

# JavaWeb

## 001. 教程简介 哔哩哔哩bilibili

### HTML

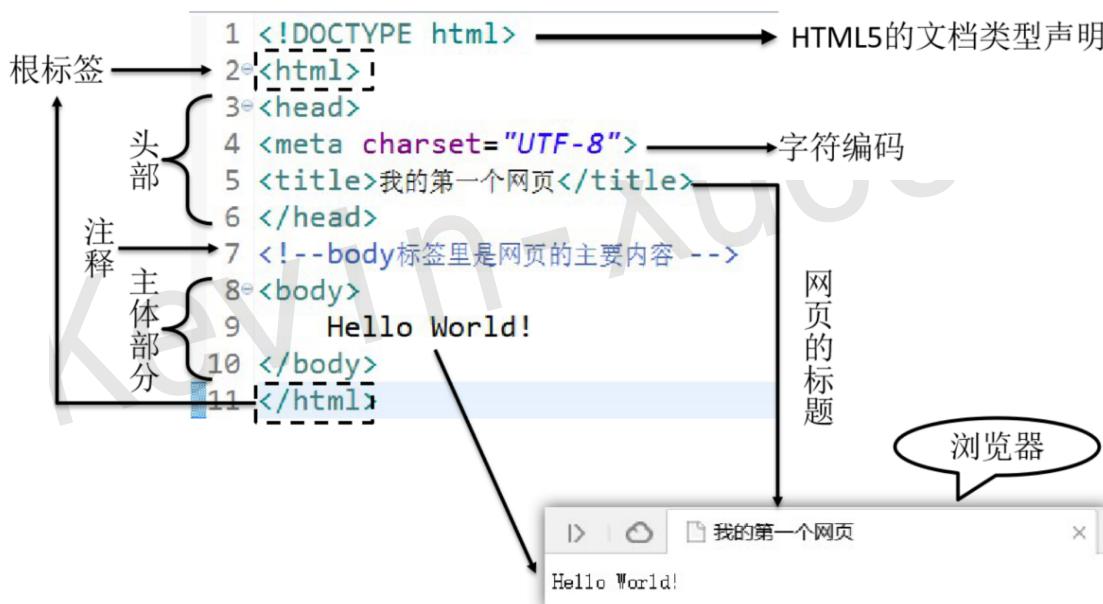
#### 入门

##### 1. 介绍

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。

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。

##### 2. 基础结构



<head></head> 头标签 定义那些不直接展示在页面主体上,但是又很重要的内容

- 1 字符集
- 2 css引入
- 3 js引入
- 4 其他 ... ...

<body></body> 体标签  
1 定义要展示到页面主体的标签

##### 3. 概念

- (1) 标签 (tag) : 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。
- (2) 属性 (attribute) : 一般在开始标签中，用于定义标签的一些特征。
- (3) 文本 (text) : 双标签中间的文字，包含空格换行等结构。

(4) 元素 (element) : 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。

#### 4. 通过Live Server小型服务器运行项目

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

The screenshot shows the Visual Studio Code interface. At the top, there's a title bar with tabs for '01html的基本结构.html' and '...'. Below the title bar, the status bar displays '行 8, 列 18 空格: 4 UTF-8 CRLF HTML'. A red box highlights the 'Go Live' button in the status bar. To the right of the status bar is a context menu with several options: '重命名符号' (Shift+F6), '更改所有匹配项' (Shift+F6), '格式化文档' (Ctrl+Alt+L), '重构...' (Ctrl+Shift+Alt+T), '剪切' (Ctrl+X), '复制' (Ctrl+C), '粘贴' (Ctrl+V), 'Spelling' (with a dropdown arrow), and '添加到监视'. The bottom of the context menu has two items: 'Open with Live Server' (Alt+L Alt+O) and 'Stop Live Server' (Alt+L Alt+C). The 'Open with Live Server' item is also highlighted with a red box.

```
<!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>
```

Live Server使用完毕后,要记得关闭



## 标签(标题/段落/换行/列表)

### 1. 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. 段落

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>
```

```
<body>
    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）。
```

```
        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。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。
```

```
</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。
```

```
        当前，算力正朝智能敏捷、绿色低碳、安全可靠方向发展。
```

```
</p>
```

```
</body>
```

## 效果

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。

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）。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。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。

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。当前，算力正朝智能敏捷、绿色低碳、安全可靠方向发展。

## 3. 换行

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>
```

```
    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）。
```

```
<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>
```

## 效果

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）。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。

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。

## 4. 列表

(1) 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

- 列表标签 ol

- 列表项标签 li

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

- 效果
1. JAVA
  2. 前端
  3. 大数据

(2) 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

- 列表标签 ul
- 列表项标签 li

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

- 效果
- JAVASE
  - JAVAEE
  - 数据库

(3) 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>
```

- 1. JAVA
  - JAVASE
  - JAVAEE
  - 数据库
- 效果
- 2. 前端
- 3. 大数据

## 标签(超链接/多媒体)

### 1. 超链接

超链接标签：点击后带有链接跳转的标签，也叫作a标签。

href属性用于定义连接

- 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
- href中也可以定义完整的URL

target用于定义打开的方式

- \_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>
```

```
<!--
```

href属性用于定义连接

    href中可以使用绝对路径,以/开头,始终以一个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, 表示当前资源所在路径, 可以省略不写

        ../开头, 表示当前资源上一层

    href中也可以定义完整的URL

target用于定义打开的方式

    \_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

```
-->
```

    <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. 多媒体

### (1) img标签，用于在页面上引入图片

```
<!--
```

```
src
```

用于定义图片的连接，路径设定与超链接href相同

```
title
```

用于定义鼠标悬停时显示的文字

```
alt
```

用于定义图片加载失败时显示的提示文字

```
-->
```

```

```

### (2) audio标签，用于在页面上引入声音

```
<!--
```

```
src
```

用于定义目标声音资源

```
autoplay
```

用于控制打开页面时是否自动播放

```
controls
```

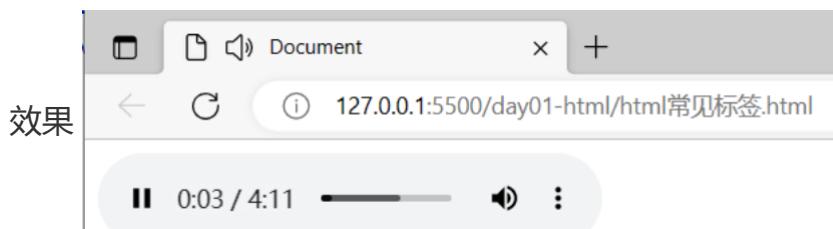
用于控制是否展示控制面板

```
loop
```

用于控制是否进行循环播放

```
-->
```

```
<audio src="img/music.mp3" autoplay="autoplay"
controls="controls" loop="loop" />
```

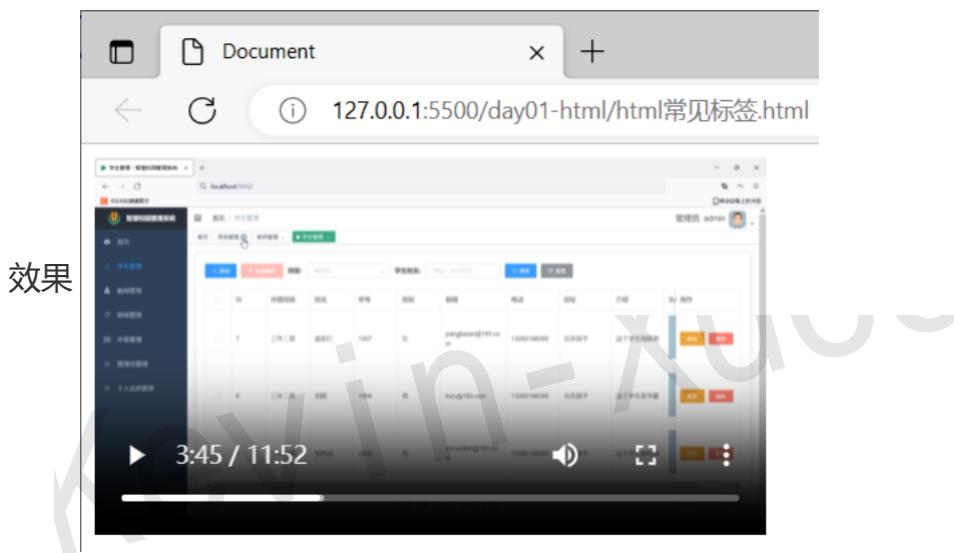


### (3) video标签，用于在页面上引入视频

```

<body>
    <!--
        src
            用于定义目标视频资源
        autoplay
            用于控制打开页面时是否自动播放
        controls
            用于控制是否展示控制面板
        loop
            用于控制是否进行循环播放
    -->
    <video src="img/movie.mp4" autoplay="autoplay"
    controls="controls" loop="loop" width="400px" />
</body>

```



## 标签(表格)

### 1. 基本使用

```


| 标签 代表行内的一格 <th>标签 自带加粗和居中效果的td </th> | 标签 自带加粗和居中效果的td |
|--------------------------------------|-----------------|
| 标签 代表行内的一格 <th>标签 自带加粗和居中效果的td </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> <!--
css 居中-->
<table border="1px" style="width: 400px; margin: 0px
auto;"> <!--表格居中，设置宽度和边框-->
<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>
    <td>99</td>
</tr>
<tr>
    <td>3</td>
    <td>王小虎</td>
    <td>98</td>
</tr>
</table>

```

效果

排名	姓名	分数
1	张小明	100
2	李小东	99
3	王小虎	98

## 2. 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" style="text-align: center;">前三名升职加薪
->
</td>
</tr>
<tr>
<td>2</td>
<td>李小东</td></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**

效果

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

### 3. 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>
    <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>

```

效果

**员工技能竞赛评分表**

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

## 标签(表单/表单项)

### 1. 表单标签

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**, **form** 标签的属性之一，用于定义信息提交的服务器的地址  
**method**, **form** 标签的属性之一，用于定义信息的提交方式  
    **get**     **get** 方式， 数据会缀到 **url** 后，以?作为参数开始的标识，多个参数用&隔开

        地址栏上只能是字符，不能提交文件

        地址栏长度有限，提交的数据量不大

**post**   **post** 方式，数据会通过请求体发送，不会在缀到 **url** 后  
        数据单独打包通过请求体发送，提交的数据量比较大

**input** 标签，主要的表单项标签，可以用于定义表单项

**name**, **input** 标签的属性之一，用于定义提交的参数名，即在 **get** 请求中会出现在?后面的参数名（提交的实参定义在表单项中的 **value** 属性中，**input** 输入值即为 **value**）

**type**, **input** 标签的属性之一，用于定义表单项类型

**text**    单行普通文本框

**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 &nbsp;&nbsp;&nbsp;码 <input type="password" name="password" /> <br>
    <input type="submit" value="登录" />
    <input type="reset" value="重置" />
</form>
```



### 2. 表单项标签

#### (1) 单行文本框

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

效果 个性签名:

## (2) 密码框

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

效果 密码:

## (3) 单选框

你的性别是:

```
<input type="radio" name="sex" value="spring" />男  
<input type="radio" name="sex" value="summer"  
checked="checked" />女
```

效果 你的性别是: 男 女

说明

1. name属性相同的radio为一组，组内互斥
2. 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
3. 设置checked="checked"属性设置默认被选中的radio
4. 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

## (4) 复选框

你喜欢的球队是:

```
<input type="checkbox" name="team" value="Brazil"/>巴西 <!--  
name是key, value是value-->  
<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 <!--默认选中-->  
<input type="checkbox" name="team" value="Italian"/>意大利
```

效果

你最喜欢的球队是: 巴西 德国 法国 中国 意大利

## (5) 隐藏域

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。

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

#### (6) 多行文本框

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

效果

自我介绍:

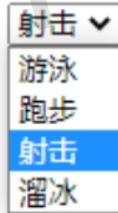
#### (7) 下拉框

你喜欢的运动是：

```
<select name="interesting">
    <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>
```

效果

你喜欢的运动是：



说明

1. 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。
2. **name** 属性在 **select** 标签中设置，**value** 属性在 **option** 标签中设置。
3. **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。
4. 通过在 **option** 标签中设置 **selected="selected"** 属性实现默认选中的效果。

#### (8) 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"/>
```

#### 说明

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

## (9) 文件标签

头像:<input type="file" name="file"/>

效果      头像:  未选择文件

## 标签(布局)&特殊字符

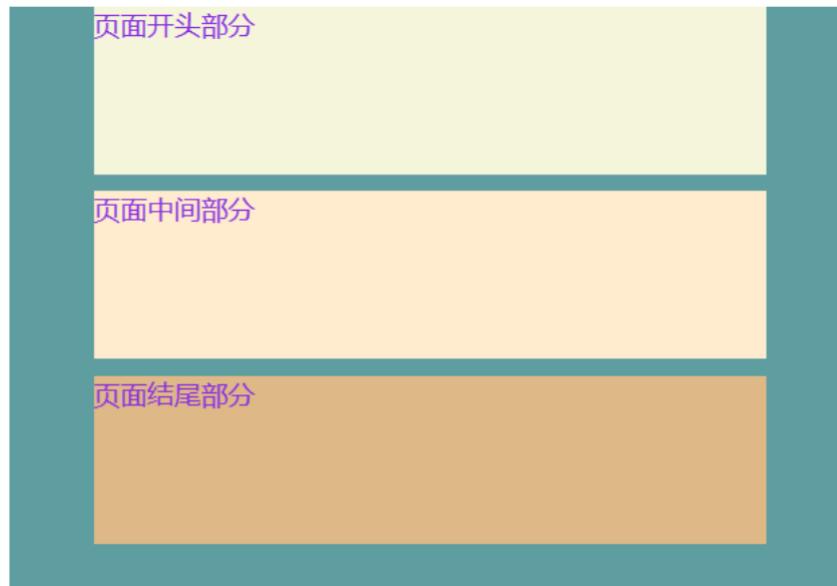
### 1. 布局相关标签

div: 俗称"块"，主要用于划分页面结构，做页面布局。块元素自己独占一行，块元素的CSS样式往往都是生效的。

span: 行内元素，俗称"层"，主要用于划分元素范围，配合CSS做页面元素样式的修饰。行内元素的CSS样式有时不生效。

```
<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>
```

效果



## 2. 特殊符号 (在w3school查找即可)

### HTML 中有用的字符实体

注释：实体名称对大小写敏感！

显示结果	描述	实体名称	实体编号
	空格	&nbsp;	&#160;
<	小于号	&lt;	&#60;
>	大于号	&gt;	&#62;
&	和号	&amp;	&#38;
"	引号	&quot;	&#34;
'	撇号	&apos; (IE不支持)	&#39;
¢	分 (cent)	&cent;	&#162;
£	镑 (pound)	&pound;	&#163;
¥	元 (yen)	&yen;	&#165;
€	欧元 (euro)	&euro;	&#8364;
§	小节	&sect;	&#167;
©	版权 (copyright)	&copy;	&#169;
®	注册商标	&reg;	&#174;
™	商标	&trade;	&#8482;
×	乘号	&times;	&#215;
÷	除号	&divide;	&#247;

# CSS

## 引入方式

### 1. 行内式

通过元素开始标签的style属性引入，样式语法为 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：

- (1) 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。
- (2) 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

通过在head标签中的style标签定义本页面的公共样式。

通过选择器确定样式的作用元素。

```
<head>
    <meta charset="UTF-8">
    <style>
        /* 通过选择器确定样式的作用范围 */
        input {
            display: block;
            width: 80px;
            height: 40px;
            background-color: rgb(140, 235, 100);
            color: white;
            border: 3px solid green;
        }
    </style>
</head>
```

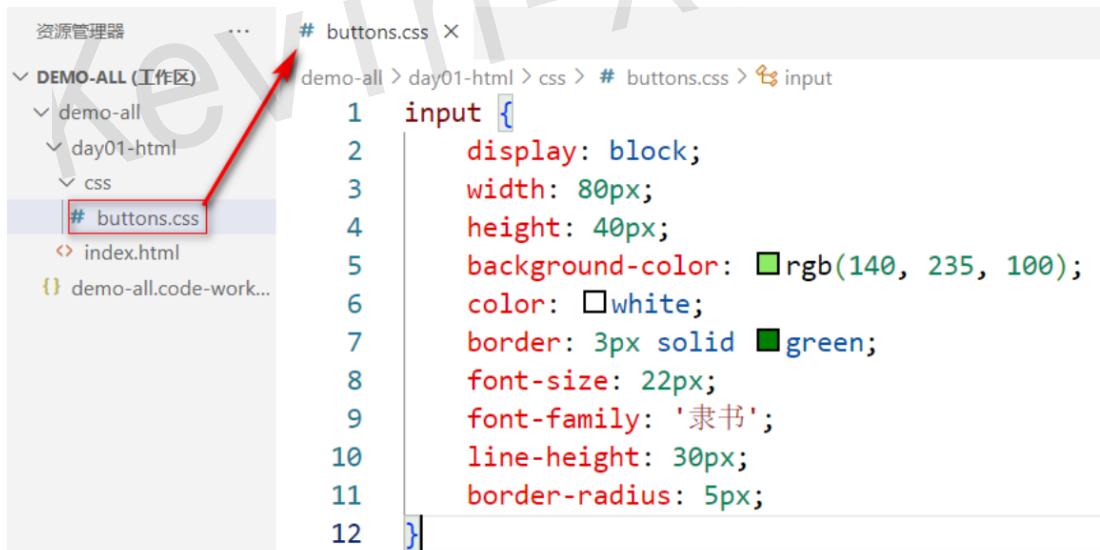
```
        font-size: 22px;
        font-family: '隶书';
        line-height: 30px;
        border-radius: 5px;
    }
</style>
</head>
<body>
    <input type="button" value="按钮1"/>
    <input type="button" value="按钮2"/>
    <input type="button" value="按钮3"/>
    <input type="button" value="按钮4"/>
</body>
```

说明

1. 内嵌式样式需要在**head**标签中，通过一对**style**标签定义**CSS**样式
  2. **CSS**样式的作用范围控制要依赖选择器
  3. **CSS**的样式代码中注释的方式为 `/* */`
  4. 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
  5. 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

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"/>
    <input type="button" value="按钮3"/>
    <input type="button" value="按钮4"/>
</body>
```

## 选择器

### 1. 元素选择器

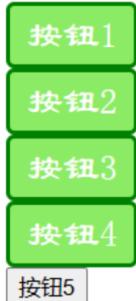
说明

1. 根据标签名确定样式的作用范围
2. 语法为 元素名 {}
3. 样式只能作用到同名标签上，其他标签不可用
4. 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

```
<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"/>
    <input type="button" value="按钮2"/>
    <input type="button" value="按钮3"/>
    <input type="button" value="按钮4"/>
</body>
```

```
<button>按钮5</button>
</body>
```

效果



## 2. id选择器

说明

1. 根据元素 **id** 属性的值确定样式的作用范围
2. 语法为 `#id值 {}`
3. **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
4. 因为 **id** 属性值不够灵活，所以使用该选择器的情况较少

```
<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>
```

效果



### 3. class选择器

说明

1. 根据元素class属性的值确定样式的作用范围
2. 语法为 .class值 {}
3. 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
4. 多个选择器的样式可以在同一个元素上进行叠加
5. 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

```
<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>

```



## 浮动&定位&盒子模型

### 1. 浮动

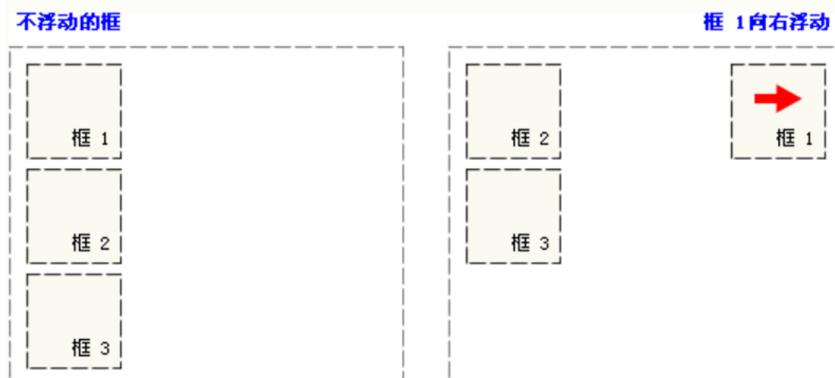
CSS 的 Float (浮动) 使元素脱离文档流 (原本的位置可以被其他元素重新占据) , 按照指定的方向 (左或右发生移动) , 直到它的外边缘碰到包含框或另一个浮动框的边框为止。

- 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。

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。

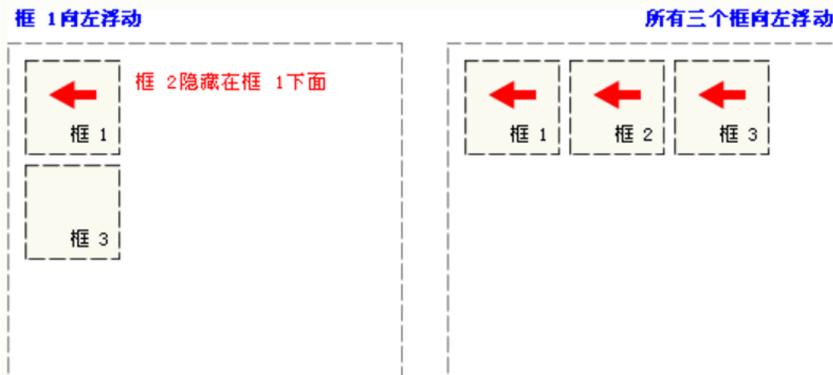
浮动原理：

(1) 当把框 1 向右浮动时, 它脱离文档流并且向右移动, 直到它的右边缘碰到包含框的右边缘。而框 2 和框 3 会随之占据框 1 原来的位置。考虑三个木箱子在水中, 一个向右漂浮后, 下面的箱子会上升。

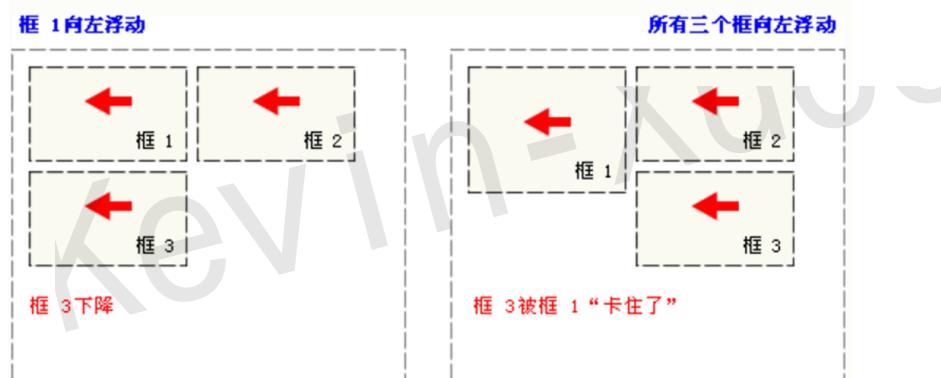


(2) 当框 1 向左浮动时, 它脱离文档流并且向左移动, 直到它的左边缘碰到包含框的左边缘。因为它不再处于文档流中, 所以它不占据空间, 实际上覆盖住了框 2, 使框 2 从视图中消失。但是为了遵循“浮动不会将文字挡住”的初衷, “框 2”这两个字会出现并遮住“框 3”。

如果把所有三个框都设置为向左移动, 那么框 1 向左浮动直到碰到包含框, 另外两个框向左浮动直到碰到前一个浮动框。



(3) 如果包含框太窄, 无法容纳水平排列的三个浮动元素, 那么其它浮动块向下移动, 直到有足够的空间。如果浮动元素的高度不同, 那么当它们向下移动时可能被其它浮动元素“卡住”。



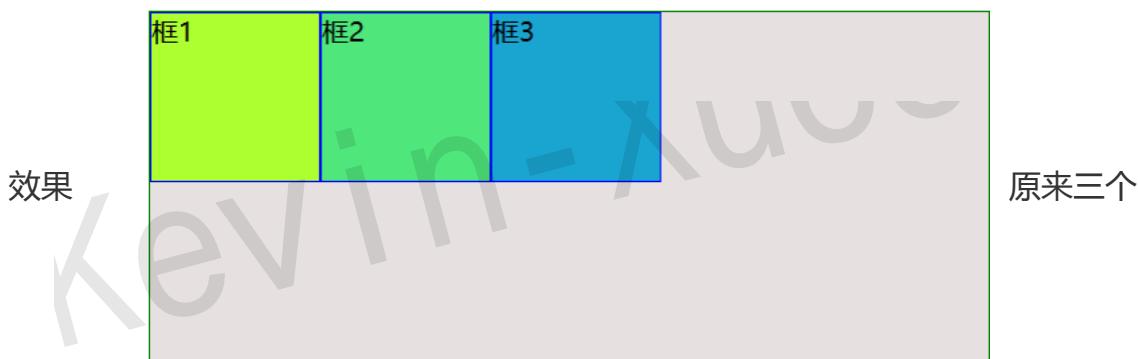
```
<head>
    <meta charset="UTF-8">
    <style>
        .outerDiv {
            width: 500px;
            height: 300px;
            border: 1px solid green;
            background-color: rgb(230, 224, 224);
        }
        .innerDiv{
            width: 100px;
            height: 100px;
            border: 1px solid blue;
            float: left;
        }
    </style>

```

```

.d1{
    background-color: greenyellow;
    /* float: right; */
}
.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>

```



div会占据三行，浮动后在同一行

## 2. 定位

position 属性指定了元素的定位类型。

position属性有如下值

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）。

设置完值后，可以用top bottom left right属性进行定位设置。

(1) static: 不设置的时候的默认值就是static，静态定位，没有定位，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。

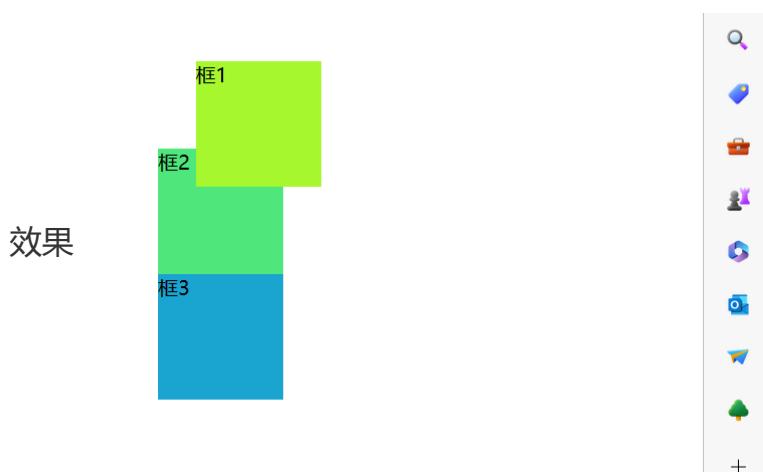
(2) 绝对定位: absolute, 通过 top left right bottom 指定元素在页面上的固定位置；定位后元素会让出原来位置，其他元素可以占用。

```
<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>
```



(3) 相对定位: relative 相对于自己原来的位置进行定位, 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>
```



(4) 固定定位: fixed 定位在浏览器窗口固定位置, 不会随着页面的上下移动而移动 (小广告)。

### 3. 盒子模型

所有HTML元素可以看作盒子，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）。



说明：

1. Margin(外边距) - 清除边框外的区域，外边距是透明的。
2. Border(边框) - 围绕在内边距和内容外的边框。
3. Padding(内边距) - 清除内容周围的区域，内边距是透明的。
4. Content(内容) - 盒子的内容，显示文本和图像。

在设置边距时，盒子的内容大小是不会变的，如设置内边距为10px，由于内容大小不变，整个盒子会拉长10px。

在浏览器上，可以通过F12工具查看盒子模型状态。

The screenshot shows the Chrome DevTools Element tab with three nested div elements labeled '框1', '框2', and '框3'. The 'innerDiv.d2' element is selected. The bottom right corner displays a detailed view of the box model structure:

margin	10
border	0.800
padding	-
content	100x100
total width	100px + 10px + 0.800px + 0.800px = 111.6px
total height	100px + 10px + 0.800px + 0.800px = 111.6px

The DevTools sidebar shows the DOM structure and the computed styles for the selected element, including width: 100px, height: 100px, border: 1px solid blue, float: left, and margin: 10px 20px 30px 40px;.

# JavaScript

## JS基础(引入/数据类型/运算符)

### 1. JS引入方式

#### (1) 内部脚本方式引入

在head中通过一对script标签定义脚本代码。

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <title>小标题</title>
    <style>
      /* 通过选择器确定样式的作用范围 */
      .btn1 {
        display: block;
        width: 150px;
        height: 40px;
        background-color: rgb(245, 241, 129);
        color: rgb(238, 31, 31);
        border: 3px solid rgb(238, 23, 66);
        font-size: 22px;
        font-family: '隶书';
        line-height: 30px;
        border-radius: 5px;
      }
    </style>
    <script>
      function surprise(){
        alert("Hello,我是惊喜")
      }
    </script>
  </head>
  <body>
    <button class="btn1" onclick="surprise()">点我有惊喜
  </button>
  </body>
</html>
```

#### (2) 引入外部脚本文件

在head中通过一对script标签引入外部JS文件（外部JS文件直接写即可，不需要加html标签、script标签等）。一个html文档中，可以有多个script标签。

但一对script标签，要么仅使用内嵌式，要么仅使用外部脚本文件，不能混用。

PS：使用外部引入方式，一对标签间不要加任何空格和换行。

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <title>小标题</title>
    <style>
      /* 通过选择器确定样式的作用范围 */
      .btn1 {
        display: block;
        width: 150px;
        height: 40px;
        background-color: rgb(245, 241, 129);
        color: rgb(238, 31, 31);
        border: 3px solid rgb(238, 23, 66);
        font-size: 22px;
        font-family: '隶书';
        line-height: 30px;
        border-radius: 5px;
      }
    </style>
    <script src="js/button.js" type="text/javascript">
  </script>
  </head>

  <body>
    <button class="btn1" onclick="surprise()">点我有惊喜
  </button>
  </body>
</html>
```

## 2. 数据类型和变量

- (1) 数值类型统一为 number, 不区分整数和浮点数。
- (2) 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'。
- (3) 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。
- (4) JS中的各种函数属于function数据类型。
- (5) 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。

(6) 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。

(7) var声明的变量可以再次声明。

### 3. 运算符

(1) / 在除0时，结果是Infinity；%在模0时，结果是NaN，意思为 not a number，而不是报错。

(2) == 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。

(3) === 符号，如果两端数据类型不一致，直接返回false，数据类型一致在比较是否相同。

## JS基础(流程控制/函数)

### 1. 流程控制

(1) if: if()中的非空字符串会被认为是true，非零数字会被认为是true。

(2) Switch、while、for和Java相同。

```
var monthStr=prompt("请输入月份","例如:10 "); //产生一个弹窗，返回一个字符串
var month= Number.parseInt(monthStr) //将字符串转化为int
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) for each循环迭代数组时，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>") //document.write往浏览器窗口上打印,
console.log往控制台窗口打印
for(var index in cities){
    document.write("<li>" +cities[index]+ "</li>")
}
document.write("</ul>")
```

- 效果
- 北京
  - 上海
  - 深圳
  - 武汉
  - 西安
  - 成都

## 2. 函数

函数声明的语法

```
/*
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);
```

## 函数和Java比较的特点

### 说明

1. 函数没有权限控制符(如public)
2. 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
3. 参数列表中, 无需数据类型
4. 调用函数时, 实参和形参的个数可以不一致
5. 声明函数时需要用function关键字
6. 函数没有异常列表

## 对象和JSON

### 1. 声明对象

#### (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();
```

效果	张小明 10 10岁的张小明喜欢吃: 苹果 橘子 香蕉 葡萄
----	--

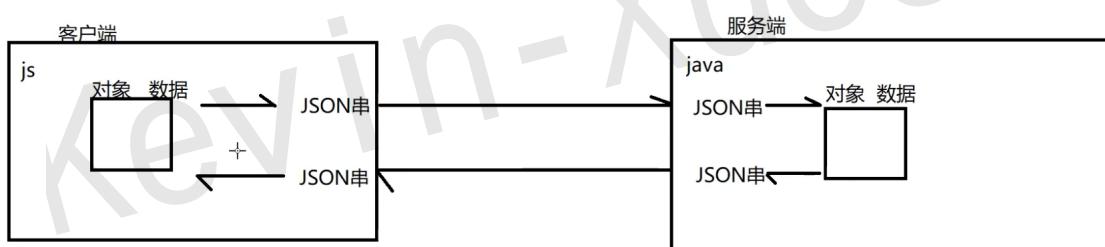
#### (2) 通过{}形式创建对象

语句为 var 对象名 = {"属性名": "属性值", "属性名": "属性值", "函数名": 函数}

```
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();
```

## 2. JSON

JSON 是一种字符串格式，这种格式无论是在前端还是在后端，都可以很容易的转换成对象，所以常用于前后端数据传递。



### JSON格式语法

```
var obj='{"属性名": "属性值", "属性名": {"属性名": "属性值"}, "属性名": ["值1', '值2', '值3"]}'
```

属性名必须用" "，属性值若是字符串必须用" "；属性值可以是数字、数组、对象等。

### JSON串与对象的转换

- (1) 通过JSON.parse()方法可以将一个JSON串转换成对象。
- (2) 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)
```

### 3. JSON在服务端使用

使用Jackson、Fastjson等jar包，可以将对象转化为JSON串

```
public void testMapToJson throws Exception {  
    Person person = new Person("xzs", 22);  
    ObjectMapper objectMapper = new ObjectMapper();  
    String personStr =  
        objectMapper.writeValueAsString(person); //jackson jar包下的方法，对象转JSON  
}
```

map转JSON

```
public void testMapToJson throws Exception {  
    Map data = new HashMap();  
    data.put("a", "valuea");  
    data.put("b", "valueb");  
  
    ObjectMapper objectMapper = new ObjectMapper();  
    String s = objectMapper.writeValueAsString(data);  
    System.out.println(s);  
}
```

List Array转JSON

```
public void testListToJson throws Exception {  
    List data = new ArrayList();  
    data.add("a");  
    data.add("b");  
  
    ObjectMapper objectMapper = new ObjectMapper();  
    String s = objectMapper.writeValueAsString(data);  
    System.out.println(s);  
}
```

### 4. JS常见对象

#### (1) 数组

创建方式

```

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

```

JS的数组更像JAVA中的集合，里面可以是不同的数据类型。

方法	描述
<a href="#"><u>concat()</u></a>	连接两个或更多的数组，并返回结果。
<a href="#"><u>copyWithin()</u></a>	从数组的指定位置拷贝元素到数组的另一个指定位置中。
<a href="#"><u>entries()</u></a>	返回数组的可迭代对象。
<a href="#"><u>every()</u></a>	检测数值元素的每个元素是否都符合条件。
<a href="#"><u>fill()</u></a>	使用一个固定值来填充数组。
<a href="#"><u>filter()</u></a>	检测数值元素，并返回符合条件所有元素的数组。
<a href="#"><u>find()</u></a>	返回符合传入测试（函数）条件的数组元素。
<a href="#"><u>findIndex()</u></a>	返回符合传入测试（函数）条件的数组元素索引。
<a href="#"><u>forEach()</u></a>	数组每个元素都执行一次回调函数。
<a href="#"><u>from()</u></a>	通过给定的对象中创建一个数组。
<a href="#"><u>includes()</u></a>	判断一个数组是否包含一个指定的值。
<a href="#"><u>indexOf()</u></a>	搜索数组中的元素，并返回它所在的位置。
<a href="#"><u>isArray()</u></a>	判断对象是否为数组。
<a href="#"><u>join()</u></a>	把数组的所有元素放入一个字符串。
<a href="#"><u>keys()</u></a>	返回数组的可迭代对象，包含原始数组的键(key)。
<a href="#"><u>lastIndexOf()</u></a>	搜索数组中的元素，并返回它最后出现的位置。
<a href="#"><u>map()</u></a>	通过指定函数处理数组的每个元素，并返回处理后的数组。
<a href="#"><u>pop()</u></a>	删除数组的最后一个元素并返回删除的元素。

方法	描述
<a href="#"><u>push()</u></a>	向数组的末尾添加一个或更多元素，并返回新的长度。
<a href="#"><u>reduce()</u></a>	将数组元素计算为一个值（从左到右）。
<a href="#"><u>reduceRight()</u></a>	将数组元素计算为一个值（从右到左）。
<a href="#"><u>reverse()</u></a>	反转数组的元素顺序。
<a href="#"><u>shift()</u></a>	删除并返回数组的第一个元素。
<a href="#"><u>slice()</u></a>	选取数组的一部分，并返回一个新数组。
<a href="#"><u>some()</u></a>	检测数组元素中是否有元素符合指定条件。
<a href="#"><u>sort()</u></a>	对数组的元素进行排序。
<a href="#"><u>splice()</u></a>	从数组中添加或删除元素。
<a href="#"><u>toString()</u></a>	把数组转换为字符串，并返回结果。
<a href="#"><u>unshift()</u></a>	向数组的开头添加一个或更多元素，并返回新的长度。
<a href="#"><u>valueOf()</u></a>	返回数组对象的原始值。
<a href="#"><u>Array.of()</u></a>	将一组值转换为数组。
<a href="#"><u>Array.at()</u></a>	用于接收一个整数值并返回该索引对应的元素，允许正数和负数。负整数从数组中的最后一个元素开始倒数。
<a href="#"><u>Array.flat()</u></a>	创建一个新数组，这个新数组由原数组中的每个元素都调用一次提供的函数后的返回值组成。
<a href="#"><u>Array.flatMap()</u></a>	使用映射函数映射每个元素，然后将结果压缩成一个新数组。

(2) JS还提供了Boolean对象、Date对象、Math对象、Number对象、String对象，使用方法类似数组，查阅API文档即可。

## 事件

1. 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，我们称之为事件发生。
2. 常见事件

| 鼠标事件

属性	描述
<a href="#"><u>onclick</u></a>	当用户点击某个对象时调用的事件句柄。
<a href="#"><u>oncontextmenu</u></a>	在用户点击鼠标右键打开上下文菜单时触发
<a href="#"><u>ondblclick</u></a>	当用户双击某个对象时调用的事件句柄。
<a href="#"><u>onmousedown</u></a>	鼠标按钮被按下。
<a href="#"><u>onmouseenter</u></a>	当鼠标指针移动到元素上时触发。
<a href="#"><u>onmouseleave</u></a>	当鼠标指针移出元素时触发
<a href="#"><u>onmousemove</u></a>	鼠标被移动。
<a href="#"><u>onmouseover</u></a>	鼠标移到某元素之上。
<a href="#"><u>onmouseout</u></a>	鼠标从某元素移开。
<a href="#"><u>onmouseup</u></a>	鼠标按键被松开。

## 键盘事件

属性	描述
<a href="#"><u>onkeydown</u></a>	某个键盘按键被按下。
<a href="#"><u>onkeypress</u></a>	某个键盘按键被按下并松开。
<a href="#"><u>onkeyup</u></a>	某个键盘按键被松开。

## 表单事件

属性	描述
<a href="#"><u>onblur</u></a>	元素失去焦点时触发

属性	描述
<a href="#"> onchange</a>	<p>该事件在表单元素的内容改变时触发( <input type="text"/> , <input checked="" type="checkbox"/> </p> <p>说明</p> <ol style="list-style-type: none"> <li>1. 通过事件属性绑定函数，在行为发生时会自动执行函数</li> <li>2. 一个事件可以同时绑定多个函数</li> <li>3. 一个元素可以同时绑定多个事件</li> <li>4. 方法中可以传入 <code>this</code> 对象，代表当前元素</li> </ol> <pre> &lt;head&gt;     &lt;meta charset="UTF-8"&gt;     &lt;title&gt;小标题&lt;/title&gt;      &lt;script&gt;         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")         }     &lt;/script&gt; &lt;/head&gt;  &lt;body&gt;     &lt;input type="text"         onkeydown="testDown1(),testDown2()"         onfocus="testFocus()"         onblur="testBlur()"         onchange="testChange(this)"         onmouseover="testMouseOver()"         onmouseleave="testMouseLeave()"         onmousemove="testMouseMove()"     /&gt; &lt;/body&gt; </pre>

## (2) 通过DOM编程动态绑定与触发

### 事件绑定

错误绑定：如下，代码由上自下解释执行，扫描到script时，发现要获得的in1元素还没定义，会产生错误。

```

<head>
    <meta charset="UTF-8">
    <title>小标题</title>

    <script>
```

属性	描述
	<pre>         var in1 =document.getElementById("in1"); /*获得整个id为in1的元素对象*/         in1.onchange=function(){             console.log("内容改变")             console.log(event.target.value);         }     &lt;/script&gt; &lt;/head&gt;  &lt;body&gt;     &lt;input id="in1" type="text" /&gt; &lt;/body&gt; </pre>
正确绑定：使用页面加载事件onload，只有整个页面都加载完时，才会触发该事件	
<pre> &lt;head&gt;     &lt;meta charset="UTF-8"&gt;     &lt;title&gt;小标题&lt;/title&gt;      &lt;script&gt;         function ready(){ /*此时只是定义函数，并未执行*/             var in1 =document.getElementById("in1"); /*获得整个id为in1的元素对象*/             in1.onchange=function(){                 console.log("内容改变")                 console.log(event.target.value);             }         }     &lt;/script&gt; &lt;/head&gt;  &lt;body onload="ready()&gt;     &lt;input id="in1" type="text" /&gt; &lt;/body&gt; </pre>	
<p>简写</p> <pre> &lt;head&gt;     &lt;meta charset="UTF-8"&gt;     &lt;title&gt;小标题&lt;/title&gt;      &lt;script&gt;         window.onload=function(){             var in1 =document.getElementById("in1");             // 通过DOM编程绑定事件             in1.onchange=testChange();             console.log("内容改变")             console.log(event.target.value);         }     &lt;/script&gt; &lt;/head&gt;  &lt;body&gt;     &lt;input id="in1" type="text" /&gt; &lt;/body&gt; </pre>	
事件触发	
<pre> &lt;head&gt;     &lt;meta charset="UTF-8"&gt;     &lt;title&gt;小标题&lt;/title&gt;      &lt;script&gt;         // 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              var btn1 =document.getElementById("btn1"); </pre>	

属性	描述
	<pre>         btn1.onclick=function (){             console.log("按钮点击了")             // 调用事件方法触发事件             in1.onchange()         }     }      function testChange(){         console.log("内容改变")         console.log(event.target.value);     } </pre> <p>&lt;/script&gt;</p> <p>&lt;/head&gt;</p> <p>&lt;body&gt;</p> <p style="padding-left: 40px;">&lt;input id="in1" type="text" /&gt;</p> <p style="padding-left: 40px;">&lt;br&gt;</p> <p style="padding-left: 40px;">&lt;button id="btn1"&gt;按钮&lt;/button&gt;</p> <p>&lt;/body&gt;</p>

## BOM编程

### 1. BOM概述

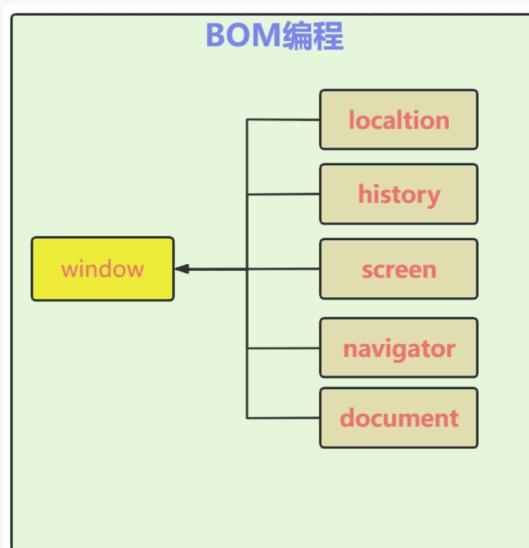
BOM是Browser Object Model的简写，即浏览器对象模型。

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(通过window对象及属性的一系列方法 控制浏览器行为的一种编程)。

### 2. BOM编程对象结构

window	顶级对象,代表整个浏览器窗口
常见属性:	
location对象	window对象的属性之一,代表浏览器的地址栏
history对象	window对象的属性之一,代表浏览器的访问历史
screen对象	window对象的属性之一,代表屏幕
navigator对象	window对象的属性之一,代表浏览器软件本身
document对象	window对象的属性之一,代表浏览器窗口目前解析的html文档
console对象	window对象的属性之一,代表浏览器开发者工具的控制台
localStorage对象	window对象的属性之一,代表浏览器的本地数据持久化存储
sessionStorage对象	window对象的属性之一,代表浏览器的本地数据会话级存储

BOM编程研究的就是window对象及其属性的API能干什么，能控制浏览器的什么属性和行为。



### 3. Window对象的常见API

常见API及window对象属性的常见API都可以查阅文档

方法	描述
<a href="#">alert()</a>	显示带有一段消息和一个确认按钮的警告框。
<a href="#">atob()</a>	解码一个 base-64 编码的字符串。

属性	描述	
	方法	描述
	<a href="#">btoa()</a>	创建一个 base-64 编码的字符串。
	<a href="#">blur()</a>	把键盘焦点从顶层窗口移开。
	<a href="#">clearInterval()</a>	取消由 setInterval() 设置的 timeout。
	<a href="#">clearTimeout()</a>	取消由 setTimeout() 方法设置的 timeout。
	<a href="#">close()</a>	关闭浏览器窗口。
	<a href="#">confirm()</a>	显示带有一段消息以及确认按钮和取消按钮的对话框。
	<a href="#">createPopup()</a>	创建一个 pop-up 窗口。
	<a href="#">focus()</a>	把键盘焦点给予一个窗口。
	<a href="#">getSelection()</a>	返回一个 Selection 对象，表示用户选择的文本范围或光标的当前位置。
	<a href="#">getComputedStyle()</a>	获取指定元素的 CSS 样式。
	<a href="#">matchMedia()</a>	该方法用来检查 media query 语句，它返回一个 MediaQueryList 对象。
	<a href="#">moveBy()</a>	可相对窗口的当前坐标把它移动指定的像素。
	<a href="#">moveTo()</a>	把窗口的左上角移动到一个指定的坐标。
	<a href="#">open()</a>	打开一个新的浏览器窗口或查找一个已命名的窗口。
	<a href="#">print()</a>	打印当前窗口的内容。
	<a href="#">prompt()</a>	显示可提示用户输入的对话框。
	<a href="#">resizeBy()</a>	按照指定的像素调整窗口的大小。
	<a href="#">resizeTo()</a>	把窗口的大小调整到指定的宽度和高度。
	<a href="#">scroll()</a>	已废弃。该方法已经使用了 <a href="#">scrollTo()</a> 方法来替代。
	<a href="#">scrollBy()</a>	按照指定的像素值来滚动内容。
	<a href="#">scrollTo()</a>	把内容滚动到指定的坐标。
	<a href="#">setInterval()</a>	按照指定的周期（以毫秒计）来调用函数或计算表达式。
	<a href="#">setTimeout()</a>	在指定的毫秒数后调用函数或计算表达式。
	<a href="#">stop()</a>	停止页面载入。
	<a href="#">postMessage()</a>	安全地实现跨源通信。

eg：三种弹窗方式

```
window对象由浏览器提供，无需自己new
调用window.某方法，window.可以不写，即直接写方法
1.alert
2.prompt
3.confirm
```

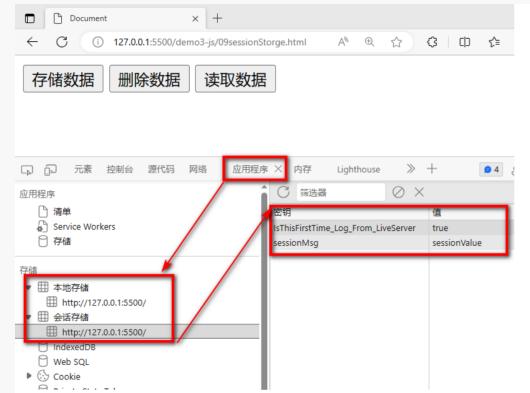
```
<head>
    <meta charset="UTF-8">
    <title>小标题</title>

    <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")
            }
        }
    </script>

```

属性	描述
	<pre>         }     }     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)     } &lt;/script&gt; &lt;/head&gt;  &lt;body&gt;     &lt;input type="button" value="提示框" onclick="testAlert()"/&gt; &lt;br&gt;     &lt;input type="button" value="确认框" onclick="testConfirm()"/&gt; &lt;br&gt;     &lt;input type="button" value="对话框" onclick="testPrompt()"/&gt; &lt;br&gt; &lt;/body&gt; </pre>
	<p>4. window对象属性的API</p> <p>eg: 页面跳转 (修改地址栏)</p> <pre> &lt;head&gt;     &lt;meta charset="UTF-8"&gt;     &lt;title&gt;小标题&lt;/title&gt;      &lt;script&gt;         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"             }         }     &lt;/script&gt; &lt;/head&gt;  &lt;body&gt;     &lt;input type="button" value="跳转到尚硅谷" onclick="goAtguigu()"/&gt; &lt;br&gt; &lt;/body&gt; </pre>
	<p>5. 通过BOM编程实现会话级和持久级数据存储</p> <p>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sessionStorage属性实现。</p> <p>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localStorage实现。</p> <pre> &lt;!DOCTYPE html&gt; &lt;html lang="en"&gt; &lt;head&gt;     &lt;meta charset="UTF-8"&gt;     &lt;meta name="viewport" content="width=device-width, initial-scale=1.0"&gt;     &lt;title&gt;Document&lt;/title&gt;     &lt;script&gt;         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(){ </pre>

属性	描述
	<pre>         console.log("read")         // 读取数据         console.log("session:"+sessionStorage.getItem("sessionMsg"))         console.log("local:"+localStorage.getItem("localMsg"))     }     &lt;/script&gt; &lt;/head&gt; &lt;body&gt;      &lt;button onclick="saveItem()"&gt;存储数据&lt;/button&gt;     &lt;button onclick="removeItem()"&gt;删除数据&lt;/button&gt;     &lt;button onclick="readItem()"&gt;读取数据&lt;/button&gt;  &lt;/body&gt; &lt;/html&gt; </pre>



在F12开发者工具的应用程序栏，可以查看数据的状态

## DOM编程

### 1. DOM编程概述

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。当服务器写完html文件并返回给客户端，浏览器会将html文档解析为document对象。客户端无法修改源代码，此时若要在客户端进行修改，实际上是修改document对象。当document对象发生改变，浏览器会实时监测并作出相应。因此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。

### 2. dom树

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。document的改变实际上就是树中节点的变化，树节点做出怎么样的改变，页面上的内容就会相应做出怎么样的改变。

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

以下面的代码为例

```

<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>

```

其树形结构 (dom树) 为：

属性	描述																																										
	<p style="text-align: center;"><b>Element-html</b></p> <p>DOM编程实际上就分为三步</p> <pre> 1. 获得document 即dom树     window.document  2. 从document中获取要操作的元素     1. 直接获取     2. 间接获取  3. 对元素进行操作     1. 操作元素的属性     2. 操作元素的样式     3. 操作元素的文本     4. 增删元素 </pre> <p>3. DOM编程获取页面元素的方式</p> <p>(1) 直接获取：在整个文档范围内查找元素节点</p> <table border="1"> <thead> <tr> <th>功能</th><th>API</th><th>返回值</th></tr> </thead> <tbody> <tr> <td>根据id值查询</td><td>document.getElementById("id值")</td><td>一个具体的元素节</td></tr> <tr> <td>根据标签名查询</td><td>document.getElementsByTagName("标签名")</td><td>元素节点数组</td></tr> <tr> <td>根据name属性值查询</td><td>document.getElementsByName("name值")</td><td>元素节点数组</td></tr> <tr> <td>根据类名查询</td><td>document.getElementsByClassName("类名")</td><td>元素节点数组</td></tr> </tbody> </table> <p>(2) 间接获取（父找子、子找父、找兄弟）</p> <table border="1"> <thead> <tr> <th>功能</th><th>API</th><th>返回值</th></tr> </thead> <tbody> <tr> <td>查找子标签</td><td>element.children</td><td>返回子标签数组</td></tr> <tr> <td>查找第一个子标签</td><td>element.firstElementChild</td><td>标签对象</td></tr> <tr> <td>查找最后一个子标签</td><td>element.lastElementChild</td><td>标签对象</td></tr> </tbody> </table> <table border="1"> <thead> <tr> <th>功能</th><th>API</th><th>返回值</th></tr> </thead> <tbody> <tr> <td>查找指定元素节点的父标签</td><td>element.parentElement</td><td>标签对象</td></tr> </tbody> </table> <table border="1"> <thead> <tr> <th>功能</th><th>API</th><th>返回值</th></tr> </thead> <tbody> <tr> <td>查找前一个兄弟标签</td><td>node.previousElementSibling</td><td>标签对象</td></tr> <tr> <td>查找后一个兄弟标签</td><td>node.nextElementSibling</td><td>标签对象</td></tr> </tbody> </table> <pre> &lt;!DOCTYPE html&gt; &lt;html lang="en"&gt; &lt;head&gt;     &lt;meta charset="UTF-8"&gt;     &lt;meta name="viewport" content="width=device-width, initial-scale=1.0"&gt;     &lt;title&gt;Document&lt;/title&gt;     &lt;script&gt;         /*         1. 获得document dom树     </pre>	功能	API	返回值	根据id值查询	document.getElementById("id值")	一个具体的元素节	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	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	标签对象	功能	API	返回值	查找指定元素节点的父标签	element.parentElement	标签对象	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
功能	API	返回值																																									
根据id值查询	document.getElementById("id值")	一个具体的元素节																																									
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																																									
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	标签对象																																									
功能	API	返回值																																									
查找指定元素节点的父标签	element.parentElement	标签对象																																									
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																																									

属性	描述
	<pre> 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     2. 操作元素的样式     3. 操作元素的文本     4. 增删元素 */ function fun1(){     //1 获得document     //2 通过document获得元素     var el1 =document.getElementById("username") // 根据元素的id值获取页面上唯一的一个元素     console.log(el1) }  function fun2(){     var els =document.getElementsByTagName("input") // 根据元素的标签名获取多个同名元素     for(var i = 0 ;i&lt;els.length;i++){         console.log(els[i])     } }  function fun3(){     var els =document.getElementsByName("aaa") // 根据元素的name属性值获得多个元素     console.log(els)     for(var i =0;i&lt; els.length;i++){         console.log(els[i])     } }  function fun4(){     var els =document.getElementsByClassName("a") // 根据元素的class属性值获得多个元素     for(var i =0;i&lt; els.length;i++){         console.log(els[i])     } }  function fun5(){     // 先获取父元素     var div01 = document.getElementById("div01")     // 获取所有子元素     var cs=div01.children // 通过父元素获取全部的子元素     for(var i =0;i&lt; cs.length;i++){         console.log(cs[i])     }      console.log(div01.firstElementChild) // 通过父元素获取第一个子元素     console.log(div01.lastElementChild) // 通过父元素获取最后一个子元素 }  function fun6(){     // 获取子元素     var pinput =document.getElementById("password")     console.log(pinput.parentElement) // 通过子元素获取父元素 }  function fun7(){     // 获取子元素     var pinput =document.getElementById("password")     console.log(pinput.previousElementSibling) // 获取前面的第一个元素 } </pre>

属性	描述
	<pre>         console.log(pinput.nextElementSibling) // 获取后面的第一个元素     }     &lt;/script&gt; &lt;/head&gt; &lt;body&gt;     &lt;div id="div01"&gt;         &lt;input type="text" class="a" id="username" name="aaa"/&gt;         &lt;input type="text" class="b" id="password" name="aaa"/&gt;         &lt;input type="text" class="a" id="email"/&gt;         &lt;input type="text" class="b" id="address"/&gt;     &lt;/div&gt;     &lt;input type="text" class="a"/&gt;&lt;br&gt;      &lt;hr&gt;     &lt;input type="button" value="通过父元素获取子元素" onclick="fun5()" id="btn05"/&gt;     &lt;input type="button" value="通过子元素获取父元素" onclick="fun6()" id="btn06"/&gt;     &lt;input type="button" value="通过当前元素获取兄弟元素" onclick="fun7()" id="btn07"/&gt;     &lt;hr&gt;      &lt;input type="button" value="根据id获取指定元素" onclick="fun1()" id="btn01"/&gt;     &lt;input type="button" value="根据标签名获取多个元素" onclick="fun2()" id="btn02"/&gt;     &lt;input type="button" value="根据name属性值获取多个元素" onclick="fun3()" id="btn03"/&gt;     &lt;input type="button" value="根据class属性值获得多个元素" onclick="fun4()" id="btn04"/&gt;  &lt;/body&gt; &lt;/html&gt;</pre>

#### 4. 操作元素属性、样式与文本

##### 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

##### 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

```

<!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
            2. 间接获取
        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
        */
        function changeAttribute(){
            var in1 =document.getElementById("in1")
            // 语法 元素.属性名=""
            // 获得属性值
            console.log(in1.type)
            console.log(in1.value)
        }
    </script>
</head>
<body>
    <input type="text" id="in1" value="123456" />
</body>

```

属性	描述
	<pre> // 修改属性值 in1.type="button" in1.value="嗨" }  function changeStyle(){     var in1 =document.getElementById("in1")     // 语法 元素.style.样式名="" 原始样式名中的"-"符号 要转换驼峰式 background-color &gt; backgroundColor     in1.style.color="green"     in1.style.borderRadius="5px"  }  function changeText(){     var div01 =document.getElementById("div01")     /*     语法 元素名.innerText 只识别文本     元素名.innerHTML 同时可以识别html代码     */     console.log(div01.innerText)     div01.innerHTML="<h1>嗨&lt;/h1&gt;" }  &lt;/script&gt; &lt;style&gt; #in1{     color: red; } &lt;/style&gt; &lt;/head&gt; &lt;body&gt;     &lt;input id="in1" type="text" value="hello"&gt;     &lt;div id="div01"&gt;         hello     &lt;/div&gt;      &lt;hr&gt;     &lt;button onclick="changeAttribute()"&gt;操作属性&lt;/button&gt;     &lt;button onclick="changeStyle()"&gt;操作样式&lt;/button&gt;     &lt;button onclick="changeText()"&gt;操作文本&lt;/button&gt; &lt;/body&gt; &lt;/html&gt; </h1></pre>

## 5. 元素的增删改查

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

```

属性	描述
	<p>2. 间接获取</p> <p>3 对元素进行操作</p> <ol style="list-style-type: none"> <li>1. 操作元素的属性   元素名.属性名=""</li> <li>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</li> <li>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</li> <li>4. 增删元素           <pre>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 */</pre> </li> </ol> <pre>function addcs(){   // 创建一个新的元素   // 创建元素   var csli =document.createElement("li") // &lt;li&gt;&lt;/li&gt;   // 设置子元素的属性和文本 &lt;li id="cs"&gt;长沙&lt;/li&gt;   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") // &lt;li&gt;&lt;/li&gt;   // 设置子元素的属性和文本 &lt;li id="cs"&gt;长沙&lt;/li&gt;   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") // 创建完后状态&lt;li&gt;&lt;/li&gt;   // 设置子元素的属性和文本 &lt;li id="cs"&gt;长沙&lt;/li&gt;   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   } */</pre>

属性	描述
	<pre>         cityul.innerHTML=""         //cityul.remove()      }  &lt;/script&gt;  &lt;/head&gt; &lt;body&gt;     &lt;ul id="city"&gt;         &lt;li id="bj"&gt;北京&lt;/li&gt;         &lt;li id="sh"&gt;上海&lt;/li&gt;         &lt;li id="sz"&gt;深圳&lt;/li&gt;         &lt;li id="gz"&gt;广州&lt;/li&gt;     &lt;/ul&gt;      &lt;hr&gt;     &lt;!-- 目标1 在城市列表的最后添加一个子标签 &lt;li id="cs"&gt;长沙&lt;/li&gt; --&gt;     &lt;button onclick="addCs()"&gt;增加长沙&lt;/button&gt;     &lt;!-- 目标2 在城市列表的深圳前添加一个子标签 &lt;li id="cs"&gt;长沙&lt;/li&gt; --&gt;     &lt;button onclick="addCsBeforeSz()"&gt;在深圳前插入长沙&lt;/button&gt;     &lt;!-- 目标3 将城市列表的深圳替换为 &lt;li id="cs"&gt;长沙&lt;/li&gt; --&gt;     &lt;button onclick="replacesz()"&gt;替换深圳&lt;/button&gt;     &lt;!-- 目标4 将城市列表删除深圳 --&gt;     &lt;button onclick="removeSz()"&gt;删除深圳&lt;/button&gt;     &lt;!-- 目标5 清空城市列表 --&gt;     &lt;button onclick="clearCity()"&gt;清空&lt;/button&gt;  &lt;/body&gt; &lt;/html&gt; </pre>

## 正则表达式

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]+\$/

## XML

### 1. XML: XML是可扩展标记语言

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。将来我们主要就是根据XML约束中的规定来编写XML配置文件，而且会在我们编写XML的时候根据约束来提示我们编写，而XML约束主要包括DTD和Schema两种。

以web.xml的约束声明为例，尖括号内都是框架规定好的约束：

```

<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">

```

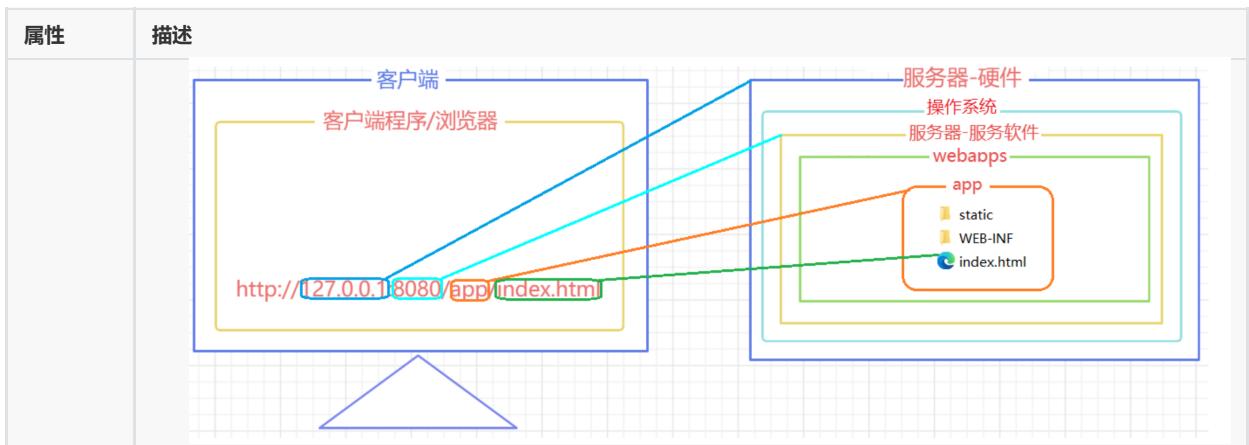
### 2. 常见配置文件

- (1) properties文件，例如druid连接池就是使用properties文件作为配置文件。
- (2) XML文件，例如Tomcat就是使用XML文件作为配置文件。
- (3) YAML文件，例如SpringBoot就是使用YAML作为配置文件。
- (4) json文件，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。

对比properties文件与XML文件的格式：

属性	描述
	<pre>atguigu.jdbc.url=jdbc:mysql://localhost:3306/atguigu atguigu.jdbc.driver=com.mysql.cj.jdbc.Driver atguigu.jdbc.username=root atguigu.jdbc.password=root</pre> <p><b>properties</b>语法规规范 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</p> <pre>&lt;?xml version="1.0" encoding="UTF-8"?&gt; &lt;students&gt;     &lt;student&gt;         &lt;name&gt;张三&lt;/name&gt;         &lt;age&gt;18&lt;/age&gt;     &lt;/student&gt;     &lt;student&gt;         &lt;name&gt;李四&lt;/name&gt;         &lt;age&gt;20&lt;/age&gt;     &lt;/student&gt; &lt;/students&gt;</pre> <h2>Tomcat</h2> <ol style="list-style-type: none"> <li>Web服务器</li> </ol> <p>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虚拟的url地址供网络中的其他人访问。</p> <p>Tomcat就是一款web服务器。如果把APP比作子弹，Tomcat就是枪，只有在Tomcat里APP才能正确运行。</p> <ol style="list-style-type: none"> <li>开启与关闭</li> </ol> <p>在Tomcat的文件夹下，找到/bin/startup.bat，打开后即可运行。关闭窗口即停止。</p> <ol style="list-style-type: none"> <li>Tomcat目录</li> </ol> <p>bin: 该目录下存放的是二进制可执行文件。</p> <p>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：</p> <ul style="list-style-type: none"> <li>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</li> <li>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。可以按照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；</li> </ul> <pre>&lt;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"&gt;     &lt;role rolename="admin-gui"/&gt;     &lt;role rolename="admin-script"/&gt;     &lt;role rolename="manager-gui"/&gt;     &lt;role rolename="manager-script"/&gt;     &lt;role rolename="manager-jmx"/&gt;     &lt;role rolename="manager-status"/&gt;     &lt;user username="admin"&gt;</pre>

属性	描述
	<pre>         password="admin"         roles="admin-gui,admin-script,manager-gui,manager-script,manager- jmx,manager-status"       /&gt; &lt;/tomcat-users&gt; </pre> <p>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!</p> <ul style="list-style-type: none"> <li>context.xml: 对所有应用的统一配置, 通常我们不会去配置它。</li> </ul> <p>lib: Tomcat的类库, 里面是一大堆jar文件。</p> <p>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。</p> <p>temp: 存放Tomcat的临时文件。</p> <p>webapps: 存放web项目的目录, 其中每个文件夹都是一个项目。</p> <p>work: 运行时生成的文件, 最终运行的文件都在这里。</p> <p>4. web项目的标准结构</p> <p>一个可以部署到 tomcat/webapps 中的标准的app目录结构</p> <p>部署目录 app</p> <p>WEB-INF(受保护的资源目录, 不可以通过浏览器直接访问的资源目录)</p> <p>classes 字节码根路径 lib 依赖的存放路径 web.xml 项目的配置文件</p> <p>C:\Program4java\apache-tomcat-10.1.7\webapps</p> <p>app 本应用根目录</p> <ol style="list-style-type: none"> <li>(1) 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)。</li> <li>(2) 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。</li> </ol> <ul style="list-style-type: none"> <li>■ classes 必要目录, src下源代码, 配置文件, 编译后会在该目录下, web项目中如果没有源码, 则该目录不会出现</li> <li>■ 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</li> <li>■ 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</li> </ul> <ol style="list-style-type: none"> <li>(3) index.html 非必要文件, index.html/index.htm/index.jsp为默认的欢迎页。</li> </ol> <p>5. url的组成部分和项目中资源的对应关系</p>



## HTTP

### 简介

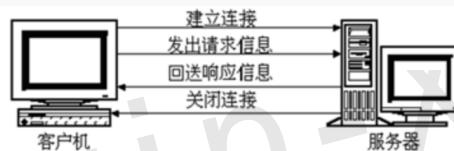
1. HTTP 超文本传输协议 (HTTP-Hyper Text transfer protocol)，是一个属于应用层的面向对象的协议，由于其简捷、快速的方式，适用于分布式超媒体信息系统。它于1990年提出，经过十几年的使用与发展，得到不断地完善和扩展。它是一种详细规定了浏览器和万维网服务器之间互相通信的规则，通过因特网传送万维网文档的数据传送协议。客户端与服务端通信时传输的内容我们称之为报文。HTTP就是一个通信规则，这个规则规定了客户端发送给服务器的报文格式，也规定了服务器发给客户端的报文格式。实际我们要学习的就是这两种报文。客户端发送给服务器的称为“请求报文”，服务器发给客户端的称为“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

#### 2. HTTP交互方式

请求：由客户端向服务端发送，请求时发送的数据称为请求报文。

响应：由服务端向客户端返回，响应时返回的数据称为响应报文。

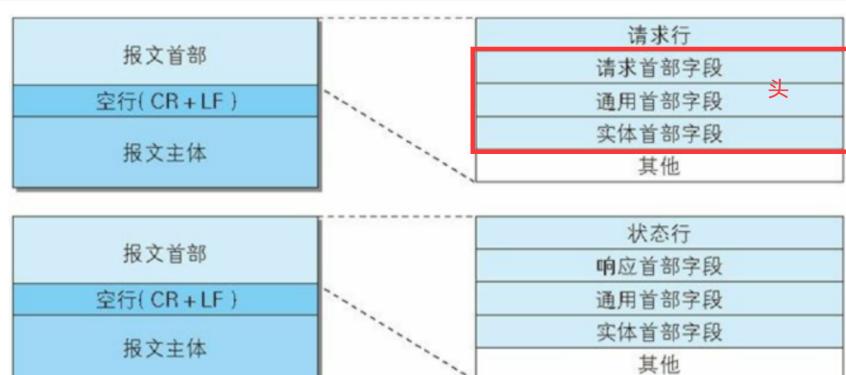
#### 3. HTTP协议的会话方式



## 请求和响应报文

### 1. 报文格式

报文被分为首部和主体，其中首部又能被细分为行和头。



### 2. 请求报文

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； POST请求才有请求体

属性	描述
	<p>▼ 请求标头</p> <p style="text-align: right;">协议格式</p> <p>已分析视图</p> <pre> GET /web01_war_exploded/ HTTP/1.1 Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7 Accept-Encoding: gzip, deflate, br Accept-Language: zh-CN,zh;q=0.9,en;q=0.8,en-US;q=0.7 Cache-Control: max-age=0 Connection: keep-alive Host: localhost:8080 If-Modified-Since: Fri, 14 Apr 2023 07:58:38 GMT If-None-Match: W/"158-1681459118058" Sec-Fetch-Dest: document Sec-Fetch-Mode: navigate Sec-Fetch-Site: none Sec-Fetch-User: ?1 Upgrade-Insecure-Requests: 1 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/112.0.0.0 Safari/537.36 Edg/112.0.1722.39 </pre> <p>(1) GET请求</p> <p>请求行：请求方式+资源路径拼接参数+协议版本</p> <pre>GET /05_web_tomcat/login_success.html?username=admin&amp;password=123213 HTTP/1.1</pre> <p>请求头（具体用到时查阅即可）：</p> <pre> - 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 </pre> <p>请求体：GET请求无请求体。</p> <p>(2) POST请求</p> <p>请求行：请求方式+资源路径+协议版本</p> <pre>POST /05_web_tomcat/login_success.html HTTP/1.1</pre> <p>请求头</p> <pre> Host: localhost:8080 Connection: keep-alive Content-Length: 31      -请求体内容的长度 Cache-Control: max-age=0 -无缓存 Origin: http://localhost:8080 Upgrade-Insecure-Requests: 1 -协议的自动升级 Content-Type: application/x-www-form-urlencoded -请求体内容类型[服务器根据类型解析请求体参数] User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/68.0.3440.75 Safari/537.36 Accept:text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8 Referer: http://localhost:8080/05_web_tomcat/login.html Accept-Encoding: gzip, deflate, br Accept-Language: zh-CN,zh;q=0.9,en-US;q=0.8,en;q=0.7 Cookie: JSESSIONID= </pre> <p>请求体：浏览器提交给服务器的数据（参数或上传的文件等）</p> <pre>username=admin&amp;password=1232131</pre> <p>3. 响应报文</p> <p>响应行：协议版本+状态码+状态码描述</p>

属性	描述																																																			
	<p><b>HTTP/1.1 200 OK</b>  说明：响应协议为<b>HTTP1.1</b>，响应状态码为<b>200</b>，表示请求成功；</p> <p><b>响应头</b></p> <pre>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](MIME类型) Content-Length: 157    响应体内容的字节数，检查响应体中数据，若两者不匹配，说明数据传输时丢失 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 Keep-Alive: timeout=20 规定时间后断开连接</pre> <p><b>响应体</b></p> <p>是以01串组成的数据，但根据响应头中的Content-Type，会将01串进行对应的解析，将解析后的数据展现出来。</p> <h3>常见响应状态码</h3> <ol style="list-style-type: none"> <li>常见响应状态码 <ul style="list-style-type: none"> <li><b>200</b>: 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；</li> <li><b>302</b>: 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；</li> <li><b>304</b>: 使用了本地缓存</li> <li><b>404</b>: 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；</li> <li><b>405</b>: 请求的方式不允许</li> <li><b>500</b>: 请求资源找到了，但服务器内部出现了错误；</li> </ul> </li> <li>更多响应状态码</li> </ol> <table border="1"> <thead> <tr> <th>状态码</th><th>状态码英文描述</th><th>中文含义</th></tr> </thead> <tbody> <tr> <td>1**</td><td></td><td></td></tr> <tr> <td>100</td><td>Continue</td><td>继续。客户端应继续其请求</td></tr> <tr> <td>101</td><td>Switching Protocols</td><td>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</td></tr> <tr> <td>2**</td><td></td><td></td></tr> <tr> <td>200</td><td>OK</td><td>请求成功。一般用于GET与POST请求</td></tr> <tr> <td>201</td><td>Created</td><td>已创建。成功请求并创建了新的资源</td></tr> <tr> <td>202</td><td>Accepted</td><td>已接受。已经接受请求，但未处理完成</td></tr> <tr> <td>203</td><td>Non-Authoritative Information</td><td>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</td></tr> <tr> <td>204</td><td>No Content</td><td>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</td></tr> <tr> <td>205</td><td>Reset Content</td><td>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</td></tr> <tr> <td>206</td><td>Partial Content</td><td>部分内容。服务器成功处理了部分GET请求</td></tr> <tr> <td>3**</td><td></td><td></td></tr> <tr> <td>300</td><td>Multiple Choices</td><td>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</td></tr> <tr> <td>301</td><td>Moved Permanently</td><td>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</td></tr> <tr> <td>302</td><td>Found</td><td>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</td></tr> <tr> <td>303</td><td>See Other</td><td>查看其它地址。与301类似。使用GET和POST请求查看</td></tr> </tbody> </table>	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
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																																																		

属性	描述		
	状态码	状态码英文描述	中文含义
	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**		
	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
	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

属性	描述
	<h2>Servlet</h2> <h3>简介&amp;Servlet运行流程</h3> <p>1. 静态资源与动态资源</p> <p>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。</p> <p>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。</p> <p>2. Servlet简介与运行流程</p> <p>Servlet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（承前启后）。</p> <p>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。</p> <p>运行流程：</p> <pre>     graph LR         Browser[浏览器] -- "请求行 /aaa 请求头 请求体" --&gt; Tomcat[Tomcat]         Tomcat -- "1 HttpServletRequest" --&gt; Service["2 /aaa class Servlet1 implements Servlet service(HttpServletRequest request, HttpServletResponse response)"]         Tomcat -- "2 HttpServletResponse" --&gt; Service         Service -- "3 从request对象中获取请求的所有信息(参数) 根据参数生成要响应给客户端的数据 将响应的数据放入 response对象" --&gt; Response[响应的报文]     </pre> <p>浏览器：只知道发送请求报文和接收响应报文，并不知道Tomcat如何处理。</p> <p>Tomcat：</p> <ol style="list-style-type: none"> <li>(1) 接收到请求后，会将请求报文的信息转换为一个HttpServletRequest对象，该对象中包含了请求中的所有信息。</li> <li>(2) Tomcat同时创建了一个HttpServletResponse对象，该对象用于承载要响应给客户端的信息。后面，该对象会被转换成响应的报文。</li> <li>(3) 我们需要写一个Servlet类，这个类需要实现Servlet接口，在这个类中需要重写Servlet接口里的service方法。同时要为写的这个类定义访问路径，让Tomcat能根据路径调用这个类。</li> <li>(4) Tomcat根据请求中的资源路径找到对应的servlet，将servlet实例化，调用service方法，同时将HttpServletRequest和HttpServletResponse对象传入。</li> <li>(5) 将对象传入后，在service方法中能做的事如上图紫色字体所示。service里的代码由我们自己定义，Tomcat执行后将HttpServletResponse对象转换成响应的报文，返回给浏览器。</li> </ol> <p>整个流程Tomcat都能自动执行，程序员需要做的是写service方法，并将Servlet类放入Tomcat服务器中。</p> <h3>Servlet开发流程</h3> <p>1. 开发流程</p> <ol style="list-style-type: none"> <li>(1) 创建JavaWeb项目，同时将Tomcat添加为当前项目的依赖。如果不在Tomcat环境中，servlet类将会报错。</li> <li>(2) 重写service方法，使用的是service(HttpServletRequest req, HttpServletResponse resp)。</li> <li>(3) 在service方法中，定义业务处理代码。</li> <li>(4) 在web.xml文件中，配置service对应的请求映射路径。（此处能用下一节注解代替）</li> </ol> <p>2. 实例</p> <p>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。</p> <ol style="list-style-type: none"> <li>(1) 开发一个web类型的module</li> <li>(2) (3) 开发一个UserServlet</li> </ol> <pre> public class UserServlet extends HttpServlet {     @Override     protected void service(HttpServletRequest req, HttpServletResponse resp) throws     ServletException, IOException {     } } </pre>

属性	描述
	<pre>// 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"); }  }</pre> <p>(4) 在web.xml为UserServlet配置请求的映射路径</p> <pre>&lt;?xml version="1.0" encoding="UTF-8"?&gt; &lt;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"&gt;      &lt;servlet&gt;         &lt;!--给UserServlet起一个别名--&gt;         &lt;servlet-name&gt;userServlet&lt;/servlet-name&gt; &lt;!--给类起别名，用于关联请求的映射路径--&gt;         &lt;servlet-class&gt;com.atguigu.servlet.UserServlet&lt;/servlet-class&gt; &lt;!--写字节码的全路径，告诉Tomcat对应的要实例化的servlet类--&gt;         &lt;/servlet&gt;          &lt;servlet-mapping&gt;             &lt;!--关联别名和映射路径--&gt;             &lt;servlet-name&gt;userServlet&lt;/servlet-name&gt;             &lt;!--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--&gt;             &lt;url-pattern&gt;/userServlet&lt;/url-pattern&gt;             &lt;!-- &lt;url-pattern&gt;/userServlet2&lt;/url-pattern&gt;--&gt;             &lt;!--                 /                 /* 表示通配所有资源，不包括jsp文件                 /*                 /a/*                 *.action 匹配所有以action为后缀的映射路径             --&gt;             &lt;!-- &lt;url-pattern&gt;/*&lt;/url-pattern&gt;--&gt;         &lt;/servlet-mapping&gt;      &lt;/web-app&gt;</pre> <p>url-pattern标签用于定义Servlet的请求映射路径 一个servlet-mapping可以对应多个不同的url-pattern，一个servlet可以对应多个不同的servlet-mapping；多个servlet不能使用相同的url-pattern。</p> <p>3. url路径匹配解释</p> <p>在浏览器输入路径<a href="http://localhost:8080/demo02/s1">http://localhost:8080/demo02/s1</a>是如何找到Servlet类的：</p> <p>localhost找到本机服务器，8080端口对应Tomcat服务器，demo02找到demo02模块，/s1先匹配到url-pattern，再找到servlet-mapping中的别名，根据别名找到servlet标签中的全路径，用反射找到全路径对应的文件并执行。</p>

属性	描述
<b>Servlet注解方式配置</b>	

1. 针对开发流程xml配置，可以用@WebServlet注解代替

注解主要用到五个属性

```

@WebServlet(
    name = "userServlet", //注解属性，相当于xml中起别名
    //value = "/user", value和urlPatterns属性作用相同
    urlPatterns = {"/*userServlet1","/*userServlet2","/*userServlet"}, //配置路径
    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");
        }
    }
}

```

## Servlet生命周期

### 1. 生命周期

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。

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

### 2. Servlet线程安全问题

(1) Servlet对象在容器中是单例的。

(2) 当多个用户向Tomcat发起请求，每个请求在容器中都会开启一个线程。

(3) Servlet的成员变量在多个线程栈之中是共享的，所以在Servlet中，我们不要轻易定义一些容易经常发生修改的成员变量。

@WebServlet中有一个属性load-on-startup，值默认是-1，含义是Tomcat启动时不会实例化该Servlet；设置为正整数时，该值越小表示越先被实例化，如果序号产生冲突，Tomcat会自动协调启动顺序。Tomcat容器中，已经定义了一些随系统启动实例化的servlet，我们自定义的servlet的load-on-startup尽量不要占用数字1-5。

### 3. default-Servlet

当给出的访问路径在Tomcat中没有找到对应的Servlet类，就会交给default-Servlet处理。比如静态资源，都是由default-Servlet处理。

## Servlet继承结构

### 1. Servlet接口

Servlet接口是最顶级的，由其他实现类来实现。接口里的方法如下：

属性	描述
	<pre>//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 void init(ServletConfig config) throws ServletException;  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  /* 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 void service(ServletRequest req, ServletResponse res) throws ServletException, IOException;  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</pre>

## 2. GenericServlet类

GenericServlet类是一个抽象类，实现了Servlet接口。侧重处理service方法以外的方法。

```
2 抽象的类 GenericServlet
public abstract class GenericServlet implements Servlet {
    private transient ServletConfig config;

    public void destroy() {
        // 将抽象方法，重写为普通方法，在方法内部没有任何的实现代码
        // 半成品实现
    }
    // tomcat在调用init方法时，会读取配置信息进入一个ServletConfig对象并将该对象传入init方法
    public void init(ServletConfig config) throws ServletException {
        // 将config对象存储为当前的属性
        this.config = config;
        // 调用了重载的无参的init
        this.init();
    }
    // 重载的初始化方法，我们重写初始化方法时对应的方法
    public void init() throws ServletException {
    }
    // 返回ServletConfig的方法
    public ServletConfig getServletConfig() {
        return this.config;
    }
    // 再次抽象声明service方法
    public abstract void service(ServletRequest var1, ServletResponse var2) throws ServletException, IOException;
}
```

## 3. HttpServlet类

HttpServlet也是一个抽象类，继承自GenericServlet。侧重service方法的处理。

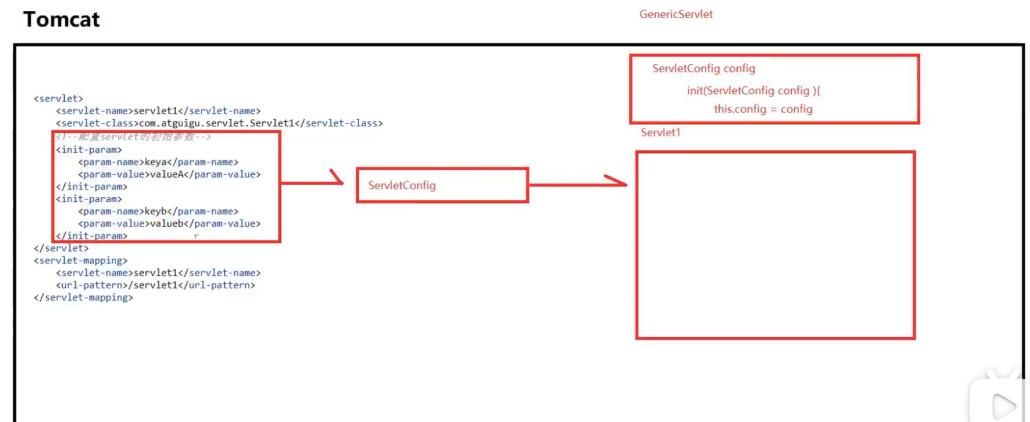
HttpServlet类里有两个service方法，其中我们写的调用的是public void service(HttpServletRequest req, HttpServletResponse res)。在HttpServlet类里的这个方法中，会调用一系列的doGet、doPost等方法实现响应。  
说明：

- (1) 部分程序员推荐在Servlet中重写do\*\*\*方法处理请求，理由是HttpServlet的service作了其他一些处理，如果直接重写service方法，那些其他处理失效。
- (2) 目前直接重写service也没有什么问题。
- (3) 后续使用SpringMVC后，我们则无需继承HttpServlet，处理请求的方法也无需是service和do\*\*\*方法。

总结：继承HttpServlet后，要么重写service，要么重写do\*\*\*。

## ServletConfig

1. ServletConfig是为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。



如上图，容器会为每个Servlet实例化一个ServletConfig对象。在xml文件或者注解中写完初始配置，初始配置会在生命周期的init()被赋给Servlet的ServletConfig对象。

属性	描述																		
	<p>ServletConfig获取初始配置信息：</p> <table border="1"> <thead> <tr> <th>方法名</th><th>作用</th></tr> </thead> <tbody> <tr> <td>getServletName()</td><td>获取&lt;servlet-name&gt;HelloServlet&lt;/servlet-name&gt;定义的Servlet名称</td></tr> <tr> <td>getServletContext()</td><td>获取ServletContext对象</td></tr> <tr> <td>getInitParameter()</td><td>获取配置Servlet时设置的『初始化参数』，根据名字获取值</td></tr> <tr> <td>getInitParameterNames()</td><td>获取所有初始化参数名组成的Enumeration对象</td></tr> </tbody> </table>	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								
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																		
<b>HttpServletRequest</b>																			
1. 获取请求行信息相关(方式,请求的url,协议及版本)																			
<table border="1"> <thead> <tr> <th>API</th><th>功能解释</th></tr> </thead> <tbody> <tr> <td>StringBuffer getRequestURL();</td><td>获取客户端请求的url</td></tr> <tr> <td>String getRequestURI();</td><td>获取客户端请求项目中的具体资源</td></tr> <tr> <td>int getServerPort();</td><td>获取客户端发送请求时的端口</td></tr> <tr> <td>int getLocalPort();</td><td>获取本应用在所在容器的端口</td></tr> <tr> <td>int getRemotePort();</td><td>获取客户端程序的端口</td></tr> <tr> <td>String getScheme();</td><td>获取请求协议</td></tr> <tr> <td>String getProtocol();</td><td>获取请求协议及版本号</td></tr> <tr> <td>String getMethod();</td><td>获取请求方式</td></tr> </tbody> </table>		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
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																		
<p>URI：统一资源标识符，项目内的资源路径（包下的路径）。</p> <p>URL：统一资源定位符，项目内资源的完整路径（网址访问路径）。</p> <p>serverport：如果有中间代理服务器，端口就是代理服务器的端口。</p>																			
2. 获得请求头信息相关																			
<table border="1"> <thead> <tr> <th>API</th><th>功能解释</th></tr> </thead> <tbody> <tr> <td>String getHeader(String headerName);</td><td>根据头名称获取请求头</td></tr> <tr> <td>Enumeration getHeaderNames();</td><td>获取所有的请求头名字</td></tr> <tr> <td>String getContentType();</td><td>获取content-type请求头</td></tr> </tbody> </table>		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										
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																		
3. 获得请求参数相关																			
<table border="1"> <thead> <tr> <th>API</th><th>功能解释</th></tr> </thead> <tbody> <tr> <td>String getParameter(String parameterName);</td><td>根据请求参数名获取请求单个参数值</td></tr> <tr> <td>String[] getParameterValues(String parameterName);</td><td>根据请求参数名获取请求多个参数值数组</td></tr> <tr> <td>Enumeration getParameterNames();</td><td>获取所有请求参数名</td></tr> <tr> <td>Map&lt;String, String[]&gt; getParameterMap();</td><td>获取所有请求参数的键值对集合</td></tr> <tr> <td>BufferedReader getReader() throws IOException;</td><td>获取读取请求体的字符输入流</td></tr> <tr> <td>ServletInputStream getInputStream() throws IOException;</td><td>获取读取请求体的字节输入流</td></tr> <tr> <td>int getContentLength();</td><td>获得请求体长度的字节数</td></tr> </tbody> </table>		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		
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																		
4. 其他API																			
<table border="1"> <thead> <tr> <th>API</th><th>功能解释</th></tr> </thead> <tbody> <tr> <td>String getServletPath();</td><td>获取请求的Servlet的映射路径</td></tr> <tr> <td>ServletContext getServletContext();</td><td>获取ServletContext对象</td></tr> <tr> <td>Cookie[] getCookies();</td><td>获取请求中的所有cookie</td></tr> </tbody> </table>		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										
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																		

属性	描述	
	API	功能解释
	HttpSession getSession();	获取Session对象
	void setCharacterEncoding(String encoding);	设置请求体字符集
<b>HttpServletResponse</b>		
1. 设置响应行相关		
	API	功能解释
	void setStatus(int code);	设置响应状态码
2. 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 setContent-Type(String contentType);	设置content-type响应头及响应字符集(设置MIME类型)
3. 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
4. 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
<b>请求转发和响应重定向</b>		
1. 概述		
	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。	
类比:		
请求转发: 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: 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。		
2. 请求转发		
	<pre> graph LR     Client[客户端] --&gt; Tomcat[Tomcat]     subgraph Tomcat [Tomcat]         direction TB         subgraph app [app]             direction LR             SA[ServletA] --- SB[ServletB]         end         Client -- "请求" --&gt; SA         SA --&gt; SB         SB -- "响应" --&gt; Client     end </pre>	
请求转发特点(背诵)		
	<ul style="list-style-type: none"> <li>■ 请求转发通过HttpServletRequest对象获取请求转发器实现</li> </ul>	

属性	描述
	<ul style="list-style-type: none"> <li>■ 请求转发是服务器内部的行为，对客户端是屏蔽的</li> <li>■ 客户端只发送了一次请求，客户端地址栏不变</li> <li>■ 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</li> <li>■ 因为全程只有一个HttpServletRequest对象，所以请求参数可以传递，请求域中的数据也可以传递</li> <li>■ 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</li> <li>■ 请求转发可以转发给WEB-INF下受保护的资源</li> <li>■ 请求转发不能转发到本项目以外的外部资源</li> </ul> <pre> @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，入参是转发的资源路径         // 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);     } } </pre> <p>3. 响应重定向</p> <p>响应重定向特点(背诵)</p> <ul style="list-style-type: none"> <li>■ 响应重定向通过HttpServletResponse对象的sendRedirect方法实现</li> <li>■ 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</li> <li>■ 客户端至少发送了两次请求，客户端地址栏是要变化的</li> <li>■ 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</li> <li>■ 因为全程产生了多个HttpServletRequest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</li> <li>■ 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</li> <li>■ 重定向不可以到给WEB-INF下受保护的资源</li> <li>■ 重定向可以到本项目以外的外部资源</li> </ul> <p>ServletA</p> <pre> @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");     } } </pre>

属性	描述
	<pre>         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");     } } </pre>

### 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");
    }
}

```

## MVC架构模式

### 1. 概念

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。

### 2. 三层具体功能

M: Model 模型层

- (1) 存放和数据库对象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

- (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

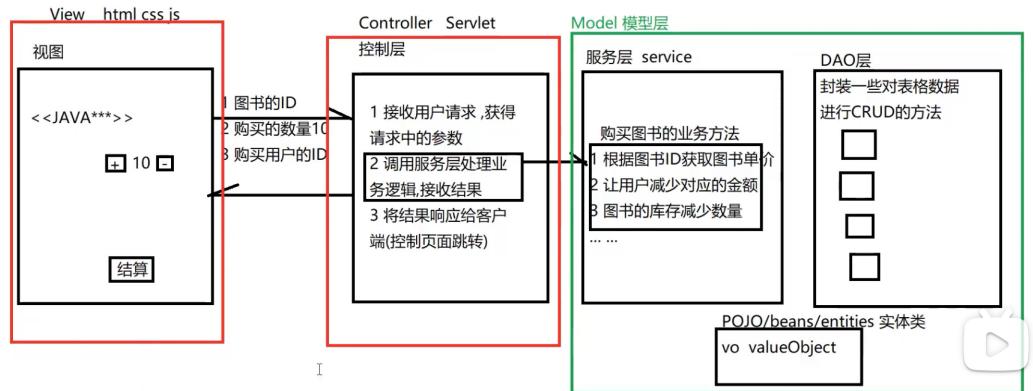
- (1) 接收客户端请求，获得请求数据
- (2) 将准备好的数据响应给客户端

### 3. MVC模式下项目中的常见包

M: 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；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；服务包(service)，专门存放对数据进行业务逻辑运算的一些类。

C: 控制层包(controller)

V: web目录下的视图资源 html css js img 等；前端工程化后,在后端项目中已经不存在了。



会话

概述

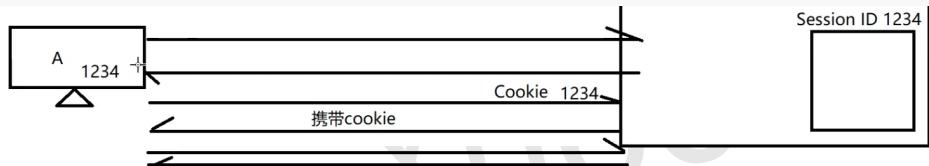
## 1. 会话和会话管理

会话是指客户端和服务端之间的多次请求与响应。

会话管理管理的是客户端与服务端之间的通信状态。HTTP协议自身不对请求和响应之间的通信状态进行保存，是无状态协议，因此需要进行会话管理。

## 2. 会话管理的实现

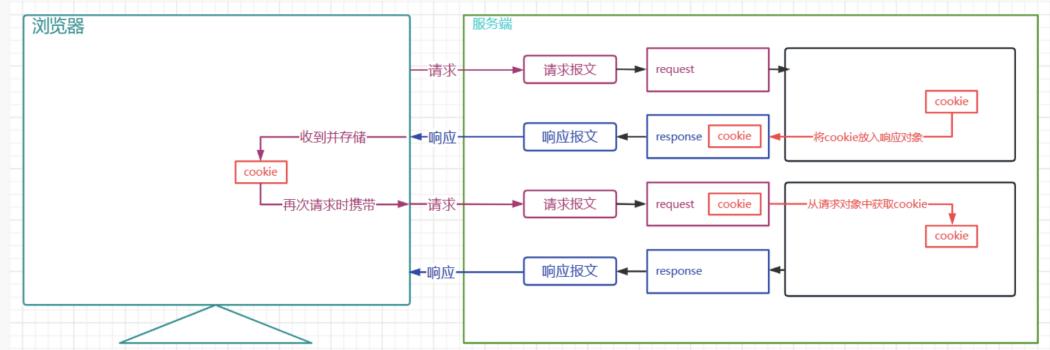
会话管理通过session和cookie实现。客户端第一次发起响应，服务端会生成一个session对象，并返回cookie。此后客户端发起响应，若携带cookie，则认为是同一次会话。



## Cookie

## 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，这样服务器会认为是同一次会话。

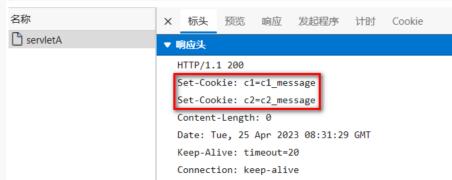


服务端创建cookie，将cookie放入响应对象中；客户端在收到cookie的响应头时，在下次请求该服务的资源时，会以cookie请求头的形式携带之前收到的Cookie。

## 2. Cookie使用

servletA向响应中增加Cookie

属性	描述
	<pre>@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放入响应回对象         resp.addCookie(cookie1);         resp.addCookie(cookie2);     } }</pre>



此后访问任何路径都会携带这两个cookie

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, 当没有设置任何cookie时, cookies是null
        Cookie[] cookies = req.getCookies();
        // 遍历cookies数组
        if (null != cookies && cookies.length!= 0) {
            for (Cookie cookie : cookies) {
                System.out.println(cookie.getName()+":"+cookie.getValue());
            }
        }
    }
}
```

### 3. Cookie时效性

(1) 会话Cookie：默认情况下，服务端没有指定Cookie存在时间，在浏览器端Cookie存在内存中。浏览器未关闭前，Cookie数据一直在，关闭浏览器后，内存中的Cookie就被释放。

(2) 持久化Cookie：服务器端设置Cookie存在时间。在浏览器端，Cookie的数据会被保存到硬盘上，直到预设的时间到达。

通过cookie的setMaxAge()方法让Cookie持久化保存到浏览器上，默认单位是秒。

```
@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放入响应回对象
        resp.addCookie(cookie1);
        resp.addCookie(cookie2);
    }
}
```

### 4. Cookie提交路径

当设置完cookie后，此后在同一次会话访问任何资源，默认都会携带设置的cookie。

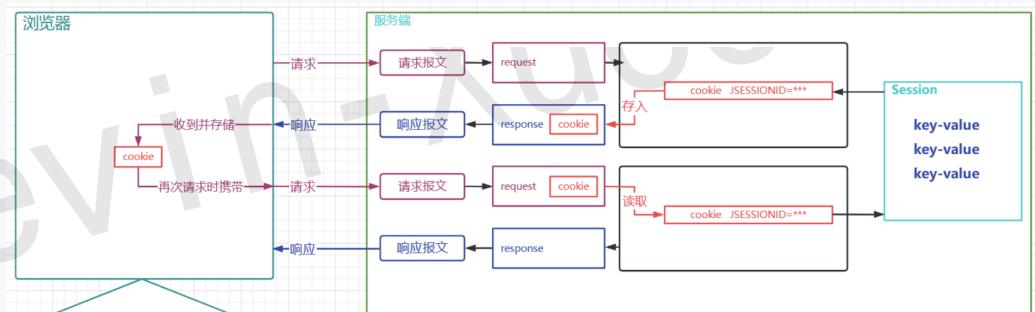
对cookie的提交路径进行设置，则只有访问设置的路径时，会携带相应的cookie。

属性	描述
	<pre>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);     } }</pre> <p style="text-align: center;"></p>

## Session

### 1. Session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。一个Session对象对应一个客户端会话。



服务端在为客户端创建session时，会同时将session对象的id，即SESSIONID以cookie的形式放入响应对象（即cookie只是一小份数据，session对象可以存更多数据，最后把sessionid当做那一小份数据传回客户端）。

后端创建完session后，客户端会收到一个特殊的cookie，叫做SESSIONID；客户端下一次请求时携带SESSIONID，后端收到后，根据SESSIONID找到对应的session对象。

session可以用来存放一些敏感信息。

### 2. Session使用

举例：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,获取之前存储的用户

定义表单页,提交用户名,提交后

```
<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

```
@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
    }
}
```

属性	描述
	<pre>         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);      } } </pre>



响应中收到了一个JSESSIONID的cookie

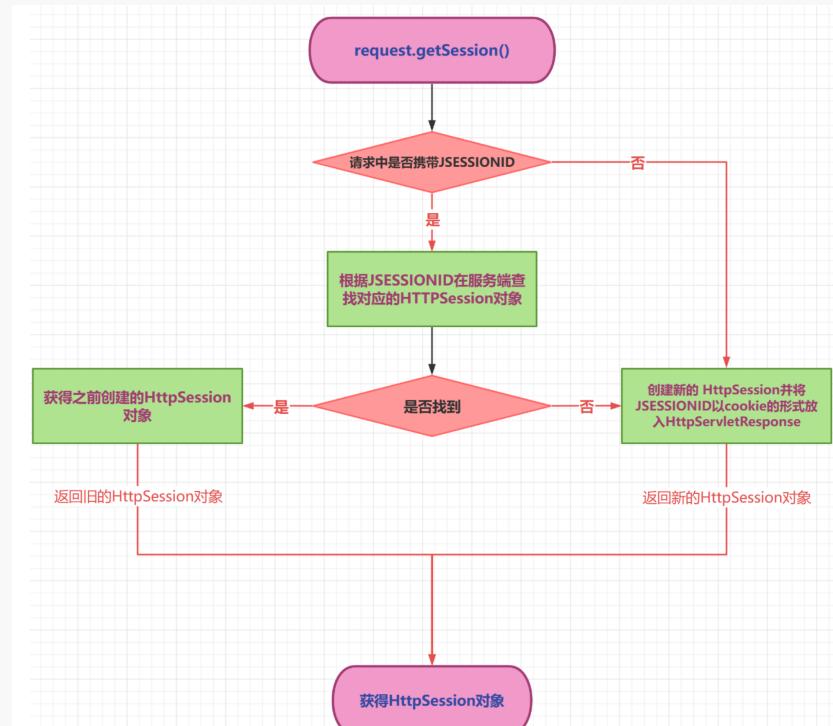
定义其他Servlet,从session中读取用户名

```

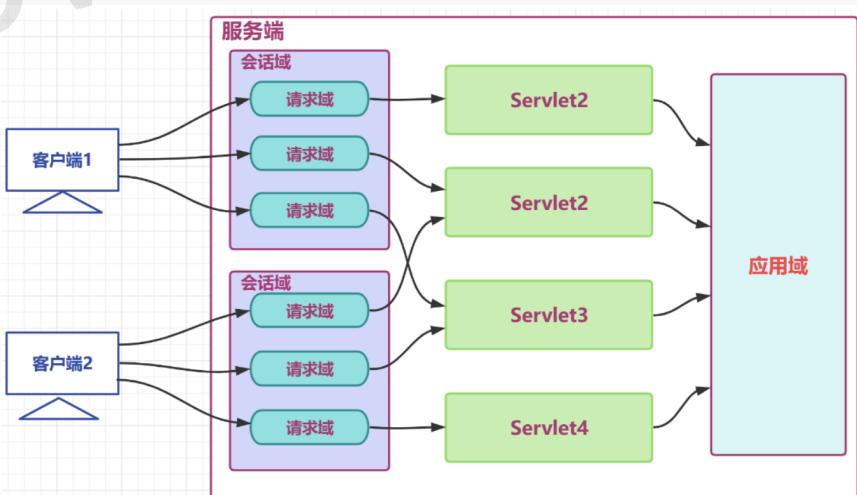
@WebServlet("/servletB")
public class ServletB extends HttpServlet {
    @Override
    protected void service(HttpServletRequest req, HttpServletResponse resp) throws
    ServletException, IOException {
        // 获取session对象
        HttpSession session = req.getSession();
        // 获取session的ID
        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);
    }
}

```

不同于Cookie需要自己new一个Cookie对象, Session对象用get方法获得, 其处理逻辑如下:



当获得的是旧的HttpSession对象, 认为是同一次会话。

属性	描述								
	<p>3. Session时效性</p> <p>设置原因</p> <p>(1) 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。</p> <p>(2) 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, 此时需要释放。</p> <p>设置方法</p> <p>(1) 在当前项目的web.xml对最大闲置时间进行重新设定 (默认30分钟)</p>  <p>(2) 通过HttpSession的API对最大闲置时间进行设定</p> <pre>// 设置最大闲置时间 session.setMaxInactiveInterval(60);</pre> <p>(3) 也可以直接让session失效</p> <pre>// 直接让session失效 session.invalidate();</pre> <h2>域对象</h2> <p>1. 三大域对象</p> <p>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。</p> <p>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。</p> <p>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。</p> <p>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。</p>  <p>2. 域对象API</p> <table border="1"> <thead> <tr> <th>API</th><th>功能</th></tr> </thead> <tbody> <tr> <td>void setAttribute(String name, String value)</td><td>向域对象中添加/修改数据</td></tr> <tr> <td>Object getAttribute(String name);</td><td>从域对象中获取数据</td></tr> <tr> <td>removeAttribute(String name);</td><td>移除域对象中的数据</td></tr> </tbody> </table>	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	removeAttribute(String name);	移除域对象中的数据
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								
removeAttribute(String name);	移除域对象中的数据								

属性	描述								
	<h2>过滤器</h2> <h3>概述&amp;使用&amp;生命周期</h3> <p>1. 概述</p> <p>过滤器过滤的是请求，其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。因此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。</p> <p>其工作位置如图</p>								
	<p>2. 使用</p> <p>过滤器的使用分为三步：实现Filter接口；重写过滤方法；配置过滤器。</p> <p>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。</p> <p>(1) Filter接口</p> <pre>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() {     } }</pre>								
	<p>API目标</p> <table border="1"> <thead> <tr> <th>API</th> <th>目标</th> </tr> </thead> <tbody> <tr> <td>default public void init(FilterConfig filterConfig)</td> <td>初始化方法,由容器调用并传入初始配置信息filterConfig对象</td> </tr> <tr> <td>public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)</td> <td>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</td> </tr> <tr> <td>default public void destroy()</td> <td>销毁方法,容器在回收过滤器对象之前调用的方法</td> </tr> </tbody> </table>	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
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								
	<p>(2) 过滤方法doFilter</p>								

属性	描述
	<p>服务端</p> <pre> graph LR     Client[客户端1] -- "请求报文" --&gt; Request[Request]     Request -- 1 --&gt; Filter[Filter]     subgraph Service [服务端]         Filter         Target[目标资源]     end     Filter -- "doFilter" --&gt; PreProc[放行前功能代码]     PreProc -- 2 --&gt; Chain[chain.doFilter]     Chain -- 3 --&gt; Target     Target -- 4 --&gt; Response[Response]     Response -- 5 --&gt; Client     </pre> <p>doFilter方法里也分为三步：</p> <ol style="list-style-type: none"> <li>1. 请求到达目标资源之前的功能代码 判断是否登录 校验权限是否满足 .....</li> <li>2. 放行代码</li> <li>3. 响应代码 HttpServletResponse转换为响应报文之前的功能代码</li> </ol> <p>以开发一个日志记录过滤器为例</p> <pre> package com.atguigu.filters;  import jakarta.servlet.*; import jakarta.servlet.annotation.WebFilter; import jakarta.servlet.http.HttpServletRequest; import jakarta.servlet.http.HttpServletResponse;  import java.io.IOException; import java.text.SimpleDateFormat; import java.util.Date; 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, 若不放行, 则无法到达后续资源         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);     } } </pre>

属性	描述
	<p>说明</p> <p>1.<code>doFilter</code>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<code>HttpServletRequest</code>和<code>HttpServletResponse</code>子接口级别的,可以安全强转。</p> <p>2.<code>filterChain.doFilter(request, response);</code>; 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。</p> <p>3.<code>filterChain.doFilter(request, response);</code>; 在放行时需要传入<code>request</code>和<code>response</code>,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<code>request</code>和<code>response</code>对象。</p> <p>(3) 注解配置过滤器</p> <p>@WebFilter, 最常用的参数为urlPatterns, 配置要经过该过滤器的资源。</p> <pre> package com.atguigu.filters;  import jakarta.servlet.*; import jakarta.servlet.annotation.WebFilter; import jakarta.servlet.annotation.WebInitParam; import jakarta.servlet.http.HttpServletRequest; import jakarta.servlet.http.HttpServletResponse;  import java.io.IOException; import java.text.SimpleDateFormat; import java.util.Date;  @WebFilter(     filterName = "loggingFilter",     initParams = {@WebInitParam(name="dateTimePattern",value="yyyy-MM-dd HH:mm:ss")},     urlPatterns = {"/*servletA", "*.html"},     servletNames = {"servletBName"} ) public class LoggingFilter implements Filter {     private SimpleDateFormat dateFormat;      /*init初始化方法,通过filterConfig获取初始化参数      * init方法中,可以用于定义一些其他初始化功能代码      */     @Override     public void init(FilterConfig filterConfig) throws ServletException {         // 获取初始参数         String dateTimePattern = filterConfig.getInitParameter("dateTimePattern");         // 初始化成员变量         dateFormat=new SimpleDateFormat(dateTimePattern);     }      @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();         String afterLogging =requestURI+"在"+time+"的请求耗时:"+ (t2-t1)+"毫秒";         // 打印日志         System.out.println(afterLogging);     } } </pre>

属性	描述		
	}		
3. 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

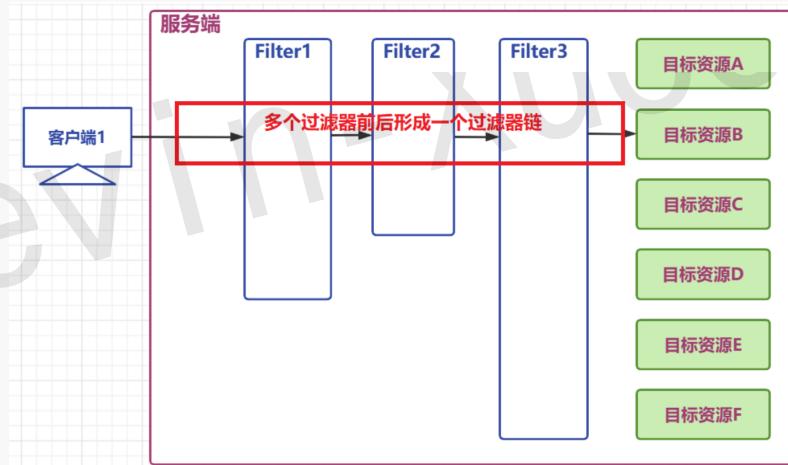
## 过滤器链

### 1. 过滤器链

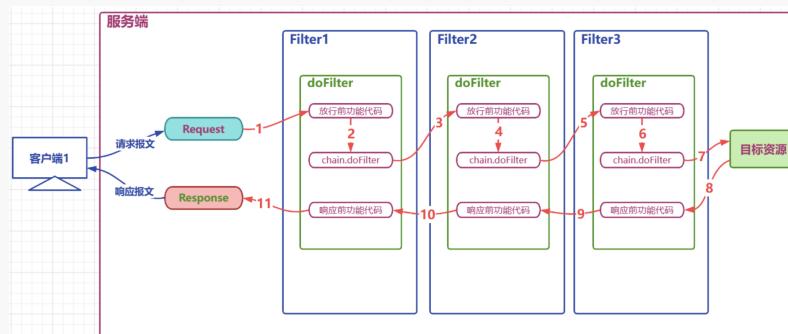
一个web项目中，可以同时定义多个过滤器，多个过滤器对同一个资源进行过滤时，工作位置有先后，整体形成一个工作链，称之为过滤器链。

若用xml进行配置，过滤器链中的过滤器的顺序由filter-mapping顺序决定；若用注解配置，过滤顺序由项目结构中的过滤器名决定（谁名字在前面先执行谁）。

不同资源使用的过滤器可能不同，如图



### 2. 过滤器链工作流程



## 监听器

### 1. 概述

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。

#### 分类

- (1) application域监听器 ServletContextListener ServletContextAttributeListener

属性	描述																																												
	<p>(2) session域监听器 HttpSessionListener HttpSessionAttributeListener HttpSessionBindingListener HttpSessionActivationListener</p> <p>(3) request域监听器 ServletRequestListener ServletRequestAttributeListener</p> <p>2. 监听器接口</p> <p>(1) application域监听器</p> <p>ServletContextListener 监听ServletContext对象的创建与销毁</p> <table border="1"> <thead> <tr> <th>方法名</th><th>作用</th></tr> </thead> <tbody> <tr> <td>contextInitialized(ServletContextEvent sce)</td><td>ServletContext创建时调用</td></tr> <tr> <td>contextDestroyed(ServletContextEvent sce)</td><td>ServletContext销毁时调用</td></tr> </tbody> </table> <ul style="list-style-type: none"> <li>■ 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。</li> </ul> <p>ServletContextAttributeListener 监听ServletContext中属性的添加、移除和修改</p> <table border="1"> <thead> <tr> <th>方法名</th><th>作用</th></tr> </thead> <tbody> <tr> <td>attributeAdded(ServletContextAttributeEvent scab)</td><td>向ServletContext中添加属性时调用</td></tr> <tr> <td>attributeRemoved(ServletContextAttributeEvent scab)</td><td>从ServletContext中移除属性时调用</td></tr> <tr> <td>attributeReplaced(ServletContextAttributeEvent scab)</td><td>当ServletContext中的属性被修改时调用</td></tr> </tbody> </table> <ul style="list-style-type: none"> <li>■ ServletContextAttributeEvent对象代表属性变化事件，它包含的方法如下：</li> </ul> <table border="1"> <thead> <tr> <th>方法名</th><th>作用</th></tr> </thead> <tbody> <tr> <td>getName()</td><td>获取修改或添加的属性名</td></tr> <tr> <td>getValue()</td><td>获取被修改或添加的属性值</td></tr> <tr> <td>getServletContext()</td><td>获取ServletContext对象</td></tr> </tbody> </table> <p>(2) session域监听器</p> <p>HttpSessionListener 监听HttpSession对象的创建与销毁</p> <table border="1"> <thead> <tr> <th>方法名</th><th>作用</th></tr> </thead> <tbody> <tr> <td>sessionCreated(HttpSessionEvent hse)</td><td>HttpSession对象创建时调用</td></tr> <tr> <td>sessionDestroyed(HttpSessionEvent hse)</td><td>HttpSession对象销毁时调用</td></tr> </tbody> </table> <ul style="list-style-type: none"> <li>■ 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。</li> </ul> <p>HttpSessionAttributeListener 监听HttpSession中属性的添加、移除和修改</p> <table border="1"> <thead> <tr> <th>方法名</th><th>作用</th></tr> </thead> <tbody> <tr> <td>attributeAdded(HttpSessionBindingEvent se)</td><td>向HttpSession中添加属性时调用</td></tr> <tr> <td>attributeRemoved(HttpSessionBindingEvent se)</td><td>从HttpSession中移除属性时调用</td></tr> <tr> <td>attributeReplaced(HttpSessionBindingEvent se)</td><td>当HttpSession中的属性被修改时调用</td></tr> </tbody> </table> <ul style="list-style-type: none"> <li>■ HttpSessionBindingEvent对象代表属性变化事件，它包含的方法如下：</li> </ul> <table border="1"> <thead> <tr> <th>方法名</th><th>作用</th></tr> </thead> <tbody> <tr> <td>getName()</td><td>获取修改或添加的属性名</td></tr> <tr> <td>getValue()</td><td>获取被修改或添加的属性值</td></tr> <tr> <td>getSession()</td><td>获取触发事件的HttpSession对象</td></tr> </tbody> </table> <p>HttpSessionBindingListener 监听当前监听器对象在Session域中的增加与移除</p>	方法名	作用	contextInitialized(ServletContextEvent sce)	ServletContext创建时调用	contextDestroyed(ServletContextEvent sce)	ServletContext销毁时调用	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	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	方法名	作用	sessionCreated(HttpSessionEvent hse)	HttpSession对象创建时调用	sessionDestroyed(HttpSessionEvent hse)	HttpSession对象销毁时调用	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	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
方法名	作用																																												
contextInitialized(ServletContextEvent sce)	ServletContext创建时调用																																												
contextDestroyed(ServletContextEvent sce)	ServletContext销毁时调用																																												
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																																												
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																																												
方法名	作用																																												
sessionCreated(HttpSessionEvent hse)	HttpSession对象创建时调用																																												
sessionDestroyed(HttpSessionEvent hse)	HttpSession对象销毁时调用																																												
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																																												
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																																												

属性	描述																																										
	<table border="1"> <thead> <tr> <th>方法名</th><th>作用</th></tr> </thead> <tbody> <tr> <td>valueBound(HttpSessionBindingEvent event)</td><td>该类的实例被放到Session域中时调用</td></tr> <tr> <td>valueUnbound(HttpSessionBindingEvent event)</td><td>该类的实例从Session中移除时调用</td></tr> </tbody> </table> <p>■ HttpSessionBindingEvent对象代表属性变化事件，它包含的方法如下：</p> <table border="1"> <thead> <tr> <th>方法名</th><th>作用</th></tr> </thead> <tbody> <tr> <td>getName()</td><td>获取当前事件涉及的属性名</td></tr> <tr> <td>getValue()</td><td>获取当前事件涉及的属性值</td></tr> <tr> <td>getSession()</td><td>获取触发事件的HttpSession对象</td></tr> </tbody> </table> <p>HttpSessionActivationListener 监听某个对象在Session中的序列化与反序列化。</p> <table border="1"> <thead> <tr> <th>方法名</th><th>作用</th></tr> </thead> <tbody> <tr> <td>sessionWillPassivate(HttpSessionEvent se)</td><td>该类实例和Session一起钝化到硬盘时调用</td></tr> <tr> <td>sessionDidActivate(HttpSessionEvent se)</td><td>该类实例和Session一起活化到内存时调用</td></tr> </tbody> </table> <p>■ HttpSessionEvent对象代表事件对象，通过getSession()方法获取事件涉及的HttpSession对象。</p> <p>(3) request域监听器</p> <p>ServletRequestListener 监听ServletRequest对象的创建与销毁</p> <table border="1"> <thead> <tr> <th>方法名</th><th>作用</th></tr> </thead> <tbody> <tr> <td>requestInitialized(ServletRequestEvent sre)</td><td>ServletRequest对象创建时调用</td></tr> <tr> <td>requestDestroyed(ServletRequestEvent sre)</td><td>ServletRequest对象销毁时调用</td></tr> </tbody> </table> <p>■ 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。</p> <p>ServletRequestAttributeListener 监听ServletRequest中属性的添加、移除和修改</p> <table border="1"> <thead> <tr> <th>方法名</th><th>作用</th></tr> </thead> <tbody> <tr> <td>attributeAdded(ServletRequestAttributeEvent srae)</td><td>向ServletRequest中添加属性时调用</td></tr> <tr> <td>attributeRemoved(ServletRequestAttributeEvent srae)</td><td>从ServletRequest中移除属性时调用</td></tr> <tr> <td>attributeReplaced(ServletRequestAttributeEvent srae)</td><td>当ServletRequest中的属性被修改时调用</td></tr> </tbody> </table> <p>■ ServletRequestAttributeEvent对象代表属性变化事件，它包含的方法如下：</p> <table border="1"> <thead> <tr> <th>方法名</th><th>作用</th></tr> </thead> <tbody> <tr> <td>getName()</td><td>获取修改或添加的属性名</td></tr> <tr> <td>getValue()</td><td>获取被修改或添加的属性值</td></tr> <tr> <td>getServletRequest ()</td><td>获取触发事件的ServletRequest对象</td></tr> </tbody> </table>	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	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	方法名	作用	sessionWillPassivate(HttpSessionEvent se)	该类实例和Session一起钝化到硬盘时调用	sessionDidActivate(HttpSessionEvent se)	该类实例和Session一起活化到内存时调用	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	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	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
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																																										
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																																										
方法名	作用																																										
sessionWillPassivate(HttpSessionEvent se)	该类实例和Session一起钝化到硬盘时调用																																										
sessionDidActivate(HttpSessionEvent se)	该类实例和Session一起活化到内存时调用																																										
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																																										
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																																										
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																																										

## Ajax

### 1. Ajax简介

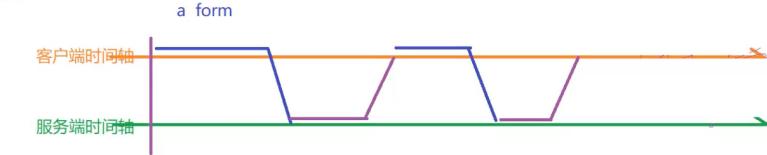
AJAX = Asynchronous JavaScript and XML (异步的JavaScript 和 XML)。

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。

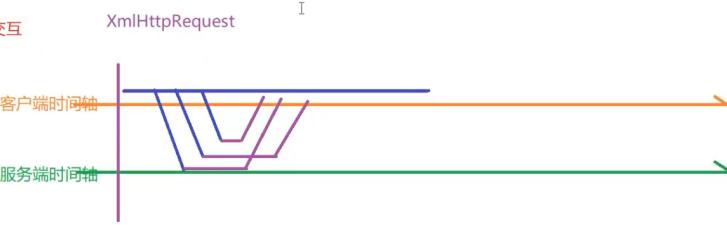
AJAX 不需要任何浏览器插件，但需要用户允许 JavaScript 在浏览器上执行。

同步与异步：

## 同步交互



## 异步交互

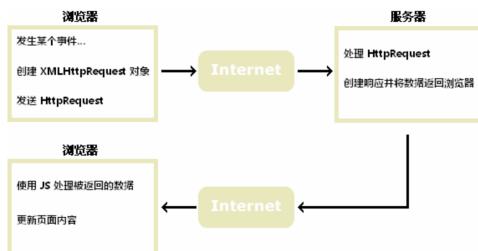


同步：当客户端发起请求，需要等服务端处理完请求后，才能继续使用客户端。

异步：客户端发起请求后能继续前进，服务端处理完后返回，客户端根据服务端处理完后的先后顺序进行响应。

## 2. Ajax实现

### (1) XMLHttpRequest



```
<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>
```

### (2) 使用框架VUE中axios