（续）

键	值
	force-cache ❑ 无论命中有效缓存还是无效缓存都通过 fetch()返回。不发送请求 ❑ 未命中缓存会发送请求，并缓存响应。然后 fetch()返回响应 only-if-cached ❑ 只在请求模式为 same-origin 时使用缓存 ❑ 无论命中有效缓存还是无效缓存都通过 fetch()返回。不发送请求 ❑ 未命中缓存返回状态码为 504（网关超时）的响应 默认为 default
credentials	用于指定在外发请求中如何包含 cookie。与 XMLHttpRequest 的 withCredentials 标签类似 必须是下列字符串值之一 ❑ omit：不发送 cookie ❑ same-origin：只在请求 URL 与发送 fetch()请求的页面同源时发送 cookie ❑ include：无论同源还是跨源都包含 cookie 在支持 Credential Management API 的浏览器中，也可以是一个 FederatedCredential 或 PasswordCredential 的实例 默认为 same-origin
headers	用于指定请求头部 必须是 Headers 对象实例或包含字符串格式键/值对的常规对象 默认值为不包含键/值对的 Headers 对象。这不意味着请求不包含任何头部，浏览器仍然会随请求发送一些头部。虽然这些头部对 JavaScript 不可见，但浏览器的网络检查器可以观察到
integrity	用于强制子资源完整性 必须是包含子资源完整性标识符的字符串 默认为空字符串
keepalive	用于指示浏览器允许请求存在时间超出页面生命周期。适合报告事件或分析，比如页面在 fetch()请求后很快卸载。设置 keepalive 标志的 fetch()请求可用于替代 Navigator.sendBeacon() 必须是布尔值 默认为 false
method	用于指定 HTTP 请求方法 基本上就是如下字符串值： ❑ GET ❑ POST ❑ PUT ❑ PATCH ❑ DELETE ❑ HEAD ❑ OPTIONS ❑ CONNECT ❑ TARCE 默认为 GET

（续）

键	值
mode	用于指定请求模式。这个模式决定来自跨源请求的响应是否有效，以及客户端可以读取多少响应。违反这里指定模式的请求会抛出错误 必须是下列字符串值之一 ❑ cors：允许遵守 CORS 协议的跨源请求。响应是“CORS 过滤的响应”，意思是响应中可以访问的浏览器头部是经过浏览器强制白名单过滤的 ❑ no-cors：允许不需要发送预检请求的跨源请求（HEAD、GET 和只带有满足 CORS 请求头部的 POST）。响应类型是 opaque，意思是不能读取响应内容 ❑ same-origin：任何跨源请求都不允许发送 ❑ navigate：用于支持 HTML 导航，只在文档间导航时使用。基本用不到 在通过构造函数手动创建 Request 实例时，默认为 cors；否则，默认为 no-cors
redirect	用于指定如何处理重定向响应（状态码为 301、302、303、307 或 308） 必须是下列字符串值之一 ❑ follow：跟踪重定向请求，以最终非重定向 URL 的响应作为最终响应 ❑ error：重定向请求会抛出错误 ❑ manual：不跟踪重定向请求，而是返回 opaqueredirect 类型的响应，同时仍然暴露期望的重定向 URL。允许以手动方式跟踪重定向 默认为 follow
referrer	用于指定 HTTP 的 Referer 头部的内容 必须是下列字符串值之一 ❑ no-referrer：以 no-referrer 作为值 ❑ client/about:client：以当前 URL 或 no-referrer（取决于来源策略 referrerPolicy）作为值 ❑ <URL>：以伪造 URL 作为值。伪造 URL 的源必须与执行脚本的源匹配 默认为 client/about:client
referrerPolicy	用于指定 HTTP 的 Referer 头部 必须是下列字符串值之一 no-referrer ❑ 请求中不包含 Referer 头部 no-referrer-when-downgrade ❑ 对于从安全 HTTPS 上下文发送到 HTTP URL 的请求，不包含 Referer 头部 ❑ 对于所有其他请求，将 Referer 设置为完整 URL origin ❑ 对于所有请求，将 Referer 设置为只包含源 same-origin ❑ 对于跨源请求，不包含 Referer 头部 ❑ 对于同源请求，将 Referer 设置为完整 URL

（续）

键	值
	strict-origin ❑ 对于从安全 HTTPS 上下文发送到 HTTP URL 的请求，不包含 Referer 头部 ❑ 对于所有其他请求，将 Referer 设置为只包含源 origin-when-cross-origin ❑ 对于跨源请求，将 Referer 设置为只包含源 ❑ 对于同源请求，将 Referer 设置为完整 URL strict-origin-when-cross-origin ❑ 对于从安全 HTTPS 上下文发送到 HTTP URL 的请求，不包含 Referer 头部 ❑ 对于所有其他跨源请求，将 Referer 设置为只包含源 ❑ 对于同源请求，将 Referer 设置为完整 URL unsafe-url ❑ 对于所有请求，将 Referer 设置为完整 URL 默认为 no-referrer-when-downgrade
signal	用于支持通过 AbortController 中断进行中的 fetch()请求 必须是 AbortSignal 的实例 默认为未关联控制器的 AbortSignal 实例

24.5.2　常见 Fetch 请求模式

与 XMLHttpRequest 一样，fetch()既可以发送数据也可以接收数据。使用 init 对象参数，可以配置 fetch()在请求体中发送各种序列化的数据。

1. 发送 JSON 数据

可以像下面这样发送简单 JSON 字符串：

```
let payload = JSON.stringify({
  foo: 'bar'
});

let jsonHeaders = new Headers({
  'Content-Type': 'application/json'
});

fetch('/send-me-json', {
  method: 'POST',    // 发送请求体时必须使用一种 HTTP 方法
  body: payload,
  headers: jsonHeaders
});
```

2. 在请求体中发送参数

因为请求体支持任意字符串值，所以可以通过它发送请求参数：

```
let payload = 'foo=bar&baz=qux';

let paramHeaders = new Headers({
  'Content-Type': 'application/x-www-form-urlencoded; charset=UTF-8'
});
```

```
fetch('/send-me-params', {
  method: 'POST',  // 发送请求体时必须使用一种 HTTP 方法
  body: payload,
  headers: paramHeaders
});
```

3. 发送文件

因为请求体支持 FormData 实现，所以 fetch()也可以序列化并发送文件字段中的文件：

```
let imageFormData = new FormData();
let imageInput = document.querySelector("input[type='file']");

imageFormData.append('image', imageInput.files[0]);

fetch('/img-upload', {
  method: 'POST',
  body: imageFormData
});
```

这个 fetch()实现可以支持多个文件：

```
let imageFormData = new FormData();
let imageInput = document.querySelector("input[type='file'][multiple]");

for (let i = 0; i < imageInput.files.length; ++i) {
  imageFormData.append('image', imageInput.files[i]);
}

fetch('/img-upload', {
  method: 'POST',
  body: imageFormData
});
```

4. 加载 Blob 文件

Fetch API 也能提供 Blob 类型的响应，而 Blob 又可以兼容多种浏览器 API。一种常见的做法是明确将图片文件加载到内存，然后将其添加到 HTML 图片元素。为此，可以使用响应对象上暴露的 blob()方法。这个方法返回一个期约，解决为一个 Blob 的实例。然后，可以将这个实例传给 URL.createObjectUrl()以生成可以添加给图片元素 src 属性的值：

```
const imageElement = document.querySelector('img');

fetch('my-image.png')
  .then((response) => response.blob())
  .then((blob) => {
    imageElement.src = URL.createObjectURL(blob);
  });
```

5. 发送跨源请求

从不同的源请求资源，响应要包含 CORS 头部才能保证浏览器收到响应。没有这些头部，跨源请求会失败并抛出错误。

```
fetch('//cross-origin.com');
// TypeError: Failed to fetch
// No 'Access-Control-Allow-Origin' header is present on the requested resource.
```

如果代码不需要访问响应，也可以发送 no-cors 请求。此时响应的 type 属性值为 opaque，因此

无法读取响应内容。这种方式适合发送探测请求或者将响应缓存起来供以后使用。

```
fetch('//cross-origin.com', { method: 'no-cors' })
  .then((response) => console.log(response.type));

// opaque
```

6. 中断请求

Fetch API 支持通过 AbortController/AbortSignal 对中断请求。调用 AbortController.abort()会中断所有网络传输，特别适合希望停止传输大型负载的情况。中断进行中的 fetch()请求会导致包含错误的拒绝。

```
let abortController = new AbortController();

fetch('wikipedia.zip', { signal: abortController.signal })
  .catch(() => console.log('aborted!');

// 10 毫秒后中断请求
setTimeout(() => abortController.abort(), 10);

// 已经中断
```

24.5.3　Headers 对象

Headers 对象是所有外发请求和入站响应头部的容器。每个外发的 Request 实例都包含一个空的 Headers 实例，可以通过 Request.prototype.headers 访问，每个入站 Response 实例也可以通过 Response.prototype.headers 访问包含着响应头部的 Headers 对象。这两个属性都是可修改属性。另外，使用 new Headers()也可以创建一个新实例。

1. Headers 与 Map 的相似之处

Headers 对象与 Map 对象极为相似。这是合理的，因为 HTTP 头部本质上是序列化后的键/值对，它们的 JavaScript 表示则是中间接口。Headers 与 Map 类型都有 get()、set()、has()和 delete()等实例方法，如下面的代码所示：

```
let h = new Headers();
let m = new Map();

// 设置键
h.set('foo', 'bar');
m.set('foo', 'bar');

// 检查键
console.log(h.has('foo')); // true
console.log(m.has('foo')); // true
console.log(h.has('qux')); // false
console.log(m.has('qux')); // false

// 获取值
console.log(h.get('foo')); // bar
console.log(m.get('foo')); // bar

// 更新值
h.set('foo', 'baz');
m.set('foo', 'baz');
```

```
// 取得更新的值
console.log(h.get('foo')); // baz
console.log(m.get('foo')); // baz

// 删除值
h.delete('foo');
m.delete('foo');

// 确定值已经删除
console.log(h.get('foo')); // undefined
console.log(m.get('foo')); // undefined
```

Headers 和 Map 都可以使用一个可迭代对象来初始化，比如：

```
let seed = [['foo', 'bar']];

let h = new Headers(seed);
let m = new Map(seed);

console.log(h.get('foo')); // bar
console.log(m.get('foo')); // bar
```

而且，它们也都有相同的 keys()、values()和 entries()迭代器接口：

```
let seed = [['foo', 'bar'], ['baz', 'qux']];

let h = new Headers(seed);
let m = new Map(seed);

console.log(...h.keys());    // foo, baz
console.log(...m.keys());    // foo, baz

console.log(...h.values());  // bar, qux
console.log(...m.values());  // bar, qux

console.log(...h.entries()); // ['foo', 'bar'], ['baz', 'qux']
console.log(...m.entries()); // ['foo', 'bar'], ['baz', 'qux']
```

2. Headers 独有的特性

Headers 并不是与 Map 处处都一样。在初始化 Headers 对象时，也可以使用键/值对形式的对象，而 Map 则不可以：

```
let seed = {foo: 'bar'};

let h = new Headers(seed);
console.log(h.get('foo')); // bar

let m = new Map(seed);
// TypeError: object is not iterable
```

一个 HTTP 头部字段可以有多个值，而 Headers 对象通过 append()方法支持添加多个值。在 Headers 实例中还不存在的头部上调用 append()方法相当于调用 set()。后续调用会以逗号为分隔符拼接多个值：

```
let h = new Headers();

h.append('foo', 'bar');
console.log(h.get('foo')); // "bar"
```

```
h.append('foo', 'baz');
console.log(h.get('foo')); // "bar, baz"
```

3. 头部护卫

某些情况下，并非所有 HTTP 头部都可以被客户端修改，而 Headers 对象使用护卫来防止不被允许的修改。不同的护卫设置会改变 set()、append()和 delete()的行为。违反护卫限制会抛出 TypeError。

Headers 实例会因来源不同而展现不同的行为，它们的行为由护卫来控制。JavaScript 可以决定 Headers 实例的护卫设置。下表列出了不同的护卫设置和每种设置对应的行为。

护　卫	适用情形	限　制
none	在通过构造函数创建 Headers 实例时激活	无
request	在通过构造函数初始化 Request 对象，且 mode 值为非 no-cors 时激活	不允许修改禁止修改的头部（参见 MDN 文档中的 forbidden header name 词条）
request-no-cors	在通过构造函数初始化 Request 对象，且 mode 值为 no-cors 时激活	不允许修改非简单头部（参见 MDN 文档中的 simple header 词条）
response	在通过构造函数初始化 Response 对象时激活	不允许修改禁止修改的响应头部（参见 MDN 文档中的 forbidden response header name 词条）
immutable	在通过 error()或 redirect()静态方法初始化 Response 对象时激活	不允许修改任何头部

24.5.4　Request 对象

顾名思义，Request 对象是获取资源请求的接口。这个接口暴露了请求的相关信息，也暴露了使用请求体的不同方式。

> **注意**　与请求体相关的属性和方法将在本章 24.5.6 节介绍。

1. 创建 Request 对象

可以通过构造函数初始化 Request 对象。为此需要传入一个 input 参数，一般是 URL：

```
let r = new Request('https://foo.com');
console.log(r);
// Request {...}
```

Request 构造函数也接收第二个参数——一个 init 对象。这个 init 对象与前面介绍的 fetch()的 init 对象一样。没有在 init 对象中涉及的值则会使用默认值：

```
// 用所有默认值创建 Request 对象
console.log(new Request(''));

// Request {
//   bodyUsed: false
//   cache: "default"
//   credentials: "same-origin"
//   destination: ""
//   headers: Headers {}
```

```
//   integrity: ""
//   keepalive: false
//   method: "GET"
//   mode: "cors"
//   redirect: "follow"
//   referrer: "about:client"
//   referrerPolicy: ""
//   signal: AbortSignal {aborted: false, onabort: null}
//   url: "<current URL>"
// }

// 用指定的初始值创建 Request 对象
console.log(new Request('https://foo.com',
                        { method: 'POST' }));

// Request {
//   bodyUsed: false
//   cache: "default"
//   credentials: "same-origin"
//   destination: ""
//   headers: Headers {}
//   integrity: ""
//   keepalive: false
//   method: "POST"
//   mode: "cors"
//   redirect: "follow"
//   referrer: "about:client"
//   referrerPolicy: ""
//   signal: AbortSignal {aborted: false, onabort: null}
//   url: "https://foo.com/"
// }
```

2. 克隆 Request 对象

Fetch API 提供了两种不太一样的方式用于创建 `Request` 对象的副本：使用 `Request` 构造函数和使用 `clone()` 方法。

将 `Request` 实例作为 `input` 参数传给 `Request` 构造函数，会得到该请求的一个副本：

```
let r1 = new Request('https://foo.com');
let r2 = new Request(r1);

console.log(r2.url); // https://foo.com/
```

如果再传入 `init` 对象，则 `init` 对象的值会覆盖源对象中同名的值：

```
let r1 = new Request('https://foo.com');
let r2 = new Request(r1, {method: 'POST'});

console.log(r1.method); // GET
console.log(r2.method); // POST
```

这种克隆方式并不总能得到一模一样的副本。最明显的是，第一个请求的请求体会被标记为“已使用”：

```
let r1 = new Request('https://foo.com',
                     { method: 'POST', body: 'foobar' });
let r2 = new Request(r1);

console.log(r1.bodyUsed); // true
console.log(r2.bodyUsed); // false
```

如果源对象与创建的新对象不同源，则 referrer 属性会被清除。此外，如果源对象的 mode 为 navigate，则会被转换为 same-origin。

第二种克隆 Request 对象的方式是使用 clone()方法，这个方法会创建一模一样的副本，任何值都不会被覆盖。与第一种方式不同，这种方法不会将任何请求的请求体标记为“已使用”：

```
let r1 = new Request('https://foo.com', { method: 'POST', body: 'foobar' });
let r2 = r1.clone();

console.log(r1.url);      // https://foo.com/
console.log(r2.url);      // https://foo.com/

console.log(r1.bodyUsed); // false
console.log(r2.bodyUsed); // false
```

如果请求对象的 bodyUsed 属性为 true（即请求体已被读取），那么上述任何一种方式都不能用来创建这个对象的副本。在请求体被读取之后再克隆会导致抛出 TypeError。

```
let r = new Request('https://foo.com');
r.clone();
new Request(r);
// 没有错误

r.text();  // 设置 bodyUsed 为 true
r.clone();
// TypeError: Failed to execute 'clone' on 'Request': Request body is already used

new Request(r);
// TypeError: Failed to construct 'Request': Cannot construct a Request with a
Request object that has already been used.
```

3. 在 fetch()中使用 Request 对象

fetch()和 Request 构造函数拥有相同的函数签名并不是巧合。在调用 fetch()时，可以传入已经创建好的 Request 实例而不是 URL。与 Request 构造函数一样，传给 fetch()的 init 对象会覆盖传入请求对象的值：

```
let r = new Request('https://foo.com');

// 向 foo.com 发送 GET 请求
fetch(r);

// 向 foo.com 发送 POST 请求
fetch(r, { method: 'POST' });
```

fetch()会在内部克隆传入的 Request 对象。与克隆 Request 一样，fetch()也不能拿请求体已经用过的 Request 对象来发送请求：

```
let r = new Request('https://foo.com',
                    { method: 'POST', body: 'foobar' });

r.text();

fetch(r);
// TypeError: Cannot construct a Request with a Request object that has already
been used.
```

关键在于，通过 fetch 使用 Request 会将请求体标记为已使用。也就是说，有请求体的 Request 只能在一次 fetch 中使用。（不包含请求体的请求不受此限制。）演示如下：

```
let r = new Request('https://foo.com',
                    { method: 'POST', body: 'foobar' });

fetch(r);

fetch(r);
// TypeError: Cannot construct a Request with a Request object that has already
been used.
```

要想基于包含请求体的相同 Request 对象多次调用 fetch()，必须在第一次发送 fetch()请求前调用 clone()：

```
let r = new Request('https://foo.com',
                    { method: 'POST', body: 'foobar' });

// 3 个都会成功
fetch(r.clone());
fetch(r.clone());
fetch(r);
```

24.5.5 Response 对象

顾名思义，Response 对象是获取资源响应的接口。这个接口暴露了响应的相关信息，也暴露了使用响应体的不同方式。

> **注意** 与响应体相关的属性和方法将在本章 24.5.6 节介绍。

1. 创建 Response 对象

可以通过构造函数初始化 Response 对象且不需要参数。此时响应实例的属性均为默认值，因为它并不代表实际的 HTTP 响应：

```
let r = new Response();
console.log(r);
// Response {
//   body: (...)
//   bodyUsed: false
//   headers: Headers {}
//   ok: true
//   redirected: false
//   status: 200
//   statusText: "OK"
//   type: "default"
//   url: ""
// }
```

Response 构造函数接收一个可选的 body 参数。这个 body 可以是 null，等同于 fetch()参数 init 中的 body。还可以接收一个可选的 init 对象，这个对象可以包含下表所列的键和值。