Fetch API 教程

目录
基本用法
Response 对象：处理 HTTP 回应
Response 对象的同步属性
判断请求是否成功
Response.headers 属性
读取内容的方法
Response.clone()
Response.body 属性
fetch()的第二个参数：定制 HTTP 请求
fetch()配置对象的完整 API
取消 fetch()请求
参考链接

fetch() 是 XMLHttpRequest 的升级版，用于在 JavaScript 脚本里面发出 HTTP 请求。

浏览器原生提供这个对象。本章详细介绍它的用法。

基本用法
fetch() 的功能与 XMLHttpRequest 基本相同，都是向服务器发出 HTTP 请求，但有三个主要的差异。

（ 1） fetch()使用 Promise，不使用回调函数，因此大大简化了写法，写起来更简洁。

（ 2） fetch()采用模块化设计， API 分散在多个对象上（ Response 对象、 Request 对象、 Headers
对象），更合理一些；相比之下， XMLHttpRequest 的 API
设计并不是很好，输入、输出、状态都在同一个接口管理，容易写出非常混乱的代码。

（ 3） fetch()通过数据流（ Stream
对象）处理数据，可以分块读取，有利于提高网站性能表现，减少内存占用，对于请求大文件或者网速慢的场
景相当有用。 XMLHttpRequest
对象不支持数据流，所有的数据全部放在缓存里，不支持分块读取，必须等待全部获取后，再一次性读取。

用法上，`fetch()`接受一个 URL 字符串作为参数，默认向该网址发出 GET 请求，返回一个 Promise 对象。它的基本用法如下。

```
fetch(url)
  .then(...)
  .catch(...)
```

下面是一个例子，从服务器获取 JSON 数据。

```
fetch('https://api.github.com/users/ruanyf')
  .then(response => response.json())
  .then(json => console.log(json))
  .catch(err => console.log('Request Failed', err));
```

上面示例中，`fetch()`接收到的`response`是一个 Stream 对象，里面的数据本例是 JSON 数据，所以使用`response.json()`方法，将其转为 JSON 对象。它是一个异步操作，返回一个 Promise 对象。

Promise 可以使用 await 语法改写，使得语义更清晰。

```
async function getJSON() {
  let url = 'https://api.github.com/users/ruanyf';
  try {
    let response = await fetch(url);
    return await response.json();
  } catch (error) {
    console.log('Request Failed', error);
  }
}
```

上面示例中，`await`语句必须放在`try...catch`里面，这样才能捕捉异步操作中可能发生的错误。

后文都采用`await`的写法，不再使用`.then()`的写法。

Response 对象：处理 HTTP 回应

Response 对象的同步属性

`fetch()`请求成功以后，得到的是一个 Response 对象。它对应服务器的 HTTP 回应。

```
const response = await fetch(url);
```

前面说过，Response 包含的数据通过 Stream 接口异步读取，但是它还包含一些同步属性，对应 HTTP 回应的标头信息（Headers），可以立即读取。

```
async function fetchText() {
  let response = await fetch('/readme.txt');
  console.log(response.status);
  console.log(response.statusText);
}
```

上面示例中，`response.status`和`response.statusText`就是 Response 的同步属性，可以立即读取。

标头信息属性有下面这些。

Response.ok

`Response.ok`属性返回一个布尔值，表示请求是否成功，`true`对应 HTTP 请求的状态码 200 到 299，`false`对应其他的状态码。

Response.status

`Response.status`属性返回一个数字，表示 HTTP 回应的状态码（例如 200，表示成功请求）。

Response.statusText

`Response.statusText`属性返回一个字符串，表示 HTTP 回应的状态信息（例如请求成功以后，服务器返回“OK”）。

Response.url

`Response.url`属性返回请求的 URL。如果 URL 存在跳转，该属性返回的是最终 URL。

Response.type

`Response.type`属性返回请求的类型。可能的值如下：

- `basic`：普通请求，即同源请求。
- `cors`：跨源请求。
- `error`：网络错误，主要用于 Service Worker。
- `opaque`：如果`fetch()`请求的`type`属性设为`no-cors`，就会返回这个值，详见请求部分。表示发出的是简单的跨源请求，类似`<form>`表单的那种跨源请求。
- `opaqueredirect`：如果`fetch()`请求的`redirect`属性设为`manual`，就会返回这个值，详见请求部分。

Response.redirected

`Response.redirected`属性返回一个布尔值，表示请求是否发生过跳转。

判断请求是否成功

`fetch()`发出请求以后，有一个很重要的注意点：只有网络错误，或者无法连接时，`fetch()`才会报错，其他情况都不会报错，而是认为请求成功。

这就是说，即使服务器返回的状态码是 4xx 或 5xx，`fetch()`也不会报错（即 Promise 不会变为`rejected`状态）。

只有通过`Response.status`属性，得到 HTTP 回应的真实状态码，才能判断请求是否成功。请看下面的例子。

```
async function fetchText() {
  let response = await fetch('/readme.txt');
  if (response.status >= 200 && response.status < 300) {
    return await response.text();
  } else {
    throw new Error(response.statusText);
  }
}
```

上面示例中，`response.status`属性只有等于 2xx（200~299），才能认定请求成功。这里不用考虑网址跳转（状态码为 3xx），因为`fetch()`会将跳转的状态码自动转为 200。

另一种方法是判断`response.ok`是否为`true`。

```
if (response.ok) {
  // 请求成功
} else {
  // 请求失败
}
```

Response.headers 属性

Response 对象还有一个`Response.headers`属性，指向一个 Headers 对象，对应 HTTP 回应的所有标头。

Headers 对象可以使用`for...of`循环进行遍历。

```
const response = await fetch(url);

for (let [key, value] of response.headers) {
  console.log(`${key} : ${value}`);
}

// 或者
for (let [key, value] of response.headers.entries()) {
  console.log(`${key} : ${value}`);
}
```

Headers 对象提供了以下方法：

> - `Headers.get()` 根据指定的键名返回键值。
> - `Headers.has()` 返回一个布尔值，表示是否包含某个键。
> - `Headers.set()` 将指定的键名设置为新的键值，如果该键名不存在则会添加。
> - `Headers.append()` 添加键值对。
> - `Headers.delete()` 删除键值对。
> - `Headers.keys()` 返回一个遍历器，可以依次遍历所有键名。
> - `Headers.values()` 返回一个遍历器，可以依次遍历所有键值。
> - `Headers.entries()` 返回一个遍历器，可以依次遍历所有键值对（`[key, value]`）。
> - `Headers.forEach()` 依次遍历所有键值对，参数为遍历器的回调函数。

下面是一个读取 Headers 对象的例子。

```
let response =  await  fetch(url);
response.headers.get('Content-Type')
// application/json; charset=utf-8
```

`Headers.keys()` 和 `Headers.values()` 用来遍历所有键名和键值。

```
// 或者
for(let key of myHeaders.keys()) {
  console.log(key);
}
```

```
// 键值
for(let value of myHeaders.values()) {
  console.log(value);
}
```

`Headers.forEach()` 方法也可以遍历所有的键值和键名。

```
let response = await fetch(url);
response.headers.forEach(
  (value, key) => console.log(key, ':', value)
);
```

读取内容的方法

`Response` 对象根据服务器返回的不同类型的数据，提供了不同的读取方法。

> - `response.text()`：得到文本字符串。
> - `response.json()`：得到 JSON 对象。
> - `response.blob()`：得到二进制 Blob 对象。
> - `response.formData()`：得到 FormData 表单对象。
> - `response.arrayBuffer()`：得到二进制 ArrayBuffer 对象。

上面 5个读取方法都是异步的，返回的都是 Promise 对象。必须等到异步操作结束，才能得到服务器返回的完整数据。

response.text()

`response.text()` 可以用于获取文本数据，比如 HTML 文件。

```
const response = await fetch('/users.html');
const body = await response.text();
document.body.innerHTML = body
```

response.json()

`response.json()` 主要用于获取服务器返回的 JSON 数据，前面已经举过例子了。

response.formData()

`response.formData()` 主要用在 Service Worker 里面，拦截用户提交的表单，修改某些数据以后，再提交给服务器。

response.blob()

`response.blob()`用于获取二进制文件。

```
const response = await fetch('flower.jpg');
const myBlob = await response.blob();
const objectURL = URL.createObjectURL(myBlob);

const myImage = document.querySelector('img');
myImage.src = objectURL;
```

上面示例读取图片文件 `flower.jpg`，显示在网页上。

response.arrayBuffer()

`response.arrayBuffer()`主要用于获取流媒体文件。

```
const audioCtx = new window.AudioContext();
const source = audioCtx.createBufferSource();

const response = await fetch('song.ogg');
const buffer = await response.arrayBuffer();

const decodeData = await audioCtx.decodeAudioData(buffer);
source.buffer = buffer;
source.connect(audioCtx.destination);
source.loop = true;
```

上面示例是 `response.arrayBuffer()`获取音频文件 `song.ogg`，然后在线播放的例子。

Response.clone()

Stream 对象只能读取一次，读取完就没了。这意味着，前一节的五个读取方法，只能使用一个，否则会报错。

```
let text =  await response.text();
let json =  await response.json();  // 报错
```

上面示例先使用了 `response.text()`，就把 Stream 读完了。后面再调用 `response.json()`，就没有内容可读了，所以报错。

Response 对象提供 `Response.clone()`方法，创建 `Response`对象的副本，实现多次读取。

```
const response1 = await fetch('flowers.jpg');
const response2 = response1.clone();

const myBlob1 = await response1.blob();
const myBlob2 = await response2.blob();

image1.src = URL.createObjectURL(myBlob1);
image2.src = URL.createObjectURL(myBlob2);
```

上面示例中，`response.clone()`复制了一份 Response 对象，然后将同一张图片读取了两次。

Response 对象还有一个`Response.redirect()`方法，用于将 Response 结果重定向到指定的 URL。该方法一般只用在 Service Worker 里面，这里就不介绍了。

Response.body 属性

`Response.body`属性是 Response 对象暴露出的底层接口，返回一个 ReadableStream 对象，供用户操作。

它可以用来分块读取内容，应用之一就是显示下载的进度。

```
const response = await fetch('flower.jpg');
const reader = response.body.getReader();

while(true) {
  const {done, value} = await reader.read();

  if (done) {
    break;
  }

  console.log(`Received ${value.length} bytes`)
}
```

上面示例中，`response.body.getReader()`方法返回一个遍历器。这个遍历器的`read()`方法每次返回一个对象，表示本次读取的内容块。

这个对象的`done`属性是一个布尔值，用来判断有没有读完；`value`属性是一个 arrayBuffer 数组，表示内容块的内容，而`value.length`属性是当前块的大小。

`fetch()`的第二个参数：定制 HTTP 请求

`fetch()`的第一个参数是 URL，还可以接受第二个参数，作为配置对象，定制发出的 HTTP 请求。

```
fetch(url, optionObj)
```

上面命令的 `optionObj` 就是第二个参数。

HTTP 请求的方法、标头、数据体都在这个对象里面设置。下面是一些示例。

（1）POST 请求

```
const response = await fetch(url, {
  method: 'POST',
  headers: {
    "Content-type": "application/x-www-form-urlencoded; charset=UTF-8",
  },
  body: 'foo=bar&lorem=ipsum',
});

const json = await response.json();
```

上面示例中，配置对象用到了三个属性。

> - `method`：HTTP 请求的方法，`POST`、`DELETE`、`PUT`都在这个属性设置。
> - `headers`：一个对象，用来定制 HTTP 请求的标头。
> - `body`：POST 请求的数据体。

注意，有些标头不能通过`headers`属性设置，比如`Content-Length`、`Cookie`、`Host`等等。它们是由浏览器自动生成，无法修改。

（2）提交 JSON 数据

```
const user =  { name:  'John', surname:  'Smith'  };
const response = await fetch('/article/fetch/post/user', {
  method: 'POST',
  headers: {
```

```
    'Content-Type': 'application/json;charset=utf-8'
  },
  body: JSON.stringify(user)
});
```

上面示例中，标头 `Content-Type` 要设成 `'application/json;charset=utf-8'`。因为默认发送的是纯文本，`Content-Type` 的默认值是 `'text/plain;charset=UTF-8'`。

（3）提交表单

```
const form = document.querySelector('form');

const response = await fetch('/users', {
  method: 'POST',
  body: new FormData(form)
})
```

（4）文件上传

如果表单里面有文件选择器，可以用前一个例子的写法，上传的文件包含在整个表单里面，一起提交。

另一种方法是用脚本添加文件，构造出一个表单，进行上传，请看下面的例子。

```
const input = document.querySelector('input[type="file"]');

const data = new FormData();
data.append('file', input.files[0]);
data.append('user', 'foo');

fetch('/avatars', {
  method: 'POST',
  body: data
});
```

上传二进制文件时，不用修改标头的 `Content-Type`，浏览器会自动设置。

（5）直接上传二进制数据

`fetch()` 也可以直接上传二进制数据，将 Blob 或 arrayBuffer 数据放在 `body` 属性里面。

```
let blob = await new Promise(resolve =>
  canvasElem.toBlob(resolve,  'image/png')
```

```
);

let response = await fetch('/article/fetch/post/image', {
  method:  'POST',
  body: blob
});
```

`fetch()`配置对象的完整 API

`fetch()`第二个参数的完整 API 如下。

```
const response = fetch(url, {
  method: "GET",
  headers: {
    "Content-Type": "text/plain;charset=UTF-8"
  },
  body: undefined,
  referrer: "about:client",
  referrerPolicy: "no-referrer-when-downgrade",
  mode: "cors",
  credentials: "same-origin",
  cache: "default",
  redirect: "follow",
  integrity: "",
  keepalive: false,
  signal: undefined
});
```

`fetch()`请求的底层用的是 Request() 对象的接口，参数完全一样，因此上面的 API 也是 `Request()`的 API。

这些属性里面，`headers`、`body`、`method`前面已经给过示例了，下面是其他属性的介绍。

cache

`cache`属性指定如何处理缓存。可能的取值如下：

- `default`：默认值，先在缓存里面寻找匹配的请求。
- `no-store`：直接请求远程服务器，并且不更新缓存。
- `reload`：直接请求远程服务器，并且更新缓存。
- `no-cache`：将服务器资源跟本地缓存进行比较，有新的版本才使用服务器资源，否则使用缓存。

- `force-cache`：缓存优先，只有不存在缓存的情况下，才请求远程服务器。
- `only-if-cached`：只检查缓存，如果缓存里面不存在，将返回 504错误。

mode

`mode`属性指定请求的模式。可能的取值如下：

- `cors`：默认值，允许跨源请求。
- `same-origin`：只允许同源请求。
- `no-cors`：请求方法只限于 GET、POST 和 HEAD，并且只能使用有限的几个简单标头，不能添加跨源的复杂标头，相当于提交表单、`<script>`加载脚本、`<img>`加载图片等传统的跨源请求方法。

credentials

`credentials`属性指定是否发送 Cookie。可能的取值如下：

- `same-origin`：默认值，同源请求时发送 Cookie，跨源请求时不发送。
- `include`：不管同源请求，还是跨源请求，一律发送 Cookie。
- `omit`：一律不发送。

跨源请求发送 Cookie，需要将 `credentials`属性设为 `include`。

```
fetch('http://another.com', {
  credentials: "include"
});
```

signal

`signal`属性指定一个 AbortSignal 实例，用于取消 `fetch()`请求，详见下一节。

keepalive

`keepalive`属性用于页面卸载时，告诉浏览器在后台保持连接，继续发送数据。

一个典型的场景就是，用户离开网页时，脚本向服务器提交一些用户行为的统计信息。这时，如果不用`keepalive`属性，数据可能无法发送，因为浏览器已经把页面卸载了。

```
window.onunload = function() {
  fetch('/analytics', {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json'
    },
    body: JSON.stringify({
```

```
      some: 'data'
    }),
    keepalive: true
  });
};
```

redirect

`redirect` HTTP

- `follow` `fetch()` HTTP
- `error` `fetch()`
- `manual` `fetch()` HTTP `response.url` URL `response.redirected` `true`

integrity

`integrity` HTTP

SHA-256

```
fetch('http://site.com/file', {
  integrity: 'sha256-abcdef'
});
```

referrer

`referrer` `fetch()` `referer`

`referer`

```
fetch('/page', {
  referrer: ''
});
```

referrerPolicy

`referrerPolicy` `Referer`

- `no-referrer-when-downgrade` `Referer` HTTPS HTTP
- `no-referrer` `Referer`
- `origin` `Referer`
- `origin-when-cross-origin` `Referer`

- `same-origin`：跨源请求不发送 `Referer`，同源请求发送。
- `strict-origin`： `Referer` 标头只包含域名， HTTPS 页面请求 HTTP 资源时不发送 `Referer` 标头。
- `strict-origin-when-cross-origin`：同源请求时 `Referer` 标头包含完整路径，跨源请求时只包含域名， HTTPS 页面请求 HTTP 资源时不发送该标头。
- `unsafe-url`：不管什么情况，总是发送 `Referer` 标头。

取消 `fetch()` 请求

`fetch()` 请求发送以后，如果中途想要取消，需要使用 `AbortController` 对象。

```
let controller = new AbortController();
let signal = controller.signal;

fetch(url, {
  signal: controller.signal
});

signal.addEventListener('abort',
  () => console.log('abort!')
);

controller.abort(); // 取消

console.log(signal.aborted); // true
```

上面示例中，首先新建 AbortController 实例，然后发送 `fetch()` 请求，配置对象的 `signal` 属性必须指定接收 AbortController 实例发送的信号 `controller.signal`。

`controller.abort()` 方法用于发出取消信号。这时会触发 `abort` 事件，这个事件可以监听，也可以通过 `controller.signal.aborted` 属性判断取消信号是否已经发出。

下面是一个 1秒后自动取消请求的例子。

```
let controller = new AbortController();
setTimeout(() => controller.abort(), 1000);

try {
  let response = await fetch('/long-operation', {
    signal: controller.signal
```

```
  });
} catch(err) {
  if (err.name == 'AbortError') {
    console.log('Aborted!');
  } else {
    throw err;
  }
}
```

□□□

- Network requests: Fetch
- node-fetch
- Introduction to fetch()
- Using Fetch
- Javascript Fetch API: The XMLHttpRequest evolution
- A Guide to Faster Web App I/O and Data Operations with Streams

---

□□□ #2
□ □□□ □□□ 2017-11-27 17:53:41 UTC
□ □□□ □□□ 2023-09-09 19:46:22 UTC