

AJAX

AJAX 即 “Asynchronous Javascript And XML”（异步JavaScript和XML），是指一种创建交互式网页应用的网页开发技术。

- AJAX = 异步 JavaScript和XML（标准通用标记语言的子集）
- 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。
- 传统的网页（不使用 AJAX）如果需要更新内容，必须重载整个网页页面。

什么是AJAX?

AJAX只是是一个前端技术，不是一个新的语言。它是利用浏览器提供操作 HTTP 的接口 (XMLHttpRequest 或者 XMLHttpRequest)来操作 HTTP 以达到异步的效果。

网页渲染的同步与异步的区别

- 同步：当你在浏览器的地址栏里输入百度网址并访问的时候，浏览器都是创建新的 tabpage、新的内存块、加载网页的全部资源、渲染加载过来的资源。这些那都是从头开始的，就想上帝创作世界一样。只要与后台交互数据，那怕数据只有那么一丢丢，也得重新创造一次世界，如此反复。浏览器自己控制 HTTP 操作
- 异步：不需要重新创造一次世界，用之前已经存在的世界来达到目的。与后台交互数据不需要重新来渲染页面.自己来控制 HTTP 操作。

HTTP介绍

HTTP (HyperText Transfer Protocol) 超文本传输协议 在当前web环境中走的流量大部分都是走的 HTTP 流量，也就是说你在浏览器中访问任何东西，那怕是一张小图片也是 HTTP 来给你当搬运工显示在你面前的。而且AJAX就是基于HTTP来传输数据的。所以要想精通AJAX，适当的了解并掌握HTTP是十分必要的。

web客户端和服务端

web内容都是存储在web服务器上的，web服务器所使用的是http协议，因此经常会被称为http服务器。这些http服务器存储量因特网中的数据，如果http客户端发出请求的话，它们会提供数据。客户端向服务器发送http请求，服务器会在http响应中会送所请求的数据。

http客户端和http服务器共同构成了万维网的基本组建。可能你每天都是使用http客户端。最常见的http客户端就是浏览器。web浏览器向服务器请求http对象，并将这些对象显示在你的屏幕上。

HTTP事务

一个http事务由一条(从客户端发往服务器的)请求命令和一个(从服务器发回客户端的)响应结果组成。这种通信时通过名为 `HTTP message` 的格式化数据块进行的。只有当请求和响应都成功时此http事务才算成功，也就是这条http才算成功。只有当其中任意一个命令(请求或者响应)失败，那么这个http就算失败。一个http就是一个http事务，且http事务完成之后此http不可在复用。

http报文

http报文是由一行一行的简单字符串组成的。http报文都是纯文本，不是二进制代码，所以人们可以很方便地对其进行读写。

http报文分为三部分：

- 起始行 报文的第一行就是起始行，在请求报文中用来说明做些什么，在响应报文中说明出现了什么情况。
- 首部字段 起始行后面有零个或多个首部字段。每个首部字段都包含了一个名字和一个值，首部分为5种类型：
`通用首部、请求首部、响应首部、实体首部、扩展首部`
- 主体 报文主体包含了所有类型的数据。请求主体中报错了要发送给web服务器的数据；响应主体中装载了要返回给客户端的数据。起始行和首部字段都是结构化的文本形式的，而主体可以包含任意的二进制数据。当然，主体中也可以包含文本。

HTTP 方法

http支持几种不同的请求命令，这些命令被称为 `http方法` 每条http请求报文都包括一个方法。这个方法会告诉服务器执行什么动作(获取一个web页面、运行一个网关程序、删除一个文件)。

常见来http方法如下

- GET 从服务器向客户端发送命名资源，主要是传给服务器一些参数来获取服务器上指定的资源。
- POST 将客户端数据发送到一个服务器网关程序。
- DELETE 从服务器上删除命名资源
- HEAD 仅发送命名资源中的http首部
- PUT 将来自客户端的数据存储到一个服务器资源中去
- TRACE 对报文进行追踪
- OPTIONS 决定可以从服务器上执行哪些方法

GET与POST的区别：

- URL长度限制 浏览器对URL有大小限制，chrome 8k firefox 7k ie 2k
- 资源大小限制：get方法限制大小，get是将数据直接拼接在URL后端query部分，而浏览器是对URL有长度限制的，所以get有大小限制。post不限制大小。因为post是将数据放到请求的主体里，而主体是不限制大小的，所以post没有大小限制。
- 功能 get主要是用来从服务器拉取数据，而post主要是用来将数据发送到服务器。
- 安全 get可以看到发送给服务器的数据，而post不会被看到，因为post把数据放到主体里了。

HTTP 状态码

每条http响应报文返回时都会携带一个状态码。状态码是一个三位数字的代码，告知客户端请求是否成功，或者是否需要采取其他操作。

几种常见的状态码：

- 200 OK 文档正确返回
- 301 Redirect 永久重定向。一直从其他地方去获取资源
- 302 Redirect 临时重定向。临时到其他地方去获取资源
- 303 see other、307 Temporary Redirect 将客户端重定向到一个负载不大的服务器上，用于负载均衡和服务器失联
- 404 Not Found 无法找到这个资源
- 500 Internal Server Error 服务器错误

伴随着每个数字状态码，http还会发送一条解释性的 原因短语 文本。包含文本短语主要是为了进行描述，所有的处理过程使用的都是数字码。

http软件处理下列状态码和原因短语的方式是一样的：

- 200 OK
- 200 Document attached
- 200 Success
- 200 All's cool, dude

MIME Type

因特网上有数千种不同的数据类型，http仔细地给每种要通过web传输的对象都打上了名为MIME类型(MIME Type)的数据格式标签。最初设计MIME(Multipurpose Internet Mail Extension，多用途因特网邮件扩展)是为了解决在不同的电子邮件系统之间搬移报文时存在的问题。MIME在电子邮件系统中工作得非常好，因此HTTP也采纳了它，用它来描述并标记多媒体内容。

web服务器会为所有的http对象数据附加一个MIME类型。当web浏览器从服务器中取回一个对象时，回去查看相关的MIME类型，看看它是否知道应该如何处理这个对象。大多数浏览器都可以处理数百种常见的对象类型。

MIME Type就是告诉浏览器用什么方式来处理这个数据。

MIME类型是一种文本标记，表示一种主要的对象类型和一个特定的子类型，中间由一条斜杠来分隔。

- html格式的文本文档由 `text/html` 类型来标记
- 普通的ASCII文本文档由 `text/plain` 类型来标记
- JPEG格式的图片为 `image/jpeg` 类型
- GIF格式的图片为 `image/gif` 类型
- 表单提交由 `application/x-www-form-urlencoded` 类型来标记

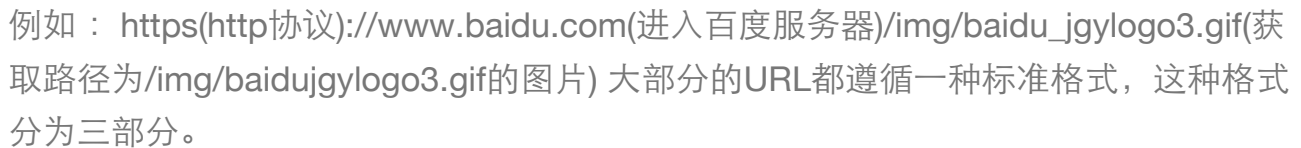
MIME类型在HTTP协议中的表现为 `Request Header` 或者 `Response Header` 中的 `Content-Type`

URI、URL、URN

URI：每个web服务器资源都有一个名字，这样客户端就可以说明他们感兴趣的资源是什么了。服务器资源名被称为 统一资源标识符(Uniform Resource Identifier, URI) URI 就像因特网上的邮政地址一样，在世界范围内唯一标识并定位信息资源。

例如 https://www.baidu.com/img/baidu_jgylogo3.gif 这是一个百度服务器上一个图片资源的URI

URL：统一资源定位符(URL) 是资源标识符最常见的形式。URL描述了一台特定服务器上某资源的特定位置。它们可以明确说明如何从一个精确、固定的位置获取资源。

例如：[https\(http协议\)://www.baidu.com\(进入百度服务器\)/img/baidu_jgylogo3.gif](https://www.baidu.com/img/baidu_jgylogo3.gif)(获取路径为img/baidujgylogo3.gif的图片) 大部分的URL都遵循一种标准格式，这种格式分为三部分。

- 第一部分 方案(scheme)，说明了访问资源所使用的协议类型。着部分通常就是http协议。
- 第二部分 服务器位置，给出来服务器的因特网地址(比如，www.baidu.com)。
- 第三部分 资源路径，指定了web服务器上的某个资源(比如，[/img/baidujgylogo3.gif](http://www.baidu.com/img/baidujgylogo3.gif))。现在，几乎所有的URI都是URL 大多数的URL方案的URL语法都建立在这个有9部分构成的通用格式上：

```
<scheme>://<user>:<password>@<host>:<port>/<path>;<params>?<query>#<frag>
```

URN：URI 的第二种形式就是 统一资源名称(URN)。URN是作为特定内容的唯一名称使用的，与目前资源地无关。使用这些与位置无关的URN，就可以将资源四处搬移。通过URN，还可以用同一个名字通过多种网络访问协议来访问资源。

比如，不论因特网标准文档RFC 2141 位于何处(甚至可以将其复制到多个地方)，都可以用下列URN来命名它：

```
urn:ietf:rdc:2141
```

URN目前仍然处于试验阶段。

URI包括两种形式，一种是URL一种是URN。目前大部分会不加区别的使用URI和URL。因为URL是URI的一个子集。

方案

方案 实际上是规定如何访问指定资源的主要标识符，它会告诉负责解析URL的应用程序应该使用什么协议，最常见的就是HTTP方案。 常见的方案格式：

- http 超文本传输协议方案，默认端口为80
- https 方案https和方案http是一对，为一个区别在于使用了网景的SSL，SSL为http提供了端到端的加密机制。默认端口为443
- mailto 指向Email地址。
- ftp 文件传输协议，可以用来从ftp服务器上传下载文件
- file 访问本地文件
- telnet 用于交互式访问业务

浏览器兼容性

在IE7以下版本的IE系列浏览器中，要应用AJAX必须使用 `ActiveXObject`(这个对象是一个微软推广和支持在Internet Explorer中，不在Windows应用商店的应用程序。) 方法。在标准浏览器(chrome、firefox、opera、safari、ie7+)当中则使用 `XMLHttpRequest` 对象。

如何发起AJAX?

在低版本IE(7-)中使用 `ActiveXObject` 构造AJAX对象时需要传入一个String类型的参数 `Microsoft.XMLHTTP`，也可以使用 `Msxml3.XMLHTTP` 和 `Msxml2.XMLHTTP`。因为一开始是 `Microsoft.XMLHTTP` 之后变成 `Msxml2.XMLHTTP` 及更新版的 `Msxml3.XMLHTTP`

```
// code for IE6, IE5
var xmlhttp1 = new ActiveXObject("Microsoft.XMLHTTP");
var xmlhttp2 = new ActiveXObject("Msxml2.XMLHTTP");
var xmlhttp3 = new ActiveXObject("Msxml3.XMLHTTP");
```

在标准浏览器中则使用 `XMLHttpRequest` 对象

```
// code for IE8+, Firefox, Chrome, Opera, Safari
var xmlhttp = new XMLHttpRequest();
```

为了在项目中可以在任何浏览器中使用AJAX所以我们必须做一个判断，如果浏览器为低版本IE就使用 `ActiveXObject` 对象否则使用 `XMLHttpRequest` 对象。代码如下：

```
var XHR = function () {  
    var xmlhttp;  
    if (window.XMLHttpRequest) {  
        // code for IE7+, Firefox, Chrome, Opera, Safari  
        xmlhttp = new XMLHttpRequest();  
    }  
    else {  
        // code for IE6, IE5  
        xmlhttp = new ActiveXObject("Microsoft.XMLHTTP");  
    }  
    return xmlhttp;  
};  
var xmlObj=XHR();  
console.log(xmlObj);
```

这样的话，我们就可以得到一个在任何浏览器都能发起AJAX的方法。但是这样左右一个坏处，就是每次获取AJAX对象时都会判断一次，这样做很费时费力。所以，我们利用 惰性函数 的概念来实现一个只需要第一次判断，后面都不需要判断的方法。

```

var XHR = function () {
    // 将浏览器支持的AJAX对象放入一个function中，并且根据固定的顺序
    // 放到一个队列里。
    for (var AJAXObj = [function () {
        return new XMLHttpRequest
    }, function () {
        return new ActiveXObject("Msxml2.XMLHTTP")
    }, function () {
        return new ActiveXObject("Msxml3.XMLHTTP")
    }, function () {
        return new ActiveXObject("Microsoft.XMLHTTP")
    }], val = null, index = 0; index < AJAXObj.length; index++) {
        // 此方法的核心，如果当前浏览器支持此对象就用val保存起来
        // 用保存当前最适合ajax对象的function替换XHR方法，并且结束该循环
        // 这样第二次执行XHR方法时就不需要循环，直接就能得到当前浏览器
        // 最适合ajax对象。如果都不支持就抛出自定义引用错误。
        try {
            val = AJAXObj[index]()
        } catch (b) {
            continue
        }
        // 假设当前浏览器为标准浏览器，此处执行完毕之后
        console.log(XHR);
        // 结果为：function () {
        //   return new XMLHttpRequest
        // };XHR成功替换。
        XHR=AJAXObj[index];
        break
    }
    if (!val) {
        throw new ReferenceError("XMLHttpRequest is not supported")
    }
    return val;
};
var xmlObj=XHR();
console.log(xmlObj);

```

本方法的核心就是利用 **惰性函数**。这才是正点。第一次计算得到的值，供内部函数调用，然后用这个内部函数重置外部函数（因为同名），以后就不用计算了，也不用判断分支条件。这时函数就相当于一个被赋值的变量。

接下来我们依次介绍 **XMLHttpRequest** 和 **ActiveXObject** 如何使用。

使用XMLHttpRequest

XMLHttpRequest 是一个 JavaScript 对象,它最初由微软设计,随后被 Mozilla,Apple, 和

Google采纳. 如今,该对象已经被 W3C组织标准化. 通过它,你可以很容易的取回一个URL上的资源数据. 尽管名字里有XML, 但XMLHttpRequest 可以取回所有类型的数据资源,并不局限于XML. 而且除了HTTP ,它还支持file 和 ftp 协议.

在浏览器中创建并使用一个 XMLHttpRequest 实例, 可以使用如下语句:

```
var req = new XMLHttpRequest();
//do something...
```

XMLHttpRequest 让发送一个HTTP请求变得非常容易。你只需要简单的创建一个请求对象实例, 打开一个URL, 然后发送这个请求。当传输完毕后, 结果的HTTP状态以及返回的响应内容也可以从请求对象中获取。本页把这个强大的JavaScript对象的一些常用的甚至略有晦涩的使用案例进行了一下概述。

XMLHttpRequest对象方法概述:

- 返回值 方法(参数)
- void abort ();中止操作
- DOMString getAllResponseHeaders ();得到所有响应头
- DOMString? getResponseHeader (DOMString header);得到指定响应头
- void open (DOMString method, DOMString url, optional boolean async, optional DOMString? user, optional DOMString? + password);开启XMLHttpRequest对象
- void overrideMimeType (DOMString mime);重写MIME类型
- void send ();发送请求, 此方法有六种重载
- void send (ArrayBuffer data);发送二进制流
- void send (Blob data);发送二进制块
- void send (Document data);发送文档
- void send (DOMString? data);发送字符串
- void send (FormData data);发送格式化表单数据
- void setRequestHeader (DOMString header, DOMString value);设置请求头

XMLHttpRequest对象属性概述:

- 属性名 格式类型 说明
- onreadystatechange Function? 一个JavaScript函数对象, 当readyState属性改变时会调用它。回调函数会在用户接口线程中调用。(警告: 不能在本地代码中使用. 也不应该在同步模式的请求中使用.)
- readyState unsigned short 请求的五种状态: 0 UNSENT (未打开) open()方法还未被调用、 1 OPENED (未发送) send()方法还未被调用、 2 HEADERS_RECEIVED (已获取响应头) send()方法已经被调用, 响应头和响应状态已经返回、 3 LOADING (正在下载响

应体) 响应体下载中; `responseText`中已经获取了部分数据、 4 DONE (请求完成) 整个请求过程已经完毕.

- `response` `varies` 响应实体的类型由 `responseType` 来指定, 可以是 `ArrayBuffer`, `Blob`, `Document`, `JavaScript` 对象 (即 "json"), 或者是字符串。如果请求未完成或失败, 则该值为 `null`。
- `responseText` `DOMString` 此次请求的响应为文本, 或是当请求未成功或还未发送时为 `null`。只读。
- `responseType` `XMLHttpRequestResponseType` 设置该值能够改变响应类型。就是告诉服务器你期望的响应格式: `"` (空字符串) 字符串(默认值)、 `"ArrayBufferView"` `ArrayBufferView`、 `"blob"` `Blob`、 `"document"` `Document`、 `"json"` `JavaScript` `Object`、 `"text"` 字符串。
- `responseXML` `Document?` 本次请求的响应是一个 `Document` 对象, 如果是以下情况则值为 `null`: 请求未成功, 请求未发送, 或响应无法被解析成 XML 或 HTML。当响应为 `text/xml` 流时会被解析。当 `responseType` 设置为"document", 并且请求为异步的, 则响应会被当做 `text/html` 流来解析。只读。(注意: 如果服务器不支持 `text/xml` `Content-Type` 头, 你可以使用 `overrideMimeType()` 强制 `XMLHttpRequest` 将响应解析为 XML。)
- `status` `unsigned short` 该请求的响应状态码 (例如, 状态码200 表示一个成功的请求)。只读。
- `statusText` `DOMString` 该请求的响应状态信息, 包含一个状态码和原因短语 (例如 "200 OK")。只读。
- `upload` `XMLHttpRequestUpload` 可以在 `upload` 上添加一个事件监听来跟踪上传过程。
- `withCredentials` `boolean` 表明在进行跨站(cross-site)的访问控制(Access-Control)请求时, 是否使用认证信息(例如cookie或授权的header)。默认为 `false`。注意: 这不会影响同站(same-site)请求。

XMLHttpRequest和本地文件

网页中可以使用相对URL的能力通常意味着我们能使用本地文件系统来开发和测试HTML, 并避免对web服务器进行不必要的部署。然后当使用XMLHttpRequest进行Ajax编程时, 这通常是不可行的。XMLHttpRequest用于HTTP和HTTPS协议一起工作。理论上, 它能够同像FTP这样的其他协议一起工作, 但比如像请求方法和响应状态码等部分API是HTTP特有的。如果从本地文件中加载网页, 那么该网页中的脚本将无法通过相对URL使用XMLHttpRequest, 因为这些URL将相对于 `file://URL` 而不是 `http://URL`。而同源策略通常会阻止使用绝对 `http://URL`。结果是当使用XMLHttpRequest时, 为了测试它们通常必须把文件上传到web服务器(或运行一个本地服务器)

方法

abort()

```
req.abort();
```

如果请求已经被发送,则立刻中止请求(cancel).abort()方法在所有的XMLHttpRequest版本和XHR2中可用,调用abort()方法在这个对象上触发abort事件。可以通过XMLHttpRequest对象的 `onabort` 属性是否存在来判断。

```
//if成立的话 abort() 存在,否则不存在
if('onabort' in req){
    req.abort();
}
```

getAllResponseHeaders()

```
var allHeaders = req.getAllResponseHeaders();
```

返回所有响应头信息(响应头名和值),如果响应头还没接收,则返回null。(注意:对于多部分请求,这将返回头从请求的当前一部分,而不是从原来的通道上。)

getResponseHeader()

```
var dateHeader = req.getResponseHeader("Date");
```

返回指定的响应头的值,如果响应头还没被接受,或该响应头不存在,则返回null.

open()

注意:调用此方法必须已经发出主动请求 (`open()` 或 `openRequest()` 已经被调用) 是相当于调用 `abort()`。

```
req.open(http Method,URL,isAsync,userName,password);
// 参 数
//http Method 请求所使用的HTTP方法；"POST" 或者 "GET"。如果下个参数是非HTTP(S)的URL,则忽略该参数。
//URL 该请求所要访问的URL
//isAsync 一个可选的布尔参数，默认为真，指示是否异步执行操作。如果该值为false时，send()方法不返回直到收到响应。如果为true，完成交易的通知使用事件侦听器提供。这必须是真实的，如果多部分属性为true，或将引发异常。
//userName 可选的用户名，以用于身份验证的目的；默认情况下，这是一个空字符串。
//password 可选的密码用于认证的目的；默认情况下，这是一个空字符串。
```

初始化一个请求. 该方法用于JavaScript代码中;如果是本地代码, 使用 openRequest()方法代替.

如果传入的http method不区分大小写与 *CONNECT, DELETE, GET, HEAD, OPTIONS, POST, PUT, TRACE*, 或者 *TRACK* 匹配上, 从范围0x61 (ASCII a) 每个字节0x7A (ASCII a) 减去0x20。把小写转换成大写(如果它不匹配任何上述情况, 它是通过传递字面上, 包括在最后的请求。)

如果http method 不区分大小写匹配到 *CONNECT, TRACE*, 或者 *TRACK* 这三个方法, 抛出 "SecurityError" 异常, 因为安全风险已被明确禁止。旧浏览器并不支持所有这些方法, 但至少 *HEAD* 得到广泛支持

overrideMimeType()

```
req.overrideMimeType("text/plain");
// 参 数 必 须 为 MIME Type 格式
```

重写由服务器返回的MIME类型。这可以用于一下情况。假如你将下载XML文件, 而你计划把它当成纯文本对待。可以使用overrideMimeType()让XMLHttpRequest知道它不需要把文件解析为XML文档:

```
// 不要 把 响 应 作 为 XML 文 档 处 理
req.overrideMimeType('text/plain; charset=utf-8');
```

这个方法必须send()之前调用。

onload

```
req.onload=function(){
    console.log(this.responseText,this.responseType,this.response);
}
```

当XMLHttpRequest对象加载完成时(readyState为4)触发。且只与readyState有关，与status和statusText无关。所以当注册onload的方法执行时不一定为成功的状态。只是也仅仅是这个条http事务完成而已。不注册此方法则onload默认为null。

onreadystatechange

```
req.onreadystatechange=function(){
//判断ajax成功,此写法有兼容性。
    if(this.readyState==this.DONE&&this.statusText=="OK"){
        console.log(this.response,'成功');
    }
//判断ajax成功还有另外一种写法
    if(this.readyState==4&&this.status==200){
        console.log(this.responseText,'成功');
    }
}
```

每当readyState的值改变时就会出发该方法。不注册此方法则onreadystatechange默认为null。

send()

注意: 所有相关的事件绑定必须在调用send()方法之前进行。

```
req.send(undefined||null||ArrayBufferView|Blob|XML|string|FormData);
//此方法有7种参数重载
```

发送请求. 如果该请求是异步模式(默认),该方法会立刻返回. 相反,如果请求是同步模式,则直到请求的响应完全接受以后,该方法才会返回.

GET请求 绝对 没有主体, 所以应该传递null或者省略这个参数。POST请求通常拥有主体, 同时它应该匹配使用setRequestHeader()指定的 Content-Type 头。

如果数据是一个Document, 它在发送之前被序列化。当发送文件时, Firefox之前的版本3的版本总是使用UTF-8编码发送请求; Firefox 3的正常使用发送, 如果没有指定编码由body.xml编码, 或者UTF-8指定的编码文件。

setRequestHeader()

```
req.setRequestHeader("header","value");  
// 设置制定的请求头，此方法必须在 send() 执行之前执行。  
// header 将要被赋值的请求头名称。  
// value 给指定的请求头赋的值。
```

给指定的HTTP请求头赋值.在这之前,你必须确认已经调用 `open()` 方法打开了一个url. 如果对相同的头调用`setRequestHeader()`多次, 新值不会取代之前指定的值, 相反, HTTP 请求将包含这个头的多个副本或者个头将指定多个值。例如:

```
req.setRequestHeader("Accepts","text/html");  
req.setRequestHeader("Accepts","text/css");  
// 那个请求头中的Accepts的值为 "text/html,text/css"
```

你不能制定 `Content-Length`、`Date`、`Referer` 或 `User-Agent` 头, `XMLHttpRequest`将自动添加这些头而防止伪造他们。类似地, `XMLHttpRequest`对象自动处理cookie、连接时间、字符集和编码判断, 所以你不能向`setRequestHeader()`传递这些头:

- `Accept-Charset`
- `Content-Transfer-Encoding`
- `Date`
- `Connection`
- `Expect`
- `Content-Length`
- `Host`
- `Cookie`
- `Keep-Alive`
- `User-Agent`
- `Cookie2`
- `Referer`

你能为请求指定 `Authorization` 头, 但通常不需要这么做。如果请求一个受密码保护的URL, 把用户名和密码作为第四个和第五个参数传递给`open()`, 则`XMLHttpRequest`将设置合适的头。

顺序问题

HTTP请求的各部分有指定顺序: 请求方法和URL首先到达, 然后是请求头, 最后是请求主题。MXLHttpRequest实现通常直到调用`send()`方法才开始启动网络。单XMLHttpRequest API的设计似乎使每个方法都将写入网络流。这意味着调用XMLHttpRequest方法的顺序必须匹配HTTP请求的架构。例如, `setRequestHeader()`方法的调用必须在调用`send()`之前但在

调用`open()`之后，否则它将抛出异常。

示例代码：

```
// 一些简单的代码做一些与数据通过网络获取的XML文档
function processData(data) {
    // taking care of data
}

function handler() {
    if(this.readyState == this.DONE) {
        if(this.status == 200 &&
           this.responseXML != null &&
           this.responseXML.getElementById('test').textContent) {
            // success!
            processData(this.responseXML.getElementById('test').textContent);
            return;
        }
        // something went wrong
        processData(null);
    }
}

var client = new XMLHttpRequest();
client.onreadystatechange = handler;
client.open("GET", "unicorn.xml");
client.send();

// 如果你只是想记录一个消息服务器
function log(message) {
    var client = new XMLHttpRequest();
    client.open("POST", "/log");
    client.setRequestHeader("Content-Type", "text/plain; charset=UTF-8");
    client.send(message);
}

// 或者，如果您要检查服务器上的文档的状态
function fetchStatus(address) {
    var client = new XMLHttpRequest();
    client.onreadystatechange = function() {
        // in case of network errors this might not give reliable results
        if(this.readyState == this.DONE)
            returnStatus(this.status);
    }
    client.open("HEAD", address);
    client.send();
}
```


在低版本IE浏览器中使用ActiveXObject时需要注意的地方

使用ActiveXObject与XMLHttpRequest对象大体相同。不过还是有许多不同的地方。

不同点：

- ActiveXObject对象中没有timeout属性，没有ontimeout方法。
- ActiveXObject对象中不支持statustext属性。
- ActiveXObject对象中没有DONE、OPEN、UNSENT、HEADERS_RECEIVED、DONE 这些属性。
- ActiveXObject对象中没有onload方法。
- ActiveXObject对象中send()不支持 `ArrayBuffer|Blob|FormData` 等类型重载。
- ActiveXObject对象中没有withCredentials属性。

由于ActiveXObject对象只在IE5、IE6中使用，所以很多功能都没有。所以使用时需注意。

ActiveXObject对象用法：

```
var http;
if(window.XMLHttpRequest){
    http = new XMLHttpRequest();
} else if (window.ActiveXObject){
    http = new ActiveXObject("Microsoft.XMLHTTP");
    if (!http){
        http = new ActiveXObject("Msxml2.XMLHTTP");
    }
}
function handleStateResponse() {
    if (this.readyState == 4) {
        if(this.status == 200) {
            var results = this.responseText;
            console.log(results);
        } else {
            alert ( "Not able to retrieve description" );
        }
    }
}
http.open();
http.onreadystatechange=handleStateResponse;
http.send();
```

AJAX 示例

上面的基础知识和使用处理ajax的对象需要注意的一些地方我们都都已经提到了，现在我们

开始写ajax示例，以便更好的巩固ajax知识。

这个示例是参照jQuery的ajax函数使用方法来编写的,并且可以使用链式调用以增强体验。

最终代码如下：

```
// 编写一个$http对象，这个对象下有get()、post()、getScript()、ajax()这些方法。
// 每个方法都返回一个可供链式注册的对象。
// ajax()方法提供多个固定的参数以达到高配置性。
(function (global, undefined) {
    // 如果全局已经存在$http对象的话就直接返回，不执行任何代码。
    if (global.$http) {
        return
    }
    var http = global.$http = {};
    // 判断类型
    function isType(str) {
        return function (obj) {
            return Object.prototype.toString.call(obj) == '[object ' + str + ']';
        }
    }

    var isObject = isType("Object");
    var isFunction = isType("Function");
    var isNumber = isType("Number");
    var isString = isType("String");
    var isBoolean = isType('Boolean');
    var isArray = isType('Array');
    // 检测参数中有没有`?`
    var hasSearch = function (url) {
        return /^.+?\?[^\?]*$/g.test(url)
    };
    // 循环帮助函数
    var each=(function(){
        if ( [].forEach )
            return function(arr,func){
                [].forEach.call(arr, func);
            }
        else
            return function(arr,func){
                for (var i = 0; i < arr.length; i++) {
                    func.call(arr, arr[i], i, arr);
                }
            }
    })();

    // 获取 ajax 对象，
```

```

var getXHR = function () {
    for (var list = [function () {
        return new XMLHttpRequest
    }, function () {
        return new ActiveXObject("Microsoft.XMLHTTP")
    }, function () {
        return new ActiveXObject("Msxml2.XMLHTTP")
    }, function () {
        return new ActiveXObject("Msxml3.XMLHTTP")
    }], temp = null, index = 0; index < list.length; index++) {
        try {
            temp = list[index]()
        } catch (ex) {
            continue
        }
        getXHR = list[index];
        break;
    }
    if (!temp) {
        throw new ReferenceError("browser is not supported")
    }
    return temp;
};

```

```

http.ajax = function (options) {
    // 如果options不是一个对象那么此方法后续将不执行。
    if (!isObject(options)) return;

```

// 默认参数对象

```
var defaultOptions = {
```

// 添加到请求头`Accepts`中，用来说明这个Ajax接收什么MIME Type的数据。Array类型

```
    accepts: undefined,
```

// 是否为异步，默认为true。Boolean类型

```
    async: true,
```

// 在执行send()方法之前调用的函数。Function类型

```
    beforeSend: undefined,
```

// 是否缓存。Boolean类型

```
    cache: false,
```

// 不管成功或失败都会执行的函数。Function类型

```
    complete: function () {},
```

// 添加到请求头`Content-Type`中，标识此http是什么MIME Type，默认为`application/x-www-form-urlencoded; charset=UTF-8`。String类型

```
    contentType: 'application/x-www-form-urlencoded; charset=UTF-8',
```

// 宿主对象。Object类型

```
    context: undefined,
```

// 发送给服务器的数据。Object或者String类型

```
    data: undefined,
```

```

// 服务器返回的数据格式，可以为
`xml|bolb|arraybuffer|html|script|json|text`。String 类型

dataType: 'text',
// 过滤服务器返回数据。Function 类型
dataFilter: undefined,
// 执行失败、parseError 或者 timeout 执行的函数。Function 类型
error: function (xhr, xhr.status, text) {},
// 自定义头信息。Object 类型
headers: {},
// 重写服务器返回 MIME Type。String 类型
mimeType: '',
// URL 认证密码。String 类型
password: undefined,
// 注册对应 http 状态码时执行的函数。Object 类型
statusCode: {},
// Ajax 成功时执行的函数。Function 类型
success: function (response) {},
// 超时毫秒值，此值必须大于 500 毫秒，否则不生效。Number
类型
timeout: undefined,
// 请求的 http 方法，可以为 `get|post|head|put|delete`。String 类
型
type: '',
// 请求的 URL，此 URL 的方案不可以为 `file`。String 类型
url: '',
// URL 认证账号。String 类型
username: undefined,
// 在 send() 执行前，通过此函数操作 xhr 对象。Function 类型
setXhrFields: function (xhr){}
},
// 临时变量
tempVal;

// 覆盖默认参数对象
for (tempVal in defaultOptions) {
    if (options.hasOwnProperty(tempVal) && options[tempVal])
        defaultOptions[tempVal] = options[tempVal];
}

var xhr = getXHR(), _promise = new __promise();

if (!/^(get|post|head|put|delete)$/img.test(defaultOptions.type)) {
    throw new ReferenceError('not supported this http method');
}
if
(!/^(xml|bolb|arraybuffer|html|script|json|text)$/img.test(defaultOptions.dataType))
{
    throw new TypeError('not supported ' + defaultOptions.dataType + '
data type');

```

```
}
if (!isString(defaultOptions.url)) {
    throw new TypeError('url must be a string')
}
// 是否使用缓存，不使用则在URL后添加一个随机数
if (defaultOptions.cache) {
    defaultOptions.url += (hasSearch(defaultOptions.url) ? '_' : '?_')
+ Math.random() * (1 << 24) | 0;
}
// 将发送给服务器的数据从Object类型转为String类型
if (defaultOptions.data) {
    if (isObject(defaultOptions.data)) {
        var arr = [];
        for (tempVal in defaultOptions.data) {
            if (defaultOptions.data.hasOwnProperty(tempVal)) {
                // 此处参见下文的表单操作
                arr.push(encodeURIComponent(tempVal) + "=" +
encodeURIComponent(defaultOptions.data[tempVal]));
            }
        }
        defaultOptions.data = arr.join('&');
    }
    // 当请求的http方法为get、head、delete中一个时直接拼接到
URL后并且删除data。参见send()方法解释。
    if (/^(get|head|delete)$/img.test(defaultOptions.type)) {
        defaultOptions.url += (hasSearch(defaultOptions.url) ? ' ' : '?')
+ defaultOptions.data;
        delete defaultOptions.data;
    }
}

defaultOptions.setXhrFields(xhr);

xhr.open(defaultOptions.type, defaultOptions.url, defaultOptions.async,
defaultOptions.username, defaultOptions.password);
if(isArray(defaultOptions.accepts)){
    each(defaultOptions.accepts,function(x){
        xhr.setRequestHeader('Accepts',x);
    })
}
for (tempVal in defaultOptions.headers) {
    if (defaultOptions.headers.hasOwnProperty(tempVal))
        xhr.setRequestHeader(tempVal, defaultOptions.headers[tempVal]);
}
xhr.setRequestHeader("Content-Type", defaultOptions.contentType);

if (defaultOptions.context) {
    // 绑定宿主对象
    defaultOptions.complete =
```

```
defaultOptions.complete.bind(defaultOptions.context);
    defaultOptions.success =
defaultOptions.success.bind(defaultOptions.context);
    defaultOptions.error =
defaultOptions.error.bind(defaultOptions.context);
}

// 将 success、error、complete 方法包裹起来
var _success = function (text) {
    defaultOptions.success(text);
    _promise._done(text);
    defaultOptions.complete(xhr);
    _promise._always();
}, _error = function (text) {
    defaultOptions.error(xhr, xhr.status, text);
    _promise._fail(xhr, xhr.status, text);
    defaultOptions.complete(xhr);
    _promise._always();
};

// 检测 xhr 对象是否有 responseType，如果有则将 dataType 赋给它
('responseType' in xhr)&& (xhr.responseType = defaultOptions.dataType);
xhr.onreadystatechange = function () {
    if (this.readyState == 4) {
        if (defaultOptions.mimeType) {
            // 重写 MIME Type
            xhr.overrideMimeType(defaultOptions.mimeType);
        }
        // 当 http 状态码为 2xx 时执行 success，4xx 或 5xx 时执行 error
        if (/^2\d{2}$/.test(this.status)) {
            var returnVal = undefined;
            if (xhr.responseType) {
                returnVal = this.response;
            } else {
                // 数据处理
                switch (defaultOptions.dataType.toLowerCase()) {
                    case 'xml':
                        returnVal = this.responseXML;
                        break;
                    case 'html':
                        var frag = document.createDocumentFragment();
                        frag.innerHTML = this.responseText;
                        returnVal = frag;
                        break;
                    case 'script':
                        var scr = document.createElement("script");
                        scr.innerHTML = this.responseText;
                        returnVal = scr;
                }
            }
        }
    }
}
```

```

        break;
        case 'json':
            if (global.JSON.parse) {
                try {
                    returnVal =
JSON.parse(this.responseText);
                } catch (ex) {
                    _error(ex)
                }
            } else {
                returnVal = eval('(' + this.responseText +
''));
            }
            break;
        case 'arraybuffer':
            throw new ReferenceError('not supported
arraybuffer');

        case 'blob':
            throw new ReferenceError('not supported blob');
        default:
            returnVal = this.responseText;
        }
        _success(defaultOptions.dataFilter(returnVal)||returnVal);
    }
    if (/^(4|5)\d{2}$/.test(this.status)) {
        _error();
    }
    //根据http状态码执行statusCode中对应的函数
    (this.status.toString() in defaultOptions.statusCode) &&
defaultOptions.statusCode[this.status]();
    }
};
if (defaultOptions.beforeSend) {
    defaultOptions.beforeSend(xhr);
}
xhr.send(defaultOptions.data);
xhr.onerror = _error;
//超时设置，此方法有兼容性。所以执行前先做特性判断
if (isNumber(defaultOptions.timeout) && defaultOptions.timeout > 500) {
    if ('timeout' in xhr) {
        xhr.timeout = defaultOptions.timeout;
        xhr.ontimeout = defaultOptions.error;
    } else {
        setTimeout(function () {
            if (xhr.readyState != 4) {
                _error();
            }
        }, defaultOptions.timeout);
    }
}

```

```
    }
  }
  // 返回链式注册对象
  return _promise;
};
// 利用ajax方法生产get、post方法
each(['get', 'post'], function (x) {
  http[x] = function (url, data, func, datatype) {
    if (arguments.length !== 4)
      throw new TypeError('lacking arguments');
    return http.ajax({
      url: url,
      data: data,
      success: func,
      dataType: datatype,
      type: x
    })
  }
});
// 利用ajax生产getScript方法
http.getScript = function (url, data, func) {
  return http.ajax({
    url: url,
    type: 'get',
    data: data,
    success: func,
    dataType: 'script'
  })
};
// 链式调用对象
function __promise() {
  // 设置默认函数
  this._done = this._fail = this._always = function () {
  };
}

// 这三个方法执行完之后都必须将自己返回，否则无法链式注册
__promise.prototype.done = function (func) {
  this._done = func;
  return this;
};
__promise.prototype.fail = function (func) {
  this._fail = func;
  return this;
};
__promise.prototype.always = function (func) {
  this._always = func;
  return this;
};
```

```
//AMD, CommonJs, then globals
if (typeof define === 'function' && define.amd) {
    define([], function(){
        return http;
    });
} else if (typeof exports === 'object') {
    module.exports = http;
} else {
    global.$http = global.$http || http;
}
})(window);
```

如何使用上面的Ajax库

```
window.onload=function(){
    //假设本地有一台端口为1111的web服务器，且这个web服务器上有一个名为ajaxAPI的接口
    //可以这样使用
    $http.get('http://localhost:1111/ajaxAPI', 'arg=1', function(x){
        console.log(toString.call(x), x)
    }, 'json').done(function(){
        console.log('done')
    }).fail(function(e){
        console.error(e)
    }).always(function(){
        console.info('always');
    });
    //也可以这样使用
    $http.ajax({
        type: 'get',
        url: 'http://localhost:1111/ajaxAPI',
        context: {a: '123'},
        success: function(x){
            console.log(x, this.a);
        }
    });
    //... 你还可以根据ajax中提供的参数编写功能更加强大的函数
}
```

表单操作

考虑到HTML表单。当用户提交表单时，表单中的数据(每个表单元素的名字和值)编码到一个字符串中随请求发送。默认情况下，HTML表单通过POST方法发送给服务器，而编码之

后的表单数据则用做请求主体。对表单数据使用的编码方案相对简单：对每个表单元素的名字和值执行偶痛的URL编码(使用十六进制转义码替换特殊字符；编码利用 `encodeURIComponent()` 方法，解码利用 `decodeURIComponent()` 方法)，使用等号把编码后的名字和值分开，并使用 `&` 符号分开 名/值 对。一个简单表单的编码如下这样：

```
name=pizza&age=18&address=%E5%8C%97%E4%BA%AC
```

表单数据编码格式有一个正式的MIME类型

```
application/x-www-form-urlencoded
```

当使用 `post` 方法提交这顺序的表单数据时，必须设置 `Content-Type` 请求头为这个值。

注意：这种类型的编码并不需要HTML表单，在本章我们实际上将不需要直接使用表单。在Ajax应用中，你希望发给服务器的可能是个javascript对象。前面展示的数据变成javascript对象的表单编码形式可能为：

```
{
  name:pizza,
  age:18,
  address:'北京'
}
```

表单编码在web上如此广泛的使用，同时所有服务器端的编程语言都能很好的支持，所以非表单数据的表单编码也是容易实现的事情。如下代码：

```
function encodeFormData(data){
  if(!data) return '';
  var arr=[];
  for(var name in data){
    if(!data.hasOwnProperty(name)) continue;
    if(typeof data[name] === 'function') continue;
    var value = data[name] + '';
    name=encodeURIComponent(name);
    value=encodeURIComponent(value);
    arr.push(name + '=' + value);
  }
  return arr.join('&');
}
```

使用已定义的 `encodeFormData()` 函数，我们能容易的将javascript对象转化为表单格式的数据。

表单数据同样可以通过 `get` 请求来提交，既然表单提交的目的是为了执行只读查询，因此 `get` 请求比 `post` 更合适。 `get` 请求从来没有主体，所以需要发送给服务器的表单编码数据 负载 要

作为URL的查询(search)部分。

利用ajax模拟表单提交

在应用场景中，模拟表单提交是十分常见的。在HTML中form元素提交之后会重新加载页面，这个现象的副作用是很大的，需要重新加载整个页面的数据，所以更多的是通过ajax模拟表单提交数据，这样的话实现的功能是和html的表单提交一摸一样但是不需要重新加载页面。在实现模拟表单提交之前需要先实现一个表单序列化的帮助函数。代码如下：

```
http.serialize = function (form) {
    var parts = [], optValue = "";
    each(form.elements, function (ele) {
        switch (ele.type) {
            case 'select-one':
            case 'select-multiple':
                if (ele.name) {
                    each(ele.options, function (option) {
                        if (option.selected) {
                            if (option.hasAttribute) {
                                optValue = (option.hasAttribute("value") ?
                                option.value : option.text);
                            } else {
                                optValue =
                                (option.attributes["value"].specified ? option.value : option.text);
                            }
                            parts.push(encodeURIComponent(ele.name) + "=" +
                            encodeURIComponent(optValue));
                        }
                    })
                }
                break;
            case undefined:
            case 'submit':
            case 'reset':
            case 'button':
            case 'file':
                break;
            case 'radio':
            case 'checkbox':
                if (!ele.checked) {
                    break;
                }
            default :
                if (ele.name) {
                    parts.push(encodeURIComponent(ele.name) + "=" +
                    encodeURIComponent(ele.value));
                }
        }
    })
}
```

```
    }  
  }  
});  
return parts.join('&');  
}
```

// 给 表 单 注 册 onsubmit 事 件 。 当 提 交 时 就 序 列 化 表 单 数 据 并 发 送 给 服 务 器

```
document.forms[0].onsubmit=function(){  
  $http.get('/serialize',$http.serialize(this),function(res){  
    console.log(res);  
  });  
  // 阻 止 默 认 行 为  
  return false;  
};
```

上面这个serialize()函数首先定义了一个名为parts的数组，用于保存将要创建的字符串的各个部分。然后，通过for循环迭代每个表单字段，并将其保存在field变量中。在获得了一个字段的引用之后，使用switch语句检测其type属性。序列化过程最麻烦的就是 select 元素，它可能是单选框也可能是多选框，值可能有一个选中项，而多选框则可能有零或多个选中项。这里的代码适用于这两种选择框，至于可选框的数量是由浏览器控制的。在找到了一个选中项之后，需要确定使用什么值。如果不存在value特性，或者虽然存在该特性，但值为空字符串，要使用选项的文本代替。为检查这个特性，在DOM兼容的浏览器中需要使用hasAttribute()方法，而在IE中需要使用特性的specified属性。

如果表单中包含 fieldset 元素，则该元素会出现在元素集合中，但没有type属性。因此，如果type属性未定义，则不需要对其进行序列化。同样，对于各种按钮以及文件输入字段也是如此。对于单选按钮和复选框，要检查其checked属性是否被设置为false，如果是则退出switch语句。如果checked属性为ture，则继续执行default语句，即将当前字段的名称和值进行编码，然后添加到parts数组中。函数的最后一步，就是使用join()格式化整个字符串，也就是用和号来分割每一个表单字段。

最后，serialize()函数会以查询字符串的格式输出序列化之后的字符串。当然，要序列化成其它格式，也不是什么困难的事。

跨域请求操作

在应用场景中会经常出现不同源的数据请求，这种情况下需要做相应的处理操作。由于同源策略和浏览器兼容等问题的存在，所以处理不同来源的请求需要注意的地方有很多。下面会为大家列出相应的解决方案以及相应的概念普及。

同源策略

同源策略就是规定了javascript可以操作那些web内容的一个完整的安全限制。

什么是同源？

同源就是规定多个web资源的url中 `scheme`、`hostname`、`port` 必须相同，只要有一项不同那么这个web资源就不是同源的。同时，同源策略就会其相应的作用来限制这个web资源。

同源策略为什么会出现？

对于防止脚本窃取所有内容来说，同源策略是非常有必要的。如果没有这一个限制，恶意脚本可能会打开一个空页面，诱导用户进入并使用这个窗口在内网浏览操作文件。这样的话，恶意脚本就能够读取窗口内的内容发送到自己的服务器来达到窃取数据的目的。而同源策略就是限制了这种行为。

思考：web的安全性如何考虑？

跨域HTTP请求

什么是跨域？

当请求的资源的URL与当前页面的URL中的 `scheme`、`hostname`、`port` 有一个不同的时候就算是跨域操作。请参见上面的同源。

因为有同源策略的限制，XMLHttpRequest仅可以发起操作同域(同源)下的请求。虽然这个限制关闭了安全漏洞但是也阻止了大量合法的适合使用的跨域请求。不过这种情况下也可以在页面中使用 `img`、`form`、`iframe` 等元素中使用跨域URL，最中在浏览器中显示这些数据。但是因为同源策略，浏览器不允许操作或者不能良好的显示跨域文档内容。如果此处使用XMLHttpRequest来操作跨域请求，那么所有的文档内容都将在responseText属性中暴露，所以同源策略不允许XMLHttpRequest进行跨域请求。

注意：img 元素会把返回的内容强制转换为图片。iframe 元素不允许操作跨域数据。

但是需要强调的是 `script` 元素并未真正受到同源策略的限制，因为script有可能需要加载不同域的javascript资源。需要加载并执行任何来源的脚本。正因为如此，`script` 的灵活性使其成为在跨域操作中代替XMLHttpRequest的主流Ajax传输协议：`JSONP`。

JSONP

`script` 元素可以作为一种Ajax传输协议，只需设置 `script` 元素的src属性并且插入到DOM中，浏览器就会发出一个HTTP请求到src属性所指向的URL。使用 `script` 元素进行Ajax传输的一个主要原因就是因为它不受同源策略的影响。因此可以发送一个不同源的请求。而另外一个原因就是 `script` 元素会自动解码并执行(浏览器会当做javascript来处理)下载的数据。

JSONP带来的安全性考虑： 为了使用 `script` 元素来进行Ajax传输，你必须允许web页面信任并执行目标服务器返回过来的任何数据。这意味这对于不信任的服务器，不应该采取该技术。在与信任的服务器交互是还要提防攻击者可能会进入服务器中。所以作为Ajax数据传输的 `script` 与可信的服务器交互，是相当危险的事情。

使用这种 `script` 元素来进行Ajax数据的传输的技术就叫做'JSONP'，也就是 `JSON - padding` . 这个 `P (padding)`代表的是 填充、补充、前缀。在于服务器返回的数据必须用javascript的方法名加括号包裹住才行。而不是仅仅发送一段JSON格式的数据而已，要是这样的话浏览器只会对返回的数据进行JSON解码，结果还是数据，并没有做任何相应的处理。因此在使用JSONP的时候需要注意的是，服务器返回的数据是有固定格式的。例如：

```
// 服务器 不可以 返回 这样的 数据
["jeams","bond",{NAME:"OBAMA", AGE:56}]
// 服务器 会 返回 一个 这样的 响应
functionName(["jeams","bond",{NAME:"OBAMA", AGE:56}])
```

其中的functionName必须是在window下可以访问的名称。这样的话服务器就不仅仅只是返回一段JSON数据而已了，同时还会执行对应的操作。包裹后的响应会成为这个 `script` 元素的内容，它先判断JSON编码后的数据(因为是一个javascript表达式)，然后把数据传给functionName函数。此处我们可以假设functionName会那这些数据做有用的事情。但是为了可行起见，我们必须要把需要包裹数据的那个javascript方法名告诉服务器，也就是上面例子里的那个functionName。这样的话服务器就会知道用什么来包裹需要返回的数据了。服务器也可以知道返回的是一个JSONP数据而不是一个普通的JSON数据。例如可以在请求的URL后面加上 `?callback=functionName` 。

```

//实现一个简单的JSONP请求
//请求的url、包裹方法名称、回调函数
function JSONP(url,callbackName,callback){
    //为本次请求创建一个唯一的callback名称
    var cbnum="cb"+JSONP.count++; //计数器 生成一个唯一的名称
    var cbname="JSONP."+cbnum; //作为JSONP方法的一个静态属性

    if(url.indexOf("?")==-1){
        url+="?" +callbackName+"="+cbname;
    }else{
        url+="&" +callbackName+"="+cbname;
    }

    JSONP[cbnum]=function(response){
        try{
            callback(response);
        }catch (ex){

        }finally{
            //执行完毕之后就删掉，因为没什么用了
            delete JSONP[cbnum];
            script.parentNode.removeChild(script);
        }
    }
    var script=document.createElement("script");
    script.src=url;
    document.body.appendChild(script);
}

//初始化用于创建唯一名称的计数器
JSONP.count=0;

//发起JSONP请求。
JSONP("http://suggestion.baidu.com/su?wd=xxx","cb",function(data){
    //将百度返回的数据输出到控制台中
    console.log(data)
});

```

注意：当多次执行同一个url的JSONP操作时需要注意缓存问题。

可用的jsonp接口：

查询淘宝商品：

1:<http://suggest.taobao.com/sug?code=utf-8&q=商品关键字&callback=cb;>

快递查询：

2:<http://www.kuaidi100.com/query?type=quanfengkuaidi&postid=390011492112;>

(ps:快递公司编码:申通="shentong" EMS="ems" 顺丰="shunfeng" 圆通="yuantong" 中通="zhongtong" 韵达="yunda" 天天="tiantian" 汇通="huitongkuaidi" 全峰="quanfengkuaidi" 德邦="debangwuliu" 宅急送="zhaijisong")

天气查询：

3:http://php.weather.sina.com.cn/iframe/index/w_cl.php?code=js&day=0&city=&dfc=1&charset=utf-8;

手机号查询：

4:http://tcc.taobao.com/cc/json/mobile_tel_segment.htm?tel=手机号;

百度搜索：

5:<http://suggestion.baidu.com/su?wd=a&cb=xxx;>

跨域资源共享

由于浏览器的同源策略，限制了XMLHttpRequest的跨域请求的操作。但是在XHR2中浏览器选择允许发送合适的CORS(cross-origin resource sharing，跨域资源共享)来跨域请求数据。在标准浏览器中依旧使用XMLHttpRequest对象，而在IE8 - 9中则使用XDomainRequest对象来请求跨域资源。

虽然实现CORS不需要做任何事情，但是还有一些安全细节需要了解。首先，如果通过XMLHttpRequest的open()方法传入用户名和密码(详情见open方法)，那么它们绝不会通过跨域请求发送。另外跨域请求也不会包含其他任何的用户证书：cookie和HTTP身份认证的令牌(TOKEN)通常不会作为请求的内容发送到对方的服务器且对方服务器返回任何数据(cookie以及其他的一些响应头)都将被丢弃。如果跨域请求必须需要传入这几种用户证书才能成功，那么就必须在调用send()方法之前设置XMLHttpRequest的WithCredentials为true，此属性默认为false。也可以检索XMLHttpRequest对象有没有该属性来判断是否它支持CORS操作。

withCredentials属性

默认情况下，在浏览器中使用XMLHttpRequest进行跨源请求不提供凭据(cookie、HTTP认证及客户端SSL证明等)。通过将XMLHttpRequest的withCredentials属性设置为true，可以指定某个请求应该发送凭据。如果服务器接收带凭据的请求，会用下面的HTTP头部来响应。


```
// 服务器端返回此响应头
Access-Control-Allow-Credentials: true
//XMLHttpRequest的withCredentials设置为true
xhr.withCredentials=true;
```

如果发送的是带凭据的请求，但服务器的响应中没有包含这个头部，那么浏览器就不会把相应交给JavaScript(于是，`responseText`中将是空字符串，`status`的值为0，而且会调用`onerror()`事件处理程序)。另外，服务器还可以在响应中发送这个HTTP头部，表示允许源发送带凭据的请求。

使用XDomainRequest对象时需要注意的地方：

- 1、XDomainRequest对象没有`onreadystatechange`属性。
- 2、此对象只有IE8中有。
- 3、此方法只可使用http方案和https方案。
- 4、此对象还有一个特殊的`contentType`属性，用来获得响应头中的Content-Type。

当浏览器使用跨域资源共享时，不管是使用XMLHttpRequest还是XDomainRequest。服务器都必须在响应头中设置 `Access-Control-Allow-Origin` 。

```
// 在 java 或者 C# 中
<% Response.AddHeader("Access-Control-Allow-Origin","*") %>
//nodejs 中
response.writeHead(200,{"Access-Control-Allow-Origin":"*"})
```

其中 `*` 代码允许任何源请求本服务器，也可以改成固定的源。例如：`{"Access-Control-Allow-Origin":"http://localhost:63342"}` 只允许URL为 `http://localhost:63342` 的请求源请求本服务器。警告：如果将XMLHttpRequest的`withCredentials`属性设置为`true`的时候，`Access-Control-Allow-Origin` 这个响应头不可以设置为 `*` 。

W3C规定的跨域资源共享中服务器可以返回的头信息如下：

- Access-Control-Allow-Origin 使用格式：Access-Control-Allow-Origin = “Access-Control-Allow-Origin” “:” ascii-origin | “*”
- Access-Control-Max-Age 使用格式：Access-Control-Max-Age = “Access-Control-Max-Age” “:” delta-seconds
- Access-Control-Allow-Credentials 使用格式：Access-Control-Allow-Credentials: “Access-Control-Allow-Credentials” “:” “true”
- Access-Control-Allow-Methods 使用格式：Access-Control-Allow-Methods: “Access-Control-Allow-Methods” “:” #Method
- Access-Control-Allow-Headers 使用格式：Access-Control-Allow-Headers: “Access-Control-Allow-Headers” “:” #field-name
- Access-Control-Request-Method 使用格式：Access-Control-Request-Method: “Access-Control-Request-Method” “:” Method
- Access-Control-Request-Headers 使用格式：Access-Control-Request-Headers: “Access-Control-Request-Headers” “:” #field-name

```

(function (global, undefined) {
    // 根据 有无withCredentials来判断浏览器时候支持XMLHttpRequest跨域请求操作
    var XMLHttpRequestCORS = (function () {
        if (!('XMLHttpRequest' in global))
            return false;
        var a = new XMLHttpRequest();
        return a.withCredentials !== undefined;
    })(), request = function () {
        // 判断浏览器兼容性
        if ('XDomainRequest' in global)
            return new XDomainRequest();
        // 是否支持跨域请求
        if ('XMLHttpRequest' in global && XMLHttpRequestCORS)
            return new XMLHttpRequest();
        return false;
    };
    var xhr = request();
    if(xhr){
        xhr.open("get", "http://localhost:1111");
        xhr.onload = function () {
            //onload方法表示请求已经完成，参见上面XMLHttpRequest的onload属性解释
            console.log(this.responseText);
        };
        xhr.send();
    }

})(window);

```

结束语

现在你可能已经准备开始编写第一个 Ajax 应用程序了，不过可以首先从这些应用程序如何工作的基本概念开始，对 XMLHttpRequest 对象有基本的了解。

现在先花点儿时间考虑考虑 Ajax 应用程序有多么强大。设想一下，当单击按钮、输入一个字段、从组合框中选择一个选项或者用鼠标在屏幕上拖动时，Web 表单能够立刻作出响应会是什么情形。想一想异步 究竟意味着什么，想一想 JavaScript 代码运行而且不等待 服务器对它的请求作出响应。会遇到什么样的问题？会进入什么样的领域？考虑到这种新的方法，编程的时候应如何改变表单的设计？

如果在这些问题上花一点儿时间，与简单地剪切/粘贴某些代码到您根本不理解的应用程序中相比，收益会更多。

参考

- <http://bob.ippoli.to/archives/2005/12/05/remote-json-jsonp/>
- <http://www.w3.org/TR/cors/>
- <http://www.w3.org/TR/XMLHttpRequest/>
- [https://msdn.microsoft.com/zh-cn/library/ie/cc288060\(v=vs.85\).aspx](https://msdn.microsoft.com/zh-cn/library/ie/cc288060(v=vs.85).aspx)
- <https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest>
- <https://developer.mozilla.org/en-US/docs/Web/HTTP>

AUTHOR: 张亚涛

VERSION: BATE 0.1

NOTICE: 本文仅供珠峰培训学员内部学习参考使用