```
    }

    static getAll(name) {
      let cookieName = encodeURIComponent(name) + "=",
          cookieStart = document.cookie.indexOf(cookieName),
          cookieValue = null,
          cookieEnd,
          subCookies,
          parts,
          result = {};

      if (cookieStart > -1) {
        cookieEnd = document.cookie.indexOf(";", cookieStart);
        if (cookieEnd == -1) {
          cookieEnd = document.cookie.length;
        }
        cookieValue = document.cookie.substring(cookieStart +
                                                cookieName.length, cookieEnd);
        if (cookieValue.length > 0) {
          subCookies = cookieValue.split("&");

          for (let i = 0, len = subCookies.length; i < len; i++) {
            parts = subCookies[i].split("=");
            result[decodeURIComponent(parts[0])] =
              decodeURIComponent(parts[1]);
          }

          return result;
        }
      }
      return null;
    }

    // 省略其他代码
};
```

取得子 cookie 有两个方法：get()和 getAll()。get()用于取得一个子 cookie 的值，getAll()用于取得所有子 cookie，并以对象形式返回，对象的属性是子 cookie 的名称，值是子 cookie 的值。get()方法接收两个参数：cookie 的名称和子 cookie 的名称。这个方法先调用 getAll()取得所有子 cookie，然后返回要取得的子 cookie（如果不存在则返回 null）。

SubCookieUtil.getAll()方法在解析 cookie 值方面与 CookieUtil.get()方法非常相似。不同的是 SubCookieUtil.getAll()方法不会立即解码 cookie 的值，而是先用和号（&）拆分，将所有子 cookie 保存到数组。然后，再基于等号（=）拆分每个子 cookie，使 parts 数组的第一个元素是子 cookie 的名称，第二个元素是子 cookie 的值。两个元素都使用 decodeURIComponent()解码，并添加到 result 对象，最后返回 result 对象。如果 cookie 不存在则返回 null。

可以像下面这样使用这些方法：

```
// 假设 document.cookie=data=name=Nicholas&book=Professional%20JavaScript

// 取得所有子 cookie
let data = SubCookieUtil.getAll("data");
alert(data.name); // "Nicholas"
alert(data.book); // "Professional JavaScript"
```

```
// 取得个别子 cookie
alert(SubCookieUtil.get("data", "name")); // "Nicholas"
alert(SubCookieUtil.get("data", "book")); // "Professional JavaScript"
```

要写入子 cookie，可以使用另外两个方法：`set()`和 `setAll()`。这两个方法的实现如下：

```
class SubCookieUtil {
  // 省略之前的代码

  static set(name, subName, value, expires, path, domain, secure) {
    let subcookies = SubCookieUtil.getAll(name) || {};
    subcookies[subName] = value;
    SubCookieUtil.setAll(name, subcookies, expires, path, domain, secure);
  }

  static setAll(name, subcookies, expires, path, domain, secure) {
    let cookieText = encodeURIComponent(name) + "=",
        subcookieParts = new Array(),
        subName;
    for (subName in subcookies){
      if (subName.length > 0 && subcookies.hasOwnProperty(subName)){
        subcookieParts.push(
          '${encodeURIComponent(subName)}=${encodeURIComponent(subcookies[subName])}');
      }
    }

    if (cookieParts.length > 0) {
      cookieText += subcookieParts.join("&");

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

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

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

      if (secure) {
        cookieText += "; secure";
      }
    } else {
      cookieText += `; expires=${(new Date(0)).toGMTString()}`;
    }
    document.cookie = cookieText;
  }

  // 省略其他代码
};
```

`set()`方法接收 7 个参数：cookie 的名称、子 cookie 的名称、子 cookie 的值、可选的 `Date` 对象用于设置 cookie 的过期时间、可选的 cookie 路径、可选的 cookie 域和可选的布尔值 `secure` 标志。所有可选的参数都作用于 cookie 本身，而不是子 cookie。为了在同一个 cookie 中存储多个子 cookie，路径、域和 `secure` 标志也必须相同。过期时间作用于整个 cookie，可以在写入个别子 cookie 时另行设置。在这个方法内部，第一步是取得给定 cookie 名称下包含的所有子 cookie。逻辑或操作符（`||`）在这里用于

在 getAll()返回 null 的情况下将 subcookies 设置为新对象。然后，在 subcookies 上设置完子 cookie 的值，再将参数传给 setAll()。

setAll()方法接收 6 个参数：cookie 的名称、包含所有子 cookie 的对象，然后是 set()方法中使用的 4 个可选参数。这个方法会在 for-in 循环中迭代第二个参数的属性。为保证只存储合适的数据，这里使用了 hasOwnProperty()方法确保只有实例属性才会序列化为子 cookie。因为存在属性名等于空字符串的可能，所以在添加到 subcookieParts 数组之前也要检查属性名的长度。subcookieParts 数组包含了子 cookie 的名/值对，这样我们可以方便地使用 join()方法用和号将它们拼接成字符串。剩下的逻辑与 CookieUtil.set()一样。

可以像下面这样使用这些方法：

```
// 假设 document.cookie=data=name=Nicholas&book=Professional%20JavaScript

// 设置两个子 cookie
SubCookieUtil.set("data", "name", "Nicholas");
SubCookieUtil.set("data", "book", "Professional JavaScript");

// 设置所有子 cookie 并传入过期时间
SubCookieUtil.setAll("data", { name: "Nicholas", book: "Professional JavaScript" },
    new Date("January 1, 2010"));

// 修改"name"的值并修改整个 cookie 的过期时间
SubCookieUtil.set("data", "name", "Michael", new Date("February 1, 2010"));
```

最后一组子 cookie 相关的方法是要删除子 cookie 的。常规 cookie 可以通过直接设置过期时间为某个过去的时间删除，但删除子 cookie 没有这么简单。为了删除子 cookie，需要先取得所有子 cookie，把要删除的那个删掉，然后再把剩下的子 cookie 设置回去。下面是相关方法的实现：

```
class SubCookieUtil {
  // 省略之前的代码

  static unset(name, subName, path, domain, secure) {
    let subcookies = SubCookieUtil.getAll(name);
    if (subcookies){
      delete subcookies[subName]; // 删除
      SubCookieUtil.setAll(name, subcookies, null, path, domain, secure);
    }
  }

  static unsetAll(name, path, domain, secure) {
    SubCookieUtil.setAll(name, null, new Date(0), path, domain, secure);
  }
}
```

这里定义的这两个方法有两个不同的目的。unset()方法用于从 cookie 中删除一个子 cookie，其他子 cookie 不受影响；而 unsetAll()方法与 CookieUtil.unset()一样，会删除整个 cookie。与 set()和 setAll()一样，路径、域和 secure 标志必须与创建 cookie 时使用的一样。可以像下面这样使用这两个方法：

```
// 只删除"name"子 cookie
SubCookieUtil.unset("data", "name");

// 删除整个 cookie
SubCookieUtil.unsetAll("data");
```

如果实际开发中担心碰到每个域的 cookie 限制，则可以考虑使用子 cookie 这个方案。此时要特别注意 cookie 的大小，不要超过对单个 cookie 大小的限制。

25.1.5 使用 cookie 的注意事项

还有一种叫作 HTTP-only 的 cookie。HTTP-only 可以在浏览器设置，也可以在服务器设置，但只能在服务器上读取，这是因为 JavaScript 无法取得这种 cookie 的值。

因为所有 cookie 都会作为请求头部由浏览器发送给服务器，所以在 cookie 中保存大量信息可能会影响特定域浏览器请求的性能。保存的 cookie 越大，请求完成的时间就越长。即使浏览器对 cookie 大小有限制，最好还是尽可能只通过 cookie 保存必要信息，以避免性能问题。

对 cookie 的限制及其特性决定了 cookie 并不是存储大量数据的理想方式。因此，其他客户端存储技术出现了。

> **注意** 不要在 cookie 中存储重要或敏感的信息。cookie 数据不是保存在安全的环境中，因此任何人都可能获得。应该避免把信用卡号或个人地址等信息保存在 cookie 中。

25.2 Web Storage

Web Storage 最早是网页超文本应用技术工作组（WHATWG，Web Hypertext Application Technical Working Group）在 Web Applications 1.0 规范中提出的。这个规范中的草案最终成为了 HTML5 的一部分，后来又独立成为自己的规范。Web Storage 的目的是解决通过客户端存储不需要频繁发送回服务器的数据时使用 cookie 的问题。

Web Storage 规范最新的版本是第 2 版，这一版规范主要有两个目标：

- 提供在 cookie 之外的存储会话数据的途径；
- 提供跨会话持久化存储大量数据的机制。

Web Storage 的第 2 版定义了两个对象：`localStorage` 和 `sessionStorage`。`localStorage` 是永久存储机制，`sessionStorage` 是跨会话的存储机制。这两种浏览器存储 API 提供了在浏览器中不受页面刷新影响而存储数据的两种方式。2009 年之后所有主要供应商发布的浏览器版本在 window 对象上支持 `localStorage` 和 `sessionStorage`。

> **注意** Web Storage 第 1 版曾使用过 `globalStorage`，不过目前 `globalStorage` 已废弃。

25.2.1 Storage 类型

`Storage` 类型用于保存名/值对数据，直至存储空间上限（由浏览器决定）。`Storage` 的实例与其他对象一样，但增加了以下方法。

- `clear()`：删除所有值；不在 Firefox 中实现。
- `getItem(name)`：取得给定 *name* 的值。
- `key(index)`：取得给定数值位置的名称。
- `removeItem(name)`：删除给定 *name* 的名/值对。

❑ setItem(*name*, *value*)：设置给定 *name* 的值。

getItem()、removeItem()和 setItem()方法可以直接或间接通过 Storage 对象调用。因为每个数据项都作为属性存储在该对象上，所以可以使用点或方括号操作符访问这些属性，通过同样的操作来设置值，也可以使用 delete 操作符删除属性。即便如此，通常还是建议使用方法而非属性来执行这些操作，以免意外重写某个已存在的对象成员。

通过 length 属性可以确定 Storage 对象中保存了多少名/值对。我们无法确定对象中所有数据占用的空间大小，尽管 IE8 提供了 remainingSpace 属性，用于确定还有多少存储空间（以字节计）可用。

> **注意**　Storage 类型只能存储字符串。非字符串数据在存储之前会自动转换为字符串。注意，这种转换不能在获取数据时撤销。

25.2.2　sessionStorage 对象

sessionStorage 对象只存储会话数据，这意味着数据只会存储到浏览器关闭。这跟浏览器关闭时会消失的会话 cookie 类似。存储在 sessionStorage 中的数据不受页面刷新影响，可以在浏览器崩溃并重启后恢复。（取决于浏览器，Firefox 和 WebKit 支持，IE 不支持。）

因为 sessionStorage 对象与服务器会话紧密相关，所以在运行本地文件时不能使用。存储在 sessionStorage 对象中的数据只能由最初存储数据的页面使用，在多页应用程序中的用处有限。

因为 sessionStorage 对象是 Storage 的实例，所以可以通过使用 setItem()方法或直接给属性赋值给它添加数据。下面是使用这两种方式的例子：

```
// 使用方法存储数据
sessionStorage.setItem("name", "Nicholas");

// 使用属性存储数据
sessionStorage.book = "Professional JavaScript";
```

所有现代浏览器在实现存储写入时都使用了同步阻塞方式，因此数据会被立即提交到存储。具体 API 的实现可能不会立即把数据写入磁盘（而是使用某种不同的物理存储），但这个区别在 JavaScript 层面是不可见的。通过 Web Storage 写入的任何数据都可以立即被读取。

老版 IE 以异步方式实现了数据写入，因此给数据赋值的时间和数据写入磁盘的时间可能存在延迟。对于少量数据，这里的差别可以忽略不计，但对于大量数据，就可以注意到 IE 中 JavaScript 恢复执行的速度比其他浏览器更快。这是因为实际写入磁盘的进程被转移了。在 IE8 中可以在数据赋值前调用 begin()、之后调用 commit()来强制将数据写入磁盘。比如：

```
// 仅适用于 IE8
sessionStorage.begin();
sessionStorage.name = "Nicholas";
sessionStorage.book = "Professional JavaScript";
sessionStorage.commit();
```

以上代码确保了"name"和"book"在 commit()调用之后会立即写入磁盘。调用 begin()是为了保证在代码执行期间不会有写入磁盘的操作。对于少量数据，这个过程不是必要的，但对于较大的数据量，如文档，则可以考虑使用这种事务性方法。

对存在于 sessionStorage 上的数据，可以使用 getItem()或直接访问属性名来取得。下面是使用这两种方式的例子：

```
// 使用方法取得数据
let name = sessionStorage.getItem("name");

// 使用属性取得数据
let book = sessionStorage.book;
```

可以结合 sessionStorage 的 length 属性和 key()方法遍历所有的值：

```
for (let i = 0, len = sessionStorage.length; i < len; i++){
  let key = sessionStorage.key(i);
  let value = sessionStorage.getItem(key);
  alert(`${key}=`${value}`);
}
```

这里通过 key()先取得给定位置中的数据名称，然后使用该名称通过 getItem()取得值，可以依次访问 sessionStorage 中的名/值对。

也可以使用 for-in 循环迭代 sessionStorage 的值：

```
for (let key in sessionStorage){
  let value = sessionStorage.getItem(key);
  alert(`${key}=${value}`);
}
```

每次循环，key 都会被赋予 sessionStorage 中的一个名称；这里不会返回内置方法或 length 属性。

要从 sessionStorage 中删除数据，可以使用 delete 操作符直接删除对象属性，也可以使用 removeItem()方法。下面是使用这两种方式的例子：

```
// 使用 delete 删除值
delete sessionStorage.name;

// 使用方法删除值
sessionStorage.removeItem("book");
```

sessionStorage 对象应该主要用于存储只在会话期间有效的小块数据。如果需要跨会话持久存储数据，可以使用 globalStorage 或 localStorage。

25.2.3 localStorage 对象

在修订的 HTML5 规范里，localStorage 对象取代了 globalStorage，作为在客户端持久存储数据的机制。要访问同一个 localStorage 对象，页面必须来自同一个域（子域不可以）、在相同的端口上使用相同的协议。

因为 localStorage 是 Storage 的实例，所以可以像使用 sessionStorage 一样使用 localStorage。比如下面这几个例子：

```
// 使用方法存储数据
localStorage.setItem("name", "Nicholas");

// 使用属性存储数据
localStorage.book = "Professional JavaScript";

// 使用方法取得数据
let name = localStorage.getItem("name");

// 使用属性取得数据
let book = localStorage.book;
```

两种存储方法的区别在于，存储在 localStorage 中的数据会保留到通过 JavaScript 删除或者用户清除浏览器缓存。localStorage 数据不受页面刷新影响，也不会因关闭窗口、标签页或重新启动浏览器而丢失。

25.2.4　存储事件

每当 Storage 对象发生变化时，都会在文档上触发 storage 事件。使用属性或 setItem()设置值、使用 delete 或 removeItem()删除值，以及每次调用 clear()时都会触发这个事件。这个事件的事件对象有如下 4 个属性。

- domain：存储变化对应的域。
- key：被设置或删除的键。
- newValue：键被设置的新值，若键被删除则为 null。
- oldValue：键变化之前的值。

可以使用如下代码监听 storage 事件：

```
window.addEventListener("storage",
    (event) => alert('Storage changed for ${event.domain}'));
```

对于 sessionStorage 和 localStorage 上的任何更改都会触发 storage 事件，但 storage 事件不会区分这两者。

25.2.5　限制

与其他客户端数据存储方案一样，Web Storage 也有限制。具体的限制取决于特定的浏览器。一般来说，客户端数据的大小限制是按照每个源（协议、域和端口）来设置的，因此每个源有固定大小的数据存储空间。分析存储数据的页面的源可以加强这一限制。

不同浏览器给 localStorage 和 sessionStorage 设置了不同的空间限制，但大多数会限制为每个源 5MB。关于每种媒介的新配额限制信息表，可以参考 web.dev 网站上的文章“Storage for the Web”。

要了解关于 Web Storage 限制的更多信息，可以参考 dev-test.nemikor 网站的“Web Storage Support Test”页面。

25.3　IndexedDB

Indexed Database API 简称 IndexedDB，是浏览器中存储结构化数据的一个方案。IndexedDB 用于代替目前已废弃的 Web SQL Database API。IndexedDB 背后的思想是创造一套 API，方便 JavaScript 对象的存储和获取，同时也支持查询和搜索。

IndexedDB 的设计几乎完全是异步的。为此，大多数操作以请求的形式执行，这些请求会异步执行，产生成功的结果或错误。绝大多数 IndexedDB 操作要求添加 onerror 和 onsuccess 事件处理程序来确定输出。

2017 年，新发布的主流浏览器（Chrome、Firefox、Opera、Safari）完全支持 IndexedDB。IE10/11 和 Edge 浏览器部分支持 IndexedDB。

25.3.1 数据库

IndexedDB 是类似于 MySQL 或 Web SQL Database 的数据库。与传统数据库最大的区别在于，IndexedDB 使用对象存储而不是表格保存数据。IndexedDB 数据库就是在一个公共命名空间下的一组对象存储，类似于 NoSQL 风格的实现。

使用 IndexedDB 数据库的第一步是调用 `indexedDB.open()`方法，并给它传入一个要打开的数据库名称。如果给定名称的数据库已存在，则会发送一个打开它的请求；如果不存在，则会发送创建并打开这个数据库的请求。这个方法会返回 `IDBRequest` 的实例，可以在这个实例上添加 `onerror` 和 `onsuccess` 事件处理程序。举例如下：

```
let db,
    request,
    version = 1;

request = indexedDB.open("admin", version);
request.onerror = (event) =>
  alert(`Failed to open: ${event.target.errorCode}`);
request.onsuccess = (event) => {
  db = event.target.result;
};
```

以前，IndexedDB 使用 `setVersion()`方法指定版本号。这个方法目前已废弃。如前面代码所示，要在打开数据库的时候指定版本。这个版本号会被转换为一个 `unsigned long long` 数值，因此不要使用小数，而要使用整数。

在两个事件处理程序中，`event.target` 都指向 `request`，因此使用哪个都可以。如果 `onsuccess` 事件处理程序被调用，说明可以通过 `event.target.result` 访问数据库（`IDBDatabase`）实例了，这个实例会保存到 `db` 变量中。之后，所有与数据库相关的操作都要通过 `db` 对象本身来进行。如果打开数据库期间发生错误，`event.target.errorCode` 中就会存储表示问题的错误码。

> **注意** 以前，出错时会使用 `IDBDatabaseException` 表示 IndexedDB 发生的错误。目前它已被标准的 `DOMExceptions` 取代。

25.3.2 对象存储

建立了数据库连接之后，下一步就是使用对象存储。如果数据库版本与期待的不一致，那可能需要创建对象存储。不过，在创建对象存储前，有必要想一想要存储什么类型的数据。

假设要存储包含用户名、密码等内容的用户记录。可以用如下对象来表示一条记录：

```
let user = {
  username: "007",
  firstName: "James",
  lastName: "Bond",
  password: "foo"
};
```

观察这个对象，可以很容易看出最适合作为对象存储键的 `username` 属性。用户名必须全局唯一，它也是大多数情况下访问数据的凭据。这个键很重要，因为创建对象存储时必须指定一个键。

数据库的版本决定了数据库模式，包括数据库中的对象存储和这些对象存储的结构。如果数据库还不存在，open()操作会创建一个新数据库，然后触发 upgradeneeded 事件。可以为这个事件设置处理程序，并在处理程序中创建数据库模式。如果数据库存在，而你指定了一个升级版的版本号，则会立即触发 upgradeneeded 事件，因而可以在事件处理程序中更新数据库模式。

下面的代码演示了为存储上述用户信息如何创建对象存储：

```
request.onupgradeneeded = (event) => {
  const db = event.target.result;

  // 如果存在则删除当前 objectStore。测试的时候可以这样做
  // 但这样会在每次执行事件处理程序时删除已有数据
  if (db.objectStoreNames.contains("users")) {
    db.deleteObjectStore("users");
  }

  db.createObjectStore("users", { keyPath: "username" });
};
```

这里第二个参数的 keyPath 属性表示应该用作键的存储对象的属性名。

25.3.3　事务

创建了对象存储之后，剩下的所有操作都是通过**事务**完成的。事务要通过调用数据库对象的 transaction()方法创建。任何时候，只要想要读取或修改数据，都要通过事务把所有修改操作组织起来。最简单的情况下，可以像下面这样创建事务：

```
let transaction = db.transaction();
```

如果不指定参数，则对数据库中所有的对象存储有只读权限。更具体的方式是指定一个或多个要访问的对象存储的名称：

```
let transaction = db.transaction("users");
```

这样可以确保在事务期间只加载 users 对象存储的信息。如果想要访问多个对象存储，可以给第一个参数传入一个字符串数组：

```
let transaction = db.transaction(["users", "anotherStore"]);
```

如前所述，每个事务都以只读方式访问数据。要修改访问模式，可以传入第二个参数。这个参数应该是下列三个字符串之一："readonly"、"readwrite"或"versionchange"。比如：

```
let transaction = db.transaction("users", "readwrite");
```

这样事务就可以对 users 对象存储读写了。

有了事务的引用，就可以使用 objectStore()方法并传入对象存储的名称以访问特定的对象存储。然后，可以使用 add()和 put()方法添加和更新对象，使用 get()取得对象，使用 delete()删除对象，使用 clear()删除所有对象。其中，get()和 delete()方法都接收对象键作为参数，这 5 个方法都创建新的请求对象。来看下面的例子：

```
const transaction = db.transaction("users"),
    store = transaction.objectStore("users"),
    request = store.get("007");
request.onerror = (event) => alert("Did not get the object!");
request.onsuccess = (event) => alert(event.target.result.firstName);
```

因为一个事务可以完成任意多个请求，所以事务对象本身也有事件处理程序：onerror 和 oncomplete。这两个事件可以用来获取事务级的状态信息：

```
transaction.onerror = (event) => {
  // 整个事务被取消
};
transaction.oncomplete = (event) => {
  // 整个事务成功完成
};
```

注意，不能通过 oncomplete 事件处理程序的 event 对象访问 get()请求返回的任何数据。因此，仍然需要通过这些请求的 onsuccess 事件处理程序来获取数据。

25.3.4 插入对象

拿到了对象存储的引用后，就可以使用 add()或 put()写入数据了。这两个方法都接收一个参数，即要存储的对象，并把对象保存到对象存储。这两个方法只在对象存储中已存在同名的键时有区别。这种情况下，add()会导致错误，而 put()会简单地重写该对象。更简单地说，可以把 add()想象成插入新值，而把 put()想象为更新值。因此第一次初始化对象存储时，可以这样做：

```
// users 是一个用户数据的数组
for (let user of users) {
   store.add(user);
}
```

每次调用 add()或 put()都会创建对象存储的新更新请求。如果想验证请求成功与否，可以把请求对象保存到一个变量，然后为它添加 onerror 和 onsuccess 事件处理程序：

```
// users 是一个用户数据的数组
let request,
    requests = [];
for (let user of users) {
  request = store.add(user);
  request.onerror = () => {
    // 处理错误
  };
  request.onsuccess = () => {
    // 处理成功
  };
  requests.push(request);
}
```

创建并填充了数据后，就可以查询对象存储了。

25.3.5 通过游标查询

使用事务可以通过一个已知键取得一条记录。如果想取得多条数据，则需要在事务中创建一个**游标**。游标是一个指向结果集的指针。与传统数据库查询不同，游标不会事先收集所有结果。相反，游标指向第一个结果，并在接到指令前不会主动查找下一条数据。

需要在对象存储上调用 openCursor()方法创建游标。与其他 IndexedDB 操作一样，openCursor()方法也返回一个请求，因此必须为它添加 onsuccess 和 onerror 事件处理程序。例如：