XMLHttpRequest 对象

简介

浏览器与服务器之间，采用 HTTP 协议通信。用户在浏览器地址栏键入一个网址，或者通过网页表单向服务器提交内容，这时浏览器就会向服务器发出 HTTP 请求。

1999年，微软公司发布 IE 浏览器5.0版，第一次引入新功能：允许 JavaScript 脚本向服务器发起 HTTP 请求。这个功能当时并没有引起注意，直到2004年 Gmail 发布和2005年 Google Map 发布，才引起广泛重视。2005年2月，AJAX 这个词第一次正式提出，它是 Asynchronous JavaScript and XML 的缩写，指的是通过 JavaScript 的异步通信，从服务器获取 XML 文档从中提取数据，再更新当前网页的对应部分，而不用刷新整个网页。后来，AJAX 这个词就成为 JavaScript 脚本发起 HTTP 通信的代名词，也就是说，只要用脚本发起通信，就可以叫做 AJAX 通信。W3C 也在2006年发布了它的国际标准。

具体来说，AJAX 包括以下几个步骤。

1. 创建 XMLHttpRequest 实例
1. 发出 HTTP 请求
1. 接收服务器传回的数据
1. 更新网页数据

概括起来，就是一句话，AJAX 通过原生的`XMLHttpRequest`对象发出 HTTP 请求，得到服务器返回的数据后，再进行处理。现在，服务器返回的都是 JSON 格式的数据，XML 格式已经过时了，但是 AJAX 这个名字已经成了一个通用名词，字面含义已经消失了。

`XMLHttpRequest`对象是 AJAX 的主要接口，用于浏览器与服务器之间的通信。尽管名字里面有`XML`和`Http`，它实际上可以使用多种协议（比如`file`或`ftp`），发送任何格式的数据（包括字符串和二进制）。

`XMLHttpRequest`本身是一个构造函数，可以使用`new`命令生成实例。它没有任何参数。

```javascript
var xhr = new XMLHttpRequest();
```

一旦新建实例，就可以使用`open()`方法指定建立 HTTP 连接的一些细节。

```javascript
xhr.open('GET', 'http://www.example.com/page.php', true);
```

上面代码指定使用 GET 方法，跟指定的服务器网址建立连接。第三个参数`true`，表示请求是异步的。

然后，指定回调函数，监听通信状态（`readyState`属性）的变化。

```javascript
xhr.onreadystatechange = handleStateChange;

function handleStateChange() {
  // ...
}
```

上面代码中，一旦`XMLHttpRequest`实例的状态发生变化，就会调用监听函数`handleStateChange`

最后使用`send()`方法，实际发出请求。

```javascript
xhr.send(null);
```

上面代码中，`send()`的参数为`null`，表示发送请求的时候，不带有数据体。如果发送的是 POST 请求，这里就需要指定数据体。

一旦拿到服务器返回的数据，AJAX 不会刷新整个网页，而是只更新网页里面的相关部分，从而不打断用户正在做的事情。

注意，AJAX 只能向同源网址（协议、域名、端口都相同）发出 HTTP 请求，如果发出跨域请求，就会报错（详见《同源政策》和《CORS 通信》两章）。

下面是`XMLHttpRequest`对象简单用法的完整例子。

```javascript
var xhr = new XMLHttpRequest();

xhr.onreadystatechange = function(){
  // 通信成功时，状态值为4
  if (xhr.readyState === 4){
    if (xhr.status === 200){
      console.log(xhr.responseText);
    } else {
      console.error(xhr.statusText);
    }
  }
};

xhr.onerror = function (e) {
  console.error(xhr.statusText);
};

xhr.open('GET', '/endpoint', true);
```

```
xhr.send(null);
```

XMLHttpRequest 的实例属性

XMLHttpRequest.readyState

`XMLHttpRequest.readyState`返回一个整数，表示实例对象的当前状态。该属性只读。它可能返回以下值。

- 0，表示 XMLHttpRequest 实例已经生成，但是实例的`open()`方法还没有被调用。
- 1，表示`open()`方法已经调用，但是实例的`send()`方法还没有调用，仍然可以使用实例的`setRequestHeader()`方法，设定 HTTP 请求的头信息。
- 2，表示实例的`send()`方法已经调用，并且服务器返回的头信息和状态码已经收到。
- 3，表示正在接收服务器传来的数据体（body 部分）。这时，如果实例的`responseType`属性等于`text`或者空字符串，`responseText`属性就会包含已经收到的部分信息。
- 4，表示服务器返回的数据已经完全接收，或者本次接收已经失败。

通信过程中，每当实例对象发生状态变化，它的`readyState`属性的值就会改变。这个值每一次变化，都会触发`readyStateChange`事件。

```javascript
var xhr = new XMLHttpRequest();

if (xhr.readyState === 4) {
  // 请求结束，处理服务器返回的数据
} else {
  // 显示提示“加载中……”
}
```

上面代码中，`xhr.readyState`等于`4`时，表明脚本发出的 HTTP 请求已经完成。其他情况，都表示 HTTP 请求还在进行中。

XMLHttpRequest.onreadystatechange

`XMLHttpRequest.onreadystatechange`属性指向一个监听函数。`readystatechange`事件发生时（实例的`readyState`属性变化），就会执行这个属性。

另外，如果使用实例的`abort()`方法，终止 XMLHttpRequest 请求，也会造成`readyState`属性变化，导致调用`XMLHttpRequest.onreadystatechange`属性。

下面是一个例子。

```javascript
var xhr = new XMLHttpRequest();
```

```
xhr.open( 'GET', 'http://example.com' , true );
xhr.onreadystatechange = function () {
  if (xhr.readyState !== 4 || xhr.status !== 200) {
    return;
  }
  console.log(xhr.responseText);
};
xhr.send();
```

XMLHttpRequest.response

`XMLHttpRequest.response`属性表示服务器返回的数据体（即 HTTP 回应的 body 部分）。它可能是任何数据类型，比如字符串、对象、二进制对象等等，具体的类型由`XMLHttpRequest.responseType`属性决定。该属性只读。

如果本次请求没有成功或者数据不完整，该属性等于`null`。但是，如果`responseType`属性等于`text`或空字符串，在请求没有结束之前（`readyState`等于3的阶段），`response`属性包含服务器已经返回的部分数据。

```javascript
var xhr = new XMLHttpRequest();

xhr.onreadystatechange = function () {
  if (xhr.readyState === 4) {
    handler(xhr.response);
  }
}
```

XMLHttpRequest.responseType

`XMLHttpRequest.responseType`属性是一个字符串，表示服务器返回数据的类型。这个属性是可写的，可以在调用`open()`方法之后、调用`send()`方法之前，设置这个属性的值，告诉服务器返回指定类型的数据。如果`responseType`设为空字符串，就等同于默认值`text`。

`XMLHttpRequest.responseType`属性可以等于以下值。

- ""（空字符串）：等同于`text`，表示服务器返回文本数据。
- "arraybuffer"：ArrayBuffer 对象，表示服务器返回二进制数组。
- "blob"：Blob 对象，表示服务器返回二进制对象。
- "document"：Document 对象，表示服务器返回一个文档对象。
- "json"：JSON 对象。
- "text"：字符串。

上面几种类型之中，`text`类型适合大多数情况，而且直接处理文本也比较方便。`document`类型适合返回 HTML / XML 文档的情况，这意味着，对于那些打开 CORS 的网站，可以直接用 Ajax

抓取网页，然后不用解析 HTML 字符串，直接对抓取回来的数据进行 DOM 操作。`blob`类型适合读取二进制数据，比如图片文件。

```javascript
var xhr = new XMLHttpRequest();
xhr.open('GET', '/path/to/image.png', true);
xhr.responseType = 'blob';

xhr.onload = function(e) {
  if (this.status === 200) {
    var blob = new Blob([xhr.response], {type: 'image/png'});
    // 或者
    var blob = xhr.response;
  }
};

xhr.send();
```

如果将这个属性设为`ArrayBuffer`，就可以按照数组的方式处理二进制数据。

```javascript
var xhr = new XMLHttpRequest();
xhr.open('GET', '/path/to/image.png', true);
xhr.responseType = 'arraybuffer';

xhr.onload = function(e) {
  var uInt8Array = new Uint8Array(this.response);
  for (var i = 0, len = uInt8Array.length; i < len; ++i) {
    // var byte = uInt8Array[i];
  }
};

xhr.send();
```

如果将这个属性设为`json`，浏览器就会自动对返回数据调用`JSON.parse()`方法。也就是说，从`xhr.response`属性（注意，不是`xhr.responseText`属性）得到的不是文本，而是一个 JSON 对象。

XMLHttpRequest.responseText

`XMLHttpRequest.responseText`属性返回从服务器接收到的字符串，该属性为只读。只有 HTTP 请求完成接收以后，该属性才会包含完整的数据。

```javascript
var xhr = new XMLHttpRequest();
xhr.open('GET', '/server', true);

xhr.responseType = 'text';
```

```
xhr.onload = function () {
  if (xhr.readyState === 4 && xhr.status === 200) {
    console.log(xhr.responseText);
  }
};

xhr.send(null);
```

XMLHttpRequest.responseXML

`XMLHttpRequest.responseXML`属性返回从服务器接收到的 HTML 或 XML 文档对象，该属性为只读。如果本次请求没有成功，或者收到的数据不能被解析为 XML 或 HTML，该属性等于`null`。

该属性生效的前提是 HTTP 回应的`Content-Type`头信息等于`text/xml`或`application/xml`。这要求在发送请求前，`XMLHttpRequest.responseType`属性要设为`document`。如果 HTTP 回应的`Content-Type`头信息不等于`text/xml`和`application/xml`，但是想从`responseXML`拿到数据（即把数据按照 DOM 格式解析），那么需要手动调用`XMLHttpRequest.overrideMimeType()`方法，强制进行 XML 解析。

该属性得到的数据，是直接解析后的文档 DOM 树。

```javascript
var xhr = new XMLHttpRequest();
xhr.open('GET', '/server', true);

xhr.responseType = 'document';
xhr.overrideMimeType('text/xml');

xhr.onload = function () {
  if (xhr.readyState === 4 && xhr.status === 200) {
    console.log(xhr.responseXML);
  }
};

xhr.send(null);
```

XMLHttpRequest.responseURL

`XMLHttpRequest.responseURL`属性是字符串，表示发送数据的服务器的网址。

```javascript
var xhr = new XMLHttpRequest();
xhr.open('GET', 'http://example.com/test', true);
xhr.onload = function () {
  // 返回 http://example.com/test
  console.log(xhr.responseURL);
};
```

xhr.send(null);
```

注意，这个属性的值与`open()`方法指定的请求网址不一定相同。如果服务器端发生跳转，这个属性返回最后实际返回数据的网址。另外，如果原始 URL 包括锚点（fragment），该属性会把锚点剥离。

### XMLHttpRequest.status，XMLHttpRequest.statusText

`XMLHttpRequest.status`属性返回一个整数，表示服务器回应的 HTTP 状态码。一般来说，如果通信成功的话，这个状态码是200；如果服务器没有返回状态码，那么这个属性默认是200。请求发出之前，该属性为`0`。该属性只读。

- 200, OK，访问正常
- 301, Moved Permanently，永久移动
- 302, Moved temporarily，暂时移动
- 304, Not Modified，未修改
- 307, Temporary Redirect，暂时重定向
- 401, Unauthorized，未授权
- 403, Forbidden，禁止访问
- 404, Not Found，未发现指定网址
- 500, Internal Server Error，服务器发生错误

基本上，只有2xx和304的状态码，表示服务器返回是正常状态。

```javascript
if (xhr.readyState === 4) {
  if ( (xhr.status >= 200 && xhr.status < 300)
    || (xhr.status === 304) ) {
    // 处理服务器的返回数据
  } else {
    // 出错
  }
}
```

`XMLHttpRequest.statusText`属性返回一个字符串，表示服务器发送的状态提示。不同于`status`属性，该属性包含整个状态信息，比如“OK”和“Not Found”。在请求发送之前（即调用`open()`方法之前），该属性的值是空字符串；如果服务器没有返回状态提示，该属性的值默认为“OK”。该属性为只读属性。

XMLHttpRequest.timeout，XMLHttpRequestEventTarget.ontimeout
```

`XMLHttpRequest.timeout`属性返回一个整数，表示多少毫秒后，如果请求仍然没有得到结果，就会自动终止。如果该属性等于0，就表示没有时间限制。

`XMLHttpRequestEventTarget.ontimeout`属性用于设置一个监听函数，如果发生 timeout 事件，就会执行这个监听函数。

下面是一个例子。

```javascript
var xhr = new XMLHttpRequest();
var url = '/server';

xhr.ontimeout = function () {
  console.error('The request for ' + url + ' timed out.');
};

xhr.onload = function() {
  if (xhr.readyState === 4) {
    if (xhr.status === 200) {
      // 处理服务器返回的数据
    } else {
      console.error(xhr.statusText);
    }
  }
};

xhr.open('GET', url, true);
// 指定 10 秒钟超时
xhr.timeout = 10 * 1000;
xhr.send(null);
```

事件监听属性

XMLHttpRequest 对象可以对以下事件指定监听函数。

- XMLHttpRequest.onloadstart：loadstart 事件（HTTP 请求发出）的监听函数
- XMLHttpRequest.onprogress：progress事件（正在发送和加载数据）的监听函数
- XMLHttpRequest.onabort：abort 事件（请求中止，比如用户调用了`abort()`方法）的监听函数
- XMLHttpRequest.onerror：error 事件（请求失败）的监听函数
- XMLHttpRequest.onload：load 事件（请求成功完成）的监听函数
- XMLHttpRequest.ontimeout：timeout 事件（用户指定的时限超过了，请求还未完成）的监听函数
- XMLHttpRequest.onloadend：loadend 事件（请求完成，不管成功或失败）的监听函数

下面是一个例子。

```javascript
xhr.onload = function() {
 var responseText = xhr.responseText;
 console.log(responseText);
 // process the response.
};

xhr.onabort = function () {
  console.log('The request was aborted');
};

xhr.onprogress = function (event) {
  console.log(event.loaded);
  console.log(event.total);
};

xhr.onerror = function() {
  console.log('There was an error!');
};
```

`progress`事件的监听函数有一个事件对象参数，该对象有三个属性：`loaded`属性返回已经传输的数据量，`total`属性返回总的数据量，`lengthComputable`属性返回一个布尔值，表示加载的进度是否可以计算。所有这些监听函数里面，只有`progress`事件的监听函数有参数，其他函数都没有参数。

注意，如果发生网络错误（比如服务器无法连通），`onerror`事件无法获取报错信息。也就是说，可能没有错误对象，所以这样只能显示报错的提示。

XMLHttpRequest.withCredentials

`XMLHttpRequest.withCredentials`属性是一个布尔值，表示跨域请求时，用户信息（比如 Cookie 和认证的 HTTP 头信息）是否会包含在请求之中，默认为`false`，即向`example.com`发出跨域请求时，不会发送`example.com`设置在本机上的 Cookie（如果有的话）。

如果需要跨域 AJAX 请求发送 Cookie，需要`withCredentials`属性设为`true`。注意，同源的请求不需要设置这个属性。

```javascript
var xhr = new XMLHttpRequest();
xhr.open('GET', 'http://example.com/', true);
xhr.withCredentials = true;
xhr.send(null);
```

为了让这个属性生效，服务器必须显式返回`Access-Control-Allow-Credentials`这个头信息。

```javascript
Access-Control-Allow-Credentials: true
```

```

`withCredentials`属性打开的话，跨域请求不仅会发送 Cookie，还会设置远程主机指定的 Cookie。反之也成立，如果`withCredentials`属性没有打开，那么跨域的 AJAX 请求即使明确要求浏览器设置 Cookie，浏览器也会忽略。

注意，脚本总是遵守同源政策，无法从`document.cookie`或者 HTTP 回应的头信息之中，读取跨域的 Cookie，`withCredentials`属性不影响这一点。

### XMLHttpRequest.upload

XMLHttpRequest 不仅可以发送请求，还可以发送文件，这就是 AJAX 文件上传。发送文件以后，通过`XMLHttpRequest.upload`属性可以得到一个对象，通过观察这个对象，可以得知上传的进展。主要方法就是监听这个对象的各种事件：loadstart、loadend、load、abort、error、progress、timeout。

假定网页上有一个`<progress>`元素。

```http
<progress min="0" max="100" value="0">0% complete</progress>
```

文件上传时，对`upload`属性指定`progress`事件的监听函数，即可获得上传的进度。

```javascript
function upload(blobOrFile) {
  var xhr = new XMLHttpRequest();
  xhr.open('POST', '/server', true);
  xhr.onload = function (e) {};

  var progressBar = document.querySelector('progress');
  xhr.upload.onprogress = function (e) {
    if (e.lengthComputable) {
      progressBar.value = (e.loaded / e.total) * 100;
      // 兼容不支持 <progress> 元素的老式浏览器
      progressBar.textContent = progressBar.value;
    }
  };

  xhr.send(blobOrFile);
}

upload(new Blob(['hello world'], {type: 'text/plain'}));
```

XMLHttpRequest 的实例方法

XMLHttpRequest.open()
```

`XMLHttpRequest.open()`方法用于指定 HTTP 请求的参数，或者说初始化 XMLHttpRequest 实例对象。它一共可以接受五个参数。

```javascript
void open(
   string method,
   string url,
   optional boolean async,
   optional string user,
   optional string password
);
```

- `method`：表示 HTTP 动词方法，比如`GET`、`POST`、`PUT`、`DELETE`、`HEAD`等。
- `url`：表示请求发送目标 URL。
- `async`：布尔值，表示请求是否为异步，默认为`true`。如果设为`false`，则`send()`方法只有等到收到服务器返回了结果，才会进行下一步操作。该参数可选。由于同步 AJAX 请求会造成浏览器失去响应，许多浏览器已经禁止在主线程使用，只允许 Worker 里面使用。所以，这个参数轻易不应该设为`false`。
- `user`：表示用于认证的用户名，默认为空字符串。该参数可选。
- `password`：表示用于认证的密码，默认为空字符串。该参数可选。

注意，如果对使用过`open()`方法的 AJAX 请求，再次使用这个方法，等同于调用`abort()`，即终止请求。

下面发送 POST 请求的例子。

```javascript
var xhr = new XMLHttpRequest();
xhr.open('POST', encodeURI('someURL'));
```

XMLHttpRequest.send()

`XMLHttpRequest.send()`方法用于实际发出 HTTP 请求。它的参数是可选的，如果不带参数，就表示 HTTP 请求只有一个 URL，没有数据体，典型例子就是 GET 请求；如果带有参数，就表示除了头信息，还带有包含具体数据的信息体，典型例子就是 POST 请求。

下面是 GET 请求的例子。

```javascript
var xhr = new XMLHttpRequest();
xhr.open('GET',
  'http://www.example.com/?id=' + encodeURIComponent(id),
  true
);
xhr.send(null);
```

```

上面代码中，`GET`请求的参数，作为查询字符串附加在 URL 后面。

下面是发送 POST 请求的例子。

```javascript
var xhr = new XMLHttpRequest();
var data = 'email='
  + encodeURIComponent(email)
  + '&password='
  + encodeURIComponent(password);

xhr.open('POST', 'http://www.example.com', true);
xhr.setRequestHeader('Content-Type', 'application/x-www-form-urlencoded');
xhr.send(data);
```

注意，所有 XMLHttpRequest 的监听事件，都必须在`send()`方法调用之前设定。

`send`方法的参数就是发送的数据。多种格式的数据，都可以作为它的参数。

```javascript
void send();
void send(ArrayBufferView data);
void send(Blob data);
void send(Document data);
void send(String data);
void send(FormData data);
```

如果`send()`发送 DOM 对象，在发送之前，数据会先被串行化。如果发送二进制数据，最好是发送`ArrayBufferView`或`Blob`对象，这使得通过 Ajax 上传文件成为可能。

下面是发送表单数据的例子。`FormData`对象可以用于构造表单数据。

```javascript
var formData = new FormData();

formData.append('username', '张三');
formData.append('email', 'zhangsan@example.com');
formData.append('birthDate', 1940);

var xhr = new XMLHttpRequest();
xhr.open('POST', '/register');
xhr.send(formData);
```

上面代码中，`FormData`对象构造了表单数据，然后使用`send()`方法发送。它的效果与发送下面的表单数据是一样的。
```

```html
<form id='registration' name='registration' action='/register'>
  <input type='text' name='username' value='张三'>
  <input type='email' name='email' value='zhangsan@example.com'>
  <input type='number' name='birthDate' value='1940'>
  <input type='submit' onclick='return sendForm(this.form);'>
</form>
```

下面的例子是使用`FormData`对象加工表单数据，然后再发送。

```javascript
function sendForm(form) {
  var formData = new FormData(form);
  formData.append('csrf', 'e69a18d7db1286040586e6da1950128c');

  var xhr = new XMLHttpRequest();
  xhr.open('POST', form.action, true);
  xhr.onload = function() {
    // ...
  };
  xhr.send(formData);

  return false;
}

var form = document.querySelector('#registration');
sendForm(form);
```

XMLHttpRequest.setRequestHeader()

`XMLHttpRequest.setRequestHeader()`方法用于设置浏览器发送的 HTTP 请求的头信息。该方法必须在`open()`之后、`send()`之前调用。如果该方法多次调用，设定同一个字段，则每一次调用的值会被合并成一个单一的值发送。

该方法接受两个参数。第一个参数是字符串，表示头信息的字段名，第二个参数是字段值。

```javascript
xhr.setRequestHeader('Content-Type', 'application/json');
xhr.setRequestHeader('Content-Length', JSON.stringify(data).length);
xhr.send(JSON.stringify(data));
```

上面代码首先设置头信息`Content-Type`，表示发送 JSON 格式的数据；然后设置`Content-Length`，表示数据长度；最后发送 JSON 数据。

XMLHttpRequest.overrideMimeType()

`XMLHttpRequest.overrideMimeType()`方法用来指定 MIME 类型，覆盖服务器返回的真正的 MIME 类型，从而让浏览器进行不一样的处理。举例来说，服务器返回的数据类型是`text/xml`，由于种种原因浏览器解析不成功报错，这时就拿不到数据了。为了拿到原始数据，我们可以把 MIME 类型改成`text/plain`，这样浏览器就不会去自动解析，从而我们就可以拿到原始文本了。

```javascript
xhr.overrideMimeType('text/plain')
```

注意，该方法必须在`send()`方法之前调用。

修改服务器返回的数据类型，不是正常情况下应该采取的方法。如果希望服务器返回指定的数据类型，可以用`responseType`属性告诉服务器，就像下面的例子。只有在服务器无法返回某种数据类型时，才使用`overrideMimeType()`方法。

```javascript
var xhr = new XMLHttpRequest();
xhr.onload = function(e) {
  var arraybuffer = xhr.response;
  // ...
}
xhr.open('GET', url);
xhr.responseType = 'arraybuffer';
xhr.send();
```

XMLHttpRequest.getResponseHeader()

`XMLHttpRequest.getResponseHeader()`方法返回 HTTP 头信息指定字段的值，如果还没有收到服务器回应或者指定字段不存在，返回`null`。该方法的参数不区分大小写。

```javascript
function getHeaderTime() {
  console.log(this.getResponseHeader("Last-Modified"));
}

var xhr = new XMLHttpRequest();
xhr.open('HEAD', 'yourpage.html');
xhr.onload = getHeaderTime;
xhr.send();
```

如果有多个字段同名，它们的值会被连接为一个字符串，每个字段之间使用“逗号+空格”分隔。

XMLHttpRequest.getAllResponseHeaders()

`XMLHttpRequest.getAllResponseHeaders()`方法返回一个字符串，表示服务器发来的所有 HTTP 头信息。格式为字符串，每个头信息之间使用`CRLF`分隔（回车+换行），如果没有收到服务器回应，该属性为`null`。如果发生网络错误，该属性为空字符串。

```javascript
var xhr = new XMLHttpRequest();
xhr.open('GET', 'foo.txt', true);
xhr.send();

xhr.onreadystatechange = function () {
  if (this.readyState === 4) {
    var headers = xhr.getAllResponseHeaders();
  }
}
```

上面代码用于获取服务器返回的所有头信息。它可能是下面这样的字符串。

```http
date: Fri, 08 Dec 2017 21:04:30 GMT\r\n
content-encoding: gzip\r\n
x-content-type-options: nosniff\r\n
server: meinheld/0.6.1\r\n
x-frame-options: DENY\r\n
content-type: text/html; charset=utf-8\r\n
connection: keep-alive\r\n
strict-transport-security: max-age=63072000\r\n
vary: Cookie, Accept-Encoding\r\n
content-length: 6502\r\n
x-xss-protection: 1; mode=block\r\n
```

然后，对这个字符串进行处理。

```javascript
var arr = headers.trim().split(/[\r\n]+/);
var headerMap = {};

arr.forEach(function (line) {
  var parts = line.split(': ');
  var header = parts.shift();
  var value = parts.join(': ');
  headerMap[header] = value;
});

headerMap['content-length'] // "6502"
```

XMLHttpRequest.abort()

`XMLHttpRequest.abort()`方法用来终止已经发出的 HTTP 请求。调用这个方法以后，`readyState`属性变为`4`，`status`属性变为`0`。

```javascript
var xhr = new XMLHttpRequest();
```

```
xhr.open('GET', 'http://www.example.com/page.php', true);
setTimeout(function () {
  if (xhr) {
    xhr.abort();
    xhr = null;
  }
}, 5000);
```

上面代码在发出5秒之后，终止一个 AJAX 请求。

XMLHttpRequest 实例的事件

readyStateChange 事件

`readyState`属性的值发生改变，就会触发 readyStateChange 事件。

我们可以通过`onReadyStateChange`属性，指定这个事件的监听函数，对不同状态进行不同处理。尤其是当状态变为`4`的时候，表示通信成功，这时回调函数就可以处理服务器传送回来的数据。

progress 事件

上传文件时，XMLHttpRequest 实例对象本身和实例的`upload`属性，都有一个`progress`事件，会不断返回上传的进度。

```javascript
var xhr = new XMLHttpRequest();

function updateProgress (oEvent) {
  if (oEvent.lengthComputable) {
    var percentComplete = oEvent.loaded / oEvent.total;
  } else {
    console.log('无法计算进展');
  }
}

xhr.addEventListener('progress', updateProgress);

xhr.open();
```

load 事件、error 事件、abort 事件

load 事件表示服务器传来的数据接收完毕，error 事件表示请求出错，abort 事件表示请求被中断（比如用户取消请求）。

```javascript
var xhr = new XMLHttpRequest();
```

```
xhr.addEventListener('load', transferComplete);
xhr.addEventListener('error', transferFailed);
xhr.addEventListener('abort', transferCanceled);

xhr.open();

function transferComplete() {
  console.log('数据接收完毕');
}

function transferFailed() {
  console.log('数据接收出错');
}

function transferCanceled() {
  console.log('用户取消接收');
}
```

loadend 事件

`abort`、`load`和`error`这三个事件，会伴随一个`loadend`事件，表示请求结束，但不知道其是否成功。

```javascript
xhr.addEventListener('loadend', loadEnd);

function loadEnd(e) {
  console.log('请求结束，状态未知');
}
```

timeout 事件

服务器超过指定时间还没有返回结果，就会触发 timeout 事件，具体的例子参见`timeout`属性一节。

Navigator.sendBeacon()

用户卸载网页的时候，有时需要向服务器发一些数据。很自然的做法是在`unload`事件或`beforeunload`事件的监听函数里面，使用`XMLHttpRequest`对象发送数据。但是，这样做不是很可靠，因为`XMLHttpRequest`对象是异步发送，很可能在它即将发送的时候，页面已经卸载了，从而导致发送取消或者发送失败。

解决方法就是`unload`事件里面，加一些很耗时的同步操作。这样就能留出足够的时间，保证异步 AJAX 能够发送成功。

```javascript
function log() {
  let xhr = new XMLHttpRequest();
  xhr.open('post', '/log', true);
  xhr.setRequestHeader('Content-Type', 'application/x-www-form-urlencoded');
  xhr.send('foo=bar');
}

window.addEventListener('unload', function(event) {
  log();

  // a time-consuming operation
  for (let i = 1; i < 10000; i++) {
    for (let m = 1; m < 10000; m++) { continue; }
  }
});
```

上面代码中，强制执行了一次双重循环，拖长了`unload`事件的执行时间，导致异步 AJAX 能够发送成功。

类似的还可以使用`setTimeout`。下面是追踪用户点击的例子。

```javascript
// HTML 代码如下
// <a id="target" href="https://baidu.com">click</a>
const clickTime = 350;
const theLink = document.getElementById('target');

function log() {
  let xhr = new XMLHttpRequest();
  xhr.open('post', '/log', true);
  xhr.setRequestHeader('Content-Type', 'application/x-www-form-urlencoded');
  xhr.send('foo=bar');
}

theLink.addEventListener('click', function (event) {
  event.preventDefault();
  log();

  setTimeout(function () {
    window.location.href = theLink.getAttribute('href');
  }, clickTime);
});
```

上面代码使用`setTimeout`，拖延了350毫秒，才让页面跳转，因此使得异步 AJAX 有时间发出。

这些做法的共同问题是，卸载的时间被硬生生拖长了，后面页面的加载被推迟了，用户体验不好。

为了解决这个问题，浏览器引入了`Navigator.sendBeacon()`方法。这个方法还是异步发出请求，但是请求与当前页面线程脱钩，作为浏览器进程的任务，因此可以保证会把数据发出去，不拖延卸载流程。

```javascript
window.addEventListener('unload', logData, false);

function logData() {
  navigator.sendBeacon('/log', analyticsData);
}
```

`Navigator.sendBeacon`方法接受两个参数，第一个参数是目标服务器的 URL，第二个参数是所要发送的数据（可选），可以是任意类型（字符串、表单对象、二进制对象等等）。

```javascript
navigator.sendBeacon(url, data)
```

这个方法的返回值是一个布尔值，成功发送数据为`true`，否则为`false`。

该方法发送数据的 HTTP 方法是 POST，可以跨域，类似于表单提交数据。它不能指定回调函数。

下面是一个例子。

```javascript
// HTML 代码如下
// <body onload="analytics('start')" onunload="analytics('end')">

function analytics(state) {
  if (!navigator.sendBeacon) return;

  var URL = 'http://example.com/analytics';
  var data = 'state=' + state + '&location=' + window.location;
  navigator.sendBeacon(URL, data);
}
```