

第 11 章 使用 JavaScript 实现浏览器的动态 HTTP 请求

[日] 涩川喜规 · 详解HTTP：协议基础与Go语言实现

本章将介绍 JavaScript 环境提供的一些 API。为了让大家理解浏览器的行为背后所进行的 HTTP 通信，前面的章节重点介绍了浏览器的动作，比如第一次加载页面时的 GET 请求、Cookie 和缓存等的相关内容。

除全局导航之外，其他的动态服务器访问都是使用 JavaScript 实现的。JavaScript 可以实现的操作越广泛，网站实现中能够使用的功能就越多。JavaScript 是拓宽 Web 可能性必不可少的一种手段。本章将介绍浏览器上的 JavaScript 所拥有的可以进行各种通信及 HTTP 相关处理的 API。

11.1 浏览器的 HTTP 与生命周期

浏览器执行 HTTP 访问的地方主要有两处，分别是加载网站和加载后使用 JavaScript 进行访问。

在地址栏中输入 URL，点击 `<a>` 标签的链接，执行表单发送等操作后，浏览器就会获取新的内容。该访问是通过 HTTP 执行的。

浏览器查看返回的内容的 `Content-Type`，如果能显示该内容，就清除整个画面，显示该内容。这种重置浏览器画面的请求称为全局导航。当内容是 HTML 时，其引用的样式表、图像、Web 字体和 JavaScript 等文件也是通过 HTTP 请求并下载的。

如果返回的内容中存在 `Content-Disposition` 首部，指示下载的内容不显示在浏览器中，那么将不清除画面，将内容下载到计算机的“下载”文件夹等中。

使用 JavaScript 的 API 也可以执行 HTTP 访问。在画面显示结束、执行 JavaScript 时，如果其中使用了进行通信的功能，就也会进行通信。在 XMLHttpRequest、Fetch 和 Server Sent Events 的情况下进行的是 HTTP 通信，否则进行 WebSocket 或 WebRTC 通信。使用 JavaScript 实现的通信所获取的内容由 JavaScript 处理。

11.2 XMLHttpRequest

XMLHttpRequest 最早由 Microsoft 引入，后来实现了标准化。XMLHttpRequest 在技术上能够实现 Ajax，由此形成了新的 Web 表现形式，掀起了被称为 Web 2.0 的运动。第 4 章介绍了 XMLHttpRequest 的历史，本节将介绍它的用法。

复制代码

```
1 var xhr = new XMLHttpRequest();
2 xhr.open("GET", "/json", true);
3 xhr.onload = function () {
4     // 返回响应时调用的方法
5     if (xhr.status === 200) {
6         // 解析 JSON 并显示
7         console.log(JSON.parse(xhr.responseText));
8     }
9 };
10 xhr.setRequestHeader("MyHeader", "HeaderValue")
11 xhr.send();
```

URL 是相对地址，所以上面的代码不会实际运行，它与下面的模拟命令表示的含义相同。

复制代码

```
1 $ curl -H "MyHeader=HeaderValue" /json
```

`open()` 方法用于指定方法和发送目的地。如果将第 3 个参数设置为 `true`，则变为异步执行。在同步执行的情况下，`send()` 方法在响应返回之前不会结束。而且在发生阻塞后，用户的操作不会得到响应，这时大家会觉得网站的反应速度太慢。因此，相信不会有人使用 `true` 之外的值。

使用 `send()` 方法开始发送数据。将数据传递给该方法，该方法就会将数据发送给服务器。


复制代码

```
1 var xhr = new XMLHttpRequest();
2 xhr.open("POST", "/json", true);
3 xhr.onload = function () {
```

```
4 // 返回响应时调用的方法
5 }
6 xhr.send(JSON.stringify({"message": "hello world"}));
```

curl 中使用 `-d` 传递数据。要想从文件中读取要发送的 JSON，需要使用 `@` 指定文件名，或者使用 `-T`。

```
1 $ curl -d @send.json /json
2
3 $ curl -T send.json /json
```

 复制代码

另外，XMLHttpRequest 存在两个版本，这里介绍的是名为 Level 2 的新版本。上网查一下 XMLHttpRequest 就可以找到许多关于旧版本的信息。虽然两个版本在后面介绍的安全问题上存在差别，但二者的设计框架和思想基本相同，因此本书只介绍 Level 2。

处理各种数据格式

XMLHttpRequest 支持许多数据格式。通过在 `responseType` 中设置字符串，可以指定在 `response` 中存储哪种对象的返回值。下面的代码用于接收持有文件种类和二进制数据的 `Blob` 对象，并使用 `img` 标签进行显示。

```

1 xhr.responseType = 'blob';
2 xhr.onload = function(e) {
3     if (this.status == 200) {
4         var blob = this.response;
5         var img = document.createElement('img');
6         img.onload = function(e) {
7             window.URL.revokeObjectURL(img.src);
8         };
9         img.src = window.URL.createObjectURL(blob);
10        document.body.appendChild(img);
11        :
12    }
13 };
14 xhr.send();

```

`responseType` 中可以设置的字符串如表 11-1 所示。

表 11-1 `responseType` 中可以设置的字符串

设置值	数据类型
<code>arraybuffer</code>	ArrayBuffer
<code>blob</code>	Blob
<code>document</code>	Document
<code>json</code>	JSON
<code>xml</code>	XML
<code>text</code>	string (默认)

在未设置 `responseType` 的情况下，如果数据是 XML 的 MIME 类型（`text/xml`、`application/xml`），就以字符串形式存储在 `responseXML` 中，否则存储在 `responseText` 中。如果服务器未设置 MIME 类型，则由客户端指定 MIME 类型。

11.3 FetchAPI


Fetch API 是实现了标准化的 API，比 XMLHttpRequest 更易使用，XMLHttpRequest 能做的事情也都可以由 Fetch API 实现。除此之外，Fetch API 还可以实现缓存控制等细致的内容。

本节将介绍 Fetch API 的使用方法。

11.3.1 FetchAPI 的基础内容

代码清单 11-1 是使用 Fetch API 的示例代码。

代码清单 11-1 使用 Fetch API 的 JavaScript 示例代码


 复制代码

```
1  const response = await fetch("news.json", { // ❶ ❸
2      method: 'GET',                        // ❷
3      mode: 'cors',
4      credentials: 'include',
5      cache: 'default',
6      headers: {
7          'Content-Type': 'application/json'
8      }
9  });
10 console.log(response.status);              // ❹
11 if (response.ok) {                        // ❺
12     const json = await response.json();    // ❻
13     console.log(json);
14 }
```

- ❶ 和 XMLHttpRequest 一样，不创建对象，而是调用 `fetch()` 函数。
- ❷ `fetch()` 函数的第二个参数是可选对象（可省略）。
- ❸ 由于 `fetch()` 是异步处理，所以加上 `await` 以等待服务器响应。
- ❹ 服务器响应中包含首部和状态码。`status` 属性中包含状态码。
- ❺ 当状态码是 200 ~ 299 的正常响应时，`ok` 属性为 `true`。
- ❻ 由于响应主体的读取和解析比较耗时，所以加上 `await` 进行等待。

下面的示例代码中省略了所有的选项，仅使用 GET 读取 JSON，代码很短。

```
1 const response = await fetch("news.json");
2 if (response.ok) {
3     const json = await response.json();
4     console.log(json);
5 }
```

 复制代码

JavaScript 中使用 `await` 来等待比较耗时的处理的结果。`fetch()` 等函数在执行繁重（需要等待）的处理时会返回 `Promise` 对象，`Promise` 对象根据结束时的返回值来判断处理是否正常结束。`Promise` 是表示“稍后通知处理结束的结果哦”的“约定”。将 `await` 添加到返回 `Promise` 的函数之前，以等待约定达成。

`fetch()` 只处理一个来回的通信，所以看起来只有一处处理比较繁重，但其实有两处。

进行 HTTP 通信，等待服务器返回响应，完成最开始的首部部分的解析
然后读取整个主体

11.3.2 Fetch 的选项

前面的示例代码采用了 JSON 形式来读取数据，Fetch API 支持的数据类型如表 11-2 所示。

表 11-2 Fetch API 支持的数据类型

方法	类型	说明
<code>arrayBuffer()</code>	<code>ArrayBuffer</code>	固定长度的二进制数据。可以使用 <code>Typed Array</code> 进行读写

我们可以使用选项来自定义各种项目。如表 11-3 所示，有一部分方法无法使用。另外，`GET`、`HEAD`、`POST` 这 3 个方法会以 `CORS` 安全的方式处理。

表 11-3 Fetch API 中可使用的方法

项目	方法
CORS 安全	<code>GET</code> 、 <code>HEAD</code> 、 <code>POST</code>
禁止方法	<code>CONNECT</code> 、 <code>TRACE</code> 、 <code>TRACK</code>

作为安全对策，Fetch API 通过如表 11-4 所示的设置值来表示 `CORS` 模式。在 `XMLHttpRequest` 的情况下，模式无法修改。

表 11-4 Fetch API 的 CORS 模式

设置值	Fetch 默认	XHR 默认	说明
<code>cors</code>		√	允许访问其他源服务器
<code>same-origin</code>			在访问其他源服务器时发生错误
<code>no-cors</code>	√		无视 <code>CORS</code> 连接，返回空响应

除此之外，可设置的值还有 `navigate`、`websocket`、`cors-with-forced-preflight` 等。

对 Cookie 的限制是在 `credentials` 中设置的，能够设置的值如表 11-5 所示。在 `XMLHttpRequest` 中，将 `withCredential` 属性设置为 `true` 与设置 `include` 的效果是一样的。

表 11-5 Fetch API 的 `credentials` 中可以设置的值

设置值	Fetch 默认	XHR 默认	说明
<code>omit</code>	√		不发送 Cookie
<code>same-origin</code>		√	仅在同源的情况下发送 Cookie
<code>include</code>			发送 Cookie

Web 的发展是速度和安全性两方面提升的结果，Fetch API 属于后者。它的设计理念是，默认选择更严格的设置，然后根据需要显式解除该设置。

11.3.3 创建和解析查询参数

在使用 `GET` 方法发送数据时，可以使用 `URLSearchParams` 创建和解析参数。虽然现在仍有很多示例代码是使用历史悠久的 `encodeURIComponent()` 编写的，但大家尽量还是使用 `URLSearchParams` 比较好。这样一来，这些代码也可以用于 Internet Explorer 之外的浏览器。

`set()` 方法可以用来向查询参数中添加数据。除此之外，也可以使用 `append()` 方法。查询参数类似于编程语言中提供的映射、散列和字典等数据结构，一个键对应多个值。使用 `append()` 可以为一个键设置多个值（代码清单 11-2）。

代码清单 11-2 查询参数的结构

复制代码


```
1 const params = new URLSearchParams();
2 // 使用 set 添加数据
3 params.set("name", " 惠比寿东公园");
```



```
4 // 也可以使用 append 添加数据
5 params.append("hasTako", "true");
6 // 使用 toString() 获取字符串格式的数据
7 console.log(params.toString());
8 // name=%E6%81%B5%E6%AF%94%E5%AF%BF%E6%9D%B1%E5%85%AC%E5%9C%92&hasTako=true
```

查询参数的解析使用同一个类。将查询参数的字符串设置到 `constructor` 属性中，可以创建一个已经设置好值的实例，然后就可以使用该实例的 `has()` 方法或 `get()` 方法来检查键是否存在，并获取值（代码清单 11-3）。

代码清单 11-3 解析查询参数


 复制代码

```
1 const params = new URLSearchParams(location.search);
2 // 使用 has 判断指定的键是否存在
3 params.has("name");
4 // 使用 get 获取指定键的值
5 params.get("hasTako");
```

当同一个键包含多个值时，可以使用 `getAll()` 方法获取一个包含所有值的数组。

JavaScript 自带的 `Map` 结构与方法名一样用起来很方便。我们也可以使用 `for` 循环来一并获取键和值（代码清单 11-4）。

代码清单 11-4 使用循环获取键和值

 复制代码

```
1 for (const [key, value] of params.entries()) {
2     console.log(`${key}: ${value}`);
3 }
```

11.3.4 发送主体

本章开头的示例代码中包含了 HTTP 的 4 个基本元素中的以下 3 个。另外，上一节还介绍了路径元素中查询参数的相关内容。

方法和路径

首部


状态码

而主体可以通过 Fetch API 的第 2 个参数来实现。

application/x-www-form-urlencoded 形式的表单发送

首先，我们按表单的默认发送方式（无法发送文件），使用 JavaScript 发送相同的内容。

我们试着从浏览器发送第 2 章的示例代码中的 curl 命令，如下所示。

 复制代码

```
1 $ curl --http1.0 -d title="The Art of Community" -d author="Jono Bacon" http://  
2 localhost:18888
```

严格来讲，`application/x-www-form-urlencoded` 的处理与其存在不同之处，但它们的 URL 都使用了百分号编码。因此，通过 `URLSearchParams`，我们就可以使用 JavaScript 来模拟表单的发送（代码清单 11-5）。

代码清单 11-5 以 application/x-www-form-urlencoded 形式发送

复制代码

```
1 const form = new URLSearchParams();
2 form.set("title", "The Art of Community");
3 form.set("author", "Jono Bacon");
4
5 const res = await fetch("/post", {
6   method: "POST",
7   body: form
8 });
```

multipart/form-data 形式的表单发送

我们也可以使用 Fetch 来实现 multipart/form-data 形式的表单发送。

复制代码

```
1 $ curl --http1.0 -F title="The Art of Community" -F author="Jono Bacon" -F
2 attachment-file=@test.txt http://localhost:18888
```

将 URLSearchParams 替换为 FormData 基本上就可以了（代码清单 11-6）。


代码清单 11-6 以 multipart/form-data 形式发送

复制代码

```
1 const form = new FormData();
2 form.set("title", "The Art of Community");
3 form.set("author", "Jono Bacon");
4 // 添加文件
5 const content = "Hello World";
6 const blob = new Blob([content], { type: "text/plain" });
7 form.set("attachement-file", blob, "test.txt");
8
9 const res = await fetch("/post", {
10   method: "POST",
11   body: form
12 });
```

虽然这里使用 JavaScript 创建了相当于文件内容的内容，但我们也可以添加本地文件进行发送。不过，根据安全方面的要求，浏览器不可以自行查找本地文件。浏览器只可以访问经用户允许的文件。具体来讲，就是使用 `<input>` 标签创建文件选择器，访问用户选择的指定文件（代码清单 11-7）。

代码清单 11-7 发送使用 input 标签的选择器选择的文件


 复制代码

```
1 const fileElement = document.querySelector("input[type=file]");
2 form.set("attachement-file", fileElement.files[0]);
```

发送和接收 JSON

近年来，作为 Web 应用程序的主角的 JSON 的发送操作也很简单。如果发送字符串，那么我们只要指定合适的 `Content-Type` 就可以了（代码清单 11-8）。

代码清单 11-8 以 JSON 形式发送

 复制代码

```
1 const res = await fetch("/post", {
2   method: "POST",
3   headers: {
4     "Content-Type": "application/json"
5   },
6   body: JSON.stringify({
7     "title": "The Art of Community",
8     "author": "Jono Bacon"
9   })
10 });
```

11.3.5 只有 FetchAPI 可以执行的操作

缓存控制

在介绍 Fetch API 的特征时，人们最常提到的就是缓存。Fetch API 可以对缓存进行控制，其中，可设置的值如表 11-6 所示。

表 11-6 Fetch API 的缓存控制中可以设置的值

设置值	默认	说明
<code>default</code>	√	遵循标准的浏览器动作
<code>no-store</code>		请求时忽略缓存，也不缓存结果
<code>reload</code>		与浏览器的重新加载一样，请求时忽略缓存，不发送 ETag 等。如果可以缓存，则缓存结果
<code>no-cache</code>		即使缓存未过期，也发送 HTTP 请求，同时发送本地缓存的 ETag 等，如果服务器返回 304，则使用缓存的内容
<code>force-cache</code>		即使缓存过期，也可以使用。如果没有缓存，则发送 HTTP 请求
<code>only-if-cached</code>		即使缓存过期，也可以使用。如果没有缓存，则报错

在使用 `no-store`、`reload`、`no-cache` 的情况下，无论缓存状态如何，浏览器都会强行发起请求。在使用 `no-cache` 的情况下，由于会发送缓存相关的信息，所以在不发送主体的情况下，也可能会收到 304 Not Modified。

相反，主动使用缓存的是 `force-cache` 和 `only-if-cached`。即使超过了 Max-Age 首部指定的期限，也会使用缓存。`only-if-cached` 在没有缓存的情况下会报错，因此不会向外部发送请求。

重定向控制

在重定向控制方面，可设置的值如表 11-7 所示。

表 11-7 Fetch API 的重定向控制中可以设置的值

设置值	默认	说明
<code>follow</code>	√	跟随重定向（最多 20 个重定向）
<code>manual</code>		不跟随重定向，仅表明存在重定向
<code>error</code>		抛出网络错误

当值为 `manual` 时，如果存在重定向，那么返回的就不是响应本身，而是对响应进行包装、过滤后的内容。该响应的 `type` 属性中仅存储 `opaqueredirect` 字符串，其他信息都被过滤掉了，这是因为重定向的过程中包含了不可以泄露的 URL 和首部。由于主体为 `null`，状态码为 `0`，而且无法获取首部，所以我们只能知道“存在重定向”这件事。与 `error` 不同，该值不会报错。

支持 Service Worker

有一些功能能够通过 Fetch API 实现，但无法通过 XMLHttpRequest 实现，其中最典型的的就是 Service Worker。当前的规范规定，当从 Service Worker 内部连接外部服务时，仅可以使用 Fetch API。

以 Google 为中心的组织提出了让 Web 拥有应用程序的功能特性的 PWA（Progressive Web App，渐进式网页应用）技术。Service Worker 作为 PWA 的核心，是以控制应用程序的生命周期和通信内容为目的开发的。支持 Service Worker 的 Web 服务可以离线运行，也可以处理通知。Service Worker 是在 Web 服务的前端 JavaScript 和服务器之间运行的中间层。


11.3.6 使用 FetchAPI 时常见的错误

Fetch API 提供了灵活易用的 API，所谓灵活，就是即使设置上有些许错误，也能得到正确的结果。

首部的 `Content-Type` 中忘记设置 `application/json` 是发送 JSON 时的常见错误之一。在这种情况下，虽然可以发送 JSON，但实现比较严格的服务器有时无法判断内容，从而发生错误。除此之外，由于 JSON 无法自动转换为文本，所以忘记设置 `JSON.stringify()` 也会引发错误。

另外，我们还要注意浏览器默认并不发送 Cookie。要想发送 Cookie，第 2 个参数必须设置为 `credentials: true`（代码清单 11-9）。


代码清单 11-9 以 JSON 形式发送

 复制代码

```
1 const res = await fetch("/post", {
2   method: "POST",
3   credentials: true,
4   headers: {
5     "Content-Type": "application/json"
6   },
7   body: JSON.stringify({
8     "title": "The Art of Community",
9     "author": "Jono Bacon"
10  })
11 });
```

在 API 需要认证的情况下，如果不加上 `credentials`，服务器就会返回错误。而且，如果未使用 `Response` 的 `ok` 属性确认状态码，`json()` 方法就会抛出异常。该方法内部会直接调用 `JSON.parse()`。当状态码为 403 时，如果解析 `Forbidden` 字符串，就会发生 “Unexpected token F in JSON at position 0” 的错误（代码清单 11-10）。

代码清单 11-10 以 JSON 形式发送

 复制代码

```
1 const json = await response.json();
2 // SyntaxError: Unexpected token F in JSON at position 0
```

11.3.7 在浏览器之外的 JavaScript 环境中使用 Fetch API

Node.js 默认不支持 Fetch API，但可以通过 npm 中的 `node-fetch` 包使用 Fetch API。如果使用 `isomorphic-fetch`、`isomorphic-unfetch`、`cross-fetch` 等包，则浏览器和 Node.js 可以通过相同的代码来使用 Fetch API。除此之外，`cross-fetch` 使得 React Native 也可以使用 Fetch API。


浏览器会显示某个网站，Fetch API 也可以指定访问相对路径或无域名的绝对路径（以 `/` 开头）。在这种情况下，我们可以根据浏览器中显示的 URL 导出访问目的地。由于 Node.js 中没有当前路径的概念，所以 Fetch API 中只能指定包含域名的 URL。这一点请大家注意。

11.4 使用 JavaScript 重新加载浏览器时的 HTTP 访问

前面介绍了通过 XMLHttpRequest 或 Fetch API 可以使用 JavaScript 进行通信，并处理通信结果。这样一来，即使浏览器未进行全局导航，也能获取或发送数据，避免画面在切换时闪烁（页面一瞬间全白）。


不过，有时我们也希望与用户在地址栏中输入 URL 或者点击链接等时一样，实现 HTTP 访问，清除浏览器当时的状态并加载内容。

最简单的方法是在 `location.href` 中设置要跳转到的 URL。

 复制代码

```
1 location.href = "https://tako.***.com";
```

这时只可以使用 GET 方法发起请求。另一种方法是使用 JavaScript 动态创建表单，然后调用 `submit()` 方法。如果将 `method` 属性设置为 `"post"`，就可以使用 POST 方法发起请求。

 复制代码

```
1 const form = document.createElement("form");
2 form.action = "https://tako.***.com";
3 form.method = "post";
```



```
4 document.body.appendChild(form);
5 form.submit();
6 document.body.removeChild(form);
```

11.5 下载文件

当存在 `Content-Type` 无法表示的内容，或使用 `Content-Disposition` 首部指示下载时，浏览器会进行下载，但不重置画面。当下载内容可以在浏览器中显示时，我们可以使用带有 `download` 属性的 `<a>` 标签来实现。我们还可以使用 `JavaScript` 来触发下载。

复制代码

```
1 const anchor = document.createElement("a");
2 anchor.href = "https://tako.xxxx.com";
3 anchor.download = "tako.json";
4 document.body.appendChild(anchor);
5 anchor.click();
6 document.body.removeChild(anchor);
```

如果下载时需要设置特殊的首部，或者必须使用 `GET` 之外的发送方法，那么我们可以使用 `Fetch` 进行下载。

复制代码

```
1 const res = await fetch("https://tako.xxxx.com", {
2   headers: {
3     Authorization: "Basic XXXXX"
4   }
5 });
6 if (res.ok) {
7   const anchor = document.createElement("a");
8   anchor.href = URL.createObjectURL(await res.blob());
9   anchor.download = "tako.json";
10  document.body.appendChild(anchor);
11  anchor.click();
12  URL.revokeObjectURL(anchor.href);
13  document.body.removeChild(anchor);
14 }
```


动态创建内容并下载

借助第 4 章介绍的 Data URI 方案，我们可以使用 JavaScript 动态创建文件。虽然可以在 HTML 中直接插入内容（图像等）并使用，不过这里我们来尝试下载 Excel 文件。

代码清单 11-11 中使用 SheetJS 库来创建 Excel 文件。虽然也可以自行创建 Data URI 方案形式的字符串，不过这里使用了高级功能 `URL.createObjectURL()` 来生成。创建 `<a>` 标签并设置其属性，然后点击标签设置的链接，就可以下载文件了 1。

这样一来，即使不进行 HTTP 通信，浏览器也可以远程下载文件。

代码清单 11-11 使用 JavaScript 创建并下载文件

 复制代码

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="UTF-8">
5   <script src="https://unpkg.com/xlsx/dist/xlsx.full.min.js"></script>
6 </head>
7 <body>
8   <button onclick="download()">Download XLSX</button>
9   <script>
10     function download() {
11       const contentType = "application/vnd.openxmlformats-officedocument.
12 spreadsheetml.sheet";
13       const wb = XLSX.utils.book_new();
14       const ws = XLSX.utils.aoa_to_sheet([
15         [" 瑞穗银行", "0001"],
16         [" 三菱 UFJ 银行", "0005"],
17         [" 三井住友银行", "0009"],
18         [" 理索纳银行", "0010"],
19         [" 埼玉理索纳银行", "0017"],
20       ]);
21       XLSX.utils.book_append_sheet(wb, ws, "Bank Codes");
22       const xlsx = XLSX.write(wb, { type: "array" });
23       const dataUri = URL.createObjectURL(new Blob([xlsx], {
24         type: contentType
25       }));
26       const a = document.createElement("a");
27       a.href = dataUri;
28       a.download = "bankcode.xlsx";
```


```
29         a.click();
30         URL.revokeObjectURL(url);
31     }
32 </script>
33 </body>
34 </html>
```

11.6 Server-SentEvents

与 XMLHttpRequest 和 Fetch API 不同，Server-Sent Events 是实现从服务器端开始通信的协议和 JavaScript API。另外，基于 Server-Sent Events 的通信并不是进行一次就结束了，而是一直保持连接，服务器能够多次返回响应。Server-Sent Events 还拥有断开后重新连接、读取未接收完的内容等功能。

JavaScript 端使用 `EventSource` 类来访问 Server-Sent Events。该类负责传递事件流、在连接中断时重新连接等。代码清单 11-12 是 Mozilla 网站的 Server-Sent Events 的示例代码。

代码清单 11-12 Server-Sent Events 的示例代码（摘自 Mozilla 网站）

 复制代码

```
1  const evtSource = new EventSource("ssedemo.php");
2
3  // 消息的事件处理器
4  evtSource.onmessage = (e) => {
5      const newElement = document.createElement("li");
6      newElement.innerHTML = "message: " + e.data;
7      eventList.appendChild(newElement);
8  }
9  };
10
11 evtSource.addEventListener("ping", (e) => {
12     const newElement = document.createElement("li");
13
14     const obj = JSON.parse(e.data);
15     newElement.innerHTML = "ping at " + obj.time;
16     eventList.appendChild(newElement);
17 }, false);
```

代码清单 11-12 中设置了两种事件处理器 (event handler)。`onmessage` 是在接收到无事件标签的 `data` 标签的消息时调用的回调函数。`addEventListener()` 指定事件名称并注册回调函数, 它仅处理特定的带有事件标签的消息。

重新连接和保持连接的处理在 JavaScript 无法查看的层中完成。客户端记录消息 ID, 在重新连接时, 将最后接收到的 ID 作为 `Last-Event-ID` 首部发送。服务器通过查看该首部, 就能够知道客户端成功地接收到了前面的事件, 从而只发送此后的事件。

11.7 WebSocket

WebSocket 是提供双向通信功能的 API, 该功能与 HTTP 底层的 TCP 套接字相似。在 HTTP 通信中, 首部的数据量比较大, 在频繁处理比较小的数据时, 开销会变大。WebSocket 会消除这种开销, 在频繁处理比较小的数据时更高效地进行传输, 并且既可以从客户端发送数据, 也可以从服务器发送数据。

JavaScript 的 API 类似于 TCP 套接字的 API。在通信时, 由客户端发起连接请求, 服务器处于等待接收的状态。

1. 使用指定的 IP 地址、端口号启动服务器 (`Listen`)。
2. 客户端 (浏览器) 声明开始与服务器进行通信 (`Connect`)。
3. 服务器接收来自客户端的连接请求 (`Accept`)。
4. 将套接字类的实例传递给服务器。
5. 在服务器受理后, 客户端套接字实例的发送功能和接收功能开始有效。


这里的 `Listen`、`Connect` 和 `Accept` 是为了方便读者理解而添加的, 它们是笔者在介绍系统编程的套接字时使用的函数名称。JavaScript 的 WebSocket 的 API 名称与此不同, 但基本思想是一样的。实际上, 因为使用 HTTP 连接后客户端会请求进行协议升级, 所以内部的步骤有点复杂, 但从外部来看, 顺序是一样的。

代码清单 11-13 是服务器和客户端进行连接和发送数据的代码。通过 `WebSocket` 类的构造函数指定连接目标的 URL, 然后通过 `send()` 方法发送数据。在上面的步骤中, 只有步

步骤 2 是需要客户端在连接时完成的。该构造函数和 `onopen` 事件监听器的后台仅执行该操作。

代码清单 11-13 使用 WebSocket 进行连接和发送数据

```
1 var socket = new WebSocket('ws://xxxx.com');
2 socket.onopen = () => {
3     setInterval(() => {
4         if (socket.bufferedAmount === 0) {
5             socket.send(getUpdateData());
6         }
7     }, 50);
8 };
```

 复制代码

针对连接之后的套接字，客户端需要执行以下操作。

`send([数据])`

将数据发送给服务器。

`onmessage`

使用事件处理器接收服务器发送来的数据。

`close([状态码 [, 原因]])`

关闭套接字。

数据可以是字符串、Blob 和 ArrayBuffer 等。`onmessage` 方法用于接收数据。`onmessage` 事件的用法与 Server-Sent Events 相同。

11.8 本章小结

本章介绍了使用 JavaScript 进行通信的具体方法。

与 Go 语言不同，JavaScript 大多用于开发在浏览器上查看的网站，因此实现 HTTP 通信的 API 也有很大不同。另外，为了实现特殊的通信，有时还需要用到一些复杂的技术。Server-

Sent Events、WebSocket 等通信并非只进行一次，这也是与主要用于服务器端实现的 Go 语言的不同之处。

考虑到 Go 语言可能会在云原生时代迅猛发展，并且其本身也是一门简单易学的编程语言，所以本书使用 Go 语言进行了介绍，但一种编程语言并不能覆盖所有领域，我们需要根据不同的用途使用不同的语言。如果大家抱着多学一门编程语言也没坏处的想法，那么恐怕最后还是选 JavaScript，因为 JavaScript 是浏览器唯一支持的编程语言。

浏览器还拥有使用 Service Worker 进行通信缓存、离线访问服务器等功能，本书并未对这些功能进行介绍。Web 还在发展，相信未来会出现更多高级功能。

AI智能总结

本文深入介绍了使用 JavaScript 在浏览器中实现动态的 HTTP 请求的方法和技术。首先讲解了浏览器的 HTTP 生命周期和使用 JavaScript 进行访问的方式，然后详细介绍了 XMLHttpRequest 的用法，包括发送 GET 和 POST 请求以及处理各种数据格式的方法。接着介绍了 Fetch API 的基础内容和选项，以及与 XMLHttpRequest 相比的优势，包括缓存控制和重定向控制等特性。此外，还介绍了如何发送和接收 JSON 数据，以及在浏览器之外的 JavaScript 环境中使用 Fetch API。另外，还介绍了 Server-Sent Events 和 WebSocket 这两种实现从服务器端开始通信的协议和 JavaScript API。通过本文，读者可以全面了解如何使用 JavaScript 在浏览器中进行动态的 HTTP 请求，以及不同 API 的使用方法和特点。文章内容涵盖了前端开发中常用的通信技术，对于想要深入了解前端通信的开发人员来说，是一份非常有价值的技术指南。

[1]: 以前，即使不创建 标签，通过直接将下载内容的 URL 传递给 或 ，也可以打开新窗口进行下载，但在 Chrome 60 和 Firefox 58 中不能这么做了。

精选留言

由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。