```
function submitData() {
  let xhr = new XMLHttpRequest();
  xhr.onreadystatechange = function() {
    if (xhr.readyState == 4) {
      if ((xhr.status >= 200 && xhr.status < 300) || xhr.status == 304) {
        alert(xhr.responseText);
      } else {
        alert("Request was unsuccessful: " + xhr.status);
      }
    }
  };

  xhr.open("post", "postexample.php", true);
  xhr.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
  let form = document.getElementById("user-info");
  xhr.send(serialize(form));
}
```

在这个函数中，来自 ID 为"user-info"的表单中的数据被序列化之后发送给了服务器。PHP 文件 postexample.php 随后可以通过$_POST 取得 POST 的数据。比如：

```
<?php
  header("Content-Type: text/plain");
  echo <<<EOF
Name: {$_POST['user-name']}
Email: {$_POST['user-email']}
EOF;
?>
```

假如没有发送 Content-Type 头部，PHP 的全局$_POST 变量中就不会包含数据，而需要通过 $HTTP_RAW_POST_DATA 来获取。

> **注意**　POST 请求相比 GET 请求要占用更多资源。从性能方面说，发送相同数量的数据，GET 请求比 POST 请求要快两倍。

24.1.5　XMLHttpRequest Level 2

XHR 对象作为事实标准的迅速流行，也促使 W3C 为规范这一行为而制定了正式标准。XMLHttpRequest Level 1 只是把已经存在的 XHR 对象的实现细节明确了一下。XMLHttpRequest Level 2 又进一步发展了 XHR 对象。并非所有浏览器都实现了 XMLHttpRequest Level 2 的所有部分，但所有浏览器都实现了其中部分功能。

1. FormData 类型

现代 Web 应用程序中经常需要对表单数据进行序列化，因此 XMLHttpRequest Level 2 新增了 FormData 类型。FormData 类型便于表单序列化，也便于创建与表单类似格式的数据然后通过 XHR 发送。下面的代码创建了一个 FormData 对象，并填充了一些数据：

```
let data = new FormData();
data.append("name", "Nicholas");
```

append()方法接收两个参数：键和值，相当于表单字段名称和该字段的值。可以像这样添加任意多个键/值对数据。此外，通过直接给 FormData 构造函数传入一个表单元素，也可以将表单中的数据

作为键/值对填充进去：

```
let data = new FormData(document.forms[0]);
```

有了 FormData 实例，可以像下面这样直接传给 XHR 对象的 send()方法：

```
let xhr = new XMLHttpRequest();
xhr.onreadystatechange = function() {
  if (xhr.readyState == 4) {
    if ((xhr.status >= 200 && xhr.status < 300) || xhr.status == 304) {
      alert(xhr.responseText);
    } else {
      alert("Request was unsuccessful: " + xhr.status);
    }
  }
};
xhr.open("post", "postexample.php", true);
let form = document.getElementById("user-info");
xhr.send(new FormData(form));
```

使用 FormData 的另一个方便之处是不再需要给 XHR 对象显式设置任何请求头部了。XHR 对象能够识别作为 FormData 实例传入的数据类型并自动配置相应的头部。

2. 超时

IE8 给 XHR 对象增加了一个 timeout 属性，用于表示发送请求后等待多少毫秒，如果响应不成功就中断请求。之后所有浏览器都在自己的 XHR 实现中增加了这个属性。在给 timeout 属性设置了一个时间且在该时间过后没有收到响应时，XHR 对象就会触发 timeout 事件，调用 ontimeout 事件处理程序。这个特性后来也被添加到了 XMLHttpRequest Level 2 规范。下面看一个例子：

```
let xhr = new XMLHttpRequest();
xhr.onreadystatechange = function() {
  if (xhr.readyState == 4) {
    try {
      if ((xhr.status >= 200 && xhr.status < 300) || xhr.status == 304) {
        alert(xhr.responseText);
      } else {
        alert("Request was unsuccessful: " + xhr.status);
      }
    } catch (ex) {
      // 假设由 ontimeout 处理
    }
  }
};

xhr.open("get", "timeout.php", true);
xhr.timeout = 1000; // 设置 1 秒超时
xhr.ontimeout = function() {
  alert("Request did not return in a second.");
};
xhr.send(null);
```

这个例子演示了使用 timeout 设置超时。给 timeout 设置 1000 毫秒意味着，如果请求没有在 1 秒钟内返回则会中断。此时则会触发 ontimeout 事件处理程序，readyState 仍然会变成 4，因此也会调用 onreadystatechange 事件处理程序。不过，如果在超时之后访问 status 属性则会发生错误。为做好防护，可以把检查 status 属性的代码封装在 try/catch 语句中。

3. overrideMimeType()方法

Firefox 首先引入了 overrideMimeType()方法用于重写 XHR 响应的 MIME 类型。这个特性后来也被添加到了 XMLHttpRequest Level 2。因为响应返回的 MIME 类型决定了 XHR 对象如何处理响应，所以如果有办法覆盖服务器返回的类型，那么是有帮助的。

假设服务器实际发送了 XML 数据，但响应头设置的 MIME 类型是 text/plain。结果就会导致虽然数据是 XML，但 responseXML 属性值是 null。此时调用 overrideMimeType()可以保证将响应当成 XML 而不是纯文本来处理：

```
let xhr = new XMLHttpRequest();
xhr.open("get", "text.php", true);
xhr.overrideMimeType("text/xml");
xhr.send(null);
```

这个例子强制让 XHR 把响应当成 XML 而不是纯文本来处理。为了正确覆盖响应的 MIME 类型，必须在调用 send()之前调用 overrideMimeType()。

24.2　进度事件

Progress Events 是 W3C 的工作草案，定义了客户端–服务器端通信。这些事件最初只针对 XHR，现在也推广到了其他类似的 API。有以下 6 个进度相关的事件。

- loadstart：在接收到响应的第一个字节时触发。
- progress：在接收响应期间反复触发。
- error：在请求出错时触发。
- abort：在调用 abort()终止连接时触发。
- load：在成功接收完响应时触发。
- loadend：在通信完成时，且在 error、abort 或 load 之后触发。

每次请求都会首先触发 loadstart 事件，之后是一个或多个 progress 事件，接着是 error、abort 或 load 中的一个，最后以 loadend 事件结束。

这些事件大部分都很好理解，但其中有两个需要说明一下。

24.2.1　load 事件

Firefox 最初在实现 XHR 的时候，曾致力于简化交互模式。最终，增加了一个 load 事件用于替代 readystatechange 事件。load 事件在响应接收完成后立即触发，这样就不用检查 readyState 属性了。onload 事件处理程序会收到一个 event 对象，其 target 属性设置为 XHR 实例，在这个实例上可以访问所有 XHR 对象属性和方法。不过，并不是所有浏览器都实现了这个事件的 event 对象。考虑到跨浏览器兼容，还是需要像下面这样使用 XHR 对象变量：

```
let xhr = new XMLHttpRequest();
xhr.onload = function() {
  if ((xhr.status >= 200 && xhr.status < 300) || xhr.status == 304) {
    alert(xhr.responseText);
  } else {
    alert("Request was unsuccessful: " + xhr.status);
  }
};
xhr.open("get", "altevents.php", true);
xhr.send(null);
```

只要是从服务器收到响应，无论状态码是什么，都会触发 load 事件。这意味着还需要检查 status 属性才能确定数据是否有效。Firefox、Opera、Chrome 和 Safari 都支持 load 事件。

24.2.2 progress 事件

Mozilla 在 XHR 对象上另一个创新是 progress 事件，在浏览器接收数据期间，这个事件会反复触发。每次触发时，onprogress 事件处理程序都会收到 event 对象，其 target 属性是 XHR 对象，且包含 3 个额外属性：lengthComputable、position 和 totalSize。其中，lengthComputable 是一个布尔值，表示进度信息是否可用；position 是接收到的字节数；totalSize 是响应的 Content-Length 头部定义的总字节数。有了这些信息，就可以给用户提供进度条了。以下代码演示了如何向用户展示进度：

```
let xhr = new XMLHttpRequest();
xhr.onload = function(event) {
  if ((xhr.status >= 200 && xhr.status < 300) ||
      xhr.status == 304) {
    alert(xhr.responseText);
  } else {
    alert("Request was unsuccessful: " + xhr.status);
  }
};
xhr.onprogress = function(event) {
  let divStatus = document.getElementById("status");
  if (event.lengthComputable) {
    divStatus.innerHTML = "Received " + event.position + " of " +
      event.totalSize +
" bytes";
  }
};

xhr.open("get", "altevents.php", true);
xhr.send(null);
```

为了保证正确执行，必须在调用 open()之前添加 onprogress 事件处理程序。在前面的例子中，每次触发 progress 事件都会更新 HTML 元素中的信息。假设响应有 Content-Length 头部，就可以利用这些信息计算出已经收到响应的百分比。

24.3 跨源资源共享

通过 XHR 进行 Ajax 通信的一个主要限制是跨源安全策略。默认情况下，XHR 只能访问与发起请求的页面在同一个域内的资源。这个安全限制可以防止某些恶意行为。不过，浏览器也需要支持合法跨源访问的能力。

跨源资源共享（CORS，Cross-Origin Resource Sharing）定义了浏览器与服务器如何实现跨源通信。CORS 背后的基本思路就是使用自定义的 HTTP 头部允许浏览器和服务器相互了解，以确实请求或响应应该成功还是失败。

对于简单的请求，比如 GET 或 POST 请求，没有自定义头部，而且请求体是 text/plain 类型，这样的请求在发送时会有一个额外的头部叫 Origin。Origin 头部包含发送请求的页面的源（协议、域名和端口），以便服务器确定是否为其提供响应。下面是 Origin 头部的一个示例：

```
Origin: http://www.nczonline.net
```

如果服务器决定响应请求，那么应该发送 Access-Control-Allow-Origin 头部，包含相同的源；或者如果资源是公开的，那么就包含"*"。比如：

```
Access-Control-Allow-Origin: http://www.nczonline.net
```

如果没有这个头部，或者有但源不匹配，则表明不会响应浏览器请求。否则，服务器就会处理这个请求。注意，无论请求还是响应都不会包含 cookie 信息。

现代浏览器通过 XMLHttpRequest 对象原生支持 CORS。在尝试访问不同源的资源时，这个行为会被自动触发。要向不同域的源发送请求，可以使用标准 XHR 对象并给 open()方法传入一个绝对 URL，比如：

```
let xhr = new XMLHttpRequest();
xhr.onreadystatechange = function() {
  if (xhr.readyState == 4) {
    if ((xhr.status >= 200 && xhr.status < 300) || xhr.status == 304) {
      alert(xhr.responseText);
    } else {
      alert("Request was unsuccessful: " + xhr.status);
    }
  }
};
xhr.open("get", "http://www.somewhere-else.com/page/", true);
xhr.send(null);
```

跨域 XHR 对象允许访问 status 和 statusText 属性，也允许同步请求。出于安全考虑，跨域 XHR 对象也施加了一些额外限制。

- 不能使用 setRequestHeader()设置自定义头部。
- 不能发送和接收 cookie。
- getAllResponseHeaders()方法始终返回空字符串。

因为无论同域还是跨域请求都使用同一个接口，所以最好在访问本地资源时使用相对 URL，在访问远程资源时使用绝对 URL。这样可以更明确地区分使用场景，同时避免出现访问本地资源时出现头部或 cookie 信息访问受限的问题。

24.3.1　预检请求

CORS 通过一种叫**预检请求**（preflighted request）的服务器验证机制，允许使用自定义头部、除 GET 和 POST 之外的方法，以及不同请求体内容类型。在要发送涉及上述某种高级选项的请求时，会先向服务器发送一个“预检”请求。这个请求使用 OPTIONS 方法发送并包含以下头部。

- Origin：与简单请求相同。
- Access-Control-Request-Method：请求希望使用的方法。
- Access-Control-Request-Headers：（可选）要使用的逗号分隔的自定义头部列表。

下面是一个假设的 POST 请求，包含自定义的 NCZ 头部：

```
Origin: http://www.nczonline.net
Access-Control-Request-Method: POST
Access-Control-Request-Headers: NCZ
```

在这个请求发送后，服务器可以确定是否允许这种类型的请求。服务器会通过在响应中发送如下头

部与浏览器沟通这些信息。

- Access-Control-Allow-Origin：与简单请求相同。
- Access-Control-Allow-Methods：允许的方法（逗号分隔的列表）。
- Access-Control-Allow-Headers：服务器允许的头部（逗号分隔的列表）。
- Access-Control-Max-Age：缓存预检请求的秒数。

例如：

```
Access-Control-Allow-Origin: http://www.nczonline.net
Access-Control-Allow-Methods: POST, GET
Access-Control-Allow-Headers: NCZ
Access-Control-Max-Age: 1728000
```

预检请求返回后，结果会按响应指定的时间缓存一段时间。换句话说，只有第一次发送这种类型的请求时才会多发送一次额外的 HTTP 请求。

24.3.2 凭据请求

默认情况下，跨源请求不提供凭据（cookie、HTTP 认证和客户端 SSL 证书）。可以通过将 withCredentials 属性设置为 true 来表明请求会发送凭据。如果服务器允许带凭据的请求，那么可以在响应中包含如下 HTTP 头部：

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

如果发送了凭据请求而服务器返回的响应中没有这个头部，则浏览器不会把响应交给 JavaScript（responseText 是空字符串，status 是 0，onerror()被调用）。注意，服务器也可以在预检请求的响应中发送这个 HTTP 头部，以表明这个源允许发送凭据请求。

24.4 替代性跨源技术

CORS 出现之前，实现跨源 Ajax 通信是有点麻烦的。开发者需要依赖能够执行跨源请求的 DOM 特性，在不使用 XHR 对象情况下发送某种类型的请求。虽然 CORS 目前已经得到广泛支持，但这些技术仍然没有过时，因为它们不需要修改服务器。

24.4.1 图片探测

图片探测是利用<img>标签实现跨域通信的最早的一种技术。任何页面都可以跨域加载图片而不必担心限制，因此这也是在线广告跟踪的主要方式。可以动态创建图片，然后通过它们的 onload 和 onerror 事件处理程序得知何时收到响应。

这种动态创建图片的技术经常用于**图片探测**（image pings）。图片探测是与服务器之间简单、跨域、单向的通信。数据通过查询字符串发送，响应可以随意设置，不过一般是位图图片或值为 204 的状态码。浏览器通过图片探测拿不到任何数据，但可以通过监听 onload 和 onerror 事件知道什么时候能接收到响应。下面看一个例子：

```
let img = new Image();
img.onload = img.onerror = function() {
  alert("Done!");
};
img.src = "http://www.example.com/test?name=Nicholas";
```

这个例子创建了一个新的 Image 实例，然后为它的 onload 和 onerror 事件处理程序添加了同一个函数。这样可以确保请求完成时无论什么响应都会收到通知。设置完 src 属性之后请求就开始了，这个例子向服务器发送了一个 name 值。

图片探测频繁用于跟踪用户在页面上的点击操作或动态显示广告。当然，图片探测的缺点是只能发送 GET 请求和无法获取服务器响应的内容。这也是只能利用图片探测实现浏览器与服务器单向通信的原因。

24.4.2 JSONP

JSONP 是“JSON with padding”的简写，是在 Web 服务上流行的一种 JSON 变体。JSONP 看起来跟 JSON 一样，只是会被包在一个函数调用里，比如：

```
callback({ "name": "Nicholas" });
```

JSONP 格式包含两个部分：回调和数据。回调是在页面接收到响应之后应该调用的函数，通常回调函数的名称是通过请求来动态指定的。而数据就是作为参数传给回调函数的 JSON 数据。下面是一个典型的 JSONP 请求：

```
http://freegeoip.net/json/?callback=handleResponse
```

这个 JSONP 请求的 URL 是一个地理位置服务。JSONP 服务通常支持以查询字符串形式指定回调函数的名称。比如这个例子就把回调函数的名字指定为 handleResponse()。

JSONP 调用是通过动态创建<script>元素并为 src 属性指定跨域 URL 实现的。此时的<script>与<img>元素类似，能够不受限制地从其他域加载资源。因为 JSONP 是有效的 JavaScript，所以 JSONP 响应在被加载完成之后会立即执行。比如下面这个例子：

```
function handleResponse(response) {
  console.log(`
      You're at IP address ${response.ip}, which is in
      ${response.city}, ${response.region_name}`);
}
let script = document.createElement("script");
script.src = "http://freegeoip.net/json/?callback=handleResponse";
document.body.insertBefore(script, document.body.firstChild);
```

这个例子会显示从地理位置服务获取的 IP 地址及位置信息。

JSONP 由于其简单易用，在开发者中非常流行。相比于图片探测，使用 JSONP 可以直接访问响应，实现浏览器与服务器的双向通信。不过 JSONP 也有一些缺点。

首先，JSONP 是从不同的域拉取可执行代码。如果这个域并不可信，则可能在响应中加入恶意内容。此时除了完全删除 JSONP 没有其他办法。在使用不受控的 Web 服务时，一定要保证是可以信任的。

第二个缺点是不好确定 JSONP 请求是否失败。虽然 HTML5 规定了<script>元素的 onerror 事件处理程序，但还没有被任何浏览器实现。为此，开发者经常使用计时器来决定是否放弃等待响应。这种方式并不准确，毕竟不同用户的网络连接速度和带宽是不一样的。

24.5 Fetch API

Fetch API 能够执行 XMLHttpRequest 对象的所有任务，但更容易使用，接口也更现代化，能够在 Web 工作线程等现代 Web 工具中使用。XMLHttpRequest 可以选择异步，而 Fetch API 则必须是异步。

Fetch API 是 WHATWG 的一个“活标准”（living standard），用规范原文说，就是“Fetch 标准定义请求、响应，以及绑定二者的流程：获取（fetch）”。

Fetch API 本身是使用 JavaScript 请求资源的优秀工具，同时这个 API 也能够应用在服务线程（service worker）中，提供拦截、重定向和修改通过 `fetch()`生成的请求接口。

24.5.1 基本用法

`fetch()`方法是暴露在全局作用域中的，包括主页面执行线程、模块和工作线程。调用这个方法，浏览器就会向给定 URL 发送请求。

1. 分派请求

`fetch()`只有一个必需的参数 `input`。多数情况下，这个参数是要获取资源的 URL。这个方法返回一个期约：

```
let r = fetch('/bar');
console.log(r); // Promise <pending>
```

URL 的格式（相对路径、绝对路径等）的解释与 XHR 对象一样。

请求完成、资源可用时，期约会解决为一个 `Response` 对象。这个对象是 API 的封装，可以通过它取得相应资源。获取资源要使用这个对象的属性和方法，掌握响应的情况并将负载转换为有用的形式，如下所示：

```
fetch('bar.txt')
  .then((response) => {
    console.log(response);
  });

// Response { type: "basic", url: ... }
```

2. 读取响应

读取响应内容的最简单方式是取得纯文本格式的内容，这要用到 `text()`方法。这个方法返回一个期约，会解决为取得资源的完整内容：

```
fetch('bar.txt')
  .then((response) => {
    response.text().then((data) => {
      console.log(data);
    });
  });

// bar.txt 的内容
```

内容的结构通常是打平的：

```
fetch('bar.txt')
  .then((response) => response.text())
  .then((data) => console.log(data));

// bar.txt 的内容
```

3. 处理状态码和请求失败

Fetch API 支持通过 `Response` 的 `status`（状态码）和 `statusText`（状态文本）属性检查响应状态。成功获取响应的请求通常会产生值为 200 的状态码，如下所示：

```
fetch('/bar')
  .then((response) => {
    console.log(response.status);     // 200
    console.log(response.statusText); // OK
  });
```

请求不存在的资源通常会产生值为 404 的状态码：

```
fetch('/does-not-exist')
  .then((response) => {
    console.log(response.status);     // 404
    console.log(response.statusText); // Not Found
  });
```

请求的 URL 如果抛出服务器错误会产生值为 500 的状态码：

```
fetch('/throw-server-error')
  .then((response) => {
    console.log(response.status);     // 500
    console.log(response.statusText); // Internal Server Error
  });
```

可以显式地设置 `fetch()`在遇到重定向时的行为（本章后面会介绍），不过默认行为是跟随重定向并返回状态码不是 300~399 的响应。跟随重定向时，响应对象的 `redirected` 属性会被设置为 `true`，而状态码仍然是 200：

```
fetch('/permanent-redirect')
  .then((response) => {
    // 默认行为是跟随重定向直到最终 URL
    // 这个例子会出现至少两轮网络请求
    // <origin url>/permanent-redirect -> <redirect url>
    console.log(response.status);     // 200
    console.log(response.statusText); // OK
    console.log(response.redirected); // true
  });
```

在前面这几个例子中，虽然请求可能失败（如状态码为 500），但都只执行了期约的**解决**处理函数。事实上，只要服务器返回了响应，`fetch()`期约都会解决。这个行为是合理的：系统级网络协议已经成功完成消息的一次往返传输。至于真正的“成功”请求，则需要在处理响应时再定义。

通常状态码为 200 时就会被认为成功了，其他情况可以被认为未成功。为区分这两种情况，可以在状态码非 200~299 时检查 `Response` 对象的 `ok` 属性：

```
fetch('/bar')
  .then((response) => {
    console.log(response.status); // 200
    console.log(response.ok);     // true
  });
fetch('/does-not-exist')
  .then((response) => {
    console.log(response.status); // 404
    console.log(response.ok);     // false
  });
```

因为服务器没有响应而导致浏览器超时，这样真正的 `fetch()`失败会导致期约被拒绝：

```
fetch('/hangs-forever')
  .then((response) => {
    console.log(response);
  }, (err) => {
```

```
    console.log(err);
  });

// (浏览器超时后)
// TypeError: "NetworkError when attempting to fetch resource."
```

违反 CORS、无网络连接、HTTPS 错配及其他浏览器/网络策略问题都会导致期约被拒绝。

可以通过 url 属性检查通过 fetch()发送请求时使用的完整 URL：

```
// foo.com/bar/baz 发送的请求
console.log(window.location.href); // https://foo.com/bar/baz

fetch('qux').then((response) => console.log(response.url));
// https://foo.com/bar/qux

fetch('/qux').then((response) => console.log(response.url));
// https://foo.com/qux

fetch('//qux.com').then((response) => console.log(response.url));
// https://qux.com

fetch('https://qux.com').then((response) => console.log(response.url));
// https://qux.com
```

4. 自定义选项

只使用 URL 时，fetch()会发送 GET 请求，只包含最低限度的请求头。要进一步配置如何发送请求，需要传入可选的第二个参数 init 对象。init 对象要按照下表中的键/值进行填充。

键	值
body	指定使用请求体时请求体的内容 必须是 Blob、BufferSource、FormData、URLSearchParams、ReadableStream 或 String 的实例
cache	用于控制浏览器与 HTTP 缓存的交互。要跟踪缓存的重定向，请求的 redirect 属性值必须是"follow"，而且必须符合同源策略限制。必须是下列值之一 Default ❑ fetch()返回命中的有效缓存。不发送请求 ❑ 命中无效（stale）缓存会发送条件式请求。如果响应已经改变，则更新缓存的值。然后 fetch()返回缓存的值 ❑ 未命中缓存会发送请求，并缓存响应。然后 fetch()返回响应 no-store ❑ 浏览器不检查缓存，直接发送请求 ❑ 不缓存响应，直接通过 fetch()返回 reload ❑ 浏览器不检查缓存，直接发送请求 ❑ 缓存响应，再通过 fetch()返回 no-cache ❑ 无论命中有效缓存还是无效缓存都会发送条件式请求。如果响应已经改变，则更新缓存的值。然后 fetch()返回缓存的值 ❑ 未命中缓存会发送请求，并缓存响应。然后 fetch()返回响应