# 第十章 Web Workers API

# Web Worker

## 问题

JavaScript是单线程的，因此某个计算持续时间较长，就会出现网页假死现象。

## 解决

HTML5的Web Workers可以让Web应用程序具备后台处理能力，具有很好地多线程支持能力。

**缺点：**无法访问Web页面的DOM对象。

# 什么是Web Worker

Web Worker 是运行在后台的 JavaScript，独立于其他脚本，不会影响页面的性能。您可以继续做任何愿意做的事情：点击、选取内容等等，而此时 Web Worker 在后台运行。

# 使用Web Workers API

## 1、创建Web Worker

创建一个Web Workers对象

```
worker = new Worker("echoWorker.js");
```

传入需要执行的JavaScript文件

# 使用Web Workers API

- 内联Worker

type通知浏览器及其JavaScript引擎不要解析和运行此脚本

```
<script id="myWorker" type="javascript/worker"></script>
```

- 内联Worker：能够为同源的多个页面所共享

```
shareWorker = new ShareWorker(shareEchoWorker.js);
```

# 使用Web Workers API

- 多个JavaScript文件的加载与执行

```
importScript("helper.js");
importScript("helper.js", "anotherHelper.js");
```

# 使用Web Workers API

2、向脚本发送消息

```
worker.postMessage("hello Worker");
```

3、主进程监听postMessage事件

```
worker.addEventListener("message", handleFun);
```

4、副进程监听message事件

```
addEventListener("message", handleMsgFun, true);
```

# 使用Web Workers API

## 5、主进程监听worker错误

```
worker.addEventListener("error", handleErrorFun);
```

- 错误事件有以下三个用户关心的字段：
  - **message** 可读性良好的错误消息。
  - **filename** 发生错误的脚本文件名。
  - **lineno** 发生错误时所在脚本文件的行号。

# 使用Web Workers API

- 停止Web Worker

```
worker.terminate();
```

# 第十一章 Web Storage API

# Web Storage概述

- Web storage(也称DOMStorage)，用于在Web请求之间持久化数据。
- Storage API 允许网站的代码、Web 应用程序知道它们可以使用、已经使用多少存储空间。空间不足时，用户代理会自动清理站点数据，以便为其他用途腾出空间。
- Storage API 甚至可以控制：在执行清理之前，是否需要提醒代码或 Web 应用程序，以便作出反应。

# Web Storage概述

## Web Storage API出现前

- 远程Web服务器需要存储客户端和服务器交互使用的所有数据。

## Web Storage API出现后

- 开发者可以将需要跨请求重复访问的数据直接存储在客户端的浏览器中，还可以关闭浏览器很久后再次打开时恢复数据，以减小网络流量。

# Web Storage与cookie的区别

| | Web Storage | cookie |
|---|---|---|
| 存储空间 | 能够提供5M的存储空间（不同的浏览器不同）；<br>每个域（包括子域）都有独立的存储空间，各个存储空间是完全独立的，不会造成数据混淆 | 提供4K的空间 |
| 与服务器端的交互 | 数据存储仅仅是本地存储，不会和服务器发生任何交互 | 内容会随着请求一并发送到服务器 |
| 接口 | 提供了许多丰富易用的接口，拥有setItem、removeItem、getItem、clear、key等方法，数据操作更简单 | 需要自己封装setCookie，getCookie |

# 设置和获取数据

- 设置数据

```
/*方法1*/sessionStorage.setItem("myFirstKey", "myFirstValue");
/*方法2*/sessionStorage.myFirstKey = "myFirstValue";
/*方法3*/sessionStorage["myFirstKey"] = "myFirstValue";
```

- 获取数据

```
/*方法1*/sessionStorage.getItem("myFirstKey");
/*方法2*/sessionStorage.myFirstKey;
```

# 封堵数据泄露

对于设置到sessionStorage中的对象，只要浏览器窗口（或标签）不关闭他们就会一直存在。当用户关闭窗口或浏览器，sessionStorage数据将被清除。

# localStorage与sessionStorage的区别

WebStorage提供两种类型的API： localStorage和sessionStorage

| | sessionStorage | localStorage |
|---|---|---|
| 区别 | 数据会保存到存储它的窗口或标签页关闭时数据只会构建它们在窗口或者标签页内可见<br>（浏览器刷新可以存储数据，浏览器关闭时不可以） | 数据的生命期比窗口或浏览器的生命期长，<br>数据可被同源的每个窗口或者标签页共享 |

# Web Storage API的其他特性和函数

```
interface Storage{
    // length特性表示目前Storage对象中存储的键值对的数量
    readonly attribute unsigned long length;
    // key(index)方法允许获取一个制定位置的键。
    getter DOMString key(in unsigned long index);
    // getItem(key)函数是根据给定的键返回相应数据的一种方式。
    getter any getIten(in DOMString key);
    // setItem(key,value)
    将数据存入制定键对应的位置，如果值已存在，则替换原值。
    setter creator void setItem(in DOMString key,in any data);
    // removeItem(key)删除数据项
    deleter void removeItem(in DOMString key);
    // clear()删除存储列表中的所有数据。
    void clear();
}
```

# 第十二章 构建离线Web应用

# HTML5离线Web应用概述

HTML5新增了离线应用，离线应用使得我们可以在网页或应用在没有网络的情况下依然可以使用。

◆离线应用的适用场景

- 阅读和撰写电子邮件
- 编辑文档
- 编辑和显示演示文档
- 创建待办事宜列表

# HTML5离线Web应用概述

## 离线Web应用的运行机制

- 每个需要离线使用的网页都指定一个后缀名为 .manifest 的文本文件。
- 这个文本文件罗列了该网页离线使用时所需的所有资源文件（HTML、图片JavaScript等等）。
- 支持离线 Web应用的浏览器会自动读取 .manifest 文件，下载文件中所罗列的资源文件，并将其缓存在本地以备网络断开时使用。

# HTML5离线Web应用概述

**离线应用的使用需要以下几个步骤：**

- 离线检测（确定是否联网）
- 访问一定的资源
- 有一块本地空间用于保存数据（无论是否上网都不妨碍读写）

# 检查浏览器的支持情况

```
if(window.applicationCache){
    console.log("您的浏览器支持离线Web应用！");
} else {
    console.log("您的浏览器不支持离线Web应用！");
}
```

# 搭建简单的离线应用程序

```
<!DOCTYPE HTML>
<html manifest="demo.appcache">
<head>
    <meta charset="UTF-8">
    <title>Document</title>
</head>
<body>
    ......
</body>
</html>
```

Index.html

```
CACHE MANIFEST
# 2012-02-21 v1.0.0
/theme.css
/logo.gif
/main.js

NETWORK:
login.asp

FALLBACK:
/html5/ /404.html
```

Demo.appcache

# 支持离线行为

- HTML5引入了新的事件用来检测网络是否正常

```
function loadDemo(){
    if(navigator.onLine){
        console.log("在线！");
    } else {
        console.log("离线");
    }
}

window.addEventListener("online", funct
    console.log("online");
}, true);
```

页面加载的时候，输出状态为online或者offline

添加事件监听器，在线状态发生变化时，输出相应信息

# manifest文件

```
1   CACHE MANIFEST
2   # 2012-02-21 v1.0.0
3   /theme.css
4   /logo.gif
5   /main.js
6
7   NETWORK:
8   login.asp
9
10  FALLBACK:
11  /html5/ /404.html
```

#表示注释

要缓存的文件

不缓存的文件

FALLBACK 提供了获取不到缓存资源时的备选资源路径，本例用 “404.html” 替代 /html5/ 目录中的所有文件

Demo.appcache

# applicationCache API

- applicationCache API是一个操作应用缓存的接口。新的window.applicationCache对象可以触发一系列与缓存状态相关的事件。
- 这个对象有一个status属性，值为常量，表示缓存状态。

# applicationCache API

- status属性，值为常量，表示缓存状态

| 数值型属性 | 缓存状态 | 描述 |
|---|---|---|
| **0** | UNCACHED | 没有与页面相关的应用缓存（未缓存） |
| **1** | IDLE | 应用缓存未得到更新（空闲） |
| **2** | CHECKING | 正在下载描述文件并检查更新（检查中） |
| **3** | DOWNLOADING | 应用缓存正在下载描述文件中指定的资源（下载中） |
| **4** | UPDATEREADY | 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（更新就绪） |
| **5** | OBSOLETE | 应用缓存的描述文件不存在了，页面无法再访问应用缓存（已过期） |

# applicationCache API

- applicationCache对象有以下事件，表示其状态的改变

| 事件 | 缓存状态 |
| --- | --- |
| onchecking | CHECKING |
| ondownloading | DOWNLOADING |
| onupdateready | UPDATEREADY |
| onobsolete | OBSOLETE |
| oncached | IDLE |
| Onerror/onnoupdate/onprogress | |

# applicationCache API

| 事件 | 缓存状态 |
| --- | --- |
| checking | 当user agent检查更新时，或第一次下载manifest清单时，它往往是第一个被触发的事件 |
| downloading | 第一次下载或更新manifest清单文件时，触发该事件 |
| updateready | 此事件的含义表示缓存清单文件已经下载完毕，可以通过重新加载页面读取缓存文件或者通过方法swapCache()切换到新的缓存文件。常用于本地缓存更新版本后的提示 |
| obsolete | 访问manifest文件返回http404错误（页面未找到）或410错误（永久消失）时，触发该事件 |
| cached | 当manifest清单文件下载完毕及成功缓存后，触发该事件 |
| noupdate | 当检查到manifest文件不需要更新时，触发该事件 |
| progress | 与downloading类似，但是downloading只触发一次，progress则在清单文件下载过程中周期性触发 |

# applicationCache API

| 事件 | 缓存状态 |
| --- | --- |
| error | 如果要达到触发该事件，需要满足以下几种情况之一：<br>● 已经触发obsolete事件<br>● Manifest文件没有改变，但缓存文件中存在文件下载失败<br>● 获取manifest资源文件时发生致命错误<br>● 当更新本地缓存时，manifest文件再次被更改 |

# applicationCache API

- applicationCache有以下事件，表示其状态的改变
  - 如果应用程序已经缓存，并且清单文件没有改动，则浏览器触发noupdate事件
  - 每次载入一个设置了manifest属性的HTML文件，首先会触发checking事件
  - 如果应用程序已经缓存，并且清单文件发生改动，则浏览器触发downloading事件，下载完毕后触发updateready事件

# applicationCache API

- applicationCache有以下事件，表示其状态的改变
  - 如果应用程序未缓存，则downloading事件和progress事件都会触发，但是下载完成后触发cached事件而不是updateready事件
  - 如果处于离线，无法检测清单状态，则触发error事件，如果引用一个不存在的清单文件，也会触发error事件
  - 如果处于在线，应用也缓存了，但是清单文件不存在，则会触发obsolete事件，并将应用程序从缓存中清除。

# applicationCache API

```
applicationCache.onupdateready = function(){
    applicationCache.swapCache();
};

window.applicationCache.onupdateready = function(){
    var con = comfirm('有新内容可用，是否重新加载？');
    if(con){
        location.reload();
    }
}
```

# 运行中的应用缓存

```
CACHE MANIFEST
# 要缓存的文件
about.html
html5.css
index.html
happy-trails-rc.gif
lake-tahoe.JPG

#不缓存登录页面
NETWORK:
signup.html

FALLBACK:
signup.html   offline.html
/app/ajax/    default.html
```

用左边manifest文件

追踪示例场景