键	值
headers	必须是 Headers 对象实例或包含字符串键/值对的常规对象实例
	默认为没有键/值对的 Headers 对象
status	表示 HTTP 响应状态码的整数
	默认为 200
statusText	表示 HTTP 响应状态的字符串
	默认为空字符串

可以像下面这样使用 body 和 init 来构建 Response 对象：

```
let r = new Response('foobar', {
  status: 418,
  statusText: 'I\'m a teapot'
});
console.log(r);

// Response {
//   body: (...)
//   bodyUsed: false
//   headers: Headers {}
//   ok: false
//   redirected: false
//   status: 418
//   statusText: "I'm a teapot"
//   type: "default"
//   url: ""
// }
```

大多数情况下，产生 Response 对象的主要方式是调用 fetch()，它返回一个最后会解决为 Response 对象的期约，这个 Response 对象代表实际的 HTTP 响应。下面的代码展示了这样得到的 Response 对象：

```
fetch('https://foo.com')
  .then((response) => {
    console.log(response);
  });

// Response {
//   body: (...)
//   bodyUsed: false
//   headers: Headers {}
//   ok: true
//   redirected: false
//   status: 200
//   statusText: "OK"
//   type: "basic"
//   url: "https://foo.com/"
// }
```

Response 类还有两个用于生成 Response 对象的静态方法：Response.redirect()和 Response.error()。前者接收一个 URL 和一个重定向状态码（301、302、303、307 或 308），返回重定向的 Response 对象：

```
console.log(Response.redirect('https://foo.com', 301));
// Response {
//   body: (...)
//   bodyUsed: false
//   headers: Headers {}
//   ok: false
//   redirected: false
//   status: 301
//   statusText: ""
//   type: "default"
//   url: ""
// }
```

提供的状态码必须对应重定向，否则会抛出错误：

```
Response.redirect('https://foo.com', 200);
// RangeError: Failed to execute 'redirect' on 'Response': Invalid status code
```

另一个静态方法 `Response.error()`用于产生表示网络错误的 `Response` 对象（网络错误会导致 `fetch()`期约被拒绝）。

```
console.log(Response.error());
// Response {
//   body: (...)
//   bodyUsed: false
//   headers: Headers {}
//   ok: false
//   redirected: false
//   status: 0
//   statusText: ""
//   type: "error"
//   url: ""
// }
```

2. 读取响应状态信息

`Response` 对象包含一组只读属性，描述了请求完成后的状态，如下表所示。

属　性	值
`headers`	响应包含的 `Headers` 对象
`ok`	布尔值，表示 HTTP 状态码的含义。200~299 的状态码返回 `true`，其他状态码返回 `false`
`redirected`	布尔值，表示响应是否至少经过一次重定向
`status`	整数，表示响应的 HTTP 状态码
`statusText`	字符串，包含对 HTTP 状态码的正式描述。这个值派生自可选的 HTTP Reason-Phrase 字段，因此如果服务器以 Reason-Phrase 为由拒绝响应，这个字段可能是空字符串
`type`	字符串，包含响应类型。可能是下列字符串值之一 ❑ `basic`：表示标准的同源响应 ❑ `cors`：表示标准的跨源响应 ❑ `error`：表示响应对象是通过 `Response.error()`创建的 ❑ `opaque`：表示 `no-cors` 的 `fetch()`返回的跨源响应 ❑ `opaqueredirect`：表示对 `redirect` 设置为 `manual` 的请求的响应
`url`	包含响应 URL 的字符串。对于重定向响应，这是最终的 URL，非重定向响应就是它产生的

以下代码演示了返回 200、302、404 和 500 状态码的 URL 对应的响应：

```
fetch('//foo.com').then(console.log);
// Response {
//   body: (...)
//   bodyUsed: false
//   headers: Headers {}
//   ok: true
//   redirected: false
//   status: 200
//   statusText: "OK"
//   type: "basic"
//   url: "https://foo.com/"
// }

fetch('//foo.com/redirect-me').then(console.log);
// Response {
//   body: (...)
//   bodyUsed: false
//   headers: Headers {}
//   ok: true
//   redirected: true
//   status: 200
//   statusText: "OK"
//   type: "basic"
//   url: "https://foo.com/redirected-url/"
// }

fetch('//foo.com/does-not-exist').then(console.log);
// Response {
//   body: (...)
//   bodyUsed: false
//   headers: Headers {}
//   ok: false
//   redirected: true
//   status: 404
//   statusText: "Not Found"
//   type: "basic"
//   url: "https://foo.com/does-not-exist/"
// }

fetch('//foo.com/throws-error').then(console.log);
// Response {
//   body: (...)
//   bodyUsed: false
//   headers: Headers {}
//   ok: false
//   redirected: true
//   status: 500
//   statusText: "Internal Server Error"
//   type: "basic"
//   url: "https://foo.com/throws-error/"
// }
```

3. 克隆 Response 对象

克隆 `Response` 对象的主要方式是使用 `clone()`方法，这个方法会创建一个一模一样的副本，不会覆盖任何值。这样不会将任何请求的请求体标记为已使用：

```
let r1 = new Response('foobar');
let r2 = r1.clone();

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

如果响应对象的 bodyUsed 属性为 true（即响应体已被读取），则不能再创建这个对象的副本。在响应体被读取之后再克隆会导致抛出 TypeError。

```
let r = new Response('foobar');
r.clone();
// 没有错误

r.text();  // 设置 bodyUsed 为 true

r.clone();
// TypeError: Failed to execute 'clone' on 'Response': Response body is
already used
```

有响应体的 Response 对象只能读取一次。（不包含响应体的 Response 对象不受此限制。）比如：

```
let r = new Response('foobar');

r.text().then(console.log); // foobar

r.text().then(console.log);
// TypeError: Failed to execute 'text' on 'Response': body stream is locked
```

要多次读取包含响应体的同一个 Response 对象，必须在第一次读取前调用 clone()：

```
let r = new Response('foobar');

r.clone().text().then(console.log); // foobar
r.clone().text().then(console.log); // foobar
r.text().then(console.log);         // foobar
```

此外，通过创建带有原始响应体的 Response 实例，可以执行伪克隆操作。关键是这样不会把第一个 Response 实例标记为已读，而是会在两个响应之间**共享**：

```
let r1 = new Response('foobar');
let r2 = new Response(r1.body);

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

r2.text().then(console.log); // foobar
r1.text().then(console.log);
// TypeError: Failed to execute 'text' on 'Response': body stream is locked
```

24.5.6 Request、Response 及 Body 混入

Request 和 Response 都使用了 Fetch API 的 Body 混入，以实现两者承担有效载荷的能力。这个混入为两个类型提供了只读的 body 属性（实现为 ReadableStream）、只读的 bodyUsed 布尔值（表示 body 流是否已读）和一组方法，用于从流中读取内容并将结果转换为某种 JavaScript 对象类型。

通常，将 Request 和 Response 主体作为流来使用主要有两个原因。一个原因是有效载荷的大小可能会导致网络延迟，另一个原因是流 API 本身在处理有效载荷方面是有优势的。除此之外，最好是一

次性获取资源主体。

Body 混入提供了 5 个方法，用于将 ReadableStream 转存到缓冲区的内存里，将缓冲区转换为某种 JavaScript 对象类型，以及通过期约来产生结果。在解决之前，期约会等待主体流报告完成及缓冲被解析。这意味着客户端必须等待响应的资源完全加载才能访问其内容。

1. Body.text()

Body.text()方法返回期约，解决为将缓冲区转存得到的 UTF-8 格式字符串。下面的代码展示了在 Response 对象上使用 Body.text()：

```
fetch('https://foo.com')
  .then((response) => response.text())
  .then(console.log);

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

以下代码展示了在 Request 对象上使用 Body.text()：

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

request.text()
  .then(console.log);

// barbazqux
```

2. Body.json()

Body.json()方法返回期约，解决为将缓冲区转存得到的 JSON。下面的代码展示了在 Response 对象上使用 Body.json()：

```
fetch('https://foo.com/foo.json')
  .then((response) => response.json())
  .then(console.log);

// {"foo": "bar"}
```

以下代码展示了在 Request 对象上使用 Body.json()：

```
let request = new Request('https://foo.com',
                          { method:'POST', body: JSON.stringify({ bar: 'baz' }) });

request.json()
  .then(console.log);

// {bar: 'baz'}
```

3. Body.formData()

浏览器可以将 FormData 对象序列化/反序列化为主体。例如，下面这个 FormData 实例：

```
let myFormData = new FormData();
myFormData.append('foo', 'bar');
```

在通过 HTTP 传送时，WebKit 浏览器会将其序列化为下列内容：

```
------WebKitFormBoundarydR9Q2kOzE6nbN7eR
Content-Disposition: form-data; name="foo"
```

```
bar
------WebKitFormBoundarydR9Q2kOzE6nbN7eR--
```

Body.formData()方法返回期约，解决为将缓冲区转存得到的 FormData 实例。下面的代码展示了在 Response 对象上使用 Body.formData()：

```
fetch('https://foo.com/form-data')
  .then((response) => response.formData())
  .then((formData) => console.log(formData.get('foo'));

// bar
```

以下代码展示了在 Request 对象上使用 Body.formData()：

```
let myFormData = new FormData();
myFormData.append('foo', 'bar');

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

request.formData()
  .then((formData) => console.log(formData.get('foo'));

// bar
```

4. **Body.arrayBuffer()**

有时候，可能需要以原始二进制格式查看和修改主体。为此，可以使用 Body.arrayBuffer()将主体内容转换为 ArrayBuffer 实例。Body.arrayBuffer()方法返回期约，解决为将缓冲区转存得到的 ArrayBuffer 实例。下面的代码展示了在 Response 对象上使用 Body.arrayBuffer()：

```
fetch('https://foo.com')
  .then((response) => response.arrayBuffer())
  .then(console.log);

// ArrayBuffer(...) {}
```

以下代码展示了在 Request 对象上使用 Body.arrayBuffer()：

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

// 以整数形式打印二进制编码的字符串
request.arrayBuffer()
  .then((buf) => console.log(new Int8Array(buf)));

// Int8Array(7) [97, 98, 99, 100, 101, 102, 103]
```

5. **Body.blob()**

有时候，可能需要以原始二进制格式使用主体，不用查看和修改。为此，可以使用 Body.blob()将主体内容转换为 Blob 实例。Body.blob()方法返回期约，解决为将缓冲区转存得到的 Blob 实例。下面的代码展示了在 Response 对象上使用 Body.blob()：

```
fetch('https://foo.com')
  .then((response) => response.blob())
  .then(console.log);

// Blob(...) {size:..., type: "..."}
```

以下代码展示了在 Request 对象上使用 Body.blob()：

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

request.blob()
  .then(console.log);

// Blob(7) {size: 7, type: "text/plain;charset=utf-8"}
```

6. 一次性流

因为 Body 混入是构建在 ReadableStream 之上的，所以主体流只能使用一次。这意味着所有主体混入方法都只能调用一次，再次调用就会抛出错误。

```
fetch('https://foo.com')
  .then((response) => response.blob().then(() => response.blob()));

// TypeError: Failed to execute 'blob' on 'Response': body stream is locked
let request = new Request('https://foo.com',
                          { method: 'POST', body: 'foobar' });

request.blob().then(() => request.blob());
// TypeError: Failed to execute 'blob' on 'Request': body stream is locked
```

即使是在读取流的过程中，所有这些方法也会在它们被调用时给 ReadableStream 加锁，以阻止其他读取器访问：

```
fetch('https://foo.com')
  .then((response) => {
    response.blob(); // 第一次调用给流加锁
    response.blob(); // 第二次调用再次加锁会失败
  });

// TypeError: Failed to execute 'blob' on 'Response': body stream is locked
let request = new Request('https://foo.com',
                          { method: 'POST', body: 'foobar' });

request.blob(); // 第一次调用给流加锁
request.blob(); // 第二次调用再次加锁会失败
// TypeError: Failed to execute 'blob' on 'Request': body stream is locked
```

作为 Body 混入的一部分，bodyUsed 布尔值属性表示 ReadableStream 是否已**摄受**（disturbed），意思是读取器是否已经在流上加了锁。这不一定表示流已经被完全读取。下面的代码演示了这个属性：

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

console.log(request.bodyUsed);     // false
console.log(response.bodyUsed);    // false

request.text().then(console.log);  // foobar
response.text().then(console.log); // foobar

console.log(request.bodyUsed);     // true
console.log(response.bodyUsed);    // true
```

7. 使用 ReadableStream 主体

JavaScript 编程逻辑很多时候会将访问网络作为原子操作，比如请求是同时创建和发送的，响应数据也是以统一的格式一次性暴露出来的。这种约定隐藏了底层的混乱，让涉及网络的代码变得很清晰。

从 TCP/IP 角度来看，传输的数据是以分块形式抵达端点的，而且速度受到网速的限制。接收端点会为此分配内存，并将收到的块写入内存。Fetch API 通过 ReadableStream 支持在这些块到达时就实时读取和操作这些数据。

> **注意** 本节会以获取 Fetch API 规范的 HTML 为例。这个页面差不多有 1MB 大小，足以让示例中接收的数据分成多个块。

正如 Stream API 所定义的，ReadableStream 暴露了 getReader()方法，用于产生 ReadableStream-DefaultReader，这个读取器可以用于在数据到达时异步获取数据块。数据流的格式是 Uint8Array。

下面的代码调用了读取器的 read()方法，把最早可用的块打印了出来：

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

    console.log(reader); // ReadableStreamDefaultReader {}

    reader.read()
      .then(console.log);
  });

// { value: Uint8Array{}, done: false }
```

在随着数据流的到来取得整个有效载荷，可以像下面这样递归调用 read()方法：

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

    function processNextChunk({value, done}) {
      if (done) {
        return;
      }

      console.log(value);

      return reader.read()
          .then(processNextChunk);
    }

    return reader.read()
        .then(processNextChunk);
  });

// { value: Uint8Array{}, done: false }
// { value: Uint8Array{}, done: false }
// { value: Uint8Array{}, done: false }
// ...
```

异步函数非常适合这样的 fetch()操作。可以通过使用 async/await 将上面的递归调用打平：

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

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

        if (done) {
          break;
        }

        console.log(value);
      }
    });

// { value: Uint8Array{}, done: false }
// { value: Uint8Array{}, done: false }
// { value: Uint8Array{}, done: false }
// ...
```

另外，read()方法也可以真接封装到 Iterable 接口中。因此就可以在 for-await-of 循环中方便地实现这种转换：

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

    let asyncIterable = {
      [Symbol.asyncIterator]() {
        return {
          next() {
            return reader.read();
          }
        };
      }
    };

    for await (chunk of asyncIterable) {
      console.log(chunk);
    }
  });

// { value: Uint8Array{}, done: false }
// { value: Uint8Array{}, done: false }
// { value: Uint8Array{}, done: false }
// ...
```

通过将异步逻辑包装到一个生成器函数中，还可以进一步简化代码。而且，这个实现通过支持只读取部分流也变得更稳健。如果流因为耗尽或错误而终止，读取器会释放锁，以允许不同的流读取器继续操作：

```
async function* streamGenerator(stream) {
  const reader = stream.getReader();
```

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

      if (done) {
        break;
      }

      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(chunk);
    }
  });
```

在这些例子中，当读取完 Uint8Array 块之后，浏览器会将其标记为可以被垃圾回收。对于需要在不连续的内存中连续检查大量数据的情况，这样可以节省很多内存空间。

缓冲区的大小，以及浏览器是否等待缓冲区被填充后才将其推到流中，要根据 JavaScript 运行时的实现。浏览器会控制等待分配的缓冲区被填满，同时会尽快将缓冲区数据（有时候可能未填充数据）发送到流。

不同浏览器中分块大小可能不同，这取决于带宽和网络延迟。此外，浏览器如果决定不等待网络，也可以将部分填充的缓冲区发送到流。最终，我们的代码要准备好处理以下情况：

- 不同大小的 Uint8Array 块；
- 部分填充的 Uint8Array 块；
- 块到达的时间间隔不确定。

默认情况下，块是以 Uint8Array 格式抵达的。因为块的分割不会考虑编码，所以会出现某些值作为多字节字符被分散到两个连续块中的情况。手动处理这些情况是很麻烦的，但很多时候可以使用 Encoding API 的可插拔方案。

要将 Uint8Array 转换为可读文本，可以将缓冲区传给 TextDecoder，返回转换后的值。通过设置 stream: true，可以将之前的缓冲区保留在内存，从而让跨越两个块的内容能够被正确解码：

```
let decoder = new TextDecoder();

async function* streamGenerator(stream) {
  const reader = stream.getReader();

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

      if (done) {
        break;
      }
```