```
    "edition": 1,
    "year": 2006
  }
]
```

前面这个数组包含了很多表示书的对象。每个对象都包含一些键，其中一个是"authors"，对应的值也是一个数组。对象和数组通常会作为 JSON 数组的顶级结构（尽管不是必需的），以便创建大型复杂数据结构。

23.2　解析与序列化

JSON 的迅速流行并不仅仅因为其语法与 JavaScript 类似，很大程度上还因为 JSON 可以直接被解析成可用的 JavaScript 对象。与解析为 DOM 文档的 XML 相比，这个优势非常明显。为此，JavaScript 开发者可以非常方便地使用 JSON 数据。比如，前面例子中的 JSON 包含很多图书，通过如下代码就可以获取第三本书的书名：

```
books[2].title
```

当然，以上代码假设把前面的数据结构保存在了变量 books 中。相比之下，遍历 DOM 结构就显得麻烦多了：

```
doc.getElementsByTagName("book")[2].getAttribute("title");
```

看看这些方法调用，就不难想象为什么 JSON 大受 JavaScript 开发者欢迎了。JSON 出现之后就迅速成为了 Web 服务的事实序列化标准。

23.2.1　JSON 对象

早期的 JSON 解析器基本上就相当于 JavaScript 的 eval()函数。因为 JSON 是 JavaScript 语法的子集，所以 eval()可以解析、解释，并将其作为 JavaScript 对象和数组返回。ECMAScript 5 增加了 JSON 全局对象，正式引入解析 JSON 的能力。这个对象在所有主流浏览器中都得到了支持。旧版本的浏览器可以使用垫片脚本（参见 GitHub 上 douglascrockford/JSON-js 中的 JSON in JavaScript）。考虑到直接执行代码的风险，最好不要在旧版本浏览器中只使用 eval()求值 JSON。这个 JSON 垫片脚本最好只在浏览器原生不支持 JSON 解析时使用。

JSON 对象有两个方法：stringify()和 parse()。在简单的情况下，这两个方法分别可以将 JavaScript 序列化为 JSON 字符串，以及将 JSON 解析为原生 JavaScript 值。例如：

```
let book = {
  title: "Professional JavaScript",
  authors: [
    "Nicholas C. Zakas",
    "Matt Frisbie"
  ],
  edition: 4,
  year: 2017
};
let jsonText = JSON.stringify(book);
```

这个例子使用 JSON.stringify()把一个 JavaScript 对象序列化为一个 JSON 字符串，保存在变量 jsonText 中。默认情况下，JSON.stringify()会输出不包含空格或缩进的 JSON 字符串，因此

jsonText 的值是这样的：

```
{"title":"Professional JavaScript","authors":["Nicholas C. Zakas","Matt Frisbie"],
"edition":4,"year":2017}
```

在序列化 JavaScript 对象时，所有函数和原型成员都会有意地在结果中省略。此外，值为 undefined 的任何属性也会被跳过。最终得到的就是所有实例属性均为有效 JSON 数据类型的表示。

JSON 字符串可以直接传给 JSON.parse()，然后得到相应的 JavaScript 值。比如，可以使用以下代码创建与 book 对象类似的新对象：

```
let bookCopy = JSON.parse(jsonText);
```

注意，book 和 bookCopy 是两个完全不同的对象，没有任何关系。但是它们拥有相同的属性和值。

如果给 JSON.parse()传入的 JSON 字符串无效，则会导致抛出错误。

23.2.2 序列化选项

实际上，JSON.stringify()方法除了要序列化的对象，还可以接收两个参数。这两个参数可以用于指定其他序列化 JavaScript 对象的方式。第一个参数是过滤器，可以是数组或函数；第二个参数是用于缩进结果 JSON 字符串的选项。单独或组合使用这些参数可以更好地控制 JSON 序列化。

1. 过滤结果

如果第二个参数是一个数组，那么 JSON.stringify()返回的结果只会包含该数组中列出的对象属性。比如下面的例子：

```
let book = {
  title: "Professional JavaScript",
  authors: [
    "Nicholas C. Zakas",
    "Matt Frisbie"
  ],
  edition: 4,
  year: 2017
};
let jsonText = JSON.stringify(book, ["title", "edition"]);
```

在这个例子中，JSON.stringify()方法的第二个参数是一个包含两个字符串的数组："title"和"edition"。它们对应着要序列化的对象中的属性，因此结果 JSON 字符串中只会包含这两个属性：

```
{"title":"Professional JavaScript","edition":4}
```

如果第二个参数是一个函数，则行为又有不同。提供的函数接收两个参数：属性名（key）和属性值（value）。可以根据这个 key 决定要对相应属性执行什么操作。这个 key 始终是字符串，只是在值不属于某个键/值对时会是空字符串。

为了改变对象的序列化，返回的值就是相应 key 应该包含的结果。注意，返回 undefined 会导致属性被忽略。下面看一个例子：

```
let book = {
  title: "Professional JavaScript",
  authors: [
    "Nicholas C. Zakas",
    "Matt Frisbie"
  ],
  edition: 4,
```

```
  year: 2017
};
let jsonText = JSON.stringify(book, (key, value) => {
  switch(key) {
    case "authors":
      return value.join(",")
    case "year":
      return 5000;
    case "edition":
      return undefined;
    default:
      return value;
  }
});
```

这个函数基于键进行了过滤。如果键是"authors"，则将数组值转换为字符串；如果键是"year"，则将值设置为 5000；如果键是"edition"，则返回 undefined 忽略该属性。最后一定要提供默认返回值，以便返回其他属性传入的值。第一次调用这个函数实际上会传入空字符串 key，值是 book 对象。最终得到的 JSON 字符串是这样的：

```
{"title":"Professional JavaScript","authors":"Nicholas C. Zakas,Matt
Frisbie","year":5000}
```

注意，函数过滤器会应用到要序列化的对象所包含的所有对象，因此如果数组中包含多个具有这些属性的对象，则序列化之后每个对象都只会剩下上面这些属性。

Firefox 3.5~3.6 在 JSON.stringify()的第二个参数是函数时有一个 bug：此时函数只能作为过滤器，返回 undefined 会导致跳过属性，返回其他值则会包含属性。Firefox 4 修复了这个 bug。

2. 字符串缩进

JSON.stringify()方法的第三个参数控制缩进和空格。在这个参数是数值时，表示每一级缩进的空格数。例如，每级缩进 4 个空格，可以这样：

```
let book = {
  title: "Professional JavaScript",
  authors: [
    "Nicholas C. Zakas",
    "Matt Frisbie"
  ],
  edition: 4,
  year: 2017
};
let jsonText = JSON.stringify(book, null, 4);
```

这样得到的 jsonText 格式如下：

```
{
    "title": "Professional JavaScript",
    "authors": [
        "Nicholas C. Zakas",
        "Matt Frisbie"
    ],
    "edition": 4,
    "year": 2017
}
```

注意，除了缩进，JSON.stringify()方法还为方便阅读插入了换行符。这个行为对于所有有效的

缩进参数都会发生。（只缩进不换行也没什么用。）最大缩进值为 10，大于 10 的值会自动设置为 10。

如果缩进参数是一个字符串而非数值，那么 JSON 字符串中就会使用这个字符串而不是空格来缩进。使用字符串，也可以将缩进字符设置为 Tab 或任意字符，如两个连字符：

```
let jsonText = JSON.stringify(book, null, "--" );
```

这样，jsonText 的值会变成如下格式：

```
{
--"title": "Professional JavaScript",
--"authors": [
----"Nicholas C. Zakas",
----"Matt Frisbie"
--],
--"edition": 4,
--"year": 2017
}
```

使用字符串时同样有 10 个字符的长度限制。如果字符串长度超过 10，则会在第 10 个字符处截断。

3. toJSON()方法

有时候，对象需要在 JSON.stringify()之上自定义 JSON 序列化。此时，可以在要序列化的对象中添加 toJSON()方法，序列化时会基于这个方法返回适当的 JSON 表示。事实上，原生 Date 对象就有一个 toJSON()方法，能够自动将 JavaScript 的 Date 对象转换为 ISO 8601 日期字符串（本质上与在 Date 对象上调用 toISOString()方法一样）。

下面的对象为自定义序列化而添加了一个 toJSON()方法：

```
let book = {
  title: "Professional JavaScript",
  authors: [
    "Nicholas C. Zakas",
    "Matt Frisbie"
  ],
  edition: 4,
  year: 2017,
  toJSON: function() {
    return this.title;
  }
};
let jsonText = JSON.stringify(book);
```

这里 book 对象中定义的 toJSON()方法简单地返回了图书的书名（this.title）。与 Date 对象类似，这个对象会被序列化为简单字符串而非对象。toJSON()方法可以返回任意序列化值，都可以起到相应的作用。如果对象被嵌入在另一个对象中，返回 undefined 会导致值变成 null；或者如果是顶级对象，则本身就是 undefined。注意，箭头函数不能用来定义 toJSON()方法。主要原因是箭头函数的词法作用域是全局作用域，在这种情况下不合适。

toJSON()方法可以与过滤函数一起使用，因此理解不同序列化流程的顺序非常重要。在把对象传给 JSON.stringify()时会执行如下步骤。

(1) 如果可以获取实际的值，则调用 toJSON()方法获取实际的值，否则使用默认的序列化。

(2) 如果提供了第二个参数，则应用过滤。传入过滤函数的值就是第(1)步返回的值。

(3) 第(2)步返回的每个值都会相应地进行序列化。

(4) 如果提供了第三个参数，则相应地进行缩进。

理解这个顺序有助于决定是创建 toJSON()方法，还是使用过滤函数，抑或是两者都用。

23.2.3　解析选项

JSON.parse()方法也可以接收一个额外的参数，这个函数会针对每个键/值对都调用一次。为区别于传给 JSON.stringify()的起过滤作用的**替代函数**（replacer），这个函数被称为**还原函数**（reviver）。实际上它们的格式完全一样，即还原函数也接收两个参数，属性名（key）和属性值（value），另外也需要返回值。

如果还原函数返回 undefined，则结果中就会删除相应的键。如果返回了其他任何值，则该值就会成为相应键的值插入到结果中。还原函数经常被用于把日期字符串转换为 Date 对象。例如：

```
let book = {
  title: "Professional JavaScript",
  authors: [
    "Nicholas C. Zakas",
    "Matt Frisbie"
  ],
  edition: 4,
  year: 2017,
  releaseDate: new Date(2017, 11, 1)
};
let jsonText = JSON.stringify(book);
let bookCopy = JSON.parse(jsonText,
    (key, value) => key == "releaseDate" ? new Date(value) : value);
alert(bookCopy.releaseDate.getFullYear());
```

以上代码在 book 对象中增加了 releaseDate 属性，是一个 Date 对象。这个对象在被序列化为 JSON 字符串后，又被重新解析为一个对象 bookCopy。这里的还原函数会查找"releaseDate"键，如果找到就会根据它的日期字符串创建新的 Date 对象。得到的 bookCopy.releaseDate 属性又变回了 Date 对象，因此可以调用其 getFullYear()方法。

23.3　小结

JSON 是一种轻量级数据格式，可以方便地表示复杂数据结构。这个格式使用 JavaScript 语法的一个子集表示对象、数组、字符串、数值、布尔值和 null。虽然 XML 也能胜任同样的角色，但 JSON 更简洁，JavaScript 支持也更好。更重要的是，所有浏览器都已经原生支持全局 JSON 对象。

ECMAScript 5 定义了原生 JSON 对象，用于将 JavaScript 对象序列化为 JSON 字符串，以及将 JSON 数组解析为 JavaScript 对象。JSON.stringify()和 JSON.parse()方法分别用于实现这两种操作。这两个方法都有一些选项可以用来改变默认的行为，以实现过滤或修改流程。

第24章

网络请求与远程资源

本章内容

- 使用 XMLHttpRequest 对象
- 处理 XMLHttpRequest 事件
- 源域 Ajax 限制
- Fetch API
- Streams API

2005 年，Jesse James Garrett 撰写了一篇文章，“Ajax—A New Approach to Web Applications”。这篇文章中描绘了一个被他称作 Ajax（Asynchronous JavaScript+XML，即异步 JavaScript 加 XML）的技术。这个技术涉及发送服务器请求额外数据而不刷新页面，从而实现更好的用户体验。Garrett 解释了这个技术怎样改变自 Web 诞生以来就一直延续的传统单击等待的模式。

把 Ajax 推到历史舞台上的关键技术是 XMLHttpRequest（XHR）对象。这个对象最早由微软发明，然后被其他浏览器所借鉴。在 XHR 出现之前，Ajax 风格的通信必须通过一些黑科技实现，主要是使用隐藏的窗格或内嵌窗格。XHR 为发送服务器请求和获取响应提供了合理的接口。这个接口可以实现异步从服务器获取额外数据，意味着用户点击不用页面刷新也可以获取数据。通过 XHR 对象获取数据后，可以使用 DOM 方法把数据插入网页。虽然 Ajax 这个名称中包含 XML，但实际上 Ajax 通信与数据格式无关。这个技术主要是可以实现在不刷新页面的情况下从服务器获取数据，格式并不一定是 XML。

实际上，Garrett 所称的这种 Ajax 技术已经出现很长时间了。在 Garrett 那篇文章之前，一般称这种技术为**远程脚本**。这种浏览器与服务器的通信早在 1998 年就通过不同方式实现了。最初，JavaScript 对服务器的请求可以通过中介（如 Java 小程序或 Flash 影片）来发送。后来 XHR 对象又为开发者提供了原生的浏览器通信能力，减少了实现这个目的的工作量。

XHR 对象的 API 被普遍认为比较难用，而 Fetch API 自从诞生以后就迅速成为了 XHR 更现代的替代标准。Fetch API 支持期约（promise）和服务线程（service worker），已经成为极其强大的 Web 开发工具。

> **注意** 本章会全面介绍 XMLHttpRequest，但它实际上是过时 Web 规范的产物，应该只在旧版本浏览器中使用。实际开发中，应该尽可能使用 fetch()。

24.1 XMLHttpRequest 对象

IE5 是第一个引入 XHR 对象的浏览器。这个对象是通过 ActiveX 对象实现并包含在 MSXML 库中的。为此，XHR 对象的 3 个版本在浏览器中分别被暴露为 MSXML2.XMLHttp、MSXML2.XMLHttp.3.0 和 MXSML2.XMLHttp.6.0。

所有现代浏览器都通过 XMLHttpRequest 构造函数原生支持 XHR 对象：

```
let xhr = new XMLHttpRequest();
```

24.1.1　使用 XHR

使用 XHR 对象首先要调用 open()方法，这个方法接收 3 个参数：请求类型（"get"、"post"等）、请求 URL，以及表示请求是否异步的布尔值。下面是一个例子：

```
xhr.open("get", "example.php", false);
```

这行代码就可以向 example.php 发送一个同步的 GET 请求。关于这行代码需要说明几点。首先，这里的 URL 是相对于代码所在页面的，当然也可以使用绝对 URL。其次，调用 open()不会实际发送请求，只是为发送请求做好准备。

> **注意**　只能访问同源 URL，也就是域名相同、端口相同、协议相同。如果请求的 URL 与发送请求的页面在任何方面有所不同，则会抛出安全错误。

要发送定义好的请求，必须像下面这样调用 send()方法：

```
xhr.open("get", "example.txt", false);
xhr.send(null);
```

send()方法接收一个参数，是作为请求体发送的数据。如果不需要发送请求体，则必须传 null，因为这个参数在某些浏览器中是必需的。调用 send()之后，请求就会发送到服务器。

因为这个请求是同步的，所以 JavaScript 代码会等待服务器响应之后再继续执行。收到响应后，XHR 对象的以下属性会被填充上数据。

- responseText：作为响应体返回的文本。
- responseXML：如果响应的内容类型是"text/xml"或"application/xml"，那就是包含响应数据的 XML DOM 文档。
- status：响应的 HTTP 状态。
- statusText：响应的 HTTP 状态描述。

收到响应后，第一步要检查 status 属性以确保响应成功返回。一般来说，HTTP 状态码为 2*xx* 表示成功。此时，responseText 或 responseXML（如果内容类型正确）属性中会有内容。如果 HTTP 状态码是 304，则表示资源未修改过，是从浏览器缓存中直接拿取的。当然这也意味着响应有效。为确保收到正确的响应，应该检查这些状态，如下所示：

```
xhr.open("get", "example.txt", false);
xhr.send(null);

if ((xhr.status >= 200 && xhr.status < 300) || xhr.status == 304) {
  alert(xhr.responseText);
} else {
  alert("Request was unsuccessful: " + xhr.status);
}
```

以上代码可能显示服务器返回的内容，也可能显示错误消息，取决于 HTTP 响应的状态码。为确定下一步该执行什么操作，最好检查 status 而不是 statusText 属性，因为后者已经被证明在跨浏览器的情况下不可靠。无论是什么响应内容类型，responseText 属性始终会保存响应体，而 responseXML

则对于非 XML 数据是 null。

虽然可以像前面的例子一样发送同步请求，但多数情况下最好使用异步请求，这样可以不阻塞 JavaScript 代码继续执行。XHR 对象有一个 readyState 属性，表示当前处在请求/响应过程的哪个阶段。这个属性有如下可能的值。

- 0：未初始化（Uninitialized）。尚未调用 open()方法。
- 1：已打开（Open）。已调用 open()方法，尚未调用 send()方法。
- 2：已发送（Sent）。已调用 send()方法，尚未收到响应。
- 3：接收中（Receiving）。已经收到部分响应。
- 4：完成（Complete）。已经收到所有响应，可以使用了。

每次 readyState 从一个值变成另一个值，都会触发 readystatechange 事件。可以借此机会检查 readyState 的值。一般来说，我们唯一关心的 readyState 值是 4，表示数据已就绪。为保证跨浏览器兼容，onreadystatechange 事件处理程序应该在调用 open()之前赋值。来看下面的例子：

```
let xhr = new XMLHttpRequest();
xhr.onreadystatechange = function() {
  if (xhr.readyState == 4) {
    if ((xhr.status >= 200 && xhr.status < 300) || xhr.status == 304) {
      alert(xhr.responseText);
    } else {
      alert("Request was unsuccessful: " + xhr.status);
    }
  }
};
xhr.open("get", "example.txt", true);
xhr.send(null);
```

以上代码使用 DOM Level 0 风格为 XHR 对象添加了事件处理程序，因为并不是所有浏览器都支持 DOM Level 2 风格。与其他事件处理程序不同，onreadystatechange 事件处理程序不会收到 event 对象。在事件处理程序中，必须使用 XHR 对象本身来确定接下来该做什么。

> **注意** 由于 onreadystatechange 事件处理程序的作用域问题，这个例子在 onreadystatechange 事件处理程序中使用了 xhr 对象而不是 this 对象。使用 this 可能导致功能失败或导致错误，取决于用户使用的是什么浏览器。因此还是使用保存 XHR 对象的变量更保险一些。

在收到响应之前如果想取消异步请求，可以调用 abort()方法：

```
xhr.abort();
```

调用这个方法后，XHR 对象会停止触发事件，并阻止访问这个对象上任何与响应相关的属性。中断请求后，应该取消对 XHR 对象的引用。由于内存问题，不推荐重用 XHR 对象。

24.1.2 HTTP 头部

每个 HTTP 请求和响应都会携带一些头部字段，这些字段可能对开发者有用。XHR 对象会通过一些方法暴露与请求和响应相关的头部字段。

默认情况下，XHR 请求会发送以下头部字段。

- Accept：浏览器可以处理的内容类型。
- Accept-Charset：浏览器可以显示的字符集。
- Accept-Encoding：浏览器可以处理的压缩编码类型。
- Accept-Language：浏览器使用的语言。
- Connection：浏览器与服务器的连接类型。
- Cookie：页面中设置的 Cookie。
- Host：发送请求的页面所在的域。
- Referer：发送请求的页面的 URI。注意，这个字段在 HTTP 规范中就拼错了，所以考虑到兼容性也必须将错就错。（正确的拼写应该是 Referrer。）
- User-Agent：浏览器的用户代理字符串。

虽然不同浏览器发送的确切头部字段可能各不相同，但这些通常都是会发送的。如果需要发送额外的请求头部，可以使用 setRequestHeader()方法。这个方法接收两个参数：头部字段的名称和值。为保证请求头部被发送，必须在 open()之后、send()之前调用 setRequestHeader()，如下面的例子所示：

```
let xhr = new XMLHttpRequest();
xhr.onreadystatechange = function() {
  if (xhr.readyState == 4) {
    if ((xhr.status >= 200 && xhr.status < 300) || xhr.status == 304) {
      alert(xhr.responseText);
    } else {
      alert("Request was unsuccessful: " + xhr.status);
    }
  }
};
xhr.open("get", "example.php", true);
xhr.setRequestHeader("MyHeader", "MyValue");
xhr.send(null);
```

服务器通过读取自定义头部可以确定适当的操作。自定义头部一定要区别于浏览器正常发送的头部，否则可能影响服务器正常响应。有些浏览器允许重写默认头部，有些浏览器则不允许。

可以使用 getResponseHeader()方法从 XHR 对象获取响应头部，只要传入要获取头部的名称即可。如果想取得所有响应头部，可以使用 getAllResponseHeaders()方法，这个方法会返回包含所有响应头部的字符串。下面是调用这两个方法的例子：

```
let myHeader = xhr.getResponseHeader("MyHeader");
let allHeaders  xhr.getAllResponseHeaders();
```

服务器可以使用头部向浏览器传递额外的结构化数据。getAllResponseHeaders()方法通常返回类似如下的字符串：

```
Date: Sun, 14 Nov 2004 18:04:03 GMT
Server: Apache/1.3.29 (Unix)
Vary: Accept
X-Powered-By: PHP/4.3.8
Connection: close
Content-Type: text/html; charset=iso-8859-1
```

通过解析以上头部字段的输出，就可以知道服务器发送的所有头部，而不需要单独去检查了。

24.1.3 GET 请求

最常用的请求方法是 GET 请求，用于向服务器查询某些信息。必要时，需要在 GET 请求的 URL 后面添加查询字符串参数。对 XHR 而言，查询字符串必须正确编码后添加到 URL 后面，然后再传给 `open()`方法。

发送 GET 请求最常见的一个错误是查询字符串格式不对。查询字符串中的每个名和值都必须使用 `encodeURIComponent()`编码，所有名/值对必须以和号（`&`）分隔，如下面的例子所示：

```
xhr.open("get", "example.php?name1=value1&name2=value2", true);
```

可以使用以下函数将查询字符串参数添加到现有的 URL 末尾：

```
function addURLParam(url, name, value) {
  url += (url.indexOf("?") == -1 ? "?" : "&");
  url += encodeURIComponent(name) + "=" + encodeURIComponent(value);
  return url;
}
```

这里定义了一个 `addURLParam()`函数，它接收 3 个参数：要添加查询字符串的 URL、查询参数和参数值。首先，这个函数会检查 URL 中是否已经包含问号（以确定是否已经存在其他参数）。如果没有，则加上一个问号；否则就加上一个和号。然后，分别对参数名和参数值进行编码，并添加到 URL 末尾。最后一步是返回更新后的 URL。

可以使用这个函数构建请求 URL，如下面的例子所示：

```
let url = "example.php";

// 添加参数
url = addURLParam(url, "name", "Nicholas");
url = addURLParam(url, "book", "Professional JavaScript");

// 初始化请求
xhr.open("get", url, false);
```

这里使用 `addURLParam()`函数可以保证通过 XHR 发送请求的 URL 格式正确。

24.1.4 POST 请求

第二个最常用的请求是 POST 请求，用于向服务器发送应该保存的数据。每个 POST 请求都应该在请求体中携带提交的数据，而 GET 请求则不然。POST 请求的请求体可以包含非常多的数据，而且数据可以是任意格式。要初始化 POST 请求，`open()`方法的第一个参数要传`"post"`，比如：

```
xhr.open("post", "example.php", true);
```

接下来就是要给 `send()`方法传入要发送的数据。因为 XHR 最初主要设计用于发送 XML，所以可以传入序列化之后的 XML DOM 文档作为请求体。当然，也可以传入任意字符串。

默认情况下，对服务器而言，POST 请求与提交表单是不一样的。服务器逻辑需要读取原始 POST 数据才能取得浏览器发送的数据。不过，可以使用 XHR 模拟表单提交。为此，第一步需要把 `Content-Type` 头部设置为`"application/x-www-formurlencoded"`，这是提交表单时使用的内容类型。第二步是创建对应格式的字符串。POST 数据此时使用与查询字符串相同的格式。如果网页中确实有一个表单需要序列化并通过 XHR 发送到服务器，则可以使用第 14 章的 `serialize()`函数来创建相应的字符串，如下所示：