

## 第六章 JavaScript高级语法

### 6.1、Exception

#### 6.1.1、异常概述

在ES3之前JavaScript代码执行的过程中，一旦出现错误，整个JavaScript代码都会停止执行，这样就显的代码非常的不健壮。

在Java或C#等一些高级语言中，都提供了异常处理机制，可以处理出现的异常，而不会停止整个应用程序。

从ES3开始，JavaScript也提供了类似的异常处理机制，从而让JavaScript代码变的更健壮，即使执行的过程中出现了异常，也可以让程序具有了一部分的异常恢复能力。

当错误发生时，JavaScript 提供了错误信息的内置 error 对象。

error 对象提供两个有用的属性：name 和 message

#### Error 对象属性

属性	描述
name	设置或返回错误名
message	设置或返回错误消息（一条字符串）

#### Error Name Values

error 的 name 属性可返回六个不同的值：

错误名	描述
EvalError	已在 eval() 函数中发生的错误
RangeError	已发生超出数字范围的错误
ReferenceError	已发生非法引用
SyntaxError	已发生语法错误
TypeError	已发生类型错误
URIError	在 encodeURIComponent() 中已发生的错误

### 6.1.2、异常捕捉

ES3开始引入了 try-catch 语句，是 JavaScript 中处理异常的标准方式。

语法格式：

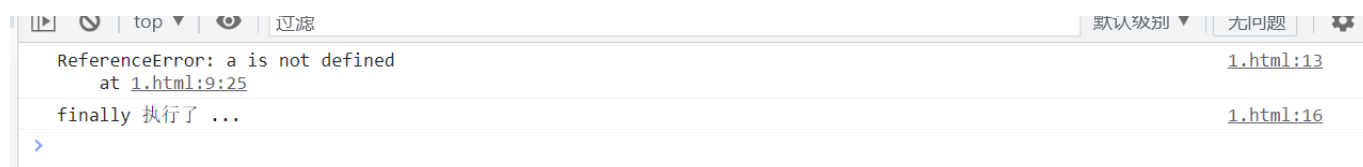
```
<script>
  try {
    // 可能发生异常的代码
  } catch (error) {
    // 发生错误执行的代码
  } finally {
    // 无论是否出错都会执行的代码
  }
</script>
```

在 try...catch 中，try 中一旦出现错误则其它语句不能执行，如果不出现错误则 catch 中的语句不会执行。

Javascript 参考其它编程语言，也提供了一种 finally 语句：不管 try 中的语句有没有错误，在最后都会执行 finally 中的语句。也就是说，try 中语句不发生错误执行完毕后会执行 finally 中的语句，try 中的语句发生错误，则执行 catch 中的语句，catch 中的语句执行完毕后会执行 finally 中的语句。

案例演示：

```
<script>
  try {
    console.log(a);
    console.log("a未定义肯定报错，你看不见我");
  } catch (error) {
    // 发生错误执行的代码
    console.log(error);
  } finally {
    // 无论是否出错都会执行的代码
    console.log("finally 执行了 ...")
  }
</script>
```



在JavaScript中，如果添加了 finally 语句，则 catch 语句可以省略。但是如果没有 catch 语句，则一旦发生错误就无法捕获这个错误，所以在执行完 finally 中的语句后，程序就会立即停止了。所以，在实际使用中，最好一直带着 catch 语句。如果你写了 catch 语句，则finally 语句也是可以省略的。

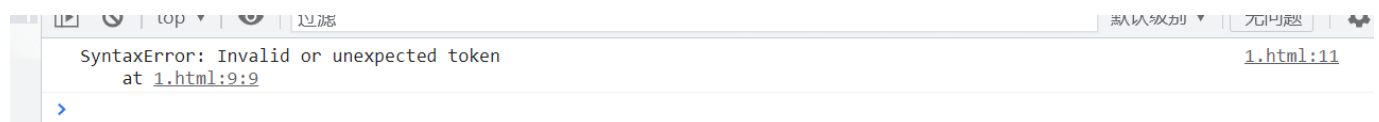
### 6.1.3、异常演示

#### 6.1.3.1、Eval 错误

EvalError 指示 eval() 函数中的错误。更新版本的 JavaScript 不会抛出任何 EvalError，请使用 SyntaxError 代替。

案例演示：

```
<script>
  try {
    eval("alert('Hello')"); // 缺少 ' 会产生错误
  } catch (error) {
    console.log(error)
  }
</script>
```

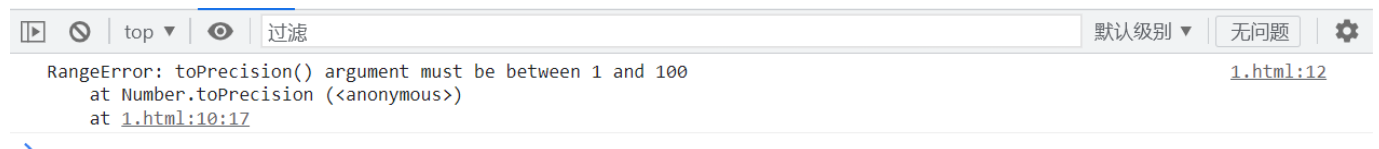


#### 6.1.3.2、范围错误

RangeError 会在您使用了合法值的范围之外的数字时抛出。

案例演示：您不能将数字的有效位数设置为 500。

```
<script>
  var num = 1;
  try {
    num.toPrecision(500); // 数无法拥有 500 个有效数
  } catch (error) {
    console.log(error)
  }
</script>
```

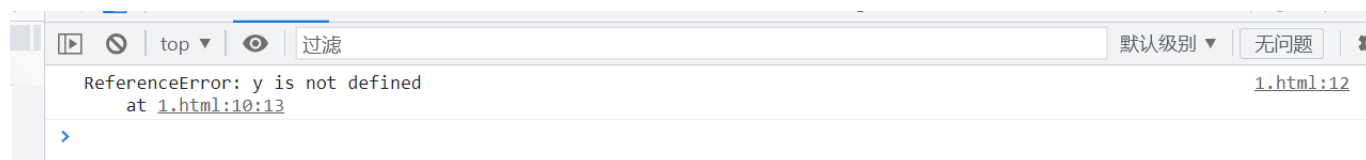


### 6.1.3.3、引用错误

假如您使用（引用）了尚未声明的变量，则 `ReferenceError` 会被抛出：

案例演示：

```
<script>
  var x;
  try {
    x = y + 1;    // y 无法被引用（使用）
  } catch (error) {
    console.log(error)
  }
</script>
```

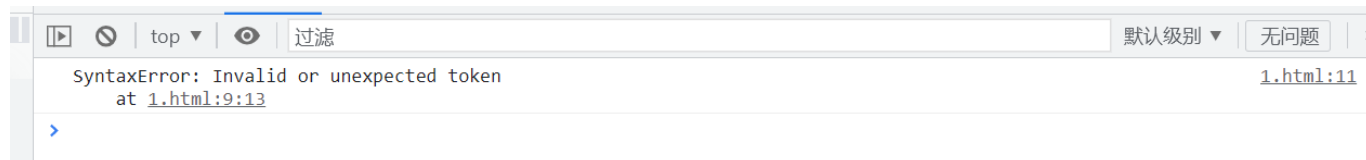


### 6.1.3.4、语法错误

假如您计算带语法错误的代码，会 `SyntaxError` 被抛出：

案例演示：

```
<script>
  try {
    eval("alert('Hello)"); // 缺少 ' 会产生错误
  } catch (error) {
    console.log(error)
  }
</script>
```

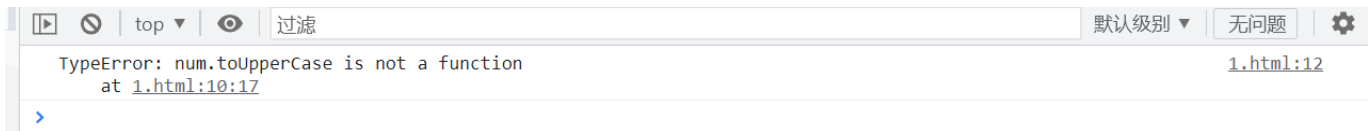


### 6.1.3.5、类型错误

假如您使用的值不在期望值的范围之内，则 `TypeError` 被抛出：

案例演示：

```
<script>
  var num = 1;
  try {
    num.toUpperCase();    // 您无法将数字转换为大写
  } catch (error) {
    console.log(error)
  }
</script>
```

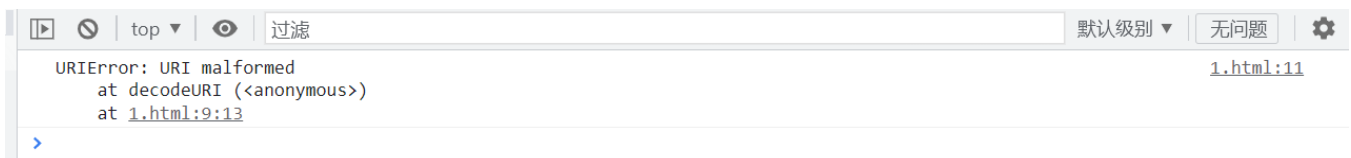


#### 6.1.3.6、URI 错误

假如您在 URI 函数中使用非法字符，则 URIError 被抛出：

案例演示：

```
<script>
  try {
    decodeURI("%%%");    // 您无法对这些百分号进行 URI 编码
  } catch (error) {
    console.log(error)
  }
</script>
```



#### 6.1.4、异常抛出

在大部分的代码执行过程中，都是出现错误的时候，由浏览器(javascript引擎)抛出异常，然后程序或者停止执行或被try...catch 捕获。

然而有时候我们在检测到一些不合理的情况发生的时候也可以主动抛出错误，请使用 throw 关键字抛出来主动抛出异常。

注意事项：

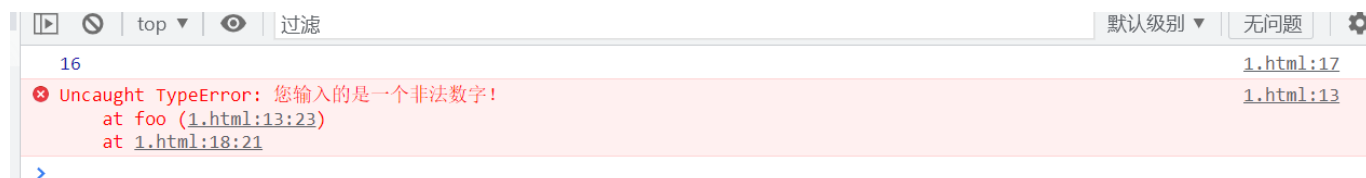
1.throw后面就是我们要抛出的异常对象，在以前的时候都是出现错误的时候浏览器抛出异常对象，只是现在是我们自己主动抛出的异常对象。 2.只要有异常对象抛出，不管是浏览器抛出的，还是代码主动抛出，都会让程序停止执行。如果想让程序继续执行，则有也可以用try...catch来捕

获。3.每一个错误类型都可以传入一个参数，表示实际的错误信息。4.我们可以在适当的时候抛出任何我们想抛出的异常类型。throw new SyntaxError("语法错误...");

#### 6.1.4.1、主动抛出内置异常

```
<script>
  /*该函数接收一个数字，返回它的平方。*/
  function foo(num) {
    if (typeof num == "number") {
      return num * num;
    } else {
      throw new TypeError("您输入的是一个非法数字！")
    }
  }

  console.log(foo(4));
  console.log(foo("abc"));
</script>
```



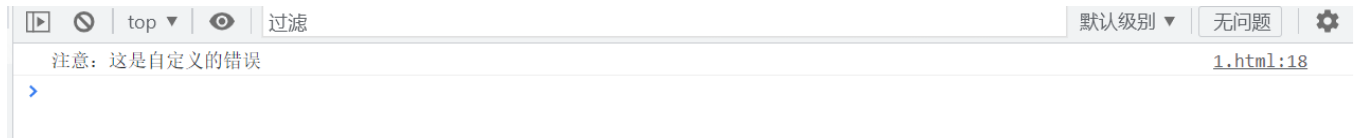
#### 6.1.4.2、主动抛出自定义异常

我们不仅仅可以抛出js内置的错误类型的对象，也可以自定义错误类型，然后抛出自定义错误类型的对象。

如果要自定义错误类型，只需要继承任何一个自定义错误类型都可以，一般直接继承Error即可。

```
<script>
  /*自定义错误*/
  function MyError(message) {
    this.message = "注意：这是自定义的错误"
    this.name = "自定义错误";
  }
  MyError.prototype = new Error();

  try {
    throw new MyError("注意：这是自定义错误类型")
  } catch (error) {
    console.log(error.message)
  }
</script>
```



## 6.2、JSON

### 6.2.1、JSON概述

JSON: JavaScript Object Notation (JavaScript 对象标记法)，它是一种存储和交换数据的语法。

当数据在浏览器与服务器之间进行交换时，这些数据只能是文本，JSON 属于文本并且我们能够把任何 JavaScript 对象转换为 JSON，然后将 JSON 发送到服务器。我们也能把从服务器接收到的任何 JSON 转换为 JavaScript 对象。以这样的方式，我们能够把数据作为 JavaScript 对象来处理，无需复杂的解析和转译。

### 6.2.2、JSON语法

在json中，每一个数据项，都是由一个键值对（或者说是名值对）组成的，但是键必须是字符串，且由双引号包围，而值必须是以下数据类型之一：

字符串（在 JSON 中，字符串值必须由双引号编写）

- 数字
- 对象（JSON 对象）
- 数组
- 布尔
- null

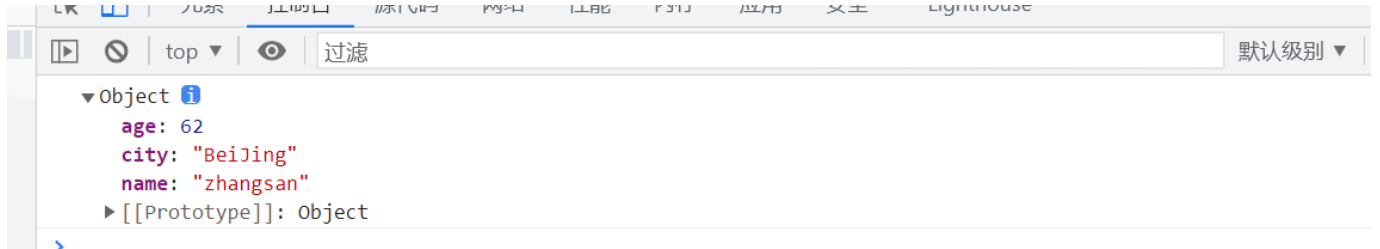
JSON 的值不可以是以下数据类型之一：

- 函数
- 日期
- undefined

因为 JSON 语法由 JavaScript 对象标记法衍生而来，所以很少需要其它额外的软件来处理 JavaScript 中的 JSON。

通过 JavaScript，您能够创建对象并向其分配数据，就像这样：

```
<script>
  var person = {"name": "zhangsan", "age": 62, "city": "Beijing"};
  console.log(person);
</script>
```



### 6.2.3、JSON数据类型

#### 6.2.3.1、JSON 字符串

JSON 中的字符串必须用双引号包围。

```
{"name": "John"}
```

#### 6.2.3.2、JSON 数字

JSON 中的数字必须是整数或浮点数。

```
{"age": 30}
```

#### 6.2.3.3、JSON 对象

JSON 中的值可以是对象，JSON 中作为值的对象必须遵守与 JSON 对象相同的规则。

```
{
  "employee": {"name": "Bill Gates", "age": 62, "city": "Seattle"}
}
```

#### 6.2.3.4、JSON 数组

JSON 中的值可以是数组。

```
{
  "employees": ["Bill", "Steve", "David"]
}
```



#### 6.2.3.5、JSON 布尔

JSON 中的值可以是 true/false。

```
{"sale": true}
```

#### 6.2.3.6、JSON null

JSON 中的值可以是 null。

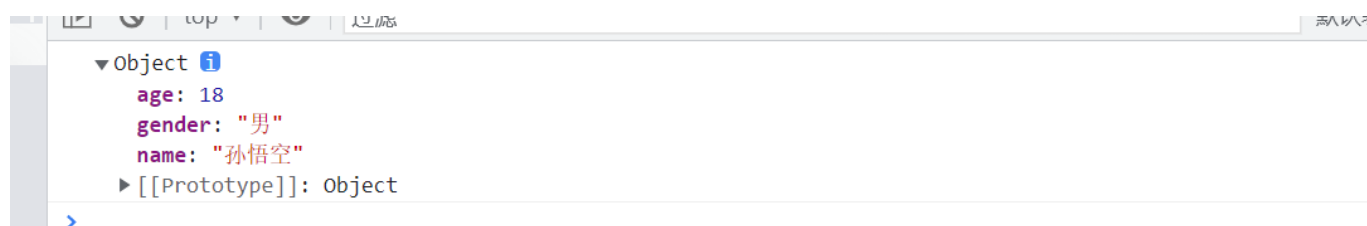
```
{"middlename": null}
```

#### 6.2.4、JSON字符串转JS对象

JSON.parse(): 可以将以JSON字符串转换为JS对象，它需要一个JSON字符串作为参数，会将该字符串转换为JS对象并返回

案例演示：

```
<script>
  var jsonStr = '{"name":"孙悟空","age":18,"gender":"男"}';
  var obj = JSON.parse(jsonStr);
  console.log(obj);
</script>
```



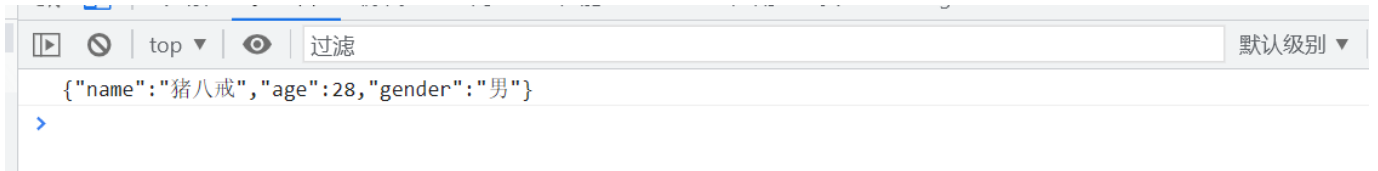
注意：JSON这个对象在IE7及以下的浏览器中不支持，所以在这些浏览器中调用时会报错

#### 6.2.5、JS对象转JSON字符串

JSON.stringify(): 可以将一个JS对象转换为JSON字符串，需要一个js对象作为参数，会返回一个JSON字符串

案例演示：

```
<script>
    var obj = {name: "猪八戒", age: 28, gender: "男"};
    var jsonStr = JSON.stringify(obj);
    console.log(jsonStr);
</script>
```



注意：JSON这个对象在IE7及以下的浏览器中不支持，所以在这些浏览器中调用时会报错

## 6.3、AJAX

### 6.3.1、AJAX概述

传统的web交互是用户触发一个http请求服务器，然后服务器收到之后，在做出响应到用户，并且返回一个新的页面，每当服务器处理客户端提交的请求时，客户都只能空闲等待，并且哪怕只是一次很小的交互、只需从服务器端得到很简单的一个数据，都要返回一个完整的HTML页，而用户每次都要浪费时间和带宽去重新读取整个页面。这个做法浪费了许多带宽，由于每次应用的交互都需要向服务器发送请求，应用的响应时间就依赖于服务器的响应时间，这导致了用户界面的响应比本地应用慢得多。

AJAX 的出现刚好解决了传统方法的缺陷，AJAX 是一种用于创建快速动态网页的技术，通过在后台与服务器进行少量数据交换，AJAX 可以使网页实现异步更新，这意味着可以在不重新加载整个网页的情况下，对网页的某部分进行更新。

### 6.3.2、AJAX的XMLHttpRequest对象

AJAX 的核心是 XMLHttpRequest 对象。所有现代浏览器都支持 XMLHttpRequest 对象。

XMLHttpRequest 对象用于幕后同服务器交换数据，这意味着可以更新网页的部分，而不需要重新加载整个页面。

所有现代浏览器（Chrom、IE7+、Firefox、Safari 以及 Opera）都有内建的 XMLHttpRequest 对象。

创建 XMLHttpRequest 的语法是：

```
<script>
    variable = new XMLHttpRequest();
</script>
```

老版本的 Internet Explorer（IE5 和 IE6）使用 ActiveX 对象：

```
<script>
    variable = new ActiveXObject("Microsoft.XMLHTTP");
</script>
```

为了应对所有浏览器，包括 IE5 和 IE6，请检查浏览器是否支持 XMLHttpRequest 对象。如果支持，创建 XMLHttpRequest 对象，如果不支持，则创建 ActiveX 对象：

```
<script>
    var xhttp;
    if (window.XMLHttpRequest) {
        xhttp = new XMLHttpRequest();
    } else {
        // code for IE6, IE5
        xhttp = new ActiveXObject("Microsoft.XMLHTTP");
    }
</script>
```

但是需要注意的是，出于安全原因，现代浏览器不允许跨域访问，这意味着尝试加载的网页和 XML 文件都必须位于相同服务器上。

6.3.3、AJAX的XMLHttpRequest对象方法

方法	描述
new XMLHttpRequest()	创建新的 XMLHttpRequest 对象
abort()	取消当前请求
getAllResponseHeaders()	返回头部信息
getResponseHeader()	返回特定的头部信息
open( <i>method, url, async, user, psw</i> )	规定请求method：请求类型 GET 或 POST url：文件位置 async：true（异步）或 false（同步） user：可选的用户名称 psw：可选的密码
send()	将请求发送到服务器，用于 GET 请求
send( <i>string</i> )	将请求发送到服务器，用于 POST 请求
setRequestHeader()	向要发送的报头添加标签/值对

6.3.4、AJAX的XMLHttpRequest对象属性

属性	描述
onreadystatechange	定义当 readyState 属性发生变化时被调用的函数
readyState	保存 XMLHttpRequest 的状态。 0: 请求未初始化 1: 服务器连接已建立 2: 请求已收到 3: 正在处理请求 4: 请求已完成且响应已就绪
responseText	以字符串返回响应数据
responseXML	以 XML 数据返回响应数据
status	返回请求的状态号 200: "OK" 403: "Forbidden" 404: "Not Found" 如需完整列表请访问 <a href="#">Http 消息参考手册</a>
statusText	返回状态文本（比如 "OK" 或 "Not Found"）

6.3.5、AJAX的GET请求

6.3.6、AJAX的POST请求

6.3.7、AJAX的请求整合

[学习JavaScript这一篇就够了\\_javascript学习这一篇就够了-CSDN博客](#)

6.4、Cookie

6.4.1、Cookie概述

Cookie 是一些数据，存储于你电脑上的文本文件中，当 web 服务器向浏览器发送 web 页面时，在连接关闭后，服务端不会记录用户的信息，Cookie 的作用就是用于解决 “如何记录客户端的用户信息”：

当用户访问 web 页面时，它的名字可以记录在 cookie 中。在用户下一次访问该页面时，可以在 cookie 中读取用户访问记录。Cookie 以名/值对形式存储，如下所示：

```
username=zhangsan
```

当浏览器从服务器上请求 web 页面时，属于该页面的 cookie 会被添加到该请求中，服务端通过这种方式来获取用户的信息。

JavaScript 可以使用 document.cookie 属性来创建、读取、及删除 Cookie。

#### 6.4.2、Cookie创建

JavaScript 中，创建 cookie 如下所示：

```
<script>
    document.cookie = "username=zhangsan";
</script>
```

您还可以为 cookie 添加一个过期时间（以 UTC 或 GMT 时间）。默认情况下，cookie 在浏览器关闭时删除。

```
<script>
    document.cookie = "username=zhangsan;
                        expires=Thu, 18 Dec 2043 12:00:00 GMT";
</script>
```

您可以使用 path 参数告诉浏览器 cookie 的路径。默认情况下，cookie 属于当前页面。

```
<script>
    document.cookie = "username=zhangsan;
                        expires=Thu, 18 Dec 2043 12:00:00 GMT; path=/";
</script>
```

#### 6.4.3、Cookie读取

JavaScript 中，读取 cookie 如下所示：

document.cookie 将以字符串的方式返回所有的 cookie，类型格式： cookie1=value; cookie2=value; cookie3=value;

```
<script>
    document.cookie = "username=zhangsan";
    var cookies = document.cookie;
    console.log(cookies);
</script>
```

#### 6.4.4、Cookie修改

JavaScript 中，修改 cookie 如下所示：

使用 document.cookie 将旧的 cookie 将被覆盖就是修改。

```
<script>
    document.cookie = "username=zhangsan";
    document.cookie = "username=lisi";
    var cookies = document.cookie;
    console.log(cookies);
</script>
```

#### 6.4.5、Cookie删除

JavaScript 中，删除 cookie 如下所示：

删除 cookie 非常简单，您只需要设置 expires 参数为以前的时间即可，如下所示，设置为 Thu, 01 Jan 1970 00:00:00 GMT:

```
<script>
    document.cookie = "username=zhangsan";
    document.cookie = "username=; expires=Thu,
                        01 Jan 1970 00:00:00 GMT";
    var cookies = document.cookie;
    console.log(cookies);
</script>
```

#### 6.4.6、Cookie值设置函数

```
<script>
    /**
     * Cookie值设置函数
     * @param cname    cookie名称
     * @param cvalue    cookie值
     * @param exdays    过期天数
     */
    function setCookie(cname, cvalue, exdays) {
        var d = new Date();
        d.setTime(d.getTime() + (exdays * 24 * 60 * 60 * 1000));
        var expires = "expires=" + d.toGMTString();
        document.cookie = cname + "=" + cvalue + "; " + expires;
    }
</script>
```

#### 6.4.7、Cookie值获取函数

```

<script>
    /**
     * Cookie值获取函数
     * @param cname    cookie名称
     * @returns {string}
     */
    function getCookie(cname) {
        var name = cname + "=";
        var ca = document.cookie.split(';');
        for (var i = 0; i < ca.length; i++) {
            var c = ca[i].trim();
            if (c.indexOf(name) == 0)
                return c.substring(name.length, c.length);
        }
        return "";
    }

</script>

```

## 6.5、WebStorage

### 6.5.1、WebStorage概述

WebStorage是HTML5中本地存储的解决方案之一，在HTML5的WebStorage概念引入之前除去IE User Data、Flash Cookie、Google Gears等看名字就不靠谱的解决方案，浏览器兼容的本地存储方案只有使用Cookie。有同学可能会问，既然有了Cookie本地存储，为什么还要引入WebStorage的概念？那就要说一说Cookie的缺陷了：

- 1.数据大小：作为存储容器，Cookie的大小限制在4KB左右这是非常坑爹的，尤其对于现在复杂的业务逻辑需求，4KB的容量除了存储一些配置字段还简单单值信息，对于绝大部分开发者来说真的不知指望什么了。
- 2.安全性问题：由于在HTTP请求中的Cookie是明文传递的（HTTPS不是），带来的安全性问题还是很大的。
- 3.网络负担：我们知道Cookie会被附加在每个HTTP请求中，在HttpRequest和HttpResponse的header中都是要被传输的，所以无形中增加了一些不必要的流量损失。

虽然WebStorage是HTML5新增的本地存储解决方案之一，但并不是为了取代Cookie而制定的标准，Cookie作为HTTP协议的一部分用来处理客户端和服务端通信是不可或缺的，session正是依赖于实现的客户端状态保持。WebStorage的意图在于解决本来不应该Cookie做，却不得不用Cookie的本地存储的应用场景。

### 6.5.2、WebStorage分类

Web Storage又分为两种： `sessionStorage` 和 `localStorage`，即这两个是Storage的一个实例。从字面意思就可以很清楚的看出来，`sessionStorage`将数据保存在session中，浏览器关闭也就没了；而`localStorage`则一直将数据保存在客户端本地；不管是`sessionStorage`，还是`localStorage`，使用的API都相同。

`localStorage`和`sessionStorage`只能存储字符串类型，对于复杂的对象可以使用ECMAScript提供的JSON对象的`stringify`和`parse`来处理，低版本IE可以使用`json2.js`

### 6.5.3、localStorage方法

对象介绍：

`localStorage`在本地永久性存储数据，除非显式将其删除或清空。

常见方法：

- 保存单个数据：`localStorage.setItem(key,value);`
- 读取单个数据：`localStorage.getItem(key);`
- 删除单个数据：`localStorage.removeItem(key);`
- 删除所有数据：`localStorage.clear();`
- 获取某个索引的key：`localStorage.key(index);`

案例演示：



```
<script>
// 保存数据
localStorage.setItem("username", "zhangsan");

// 读取单个数据
console.log(localStorage.getItem("username"));
console.log("=====");

// 删除单个数据
localStorage.removeItem("username");
console.log(localStorage.getItem("username"));
console.log("=====");

// 保存两个数据
localStorage.setItem("age", 18);
localStorage.setItem("sex", "男");
console.log("age=" + localStorage.getItem("age"));
console.log("sex=" + localStorage.getItem("sex"));
console.log("=====");

// 使用for-in循环来迭代localStorage中的键值对、属性和方法:
for (var key in localStorage) {
    console.log(key + "=" + localStorage[key]);
}
console.log("=====");

// 使用for循环来迭代localStorage中的键值对:
for (var i = 0; i < localStorage.length; i++) {
    var key = localStorage.key(i);
    var value = localStorage.getItem(key);
    console.log(key + "=" + value);
}
console.log("=====");

// 删除所有数据
localStorage.clear();
</script>
```



#### 6.5.4、sessionStorage方法

对象介绍：

sessionStorage对象存储特定于某个对话的数据，也就是它的生命周期为当前窗口或标签页，一旦窗口或标签页被永久关闭了，那么所有通过sessionStorage存储的数据也就被清空了。存储在sessionStorage中的数据可以跨越页面刷新而存在，同时如果浏览器支持，浏览器崩溃并重启之后依然可以使用（注意：Firefox和Webkit都支持，IE则不行）。

因为sessionStorage对象绑定于某个服务器会话，所以当文件在本地运行的时候是不可用的。存储在sessionStorage中的数据只能由最初给对象存储数据的页面访问到，所以对多页面应用有限制。

不同浏览器写入数据方法略有不同。Firefox和Webkit实现了同步写入，所以添加到存储空间中的数据是立刻被提交的。而IE的实现则是异步写入数据，所以在设置数据和将数据实际写入磁盘之间可能有一些延迟。

常见方法：

- 保存单个数据：sessionStorage.setItem(key,value);
- 读取单个数据：sessionStorage.getItem(key);
- 删除单个数据：sessionStorage.removeItem(key);

- 删除所有数据: `sessionStorage.clear()`;
- 获取某个索引的key: `sessionStorage.key(index)`;

案例演示:

```
<script>
    // 保存数据
    sessionStorage.setItem("username", "zhangsan");

    // 读取单个数据
    console.log(sessionStorage.getItem("username"));
    console.log("=====");

    // 删除单个数据
    sessionStorage.removeItem("username");
    console.log(sessionStorage.getItem("username"));
    console.log("=====");

    // 保存两个数据
    sessionStorage.setItem("age", 18);
    sessionStorage.setItem("sex", "男");
    console.log("age=" + sessionStorage.getItem("age"));
    console.log("sex=" + sessionStorage.getItem("sex"));
    console.log("=====");

    // 使用for-in循环来迭代sessionStorage中的键值对、属性和方法:
    for (var key in sessionStorage) {
        console.log(key + "=" + sessionStorage[key]);
    }
    console.log("=====");

    // 使用for循环来迭代sessionStorage中的键值对:
    for (var i = 0; i < sessionStorage.length; i++) {
        var key = sessionStorage.key(i);
        var value = sessionStorage.getItem(key);
        console.log(key + "=" + value);
    }
    console.log("=====");

    // 删除所有数据
    sessionStorage.clear();

</script>
```

```

zhangsan
=====
null
=====
age=18
sex=男
=====
age=18
sex=男
length=2
clear=function clear() { [native code] }
getItem=function getItem() { [native code] }
key=function key() { [native code] }
removeItem=function removeItem() { [native code] }
setItem=function setItem() { [native code] }
=====
age=18
sex=男
=====

```

## 6.6、Closure

### 6.6.1、闭包引入

需求信息：点击某个按钮，提示"点击的是第n个按钮"

**第一种解决方法：将btn所对应的下标保存在btn上**

```

<script>
    var btns = document.getElementsByTagName('button');

    //将btn所对应的下标保存在btn上
    for (var i = 0, length = btns.length; i < length; i++) {
        var btn = btns[i];
        btn.index = i;
        btn.onclick = function () {
            alert('第' + (this.index + 1) + '个');
        }
    }

</script>

```

**第二种解决方法：利用闭包延长局部变量的生命周期**

```
<script>
  var btns = document.getElementsByTagName('button');

  // 利用闭包延长局部变量的生命周期
  for (var i = 0, length = btns.length; i < length; i++) {
    (function (j) {
      var btn = btns[j];
      btn.onclick = function () {
        alert('第' + (j + 1) + '个');
      }
    })(i);
  }
</script>
```

### 6.6.2、闭包概念

- 如何产生闭包？

- 当一个嵌套的内部(子)函数引用了嵌套的外部(父)函数的变量(函数)时，就产生了闭包

- 什么才是闭包？

- 理解一：闭包是嵌套的内部函数(绝大部分人认为)
- 理解二：包含被引用变量(函数)的对象(极少部分人认为)

- 闭包的作用？

- 它的最大用处有两个，一个是读取函数内部的变量，另一个就是让这些变量的值始终保持在内存中

### 6.6.3、闭包演示

```

<script>
    function fun1() {
        var a = 2;
        function subFun() {
            a++;
            console.log(a);
        }
        return subFun;
    }

    var f1 = fun1();
    f1();//3
    f1();//4
    console.log("=====");

    function fun2() {
        var a = 2;
        function subFun() {
            a--;
            console.log(a);
        }
        return subFun;
    }

    var f2 = fun2();
    f2();//1
    f2();//0
    console.log("=====");

</script>

```

#### 6.6.4、闭包生命周期

生命周期：

- 产生：在嵌套内部函数定义执行完时就产生了(不是在调用)
- 死亡：在嵌套的内部函数成为垃圾对象时就死亡了

演示说明：

```

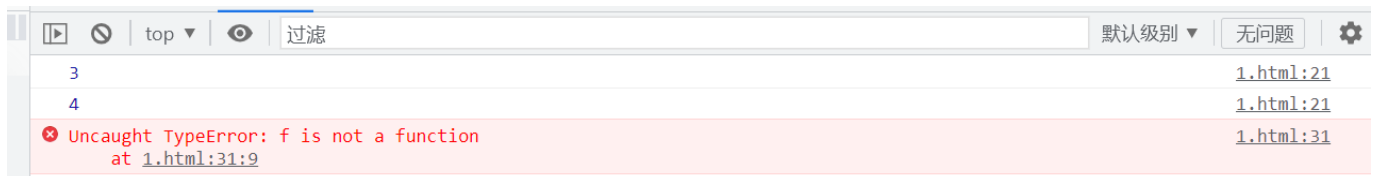
<script>
  function fn1() {
    //此时闭包就已经产生了(函数提升，内部函数对象已经创建了)
    var a = 2;

    function fn2() {
      a++;
      console.log(a);
    }

    return fn2;
  }

  var f = fn1();
  f(); // 3
  f(); // 4
  f = null; //闭包死亡(包含闭包的函数对象成为垃圾对象)
  f();
</script>

```



### 6.6.5、闭包应用

闭包应用：定义JS模块

具有特定功能的js文件 将所有数据和功能都封装在一个函数内部(私有的) 只向外暴露一个包含n个方法的对象或函数 模块的使用者，只需要通过模块暴露的对象调用方法来实现对应的功能 案例演示：

第一种格式：myModule.js

```

<script>
  function myModule() {
    //私有数据
    var msg = 'Hello, World';

    //操作数据的函数
    function doSomething() {
      console.log('doSomething() ' + msg.toUpperCase());
    }

    function doOtherthing() {
      console.log('doOtherthing() ' + msg.toLowerCase());
    }

    //向外暴露对象(给外部使用的方法)
    return {
      doSomething: doSomething,
      doOtherthing: doOtherthing
    }
  }

  var module = myModule();
  module.doSomething();
  module.doOtherthing();

</script>

```

<div> <div></div> <div></div> <div>top ▾</div> <div></div> <div>过滤</div> </div>	默认级别 ▾
doSomething() HELLO, WORLD	:
doOtherthing() hello, world	:
>	

第二种格式: myModule.js



```
<script>
  (function (window) {
    //私有数据
    var msg = 'Hello, World';




    //操作数据的函数
    function doSomething() {
      console.log('doSomething() ' + msg.toUpperCase());
    }

    function doOtherthing() {
      console.log('doOtherthing() ' + msg.toLowerCase());
    }

    //向外暴露对象(给外部使用的方法)
    window.myModule = {
      doSomething: doSomething,
      doOtherthing: doOtherthing
    }
  })(window);

  myModule.doSomething();
  myModule.doOtherthing();

</script>
```

		top ▼		过滤	默认级别 ▼
doSomething() HELLO, WORLD					
doOtherthing() hello, world					
>					