

ConardLi Lv6

2019年03月27日 阅读 17785

[关注](#)

全面分析前端的网络请求方式

一、前端进行网络请求的关注点

大多数情况下，在前端发起一个网络请求我们只需关注下面几点：

- 传入基本参数 (`url` ，请求方式)
- 请求参数、请求参数类型
- 设置请求头
- 获取响应的方式
- 获取响应头、响应状态、响应结果
- 异常处理
- 携带 `cookie` 设置
- 跨域请求

二、前端进行网络请求的方式

- `form` 表单、`iframe` 、刷新页面
- `Ajax` - 异步网络请求的开山鼻祖
- `jQuery` - 一个时代
- `fetch` - `Ajax` 的替代者
- `axios`、`request` 等众多开源库

三、关于网络请求的疑问

- `Ajax` 的出现解决了什么问题
- 原生 `Ajax` 如何使用
- `jQuery` 的网络请求方式
- `fetch` 的用法以及坑点

[首页](#) ▼[探索掘金](#)

带着以上这些问题、关注点我们对几种网络请求进行一次全面的分析。

四、Ajax的出现解决了什么问题

在 Ajax 出现之前，web 程序是这样工作的：



这种交互的缺陷是显而易见的，任何和服务器的交互都需要刷新页面，用户体验非常差，Ajax 的出现解决了这个问题。Ajax 全称 **Asynchronous JavaScript + XML**（异步 JavaScript 和 XML）

使用 Ajax，网页应用能够快速地将增量更新呈现在用户界面上，而不需要重载（刷新）整个页面。

Ajax 本身不是一种新技术，而是用来描述一种使用现有技术集合实现的一个技术方案，浏览器的 XMLHttpRequest 是实现 Ajax 最重要的对象（IE6 以下使用 ActiveXObject）。

尽管 X 在 Ajax 中代表 XML，但由于 JSON 的许多优势，比如更加轻量以及作为 Javascript 的一部分，目前 JSON 的使用比 XML 更加普遍。

五、原生Ajax的用法

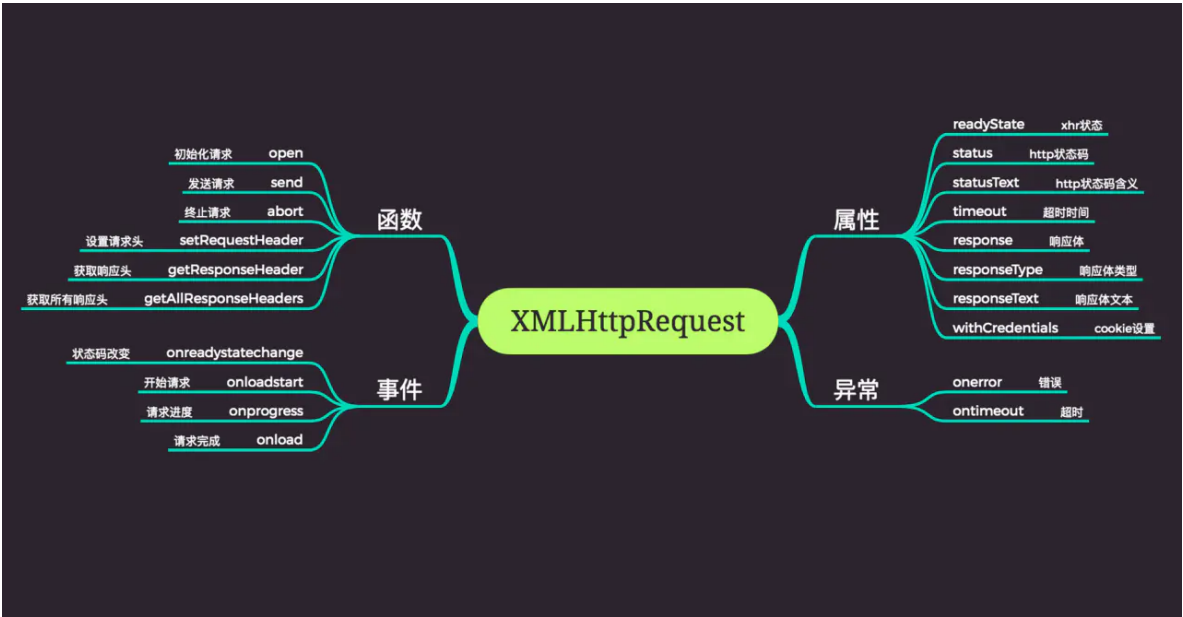
这里主要分析 XMLHttpRequest 对象，下面是它的一段基础使用：

复制代码

```
var xhr = new XMLHttpRequest();
xhr.open('post', 'www.xxx.com', true)
// 接收返回值
xhr.onreadystatechange = function(){
    if(xhr.readyState === 4 ){
        if(xhr.status >= 200 && xhr.status < 300) || xhr.status == 304){
            console.log(xhr.responseText);
        }
    }
}
```

```
// 处理请求参数
postData = {"name1":"value1","name2":"value2"};
postData = (function(value){
    var dataString = "";
    for(var key in value){
        dataString += key+"="+value[key]+"&";
    };
    return dataString;
})(postData));
// 设置请求头
xhr.setRequestHeader("Content-type","application/x-www-form-urlencoded");
// 异常处理
xhr.onerror = function() {
    console.log('Network request failed')
}
// 跨域携带cookie
xhr.withCredentials = true;
// 发出请求
xhr.send(postData);
```

下面分别对 XMLHttpRequest 对象常用的的函数、属性、事件进行分析。



函数

open

用于初始化一个请求，用法：

- **method** : 请求方式, 如 `get`、`post`
- **url** : 请求的 `url`
- **async** : 是否为异步请求

send

用于发送 **HTTP** 请求, 即调用该方法后 **HTTP** 请求才会被真正发出, 用法:

[复制代码](#)

```
xhr.send(param)
```

- **param** : http请求的参数, 可以为 `string`、`Blob` 等类型。

abort

用于终止一个 **ajax** 请求, 调用此方法后 **readyState** 将被设置为 `0`, 用法:

[复制代码](#)

```
xhr.abort()
```

setRequestHeader

用于设置 **HTTP** 请求头, 此方法必须在 `open()` 方法和 `send()` 之间调用, 用法:

[复制代码](#)

```
xhr.setRequestHeader(header, value);
```

getResponseHeader

用于获取 **http** 返回头, 如果在返回头中有多个一样的名称, 那么返回的值就会是用逗号和空格将值分隔的字符串, 用法:

[复制代码](#)

```
var header = xhr.getResponseHeader(name);
```

属性

readyState

用来标识当前 **XMLHttpRequest** 对象所处的状态, **XMLHttpRequest** 对象总是位于下列状态中的一个。

[首页](#) ▼[探索掘金](#)

值	状态	描述
0	UNSENT	代理被创建，但尚未调用 <code>open()</code> 方法。
1	OPENED	<code>open()</code> 方法已经被调用。
2	HEADERS_RECEIVED	<code>send()</code> 方法已经被调用，并且头部和状态已经可获得。
3	LOADING	下载中； <code>responseText</code> 属性已经包含部分数据。
4	DONE	下载操作已完成。

status

表示 `http` 请求的状态, 初始值为 `0`。如果服务器没有显式地指定状态码, 那么 `status` 将被设置为默认值, 即 `200`。

responseType

表示响应的数据类型，并允许我们手动设置，如果为空，默认为 `text` 类型，可以有下面的取值：

值	描述
<code>""</code>	将 <code>responseType</code> 设为空字符串与设置为 <code>"text"</code> 相同，是默认类型（实际上是 <code>DOMString</code> ）。
<code>"arraybuffer"</code>	<code>response</code> 是一个包含二进制数据的 <code>JavaScript ArrayBuffer</code> 。
<code>"blob"</code>	<code>response</code> 是一个包含二进制数据的 <code>Blob</code> 对象。
<code>"document"</code>	<code>response</code> 是一个 <code>HTML Document</code> 或 <code>XML XMLHttpRequest</code> ，这取决于接收到的数据的 MIME 类型。
<code>"json"</code>	<code>response</code> 是一个 <code>JavaScript</code> 对象。这个对象是通过将接收到的数据类型视为 <code>JSON</code> 解析得到的。
<code>"text"</code>	<code>response</code> 是包含在 <code>DOMString</code> 对象中的文本。

返回响应的正文，返回的类型由上面的 `responseType` 决定。

withCredentials

`ajax` 请求默认会携带同源请求的 `cookie`，而跨域请求则不会携带 `cookie`，设置 `xhr` 的 `withCredentials` 的属性为 `true` 将允许携带跨域 `cookie`。

事件回调

onreadystatechange

```
xhr.onreadystatechange = callback;
```

[复制代码](#)

当 `readyState` 属性发生变化时，`callback` 会被触发。

onloadstart

```
xhr.onloadstart = callback;
```

[复制代码](#)

在 `ajax` 请求发送之前（`readyState==1` 后，`readyState==2` 前），`callback` 会被触发。

onprogress

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

[复制代码](#)

回调函数可以获取资源总大小 `total`，已经加载的资源大小 `loaded`，用这两个值可以计算加载进度。

onload

```
xhr.onload = callback;
```

[复制代码](#)

当一个资源及其依赖资源已完成加载时，将触发 `callback`，通常我们会在 `onload` 事件中处理返回值。

[首页](#) ▼[探索掘金](#)

onerror

[复制代码](#)

```
xhr.onerror = callback;
```

当 `ajax` 资源加载失败时会触发 `callback` 。

ontimeout

[复制代码](#)

```
xhr.ontimeout = callback;
```

当进度由于预定时间到期而终止时，会触发 `callback`，超时时间可使用 `timeout` 属性进行设置。

六、jQuery对Ajax的封装

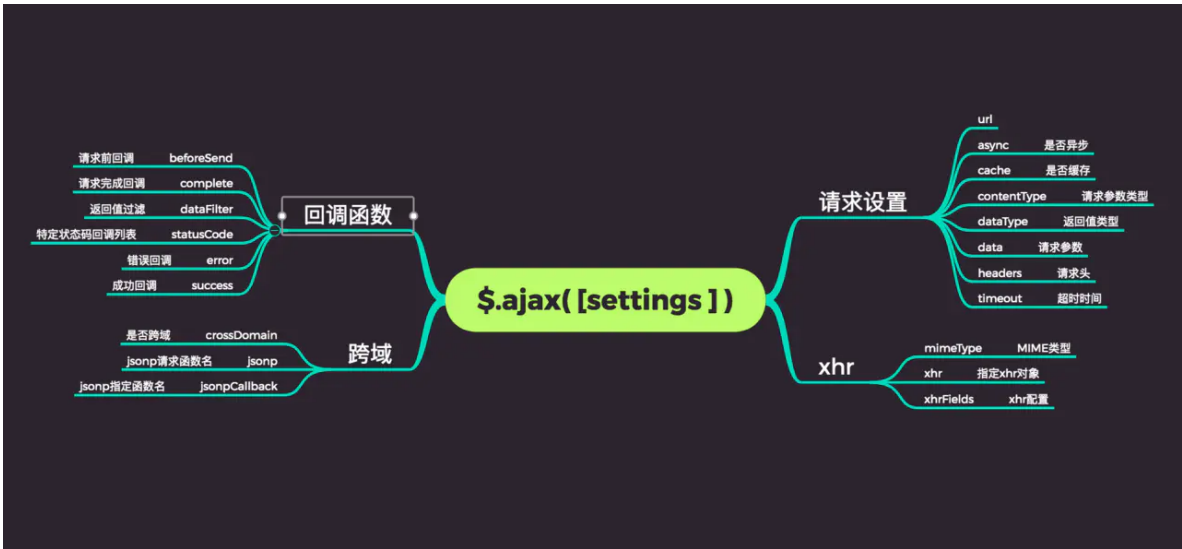
在很长一段时间里，人们使用 `jQuery` 提供的 `ajax` 封装进行网络请求，包括 `$.ajax`、`$.get`、`$.post` 等，这几个方法放到现在，我依然觉得很实用。

[复制代码](#)

```
$.ajax({
  dataType: 'json', // 设置返回值类型
  contentType: 'application/json', // 设置参数类型
  headers: {'Content-Type', 'application/json'}, // 设置请求头
  xhrFields: { withCredentials: true }, // 跨域携带cookie
  data: JSON.stringify({a: [{b:1, a:1}]}), // 传递参数
  error: function(xhr, status) { // 错误处理
    console.log(xhr, status);
  },
  success: function (data, status) { // 获取结果
    console.log(data, status);
  }
})
```

`$.ajax` 只接收一个参数，这个参数接收一系列配置，其自己封装了一个 `jqXHR` 对象，有兴趣可以阅读一下[jQuery-ajax 源码](#)

[首页](#) ▼[探索掘金](#)



常用配置：

url

当前页地址。发送请求的地址。

type

类型：String 请求方式 ("POST" 或 "GET")，默认为 "GET"。注意：其它 HTTP 请求方法，如 PUT 和 DELETE 也可以使用，但仅部分浏览器支持。

timeout

类型：Number 设置请求超时时间（毫秒）。此设置将覆盖全局设置。

success

类型：Function 请求成功后的回调函数。

jsonp

在一个 jsonp 请求中重写回调函数的名字。这个值用来替代在 "callback=?" 这种 GET 或 POST 请求中 URL 参数里的 "callback" 部分。

error 类型：Function。请求失败时调用此函数。

注意：源码里对错误的判定：

返回值除了这几个状态码都会进 `error` 回调。

dataType

[复制代码](#)

`"xml"`: 返回 XML 文档，可用 jQuery 处理。

`"html"`: 返回纯文本 HTML 信息；包含的 `script` 标签会在插入 dom 时执行。

`"script"`: 返回纯文本 JavaScript 代码。不会自动缓存结果。除非设置了 `"cache"` 参数。注意：在远程请求时(不在同

`"json"`: 返回 JSON 数据。

`"jsonp"`: JSONP 格式。使用 JSONP 形式调用函数时，如 `"myurl?callback=?"` jQuery 将自动替换 `?` 为正确的函数名

`"text"`: 返回纯文本字符串

data

类型：`String` 使用 `JSON.stringify` 转码

complete

类型：`Function` 请求完成后回调函数 (请求成功或失败之后均调用)。

async

类型：`Boolean` 默认值: `true`。默认设置下，所有请求均为异步请求。如果需要发送同步请求，请将此选项设置为 `false`。

contentType

类型：`String` 默认值: `"application/x-www-form-urlencoded"`。发送信息至服务器时内容编码类型。

键值对这样组织在一般的情况下是没有什么问题的，这里说的一般是，不带嵌套类型 `JSON`，也就是简单的 `JSON`，形如这样：

[复制代码](#)

```
{
  a: 1,
  b: 2,
  c: 3
}
```

但是在一些复杂的情况下就有问题了。例如在 `Ajax` 中你要传一个复杂的 `json` 对象，也就是说嵌套数组，数组中包括对象，你这样传：`application/x-www-form-urlencoded` 这种形式是没有办法付

```
{
  data: {
    a: [{
      x: 2
    }]
  }
}
```

可以用如下方式传递复杂的 `json` 对象

```
$.ajax({
  dataType: 'json',
  contentType: 'application/json',
  data: JSON.stringify({a: [{b:1, a:1}]})
})
```

七、jQuery的替代者

近年来前端 `MV*` 的发展壮大，人们越来越少的使用 `jQuery`，我们不可能单独为了使用 `jQuery` 的 `Ajax api` 来单独引入他，无可避免的，我们需要寻找新的技术方案。

尤雨溪在他的文档中推荐大家用 `axios` 进行网络请求。`axios` 基于 `Promise` 对原生的 `XHR` 进行了非常全面的封装，使用方式也非常的优雅。另外，`axios` 同样提供了在 `node` 环境下的支持，可谓是网络请求的首选方案。

未来必定还会出现更优秀的封装，他们有非常周全的考虑以及详细的文档，这里我们不多做考究，我们把关注的重点放在更底层的API `fetch`。

`Fetch API` 是一个用于访问和操纵HTTP管道的强大的原生 API。

这种功能以前是使用 `XMLHttpRequest` 实现的。`Fetch` 提供了一个更好的替代方法，可以很容易地被其他技术使用，例如 `Service Workers`。`Fetch` 还提供了单个逻辑位置来定义其他HTTP相关概念，例如 `CORS` 和 `HTTP` 的扩展。

可见 `fetch` 是作为 `XMLHttpRequest` 的替代品出现的。

使用 `fetch`，你不需要再额外加载一个外部资源。但它还没有被浏览器完全支持，所以你仍然需

^ 1.0.0



探索掘金



八、fetch的使用

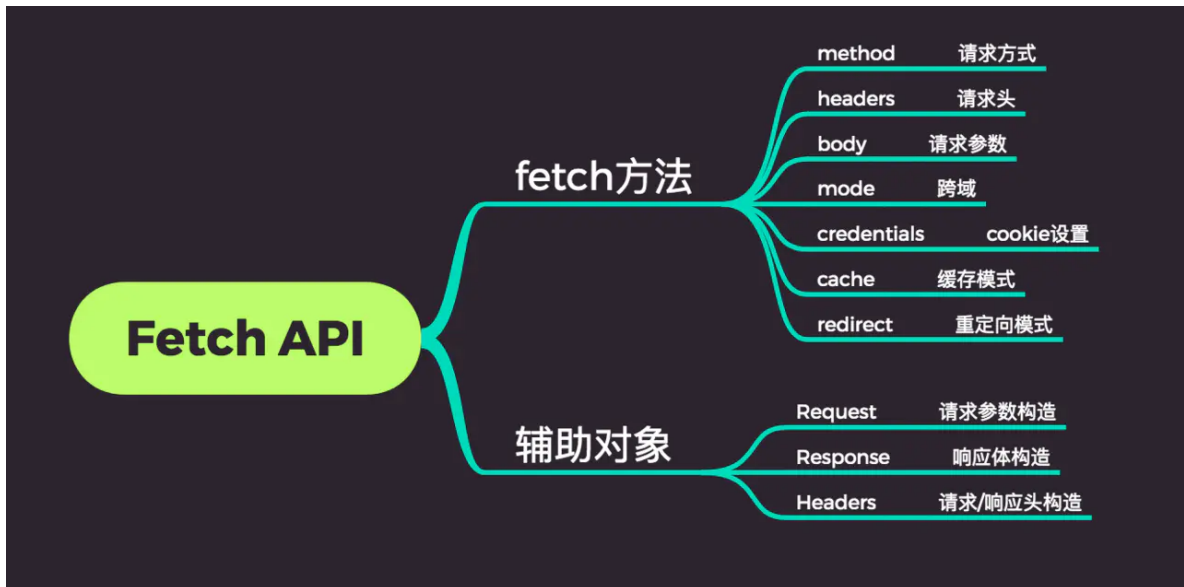
一个基本的 fetch请求：

[复制代码](#)

```
const options = {
  method: "POST", // 请求参数
  headers: { "Content-Type": "application/json" }, // 设置请求头
  body: JSON.stringify({ name: '123' }), // 请求参数
  credentials: "same-origin", // cookie设置
  mode: "cors", // 跨域
}

fetch('http://www.xxx.com', options)
  .then(function(response) {
    return response.json();
  })
  .then(function(myJson) {
    console.log(myJson); // 响应数据
  })
  .catch(function(err) {
    console.log(err); // 异常处理
  })
```

Fetch API 提供了一个全局的 `fetch()` 方法，以及几个辅助对象来发起一个网络请求。



- fetch()

`fetch()` 方法用于发起获取资源的请求。它返回一个 `promise`，这个 `promise` 会在请求响应后被 `resolve`，并传回 `Response` 对象。

[首页](#)[探索掘金](#)

可以通过 `Headers()` 构造函数来创建一个你自己的 `headers` 对象，相当于 `response/request` 的头信息，可以使你查询到这些头信息，或者针对不同的结果做不同的操作。

[复制代码](#)

```
var myHeaders = new Headers();
myHeaders.append("Content-Type", "text/plain");
```

- Request

通过 `Request()` 构造函数可以创建一个 `Request` 对象，这个对象可以作为 `fetch` 函数的第二个参数。

- Response

在 `fetch()` 处理完 `promises` 之后返回一个 `Response` 实例，也可以手动创建一个 `Response` 实例。

九、fetch polyfill源码分析

由于 `fetch` 是一个非常底层的 `API`，所以我们无法进一步的探究它的底层，但是我们可以借助它的 `polyfill` 探究它的基本原理，并找出其中的坑点。

代码结构

```
export function Headers(headers) {
  // ...
}

export function Request(input, options) {
  // ...
}

export function Response(bodyInit, options) {
  // ...
}

export function fetch(input, init) {
  // ...
}

fetch.polyfill = true

if (!self.fetch) {
  self.fetch = fetch
  self.Headers = Headers
  self.Request = Request
  self.Response = Response
}
```

[首页](#)[探索掘金](#)

fetch 封装

```
export function fetch(input, init) {  
  return new Promise(function (resolve, reject) {  
    var request = new Request(input, init)  
    var xhr = new XMLHttpRequest()  
    xhr.open(request.method, request.url, true)  
  
    xhr.onload = function () {  
      var options = {  
        status: xhr.status,  
        statusText: xhr.statusText,  
        headers: parseHeaders(xhr.getAllResponseHeaders() || '')  
      }  
      options.url = 'responseURL' in xhr ? xhr.responseURL : options.headers.get('X-Request-URL')  
      var body = 'response' in xhr ? xhr.response : xhr.responseText  
      resolve(new Response(body, options))  
    }  
  
    request.headers.forEach(function (value, name) {  
      xhr.setRequestHeader(name, value)  
    })  
  
    xhr.send()  
  })  
}
```

代码非常清晰：

- 构造一个 `Promise` 对象并返回
- 创建一个 `Request` 对象
- 创建一个 `XMLHttpRequest` 对象
- 取出 `Request` 对象中的请求 `url`，请求方法，`open` 一个 `xhr` 请求，并将 `Request` 对象中存储的 `headers` 取出赋给 `xhr`
- `xhr onload` 后取出 `response` 的 `status`、`headers`、`body` 封装 `Response` 对象，调用 `resolve`。

异常处理



首页 ▾

探索掘金



```
return new Promise(function (resolve, reject) {
  // ...
  function abortXhr() {
    xhr.abort()
  }

  xhr.onerror = function () {
    reject(new TypeError('Network request failed'))
  }

  xhr.ontimeout = function () {
    reject(new TypeError('Network request failed'))
  }

  xhr.onabort = function () {
    reject(new DOMException('Aborted', 'AbortError'))
  }

  if (request.signal) {
    request.signal.addEventListener('abort', abortXhr)
    xhr.onreadystatechange = function () {
      if (xhr.readyState === 4) {
        request.signal.removeEventListener('abort', abortXhr)
      }
    }
  }
  // ...
})
```

可以发现，调用 `reject` 有三种可能：

- 1.请求超时
- 2.请求失败

注意：当和服务器建立简介，并收到服务器的异常状态码如 `404`、`500` 等并不能触发 `onerror`。当网络故障时或请求被阻止时，才会标记为 `reject`，如跨域、`url` 不存在，网络异常等会触发 `onerror`。

所以使用`fetch`当接收到异常状态码都是会进入`then`而不是`catch`。这些错误请求往往要手动处理。

- 3.手动终止

可以在 `request` 参数中传入 `signal` 对象，并对 `signal` 对象添加 `abort` 事件监听，当 `xhr.readyState` 变为 `4`（响应内容解析完成）后将`signal`对象的`abort`事件监听移除掉。

这表示，在一个 `fetch` 请求结束之前可以调用 `signal.abort` 将其终止。在浏览器中可以使用 `AbortController()` 构造函数创建一个控制器，然后使用 `AbortController.signal` 属性

这是一个实验中的功能，此功能某些浏览器尚在开发中

```
export function Headers(headers) {  
  this.map = {}  
  
  if (headers instanceof Headers) {  
    headers.forEach(function (value, name) {  
      this.append(name, value)  
    }, this)  
  } else if (Array.isArray(headers)) {  
    headers.forEach(function (header) {  
      this.append(header[0], header[1])  
    }, this)  
  } else if (headers) {  
    Object.getOwnPropertyNames(headers).forEach(function (name) {  
      this.append(name, headers[name])  
    }, this)  
  }  
}
```

在header对象中维护了一个 `map` 对象，构造函数中可以传入 `Header` 对象、数组、普通对象类型的 `header`，并将所有的值维护到 `map` 中。

之前在 `fetch` 函数中看到调用了 `header` 的 `forEach` 方法，下面是它的实现：

```
Headers.prototype.forEach = function (callback, thisArg) {  
  for (var name in this.map) {  
    if (this.map.hasOwnProperty(name)) {  
      callback.call(thisArg, this.map[name], name, this)  
    }  
  }  
}
```

可见 `header` 的遍历即其内部 `map` 的遍历。

另外 `Header` 还提供了 `append`、`delete`、`get`、`set` 等方法，都是对其内部的 `map` 对象进行操作。

Request对象



首页 ▾

探索掘金



```
export function Request(input, options) {
  options = options || {}
  var body = options.body

  if (input instanceof Request) {
    this.url = input.url
    this.method = input.method
    // ...
  } else {
    this.url = String(input)
  }
  this.credentials = options.credentials || this.credentials || 'same-origin'
  if (options.headers || !this.headers) {
    this.headers = new Headers(options.headers)
  }
  this.method = normalizeMethod(options.method || this.method || 'GET')
  this.mode = options.mode || this.mode || null
  this.signal = options.signal || this.signal
  this.referrer = null
  // ...
  this._initBody(body)
}
```

`Request` 对象接收的两个参数即 `fetch` 函数接收的两个参数，第一个参数可以直接传递 `url`，也可以传递一个构造好的 `request` 对象。第二个参数即控制不同配置的 `option` 对象。

可以传入 `credentials`、`headers`、`method`、`mode`、`signal`、`referrer` 等属性。

这里注意：

- 传入的 `headers` 被当作 `Headers` 构造函数的参数来构造 `header` 对象。

cookie处理

`fetch` 函数中还有如下的代码：

```
if (request.credentials === 'include') {
  xhr.withCredentials = true
} else if (request.credentials === 'omit') {
  xhr.withCredentials = false
}
```

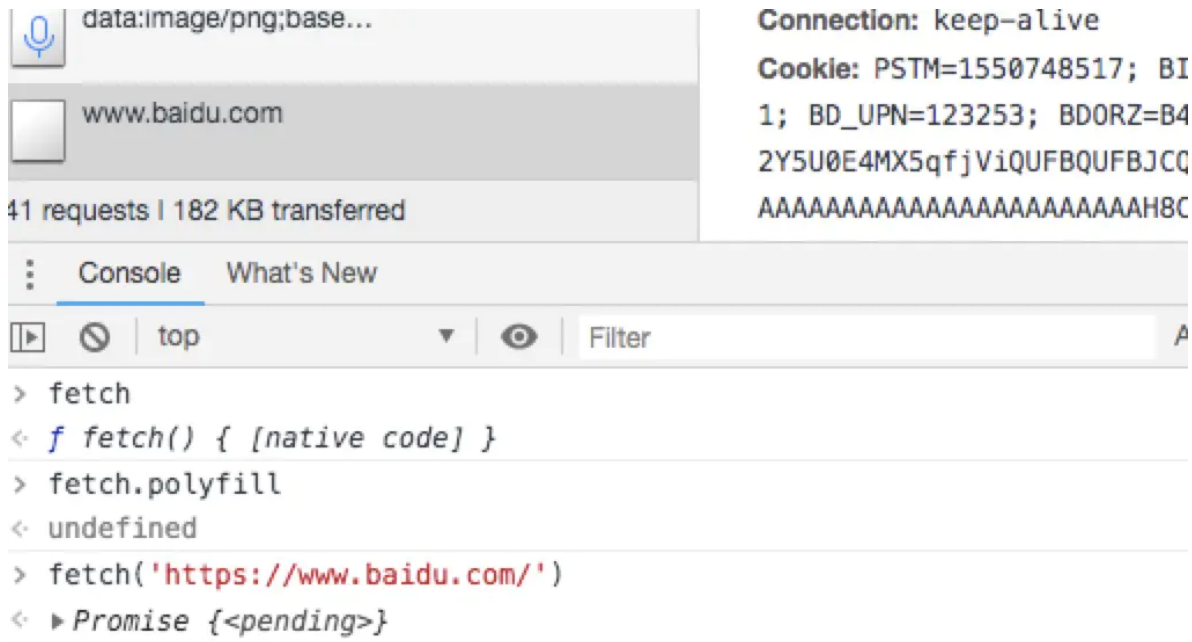
复制代码

默认的 `credentials` 类型为 `same-origin`，即可携带同源请求的 `cookie`。

然后我发现这里 `polyfill` 的实现和 [MDN-使用Fetch](#) 以及很多资料是不一致的：

mdn: 默认情况下，`fetch` 不会从服务端发送或接收任何 `cookies`

于是我分别实验了下使用 `polyfill` 和使用原生 `fetch` 携带cookie的情况，发现在不设置 `credentials` 的情况下居然都是默认携带同源 `cookie` 的，这和文档的说明说不一致的，查阅了许多资料后都是说 `fetch` 默认不会携带cookie，下面是使用原生 `fetch` 在浏览器进行请求的情况：



然后我发现在[MDN-Fetch-Request](#)已经指出新版浏览器 `credentials` 默认值已更改为 `same-origin`，旧版依然是 `omit`。

确实[MDN-使用Fetch](#)这里的文档更新的有些不及时，误人子弟了...

Response对象

`Response` 对象是 `fetch` 调用成功后的返回值：

回顾下 `fetch` 中对 `Response` 的操作：

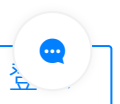
```
xhr.onload = function () {  
  var options = {  
    status: xhr.status,  
    statusText: xhr.statusText,  
    headers: parseHeaders(xhr.getAllResponseHeaders() || '')  
  }  
  options.url = 'responseURL' in xhr ? xhr.responseURL : options.headers.get('X-Request-URL')  
  var body = 'response' in xhr ? xhr.response : xhr.responseText  
  resolve(new Response(body, options))  
}
```

复制代码



首页 ▾

探索掘金



可见在构造函数中主要对 `options` 中的 `status`、`statusText`、`headers`、`url` 等分别做了处理并挂载到 `Response` 对象上。

构造函数里面并没有对 `responseText` 的明确处理，最后交给了 `_initBody` 函数处理，而 `Response` 并没有主动声明 `_initBody` 属性，代码最后使用 `Response` 调用了 `Body` 函数，实际上 `_initBody` 函数是通过 `Body` 函数挂载到 `Response` 身上的，先来看看 `_initBody` 函数：

可见，`_initBody` 函数根据 `xhr.response` 的类型（`Blob`、`FormData`、`String...`），为不同的参数进行赋值，这些参数在 `Body` 方法中得到不同的应用，下面具体看看 `Body` 函数还做了哪些其他的操作：



`Body` 函数中还为 `Response` 对象挂载了四个函数，`text`、`json`、`blob`、`formData`，这些函数中的操作就是将 `_initBody` 中得到的不同类型的返回值返回。

这也说明了，在 `fetch` 执行完毕后，不能直接在 `response` 中获取到返回值而必须调用 `text()`、`json()` 等函数才能获取到返回值。

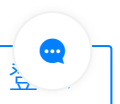
这里还有一点需要说明：几个函数中都有类似下面的逻辑：

```
var rejected = consumed(this)
if (rejected) {
  return rejected
}
```

[复制代码](#)

`consumed`函数：

```
function consumed(body) {
  if (body.bodyUsed) {
    return Promise.reject(new TypeError('Already read'))
  }
}
```

[复制代码](#)[首页](#) ▼[探索掘金](#)

每次调用 `text()`、`json()` 等函数后会将 `bodyUsed` 变量变为 `true`，用来标识返回值已经读取过了，下一次再读取直接抛出 `TypeError('Already read')`。这也遵循了原生 `fetch` 的原则：

因为 `Responses` 对象被设置为了 `stream` 的方式，所以它们只能被读取一次

十、fetch的坑点

`VUE` 的文档中对 `fetch` 有下面的描述：

使用 `fetch` 还有很多别的注意事项，这也是为什么大家现阶段还是更喜欢 `axios` 多一些。当然这个事情在未来可能会发生改变。

由于 `fetch` 是一个非常底层的 `API`，它并没有被进行很多封装，还有许多问题需要处理：

- 不能直接传递 `JavaScript` 对象作为参数
- 需要自己判断返回值类型，并执行响应获取返回值的方法
- 获取返回值方法只能调用一次，不能多次调用
- 无法正常的捕获异常
- 老版浏览器不会默认携带 `cookie`
- 不支持 `jsonp`

十一、对fetch的封装

请求参数处理

支持传入不同的参数类型：

复制代码

```
function stringify(url, data) {  
  var dataString = url.indexOf('?') == -1 ? '?' : '&';  
  for (var key in data) {  
    dataString += key + '=' + data[key] + '&';  
  };  
  return dataString;  
}
```



首页 ▾

探索掘金



```
} else if (/^get$/i.test(request.method)) {
  request.url = `${request.url}${stringify(request.url, request.data)}`;
} else if (request.form) {
  request.headers.set('Content-Type', 'application/x-www-form-urlencoded;charset=UTF-8');
  request.body = stringify(request.data);
} else {
  request.headers.set('Content-Type', 'application/json;charset=UTF-8');
  request.body = JSON.stringify(request.data);
}
```

cookie携带

`fetch` 在新版浏览器已经开始默认携带同源 `cookie`，但在老版浏览器中不会默认携带，我们需要对他进行统一设置：

```
request.credentials = 'same-origin'; // 同源携带
request.credentials = 'include'; // 可跨域携带
```

[复制代码](#)

异常处理

当接收到一个代表错误的 HTTP 状态码时，从 `fetch()` 返回的 `Promise` 不会被标记为 `reject`，即使该 HTTP 响应的状态码是 404 或 500。相反，它会将 `Promise` 状态标记为 `resolve`（但是会将 `resolve` 的返回值的 `ok` 属性设置为 `false`），仅当网络故障时或请求被阻止时，才会标记为 `reject`。

因此我们要对 `fetch` 的异常进行统一处理

```
.then(response => {
  if (response.ok) {
    return Promise.resolve(response);
  } else {
    const error = new Error(`请求失败！状态码: ${response.status}, 失败信息: ${response.statusText}`);
    error.response = response;
    return Promise.reject(error);
  }
});
```

[复制代码](#)[首页](#)[探索掘金](#)

对不同的返回值类型调用不同的函数接收，这里必须提前判断好类型，不能多次调用获取返回值的方法：

[复制代码](#)

```
.then(response => {  
  let contentType = response.headers.get('content-type');  
  if (contentType.includes('application/json')) {  
    return response.json();  
  } else {  
    return response.text();  
  }  
});
```

jsonp

`fetch` 本身没有提供对 `jsonp` 的支持，`jsonp` 本身也不属于一种非常好的解决跨域的方式，推荐使用 `cors` 或者 `nginx` 解决跨域，具体请看下面的章节。

`fetch`封装好了，可以愉快的使用了。

嗯，`axios`真好用...

十二、跨域总结

谈到网络请求，就不得不提跨域。

浏览器的同源策略限制了从同一个源加载的文档或脚本如何与来自另一个源的资源进行交互。这是一个用于隔离潜在恶意文件的重要安全机制。通常不允许不同源间的读操作。

跨域条件：协议，域名，端口，有一个不同就算跨域。

下面是解决跨域的几种方式：

nginx

使用 `nginx` 反向代理实现跨域，参考我这篇文章：[前端开发者必备的nginx知识](#)

cors

[探索掘金](#)

服务端设置 `Access-Control-Allow-Origin` 就可以开启 `CORS` 。该属性表示哪些域名可以访问资源，如果设置通配符则表示所有网站都可以访问资源。

[复制代码](#)

```
app.all('*', function (req, res, next) {
  res.header("Access-Control-Allow-Origin", "*");
  res.header("Access-Control-Allow-Headers", "X-Requested-With");
  res.header("Access-Control-Allow-Methods", "PUT,POST,GET,DELETE,OPTIONS");
  next();
});
```

jsonp

`script` 标签的 `src` 属性中的链接可以访问跨域的 `js` 脚本，利用这个特性，服务端不再返回 `JSON` 格式的数据，而是返回一段调用某个函数的 `js` 代码，在 `src` 中进行了调用，这样实现了跨域。

`jquery` 对 `jsonp` 的支持：

[复制代码](#)

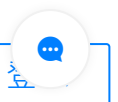
```
$.ajax({
  type : "get",
  url : "http://xxx"
  dataType: "jsonp",
  jsonp:"callback",
  jsonpCallback: "doo",
  success : function(data) {
    console.log(data);
  }
});
```

`fetch`、`axios` 等并没有直接提供对 `jsonp` 的支持，如果需要使用这种方式，我们可以尝试进行手动封装：

[复制代码](#)

```
(function (window,document) {
  "use strict";
  var jsonp = function (url,data,callback) {

    // 1. 将传入的data数据转化为url字符串形式
    // {id:1,name:'jack'} => id=1&name=jack
    var dataString = url.indexOf('?') == -1? '?': '&';
    for(var key in data){
      dataString += key + '=' + data[key] + '&';
    }
  }
```

[首页](#)[探索掘金](#)

```
// cbFuncName回调函数的名字 : my_json_cb_ 名字的前缀 + 随机数 (把小数点去掉)
var cbFuncName = 'my_json_cb_' + Math.random().toString().replace('.', '');
dataString += 'callback=' + cbFuncName;

// 3. 创建一个script标签并插入到页面中
var scriptEle = document.createElement('script');
scriptEle.src = url + dataString;

// 4. 挂载回调函数
window[cbFuncName] = function (data) {
    callback(data);
    // 处理完回调函数的数据之后, 删除jsonp的script标签
    document.body.removeChild(scriptEle);
}

document.body.appendChild(scriptEle);
}

window.$jsonp = jsonp;

})(window, document)
```

postMessage跨域

`postMessage()` 方法允许来自不同源的脚本采用异步方式进行有限的通信, 可以实现跨文本档、多窗口、跨域消息传递。

[复制代码](#)

```
//捕获iframe
var domain = 'http://scriptandstyle.com';
var iframe = document.getElementById('myIFrame').contentWindow;

//发送消息
setInterval(function(){
    var message = 'Hello! The time is: ' + (new Date().getTime());
    console.log('blog.local: sending message: ' + message);
    //send the message and target URI
    iframe.postMessage(message, domain);
}, 6000);
```

[复制代码](#)

```
//响应事件
window.addEventListener('message', function(event) {
    if(event.origin !== 'http://davidwalsh.name') return;
    console.log('message received: ' + event.data, event);
```

[首页](#)[探索掘金](#)

`postMessage` 跨域适用于以下场景：同浏览器多窗口间跨域通信、`iframe` 间跨域通信。

WebSocket

`WebSocket` 是一种双向通信协议，在建立连接之后，`WebSocket` 的 `server` 与 `client` 都能主动向对方发送或接收数据而不受同源策略的限制。

[复制代码](#)

```
function WebSocketTest(){
    if ("WebSocket" in window){
        alert("您的浏览器支持 WebSocket!");
        // 打开一个 web socket
        var ws = new WebSocket("ws://localhost:3000/abcd");
        ws.onopen = function(){
            // Web Socket 已连接上，使用 send() 方法发送数据
            ws.send("发送数据");
            alert("数据发送中...");
        };
        ws.onmessage = function (evt) {
            var received_msg = evt.data;
            alert("数据已接收...");
        };
        ws.onclose = function(){
            // 关闭 websocket
            alert("连接已关闭...");
        };
    } else{
        // 浏览器不支持 WebSocket
        alert("您的浏览器不支持 WebSocket!");
    }
}
```

文章首发

想阅读更多优质文章，或者需要文章中思维导图源文件可关注我的[github博客](#)，欢迎star🌟。

推荐大家使用[Fundebug](#)，一款很好用的BUG监控工具~

文中如有错误，欢迎在评论区指正，谢谢阅读。

关注公众号后回复【加群】拉你进入优质前端交流群。

关注下面的标签，发现更多相似文章

JavaScript

ConardLi Lv6

fe @ 字节跳动 (大量HC 欢迎来撩...
获得点赞 23,993 · 获得阅读 865,325

关注

安装掘金浏览器插件

打开新标签页发现好内容，掘金、GitHub、Dribbble、ProductHunt 等站点内容轻松获取。快来安装掘金浏览器插件获取高质量内容吧！

输入评论...

czpcalm Lv2

原生AJAX函数那里send(param)解释参数param为请求的参数有问题吧。是send([body])请求的主体才对

1月前



回复

10a55210067d11e...

受教了。



首页 ▾

探索掘金



莹本尊44210 底层打字员

if(xhr.status >= 200 && xhr.status < 300) || xhr.status == 304)

这行代码少了一个括号

1年前



回复

山口谈退隐去了 一只小菜鸟

请问fetch章节中，一个基本的fetch请求的代码事例里，const的options，在下面的fetch中并没有被使用，理解不了啊？求教大神

1年前



回复

ConardLi Lv6 (作者) fe @ 字节跳动 (大...

忘记传参了，已经修正了~

1年前

hongxb

iframe ?

1年前



回复

ConardLi Lv6 (作者) fe @ 字节跳动 (大...

很久以前使用iframe达到局部刷新的效果

1年前

女留微信男自强 砌墙师傅

回复 ConardLi: 此时的qq邮箱也是这个套路。

1年前

博客 全栈开发 @ found...

回复 ConardLi: 是在说iframe吧 🤔

1年前

恪晨 Lv1 小前端 @ 小公司

回复 ConardLi: 单词单词 🤔

1年前

冷石Boy Lv3 上单 @ 艾欧尼亚

赞一个 🙌

1年前

1

回复

鲁大师的故事 Lv1 前端工程师 @ 磁贝

不过async+await语法糖我觉得也可以加下，那个用起来也很舒服

1年前

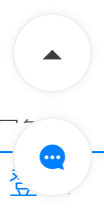


回复



首页 ▾

探索掘金



一直在找这方面的总结，感谢

1年前



回复

向大佬致敬 前端小学生

还是axios好用呀

1年前



回复

我可能会飞 Lv1

好文呀，楼主加油继续写

1年前

1

回复

workingWOW 前端 @ 腾讯科技

嗯，axios真好用...

1年前

2

回复

ConardLi Lv6 (作者) fe @ 字节跳动 (大...



1年前

yeyan1996 Lv5 less is more @ byteda...

fetch的使用那一章,fetch少了第二个参数

const options = {

method: "POST", // 请求参数

headers: { "Content-Type": "application/json"}, // 设置请求头

body: JSON.stringify({name:'123'}), //...

[展开](#)

1年前

1

回复

ConardLi Lv6 (作者) fe @ 字节跳动 (大...

谢谢，已更正

1年前

我真的好想学习啊 Lv2 前端架构师 @ 腾...

回复 ConardLi:                                     

1年前

fe前端coder 前端开发 @ 猪场
到位
1年前

1 回复

相关推荐

梨香 · 4天前 · JavaScript
别光知道用console.log调试了，快来试试这些高效的调试方法！
 409 39

笨笨橙 · 18小时前 · JavaScript
JavaScript常见的内存泄漏
 20 1

网易云音乐大前端团队 · 3天前 · 函数式编程 / JavaScript
函数式编程进阶：应用函子
 122 14

前端格局 · 1天前 · 前端 / JavaScript
如何突破自己的技术瓶颈期
 44 8

编码小孩 · 16小时前 · JavaScript
underscore之防抖函数
 5 1

JavaGuide · 3天前 · JavaScript / Java
这个项目可以让你在几分钟快速了解某个编程语言
 78 3

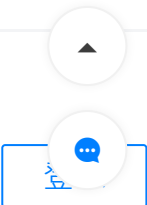
axuebin · 8天前 · JavaScript / 前端
2020 年前端分类推荐书单【结合思维导图】
 321 35

_风清洋 · 11天前 · JavaScript



首页 ▾

探索掘金




yck · 9天前 · JavaScript / 前端

前端前沿观察 · Cookie 居然可以这样整了

 238  44

Daniel-Fang · 18小时前 · JavaScript

不得不知道的JavaScript原生拖放实现

 5 