```
const transaction = db.transaction("users"),
    store = transaction.objectStore("users"),
    request = store.openCursor();
request.onsuccess = (event) => {
  // 处理成功
};
request.onerror = (event) => {
  // 处理错误
};
```

在调用 onsuccess 事件处理程序时，可以通过 event.target.result 访问对象存储中的下一条记录，这个属性中保存着 IDBCursor 的实例（有下一条记录时）或 null（没有记录时）。这个 IDBCursor 实例有几个属性。

- direction：字符串常量，表示游标的前进方向以及是否应该遍历所有重复的值。可能的值包括：NEXT("next")、NEXTUNIQUE("nextunique")、PREV("prev")、PREVUNIQUE("prevunique")。
- key：对象的键。
- value：实际的对象。
- primaryKey：游标使用的键。可能是对象键或索引键（稍后讨论）。

可以像下面这样取得一个结果：

```
request.onsuccess = (event) => {
  const cursor = event.target.result;
  if (cursor) { // 永远要检查
    console.log(`Key: ${cursor.key}, Value: ${JSON.stringify(cursor.value)}`);
  }
};
```

注意，这个例子中的 cursor.value 保存着实际的对象。正因为如此，在显示它之前才需要使用 JSON 来编码。

游标可用于更新个别记录。update()方法使用指定的对象更新当前游标对应的值。与其他类似操作一样，调用 update()会创建一个新请求，因此如果想知道结果，需要为 onsuccess 和 onerror 赋值：

```
request.onsuccess = (event) => {
  const cursor = event.target.result;
  let value,
      updateRequest;

  if (cursor) { // 永远要检查
    if (cursor.key == "foo") {
      value = cursor.value;                    // 取得当前对象
      value.password = "magic!";               // 更新密码
      updateRequest = cursor.update(value); // 请求保存更新后的对象
      updateRequest.onsuccess = () => {
        // 处理成功
      };
      updateRequest.onerror = () => {
        // 处理错误
      };
    }
  }
};
```

也可以调用 delelte()来删除游标位置的记录，与 update()一样，这也会创建一个请求：

```
request.onsuccess = (event) => {
  const cursor = event.target.result;
  let value,
      deleteRequest;
  if (cursor) { // 永远要检查
    if (cursor.key == "foo") {
      deleteRequest = cursor.delete();  // 请求删除对象
      deleteRequest.onsuccess = () => {
        // 处理成功
      };
      deleteRequest.onerror = () => {
        // 处理错误
      };
    }
  }
};
```

如果事务没有修改对象存储的权限，update()和delete()都会抛出错误。

默认情况下，每个游标只会创建一个请求。要创建另一个请求，必须调用下列中的一个方法。

- continue(*key*)：移动到结果集中的下一条记录。参数 *key* 是可选的。如果没有指定 key，游标就移动到下一条记录；如果指定了，则游标移动到指定的键。
- advance(*count*)：游标向前移动指定的 *count* 条记录。

这两个方法都会让游标重用相同的请求，因此也会重用 onsuccess 和 onerror 处理程序，直至不再需要。例如，下面的代码迭代了一个对象存储中的所有记录：

```
request.onsuccess = (event) => {
  const cursor = event.target.result;
  if (cursor) {  // 永远要检查
    console.log(`Key: ${cursor.key}, Value: ${JSON.stringify(cursor.value)}`);
    cursor.continue();  // 移动到下一条记录
  } else {
    console.log("Done!");
  }
};
```

调用 cursor.continue()会触发另一个请求并再次调用 onsuccess 事件处理程序。在没有更多记录时，onsuccess 事件处理程序最后一次被调用，此时 event.target.result 等于 null。

25.3.6 键范围

使用游标会给人一种不太理想的感觉，因为获取数据的方式受到了限制。使用**键范围**（key range）可以让游标更容易管理。键范围对应 IDBKeyRange 的实例。有四种方式指定键范围，第一种是使用only()方法并传入想要获取的键：

```
const onlyRange = IDBKeyRange.only("007");
```

这个范围保证只获取键为"007"的值。使用这个范围创建的游标类似于直接访问对象存储并调用get("007")。

第二种键范围可以定义结果集的下限。下限表示游标开始的位置。例如，下面的键范围保证游标从"007"这个键开始，直到最后：

```
// 从"007"记录开始，直到最后
const lowerRange = IDBKeyRange.lowerBound("007");
```

如果想从"007"后面的记录开始，可以再传入第二个参数 true：

```
// 从"007"的下一条记录开始，直到最后
const lowerRange = IDBKeyRange.lowerBound("007", true);
```

第三种键范围可以定义结果集的上限，通过调用 upperBound()方法可以指定游标不会越过的记录。下面的键范围保证游标从头开始并在到达键为"ace"的记录停止：

```
// 从头开始，到"ace"记录为止
const upperRange = IDBKeyRange.upperBound("ace");
```

如果不想包含指定的键，可以在第二个参数传入 true：

```
// 从头开始，到"ace"的前一条记录为止
const upperRange = IDBKeyRange.upperBound("ace", true);
```

要同时指定下限和上限，可以使用 bound()方法。这个方法接收四个参数：下限的键、上限的键、可选的布尔值表示是否跳过下限和可选的布尔值表示是否跳过上限。下面是几个例子：

```
// 从"007"记录开始，到"ace"记录停止
const boundRange = IDBKeyRange.bound("007", "ace");
// 从"007"的下一条记录开始，到"ace"记录停止
const boundRange = IDBKeyRange.bound("007", "ace", true);
// 从"007"的下一条记录开始，到"ace"的前一条记录停止
const boundRange = IDBKeyRange.bound("007", "ace", true, true);
// 从"007"记录开始，到"ace"的前一条记录停止
const boundRange = IDBKeyRange.bound("007", "ace", false, true);
```

定义了范围之后，把它传给 openCursor()方法，就可以得到位于该范围内的游标：

```
const store = db.transaction("users").objectStore("users"),
    range = IDBKeyRange.bound("007", "ace");
    request = store.openCursor(range);
request.onsuccess = function(event){
  const cursor = event.target.result;
  if (cursor) { // 永远要检查
    console.log(`Key: ${cursor.key}, Value: ${JSON.stringify(cursor.value)}`);
    cursor.continue(); // 移动到下一条记录
  } else {
    console.log("Done!");
  }
};
```

这个例子只会输出从键为"007"的记录开始到键为"ace"的记录结束的对象，比上一节的例子要少。

25.3.7　设置游标方向

openCursor()方法实际上可以接收两个参数，第一个是 IDBKeyRange 的实例，第二个是表示方向的字符串。通常，游标都是从对象存储的第一条记录开始，每次调用 continue()或 advance()都会向最后一条记录前进。这样的游标其默认方向为"next"。如果对象存储中有重复的记录，可能需要游标跳过那些重复的项。为此，可以给 openCursor()的第二个参数传入"nextunique"：

```
const transaction = db.transaction("users"),
    store = transaction.objectStore("users"),
    request = store.openCursor(null, "nextunique");
```

注意，openCursor()的第一个参数是 null，表示默认的键范围是所有值。此游标会遍历对象存储中的记录，从第一条记录开始迭代，到最后一条记录，但会跳过重复的记录。

另外，也可以创建在对象存储中反向移动的游标，从最后一项开始向第一项移动。此时需要给openCursor()传入"prev"或"prevunique"作为第二个参数（后者的意思当然是避免重复）。例如：

```
const transaction = db.transaction("users"),
    store = transaction.objectStore("users"),
    request = store.openCursor(null, "prevunique");
```

在使用"prev"或"prevunique"打开游标时，每次调用 continue()或 advance()都会在对象存储中反向移动游标。

25.3.8 索引

对某些数据集，可能需要为对象存储指定多个键。例如，如果同时记录了用户 ID 和用户名，那可能需要通过任何一种方式来获取用户数据。为此，可以考虑将用户 ID 作为主键，然后在用户名上创建索引。

要创建新索引，首先要取得对象存储的引用，然后像下面的例子一样调用 createIndex()：

```
const transaction = db.transaction("users"),
    store = transaction.objectStore("users"),
    index = store.createIndex("username", "username", { unique: true });
```

createIndex()的第一个参数是索引的名称，第二个参数是索引属性的名称，第三个参数是包含键 unique 的 options 对象。这个选项中的 unique 应该必须指定，表示这个键是否在所有记录中唯一。因为 username 可能不会重复，所以这个键是唯一的。

createIndex()返回的是 IDBIndex 实例。在对象存储上调用 index()方法也可以得到同一个实例。例如，要使用一个已存在的名为"username"的索引，可以像下面这样：

```
const transaction = db.transaction("users"),
    store = transaction.objectStore("users"),
    index = store.index("username");
```

索引非常像对象存储。可以在索引上使用 openCursor()方法创建新游标，这个游标与在对象存储上调用 openCursor()创建的游标完全一样。只是其 result.key 属性中保存的是索引键，而不是主键。下面看一个例子：

```
const transaction = db.transaction("users"),
    store = transaction.objectStore("users"),
    index = store.index("username"),
    request = index.openCursor();
request.onsuccess = (event) => {
  // 处理成功
};
```

使用 openKeyCursor()方法也可以在索引上创建特殊游标，只返回每条记录的主键。这个方法接收的参数与 openCursor()一样。最大的不同在于，event.result.key 是索引键，且 event.result.value 是主键而不是整个记录。

```
const transaction = db.transaction("users"),
    store = transaction.objectStore("users"),
    index = store.index("username"),
    request = index.openKeyCursor();
request.onsuccess = (event) => {
  // 处理成功
  // event.result.key 是索引键，event.result.value 是主键
};
```

可以使用 get()方法并传入索引键通过索引取得单条记录，这会创建一个新请求：

```
const transaction = db.transaction("users"),
    store = transaction.objectStore("users"),
    index = store.index("username"),
    request = index.get("007");
request.onsuccess = (event) => {
  // 处理成功
};
request.onerror = (event) => {
  // 处理错误
};
```

如果想只取得给定索引键的主键，可以使用 getKey()方法。这样也会创建一个新请求，但 result.value 等于主键而不是整个记录：

```
const transaction = db.transaction("users"),
    store = transaction.objectStore("users"),
    index = store.index("username"),
    request = index.getKey("007");
request.onsuccess = (event) => {
  // 处理成功
  // event.target.result.key 是索引键，event.target.result.value 是主键
};
```

在这个 onsuccess 事件处理程序中，event.target.result.value 中应该是用户 ID。

任何时候，都可以使用 IDBIndex 对象的下列属性取得索引的相关信息。

- name：索引的名称。
- keyPath：调用 createIndex()时传入的属性路径。
- objectStore：索引对应的对象存储。
- unique：表示索引键是否唯一的布尔值。

对象存储自身也有一个 indexNames 属性，保存着与之相关索引的名称。使用如下代码可以方便地了解对象存储上已存在哪些索引：

```
const transaction = db.transaction("users"),
    store = transaction.objectStore("users"),
    indexNames = store.indexNames
for (let indexName in indexNames) {
    const index = store.index(indexName);
    console.log(`Index name: ${index.name}
                 KeyPath: ${index.keyPath}
                 Unique: ${index.unique}`);
}
```

以上代码迭代了每个索引并在控制台中输出了它们的信息。

在对象存储上调用 deleteIndex()方法并传入索引的名称可以删除索引：

```
const transaction = db.transaction("users"),
    store = transaction.objectStore("users"),
    store.deleteIndex("username");
```

因为删除索引不会影响对象存储中的数据，所以这个操作没有回调。

25.3.9　并发问题

IndexedDB 虽然是网页中的异步 API，但仍存在并发问题。如果两个不同的浏览器标签页同时打开

了同一个网页，则有可能出现一个网页尝试升级数据库而另一个尚未就绪的情形。有问题的操作是设置数据库为新版本，而版本变化只能在浏览器只有一个标签页使用数据库时才能完成。

第一次打开数据库时，添加 onversionchange 事件处理程序非常重要。另一个同源标签页将数据库打开到新版本时，将执行此回调。对这个事件最好的回应是立即关闭数据库，以便完成版本升级。例如：

```
let request, database;

request = indexedDB.open("admin", 1);
request.onsuccess = (event) => {
  database = event.target.result;
  database.onversionchange = () => database.close();
};
```

应该在每次成功打开数据库后都指定 onversionchange 事件处理程序。记住，onversionchange 有可能会被其他标签页触发。

通过始终都指定这些事件处理程序，可以保证 Web 应用程序能够更好地处理与 IndexedDB 相关的并发问题。

25.3.10 限制

IndexedDB 的很多限制实际上与 Web Storage 一样。首先，IndexedDB 数据库是与页面源（协议、域和端口）绑定的，因此信息不能跨域共享。这意味着 www.wrox.com 和 p2p.wrox.com 会对应不同的数据存储。

其次，每个源都有可以存储的空间限制。当前 Firefox 的限制是每个源 50MB，而 Chrome 是 5MB。移动版 Firefox 有 5MB 限制，如果用度超出配额则会请求用户许可。

Firefox 还有一个限制——本地文本不能访问 IndexedDB 数据库。Chrome 没有这个限制。因此在本地运行本书示例时，要使用 Chrome。

25.4 小结

Web Storage 定义了两个对象用于存储数据：sessionStorage 和 localStorage。前者用于严格保存浏览器一次会话期间的数据，因为数据会在浏览器关闭时被删除。后者用于会话之外持久保存数据。

IndexedDB 是类似于 SQL 数据库的结构化数据存储机制。不同的是，IndexedDB 存储的是对象，而不是数据表。对象存储是通过定义键然后添加数据来创建的。游标用于查询对象存储中的特定数据，而索引可以针对特定属性实现更快的查询。

有了这些存储手段，就可以在客户端通过使用 JavaScript 存储可观的数据。因为这些数据没有加密，所以要注意不能使用它们存储敏感信息。

第26章 模　块

本章内容

- ❑ 理解模块模式
- ❑ 凑合的模块系统
- ❑ 使用前 ES6 模块加载器
- ❑ 使用 ES6 模块

现代 JavaScript 开发毋庸置疑会遇到代码量大和广泛使用第三方库的问题。解决这个问题的方案通常需要把代码拆分成很多部分，然后再通过某种方式将它们连接起来。

在 ECMAScript 6 模块规范出现之前，虽然浏览器原生不支持模块的行为，但迫切需要这样的行为。ECMAScript 同样不支持模块，因此希望使用模块模式的库或代码库必须基于 JavaScript 的语法和词法特性“伪造”出类似模块的行为。

因为 JavaScript 是异步加载的解释型语言，所以得到广泛应用的各种模块实现也表现出不同的形态。这些不同的形态决定了不同的结果，但最终它们都实现了经典的模块模式。

26.1　理解模块模式

将代码拆分成独立的块，然后再把这些块连接起来可以通过模块模式来实现。这种模式背后的思想很简单：把逻辑分块，各自封装，相互独立，每个块自行决定对外暴露什么，同时自行决定引入执行哪些外部代码。不同的实现和特性让这些基本的概念变得有点复杂，但这个基本的思想是所有 JavaScript 模块系统的基础。

26.1.1　模块标识符

模块标识符是所有模块系统通用的概念。模块系统本质上是键/值实体，其中每个模块都有个可用于引用它的标识符。这个标识符在模拟模块的系统中可能是字符串，在原生实现的模块系统中可能是模块文件的实际路径。

有的模块系统支持明确声明模块的标识，还有的模块系统会隐式地使用文件名作为模块标识符。不管怎样，完善的模块系统一定不会存在模块标识冲突的问题，且系统中的任何模块都应该能够无歧义地引用其他模块。

将模块标识符解析为实际模块的过程要根据模块系统对标识符的实现。原生浏览器模块标识符必须提供实际 JavaScript 文件的路径。除了文件路径，Node.js 还会搜索 node_modules 目录，用标识符去匹配包含 index.js 的目录。

26.1.2 模块依赖

模块系统的核心是管理依赖。指定依赖的模块与周围的环境会达成一种契约。本地模块向模块系统声明一组外部模块（依赖），这些外部模块对于当前模块正常运行是必需的。模块系统检视这些依赖，进而保证这些外部模块能够被加载并在本地模块运行时初始化所有依赖。

每个模块都会与某个唯一的标识符关联，该标识符可用于检索模块。这个标识符通常是 JavaScript 文件的路径，但在某些模块系统中，这个标识符也可以是在模块本身内部声明的命名空间路径字符串。

26.1.3 模块加载

加载模块的概念派生自依赖契约。当一个外部模块被指定为依赖时，本地模块期望在执行它时，依赖已准备好并已初始化。

在浏览器中，加载模块涉及几个步骤。加载模块涉及执行其中的代码，但必须是在所有依赖都加载并执行之后。如果浏览器没有收到依赖模块的代码，则必须发送请求并等待网络返回。收到模块代码之后，浏览器必须确定刚收到的模块是否也有依赖。然后递归地评估并加载所有依赖，直到所有依赖模块都加载完成。只有整个依赖图都加载完成，才可以执行入口模块。

26.1.4 入口

相互依赖的模块必须指定一个模块作为入口（entry point），这也是代码执行的起点。这是理所当然的，因为 JavaScript 是顺序执行的，并且是单线程的，所以代码必须有执行的起点。入口模块也可能依赖其他模块，其他模块同样可能有自己的依赖。于是模块化 JavaScript 应用程序的所有模块会构成依赖图。

可以通过有向图来表示应用程序中各模块的依赖关系。图 26-1 展示了一个想象中应用程序的模块依赖关系图。

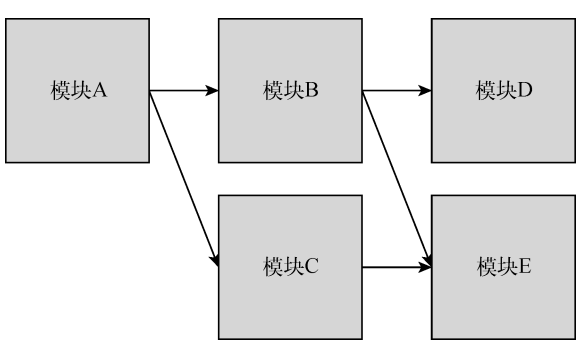

图 26-1

图中的箭头表示依赖方向：模块 A 依赖模块 B 和模块 C，模块 B 依赖模块 D 和模块 E，模块 C 依赖模块 E。因为模块必须在依赖加载完成后才能被加载，所以这个应用程序的入口模块 A 必须在应用程序的其他部分加载后才能执行。

在 JavaScript 中，“加载”的概念可以有多种实现方式。因为模块是作为包含将立即执行的 JavaScript 代码的文件实现的，所以一种可能是按照依赖图的要求依次请求各个脚本。对于前面的应用程序来说，

下面的脚本请求顺序能够满足依赖图的要求：

```
<script src="moduleE.js"></script>
<script src="moduleD.js"></script>
<script src="moduleC.js"></script>
<script src="moduleB.js"></script>
<script src="moduleA.js"></script>
```

模块加载是“阻塞的”，这意味着前置操作必须完成才能执行后续操作。每个模块在自己的代码到达浏览器之后完成加载，此时其依赖已经加载并初始化。不过，这个策略存在一些性能和复杂性问题。为一个应用程序而按顺序加载五个 JavaScript 文件并不理想，并且手动管理正确的加载顺序也颇为棘手。

26.1.5　异步依赖

因为 JavaScript 可以异步执行，所以如果能按需加载就好了。换句话说，可以让 JavaScript 通知模块系统在必要时加载新模块，并在模块加载完成后提供回调。在代码层面，可以通过下面的伪代码来实现：

```
// 在模块 A 里面
load('moduleB').then(function(moduleB) {
  moduleB.doStuff();
});
```

模块 A 的代码使用了 `moduleB` 标识符向模块系统请求加载模块 B，并以模块 B 作为参数调用回调。模块 B 可能已加载完成，也可能必须重新请求和初始化，但这里的代码并不关心。这些事情都交给了模块加载器去负责。

如果重写前面的应用程序，只使用动态模块加载，那么使用一个`<script>`标签即可完成模块 A 的加载。模块 A 会按需请求模块文件，而不会生成必需的依赖列表。这样有几个好处，其中之一就是性能，因为在页面加载时只需同步加载一个文件。

这些脚本也可以分离出来，比如给`<script>`标签应用 `defer` 或 `async` 属性，再加上能够识别异步脚本何时加载和初始化的逻辑。此行为将模拟在 ES6 模块规范中实现的行为，本章稍后会对此进行讨论。

26.1.6　动态依赖

有些模块系统要求开发者在模块开始列出所有依赖，而有些模块系统则允许开发者在程序结构中动态添加依赖。动态添加的依赖有别于模块开头列出的常规依赖，这些依赖必须在模块执行前加载完毕。

下面是动态依赖加载的例子：

```
if (loadCondition) {
  require('./moduleA');
}
```

在这个模块中，是否加载 `moduleA` 是运行时确定的。加载 `moduleA` 时可能是阻塞的，也可能导致执行，且只有模块加载后才会继续。无论怎样，模块内部的代码在 `moduleA` 加载前都不能执行，因为 `moduleA` 的存在是后续模块行为正确的关键。

动态依赖可以支持更复杂的依赖关系，但代价是增加了对模块进行静态分析的难度。

26.1.7　静态分析

模块中包含的发送到浏览器的 JavaScript 代码经常会被静态分析，分析工具会检查代码结构并在不

实际执行代码的情况下推断其行为。对静态分析友好的模块系统可以让模块打包系统更容易将代码处理为较少的文件。它还将支持在智能编辑器里智能自动完成。

更复杂的模块行为，例如动态依赖，会导致静态分析更困难。不同的模块系统和模块加载器具有不同层次的复杂度。至于模块的依赖，额外的复杂度会导致相关工具更难预测模块在执行时到底需要哪些依赖。

26.1.8 循环依赖

要构建一个没有循环依赖的 JavaScript 应用程序几乎是不可能的，因此包括 CommonJS、AMD 和 ES6 在内的所有模块系统都支持循环依赖。在包含循环依赖的应用程序中，模块加载顺序可能会出人意料。不过，只要恰当地封装模块，使它们没有副作用，加载顺序就应该不会影响应用程序的运行。

在下面的模块代码中（其中使用了模块中立的伪代码），任何模块都可以作为入口模块，即使依赖图中存在循环依赖：

```
require('./moduleD');
require('./moduleB');

console.log('moduleA');
require('./moduleA');
require('./moduleC');

console.log('moduleB');
require('./moduleB');
require('./moduleD');

console.log('moduleC');

require('./moduleA');
require('./moduleC');

console.log('moduleD');
```

修改主模块中用到的模块会改变依赖加载顺序。如果 `moduleA` 最先加载，则会打印如下输出，这表示模块加载完成时的绝对顺序：

```
moduleB
moduleC
moduleD
moduleA
```

以上模块加载顺序可以用图 26-2 的依赖图来表示，其中加载器会执行深度优先的依赖加载：

如果 `moduleC` 最先加载，则会打印如下输出，这表示模块加载的绝对顺序：

```
moduleD
moduleA
moduleB
moduleC
```

以上模块加载顺序可以通过图 26-3 的依赖图来表示，其中加载器会执行深度优先的依赖加载：