目录

[HTML中的window对象和document对象详解 - 1 -](#_Toc498616053)

[Iframe - 4 -](#_Toc498616054)

[void操作符用法格式如下： - 5 -](#_Toc498616055)

[FileReader - 6 -](#_Toc498616056)

[状态常量readyState - 7 -](#_Toc498616057)

[属性 - 7 -](#_Toc498616058)

[jQuery获取页面及个元素高度、宽度 - 10 -](#_Toc498616059)

[jquery - 10 -](#_Toc498616060)

[html - 10 -](#_Toc498616061)

[css - 10 -](#_Toc498616062)

[屏幕宽高 - 10 -](#_Toc498616063)

[可视区域宽高 - 10 -](#_Toc498616064)

[[HTML5] Blob对象 12](#_Toc498616065)

[写在前面 12](#_Toc498616066)

[创建Blob 12](#_Toc498616067)

[通过构造函数 12](#_Toc498616068)

[通过Blob.slice() 13](#_Toc498616069)

[通过canvas.toBlob() 14](#_Toc498616070)

[应用场景 14](#_Toc498616071)

[分片上传 14](#_Toc498616072)

[通过url下载文件 17](#_Toc498616073)

[通过url显示图片 18](#_Toc498616074)

[小结 18](#_Toc498616075)

[fetch.js 19](#_Toc498616076)

[XMLHttpRequest 19](#_Toc498616077)

[响应的对象Response类型 20](#_Toc498616078)

[兼容浏览器的处理 22](#_Toc498616079)

[3.2 一般构造请求的方法 23](#_Toc498616080)

[3.3 fetch 构成函数的其他选项 23](#_Toc498616081)

[3.4 返回的数据结构 24](#_Toc498616082)

[3.5 Body 参数 25](#_Toc498616083)

[串联 Promises 26](#_Toc498616084)

[用fetch执行表单数据提交 27](#_Toc498616085)

[在Fetch请求里发送用户身份凭证信息 28](#_Toc498616086)

[**特性检测** 28](#_Toc498616087)

[**简单的fetching示例** 29](#_Toc498616088)

[Headers 30](#_Toc498616089)

[Request 32](#_Toc498616090)

[Response 35](#_Toc498616091)

[处理body 37](#_Toc498616092)

[**流和克隆** 38](#_Toc498616093)

[**未来的改进** 40](#_Toc498616094)

[New formdata() 40](#_Toc498616095)

[FormData 上传文件实例 40](#_Toc498616096)

[FormData 对象方法介绍 45](#_Toc498616097)

[FormData兼容性问题 50](#_Toc498616098)

[Touch events 51](#_Toc498616099)

[URLSearchParams 接口 51](#_Toc498616100)

[JS与DOM属性和方法 54](#_Toc498616101)

[http 56](#_Toc498616102)

# [HTML中的window对象和document对象详解](http://blog.csdn.net/mafan121/article/details/51088975)

Window -- 代表浏览器中一个打开的窗口：

对象属性  
window //窗口自身  
**window.self** //引用本窗户window=window.self  
window.name //为窗口命名  
window.defaultStatus //设定窗户状态栏信息  
window.location //URL地址，配备布置这个属性可以打开新的页面

对象方法  
window.alert("text") //提示信息会话框  
window.confirm("text") //确认会话框  
window.prompt("text") //要求键盘输入会话框  
window.setIntervel("action",time) //每一隔指定的时间(毫秒)就执行一次操作  
window.clearInterval() //清除时间配备布置作用就是终止轮回  
window.setTimeout(action,time) //隔了指定的时间(毫秒)执行一次操作  
window.open() //打开新的窗口  
window.close() //关闭窗口

成员对象  
window.event  
window.document //见document对象详解  
window.history  
window.screen  
window.navigator  
**window.external**  
-------------------------------------------------- -------------------  
**window.history对象**  
window.history.length //浏览过的页面数  
history.back() //后退  
history.forward() //前进  
history.[go](http://lib.csdn.net/base/go)(i) //前进或后退到历史记录的第i个页面  
//i>0进步,i<0 后退  
   ------------------------------------------------- -------------------  
**window.screen对象**  
 window.screen.width //屏幕宽度  
window.screen.height //屏幕高度  
window.screen.colorDepth //屏幕色深  
window.screen.availWidth //可用宽度  
window.screen.availHeight //可用高度(除去任务栏的高度)  
-------------------------------------------------- -------------------  
**window.external对象**  
window.external.AddFavorite("地址","标题" ) //把网站新增到保藏夹  
-------------------------------------------------- -------------------  
**window.navigator对象**  
window.navigator.appCodeName //浏览器代码名  
window.navigator.appName //浏览器应用程序名  
window.navigator.appMinorVersion //浏览器补丁版本  
window.navigator.cpuClass //cpu类型 x86  
window.navigator.platform //操作体系类型 win32  
window.navigator.plugins  
window.navigator.opsProfile  
window.navigator.userProfile  
window.navigator.systemLanguage //客户体系语言 zh-cn简体中文  
window.navigator.userLanguage //用户语言,同上  
window.navigator.appVersion //浏览器版本  
window.navigator.userAgent  
window.navigator.onLine //用户否在线  
window.navigator.cookieEnabled //浏览器是否撑持cookie  
window.navigator.mimeTypes

document对象 -- 代表整个HTML 文档,可用来访问页面中的所有元素：

对象属性

document.title                 //设置文档标题等价于HTML的<title>标签  
document.bgColor               //设置页面背景色  
document.fgColor               //设置前景色(文本颜色)  
**document.linkColor             //未点击过的链接颜色  
document.alinkColor            //激活链接(焦点在此链接上)的颜色  
document.vlinkColor            //已点击过的链接颜色  
document.URL                   //设置URL属性从而在同一窗口打开另一网页  
document.fileCreatedDate       //文件建立日期，只读属性  
document.fileModifiedDate      //文件修改日期，只读属性  
document.fileSize              //文件大小，只读属性  
document.cookie                //设置和读出cookie  
document.charset               //设置字符集 简体中文:gb2312**

常用对象方法

document.write()                      //动态向页面写入内容  
document.createElement(Tag)           //创建一个html标签对象  
document.getElementById(ID)           //获得指定ID值的对象  
document.getElementsByName(Name)      //获得指定Name值的对象  
document.body.appendChild(oTag)

body-主体子对象

document.body                   //指定文档主体的开始和结束等价于<body></body>  
document.body.bgColor           //设置或获取对象后面的背景颜色  
document.body.link              //未点击过的链接颜色  
document.body.alink             //激活链接(焦点在此链接上)的颜色  
document.body.vlink             //已点击过的链接颜色  
document.body.text              //文本色  
document.body.innerText         //设置<body>...</body>之间的文本  
document.body.innerHTML         //设置<body>...</body>之间的HTML代码  
document.body.topMargin         //页面上边距  
document.body.leftMargin        //页面左边距  
document.body.rightMargin       //页面右边距  
document.body.bottomMargin      //页面下边距  
document.body.background        //背景图片  
document.body.appendChild(oTag) //动态生成一个HTML对象

常用对象事件

document.body.onclick="func()"              //鼠标指针单击对象是触发  
document.body.onmouseover="func()"          //鼠标指针移到对象时触发  
document.body.onmouseout="func()"           //鼠标指针移出对象时触发

location-位置子对象

document.location.hash          // #号后的部分  
document.location.host          // 域名+端口号  
document.location.hostname      // 域名  
document.location.href          // 完整URL  
document.location.pathname      // 目录部分  
document.location.port          // 端口号  
document.location.protocol      // 网络协议(http:)  
document.location.search        // ?号后的部分

常用对象事件

documeny.location.reload()          //刷新网页  
document.location.reload(URL)       //打开新的网页  
document.location.assign(URL)       //打开新的网页  
document.location.replace(URL)      //打开新的网页  
========================================================================  
selection-选区子对象  
document.selection  
========================================================================  
images集合(页面中的图象):  
----------------------------  
a)通过集合引用  
document.images                 //对应页面上的<img>标签  
document.images.length          //对应页面上<img>标签的个数  
document.images[0]              //第1个<img>标签             
document.images[i]              //第i-1个<img>标签  
----------------------------  
b)通过nane属性直接引用  
<img name="oImage">  
document.images.oImage          //document.images.name属性  
----------------------------  
c)引用图片的src属性  
document.images.oImage.src      //document.images.name属性.src

# **Iframe**

是个比较新的标识。但是从它的参数来说和普通的html标识没有什么大的区别，相信大家都用过frame（框架）来布局你的网页，其实iframe和frame功能一样，不同的是它是个浮动框架，你可以把iframe布置在你网页中的任何位置，包括

层中，这种极大的自由度可以给网页设计带来很大的灵活性，所以学会使用它是非常必要的。  
下面介绍它的主要属性  
Name="" 给框架命名，这样就可以使用你的命名为链接中的target属性提供参数  
Src="" 当前框架所链接的页面地址  
Frameborder="" 浮动框架的边框大小，默认值为1显示边框，一般最好设为0不显示  
Marginwidth="" 浮动框架框边与插入页面之间空白的宽度，单位pixels  
Marginheight="" 浮动框架框边与插入页面空白的高度，单位pixels  
Scrolling="" 滚动条，有3个值：auto自动，yes 总是显示，no 不显示  
align="MIDDLE" 可选值为 left, right, top, middle, bottom，作用不大  
width=""  
height="" 浮动框架的宽及长，单位pixels。  
Noresize 浮动框架大小不可变  
其它的属性就是id ,class ,title ,style。

# void操作符用法格式如下：

1. javascript:void (expression)  
2. javascript:void expression

expression 是一个要计算的 Javascript 标准的表达式。表达式外侧的圆括号是选的，但是写上去是一个好习惯。 (实现版本   Navigator 3.0   ) 1.\_blank       <a href="document.html" target="\_blank">my document</a>         浏览器会另开一个新窗口显示document.html文档

2.\_parent      <a href="document.html" target="\_parent">my document</a>       指向父frameset文档

3.\_self          <a href="document.html" target="\_self">my document</a>           把文档调入当前页框

4.\_top          <a href="document.html" target="\_top">my document</a>            去掉所有页框并用

你以使用 void 操作符指定超级链接。表达式会被计算但是不会当前文档处装入任何内容。

下面的代码创建了一个超级链接，当用户以后不会发生任何事。当用户链接时，void(0) 计算为 0，但 Javascript 上没有任何效果。

<A HREF="javascript:void(0)">单此处什么也不会发生</A>

下面的代码创建了一个超级链接，用户单时会提交表单。

<A HREF="javascript:void(document.form.submit())">单此处提交表单</A>

href=”#”,包含了一个位置信息.默认的锚是#top,也就是网页的上端，当连续快速点击此链接时会导致浏览器巨慢甚至崩溃。  
当然我们一般用三个 href="###",不过看了这篇文章我们以后就可以使用javascript:;(一个冒号一个分号)

# FileReader

因为Google还不提供同步插件数据的功能，所以导入和导出插件配置就必须和文件打交道了。而出于安全原因，只有IE才提供访问文件的API；但随着HTML 5的到来，其他浏览器也纷纷支持了。一个 Blob对象表示一个不可变的, 原始数据的类似文件对象。Blob表示的数据不一定是一个JavaScript原生格式。 File 接口基于Blob，继承 blob功能并将其扩展为支持用户系统上的文件。

首先说读取文件。W3C提供了一些File API，其中最重要的是FileReader这个类。

先列出需要用到的HTML标签：

复制代码 代码如下:

<input type="file" id="file" onchange="handleFiles(this.files)"/>

当选择了一个文件时，就会把包含这个文件的列表（一个FileList对象）作为参数传给handleFiles()函数了。  
这个FileList对象类似一个数组，可以知道文件的数目，而它的元素就是File对象了。  
从这个File对象可以获取name、size、lastModifiedDate和type等属性。  
把这个File对象传给FileReader对象的读取方法，就能读取文件了。

FileReader共有4种读取方法：  
**1.readAsArrayBuffer(file)：**将文件读取为ArrayBuffer。  
**2.readAsBinaryString(file)：**将文件读取为二进制字符串  
**3.readAsDataURL(file)：**将文件读取为Data URL  
**4.readAsText(file, [encoding])：**将文件读取为文本，encoding缺省值为'UTF-8'  
此外，abort()方法可以停止读取文件。

FileReader对象在读取文件后，还需要进行处理。为了不阻塞当前线程，API采用了事件模型，可以注册这些事件：  
1.onabort：中断时触发  
2.onerror：出错时触发  
3.onload：文件成功读取完毕时触发  
4.onloadend：文件读取完毕时触发，无论是否失败  
5.onloadstart：文件开始读取时触发  
6.onprogress：当文件读取时，周期性地触发

## 状态常量readyState

|  |  |  |
| --- | --- | --- |
| **常量名** | **值** | **描述** |
| EMPTY | 0 | 还没有加载任何数据. |
| LOADING | 1 | 数据正在被加载. |
| DONE | 2 | 已完成全部的读取请求. |

## 属性

|  |  |  |
| --- | --- | --- |
| **属性名** | **类型** | **描述** |
| error | [DOMError](https://developer.mozilla.org/zh-CN/docs/Web/API/DOMError) | 在读取文件时发生的错误. **只读.** |
| readyState | [unsigned short](https://developer.mozilla.org/zh-CN/unsigned_short) | 表明FileReader对象的当前状态. 值为[State constants](https://developer.mozilla.org/zh-CN/docs/Web/API/FileReader#State_constants)中的一个. **只读** |
| result | [jsval](https://developer.mozilla.org/zh-CN/docs/Mozilla/Projects/SpiderMonkey/JSAPI_reference/jsval) | 读取到的文件内容.这个属性只在读取操作完成之后才有效,并且数据的格式取决于读取操作是由哪个方法发起的. **只读.** |

有了这些方法以后，就可以处理文件了。  
先来试试读取文本文件：

复制代码 代码如下:

function handleFiles(files) {  
    if (files.length) {  
        var file = files[0];  
        var reader = new FileReader();  
        if (/text\/\w+/.test(file.type)) {  
            reader.onload = function() {  
                $('<pre>' + this.result + '</pre>').appendTo('body');  
            }  
            reader.readAsText(file);  
        }  
    }  
}

这里的this.result实际上就是reader.result，也就是读取出来的文件内容。  
测试一下你会发现这个文件的内容被添加到网页中了。如果是用Chrome的话，必须把网页放在服务器上或插件里，file协议下会失败。

再来试试图片，因为浏览器可以直接显示Data URI协议的图片，所以这次就添加图片：

复制代码 代码如下:

function handleFiles(files) {  
    if (files.length) {  
        var file = files[0];  
        var reader = new FileReader();  
        if (/text\/\w+/.test(file.type)) {  
            reader.onload = function() {  
                $('<pre>' + this.result + '</pre>').appendTo('body');  
            }  
            reader.readAsText(file);  
        } else if(/image\/\w+/.test(file.type)) {  
            reader.onload = function() {  
                $('<img src="' + this.result + '"/>').appendTo('body');  
            }  
            reader.readAsDataURL(file);  
        }  
    }  
}

其实input:file控件还支持选择多个文件：

复制代码 代码如下:

<input type="file" id="files" multiple="" onchange="handleFiles(this.files)"/>

这样handleFiles()里就需要遍历处理files了。

如果只想读取部分数据的话，File对象还有webkitSlice()或mozSlice()方法，用于生成Blob对象。这个对象可以和File对象一样被FileReader读取。这2个方法接收3个参数：第1个参数是起始位置；第2个是结束位置，省略时则读到文件结尾；第3个是content type。  
例子可以参考《Reading local files in JavaScript》。

当然，除了导入数据和显示文件以外，它还可以用来做AJAX上传，代码可以参考《Using files from web applications》。

接下来说保存文件。  
实际上File API: Writer提供了4个接口，但目前只有部分浏览器（Chrome 8+和Firefox 4+）实现了BlobBuilder，其余接口都不可用。  
对于不支持的浏览器，可以使用BlobBuilder.js和FileSaver.js来获得支持。  
我研究了一下，发现了其中的奥秘。

BlobBuilder可以创建一个Blob对象。把这个Blob对象传递给URL.createObjectURL()方法，就可以拿到一个object URL。而这个object URL就是这个Blob对象的下载地址。  
拿到下载地址后，创建一个a元素，将下载地址赋值给href属性，文件名赋值给download属性（Chrome 14+支持）。  
然后再创建一个click事件，交给这个a元素处理，就会导致浏览器开始下载这个Blob对象了。  
最后，用URL.revokeObjectURL()来释放这个object URL，通知浏览器可以不必继续引用这个文件了。

下面就是一段化简的代码：

复制代码 代码如下:

var BlobBuilder = BlobBuilder || WebKitBlobBuilder || MozBlobBuilder;  
var URL = URL || webkitURL || window;

function saveAs(blob, filename) {  
    var type = blob.type;  
    var force\_saveable\_type = 'application/octet-stream';  
    if (type && type != force\_saveable\_type) { // 强制下载，而非在浏览器中打开  
        var slice = blob.slice || blob.webkitSlice || blob.mozSlice;  
        blob = slice.call(blob, 0, blob.size, force\_saveable\_type);  
    }

    var url = URL.createObjectURL(blob);  
    var save\_link = document.createElementNS('http://www.w3.org/1999/xhtml', 'a');  
    save\_link.href = url;  
    save\_link.download = filename;

    var event = document.createEvent('MouseEvents');  
    event.initMouseEvent('click', true, false, window, 0, 0, 0, 0, 0, false, false, false, false, 0, null);  
    save\_link.dispatchEvent(event);  
    URL.revokeObjectURL(url);  
}

var bb = new BlobBuilder;  
bb.append('Hello, world!');  
saveAs(bb.getBlob('text/plain;charset=utf-8'), 'hello world.txt');

# jQuery获取页面及个元素高度、宽度

 (2013-09-04 15:44:30)

[[http://simg.sinajs.cn/blog7style/images/common/sg_trans.gif](javascript:;)转载▼](javascript:;)

|  |  |
| --- | --- |
| 标签： [jquery](http://search.sina.com.cn/?c=blog&q=jquery&by=tag)  [html](http://search.sina.com.cn/?c=blog&q=html&by=tag)  [css](http://search.sina.com.cn/?c=blog&q=css&by=tag)  [屏幕宽高](http://search.sina.com.cn/?c=blog&q=%C6%C1%C4%BB%BF%ED%B8%DF&by=tag)  [可视区域宽高](http://search.sina.com.cn/?c=blog&q=%BF%C9%CA%D3%C7%F8%D3%F2%BF%ED%B8%DF&by=tag) | 分类： [jquery](http://blog.sina.com.cn/s/articlelist_1359252903_1_1.html) |

获取浏览器显示区域（可视区域）的高度 ：

$(window).height();

获取浏览器显示区域（可视区域）的宽度 ：

$(window).width();

获取页面的文档高度

$(document).height();

获取页面的文档宽度 ：

$(document).width();

浏览器当前窗口文档body的高度：

$(document.body).height();

浏览器当前窗口文档body的宽度：

$(document.body).width();

获取滚动条到顶部的垂直高度 (即网页被卷上去的高度)

$(document).scrollTop();

获取滚动条到左边的垂直宽度 ：

$(document).scrollLeft();

获取或设置元素的宽度：

$(obj).width();

获取或设置元素的高度：

$(obj).height();

某个元素的上边界到body最顶部的距离：obj.offset().top;（在元素的包含元素不含滚动条的情况下）

某个元素的左边界到body最左边的距离：obj.offset().left;（在元素的包含元素不含滚动条的情况下）

返回当前元素的上边界到它的包含元素的上边界的偏移量：obj.offset().top（在元素的包含元素含滚动条的情况下）

返回当前元素的左边界到它的包含元素的左边界的偏移量：obj.offset().left（在元素的包含元素含滚动条的情况下）

**scrollTop(), scrollLeft()**

设置或返回已经滚动到元素的左边界或上边界的像素数。简单地说，就是设置或者获取匹配元素相对滚动条上侧或者左侧的偏移。只有在元素有滚动条的时候，例如，元素的 CSS overflow 属性设置为 auto 的时候，这些像素才有用。这些属性也只在文档的 <body> 或 <html> 标记上定义（这和浏览器有关），并且一起来制定滚动文档的位置。注意，这些属性并不会指定一个 <iframe> 标记的滚动量。这是非标准的但却得到很好支持的属性

**pageX/pageY:**  
鼠标相对于整个页面的X/Y坐标。  
注意，整个页面的意思就是你整个网页的全部，比如说网页很宽很长，宽2000px，高3000px，那pageX,pageY的最大值就是它们了。  
特别说明：IE不支持！  
**clientX/clientY：**  
事件发生时鼠标在浏览器内容区域的X/Y坐标（不包含滚动条）。  
浏览器内容区域即浏览器窗口中用来显示网页的可视区域，注意这个可视，也就是说需要拖动滚动条才能看到的区域不算。  
当你将浏览器窗口缩小时，clientX/clientY的最大值也会缩小，但始终，它们的最大值不会超过你浏览器可视区域。  
特别说明：IE下此属性不规范，它们的最小值不是0而是2，也就是说IE下的clientX/clientY比火狐下始终大2px。  
**screenX/screenY**  
鼠标在屏幕上的坐标。screenX,screenY的最大值不会超过屏幕分辨率。  
**offsetX/offsetY:**  
得出的结果跟pageX/pageY一样，既然如此，它有什么存在价值？因为：  
特别说明：只有IE支持！相当于IE下的pageX,pageY。  
  
e.clientX + document.documentElement.scrollLeft - document.documentElement.clientLeft = e.pageX  
e.clientY + document.documentElement.scrollTop  - document.documentElement.clientTop = e.pageY

# [[HTML5] Blob对象](http://www.cnblogs.com/hhhyaaon/p/5928152.html)

# 写在前面

本篇主要总结Blob对象属性及作用，通过DEMO介绍Blob对象的应用场景。

一直以来，JS都没有比较好的可以直接处理二进制的方法。而Blob的存在，允许我们可以通过JS直接操作二进制数据。

一个Blob对象就是一个包含有只读原始数据的类文件对象。Blob对象中的数据并不一定得是JavaScript中的原生形式。File接口基于Blob，继承了Blob的功能,并且扩展支持了用户计算机上的本地文件。

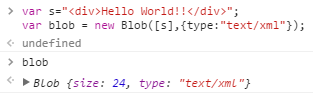
Blob对象可以看做是存放二进制数据的容器，此外还可以通过Blob设置二进制数据的MIME类型。

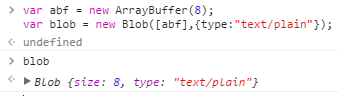
## 创建Blob

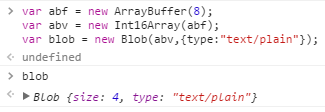
### 通过构造函数

var blob = new Blob(dataArr:Array<any>, opt:{type:string});

* dataArray：数组，包含了要添加到Blob对象中的数据，数据可以是任意多个ArrayBuffer，ArrayBufferView， Blob，或者 DOMString对象。
* opt：对象，用于设置Blob对象的属性（如：MIME类型）

1、创建一个装填DOMString对象的Blob对象  


2、创建一个装填ArrayBuffer对象的Blob对象  


3、创建一个装填ArrayBufferView对象的Blob对象（ArrayBufferView可基于ArrayBuffer创建，返回值是一个类数组。如下：创建一个8字节的ArrayBuffer，在其上创建一个每个数组元素为2字节的“视图”）  


### 通过Blob.slice()

此方法返回一个新的Blob对象，包含了原Blob对象中指定范围内的数据

Blob.slice(start:number, end:number, contentType:string)

* start：开始索引，默认为0
* end：截取结束索引（不包括end）
* contentType：新Blob的MIME类型，默认为空字符串



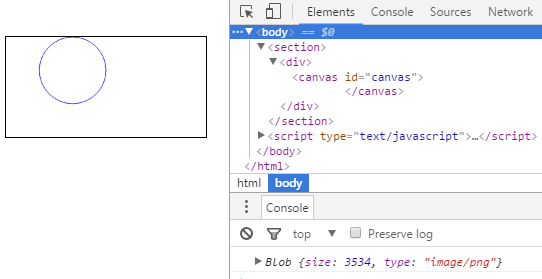
### 通过canvas.toBlob()

var canvas = document.getElementById("canvas");

canvas.toBlob(function(blob){

console.log(blob);

});



# 应用场景

前面提到，**File接口基于Blob**，继承了Blob的功能并进行了扩展，故我们可以像使用Blob一样使用File对象。Enen

## 分片上传

通过Blob.slice方法，可以将大文件分片，轮循向后台提交各文件片段，即可实现文件的分片上传。  
分片上传逻辑如下：

* 获取要上传文件的File对象，根据chunk（每片大小）对文件进行分片
* 通过post方法轮循上传每片文件，其中url中拼接querystring用于描述当前上传的文件信息；post body中存放本次要上传的二进制数据片段
* 接口每次返回offset，用于执行下次上传

下面是分片上传的简单实现：

initUpload();

//初始化上传

function initUpload() {

var chunk = 100 \* 1024; //每片大小

var input = document.getElementById("file"); //input file

input.onchange = function (e) {

var file = this.files[0];

var query = {};

var chunks = [];

if (!!file) {

var start = 0;

//文件分片

for (var i = 0; i < Math.ceil(file.size / chunk); i++) {

var end = start + chunk;

chunks[i] = file.slice(start , end);

start = end;

}

// 采用post方法上传文件

// url query上拼接以下参数，用于记录上传偏移

// post body中存放本次要上传的二进制数据

query = {

fileSize: file.size,

dataSize: chunk,

nextOffset: 0

}

upload(chunks, query, successPerUpload);

}

}

}

// 执行上传

function upload(chunks, query, cb) {

var queryStr = Object.getOwnPropertyNames(query).map(key => {

return key + "=" + query[key];

}).join("&");

var xhr = new XMLHttpRequest();

xhr.open("POST", "http://xxxx/opload?" + queryStr);

xhr.overrideMimeType("application/octet-stream");

//获取post body中二进制数据

var index = Math.floor(query.nextOffset / query.dataSize);

getFileBinary(chunks[index], function (binary) {

if (xhr.sendAsBinary) {

xhr.sendAsBinary(binary);

} else {

xhr.send(binary);

}

});

xhr.onreadystatechange = function (e) {

if (xhr.readyState === 4) {

if (xhr.status === 200) {

var resp = JSON.parse(xhr.responseText);

// 接口返回nextoffset

// resp = {

// isFinish:false,

// offset:100\*1024

// }

if (typeof cb === "function") {

cb.call(this, resp, chunks, query)

}

}

}

}

}

// 每片上传成功后执行

function successPerUpload(resp, chunks, query) {

if (resp.isFinish === true) {

alert("上传成功");

} else {

//未上传完毕

query.offset = resp.offset;

upload(chunks, query, successPerUpload);

}

}

// 获取文件二进制数据

function getFileBinary(file, cb) {

var reader = new FileReader();

reader.readAsArrayBuffer(file);

reader.onload = function (e) {

if (typeof cb === "function") {

cb.call(this, this.result);

}

}

}

以上是文件分片上传前端的简单实现，当然，此功能还可以更加完善，如后台需要对合并后的文件大小进行校验；或者前端加密文件，全部上传完毕后后端解密校验等，此处不做赘述。

## 通过url下载文件

window.URL对象可以为Blob对象生成一个网络地址，结合a标签的download属性，可以实现点击url下载文件  
实现如下：

createDownload("download.txt","download file");

function createDownload(fileName, content){

var blob = new Blob([content]);

var link = document.createElement("a");

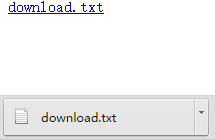
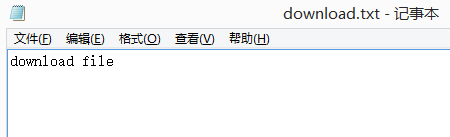
link.innerHTML = fileName;

link.download = fileName;

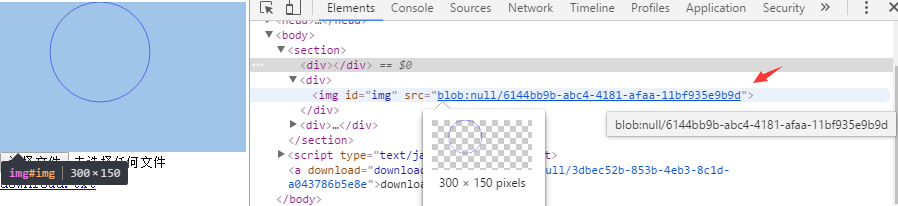
link.href = URL.createObjectURL(blob);

document.getElementsByTagName("body")[0].appendChild(link);

}

执行后页面上会生成此Blob对象的地址，点击后可下载：  
  
查看下载结果：  


## 通过url显示图片

我们知道，img的src属性及background的url属性，都可以通过接收图片的网络地址或base64来显示图片，同样的，我们也可以把图片转化为Blob对象，生成URL（URL.createObjectURL(blob)），来显示图片。  


# 小结

本文主要介绍了Blob对象的属性和使用场景，其实我们可以看到，Blob对象作为一个装填二进制数据的基本对象，其作用也仅仅是一个容器，而真正的业务功能则需要通过FileReader、URL、Canvas等对象实现，之后会继续对这些对象的功能和应用场景进行归纳。

[**URL.createObjectURL()**](https://developer.mozilla.org/zh-CN/docs/Web/API/URL/createObjectURL)

返回一个[DOMString](https://developer.mozilla.org/zh-CN/docs/Web/API/DOMString) ，包含一个唯一的blob链接（该链接协议为以blob:，后跟唯一标识浏览器中的对象的掩码）。

[**URL.revokeObjectURL()**](https://developer.mozilla.org/zh-CN/docs/Web/API/URL/revokeObjectURL)

销毁之前使用[URL.createObjectURL()](https://developer.mozilla.org/zh-CN/docs/Web/API/URL/createObjectURL)方法创建的URL实例。

# [fetch.js](http://www.cnblogs.com/mina-huojian66/p/6519699.html)

与XMLHttpRequest(XHR)类似，fetch()方法允许你发出AJAX请求。区别在于Fetch API使用Promise，因此是一种简洁明了的API，比XMLHttpRequest更加简单易用。

### XMLHttpRequest

一个 **XMLHttpRequest** 请求需要两个监听器来捕捉 success 和 error 两种情形，而且需要调用 **open()**和 **send()** 方法

function reqListener() {

var data = JSON.parse(this.responseText);

console.log(data);

}

function reqError(err) {

console.log('Fetch Error :-S', err);

}

var oReq = new XMLHttpRequest();

oReq.onload = reqListener; //或是使用onreadystatechange监听

oReq.onerror = reqError;

oReq.open('get', './api/some.json', true);

oReq.send();

不用new

fetch("../students.json").then(function(response){

if(response.status!==200){

console.log("存在一个问题，状态码为："+response.status);

return;

}

//检查响应文本

response.json().then(function(data){

console.log(data);

});

}).catch(function(err){

console.log("Fetch错误:"+err);

})

[复制代码](javascript:void(0);)

mode属性用来决定是否允许跨域请求，以及哪些response属性可读。可选的mode属性值为 same-origin，no-cors（默认）以及 cores;

* same-origin模式很简单，如果一个请求是跨域的，那么返回一个简单的error，这样确保所有的请求遵守同源策略
* no-cors模式允许来自CDN的脚本、其他域的图片和其他一些跨域资源，但是首先有个前提条件，就是请求的method只能是"HEAD","GET"或者"POST"
* cors模式我们通常用作跨域请求来从第三方提供的API获取数据

## 响应的对象Response类型

当我们执行一个fetch请求时，响应的数据的类型**response.type**可以是“[basic](https://fetch.spec.whatwg.org/#concept-filtered-response-basic)”, “[cors](https://fetch.spec.whatwg.org/#concept-filtered-response-cors)” 或 “[opaque](https://fetch.spec.whatwg.org/#concept-filtered-response-opaque)”。这些类型用来说明应该如何对待这些数据和数据的来源。

Response 也有一个type属性，它的值可能是"basic","cors","default","error"或者"opaque";

* "basic": 正常的，同域的请求，包含所有的headers除了"Set-Cookie"和"Set-Cookie2"。
* "cors": Response从一个合法的跨域请求获得， 一部分header和body 可读。(限定只能在响应头中看见“Cache-Control”、“Content-Language”、“Content-Type”、“Expires”、“Last-Modified”以及“Progma”)
* "error": 网络错误。Response的status是0，Headers是空的并且不可写。(当Response是从Response.error()中得到时，就是这种类型)
* "opaque": Response从"no-cors"请求了跨域资源。依靠Server端来做限制。(将不能查看数据，也不能查看响应状态，也就是说我们不能检查请求成功与否;目前为止不能在页面脚本中请求其他域中的资源)

[复制代码](javascript:void(0);)

function status(response){

if(response.status>=200 && response.status<300){

return Promise.resolve(response);

}else{

return Promise.reject(new Error(response.statusText));

}

}

function json(response){

return response.json();

}

fetch("../students.json"**,{mode:"cors"}**)//响应类型“cors”,一般为“basic”;是指对应的response.type

.then(status)//可以链接方法  
.then(json)   
.then(function(data){   
console.log("请求成功，JSON解析后的响应数据为:",data); })  
.then(function(response){   
console.log(**response.headers.get('Content-Type'**)); //application/json   
console.log(response.headers.get('Date')); //Wed, 08 Mar 2017 06:41:44 GMT   
console.log(response.status); //200   
console.log(response.statusText); //ok   
console.log(**response.type**); //cors   
console.log(response.url); //http://.../students.json })  
.catch(function(err){   
console.log("Fetch错误:"+err);   
})

[复制代码](javascript:void(0);)

**使用POST方法**提交页面中的一些数据:将method属性值设置为post，并且在body属性值中设置需要提交的数据;

## **credentials属性决定了cookies**

是否能跨域得到 : "omit"(默认),"same-origin"以及"include";

[复制代码](javascript:void(0);)

var url='...';

fetch(url,{

method:"post",//or 'GET'

**credentials**: "same-origin",//or "include","same-origin":只在请求同域中资源时成功，其他请求将被拒绝。  
headers:{  
"Content-type":"application:/x-www-form-urlencoded;charset=UTF-8"  
},  
body:"name=lulingniu&age=40"  
})

.then(status)

.then(json) //JSON进行解析来简化代码

.then(function(data){

console.log("请求成功，JSON解析后的响应数据为:",data);

})

.catch(function(err){

console.log("Fetch错误:"+err);

});

[复制代码](javascript:void(0);)

## 兼容浏览器的处理

可以通过下面的语句处理浏览器兼容的问题。

{% codeblock lang:js%}  
**if(self.fetch) {**  
// 使用 fetch 框架处理  
} else {  
// 使用 XMLHttpRequest 或者其他封装框架处理  
}  
{% endcodeblock %}

## 3.2 一般构造请求的方法

使用 fetch 的构造函数请求数据后，返回一个 [Promise](https://www.promisejs.org/) 对象，处理即可。  
{% codeblock lang:js%}  
fetch("[http://blog.parryqiu.com](http://blog.parryqiu.com/)")  
.then(function(response){  
// do something...  
})  
{% endcodeblock %}

## 3.3 fetch 构成函数的其他选项

我们可以将与 HTTP Headers 兼容的格式加入到请求的头中，如每次 API 的请求我们想不受缓存的影响，那么可以像下面这样请求：

{% codeblock lang:js%}  
fetch("[http://blog.parryqiu.com](http://blog.parryqiu.com/)", {  
headers: {  
'Cache-Control': 'no-cache'  
}  
})  
.then(function(response){  
// do something...  
})  
{% endcodeblock %}

具体的可选参数可以查看 [这里](https://fetch.spec.whatwg.org/#concept-request-initiator)。  
如我们还可以这样使用，添加了更多的头部参数：  
{% codeblock lang:js%}  
**var myHeaders = new Headers();**  
myHeaders.append("Content-Type", "text/plain");  
myHeaders.append("Content-Length", content.length.toString());  
myHeaders.append("X-Custom-Header", "ProcessThisImmediately");

var myInit = {  
method: 'GET',  
headers: myHeaders,  
mode: 'cors',  
cache: 'default'  
};

fetch("[http://blog.parryqiu.com](http://blog.parryqiu.com/)", myInit)  
.then(function(response){  
// do something...  
})  
{% endcodeblock %}

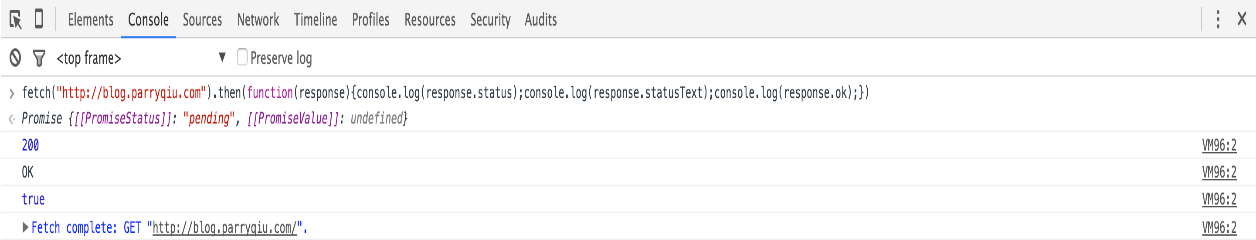
## 3.4 返回的数据结构

在请求后的 Response 中，具体的定义在 [这里](https://fetch.spec.whatwg.org/#dom-response)。  
常用的有：

* Response.status 也就是 StatusCode，如成功就是 200；
* Response.statusText 是 StatusCode 的描述文本，如成功就是 OK；
* Response.ok 一个 Boolean 类型的值，判断是否正常返回，也就是 StatusCode 为 200-299。

做如下请求：  
{% codeblock lang:js%}  
fetch("[http://blog.parryqiu.com](http://blog.parryqiu.com/)")  
.then(function(response){  
console.log(response.status);  
console.log(response.statusText);  
console.log(response.ok);  
})  
{% endcodeblock %}

返回的数据：



## 3.5 Body 参数

因为在 Request 和 Response 中都包含 Body 的实现，所以包含以下类型：

* ArrayBuffer
* ArrayBufferView (Uint8Array and friends)
* Blob/File
* string
* **URLSearchParams**
* FormData

在 fetch 中实现了对应的方法，并返回的都是 [Promise](https://www.promisejs.org/) 类型。

* [arrayBuffer()](https://developer.mozilla.org/en-US/docs/Web/API/Body/arrayBuffer)
* [blob()](https://developer.mozilla.org/en-US/docs/Web/API/Body/blob)
* [json()](https://developer.mozilla.org/en-US/docs/Web/API/Body/json)
* [text()](https://developer.mozilla.org/en-US/docs/Web/API/Body/text)
* [formData()](https://developer.mozilla.org/en-US/docs/Web/API/Body/formData)

这样处理返回的数据类型就会变的特别地方便，如处理 json 格式的数据：  
{% codeblock lang:js%}  
var myRequest = new Request('<http://api.com/products.json>');

fetch(myRequest).then(function(response) {  
return **response.json()**.then(function(json) {  
for(i = 0; i < json.products.length; i++) {  
var name = json.products[i].Name;  
var price = json.products[i].Price;  
// do something more...  
}  
});  
});  
{% endcodeblock %}

## 串联 Promises

Promises最大的一个特征是，你可以串联各种操作。对于fetch来说，我们可以在各个fetch操作里共享一些逻辑操作。

在使用JSON API时，我们需要检查每次请求响应的状态，然后解析成JSON对象。使用promise，我们可以简单的将分析状态和解析JSON的代码放到一个单独函数里，然后当做promise返回，这样就是代码更条理了。

function status(response) {

if (response.status >= 200 && response.status < 300) {

return Promise.resolve(response)

} else {

return Promise.reject(new Error(response.statusText))

}

}

function json(response) {

return response.json()

}

fetch('users.json')

.then(status)

.then(json)

.then(function(data) {

console.log('Request succeeded with JSON response', data);

}).catch(function(error) {

console.log('Request failed', error);

});

我们用 **status** 函数来检查 **response.status** 并返回 Promise.resolve() 或 Promise.reject() 的结果，这个结果也是一个 Promise。我们的**fetch()** 调用链条中，首先如果**fetch()**执行结果是 resolve，那么，接着会调用 **json()** 方法，这个方法返回的也是一个 Promise，这样我们就得到一个分析后的JSON对象。如果分析失败，将会执行reject函数和catch语句。

你会发现，在fetch请求中，我们可以共享一些业务逻辑，使得代码易于维护，可读性、可测试性更高。

## 用fetch执行表单数据提交

在WEB应用中，提交表单是非常常见的操作，用fetch来提交表单数据也是非常简洁。

fetch里提供了 **method** 和 **body** 参数选项。

fetch(url, {

method: 'post',

headers: {

"Content-type": "application/x-www-form-urlencoded; charset=UTF-8"

},

body: 'foo=bar&lorem=ipsum'

})

.then(json)

.then(function (data) {

console.log('Request succeeded with JSON response', data);

})

.catch(function (error) {

console.log('Request failed', error);

});

## 在Fetch请求里发送用户身份凭证信息

如果你想在fetch请求里附带cookies之类的凭证信息，可以将 **credentials** 参数设置成 **“include”** 值。

fetch(url, {

credentials: 'include'

})

显而易见，fetch API相比起传统的 XMLHttpRequest (XHR) 要简单的多，相比起jQuery里提供ajax API也丝毫不逊色。

新的 [Fetch](https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API) API打算修正上面提到的那些缺陷。 它向JS中引入和HTTP协议中同样的原语（即Fetch——译者注）。具体而言，它引入一个实用的函数fetch()用来简洁捕捉从网络上检索一个资源的意图。

[Fetch 规范](https://fetch.spec.whatwg.org/)的API明确了用户代理获取资源的语义。它结合ServiceWorkers，尝试达到以下优化：

1. 改善离线体验
2. 保持可扩展性

到写这篇文章的时候，Fetch API被Firefox 39（Nightly版）以及Chrome 42（开发版）支持。在github上，有基于低版本浏览器的[兼容实现](https://github.com/github/fetch)

**特性检测**

要检查是否支持Fetch API，可以通过检查 Headers, Request, Response 或者 fetch 在 window 或者 worker 作用域中是否存在。

**简单的fetching示例**

在Fetch API中，最常用的就是fetch()函数。它接收一个URL参数，返回一个promise来处理response。response参数带着一个Response对象。

fetch("/data.json").then(function(res) {

// **res instanceof Response == true.//说明有Response对象**

if (**res.ok**) {

res.json().then(function(data) {

console.log(data.entries);

});

} else {

console.log("Looks like the response wasn't perfect, got status", res.status);

}

}, function(e) {

console.log("Fetch failed!", e);

});

如果是提交一个POST请求，代码如下：

fetch("http://www.example.org/submit.php", {

method: "POST",

headers: {

"Content-Type": "application/x-www-form-urlencoded"

},

body: "firstName=Nikhil&favColor=blue&password=easytoguess"

}).then(function(res) {

if (res.ok) {

alert("Perfect! Your settings are saved.");

} else if (res.status == 401) {

alert("Oops! You are not authorized.");

}

}, function(e) {

alert("Error submitting form!");

});

fetch()函数的参数和传给Request()构造函数的参数保持完全一致，所以你可以直接传任意复杂的request请求给fetch()。

## Headers

Fetch引入了3个接口，它们分别是 Headers,Request 以及 Response 。他们直接对应了相应的HTTP概念，但是基于安全考虑，有些区别，例如支持CORS规则以及保证cookies不能被第三方获取。

Headers接口是一个简单的多映射的名-值表

var content = "Hello World";

var reqHeaders = new Headers();

reqHeaders.append("Content-Type", "text/plain");

reqHeaders.append("Content-Length", content.length.toString());

reqHeaders.append("X-Custom-Header", "ProcessThisImmediately");

也可以传一个多维数组或者json：

reqHeaders = new Headers({

"Content-Type": "text/plain",

"Content-Length": content.length.toString(),

"X-Custom-Header": "ProcessThisImmediately",

});

Headers的内容可以被检索：

console.log(reqHeaders.has("Content-Type")); // true

console.log(reqHeaders.has("Set-Cookie")); // false

reqHeaders.set("Content-Type", "text/html");

reqHeaders.append("X-Custom-Header", "AnotherValue");

console.log(reqHeaders.get("Content-Length")); // 11

console.log(reqHeaders.getAll("X-Custom-Header")); // ["ProcessThisImmediately", "AnotherValue"]

reqHeaders.delete("X-Custom-Header");

console.log(reqHeaders.getAll("X-Custom-Header")); // []

一些操作不仅仅对ServiceWorkers有用，本身也提供了更方便的操作Headers的API（相对于XMLHttpRequest来说——译者注）。

由于Headers可以在request请求中被发送或者在response请求中被接收，并且规定了哪些参数是可写的，**Headers对象有一个特殊的guard属性。**这个属性没有暴露给Web，但是它影响到哪些内容可以在Headers对象中被改变。

可能的值如下：

* “none”: 默认的
* “request”: 从Request中获得的Headers只读。
* “request-no-cors”：从不同域的Request中获得的Headers只读。
* “response”: 从Response获得的Headers只读。
* “immutable” 在ServiceWorkers中最常用的，所有的Headers都只读。

哪一种 guard 作用于 Headers 导致什么行为，详细定义在了这个[规范](https://fetch.spec.whatwg.org/)中。例如，你不可以添加或者修改一个guard属性是”request”的Request Headers的”Content-Length”属性。同样地，插入”Set-Cookie”属性到一个Response headers是不允许的，因此ServiceWorkers是不能给合成的Response的headers添加一些cookies。

如果使用了一个不合法的HTTP Header属性名，那么Headers的方法通常都抛出 TypeError 异常。如果不小心写入了一个不可写的属性，也会抛出一个 TypeError 异常。除此以外的情况，失败了并不抛出异常。例如：

var res = Response.error();

try {

res.headers.set("Origin", "http://mybank.com");

} catch(e) {

console.log("Cannot pretend to be a bank!");

}

## Request

Request接口定义了通过HTTP请求资源的request格式。参数需要URL、method和headers，同时Request也接受一个特定的body，mode，credentials以及cache hints.

最简单的 Request 当然是一个URL，可以通过URL来GET一个资源。

var req = new Request("/index.html");

console.log(req.method); // "GET"

console.log(req.url); // "http://example.com/index.html"

你也可以将一个建好的Request对象传给构造函数，这样将复制出一个新的Request。

var copy = new Request(req);

console.log(copy.method); // "GET"

console.log(copy.url); // "http://example.com/index.html"

这种用法通常见于ServiceWorkers。

URL以外的其他属性的初始值能够通过第二个参数传给Request构造函数。这个参数是一个json：

var uploadReq = new Request("/uploadImage", {

method: "POST",

headers: {

"Content-Type": "image/png",

},

body: "image data"

});

mode属性用来决定是否允许跨域请求，以及哪些response属性可读。可选的mode属性值为same-origin，no-cors（默认）以及cors。

same-origin模式很简单，如果一个请求是跨域的，那么返回一个简单的error，这样确保所有的请求遵守同源策略。

var arbitraryUrl = document.getElementById("url-input").value;

fetch(arbitraryUrl, { mode: "same-origin" }).then(function(res) {

console.log("Response succeeded?", res.ok);

}, function(e) {

console.log("Please enter a same-origin URL!");

});

no-cors模式允许来自CDN的脚本、其他域的图片和其他一些跨域资源，但是首先有个前提条件，就是请求的method只能是”HEAD”,”GET”或者”POST”。此外，任何 ServiceWorkers 拦截了这些请求，它不能随意添加或者改写任何headers，除了[这些](https://fetch.spec.whatwg.org/#simple-header)。第三，JavaScript不能访问Response中的任何属性，这保证了 ServiceWorkers 不会导致任何跨域下的安全问题而隐私信息泄漏。

cors模式我们通常用作跨域请求来从第三方提供的API获取数据。这个模式遵守[CORS协议](https://developer.mozilla.org/en-US/docs/Web/HTTP/Access_control_CORS)。只有有限的[一些](https://fetch.spec.whatwg.org/#concept-filtered-response-cors)headers被暴露给Response对象，但是body是可读的。例如，你可以获得一个Flickr的[最感兴趣的照片](https://www.flickr.com/services/api/flickr.interestingness.getList.html)的清单：

var u = new URLSearchParams();

u.append('method', 'flickr.interestingness.getList');

u.append('api\_key', '<insert api key here>');

u.append('format', 'json');

u.append('nojsoncallback', '1');

var apiCall = fetch('https://api.flickr.com/services/rest?' + u);

apiCall.then(function(response) {

return response.json().then(function(json) {

// photo is a list of photos.

return json.photos.photo;

});

}).then(function(photos) {

photos.forEach(function(photo) {

console.log(photo.title);

});

});

你无法从Headers中读取”Date”属性，因为Flickr在Access-Control-Expose-Headers中设置了不允许读取它。

response.headers.get("Date"); // null

credentials枚举属性决定了cookies是否能跨域得到。这个属性与XHR的withCredentials标志相同，但是只有三个值，分别是”omit”（默认）,”same-origin”以及”include”。

Request对象也可以提供 caching hints 给用户代理。这个属性还在安全复审阶段。Firefox 提供了这个属性，但是它目前还不起作用。

Request还有两个只读的属性与ServiceWorks拦截有关。其中一个是referrer，表示Request的来源，可能为空。另外一个是context，是一个非常大的枚举集合定义了获得的资源的种类，它可能是image比如请求来自于img标签，可能是worker如果是一个worker脚本，等等。如果使用fetch()函数，这个值是fetch。

## Response

Response实例通常在fetch()的回调中获得。但是它们也可以用JS构造，不过通常这招只用于ServiceWorkers。

Response中最常见的成员是status（一个整数默认值是200）和statusText（默认值是”OK”），对应HTTP请求的status和reason。还有一个”ok”属性，当status为2xx的时候它是true。

headers 属性是Response的Headers对象，它是只读的(with guard “response”)，url属性是当前Response的来源URL。

Response 也有一个type属性，它的值可能是”basic”,”cors”,”default”,”error”或者”opaque。

* “basic”: 正常的，同域的请求，包含所有的headers除开”Set-Cookie”和”Set-Cookie2″。
* “cors”: Response从一个合法的跨域请求获得，[一部分header和body](https://fetch.spec.whatwg.org/#concept-filtered-response-cors)可读。
* “error”: 网络错误。Response的status是0，Headers是空的并且不可写。当Response是从Response.error()中得到时，就是这种类型。
* “opaque”: Response从”no-cors”请求了跨域资源。依靠Server端来做限制。

“error”类型会导致fetch()函数的Promise被reject并回调出一个TypeError。

还有一些属性只在ServerWorker作用域下有效。以正确的方式 返回一个Response针对一个被ServiceWorkers拦截的Request，可以像下面这样写：

addEventListener('fetch', function(event) {

event.respondWith(new Response("Response body", {

headers: { "Content-Type" : "text/plain" }

});

});

如你所见，Response有个接收两个可选参数的构造器。第一个参数是返回的body，第二个参数是一个json，设置status、statusText以及headers。

静态方法Response.error()简单返回一个错误的请求。类似的，Response.redirect(url, status)返回一个跳转URL的请求。

## 处理body

无论Request还是Response都可能带着body。由于body可以是各种类型，比较复杂，所以前面我们故意先略过它，在这里单独拿出来讲解。

body可以是以下任何一种类型的实例：

* [ArrayBuffer](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/ArrayBuffer)
* [ArrayBufferView](https://developer.mozilla.org/en-US/docs/Web/API/ArrayBufferView)(Uint8Array and friends)
* [Blob](https://developer.mozilla.org/en-US/docs/Web/API/Blob)/[File](https://developer.mozilla.org/en-US/docs/Web/API/File)
* 字符串
* [URLSearchParams](https://url.spec.whatwg.org/#interface-urlsearchparams)
* [FormData](https://developer.mozilla.org/en-US/docs/Web/API/FormData)——目前不被Gecko和Blink支持，Firefox预计在版本39和Fetch的其他部分一起推出。

此外，Request和Response都为他们的body提供了以下方法，这些方法都返回一个Promise对象。

* arrayBuffer()
* blob()
* json()
* text()
* formData()

在使用非文本的数据方面，Fetch API和XHR相比提供了极大的便利。

可以通过传body参数来设置Request的body：

var form = new FormData(document.getElementById('login-form'));

fetch("/login", {

method: "POST",

body: form

})

Response的第一个参数是body：

var res = new Response(new File(["chunk", "chunk"], "archive.zip",

{ type: "application/zip" }));

Request和Response（通过fetch()方法）都能够[自动识别自己的content type](https://fetch.spec.whatwg.org/#concept-bodyinit-extract)，Request还可以自动设置”Content-Type” header，如果开发者没有设置它的话。

**流和克隆**

非常重要的一点要说明，那就是Request和Response的body只能被读取一次！它们有一个属性叫bodyUsed，读取一次之后设置为true，就不能再读取了。

var res = new Response("one time use");

console.log(res.bodyUsed); // false

res.text().then(function(v) {

console.log(res.bodyUsed); // true

});

console.log(res.bodyUsed); // true

res.text().catch(function(e) {

console.log("Tried to read already consumed Response");

});

这样设计的目的是为了之后兼容基于流的API，让应用一次消费data，这样就允许了JavaScript处理大文件例如视频，并且可以支持实时压缩和编辑。

有时候，我们希望多次访问body，例如，你可能想用即将支持的[Cache API](http://slightlyoff.github.io/ServiceWorker/spec/service_worker/index.html#cache-objects)去缓存Request和Response，以便于可以离线使用，Cache要求body能被再次读取。

所以，我们该如何让body能经得起多次读取呢？API提供了一个clone()方法。调用这个方法可以得到一个克隆对象。不过要记得，clone()必须要在读取之前调用，也就是先clone()再读取。

addEventListener('fetch', function(evt) {

var sheep = new Response("Dolly");

console.log(sheep.bodyUsed); // false

var clone = sheep.clone();

console.log(clone.bodyUsed); // false

clone.text();

console.log(sheep.bodyUsed); // false

console.log(clone.bodyUsed); // true

evt.respondWith(cache.add(sheep.clone()).then(function(e) {

return sheep;

});

});

**未来的改进**

为了支持流，Fetch最终将提供可以中断执行读取资源的能力，并且提供可以得到读取进度的API。这些能力在XHR中有，但是想要实现成Promise-based的Fetch API有些麻烦。

# New formdata()

XMLHttpRequest 是一个浏览器接口，通过它，我们可以使得 Javascript 进行 HTTP (S) 通信。XMLHttpRequest 在现在浏览器中是一种常用的前后台交互数据的方式。2008年 2 月，XMLHttpRequest Level 2 草案提出来了，相对于上一代，它有一些新的特性，其中 FormData 就是 XMLHttpRequest Level 2 新增的一个对象，利用它来提交表单、模拟表单提交，当然最大的优势就是可以上传二进制文件。下面就具体介绍一下如何利用 FormData 来上传文件。

### ****FormData 上传文件实例****

首先看一下formData的基本用法：FormData对象，可以把所有表单元素的name与value组成一个queryString，提交到后台。只需要把 form 表单作为参数传入 FormData 构造函数即可：

var form = document.getElementById("form1");

var fd = new FormData(form);

这样就可以直接通过ajax 的 send() 方法将 fd 发送到后台。

以下创建了一个表单 form，表单中除了普通的数据外，还有文件上传，我们直接将 form对象作为参数传入FormData对象：

<form name="form1" id="form1">

<p>name:<input type="text" name="name" /></p>

<p>gender:<input type="radio" name="gender" value="1" />male <input type="radio" name="gender" value="2" />female</p>

<p>stu-number：<input type="text" name="number" /></p>

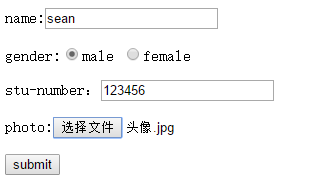
<p>photo:<input type="file" name="photo" id="photo"></p>

<p><input type="button" name="b1" value="submit" onclick="fsubmit()" /></p>

</form>

<div id="result"></div>

上述代码创建一个form，简单的填写一些信息，以及选择一张图片作为头像，设置一个div来存放返回的结果。



为了简便，我们还是采用jquery封装的ajax来向后台传输数据：

function fsubmit() {

var form=document.getElementById("form1");

var fd =new FormData(form);

$.ajax({

url: "server.php",

type: "POST",

data: fd,

processData: false, // 告诉jQuery不要去处理发送的数据

contentType: false, // 告诉jQuery不要去设置Content-Type请求头

success: function(response,status,xhr){

console.log(xhr);

var json=$.parseJSON(response);

var result = '';

result +="个人信息：<br/>name:"+json['name']+"<br/>gender:"+json['gender']+"<br/>number:"+json['number'];

result += '<br/>头像：<img src="' + json['photo'] + '" height="100" style="border-radius: 50%;" />';

$('#result').html(result);

}

});

return false;

}

* 1

上述代码中的 server.php 是服务器端的文件，接收ajax请求，并将接收结果返回，具体代码如下：

<?php

$name = isset($\_POST['name'])? $\_POST['name'] : '';

$gender = isset($\_POST['gender'])? $\_POST['gender'] : '';

$number = isset($\_POST['number'])? $\_POST['number'] : '';

$filename = time().substr($\_FILES['photo']['name'], strrpos($\_FILES['photo']['name'],'.'));

$response = array();

if(move\_uploaded\_file($\_FILES['photo']['tmp\_name'], $filename)){

$response['isSuccess'] = true;

$response['name'] = $name;

$response['gender'] = $gender;

$response['number'] = $number;

$response['photo'] = $filename;

}else{

$response['isSuccess'] = false;

}

echo json\_encode($response);

?>

填写好信息后，点击 submit，页面中能得到以下效果，去服务器端对应的文件夹下也能发现上传的图片。



如果你是原生 JavaScript 爱好者，当然一样能实现以上功能，下面是简单的JavaScript实现代码：

function fsubmit() {

var form=document.getElementById("form1");

var formData=new FormData(form);

alert(formData.name);

var oReq = new XMLHttpRequest();

oReq.onreadystatechange=function(){

if(oReq.readyState==4){

if(oReq.status==200){

console.log(typeof oReq.responseText);

var json=JSON.parse(oReq.responseText);

var result = '';

result +="个人信息：<br/>name:"+json['name']+"<br/>gender:"+json['gender']+"<br/>number:"+json['number'];

result += '<br/>头像：<img src="' + json['photo'] + '" height="50" style="border-radius: 50%;" />';

$('#result').html(result);

}

}

};

oReq.open("POST", "server.php");

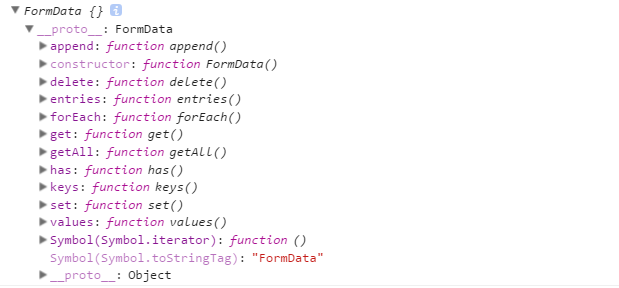
oReq.send(formData);

return false;

}

## ****FormData 对象方法介绍****

FormData 除了上面的创建新对象时直接将 form 作为参数传入外，还有其他的功能。网上大部分关于 FormData 介绍的文章都只提到了append()方法，那么FormData 对象到底有些什么方法呢？我们console 一下就知道：



console 之后我们有重大的发现，FormData 对象竟然有这么方法，所以还是自己测试才能发现真相，下面就对这些方法一一进行讲解：

**1、append()**

append()方法用于向 FormData 对象中添加键值对：

fd.append('key1',"value1");

fd.append('key2',"value2");

fd是 FormData 对象，可以新建的空的对象，也可以是已经包含 form 表单或其他键值对。

**2、set()**

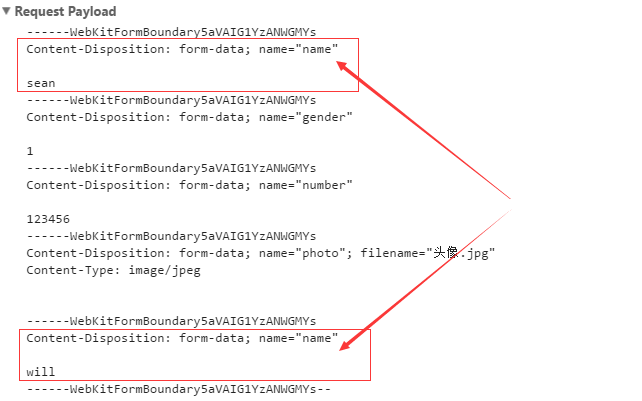
设置对应的键 key 对应的值 value(s)

fd.set('key1',"value1");

fd.set('key2',"value2");

看起来跟append() 方法有点类似，这两者的区别就是，当指定的 key 值存在时，append()方法是将新增的添加的所以的键值对最后，而set()方法将会覆盖前面的设置的键值对。还是通过实例来对比，我们在前面的 form 的基础上 append() 或 set() 新的键值对：

fd.append('name',"will");

有两个key为name的键值对：   


fd.set('name',"will");

只有一个key为name的键值对：   


以上就是 append() 和 set() 的区别。如果设置的key值不存在，那么两者的效果是一样的。

**3、delete()**

接收一个参数，表示你要删除的 key 值的名字，如果有多个相同 key 值，会一并删除：

fd.append('name','will');

fd.delete('name');

form 中的 name 信息以及通过append() 新增的name 的信息都被删除了。

**4、get() 和 getAll()**

接收一个参数，表示需要查找的 key 的名称，返回第一个该 key 对应的 value 值。如果有多个相同的 key， 而且要返回所有的这个 key 对应的 value 值。

同样以上面的 form 表单为基础：

fd.append('name','will');

console.log(fd.get('name')); // sean

fd.append('name','will');

console.log(fd.getAll('name')); // ["sean", "will"]

**5、has()**

该方法也接收一个参数，同样是 key 的名称，返回一个Boolean 值， 用来判断FormData 对象是否含有该 key。以上面的form为例：

console.log(fd.has('name')); // true

console.log(fd.has('Name')); // false

**6、keys()**

该方法不需要接收参数，返回一个迭代器，通过这个迭代器，我们可以遍历FormData 对象中所有的 key。以上面的form为例：

for (var key of fd.keys()) {

console.log(key);

}

结果为：

name

gender

number

photo

**7、values()**

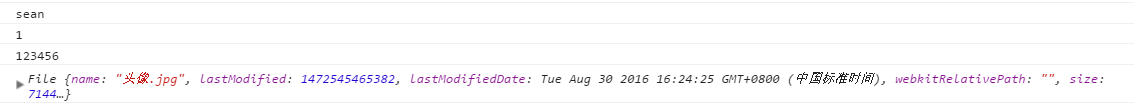
有遍历 key 的迭代，当然也就少不了遍历 value 的迭代器了。values()就是遍历value 的迭代器，用法与 keys() 类似：

for (var value of fd.values()) {

console.log(value);

}

结果：



**8、entries()**

有遍历 key 的迭代器，也有遍历 value 的迭代器，为何不搞一个两者一起的呢！entries()就是返回一个包含键值对的迭代器：

for(var pair of fd.entries()) {

console.log(pair[0]+ ', '+ pair[1]);

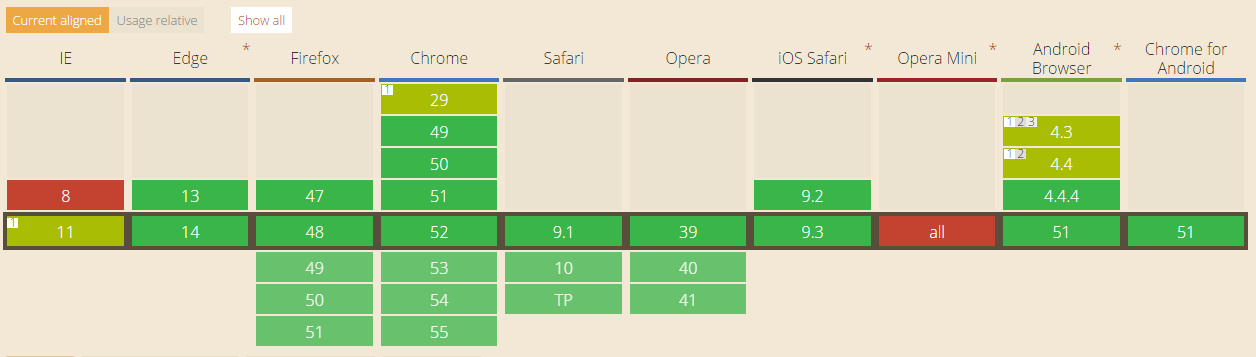
}

结果：



### ****FormData兼容性问题****

由于 FormData 是 XMLHttpRequest Level 2 新增的接口，现在 低于IE10 的IE浏览器不支持 FormData ，至于 上面介绍的 FormData 对象的方法经过测试，在 IE 浏览器中都不支持，具体的各大浏览器的支持情况可以参照下图：



# Touch events

“touch”模块添加以下事件，可以使用 [on](http://www.runoob.com/manual/zeptojs.html#on) 和 [off](http://www.runoob.com/manual/zeptojs.html#off)。

* tap —元素tap的时候触发。
* singleTap and doubleTap — 这一对事件可以用来检测元素上的单击和双击。(如果你不需要检测单击、双击，使用 tap 代替)。
* longTap — 当一个元素被按住超过750ms触发。
* swipe, swipeLeft, swipeRight, swipeUp, swipeDown — 当元素被划过时触发。(可选择给定的方向)

这些事件也是所有Zepto对象集合上的快捷方法。

**<style>.delete** { **display**: **none**; }**</style>**

**<ul** id=items**>**

**<li>**List item 1 **<span** class=delete**>**DELETE**</span></li>**

**<li>**List item 2 **<span** class=delete**>**DELETE**</span></li>**

**</ul>**

**<script>**

// show delete buttons on swipe

$('#items li').swipe(**function**(){

$('.delete').hide()

$('.delete', **this**).show()

})

// delete row on tapping delete button

$('.delete').tap(**function**(){

$(**this**).parent('li').remove()

})

**</script>**

# [URLSearchParams 接口](https://segmentfault.com/a/1190000005980048)

[**vanisseg**](https://segmentfault.com/u/vanisseg) 2016年07月15日发布

URLSearchParams 接口定义了很多个用来处理 URL 参数串的方法。

基本使用方法如下

**var** paramsString = "q=URLUtils.searchParams&topic=api" // location.search.slice(1)

**var** searchParams = **new** **URLSearchParams**(paramsString);

searchParams.has('topic') // true

searchParams.**get**('topic') // "api"

searchParams.getAll('topic') // ["api"]

searchParams.**get**('foo') // null，注意Firefox返回空字符串

searchParams.**set**('foo', 2);

searchParams.**get**('foo') // 2

searchParams.append('topic', 'webdev');

searchParams.toString() // "q=URLUtils.searchParams&topic=api&foo=2&topic=webdev"

searchParams.append('foo', 3);

searchParams.getAll('foo') // [2, 3]

searchParams.delete('topic');

searchParams.toString() // "q=URLUtils.searchParams&foo=2&foo=3"

URLSearchParams 还有三个方法返回迭代器对象：  
keys() 遍历所有参数名  
values() 遍历所有参数值  
entries() 遍历所有参数的键值对

URLSearchParams实例可以当作POST数据发送，所有数据都会URL编码。

**fetch**('https://example.com/api', {

method: 'POST',

body: params

}).then(...)

DOM 的 **a 元素节点的 searchParams 属性，**就是一个 URLSearchParams 实例。

**var** a = document.createElement('a');

a.href = 'https://example.com?filter=api';

a.searchParams.**get**('filter') // "api"

URLSearchParams 还可以与 URL 接口结合使用。

**var** url = **new** **URL**(location);

**var** foo = url.searchParams.**get**('foo') || 'somedefault';

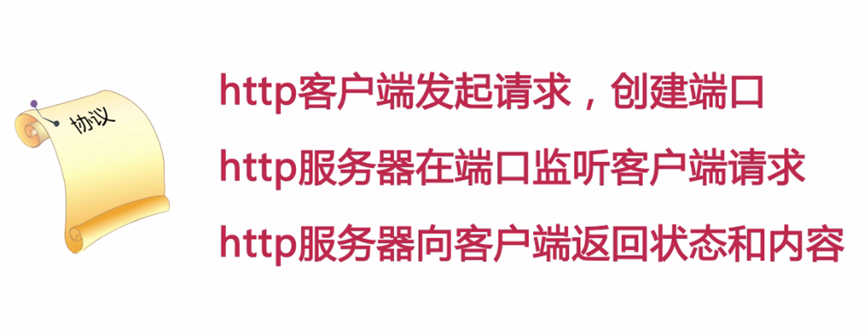
# JS与DOM属性和方法

下面的属性和方法可用于所有 HTML 元素上：

|  |  |
| --- | --- |
| **属性 / 方法** | **描述** |
| [element.accessKey](http://www.w3school.com.cn/jsref/prop_html_accesskey.asp) | 设置或返回元素的快捷键。 |
| [element.appendChild()](http://www.w3school.com.cn/jsref/met_node_appendchild.asp) | 向元素添加新的子节点，作为最后一个子节点。 |
| [element.attributes](http://www.w3school.com.cn/jsref/prop_node_attributes.asp) | 返回元素属性的 NamedNodeMap。 |
| [element.childNodes](http://www.w3school.com.cn/jsref/prop_node_childnodes.asp) | 返回元素子节点的 NodeList。 |
| [element.className](http://www.w3school.com.cn/jsref/prop_html_classname.asp) | 设置或返回元素的 class 属性。 |
| element.clientHeight | 返回元素的可见高度。 |
| element.clientWidth | 返回元素的可见宽度。 |
| [element.cloneNode()](http://www.w3school.com.cn/jsref/met_node_clonenode.asp) | 克隆元素。 |
| [element.compareDocumentPosition()](http://www.w3school.com.cn/jsref/met_node_comparedocumentposition.asp) | 比较两个元素的文档位置。 |
| [element.contentEditable](http://www.w3school.com.cn/jsref/prop_html_contenteditable.asp) | 设置或返回元素的文本方向。 |
| [element.dir](http://www.w3school.com.cn/jsref/prop_html_dir.asp) | 设置或返回元素的内容是否可编辑。 |
| [element.firstChild](http://www.w3school.com.cn/jsref/prop_node_firstchild.asp) | 返回元素的首个子。 |
| [element.getAttribute()](http://www.w3school.com.cn/jsref/met_element_getattribute.asp) | 返回元素节点的指定属性值。 |
| [element.getAttributeNode()](http://www.w3school.com.cn/jsref/met_element_getattributenode.asp) | 返回指定的属性节点。 |
| [element.getElementsByTagName()](http://www.w3school.com.cn/jsref/met_element_getelementsbytagname.asp) | 返回拥有指定标签名的所有子元素的集合。 |
| element.getFeature() | 返回实现了指定特性的 API 的某个对象。 |
| element.getUserData() | 返回关联元素上键的对象。 |
| [element.hasAttribute()](http://www.w3school.com.cn/jsref/met_element_hasattribute.asp) | 如果元素拥有指定属性，则返回true，否则返回 false。 |
| [element.hasAttributes()](http://www.w3school.com.cn/jsref/met_node_hasattributes.asp) | 如果元素拥有属性，则返回 true，否则返回 false。 |
| [element.hasChildNodes()](http://www.w3school.com.cn/jsref/met_node_haschildnodes.asp) | 如果元素拥有子节点，则返回 true，否则 false。 |
| [element.id](http://www.w3school.com.cn/jsref/prop_html_id.asp) | 设置或返回元素的 id。 |
| [element.innerHTML](http://www.w3school.com.cn/jsref/prop_html_innerhtml.asp) | 设置或返回元素的内容。 |
| [element.insertBefore()](http://www.w3school.com.cn/jsref/met_node_insertbefore.asp) | 在指定的已有的子节点之前插入新节点。 |
| [element.isContentEditable](http://www.w3school.com.cn/jsref/prop_html_iscontenteditable.asp) | 设置或返回元素的内容。 |
| [element.isDefaultNamespace()](http://www.w3school.com.cn/jsref/met_node_isdefaultnamespace.asp) | 如果指定的 namespaceURI 是默认的，则返回 true，否则返回 false。 |
| [element.isEqualNode()](http://www.w3school.com.cn/jsref/met_node_isequalnode.asp) | 检查两个元素是否相等。 |
| [element.isSameNode()](http://www.w3school.com.cn/jsref/met_node_issamenode.asp) | 检查两个元素是否是相同的节点。 |
| [element.isSupported()](http://www.w3school.com.cn/jsref/met_node_issupported.asp) | 如果元素支持指定特性，则返回 true。 |
| [element.lang](http://www.w3school.com.cn/jsref/prop_html_lang.asp) | 设置或返回元素的语言代码。 |
| [element.lastChild](http://www.w3school.com.cn/jsref/prop_node_lastchild.asp) | 返回元素的最后一个子元素。 |
| [element.namespaceURI](http://www.w3school.com.cn/jsref/prop_node_namespaceuri.asp) | 返回元素的 namespace URI。 |
| [element.nextSibling](http://www.w3school.com.cn/jsref/prop_node_nextsibling.asp) | 返回位于相同节点树层级的下一个节点。 |
| [element.nodeName](http://www.w3school.com.cn/jsref/prop_node_nodename.asp) | 返回元素的名称。 |
| [element.nodeType](http://www.w3school.com.cn/jsref/prop_node_nodetype.asp) | 返回元素的节点类型。 |
| [element.nodeValue](http://www.w3school.com.cn/jsref/prop_node_nodevalue.asp) | 设置或返回元素值。 |
| [element.normalize()](http://www.w3school.com.cn/jsref/met_node_normalize.asp) | 合并元素中相邻的文本节点，并移除空的文本节点。 |
| element.offsetHeight | 返回元素的高度。 |
| element.offsetWidth | 返回元素的宽度。 |
| element.offsetLeft | 返回元素的水平偏移位置。 |
| element.offsetParent | 返回元素的偏移容器。 |
| element.offsetTop | 返回元素的垂直偏移位置。 |
| [element.ownerDocument](http://www.w3school.com.cn/jsref/prop_node_ownerdocument.asp) | 返回元素的根元素（文档对象）。 |
| [element.parentNode](http://www.w3school.com.cn/jsref/prop_node_parentnode.asp) | 返回元素的父节点。 |
| [element.previousSibling](http://www.w3school.com.cn/jsref/prop_node_previoussibling.asp) | 返回位于相同节点树层级的前一个元素。 |
| [element.removeAttribute()](http://www.w3school.com.cn/jsref/met_element_removeattribute.asp) | 从元素中移除指定属性。 |
| [element.removeAttributeNode()](http://www.w3school.com.cn/jsref/met_element_removeattributenode.asp) | 移除指定的属性节点，并返回被移除的节点。 |
| [element.removeChild()](http://www.w3school.com.cn/jsref/met_node_removechild.asp) | 从元素中移除子节点。 |
| [element.replaceChild()](http://www.w3school.com.cn/jsref/met_node_replacechild.asp) | 替换元素中的子节点。 |
| element.scrollHeight | 返回元素的整体高度。 |
| element.scrollLeft | 返回元素左边缘与视图之间的距离。 |
| element.scrollTop | 返回元素上边缘与视图之间的距离。 |
| element.scrollWidth | 返回元素的整体宽度。 |
| [element.setAttribute()](http://www.w3school.com.cn/jsref/met_element_setattribute.asp) | 把指定属性设置或更改为指定值。 |
| [element.setAttributeNode()](http://www.w3school.com.cn/jsref/met_element_setattributenode.asp) | 设置或更改指定属性节点。 |
| element.setIdAttribute() |  |
| element.setIdAttributeNode() |  |
| element.setUserData() | 把对象关联到元素上的键。 |
| element.style | 设置或返回元素的 style 属性。 |
| [element.tabIndex](http://www.w3school.com.cn/jsref/prop_html_tabindex.asp) | 设置或返回元素的 tab 键控制次序。 |
| [element.tagName](http://www.w3school.com.cn/jsref/prop_element_tagname.asp) | 返回元素的标签名。 |
| [element.textContent](http://www.w3school.com.cn/jsref/prop_node_textcontent.asp) | 设置或返回节点及其后代的文本内容。 |
| [element.title](http://www.w3school.com.cn/jsref/prop_html_title.asp) | 设置或返回元素的 title 属性。 |
| element.toString() | 把元素转换为字符串。 |
| [nodelist.item()](http://www.w3school.com.cn/jsref/met_nodelist_item.asp) | 返回 NodeList 中位于指定下标的节点。 |
| [nodelist.length](http://www.w3school.com.cn/jsref/prop_nodelist_length.asp) | 返回 NodeList 中的节点数。 |

# http

1 输入网址  
2 域名解析  
2.1 浏览器自身缓存（chrome://net-internals/#dns）  
2.2 操作系统自身的dns缓存  
2.3 读取本地的host文件  
2.4 浏览器发起一个DNS的系统调用（本地主控DNS服务器：宽带运营商服务器查看本身缓存）  
2.5 运营商服务器发起一个迭代DNS解析的请求（->根域.com服务器->imooc.com域服务器->运营商服务器得到(结果)->操作系统内核并缓存->浏览器）  
3 浏览器与服务器的web程序tcp链接->**三次握手**  
4 发送http请求  
5 服务器处理请求，并将结果返回  
6 浏览器根据返回结果进行处理或页面渲染



# Webpack 3.0

**//安装之前的npm I –D**

**//package.json**

**"scripts": {**

**server": "webpack-dev-server --config webpack.config.js --open"**

**"build":"webpack"**

**},**

**//模板html，支持ejs，可以把css imgd src 都放里面，当然css的引入也可放在入口文件**

**//webpck配置概览**

|  |  |
| --- | --- |
|  | var website ={      publicPath:"http://192.168.1.108:1717/"  } |

const path = require('path');

const webpack = require('webpack')

const HtmlWebpackPlugin = require('html-webpack-plugin');

const CleanWebpackPlugin = require('clean-webpack-plugin');

const HtmlWebpackInlineSourcePlugin = require('html-webpack-inline-source-plugin');

const ExtractTextPlugin = require('extract-text-webpack-plugin')

module.exports = {

// devtool: 'source-map',

entry: {

main: path.resolve(\_\_dirname,'src/script/main.js'),

aa: path.resolve(\_\_dirname,'src/script/aa.js'),

test: path.resolve(\_\_dirname,'src/script/test.js')

},

output: {

path: path.resolve(\_\_dirname, 'dist'),

filename: 'js/[name].bundle.js',

publicPath:website.publicPath

},

module: {

rules: [

{

test: /\.js$/,

use: 'babel-loader',

exclude: /node\_modules/

},

{

test: /\.less$/,

use: ExtractTextPlugin.extract({

fallback: 'style-loader',

use: [

'css-loader',

'postcss-loader',

'less-loader'

]

}),

exclude: [/aa.less$/]

},

{

test: /\.less$/,

use: ExtractTextPlugin.extract({

fallback: 'style-loader',

use: [

'css-loader',

'postcss-loader',

'less-loader'

]

}),

include: [/aa.less$/]

},

{

test: /\.css$/,

use: [

'style-loader',

{

loader: 'css-loader',

options: {

importLoaders: 1

}

},

'postcss-loader',

]

}

]

},

plugins: [

new CleanWebpackPlugin(['dist']),

new ExtractTextPlugin('css/[name].css'),

new HtmlWebpackPlugin({

template: path.resolve(\_\_dirname,'src/index.html'),

filename: 'index.html',

chunks:['main'],

}),

new HtmlWebpackPlugin({

template: path.resolve(\_\_dirname,'src/aa.html'),

filename: 'aa.html',

chunks: ['aa'],

inlineSource: '.css$'

}),

new HtmlWebpackPlugin({

template: path.resolve(\_\_dirname,'src/test.html'),

filename: 'test.html',//默认放在output配置的path下的文件夹

chunks: ['test'],//使用的是哪个bundle

excludeChunks:[],

inlineSource: '.(js|css)$',

title: 'this is main.html',//配合ejs用在模板index.html

//title: 绑定在 htmlWebpackPlugin.options 上的参数，可在模板内使用

}),

new HtmlWebpackInlineSourcePlugin(),

//这是一个依赖于 html-webpack-plugin 的插件，目的是将 js 和 css 文件内联至 html，

new webpack.HotModuleReplacementPlugin(),

],

devServer: {

contentBase: './dist',

historyApiFallback: true,

hot: true,

inline: true

}

}

 extract-text-webpack-plugin插件能将 js 中引入的 css 文件分离出来，不再以 <style></style> 的形式插入页面，单独以 <link> 的方式引入。

 html-webpack-inline-source-plugin插件可以单独将 css 以 <style></style> 的形式打包完写入页面。然而并不可以，结合之前的几个处理 css 的 loader 可知，只能将其内联至 <script></script> 中，页面加载时动态插入至 html 中。

将 extract-text-webpack-plugin 和 html-webpack-inline-source-plugin 联合使用可以将 css 以 <style></style> 的形式打包完写入页面。此方法不会因为 css 的内联将分离生成的 .css文件清除，依然会保留，只是不会引入页面内。

在 html 模板页面中通过 ejs 语法配合 webpack 静态资源引入方法 require('xxx')，将其引入打包流程中，配合前两篇笔记提及的几个 loader 对资源文件进行优化分离并获取生成后地址，插入到新构建的 html 页面中。

const path = require('path')

const url = require('url')

const webpack = require('webpack')

const CleanWebpackPlugin = require('clean-webpack-plugin')

const HtmlWebpackPlugin = require('html-webpack-plugin')

module.exports = {

entry: {

app: path.resolve(\_\_dirname,'src/script/index.js'),

one: path.resolve(\_\_dirname,'src/script/one.js'),

vendor: [

'lodash'

]

},

output: {

filename: 'js/[name].bundle.js',

path: path.resolve(\_\_dirname, 'dist'),

},

module:{

rules: [

{

test: /\.js$/,

use: 'babel-loader',

include: path.resolve(\_\_dirname,'src'),

exclude: path.resolve(\_\_dirname,'node\_modules')

},

{

test: /\.css$/,

use: [

{

loader: 'file-loader',

options: {

name: '[name]-[hash:6].[ext]',

outputPath: 'css/'

}

},

"extract-loader",

{

loader: "css-loader",

options: {

minimize: true

}

}

]

},

{

test: /\.(jpe?g|png|gif|svg)$/i,

use: [

{

loader: 'url-loader',

options: {

name: '[name]-[hash:6].[ext]',

outputPath: 'image/',

limit: 1,

publicPath: url.format({

hostname:'localhost',

protocol:'http:',

port:8080,

pathname:'/dist/'

})

}

},

'image-webpack-loader'

]

}

]

},

plugins: [

new CleanWebpackPlugin(['dist']),

new HtmlWebpackPlugin({

template: path.resolve(\_\_dirname,'src/index.html'),

filename: 'index.html'

}),

new webpack.optimize.CommonsChunkPlugin({

name: 'vendor',

minChunks: function(module){

return module.context && module.context.indexOf("node\_modules") !== -1;

}

}),

new webpack.optimize.CommonsChunkPlugin({

name: "manifest",

minChunks: Infinity

})

]

};

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<meta http-equiv="X-UA-Compatible" content="ie=edge">

<title>this is webpack loader test</title>

<link rel="stylesheet" href="<%= require('./style/index.css') %>">

<link rel="stylesheet" href="<%= require('./style/one.css') %>">

</head>

<body>

<img src="<%= require('./image/shana.jpg')%>" alt="shana" />

<div class="yintama"></div>

</body>

</html>

在 html 模板页面中通过 ejs 语法配合 webpack 静态资源引入方法 require('xxx')，将其引入打包流程中，配合前两篇笔记提及的几个 loader 对资源文件进行优化分离并获取生成后地址，插入到新构建的 html 页面中。

# 浏览器进程与线程

https://mp.weixin.qq.com/s/HKfeCPIm5hV6s62U-TEARQ

## 浏览器都包含哪些进程？

知道了浏览器是多进程后，再来看看它到底包含哪些进程：（为了简化理解，仅列举主要进程）

1.Browser进程：浏览器的主进程（负责协调、主控），只有一个。作用有负责浏览器界面显示，与用户交互。如前进，后退等负责各个页面的管理，创建和销毁其他进程将Renderer进程得到的内存中的Bitmap，绘制到用户界面上，**网络资源的管理，下载等**

2.第三方插件进程：每种类型的插件对应一个进程，仅当使用该插件时才创建

3.GPU进程：最多一个，用于3D绘制等

4.浏览器渲染进程（浏览器内核）（Renderer进程，内部是多线程的）：默认每个Tab页面一个进程，互不影响。主要作用为

强化记忆：在浏览器中打开一个网页相当于新起了一个进程（进程内有自己的多线程）

## 重点是浏览器内核（渲染进程）

重点来了，我们可以看到，上面提到了这么多的进程，那么，对于普通的前端操作来说，最终要的是什么呢？答案是**渲染进程**

可以这样理解，页面的渲染，JS的执行，事件的循环，都在这个进程内进行。接下来重点分析这个进程

**请牢记，浏览器的渲染进程是多线程的**（这点如果不理解，**请回头看进程和线程的区分**）

终于到了线程这个概念了😭，好亲切。那么接下来看看它都包含了哪些线程（列举一些主要常驻线程）：

## 1.GUI渲染线程

* 负责渲染浏览器界面，解析HTML，CSS，构建DOM树和RenderObject树，布局和绘制等。
* 当界面需要重绘（Repaint）或由于某种操作引发回流(reflow)时，该线程就会执行
* 注意，**GUI渲染线程与JS引擎线程是互斥的**，当JS引擎执行时GUI线程会被挂起（相当于被冻结了），GUI更新会被保存在一个队列中**等到JS引擎空闲时**立即被执行。

## 2.JS引擎线程

* 也称为JS内核，负责处理Javascript脚本程序。（例如V8引擎）
* JS引擎线程负责解析Javascript脚本，运行代码。
* JS引擎一直等待着任务队列中任务的到来，然后加以处理，浏览器无论什么时候都只有一个JS线程在运行JS程序
* 同样注意，**GUI渲染线程与JS引擎线程是互斥的**，所以如果JS执行的时间过长，这样就会造成页面的渲染不连贯，导致页面渲染加载阻塞。

## 3.事件触发线程

* 归属于浏览器而不是JS引擎，用来控制事件循环（可以理解，JS引擎自己都忙不过来，需要浏览器另开线程协助）
* 当JS引擎执行代码块如setTimeOut时（也可来自浏览器内核的其他线程,如鼠标点击、AJAX异步请求等），会将对应任务添加到事件线程中
* 当对应的事件符合触发条件被触发时，该线程会把事件添加到待处理队列的队尾，等待JS引擎的处理
* 注意，由于JS的单线程关系，所以这些待处理队列中的事件都得排队等待JS引擎处理（当JS引擎空闲时才会去执行）

## 4.定时触发器线程

* 传说中的 setInternal与 setTimeout所在线程
* 浏览器定时计数器并不是由JavaScript引擎计数的,（因为JavaScript引擎是单线程的, 如果处于阻塞线程状态就会影响记计时的准确）
* 因此通过单独线程来计时并触发定时（计时完毕后，添加到事件队列中，等待JS引擎空闲后执行）
* 注意，W3C在HTML标准中规定，规定要求setTimeout中低于4ms的时间间隔算为4ms。

## 5.异步http请求线程

* 在XMLHttpRequest在连接后是通过浏览器新开一个线程请求
* 将检测到状态变更时，如果设置有回调函数，异步线程就**产生状态变更事件**，将这个回调再放入事件队列中。再由JavaScript引擎执行。

看到这里，如果觉得累了，可以先休息下，这些概念需要被消化，毕竟后续将提到的事件循环机制就是基于 事件触发线程的，所以如果仅仅是看某个碎片化知识， 可能会有一种似懂非懂的感觉。要完成的梳理一遍才能快速沉淀，不易遗忘。放张图巩固下吧：



再说一点，为什么JS引擎是单线程的？额，这个问题其实应该没有标准答案，譬如，可能仅仅是因为由于多线程的复杂性，譬如多线程操作一般要加锁，因此最初设计时选择了单线程。。。