# JavaWeb

## Html

常见标签的使用

## CSS

### 盒子模型

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，边框，填充，和实际内容。

下面的图片说明了盒子模型(Box Model)：


不同部分的说明：

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

### 选择器

1. 元素选择器

```
p {
 color: red;
}
```

2. id选择器

```
#content {
 color: red;
}
```

3. class选择器

```
.content {
 color: red;
}
```

# JS

JavaScript是一种**基于对象(Object)**和**事件驱动(Event Driven)**并具有安全性能的脚本语言。

## 数据类型

字符串（String）、数字(Number)、布尔(Boolean)、数组(Array)、对象(Object)、空（Null）、未定义（Undefined）。

## 选择器

1. 元素选择器

```
var array = document.getElementsByTagName("p")
```

2. id选择器

```
var one = document.getElementById("id");
```

3. class选择器

```
var array = document.getElementsByClassName("class");
```

4. 元素名选择器

```
var array = document.getElementsByName("name");
```

## 正则表达式

正则表达式（英语：Regular Expression，在代码中常简写为regex、regexp或RE）使用单个字符串来描述、匹配一系列符合某个句法规则的字符串搜索模式。

### 符号

| 元字符 | 描述 |
| --- | --- |
| \ | 将下一个字符标记符、或一个向后引用、或一个八进制转义符。例如，“\n”匹配\n。“\n”匹配换行符。序列“\”匹配“\”而“(”则匹配“(”。即相当于多种编程语言中都有的“转义字符”的概念。 |
| ^ | 匹配输入字行首。如果设置了RegExp对象的Multiline属性，^也匹配“\n”或“\r”之后的位置。 |
| $ | 匹配输入行尾。如果设置了RegExp对象的Multiline属性，$也匹配“\n”或“\r”之前的位置。 |
| * | 匹配前面的子表达式任意次。例如，zo*能匹配“z”，也能匹配“zo”以及“zoo”。*等价于o{0,} |
| + | 匹配前面的子表达式一次或多次(大于等于1次）。例如，“zo+”能匹配“zo”以及“zoo”，但不能匹配“z”。+等价于{1,}。 |
| ? | 匹配前面的子表达式零次或一次。例如，“do(es)?”可以匹配“do”或“does”中的“do”。?等价于{0,1}。 |
| {*n*} | *n*是一个非负整数。匹配确定的*n*次。例如，“o{2}”不能匹配“Bob”中的“o”，但是能匹配“food”中的两个o。 |
| {*n*,} | *n*是一个非负整数。至少匹配*n*次。例如，“o{2,}”不能匹配“Bob”中的“o”，但能匹配“foooood”中的所有o。“o{1,}”等价于“o+”。“o{0,}”则等价于“o*”。 |
| {*n*,*m*} | *m*和*n*均为非负整数，其中*n*<=*m*。最少匹配*n*次且最多匹配*m*次。例如，“o{1,3}”将匹配“fooooood”中的前三个o为一组，后三个o为一组。“o{0,1}”等价于“o?”。请注意在逗号和两个数之间不能有空格。 |
| ? | 当该字符紧跟在任何一个其他限制符（*,+,?，{n}，{n,}，{n,m}）后面时，匹配模式是非贪婪的。非贪婪模式尽可能少的匹配所搜索的字符串，而默认的贪婪模式则尽可能多的匹配所搜索的字符串。例如，对于字符串“oooo”，“o+”将尽可能多的匹配“o”，得到结果[“oooo”]，而“o+?”将尽可能少的匹配“o”，得到结果 ['o', 'o', 'o', 'o'] |
| .点 | 匹配除“\n”之外的任何单个字符。要匹配包括“\n”在内的任何字符，请使用像“[\s\S]”的模式。 |
| (pattern) | 匹配pattern并获取这一匹配。所获取的匹配可以从产生的Matches集合得到，在VBScript中使用SubMatches集合，在JScript中则使用$0 . . .9$属性。要匹配圆括号字符，请使用“(”或“)”。 |
| (?:pattern) | 非获取匹配，匹配pattern但不获取匹配结果，不进行存储供以后使用。这在使用或字符“(\|)”来组合一个模式的各个部分时很有用。例如“industr(?:y\|ies)”就是一个比“industry\|industries”更简略的表达式。 |
| (?=pattern) | 非获取匹配，正向肯定预查，在任何匹配pattern的字符串开始处匹配查找字符串，该匹配不需要获取供以后使用。例如，“Windows(?=95\|98\|NT\|2000)”能匹配“Windows2000”中的“Windows”，但不能匹配“Windows3.1”中的“Windows”。预查不消耗字符，也就是说，在一个匹配发生后，在最后一次匹配之后立即开始下一次匹配的搜索，而不是从包含预查的字符之后开始。 |
| (?!pattern) | 非获取匹配，正向否定预查，在任何不匹配pattern的字符串开始处匹配查找字符串，该匹配不需要获取供以后使用。例如“Windows(?!95\|98\|NT\|2000)”能匹配“Windows3.1”中的“Windows”，但不能匹配“Windows2000”中的“Windows”。 |

| 元字符 | 描述 |
| --- | --- |
| (?<=pattern) | 非获取匹配，反向肯定预查，与正向肯定预查类似，只是方向相反。例如，“(?<=95\|98\|NT\|2000)Windows”能匹配“2000Windows”中的“Windows”，但不能匹配“3.1Windows”中的“Windows”。 |
| (?<!pattern) | 非获取匹配，反向否定预查，与正向否定预查类似，只是方向相反。例如“(?<!95\|98\|NT\|2000)Windows”能匹配“3.1Windows”中的“Windows”，但不能匹配“2000Windows”中的“Windows”。这个地方不正确，有问题此处用或任意一项都不能超过2位，如“(?<!95\|98\|NT\|20)Windows正确，“(?<!95\|980\|NT\|20)Windows 报错，若是单独使用则无限制，如(?<!2000)Windows 正确匹配 |
| x\|y | 匹配x或y。例如，“z\|food”能匹配“z”或“food”(此处请谨慎)。“[zf]ood”则匹配“zood”或“food”。 |
| [xyz] | 字符集合。匹配所包含的任意一个字符。例如，“[abc]”可以匹配“plain”中的“a”。 |
| [^xyz] | 负值字符集合。匹配未包含的任意字符。例如，“[^abc]”可以匹配“plain”中的“plin”。 |
| [a-z] | 字符范围。匹配指定范围内的任意字符。例如，“[a-z]”可以匹配“a”到“z”范围内的任意小写字母字符。注意:只有连字符在字符组内部时,并且出现在两个字符之间时,才能表示字符的范围; 如果出字符组的开头,则只能表示连字符本身. |
| [^a-z] | 负值字符范围。匹配任何不在指定范围内的任意字符。例如，“[^a-z]”可以匹配任何不在“a”到“z”范围内的任意字符。 |
| \b | 匹配一个单词边界，也就是指单词和空格间的位置（即正则表达式的“匹配”有两种概念，一种是匹配字符，一种是匹配位置，这里的\b就是匹配位置的）。例如，“er\b”可以匹配“never”中的“er”，但不能匹配“verb”中的“er”。 |
| \B | 匹配非单词边界。“er\B”能匹配“verb”中的“er”，但不能匹配“never”中的“er”。 |
| \cx | 匹配由x指明的控制字符。例如，\cM匹配一个Control-M或回车符。x的值必须为A-Z或a-z之一。否则，将c视为一个原义的“c”字符。 |
| \d | 匹配一个数字字符。等价于[0-9]。grep 要加上-P，perl正则支持 |
| \D | 匹配一个非数字字符。等价于[^0-9]。grep要加上-P，perl正则支持 |
| \f | 匹配一个换页符。等价于\x0c和\cL。 |
| \n | 匹配一个换行符。等价于\x0a和\cJ。 |
| \r | 匹配一个回车符。等价于\x0d和\cM。 |
| \s | 匹配任何不可见字符，包括空格、制表符、换页符等等。等价于[ \f\n\r\t\v]。 |
| \S | 匹配任何可见字符。等价于[^ \f\n\r\t\v]。 |
| \t | 匹配一个制表符。等价于\x09和\cI。 |
| \v | 匹配一个垂直制表符。等价于\x0b和\cK。 |

| 元字符 | 描述 |
| --- | --- |
| \w | 匹配包括下划线的任何单词字符。类似但不等价于“[A-Za-z0-9_]”，这里的"单词"字符使用Unicode字符集。 |
| \W | 匹配任何非单词字符。等价于“[^A-Za-z0-9_]”。 |
| \x*n* | 匹配*n*，其中*n*为十六进制转义值。十六进制转义值必须为确定的两个数字长。例如，“\x41”匹配“A”。“\x041”则等价于“\x04&1”。正则表达式中可以使用ASCII编码。 |
| *num* | 匹配*num*，其中*num*是一个正整数。对所获取的匹配的引用。例如，“(.)\1”匹配两个连续的相同字符。 |
| *n* | 标识一个八进制转义值或一个向后引用。如果*n*之前至少*n*个获取的子表达式，则*n*为向后引用。否则，如果*n*为八进制数字（0-7），则*n*为一个八进制转义值。 |
| *nm* | 标识一个八进制转义值或一个向后引用。如果*nm之前至少有nm个获得子表达式，则nm为向后引用。如果*nm之前至少有*n*个获取，则*n*为一个后跟文字*m*的向后引用。如果前面的条件都不满足，若*n*和*m*均为八进制数字（0-7），则*nm将匹配八进制转义值nm*。 |
| *nml* | 如果*n*为八进制数字（0-7），且*m*和*l均为八进制数字（0-7），则匹配八进制转义值nml*。 |
| \u*n* | 匹配*n*，其中*n*是一个用四个十六进制数字表示的Unicode字符。例如，\u00A9匹配版权符号（©）。 |
| \p{P} | 小写 p 是 property 的意思，表示 Unicode 属性，用于 Unicode 正表达式的前缀。中括号内的“P”表示Unicode 字符集七个字符属性之一：标点字符。其他六个属性：L：字母；M：标记符号（一般不会单独出现）；Z：分隔符（比如空格、换行等）；S：符号（比如数学符号、货币符号等）；N：数字（比如阿拉伯数字、罗马数字等）；C：其他字符。*注：此语法部分语言不支持，例：javascript。 |
| <> | 匹配词（word）的开始（<）和结束（>）。例如正则表达式<the>能够匹配字符串"for the wise"中的"the"，但是不能匹配字符串"otherwise"中的"the"。注意：这个元字符不是所有的软件都支持的。 |
| ( ) | 将( 和 ) 之间的表达式定义为“组”（group），并且将匹配这个表达式的字符保存到一个临时区域（一个正则表达式中最多可以保存9个），它们可以用 \1 到\9 的符号来引用。 |
| \| | 将两个匹配条件进行逻辑“或”（Or）运算。例如正则表达式(him\|her) 匹配"it belongs to him"和"it belongs to her"，但是不能匹配"it belongs to them."。注意：这个元字符不是所有的软件都支持的。 |

## 用法

```
//第一种
var password = "abc123456";
var reg = new RegExp("/^\\w{6}$/");
if(reg.test(password)){
    alert("验证通过！");
}

//第二种
```

```
/*是否带有小数*/
function isDecimal(strValue )  {
   var objRegExp= /^\d+\.\d+$/;
   return  objRegExp.test(strValue);
}

/*校验是否中文名称组成 */
function ischina(str) {
    var reg=/^[\u4E00-\u9FA5]{2,4}$/;   /*定义验证表达式*/
    return reg.test(str);     /*进行验证*/
}

/*校验是否全由8位数字组成 */
function isStudentNo(str) {
    var reg=/^[0-9]{8}$/;   /*定义验证表达式*/
    return reg.test(str);     /*进行验证*/
}

/*校验电话码格式 */
function isTelCode(str) {
    var reg= /^((0\d{2,3}-\d{7,8})|(1[3584]\d{9}))$/;
    return reg.test(str);
}

/*校验邮件地址是否合法 */
function IsEmail(str) {
    var reg=/^([a-zA-Z0-9_-])+@([a-zA-Z0-9_-])+(\.[a-zA-Z0-9_-])+$/;
    return reg.test(str);
}
```

## 计时

在 JavaScritp 中使用计时事件是很容易的，两个关键方法是:

- setInterval() - 间隔指定的毫秒数不停地执行指定的代码。
- setTimeout() - 在指定的毫秒数后执行指定代码。

**注意:** setInterval() 和 setTimeout() 是 HTML DOM Window对象的两个方法。

```
//setInterval()
var interval = setInterval(function(){alert("Hello")},3000);
//停止
clearInterval(interval);

//setTimeout()
var timeout = setTimeout(function(){alert("Hello")},3000);
//停止
clearTimeout(timeout);
```

### 常规window用法

1. window.location.href = "www.baidu.com";跳转到某个地址
2. window.history.back() - 与在浏览器点击后退按钮相同
3. window.history.forward() - 与在浏览器中点击向前按钮相同

## 事件

1. onclick
2. onload
3. onsubmit
4. onfocus
5. onblur
6. onmouseup
7. onmousedown

```
//第一种
window.onload = function(){
    var p = document.getElementById("id");
    p.onclick = function(){
        alert(1);
    }
}

//第二种
function click(){
   alert(1);
}
<p onclick="click()"></p>
```

# Jquery

jQuery 是一个 JavaScript 库。

jQuery 极大地简化了 JavaScript 编程。

## Jquery对象与dom对象互转

### jquery对象转dom对象

1. 通过 [index] 的方法得到对应的 DOM对象

```
//获取jquery对象
var $cr = $("#cr");
//jquery对象转dom对象
var cr = $cr[0];
```

2. 使用 jQuery 中的 get(index) 方法得到相应的 DOM 对象

```
//获取jquery对象
var $cr = $("#cr");
//jquery对象转dom对象
var cr = $cr.get(0);
```

### dom对象转jquery对象

对于一个 DOM 对象, 只需要用 $() 把 DOM 对象包装起来(jQuery 对象就是通过 jQuery 包装 DOM 对象后产生的对象), 就可以获得一个 jQuery 对象。

```
//获取dom对象
var cr = document.getElementById("cr");
//dom对象转jquery对象
var $cr = $(cr);
```

## 选择器

1. 元素选择器

   $("p")

2. id选择器

   $("#id")

3. class选择器

   $(".class")

4. 复杂选择器

| **$("p.intro")** | **选取 class 为 intro 的**<br>**元素** |
| --- | --- |
| $("p:first") | 选取第一个<br>元素 |
| $("ul li:first") | 选取第一个<br>元素的第一个<br>○ 元素 |
| $("ul li:first-child") | 选取每个<br>元素的第一个<br>○ 元素 |
| $("[href]") | 选取带有 href 属性的元素 |

## 元素操作

1.text()：获取文本

2.html()：获取html代码

3.val()：表单的值，获取input、select的值

4.append()：追加html代码

5.remove()： 删除被选元素（及其子元素）

6.empty()：从被选元素中删除子元素

## 遍历操作

1.parent()：返回被选元素的直接父元素

2.children()：返回被选元素的所有直接子元素

3.find()：返回被选元素的后代元素，一路向下直到最后一个后代

4.siblings()：返回被选元素的所有同胞元素

5.first()：返回被选元素的首个元素

## 效果操作

1.hide() 和 show() 方法来隐藏和显示 HTML 元素

2.jQuery animate() 方法用于创建自定义动画。

**语法：**

$(*selector*).animate({*params*},*speed*,*callback*);

必需的 params 参数定义形成动画的 CSS 属性。

可选的 speed 参数规定效果的时长。它可以取以下值：“slow”、“fast” 或毫秒。

可选的 callback 参数是动画完成后所执行的函数名称。

## 跨域访问

Jsonp(JSON with Padding) 是 json 的一种"使用模式"，可以让网页从别的域名（网站）那获取资料，即跨域读取数据。

1. 服务器端

```
private void GetTop10() throws IOException {
  String jsonCallback = _request.getParameter("jsonCallback");
  String id = _request.getParameter("id");
  String return_JSONP = top10Setup.ReadTop10(id );
  return_JSONP = jsonCallback + "(" + return_JSONP  + ");";
  System.out.println( return_JSONP );
  _response.getWriter().println( return_JSONP  );
}
```

2. 客户端

```
function appendListOne() {
  $.jsonp({
    'contentType': "application/json; charset=utf-8",
    'url': AjaxGetUrl,
    'data': {
      dz_type: 'GetTop10',
      id: 17
    },
    'dataType':"jsonp",
    'callbackParameter':"jsonCallback",
    'async':false,
    'type': 'post',
    'success': function (data) {
      console.warn( data );
      $("#list1>tbody").empty();
      var str="";
      for(var key=0;key<data.dblist.length;key++){
        var dd=data.dblist[key];
        if(key <5){
          str+='<tr><td><b class="ranking">' +(parseInt(key)+1)+
            '</b><span class="name">'+dd.data_name+'</span></td>' +
            '<td> <span
class="detail">'+dd.data_value+'.'+dd.data_value2+'.'+dd.data_value3+'.'+dd.data_value4
            +'</span> </td> </tr>';
        }
      }
      $("#list1>tbody").append(str);
    }
  });
}
```

# Ajax

AJAX = 异步 JavaScript 和 XML（Asynchronous JavaScript and XML）。简短地说，在不重载整个网页的情况下，AJAX 通过后台加载数据，并在网页上进行显示。

## 核心

Ajax核心是通过组件**XMLHttpRequest**对象进行异步数据读取。

## 优点

1. 不用跳转实现数据的更新，用户体验好
2. 由于不用全部更新页面数据，所以减少服务器资源消耗，减少流量

## 缺点

1. 页面局部刷新，导致后退等功能失效
2. 对搜索引擎不友好，不利于SEO

## 实例一

1. 获取XMLHttpRequest对象
2. 建立连接
3. 发送请求
4. 监听请求状态
5. 获取并处理响应内容

```
//获取XMLHttpRequest对象
function getXMLHttpRequest(){
    var xhr = null;
    //根据浏览器不同采用不同的获取XMLHttpRequest对象方式
    if(window.XMLHttpRequest){
        //现代浏览器，比如Google、FireFox等
        xhr = new XMLHttpRequest();
    }else if(window.ActiveXObject){
        //IE浏览器
        xhr = new ActiveXObject("Microsoft.XMLHTTP");
    }
    return xhr;
}

//get方式
previous.onclick=function(){
    //清空错误信息
    msg.innerHTML = "";

    //获取XMLHttpRequest对象
    var request = getXMLHttpRequest();
    //建立连接
    request.open("GET", "${ctx}/changeImg?option=previous&index=" + index, true);
    //发送请求
    request.send(null);
    //监听请求状态
    request.onreadystatechange = function(){
        //请求状态==4代表请求完毕
        if(request.readyState == 4){
            //响应状态==200||304代表响应完成
            if(request.status == 200 || request.status == 304){
                //获取响应内容
                var returnIndex = request.responseText;
                //判断返回的数据是否是图片的位置
                if(isNaN(parseInt(returnIndex))){
                    //错误信息
                    msg.innerHTML = returnIndex;
                    //显示错误
                    msg.style.display = "block";
                    return;
                }
```

```
                //改变图片地址
                img.src="img/timg (" + returnIndex + ").jpg";
                //改变当前位置
                index = returnIndex;
            }
        }
    }
}

//post方式
//下一张
next.onclick=function(){
    //清空错误信息
    msg.innerHTML = "";

    //获取XMLHttpRequest对象
    var request = getXMLHttpRequest();
    //建立连接
    request.open("POST", "${ctx}/changeImg", true);
    //设置内容编码类型
    request.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
    //发送请求
    request.send("option=next&index=" + index);
    //监听请求状态
    request.onreadystatechange = function(){
        //请求状态==4代表请求完毕
        if(request.readyState == 4){
            //响应状态==200||304代表响应完成
            if(request.status == 200 || request.status == 304){
                //获取响应内容
                var returnIndex = request.responseText;
                //判断返回的数据是否是图片的位置
                if(isNaN(parseInt(returnIndex))){
                    //错误信息
                    msg.innerHTML = returnIndex;
                    //显示错误
                    msg.style.display = "block";
                    return;
                }
                //改变图
                //改变图片地址
                img.src="img/timg (" + returnIndex + ").jpg";
                //改变当前位置
                index = returnIndex;
            }
        }
    }
}
```

## 实例二

1. $.ajax
2. $.get
3. $.post
4. $.getJson

```
//$.ajax
$("#ajax").click(function(){
    $.ajax({
        url: "${ctx}/getActor",
        type: "get",//请求类型
        data: {"id": $("#id").val()},//发送给服务器的数据
        dataType: "json",//服务器响应类型
        success: function(data){//请求成功后的回调函数
            //判断是否错误
            if(data.code){
                $("#content").text(data.msg);
            }else{
                $("#content").text(data.actorId + " " + data.firstName + " " + data.lastName  +
" " + data.lastUpdate);
            }
            $("#content").show();
        },
        error: function(msg){//请求失败后的回调函数
            console.info(msg);
        }
    });
});

//$.get
$("#get").click(function(){
    $.get("${ctx}/getActor", {"id": $("#id").val()}, function(data){
        //判断是否错误
        if(data.code){
            $("#content").text(data.msg);
        }else{
            $("#content").text(data.actorId + " " + data.firstName + " " + data.lastName  + " "
+ data.lastUpdate);
        }
        $("#content").show();
    }, "json");
});

//$.post
$("#post").click(function(){
    $.post("${ctx}/getActor", {"id": $("#id").val()}, function(data){
        //判断是否错误
        if(data.code){
            $("#content").text(data.msg);
        }else{
            $("#content").text(data.actorId + " " + data.firstName + " " + data.lastName  + " "
+ data.lastUpdate);
        }
```

```
        $("#content").show();
    }, "json");
});

//$.getJson
$("#json").click(function(){
    $.getJSON("${ctx}/getActor", {"id": $("#id").val()}, function(data){
        //判断是否错误
        if(data.code){
            $("#content").text(data.msg);
        }else{
            $("#content").text(data.actorId + " " + data.firstName + " " + data.lastName  + " "
+ data.lastUpdate);
        }
        $("#content").show();
    });
});
```

# JDBC

Java程序访问数据库的一种技术。

## 声明

1. Statement：普通声明，通过拼接sql语句方式，存在sql注入风险
2. PreparedStatement：预编译声明，通过预编译sql的方式，速度快，安全性高
3. CallableStatement：回调声明，用来执行存储过程

## 步骤

1. 加载驱动
2. 获取连接
3. 创建声明
4. 执行sql
5. 处理数据
6. 关闭连接

## 实例

```
//statement
public static int insert(Actor actor) {
    int result = 0;
    Connection connection = null;
    Statement statement = null;
    try {
        // 获取连接
        connection = JDBCUtil.getConnection();
        // 获取声明
        statement = JDBCUtil.getStatement(connection);
```

```
        // 执行sql语句
        String sql = "insert into actor(first_name, last_name, last_update) values ('" +
actor.getFirstName()
            + "','" + actor.getLastName() + "','" + actor.getLastUpdate().toString() + "') ";
        // 执行
        result = statement.executeUpdate(sql);
    } catch (SQLException e) {
        e.printStackTrace();
    } finally {
        // 关闭
        JDBCUtil.close(connection, statement, null);
    }

    return result;
}

//preparedStatment
public static int insert(Actor actor) {
    int result = 0;
    Connection connection = null;
    PreparedStatement statement = null;
    try {
        // 获取连接
        connection = JDBCUtil2.getConnection();
        // 执行sql语句
        String sql = "insert into actor(first_name, last_name, last_update) values (?,?,?)";
        // 获取声明
        statement = JDBCUtil2.getPreparedStatement(connection, sql);
        // 设置参数
        statement.setString(1, actor.getFirstName());
        statement.setString(2, actor.getLastName());
        statement.setTimestamp(3, actor.getLastUpdate());
        // 执行，注意这样不要传入sql语句
        result = statement.executeUpdate();
    } catch (SQLException e) {
        e.printStackTrace();
    } finally {
        // 关闭
        JDBCUtil2.close(connection, statement, null);
    }

    return result;
}

//callableStatement
public static void main(String[] args) {
    Connection connection = null;
    CallableStatement callableStatement = null;
    try {
        // 加载驱动
        Class.forName(PropertiesUtil.getPropertis("jdbc.driver.name"));
        // 获取连接
        connection = DriverManager.getConnection(PropertiesUtil.getPropertis("jdbc.url"),
```

```
                                                PropertiesUtil.getPropertis("jdbc.user"),
PropertiesUtil.getPropertis("jdbc.password"));
        // 创建sql语句
        String sql = "call proc_insert_actor(?,?,?,?)";
        // 获取声明
        callableStatement = connection.prepareCall(sql);
        // 设置输入参数
        callableStatement.setString(1, "小茗");
        callableStatement.setString(2, "李");
        callableStatement.setTimestamp(3, new Timestamp(new Date().getTime()));
        // 注册输出参数
        callableStatement.registerOutParameter(4, Types.VARCHAR);
        //执行
        callableStatement.execute();
        // 获取存储过程输出结果
        String result = callableStatement.getString(4);
        System.out.println("存储过程输出结果: " + result);
    } catch (ClassNotFoundException e) {
        e.printStackTrace();
    } catch (SQLException e) {
        e.printStackTrace();
    } finally {
        if (callableStatement != null) {
            try {
                callableStatement.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
        if (connection != null) {
            try {
                connection.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }
}
```

# JSP

JSP 技术是以 Java 语言作为脚本语言的，JSP 网页为整个服务器端的 Java 库单元提供了一个接口来服务于HTTP的应用程序。

## 运行原理

请求-》jsp引擎-》servlet引擎（第一次访问才需要）-》处理-》响应

## 语法

1. 表达式

```
<%=out.print("Hello")%>
```

2. 片段

```
<%

    String name = request.getParameter("name");

    out.print(name);

%>

<%
  for(int i = 0; i < 10; i++){
%>
    <p>正在打印<%=i%></p>
<%
  }
%>
```

3. 声明

```
//用于定义JSP页面转换成的Servlet程序的静态代码块、成员变量和方法， 不能写逻辑
<%!
    //静态变量，静态的变量、静态代码块、静态方法必须写在jsp声明中
    static int num = 5;

    //非静态变量
    int num2 = 5;

    //静态代码库
    static {
        System.out.println("执行静态代码库");
    }

    //静态方法
    public static int sum(int x, int y){
        return x + y;
    }

    //非静态方法
    public int sum2(int x, int y){
        return x + y;
    }
%>
```

4. 指令

   <%@ 指令 属性名="值" %>

   比如：

   **page指令**

```
JSP 2.0规范中定义的page指令的完整语法：
<%@ page
    [ language="java" ]
    [ extends="package.class" ]
    [ import="{package.class | package.*}, ..." ]
    [ session="true | false" ]
    [ buffer="none | 8kb | sizekb" ]
    [ autoFlush="true | false" ]
    [ isThreadSafe="true | false" ]
    [ info="text" ]
    [ errorPage="relative_url" ]
    [ isErrorPage="true | false" ]
    [ contentType="mimeType [ ;charset=characterSet ]" | "text/html ; charset=ISO-8859-1" ]
    [ pageEncoding="characterSet | ISO-8859-1" ]
    [ isELIgnored="true | false" ]
%>
```

   **include指令**

   include指令用于通知JSP引擎在翻译当前JSP页面时将其他文件中的内容合并进当前JSP页面转换成的Servlet源文件中，这种在源文件级别进行引入的方式称之为静态引入，当前JSP页面与静态引入的页面紧密结合为一个Servlet。

   <%@ include file="relativeURL"%>

```
//静态引入可以访问被引入的jsp页面定义的变量、方法等，动态引入不可以
<%@ include file=“b.jspf”%>
```

## 内置对象

1. request
2. response
3. session
4. page
5. pageContext
6. application
7. config
8. out
9. exception

## 标签

1. <jsp:include>标签

<jsp:include>标签用于把另外一个资源的输出内容插入进当前JSP页面的输出内容之中，这种在JSP页面执行时的引入方式称之为动态引入。

```
<jsp:include page="include/header.jsp" />
```

2. <jsp:forward>标签：用于把请求转发给另外一个资源
3. <jsp:param>标签：传递参数

## <jsp:include>标签和include指令的区别

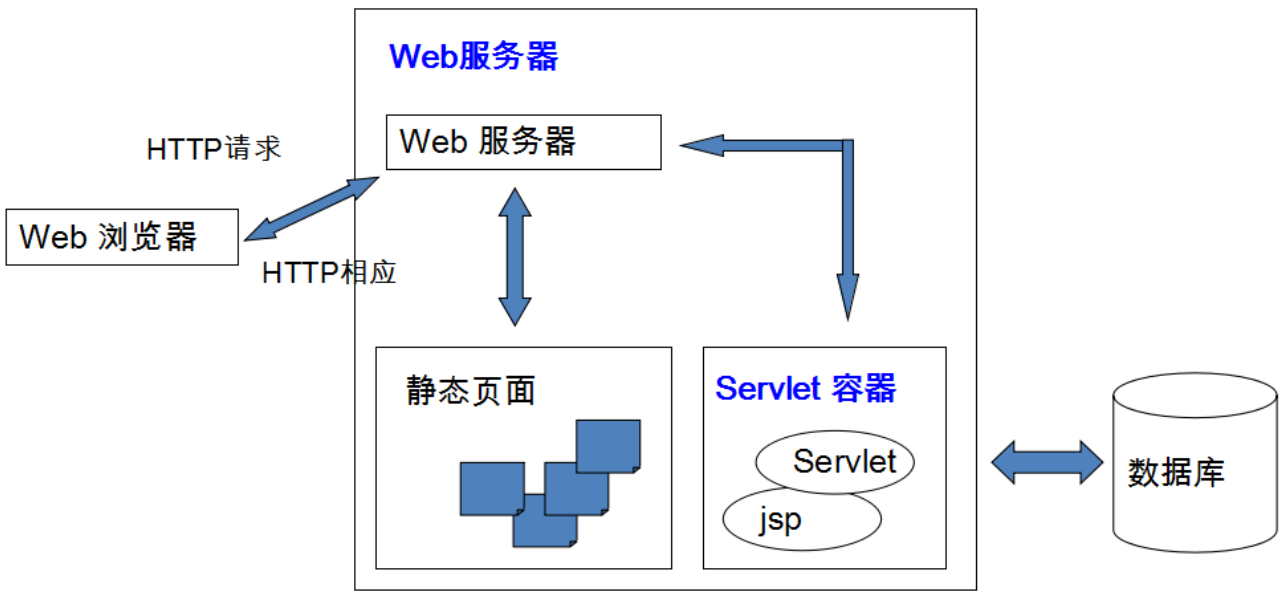
1. <jsp:include>标签将当前jsp和被引入的jsp先单独编译成两个servlet在执行后再合并，include指令当前jsp和被引入的jsp先在编译成servlet前合并再编译成一个servlet执行
2. <jsp:include>标签在被引入的jsp页面声明的变量、方法等，不能在引入的jsp在引入位置之后的脚本片段中使用，相反include指令静态引入的jsp可以使用

## 转发和重定向区别

1. 重定向浏览器地址栏的url会改变，转发则不会还是初始的url
2. 重定向是在浏览器端发起二次请求进行跳转，转发在服务器内部进行转发
3. 重定向后request不是原来的request所以保存在里面的数据丢失，但是转发因为是同一个请求在服务器进行转发所以request的数据依然存在

# Servlet

Java Servlet 是运行在 Web 服务器或应用服务器上的程序，它是作为来自 Web 浏览器或其他 HTTP 客户端的请求和 HTTP 服务器上的数据库或应用程序之间的中间层。



## Servlet容器

Servlet容器为JavaWeb应用提供运行时环境，它负责管理Servlet和JSP的生命周期，以及管理它们的共享数据。

目前最流行的Servlet容器软件括:

**Tomcat**

JBoss

Jetty

WebsphereResin

J2EE服务器（如Weblogic）中也提供了内置的Servlet容器

## Servlet处理请求的过程

1. Servlet引擎检查是否已经装载并创建了该Servlet的实例对象。如果是，则直接执行第④步，否则，执行第②步。
2. 装载并创建该Servlet的一个实例对象：调用该 Servlet 的构造器 调用Servlet实例对象的init()方法。
3. 创建一个用于封装请求的ServletRequest对象和一个代表响应消息的ServletResponse对象，然后调用Servlet的service()方法并将请求和响应对象作为参数传递进去。
4. WEB应用程序被停止或重新启动之前，Servlet引擎将卸载Servlet，并在卸载之前调用Servlet的destroy()方法。

## Servlet生命周期

Servlet 生命周期可被定义为从创建直到毁灭的整个过程。以下是 Servlet 遵循的过程：

- Servlet 通过调用 **init ()** 方法进行初始化。
- Servlet 调用 **service()** 方法来处理客户端的请求。
- Servlet 通过调用 **destroy()** 方法终止（结束）。
- 最后，Servlet 是由 JVM 的垃圾回收器进行垃圾回收的。

Servlet是单例、多线程的，避免使用成员变量

## Servlet和Jsp区别和联系

区别：

```
1.jsp经编译后就变成了Servlet.(JSP的本质就是Servlet，JVM只能识别java的类，不能识别JSP的代码,Web容器将JSP的代码编译成JVM能够识别的java类)
2.jsp更擅长表现于页面显示,servlet更擅长于逻辑控制.
3.Servlet中没有内置对象，Jsp中的内置对象都是必须通过HttpServletRequest对象，HttpServletResponse对象以及HttpServlet对象得到.
```

联系：

```
JSP是Servlet技术的扩展，本质上就是Servlet的简易方式。JSP编译后是“类servlet”。Servlet和JSP最主要的不同点在于，Servlet的应用逻辑是在Java文件中，并且完全从表示层中的HTML里分离开来。而JSP的情况是Java和HTML可以组合成一个扩展名为.jsp的文件。JSP侧重于视图，Servlet主要用于控制逻辑。
```

# 会话与状态管理

HTTP协议是一种无状态的协议，WEB服务器本身不能识别出哪些请求是同一个浏览器发出的，浏览器的每一次请求都是完全孤立的。作为 web 服务器，必须能够采用一种机制来唯一地标识一个用户，同时记录该用户的状态。

## 会话跟踪两种方式

1. Cookie
2. Session

## session和cookie区别

1. session是保存的服务器端，cookie保存在客户端
2. session通过cookie或者地址重写的方式保存sessionId，每次请求带上sessionId，服务器根据sessionId来找到相应的session

# EL&JSTL

## EL

EL 全名为 Expression Language，方便取数据所自定义的语言。

**语法**

EL 都是以 ${ 为起始、以} 为结尾的 。

| 范　例 | 说　明 |
| --- | --- |
| ${pageScope.username} | 取出 Page 范围的 username 变量 |
| ${requestScope.username} | 取出 Request 范围的 username 变量 |
| ${sessionScope.username} | 取出 Session 范围的 username 变量 |
| ${applicationScope.username} | 取出 Application 范围的 username 变量 |

## JSTL

JSP标准标签库（JSTL）是一个JSP标签集合，它封装了JSP应用的通用核心功能。

1. **核心标签**

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<c:set var="ctx" value="${pageContext.request.ContextPath}"></c:set>
<c:if test="${num > 3}"></c:if>
<c:choose>
    <c:when test="${num > 3}"></c:when>
    <c:otherwise></c:otherwise>
</c:choose>
<c:forEach var="item" items="${items}" begin="0" end="3"></c:forEach>
```

2. **格式化标签**

```
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>

<fmt:formatDate type="date" value="${createDate}"></fmt:formatDate>
<fmt:formatNumber type="PERCENT" value="${rate}"></fmt:formatNumber>
```

3. **JSTL函数**

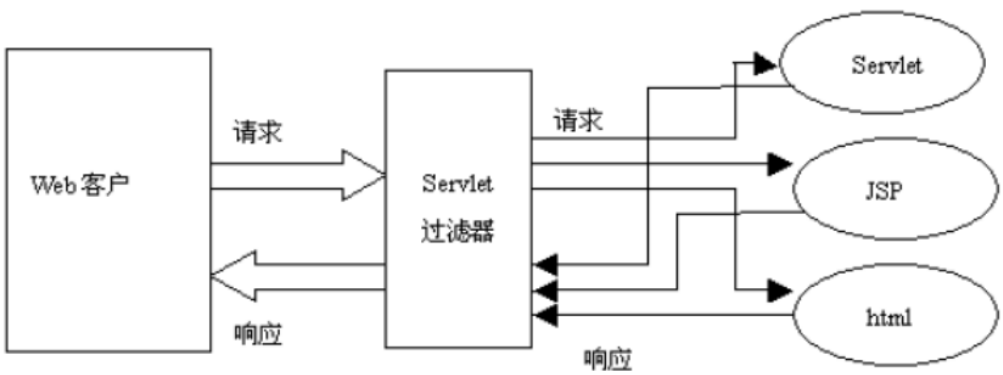
```
<%@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions" %>

<input value="${fn:length(title) > 10? fn:substring(content, 0, 5) : title}">
```

# 过滤器和监听器

## 过滤器

Filter （接口）的基本功能是对 Servlet 容器调用 Servlet 的过程进行拦截，从而在 Servlet 进行响应处理的前后实现一些特殊的功能。

**执行过程**



# 监听器

Servlet 监听器：Servlet 规范中定义的一种特殊类，它用于监听 web 应用程序中的 **ServletContext**, **HttpSession** 和 **ServletRequest** 等域对象的创建与销毁事件，以及监听这些域对象中的属性发生修改的事件。

**分类**

按监听的事件类型 Servlet 监听器可分为如下三种类型：

1. 监听域对象自身的创建和销毁的事件监听器
2. 监听域对象中的属性的增加和删除的事件监听器
3. 监听绑定到 HttpSession 域中的某个对象的状态的事件监听器