成员的顺序可以不是暴露的顺序
    一个模块中可以同时有多个import
    多个import可以导入多个不同的模块,也可以是同一个模块
*/
//import {PI ,Person ,sum }  from './module.js'
//import {PI as pi,Person as People,sum as add}  from './module.js'
import {PI ,Person ,sum,PI as pi,Person as People,sum as add}  from './module.js'
// 使用暴露的属性
console.log(PI)
console.log(pi)
// 调用暴露的方法
let result1 =sum(10,20)
console.log(result1)
let result2 =add(10,20)
console.log(result2)
// 使用暴露的Person类
let person1 =new Person('张三',10)
person1.sayHello()
let person2 =new People('李四',11)
person2.sayHello()
```

2.6.4 默认导出

- modules混合向外导出:

```
// 3默认和混合暴露
/*
    默认暴露语法  export default sum
    默认暴露相当于是在暴露的对象中增加了一个名字为default的属性
    三种暴露方式可以在一个module中混合使用
*/
export const PI = 3.14
// 导出一个函数
function sum(a, b) {
  return a + b;
}
// 导出一个类
class Person {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }
  sayHello() {
    console.log(`Hello, my name is ${this.name}, I'm ${this.age} years old.`);
  }
}
```

```
// 导出默认
export default sum
// 统一导出
export {
   Person
}
```

- app.js 的default和其他导入写法混用：

```
/*
    *代表module.js中的所有成员
    m1代表所有成员所属的对象
*/
import * as m1 from './module.js'
import {default as add} from './module.js' // 用的少
import add2 from './module.js' // 等效于 import {default as add2} from './module.js'
// 调用暴露的方法
let result =m1.default(10,20)
console.log(result)
let result2 =add(10,20)
console.log(result2)
let result3 =add2(10,20)
console.log(result3)

// 引入其他方式暴露的内容
import {PI,Person} from './module.js'
// 使用暴露的Person类
let person =new Person('张三',10)
person.sayHello()
// 使用暴露的属性
console.log(PI)
```

三、前端工程化环境搭建

3.1 Nodejs的简介和安装

3.1.1 什么是Nodejs

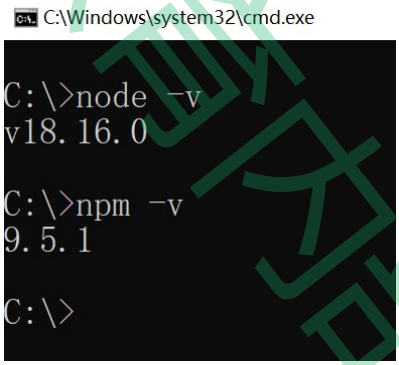
> Node.js 是一个基于 Chrome V8 引擎的 JavaScript 运行时环境，可以使 JavaScript 运行在服务器端。使用 Node.js，可以方便地开发服务器端应用程序，如 Web 应用、API、后端服务，还可以通过 Node.js 构建命令行工具等。相比于传统的服务器端语言（如 PHP、Java、Python 等），Node.js 具有以下特点：

- 单线程，但是采用了事件驱动、异步 I/O 模型，可以处理高并发请求；
- 轻量级，使用 C++ 编写的 V8 引擎让 Node.js 的运行速度很快；
- 模块化，Node.js 内置了大量模块，同时也可以通过第三方模块扩展功能；
- 跨平台，可以在 Windows、Linux、Mac 等多种平台下运行；

> Node.js 的核心是其管理事件和异步 I/O 的能力。Node.js 的异步 I/O 使其能够处理大量并发请求，并且能够避免在等待 I/O 资源时造成的阻塞。此外，Node.js 还拥有高性能网络库和文件系统库，可用于搭建 WebSocket 服务器、上传文件等。`在 Node.js 中，我们可以使用 JavaScript 来编写服务器端程序，这也使得前端开发人员可以利用自己已经熟悉的技能来开发服务器端程序，同时也让 JavaScript 成为一种全栈语言。` Node.js 受到了广泛的应用，包括了大型企业级应用、云计算、物联网、游戏开发等领域。常用的 Node.js 框架包括 Express、Koa、Egg.js 等，它们能够显著提高开发效率和代码质量。

3.1.2 如何安装Nodejs

1. 打开官网 https://nodejs.org/en 下载对应操作系统的 LTS 版本。
2. 双击安装包进行安装，安装过程中遵循默认选项即可(或者参照 https://www.runoob.com/nodejs/nodejs-install-setup.html)。安装完成后，可以在命令行终端输入 `node -v` 和 `npm -v` 查看 Node.js 和 npm 的版本号。



3. 定义一个app.js文件，文件内定义如下代码。cmd到该文件所在目录，然后在dos上通过 `node app.js` 命令即可运行。

```
function sum(a,b){
    return a+b;
}
function main(){
    console.log(sum(10,20))
}
main()
```

3.2 npm 配置和使用

3.2.1 npm介绍

> NPM全称Node Package Manager，是Node.js包管理工具，是全球最大的模块生态系统，里面所有的模块都是开源免费的；也是Node.js的包管理工具，相当于后端的Maven的部分功能 。

3.2.2 npm 安装和配置

> 1、安装：安装Nodejs，自动安装npm包管理工具！

> 2、配置依赖下载使用阿里镜像：

- npm 安装依赖包时默认使用的是官方源，由于国内网络环境的原因，有时会出现下载速度过慢的情况。为了解决这个问题，可以配置使用阿里镜像来加速 npm 的下载速度，打开命令行终端，执行以下命令，配置使用阿里镜像：

```
npm config set registry https://registry.npmmirror.com
```

- 验证配置已，查看当前 registry 的配置：如果输出结果为 `https://registry.npmmirror.com`，说明配置已成功生效。

```
npm config get registry
```

- 如果需要恢复默认的官方源，可以执行以下命令：

```
npm config set registry https://registry.npmjs.org/
```

> 3、配置全局依赖下载后存储位置：

- 在 Windows 系统上，npm 的全局依赖默认安装在 `<用户目录>\AppData\Roaming\npm` 目录下。
- 如果需要修改全局依赖的安装路径，可以按照以下步骤操作：
  1. 创建一个新的全局依赖存储目录，例如 `D:\GlobalNodeModules`。
  2. 打开命令行终端，执行以下命令来配置新的全局依赖存储路径：

     ```
     npm config set prefix "D:\GlobalNodeModules"
     ```

  3. 确认配置已生效，可以使用以下命令查看当前的全局依赖存储路径：

     ```
     npm config get prefix
     ```

> 4、升级npm版本：

- cmd 输入npm -v 查看版本，如果node中自带的npm版本过低！则需要升级至9.6.6！

```
npm install -g npm@9.6.6
```

3.2.3 npm 常用命令

> 1、项目初始化：

- npm init
  - 进入一个vscode创建好的项目中，执行 npm init 命令后，npm 会引导您在命令行界面上回答一些问题，例如项目名称、版本号、作者、许可证等信息，并最终生成一个package.json 文件。package.json信息会包含项目基本信息！类似maven的pom.xml。
- npm init -y
  - 执行，-y yes的意思，所有信息使用当前文件夹的默认值！不用挨个填写！

> 2、安装依赖（查看所有依赖地址 **https://www.npmjs.com** ）：

- npm install 包名 或者 npm install 包名@版本号
  - 安装包或者指定版本的依赖包(安装到当前项目中)。
- npm install -g 包名
  - 安装全局依赖包(安装到d:/GlobalNodeModules)则可以在任何项目中使用它，而无需在每个项目中独立安装该包。
- npm install
  - 安装package.json中的所有记录的依赖。

> 3、升级依赖：

- npm update 包名
  - 将依赖升级到最新版本。

> 4、卸载依赖：

- npm uninstall 包名

> 5、查看依赖：

- npm ls
  - 查看项目依赖。
- npm list -g
  - 查看全局依赖。

> 6、运行命令：

- npm run 命令是在执行 npm 脚本时使用的命令。npm 脚本是一组在 package.json 文件中定义的可执行命令。npm 脚本可用于启动应用程序，运行测试，生成文档等，还可以自定义命令以及配置需要运行的脚本。
- 在 package.json 文件中，scripts 字段是一个对象，其中包含一组键值对，键是要运行的脚本的名称，值是要执行的命令。例如，以下是一个简单的 package.json 文件：

```
{
    "name": "my-app",
    "version": "1.0.0",
    "scripts": {
        "start": "node index.js",
```

```
        "test": "jest",
        "build": "webpack"
    },
    "dependencies": {
        "express": "^4.17.1",
        "jest": "^27.1.0",
        "webpack": "^5.39.0"
    }
}
```

- scripts 对象包含 start、test 和 build 三个脚本。当您运行 npm run start 时，将运行 node index.js，并启动应用程序。同样，运行 npm run test 时，将运行 Jest 测试套件，而 npm run build 将运行 webpack 命令以生成最终的构建输出。
- 总之，npm run 命令为您提供了一种在 package.json 文件中定义和管理一组指令的方法，可以在项目中快速且灵活地运行各种操作。

四、Vue3简介和快速体验

4.1 Vue3介绍

渐进式

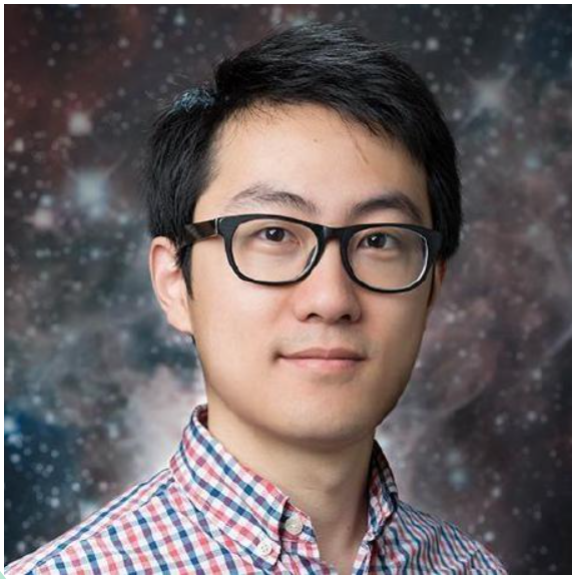
JavaScript 框架

> Vue (发音为 /vjuː/，类似 **view**) 是一款用于构建用户界面的 JavaScript 框架。它基于标准 HTML、CSS 和 JavaScript 构建，并提供了一套声明式的、组件化的编程模型，帮助你高效地开发用户界面。无论是简单还是复杂的界面，Vue 都可以胜任。官网为: **https://cn.vuejs.org/**

Vue的两个核心功能：

- **声明式渲染**：Vue 基于标准 HTML 拓展了一套模板语法，使得我们可以声明式地描述最终输出的 HTML 和 JavaScript 状态之间的关系；
- **响应性**：Vue 会自动跟踪 JavaScript 状态并在其发生变化时响应式地更新 DOM ；

> VUE作者:尤雨溪



- 尤雨溪（Evan You），毕业于科尔盖特大学，前端框架Vue.js的作者、HTML5版Clear的打造人、独立开源开发者。曾就职于Google Creative Labs和Meteor Development Group。由于工作中大量接触开源的JavaScript项目，最后自己也走上了开源之路，现全职开发和维护Vue.js；
- 尤雨溪毕业于上海复旦附中，在美国完成大学学业，本科毕业于Colgate University，后在Parsons设计学院获得Design & Technology艺术硕士学位，任职于纽约Google Creative Lab；
- 尤雨溪大学专业并非是计算机专业，在大学期间他学习专业是室内艺术和艺术史，后来读了美术设计和技术的硕士，正是在读硕士期间，他偶然接触到了JavaScript，从此被这门编程语言深深吸引，开启了自己的前端生涯；

4.2 Vue3快速体验(非工程化方式)

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8">
        <meta http-equiv="X-UA-Compatible" content="IE=edge">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
        <title>Document</title>
    </head>
    <body>
        <!-- 这里也可以用浏览器打开连接,然后将获得的文本单独保存进入一个vue.js的文件,导入vue.js文件即可 -->
        <script src="https://unpkg.com/vue@3/dist/vue.global.js"></script>
        <div id="app">
            <!-- 给style属性绑定colorStyle数据 -->
            <!-- {{插值表达式 直接将数据放在该位置}} -->
           <h1 v-bind:style="colorStyle">{{headline}}</h1>
           <!-- v-text设置双标签中的文本 -->
           <p v-text="article"></p>
           <!-- 给type属性绑定inputType数据 -->
           <input v-bind:type ="inputType" value="helloVue3"> <br>
           <!-- 给按钮单击事件绑定函数 -->
           <button  @click="sayHello()">hello</button>
        </div>
        <script>
            //组合api
            const app = Vue.createApp({
                // 在setup内部自由声明数据和方法即可!最终返回!
                setup(){
                    //定义数据
```

```
                    //在VUE中实现DOM的思路是: 通过修改修数据而影响页面元素
                    // vue3中,数据默认不是响应式的,需要加ref或者reactive处理,后面会详细讲解
                    let inputType ='text'
                    let headline ='hello vue3'
                    let article ='vue is awesome'
                    let colorStyle ={'color':'red'}
                    // 定义函数
                    let sayHello =()=>{
                        alert("hello Vue")
                    }
                    //在setup函数中,return返回的数据和函数可以在html使用
                    return {
                       inputType,
                       headline,
                       article,
                       colorStyle,
                       sayHello
                    }
                }
            });
            //挂载到视图
            app.mount("#app");
        </script>
    </body>
</html>
```

五、Vue3通过Vite实现工程化

5.1 Vite的介绍

Vite

下一代的前端工具链

> 在浏览器支持 ES 模块之前，JavaScript 并没有提供原生机制让开发者以模块化的方式进行开发。这也正是我们对 “打包” 这个概念熟悉的原因：使用工具抓取、处理并将我们的源码模块串联成可以在浏览器中运行的文件。时过境迁，我们见证了诸如 webpack、 Rollup 和 Parcel 等工具的变迁，它们极大地改善了前端开发者的开发体验。

- 当我们开始构建越来越大型的应用时，需要处理的 JavaScript 代码量也呈指数级增长；
- 包含数千个模块的大型项目相当普遍。基于 JavaScript 开发的工具就会开始遇到性能瓶颈：通常需要很长时间（甚至是几分钟！）才能启动开发服务器，即使使用模块热替换（HMR），文件修改后的效果也需要几秒钟才能在浏览器中反映出来。如此循环往复，迟钝的反馈会极大地影响开发者的开发效率和幸福感；

> Vite 旨在利用生态系统中的新进展解决上述问题：浏览器开始原生支持 ES 模块，且越来越多 JavaScript 工具使用编译型语言编写。 https://cn.vitejs.dev/guide/why.html 。前端工程化的作用包括但不限于以下几个方面：

1. 快速创建项目：使用脚手架可以快速搭建项目基本框架，避免从零开始搭建项目的重复劳动和繁琐操作，从而节省时间和精力；
2. 统一的工程化规范：前端脚手架可以预设项目目录结构、代码规范、git提交规范等统一的工程化规范，让不同开发者在同一个项目上编写出风格一致的代码，提高协作效率和质量；
3. 代码模板和组件库：前端脚手架可以包含一些常用的代码模板和组件库，使开发者在实现常见功能时不再重复造轮子，避免因为轮子质量不高带来的麻烦，能够更加专注于项目的业务逻辑；
4. 自动化构建和部署：前端脚手架可以自动进行代码打包、压缩、合并、编译等常见的构建工作，可以通过集成自动化部署脚本，自动将代码部署到测试、生产环境等；

5.2 Vite创建Vue3工程化项目

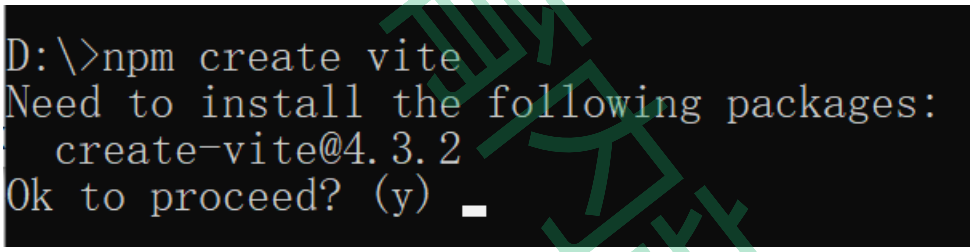
5.2.1 Vite+Vue3项目的创建、启动、停止

> 1 使用命令行创建工程。

- 在磁盘的合适位置上，创建一个空目录用于存储多个前端项目；
- 用vscode打开该目录；
- 在vocode中打开命令行运行如下命令；

```
npm create vite@latest
```

- 第一次使用Vite时会提示下载vite，输入y回车即可，下次使用Vite就不会出现了；



- 注意： 输入项目名称，选择Vue+JavaScript选项即可；

> 2 安装项目所需依赖：

- cd进入刚刚创建的项目目录；
- npm install命令安装基础依赖；

```
cd ./vue3-demo1
npm install
```

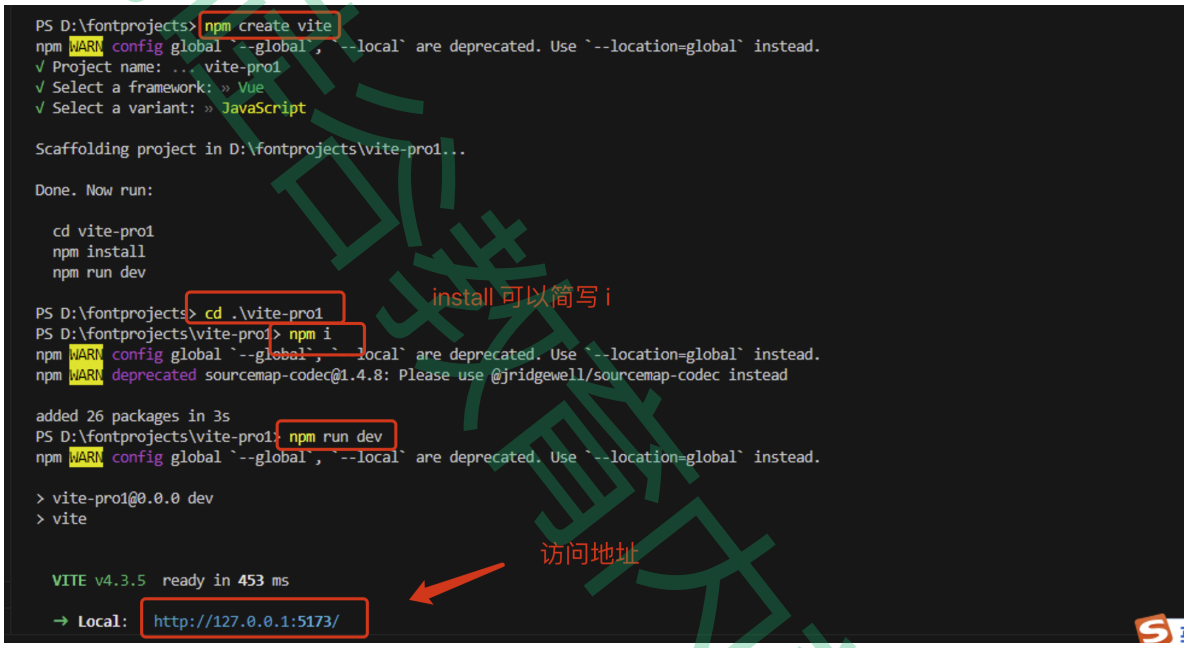
> 3 启动项目：

- 查看项目下的package.json

```
{
  "name": "vue3-demo1",
  "private": true,
  "version": "0.0.0",
  "type": "module",
  "scripts": {
```

```
    "dev": "vite",
    "build": "vite build",
    "preview": "vite preview"
  },
  "dependencies": {
    "bootstrap": "^5.2.3",
    "sass": "^1.62.1",
    "vue": "^3.2.47"
  },
  "devDependencies": {
    "@vitejs/plugin-vue": "^4.1.0",
    "vite": "^4.3.2"
  }
}
```

```
npm run dev
```

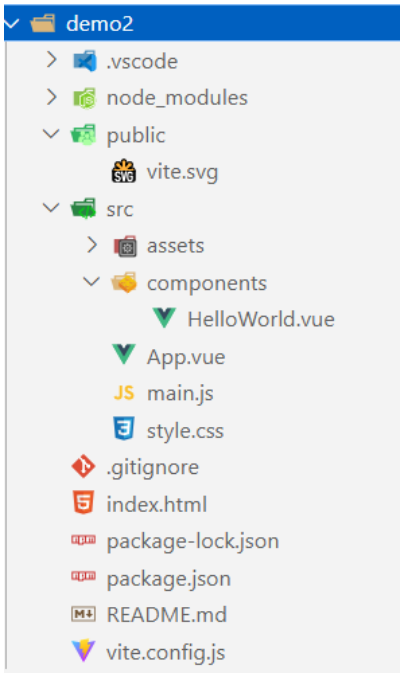


> 5 停止项目：

- 命令行上 ctrl+c

5.2.2 Vite+Vue3项目的目录结构

> 1.下面是 Vite 项目结构和入口的详细说明：



- public/ 目录：用于存放一些公共资源，如 HTML 文件、图像、字体等，这些资源会被直接复制到构建出的目标目录中。
- src/ 目录：存放项目的源代码，包括 JavaScript、CSS、Vue 组件、图像和字体等资源。在开发过程中，这些文件会被 Vite 实时编译和处理，并在浏览器中进行实时预览和调试。以下是src内部划分建议：
    1. `assets/` 目录：用于存放一些项目中用到的静态资源，如图片、字体、样式文件等。
    2. `components/` 目录：用于存放组件相关的文件。组件是代码复用的一种方式，用于抽象出一个可复用的 UI 部件，方便在不同的场景中进行重复使用。
    3. `layouts/` 目录：用于存放布局组件的文件。布局组件通常负责整个应用程序的整体布局，如头部、底部、导航菜单等。
    4. `pages/` 目录：用于存放页面级别的组件文件，通常是路由对应的组件文件。在这个目录下，可以创建对应的文件夹，用于存储不同的页面组件。
    5. `plugins/` 目录：用于存放 Vite 插件相关的文件，可以按需加载不同的插件来实现不同的功能，如自动化测试、代码压缩等。
    6. `router/` 目录：用于存放 Vue.js 的路由配置文件，负责管理视图和 URL 之间的映射关系，方便实现页面之间的跳转和数据传递。
    7. `store/` 目录：用于存放 Vuex 状态管理相关的文件，负责管理应用程序中的数据和状态，方便统一管理和共享数据，提高开发效率。
    8. `utils/` 目录：用于存放一些通用的工具函数，如日期处理函数、字符串操作函数等。
- vite.config.js 文件：Vite 的配置文件，可以通过该文件配置项目的参数、插件、打包优化等。该文件可以使用 CommonJS 或 ES6 模块的语法进行配置。
- package.json 文件：标准的 Node.js 项目配置文件，包含了项目的基本信息和依赖关系。其中可以通过 scripts 字段定义几个命令，如 dev、build、serve 等，用于启动开发、构建和启动本地服务器等操作。
- Vite 项目的入口为 src/main.js 文件，这是 Vue.js 应用程序的启动文件，也是整个前端应用程序的入口文件。在该文件中，通常会引入 Vue.js 及其相关插件和组件，同时会创建 Vue 实例，挂载到 HTML 页面上指定的 DOM 元素中。

> 2.vite的运行界面：

- 在安装了 Vite 的项目中，可以在 npm scripts 中使用 `vite` 可执行文件，或者直接使用 `npx vite` 运行它。下面是通过脚手架创建的 Vite 项目中默认的 npm scripts：(package.json)。

```
{
  "scripts": {
    "dev": "vite", // 启动开发服务器，别名：`vite dev`，`vite serve`
    "build": "vite build", // 为生产环境构建产物
    "preview": "vite preview" // 本地预览生产构建产物
  }
}
```

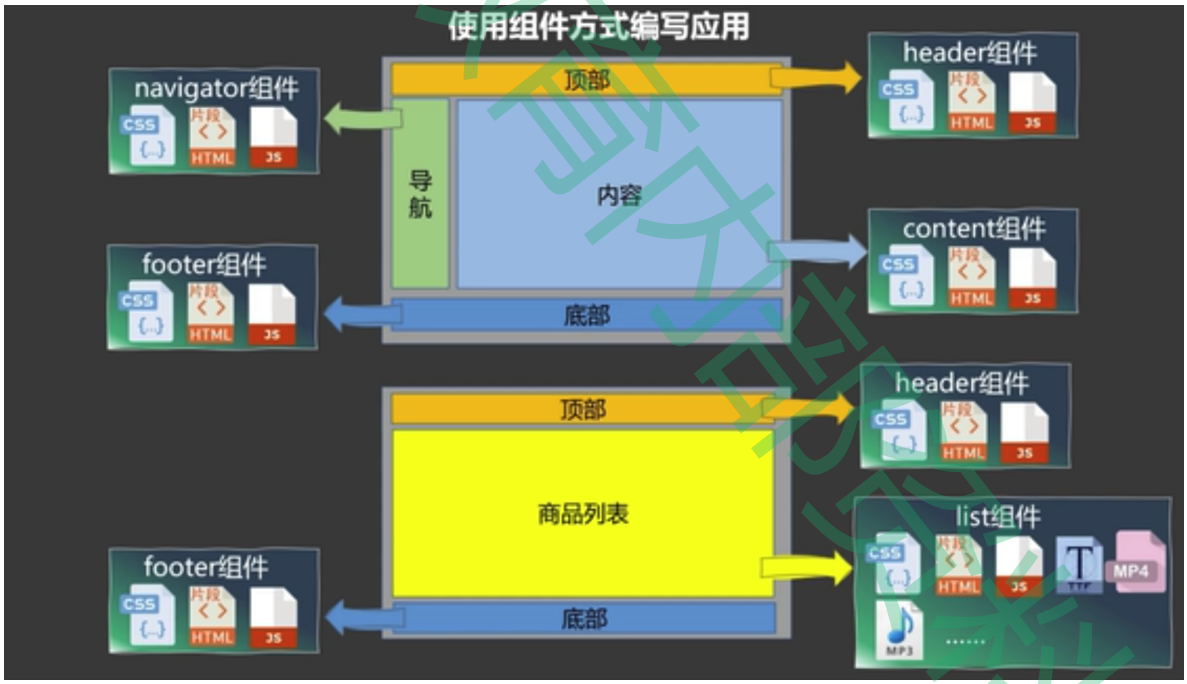
- 运行设置端口号：(vite.config.js)。

```
//修改vite项目配置文件 vite.config.js
export default defineConfig({
  plugins: [vue()],
  server:{
    port:3000
  }
})
```

5.2.3 Vite+Vue3项目组件(SFC入门)

> 什么是VUE的组件?

- 一个页面作为整体，是由多个部分组成的，每个部分在这里就可以理解为一个组件；
- 每个.vue文件就可以理解为一个组件，多个.vue文件可以构成一个整体页面；
- 组件化给我们带来的另一个好处就是组件的复用和维护非常的方便；

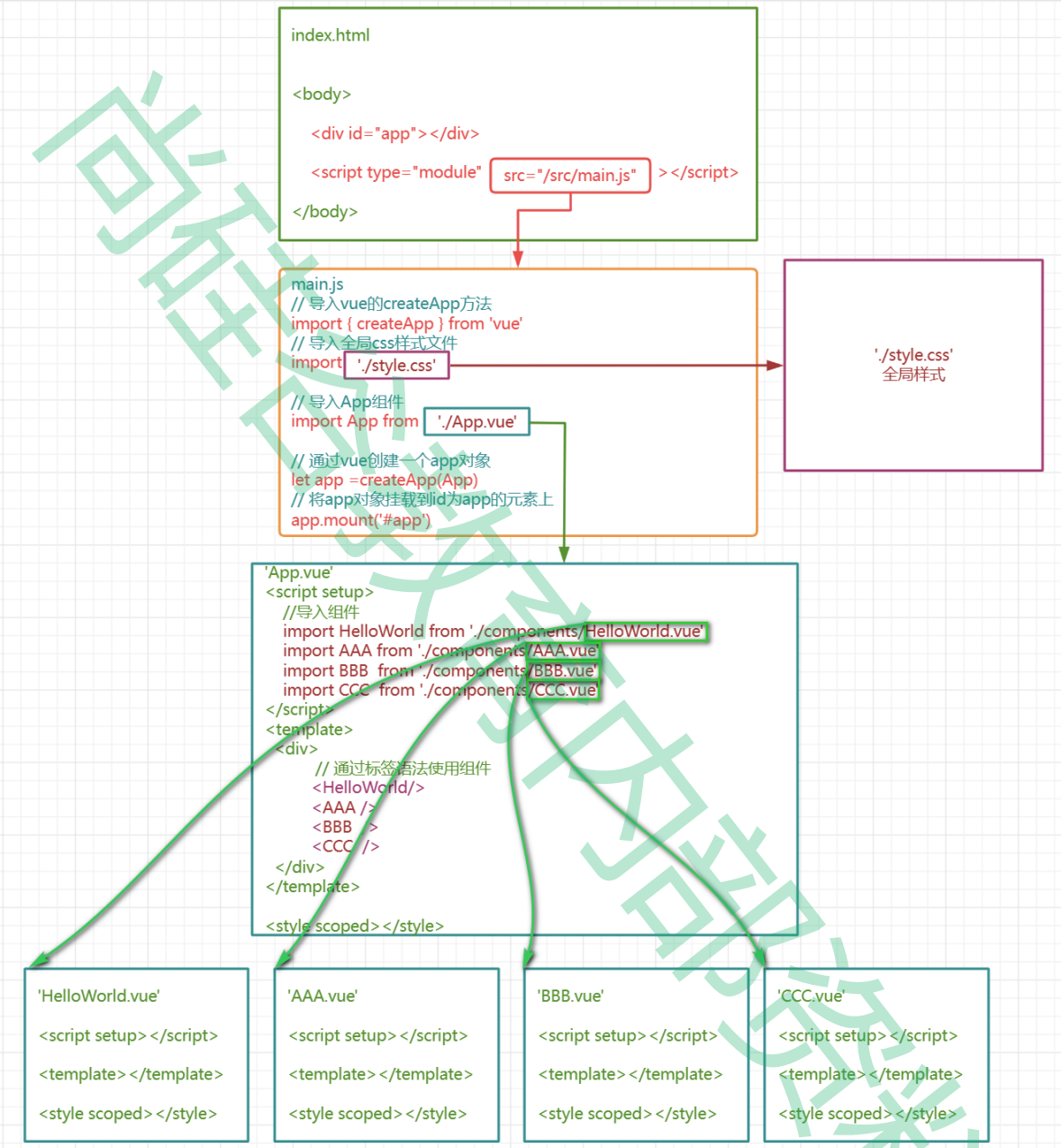


> 什么是.vue文件?

- 传统的页面有html文件css文件和js文件三个文件组成(多文件组件)；
- vue将这文件合并成一个vue文件(Single-File Component，简称 SFC，单文件组件)；
- vue文件对js/css/html统一封装，这是VUE中的概念，该文件由三个部分组成 `<script>` `<template>` `<style>`；
  - template标签　代表组件的html部分代码，代替传统的html文件；
  - script标签　　代表组件的js代码，代替传统的js文件；
  - style标签　　代表组件的css样式代码，代替传统的css文件 ；

> 工程化vue项目如何组织这些组件?

- index.html是项目的入口，其中 `<div id ='app'></div>` 是用于挂载所有组建的元素；
- index.html中的script标签引入了一个main.js文件，具体的挂载过程在main.js中执行；
- main.js是vue工程中非常重要的文件，他决定这项目使用哪些依赖，导入的第一个组件；
- App.vue是vue中的核心组件，所有的其他组件都要通过该组件进行导入，该组件通过路由可以控制页面的切换；



5.2.4 Vite+Vue3响应式入门和setup函数

> 1 使用vite创建一个 vue+JavaScript项目：

```
npm create vite
npm install
npm run dev
```

- App.vue

```
<script>
    //存储vue页面逻辑js代码
</script>
<template>
    <!-- 页面的样式的是html代码-->
</template>
<style scoped>
    /** 存储的是css代码! <style scoped> 是 Vue.js 单文件组件中用于设置组件样式的一种方式。
    它的含义是将样式局限在当前组件中，不对全局样式造成影响。 */
</style>
```

2 Vue3响应式数据入门:

```
<script type="module">
    //存储vue页面逻辑js代码
    import {ref} from 'vue'
    export default{
        setup(){
            //非响应式数据: 修改后VUE不会更新DOM
            //响应式数据:  修改后VUE会更新DOM
            //VUE2中数据默认是响应式的
            //VUE3中数据要经过ref或者reactive处理后才是响应式的
            //ref是VUE3框架提供的一个函数,需要导入
            //let counter = 1
            //ref处理的响应式数据在js编码修改的时候需要通过.value操作
            //ref响应式数据在绑定到html上时不需要.value
            let counter = ref(1)
            function increase(){
                // 通过.value修改响应式数据
                counter.value++
            }
            function decrease(){
                counter.value--
            }
            return {
                counter,
                increase,
                decrease
            }
        }
    }
</script>
<template>
    <div>
      <button @click="decrease()">-</button>
      {{ counter }}
      <button @click="increase()">+</button>
    </div>
</template>
<style scoped>
    button{
        border: 1px solid red;
    }
</style>
```

3 Vue3 setup函数和语法糖:

- 位置：src/App.vue。

```
<script type="module" setup>
/* 通过setup关键字，可以省略 export default {setup(){   return{}}}这些冗余的语法结构 */
    import {ref} from 'vue'
    // 定义响应式数据
    let counter = ref(1)
    // 定义函数
    function increase(){
        counter.value++
    }
    function decrease(){
        counter.value--
    }
</script>
<template>
    <div>
      <button @click="decrease()">-</button>
      {{ counter }}
      <button @click="increase()">+</button>
    </div>
</template>
<style scoped>
    button{
        border: 1px solid red;
    }
</style>
```

5.2.5 Vite+Vue3关于样式的导入方式

1. 全局引入main.js；

```
import './style/reset.css' //书写引入的资源的相对路径即可！
```

2. vue文件script代码引入；

```
import './style/reset.css'
```

3. Vue文件style代码引入；

```
@import './style/reset.css'
```

5.2.6 关于JS和TS选择的问题

> TS是JS的一个超集，使用TS之后，JS的语法更加的像JAVA，实际开发中用的确实更多，那么这里为什么我们没有立即使用TS进行开发，原因如下：

1 为了降低难度，提高前端工程化的学习效率；

2 对于学JAVA的我们来说，学习TS非常容易，但是还是需要一些时间；

3 TS不是非学不可，不用TS仍然可以正常开发工程化的前端项目；

4 尚硅谷已经发布了TS的专项课程，请大家在B站上自行搜索 "尚硅谷 TS"；

5 建议大家先学完完整的前端工程化内容，然后再根据需求单独学习TS即可；

六、Vue3视图渲染技术

6.1 模版语法

> Vue 使用一种基于 HTML 的模板语法，使我们能够声明式地将其组件实例的数据绑定到呈现的 DOM 上。所有的 Vue 模板都是语法层面合法的 HTML，可以被符合规范的浏览器和 HTML 解析器解析。在底层机制中，Vue 会将模板编译成高度优化的 JavaScript 代码。结合响应式系统，当应用状态变更时，Vue 能够智能地推导出需要重新渲染的组件的最少数量，并应用最少的 DOM 操作。

6.1.1 插值表达式和文本渲染

> 插值表达式：最基本的数据绑定形式是文本插值，它使用的是“Mustache”语法 ，即双大括号 `{{}}`：

- 插值表达式是将数据渲染到元素的指定位置的手段之一；
- 插值表达式不绝对依赖标签，其位置相对自由；
- 插值表达式中支持javascript的运算表达式；
- 插值表达式中也支持函数的调用；

```
<script setup type="module">
  let msg ="hello vue3"
  let getMsg= ()=>{
    return 'hello vue3 message'
  }
  let age = 19
  let bee = '蜜 蜂'
  // 购物车
  const carts = [{name:'可乐',price:3,number:10},{name:'薯片',price:6,number:8}];
  //计算购物车总金额
  function compute(){
      let count = 0;
      for(let index in carts){
          count += carts[index].price*carts[index].number;
      }
      return count;
  }
</script>
<template>
  <div>
    <h1>{{ msg }}</h1>
    msg的值为: {{ msg }} <br>
    getMsg返回的值为:{{ getMsg() }}  <br>
    是否成年: {{ age>=18?'true':'false' }} <br>
    反转: {{ bee.split(' ').reverse().join('-') }} <br>
    购物车总金额: {{ compute() }} <br/>
    购物车总金额: {{carts[0].price*carts[0].number + carts[1].price*carts[1].number}}
<br>
  </div>
</template>
<style scoped>
</style>
```

> 为了渲染双标中的文本，我们也可以选择使用 `v-text` 和 `v-html` 命令：

- v-*** 这种写法的方式使用的是vue的命令；
- v-***的命令必须依赖元素，并且要写在元素的开始标签中；
- v-***指令支持ES6中的字符串模板；
- 插值表达式中支持javascript的运算表达式；
- 插值表达式中也支持函数的调用；
- v-text可以将数据渲染成双标签中间的文本，但是不识别html元素结构的文本；
- v-html可以将数据渲染成双标签中间的文本，识别html元素结构的文本；

```
<script setup type="module">
  let msg ='hello vue3'
  let getMsg= ()=>{
    return msg
  }
  let age = 19
  let bee = '蜜 蜂'
  let redMsg ='<font color=\'red\'>msg</font>'
  let greenMsg =`<font color=\'green\'>${msg}</font>`
</script>
<template>
  <div>
    <span v-text='msg'></span> <br>
    <span v-text='redMsg'></span> <br>
    <span v-text='getMsg()'></span> <br>
    <span v-text='age>18?"成年":"未成年"'></span> <br>
    <span v-text='bee.split(" ").reverse().join("-")'></span> <br>
    <span v-html='msg'></span> <br>
    <span v-html='redMsg'></span> <br>
    <span v-html='greenMsg'></span> <br>
    <span v-html="`<font color='green'>${msg}</font>`"></span> <br>
  </div>
</template>
<style scoped>
</style>
```

6.1.2 Attribute属性渲染

> 想要渲染一个元素的 attribute，应该使用 `v-bind` 指令。

- 由于插值表达式不能直接放在标签的属性中，要渲染元素的属性就应该使用v-bind；
- v-bind可以用于渲染任何元素的属性，语法为 `v-bind:属性名='数据名'` ，可以简写为 `:属性名='数据名'` ；

```
<script setup type="module">
  const data = {
    name:'尚硅谷',
    url:"http://www.atguigu.com",
    logo:"http://www.atguigu.com/images/index_new/logo.png"
  }
</script>
<template>
  <div>
    <a
      v-bind:href='data.url'
      target="_self">
      <img
        :src="data.logo"
```

```
        :title="data.name">
      <br>
      <input type="button"
             :value="`点击访问${data.name}`">
    </a>
  </div>
</template>
<style scoped>
</style>
```

6.1.3 事件的绑定

> 我们可以使用 `v-on` 来监听 DOM 事件，并在事件触发时执行对应的 Vue的JavaScript代码。

- 用法：`v-on:click="handler"` 或简写为 `@click="handler"`；
- vue中的事件名=原生事件名去掉 `on` 前缀 如: `onClick --> click`；
- handler的值可以是方法事件处理器，也可以是内联事件处理器；
- 绑定事件时，可以通过一些绑定的修饰符，常见的事件修饰符如下：
  - `.once：只触发一次事件。[重点]`；
  - `.prevent：阻止默认事件。[重点]`；
  - .stop：阻止事件冒泡；
  - .capture：使用事件捕获模式而不是冒泡模式；
  - .self：只在事件发送者自身触发时才触发事件；

```
<script setup type="module">
  import {ref} from 'vue'
  // 响应式数据 当发生变化时,会自动更新 dom树
  let count=ref(0)
  let addCount= ()=>{
    count.value++
  }
  let incrCount= (event)=>{
    count.value++
    // 通过事件对象阻止组件的默认行为
    event.preventDefault();
  }
</script>
<template>
  <div>
    <h1>count的值是:{{ count }}</h1>
    <!-- 方法事件处理器 -->
    <button v-on:click="addCount()">addCount</button> <br>
    <!-- 内联事件处理器 -->
    <button @click="count++">incrCount</button> <br>
    <!-- 事件修饰符 once 只绑定事件一次 -->
    <button @click.once="count++">addOnce</button> <br>
    <!-- 事件修饰符 prevent 阻止组件的默认行为 -->
    <a href="http://www.atguigu.com" target="_blank"
@click.prevent="count++">prevent</a> <br>
    <!-- 原生js方式阻止组件默认行为 (推荐) -->
    <a href="http://www.atguigu.com" target="_blank"
@click="incrCount($event)">prevent</a> <br>
  </div>
</template>
```

```
<style scoped>
</style>
```

6.2 响应式基础

> 此处的响应式是指 ： 数据模型（自定义的变量、对象）发生变化时，自动更新DOM树内容，页面上显示的内容会进行同步变化。vue3的数据模型不是自动响应式的，需要我们做一些特殊的处理。

6.2.1 响应式需求案例

> 需求：实现 + - 按钮,实现数字加一减一：

```
<script type="module" setup>
    let counter = 0;
    function show(){
        alert(counter);
    }
</script>
<template>
  <div>
    <button @click="counter--">-</button>
    {{ counter }}
    <button @click="counter++">+</button>
    <hr>
    <!-- 此案例,我们发现counter值,会改变,但是页面不改变! 默认Vue3的数据是非响应式的!-->
    <button @click="show()">显示counter值</button>
   </div>
</template>
<style scoped>
</style>
```

6.2.2 响应式实现关键字ref

> `ref` 可以将一个基本类型的数据（如字符串，数字等）转换为一个响应式对象。 `ref` 只能包裹单一元素。

```
<script type="module" setup>
    /* 从vue中引入ref方法 */
    import {ref} from 'vue'
    let counter = ref(0);
    function show(){
        alert(counter.value);
    }
    /* 函数中要操作ref处理过的数据,需要通过.value形式 */
    let decr = () =>{
      counter.value--;
    }
    let incr = () =>{
      counter.value++;
    }
</script>
<template>
  <div>
```

```
    <button @click="counter--">-</button>
    <button @click="decr()">-</button>
    {{ counter }}
    <button @click="counter++">+</button>
    <button @click="incr()">+</button>
    <hr>
    <button @click="show()">显示counter值</button>
   </div>
</template>
<style scoped>
</style>
```

- 在上面的例子中，我们使用 `ref` 包裹了一个数字，在代码中给这个数字加 1 后，视图也会跟着动态更新。需要注意的是，由于使用了 `ref`，因此需要在访问该对象时使用 `.value` 来获取其实际值。

6.2.3 响应式实现关键字reactive

> 我们可以使用 reactive() 函数创建一个响应式对象或数组：

```
<script type="module" setup>
    /* 从vue中引入reactive方法 */
    import {ref,reactive} from 'vue'
    let data = reactive({
      counter:0
    })
    function show(){
        alert(data.counter);
    }
    /* 函数中要操作reactive处理过的数据,需要通过 对象名.属性名的方式 */
    let decr = () =>{
      data.counter--;
    }
    let incr = () =>{
      data.counter++;
    }
</script>
<template>
  <div>
    <button @click="data.counter--">-</button>
    <button @click="decr()">-</button>
    {{ data.counter }}
    <button @click="data.counter++">+</button>
    <button @click="incr()">+</button>
    <hr>
    <button @click="show()">显示counter值</button>
   </div>
</template>
<style scoped>
</style>
```

> 对比ref和reactive：

- 使用 `ref` 适用于以下开发场景：

    - 包装基本类型数据：ref 主要用于包装基本类型数据（如字符串、数字等），即只有一个值的数据，如果你想监听这个值的变化，用 ref最为方便。在组件中使用时也很常见。
    - 访问方式简单：ref 对象在访问时与普通的基本类型值没有太大区别，只需要通过 .value访问其实际值即可。
- 使用 `reactive` 适用于以下开发场景：
    - 包装复杂对象：reactive可以将一个普通对象转化为响应式对象，这样在数据变化时会自动更新界面，特别适用于处理复杂对象或者数据结构。
    - 需要递归监听的属性：使用 reactive 可以递归追踪所有响应式对象内部的变化，从而保证界面的自动更新。
- 综上所述，ref适用与简单情形下的数据双向绑定，对于只有一个字符等基本类型数据或自定义组件等情况，建议可以使用 ref；而对于对象、函数等较为复杂的数据结构，以及需要递归监听的属性变化，建议使用 reactive。当然，在实际项目中根据需求灵活选择也是十分必要的。

6.2.4 扩展响应式关键字toRefs 和 toRef

> toRef基于reactive响应式对象上的一个属性，创建一个对应的 ref响应式数据。这样创建的 ref 与其源属性保持同步：改变源属性的值将更新 ref 的值，反之亦然。toRefs将一个响应式对象多个属性转换为一个多个ref数据，这个普通对象的每个属性都是指向源对象相应属性的 ref。每个单独的 ref 都是使用 **toRef()** 创建的。

> 案例：响应显示reactive对象属性：

```
<script type="module" setup>
    /* 从vue中引入reactive方法 */
    import {ref,reactive,toRef,toRefs} from 'vue'
    let data = reactive({
      counter:0,
      name:"test"
    })
    // 将一个reactive响应式对象中的某个属性转换成一个ref响应式对象
    let ct =toRef(data,'counter');
    // 将一个reactive响应式对象中的多个属性转换成多个ref响应式对象
    let {counter,name} = toRefs(data)
    function show(){
        alert(data.counter);
        // 获取ref的响应对象,需要通过.value属性
        alert(counter.value);
        alert(name.value)
    }
    /* 函数中要操作ref处理过的数据,需要通过.value形式 */
    let decr = () =>{
      data.counter--;
    }
    let incr = () =>{
      /* ref响应式数据,要通过.value属性访问 */
      counter.value++;
    }
</script>
<template>
  <div>
    <button @click="data.counter--">-</button>
    <button @click="decr()">-</button>
    {{ data.counter }}
    &amp;
```

```
    {{ ct }}
    <button @click="data.counter++">+</button>
    <button @click="incr()">+</button>
    <hr>
    <button @click="show()">显示counter值</button>
   </div>
</template>
<style scoped>
</style>
```

6.3 条件和列表渲染

6.3.1 条件渲染

> `v-if` 条件渲染：

- `v-if='表达式'` 只会在指令的表达式返回真值时才被渲染
- 也可以使用 `v-else` 为 `v-if` 添加一个“else 区块”。
- 一个 `v-else` 元素必须跟在一个 `v-if` 元素后面，否则它将不会被识别。

```
<script type="module" setup>
    import {ref} from 'vue'
    let awesome = ref(true)
</script>
<template>
  <div>
    <h1 v-if="awesome">Vue is awesome!</h1>
    <h1 v-else>Oh no 😢</h1>
    <button @click="awesome = !awesome">Toggle</button>
  </div>
</template>
<style scoped>
</style>
```

> `v-show` 条件渲染扩展：

- 另一个可以用来按条件显示一个元素的指令是 `v-show`。其用法基本一样；
- 不同之处在于 `v-show` 会在 DOM 渲染中保留该元素；`v-show` 仅切换了该元素上名为 `display` 的 CSS 属性；
- `v-show` 不支持在 `<template>` 元素上使用，也不能和 `v-else` 搭配使用；

```
<script type="module" setup>
    import {ref} from 'vue'
    let awesome = ref(true)
</script>
<template>
  <div>
    <h1 id="ha"  v-show="awesome">Vue is awesome!</h1>
    <h1 id="hb"  v-if="awesome">Vue is awesome!</h1>
    <h1 id="hc"  v-else>Oh no 😢</h1>
    <button @click="awesome = !awesome">Toggle</button>
  </div>
</template>
<style scoped>
</style>
```

> `v-if` vs `v-show` :

- `v-if` 是"真实的"按条件渲染，因为它确保了在切换时，条件区块内的事件监听器和子组件都会被销毁与重建；
- `v-if` 也是**惰性**的：如果在初次渲染时条件值为 false，则不会做任何事。条件区块只有当条件首次变为 true 时才被渲染；
- 相比之下，`v-show` 简单许多，元素无论初始条件如何，始终会被渲染，只有 CSS `display` 属性会被切换；
- 总的来说，`v-if` 有更高的切换开销，而 `v-show` 有更高的初始渲染开销。因此，如果需要频繁切换，则使用 `v-show` 较好；如果在运行时绑定条件很少改变，则 `v-if` 会更合适；

6.3.2 列表渲染

> 我们可以使用 `v-for` 指令基于一个数组来渲染一个列表：

- `v-for` 指令的值需要使用 `item in items` 形式的特殊语法，其中 `items` 是源数据的数组，而 `item` 是迭代项的别名；
- 在 `v-for` 块中可以完整地访问父作用域内的属性和变量。`v-for` 也支持使用可选的第二个参数表示当前项的位置索引；

```
<script type="module" setup>
    import {ref,reactive} from 'vue'
    let parentMessage= ref('产品')
    let items =reactive([
      {
        id:'item1',
        message:"薯片"
      },
      {
        id:'item2',
        message:"可乐"
      }
    ])
</script>
<template>
  <div>
    <ul>
      <!-- :key不写也可以 -->
      <li v-for='item in items' :key='item.id'>
```

```
        {{ item.message }}
      </li>
    </ul>
    <ul>
      <!-- index表示索引 -->
      <li v-for="(item, index) in items" :key="index">
        {{ parentMessage }} - {{ index }} - {{ item.message }}
      </li>
    </ul>
  </div>
</template>
<style scoped>
</style>
```

- 案例：实现购物车显示和删除购物项

```
<script type="module" setup>
    //引入模块
    import { reactive} from 'vue'
    //准备购物车数据,设置成响应数据
    const carts = reactive([{name:'可乐',price:3,number:10},{name:'薯片',price:6,number:8}])
    //计算购物车总金额
    function compute(){
      let count = 0;
      for(let index in carts){
        count += carts[index].price*carts[index].number;
      }
      return count;
    }
    //删除购物项方法
    function removeCart(index){
      carts.splice(index,1);
    }
</script>
<template>
    <div>
        <table>
           <thead>
               <tr>
                  <th>序号</th>
                  <th>商品名</th>
                  <th>价格</th>
                  <th>数量</th>
                  <th>小计</th>
                  <th>操作</th>
               </tr>
           </thead>
           <tbody v-if="carts.length > 0">
               <!-- 有数据显示-->
               <tr v-for="cart,index in carts" :key="index">
                  <th>{{ index+1 }}</th>
                  <th>{{ cart.name }}</th>
                  <th>{{ cart.price + '元' }}</th>
                  <th>{{ cart.number }}</th>
                  <th>{{ cart.price*cart.number  + '元'}}</th>
                  <th> <button @click="removeCart(index)">删除</button> </th>
               </tr>
```

```
            </tbody>
            <tbody v-else>
                <!-- 没有数据显示-->
                <tr>
                   <td colspan="6">购物车没有数据!</td>
                </tr>
            </tbody>
        </table>
        购物车总金额: {{ compute() }} 元
    </div>
</template>
<style scoped>
</style>
```

6.4 双向绑定

> 单项绑定和双向绑定:

- 单向绑定：响应式数据的变化会更新dom树，但是dom树上用户的操作造成的数据改变 不会同步更新到响应式数据 ；
- 双向绑定：响应式数据的变化会更新dom树，但是dom树上用户的操作造成的数据改变 会同步更新到响应式数据 ；
  - 用户通过表单标签才能够输入数据，所以双向绑定都是应用到表单标签上的，其他标签不行；
  - v-model专门用于双向绑定表单标签的value属性，语法为 `v-model:value=''` ，可以简写为 `v-model=''` ；
  - v-model还可以用于各种不同类型的输入， `<textarea>` 、 `<select>` 元素；

```
<script type="module" setup>
  //引入模块
  import { reactive,ref} from 'vue'
  let hbs = ref([]); //装爱好的值
  let user = reactive({username:null,password:null,introduce:null,pro:null})
  function login(){
    alert(hbs.value);
    alert(JSON.stringify(user));
  }
  function clearx(){
    //user = {};// 这中写法会将数据变成非响应的,应该是user.username=""
    user.username=''
    user.password=''
    user.introduce=''
    user.pro=''
    hbs.value.splice(0,hbs.value.length);;
  }
</script>
<template>
  <div>
      账号:  <input type="text" placeholder="请输入账号! " v-model="user.username"> <br>
      密码:  <input type="text" placeholder="请输入账号! " v-model="user.password"> <br>
      爱好:
        吃 <input type="checkbox" name="hbs" v-model="hbs" value="吃">
        喝 <input type="checkbox" name="hbs" v-model="hbs" value="喝">
        玩 <input type="checkbox" name="hbs" v-model="hbs" value="玩">
```

```
        乐 <input type="checkbox" name="hbs" v-model="hbs" value="乐">
      <br>
      简介:<textarea v-model="user.introduce"></textarea>
      <br>
      籍贯:
          <select v-model="user.pro">
            <option value="1">黑</option>
            <option value="2">吉</option>
            <option value="3">辽</option>
            <option value="4">京</option>
            <option value="5">津</option>
            <option value="6">冀</option>
          </select>
      <br>
      <button @click="login()">登录</button>
      <button @click="clearx()">重置</button>
      <hr>
      显示爱好:{{ hbs }}
      <hr>
      显示用户信息:{{ user }}
  </div>
</template>
<style scoped>
</style>
```

6.5 属性计算

> 模板中的表达式虽然方便，但也只能用来做简单的操作。如果在模板中写太多逻辑，会让模板变得臃肿，难以维护。比如说，我们有这样一个包含嵌套数组的对象：

```
<script type="module" setup>
  //引入模块
  import { reactive,computed} from 'vue'
  const author = reactive({
    name: 'John Doe',
    books: [
      'Vue 2 - Advanced Guide',
      'Vue 3 - Basic Guide',
      'Vue 4 - The Mystery'
    ]
  })
</script>
<template>
  <div>
    <p>{{author.name}} Has published books?:</p>
    <span>{{ author.books.length > 0 ? 'Yes' : 'No' }}</span>
  </div>
</template>
<style scoped>
</style>
```

- 这里的模板看起来有些复杂。我们必须认真看好一会儿才能明白它的计算依赖于 `author.books`。更重要的是，如果在模板中需要不止一次这样的计算，我们可不想将这样的代码在模板里重复好多遍。

> 因此我们推荐使用**计算属性**来描述依赖响应式状态的复杂逻辑。这是重构后的示例：

```
<script type="module" setup>
  //引入模块
  import { reactive,computed} from 'vue'
  const author = reactive({
    name: 'John Doe',
    books: [
      'Vue 2 - Advanced Guide',
      'Vue 3 - Basic Guide',
      'Vue 4 - The Mystery'
    ]
  })
  // 一个计算属性 ref
  const publishedBooksMessage = computed(() => {
    console.log("publishedBooksMessage")
    return author.books.length > 0 ? 'Yes' : 'No'
  })
  // 一个函数
  let hasBooks = ()=>{
    console.log("hasBooks")
    return author.books.length > 0?'Yes':'no'
  }
</script>
<template>
  <div>
    <p>{{author.name}} Has published books?:</p>
    <span>{{ author.books.length > 0 ? 'Yes' : 'No' }}</span>
    <span>{{ hasBooks() }}</span><!-- 调用方法,每个标签都会调用一次 -->
    <span>{{ hasBooks() }}</span>

    <p>{{author.name}} Has published books?:</p>
    <span>{{ publishedBooksMessage }}</span><!-- 属性计算,属性值不变时,多个标签只会调用一次
-->
    <span>{{ publishedBooksMessage }}</span>
  </div>
</template>
<style scoped>
</style>
```

- 我们在这里定义了一个计算属性 `publishedBooksMessage`。`computed()` 方法期望接收一个 getter 函数，返回值为一个**计算属性 ref**。和其他一般的 ref 类似，你可以通过 `publishedBooksMessage.value` 访问计算结果。计算属性 ref 也会在模板中自动解包，因此在模板表达式中引用时无需添加 `.value`。
- Vue 的计算属性会自动追踪响应式依赖。它会检测到 `publishedBooksMessage` 依赖于 `author.books`，所以当 `author.books` 改变时，任何依赖于 `publishedBooksMessage` 的绑定都会同时更新。

> 计算属性缓存 vs 方法：

- 若我们将同样的函数定义为一个方法而不是计算属性，两种方式在结果上确实是完全相同的，然而，不同之处在于**计算属性值会基于其响应式依赖被缓存**。一个计算属性仅会在其响应式依赖更新时才重新计算。这意味着只要 `author.books` 不改变，无论多少次访问 `publishedBooksMessage` 都会立即返回先前的计算结果!

6.6 数据监听器

> 计算属性允许我们声明性地计算衍生值。然而在有些情况下，我们需要在状态变化时执行一些“副作用”：例如更改 DOM，或是根据异步操作的结果去修改另一处的状态。我们可以使用 **watch 函数** 在每次响应式状态发生变化时触发回调函数：

- watch主要用于以下场景：
  - 当数据发生变化时需要执行相应的操作；
  - 监听数据变化，当满足一定条件时触发相应操作；
  - 在异步操作前或操作后需要执行相应的操作；

> 监控响应式数据（watch）：

```
<script type="module" setup>
  import { ref,reactive,watch} from 'vue'
  let firstname=ref('')
  let lastname=reactive({name:''})
  let fullname=ref('')
  //监听一个ref响应式数据
  watch(firstname,(newValue,oldValue)=>{
    console.log(`${oldValue}变为${newValue}`)
    fullname.value=firstname.value+lastname.name
  })
  //监听reactive响应式数据的指定属性
  watch(()=>lastname.name,(newValue,oldValue)=>{
    console.log(`${oldValue}变为${newValue}`)
    fullname.value=firstname.value+lastname.name
  })
  //监听reactive响应式数据的所有属性(深度监视,一般不推荐)
  //deep:true 深度监视
  //immediate:true 深度监视在进入页面时立即执行一次
  watch(()=>lastname,(newValue,oldValue)=>{
    // 此时的newValue和oldValue一样,都是lastname
    console.log(newValue)
    console.log(oldValue)
    fullname.value=firstname.value+lastname.name
  },{deep:true,immediate:false})
</script>
<template>
  <div>
    全名:{{fullname}} <br>
    姓氏:<input type="text" v-model="firstname"> <br>
    名字:<input type="text" v-model="lastname.name" > <br>
  </div>
</template>
<style scoped>
</style>
```

> 监控响应式数据(watchEffect)：

- watchEffect默认监听所有的响应式数据

```
<script type="module" setup>
  import { ref,reactive,watch, watchEffect} from 'vue'
  let firstname=ref('')
```

```
  let lastname=reactive({name:''})
  let fullname=ref('')
  //监听所有响应式数据
  watchEffect(()=>{
    //直接在内部使用监听属性即可！不用外部声明
    //也不需要，即时回调设置！默认初始化就加载！
    console.log(firstname.value)
    console.log(lastname.name)
    fullname.value=`${firstname.value}${lastname.name}`
  })
</script>
<template>
  <div>
    全名:{{fullname}} <br>
    姓氏:<input type="text" v-model="firstname"> <br>
    名字:<input type="text" v-model="lastname.name" > <br>
  </div>
</template>
<style scoped>
</style>
```

> `watch` vs. `watchEffect`：

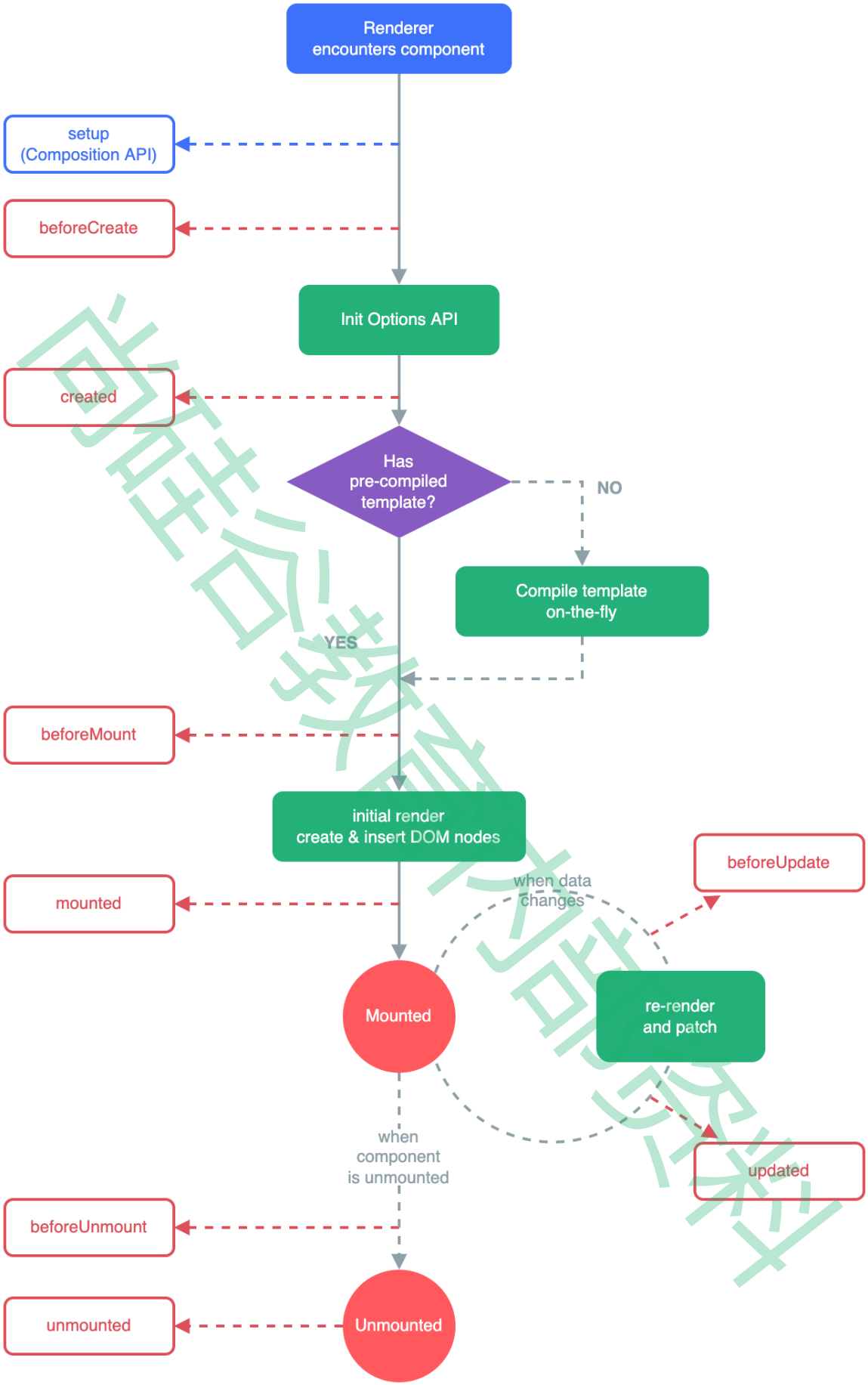
- `watch` 和 `watchEffect` 都能响应式地执行有副作用的回调。它们之间的主要区别是追踪响应式依赖的方式：
  - `watch` 只追踪明确侦听的数据源。它不会追踪任何在回调中访问到的东西。另外，仅在数据源确实改变时才会触发回调。`watch` 会避免在发生副作用时追踪依赖，因此，我们能更加精确地控制回调函数的触发时机；
  - `watchEffect`，则会在副作用发生期间追踪依赖。它会在同步执行过程中，自动追踪所有能访问到的响应式属性。这更方便，而且代码往往更简洁，但有时其响应性依赖关系会不那么明确；

6.7. Vue生命周期

6.7.1 生命周期简介

> 每个 Vue 组件实例在创建时都需要经历一系列的初始化步骤，比如设置好数据侦听，编译模板，挂载实例到 DOM，以及在数据改变时更新 DOM。在此过程中，它也会运行被称为 生命周期钩子的函数，让开发者有机会在特定阶段运行自己的代码!

- 周期图解：



- 常见钩子函数：
  - onMounted()　　注册一个回调函数，在组件挂载完成后执行；
  - onUpdated()　　注册一个回调函数，在组件因为响应式状态变更而更新其 DOM 树之后调用；

- onUnmounted()　注册一个回调函数，在组件实例被卸载之后调用；
- onBeforeMount()　注册一个钩子，在组件被挂载之前被调用；
- onBeforeUpdate()　注册一个钩子，在组件即将因为响应式状态变更而更新其 DOM 树之前调用；
- onBeforeUnmount() 注册一个钩子，在组件实例被卸载之前调用；

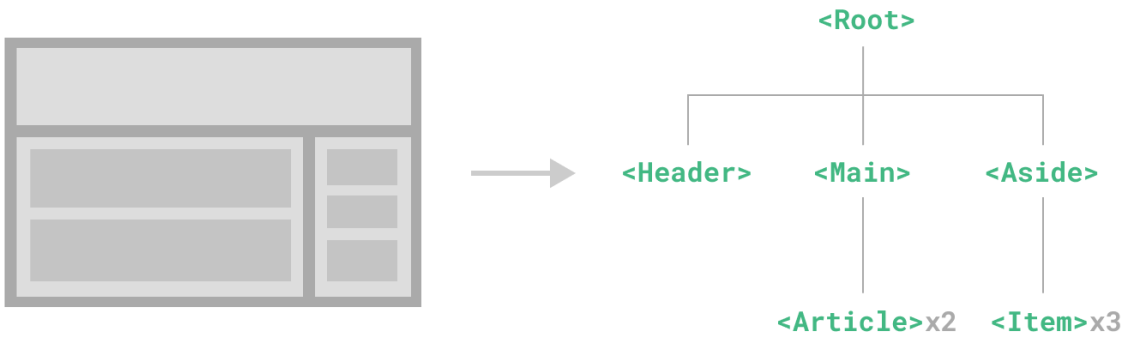
6.7.2 生命周期案例

```
<script setup>
    import {ref,onUpdated,onMounted,onBeforeUpdate} from 'vue'
    let message =ref('hello')
    // 挂载完毕生命周期
    onMounted(()=>{
      console.log('-----------onMounted---------')
      let span1 =document.getElementById("span1")
      console.log(span1.innerText)
    })
    // 更新前生命周期
    onBeforeUpdate(()=>{
      console.log('-----------onBeforeUpdate---------')
      console.log(message.value)
      let span1 =document.getElementById("span1")
      console.log(span1.innerText)
    })
    // 更新完成生命周期
    onUpdated(()=>{
      console.log('-----------onUpdated---------')
      let span1 =document.getElementById("span1")
      console.log(span1.innerText)
    })
</script>
<template>
  <div>
    <span id="span1" v-text="message"></span> <br>
    <input type="text" v-model="message">
  </div>
</template>
<style scoped>
</style>
```

6.8 Vue组件

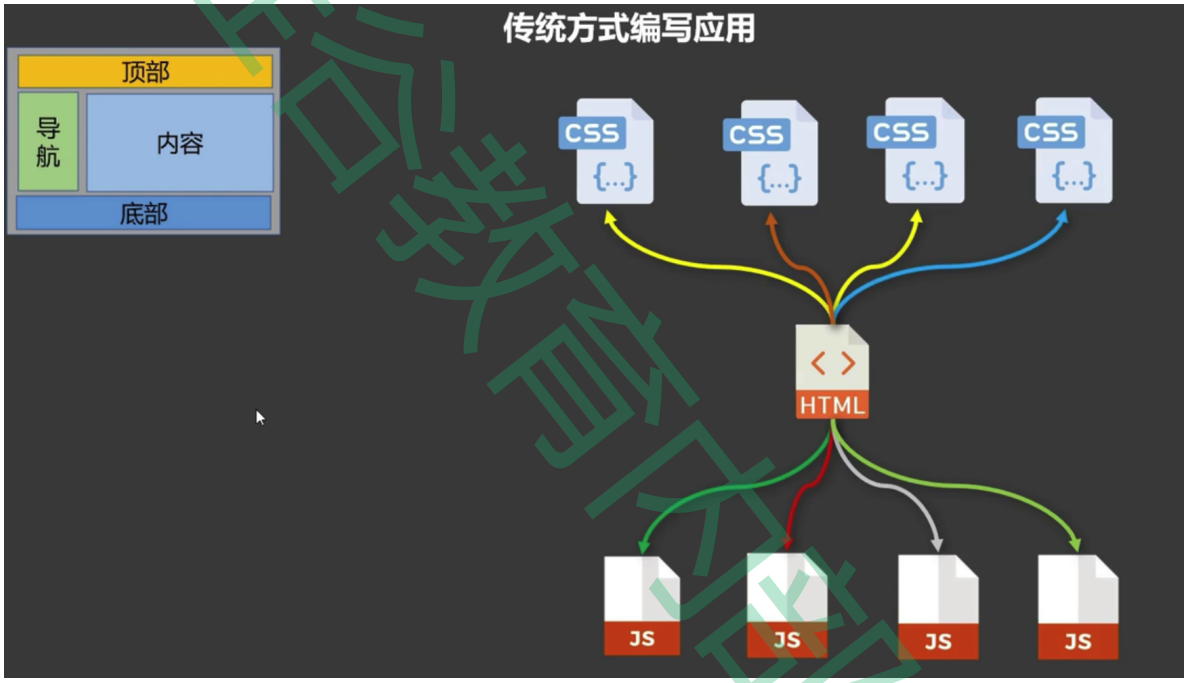
6.8.1 组件基础

> 组件允许我们将 UI 划分为独立的、可重用的部分，并且可以对每个部分进行单独的思考。组件就是实现应用中局部功能代码和资源的集合！在实际应用中，组件常常被组织成层层嵌套的树状结构：

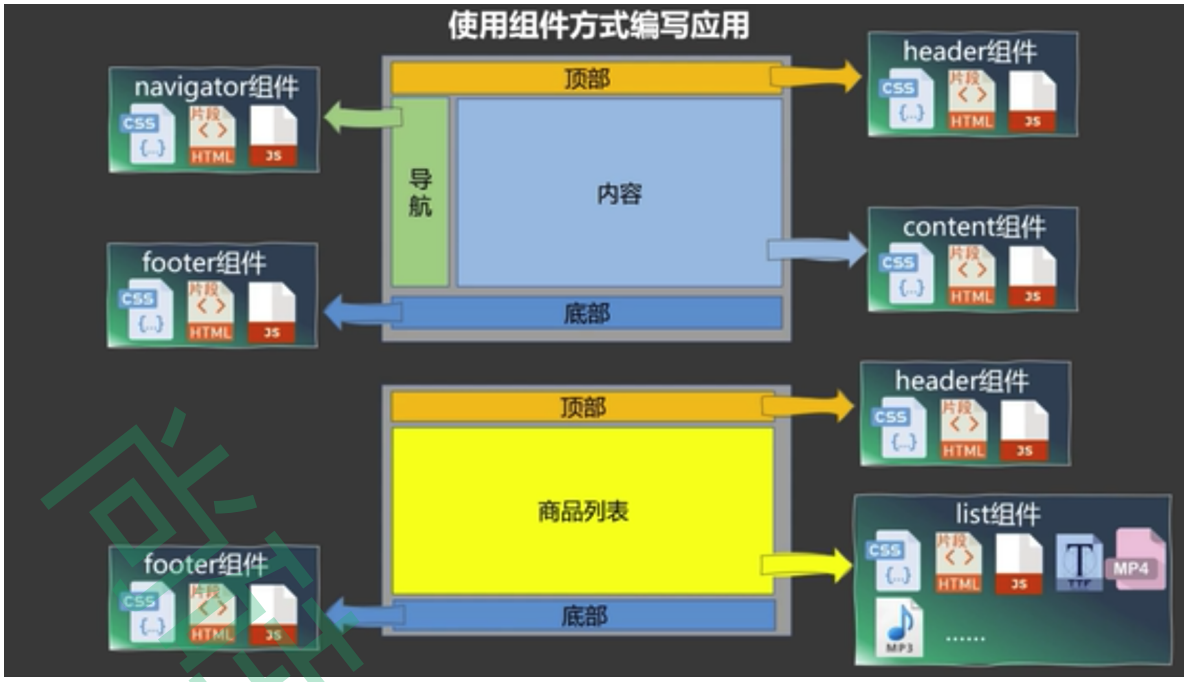


- 这和我们嵌套 HTML 元素的方式类似，Vue 实现了自己的组件模型，使我们可以在每个组件内封装自定义内容与逻辑。

> 传统方式编写应用：



> 组件方式编写应用：



- 组件化：对js/css/html统一封装，这是Vue中的概念；
- 模块化：对js的统一封装,这是ES6中的概念；
- 组件化中，对js部分代码的处理使用ES6中的模块化；

6.8.2 组件化入门案例

> 案例需求： 创建一个页面，包含头部和菜单以及内容显示区域，每个区域使用独立组建！

> 1 准备Vue项目：

```
npm create vite
cd vite项目
npm install
```

> 2 安装相关依赖：

```
npm install sass
npm install bootstrap
```

3 创建子组件 在src/components文件下 vscode需要安装Vetur插件，这样Vue文件有快捷提示：

- Header.vue

```
<script setup type="module">
</script>
<template>
    <div>
        欢迎： xx <a href="#">退出登录</a>
    </div>
</template>
<style>
</style>
```

- Navigator.vue

```
<script setup type="module">
</script>
<template>
    <!-- 推荐写一个根标签-->
    <div>
       <ul>
          <li>学员管理</li>
          <li>图书管理</li>
          <li>请假管理</li>
          <li>考试管理</li>
          <li>讲师管理</li>
       </ul>
    </div>
</template>
<style>
</style>
```

- Content.vue

```
<script setup type="module">
</script>
<template>
    <div>
        展示的主要内容!
    </div>
</template>
<style>
</style>
```

- App.vue 入口组件App引入组件：

```
<script setup>
    import Header  from './components/Header.vue'
    import Navigator  from './components/Navigator.vue'
    import Content  from './components/Content.vue'
</script>
<template>
  <div>
```

```
    <Header class="header"></Header>
    <Navigator class="navigator"></Navigator>
    <Content class="content"></Content>
  </div>
</template>
<style scoped>
    .header{
       height: 80px;
       border: 1px solid red;
    }
    .navigator{
      width: 15%;
      height: 800px;
      display: inline-block;
      border: 1px blue solid;
      float: left;
    }
    .content{
      width: 83%;
      height: 800px;
      display: inline-block;
      border: 1px goldenrod solid;
      float: right;
    }
</style>
```

> 4 启动测试

```
npm run dev
```

6.8.3 组件之间传递数据

6.8.3.1 父传子

> Vue3 中父组件向子组件传值可以通过 props 进行，具体操作如下：

1. 首先，在父组件中定义需要传递给子组件的值，接着，在父组件的模板中引入子组件，同时在引入子组件的标签中添加 props 属性并为其设置需要传递的值。
2. 在 Vue3 中，父组件通过 props 传递给子组件的值是响应式的。也就是说，如果在父组件中的传递的值发生了改变，子组件中的值也会相应地更新。

- 父组件代码：App.vue

```
<script setup>
  import Son from './components/Son.vue'
  import {ref,reactive,toRefs} from 'vue'
  let message = ref('parent data!')
  let title = ref(42)
  function changeMessage(){
    message.value = '修改数据！'
    title.value++
  }
</script>
<template>
  <div>
    <h2>{{ message }}</h2>
```

```
    <hr>
    <!-- 使用子组件，并且传递数据！ -->
    <Son :message="message" :title="title"></Son>
    <hr>
    <button @click="changeMessage">点击更新</button>
  </div>
</template>
<style scoped>
</style>
```

- 子组件代码：Son.vue

```
<script setup type="module">
    import {ref,isRef,defineProps} from 'vue'
    //声明父组件传递属性值
    defineProps({
        message:String ,
        title:Number
    })
</script>
<template>
    <div>
    <div>{{ message }}</div>
    <div>{{ title }}</div>
    </div>
</template>
<style>
</style>
```

6.8.3.2 子传父

- 父组件： App.vue

```
<script setup>
    import Son from './components/Son.vue'
    import {ref} from 'vue'
    let pdata = ref('')
    const padd = (data) => {
        console.log('2222');
        pdata.value =data;
    }
    //自定义接收，子组件传递数据方法！ 参数为数据！
    const psub = (data) => {
        console.log('11111');
        pdata.value = data;
    }
</script>
<template>
    <div>
        <!-- 声明@事件名应该等于子模块对应事件名！调用方法可以是当前自定义！-->
        <Son @add="padd" @sub="psub"></Son>
        <hr>
        {{ pdata }}
    </div>
</template>
<style>
</style>
```

- 子组件：Son.vue

```
<script setup>
    import {ref,defineEmits} from 'vue'
    //1.定义要发送给父组件的方法，可以1或者多个
    let emites = defineEmits(['add','sub']);
    let data = ref(1);
    function sendMsgToParent(){
        //2.出发父组件对应的方法，调用defineEmites对应的属性
        emites('add','add data!'+data.value)
        emites('sub','sub data!'+data.value)
        data.value ++;
    }
</script>
<template>
    <div>
      <button @click="sendMsgToParent">发送消息给父组件</button>
    </div>
</template>
```

6.8.3.3 兄弟传参

- Navigator.vue: 发送数据到App.vue

```
<script setup type="module">
    import {defineEmits} from 'vue'
    const emits = defineEmits(['sendMenu']);
    //触发事件，向父容器发送数据
    function send(data){
        emits('sendMenu',data);
    }
</script>
<template>
    <!-- 推荐写一个根标签-->
    <div>
       <ul>
          <li @click="send('学员管理')">学员管理</li>
```

```
            <li @click="send('图书管理')">图书管理</li>
            <li @click="send('请假管理')">请假管理</li>
            <li @click="send('考试管理')">考试管理</li>
            <li @click="send('讲师管理')">讲师管理</li>
        </ul>
    </div>
</template>
<style>
</style>
```

- App.vue: 发送数据到Content.vue

```
<script setup>
  import Header  from './components/Header.vue'
  import Navigator  from './components/Navigator.vue'
  import Content  from './components/Content.vue'
  import {ref} from "vue"
  //定义接受navigator传递参数
  var navigator_menu = ref('ceshi');
  const receiver = (data) =>{
    navigator_menu.value = data;
  }
</script>
<template>
  <div>
      <hr>
      {{ navigator_menu }}
      <hr>
     <Header class="header"></Header>
     <Navigator @sendMenu="receiver" class="navigator"></Navigator>
     <!-- 向子组件传递数据-->
     <Content class="content" :message="navigator_menu"></Content>
    </div>
</template>
<style scoped>
    .header{
       height: 80px;
       border: 1px solid red;
    }
    .navigator{
      width: 15%;
      height: 800px;
      display: inline-block;
      border: 1px blue solid;
      float: left;
    }
    .content{
      width: 83%;
      height: 800px;
      display: inline-block;
      border: 1px goldenrod solid;
      float: right;
    }
</style>
```

- Content.vue

```
<script setup type="module">
```

```
    defineProps({
        message:String
    })
</script>
<template>
    <div>
        展示的主要内容!
        <hr>
        {{ message }}
    </div>
</template>
<style>
</style>
```

七、Vue3路由机制Router

7.1 路由简介

> 1 什么是路由?

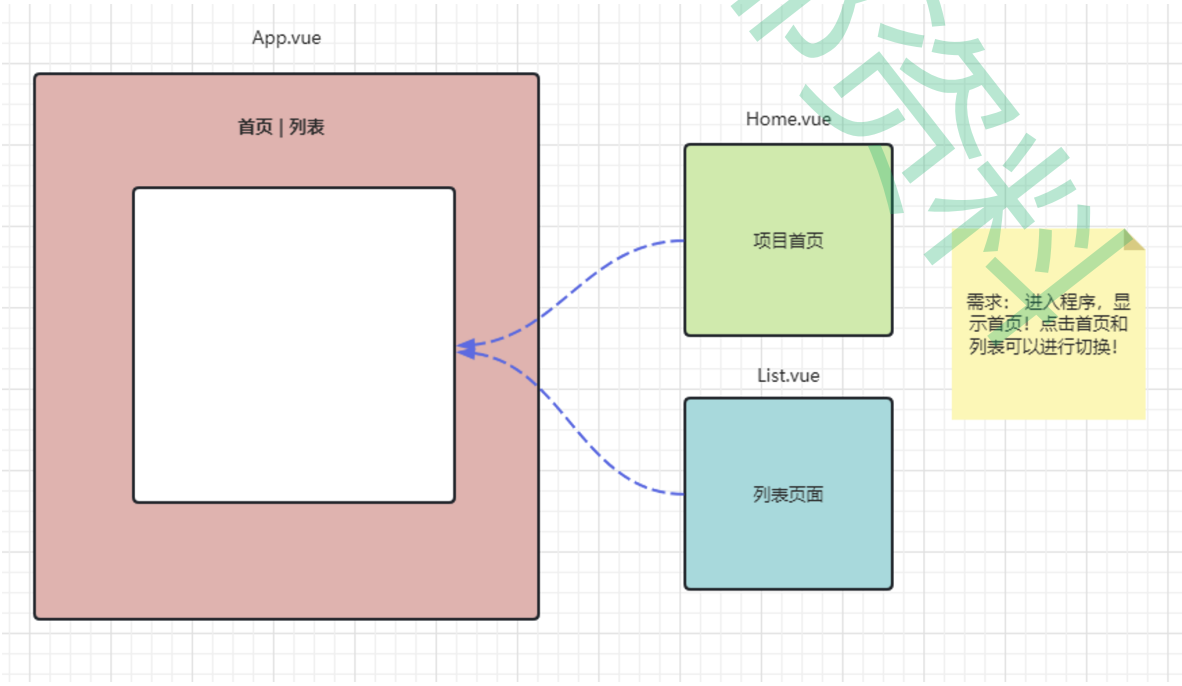
- 定义：路由就是根据不同的 URL 地址展示不同的内容或页面；
- 通俗理解：路由就像是一个地图，我们要去不同的地方，需要通过不同的路线进行导航；

> 2 路由的作用：

- 单页应用程序（SPA）中，路由可以实现不同视图之间的无刷新切换，提升用户体验；
- 路由还可以实现页面的认证和权限控制，保护用户的隐私和安全；
- 路由还可以利用浏览器的前进与后退，帮助用户更好地回到之前访问过的页面；

7.2 路由入门案例

> 1 案例需求分析：



2 创建项目和导入路由依赖：

```
npm create vite //创建项目cd 项目文件夹 //进入项目文件夹
npm install //安装项目需求依赖
npm install vue-router@4 --save //安装全局的vue-router 4版本
```

3 准备页面和组件 ：

- components/Home.vue

```
<script setup>
</script>
<template>
    <div>
        <h1>Home页面</h1>
    </div>
</template>
<style scoped>
</style>
```

- components/List.vue

```
<script setup>
</script>
<template>
    <div>
        <h1>List页面</h1>
    </div>
</template>
<style scoped>
</style>
```

- components/Add.vue

```
<script setup>
</script>
<template>
    <div>
        <h1>Add页面</h1>
    </div>
</template>
<style scoped>
</style>
```

- components/Update.vue

```
<script setup>
</script>
<template>
    <div>
        <h1>Update页面</h1>
    </div>
</template>
<style scoped>
</style>
```

- App.vue

```
<script setup>
</script>
<template>
    <div>
      <h1>App页面</h1>
      <hr/>
        <!-- 路由的连接 -->
        <router-link to="/">home页</router-link> <br>
        <router-link to="/list">list页</router-link> <br>
        <router-link to="/add">add页</router-link> <br>
        <router-link to="/update">update页</router-link> <br>
      <hr/>
      <!-- 路由连接对应视图的展示位置 -->
      <hr> 默认展示位置:<router-view></router-view>
      <hr> Home视图展示:<router-view name="homeView"></router-view>
      <hr> List视图展示:<router-view name="listView"></router-view>
      <hr> Add视图展示:<router-view name="addView"></router-view>
      <hr> Update视图展示:<router-view name="updateView"></router-view>
    </div>
</template>
<style scoped>
</style>
```

4 准备路由配置:

- src/routers/router.js

```
// 导入路由创建的相关方法
import {createRouter,createWebHashHistory} from 'vue-router'
// 导入vue组件
import Home from '../components/Home.vue'
import List from '../components/List.vue'
import Add from '../components/Add.vue'
import Update from '../components/Update.vue'
// 创建路由对象,声明路由规则
const router = createRouter({
    //createWebHashHistory() 是 Vue.js 基于 hash 模式创建路由的工厂函数。在使用这种模式下，路由信息保存在 URL 的 hash 中，
    //使用 createWebHashHistory() 方法，可以创建一个路由历史记录对象，用于管理应用程序的路由。在 Vue.js 应用中，
    //通常使用该方法来创建路由的历史记录对象。
    //就是路由中缓存历史记录的对象，vue-router提供
    history: createWebHashHistory(),
    routes:[
        {
            path:'/',
            /*
                component指定组件在默认的路由视图位置展示
                components:Home
                components指定组件在name为某个值的路由视图位置展示
                components:{
                    default:Home,// 默认路由视图位置
                    homeView:Home// name为homeView的路由视图位置
                }
            */
            components:{
                default:Home,
```

```
                homeView:Home
            }
        },
        {
            path:'/list',
            components:{
                listView : List
            }
        },
        {
            path:'/add',
            components:{
                addView:Add
            }
        },
        {
            path:'/update',
            components:{
                updateView:Update
            }
        },
    ]

})
// 对外暴露路由对象
export default router;
```

5 main.js引入Router配置：

- 修改文件：main.js (入口文件)

```
import { createApp } from 'vue'
import './style.css'
import App from './App.vue'
//导入router模块
import router from './routers/router.js'
let app = createApp(App)
//绑定路由对象
app.use(router)
//挂载视图
app.mount("#app")
```

6 启动测试：

```
npm run dev
```

7.3 路由重定向

重定向的作用：将一个路由重定向到另一个路由上。

- 修改案例：访问/list和/showAll都定向到List.vue.
- router.js

```
{
   path:'/showAll',
   // 重定向
   redirect :'/list'
}
```

- App.vue

```
<script setup>
</script>
<template>
    <div>
      <h1>App页面</h1>
      <hr/>
        <!-- 路由的连接 -->
        <router-link to="/">home页</router-link> <br>
        <router-link to="/list">list页</router-link> <br>
        <router-link to="/showAll">showAll页</router-link> <br>
        <router-link to="/add">add页</router-link> <br>
        <router-link to="/update">update页</router-link> <br>
      <hr/>
      <!-- 路由连接对应视图的展示位置 -->
      <hr> 默认展示位置:<router-view></router-view>
      <hr> Home视图展示:<router-view name="homeView"></router-view>
      <hr> List视图展示:<router-view name="listView"></router-view>
      <hr> Add视图展示:<router-view name="addView"></router-view>
      <hr> Update视图展示:<router-view name="updateView"></router-view>
    </div>
</template>
<style scoped>
</style>
```

7.4 编程式路由(useRouter)

> 声明式路由：

- `<router-link to="/list">list页</router-link>` 这种路由，点击后只能切换/list对应组件，是固定的。

> 编程式路由：

- 通过useRouter，动态决定向那个组件切换的路由；
- 在 Vue 3 和 Vue Router 4 中，你可以使用 `useRouter` 来实现动态路由(编程式路由)；
- 这里的 `useRouter` 方法返回的是一个 router 对象，你可以用它来做如导航到新页面、返回上一页面等操作；

> 案例需求：通过普通按钮配合事件绑定实现路由页面跳转，不直接使用router-link标签。

- App.vue

```
<script setup type="module">
  import {useRouter} from 'vue-router'
  import {ref} from 'vue'
  //创建动态路由对象
  let router = useRouter()
```

```
  let  routePath =ref('')
  let  showList= ()=>{
      // 编程式路由
      // 直接push一个路径
      // router.push('/list')
      // push一个带有path属性的对象
      router.push({path:'/list'})
  }
</script>
<template>
    <div>
      <h1>App页面</h1>
      <hr/>
       <!-- 路由的连接 -->
        <router-link to="/">home页</router-link> <br>
        <router-link to="/list">list页</router-link> <br>
        <!-- 动态输入路径,点击按钮,触发单击事件的函数,在函数中通过编程是路由切换页面 -->
         <button @click="showList()">showList</button> <br>
      <hr/>
      <!-- 路由连接对应视图的展示位置 -->
      <hr> 默认展示位置:<router-view></router-view>
      <hr> Home视图展示:<router-view name="homeView"></router-view>
      <hr> List视图展示:<router-view name="listView"></router-view>
    </div>
</template>
<style scoped>
</style>
```

7.5 路由传参(useRoute)

> 路径参数:

- 在路径中使用一个动态字段来实现，我们称之为 **路径参数**:
  - 例如： 查看数据详情 `/showDetail/1` ， `1` 就是要查看详情的id，可以动态添值。

> 键值对参数:

- 类似与get请求通过url传参，数据是键值对形式的:
  - 例如: 查看数据详情 `/showDetail?hid=1` ， `hid=1` 就是要传递的键值对参数。

> 读取参数:

- 在 Vue 3 和 Vue Router 4 中，你可以使用 `useRoute` 这个函数从 Vue 的组合式 API 中获取路由对象;
- `useRoute` 方法返回的是当前的 route 对象，你可以用它来获取关于当前路由的信息，如当前的路径、键值对参数等;

> 案例需求 : 切换到ShowDetail.vue组件时,向该组件通过路由传递参数。

- 修改App.vue文件

```
<script setup type="module">
  import {useRouter} from 'vue-router'
```

```
  //创建动态路由对象
  let router = useRouter()
  //动态路由路径传参方法
  let showDetail= (id,language)=>{
      // 尝试使用拼接字符串方式传递路径参数
      //router.push(`showDetail/${id}/${languange}`)
      /*路径参数,需要使用params  */
      router.push({name:"showDetail",params:{id:id,language:language}})
  }
  let showDetail2= (id,language)=>{
      /*uri键值对参数,需要使用query */
      router.push({path:"/showDetail2",query:{id:id,language:language}})
  }
</script>
<template>
    <div>
      <h1>App页面</h1>
      <hr/>
      <!-- 路径参数  -->
      <router-link to="/showDetail/1/JAVA">sh路径传参JAVA</router-link>
      <button @click="showDetail(1,'JAVA')">编程式路由路径传参显示JAVA</button>
      <hr/>
      <!-- 键值对参数 -->
      <router-link v-bind:to="{path:'/showDetail2',query:{id:1,language:'Java'}}">键值
对传参JAVA</router-link>
      <button @click="showDetail2(1,'JAVA')">编程式路由键值对传参JAVA</button>
      <hr> showDetail视图展示:<router-view name="showDetailView"></router-view>
      <hr> showDetail2视图展示:<router-view name="showDetailView2"></router-view>
    </div>
</template>
<style scoped>
</style>
```

- 修改router.js增加路径参数占位符

```
import {createRouter,createWebHashHistory} from 'vue-router'
import ShowDetail from '../components/ShowDetail.vue'
import ShowDetail2 from '../components/ShowDetail2.vue'
// 创建路由对象,声明路由规则
const router = createRouter({
    history: createWebHashHistory(),
    routes:[
        {
            /* 此处:id  :language作为路径的占位符 */
            path:'/showDetail/:id/:language',
            /* 动态路由传参时,根据该名字找到该路由 */
            name:'showDetail',
            components:{
                showDetailView:ShowDetail
            }
        },
        {
            path:'/showDetail2',
            components:{
                showDetailView2:ShowDetail2
            }
        },
    ]
```

```
})
// 对外暴露路由对象
export default router;
```

- ShowDetail.vue 通过useRoute获取路径参数

```
<script setup type="module">
    import{useRoute} from 'vue-router'
    import { onUpdated,ref } from 'vue';
    // 获取当前的route对象
    let route =useRoute()
    let languageId = ref(0)
    let languageName = ref('')
    //  借助更新时生命周期,将数据更新进入响应式对象
    onUpdated (()=>{
        // 获取对象中的参数
        languageId.value=route.params.id
        languageName.value=route.params.language
        console.log(languageId.value)
        console.log(languageName.value)
    })
</script>
<template>
    <div>
        <h1>ShowDetail页面</h1>
        <h3>编号{{route.params.id}}:{{route.params.language}}是世界上最好的语言</h3>
        <h3>编号{{languageId}}:{{languageName}}是世界上最好的语言</h3>
    </div>
</template>
<style scoped>
</style>
```

- ShowDetail2.vue通过useRoute获取键值对参数

```
<script setup type="module">
    import{useRoute} from 'vue-router'
    import { onUpdated,ref } from 'vue';
    // 获取当前的route对象
    let route =useRoute()
    let languageId = ref(0)
    let languageName = ref('')
    //  借助更新时生命周期,将数据更新进入响应式对象
    onUpdated (()=>{
        // 获取对象中的参数(通过query获取参数,此时参数是key-value形式的)
        console.log(route.query)
        console.log(languageId.value)
        console.log(languageName.value)
        languageId.value=route.query.id
        languageName.value=route.query.language
    })
</script>
<template>
    <div>
        <h1>ShowDetail2页面</h1>
        <h3>编号{{route.query.id}}:{{route.query.language}}是世界上最好的语言</h3>
        <h3>编号{{languageId}}:{{languageName}}是世界上最好的语言</h3>
    </div>
</template>
```

```
<style scoped>
</style>
```

7.6 路由守卫

> 在 Vue 3 中，路由守卫是用于在路由切换期间进行一些特定任务的回调函数。路由守卫可以用于许多任务，例如验证用户是否已登录、在路由切换前提供确认提示、请求数据等。Vue 3 为路由守卫提供了全面的支持，并提供了以下几种类型的路由守卫：

1. **全局前置守卫**：在路由切换前被调用，可以用于验证用户是否已登录、中断导航、请求数据等；
2. **全局后置守卫**：在路由切换之后被调用，可以用于处理数据、操作 DOM 、记录日志等；
3. **守卫代码的位置**: 在router.js中

```
//全局前置路由守卫
router.beforeEach( (to,from,next) => {
    //to 是目标地包装对象 .path属性可以获取地址
    //from 是来源地包装对象 .path属性可以获取地址
    //next是方法，不调用默认拦截！ next() 放行,直接到达目标组件
    //next('/地址')可以转发到其他地址,到达目标组件前会再次经过前置路由守卫
    console.log(to.path,from.path,next)
    //需要判断，注意避免无限重定向
    if(to.path == '/index'){
        next()
    }else{
        next('/index')
    }
} )
//全局后置路由守卫
router.afterEach((to, from) => {
    console.log(`Navigate from ${from.path} to ${to.path}`);
});
```

> 登录案例，登录以后才可以进入home，否则必须进入login。

- 定义Login.vue

```
<script setup>
    import {ref} from 'vue'
    import {useRouter} from 'vue-router'
    let username =ref('')
    let password =ref('')
    let router = useRouter();
    let login = () =>{
        console.log(username.value,password.value)
        if(username.value == 'root' & password.value == '123456'){
            router.push({path:'/home',query:{'username':username.value}})
            //登录成功利用前端存储机制，存储账号!
            localStorage.setItem('username',username.value)
            //sessionStorage.setItem('username',username)
        }else{
            alert('登录失败，账号或者密码错误！');
        }
    }

</script>
```

```
<template>
    <div>
        账号： <input type="text" v-model="username" placeholder="请输入账号！"><br>
        密码： <input type="password" v-model="password" placeholder="请输入密码！"><br>
        <button @click="login()">登录</button>
    </div>
</template>
<style scoped>
</style>
```

- 定义Home.vue

```
<script setup>
 import {ref} from 'vue'
 import {useRoute,useRouter} from 'vue-router'
 let route =useRoute()
 let router = useRouter()
 //  并不是每次进入home页时,都有用户名参数传入
 //let username = route.query.username
 let username =window.localStorage.getItem('username');
 let logout= ()=>{
    // 清除localStorge中的username
    //window.sessionStorage.removeItem('username')
    window.localStorage.removeItem('username')
    // 动态路由到登录页
    router.push("/login")
 }
</script>
<template>
    <div>
        <h1>Home页面</h1>
        <h3>欢迎{{username}}登录</h3>
        <button @click="logout">退出登录</button>
    </div>
</template>
<style scoped>
</style>
```

- App.vue

```
<script setup type="module">
</script>
<template>
    <div>
      <router-view></router-view>
    </div>
</template>
<style scoped>
</style>
```

- 定义routers.js

```
import {createRouter,createWebHashHistory} from 'vue-router'
import Home from '../components/Home.vue'
import Login from '../components/login.vue'
const router = createRouter({
    history: createWebHashHistory(),
    routes:[
```

```
        {
            path:'/home',
            component:Home
        },
        {
            path:'/',
            redirect:"/home"
        },
        {
            path:'/login',
            component:Login
        },
    ]
})
// 设置路由的全局前置守卫
router.beforeEach((to,from,next)=>{
    /*
    to 要去那
    from 从哪里来
    next 放行路由时需要调用的方法,不调用则不放行
    */
    console.log(`从哪里来:${from.path},到哪里去:${to.path}`)
    if(to.path == '/login'){
        //放行路由  注意放行不要形成循环
        next()
    }else{
        //let username =window.sessionStorage.getItem('username');
        let username =window.localStorage.getItem('username');
        if(null != username){
            next()
        }else{
            next('/login')
        }
    }
})
// 设置路由的全局后置守卫
router.afterEach((to,from)=>{
    console.log(`从哪里来:${from.path},到哪里去:${to.path}`)
})
// 对外暴露路由对象
export default router;
```

- 启动测试

```
npm run dev
```

八、 案例开发-日程管理-第五期

8.1 重构前端工程

> 业务目标展示:

- 登录页

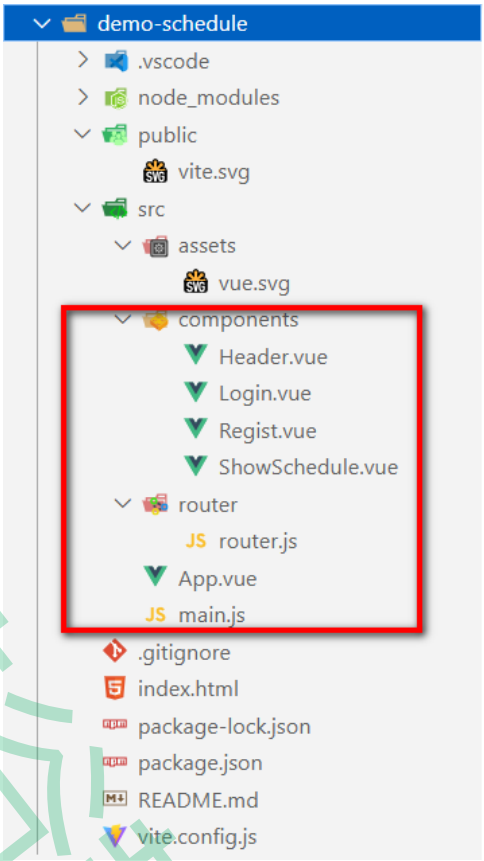
- 注册页

- 日程管理页

> 创建项目,安装依赖

```
npm create vite
cd 项目目录
npm  install
npm  install vue-router
```

- 项目结构如下



开发视图：

- Header.vue视图

```
<script setup>
</script>
<template>
  <div>
    <h1 class="ht">欢迎使用日程管理系统</h1>
    <div>
      <div  class="optionDiv">
        <router-link to="/login">
          <button class="b1s">登录</button>
        </router-link>
        <router-link to="/regist">
          <button class="b1s">注册</button>
        </router-link>
      </div>
      <div   class="optionDiv">
        欢迎xxx
        <button class="b1b">退出登录</button>
        <router-link to="/showSchedule">
          <button class="b1b">查看我的日程</button>
        </router-link>
      </div>
      <br>
    </div>
  </div>
</template>
<style scoped>
  .ht{
      text-align: center;
      color: cadetblue;
```

```
        font-family: 幼圆;
    }
    .b1s{
          border: 2px solid powderblue;
          border-radius: 4px;
          width:60px;
          background-color: antiquewhite;

      }
      .b1b{
          border: 2px solid powderblue;
          border-radius: 4px;
          width:100px;
          background-color: antiquewhite;
      }
      .optionDiv{
        width: 300px;
        float: right;
      }
</style>
```

- Login.vue视图

```
<script setup>
       import{ ref,reactive} from 'vue'
       // 响应式数据,保存用户输入的表单信息
       let loginUser =reactive({
        username:'',
        userPwd:''
       })
       // 响应式数据,保存校验的提示信息
       let usernameMsg =ref('')
       let userPwdMsg = ref('')
        // 校验用户名的方法
        function checkUsername(){
            // 定义正则
            var usernameReg=/^[a-zA-Z0-9]{5,10}$/
            // 校验用户名
            if(!usernameReg.test(loginUser.username)){
                // 格式不合法
                usernameMsg.value="格式有误"
                return false
            }
            usernameMsg.value="ok"
            return true
        }
        // 校验密码的方法
        function checkUserPwd(){
            // 定义正则
            var passwordReg=/^[0-9]{6}$/
             // 校验密码
             if(!passwordReg.test(loginUser.userPwd)){
                // 格式不合法
                userPwdMsg.value="格式有误"
                return false
            }
            userPwdMsg.value="ok"
            return true
```

```
        }
</script>
<template>
  <div>
    <h3 class="ht">请登录</h3>
        <table class="tab" cellspacing="0px">
            <tr class="ltr">
                <td>请输入账号</td>
                <td>
                    <input class="ipt"
                    type="text"
                    v-model="loginUser.username"
                    @blur="checkUsername()">
                    <span id="usernameMsg" v-text="usernameMsg"></span>
                </td>
            </tr>
            <tr class="ltr">
                <td>请输入密码</td>
                <td>
                    <input class="ipt"
                    type="password"
                    v-model="loginUser.userPwd"
                    @blur="checkUserPwd()">
                    <span id="userPwdMsg" v-text="userPwdMsg"></span>
                </td>
            </tr>
            <tr class="ltr">
                <td colspan="2" class="buttonContainer">
                    <input class="btn1" type="button" value="登录">
                    <input class="btn1" type="button" value="重置">
                    <router-link to="/regist">
                      <button class="btn1">去注册</button>
                    </router-link>
                </td>
            </tr>
        </table>
  </div>
</template>
<style scoped>
        .ht{
            text-align: center;
            color: cadetblue;
            font-family: 幼圆;
        }
        .tab{
            width: 500px;
            border: 5px solid cadetblue;
            margin: 0px auto;
            border-radius: 5px;
            font-family: 幼圆;
        }
        .ltr td{
            border: 1px solid  powderblue;
        }
        .ipt{
            border: 0px;
            width: 50%;
        }
```

```
        .btn1{
            border: 2px solid powderblue;
            border-radius: 4px;
            width:60px;
            background-color: antiquewhite;
        }
        #usernameMsg , #userPwdMsg {
            color: gold;
        }
        .buttonContainer{
            text-align: center;
        }
</style>
```

- Regist.vue视图

```
<script setup>
      import{ ref,reactive} from 'vue'
       // 响应式数据,保存用户输入的表单信息
       let registUser =reactive({
        username:'',
        userPwd:''
       })
       // 响应式数据,保存校验的提示信息
       let reUserPwd =ref('')
       let reUserPwdMsg =ref('')
       let usernameMsg =ref('')
       let userPwdMsg = ref('')

          // 校验用户名的方法
        function checkUsername(){
            // 定义正则
            let usernameReg=/^[a-zA-Z0-9]{5,10}$/
            // 校验
            if(!usernameReg.test(registUser.username)){
              // 提示
              usernameMsg.value = "不合法"
              return false
            }
            // 通过校验
            usernameMsg.value="OK"
            return true
        }
        // 校验密码的方法
        function checkUserPwd(){
            // 定义正则
            let passwordReg=/^[0-9]{6}$/
             // 校验
             if(!passwordReg.test(registUser.userPwd)){
              // 提示
              userPwdMsg.value = "不合法"
              return false
            }
            // 通过校验
            userPwdMsg.value="OK"
            return true
        }
        // 校验密码的方法
```

```
        function checkReUserPwd(){
            // 定义正则
            let passwordReg=/^[0-9]{6}$/
            // 校验
            if(!passwordReg.test(reUserPwd.value)){
              // 提示
              reUserPwdMsg.value = "不合法"
              return false
            }
            console.log(registUser.userPwd,reUserPwd.value)
            // 校验
            if(!(registUser.userPwd==reUserPwd.value)){
              // 提示
              reUserPwdMsg.value = "不一致"
              return false
            }
            // 通过校验
            reUserPwdMsg.value="OK"
            return true
        }
</script>
<template>
  <div>
    <h3 class="ht">请注册</h3>
    <table class="tab" cellspacing="0px">
        <tr class="ltr">
            <td>请输入账号</td>
            <td>
                <input class="ipt"
                  id="usernameInput"
                  type="text"
                  name="username"
                  v-model="registUser.username"
                  @blur="checkUsername()">

                <span id="usernameMsg" class="msg" v-text="usernameMsg"></span>
            </td>
        </tr>
        <tr class="ltr">
            <td>请输入密码</td>
            <td>
                <input class="ipt"
                id="userPwdInput"
                type="password"
                name="userPwd"
                v-model="registUser.userPwd"
                @blur="checkUserPwd()">
                <span id="userPwdMsg" class="msg" v-text="userPwdMsg"></span>
            </td>
        </tr>
        <tr class="ltr">
            <td>确认密码</td>
            <td>
                <input class="ipt"
                id="reUserPwdInput"
                type="password"
                v-model="reUserPwd"
                @blur="checkReUserPwd()">
```

```
                <span id="reUserPwdMsg" class="msg" v-text="reUserPwdMsg"></span>
            </td>
        </tr>
        <tr class="ltr">
            <td colspan="2" class="buttonContainer">
                <input class="btn1" type="button" value="注册">
                <input class="btn1" type="button" value="重置">
                <router-link to="/login">
                  <button class="btn1">去登录</button>
                </router-link>
            </td>
        </tr>
    </table>
  </div>
</template>
<style scoped>
        .ht{
            text-align: center;
            color: cadetblue;
            font-family: 幼圆;
        }
        .tab{
            width: 500px;
            border: 5px solid cadetblue;
            margin: 0px auto;
            border-radius: 5px;
            font-family: 幼圆;
        }
        .ltr td{
            border: 1px solid  powderblue;

        }
        .ipt{
            border: 0px;
            width: 50%;

        }
        .btn1{
            border: 2px solid powderblue;
            border-radius: 4px;
            width:60px;
            background-color: antiquewhite;
        }
        .msg {
            color: gold;
        }
        .buttonContainer{
            text-align: center;
        }
</style>
```

- ShowSchedule.vue视图

```
<script setup>
</script>
<template>
  <div>
    <h3 class="ht">您的日程如下</h3>
```

```
<table class="tab" cellspacing="0px">
    <tr class="ltr">
        <th>编号</th>
        <th>内容</th>
        <th>进度</th>
        <th>操作</th>
    </tr>
    <tr class="ltr">
        <td></td>
        <td></td>
        <td></td>
        <td class="buttonContainer">
            <button class="btn1">删除</button>
            <button class="btn1">保存修改</button>
        </td>
    </tr>
    <tr class="ltr buttonContainer" >
        <td colspan="4">
            <button class="btn1">新增日程</button>
        </td>
    </tr>
</table>
  </div>
</template>
<style scoped>
      .ht{
            text-align: center;
            color: cadetblue;
            font-family: 幼圆;
        }
        .tab{
            width: 80%;
            border: 5px solid cadetblue;
            margin: 0px auto;
            border-radius: 5px;
            font-family: 幼圆;
        }
        .ltr td{
            border: 1px solid  powderblue;
        }
        .ipt{
            border: 0px;
            width: 50%;
        }
        .btn1{
            border: 2px solid powderblue;
            border-radius: 4px;
            width:100px;
            background-color: antiquewhite;
        }
        #usernameMsg , #userPwdMsg {
            color: gold;
        }
        .buttonContainer{
            text-align: center;
        }
</style>
```

- App.vue视图

```
<script setup>
import Header from './components/Header.vue'
</script>
<template>
  <div>
    <Header></Header>
    <hr>
    <router-view></router-view>
  </div>
</template>
<style scoped>
</style>
```

- 配置路由

```
import {createRouter,createWebHashHistory} from 'vue-router'
import Login from '../components/Login.vue'
import Regist from '../components/Regist.vue'
import ShowScedule from '../components/ShowSchedule.vue'
let  router = createRouter({
    history:createWebHashHistory(),
    routes:[
        {
            path:"/",
            component:Login
        },
        {
            path:"/login",
            component:Login
        },
        {
            path:"/showSchedule",
            component:ShowScedule
        },
        {
            path:"/regist",
            component:Regist
        }
    ]
})
export default router
```

- 配置main.js

```
import { createApp } from 'vue'
import App from './App.vue'
// 导入路由
import router from './router/router.js'
let app =createApp(App)
// 全局使用路由
app.use(router)
app.mount('#app')
```

九、Vue3数据交互Axios

9.0 预讲知识-Promise

9.0.1 普通函数和回调函数

> 普通函数：正常调用的函数，一般函数执行完毕后才会继续执行下一行代码。

```
<script>
    let fun1 = () =>{
        console.log("fun1 invoked")
    }
    // 调用函数
    fun1()
    // 函数执行完毕,继续执行后续代码
    console.log("other code processon")
</script>
```

> 回调函数：一些特殊的函数，表示未来才会执行的一些功能，后续代码不会等待该函数执行完毕就开始执行了。

```
<script>
    // 设置一个2000毫秒后会执行一次的定时任务
    setTimeout(function (){
        console.log("setTimeout invoked")
    },2000)
    console.log("other code processon")
</script>
```

9.0.2 Promise 简介

> 前端中的异步编程技术，类似Java中的多线程+线程结果回调!

- Promise 是异步编程的一种解决方案，比传统的解决方案回调函数和事件更合理和更强大。它由社区最早提出和实现，ES6将其写进了语言标准，统一了用法，原生提供了 `Promise` 对象；
- 所谓 `Promise` ，简单说就是一个容器，里面保存着某个未来才会结束的事件（通常是一个异步操作）的结果。从语法上说，Promise 是一个对象，从它可以获取异步操作的消息。Promise 提供统一的 API，各种异步操作都可以用同样的方法进行处理；

> `Promise` 对象有以下两个特点：

1. Promise对象代表一个异步操作，有三种状态： `Pending` （进行中）、 `Resolved` （已完成，又称 Fulfilled）和 `Rejected` （已失败）。只有异步操作的结果，可以决定当前是哪一种状态，任何其他操作都无法改变这个状态。这也是 `Promise` 这个名字的由来，它的英语意思就是“承诺”，表示其他手段无法改变；
2. 一旦状态改变，就不会再变，任何时候都可以得到这个结果。Promise对象的状态改变，只有两种可能：从 `Pending` 变为 `Resolved` 和从 `Pending` 变为 `Rejected` 。只要这两种情况发生，状态就凝固了，不会再变了，会一直保持这个结果；

9.0.3 Promise 基本用法

> ES6规定，Promise对象是一个构造函数，用来生成Promise实例。

```
    <script>
```

```
        /*
        1.实例化promise对象,并且执行(类似Java创建线程对象,并且start)
        参数: resolve,reject随意命名,但是一般这么叫!
        参数: resolve,reject分别处理成功和失败的两个函数! 成功resolve(结果)  失败reject(结果)
        参数: 在function中调用这里两个方法,那么promise会处于两个不同的状态
        状态: promise有三个状态
                pending    正在运行
                resolved   内部调用了resolve方法
                rejected   内部调用了reject方法
        参数: 在第二步回调函数中就可以获取对应的结果
        */
        let promise =new Promise(function(resolve,reject){
            console.log("promise do some code ... ...")
            //resolve("promise success")
            reject("promise fail")
        })
        console.log('other code1111 invoked')
        //2.获取回调函数结果  then在这里会等待promise中的运行结果,但是不会阻塞代码继续运行
        promise.then(
            function(value){console.log(`promise中执行了resolve:${value}`)},
            function(error){console.log(`promise中执行了reject:${error}`)}
        )
        // 3 其他代码执行
        console.log('other code2222 invoked')
    </script>
```

9.0.4 Promise catch()

> `Promise.prototype.catch` 方法是 `.then(null, rejection)` 的别名，用于指定发生错误时的回调函数。

```
<script>
    let promise =new Promise(function(resolve,reject){
        console.log("promise do some code ... ...")
        // 故意响应一个异常对象
        throw new Error("error message")
    })
    console.log('other code1111 invoked')
    /*
        then中的reject()的对应方法可以在产生异常时执行,接收到的就是异常中的提示信息
        then中可以只留一个resolve()的对应方法,reject()方法可以用后续的catch替换
        then中的reject对应的回调函数被后续的catch替换后,catch中接收的数据是一个异常对象
        */
    promise.then(
        function(resolveValue){console.log(`promise中执行了resolve:${resolveValue}`)}
        //,
        //function(rejectValue){console.log(`promise中执行了reject:${rejectValue}`)}
    ).catch(
        function(error){console.log(error)}
    )
    console.log('other code2222 invoked')
</script>
```

9.0.5 async和await的使用

> async和await是ES6中用于处理异步操作的新特性。通常，异步操作会涉及到Promise对象，而async/await则是在Promise基础上提供了更加直观和易于使用的语法。

> async 用于标识函数的：

1. async标识函数后，async函数的返回值会变成一个Promise对象；
2. 如果函数内部返回的数据是一个非Promise对象，async函数的结果会返回一个成功状态 Promise对象；
3. 如果函数内部返回的是一个Promise对象，则async函数返回的状态与结果由该对象决定；
4. 如果函数内部抛出的是一个异常，则async函数返回的是一个失败的Promise对象；
5. async其实就是给我们提供了一个快捷声明回调函数的语法，有了它无需编写 new Promise(... ...) 这样的代码了；

```
<script>
        async function fun1(){
            //return 10
            //throw new Error("something wrong")
            let promise = Promise.reject("something wrong")
            return promise
        }
        let promise =fun1()
        promise.then(
            function(value){
                console.log("success:"+value)
            }
        ).catch(
            function(value){
                console.log("fail:"+value)
            }
        )
</script>
```

> await：

1. await右侧的表达式一般为一个Promise对象，但是也可以是一个其他值；
2. 如果表达式是Promise对象，await返回的是Promise成功的值；
3. 如果表达式是其他值，则直接返回该值；
4. await会等右边的Promise对象执行结束，然后再获取结果，所在方法的后续代码也会等待await的执行；
5. await必须在async函数中，但是async函数中可以没有await；
6. 如果await右边的Promise失败了，就会抛出异常，可以通过 try ... catch捕获处理；
7. await其实就是给我们提供了一个快捷获得Promise对象成功状态的语法，无需编写promise.then(... ...)这样的代码了；

```
<script>
        async function fun1(){
            return 10
        }
        async function fun2(){
            try{
                let res = await fun1()
                //let res = await Promise.reject("something wrong")
            }catch(e){
                console.log("catch got:"+e)
```

```
            }
            console.log("await got:"+res)
        }
        fun2()
</script>
```

9.1 Axios介绍

> AJAX：

- AJAX = Asynchronous JavaScript and XML（异步的 JavaScript 和 XML）；
- AJAX 不是新的编程语言，而是一种使用现有标准的新方法；
- AJAX 最大的优点是在不重新加载整个页面的情况下，可以与服务器交换数据并更新部分网页内容；
- AJAX 不需要任何浏览器插件，但需要用户允许 JavaScript 在浏览器上执行；
- XMLHttpRequest 只是实现 Ajax 的一种方式，本次我们使用Vue Axios方式实现；

> 什么是axios 官网介绍: **https://axios-http.com/zh/docs/intro**

- Axios 是一个基于 Promise网络请求库，作用于**node.js** 和浏览器中。 它是 ***isomorphic*** 的(即同一套代码可以运行在浏览器和node.js中)。在服务端它使用原生 node.js `http` 模块，而在客户端 (浏览端) 则使用 XMLHttpRequests。它有如下特性：
  - 从浏览器创建 **XMLHttpRequests**
  - 从 node.js 创建 **http** 请求
  - 支持 **Promise** API
  - 拦截请求和响应
  - 转换请求和响应数据
  - 取消请求
  - 自动转换JSON数据
  - 客户端支持防御 **XSRF**

9.2 Axios 入门案例

> 1 案例需求：请求后台获取随机土味情话。

- 请求的url

```
https://api.uomg.com/api/rand.qinghua?format=json  或者使用
http://forum.atguigu.cn/api/rand.qinghua?format=json
```

- 请求的方式

```
GET/POST
```

- 数据返回的格式

```
{"code":1,"content":"我努力不是为了你而是因为你。"}
```

> 2 准备项目：

```
npm create vite
npm install
```

3 安装Axios：

```
npm install axios
```

4 设计页面（App.Vue）：

```
<script setup type="module">
  import axios from 'axios'
  import { onMounted,reactive } from 'vue';
  let jsonData =reactive({code:1,content:'我努力不是为了你而是因为你'})
  let getLoveMessage =()=>{
    axios({
      method:"post", // 请求方式
      url:"https://api.uomg.com/api/rand.qinghua?format=json",  // 请求的url
      data:{ // 当请求方式为post时,data下的数据以JSON串放入请求体,否则以key=value形式放url后
        username:"123456"
      }
    }).then( function (response){//响应成功时要执行的函数
      console.log(response)
      Object.assign(jsonData,response.data)
    }).catch(function (error){// 响应失败时要执行的函数
      console.log(error)
    })
  }
  /* 通过onMounted生命周期,自动加载一次 */
  onMounted(()=>{
    getLoveMessage()
  })
</script>
<template>
    <div>
      <h1>今日土味情话:{{jsonData.content}}</h1>
      <button  @click="getLoveMessage">获取今日土味情话</button>
    </div>
</template>
<style scoped>
</style>
```

5 启动测试：

```
npm run dev
```

异步响应的数据结构：

- 响应的数据是经过包装返回的！一个请求的响应包含以下信息。

```
{
  // `data` 由服务器提供的响应
  data: {},
  // `status` 来自服务器响应的 HTTP 状态码
  status: 200,
  // `statusText` 来自服务器响应的 HTTP 状态信息
  statusText: 'OK',
  // `headers` 是服务器响应头
```

```
  // 所有的 header 名称都是小写，而且可以使用方括号语法访问
  // 例如: `response.headers['content-type']`
  headers: {},
  // `config` 是 `axios` 请求的配置信息
  config: {},
  // `request` 是生成此响应的请求
  // 在node.js中它是最后一个ClientRequest实例 (in redirects),
  // 在浏览器中则是 XMLHttpRequest 实例
  request: {}
}
```

- then取值

```
then(function (response) {
    console.log(response.data);
    console.log(response.status);
    console.log(response.statusText);
    console.log(response.headers);
    console.log(response.config);
});
```

6 通过async和await处理异步请求：

```
<script setup type="module">
  import axios from 'axios'
  import { onMounted,reactive } from 'vue';
  let jsonData =reactive({code:1,content:'我努力不是为了你而是因为你'})
  let getLoveWords = async ()=>{
    return await axios({
      method:"post",
      url:"https://api.uomg.com/api/rand.qinghua?format=json",
      data:{
        username:"123456"
      }
    })
  }
  let getLoveMessage =()=>{
     let {data}  = await getLoveWords()
     Object.assign(message,data)
  }
  /* 通过onMounted生命周期,自动加载一次 */
  onMounted(()=>{
    getLoveMessage()
  })
</script>
<template>
    <div>
      <h1>今日土味情话:{{jsonData.content}}</h1>
      <button  @click="getLoveMessage">获取今日土味情话</button>
    </div>
</template>
<style scoped>
</style>
```

axios在发送异步请求时的可选配置：

详情见 https://axios-http.com/zh/docs/req_config

9.3 Axios get和post方法

> 配置添加语法：

```
axios.get(url[, config])
axios.get(url,{
    上面指定配置key:配置值,
    上面指定配置key:配置值
})

axios.post(url[, data[, config]])
axios.post(url,{key:value //此位置数据，没有空对象即可{}},{
    上面指定配置key:配置值,
    上面指定配置key:配置值
})
```

> 测试axios.get(... ...)：

```
<script setup type="module">
  import axios from 'axios'
  import { onMounted,ref,reactive,toRaw } from 'vue';
  let jsonData =reactive({code:1,content:'我努力不是为了你而是因为你'})

  let getLoveWords= async ()=>{
    try{
      return await axios.get(
        'https://api.uomg.com/api/rand.qinghua',
        {
          params:{// 向url后添加的键值对参数
            format:'json',
            username:'zhangsan',
            password:'123456'
          },
          headers:{// 设置请求头
            'Accept' : 'application/json, text/plain, text/html,*/*'
          }
        }
      )
    }catch (e){
      return await e
    }
  }
  let getLoveMessage =()=>{
     let {data}  = await getLoveWords()
     Object.assign(message,data)
  }
  /* 通过onMounted生命周期,自动加载一次 */
  onMounted(()=>{
    getLoveMessage()
  })
</script>
<template>
    <div>
```

```
      <h1>今日土味情话:{{jsonData.content}}</h1>
      <button  @click="getLoveMessage">获取今日土味情话</button>
    </div>
</template>
<style scoped>
</style>
```

测试 axios.post(... ...):

```
<script setup type="module">
  import axios from 'axios'
  import { onMounted,ref,reactive,toRaw } from 'vue';
  let jsonData =reactive({code:1,content:'我努力不是为了你而是因为你'})
  let getLoveWords= async ()=>{
    try{
      return await axios.post(
        'https://api.uomg.com/api/rand.qinghua',
        {//请求体中的JSON数据
            username:'zhangsan',
            password:'123456'
        },
        {// 其他参数
            params:{// url上拼接的键值对参数
                format:'json',
            },
            headers:{// 请求头
                'Accept' : 'application/json, text/plain, text/html,*/*',
                'X-Requested-With': 'XMLHttpRequest'
            }
        }
      )
    }catch (e){
      return await e
    }
  }
  let getLoveMessage =()=>{
     let {data}  = await getLoveWords()
     Object.assign(message,data)
  }
  /* 通过onMounted生命周期,自动加载一次 */
  onMounted(()=>{
    getLoveMessage()
  })
</script>
<template>
    <div>
      <h1>今日土味情话:{{jsonData.content}}</h1>
      <button  @click="getLoveMessage">获取今日土味情话</button>
    </div>
</template>
<style scoped>
</style>
```

9.4 Axios 拦截器

> 如果想在axios发送请求之前，或者是数据响应回来在执行then方法之前做一些额外的工作，可以通过拦截器完成：

```
// 添加请求拦截器 请求发送之前
axios.interceptors.request.use(
  function (config) {
    // 在发送请求之前做些什么
    return config;
  },
  function (error) {
    // 对请求错误做些什么
    return Promise.reject(error);
  }
);
// 添加响应拦截器 数据响应回来
axios.interceptors.response.use(
  function (response) {
    // 2xx 范围内的状态码都会触发该函数。
    // 对响应数据做点什么
    return response;
  },
  function (error) {
    // 超出 2xx 范围的状态码都会触发该函数。
    // 对响应错误做点什么
    return Promise.reject(error);
  }
);
```

- 定义src/axios.js提取拦截器和配置语法

```
import axios from 'axios'
//  创建instance实例
const instance = axios.create({
    baseURL:'https://api.uomg.com',
    timeout:10000
})
//  添加请求拦截
instance.interceptors.request.use(
    // 设置请求头配置信息
    config=>{
        //处理指定的请求头
        console.log("before request")
        config.headers.Accept = 'application/json, text/plain, text/html,*/*'
        return config
    },
    // 设置请求错误处理函数
    error=>{
        console.log("request error")
        return Promise.reject(error)
    }
)
// 添加响应拦截器
instance.interceptors.response.use(
    // 设置响应正确时的处理函数
    response=>{
        console.log("after success response")
        console.log(response)
```

```
        return response
    },
    // 设置响应异常时的处理函数
    error=>{
        console.log("after fail response")
        console.log(error)
        return Promise.reject(error)
    }
)
// 默认导出
export default instance
```

- App.vue

```
<script setup type="module">
  // 导入自己定义的axios.js文件,而不是导入axios依赖
  import axios from './axios.js'
  import { onMounted,ref,reactive,toRaw } from 'vue';
  let jsonData =reactive({code:1,content:'我努力不是为了你而是因为你'})
  let getLoveWords= async ()=>{
    try{
      return await axios.post(
        'api/rand.qinghua',
        {
            username:'zhangsan',
            password:'123456'
        },//请求体中的JSON数据
        {
          params:{
            format:'json',
          }
        }// 其他键值对参数
      )
    }catch (e){
      return await e
    }
  }
  let getLoveMessage =()=>{
    let {data}  = await getLoveWords()
     Object.assign(message,data)
  }
  /* 通过onMounted生命周期,自动加载一次 */
  onMounted(()=>{
    getLoveMessage()
  })
</script>
<template>
    <div>
      <h1>今日土味情话:{{jsonData.content}}</h1>
      <button  @click="getLoveMessage">获取今日土味情话</button>
    </div>
</template>
<style scoped>
</style>
```

十、案例开发-日程管理-第六期

10.1 前端代码处理

10.1.1 创建src/utils/request.js文件

```
import axios from 'axios'
//  创建instance实例
const instance = axios.create({
    baseURL:'http://localhost:8080/'
})
//  添加请求拦截
instance.interceptors.request.use(
    // 设置请求头配置信息
    config=>{
        //处理指定的请求头
        return config
    },
    // 设置请求错误处理函数
    error=>{
        return Promise.reject(error)
    }
)
// 添加响应拦截器
instance.interceptors.response.use(
    // 设置响应正确时的处理函数
    response=>{
        return response
    },
    // 设置响应异常时的处理函数
    error=>{
        return Promise.reject(error)
    }
)
// 默认导出
export default instance
```

10.1.2 注册页面完成注册

```
<script setup>
    import {ref,reactive} from 'vue'
    /* 导入发送请求的axios对象 */
    import request from'../utils/request'
    import {useRouter} from 'vue-router'
    const router = useRouter()
    let registUser = reactive({
        username:"",
        userPwd:""
    })
    let usernameMsg=ref('')
    let userPwdMsg=ref('')
    let reUserPwdMsg=ref('')
    let reUserPwd=ref('')
    async function checkUsername(){
        let usernameReg= /^[a-zA-Z0-9]{5,10}$/
        if(!usernameReg.test(registUser.username)){
```

```
            usernameMsg.value="格式有误"
            return false
        }
        // 发送异步请求    继续校验用户名是否被占用
        let {data} = await request.post(`user/checkUsernameUsed?
username=${registUser.username}`)
        if(data.code != 200){
            usernameMsg.value="用户名占用"
            return false
        }
        usernameMsg.value="可用"
        return true
    }
    function checkUserPwd(){
        let userPwdReg = /^[0-9]{6}$/
        if(!userPwdReg.test(registUser.userPwd)){
            userPwdMsg.value="格式有误"
            return false
        }
        userPwdMsg.value="OK"
        return true
    }
    function checkReUserPwd(){
        let userPwdReg = /^[0-9]{6}$/
        if(!userPwdReg.test(reUserPwd.value)){
            reUserPwdMsg.value="格式有误"
            return false
        }
        if(registUser.userPwd != reUserPwd.value){
            reUserPwdMsg.value="两次密码不一致"
            return false
        }
        reUserPwdMsg.value="OK"
        return true
    }
    // 注册的方法
    async function regist(){
        // 校验所有的输入框是否合法
        let flag1 =await checkUsername()
        let flag2 =await checkUserPwd()
        let flag3 =await checkReUserPwd()
        if(flag1 && flag2 && flag3){
          let  {data}= await request.post("user/regist",registUser)
          if(data.code == 200){
            // 注册成功跳转 登录页
            alert("注册成功,快去登录吧")
            router.push("/login")
          }else{
            alert("抱歉,用户名被抢注了")
          }
        }else{
            alert("校验不通过,请求再次检查数据")
        }
    }
    function clearForm(){
        registUser.username=""
        registUser.userPwd=""
        usernameMsg.value=""
```

```
        userPwdMsg.value=""
        reUserPwd.value=""
        reUserPwdMsg.value=""
    }
</script>
<template>
  <div>
    <h3 class="ht">请注册</h3>
    <table class="tab" cellspacing="0px">
        <tr class="ltr">
            <td>请输入账号</td>
            <td>
                <input class="ipt"
                  id="usernameInput"
                  type="text"
                  name="username"
                  v-model="registUser.username"
                  @blur="checkUsername()">

                <span id="usernameMsg" class="msg" v-text="usernameMsg"></span>
            </td>
        </tr>
        <tr class="ltr">
            <td>请输入密码</td>
            <td>
                <input class="ipt"
                id="userPwdInput"
                type="password"
                name="userPwd"
                v-model="registUser.userPwd"
                @blur="checkUserPwd()">
                <span id="userPwdMsg" class="msg" v-text="userPwdMsg"></span>
            </td>
        </tr>
        <tr class="ltr">
            <td>确认密码</td>
            <td>
                <input class="ipt"
                id="reUserPwdInput"
                type="password"
                v-model="reUserPwd"
                @blur="checkReUserPwd()">
                <span id="reUserPwdMsg" class="msg" v-text="reUserPwdMsg"></span>
            </td>
        </tr>
        <tr class="ltr">
            <td colspan="2" class="buttonContainer">
                <input class="btn1" type="button" @click="regist()" value="注册">
                <input class="btn1" type="button" @click="clearForm()" value="重置">
                <router-link to="/login">
                  <button class="btn1">去登录</button>
                </router-link>
            </td>
        </tr>
    </table>
  </div>
</template>
<style scoped>
```

```
        .ht{
            text-align: center;
            color: cadetblue;
            font-family: 幼圆;
        }
        .tab{
            width: 500px;
            border: 5px solid cadetblue;
            margin: 0px auto;
            border-radius: 5px;
            font-family: 幼圆;
        }
        .ltr td{
            border: 1px solid  powderblue;
        }
        .ipt{
            border: 0px;
            width: 50%;
        }
        .btn1{
            border: 2px solid powderblue;
            border-radius: 4px;
            width:60px;
            background-color: antiquewhite;
        }
        .msg {
            color: gold;
        }
        .buttonContainer{
            text-align: center;
        }
</style>
```

10.1.3 登录页面完成登录

```
<script setup>
        import {ref,reactive} from 'vue'
        import {useRouter} from 'vue-router'
        const router = useRouter()
        import request  from '../utils/request'
        let loginUser =reactive({
                username:"",
                userPwd:""
        })
       let usernameMsg =ref("")
       let userPwdMsg =ref("")
       function checkUsername(){
            let usernameReg= /^[a-zA-Z0-9]{5,10}$/
            if(!usernameReg.test(loginUser.username)){
                usernameMsg.value="格式有误"
                return false
            }
            usernameMsg.value="OK"
            return true
       }

       function checkUserPwd(){
```

```
            let userPwdReg = /^[0-9]{6}$/

            if(!userPwdReg.test(loginUser.userPwd)){
                userPwdMsg.value="格式有误"
                return false
            }

            userPwdMsg.value="OK"
            return true
        }
        async function login(){
            // 表单数据格式都正确再提交
            let flag1 =checkUsername()
            let flag2 =checkUserPwd()
            if(!(flag1 && flag2)){
                return
            }
            let {data} = await request.post("user/login",loginUser)
            if(data.code == 200 ){
                alert("登录成功")
                // 跳转到showSchedule
                router.push("/showSchedule")
            } else if( data.code == 503){
                alert("密码有误")

            }else if (data.code == 501 ){
                alert("用户名有误")
            }else {
                alert("未知错误")
            }
        }
</script>
<template>
  <div>
    <h3 class="ht">请登录</h3>
        <table class="tab" cellspacing="0px">
            <tr class="ltr">
                <td>请输入账号</td>
                <td>
                    <input class="ipt"
                    type="text"
                    v-model="loginUser.username"
                    @blur="checkUsername()">
                    <span id="usernameMsg" v-text="usernameMsg"></span>
                </td>
            </tr>
            <tr class="ltr">
                <td>请输入密码</td>
                <td>
                    <input class="ipt"
                    type="password"
                    v-model="loginUser.userPwd"
                    @blur="checkUserPwd()">
                    <span id="userPwdMsg" v-text="userPwdMsg"></span>
                </td>
            </tr>
            <tr class="ltr">
                <td colspan="2" class="buttonContainer">
```

```
                    <input class="btn1" type="button" @click="login()" value="登录">
                    <input class="btn1" type="button" value="重置">
                    <router-link to="/regist">
                      <button class="btn1">去注册</button>
                    </router-link>
                </td>
            </tr>
        </table>
  </div>
</template>
<style scoped>
        .ht{
            text-align: center;
            color: cadetblue;
            font-family: 幼圆;
        }
        .tab{
            width: 500px;
            border: 5px solid cadetblue;
            margin: 0px auto;
            border-radius: 5px;
            font-family: 幼圆;
        }
        .ltr td{
            border: 1px solid  powderblue;
        }
        .ipt{
            border: 0px;
            width: 50%;
        }
        .btn1{
            border: 2px solid powderblue;
            border-radius: 4px;
            width:60px;
            background-color: antiquewhite;
        }
        #usernameMsg , #userPwdMsg {
            color: gold;
        }
        .buttonContainer{
            text-align: center;
        }
</style>
```

10.2 后端代码处理

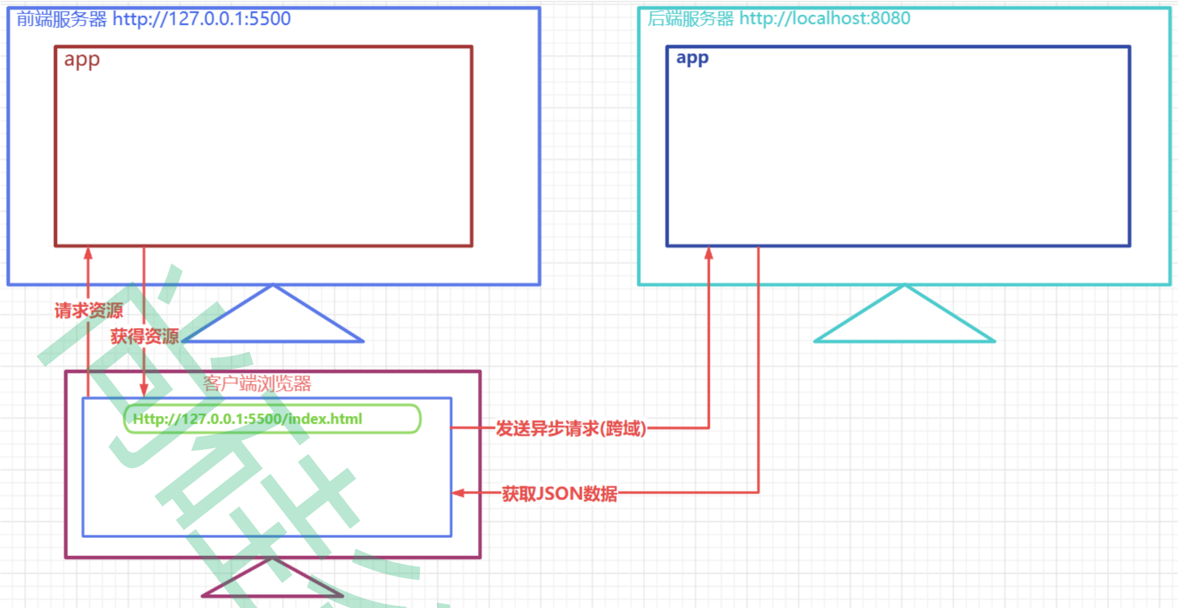
10.2.1 添加跨域处理器

10.2.1.1 什么是跨域

> 同源策略（SameOriginPolicy）是浏览器最核心也最基本的安全功能，如果缺少了同源策略，则浏览器的正常功能可能都会受到影响。可以说Web是构建在同源策略基础之上的，浏览器只是针对同源策略的一种实现。 同源策略会阻止一个域的**javascript**脚本和另外一个域的内容进行交互。所谓同源（即指在同一个域）就是两个页面具有相同的协议（**protocol**），主机（**host**）和端口号 。

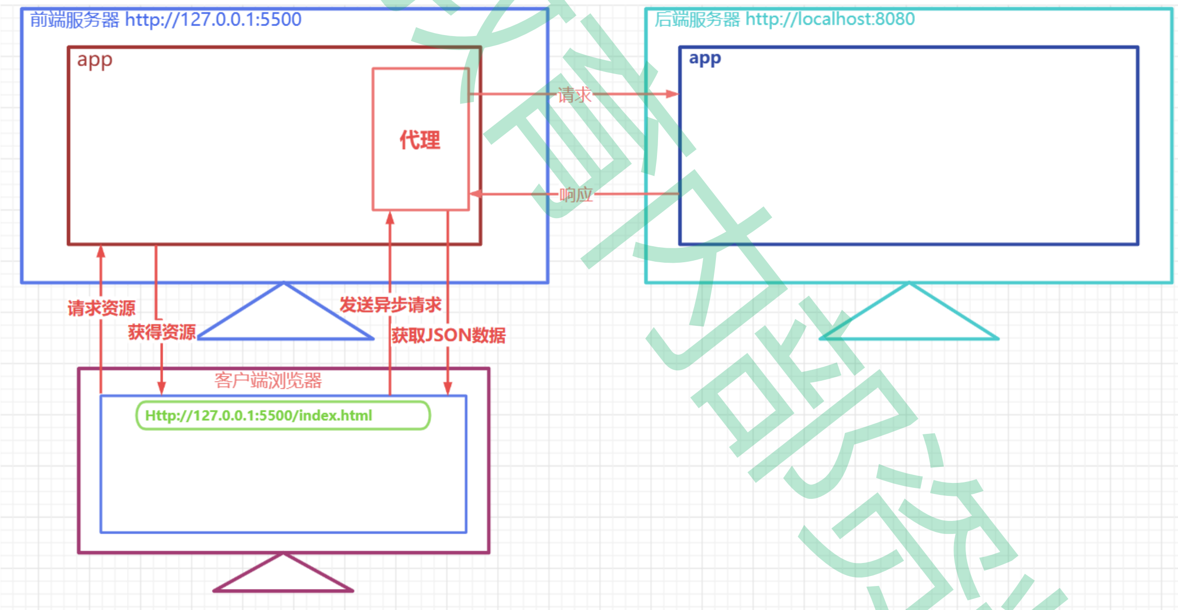
10.2.1.2为什么会产生跨域

前后端分离模式下，客户端请求前端服务器获取视图资源，然后客户端自行向后端服务器获取数据资源，前端服务器的 协议、IP和端口和后端服务器很可能是不一样的、这样就产生了跨域。

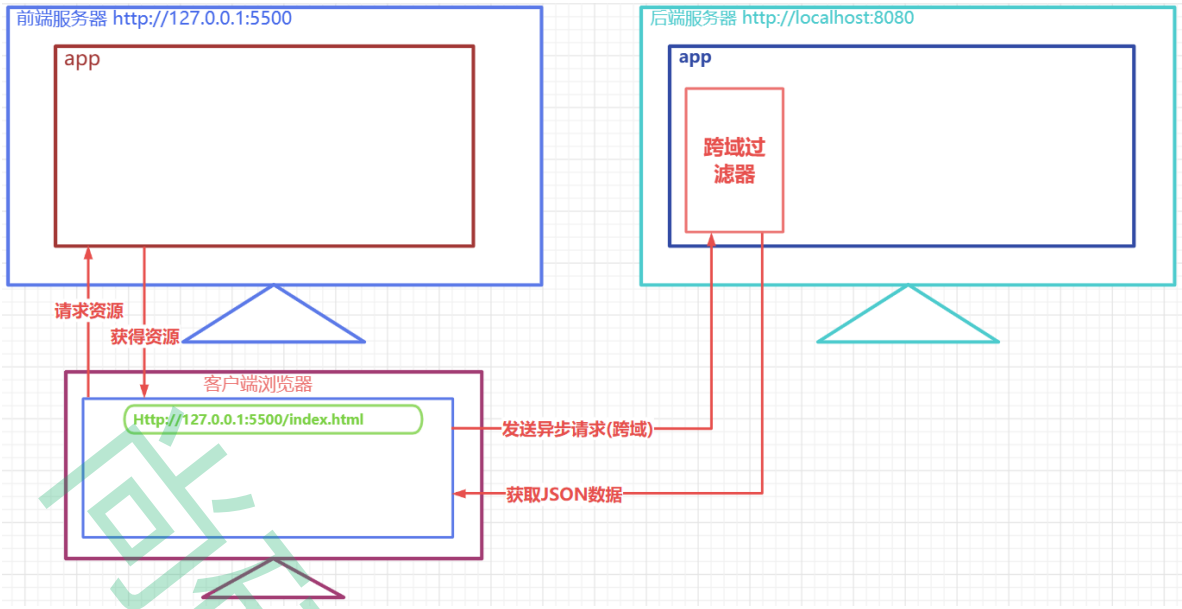


10.2.1.3 如何解决跨域

前端项目代理模式处理：



后端跨域过滤器方式处理：



- CrosFilter过滤器

```
package com.atguigu.schedule.filter;
import com.atguigu.schedule.common.Result;
import com.atguigu.schedule.util.WebUtil;
import jakarta.servlet.*;
import jakarta.servlet.annotation.WebFilter;
import jakarta.servlet.annotation.WebServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import java.io.IOException;
@WebFilter("/*")
public class CrosFilter implements Filter {
    @Override
    public void doFilter(ServletRequest servletRequest, ServletResponse
servletResponse, FilterChain filterChain) throws IOException, ServletException {
        HttpServletRequest request = (HttpServletRequest) servletRequest;
        System.out.println(request.getMethod());
        HttpServletResponse response = (HttpServletResponse) servletResponse;
        response.setHeader("Access-Control-Allow-Origin", "*");
        response.setHeader("Access-Control-Allow-Methods", "POST, GET, PUT,OPTIONS,
DELETE, HEAD");
        response.setHeader("Access-Control-Max-Age", "3600");
        response.setHeader("Access-Control-Allow-Headers", "access-control-allow-
origin, authority, content-type, version-info, X-Requested-With");
        // 如果是跨域预检请求,则直接在此响应200业务码
        if(request.getMethod().equalsIgnoreCase("OPTIONS")){
            WebUtil.writeJson(response, Result.ok(null));
        }else{
            // 非预检请求,放行即可
            filterChain.doFilter(servletRequest, servletResponse);
        }
    }
}
```

- 未来我们使用框架,直接用一个@CrossOrigin 就可以解决跨域问题了。

10.2.2 重构UserController

```
package com.atguigu.schedule.controller;
import com.atguigu.schedule.common.Result;
```

```
import com.atguigu.schedule.common.ResultCodeEnum;
import com.atguigu.schedule.pojo.SysUser;
import com.atguigu.schedule.service.SysUserService;
import com.atguigu.schedule.service.impl.SysUserServiceImpl;
import com.atguigu.schedule.util.MD5Util;
import com.atguigu.schedule.util.WebUtil;
import jakarta.servlet.ServletException;
import jakarta.servlet.annotation.WebServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import java.io.IOException;
@WebServlet("/user/*")
public class UserController extends BaseController{
    private SysUserService userService =new SysUserServiceImpl();
    /**
     * 注册时校验用户名是否被占用的业务接口
     * @param req
     * @param resp
     * @throws ServletException
     * @throws IOException
     */
    protected void checkUsernameUsed(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
        String username = req.getParameter("username");
       SysUser registUser = userService.findByUsername(username);

        //封装结果对象
        Result result=null;
        if(null ==registUser){
            // 未占用,创建一个code为200的对象
            result= Result.ok(null);
        }else{
            // 占用, 创建一个结果为505的对象
            result= Result.build(null, ResultCodeEnum.USERNAME_USED);
        }
        // 将result对象转换成JSON并响应给客户端
        WebUtil.writeJson(resp,result);
    }
    /**
     * 用户注册的业务接口
     * @param req
     * @param resp
     * @throws ServletException
     * @throws IOException
     */
    protected void regist(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        // 接收要注册的用户信息
        SysUser registUser = WebUtil.readJson(req, SysUser.class);
        // 调用服务层方法,将用户注册进入数据库
        int rows =userService.regist(registUser);
        Result result =null;
        if(rows>0){
           result=Result.ok(null);
        }else{
           result =Result.build(null,ResultCodeEnum.USERNAME_USED);
        }
        WebUtil.writeJson(resp,result);
```

```
    }
    /**
     * 用户登录的业务接口
     * @param req
     * @param resp
     * @throws ServletException
     * @throws IOException
     */
    protected void login(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        // 接收用户请求参数
        // 获取要登录的用户名密码
        SysUser inputUser = WebUtil.readJson(req, SysUser.class);
        // 调用服务层方法,根据用户名查询数据库中是否有一个用户
        SysUser loginUser =userService.findByUsername(inputUser.getUsername());

        Result result = null;

        if(null == loginUser){
            // 没有根据用户名找到用户,说明用户名有误
            result=Result.build(null,ResultCodeEnum.USERNAME_ERROR);
        }else if(!
loginUser.getUserPwd().equals(MD5Util.encrypt(inputUser.getUserPwd()))){
            // 用户密码有误,
           result=Result.build(null,ResultCodeEnum.PASSWORD_ERROR);
        }else{
            // 登录成功
            result=Result.ok(null);
        }
        WebUtil.writeJson(resp,result);
    }
}
```

10.2.3 删除登录校验过滤器

- 这里不使用cookie和session方式记录用户状态，未来使用token，所以登录过滤器删除即可。

十一、Vue3状态管理Pinia

11.1 Pinia介绍

> 如何实现多个组件之间的数据传递?

- 方式1 组件传参；
- 方式2 路由传参；
- 方式3 通过pinia状态管理定义共享数据;

> 当我们有 多个组件共享一个共同的状态(数据源) 时，多个视图可能都依赖于同一份状态。来自不同视图的交互也可能需要更改同一份状态。虽然我们的手动状态管理解决方案（props，组件间通信，模块化）在简单的场景中已经足够了，但是在大规模的生产应用中还有很多其他事项需要考虑:

- 更强的团队协作约定;

- 与 Vue DevTools 集成，包括时间轴、组件内部审查和时间旅行调试；
- 模块热更新 (HMR)；
- 服务端渲染支持；

> Pinia 就是一个实现了上述需求的状态管理库，由 Vue 核心团队维护，对 Vue 2 和 Vue 3 都可用。
> https://pinia.vuejs.org/zh/introduction.html

11.2 Pinia基本用法

> 1 准备Vite项目：

```
npm create vite
npm install
npm install vue-router@4 --save
```

> 2 安装Pinia：

```
npm install pinia
```

> 3 定义pinia store对象 src/store/store.js （推荐这么命名不是强制）：

```
import {defineStore } from 'pinia'
//定义数据并且对外暴露
// store就是定义共享状态的包装对象
// 内部包含四个属性： id 唯一标识 state 完整类型推理，推荐使用箭头函数 存放的数据 getters 类似属性计算，存储放对数据
// 操作的方法  actions 存储数据的复杂业务逻辑方法
// 理解： store类似Java中的实体类， id就是类名， state 就是装数据值的属性  getters就是get方法，actions就是对数据操作的其他方法
export const definedPerson = defineStore(
    {
        id: 'personPinia', //必须唯一
        state:()=>{ // state中用于定义数据
            return {
                username:'张三',
                age:0,
                hobbies:['唱歌','跳舞']
            }
        },
        getters:{// 用于定义一些通过数据计算而得到结果的一些方法 一般在此处不做对数据的修改操作
                 // getters中的方法可以当做属性值方式使用
            getHobbiesCount(){
                return this.hobbies.length
            },
            getAge(){
                return this.age
            }
        },
        actions:{ // 用于定义一些对数据修改的方法
            doubleAge(){
                this.age=this.age*2
            }
        }
```

```
    }
)
```

4 在main.js配置Pinia组件到Vue中：

```
import { createApp } from 'vue'
import App from './App.vue'
import router from './routers/router.js'
// 导pinia
import { createPinia } from 'pinia'
// 创建pinia对象
let pinia= createPinia()
let app =createApp(App)
app.use(router)
// app中使用pinia功能
app.use(pinia)
app.mount('#app')
```

5 Operate.vue 中操作Pinia数据：

```
<script setup type="module">
    import { ref} from 'vue';
    import { definedPerson} from '../store/store';
    // 读取存储的数据
    let person= definedPerson()
    let hobby = ref('')
</script>
<template>
    <div>
        <h1>operate视图,用户操作Pinia中的数据</h1>
        请输入姓名:<input type="text" v-model="person.username"> <br>
        请输入年龄:<input type="text" v-model="person.age"> <br>
        请增加爱好:
        <input type="checkbox" value="吃饭"  v-model="person.hobbies"> 吃饭
        <input type="checkbox" value="睡觉"  v-model="person.hobbies"> 睡觉
        <input type="checkbox" value="打豆豆"  v-model="person.hobbies"> 打豆豆 <br>

        <!-- 事件中调用person的doubleAge()方法 -->
        <button @click="person.doubleAge()">年龄加倍</button> <br>
        <!-- 事件中调用pinia提供的$reset()方法恢复数据的默认值 -->
        <button @click="person.$reset()">恢复默认值</button> <br>
        <!-- 事件中调用$patch方法一次性修改多个属性值 -->
        <button @click="person.$patch({username:'奥特曼',age:100,hobbies:['晒太阳','打怪
兽']})">变身奥特曼</button> <br>
        显示pinia中的person数据:{{person}}
    </div>
</template>
<style scoped>
</style>
```

6 List.vue中展示Pinia数据：

```
<script setup type="module">
    import { definedPerson} from '../store/store';
    // 读取存储的数据
```

```
    let person= definedPerson()
</script>
<template>
    <div>
        <h1>List页面,展示Pinia中的数据</h1>
        读取姓名:{{person.username}} <br>
        读取年龄:{{person.age}} <br>
        通过get年龄:{{person.getAge}} <br>
        爱好数量:{{person.getHobbiesCount}} <br>
        所有的爱好:
        <ul>
            <li v-for='(hobby,index) in person.hobbies' :key="index" v-text="hobby">
</li>
        </ul>
    </div>
</template>
<style scoped>
</style>
```

7 定义组件路由router.js:

```
// 导入路由创建的相关方法
import {createRouter,createWebHashHistory} from 'vue-router'
// 导入vue组件
import List  from '../components/List.vue'
import Operate  from '../components/Operate.vue'
// 创建路由对象,声明路由规则
const router = createRouter({
    history: createWebHashHistory(),
    routes:[
        {
            path:'/opearte',
            component:Operate
        },

        {
            path:'/list',
            component:List
        },
    ]
})
// 对外暴露路由对象
export default router;
```

8 App.vue中通过路由切换组件:

```
<script setup type="module">
</script>
<template>
    <div>
      <hr>
      <router-link to="/opearte">显示操作页</router-link> <br>
      <router-link to="/list">显示展示页</router-link> <br>
      <hr>
      <router-view></router-view>
    </div>
</template>
<style scoped>
</style>
```

> 9 启动测试：

```
npm run dev
```

11.3 Pinia其他细节

> State (状态) 在大多数情况下，state 都是你的 store 的核心。人们通常会先定义能代表他们 APP 的 state。在 Pinia 中，state 被定义为一个返回初始状态的函数。

- store.js

```
import {defineStore} from 'pinia'
export const definedPerson = defineStore('personPinia',
    {
        state:()=>{
            return {
                username:'',
                age:0,
                hobbies:['唱歌','跳舞']
            }
        },
        getters:{
            getHobbiesCount(){
                return this.hobbies.length
            },
            getAge(){
                return this.age
            }
        },
        actions:{
            doubleAge(){
                this.age=this.age*2
            }
        }
    }
)
```

- Operate.vue

```
<script setup type="module">
    import { ref} from 'vue';
```

```
    import { definedPerson} from '../store/store';
    // 读取存储的数据
    let person= definedPerson()
    let hobby = ref('')
    let addHobby= ()=> {
        console.log(hobby.value)
        person.hobbies.push(hobby.value)
    }
    // 监听状态
    person.$subscribe((mutation,state)=>{
        console.log('---subscribe---')
        /*
        mutation.storeId
            person.$id一样
        mutation.payload
            传递给 cartStore.$patch() 的补丁对象。
        state 数据状态,其实是一个代理
        */
        console.log(mutation)
        console.log(mutation.type)
        console.log(mutation.payload)
        console.log(mutation.storeId)
        console.log(person.$id)
        // 数据 其实是一个代理对象
        console.log(state)
    })
</script>
<template>
    <div>
        <h1>operate视图,用户操作Pinia中的数据</h1>
        请输入姓名:<input type="text" v-model="person.username"> <br>
        请输入年龄:<input type="text" v-model="person.age"> <br>
        请增加爱好:
        <input type="checkbox" value="吃饭"  v-model="person.hobbies"> 吃饭
        <input type="checkbox" value="睡觉"  v-model="person.hobbies"> 睡觉
        <input type="checkbox" value="打豆豆"  v-model="person.hobbies"> 打豆豆 <br>
        <input type="text" @change="addHobby" v-model="hobby"> <br>
        <!-- 事件中调用person的doubleAge()方法 -->
        <button @click="person.doubleAge()">年龄加倍</button> <br>
        <!-- 事件中调用pinia提供的$reset()方法恢复数据的默认值 -->
        <button @click="person.$reset()">恢复默认值</button> <br>
        <!-- 事件中调用$patch方法一次性修改多个属性值 -->
        <button @click="person.$patch({username:'奥特曼',age:100,hobbies:['晒太阳','打怪
兽']})">变身奥特曼</button> <br>
        person:{{person}}
    </div>
</template>
<style scoped>
</style>
```

2 Getter 完全等同于 store 的 state 的**计算值**。可以通过 `defineStore()` 中的 `getters` 属性来定义它们。推荐使用箭头函数，并且它将接收 `state` 作为第一个参数：

```
export const useStore = defineStore('main', {
  state: () => ({
    count: 0,
  }),
  getters: {
    doubleCount: (state) => state.count * 2,
  },
})
```

3 Action 相当于组件中的 **method**。它们可以通过 `defineStore()` 中的 `actions` 属性来定义，并且它们也是定义业务逻辑的完美选择。类似 **getter**，action 也可通过 `this` 访问整个 store 实例，并支持完整的类型标注(以及自动补全)。不同的是，`action` 可以是异步的，你可以在它们里面 `await` 调用任何 API，以及其他 action！

```
export const useCounterStore = defineStore('main', {
  state: () => ({
    count: 0,
  }),
  actions: {
    increment() {
      this.count++
    },
    randomizeCounter() {
      this.count = Math.round(100 * Math.random())
    },
  },
})
```

十二、案例开发-日程管理-第七期

12.1 前端使用Pinia存储数据

- 安装Pinia依赖

```
npm install pinia
```

- src下创建pinia.js文件

```
// 导入pinia组件
import {createPinia} from 'pinia'
// 创建pinia对象
let pinia = createPinia()
// 导出默认的pinia
export default pinia
```

- main.js中使用Pinia

```
import { createApp } from 'vue'
import App from './App.vue'
// 导入路由
import router from './router/router.js'
// 导入pinia对象
import pinia from './pinia.js'
let app =createApp(App)
// 全局使用路由
app.use(router)
// 全局使用pinia
app.use(pinia)
app.mount('#app')
```

- src/store/userStore.js 用于存储用户信息

```
import {defineStore} from 'pinia'
export const defineUser = defineStore('loginUser',{
    state:()=>{
        return {
            uid:0,
            username:''
        }
    },
    getters :{
    }
})
```

- src/store/scheduleStore.js 用于存储用户的日程信息

```
import {defineStore} from 'pinia'
export const defineSchedule = defineStore('scheduleList',{
    state:()=>{
        return {
            itemList:[
                /*
                {sid:1,uid:1,title:'学java',completed:1},
                {sid:1,uid:1,title:'学java',completed:1}
                */
            ]
        }
    },
    getters :{},
    actions:{}
})
```

- Header.vue中，通过pinia的数据来判断展示何种提示视图

```
<script setup>
    /* 导入pinia中的user数据 */
    import {defineUser} from '../store/userStore.js'
    import {defineSchedule} from '../store/scheduleStore.js'
    let sysUser =defineUser()
    let schedule = defineSchedule();
    /* 导入编程式路由 */
    import {useRouter} from 'vue-router'
    let  router =useRouter()
    /* 退出登录接口 */
    function logout(){
```

```
      // 清除userPina 和schedulepinia
      sysUser.$reset()
      schedule.$reset()
      // 通过路由回到登录页
      router.push("/login")
    }
</script>
<template>
  <div>
    <h1 class="ht">欢迎使用日程管理系统</h1>
    <div>
      <div  class="optionDiv" v-if="sysUser.username == ''">
        <router-link to="/login">
          <button class="b1s">登录</button>
        </router-link>
        <router-link to="/regist">
          <button class="b1s">注册</button>
        </router-link>
      </div>
      <div   class="optionDiv" v-else>
        欢迎{{sysUser.username}}
        <button class="b1b" @click="logout()">退出登录</button>
        <router-link to="/showSchedule">
          <button class="b1b">查看我的日程</button>
        </router-link>
      </div>
      <br>
    </div>
  </div>
</template>
<style scoped>

  .ht{
      text-align: center;
      color: cadetblue;
      font-family: 幼圆;
  }
  .b1s{
        border: 2px solid powderblue;
        border-radius: 4px;
        width:60px;
        background-color: antiquewhite;
    }
    .b1b{
        border: 2px solid powderblue;
        border-radius: 4px;
        width:100px;
        background-color: antiquewhite;
    }
    .optionDiv{
      width: 500px;
      float: right;
    }
</style>
```

- Login.vue中，登录成功后，接收后端响应回来的用户id和用户名，存储于userStore中

```
<script setup>
```

```
/* 导入pinia中的user数据 */
import {defineUser} from '../store/userStore.js'
let sysUser =defineUser()
/* 获取 编程式路由对象 */
import {useRouter} from 'vue-router'
let router =useRouter();
/* 导入axios请求对象 */
import request from '../utils/request.js'
// 导入ref,reactive处理响应式数据的方法
import{ ref,reactive} from 'vue'
// 响应式数据,保存用户输入的表单信息
let loginUser =reactive({
 username:'',
 userPwd:''
})
// 响应式数据,保存校验的提示信息
let usernameMsg =ref('')
let userPwdMsg = ref('')
 // 校验用户名的方法
 function checkUsername(){
     // 定义正则
     var usernameReg=/^[a-zA-Z0-9]{5,10}$/
     // 校验用户名
     if(!usernameReg.test(loginUser.username)){
         // 格式不合法
         usernameMsg.value="格式有误"
         return false
     }
     usernameMsg.value="ok"
     return true
 }
 // 校验密码的方法
 function checkUserPwd(){
     // 定义正则
     var passwordReg=/^[0-9]{6}$/
      // 校验密码
      if(!passwordReg.test(loginUser.userPwd)){
         // 格式不合法
         userPwdMsg.value="格式有误"
         return false
     }
     userPwdMsg.value="ok"
     return true
 }
 // 登录的函数
 async function  login(){
     console.log("发送异步请求")
     let {data} = await request.post("/user/login",loginUser)
     if(data.code == 200){
         alert("登录成功")
         // 更新pinia数据
         sysUser.uid =data.data.loginUser.uid
         sysUser.username =data.data.loginUser.username
         // 跳转到日程查询页
         router.push("/showSchedule")

     }else if(data.code == 501){
         alert("用户名有误,请重新输入")
```

```
            }else if(data.code == 503){
                alert("密码有误,请重新输入")
            }else {
                alert("出现未知名错误")
            }
        }
        // 清除表单信息的方法
        function clearForm(){
            loginUser.username=''
            loginUser.userPwd=''
            usernameMsg.value=''
            userPwdMsg.value=''
        }
</script>
<template>
  <div>
    <h3 class="ht">请登录</h3>
        <table class="tab" cellspacing="0px">
            <tr class="ltr">
                <td>请输入账号</td>
                <td>
                    <input class="ipt"
                    type="text"
                    v-model="loginUser.username"
                    @blur="checkUsername()">
                    <span id="usernameMsg" v-text="usernameMsg"></span>
                </td>
            </tr>
            <tr class="ltr">
                <td>请输入密码</td>
                <td>
                    <input class="ipt"
                    type="password"
                    v-model="loginUser.userPwd"
                    @blur="checkUserPwd()">
                    <span id="userPwdMsg" v-text="userPwdMsg"></span>
                </td>
            </tr>
            <tr class="ltr">
                <td colspan="2" class="buttonContainer">
                    <input class="btn1" type="button" @click="login()" value="登录">
                    <input class="btn1" type="button" @click="clearForm()" value="重
置">
                    <router-link to="/regist">
                      <button class="btn1">去注册</button>
                    </router-link>
                </td>
            </tr>
        </table>
  </div>
</template>
<style scoped>
        .ht{
            text-align: center;
            color: cadetblue;
            font-family: 幼圆;
        }
        .tab{
```

```
            width: 500px;
            border: 5px solid cadetblue;
            margin: 0px auto;
            border-radius: 5px;
            font-family: 幼圆;
        }
        .ltr td{
            border: 1px solid  powderblue;
        }
        .ipt{
            border: 0px;
            width: 50%;
        }
        .btn1{
            border: 2px solid powderblue;
            border-radius: 4px;
            width:60px;
            background-color: antiquewhite;
        }
        #usernameMsg , #userPwdMsg {
            color: gold;
        }
        .buttonContainer{
            text-align: center;
        }
</style>
```

- 服务端登录处理方法，登录成功，返回登录用户的信息

```
   /**
     * 用户登录的业务接口
     * @param req
     * @param resp
     * @throws ServletException
     * @throws IOException
     */
    protected void login(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {

        // 接收用户请求参数
        // 获取要登录的用户名密码
        SysUser inputUser = WebUtil.readJson(req, SysUser.class);
        // 调用服务层方法,根据用户名查询数据库中是否有一个用户
        SysUser loginUser =userService.findByUsername(inputUser.getUsername());

        Result result = null;

        if(null == loginUser){
            // 没有根据用户名找到用户,说明用户名有误
            result=Result.build(null,ResultCodeEnum.USERNAME_ERROR);
        }else if(!
loginUser.getUserPwd().equals(MD5Util.encrypt(inputUser.getUserPwd()))){
            // 用户密码有误,
           result=Result.build(null,ResultCodeEnum.PASSWORD_ERROR);
        }else{
            // 登录成功,将用户信息存入session
            req.getSession().setAttribute("sysUser",loginUser);
            // 登录成功
```

```
            // 将密码请空后,将用户信息响应给客户端
            loginUser.setUserPwd("");
            Map<String,Object> data =new HashMap<>();
            data.put("loginUser",loginUser);
            result=Result.ok(data);
        }

        WebUtil.writeJson(resp,result);

    }
```

- router.js中，通过路由守卫控制只有登录状态下才可以进入showSchedule.vue

```
import {createRouter,createWebHashHistory} from 'vue-router'
import pinia from '../pinia.js'
// 导入pinia数据
import {defineUser} from '../store/userStore.js'
let sysUser = defineUser(pinia)
import Login from '../components/Login.vue'
import Regist from '../components/Regist.vue'
import ShowScedule from '../components/ShowSchedule.vue'

let  router = createRouter({
    history:createWebHashHistory(),
    routes:[
        {
            path:"/",
            component:Login
        },
        {
            path:"/login",
            component:Login
        },
        {
            path:"/showSchedule",
            component:ShowScedule
        },
        {
            path:"/regist",
            component:Regist
        }
    ]
})
/* 配置路由的前置守卫,在登录状态下才可以范文showSchedule.vue */
router.beforeEach( (to,from,next) =>{
    // 如果是查看日程
    if(to.path=="/showSchedule"){
        // 如果尚未的登录
        if(sysUser.username == ''){
            alert("您尚未登录,请登录后再查看日程")
            next("/login")
        }else{
            // 已经登录 放行
            next()
        }
        // 其他资源 放行
    }else{
        next()
```

```
    }
})
export default router
```

12.2 显示所有日程数据

- ShowSchedule.vue中向后端发送异步请求查询数据并展示

```
<script setup>
    /* 引入axios */
    import request from '../utils/request.js'
    /* 引入pinia数据 */
    import {defineSchedule} from '../store/scheduleStore.js'
    import {defineUser} from '../store/userStore.js'
    let schedule = defineSchedule();
    let sysUser = defineUser()
    /* 引入挂载生命周期 */
    import { onMounted,onUpdated,ref,reactive } from 'vue';
    // 第一次挂载就立刻向后端发送请求,获取最新数据
    onMounted(async function (){
        showSchedule()
    })
    async function showSchedule(){
        let {data} = await request.get("/schedule/findAllSchedule",{params:
{"uid":sysUser.uid}})
        schedule.itemList =data.data.itemList
    }
</script>
<template>
  <div>
    <h3 class="ht">您的日程如下</h3>
<table class="tab" cellspacing="0px">
    <tr class="ltr">
        <th>编号</th>
        <th>内容</th>
        <th>进度</th>
        <th>操作</th>
    </tr>
    <tr class="ltr" v-for="item,index in schedule.itemList" :key="index">
        <td v-text="index+1">
        </td>
        <td>
            <input type="input" v-model="item.title">
        </td>
        <td>
            <input type="radio"   value="1" v-model="item.completed"> 已完成
            <input type="radio"   value="0" v-model="item.completed"> 未完成
        </td>
        <td class="buttonContainer">
             <button class="btn1">删除</button>
            <button class="btn1">保存修改</button>
        </td>
    </tr>
    <tr class="ltr buttonContainer" >
        <td colspan="4">
            <button class="btn1">新增日程</button>
```

```
        </td>
    </tr>
</table>
  </div>
</template>
<style scoped>
      .ht{
            text-align: center;
            color: cadetblue;
            font-family: 幼圆;
        }
        .tab{
            width: 80%;
            border: 5px solid cadetblue;
            margin: 0px auto;
            border-radius: 5px;
            font-family: 幼圆;
        }
        .ltr td{
            border: 1px solid  powderblue;

        }
        .ipt{
            border: 0px;
            width: 50%;

        }
        .btn1{
            border: 2px solid powderblue;
            border-radius: 4px;
            width:100px;
            background-color: antiquewhite;

        }
        #usernameMsg , #userPwdMsg {
            color: gold;
        }
        .buttonContainer{
            text-align: center;
        }
</style>
```

- SysScheduleController中查询数据并响应json

```
package com.atguigu.schedule.controller;
import com.atguigu.schedule.common.Result;
import com.atguigu.schedule.pojo.SysSchedule;
import com.atguigu.schedule.service.SysScheduleService;
import com.atguigu.schedule.service.impl.SysScheduleServiceImpl;
import com.atguigu.schedule.util.WebUtil;
import jakarta.servlet.ServletException;
import jakarta.servlet.annotation.WebServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
```

```
@WebServlet("/schedule/*")
public class SysScheduleController  extends BaseController{
    private SysScheduleService scheduleService =new SysScheduleServiceImpl();
    /**
     * 查询所有日程接口
     * @param req
     * @param resp
     * @throws ServletException
     * @throws IOException
     */
    protected void findAllSchedule(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
        int uid = Integer.parseInt(req.getParameter("uid"));
        //  调用服务层方法,查询所有日程
        List<SysSchedule> itemList = scheduleService.findItemListByUid(uid);
        // 将日程信息装入result,转换JSON给客户端
        Map<String,Object> data =new HashMap<>();
        data.put("itemList",itemList);
        WebUtil.writeJson(resp,Result.ok(data));
    }
}
```

- SysScheduleService接口和实现类代码

```
package com.atguigu.schedule.service;
import com.atguigu.schedule.pojo.SysSchedule;
import java.util.List;
public interface SysScheduleService {
    List<SysSchedule> findItemListByUid(int uid);
}
```

```
package com.atguigu.schedule.service.impl;
import com.atguigu.schedule.dao.SysScheduleDao;
import com.atguigu.schedule.dao.impl.SysScheduleDaoImpl;
import com.atguigu.schedule.pojo.SysSchedule;
import com.atguigu.schedule.service.SysScheduleService;
import java.util.List;
public class SysScheduleServiceImpl implements SysScheduleService {
    private SysScheduleDao scheduleDao =new SysScheduleDaoImpl();
    @Override
    public List<SysSchedule> findItemListByUid(int uid) {
        return scheduleDao.findItemListByUid(uid);
    }
}
```

- SysScheduleDao接口和实现类代码

```
package com.atguigu.schedule.dao;
import com.atguigu.schedule.pojo.SysSchedule;
import java.util.List;
public interface SysScheduleDao {
    List<SysSchedule> findItemListByUid(int uid);
}
```

```
package com.atguigu.schedule.dao.impl;
import com.atguigu.schedule.dao.BaseDao;
import com.atguigu.schedule.dao.SysScheduleDao;
import com.atguigu.schedule.pojo.SysSchedule;
import java.util.List;
public class SysScheduleDaoImpl extends BaseDao implements SysScheduleDao {
    @Override
    public List<SysSchedule> findItemListByUid(int uid) {
        String sql ="select sid,uid,title, completed from sys_schedule  where uid = ?
";
        return baseQuery(SysSchedule.class,sql,uid);
    }
}
```

12.3 增加和修改日程数据

- ShowSchedule.vue下,为增加和修改按钮绑定事件

```
<script setup>
    /* 引入axios */
    import request from '../utils/request.js'
    /* 引入pinia数据 */
    import {defineSchedule} from '../store/scheduleStore.js'
    import {defineUser} from '../store/userStore.js'
    let schedule = defineSchedule();
    let sysUser = defineUser()
    /* 引入挂载生命周期 */
    import { onMounted,onUpdated,ref,reactive } from 'vue';
    // 第一次挂载就立刻向后端发送请求,获取最新数据
    onMounted(async function (){
        // 加载完毕后,立刻调用查询数据的方法
        showSchedule()
    })
    async function  showSchedule(){
        let {data} = await request.get("/schedule/findAllSchedule",{params:
{"uid":sysUser.uid}})
        schedule.itemList =data.data.itemList
    }
    // 新增日程
    async function addItem(){
        // 立刻向后端发送一个请求,让后端先为当前用户在数据库中增加一个默认格式的空数据
        let {data} = await request.get("/schedule/addDefaultSchedule",{params:
{"uid":sysUser.uid}})
        if(data.code == 200){
            // 然后调用刷新页面数据方法,立刻获取最新数据
            showSchedule()
        }else{
            alert("添加异常")
        }
    }
    // 更新日程的方法
    async function updateItem(index){
        // 根据索引获取元素
        // 将元素通过 JSON串的形式 发送给服务端
```

```
        let {data} =await
request.post("/schedule/updateSchedule",schedule.itemList[index])
        if(data.code == 200){
            // 服务端修改完毕后,刷新页面数据
            showSchedule()
        }else{
            alert("更新异常")
        }
    }
</script>

<template>
  <div>
    <h3 class="ht">您的日程如下</h3>
<table class="tab" cellspacing="0px">
    <tr class="ltr">
        <th>编号</th>
        <th>内容</th>
        <th>进度</th>
        <th>操作</th>
    </tr>
    <tr class="ltr" v-for="item,index in schedule.itemList" :key="index">
        <td v-text="index+1">
        </td>
        <td>
            <input type="input" v-model="item.title">
        </td>
        <td>
            <input type="radio"   value="1" v-model="item.completed"> 已完成
            <input type="radio"   value="0" v-model="item.completed"> 未完成
        </td>
        <td class="buttonContainer">
             <button class="btn1">删除</button>
            <button class="btn1" @click="updateItem(index)">保存修改</button>
        </td>
    </tr>
    <tr class="ltr buttonContainer" >
        <td colspan="4">
            <button class="btn1" @click="addItem()">新增日程</button>
        </td>
    </tr>
</table>
  </div>
</template>
<style scoped>

      .ht{
            text-align: center;
            color: cadetblue;
            font-family: 幼圆;
        }
        .tab{
            width: 80%;
            border: 5px solid cadetblue;
            margin: 0px auto;
            border-radius: 5px;
            font-family: 幼圆;
        }
```

```
        .ltr td{
            border: 1px solid  powderblue;

        }
        .ipt{
            border: 0px;
            width: 50%;

        }
        .btn1{
            border: 2px solid powderblue;
            border-radius: 4px;
            width:100px;
            background-color: antiquewhite;

        }
        #usernameMsg , #userPwdMsg {
            color: gold;
        }
        .buttonContainer{
            text-align: center;
        }
</style>
```

- SysScheduleController处理 新增和保存修改业务处理接口

```
package com.atguigu.schedule.controller;
import com.atguigu.schedule.common.Result;
import com.atguigu.schedule.pojo.SysSchedule;
import com.atguigu.schedule.service.SysScheduleService;
import com.atguigu.schedule.service.impl.SysScheduleServiceImpl;
import com.atguigu.schedule.util.WebUtil;
import jakarta.servlet.ServletException;
import jakarta.servlet.annotation.WebServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
@WebServlet("/schedule/*")
public class SysScheduleController  extends BaseController{
    private SysScheduleService scheduleService =new SysScheduleServiceImpl();
    /**
     * 向数据库中增加一个新的默认数据的方法
     * @param req
     * @param resp
     * @throws ServletException
     * @throws IOException
     */
    protected void addDefaultSchedule(HttpServletRequest req, HttpServletResponse
resp) throws ServletException, IOException {
        int uid = Integer.parseInt(req.getParameter("uid"));
        //  调用服务层方法,为当前用户新增一个默认空数据
        scheduleService.addDefault(uid);
        WebUtil.writeJson(resp,Result.ok(null));
    }
    /**
```

```
     * 更新日程业务接口
     * @param req
     * @param resp
     * @throws ServletException
     * @throws IOException
     */
    protected void updateSchedule(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
        SysSchedule sysSchedule = WebUtil.readJson(req, SysSchedule.class);
        // 调用服务层方法,更新数据
        scheduleService.updateSchedule(sysSchedule);
        // 响应成功
        WebUtil.writeJson(resp,Result.ok(null));
    }
}
```

- SysScheduleService接口和实现类处理业务逻辑

```
package com.atguigu.schedule.service;
import com.atguigu.schedule.pojo.SysSchedule;
import java.util.List;
public interface SysScheduleService {
    Integer addDefault(int uid);
    Integer updateSchedule(SysSchedule sysSchedule);
}
```

```
package com.atguigu.schedule.service.impl;
import com.atguigu.schedule.dao.SysScheduleDao;
import com.atguigu.schedule.dao.impl.SysScheduleDaoImpl;
import com.atguigu.schedule.pojo.SysSchedule;
import com.atguigu.schedule.service.SysScheduleService;
import java.util.List;
public class SysScheduleServiceImpl implements SysScheduleService {
    private SysScheduleDao scheduleDao =new SysScheduleDaoImpl();
    @Override
    public Integer addDefault(int uid) {
        return scheduleDao.addDefault(uid);
    }
    @Override
    public Integer updateSchedule(SysSchedule sysSchedule) {
        return scheduleDao.updateSchedule(sysSchedule);
    }
}
```

- SysScheduleDao接口和实现类操作数据

```
package com.atguigu.schedule.dao;
import com.atguigu.schedule.pojo.SysSchedule;
import java.util.List;
public interface SysScheduleDao {
    Integer addDefault(int uid);
    Integer updateSchedule(SysSchedule sysSchedule);
}
```

```
package com.atguigu.schedule.dao.impl;
import com.atguigu.schedule.dao.BaseDao;
import com.atguigu.schedule.dao.SysScheduleDao;
import com.atguigu.schedule.pojo.SysSchedule;
```

```
import java.util.List;
public class SysScheduleDaoImpl extends BaseDao implements SysScheduleDao {
    @Override
    public Integer addDefault(int uid) {
        String sql = "insert into sys_schedule value(default,?,'请输入日程',0)";
        return baseUpdate(sql,uid);
    }
    @Override
    public Integer updateSchedule(SysSchedule sysSchedule) {
        String sql ="update sys_schedule set title = ? ,completed =  ? where sid =?";
        return
baseUpdate(sql,sysSchedule.getTitle(),sysSchedule.getCompleted(),sysSchedule.getSid())
;
    }
}
```

12.5 删除日程数据

- ShowSchedule.vue中,为删除按钮增加事件

```
<script setup>
    /* 引入axios */
    import request from '../utils/request.js'
    /* 引入pinia数据 */
    import {defineSchedule} from '../store/scheduleStore.js'
    import {defineUser} from '../store/userStore.js'
    let schedule = defineSchedule();
    let sysUser = defineUser()
    /* 引入挂载生命周期 */
    import { onMounted,onUpdated,ref,reactive } from 'vue';

    // 第一次挂载就立刻向后端发送请求,获取最新数据
    onMounted(async function (){
        // 加载完毕后,立刻调用查询数据的方法
        showSchedule()
    })
    async function  showSchedule(){
        let {data} = await request.get("/schedule/findAllSchedule",{params:
{"uid":sysUser.uid}})
        schedule.itemList =data.data.itemList
    }
    // 新增日程
    async function addItem(){
        // 立刻向后端发送一个请求,让后端先为当前用户在数据库中增加一个默认格式的空数据
        let {data} = await request.get("/schedule/addDefaultSchedule",{params:
{"uid":sysUser.uid}})
        if(data.code == 200){
            // 然后调用刷新页面数据方法,立刻获取最新数据
            showSchedule()
        }else{
            alert("添加异常")
        }
    }
    // 更新日程的方法
    async function updateItem(index){
        // 根据索引获取元素
        // 将元素通过 JSON串的形式 发送给服务端
```

```
        let {data} =await
request.post("/schedule/updateSchedule",schedule.itemList[index])
        if(data.code = 200){
            // 服务端修改完毕后,刷新页面数据
            showSchedule()
        }else{
            alert("更新异常")
        }

    }
    // 删除日程的方法
    async function removeItem(index){
        // 弹窗提示是否删除
        if(confirm("确定要删除该条数据")){
            // 根据索引获取要删除的item的id
            let sid = schedule.itemList[index].sid
            // 向服务端发送请求删除元素
            let{data} = await request.get("/schedule/removeSchedule",{params:
{"sid":sid}})
            //根据业务码判断删除是否成功
            if(data.code == 200){
                // 删除成功,更新数据
                showSchedule()
            }else{
                // 删除失败,提示失败
                alert("删除失败")
            }
        }
    }
</script>

<template>
  <div>
    <h3 class="ht">您的日程如下</h3>
<table class="tab" cellspacing="0px">
    <tr class="ltr">
        <th>编号</th>
        <th>内容</th>
        <th>进度</th>
        <th>操作</th>
    </tr>
    <tr class="ltr" v-for="item,index in schedule.itemList" :key="index">
        <td v-text="index+1">
        </td>
        <td>
            <input type="input" v-model="item.title">
        </td>
        <td>
            <input type="radio"   value="1" v-model="item.completed"> 已完成
            <input type="radio"   value="0" v-model="item.completed"> 未完成
        </td>
        <td class="buttonContainer">
             <button class="btn1"  @click="removeItem(index)">删除</button>
            <button class="btn1" @click="updateItem(index)">保存修改</button>
        </td>
    </tr>
    <tr class="ltr buttonContainer" >
        <td colspan="4">
```

```
            <button class="btn1" @click="addItem()">新增日程</button>
        </td>

    </tr>
</table>
  </div>
</template>
<style scoped>

      .ht{
            text-align: center;
            color: cadetblue;
            font-family: 幼圆;
        }
        .tab{
            width: 80%;
            border: 5px solid cadetblue;
            margin: 0px auto;
            border-radius: 5px;
            font-family: 幼圆;
        }
        .ltr td{
            border: 1px solid  powderblue;

        }
        .ipt{
            border: 0px;
            width: 50%;

        }
        .btn1{
            border: 2px solid powderblue;
            border-radius: 4px;
            width:100px;
            background-color: antiquewhite;
        }
        #usernameMsg , #userPwdMsg {
            color: gold;
        }
        .buttonContainer{
            text-align: center;
        }
</style>
```

- SysScheduleController中添加删除业务接口

```
package com.atguigu.schedule.controller;
import com.atguigu.schedule.common.Result;
import com.atguigu.schedule.pojo.SysSchedule;
import com.atguigu.schedule.service.SysScheduleService;
import com.atguigu.schedule.service.impl.SysScheduleServiceImpl;
import com.atguigu.schedule.util.WebUtil;
import jakarta.servlet.ServletException;
import jakarta.servlet.annotation.WebServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.util.HashMap;
```

```
import java.util.List;
import java.util.Map;
@WebServlet("/schedule/*")
public class SysScheduleController  extends BaseController{
    private SysScheduleService scheduleService =new SysScheduleServiceImpl();
    /**
     * 删除日程业务接口
     * @param req
     * @param resp
     * @throws ServletException
     * @throws IOException
     */
    protected void removeSchedule(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
        // 获取要删除的日程id
        int sid = Integer.parseInt(req.getParameter("sid"));
        // 调用服务层方法,删除日程
        scheduleService.removeSchedule(sid);
        // 响应200
        WebUtil.writeJson(resp,Result.ok(null));
    }
}
```

- SysScheduleService层处理删除业务的接口和实现类

```
package com.atguigu.schedule.service;
import com.atguigu.schedule.pojo.SysSchedule;
import java.util.List;
public interface SysScheduleService {
    Integer removeSchedule(int sid);
}
```

```
package com.atguigu.schedule.service.impl;
import com.atguigu.schedule.dao.SysScheduleDao;
import com.atguigu.schedule.dao.impl.SysScheduleDaoImpl;
import com.atguigu.schedule.pojo.SysSchedule;
import com.atguigu.schedule.service.SysScheduleService;
import java.util.List;
public class SysScheduleServiceImpl implements SysScheduleService {
    private SysScheduleDao scheduleDao =new SysScheduleDaoImpl();
    @Override
    public Integer removeSchedule(int sid) {
        return scheduleDao.removeBySid(sid);
    }
}
```

- SysScheduleDao操作数据库的接口和实现类

```
package com.atguigu.schedule.dao;
import com.atguigu.schedule.pojo.SysSchedule;
import java.util.List;
public interface SysScheduleDao {
    Integer removeBySid(int sid);
}
```

```
package com.atguigu.schedule.dao.impl;
import com.atguigu.schedule.dao.BaseDao;
import com.atguigu.schedule.dao.SysScheduleDao;
import com.atguigu.schedule.pojo.SysSchedule;
import java.util.List;
public class SysScheduleDaoImpl extends BaseDao implements SysScheduleDao {
    @Override
    public Integer removeBySid(int sid) {
        String sql ="delete from sys_schedule where sid = ?";
        return baseUpdate(sql,sid);
    }
}
```

十三、Element-plus组件库

13.1 Element-plus介绍

> Element Plus 是一套基于 Vue 3 的开源 UI 组件库，是由饿了么前端团队开发的升级版本 Element UI。Element Plus 提供了丰富的 UI 组件、易于使用的 API 接口和灵活的主题定制功能，可以帮助开发者快速构建高质量的 Web 应用程序。

- Element Plus 支持按需加载，且不依赖于任何第三方 CSS 库，它可以轻松地集成到任何 Vue.js 项目中。Element Plus 的文档十分清晰，提供了各种组件的使用方法和示例代码，方便开发者快速上手。
- Element Plus 目前已经推出了大量的常用 UI 组件，如按钮、表单、表格、对话框、选项卡等，此外还提供了一些高级组件，如日期选择器、时间选择器、级联选择器、滑块、颜色选择器等。这些组件具有一致的设计和可靠的代码质量，可以为开发者提供稳定的使用体验。
- 与 Element UI 相比，Element Plus 采用了现代化的技术架构和更加先进的设计理念，同时具备更好的性能和更好的兼容性。Element Plus 的更新迭代也更加频繁，可以为开发者提供更好的使用体验和更多的功能特性。
- Element Plus 可以在支持 ES2018 和 ResizeObserver 的浏览器上运行。 如果您确实需要支持旧版本的浏览器，请自行添加 Babel 和相应的 Polyfill
- 官网 https://element-plus.gitee.io/zh-CN/
- 由于 Vue 3 不再支持 IE11，Element Plus 也不再支持 IE 浏览器。

Edge ≥ 79	Firefox ≥ 78	Chrome ≥ 64	Safari ≥ 12

13.2 Element-plus入门案例

> 1 准备vite项目：

```
npm create vite  // 注意选择 vue+typeScript
npm install
npm install vue-router@4 --save
npm install pinia
npm install axios
```

2 安装element-plus：

```
npm install element-plus
```

3 完整引入element-plus：

- main.js

```
import { createApp } from 'vue'
//导入element-plus相关内容
import ElementPlus from 'element-plus'
import 'element-plus/dist/index.css'
import App from './App.vue'
const app = createApp(App)
app.use(ElementPlus)
app.mount('#app')
```

4 入门案例：

- App.vue

```
<script setup>
  import { ref } from 'vue'
  const value = ref(true)
</script>
<template>
  <div>
    <!-- 直接使用element-plus组件即可 -->
    <el-button>按钮</el-button>
    <br>
    <el-switch
      v-model="value"
      size="large"
      active-text="Open"
      inactive-text="Close"
    />
    <br />
    <el-switch v-model="value" active-text="Open" inactive-text="Close" />
    <br />
    <el-switch
      v-model="value"
      size="small"
      active-text="Open"
      inactive-text="Close"
    />
  </div>
</template>
<style scoped>
</style>
```

5 启动测试：

```
npm run dev
```

13.3 Element-plus常用组件

https://element-plus.gitee.io/zh-CN/component/button.html

第八章 微头条项目开发

一 项目简介

1.1 微头条业务简介

> 微头条新闻发布和浏览平台，主要包含业务如下：

- 用户功能
  - 注册功能
  - 登录功能
- 头条新闻
  - 新闻的分页浏览
  - 通过标题关键字搜索新闻
  - 查看新闻详情
  - 新闻的修改和删除
- 权限控制
  - 用户只能修改和自己发布的头条新闻

1.2 技术栈介绍

> 前端技术栈：

- ES6作为基础JS语法；
- nodejs用于运行环境；
- npm用于项目依赖管理工具；
- Vite用于项目的构建架工具；
- Vue3用于项目数据的渲染框架；
- Axios用于前后端数据的交互；
- Router用于页面的跳转；
- Pinia用于存储用户的数据；
- LocalStorage作为用户校验token的存储手段；
- Element-Plus提供组件；

> 后端技术栈：

- JAVA作为开发语言，版本为JDK17；
- Tomcat作为服务容器，版本为10.1.7；
- MySQL8用于项目存储数据；
- Servlet用于控制层实现前后端数据交互；
- JDBC用于实现数据的CURD；
- Druid用于提供数据源的连接池；
- MD5用于用户密码的加密；

- Jwt用于token的生成和校验；
- Jackson用于转换JSON；
- Filter用于用户登录校验和跨域处理；
- Lombok用于处理实体类；

1.3 功能展示

头条首页信息搜索：

登录功能：

注册功能：

权限控制功能：

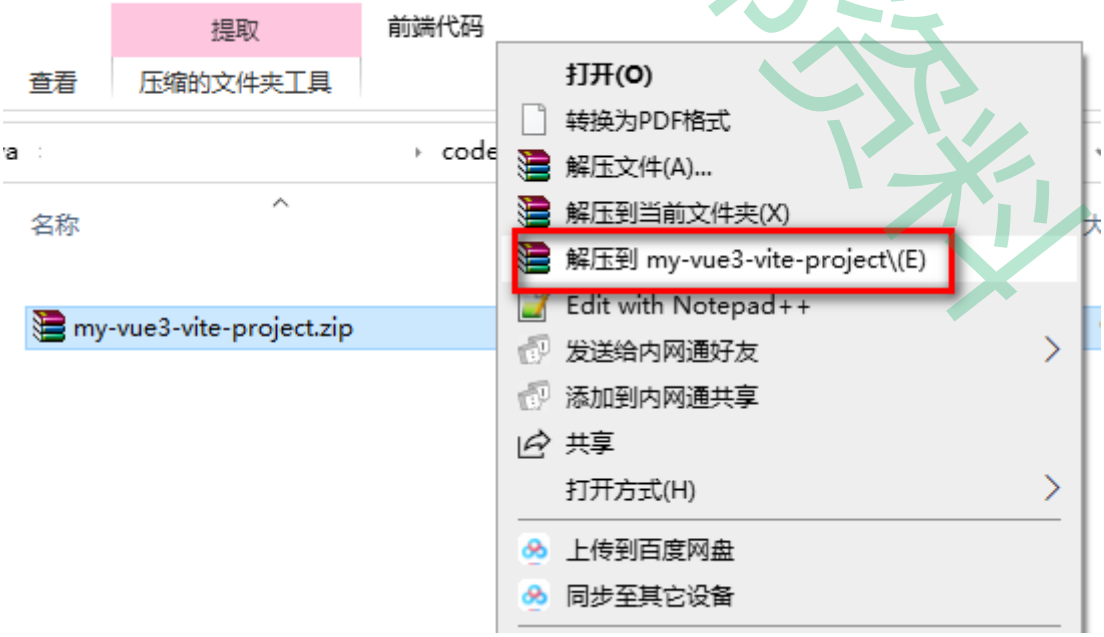
发布头条功能：

修改头条功能：

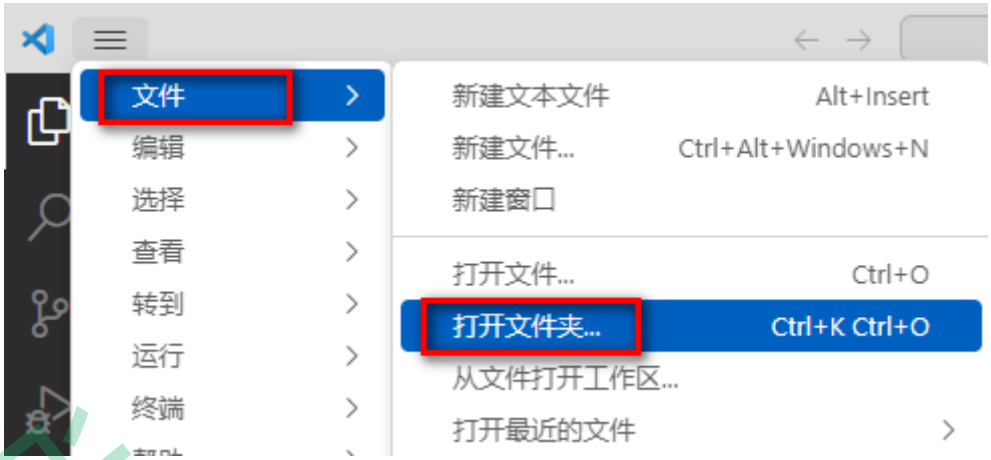
删除头条功能：

二 前端项目环境搭建

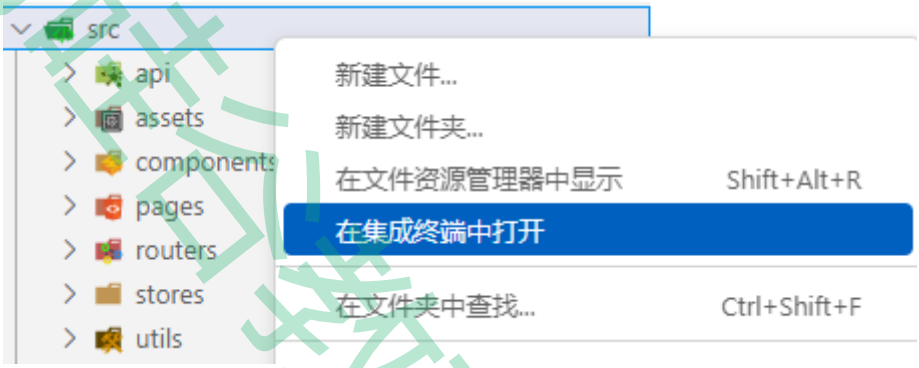
- 解压前端项目代码并存放到磁盘的合适位置



- 使用vscode打开工程



- 进入项目后打开集成终端或者在src上右击选择在集成终端中打开

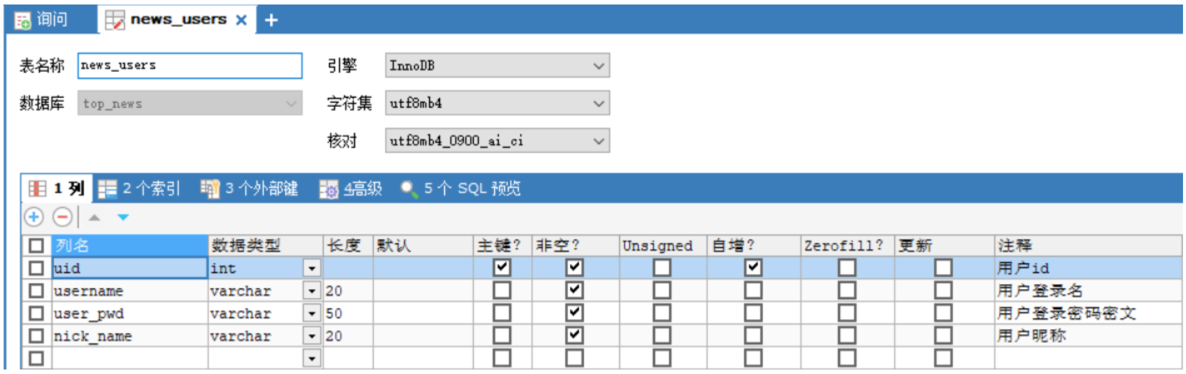


- 通过 npm run dev启动前端项目

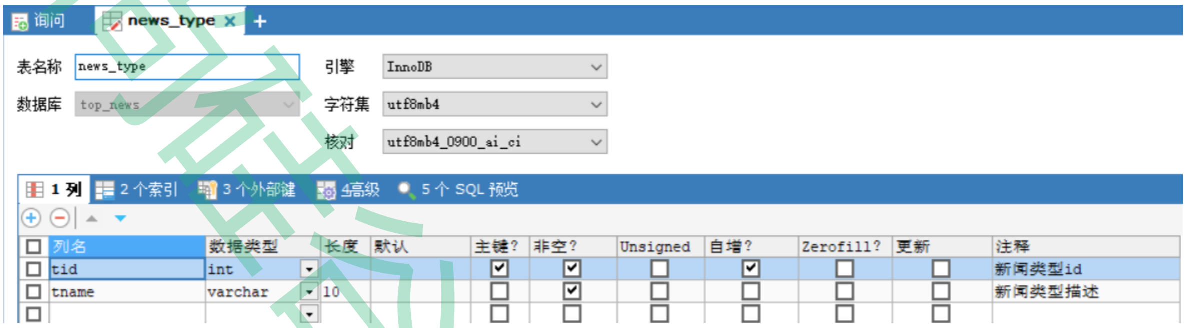
三 后端项目环境搭建

3.1 数据库准备

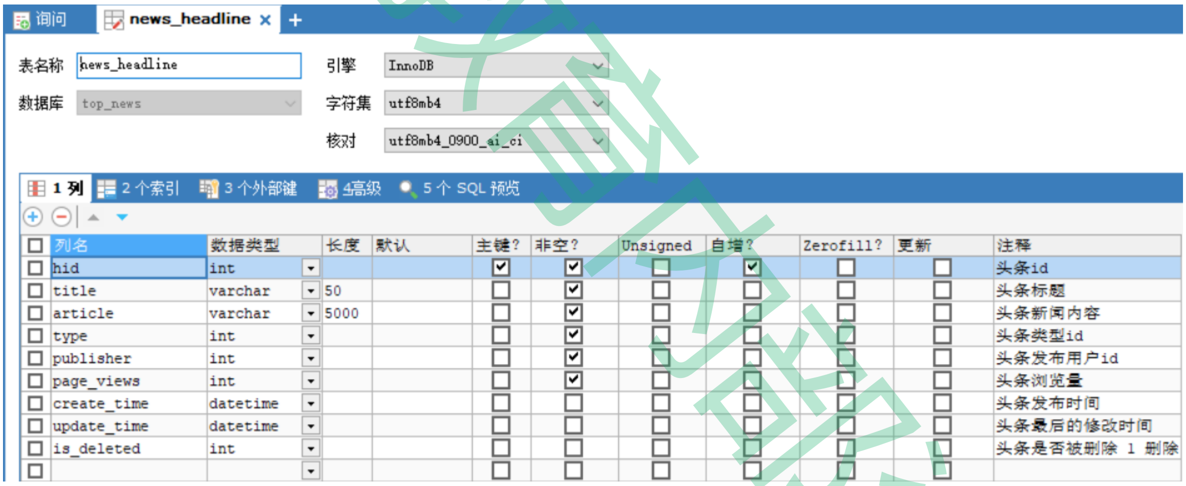
> `news_users` 用户表:



news_type 新闻类型表:



news_headline 新闻信息表:

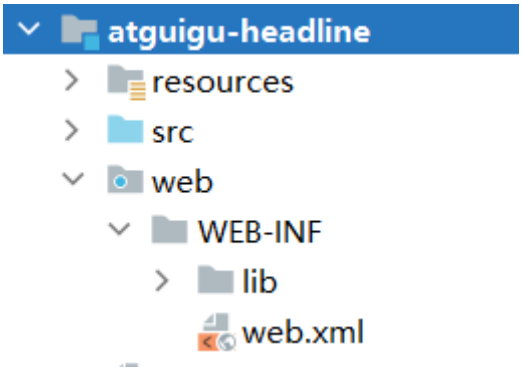


数据库创建SQL:

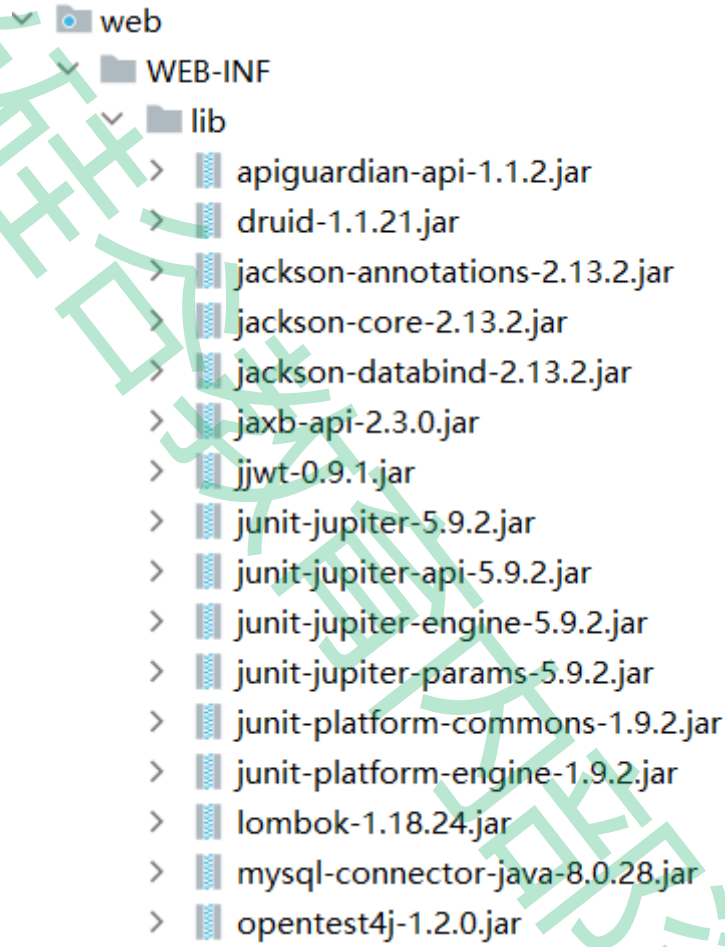
- 导入资料中的top_news.sql文件即可

3.2 搭建项目

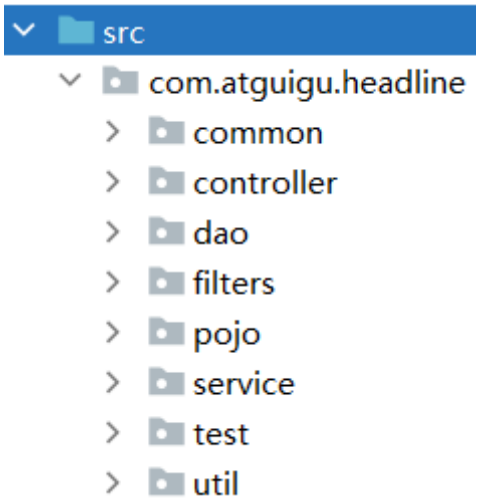
3.2.1 创建WEB项目



3.2.2 导入依赖



3.2.3 准备包结构



- controller 控制层代码,主要由Servlet组成;
- service 服务层代码,主要用于处理业务逻辑;
- dao 数据访问层,主要用户定义对于各个表格的CURD的方法;
- pojo 实体类层,主要用于存放和数据库对应的实体类以及一些VO对象;
- util 工具类包,主要用存放一些工具类;
- common 公共包,主要用户存放一些其他公共代码;
- filters 过滤器包,专门用于存放一些过滤器;
- test 测试代码包,专门用于定义一些测试的功能代码,上线前应该删掉,后期用maven可以自动处理掉;

3.3 准备工具类

3.3.1 异步响应规范格式类

- Result类

```
package com.atguigu.headline.common;
/**
 * 全局统一返回结果类
 */
public class Result<T> {
    // 返回码
    private Integer code;
    // 返回消息
    private String message;
    // 返回数据
    private T data;
    public Result(){}
    // 返回数据
    protected static <T> Result<T> build(T data) {
        Result<T> result = new Result<T>();
        if (data != null)
            result.setData(data);
        return result;
    }
    public static <T> Result<T> build(T body, Integer code, String message) {
        Result<T> result = build(body);
        result.setCode(code);
        result.setMessage(message);
        return result;
    }
    public static <T> Result<T> build(T body, ResultCodeEnum resultCodeEnum) {
        Result<T> result = build(body);
        result.setCode(resultCodeEnum.getCode());
        result.setMessage(resultCodeEnum.getMessage());
        return result;
    }
    /**
     * 操作成功
     * @param data  baseCategory1List
     * @param <T>
     * @return
     */
    public static<T> Result<T> ok(T data){
```

```
        Result<T> result = build(data);
        return build(data, ResultCodeEnum.SUCCESS);
    }
    public Result<T> message(String msg){
        this.setMessage(msg);
        return this;
    }
    public Result<T> code(Integer code){
        this.setCode(code);
        return this;
    }
    public Integer getCode() {
        return code;
    }
    public void setCode(Integer code) {
        this.code = code;
    }
    public String getMessage() {
        return message;
    }
    public void setMessage(String message) {
        this.message = message;
    }
    public T getData() {
        return data;
    }
    public void setData(T data) {
        this.data = data;
    }
}
```

- ResultCodeEnum 枚举类

```
package com.atguigu.headline.common;
/**
 * 统一返回结果状态信息类
 */
public enum ResultCodeEnum {

    SUCCESS(200,"success"),
    USERNAME_ERROR(501,"usernameError"),
    PASSWORD_ERROR(503,"passwordError"),
    NOTLOGIN(504,"notLogin"),
    USERNAME_USED(505,"userNameUsed")
    ;

    private Integer code;
    private String message;
    private ResultCodeEnum(Integer code, String message) {
        this.code = code;
        this.message = message;
    }
    public Integer getCode() {
        return code;
    }
    public String getMessage() {
        return message;
    }
```

```
}
```

3.3.2 MD5加密工具类

```
package com.atguigu.headline.util;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
public final class MD5Util {
    public static String encrypt(String strSrc) {
        try {
            char hexChars[] = { '0', '1', '2', '3', '4', '5', '6', '7', '8',
                    '9', 'a', 'b', 'c', 'd', 'e', 'f' };
            byte[] bytes = strSrc.getBytes();
            MessageDigest md = MessageDigest.getInstance("MD5");
            md.update(bytes);
            bytes = md.digest();
            int j = bytes.length;
            char[] chars = new char[j * 2];
            int k = 0;
            for (int i = 0; i < bytes.length; i++) {
                byte b = bytes[i];
                chars[k++] = hexChars[b >>> 4 & 0xf];
                chars[k++] = hexChars[b & 0xf];
            }
            return new String(chars);
        } catch (NoSuchAlgorithmException e) {
            e.printStackTrace();
            throw new RuntimeException("MD5加密出错！！+" + e);
        }
    }
}
```

3.3.3 JDBCUtil连接池工具类

```
package com.atguigu.headline.util;
import com.alibaba.druid.pool.DruidDataSourceFactory;
import javax.sql.DataSource;
import java.io.IOException;
import java.io.InputStream;
import java.sql.Connection;
import java.sql.SQLException;
import java.util.Properties;
public class JDBCUtil {
    private static ThreadLocal<Connection> threadLocal =new ThreadLocal<>();
    private static DataSource dataSource;
    // 初始化连接池
    static{
        // 可以帮助我们读取.properties配置文件
        Properties properties =new Properties();
        InputStream resourceAsStream =
JDBCUtil.class.getClassLoader().getResourceAsStream("jdbc.properties");
        try {
            properties.load(resourceAsStream);
            dataSource = DruidDataSourceFactory.createDataSource(properties);
        } catch (IOException e) {
```

```
            throw new RuntimeException(e);
        }
    }
    /*1 向外提供连接池的方法*/
    public static DataSource getDataSource(){
        return dataSource;
    }
    /*2 向外提供连接的方法*/
    public static Connection getConnection(){
        Connection connection = threadLocal.get();
        if (null == connection) {
            try {
                connection = dataSource.getConnection();
            } catch (SQLException e) {
                throw new RuntimeException(e);
            }
            threadLocal.set(connection);
        }

        return connection;
    }
    /*定义一个归还连接的方法 (解除和ThreadLocal之间的关联关系) */
    public static void releaseConnection(){
        Connection connection = threadLocal.get();
        if (null != connection) {
            threadLocal.remove();
            // 把连接设置回自动提交的连接
            try {
                connection.setAutoCommit(true);
                // 自动归还到连接池
                connection.close();
            } catch (SQLException e) {
                throw new RuntimeException(e);
            }
        }
    }
}
```

- 添加jdbc.properties配置文件

```
driverClassName=com.mysql.cj.jdbc.Driver
url=jdbc:mysql://localhost:3306/top_news
username=root
password=root
initialSize=5
maxActive=10
maxWait=1000
```

3.3.4 JwtHelper工具类

```
package com.atguigu.headline.util;
import com.alibaba.druid.util.StringUtils;
import io.jsonwebtoken.*;
import java.util.Date;
public class JwtHelper {
    private static long tokenExpiration = 24*60*60*1000;
    private static String tokenSignKey = "123456";
```

```
    //生成token字符串
    public static String createToken(Long userId) {
        String token = Jwts.builder()

                .setSubject("YYGH-USER")
                .setExpiration(new Date(System.currentTimeMillis() + tokenExpiration))
                .claim("userId", userId)
                .signWith(SignatureAlgorithm.HS512, tokenSignKey)
                .compressWith(CompressionCodecs.GZIP)
                .compact();
        return token;
    }
    //从token字符串获取userid
    public static Long getUserId(String token) {
        if(StringUtils.isEmpty(token)) return null;
        Jws<Claims> claimsJws =
Jwts.parser().setSigningKey(tokenSignKey).parseClaimsJws(token);
        Claims claims = claimsJws.getBody();
        Integer userId = (Integer)claims.get("userId");
        return userId.longValue();
    }
    //判断token是否有效
    public static boolean isExpiration(String token){
        try {
            boolean isExpire = Jwts.parser()
                    .setSigningKey(tokenSignKey)
                    .parseClaimsJws(token)
                    .getBody()
                    .getExpiration().before(new Date());
            //没有过期，有效，返回false
            return isExpire;
        }catch(Exception e) {
            //过期出现异常，返回true
            return true;
        }
    }
}
```

3.3.5 JSON转换的WEBUtil工具类

```
package com.atguigu.headline.util;
import com.atguigu.headline.common.Result;
import com.fasterxml.jackson.databind.ObjectMapper;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import java.io.BufferedReader;
import java.io.IOException;
import java.text.SimpleDateFormat;
public class WebUtil {
    private static ObjectMapper objectMapper;
    // 初始化objectMapper
    static{
        objectMapper=new ObjectMapper();
        // 设置JSON和Object转换时的时间日期格式
        objectMapper.setDateFormat(new SimpleDateFormat("yyyy-MM-dd HH:mm:ss"));
    }
```

```
    // 从请求中获取JSON串并转换为Object
    public static <T> T readJson(HttpServletRequest request,Class<T> clazz){
        T t =null;
        BufferedReader reader = null;
        try {
            reader = request.getReader();
            StringBuffer buffer =new StringBuffer();
            String line =null;
            while((line = reader.readLine())!= null){
                buffer.append(line);
            }
            t= objectMapper.readValue(buffer.toString(),clazz);
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
        return t;
    }
    // 将Result对象转换成JSON串并放入响应对象
    public static void writeJson(HttpServletResponse response, Result result){
        response.setContentType("application/json;charset=UTF-8");
        try {
            String json = objectMapper.writeValueAsString(result);
            response.getWriter().write(json);
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
}
```

3.4 准备各层的接口和实现类

3.4.1 准备实体类和VO对象

> NewsUser:

```
package com.atguigu.headline.pojo;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import java.io.Serializable;
@AllArgsConstructor
@NoArgsConstructor
@Data
public class NewsUser implements Serializable {
    private Integer uid;
    private String username;
    private String userPwd;
    private String nickName;
}
```

> NewsType:

```
package com.atguigu.headline.pojo;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import java.io.Serializable;
@AllArgsConstructor
@NoArgsConstructor
@Data
public class NewsType implements Serializable {
    private Integer tid;
    private String tname;
}
```

NewsHeadline：

```
package com.atguigu.headline.pojo;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import java.io.Serializable;
import java.util.Date;
@AllArgsConstructor
@NoArgsConstructor
@Data
public class NewsHeadline implements Serializable {
    private Integer hid;
    private String title;
    private String article;
    private Integer type;
    private Integer publisher;
    private Integer pageViews;
    private Date createTime;
    private Date updateTime;
    private Integer isDeleted;
}
```

HeadlineQueryVo：

```
package com.atguigu.headline.pojo.vo;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import java.io.Serializable;
@AllArgsConstructor
@NoArgsConstructor
@Data
public class HeadlineQueryVo implements Serializable {
    private String keyWords;
    private Integer type ;
    private Integer pageNum;
    private Integer pageSize;
}
```

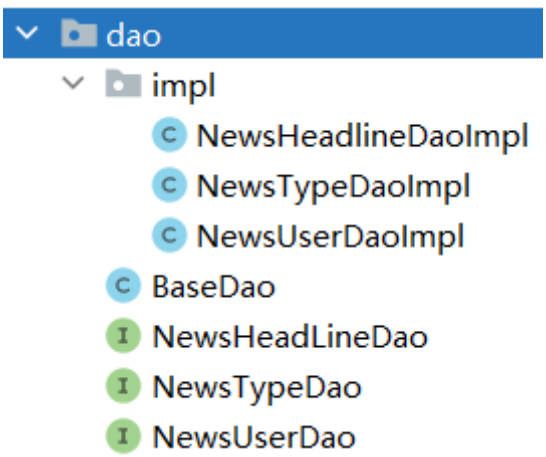
HeadlinePageVo：

```
package com.atguigu.headline.pojo.vo;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import java.io.Serializable;
@AllArgsConstructor
@NoArgsConstructor
@Data
public class HeadlinePageVo implements Serializable {
    private Integer hid;
    private String title;
    private Integer type;
    private Integer pageViews;
    private Long pastHours;
    private Integer publisher;
}
```

HeadlineDetailVo:

```
package com.atguigu.headline.pojo.vo;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import java.io.Serializable;
@AllArgsConstructor
@NoArgsConstructor
@Data
public class HeadlineDetailVo implements Serializable {
    private Integer hid;
    private String title;
    private String article;
    private Integer type;
    private String typeName;
    private Integer pageViews;
    private Long pastHours;
    private Integer publisher;
    private String author;
}
```

3.4.2 DAO层接口和实现类



BaseDao基础类,封装了公共的查询方法和公共的增删改方法:

- 注意,所有的Dao接口的实现类都要继承BaseDao

```
package com.atguigu.headline.dao;
import com.atguigu.headline.util.JDBCUtil;
import java.lang.reflect.Field;
import java.sql.*;
import java.time.LocalDateTime;
import java.util.ArrayList;
import java.util.List;
public class BaseDao {
    // 公共的查询方法  返回的是单个对象
    public <T> T baseQueryObject(Class<T> clazz, String sql, Object ... args) {
        T t = null;
        Connection connection = JDBCUtil.getConnection();
        PreparedStatement preparedStatement = null;
        ResultSet resultSet = null;
        int rows = 0;
        try {
            // 准备语句对象
            preparedStatement = connection.prepareStatement(sql);
            // 设置语句上的参数
            for (int i = 0; i < args.length; i++) {
                preparedStatement.setObject(i + 1, args[i]);
            }

            // 执行 查询
            resultSet = preparedStatement.executeQuery();
            if (resultSet.next()) {
                t = (T) resultSet.getObject(1);
            }
        } catch (Exception e) {
            throw new RuntimeException(e);
        } finally {
            if (null != resultSet) {
                try {
                    resultSet.close();
                } catch (SQLException e) {
                    throw new RuntimeException(e);
                }
            }
            if (null != preparedStatement) {
                try {
                    preparedStatement.close();
                } catch (SQLException e) {
                    throw new RuntimeException(e);
                }

            }
            JDBCUtil.releaseConnection();
        }
        return t;
    }
    // 公共的查询方法  返回的是对象的集合

    public <T> List<T> baseQuery(Class clazz, String sql, Object ... args){
        List<T> list =new ArrayList<>();
        Connection connection = JDBCUtil.getConnection();
        PreparedStatement preparedStatement=null;
```

```
        ResultSet resultSet =null;
        int rows = 0;
        try {
            // 准备语句对象
            preparedStatement = connection.prepareStatement(sql);
            // 设置语句上的参数
            for (int i = 0; i < args.length; i++) {
                preparedStatement.setObject(i+1,args[i]);
            }

            // 执行 查询
            resultSet = preparedStatement.executeQuery();

            ResultSetMetaData metaData = resultSet.getMetaData();
            int columnCount = metaData.getColumnCount();

            // 将结果集通过反射封装成实体类对象
            while (resultSet.next()) {
                // 使用反射实例化对象
                Object obj =clazz.getDeclaredConstructor().newInstance();

                for (int i = 1; i <= columnCount; i++) {
                    String columnName = metaData.getColumnLabel(i);
                    Object value = resultSet.getObject(columnName);
                    // 处理datetime类型字段和java.util.Data转换问题
                    if(value.getClass().equals(LocalDateTime.class)){
                        value= Timestamp.valueOf((LocalDateTime) value);
                    }
                    Field field = clazz.getDeclaredField(columnName);
                    field.setAccessible(true);
                    field.set(obj,value);
                }

                list.add((T)obj);
            }

        } catch (Exception e) {
            throw new RuntimeException(e);
        } finally {
            if (null !=resultSet) {
                try {
                    resultSet.close();
                } catch (SQLException e) {
                    throw new RuntimeException(e);
                }
            }
            if (null != preparedStatement) {
                try {
                    preparedStatement.close();
                } catch (SQLException e) {
                    throw new RuntimeException(e);
                }
            }
            JDBCUtil.releaseConnection();
        }
        return list;
    }
    // 通用的增删改方法
```

```
    public int baseUpdate(String sql,Object ... args) {
        // 获取连接
        Connection connection = JDBCUtil.getConnection();
        PreparedStatement preparedStatement=null;
        int rows = 0;
        try {
            // 准备语句对象
            preparedStatement = connection.prepareStatement(sql);
            // 设置语句上的参数
            for (int i = 0; i < args.length; i++) {
                preparedStatement.setObject(i+1,args[i]);
            }

            // 执行 增删改 executeUpdate
            rows = preparedStatement.executeUpdate();
            // 释放资源(可选)
        } catch (SQLException e) {
            throw new RuntimeException(e);
        } finally {
            if (null != preparedStatement) {
                try {
                    preparedStatement.close();
                } catch (SQLException e) {
                    throw new RuntimeException(e);
                }
            }
            JDBCUtil.releaseConnection();
        }
        // 返回的是影响数据库记录数
        return rows;
    }
}
```

Dao层的所有接口：

```
package com.atguigu.headline.dao;
public interface NewsHeadLineDao {
}
```

```
package com.atguigu.headline.dao;
public interface NewsTypeDao {
}
```

```
package com.atguigu.headline.dao;
public interface NewsUserDao {
}
```

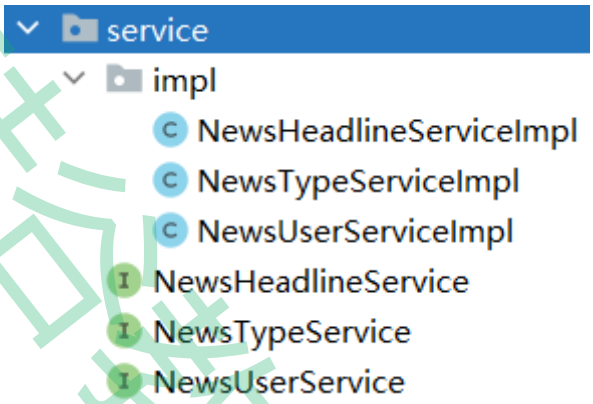
Dao层所有实现类：

```
package com.atguigu.headline.dao.impl;
import com.atguigu.headline.dao.BaseDao;
import com.atguigu.headline.dao.NewsHeadLineDao;
public class NewsHeadlineDaoImpl extends BaseDao implements NewsHeadLineDao{
}
```

```
package com.atguigu.headline.dao.impl;
import com.atguigu.headline.dao.BaseDao;
import com.atguigu.headline.dao.NewsTypeDao;
public class NewsTypeDaoImpl extends BaseDao implements NewsTypeDao{
}
```

```
package com.atguigu.headline.dao.impl;
import com.atguigu.headline.dao.BaseDao;
import com.atguigu.headline.dao.NewsUserDao;
public class NewsUserDaoImpl extends BaseDao implements NewsUserDao{
}
```

3.4.3 Service层接口和实现类



> Service层所有接口:

```
package com.atguigu.headline.service;
public interface NewsHeadlineService {
}
```

```
package com.atguigu.headline.service;
public interface NewsTypeService {
    List<NewsType> findAll();
}
```

```
package com.atguigu.headline.service;
public interface NewsUserService {
}
```

> Service层所有实现类:

```
package com.atguigu.headline.service.impl;
import com.atguigu.headline.service.NewsHeadlineService;
public class NewsHeadlineServiceImpl  implements NewsHeadlineService {
}
```

```
package com.atguigu.headline.service.impl;
import com.atguigu.headline.service.NewsTypeService;
public class NewsTypeServiceImpl implements NewsTypeService {
}
```

```
package com.atguigu.headline.service.impl;
import com.atguigu.headline.service.NewsUserService;
public class NewsUserServiceImpl implements NewsUserService {
}
```

3.4.4 Controller层接口和实现类

> BaseController 用于将路径关联到处理方法的基础控制器：

- 所有的Controller都要继承该类

```
package com.atguigu.headline.controller;
import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.lang.reflect.Method;
public class BaseController extends HttpServlet {
    @Override
    protected void service(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        // 响应的MIME类型和乱码问题
        resp.setContentType("application/json;charset=UTF-8");

        String requestURI = req.getRequestURI();
        String[] split = requestURI.split("/");
        String methodName =split[split.length-1];
        // 通过反射获取要执行的方法
        Class clazz = this.getClass();
        try {
            Method
method=clazz.getDeclaredMethod(methodName,HttpServletRequest.class,HttpServletResponse
.class);
            // 设置方法可以访问
            method.setAccessible(true);
            // 通过反射执行代码
            method.invoke(this,req,resp);
        } catch (Exception e) {
            e.printStackTrace();
            throw new RuntimeException(e.getMessage());
        }
    }
}
```

> 所有的Controller类：

```
package com.atguigu.headline.controller;
import jakarta.servlet.annotation.WebServlet;
@WebServlet("/headline/*")
public class NewsHeadlineController extends BaseController {
}
```

```
package com.atguigu.headline.controller;
import jakarta.servlet.annotation.WebServlet;
@WebServlet("/type/*")
public class NewsTypeController {
}
```

```
package com.atguigu.headline.controller;
import jakarta.servlet.annotation.WebServlet;
@WebServlet("/user/*")
public class NewsUserController extends BaseController{
}
```

```
package com.atguigu.headline.controller;
import jakarta.servlet.annotation.WebServlet;
@WebServlet("/portal/*")
public class PortalController extends BaseController{
}
```

3.5 开发跨域CORS过滤器

> CrosFilter过滤器：

```
package com.atguigu.headline.filters;
import jakarta.servlet.*;
import jakarta.servlet.annotation.WebFilter;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import java.io.IOException;
@WebFilter("/*")
public class CrosFilter implements Filter {
    @Override
    public void doFilter(ServletRequest servletRequest, ServletResponse
servletResponse, FilterChain filterChain) throws IOException, ServletException {
        HttpServletResponse response = (HttpServletResponse) servletResponse;
        HttpServletRequest request =(HttpServletRequest) servletRequest;
        response.setHeader("Access-Control-Allow-Origin", "*");
        response.setHeader("Access-Control-Allow-Methods", "POST, GET, OPTIONS,
DELETE, HEAD");
        response.setHeader("Access-Control-Max-Age", "3600");
        response.setHeader("Access-Control-Allow-Headers", "access-control-allow-
origin, authority, content-type, version-info, X-Requested-With");
        // 非预检请求,放行即可,预检请求,则到此结束,不需要放行
        if(!request.getMethod().equalsIgnoreCase("OPTIONS")){
            filterChain.doFilter(servletRequest, servletResponse);
        }
    }
}
```

- 未来我们使用框架,直接用一个@CrossOrigin 就可以解决跨域问题了。

四 PostMan测试工具

4.1 什么是PostMan

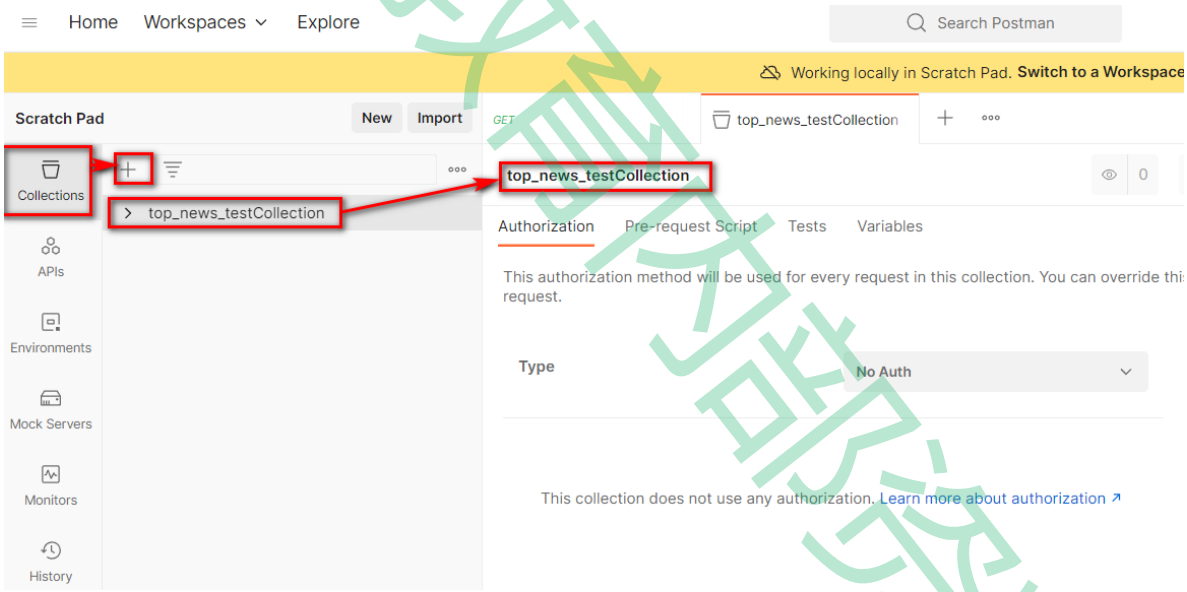
- Postman是一个 接口测试工具 ，在做接口测试的时候，Postman相当于一个客户端，它可以模拟用户发起的各类HTTP请求，将请求数据发送至服务端,获取对应的响应结果,，从而验证响应中的结果数据是否和预期值相匹配，并确保开发人员能够及时处理接口中的bug，进而保证产品上线之后的稳定性和安全性。 它主要是用来模拟各种HTTP请求的(如:GET/POST/DELETE/PUT..等等，Postman与浏览器的区别在于有的浏览器不能输出Json格式，而Postman更直观接口返回的结果。

4.2 怎么安装PostMan

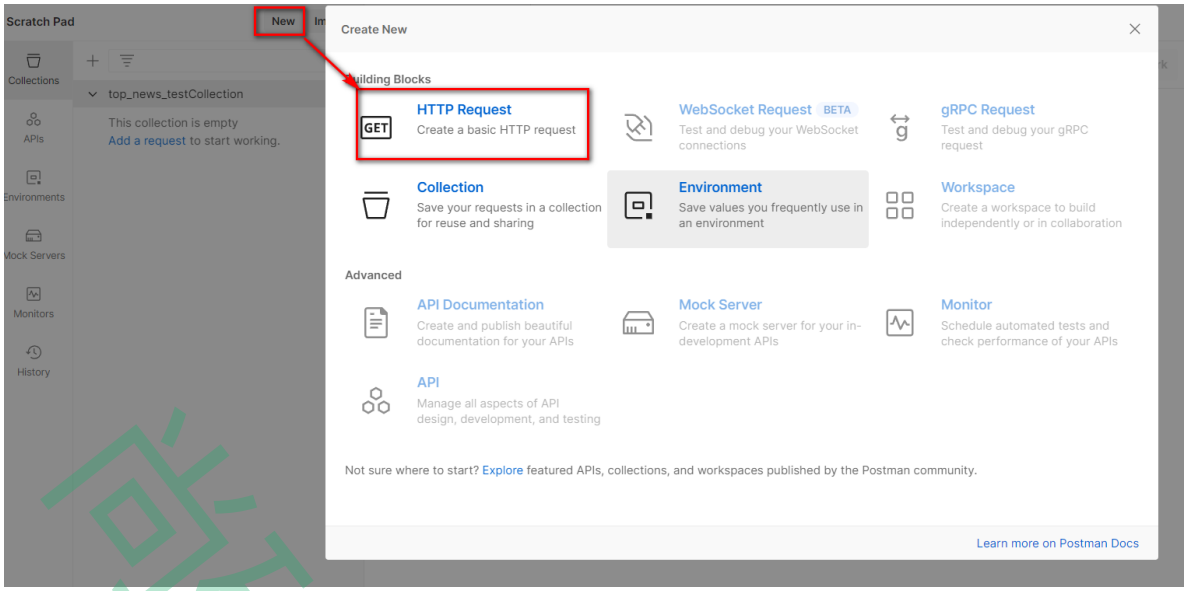
- 官网下载地址: https://www.getpostman.com ,或者使用资料中提供的安装包;
- 安装过程简单,一路next即可;
- 第一次启动postman 会要求输入用户名和密码，如果没有的话，关闭，再次启动就可以直接进入了;

4.3 怎么使用PostMan

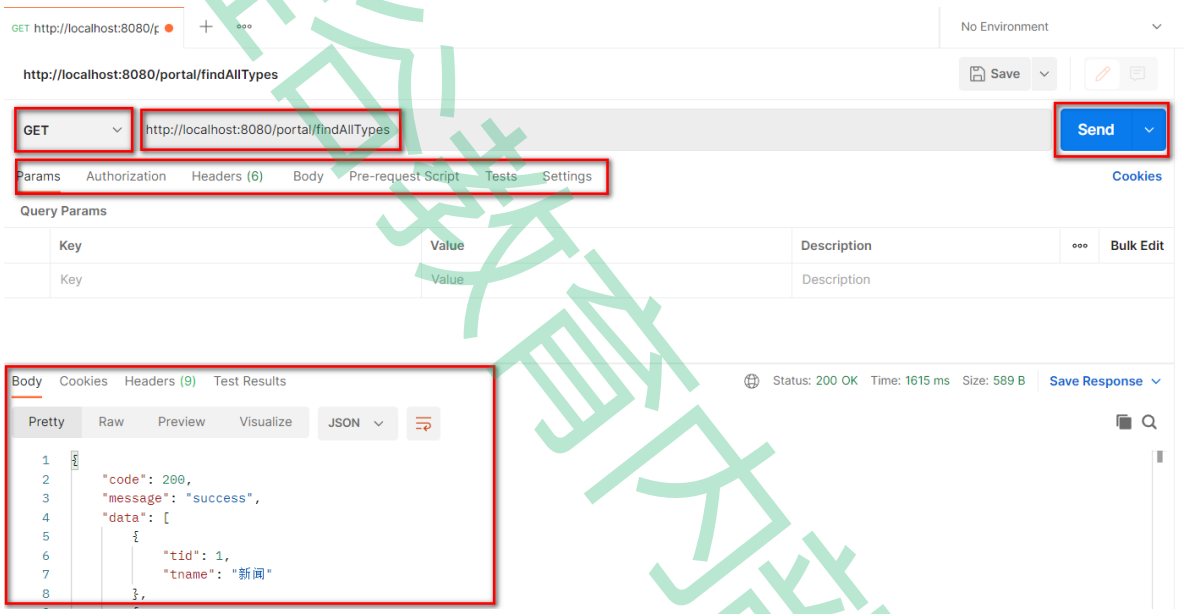
> 启动PostMan后，创建一个collection，在该collection下专门存放和微头条项目相关的测试:



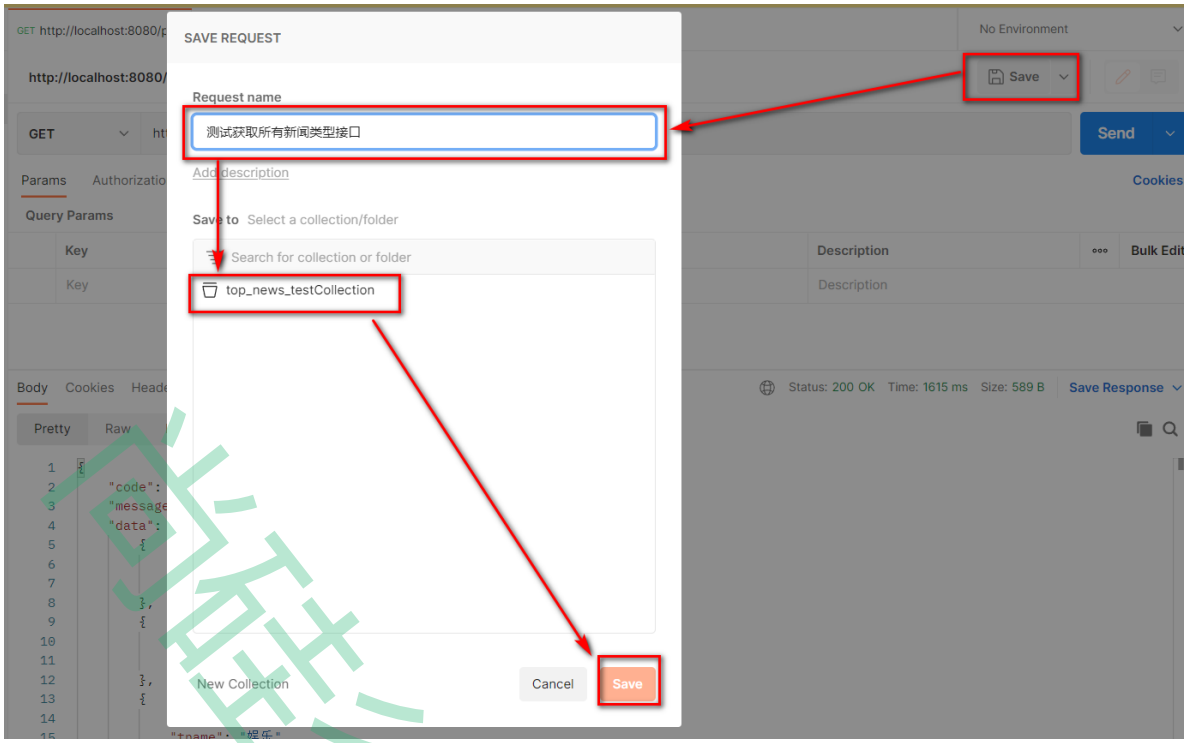
> 创建完毕后，增加新的接口测试:



填写要测试的接口相关的路径、参数、请求体内容等信息：



测试完毕后，可以选择将该接口的测试进行保存，方便后续随时再次测试：



五 登录注册功能

5.1 登录表单提交

> 需求描述：用户在客户端输入用户名密码并向后端提交，后端根据用户名和密码判断登录是否成功，用户有误或者密码有误响应不同的提示信息。

> uri：user/login

> 请求方式：POST

请求参数：

```
{
    "username":"zhangsan", //用户名
    "userPwd":"123456"     //明文密码
}
```

响应示例：

- 登录成功

```
{
    "code":"200",          // 成功状态码
    "message":"success"    // 成功状态描述
    "data":{
        "token":"... ..." // 用户id的token
    }
}
```

- 用户名有误

```
{
    "code":"501",
    "message":"用户名有误"
    "data":{}
}
```

- 密码有误

```
{
    "code":"503",
    "message":"密码有误"
    "data":{}
}
```

后端代码：

- NewsUserController

```
package com.atguigu.headline.controller;
import com.atguigu.headline.common.Result;
import com.atguigu.headline.common.ResultCodeEnum;
import com.atguigu.headline.pojo.NewsHeadline;
import com.atguigu.headline.pojo.NewsUser;
import com.atguigu.headline.service.NewsUserService;
import com.atguigu.headline.service.impl.NewsUserServiceImpl;
import com.atguigu.headline.util.JwtHelper;
import com.atguigu.headline.util.MD5Util;
import com.atguigu.headline.util.WebUtil;
import jakarta.servlet.ServletException;
import jakarta.servlet.annotation.WebServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.util.HashMap;
import java.util.Map;
```

```
@WebServlet("/user/*")
public class NewsUserController extends BaseController{
    private NewsUserService newsUserService =new NewsUserServiceImpl();
     /**
     * 登录接口
     * @param req
     * @param resp
     * @throws ServletException
     * @throws IOException
     */
    protected void login(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        NewsUser newsUser = WebUtil.readJson(req, NewsUser.class);

        Result result =null;
        NewsUser loginNewsUser
=newsUserService.findByUserName(newsUser.getUsername());
        // 判断用户名
        if (null != loginNewsUser) {
            // 判断密码

 if(loginNewsUser.getUserPwd().equals(MD5Util.encrypt(newsUser.getUserPwd()))){
                // 密码正确
                Map<String,Object> data =new HashMap<>();
                // 生成token口令
                String token =
JwtHelper.createToken(loginNewsUser.getUid().longValue());
                // 封装数据map
                data.put("token",token);
                // 封装结果
                result=Result.ok(data);
            }else{
                // 封装密码错误结果
                result=Result.build(null, ResultCodeEnum.PASSWORD_ERROR);
            }
        }else{
            // 封装用户名错误结果
            result=Result.build(null, ResultCodeEnum.USERNAME_ERROR);
        }
        // 响应结果
        WebUtil.writeJson(resp,result);
    }
}
```

- NewsUserService

```
package com.atguigu.headline.service;
import com.atguigu.headline.pojo.NewsUser;
public interface NewsUserService {
    /**
     * 根据用户名,获得查询用户的方法
     * @param username 要查询的用户名
     * @return 如果找到返回NewsUser对象,找不到返回null
     */
    NewsUser findByUserName(String username);
}
```

- NewsUserServiceImpl

```
package com.atguigu.headline.service.impl;
import com.atguigu.headline.dao.NewsUserDao;
import com.atguigu.headline.dao.impl.NewsUserDaoImpl;
import com.atguigu.headline.pojo.NewsUser;
import com.atguigu.headline.service.NewsTypeService;
import com.atguigu.headline.service.NewsUserService;
import com.atguigu.headline.util.MD5Util;
public class NewsUserServiceImpl implements NewsUserService {
    private NewsUserDao newsUserDao =new NewsUserDaoImpl();
    @Override
    public NewsUser findByUserName(String username) {
        return newsUserDao.findByUserName(username);
    }
}
```

- NewUserDao

```
package com.atguigu.headline.dao;
import com.atguigu.headline.pojo.NewsUser;
public interface NewsUserDao {
    /**
     * 根据用户名查询用户信息
     * @param username 要查询的用户名
     * @return 找到返回NewsUser对象,找不到返回null
     */
    NewsUser findByUserName(String username);
}
```

- NewsUserDaoImpl

```
package com.atguigu.headline.dao.impl;
import com.atguigu.headline.dao.BaseDao;
import com.atguigu.headline.dao.NewsUserDao;
import com.atguigu.headline.pojo.NewsUser;
import java.util.List;
public class NewsUserDaoImpl extends BaseDao implements NewsUserDao {
    @Override
    public NewsUser findByUserName(String username) {
        // 准备SQL
        String sql ="select uid,username,user_pwd userPwd ,nick_name nickName from
news_user where username = ?";
        // 调用BaseDao公共查询方法
        List<NewsUser> newsUserList = baseQuery(NewsUser.class, sql, username);
        // 如果找到,返回集合中的第一个数据(其实就一个)
        if (null != newsUserList && newsUserList.size()>0){
            return  newsUserList.get(0);
        }
        return null;
    }
}
```

5.2 根据token获取完整用户信息

> 需求描述：客户端发送请求，提交token请求头，后端根据token请求头获取登录用户的详细信息，并响应给客户端进行存储。

uri：user/getUserInfo

请求方式：GET

请求头：token: ... ...

响应示例：

- 成功获取

```
{
    "code": 200,
    "message": "success",
    "data": {
        "loginUser": {
            "uid": 1,
            "username": "zhangsan",
            "userPwd": "",
            "nickName": "张三"
        }
    }
}
```

- 获取失败

```
{
    "code": 504,
    "message": "notLogin",
    "data": null
}
```

后端代码：

- NewsUserController

```
package com.atguigu.headline.controller;
import com.atguigu.headline.common.Result;
import com.atguigu.headline.common.ResultCodeEnum;
import com.atguigu.headline.pojo.NewsHeadline;
import com.atguigu.headline.pojo.NewsUser;
import com.atguigu.headline.service.NewsUserService;
import com.atguigu.headline.service.impl.NewsUserServiceImpl;
import com.atguigu.headline.util.JwtHelper;
import com.atguigu.headline.util.MD5Util;
import com.atguigu.headline.util.WebUtil;
import jakarta.servlet.ServletException;
import jakarta.servlet.annotation.WebServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.util.HashMap;
import java.util.Map;
@WebServlet("/user/*")
```

```
public class NewsUserController extends BaseController{
    private NewsUserService newsUserService =new NewsUserServiceImpl();
    /**
     * 接收token,根据token查询完整用户信息
     * @param req
     * @param resp
     * @throws ServletException
     * @throws IOException
     */
    protected void getUserInfo(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
        String token = req.getHeader("token");
        Result  result =Result.build(null,ResultCodeEnum.NOTLOGIN);
        if(null!= token){
            if (!JwtHelper.isExpiration(token)) {
                Integer uid = JwtHelper.getUserId(token).intValue();
                NewsUser newsUser =newsUserService.findByUid(uid);
                newsUser.setUserPwd("");
                Map<String,Object> data =new HashMap<>();
                data.put("loginUser",newsUser);
                result=Result.ok(data);
            }
        }
        WebUtil.writeJson(resp,result);
    }
}
```

- NewsUserService

```
package com.atguigu.headline.service;
import com.atguigu.headline.pojo.NewsUser;
public interface NewsUserService {
    /**
     * 根据用户id查询用户信息
     * @param uid 要查询的用户id
     * @return 找到返回NewsUser对象,找不到返回null
     */
    NewsUser findByUid(Integer uid);
}
```

- NewsUserServiceImpl

```
package com.atguigu.headline.service.impl;
import com.atguigu.headline.dao.NewsUserDao;
import com.atguigu.headline.dao.impl.NewsUserDaoImpl;
import com.atguigu.headline.pojo.NewsUser;
import com.atguigu.headline.service.NewsTypeService;
import com.atguigu.headline.service.NewsUserService;
import com.atguigu.headline.util.MD5Util;
public class NewsUserServiceImpl implements NewsUserService {
    private NewsUserDao newsUserDao =new NewsUserDaoImpl();
    @Override
    public NewsUser findByUid(Integer uid) {
        return newsUserDao.findByUid(uid);
    }
}
```

- NewUserDao

```
package com.atguigu.headline.dao;
import com.atguigu.headline.pojo.NewsUser;
public interface NewsUserDao {
    /**
     * 根据用户id连接数据库查询用户信息
     * @param uid  要查询的用户id
     * @return 找到返回NewsUser对象,找不到返回null
     */
    NewsUser findByUid(Integer uid);
}
```

- NewUserDaoImpl

```
package com.atguigu.headline.dao.impl;
import com.atguigu.headline.dao.BaseDao;
import com.atguigu.headline.dao.NewsUserDao;
import com.atguigu.headline.pojo.NewsUser;
import java.util.List;
public class NewsUserDaoImpl extends BaseDao implements NewsUserDao {
    @Override
    public NewsUser findByUid(Integer uid) {
        String sql ="select uid,username,user_pwd userPwd ,nick_name nickName from
news_user where uid = ?";
        List<NewsUser> newsUserList = baseQuery(NewsUser.class, sql, uid);
        if (null != newsUserList && newsUserList.size()>0){
            return  newsUserList.get(0);
        }
        return null;
    }
}
```

5.3 注册时用户名占用校验

> 需求说明：用户在注册时输入用户名时，立刻将用户名发送给后端，后端根据用户名查询用户名是否可用并做出响应。

> uri：user/checkUserName

> 请求方式：POST

> 请求参数：username=zhangsan

> 响应示例：

- 用户名校验通过

```
{
    "code":"200",
    "message":"success"
    "data":{}
}
```

- 用户名占用

```
{
    "code":"505",
    "message":"用户名占用"
    "data":{}
}
```

> 后端代码：

- NewsUserController

```
package com.atguigu.headline.controller;
import com.atguigu.headline.common.Result;
import com.atguigu.headline.common.ResultCodeEnum;
import com.atguigu.headline.pojo.NewsHeadline;
import com.atguigu.headline.pojo.NewsUser;
import com.atguigu.headline.service.NewsUserService;
import com.atguigu.headline.service.impl.NewsUserServiceImpl;
import com.atguigu.headline.util.JwtHelper;
import com.atguigu.headline.util.MD5Util;
import com.atguigu.headline.util.WebUtil;
import jakarta.servlet.ServletException;
import jakarta.servlet.annotation.WebServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.util.HashMap;
import java.util.Map;
@WebServlet("/user/*")
public class NewsUserController extends BaseController{
    private NewsUserService newsUserService =new NewsUserServiceImpl();
    /**
     * 注册时校验用户名是否被占用
     * @param req
     * @param resp
     * @throws ServletException
     * @throws IOException
```

```
     */
    protected void checkUserName(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
        String username = req.getParameter("username");
        NewsUser newsUser = newsUserService.findByUserName(username);
        Result result=null;
        if (null == newsUser){
            result=Result.ok(null);
        }else{
            result=Result.build(null,ResultCodeEnum.USERNAME_USED);
        }
        WebUtil.writeJson(resp,result);
    }
}
```

5.4 注册表单提交

> 需求说明：客户端将新用户信息发送给服务端，服务端将新用户存入数据库，存入之前做用户名是否被占用校验，校验通过响应成功提示，否则响应失败提示。

> uri：user/regist

> 请求方式：POST

> 请求参数：

```
{
    "username":"zhangsan",
    "userPwd":"123456",
    "nickName":"张三"
}
```

响应示例：

- 注册成功

```
{
    "code":"200",
    "message":"success"
    "data":{}
}
```

- 用户名占用

```
{
    "code":"505",
    "message":"用户名占用"
    "data":{}
}
```

后端代码：

- NewsUserController

```
package com.atguigu.headline.controller;

import com.atguigu.headline.common.Result;
import com.atguigu.headline.common.ResultCodeEnum;
import com.atguigu.headline.pojo.NewsHeadline;
import com.atguigu.headline.pojo.NewsUser;
import com.atguigu.headline.service.NewsUserService;
import com.atguigu.headline.service.impl.NewsUserServiceImpl;
import com.atguigu.headline.util.JwtHelper;
import com.atguigu.headline.util.MD5Util;
import com.atguigu.headline.util.WebUtil;
import jakarta.servlet.ServletException;
import jakarta.servlet.annotation.WebServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.util.HashMap;
import java.util.Map;


@WebServlet("/user/*")
public class NewsUserController extends BaseController{
    private NewsUserService newsUserService =new NewsUserServiceImpl();
    /**
     * 注册功能接口
     * @param req
     * @param resp
     * @throws ServletException
     * @throws IOException
     */
    protected void regist(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        NewsUser newsUser = WebUtil.readJson(req, NewsUser.class);
        NewsUser usedUser = newsUserService.findByUserName(newsUser.getUsername());
```

```
        Result result=null;
        if (null == usedUser){
            newsUserService.registUser(newsUser);
            result=Result.ok(null);
        }else{
            result=Result.build(null,ResultCodeEnum.USERNAME_USED);
        }
        WebUtil.writeJson(resp,result);
    }
}
```

- NewsUserService

```
package com.atguigu.headline.service;
import com.atguigu.headline.pojo.NewsUser;
public interface NewsUserService {
    /**
     * 注册用户信息,注册成功返回大于0的整数,失败返回0
     * @param newsUser
     * @return
     */
    int registUser(NewsUser newsUser);
}
```

- NewsUserServiceImpl

```
package com.atguigu.headline.service.impl;
import com.atguigu.headline.dao.NewsUserDao;
import com.atguigu.headline.dao.impl.NewsUserDaoImpl;
import com.atguigu.headline.pojo.NewsUser;
import com.atguigu.headline.service.NewsTypeService;
import com.atguigu.headline.service.NewsUserService;
import com.atguigu.headline.util.MD5Util;
public class NewsUserServiceImpl implements NewsUserService {
    @Override
    public int registUser(NewsUser newsUser) {
        // 密码明文转密文
        newsUser.setUserPwd(MD5Util.encrypt(newsUser.getUserPwd()));
        // 存入数据库
        return newsUserDao.insertNewsUser(newsUser);
    }
}
```

- NewUserDao

```
package com.atguigu.headline.dao;
import com.atguigu.headline.pojo.NewsUser;
public interface NewsUserDao {
    /**
     * 将用户信息存入数据库
     * @param newsUser
     * @return
     */
    int insertNewsUser(NewsUser newsUser);
}
```

- NewUserDaoImpl

```
package com.atguigu.headline.dao.impl;
import com.atguigu.headline.dao.BaseDao;
import com.atguigu.headline.dao.NewsUserDao;
import com.atguigu.headline.pojo.NewsUser;
import java.util.List;
public class NewsUserDaoImpl extends BaseDao implements NewsUserDao {
    @Override
    public int insertNewsUser(NewsUser newsUser) {
        String sql ="insert into news_user values(DEFAULT,?,?,?)";
        return
baseUpdate(sql,newsUser.getUsername(),newsUser.getUserPwd(),newsUser.getNickName());
    }
}
```

六 头条首页功能

6.1 查询所有头条分类

需求说明：进入新闻首页，查询所有分类并动态展示新闻类别栏位

uri：portal/findAllTypes

请求方式：GET

请求参数：无

响应示例：

```
{
    "code":"200",
    "message":"OK"
    "data":
            [
                {
                    "tid":"1",
                    "tname":"新闻"
                },
                {
                    "tid":"2",
                    "tname":"体育"
                },
                {
                    "tid":"3",
                    "tname":"娱乐"
                },
                {
                    "tid":"4",
```

```
                "tname":"科技"
            },
            {
                "tid":"5",
                "tname":"其他"
            }
        ]
}
```

> 后端代码:

- PortalController

```java
package com.atguigu.headline.controller;
import com.atguigu.headline.common.Result;
import com.atguigu.headline.pojo.NewsType;
import com.atguigu.headline.pojo.vo.HeadlineDetailVo;
import com.atguigu.headline.pojo.vo.HeadlineQueryVo;
import com.atguigu.headline.service.NewsHeadlineService;
import com.atguigu.headline.service.NewsTypeService;
import com.atguigu.headline.service.impl.NewsHeadlineServiceImpl;
import com.atguigu.headline.service.impl.NewsTypeServiceImpl;
import com.atguigu.headline.util.WebUtil;
import jakarta.servlet.ServletException;
import jakarta.servlet.annotation.WebServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
@WebServlet("/portal/*")
public class PortalController extends BaseController{
    private NewsHeadlineService headlineService=new NewsHeadlineServiceImpl();
    private NewsTypeService newsTypeService=new NewsTypeServiceImpl();
    /**
     * 查询所有新闻类型
     * @param req
     * @param resp
     * @throws ServletException
     * @throws IOException
     */
    protected void findAllTypes(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
        List<NewsType> newsTypeList =newsTypeService.findAll();
        WebUtil.writeJson(resp,Result.ok(newsTypeList));
    }
}
```

- NewsTypeService

```
package com.atguigu.headline.service;
import com.atguigu.headline.pojo.NewsType;
import java.util.List;
public interface NewsTypeService {
    /**
     * 查询全部新闻类型
     * @return
     */
    List<NewsType> findAll();
}
```

- NewsTypeServiceImpl

```
package com.atguigu.headline.service.impl;
import com.atguigu.headline.dao.NewsTypeDao;
import com.atguigu.headline.dao.impl.NewsTypeDaoImpl;
import com.atguigu.headline.pojo.NewsType;
import com.atguigu.headline.service.NewsTypeService;
import java.util.List;
public class NewsTypeServiceImpl implements NewsTypeService {
    private NewsTypeDao  newsTypeDao =new NewsTypeDaoImpl();
    @Override
    public List<NewsType> findAll() {
        return newsTypeDao.findAll();
    }
}
```

- NewUserDao

```
package com.atguigu.headline.dao;
import com.atguigu.headline.pojo.NewsType;
import java.util.List;
public interface NewsTypeDao {
    /**
     * 从数据库中查询全部新闻类型
     * @return
     */
    List<NewsType> findAll();
}
```

- NewsTypeDaoImpl

```
package com.atguigu.headline.dao.impl;
import com.atguigu.headline.dao.BaseDao;
import com.atguigu.headline.dao.NewsTypeDao;
import com.atguigu.headline.pojo.NewsType;
import java.util.List;
public class NewsTypeDaoImpl extends BaseDao implements NewsTypeDao {
    @Override
    public List<NewsType> findAll() {
        String sql ="select tid,tname from news_type";
        return  baseQuery(NewsType.class, sql);
    }
}
```

6.2 分页带条件查询所有头条

需求说明：客户端向服务端发送查询关键字、新闻类别、页码数、页大小。服务端根据条件搜索分页信息，返回含页码数、页大小、总页数、总记录数、当前页数据等信息。并根据时间降序，浏览量降序排序。

uri：portal/findNewsPage

请求方式：POST

请求参数：

```
{
    "keyWords":"马斯克", // 搜索标题关键字
    "type":0,           // 新闻类型
    "pageNum":1,        // 页码数
    "pageSize":"10"     // 页大小
}
```

响应示例：

```
{
    "code":"200",
    "message":"success"
    "data":{
        "pageInfo":{
            "pageData":[                          // 本页的数据
                {
                    "hid":"1",                    // 新闻id
                    "title":"尚硅谷宣布 ... ...",  // 新闻标题
                    "type":"1",                   // 新闻所属类别编号
                    "pageViews":"40",             // 新闻浏览量
                    "pastHours":"3" ,             // 发布时间已过小时数
                    "publisher":"1"               // 发布用户ID
                },
                {
                    "hid":"1",                    // 新闻id
                    "title":"尚硅谷宣布 ... ...",  // 新闻标题
                    "type":"1",                   // 新闻所属类别编号
                    "pageViews":"40",             // 新闻浏览量
                    "pastHours":"3",              // 发布时间已过小时数
                    "publisher":"1"               // 发布用户ID
                },
                {
                    "hid":"1",                    // 新闻id
                    "title":"尚硅谷宣布 ... ...",  // 新闻标题
```

```
                    "type":"1",                     // 新闻所属类别编号
                    "pageViews":"40",               // 新闻浏览量
                    "pastHours":"3",                // 发布时间已过小时数
                    "publisher":"1"                 // 发布用户ID
                }
            ],
            "pageNum":1,     //页码数
            "pageSize":10,  // 页大小
            "totalPage":20, // 总页数
            "totalSize":200 // 总记录数
        }
    }
}
```

后端代码：

- PortalController

```
package com.atguigu.headline.controller;
import com.atguigu.headline.common.Result;
import com.atguigu.headline.pojo.NewsType;
import com.atguigu.headline.pojo.vo.HeadlineDetailVo;
import com.atguigu.headline.pojo.vo.HeadlineQueryVo;
import com.atguigu.headline.service.NewsHeadlineService;
import com.atguigu.headline.service.NewsTypeService;
import com.atguigu.headline.service.impl.NewsHeadlineServiceImpl;
import com.atguigu.headline.service.impl.NewsTypeServiceImpl;
import com.atguigu.headline.util.WebUtil;
import jakarta.servlet.ServletException;
import jakarta.servlet.annotation.WebServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
@WebServlet("/portal/*")
public class PortalController extends BaseController{
    private NewsHeadlineService headlineService=new NewsHeadlineServiceImpl();
    private NewsTypeService newsTypeService=new NewsTypeServiceImpl();
    /**
     * 分页带条件查询新闻
     * @param req
     * @param resp
     * @throws ServletException
     * @throws IOException
     */
    protected void findNewsPage(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
        HeadlineQueryVo headLineQueryVo = WebUtil.readJson(req,
HeadlineQueryVo.class);
        // 查询分页五项数据
        Map<String,Object> pageInfo =headlineService.findPage(headLineQueryVo);
        // 将分页五项数据放入PageInfoMap
        Map<String,Object> pageInfoMap=new HashMap<>();
        pageInfoMap.put("pageInfo",pageInfo);
        // 响应JSON
```

```
        WebUtil.writeJson(resp, Result.ok(pageInfoMap));
    }
}
```

- NewsHeadlineService

```
package com.atguigu.headline.service;
import com.atguigu.headline.pojo.NewsHeadline;
import com.atguigu.headline.pojo.vo.HeadlineDetailVo;
import com.atguigu.headline.pojo.vo.HeadlineQueryVo;
import java.util.List;
import java.util.Map;
public interface NewsHeadlineService {
    /**
     * 分页查询头条新闻方法
     * @param headLineQueryVo
     * @return
     */
    Map<String, Object> findPage(HeadlineQueryVo headLineQueryVo);
}
```

- NewsHeadlineServiceImpl

```
package com.atguigu.headline.service.impl;
import com.atguigu.headline.dao.NewsHeadLineDao;
import com.atguigu.headline.dao.impl.NewsHeadlineDaoImpl;
import com.atguigu.headline.pojo.NewsHeadline;
import com.atguigu.headline.pojo.vo.HeadlineDetailVo;
import com.atguigu.headline.pojo.vo.HeadlinePageVo;
import com.atguigu.headline.pojo.vo.HeadlineQueryVo;
import com.atguigu.headline.service.NewsHeadlineService;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
public class NewsHeadlineServiceImpl  implements NewsHeadlineService {
    private NewsHeadLineDao newsHeadLineDao =new NewsHeadlineDaoImpl();
    @Override
    public Map<String, Object> findPage(HeadlineQueryVo headLineQueryVo) {
        // 准备一个map,用于装分页的五项数据
        Map<String,Object> pageInfo =new HashMap<>();
        // 分页查询本页数据
        List<HeadlinePageVo>  pageData =newsHeadLineDao.findPageList(headLineQueryVo);
        // 分页查询满足记录的总数据量
        int totalSize = newsHeadLineDao.findPageCount(headLineQueryVo);
        // 页大小
        int pageSize =headLineQueryVo.getPageSize();
        // 总页码数
        int totalPage=totalSize%pageSize == 0 ?  totalSize/pageSize  :
totalSize/pageSize+1;
        // 当前页码数
        int pageNum= headLineQueryVo.getPageNum();
        pageInfo.put("pageData",pageData);
        pageInfo.put("pageNum",pageNum);
        pageInfo.put("pageSize",pageSize);
        pageInfo.put("totalPage",totalPage);
        pageInfo.put("totalSize",totalSize);
        return pageInfo;
    }
```

```
}
```

- NewsHeadLineDao

```
package com.atguigu.headline.dao;
import com.atguigu.headline.pojo.NewsHeadline;
import com.atguigu.headline.pojo.vo.HeadlineDetailVo;
import com.atguigu.headline.pojo.vo.HeadlinePageVo;
import com.atguigu.headline.pojo.vo.HeadlineQueryVo;
import java.util.List;
public interface NewsHeadLineDao {
    /**
     * 根据查询条件,查询满足条件的记录数
     * @param headLineQueryVo
     * @return
     */
    int findPageCount(HeadlineQueryVo headLineQueryVo);
    /**
     * 根据查询条件,查询当前页数据
     * @param headLineQueryVo
     * @return
     */
    List<HeadlinePageVo> findPageList(HeadlineQueryVo headLineQueryVo);
}
```

- NewsHeadlineDaoImpl

```
package com.atguigu.headline.dao.impl;
import com.atguigu.headline.dao.BaseDao;
import com.atguigu.headline.dao.NewsHeadLineDao;
import com.atguigu.headline.pojo.NewsHeadline;
import com.atguigu.headline.pojo.vo.HeadlineDetailVo;
import com.atguigu.headline.pojo.vo.HeadlinePageVo;
import com.atguigu.headline.pojo.vo.HeadlineQueryVo;
import java.util.LinkedList;
import java.util.List;
public class NewsHeadlineDaoImpl extends BaseDao implements NewsHeadLineDao{

    @Override
    public int findPageCount(HeadlineQueryVo headLineQueryVo) {
        //  拼接动态 SQL,拼接参数
        List<Object> args =new LinkedList<>();
        String  sql="select count(1) from news_headline where is_deleted=0 ";
        StringBuilder sqlBuffer =new StringBuilder(sql) ;
        String keyWords = headLineQueryVo.getKeyWords();
        //判断并动态拼接条件
        if (null != keyWords && keyWords.length()>0){
            sqlBuffer.append("and title like ? ");
            args.add("%"+keyWords+"%");
        }
        //  判断并动态拼接条件
        Integer type = headLineQueryVo.getType();
        if(null != type  && type != 0){
            sqlBuffer.append("and type  =  ? ");
            args.add(type);
        }

        // 参数转数组
```

```
        Object[] argsArr = args.toArray();
        System.out.println(sqlBuffer.toString());
        Long totalSize = baseQueryObject(Long.class, sqlBuffer.toString(), argsArr);
        // 返回数据
        return totalSize.intValue();
    }

    @Override
    public List<HeadlinePageVo> findPageList(HeadlineQueryVo headLineQueryVo) {
        //  拼接动态 SQL,拼接参数
        List<Object> args =new LinkedList<>();
        String  sql="select hid,title,type,page_views
pageViews,TIMESTAMPDIFF(HOUR,create_time,NOW()) pastHours,publisher from news_headline
where is_deleted=0 ";
        StringBuilder sqlBuffer =new StringBuilder(sql) ;
        String keyWords = headLineQueryVo.getKeyWords();
        if (null != keyWords && keyWords.length()>0){
            sqlBuffer.append("and title like ? ");
            args.add("%"+keyWords+"%");
        }
        Integer type = headLineQueryVo.getType();
        if(null != type  && type != 0){
            sqlBuffer.append("and type  =  ? ");
            args.add(type);
        }

        sqlBuffer.append("order by pastHours , page_views desc ");
        sqlBuffer.append("limit ?, ?");
        args.add((headLineQueryVo.getPageNum()-1)*headLineQueryVo.getPageSize());
        args.add(headLineQueryVo.getPageSize());

        // 参数转数组
        Object[] argsArr = args.toArray();
        System.out.println(sqlBuffer.toString());
        List<HeadlinePageVo> pageData = baseQuery(HeadlinePageVo.class,
sqlBuffer.toString(), argsArr);

        return pageData;
    }
}
```

6.3 查看头条详情

事关国运的芯片，我们能否弯道超车？

新闻　2浏览　22小时前

芯片已经是一件事关中国国运的大事。众所周知，近几年中美芯片战愈演愈烈，美国政府打定主意在芯片上卡中国脖子。中国当然不能服输，但我们需要理解，这不是一场寻常的争夺。这里争的绝不仅仅是技术，更是创新文化、市场、制度、做事方法和冒险精神。在某些人心目中，中国经过 40 多年的改革开放，经济高度发展，拥有世界最强的供应链体系，中国制造已经统治全球，中国的量子信息、5G 技术等正在引领“第四次工业革命”.....中国已经“赢麻”了。而我相信你读了这本《芯片战争》，会有完全不同的看法。

需求说明：用户点击"查看全文"时，向服务端发送新闻id，后端根据新闻id查询完整新闻文章信息并返回。后端要同时让新闻的浏览量+1

uri：portal/showHeadlineDetail

请求方式：POST

请求参数：hid=1

响应示例：

```
{
    "code":"200",
    "message":"success",
    "data":{
        "headline":{
            "hid":"1",                          // 新闻id
            "title":"马斯克宣布 ... ...",        // 新闻标题
            "article":"... ..."                 // 新闻正文
            "type":"1",                         // 新闻所属类别编号
            "typeName":"科技",                  // 新闻所属类别
            "pageViews":"40",                   // 新闻浏览量
            "pastHours":"3" ,                   // 发布时间已过小时数
            "publisher":"1" ,                    // 发布用户ID
            "author":"张三"                     // 新闻作者
        }
    }
}
```

后端代码：

- PortalController

```
package com.atguigu.headline.controller;
import com.atguigu.headline.common.Result;
import com.atguigu.headline.pojo.NewsType;
import com.atguigu.headline.pojo.vo.HeadlineDetailVo;
import com.atguigu.headline.pojo.vo.HeadlineQueryVo;
```

```
import com.atguigu.headline.service.NewsHeadlineService;
import com.atguigu.headline.service.NewsTypeService;
import com.atguigu.headline.service.impl.NewsHeadlineServiceImpl;
import com.atguigu.headline.service.impl.NewsTypeServiceImpl;
import com.atguigu.headline.util.WebUtil;
import jakarta.servlet.ServletException;
import jakarta.servlet.annotation.WebServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
@WebServlet("/portal/*")
public class PortalController extends BaseController{

    private NewsHeadlineService headlineService=new NewsHeadlineServiceImpl();
    private NewsTypeService newsTypeService=new NewsTypeServiceImpl();
    /**
     * 查询单个新闻详情
     * @param req
     * @param resp
     * @throws ServletException
     * @throws IOException
     */
    protected void showHeadlineDetail(HttpServletRequest req, HttpServletResponse
resp) throws ServletException, IOException {
        // 获取要查询的详情新闻id
        Integer hid =Integer.parseInt(req.getParameter("hid"));

        // 查询新闻详情vo
        HeadlineDetailVo headlineDetailVo =headlineService.findHeadlineDetail(hid);
        // 封装data内容
        Map<String ,Object> data =new HashMap<>();
        data.put("headline",headlineDetailVo);
        // 响应JSON
        WebUtil.writeJson(resp,Result.ok(data));
    }
}
```

- NewsHeadlineService

```
package com.atguigu.headline.service;
import com.atguigu.headline.pojo.NewsHeadline;
import com.atguigu.headline.pojo.vo.HeadlineDetailVo;
import com.atguigu.headline.pojo.vo.HeadlineQueryVo;
import java.util.List;
import java.util.Map;
public interface NewsHeadlineService {
    /**
     * 根据头条id,显示头条详情
     * @param hid
     * @return
     */
    HeadlineDetailVo findHeadlineDetail(Integer hid);
}
```

- NewsHeadlineServiceImpl

```
package com.atguigu.headline.service.impl;
import com.atguigu.headline.dao.NewsHeadLineDao;
import com.atguigu.headline.dao.impl.NewsHeadlineDaoImpl;
import com.atguigu.headline.pojo.NewsHeadline;
import com.atguigu.headline.pojo.vo.HeadlineDetailVo;
import com.atguigu.headline.pojo.vo.HeadlinePageVo;
import com.atguigu.headline.pojo.vo.HeadlineQueryVo;
import com.atguigu.headline.service.NewsHeadlineService;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
public class NewsHeadlineServiceImpl  implements NewsHeadlineService {
    private NewsHeadLineDao newsHeadLineDao =new NewsHeadlineDaoImpl();
    @Override
    public HeadlineDetailVo findHeadlineDetail(Integer hid) {
        // 修改新闻信息浏览量+1
        newsHeadLineDao.increasePageViews(hid);
        // 查询新闻详情
        return newsHeadLineDao.findHeadlineDetail(hid);
    }
}
```

- NewsHeadLineDao

```
package com.atguigu.headline.dao;
import com.atguigu.headline.pojo.NewsHeadline;
import com.atguigu.headline.pojo.vo.HeadlineDetailVo;
import com.atguigu.headline.pojo.vo.HeadlinePageVo;
import com.atguigu.headline.pojo.vo.HeadlineQueryVo;
import java.util.List;
public interface NewsHeadLineDao {
    /**
     * 多表查询新闻详情
     * @param hid
     * @return
     */
    HeadlineDetailVo findHeadlineDetail(Integer hid);
    int increasePageViews(Integer hid);
}
```

- NewsHeadlineDaoImpl

```
package com.atguigu.headline.dao.impl;
import com.atguigu.headline.dao.BaseDao;
import com.atguigu.headline.dao.NewsHeadLineDao;
import com.atguigu.headline.pojo.NewsHeadline;
import com.atguigu.headline.pojo.vo.HeadlineDetailVo;
import com.atguigu.headline.pojo.vo.HeadlinePageVo;
import com.atguigu.headline.pojo.vo.HeadlineQueryVo;
import java.util.LinkedList;
import java.util.List;
public class NewsHeadlineDaoImpl extends BaseDao implements NewsHeadLineDao{
    @Override
    public HeadlineDetailVo findHeadlineDetail(Integer hid) {
        String sql ="select hid,title,article,type, tname typeName ,page_views
pageViews,TIMESTAMPDIFF(HOUR,create_time,NOW()) pastHours,publisher,nick_name author
from news_headline h left join  news_type t on h.type = t.tid left join news_user u
on h.publisher = u.uid where hid = ?";
```

```
        List<HeadlineDetailVo> headlineDetailVoList =
baseQuery(HeadlineDetailVo.class, sql, hid);
        if(null != headlineDetailVoList && headlineDetailVoList.size()>0)
            return headlineDetailVoList.get(0);
        return null;
    }

    @Override
    public int increasePageViews(Integer hid) {
        String sql ="update news_headline set page_views = page_views +1 where hid
=?";
        return baseUpdate(sql,hid);
    }
}
```

七 头条发布修改和删除

7.1 登录校验

> 需求说明：客户端在进入发布页前、发布新闻前、进入修改页前、修改前、删除新闻前先向服务端发送请求携带token请求头。后端接收token请求头后，校验用户登录是否过期并做响应。前端根据响应信息提示用户进入登录页还是进入正常业务页面

> uri：user/checkLogin

> 请求方式：GET

> 请求参数：无

> 请求头：token: ... ...

> 响应示例：

- 登录未过期

```
{
    "code":"200",
    "message":"success",
    "data":{}
}
```

- 登录已过期

```
{
    "code":"504",
    "message":"loginExpired",
    "data":{}
}
```

后端代码：

- NewsUserController

```
package com.atguigu.headline.controller;
import com.atguigu.headline.common.Result;
import com.atguigu.headline.common.ResultCodeEnum;
import com.atguigu.headline.pojo.NewsHeadline;
import com.atguigu.headline.pojo.NewsUser;
import com.atguigu.headline.service.NewsUserService;
import com.atguigu.headline.service.impl.NewsUserServiceImpl;
import com.atguigu.headline.util.JwtHelper;
import com.atguigu.headline.util.MD5Util;
import com.atguigu.headline.util.WebUtil;
import jakarta.servlet.ServletException;
import jakarta.servlet.annotation.WebServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.util.HashMap;
import java.util.Map;
@WebServlet("/user/*")
public class NewsUserController extends BaseController{
    private NewsUserService newsUserService =new NewsUserServiceImpl();
    /**
     * 通过token检验用户登录是否过期
     * @param req
     * @param resp
     * @throws ServletException
     * @throws IOException
     */
    protected void checkLogin(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        String token = req.getHeader("token");
        Result  result =Result.build(null,ResultCodeEnum.NOTLOGIN);
        if(null!= token){
            if (!JwtHelper.isExpiration(token)) {
                result=Result.ok(null);
            }
        }
        WebUtil.writeJson(resp,result);
    }
}
```

- 登录校验过滤器

```
package com.atguigu.headline.filters;
import com.atguigu.headline.common.Result;
import com.atguigu.headline.common.ResultCodeEnum;
import com.atguigu.headline.util.JwtHelper;
import com.atguigu.headline.util.WebUtil;
import jakarta.servlet.*;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import java.io.IOException;
public class LoginFilter implements Filter {
    @Override
```

```
    public void doFilter(ServletRequest servletRequest, ServletResponse
servletResponse, FilterChain filterChain) throws IOException, ServletException {
        HttpServletRequest request =(HttpServletRequest)  servletRequest;
        String token = request.getHeader("token");
        boolean flag =false;
        // token不为空并且没过期
        if (null  != token ){
            boolean expiration = JwtHelper.isExpiration(token);
            if (!expiration ){
               flag=true;
            }
        }
        if (flag){
            filterChain.doFilter(servletRequest,servletResponse);
        }else{
            WebUtil.writeJson((HttpServletResponse) servletResponse,
Result.build(null, ResultCodeEnum.NOTLOGIN));
        }
    }
}
```

- web.xml中配置登录校验过滤器

```
    <!--登录校验过滤器-->
    <filter>
        <filter-name>loginFilter</filter-name>
        <filter-class>com.atguigu.headline.filters.LoginFilter</filter-class>
    </filter>
    <filter-mapping>
        <filter-name>loginFilter</filter-name>
        <url-pattern>/headline/*</url-pattern>
    </filter-mapping>
```

7.2 提交发布头条

需求说明：用户在客户端输入发布的新闻信息完毕后，发布前先请求后端的登录校验接口验证登录。登录通过则提交新闻信息，后端将新闻信息存入数据库。

uri：headline/publish

请求方式：POST

请求头：token: ... ...

请求参数：

```
{
    "title":"尚硅谷宣布 ... ...",    // 文章标题
    "article":"... ...",            // 文章内容
    "type":"1"                      // 文章类别
}
```

响应示例：

- 发布成功

```
{
    "code":"200",
    "message":"success",
    "data":{}
}
```

- 失去登录状态发布失败

```
{
    "code":"504",
    "message":"loginExpired",
    "data":{}
}
```

后端代码：

- NewsHeadlineController

```
package com.atguigu.headline.controller;
import com.atguigu.headline.common.Result;
import com.atguigu.headline.pojo.NewsHeadline;
import com.atguigu.headline.service.NewsHeadlineService;
import com.atguigu.headline.service.impl.NewsHeadlineServiceImpl;
import com.atguigu.headline.util.JwtHelper;
import com.atguigu.headline.util.WebUtil;
import jakarta.servlet.ServletException;
import jakarta.servlet.annotation.WebServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.util.HashMap;
```

```
import java.util.Map;
@WebServlet("/headline/*")
public class NewsHeadlineController extends BaseController {
    private NewsHeadlineService newsHeadlineService =new NewsHeadlineServiceImpl();
    /**
     * 发布新闻
     * @param req
     * @param resp
     * @throws ServletException
     * @throws IOException
     */
    protected void publish(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        // 读取新闻信息
        NewsHeadline newsHeadline = WebUtil.readJson(req, NewsHeadline.class);
        // 通过token获取发布者ID
        String token = req.getHeader("token");
        Long userId = JwtHelper.getUserId(token);
        newsHeadline.setPublisher(userId.intValue());
        // 将新闻存入数据库
        newsHeadlineService.addNewsHeadline(newsHeadline);
        WebUtil.writeJson(resp,Result.ok(null));
    }
}
```

- NewsHeadlineService

```
package com.atguigu.headline.service;
import com.atguigu.headline.pojo.NewsHeadline;
import com.atguigu.headline.pojo.vo.HeadlineDetailVo;
import com.atguigu.headline.pojo.vo.HeadlineQueryVo;
import java.util.List;
import java.util.Map;
public interface NewsHeadlineService {
    /**
     * 新增头条
     * @param newsHeadline
     * @return
     */
    int addNewsHeadline(NewsHeadline newsHeadline);
}
```

- NewsHeadlineServiceImpl

```
package com.atguigu.headline.service.impl;
import com.atguigu.headline.dao.NewsHeadLineDao;
import com.atguigu.headline.dao.impl.NewsHeadlineDaoImpl;
import com.atguigu.headline.pojo.NewsHeadline;
import com.atguigu.headline.pojo.vo.HeadlineDetailVo;
import com.atguigu.headline.pojo.vo.HeadlinePageVo;
import com.atguigu.headline.pojo.vo.HeadlineQueryVo;
import com.atguigu.headline.service.NewsHeadlineService;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
public class NewsHeadlineServiceImpl  implements NewsHeadlineService {
    private NewsHeadLineDao newsHeadLineDao =new NewsHeadlineDaoImpl();
    public int addNewsHeadline(NewsHeadline newsHeadline) {
```

```
        return newsHeadLineDao.addNewsHeadline(newsHeadline);
    }
}
```

- NewsHeadLineDao

```
package com.atguigu.headline.dao;
import com.atguigu.headline.pojo.NewsHeadline;
import com.atguigu.headline.pojo.vo.HeadlineDetailVo;
import com.atguigu.headline.pojo.vo.HeadlinePageVo;
import com.atguigu.headline.pojo.vo.HeadlineQueryVo;
import java.util.List;
public interface NewsHeadLineDao {
    /**
     * 头条存入数据库
     * @param newsHeadline
     * @return
     */
    int addNewsHeadline(NewsHeadline newsHeadline);
}
```

- NewsHeadlineDaoImpl

```
package com.atguigu.headline.dao.impl;
import com.atguigu.headline.dao.BaseDao;
import com.atguigu.headline.dao.NewsHeadLineDao;
import com.atguigu.headline.pojo.NewsHeadline;
import com.atguigu.headline.pojo.vo.HeadlineDetailVo;
import com.atguigu.headline.pojo.vo.HeadlinePageVo;
import com.atguigu.headline.pojo.vo.HeadlineQueryVo;
import java.util.LinkedList;
import java.util.List;
public class NewsHeadlineDaoImpl extends BaseDao implements NewsHeadLineDao{
    @Override
    public int addNewsHeadline(NewsHeadline newsHeadline) {
        String sql = "insert into news_headline
values(DEFAULT,?,?,?,?,0,NOW(),NOW(),0)";

        return baseUpdate(
                sql,
                newsHeadline.getTitle(),
                newsHeadline.getArticle(),
                newsHeadline.getType(),
                newsHeadline.getPublisher()
        );
    }
}
```

7.3 修改头条回显

需求说明：前端先调用登录校验接口，校验登录是否过期。登录校验通过后，则根据新闻id查询新闻的完整信息并响应给前端。

uri：headline/findHeadlineByHid

请求方式：POST

请求参数：hid=1

响应示例：

- 查询成功

```
{
    "code":"200",
    "message":"success",
    "data":{
        "headline":{
            "hid":"1",
            "title":"马斯克宣布",
            "article":"... ... ",
            "type":"2"
        }
    }
}
```

后端代码：

- NewsHeadlineController

```
package com.atguigu.headline.controller;
import com.atguigu.headline.common.Result;
import com.atguigu.headline.pojo.NewsHeadline;
import com.atguigu.headline.service.NewsHeadlineService;
import com.atguigu.headline.service.impl.NewsHeadlineServiceImpl;
import com.atguigu.headline.util.JwtHelper;
import com.atguigu.headline.util.WebUtil;
import jakarta.servlet.ServletException;
import jakarta.servlet.annotation.WebServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.util.HashMap;
import java.util.Map;
@WebServlet("/headline/*")
public class NewsHeadlineController extends BaseController {
    private NewsHeadlineService newsHeadlineService =new NewsHeadlineServiceImpl();
    /**
     * 修改新闻回显
     * @param req
     * @param resp
     * @throws ServletException
     * @throws IOException
     */
    protected void findHeadlineByHid(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
        Integer hid = Integer.parseInt(req.getParameter("hid"));
        NewsHeadline newsHeadline =newsHeadlineService.findHeadlineByHid(hid);
        Map<String ,Object> data =new HashMap<>();
        data.put("headline",newsHeadline);
        WebUtil.writeJson(resp,Result.ok(data));

    }
}
```

- NewsHeadlineService

```
package com.atguigu.headline.service;
import com.atguigu.headline.pojo.NewsHeadline;
import com.atguigu.headline.pojo.vo.HeadlineDetailVo;
import com.atguigu.headline.pojo.vo.HeadlineQueryVo;
import java.util.List;
import java.util.Map;
public interface NewsHeadlineService {
    /**
     * 根据新闻id查询单个新闻
     * @param hid
     * @return
     */
    NewsHeadline findHeadlineByHid(Integer hid);
}
```

- NewsHeadlineServiceImpl

```
package com.atguigu.headline.service.impl;
import com.atguigu.headline.dao.NewsHeadLineDao;
```

```
import com.atguigu.headline.dao.impl.NewsHeadlineDaoImpl;
import com.atguigu.headline.pojo.NewsHeadline;
import com.atguigu.headline.pojo.vo.HeadlineDetailVo;
import com.atguigu.headline.pojo.vo.HeadlinePageVo;
import com.atguigu.headline.pojo.vo.HeadlineQueryVo;
import com.atguigu.headline.service.NewsHeadlineService;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
public class NewsHeadlineServiceImpl  implements NewsHeadlineService {
    private NewsHeadLineDao newsHeadLineDao =new NewsHeadlineDaoImpl();
    @Override
    public NewsHeadline findHeadlineByHid(Integer hid) {
        return newsHeadLineDao.findHeadlineByHid(hid);
    }
}
```

- NewsHeadLineDao

```
package com.atguigu.headline.dao;
import com.atguigu.headline.pojo.NewsHeadline;
import com.atguigu.headline.pojo.vo.HeadlineDetailVo;
import com.atguigu.headline.pojo.vo.HeadlinePageVo;
import com.atguigu.headline.pojo.vo.HeadlineQueryVo;
import java.util.List;
public interface NewsHeadLineDao {
    NewsHeadline findHeadlineByHid(Integer hid);
}
```

- NewUserDaoImpl

```
package com.atguigu.headline.dao.impl;
import com.atguigu.headline.dao.BaseDao;
import com.atguigu.headline.dao.NewsHeadLineDao;
import com.atguigu.headline.pojo.NewsHeadline;
import com.atguigu.headline.pojo.vo.HeadlineDetailVo;
import com.atguigu.headline.pojo.vo.HeadlinePageVo;
import com.atguigu.headline.pojo.vo.HeadlineQueryVo;
import java.util.LinkedList;
import java.util.List;
public class NewsHeadlineDaoImpl extends BaseDao implements NewsHeadLineDao{
    @Override
    public NewsHeadline findHeadlineByHid(Integer hid) {
        String sql ="select hid,title,article,type,publisher,page_views pageViews from
news_headline where hid =?";
        List<NewsHeadline> newsHeadlineList = baseQuery(NewsHeadline.class, sql, hid);
        if(null != newsHeadlineList && newsHeadlineList.size()>0)
            return newsHeadlineList.get(0);
        return null;
    }
}
```

7.4 保存修改

需求描述：客户端将新闻信息修改后，提交前先请求登录校验接口校验登录状态。登录校验通过则提交修改后的新闻信息，后端接收并更新进入数据库。

uri：headline/update

请求方式： POST

请求参数：

```
{
    "hid":"1",
    "title":"尚硅谷宣布 ... ...",
    "article":"... ...",
    "type":"2"
}
```

响应示例：

- 修改成功

```
{
    "code":"200",
    "message":"success",
    "data":{}
}
```

- 修改失败

```
{
    "code":"504",
    "message":"loginExpired",
    "data":{}
}
```

后端代码：

- NewsHeadlineController

```
package com.atguigu.headline.controller;
import com.atguigu.headline.common.Result;
import com.atguigu.headline.pojo.NewsHeadline;
import com.atguigu.headline.service.NewsHeadlineService;
import com.atguigu.headline.service.impl.NewsHeadlineServiceImpl;
import com.atguigu.headline.util.JwtHelper;
import com.atguigu.headline.util.WebUtil;
import jakarta.servlet.ServletException;
import jakarta.servlet.annotation.WebServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.util.HashMap;
import java.util.Map;
@WebServlet("/headline/*")
```

```java
public class NewsHeadlineController extends BaseController {
    private NewsHeadlineService newsHeadlineService =new NewsHeadlineServiceImpl();
    /**
     * 更新新闻信息
     * @param req
     * @param resp
     * @throws ServletException
     * @throws IOException
     */
    protected void update(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        NewsHeadline newsHeadline = WebUtil.readJson(req, NewsHeadline.class);
        newsHeadlineService.updateNewsHeadline(newsHeadline);
        WebUtil.writeJson(resp,Result.ok(null));
    }
}
```

- NewsHeadlineService

```java
package com.atguigu.headline.service;
import com.atguigu.headline.pojo.NewsHeadline;
import com.atguigu.headline.pojo.vo.HeadlineDetailVo;
import com.atguigu.headline.pojo.vo.HeadlineQueryVo;
import java.util.List;
import java.util.Map;
public interface NewsHeadlineService {
    int updateNewsHeadline(NewsHeadline newsHeadline);
}
```

- NewsHeadlineServiceImpl

```java
package com.atguigu.headline.service.impl;
import com.atguigu.headline.dao.NewsHeadLineDao;
import com.atguigu.headline.dao.impl.NewsHeadlineDaoImpl;
import com.atguigu.headline.pojo.NewsHeadline;
import com.atguigu.headline.pojo.vo.HeadlineDetailVo;
import com.atguigu.headline.pojo.vo.HeadlinePageVo;
import com.atguigu.headline.pojo.vo.HeadlineQueryVo;
import com.atguigu.headline.service.NewsHeadlineService;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
public class NewsHeadlineServiceImpl  implements NewsHeadlineService {
    private NewsHeadLineDao newsHeadLineDao =new NewsHeadlineDaoImpl();
    @Override
    public int updateNewsHeadline(NewsHeadline newsHeadline) {
        return newsHeadLineDao.updateNewsHeadline(newsHeadline);
    }
}
```

- NewsHeadLineDao

```
package com.atguigu.headline.dao;
import com.atguigu.headline.pojo.NewsHeadline;
import com.atguigu.headline.pojo.vo.HeadlineDetailVo;
import com.atguigu.headline.pojo.vo.HeadlinePageVo;
import com.atguigu.headline.pojo.vo.HeadlineQueryVo;
import java.util.List;
public interface NewsHeadLineDao {
    int updateNewsHeadline(NewsHeadline newsHeadline);
}
```

- NewUserDaoImpl

```
package com.atguigu.headline.dao.impl;
import com.atguigu.headline.dao.BaseDao;
import com.atguigu.headline.dao.NewsHeadLineDao;
import com.atguigu.headline.pojo.NewsHeadline;
import com.atguigu.headline.pojo.vo.HeadlineDetailVo;
import com.atguigu.headline.pojo.vo.HeadlinePageVo;
import com.atguigu.headline.pojo.vo.HeadlineQueryVo;
import java.util.LinkedList;
import java.util.List;
public class NewsHeadlineDaoImpl extends BaseDao implements NewsHeadLineDao{
    @Override
    public int updateNewsHeadline(NewsHeadline newsHeadline) {
        String sql ="update news_headline set title = ?, article= ? , type =? ,
update_time = NOW() where hid = ? ";
        return baseUpdate(
                sql,
                newsHeadline.getTitle(),
                newsHeadline.getArticle(),
                newsHeadline.getType(),
                newsHeadline.getHid()
        );
    }
}
```

7.5 删除头条

需求说明：将要删除的新闻id发送给服务端，服务端校验登录是否过期，未过期则直接删除，过期则响应登录过期信息。

uri：headline/removeByHid

> 请求方式：POST

> 请求参数：hid=1

> 响应示例：

- 删除成功

```
{
    "code":"200",
    "message":"success",
    "data":{}
}
```

- 删除失败

```
{
    "code":"504",
    "message":"loginExpired",
    "data":{}

}
```

> 后端代码：

- NewsHeadlineController

```
package com.atguigu.headline.controller;
import com.atguigu.headline.common.Result;
import com.atguigu.headline.pojo.NewsHeadline;
import com.atguigu.headline.service.NewsHeadlineService;
import com.atguigu.headline.service.impl.NewsHeadlineServiceImpl;
import com.atguigu.headline.util.JwtHelper;
import com.atguigu.headline.util.WebUtil;
import jakarta.servlet.ServletException;
import jakarta.servlet.annotation.WebServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.util.HashMap;
import java.util.Map;
@WebServlet("/headline/*")
public class NewsHeadlineController extends BaseController {
    private NewsHeadlineService newsHeadlineService =new NewsHeadlineServiceImpl();
    /**
     * 删除新闻
     * @param req
     * @param resp
     * @throws ServletException
     * @throws IOException
     */
    protected void removeByHid(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
        Integer hid = Integer.parseInt(req.getParameter("hid"));
        newsHeadlineService.removeByHid(hid);
```

```
            WebUtil.writeJson(resp,Result.ok(null));
        }
    }
```

- NewsHeadlineService

```
package com.atguigu.headline.service;
import com.atguigu.headline.pojo.NewsHeadline;
import com.atguigu.headline.pojo.vo.HeadlineDetailVo;
import com.atguigu.headline.pojo.vo.HeadlineQueryVo;
import java.util.List;
import java.util.Map;
public interface NewsHeadlineService {
    int removeByHid(Integer hid);
}
```

- NewsHeadlineServiceImpl

```
package com.atguigu.headline.service.impl;
import com.atguigu.headline.dao.NewsHeadLineDao;
import com.atguigu.headline.dao.impl.NewsHeadlineDaoImpl;
import com.atguigu.headline.pojo.NewsHeadline;
import com.atguigu.headline.pojo.vo.HeadlineDetailVo;
import com.atguigu.headline.pojo.vo.HeadlinePageVo;
import com.atguigu.headline.pojo.vo.HeadlineQueryVo;
import com.atguigu.headline.service.NewsHeadlineService;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
public class NewsHeadlineServiceImpl  implements NewsHeadlineService {
    private NewsHeadLineDao newsHeadLineDao =new NewsHeadlineDaoImpl();
    @Override
    public int removeByHid(Integer hid) {
        return newsHeadLineDao.removeByHid(hid);
    }
}
```

- NewsHeadLineDao

```
package com.atguigu.headline.dao;
import com.atguigu.headline.pojo.NewsHeadline;
import com.atguigu.headline.pojo.vo.HeadlineDetailVo;
import com.atguigu.headline.pojo.vo.HeadlinePageVo;
import com.atguigu.headline.pojo.vo.HeadlineQueryVo;
import java.util.List;
public interface NewsHeadLineDao {
    int removeByHid(Integer hid);
}
```

- NewsHeadlineDaoImpl

```
package com.atguigu.headline.dao.impl;
import com.atguigu.headline.dao.BaseDao;
import com.atguigu.headline.dao.NewsHeadLineDao;
import com.atguigu.headline.pojo.NewsHeadline;
import com.atguigu.headline.pojo.vo.HeadlineDetailVo;
import com.atguigu.headline.pojo.vo.HeadlinePageVo;
import com.atguigu.headline.pojo.vo.HeadlineQueryVo;
```

```
import java.util.LinkedList;
import java.util.List;
public class NewsHeadlineDaoImpl extends BaseDao implements NewsHeadLineDao{
    @Override
    public int removeByHid(Integer hid) {
        String sql ="update news_headline set is_deleted =1 ,  update_time =NOW()
where hid = ? ";
        return baseUpdate(sql,hid);
    }
}
```