**离线应用与客户端存储**

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。

**离线检测**

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。实际应用中，navigator.onLine 在不同浏览器间还有些小的差异。

q 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。

q 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。

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

}

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。

EventUtil.addHandler(window, "online", function(){

alert("Online");

});

EventUtil.addHandler(window, "offline", function(){

alert("Offline");

});

**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。下面是一个简单的描述文件示例。

CACHE MANIFEST

#Comment

file.js

file.css

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">

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。

q 0：无缓存，即没有与页面相关的应用缓存。

q 1：闲置，即应用缓存未得到更新。

q 2：检查中，即正在下载描述文件并检查更新。

q 3：下载中，即应用缓存正在下载描述文件中指定的资源。

q 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。

q 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。

q checking：在浏览器为应用缓存查找更新时触发。

q error：在检查更新或下载资源期间发生错误时触发。

q noupdate：在检查描述文件发现文件无变化时触发。

q downloading：在开始下载应用缓存资源时触发。

q progress：在文件下载应用缓存的过程中持续不断地触发。

q 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。

q cached：在应用缓存完整可用时触发。

**数据存储**

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。名称和值在传送时都必须是URL 编码的。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

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。

**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。

每个域的 cookie 总数是有限的，不过浏览器之间各有不同。

q IE6 以及更低版本限制每个域名最多 20 个 cookie。

q 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。

q Firefox 限制每个域最多 50 个 cookie。

q Opera 限制每个域最多 30 个 cookie。

q 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。**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。

**Cookie的构成**

cookie 由浏览器保存的以下几块信息构成。

q 名称：一个唯一确定 cookie 的名称。cookie 名称是不区分大小写的。然而，某些服务器会按照区分大小写来处理 cookie。cookie 的名称**必须是经过 URL 编码的。**

q 值：储存在 cookie 中的字符串值。值必须被 URL 编码。

q 域：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，那么作用域会被认作来自设置 cookie 的那个域。

q 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。

q 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。

q 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，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

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。

**Js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。

document.cookie会取得当前域所以cookie，

如下设置一个cookie

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。Demo1