```
      yield value;
    }
  } finally {
    reader.releaseLock();
  }
}

fetch('https://fetch.spec.whatwg.org/')
  .then((response) => response.body)
  .then(async function(body) {
    for await (chunk of streamGenerator(body)) {
      console.log(decoder.decode(chunk, { stream: true }));
    }
  });

// <!doctype html><html lang="en"> ...
// whether a <a data-link-type="dfn" href="#concept-header" ...
// result to <var>rangeValue</var>. ...
// ...
```

因为可以使用 ReadableStream 创建 Response 对象，所以就可以在读取流之后，将其通过管道导入另一个流。然后在这个新流上再使用 Body 的方法，如 text()。这样就可以随着流的到达实时检查和操作流内容。下面的代码展示了这种双流技术：

```
fetch('https://fetch.spec.whatwg.org/')
  .then((response) => response.body)
  .then((body) => {
    const reader = body.getReader();

    // 创建第二个流
    return new ReadableStream({
      async start(controller) {
        try {
          while (true) {
            const { value, done } = await reader.read();

            if (done) {
              break;
            }

            // 将主体流的块推到第二个流
            controller.enqueue(value);
          }
        } finally {
          controller.close();
          reader.releaseLock();
        }
      }
    })
  })
  .then((secondaryStream) => new Response(secondaryStream))
  .then(response => response.text())
  .then(console.log);

// <!doctype html><html lang="en"><head><meta charset="utf-8"> ...
```

24.6 Beacon API

为了把尽量多的页面信息传到服务器，很多分析工具需要在页面生命周期中尽量晚的时候向服务器发送遥测或分析数据。因此，理想的情况下是通过浏览器的 unload 事件发送网络请求。这个事件表示用户要离开当前页面，不会再生成别的有用信息了。

在 unload 事件触发时，分析工具要停止收集信息并把收集到的数据发给服务器。这时候有一个问题，因为 unload 事件对浏览器意味着没有理由再发送任何结果未知的网络请求（因为页面都要被销毁了）。例如，在 unload 事件处理程序中创建的任何异步请求都会被浏览器取消。为此，异步 XMLHttpRequest 或 fetch()不适合这个任务。分析工具可以使用同步 XMLHttpRequest 强制发送请求，但这样做会导致用户体验问题。浏览器会因为要等待 unload 事件处理程序完成而延迟导航到下一个页面。

为解决这个问题，W3C 引入了补充性的 Beacon API。这个 API 给 navigator 对象增加了一个 sendBeacon()方法。这个简单的方法接收一个 URL 和一个数据有效载荷参数，并会发送一个 POST 请求。可选的数据有效载荷参数有 ArrayBufferView、Blob、DOMString、FormData 实例。如果请求成功进入了最终要发送的任务队列，则这个方法返回 true，否则返回 false。

可以像下面这样使用这个方法：

```
// 发送 POST 请求
// URL: 'https://example.com/analytics-reporting-url'
// 请求负载: '{foo: "bar"}'

navigator.sendBeacon('https://example.com/analytics-reporting-url', '{foo: "bar"}');
```

这个方法虽然看起来只不过是 POST 请求的一个语法糖，但它有几个重要的特性。

- sendBeacon()并不是只能在页面生命周期末尾使用，而是任何时候都可以使用。
- 调用 sendBeacon()后，浏览器会把请求添加到一个内部的请求队列。浏览器会主动地发送队列中的请求。
- 浏览器保证在原始页面已经关闭的情况下也会发送请求。
- 状态码、超时和其他网络原因造成的失败完全是不透明的，不能通过编程方式处理。
- 信标（beacon）请求会携带调用 sendBeacon()时所有相关的 cookie。

24.7 Web Socket

Web Socket（套接字）的目标是通过一个长时连接实现与服务器全双工、双向的通信。在 JavaScript 中创建 Web Socket 时，一个 HTTP 请求会发送到服务器以初始化连接。服务器响应后，连接使用 HTTP 的 Upgrade 头部从 HTTP 协议切换到 Web Socket 协议。这意味着 Web Socket 不能通过标准 HTTP 服务器实现，而必须使用支持该协议的专有服务器。

因为 Web Socket 使用了自定义协议，所以 URL 方案（scheme）稍有变化：不能再使用 http://或 https://，而要使用 ws://和 wss://。前者是不安全的连接，后者是安全连接。在指定 Web Socket URL 时，必须包含 URL 方案，因为将来有可能再支持其他方案。

使用自定义协议而非 HTTP 协议的好处是，客户端与服务器之间可以发送非常少的数据，不会对 HTTP 造成任何负担。使用更小的数据包让 Web Socket 非常适合带宽和延迟问题比较明显的移动应用。使用自定义协议的缺点是，定义协议的时间比定义 JavaScript API 要长。Web Socket 得到了所有主流浏览器支持。

24.7.1　API

要创建一个新的 Web Socket，就要实例化一个 WebSocket 对象并传入提供连接的 URL：

```
let socket = new WebSocket("ws://www.example.com/server.php");
```

注意，必须给 WebSocket 构造函数传入一个绝对 URL。同源策略不适用于 Web Socket，因此可以打开到任意站点的连接。至于是否与来自特定源的页面通信，则完全取决于服务器。（在握手阶段就可以确定请求来自哪里。）

浏览器会在初始化 WebSocket 对象之后立即创建连接。与 XHR 类似，WebSocket 也有一个 readyState 属性表示当前状态。不过，这个值与 XHR 中相应的值不一样。

- WebSocket.OPENING（0）：连接正在建立。
- WebSocket.OPEN（1）：连接已经建立。
- WebSocket.CLOSING（2）：连接正在关闭。
- WebSocket.CLOSE（3）：连接已经关闭。

WebSocket 对象没有 readystatechange 事件，而是有与上述不同状态对应的其他事件。readyState 值从 0 开始。

任何时候都可以调用 close()方法关闭 Web Socket 连接：

```
socket.close();
```

调用 close()之后，readyState 立即变为 2（连接正在关闭），并会在关闭后变为 3（连接已经关闭）。

24.7.2　发送和接收数据

打开 Web Socket 之后，可以通过连接发送和接收数据。要向服务器发送数据，使用 send()方法并传入一个字符串、ArrayBuffer 或 Blob，如下所示：

```
let socket = new WebSocket("ws://www.example.com/server.php");

let stringData = "Hello world!";
let arrayBufferData = Uint8Array.from(['f', 'o', 'o']);
let blobData = new Blob(['f', 'o', 'o']);

socket.send(stringData);
socket.send(arrayBufferData.buffer);
socket.send(blobData);
```

服务器向客户端发送消息时，WebSocket 对象上会触发 message 事件。这个 message 事件与其他消息协议类似，可以通过 event.data 属性访问到有效载荷：

```
socket.onmessage = function(event) {
  let data = event.data;
  // 对数据执行某些操作
};
```

与通过 send()方法发送的数据类似，event.data 返回的数据也可能是 ArrayBuffer 或 Blob。这由 WebSocket 对象的 binaryType 属性决定，该属性可能是"blob"或"arraybuffer"。

24.7.3　其他事件

WebSocket 对象在连接生命周期中有可能触发 3 个其他事件。

- open：在连接成功建立时触发。
- error：在发生错误时触发。连接无法存续。
- close：在连接关闭时触发。

WebSocket 对象不支持 DOM Level 2 事件监听器，因此需要使用 DOM Level 0 风格的事件处理程序来监听这些事件：

```
let socket = new WebSocket("ws://www.example.com/server.php");
socket.onopen = function() {
  alert("Connection established.");
};
socket.onerror = function() {
  alert("Connection error.");
};
socket.onclose = function() {
  alert("Connection closed.");
};
```

在这些事件中，只有 close 事件的 event 对象上有额外信息。这个对象上有 3 个额外属性：wasClean、code 和 reason。其中，wasClean 是一个布尔值，表示连接是否干净地关闭；code 是一个来自服务器的数值状态码；reason 是一个字符串，包含服务器发来的消息。可以将这些信息显示给用户或记录到日志：

```
socket.onclose = function(event) {
  console.log(`as clean? ${event.wasClean} Code=${event.code} Reason=${
              event.reason}`);
};
```

24.8 安全

探讨 Ajax 安全的文章已经有了很多，事实上也出版了很多专门讨论这个话题的书。大规模 Ajax 应用程序需要考虑的安全问题非常多，但在通用层面上一般需要考虑以下几个问题。

首先，任何 Ajax 可以访问的 URL，也可以通过浏览器或服务器访问，例如下面这个 URL：

```
/getuserinfo.php?id=23
```

请求这个 URL，可以假定返回 ID 为 23 的用户信息。访问者可以将 23 改为 24 或 56，甚至其他任何值。getuserinfo.php 文件必须知道访问者是否拥有访问相应数据的权限。否则，服务器就会大门敞开，泄露所有用户的信息。

在未授权系统可以访问某个资源时，可以将其视为跨站点请求伪造（CSRF，cross-site request forgery）攻击。未授权系统会按照处理请求的服务器的要求伪装自己。Ajax 应用程序，无论大小，都会受到 CSRF 攻击的影响，包括无害的漏洞验证攻击和恶意的数据盗窃或数据破坏攻击。

关于安全防护 Ajax 相关 URL 的一般理论认为，需要验证请求发送者拥有对资源的访问权限。可以通过如下方式实现。

- 要求通过 SSL 访问能够被 Ajax 访问的资源。
- 要求每个请求都发送一个按约定算法计算好的令牌（token）。

注意，以下手段对防护 CSRF 攻击是无效的。

- 要求 POST 而非 GET 请求（很容易修改请求方法）。

- 使用来源 URL 验证来源（来源 URL 很容易伪造）。
- 基于 cookie 验证（同样很容易伪造）。

24.9　小结

Ajax 是无须刷新当前页面即可从服务器获取数据的一个方法，具有如下特点。

- 让 Ajax 迅速流行的中心对象是 `XMLHttpRequest`（XHR）。
- 这个对象最早由微软发明，并在 IE5 中作为通过 JavaScript 从服务器获取 XML 数据的一种手段。
- 之后，Firefox、Safari、Chrome 和 Opera 都复刻了相同的实现。W3C 随后将 XHR 行为写入 Web 标准。
- 虽然不同浏览器的实现有些差异，但 XHR 对象的基本使用在所有浏览器中相对是规范的，因此可以放心地在 Web 应用程序中使用。

XHR 的一个主要限制是同源策略，即通信只能在相同域名、相同端口和相同协议的前提下完成。访问超出这些限制之外的资源会导致安全错误，除非使用了正式的跨域方案。这个方案叫作跨源资源共享（CORS，Cross-Origin Resource Sharing），XHR 对象原生支持 CORS。图片探测和 JSONP 是另外两种跨域通信技术，但没有 CORS 可靠。

Fetch API 是作为对 XHR 对象的一种端到端的替代方案而提出的。这个 API 提供了优秀的基于期约的结构、更直观的接口，以及对 Stream API 的最好支持。

Web Socket 是与服务器的全双工、双向通信渠道。与其他方案不同，Web Socket 不使用 HTTP，而使用了自定义协议，目的是更快地发送小数据块。这需要专用的服务器，但速度优势明显。

第25章 客户端存储

本章内容
- cookie
- 浏览器存储 API
- IndexedDB

随着 Web 应用程序的出现，直接在客户端存储用户信息的需求也随之出现。这背后的想法是合理的：与特定用户相关的信息应该保存在用户的机器上。无论是登录信息、个人偏好，还是其他数据，Web 应用程序提供者都需要有办法把它们保存在客户端。对该问题的第一个解决方案就是 cookie，cookie 由古老的网景公司发明，由一份名为 *Persistent Client State: HTTP Cookies* 的规范定义。今天，cookie 只是在客户端存储数据的一个选项。

25.1 cookie

HTTP cookie 通常也叫作 cookie，最初用于在客户端存储会话信息。这个规范要求服务器在响应 HTTP 请求时，通过发送 `Set-Cookie` HTTP 头部包含会话信息。例如，下面是包含这个头部的一个 HTTP 响应：

```
HTTP/1.1 200 OK
Content-type: text/html
Set-Cookie: name=value
Other-header: other-header-value
```

这个 HTTP 响应会设置一个名为`"name"`，值为`"value"`的 cookie。名和值在发送时都会经过 URL 编码。浏览器会存储这些会话信息，并在之后的每个请求中都会通过 HTTP 头部 cookie 再将它们发回服务器，比如：

```
GET /index.jsl HTTP/1.1
Cookie: name=value
Other-header: other-header-value
```

这些发送回服务器的额外信息可用于唯一标识发送请求的客户端。

25.1.1 限制

cookie 是与特定域绑定的。设置 cookie 后，它会与请求一起发送到创建它的域。这个限制能保证 cookie 中存储的信息只对被认可的接收者开放，不被其他域访问。

因为 cookie 存储在客户端机器上，所以为保证它不会被恶意利用，浏览器会施加限制。同时，cookie 也不会占用太多磁盘空间。

通常，只要遵守以下大致的限制，就不会在任何浏览器中碰到问题：

- 不超过 300 个 cookie；
- 每个 cookie 不超过 4096 字节；
- 每个域不超过 20 个 cookie；
- 每个域不超过 81 920 字节。

每个域能设置的 cookie 总数也是受限的，但不同浏览器的限制不同。例如：

- 最新版 IE 和 Edge 限制每个域不超过 50 个 cookie；
- 最新版 Firefox 限制每个域不超过 150 个 cookie；
- 最新版 Opera 限制每个域不超过 180 个 cookie；
- Safari 和 Chrome 对每个域的 cookie 数没有硬性限制。

如果 cookie 总数超过了单个域的上限，浏览器就会删除之前设置的 cookie。IE 和 Opera 会按照最近最少使用（LRU，Least Recently Used）原则删除之前的 cookie，以便为新设置的 cookie 腾出空间。Firefox 好像会随机删除之前的 cookie，因此为避免不确定的结果，最好不要超出限制。

浏览器也会限制 cookie 的大小。大多数浏览器对 cookie 的限制是不超过 4096 字节，上下可以有一个字节的误差。为跨浏览器兼容，最好保证 cookie 的大小不超过 4095 字节。这个大小限制适用于一个域的所有 cookie，而不是单个 cookie。

如果创建的 cookie 超过最大限制，则该 cookie 会被静默删除。注意，一个字符通常会占 1 字节。如果使用多字节字符（如 UTF-8 Unicode 字符），则每个字符最多可能占 4 字节。

25.1.2　cookie 的构成

cookie 在浏览器中是由以下参数构成的。

- **名称**：唯一标识 cookie 的名称。cookie 名不区分大小写，因此 `myCookie` 和 `MyCookie` 是同一个名称。不过，实践中最好将 cookie 名当成区分大小写来对待，因为一些服务器软件可能这样对待它们。cookie 名必须经过 URL 编码。
- **值**：存储在 cookie 里的字符串值。这个值必须经过 URL 编码。
- **域**：cookie 有效的域。发送到这个域的所有请求都会包含对应的 cookie。这个值可能包含子域（如 www.wrox.com），也可以不包含（如.wrox.com 表示对 wrox.com 的所有子域都有效）。如果不明确设置，则默认为设置 cookie 的域。
- **路径**：请求 URL 中包含这个路径才会把 cookie 发送到服务器。例如，可以指定 cookie 只能由 http://www.wrox.com/books/访问，因此访问 http://www.wrox.com/下的页面就不会发送 cookie，即使请求的是同一个域。
- **过期时间**：表示何时删除 cookie 的时间戳（即什么时间之后就不发送到服务器了）。默认情况下，浏览器会话结束后会删除所有 cookie。不过，也可以设置删除 cookie 的时间。这个值是 GMT 格式（Wdy, DD-Mon-YYYY HH:MM:SS GMT），用于指定删除 cookie 的具体时间。这样即使关闭浏览器 cookie 也会保留在用户机器上。把过期时间设置为过去的时间会立即删除 cookie。
- **安全标志**：设置之后，只在使用 SSL 安全连接的情况下才会把 cookie 发送到服务器。例如，请求 https://www.wrox.com 会发送 cookie，而请求 http://www.wrox.com 则不会。

这些参数在 `Set-Cookie` 头部中使用分号加空格隔开，比如：

```
HTTP/1.1 200 OK
Content-type: text/html
Set-Cookie: name=value; expires=Mon, 22-Jan-07 07:10:24 GMT; domain=.wrox.com
Other-header: other-header-value
```

这个头部设置一个名为"name"的 cookie，这个 cookie 在 2007 年 1 月 22 日 7:10:24 过期，对 www.wrox.com 及其他 wrox.com 的子域（如 p2p.wrox.com）有效。

安全标志 secure 是 cookie 中唯一的非名/值对，只需一个 secure 就可以了。比如：

```
HTTP/1.1 200 OK
Content-type: text/html
Set-Cookie: name=value; domain=.wrox.com; path=/; secure
Other-header: other-header-value
```

这里创建的 cookie 对所有 wrox.com 的子域及该域中的所有页面有效（通过 path=/指定）。不过，这个 cookie 只能在 SSL 连接上发送，因为设置了 secure 标志。

要知道，域、路径、过期时间和 secure 标志用于告诉浏览器什么情况下应该在请求中包含 cookie。这些参数并不会随请求发送给服务器，实际发送的只有 cookie 的名/值对。

25.1.3 JavaScript 中的 cookie

在 JavaScript 中处理 cookie 比较麻烦，因为接口过于简单，只有 BOM 的 document.cookie 属性。根据用法不同，该属性的表现迥异。要使用该属性获取值时，document.cookie 返回包含页面中所有有效 cookie 的字符串（根据域、路径、过期时间和安全设置），以分号分隔，如下面的例子所示：

```
name1=value1;name2=value2;name3=value3
```

所有名和值都是 URL 编码的，因此必须使用 decodeURIComponent()解码。

在设置值时，可以通过 document.cookie 属性设置新的 cookie 字符串。这个字符串在被解析后会添加到原有 cookie 中。设置 document.cookie 不会覆盖之前存在的任何 cookie，除非设置了已有的 cookie。设置 cookie 的格式如下，与 Set-Cookie 头部的格式一样：

```
name=value; expires=expiration_time; path=domain_path; domain=domain_name; secure
```

在所有这些参数中，只有 cookie 的名称和值是必需的。下面是个简单的例子：

```
document.cookie = "name=Nicholas";
```

这行代码会创建一个名为"name"的会话 cookie，其值为"Nicholas"。这个 cookie 在每次客户端向服务器发送请求时都会被带上，在浏览器关闭时就会被删除。虽然这样直接设置也可以，因为不需要在名称或值中编码任何字符，但最好还是使用 encodeURIComponent()对名称和值进行编码，比如：

```
document.cookie = encodeURIComponent("name") + "=" +
                  encodeURIComponent("Nicholas");
```

要为创建的 cookie 指定额外的信息，只要像 Set-Cookie 头部一样直接在后面追加相同格式的字符串即可：

```
document.cookie = encodeURIComponent("name") + "=" +
                  encodeURIComponent("Nicholas") + "; domain=.wrox.com; path=/";
```

因为在 JavaScript 中读写 cookie 不是很直观，所以可以通过辅助函数来简化相应的操作。与 cookie 相关的基本操作有读、写和删除。这些在 CookieUtil 对象中表示如下：

```
class CookieUtil {
  static get(name) {
    let cookieName = `${encodeURIComponent(name)}=`,
        cookieStart = document.cookie.indexOf(cookieName),
        cookieValue = null;

    if (cookieStart > -1){
      let cookieEnd = document.cookie.indexOf(";", cookieStart);
      if (cookieEnd == -1){
        cookieEnd = document.cookie.length;
      }
      cookieValue = decodeURIComponent(document.cookie.substring(cookieStart
          + cookieName.length, cookieEnd));
      }

      return cookieValue;
    }

    static set(name, value, expires, path, domain, secure) {
      let cookieText =
        `${encodeURIComponent(name)}=${encodeURIComponent(value)}`

      if (expires instanceof Date) {
        cookieText += `; expires=${expires.toGMTString()}`;
      }

      if (path) {
        cookieText += `; path=${path}`;
      }

      if (domain) {
        cookieText += `; domain=${domain}`;
      }

      if (secure) {
        cookieText += "; secure";
      }

      document.cookie = cookieText;
    }

    static unset(name, path, domain, secure) {
      CookieUtil.set(name, "", new Date(0), path, domain, secure);
  }
};
```

CookieUtil.get()方法用于取得给定名称的 cookie 值。为此，需要在 document.cookie 返回的字符串中查找是否存在名称后面加上等号。如果找到了，则使用 indexOf()再查找该位置后面的分号（表示该 cookie 的末尾）。如果没有找到分号，说明这个 cookie 在字符串末尾，因此字符串剩余部分都是 cookie 的值。取得 cookie 值后使用 decodeURIComponent()解码，然后返回。如果没有找到 cookie，则返回 null。

CookieUtil.set()方法用于设置页面上的 cookie，接收多个参数：cookie 名称、cookie 值、可选的 Date 对象（表示何时删除 cookie）、可选的 URL 路径、可选的域以及可选的布尔值（表示是否添加 secure 标志）。这些参数以它们的使用频率为序，只有前两个是必需的。在方法内部，使用了

encodeURIComponent()对名称和值进行编码，然后再依次检查其他参数。如果 expires 参数是 Date 对象，则使用 Date 对象的 toGMTString()方法添加一个 expires 选项来获得正确的日期格式。剩下的代码就是简单地追加 cookie 字符串，最终设置给 document.cookie。

没有直接删除已有 cookie 的方法。为此，需要再次设置同名 cookie（包括相同路径、域和安全选项），但要将其过期时间设置为某个过去的时间。CookieUtil.unset()方法实现了这些处理。这个方法接收 4 个参数：要删除 cookie 的名称、可选的路径、可选的域和可选的安全标志。

这些参数会传给 CookieUtil.set()，将 cookie 值设置为空字符串，将过期时间设置为 1970 年 1 月 1 日（以 0 毫秒初始化的 Date 对象的值）。这样可以保证删除 cookie。

可以像下面这样使用这些方法：

```
// 设置 cookie
CookieUtil.set("name", "Nicholas");
CookieUtil.set("book", "Professional JavaScript");

// 读取 cookie
alert(CookieUtil.get("name")); // "Nicholas"
alert(CookieUtil.get("book")); // "Professional JavaScript"

// 删除 cookie
CookieUtil.unset("name");
CookieUtil.unset("book");

// 设置有路径、域和过期时间的 cookie
CookieUtil.set("name", "Nicholas", "/books/projs/", "www.wrox.com",
               new Date("January 1, 2010"));

// 删除刚刚设置的 cookie
CookieUtil.unset("name", "/books/projs/", "www.wrox.com");

// 设置安全 cookie
CookieUtil.set("name", "Nicholas", null, null, null, true);
```

这些方法通过处理解析和 cookie 字符串构建，简化了使用 cookie 存储数据的操作。

25.1.4 子 cookie

为绕过浏览器对每个域 cookie 数的限制，有些开发者提出了**子 cookie** 的概念。子 cookie 是在单个 cookie 存储的小块数据，本质上是使用 cookie 的值在单个 cookie 中存储多个名/值对。最常用的子 cookie 模式如下：

```
name=name1=value1&name2=value2&name3=value3&name4=value4&name5=value5
```

子 cookie 的格式类似于查询字符串。这些值可以存储为单个 cookie，而不用单独存储为自己的名/值对。结果就是网站或 Web 应用程序能够在单域 cookie 数限制下存储更多的结构化数据。

要操作子 cookie，就需要再添加一些辅助方法。解析和序列化子 cookie 的方式不一样，且因为对子 cookie 的使用而变得更复杂。比如，要取得某个子 cookie，就需要先取得 cookie，然后在解码值之前需要先像下面这样找到子 cookie：

```
class SubCookieUtil {
  static get(name, subName) {
    let subCookies = SubCookieUtil.getAll(name);
    return subCookies ? subCookies[subName] : null;
```