第23章

离线应用与客户端存储

本章内容

- 进行离线检测
- 使用离线缓存
- 在浏览器中保存数据

支持离线 Web 应用开发是 HTML5 的另一个重点。所谓离线 Web 应用，就是在设备不能上网的情况下仍然可以运行的应用。HTML5 把离线应用作为重点，主要是基于开发人员的心愿。前端开发人员一直希望 Web 应用能够与传统的客户端应用同场竞技，起码做到只要设备有电就能使用。

开发离线 Web 应用需要几个步骤。首先是确保应用知道设备是否能上网，以便下一步执行正确的操作。然后，应用还必须能访问一定的资源（图像、JavaScript、CSS 等），只有这样才能正常工作。最后，必须有一块本地空间用于保存数据，无论能否上网都不妨碍读写。HTML5 及其相关的 API 让开发离线应用成为现实。

23.1 离线检测

开发离线应用的第一步是要知道设备是在线还是离线，HTML5 为此定义了一个 navigator.onLine 属性，这个属性值为 true 表示设备能上网，值为 false 表示设备离线。这个属性的关键是浏览器必须知道设备能否访问网络，从而返回正确的值。实际应用中，navigator.onLine 在不同浏览器间还有些小的差异。

- IE6+和 Safari 5+能够正确检测到网络已断开，并将 navigator.onLine 的值转换为 false。
- Firefox 3+和 Opera 10.6+支持 navigator.onLine 属性，但你必须手工选中菜单项“文件→Web 开发人员（设置）→脱机工作”才能让浏览器正常工作。
- Chrome 11 及之前版本始终将 navigator.onLine 属性设置为 true。这是一个有待修复的 bug[①]。

由于存在上述兼容性问题，单独使用 navigator.onLine 属性不能确定网络是否连通。即便如此，在请求发生错误的情况下，检测这个属性仍然是管用的。以下是检测该属性状态的示例。

```
if (navigator.onLine){
    //正常工作
} else {
    //执行离线状态时的任务
```

① 这个 bug 在 2011 年 10 月已被修复（http://code.google.com/p/chromium/issues/detail?id=7469）。

```
}
```

OnLineExample01.htm

除 navigator.onLine 属性之外，为了更好地确定网络是否可用，HTML5 还定义了两个事件：online 和 offline。当网络从离线变为在线或者从在线变为离线时，分别触发这两个事件。这两个事件在 window 对象上触发。

```
EventUtil.addHandler(window, "online", function(){
    alert("Online");
});
EventUtil.addHandler(window, "offline", function(){
    alert("Offline");
});
```

OnlineEventsExample01.htm

为了检测应用是否离线，在页面加载后，最好先通过 navigator.onLine 取得初始的状态。然后，就是通过上述两个事件来确定网络连接状态是否变化。当上述事件触发时，navigator.onLine 属性的值也会改变，不过必须要手工轮询这个属性才能检测到网络状态的变化。

支持离线检测的浏览器有 IE 6+（只支持 navigator.onLine 属性）、Firefox 3、Safari 4、Opera 10.6、Chrome、iOS 3.2 版 Safari 和 Android 版 WebKit。

23.2 应用缓存

HTML5 的**应用缓存**（application cache），或者简称为 appcache，是专门为开发离线 Web 应用而设计的。Appcache 就是从浏览器的缓存中分出来的一块缓存区。要想在这个缓存中保存数据，可以使用一个**描述文件**（manifest file），列出要下载和缓存的资源。下面是一个简单的描述文件示例。

```
CACHE MANIFEST
#Comment

file.js
file.css
```

在最简单的情况下，描述文件中列出的都是需要下载的资源，以备离线时使用。

设置描述文件的选项非常多，本书不打算详细解释每一个选项。要了解这些选项，推荐读者阅读 HTML5Doctor 中的文章“Go offline with application cache”，网址为 http://html5doctor.com/go-offline-with-application-cache。

要将描述文件与页面关联起来，可以在<html>中的 manifest 属性中指定这个文件的路径，例如：

```
<html manifest="/offline.manifest">
```

以上代码告诉页面，/offline.manifest 中包含着描述文件。这个文件的 MIME 类型必须是 text/cache-manifest[①]。

① 描述文件的扩展名以前推荐用 manifest，但现在推荐的是 appcache。

虽然应用缓存的意图是确保离线时资源可用，但也有相应的 JavaScript API 让你知道它都在做什么。这个 API 的核心是 applicationCache 对象，这个对象有一个 status 属性，属性的值是常量，表示应用缓存的如下当前状态。

- 0：无缓存，即没有与页面相关的应用缓存。
- 1：闲置，即应用缓存未得到更新。
- 2：检查中，即正在下载描述文件并检查更新。
- 3：下载中，即应用缓存正在下载描述文件中指定的资源。
- 4：更新完成，即应用缓存已经更新了资源，而且所有资源都已下载完毕，可以通过 swapCache() 来使用了。
- 5：废弃，即应用缓存的描述文件已经不存在了，因此页面无法再访问应用缓存。

应用缓存还有很多相关的事件，表示其状态的改变。以下是这些事件。

- checking：在浏览器为应用缓存查找更新时触发。
- error：在检查更新或下载资源期间发生错误时触发。
- noupdate：在检查描述文件发现文件无变化时触发。
- downloading：在开始下载应用缓存资源时触发。
- progress：在文件下载应用缓存的过程中持续不断地触发。
- updateready：在页面新的应用缓存下载完毕且可以通过 swapCache() 使用时触发。
- cached：在应用缓存完整可用时触发。

一般来讲，这些事件会随着页面加载按上述顺序依次触发。不过，通过调用 update() 方法也可以手工干预，让应用缓存为检查更新而触发上述事件。

```
applicationCache.update();
```

update() 一经调用，应用缓存就会去检查描述文件是否更新（触发 checking 事件），然后就像页面刚刚加载一样，继续执行后续操作。如果触发了 cached 事件，就说明应用缓存已经准备就绪，不会再发生其他操作了。如果触发了 updateready 事件，则说明新版本的应用缓存已经可用，而此时你需要调用 swapCache() 来启用新应用缓存。

```
EventUtil.addHandler(applicationCache, "updateready", function(){
    applicationCache.swapCache();
});
```

支持 HTML5 应用缓存的浏览器有 Firefox 3+、Safari 4+、Opera 10.6、Chrome、iOS 3.2+版 Safari 及 Android 版 WebKit。在 Firefox 4 及之前版本中调用 swapCache() 会抛出错误。

23.3　数据存储

随着 Web 应用程序的出现，也产生了对于能够直接在客户端上存储用户信息能力的要求。想法很合乎逻辑，属于某个特定用户的信息应该存在该用户的机器上。无论是登录信息、偏好设定或其他数据，Web 应用提供者发现他们在找各种方式将数据存在客户端上。这个问题的第一个方案是以 cookie 的形式出现的，cookie 是原来的网景公司创造的。一份题为“Persistent Client State: HTTP Cookes”（持久客户端状态：HTTP Cookies）的标准中对 cookie 机制进行了阐述（该标准还可以在这里看到：http://curl.haxx.se/rfc/cookie_spec.html）。今天，cookie 只是在客户端存储数据的其中一种选项。

23.3.1 Cookie

HTTP Cookie，通常直接叫做 cookie，最初是在客户端用于存储会话信息的。该标准要求服务器对任意 HTTP 请求发送 Set-Cookie HTTP 头作为响应的一部分，其中包含会话信息。例如，这种服务器响应的头可能如下：

```
HTTP/1.1 200 OK
Content-type: text/html
Set-Cookie: name=value
Other-header: other-header-value
```

这个 HTTP 响应设置以 name 为名称、以 value 为值的一个 cookie，名称和值在传送时都必须是 URL 编码的。浏览器会存储这样的会话信息，并在这之后，通过为每个请求添加 Cookie HTTP 头将信息发送回服务器，如下所示：

```
GET /index.html HTTP/1.1
Cookie: name=value
Other-header: other-header-value
```

发送回服务器的额外信息可以用于唯一验证客户来自于发送的哪个请求。

1. 限制

cookie 在性质上是绑定在特定的域名下的。当设定了一个 cookie 后，再给创建它的域名发送请求时，都会包含这个 cookie。这个限制确保了储存在 cookie 中的信息只能让批准的接受者访问，而无法被其他域访问。

由于 cookie 是存在客户端计算机上的，还加入了一些限制确保 cookie 不会被恶意使用，同时不会占据太多磁盘空间。每个域的 cookie 总数是有限的，不过浏览器之间各有不同。如下所示。

- IE6 以及更低版本限制每个域名最多 20 个 cookie。
- IE7 和之后版本每个域名最多 50 个。IE7 最初是支持每个域名最大 20 个 cookie，之后被微软的一个补丁所更新。
- Firefox 限制每个域最多 50 个 cookie。
- Opera 限制每个域最多 30 个 cookie。
- Safari 和 Chrome 对于每个域的 cookie 数量限制没有硬性规定。

当超过单个域名限制之后还要再设置 cookie，浏览器就会清除以前设置的 cookie。IE 和 Opera 会删除最近最少使用过的（LRU，Least Recently Used）cookie，腾出空间给新设置的 cookie。Firefox 看上去好像是随机决定要清除哪个 cookie，所以考虑 cookie 限制非常重要，以免出现不可预期的后果。

浏览器中对于 cookie 的尺寸也有限制。大多数浏览器都有大约 4096B（加减 1）的长度限制。为了最佳的浏览器兼容性，最好将整个 cookie 长度限制在 4095B（含 4095）以内。尺寸限制影响到一个域下所有的 cookie，而并非每个 cookie 单独限制。

如果你尝试创建超过最大尺寸限制的 cookie，那么该 cookie 会被悄无声息地丢掉。注意，虽然一个字符通常占用一字节，但是多字节情况则有不同。

2. cookie 的构成

cookie 由浏览器保存的以下几块信息构成。

- 名称：一个唯一确定 cookie 的名称。cookie 名称是不区分大小写的，所以 myCookie 和 MyCookie 被认为是同一个 cookie。然而，实践中最好将 cookie 名称看作是区分大小写的，因为某些服务

器会这样处理 cookie。cookie 的名称必须是经过 URL 编码的。

- **值**：储存在 cookie 中的字符串值。值必须被 URL 编码。
- **域**：cookie 对于哪个域是有效的。所有向该域发送的请求中都会包含这个 cookie 信息。这个值可以包含子域(subdomain，如 www.wrox.com)，也可以不包含它(如.wrox.com，则对于 wrox.com 的所有子域都有效)。如果没有明确设定，那么这个域会被认作来自设置 cookie 的那个域。
- **路径**：对于指定域中的那个路径，应该向服务器发送 cookie。例如，你可以指定 cookie 只有从 http://www.wrox.com/books/ 中才能访问，那么 http://www.wrox.com 的页面就不会发送 cookie 信息，即使请求都是来自同一个域的。
- **失效时间**：表示 cookie 何时应该被删除的时间戳（也就是，何时应该停止向服务器发送这个 cookie)。默认情况下，浏览器会话结束时即将所有 cookie 删除；不过也可以自己设置删除时间。这个值是个 GMT 格式的日期（Wdy, DD-Mon-YYYY HH:MM:SS GMT），用于指定应该删除 cookie 的准确时间。因此，cookie 可在浏览器关闭后依然保存在用户的机器上。如果你设置的失效日期是个以前的时间，则 cookie 会被立刻删除。
- **安全标志**：指定后，cookie 只有在使用 SSL 连接的时候才发送到服务器。例如，cookie 信息只能发送给 https://www.wrox.com，而 http://www.wrox.com 的请求则不能发送 cookie。

每一段信息都作为 Set-Cookie 头的一部分，使用分号加空格分隔每一段，如下例所示。

```
HTTP/1.1 200 OK
Content-type: text/html
Set-Cookie: name=value; expires=Mon, 22-Jan-07 07:10:24 GMT; domain=.wrox.com
Other-header: other-header-value
```

该头信息指定了一个叫做 name 的 cookie，它会在格林威治时间 2007 年 1 月 22 日 7:10:24 失效，同时对于 www.wrox.com 和 wrox.com 的任何子域（如 p2p.wrox.com）都有效。

secure 标志是 cookie 中唯一一个非名值对儿的部分，直接包含一个 secure 单词。如下：

```
HTTP/1.1 200 OK
Content-type: text/html
Set-Cookie: name=value; domain=.wrox.com; path=/; secure
Other-header: other-header-value
```

这里，创建了一个对于所有 wrox.com 的子域和域名下（由 path 参数指定的）所有页面都有效的 cookie。因为设置了 secure 标志，这个 cookie 只能通过 SSL 连接才能传输。

尤其要注意，域、路径、失效时间和 secure 标志都是服务器给浏览器的指示，以指定何时应该发送 cookie。这些参数并不会作为发送到服务器的 cookie 信息的一部分，只有名值对儿才会被发送。

3. JavaScript 中的 cookie

在 JavaScript 中处理 cookie 有些复杂，因为其众所周知的蹩脚的接口，即 BOM 的 document.cookie 属性。这个属性的独特之处在于它会因为使用它的方式不同而表现出不同的行为。当用来获取属性值时，document.cookie 返回当前页面可用的（根据 cookie 的域、路径、失效时间和安全设置）所有 cookie 的字符串，一系列由分号隔开的名值对儿，如下例所示。

```
name1=value1;name2=value2;name3=value3
```

所有名字和值都是经过 URL 编码的，所以必须使用 decodeURIComponent()来解码。

当用于设置值的时候，document.cookie 属性可以设置为一个新的 cookie 字符串。这个 cookie 字符串会被解释并添加到现有的 cookie 集合中。设置 document.cookie 并不会覆盖 cookie，除非设置的

cookie 的名称已经存在。设置 cookie 的格式如下，和 Set-Cookie 头中使用的格式一样。

```
name=value; expires=expiration_time; path=domain_path; domain=domain_name; secure
```

这些参数中，只有 cookie 的名字和值是必需的。下面是一个简单的例子。

```
document.cookie = "name=Nicholas";
```

这段代码创建了一个叫 name 的 cookie，值为 Nicholas。当客户端每次向服务器端发送请求的时候，都会发送这个 cookie；当浏览器关闭的时候，它就会被删除。虽然这段代码没问题，但因为这里正好名称和值都无需编码，所以最好每次设置 cookie 时都像下面这个例子中一样使用 encodeURI-Component()。

```
document.cookie = encodeURIComponent("name") + "=" +
                  encodeURIComponent("Nicholas");
```

要给被创建的 cookie 指定额外的信息，只要将参数追加到该字符串，和 Set-Cookie 头中的格式一样，如下所示。

```
document.cookie = encodeURIComponent("name") + "=" +
                  encodeURIComponent("Nicholas") + "; domain=.wrox.com; path=/";
```

由于 JavaScript 中读写 cookie 不是非常直观，常常需要写一些函数来简化 cookie 的功能。基本的 cookie 操作有三种：读取、写入和删除。它们在 CookieUtil 对象中如下表示。

```
var CookieUtil = {

    get: function (name){
        var cookieName = encodeURIComponent(name) + "=",
            cookieStart = document.cookie.indexOf(cookieName),
            cookieValue = null;

        if (cookieStart > -1){
            var cookieEnd = document.cookie.indexOf(";", cookieStart);
            if (cookieEnd == -1){
                cookieEnd = document.cookie.length;
            }
            cookieValue = decodeURIComponent(document.cookie.substring(cookieStart
                          + cookieName.length, cookieEnd));
        }

        return cookieValue;
    },

    set: function (name, value, expires, path, domain, secure) {
        var cookieText = encodeURIComponent(name) + "=" +
                         encodeURIComponent(value);

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

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

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

        if (secure) {
            cookieText += "; secure";
        }

        document.cookie = cookieText;
    },

    unset: function (name, path, domain, secure){
        this.set(name, "", new Date(0), path, domain, secure);
    }

};
```

CookieUtil.js

CookieUtil.get()方法根据 cookie 的名字获取相应的值。它会在 document.cookie 字符串中查找 cookie 名加上等于号的位置。如果找到了，那么使用 indexOf()查找该位置之后的第一个分号（表示了该 cookie 的结束位置）。如果没有找到分号，则表示该 cookie 是字符串中的最后一个，则余下的字符串都是 cookie 的值。该值使用 decodeURIComponent()进行解码并最后返回。如果没有发现 cookie，则返回 null。

CookieUtil.set()方法在页面上设置一个 cookie，接收如下几个参数：cookie 的名称，cookie 的值，可选的用于指定 cookie 何时应被删除的 Date 对象，cookie 的可选的 URL 路径，可选的域，以及可选的表示是否要添加 secure 标志的布尔值。参数是按照它们的使用频率排列的，只有头两个是必需的。在这个方法中，名称和值都使用 encodeURIComponent()进行了 URL 编码，并检查其他选项。如果 expires 参数是 Date 对象，那么会使用 Date 对象的 toGMTString()方法正确格式化 Date 对象，并添加到 expires 选项上。方法的其他部分就是构造 cookie 字符串并将其设置到 document.cookie 中。

没有删除已有 cookie 的直接方法。所以，需要使用相同的路径、域和安全选项再次设置 cookie，并将失效时间设置为过去的时间。CookieUtil.unset()方法可以处理这种事情。它接收 4 个参数：要删除的 cookie 的名称、可选的路径参数、可选的域参数和可选的安全参数。

这些参数加上空字符串并设置失效时间为 1970 年 1 月 1 日（初始化为 0ms 的 Date 对象的值），传给 CookieUtil.set()。这样就能确保删除 cookie。

可以像下面这样使用上述方法。

```
//设置 cookie
CookieUtil.set("name", "Nicholas");
CookieUtil.set("book", "Professional JavaScript");

//读取 cookie 的值
alert(CookieUtil.get("name")); //"Nicholas"
alert(CookieUtil.get("book")); //"Professional JavaScript"

//删除 cookie
CookieUtil.unset("name");
CookieUtil.unset("book");
```

```
//设置 cookie，包括它的路径、域、失效日期
CookieUtil.set("name", "Nicholas", "/books/projs/", "www.wrox.com",
               new Date("January 1, 2010"));

//删除刚刚设置的 cookie
CookieUtil.unset("name", "/books/projs/", "www.wrox.com");

//设置安全的 cookie
CookieUtil.set("name", "Nicholas", null, null, null, true);
```

CookieExample01.htm

这些方法通过处理解析、构造 cookie 字符串的任务令在客户端利用 cookie 存储数据更加简单。

4. 子 cookie

为了绕开浏览器的单域名下的 cookie 数限制，一些开发人员使用了一种称为子 cookie（subcookie）的概念。子 cookie 是存放在单个 cookie 中的更小段的数据。也就是使用 cookie 值来存储多个名称值对儿。子 cookie 最常见的的格式如下所示。

```
name=name1=value1&name2=value2&name3=value3&name4=value4&name5=value5
```

子 cookie 一般也以查询字符串的格式进行格式化。然后这些值可以使用单个 cookie 进行存储和访问，而非对每个名称-值对儿使用不同的 cookie 存储。最后网站或者 Web 应用程序可以无需达到单域名 cookie 上限也可以存储更加结构化的数据。

为了更好地操作子 cookie，必须建立一系列新方法。子 cookie 的解析和序列化会因子 cookie 的期望用途而略有不同并更加复杂些。例如，要获得一个子 cookie，首先要遵循与获得 cookie 一样的基本步骤，但是在解码 cookie 值之前，需要按如下方法找出子 cookie 的信息。

```
var SubCookieUtil = {

    get: function (name, subName){
        var subCookies = this.getAll(name);
        if (subCookies){
            return subCookies[subName];
        } else {
            return null;
        }
    },

    getAll: function(name){
        var cookieName = encodeURIComponent(name) + "=",
            cookieStart = document.cookie.indexOf(cookieName),
            cookieValue = null,
            cookieEnd,
            subCookies,
            i,
            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 (i=0, len=subCookies.length; i < len; i++){
                    parts = subCookies[i].split("=");
                    result[decodeURIComponent(parts[0])] =
                        decodeURIComponent(parts[1]);
                }

                return result;
            }
        }

        return null;
    },

    //省略了更多代码
};
```

SubCookieUtil.js

获取子 cookie 的方法有两个：get()和 getAll()。其中 get()获取单个子 cookie 的值，getAll()获取所有子 cookie 并将它们放入一个对象中返回，对象的属性为子 cookie 的名称，对应值为子 cookie 对应的值。get()方法接收两个参数：cookie 的名字和子 cookie 的名字。它其实就是调用 getAll()获取所有的子 cookie，然后只返回所需的那一个（如果 cookie 不存在则返回 null）。

SubCookieUtil.getAll()方法和 CookieUtil.get()在解析 cookie 值的方式上非常相似。区别在于 cookie 的值并非立即解码，而是先根据&字符将子 cookie 分割出来放在一个数组中，每一个子 cookie 再根据等于号分割，这样在 parts 数组中的前一部分便是子 cookie 名，后一部分则是子 cookie 的值。这两个项目都要使用 decodeURIComponent()来解码，然后放入 result 对象中，最后作为方法的返回值。如果 cookie 不存在，则返回 null。

可以像下面这样使用上述方法：

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

//取得全部子 cookie
var data = SubCookieUtil.getAll("data");
alert(data.name);  //"Nicholas"
alert(data.book);  //"Professional JavaScript"

//逐个获取子 cookie
alert(SubCookieUtil.get("data", "name")); //"Nicholas"
alert(SubCookieUtil.get("data", "book")); //"Professional JavaScript"
```

SubCookiesExample01.htm

要设置子 cookie，也有两种方法：set()和 setAll()。以下代码展示了它们的构造。

```
var SubCookieUtil = {

    set: function (name, subName, value, expires, path, domain, secure) {
        var subcookies = this.getAll(name) || {};
```

```
        subcookies[subName] = value;
        this.setAll(name, subcookies, expires, path, domain, secure);
    },

    setAll: function(name, subcookies, expires, path, domain, secure){

        var 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;

    },

    //省略了更多代码
};
```

SubCookieUtil.js

这里的 `set()`方法接收 7 个参数：cookie 名称、子 cookie 名称、子 cookie 值、可选的 cookie 失效日期或时间的 `Date` 对象、可选的 cookie 路径、可选的 cookie 域和可选的布尔 `secure` 标志。所有的可选参数都是作用于 cookie 本身而非子 cookie。为了在同一个 cookie 中存储多个子 cookie，路径、域和 secure 标志必须一致；针对整个 cookie 的失效日期则可以在任何一个单独的子 cookie 写入的时候同时设置。在这个方法中，第一步是获取指定 cookie 名称对应的所有子 cookie。逻辑或操作符“||”用于当 `getAll()`

返回 null 时将 subcookies 设置为一个新对象。然后，在 subcookies 对象上设置好子 cookie 值并传给 setAll()。

而 setAll()方法接收 6 个参数：cookie 名称、包含所有子 cookie 的对象以及和 set()中一样的 4 个可选参数。这个方法使用 for-in 循环遍历第二个参数中的属性。为了确保确实是要保存的数据，使用了 hasOwnProperty()方法，来确保只有实例属性被序列化到子 cookie 中。由于可能会存在属性名为空字符串的情况，所以在把属性名加入结果对象之前还要检查一下属性名的长度。将每个子 cookie 的名值对儿都存入 subcookieParts 数组中，以便稍后可以使用 join()方法以&号组合起来。剩下的方法则和 CookieUtil.set()一样。

可以按如下方式使用这些方法。

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

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

//设置全部子 cookie 和失效日期
SubCookieUtil.setAll("data", { name: "Nicholas", book: "Professional JavaScript" },
    new Date("January 1, 2010"));

//修改名字的值，并修改 cookie 的失效日期
SubCookieUtil.set("data", "name", "Michael", new Date("February 1, 2010"));
```

SubCookiesExample01.htm

子 cookie 的最后一组方法是用于删除子 cookie 的。普通 cookie 可以通过将失效时间设置为过去的时间的方法来删除，但是子 cookie 不能这样做。为了删除一个子 cookie，首先必须获取包含在某个 cookie 中的所有子 cookie，然后仅删除需要删除的那个子 cookie，然后再将余下的子 cookie 的值保存为 cookie 的值。请看以下代码。

```
var SubCookieUtil = {

    //这里省略了更多代码

    unset: function (name, subName, path, domain, secure){
        var subcookies = this.getAll(name);
        if (subcookies){
            delete subcookies[subName];
            this.setAll(name, subcookies, null, path, domain, secure);
        }
    },

    unsetAll: function(name, path, domain, secure){
        this.setAll(name, null, new Date(0), path, domain, secure);
    }

};
```

SubCookieUtil.js

这里定义的两个方法用于两种不同的目的。unset()方法用于删除某个 cookie 中的单个子 cookie 而不影响其他的；而 unsetAll()方法则等同于 CookieUtil.unset()，用于删除整个 cookie。和 set()

及 setAll()一样，路径、域和 secure 标志必须和之前创建的 cookie 包含的内容一致。这两个方法可以像下面这样使用。

```
//仅删除名为 name 的子 cookie
SubCookieUtil.unset("data", "name");

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

如果你担心开发中可能会达到单域名的 cookie 上限，那么子 cookie 可是一个非常有吸引力的备选方案。不过，你需要更加密切关注 cookie 的长度，以防超过单个 cookie 的长度限制。

5. 关于 cookie 的思考

还有一类 cookie 被称为“HTTP 专有 cookie”。HTTP 专有 cookie 可以从浏览器或者服务器设置，但是只能从服务器端读取，因为 JavaScript 无法获取 HTTP 专有 cookie 的值。

由于所有的 cookie 都会由浏览器作为请求头发送，所以在 cookie 中存储大量信息会影响到特定域的请求性能。cookie 信息越大，完成对服务器请求的时间也就越长。尽管浏览器对 cookie 进行了大小限制，不过最好还是尽可能在 cookie 中少存储信息，以避免影响性能。

cookie 的性质和它的局限使得其并不能作为存储大量信息的理想手段，所以又出现了其他方法。

一定不要在 cookie 中存储重要和敏感的数据。cookie 数据并非存储在一个安全环境中，其中包含的任何数据都可以被他人访问。所以不要在 cookie 中存储诸如信用卡号或者个人地址之类的数据。

23.3.2 IE 用户数据

在 IE5.0 中，微软通过一个自定义行为引入了持久化用户数据的概念。用户数据允许每个文档最多 128KB 数据，每个域名最多 1MB 数据。要使用持久化用户数据，首先必须如下所示，使用 CSS 在某个元素上指定 userData 行为：

```
<div style="behavior:url(#default#userData)" id="dataStore"></div>
```

一旦该元素使用了 userData 行为，那么就可以使用 setAttribute()方法在上面保存数据了。为了将数据提交到浏览器缓存中，还必须调用 save()方法并告诉它要保存到的数据空间的名字。数据空间名字可以完全任意，仅用于区分不同的数据集。请看以下例子。

```
var dataStore = document.getElementById("dataStore");
dataStore.setAttribute("name", "Nicholas");
dataStore.setAttribute("book", "Professional JavaScript");
dataStore.save("BookInfo");
```

UserDataExample01.htm

在这段代码中， 元素上存入了两部分信息。在用 setAttribute()存储了数据之后，调用了 save()方法，指定了数据空间的名称为 BookInfo。下一次页面载入之后，可以使用 load()方法指定同样的数据空间名称来获取数据，如下所示。

```
dataStore.load("BookInfo");
```

```
alert(dataStore.getAttribute("name")); //"Nicholas"
alert(dataStore.getAttribute("book")); //"Professional JavaScript"
```

UserDataExample01.htm

对 load()的调用获取了 BookInfo 数据空间中的所有信息，并且使数据可以通过元素访问；只有到载入确切完成之后数据方能使用。如果 getAttribute()调用了不存在的名称或者是尚未载入的名程，则返回 null。

你可以通过 removeAttribute()方法明确指定要删除某元素数据，只要指定属性名称。删除之后，必须像下面这样再次调用 save()来提交更改。

```
dataStore.removeAttribute("name");
dataStore.removeAttribute("book");
dataStore.save("BookInfo");
```

UserDataExample01.htm

这段代码删除了两个数据属性，然后将更改保存到缓存中。

对 IE 用户数据的访问限制和对 cookie 的限制类似。要访问某个数据空间，脚本运行的页面必须来自同一个域名，在同一个路径下，并使用与进行存储的脚本同样的协议。和 cookie 不同的是，你无法将用户数据访问限制扩展到更多的客户。还有一点不同，用户数据默认是可以跨越会话持久存在的，同时也不会过期；数据需要通过 removeAttribute()方法专门进行删除以释放空间。

和 cookie 一样，IE 用户数据并非安全的，所以不能存放敏感信息。

23.3.3 Web 存储机制

Web Storage 最早是在 Web 超文本应用技术工作组（WHAT-WG）的 Web 应用 1.0 规范中描述的。这个规范的最初的工作最终成为了 HTML5 的一部分。Web Storage 的目的是克服由 cookie 带来的一些限制，当数据需要被严格控制在客户端上时，无须持续地将数据发回服务器。Web Storage 的两个主要目标是：

- 提供一种在 cookie 之外存储会话数据的途径；
- 提供一种存储大量可以跨会话存在的数据的机制。

最初的 Web Storage 规范包含了两种对象的定义：sessionStorage 和 globalStorage。这两个对象在支持的浏览器中都是以 windows 对象属性的形式存在的，支持这两个属性的浏览器包括 IE8+、Firefox 3.5+、Chrome 4+和 Opera 10.5+。

Firefox 2 和 3 基于早期规范的内容部分实现了 Web Storage，当时只实现了 globalStorage，没有实现 localStorage。

1. Storage 类型

Storage 类型提供最大的存储空间（因浏览器而异）来存储名值对儿。Storage 的实例与其他对象类似，有如下方法。

- clear()：删除所有值；Firefox 中没有实现 。

- getItem(name)：根据指定的名字 name 获取对应的值。
- key(index)：获得 index 位置处的值的名字。
- removeItem(name)：删除由 name 指定的名值对儿。
- setItem(name, value)：为指定的 name 设置一个对应的值。

其中，getItem()、removeItem()和 setItem()方法可以直接调用，也可通过 Storage 对象间接调用。因为每个项目都是作为属性存储在该对象上的，所以可以通过点语法或者方括号语法访问属性来读取值，设置也一样，或者通过 delete 操作符进行删除。不过，我们还建议读者使用方法而不是属性来访问数据，以免某个键会意外重写该对象上已经存在的成员。

还可以使用 length 属性来判断有多少名值对儿存放在 Storage 对象中。但无法判断对象中所有数据的大小，不过 IE8 提供了一个 remainingSpace 属性，用于获取还可以使用的存储空间的字节数。

Storage 类型只能存储字符串。非字符串的数据在存储之前会被转换成字符串。

2. sessionStorage 对象

sessionStorage 对象存储特定于某个会话的数据，也就是该数据只保持到浏览器关闭。这个对象就像会话 cookie，也会在浏览器关闭后消失。存储在 sessionStorage 中的数据可以跨越页面刷新而存在，同时如果浏览器支持，浏览器崩溃并重启之后依然可用（Firefox 和 WebKit 都支持，IE 则不行）。

因为 seesionStorage 对象绑定于某个服务器会话，所以当文件在本地运行的时候是不可用的。存储在 sessionStorage 中的数据只能由最初给对象存储数据的页面访问到，所以对多页面应用有限制。

由于 sessionStorage 对象其实是 Storage 的一个实例，所以可以使用 setItem()或者直接设置新的属性来存储数据。下面是这两种方法的例子。

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

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

SessionStorageExample01.htm

不同浏览器写入数据方面略有不同。Firefox 和 WebKit 实现了同步写入，所以添加到存储空间中的数据是立刻被提交的。而 IE 的实现则是异步写入数据，所以在设置数据和将数据实际写入磁盘之间可能有一些延迟。对于少量数据而言，这个差异是可以忽略的。对于大量数据，你会发现 IE 要比其他浏览器更快地恢复执行，因为它会跳过实际的磁盘写入过程。

在 IE8 中可以强制把数据写入磁盘：在设置新数据之前使用 begin()方法，并且在所有设置完成之后调用 commit()方法。看以下例子。

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

这段代码确保了 name 和 book 的值在调用 commit()之后立刻被写入磁盘。调用 begin()是为了确保在这段代码执行的时候不会发生其他磁盘写入操作。对于少量数据而言，这个过程不是必需的；不

过，对于大量数据（如文档之类的）可能就要考虑这种事务形式的方法了。

sessionStorage 中有数据时，可以使用 getItem()或者通过直接访问属性名来获取数据。两种方法的例子如下。

```
//使用方法读取数据
var name = sessionStorage.getItem("name");

//使用属性读取数据
var book = sessionStorage.book;
```

SessionStorageExample01.htm

还可以通过结合 length 属性和 key()方法来迭代 sessionStorage 中的值，如下所示。

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

SessionStorageExample01.htm

它是这样遍历 sessionStorage 中的名值对儿的：首先通过 key()方法获取指定位置上的名字，然后再通过 getItem()找出对应该名字的值。

还可以使用 for-in 循环来迭代 sessionStorage 中的值：

```
for (var key in sessionStorage){
    var value = sessionStorage.getItem(key);
    alert(key + "=" + value);
}
```

每次经过循环的时候，key 被设置为 sessionStorage 中下一个名字，此时不会返回任何内置方法或 length 属性。

要从 sessionStorage 中删除数据，可以使用 delete 操作符删除对象属性，也可调用 removeItem()方法。以下是这些方法的例子。

```
//使用 delete 删除一个值——在 WebKit 中无效
delete sessionStorage.name;

//使用方法删除一个值
sessionStorage.removeItem("book");
```

SessionStorageExample01.htm

在撰写本书时，delete 操作符在 WebKit 中无法删除数据，removeItem()则可以在各种支持的浏览器中正确运行。

sessionStorage 对象应该主要用于仅针对会话的小段数据的存储。如果需要跨越会话存储数据，那么 globalStorage 或者 localStorage 更为合适。

3. globalStorage 对象

Firefox 2 中实现了 globalStorage 对象。作为最初的 Web Storage 规范的一部分，这个对象的目的是跨越会话存储数据，但有特定的访问限制。要使用 globalStorage，首先要指定哪些域可以访问

该数据。可以通过方括号标记使用属性来实现，如以下例子所示。

```
//保存数据
globalStorage["wrox.com"].name = "Nicholas";

//获取数据
var name = globalStorage["wrox.com"].name;
```

GlobalStorageExample01.htm

在这里，访问的是针对域名 wrox.com 的存储空间。globalStorage 对象不是 Storage 的实例，而具体的 globalStorage["wrox.com"]才是。这个存储空间对于 wrox.com 及其所有子域都是可以访问的。可以像下面这样指定子域名。

```
//保存数据
globalStorage["www.wrox.com"].name = "Nicholas";

//获取数据
var name = globalStorage["www.wrox.com"].name;
```

GlobalStorageExample01.htm

这里所指定的存储空间只能由来自 www.wrox.com 的页面访问，其他子域名都不行。

某些浏览器允许更加宽泛的访问限制，比如只根据顶级域名进行限制或者允许全局访问，如下面例子所示。

```
//存储数据，任何人都可以访问——不要这样做!
globalStorage[""].name = "Nicholas";

//存储数据，可以让任何以.net 结尾的域名访问——不要这样做!
globalStorage["net"].name = "Nicholas";
```

虽然这些也支持，但是还是要避免使用这种可宽泛访问的数据存储，以防止出现潜在的安全问题。考虑到安全问题，这些功能在未来可能会被删除或者是被更严格地限制，所以不应依赖于这类功能。当使用 globalStorage 的时候一定要指定一个域名。

对 globalStorage 空间的访问，是依据发起请求的页面的域名、协议和端口来限制的。例如，如果使用 HTTPS 协议在 wrox.com 中存储了数据，那么通过 HTTP 访问的 wrox.com 的页面就不能访问该数据。同样，通过 80 端口访问的页面则无法与同一个域同样协议但通过 8080 端口访问的页面共享数据。这类似于 Ajax 请求的同源策略。

globalStorage 的每个属性都是 Storage 的实例。因此，可以像如下代码中这样使用。

```
globalStorage["www.wrox.com"].name = "Nicholas";
globalStorage["www.wrox.com"].book = "Professional JavaScript";

globalStorage["www.wrox.com"].removeItem("name");

var book = globalStorage["www.wrox.com"].getItem("book");
```

GlobalStorageExample01.htm

如果你事先不能确定域名，那么使用 location.host 作为属性名比较安全。例如：

```
globalStorage[location.host].name = "Nicholas";
var book = globalStorage[location.host].getItem("book");
```

GlobalStorageExample01.htm

如果不使用 removeItem()或者 delete 删除，或者用户未清除浏览器缓存，存储在 globalStorage 属性中的数据会一直保留在磁盘上。这让 globalStorage 非常适合在客户端存储文档或者长期保存用户偏好设置。

4. localStorage 对象

localStorage 对象在修订过的 HTML 5 规范中作为持久保存客户端数据的方案取代了 globalStorage。与 globalStorage 不同，不能给 localStorage 指定任何访问规则；规则事先就设定好了。要访问同一个 localStorage 对象，页面必须来自同一个域名（子域名无效），使用同一种协议，在同一个端口上。这相当于 globalStorage[location.host]。

由于 localStorage 是 Storage 的实例，所以可以像使用 sessionStorage 一样来使用它。下面是一些例子。

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

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

//使用方法读取数据
var name = localStorage.getItem("name");

//使用属性读取数据
var book = localStorage.book;
```

LocalStorageExample01.htm

存储在 localStorage 中的数据和存储在 globalStorage 中的数据一样，都遵循相同的规则：数据保留到通过 JavaScript 删除或者是用户清除浏览器缓存。

为了兼容只支持 globalStorage 的浏览器，可以使用以下函数。

```
function getLocalStorage(){
    if (typeof localStorage == "object"){
        return localStorage;
    } else if (typeof globalStorage == "object"){
        return globalStorage[location.host];
    } else {
        throw new Error("Local storage not available.");
    }
}
```

GlobalAndLocalStorageExample01.htm

然后，像下面这样调用一次这个函数，就可以正常地读写数据了。

```
var storage = getLocalStorage();
```

GlobalAndLocalStorageExample01.htm

在确定了使用哪个 Storage 对象之后，就能在所有支持 Web Storage 的浏览器中使用相同的存取规则操作数据了。

5. storage 事件

对 Storage 对象进行任何修改，都会在文档上触发 storage 事件。当通过属性或 setItem()方法保存数据，使用 delete 操作符或 removeItem()删除数据，或者调用 clear()方法时，都会发生该事件。这个事件的 event 对象有以下属性。

- domain：发生变化的存储空间的域名。
- key：设置或者删除的键名。
- newValue：如果是设置值，则是新值；如果是删除键，则是 null。
- oldValue：键被更改之前的值。

在这四个属性中，IE8 和 Firefox 只实现了 domain 属性。在撰写本书的时候，WebKit 尚不支持 storage 事件：

以下代码展示了如何侦听 storage 事件：

```
EventUtil.addHandler(document, "storage", function(event){
    alert("Storage changed for " + event.domain);
});
```

StorageEventExample01.htm

无论对 sessionStorage、globalStorage 还是 localStorage 进行操作，都会触发 storage 事件，但不作区分。

6. 限制

与其他客户端数据存储方案类似，Web Storage 同样也有限制。这些限制因浏览器而异。一般来说，对存储空间大小的限制都是以每个来源（协议、域和端口）为单位的。换句话说，每个来源都有固定大小的空间用于保存自己的数据。考虑到这个限制，就要注意分析和控制每个来源中有多少页面需要保存数据。

对于 localStorage 而言，大多数桌面浏览器会设置每个来源 5MB 的限制。Chrome 和 Safari 对每个来源的限制是 2.5MB。而 iOS 版 Safari 和 Android 版 WebKit 的限制也是 2.5MB。

对 sessionStorage 的限制也是因浏览器而异。有的浏览器对 sessionStorage 的大小没有限制，但 Chrome、Safari、iOS 版 Safari 和 Android 版 WebKit 都有限制，也都是 2.5MB。IE8+和 Opera 对 sessionStorage 的限制是 5MB。

有关 Web Storage 的限制，请参考 http://dev-test.nemikor.com/web-storage/support-test/。

23.3.4 IndexedDB

Indexed Database API，或者简称为 IndexedDB，是在浏览器中保存结构化数据的一种数据库。IndexedDB 是为了替代目前已被废弃的 Web SQL Database API（因为已废弃，所以本书未介绍）而出现的。IndexedDB 的思想是创建一套 API，方便保存和读取 JavaScript 对象，同时还支持查询及搜索。

IndexedDB 设计的操作完全是异步进行的。因此，大多数操作会以请求方式进行，但这些操作会在后期执行，然后如果成功则返回结果，如果失败则返回错误。差不多每一次 IndexedDB 操作，都需要你注册 onerror 或 onsuccess 事件处理程序，以确保适当地处理结果。

在得到完整支持的情况下，IndexedDB 将是一个作为 API 宿主的全局对象。由于 API 仍然可能有

变化，浏览器也都使用提供商前缀，因此这个对象在 IE10 中叫 msIndexedDB，在 Firefox 4 中叫 mozIndexedDB，在 Chrome 中叫 webkitIndexedDB。为了清楚起见，本节示例中将使用 IndexedDB，而实际上每个示例前面都应该加上下面这行代码：

```
var indexedDB = window.indexedDB || window.msIndexedDB || window.mozIndexedDB ||
window.webkitIndexedDB;
```

IndexedDBExample01.htm

1. 数据库

IndexedDB 就是一个数据库，与 MySQL 或 Web SQL Database 等这些你以前可能用过的数据库类似。IndexedDB 最大的特色是使用对象保存数据，而不是使用表来保存数据。一个 IndexedDB 数据库，就是一组位于相同命名空间下的对象的集合。

使用 IndexedDB 的第一步是打开它，即把要打开的数据库名传给 indexDB.open()。如果传入的数据库已经存在，就会发送一个打开它的请求；如果传入的数据库还不存在，就会发送一个创建并打开它的请求。总之，调用 indexDB.open()会返回一个 IDBRequest 对象，在这个对象上可以添加 onerror 和 onsuccess 事件处理程序。先来看一个例子。

```
var request, database;

request = indexedDB.open("admin");
request.onerror = function(event){
    alert("Something bad happened while trying to open: " +
          event.target.errorCode);
};
request.onsuccess = function(event){
    database = event.target.result;
};
```

IndexedDBExample01.htm

在这两个事件处理程序中，event.target 都指向 request 对象，因此它们可以互换使用。如果响应的是 onsuccess 事件处理程序，那么 event.target.result 中将有一个数据库实例对象（IDBDatabase），这个对象会保存在 database 变量中。如果发生了错误，那 event.target.errorCode 中将保存一个错误码，表示问题的性质。以下就是可能的错误码（这个错误码适合所有操作）。

- IDBDatabaseException.UNKNOWN_ERR(1)：意外错误，无法归类。
- IDBDatabaseException.NON_TRANSIENT_ERR(2)：操作不合法。
- IDBDatabaseException.NOT_FOUND_ERR(3)：未发现要操作的数据库。
- IDBDatabaseException.CONSTRAINT_ERR(4)：违反了数据库约束。
- IDBDatabaseException.DATA_ERR(5)：提供给事务的数据不能满足要求。
- IDBDatabaseException.NOT_ALLOWED_ERR(6)：操作不合法。
- IDBDatabaseException.TRANSACTION_INACTIVE_ERR(7)：试图重用已完成的事务。
- IDBDatabaseException.ABORT_ERR(8)：请求中断，未成功。
- IDBDatabaseException.READ_ONLY_ERR(9)：试图在只读模式下写入或修改数据。
- IDBDatabaseException.TIMEOUT_ERR(10)：在有效时间内未完成操作。
- IDBDatabaseException.QUOTA_ERR(11)：磁盘空间不足。

默认情况下，IndexedDB 数据库是没有版本号的，最好一开始就为数据库指定一个版本号。为此，可以调用 setVersion()方法，传入以字符串形式表示的版本号。同样，调用这个方法也会返回一个请求对象，需要你再指定事件处理程序。

```
if (database.version != "1.0"){
    request = database.setVersion("1.0");
    request.onerror = function(event){
        alert("Something bad happened while trying to set version: " +
              event.target.errorCode);
    };
    request.onsuccess = function(event){
        alert("Database initialization complete. Database name: " + database.name +
              ", Version: " + database.version);
    };
} else {
    alert("Database already initialized. Database name: " + database.name +
          ", Version: " + database.version);
}
```

IndexedDBExample01.htm

这个例子尝试把数据库的版本号设置为 1.0。第一行先检测 version 属性，看是否已经为数据库设置了相应的版本号。如果没有，就调用 setVersion()创建修改版本的请求。如果请求成功，显示一条消息，表示版本修改成功。（在真实的项目开发中，你应该在这里建立对象存储空间。详细内容请看下一节。）

如果数据库的版本号已经被设置为 1.0，则显示一条消息，说明数据库已经初始化过了。总之，通过这种模式，就能知道你想使用的数据库是否已经设置了适当的对象存储空间。在整个 Web 应用中，随着对数据库结构的更新和修改，可能会产生很多个不同版本的数据库。

2. 对象存储空间

在建立了与数据库的连接之后，下一步就是使用对象存储空间[①]。如果数据库的版本与你传入的版本不匹配，那可能就需要创建一个新的对象存储空间。在创建对象存储空间之前，必须要想清楚你想要保存什么数据类型。

假设你要保存的用户记录由用户名、密码等组成，那么保存一条记录的对象应该类似如下所示：

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

有了这个对象，很容易想到 username 属性可以作为这个对象存储空间的键。这个 username 必须全局唯一，而且大多数时候都要通过这个键来访问数据。这一点非常重要，因为在创建对象存储空间时，必须指定这么一个键。以下是就是为保存上述用户记录而创建对象存储空间的示例。

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

IndexedDBExample02.htm

① 有关系数据库经验的读者，可以把这里的对象存储空间（object storge）想象成表，而把其中保存的对象想象成表中的记录。

其中第二个参数中的 keyPath 属性，就是空间中将要保存的对象的一个属性，而这个属性将作为存储空间的键来使用。

好，现在有了一个对存储空间的引用。接下来可以使用 add()或 put()方法来向其中添加数据。这两个方法都接收一个参数，即要保存的对象，然后这个对象就会被保存到存储空间中。这两个方法的区别在空间中已经包含键值相同的对象时会体现出来。在这种情况下，add()会返回错误，而 put()则会重写原有对象。简单地说，可以把 add()想象成插入新值，把 put()想象成更新原有的值。在初始化对象存储空间时，可以使用类似下面这样的代码。

```
//users 中保存着一批用户对象
var i=0,
    len = users.length;

while(i < len){
    store.add(users[i++]);
}
```

IndexedDBExample02.htm

每次调用 add()或 put()都会创建一个新的针对这个对象存储空间的更新请求。如果想验证请求是否成功完成，可以把返回的请求对象保存在一个变量中，然后再指定 onerror 或 onsuccess 事件处理程序。

```
//users 中保存着一批用户对象
var i=0,
    request,
    requests = [],
    len = users.length;

while(i < len){
    request = store.add(users[i++]);
    request.onerror = function(){
        //处理错误
    };
    request.onsuccess = function(){
        //处理成功
    };
    requests.push(request);
}
```

创建了对象存储空间并向其中添加了数据之后，就该查询数据了。

3. 事务

跨过创建对象存储空间这一步之后，接下来的所有操作都是通过事务来完成的。在数据库对象上调用 transaction()方法可以创建事务。任何时候，只要想读取或修改数据，都要通过事务来组织所有操作。在最简单的情况下，可以像下面这样创建事务[①]。

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

如果没有参数，就只能通过事务来读取数据库中保存的对象。最常见的方式是传入要访问的一或多个对象存储空间。

① 以下示例代码中的 db 即前面示例代码中的 database，正文中提到的“数据库对象”也是指它。

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

这样就能保证只加载 users 存储空间中的数据，以便通过事务进行访问。如果要访问多个对象存储空间，也可以在第一个参数的位置上传入字符串数组。

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

如前所述，这些事务都是以只读方式访问数据。要修改访问方式，必须在创建事务时传入第二个参数，这个参数表示访问模式，用 IDBTransaction 接口定义的如下常量表示：READ_ONLY（0）表示只读，READ_WRITE（1）表示读写，VERSION_CHANGE（2）表示改变。IE10+和 Firefox 4+实现的是 IDBTransaction，但在 Chrome 中则叫 webkitIDBTransaction，所以使用下面的代码可以统一接口：

```
var IDBTransaction = window.IDBTransaction || window.webkitIDBTransaction;
```

IndexedDBExample03.htm

有了这行代码，就可以更方便地为 transaction()指定第二个参数了。

```
var transaction = db.transaction("users", IDBTransaction.READ_WRITE);
```

IndexedDBExample03.htm

这个事务能够读写 users 存储空间。

取得了事务的索引后，使用 objectStore()方法并传入存储空间的名称，就可以访问特定的存储空间。然后，可以像以前一样使用 add()和 put()方法，使用 get()可以取得值，使用 delete()可以删除对象，而使用 clear()则可以删除所有对象。get()和 delete()方法都接收一个对象键作为参数，而所有这 5 个方法都会返回一个新的请求对象。例如：

```
var request = db.transaction("users").objectStore("users").get("007");
request.onerror = function(event){
    alert("Did not get the object!");
};
request.onsuccess = function(event){
    var result = event.target.result;
    alert(result.firstName);     //"James"
};
```

IndexedDBExample02.htm

因为一个事务可以完成任何多个请求，所以事务对象本身也有事件处理程序：onerror 和 oncomplete。这两个事件可以提供事务级的状态信息。

```
transaction.onerror = function(event){
    //整个事务都被取消了
};

transaction.oncomplete = function(event){
    //整个事务都成功完成了
};
```

注意，通过 oncomplete 事件的事件对象（event）访问不到 get()请求返回的任何数据。必须在相应请求的 onsuccess 事件处理程序中才能访问到数据。

4. 使用游标查询

使用事务可以直接通过已知的键检索单个对象。而在需要检索多个对象的情况下，则需要在事务内部创建游标。游标就是一指向结果集的指针。与传统数据库查询不同，游标并不提前收集结果。游标指针会先指向结果中的第一项，在接到查找下一项的指令时，才会指向下一项。

在对象存储空间上调用 openCursor()方法可以创建游标。与 IndexedDB 中的其他操作一样，openCursor()方法返回的是一个请求对象，因此必须为该对象指定 onsuccess 和 onerror 事件处理程序。例如：

```
var store = db.transaction("users").objectStore("users"),
    request = store.openCursor();

request.onsuccess = function(event){
    //处理成功
};

request.onerror = function(event){
    //处理失败
};
```

IndexedDBExample04.htm

在 onsuccess 事件处理程序执行时，可以通过 event.target.result 取得存储空间中的下一个对象。在结果集中有下一项时，这个属性中保存一个 IDBCursor 的实例，在没有下一项时，这个属性的值为 null。IDBCursor 的实例有以下几个属性。

- direction：数值，表示游标移动的方向。默认值为 IDBCursor.NEXT（0），表示下一项。IDBCursor.NEXT_NO_DUPLICATE（1）表示下一个不重复的项，DBCursor.PREV（2）表示前一项，而 IDBCursor.PREV_NO_DUPLICATE 表示前一个不重复的项。
- key：对象的键。
- value：实际的对象。
- primaryKey：游标使用的键。可能是对象键，也可能是索引键（稍后讨论索引键）。

要检索某一个结果的信息，可以像下面这样：

```
request.onsuccess = function(event){
    var cursor = event.target.result;
    if (cursor){ //必须要检查
        console.log("Key: " + cursor.key + ", Value: " +
                    JSON.stringify(cursor.value));
    }
};
```

请记住，这个例子中的 cursor.value 是一个对象，这也是为什么在显示它之前先将它转换成 JSON 字符串的原因。

使用游标可以更新个别的记录。调用 update()方法可以用指定的对象更新当前游标的 value。与其他操作一样，调用 update()方法也会创建一个新请求，因此如果你想知道结果，就要为它指定 onsuccess 和 onerror 事件处理程序。

```
request.onsuccess = function(event){
    var cursor = event.target.result,
        value,
```

```
        updateRequest;

    if (cursor){ //必须要检查
        if (cursor.key == "foo"){
            value = cursor.value;                    //取得当前的值
            value.password = "magic!";               //更新密码

            updateRequest = cursor.update(value);    //请求保存更新
            updateRequest.onsuccess = function(){
                //处理成功
            };
            updateReqeust.onerror = function(){
                //处理失败
            };
        }
    }
};
```

此时，如果调用 delete()方法，就会删除相应的记录。与 update()一样，调用 delete()也返回一个请求。

```
request.onsuccess = function(event){
    var cursor = event.target.result,
        value,
        deleteRequest;

    if (cursor){  //必须要检查
        if (cursor.key == "foo"){
            deleteRequest = cursor.delete();      //请求删除当前项
            deleteRequest.onsuccess = function(){
                //处理成功
            };
            deleteRequest.onerror = function(){
                //处理失败
            };
        }
    }
};
```

如果当前事务没有修改对象存储空间的权限，update()和 delete()会抛出错误。

默认情况下，每个游标只发起一次请求。要想发起另一次请求，必须调用下面的一个方法。

- continue(*key*)：移动到结果集中的下一项。参数 *key* 是可选的，不指定这个参数，游标移动到下一项；指定这个参数，游标会移动到指定键的位置。
- advance(*count*)：向前移动 *count* 指定的项数。

这两个方法都会导致游标使用相同的请求，因此相同的 onsuccess 和 onerror 事件处理程序也会得到重用。例如，下面的例子遍历了对象存储空间中的所有项。

```
request.onsuccess = function(event){
    var cursor = event.target.result;
    if (cursor){ //必须要检查
        console.log("Key: " + cursor.key + ", Value: " +
                    JSON.stringify(cursor.value));
        cursor.continue(); //移动到下一项
    } else {
        console.log("Done!");
```

```
    }
};
```

调用 continue()会触发另一次请求，进而再次调用 onsuccess 事件处理程序。在没有更多项可以迭代时，将最后一次调用 onsuccess 事件处理程序，此时 event.target.result 的值为 null。

5. 键范围

使用游标总让人觉得不那么理想，因为通过游标查找数据的方式太有限了。键范围（key range）为使用游标增添了一些灵活性。键范围由 IDBKeyRange 的实例表示。支持标准 IDBKeyRange 类型的浏览器有 IE10+和 Firefox 4+，Chrome 中的名字叫 webkitIDBKeyRange。与使用 IndexedDB 中的其他类型一样，你最好先声明一个本地的类型，同时要考虑到不同浏览器中的差异。

```
var IDBKeyRange = window.IDBKeyRange || window.webkitIDBKeyRange;
```

有四种定义键范围的方式。第一种是使用 only()方法，传入你想要取得的对象的键。

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

这个范围可以保证只取得键为"007"的对象。使用这个范围创建的游标与直接访问存储空间并调用 get("007")差不多。

第二种定义键范围的方式是指定结果集的下界。下界表示游标开始的位置。例如，以下键范围可以保证游标从键为"007"的对象开始，然后继续向前移动，直至最后一个对象。

```
//从键为"007"的对象开始，然后可以移动到最后
var lowerRange = IDBKeyRange.lowerBound("007");
```

如果你想忽略键为"007"的对象，从它的下一个对象开始，那么可以传入第二个参数 true：

```
//从键为"007"的对象的下一个对象开始，然后可以移动到最后
var lowerRange = IDBKeyRange.lowerBound("007", true);
```

第三种定义键范围的方式是指定结果集的上界，也就是指定游标不能超越哪个键。指定上界使用 upperRange()方法。下面这个键范围可以保证游标从头开始，到取得键为"ace"的对象终止。

```
//从头开始，到键为"ace"的对象为止
var upperRange = IDBKeyRange.upperBound("ace");
```

如果你不想包含键为指定值的对象，同样，传入第二个参数 true：

```
//从头开始，到键为"ace"的对象的上一个对象为止
var upperRange = IDBKeyRange.upperBound("ace", true);
```

第四种定义键范围的方式——没错，就是同时指定上、下界，使用 bound()方法。这个方法可以接收 4 个参数：表示下界的键、表示上界的键、可选的表示是否跳过下界的布尔值和可选的表示是否跳过上界的布尔值。以下是几个例子。

```
//从键为"007"的对象开始，到键为"ace"的对象为止
var boundRange = IDBKeyRange.bound("007", "ace");

//从键为"007"的对象的下一个对象开始，到键为"ace"的对象为止
var boundRange = IDBKeyRange.bound("007", "ace", true);

//从键为"007"的对象的下一个对象开始，到键为"ace"的对象的上一个对象为止
var boundRange = IDBKeyRange.bound("007", "ace", true, true);
```

```
//从键为"007"的对象开始，到键为"ace"的对象的上一个对象为止
var boundRange = IDBKeyRange.bound("007", "ace", false, true);
```

无论如何，在定义键范围之后，把它传给 openCursor()方法，就能得到一个符合相应约束条件的游标。

```
var store = db.transaction("users").objectStore("users"),
    range = IDBKeyRange.bound("007", "ace");
    request = store.openCursor(range);

request.onsuccess = function(event){
    var cursor = event.target.result;
    if (cursor){ //必须要检查
        console.log("Key: " + cursor.key + ", Value: " +
                    JSON.stringify(cursor.value));
        cursor.continue(); //移动到下一项
    } else {
        console.log("Done!");
    }
};
```

这个例子输出的对象的键为"007"到"ace"，比上一节最后那个例子输出的值少一些。

6. 设定游标方向

实际上，openCursor()可以接收两个参数。第一个参数就是刚刚看到的 IDBKeyRange 的实例，第二个是表示方向的数值常量。作为第二个参数的常量是前面讲查询时介绍的 IDBCursor 中的常量。Fire fox4 +和 Chrome 的实现又有不同，因此第一步还是在本地消除差异：

```
var IDBCursor = window.IDBCursor || window.webkitIDBCursor;
```

正常情况下，游标都是从存储空间的第一项开始，调用 continue()或 advance()前进到最后一项。游标的默认方向值是 IDBCursor.NEXT。如果对象存储空间中有重复的项，而你想让游标跳过那些重复的项，可以为 openCursor 传入 IDBCursor.NEXT_NO_DUPLICATE 作为第二个参数：

```
var store = db.transaction("users").objectStore("users"),
    request = store.openCursor(null, IDBCursor.NEXT_NO_DUPLICATE);
```

注意，openCursor()的第一个参数是 null，表示使用默认的键范围，即包含所有对象。这个游标可以从存储空间中的第一个对象开始，逐个迭代到最后一个对象——但会跳过重复的对象。

当然，也可以创建一个游标，让它在对象存储空间中向后移动，即从最后一个对象开始，逐个迭代，直至第一个对象。此时，要传入的常量是 IDBCursor.PREV 和 IDBCursor.PREV_NO_DUPLICATE。例如：

```
var store = db.transaction("users").objectStore("users"),
    request = store.openCursor(null, IDBCursor.PREV);
```

IndexedDBExample05.htm

使用 IDBCursor.PREV 或 IDBCursor.PREV_NO_DUPLICATE 打开游标时，每次调用 continue()或 advance()，都会在存储空间中向后而不是向前移动游标。

7. 索引

对于某些数据，可能需要为一个对象存储空间指定多个键。比如，若要通过用户 ID 和用户名两种方式来保存用户资料，就需要通过这两个键来存取记录。为此，可以考虑将用户 ID 作为主键，然后为

用户名创建索引。

要创建索引，首先引用对象存储空间，然后调用 createIndex()方法，如下所示。

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

createIndex()的第一个参数是索引的名字，第二个参数是索引的属性的名字，第三个参数是一个包含 unique 属性的选项（options）对象。这个选项通常都必须指定，因为它表示键在所有记录中是否唯一。因为 username 有可能重复，所以这个索引不是唯一的。

createIndex()的返回值是 IDBIndex 的实例。在对象存储空间上调用 index()方法也能返回同一个实例。例如，要使用一个已经存在的名为"username"的索引，可以像下面这样取得该索引。

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

索引其实与对象存储空间很相似。在索引上调用 openCursor()方法也可以创建新的游标，除了将来会把索引键而非主键保存在 event.result.key 属性中之外，这个游标与在对象存储空间上调用 openCursor()返回的游标完全一样。来看下面的例子。

```
var store = db.transaction("users").objectStore("users"),
    index = store.index("username"),
    request = index.openCursor();

request.onsuccess = function(event){
    //处理成功
};
```

在索引上也能创建一个特殊的只返回每条记录主键的游标，那就要调用 openKeyCursor()方法。这个方法接收的参数与 openCursor()相同。而最大的不同在于，这种情况下 event.result.key 中仍然保存着索引键，而 event.result.value 中保存的则是主键，而不再是整个对象。

```
var store = db.transaction("users").objectStore("users"),
    index = store.index("username"),
    request = index.openKeyCursor();

request.onsuccess = function(event){
    //处理成功
    // event.result.key 中保存索引键，而 event.result.value 中保存主键
};
```

同样，使用 get()方法能够从索引中取得一个对象，只要传入相应的索引键即可；当然，这个方法也将返回一个请求。

```
var store = db.transaction("users").objectStore("users"),
    index = store.index("username"),
    request = index.get("007");

request.onsuccess = function(event){
    //处理成功
};

request.onerror = function(event){
    //处理失败
};
```

要根据给定的索引键取得主键，可以使用 getKey()方法。这个方法也会创建一个新的请求，但 event.result.value 等于主键的值，而不是包含整个对象。

```
var store = db.transaction("users").objectStore("users"),
    index = store.index("username"),
    request = index.getKey("007");

request.onsuccess = function(event){
    //处理成功
    //event.result.key 中保存索引键，而 event.result.value 中保存主键
};
```

在这个例子的 onsuccess 事件处理程序中，event.result.value 中保存的是用户 ID。

任何时候，通过 IDBIndex 对象的下列属性都可以取得有关索引的相关信息。

- name：索引的名字。
- keyPath：传入 createIndex()中的属性路径。
- objectStore：索引的对象存储空间。
- unique：表示索引键是否唯一的布尔值。

另外，通过对象存储对象的 indexName 属性可以访问到为该空间建立的所有索引。通过以下代码就可以知道根据存储的对象建立了哪些索引。

```
var store = db.transaction("users").objectStore("users"),
    indexNames = store.indexNames,
    index,
    i = 0,
    len = indexNames.length;

while(i < len){
    index = store.index(indexNames[i++]);
    console.log("Index name: " + index.name + ", KeyPath: " + index.keyPath +
        ", Unique: " + index.unique);
}
```

以上代码遍历了每个索引，在控制台中输出了它们的信息。

在对象存储空间上调用 deleteIndex()方法并传入索引的名字可以删除索引。

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

因为删除索引不会影响对象存储空间中的数据，所以这个操作没有任何回调函数。

8. 并发问题

虽然网页中的 IndexedDB 提供的是异步 API，但仍然存在并发操作的问题。如果浏览器的两个不同的标签页打开了同一个页面，那么一个页面试图更新另一个页面尚未准备就绪的数据库的问题就有可能发生。把数据库设置为新版本有可能导致这个问题。因此，只有当浏览器中仅有一个标签页使用数据库的情况下，调用 setVersion()才能完成操作。

刚打开数据库时，要记着指定 onversionchange 事件处理程序。当同一个来源的另一个标签页调用 setVersion()时，就会执行这个回调函数。处理这个事件的最佳方式是立即关闭数据库，从而保证版本更新顺利完成。例如：

```
var request, database;

request = indexedDB.open("admin");
request.onsuccess = function(event){
    database = event.target.result;

    database.onversionchange = function(){
        database.close();
    };
};
```

每次成功打开数据库，都应该指定 onversionchange 事件处理程序。

调用 setVersion()时，指定请求的 onblocked 事件处理程序也很重要。在你想要更新数据库的版本但另一个标签页已经打开数据库的情况下，就会触发这个事件处理程序。此时，最好先通知用户关闭其他标签页，然后再重新调用 setVersion()。例如：

```
var request = database.setVersion("2.0");
request.onblocked = function(){
    alert("Please close all other tabs and try again.");
};

request.onsuccess = function(){
    //处理成功，继续
};
```

请记住，其他标签页中的 onversionchange 事件处理程序也会执行。

通过指定这些事件处理程序，就能确保你的 Web 应用妥善地处理好 IndexedDB 的并发问题。

9. 限制

对 IndexedDB 的限制很多都与对 Web Storage 的类似。首先，IndexedDB 数据库只能由同源（相同协议、域名和端口）页面操作，因此不能跨域共享信息。换句话说，www.wrox.com 与 p2p.wrox.com 的数据库是完全独立的。

其次，每个来源的数据库占用的磁盘空间也有限制。Firefox 4+目前的上限是每个源 50MB，而 Chrome 的限制是 5MB。移动设备上的 Firefox 最多允许保存 5MB，如果超过了这个配额，将会请求用户的许可。

Firefox 还有另外一个限制，即不允许本地文件访问 IndexedDB。Chrome 没有这个限制。如果你在本地运行本书的示例，请使用 Chrome。

23.4　小结

*离线 Web 应用和客户端存储数据的能力对未来的 Web 应用越来越重要。浏览器已经能够检测到用户是否离线，并触发 JavaScript 事件以便应用做出处理。*可以指定在应用缓存中保存哪些文件以便离线时使用。对于应用缓存的状态及变化，也有相应的 JavaScript API 可以调用检测。

本书还讨论了客户端存储的以下几方面内容。

- 以前，这种存储只能使用 cookie 完成，cookie 是一小块可以客户端设置也可以在服务器端设置的信息，每次发起请求时都会传送它。
- 在 JavaScript 中通过 document.cookie 可以访问 cookie。
- cookie 的限制使其可以存储少量数据，然而对于大量数据效率很低。

IE 发明了一种叫做用户数据的行为，可以应用到页面的某个元素上，它有以下特点。

- 一旦应用后，该元素便可以从一个命名数据空间中载入数据，然后可以通过 `getAttribute()`、`setAttribute()`和 `removeAttribute()`方法访问。
- 数据必须明确使用 `save()`方法保存到命名数据空间中，以便能在会话之间持久化数据。

Web Storage 定义了两种用于存储数据的对象：`sessionStorage` 和 `localStorage`。前者严格用于在一个浏览器会话中存储数据，因为数据在浏览器关闭后会立即删除；后者用于跨会话持久化数据并遵循跨域安全策略。

IndexedDB 是一种类似 SQL 数据库的结构化数据存储机制。但它的数据不是保存在表中，而是保存在对象存储空间中。创建对象存储空间时，需要定义一个键，然后就可以添加数据。可以使用游标在对象存储空间中查询特定的对象。而索引则是为了提高查询速度而基于特定的属性创建的。

有了以上这些选择，就可以在客户端机器上使用 JavaScript 存储大量数据了。但你必须小心，不要在客户端存储敏感数据，因为数据缓存不会加密。