day11 事件上

事件的概述

事件是指代一个东西的操作被另外一个东西进行监听以后的一个过程（事件），这个过程可以完成对应的操作（处理函数）。事件监听器是一个标准的观察者模式（observer） 也被称为发布、订阅者模式。

示例

我跟一个人告白了，这个人说要我等等再给我回应，那么现在一旦有回应了以后就会通知我，然后我再进行相关操作。

- **事件名**
- 告白 、 回应
- **执行对象**
- 我、这个人
- **处理对象**
- 这个人、我
- **处理函数**
- 等等、送她一个男朋友

示例（标准示例）

点击按钮触发一个操作改变按钮的颜色

- **事件名 点击**
- **执行对象 按钮**
- **处理对象 js引擎**
- **处理函数 改变按钮颜色**

事件的模式

内联模式 （嵌入到标签中的）

脚本模式 （分离于标签的）

```
<!-- 内联模式相当于直接帮你调用代码 以浏览器本身来直接执行的 window来帮你执行的 -->
<!--在里面直接嵌入代码-->
<button onclick="alert('你好')">点击弹窗</button>
<!--在里面调用函数 以执行代码的形式执行的-->
<button onclick="handlerClick()">点击弹窗</button>
<button class="btn">脚本模式匿名函数</button>
<button class="btn">脚本模式具名函数</button>
<script>
    function handlerClick() {
      alert('hello')
      console.log(this)//指向window
```

```
    }
    //脚本模式相当于把对应的onclick当成一个属性 也就是把元素element当成一个对象
    var btn = document.querySelector('.btn') //类型为对象
    console.log(typeof btn) //object 对象可以定义属性
    // btn['onclick'] = function(){
    //   console.log('hello world')
    //   console.log(this)
    // }
    btn.onclick = function(){
      console.log('hello world')
      console.log(this)
    }
    //具名函数
    document.querySelectorAll('.btn')[1].onclick = sayHello
    function sayHello(){
      alert('hello')
    }
</script>
```

内联模式和脚本模式的区别

- 内联模式相等于属性赋值 里面的代码是window对象帮你执行，而对应的脚本模式里面的this执行当前的调用元素
- 脚本模式看不见调用函数 内联模式是直接可以看到执行的函数名

事件名的分类

事件的构成

- 触发的对象.on+事件名 = 处理函数

鼠标事件 （mouse）

- 点击事件 click
- 双击事件 dblclick
- 鼠标按下 mousedown
- 鼠标弹起 mouseup
- 鼠标移入 mouseenter
- 鼠标移出 mouseleave
- 鼠标移入 mouseover
- 鼠标移出 mouseout
- 鼠标右键 contextmuen

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
  <style>
    div {
      width: 100px;
      height: 100px;
      background-color: red;
```

```
    }
  </style>
</head>

<body>
  <div>
    <button>按钮</button>
  </div>
  <script>
    var div = document.querySelector('div')
    //点击事件
    div.onclick = function () {
      console.log('点击事件')
    }
    //双击会触发俩次点击
    div.ondblclick = function () {
      console.log('双击')
    }
    //按下
    div.onmousedown = function () {
      console.log('按下')
    }
    //弹起
    div.onmouseup = function () {
      console.log('弹起')
    }
    //先按下 -- 再弹起  -- 再点击
    // 并不会向下执行  冒泡 从下到上 (从里到外)  捕获 从上到下(从外到里)
    div.onmouseenter = function () {
      console.log('移入 enter')
    }
    div.onmouseleave = function () {
      console.log('移出 leave')
    }
    //向下执行 冒泡
    div.onmouseover = function () {
      console.log('移入 over')
    }
    div.onmouseout = function () {
      console.log('移出 out')
    }
    div.oncontextmenu = function(){
      console.log('鼠标右键')
    }
    //鼠标移动 (动画之类的)
    div.onmousemove = function(){
      console.log('鼠标移动')
    }
    //在移动端 没有点击 只有触发 touch
  </script>
</body>

</html>
```

注意事项

- 执行顺序 mousedown --- mouseup --- click

- mouseenter/mouseleave 及 mouseover/mouseout的区别，前置不会发送事件冒泡（也就是子元素不会触发）后置会发生事件冒泡（子元素会触发）

键盘事件 （key）

- 按下 keydown
- 弹起 keyup
- 按下非功能键 keypress （功能不会执行）

```
window.onkeydown = function(){
    console.log('按下')
}
window.onkeyup = function(){
    console.log('弹起')
}
window.onkeypress = function(){
    console.log('非功能键按下')
}
```

- 非功能键触发的过程 keydown -- keypress --- keyup

HTML事件

window窗口相关事件

- load 加载事件
- unload 卸载事件
- close 关闭
- beforeunload 卸载之前
- beforeprint 打印之前
- error 错误
- resize 重新设置大小
- reset 重新设置位置
- scroll 滚动栏滚动
- hashchange hash值变化
- popstate history的state发生变化
- ...

```
window.onload = function(){
    console.log('窗口加载的时候触发的')
}
window.onunload = function(){
    console.log('窗口卸载的时候触发的')
}
window.onclose = function(){
    console.log('窗口关闭的时候触发的')
}
//页面关闭之前
window.onbeforeunload = function(){
    console.log('在卸载之前触发的')
}
window.onbeforeprint = function(){
    console.log('打印之前触发的')
}
window.onerror = function(){
    console.log('窗口出错的时候触发的')
```

```
}
window.onresize = function(){
    console.log('窗口可操作空间大小发生变化的时候触发的')
}
window.onreset = function(){
    console.log('窗口重新设置位置的时候触发的')
}
window.onscroll = function(){
    console.log('窗口滚动栏发生变化的时候触发的')
}
window.onhashchange = function(){
    console.log('hash值发生变化的时候触发的')
}
window.onpopstate = function(){
    console.log('history中的state发生变化的时候触发的')
}
```

表单相关事件

- input 输入
- change value值发生变化
- focus 获取焦点
- blur 失去焦点
- select 内容被选择
- submit 提交
- reset 重置
- ...

```
//输入事件
inp.oninput = function () {
    console.log('表单输入')
}
//必须先失去焦点
inp.onchange = function () {
    console.log('值变化')
}
//value改变事件
select.onchange = function () {
    console.log('value发生变化')
}
//获取焦点事件
inp.onfocus = function () {
    console.log('获取焦点')
}
//失去焦点事件
inp.onblur = function () {
    console.log('失去焦点')
}
//选中内容触发的事件
inp.onselect = function () {
    console.log('内容被选中触发')
}
//form表单的事件 reset submit
document.forms[0].onsubmit = function(){
    console.log('提交')
}
```

```
document.forms[0].onreset = function(){
    console.log('重置')
}
```

....

event

概述

event是一个事件源对象，他包含了事件触发过程的内容，以及对应的元素的内容。他会默认传入给对应的事件的处理函数。

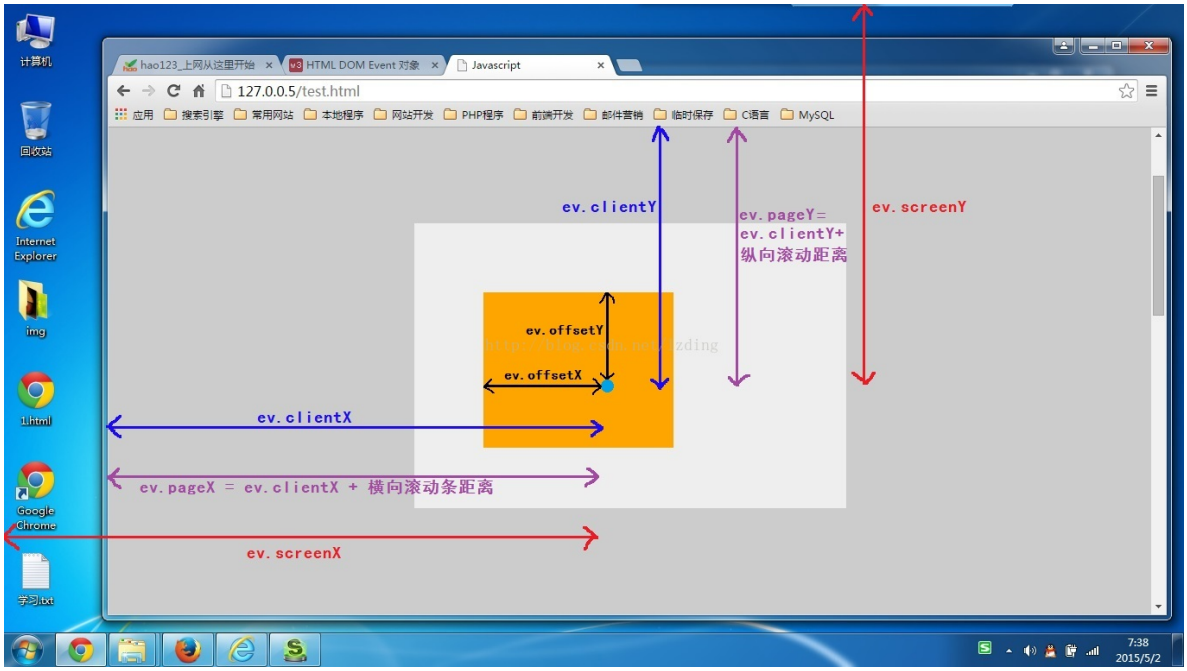
处理函数的arguments

```
var btn = document.querySelector('button')
btn.onclick = function (e) { //这个参数e就相当于接收的第一个实参
    console.log(arguments)
    console.log(arguments.length) //1 当前的这个处理函数内容只有一个参数 这个参数是一个event对象
    //pointerEvent对象 这个对象就是一个事件源对象
    console.log(arguments[0]) //传入的第一个实参
    // 这个地方的e就相当于是arguments[0]
    console.log(e)
    console.log(e == arguments[0])
    //这个接收的event对象是存在兼容问题 所以为了防止兼容问题的产生 我们一般会在里面先写对应的兼容
    e = e || window.event //兼容写法
}
//从上述可以得到对应的处理函数会被默认传递一个参数这个参数就是对应的事件源对象
//所以我们就可以直接用对应的形参接收实参的方法 通过形参e来接收这个对应的实参event
```

- 对应的处理函数的arguments会传递一个参数 而这个参数就是对应的事件源对象
- 这个事件源对象一般使用形参e来表示 一般的兼容写法为 e = e || window.event

event对象里面的相关属性

- type 触发事件的类型 *
- target 触发的目标元素 *
- currentTarget 当前加事件的元素 *
- button 鼠标点击的按钮
- screenX screenY 获取当前鼠标在屏幕上的位置
- pageX pageY 获取当前鼠标在页面上的位置（包含滚动栏位置）
- clientX clientY 获取当前鼠标在可视区的位置 （不包含滚动栏）
- offsetX offsetY 获取当前鼠标在目标元素上的位置
- altKey shiftKey ctrlKey 是否按下对应的功能键 *
- cancelBubble 取消冒泡 *
- returnValue 是否执行对应的默认行为 *
- bubbles 是否冒泡 *




```
//对应的属性
console.log(e.type) //获取对应的事件名
console.log(e.target) //获取对应的事件的目标元素 你当前执行事件的目标元素
console.log('按钮触发了')
console.log(e.currentTarget) //获取当前加事件的元素
//位置 鼠标点击的坐标
//获取鼠标点击位置的页面坐标 包含滚动栏
console.log(e.pageX)
console.log(e.pageY)
//获取鼠标在页面可视区的位置 不包含滚动栏的
console.log(e.clientX)
console.log(e.clientY)
//获取鼠标在对应的盒子里面的位置 也就是在按钮中的位置
console.log(e.offsetX)
console.log(e.offsetY)
//获取鼠标处在屏幕的位置 不包含滚动栏
console.log(e.screenX)
console.log(e.screenY) //加上导航栏的位置
//ctrlKey  shiftKey  altKey 是否按照ctrl键 是否按照shift键 是否按着alt键
console.log(e.ctrlKey,e.shiftKey,e.altKey)
//按那么键 0 1 2
console.log(e.button)
//取消冒泡
console.log(e.cancelBubble) //默认值为false
//返回的value值 返回true就可以走 返回false就不能走 走不走对应的默认行为
console.log(e.returnValue) //默认值为true
//是否冒泡
console.log(e.bubbles)
```

键盘相关的event属性

- key 按下的键 返回键的字符串
- keyCode 按下键的ascii码 （返回大写的ascii码 如果是keypress事件那么返回是对应的）
- charCode 字符键按下才有的 ascii 如果是给keypress事件那么对应的keyCode和charCode是一致的

```
//属于键盘事件源的属性
console.log(e.key) //按下的键 返回键的字符串
console.log(e.keyCode) //按下键的ascii码 返回大写的ascii码
console.log(e.charCode) //字符键按下才有的 ascii 如果是给keypress事件那么对应的keyCode和charCode是一致的
```

事件委托机制（事件代理）

概述

事件委托机制就是将自己的事件**委托给对应的父元素**去添加，在内部利用对应的**target来指向执行元素**的特性来进行相关的操作。

示例

给所有的li添加一个点击事件点击添加背景颜色

```
//原本的做法 获取所有的li 遍历添加点击事件
//利用事件委托 将对应的点击事件委托给对应的父元素 ul
var ul = document.querySelector('ul')
ul.onmouseover = function(e){
    e = e || window.event
    //利用e.target来获取对应的执行的元素
    if(e.target.tagName = 'LI'){
        //排他思想
        //将所有的li颜色变为白色
        for(var li of ul.children){
            li.style.backgroundColor = '#fff'
        }
        //给他添加背景颜色
        e.target.style.backgroundColor = '#f00'
    }
}
//应用场景 如果有多个共同的元素要添加一个事件 那么可以委托给他的父元素来进行代理
//注意事项 如果需要使用事件委托 那么对应的事件一定要支持事件冒泡(利用事件冒泡来完成操作的)
```

注意事项

- 如果需要使用事件委托 那么对应的事件一定要支持事件冒泡(利用事件冒泡来完成操作的)
- 如移入移出功能需要添加事件委托 那么必须使用mouseover/mouseout 不能使用mouseenter/mouseleave